Table of Contents

CHAPTER 1 – CODES, DIGITAL DEVICES, AND MEMORIES

In order to understand the architecture of a microprocessor and to design microprocessor-based systems, it is important that the key concepts of basic digital hardware devices and design are fresh in your mind. Therefore, after a refresher on computer number systems and codes, the majority of this chapter is a review of digital hardware devices and design methods at the level needed to understand the operation of a microprocessor, to design digital interface circuitry for a microcomputer system, and to follow the discussions throughout this book. It is not intended to replace a full, "stand-alone" digital systems design course. If the material in this chapter is not clear to you after careful study, you may want to study a basic digital design text to gain a more thorough understanding of the concepts and techniques as needed.

In the last major section of the chapter, we discuss semiconductor memory devices currently used in microprocessor systems. The discussion of these memory devices is intended to review the basic memory types and characteristics or to serve as an introduction, if you have not studied these before. In Chapter 6 we discuss these devices further and show you how to design the memory section of a microprocessor system.

Objectives

At the conclusion of this chapter, you should be able to:

1. Convert decimal numbers to binary, and binary numbers to decimal.
2. Express positive and negative numbers in 2's complement sign-and-magnitude form.
3. Show the hexadecimal representation for a given binary number.
4. Describe the characteristics of standard ASCII and Unicode alphanumeric codes.
5. Express a combinational logic problem in several different forms including a truth table, Boolean equation, logic symbol(s), and an if-then-else statement.
6. Describe the operations and uses of multiplexers and demultiplexers, decoders and encoders.
7. Describe the operation of D latches and D flip-flops, use timing diagrams to illustrate the differences in their operation, and identify applications where the use of each is appropriate.
8. Draw a block diagram for a state machine, describe the operation of a variety of state machines, design a state machine for a specific function, calculate the maximum clock frequency for a given state machine circuit, and describe the design process for implementing combinational logic or a state machine in a PLD or ASIC.
9. Describe the basic characteristics and operation of current semiconductor memory devices, including MROM, PROM, EEPROM, Flash EEPROM, Synchronous Flash EEPROM, SRAM, SSRAM, DRAM, FPM DRAM, SDRAM, and DDR SDRAM.
10. Use a manufacturer's data sheet to determine specific timing parameters for a given memory device.

Refresher on number systems and codes

THE BINARY NUMBER SYSTEM

$2^0 = 1$	$2^8 = 256$	$2^{16} = 65{,}536$	$2^{24} = 16{,}777{,}216$
$2^1 = 2$	$2^9 = 512$	$2^{17} = 131{,}072$	$2^{25} = 33{,}554{,}432$
$2^2 = 4$	$2^{10} = 1{,}024$	$2^{18} = 262{,}144$	$2^{26} = 67{,}108{,}864$
$2^3 = 8$	$2^{11} = 2{,}048$	$2^{19} = 524{,}288$	$2^{27} = 134{,}217{,}728$
$2^4 = 16$	$2^{12} = 4{,}096$	$2^{20} = 1{,}048{,}576$	$2^{28} = 268{,}435{,}456$
$2^5 = 32$	$2^{13} = 8{,}192$	$2^{21} = 2{,}097{,}152$	$2^{29} = 536{,}870{,}912$
$2^6 = 64$	$2^{14} = 16{,}384$	$2^{22} = 4{,}194{,}304$	$2^{30} = 1{,}073{,}741{,}824$
$2^7 = 128$	$2^{15} = 32{,}768$	$2^{23} = 8{,}388{,}608$	$2^{31} = 2{,}147{,}483{,}648$

(a)

Binary Number	1	0	1	0	1	1	0	.	0	1	
Power of 2	2^6	2^5	2^4	2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	
Decimal value	64	32	16	8	4	2	1		0.5	0.25	
Convert to Decimal (sum)	64	0	16	0	4	2	0		0	0.25	= 86.25

(b)

Figure 1-1. Binary number system. (a) Decimal values for powers of 2. (b) Decimal values for binary digits.

Each digit in a *binary* or base-2 number represents a power of 2. Figure 1-1a shows the decimal equivalents for the first 32 powers of 2, and Figure 1-1b shows the decimal equivalents for the lowest digits of the binary number system. Since the number of possible values for a digit in a binary or base-2 number system is two, the value of a single binary digit can only be a 0 or a 1. Each digit of a binary number is referred to as a *bit*. The rightmost bit is referred to as the Least Significant Bit or LSB and the leftmost bit is referred to as the Most Significant Bit or MSB. For the three least significant bits, you count up in binary as follows: 000, 001, 010, 011, 100, 101, 110, 111, etc. A four bit binary count is shown in Table 1-1. The number of different values that you can represent with an N digit binary number is 2^N. An 8-bit binary number, for example, can represent 2^8 values, 00000000-11111111, or 0-255 decimal.

Binary numbers with certain quantities of bits have been given special names. A 4-bit binary number is sometimes called a *nibble* and an 8-bit binary number is commonly referred to as a *byte*. In some systems, a 16-bit binary number is referred to as a *word* and a 32-bit number is referred to as a *double-word*. In other systems, a 16-bit quantity is referred to as a *half-word* and a 32-bit quantity is referred to as a *word*. The processors that are used as examples in this book all follow the 16-bit half word, 32-bit word naming convention.

To convert a binary number to its decimal equivalent then, just multiply each of the 1 bits in the number by its decimal value and sum the results. For instance, the binary number 101 represents $(1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$, or decimal 5. To convert a decimal number to its binary equivalent, either use your calculator or proceed on paper as follows. Find the largest power of two that is less than the decimal number (using Figure 1-1a as a guide) and put a 1 in this bit position in the binary number. Subtract the value of this power of 2 from the original decimal number. Then, find the next largest power of two that is less than the remainder, put a 1 in this bit position, and subtract the decimal value from the remainder left from the determination of the

previous bit. Repeat the process until the result of the subtraction is 0 or 1. This 0 or 1 will be the value of the LSB. For example, to convert the decimal number 21 to binary, you can see from Figure 1-1a that the largest power 2 that will fit in 21 is 2^4 or 16 decimal, so put a 1 in the 2^4 bit position and subtract 16 from 21 to give a remainder of 5. Since 2^3 will not fit in the remainder of 5, put a 0 in the 2^3 bit position. However, 2^2 or 4 will fit in the remainder of 5, so put a 1 in the 2^2 bit position and subtract 4 from the 5. The remainder of 1 is equal to the least significant binary bit, so put a 1 in the least significant bit position. The result then is that decimal 21 is 10101 in binary. Note that bits to the right of the binary point in Figure 1-b represent fractional values and are used to represent values less than 1. If you want to convert a decimal number that has a fractional part to its binary equivalent, you just continue the binary to decimal conversion process we have just described until you have a binary number with the desired degree of precision.

2'S COMPLEMENT SIGN-AND-MAGNITUDE BINARY CODE

When you write a number by hand, you simply put a "+" sign in front of the number to indicate that it is positive or a "–" sign to indicate that it is negative. However, when you want to store signed numbers in a computer, you need to have some other way to identify numbers as positive or negative because a computer can only store 0's and 1's and not the signs. A common way to represent signed numbers is to reserve the most significant bit of a binary data word as a *sign bit* and use the remainder of the bits to represent the size (magnitude) of the number. The usual convention is to represent a positive number with a 0 sign bit and a negative number with a 1 sign bit. To make computations with signed numbers easier, the magnitude of negative numbers is represented in a special form called *2's complement*. The 2's complement of a positive binary number is just the standard binary representation of the magnitude of the number with a 0 in the MSB as the sign bit. The 2's complement representation for a negative binary number is formed by inverting each bit of the positive binary number, including the 0 sign bit, and adding 1 to the result. Some examples should help clarify this in your mind. For simplicity we will use 8-bit binary words for the examples here, but the principles are the same for binary numbers with any number of bits.

The decimal number +7 is represented in 8-bit sign and magnitude form as 00000111. Since the number is positive, the sign bit is 0 and the 7 bits for the magnitude are just the straight binary value for 7. To represent –7 decimal in 2's complement sign and magnitude form, start with the 8-bit code for +7, or 00000111. Invert each bit, including the sign bit, to get 11111000. Finally, add 1 to get 11111001, which is the correct 2's complement representation for –7 decimal. Figure 1-2 shows more examples of positive and negative numbers expressed in 2's complement sign and magnitude form.

	Sign bit	Magnitude
+7	0	000 0111
+46	0	010 1110
+105	0	110 1001
-12	1	111 0100
-54	1	100 1010
-117	1	000 1011

Figure 1-2. Examples of numbers represented in 8-bit 2's complement sign-and-magnitude form.

To find the magnitude of a number expressed in 2's complement sign and magnitude form, proceed as follows. If the number is positive as indicated by a 0 sign bit, then the least significant bits of the word represent the magnitude directly. If the number is negative as indicated by a 1 sign bit, then the magnitude is expressed in 2's complement form. To determine the absolute value of this magnitude, invert each bit of the word, including the sign bit, and add 1 to the result. For example, given the word 11101011, invert each bit to get 00010100 and then add 1 to get 00010101. This binary value is equal to 21 decimal, so you know the original 2's complement number represents –21 decimal

Next let's take a look at the range of numbers that can be represented by 8-bit 2's complement sign and magnitude form. Eight bits can represent a maximum of 2^8 or 256 numbers. The sign and magnitude form is being used to represent both positive and negative numbers, so half of this range will be positive and half will be negative. Since 0 counts as a positive number, the range of 128 positive values that can be represented is 0 to +127. The range of negative values that can be represented is -1 to – 128. The representations of the boundary values for 8-bit signed numbers are as follows:

```
01111111  =  +127
          :
00000001  =  +1
00000000  =  0
11111111  =  –1
          :
10000000  =  –128
```

If you like patterns you might notice that, in this scheme, the values used for +128 to + 255 in standard binary are moved downward below zero to represent –128 to –1 in 2's complement. To extend the concept, note that in a system that uses 16-bit signed numbers, the range of signed numbers that can be represented is –32,768 to + 32,767. We leave it to you to think about the range of numbers possible with a 32-bit signed number. You can use Figure 1-1a to help with this.

Figure 1-3 shows some examples of how you add 8-bit signed numbers. Sign bits are added together just as other bits are. Figure 1-3a shows the result of adding two positive numbers. Since the sign bit of the result is positive, the result is positive and the magnitude is in true binary form.

The example in Figure 1-3b shows the result of adding a negative number to a positive number or, in other words, subtracting 9 from +13.

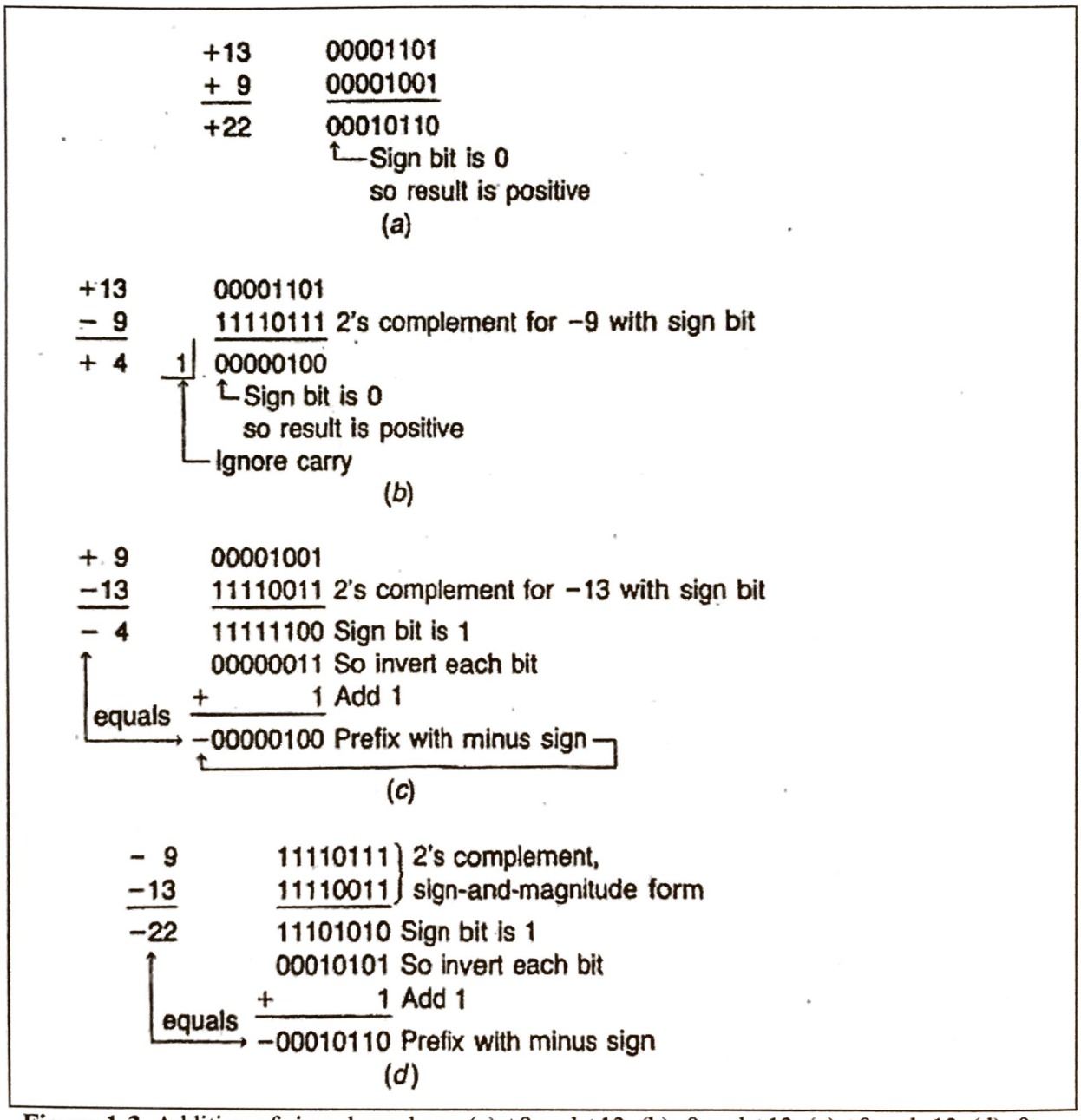


Figure 1-3. Addition of signed numbers. (a) +9 and +13. (b) -9 and +13. (c) +9 and -13. (d) -9 and -13.

The sign bit for the result is a 0, so we know the result is positive and in true binary form. Figure 1-3c shows the result of adding –13 to +9 or, in other words subtracting 13 from 9. The sign bit for the result is a 1, indicating the result is negative and the magnitude is in 2's complement form. To determine the absolute magnitude, just invert the entire word and add 1 as we described earlier. The next example in Figure 1-3d shows the result of adding two negative

numbers. Again, the sign bit of the result is a 1, so the result is negative and in 2's complement form.

An important point to keep in mind when you are doing signed arithmetic by hand or with a computer is that, if the result is too large to fit in the bits reserved for the magnitude in a word, the result will *overflow* into the bit reserved for the sign. The result will then be incorrect. For example, if the signed positive number 01001001 is added to the signed positive number 01101101, the result is 10110110. The 1 in the MSB of this result indicates that it is negative, which is obviously incorrect for the sum of two positive numbers. Fortunately, most microprocessors have built-in hardware to detect and notify you if an overflow occurs during a calculation.

THE HEXADECIMAL NUMBER SYSTEM

Computers can only work directly with binary values, but people have trouble remembering and accurately working with, for example, the long string of 1s and 0s in a 32-bit word. One solution for this problem is to use the *hexadecimal* or base-16 number system to represent the value of the long binary words.

16^0	=	1
16^1	=	16
16^2	=	256
16^3	=	4,096
16^4	=	65,536
16^5	=	1,048,576
16^6	=	16,777,216
16^7	=	268,435,456

(a)

Dec	Hex	Dec	Hex
0	0	8	8
1	1	9	9
2	2	10	A
3	3	11	B
4	4	12	C
5	5	13	D
6	6	14	E
7	7	15	F

(b)

Figure 1-4. Hexadecimal (base-16) number system. (a) Decimal values for powers of 16. (b) Symbols used to represent values in a hexadecimal digit.

In a base-16 number system, each digit represents a power of 16. Figure 1-4a shows the decimal equivalents for the first 8 powers of 16, and Figure 1-4b shows the symbols used to represent hexadecimal digits. The number of different symbols required to represent the digit values in a particular base number system is equal to the base number, so the base-16 or hexadecimal number system requires 16 symbols. As shown in Figure 1-4b, we use the numbers 0-9 for the first ten values and the letters A-F for the last 6 values. You count up in hexadecimal as follows: 00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 0A, 0B, 0C, 0D, 0E, 0F, 10, 11, 12, etc. The key point here is that, since one hexadecimal digit can represent the same number of values as 4 binary digits, you can use one hexadecimal number to represent each group of 4 bits in a binary word, starting from the LSB. The 8-bit binary number 0011 0101, for example, can simply be represented with its hexadecimal equivalent of 35 hexadecimal. Likewise, the binary number 1010 1001 can be represented with its hexadecimal equivalent of A9. The two-digit hexadecimal representations are much easier to remember and accurately communicate than the 8-bit binary numbers. In this book we use a lot of 32-bit binary quantities and in most cases we will use the hexadecimal representations rather than show the binary values directly. To indicate that a given

number represents a hexadecimal or "hex" value, we either put an upper case H after the number, as for example, A9H, or a 0x in front of the number, as for example, 0xA9. In this book we will for the most part use the latter notation, but both are used in manufacturer's data books so you should recognize either of them

To convert a hexadecimal number to its binary equivalent, simply write the 4-bit binary value for each hex digit, starting from the least significant digit at the right end of the hex number. As an example, suppose that the manual for an instrument says that the value on a signal bus at a particular time should be 0xD6. The 4-bit binary equivalent of hexadecimal D is 1101 and the 4-bit binary equivalent of hex 6 is 0110, so you know that the binary levels on the eight signal lines are 11010110. Note that you always need to use 4 binary values for each hexadecimal digit, even if 4 bits are not required to represent the value. For values less than 8, you add 0's in the MSB positions to fill the binary value to 4 bits. In the preceding example, for instance, only 3 bits are required to represent 6 but, since the 6 represents a hexadecimal digit, we wrote 0110 as the binary representation.

Decimal	Binary	Hex.	BCD Code	Gray Code	7-Segment Display (1=on)	
					abc defg	Display
0	0000	0	0000	0000	111 1110	0
1	0001	1	0001	0001	011 0000	1
2	0010	2	0010	0011	110 1101	2
3	0011	3	0011	0010	111 1001	3
4	0100	4	0100	0110	011 0011	4
5	0101	5	0101	0111	101 1011	5
6	0110	6	0110	0101	101 1111	6
7	0111	7	0111	0100	111 0000	7
8	1000	8	1000	1100	111 1111	8
9	1001	9	1001	1101	111 0011	9
10	1010	A	0001 0000	1111	111 1101	A
11	1011	B	0001 0001	1110	001 1111	B
12	1100	C	0001 0010	1010	000 1101	C
13	1101	D	0001 0011	1011	011 1101	D
14	1110	E	0001 0100	1001	110 1111	E
15	1111	F	0001 0101	1000	100 0111	F

Table 1-1. Common number codes: binary, hexadecimal, BCD code, Gray code, 7-Segment Display code for common-cathode displays.

BCD CODE

In applications such as frequency counters, digital voltmeters, or electronic scales where the output is a decimal display, a *binary-coded-decimal* or BCD code is often used to represent decimal digits. BCD uses a 4-bit binary code to individually represent each decimal digit in a decimal number. The most common BCD code uses the 4-bit binary values 0000 - 1001 to represent the decimal numbers 0-9. Table 1-1 shows the BCD codes for the first 15 decimal numbers. To convert a decimal number to its BCD equivalent, you just represent each decimal digit with its 4-bit binary equivalent. The decimal number 529, for example, is represented in BCD as 0101 0010 1001.

To convert a BCD number to its decimal equivalent, just divide the BCD number into groups of 4, starting from the least significant bit, and write the decimal equivalent for each group of 4 bits. Note that each BCD digit value must include all four bits, even if four bits are not required, to represent a particular decimal digit. Also note that the binary values 1010, 1011, 1100, 1101, 1110, and 1111 are not valid BCD values because they do not represent single decimal digits. The correct BCD representation of the decimal number 12, for example, is 0001 0010, not the binary value of 1100.

GRAY CODE

Gray code is another important binary code that is often used for encoding the position of a rotating shaft in a machine such as a computer controlled milling machine. For a given number of bits Gray code has the same number of possible combinations as does standard binary, but the codes are ordered so that only one bit changes at a time as you count up from 0. For example, the values (in order) for a 3-bit Gray code would be 000, 001, 011, 010, 110, 111, 101, 100. Table 1-1 shows the values for a 4-bit Gray code. In a later chapter discussion about optical encoders, you will see that the advantage of Gray coding is that, since only one bit changes at a time, the error in reading a number at a boundary is at most one bit. With standard binary, an error in reading the boundary between 000 and 111, for example, would result in an error of 111, the full-scale value.

SEVEN SEGMENT DISPLAY CODE

For digital instruments that just require a numeric display, we often use seven segment Light-Emitting-Diode (LED) displays. Figure 1-5a shows the segment identifier letters for a 7-segment display and Figure 1-5b shows how the LEDs are internally connected in a common cathode display. For this type of display, a logic high (binary 1) must be applied to light a desired segment. Figure 1-5c shows the internal connection for a common anode LED display. For this type of display, a logic low (binary 0) must be applied to light each LED.

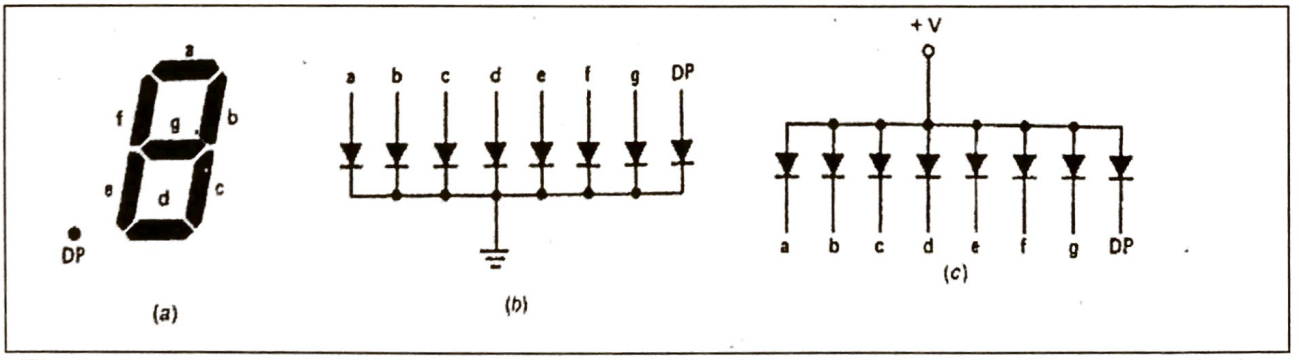


Figure 1-5. 7-Segment LED display. (a) Segment identifiers. (b) Schematic of common cathode displays. (c) Schematic of common anode displays.

In a later chapter we discuss LED interfacing in greater detail. However, to introduce you to 7-segment displays, Table 1-1 shows the codes required to display the decimal digits 0 through 9 and the letters A through F on a common cathode 7-segment LED display. These codes would simply be inverted for a common-anode display.

ASCII CODE

When communicating with or between computers, you need a binary-based code that can represent all of the letters of the alphabet as well as numbers and a variety of control signals. Codes capable of doing this are referred to as *alphanumeric codes*. Probably the most common alphanumeric code is ASCII, the American Standard Code for Information Interchange, which has been used in systems since the late 1960's. For future reference, Table 1-2 shows the hexadecimal representations for the 7-bit ASCII code. As you can see in Table 1-2, the ASCII codes for the numbers 0 through 9 are 0x30 – 0x39, the ASCII codes for upper case letters are 0x41 – 5A, and the ASCII codes for the lower case letters are 0x61 – 0x7A. The ASCII codes from 0x00 – 0x2F are reserved for punctuation and control characters. The ASCII code 0x0D, for example, is the code for a Carriage Return and the ASCII code 0x1B is the code for the Esc key. Note that the names of these control codes in the lower range of ASCII are standard, but the codes in this lower range are used for different functions in different systems. Therefore, you need to consult the manual for the system you are working with to determine just how they are used in that system.

ASCII standard codes have only 7 bits, but there are several current ASCII code extensions that that include an additional bit in the MSB position and thus give an extension of 128 additional codes for symbols and control characters. Again, different systems use different assignments for these extensions, so you need to consult each system's manual to determine the specific code assignments for that system. The Unicode described in the next section is a newer alphanumeric code that is a standard, and it is much more versatile than basic ASCII code.

ASCII Symbol	Hex Code for 7-Bit ASCII	ASCII Symbol	Hex Code for 7-Bit ASCII	ASCII Symbol	Hex Code for 7-Bit ASCII
NUL	00	*	2A	T	54
SOH	01	+	2B	U	55
STX	02	,	2C	V	56
ETX	03	-	2D	W	57
EOT	04	.	2E	X	58
ENQ	05	/	2F	Y	59
ACK	06	0	30	Z	5A
BEL	07	1	31	[	5B
BS	08	2	32	λ	5C
HT	09	3	33	]	5D
LF	0A	4	34	^	5E
VT	0B	5	35	_	5F
FF	0C	6	36	`	60
CR	0D	7	37	a	61
S0	0E	8	38	b	62
S1	0F	9	39	c	63
DLE	10	:	3A	d	64
DC1	11	;	3B	e	65
DC2	12	—	3C	f	66
DC3	13	=	3D	g	67
DC4	14	\	3E	h	68
NAK	15	?	3F	i	69
SYN	16	@	40	j	6A
ETB	17	A	41	k	6B
CAN	18	B	42	l	6C
EM	19	C	43	m	6D
SUB	1A	D	44	n	6E
ESC	1B	E	45	o	6F
FS	1C	F	46	p	70
GS	1D	G	47	q	71
RS	1E	H	48	r	72
US	1F	I	49	s	73
SP	20	J	4A	t	74
!	21	K	4B	u	75
"	22	L	4C	v	76
#	23	M	4D	w	77
$	24	N	4E	x	78
%	25	O	4F	y	79
&	26	P	50	z	7A
'	27	Q	51	{	7B
(	28	R	52	|	7C
)	29	S	53	}	7D
				~	7E
				DEL	7F

Table 1-2. ASCII symbol names and hexadecimal codes.

UNICODE

The Unicode Standard, developed by the Unicode Consortium and available on the Internet at http://www.unicode.org, provides codes for characters in almost all of the written languages of the world. Each code has an identifying name and a *code point*. The code point is simply the numerical value of the code. The current Unicode 4.0 standard provides 1,114,112 code points and since most of these are available for representing characters, the total number of possible character representations is very large. However, the characters needed for most of the major languages of the world are contained in the first 64 K codes, 0x0000 – 0xFFFF. These first 64K codes are called the *Basic Multilingual Plane* (BMP).

The Unicode Standard has three encoding forms, UTF-32, UTF-16, and UTF-8. UTF-32 uses 32bits (4 octets in Unicode terms) to represent each character and can represent all of the defined code points. UTF-32 can represent a very large number of characters but since it requires 4 octets for each character, it is not very efficient in terms of memory space or data transmission. UTF-16 is more space-efficient for cases when the full capabilities of UTF-32 are not needed. UTF-16 uses 2 octets to represent each of the codes in the Basic Multilingual Plane. The BMP contains enough code points to represent most of the characters needed for the most common languages and, using some spaces intentionally left in the BMP, UTF-16 can also represent some additional characters beyond those in the BMP. UTF-8 encoding was designed for compatibility with ASCII.Character codes less than 128 decimal (the most significant bit of the first code point octet is a 0), are treated as ASCII codes. Character codes greater than 128 are treated as Unicodes. UTF-8 encoding is used for HTML and XML documents and is supported in most C++ and Java systems. This availability makes it easier to develop software and web pages for international markets.

Combinational Logic Analysis and Design

ANALYSIS OF COMBINATIONAL LOGIC CIRCUITS

In order to understand the combinational logic diagrams in manufacturers' data sheets and in some cases the "glue" logic between the large devices in a microprocessor system, it is important for you to be able to predict the logic level at any point in a digital circuit directly from the schematic, rather than having to work your way through the truth table for each gate. The way that the schematic symbols are drawn in modern schematics makes it very easy to predict the output of a logic gate by following two simple rules:

1. The shape of the symbol indicates the operation or logic function performed by the device.
2. A bubble on an input or output indicates that the input or output has an active low assertion level. No bubble on an input or an output indicates that the input or output has an active high assertion level.

Some examples will quickly refresh your memory on how to read logic symbols in this way or teach you how to do it, if you haven't done it before.

Figure 1-6a shows the schematic symbols and truth tables for non-inverting and inverting buffers. A buffer is used to increase the drive current for a signal and/or change the logic level of

a signal. The first symbol for a buffer in Figure 1-6a has no bubbles on the input or output, so the input is active high and the output is active high. You read this symbol as follows. If the input, A, is asserted high, then the output, X, will be asserted high. The rest of the truth table tells you that if the A input is not asserted high, then the X output will not be asserted high. For the buffer symbol with a bubble on the output and no bubble on the input, you can express the operation performed with the simple statement, "If the input is asserted high, then the Y output will be asserted low – Else the output will be asserted high." In a similar manner, the operation for the buffer symbol with a bubble on the input and no bubble on the output can be expressed as, "If the input is asserted low, then the Y output will be asserted high – Else the output will be low. Note that both symbols represent the same truth table with just the If-Then and Else clauses reversed in the two different expressions for the operation performed. A little later we will build on this If-Then-Else way of expressing logic operations but next we discuss another type of buffer, which is very important in microcomputer systems.

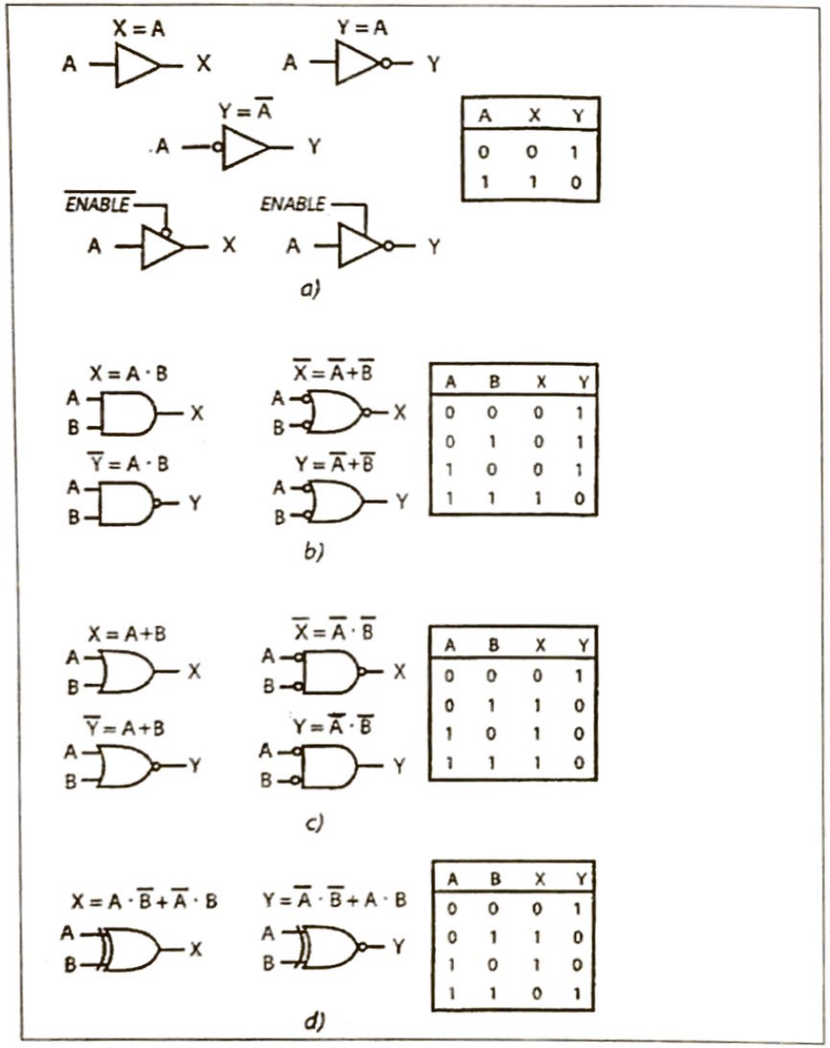
The bottom two symbols in Figure 1-6a represent buffers with three-state outputs. If the Enable input of the device on the bottom right in Figure 1-6a is asserted high, then the device output is enabled and the device will operate as a simple inverting buffer. If the Enable signal is asserted low, the buffer output will be internally disconnected from the output pin and the output will be effectively disabled. We commonly say that a disabled output is in a high-impedance or "floating" state. Because a disabled buffer is not asserting any logic level on its output pin, the logic level present on the pin and on the signal line to which the line is connected will be determined by some other device output connected to the same signal line. You need to use three-state output devices whenever you want any one of several different device outputs to be able to drive a signal line to which they are all connected. You also need to make sure that only one of the three-state outputs connected to a particular line is enabled at a time, so that there will never be a conflict between two or more outputs. Remember that, if you connect the outputs of two standard-output gates together and one output goes low while the other output is high, a short-circuit will be produced between the power supply and ground through the two outputs. This may cause the devices to get hot and possibly burn out. (In some systems we connect a pull-up resistor to signal lines that have several three-state outputs on them, so that if none of the devices are driving the line, it will have a valid logic state on the line, rather than a high impedance, indeterminate level that might easily pick up electrical noise.)

The three-state buffer on the bottom left in Figure 1-6a has a bubble on its Enable input, so the buffer output will be enabled by a low on the Enable input. There is no bubble on the output of this buffer, so the logic signal will be passed though the buffer without being inverted.

Figure 1-6b shows the logic symbols and truth tables for AND and NAND gates. Using the two rules above and the truth table for an AND gate, you should see that the function of an AND gate can be expressed as "If the A input is high AND the B input is high, the X output is high – Else the output will be low." The IF clause in this expression represents the case for the bottom line in the truth table for the AND gate and the else clause represents the other three lines in the truth table. In Boolean algebra, the AND function can be expressed by either $X = A \cdot B$, or $X = AB$.

The symbol to the right of the basic AND symbol has the OR symbol shape, so it performs the OR function. The bubbles on the inputs tell you that the inputs are active low and the bubble on the output tells you that the output is active low. Using these observations you can directly write the expression for the symbol as "If the A input is low OR the B input is low, the output will be low – Else the output will be high." The IF clause in this expression represents the top three cases in the AND gate truth table and the Else clause represents the bottom case in the truth table. The Boolean expression for this OR function can be written as $Y\# = A\# + B\#$. (The significance of the # symbol will be discussed later in this section.) Since the AND symbol and the bubbled input/output OR symbol represent the same truth table, they must be equivalent and they are referred to as the *DeMorgan equivalent symbols*. Figure 1-6b also shows the traditional Boolean algebra representations for each of the alternative symbols. The bottom two symbols in Figure 1-6b represent the NAND function. For practice, write the If-Then-Else expressions for each of these and compare your results with the truth table entries.

Figure 1-6c shows the DeMorgan equivalent symbols for an OR gate and for a NOR gate. The expression for the traditional OR symbol with active high inputs and an active high output is, "If the A input is high OR the B input is high, the output will be high – Else the output will be low." The If clause in this expression represents the bottom three entries in the truth table and the Else clause represents the top entry in the table. For the DeMorgan equivalent OR symbol the expression is, "If the A input is low AND the B input is low, the output will be low – Else the output will be high." Again, for more practice, write the If-Then-Else expressions for the NOR gate in the bottom of Figure 1-6c and compare your results with the truth table.



A	X	Y
0	0	1
1	1	0

A	B	X	Y
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

A	B	X	Y
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

A	B	X	Y
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

Figure 1-6. Buffers & logic gates. (a) Buffers. (b) AND - NAND. (c) OR - NOR. (d) Exclusive OR - Exclusive NOR.

Before we go on to show you how to analyze and implement more complex combinational logic functions, we need to review one more simple logic function. Figure 1-6d shows the symbols, Boolean expressions, and truth tables for an Exclusive OR gate and for an Exclusive NOR gate. As you can see from the truth table or the Boolean expression, the output of an Exclusive OR will be high if the A input is high OR the B input is high but not if both the A and B inputs are high. This function is called Exclusive OR (XOR) because it "excludes" the case where both inputs are high. (The function for the gate shown at the top of Figure 1-6c is sometimes called an "Inclusive OR" because it includes the case where both the A and B inputs are high, but usually we just refer to it as an OR gate.). The XOR function can be expressed in Boolean algebra by $X = A \oplus B$. Remember that AND, NAND, OR, and NOR gates can have any number of inputs but XOR and XNOR gates can have only 2 inputs on each gate. To implement an EXOR function for more than 2 variables, you simply connect the XOR gates in a multi-level "tree" structure that has the required number of inputs.

Now that we have reviewed how you predict the outputs of basic gates, we need to clarify the concepts of signal assertion levels and how the active levels for signals are represented in logic expressions and in schematic symbols. The *active* or *asserted* logic level for a particular signal is the level that causes some desired action to occur in a circuit or device. For example, the Enable input on the inverting buffer in Figure 1-6a is an active high signal because it is the high level of this signal that enables the buffer. In contrast, the non-inverting buffer in Figure 1-6a is enabled by a low on the Enable input, as indicated by the bubble on this input. To indicate in a system schematic that this signal is active low, it would be identified as Enable (with an overbar), Enable/, Enable*, nEnable, or Enable#. In most of our writing, we will use the Enable# representation because it is easy to type and read. However, you will see all of these in the different manufacturers' data sheets we use in the book, so you should immediately recognize the meaning of any one of these. An additional point is that some system signals cause one action in the high state and another action in the low state. One example of this is a common microprocessor signal labeled R/W#. The name of this signal tells you that the signal will be high if the microprocessor is doing a Read operation, and low if the microprocessor is doing a Write operation.

Some additional rules you need to remember for analyzing combinational logic circuits with mixed assertion levels and two or more levels of logic gates are:

1. Two bubbles in series on a signal line "cancel", just as the effects of two inverters in series do, and the term for the signal in the output expression is unchanged. If, for example, you connect the output of the first inverter in Figure 1-6a to the input of the second inverter in Figure 1-6a, the result on the final output is just A.
2. An active low signal connected to an active low input asserts that input. For example, the signal Enable# connected to the bubbled enable input of the non-inverting three-state buffer in Figure 1-6a enables the buffer when the Enable# signal is low.
3. If the active level for a signal is different from the active level of the input to which it is connected, that signal will be represented as the complement of the signal in the output expression. For example, an active high signal, B, connected to a bubbled input of a gate will appear in the output expression for the gate as B#. For another example, if an active high signal, A, is connected to an active low input such as one of the inputs on the bubbled-input OR shaped symbol in Figure 1-6b, the A term will be complemented in the output expression as shown above the signal.

The key point of the preceding discussion is that, once you get this simple system for analyzing gate circuits in your mind, it is very easy for you to predict the response of a logic gate you see in a circuit or to implement a simple logic function you need in a design. The circuit in Figure 1-17 and some problems at the end of the chapter will give you a little practice using these rules.

Implementing Combinational Logic Functions in Simple PLDs

The preceding section reviewed basic logic gates and showed you how to easily predict their outputs for given sets of input signals. However, most real designs require more complex combinational logic expressions than can be implemented by a single logic gate. A design might, for example, require a multi-level logic function expressed by the Sum-Of-Products (SOP) expression Y= A•C•D + A•B + B•D or a logic function expressed in Product-Of-Sums (POS) form as X = (A+C+D)(B+E). The traditional steps in developing the designs for a multi-level combinational logic circuit such as these are:

1. Draw a block diagram for the desired circuit showing all input and output signals.
2. Write a truth table that shows the desired output signal for each of the possible values of the input signals.
3. Write a Sum-of-Products term for each truth table input combination for which the output is asserted.
4. Combine the terms into a Sum-Of-Products (SOP) or Product-Of-Sums (POS) expression.
5. Use Boolean algebra to simplify the SOP or POS expression. (For logic functions with six or fewer input variables, you may convert the truth table to a Karnaugh map and derive the simplified SOP or POS expression directly from the Karnaugh map by circling adjacent 1s or adjacent 0s.)
6. Implement the simplified SOP or POS expression with logic gates from a logic family that has the desired speed/power characteristics.

Boolean algebra simplification is tedious and error prone for equations with a large number of input variables. Karnaugh map simplification is less error prone but it is only useful for logic functions with at most six input variables. Since the logic functions required in current designs often have a large number of input variables, we now use computer-based tools to generate the simplified logic expressions for our designs and to map the simplified expressions directly into single, Programmable Logic Devices (PLDs) such as Programmable Logic Arrays (PLAs), Programmable Read-Only Memories (PROMs), Programmable Array Logic devices (PALs), or Field Programmable Gate Arrays (FPGAs). Therefore, instead of discussing Boolean algebra and Karnaugh maps, we will give you a review or introduction to how you implement combinational logic functions in the simplest of these devices.

Figure 1-7 shows a simplified representation for the combinational logic section of a Programmable Array Logic (PAL) device. The basic structure is an AND-OR matrix. Each AND gate in the figure has four inputs but to simplify the drawing, only a single input line is shown. Likewise, each OR gate has two inputs but is shown with a single line for simplicity. These devices are programmed by blowing fuses or closing electronic switches on the inputs of the AND gates to produce the desired product terms for the input variables. An X in the diagram

represents an intact fuse or a closed switch that makes a connection between a buffer output and one of the inputs on an AND gate. The expressions along the right side of the figure show some of the product terms that can be generated by programming the AND matrix.

The dots in the output OR matrix in Figure 1-7 represent fixed connections made during the manufacture of the device and show the product terms that will be ORed together to produce the Sum-Of-Products (SOP) output expressions. Shown next to each OR gate in Figure 1-7 is the SOP expression produced by the specific programming of the AND matrix.

The key point here is that the AND-OR matrix architecture allows you to implement any AND-OR (SOP) function of the input variables. Since any truth table and thereby any logic function can be expressed in SOP form, a PAL can be used to implement any combinational logic function, assuming that the AND gates and OR gates in the PAL have enough inputs. The device shown in Figure 1-7 has inputs for only 2 variables but available PAL devices such as the GAL 22V10 from Lattice Semiconductor and others can have up to 22 inputs and up to 10 outputs, so they can be used to implement large logic functions in a single programmable device.

Programmable Read-Only Memories (PROMs) have the same AND-OR structure as PALs, but in PROMs the AND matrix connections are fixed and the OR matrix connections are programmable. PROMS are hard-wired to implement all of the possible product terms for the input variables, so they are useful as memories and as code converters.

Programmable Logic Array (PLA) devices have both a programmable AND matrix and a programmable OR matrix. PLAs are very flexible but they are harder to program and harder to test, so they are used less often than PALs and PROMs

To develop the files for programming any of these devices, you develop the block diagram for the design as in step 1 above, write a truth table or equations for the desired functions, and then use computer-based tools to generate the programming file. One example of a commonly available tool for implementing logic designs in PALs is the Warp6 toolset from Cypress Semiconductor. Warp6 allows you to enter the design data in the editor as Boolean equations, a truth table, a Verilog source file, or a VHDL source file. You compile the source file to optimize the logic functions and create an output file. Using the output file, you can simulate the design with the Warp6 waveform simulator to see if the output is correct for each input combination. After the design passes simulation, you then use the output file to produce the "fusemap" or JEDEC (.jed) file for the specific PAL you are using. You then download the JEDEC file to a programmer and use it to program your device.

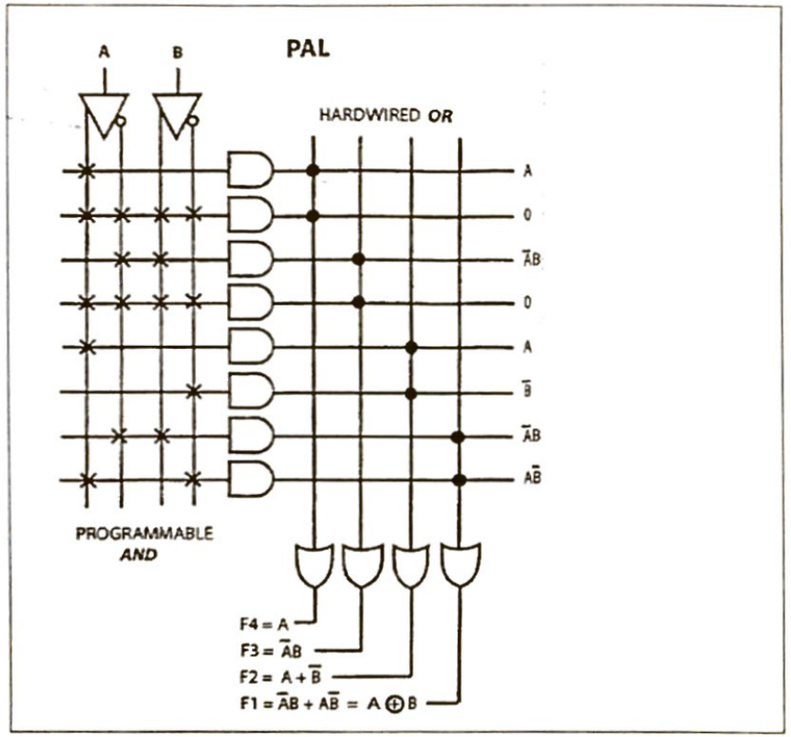


Figure 1-7. Basic architecture of a Programmable Array Logic (PAL) device, programmed to implement some simple logic functions.

In a later section we will discuss how tools such as this can be used to implement state machines in PALs and FPGAs. Before that, however, we need to discuss some common combinational logic functions that are very important in microprocessors and microcomputer systems. Although we seldom use them in current designs, we will use some older, Medium Scale Integration (MSI) devices to help explain the operation and uses of these functions.

Multiplexers and Demultiplexers

Figure 1-8 shows how a multiplexer and demultiplexer can be used to send samples of eight different signals on a single wire. The term multiplexing or, more specifically for our discussion here, time-division multiplexing, means sending different signals over a common wire or set of wires at different times. Here's how this works.

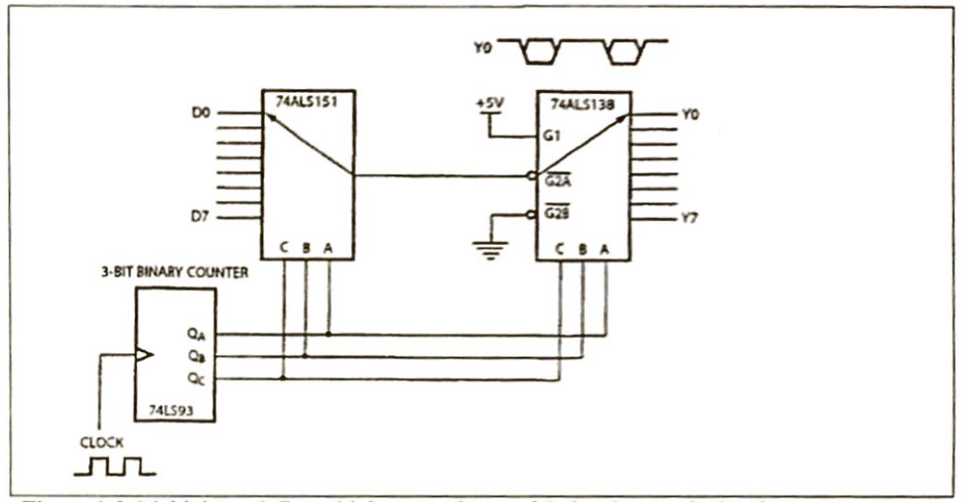


Figure 1-8. Multiplexer & Demultiplexer used to send 8 signals on a single wire.

The 74ALS151 on the left side of Figure 1-8 represents the mechanical switch equivalent of an 8-input multiplexer (MUX) or, as it is also called, an 8-input data selector. At any given time, the signal on one of the 8 data inputs, D0-D7, will be routed to the Y output. The 3-bit code applied to the select inputs (C, B, A) determines which input signal is routed to the output. If a 3-bit binary counter is connected to the select inputs as shown and the counter is clocked, the multiplexer will cycle through the eight inputs like a rotary switch. Digital samples from each of the eight inputs will be sent out the Y output and down the wire one after the other in sequence. We say then that samples of the eight signals are *time-division multiplexed* on the line, each sample in its own time slot.

As connected, the 74ALS138 at the other end of the connecting wire functions as an 8-output demultiplexer (DMUX). The mechanical switch equivalent drawing shows that, at any given time, the input of the 74ALS138 will be effectively connected to one of the 8 outputs. Again, the 3-bit code on the select inputs, C, B, and A, determines the output to which the input is connected. Since the select inputs of the demultiplexer are connected to the same counter outputs as the select inputs of the multiplexer, the two devices are "synchronized." For example, with a select code of 000, the multiplexer will route data from its D0 input on the wire and the demultiplexer will route this data to its Y0 output. Note that although we use a clocked counter to cycle the multiplexers and demultiplexers through the inputs, the multiplexers and

demultiplexers themselves are not clocked and internally each consists of just a couple layers of logic gates.

The advantage of multiplexing is that it allows a common signal line or group of signal lines to be used to transmit different signals at different times. This reduces the number of physical signal lines required. The simple circuit in Figure 1-8, for example, requires only 4 wires (1 signal line and 3 select lines) to transmit 8 signals. One problem with this type of multiplexing is that the multiplexed signals must be latched at the receiving end, so that the data value for a particular signal is held by the receiver and not lost when other data is put on the shared line(s). To illustrate this problem, note in the Y0 output waveform above the demultiplexer in Figure 1-8, that the data for the D0 input of the multiplexer is only present on the Y0 output when the D0 input and the Y0 output are selected. Between updates, the Y0 output always goes high for this demultiplexer. A little later we describe how a D latch on each demultiplexer output can be used to hold the values from the Y outputs between updates. Note also that for data transmission multiplexing such as that shown in Figure 1-8, the update rate or rate at which "the rotary switches are rotating" must be high enough so that details of the input waveforms between updates are not lost.

Time-division multiplexing is also often used to reduce the number of pins required on ICs and connectors. As an example of this, a given set of pins on a DRAM memory device is used to send in half of a large address in one operation and the other half of the address in a following operation. Multiplexers that are used to put either one set of signals or another set of signals on a group of parallel lines in this way are commonly called *bus multiplexers*. You will see many examples of this in a later discussion on DRAMs. One problem with this type of multiplexing is that transferring the two halves of the address, one after the other, takes more time than transferring all the address bits in parallel on a larger number of lines at the same time.

Decoders and Encoders

The terms decoder and encoder are often used incorrectly, so we will give you a simple scheme you can use to determine which device to look for when you need to solve a particular problem in one of your designs. The term *decoder* is used to describe a device which converts a compact code such as 4-bit Binary-Coded Decimal (BCD) code to a less compact code such as the seven-segment code required to drive 7-segment LED displays. By compact code, we mean one that uses all or most all of its available values. BCD, for example, is a four-bit code that uses 10 of the possible 16 values to represent decimal numbers 0-9, so it is a relatively compact code. Seven-segment code has 7 binary bits, so it has 128 possible values. However, only 10 or at most 16 of the 128 possible values are usually used, so this is an example of less compact code.

One common decoder is the 74LS47 BCD to 7-segment decoder. Another is the 74ALS138 1-of-8 low decoder shown in Figure 1-9a. If a 74ALS138 is enabled by asserting its G2A# input low, its G2B# input low, and its G1 input high, this device will assert one of its Y outputs low. The output asserted low is determined by the 3-bit code on the C, B, and A select inputs. This device well illustrates a concept of decoding that you will encounter many times in the context of microprocessors as, for example, in instruction decoding or address decoding. The 74ALS138 decodes the 3-bit code on its inputs to assert a low on one of its 8 individual outputs. Each of the eight outputs could, for example, be used to enable a specific data path in a microprocessor or select a specific memory device in a system. The device selected would then be determined by the three-bit code (address) applied to the select inputs.

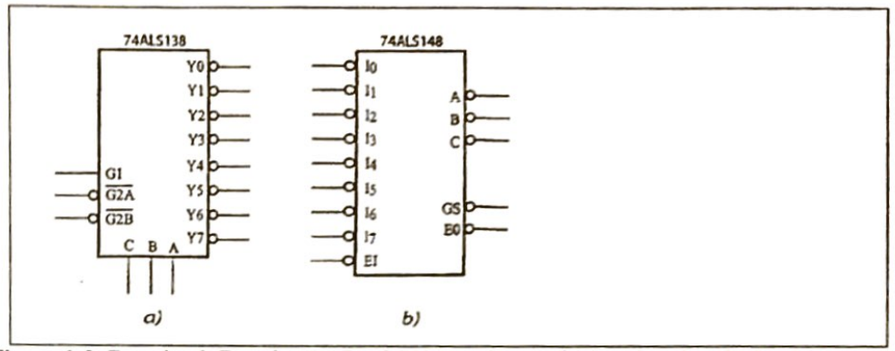


Figure 1-9. Decoder & Encoder. (a) 74ALS128 used as 1-of-8 low decoder. (b) 74ALS128 Priority encoder.

The term *encoder* is used to describe a device that converts a less compact code to a more compact code. Following our physical analogy, you could think of an encoder as passing the data through a funnel from the big end to the small end. A keyboard encoder, for example, might be used to convert a 16-bit, 1-of-16 keys pressed code to a more compact 4-bit binary code. As an example of an older, discrete device encoder, the 74ALS148 priority encoder, shown in Figure 1-9b, generates an active-low, 3-bit binary code that corresponds to the highest numbered data input which has a logic low asserted on it. For example, if the I3 input is asserted low and the I5 input is asserted low, the CBA outputs will show the asserted-low code for the highest numbered input with a low, the I5 input. The binary code for 5 is 101, so the active low code for 5 is 010, the inverse of 101, the active high binary code for 5. In the context of microprocessor systems, you will encounter this function in a Priority Interrupt Controller (PIC) that allows you to assign priorities to hardware devices that can interrupt normal program execution.

D Latches and D Flip-Flops – Operation, Timing, and Uses

Now that we have introduced important combinational logic concepts and devices, the next step is to discuss the clocked logic devices, circuits, and design techniques needed for our discussions of state machines and memory devices later in this chapter, and our discussions of microprocessors and microprocessor systems in later chapters.

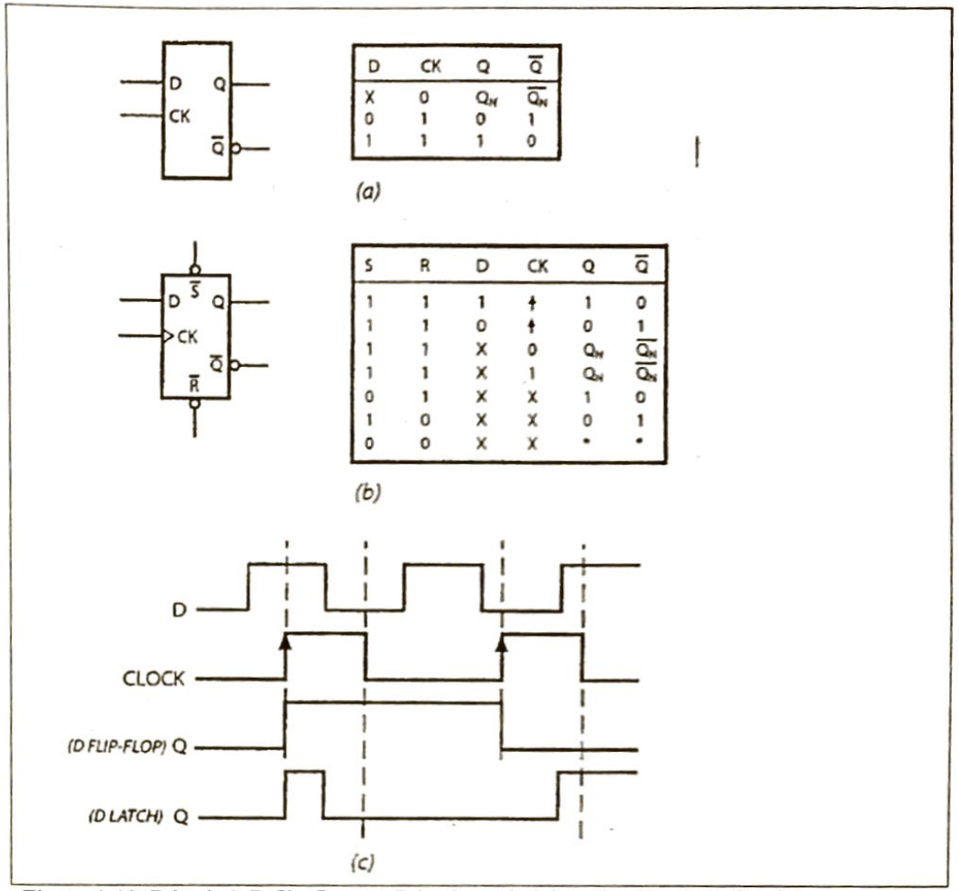
D LATCHES AND D FLIP-FLOPS

Figure 1-10a shows the schematic symbol and truth table for a *D latch*. In the symbol, D represents the data input, CK represents the clock or enable input, and Q and Q# represent outputs that are defined to have opposite or complementary logic levels. From the bottom two entries in the truth table you can see that, if the CK input is high the logic level on the Q output will be the same as that on the D input. To assist in analyzing timing waveforms, the way we like to express this is, "If the CK is high, Q follows D." If this is not clear to you, then it might help you to think of the operation of a D latch as, "If the CK is high, the device functions as if a piece of wire were connected between D and Q." Work your way across the timing waveform for the

D latch in Figure 1-10c to see examples of this. You should see that for the sections of the waveform where the CK signal is high, the Q output exactly follows the waveform on the D input. With an actual device, of course, there is some time delay required for a signal to propagate from the D input to the Q output.

These last two entries in the truth table are very similar to the truth tables for simple combinational logic gates in previous sections because a change on the input directly causes a change on the output(s). However, the first entry in the truth table for a D latch clearly shows how the behavior of this device is different from that of simple combinational logic device. This line in the truth table indicates that, if the CK input is low, the value on the Q output will stay the same and will not be affected by any change on the D input. The X in the D column for this line in the truth table indicates that the D input is a "don't care" when the CK input is low. A good way to state this behavior is, "When the CK input goes low, the logic level on the output at that time will be latched or held on the output and remain there until CK goes high again." The D latch waveform in figure 1-10c shows this. The first time CK goes from high to low, D is low, so the Q output is latched low. Q does not change until CK goes high again, even though the data on D changes. At the time when CK goes low the second time in the waveform in Figure 1-10c, D is high, so Q is latched high.

A D latch can be used to solve one of the problems we identified in our earlier discussion of multiplexers. Specifically, a D latch could be added to each of the eight outputs of the demultiplexer in the MUX-DMUX circuit in Figure 1-8 to hold the output signals constant between updates. The CK signals for the eight D latches could be generated from the 3-bit select signal with a 3-to-8 decoder similar to the 74LS138 in Figure 1-9a. For the earlier example we gave of multiplexing the halves of an address into a DRAM memory device, a bank of D latches is used to latch and hold the first half of the address, so it will be available to the internal circuitry while the input lines are being used to send in the second half of the address.



D	CK	Q	$\overline{Q}$
X	0	Q_N	$\overline{Q_N}$
0	1	0	1
1	1	1	0

S	R	D	CK	Q	$\overline{Q}$
1	1	1	↑	1	0
1	1	0	↑	0	1
1	1	X	0	Q_N	$\overline{Q_N}$
1	1	X	1	Q_N	$\overline{Q_N}$
0	1	X	X	1	0
1	0	X	X	0	1
0	0	X	X	*	*

Figure 1-10. D latch & D flip-flop. (a) D latch symbol & truth table. (b) D flip-flop symbol & timing truth table. (c) Timing diagram comparing responses of D latch & D flip-flop.

Figure 1-10b shows the schematic symbol and truth table for a *D flip-flop*. This symbol looks similar to that for a D latch but the behaviors of the two devices are very different. The key difference is that a D latch is a *level-triggered* device and a D flip-flop is an *edge-triggered* device. The small triangle next to the CK input on the D flip-flop symbol tells you that the signal on the D input will only be transferred to the Q output on the rising edge of the clock signal. This is represented by the top two lines in the truth table for the D flip-flop in Figure 1-10b. Work your way across the waveform for the D flip-flop in Figure 1-10c to see this in action. The D flip-flop takes a sample of the signal present on the D input on the rising edge of the CK and holds this value on the Q output until the next rising clock edge, when the Q output is updated with the level present on the D input at that time. In Figure 1-10c, if you compare the output

waveform for an edge-triggered D flip-flop with that for a level-triggered D latch, you can see that they are
very different. For a D latch, Q tracks the D input when the CK is high and then latches the value present on D when the clock goes low. A D flip-flop takes a "picture" of the signal present on the D input when the clock goes high and holds that picture until it takes a new picture on the next rising edge of the clock. You can use these waveforms to help you decide which device to use for a specific application in a design.

The D flip-flop shown in Figure 1-10b also has Set and Reset inputs. The bubbles on these inputs and the three bottom entries in the truth table indicate that these inputs are active low. As indicated in the truth table, if the S input is asserted low, the Q output will go to the high or Set state, regardless of the logic level on the D and CK inputs. Likewise, if the R input is asserted low, the Q output will go to the low or Reset state, regardless of the logic levels on the D and CK inputs. Set and Reset inputs that function in this way are referred to as "direct" or "asynchronous" because they can directly change the output without the need for a CK transition. For devices with synchronous Set and Reset inputs, the effect of asserting one of these inputs will only change the output when the next clock pulse occurs. The two asterisks in the bottom line of the truth table represent an indeterminate state and tell you that the output is unpredictable if both Set and Reset are asserted at the same time. In the next sections, we show you some common applications for D flip-flops but first we need to discuss some very important timing parameters that you must consider in your designs that use D latches or D flip-flops.

D-LATCH AND D FLIP-FLOP TIMING PARAMETERS

The three timing parameters are *setup time*, *hold time*, and *minimum pulse width*. Setup time and hold time are specified with respect to the active clock edge for a particular device. For a D latch such as that in Figure 1-10a, these times are measured from the falling edge of the clock because that is when the data input value is latched on the output. For positive, edge-triggered D flip-flops such as that in Figure 1-10b these times are specified with respect to the positive edge of the clock signal. Setup time, t_{SU}, is the minimum time that data must be present and held stable on the D input *before* the active clock edge. Hold time, t_H, is the minimum time that the data must be held stable on the D input *after* the active clock edge. Minimum pulse width is the minimum time the clock signal must be in the active state in order for the data to be transferred to the output correctly.

If you violate any of these three timing requirements in one of your designs, as for example by allowing the data signal to change too close in time to the active clock edge, the output of the latch or flip-flop may go into a ***metastable state***. Depending on the particular device, an output in a metastable state may stay at a voltage level that is half way between a logic high and a logic low or it may oscillate at a high frequency between logic high and logic low. Since a device output can remain in a metastable state for a time much longer than the time it normally takes for a signal to pass through the device, the device, metastability can easily propagate through an entire system and possibly cause the entire system to fail. The way to prevent the metastability problem is to carefully check every path through your design to make sure that data always arrives at the D inputs before the setup time minimum and stays valid until after the hold time minimum. This is relatively straightforward if the data signals in your design all have fixed time relationships to the clock signal. However, if a data signal occurs at random times with respect to the clock signal input, there will always be a chance that the signal will violate the setup or hold time and cause metastability. For this case, you can reduce the chances of metastability by

passing the asynchronous data signal to the D input of the flip-flop through a circuit called a *synchronizer*. A synchronizer consists of two or more D flip-flops connected in series and clocked by the system clock. Passing the signal through these additional flip-flops reduces but does not totally eliminate the possibility of violating the setup or hold time of the initial flip-flop. A synchronizer then reduces the possibility of propagating a metastable signal through the system but does not totally eliminate it. We will further discuss clocked circuit timing in the next section on shift registers, in a following section on state machines, in several later chapters.

Registers and Shift Registers

CONNECTIONS AND OPERATION

D flip-flops can be used individually or in groups to store binary data. A *register* is a group of D flip-flops connected with a common clock as shown in Figure 1-11a. A binary word applied in parallel to the D inputs of the flip-flops will be transferred to the Q outputs when the CK input is pulsed high. The binary word will be stored on the outputs of the register for use by other circuitry until a new word is written to the register. A new word is "written" to the register by simply applying the word to the D inputs and pulsing the CK input. As we will show in the next chapter, registers of this type are extensively used in microprocessors.

If the Q output of each flip-flop in a register is connected to the D input of the next device to the right, as shown in Figure 1-11b, then the register can function as a *shift register*. A logic high applied to the D input of the first flip-flop on the left in Figure 1-11b will be transferred or shifted to the Q output of that flip-flop and to the D input of the next flip-flop to the right when a rising edge occurs on the CK. The next rising CK edge will shift this data high to the output of the second flip-flop to the right. Each additional CK pulse will shift the logic high one bit position to the right through the register. One use of a serial shift register such as this is to delay a digital signal for a total time equal to the clock period multiplied by the number of flip-flops in the shift register.

Adding a two-input multiplexer on the D input of each of the flip-flops as shown in Figure 1-11c can allow the D inputs to be connected to either the Q of the preceding stage or to an external data input. If the multiplexers are switched so that the D inputs are connected to an external data source, a binary word can be written to the register in parallel. If the multiplexer inputs are then switched so they connect the D inputs to the Q outputs of the preceding flip-flop, the written word can be shifted through the register and out the Q0 pin of the register one bit at a time. When used this way, the shift register effectively converts the parallel word written in the register to a serial word shifted out on the Data Out, or Q0 pin. This is exactly the operation that is needed to convert the parallel data words in a microprocessor to the serial form in which data is transmitted over a network or the Internet. You will encounter this parallel-serial conversion function many times in later chapters.

Just as a shift register can be used to convert parallel data to serial form to send out on the Internet, a shift register can also be used to convert the serial data coming off the Internet to the parallel form needed in a microprocessor. For this conversion, the serial data is simply shifted serially into the Data Input of the register. When the complete word is shifted into the register, the parallel word on the Q outputs of the register is read by a microprocessor. As you will see in a block diagram for a current microprocessor in the next chapter, the shift register for parallel-serial conversion and the shift register for the serial-parallel conversion are usually put in a

single function block that is referred to as a Universal Asynchronous Receiver Transmitter or UART. In Chapter 5 we show how a UART is used to communicate with a Speech Synthesizer board.

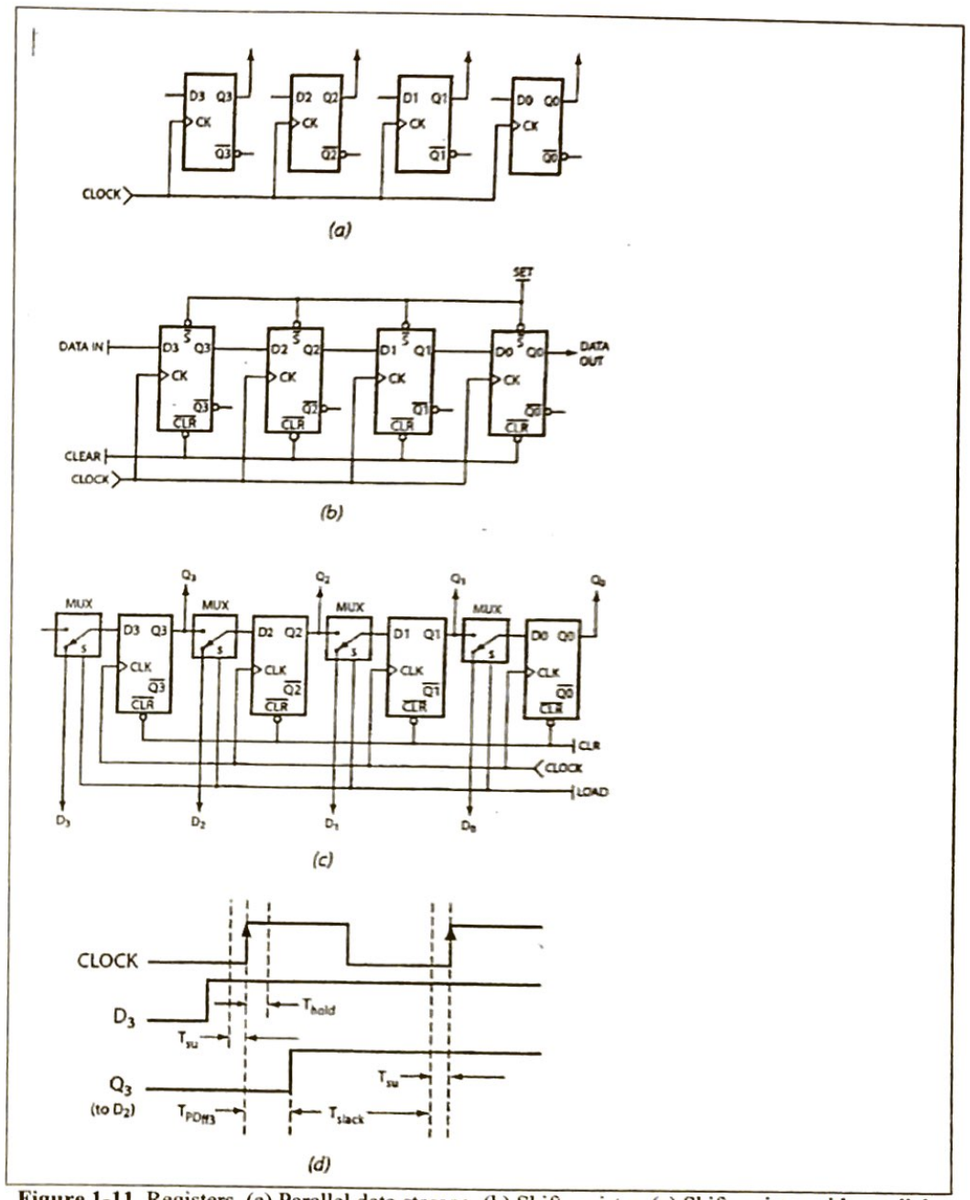


Figure 1-11. Registers. (a) Parallel data storage. (b) Shift register. (c) Shift register with parallel load capability. (d) Shift register timing diagram.

To summarize then, a shift register can be used to delay digital signals, to convert parallel data words into serial form, or to convert serial data words to parallel form. Before we go on to review another flip-flop based device that plays an important role in microprocessors however, we will use a simple shift register to take a first look at timing calculations for circuits containing flip-flops.

SHIFT REGISTER TIMING CALCULATIONS

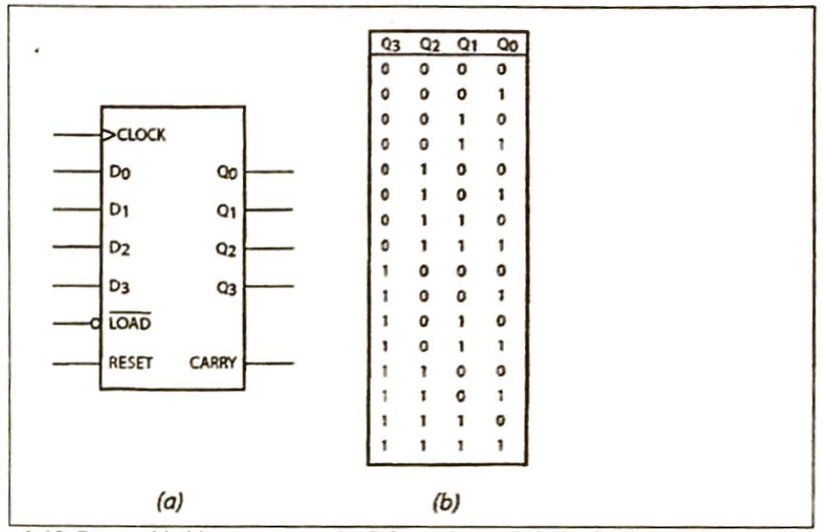
Now, suppose that you need to calculate the maximum frequency (minimum clock period) at which you can clock the shift register in Figure 1-11b. If you assume that the data input to the first flip-flop obeys the required setup and hold times, then with the help of the timing diagram for the shift register in Figure 1-11d, it is a simple matter to calculate the maximum frequency at which the data can be shifted through the register.

The data applied to the D input of the first flip-flop appears on the output of the flip-flop after a propagation delay from the clock rising edge that we will call t_{PD}. From a previous discussion, you know that to avoid metastability, data must be present and stable on the D input of the next flip-flop for a time called t_{SU} before the next rising edge of the clock pulse occurs. The minimum time between clock pulses, or in other words, the minimum clock period, is then just the sum of the t_{PD} and t_{SU}. As a specific example, if the t_{PD} is 3ns and the t_{SU} is 1ns, the minimum clock period is 4ns, which corresponds to a clock frequency of 1/4ns or 250 MHz. If the shift register is clocked at a frequency less than the maximum frequency of 250 MHz, there will be sometime between when the data appears on the output of one flip-flop and the time when it needs to be present on the input of the following stage. This time is commonly called *slack time* or t_{SLACK}. Note that the t_{HOLD} requirement for the flip-flop usually does not enter into this calculation because the propagation delay through a preceding stage will guarantee that the data will always be present on the D input long enough to satisfy the t_{HOLD} requirements.

PRESETTABLE/LOADABLE COUNTERS

Another very important flip-flop based device is a presettable counter such as the small, 4-bit counter shown in Figure 1-12a. If the RESET input on this counter is pulsed high, the outputs will all be forced to 0's. When the counter is clocked, the counter will step through the binary count sequence shown in Figure 1-12b. If the outputs are at 1111, then the next clock pulse will cause the outputs to "roll over" to 0000 and a Carry pulse will be produced on the CARRY output. This CARRY output could be used to enable another counter that generates the next four more significant bits of an 8-bit total count. The maximum count for a binary counter is 2^N-1, where N is the number of bits.

The most interesting feature of the counter in Figure 1-12a is that it is *loadable* or *presettable*. This means that, if the LOAD# input is asserted low, the data values on the D inputs will be transferred to or "loaded" on the Q outputs when the next clock pulse occurs. When the LOAD# signal is raised high again, the counter will increment from the loaded count on the next clock pulse. For future discussions, the key point to remember is that this type of counter can be "jumped" to a desired count by simply loading that count. The counter can then be counted up from that count.



Q3	Q2	Q1	Q0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

Figure 1-12. Presettable binary counter. (a) Schematic symbol. (b) Count sequence.

This counter is a special type of synchronous state machine in which the outputs change from one state to the next when the machine is clocked. Synchronous state machines are very important parts of microprocessors, common interface devices, and custom interfaces you may need to design. Therefore, it is important that you understand very well how state machines work and how to design them accurately and efficiently. In the next section we give a solid overview of the key concepts of general state machine operation and basic state machine design techniques, to help introduce these to you or review them for you.

Synchronous State Machines

STATE MACHINE STRUCTURE AND OPERATION

Figure 1-13a shows the block diagram for a *synchronous state machine*. By synchronous, we mean that the outputs of the flip-flops can only change from one state to another when a clock pulse occurs or, in other words, synchronously with clock pulses. For comparison, the outputs of asynchronous circuits, such as simple combinational logic devices, change immediately after some propagation delay when an input signal changes. For asynchronous devices, no clock signal is needed to cause a change on an output.

A synchronous state machine such as that in Figure 1-13a functions as follows:

1. The present state is held on the outputs of the flip-flops.
2. The present state and possibly some external signals are *decoded* by the Next State Decoder logic to produce the desired next state signals or *excitation* on the inputs of the flip-flops.

3. When a clock pulse occurs, the desired next state values on the inputs of the flip-flops are transferred to the outputs of the flip-flops.
4. This makes the state present on the outputs of the flip-flops the desired next state.
5. These present state outputs and possibly some external signals are *decoded* by the Output Decoder to produce the output signal(s) desired for that new state.

Working through the example state machine schematic in Figure 1-13b should help clarify the operation and terms in your mind. To start, the Next State Decoder logic in the dotted box decodes the state outputs, QA and QB, and the external signal E to produce the desired next state values on the inputs of the D flip-flops. When the clock signal goes high, this decoded next state on the D inputs will be transferred to the Q outputs and the state on the outputs will be the desired next state. Now let's look at the Output Decoder.

You can tell directly from the schematic symbol and connections that the P output signal will be high if QA is low AND QB is low or, in other words, when the state machine is in state 00. The P signal is an example of a *Moore output signal*. A Moore-type output signal is one that depends only on the present state outputs. A key point to remember about a Moore output signal is that, since it is only determined by the present state, it will only change when the state changes. Another way to say this is that Moore output signals will always change synchronously with the clock signal. For some applications, we assign the binary values for the states such that the desired Moore outputs are produced directly on the outputs of the flip-flops. This is referred to as a *direct output* state machine. The advantage of a direct output state machine is that, since the desired output signal does not need to propagate through any output decoder logic gates, there is less propagation delay between the rising edge of the clock and the time when the desired Moore output signal is asserted. A special case of a direct output state machine is a *one-hot* state machine, which is designed so that only one flip-flop output at a time is ever asserted. An example sequence for a one-hot type state machine might be 000, 001,010, 100, etc. (One disadvantage of a one-hot state machine is that the number of flip-flops needed is equal to the number of states, so a one-hot machine usually needs more flip-flops than does a decoded output state machine.)

Now take a look at the Z signal in Figure 1-13b. From the schematic symbol that generates the Z signal, you can see that the Z signal will be high if QA is high, QB is high, AND the external M signal is high. In other words, the Z output will be high, if the machine is in state 11 AND the M signal is high. A state machine output that is determined by the state AND by one or more external signals is called a *Mealy output signal*. The key point to remember about a Mealy output is that, since the output is a function of both the state and the external signal(s), a Mealy output signal can change when the state changes OR when the external signal changes. To summarize then, you implement a Moore type output if you want the output to only change synchronously with the clock, and you implement a Mealy type output if you want the output to change when the state changes or when an external signal changes. Now that basic state machine operation and terminology are fresh in your mind, let's look at how you design and implement a state machine for a given application.

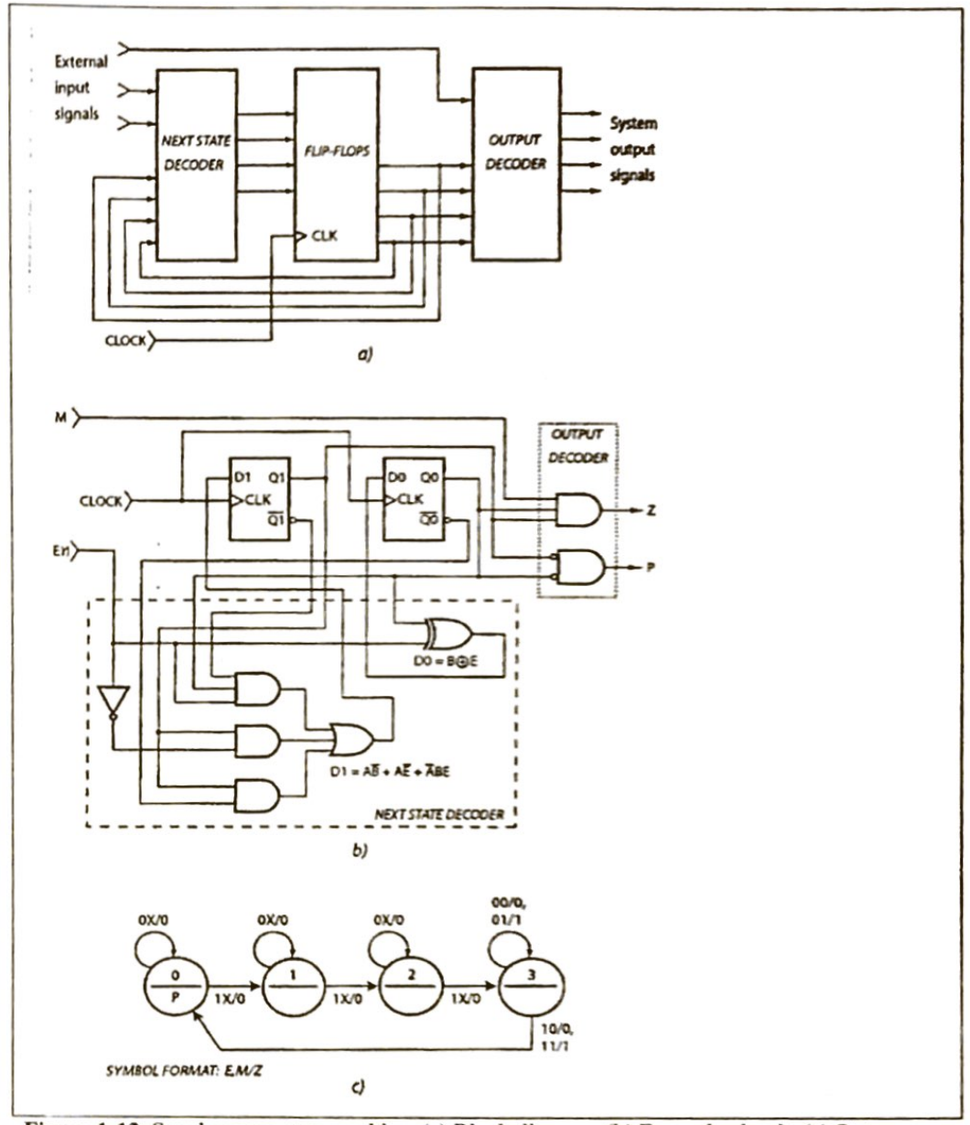


Figure 1-13. Synchronous state machine. (a) Block diagram. (b) Example circuit. (c) State diagram for example circuit.

STATE MACHINE DESIGN

The major steps we use to design synchronous state machines are as follows:

1. As in any design project, research and think a great deal about the problem you want the circuit to solve.
2. Draw a rough block diagram for the machine, showing the external input signals and the desired output signals.
3. Draw a detailed timing diagram showing the desired time relationship between the clock, the input signals, and the desired output signals.
4. Divide the timing diagram into states, based on when you want the output signals to be asserted.
5. Decide whether you want each output signal to be a Moore signal or a Mealy signal.
6. For each state decide on the possible next state(s) and the input condition(s) that will cause a transition to a particular next state.
7. Draw a state diagram that shows the desired states, binary state assignments if required, state transition conditions, and output signals.
8. Translate the state diagram into a Hardware Description Language (HDL) such as Verilog or VHDL.
9. Compile the HDL description of the design and correct any errors.
10. Simulate the design to verify that the logic and timing are correct.
11. Map the design into a Programmable Logic Device such as a PAL or an FPGA, or an Application Specific Integrated Circuit (ASIC).
12. Insert the device in a hardware prototype for final testing.

Since we don't have space here to go through the entire process in great detail, we will just discuss steps 7 –12. As a very simple example for step 7, Figure 1-13c shows the state diagram (we often call them bubble diagrams) for the state machine design in Figure 1-13b. In these diagrams, each state is represented by a large circle. The name of the state is written in the top of the circle and the names for any Moore type signals asserted during a particular state are written in the bottom of the state bubble. Note that the P output signal is shown in the bottom of the state S0 bubble because we want it to be asserted in state S0. An arrow indicates the action that takes place when a clock pulse occurs. A looping arrow to the same bubble indicates that the machine stays in the same state when a clock pulse occurs. An arrow to another bubble indicates a transition to that state when a clock pulse occurs.

Input signals and Mealy output signals are shown by symbols next to each transition arrow. The symbol format statement under the state diagram tells you that the order of the signals is E,M/Z. On the state diagram, note that if the E signal is a 1, the machine will transition to the next state when a clock pulse occurs. If the E signal is a 0 when a clock pulse occurs, the machine will stay in the same state. Now let's look at the M input signal and the Z Mealy output signal.

For this discussion, it is helpful to directly see the relationship between the state diagram and the final circuit in Figure 1-13b. From the circuit you can see that we wanted the Z signal to be asserted only in state S3 (QB = 1 and QA = 1) and then only if the M signal was also high. Take a look at the symbols next to state S3 to see how you represent this in the state diagram. For the symbols next to the tail of the looped arrow, there are two cases. If M = 0, then Z = 0 and if M = 1, then Z = 1 as desired. For the symbols next to the arrow leaving the right side of state S3, there

are the same two cases. To summarize then, the state diagram shows that as long as the machine is in state S3, the M signal will determine the logic level on the Z signal. The E value of 1 in the symbol to the right of the state S3 bubble indicates that on the next clock pulse the machine will transition back to state S0. In state S0, we want the Z signal to be a 0, regardless of the level on the M signal. A shorthand way to indicate this is to put an X in the M position in the symbol to represent a "don't care" for the M signal.

For more complex state machines with many states and many output signals, we usually use a table to show the signals that will be asserted in each state, rather than writing them in the bubbles or in the lines next to the transition arrows. For either case, it is usually an easy step to translate a carefully developed state diagram to an HDL circuit description that can be compiled, simulated, and implemented in a PLD as we describe briefly in the next section.

USING AN HDL TO IMPLEMENT A STATE MACHINE IN A PLD

As mentioned in the last section, once you have a state diagram and signal output table for a desired state machine, the next step is to translate the diagram into a Hardware Description Language (HDL) such as Verilog or VHDL. As a simple example of this translation, Figure 1-14a shows the Verilog HDL description for the simple state machine in Figure 1-13. We chose Verilog for this example because all of the major companies that we work with are now using Verilog for their designs. (If you are familiar with VHDL, you will notice that Verilog has a somewhat different syntax from VHDL but the basic structure of the Verilog description is very similar to a VHDL description.)

We obviously can't cover all of the Verilog language here but we can show you enough for you to see how easy it is to translate the state diagram for a desired state machine to an HDL and then simulate the design before mapping it into a Programmable Logic Device (PLD) or Application Specific IC (ASIC). If you plan a career in microprocessor system design and have not already learned this process, you should put learning the HDL to hardware path on your "to do" list for future action, because it is a very useful tool to have in your skill set toolbox. Verilog is in many ways very similar to the C programming language so, if you are familiar with C, it is a relatively easy step to learn Verilog. This similarity also strongly makes the point that there is really no longer a sharp dividing line between hardware and software in the design world.

```
// Verilog for simple state machine with
// Moore and Mealy outputs.
// Copyright Douglas V. Hall, Aug. 2016

module simple_sm (En,clock,Reset,M,P,Z,pstate);
  input En,clock,Reset,M;
  output Z,P,pstate;
  // pstate = present state
  reg Z,P;
  reg [1:0] pstate,nstate;
  // nstate = next state
```

```
    parameter S0=2'b00,S1=2'b01,S2=2'b10,S3=2'b11;
    // state assignments

  always@(posedge clock or negedge Reset)
  // flip-flop block
    if (Reset==1'b0) pstate <=S0;
    // asynchronous reset
    else pstate <= nstate;
    // response to rising edge of clock

  always@(pstate or En)
  // next state decoder
    case(pstate)
      S0: if (En) nstate=S1;
      S1: if (En) nstate=S2;
      S2: if (En) nstate=S3;
      S3: if (En) nstate=S0;
      default: nstate=S0;
      // not needed, all states used
    endcase

  always@(pstate or M)
  // output decoder
    case(pstate)
      S0: begin P=1'b1; Z=1'b0; end
      S1: begin P=1'b0; Z=1'b0; end
      S2: begin P=1'b0; Z=1'b0; end
      S3: begin P=1'b0; if(M==1'b1) Z=1'b1;
          else Z=1'b0; end
    endcase

  endmodule
```

(a)

```
    // Verilog testbench for simple Moore and Mealy state machine
    // Copyright Douglas V. Hall, Aug. 2016

    module t_simple_sm();
    wire P,Z;                          // specify as wires to make visible for
simulation
    wire [1:0] pstate;
    reg clock,Reset,M,En;              // register so simulator maintains values
    simple_sm M1 (En,clock,Reset,M,P,Z,pstate);      // UUT instantiation

    initial begin
    #100 $finish;                                  // run simulation for 100 time units
    end

    initial begin                                  // specify stimulus signals
    clock=1; Reset=1; En=0; M=0;
    #5 Reset=0;
    #5 Reset=1; En=1; M=1;
    #25 M=0;
    #5 En=0;
    end

    always
      #5 clock=~clock;                             // generate clock with 10 unit period

    endmodule
```

(b)

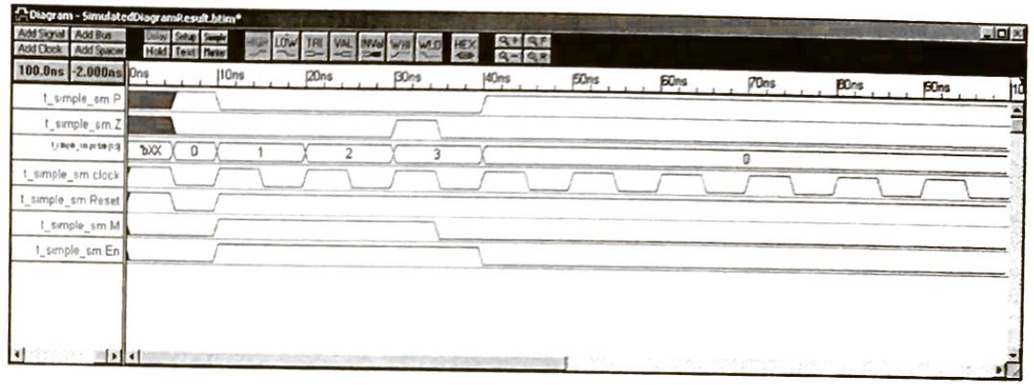


Figure 1-14. Implementing the state machine in Figure 1-13 using Verilog HDL. (a) Verilog description for the state machine. (b) Verilog testbench for the design in part a. (c) Verilog output waveforms.

The module statement at the start of the Verilog description in Figure 1-14a gives the module the name `simple_sm` and lists the inputs and outputs for the module in a port list. The next four lines specify which ports are inputs, which ports are outputs, and which ports should be treated as registers. Ports identified as registers maintain their values between evaluations during simulation. The parameter statement identifies four constants, S0, S1, S2, and S3 that are the binary state assignments for the four states of our state machine. The `S0=2'b00` clause in the statement identifies S0 as a 2-bit binary quantity and assigns it the binary value 00. The first section of the description is similar to declaring variables and constants at the start of any high-level language program.

The next three sections of the Verilog description directly correspond to the three major blocks of a state machine shown in Figure 1-13a. There are many styles for writing Verilog descriptions but for state machines, we like to separate the major parts of a design in this way so that the operations in each section are clearly visible and understandable. In Figure 1-14a, the section that begins with `always@(posedge clock or negedge Reset)` corresponds to the flip-flop section of the state machine block diagram. The 'always' statement specifies that the actions immediately below the statement should be executed whenever a positive (rising) edge occurs on the clock input or a negative (falling) edge occurs on the Reset input. If the execution was triggered by Reset = 0, then the flip-flops will be reset to state S0. This is an asynchronous reset because it does not require a clock signal to cause the flip-flops to go back to state S0. If the execution of this always statement was triggered by a rising edge on the clock signal, then the flip-flops will go to the next state as determined by the Next State Decoder.

The section of Figure 1-14a that begins with always@(pstate or En) implements the Next State Decoder for the state machine. Execution of this section will be triggered whenever pstate changes or the En input changes. The lines in the case statement here directly follow the state diagram in Figure 1-13c, so they need little comment. Note that we did include the default: nstate=S0 statement so that for any state not explicitly stated in the case statement, the machine will be returned to state S0 on the next clock pulse. This line is not needed in this example

because all four of the possible states for the two flip-flops are used, but we included it to remind you to do this in any machine where you don't explicitly use all of the states available with the number of flip-flops you are using.

The section of Figure 1-14a that begins with always@(pstate or M) represents the Output Decoder block in Figure 1-13a and specifies the desired P and Z outputs for each state. The output P is a Moore-type signal that is unconditionally asserted high only in state S0. The Z signal is a Mealy-type signal that is asserted in state S3 only if the M input is high. It is a small point but note that in the output decoder section, we specified the desired level on P and Z for each state. If you don't do this, the synthesis tool will imply that these signals should be latched to hold their values and implement latches in the Output Decoder instead of just the desired combinational logic. Note that begin and end keywords are used to bracket multiple statements for each case action in this section.

After you create an HDL description and get it to compile without errors, the next step is to simulate the design with a simulator program to verify that its operation is correct. In order to simulate the design, you need to create stimulus signals that the simulator applies to the inputs of the design. You can create these stimulus signals with an HDL test bench or with a graphical test vector editor. Figure 1-14b shows a Verilog testbench for the state machine design in Figure 1-14a. We don't have space here to go into all the details, but the key points are that at the start of a test bench module, you identify the outputs that you want to see in the simulation waveforms as wires. You identify as type reg the inputs that you will be applying so that the simulator holds these signals constant between updates. Next you *instantiate,* or in other words specify an instance of the module that you are simulating, and give it a name. In this example the statement simple_sm M1(En,clock,Reset,M,P,Z,pstate) instantiates an instance of our state machine and gives it the name M1.

The first section of Figure 1-14b that begins with initial begin tells the simulator to run the simulation for 100 simulator time units and then return control to the user. The second section of Figure 1-14b that begins with initial begin specifies the actual stimulus signals that will be generated by the simulator. To create these signals, we usually draw the desired waveforms on a piece of graph paper and then just translate them into the appropriate time statements in the testbench, or draw them directly into the simulator using a waveform editor.

In Figure 1-14b, we start by setting clock=1, Reset=1, En=0, and M=0. Then after 5 simulator time units, we set Reset=0. After another 5 simulator time units, we set Reset high again, set En=1 to enable counting, and set M=1 so that Z will be asserted when the machine reaches state S3. After another 25 simulator time units, we set M=0 again to determine if the Mealy-type Z signal works correctly. The always, #5 clock=~clock statement is simply an efficient way to create a clock signal that toggles every 5 simulator time units and thus produces a clock signal with a period of 10 simulator time units. Note that specific delays in nanoseconds or picoseconds can be inserted in the HDL description and the simulator can be instructed to execute a simulation with appropriate simulation time units of nanoseconds or picoseconds, so you can get first estimates of the overall timing for a design. However, for the example here we left these delays out in order to keep the example as simple and uncluttered as possible.

Figure 1-14c shows the simulator output waveforms produced by Verilogger Pro from SynaptiCAD for the design in Figure 1-14a and the testbench in Figure 1-14b. As you can see, the machine is reset to state S0 by Reset being asserted low and, as desired, the P output is asserted high in state S0. After the machine is enabled by En being asserted high, the machine steps thought the desired sequence of states. At the start of state S3 the M signal is high, so the Z

signal is asserted high. However when the M input drops low halfway through state S3, the Mealy Z signal immediately drops low as desired. As a final point, note that the counter advances to state S0 after state S3 even though the En signal is low at the time of the rising edge of the clock at the end of state S3. The reason for this is that due to the way the code is written in Figure 1-14a, the next state is determined when the present state, pstate, changes at the start of state S3. At this time, the En input is still asserted, so the next state, nstate, will be correctly determined as state S0. On the next rising clock edge, the machine will transition to state S0. However, from then on, En = 0 is detected in the Next State Decoder and the machine stays in state S0.

Once the simulation shows that your design functions perfectly, you then use the Design Automation tools to *map* the design into a Programmable Logic Device such as a PAL or an FPGA, or an ASIC. The term *map* here means to connect the programmable logic blocks in a specific device to implement the desired functions. The design tools automatically implement the Next State Decoder, flip-flops, and Output Decoder and all the interconnections for your state machine. The tools also create the file needed to actually program the device to implement your design in a PLD. Although we have shown a very simple state machine example here, some current FPGA devices contain over 1,000,000 gates and contain enough Configurable Logic Blocks (CLBs) to implement a complete 32-bit microprocessor with a large amount of custom interface circuitry. However, whether you do a simple design or a very complex design, you need to verify not only that the logic works correctly but also that the timing is correct. In the next section we show you how to calculate the maximum clock frequency for a simple state machine such as the one we have just analyzed.

STATE MACHINE TIMING CALCULATIONS

Calculating the minimum clock period for a general synchronous state machine is just a small extension of the technique described in a previous section for calculating the minimum clock period for a simple shift register. In that case, the minimum clock period was just the sum of the delay, t_{PD} , for the flip-flops plus the t_{SU}, because the data is connected directly from the output of one flip-flop to the input of the next. However, for a synchronous state machine such as that in Figure 1-13a, the signals on the Q outputs loop around and propagate through the Next State Decoder to get to the D inputs. Therefore, the propagation delay of the Next State Decoder must be included in the calculation of the time from a rising clock edge to valid data on the D flip-flop inputs. The minimum clock period then is just the sum of the three times around the loop, t_{PD} + t_{NSD} + t_{SU}. Note that the Output decoder is not part of the feedback loop, so it is not included in this timing calculation. As a specific example, assume that the flip-flops have a t_{PD} of 3ns, the NSD has a t_{NSD} of 2ns, and the flip-flops have a t_{SU} of 1ns. The minimum clock period would be 3ns + 2ns + 1ns = 6ns. This corresponds to a maximum clock frequency of 1/6ns = 167 MHz.

To summarize, you calculate the minimum clock period and indirectly the maximum clock frequency for a state machine such as this by simply summing the times around the Flip-flop and Next state decoder loop from one clock edge to the next clock edge. **An important design concept for you to stick in your mind for future reference is that, any increase in the propagation delay in the loop will reduce the maximum frequency at which you can clock the machine.**

Now that we have reviewed combinational logic, flip-flops, registers, counters, and state machines, the only major components left to review here are memory devices. Then in the next chapter, we show you how to put the pieces together to make a microprocessor and a

microcomputer. Note that the discussions of memory devices here are intended to review the basic memory types and characteristics or serve as an introduction if you have not studied these before. In Chapter 6 we will discuss in greater detail the interfacing and timing considerations for a variety of these memory devices.

Semiconductor Memory Devices

INTRODUCTION

A register stores a single binary data word but a memory device can store thousands of data words. To refresh or introduce you to the basic structure and terminology used for memory devices, we will use a simple *Read-Only Memory* or ROM as an example. The term read-only memory is somewhat misleading because several current types of ROMs allow new data to be written to the devices as well as allowing data to be read. The key point to remember about ROMs is that they are *non-volatile*. Non-volatile means that stored data is retained by the device and not lost when the power is turned off. This characteristic is important for applications where we want stored instructions or data to be present immediately when the power is applied to a system.

Figure 1-15a shows the schematic symbol for a small ROM device and Figure 1-15b shows how two of these devices can be connected in a system to store twice as many binary words as could be stored by a single device. You can determine considerable information about the memory devices by just looking carefully at these diagrams

The eight data outputs, D0-D7, on each device tell you immediately that this memory device stores 8-bit data words. You can think of the data words stored in a memory as being in a long numbered list and the location of a stored word in the list is called its *address*. The number of words stored in a memory device is just 2^N, where N is the number of address lines. The device in Figure 1-15a has 15 address lines, A0-A14, so the device stores 2^{15} or 32,768 words. For simplicity, we simply refer to this as a 32K x 8 or 32 KByte device.

The data outputs on the ROM in Figure 1–15a are *three-state* outputs, which, as we stated earlier in the chapter, means that an output can be at a logic low state, a logic high state, or a high-impedance floating state. To read a desired word from this ROM, you need to supply three types of signals. First, you need to apply the address of the desired word to the address inputs of the device. Second, you need to assert the Chip Enable, CE#, input low to power up the device from a low-power state to normal operating state. Third, you need to assert the OE# input to enable the three-state Data outputs. Remember from a previous discussion that, if three-state outputs are enabled, they output a standard logic low or logic high. If three state outputs are disabled they go to a high-impedance floating state and are essentially disconnected. To prevent logic conflicts when you connect several ROMs or other memory devices on a shared *data bus* as shown in Figure 1-15b, you must use devices that have three-state outputs and you must make sure that only one device at a time is enabled to output data onto the bus. In Chapter 6 we will discuss the design of a circuit called an *address decoder*, which decodes upper address lines from the microprocessor to produce a signal that enables a desired memory device and only that device at a particular time. Note that the *address bus* lines in Figure 1-15b are connected to the two devices in parallel. An address applied to these inputs will select the same addressed byte in each of the devices. However, since the address decoder enables only one device at a time, only

the enabled device will output the addressed word on the data bus. Now that we have reviewed basic memory terminology and connections, let's take a first look at memory device timing.

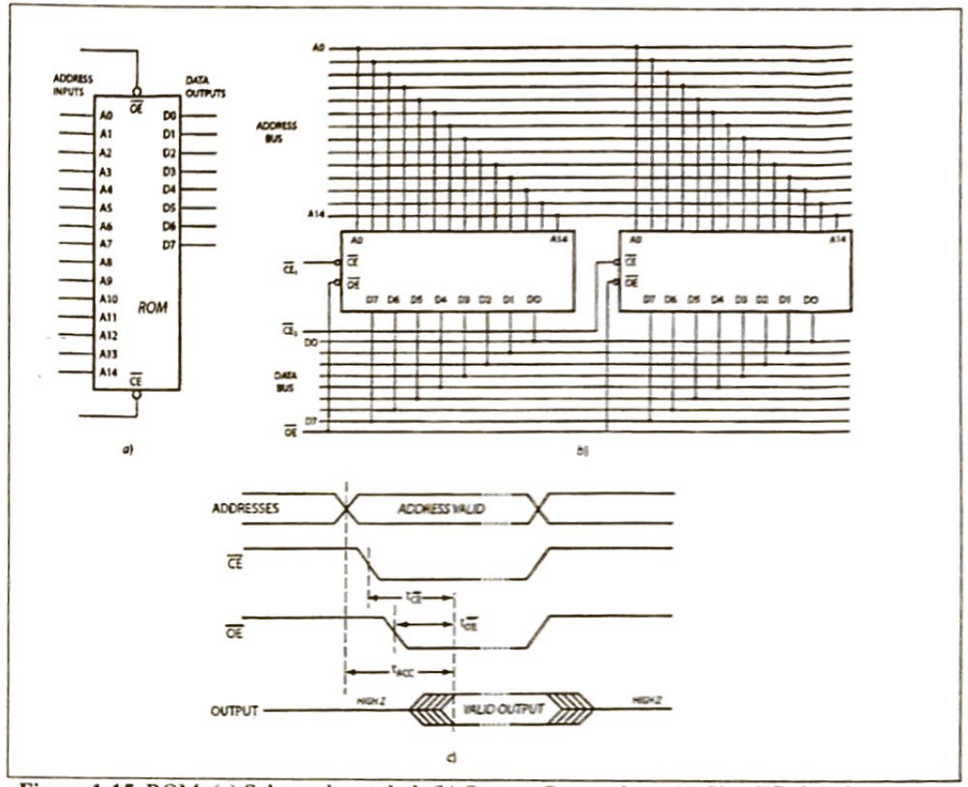


Figure 1-15. ROM. (a) Schematic symbol. (b) System Connections. (c) Simplified timing diagram.

Figure 1-15c shows a simplified timing diagram for an asynchronous ROM such as the one in Figure 1-15a. Assuming the CE# is already asserted when an address is applied to the address inputs of the ROM, there will be some propagation delay before the addressed data appears on the data outputs. This time is called the *Address Access Time* and is represented on the timing diagram as t_{ACC}. Basically, this is the time you need to wait for valid data to be present on the outputs after an address is applied to the device, assuming that the device is already powered up by CE# being asserted and the outputs enabled by OE# asserted. Likewise, if an address is already present on the address inputs when the CE# is asserted, it takes some time for the device to power up and for the output buffers to turn on. This time is called the *Chip Enable Access Time*, and is represented as t_{CE} on the timing diagram. The time labeled t_{OE} in Figure 1-15c

represents the *Output Enable Time*. This time is the time required for the output buffers to turn on and supply data on the outputs after the address has been applied and the CE# is asserted. As we demonstrate many times throughout the book, these memory access times are one of the major limiting factors for the performance of a microcomputer system and are therefore very important design considerations. For now, however, let's take a look at the different types of memory devices used in current systems and their basic characteristics.

ROM Types

Some of the most common types of ROM are:

MROM – Mask-programmed with data during manufacturing; cannot be altered by user.

PROM – Programmed by user by blowing fuses; cannot be altered except as can be done by blowing additional fuses.

EPROM – Electrically Programmable by user; erased by shining ultraviolet light through a quartz window on top of package. (Only found in old systems.)

EEPROM – Electrically Erasable PROM; programmable by user; erased by electrical signals so it can be erased and reprogrammed in the circuit. They are erased and reprogrammed one byte at a time, so they are flexible. However, the write speed is much slower than the read operation.

Flash EEPROM – A type of EEPROM electronically programmable in blocks rather than one byte at a time as with standard EEPROMs. The two types currently in use are NOR Flash devices and NAND flash devices. NOR Flash devices are accessed in parallel as are standard ROMs such as that in Figure 1-15a. NOR devices have fast read speeds, so they are appropriate for holding executable code such as the boot code for a system. NAND Flash devices have faster write speeds and higher storage capacity than NOR devices. Data is written to or read from a NAND flash device as a serial string of bytes or half words in much the same way as they are written to and read from a hard-disk drive. Therefore, they are appropriate for storing data as required in a Solid State Drive (SSD). In Chapter 6, we discuss Flash memory interfacing and timing in detail.

Synchronous Flash EEPROM - Synchronous version of the asynchronous NOR Flash described in the preceding paragraph. Addresses and control signals for read operations are transferred into the device on the rising edge of a memory bus clock signal and data is transferred out on a later rising edge of the memory bus clock signal. The operation of the device is then said to be synchronous with the memory bus clock.

Except for Synchronous Flash EEPROM, all of the types of ROMs we have discussed are asynchronous, because their inputs and outputs are controlled by signals such as addresses, CS#, and OE# rather than by a system clock. As we discuss in detail later, synchronous devices allow faster data transfer than do asynchronous devices.

Static RAM Types - SRAM and SSRAM

The name RAM stands for *Random-Access-Memory*, but since ROMs are also random access, the name probably should have been "read-write" memories because the devices can be written to and read from at the same speed, unlike the EEPROMs and Flash EEPROMs discussed in the previous section. A static RAM or SRAM is essentially an addressable array of flip-flops. The term *static* here means that data stored in the RAM will be retained until new data is written or the power is turned off.

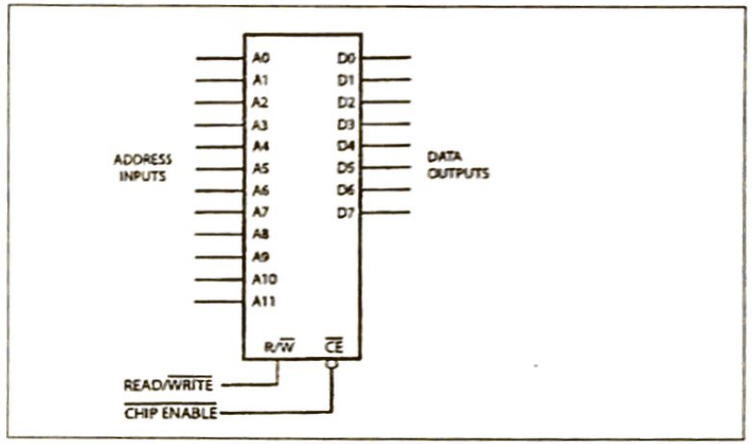


Figure 1-16. Schematic symbol for Static RAM

Figure 1-16 shows the schematic symbol for a small static RAM. To read a byte from this device, a memory controller applies an address to the address inputs, asserts the R/W# input high and asserts the CE# input low. After some access time, the addressed data word is present on the Data Outputs. To write to the RAM, a memory controller applies an address to the address inputs, asserts the R/W# input low, asserts the CE# input low, and applies the word to be written on the Data Outputs.
The RAM in Figure 1-16 has no clock input, so the input and output signals are asynchronous. This means that they have no fixed time relationship with the overall system clock and therefore create memory access timing problems at high system clock frequencies. Furthermore, the address inputs are not latched in this device, so the memory controller needs to keep the address stable on the address inputs for the entire memory access cycle. Likewise, the memory controller must keep the R/W# and CE# signals present for the entire memory access cycle. To solve these problems and thus greatly improve performance, most systems now use Synchronous SRAMs or SSRAMs.

To convert an asynchronous memory device to a synchronous device, you add D flip flops on the address inputs and the control inputs as shown in Figure 1-17. When clocked by a system clock, these flip flops will latch the address and control signals, so the memory controller does not have to keep the address and control signals on the lines for the entire memory cycle.

This leaves the memory controller free to generate the next address. You also add D flip flops on the memory array data outputs. When clocked, these flip flops transfer the data read from the memory array to their outputs. Since this data is latched on the outputs of the flip flips, the memory device can be working on the reading the next requested word. (Remember that in an asynchronous memory device, the address and control signals must be held on the device until the word is read from the device and a second read cannot be started until the first is fully done.) To see more detail of how this is implemented and see the timing for an actual device, take a look at Figure 1-18a.

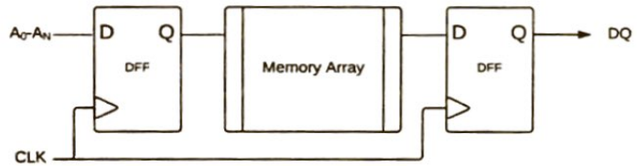


Figure 1-17. Converting asynchronous memory device to synchronous.

Figure 1-18a shows the internal architecture of a Cypress Semiconductor CY7C1327B Synchronous SRAM. (This specific device is now out of production but it is easier to see the basic architecture and operation in this device than in the current versions that have additional features to improve performance.) The key difference from the asynchronous SRAM is that, in this device, all of the input and output signals have registers that are clocked by the memory system clock. Figure 1-18b shows a read-cycle timing diagram for the device. Once an applied address is latched by a rising clock edge, the memory controller can remove that address from the inputs and work on generating the next memory address. Likewise, storing the memory control signals in registers means that they only need to be present on the register inputs long enough to be latched. If you work your way across the timing diagram in Figure 1-18b, you should see that an address and the control signals are transferred into the SRAM core on the first clock edge. These then propagate though the SRAM and produce the desired word on the inputs of the output latches. The next rising clock edge transfers this data word to the data output latches and the third rising clock edge transfers the data to the device outputs. This third clock edge also transfers a new address and new control signals into the SRAM core. For a *burst mode read*, the SSRAM will internally increment the address supplied for the first read and output another data word on each rising clock edge. If you think a little about this you might make the connection in your mind that this is the same way that data moves through a shift register. Systems that move data through one shift register-type stage on every clock cycle in this way are often referred to as *pipelined*. The full name for the device shown in Figure 1-18a then is "Pipelined Synchronous Cache RAM." If you read the specification for a current PC, you might see a statement that it contains a "pipelined synchronous SRAM cache." We will discuss caches in the next chapter but next here we will move on to another common type of RAM used in microcomputer systems.

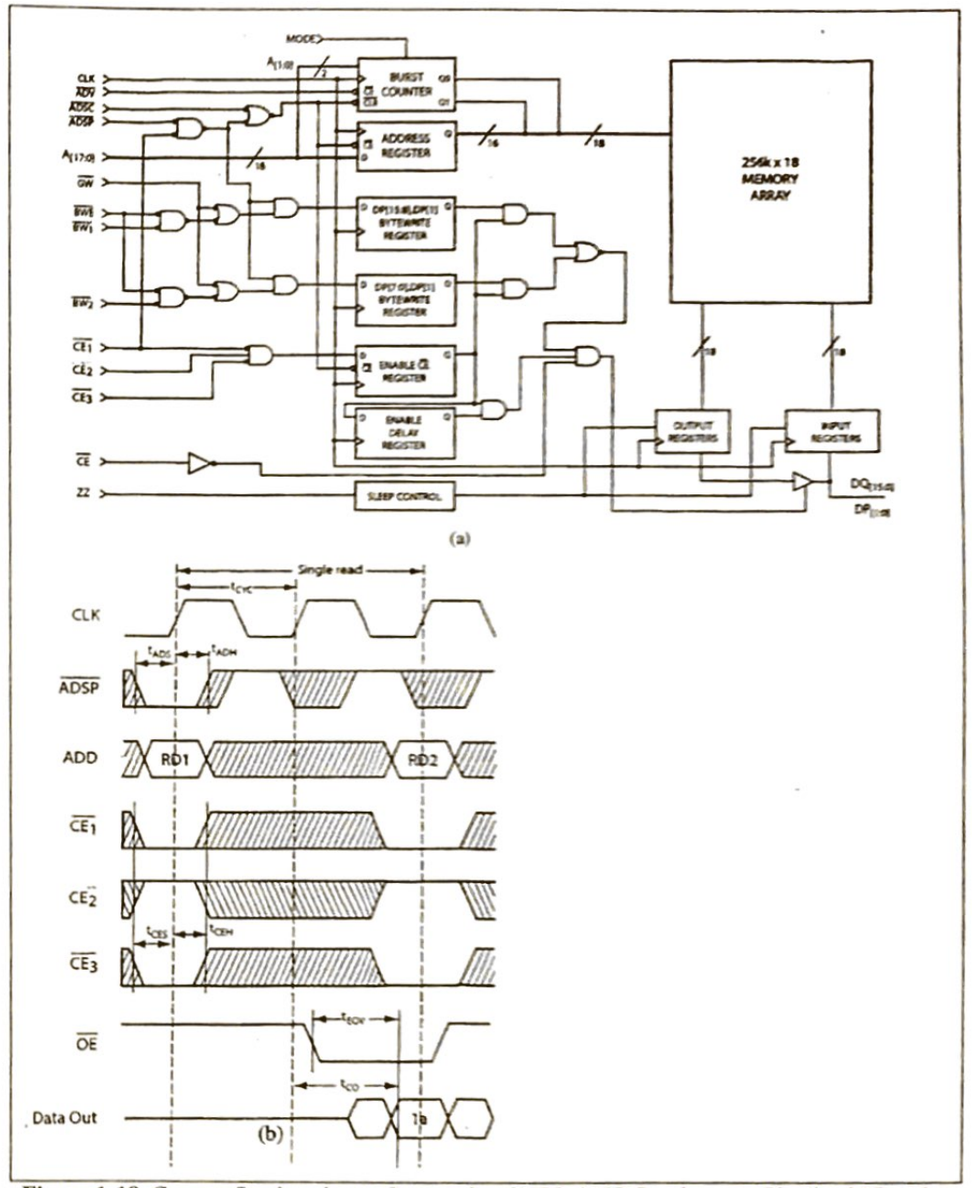


Figure 1-18. Cypress Semiconductor Corporation CY7C1327B Synchronous Pipelined SRAM. (a) Internal architecture. (b) Simplified timing diagram.

Dynamic RAM Types – DRAM, FPM DRAM, SDRAM, DDR SDRAM

STANDARD DRAM

The SRAMs discussed in the preceding section are very fast or in other words, they have short access times. However, since they typically use six transistors to store each bit, they dissipate a considerable amount of power per bit. Both the chip "real-estate" per bit and the power dissipation per bit limit the number of bits that can be stored in an SRAM. Dynamic RAMs or DRAMs store each binary 1 as an electric charge and each binary 0 as no electric charge on a tiny capacitor. Since only a single transistor switch is required to read a value from the capacitor or write a value to it, DRAMs require much less chip real estate and power per bit than do SRAMs. Therefore, a DRAM chip can store many more bits, dissipate much less power, and cost less per bit than an SRAM device built with the same IC technology. One major disadvantage of DRAMs is that the charges stored on the tiny capacitors tend to "leak off." Therefore, the data stored in a DRAM must be *refreshed* every few milliseconds.

Figure 1-19a shows a block diagram for a basic DRAM and Figure 1-19b shows a simplified timing diagram for the device. The actual storage architecture in a DRAM is a matrix of rows and columns. For a read operation of the device in Figure 1-19a, a 10-bit row address selects one of the rows and a 10-bit column address selects one of the columns. The data bit from the intersection of the selected row and the selected column is transferred to the Data Out Register. Note that although a total of 20 address bits are required to select one of the 1,048,576 bits in the device, Figure 1-19a shows only 10 address lines entering the device. The trick here, of course, is that the 20 address bits are multiplexed into the device, 10 at a time on the 10 address inputs. The timing diagram in Figure 1-18b shows how this works.

An external controller sends out the row address to the DRAM and asserts the RAS# (Row Address Strobe) signal to latch the row address in the DRAM. After an appropriate time, the external controller then sends the column address to the DRAM and asserts the CAS# (Column Address Strobe) signal to latch the column address in the DRAM. After an access time, the addressed data bit appears on the Q output. The external controller for a basic DRAM such as this has three major tasks. It needs to multiplex the two halves of the address and supply the RAS# and CAS# strobes, refresh each row in the DRAM every few ms, and *arbitrate* the conflict that occurs if a microprocessor or some other system attempts to access the DRAM during a refresh cycle. Although DRAM devices need this external controller and are considerably slower (longer access time) than SRAMs, the low cost and low power dissipation of DRAMs have led to their being used for the main memory in most microcomputer systems. However, the fact that microprocessor speeds have evolved much faster than DRAM speeds has motivated a lot of research on ways to reduce the access times and increase the effective data rate of DRAMs. As a result there are several DRAM types. For reference, we will summarize here the main characteristics of a few that are important from an evolutionary standpoint and/or are used in a significant number of current systems.

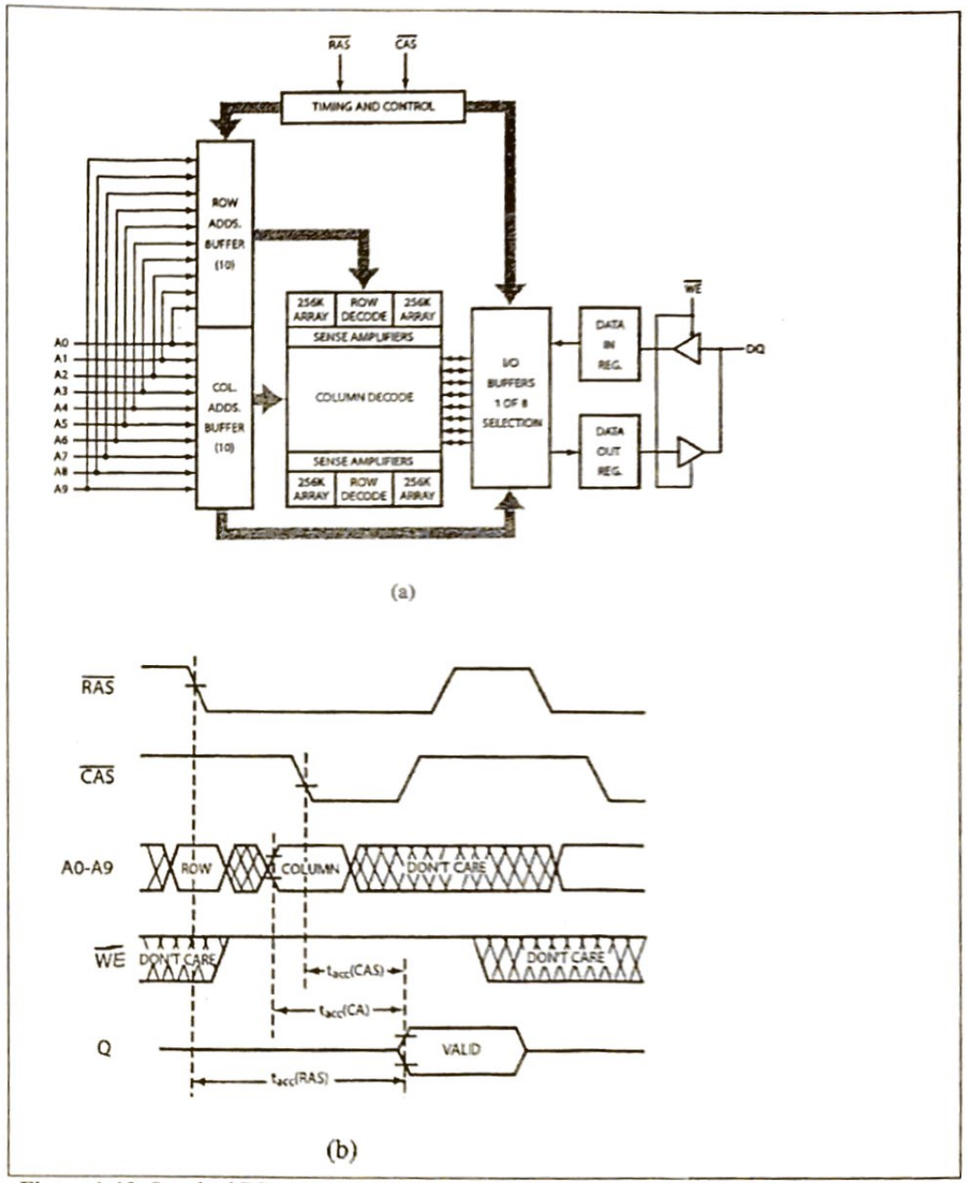


Figure 1-19. Standard DRAM. (a) Internal block diagram. (b) Simplified timing diagram.

FAST PAGE MODE DRAM (FPM DRAM)

The first major improvement in DRAMS was to add Fast Page Mode that works as follows. Since reading a bit from the DRAM array is partially destructive, the row that has been read must be restored or *precharged* before the next access is allowed. After completing one access, the controller must wait this additional precharge time before it starts another read or write access. This precharge time is about the same as the access time, so the effective data rate or *bandwidth* for the DRAM is about half of what it would be without the precharge time requirement. Fast page mode DRAMs do not require a precharge time for each read in a sequence of reads, as long as the reads are to the same row (page) in the DRAM. An external comparator in the DRAM controller determines if an access is to the same row (page) and if so, the controller does not send a new row address and RAS# signal. It just sends a new column address and CAS# to select the desired column(s) in the row. If the comparator in the controller detects that an access request is to a different row, the controller waits while circuitry in the DRAM automatically does the precharge. The controller then sends out the new row address and RAS#, followed by the new column address and CAS#. The key point is that Fast Page Mode devices do not require the RAS# part of the memory cycle or the precharge part of the cycle, as long as successive accesses are to columns in the same row (page). Since most microcomputer program memory reads are to sequential locations, the overall average time per read for a Fast Page Mode DRAM is much less than that for the original DRAM devices.

SYNCHRONOUS DRAM (SDRAM)

As stated in the earlier discussion of synchronous SRAMS, synchronizing the data transfer with the system clock or a clock derived from the system clock allows for a higher data transfer rate. Figure 1-20a shows a block diagram for a Micron Technology MT48LC16M16A2 Synchronous DRAM . Note the CLK input present on this device but not in the basic DRAM in Figure 1-19a . The second feature to note for the SDRAM is that the DRAM array is divided into four banks. The big advantage of this feature is that for many accesses, one bank can be precharged while another bank is being accessed. When a series of sequential accesses reaches the end of a row in one bank, the accesses can continue into another bank without waiting for a precharge of the accessed row in the first bank, if the next data in the sequence is stored in the other bank. If the next sequential data is stored in a different row in the bank being accessed, then a precharge will be required before the new row can be accessed.

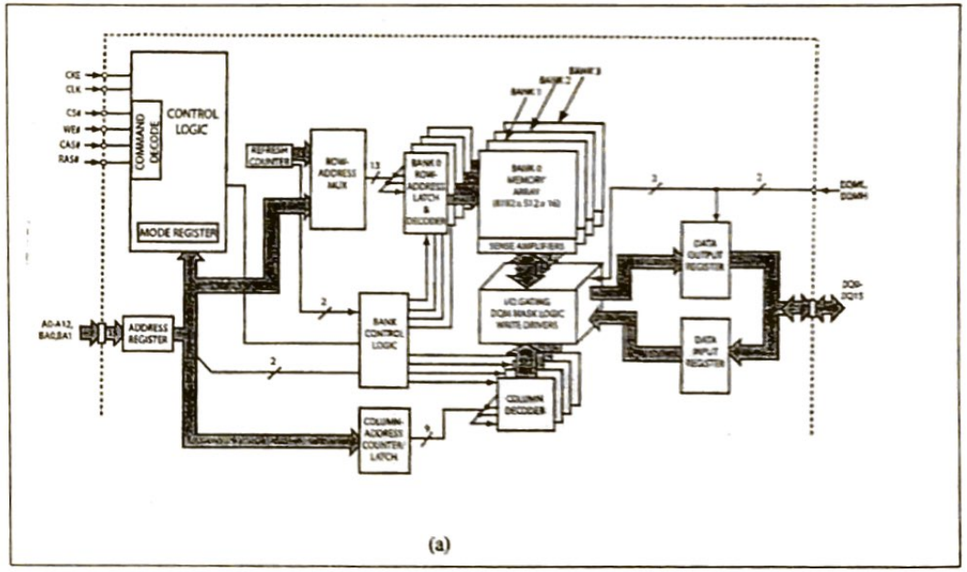
Now, we need to look at the timing diagram in Figure 1-20b in order to understand an additional improvement built into SDRAMs. The first section of the waveform shows that the memory controller *activates* a row and bank by sending a row address and a bank address, and then asserting RAS# low. For a burst *read* operation such as that in the second section of the waveform in Figure 1-19b, the memory controller sends the column address for the first desired column and pulses CAS# low after a RAS latency time of three clock cycles. The term *latency time* is the name given to the time you need to wait for the response to some action. In this case, it is the time you must wait for the device to accept the row/bank address and RAS# before applying the column address and asserting CAS#. After receiving the CAS#, the SDRAM then outputs a pre-programmed number of data words without requiring an additional column address and CAS# signal for each output word. This is possible because, as shown in Figure 1-20a, the SDRAM contains a column address counter/latch that holds the first column address sent in to

start the read operation, and then automatically increments the column address for successive words in the burst. For a burst read operation or a burst write operation, this allows a data word to be read out from or written to the SDRAM on the rising clock edge of every memory clock cycle after the initial CAS latency time of typically two or three clock cycles. For burst accesses then this is much more efficient than the timing of the basic DRAM shown in Figure 1-18b. However, for non-sequential accesses to individual words, each access requires an Activate with RAS# and a Read with CAS#. As you will see later, in a microcomputer system most memory accesses are burst oriented, so the performance gain provided by SDRAMs is significant.

The question about burst access that might occur to you at this point is, "How does the SDRAM know how many data words to burst out?" The answer is that, the number of words desired in a burst is written to the Mode Register in the SDRAM as part of the system initialization process or dynamically during different memory accesses.

To further improve efficiency, SDRAMs allow one bank to be opened while another is being read from or written to. Also, SDRAMs contain a built-in Refresh Counter to simplify the required refresh operation.

In summary, the SDRAM features that significantly improve memory performance are: synchronous data transfer, burst transfer, programmable burst length with a column address and CAS# only required for the first access, and multiple banks that in many cases allow the precharge of one bank during access to another bank. In the next section we discuss a newer type of SDRAM with additional performance features.



(a)

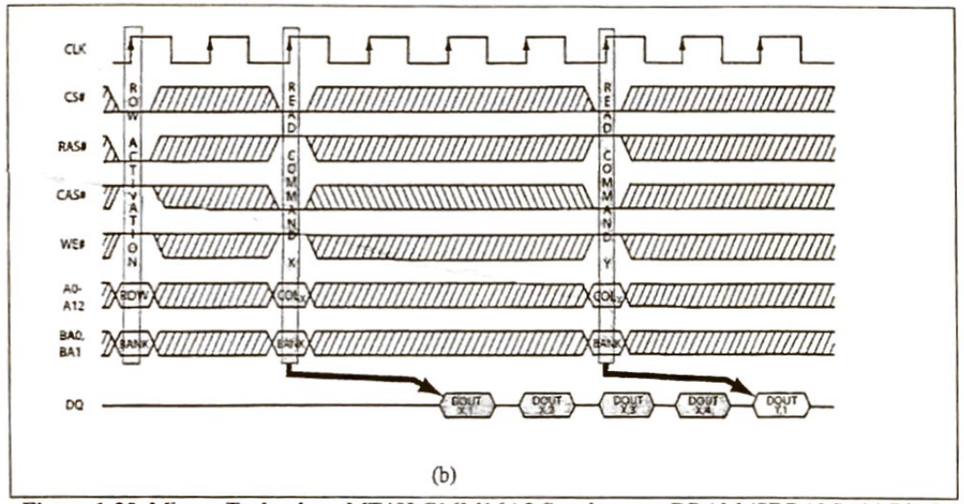


Figure 1-20. Micron Technology MT48LC16M16A2 Synchronous DRAM (SDRAM). (a) Block diagram. (b) Activate & burst read timing diagram.

DOUBLE DATA RATE SDRAM (DDR SDRAM)

In the preceding section we showed that, during a burst read operation, a memory controller can read a data word from an SDRAM every clock cycle after the CAS latency time of three cycles for the first word. The minimum clock period for the SDRAM is limited by the access time for the DRAM array in the device. The SDRAM cannot be clocked faster than the data can be read from the storage cells in the device and this limits the memory bandwidth or, in other words, the rate at which data can be supplied to a requesting system. However, it was found that a word twice as wide as the desired word can be read from the storage cells in the DRAM array in the same time as a word of the desired output width can be read. In a basic Double Data Rate SDRAM, the internal circuitry does a *2n prefetch*. In other words, if the device has 8 data outputs, the internal circuitry will read 16 bits from a row in the array, instead of just the 8 bits to be output. The two halves of this 2n word are then multiplexed onto the data bus, one after the other. Since two bytes are available for sending in the same time as one would be in a device without 2n prefetch, data bytes can be sent down the bus at a rate twice as fast at the normal bus transfer rate. Also, instead of using the memory system clock to transfer data, as in standard SDRAMs, separate *strobe* signals are created from the clock for each read access to synchronize the data transfer. As shown by the timing waveforms in Figure 1-21, the strobe signals are timed so that effectively, one of the 2n data words is transferred on the rising edge of the clock signal and one is transferred on the falling edge of the clock, thus the name Double Data Rate. The strobe signals are created and output by the device supplying the data for a particular operation. For a memory write operation, the memory controller creates the strobe signals and sends them to the DDR SDRAM along with the data to be written to the SDRAM. For a read operation, the DDR SDRAM generates the strobe signals and sends them to the memory controller along with the requested data. Sending closely timed strobe signals along with the data and using these

strobe signals to latch the data in the receiving systems is called *source synchronous data transfer*. The close timing of the strobes with the data in source synchronous data transfer allows for much higher data rates than are possible with standard SDRAM systems that just use the memory system clock for data transfers.

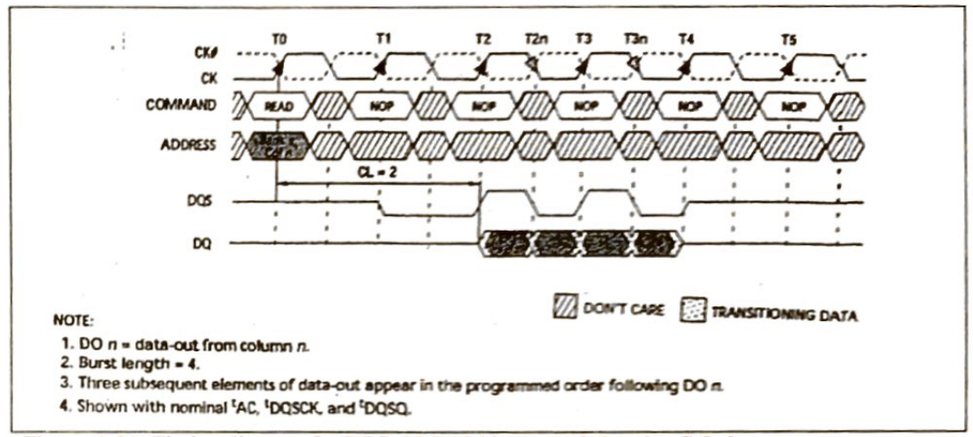


Figure 1-21. Timing diagram for DDR SDRAM burst read showing DQ data output on effectively both edges of Data Strobe, DQS, to give Double Data Rate.

The reason is that, in a standard SDRAM system, the memory system clock signal arrives at different devices at slightly different times due to the different signal path lengths to the different devices. This difference in signal arrival times is called *timing skew*. Because of timing skew, a memory system using the memory system clock to clock the receiving latches needs to allow a greater timing margin or slack time in order to avoid violating the setup times and hold times of the latches that are receiving the data. To provide this greater timing margin, the memory must be clocked at a lower frequency. This decreases the memory bandwidth (data transfer rate) and thus decreases overall performance. With DDR SDRAM, the strobe signals take the same signal path as the data signals, so there is minimal timing skew and data can be transferred at a higher frequency.

DDR memory technology has continued to evolve in speed and capacity past the first generation DDR. Succeeding generations are identified as DDR2, DDR3, DDR4, DDR5, etc. Often, several of the basic DDR devices are mounted on small printed circuit boards called *Dual In-line Memory Modules* or *DIMMs* , such as the DDR3 DIMM shown in Figure 1-22 These DIMMs are plugged into slots in a PC or laptop to provide a large amount of DRAM.

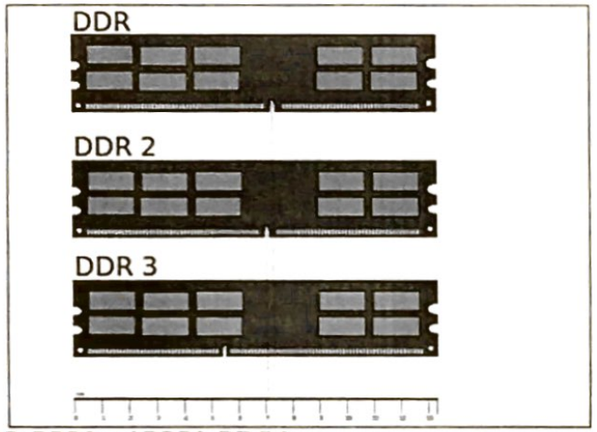


Figure 1-22. DDR, DDR2 and DDR3 DIMMs

In summary, the key points to remember about DDR SDRAM are the 2n or larger prefetch and the source synchronous data transfer that uses tightly timed strobe signals to transfer data at twice the rate of the memory system clock.

In the next chapter we will show and discuss how the basic digital devices and the memory devices we have discussed in this chapter are used to build microprocessors and microcomputers. In a later chapter we dig much more deeply into the interfacing of these memory devices.

References and Further Sources of Information

(1) http://www.intel.com - Data sheets for Flash EEPROMs
(2) http://www.cypress.com - Data sheets for SRAMs and the Warp 6 Design automation tools to implement combinational logic and State machines in PLDs
(3) http://www.micron.com - Data sheets for DRAMs

Checklist of Important Terms and Concepts

Active high signal, active low signal
Three-state output
DeMorgan Equivalent logic symbols
PLAs, PALs, PROMs
Time Division Multiplexing
Multiplexer (MUX), Demultiplexer (DMUX)
Decoder, encoder
D Latch operation and timing
D Flip-flop operation and timing
Level-triggered device
Edge-triggered device
Setup time hold time, minimum pulse width

Metastability
Register, shift register
Slack time
Presettable/Loadable counter
Next State Decoder, Output Decoder
Moore type output signal
Mealy type output signal
State diagram
MROM, EPROM, EEPROM
Flash EEPROM, Synchronous Flash EEPROM
SRAM, Synchronous SRAM
DRAM, FPM DRAM, Synchronous DRAM
Burst mode transfer
RAS Latency time, CAS Latency time
DDR DRAM
2n-8n Prefetch in SDRAM
Clock Timing skew
Source synchronous data transfer

Review Questions and Problems

1. Write the decimal equivalents for 2^{10}, 2^{24}, and 2^{32}.

2. Convert the decimal numbers 22, 76, and 500 to their binary equivalents

3. Convert the binary numbers 10 0111 and 1101 0001 to their decimal equivalents.

4. Express the decimal numbers +26 and –47 in 8-bit binary, two's complement sign and magnitude form.

5. In this chapter we showed how to represent integer numbers in two's complement sign and magnitude form. For use in a microcomputer, numbers such as 21.435, that have fractional parts as well as integer parts, are represented in single precision or double precision IEEE 754 Floating point format. These formats are now called binary32 and binary64 respectively. Most of the higher level microprocessors now have specialized coprocessors that work directly with numbers in these formats. Do an internet search to determine how numbers are represented in these formats and briefly describe the fields in the binary32 format.

6. Convert the binary numbers 0110 1100 0101 and 1100 0011 0101 0001 to their hexadecimal equivalents.

7. Convert the hexadecimal numbers D3H and 47FFH to their binary equivalents.

8. Write the BCD representations for the decimal numbers 86 and 136.

9. Perform the indicated operations on the following numbers

a. 0x3A + 0x92
b. 0x17A – 0x4C

10. We deliberately omitted a discussion of parity in this chapter. Use a digital design book, the Internet, or some other source to find the meaning of the term parity and explain briefly how it is used to detect single-bit errors in transmitted ASCII data words.

11. Use the bubbled input and bubbled output rules to help you quickly write the logic expression for the Y output of the circuit in Figure 1-23.

12. Figure 1-17a shows the block diagram for a Synchronous RAM. Use the rules we gave you for analyzing multi-level circuits with bubbles to help you write the logic expression for the signals that assert the CLR# signal in the Burst Counter.

13. Briefly describe the major advantage of time division multiplexing and the major disadvantage(s).

14. Given the clock and data waveforms in Figure 1-24, show the results that these waveforms would produce on the output of a D-latch and on the output of a D flip-flop.

15. Assuming that the multiplexers in Figure 1-11c are in the up position so the circuit functions as a simple shift register, calculate the maximum clock frequency assuming that the t_{PD} of the flip flops is 2ns, the t_{SU} of the flip-flops is 1ns, and the t_{PD} of the multiplexers is 1ns.

16. For a hypothetical state machine you have just designed, the flip-flops have a propagation delay of 4ns and a setup time of 1ns. The output decoder has a propagation delay of 2ns and the next state decoder has a propagation delay of 2ns. Calculate the maximum clock frequency for this state machine and describe what will likely happen if the machine is clocked at a higher frequency than this.

17. Describe what is required to convert an asynchronous memory device to a synchronous memory device and bri20ly describe the advantages that a synchronous SRAM has over an asynchronous SRAM with respect to data transfer rates.

18. List and briefly describe the three tasks that must be done by the memory controller or DRAM internal circuitry for a block of DRAM.

19. Briefly describe the two major techniques used by DDR DRAMs to achieve a data transfer rate that is two times the rate that would be possible without the use of these techniques.

20. As a design engineer and for some of the work in later chapters of this book, you will be getting many of the data sheets you need from the Internet. The task here will give you a first chance to download and look at the complete data sheet for a real device that we will refer to in later chapters. In an earlier section of the chapter we discussed the address access time, t_{ACC}, and the chip enable access time t_{CE} for asynchronous memory devices. In a later discussion of Flash EEPROMs, we mentioned the Intel Strata Flash 28F128J3A device. For this question we want you to find the timing parameters equivalent to t_{ACC} and t_{CE} for the

fastest version of Intel Strata Flash 28F128J3A, 128 Megabit device. Do an Internet search for the "28F128J3A data sheet." (Note that Intel no longer produces the device but others do and the data sheet is easily available.) When the search list appears, look for an entry that points to the 28F128J3A data sheet and click on it. Download the PDF data sheet, which should be about 68 pages. Scroll through the data sheet to get a feeling for the type of information given in this type data sheet. Then go to the AC Characteristics section and study the table of timing parameters for a Read Operation until you find the two times that are equivalent to t_{ACC} and t_{CE} but in this data sheet have different names.

21. On the Internet, go to www.micron.com and search for the data sheet for the MT41K256M16HA-125 DDR3 DRAM device. Download and read through the first couple of pages of the PDF data sheet for it. Briefly describe and give the benefit of at least 2 features that are improvements over those of the basic four bank device in Figure 1-19a and the DDR 2n prefetch scheme described in the DDR section of the chapter.

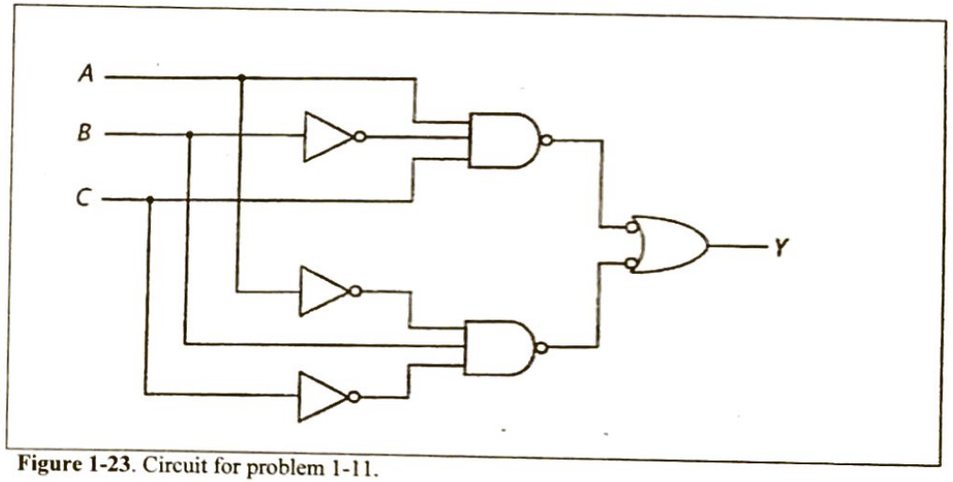


Figure 1-23. Circuit for problem 1-11.

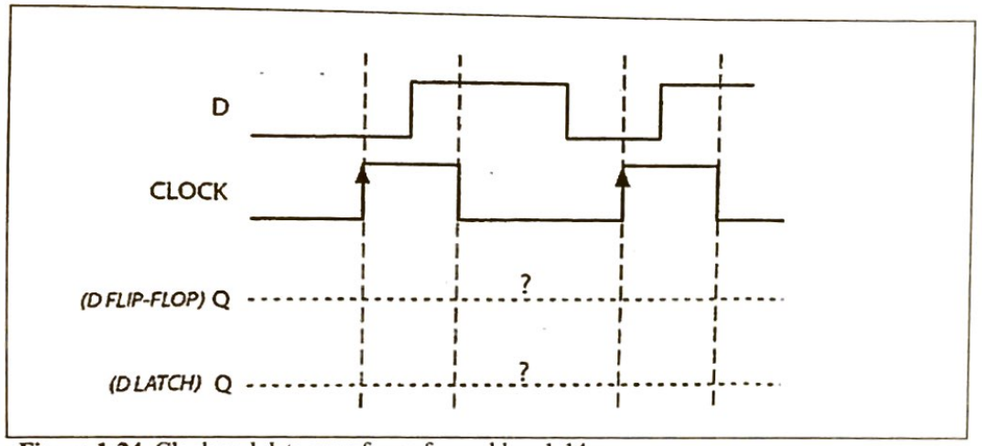


Figure 1-24. Clock and data waveforms for problem 1-14.

CHAPTER 2 – MICROPROCESSOR ARCHITECTURES AND EVOLUTION

The discussions of microprocessor and microcomputer architecture in this chapter are not intended to function as an in-depth course in computer architecture. They are intended to refresh or teach you architectural concepts at a level needed to understand the basic features and operation of current microprocessors. We will build on these foundations in later chapters where we teach you how to write efficient programs for current microprocessors, and how to design the interfaces for a variety of microprocessor peripheral devices. To provide an historical context for the current microprocessors, we also give a brief overview microprocessor evolution in this chapter.

Objectives

At the conclusion of this chapter, you should be able to:

1. Draw a block diagram showing how a microprocessor can be connected with memory and I/O devices to build a microcomputer and describe the basic operation of a microcomputer.
2. Draw a block diagram showing how an ALU, multiplexers, registers, and a state machine can be connected to implement the data path and controller for a simple microprocessor, and describe its operation.
3. Explain the operation and advantages of a pipelined microprocessor using an architectural diagram and a pipeline diagram.
4. Explain how resource duplication, instruction scheduling, caches, write buffers, Branch Target Buffers (BTBs), and branch prediction improve the performance of a pipelined microprocessor
5. Describe the differences between a CISC processor and a RISC processor and explain the advantages and disadvantages of each.
6. Describe the major features of microprocessors designed for embedded systems and those designed for desktop systems.

Architecture and Operation of a Basic Microcomputer

Figure 2-1a shows a basic block diagram for a very simple microprocessor-based microcomputer system. The heart of the system is the microprocessor, which is identified as the Central Processing Unit or CPU in classic computer block diagrams such as those you might have seen in a high-level language programming class. Another key component is the memory, which is used to hold binary-coded program instructions and data. The binary form of program instructions stored in memory is commonly referred to as *machine code*, because these binary bits tell the machine (microprocessor) what action to perform.

Other key components in the microcomputer are the Input ports that are used to read data into the microprocessor from external devices, and the Output ports that are used to write data to external devices. Together these are often referred to as I/O ports. Physically, an I/O port device may be just a group of parallel latches that are enabled by a read or write control signal, or it may be a very complex device such as an Ethernet interface controller. The microprocessor memory and I/O ports are connected with three buses: the address bus, the data bus, and the control bus.

To give an overall view of the bus and other actions that need to be performed to execute a simple program, we will quickly walk though the sequence of actions that would be taken by an early microprocessor system to do this. As we show later in the chapter, current microprocessors overlap many of these operations and therefore operate much more efficiently.

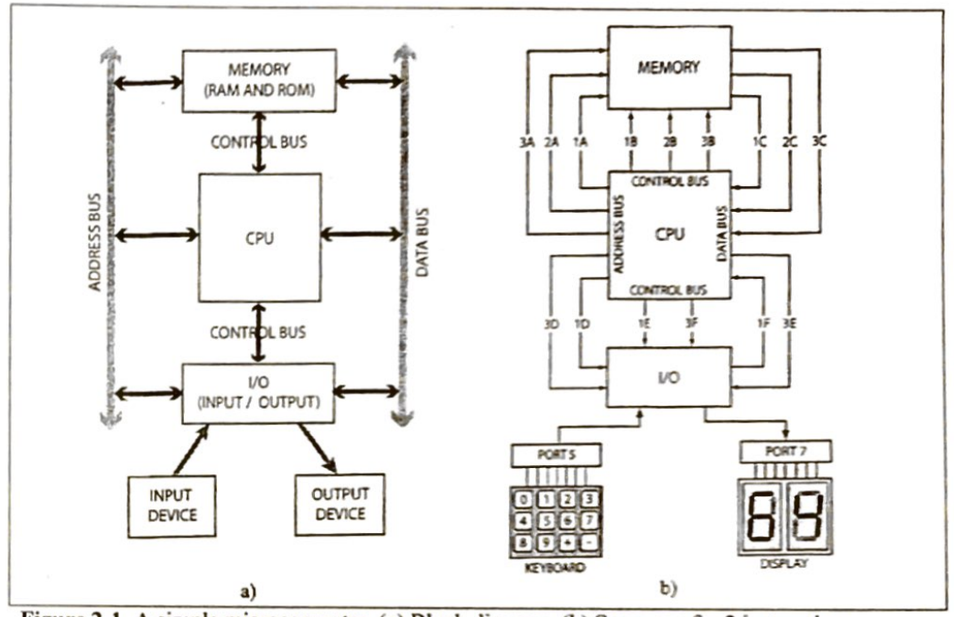


Figure 2-1. A simple microcomputer. (a) Block diagram. (b) Sequence for 3 instruction program.

The labeled arrows in Figure 2-1b show the sequence of actions for a simple, three instruction program that reads a value from microcomputer port 5, adds 7 to the value, and writes the result to port 7. Here is a sequential list of the actions to help you follow the flow. Note that "µP" in these steps stands for microprocessor.

1A – µP sends out memory address for Input instruction on Address bus.
1B – µP sends out memory read control signal on control bus.
1C – Machine code for "Input" instruction returned to µP on data bus.
1D – µP decodes instruction and sends out port address (05) on address bus.
1E – µP sends out port read control signal on control bus.
1F – Input port device returns byte from port 05 to a register in the µP on data bus.
2A – µP sends out memory address for Add instruction on the address bus.
2B – µP sends out control signal for memory read on control bus.
2C – Memory returns machine code for the "Add" instruction on data bus. The processor internally adds 7 to the data word read from the port and holds the result in a register.
3A – µP sends out the memory address for the Output instruction on the Address bus.

3B – µP sends out a memory read signal on the control bus.
3C – Memory returns machine code for "Output" instruction on the data bus.
3D – µP decodes instruction and sends output port address (07) on address bus.
3E – µP sends sum from register to output port on the data bus.
3F – µP sends out port write control signal on control bus.

To summarize the operation of a microprocessor-based microcomputer then, the machine code for the program instructions are stored in memory. For this early type of microprocessor, the microprocessor fetches an instruction, decodes the instruction, executes the instruction, and then fetches the next instruction, etc. In the next section we show you how you design a very simple microprocessor to perform these steps and then in the following sections we discuss additional architectural features that greatly improve the efficiency of a microprocessor. For these discussions, we take a bottom-up approach because we have found that this approach best helps people understand hardware design principles and, in later chapters, software optimization techniques.

Designing a Single Instruction Microprocessor

ADDING AND SUBTRACTING WITH GATES

Figure 2-2a shows the full-adder truth table for the addition of two binary bits, A and B, plus a Carry In from a previous bit position. Figure 2-2b shows an example of binary "pencil" addition using these rules. As shown in Figure 2-2a, the simplified logic expression for the Sum bit, S, is S = A EXOR B EXOR C_{IN} (symbolically $S = A \oplus B \oplus C_{IN}$), and the simplified logic expression for the carry out bit from one column to the next is C_{OUT} = A AND B OR C_{IN} (A EXOR B) (symbolically $C_{OUT} = A \cdot B + C_{IN} (A \oplus B)$). Since these are just simple combinational logic functions, they can be implemented with basic logic gates as shown in Figure 2-2c. This circuit could be duplicated and *cascaded* to create a circuit that adds multiple-bit binary words. The term cascading here means that the C_{OUT} from one bit position is connected to the C_{IN} of the adder for the next most significant bit. This is exactly what you do with a carry when you do "pencil" addition as in Figure 2-2b. The key point here is that it is very easy to build a combinational logic circuit that performs binary addition. It is also very easy to build a combinational logic circuit that performs binary subtraction, because the simplified expression for the Difference bit for two binary bits and a Borrow In is: Difference = $A \oplus B \oplus B_{IN}$ and the expression for the Borrow Out bit is: $B_{OUT} = A\# \cdot B + B_{IN} (A \oplus B)\#$. In the next section we show a combinational logic device that can perform a wide variety of arithmetic and logic operations.

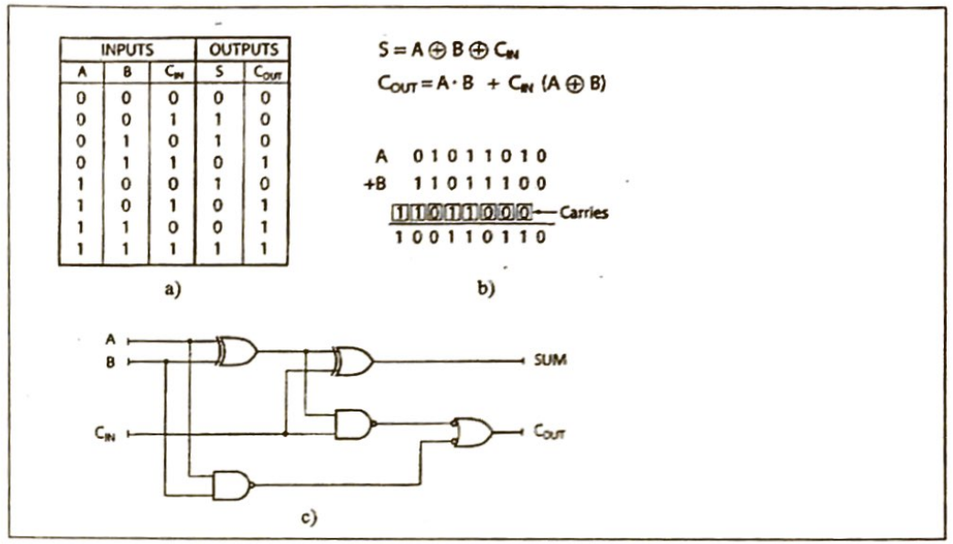
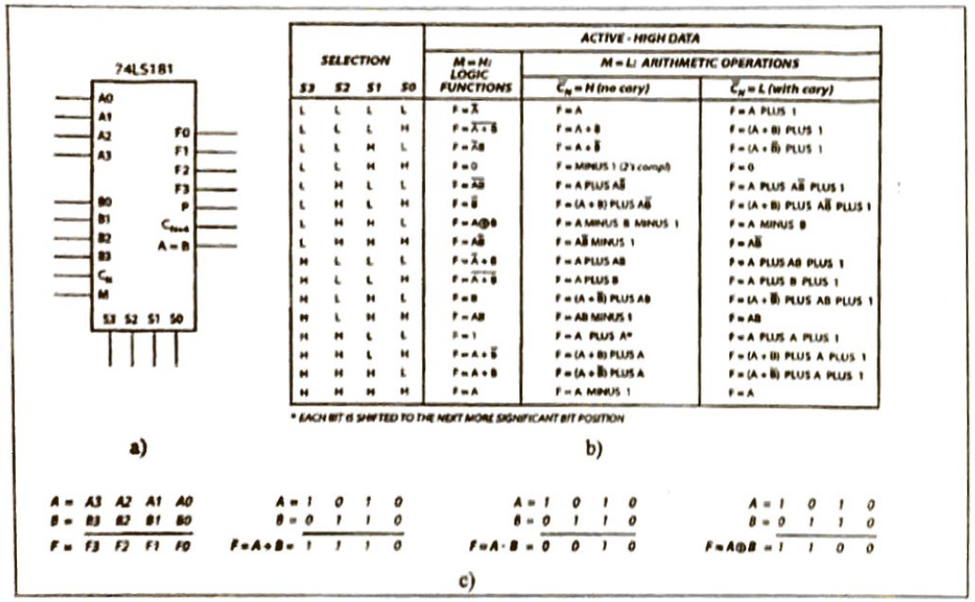


Figure 2-2. Binary addition. (a) Full adder truth table. (b) Example of binary addition. (c) Circuit to add 1-bit binary numbers plus carry.

AN ARITHMETIC LOGIC UNIT (ALU)

Figure 2-3a shows the schematic symbol for a combinational logic circuit called an Arithmetic Logic Unit (ALU). An ALU can perform addition, subtraction, or a wide variety of other arithmetic and logic functions on two N-bit binary words. For the ALU in Figure 2-3a, the A and B data inputs are each 4 bits wide. The function performed on the two binary words applied to the A and B inputs is determined by the 4-bit code applied to the Select inputs, S3-S0, and the 1-bit code applied to the Mode Input. Note that if the Mode input is High, the device performs one of the 16 possible logic functions on the input words as shown in Figure 2-3b. For example, if the Mode input is high and the S inputs have the values HHHL, each bit in the A input will be logically ORed with the corresponding bit on the B input, as if the bits were connected to the inputs of a 2-input OR gate. The result will be output on the corresponding numbered bit of the F outputs. Figure 2-3c shows some examples of performing logic operations on 4-bit binary words.



SELECTION				ACTIVE - HIGH DATA		
				M = H: LOGIC FUNCTIONS	M = L: ARITHMETIC OPERATIONS	
S3	S2	S1	S0		C_n = H (no carry)	C_n = L (with carry)
L	L	L	L	$F = \overline{A}$	F = A	F = A PLUS 1
L	L	L	H	$F = \overline{A+B}$	F = A + B	F = (A + B) PLUS 1
L	L	H	L	$F = \overline{A}B$	$F = A + \overline{B}$	$F = (A + \overline{B})$ PLUS 1
L	L	H	H	F = 0	F = MINUS 1 (2's comp)	F = 0
L	H	L	L	$F = \overline{AB}$	$F = A \text{ PLUS } A\overline{B}$	$F = A \text{ PLUS } A\overline{B}$ PLUS 1
L	H	L	H	$F = \overline{B}$	$F = (A + B) \text{ PLUS } A\overline{B}$	$F = (A + B) \text{ PLUS } A\overline{B}$ PLUS 1
L	H	H	L	$F = A \oplus B$	F = A MINUS B MINUS 1	F = A MINUS B
L	H	H	H	$F = A\overline{B}$	$F = A\overline{B}$ MINUS 1	$F = A\overline{B}$
H	L	L	L	$F = \overline{A} + B$	F = A PLUS AB	F = A PLUS AB PLUS 1
H	L	L	H	$F = \overline{A+B}$	F = A PLUS B	F = A PLUS B PLUS 1
H	L	H	L	F = B	$F = (A + \overline{B}) \text{ PLUS } AB$	$F = (A + \overline{B}) \text{ PLUS } AB \text{ PLUS } 1$
H	L	H	H	F = AB	F = AB MINUS 1	F = AB
H	H	L	L	F = 1	F = A PLUS A*	F = A PLUS A PLUS 1
H	H	L	H	$F = A + \overline{B}$	$F = (A + B) \text{ PLUS } A$	$F = (A + B) \text{ PLUS A PLUS 1}$
H	H	H	L	$F = A + B$	$F = (A + \overline{B}) \text{ PLUS A}$	$F = (A + \overline{B}) \text{ PLUS A PLUS 1}$
H	H	H	H	F = A	F = A MINUS 1	F = A

* EACH BIT IS SHIFTED TO THE NEXT MORE SIGNIFICANT BIT POSITION

A =	A3	A2	A1	A0	A =	1	0	1	0	A =	1	0	1	0	A = 1 0 1 0
B =	B3	B2	B1	B0	B =	0	1	1	0	B =	0	1	1	0	B = 0 1 1 0
F =	F3	F2	F1	F0	F=A+B=	1	1	1	0	F=A·B =	0	0	1	0	F=A⊕B = 1 1 0 0

Figure 2-3. An Arithmetic Logic Unit (ALU). (a) Schematic symbol. (b) Truth table. (c) Examples.

If the Mode input is low, the ALU performs one of 16 possible arithmetic functions on the two input words as shown in Figure 2-3b and outputs the result on the four F outputs, F3-F0, and on the Carry/Borrow output, C_{N+4}. (The C_{N+4}, G and P signals can be used to cascade two or more of these ALUs to perform arithmetic or logic functions on larger binary words.)

DESIGNING A SINGLE INSTRUCTION MICROPROCESSOR WITH DATA PATH AND CONTROLLER

An ALU is an important part of a microprocessor. However, in order to perform a sequence of operations as is usually required, a basic microprocessor also needs to have registers to hold operands, multiplexers to select desired operands, and a state machine to step the microprocessor through the desired sequence of operations. As a first example of basic microprocessor architecture, we will show and describe the operation of a microprocessor that executes a single sequence of operations or in other words, a single microprocessor instruction. Specifically, our very simple microprocessor will execute an instruction that adds two Binary-Coded Decimal (BCD) digits and gives a correct BCD result. To start the discussion, here's a review of BCD addition.

BCD, as discussed in Chapter 1, is a four-bit binary code for representing the decimal digits 0-9. The BCD values for the decimal digits 0-9 are just the binary values 0000-1001. The other six possible binary values, 1010-1111, are not used and are therefore not valid BCD values.

Figure 2-4 shows some examples of BCD addition. We have separated the bits in groups of 4 to show that each group of four bits represents a decimal digit. The initial addition of BCD values is done using the same rules as those for binary addition. In the first addition example in Figure 2-4a, the addition of the least significant four bits gives a sum of 1000 binary or 8 as expected and, since there is no carry from the bits for this digit, the addition of 3 plus 2 in the next most significant digit gives 0101 or 5 as expected.

```
                BCD
  35        0011 0101
 +23       +0010 0011
 ---       ----------
  58        0101 1000
                a)

                BCD
   7            0111
 + 5          +0101
 ---          -----
  12            1100   INCORRECT BCD
              +0110    ADD 6
              ---------
            0001 0010  CORRECT BCD 12
                b)

                BCD
   9            1001
 + 8          +1000
 ---          -----
  17        0001 0001  INCORRECT BCD
              +0110    ADD 6
              ---------
            0001 0111  CORRECT BCD 17
                c)
```

Figure 2-4. Binary Coded Decimal (BCD) addition examples. (a) No correction needed. (b) Correction needed because result is greater than 1001. (c) Correction needed because of carry from bit 3 to next digit.

In the second example in Figure 2-4b, we add 0111 or 7 and 0101 or 5 and get a sum of 1100. This is the correct binary result equal to decimal 12 but, since 1100 is not a valid BCD value, the answer is not the correct BCD result. To correct the result, you add 0110 or 6, to force a carry into the next BCD digit. This second addition gives a result of 0001 0010, which is the correct BCD representation of decimal 12.

The third example in Figure 2-4c shows another case where the correction factor of 0110 must be added to give the correct BCD result. In this case, the fact that the first addition generates a carry into the second BCD digit tells you that the correction factor is needed. The first addition produces 0001 0001, which is the correct binary result but not the correct BCD result. Adding 0110 to the least significant BCD digit gives 0001 0111, which is the correct BCD representation for decimal 17.

Based on these examples, you can summarize the simple sequence of operations for adding two BCD digits as follows:

Get two BCD digits, A and B, from storage
Add A + B + C_{IN}
If SUM > 1001 OR C_{OUT} = 1
    Add 0110 to SUM
Else
    Add 0000 to SUM (does nothing but keeps same number of steps for all cases)

Data Path Components

Now let's think about what you need to add to the basic ALU in Figure 2-3 to step through this sequence. First, you need registers to hold the A and B digits fetched from storage for the ALU, and you need a register to hold the initial sum so it is available for the second addition. Second, you need a multiplexer on the A inputs of the ALU to select the A operand for the first addition and select the intermediate sum for the second addition. You need a multiplexer on the B inputs to select the B operand during the first addition and either a correction factor of 0000 or a correction factor of 0110 to be added to the temporary sum during the correction step in the sequence. Next, you need a combinational logic circuit to evaluate the If clause or, in other words, to determine if the first addition produced a result larger than 1001 or produced a C_{OUT}. Finally, you need a state machine to produce the required control signals for each step in the sequence. Figure 2-5a shows one way you can connect all these pieces together to execute this sequence of BCD addition operations.

First note the register file in Figure 2-5a. A register file is simply an array of registers used to hold multiple operands. A specific register in a register file is read by applying the register number for the desired register as an address and asserting a Read signal. Likewise a value is written to a desired register by applying the register number and asserting a Write signal. Most register files are *multi-ported*, which means that two or more registers in the file can be accessed at the same time. In fact, for reasons that we will explain later, it is quite common for a microprocessor register file to have at least two Read ports and one Write port.

The A and B registers in Figure 2-5a hold the A and B digit values fetched from the register file. Multiplexer A (MUX A) selects either the A operand for the first addition, or the temporary sum from the ALU Output register for the second addition. MUX B selects either the B operand for the first addition or the desired correction factor for the second addition. Also, a status register is added to hold the C_{OUT} signal and other result status signals such as Sign, Zero, and Overflow. (An ALU is simply combinational logic, so if this register were not put in the C_{OUT}-to-C_{IN} loop, a race condition and likely oscillation could exist in the loop.) Status bits such as Carry, Zero, Overflow, and Sign stored in a register in this way are often called flags.

The ALU, registers, and multiplexers together form what is called the *data path* part of the microprocessor because the data flows though these. The *controller* in Figure 2-5a produces the register and multiplexer enable signals needed to move operands through the data path. To help with our later discussions of current microprocessor architecture, we will briefly describe how you design a controller for a simple data path circuit such as this.

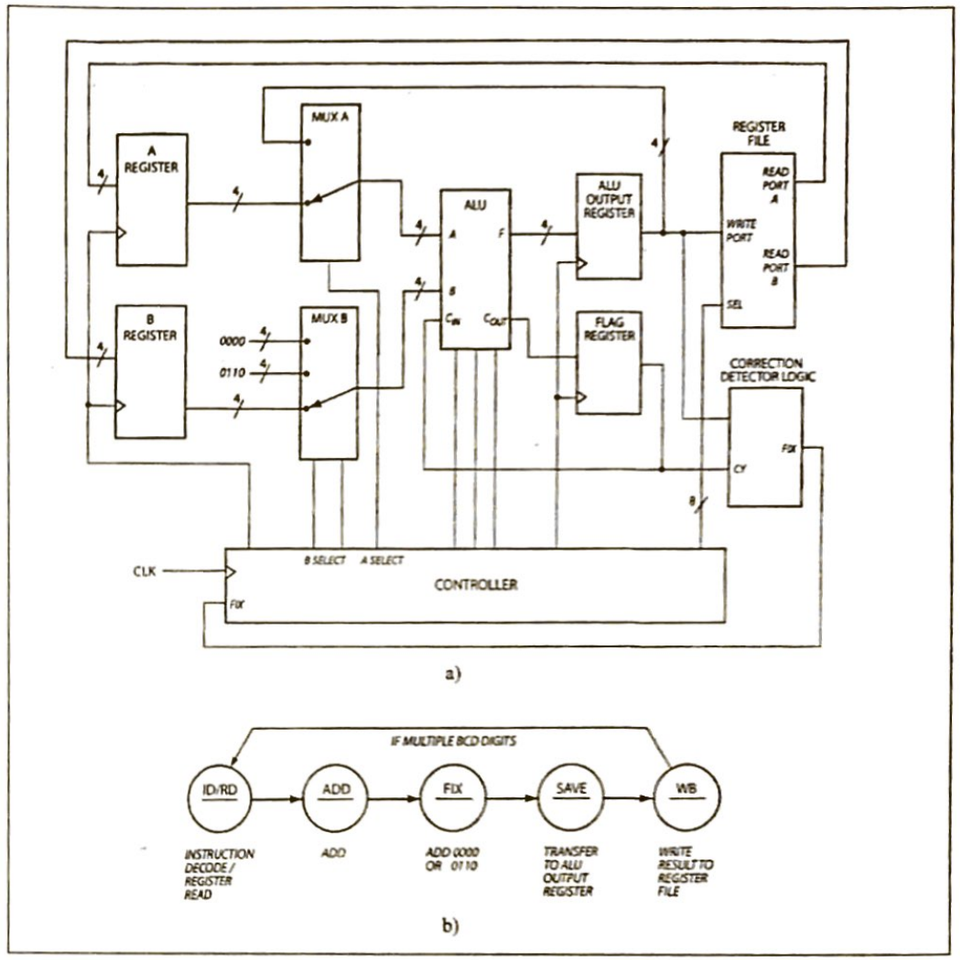


Figure 2-5. A "microprocessor" that adds two BCD numbers. (a) Circuit. (b) States for controller.

Designing A Data Path Controller

After you have drawn the block diagram of the hardware as in Figure 2-5a, the next step is to mentally step the hardware through the desired sequence of operations and write down the required actions in each step. For the circuit in Figure 2-5a, a reasonable sequence of steps and actions in each step might be as follows:

1. Select desired operand registers in register file for operand Read
2. Transfer the A operand from register file to the A operand register
   Transfer the B operand from register file to the B operand register
   Set MUX A to select the A operand for ALU
   Set MUX B to select the B operand for ALU
   Apply the code for ADD to ALU select inputs
3. Transfer first sum to the output of ALU Output register
   Transfer C_{OUT} and other flags to the output of the Status register
   Set MUX A to select the temporary sum for the ALU A input
   Set MUX B to select correction factor of 0110 or 0000, based on first result
4. Transfer final sum to the ALU output register.
   Select register in register file to write final result
   (Select registers in register file for read of next BCD digits, if multi-digit BCD numbers are being processed.)
5. Transfer (Write) result to selected register in the register file

Each of these blocks of steps and actions then becomes a state in your controller state machine. Figure 2-5b shows a traditional "bubble diagram" for the states represented by the list above.

The next step in the design process is to determine the signal assertion level for each control signal in each state. You also decide for each signal whether it should be a Moore-type signal or a Mealy-type signal. (We usually write these signal names and assertion levels next to corresponding action in the action list, rather than trying to put them in a bubble diagram.) For the circuit in Figure 2-5a, the enable signals for the registers will be Moore-type signals because we want them to always be asserted in particular states. The select signals for MUX B will be Mealy-type signals because they depend on the state AND on the evaluation of the "If SUM > 1001 OR C_{OUT} = 1" clause by some combinational logic. If the evaluation is true, then in state 3, the MUX B select signals should be asserted to select a correction factor of 0110; if the evaluation is false, the MUX B select inputs should be asserted to select a correction factor of 0000.

Once you have defined all the states and signals for the controller, the next step is to describe it in an HDL such as Verilog, compile the HDL description, simulate the design, and map the design into hardware as described in the previous chapter. Some review questions at the end of the chapter will give you a chance to think more about the "single instruction" microprocessor in Figure 2-5a.

In the meantime, we need to talk about how the circuit can be improved so that it can execute any one of many different instructions and execute sequences of program instructions. The microcomputer block diagram and sequence of actions shown in Figure 2-1b strongly suggest the features needed to do this. To be able to execute any one of many different instructions, the microprocessor controller needs to be designed so that it can decode each of these instructions and enable the data path components as needed to execute each of these instructions. To sequentially fetch a series of program instructions from the program memory, the microprocessor

needs to have a *program counter* (PC) that holds the address of the instruction that is currently being fetched and is automatically incremented to point to the next instruction after each instruction fetch.

Microprocessors designed in the 1970's contained these basic components and when used in a microcomputer system, they functioned as described in the sequence shown in 2-1b. The microprocessor fetched an instruction, decoded the instruction, executed the instruction, and then fetched the next instruction. In the following section, we briefly describe the evolution of microprocessors from these early versions and discuss some of the major features that have been added to greatly improve the efficiency of microprocessors.

CISC, RISC, and A Short Microprocessor History

In the early 1970's, computer memory was relatively expensive and small in size. Therefore, the engineers who designed the instruction sets for microprocessors at that time worked very hard to minimize the amount of memory required to hold the instructions for a program. They did this by creating complex instructions that packed as many actions as possible into each instruction. Computers that use this design philosophy are called *Complex Instruction Set Computers* or *CISC machines*. The BCD-Add microprocessor we described in the preceding section is a very simple example of a CISC machine because it includes both the Add operation and the Correction operation in one instruction. As an extreme example of a CISC type instruction, a single REPNZ MOVS instruction in the Intel 80X86 family of processors will copy a 64 KByte block of words from one location in memory to another location in memory. Another rationale for the design of CISC machines is that the low-level CISC machine instructions are close to high-level language statements, and in theory this should make it easier for a compiler to generate efficient machine code from high-level language programs. Examples of 16-bit CISC microprocessors developed in the late 1970's for use in desktop microcomputer systems were the Intel 8086/8088, used in the original IBM PC, and the Motorola MC68000.

However by the early 1980's, as semiconductor memory became larger and less expensive per bit, researchers began to question the CISC assumptions. At UC Berkeley, Ditzel and Patterson (2) presented research showing that a simpler architecture would be a more efficient target for high-level language compilers and in a later paper, Patterson and Ditzel (3) proposed a simple pipelined architecture that became known as a *Reduced Instruction Set Computer* or *RISC machine*. In the early 1980's, Professor Patterson and his colleagues at UC Berkeley produced the experimental RISC-I and RISC-II machines in MOS technology. At about the same time, Hennessy (4) and his colleagues at Stanford described the Stanford MIPS machine which advanced the concepts in the RISC-I and RISC-II machines. Also, as described by Radin (5), other early work on RISC architectures was done at IBM as part of the 801 project.

For pure RISC machines, the distinguishing features and the advantages of these features are as follows:

1. All RISC instruction codes have the same number of bits, typically 32. This is in contrast to some CISC machines, which have instruction codes that can vary in length from one byte to tens of bytes. Fixed-width instructions are much easier to pipeline than variable-width instructions.
2. The RISC instruction set includes only very simple operations that can ideally be executed in a small number of clock cycles. These short instructions can then move through the pipeline quickly and not hold up instructions following them through the

pipeline. As you will see later, these short instructions also support the pipeline concept of completing one instruction every clock cycle. In a RISC machine design, for example, the BCD-Add instruction in our non-pipelined machine in Figure 2-5a would be divided into an Add instruction and a Correction instruction.

3. The RISC instructions for reading data from memory include only a single operand Load instruction and a single operand Store instruction. (Because of this, basic RISC machines are often referred to as "Load and Store" machines.) In contrast, most CISC machines have single instructions that can, for example, can, load an operand from memory and add the operand to a register. Two advantages of the simple load and store instructions are that their machine codes require only a few bits and they can be quickly decoded. As we will show later, both of these features make RISC instructions easier to pipeline.

4. The RISC instruction set is designed so that compilers can efficiently translate high-level language constructs to the instruction codes for the machine, and the result is easily portable from one machine to another. (It seems that compiler writers took little advantage of the powerful CISC instructions available on the CISC machines because they were so machine specific and would have made a compiler not easily portable from one machine to another. Therefore, the compiler writers wrote compilers to translate high-level programs to sequences of simple instructions. They were then essentially treating CISC machines as RISC machines.)

To summarize, the RISC philosophy is to accomplish a given task with many small, easily pipelined instructions, instead of a smaller number of much larger CISC instructions. The problem with the "pure" RISC approach is that the large number of small RISC instructions required to do program operations may cause a program to use an excessive amount of memory. This excessive use of memory is commonly called *code bloat*. In addition to the excessive use of memory, code bloat requires more instructions to be fetched from memory in a given time and thus requires a system to have greater memory bandwidth.

In spite of the code bloat problem, the overall advantages of RISC machines have led to the development of a considerable number of commercial microprocessors based on the RISC philosophy. We obviously don't have space here to list and describe all of them but some of the key RISC-based products that have been developed include the MIPS processors developed by MIPS Inc. (www.mips.com), the Hewlett Packard PA-RISC (http://cpus.hp.com), the IBM Power PC (http://www.freescale.com), the Scaleable Processor Architecture (SPARC) (www.sparc.com, and the ARM-based processor cores (www.arm.com).

As we will discuss more in a later section of the chapter where we look at examples of current RISC-type processors, the designers of the current RISC machines have used several techniques to reduce RISC code bloat and solve other pipelined processor problems. The first technique is to compromise on the pure RISC philosophy of "one operation per instruction" and include some instructions that specify two or more simple operations in a single instruction. ARM-based microprocessors, for example, can specify a register rotate operation and a register arithmetic operation in a single instruction, and they can Load multiple registers with data from successive memory locations or Store data from registers to successive memory locations with a single instruction. All of the current generation RISC processors that we are familiar with make a RISC-CISC compromise to some extent, in order to reduce code bloat and increase performance.

Likewise, the current generation CISC processors with which we are familiar all use some RISC techniques to improve performance. As a future designer, the lesson for you here is that the best final design often comes from taking the best features from different approaches that initially seem to be incompatible. Now let's take a look at the internal operation of a simple, pipelined RISC microprocessor.

Basic Operation of a Pipelined RISC Microprocessor

Figure 2-6a shows a block diagram for a pipelined microprocessor that is much more realistic than the single instruction machine we discussed earlier. The microprocessor in Figure 2-6a is based on a machine created by Hennessy and Patterson(1) to explain the operation of a pipelined RISC processor. We choose to build on their basic model because it illustrates many concepts clearly, it is very widely used in other references, and you will likely meet it again, if you take an advanced class in computer architecture. Furthermore, it is quite close to the architecture of the ARM-based microprocessors that we use as examples extensively in this book and to the architecture of many other current microprocessors used for embedded applications.

The first improvement to note in Figure 2-6a is the addition of an Instruction Memory to hold sequences of instructions. Along with this, note the presettable *program counter* (PC) that holds the address of the next instruction to be fetched from memory. After an instruction is fetched from the Instruction Memory, the PC is automatically incremented or loaded with a new value to point to the desired next instruction in memory. We will discuss this more a little later.

Next note the addition of a Data Memory. This addition allows the machine to access much more data than can be held in the register file. The Data Memory and the Instruction Memory may be separate as shown or they may be physically the same memory. Again, we will talk much more about this a little later. Next let's look at the data path for this machine.

The functional sections of the data path are separated by 100-200 bit wide registers called *pipeline registers*. These registers allow the data path to operate as a large shift register or *pipeline*. An instruction and the data for that instruction are shifted through the pipeline, one stage at a time by the system clock. Since an instruction is shifted along with the data for that instruction, the controller for a particular stage has all the information it needs to perform the operations desired in that stage. The stages are independent to a large extent. Unlike the machine described in Figure 2-5a, where one instruction was fetched and completely executed before the next instruction was fetched, a pipelined machine can have several instructions moving through the pipeline at the same time, each in a different stage of execution. The easiest way to show this is with a *pipeline diagram* such as that shown in Figure 2-6b. On the first clock, the machine fetches the first instruction in the Instruction Fetch (IF) stage and loads it into the Instruction Register (IR) section of the pipeline register. On the second clock, this first instruction is shifted along the pipeline to the Instruction Decode (ID) stage and a second instruction is fetched by the IF stage. During the ID stage the first instruction is decoded and the registers needed by the instruction are read from the register file. On the third clock, the first instruction is shifted to the Execute (EX) stage, the second instruction is shifted into the ID stage, and a third instruction is fetched by the IF stage. In the Execute stage, the ALU performs the operation decoded from the instruction in the ID stage. On the fourth clock, the first instruction moves into the Memory (MEM) stage where a computed address is sent out to access the Data Memory if needed, the second instruction moves into EX, the third instruction moves into ID, and a new instruction is fetched by the IF stage. Finally, on the fifth clock, the first instruction is shifted into the Write Back (WB) stage where the result of a computation is written back to the register file, the second

instruction moves to the MEM stage, the third instruction moves to the EX stage, the fourth instruction is shifted to the ID stage, and a fifth instruction is fetched by the IF stage.

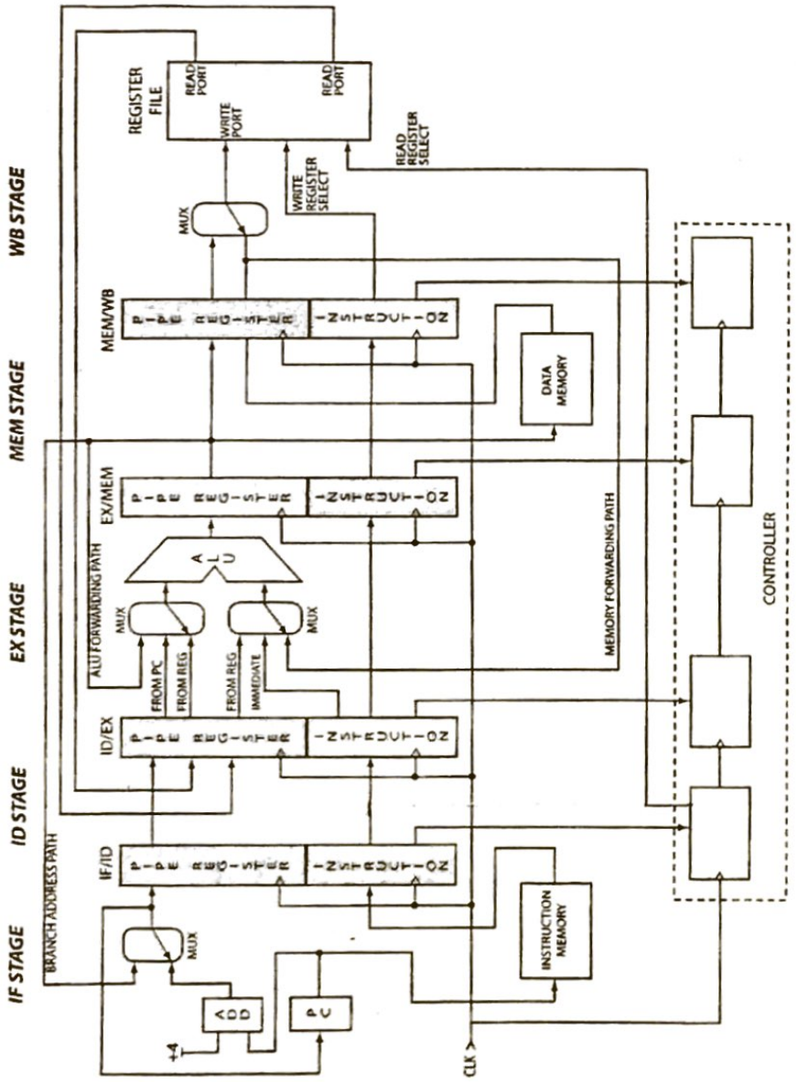


Figure 2-6a. . Block diagram for a pipelined microprocessor.

Clock # →	1	2	3	4	5	6	7	8	9
Instruction #1	*IF*	*ID*	*EX*	*MEM*	*WB*				
Instruction #2		*IF*	*ID*	*EX*	*MEM*	*WB*			
Instruction #3			*IF*	*ID*	*EX*	*MEM*	*WB*		
Instruction #4				*IF*	*ID*	*EX*	*MEM*	*WB*	
Instruction #5					*IF*	*ID*	*EX*	*MEM*	*WB*

Figure 2-6b. Pipeline diagram showing multiple instructions being executed during each clock cycle in a pipelined microprocessor.

Since the pipeline has five stages, the pipeline is full after five clock cycles and holds five instructions, each in a different stage of execution. The processor is essentially executing five instructions at the same time, so this is an example of *parallel processing*. Although each instruction takes 5 clock cycles to move through the pipeline, ideally one instruction completes every clock cycle, once the pipeline is full. Another way to look at this is that, since five instructions can complete in the same time that a non-pipelined machine requires for one 5 cycle instruction, this pipelined machine has a *speedup* of 5 over a non-pipelined machine such as our BCD Add machine. Remember that the BCD Add machine requires the execution of the BCD Add loop on one set of BCD values to complete (5 clock cycles) before execution on another set of BCD operands can start. There are many conditions that can decrease the performance of a pipelined machine but in most cases, a pipelined machine is obviously more efficient than a comparable non-pipelined machine.

Next, let's look at the controller for the microprocessor in Figure 2-6a. Although the controller is likely implemented in a single logic block, we drew it as separate blocks to remind you that the code for each instruction is shifted through the IR sections of the pipeline registers along with the data for that instruction, and is used by the controller to determine the required signals at each stage.

Now that you have an overview of a basic pipelined microprocessor, the next step is to describe some pipelined processor problems that can potentially reduce overall performance significantly, and then describe techniques used to reduce these problems. It is important for you to know about these, so you understand the features in current processors and so that when we teach you how to write programs, you will better understand the techniques used to write programs that run most efficiently on pipelined processors.

REDUCING RESOURCE CONFLICTS WITH RESOURCE DUPLICATION

A problem is created if a pipelined processor does not have enough of some resource for all the stages that need it at a particular time. As a first example of this resource conflict, suppose that a processor has only one memory for both data and instructions. As shown in the pipeline diagram in Figure 2-6b, during clock cycle 4, the processor is potentially accessing memory to Read or Write data for instruction 1, and in the same clock cycle the processor is attempting to do an instruction fetch for instruction 4. If there is only a single memory, both operations cannot access the memory at the same time, so the Instruction Fetch for instruction 4 must be *stalled* for one clock cycle until instruction 1 has finished accessing memory for its data Read/Write. Furthermore, if instruction 2 accesses memory during its MEM stage, the instruction fetch for instruction 4 will need to be stalled another clock cycle until the memory is available. These stalls delay the execution of instruction 4 and all the instructions after it, and thus decrease overall performance. The cure for most cases of this conflict is to design the system with separate code and data memories as shown in Figure 2-6a.

A second type of resource conflict could occur if the register file did not have enough ports to allow operands to be read from the register file in an ID stage that occurs during the same clock cycle as a Write to the register file in the WB stage of an earlier instruction. As we mentioned previously, a microprocessor register file is usually designed with at least two Read ports and one Write port to avoid this conflict.

REDUCING DATA AVAILABILITY PROBLEMS WITH FORWARDING AND CACHES

A second pipelined processor problem is caused by data operands not being available when needed. In manufacturers' data books this is commonly referred to as a *data hazard*. The most common type of data hazard is a Read After Write or RAW hazard. The name RAW comes from the fact that for this type of hazard, the processor needs to wait to **R**ead an operand until **A**fter it is **W**ritten to the register file by an earlier instruction. To help you better understand this, we will use a simple sequence of low-level instructions for a typical pipelined RISC processor. The instruction sequence with explanations after the @ symbols is as follows:

```
ADD R1, R2, R3  @ Add the contents of Register R3 to the contents of Register R2
                @ and put the sum in Register R1
ADD R5, R1, R4  @ Add the contents of Register R4 to the contents of Register R1
                @ and put the sum in Register R5
SUB R7, R9, R8  @ Subtract the contents of Register R8 to the contents of Register
                @ R9 and put the difference in Register R7
```

The potential problem with this sequence is that the result in R1 produced by the first ADD instruction is one of the source operands for the second ADD instruction. As shown by the pipeline diagram in Figure 2-6b, the first ADD instruction writes the result to the register file in

its WB stage but instruction 2 needs to read the operand from the register file at the start of its ID stage, which is three clock cycles earlier in time. If instruction 2 cannot obtain the data until the data is written back to the register file, instruction 2 will need to stall for three clock cycles until the data is available in the register file. These stall cycles would, of course, decrease performance. The cure for this problem is to provide a "shortcut" or *bypass* connection. This connection allows the output of the ALU Output register to be fed back or *forwarded* directly to the multiplexer on the input of the ALU, as well as being written to the register file. Look closely at Figure 2-6a and you should be able to find this connection. This forwarding path through the multiplexer will be enabled, if the controller finds that an operand is needed before it will have been written back to the register file. In some machines, forwarding does not totally eliminate the stalls caused when an operand produced in one instruction is needed in the immediately following instruction.

As we discuss more in a later chapter, you can often eliminate these stalls by placing the instruction that needs the operand a little later in the program. As a preview of this, note that the SUB instruction in the sequence above is not dependent on the results from either of the ADD instructions.

The SUB instruction can simply moved or *scheduled* before the second ADD as follows:

```
ADD R1, R2, R3  @ Add the contents of Register R3 to the contents of Register R2 @ and
                put the sum in Register R1
SUB R7, R9, R8  @ Subtract the contents of Register R8 to the contents of Register
                @ R9 and put the difference in Register R7
ADD R5, R1, R4  @ Add the contents of Register R4 to the contents of Register R1
                @ and put the sum in Register R5
```

The second ADD will now be later in the pipeline and therefore one more clock cycle is available for the result of the first ADD to be produced before it is needed. This technique is called *pipeline scheduling*

Another example of the problem caused by operands not being ready when needed is the case where an instruction needing an operand from the Data Memory is started in the pipeline immediately after the instruction that loads the data from memory. This situation is shown with low-level instructions and comments as follows:

```
LDR R1, [R2]    ; Load (Copy) the value from the memory at the address in R2 to
                ; to register R1
ADD R4, R1, R3  ; Add the contents of Register R3 to the contents of Register R1
                ; and put the sum in Register R4
```

As shown in the pipeline diagram in Figure 2-6b, the LDR instruction will write the value from memory into the register file during its WB stage. The ADD instruction needs to read this value at the start of its ID stage, which is three clock cycles earlier in time.

One way to reduce the number of these data stalls is to provide a forwarding path from the pipeline MEM stage to a multiplexer on the ALU input, so the following instruction does not need to wait for data from memory to be written to the register file. This is the same forwarding trick we used to avoid data stalls for operands produced by the ALU. If you look carefully, you should find this MEM forwarding path in Figure 2-6a.

However, although forwarding does help reduce memory load stall cycles by bypassing the register file, it does not overcome the major cause of memory stalls, which is the time it takes to read the data from memory. Over the last few years, processor clock frequencies have increased much faster than the memory clock frequencies have increased. One result of this is that it may take 100 or more processor clock cycles for an addressed data word to be returned from the microprocessor system's main memory to the register file. The instruction waiting for the data word and all the instructions after it in the program must stall for this number of clock cycles until the data word becomes available. These memory access stall cycles can significantly decrease performance.

The number of stalls caused by the long memory access time can be reduced by scheduling the instruction that uses the data as many instructions after the memory read instruction, as possible. This gives more time for the data to get from memory to the processor. In a later chapter we will further discuss this and show examples of how to efficiently schedule instructions in your programs. However, the major way to decrease the average number of stalls for both data accesses and instructions fetches is to add caches.

A *cache* is a block of fast SRAM that is usually on the same chip as the processor. It can therefore be accessed in many less clock cycles than the DRAM main memory can. Figure 2-7 shows how caches and their controllers are inserted in the *memory hierarchy* for a microcomputer system. As you can see, the caches are located between the instruction register or register file and the DRAM main memory. For a start on understanding the operation of caches, here's how an Instruction Cache helps reduce the average number of stall cycles in a program.

The first time that an instruction is requested by the processor, that instruction is usually not present in the instruction cache, so the cache controller needs to access the main memory through the DRAM controller to read the instruction. Due to this *cache miss*, the processor suffers a significant number of stall cycles in the IF stage while it waits for the instruction to be fetched from main memory. However, at the same time the instruction is returned to the processor, it is also written to the cache and the cache controller makes an entry in the cache directory to indicate that instruction is present in the cache. The next time the processor sends out an address to access the same instruction, as for example, if the processor is executing a program loop, the cache controller detects that the desired instruction is in the cache. The cache controller then supplies the requested instruction directly from the cache. Since the SRAM cache is much faster than DRAM main memory, the processor will suffer fewer stall cycles for this *cache hit* than it would if the instruction had to come from main memory.

Adding a data cache to a pipelined machine can also reduce the average number of stall cycles for data accesses. As with accessing instructions, the first time the processor accesses a particular data word, it will be a *cache miss*, if the word is not already in the cache from a previous cache line fill operation. The processor will suffer a large number of stall cycles while it waits for the data to be returned from main memory. However, if the data word is written to the cache at the same time as it is returned to the processor, later requests for that word will be *cache hits* and the word will be supplied directly from the cache with fewer stall cycles.

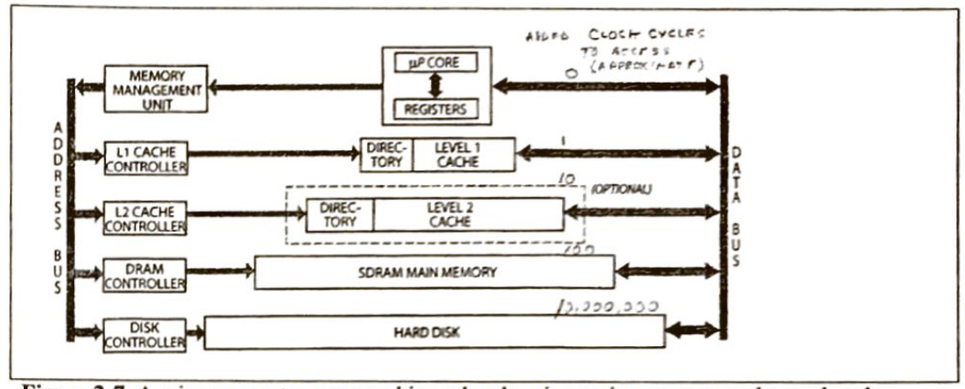


Figure 2-7. A microcomputer memory hierarchy showing main memory, caches and cache controller.

The reason that caches reduce the average number of stall cycles is that most programs have large amounts of *temporal locality* and *spatial locality*. Temporal locality means that, if a program accesses an instruction or data word at a particular time, it is very likely that the processor will access that instruction or data word again in the near future. Program loops cycle through a sequence of the same instructions over and over and thus demonstrate the concept of temporal locality. In fact, as stated in many references, most common programs spend 90% of their time executing 10% of their code and thus show a large amount of temporal locality. Spatial locality means that if a program accesses a particular memory location, it is very likely that the processor will soon access a location very near that memory location. Sequential execution of instructions is an example of spatial locality for instructions. Arrays are examples of spatial locality for data. To take maximum advantage of spatial locality, most cache controllers read an entire *cache line* of 32, 64, or more bytes into the cache from main memory, instead of just reading the requested instruction or data word that caused a cache miss. Reading in the additional bytes ahead of time increases the cache *hit rate*. The reason for this is that, if the processor next requests a data word that was already loaded into the cache as part of a previous cache line fill, the request will produce a cache hit. Because the word is already in the cache as a result of the line fill, the request will be supplied from the cache, even though this may be the first request for that particular word. Note that as in the memory hierarchy diagram in figure 2-7, most systems now have several levels of cache. The L1 cache is limited in size but it is close to the processors and very fast. The higher level caches are further from the processor, so they can be larger but the tradeoff is that they are somewhat slower. However, all of the caches are much faster than the SDRAM main memory.

A great deal of research has been done on improving cache performance and the result is that today's systems typically have a 90-95% cache hit rate. In a later chapter we discuss cache characteristics in more detail, but the preceding discussion should help you understand the caches you will see in the microprocessor examples later in this chapter. In the next section, we discuss another significant cause of stalls in pipelined microprocessors and describe some of the basic techniques used to minimize these stalls.

REDUCING BRANCH STALLS WITH BRANCH TARGET BUFFERS AND BRANCH PREDICTION

In a previous high-level language programming class, you may have written some If-Then-Else code of the form:

```
If (temp > max)
    alarm = on;
    heater = off;
Else
    alarm = off;
    conveyor_motor = on;
```

The execution flow and code layout in memory for this simple If-Then-Else program section is shown in Figure 2-8. The diamond shaped box represents the instruction that checks if temp is greater than max and directs execution to the correct code sequence based on this determination. If temp is greater than max, then execution goes to the instruction that turns on the alarm and then to the instruction that turns the heater off. After executing these two instructions, the program counter is loaded with a new value that *jumps* or *branches* execution to the next instruction after the "Else" block of code. This type of branch or jump is called an unconditional branch or an unconditional jump because the branch or jump will always be taken. It does not depend on any condition. Now let's look at the other path through the program flow.

If temp is less than or equal to max, then the program counter will be loaded with a new value that *jumps* or *branches* execution to the Else block of code that turns off the alarm and turns on the conveyor belt motor. This type of change in program flow is called a *conditional branch* or *conditional jump* because some condition is evaluated to determine if the branch is taken or not. Both conditional and unconditional branches can cause stall cycles in pipelined microprocessors. We will use the pipeline diagram in Figure 2-6b to help explain why.

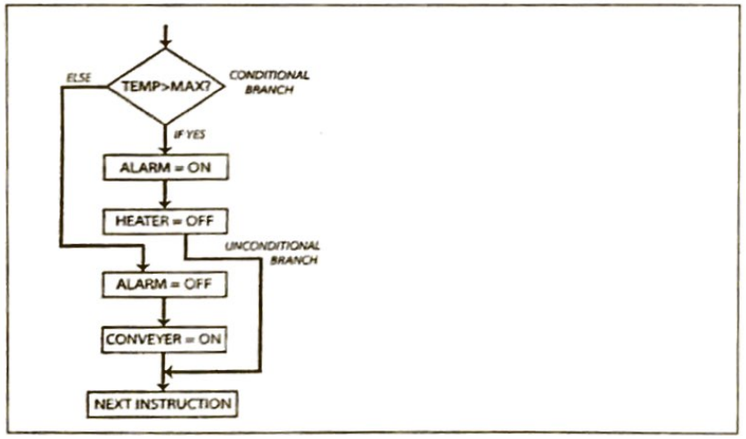


Figure 2-8. Instruction execution flow and memory storage for simple IF-THEN-ELSE program section.

The major advantage of a pipelined processor is that it does not wait until one instruction has been fetched and executed before it fetches the next instruction. As soon as one instruction moves into to the ID stage, the next instruction is fetched; and as soon as that second instruction moves into the ID stage, a third instruction is fetched, etc. For normal execution without a branch, the program counter is just incremented and instructions are fetched from sequential memory locations in order and shifted through the pipeline. For the instruction flow in Figure 2-8, the processor will fetch the instruction used to evaluate the (temp > max) condition and branch if the condition is false. By the time this conditional branch instruction gets to the EX stage where the branch destination or *branch target* is determined, the processor will have fetched the instruction that turns on the alarm and fetched the instruction that turns the heater off. (You can see in Figure 2-6a, that the computed branch target address and signal which controls the address multiplexer in the IF stage both come from the EX stage.) However, if the condition is false, we don't want to execute the "turn on alarm" and "turn off heater" instructions that are in the pipeline. We want to execute the "Else block" instructions. To do this, we need to: disable the "If Block" instructions in the pipeline behind the compare/branch instruction, load the program counter with the starting address for the "Else block" instructions, and start fetching and filling the pipeline with the instructions from the "Else block." This situation is similar to taking a wrong decision at a fork in the road while you are driving and not discovering your mistake until you have driven three miles down the wrong road. You then have to go back the three miles to the turning point before you can go forward down the correct road. Just as it takes time to correct a driving mistake such as this, it takes a pipelined processor some number of clock cycles to recover from fetching instructions along the wrong path and then fetch instructions from the right path. The number of clock cycles required for this is commonly called a *branch penalty*.

Another example of a case where execution encounters a branch penalty is the unconditional branch at the end of the "If block" instructions. By the time the branch instruction after the heater=off instruction in the "If block" is in its' EX stage, where the branch target is known, the processor will already have fetched the two instructions from the "Else block" because these are in memory immediately after the branch instruction at the end of the "If block." When the processor determines that the unconditional branch around the 'Else block" instructions is to be taken, it needs to disable the "Else block" instructions already in the pipeline, reload the program counter with the address of the next instruction after the Else block", and start fetching instructions from this branch target address. Flushing the instructions from the pipeline and loading the correct instructions, of course, results in a branch penalty of some number of clock cycles.

The two most common hardware techniques used to reduce the branch penalty for pipelined processors are *branch target buffers* and *dynamic branch prediction*. Because these techniques are used in most current microprocessors, we will give an explanation of how they work at a depth needed for our discussions in later chapters.

Branch Target Buffers

A branch target buffer is essentially a cache that holds the target addresses of branches that have been taken before, during the currently executing program. The first time a processor takes a particular branch, the processor has to compute the target address during its EX stage and feed the computed target address back to the IF stage. The IF stage can then use that address to start fetching instructions from the branch target address. As previously discussed, this process causes

stall cycles. In a processor with a Branch Target Buffer (BTB), the computed branch target address for a taken branch is also stored in the BTB for future reference. If the processor fetches the same branch instruction again, as for example it might do in executing a program loop over and over, the branch target address for this branch has already been computed and stored in the BTB. For the pipeline in Figure 2-6b, it can be copied from the BTB to the Program Counter during the ID stage of the branch instruction and used to start fetching instructions from the branch target address during that clock cycle. In this case the instructions will be fetched from the branch target address with no stall cycles. As with instruction and data caches, a BTB usually improves performance but it doesn't always. Here is a case where it doesn't.

Suppose that the target for a branch instruction is in the BTB and the processor uses it to start fetching instructions from the branch target address but, for this particular case, the branch should actually not be taken. In this case, the processor must disable or *flush* the instructions fetched from the branch target address and loaded into the pipeline, load the Program Counter with the address of the next instruction after the branch, and start fetching instructions from that address. This, of course, produces stall cycles. Dynamic branch prediction can help reduce the overall number of stall cycles caused by branches.

Dynamic Branch Prediction

The problem with the basic BTB approach is that the presence of the branch target address in the BTB does not always mean that the branch should be taken. What is needed is some way of predicting for each branch instruction fetched, whether to use an address from the BTB or the address of the next sequential instruction after the branch instruction. Based on the common wisdom that "past behavior is a good predictor of future behavior," we do this by storing *predictor bits* with each address entry in the BTB. The values of the predictor bits are determined by whether that particular branch was taken or not taken during the previous times the branch instruction was executed. Figure 2-9a shows the states and actions for a commonly used 2-bit predictor. The four states in this predictor are Strongly Taken (ST), Weakly Taken (WT), Weakly Not Taken (WNT), and Strongly Not Taken (SNT). We will use the program flow diagram for the program loop in Figure 2-9b to briefly explain how this 2-bit predictor works with the BTB.

When execution reaches the branch at the end of the loop the first time, the branch target address is not in the BTB, so it is a BTB miss and the processor will suffer branch stall cycles. However, the calculated branch target address will be put in the BTB. Assuming the branch prediction bits were initialized to Weakly Taken, the branch was taken, so the prediction bits will be changed from Weakly Taken to Strongly Taken. The next time execution reaches the branch instruction at the end of the loop, the branch target address is in the BTB and the branch prediction is Strongly Taken. The processor will load the branch target address from the BTB into the PC and fetch instructions from the branch target address with no stall cycles. The loop will then execute over and over with no stall cycles for the branch instruction. However, when execution leaves the loop, the processor will suffer some stall cycles because the prediction of Strongly Taken was wrong. In this case, the processor must flush the pipeline and start fetching instructions from the address immediately after the branch instruction. Because the prediction was wrong, the prediction state is changed to Weakly Taken. The next time this loop is executed in the overall program, there will be no stall cycles during any number of loop executions, as long as the branch target address has not been overwritten in the BTB. Stall cycles will only occur when execution leaves the loop. Without the BTB and prediction bits, the processor would suffer stall

cycles each time around the loop and at the exit from the loop. Therefore, these features significantly reduce stall cycles and improve performance. However, it is not a perfect solution, and the processor will still suffer stall cycles for the cases where the branch target address is not in the BTB, or the cases where the branch prediction is wrong. To add to your understanding of predictor bits, we leave it for you as an end of chapter problem to figure out how this 2-bit predictor and BTB system would respond to the program loop in Figure 2-9b if the predictor bits were initially set to Strongly Not Taken instead of Weakly Taken.

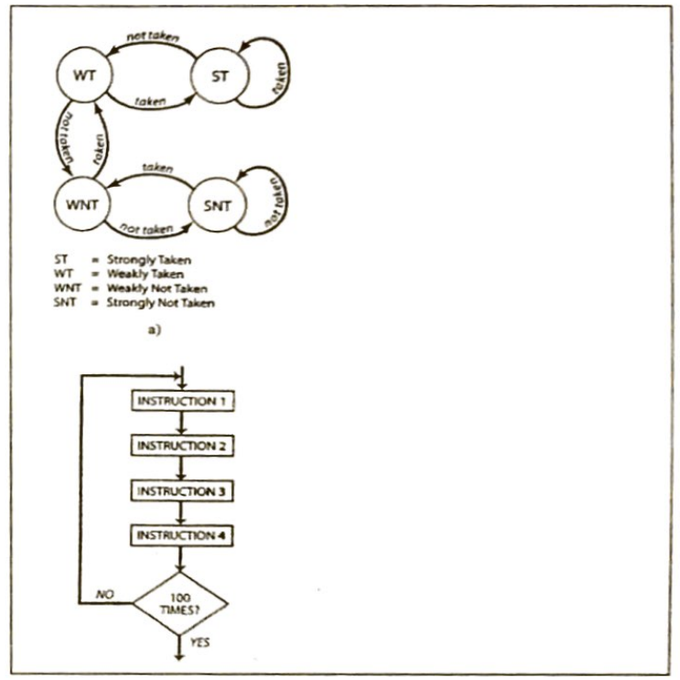


Figure 2-9. Branch prediction bits. (a) Actions and stages for 2-bit branch predictor. (b) Program flow example to show use of prediction bits.

ADDING A MEMORY MANAGEMENT UNIT FOR VIRTUAL MEMORY AND PROTECTION

Virtual memory is essentially an extension of the caching scheme we described for the main memory that we discussed previously. To implement virtual memory, a hard drive is added to the memory hierarchy as shown in Figure 2-7. The DRAM main memory functions as a high-level cache for the program sections currently being executed by the microcomputer. The term *virtual*

memory here refers to memory space that appears to be present from a programmers' viewpoint but is not all available in physical memory at any one time. In other words, if you are writing a program for a system that implements virtual memory, you can write a program that would, for example, fill 8 Gigabytes of memory, even though the microcomputer has only 4 Gigabytes of physical DRAM memory installed. The trick here is that, just as only the current code and data sets are present in a cache during program execution, only the currently executing program sections are present in the DRAM main memory at a particular time. Program sections are read in from disk to physical memory as needed, and program sections no longer needed are swapped back out to disk to make room for the sections needed. The Locality of Reference characteristic found in most programs makes this possible and efficient.

Program sections are swapped between the hard drive or SSD and DRAM in blocks or pages of typically a few Kbytes. Managing virtual memory can be done totally in software by the operating system. However, microprocessors for all but the simplest applications have an on-chip controller called a *Memory-Management Unit* or MMU to manage virtual memory more efficiently.

An MMU functions in much the same way as a cache controller. It checks each memory access to see if the page containing the requested address is present in the DRAM main memory. If the page is present, the address is sent out to main memory and all is well. If the MMU finds that the page containing the desired address is not present in main memory, the MMU interrupts the executing program. An interrupt procedure copies a page from DRAM out to disk to make space in DRAM main memory, if necessary, and then copies the needed page from the hard disk to the DRAM. At the end of this interrupt procedure, execution returns to the interrupted program. The processor then reads the desired word from the page now in physical memory and continues. .

In addition to managing virtual memory, an MMU can also be used to protect operating system code and data from access by user programs, and protect one user program from unintended access by another user program. We will discuss this much more in a later chapter but the basic principle is that the MMU uses tables to keep track of the pages. The entry for each page in a page table contains a privilege bit that indicates whether the page has supervisor-level privilege or user-level privilege. Operating systems page entries are marked as "supervisor" and user page entries are marked as "user." If a user program attempts to access an address that is in a page marked "supervisor," then the MMU will generate a signal that interrupts the executing program to indicate that an error has occurred. In a system executing multiple user programs, each user program has its own table of pages and only the operating system can switch from one user's page table to another user's program page table. Therefore, user programs cannot access each other's page tables or pages. As we said earlier, we will discuss virtual memory and MMUs much more in a later chapter, but the discussion here should give you a basic understanding of these when you see an MMU in the microprocessor examples later in the chapter. Next let's take a look at how microcomputer performance has been improved by integrating peripheral interface devices into microprocessors.

ADDING PERIPHERAL INTERFACE FUNCTIONS

In most early microprocessors, there was only enough "chip real estate" or, in other words, transistors available to implement the microprocessor on the chip. As technology evolved to allow more transistors on a chip, some microprocessors added a programmable timer, a priority

interrupt controller, some parallel port lines, and some memory on the chips with the microprocessor. The advantages of having the peripherals on-chip with the microprocessor include: faster access, less PC board area, less overall cost, and since system failure rate is a function of the number of devices, lower overall failure rates.

As semiconductor technology advanced, it became possible to include memory controllers, floating point processors, graphics processors, display/touchscreen controllers, FPGAs, network interfaces, and many other controllers for peripheral devices. As we reach one-billion transistor ICs, this trend of more and more integration has lead to complete, multi-processor, microcomputer systems on single ICs. As a next step here, let's take a first look at the features of a few examples of current generation microprocessors.

Armed-Based Processors

HISTORY

The two traditional categories for classifying microprocessors were those designed for use in "embedded microprocessor systems" such as washing machines, camcorders, microwave ovens, automobiles, etc. and those designed for use in "desktop systems" such as PCs. For the early microprocessors intended for use in embedded applications, the designers focused on low cost, low power, and small package size. Therefore these processors, or microcontrollers as they are usually called, were designed to: only work with 4-bit or 8-bit words; operate with low clock frequencies; and include some RAM and ROM memory, I/O ports, and other interface circuitry on the same IC as the processor. Some representative examples of 8-bit embedded controllers of this type include the Intel 8051 family, the Motorola MC68HC11/MC68HC12 family, the Texas Instruments TMS370 family, and the Microchip PIC12C5 family.

For microprocessors designed for use in desktop systems, the design criteria have been: maximum clock frequency, large memory address range, maximum performance on benchmark programs, efficient processing of floating point and video graphics data, and ability to communicate over high speed networks. Processors designed to meet these criteria usually dissipate considerable power, and systems using them may require cooling fans or, in the case of the AMD FX devices, a built in liquid cooling system, to remove the heat generated by the processor. Also, to reduce the power generated in the processor IC, many of the system interfacing and I/O functions are implemented in external ICs or even on a peripheral printed circuit board, rather than in the processor. Current examples of this type of microprocessor include the 64-bit Intel Pentium 4 based processors such as the Core 3, Core 5, and Core 7 devices and the 64-bit AMD FX-series devices.

The dividing line between the two traditional microprocessor categories has essentially disappeared with the introduction of laptop computers, smart phones , and other battery-powered systems that require features from both categories, specifically high performance, small package size, low power dissipation, networking capabilities, and multimedia capabilities.

However, as microprocessors became more popular and were used in a wider range of applications, a large number of companies designed and produced microprocessors. For the most part, each company developed its own architecture, instructions set, interfaces, etc. A considerable number of companies even produced several different processors with different architectures and instruction sets within their own product lines. This lack of standardization made it difficult and time consuming for a designer to move from one processor to another as

needed for different products. In recent years, this situation has been somewhat improved by the fact that many companies have developed microprocessors, based on processor cores that use the Advanced RISC Machine (ARM) architecture and instruction set developed by ARM Ltd. in the UK. (At this writing there are at least 25 companies that have developed ARM-based microprocessors and the number of companies keeps growing.)

According to Furber (6), the first ARM processor was produced by in the ARM Ltd. UK in 1985 for use in the Acorn personal computer. In the years since then, ARM Ltd. has developed a series of increasingly powerful ARM processor *cores* that are licensed to semiconductor manufacturers for use in those companies' microprocessor designs. Cores are available either as HDL descriptions or as physical IC designs. Starting with the core processor design, each company adds memory and peripheral interface circuit blocks around the core to produce a complete microprocessor IC or other *System-On-a-Chip* (SOC) with the desired features. Some of the advantages of starting with an already developed core are: the basic processor core has already been tested and debugged; the Instruction Set Architecture for the core processor may already be well known to programmers; and extensive software development tools and application programs, such as an operating system, are very likely available for the core processor. All of these factors significantly decrease the "time-to-market" for a new processor and increase the chances that the new processor will make a profit for the company. The design tradeoff here, of course, is reduced effort and time-to-market versus the cost of licensing the microprocessor core.

Several years ago, data presented at an Embedded Systems Conference in San Francisco indicated that a significant number of embedded application designers were moving from 8-bit and 16-bit microprocessors to 32-bit processors and that ARM-based processors accounted for a substantial majority of the 32-bit processors being used in new embedded designs. Based on this trend and the fact that there are a large number of engineering jobs available in the area of embedded system design, we use an ARM-based microprocessor and development board for the embedded design and interfacing examples in this book. The data for all the many different ARM cores is available on http://www.arm.com/products/processors/, but for our discussions here, we will focus on the ARM Cortex-A8 that is used in the Texas Instruments Sitara AM3359 processor. This TI processor is the one used in the Beagle Bone Black board that we use as the example development board for the programming and interfacing examples in the rest of the book.

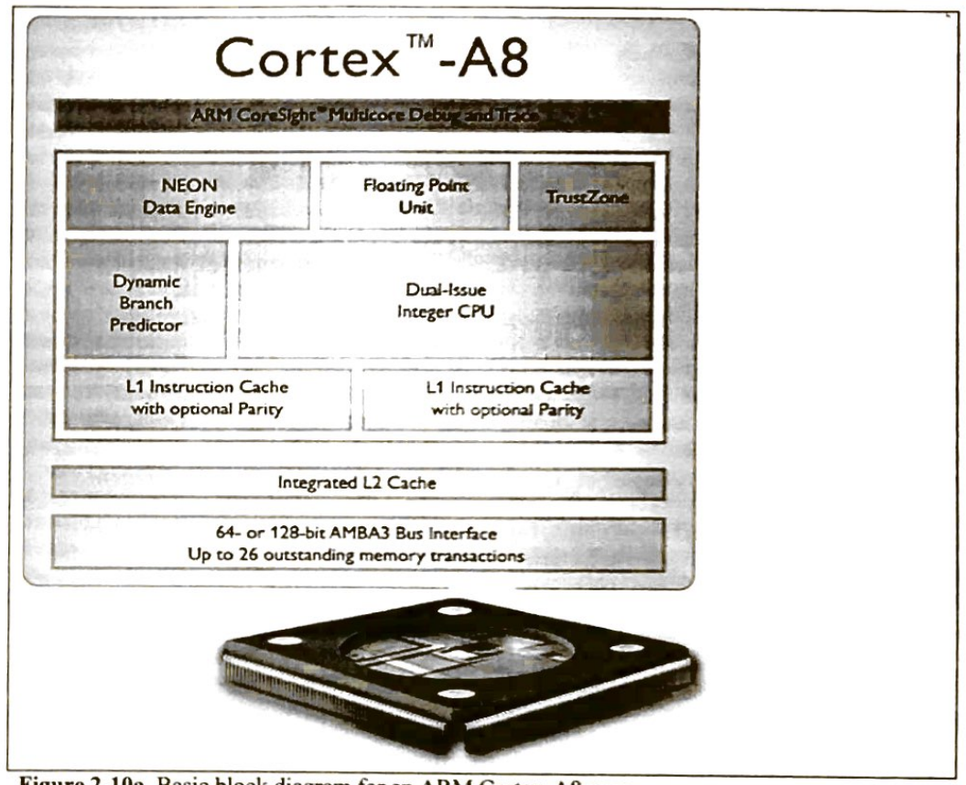


Figure 2-10a. Basic block diagram for an ARM Cortex-A8 core.

Figure 2-10a shows a basic block diagram for an ARM Cortex-A8 core and Figure 2-10b shows the pipeline architectures for this core. Note that the Integer pipeline has two ALU pipelines, ALU pipe 0 and ALU pipe 1, so two ALU instructions can be started or *issued* in parallel the same clock cycle. This is indicated by the Dual-issue Integer CPU block in Figure 2-10a.

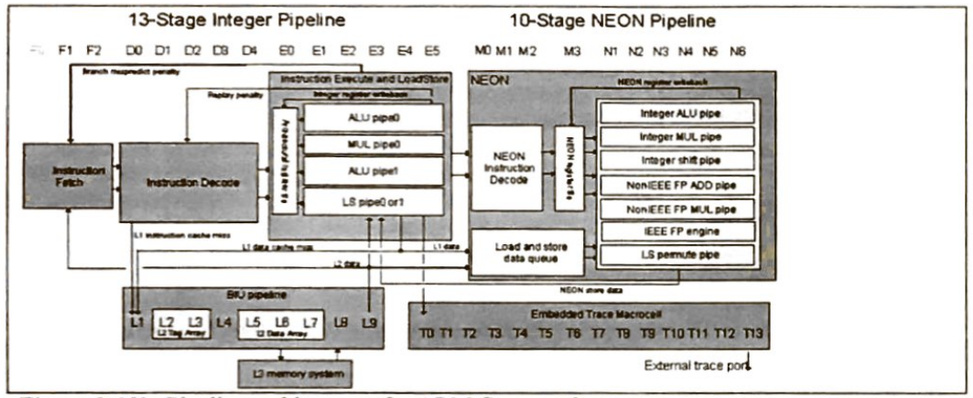


Figure 2-10b. Pipeline architectures for ARM Cortex-A8.

The Neon Pipeline section of the Cortex A-8 provides high performance media processing for some audio, video, and graphics workloads as well as standard floating-point processing. Note that the core has a fast Level 1 instruction cache and a fast Level 1 data cache. As explained in an earlier section of the chapter, having separate instruction and data caches allows both an instruction and data to be fetched in the same clock cycle and this reduces pipeline stall cycles. The core also has a larger Level 2 cache that is shared and contains both instructions and data. The L2 cache is farther away from the processor, so it can be larger but the tradeoff is that it will be slower as shown in the memory hierarchy in Figure 2-7.

The core also has branch prediction circuitry that uses a branch target buffer and global history/prediction logic. As we described previously, this hardware reduces branch stalls. Also present in the core is the Trust Zone block which supplies hardware support for security for code and data.

Also present in the core but not shown in the block diagram is a full Memory Management Unit (MMU) that supports virtual memory. Also, this ARM core has a Jazelle capability that allows it to execute 8-bit Java byte code instructions with a combination of hardware and software as needed in Java-enabled 4G mobile phones and PDAs. Furthermore, this core is a 32-bit processor but this core has capability to use 16-bit instructions called the *Thumb instructions*. The Thumb instructions are a subset of the full 32-bit instructions and help reduce code bloat because two 16-bit Thumb instructions occupy the same memory space as a single 32-bit instruction. The Cortex A-8 core also
contains *Joint Test Action Group* (JTAG) interface circuitry. The JTAG interface is a dedicated serial interface that is used to serially shift test data into the processor and shift test results out from the processor. The JTAG interface is also used by an external device called an *In-Circuit-Emulator* (ICE) to transmit debugging information between a program development system and the microprocessor. In Chapter 3, we describe the use of an ICE for debugging prototype hardware and software. For some systems, the JTAG interface is also used to program on-board Flash EEPROM.

Now, that you have an overview of the ARM Cortex A8 core, let's look at an example of a current microprocessor family that is built on this core.

TEXAS INTRUMENTS SITARA AM335X MICROPROCESSOR FAMILY

For the example microprocessors in this book, we obviously could not cover even a small percentage of the available microprocessors or even a small percentage of the microprocessors based on ARM cores. In this section of the chapter, we introduce the TI Sitara AM335X family of microprocessors that are based on the ARM Cortex A-8 core. We use a member of this family processors for the examples in this book because, with this level processor, we can lead you all the way from simple controller applications to more complex, Android/Linux-based systems such as an intelligent robot with camera, GPS, wireless, audio, and video interfaces. Also, low-cost development boards are available for this processor so, if you want, you can obtain one of these boards to experiment with the applications shown in the book and/or develop your own microprocessors applications. This is important because it is by designing real systems and learning from your mistakes that you gain maturity as a designer and go beyond the usual "textbook" level of understanding. An additional point here is that once you learn how to work with a more complex processor such as one of those in the AM335X family, it is very easy to "step back" to a less powerful processor that may be more appropriate and cost effective for a given application. The reason it is easy to make this transition from one ARM-based processor to another simpler one is that the instruction set and basic instruction set architectures for all the ARM-based processors are nearly the same. The main differences are in the peripheral interfaces available and in the memory address assignments. In a later section of the chapter, to give a comparison with desktop processors, we give an overview of the Intel Pentium 4 family of processors that are used in a large number of desktop systems. Then, in the next chapter we start showing you how to write low-level programs for an ARM-based microprocessor system.

Figure 2-11 shows a block diagram for the TI AM335X family of microprocessors that are built on an ARM Cortex A-8 core. For a comparison of the major features of the different members of this family, you can go to http://www.ti.com/product/am3359. However, here we will take a first look at the features shown in the AM335X block diagram in Figure 2-11. An important point to remember, whenever you are approaching something new like this, is: don't be overwhelmed by what you don't recognize. Start by just skimming over the diagram and picking out features that you recognize to get an overview.

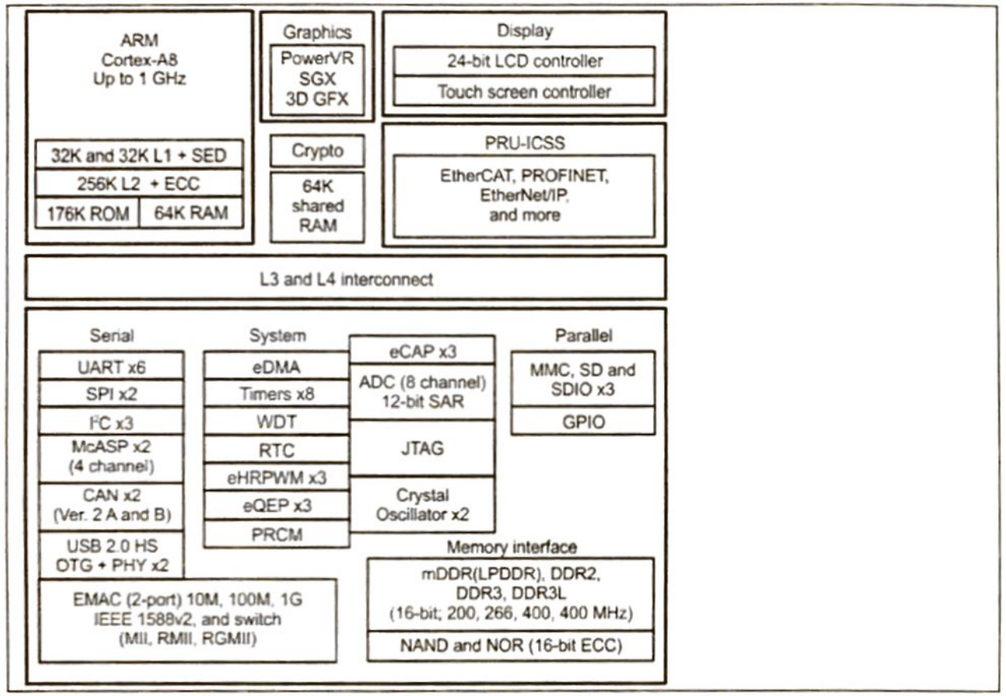


Figure 2-11. Block diagram for the TI AM335X family of microprocessors.

Then dig into the data sheet as needed to find descriptions of the blocks you don't recognize. For a start, first note the ARM Cortex A-8 core with a 32 Kbyte instruction cache, a 32 Kbyte data cache, and a 256 Kbyte L2 cache that we described previously. Next, perhaps notice the Graphics processor that can handle all of the current types of 2D graphics data as well as 3D graphics. Then, notice the Display Controller block that, not only provides full color capability for LCD displays, but also includes a touch screen controller. Next, take a look at the Memory Controller that can be used to interface with DDR type DRAMs, NOR flash devices, or NAND Flash devices. As an overview of some of the other blocks, the Serial Interface block contains interface controllers for many serial communications lines such as USB, CAN, and Ethernet. The Parallel block contains controllers for some common parallel interfaces such as SD cards and some General Purpose Input/Output (GPIO)lines . The System Block contains, along with some other stuff that we don't need to discuss right now, several programmable counter/timers and a Real Time Clock that can be used to time events or keep track of clock and calendar time. To see a detailed list of all of the features of the AM 335X processors, skim though the first few pages of the data sheet at http://www.ti.com/lit/ds/symlink/am3358.pdf. We will discuss the operation and use of many of these features in later sections of the book.

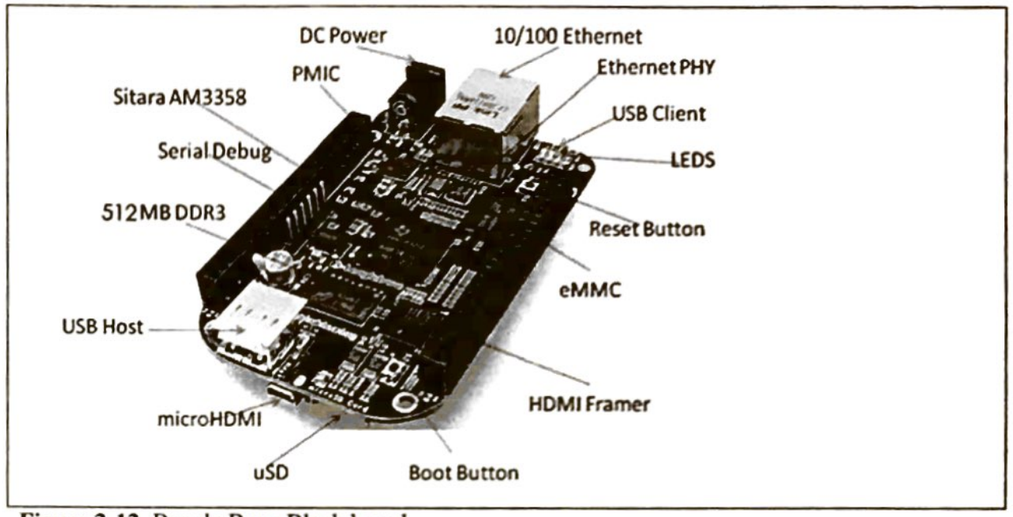


Figure 2-12. Beagle Bone Black board.

As shown in Figure 2-11, the Sitara AM 335X family devices contain a current generation processor and the controllers and interfaces required for a wide variety of low-power hand-held and other products. All that is needed for many products are a battery or power supply, connectors, a keyboard, an LCD display, a camera, and the low-level hardware needed to implement a wireless interface. If carefully designed, a system such as this can fit on a very small Printed Circuit Board (PCB) and have very low power dissipation. These features make it ideal for use in a handheld device or for use in any type of "smart" embedded microprocessor system such as an autonomous robot, medical instrument, or automotive control system. Figure 2-12 shows a photograph of the Beagle Bone Black board available through distributors listed in http://beagleboard.org/. This board contains a Sitara AM 3358 processor, Flash memory, DDR3 DRAM, and a variety of interface circuitry and connectors. To make it easier, we will from now on refer to the Beagle Bone Black board as simply the B3 board. This board is inexpensive but it has a very wide range of built in interfaces and free software development tools, Linux, and Android. Therefore, it can be used as the processor for any of a great many different "smart" embedded microprocessor systems. We will use the B3 board as the example system for the rest of this book. Throughout the rest of the book, we will discuss the operation and the use of this board for many different applications, ranging from turning on simple LEDs to much more complex systems. However, to complete this chapter and provide a contrast with the embedded system-type processors described in the preceding sections, we give in the next section an introduction to the architecture of the Intel Core Family Processors and an overview of a typical desktop microcomputer system using one of these processors.

Intel Pentium 4 Based Processors

For the design of a desktop microcomputer system, minimizing power dissipation is certainly one consideration; but the main design criteria from a marketability standpoint has been maximizing clock frequency, maximizing compute performance as measured by standard benchmark programs, maximizing the speed of multimedia features such as 3-D graphics, and minimizing cost. As we mentioned earlier, examples of microprocessors used in current desktop systems are the 64-bit Intel Core i3, Core i5 and Core i7 processors and the 64-bit AMD FX series processors. Since the Intel Core processors are used in a large percentage of the current desktop systems, we will use them it as examples here. These processors are too complex to discuss in detail, so we will just briefly discuss a few main features that distinguish them from the basic embedded system type processors.

The table at http://en.wikipedia.org/wiki/Comparison_of_Intel_processors shows a comparison of some of the features of the Intel desktop-type microprocessors from the Pentium in 1993 to the current Core i7 devices. If you take a look at this table, note the large increase in clock frequency from 60 MHz to 4.4 GHz, the increase in the number and sizes of caches, the increase in the number of processors cores, the increase in bus transfer rate, and the increase in power dissipation from less that 10 Watts to as much as 135 Watts over this approximately 20 year period.

The Intel Pentium processors in the early 1990s and all the processors after them contain Floating Point Units (FPU). These are designed to accurately perform floating-point computations in hardware, controlled by special floating-point instructions that were added to the basic microprocessor instruction set. Doing floating-point computations with this dedicated hardware is many times faster than doing them with the regular microprocessor instructions. Likewise, the Pentium II and later processors each contain a Multi-Media (MMX) coprocessor that is designed to efficiently perform operations on video and audio data in hardware using special Multi-Media instructions that were added to the basic processor instruction set. In the Pentium III, and in later processors, the multimedia capabilities were enhanced by adding 128-bit registers and the Streaming Single Instruction Multiple Data (SIMD) instructions that allow the same operation to be performed, for example, on the values for multiple image pixels at the same time.

Other major features added during that time include Out-of-Order instruction execution, Hyper-Threading Technology, and Multi-core architectures. Here's a quick overview of how these features work and how they improve system performance.

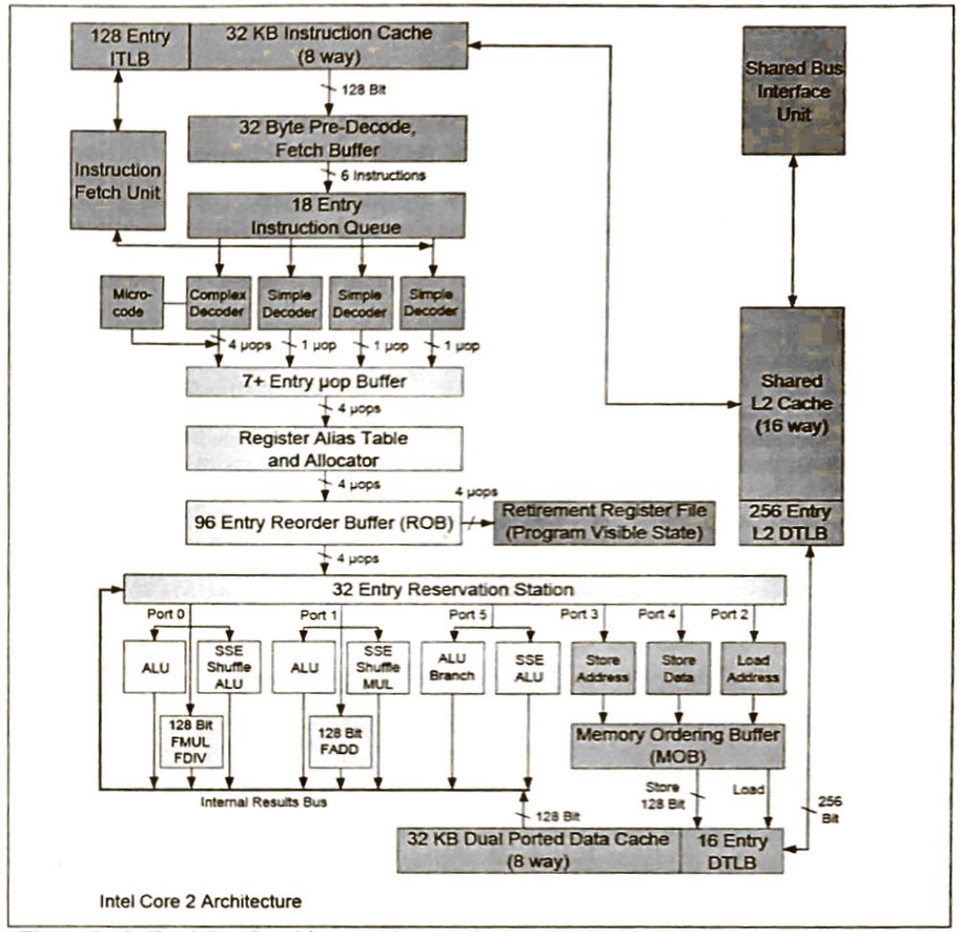


Figure 2-13. "Intel Core2 architecture" by Appaloosa - Own work

Figure 2-13 shows a block diagram of the Intel Nehalem architecture used in some of the Intel Core i7 processors. Circuitry in the top blocks fetches a sequence of instructions from the predicted code path, decodes the x86 CISC instructions, and breaks these into smaller operations called *micro-ops* or □□*ps*. Needed registers are assigned to the *micro-ops* in the Register Allocation Table. And the result is put into the Reorder Buffer that functions as an instruction pool.

When there is space available in the Reservation Station the *micro-op* is moved to the Reservation Station, where it waits until its operands are available. As soon as a *micro-op* has its operands and the needed Functional Unit is available, the *micro-op* is dispatched to the

Functional Unit. (Note that the Functional Units are pipelined, so several *micro-ops* can be moving through a Functional Unit at the same time.) The *micro-op* and the result from the Functional Unit is returned to the Reorder Buffer to wait for retirement. When all of the *micro-ops* for a CISC instruction have completed and the results of all the instructions before that CISC instruction have been written back to memory or the register file, the results of this instruction will be written back to memory or the register file, or in other words retired.

The important point here is that *micro-ops* can be executed as soon as their operands are available and the needed Functional Unit is available. There is no need for a previous instruction to complete before execution of this *micro-op* can get started. This Out-Of –Order execution of *micro-ops* provides a solution for many of the stalls suffered by simple pipeline architectures we described previously. Remember that in a simple pipeline, as in the grocery store checkout line, if one instruction stalls while waiting for data, all of the instructions behind that instruction stall, even if the following instructions have all the data they need to execute and the ALU is available. However, in order for the program results to be correct, the results must be written back to registers and memory in the same order they would be if the instructions were executed in the original program order. The major advantage of this "Out-of-Order" execution of *micro-ops* is that it keeps the pipelined functional units busy doing useful work as much of the time as possible. However, the overall CISC instructions must be retired in the original program order, so that the overall result will be the same as if the program were executed on a simple, in-order execution machine. This order is also necessary, so that interrupts can be handled in a way that assures the results for the interrupted program are correct.

To summarize the overall process, CISC Instructions are fetched "In-Order", CISC instructions are converted to *micro-ops, micro-ops* are executed Out-of-Order, and the CISC instructions are retired "In-Order". One disadvantage of this approach and the most likely reason it is not used in basic embedded processors is that it takes a lot of transistors to implement the all the needed hardware. Not only do these transistors increase the die size but also, clocking all these transistors at a high frequency to obtain the desired performance tends to increase overall power dissipation beyond acceptable levels for a handheld system.

However, the increasing ability to put more and more transistors on a IC die made it possible for Intel to implement *Hyper-Threading* in the later versions of the X86 processors A *thread* is a small section of executing code that is part of a larger program. In current systems it is very common for several different threads to be executing "at the same time" or "concurrently". (As an example of this, you may have noticed a word processor program doing an auto-save to disk operation at the same time as you were typing in text.) However, if there is only one processor in your machine, only one thread is actually executing at any particular instant in time. Execution is switched from one thread to another often enough that it appears to you that several are executing concurrently. Each thread switch requires saving the registers from one thread in memory and copying the register values for the new thread from memory into the working registers. This *context switch* takes valuable time and slows overall execution. The ideal situation would be to have multiple register sets and multiple processors that could be actually executing two or more threads simultaneously. A processor with Hyper-Threading Technology essentially implements two "logical processors" by having two program counters and two complete register sets. The context for a thread can be loaded into one set of registers while the other set is being used for the executing thread. When it is time to switch to the second thread, this can be done very quickly because the alternative registers have already been loaded with the values for the new thread. While this second thread is executing, the register values for the next thread to

execute can be loaded into the registers that were used for the first thread. Since the two "logical processors" are in the same processor chip, the multi-threading efficiency is usually greater than it would be for a system with two separate processors using a shared external bus.

The logical extension of the having two "logical processors" by duplicating registers as we described in the previous section is to have two or more complete core processors on a single processor IC. As semiconductor technology has scaled transistor sizing down to a few tens of nanometers, is has become very possible to put billions of transistors on a single IC. (The Intel i7 Haswell processor has about 1.6 billion transistors.) This large number of available transistors has made it possible to put multiple cores on an IC. As one example, the Intel i7 Haswell processor is available with up to 8 cores at this point and each core can maintain 2 "virtual processors", so ideally the device could handle 16 threads concurrently. .

The major difficulty with this assumption, however, is that it assumes that there are enough threads or enough tasks available to keep all of the cores busy a significant percentage of the time. The number of threads available depends on the nature of the program and on how the program is written. A server program for an Internet site or a server program for processing ATM machine requests may be easily written to have a lot of thread level parallelism and thus be able to take advantage of multicore processors. However, for other types of programs it may not be so obvious how to implement enough threads to take full advantage of a multicore processor. A key point here is that the movement to multicore processors transfers much of the responsibility for program performance from the hardware design to the programmers. The hardware designers are giving the programmers 2, 4, or more cores in a single microprocessor and telling them, "You figure out how to write programs that use these cores efficiently."

INTEL CORE I7 SYSTEM ARCHITECTURE

Figure 2-14 shows the typical configuration of an Intel Core i7 system using the Intel Z77 Express *chipset*. A chipset is an IC or group of ICs that are added to the basic processor to provide system interfaces. As shown in Figure 2-13, the Uncore section of the processor has built in interfaces for DDR3 DRAM. It also has High speed PCI Express interfaces and a very high speed Quick Path Interface for communicating with peripheral chips.

The Z77 Chipset contains display interfaces such as VGA and HDMI, several USB ports, a Gigabit Ethernet interface, and 6 SATA ports for connecting with hard disks, DVDs, and SSDs. It also has additional PCI Express connections that can be used to interface with a wide variety of devices. If you happen to take the cover off a PC, you should notice the PCI Express connectors (slots) on the main PC board. Small PC boards are plugged into these PCI Express connectors to provide custom interface capabilities.

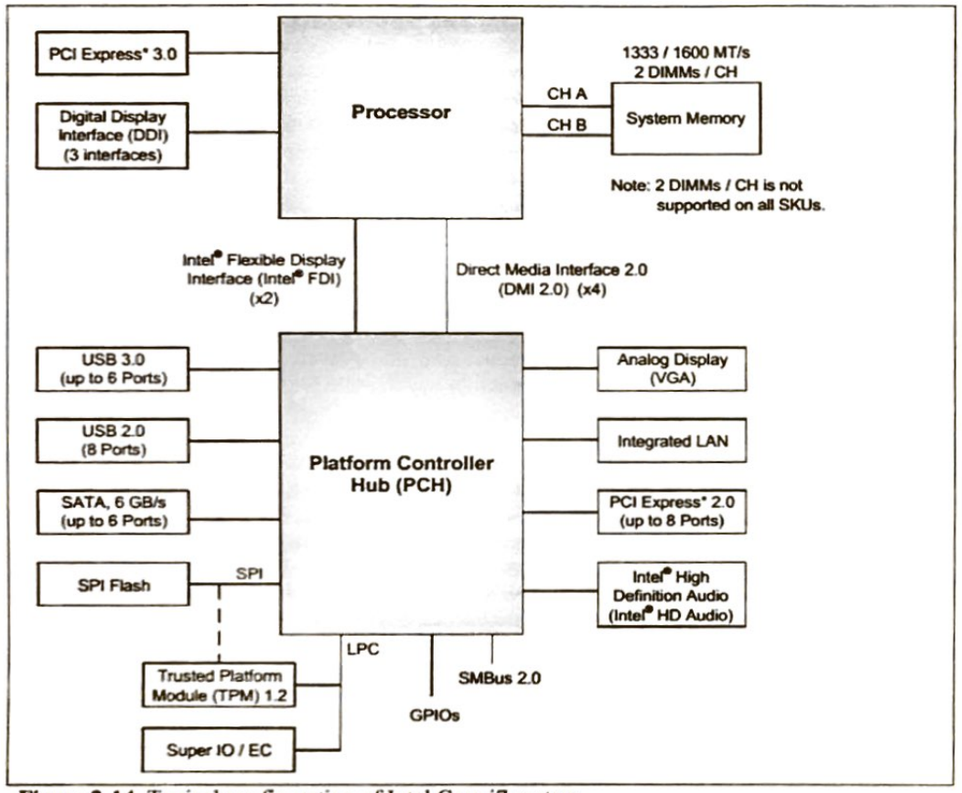


Figure 2-14. Typical configuration of Intel Core i7 system.

Summary

We have chosen to use an ARM Cortex A8-based processors and a B3 board for the examples in this book instead of using an Intel Core i7 processor system. One major reason for our choice was that by using a development board such as the B3 as a base, you can design and build microprocessor systems of your own for a wide variety of control and other applications. A second major reason for our choice was the fact that many of the current job opportunities are in the embedded system world. In this embedded world a great many ARM-based processors are used and therefore knowledge of ARM-based processors is a good asset to have on your resume. However, the many interfaces that you will learn about, for example, memories, keyboards, displays, serial buses, wireless networks, cameras, motors, and other devices are used in both embedded and desktop systems. Therefore, you would have a good start if, at some point, you do need to work with an Intel Core i7 type system. In any case, now that you have had an introduction to the TI AM335X family processors, we will in the next chapter start showing you

how to efficiently program an ARM-based microprocessor. Note that the basic machine codes and basic instruction sets for all of the ARM-based processors are essentially the same, so that, except for controller details and addresses, most of what you learn about programming a TI AM335X is directly applicable to other ARM-based processors.

References

(1) Hennessy and Patterson, *Computer Architecture, A Quantitative Approach*, 3rd edition, Morgan-Kaufmann Publishers, San Francisco, CA, 2003.

(2) Ditzel, D.R. and D.A. Patterson [1980]. "Retrospective on high-level language computer architecture," in *Proc. Seventh Annual Symposium on Computer Architecture*, Le Baule, France (June), 97-104.

(3) Patterson, D. A. and D.R. Ditzel [1980]. "The case for the reduced instruction set computer," *Computer Architecture News* 8:6(October) 25-23.

(4) Hennessy, J., N.Jouppi, F. Baskett, and J.Gill [1981] "MIPS: A VLSI processor architecture,"*Proc CMU Conf.on VLSI Systems and Computations* (October), Computer Science Press, Rockville, Md.

(5) Radin, G. [1982] "the 801 minicomputer," *Proc. Symposium Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA (March), 39-47.

(6) Furber, Steve, *ARM System-on-Chip Architecture*, 2nd Ed, Addison-Wesley Pearson Education, Harlow England, 2000.

(7) http://www.arm.com/products/processors/

(8) http://www.ti.com/product/am3359

(9) http://beagleboard.org/

(10) http://en.wikipedia.org/wiki/Comparison_of_Intel_processors

(11) http://www.realworldtech.com/haswell-cpu/4/

Checklist of Important Terms and Concepts

ALU, register file, multi-ported register file
Data path
RISC machine characteristics, advantages, and disadvantages
CISC machine characteristics, advantages, and disadvantages
Pipelined microprocessor, pipeline registers
Pipeline diagram, pipeline speedup
Hardwired controller, micro-programmed controller
Data hazard, RAW hazard, data forwarding
Temporal locality, spatial locality
Memory hierarchy
Cache hit, cache miss, cache hit rate
Conditional branch, unconditional branch, branch stalls
Branch Target Buffer (BTB), dynamic branch prediction
Virtual Memory
Memory Management Unit (MMU)
Embedded system type microprocessor characteristics
Processor core, System-On-Chip (SOC)
Desktop system type microprocessor characteristics

Floating-Point Unit (FPU), MMX co-processor
Out- Of-Order- Micro-op execution
Hyper-threading
Multi-core processors
Microprocessor chipset

Review Questions and Problems

1. Show the result of ORing the binary number 11110000 with 10101010 as might be done by an ALU. Make a statement about the effect of ORing a bit with a 1 and the effect of ORing a bit with a 0.

2. ANDing an 8-bit binary word with 11110000 is sometimes referred to as "masking" the lower 4 bits. Explain what this means and make a statement about the effect of ANDing a bit with a 1 and the effect of ANDing a bit with a 0.

3. For the single instruction "microprocessor" in Figure 2-5a, design the combinational logic that determines if a correction factor of 0110 should be added to the initial sum of the BCD digits.

4. Communicating complex ideas to others in ways that they can understand is an important skill for you to develop as an engineer. Assuming that your mother is not an electrical engineer or computer engineer, briefly tell how you would explain to her the difference between a RISC machine and a CISC machine and how you would explain the advantages and disadvantages of each approach.

5. For the pipelined processor in Figure 2-6a, why must a pipeline register be included in the forwarding path from the ALU output to the ALU input, rather than just connecting the ALU output directly to the multiplexer on the input of the ALU? Hint: Remember that an ALU consists of just a couple layers of combinational logic and think about the behavior of a circuit created by looping the output of a series connection of two inverters around to the input.

6. The pipelined microprocessor in Figure 2-6a has five stages, so 5 instructions are executing at any given time and one instruction completes every clock cycle. The ideal speedup of this machine over that of a non-pipelined machine that only completes an instruction every 5 clock cycles is then 5. It would seem that decreasing the amount of work per stage and increasing the number of stages would allow the pipeline to be clocked faster and give an even greater speedup. Use an automobile assembly line analogy for a pipelined processor to help you think of some possible problems with, for example, building a pipeline with 40 stages.

7. Use a pipeline diagram to help explain the major advantage of having separate code and data memories for pipelined microprocessors.

8. When a data cache access misses the cache in some current pipelined processor systems, it may take 100 processor clock cycles for the desired data word to be read from main memory. Describe the effect this will have on the operation of the pipeline and briefly describe a programming technique that can often be used to reduce this effect on the pipeline.

9. In our discussion of Branch Target Buffers (BTBs) and dynamic prediction bits, we assumed that the prediction bits were initialized in the Weakly Taken state. Describe the sequence of actions that this processor with a BTB and a 2-bit predictor would likely take for the program loop in Figure 2-9b, if the predictor starts in the Strongly Not Taken state instead of in the Weakly Taken state.

10. Assuming that an If-Then-Else program structure is contained in a loop, so that it executes each time that the program goes around the loop. Further assume that the probabilities of the two branches of the If-Then-Else program structure being taken are equal. (In other words, there is a 50% chance that execution will follow the If branch and a 50% chance that execution will follow the Else branch each time the overall execution reaches the If-Then-Else structure.)
    a. Describe the operation of a BTB and branch prediction bits for this If-Then-Else structure.
    b. Compare the performance of a system with a BTB and the performance of a system without a BTB for this If-Then-Else structure.

11. You are designing a simple controller for a chemical process system. As part of this controller, you need two A/D channels and one D/A channel. Your company already uses several different Atmel AT91 processors in their products, so for cost and inventory efficiency, you want to use a processor that the company already buys in large quantities. Go to the Atmel website, www.atmel.com, go to products > microcontrollers > ARM Thumb to find the AT91M42800A and the AT91M55800A product descriptions. Based on these descriptions, determine the part that would be the best choice for your design and explain the reasons for your choice.

12. Based on your own experience and on the discussion in this chapter, list some major design criteria for hand-held microprocessor-based devices.

13. We briefly introduced the TI Sitara AM3359 processor. Go to http://www.ti.com/product/am3359 to see the complete data sheet. Skim through the device summary to determine if the processor has an A/D converter(ADC) and a Pulse Width Modulation output that you need for a motor control application.

14. Describe the major method that the Intel i3, i5, and i7 processors use to keep the functional units busy as much of the time as possible and overcome pipeline stalls.

CHAPTER 3 – INTRODUCTION TO MICROPROCESSOR SYSTEM PROGRAMMING

In the last chapter, we focused on the basic architecture of microprocessors and microprocessor systems. The major goal of this chapter is to give a solid introduction to the languages, tools and techniques commonly used for programming a microprocessor system. While the architecture models from the last chapter are still fresh in your mind, we start by discussing the Programmers' Model for the ARM-based processors. Then in the next section of the chapter we describe and show examples of the three different levels of programming language available for programming microprocessor systems: machine language, assembly language, and high-level languages. The next section of the chapter introduces the features and operation of the computer-based tools you will use to write, test, and debug your programs. Finally in this chapter, we describe the sequence of steps you take to systematically and efficiently develop programs for microprocessor systems.

Objectives

At the conclusion of this chapter, you should be able to:

1. List and briefly describe the features of the Programmers' Model or Instruction Set Architecture for the ARM-based microprocessors.
2. List and briefly describe the basic types of exceptions that the ARM-based processors support.
3. Explain the terms Big-Endian memory storage and Little-Endian memory storage, and show how the bytes of a word would be stored in memory for each.
4. Describe the difference between Memory-Mapped I/O and Direct I/O and give the advantages and disadvantages of each.
5. Describe the operation(s) performed by basic ARM assembly language instructions.
6. Write ARM assembly language instructions to perform specified simple operations.
7. Describe the tradeoffs between assembly language programming and high-level language programming.
8. Describe the types of applications appropriate for assembly language programming and those appropriate for high-level language programming.
9. Describe the flow of a program through a typical program development tool chain.
10. Briefly describe the use of an Instruction Set Simulator, an on-board debugger, and an In-Circuit-Emulator.
11. List and describe a sequence of steps for systematically developing, testing, and debugging assembly language programs.
12. Represent the program solution (algorithm) for a given problem using standard program structures.

Programmer's View of Arm-Based Microprocessors

Whenever you take on the task of writing programs for a microprocessor that is new to you, one of the first steps is to dig through the data books for that processor until you find the description of the *Instruction Set Architecture* (*ISA*) or *Programmer's View* for the processor. As implied by the name, the ISA includes the features and characteristics of the processor that affect or are affected by instructions. The ISA includes the ALU, register file, operating modes, interrupts (exceptions), memory organization, I/O organization, status registers, configuration registers, and to some extent, an overview of the actual physical system. The Instruction Set Architecture discussions in the manufacturers' manuals are gateways to an unbelievable amount of detail for each of these features and the total is overwhelming if you try to absorb it all at once. On your first pass through the manufacturers' data manuals, you should just try to get an overview and make some notes about key features that seem important. As part of the design process, you will spiral through the data manuals many times and learn more and more specific details as you need them. In addition to introducing you to the ARM ISA, the discussions that follow give you an example of the level of understanding you should attempt to gain in your first read through the manufacturers' data sheets for a new processor and a new system board.

THE ALU, DATA PATH, REGISTER SET, AND OPERATING MODES

The ARM Cortex A-8 cores are classified as 32-bit processors, because they have 32-bit ALUs and all of their internal data path operations such as addition, subtraction, etc. are performed on 32-bit operands. As we will discuss more a little later, however, the ARM load instruction can copy a byte or a half-word (16-bits) from memory into the lower part of a specified 32-bit register. For those cases, however, the remaining bits in the 32-bit destination register will be filled with zeros if the data is specified in the instruction as unsigned, or it will be filled with copies of the sign bit if the data is specified as signed in the instruction. Therefore the operation is really a 32-bit operation.

Figure 3-1 shows the register file or programmer "visible" 32-bit registers for an ARM processor core in each of its seven operating modes. The registers shown in the first column in Figure 3-1 are those that are accessible when the processor is operating in *User* mode. The processor is run in User mode for general application and data processing programs. The first program example we show a little later in the chapter is intended to run in user mode. Registers R0-R14 can all be used as general purpose registers for holding data operands. Register R15 is the Program Counter or PC. In addition to its use as a general purpose registers, R14 is used to hold the address that execution will return to at the end of a procedure, so it is usually referred to as Link Register or LR in programs. R13 is used to hold a pointer to a location set aside in memory as a "stack", so it is usually referred to as Stack Pointer or SP in programs. In Chapter 4, we describe and show in detail how these registers are used in programs.

The remaining columns in Figure 3-1 show the registers that are available to programs operating in one of six *Privileged* modes. These six modes are called "Privileged" because User mode programs cannot switch to one of these modes to access system resources, except through special mechanisms. This restriction provides one way to protect System programs from malfunctioning User programs. However, a program operating in one of the privileged modes can directly change the operating mode to any one of the other privileged modes or to User mode.

Modes						
		Privileged Modes				
User	System	Supervisor	Abort	Undefined	Interrupt	Fast Interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

* Shading indicates that the normal register has been replaced by one specific to the exception mode.

Figure 3-1 Programmer "visible" registers for the ARM processor core.

As shown at the top of Figure 3-1, the Supervisor, Abort, Undefined, Interrupt and Fast Interrupt modes are entered when the processor experiences some *exception* that causes a change in program execution flow. For example, asserting the Fast Interrupt (FIQ) input pin on the processor will interrupt the executing program and cause the processor to switch to the FIQ execution mode. This is an example of a *hardware exception* or *interrupt*. An example of a *software exception* is the software interrupt instruction (SWI). Execution of the SWI instruction will automatically switch the processor to the Supervisor execution mode.

When the processor switches from one mode to another, it switches to a different register set for some of the registers. The registers shown as shaded in Figure 3-1 are the sets that are switched when the processors enters that mode. For example, if the processors is in User mode and the FIQ input is asserted, the processor will switch to Fast Interrupt mode and will now have access to the FIQ R8-R15 register set. The FIQ R8-R15 register set is totally independent from the R8-R15 register sets in the other modes. Note that the R0-R7 register set for the Fast Interrupt mode is not shaded, so it is the same as the R0-R7 register set for all the other modes.

We will, of course, discuss and show the use of these modes extensively in later chapters. However, to give you an introduction to the ARM Cortex A-8 operating modes, here are brief descriptions of the seven modes and some notes about the register sets that are specific to each mode.

User(usr) – Normal application program execution mode

System(sys) – In a processor with an operating system, the operating system is usually run in System mode. The processor is switched into the System mode from one of the other privileged modes. For example, when the system power is first turned on, the Reset signal is asserted so the processor will go to Supervisor mode. The Supervisor mode program can then switch the processor to System mode to run the operating system. In System mode, the processor accesses exactly the same register set that is used for User mode programs.

Supervisor(svc) – The processor enters Supervisor mode in response to a signal on the processor Reset input or in response to the Software Interrupt (SWI) Instruction that is often used to call operating system procedures. As shown in Figure 3-1, the R13_svc and R14_svc registers will be used by the Supervisor mode program in place of the R13 and R14 registers used in the mode the processor was operating in when the Reset signal or SWI occurred.

Abort(abt) – The processor enters Abort mode if it attempts to fetch an instruction or data from memory and finds that the instruction or data has not been loaded from disk into the physical memory on the board. As shown in Figure 3-1, the R13_abt and R14_abt registers will be used by the Abort mode program in place of the R13 and R14 registers used in the mode the processor was operating in when the memory fault occurred. A procedure operating in the Abort mode program will load the missing code or data from disk into physical memory and return execution to the interrupted program where the memory fault occurred.

Undefined(und) – The processor enters Undefined mode if it experiences an "Invalid Instruction Code" fault or, in other words, it finds that the word fetched as an instruction is not one of the instruction codes defined for the core processor. In addition to detecting an illegal instruction, this mechanism is used to discriminate between instructions intended for the core processor and, for example, floating-point instructions intended for a floating-point coprocessor that shares the instruction stream from memory with the core processor. If the System On a Chip (SOC) has a floating-point coprocessor, the core processor will wait and the floating-point coprocessor will execute the floating-point instruction. If the SOC does not have a floating-point coprocessor, the Undefined mode program can call a procedure that implements the floating-point instruction with a sequence of core processor instructions. The R13_und and R14_und registers will be used by the Undefined mode program in place of the R13 and R14 registers used in the mode the processor was operating in when the undefined instruction exception occurred.

IRQ(irq) - If the Interrupt Input (IRQ) is enabled and a signal asserts the IRQ input pin on the processor, the processor will switch to IRQ mode operation. As shown in Figure 3-1, the R13_irq and R14_irq registers will be used by the IRQ mode program in place of the R13 and R14 registers used in the mode the processor was operating in when the IRQ occurred.

FIQ(fiq) – Fast Interrupt Mode. If the Fast Interrupt Input (FIQ) pin on the processor is enabled and a signal asserts the FIQ input, the processor will automatically switch to FIQ mode operation. As shown in Figure 3-1, the FIQ registers R8_fiq – R14_fiq will then be used by the FIQ mode program in place of the corresponding registers from the mode the processor was operating in when the FIQ interrupt occurred.

The registers labeled CPSR at the bottom of the register columns in Figure 3-1 are the Current Program Status Registers. The functions for the bits in the CPSR are shown in Figure 3-2a. The N, Z, C, V, and Q at the high end of the register may look like part of an eye test chart but N, Z, C, and V here represent *condition code bits* or *flags*. These flags indicate some condition such as a carry produced by an ALU operation. As we show in detail later, the state of one of these flags is often used to determine whether a conditional branch is taken or not taken, or whether a particular instruction is executed or not. Unlike many processors, the ARM-based processors allow you to specify in each ALU-type instruction whether you want the flags to be affected (updated) by that instruction or not. The meanings for the ARM condition flags are as follows:

N: The N or Negative bit is the sign bit. This bit will be a 1 if the last instruction that was specified to affect the flags produced a 1 in the Most Significant Bit (MSB) of the result. (Remember from the exercises at the end of Chapter 1 that in the Two's Complement representation, the sign bit for a negative number will be a 1 and the magnitude of the number will be in two's complement form.)

Z: The Z or Zero flag will be a 1 if the last instruction that was specified to affect the flags left all zeros in the result register.

C: The C or Carry flag will be affected as follows:

1. C will be set to a 1 if an addition instruction produced a carry.
2. C will be set to a 1 if a subtraction instruction produced a borrow.
3. For some shift operations, C will contain the last bit shifted out of a register. We will show examples of this in Chapter 4.

V: The V or oVerflow flag will be a 1 if the last addition or subtraction instruction that was specified to affect the flags produced a signed result that "overflowed" the number of bits available for the magnitude of the result. To refresh your memory of the overflow discussion in Chapter 1, here's an example. If you add the 8-bit signed number 01010110 (+86) to 01101010 (+106), the binary result is 11000000. The 1 in the MSB position indicates that the signed result is negative, which is obviously incorrect for the sum of two positive numbers. The problem here is that the binary sum of the two numbers is too large to fit in the 7 bits available for the magnitude in an 8-bit signed number, and the result has "overflowed" into the sign bit position.

Q: The Q bit is not one of the flags for the basic ARM instructions, but is used to indicate an overflow/saturation condition for some of the enhanced Digital Signal Processing instructions present in the later versions of the ARM core processor family.

31	30	29	28	27	26	8	7	6	5	4	3	2	1	0
N	Z	C	V	Q	DNM(RAZ)		I	F	T	M4	M3	M2	M1	M0

(a)

M[4:0]	Mode	Accessible Registers
0b10000	User	PC, R14 to R0, CPSR
0b10001	FIQ	PC, R14_fiq to R8_fiq, R7 to R0, CPSR, SPSR_fiq
0b10010	IRQ	PC, R14_irq, R13_irq, R12 to R0, CPSR, SPSR_irq
0b10011	Supervisor	PC, R14_svc, R13_svc, R12 to R0, CPSR, SPSR_svc
0b10111	Abort	PC, R14_abt, R13_abt, R12 to R0, CPSR, SPSR_abt
Ob11111	Undefined	PC, R14_und, R13_und, R12 to R0, CPSR, SPSR_und
0b11111	System	PC, R14 to R0, CPSR (ARM architecture v4 and above)

(b)

Figure 3-2 ARM current program status register. (a) Bit format. (b) Mode bit assignments.

As shown in Figure 3-2b, bits 0-4 in the CPSR indicate the mode in which the processor is currently operating. When the processor switches from one mode of operation to another, the CPSR for the previous mode of operation is stored in the Saved Program Status Register (SPSR) for the new mode. For example, if the IRQ input pin is enabled and the pin is asserted, the processor switches from User mode to IRQ mode. The CPSR for the User mode program will be stored in the SPSR_irq register. Saving the CPSR is necessary so that execution can be returned correctly to the old execution mode after, for example, the IRQ service procedure completes. In Chapter 5, we show you how to write and use interrupt service procedures.

The T bit (bit 5) in the CPSR indicates whether the processor is operating in standard instruction mode or in "Thumb" instruction mode. In standard instruction mode, all instruction codes are 32-bits in length and all register and data operations involve 32-bit operands. In Thumb mode, register and data operations also involve 32-bit operands but all instruction codes are only 16-bits in length. The 16-bit Thumb instruction set and capabilities it provides are essentially a subset of those for the 32-bit ARM instruction set. Remember from a discussion in Chapter 2 that RISC-only processors tend to suffer from "code bloat." One of the solutions for code bloat is to implement a compressed instruction set that requires fewer bits per instruction and thus less memory space for programs. The tradeoff, of course, is the reduced memory space for the Thumb instructions versus increased capabilities for the full 32-bit ARM instruction set. You should be aware that the Thumb instruction set exists in case you need to design a system where memory usage must be absolutely minimized. However, in this book we will focus just on the full 32-bit instruction set and the features it provides, because performance is a key design criterion in most current designs.

The F bit (bit 6) indicates whether the FIQ interrupt input is enabled or not. If this bit is set to a 1, as it is by default after a Reset or power on, the processor will not respond to an interrupt signal on the FIQ input. Likewise, the I bit (bit 7) in the CPSR in Figure 3-2a indicates whether the IRQ interrupt input is enabled or disabled. If the I bit is set to a 1, the processor will not respond to an interrupt signal on the IRQ input. Other bits are used for more advanced functions or are reserved and should Remain At Zero for a Cortex A-8. Now that you have had an introduction to the ARM data path, register set, operating modes, and flags, the next step is to take a look at ARM system memory organization.

MEMORY ORGANIZATION

In an ARM system, each address represents a byte of data stored in memory, so a 32-bit data word or instruction occupies four memory addresses and a 16-bit half-word occupies two memory addresses. For easier access, 32-bit instruction words and data words are always stored in memory at *word aligned* addresses. The term "word aligned" means that the addresses are integral multiples of 4 bytes, starting from 0x00000000. For example, a program might be loaded into memory with the first word at 0x00008000, the second at 0x00008004, the third at 0x00008008, etc. Half-words are always "half-word aligned." This means that they are stored at addresses which are an even multiple of 2 as, for example, 0x00008080, 0x00008082, 0x00008084, etc. Now let's look more closely at how the bytes of a 32-bit word are stored in four successive memory locations.

The ARM "Load Word" instruction copies a 32-bit or four-byte word from a specified starting address in memory to a specified 32-bit register. The "Store Word" instruction copies a 32-bit or four-byte word from a specified 32-bit register to a specified starting address in memory. This seems simple enough but there is a problem.

The problem is that, as shown in Figure 3-3, there are two possible orders in which the four bytes of a word can be stored in four memory addresses. A system that copies the "Little End" or **Least** Significant Byte of a register into the **lowest** address during a Store operation is said to use *Little Endian* ordering. Figure 3-3b shows the order in which the bytes would be copied to memory for Little Endian storage.

The easy way to remember how Little Endian ordering works is probably with the expression Little Endian = Low Byte @ Low Address. If, for example, a register contains 0x0144Fb89 and the processor executes an instruction to store the word starting in memory at 0x1020, the Least Significant Byte of 0x89 will be written to address 0x1020 as shown in Figure 3-3b. The next Most Significant Byte, 0x7B, will be written to address 0x1021, etc. Likewise, if the word is read from memory to a register, the byte from the lowest address will be written to the LSB of the register as shown in Figure 3-3b. Incidentally, the Intel desktop system type processors such as the Core i7 use Little Endian byte ordering.

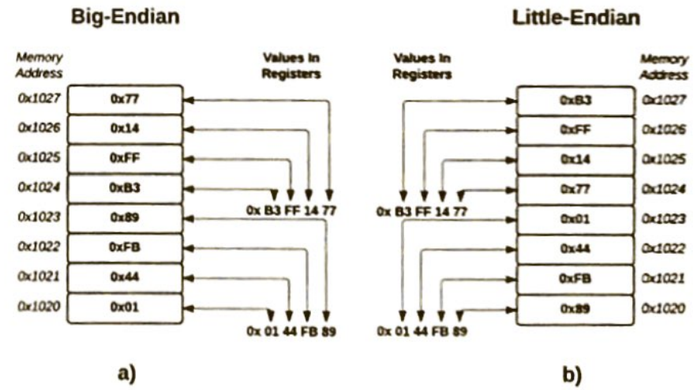


Figure 3-3 Byte ordering. (a) Big Endian ordering example. (b) Little Endian ordering example. (Improved art version here provided by Eric Krause)

Figure 3-3a shows how the bytes from a 32-bit register would be stored in memory for a machine that uses *Big Endian* ordering. As you can see, in this system the byte from the "Big End" or **Most** Significant Byte of the register is copied to the **lowest** address in memory during a Store operation. Likewise, a byte from the low address is put in the Most Significant Byte of a register when the word is read from memory to a register. Following the example in the last paragraph, you can write your own rule to help you remember how Big Endian ordering works. The Motorola MC68000 family of desktop type microprocessors were one common example of microprocessors that use Big Endian byte ordering. A key point is that, as shown in figure 3-3, the order of the words in memory is the same in both systems but the order in which the bytes are stored or read is different.

There have been many lengthy discussions as to whether Little Endian or Big Endian byte ordering is the better approach. As in the endless discussions about the better way to install a roll of toilet paper in the holder, there is no final agreement as to which is the better order, so both are commonly used. The ARM-based processors actually allow a system designer to specify either Little Endian byte ordering or Big Endian byte ordering. The default for many ARM-based processors such as the TI AM335X family of processors is Little Endian byte ordering, so we will use Little Endian ordering throughout this book.

Note that most of the time when you are working with a given system, you don't really care which ordering is used. In either case, when you write a word to memory and then read it back, you get back the same word you wrote. One case where you might need to know the byte order is the case where you are writing data to memory as words and reading data back one byte at a time for some operation on the bytes. Also, when debugging programs, it is useful to know which ordering is used in a system so you can better understand the memory display produced by the debugger. A third case where it is important to know which ordering is used is the case where you are looking at data bus signals with a logic analyzer. Since the data bus is 32-bits wide, all four bytes of a word will be on the data bus at the same time, but the bytes will be on different data lines for a Little Endian system than they will be for a Big Endian system. We will talk much more about program debuggers and logic analyzers later. For now, however, let's continue on our path through the ARM Instruction Set Architecture with a discussion of how input ports and output ports are set up in a system.

INPUT PORTS AND OUTPUT PORTS

When a microcomputer program instruction reads an input port, the data supplied by the port device is copied to a register in the processor. As mentioned in Chapter 2, an input port may be a simple set of parallel latches with their outputs connected to the data bus. When the latches are clocked by a read operation the data is transferred to the data bus and flows into the processor. An input port may also be a complex device such as an Ethernet interface IC. Likewise, when a microcomputer instruction writes a word to an output port, the word is copied from a register in the processor to the output port device on the data bus. Again, the output device may be a simple set of parallel latches or a complex programmable device such as a USB controller. In any case, each input port and each output port must have a unique address, so that only one port at a time is ever enabled. The two ways of setting up the addressing for I/O ports are called *Memory-Mapped I/O* and *Direct I/O*.

In a system using Memory-Mapped I/O, the memory system address decoder is designed to assign addresses to the I/O devices as if they were simply additional memory devices. In these

systems, an input port is read with the same instruction that is used to read a memory location and a word is written to an output port with the same instruction that is used to write a word to a memory location. The advantages of memory-mapped I/O are simpler address decoding and no need for separate input and output instructions. The disadvantage of memory-mapped I/O is that the I/O ports take up some of the available system address space and this space cannot then be used for memory devices. This lost address space is not a big consideration with current processors such as ARM-based processors that have 32-bit address buses and thereby have 4 GBytes of available address space. Therefore, ARM-based processors use memory-mapped I/O and do not have direct I/O capability.

Processors designed to implement direct I/O have a separate I/O address space in addition to the memory address space. Each address space has its own address decoders to assign addresses to the devices in its space. A special "IN" instruction is used to read a word from an input port in the direct I/O address space, and a special "OUT" instruction is used to write a data word to an output port in the direct I/O address space. The obvious advantage of direct I/O is that in a direct I/O system, I/O ports do not use up any of the system memory address space. The disadvantage of direct I/O, of course, is the need for an additional address decoder and the associated control logic. Note that a processor designed with direct I/O capability can also be used to implement memory-mapped I/O in addition to the direct I/O or in place of it. The Intel desktop system-type processors, such as the Intel Core processors , are examples of processors that have direct I/O capability as well as memory mapped I/O capability.

As we stated at the beginning of this section, when you are faced with writing programs for a microprocessor system that is new to you, the first step is to study the Instruction Set Architecture to get an overview of the features and capabilities of the processor. As you do this, it is good practice to summarize your findings in compact, digestible form in your design logbook. Before taking a look at an example ARM-based processor system, here's a summary of the features of the ARM Instruction Set Architecture (ISA). The ARM ISA has:

1. A 32-bit ALU, a 32-bit data path, and a Load-and-Store architecture where only the Load and Store instructions can access the data memory or I/O devices.
2. A data path register file with 15, 32-bit general purpose registers (R0-R14) and a Program Counter (PC).
3. Exceptions that include: two hardware interrupt inputs, FIQ and IRQ; a Software Interrupt Instruction (SWI) that can be used to call operating system procedures; and faults that change the processor operating mode when some error condition such as "undefined instruction" occurs during program execution.
4. A User operating mode and the six privileged operating modes: FIQ, IRQ, SVC, Abort, Undefined, and System. The operating modes FIQ, IRQ, SVC, Abort, and Undefined are entered in response to specific exceptions or faults. System mode is entered from one of the other privileged modes and is used for operating system code.
5. A Current Program Status Register (CPSR) with condition code bits (flags) to indicate Negative (sign), Zero, Carry, and Overflow; bits to indicate the operating mode (User or one of the six privileged modes); bits to represent the enable status of the FIQ and IRQ interrupt inputs; and a bit to indicate whether the processor is executing Thumb compressed instructions or regular 32-bit instructions.
6. Programmer selectable Little-Endian byte ordering or Big-Endian byte ordering for words stored in memory. The default is Little-Endian ordering.

7. Memory-mapped I/O port access only.

The next step as you are learning a new processor on the job is to is to study the specific physical system for which you will be writing programs so that you can determine: memory types and memory addresses; General Purpose I/O (GPIO) port connections and addresses; specialized I/O controllers and their addresses; and I/O configuration register addresses. The extensive memory map for the controllers provided by the Cortex-A8 in the Sitara AM3359 on the B3 board is shown in the AM3359 Technical Reference Manual at http://www.ti.com/lit/ug/spruh73n/spruh73n.pdf, starting on Page 175 in the current version. You can skim through these tables to get an overview of the many components in the processor. We will refer to this table to get needed addresses as we move through the interfacing sections of the book. Also, for future reference, the starting address for the system memory DRAM on the B3 board is 0x80000000. The TI Code Composer Studio tools that we describe later in the chapter are set to automatically load your low level programs starting at this address when set up for the BeagleBone Black board. Now that you have an overview of the ISA and the hardware for an ARM-based processor system, the next step here is to start showing you how to write programs for one of these systems.

INTRODUCTION TO ARM ASSEMBLY LANGUAGE PROGRAMMING

In the last chapter we reviewed how a microprocessor fetches a binary-coded instruction from memory, decodes the instruction, and executes it. These binary-coded instructions that the processor understands are commonly called the *machine language* of the processor. You could write a program for a microprocessor by simply constructing the required sequence of machine codes using the instruction templates, loading the machine codes into memory in some way, and executing the sequence of machine codes. (In fact, your grandfather or grandmother may tell you about how, in the old days, he or she had to set up each machine instruction for a microprocessor program with manual switches on the front panel and then press a button to enter that machine instruction into the processor's memory.) The immediately obvious problems with this machine-level programming are that it is very tedious, time consuming, and error prone. As microprocessors have become much more complex and programs have become much larger, we have moved more and more toward higher-level programming languages for writing microprocessor programs. For that reason, we will emphasize machine-language coding much less in this book than we did in our earlier microprocessor books.

The next step up from machine language programming is *assembly language* programming. In assembly language programming, you use more "English-like" instructions to identify each desired machine action and then you use a program called an *assembler* to translate the assembly language to the actual machine codes that the processor

The ARM Cortex A-8 uses the ARM v7-A instruction set and the complete manual for the instructions set is available at https://silver.arm.com/download/ARM_and_AMBA_Architecture/AR570-DA-70000-r0p0-00rel2/DDI0406C_C_arm_architecture_reference_manual.pdf with proper registration. This instruction set has several functional groups as follows:

Data Processing Instructions
Single Register Load and Store Instructions
Branch Instructions
Multiple Register Load and Store Instructions
Status Register Access Instructions
Exception Generating/Handling Instructions
Semaphore Instructions
Coprocessor Instructions
Advanced SIMD Instructions
Floating Point Instructions

Rather than showing you all of the instructions in each of the groups and discussing each, we will in this chapter concentrate on key instructions in the first three groups and then show some simple program examples using instructions from these groups. As with learning any new language, it is more productive to start with a few useful words and sentences than it is to try to memorize the entire dictionary for that language. As we move through the following chapters, we will introduce more instructions as needed for the programming examples. In the next chapter, for example, we will discuss the Multiple Register and the Status Register access and use instructions.

Opcode	Mnemonic	Operation	Action
0000	AND	Logical AND	Rd:=Rn AND shifter_operand
0001	EOR	Logical Exclusive OR	Rd:=Rn EOR shifter_operand
0010	SUB	Subtract	Rd:=Rn-shifter-operand
0011	RSB	Reverse Subtract	Rd:=shifter_operand-Rn
0100	ADD	Add	Rd:=Rn + shifter_operand
0101	ADC	Add with carry	Rd:=Rn + shifter_operand + Carry Flag
0110	SBC	Subtract with carry	Rd:=Rn – shifter_operand – NOT (Carry Flag)
0111	RSC	Reverse Subtract w/ carry	Rd:=shifter_operand – Rn – NOT (Carry Flag)
1000	TST	Test	Update flags after Rn AND shifter_operand
1001	TEQ	Test Equivalence	Update flags after Rn – shifter_operand
1010	CMP	Compare	Update flags after Rn – shifter_operand
1011	CMN	Compare Negated	Update flags after Rn + shifter_operand
1100	ORR	Logical (Inclusive) OR	Rd:=Rn OR shifter_operand
1101	MOV	Move	Rd:=shifter_operand (no first operand)
1110	BIC	Bit Clear	Rd:=Rn AND NOT (shifter_operand)
1111	MVN	Move Not	Rd:=NOT shifter_operand (no first operand)

Most data-processing instructions take two source operands, though Move and Move Not take only one. The compare and test instructions only update the condition flags. Other data-processing instructions store a result to a register and optionally update the condition flags as well.

Of the two source operands, one is always a register. The other is called a *shifter operand* and is either an immediate value or a register. If the second operand is a register value, it can have a shift applied to it.

CMP, CMN, TST and TEQ always update the condition code flags. The assembler automatically sets the S bit in the instruction for them, and the corresponding instruction with the S bit clear is not a data-processing instruction, but instead lies in one of the instruction extension spaces.

Table 3-1 ARM/XScale data processing instructions and 4-bit opcode values for each

For assembly language programming, you use 2 to 6 letter *mnemonics* to represent desired instructions. A mnemonic is a "shorthand" or abbreviation that helps you remember something. (The first "m" in mnemonic is silent, so you say the word as if it were spelled "nemonic".) If you look at the assembly language mnemonics for the 16 basic ARM data processing instructions in Table 3-1, you should be able to easily relate most of the instruction mnemonics to the specified operation. For future reference, Table 3-1 also shows the 4-bit binary "op-codes" that are put in the machine codes for each of these instructions by the assembler.

In ARM assembly language program statements, you use the register number to specify a desired source or destination register. The assembly language instruction ADD R3, R2, R1, for example, tells the assembler to produce the ARM machine code for an instruction that adds the value in register R1 to the value in register R2, and puts the result in register R3 without affecting R1 or R2. Note that the destination register, R3, is specified first, just after the ADD and before the source registers.

For future reference, Appendix B gives brief explanations and examples of all of the basic ARM/XScale assembly language instructions. However, the commented examples shown and discussed in the following sections here should enable you to easily understand the simple program examples we give later and to write some simple programs of your own. Note that we certainly do not expect you to absorb all of the information contained in these examples on your first or even second reading. You will gradually absorb the majority of this information when you cycle through it as needed for homework assignments and for writing your first simple programs.

ARM DATA PROCESSING INSTRUCTION EXAMPLES

Below are some first examples to show you how to write ARM data processing instructions and the MUL instruction for a desired operation. We suggest that you read through these in sequence because each adds a little more to your overall knowledge. Except as indicated, the assembly language syntax shown in these examples is the same for all of the data processing instructions in Table 3-2. Also note that any of the R0-R14 registers may be used in place of the registers we have used in these examples. Anything written after a "@" is a comment or explanation that is not part of the instruction. (Some assemblers use a ";" to identify a comment but we will use the @ symbol that is standard for the gnu program development tools we introduce a little later.) The # sign is used to indicate an "immediate" number that will be inserted directly in the machine code for an instruction.

```
ADD R3, R2, R1      @ Add R1 + R2 and put result in R3. Flags, R1 and
                    @  R2 are not affected.

ADD R3, R3, #4      @ The # symbol is used to indicate an "immediate
                    @  number" that is coded in the instruction. This
                    @  instruction then says, add 4 to the value from
                    @  R3 and put the result in R3, or in other words
                    @  increment R3 by 4. Note that only 8 bits are
                    @  available in the 32-bit instruction template
                    @  for the immediate number, but the immediate
                    @  number can be rotated to give larger
                    @  numbers. Flags are not affected.

ADDS R3, R2, R1     @ Add value from R1 to value from R2 and put
                    @  result in R3. The S on the ADD mnemonic tells
                    @  the assembler to code instruction so that
                    @  flags are updated based on result of the
                    @  addition.

ADC R3, R2, R1      @ Add R1 + R2 + value in Carry flag, result in R3.

ADD R5, R4,R3,LSL #3  @ R5 = R4 + contents of R3 shifted three bit
                      @  positions to the left. Left shift of N bit
                      @  positions multiplies a binary number by 2^N, so
                      @  R5 = R4 + 8 * (contents of R3).
```

```
ADD R5, R4,R3,LSL R2  ; R5 = R4 + contents of R3 shifted left the
                      @ number of bit positions specified in R2. If R2
                      @ contains the number 2, a left shift of 2 bit
                      @ positions multiplies a binary number by 4, so
                      @ R5 = R4 + 4 * (contents of R3).

AND R1, R2, R1        @ Logically AND the word in R1 with the word in R2
                      @ on a bit-by-bit basis and write the result to
                      @ R1. Used for masking bits as described in
                      @ chapter 2. Flags not affected because no S
                      @ after AND.

TST R1, R2            @ TEST - Logically AND the word from R1 with the
                      @ word from R2 on a bit-by-bit basis and update
                      @ flags based on result. R1 and R2 not changed.

ORR R1, R2, R3        @ Logically OR word in R3 with word in R2 on a
                      @ bit-by-bit basis and put result in R1. Used
                      @ for setting bits in a word as described in
                      @ chapter 2. Flags not affected unless S added
                      @ to ORR mnemonic.

CMP R1, R2            @ Compare R2 with R1
                      @ Subtract R2 from R1 and set flags as follows
                      @ R2 = R1          Z = 1, N = 0
                      @ R2 < R1          Z = 0, N = 0
                      @ R2 > R1          Z = 0, N = 1
                      @ R2 and R1 are NOT changed

ADDNE R2, R1, R3      @ ADD contents of R3 and contents of R1 and
                      @ put the result in R2, IF the Zero flag is not
                      @ set. ELSE do not execute the instruction.
                      @ This is an example of specifying conditional
                      @ execution. The specified condition can be any
                      @ of those represented by the Mnemonic
                      @ Extensions shown in Table 3-2.

MOV R3, R2            @ Copy the contents of R2 to R3. R2 not changed.

MOV R3, #0x20         @ Load the immediate number 20 hexadecimal into
                      @ R3. Note that the size of the immediate number
                      @ that can be directly moved into a register is
                      @ 8 bits, but later we show methods that can be
                      @ used to load larger numbers into a register.
```



```
MUL R8, R6, R7        @ Multiply the contents of R7 by the contents of
                      @ R6 and put lowest 32 bits of the product in
                      @ R8. Note that a product greater than 32 bits
                      @ will be truncated to just the lowest 32 bits.
```

Condition [31:28]	Mnemonic Extension	Meaning	Condition Flag State
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/Unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	V set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0, N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z ==1 or N != V)
1110	AL	Always (unconditional)	
1111	(NV)	Not Valid	

Table 3-2 Conditions that can be specified for conditional execution of an instruction and the code bits for each.

ARM LOAD AND STORE INSTRUCTION EXAMPLES

As stated in earlier discussions, the ARM-based processors are sometimes referred to as "Load-and-Store" machines. Other than a couple of exceptions, the only instruction that can read a word from the data memory is the Load instruction and the only instruction that can write a word to the data memory is a Store instruction. In this section we show examples of the ARM instructions that load a single data value from memory to a register and those that store a single data value from a register to the data memory. Contrary to the basic RISC philosophy of single function instructions, the ARM-based processors actually also have an instruction that loads a sequence of memory locations into a specified set of registers and an instruction that copies a specified set of registers to a sequence of memory locations. However, we will wait until the next chapter to show you how to use these.

The address specified for a Load or Store operation is commonly called the *Effective Address* or EA. As you will see in the following examples, there are a variety of ways that you can tell the assembler or processor to determine the EA for a memory access. Cycle through these examples until the basic ways of specifying a memory address are clear. You will use these often in your programs. Again note that any of the R0-R14 registers may be used in place of the registers we have used in these examples.

```
LDR R1, [R2]        @ The [] are shorthand for "the contents of the
                    @ memory location pointed to by." This
                    @ instruction then says, copy the word from the
                    @ memory address contained in R2 into R1. In
                    @ other words, R2 contains the Effective Address
                    @ (EA) for the memory access.

LDR R1, [R2, R5]    @ EA = R2 + R5. Copy word from memory at EA to R1.

LDR R1, [R2, #0x10]     @ EA = contents of R2 + 10 Hexadecimal. Copy
                    @ 32-bit word from memory at EA to R1. Value
                    @ in R2 is not changed.

LDRB R1, [R2]       @ Copy byte from EA in R2 to lowest byte position
                    @ in R1. Put zeroes in top 3 bytes of R1.

LDRSB R1, [R2]      @ Copy Signed byte from memory at EA in R2 to
                    @ lowest byte position in R1. Fill upper 3 bytes
                    @ with copies of the sign bit (MSB) of the
                    @ copied byte.

LDRH R1, [R2]       @ Copy Half word (16 bits) from memory at EA in R2
                    @ to R1. Upper two bytes of R1 filled with
                    @ zeroes.

LDRSH R1, [R2]      @ Copy Signed Half word from memory at EA in R2 to
                    @ R1. Upper two bytes of R1 filled with copies
                    @ of sign bit from half word.

LDR R1, [R2], #4    @ Copy word from memory at EA in R2 to R1 and THEN
                    @ add 4 to R2 so R2 points to next sequential
                    @ word in memory. This is an example of Post-
                    @ Indexed addressing mode because the address in
                    @ R2 is incremented after the access.
                    @ It is useful for sequentially accessing the
                    @ elements of an array.

LDR R1, [R2, #4]!   @ EA = R2 + 4. The "!" here specifies that the
                    @ processor should add 4 to the value in R2 and
                    @ use this as the EA for the access. This is an
                    @ example of the Pre-Indexed addressing mode
                    @ because the immediate number is added to the
                    @ value in R2 before memory is accessed. This
                    @ addressing mode is useful for accessing the
                    @ elements of an array of data values in
                    @ sequence. Value left in R2 = old R2 + 4.
```



```
STR R1, [R2]        @ Copy word from R1 to memory at EA contained in
                    @ R2.

STR R1, [R2, #4]    @Copy word from R1 to memory at EA computed by
                    @ adding 4 to contents of R2. R2 is not changed.

STRB R1, [R2, R3]   @ Copy byte from lowest byte position in R1 to
                    @ memory at EA computed by adding value in R2
                    @ and value in R3.

STR R1, [R2], #4    @ Copy word from R1 to EA in R2 and then add 4 to
                    @ R2 so R2 points to next sequential word in
                    @ memory. This is an example of Post-Indexed
                    @ addressing mode because the immediate number
                    @ is added to the value in R2 after the value in
                    @ R2 is used to access memory. This address mode
                    @ mode is useful for storing the elements of an
                    @ array.

STR R1, [R2, #4]!   @ The "!" here specifies that the processor should
                    @ compute the EA by adding 4 to the contents of
                    @ R2. The value left in R2 will be the old value
                    @ + 4. This is an example of Pre-Indexed
                    @ addressing mode. It is useful for storing
                    @ the elements of an array of data values in
                    @ sequence.
```

ARM BRANCH INSTRUCTION EXAMPLES

The ARM-based processors have both *unconditional* branch instructions and *conditional* branch instructions. When the processor executes an unconditional branch instruction, it always fetches the next instruction from the specified target address. If the specified branch condition is true when the processor executes a conditional branch instruction, the next instruction will be fetched from the specified target address. If the specified branch condition is not true when a conditional branch instruction is executed, the branch will not be taken and the next instruction directly after the branch instruction will be executed. Any of the conditions listed in Table 3-2 can be used to specify the branch condition for a conditional branch instruction. The following examples will show some of the ways used to specify the target address in conditional and unconditional branch instructions.

```
B label             @ Unconditional Branch to the instruction that has the specified name or
                    @ label in front of it. (We will show an example of this a little later
                    @ in the chapter.) The assembler calculates the correct number
                    @ of words from the Branch instruction to the Branch target address and
                    @ inserts this count into the Branch instruction code as a signed number.
                    @ Since the Branch instruction has 24 bits reserved for this program
```

@ *counter relative* address, the target address for this type PC relative
@ branch can be up to about 2^{23} bytes backward from the current PC or
@ 2^{23} bytes forward from the current PC.

BNE label @ Branch to the address identified with the specified label if the Zero
@ flag is not set. Often used after a Compare instruction such as CMP
@ R1, R2 to send execution to the specified label, if R1 is not equal to
@ R2.

BL function @ Branch to function (procedure) and Link for return. Call the specified
@ function by Branching unconditionally to the first instruction in the
@ specified function. Link to the calling program by copying address
@ of the next instruction after the Branch instruction into R14, which is
@ also known as the *Link Register* or LR. As shown in the next example,
@; the value saved in the Link Register is used to return execution to the
@ calling program at the end of the function. In the next chapter, we
@ show several examples of writing and calling assembly language
@ functions.

MOV PC, LR @ Copy value in Link Register (R14) into the Program Counter
@ (PC). This instruction is often used at the end of a function
@ (procedure) to return execution to the instruction after the BL
@ instruction that called the function.

SIMPLE ARM ASSEMBLY LANGUAGE PROGRAM EXAMPLES

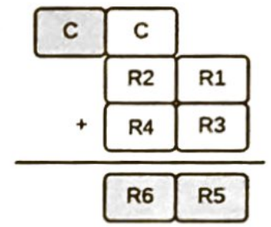
In the preceding section, we introduced you to the basic syntax or, in other words, the "vocabulary and sentence structure" of ARM assembly language. The exercises at the end of the chapter will give you practice in predicting the results of given instructions and in writing single or small groups of assembly language instructions that will perform a specified operation. Next here we show and discuss a couple of simple but complete assembly language program examples. In addition to demonstrating the use of assembly language instructions, these examples also show you how to format your assembly language programs for use with an assembler program. In the next chapter we will show you how to efficiently write assembly language programs for a variety of applications.

```
@ First example of ARM Assembly Language Programming.
@ Unsigned addition
@ This program adds a 64-bit number loaded into R2-R1 to a 64-bit number
@ loaded into R4-R3 to produce a correct 64-bit sum in R6-R5 and Carry
@ Uses R1-R6 and Cary
@ Douglas V. Hall September 2016

.text                                   @ Indicates instructions follow
.global _start                          @ makes label _start accessible globally

_start:                                 @ Identifies start of your code
            MOV R2, #0x40               @ Put values in working registers
            MOV R1, #0x80
```



```
            MOV R4, #0x30
            MOV R3, #0x50
            ADDS R5,R1,R3               @ Add low words in R1 and R3, put low word of
                                        @ result in R5. Carry set if carry produced
            ADDCS R6,R2,R4              @ Add high word in R2 and R4 plus status of
                                        @ C flag. Result in R6 and C flag.

.END                                    @ Indicates end of program.
```

(a)



(b)

Figure 3-4 Assembly language file for a simple program that adds two 64-bit numbers. (a) Source code for program. (b) Diagram showing register usage (Added by Eric Krause for clarification).

Figure 3-4a shows the ARM assembly language *source file* for a simple program that adds two 64-bit numbers, 32 bits at a time, to produce a 64-bit sum. Whenever you are faced with analyzing a program you have not seen before, the first step is to read the header description of the operations that the program performs. Note that when you write a program, you should always include a *header* that describes the function of the program, lists the program inputs and outputs, and gives any other information that would help a reader understand the program. As shown, comments can start in any column of the source program, so the header can use the full page width.

The second step in analyzing a new program is to skim through the entire program to find the major blocks and get a feeling for the overall structure of the program. To help with this understanding, it is good idea to read the through the comments section without digging into the details of the actual instructions. If the program author did a good job of writing the comments, they should give you an understanding of the operation of the program at a higher level than that provided by reading the detailed instructions.

The third step in analyzing a program is to work through the actual program instructions, one step at a time. As you read the program in Figure 3-4, first note the words such as .text, .global, and .end, that have a period in front of them. Terms that have a period in front of them are called *assembler directives*. They tell the assembler program how to assemble your program. The .text directive, for example, tells the assembler that the assembly language instructions will be found in the following statements. The .global directive tells the assembler that the label _start should be allowed to be accessed from other program modules. The .end directive tells the assembler that this is the end of the source statements for this program module. Now let's look at the program comments and work through the actions in the program.

The first two MOV instructions in the program put test values for the first double word (64-bit number) in R2 and R1. (Remember that the # sign is used in an instruction to tell the assembler to insert the specified (immediate) number directly in the machine code for the instruction. The second two MOV instructions put a 64-bit test value in R4 and R3. In a more realistic program, these values would most likely be loaded from a data structure in memory. However, since we wanted to keep this first example as simple as possible, we just loaded some small, immediate numbers directly into registers with MOV instructions.

As indicated by the comment, the ADDS R5, R1, R3 instruction in the example program then adds the lower words (32 bits) of the desired double words and leaves the result in R5 as shown in Figure 3-4b. Because the ADD instruction has the S suffix, the flags will be updated when the instruction executes. For example, if the addition produced a Carry, the C flag will be set. The ADCS R6, R2, R4 instruction next in the program will then add the upper words of the two double word plus the status of the C flag and put the result in R6, as shown in Figure 3-4b. Because this instruction also has an S suffix, the flags will be updated when the instruction executes. Therefore, if the second addition produced a Carry, the C flag would be set. Now that you have gained some familiarity with the basic structure of an assembly language program, let's take a look at a more interesting and more realistic example program.

Figure 3-5 shows an assembly language program that multiplies each element in one array of 16-bit half words by the corresponding element in another array of half words and stores the product in the same numbered element in an array of words. (Remember that if you multiply a 16-bit binary number by a 16-bit binary number, the product can be as large as 32 bits, so a 32-bit word must be reserved in memory for each product.) After you read the header, the next step in understanding a data processing program such as this is to look closely at the variables and other data structures for the program.

```
@ Array Multiply Program
@ This program multiplies each half word from the Multiplicands
@ array by the same numbered half word in MULTIPLIERS array and
@ puts the result in the same numbered element in PRODUCTS array.
@ Uses R1-R4, R6-R8
@ Douglas V. Hall September 2016

.text
.global _start
_start:
.equ   NUM, 4
            LDR R1,=MULTIPLICANDS     @ Load pointer to MULTIPLICANDS array
            LDR R2, =MULTIPLIERS      @ Load pointer to MULTIPLIERS array
            LDR R3, =PRODUCTS         @ Load pointer to PRODUCTS array
            MOV R4,#NUM               @ Initialize loop counter
NEXT:  LDRH R6,[R1]                   @ Load a MULTIPLICAND Half word in R6
            LDRH R7,[R2]              @ Load a MULTIPLIER half Word in R7
            MUL R8, R6, R7            @ Multiply
            STR R8, [R3]              @ Store result in PRODUCTS array
            ADD R1,R1,#2              @ Increment MULTIPLICANT pointer to next
            ADD R2,R2,#2              @ Increment MULTIPLIER pointer to next
```



```
            ADD R3,R3,#4              @ Increment PRODUCTS pointer to next
            SUBS R4,#1                @ Decrement loop counter by 1
            BNE NEXT                  @ Go to NEXT if all elements not multiplied
            NOP                       @ Instruction for breakpoint. Does nothing.


.data

MULTIPLICANDS: .HWORD 0x1111, 0x2222, 0x3333, 0x4444
MULTIPLIERS:   .HWORD 0x1111, 0x2222, 0X3333, 0x4444
PRODUCTS:       .WORD  0x0, 0x0, 0x0, 0x0
.END
```

Figure 3-5 Assembly language program that multiplies each element in one array of half words by the corresponding element in another array of half words and stores the product in the same numbered element in an array of words.

The .data assembler directive near the bottom of the program in Figure 3-5 tells the assembler that the program statements following the directive contain data declarations. In the final version of a system, this data would likely be written in Read/Write memory so the data could be dynamically changed during program executions. The MULTIPLICANDS: .HWORD 0x1111, 0x2222, 0x3333, 0x4444 statement declares an array variable named MULTIPLICANDS indicates that the variable is of type half word, and tells the assemble to initialize the four elements of the array with the specified half word values. Likewise, the statement MULTIPLIERS: .HWORD 0x1111, 0x2222,0x3333,0x4444 declares a half word array variable named MULTIPLIERS, and tells the assembler to initialize the four elements of the array with the four hexadecimal values shown. The PRODUCTS: .word 0x0,0x0,0x0,0x0 statement declares a word (32-bit) type array variable named PRODUCTS, and initialize the four elements of the array with zeros.

Next, read through the comments in the program in Figure 3-5 to get a feeling for the sequence of actions or *algorithm* for the program. The terminology and the algorithm are very "English-like" and are very similar to the method you would use to manually process the data arrays one element at a time. We start by loading the addresses of the three arrays into registers, so we can use the registers to access the elements of the arrays. Addresses such as these are commonly called *pointers*. Next we initialize a counter with the number of elements in the array. This will be used to keep track of how many elements have been processed and stop when done. The next section of code, from the NEXT label to the BNE NEXT instruction, forms a program loop that repeats until all of the elements in the MULTIPLICANDS AND MULTIPLIER arrays are processed. As shown by the comments, the specific steps are: get two half word operands, do the multiplication, store the result, increment all three pointers to point to the next elements in the three arrays, decrement the counter, and determine if all of the elements have been processed. Now let's work though the details of the instruction column in the program.

The .EQU NUM,4 statement just after the _start: label near the beginning of the program is an assembler directive that tells the assembler to create a constant named NUM that has the value 4. Once a constant is defined with an EQU directive, you can refer to it by name anywhere in your program. The MOV R4, #NUM statement in the program, for example, will load the number 4 into R4. The advantage of defining a constant with an EQU is that, if the constant is used several times in a program, you can change the value at once in all of the instructions where the value is used by simply changing the EQU value and re-assembling the program. Without the

EQU approach, you would need to go through the program by hand, try to find all of the instructions where that particular constant was used, and change each individually.

The "=" sign in the LDR R1, =MULTIPLICANDS statement at the start of the program is a directive to the assembler to determine the address of the start of the MULTIPLICANDS array and generate an instruction code that will load this address into R1. If the MULTIPLICANDS array were in the same program section as the LDR instruction and close enough to the LDR instruction, the assembler would replace the LDR instruction with an "ADD R1, PC, Offset" instruction, where Offset is an immediate number equal to the number of bytes from the value in the PC to the start of the MULTIPLICANDS array. If the MULTIPLICANDS array is in a different program section from the LDR instruction, as in our example where it is in .data, the assembler will use an indirect approach to load the register. The reason the indirect approach is used is that in most cases, an ARM-based processor can only load a 32-bit value into a register from another register or from a memory address. For the cases where the 32 –bit address cannot be loaded directly, the assembler determines the absolute address of, for example, the MULTIPLICANDS array, and sets aside four byte-sized memory locations in the .text program section to hold that address. The memory set aside by the assembler to hold addresses such as these is called a *literal pool*. As we discuss later in the chapter, the assembler will calculate the offset from the LDR instruction to the location of the MULTIPLICAND address in the literal pool and code this offset into the LDR instruction. When the LDR R1 instruction executes, the address will be copied from the literal pool into R1. The assembler will also generate entries in the literal pool for the LDR R2, =MULTIPLIERS instruction and for the LDR R3, =PRODUCTS instruction. (Don't worry if this is not 100% clear to you at this point. We will come back to it a little later and use a debugger display to show exactly how the assembler sets up and uses the literal pool.)

The LDRH R6, [R1] instruction loads a halfword from the address "pointed to" by R1 into the lower half of R6 and fills the upper half of R6 with zeros. Likewise, the LDRH R7, [R2] instruction loads the halfword pointed to by R2 into the lower half of R7 and fills the upper half of R7 with zeros. The MUL instruction multiplies the specified word registers and leaves the lower 32 bits of the result in R8. Multiplying a 32-bit word by a 32-bit word could produce a 64-bit product but the basic multiply instruction, MUL, always truncates (discards) the upper 32 bits. For our example here, we are multiplying two half words, so the result can never be more than 32 bits. Therefore the truncation of the upper 32 bits is not a problem. For cases where the full 64-bit result is needed, you can use the UMUL unsigned multiply instruction or the IMUL signed multiply instruction, if these are available on the particular ARM-based processor you are using.

After the MUL instruction produces the product in R8, the STR R8, [R3] instruction copies the result to the memory location pointed to by R3. Since Register R3 points to a location in the PRODUCTS array, the word will be copied to the Products array. The next three instructions increment the three pointers to point to the next elements in each of the three arrays. Note that R1 and R2 are incremented by 2 to point to the next halfword in the MULTIPLICANDS and MULTIPLIERS arrays. R3 is incremented by 4 because the elements in the PRODUCTS array are 4-byte words.

The SUBS R4, R4, #1 instruction decrements our loop counter by 1 and, since it has the "S" suffix, updates the condition flags. If the result of the subtraction leaves zero in R4, the Z flag will be set. The BNE NEXT instruction, which can be thought of as "Branch if Not Equal zero", will cause execution to loop back to the instruction at the NEXT label if the Z flag is not set. If

the Z flag is set, indicating a zero result after the subtraction, execution will exit the loop. Note that words such as _start: and NEXT: that are followed by colons are called *labels*. Labels are used to identify addresses. The _start: label, for example identifies the starting address for the program and the label NEXT: indicates the target address for the BNE NEXT instruction. Labels always start in the very first column of the source program.

In this section, we have given you a fairly solid introduction to basic ARM assembly language programming. Assembly language allows you to represent desired program actions with English-like mnemonics and syntax instead of specifying the binary coding for each instruction. Even though you will not be writing your programs at the binary code level, it is very useful in writing efficient programs and in debugging to understand the format of the binary codes for the instructions on a processor. Therefore, before we go on to discuss high-level language programming, which is even further away from the bit-by-bit coding, we will take a short side-trip and show you how the machine level codes for ARM processor instructions are built.

ARM Machine Language Programming and Instruction Code Templates

Figure 3-6 shows the machine language coding template for the 32-bit ARM processor data processing instructions and, for reference, Appendix A shows the coding templates for all of the ARM-based processor instructions. You do not need to memorize these templates, but when debugging programs it is very useful to understand how they are set up, so you can understand the coding of an instruction when you have to do so. To introduce you to how these templates are set up, here are brief descriptions for the bit fields in the data processing instruction template. Note that bits 26 and 27 in these instructions are fixed at 0.

A. Most of the ARM instructions can be coded to execute or not execute based on some specified condition of the flags in the Current Program Status Register (CPSR) when the instruction is decoded. Table 3-2 shows the conditions that can be specified and the code that would be put in the condition field (bits 28-31) of an instruction for each. If, for example, and ADD instruction is written for example as ADDCS R2, R3, R4, the CS after the ADD mnemonic tells the assembler to put 0010 in bits 28-31 of the machine code for the ADD instruction. When the processor decodes the ADD instruction, the specified addition will only be performed, if the Carry flag in the CPSR is Set. If the Carry flag is not set, the ADD instruction will just be passed through the processor pipeline without doing the ADD operation, changing the specified destination register, or updating the flags.

B. The *opcode* field (bits 21-24) in the template specifies the arithmetic or logic operation to be performed. Table 3-2 shows the ARM data processing instructions and the 4-bit opcode that is used to specify each operation.

C. The S bit (bit 20) indicates whether the N, C, V, and Z flags in the CPSR should be updated or left unchanged as a result of the execution of this instruction.

D. The field labeled Rn (bits 16-19) in the template is used to identify one source register, the first operand, for the operation. The 4-bit code for the desired number register is put

in these bits. For the instruction ADD R7, R2, R4, Rn would be R2, so 0010 would be put in these bits.

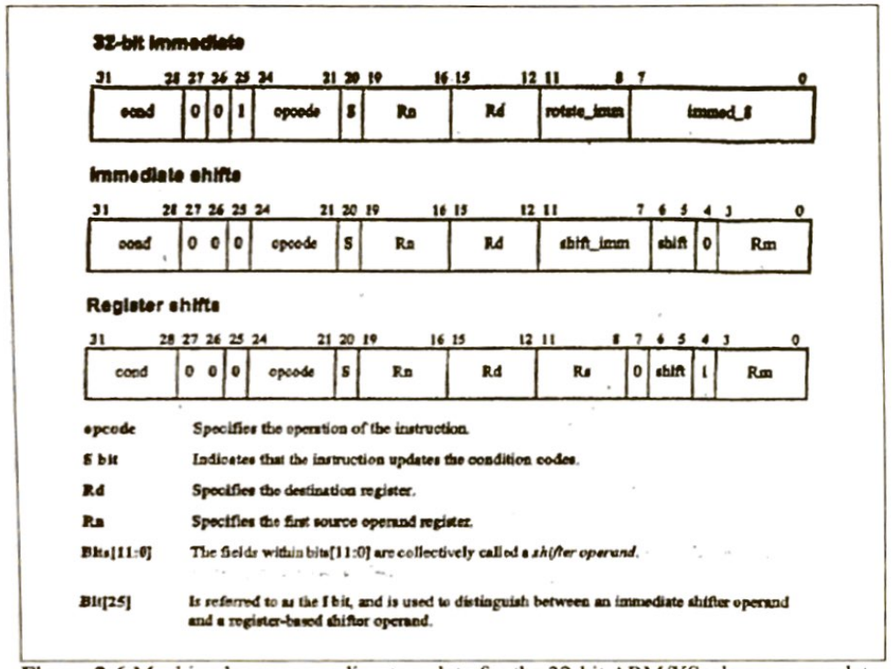


Figure 3-6 Machine language coding template for the 32-bit ARM/XScale processor data processing instructions.

E. A second source operand can be another register or an *immediate* number specified in the instruction. Bits 0-11 are used to specify the source of the second data operand. If the I bit (bit 25) in the instruction is a 1, then the second operand is an 8-bit immediate number contained in the lower 8 bits of the bit 0-11 field as shown in the top template in Figure 3-6. This number can be rotated an even number of bits, as specified by the upper 4 bits (bits 8-11) of the bit 0-11 field. If bit 25, the I bit, in the instruction is a 0, the second operand for the data operation is a register, Rm. The 4-bit code for this register is put in bits 0-3 in the instruction as shown in the second and third templates in Figure 3-6. The operand from the second register can be shifted or rotated by some number of bits specified as an immediate number in bits 7-11 of the bit 0-11 field as shown by the second template in Figure 3-6, or by the number of bits contained in the lower 8 bits of a register, Rs, identified with its 4-bit code in bits 8-11 of the bit 0-11 field as shown in the third template in Figure 3-6.

F. The result of the operation is put into a specified destination register, Rd, which can be the same as one of the source registers or a different register. The 4-bit code (number) for the destination register is put in bits 12-15 of the instruction.

As a first specific example of coding an instruction, suppose that you needed an instruction to simply AND the contents of register R3 with the contents of register R4, put the result in register R5, and set the flags according to the result. The assembly language for this instruction would be ANDS R5, R4, R3. To develop the code for this instruction, we will follow the second template in Figure 3-6 because, all the specified operands for the instruction are registers, the specified instruction does not contain an immediate number, and there is no shift specified in the instruction. Using this template the machine code for the ANDS R5, R4, R3 instruction would be:

```
1110 00 0 0000 1 0100 0101 00000 00 0 0011
  |   | | |  | |    |    |     |  | |  |__ Rm =R3 = second source register
  |   | | |  | |    |    |     |  | |__ Fixed
  |   | | |  | |    |    |     |  |__ No Shift or rotate
  |   | | |  | |    |    |     |__ Shift_imm value = 00000
  |   | | |  | |    |    |__ Rd = R5, destination register
  |   | | |  | |    |__ Rn = R4, first source register
  |   | | |  | |__ S = 1, Update flags
  |   | | |  |__ Opcode for AND operation = 0000
  |   | |  |__ I = 0, second operand is register specified in Rm bits
  |   |  |__Fixed
  |__Cond = 1110, unconditionally (always) execute AND instruction
```

As a second example of coding an instruction, suppose that you want to move the immediate binary number 1000000 into R7 if the zero flag is set but not update the flags based on the move. The assembly language for this instruction would be MOVEQ R7, #0x40. Since this instruction contains an immediate number, we use the first template in Figure 3-6 to determine the code for this instruction. The resulting code is as follows:

```
0000 00 1 1101 0 0000 0111 0000 01000000
  |   | | |  | |    |    |     |      |__ 8-bit immediate number
  |   | | |  | |    |    |     |__ Number bits to rotate number left (must be even)
  |   | | |  | |    |    |__ Rd = R7, destination
  |   | | |  | |    |__ No source register, so default to 0000
  |   | | |  | |__ S = 0, do not update flags
  |   | | |  |__ Opcode for MOV operation
  |   | |  |__ I = 1, operand is a specified 8-bit immediate number
  |   |  |__Fixed
  |__Cond = 0000, execute instruction only if Zero flag set
```

Some questions at the end of the chapter will give you a little more practice with developing machine codes for ARM instructions but in the next section here we discuss high-level language programming that lifts you even further up from the bit-by-bit machine language level than

assembly language. Then in the following section we summarize the uses and tradeoffs for the different levels of programming languages.

ARM High-Level Language Programming

Even though there has recently been some movement to use the Java high-level language for embedded application programming, there is a vast amount of embedded application programs in the C language. The C language is still the most commonly used high-level language for embedded application programming. Therefore, at this point we will show a very simple C program to give you a comparison of C programming with assembly language programming. Even if you haven't had a specific course in C programming, you should be able to follow the discussion for this basic program.

```
/* C program to multiply two arrays */
/* of short ints */

unsigned short MULTIPLICANDS[] ={0x1111,0x2222,0x3333,0x4444};
unsigned short MULTIPLIERS[] = {0x1111,0x2222,0x3333,0x4444};
unsigned int PRODUCTS[] = {0x0, 0x0, 0x0, 0x0};
int index;

int main()
{
    for (index =3; index >=0; index --)
    PRODUCTS[index] = MULTIPLICANDS[index]*MULTIPLIERS[index];
}
```

Figure 3-7 C program that implements the same algorithm as that implemented by the assembly language program in Figure 3-5.

Figure 3-7 shows a C program that implements essentially the same multiplication algorithm as implemented by the assembly language program in Figure 3-5. (Technically, the C program processes the array elements from highest to lowest instead of from lowest to highest and uses the index value in computing pointers as well as using it as a counter, but the results are the same for the two programs.)

In C programs, comments are enclosed by /* */ as shown at the top of the example program. The data type "unsigned short" is used to identify 16-bit unsigned quantities or halfwords, so the statement unsigned short MULTIPLICANDS[] ={0x1111,0x2222,0x3333,0x4444}; declares an array of four halfwords and initializes the four elements of the array with the values shown. Likewise, the statement unsigned short MULTIPLIERS[] = {0x1111,0x2222,0x3333,0x4444}; declares an array of four halfwords and initializes the elements of the array with the values shown. The statement unsigned int PRODUCTS[] = {0x0, 0x0, 0x0, 0x0}; declares an array of unsigned 32-bit quantities or words and initializes each of the elements of the array with all zeroes. The int index; statement declares an integer variable named index. The default for the int data type in ARM programming is a 32-bit signed number.

The int main() statement next in the program identifies the start of a function called main. Every C program must have a main function but will also likely have many other functions. The

action in the main function is contained between the curly braces {} shown on the left side of the program. The action shown in the function is a simple FOR-DO loop. Expanding this into even more English-like words, the two statements specify the following actions:

```
        Set the value of index to 3
Repeat
            Multiply the value indexed in MULTIPLICANDS times
                the element indexed in MULTIPLIERS
            Put the product in the indexed element of PRODUCTS
            Decrement index
Until index no longer >= 0
```

As you can see from this simple example, the C program specifies the desired actions at a "higher level of abstraction" than the equivalent assembly language program. In the C version, you don't need to keep track of register usage, incrementing pointers by the correct amount, etc. In the C program, you just specify the data types, data structures, and actions. The compiler then handles the low-level details as it creates the machine codes for your program.

Based on these last two statements, the logical question would be, "If I write my programs in C or in some other high-level language, the compiler handles most of the low-level details, so why would I ever want to write any programs in assembly language?" An analogy that comes to mind to give an answer to this question is the process involved in preparing the land around a new house for planting a lawn. You use a large bulldozer to do most of the leveling because it is much more efficient at moving a large amount of dirt in a short time. However, you use a less efficient hand rake to give the smoothest final result. Here's how this relates to the choice of a programming language for a particular program or program section.

You will usually use a high-level language to develop the majority of the program for an application because programming in a high-level language is much faster and more efficient than programming in a low-level language. Common folklore in the industry is that a good programmer can produce an average of about 10 lines of fully tested, debugged, and documented code a day. Ten lines of high-level language code can obviously specify much more work than 10 lines of assembly language code. The tradeoff here is that, if you program only with a high-level language, you are "stuck" with the machine code produced by the compiler. Compilers have improved a great deal in the last 20 years and current compilers do allow you to specify several types of optimization in the machine code produced. However, the code produced by the compiler may not be optimal for your particular application or at least for sections of your application. You can often improve the performance of these sections by rewriting them in assembly language. The overall approach we use is to write as much as possible of a program in a high-level language, then test and debug the program. Once the program is working, we use a tool called a *Profiler* to identify modules of the program where most of the program execution time is spent. Since these sections occupy a large percentage of the execution time, it may be worthwhile to attempt to optimize them. We also look for any sections that are not executing efficiently enough to generate the proper handshake signals with external circuitry. We then recompile the program and tell the compiler to compile the program to assembly language instead of directly to machine language. Next we analyze the compiler-generated assembly language for each of the sections we have decided to optimize and try to find ways we can improve the efficiency of the code for each of these sections. After rewriting the assembly

language for these sections as separate assembly language procedures, we rebuild, run, and debug the program. Creating truly optimal code for a modern, pipelined processor often requires both considerable thinking and considerable experimentation. We often cycle through these steps a number of times. The key point is that, to efficiently write a reasonably optimal program for one of the current microprocessors, you need knowledge of a high-level language, of assembly language, of the processor architecture, and of the program development tools.

One additional point about assembly language level programming is that it is often very useful and efficient for writing code sections that have to run at the fastest possible speed, the instructions for device initializations, the instructions for interrupt sections of low-level driver programs, and for some operating system sections. Furthermore, it is very important for you to have an understanding of assembly language instructions in order to efficiently use debuggers and In-Circuit Emulators (ICE units) units on programs that work with low-level hardware.

Later in this chapter we show you how to efficiently develop assembly language programs or program sections. Then in the next chapter, we give you some practice writing assembly language programs and discuss some of the basic optimization techniques you should keep in mind when you write assembly language programs. Also, in the next chapter we show you how to develop programs that contain both C and assembly language modules. Next here, however, we will introduce you to computer-based tools you can use to develop programs in assembly language, in a high-level language, or in a combination of the two.

Arm Program Development Tools

Most of the computer-based tools used to develop programs for ARM-based processor systems are *cross-development* tools. The term "cross development" means that the tools run on a computer different from the target system. The ARM Development System (ADS) program development tools, for example, are designed to run on PCs with some version of the Microsoft Windows operating system, on Unix systems, or on Linux systems. The major reason for using cross-development tools is that the tools run much more efficiently on the desktop systems with their large storage and computing power than they would in the more limited environment of the embedded system being developed. Also, during development, the system under development is unlikely to be able to run development tools. Here's an introduction to how you use cross-development tools to create programs

PROGRAM DEVELOPMENT TOOL CHAINS

The set of computer-based tools you use to develop a functioning program is commonly called a *tool chain*. ARM Ltd. and Texas Instruments are examples of companies that sell tool chains for developing ARM system programs. Also available for this are the open-source, GNU tools. These tools are available free and are widely used both in industry and educational environments. In fact, most of the commercially available ARM processor tool chains are basically GNU tools with added Graphical User Interfaces (GUIs) and some other features that represent the "value added" for the companies that sell them. The tool chains with GUIs are commonly referred to as *Integrated Development Environments* or IDEs, because you access all of the tools through menus that are available from a single window on the screen, rather than typing in commands at a command prompt. For this book we will use the Texas Instruments Code Composer Studio IDE that works with the B3 board and the Blackhawk ICE. It is available

for free for educational use from http:///www.ti.com/ccs. Available separately from this book is a tutorial that shows an introduction to the installation and use of the TI CCS IDE for simple program development. The next step here is to give an overview of how you develop, run and debug a program a tool chain such as CCS.

USING A DEVELOPMENT TOOL CHAIN

Figure 3-8 shows the general GNU tool chain and the steps you take to develop programs for microprocessor-based systems using this tool chain. Note that except for possibly a few minor name changes, the process and tools are essentially the same for any program development tool chain.

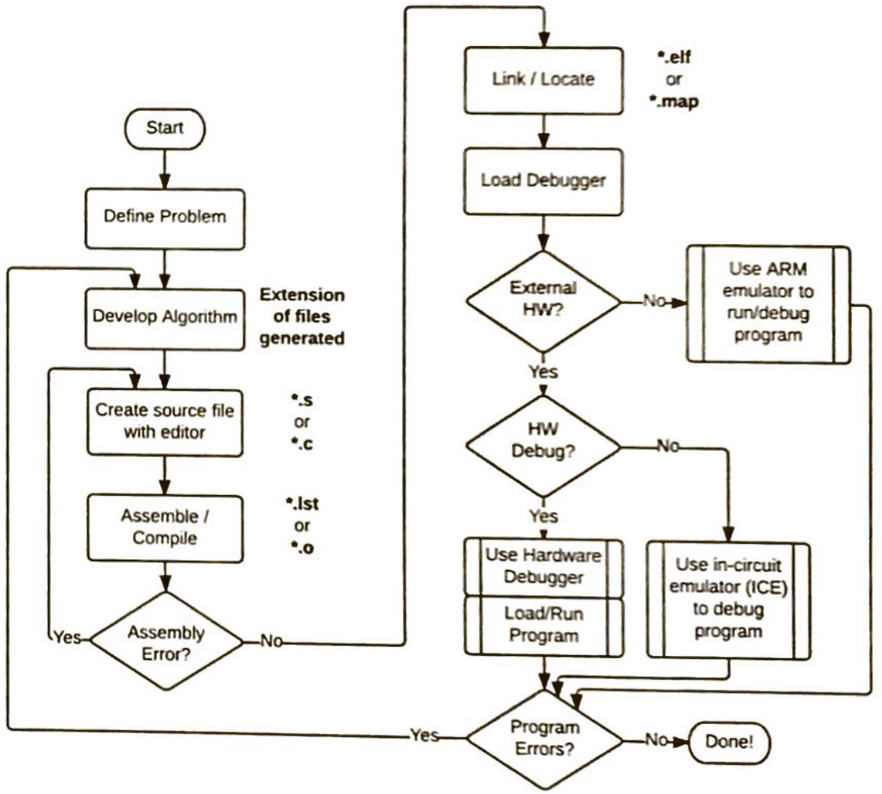


Figure 3-8 GNU tool chain & steps taken to develop programs for microprocessor systems.

As we will describe in greater detail in a later section of the chapter, the first and most important step in developing any type of program is to very carefully define the problem you are trying to solve or the task you are trying to accomplish with the program. This is illustrated in the first block (after Start) in Figure 3-8. The next step is to solve the problem or describe the desired action(s) at a high level with just words and pictures, rather than thinking about it in a particular programming language. Once you have a high-level English-like statement of the solution

(algorithm) for the problem, the next step is to translate the high-level, algorithm solution of the problem to an appropriate programming language. You then use an editor program to create a *source file* that contains the high-level language or assembly language instructions for your program. C program source files are given a .c extension, as for example mult.c, and assembly language source files are given a .s extension, as for example add64.s. For future reference, Figure 3-8 shows both the C program path and the assembly language path, but for now we will just lead you down the assembly language path. Note that for creating source files for use with the GNU tools, it is important that you produce files that do not contain embedded formatting characters. Formatting characters confuse the compiler and produce a lot of error messages. If you use an editor such as Notepad in Windows, you have to save the source file as type "All files" to prevent the insertion of formatting characters. The Code Composer Studio editor we use automatically generates the correct format file.

Assuming you are developing a simple assembly language program, the next step after creating the source file is to assemble your program. The complete manual for the CCS assembler is the Assembly Language Tools V5.1 Users Guide at http://www.ti.com/lit/ug/spnu118l/spnu118l.pdf . If there any syntax errors in your source file, the assembler output will display lines that contain errors. If there are errors, you cycle through the edit-assemble loop until there are no more errors.

Once there are no errors that the assembler can identify, the assembler produces an object (.o) file that contains the actual machine codes for your program and if you tell it to do so, information that will be used by a debugger to run and debug your program. You can also tell the assembler to produce a list (.lst) file. The list file contains the program source code, machine codes, and relative addresses. Since the list file allows you to see how the assembler treated your program instructions and assembler directives, it is a good idea to produce a .lst file and read through the file, to make sure the assembler produced the result you intended, before you go to the next step in the tool chain. Figure 3-9 shows a slightly-edited printout of the .lst file for the ARM assembly language program in Figure 3-5. Let's take a look at this to see how the values assigned by the assembler relate to the assembly language instructions, data structures, and assembler directives in the source program.

The first column in the list file in Figure 3-9 contains the statement (line) numbers assigned by the assembler. If a program contains assembly errors, the assembler will use these statement numbers to indicate which statement(s) contain errors. The numbers in the second column in Figure 3-9 are *location counter* values that indicate the location of each instruction or data item in its program section. For example, the location counter value for the LDR R1,=MULTIPLICANDS instruction is 00000000 because this instruction is the first item in the .text section. Since each ARM instruction is four bytes, the location counter value for the next instruction, LDR R2,=MULTIPLIERS, is 00000004. Likewise, the location counter value for the MULTIPLICANDS declaration in the .data section is 00000000 because this is the first item in that section. The MULTIPLICANDS array requires 8 bytes, so the location counter value for the MULTIPLIERS array, the next data item in the .data section, is 00000008.
The third column in the list file contains the actual machine code for each program instruction or, in a data section, it contains the data values. We will dig into these deeper when we look at the debugger printout but for now, note that the bytes shown in this column are in Little-Endian byte order. The entry 2C109FE5 actually represents an op code of 0xE59F102C. The byte 2C is at location 0x00000000, the byte 0x10 is at location 0x00000001, etc.


```
ARM GAS  ../mult1.s

   1 @ Array Multiply Program
   2 @ This program multiplies each half word from the Multiplicands
   3 @ array by the same numbered half word in MULTIPLIERS array and
   4 @ puts the result in the same numbered element in PRODUCTS array.
   5 @ Uses R1-R4, R6-R8
   6 @ Douglas V. Hall September 2016
   7
   8              .text
   9              .global _start
  10              _start:
  11              .equ  NUM, 4
  12 0000 30109FE5 LDR R1,=MULTIPLICANDS @ Load pointer to MULTIPLICANDS
  13 0004 30209FE5 LDR R2, =MULTIPLIERS   @ Load pointer to MULTIPIERS
  14 0008 30309FE5 LDR R3, =PRODUCTS      @ Load pointer to PRODUCTS
  15 000c 0440A0E3 MOV R4,#NUM            @ Initialize loop counter
  16 0010 B060D1E1 NEXT:LDRH R6,[R1]@ Load a MULTIPLICAND Half word
  17 0014 B070D2E1 LDRH R7,[R2]           @ Load a MULTIPLIER half Word
  18 0018 960708E0 MUL R8, R6, R7         @ Multiply
  19 001c 008083E5 STR R8, [R3]           @ Store result in PRODUCTS array
  20 0020 021081E2 ADD R1,R1,#2           @ Increment MULTIPLICAN pointer
  21 0024 022082E2 ADD R2,R2,#2           @ Increment MULTIPLIER pointer  n
  22 0028 043083E2 ADD R3,R3,#4           @ Increment PRODUCTS pointer
  23 002c 014054E2 SUBS R4,#1          @ Decrement loop counter by 1
  24 0030 F6FFFF1A BNE NEXT               @ Go to NEXT if not all
  25 0034 00F020E3 NOP                       @ Instruction does nothing.
  26
  27
  28              .data
  29
  30 0000 11112222 MULTIPLICANDS: .HWORD 0x1111, 0x2222, 0x3333, 0x4444
  30      33334444
  31 0008 11112222 MULTIPLIERS:   .HWORD 0x1111, 0x2222, 0X3333, 0x4444
  31      33334444
  32 0010 00000000       PRODUCTS:   .WORD  0x0, 0x0, 0x0, 0x0
  32      00000000
  32      00000000
  32      00000000
  33              .END

DEFINED SYMBOLS
          ../mult1.s:10      .text:00000000 _start
          ../mult1.s:11      *ABS*:00000004 NUM
          ../mult1.s:30      .data:00000000 MULTIPLICANDS
          ../mult1.s:12      .text:00000000 $a
          ../mult1.s:31      .data:00000008 MULTIPLIERS
          ../mult1.s:32      .data:00000010 PRODUCTS
          ../mult1.s:16      .text:00000010 NEXT
          ../mult1.s:28      .text:00000038 $d
                    .debug_aranges:0000000c $d
NO UNDEFINED SYMBOLS
```

Figure 3-9 CCS List file for mult1 program in Figure 3-5.

USING A LINKER PROGRAM TO CREATE AN EXECUTABLE IMAGE FILE

After your program assembles with no errors, the next step is to run the Linker program on the Object code (.o) file created by the assembler. The linker produces an executable (.elf) file from one or more object files. This executable file is the one you will load with a debugger or operating systems to run and debug. As shown in Figure 3-8, the object modules can come from compiled C programs or from assembly language programs. Note that you need to run the linker on your program even if you only have one module, because the linker fills in values, such as those in the literal pool, that are needed for the program to execute. For a program that will be loaded into a ROM or executed at a specific address in memory, the linker is used to assign absolute addresses to the program sections.

The manual for the GNU linker, ld, can be found at www.gnu.org. This manual has a large number of pages because the GNU linker has a great many capabilities. We will explain the use of some of the more advanced ld capabilities as needed throughout the book but, to link the mult1.o example here, all that is needed is a command line entry of the form: "processor type-elf.ld mult1.o -o mult1.elf –M mult1.mp" This command tells the linker to link the object file mult1.o to produce an output executable file called mult1.elf and a link map file called mult1.mp.

When you are debugging a program, you will find the map produced by the linker very useful in helping you find where all the program sections are located. The map files are quite lengthy but, you usually need only a little information from the map file and this is easily found. As an example, Figure 3-10 shows the relevant sections of the map file produced by linking the mult1.o file with the command above.

You can see from the Memory Configuration section that the DDR3 DRAM starts at 0x80000000 and has a length of 0x40000000. A little further down, you can see that the linker put the .text section of the program at 0x80000070 and that the .text section requires 0x4 bytes. The address assigned to the .text section by the linker is the absolute address where the program will be loaded into memory by a debugger for execution. In the next chapter, we will show you how you tell the linker to use some other starting address and, in a later chapter, we will show you how to tell the linker to make the program *relocatable* so that an operating system can load the program at any available location in memory for execution.

Note in Figure 3-10 that the linker located the .data section at 0x800000b4, just above the .text section and that the .data section required 0x20 bytes. Also note in Figure 3-10 that the linker assigned an address of 0x800000d4 to the .bss section but that this section contains 0x0 bytes. The .bss section of a program is used for data that is not assigned initial values. Since the mult1 program has no uninitialized data, this section has zero bytes. In any case, once you have an executable file from the linker and know the starting address for the program, the next step is to execute and debug the program.

```
Memory Configuration

Name             Origin             Length             Attributes
SRAM             0x402f0400         0x0000fc00
L3OCMC0          0x40300000         0x00010000
M3SHUMEM         0x44d00000         0x00004000
M3SHDMEM         0x44d80000         0x00002000
DDR0             0x80000000         0x40000000
*default*        0x00000000         0xffffffff

Linker script and memory map
```



```
                0x00018000                STACKSIZE = 0x18000
                0x00000400                HEAPSIZE = 0x400
LOAD ./mult1.o
LOAD ./startup_ARMCA8.o

.rsthand        0x80000000        0x70
                0x80000000                . = ALIGN (0x10000)
 *(.isr_vector)
 .isr_vector    0x80000000        0x50 ./startup_ARMCA8.o
                0x80000000                __isr_vector
                0x8000004e                DEF_IRQHandler
 *startup_ARMCA8.o(.text)
 .text          0x80000050        0x20 ./startup_ARMCA8.o
                0x80000050                Entry
                0x80000070                . = ALIGN (0x4)

.text           0x80000070        0x44
 *(.text*)
 .text          0x80000070        0x44 ./mult1.o
                0x80000070                _start

.data           0x800000b4        0x20
                0x800000b4                . = ALIGN (0x4)
                0x800000b4                __data_start__ = .

.bss            0x800000d4         0x0
                0x800000d4                . = ALIGN (0x4)
                0x800000d4                __bss_start__ = .
 *(.bss*)
 .bss           0x800000d4         0x0 ./mult1.o
 .bss           0x800000d4         0x0 ./startup_ARMCA8.o
 *(COMMON)
                0x800000d4                __bss_end__ = .
```

Figure 3-10 CCS Map file for mult1 program

RUNNING AND DEBUGGING PROGRAMS

For running, testing, and debugging a low-level ARM program, such as the mult1 program in Figure 3-5 you can use a debugger such as the gdb debugger that is part of the GNU program development tool chain or the GNU debugger that is part of the CCS tool chain. As we will describe in detail a little later, a debugger is a program that allows you to load a program, run the program to a specified breakpoint, single-step though the program one instruction at a time, and examine variables, registers, or memory locations at any point in the program. However, before we discuss the features and use of a debugger, we will briefly discuss the three possible program execution environments you can use with a debugger. As shown at the bottom of the flowchart in Figure 3-8, the three debugging environments are as follows:

USING THE DEBUGGER WITH AN INSTRUCTION SET SIMULATOR.

If you do not have a development board for then particular processor you are using or the board does not have hardware support for debugging, you can use an *Instruction Set Emulator* to run and debug programs. An instruction set emulator is a program that simulates the operation

of one machine while running on another. For emulating a program, the emulator reads each machine instruction from the linker output file and, using a software model of the destination processor architecture on the host system, it generates the same result as would be generated by that instruction executing on the target system. To assist in timing calculations, some Instruction Set emulators can even give an estimate of the number of clock cycles that would be required by a given set of instructions if they were run directly on the target machine. A program will, of course, run much more slowly on an Instruction Set Simulator than it would on the actual hardware but the results are the same. Obviously, an Instruction Set Simulator is the only choice if, for example, a development board for the target processor is not available. An Instruction Set Simulator for many different processors is one of the options in the GDB debugger that is part of the GNU tool chain.

USING A DEBUGGER WITH AN IN CIRCUIT EMULATOR.

If you are developing a board for a particular processor and the board has hardware support for debugging but the board is not yet able to independently execute software, then you use an *In Circuit Emulator* or ICE to run and debug both the hardware and software on the board. An ICE unit is a combination of hardware and software that allows you to use the host system to directly execute instructions on the core processor on a board. Figure 3-11 shows a Blackhawk USB 100v2 JTAG Emulator that works with the B3 board. It connects a USB port on a host system to the processor JTAG port that we briefly mentioned in our discussion of the AM3359 architecture in the last chapter.

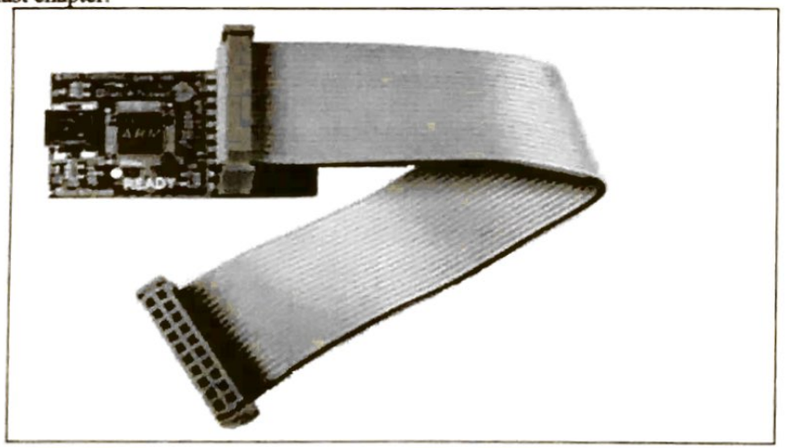


Figure 3-11 Blackhawk USB 100v2 JTAG Emulator for use with Beaglebone Black board.

In a later chapter, we will discuss the signals and use of the JTAG port in detail, but for now just think of it as a way to get control signals directly into the processor core. The software part of the ICE runs on the host system and functions as an interface between the debugger and the ICE hardware. Since an ICE controls the on-board processor directly, it does not require the execution of a debug server or other software on the development board. An ICE unit is very

useful when you are doing the first debug of the hardware for a new board. With an ICE unit you can, for example, execute an instruction that writes a value to a memory location and then execute an instruction that reads the memory location. This gives you a first indication of whether the memory is working correctly or not. You can also execute an instruction that writes a value to and output port and then use an oscilloscope or logic analyzer determine if the value you wrote made it to the output port pins. In addition to accessing the core processor, an ICE such as the Blackhawk allows you to access all of the peripheral devices on an ARM-based microprocessor board.

USING THE HOST SYSTEM DEBUGGER WITH AN ON-BOARD DEBUGGER.

If you have an external development board that is capable of running the on-board part of a debugger and it is connected to your cross-development system with an RS-232 or Ethernet connection, you can use this combination of debuggers to run and debug your programs. In this type of system, the debugger program on the host system works with the on-board debug server to download and execute programs. The terminology often used for this relationship in manuals is that the on-board gdb debugger acts as a debug server for the host debug client. This simply means that the on-board debugger executes commands as requested by the host debugger and returns results of instruction execution to the host debugger. With this combination of debuggers, you use all of the same debugger commands as you do when using the debugger with an Instruction Set Emulator or ICE unit. The difference is that with this setup, the instructions are loaded and run directly on the target processor board, instead of being simulated on the host or downloaded through and ICE unit. Your programs can then directly access all of the hardware on or connected to the development board. However, due to the interaction with the debugger, programs may not run at full speed. To overcome this problem, the next step in the development process is to load the program in a Flash memory on the board and have it run independently of the debugger, when the Reset button on the target system is pushed.

SUMMARY OF DEBUGGING APPROACHES

When you have no external processor-based board, you use an Instruction Set Emulator running in the host system to execute and debug programs. When you are developing a processor-based board you use an ICE to directly control the processor on the target board. This allows you to test and debug or "bring up" the hardware on the new board. When the new board is functioning well enough to run a debug server, you use this server to run and debug programs on the target board. For all three of these approaches, you use a debugger such as the GDB debugger running on the host development system. Now that you have an overview of the three techniques, we will introduce you to the use of a debugger and take a quick look at the type of information typically provided by a debugger.

DEBUGGER OPERATION AND OUTPUTS

Figure 3-12a shows the Graphical User Interface (GUI) screen that appears when the debugger is accessed through the TI Code Composer Studio tools. One window in Figure 3-12a

shows the source code for the mult1 program. This debugger is a *source-level debugger*, which means that you can use it to step through your source code one line at a time and observe the values of variables, registers, and memory locations after each step. The debugger also allows you to see a *disassembly* of the executable file. Essentially the dissasembler recreates your assembly language program from the executable file. As shown in Figure 3-12b the disassembly window shows addresses, the machine codes, and the assembly language instructions that the disassemble created from the machine code. This display is very useful in helping you figure out exactly what is happening in a program. We will use it here to explain how a literal pool is used to store 32-bit values that cannot be directly coded into an instruction.

Note at the _start address of 0x80000070 in the Disassembly window in Figure 3-11b that the instruction LDR R1,= MULTIPLICANDS instruction in the source program was translated by the assembler into the instruction LDR R1, 0x800000A8. This instruction tells the processor to load the contents of memory at location 0x800000A8 into R1. (Remember that the 32-bit address of the MULTIPLICANDS array cannot be specified directly in the instruction as an immediate number. To work around this problem, the assembler creates a literal pool to hold the address of MULTIPLICANDS and codes the LDR instruction to load the desired address from a specified location in the literal pool.)

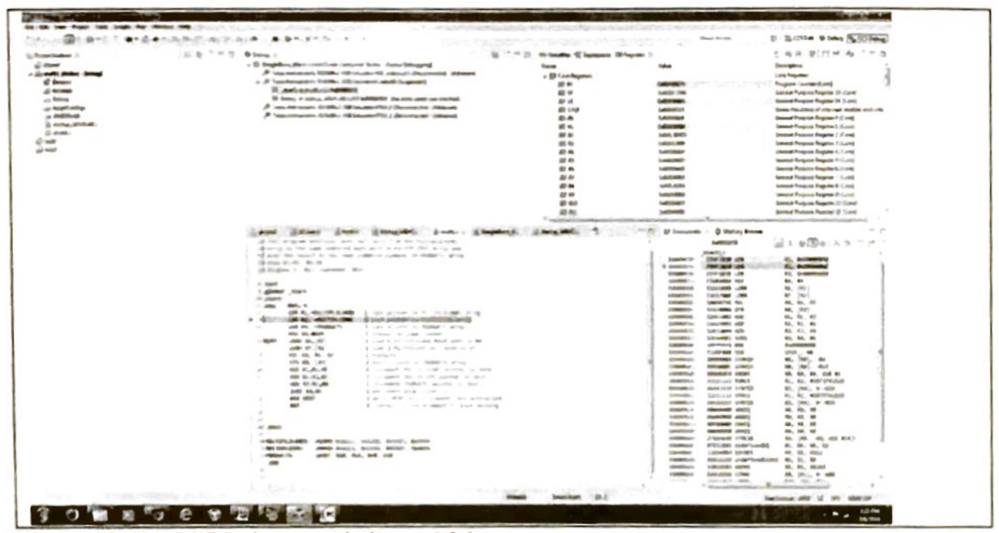


Figure 3-12a CCS Debugger windows - Main screen.

Now, in Figure 3-12b, if you look at the contents of address 0x800000A8, you can see that it contains the address 0x800000B4, the address of MULTIPLICANDS. If you then look at the contents of address 0x800000B4, you should see the first two half words of the MULTIPLICANDS array, 0x11112222. Note that the debugger happens to display these as they would be loaded into a register in a little endian system such as this.

Key point here is that addresses 0x800000A8, 0x800000AC, and 0x800000B0 serve as a literal pool to hold the 32-bit addresses needed for the LDR instructions. To further see the operation of the literal pool, use Figure 3-11b to work through the use of the pool for the other two LDR instructions in the program.

The actual coding of the LDR R1,= MULTIPLICANDS instruction uses Program Counter (PC) relative addressing, since the LDR instruction obviously cannot contain the 32-bit address of the desired location in the literal pool. To solve this problem, the assembler calculates the distance, in bytes, from the current PC value to the desired value in the literal pool and codes this value in the machine code for the LDR instruction. The 0x30 in the low byte of the codes for the LDR instructions is the offset added to the PC to access the literal pool. Note that when the LDR R1=MULTIPLICANDS instruction at 0x80000070 is in its EX stage and does the addition, the PC will have incremented by 2 word values to 0x80000078. Adding the offset of 0x30 to this PC value gives the desired literal pool address of 0x800000A8, where the MULTIPLICANDS address is stored.

Another example of Program Counter (PC) relative addressing is provided by the "BNE NEXT" instruction coding. As you may remember from the program listing in Figure 3-5 and a previous discussion, the BNE NEXT instruction will cause execution to go to the instruction at the NEXT label if the Zero flag (Z) is not set. If the Z is set by R4 reaching zero after the SUBS R4, R4, #1 instruction, execution falls through to the end of the program. In Appendix A, the coding template for the Branch instruction shows that the lower 24 bits in the machine code represents a signed offset from the PC to the branch target. If you look at the code for the BNE instruction at address 0x800000A0 in the Disassembly display in Figure 3-12b, you will see that the code for the instruction is 0x1afffff6. The 0xffffff6 coded into the BNE NEXT instruction represents a signed displacement of -0xA words or -0x28 bytes. The instruction at the NEXT label is at address 0x80000080 and the BNE NEXT instruction is at 0x800000A0. However, due to pipelining, the PC will have incremented to 0x800000A8 by the time the BNE instruction gets to its' EX stage. The displacement from 0x800000A8 back to 0x80000080 is then –0x28 bytes or – 0xA words as coded in the instruction. Note that the offset is coded as the number of 4 byte words, not bytes, since the branch destination is always some number of instructions that are each one word. As a final comment, you can see in the Register display in Figure 3-12a, that since we did the printout just after the first multiplication, register R8 contains 0x1234321, the result of the first multiplication.

```
        Entry():
80000050:  E59F0010 LDR      R0, 0x80000068
80000054:  EE0C0F10 MCR       P15, #0, R0, C12, C0, #0
80000058:  E59FA00C LDR      R10, 0x8000006C
8000005c:  E1A0E00F MOV      R14, PC
80000060:  E12FFF1A BX       R10
80000064:  E24FF008 SUB      PC, PC, #8
80000068:  80000000 ANDHI     R0, R0, R0
8000006c:  80000070 ANDHI     R0, R0, R0, ROR R0
      _start():
80000070:  E59F1030 LDR      R1, 0x800000A8
80000074:  E59F2030 LDR      R2, 0x800000AC
80000078:  E59F3030 LDR      R3, 0x800000B0
8000007c:  E3A04004 MOV       R4, #4
80000080:  E1D160B0 LDRH      R6, [R1]
80000084:  E1D270B0 LDRH      R7, [R2]
80000088:  E0080796 MUL       R8, R6, R7
8000008c:  E5838000 STR      R8, [R3]
80000090:  E2811002 ADD       R1, R1, #2
80000094:  E2822002 ADD       R2, R2, #2
```

```
80000098:  E2833004 ADD      R3, R3, #4
8000009c:  E2544001 SUBS     R4, R4, #1
800000a0:  1AFFFFF6 BNE      0x80000080
800000a4:  E320F000 MSR      CPSR_, #0
800000a8:  800000B4 STRHIH   R0, [R0], -R4
800000ac:  800000BC STRHIH   R0, [R0], -R12
800000b0:  800000C4 ANDHI    R0, R0, R4, ASR #1
800000b4:  22221111 EORCS    R1, R2, #1073741828
800000b8:  44443333 STRMIB   R3, [R4], #--819
800000bc:  22221111 EORCS    R1, R2, #1073741828
800000c0:  44443333 STRMIB   R3, [R4], #--819
800000c4:  00000000 ANDEQ    R0, R0, R0
800000c8:  00000000 ANDEQ    R0, R0, R0
800000cc:  00000000 ANDEQ    R0, R0, R0
800000d0:  00000000 ANDEQ    R0, R0, R0
```

(b)

Figure 3-12b CCS Debugger windows - Disassembly.

USING A DEBUGGER TO HELP DEBUG A PROGRAM

The preceding discussion clearly shows the need for you to understand the processor architecture, machine language coding templates, and assembly language in order for you to be able to understand and debug low-level programs. Given these skills, you can use a debugger to determine if a program is doing what you want it to do, instruction by instruction or, if you don't want to step through every instruction one at a time, you can set a breakpoint at a particular instruction and tell the debugger to execute instructions until it reaches the breakpoint instruction and then stop. In any case, however, debugging can be a very time consuming and frustrating process unless you do it in a very systematic way. As part of this, it is important to remember that the processor always does exactly what you tell it to do, assuming that the processor is working correctly. Therefore, if the result of your program is not correct, then it is likely that you did not give the processor the correct instruction or the correct sequence of instructions. Here are some hints on how you use a debugger to quickly find the cause of a problem instead of just demonstrating the problem to yourself over and over.

If you did not already do so as part of your program development process, start by carefully working some test data values through your algorithm on paper and predicting the result that your algorithm should produce for each. You then mentally work the same test values through your source program and predict the result that should be produced for each. Obviously, the result of these should be the same, if you translated the algorithm correctly to instructions. If they are not, you need to correct the translation and cycle through the tool chain again. If they are the same on paper, then you step through the program one instruction at a time to find where the result is different from the predicted result. The key point here is that it does you little good to see the result your program produces unless have figured out exactly what the correct result should be. Once you localize the problem to a single instruction or a few instructions, you can usually find the problem quickly. One of the errors often made by the author is to change a destination register from, for example, R5 to R3 in one section of a program, forget that he did this change when writing a later section of the program the next day, and then wonder why the desired value is not in R5 as he remembered. The point here is that, if he had followed his own

advice and written down the predicted contents of each affected register at each step of the program, he would catch the error before running the program. However, he is human and error prone so he sometimes gets in a hurry and takes what seems like a shortcut. The debugger makes it relatively easy to find this type of error, BUT experience strongly shows that attempted shortcuts in program design and lack of mental testing ALWAYS result in greatly increased debugging time and a higher level of stress.

At this point you should have enough understanding of the command line tool chain that you can work though the tutorial available on the text web site and become comfortable with the CCS toolset. Note that the compete instructions for the assembly language programming with CCS IDE are in the Assembly Language Tools V5.1 Users Guide at http://www.ti.com/lit/ug/spnu118o/spnu118o.pdf. However, the next step here is to show you a program development process that is much faster and more efficient that just writing down a sequence of assembly language or C instructions "off the top your our head."

A Systematic Program Development Process

THE "FAST IS SLOW" RULE

In the preceding sections of the chapter, we have described the ARM-based processors programmer's model, shown three alternative programming language levels, and discussed the computer-based tools you use to develop programs. This would seem to be all you need to develop programs, but the big missing piece is a systematic program design process. A common mistake of beginning programmers and even some experienced programmers is to sit down and immediately start writing assembly language instructions or C program statements when given a programming assignment. The problem in doing it this way is that it is very easy for you to get lost in the details of the instructions or in pointer arithmetic and totally lose sight of the big picture of what you are trying to do. In our observation, the people who attempt to program in this way usually suffer very long and painful debugging session(s) instead of getting their programs up and running quickly. The most efficient way to successfully develop a program is to follow the "**Fast is Slow**" rule that we use. This rule is stated as "The **Fastest** way to develop a working program is to **Slowly** and carefully work your way through a step-by-step design process."

We briefly described some of the steps in this process in the section on program development tools and debugging, but here we will give a more complete list and then discuss some of the steps in detail. A reasonable sequence of program development steps is as follows:

1. Analyze the problem you are trying to solve or the programming task until you understand it completely. The test for this is that you can clearly explain it to someone else. In practice, it is a good idea to do this because the other person's questions may help clarify the problem in your own mind.
2. Define the data structure(s) needed by the program, list and describe the program inputs, and list and describe the program outputs. This is equivalent to drawing a block diagram with inputs and outputs for a hardware design. We usually cycle through steps 1 and 2 several times before the whole picture of the project becomes clear.
3. Research within your company and on the Web to see if someone else has developed an algorithm or even a non-copyrighted program solution for the problem. This does not mean that you should just use your classmate's solution for a programming assignment in

a college class, but on the job it may save you from "re-inventing the wheel" and save the company a lot of development time and money.

4. If the programming project is a large one, attempt to divide the project into functional modules. It is much easier to write, test, and debug one small module at a time than to work with a huge program all at once. Also, modules can be assigned to different programmers. Furthermore, tested, well-documented modules can often be reused in other programs.
5. Develop a structured algorithm for one of the modules of the program and test your algorithm by mentally working a range of data values through the algorithm. A little later in the chapter, we show you how to develop structured algorithms.
6. Translate the algorithm for the module to an appropriate programming language, and mentally work a variety of data values through your program.
7. Run the program through the development tool chain to test and debug it as described earlier.
8. As each module becomes functional, link it with other functioning modules, then test and debug the combined modules.
9. Repeat steps 5 through 8 until all modules work correctly.
10. Thoroughly test the complete program with a variety of data values. Make sure to try *boundary values* such as less than a decision value, equal to a decision value, and greater than a decision value. What you really try to do at this point is to try to "break" the program. It is much better for your personal ego and your company if you find and fix a problem before the product is shipped to customers, rather than afterward.

STRUCTURED PROGRAMMING HISTORY AND OVERVIEW

In the early days of computers, a single brilliant person might write even a relatively large program single-handedly. The main concerns in this case were, "Does the program work?" and "What do we do if this person leaves the company?" As the number of computers and the complexity of the programs being written increased, large programming jobs were usually turned over to a team of programmers. In this case the compatibility of parts written by different programmers became an important concern and it became obvious to many professional programmers that, in order for team programming to work, a systematic overall approach and standardized program representations were absolutely necessary.

One systematic overall approach is called *top-down design*. In this approach, a large programming problem is first divided into major *modules*. The top level of the outline shows the relationship, function, inputs and outputs of each module. This top level then presents a one-page overview of the entire program. Each of the major modules is broken down into still smaller modules on following pages. The division is continued until the steps in each module are clearly understandable. Each programmer can then be assigned a module or set of modules to develop for the program. Another advantage of this approach is that people who later want to learn about the program can start with the overview and work their way down to the level of detail they need. This approach is the same as drawing the complete plans for a house before starting to build it.

The opposite of top-down design is *bottom-up design*. In this approach, each programmer starts writing low-level modules and hopes that all the pieces will eventually fit together. When completed, the result should be similar to that produced by the top-down design. Most modern programming teams use a combination of the two techniques. They do the top-down design first,

then build, test, and link modules starting from the smallest and working upward as we described above.

Along with developing the overall programming project in a systematic way, it is important to develop each module in a systematic way. The first step is to represent the actions desired in a module in a high-level algorithmic form rather than in a particular programming language. The following sections describe a couple of ways to do this.

REPRESENTING PROGRAM OPERATIONS

As we said before, the formula or sequence of operations used to solve a programming problem is often called the *algorithm* of the program. The following sections show you two common ways of representing the algorithm for a program or program segment.

FLOWCHARTS

Flowcharts use graphic shapes to represent different types of program operations. To some people, flowcharts are considered somewhat obsolete because they take a lot of space and are therefore not useful for describing the operations in large programs. However, we have found them very useful in explaining the operation of small program sections and sequences of actions as we did in Figure 3-8. Therefore, we will show you how to use them before we show you a more compact way of representing program actions. Note that the flowchart symbols are available in Microsoft Word under the AutoShapes menu, so you can easily access and use them on a computer if you want to use them for small program sections.

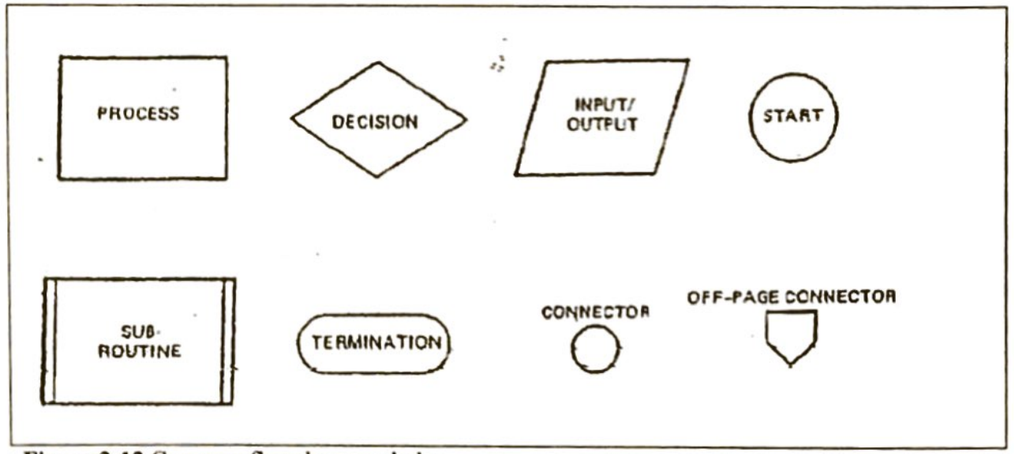


Figure 3-13 Common flowchart symbols

Figure 3-13 shows some of the common flowchart symbols. Figure 3-14 shows a flowchart for a program to read in 24 data samples from a temperature sensor at 1-hour intervals, add 7 to each, and store each result in a memory location. A racetrack or circular-shaped symbol labeled START is used to indicate the *beginning* of the program. A parallelogram is used to represent an *input* or an *output* operation. In the example, we use it to indicate reading data from the

temperature sensor. A rectangular box symbol is used to represent *simple operations* other than input and output operations. The box containing "add 7" in Figure 3-14 is an example of this.

A rectangular box with double lines at each end is often used to represent a *subroutine* or *procedure* that will be written separately from the main program. When a set of operations must be done several times during a program, it is usually more efficient to write the series of operations once as a separate subprogram or function and then just "call" this subprogram each time it is needed. For example, suppose that there are several places in a program where you need to compute the square root of a number. Instead of writing the series of instructions for computing a square root in the program each time you need the operation, you can write the instruction sequence once as a separate procedure. As we mentioned earlier in the chapter, the Branch and Link instruction, BL, can be used to branch to this procedure each time you need to compute a square root and a MOV PC, LR instruction at the end of the procedure can be used to return execution to the main program. In the flowchart in Figure 3-14, we use the double-ended box to indicate that the "Wait 1 Hour" operation will be programmed as a procedure or subroutine.

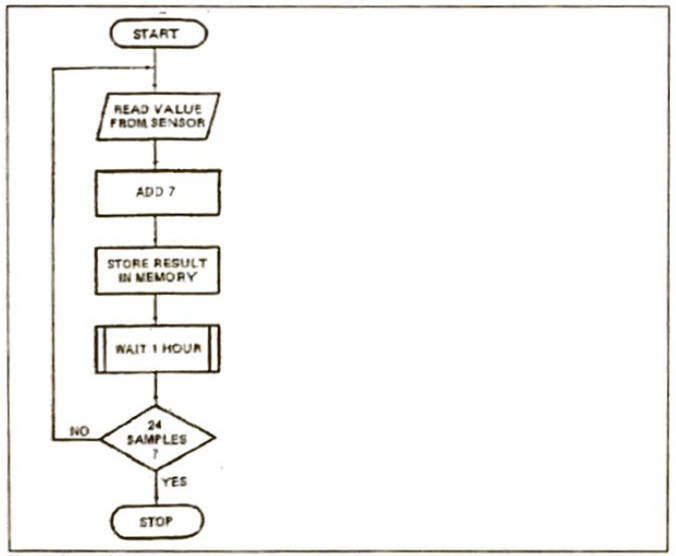


Figure 3-14 Flowchart for program to read in 24 data samples from a port, correct each value, and store each in a memory location.

A diamond-shaped box is used in flowcharts to represent a *decision* point or crossroads. Usually it indicates that some condition is to be checked at this point in the program. If the condition is found to be *true*, one set of actions is to be done; if the condition to be found to be *false*, another set of actions is to be done. In the example flowchart in Figure 3-14 the condition to be checked is whether 24 samples have been read in and processed. If 24 samples have not been read in and processed, the arrow labeled NO in the flowchart indicates that we want the computer to jump back and execute the read, add, store, and wait steps again. If 24 samples have

been read in, the arrow labeled YES in the flowchart indicates that all of the desired operations have been done. The racetrack-shaped symbol at the bottom of the flowchart indicates the *end* of the program.

The two additional flowchart symbols in Figure 3-13 are *connectors*. If a flowchart column gets to the bottom of the paper, but not all the program has been represented, you can put a small circle with a letter in it at the bottom of the column. You then start the next column at the top of the same paper with a small circle containing the same letter. If you need to continue a flowchart on another page, you can end the flowchart on the first page with the five-sided off-page connector symbol containing a letter or number. You then start the flowchart on the next page with an off-page connector symbol containing the same letter or number.

PSEUDOCODE

The development of structured programming methods was helped by the discovery that any desired program actions can be represented with three basic operation types. The first operation type is *sequence*, which means simply doing a series of actions. The second basic operation type is *decision*, or *selection*, which means choosing between two or more alternative actions. The third basic operation type is *repetition* or *iteration*, which means repeating a series of actions until some condition is or is not present.

On the basis of this observation, the suggestion was made that programmers use a set of three to seven standard *structures* to represent all the operations in their programs. Only three structures, SEQUENCE, IF-THEN-ELSE, and WHILE-DO, are required to represent any desired program action, but three or four more structures derived from these often make program operations clearer. If you have previously written programs in a high-level language, then these structures are probably already familiar to you. If not, the flowcharts for each in Figure 3-15 should help you quickly see the operation of each. In actual program development, however, we usually use more English-like statements called *pseudocode* rather than space-consuming flowchart symbols. Figure 3-15 also shows the pseudocode format and an everyday example for each of the standard structures.

A very important point here is that the program algorithm is ALWAYS written so that each structure has only *one entry point* and *one exit point*. The output of one structure is connected to the input of the next structure. Program execution then proceeds through a series of these structures. The one entry point and one exit point feature makes debugging the final program module much easier. You can set a breakpoint on the instruction at the start of a structure and check if the data is correct at that point. You can then set a breakpoint on the instruction at the end of the code for the structure, run to that breakpoint, and check it the result is correct at that point. If the result is correct, you can repeat the process for the code of the next structure in the program. If the result is not correct, you can debug the module before going on to the next. The point is that with this structured approach you are debugging just small program sections at a time instead of attempting to debug a single, large block of code.

Any structure can be used within another. An IF-THEN-ELSE structure, for example, can contain a sequence of statements. Any place that the term *statement(s)* appears in Figure 3-15 one of the other structures could be substituted for that statement. The term *statements(s)* can also represent a subprogram or procedure that is called to do a series of actions. Now, let's look more closely at these structures in Figure 3-17a.

STANDARD PROGRAMMING STRUCTURES

The structure shown in Figure 3-15a is an example of a simple sequence. In this structure, the actions are simply written down in the desired order. An example is:

Read temperature from sensor.
Add correction factor of +7.
Store corrected value in memory.

Figure 3-15b shows an IF-THEN-ELSE example of the decision operation. This structure is used to direct operation to one of two different actions based on some condition. An example is:

IF temperature less than 70 degrees THEN
    Turn on heater.
ELSE
    Turn off heater.

The example says that if the temperature is below the thermostat setting, we want to turn the heater on. If the temperature is equal to or above the thermostat setting, we want to turn the heater off.

The IF-THEN structure shown in Figure 3-15c is the same as the IF-THEN-ELSE except that one of the paths contains no action. An example of this is:

IF hungry THEN
    Get food

The assumption for this example is that if you are not hungry, you will just continue on with your next task.

To represent a situation in which you want to select one of several actions based on some condition, you can use a nested IF-THEN-ELSE structure such as that shown in Figure 3-15d. This everyday example describes the thinking a soup cook might go through. Note that in this example the last IF-THEN has no ELSE after it because all the possible days have been checked. You can, if you want, add the final ELSE to the IF-THEN-ELSE chain to send an error message if the data does not match any of the choices.

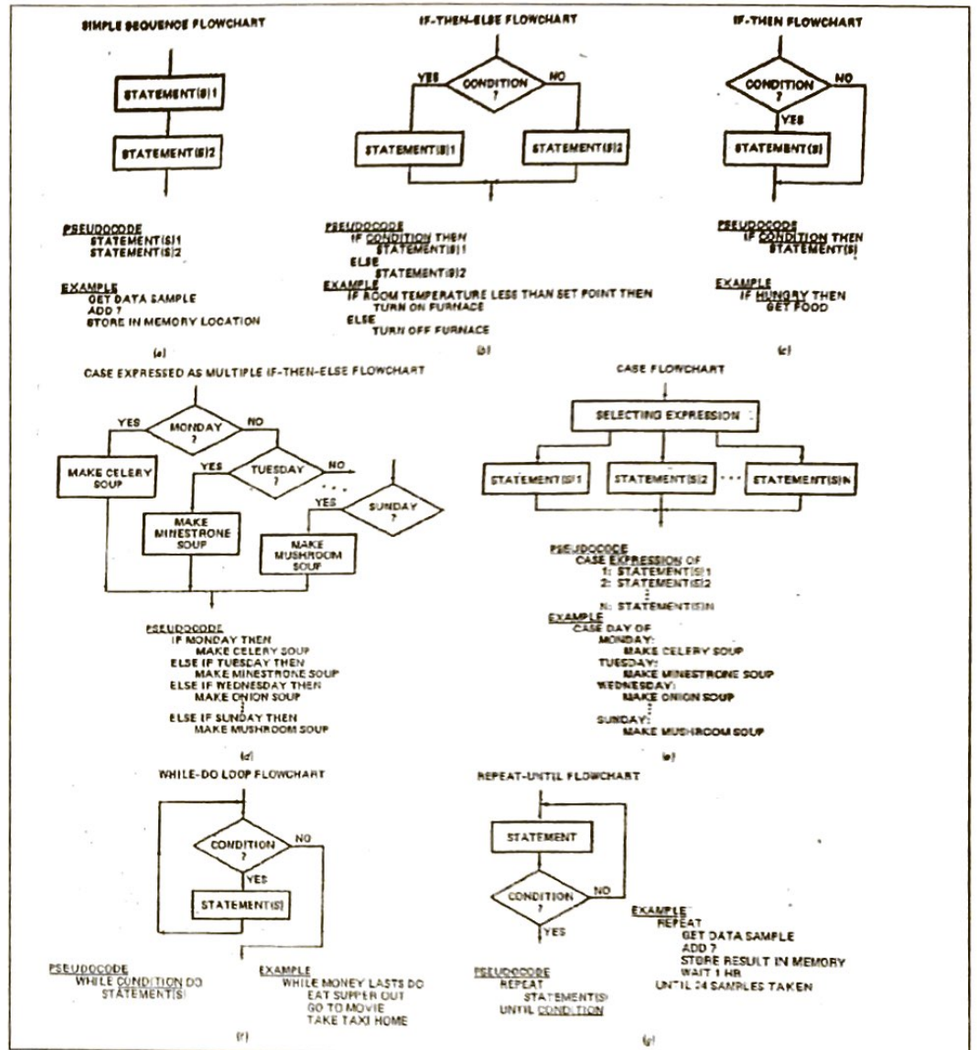


Figure 3-15 Standard program structures. (a) Sequence. (b) IF-THEN-ELSE. (c) IF-THEN. (d) CASE statement expressed as nested IF-THEN-ELSE. (e) CASE. (f) WHILE-DO. (g) REPEAT-UNTIL.

The CASE structure shown in Figure 3-15e is really just a compact way to represent a complex IF-THEN-ELSE structure. The choice of action is determined by testing some variable.

The cook or the computer checks the value of the variable called "day" and selects the appropriate actions for that day. Each of the indicated actions, such as "Make celery soup," is itself a sequence of actions that could be represented by the structures we have described. Note that the CASE structure does not contain the final ELSE clause, unless you specifically add a default case to do this. A final ELSE clause can be used to detect the error condition where the variable does not have one of the legal values.

The CASE form is more compact for documentation purposes, and some high-level languages allow you to implement it directly. However, the nested IF-THEN-ELSE structure gives you a much better idea of how you write an assembly language program section to choose between several alternative actions.

The WHILE-DO structure in Figure 3-15f is one form of repetition. It indicates that you want to do some action or sequence of actions as long as some condition is present. This structure represents one type of *program loop*. The example in Figure 3-15f is:

```
WHILE money lasts DO
    Eat supper out.
    Go to movie.
    Take taxi home.
```

This example shows a sequence of actions you might do each evening until you ran out of money. Note that in this structure, *the condition is checked before the action is done* the first time. You certainly want to check how much money you have before eating out.

Another useful repetition structure is the REPEAT-UNTIL structure shown in Figure 3-15g. You use this structure to indicate that your want the program to repeat some action or series of actions until some condition is present. A good example of the use of this structure is the programming problem we used in the discussion of flowcharts. The example is

```
REPEAT
    Get data sample from sensor.
    Add correction of +7.
    Store result in a memory location.
    Wait 1 hour.
UNTIL 24 samples taken.
```

Note that in a REPEAT-UNTIL structure, *the action or actions are done once before the condition is checked*. If you want the condition to be checked before any action is done, then you can write the algorithm with a WHILE-DO structure as follows:

```
WHILE NOT 24 samples DO
    Read data sample from temperature sensor.
    Adds correction factor of +7.
    Store result in memory location.
    Wait 1 hour.
```

Remember, a REPEAT-UNTIL structure indicates that the condition is first checked after the statement(s) is performed, so the action or series of actions will always be done at least once. If

you don't want this to happen, then use the WHILE-DO structure, which indicates that the condition is checked *before* any action is taken. As we will show later, the structure you use makes a difference in the assembly language program you would write to implement the program.

The WHILE-DO and REPEAT-UNTIL structures contain a simple IF-THEN-ELSE decision operation. However, since this decision is an implied part of these two structures, we don't indicate the decision separately in them.

Another form of the repetition operation that you might see in high-level language programs is the FOR-DO loop shown in the C program in Figure 3-8. This structure has the form:

```
FOR count = 1 TO n, DO
    statement
    statement
```

A FOR-DO loop simply repeats the sequence of actions n times, so for assembly language algorithms we usually implement this type of operation with a REPEAT-UNTIL structure.

Now that you have had an introduction to assembly language programming, to the program development tools, and to a structured approach to developing programs, it is time for you to write some programs. In the next chapter we show you how to write algorithms for a variety of programming problems using standard structures and how to translate each into efficient assembly language. We also show you how to write programs that contain both C and assembly language modules.

References

(1) https://silver.arm.com/download/ARM_and_AMBA_Architecture/AR570-DA-70000-r0p0-00rel2/DDI0406C_C_arm_architecture_reference_manual.pdf .

(2) CCS Assembly Language Tools V5.1 Users Guide at http://www.ti.com/lit/ug/spnu118o/spnu118o.pdf

Checklist of Important Terms and Concepts

Instruction Set Architecture
Programmer visible registers
ARM User mode
ARM Privileged modes
Current Program Status Register (CPSR)
N, Z, C, and V flags in the CPSR
Word aligned address
Little Endian Byte ordering
Big Endian Byte ordering
Memory-mapped I/O
Direct I/O
GPIO pins
Peripheral Control Registers
Machine language programming
Conditional execution

Assembly language programming
Mnemonic(s)
Load-and-Store machine
Effective Address (EA)
Unconditional and Conditional Branches
PC relative addressing
Source file
Program header
Assembler directives
High-level language programming
Cross-development tools
Program development tool chain
Assembler/compiler, linker/locator
Object file, list file, map file
Executable image file
Debugger, Source-level debugger
Instruction Set Emulator
In-Circuit-Emulator (ICE)
Integrated Development Environment (IDE)
"Fast is Slow" rule
Boundary values for testing programs
Top-Down design and Bottom-Up design
Algorithm
Flow chart symbols
Pseudocode
Standard program structures

Review Questions and Problems

1. Briefly describe the register set visible to the programmer in the User mode of the ARM-based processors.

2. List and briefly describe the six privileged modes in which an ARM-based processor can operate.

3. Assuming that an ARM-based processor register contains 0x12345678, show the order in which the bytes of this word would be written in memory starting at address 0x00000000 in a machine that uses Little-Endian byte ordering.

4. Describe how I/O ports are accessed in system that uses Memory-Mapped I/O and how they are accessed in a system that uses Direct I/O. List the advantages and disadvantages of each type of port addressing.

5. You have been assigned the task of developing the program for a system that accumulates and analyzes data from 20 remote seismic sensors that communicate with a central system

over wireless links. Give some reasons why you would most likely write the majority of this program in a high-level language.

6. Briefly describe some programming tasks where assembly language might be a good programming language choice.

7. Given the User mode register and memory contents shown below, predict the register contents and, where appropriate, the flag states produced by each of the following ARM-based processor instructions. Assume the values shown for register and memory contents are all in hexadecimal.

R0 – 00004037	R8 – 00000008
R1 – FFFF0000	R9 –
R2 – AAAA5555	R10 –
R3 – 5555AAAA	R11 –
R4 – 00000000	R12 –
R5 – 00008050	R13 –
R6 – 00008058	R14 –
R7 –	PC(R15) – 00008020

	ADDRESS
SCORES	00008050 04 77 59 27
	00008054 99 22 14 53
	00008058 27 98 42 70

   a. MOV R1, R2
   b. ORR R2, R0, R1
   c. ADDS R5, R2, R3
   d. ANDS R2, R2, R3
   e. CMP R3, R2
   f. MUL R9, RO, R8
   g. LDR R4, [R5]
   h. LDR R9, [R5], #4
   i. LDR R9, [R5, #4]!
   j. STR R3, [R6]
   k. BEQ NEXT ('NEXT' is a label)
   l. B STOP ('STOP' is a label)

8. With the help of the examples in this chapter and Appendix B, write the ARM-based processor assembly language instruction(s) that will perform each of the indicated operations shown below.
   a. Increment R0 by 1.
   b. Copy R3 to R9.
   c. Put the immediate number 0x64 in R10.
   d. Store the contents of R3 at the address contained in R6.
   e. Mask the lower 8 bits of R3.
   f. Set bit 3 of R4 to a 1 but leave all other bits unchanged.

g. Determine if R2 > R1.
h. Add 5 to R8 if carry flag set.
i. Load starting address of array SCORES into R4.
j. Left-shift contents of R8 two bit positions to the left, add result to R5, and put sum in R7.

9. Write a sequence of two ARM-based processor instructions that will determine if R2 is equal to R3 and add R0 to R1 if they are equal.

10. The ARM auto-increment addressing mode can be used to automatically increment a pointer as part of a load or store operation. Using the auto-increment mode, rewrite the two LDR instructions and the STR instruction in the loop section of the assembly language program in Figure 3-5 to eliminate the three separate pointer increment instructions of the form ADD Rn, Rn, #2.

11. Using the coding templates in Appendix A, construct the machine codes for the following ARM instructions.

a. ADD R3, R3, R2
b. ORRS R4, R3, R2
c. MUL R8, R6, R7
d. LDRH R6, [R1]

12. List and briefly describe the major steps in the program development tool chain for an assembly language program that will be run on an Instruction Set Simulator.

13. The Debugger display in Figure 3-12b shows that the space for the PRODUCT array was located in memory starting at 0x800000C4. The product of the first two half words, 0x00001111 times 0x00001111, is 0x01234321. Assuming Little-Endian storage, show the order in which the four bytes of the result word will be written in memory addresses 0x800000C4 to 0x800000C7.

14. Briefly describe the advantages of using an Integrated Development Environment to develop multiple-module programs for a microprocessor-based system.

15. Briefly describe at least two advantages of the top-down design approach to solving a programming problem.

16. Explain why is it very important to develop a detailed algorithm for a program before you write any actual program instructions.

17. When you properly develop the algorithm for a program using the standard structures, each block of the resulting code has only one entry point and one exit point. Briefly explain how this one-entry, one-exit structure helps makes debugging much easier.

18. An algorithm is essentially a recipe. Use a flowchart or pseudocode to clearly represent the algorithm for the following recipe written in traditional recipe form. The point is to make the algorithm so clear that someone doesn't have to read it 23 times in order to follow the directions accurately. When you finish, instead of implementing the algorithm in assembly language, implement it in your microwave oven and use the result to help you get through the book.

Peanut Brittle:

1 cup sugar	1 teaspoon butter	½ cup white corn syrup
1 teaspoon vanilla	1 cup unsalted peanuts	1 teaspoon baking soda

i. Put sugar and syrup in 1½ quart casserole dish and stir until thoroughly mixed.
ii. Microwave on HIGH setting for 3 ½ - 4 minutes. (Time depends on oven power.)
iii. Add peanuts and stir until thoroughly mixed.
iv. Microwave on HIGH for 3 ½ - 4 minutes.
v. Add butter and vanilla, stir until thoroughly mixed, and microwave on HIGH for two more minutes
vi. Add baking soda and gently stir until light and foamy. Pour mixture onto a non-stick cookie sheet, let cool for one hour and, when cool, break into pieces. (Makes 1 pound.)

19. Use the program comments to help you write the pseudocode representation for the assembly language program example in Figure 3-5.

20. Use a flowchart or pseudocode to show the algorithm for a simple program that reads a temperature value from a port, lights a green light if the temperature value is less than 0x40, and lights a red light if the temperature value is greater than or equal to 0x40. (Don't be concerned with the details of how you actually turn on a light. We will show you how to do this in the next chapter.)

21. Use a flowchart or pseudocode to show the algorithm for a simple program that reads a temperature value from a port, lights a green light if the value is less than 0x40, lights a yellow light if the value is 0x40-0x49, and lights a red light if the value is equal to or greater than 0x50.

CHAPTER 4 – INTRODUCTION TO ARM ASSEMBLY LANGUAGE PROGRAMMING TECHNIQUES

In Chapter 2 we introduced you to the architecture of ARM-based processors. In Chapter 3 we described the different levels of programming languages you can use to write programs for ARM-based processors, discussed program development tool chains, and showed you how to use standard programming structures to represent algorithms for your programs. In this chapter we pull all of these concepts together and show you how to translate standard program structures into assembly language programs that work most efficiently with the ARM-based processor architectures. We feel that it is important that you learn basic program optimization techniques right from the start as you are learning ARM assembly language, so that these techniques become automatic to you. This chapter provides a solid base of how you systematically write basic assembly language programs and also how you write programs that contain both assembly language modules and C language modules. Later chapters will give you much more experience with programming as we lead you through the development of programs for a wide variety of hardware interfacing and system tasks.

Objectives

After working through this chapter, you should be able to:

1. Translate standard program structures into efficient blocks of ARM assembly language code.
2. Describe and show the use of the ARM Shift and Rotate instructions.
3. Initialize and access GPIO pins on a BeagleBone Black board.
4. Implement Repeat-until and While-Do program structures in ARM assembly language.
5. List and explain the general rules for writing assembly language programs that execute efficiently on an ARM-based processor.
6. Use the ARM conditional instruction execution capability in a program.
7. Write User-mode ARM assembly language programs that: set up and use a stack, call and return from a procedure, maintain calling program state, and pass parameters in registers or the stack.
8. Write an ARM assembly language procedure that can be called from a C program.
9. Write a C program that contains a short section of in-line assembly language.

Translating Standard Program Structures to Assembly Language

In the last chapter we stressed that, if you want to get a program working quickly, you never try to write it directly in assembly language or directly in any programming language for that matter. Instead you first develop the algorithm, represent the algorithm using standard program structures, and then translate the algorithm into assembly language or into a high-level language. To begin this chapter, we discuss some additional techniques that will help you quickly develop algorithms for low-level operations such as those that you would likely want to implement in assembly language. Then we show you how you translate the standard program structures discussed in the last chapter into assembly language.

DEVELOPING AND TRANSLATING SEQUENCES OF OPERATIONS TO ASSEMBLY LANGUAGE

Sequences of low-level operations that are likely to be programmed in assembly language often seem deceptively simple, so it is very easy to generate an incorrect program. In the following sections we use two very simple examples to show some techniques we have found helpful in getting the right answer the first time. Note that the program development process is the important point here, not the simple, specific examples.

CONVERTING TWO ASCII CODES TO PACKED BCD

Data is often transmitted to and from a robot over an RS-232C serial data line, using a seven- or eight-bit ASCII code to represent numbers, letters, and other characters. Table 1-2 shows the complete ASCII codes and Figure 4-1 shows the 8-bit ASCII codes for the decimal numbers 0-9.

ASCII Code	BCD Code	Decimal #
00110000	0000	0
00110001	0001	1
00110010	0010	2
00110011	0011	3
00110100	0100	4
00110101	0101	5
00110110	0110	6
00110111	0111	7
00111000	1000	8
00111001	1001	9

Figure 4-1. ASCII and BCD codes for decimal numbers 0-9.

From this you can see, for example, that the ASCII code for 5 is 00110101 and the ASCII code for 9 is 00111001. For many applications, such as reading the number of optical encoder holes being sent from a robot motor encoder, we want to convert the ASCII codes for numbers to their BCD equivalents and pack two BCD digits in a single byte. For example, if the motor encoder sent an ASCII 5 and then an ASCII 9, we would want to convert the ASCII codes from the encoder to their BCD equivalents and pack the two BCD codes into a byte to give BCD 0101 1001 or 59.

The process we use to determine the sequence of operations for a problem such as this is shown in Figure 4-2. A key point is that you should develop the sequence of operations and results at each stage on paper, rather than trying to just remember them all in your head. For the example here, the first step is to write an example of a data input value and below it the desired output value for that input. To get from the ASCII code for 9, or 0011 1001, to the BCD code for 9, or 1001, you can see that all you need to do is mask the upper four bits of the ASCII code. Likewise, to get from the ASCII code for 5 to the BCD code for 5, you also need to mask the upper four bits. As described in Chapter 2, you mask a bit by ANDing it with a 0.

ASCII 9	0011 1001	
		Mask with 0000 1111
Unpacked BCD 9	0000 1001	
ASCII 5	0011 0101	
		Mask with 0000 1111
Unpacked BCD 5	0000 0101	
		Shift left 4 bit positions
Unpacked BCD 5 Moved to upper nibble	0101 0000	
		Add shifted BCD 5 and BCD 9
Packed BCD	0101 1001	

Figure 4-2. Actions to perform in each step for converting two ASCII codes to packed BCD.

From the next line in Figure 4-2, you should see that the next step is to shift the BCD code 0101 (BCD code for 5) to the upper four bit positions in the byte. As you will soon see, the ARM-based processors have an instruction that easily does this shift. The final step is to combine or "pack" the BCD code 00001001 with the BCD code 01010000 to give 01011001, the desired BCD result for 59. By having the numbers written over each other as in Figure 4-2, you can see that you can combine them by simply ADDing or ORing 01010000 with 00001001. The point here is that, for most of us, it really helps to see on paper how the bits need to be manipulated for each operation.

From the comments along the right side of Figure 4-2, you can see that the required actions are masking, a shift left by 4 bit positions, and an ADD or OR. To write the assembly language program for this algorithm, you first put the two ASCII operands in appropriate registers because you know that in an ARM-based processor, all operations are done on registers. You then look up the instruction needed to perform each of the needed operations in the sequence and work out the syntax for each instruction.

As discussed in Chapter 1, you mask a bit in a word by ANDing it with a 0. To mask the upper four bits of a byte, you AND the byte with 0x0F. Remember that bits ANDed with a 1 remain unchanged.

Finding the instruction to do the required shift-left operation is a little more complicated because the ARM-based processors provide five types of shift/rotate operations. Also, the shift/rotate operations are always specified as part of an ARM data processing instruction, rather than as separate instructions. As a first example of this, the instruction ADD R1, R2, R3, LSL#2 will shift the contents of R3 two bit positions to the left, add the result to R2, and put the result in R1. (Zeros will be shifted into the vacated bit 0 and bit 1 positions in R3.) As another example, the instruction MOV R2, R2, LSR #2 will shift the R2 register contents 2 bit positions to the right and put zeros in the bit 31 and bit 30 positions left vacant by the shift.

Figure 4-3 shows the syntax for each of the ARM Shift and Rotate instructions. The figure also graphically shows the effect that each operation has on the specified register and on the C (Carry) flag. When you need to perform a shift or rotate operation in one of your programs, you can mentally try each of these operations to find the one you need for your specific application. Note that in Figure 4-3 we used the ADD instruction for the examples, but the operations can be used with any of the data processing instructions. For the BCD packing example here, we just need to shift the 00000101 four bit positions to the left and put zeros in the lower four bit

positions. The Logical Shift Left or LSL Immediate operation seems to be a good choice for this. The simplest instruction for this would have the form MOV R2, R2, LSL #4.

Logical Shift Left

LSL #Immediate or Rs – ADD R2, R1, R3 LSL #4; ADD R2, R1, R3 LSL R4

C flag = last bit shifted out if ADDS. C = 0 if register shift > 32 bits

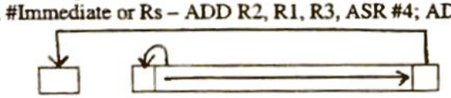
Logical Shift Right

LSR #Immediate or Rs – ADD R2, R1, R3 LSR #4; ADD R2, R1, R3 LSR R4

C flag = last bit shifted out, if ADDS. C = 0 if shift = 0 or register shift > 32 bits

Arithmetic Shift Right

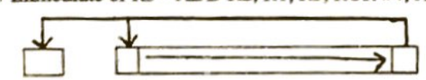
ASR #Immediate or Rs – ADD R2, R1, R3, ASR #4; ADD R2, R1, R3, ASR R4



C Flag = last bit shifted out, if ADDS. Sign bit copied to vacated upper bits

Rotate Right

ROR # Immediate or Rs – ADD R2, R1, R3, ROR #4; ADD R2, R1, R3, ROR R4



C Flag = last bit shifted out, if ADDS

Rotate Right with Extend

RRX - MOVS R2, R2, RRX ; rotates one bit position only

C flag = last bit shifted out, if S on MOV. Then C flag is in rotate loop

Figure 4-3. Operations performed by ARM Shift and Rotate instructions

The final operation in our sequence is to pack the two BCD digits into a single byte. As we mentioned previously, this can be done with a simple ADD instruction of the form ADD R2, R1, R2 or an ORR instruction of the form ORR R2, R1, R2. Figure 4-4 shows a complete assembly language program for our BCD packing algorithm. Note that for this example we simply loaded

some immediate ASCII values into registers. (The assembler will automatically insert the ASCII codes for any text enclosed in single quotation marks.) As you work your way through the program, you should see that it exactly follows the steps along the right side of the development process in Figure 4-2. However, as is often the case, once you write the actual program and think about it more, you find ways to make it more efficient. As an example of this and as a preview of an end-of-chapter problem, you might think about how the Shift and Add operations in the program in Figure 4-4 could be done with a single ARM instruction. In the next section we give another quick example of how you "walk through the bits" to develop an accurate algorithm and program for a sequence of low-level operations.

PackBCD.s

```
@ ARM/XScale Assembly Language Program to Create Packed BCD
@ from two ASCII code values.
@ Copyright Douglas V. Hall, Portland, OR

@ Creates Packed BCD result from two ASCII numerical values
@ ASCII for Most Significant BCD digit in R2@; ASCII for Least Significant BCD digit in R1
@ Packed BCD result left in R2
@ No check is done to determine if the ASCII value supplied represents a
@ valid BCD digit

.text
.global _start
_start:

        MOV R1, #'9'        @ Load ASCII code for 9 in R1
        MOV R2, #'5'        @ Load ASCII code for 5 in R2
        AND R1, R1, #0x0F   @ Mask all but lower 4 bits of R1, leaves BCD
        AND R2, R2, #0x0F   @ Mask all but lower 4 bits of R2, leaves BCD
        MOV R2, R2, LSL #4  @ Shift BCD in R2 4 bit positions left to upper
                            @ nibble position
        ADD R2, R1, R2      @ Combine two BCD digits in R2.
                            @ (Could use ORR R2,R2, R1 instead of ADD)
.end
```

Figure 4-4. ARM assembly language program to convert two ASCII codes to packed BCD

CREATING ROUNDED AVERAGES FOR ELEMENTS OF TWO ARRAYS

You will usually write programs for processing arrays in a high-level language, but here we develop a simple assembly language array program to further show how you "walk the bits" to make sure your algorithm and your translation to assembly language are correct. This example also shows another use of the shift/rotate instructions and reviews how you translate the REPEAT-UNTIL structure into assembly language.

The problem statement for this example is: Given two, 4-element arrays of bytes, calculate the rounded averages for the same numbered elements in the two arrays and put the results in the same numbered element in a third array.

The first step is to think about the data structures required. For this problem, the data is simply three, four-byte arrays. The next step is to think about the sequence of actions that need to be done for each set of elements in the arrays. Figure 4-5 shows the sequence of actions we need to do. We want to get a byte from one array, get a byte from the second array, add the two

bytes, divide by 2 to get the average, round the average, and finally put the rounded average in the appropriate element of the third array. Since this sequence of actions needs to be repeated until all four elements in the source arrays are done, these steps will be enclosed in a REPEAT-UNTIL structure as shown. Now let's see how you can translate the algorithm in Figure 4-5 into an assembly language program.

```
Initialize pointers to Test1, Test2 and Average arrays
Initialize element counter to 4
Repeat
    Load byte from Test1 array
    Load byte from Test2 array
    Add bytes
    Divide sum by 2 to get average
```


```
    Round average - Add 1 to result if remainder in C = 1
    Store rounded average in Average array
    Increment pointers
Until 4 elements done
```

Figure 4-5 Algorithm development for averaging and rounding elements of two arrays of bytes

When writing a new program, you can often save typing time by editing (or doing cut-and-paste from) a copy of a program you wrote before. If you are ambitious, you could also create a program "template" and use that as a base for many of your programs. At any rate, to demonstrate that you can often speed up the development work by building on past work, we deliberately made this averaging example very similar to the example shown in Figure 3-5.

The program in Figure 3-5 implements a REPEAT-UNTIL structure to multiply the elements of two arrays of half words and put the results in a third array. The Repeat-Until structure for the averaging example here will be implemented in the same way as in Figure 3-5. The two major differences between the program in Figure 3-5 and what we need here are: the data arrays in Figure 3-5 are half words and in this program they are bytes; and the major action in Figure 3-5 is multiplication and the major actions in this program are adding, dividing, and rounding. Therefore, we can use the basic structure of the program in Figure 3-6 and just modify it as needed for the program here.

Figure 4-6 shows an assembly language program that implements this array-averaging example. Work your way through the program and mentally execute each of the instructions to understand how each performs the operation in the algorithm. On your way through, note that we used the ARM auto-increment addressing mode with the LDRB and STRB instructions to eliminate the need for separate instructions to increment the array pointers. Also, note that the test numbers we put in the arrays include values that produce a carry from the division and values that do not. It is very important to always test all possible paths through your programs.

The .byte directive can be used to declare a named byte or a named array of bytes as, for example, TEST1: .byte 0x1, 0x2, 0x3, 0x4. The LDRB instruction can be used to load a byte from memory into the lowest byte position in a register and fill the rest of the register with zeros. If, for example, R1 contains the address of the TEST1 array, the instruction LDRB R6, [R1], #1 will load a byte from the array into the least significant byte of R6, fill the upper three bytes of R6 with zeros, and increment R1 to point to the next byte in the TEST1 array.

avg1.s

```
@ ARM/XScale Program to average elements of two arrays
@ Copyright Douglas V. Hall, Portland, OR
@ Averages each element of array TEST1 with same numbered
@ element in array TEST2 and puts the rounded average in
@  the same numbered element in array AVERAGE.

.text
.global _start
_start:
.EQU NUM, 4

        LDR     R1, =TEST1      @ Load Pointer to TEST1 array
        LDR     R2, =TEST2      @ Load Pointer to TEST2 array
        LDR     R3, =AVERAGE    @ Load Pointer to AVERAGE array
        MOV     R4, #NUM        @ Initialize counter
NEXT:   LDRB R6, [R1],#1        @ Get a TEST1 byte, increment pointer
        LDRB    R7, [R2], #1    @ Get a TEST2 byte, increment pointer
        ADDS    R6,R6,R7        @ Add, sum in R6, no carry produced
        MOVS    R6,R6,LSR #1    @ Divide by 2, lsb to carry
        ADC     R6,R6,#0        @ Add contents of carry flag to sum in r6
                                @ to round result up if CY =1
        STRB    R6, [R3], #1    @ Copy result byte to AVERAGE array
        SUBS    R4, R4, #1      @ Decrement element counter, set flags
        BNE     NEXT            @ Repeat loop until 4 averages done

.data
TEST1:    .byte 0x1, 0x2, 0x3, 0x4
TEST2:    .byte 0x2, 0x2, 0x4, 0x4
AVERAGE:  .byte 0x0, 0x0, 0x0, 0x0

.end
```

Figure 4-6 ARM program to average and round elements in two arrays of bytes

As shown in Figure 4-6, the ADD R6, R6, R7 instruction is used to add the two bytes fetched from the source arrays. For this case, we didn't have to put an "S" on the ADD instruction to update the Carry flag after the addition because the addition of two bytes can never produce a carry from the 32-bit R6 register. The ARM-based processors do not have a DIVIDE instruction available to divide the sum of the addition by 2. However, as you should remember, you can divide a binary number by 2^N simply by shifting it N bit positions to the right, or you can multiply a binary number by 2^N simply by shifting it N bit positions to the left. (You might try this on a couple of binary numbers to refresh your memory and/or prove to yourself that it works.) For averaging two numbers, we need to divide the sum by two. Since two is equal to 2^1, we can do this division by shifting the sum one bit to the right. A look at the actions of the Shift and Rotate instructions in Figure 4-3 tells us that the LSR #1 directive added to a data processing instruction will do the shift we need to divide by two. As also shown in Figure 4-3, if we add an "S" to the data processing instruction that we are using with the LSR operation, the LSR #1 operation will also copy the least significant bit of the register into the C (carry) flag. For a division by two, this value in the C flag is the remainder from the division. The MOVS R6, R6, LSR #1 instruction does the desired actions here.

The algorithm for rounding the average states that if this remainder is 1, we want to add one to the average to round it up and if the remainder is 0, we want to add 0 to the average to leave it

the same. To round the average then, all we need to do is add the contents of the C flag to the average with an instruction such as ADC R6, R6, #0. This instruction adds the immediate number #0 plus the bit in the carry flag to R6 and puts the result in R6.

The SRTB R6, [R3], #1 stores the rounded result in the TEST2 array and increments the pointer in R3 by 1 to point to the next element in this array of bytes. The SUBS R4, R4, #1 instruction decrements the element counter by 1 and sets the flags. If the result from the subtraction is not zero, the BNE NEXT instruction causes execution to loop through the REPEAT-UNTIL block again. This loop, of course, is another good example of how you implement a REPEAT-UNTIL structure in ARM assembly language. In the next section, we show you how to implement other IF-THEN-ELSE structures in ARM assembly language.

Developing and Translating IF-THEN-ELSE Structures

INTRODUCTION

As shown in Chapter 3, the basic IF-THEN-ELSE structure has the form:

```
IF some condition is true THEN
    Specified action(s)
ELSE
    Specified action(s)
```

The IF-THEN structure is just a special case of this structure where the ELSE section has no actions. As an example of an IF-THEN structure, the REPEAT-UNTIL loop in the previous section contains an IF-THEN structure that determines whether to repeat the loop actions or just "fall through" to the rest of the program without doing any specific action as part of the structure. In Chapter 3 we also showed you how to implement a nested IF-THEN-ELSE structure. In this section, instead of using a pure programming example, we will use a hardware-based, real-world example to show you how to translate a nested IF-THEN-ELSE structure to assembly language as well as introduce you to how you work with memory-mapped I/O ports in ARM-based processors.

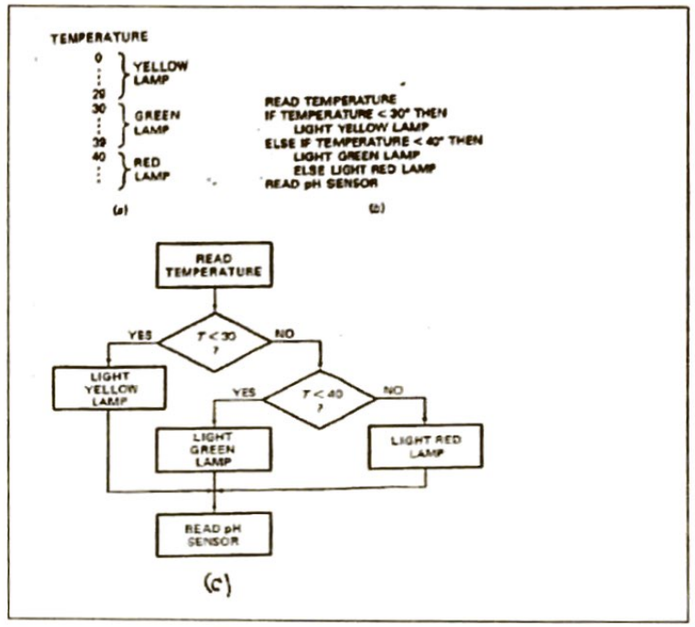


Figure 4-7 Algorithm for PC board machine with temperature A/D and yellow, green, and red LEDs.

Suppose that we are developing a microprocessor-based machine to make printed circuit boards (PC boards). Part of the job of the controller for our machine is to read the temperature value for a tank of solvent from an Analog to Digital (A/D) converter; turn on a yellow LED if the temperature is below 30 degrees Celsius, turn on a green LED if the temperature is equal to or greater than 30 degrees Celsius but less than 40 degrees Celsius; turn on a red LED if the temperature is at or above 40 degrees Celsius. Then the next program section will read the value from another A/D converter connected to a pH sensor and use this value to check/correct the pH of the solution in the tank. Figure 4-7 shows three ways to represent the algorithm for the temperature check section of the program. We will come back to this later when we show how to implement the actual program section but first we need to dig into the hardware connections for this part of the PC board machine.

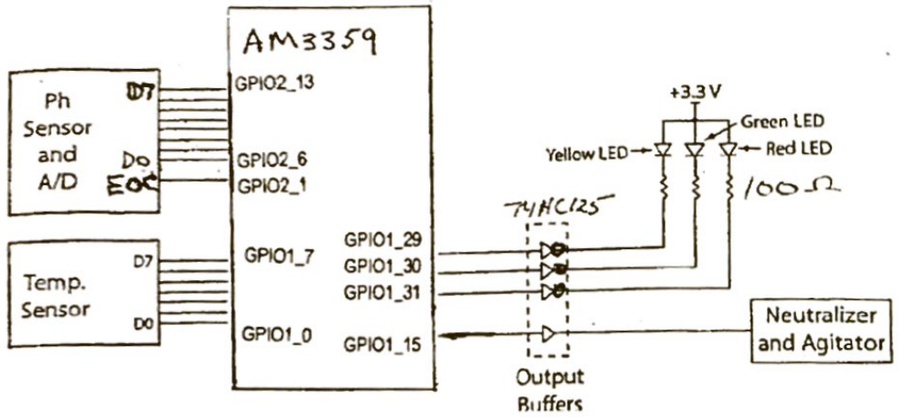


Figure 4-8 Circuit connections for PC board machine with temperature A/D, 3 LEDs, pH A/D and Neutralizer/Agitator

Figure 4-8 shows how the required A/D converters, LEDs, and Ph adjust circuits can be connected to AM3358 GPIO pins available on the P8 connector of a BeagleBone Black board. The next step here is to discuss how the AM3359 GPIO pins are enabled, read, and written to.

THE TI AM 335X GPIO STRUCTURE AND PROGRAMMING

Figure 4-9 shows the four GPIO modules, GPIO0-GPIO3, in the AM3358. Each of these modules has available potentially up to 32 GPIO pins, so there is potentially a total of 128 GPIO pins available on the AM3359. The reason we say there is "potentially" 128 GPIO pins available on the device, instead of simply saying they are available is that the number of pins on the device is limited and not all of the possible internal signals can be connected to device pins to be accessed by external devices. Therefore, many of the actual pins on the device have the output of an 8 input multiplexer, such as the 74151 in Figure 1-8 connected to it. The user programs these multiplexers to connect one of eight internal signals to each pad. The signals chosen depend on the specific application. In Chapter 5 we will show you how to program one of these multiplexers but for now, Figure 4-10 shows the GPIO signals that are multiplexed to the AM3359 pads and connected to the P8 connector on the BeagleBone Black board. For easy reference, Figure 4-10 shows how some of the GPIO signals available on the P8 connector are used for the PC board machine controller.

To minimize power consumption, most of the modules in the AM3359 are turned off when the device is first turned on. This feature allows a designer to just turn on the clocks for the modules needed for a given system application. For basic setup of the DRAM controller for the DDR3 and setup of some other basic circuitry, TI supplies a .GEL file that is then used in the run/debug process. In order to turn on another module, you have to write a control word to the appropriate AM3358 Control Module Power Management register.

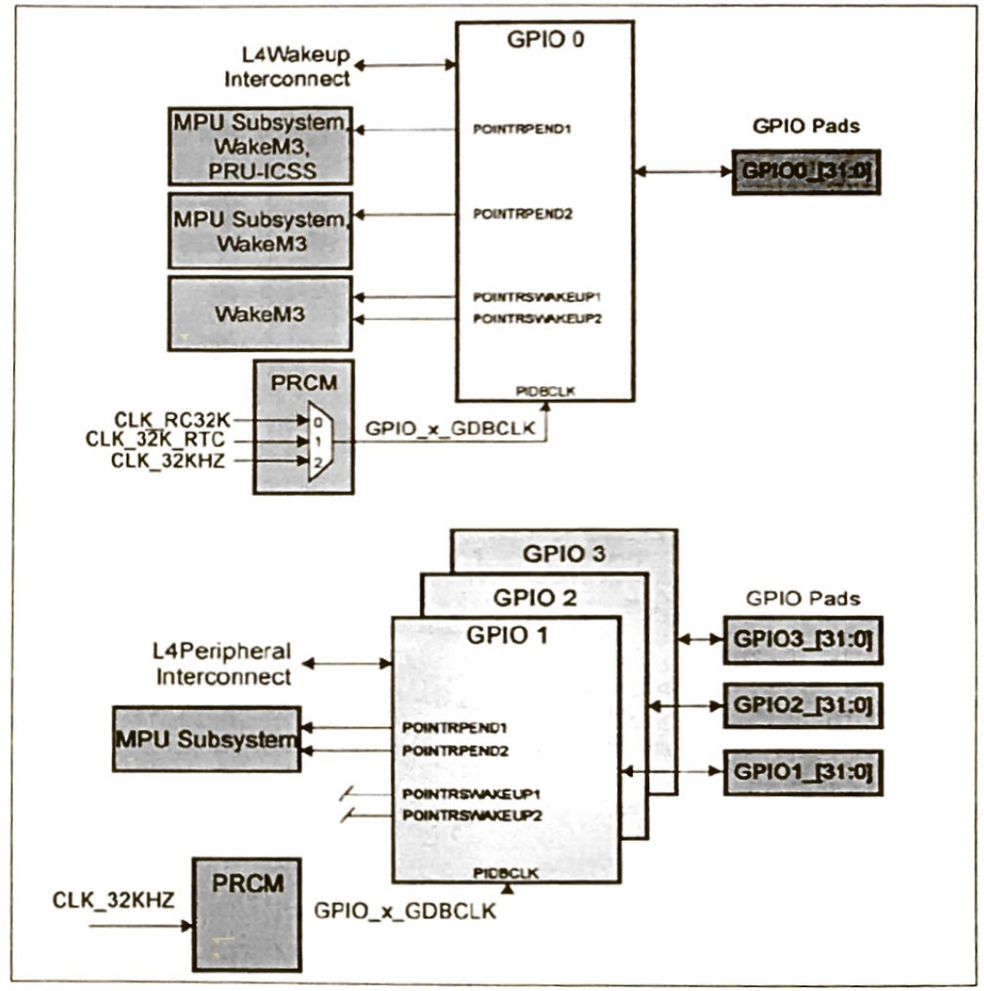


Figure 4-9 Cortex A-8 GPIO modules 0-3 block diagrams

PIN	PROC	NAME	MODE0	MODE1	MODE2	MODE3	MODE4	MOD
1,2					GND			
3	R9	GPIO1_6	gpmc_ad6	mmc1_dat6				
4	T9	GPIO1_7	gpmc_ad7	mmc1_dat7				
5	R5	GPIO1_2	gpmc_ad2	mmc1_dat2				
6	T8	GPIO1_3	gpmc_ad3	mmc1_dat3				
7	R7	TIMER4	gpmc_advn_ale		timer4			
8	T7	TIMER7	gpmc_oen_ren		timer7			
9	T6	TIMER5	gpmc_benn_cle		timer5			
10	U6	TIMER6	gpmc_wen		timer6			
11	R12	GPIO1_13	gpmc_ad13	lcd_data18	mmc1_dat5	mmc2_dat1	eQEP2B_in	
12	T12	GPIO1_12	GPMC_AD12	LCD_DATA18	MMC1_dat4	MMC2_DAT0	EQEP2A_IN	
13	T10	EHRPWM2B	gpmc_ad9	lcd_data22	mmc1_dat1	mmc2_dat5	ehrpwm2B	
14	T11	GPIO0_26	gpmc_ad10	lcd_data21	mmc1_dat2	mmc2_dat6	ehrpwm2_tripzone_in	
15	U13	GPIO1_15	gpmc_ad15	lcd_data16	mmc1_dat7	mmc2_dat3	eQEP2_strobe	
16	V13	GPIO1_14	gpmc_ad14	lcd_data17	mmc1_dat6	mmc2_dat2	eQEP2_index	
17	U12	GPIO0_27	gpmc_ad11	lcd_data20	mmc1_dat3	mmc2_dat7	ehrpwm0_synco	
18	V12	GPIO0_1	gpmc_clk_mux0	lcd_memory_clk	gpmc_wait1	mmc2_clk		
19	U10	EHRPWM2A	gpmc_ad8	lcd_data23	mmc1_dat0	mmc2_dat4	ehrpwm2A	
20	V9	GPIO1_31	gpmc_csn2	gpmc_be1n	mmc1_clk			
21	U9	GPIO1_30	gpmc_csn1	gpmc_clk	mmc1_clk			
22	V8	GPIO1_5	gpmc_ad5	mmc1_dat5				
23	U8	GPIO1_4	gpmc_ad4	mmc1_dat4				
24	V7	GPIO1_1	gpmc_ad1	mmc1_dat1				
25	U7	GPIO1_0	gpmc_ad0	mmc1_dat0				
26	V6	GPIO1_29	gpmc_csn0					
27	U5	GPIO2_22	lcd_vsync	gpmc_a8				
28	V5	GPIO2_24	lcd_pclk	gpmc_a10				
29	R5	GPIO2_23	lcd_hsync	gpmc_a9				
30	R6	GPIO2_25	lcd_ac_bias_en	gpmc_a11				
31	V4	UART5_CTSN	lcd_data14	gpmc_a16	eQEP1_index	mcasp0_axr1	uart5_rxd	
32	T5	UART5_RTSN	lcd_data15	gpmc_a19	eQEP1_strobe	mcasp0_aclkx	mcasp0_axr3	
33	V3	UART4_RTSN	lcd_data13	gpmc_a17	eQEP1B_in	mcasp0_fsr	mcasp0_axr3	
34	U4	UART3_RTSN	lcd_data11	gpmc_a15	ehrpwm1B	mcasp0_aclkr	mcasp0_axr2	
35	V2	UART4_CTSN	lcd_data12	gpmc_a18	eQEP1A_in	mcasp0_axr3	mcasp0_axr2	
36	U3	UART3_CTSN	lcd_data10	gpmc_a14	ehrpwm1A	mcasp0_axr0		
37	U1	UART5_TXD	lcd_data8	gpmc_a12	ehrpwm1_tripzone_in	mcasp0_aclkx	uart5_txd	
38	U2	UART3_RXD	lcd_data9	gpmc_a13	ehrpwm0_synco	mcasp0_fsx	uart5_rxd	
39	T3	GPIO2_12	lcd_data6	gpmc_a6		eQEP2_index		
40	T4	GPIO2_13	lcd_data7	gpmc_a7		eQEP2_strobe	prt_edio_data_out7	
41	T1	GPIO2_10	lcd_data4	gpmc_a4		eQEP2A_in		
42	T2	GPIO2_11	lcd_data5	gpmc_a5		eQEP2B_in		
43	R3	GPIO2_8	lcd_data2	gpmc_a2		ehrpwm2_tripzone_in		
44	R4	GPIO2_9	lcd_data3	gpmc_a3		ehrpwm0_synco		
45	R1	GPIO2_6	lcd_data0	gpmc_a0		ehrpwm2A		
46	R2	GPIO2_7	lcd_data1	gpmc_a1		ehrpwm2B		

Figure 4-10 BeagleBone Black P8 Header Connections

To turn on the clocks to all four of the GPIO modules, the instructions below are added to the initialization section at the start of any program using one or more of the GPIO modules. Of course, if all 4 modules are not needed, you can save power by just turning on clocks to the ones you need by writing 0x02 to the CLKCTRL Register in that module..

@ Instructions to enable clocks to GPIO Modules in AM3358 Processor

```
LDR  R0,#0x02                @ Value to enable clock for a GPIO module
LDR  R1,= 0x44E00408  @Address of CM_WKUP_GPIO0_CLKCTRL Register
STR R0, [R1]                 @ Write to register
LDR R1,=0x44E000AC   @ Address of  CM_PER_GPIO1_CLKCTRL Register
STR R0, [R1]                 @ Write #02 to register
LDR R1,=0x44E000B0   @ Address of  CM_PER_GPIO2_CLKCTRL Register
STR R0, [ R1]                @ Write #02 to register
LDR R1,=0x44E000B4   @ Address of  CM_PER_GPIO3_CLKCTRL Register
STR R0, [R1]                 @ Write #02 to register
```

Later we will explain in detail how you determine these addresses and the value to write to them but for now, as shown in the L4_WKUP Peripheral Memory Map in the Appendix, the base

address for the CM_WKUP registers is 0x44E00400 and as shown in section 8.1.12.2.2 of the AM 3358 data sheet, the offset of the ***CM_WKUP_GPIO0_CLKCTRL Register*** is 0x8. The address of the register then is 0x44E00408. Likewise, the base address of the other three registers that are under CM_PER line in the L4_WKUP Peripheral Memory Map in the Appendix is 0x44E00000. The offsets for the three CLKCTRL registers are 0xAC, 0xB0, and 0xB4, so the three address, as shown above, are 0x44E000AC, 0x44E000B0, and 0x44E000B4.

Once the GPIO modules are turned on, they are controlled by memory mapped registers. According to the Cortex A-8 memory map in the Appendix, the base addresses for the four GPIO module control registers are:

GPIO0 base address – 0x44E07000
GPIO1 base address – 0x4804C000
GPIO2 base address – 0x481AC000
GPIO3 base address – 0x481AE000

Figure 4-11 shows the offsets that you add to the base address of a module to access the desired control register in that module. A quick read through the register list in Figure 4-11 shows there are registers for reading data, outputting data, enabling a pin to function as an output, setting an output to logic high, clearing an output to logic low, detecting a rising edge on a signal, detecting a falling edge on a signal and one of several other functions. We will start with a discussion of the GPIO Output Enable register.

When the GPIO modules are turned on, all the GPIO module signal lines are automatically set as inputs. If lines were initially set as outputs, and one of the outputs was outputting a logic high at the same time that a connected device was outputting a low on the connecting line, the high and low and likely destroy the GPIO output transistor. (Remember from basic digital design that you never connect one digital device output to another unless they are three-state and only one is enabled at a time. In order to have one or more of the GPIO lines in a module function as an output, you have to write a control word to the GPIO_OE register for those pins.

0h	GPIO_REVISION	Section 25.4.1.1
10h	GPIO_SYSCONFIG	Section 25.4.1.2
20h	GPIO_EOI	Section 25.4.1.3
24h	GPIO_IRQSTATUS_RAW_0	Section 25.4.1.4
28h	GPIO_IRQSTATUS_RAW_1	Section 25.4.1.5
2Ch	GPIO_IRQSTATUS_0	Section 25.4.1.6
30h	GPIO_IRQSTATUS_1	Section 25.4.1.7
34h	GPIO_IRQSTATUS_SET_0	Section 25.4.1.8
38h	GPIO_IRQSTATUS_SET_1	Section 25.4.1.9
3Ch	GPIO_IRQSTATUS_CLR_0	Section 25.4.1.10
40h	GPIO_IRQSTATUS_CLR_1	Section 25.4.1.11
44h	GPIO_IRQWAKEN_0	Section 25.4.1.12
48h	GPIO_IRQWAKEN_1	Section 25.4.1.13
114h	GPIO_SYSSTATUS	Section 25.4.1.14
130h	GPIO_CTRL	Section 25.4.1.15
134h	GPIO_OE	Section 25.4.1.16
138h	GPIO_DATAIN	Section 25.4.1.17
13Ch	GPIO_DATAOUT	Section 25.4.1.18
140h	GPIO_LEVELDETECT0	Section 25.4.1.19
144h	GPIO_LEVELDETECT1	Section 25.4.1.20
148h	GPIO_RISINGDETECT	Section 25.4.1.21
14Ch	GPIO_FALLINGDETECT	Section 25.4.1.22
150h	GPIO_DEBOUNCENABLE	Section 25.4.1.23
154h	GPIO_DEBOUNCINGTIME	Section 25.4.1.24
190h	GPIO_CLEARDATAOUT	Section 25.4.1.25
194h	GPIO_SETDATAOUT	Section 25.4.1.26

Figure 4-11 ARM Cortex A-8 GPIO Control register functions and offsets from base address

Figure 4-12 shows the AM3359 data sheet section for the GPIO_OE register. Note that, after a reset, all of the 32 register bits are 1, so all the outputs are set as inputs. To enable a GPIO line as an output, you write a 1 to that bit. As a first example of how to do this correctly, suppose that you want to initialize pin 22 of the GPIO1 module to function as an output and have an initial high state.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OUTPUTEN[n]																															
R/W-FFFFFFFFh																															

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Bit	Field	Type	Reset	Description
31-0	OUTPUTEN[n]	R/W	FFFFFFFFh	Output Data Enable 0h = The corresponding GPIO port is configured as an output. 1h = The corresponding GPIO port is configured as an input.

Figure 4-12 GPIO_OE register data sheet – GPIO_OE register & field descriptions

To enable bit 22 as output, you have to write a 0 to bit 22 of the GPIO1_OE register. To accurately keep track of the bits when working with these 32-bit control registers, you write the required bit(s) in a template such as that shown in Figure 4-13. As shown in Figure 4-13, we put a 0 in bit 22 and 1's in all the rest of the bits. By marking off each group of 4 bits in the template,

it is easy to see that the hexadecimal value for the required control word to enable bit 22 is 0xFFBFFFFF.

GPIO1 bit#	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Outputs	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1
Inputs																
Hex Val	F				F				B				F			

GPIO1 bit#	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Outputs	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Inputs																
Hex Val	F				F				F				F			

Figure 4-13 Template for initializing GPIO1_22 as output

To determine the address of the GPIO1_OE register you look at Figure 4-11 to find that the offset for a GPIO_OE register is 0x134. Since you are working with the GPIO1 register module, you add this offset to 0x4804C000, the base address of the GPIO1 register set. The address of the GPIO1_OE register is then 0x4804C134. However, you do not write this control word directly to the GPIO1_OE register. The reason for this is that you only want to change the GPIO_22 line and not the other 31 GPIO1 lines. The correct way to do this is use a *Read-Modify-Write process*, that is done as follows. You Read the current value from the GPIO1_OE, Modify the word by ANDing the value read from GPIO1_OE with a word that has zeros in the bits you want to make outputs and ones in all the rest of the bits. Then you write the modified word back to the GPIO_OE register.

For this example, the sequence of instructions would be:

```
LDR  R0,= 0xFFBFFFFF @ Load word to program GPIO22 as output
LDR  R1,= 0x4804C134 @ Address of GPIO1_OE register
LDR  R2, [R1]        @ READ GPIO1_OE register
AND R2, R2, R0       @ MODIFY word read in
STR  R2, [R1]        @ WRITE back to GPIO1_OE register
```

You always have to use this *Read-Modify-Write* process whenever you change a value on a control register, so that you only change the desired bits. You will see this used many time in the examples throughout the rest of the book.

The next step in this first example is to figure out how to output a one on the pin connected to GPIO1_22 line. Your first thought might be to just write a word with a 1 in bit 22 to the GPIO1_DATAOUT register at offset 0x13C in the GPIO1 register set. The problem with using this method is that you would have to use the *Read-Modify-Process* to do it, so that you don't change other output bits. To avoid having to do this three step process for every output operation, the designers of the ARM processors have included direct set and clear ability. To output a logic 1 on a set of GPIO pins programmed as outputs, you can simply write a word with 1's in the bit positions for which you want to output a 1 and zeros in all the other bit positions to the GPIO_SETDATAOUT register at offset 0x194 in the GPIO register set. To output a logic 0 on a set of GPIO pins programmed as outputs, you can simply write a word with 1's in the bit

positions for which you want to output a 0 and zeros in all the other bit positions to the GPIO_CLEARDATAOUT register at offset 0x190 in the GPIO register set. The key here is that for either of these, a 1 in a bit tells the controller to do the Set or Clear operation for that bit position and a 0 in a bit position tells the controller to do nothing to the bit in that position. For this first example we specified that you want a 1 on the pin connected to GPIO1_22. You can do this by simply writing a word with a 1 in bit 22 to the GPIO1_SETDATAOUT register at 0x4804C000 + 0x194 or 0x4804C194. You do this before you make GPIO1_22 and output as follows:

```
MOV R5, 0x00400000    @ Load value to set GPIO1_22 to logic high when output
LDR R6,= 0x4804C194   @ Load address of GPIO1_SETDATAOUT register
STR R5,[R6]           @ Write to GPIO1_SETDATAOUT register
LDR  R0,= 0xFFBFFFFF  @ Load word to program GPIO22 as output
LDR  R1,= 0x4804C134  @ Address of GPIO1_OE register
LDR  R2, [R1]         @ READ GPIO1_OE register
AND R2, R2, R0        @ MODIFY word read in
STR  R2, [R1]         @ WRITE back to GPIO1_OE register
```

The reason you set the GPIO_22 signal to the desired 1 value before programming it to be an output is so that the output pin will immediately have the desired logic level on it when the pin is converted to an output. Now that you know how to initialize, read from and write to GPIO pins, the next step is to get back to developing the actual program for the 3 LED temperature sensor algorithm in Figure 4-7.

The first step in developing a program such as this is to do a high level algorithm such as that in Figure 4-7. This algorithm is a top level view and does not reference any registers or specific GPIO pins, so it can be used to develop the program for any microprocessor. When you know the specific machine the program is going to run on and the specific GPIO pins, then you can develop a lower level algorithm that will make it easy for you to write an assembly language or C program for the system. This low level algorithm and task list contains control register addresses, required initialization words, etc. based on the circuit connections and the desired actions. It is very important to carefully work out all of the values for this and then just translate the result into assembly language or C.

GPIO 1 bit#	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Function	R	GR	Y													
OE	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
Hex Val	1				F				F				F			
OUT (Y)	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Hex Val	2				0				0				0			

GPIO 1 bit#	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Function									Temperature A/D							
OE	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Hex Val	F				F				F				F			
OUT (Y)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Hex Val	0				0				0				0			

Figure 4-14 GPIO1 template for 3-LED Temperature Sensor

Based on many years of successful experience, I can tell you that unless you really like debugging, you never try to write a program directly in assembly language or in any language, without developing this worked out, low-level algorithm. For the algorithm shown in Figure 4-7, the GPIO connections shown in Figure 4-8, and the register template in Figure 4-14, an appropriate low level algorithm would be the one shown in Figure 4-15.

```
Note: Base address for GPIO1_ control registers is 0x4804C000
Set GPIO1 bits 29-31 to low by write 0xE0000000 to GPIO1_CLEARDATAOUT at
    0x4804C190 (0x4804C000+ 0x190 offset for GPIO_CLEARDATAOUT)
Set GPIO1 bits 29-31 to outputs by RMW 0x1FFFFFFF to GPIO1_EN at
    0x4804C134 (0x4084C000 + 0x134 offset for GPIO1_EN register)
Read Temperature data from GPIO1_0-7 at 0x4804C138
    (0x4804C000 + offset 0x138 for GPIODATAIN register)
Mask all but 8 LSB with 0x000000FF
Compare with 30
If less than 30
    Go load 0x20000000 value to turn on yellow LED and send
Else
    Compare with 40
    If less than 40
      Go load 0x40000000 value for green LED and send
    Else
      Load value of 0x80000000 value for red LED and go send.
Write LED value to GPIO1_SETDATAOUT register at 0x4804C194
      (0x4804C000 + offset of 0x194 for GPIO1_SETDATAOUT)
```

Figure 4-15 Low-Level Algorithm for 3 LED Temperature Sensor Program

Note the most programs will start with a sequence of statements to initialize GPIO pins and peripheral device controllers such as UARTS, Timers, etc.

```
@ PC Board Machine Temperature Sensor program
@ Runs on BeagleBone Black Board.
@ Program reads a binary temperature value from an A/D conected to GPIO1_0-7.
@ If the temperature is below 30 degrees the program lights a yellow LED connected
@ to GPIO1_29. If the temperature is in the  range of 30-39 degrees, the program
@ will light a green LED connected to GPIO1_30. If the temperature is at or above
@ 40 degrees, the program will light a red LED connected to GPIO1_31.
@ Copyright Douglas V. Hall Fall 2016

.text
.global _start

_start: LDR  R0,#0x02          @ Value to enable clocks for a GPIO modules
        LDR  R1,= 0x44E00408   @Address of CM_WKUP_GPIO0_CLKCTRL Register
        STR R0, [R1]           @ Write to register
        LDR R1,=0x44E000AC     @ Address of  CM_PER_GPIO1_CLKCTRL Register
        STR R0, [ R1]          @ Write #02 to register
        LDR R1,=0x44E000B0     @ Address of  CM_PER_GPIO2_CLKCTRL Register
        STR R0, [R1]           @ Write #02 to register
        LDR R1,=0x44E000B4     @ Address of  CM_PER_GPIO3_CLKCTRL Register
```

```
                STR R0, [R1]                @ Write #02 to register
        LDR R0,=0x4804C000      @ Base address for GPIO1 registers
                ADD R2, R0,#0x138       @ GPIO1 DATAIN register to read A/D on GPIO1_0-7
@ Load value to turn off all three LEDS
                MOV R7,#0xE0000000      @ GPIO1 29-31 off with GPIO1_CLEARDATAOUT register
                ADD R4,R0,#0x190        @ Make GPIO1_CLEARDATAOUT register address
                STR R7,[R4]             @ Write to GPIO1_CLEARDATAOUT register
@ Program GPIO1_29-31 as outputs
                ADD R1,R0,#0x0134       @ Make GPIO1_OE register address
                LDR R6,[R1]             @ READ current GPIO1 Output Enable register
                MOV R7,#0x01FFFFFF      @ Word to enable GPIO1_29-31 as outputs
                AND R6,R7,R6            @ Clear bits 29-31 (MODIFY)
                STR R6, [R1]            @ WRITE to GPIO1 Output Enable register
@ Read Temperature from A/D outputs
                LDR R8,[R2]             @ Read GPIO1_0-7 port to get value from A/D
                MOV R9,#0x000000FF      @ Mask for all but 8 LSB
                AND R8,R8,R9            @ Mask all but 8 LSB that hold Temp value
                CMP R8,#30              @ Compare Temp with lowest OK value
                BMI YELLOW              @ Less than 30, light yellow LED
                CMP R8, #40             @ Else, check if less than 40
                BMI GREEN               @ Yes, go light green LED
                MOV R9,#0x80000000      @ Else load value for red LED
                B SEND                  @ Go send
YELLOW:         MOV R9,#0x20000000      @ Load value to light yellow LED
                B SEND                  @ Go send
GREEN:  MOV R9,#0x40000000      @ Load value for Green LED
SEND:   ADD R5,R0,#0x194        @ Load address of GPIO1_SETDATAOUT register
                STR R9,[R5]             @ Use GPIO1_SETDATAOUT register to light desired LED
                NOP                     @
.end
```

Figure 4-16. Assembly Language program for 3-LED Temperature Sensor

Figure 4-16 shows the assembly language program developed from this low level algorithm. Skim through the program and comments to get an overview. As shown by the comments in the program, the low level algorithm translates directly into the assembly language program.

Now in the program note that to create the address for one of the GPIO1 control registers, we added the offset for that register to the control register base address in R0 with an instruction such as ADD R1, R0, 0x134, instead of loading R1 directly with an LDR as we did to load the base address in R0. To understand the reason for this, remember that an LDR instruction such as the LDR R0,=0x4804C000 instruction has to access the literal pool in memory to get the 32-bit address and this takes time. An instruction such as ADD R1, R0, #0x134 does not require access to the literal pool and executes in just one clock cycle. Also, note that we used a MOV instruction in the MOV R7, #0xE0000000, instead of an LDR R1,= #0xE0000000 instruction. As we will show more a little later in the chapter, if the immediate number has less than eight 1 bits in it, the assembler can code it as a MOV instruction that only takes one clock cycle. Also note that we used the Branch if Minus instruction, BMI, to branch to YELLOW: and GREEN:. Remember from the examples in Chapter 3 that the Compare instruction does the compare by subtracting the second operand from the first. If the second number is larger than the first, then the result will be minus.

You can extend this "branch or fall through" translation to implement any number of nested IF-THEN-ELSE structures or a CASE structure. In a later section of the chapter on program optimization, we show that in some applications you can use the ARM conditional execution capability to implement simple IF-THEN-ELSE structures without having to use branch

instructions. However, a very important point here is that the assembly language implementation of this structure should always have one entry point and one exit point shown by the algorithm in Figure 4-7 and the program in Figure 4-16. If you follow this one-entry, one-exit rule for all structures it will make your debugging much easier. Next here we will make a few more comments about how you translate REPEAT-UNTIL structures into ARM assembly language and show an example of how you translate a WHILE-DO structure into ARM assembly language.

Developing and Translating REPEAT-UNTIL and WHILE-DO Structures

The array multiplication program in Figure 3-6 and the array average program in Figure 4-6 showed you how to implement a simple REPEAT-UNTIL structure in assembly language. The main point in implementing a REPEAT-UNTIL structure is that you do the action(s) in the structure once and then check if the desired result has been achieved. If not, you repeat the action(s) in the loop and then check again to see if the desired result has been achieved.

To basic principle of a WHILE-DO structure is that you check the condition first before doing the action(s) in the structure. For our WHILE-DO example, we will use an extension to the PC board machine we used for two previous examples. Specifically, we will add the capability to measure the pH of the solution in a tank and if the pH is above 7, add an acid and agitate to reduce the pH to the desired value of 7. Figure 4-8 shows the PC board machine connections with the additional connections needed for measuring the pH and turning on the neutralizer/agitator.

AM335X GPIO PROGRAMMING FOR A PH CORRECTION PROGRAM

The pH sensor produces an analog voltage proportional to the pH of the solution. To obtain the digital value we need for our program, we use another A/D converter. We will discuss the operation and interfacing of A/D converters in detail in Chapter 9, but for this example all you need to know is that the A/D converter produces an 8-bit binary value proportional to the pH of the solution and that the converter is set up so that it is continuously producing a digital value for the analog voltage produced by the pH sensor. Each time the A/D converter completes a conversion, it outputs the new 8-bit binary value and asserts an End-Of-Conversion, EOC, signal high. As you will soon see, we "poll" or, in other words, read this EOC signal over and over until we find it asserted high and then read the data value from the parallel data lines of the A/D converter. As shown in Figure 4-8, the 8 data outputs of the pH A/D converter are connected to GPIO2_6-13, the pH A/D EOC signal is connected to GPIO2_1, and the pH Neutralizer/Agitator control line is connected to GPIO1_15. Figure 4-17 shows the GPIO2 register template for the pH section data and EOC connections. Figure 4-18a shows the high level algorithm for this program section.

GPIO 2 bit#	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Function																
OE	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Hex Val	F				F				F				F			
OUT (Y)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Hex Val	0				0				0				0			

GPIO2 bit#	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Function			pH sensor A/D outputs												EOC	
OE	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Hex Val	F				F				F				F			
OUT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Hex Val	0				0				0				0			

Figure 4-17. GPIO2 template for pH Program Section

Specifically, this program section checks the pH of the chemical solution in the PC board machine and, if the pH is above the desired value of 7, adds an acid solution in small increments to reduce the pH toward 7. The key point here is that we obviously want to check the condition (current pH value) before we add any of the neutralizing acid and agitate the solution. The overall point of the algorithm is that WHILE the pH is greater than 7, we want to DO the following: read a new value from the A/D converter, compare this value with the value equivalent to a pH of 7 and, if the pH is greater than 7, turn on the neutralizer and the agitator for a few seconds. To determine when a valid value is available on the data outputs of the A/D, we implement a REPEAT-UNTIL structure that polls the EOC line over and over until it finds the EOC line high. We then read the 8-bit data value from the A/D converter and compare it with the value that corresponds to a pH of 7. As you probably remember, the pH scale ranges from 0 to 14. Acids have a pH below 7 and bases have a pH above 7. A solution with a pH of 7, the midpoint of the scale, is considered neutral. The output data values for an 8-bit A/D converter with straight binary-coded outputs ranges from 00000000 to 11111111 or in hexadecimal from 00-FF. Since there are an even number of values, the midpoint of this range (equivalent to a pH of 7) is halfway between 0x00 and 0xFF. For this example, we will use a value of 0x7F as equivalent to a pH of 7, because this is accurate to within ½ of a least significant bit for the converter. You could just as well justify using 0x80.

If the pH is above 7, we pulse the neutralizer and the agitator for a few seconds. The neutralizer drips an acid into the solution, and the agitator stirs the solution to make sure it is thoroughly mixed. To pulse the neutralizer and agitator, we just output a high and then a low on the port pin connected to these devices. In Chapter 8 we show you the details of how to interface with solenoid valves, motors, etc. but for now, all you need to know is that a logic high turns these devices on and a logic low turns them off.


```
Turn off Neutralizer and set pin as output.
WHILE ph>7  DO
        REPEAT
                Read EOC line
                Check level
        UNTIL EOC line high
        Read pH value from A/D
        Compare with Ph value for 7 (0x7F)
        IF pH > 7
                Turn on neutralizer drip valve and agitator
                Wait 5 seconds
                Turn off neutralizer drip valve and agitator
END WHILE
```

Figure 4-18a High level algorithm for WHILE-DO loop that adds neutralizing solution if pH > 7

```
Turn off Neutralizer and set pin as output. (Write word with 1 in Bit 15 to GPIO1_CLEARDATA OUT REGISTER
at 0x4804C4190. Write word with 0 in bit 15 to GPIO1_OE register at 0x480C4134)
WHILE ph>7  DO
        REPEAT
                Read EOC line bit 1 of GPIO2_DATAIN register at 0x481AC138
                Check level
        UNTIL EOC line high
        Read pH value from A/D on bits 6-13 GPIO2_DATAIN register at 0x481AC138
        Compare with Ph value for 7 (0x7F)
        IF pH > 7
                Turn on neutralizer drip valve and agitator- word with 1 in bit 15 to
                        GPIO1_SETDATAOUT register at 0x4804C194
                Wait 5 seconds with delay loop
                Turn off neutralizer drip valve and agitator- word with 1 in bit 15 to
                        GPIO1_CLEARDATAOUT register at 0x4804C190
END WHILE
```

Figure 4-18b Low level algorithm for WHILE-DO loop that adds neutralizing solution if pH > 7

Figure 4-18b shows the low level algorithm for the pH program section. This is mostly just the high level algorithm with control register names, addresses, and bit values added as needed to write the assembly language version. Figure 4-19 shows the assembly language program for the low level algorithm. Note that since the Neutralizer/agitator is connected to GPIO1_15, this line needs to be asserted low to start the neutralizer in the off state and initialized as an output. Since we already initialized GPIO1_29-31 low and as outputs with GPIO1_CLEARDATAOUT and GPIO_EN registers for the temperature section, we simply modified the words in these registers in the temperature to initialize GPIO1_15 as needed for the pH section. By doing this, we don't have to repeat the instructions in the pH section.

The EOC signal that we need to poll to determine if the A/D has a new value is on GPIO2_1, so to get this value we read the GPIODATAIN2 register. A simple way to check this bit is to rotate bit 1 of the word read 2 bits left into the carry flag with the MOVS R2, R2, LSR#2 instruction. The Branch if Carry Clear instruction, BCC, is used to implement the IF-THEN-ELSE. If the Carry flag is Clear, execution branches back to the POLL label and the value on the outputs of the A/D converter is read again.

The data outputs of the A/D converter are connected to GPIO2_6-13. To make it easy to compare the value with the setpoint value of 0x7F, we would like to have the 8-bit data

```
@ PC Board Machine Temperature Sensor and Ph adjust program
@ Runs on BeagleBone Black Board.
@ Program reads a binary temperature value from an A/D conected to GPIO1_0-7.
@ If the temperature is below 30 degrees the program lights a yellow LED connected
@ to GPIO1_29. If the temperature is in the  range of 30-39 degrees, the program
@ will light a green LED connected to GPIO1_30. If the temperature is at or above
@ 40 degrees, the program will light a red LED connected to GPIO2_6-13.
@ The pH section of the program reads an 8-bit pH value from an A/D connected to GPIO2_6-13.
@ Data is read from A/D when End of Conversion (EOC) signal on GPIO2_1 is high.
@ While the Ph. is greater than 7.0, the program cyles through an "add acid and agitate"
@ loop using a neutralizer connected to GPIO1_15.
@ Copyright Douglas V. Hall Fall 2016
.text
.global _start
_start:
        LDR  R0,#0x02          @ Value to enable clocks for a GPIO modules
LDR  R1,= 0x44E00408           @Address of CM_WKUP_GPIO0_CLKCTRL Register
STR R0, [R1]                   @ Write to register
LDR R1,=0x44E000AC@ Address of CM_PER_GPIO1_CLKCTRL Register
STR R0, [R1]                   @ Write #02 to register
LDR R1,=0x44E000B0 @ Address of CM_PER_GPIO2_CLKCTRL Register
STR R0, [R1]                   @ Write #02 to register
LDR R1,=0x44E000B4 @ Address of CM_PER_GPIO3_CLKCTRL Register
STR R0, [R1]                   @ Write #02 to register
LDR R0,=0x4804C000     @ Base address for GPIO1 control registers
        LDR R7,=0xE0008000     @ GPIO1_29-31,15 with GPIO1_CLEARDATAOUT register
        ADD R4,R0,#0x190       @ Make GPIO1_CLEARDATAOUT register address
        STR R7,[R4]            @ Write to GPIO1_CLEARDATAOUT register
@ Program GPIO1_29-31 as outputs
        ADD R1,R0,#0x0134      @ Make GPIO1_OE register address
        LDR R6,[R1]            @ (READ) current GPIO1 Output Enable register
        LDR R7,=0x01FF7FFF     @ Word to enable GPIO1_29-31,15 as outputs
        AND R6,R7,R6           @ Clear bits 29-31,15 (MODIFY)
        STR R6, [R1]           @ (WRITE) to GPIO1 Output Enable register
@ Read Temperature from A/D outputs
        ADD R2,R0,#0x138       @ Load R2 with address GPIO1_DATAIN register
        LDR R8,[R2]            @ Read GPIO1_0-7 port to get value from A/D
        MOV R9,#0x000000FF     @ Mask for all but 8 LSB
        AND R8,R8,R9           @ Mask all but 8 LSB that hold Temp value
        CMP R8,#30             @ Compare Temp with lowest OK value
        BMI YELLOW             @ Less than 30, light yellow LED
        CMP R8, #40            @ Else, check if less than 40
        BMI GREEN              @ Yes, go light green LED
        MOV R9,#0x80000000     @ Else load value for red LED
        B SEND                 @ Go send
YELLOW: MOV R9,#0x20000000     @ Load value to light yellow LED
        B SEND                 @ Go send
GREEN:  MOV R9,#0x40000000     @ Load value for Green LED
SEND:   ADD R5,R0,#0x194       @ Load address of GPIO1_SETDATAOUT register
        STR R9,[R5]            @ GPIO1_SETDATAOUT register to light desired LED
@ pH SECTION OF PROGRAM STARTS HERE
        LDR R1,=0x481AC138     @ Address of GPIO2 DATAIN register
```



```
POLL:   LDR R2,[R1]            @ Read EOC and pH data from A/D on GPIO2
        MOVS R2, R2, LSR #2    @ Shift EOC in bit 1 to Carry flag
        BCC POLL               @ Poll EOC until Carry set by EOC high
        LDR R2,[R1]            @ Read strobe and data from A/D on GPIO2 again
        MOV R2, R2, LSR #6     @ Shift pH data on GPIO2_6-13 to LSB position
        CMP R2, #0x7F          @ Compare with value for Ph 7
        BLS DONE               @ Done if pH <7, else turn on neutralizer
        MOV R3, #0x00008000    @ Value to turn neutralizer on/off GPIO1_15
        STR R3, [R5]           @ Send to GPIO1 SETDATAOUT register to turn on
        LDR R7,=0x00FFFFFF     @ Load delay loop constant
LOOP:   NOP
        SUBS R7, R7, #1        @ Decrement loop counter
        BNE LOOP
        STR R3,[R4]            @ Turn Neutralizer off,GPIO1 CLEARDATAOUT register
        B POLL                 @ Loop back,read pH value to see if in range now
DONE:   NOP
        NOP
.end
```

Figure 4-19. Arm assembly language program for PC Board machine with pH correction.

value in the LSBs of a register. This is easy to accomplish with the MOV R2, R2, LSR #6 (Logical Shift Right) instruction. We then mask off all but the 8 LSB with an AND R2, R2, 0x000000FF instruction to remove any signals from other devices connected to GPIO2 input pins.

After the Compare instruction, the Branch if Less than or Equal instruction, BLS sends execution to the DONE: label if the pH is less than or equal to 0x7F. Else, execution falls through to the instructions that turn on the Neutralizer for a few seconds. The timing is done by loading a large value into a register and counting the value down with a Repeat-Until loop. The delay time is equal to the time it takes to execute the loaded number times around the loop. If we assume a 1GHz clock then each clock period is 1 ns. If we assume each instruction takes one clock cycle then each execution of the two instructions in the loop required 2 ns. To get a delay of 5 seconds then we need 5sec/2ns or ideally 0x9502F900 loops. Note that if the required time produces a value that is larger than 32 bits, you can add more NOPs to increase the time of each loop or you can create a nested loop. We leave it for you to figure out how to create a nested loop as a problem at the end of the chapter and in the next chapter we show you how to use programmable hardware timers to measure off precise time intervals. Incidentally, the NOP or No Operation instruction that we used in this section is used to take up time or provide an instruction to use for a breakpoint during debugging. A NOP instruction does not change any processor registers or memory locations. It simply fills one time slot in the pipeline. For a variety of reasons, it usually requires some experimentation to get close timing with a delay loop and the value will have to be modified if the Processor clock speed is changed. In the next chapter we show you how to use a programmable timer/counter to accurately measure times and time actions such as this.

To summarize this section, we showed here that for a REPEAT-UNTIL structure, you do the desired action(s) and then check the decision condition, whereas with a WHILE-DO structure you check the decision condition first and then do the action(s). In the next section of the chapter we show you more techniques you can use to improve the memory efficiency and execution speed of your programs. We are introducing these techniques relatively early in your climb up

the assembly language programming ramp in the hope that they will become automatic to you as you write programs.

Optimization Techniques for Arm Assembly Language

The program examples in the first section of the chapter introduced you to some optimization techniques for ARM assembly language programs. This section of the chapter summarizes these techniques and systematically shows other optimization techniques that you should use in your programs as often as possible. It is especially important to use these tricks in program sections that must execute as fast as possible. We have formatted this section as lists so that you can periodically do a "memory refresh cycle" by reading the list entries and then review the examples to get any details you may need.

GENERAL INSTRUCTION OPTIMIZATION TECHNIQUES

1. **Use the MOV and MVN instructions to load constants into registers instead of using the LDR RN, =32-bit value instruction.**

   Remember from previous discussions that the =32-bit value part of this instruction is a directive to the assembler to, in some way, create an instruction that loads the specified address or other constant into the particular register. The assembler first attempts to do this with a single Move (MOV Rn) or a Move Negative (MVN Rn) instruction. If the assembler cannot create a MOV or MVN instruction that loads the desired constant, it puts the value in a literal pool and creates an LDR instruction that loads the 32-bit value from the literal table when it executes. The problems with this approach are that the literal pool takes up space in the code section of the program and, if the literal pool is not in the cache when the LDR instruction executes, it will take perhaps 100 clock cycles for the LDR instruction to get the value from the literal pool. The obvious conclusion from this is that to optimize performance, you would like to reduce the use of a literal pool by your programs.

   The way we approach this is to use the LDR RN,=0x3FFFFFFF type instructions in the first version of a program as shown in Figure 4-12. One advantage of using the LDR RN,= 0x3FFFFFFF type instructions in the first version of a program is that they show the address and constant loads clearly. Another advantage is that the assembler will automatically translate many of these into equivalent MOV or MVN instructions. (As we show a little later, the assembler will automatically do this for any LDR instructions that can be implemented with a single MOV or MVN instruction.) We then look at the debugger output file for the program. The debugger disassembles the executable file and shows the mnemonics for the actual instructions created by the assembler, as compared with the assembler list file that simply shows the instruction mnemonics as you typed them. We then replace each LDR instruction that refers to the literal pool with two or three simple instructions that load the desired values in the registers without using the literal pool. The best way to show how you do this is with some familiar examples.

   To load a value such as 0x40E00000 into R1, for example, you can first tell the assembler to load 0x00E00000 into R1 with a MOV instruction. This would seem impossible to accomplish with a simple MOV instruction because the immediate operand allowed in any arithmetic instruction can only be 8 bits. However, an arithmetic instruction such as a MOV can specify that the 8-bit operand be rotated an even number

of bit positions. The assembler will code the MOV R1, #0x00E00000 instruction with an immediate value of 0xE0 and a rotate right code of 8 that tells the processor to rotate the 0xE0 value 16 bits right before loading it in R1. (The number of bits rotated is equal to two times the value coded into the rotate field of the instruction, so that only 4 bits are needed to represent up to a 30-bit rotation.) You then use the instruction ORR R1, R1,#0x40000000 to put the rest of the required bits in R1. The assembler will code the ORR R1, R1, 0x40000000 instruction with 0x40 in the immediate field and a rotate right code of 4. This instruction then tells the processor to rotate the immediate value 0x40 right by 8 bit positions. The rotated value of 0x4000000 ORed with the 0x00E00000 gives the desired result of 0x40E00000 in R1. The trick here is to build a desired 32-bit result by using rotated 8-bit immediate values and logic operations to get the desired result.

In cases where only a single MOV or MVN instruction is needed, the assembler can figure out how to do this for you. As an example of this, the assembler automatically can automatically translate the instruction LDR R3, = 0x03FFFFFF instruction to a MVN R3, #0xFC000000 instruction. The MVN instruction puts the complement of the given immediate operand in the specified register. The operand 0xFC000000 is obviously too large to fit in the 8-bit immediate field of the instruction but a 0xFC operand fits. The assembler will code this as the immediate value and specify a Rotate of 8 bit positions. When the instruction executes, the constructed value of 0xFC000000 will be complemented to give the desired result of 0x03FFFFFF.

As an example of another technique you can use to load a particular type of 32-bit constant into a register, suppose you want to load the 32-bit value 0x0001FC00 in R3. You can use a MVN R3,0x00 instruction to load R3 with 0xFFFFFFFF. You can then use a MOV R3,R3,LSL #15 instruction to Logically Shift this value Left 15 bits to put zeros in the lower 15 bits and then use an MOV R3,R3,LSR #15 instruction to shift the value back into position and leave zeros in the upper 15 bits that have to be zero as desired. This may not be immediately obvious but once you have perhaps made a game of it and done it a few times it becomes almost automatic.

As a final example of these techniques, you can build a value of 0x3FF in R4 by first executing a MOV R4,#0xFF instruction to load 0xFF in R4, and then use and ADD R4,R4, #0x300 to get the desired result. Again, an operand of 0x300 is too large to fit directly in the immediate field of an instruction, but the assembler codes an immediate value that will produce the same result when rotated and added to the 0xFF in R4.

In summary, you can load constants into registers using at most four simple instructions but usually one or two will do the job. Once you understand the principle, it is fun to figure out the simple instructions that will load a desired operand in a register without reading the value from a literal pool. Other than the intellectual satisfaction of doing it successfully, the main reward is a program that executes faster because it does not need to access a literal pool to load 32-bit values into registers. Of course, if the particular application does not need to execute quickly, you can just leave your program with the initial LDR RN, =constant instructions to reduce the overall program development time and get the product to market more quickly.

2. **When setting up pointers to several GPIO registers, initialize one register with the base address for all of the GPIO registers and then create the other pointers by simply adding the offsets of these registers to the base address.**

This technique is well demonstrated in the program in Figure 4-16. In this program, for example, we first initialized R0 to point to 0x4804C000, the base address for all the GPIO1 control registers with the LDR R0,=0x4804C000 instruction. Then we initialized R2 to point at GPIO1_DATAIN register with the simple ADD R2,R0,#0x138 instruction and initialized R4 to point at GPIO1_CLEARDATAOUT with the simple ADD R4,R0,#0x190 instruction. For the cases where the offsets of the desired GPIO ports fall within an 8-bit range of the base address, this add-to-base technique avoids building each 32-bit pointer in a register using an entry in the literal pool.

3. **Have loops count down to zero instead of up to some number, and add the 'S' suffix to data processing instructions to set flags instead of using separate compares to set flags.**

There are many examples of this in the programs in Chapters 3 and 4. The array program in Figure 4-6 used the SUBS R4,R4,#1 instruction to decrement the element counter in R4 and update the flags. Using the S suffix to set the flags eliminates the need for a CMP R4,#0 to update the flags after the SUB instruction. Note that for the same reason, when writing C programs you should also have loops count down to zero instead of having them count up to some number, as shown by the example in Figure 3-8.

4. **Use Auto-Increment and Auto-Decrement addressing modes.**

In the multiplication example in Figure 3-6 we used separate ADD instructions to increment the pointers after the operands were read from memory to make these operations very clear. However, as explained in the instruction examples in Chapter 3 and as shown in the example program in Figure 4-6, the ARM Load and Store instructions have auto-increment and auto-decrement capability. The instruction LDRB R6,[R1],#1 loads a memory byte pointed to by R1 into R6 and increments the pointer in R1 by 1 to point to the next byte in the array. The obvious advantage of these addressing modes is that they eliminate the need for separate pointer increment or decrement instructions and help reduce code bloat.

5. **Use LSL to multiply by a power of 2 and LSR to divide by power of 2.**

Instead of using the ARM MUL instruction, you can multiply a number in a register by a power of 2 faster by simply by Logically Shifting the number Left a number of bits equal to the power. For example, since 8 is 2^3, you can multiply the value in R2 by 8 with the instruction MOV R2,R2,LSL #3. Likewise, you could divide the number in R2 by 4 with the instruction MOV R2,R2,LSR #2. For the cases where it works, this method of multiplying is faster than using the MUL instruction. Several of the ARM-based processors do not have a DIV instruction, but this method does work well for dividing by a power of 2. For cases where this does not work, you can do a required division by successive subtraction, with multiplication by the reciprocal of the divisor, or by calling a C library function.

6. **Use the unique-to-ARM BIC instruction to clear a bit or up to 8 bits anywhere in a 32-bit word instead of creating a mask and then ANDing the mask with the word.**
The BIC instruction is a data processing instruction that ANDs the complement of a specified operand with the contents of a register. The BIC R1,R1,#04 instruction, for example, will AND the contents of R1 with the complement of the immediate value 4, or 11111111111111111111111111111011. This operation will clear bit 2 in R1 and leave the other bits unchanged. Likewise, the instruction BIC R1,R1,#0x0000FF00 will clear bits 8-15 of the value in R1. Note that up to 8 adjacent bits can be specified to be cleared and that the bits to be cleared can be anywhere in the 32-bit register. However, as with the examples in #1 in this list, you must construct the immediate number so that the assembler can access the desired bits by rotating the immediate number an even number of bit positions.

REDUCING RAW HAZARDS

As discussed in chapter 2, pipelined processors such as the ARM-based processors will suffer stall cycles if the processor needs to wait to Read an operand until After it has been Written by some other operation. Waiting for an operand in this way is commonly called a Read-After-Write or RAW hazard. RAW hazards can be caused by an instruction taking more than one clock cycle to produce an operand. RAW hazards are also commonly caused by an instruction or data word not being available in a cache when needed. You can eliminate or at least reduce the clock cycles lost to RAW stall cycles by trying to make sure operands are available as close to when they are needed as possible. Here are a few general rules to help with this.

1. **To reduce data processing instruction data hazards, put non-dependent instructions immediately after the instruction that produces a needed operand, and schedule dependent instructions later.**

Most ARM data processing instructions have a result latency of one clock cycle. This means that an operand produced by an instruction can be used in the following instruction with no stall cycles. However, for data processing instructions, if the result of a previous operation is used as an operand for a Shift or Rotate by immediate operation, there will be a result latency of two clock cycles. As an example of this, the MOV instruction in the following instruction sequence would suffer a one-cycle stall because the MOV instruction uses the value of R1 produced by the ADD R1,R2,R3 instruction.

```
SUB R4, R6, R7
ADD R1, R2, R3
MOV R6, R1, LSR #4
```

This stall can be eliminated by scheduling the non-dependent SUB R4, R6, R7 instruction after the ADD R1, R2, R3 instruction instead of before as follows:

```
ADD R1, R2, R3
SUB R4, R6, R7
MOV R6, R1, LSR #4
```

As another example of this, the MUL instruction has a result latency of 1-3 clock cycles, depending on the value of the operands. Therefore, you should schedule an instruction that uses the result of a MUL instruction at least two instruction slots after the MUL instruction to avoid stalls.

For the latencies of other instructions, consult the TI AM3359 Data book.

Keep data items close together, so they will be read into a cache line with a single operation after the first miss.

An L1 data cache line and an L2 cache line in the Arm Cortex-A8 processors are 64 bytes. In other words, when the processor does a cache line fill, it reads a block of 64 bytes from memory. To make this most efficient, you should start data arrays or structures on 64-byte boundaries. You do this by using the .align directive. The directive .align 6 will start the next data item on a 64-byte boundary.

2. **Use multiple register Load and Store instructions.**
The Load Double instruction, as for example, LDRD R0, [R2] loads the word pointed to in memory by R2 into R0, and the word in memory pointed to by R2 + 4 into R1. The Store Double instruction, as for example, STRD R0, [R2] instruction stores the word in R0 at the memory address pointed to by R2, and stores the word in R1 at a memory address calculated by adding 4 to the value contained in R2. In a later section of this chapter we show you how to use the Load Multiple (LDM) instruction that can be used to load a specified group of registers from a series of memory locations, and the Store Multiple (STM) instruction that can be used to store a specified group of registers in a series of memory locations.

REDUCING BRANCH STALLS

As explained in chapter 2, a taken branch causes stalls in a pipelined processor because the fall-through path instructions that were automatically loaded into the pipeline need to be flushed and new instructions fetched from the branch target address. For branches that are encountered and taken over and over, Branch Target Buffers (BTBs) and branch prediction bits can significantly reduce the number of branch stalls. However, for branches that are only encountered once in a program, BTBs and prediction bits do not help. To reduce stalls caused by these single-encountered branches, you can in many cases use the conditional instruction execution capability of the ARM-based processors. As we explained in chapter 3, for Branch instructions and for almost all of the other ARM instructions, you can specify whether the processor should execute that instruction or not, based on some condition represented by the flags. Table 3-3 shows the conditions that can be specified for conditional execution. The fact that you can attach conditions to the execution of data processing instructions often allows you to eliminate branch instructions. As a familiar example of this, suppose that our PC board machine had only 2 LEDs, yellow and green. We will use the IF-THEN-ELSE decision section of the LED example program in Figure 4-12 to show you how to do this. To refresh your memory, the implementation of the IF-THEN-ELSE section of the program with branches is as follows:


```
                CMP R0, #30          @ Compare temperature with setpoint
                BMI YELLOW           @ If temperature < 30 go light yellow
                LDR R5,=0x80000000 @ Else load mask for green LED
                B    SEND            @ Go write word to GPCR0 register
YELLOW:         LDR R5,=0x20000000 @ Load mask for yellow LED
SEND:           STR R5, [R3]         @ Write LED turn on word to GPCR2
```

Using the conditional execution capability of the ARM processor, the instruction sequence can be written with no branch instructions as follows:

```
CMP R0, #30                 @ Compare temperature with setpoint
MOVMI R5,#0x200000000  @ If temperature < 30 load mask for
                            @  yellow LED
MOVPL R5,#0x80000000   @ Else load mask for green LED
STR R5, [R3]                @ Write LED turn on word to GPCR2
```

As you can see, using the two conditional MOV instructions eliminates the BMI instruction and the unconditional Branch instruction. The MOVMI will put 0x20000000 in R5 only if the compare that is done by subtracting 30 from the value in R0 produces a negative (Minus) result. Likewise, the MOVPL instruction in this second version will only load the value of 0x80000000 in R5 if the value in R0 is equal to or greater than 30.

This second implementation of the IF-THEN-ELSE structure does not contain any branch instructions, so it will not suffer any branch stalls. However, the instructions for the IF path and the instructions for the ELSE path are both read in from memory and clocked through the pipeline. Even though the instructions for one path are not executed, it will still take some clock cycles to shift these instructions through the processor pipeline. Since no useful work is done by these non-executed instructions, they represent wasted clock cycles. In this example, there will be only one non-executed instruction for either path through the IF-THEN-ELSE structure, so there will only be one wasted clock cycle. However, for cases where the IF path and the ELSE path are each longer than four or five instructions, it may be more efficient to implement an IF-THEN-ELSE with branches and pay the branch stall penalties than to have the processor step through long sequences of non-executed conditional instructions.

Writing and Calling Procedures in Arm Assembly Language Programs

INTRODUCTION AND TERMINOLOGY

Often when writing programs you find that you need to use a particular sequence of instructions many times in the program. Rather than repeating this sequence of instructions *in-line* each time they are needed, you can write the sequence of instructions as a procedure and simply "call" the procedure each time the sequence of instructions is needed. Figure 4-20a shows how this works. A branch instruction in the mainline program, or in some cases an interrupt signal or exception condition, "calls" the procedure. After the procedure instructions execute, an instruction at the end of the procedure returns execution to the mainline program at the next instruction after the branch that called the procedure. As shown in Figure 4-20b, procedures can be "nested," which means that one procedure calls another procedure as part of its instruction

sequence. You can follow the arrows in the figure to see the overall program flow for these nested procedures.

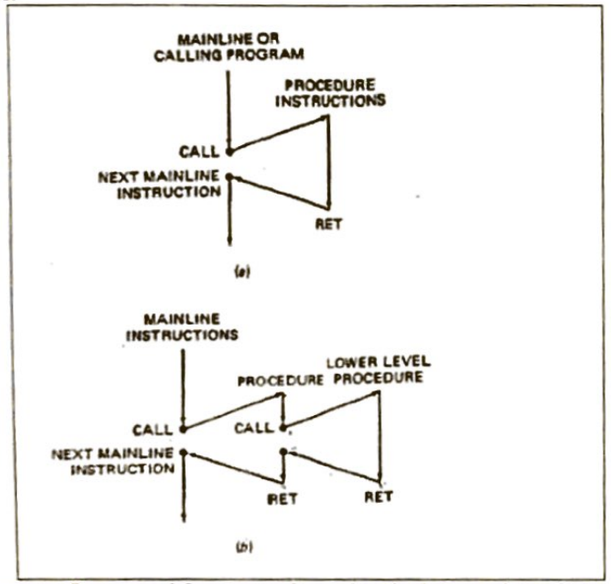


Figure 4-20. Program flow to and from procedures. (a) Single procedure. (b) Nested procedures.

Up to this point, we have been using the term "procedure" to identify the "subprograms" that you write to avoid duplicating a sequence of instructions in your mainline program. Other terms commonly used to identify these subprograms are *functions* and *subroutines*. For this book we will follow the conventions most commonly used in industry documents. For the assembly language sections of this book we will use the term 'procedure'. For the C programming language examples in the book, we will use the term 'function' when referring to an expression of the form z = sum (a, b), where a value is returned directly to a named variable such as the z in this function call. We will use the term procedure for an expression of the form printf("Hello Universe\n");

In addition to reducing the overall number of instructions and therefore the memory requirement in your programs, procedures have another very important role in the top-down design programming approach described in chapter 3. Remember that the top-down design approach involves defining the problem and then breaking the overall programming task into a hierarchy of smaller and smaller modules until the operation of each module is very clear. As an example, Figure 4-21 shows a hierarchical chart for the modules of a simple inventory control program. The modules are usually written as procedures that are called from the mainline program or from higher-level modules. The advantages of this approach are that: it is easier to develop, test, and debug each module individually than to do so with a single large program; debugged procedures can be stored in a library and re-used in other programs; and a person learning the operation of the program can start with the mainline program and work down through the levels of the hierarchy until he/she achieves the required level of understanding. In

the next section, we show you how to Call and Return from ARM assembly language procedures.

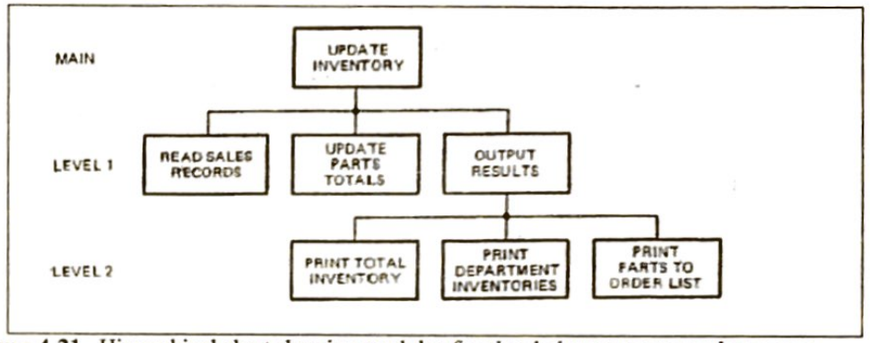


Figure 4-21. Hierarchical chart showing modules for simple inventory control program.

Figure 4-22 shows a very simple example of a procedure Call and Return in ARM assembly language. After the two MOV instructions in the mainline program load some test values into R0 and R1, the BL DOADD instruction "calls" the DOADD procedure by simply doing an unconditional branch to the procedure. The Branch and Link instruction, BL, also stores the address of the MOV R0,# 0x18 instruction, which is the next instruction after BL, in R14. Register R14 is also known as the *Link Register* or *LR*, because it holds the "link" back to the calling program.

The DOADD procedure adds the two operands passed to it in R0 and R1, and puts the result in R0 to pass back to the calling program. The MOV PC, R14 instruction then returns execution to the calling or mainline program. The MOV PC, R14 instruction simply copies the return address saved in R14 by the BL instruction into the Program Counter (PC). The PC will then use this address to fetch the next instruction. In this case, the next instruction fetched will be the MOV R0, #0x18 instruction in the mainline. Note that since LR is another name for R14, you can write the MOV PC, R14 instruction as MOV PC, LR.

CALLING AND RETURNING FROM ARM ASSEMBLY LANGUAGE PROCEDURES

```
                                                    subrout.s

@ ARM ASSEMBLY LANGUAGE PROGRAM TO ILLUSTRATE SIMPLE
@       PROCEDURE CALL AND RETURN
@ Copyright Douglas V. Hall Portland, OR

.text
.global _start:
_start:     MOV R0, #10         @ Load values to process
            MOV R1, #3
            BL  DOADD           @ Call subroutine, return addres in R14
            NOP                 @ Instruction to use for breakpoint

DOADD:  ADD R0,R0,R1            @ Add operands passed in r0 and r1
                                @ Return result in r0
            MOV PC, R14         @ Return to calling program using return
                                @     address in R14, the Link Register
.end
```

Figure 4-22. ARM assembly language example to show simple procedure call and return.

The simple example in Figure 4-22 shows the basic principles of calling and returning from a procedure, but for more complex procedures and interrupt procedures the two major additional points that you need to know about are:

1. How to save the registers and flags or *state* of the calling program and restore the state when execution returns to the calling program.
2. How to pass values (parameters) from the calling program to the procedure and how to pass values from the procedure back to the calling program.

We will start by talking about how you save the state of the calling program. Most microprocessors allow a programmer to set aside a special section of memory called a *stack*. The stack is used to save copies of the mainline program registers and flags while a procedure executes, so they can be restored after the procedure executes.

Figure 4-23 shows in diagram form how this works. For the flow shown in Figure 4-23a the calling program copies or, as we commonly say, "pushes" any registers used in the procedure on the stack and then calls the procedure. After execution returns to the calling program, calling program instructions "pop" or, in other words, copy the saved register values and flags from the stack back into the specified registers. Pushing and popping the registers in the calling program is commonly referred to as "caller saved state."

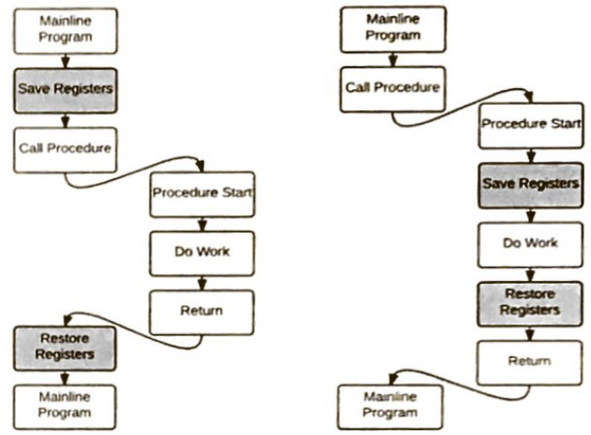


Figure 4-23. Program flow for procedure call showing register saves. (a) Pushed in calling program. (b) Pushed in procedure. (Improved artwork supplied by Eric Krause.)

In the second approach, shown in Figure 4-23b, the registers and flags are pushed on the stack at the start of the procedure and they are popped off the stack at the end of the procedure. Pushing and popping the registers in the procedure is commonly called "callee saved state." Both approaches do the job but one advantage of putting the push and pop instructions in the procedure is that you only need to write them in the program once rather than in the program each time you call the procedure. Another advantage is that, if the procedure is put into a library for re-use, you don't have to study the procedure to find out which registers to save before you call the procedure because this is already taken care of in the procedure. The disadvantage of callee saved state is that in some cases, you might not care if the calling program state is saved when you call the procedure. In these cases, the push and pop operations in the procedure are not needed and therefore add unnecessary execution time. The point, as always, is that for each individual case you must evaluate the tradeoffs and chose the state-saving approach that works best for that specific case. Now let's look at the different ways you can implement and use a stack in an ARM-based microprocessor system.

IMPLEMENTING AND USING A STACK IN ARM ASSEMBLY LANGUAGE PROGRAMS

Figure 4-24 shows how the familiar halfword multiplication example from Figure 3-6 can be rewritten so that the actual multiplications of the array elements are done in a procedure instead of in the mainline program. First let's look at how the stack is set up and accessed.

A stack is managed as a simple array of words in a memory space assigned to the stack. In the program in Figure 4-24, the STACK: .rept 256, the .byte 0x00, and the .endr statements tell the assembler to set aside 256 bytes of memory to be used as a stack and initialize all of the locations in this range with zeroes.

To access the stack in a program, a register referred to as the *Stack Pointer* (SP) register is used to hold the address of the location currently being accessed in the stack. This location is commonly called the top-of-the-stack (TOS). In ARM-based processors, R13 functions as the SP, so the assembly language statement LDR R13,=STACK at the start of the mainline in Figure 4-22 initializes R13 with the starting address of the memory space set aside for the stack. However, for compatibility with the memory usage by high-level language programs, we would like to write words to the stack, starting from the top of the assigned memory space, rather than from the bottom. Therefore, we move the SP to the top of the memory space assigned to the stack with the ADD R13, R13, #0x100 instruction. (Actually, if you are checking our arithmetic here, you would note that this instruction initializes the SP to an address that is one byte above the space set aside for the stack. However, as we will soon show, this is not an error.)

multoproc.s

```
@ ARM example program to show stack frame setup,
@ pushing and popping registers, and passing parameters
@  to a procedure in registers
@ Copyright  Douglas V  Hall Portland, OR

.text
.global _start
_start:
.equ      NUM, 4
          LDR     R13,=STACK        @ Point stack pointer to low end of  stack space
          ADD     R13, R13, #0x100  @ Point stack pointer at top of stack
          LDR     R0,=MULTIPLICANDS  @ Load array addresses (pointers) into registers
          LDR     R1,=MULTIPLIERS @   to pass to procedure
          LDR     R2,=PRODUCTS
          MOV     R3, #NUM@ Load counter to pass to procedure
          BL      MULTO             @ Call procedure to do the work

MULTO:    STMFD   R13!, {R6-R8, R14}  @ Save used registers on stack
NEXT:     LDRH    R6, [R0], #2      @ Get a multiplicand half word, increment
          LDRH    R7, [R1], #2      @ Get a multiplier half word, increment
          MUL     R8,R6,R7          @ Perform multiplication
          STR     R8, [R2], #4      @ Copy result word to products, increment
          SUBS    R3, R3, #1        @ Decrement element counter
          BNE     NEXT              @ Repeat until all 4 elements done
          LDMFD   R13!, {R6-R8,PC}  @ Restore registers and return

.data
MULTIPLICANDS:  .hword  0x1111, 0x2222, 0x3333, 0x4444
MULTIPLIERS:    .hword  0x1111, 0x2222, 0x3333, 0x4444
PRODUCTS:       .word   0x0, 0x0, 0x0, 0x0
STACK:          .rept 256         @ Reserve 256 bytes for stack and init with 0x00
                .byte 0x00
                .endr
.end
```

Figure 4-24. ARM assembly language program to show stack frame setup, pushing and popping registers, and passing parameters to a procedure in registers.

After the stack pointer is initialized, the next steps in the program in Figure 4-24 are to load the starting addresses of the MULTIPLICANDS, MULTIPLIERS, and PRODUCTS arrays in registers R0, R1, and R2, respectively. Since these pointers to the three arrays will be in the three registers when we call the MULTO procedure, they will be "passed" to the procedure in the registers. The procedure can then access the arrays using these registers as pointers. Note that we also load a single value, the number of array elements, in R3 with the statement MOV R3, #NUM to pass to the procedure. These examples show two of the possible ways to pass values (parameters) to procedures. You can pass values directly in registers, or for data structures such as arrays you can pass pointers to these structures in registers.

The BL MULTO instruction in Figure 4-24 unconditionally branches to the procedure MULTO and saves the return address in the Link Register, R14. All but the first instruction and the last instruction in the MULTO procedure are the same as in the example in Figure 3-6, so they do not require further explanation except to say that in the procedure these instructions use the pointers and the count value passed to them in registers R0-R3. The STMFD instruction at the start of the procedure and the LDMFD instruction at the end of the procedure require considerable explanation. To start, let's look at the basic LDM and STM instructions.

Contrary to the design philosophy of a pure RISC machine, the ARM-based processors have a Store Multiple (STM) instruction that can be used to store multiple registers in a sequence of memory locations, starting at an address specified in a register. They also have a Load Multiple (LDM) instruction that can be used to load a specified set of registers with values from a sequence of memory locations. The four ways that an STM instruction can be directed to store a sequence of registers in memory or specifically on the stack are as follows:

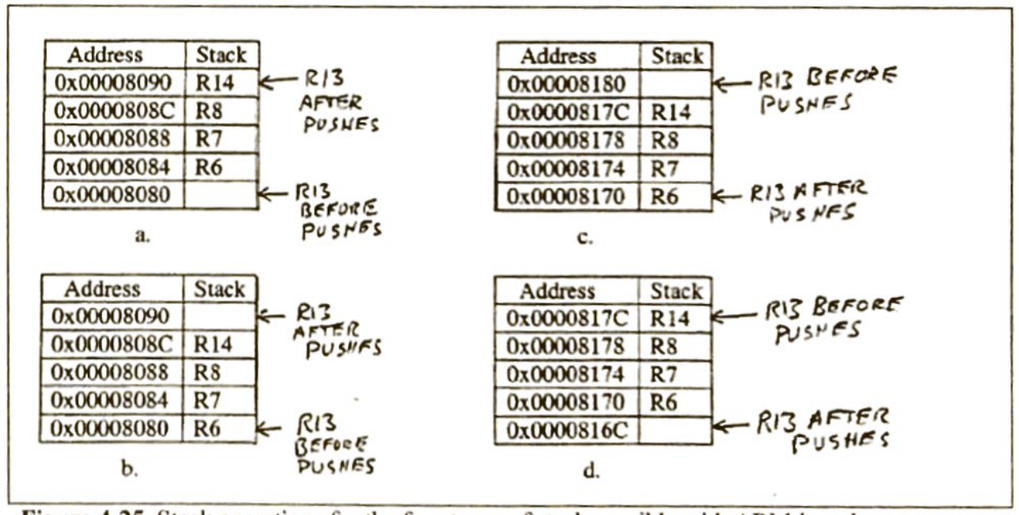


Figure 4-25. Stack operations for the four types of stack possible with ARM-based processors. (a) Full Ascending or Increment before push. (b) Empty Ascending or Increment after push. (c) Full Descending or Decrement before push. (d) Empty Descending or Decrement after push.

Full Ascending or Increment Before – STMFA R13!, {R6-R8, R14} or STMIB R13!, {R6-R8, R14}. The value in R13 is incremented by 4 and R6 is written to the location pointed to by R13.

The other registers are stored in numerical order at increasing addresses. Register R13 is left pointing to the "full' location containing a copy of R14. Traditionally this is known as an "increment and push" type system. Assuming R13 contains 0x00008080, Figure 4-25a shows in diagram form the contents of the stack and the stack pointer after execution of the STMFA R13!, {R6-R8, R14} instruction.

Empty Ascending or Increment After – STMEA R13!, {R6-R8, R14} or STMIA R13!, {R6-R8, R14}. Register R6 is written to the location pointed to by R13. The other registers are stored in numerical order at increasing addresses. After R14 is copied to the stack, register R13 is incremented by 4 and therefore is left pointing to an "empty" location at an address above the address containing a copy of R14. Traditionally this is known as a "push and increment" type system. Assuming R13 contains 0x00008080, Figure 4-25b shows the contents of the stack and the stack pointer after execution of the STMFA R13!, {R6-R8, R14} instruction.

Full Descending or Decrement Before – STMFD R13!, {R6-R8, R14} or STMDB R13!, {R6-R8, R14}. Register R13 is decremented by 4 and the highest numbered register in the list, R14 in this example, is written at that address. The other registers are written in decreasing numerical order at decreasing addresses. Register R13 is left pointing at the "full" location containing the copy of R6. Traditionally this is known as a "decrement and push" type system. As a side note, the Intel 80x86 family processors use a decrement and push type stack. Figure 4-25c shows the contents of the stack and the stack pointer after execution of the STMFA R13!, {R6-R8, R14} instruction, assuming R13 contains 0x00008180 after adding 0x100 to a stack base address of 0x00008080.

Empty Descending or Decrement After – STMED R13! {R6-R8, R14} or STMDA R13! {R6-R8}. The highest numbered register in the list, R14 in this example, is written at the address contained in R13. The other registers are written in decreasing numerical order at decreasing addresses. After the registers are copied to the stack, R13 is decremented by 4 and left pointing at the "empty" next word address below the location containing the copy of R6. Traditionally this is known as a "push and decrement" type system. Again as a side note, the Motorola MC68K family processors use a push and decrement type stack. Assuming R13 contains 0x000080FC after adding 0xFC to a stack base address of 0x00008080, Figure 4-23d shows the contents of the stack and stack pointer after execution of the STMFA R13!, {R6-R8, R14} instruction.

To implement a stack for saving registers during procedure calls, you will usually implement it as Full Descending or Empty Descending. We chose to use Full Descending to be compatible with other ARM systems, with Intel systems, and with the stack implemented by most C compilers. Since a Full Descending system decrements the specified stack pointer before copying a register to the stack, you can at the start of your program initialize the stack pointer to the next address above the actual stack. In our example program in Figure 4-24, we did this by adding 256 to the starting address of the stack in R13 instead of adding 0xFC to R13.

The STMFD R13!, {R6-R8, R14} instruction at the start of the MULTO procedure in Figure 4-24 pushes registers R6, R7, R8 and R14 on the stack and leaves R13 pointing at the copy of R6. The LDMFD R13!, {R6-R8, PC} instruction at the end of the MULTO procedure copies the saved values of R6, R7, and R8 from the stack back to registers R6, R7, and R8. It also copies

the value of R14 saved on the stack into the Program Counter instead of back into R14. Since the value copied to the stack from R14 was the return address put there by the BL instruction, loading this value into the program counter returns execution to the next address after the BL instruction in the calling program. The single LDMFD R13!, {R6-R8, PC} instruction then both restores the registers and returns execution to the calling program.

For a procedure call and return to work correctly, it is very important that the stack pointer is pointed to the correct location in the stack when you do a return instruction such as LDMFD R13!, {R6-R8, PC} that pops the return address off the stack into the PC. If you do not do this, the return instruction will pop an incorrect value into the PC and your program will likely "crash" the system. In simple procedures, you can keep the stack "balanced" so that the return works correctly by just keeping the number of registers popped equal to the number of registers pushed. In more complex procedures, where you use the stack to pass parameters, you may directly load the stack pointer with the value needed to pop the return address off the stack into the PC as we show a little later. In all cases, you use a stack map such as those in Figure 4-25 to keep track of where everything is in the stack and where the stack pointer should point in order to correctly return to the calling program. Incidentally, in chapter 5 we show you how to preserve the flags of the calling program for those cases where you need to do this. Next here, we need to discuss in more detail how you pass parameters to and from procedures.

PASSING PARAMETERS TO AND FROM PROCEDURES

In the example program in Figure 4-22 we showed how you use registers for passing single values of parameters to a procedure, and in Figure 4-24 we showed how you can also pass array pointers to a procedure. For compatibility with high-level language compilers, ARM Ltd. has defined a standard called the ARM Procedure Call Standard or APCS for the use of registers and other features in procedure calls. In order to ensure that your assembly language procedures will be callable from high-level language programs, the key APCS points you need to know and follow for your assembly language programs are:

1. **Registers R0-R3 are used to pass parameters or pointers to a procedure.**
2. **R0 is used to pass an integer result back to the calling program.**
3. **Registers R4-R10 must be saved by the procedure, if they are used in the procedure.**
4. **R11 and R12 are "scratch pad" registers and do not need to be preserved.**
5. **R13 is the Stack Pointer (SP).**
6. **R14 is the Link Register (LR).**

Note that in the program in Figure 4-22 we passed parameters to the procedure in R0 and R1, and passed the result back to the mainline program in R0 as specified by the APCS. Likewise, in the program in Figure 4-24 we passed four parameters to the procedure in R0-R3 as specified by the APCS. Now, suppose that you are writing a program and you find that you need to pass more single values or more pointers than will fit in the four registers (R0-R3) allowed by the APCS. The alternative to using registers for passing single parameters or pointers is to use the stack. If you need to pass more parameters than can be passed in R0-R3 you can, for example, load 4 pointers or other parameters into registers R0-R3 and then use an STMFD R13!, {R0-R3} instruction to push these parameters on the stack. In the procedure you can pop the passed parameters off the stack and use them as needed. To help you understand how to do this, Figure

4-26 shows an extension of the array multiplication program in Figure 4-24. Specifically, this program passes two sets of parameters to the procedure, so that one call of the procedure will process two sets of arrays instead of just one. (Note that at the risk of boring you we used an extension of the multiply program for this example, because this minimizes the amount of new material and you can then concentrate just on how you work with the stack.)

In the program, we first load the parameters for one set of arrays into registers R0-R3 and then push these registers on the stack with an STMFD R13!, {R0 – R3} instruction. The stack map in Figure 4-27 shows the location of these parameters in the stack, assuming that R13 contains 0x000081E8 before they were pushed. Note that since four parameters were pushed, R13 is left pointing at 0x000081E8 – 4 x 4 or 0x000081D8.

Next in the program in Figure 4-26, we load the parameters for the second set of arrays into R0-R3 and call the MULTO procedure with the BL MULTO instruction. In the procedure we save the values in registers R6-R8 and R14 on the stack with the STMFD R13!, {R6-R8, R14} instruction. The stack map in Figure 4-27 shows the contents of the stack and the value in R13 after this instruction executes. Again, we pushed four registers on the stack, so the value in R13, the stack pointer, will be
0x000081D8 – 0x10 or 000081C8, after this instruction executes. We have found that a stack map such as that in Figure 4-27 is essential for keeping track of where all the values are located on the stack and the value in the stack pointer (R13) at any particular time as you do push and pop operations.

multodbl.s

```
@ ARM example program to show using registers and stack to pass
@ parameters to a procedure. Procedure multiplies corresponding elements in
@ two sets of arrays
@ Copyright by Douglas V. Hall Portland, OR

.text
.global _start
_start:
.equ        NUM, 4
            LDR     R13,=STACK          @ Point stack pointer to bottom of stack space
            ADD     R13, R13, #0x100    @ Point stack pointer at top of stack
            LDR     R0,=MULTIPLICANDS2 @ Load pointers to second set of data arrays
            LDR     R1,=MULTIPLIERS2    @ in R0-R2
            LDR     R2,=PRODUCTS2
            MOV     R3, #NUM@ Load first counter in R3
            STMFD   R13!, {R0-R3}       @ Push First set of Parameters on stack
            LDR     R0,=MULTIPLICANDS1 @ Load pointers to first set of data arrays
            LDR     R1,=MULTIPLIERS1    @  in R0-R2
            LDR     R2,=PRODUCTS1
            MOV     R3, #NUM@ Load second counter in R3
            BL      MULTO               @ Call Procedure to do the work.
            NOP
MULTO:      STMFD   R13!, {R6-R8, R14}  @ Save used registers and Return address on stack
NEXT1:      LDRH    R6, [R0], #2        @ Get a multiplicand half word from array 1
            LDRH    R7, [R1], #2        @ Get a multiplier half word from array 1
            MUL     R8,R6,R7            @ Perform multiplication
            STR     R8, [R2], #4        @ Pop  result word to products array 1
            SUBS    R3, R3, #1          @ Decrement element counter
            BNE     NEXT1               @ Repeat until all 4 elements done
            ADD     R13, R13, #16       @ Add 16 to point R13 at parameters pushed on stack
            LDMFD R13!, {R0-R3}         @ Pop pushed values off stack into registers R0-R3
NEXT2:      LDRH    R6, [R0], #2        @ Get a multiplicand half word from array 2
            LDRH    R7, [R1], #2        @ Get a multiplier half word from array 2
            MUL     R8,R6,R7            @ Perform multiplication
            STR     R8, [R2], #4        @ Copy result word to products array 2
            SUBS    R3, R3, #1          @ Decrement element counter
            BNE     NEXT2               @ Repeat until all 4 elements done
            SUB     R13, R13, #32       @ Subtract 32 from R13 so it points at registers saved on
                                        @ Stack at start of procedure
            LDMFD R13!, {R6-R8, R14}    @ Pop pushed values off stack into registers R0-R3, R14
            ADD R13, R13, #16           @ Add 16 to R13 to put stack pointer back at
                                        @     original position ( balance stack)
            MOV PC, R14                 @ Return to mainline program
.data
MULTIPLICANDS1: .hword  0x1111, 0x2222, 0x3333, 0x4444
MULTIPLIERS1:   .hword 0x1111, 0x2222, 0x3333, 0x4444
PRODUCTS1:      .word  0x0, 0x0, 0x0, 0x0
MULTIPLICANDS2: .hword  0x5555, 0x6666, 0x7777, 0x8888
MULTIPLIERS2:   .hword 0x5555, 0x6666, 0x7777, 0x8888
PRODUCTS2:      .hword 0x0, 0x0, 0x0, 0x0
STACK:          .rept 2560              @ Reserve 256 bytes for stack and init with 0x00
                .byte 0x00
                .endr
.end
```

Figure 4-26. ARM assembly language program to show use of registers and the stack to pass parameters to a procedure.

Address	Stack	Comments
0x000081E8		<- R13 Before
0x000081E4	R3	Stack after
0x000081E0	R2	STMFD R13!,
0x000081DC	R1	{R0-R3}
0x000081D8	R0	<- R13 After
0x000081D4	R14	<-R13 before
0x000081D0	R8	STMFD R13!,
0x000081CC	R7	{R6-R8,R14}
0x000081C8	R6	<- R13 After
0x000081C4		

Figure 4-27. Stack map showing contents of stack and stack pointer during execution of program in Figure 4-26.

The NEXT1 block of instructions in the procedure uses the parameters passed in R0-R3 to multiply the elements in the Multiplicands1 and Multipliers1 arrays and put the results in the Products1 array. However, to access the parameters needed to process the second set of arrays, we need to pop them off the stack. To do this, we need to do a little stack pointer arithmetic with the help of the stack map in Figure 4-27. After pushing the registers at the start of the procedure, R13 was left pointing at address 0x000081C8. To get R13 pointing to address 0x000081D8, where the R0-R3 registers we passed on the stack are located, we need to add 16 to R13 with the ADD R13, R13, #16 instruction. With R13 pointing to these parameters, we can pop them off the stack into registers R0-R3 with the LDMFD R13!, {R0-R3} instruction and use them to process the second array with the instructions in the NEXT2 loop.

Note that, after registers R0-R3 are popped off the stack, R13 will be pointing to address 0x000081E8, the starting location. Before we can restore registers R6-R8 and R14, we have to move R13 back down to address 0x000081C8 where these registers were pushed on the stack. To do this, we simply subtract 32 from R13 with the SUB R13, R13, #32 instruction and use the LDMFD R13!, {R6-R8, R14} instruction to restore the initial contents of R6-R8 and R14 to get ready to return to the calling program. Before returning however, we have to *balance the stack.* The term *balance the stack* means to put the stack pointer register back to its initial value, so that it is not left pointing somewhere down in the stack space. If you don't do this in each procedure, the stack pointer may keep moving down through the stack space as you call procedures and eventually overflow down into your program space in memory. If this overflow happens, writing to the stack may write over data or even your program code and likely "crash" your program.

For the example program in Figure 4-24, R13 will contain 0x000081D8 after the LDMFD R13!, {R6-R8, R14} instruction executes. You want it to contain the initial value of 0x000081E8 before returning to the calling program. The simple instruction ADD R13, R13, #16 will balance the stack by changing R13 to 0x000081E8 as needed. The MOV PC, R14 instruction at the end of the procedure then uses the return address stored in R14 to return to the calling program.

If you need to pass more than eight parameters directly to a procedure, you can load another set of parameters in R0-R3 and use another STMFD R13!, {R0-R3} instruction to push these parameters on the stack. As another example of working with a stack map, Figure 4-28 shows a

map of the stack contents after pushing two sets of the R0-R3 registers on the stack with two STMFD R13!, {R0-R3} instructions and an STMFD R13!, {R4-R10} instruction at the start of the procedure to save R4-R10, as required by the APCS.

Address	Stack	Comments
0x00008180		<- R13 Before
0x0000817C	R3	Result after first
0x00008178	R2	STMFD R13!,
0x00008174	R1	{R0-R3}
0x00008170	R0	<- R13 After
0x0000816C	R3	Result after second
0x00008168	R2	STMFD R13!,
0x00008164	R1	{R0-R3}
0x00008160	R0	<- R13 After
0x0000815C	R10	Result after
0x00008158	R9	STMFD R13!,
0x00008154	R8	{R4-R10}
0x00008150	R7	
0x0000814C	R6	
0x00008148	R5	
0x00008144	R4	<- R13 After
0x00008140		

Figure 4-28. Stack map showing the contents of the stack after passing two sets of parameters on stack and saving registers R4-R10 on Full Descending stack.

Again as in the previous example, you have to add a value to R13, so that an LDMFD R13!, {R0-R3} instruction can access the R0-R3 values last pushed on the stack. Note that in your procedure, you only need to push the registers that you use in the set of R4-R10, so the number you must add to R13 depends on the number of registers specified in the STMFD instruction you wrote to save the used registers.

After you pop the four parameters last pushed on the stack and move them to other registers, the stack pointer (R13) will be pointing at the lowest of the first four parameters pushed on the stack. You can use another LDMFD R13!, {R0-R3} instruction to pop the four parameters first pushed on the stack into registers R0-R3 and either use the values directly or copy them into other registers to use them.

At the end of the procedure, you need to correct the value in R13 so that R13 points to the R4-R10 values stored on the stack and then use an STMFD R13!, {R4-R10} instruction to restore the registers. You can use the stack diagram in Figure 4-28 to help you determine the value you need to add to or subtract from R13, so that the LDMFD R13!, {R4-R10} instruction will load the desired values from the stack. Likewise, you can use the stack diagram to determine the additional correction you need to make to R13 after the LDMFD R13!, {R4-R10} instruction to balance the stack by moving R13 above the stack locations used to pass the two copies of R0-R3 on the stack. We leave these simple arithmetic problems for you to do as an exercise.

To summarize the preceding discussion, you can give a procedure access to desired data by passing the data directly in registers R0-R3, by directly passing pointers to the data in R0-R3, and/or by passing data or pointers on the stack. Again, we have found that a stack map such as those in Figures 4-27 and 4-28 are essential for keeping track of where all the values are located on the stack and the value in the stack pointer at any particular time as you do push and pop operations.

As a final important note here, the STM instruction we used to push registers on the stack can also be used for simply copying a block of data from a set of registers to general memory. Also, the LDM instruction used pop registers off the stack can be used for reading a block of data from general memory into a specified set of registers. For the example in this section we have used R13, the stack pointer, for STM and LDM instructions, because we were working with the stack. However, you can use other registers as the pointer in these instructions as, for example, STMFD R8!, {R4-R7} but the register used as a pointer should not be included in the register transfer list. As we mentioned in the optimization section of the chapter, the multiple Store and multiple Load instructions are more efficient than sequences of individual Store or Load instructions. The STM and LDM instructions each require only one word of memory and only one direct pipeline slot. These instructions do, of course, each require multiple clock cycles to store the specified registers or load the specified registers.

Now that you know how to write and call assembly language procedures, we can show you in the next section how to develop programs that contain both C language sections and assembly language sections.

Developing Programs That Contain Both C and Assembly Language Modules

As we have previously stated, you usually try to write as much of a system program as possible in a high-level language. The reason is that it is considerably faster and more efficient than writing the entire program in assembly language. However, for program sections that need to run in the absolute minimum amount of time, involve a lot of "bit-twiddling," or for which the compiler cannot create efficient code, we often write assembly language procedures and simply call these procedures from the high-level language program as needed. This technique is an important part of current microprocessor system design.

We will be using only assembly language for the program examples in the next few chapters but since it fits very well with our discussion of procedures and parameter passing, we will now show you how C programs call and pass parameters to functions, and how to write assembly language procedures that you can call from your C programs. One major goal of the section is to show you that the process for writing multi-language programs is understandable and doable. You can then refer back to this section of the chapter and dig more deeply into the details when you need them for later work.

THE C RUN-TIME ENVIRONMENT AND SETUP

The program environment that must be set up to run C programs is appropriately called the *C run-time environment*. The run-time environment consists of a library of pre-defined functions that your C program can call, a stack for passing parameters and saving registers, a "heap" that your programs can use for dynamic memory allocation, and perhaps some memory

protection/management capability. When you link a compiled C program, the code that sets up the run-time environment is automatically linked to the code for your C program. For a desktop system, the run-time added code may be quite large due to the large number of system functions. However, for small embedded systems, the run-time environment is kept as small as possible so that it requires minimal memory. ARM Ltd., for example, makes available a small run-time library that includes just division and remainder functions to compensate for the lack of a divide instruction; stack limit checking functions; stack and heap management functions; program start-up instructions; and program termination instructions. The linker will only link the run-time functions needed for a specific program, so for a simple embedded system the C run-time environment can be as small as a few hundred bytes.

When you run a C program, the start-up code sets up the stack and heap and then calls the function main(). All C programs must have one main () function. At the end of the main() function, unless the C program loops forever, execution returns to the operating system, debugger, or other system program that is being used to run the C program. Now that you have a basic understanding of how the run-time environment is set up for executing C programs, the next step is to look at how procedures are declared, defined, and called in C programs. With this knowledge as a base, it is an easy step to show you how to write assembly language procedures you can call from your C programs.

DECLARING, DEFINING, AND CALLING FUNCTIONS IN C PROGRAMS

To create and use a function in a C program, you must *declare* the function, *define* the function, and *call* the function. Figure 4-29 shows a template for how you do each of these, and Figure 4-30 shows several function examples in a simple C program. To understand the terms in the templates, it may help to look at corresponding parts in the example program. Don't worry about the details of the example program at this point because, after we discuss the templates, we will work through the different types of function calls in the example program.

```
TEMPLATES FOR DECLARING, CALLING AND DEFINING C FUNCTIONS

DECLARATION (PROTOTYPE)

type       function_name(variable list);
 ↑                        ↑
type of data              type and formal parameter (dummy)
returned by function      name for each variable to be passed


CALL

void main()
{
          function_name(actual arguments);
                              ↑
                        names of variables or pointers
                        to be passed to function this call
}

DEFINITION

type    function_name(formal arguments)
 ↑                          ↑              ↑
 |                          |          note: no ;
type of data           types and names of local
returned by            variables which correspond to
function               actual variables passed to function

     {
     statements;

     return(variable);
              ↑
        name of variable returned to
        calling function
     }
```

Figure 4-29. Templates for declaring, defining, and calling C functions.

The first step in writing a function is to write the function declaration or *prototype*. The function declaration is essentially a "block diagram" for the function. The int c2f(int c); statement at the top of Figure 4-30 is an example. The function declaration identifies the function for the compiler and indicates the types of data to be passed to and from the function. The compiler uses this information to make sure that the correct data is being passed to and from a function when it is called. For a large program, the function prototypes are all put in a separate header file and pulled into the program at compile time with a #include <filename.h> statement at the start of the program. The #include <stdio.h> statement at the start of the program in Figure 4-30 is an example of this. The stdio.h header file contains the function prototypes for predefined system functions such a scanf() and printf() that are used in the program.

As shown in Figure 4-29, a function declaration starts with a data type such as int, float, char, etc. The type in this case represents the data type for the variable returned to the calling program by the function. A C program can return the value of one variable directly to the calling program. If the function does not directly return a value to the calling program, you give the return type as "void." Note that the default return type of the main() function is int.


```
/* Declaring, calling, and defining functions */

#include<stdio.h>

int tempc, tempf;          /* external (global) variables */
int c2f(int c);            /* declare function c2f which returns an int value */

void get_temp(int *ptr); /* declare function which modifies a value
                            pointed to, but does not directly return a value */
void main()
{
     get_temp(&tempc);     /* call function get_temp.
                              get_temp writes directly to tempc */
     tempf = c2f(tempc);   /* call c2f function, pass value
                              of tempc to function. Returned value
                              assigned to tempf */

     printf("The temperature in Celsius is %d\n", tempc);
     printf("The temperature in Fahrenheit is %d\n",tempf);
) /* end of main */

int c2f(int c)             /* define function c2f. Note no ; at end */
     (
     int f;                /* automatic (local) variable */
     f = 9*c/5 + 32;
     return (f);
     )

void get_temp(int *ptr)    /* define function get_temp */
     (
     printf("Please enter the Celsius temperature.\n");
     scanf("%d",ptr);
     )
```

Figure 4-30. C program showing examples of declaring, defining, and calling C functions

After the function name in the function declaration, you enclose in parentheses a list of variables that will be passed to the function and specify the data type for each of these variables. These variables identified in the function declaration are often referred to in programming texts as formal arguments or formal parameters. We often call them "dummy" variables because they are placeholders for the "actual" variables that will be passed to the function when it is called. The int c2f(int c); statement at the top of Figure 4-30, for example, indicates that the value of a variable of type int will be passed to the c2f function when it is called in a program. As another example of a function prototype, the declaration void get_temp(int *ptr); in Figure 4-30 declares a function that does not return a value but requires that a pointer to an int type variable be passed to it. (Remember that pointer is just another name for an address.)

After you construct the function declaration, you define the actual function. Functions are always defined outside of main() because you cannot define one function within another. As shown in the c2f function in Figure 4-30, a function starts with a function header that has the same format as the function declaration but is not followed by a semicolon. The header indicates that the int value passed to the function when it is called will automatically be assigned to the local variable c in the function. To define the function, you will write the data declarations and action statements for the body of the function. Note that the statement block for the function is

enclosed in "curly braces." The return (f); statement in the c2f function in Figure 4-30 passes the int value of the local variable f, calculated in the function, back to the calling program.

You call a C language function with its name and a set of parentheses that contains the names of variables or pointers that you want to pass to the function. If you are not passing any values to the function, you put the term 'void' in the parentheses. The variables named in the function call are sometimes referred to in programming textbooks as *actual arguments* or *actual parameters* because these represent the values actually passed to the procedure. The actual parameters usually have different names than the formal parameters in the function prototypes, so that the functions are generic. Note that C programs always "pass by value." This means that only the value of a named variable is passed to a function or in other words, only a copy of the variable is passed to the function. The initial value of the variable is not changed by the operations performed on the value passed in the function call. If you want a function to be able to directly modify the value of a variable, you need to pass the function a pointer to the variable or in other words, the address of the variable. The function will then be able to use the pointer to access and modify the variable directly. However, note that a value returned by a function can be assigned to a variable in the calling program. To see an example of this and other examples, let's take a closer look at the program in Figure 4-30.

The overall actions for the program in Figure 4-30 are to:

1. Prompt the user to enter a Celsius temperature on a keyboard.
2. Scan the keyboard to read the entered Celsius temperature.
3. Convert the Celsius temperature to an equivalent Fahrenheit temperature.
4. Print the Celsius and Fahrenheit temperatures on a display device.

In the actual program, we first declare an int type variable called tempc that will be used to hold the value of the Celsius temperature entered by the user, and an int type variable called tempf to hold the value of the Fahrenheit temperature. As we will explain in more detail later, these variables are declared outside of any function, so they are *external* or *global*. This means that they can be directly accessed by other program modules, even if they are in separate files.

Next in the program in Figure 4-30 we declare the two functions, get_temp and c2f, that we are going to call in the program. Remember that the int *ptr in the void get_temp(int *ptr) declaration indicates that we will pass a pointer to an int type variable when we call this function. In main() we call the get_temp function with the statement get_temp(&tempc);. In C programs, the & symbol is shorthand for the words "the address of," so the &tempc can be read as "the address of the variable tempc." Since a pointer to a variable is just the address of the variable, the statement get_temp(&tempc); calls the procedure and passes it a pointer to tempc, so that the procedure can write a value in tempc.

In the get_temp procedure we call the predefined library function printf to display a prompt for the user to enter the Celsius temperature, and then we use the library function scanf to read the value entered by the user. Note that we pass the pointer passed from the mainline to the scanf function, so scanf can write the value read from the keyboard directly into tempc. The %d in the scanf function call tells scanf to write the value in tempc as a decimal value.

The tempf = c2f(tempc); statement in Figure 4-30 calls the c2f function, passes the value of tempc to the function, and writes the value returned by the function to the variable tempf. In the c2f function we declare the local variable f to hold the computed value of the Fahrenheit temperature. The term "local" here means that the variable f can only be accessed within the c2f

function and in functions called by c2f. For the actual calculation of the Fahrenheit value, the multiplication and addition operations map directly into ARM instructions. However, if a particular ARM-based processor does not have a divide instruction, the compiler will call one of the run-time functions to do the division if it cannot determine a way to do the specific division with subtract or shift instructions. Note that the priority of operators in C is multiplication/division and then addition/subtraction, so the multiplication by 9 is done first, then the division by 5, and finally the addition of 32.

The return (f); statement at the end of the c2f procedure passes the calculated value of the Fahrenheit temperature back to the mainline, where it is written to the tempf variable. Finally in the mainline program, we call the library printf function to display the Celsius temperature passed to it and then call printf again to display the Fahrenheit temperature.

Before we show you how to write assembly language procedures you can call from your C programs, we need to talk a little about two important properties of variables and functions in C programs. The two properties of any variable or function declared in a C program are lifetime and visibility or "scope." As the name implies, *lifetime* refers to how long a declared variable keeps its value in a program. The *visibility* or *scope* of a declared variable is a measure of whether it can be accessed from other source files, accessed from other blocks, or accessed just within the source file or block where it is declared. To help with our discussion of these terms, Figure 4-31 shows some simple examples.

As we mentioned earlier, variables or functions declared outside of main() are by default external or, in other words, global. This means that they are visible or accessible from anywhere in the source file where they are defined or from other files that are linked with that file. The int tempf; and the int c2f(int c); declarations in Figure 4-30 are examples of this. A global variable maintains its value for as long as the program is running. Note that, if want to access a variable or function that is declared and defined in another source file, you need to identify it as *extern* in the source file you are developing. The extern int book_total; declaration in Figure 4-31 is an example of this.

If you want a variable or function to be visible only within the current source module, you put the word *static* at the start of the declaration. The static int tempc; and the static int f2c(int f); declarations in Figure 4-31 are examples.

```
VARIABLE EXAMPLE                LIFETIME        ACCESSIBILITY
int tempf;                      program         all source files
static int tempc;               program         this file only
extern int book_total           program         defined in another file
int c2f(inc c);                 program         all source files
static int f2c(intf)            program         this source file only

void main ()
{
int count;                      block           block and sub blocks
                                                after declared
static int interrupt_cnt;       program         block and sub blocks
                                                after declared
register int index              block           block and sub blocks
                                                after declared
```

Figure 4-31. Examples showing Lifetime and Accessibility(Visibility) for C variable and function declarations.

As we also mentioned earlier, variables or functions declared within a function are local or, in other words, accessible only within that function or in functions called by that function.

Variables declared within a function are commonly called *automatic* variables. The int count; declaration under main() in Figure 4-31 is an example. Each time you call a function that declares an automatic variable, a temporary storage space for that variable is allocated in memory and that location holds the value of the variable as the function executes. When execution returns to the calling program, the memory space is de-allocated so the value of the variable is lost. An automatic variable then only "lives" during the execution of the function block where it is declared. If you want an automatic variable to retain its value after execution returns to the calling program, you put the word "static" at the start of the declaration. An example in Figure 4-31 is the static int interrupt_cnt; declaration.

Finally in Figure 4-31, note the register int index; declaration. The variable index is automatic because it is declared in the function main(), but the word "register" at the start of the declaration tells the compiler to assign this variable to a register rather than to an allocated memory location. As you know, due to memory access time, it is usually much more efficient to work with a variable in a register than to work with the variable in a memory location. Next here, we will summarize the key points that we have discussed about C functions. Then, in the next section we show you how to call assembly language procedures from C programs.

To implement a function in a C program, you need to:

1. Declare the function to tell the compiler the data types for value(s) that will be passed to the function and the data type of any value that will be returned by the function.

2. Define (write) the function and give it a header. The function header has the same format as the function declaration but is not followed by a semicolon. This must be done outside of main(). It may be done in the same source file as main() or in another source file. The header contains "dummy" names and the types for the parameters that are to be passed to the function. If a variable is declared within a function, it is automatic (local) and is accessible only to that function and to functions called by that function. The value of an automatic variable is lost when execution returns to the calling program unless the variable is declared with the word "static" in the declaration.

3. Call the function by name and pass it the values of the actual parameters that are to be used in the function. Except for the first two calls to printf, the examples in Figure 4-30 each function call passes only one parameter to a function, but you can specify multiple parameters to pass with a comma-separated list. C programs pass only a copy of a specified variable(s), so if you want a function to modify a variable directly, you must pass the function a pointer to the variable. If a value is returned by the function, it can be assigned to a declared variable in the calling program.

CALLING AN ASSEMBLY LANGUAGE PROCEDURE FROM A C PROGRAM

Calling an assembly language procedure from a C program is very much like calling a C function from a C program. The major steps you need to do for this are:

1. Write the C program and declare the assembly language procedure in the C program with a directive that identifies the procedure as *extern*. The unsigned short extern BCD2BIN (unsigned short b); declaration in the C program in Figure 4-32a, for example, tells the compiler that the function (procedure) _BCD2BIN is in another file that will be linked

with the object file for the C module at link time. The declaration also tells the compiler that an unsigned short (16-bit) value will be passed to the procedure and that an unsigned short (16-bit) value will be returned by the procedure. (You usually write the C program first and then decide to pass some task off to an assembly language procedure. From this, you know what to pass to the procedure and what you want to have returned, so you can write the function declaration.)

In the C program, write the statement that calls the assembly language procedure. The BIN = _BCD2BIN (BCD); statement in the example program in Figure 4-32a, for example, calls the _BCD2BIN procedure, passes the value of the variable BCD to it, and assigns the value returned by the procedure to the variable BIN.

2. Compile the C program and correct any syntax errors.

Write the assembly language procedure following the APCS rules as stated earlier in the chapter. A C program will pass parameters to a function or procedure in registers R0-R3 and, if there are more variables to pass than can be passed in R0-R3, the remaining will be passed on the stack. (For program efficiency, you should try to limit the number of parameters passed to four or less so you do not need to use the stack, because accessing the stack can eat up a lot of clock cycles.) If a single parameter is passed, as in the BIN = _BCD2BIN(BCD); call in Figure 4-32a, then you know that the value of BCD will be passed to the procedure in register R0 when the program is compiled to run on an ARM-based processor. If multiple parameters are passed in a procedure call, the first in the list in the procedure call parentheses will be passed in R0, the second in R1, etc In the assembly language procedure, preserve any registers you use in the set of R4-R10 by pushing them on the stack as required by the APCS. Remember that R11 and R12 can be used as scratch pad registers and do not need to be saved.

If a value needs to be returned to the calling program, move this value to R0 before returning to the calling program.

At the start of the assembly language procedure, include the .global directive that makes the assembly language procedure "public" so that it can be accessed by other modules. For the _BCD2BIN procedure you would use the directive .global _BCD2BIN.

Figure 4-32b shows the assembly language _BCD2BIN procedure that is called by the C program in Figure 4-32a. We will work through the algorithm of this program a little later, but for now just note how we made a couple of copies of the BCD value passed in R0, used R11 and R12
to do the conversion so we didn't need to push any of the R4-R10 registers, and then passed the result of the conversion back to the calling program in R0.

3. Assemble the assembly language module and correct any syntax errors.

4. Link the object file for the C module with the object file for the assembly language module to create a single executable file.
5. Use the debugger to run and test the executable file. The disassembled program display produced by the debugger gives you a lot of useful information about the program flow for the C section of a program and if needed, for the assembly language section In the next section we will take a look how the assembly language function is called and the

returned value handled in main(). Then in the next section we will briefly discuss the BCD2BIN procedure in 4-32b.

Cforasm.c

```
/* Example of C program that calls asm program */
/* Copyright Doug Hall                         */

unsigned short BCD = 0x47;
unsigned short BIN;

extern unsigned short _BCD2BIN(unsigned short b);

int main() {
  BIN = _BCD2BIN(BCD);
}
```

(a)

asmforc.s

```
@ ARM ASM procedure to convert BCD byte to Binary
@ Doug Hall

.text
.global _BCD2BIN

_BCD2BIN:
        MOV     R11,R0              @ BCD VALUE PASSED IN R0
        MOV     R12,R0              @ COPIES IN R11 AND R12 (scratch registers)
        AND     R11,R11,#0x0F       @ MASK ALL BUT LS NIBBLE
        AND     R12,R12,#0xF0       @ MASK ALL BUT SECOND MS NIBBLE
        MOV     R12,R12,LSR #4      @ MOVE TO LS NIBBLE POSITION
        ADD     R12,R12,R12,LSL #2  @ MULTIPLY BY 5
        MOV     R12,R12,LSL #1      @ MULTIPLY BY 2
        ADD     R12,R12,R11         @ ADD LS NIBBLE FROM R11
        MOV     R0,R12              @ RESULT TO R0 FOR RETURN
        MOV     PC,LR               @ RETURN

.end
```

(b)

Figure 4-32. Program with C and Assembly modules. (a) C mainline. (b) Assembly language procedure.

THE DEBUGGER ASSEMBLY LISTING FOR MAIN

```
0xa0000000 <main>:    mov   r12, sp
0xa0000004 <main+4>:  stmdb sp!, {r11, r12, lr, pc}  @ Push registers
0xa0000008 <main+8>:  sub   r11, r12, #4    ; 0x4    @ Balance stack
0xa000000c <main+12>: ldr   r12, [pc, #28]  ; 0xa0000030 <main+48>
0xa0000010 <main+16>: ldrsh r12, [r12]      @ Load BCD value from memory
0xa0000014 <main+20>: mov   r0, r12         @ Copy BCD to R0 to pass
0xa0000018 <main+24>: bl    0xa000006c <_BCD2BIN>  @ Call BCD2BIN
0xa000001c <main+28>: mov   r3, r0          @ Move value returned to R3
0xa0000020 <main+32>: ldr   r12, [pc, #12]  ; 0xa0000034 <main+52>
0xa0000024 <main+36>: strh  r3, [r12]       @ Write to BIN
0xa0000028 <main+40>: mov   r0, r12         @ Address of BIN in R0
0xa000002c <main+44>: ldmdb r11, {r11, sp, pc} @ Pop registers and return
0xa0000030 <main+48>: andge r0, r0, r4, lsr #1
0xa0000034 <main+52>: andge r0, r0, r0, asr #1
```

Figure 4-33. Disassembly of main() function for program in Figure 4-32.

One key new point in the debugger listing for main() in Figure 4-33 is that the code loads the value of BCD from memory and copies it to R0 to pass to _BCD2BIN as specified by the APCS. Another point is that upon return, the code copies the BIN value returned in R0 to R3 and then stores it in memory at the BIN location. Note that the memory access instructions use PC relative addressing.

THE BCD TO BINARY ALGORITHM

As implied by its name, the purpose of the BCD2BIN procedure is to convert a given 8-bit BCD number to its binary equivalent. The algorithm for this is to simply multiply each BCD digit by its binary value and sum the results for all the digits. For example, the BCD number 0x47 represents the decimal number 47. The decimal number 47 is equal to 4×10 + 7×1. The binary equivalent is just 4×0xA + 7×0x1. The BCD2BIN procedure separates the BCD digits and performs the operations to produce the binary equivalent of an 8-bit (2-digit) BCD number.

In the procedure, we copy the BCD value passed in R0 to R11 and R12, the scratch pad registers, to do the conversion. Then in R11 we isolate the lower BCD digit. In R12 we isolate the upper BCD digit and shift it right 4 bit positions into the LSB positions of the register. The next step is to multiply the value in R12 by decimal 10 or 0xA, since the value of this BCD digit is 0xA. Instead of doing this with a MUL instruction, we do it by first multiplying it by 5 and then multiplying it by 2 with the help of shift operations. The ADD R12, R12, R12, LSL #2 instruction adds R12 shifted two bits left to R12 and puts the result in R12. Since R12 shifted two bit positions left is R1 x 4, the result in R12 is 4×R12 + R12 = 5×R12. The MOV R12, R12, LSL #1 instruction shifts R12 one bit position to the left to multiply it by 2. This approach is faster than the MUL instruction.

At the end of the procedure we move the result from R12 to R0 for return to the calling program as specified by the APCS, and then return to the C program section with the familiar MOV PC,LR instruction. As you can see from the examples in Figure 4-31, it is not difficult to

write procedures that are callable from your C programs if you just follow the basic rules. It is also possible to call C library functions from an assembly language program but, since you will usually write the main program for an application in C or some other high-level language, it is much more likely that you will want to call an assembly language procedure from a C program as we have shown you how to do in this section. In the next and last section of this chapter, we briefly show you how it is possible to write short sections of assembly language code directly in C programs

IN-LINE ASSEMBLY LANGUAGE PROGRAMMING

Most current C compilers such have a built-in assembler that makes it possible for you to embed short sections of assembly language directly in your C programs to perform simple initializations or other bit-twiddling operations. Figure 4-34 shows how the BCD2BIN operation that we did with a procedure call in Figure 4-32 can be done within a C program as *in-line assembly code*. The assembly language section starts with two underscores followed by the keyword asm. The assembly language statements are then enclosed in parentheses marks as shown. Note that the quotation marks and the \n\t are required on each assembly language instruction. Also note the terminating ; after the close parenthesis at the end of the assembly language instructions. Further note that the comments for assembly language instructions in a C program must use the C language comment format as shown. Also, a section of in-line assembly code in a C program must be within a C function. For this simple example, we just put it directly in the main() function.

inline.c

```
/* Example of C program that uses in-line assembly */
/* Doug Hall                                       */

static unsigned short BCD = 0x47;
static unsigned short BIN;

int main(void)
{
__asm
(
        "LDR    R11,=BCD\n\t"
        "LDRH R11,[R11]\n\t"                    /* BCD VALUE INTO R11 */
        "MOV  R12,R11\n\t"                      /* R11,R12 ARE scratch registers*/
        "AND    R11,R11,#0x0F\n\t"              /* MASK ALL BUT LS NIBBLE */
        "AND    R12,R12,#0xF0\n\t"              /* MASK ALL BUT SECOND MS NIBBLE*/
        "MOV    R12,R12,LSR #4\n\t"             /* MOVE TO LS NIBBLE POSITION */
        "ADD    R12,R12,R12,LSL #2\n\t"         /* MULTIPLY BY 5 */
        "MOV    R12,R12,LSL #1\n\t"             /* MULTIPLY BY 2 */
        "ADD    R12,R12,R11\n\t"                /* ADD LS NIBBLE FROM R11*/
        "LDR    R11,=BIN\n\t"                   /* LOAD ADDRESS OF BIN*/
        "STRH   R12,[R11]");                    /* WRITE RESULT TO BIN */
}
```

Figure 4-34. Example of C program that uses inline assembly code.

In the program in Figure 4-34, we first use the LDR R11,=BCD instruction to load the address of BCD into R11 then use the LDRH R11, [R11] instruction to read the value from the

BCD memory location. Likewise, after we calculate the BIN value, we load the address of BIN in R11 and then use R11 to write the result to the BIN memory location.

One major advantage of in-line assembly is that it does not have the overhead of a procedure call. Another advantage is that it can make use of some compiler optimizations. The disadvantages of in-line assembly are as follows:

1. Using in-line assembly does not reduce code for repeated sections as does using an assembly language procedure. The in-line sections must be included in the code each time they are needed.
2. The use of physical registers or the stack in the in-line assembly may cause problems due to conflicts with the use of the registers and stack in C expressions. (You may use the debugger to help you fix one of these problems and then find that, after making a small change in the C code, the program is broken again.)
3. Using in-line code in C programs makes it more difficult to port the C code to different architectures.

Based on our experience and on the above stated disadvantages of in-line assembly, we feel that it is much better programming practice to put assembly language sections in procedures that you can call as needed. This approach avoids the duplication that may be required for in-line assembly. A further advantage is that if the assembly language is all contained in procedures, it is much easier to port to another architecture because, as we demonstrated earlier, the APCS clearly defines the interface between C and assembly language procedures. As long as you follow the APCS, you will not need to endure a lengthy debugging session trying to find mystery "side-effects" caused by in-line assembly register conflicts.

USING "MAKE" TO AUTOMATE THE COMPILE AND LINK STEPS

For simple programs, command line entry of compile and link commands works fine. For larger programs that contain a large number of separate program files, however, this method is not workable. For large programs we create a file that contains the compile commands for all of the individual source files and the link command that produces the final executable file. The file containing the commands is usually given the name makefile or Makefile. (We like to use the name Makefile, so the file appears near the top of a directory listing.) To run the sequence of commands in the Makefile, you simply enter the command "make" at the command prompt. When you run the make program, it automatically determines which files in a large program need to be recompiled/reassembled because they have been changed, executes the command(s) that will recompile those files, and then runs the command that will link all the object modules to create an updated executable. For a complete discussion of the make utility, consult the make manual at http://www.gnu.org/software/make/manual/make.html.

If you are using, the TI code Composer Studio tools, the Build or Rebuild command automatically runs a default make file that is generated using the settings you specify for the assembler, compiler, and linker in the setup menus. The makefile for a project is located in the debugger folder for the project in the specified workspace.

Summary

In this chapter, we have introduced you to assembly language and C-assembly language programming techniques for ARM-based microprocessors. The problems at the end of the chapter will give you some practice with these basic techniques, and later chapters will teach you much more about writing programs for interfacing with a wide variety of devices and systems. In the next chapter, we show you how to work with interrupts and interrupt procedures that are very important parts of ARM-based microprocessor systems.

References

(1) GNU assembler (AS) manual http://www.gnu.org/software/binutils/manual/gas-2.9.1/as.html

(2) GNU Linker (LD) manual http://www.gnu.org/software/binutils/manual/ld-2.9.1/ld.html

(3) GNU GCC manual http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/

(4) GNU make manual http://www.gnu.org/software/make/manual/make.html

(3) TI AM335X Sitara Technical Reference Manual
http://www.ti.com/lit/ug/spruh73n/spruh73n.pdf

(5) ARM Cortex A-8 Technical Reference manual,
http://dl.btc.pl/kamami_wa/ti_myc_am335x_technical_reference_manual.pdf

Checklist of Important Terms and Concepts

Packed BCD
Round, truncate
A/D converter
GPIO pins
GPIO Control Registers
Read-Modify-Write operation
Optimization techniques
- MOV Rn and MOVN Rn
- Base + Add address generation
- Loop counts
- Auto-Increment, Auto-Decrement addressing
- Multiply/divide by shift
- BIC instruction

Instruction scheduling
Load and Store Multiple instructions
Conditional execution to remove branches
Stack, top of stack

Full descending stack
Empty descending stack
Push and pop operations
Processor state
Passing parameters to a procedure
C Runtime environment
C function declaration
C function definition
C function call
C variable scope
C variable lifetime
C external variable
C automatic variable
ARM Procedure Call Standard (APCS)
In-Line-Assembly
Makefile

Review Questions and Problems

1. A simple sequence problem:
   a. Write an algorithm for a program that converts two BCD digits packed in a byte to the ASCII codes for the two BCD digits in different words.
   b. Write the assembly language for your algorithm.
   c. If you have an appropriate system available, assemble, test, and debug your program.

2. A simple Repeat-Until problem:
   a. Write an algorithm for a program that computes the truncated average of 5 unsigned byte values stored in an array of bytes in memory, and stores the result in a separate memory location reserved for the average. Hint: You can use successive subtraction to do a division.
   b. Write the assembly language for your algorithm.
   c. If you have an appropriate system available, assemble, link, test, and debug your program.

3. A nested If-Then-Else problem:
   A common problem when reading a series of ASCII codes from a keyboard for a control application is the need to filter out the those codes which represent the hexadecimal digits 0-9 and A-F, and convert these ASCII codes to the simple hexadecimal digits. For example, if we read in 0x34, the ASCII code for 4, we want to mask all but the lowest four bits to leave 0x04, the 8-bit hexadecimal code for 4. If we read in 0x42, the ASCII code for B, we want to mask all but the lowest four bits and add 9 to leave 0x0B, the 8-bit code for hexadecimal B. If we read an ASCII code that is not in the range of 0x30 – 0x39 or 0x41- 0x46, we want our program to give an error value of 0xFF. Figure 4-35 shows the desired results for each range of ASCII code values.

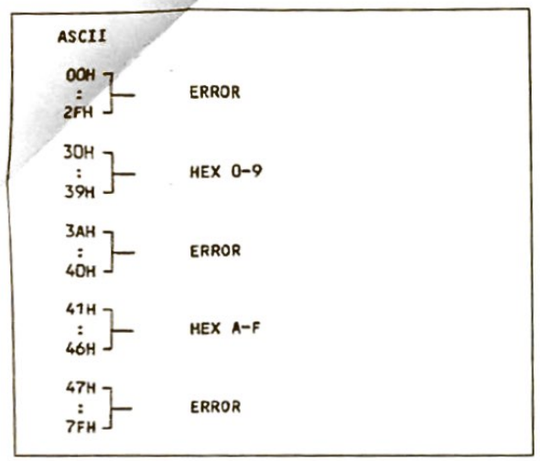


Figure 4-35. Desired action for each range of ASCII values in problem 3.

a. Write an algorithm for a program that produces the desired result for values in each range of the ASCII codes.
b. Write the assembly language for your algorithm.
c. If you have an appropriate system available, assemble, link, test, and debug your program.

4. A While-Do problem:
   The ARM-based processors have both signed and unsigned multiplication instructions but some do not have divide instructions. As shown in the this chapter, you can divide a number by a power of 2 by simply shifting it right the number of bit positions equal to the power of 2. For cases where shifting does not work, you can do division by successive subtraction of the divisor (bottom number) from the dividend (top number) until the result is less than the divisor.
   a. Write an algorithm for a program that divides a given 32-bit unsigned number by a given 16-bit unsigned integer and produces a rounded result. Make sure to think about the action(s) you want for the boundary conditions of "divisor greater than dividend" and "divisor equal to zero."
   b. Write the assembly language for your algorithm.
   c. If you have an appropriate system available, assemble, test, and debug your program.

5. For the following problems, do the indicated action or improvement.
   a. Write a simple sequence of ARM instructions that will perform the same function as the LDR R1, =0x00020026 instruction without needing to access a literal pool in memory.
   b. Write the assembly language instruction(s) that will load 0x00FFFFFF in R2 without using a literal pool.

   c. Write a single ARM assembly language instruction that will perform the same function as the two instructions MOV R2, R2, LSL#4, and ADD R2, R1, R2.
   d. Write a single ARM assembly language instruction that will multiply R3 by 64 without using the MULT instruction.
   e. Write a single ARM assembly language instruction that will multiply the contents of R4 by 9 without using the MULT instruction.
   f. Write an ARM assembly language instruction that uses the BIC instruction to clear bits 16-23 in R7.
   g. Show how you would schedule the three instructions MULT R2, R2, R1; ADD R2, R3, R2; and SUB R8, R9, R8 to reduce the chance of an RAW hazard.

6. In the text we showed you how the branch instructions in a 2-LED program could be eliminated by using the conditional execution capability of the ARM-based processors. Attempt to rewrite the 3-LED program in Figure 4-16 using the conditional execution capabilities to remove as many branch instructions as possible in the nested If-Then-Else section of the program and discuss your result.

7. In the chapter we showed how you can create an approximate delay by simply counting down a value loaded into a register.
   a. Write an algorithm for a procedure that introduces a delay time by counting down a value passed to it in a 32-bit register.
   b. Write the mainline program that calls the procedure and passes the 32-bit value to the procedure, and write the assembly language for the delay procedure.
   c. If you have an appropriate system available, assemble, test, and debug your program.

8. Summarize the APCS rules for the use and preservation of registers during procedure calls and returns.

9. Assuming that the a system has a Full-Descending type stack and that the stack pointer (R13) initially contains 0x00008400, draw a stack map that shows the contents of the stack and the value in the stack pointer after the instruction STMFD R13!, {R4-R7} executes.

10. For the example stack map in Figure 4-28:
    a. Determine the value that must be added to R13 to jump up over the locations in the stack where the registers R4-R10 were pushed in the procedure so you can access the second copy of R0-R3 that was passed to the procedure on the stack.
    b. Determine the value that must be subtracted from R13 in order to be able to pop the registers that were saved with the STMFD R13!, {R4-R10} instruction after both copies of R0-R3 on the stack have been popped off.
    c. Determine the value that must be added to R13 to "balance" the stack after an LDMFD R13!, {R4-R10} instruction is used to pop R4-R10 off the stack.

11. An ARM-based process control system outputs a measured Fahrenheit temperature to a display on its front panel. You need to write a procedure that converts the Fahrenheit value to Celsius, so the system can be successfully marketed in Europe. The relationship between

Fahrenheit and Celsius is C = (F-32) × 5/9. Assume that the Fahrenheit value will always be in the range of 50-250 degrees and you want to round the result to the nearest Celsius degree.

a. Write an algorithm for a procedure that converts a Fahrenheit temperature passed in a register to a Celsius equivalent and returns the calculated value to the calling program in a register.
b. Write the assembly language for your procedure and for the assembly language mainline that calls your procedure.
c. If you have an appropriate system available, assemble, test, and debug your program.

12. List and briefly describe the three tasks required to implement functions in C programs.

13. 2Write the algorithm for the BCD2BIN procedure in Figure 4-31b and extend the algorithm so that it works for the conversion of a 4-digit BCD number to its binary equivalent instead of just converting a 2-digit BCD number. Write the assembly language procedure for your algorithm, then build, test, and debug the combined program modules.

16. Write a simple C mainline program to call the Fahrenheit to Celsius procedure from question 11. Then modify the procedure as needed so that it can be called from the C program. Remember to follow the APCS guidelines. Build, test, and debug the combined program modules.

17. Explain why it is usually better programming practice to write and call an assembly language procedure in a high-level language program rather than using in-line assembly code.

CHAPTER 5 - INTERRUPT PROCEDURES, CONTROLLERS, AND TIMERS

In Chapter 3, during a discussion of the operating modes for ARM-based processors, we mentioned that most microprocessors allow an "exception" such an external signal, a program error condition, or an interrupt instruction to interrupt normal program execution, switch the processor to a privileged execution mode, and automatically call a special procedure that takes some action to respond to the condition that produced the exception. These special procedures are called *interrupt service procedures* or *exception handlers*. (We will use the term "interrupt service procedure" for procedures that are called by hardware signals and we will use the term "handler" for procedures that are called by error conditions.) At the end of the interrupt service procedure or exception handler, a special instruction returns execution to the interrupted program, so execution can continue from the point where the exception occurred.

Interrupt procedures and exception handlers are very important parts of nearly every microprocessor-based system, so it is important for you to become skilled in developing and using them. In this chapter we review the ARM-based processor exception types, describe how the processor responds to each of these exceptions, teach you how an interrupt vector table is used, and how to write interrupt service procedures for any ARM-based processor system. Then for a specific system example, we show you how to work with interrupts and the Interrupt Controller in a TI Sitara processor. We also discuss and show how interrupts are used in a variety of important applications. The first application example shows you how to generate and service an interrupt request from a GPIO pin. A second example shows you how to use a programmable timer to measure precise time intervals on an interrupt basis. The final interrupt application example shows you how to communicate with an external text-to-speech module on an interrupt basis, using one of the AM3358's built-in peripheral UART controllers.

Objectives

At the end of this Chapter, you should be able to:

1. List and describe the types of exceptions in ARM-based microprocessors and the execution mode associated with each.
2. Describe how an ARM-based processor responds to each type of exception.
3. Describe how an interrupt vector table is used to direct execution to a desired type of interrupt service procedure
4. Write an assembly language program that "hooks" an interrupt vector and "chains" an interrupt procedure.
5. Write an interrupt service procedure for a specified interrupt source.
6. Write general-purpose procedures so that they are reentrant and can function correctly in an interrupt-driven system.
7. Describe how interrupt procedures can be nested so a higher priority interrupt can interrupt execution of a lower priority interrupt service procedure of the same type.
8. For an ARM Cortex-A8 based processor, configure GPIO pins as interrupt inputs, unmask interrupt inputs, and direct hardware interrupts to the processor IRQ or FIQ interrupt inputs

9. Write an ARM Cortex-A8 based processor interrupt service procedure that reads data from a peripheral device on an interrupt basis rather than a polled basis.
10. Use an ARM Cortex-A8 based processor Timer to produce precisely timed interrupts.
11. Write a program that communicates with a peripheral module over an RS-232C serial line on an interrupt basis using one of the ARM Cortex A-8 based processor's UARTs.
12. Describe how a logic analyzer operates and how it can be used to help develop a program such as an RS-232C I/O driver.

Arm-Based Processor Exceptions and Exception Procedures

ARM-BASED PROCESSOR RESPONSES TO AN EXCEPTION

In response to an exception, an ARM-based processor will:

1. Change the operating mode of the processor to the mode associated with that exception.
2. Switch to a new R13 (Stack Pointer) associated with the exception mode to which the processor switches as shown in Figure 3-1. Since each mode has its own R13, each mode can have its own, independent stack that is protected to some extent because it cannot be directly accessed from other modes. Note that as shown in Figure 3-1, the FIQ mode also has duplicate registers that replace R8-R12 when execution switches to FIQ mode. These duplicate registers reduce the number of registers that have to be pushed on the stack when an FIQ interrupt is serviced.
3. Save the *return address* in R14 of the NEW mode. The return address in this case is the address of the next instruction to be executed in the interrupted program.
4. Save the Current Program Status Register (CPSR) of the interrupted program in the Saved Program Status Register (SPSR) of the new mode.
5. In the CPSR of the new mode, disable IRQ interrupts by setting bit 7 of the CPSR. If the interrupt was caused by an FIQ signal, also disable further FIQ interrupts by setting bit 6 in the CPSR of the new mode. These two are disabled to prevent interrupt signals still present on the IRQ and/or FIQ pins from producing continuous interrupts. Continuous interrupts would keep calling the interrupt procedure and not allow the procedure to service the condition that created the interrupt signal. Note that, if IRQ and FIQ were enabled in the mode that was interrupted, they will automatically be re-enabled when the CPSR of the interrupted program is restored at the end of the interrupt service procedure.
6. Automatically "calls" the procedure written to service the exception. The processor does this call by reading the first instruction of the procedure from a dedicated address in an *interrupt vector table*,usually located in memory starting at absolute address 0x00000000.
7. At the end of the interrupt service procedure, a special instruction copies the Saved Program Status Register (SPSR) from the executing privileged mode back to the Current Program Status Register (CPSR) of the mode that was interrupted, and returns execution to the interrupted program at the correct location. As mentioned previously, IRQ and FIQ will automatically be re-enabled when the CPSR of the interrupted program is restored, if they were enabled when the exception occurred.

In the following sections, we will discuss each of these actions in detail. To start, we will review the ARM-based processor execution modes that we introduced earlier.

REVIEW OF ARM-BASED PROCESSOR EXCEPTION TYPES AND EXECUTION MODES

In Chapter 3 we introduced you to the ARM processor User operating mode and the six *privileged* operating modes: System, Supervisor, Abort, Undefined, IRQ and FIQ. These six modes are called privileged because User mode programs cannot switch to one of these modes to access system resources, except through a special mechanism that we describe later. This restriction provides one way to protect system programs from malfunctioning user programs. A program operating in one of the privileged modes can, of course, directly change the operating mode to any one of the other privileged modes or to User mode by simply changing the mode bits in the CPSR to the bit pattern for the desired mode.

The five exception modes, Supervisor (SVC), Abort (ABT), Undefined (UND), Interrupt (IRQ) and Fast Interrupt (FIQ), are normally entered when the processor experiences some *exception*. FIQ is an example of a *hardware exception* or *interrupt*. Asserting the FIQ input pin on the processor will interrupt the executing program and cause the processor to switch to the FIQ execution mode. The Software Interrupt Instruction (SWI) is an example of a *software exception*. When the SWI instruction in a program executes, the processor will be switched to the Supervisor (SVC) execution mode. Detection of an illegal instruction code in the instruction code stream is an example of an *execution exception* and is commonly referred to as a *fault*. To refresh your memory of the ARM-based processor modes and the mechanism by which each is entered, here again are brief descriptions of the seven modes and, where appropriate, the condition or exception that switches the processor to that mode. Again, please refer to Figure 3-1 for the register set that is used in each of these modes. Especially note that each of the exception modes has its own R13 (Stack Pointer) to point to a separate stack for that mode, an R14 to save the return address for the interrupted program, and an SPSR to save the CPSR of the interrupted program.

USER (USR) – NORMAL APPLICATION PROGRAM EXECUTION MODE.

System (SYS) – In an embedded system with an operating system, the main operating system will likely be run in System mode. As shown in Figure 3-1, System mode uses the same register set as User mode, but unlike User mode, System mode allows direct access to the CPSR. When the system power is first turned on, the Reset signal is asserted, so the processor will go to Supervisor mode. After initializing stacks for all the different modes and initializing all the peripheral devices, the Supervisor mode program will then switch the processor to System mode to run the main part of the operating system.
Supervisor (SVC) – The processor enters Supervisor mode in response to a signal on the processor Reset input or in response to the Software Interrupt Instruction (SWI). Supervisor mode is used to initialize stacks and peripheral devices. The operating system "kernel" is also likely run in Supervisor mode, because as we will discuss later, this mode allows access to control registers that code in other modes is not allowed to access. As we will also discuss in detail later, User and System mode programs can only call "kernel" I/O procedures using a SWI instruction. This mechanism prevents User mode code from modifying the kernel code and crashing the system. Incidentally, the TI CCS Debugger operates in Supervisor mode so, if you look at the contents of the CPSR when you are debugging a program, you will see that the mode bits are 00011, the code for Supervisor mode.

Abort (ABT) – On processors such as ARM Cortex-A8 based processors, there are several conditions that can cause an Abort exception. The easiest ones to describe at this point are those caused when the processor attempts to fetch an instruction or data from memory and finds that the instruction or data has not been loaded from disk into the physical memory on the board. For these cases, the procedure that services the Abort exception will load the required code or data from disk into physical memory and return execution to the interrupted program at the instruction that caused the memory fault. This instruction will then be executed again.

Undefined (UND) – The processor enters Undefined mode if it experiences an "Invalid Instruction Code" fault or, in other words, it finds that the word fetched as an instruction is not one of the instruction codes defined for the core processor. In addition to detecting an illegal instruction, this mechanism is also used to discriminate between instructions intended for the core processor and, for example, floating point instructions intended for a floating-point coprocessor that shares the instruction stream from memory with the core processor. If the System-On-Chip has a floating-point coprocessor, the core processor will wait and the floating-point coprocessor will execute the floating-point instruction. If the SOC does not have a floating-point coprocessor, the Undefined mode exception handler can call a procedure that implements the floating point instruction with a sequence of core processor instructions.

Interrupt (IRQ) – If the IRQ Interrupt input on the processor is enabled and a hardware signal such as the End-of-Conversion strobe from an A/D converter asserts the IRQ input, the processor will switch to IRQ mode operation.

Fast Interrupt (FIQ) – If the Fast Interrupt (FIQ) input on the processor is enabled and a hardware signal such as a system power failure detector IC asserts the FIQ input, the processor will switch to FIQ mode operation.

Now that we have reviewed the ARM execution modes, the exceptions that cause the processor to enter one of the five exception modes, and the register set used for each mode, the next step is to discuss the Interrupt Vector Table that the processor uses to automatically call the interrupt procedure associated with each mode.

SETTING UP AND INITIALIZING AN INTERRUPT VECTOR TABLE

Figure 5-1 shows the address reserved for each of the exception vectors in the Interrupt Vector Table. Except for the FIQ vector at the highest address in the interrupt vector table, each table entry has only four bytes allocated. This is only enough for one 32-bit instruction. Therefore, the instruction stored in the Interrupt Vector Table for each exception type is usually some kind of unconditional branch to the start of the actual interrupt procedure. In this section we show you how an Interrupt Vector Table is set up and how the Table is initialized with instructions that branch to the desired exception procedures. Unless you are building a system from scratch, you will not usually have to do this because it is done by the system start-up code in the bootloader. However, you do need to understand how the Interrupt Vector Table is set up,

because you will have to access the table in order to install your own interrupt service procedures. An important note here is that when a system is first powered up or Reset, ROM memory containing the system startup code is located in memory starting at address 0x00000000 because this is the address the processors goes to get its first instruction after reset. However, the system startup code will usually copy the Interrupt Vector Table from ROM to some higher address in RAM. This copying to RAM is necessary so that, for example, the interrupt vector table can be modified as needed to install custom interrupt service procedures.

Exception	Mode	Vector Address
Reset	SVC	0x00000000
Undefined Instruction	UND	0x00000004
Software Interrupt	SVC	0x00000008
Abort (Instruction Fetch Fault)	ABT	0x0000000C
Abort (Data Fetch Fault)	ABT	0x00000010
Vector Reserved		0x00000014
IRQ Interrupt	IRQ	0x00000018
FIQ Interrupt	FIQ	0x0000001C

Figure 5-1 Interrupt Vector Table assignments for ARM-based processors.

The assembly language program in Figure 5-2 shows how to set up an Interrupt Vector Table and load each entry in the table with an instruction that will "call" the interrupt service procedure for that exception. In this example, the calls to the interrupt service are done with "LDR PC, address" instructions in the program block just after the _start: label. Note that, when the code for this program is loaded into memory, it is loaded starting at absolute address 0x00000000. This puts the LDR, PC instruction for each type of exception at the address shown for that exception in Figure 5-1.

When one of these LDR PC instructions executes, it copies the starting address of the desired interrupt service procedure from the literal pool explicitly declared in the second block of the program to the Program Counter. For example, when the LDR PC, IRQ_PROC_ADDR instruction executes, it will load the address from the literal pool location labeled IRQ_PROC_ADDR into the Program Counter. The statement IRQ_PROC_ADDR: .word IRQ_PROC in the second block of the program in Figure 5-2 declares a word type variable and initializes it with the starting address of the IRQ interrupt service procedure, assuming the name of the IRQ interrupt service procedure is IRQ_PROC as shown. As stated in the program comments, the address of the IRQ service procedure will be inserted in this location in the literal pool during the link stage when the program is built.

```
@ Assembly language instructions for setting up and initializing
@ the interrupt vector table for an ARM-based processor

@ The code for these instructions will be located starting at absolute
@   address 0x00000000 when the executable is built for a system with
@   ROM at address 0x00000000. This puts the LDR instructions in the
@   correct locations in the Interrupt Vector Table.

@ Note that this code explicitly declares and initializes a literal pool that
@   contains the addresses of the handler/interrupt procedures. When an LDR,
@   PC instruction executes, it will read the address of the specified
@ handler/interrupt procedure from the literal pool and load it in the PC.
@   The processor will then start fetching instructions from the
@   handler/interrupt procedure.

.text
.global _start

_start:

        LDR  PC, RESET_PROC_ADDR        @ Load PC with RESET_HANDLER address from
                                        @   literal pool
        LDR  PC, UND_HAND_ADDR          @ Load PC with UND_HANDLER address from
                                        @   literal pool
        LDR  PC, SWI_HAND_ADDR          @ Load PC with SWI HANDLER address
        LDR  PC, InstrABORT_HAND_ADDR   @ Load PC with InstrFetch Fault
                                        @   Handler address
        LDR  PC, DataABORT_HAND_ADDR    @ Load PC with DataFetch Fault
                                        @   Handler address
        NOP                             @ NOP instruction to fill space in table
                                        @   used for reserved vector (not used any longer)
        LDR  PC, IRQ_PROC_ADDR          @ Load PC with IRQ_PROCEDURE address from
                                        @   literal pool
        LDR  PC, FIQ_PROC_ADDR          @ Load PC with FIQ_PROC address from pool P

@ Note the names in the first column below identify the location in the literal
@   pool where the address for a specified exception is stored. The names to the
@   right of the DCDs are the names of the actual handlers/procedures. When the
@   program is "built", the actual addresses of the procedures will be inserted
@   in these locations in the pool to be read by the "LDR,PC" instructions.

RESET_PROC_ADDR:  .word RESET_HANDLER  @ Build inserts starting address of
                                       @   RESET_HANDLER procedure here.
UND_HAND_ADDR:    .word  UND_HANDLER   @ Build inserts UND_HANDLER address here.
SWI_HAND_ADDR:    .word  SWI_HANDLER   @ Build inserts SWI_HANDLER address here.
InstrABORT_HAND_ADDR: .word IABORT_HANDLER @ Build puts IABT_HANDLER addr here.
DataABORT_HAND_ADDR:  .word DABORT_HANDLER @ Build puts DABT_HANDLER addr here.
IRQ_PROC_ADDR:    .word  IRQ_PROCEDURE @ Build puts IRQ_PROCEDURE address here.
FIQ_PROC_ADDR:    .word  DCD  FIQ_PROCEDURE  @ Build puts FIQ_PROCEDURE address here.

.end
```

Figure 5-2. Assembly language program for setting up and initializing the interrupt vector table in an ARM based processor.

The key point here is that the Interrupt Vector Table contains LDR PC instructions. A particular LDR PC instruction will load the starting address of a desired interrupt service procedure from the literal pool into the PC and thereby send execution to that procedure. This indirect method of loading the address from a literal pool is necessary because, as explained in Chapter 3, a 32-bit address can only be loaded into a register such as the PC from another

register or from a memory location. We could have, for example, simply used an instruction of the form LDR PC,= RESET_HANDLER for the first entry in the Interrupt Vector Table and let the assembler create the literal pool to hold the address of the RESET_HANDLER procedure as we did for the LDR R1,= Multiplicand statement in the program in Figure 3-6. However, to make the literal pool operation clearer and more visible, we chose to explicitly declare and initialize it as shown in the second section of the program in Figure 5-2.

As we said earlier, the bootloader or other system level startup program sets up the Interrupt Vector Table and initializes it with the starting addresses of the specific exception procedures that are contained in the bootloader or system level program. The next step here is to show you how you "hook" the interrupt vector so that execution goes first to an interrupt service procedure you wrote for a particular interrupt type, instead of going directly to the bootloader or operating system procedure for that particular interrupt type. In your service procedure, you can then direct execution to the service procedure for your device, if it is requesting service, or pass execution on to the system level procedure, if your device is not requesting service. An example should make this very clear to you.

HOOKING AN INTERRUPT VECTOR AND CHAINING AN INTERRUPT PROCEDURE

The discussion in this section applies to any of the ARM exceptions but, because we will be using IRQ interrupts extensively throughout this chapter and the rest of the book, we will use the IRQ interrupt here. As we will show in detail later, hardware interrupts from many sources are ORed and the resulting signal is connected to the IRQ input on the processor. When an interrupt signal on one of the inputs asserts the IRQ input on the processor, the processor will use the interrupt vector table to send execution to a system level IRQ procedure. The system level procedure determines the source of the interrupt by reading the status register in each of the connected devices to find out which device is requesting service. The system level procedure then calls the interrupt procedure for the first requesting device that it finds.

If you want to add a new IRQ source to the system, you have two choices for getting the processor to respond to IRQ signals from this new source and execute the service procedure for it. One way is to modify the system program by adding your IRQ source to the section of the system program that determines which IRQ source is requesting service and calls the procedure corresponding to that source. This method usually requires rebuilding the entire system level program.

The second way to read the system level IRQ address from the literal pool, save it in memory, and write the address for a program you write in the IRQ location in the literal pool. When any device generates an IRQ interrupt, execution will then go to your program instead of to the system IRQ procedure. In your IRQ procedure you determine if the IRQ was caused by your device. If it was, you execute your procedure to service the device and return execution to the interrupted program. If the IRQ request was not caused by your device, you put the saved address of the system level IRQ procedure in the PC to pass execution to the system level IRQ procedure. Installing the address of your IRQ decision procedure in place of the address for the system level IRQ decision procedure in the literal pool is referred to as *"hooking" the interrupt vector*. Setting up a decision structure to determine the source of an interrupt and direct execution to one of several procedures is called *"chaining" interrupt procedures*. A simple example should help you understand how you actually do the hooking and chaining process.

For this first example, suppose that you have a button connected to an interrupt input on an ARM-based microprocessor and that the signal from this interrupt is routed through the processor's internal interrupt controller to the IRQ input on the processor along with other interrupt signals. Furthermore, assume that you want the interrupt service procedure for this button interrupt to increment the value of a variable called COUNT and display the count each time the user presses the button. For simplicity, also assume that existing program is the bootloader or some other system level program that is running in Supervisor mode or System mode so you can access the CPSR.

To install this interrupt service procedure in the bootloader, you create a procedure, which we call INT_DIRECTOR, that decides whether to send execution to your button service procedure or to the system IRQ procedure. Figure 5-3 shows in flowchart form how the system responds to an IRQ request without our INT_DIRECTOR and with it. Without the INT_Director procedure, execution simply goes to the bootloader IRQ procedure. With our INT_DIRECTOR procedure, execution goes to the INT_DIRECTOR procedure that determines if the IRQ request came from our button or from one of the system IRQ sources. If the IRQ interrupt was caused by the button having been pushed,, the BUTTON_SVC procedure is called to do the desired actions for the button interrupt and then return to the interrupted program. If the IRQ request was not caused by the button having been pushed, then execution is simply passed on to the system level IRQ procedure. The system level IRQ procedure will return execution to the mainline program when it finishes, as shown. You can easily extend the chain of procedures in the INT_DIRECTOR procedure by simply adding more decision boxes to the flowchart in Figure 5-3 and adding the corresponding conditional branches to the INT_DIRECTOR procedure. Now that you have an overview of the INT_DIRECTOR procedure, let's take a look at the actual assembly language code for it.

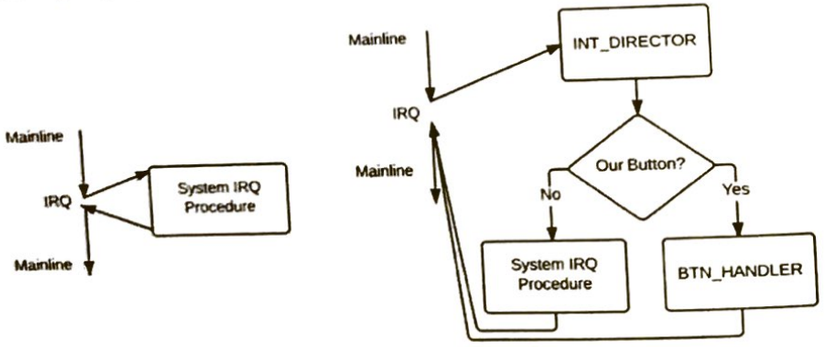


Figure 5-3. Execution flow diagram without and with our Interrupt Director procedure. (Improved artwork supplied by Eric Krause.)

The assembly language program in Figure 5-4 shows how you hook the IRQ vector to send execution to an INT_DIRECTOR procedure. The INT_DIRECTOR procedure chains the BUTTON_SVC procedure and the system level IRQ procedure. Specifically, the INT_DIRECTOR procedure sends execution to the BUTTON_SVC procedure if an interrupt signal was produced by the button being pushed and sends execution to the system level IRQ

procedure if no interrupt is pending from the button being pushed. You will use the basic structure of this example program as a template for installing interrupt procedures, so to give you a solid understanding let's work though it step by step.

```
@  PROGRAM TO SHOW HOOKING OF INTERRUPT VECTOR, CHAINING OF INTERRUPT PROCEDURES,
@ AND SERVICING AN INTERRUPT

.text
.global _start

_start:
@ HOOK IRQ PROCEDURE ADDRESS AND INSTALL OUR INT_HANDLER ADDRESS
                                @ NOTE:INTERRUPT VECTOR TABLE INITIALIZED BY
                                @ BOOTLOADER WITH LDR, PC INSTRUCTIONS THAT
                                @ FUNCTION AS UNCONDITIONAL BRANCHES
        MOV R1, #0x18           @ LOAD IRQ INTERRUPT VECTOR ADDRESS 0x18
        LDR R2, [R1]            @ READ INSTR FROM INTERRUPT VECTOR TABLE AT 0x18
                                @ THIS INSTRUCTION IS AN LDR PC,ADDRESS INSTRUCTION

                                @ THAT READS THE ABSOLUTE ADDRESS FOR THE IRQ
                                @ PROCEDURE FROM A LITERAL POOL, USING THE 12-BIT
                                @ OFFSET CONTAINED IN THE 12 LSB OF THE INSTRUCTION.
                                @ THE CODE FOR THE INSTRUCTION IS 0xE59FFXXX, WHERE
                                @ XXX IS THE OFFSET INTO THE LITERAL POOL
                                @     FROM THE CURRENT PC
        LDR R3, =0xFFF          @ CONSTRUCT MASK
        AND R2, R2,R3           @ MASK ALL BUT OFFSET PART OF INSTRUCTION
        ADD R2, R2,#0x20        @ ABSOLUTE ADDRESS OF IRQ PROCEDURE IN LITERAL POOL
                                @ = 0x18 + 8-BYTE PIPELINE ADJUSTMENT + OFFSET IN R2
                                @ ADJUSTMENT REQUIRED BECAUSE PC IS 2 WORDS OR
                                @ 8 BYTES ABOVE 0x18 WHEN OFFSET IN INSTRUCTION
                                @ ADDED TO PC DURING EX STAGE OF LDR PC INSTRUCTION.
        LDR R3,[R2]             @ READ BTLDR IRQ ADDRESS FROM LITERAL POOL
        STR R3, BTLDR_IRQ_ADDRESS @ SAVE BTLDR IRQ ADDRESS FOR USE AT END OF CHAIN IN
                                @    IRQ_DIRECTOR
        LDR R0,=INT_DIRECTOR    @ LOAD ABSOLUTE ADDRESS OF OUR INTERRUPT DIRECTOR
        STR R0,[R2]             @ STORE THIS ADDRESS IN LITERAL POOL WHERE LDR PC
                                @ INSTRUCTION AT 0x18 GOES TO GET ADDRESS FOR
                                @ HANDLER IN RESPONSE TO IRQ INTERRUPT

@ MAKE SURE IRQ INTERRUPT ON PROCESSOR ENABLED BY CLEARING BIT 7 IN CPSR
        MRS R3, CPSR            @ COPY CPSR TO R3
        BIC R3, #0x80           @ CLEAR BIT 7 (IRQ ENABLE BIT)
        MSR CPSR_C, R3          @ WRITE BACK TO LOWEST 8 BITS OF CPSR

LOOP:   NOP
        B  LOOP                 @  WAIT FOR INTERRUPT HERE (SIMULATE MAINLINE
                                @  PROGRAM EXECUTION)

INT_DIRECTOR                    @ CHAINS INTERRUPT PROCEDURES
        STMFD  SP!, {R0,R1,LR}  @ SAVE REGISTERS TO BE USED IN PROCEDURE ON STACK
        LDR R0, =INTPND         @ POINT AT INTPND REGISTER
        LDR R1, [R0]            @ READ INTPND REGISTER
        TST R1, #0x0001         @ CHECK IF BUTTON PUSHED INTERRUPT
        BNE   BUTTON_SVC        @ YES, GO SERVICE - RETURN TO WAIT LOOP FROM SVC
        LDMFD  SP!,{R0-R3,LR}   @ NO, MUST BE BOOTLOADER IRQ, RESTORE REGISTERS
        LDR  PC,BTLDR_IRQ_ADDRESS @ GO TO BOOTLOADER IRQ SERVICE PROCEDURE.
BUTTON_SVC:                     @  BOOTLOADER WILL USE STORED LR TO RETURN TO LOOP
        MOV R1,#0x00000001      @ VALUE TO CLEAR BUTTON IRQ BIT IN INTPND REGISTER
        STR  R1, [R0]           @ WRITE BACK TO INTPND REGISTER
        LDR  R1, =COUNTER       @ POINT TO COUNTER VARIABLE IN MEMORY
        LDR  R0, [R1]           @ LOAD COUNTER VALUE IN R0
        ADD R0, R0, #1          @ INCREMENT  COUNT BY 1
        STR  R0, [R1]           @ WRITE NEW COUNTER VALUE BACK IN MEMORY
        BL  DISPLAY             @ GO DISPLAY NEW COUNTER VALUE
        LDMFD  SP!,{R0,R1,LR}   @ RESTORE REGISTERS
```

```
        SUBS  PC, LR, #4                @ RETURN FROM INTERRUPT (TO WAIT LOOP ABOVE)
DISPLAY:                                @ NOTE: CALLED WITH COUNTER VALUE PASSED IN R0
                                        @   DISPLAY PROCEDURE INSTRUCTIONS HERE
        MOV PC, LR                      @ RETURN TO  BUTTON_SVC  USING LR VALUE
                                        @ SAVED BY "BL DISPLAY" INSTRUCTION IN BUTTON_SVC
BTLDR_IRQ_ADDRESS  WORD 0x0             @ SPACE TO STORE BOOTLOADER IRQ ADDRESS
.data
COUNTER .word 0x0                       @ COUNTER VARIABLE FOR BUTTON_SVC PROCEDURE
.end
```

Figure 5–4. Assembly language program to show hooking an interrupt vector and chaining an interrupt procedure.

For the hooking part of the process, we start by reading the code for the IRQ LDR PC instruction from the interrupt vector table at address 0x00000018 into R2. The lowest 12 bits of this instruction contain the offset from the address in the Program Counter to the address in the literal pool where the system level IRQ procedure address is stored. We isolate these 12 bits by ANDing the instruction code in R2 with 0xFFF. Then in R2, we construct the absolute address of the literal pool location we want to read by adding 0x20 to this offset. The reason we add 0x20 instead of just 0x18 to the offset from the instruction is that the assembler calculates the offset not from 0x18, the PC value for the LDR instruction, but from the value that will be in the PC when the offset is actually added to the PC value. Remember that, for an ARM processor, this addition is done during the EX stage of the pipeline. By the time the LDR instruction reaches the EX stage, the processor has done two more instruction fetches and the program counter has therefore been incremented by 4 twice, first to 0x1C and then to 0x20. When the LDR PC instruction executes, it computes the literal pool location by adding the offset contained in the lowest 12 bits of the instruction to 0x20 rather than to 0x18. The ADD R2, R2, #0x20 instruction in our program does the same addition and thus points R2 at the location of the system-level IRQ procedure address in the literal pool. The LDR R3, [R2] instruction then reads the system-level IRQ address from this location in the literal pool and the STR R3, BTLDR_IRQ_ADDRESS instruction stores this address in a memory location declared at the end of the program in the .text area to save this address. Note that you put the space for BTLDR_IRQ_ADDRESS in the .text section, so it can be accessed with PC relative addressing without needing a register reference. Remember that items in the .data section must be accessed with an address in a register. As you will see that in a little later case we don't want to use any registers to refer to this location. The only consideration with putting the space for the BTLDR_IRQ_ADDRESS in the .text section is that the .text section containing this location will have to be loaded into RAM, so the address can be written to the location.

In the INT_DIRECTOR procedure we will use this stored address to chain the system level IRQ procedure with our BUTTON_SVC procedure. As the last step in the hooking process, we load R0 with the starting address of our INT_DIRECTOR procedure and store this address in the literal pool where the IRQ LDR PC instruction will go to get the IRQ procedure address.

The next step in the mainline program here is to make sure the IRQ interrupt input on the processor is enabled. To do this, you read the CPSR into R3 with the special MRS R3, CPSR

instruction, clear the IRQ bit in the status word to enable IRQ, and then write the resultant word back to the CPSR with the MSR CPSR_C, R3 instruction. The underscore C on the MRS instruction indicates that the instruction should be coded to modify only the lower eight bits in the CPSR. Finally in the mainline program, you simply execute an endless program loop until an interrupt occurs. This loop essentially functions as a Wait for Interrupt instruction. Now let's look at how the INT_DIRECTOR procedure chains our BUTTON_SVC procedure with the system level IRQ procedure.

In the INT_DIRECTOR procedure, we start by pushing the registers that will be used in the INT_DIRECTOR procedure on the stack. Note that as part of this, we save R14 which contains the return address that was put there as part of the processor's automatic response to an IRQ. It is very important to do this if you call any sub-procedures from the interrupt service procedure. If this R14 value is not saved, it will be written over with a new return address, when you use a BL instruction to call a procedure from the interrupt service procedure. Remember that during any procedure call with BL, the return address is automatically stored in R14. The point is that if the initial value saved in R14 is written over by a BL instruction, it will be lost forever and execution will never be able to use it to return to the mainline program at the end of the interrupt service procedure.

Next in the INT_DIRECTOR procedure we load a pointer to an Interrupt Pending Register, INTPND, in the processor's internal Interrupt Controller. Assume for this first example that each bit in the INTPND register is connected to one of the possible interrupts sources in the system. The bit that relates to a particular interrupt source will be set if that device is asserting its interrupt signal. All you have to do to determine if a particular device is requesting service is to read in the INTPND register and check if the bit that corresponds to that device is a 1. (Several of the ARM-based processors have simple INTPND registers that function in this way, so we use it for this first example. The AM3358 processor has a more complex interrupt mechanism that we will discuss later.) For this example, assume that the button is connected such that the least significant bit of the INTPND register will be a 1 when the button is pushed. The LDR R1, [R0] instruction reads the value from the INTPND register into R1 and the TST R1, #0x0001 instruction ANDs the value in R1 with 0x0001 to check if the least significant bit is a 1. If the result of the ANDing is not equal to all zeros in R1, because the button bit in the INTPND register was a 1, the BNE BUTTON_SVC instruction will cause execution to go to the BUTTON_SVC code block. If the bit that corresponds to the button interrupt signal was not set in the INTPND register, the button was not pushed then execution will simply fall through to the LDMFD R13!,{R0-R1, R14} instruction that restores the saved registers. The LDR PC, BTLDR_IRQ_ADDRESS instruction then sends execution to the bootloader IRQ procedure by loading the bootloader IRQ address that we saved during the hooking process into the PC. Note that since we put the storage location for BTLDR_IRQ_ADDRESS in the .text section of the program, the LDR PC, BTLDR_IRQ_ADDRESS instruction can read the address directly from the memory location without using a register to access it. Using a register to access it would leave one register modified when execution was passed on to the Bootloader IRQ procedure. This would violate the basic rule that our INT_DIRECTOR procedure should pass execution with all registers the same as they were when the procedure was entered. When execution goes to the Bootloader IRQ procedure, that procedure will save R14 as we did at the start of INT_DIRECTOR and use this value to return execution to the mainline program when it finishes. Now let's look at the BUTTON_SVC procedure.

The first step in the BUTTON_SVC procedure section of the program would be to push any additional registers that are used in BUTTON_SVC procedure but were not pushed at the start of the INT_DIRECTOR procedure. It is the responsibility of any interrupt procedure to save all registers used, so that execution can be correctly returned to the interrupted program. Since the BUTTON_SVC code block only uses R0 and R1 and these were already saved at the start of INT_DIRECTOR, no additional pushes are required. Note that for this first example interrupt procedure, we assume that IRQ remains disabled while the IRQ service procedure is executing. The result of this is that another device asserting an IRQ signal cannot be recognized until the current service is completed and execution returns to the interrupted program. Later in the chapter we show you the additional code you have to add to the INT_DIRECTOR procedure, so that an IRQ procedure could be interrupted by a higher priority IRQ interrupt and still work correctly.

The second step in the BUTTON_SVC procedure is to clear the bit in the INTPND register that corresponds to the button interrupt, so that it will not cause another IRQ interrupt when we return to the mainline program at the end of the BUTTON_SVC procedure. For the simple interrupt controller we are using for this example, let's assume that you do this by writing a 1 to the bit that corresponds to the button interrupt in the INTPND register. Since R0 still contains the address of the INTPND register, the two instructions MOV R1, #0x001 and STR, R1, [R0] do this.

Next we read the COUNTER variable from memory, increment it by 1, write the new value back to memory, and call a procedure to DISPLAY the new count value. Note that, as stated earlier, if we had not pushed R14 on the stack at the start of the INT_DIRECTOR procedure, R14 would be written over when the BL DISPLAY instruction writes the return address for this call in R14. The DISPLAY procedure, of course, uses the R14 value inserted during execution of the BL instruction to return to the BUTTON_SVC procedure with the MOV PC,LR instruction at the end of the DISPLAY procedure.

Finally in the BUTTON_SVC procedure, we use the LDMFD R13!, {R0-R1, R14} instruction to pop the saved registers, including R14, off the stack. Then we use the SUBS PC, LR, #4 instruction to return execution to the mainline program. We will leave the discussion of this final instruction for the next section, where we show and explain the special instructions that are used to return from the different types of ARM interrupt/exception procedures. However, note that as shown by the arrow from the bottom of the BUTTON_SVC block in Figure 5-4 and the instructions in Figure 5-3, we return from end of the BUTTON_SVC code block directly to the interrupted program, rather than back to the INT_DIRECTOR code section. This is the standard way to do it.

RETURNING FROM INTERRUPT PROCEDURES

As shown in the list of actions that an ARM-based processor does in response to an exception, the processor saves the PC value for the interrupted program in R14 of the new mode and saves the CPSR of the interrupted program in the SPSR of the new mode. In order to return execution correctly to the interrupted program, both the PC and the CPSR must be restored in one operation, rather than individually. If the processor just used the value in the exception mode R14 to return execution to the interrupted program, execution would return to the interrupted program and not be able to access the SPSR in the exception mode to restore the CPSR of the interrupted program. Likewise, if the processor first restored the interrupted program CPSR from

the SPSR in the exception mode, execution would be switched to the mode of the interrupted program and the return address in R14 of the exception mode would not be accessible. To solve this problem, the designers of the ARM cores included special instructions that restore CPSR and load the PC with the return address from R14 in one operation.

To return from an IRQ, FIQ, or Instruction Fetch Fault, you use the special SUBS PC, LR, #4 instruction. This instruction will subtract 4 from the return address stored in the exception mode R14 and load it into the PC. The reason you have to subtract 4 from the stored return address is that the return address stored in R14 is the value that was in the PC during the EX stage of the pipeline, when the processor actually responded to the exception. If you refer to the pipeline diagram in Figure 2-6b, you can see that when the instruction where the exception is acknowledged is in its EX stage, two more instruction fetches will have been done and the value in the PC will be equal to the PC value of the interrupted instruction + 8. This then is the value of the PC that will be saved in R14 of the exception mode. For these three types of exceptions, the processor will finish the instruction that was in its EX stage when the exception was acknowledged, so we want execution to return to the next instruction after the one in which the interrupt was acknowledged. The address of this instruction will be four less than the value stored in R14. Therefore, to return to the desired instruction, you just subtract four from the value stored in R14 and load the result into the PC for the return with the SUBS PC, R14, #4 instruction. Furthermore, when the PC is specified as the destination operand in the SUBS instruction, the "S" on the mnemonic tells the assembler to code the instruction such that the CPSR is automatically restored from the SPSR of the exception mode at the same time that the return value in R14 is loaded into the PC. These two operations must be done at the same time, in order for execution to return to the interrupted program with CPSR and PC correct. If IRQ and FIQ interrupts were enabled by bits 6 and 7 in the CPSR being cleared before the exception occurred, they will automatically be re-enabled when the CPSR of the interrupted program is restored.

To return from a SWI or Undefined Instruction exception, you use a MOVS PC, R14 instruction. These exceptions are recognized during the Instruction Decode (ID) stage of the pipeline. Again referring to the pipeline diagram in Figure 2-6b, you can see that when a SWI or undefined instruction is in its ID stage, the next instruction is in its Instruction Fetch stage and the PC contains the address of this instruction. When the processor responds to one of these exceptions, the PC value of this next instruction will be stored as the return address in R14 of the mode for that exception. Since this stored value is the correct address for the instruction that we want to return to after the SWI or Undefined Instruction exception procedure, we do not need to subtract 4 from the value as we did for the IRQ, FIQ, and Instruction Fetch Fault exceptions.

To return from a Data Abort exception, you use the SUBS PC, LR, #8 instruction. The reason that you subtract 8 instead of subtracting 4 is that you want go back and retry the instruction that caused the data fault. Since the value stored in R14 for this exception is the PC for the faulting instruction + 8, you have to subtract 8 to get the PC back to the value that will cause the faulting instruction to be fetched and executed again.

Now that we have given you the framework for hooking and chaining interrupt procedures, an example of a very simple interrupt service procedure, and the instructions you use to return from the different types of exceptions, the next step is to describe in more detail how you write interrupt procedures and other procedures such that they will always perform correctly in a system that uses interrupts extensively.

GUIDELINES FOR WRITING CORRECTLY BEHAVING INTERRUPT PROCEDURES

As we will discuss in depth in later chapters, all but the simplest of the current microprocessor systems do most of their work on an *interrupt-driven* basis. In other words, they loop in a *system idle process* such as the simple endless loop in the program in Figure 5-3, service device and system requests on an interrupt basis, and return to the system idle process after each service to wait for the next request. For your interrupt service procedures and your other procedures to work correctly in this type of system, you have to follow some simple guidelines. To start, we show and discuss the guidelines you should follow when writing interrupt service procedures.

In every interrupt procedure, save ALL of the registers used in the procedure, including R14, on the stack. This is very important because an interrupt can occur at any point in a program. In order to return correctly to the interrupted program, you must save all of the *state* or, in other words, the registers and flags for the interrupted program, so the state can be correctly restored at the end of the service procedure. The flags in the CPSR of the interrupted program are automatically saved when the CPSR of the interrupted program is copied to the SPSR of the exception mode, but you have to specifically save the other registers on the stack at the start of the interrupt procedure. Remember from the discussion in an earlier section and from the IRQ_DIRECTOR procedure in Figure 5-3 that you have to make sure and save R14, so that it will not be written over and lost if you do a Branch and Link to call another procedure in your interrupt service procedure. Note that the Arm Procedure Call Standard we discussed in Chapter 4 for interfacing with C programs does not apply to interrupt procedures.

In a system where you want nested interrupt procedures or, in other words, you want an interrupt to be able to interrupt an executing interrupt procedure, you have to do some additional tasks and save both SPSR and R14 in the interrupted mode. We will show and discuss these tasks in a later section. Next here, we discuss the guidelines for writing general-purpose procedures so that they function correctly in an interrupt-driven system.

RE-ENTRANT PROCEDURES IN AN INTERRUPT-DRIVEN SYSTEM

Procedures that will be used in an interrupt-driven system should be written so that they are *re-entrant*. A re-entrant procedure is one that can be called, interrupted before it is compete, called again by an interrupt procedure, complete that call correctly, return to the interrupted execution of the procedure and complete it correctly, and then return correctly to the program that initially called the procedure. A real example and the diagram in Figure 5-5 should help you understand this term and the need for a procedure to be re-entrant.

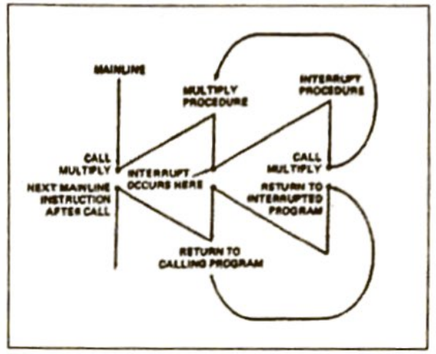


Figure 5-5. Program execution flow needed for re-entrant multiply procedure.

For this example, assume a pressure sensor on a ship's steam boiler is connected to the FIQ input on ARM-based processor. If the steam pressure goes too high the sensor will assert the FIQ interrupt input on the processor. In response to the FIQ interrupt signal, the processor will automatically stop executing the current program, switch to the FIQ mode, and branch to a procedure that will take the actions required to reduce the pressure and hopefully prevent the boiler from blowing up. The interrupt service procedure might, for example, open a valve to quickly reduce the pressure and then reduce the fuel flow to the burner. After the determining that the pressure had returned to a reasonable value, the interrupt service procedure would return execution to the interrupted program.

Further assume for this discussion that the processor has called the Multo multiply procedure shown in Figure 4-22 and is in the middle of executing that procedure when the FIQ interrupt occurs as shown in the Multo Procedure column of Figure 5-5. If the FIQ input is enabled, the processor will automatically call the interrupt procedure that has been written to service the FIQ interrupt as shown by the arrow to the Interrupt Procedure column in Figure 5-5. Now, further suppose that the interrupt procedure needs to use the Multo procedure to calculate how far to open the valve in order to reduce the steam pressure by the correct amount. To call the Multo procedure, the interrupt service procedure does a Branch and Link (BL) to the Multiply procedure as shown by the arrow looped over the top to the Multo procedure in Figure 5-5. A return instruction at the end of the Multo procedure returns execution to the interrupt procedure at the next instruction after the BL instruction that called the Multo procedure this time. This is shown by the arrow looped from the bottom of the Multo to the interrupt procedure in Figure 5-5. At the end of the interrupt service procedure, a SUBS, PC, LR, #4 instruction will return execution from the interrupt procedure to the instruction after the interrupted instruction in the first execution of the Multo procedure as shown by the straight arrow from the bottom of the interrupt procedure to Multo in Figure 5-5. This time, at the end of the Multo procedure, the LDMFD R13!, {R6-R8, PC} instruction at the end of the Multo procedure returns execution to the next mainline instruction after the BL instruction that initially called the Multo procedure. This is shown by the arrow to the left from the bottom of the Multo procedure in Figure 5-5. The point here is that the Multo procedure must be written so that it is re-entrant, in order for it to work correctly in this type of system.

Furthermore, re-entrancy is very important in current multi-tasking systems that switch from one task or one thread to another on a time-slice basis. Suppose for example, that in a multi-

tasking system, one task has called Multo and is in the middle of running the Multo procedure. Further suppose that the time slice for this task runs out and execution switches to another task that calls Multo. In order for both tasks to produce the correct results, the Multo procedure must be reentrant and the system program that switches from one task to another must carefully save the state of each task when it switches to another task. We will discuss multitasking much more in later chapters. However, we will plant some early seeds here by giving you guidelines for maintaining re-entrancy in a simple interrupt flow such as that shown in Figure 5-5. Since all but the very simplest systems you design will be multi-tasking and will use interrupts extensively, it is important for you to learn and follow these guidelines from now on as you write procedures.

The key points for maintaining re-entrancy and correct execution flow for a general purpose procedure in an interrupt-driven system are as follows:

1. In all interrupt procedures, as discussed in the preceding section, push **all** registers that are used in the interrupt service procedure, not just the R4-R10 registers specified by the APCS. If, for example, you were calling the Multo procedure in Figure 4-21 from an interrupt service procedure, you would want to push R0-R3 before loading the pointers in R0-R3 for the interrupt service procedure call to Multo. If this is not done, the R0-R3 register values being used in the first call to Multo will be lost. You then restore the saved values of R0-R3 at the end of the interrupt service procedure, before returning to the Multo procedure to complete the first call of Multo. If other registers such as R11 and R12 were being used as scratchpad registers in the interrupted procedure, as allowed by the APCS, you would also want to save these on the stack in the interrupt procedure and restore them at the end of the interrupt procedure. This will assure that these registers have the correct values upon return to the interrupted procedure.

   The guiding principle here is that, since the interrupt can occur at any point in Multo, you must in the interrupt service procedure save everything needed to successfully complete the first call of Multo, regardless where the interrupt occurs in Multo.

   Note that when an ARM-based processor responds to an interrupt, the CPSR of the interrupted program is automatically saved in the SPSR of the interrupt mode that that processor switches to in response to the interrupt. Since the flags for the interrupted program are contained in the CPSR, the flags are then automatically saved during the interrupt response process and restored by the special instruction at the end of the interrupt service procedure. This means that in your interrupt service procedures you do not need to insert specific instructions to save the flags except in the case where an interrupt of the same type can interrupt an executing interrupt procedure. In a later section we show you how to deal with this situation.

2. In the Multo procedure, push all registers used in the procedure as specified by the APCS, so that execution returns to the original calling program with the correct state at the end of the procedure. As an example of this, the Multo procedure pushes R6-R8 and R14. Although we did not actually need to push R14 for the simple Multo procedure, we did so for two reasons. The first reason was so that we could use the single LDMFD R13!, {R6-R8, PC} instruction to both restore registers and return to the calling program at the end of the procedure. The second reason was so that, if we decided to call a sub-procedure from Multo, R14 would already be saved. Remember from previous discussions that when you call a sub-procedure from another procedure, you have to first

save R14 on the stack so that it is not written over when you do a BL to call the sub-procedure. We did not push R0-R3 in the Multo procedure because these values will be changing as the Multo procedure executes and APCS does not require that these be saved for the calling program. As explained in step 1 above, it is the responsibility of the interrupt procedure to save ALL of the registers used in the interrupt procedure before calling Multo again and to restore the values of these registers before returning to the first execution of Multo. It is the responsibility of Multo to preserve and restore the registers for the calling program.

3. Pass pointers and single parameters to procedures in registers or on the stack, so that access to these parameters can be saved in the interrupt procedure and restored when the procedure is re-entered as described in steps 1 and 2. The alternative to passing single parameters and pointers in registers is to simply access variables by name in the procedure. In the Multo procedure, for example, instead of loading pointers to the arrays into registers and passing them to the procedure when we call the procedure, we could have just moved the instructions that load the pointers in R0, R1, and R2 into the procedure and thus loaded pointers to the named arrays directly in the procedure. If we had done this, the procedure would only work if the data to be processed were in the arrays named MULTIPLICANDS and MULTIPLIERS and the result put in the array PRODUCTS.

   The first problem with this approach is that it would make the Multo procedure *non-portable*. The procedure could only be called to operate on the specifically named arrays rather than on any desired arrays of half-words that you might want to multiply.

   The second problem with this approach is that it would make the Multo procedure non-reentrant. If the interrupt service procedure writes new values in the MULTIPLICANS and MULTIPLIERS arrays before calling Multo, the values from the first call of Multo will be overwritten and lost. (Yes, just as you could probably figure out a way to put toothpaste back in the tube, you could probably figure out a way to save the MULTIPLICANDS, MULTIPLIERS, and PRODUCTS arrays on the stack in the interrupt procedure before loading new values in the arrays and calling Multo. However, it is much better to completely avoid the problem by simply passing variable and array pointers to the procedure in registers as shown in the program in Figure 4-22, instead of accessing the variables by name in the procedure.)

To briefly summarize this section and the previous section, the key points for writing procedures for an interrupt-driven environment are:

1. In the interrupt procedure, save everything used
2. In a general purpose procedure, save ALL registers used in the procedure at the start of the procedure and restore them before returning.
3. In calling a general-purpose procedure, pass parameters in registers or on the stack
4. In a general-purpose procedure, do not use named variables to access data.

Next, before we talk about how you write "nested" interrupt procedures, we will take a brief side trip to discuss and illustrate the meaning of the term "recursive procedure," which is often confused with the term "re-entrant procedure."

RECURSIVE PROCEDURES

The term *recursive procedure* is often confused with the term *re-entrant procedure* that we discussed in the preceding section. A recursive procedure is a procedure that can call itself. This seems simple enough but the question you may be thinking is, "Why would we want a procedure to call itself?" The answer is that certain types of problems, such as choosing the next move in a computer chess program or the best solution for a logic optimization problem, can be most efficiently solved with a recursive procedure. Recursive procedures are commonly used to work with complex data structures called *trees*.

Except for those cases where we need the additional speed gained by writing in assembly language, we almost always write a recursive procedure in a high-level language. However, to make sure you understand the difference between re-entrant and recursive, we will give a short example of how a recursive program works. Most of the examples of recursive programs that we could think of were too complex to show here. Therefore, we have chosen a simple problem that can actually be solved more efficiently without recursion.

The example problem we have chosen to solve is to compute the factorial of a given number. To refresh your memory, the factorial of a number is the product of the number and the positive integers less than the number. For example, 5 factorial or 5! is equal to $5 \times 4 \times 3 \times 2 \times 1$, or 120. The factorial for a given number can be computed by a recursive procedure that calculates the factorial for a number passed to it and returns the calculated factorial. Given a passed number n, the basic algorithm for a factorial procedure that we call FACTO can be expressed as:

IF n = 1 THEN factorial = 1
ELSE factorial = n x factorial (n-1)

This says that if the number we pass is 1, the procedure should return the factorial of 1, which is 1 as shown in Figure 5-6a. If the number we pass is not 1, the procedure should multiply the number by the factorial of (the number –1). Now here's where the recursion comes in. Suppose we pass a 3 to the procedure. When the procedure is first called, it has the value of 3 that it needs for the multiplication but it does not have the value for the factorial of n-1. The procedure solves this problem by calling itself to compute the value of (3-1)! or, in other words, 2!. The factorial procedure calls itself over and over until the factorial of n-1 that it needs is the factorial of 1 as shown in Figure 5-6b. Figure 5-6c shows a more detailed algorithm for the actions in the FACTO procedure. To stick the concept of recursion in your mind and to give you more practice in passing parameters and saving state, we leave it to you as exercise at the end of the chapter to translate this simple recursive algorithm into ARM assembly language. As you do this, you should review the guidelines for making a procedure re-entrant, because a recursive procedure must be re-entrant.

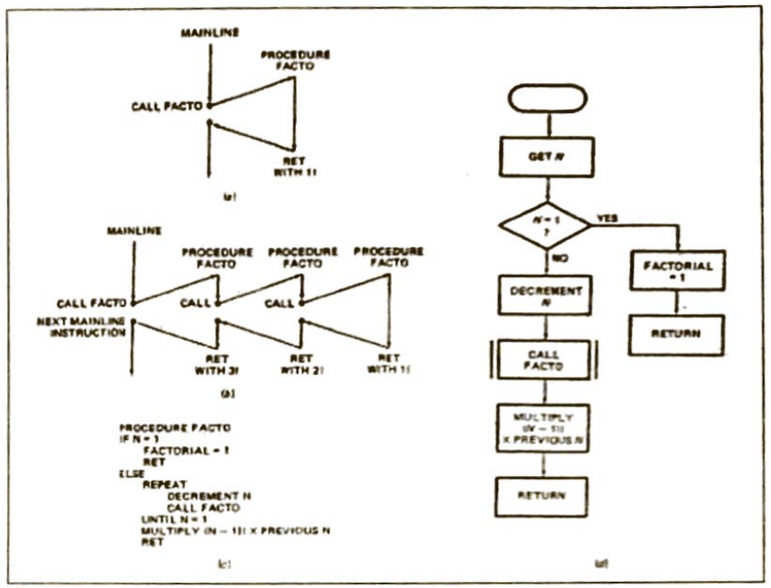


Figure 5-6. Algorithm for program to computer factorial of a number, N. (a)flow diagram for N=1. (b) flow diagram for N=3. (c) Pseudocode.(d) Flowchart.

WRITING NESTED INTERRUPT PROCEDURES

Now that you have learned how to write simple interrupt procedures and general-purpose re-entrant procedures, the next step is to give you an overview of how to write interrupt procedures that can be interrupted by a higher priority interrupt signal of the same type and still function correctly when execution returns from the higher priority interrupt service procedure to the lower priority interrupt service procedure. This is a very common situation because interrupts from many different sources are likely all directed through the Interrupt Controller to the IRQ input on the processor and these interrupts have different priorities. A high priority interrupt is one that must be serviced with low latency time or, in other words, very soon after it occurs. To minimize the latency for high priority interrupts, a high priority interrupt must be allowed to interrupt a lower priority interrupt service procedure. Suppose, for example, that a 10 Gb/sec Ethernet Controller interrupt signal and a user button service interrupt signal are both directed to the IRQ input on the processor. Assuming that the Ethernet Controller interrupt is higher priority than the user button service interrupt and that the user button service procedure is lengthy, you certainly want the Ethernet Controller signal to be able to interrupt execution of the user button service procedure. In order for a higher priority IRQ to interrupt a lower priority IRQ procedure as shown in Figure 5-7 and have both IRQ procedures complete correctly, you have to do three additional tasks in the lower priority interrupt procedure as follows:

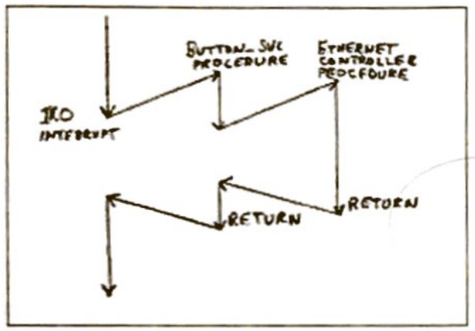


Figure 5-7. Flow diagram for nested IRQ procedures.

1. **You have to mask lower priority interrupts, so that when you re-enable interrupts to allow a higher priority interrupt to request service, only a higher priority interrupt interrupt request can be recognized but lower priority interrupts are not recognized.. The exact method for doing this depends on the specific ARM processor you are working with. The AM335X processor Interrupt Controller has a mechanism for setting the priority of an interrupt, so that only a higher priority interrupt can interrupt it.**

2. **You have to push the Saved Program Status Register (SPSR) on the stack, so it will not be lost when the processor responds to the higher priority IRQ signal.** When the processor responds to the first IRQ, it saves the CPSR of the interrupted program in the SPSR of the IRQ mode. If the first IRQ procedure cannot be interrupted by another IRQ response, then the CPSR saved in the SPSR will just be restored to the CPSR in the interrupted mode by a SUBS PC, LR, #4 instruction at the end of the IRQ procedure. However, if the IRQ input is re-enabled in the first IRQ procedure and another IRQ occurs, the CPSR of the interrupted IRQ procedure will automatically be saved in the IRQ mode SPSR. The problem here is that the IRQ mode SPSR already holds the CPSR from the original interrupted program. If you allow this to be written over during the second interrupt response, it will be lost and will not be available for restoring the CPSR of the program that was interrupted by the first IRQ response. The solution for this problem is to save the IRQ SPSR on the stack in the first IRQ procedure before re-enabling the IRQ interrupt input to allow another IRQ response. The first three instructions in the IRQ procedure in Figure 5-8 show you how to do this.

   After saving R0-R3 and R14 on the stack with the STMFD R13!, {R0-R3, R14} instruction as in previous examples, we copy the SPSR register to R3 with Move Status Register to Register instruction, MRS, R3, SPSR and save the value in R3 on the stack with the STMFD R13!, {R3} instruction. Note that since the processor switches to the IRQ mode R13 when it responds to the IRQ interrupt, the registers saved at the start of an IRQ procedure will be saved on the IRQ stack pointed to by the IRQ R13. (In a system with a bootloader, this R13 will have been initialized with

a reasonable value by the bootloader but in a stand-alone system you would have to do this as part of your initializations.

3. **Re-enable the IRQ input on the processor.** Remember that as part of the automatic response of the processor to an IRQ interrupt, the IRQ interrupt is disabled by setting bit 7 of the IRQ mode CSPR. To enable the IRQ input again, you use a Read-Modify-Write sequence of instructions with a special instruction to write a word with a 0 in bit 7 to the IRQ mode CPSR without changing the other bits in the CPSR. As shown by the instructions 4,5 and 6 in the program in Figure 5-8, you use a Move Status Register to Register instruction, MRS R3, CPSR, to copy the CPSR to R3. You then clear bit 7 of the CPSR value in R3 with the BIC R3,R3,#0x80 instruction. Finally, you write the word in R3 back to the CPSR with the Move Register to Status Register instruction, MSR CPSR_c, R3 instruction. The _c suffix on the MSR instruction tells the assembler to code this instruction such that only the lowest 8 bits of the value in R3 are written to the CPSR. The ARM-based processors do not allow you to write a 32-bit value directly to a status register. In the next section we show you how to access the flags in the status register.

```
@ IRQ procedure modified so that it can be interrupted by
@ a higher priority IRQ interrupt
      STMFD R13!, {R0-R3, R14} @ SAVE REGISTERS AND RETURN ADDRESS

@ PUSH SPSR ON STACK
      MRS R3, SPSR       @ COPY SPSR TO R3
      STMFD R13!, {R3}   @ SAVE COPY ON STACK

@ MASK LOWER PRIORITY IRQ INTERRUPTS
@   (THE CODE FOR THIS DEPENDS ON THE SPECIFIC PROCESSOR USED)

@ ENABLE IRQ TO ALLOW HIGHER PRIORITY IRQ INTERRUPT
      MRS R3, CPSR       @ COPY CPSR TO R3 TO ENABLE IRQ
      BIC   R3, #0x80    @ CLEAR BIT 7 (IRQ) IN R3 TO ENABLE IRQ
      MSR CPSR_c, R3     @ COPY BITS 7:0 OF R3 TO BITS 7:0 OF CPSR

; FIRST IRQ PROCEDURE INSTRUCTIONS HERE

; DISABLE IRQ TO PROTECT EXIT CODE "CRITICAL REGION"
      MRS R3, CPSR       @ COPY CPSR TO R3
      ORR R3, R3, #0x80  @ SET BIT 7 (IRQ) IN R3 TO DISABLE IRQ
      MSR CPSR_c, R3     @ COPY BITS 7:0 OF R3 TO BITS 7:0 OF CPSR

@ UNMASK LOWER PRIORITY IRQ INTERRUPTS
@   (THE CODE FOR THIS DEPENDS ON THE SPECIFIC PROCESSOR USED)

@ RESTORE SPSR
      LDMFD  R13!, {R3} @ LOAD SAVED VALUE OF SPSR FROM STACK
      MSR SPSR_cf, R3   @ RESTORE MODE, IRQ, FIQ, AND FLAGS TO SPSR
      LDMFD R13!,{R0-R3, R14} @ RESTORE REGISTERS AND LR
      SUBS   PC, LR, #4 @ COPY SPSR TO CPSR IN INTERRUPTED MODE
                        @ AND RETURN TO INTERUPTED PROGRAM
```

Figure 5-8. IRQ procedure modified so that it can be interrupted by a higher priority IRQ interrupt.

4. At the end of the first level IRQ procedure, you have to disable the IRQ interrupt input again, unmask the lower priority interrupts, restore the **SPSR value you pushed**

on the stack at the start of the procedure, restore other pushed registers, and return to the interrupted program as shown by the last block of instructions in Figure 5-8.

You have to disable the processor IRQ input, so that the processor cannot respond to another IRQ signals until execution has returned to the originally interrupted program. If the processor were allowed to respond to another IRQ while the SPSR was being restored or anytime before the return to the original program had been completed, the value of the SPSR would be lost. This would make it impossible to restore the CPSR of the original program when execution was returned to it. A section of code that must be executed as a block is commonly called a *critical region*. One way to protect a short critical region is by disabling interrupts that could interrupt execution of the code in the critical region.

To disable the processor IRQ interrupt input, you read the CPSR with an MRS R3, CPSR instruction, set bit 7 in this word with an ORR R3,R3 #0x80 instruction, and write the value in R3 to the lowest eight bits of the CPSR with the MRS CPSR_c, R3 instruction.

Now that IRQ is disabled, you pop the stored value for the SPSR off the stack with the LDMFD R13!, {R3} instruction. Copying the value from R3 to the SPSR is done with the MSR_cf SPSR, R3 instruction. The _cf suffix on the MSR instruction tells the assembler to code the instruction to copy the IRQ, FIQ, Mode, and Flag bits from R3 to the SPSR.

As in previous examples, the LDMFD R13!, {R0-R3, R14}instruction pops the saved registers and the return address in R14 off the stack. The SUBS PC, LR, #4 instruction copies the value now in SPSR to the CPSR in the interrupted program and returns execution to the interrupted program. Note that, as we stated earlier, IRQ will automatically be re-enabled by the restoration of CPSR during the return to the interrupted program, if IRQ was enabled in that program. This must have been the case, or the processor would not have been able to respond to the first IRQ interrupt signal.

Any IRQ procedure that can be interrupted by another IRQ must contain these code sections. However, these sections are not necessary if an IRQ procedure can only be interrupted by an FIQ interrupt or some other exception, because each of these modes has its own SPSR, R14, and in the case of FIQ, even its own R8-R13. Also, you don't have to put the four tasks listed above in a low priority procedure if it executes so quickly that it will not significantly increase the latency for the high priority procedure. For example, note that in our example program in Figure 5-3 we did not re-enable interrupts in the INT_DIRECTOR or BUTTON_SVC procedures, because we did not want to introduce unnecessary complexity at that point and because these procedures execute so quickly that their execution would not significantly delay service of a higher priority IRQ interrupt. The higher priority interrupt in this case will just be serviced immediately after the BUTTON_SVC finishes.

To make an FIQ interrupt procedure interruptible by another FIQ interrupt, you have to add essentially the same four program sections as described above for an interruptible IRQ procedure. We leave it for you as an exercise at the end of the chapter to work out the details for this case. In the next section here, we describe how an ARM-based processor responds if two or more different type exceptions occur at exactly the same time.

EXCEPTION AND INTERRUPT PRIORITIES

It is very likely that exceptions of two or more different types can occur at the same time. To determine which of the exceptions to service first, a priority is assigned to each exception in the hardware design of the ARM processors. Figure 5-9 shows the priorities of the ARM-based processor exceptions. If two different exceptions occur at the same time, the processor will automatically respond to the exception with the higher priority. Since the response is automatic, the only consideration you have to make in writing a program is to make sure that you have included procedures to service all of the exceptions possible in that particular system. However, we will make a few comments about how the processor responds to various combinations of exceptions.

First, Reset overrides all other exceptions and any other pending exceptions are automatically cancelled when the Reset pin is asserted. Second, if a Data Abort and an FIQ exception occur at the same time, execution will immediately enter the Data Abort exception handler. However, since the processor's response to a Data Abort exception does not disable the FIQ interrupt, the processor will immediately go to the FIQ interrupt procedure. At the end of the FIQ procedure, execution will return to the Data Abort handler, complete this handler, and return to the interrupted program. Finally, if an FIQ and an IRQ occur at the same time, the processor will respond to the FIQ interrupt and in the process disable the IRQ interrupt on the processor. When execution returns from the FIQ procedure to the interrupted program, the IRQ interrupt input will automatically be re-enabled when the CPSR of the interrupted program is restored. Execution will then go to the IRQ service procedure.

In this first major section of the chapter, we have described in general how you work with interrupts and interrupt procedures on any ARM-based processor. In the next major section of the chapter we describe in detail the structure, initialization, and use of the interrupt controller in the ARM-based TI AM335X family processors. Then in following sections we show examples of several important interrupt applications.

Priority		Exception
Highest	1	Reset
	2	Data Abort
	3	FIQ
	4	IRQ
	5	Prefetch Abort
Lowest	6	Undefined instruction SWI

Figure 5-9. Exception priorities for arm-based processors

AM335X Interrupt Architecture and Processing

For the architecture and initialization examples in this section, we will use the interrupt controller and the GPIO pins implemented in the ARM Cortex A-8 based Sitara AM335X processors. Specifically, we will show how a push-button switch connected to GPIO1_14 can be

used to generate an interrupt signal that calls an IRQ interrupt service procedure each time the switch is pushed. Each time the IRQ interrupt service procedure is called, it will light an LED connected to GPIO1_12 for about 1 second. The overall program flow for this is basically the same as that in the introductory program in Figure 5-3 but some additional steps are needed due to the greater capabilities (complexity) of the AM335X processor.

In this section, we will work through the required initialization steps one at a time and then show a program that puts them all together. The high level algorithm for the mainline is as follows:

1. Turn on GPIO1 module clock.
2. Initialize GPIO1_12 as a logic low and program it as an output to drive the LED.
3. Initialize GPIO1_14 to recognize and generate an interrupt signal for a falling edge on an input signal from the push-button switch connected to it.
4. Initialize the required registers in the AM335X Interrupt Controller, INTC
5. Hook the IRQ interrupt vector and chain it to your INT_DIRECTOR PROCEDURE that determines if the IRQ came from the switch or from some source serviced by the system IRQ interrupt service procedures.
6. Enable the processor IRQ input by clearing bit 7 of the Current Program Status Register, if this has not already been done by system startup program.
7. Wait in endless loop for an interrupt to occur.

GPIO1 INITIALIZATION FOR INTERRUPT GENERATION

To turn on the clock to the GPIO1 module, you use the instructions explained in Chapter 4 and shown in Figure 4-16. The process required to program GPIO1_12 to output a low and making it an output is a repeat of the GPIO discussion in chapter 4. Figure 5-10 shows the register template for the GPIO1 registers that control GPIO1_12 and GPIO1_14. To turn off the LED connected to GPIO1_12 you need to write a 1 to bit 12 of the GPIO1_CLEARDATAOUT register. As stated in Chapter 4 and shown in Appendix C, the base address for the GPIO1 control registers is 0x4804C000 and according to the GPIO Control Register list in Appendix D or Figure 4-11, the offset for the GPIO1_CLEARDATAOUT register is 0x190. Therefore, for this step you write a word with a 1 in bit 12(0x00001000) to address 0x4804C190. To enable GPIO_12 as an output, you have to write a word with a 0 in bit 12 to the GPIO1_OE register at offset 0x134 using a READ, MODIFY, WRITE process.

GPIO1_14 is by default an input but, in order to program it to detect the falling edge generated by pressing the debounced pushbutton, you have to write a 1 to bit 14 of the GPIO1_FALLINGDETECT REGISTER at offset 0x148. (Write 0x00004000 to address 0x4804C14C using a READ, MODIFY, WRITE process.)

GPIO 1 bit#	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Function																
OE	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Hex Val	0				0				0				0			

GPIO 1 bit#	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Function		BTN		LED												
OE	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
Hex Val	E for OE, 1 for clr/set				F				F				F			
Hex Val	4 for edge and enable				0				0				0			

Figure 5-10 GPIO1 template for Button IRQ program

In order to enable the GPIO module to send an interrupt signal to the Interrupt Controller, INTC, you also have to enable GPIO1_14 as an interrupt source. As shown in Figure 4-9, all enabled interrupt signals from a GPIO Module are merged into one or two interrupt request output signals called POINTRPEND1 and POINTRPEND2. Each of the 32 possible GPIO interrupt signals in a module can be enabled to generate one or both of these signals. (Two outputs are provided to allow separation of groups of interrupts.) For this example, let's just enable the GPIO_14 signal to produce and interrupt request on POINTRPEND1. To do this, you write a word with a 1 in bit 14 to the GPIO1_IRQSTATUS_SET_0 register at offset 0x34. (Write 0x00004000 to address 0x4804C034.) You don't have to use an RMW process for this one, because writing a zero has no effect, so all you have to do is write a 1 in the desired bit and zeros in all the rest. Note that if you wanted to enable the signal to generate a request on the POINTRPEND2 output, you would use the GPIO1_IRQSTATUS_SET_1 register at offset 0x38. Next, let's take a look at what you have to do enable the processor's Interrupt Controller, INTC, to respond to an interrupt request on POINTRPEND1 or POINTRPEND2.

INTC INITIALIZATION

The Interrupt Controller, INTC in the Cortex A-8 Microprocessor Unit or MPU processes interrupt requests from device modules and system sources. The INTC can handle interrupt requests from up to 128 sources. For reference, Appendix D shows the assigned interrupt number for each of the 128 possible interrupts. The INTC handles interrupt requests in priority order and sends an IRQ or FIQ interrupt signal for each to the core processor where they are serviced as we described for IRQ and FIQ interrupts in an earlier section of the chapter. A clock generator in the MPU generates the clock signals used by the INTC.

Figure 5-11 shows an internal block diagram of the INTC. The horizontal, grey rectangular boxes represent INTC control/status registers that you write a value to in order to tell the INTC how you want it to respond to a particular interrupt request or read a value from in order to determine the state of an interrupt line or interrupt request. The 128 INTC interrupts are divided into four, 32-bit banks, so each of the horizontal rectangular boxes in Figure 5-11 that has a p as part of the name actually represents four 32-bit registers, one for each bank of 32 interrupt sources. For example, the MIRp box represents 4, 32-bit registers, MIR0-MIR3, that are used to mask/unmask interrupt requests in the four banks. MIR0 is used to mask/unmask interrupt request inputs for the 32 interrupt sources in bank 0, MIR1 for the 32 in bank 1, etc. Likewise, you read the ITR1 register to determine or change the interrupt status of any of the interrupt requests in bank 1. You read the PENDING_IRQ1 register to determine if one of the interrupt requests in bank 1 has produced an IRQ request to the processor. As another example, the ISR_SET1 register can be used to generate an interrupt request on one of the bank 1 inputs and ISR_SET2 can be used to generate an interrupt request on one of the bank 2 inputs, etc. This feature makes it possible to generate a software interrupt to test an interrupt service procedure

without needing the physical device to generate the interrupt signal.

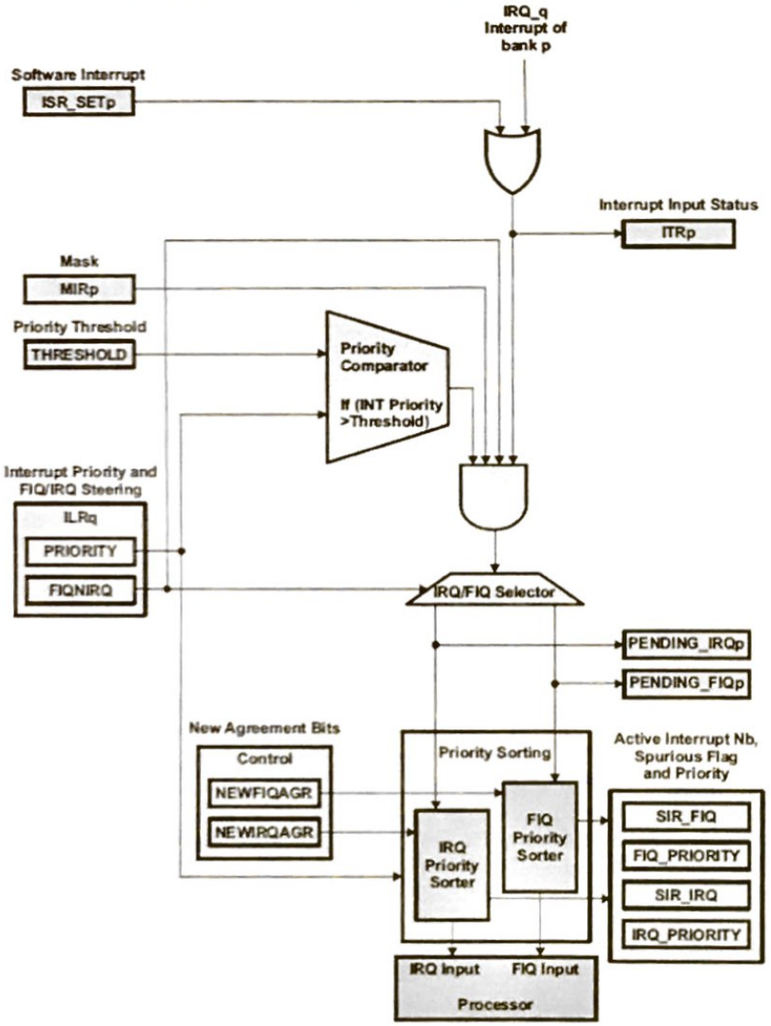


Figure 5-11. AM335X Interrupt Controller (INTC) Block Diagram

Appendix E shows the INTC registers and the offsets from the base address for each. According to the ARM Cortex A-8 memory Map in Appendix C, the base address for the INTC registers is 0x48200000. The address for one of the registers then is just the base address of 0x4820000 + the offset for the register.

The INTC allows you to set the priority of any interrupt request, so that a higher priority interrupt can be allowed to interrupt a lower priority request, but we will skip the details of this. The default priority is priority 0, which is the highest priority, so the interrupt for this example will be executed to completion without interruption. You can consult Chapter 6 of http://www.ti.com/lit/ug/spnu118o/spnu118o.pdf. if you want to implement interrupt prioritization for a particular application.

The main initialization step you have to do in the INTC for a basic system is to unmask the interrupt coming from the GPIO module, so the INTC can generate an IRQ or FIQ signal to the processor core. By default, all the INTC interrupt inputs are mapped to IRQ but an interrupt can be directed to the FIQ input by setting a bit in one of the INTC_ILRm registers at offsets 0x100-0x2FC. Since we want our GPIO interrupt to generate an IRQ signal, the default for this is fine. Now, as we stated in the introduction above, you unmask an interrupt request using the appropriate the INTC_MIR register. According to the ARM Cortex A-8 Interrupts numbers table in Appendix E, the POINTRPEND1signal from the GPIO1 module is connected to Int number 98 of the INTC and identified as GPIOINT1A. Ints #96 - #127 are enabled/disabled by the corresponding bits in the MIR3 register. Int # 98 corresponds to bit 2 in the MIR3 register. By default all MIR bits are 0's. To unmask bit 2 of the MIR3 register you can simply write 0x00000004 to the INTC_MIR_SET3 register at address 0x482000EC + offset 0xEC. (To mask the signal on bit 2 of MIR3, then you would simply write the 0x00000004 to the INTC_MIR__CLEAR3 register at 0x482000E8.

Two additional initialization steps you can take to reduce power dissipation of the INTC module involve setting some INTC clocks so they are only turned on when needed. For many embedded applications it is very important to minimize power dissipation and current microprocessors allow you to do this by only turning on circuitry is needed at a particular time. In this case, you can enable auto idle by setting bit 0 of the INTC_SYSCONFIG register at offset 0x10 and setting the Turbo bit, bit 1 of the INTC_IDLE register at offset 0x50. Consult Section 6.1.3 of http://www.ti.com/lit/ug/spnu118o/spnu118o.pdf for a more detailed explanation of these.

HOOKING AND CHAINING THE IRQ INTERRUPT VECTOR

In an earlier section of the chapter we described how an ARM-based processor automatically executes the instruction at 0x00000018 in response to an IRQ interrupt request. An LDR instruction at that address reads the starting address of the system IRQ procedure from the literal pool and loads it into the Program Counter. The processor then executes the system IRQ service procedure. In order to intercept this flow, so execution goes to your INT_DIRECTOR procedure, instead of to the system IRQ procedure, you "hook the interrupt vector" by reading the system IRQ procedure address from the literal pool, saving it for future reference, and inserting the address of your INT_DIRECTOR procedure in that location in the literal pool. Obviously the literal pool location for this address must be in RAM, so that you can write your INT_DIRECTOR procedure address to it.

In an ARM Cortex A-8 based system, the Exception Vectors are copied from memory starting at 0x00000000 to the public ROM area starting at 0x20000. The IRQ LDR instruction, for example ends up at 0x20018 in the public, on-chip ROM. The vectors are then moved into the on-chip RAM, starting at address 0x4030CE00 and the LDR instruction for the IRQ interrupt vector ends up at 0x4030CE18. As shown in Figure 5-12, the LDR instruction at this address goes to address 0x4030CE38 to read the address of the system IRQ interrupt service procedure.

This then is the address you would need to read and save what you read, so you can send execution to the system IRQ procedure, if the IRQ was not caused by your button push. After reading and saving the address from this address you would then write the starting address of your INT_DIRECTOR procedure to this address.

Address	Exception	Content
4030CE00h	Reserved	Reserved
4030CE04h	Undefined	PC = [4030CE24h]
4030CE08h	SWI	PC = [4030CE28h]
4030CE0Ch	Pre-fetch abort	PC = [4030CE2Ch]
4030CE10h	Data abort	PC = [4030CE30h]
4030CE14h	Unused	PC = [4030CE34h]
4030CE18h	IRQ	PC = [4030CE38h]
4030CE1Ch	FIQ	PC = [4030CE3Ch]
4030CE20h	Reserved	20090h
4030CE24h	Undefined	20080h
4030CE28h	SWI	20084h
4030CE2Ch	Pre-fetch abort	Address of default pre-fetch abort handler(1)
4030CE30h	Data abort	Address of default data abort handler(1)
4030CE34h	Unused	20090h
4030CE38h	IRQ	Address of default IRQ handler
4030CE3Ch	FIQ	20098h

Figure 5-12. ARM Cortex -A8 RAM Exception Vectors on BeagleBone Black board

However, when doing assembly language programming on a BeagleBone Black board using the TI CCS tools, a startup program called startup_ARMCA8.s is automatically assembled, linked with your code, and run before your code. Part of the code in this startup_ARMCA8 program relocates the Interrupt Vector Table to a location in DRAM that is just below the address where your code is loaded. Figure 5-13a shows the instructions that this startup program sets up for the Interrupt Vector Table. As an example the startup program installs the instruction ldr pc, [pc,#-8] for the IRQ vector. If you analyze the operation of this instruction carefully, you should see that the instruction performs an endless loop or "dead loop" as the manual calls it. When the processor responds to an IRQ interrupt, it will simply go to this dead loop and stay there. To get execution to go to your INT_DIRECTOR, instead of just executing the ldr pc, [pc,#-8] forever and locking up the system, you replace this instruction with a b INT_DIRECTOR instruction as shown in Figure 5-13b. This instruction will do an unconditional branch to your INT_DIRECTOR code. Note that you also have to add the .extern INT_DIRECTOR directive to tell the assembler that the symbol INT_DIRECTOR is in some other module. As you will see in the complete program later, you also have to put a .global INT_DIRECTOR directive at the start of your assembly language program to make the symbol INT_DIRECTOR accessible to other modules.

```
.section .isr_vector
    .align 4
    .globl __isr_vector
__isr_vector:
     ldr     pc, [pc,#24]       @ 0x00 Reset
     ldr     pc, [pc,#-8]       @ 0x04 Undefined Instruction
     ldr     pc, [pc,#-8]       @ 0x08 Supervisor Call
     ldr     pc, [pc,#-8]       @ 0x0C Prefetch Abort
     ldr     pc, [pc,#-8]       @ 0x10 Data Abort
     ldr     pc, [pc,#-8]       @ 0x14 Not used
     ldr     pc, [pc,#-8]       @ 0x18 IRQ interrupt
     ldr     pc, [pc,#-8]       @ 0x1C FIQ interrupt
    .long    Entry
```

(a)

```
.extern  INT_DIRECTOR
.section .isr_vector
    .align 4
    .globl __isr_vector
__isr_vector:
     ldr     pc, [pc,#24]       @ 0x00 Reset
     ldr     pc, [pc,#-8]       @ 0x04 Undefined Instruction
     ldr     pc, [pc,#24]       @ 0x08 Supervisor Call
     ldr     pc, [pc,#-8]       @ 0x0C Prefetch Abort
     ldr     pc, [pc,#-8]       @ 0x10 Data Abort
     ldr     pc, [pc,#-8]       @ 0x14 Not used
     b          INT_DIRECTOR          @ 0x18 IRQ interrupt goes to here
     ldr     pc, [pc,#-8]       @ 0x1C FIQ interrupt
```

(b)

Figure 5-13. (a) startup_ARMCA8.s code for Interrupt Vector Table with dead loops. (b) Edited startup_ARMCA8.s code to take execution to INT_DIRECTOR

ENABLING THE PROCESSOR IRQ INPUT

As we described earlier in the chapter, you do this by reading the CPSR into R3 with the special MRS R3, CPSR instruction, clearing bit 7, the IRQ bit, in the word read in to enable IRQ, and then writing the resultant word back to the CPSR with the MSR CPSR_c, R3 instruction. Remember that the underscore c on the MRS instruction tells the assembler should code the instruction to modify only the lower eight bits in the CPSR.

DETERMINING THE SOURCE OF AN IRQ IN INT_DIRECTOR

When determining the source of and interrupt, you start at the highest level in the interrupt tree. For the example program in Figure 5-3 all we had to do to determine if the IRQ request was caused by a button push was to test a bit in the INTPND register. For this example, you check at the INTC level first. If you find an interrupt request from one of the 128 INTC inputs, you work your way down through the sources connected to that INTC input to check which specific GPIO pin or other source caused that interrupt request.

A High Level Algorithm for this section would be:

Check INT_PENDING_IRQ3 register bit 2 to see if IRQ from GPIOINT1A
IF no, then restore registers and return
ELSE,
    Check GPIO1_IRQSTATUS_0 register bit 14 to see if Button pressed

If No, then restore registers and return to Wait Loop.
(BeagleboneBlack has no system IRQ procedure, so just go back to Wait Loop.)
Else go to Button Service

A Low-Level algorithm for this section would be:
Restore registers
Read INT_PENDING_IRQ3 register at 0x4820000F8,(INTC base + offset F8)
Test Bit 2, to see if GPIO1 POINTRPEND1 from GPIO1A
If Bit 2 = 0, restore registers and return from interrupt
Else read GPIO1_IRQSTATUS_0 register at 0x4804C02C, (GPIO base + offset 0x2C)
Test bit 14 with 0x000004000 to see if GPIO1_14 Button pressed
If bit 14 =0 restore registers and return from interrupt
Else go to BUTTON_SVC.

THE BUTTON SERVICE PROGRAM SECTION

The first step in servicing the GPIO1_14 interrupt request is to turn off the GPIO1 interrupt request to the INTC controller, so that when you return to the interrupted program, the interrupt request will not still be there and cause another interrupt. You do this by simply writing a 1 to bit 14 in the GPIO1_IRQSTATUS_0 register you read at address 0x4804C02C to determine if GPIO1_14 was causing an interrupt. You don't have to use a RMW process for this write, because writing 0's to the other bits in the register has no effect. As determined with the help of the register template in Figure 5-10, the word you use for checking and writing bit 14 is 0x00004000. One additional step is to Turn off the NEWIRQ bit in the INTC_CONTROL register so the processor can respond to another IRQ request. To do this, you simply write 0x01 to the INTC_CONTROL register at 0x48200048.

The instructions for turning the LED on for 2 seconds and then off are just a slight modification of those we showed you for doing it in Chapter 4. The next step here then is to show you the complete program for this example.

THE COMPETE BUTTON SERVICE PROGRAM

Figure 5-14 shows the complete assembly language for the button service program that we developed in the preceding sections. Work your way through this to see how you pull together the pieces we described in the preceding pages. Refer back to the preceding discussions to help you understand each instruction. Then, in the next section of the chapter we will show you how to accurately measure off times such as the 2 seconds with a programmable timer, instead of a delay loop.

```
@ BUTTON INTERRUPT PROGRAM
@ Runs on BeagleBone Black Board with cape.
@ WHEN THE OUTPUT FROM A DEBOUNCED SWITCH CONNECTED TO GPIO1_14 GOES LOW, THIS @
PROGRAM PRODUCES AN IRQ INTERRUPT TO THE PROCESSOR. The IRQ SERVICE
@ PROCECDURE FOR THIS REQUEST WILL LIGHT AN LED CONNECTED TO GPIO1_12 FOR
@ ABOUT 2 SECONDS
@ Copyright Douglas V. Hall Fall 2016
```



```
.text
.global _start
.global INT_DIRECTOR
_start:
      LDR R13,=STACK1          @ Point to base of STACK for SVC mode
      ADD R13, R13,#0X1000     @ Point to top of STACK
      CPS #0x12                      @ Switch to IRQ mode
      LDR R13,=STACK2          @ Point to IRQ stack
      ADD R13, R13,#0X1000     @ Point to top of STACK
      CPS #0x13                      @ Back to SVC mode
@ Turn on GPIO1 CLK
   LDR R0,#0x02                @ Value to enable clock for a GPIO module
   LDR R1,=0x44E000AC      @ Address of CM_PER_GPIO1_CLKCTRL Register
   STR R0, [R1]                      @ Write #02 to register
      LDR R0,=0x4804C000       @ Base address for GPIO1 registers
      ADD R4, R0,#0x190        @ Address of GPIO1_CLEARDATAOUT register
      MOV R7, #0x00001000      @ Load value to turn off LED on GPIO1_12
      STR R7,[R4]                    @ Write to GPIO1_CLEARDATAOUT register
@ Program GPIO1_12 as output
      ADD R1,R0,#0x0134              @ Make GPIO1_OE register address
      LDR R6,[R1]                    @ READ current GPIO1 Output Enable register
      LDR R7,=0xFFFFEFFF       @ Word to enable GPIO1_12 as output(0 enables)
      AND R6,R7,R6                   @ Clear bit 12 (MODIFY)
      STR R6, [R1]             @ WRITE to GPIO1 Output Enable register
@ Detect falling edge on GPIO1_14 and enable to assert POINTRPEND1
      ADD R1, R0, #0x14C       @ R1 = address of GPIO1_FALLINGDETECT register
      MOV R2, #0x00004000      @ LOAD VALUE FOR BIT 14
      LDR R3, [R1]             @ READ GPIO1_FALLINGDETECT register
      ORR R3, R3, R2           @ Modify (set bit 14)
      STR R3, [R1]             @ Write back
      ADD R1, R0, #0x34        @ Address of GPIO1_IRQSTATUS_SET_0 register
      STR R2, [R1]             @ Enable GPIO1_14 request on POINTRPEND1
@ Initialize INTC
      LDR R1,= 0x482000E8      @ Address of INTC_MIR_CLEAR3 register
      MOV R2,#0x04             @ Value to unmask INTC INT 98, GPIOINT1A
      STR R2,[R1]              @ Write to INTC_MIR_CLEAR3 register
@Hook IRQ vector and chain BUTTON_SVC procedure (NOT USED WITH CCS startupARMCA8.s)
      @LDR R1,= 0x4030CE38     @ Address of first instruction in SYS_IRQ procedure
      @LDR R2, [R1]            @ read SYS_IRQ address
      @LDR R3,= SYS_IRQ        @ Address where SYS_IRQ address saved
      @STR R2, [R3]            @ Save SYS_IRQ address to use if not our IRQ
      @LDR R2,=INT_DIRECTOR    @ Load address of our INT_DIRECTOR
      @STR R2, [R1]            @ Store in SYS_IRQ first address location in literal
pool
@ Make sure processor IRQ enabled in CPSR
      MRS R3, CPSR             @ Copy CPSR to R3
      BIC R3,#0x80             @ Clear bit 7
      MSR CPSR_c, R3           @ Write back to CPSR
@ Wait for interrupt
LOOP:   NOP
        B LOOP


INT_DIRECTOR:
```

```
        STMFD SP!, {R0-R3, LR}     @ Push registers on stack
        LDR R0,=0X482000F8         @ Address of INTC-PENDING_IRQ3 register
        LDR R1,[R0]                     @ Read INTC-PENDING_IRQ3 register
        TST R1,#0X00000004         @ TEST BIT 2
        BEQ PASS_ON                @ Not from GPIOINT1A, go to back to wait loop, Else
        LDR R0,=0X4804C02C         @ Load GPIO1_IRQSTATUS_0 register address
        LDR R1, [R0]                    @ Read STATUS register
        TST R1,#0x00004000         @ Check if bit 14 =1
        BNE BUTTON_SVC                  @ If bit 14 =1, then button pushed
        BEQ PASS_ON                     @ If bit 14 =0, then go to back to wait loop
PASS_ON:
        LDMFD SP!, {R0-R3,LR}      @ Restore registers
        SUBS PC, LR, #4            @ Pass execution on to wait LOOP for now

BUTTON_SVC:
        MOV R1,#0x00004000         @ Value turns off GPIO1_14 Interrupt request
                                   @ Also turns off INTC interrupt request
        STR R1,[R0]                @ Write to GPIO1_IRQSTATUS_0 register
@ Turn off NEWIRQA bit in INTC_CONTROL, so processor can respond to new IRQ
        LDR R0,=0x48200048         @ Address of INTC_CONTROL register
        MOV R1, #01                @ Value to clear bit 0
        STR R1,[R0]                @ Write to INTC_CONTROL register
@ Turn on LED ON GPIO1_12
        LDR R0,=0x4804C194         @ Load address of GPIO1_SETDATAOUT register
        MOV R1, #0x00001000        @ Load value to turn on GPIO1_12
        STR R1,[R0]                @ Write to GPIO1_SETDATAOUT register
@ Wait 2 seconds
        MOV R2,#0x00400000         @ Load delay value for 2 seconds
LOOP2:
       NOP
       SUBS R2, #1                 @ Count down
       BNE LOOP2
@ Turn off LED on GPIO1_12
        LDR R0,=0x4804C190         @ Load address of GPIO1_CLEARDATAOUT register
        STR R1, [R0]               @ Write 0x00001000 to GPIO1_CLEARDATAOUT register
@Return to wait loop
       LDMFD SP!, {R0-R3,LR}       @ Restore registers
       SUBS PC, LR, #4             @ Return from IRQ interrupt procedure
.align 2
SYS_IRQ:  .WORD 0                  @ Location to store systems IRQ address
.data
.align 2
STACK1:      .rept 1024
             .word 0x0000
             .endr
STACK2:      .rept 1024
             .word 0x0000
             .endr
.END
```

Figure 5-14 ARM Assembly language for Button Service/LED program

Using a Programmable Timer to Generate Interrupts

SITARA TIMERS OVERVIEW

The TI AM335X Sitara processor has several very versatile counter/timers as described in detail in the Timers Chapter of the AM335X Sitara Technical Reference Manual. These include DMTimer0, DMTimers2-7, DMTimer 1ms, a Real Time clock that can be used to keep track of seconds, minutes, hours, etc., and a WATCHDOG Timer that is used by an operating system to get the system out of a lockup condition. To show the setup and basic operation of a timer to generate interrupts at desired time intervals, we will use DMTimer2.

Figure 5-15 shows the basic connections for DMTimers. First note that the clock for one of these timers can come from an external source, from a 32,768 Hz clock source, or from a high frequency system clock. You will identify the desired source for a particular application by programming the multiplexer on the *pclktimer* input. Next, note that on the right side of the figure that some of the timers can generate output pulses and that these timers can be triggered by an external source. Finally, note that a timer can generate an interrupt to the MPU Subsystem on its *pointer_pend* output, similar to the way a GPIO module can as we discussed in the preceding section.

DM Timer counters are up counters. A counter can be used to count external events or the number of clock pulses. A counter can be programmed to generate an interrupt when the count matches a preloaded count or generate an interrupt when the count overflows or in other words, passes the maximum value of 0xFFFFFFFF.

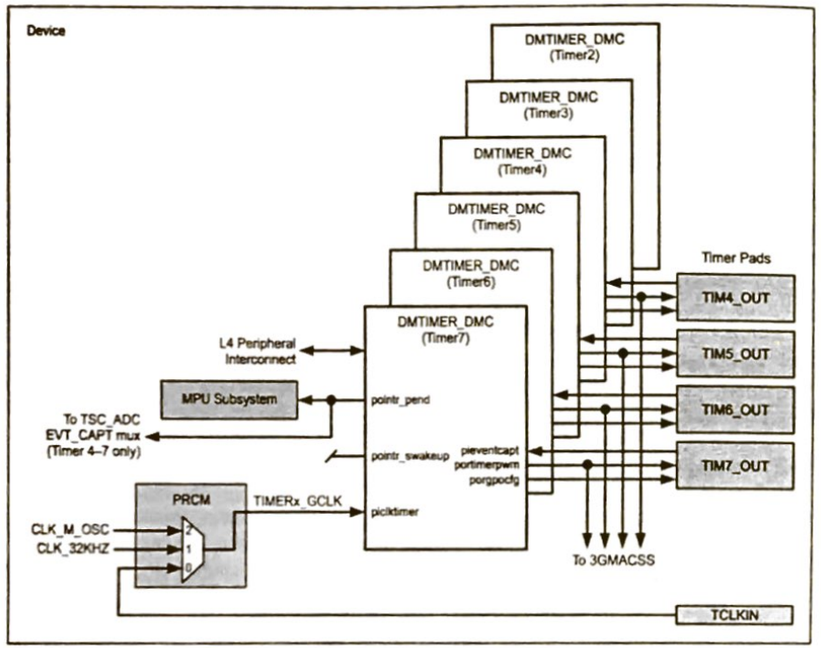


Figure 5-15 TI Sitara AM3359 timers 2-7 Connections

AN AM335X TIMER EXAMPLE PROGRAM

For our timer example, we will extend the button interrupt program in Figure 5-14 as follows. When the button connected to GPIO1_14 is pressed, the falling edge of the signal from the button will produce an IRQ interrupt. The service procedure for the Button Interrupt will turn on an LED connected to GPIO1_12 and start Timer 2 counting up using a 32.768 KHz clock. When the Timer2 count overflows after 0.5 seconds, a timer2 IRQ interrupt will be generated. The IRQ service procedure for this Timer2 interrupt request will turn off the LED and return to the wait loop. Timer2 will be set to reload the count after each overflow, so that when the count overflows again, another Timer2 interrupt will be produced. This time the Timer2 interrupt procedure will turn the LED on again. The result is that the LED will continuously blink on for 0.5 seconds and off for 0.5 seconds.

Figure 5-16 shows the High-Level Algorithm for this program. For a start, read through the High-Level Algorithm to get an overview of the program. Then skim read through the DMTimer section of the Sitara data book Timers chapter. Based the discussion of the Button service program in Figure 5-14, and what you read in the Sitara data book, start to think about how you would implement each step. Much of this should be very familiar from the explanations for the Button/LED program in Figure 5-14 but the timer sections obviously require some more explanation. We will work through these with you in the order they appear in Figure 5-16.

MAINLINE
1. Set up stacks for Supervisor Mode and IRQ mode
2. Set GPIO1_12 for LED off (low)
3. Set GPIO1_12 as output
4. Set up to detect falling edge on GPIO1_14 and generate interrupt
5. Initialize INTC – Reset INTC and Enable Timer 2 interrupt on INTC #68, button interrupt on INTC #98
6. Turn on Timer2 CLK
7. Set Timer2 functional clock input multiplexer for 32.768 KHz clock
8. Initialize timer registers for desired count and overflow interrupt generation
9. Enable IRQ input by clear bit 7 in CPSR

INT_DIRECTOR
1. Save registers
2. Check if interrupt from button
3. IF YES, Go service Button interrupt
ELSE Check for Timer2 interrupt
IF NOT Timer2 Interrupt, Restore registers and return to wait loop.
ELSE check if Timer2 Overflow interrupt.
IF NOT Timer2 Overflow interrupt, then restore registers and return
ELSE , Go toggle LED

BUTTON_SVC
1. Turn off GPIO1_14 interrupt
2. Turn on LED
3. Start timer2 and set for auto reload
4. Enable INTC for new interrupt
5. Restore registers and return to wait loop

LED
1. Turn off timer2 Overflow Interrupt request
2. Read GPIO1 and check bit 12.
3. If bit 12 = 0, LED off. Output a high on GPIO1_12
4. If bit 12 = 1, LED on, Output a low on GPIO1_12
5. Enable INTC for new interrupt
6. Restore registers and return to wait loop

Figure 5-16 High-Level Algorithm for Timer/LED Interrupt Program

To enable a timer2 interrupt on INTC, you find from the ARM Cortex Interrupt Numbers Table in Appendix D, that the interrupt associated with TIMER2 is #68. This corresponds to bit 4 of the INTC_MIR_CLR2 register at offset 0xC8 from the INTC base address of 0x48200000. You just write 0x10 to this address to enable a Timer2 interrupt in INTC. This is very similar to the way you enable a GPIO module interrupt on INTC.

To turn on the clock to the Timer2 module, you go to the ARM cortex Memory map to find the base address of the Clock Module Peripheral registers, CM_PER. This address is 0x44E00000. Then you go to the Power, Clock and Reset Management chapter of the Sitara Data

book to find the data sheet for the Timer2 CM_PER_TIMER2_CLKCTRL register at offset 0x80. As indicated in the data sheet, you write 0x02 to this register to enable the clock to the module.

The next step is to connect the correct frequency functional clock signal to the clock input of the timer. As shown in Figure 5-15 here, the PRCM multiplexer can be programmed to provide a high frequency clock signal from the CLK_M_OSCillator, a 32.768 KHz signal, or a signal from an external source. You want the 32.768 KH signal as the clock signal. As shown in Figure 8-17 in the Sitara data book, the control signal for the PRCM multiplexer comes from the PRCMCLKSEL_TIMER2_CLK.CLKSEL register. The trick is to find this register. Took me a few minutes to find this, since it is in the PRCM chapter of the Sitara data book, but not where I expected it to be. It is in the CM_DPLL register section of the chapter. If you pop up the CM_DPLL register list, you find that the offset of the PRCMCLKSEL_TIMER2 register from the CM_PER base of 0x44E00000 is 0x508. As indicated in the data sheet you write 0x02 to the resultant address of 0x44E00508 to select the 32.768 KHz clock.

Now we will work through how you initialize the timer itself using the Timer registers shown in Appendix G. As shown in The ARM Cortex Memory Map, the base address for the timer2 registers is 0x48040000, so the Timer register offsets shown in Appendix F are added to this base to access the register. First we reset Timer 2 by writing 0x1 to the Timer2 Configuration register at offset 0x10. Then we write 0x2 to the Timer2 IRQ_ENABLE_SET register at offset 0x2C to enable the Timer 2 to generate and interrupt signal when the counter overflows.

Finally in this section of the program we write values to the Timer2 Load and Count registers. The value in the Count Register is the value that gets counted up by clock pulses. The value in the Load Register is the value that gets loaded into the Count Register, if Timer2 is programmed for auto-reload on overflow. To calculate the value you put in these registers for a desired time delay you use the formula Desired Time = (FFFF FFFFh – TLDR + 1) × timer Clock period × Clock Divider (PS). The default value for the PreScaler, PS is 1, so the formula simplifies to Desired Time = (FFFF FFFFh – TLDR + 1) × timer Clock period. Since the clock for our example is 32,768 pulses per second, we would just have to count off 32,768 or 0x00008000 pulses for a time of one second. Since the Timer2 counter is an up counter, we have to basically subtract this from the overflow value which is 0xFFFFFFFF + 1 or 0x100000000. Juggling the formula a little, the value for TLDR = 0x100000000-0x00008000. The result for TLDR is 0xFFFF8000. For a half second delay, we count off half of 32,768/2 counts, or 0x4000 counts and the result for the TLDR value would be 0x100000000 – 0x4000 or 0xFFFFC000. We load this value into both the Load Register and the Count Register. You load the value in the Count Register, so it will be there for the first count when you start the counter. The value in the Load Register will be reloaded in the Count Register for the next count up, after the count overflows.

Figure 5-17 shows the Low-Level algorithm for the program and Figure 5-18 shows the complete assembly language program based on the Low-Level algorithm. Carefully work your way through these, one step at a time, until you understand each step and the overall program flow. Note we do not turn the timer on and set for auto count reload until the button is pushed.

Low-Level Algorithm for Timer/LED Interrupt Program Douglas V. Hall January, 2015
Portland, Oregon

MAINLINE
Set up stacks for Supervisor Mode and IRQ mode (See bp3.s)
Turn on GPIO1 Clock (See bp3.s)
Set GPIO1_12 for LED off (low) (See bp3.s)
3. Set GPIO1_12 as output (See bp3.s)
4. Initialize INTC – Reset INTC, Timer 2 interrupt on INTC #68, GPIO14 on INTC #98
   Write 0x2 to INTC_SYSCONFIG at 0x48200010 to reset INTC
   Write 0x00000010 to 0x482000C8 to Enable INTC #68 Timer2 interrupt input
   Write 0x00000004 to 0x482000E8 to Enable INTC #98 GPIO1_14 interrupt
5. Turn on Timer2 CLK
   Write 0x2 to CM__PER_TIMER2_CLKCTRL at 0x44E00000+0x80
6. Set Timer clock frequency multiplexer for 32.768 KHz
   Write 0x02 to PRCMCLKSEL_TIMER2 register at address 0x44E00508
7. Initialize timer registers for desired count, overflow interrupt generation
   Write 0x1 to Timer2 CFG register at 0x48040010 to Reset Timer 2
   Write 0x2 to Timer2 IRQENABLE_SET register at 0x4804002C
   Write 0xFFFFC000 to Timer2 TLDR register at 0x48040040 to get 0.5 second
   Write 0xFFFFC000 to Timer2 TCRR register at 0x4804003C to get 0.5 second
8. Enable IRQ input by clear bit 7 in CPSR
9. Wait for interrupt loop

INT_DIRECTOR
1. Save registers
2. Check if interrupt from button
   Read INTC_PENDING_IRQ3 REGISTER at 0x482000F8
   If Bit 2 = 0 then not GPIO3, go check for timer
   If bit 2 = 1 then check if GPIO1_14 from button
   Read GPIO1_IRQ_STATUS REGISTER at 0x4804C02C
   If bit 14 = 1, then go to BUTTON_SVC
   If bit 14 = 0 then, Enable new IRQ response in INTC by
      Write 0x1 to INTC_CONTROL register at 48200048 to allow new IRQ
   Restore registers and return to wait loop
3. Check in INTC if interrupt from Timer2-
   Read word from INTC PENDING_IRQ2 register at 0x482000D8
   Test with 0x00000010
   NO, Write 0x1 to INTC_CONTROL register at 48200048 to allow new IRQ
   Restore registers and return

4. YES, check if Timer2 Overflow. .
   Read value from Timer2 IRQSTATUS register at 0x48040028
   Test with 0x00000002.
   If bit 1 = 1 go to LED
   If bit 1 = 0 then

Write 0x1 to INTC_CONTROL register at 48200048 to allow new IRQ
Restore registers and return to wait loop

BUTTON_SVC

1. Turn off GPIO 1_14 interrupt request by
   Write 0x00004000 to GPIO1_IRQ_STATUS _0 register
2. Turn on LED by write 0x00001000 to GPI01_SETDATAOUT register
   at 0x4804C0194
3. Start timer2, and set for auto reload by write 0x03 to TCLR at 0x48040038
4. Write 0x1 to INTC_CONTROL register at 48200048 to allow new IRQ
5. Restore registers and return to wait loop

LED

1. Turn off timer2 Overflow Interrupt request by
   Write 0x02 to IRQSTATUS register at 0x48040028
2. Read GPIO1_DATAOUTREGISTER at 0x4804C13C
3. If bit 12 = 0, LED off. Output a high on GPIO1_12 with GPIO1_SET to turn on LED
   Write 0x00001000 to GPIO1_SET at 0x4804C194
4. If bit 12 = 1, LED off. Output a high on GPIO1_12 with GPIO1CLR to turn off LED
   Write 0x00001000 to GPIO1CLR at 0x4804C0190
5. Enable new IRQ response to INTC
   Write 0x1 to INTC_CONTROL register at 482200048 to allow new IRQ
6. Restore registers and return to wait loop

Figure 5-17 Low Level Algorithm for Timer/LED interrupt program

```
@ LED/Timer2 INTERRUPT PROGRAM
@ Runs on BeagleBone Black Board with Pushbutton/LED cape.
@
@ WHEN THE OUTPUT FROM A DEBOUNCED SWITCH ONGPIO1_14 GOES LOW,
@ THIS PROGRAM PRODUCES AN IRQ INTERRUPT TO THE PROCESSOR.The IRQ
@ SERVICE @ PROCECDURE FOR THIS REQUEST WILL LIGHT AN LED CONNECTED
@ TO GPIO1_12 and start Timer2. When the Timer 2 count overflows
@ after 0.5 seconds, an IRQ Interrupt generated. In response to
@ this interrupt,the LED will be toggled. Note: Program uses modified
@ Startup_ARMCA8 file to access IRQ Interrupt Service Procedure
@ INT_Director
@                  Copyright Douglas V. Hall Fall 2016

.text
.global _start
.global INT_DIRECTOR
_start:
      LDR R13,=STACK1  @ Point to base of STACK for SVC mode
      ADD R13, R13,#0X1000   @ Point to top of STACK
      CPS #0x12              @ Switch to IRQ mode
      LDR R13,=STACK2        @ Point to IRQ stack
      ADD R13, R13,#0X1000   @ Point to top of STACK
```



```
      CPS #0x13              @ Back to SVC mode
@ Turn on GPIO1 CLK
   LDR R0,#0x02              @ Value to enable clock for a GPIO module
   LDR R1,=0x44E000AC     @ Address of CM_PER_GPIO1_CLKCTRL Register
   STR R0, [ R1]                  @ Write #02 to register
      LDR R0,=0x4804C000     @ Base address for GPIO1 registers
      ADD R4, R0,#0x190      @ R4 = GPIO1_CLEARDATAOUT register
      MOV R7, #0x00001000    @ Load value to turn off LED on GPIO1_12
      STR R7,[R4]            @ Write to GPIO1_CLEARDATAOUT register
@ Program GPIO1_12 as output
      ADD R1,R0,#0x0134      @ Make GPIO1_OE register address
      LDR R6,[R1]            @ READ current GPIO1 Output Enable register
      LDR R7,=0xFFFFEFFF     @ Enable GPIO1_12 as output(0 enables)
      AND R6,R7,R6           @ Clear bit 12 (MODIFY)
      STR R6, [R1]           @ WRITE to GPIO1 Output Enable register
@ Detect falling edge on GPIO1_14 and enable to assert POINTRPEND1
      ADD R1, R0, #0x14C     @ GPIO1_FALLINGDETECT register
      MOV R2, #0x00004000    @ LOAD VALUE FOR BIT 14
      LDR R3, [R1]           @ READ GPIO1_FALLINGDETECT register
      ORR R3, R3, R2         @ Modify (set bit 14)
      STR R3, [R1]           @ Write back
      ADD R1, R0, #0x34      @ Address of GPIO1_IRQSTATUS_SET_0 register
      STR R2, [R1]           @ enable GPIO1_14 request on POINTRPEND1
@ Initialize INTC
      LDR R1,=0x48200000     @ Base address for INTC
      MOV R2, #0x2           @ Value to reset INTC
      STR R2, [R1,#0x10]     @ Write to INTC Config register
      MOV R2, #0x10          @ Unmask INTC INT 68, Timer2 interrupt
      STR R2, [R1,#0xC8]     @ Write to INTC_MIR_CLEAR2 register
      MOV R2,#0x04           @ Value to unmask INTC INT 98, GPIOINTA
      STR R2,[R1,#0xE8]      @ Write to INTC_MIR_CLEAR3 register
@Turn on Timer2 CLK
      MOV R2, #0x2           @ Value to enable Timer2 CLK
      LDR R1,= 0x44E00080    @ Address of CM_PER_TIMER2_CLKCTRL
      STR R2, [R1]           @ Turn on
      LDR R1,=0x44E00508     @ Address of PRCMCLKSEL_TIMER2 register
      STR R2, [R1]           @ Select 32 KHz CLK for Timer 2
@ Initialize Timer 2 registers, with count, overflow interrupt generation
      LDR R1,=0x48040000     @ Base address for Timer2 registers
      MOV R2, #0x1           @ Value to reset Timer 2
      STR R2, [R1,#0x10]     @ Write to Timer2 CFG register
      MOV R2, #0x2           @ Value to Enable Overflow interrupt
      STR R2, [R1,#0x2C]     @ Write to Timer2 IRQENABLE_SET
      LDR R2,= 0xFFFFC000    @ Count value for 0.5 seconds
      STR R2, [R1,#0x40]     @ Timer2 TLDR load register (Reload value)
      STR R2, [R1,#0x3C]     @ Write to Timer2 TCRR count register

      @ Make sure processor IRQ enabled in CPSR
      MRS R3, CPSR           @ Copy CPSR to R3
      BIC R3,#0x80           @ Clear bit 7
      MSR CPSR_c, R3         @ Write back to CPSR
@ Wait for interrupt
LOOP: NOP
      B LOOP
```

```
INT_DIRECTOR:

      STMFD SP!, {R0-R3,LR}    @ Push registers on stack
      LDR R1,=0X482000F8       @ Address of INTC-PENDING_IRQ3 register
      LDR R2,[R1]              @ Read INTC-PENDING_IRQ3 register
      TST R2,#0X00000004       @ TEST BIT 2
      BEQ TCHK                 @ Not GPIOINT1A, check if Timer2, else
      LDR R0,=0X4804C02C       @ GPIO1_IRQSTATUS_0 register address
      LDR R1, [R0]             @ Read STATUS register to see if button
      TST R1,#0x00004000       @ Check if bit 14 =1
      BNE BUTTON_SVC           @ If bit 14 =1, button pushed, service
      LDR R0,=0x48200048       @ Else, go back. INTC_CONTROL register
      MOV R1, #01              @ Value to clear bit 0
      STR R1,[R0]              @ Write to INTC_CONTROL register
      LDMFD SP!, {R0-R3,LR}    @ Restore registers
      SUBS PC, LR, #4          @ Pass execution to wait LOOP for now
TCHK:
      LDR R1,=0x482000D8       @ Address of INTC PENDING_1RQ2 register
      LDR R0,[R1]              @ Read value
      TST R0,#0x10             @ check if interrupt from Timer2
      BEQ PASS_ON              @ No, return, Yes, check for overflow
      LDR R1,=0x48040028       @ Address of Timer2 IRQSTATUS register
      LDR R0, [R1]             @ Read value
      TST R0, #0x2             @ Check bit 1
      BNE LED                  @ If Overflow, then go toggle LED
      PASS_ON:                 @ Else go back to wait loop
      LDR R0,=0x48200048       @ Address of INTC_CONTROL register
      MOV R1, #01              @ Value to clear bit 0
      STR R1,[R0]              @ Write to INTC_CONTROL register
      LDMFD SP!, {R0-R3,LR}    @ Restore registers
      SUBS PC, LR, #4          @ Pass execution to wait LOOP for now

      LDMFD SP!, {R0-R3,LR}    @ Restore registers
      SUBS PC, LR, #4          @ Pass execution to wait LOOP for now


BUTTON_SVC:
            MOV R1,#0x00004000      @ Value to turn off GPIO1_14 IRQ request
                                    @ This will turn off INTC IRQ request also
      STR R1,[R0]                   @ Write to GPIO1_IRQSTATUS_0 register
@ Turn on LED ON GPIO1_12
      LDR R0,=0x4804C194            @ Load address of GPIO1_SETDATAOUT register
      MOV R1, #0x00001000           @ Load value to turn on GPIO1_12
      STR R1,[R0]                   @ Write to GPIO1_SETDATAOUT register
      MOV R2,#0x03                  @ Load value to auto reload timer and start
      LDR R1,=0x48040038            @ address of Timer2 TCLR register
      STR R2,[R1]                   @ Write to TCLR register
@ Turn off NEWIRQA bit in INTC_CONTROL, so can respond to new IRQ
      LDR R0,=0x48200048            @ Address of INTC_CONTROL register
      MOV R1, #01                   @ Value to clear bit 0
      STR R1,[R0]                   @ Write to INTC_CONTROL register
      LDMFD SP!, {R0-R3,LR}         @ Restore registers
    SUBS PC, LR, #4      @ Pass execution on to wait LOOP for now
```



```
LED:
@ Turn off Timer 2 interrupt request and enable INTC for next IRQ
      LDR R1,= 0X48040028      @ Load address of Timer2 IRQSTATUS register
      MOV R2,#0x2              @ Value to reset Timer2 Overflow IRQ request
      STR R2,[R1]              @ Write
@ Toggle LED
      LDR R1,=0x4804C000       @ Base Address of GPIO1
      LDR R2, [R1,#0x013C]     @ Read value from GPIO1_DATAOUT
      TST R2,#0x1000           @ Check bit 12 where LED connected
      MOV R2,#0x1000           @ Value to set or clear bit 12
      BNE TOFF                 @ LED on, go turn off
      STR R2, [R1,#0x194]      @ LED off, turn on with GPIO1_SETDATAOUT
      B BACK                   @ Back to wait loop
TOFF:
    STR R2, [R1,#0x190] @ Turn LED off with GPIO1_CLEARDATAOUT
      BACK:
      LDR R1,=0x48200048       @ Address of INTC_CONTROL register
      MOV R2,#0x01             @ Value to enable new IRQ response in INTC
      STR R2,[R1]              @ Write
    LDMFD SP!, {R0-R3,LR}      @ Restore registers
    SUBS PC, LR, #4          @ Return from IRQ interrupt procedure

.data
.align 2
STACK1:     .rept 1024
            .word 0x0000
            .endr
STACK2:     .rept 1024
            .word 0x0000
            .endr

.END
```

Figure 5-18 Assembly language program for LED/Timer2 Interrupt Program.

An RS-232C Driver for an RC 8660 Speech Synthesizer Board

For the preceding interrupt application examples we described how you can service an external button press interrupt, and how you can count off time intervals on an interrupt basis. Another important use of interrupts is interfacing with external peripheral I/O controllers or with I/O controllers that are built into the current processors. Instead of polling a status register in one of these I/O controllers over and over to determine when it needs service, it is much more efficient to enable the device to generate an interrupt signal when it needs service and then service the device only when the interrupt occurs.

As an example of this type of interrupt application, we will show you how to send text messages to an RC Systems DoubleTalk RC8660 Speech Synthesis board on an interrupt basis. Specifically, this is an extension of the interrupt program in Figure 5-14 so that, instead of incrementing a count, pushing the button will cause a text message to be sent to the RC8660 board on an interrupt basis. We thought this application would add a little fun to your microprocessor learning curve and show you how you can easily make it possible for your microprocessor-controlled robot or some other embedded system to talk back to you. This example adds another level of complexity in that it shows you how to work with a controller that generates interrupt signals for several different internal conditions. You know more than enough about the use of interrupts from the previous examples to implement many interrupt-driven applications, so if you are short of time and want to get into the next chapter, you can study the introduction to RS-232C here because it is used for many systems, skim though the explanation for the interrupt service procedure, and then come back to pick up the details when you need them to help you write a driver procedure for a UART or some other peripheral controller. This example is a very good demonstration of how you work with the interrupt enable and status registers in a programmable peripheral device, so you should find it very useful if you do so. This example also shows you how to program the signal multiplexers that are used to map Sitara AM335X signals to device pads, so you can access them on the BeagleBone Black P8 and P9 connectors.

The RC8660 device used on the DoubleTalk 8660 Evaluation boards is a full-featured speech synthesis engine that implements sophisticated Text-To-Speech processing. The 8660 device allows a user to choose one of eight different standard voices; specify the speed, articulation, expression, reverb, pitch, tone and other features of the speech; generate up to three simultaneous tones; record sounds in an on-chip memory; and play back pre-recorded messages. The 8660 Evaluation Kit comes with the RC Studio software development package that runs on a PC and allows a user to interact with the board through a serial COM port on the PC to develop the desired functions in a Microsoft Windows environment. For this example we will send speech codes to the board directly through an NKC Electronics RS-232 serial adapter we added to a BeagleBone Black board and connected to one of the UARTS on a BeagleBone Black board. Working through the steps at this direct level will give you a good understanding of the details involved in a very simple I/O driver procedure as well as some more experience with interrupt procedures.

Rather than just show you the final program result, which is basically very similar to the previous examples except for some additional initialization steps and the interaction with the two different interrupts from the UART, we will use this example to show how you think and work your way through developing an I/O driver program such as this. We will list the basic steps and then work through them in some detail. We will leave the development of the full High-Level Algorithms, the full Low-Level Algorithms, and the final assembly language program for you as an end-of-chapter problem or lab exercise, instead of showing all of these in detail as we did for the Button/Timer example. The major steps and sections are as follows:

1. Read the data sheet for the RC8660 to determine the type of signal interface required for the RC8860 board. The board uses an RS-232C-type asynchronous serial interface. Finding this fact will lead you to step 2.
2. Find and read some basic background information on the RS-232C serial data transmission standard. (We will give you the information you need for this example in a section following this list.)

3. Determine the specific data format required by the device with which you are communicating. This information will include the Baud rate, number of data bits, number of stop bits, parity bit or no parity bit. Also note the RTS-CTS "handshaking" or "flow control" required to prevent the sending device from sending data faster than the receiving device can receive the data.
4. The serial COM port on your system will be driven by a UART. As discussed in Chapter 1, a UART is a Universal Asynchronous Receiver Transmitter that is used to convert parallel data to serial form and serial data to parallel form. Study the data sheet(s) for the UART(s) in the processor you are using to determine how you initialize the chosen UART for the required Baud rate, data format, and flow control you determined in step 3. Also determine how you enable the UART to generate interrupt signals when it is ready to transmit a character or receive a character.
5. Study the processor to determine how the interrupt signal from the UART is connected to the Interrupt Controller in the processor, the system level initialization needed to connect the UART signals to device pads, and the initialization required to turn on the UART clock.
6. Write the High-level initialization list for the system level steps needed to interface with the button and UART. Then develop the Low-Level algorithm and instructions for these steps.
7. Write the High-Level Algorithm for the UART initialization steps. Then write the Low-level algorithm and instructions for these steps. For this step you will need to carefully study the relevant UART registers.
8. Write the High-Level algorithm for INT_DIRECTOR PROCEDURE that decides if an interrupt came from the button or the UART. Then write the Low-Level algorithm and the instructions for this section.
9. Write the High-Level algorithm for the BUTTON_SVC section that enables a UART interrupt. Then write the Low-Level algorithm and the instructions for this section.
10. For the TALKER_SVC section, write the few first instructions that save registers and turn off the UART interrupt.
11. Build a program containing the sections you have written so far. Set a breakpoint on the first instruction in TALKER_SVC. Then run the program, push the button, and see if execution reaches the breakpoint in TALKER_SVC. If yes, celebrate. If not, debug until this much of the program works. You can use breakpoints to check inputs, outputs, and registers at each point along the way.
12. Study the RC 8660 data sheet to determine the format in which basic speech characters are sent to the speech processor. Write the algorithm for the section of the TALKER_SVC procedure that reads the characters from an array and sends them to the UART for transmission WHEN THE UART and the talker are ready. As you will see a little later, this involves checking the status of two different interrupts as well as keeping track of which character to send next. For this section you will need to carefully study a couple more of the UART registers and a flow chart we provide. Write the Low-Level algorithm and the instructions for this section and then write the assembly language instructions for this algorithm.
13. Add the TALKER_SVC section to your program, then build, test, and debug the complete program with the help of appropriate breakpoints.

Now that you have an overview of the development process, we will work through these steps and show you the new pieces you need for them to communicate with the RC 8660 Speech Processor.

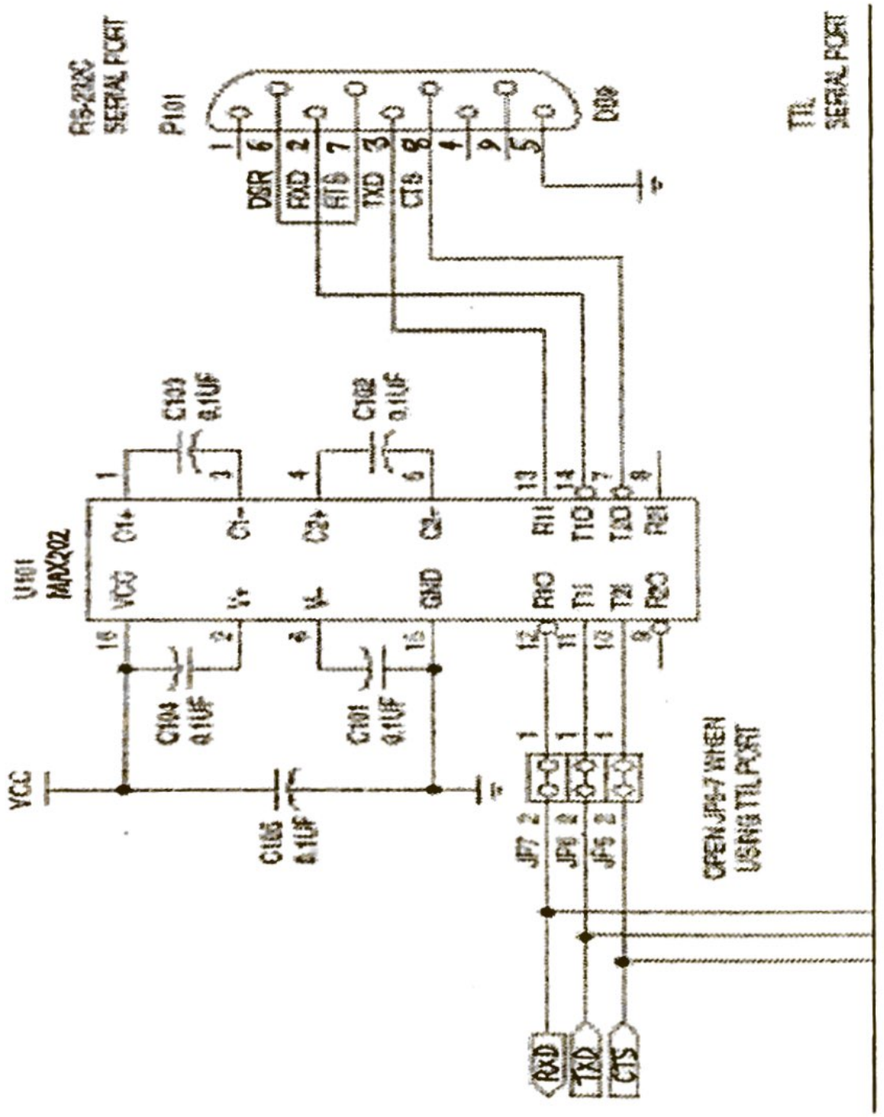


Figure 5-19. Serial Port connections on the RC Systems RC8660 Speech Synthesis Evaluation Board.

THE RC8660 SERIAL INTERFACE

Figure 5-19 shows the connections for the serial port on the RC 8660 board. In the figure first note that the interface only uses signals labeled RxD, TxD, and CTS. Second notice that the actual port is specified as RS-232C and uses a DB9 nine-pin connector. (The connector shows two additional signals, RTS and DSR that are just looped for compatibility with some systems but are not used in our example.) Finally, in Figure 5-19 note that a Maxim MAX202 device is used to convert the logic level signals on the RC8660 board to the voltage and assertion levels required for an RS-232C data link.

The data sheet for the RC 8660 board states that the board communicates using 8 data bits, no parity bit, one stop bit, and a Baud rate of up to 115,200 Baud. Furthermore, the RC 8660 can automatically determine the Baud rate for incoming data and adjust its UART timing accordingly, if a data word with the right bit pattern is sent as the first byte. The meaning of all these terms will be explained in the following sections.

ASYNCHRONOUS SERIAL DATA TRANSMISSION AND RS-232C CONNECTIONS

Figure 5-20 shows the format often used for sending asynchronous serial data. When no data is being sent, the line is in a constant high or *marking* state. The start of a data character is indicated by the line going low (the *space* state) for one bit time. This bit is called the *start bit*. The data bits for the character are then sent out on the line one after another, starting with the Least Significant Bit. Depending on the convention used in a particular system, the data word may consist of 5, 6, 7, or 8 data bits. An optional *parity bit* that can be used to detect single-bit errors in data words may be included after the data bits. Following the data bits and, if present, the parity bit, one or two always high *stop bits* are sent. The term *Baud Rate* is used to indicate the rate at which the data bits are being sent. Baud rate is defined as 1/(the time for one bit). A Baud rate of 56K Baud, for example, corresponds to a bit time of about 18 microseconds. However, Baud rate does not correspond to bits/sec because of the extra bits added to the basic data bits.

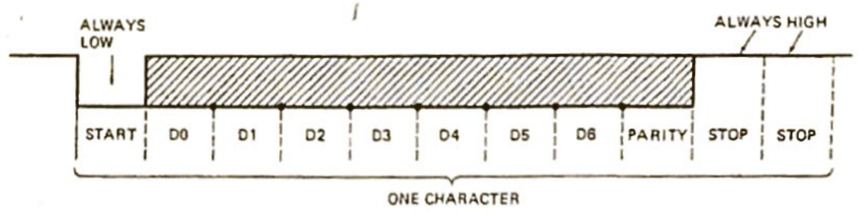


Figure 5-20 Bit format used for sending asynchronous serial data.

In the late 1960's as the use of timeshare computer terminals became more widespread, Modulator-Demodulators or MODEMs were developed so that remote terminals could use standard phone lines to communicate with mainframe computers. MODEMs or other devices used to convert the serial data and send it over the lines are referred to as *Data Communication Equipment* or *DCE*. The computer or terminal that is producing the data is referred to as *Data Terminal Equipment* or *DTE*. In response to the need for hardware handshaking or *flow control* standards for DCE and DTE, the Electronic Industries Association (EIA) developed EIA

standard RS-232C. This standard describes the function of 25 signal and handshake signals for serial data transfer. It also describes the voltage levels, impedance levels, signal rise and fall times, maximum Baud rate, and maximum capacitance for these lines. Although the RS-232C standard has been around for a relatively long time, it is still used in a large number of applications, so we will give you an introduction at the level you need to understand its use in current systems.

Figure 5-21a shows how a typical RS-232C data link is set up between a remote terminal or computer and a central server computer using two MODEMs. If you are still using a dial-up MODEM on your home computer to access the Internet, then the connection to the phone line is the same as that in Figure 5-21a (Cable MODEMs also contain a UART but interface with a router or computer with an Ethernet connection.) Figure 5-21b shows the RS-232C pin numbers, signal names, and signal directions. To start, we will discuss the "conversation" that takes place between DTE and DCE on these lines to transfer data from a terminal to the phone line. Then we will talk about the actual connectors, pin connections, and voltage levels.

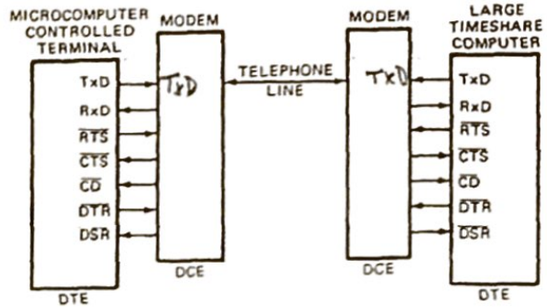
For this first discussion we will assume that full handshaking between the MODEM and a terminal or your computer is implemented. Handshaking signals are required to make sure the receiver is ready before the sender sends a data word. Most current systems only use two of the handshake signals defined by RS-232C.

After the power on the terminal or perhaps your computer is turned on, it asserts the Data Terminal Ready (DTR#) signal to tell the MODEM it is ready. When it is powered up and ready to transmit or receive data, the MODEM will then assert the Data Set Ready (DSR#) signal to the terminal. The MODEM then dials up the remote computer to establish a connection. In response, the MODEM on the remote computer will send back a specified tone sequence. Now, when the terminal has a character ready to send, it will assert the Request to Send (RTS#) signal to the MODEM. If the MODEM has received the carrier tone signal from the remote computer, it will assert the Carrier Detect (CD#) signal to the terminal. When the MODEM is fully ready to send data, it asserts the Clear To Send (CTS#) signal to the terminal. The terminal then sends serial data characters to the MODEM and the MODEM sends out the characters on the phone line one after the other. If the MODEM is not ready to send a character at any time, it can make the CTS# line to the terminal high to tell the terminal not to send another character. The important point here is that these hardware signals or a subset of them can be used for data *flow control* on a serial data link. Most current systems simply use RTS and CTS. Now let's look at RS-232C signal names and pins.

If the MODEM is not ready to send a character at any time, it can make the CTS# line to the terminal high to tell the terminal not to send another character. The important point here is that these hardware signals or a subset of them can be used for data *flow control* on a serial data link. Most current systems simply use RTS and CTS. Now let's look at RS-232C signal names and pins.

The left columns in Figure 5-21b show the connectors and pin numbers commonly used for RS-232C connections. The 25-pin connector was specified by the RS-232C standard but systems that only use a subset of the full 25-pin capability now use the smaller 9-pin connectors to save cost and space. The RS-232C standard specifies that a DTE device should use a male connector and that a DCE device should use a female connector. This convention makes it easy for you to tell just by looking at the connector whether a device is set up as DTE or DCE. (To test this, take a look at the DB-9 COM port connector on the back of a PC if present to see if it is set up as DTE or DCE.)

The common names of the RS-232C signals shown in Figure 5-21b are based on their



(a)

PIN NUMBERS FOR 9 PINS	PIN NUMBERS FOR 25 PINS	COMMON NAME	RS-232C NAME	DESCRIPTION	SIGNAL DIRECTION ON DCE
	1		AA	PROTECTIVE GROUND	–
3	2	TXD	BA	TRANSMITTED DATA	IN
2	3	RXD	BB	RECEIVED DATA	OUT
7	4	$\overline{RTS}$	CA	REQUEST TO SEND	IN
8	5	$\overline{CTS}$	CB	CLEAR TO SEND	OUT
6	6	$\overline{DSR}$	CC	DATA SET READY	OUT
5	7	GND	AB	SIGNAL GROUND (COMMON RETURN)	–
1	8	$\overline{CD}$	CF	RECEIVED LINE SIGNAL DETECTOR	OUT
	9		–	(RESERVED FOR DATA SET TESTING)	–
	10		–	(RESERVED FOR DATA SET TESTING)	–
	11			UNASSIGNED	–
	12		SCF	SECONDARY RECEIVED LINE SIGNAL DETECTOR	OUT
	13		SCB	SECONDARY CLEAR TO SEND	OUT
	14		SBA	SECONDARY TRANSMITTED DATA	IN
	15		DB	TRANSMISSION SIGNAL ELEMENT TIMING (DCE SOURCE)	OUT
	16		SBB	SECONDARY RECEIVED DATA	OUT
	17		DD	RECEIVER SIGNAL ELEMENT TIMING (DCE SOURCE)	OUT
	18			UNASSIGNED	–
	19		SCA	SECONDARY REQUEST TO SEND	IN
4	20	$\overline{DTR}$	CD	DATA TERMINAL READY	IN
	21		CG	SIGNAL QUALITY DETECTOR	OUT
9	22		CE	RING INDICATOR	OUT
	23		CH/CI	DATA SIGNAL RATE SELECTOR (DTE/DCE SOURCE)	IN/OUT
	24		DA	TRANSMIT SIGNAL ELEMENT TIMING (DTE SOURCE)	IN
	25			UNASSIGNED	–

(b)

Figure 5-21 RS-232C DCE and DTE connection and signals. (a) DTE and DCD connection for MODEM data link. (b) RS-232C pins, signals, and signal directions.

in Figure 5-21b. RS-232C connections often seem confusing but once you see that TxD is an output on DTE and an input on DCE, is makes sense that you simply connect Pin 2 on the DTE *straight through* to pin 2 on the DCE. Likewise, a straight through cable correctly connects all the other DTE to DCE signals. Note that as shown in Figure 5-20b, the pin numbers for the signals on a 9-pin connector are different from those on a 25-pin connector but again, a straight through cable will make the correct connections for a DTE-DCE

However, suppose you want to connect a DTE system to another DTE system or a DCE system to another DCE system with an RS-232C link. A straight through cable won't work because it will connect the TxD output on one DTE to the TxD output on the other DTE. RS-232C line drivers are designed so that this won't damage either output but obviously the data and handshake signals will not work correctly. The solution to this is to connect the two systems with a *Null-MODEM* connection such as that shown in Figure 5-22. The principle here is to simply cross the signal lines over so that, for example, the TxD from other DTE. You can buy Null-MODEM cables or you can buy a gender adapter and a Null-MODEM adapter that you can use to convert a straight-through

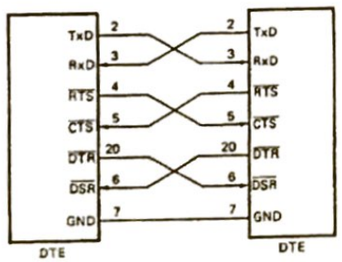


Figure 5-22 Null-MODEM connection for connecting two RS-232C DTE or two DCE type devices.

The final points to consider about RS-232C are the signal voltage levels and signal

The final points to consider about RS-232C are the signal voltage levels and assertion levels. The voltage levels for RS-232C signals are as follows. A logic high or *mark* is a voltage between –3 V and –15 V under load, or as much as –25 V with no load. A logic low or *space* is specified as a voltage between +3 V and + 15 V under load and as high as +25 V with no load. Common voltages found on RS-232C lines are ± 12 V, or ± 10 V. These voltage levels are obviously not compatible with the logic levels of current 3.3 V logic families and the assertion levels are inverted from the positive logic convention we use for most logic circuits. Therefore, a line driver/receiver device such as the MAX202 shown in Figure 5-19 is required to interface RS-

232C signals with standard logic devices. The MAX202 uses internal charge pumps to produce the positive and negative voltages required for RS-232C signal compatibility from a single +3.3 V supply. Inverters on each line in the MAX202 convert RS-232C signal assertion levels to standard logic assertion levels. For example, in Figure 5-19, when the CTS# signal from the UART on the RC8660 board is asserted as a standard logic low of near 0 V to the MAX202, the MAX202 will convert this signal to an RS-232C logic low level of about +10 V and apply it to pin 8 on the 9-pin connector. Incidentally, by studying Figure 5-19 and Figure 5-21b, you can see that the RC8660 board is set up as DCE because CTS# is an output on pin 8 of the 9-pin connector. We leave it to you to trace the TxD and RxD connections to further confirm this. For this example, we used an NKC Electronics KIT-0101 module that easily interfaces with a BeagleBone Black board and converts the 3.3V logic signals from the board to RS 232C level signals.

Now that you have an understanding of RS-232C signals and handshaking, the next step is to dig into the operation of one of the UARTs on a BeagleBone Black board and discuss the system level steps necessary to set up the processor so it can interact with the UART and the RC 8660 Talker on an interrupt basis.

INTRODUCTION TO THE SITARA AM335X UARTS

The Sitara AM335X processors have six built –in UARTs, UART0-UART5. Figure 5-23 shows how one of the UARTs with full handshake signals is set up as DTE and connected to a DCE device. The functional Clock, FCLK for a UART is 48MHz that is generated by dividing the system's 192 MHz PER_CLKOUTM2 signal by 4 in the PRCM. This clock will be divided down further to produce the Baud Rate clock. Each UART can also generate an IRQ interrupt signal to the processor INTC. As we discuss later, any one of several individual, internal conditions can assert the IRQ interrupt output if enabled in the UART to generate an interrupt. The UARTs also have *Direct Memory Access* or DMA.

In an overall computer system, Direct Memory Access capability means that, for example, a disk controller can independently send out the address and control signals necessary to transfer a block of data words directly from the controller to memory addresses. This is much faster than reading the data words from the controller with LDR instructions and writing them to memory with STR instructions. DMA also allows a block of data to be transferred directly to a disk drive or graphics card without going through the processor. Since DMA transfers are totally in hardware without the need for the processor to do read and write instructions for each data transfer, they are much faster than programmed I/O done with read and write instructions. For a UART, DMA capability means that you can write a block of data to a FIFO buffer in the UART and the control circuitry in the UART will automatically send out each data word when the receiver is ready. When the number of words left in the FIFO gets down to a programmed threshold level, the DMA controller sends a DMA request signal and/or an interrupt signal to the processor. In response to this, an interrupt procedure can download another block of data to the FIFO in the UART. For our RC 8660 Speech Processor example here, the number of data bytes to be sent is very small, so we will not use the DMA method. We will simply write one byte at a time to the UART on an interrupt basis. Now, we will discuss the system level program steps needed as part of the initialization for the "Talker" program.

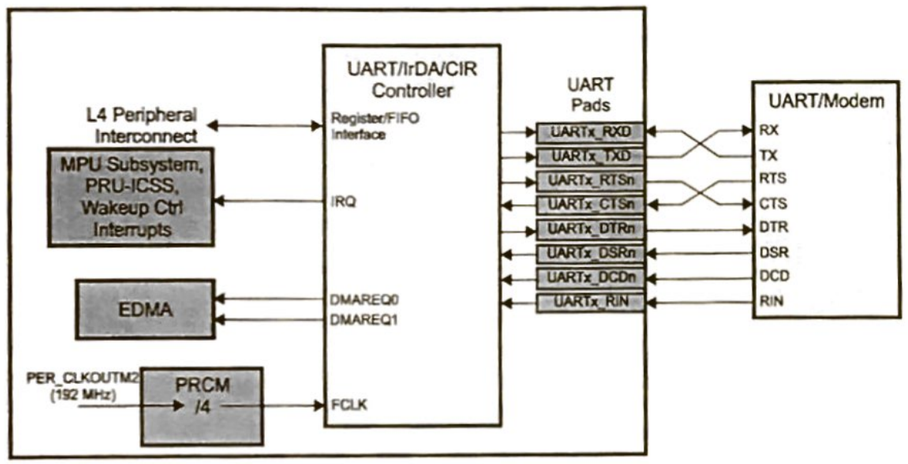


Figure 5-23 Sitara AM3359 UART Signals

SYSTEM LEVEL STEPS

For this example, we will be using the AM3358 UART5. The system level steps for initializing the processor to work with the button and UART5 are as follows:

1. Set up stacks as before.
2. Set up GPIO1 for falling edge of button signal on GPIO1_14 as before.
3. Initialize INTC for GPIO1_14 interrupt as before.
4. Initialize INTC for UART5 Interrupt. Consult Appendix E, ARM Cortex A-8 Interrupt Numbers to get interrupt number for UART5 and then map this number into the appropriate INTC_MIR_CLEARn register as with the GPIO1_14
5. Map the UART5 TxD, RxD, CTS and RTS to pins available on the BeagleBone Black P8 connector by changing the mode of the multiplexers that select the signals that go to the pads that are connected to P8 pins signals. When used as a Linux machine, the Multiplexers are initialized in MODE0 and the LCD signals are connected to pins as shown in the table below. These signals are used to drive the HDMI framer. Since we are not using the HDMI, we can switch the multiplexers so the UART5 signals go to pins as shown in the P8 PIN and NAME columns in the table.

P8 PIN	PROC	NAME	MODE 0 Assignments	
27	U5	GPIO2_22	lcd_vsync	
28	V5	GPIO2_24	lcd_pclk	
29	R5	GPIO2_23	lcd_hsync	
30	R6	GPIO2_25	lcd_ac_bias_en	
31	V4	UART5_CTSN	lcd_data14	Need to Switch to MODE 6

32	T5	UART5_RTSN	lcd_data15	Need to Switch to MODE 6
33	V3	UART4_RTSN	lcd_data13	
34	U4	UART3_RTSN	lcd_data11	
35	V2	UART4_CTSN	lcd_data12	
36	U3	UART3_CTSN	lcd_data10	
37	U1	UART5_TXD	lcd_data8	Need to Switch to MODE 4
38	U2	UART5_RXD	lcd_data9	Need Switch to MODE 4
39	T3	GPIO2_12	lcd_data6	
40	T4	GPIO2_13	lcd_data7	
41	T1	GPIO2_10	lcd_data4	
42	T2	GPIO2_11	lcd_data5	
43	R3	GPIO2_8	lcd_data2	
44	R4	GPIO2_9	lcd_data3	
45	R1	GPIO2_6	lcd_data0	
46	R2	GPIO2_7	lcd_data1	

The registers you use to change the mappings are in the Control Module. You go to the L4_WKUP Peripheral Memory Map in the ARM Cortex-A8 Memory Map to find the base address for the Control Module. Then go to this module to find the registers and offsets of the Control Module registers. The registers you need to change the mappings are identified by the lcd name in the MODE0 column. For a start, find the Control Module register offset for the lcd_data8 register that controls the signal that you want to be UART5_TXD as shown in the table above. Pop up the register for this and find the bits you use to set the desired mode. Note that the register format is the same for all of the many mapping registers, so they just show it once for all of them. Given the MODE information above and the base address for the control registers, you can write the instructions required to change each of the 4 signals to the desired P8 pins. Note that some of the signals are inputs and some are outputs, so you need to set the appropriate value in bit 5 of the control wood you send for each. Incidentally, the way I figured out the required modes for these signals was to use the TI Pin MUX Utility Program that is a free download from TI. Once you get the program running, you enter AM3359V2 and ZCZ package as directed. After the full display comes up click on the UART5 box to get all the MUX options displayed. In this display you should be able to see the required mode switches as shown above.

6. Turn on UART5 clock. The Base address for the Clock Module Peripheral Registers is 0x44E0_0000 and the offset of the CM_PER_UART5_CLKCTRL register is 0x38. As with turning on the Timer clock, you write 0x02 to this register.

Initializing UART5

Next you need to use the UART5 registers to initialize the UART for the desired Baud rate, etc. as needed for this program.
1. To find the base address for UART5, go to the Cortex A-8 memory Map in Appendix C. You should find 0x481A_A000.

2. Take a look at the UART register list in Appendix G. Note that several registers have the same offset. For example, the Transmit Holding Register, the Receiver Holding Register, and the Divisor Latch Low Register all have an offset of 0x0. The trick here is that the UART has three

different modes and different registers are accessible in different modes. The table below shows the three modes.

Mode	Condition
Configuration mode A	UART_LCR[7] = 0x1 and UART_LCR[7:0] != 0xBF
Configuration mode B	UART_LCR[7] = 0x1 and UART_LCR[7:0] = 0xBF
Operational mode	UART_LCR[7] = 0x0

As you can see, the desired mode is selected by writing the appropriate bit patterns in the Line Control Register, LCR. As shown in Appendix G, the LCR register is at offset 0x0C. To put the UART in operational mode, which is the default mode, you write a word with 0x0 in bit 7 to the LCR. If the UART is in Operational mode you do a read from the Receiver Holding Register at offset 0x0 to read a received character and to send a character, you write it to the Transmit Holding Register at offset 0x0. Internal circuitry determines whether the access is a read operation or a write operation and enables the appropriate register.

To switch to Configuration mode A, as you need to do to access the Divisor Latch Low and Divisor Latch High registers for setting the Baud rate, you send the LCR a word with bit 7 = 1 but with some value other than 0xBF in the lowest 8 bits of the word. To determine appropriate values for the lowest 7 bits in the word to be sent to the LCR register, take a look at the Register Field descriptions for the LCR register shown in Table 19-42 in the TI Sitara AM335X Technical Reference Manual. For the talker program, you want 8 data bits, one stop bit, and no parity. For these, Bit 3 will be 0, Bit 2 will be 0, and bits 1:0 will both be 1. With a 1 in bit 7, the word to send to the LCR register to switch to Mode A and set data format is 0x83.

3. Once switched to Mode A, you can access the Divisor Latch Low and Divisor Latch High registers to set the desired Baud rate. Figure 19-15 and Table 19-25 in the TI Sitara AM335X Technical Reference Manual show how to determine the values you will write to each of these registers. The 48 MHz clock is divided by the value written to the DLH and DLL registers and the result is then divided by further by 13 or 16. The second division depends on a bit you program in the Mode Definition Register 1, MDR1. For a 38.4 Kbps Baud rate and a 16x divisor, Table 19-25 shows that you want a DHL value of 0x00 and a DLL value of 0x4E.

4. The next step is to program the Mode Definition 1 Register to divide the output from the DHL-DLL divider by 16. Table 19-55 in the TI Sitara AM335X Technical Reference Manual shows the register field Descriptions for the MDR1 register. The Reset value for bits 2:0 is 111. You need to change these to 000 to program the UART for UART 16X mode. The Reset value of 0 for the rest of the bits are fine, so you can just write 0x00 to the MDR1 register.

5. The next step is to toggle the LCR register bit 7 to 0 to switch back to Operational Mode, so you can access the Transmit Holding Register and other UART5 registers you need to access to send data to the RC 8660 Talker on an interrupt basis. Along with 0 in bit 7, you want to keep the 8-bit data word length you programmed in the previous access to the LCR with bits 1:0 as 11, so you just write 0x03 to the LCR.

6. Now that you are back in Operational mode, you can enable two sources of interrupts in UART5. Specifically you want UART5 to generate and interrupt if the Transmit Holding Register is empty and ready for you to write the next byte to it. You also want UART5 to

generate an interrupt if the CTS# signal from the RC 8660 changes from low to high or high to low. You enable these two interrupt sources in the IER_UART Interrupt Enable Register at offset 0x04. You enable the THR by writing a 1 to bit 1, the THRIT bit and you enable the MODEM status interrupt by setting bit 3, the MODDEMSTSIT bit. The rest of the bits in the word can all be 0s, the reset values.

6. Finally in this UART5 initialization section, you need to make sure the TX_FIFO is cleared and disable the FIFO since you will not be using it. You do this with the FIFO Control Register at offset 0x8. Consult the **FCR Register Field Descriptions in Table 19-40** of the AM335X Technical Reference Manual to find the bits necessary to disable the RX FIFO, disable the TX FIFO and disable FIFO_EN.

7. Final step in the mainline section of the program is to enable the IRQ interrupt in the CPS register as you did in all of your other IRQ-based programs and execute an endless loop to wait for an interrupt.

The INT_DIRECTOR Section

The function of this program section is to determine if an IRQ request came from the UART or from the button being pushed. From a practical standpoint, it is best to check for a UART IRQ request first, since this request occurs more often than a button IRQ request. The general rule when chaining multiple IRQ request is to put those that occur most often or those that require the fastest servicing at the start of the list.

To check if IRQ is from UART5 you first check at the INTC level by checking the appropriate bit in the INTC_PENDING_IRQ1 register. If this is not set, you know the IRQ was not from UART5, so you go check if the IRQ request was from the button. If the UART5 bit was set in the INTC register, you go check the IIT bit of the IIR_UART register of UART5. If this bit is set, you go to the TLKR_SVC section to send a character, if all is ready. If not, you simply do the necessary cleanup and return to the wait loop.

To check if the IRQ request was from the button, you first check the appropriate bit in the INTC_PENDING_IRQ3 register. If this bit is set, the next step is to see if the interrupt came from the GPIO1 pin that is connected to the Button. To do this, you read the GPIO1_IRQSTATUS_0 register and check the appropriate bit, as shown in the previous examples in the chapter. If the bit is not set, you just do the required cleanup and return to the wait loop. If the bit is set, you go to the BUTTON_SVS section of the program.

The BUTTON_SVC Program Section

The only function of the BUTTON_SVC program section is to enable UART5 to generate interrupt signals, so characters can be sent to the RC 8660 on an interrupt basis. After turning off the GPIO1 Interrupt request from GPIO1 and the NEWIRQ bit in INTC, you enable the required UART5

Interrupts by writing a word to the UART5 Interrupt Enable Register, IER_UART register at offset 0x04. For this program you want a UART5 interrupt to be generated, if the Transmit Holding Register is empty and read to receive a new character or if the MODEM status has changed by CS going high or going low. If you check the IER_UART register field descriptions for the UART5 IER, you will see that setting bit 1 enables a THR interrupt and setting bit 3 enables a MODEM change interrupt. After these actions, you simply return to the wait loop.

The TALKER_SVC Section

Since we enabled both the MODEM Status Change interrupt and the Transmit Holding Register empty bits in the Interrupt Enable Register of UART5, either a change of the MODEM status or the THR becoming ready will cause an interrupt on the INTC and send execution to the TALKER_SVC procedure. However, because the interrupt could have been caused by either of two different conditions and both conditions have to be met before we can write a character to UART5, we can't simply send a character to the UART at this point. We first have to check that both CTS# is asserted low **and** THR is empty. To check if CTS# is asserted low, indicating that RC8660 is ready to receive a character, we read bit 4 of the Modem Status Register. Bit 4 will be a 1 if CTS# is asserted low as required for transmission. Incidentally, reading bit 4 will reset the bit, if it was a 1. To check if the THR is empty as required, we read the TXFIFOE bit of the Line Status Register, LSR_UART. If bit 5 of the LSR is a 1, then the THR is empty and ready for us to write a byte to the Transmit Holding Register, THR.

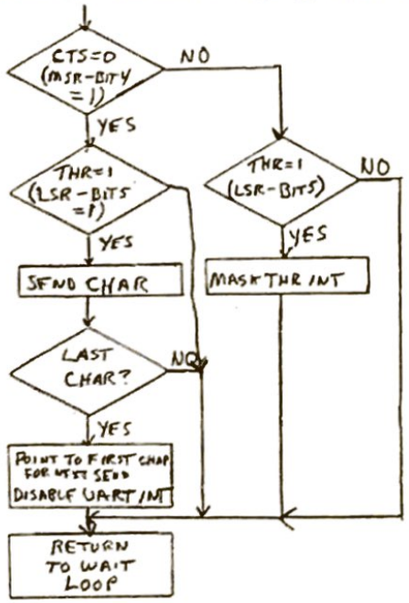


Figure 5-24 Execution flow for TLKR_SVC procedure control section.

Figure 5-24 shows the program flow through this section of the program. If we read the MSR bit 4 and find that CS# is asserted, we go on and check if the THR bit in the LSR =1. If so, all is well and we write a character to the UART5 THR. If it is the last character in the string, we reset the pointer to the first character in the string for the next send when the button is pushed again. We then disable the UART5 Interrupts by clearing bits 1 and 3 in the UART_IER register.

If we find that CTS# is not asserted when we read the MSR, we then read the LSR to find out if bit 5, the THR bit, is a 1. If it is, we disable the THR Interrupt by resetting the THR enable bit in the UART5's UART_IER Interrupt Enable Register. We do this to prevent the THR being ready from causing continuous interrupts or, as we often say, "spinning". If not prevented, this spinning would eliminate most of the advantage of using interrupts rather than polling to service the device. If the THR bit in the LSR is not set, we simply do the usual cleanup of the INTC and return to the Wait Loop in the mainline program to wait for another interrupt.

For basic text to speech conversion, you simply send the RC8660 an ASCII Carriage Return character, 0x0D, followed by a string of ASCII characters for the desired words, and a final 0x0D character. The syntax for this is:

```
.data
.align 2
MESSAGE:
.byte 0x0d
.ascii "Take me to your leader."
.byte 0x0d
.allign 2
mesage_length .word (length of your message)
```

The RC8660 uses the first 0x0D to determine the Baud rate and sets its internal clock for the frequency needed for the determined Baud rate. The final 0x0D character tells the RC8660 to speak the words contained in the ASCII string. The assembler will automatically insert the ASCII characters for any letters or numbers between double quotes as shown. The RC8660 data sheet describes how you can send control codes to change the voice, pitch, inflection, speed and other parameters of the speech.

This example has introduced you to the use of a UART controller to do RS-232C type data transmission to an RC8660 Speech Synthesis board. With relatively minor modifications and perhaps the addition of a receiver service procedure, you could use this program to communicate with a wide variety of other RS-232C modules and systems, including a mobile robot.

References

TI AM335X Sitara Technical Reference Manual,
http://www.ti.com/lit/ug/spruh73n/spruh73n.pdf

Checklist of Important Terms and Concepts

Interrupt Service Procedure
Exception handler
CPSR, SPSR
Privileged execution modes

Interrupt vector table
LDR PC, (address) instruction
Hooking an interrupt vector
Chaining an interrupt service procedure
Return from interrupt instructions
System idle process (loop)
Reentrant procedure
Recursive procedure
Nested interrupt procedure
Interrupt priority
Crystal Oscillator
Real Time Clock
Baud rate
Start bit, Stop bit, Parity bit
RS-232C Standard
MODEM
DTE, DCE
Data flow control
Straight through connection
Null-modem connection
RS-232C voltage and assertion levels

Review Questions and Problems

1. List and briefly describe the actions that an ARM-based processor will automatically do in response to an interrupt signal or exception.
2. In this chapter we showed you how to "hook" the IRQ interrupt vector in the Interrupt Vector Table. Show the ARM assembly language instructions you would use to hook the FIQ interrupt vector and save it in a memory location named System_FIQ.

3. Show how you would modify the INT_DIRECTOR procedure in Figure 5-3 to chain an additional procedure called KYBD_SVC. Assume that the interrupt for this procedure causes bit 3 in the INTPND register to be set when it occurs and that this interrupt occurs more often than the button interrupt.

4. Explain why the special instruction, `SUBS PC, LR, #4`, is required to return from an IRQ or FIQ interrupt procedure.

5. Briefly explain why it is a good practice to save R14 on the stack at the start of an interrupt procedure.

6. List and briefly describe the programming techniques required to make a procedure reentrant.

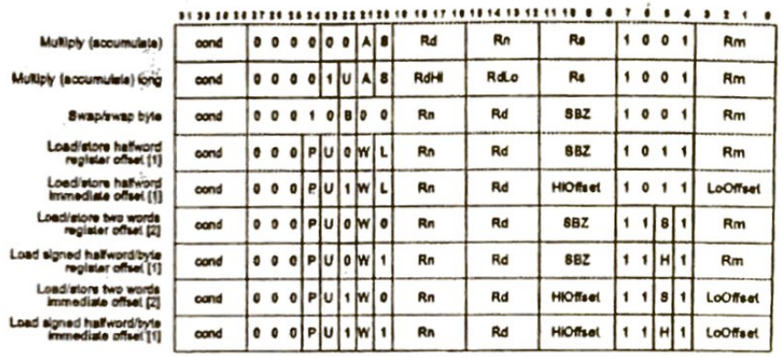
7. Write an ARM assembly language procedure that implements the FACTO algorithm given in Figure 5-6 for any number in the range of 1 to 100 using a simple loop algorithm.

8. Describe the additions that must be made to an IRQ interrupt procedure, so that it can be interrupted by a higher priority IRQ interrupt and still complete correctly when execution is returned to it at the end of the higher priority interrupt procedure.

9. (a) Briefly describe the major problems of using software delay loops to measure times in a microprocessor program.
   (b) Briefly explain how the problems given in part (a) are solved by using a crystal-controlled oscillator connected to an interrupt input to measure times in a microprocessor program.

10. If you have BeagleBone Black hardware and write, test, and debug some interface programs that are enhanced versions of the example programs in the chapter.

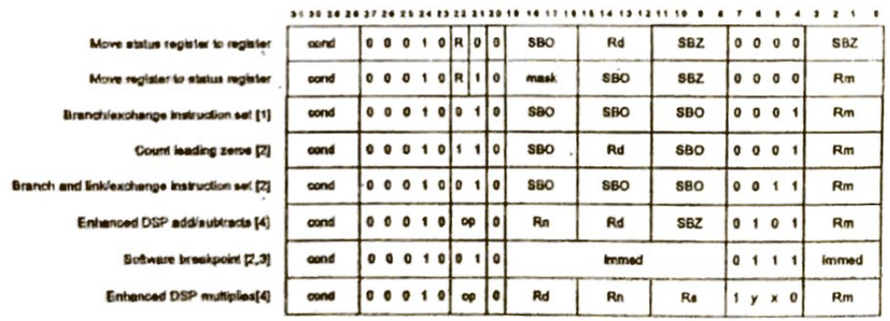
Multiplies and extra load/store instructions



	31 30 29 28 27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
Multiply (accumulate)	cond 0 0 0 0 0 0 A S	Rd	Rn	Rs	1 0 0 1	Rm
Multiply (accumulate) long	cond 0 0 0 0 1 U A S	RdHi	RdLo	Rs	1 0 0 1	Rm
Swp/swp byte	cond 0 0 0 1 0 B 0 0	Rn	Rd	SBZ	1 0 0 1	Rm
Load/store halfword register offset [1]	cond 0 0 0 P U 0 W L	Rn	Rd	SBZ	1 0 1 1	Rm
Load/store halfword immediate offset [1]	cond 0 0 0 P U 1 W L	Rn	Rd	HiOffset	1 0 1 1	LoOffset
Load/store two words register offset [2]	cond 0 0 0 P U 0 W 0	Rn	Rd	SBZ	1 1 S 1	Rm
Load signed halfword/byte register offset [1]	cond 0 0 0 P U 0 W 1	Rn	Rd	SBZ	1 1 H 1	Rm
Load/store two words immediate offset [2]	cond 0 0 0 P U 1 W 0	Rn	Rd	HiOffset	1 1 S 1	LoOffset
Load signed halfword/byte immediate offset [1]	cond 0 0 0 P U 1 W 1	Rn	Rd	HiOffset	1 1 H 1	LoOffset

3-2 Multiplies and extra load/store instructions

Miscellaneous instructions



	31 30 29 28 27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
Move status register to register	cond 0 0 0 1 0 R 0 0	SBO	Rd	SBZ	0 0 0 0	SBZ
Move register to status register	cond 0 0 0 1 0 R 1 0	mask	SBO	SBZ	0 0 0 0	Rm
Branch/exchange instruction set [1]	cond 0 0 0 1 0 0 1 0	SBO	SBO	SBO	0 0 0 1	Rm
Count leading zeros [2]	cond 0 0 0 1 0 1 1 0	SBO	Rd	SBO	0 0 0 1	Rm
Branch and link/exchange instruction set [2]	cond 0 0 0 1 0 0 1 0	SBO	SBO	SBO	0 0 1 1	Rm
Enhanced DSP add/subtracts [4]	cond 0 0 0 1 0 op 0	Rn	Rd	SBZ	0 1 0 1	Rm
Software breakpoint [2,3]	cond 0 0 0 1 0 0 1 0	Immed			0 1 1 1	immed
Enhanced DSP multiplies[4]	cond 0 0 0 1 0 op 0	Rd	Rn	Rs	1 y x 0	Rm

3-3 Miscellaneous instructions

1. Defined in ARM architecture version 5 and above, and in T variants of ARM architecture version 4.
2. This is an undefined instruction is ARM architecture version 4, and is UNPREDICTABLE prior to ARM architecture version 4.
3. If the cond field of this instruction is not 1110, it is UNPREDICTABLE.

Addressing Mode 3 - Miscellaneous Loads and Stores

There are six addressing modes used to calculate the address for load and store (signed or unsigned) halfword, load signed byte, or load and store doubleword instructions. The general instruction syntax is:

`LDR|STR{<cond>}H|SH|SB|D  <Rd>, <addressing_mode>`

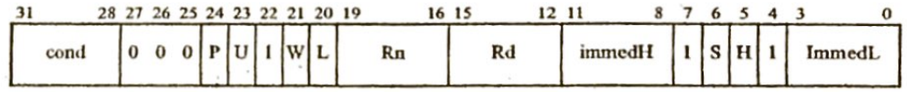
where `<addressing_mode>` is one of the following six options:

1. `[<Rn>, #+/-<offset_8>]`
   See *Miscellaneous Loads and Stores - Immediate offset* on page A5-36.
2. `[<Rn>, +/-<Rm>]`
   See *Miscellaneous Loads and Stores - Register offset* on page A5-38.
3. `[<Rn>, #+/-<offset_8>]!`
   See *Miscellaneous Loads and Stores - Immediate pre-indexed* on page A5-40.
4. `[<Rn>, +/-<Rm>]!`
   See *Miscellaneous Loads and Stores - Register pre-indexed* on page A5-42.
5. `[<Rn>], #+/-<offset_8>`
   See *Miscellaneous Loads and Stores - Immediate post-indexed* on page A5-44.
6. `[<Rn>], +/-<Rm>`
   See *Miscellaneous Loads and Stores - Register post-indexed* on page A5-46.

Encoding

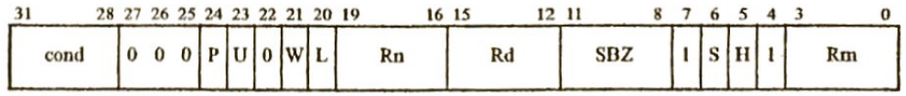
The following diagrams show the encodings for this addressing mode:

Immediate offset/index



31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	P	U	I	W	L	Rn	Rd	immedH	I	S	H	1	ImmedL

Register offset/index



31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	P	U	0	W	L	Rn	Rd	SBZ	I	S	H	1	Rm

The P bit Has two meanings:

P == 0 Indicates the use of *post-indexed addressing*. The base register value is used for the memory address, and the offset is then applied to the base register value and written back to the base register.

P == 1 Indicates the use of *offset addressing* or *pre-indexed addressing* (the W bit determines which). The memory address is generated by applying the offset to the base register value.

The U bit Indicates whether the offset is added to the base (U == 1) or subtracted from the base (U == 0).

The W bit Has two meanings:

P == 0 The W bit must be 0 or the instruction is UNPREDICTABLE.

P == 1 W == 1 indicates that the memory address is written back to the base register (pre-indexed addressing), and W == 0 that the base register is unchanged (offset addressing).

The L bit This bit distinguishes between a Load (L == 1) and a Store (L == 0) instruction.

The S bit This bit distinguishes between a signed (S == 1) and an unsigned (S == 0) halfword access.

The H bit This bit distinguishes between a halfword (H == 1) and a byte (H == 0) access.

Unsigned bytes

If S == 0 and H == 0, apparently indicating an unsigned byte, the instruction is not one that uses this addressing mode. Instead, it is a multiply instruction, a SWP or SWPB instruction, or an unallocated instruction in the arithmetic or load/store instruction extension space (see *Extending the instruction set* on page A3-27).

Unsigned bytes are accessed by the LDRB, LDRBT, STRB and STRBT instructions, which use addressing mode 2 rather than addressing mode 3.

Signed stores If S == 1 and L == 0, apparently indicating a signed store instruction, the instruction is an unallocated instruction in the load/store extension space (see *Extending the instruction set* on page A3-27).

Signed bytes and halfwords can be stored with the same STRB and STRH instructions as are used for unsigned quantities, so no separate signed store instructions are provided.

APPENDIX B – BRIEF EXPLANATIONS AND EXAMPLES OF BASIC ARM/XSCALE ASSEMBLY LANGUAGE INSTRUCTIONS

ARM® Instruction Set
Quick Reference Card

Key to Tables	
{cond}	Refer to Table **Condition Field (cond)**. Omit for unconditional execution.
<Operand2>	Refer to Table **Flexible Operand 2**. Shift and rotate are only available as part of Operand2.
<fields>	Refer to Table **PSR fields**.
<PSR>	Either CPSR (Current Processor Status Register) or SPSR (Saved Processor Status Register)
{S}	Updates condition flags if S present.
C*, V*	Flag is unpredictable in Architecture v4 and earlier, unchanged in Architecture v5 and later.
Q	Sticky flag. Always updates on overflow (no S option). Read and reset using MRS and MSR.
x,y	B meaning half-register [15:0], or T meaning [31:16].
<immed_8r>	A 32-bit constant, formed by right-rotating an 8-bit value by an even number of bits
<immed_8*4>	A 10-bit constant, formed by left-shifting an 8-bit value by two bits.

<a_mode2>	Refer to Table **Addressing Mode 2**.
<a_mode2P>	Refer to Table **Addressing Mode 2 (Post-indexed only)**.
<a_mode3>	Refer to Table **Addressing Mode 3**.
<a_mode4L>	Refer to Table **Addressing Mode 4 (Block load or Stack pop)**.
<a_mode4S>	Refer to Table **Addressing Mode 4 (Block store or Stack push)**.
<a_mode5>	Refer to Table **Addressing Mode 5**.
<reglist>	A comma-separated list of registers, enclosed in braces, { and }
{!}	Updates base register after data transfer if ! present.
+/-	+ or –. (+ may be omitted.)
§	Refer to Table **ARM architecture versions**.

Operation		§	Assembler	S updates	Q	Action
Move	Move		MOV{cond}{S} Rd, <Operand2>	N Z C		Rd := Operand2
	NOT		MVN{cond}{S} Rd, <Operand2>	N Z C		Rd := 0xFFFFFFFF EOR Operand2
	PSR to register	3	MRS{cond} Rd, <PSR>			Rd := PSR
	register to PSR	3	MSR{cond} <PSR>_<fields>, Rm			PSR := Rm (selected bytes only)
	immediate to PSR	3	MSR{cond} <PSR>_<fields>, #<immed_8r>			PSR := immed_8r (selected bytes only)
	40-bit accumulator to register	XS	MRA{cond} RdLo, RdHi, Ac			RdLo := Ac[31:0], RdHi := Ac[39:32]
	register to 40-bit accumulator	XS	MAR{cond} Ac, RdLo, RdHi			Ac[31:0] := RdLo, Ac[39:32] := RdHi
Arithmetic	Add		ADD{cond}{S} Rd, Rn, <Operand2>	N Z C V		Rd := Rn + Operand2
	with carry		ADC{cond}{S} Rd, Rn, <Operand2>	N Z C V		Rd := Rn + Operand2 + Carry
	saturating	5E	QADD{cond} Rd, Rm, Rn		Q	Rd := SAT(Rm + Rn)
	double saturating	5E	QDADD{cond} Rd, Rm, Rn		Q	Rd := SAT(Rm + SAT(Rn * 2))
	Subtract		SUB{cond}{S} Rd, Rn, <Operand2>	N Z C V		Rd := Rn – Operand2
	with carry		SBC{cond}{S} Rd, Rn, <Operand2>	N Z C V		Rd := Rn – Operand2 – NOT(Carry)
	reverse subtract		RSB{cond}{S} Rd, Rn, <Operand2>	N Z C V		Rd := Operand2 – Rn
	reverse subtract with carry		RSC{cond}{S} Rd, Rn, <Operand2>	N Z C V		Rd := Operand2 – Rn – NOT(Carry)
	saturating	5E	QSUB{cond} Rd, Rm, Rn		Q	Rd := SAT(Rm – Rn)
	double saturating	5E	QDSUB{cond} Rd, Rm, Rn		Q	Rd := SAT(Rm – SAT(Rn * 2))
	Multiply	2	MUL{cond}{S} Rd, Rm, Rs	N Z C*		Rd := (Rm * Rs)[31:0]
	accumulate	2	MLA{cond}{S} Rd, Rm, Rs, Rn	N Z C*		Rd := ((Rm * Rs) + Rn)[31:0]
	unsigned long	M	UMULL{cond}{S} RdLo, RdHi, Rm, Rs	N Z C* V*		RdHi,RdLo := unsigned(Rm * Rs)
	unsigned accumulate long	M	UMLAL{cond}{S} RdLo, RdHi, Rm, Rs	N Z C* V*		RdHi,RdLo := unsigned(RdHi,RdLo + Rm * Rs)
	signed long	M	SMULL{cond}{S} RdLo, RdHi, Rm, Rs	N Z C* V*		RdHi,RdLo := signed(Rm * Rs)
	signed accumulate long	M	SMLAL{cond}{S} RdLo, RdHi, Rm, Rs	N Z C* V*		RdHi,RdLo := signed(RdHi,RdLo + Rm * Rs)
	signed 16 * 16 bit	5E	SMULxy{cond} Rd, Rm, Rs			Rd := Rm[x] * Rs[y]
	signed 32 * 16 bit	5E	SMULWy{cond} Rd, Rm, Rs			Rd := (Rm * Rs[y])[47:16]
	signed accumulate 16 * 16 bit	5E	SMLAxy{cond} Rd, Rm, Rs, Rn		Q	Rd := Rn + Rm[x] * Rs[y]
	signed accumulate 32 * 16 bit	5E	SMLAWy{cond} Rd, Rm, Rs, Rn		Q	Rd := Rn + (Rm * Rs[y])[47:16]
	signed accumulate long 16 * 16 bit	5E	SMLALxy{cond} RdLo, RdHi, Rm, Rs			RdHi,RdLo := RdHi,RdLo + Rm[x] * Rs[y]
	Multiply with internal 40-bit accumulate	XS	MIA{cond} Ac, Rm, Rs			Ac := Ac + Rm * Rs
	packed halfword	XS	MIAPH{cond} Ac, Rm, Rs			Ac := Ac + Rm[15:0] * Rs[15:0] + Rm[31:16] * Rs[31:16]
	halfword	XS	MIAxy{cond} Ac, Rm, Rs			Ac := Ac + Rm[x] * Rs[y]
	Count leading zeroes	5	CLZ{cond} Rd, Rm			Rd := number of leading zeroes in Rm
Logical	Test		TST{cond} Rn, <Operand2>	N Z C		Update CPSR flags on Rn AND Operand2
	Test equivalence		TEQ{cond} Rn, <Operand2>	N Z C		Update CPSR flags on Rn EOR Operand2
	AND		AND{cond}{S} Rd, Rn, <Operand2>	N Z C		Rd := Rn AND Operand2
	EOR		EOR{cond}{S} Rd, Rn, <Operand2>	N Z C		Rd := Rn EOR Operand2
	ORR		ORR{cond}{S} Rd, Rn, <Operand2>	N Z C		Rd := Rn OR Operand2
	Bit Clear		BIC{cond}{S} Rd, Rn, <Operand2>	N Z C		Rd := Rn AND NOT Operand2
Compare	Compare		CMP{cond} Rn, <Operand2>	N Z C V		Update CPSR flags on Rn – Operand2
	negative		CMN{cond} Rn, <Operand2>	N Z C V		Update CPSR flags on Rn + Operand2
No Op	No operation		NOP			None

ARM Addressing Modes
Quick Reference Card

Addressing Mode 2 - Word and Unsigned Byte Data Transfer

Pre-indexed	Immediate offset	`[Rn, #+/-<immed_12>]{!}`	
	Zero offset	`[Rn]`	Equivalent to [Rn,#0]
	Register offset	`[Rn, +/-Rm]{!}`	
	Scaled register offset	`[Rn, +/-Rm, LSL #<immed_5>]{!}`	Allowed shifts 0-31
		`[Rn, +/-Rm, LSR #<immed_5>]{!}`	Allowed shifts 1-32
		`[Rn, +/-Rm, ASR #<immed_5>]{!}`	Allowed shifts 1-32
		`[Rn, +/-Rm, ROR #<immed_5>]{!}`	Allowed shifts 1-31
		`[Rn, +/-Rm, RRX]{!}`	
Post-indexed	Immediate offset	`[Rn], #+/-<immed_12>`	
	Register offset	`[Rn], +/-Rm`	
	Scaled register offset	`[Rn], +/-Rm, LSL #<immed_5>`	Allowed shifts 0-31
		`[Rn], +/-Rm, LSR #<immed_5>`	Allowed shifts 1-32
		`[Rn], +/-Rm, ASR #<immed_5>`	Allowed shifts 1-32
		`[Rn], +/-Rm, ROR #<immed_5>`	Allowed shifts 1-31
		`[Rn], +/-Rm, RRX`	

Addressing Mode 2 (Post-indexed only)

Post-indexed	Immediate offset	`[Rn], #+/-<immed_12>`	
	Zero offset	`[Rn]`	Equivalent to [Rn],#0
	Register offset	`[Rn], +/-Rm`	
	Scaled register offset	`[Rn], +/-Rm, LSL #<immed_5>`	Allowed shifts 0-31
		`[Rn], +/-Rm, LSR #<immed_5>`	Allowed shifts 1-32
		`[Rn], +/-Rm, ASR #<immed_5>`	Allowed shifts 1-32
		`[Rn], +/-Rm, ROR #<immed_5>`	Allowed shifts 1-31
		`[Rn], +/-Rm, RRX`	

Addressing Mode 3 - Halfword, Signed Byte, and Doubleword Data Transfer

Pre-indexed	Immediate offset	`[Rn, #+/-<immed_8>]{!}`	
	Zero offset	`[Rn]`	Equivalent to [Rn,#0]
	Register	`[Rn, +/-Rm]{!}`	
Post-indexed	Immediate offset	`[Rn], #+/-<immed_8>`	
	Register	`[Rn], +/-Rm`	

Addressing Mode 4 - Multiple Data Transfer

Block load		Stack pop	
IA	Increment After	FD	Full Descending
IB	Increment Before	ED	Empty Descending
DA	Decrement After	FA	Full Ascending
DB	Decrement Before	EA	Empty Ascending
Block store		**Stack push**	
IA	Increment After	EA	Empty Ascending
IB	Increment Before	FA	Full Ascending
DA	Decrement After	ED	Empty Descending
DB	Decrement Before	FD	Full Descending

Addressing Mode 5 - Coprocessor Data Transfer

Pre-indexed	Immediate offset	`[Rn, #+/-<immed_8*4>]{!}`	
	Zero offset	`[Rn]`	Equivalent to [Rn,#0]
Post-indexed	Immediate offset	`[Rn], #+/-<immed_8*4>`	
Unindexed	No offset	`[Rn], {8-bit copro. option}`	

ARM architecture versions

n	ARM architecture version *n* and above.
*n*T	T variants of ARM architecture version *n* and above.
M	ARM architecture version 3M, and 4 and above, except xM variants.
*n*E	All E variants of ARM architecture version *n* and above.
*n*E*	E variants of ARM architecture version *n* and above, except xP variants.
XS	XScale coprocessor instruction

Flexible Operand 2

Immediate value	`#<immed_8r>`	
Logical shift left immediate	`Rm, LSL #<immed_5>`	Allowed shifts 0-31
Logical shift right immediate	`Rm, LSR #<immed_5>`	Allowed shifts 1-32
Arithmetic shift right immediate	`Rm, ASR #<immed_5>`	Allowed shifts 1-32
Rotate right immediate	`Rm, ROR #<immed_5>`	Allowed shifts 1-31
Register	`Rm`	
Rotate right extended	`Rm, RRX`	
Logical shift left register	`Rm, LSL Rs`	
Logical shift right register	`Rm, LSR Rs`	
Arithmetic shift right register	`Rm, ASR Rs`	
Rotate right register	`Rm, ROR Rs`	

PSR fields (use at least one suffix)

Suffix	Meaning	
c	Control field mask byte	PSR[7:0]
f	Flags field mask byte	PSR[31:24]
s	Status field mask byte	PSR[23:16]
x	Extention field mask byte	PSR[15:8]

Condition Field (cond)

Mnemonic	Description	Description (VFP)
EQ	Equal	Equal
NE	Not equal	Not equal, or unordered
CS / HS	Carry Set / Unsigned higher or same	Greater than or equal, or unordered
CC / LO	Carry Clear / Unsigned lower	Less than
MI	Negative	Less than
PL	Positive or zero	Greater than or equal, or unordered
VS	Overflow	Unordered (at least one NaN operand)
VC	No overflow	Not unordered
HI	Unsigned higher	Greater than, or unordered
LS	Unsigned lower or same	Less than or equal
GE	Signed greater than or equal	Greater than or equal
LT	Signed less than	Less than, or unordered
GT	Signed greater than	Greater than
LE	Signed less than or equal	Less than or equal, or unordered
AL	Always (normally omitted)	Always (normally omitted)

ARM Instruction Set
Quick Reference Card

Operation		§	Assembler	Action	Notes
Branch	Branch		`B{cond} label`	R15 := label	label must be within ±32Mb of current instruction.
	with link		`BL{cond} label`	R14 := R15 – 4, R15 := label	label must be within ±32Mb of current instruction.
	and exchange	4T,5	`BX{cond} Rm`	R15 := Rm, Change to Thumb if Rm[0] is 1	
	with link and exchange (1)	5T	`BLX label`	R14 := R15 – 4, R15 := label, Change to Thumb	Cannot be conditional.
	with link and exchange (2)	5	`BLX{cond} Rm`	R14 := R15 – 4, R15 := Rm[31:1] Change to Thumb if Rm[0] is 1	label must be within ±32Mb of current instruction.
Load	Word		`LDR{cond} Rd, <a_mode2>`	Rd := [address]	Rd must not be R15.
	User mode privilege		`LDR{cond}T Rd, <a_mode2P>`		Rd must not be R15.
	branch (§ 5T: and exchange)		`LDR{cond} R15, <a_mode2>`	R15 := [address][31:1] (§ 5T: Change to Thumb if [address][0] is 1)	
	Byte		`LDR{cond}B Rd, <a_mode2>`	Rd := ZeroExtend[byte from address]	Rd must not be R15.
	User mode privilege		`LDR{cond}BT Rd, <a_mode2P>`		Rd must not be R15.
	signed	4	`LDR{cond}SB Rd, <a_mode3>`	Rd := SignExtend[byte from address]	Rd must not be R15.
	Halfword	4	`LDR{cond}H Rd, <a_mode3>`	Rd := ZeroExtent[halfword from address]	Rd must not be R15.
	signed	4	`LDR{cond}SH Rd, <a_mode3>`	Rd := SignExtend[halfword from address]	Rd must not be R15.
	Doubleword	5E*	`LDR{cond}D Rd, <a_mode3>`	Rd := [address], R(d+1) := [address + 4]	Rd must be even, and not R14.
Load multiple	Pop, or Block data load		`LDM{cond}<a_mode4L> Rn{!}, <reglist-pc>`	Load list of registers from [Rn]	
	return (and exchange)		`LDM{cond}<a_mode4L> Rn{!}, <reglist+pc>`	Load registers, R15 := [address][31:1] (§ 5T: Change to Thumb if [address][0] is 1)	
	and restore CPSR		`LDM{cond}<a_mode4L> Rn{!}, <reglist+pc>^`	Load registers, branch (§ 5T: and exchange), CPSR := SPSR	Use from exception modes only.
	User mode registers		`LDM{cond}<a_mode4L> Rn, <reglist-pc>^`	Load list of User mode registers from [Rn]	Use from privileged modes only.
Soft preload	Memory system hint	5E*	`PLD <a_mode2>`	Memory may prepare to load from address	Cannot be conditional.
Store	Word		`STR{cond} Rd, <a_mode2>`	[address] := Rd	
	User mode privilege		`STR{cond}T Rd, <a_mode2P>`	[address] := Rd	
	Byte		`STR{cond}B Rd, <a_mode2>`	[address][7:0] := Rd[7:0]	
	User mode privilege		`STR{cond}BT Rd, <a_mode2P>`	[address][7:0] := Rd[7:0]	
	Halfword	4	`STR{cond}H Rd, <a_mode3>`	[address][15:0] := Rd[15:0]	
	Doubleword	5E*	`STR{cond}D Rd, <a_mode3>`	[address] := Rd, [address + 4] := R(d+1)	Rd must be even, and not R14.
Store multiple	Push, or Block data store		`STM{cond}<a_mode4S> Rn{!}, <reglist>`	Store list of registers to [Rn]	
	User mode registers		`STM{cond}<a_mode4S> Rn{!}, <reglist>^`	Store list of User mode registers to [Rn]	Use from privileged modes only.
Swap	Word	3	`SWP{cond} Rd, Rm, [Rn]`	temp := [Rn], [Rn] := Rm, Rd := temp	
	Byte	3	`SWP{cond}B Rd, Rm, [Rn]`	temp := ZeroExtend([Rn][7:0]), [Rn][7:0] := Rm[7:0], Rd := temp	
Coprocessors	Data operations	2	`CDP{cond} <copr>, <op1>, CRd, CRn, CRm{, <op2>}`	Coprocessor defined	
	Alternative operations	5	`CDP2 <copr>, <op1>, CRd, CRn, CRm{, <op2>}`		Cannot be conditional.
	Move to ARM reg from coproc	2	`MRC{cond} <copr>, <op1>, Rd, CRn, CRm{, <op2>}`		
	Alternative moves	5	`MRC2 <copr>, <op1>, Rd, CRn, CRm{, <op2>}`		Cannot be conditional.
	Two ARM register move	5E*	`MRRC{cond} <copr>, <op1>, Rd, Rn, CRm`		
	Move to coproc from ARM reg	2	`MCR{cond} <copr>, <op1>, Rd, CRn, CRm{, <op2>}`		
	Alternative moves	5	`MCR2 <copr>, <op1>, Rd, CRn, CRm{, <op2>}`		Cannot be conditional.
	Two ARM register move	5E*	`MCRR{cond} <copr>, <op1>, Rd, Rn, CRm`		
	Load	2	`LDC{cond} <copr>, CRd, <a_mode5>`		
	Alternative loads	5	`LDC2 <copr>, CRd, <a_mode5>`		Cannot be conditional.
	Store	2	`STC{cond} <copr>, CRd, <a_mode5>`		
	Alternative stores	5	`STC2 <copr>, CRd, <a_mode5>`		Cannot be conditional.
Software interrupt			`SWI{cond} <immed_24>`	Software interrupt processor exception	24-bit value encoded in instruction.
Breakpoint		5	`BKPT <immed_16>`	Prefetch abort *or* enter debug state	Cannot be conditional.

Appendix C:- ARM cortex-A8 Addresses

L3 Memory Map

Block Name	Start_address (hex)	End_address (hex)	Size	Description
GPMC (External Memory)	0x0000_0000[1]	0x1FFF_FFFF	512MB	8-/16-bit External Memory (Ex/R/W)[2]
Reserved	0x2000_0000	0x3FFF_FFFF	512MB	Reserved
Boot ROM	0x4000_0000	0x4001_FFFF	128KB	
	0x4002_0000	0x4002_BFFF	48KB	32-bit Ex/R[2] – Public
Reserved	0x4002_C000	0x400F_FFFF	848KB	Reserved
Reserved	0x4010_0000	0x401F_FFFF	1MB	Reserved
Reserved	0x4020_0000	0x402E_FFFF	960KB	Reserved
Reserved	0x402F_0000	0x402F_03FF	64KB	Reserved
SRAM internal	0x402F_0400	0x402F_FFFF		32-bit Ex/R/W[2]
L3 OCMC0	0x4030_0000	0x4030_FFFF	64KB	32-bit Ex/R/W[2] OCMC SRAM
Reserved	0x4031_0000	0x403F_FFFF	960KB	Reserved
Reserved	0x4040_0000	0x4041_FFFF	128KB	Reserved
Reserved	0x4042_0000	0x404F_FFFF	896KB	Reserved
Reserved	0x4050_0000	0x405F_FFFF	1MB	Reserved
Reserved	0x4060_0000	0x407F_FFFF	2MB	Reserved
Reserved	0x4080_0000	0x4083_FFFF	256KB	Reserved
Reserved	0x4084_0000	0x40DF_FFFF	5888KB	Reserved
Reserved	0x40E0_0000	0x40E0_7FFF	32KB	Reserved
Reserved	0x40E0_8000	0x40EF_FFFF	992KB	Reserved
Reserved	0x40F0_0000	0x40F0_7FFF	32KB	Reserved
Reserved	0x40F0_8000	0x40FF_FFFF	992KB	Reserved
Reserved	0x4100_0000	0x41FF_FFFF	16MB	Reserved
Reserved	0x4200_0000	0x43FF_FFFF	32MB	Reserved
L3F CFG Regs	0x4400_0000	0x443F_FFFF	4MB	L3Fast configuration registers
Reserved	0x4440_0000	0x447F_FFFF	4MB	Reserved
L3S CFG Regs	0x4480_0000	0x44BF_FFFF	4MB	L3Slow configuration registers
L4_WKUP	0x44C0_0000	0x44FF_FFFF	4MB	L4_WKUP
Reserved	0x4500_0000	0x45FF_FFFF	16MB	Reserved
McASP0 Data	0x4600_0000	0x463F_FFFF	4MB	McASP0 Data Registers
McASP1 Data	0x4640_0000	0x467F_FFFF	4MB	McASP1 Data Registers
Reserved	0x4680_0000	0x46FF_FFFF	8MB	Reserved
Reserved	0x4700_0000	0x473F_FFFF	4MB	Reserved

(1) The first 1MB of address space 0x0-0xFFFFF is inaccessible externally.
(2) Ex/R/W – Execute/Read/Write.

L3 memory map (continued)

Block Name	Start_address (hex)	End_address (hex)	Size	Description
USBSS	0x4740_0000	0x4740_0FFF	20KB	USB Subsystem Registers
USB0	0x4740_1000	0x4740_12FF		USB0 Controller Registers
USB0_PHY	0x4740_1300	0x4740_13FF		USB0 PHY Registers
USB0 Core	0x4740_1400	0x4740_17FF		USB0 Core Registers
USB1	0x4740_1800	0x4740_1AFF		USB1 Controller Registers
USB1_PHY	0x4740_1B00	0x4740_1BFF		USB1 PHY Registers
USB1 Core	0x4740_1C00	0x4740_1FFF		USB1 Core Registers
USB CPPI DMA Controller	0x4740_2000	0x4740_2FFF		USB CPPI DMA Controller Registers
USB CPPI DMA Scheduler	0x4740_3000	0x4740_3FFF		USB CPPI DMA Scheduler Registers
USB Queue Manager	0x4740_4000	0x4740_4FFF		USB Queue Manager Registers
Reserved	0x4740_5000	0x477F_FFFF	4MB-20KB	Reserved
Reserved	0x4780_0000	0x4780_FFFF	64KB	Reserved
MMCHS2	0x4781_0000	0x4781_FFFF	64KB	MMCHS2
Reserved	0x4782_0000	0x47BF_FFFF	4MB-128KB	Reserved
Reserved	0x47C0_0000	0x47FF_FFFF	4MB	Reserved
L4_PER	0x4800_0000	0x48FF_FFFF	16MB	L4 Peripheral (see L4_PER table)
TPCC (EDMA3CC)	0x4900_0000	0x490F_FFFF	1MB	EDMA3 Channel Controller Registers
Reserved	0x4910_0000	0x497F_FFFF	7MB	Reserved
TPTC0 (EDMA3TC0)	0x4980_0000	0x498F_FFFF	1MB	EDMA3 Transfer Controller 0 Registers
TPTC1 (EDMA3TC1)	0x4990_0000	0x499F_FFFF	1MB	EDMA3 Transfer Controller 1 Registers
TPTC2 (EDMA3TC2)	0x49A0_0000	0x49AF_FFFF	1MB	EDMA3 Transfer Controller 2 Registers
Reserved	0x49B0_0000	0x49BF_FFFF	1MB	Reserved
Reserved	0x49C0_0000	0x49FF_FFFF	4MB	Reserved
L4_FAST	0x4A00_0000	0x4AFF_FFFF	16MB	L4_FAST
Reserved	0x4B00_0000	0x4B13_FFFF	1280KB	Reserved
Reserved	0x4B14_0000	0x4B15_FFFF	128KB	Reserved
DebugSS_DRM	0x4B16_0000	0x4B16_0FFF	4KB	Debug Subsystem: Debug Resource Manager
DebugSS_ETB	0x4B16_2000	0x4B16_2FFF	4KB	Debug Subsystem: Embedded Trace Buffer
Reserved	0x4B16_3000	0x4BFF_FFFF	15MB-396KB	Reserved
EMIF0	0x4C00_0000	0x4CFF_FFFF	16MB	EMIF0 Configuration registers
Reserved	0x4D00_0000	0x4DFF_FFFF	16MB	Reserved
Reserved	0x4E00_0000	0x4FFF_FFFF	32MB	Reserved
GPMC	0x5000_0000	0x50FF_FFFF	16MB	GPMC Configuration registers
Reserved	0x5100_0000	0x52FF_FFFF	32MB	Reserved
Reserved	0x5300_0000	0x530F_FFFF	1MB	Reserved
	0x5310_0000	0x531F_FFFF	1MB	Reserved
Reserved	0x5320_0000	0x533F_FFFF	2MB	Reserved
Reserved	0x5340_0000	0x534F_FFFF	1MB	Reserved
	0x5350_0000	0x535F_FFFF	1MB	Reserved
Reserved	0x5360_0000	0x54BF_FFFF	22MB	Reserved
ADC_TSC DMA	0x54C0_0000	0x54FF_FFFF	4MB	ADC_TSC DMA Port
Reserved	0x5500_0000	0x55FF_FFFF	16MB	Reserved

L3 Memory Map (Continued)

Block Name	Start_address (hex)	End_address (hex)	Size	Description
SGX530	0x5600_0000	0x56FF_FFFF	16MB	SGX530 Slave Port
Reserved	0x5700_0000	0x57FF_FFFF	16MB	Reserved
Reserved	0x5800_0000	0x58FF_FFFF	16MB	Reserved
Reserved	0x5900_0000	0x59FF_FFFF	16MB	Reserved
Reserved	0x5A00_0000	0x5AFF_FFFF	16MB	Reserved
Reserved	0x5B00_0000	0x5BFF_FFFF	16MB	Reserved
Reserved	0x5C00_0000	0x5DFF_FFFF	32MB	Reserved
Reserved	0x5E00_0000	0x5FFF_FFFF	32MB	Reserved
Reserved	0x6000_0000	0x7FFF_FFFF	512MB	Reserved
EMIF0 SDRAM	0x8000_0000	0xBFFF_FFFF	1GB	8-/16-bit External Memory (Ex/R/W)[3]
Reserved	0xC000_0000	0xFFFF_FFFF	1GB	Reserved

[3] Ex/R/W – Execute/Read/Write

L4_WKUP Peripheral Memory Map

Region Name	Start Address (hex)	End Address (hex)	Size	Description
L4_WKUP configuration	0x44C0_0000	0x44C0_07FF	2KB	Address/Protection (AP)
	0x44C0_0800	0x44C0_0FFF	2KB	Link Agent (LA)
	0x44C0_1000	0x44C0_13FF	1KB	Initiator Port (IP0)
	0x44C0_1400	0x44C0_17FF	1KB	Initiator Port (IP1)
Reserved	0x44C0_1800	0x44C0_1FFF	2KB	Reserved (IP2 – IP3)
Reserved	0x44C0_2000	0x44CF_FFFF	1MB-8KB	Reserved
Reserved	0x44D0_0000	0x44D0_3FFF	16KB	Reserved
	0x44D0_4000	0x44D0_4FFF	4KB	Reserved
Reserved	0x44D0_5000	0x44D7_FFFF	492KB	Reserved
Reserved	0x44D8_0000	0x44D8_1FFF	8KB	Reserved
	0x44D8_2000	0x44D8_2FFF	4KB	Reserved
Reserved	0x44D8_3000	0x44DF_FFFF	500KB	Reserved
CM_PER	0x44E0_0000	0x44E0_3FFF	1KB	Clock Module Peripheral Registers
CM_WKUP	0x44E0_0400	0x44E0_04FF	256 Bytes	Clock Module Wakeup Registers
CM_DPLL	0x44E0_0500	0x44E0_05FF	256 Bytes	Clock Module PLL Registers
CM_MPU	0x44E0_0600	0x44E0_06FF	256 Bytes	Clock Module MPU Registers
CM_DEVICE	0x44E0_0700	0x44E0_07FF	256 Bytes	Clock Module Device Registers
CM_RTC	0x44E0_0800	0x44E0_08FF	256 Bytes	Clock Module RTC Registers
CM_GFX	0x44E0_0900	0x44E0_09FF	256 Bytes	Clock Module Graphics Controller Registers
CM_CEFUSE	0x44E0_0A00	0x44E0_0AFF	256 Bytes	Clock Module Efuse Registers
PRM_IRQ	0x44E0_0B00	0x44E0_0BFF	256 Bytes	Power Reset Module Interrupt Registers
PRM_PER	0x44E0_0C00	0x44E0_0CFF	256 Bytes	Power Reset Module Peripheral Registers
PRM_WKUP	0x44E0_0D00	0x44E0_0DFF	256 Bytes	Power Reset Module Wakeup Registers
PRM_MPU	0x44E0_0E00	0x44E0_0EFF	256 Bytes	Power Reset Module MPU Registers
PRM_DEV	0x44E0_0F00	0x44E0_0FFF	256 Bytes	Power Reset Module Device Registers
PRM_RTC	0x44E0_1000	0x44E0_10FF	256 Bytes	Power Reset Module RTC Registers

L4_WKUP Peripheral Memory Map (continued)

Region Name	Start Address (hex)	End Address (hex)	Size	Description
PRM_GFX	0x44E0_1100	0x44E0_11FF	256 Bytes	Power Reset Module Graphics Controller Registers
PRM_CEFUSE	0x44E0_1200	0x44E0_12FF	256 Bytes	Power Reset Module Efuse Registers
Reserved	0x44E0_3000	0x44E0_3FFF	4KB	Reserved
	0x44E0_4000	0x44E0_4FFF	4KB	Reserved
DMTIMER0	0x44E0_5000	0x44E0_5FFF	4KB	DMTimer0 Registers
	0x44E0_6000	0x44E0_6FFF	4KB	Reserved
GPIO0	0x44E0_7000	0x44E0_7FFF	4KB	GPIO Registers
	0x44E0_8000	0x44E0_8FFF	4KB	Reserved
UART0	0x44E0_9000	0x44E0_9FFF	4KB	UART Registers
	0x44E0_A000	0x44E0_AFFF	4KB	Reserved
I2C0	0x44E0_B000	0x44E0_BFFF	4KB	I2C Registers
	0x44E0_C000	0x44E0_CFFF	4KB	Reserved
ADC_TSC	0x44E0_D000	0x44E0_EFFF	8KB	ADC_TSC Registers
	0x44E0_F000	0x44E0_FFFF	4KB	Reserved
Control Module	0x44E1_0000	0x44E1_1FFF	128KB	Control Module Registers
DDR2/3/mDDR PHY	0x44E1_2000	0x44E1_23FF		DDR2/3/mDDR PHY Registers
Reserved	0x44E1_2400	0x44E3_0FFF	4KB	Reserved
DMTIMER1_1MS (Accurate 1ms timer)	0x44E3_1000	0x44E3_1FFF	4KB	DMTimer1 1ms Registers
	0x44E3_2000	0x44E3_2FFF	4KB	Reserved
Reserved	0x44E3_3000	0x44E3_3FFF	4KB	Reserved
	0x44E3_4000	0x44E3_4FFF	4KB	Reserved
WDT1	0x44E3_5000	0x44E3_5FFF	4KB	Watchdog Timer Registers
	0x44E3_6000	0x44E3_6FFF	4KB	Reserved
SmartReflex0	0x44E3_7000	0x44E3_7FFF	4KB	L3 Registers
	0x44E3_8000	0x44E3_8FFF	4KB	Reserved
SmartReflex1	0x44E3_9000	0x44E3_9FFF	4KB	L3 Registers
	0x44E3_A000	0x44E3_AFFF	4KB	Reserved
Reserved	0x44E3_B000	0x44E3_DFFF	12KB	Reserved
RTCSS	0x44E3_E000	0x44E3_EFFF	4KB	RTC Registers
	0x44E3_F000	0x44E3_FFFF	4KB	Reserved
DebugSS Instrumentation HWMaster1 Port	0x44E4_0000	0x44E7_FFFF	256KB	Debug Registers
	0x44E8_0000	0x44E8_0FFF	4KB	Reserved
Reserved	0x44E8_1000	0x44EF_FFFF	508KB	Reserved
Reserved	0x44F0_0000	0x44FF_FFFF	1MB	Reserved

L4_PER Peripheral Memory Map

Device Name	Start_address (hex)	End_address (hex)	Size	Description
Reserved	0x4800_0000	0x4800_07FF	2KB	Reserved
	0x4800_0800	0x4800_0FFF	2KB	Reserved
	0x4800_1000	0x4800_13FF	1KB	Reserved
	0x4800_1400	0x4800_17FF	1KB	Reserved
	0x4800_1800	0x4800_1BFF	1KB	Reserved
	0x4800_1C00	0x4800_1FFF	1KB	Reserved

L4_PER Peripheral Memory Map(continued)

Device Name	Start_address (hex)	End_address (hex)	Size	Description
Reserved	0x4800_2000	0x4800_3FFF	8KB	Reserved
Reserved	0x4800_4000	0x4800_7FFF	16KB	Reserved
Reserved	0x4800_8000	0x4800_8FFF	4KB	Reserved
	0x4800_9000	0x4800_9FFF	4KB	Reserved
Reserved	0x4800_A000	0x4800_FFFF	24KB	Reserved
Reserved	0x4801_0000	0x4801_0FFF	4KB	Reserved
	0x4801_1000	0x4801_1FFF	4KB	Reserved
Reserved	0x4801_2000	0x4801_3FFF	8KB	Reserved
Reserved	0x4801_4000	0x4801_FFFF	48KB	Reserved
Reserved	0x4802_0000	0x4802_0FFF	4KB	Reserved
	0x4802_1000	0x4802_1FFF	4KB	Reserved
UART1	0x4802_2000	0x4802_2FFF	4KB	UART1 Registers
	0x4802_3000	0x4802_3FFF	4KB	Reserved
UART2	0x4802_4000	0x4802_4FFF	4KB	UART2 Registers
	0x4802_5000	0x4802_5FFF	4KB	Reserved
Reserved	0x4802_6000	0x4802_7FFF	8KB	Reserved
Reserved	0x4802_8000	0x4802_8FFF	4KB	Reserved
	0x4802_9000	0x4802_9FFF	4KB	Reserved
I2C1	0x4802_A000	0x4802_AFFF	4KB	I2C1 Registers
	0x4802_B000	0x4802_BFFF	4KB	Reserved
Reserved	0x4802_C000	0x4802_CFFF	4KB	Reserved
	0x4802_D000	0x4802_DFFF	4KB	Reserved
Reserved	0x4802_E000	0x4802_EFFF	4KB	Reserved
	0x4802_F000	0x4802_FFFF	4KB	Reserved
McSPI0	0x4803_0000	0x4803_0FFF	4KB	McSPI0 Registers
	0x4803_1000	0x4803_1FFF	4KB	Reserved
Reserved	0x4803_2000	0x4803_2FFF	4KB	Reserved
	0x4803_3000	0x4803_3FFF	4KB	Reserved
Reserved	0x4803_4000	0x4803_4FFF	4KB	Reserved
	0x4803_5000	0x4803_5FFF	4KB	Reserved
Reserved	0x4803_6000	0x4803_6FFF	4KB	Reserved
	0x4803_7000	0x4803_7FFF	4KB	Reserved
McASP0 CFG	0x4803_8000	0x4803_9FFF	8KB	McASP0 CFG Registers
	0x4803_A000	0x4803_AFFF	4KB	Reserved
Reserved	0x4803_B000	0x4803_BFFF	4KB	Reserved
McASP1 CFG	0x4803_C000	0x4803_0FFF	8KB	McASP1 CFG Registers
	0x4803_E000	0x4803_EFFF	4KB	Reserved
Reserved	0x4803_F000	0x4803_FFFF	4KB	Reserved
DMTIMER2	0x4804_0000	0x4804_0FFF	4KB	DMTimer2 Registers
	0x4804_1000	0x4804_1FFF	4KB	Reserved
DMTIMER3	0x4804_2000	0x4804_2FFF	4KB	DMTimer3 Registers
	0x4804_3000	0x4804_3FFF	4KB	Reserved
DMTIMER4	0x4804_4000	0x4804_4FFF	4KB	DMTimer4 Registers
	0x4804_5000	0x4804_5FFF	4KB	Reserved
DMTIMER5	0x4804_6000	0x4804_6FFF	4KB	DMTimer5 Registers
	0x4804_7000	0x4804_7FFF	4KB	Reserved
DMTIMER6	0x4804_8000	0x4804_8FFF	4KB	DMTimer6 Registers

L4_PER Peripheral Memory Map(continued)

Device Name	Start_address (hex)	End_address (hex)	Size	Description
	0x4804_9000	0x4804_9FFF	4KB	L4 Interconnect
DMTIMER7	0x4804_A000	0x4804_AFFF	4KB	DMTimer7 Registers
	0x4804_B000	0x4804_BFFF	4KB	Reserved
GPIO1	0x4804_C000	0x4804_CFFF	4KB	GPIO1 Registers
	0x4804_D000	0x4804_DFFF	4KB	Reserved
Reserved	0x4804_E000	0x4804_FFFF	8KB	Reserved
Reserved	0x4805_0000	0x4805_FFFF	64KB	Reserved
MMCHS0	0x4806_0000	0x4806_0FFF	4KB	MMCHS0 Registers
	0x4806_1000	0x4806_1FFF	4KB	Reserved
Reserved	0x4806_2000	0x4807_FFFF	120KB	Reserved
ELM	0x4808_0000	0x4808_FFFF	64KB	ELM Registers
	0x4809_0000	0x4809_0FFF	4KB	Reserved
Reserved	0x4809_1000	0x4809_FFFF	60KB	Reserved
Reserved	0x480A_0000	0x480A_FFFF	64KB	Reserved
	0x480B_0000	0x480B_0FFF	4KB	Reserved
Reserved	0x480B_1000	0x480B_FFFF	60KB	Reserved
Reserved	0x480C_0000	0x480C_0FFF	4KB	Reserved
	0x480C_1000	0x480C_1FFF	4KB	Reserved
Reserved	0x480C_2000	0x480C_2FFF	4KB	Reserved
	0x480C_3000	0x480C_3FFF	4KB	Reserved
Reserved	0x480C_4000	0x480C_7FFF	16KB	Reserved
Mailbox 0	0x480C_8000	0x480C_8FFF	4KB	Mailbox Registers
	0x480C_9000	0x480C_9FFF	4KB	Reserved
Spinlock	0x480C_A000	0x480C_AFFF	4KB	Spinlock Registers
	0x480C_B000	0x480C_BFFF	4KB	Reserved
Reserved	0x480C_C000	0x480F_FFFF	208KB	Reserved
Reserved	0x4810_0000	0x4811_FFFF	128KB	Reserved
	0x4812_0000	0x4812_0FFF	4KB	Reserved
Reserved	0x4812_1000	0x4812_1FFF	4KB	Reserved
Reserved	0x4812_2000	0x4812_2FFF	4KB	Reserved
	0x4812_3000	0x4812_3FFF	4KB	Reserved
Reserved	0x4812_4000	0x4813_FFFF	112KB	Reserved
Reserved	0x4814_0000	0x4815_FFFF	128KB	Reserved
	0x4816_0000	0x4816_0FFF	4K	Reserved
Reserved	0x4816_1000	0x4817_FFFF	124KB	Reserved
Reserved	0x4818_0000	0x4818_2FFF	12KB	Reserved
	0x4818_3000	0x4818_3FFF	4KB	Reserved
Reserved	0x4818_4000	0x4818_7FFF	16KB	Reserved
Reserved	0x4818_8000	0x4818_8FFF	4KB	Reserved
	0x4818_9000	0x4818_9FFF	4KB	Reserved
Reserved	0x4818_A000	0x4818_AFFF	4KB	Reserved
	0x4818_B000	0x4818_BFFF	4KB	Reserved
OCP Watchpoint	0x4818_C000	0x4818_CFFF	4KB	OCP Watchpoint Registers
	0x4818_D000	0x4818_DFFF	4KB	Reserved
Reserved	0x4818_E000	0x4818_EFFF	4KB	Reserved
	0x4818_F000	0x4818_FFFF	4KB	Reserved
Reserved	0x4819_0000	0x4819_0FFF	4KB	Reserved

L4_PER Peripheral Memory Map(continued)

Device Name	Start_address (hex)	End_address (hex)	Size	Description
	0x4819_1000	0x4819_1FFF	4KB	Reserved
Reserved	0x4819_2000	0x4819_2FFF	4KB	Reserved
	0x4819_3000	0x4819_3FFF	4KB	Reserved
Reserved	0x4819_4000	0x4819_BFFF	32KB	Reserved
I2C2	0x4819_C000	0x4819_CFFF	4KB	I2C2 Registers
	0x4819_D000	0x4819_DFFF	4KB	Reserved
Reserved	0x4819_E000	0x4819_EFFF	4KB	Reserved
	0x4819_F000	0x4819_FFFF	4KB	Reserved
McSPI1	0x481A_0000	0x481A_0FFF	4KB	McSPI1 Registers
	0x481A_1000	0x481A_1FFF	4KB	Reserved
Reserved	0x481A_2000	0x481A_5FFF	16KB	Reserved
UART3	0x481A_6000	0x481A_6FFF	4KB	UART3 Registers
	0x481A_7000	0x481A_7FFF	4KB	Reserved
UART4	0x481A_8000	0x481A_8FFF	4KB	UART4 Registers
	0x481A_9000	0x481A_9FFF	4KB	Reserved
UART5	0x481A_A000	0x481A_AFFF	4KB	UART5 Registers
	0x481A_B000	0x481A_BFFF	4KB	Reserved
GPIO2	0x481A_C000	0x481A_CFFF	4KB	GPIO2 Registers
	0x481A_D000	0x481A_DFFF	4KB	Reserved
GPIO3	0x481A_E000	0x481A_EFFF	4KB	GPIO3 Registers
	0x481A_F000	0x481A_FFFF	4KB	Reserved
Reserved	0x481B_0000	0x481B_FFFF	64KB	Reserved
	0x481C_0000	0x481C_0FFF	4KB	Reserved
Reserved	0x481C_1000	0x481C_1FFF	4KB	Reserved
	0x481C_2000	0x481C_2FFF	4KB	Reserved
Reserved	0x481C_3000	0x481C_9FFF	28KB	Reserved
Reserved	0x481C_A000	0x481C_AFFF	4KB	Reserved
	0x481C_B000	0x481C_BFFF	4KB	Reserved
DCAN0	0x481C_C000	0x481C_DFFF	8KB	DCAN0 Registers
	0x481C_E000	0x481C_FFFF	8KB	Reserved
DCAN1	0x481D_0000	0x481D_1FFF	8KB	DCAN1 Registers
	0x481D_2000	0x481D_3FFF	8KB	Reserved
Reserved	0x481D_4000	0x481D_4FFF	4KB	Reserved
	0x481D_5000	0x481D_5FFF	4KB	Reserved
Reserved	0x481D_6000	0x481D_6FFF	4KB	Reserved
	0x481D_7000	0x481D_7FFF	4KB	Reserved
MMC1	0x481D_8000	0x481D_8FFF	4KB	MMC1 Registers
	0x481D_9000	0x481D_9FFF	4KB	Reserved
Reserved	0x481D_A000	0x481F_FFFF	152KB	Reserved
Interrupt controller (INTCPS)	0x4820_0000	0x4820_0FFF	4KB	Interrupt Controller Registers
Reserved	0x4820_1000	0x4823_FFFF	252KB	Reserved
MPUSS config register	0x4824_0000	0x4824_0FFF	4KB	Host ARM non-shared device mapping
Reserved	0x4824_1000	0x4827_FFFF	252KB	Reserved
Reserved	0x4828_0000	0x4828_0FFF	4KB	Reserved
Reserved	0x4828_1000	0x482F_FFFF	508KB	Reserved

L4_PER Peripheral Memory Map(continued)

Device Name	Start_address (hex)	End_address (hex)	Size	Description
PWM Subsystem 0	0x4830_0000	0x4830_00FF	4KB	PWMSS0 Configuration Registers
eCAP0	0x4830_0100	0x4830_017F		PWMSS eCAP0 Registers
eQEP0	0x4830_0180	0x4830_01FF		PWMSS eQEP0 Registers
ePWM0	0x4830_0200	0x4830_025F		PWMSS ePWM0 Registers
	0x4830_0260	0x4830_1FFF	4KB	Reserved
PWM Subsystem 1	0x4830_2000	0x4830_20FF	4KB	PWMSS1 Configuration Registers
eCAP1	0x4830_2100	0x4830_217F		PWMSS eCAP1 Registers
eQEP1	0x4830_2180	0x4830_21FF		PWMSS eQEP1 Registers
ePWM1	0x4830_2200	0x4830_225F		PWMSS ePWM1 Registers
	0x4830_2260	0x4830_3FFF	4KB	Reserved
PWM Subsystem 2	0x4830_4000	0x4830_40FF	4KB	PWMSS2 Configuration Registers
eCAP2	0x4830_4100	0x4830_417F		PWMSS eCAP2 Registers
eQEP2	0x4830_4180	0x4830_41FF		PWMSS eQEP2 Registers
ePWM2	0x4830_4200	0x4830_425F		PWMSS ePWM2 Registers
	0x4830_4260	0x4830_5FFF	4KB	Reserved
Reserved	0x4830_6000	0x4830_DFFF	32KB	Reserved
LCD Controller	0x4830_E000	0x4830_EFFF	4KB	LCD Registers
	0x4830_F000	0x4830_FFFF	4KB	Reserved
Reserved	0x4831_0000	0x4831_1FFF	8KB	Reserved
	0x4831_2000	0x4831_2FFF	4KB	Reserved
Reserved	0x4831_3000	0x4831_7FFF	20KB	Reserved
Reserved	0x4831_8000	0x4831_BFFF	16KB	Reserved
	0x4831_C000	0x4831_CFFF	4KB	Reserved
Reserved	0x4831_D000	0x4831_FFFF	12KB	Reserved
Reserved	0x4832_0000	0x4832_5FFF	16KB	Reserved
Reserved	0x4832_6000	0x48FF_FFFF	13MB-152KB	Reserved

Appendix D:- ARM Cortex-A8 Interrupt Numbers

Int Number	Acronym/name	Source	Signal Name
0	EMUINT	MPU Subsystem Internal	Emulation interrupt (EMUICINTR)
1	COMMTX	MPU Subsystem Internal	CortexA8 COMMTX
2	COMMRX	MPU Subsystem Internal	CortexA8 COMMRX
3	BENCH	MPU Subsystem Internal	CortexA8 NPMUIRQ
4	ELM_IRQ	ELM	Sinterrupt (Error location process completion)
5	Reserved		
6	Reserved		
7	NMI	External Pin (active low)(1)	nmi_int
8	Reserved		
9	L3DEBUG	L3	l3_FlagMux_top_FlagOut1
10	L3APPINT	L3	l3_FlagMux_top_FlagOut0
11	PRCMINT	PRCM	irq_mpu
12	EDMACOMPINT	TPCC (EDMA)	tpcc_int_pend_po0
13	EDMAMPERR	TPCC (EDMA)	tpcc_mpint_pend_po
14	EDMAERRINT	TPCC (EDMA)	tpcc_errint_pend_po
15	Reserved		
16	ADC_TSC_GENINT	ADC_TSC (Touchscreen Controller)	gen_intr_pend
17	USBSSINT	USBSS	usbss_intr_pend
18	USBINT0	USBSS	usb0_intr_pend
19	USBINT1	USBSS	usb1_intr_pend
20	PRU_ICSS_EVTOUT0	pr1_host[0] output/events exported from PRU-ICSS(2)	pr1_host_intr0_intr_pend
21	PRU_ICSS_EVTOUT1	pr1_host[1] output/events exported from PRU-ICSS(2)	pr1_host_intr1_intr_pend
22	PRU_ICSS_EVTOUT2	pr1_host[2] output/events exported from PRU-ICSS(2)	pr1_host_intr2_intr_pend
23	PRU_ICSS_EVTOUT3	pr1_host[3] output/events exported from PRU-ICSS(2)	pr1_host_intr3_intr_pend
24	PRU_ICSS_EVTOUT4	pr1_host[4] output/events exported from PRU-ICSS(2)	pr1_host_intr4_intr_pend
25	PRU_ICSS_EVTOUT5	pr1_host[5] output/events exported from PRU-ICSS(2)	pr1_host_intr5_intr_pend
26	PRU_ICSS_EVTOUT6	pr1_host[6] output/events exported from PRU-ICSS(2)	pr1_host_intr6_intr_pend
27	PRU_ICSS_EVTOUT7	pr1_host[7] output/events exported from PRU-ICSS(2)	pr1_host_intr7_intr_pend
28	MMCSD1INT	MMCSD1	SINTERRUPTN
29	MMCSD2INT	MMCSD2	SINTERRUPTN
30	I2C2INT	I2C2	POINTRPEND
31	eCAP0INT	eCAP0 event/interrupt	ecap_intr_intr_pend
32	GPIOINT2A	GPIO 2	POINTRPEND1
33	GPIOINT2B	GPIO 2	POINTRPEND2
34	USBWAKEUP	USBSS	slv0p_Swakeup
35	Reserved		

(1) For differences in operation based on AM335x silicon revisions, see Silicon Revision Functional Differences and Enhancements, *Silicon Revision Functional Differences and Enhancements*.

(2) pr1_host_intr[0:7] corresponds to Host-2 to Host-9 of the PRU-ICSS interrupt controller.

ARM Cortex-A8 Interrupt Numbers(Continued)

Int Number	Acronym/name	Source	Signal Name
36	LCDCINT	LCDC	lcd_irq
37	GFXINT	SGX530	THALIAIRQ
38	Reserved		
39	ePWM2INT	eHRPWM2 (PWM Subsystem)	epwm_intr_intr_pend
40	3PGSWRXTHR0 (RX_THRESH_PULSE)	CPSW (Ethernet)	c0_rx_thresh_pend
41	3PGSWRXINT0 (RX_PULSE)	CPSW (Ethernet)	c0_rx_pend
42	3PGSWTXINT0 (TX_PULSE)	CPSW (Ethernet)	c0_tx_pend
43	3PGSWMISC0 (MISC_PULSE)	CPSW (Ethernet)	c0_misc_pend
44	UART3INT	UART3	niq
45	UART4INT	UART4	niq
46	UART5INT	UART5	niq
47	eCAP1INT	eCAP1 (PWM Subsystem)	ecap_intr_intr_pend
48	Reserved		
49	Reserved		
50	Reserved		
51	Reserved		
52	DCAN0_INT0	DCAN0	dcan_intr0_intr_pend
53	DCAN0_INT1	DCAN0	dcan_intr1_intr_pend
54	DCAN0_PARITY	DCAN0	dcan_uerr_intr_pend
55	DCAN1_INT0	DCAN1	dcan_intr0_intr_pend
56	DCAN1_INT1	DCAN1	dcan_intr1_intr_pend
57	DCAN1_PARITY	DCAN1	dcan_uerr_intr_pend
58	ePWM0_TZINT	eHRPWM0 TZ interrupt (PWM Subsystem)	epwm_tz_intr_pend
59	ePWM1_TZINT	eHRPWM1 TZ interrupt (PWM Subsystem)	epwm_tz_intr_pend
60	ePWM2_TZINT	eHRPWM2 TZ interrupt (PWM Subsystem)	epwm_tz_intr_pend
61	eCAP2INT	eCAP2 (PWM Subsystem)	ecap_intr_intr_pend
62	GPIOINT3A	GPIO 3	POINTRPEND1
63	GPIOINT3B	GPIO 3	POINTRPEND2
64	MMCSD0INT	MMCSD0	SINTERRUPTN
65	McSPI0INT	McSPI0	SINTERRUPTN
66	TINT0	Timer0	POINTR_PEND
67	TINT1_1MS	DMTIMER_1ms	POINTR_PEND
68	TINT2	DMTIMER2	POINTR_PEND
69	TINT3	DMTIMER3	POINTR_PEND
70	I2C0INT	I2C0	POINTRPEND
71	I2C1INT	I2C1	POINTRPEND
72	UART0INT	UART0	niq
73	UART1INT	UART1	niq
74	UART2INT	UART2	niq
75	RTCINT	RTC	timer_intr_pend
76	RTCALARMINT	RTC	alarm_intr_pend
77	MBINT0	Mailbox0 (mail_u0_irq)	initiator_sinterrupt_q_n0
78	M3_TXEV	Wake M3 Subsystem	TXEV
79	eQEP0INT	eQEP0 (PWM Subsystem)	eqep_intr_intr_pend
80	MCATXINT0	McASP0	mcasp_x_intr_pend

ARM Cortex-A8 Interrupt Numbers(Continued)

Int Number	Acronym/name	Source	Signal Name
81	MCARXINT0	McASP0	mcasp_r_intr_pend
82	MCATXINT1	McASP1	mcasp_x_intr_pend
83	MCARXINT1	McASP1	mcasp_r_intr_pend
84	Reserved		
85	Reserved		
86	ePWM0INT	eHRPWM0 (PWM Subsystem)	epwm_intr_intr_pend
87	ePWM1INT	eHRPWM1 (PWM Subsystem)	epwm_intr_intr_pend
88	eQEP1INT	eQEP1 (PWM Subsystem)	eqep_intr_intr_pend
89	eQEP2INT	eQEP2 (PWM Subsystem)	eqep_intr_intr_pend
90	DMA_INTR_PIN2	External DMA/Interrupt Pin2 (xdma_event_intr2)	pi_x_dma_event_intr2
91	WDT1INT (Public Watchdog)	WDTIMER1	PO_INT_PEND
92	TINT4	DMTIMER4	POINTR_PEND
93	TINT5	DMTIMER5	POINTR_PEND
94	TINT6	DMTIMER6	POINTR_PEND
95	TINT7	DMTIMER7	POINTR_PEND
96	GPIOINT0A	GPIO 0	POINTRPEND1
97	GPIOINT0B	GPIO 0	POINTRPEND2
98	GPIOINT1A	GPIO 1	POINTRPEND1
99	GPIOINT1B	GPIO 1	POINTRPEND2
100	GPMCINT	GPMC	gpmc_sinterrupt
101	DDRERR0	EMIF	sys_err_intr_pend
102	Reserved		
103	Reserved		
104	Reserved		
105	Reserved		
106	Reserved		
107	Reserved		
108	Reserved		
109	Reserved		
110	Reserved		
111	Reserved		
112	TCERRINT0	TPTC0	tptc_errint_pend_po
113	TCERRINT1	TPTC1	tptc_errint_pend_po
114	TCERRINT2	TPTC2	tptc_errint_pend_po
115	ADC_TSC_PENINT	ADC_TSC	pen_intr_pend
116	Reserved		
117	Reserved		
118	Reserved		
119	Reserved		
120	SMRFLX_MPU subsystem	Smart Reflex 0	intrpend
121	SMRFLX_Core	Smart Reflex 1	intrpend
122	Reserved		
123	DMA_INTR_PIN0	External DMA/Interrupt Pin0 (xdma_event_intr0)	pi_x_dma_event_intr0
124	DMA_INTR_PIN1	External DMA/Interrupt Pin1 (xdma_event_intr1)	pi_x_dma_event_intr1
125	McSPI1INT	McSPI1	SINTERRUPTN

Appendix E :- INTC Registers and offsets

Offset	Acronym	Register Name	Section
0h	INTC_REVISION		Section 6.5.1.1
10h	INTC_SYSCONFIG		Section 6.5.1.2
14h	INTC_SYSSTATUS		Section 6.5.1.3
40h	INTC_SIR_IRQ		Section 6.5.1.4
44h	INTC_SIR_FIQ		Section 6.5.1.5
48h	INTC_CONTROL		Section 6.5.1.6
4Ch	INTC_PROTECTION		Section 6.5.1.7
50h	INTC_IDLE		Section 6.5.1.8
60h	INTC_IRQ_PRIORITY		Section 6.5.1.9
64h	INTC_FIQ_PRIORITY		Section 6.5.1.10
68h	INTC_THRESHOLD		Section 6.5.1.11
80h	INTC_ITR0		Section 6.5.1.12
84h	INTC_MIR0		Section 6.5.1.13
88h	INTC_MIR_CLEAR0		Section 6.5.1.14
8Ch	INTC_MIR_SET0		Section 6.5.1.15
90h	INTC_ISR_SET0		Section 6.5.1.16
94h	INTC_ISR_CLEAR0		Section 6.5.1.17
98h	INTC_PENDING_IRQ0		Section 6.5.1.18
9Ch	INTC_PENDING_FIQ0		Section 6.5.1.19
A0h	INTC_ITR1		Section 6.5.1.20
A4h	INTC_MIR1		Section 6.5.1.21
A8h	INTC_MIR_CLEAR1		Section 6.5.1.22
ACh	INTC_MIR_SET1		Section 6.5.1.23
B0h	INTC_ISR_SET1		Section 6.5.1.24
B4h	INTC_ISR_CLEAR1		Section 6.5.1.25
B8h	INTC_PENDING_IRQ1		Section 6.5.1.26
BCh	INTC_PENDING_FIQ1		Section 6.5.1.27
C0h	INTC_ITR2		Section 6.5.1.28
C4h	INTC_MIR2		Section 6.5.1.29
C8h	INTC_MIR_CLEAR2		Section 6.5.1.30
CCh	INTC_MIR_SET2		Section 6.5.1.31
D0h	INTC_ISR_SET2		Section 6.5.1.32
D4h	INTC_ISR_CLEAR2		Section 6.5.1.33
D8h	INTC_PENDING_IRQ2		Section 6.5.1.34
DCh	INTC_PENDING_FIQ2		Section 6.5.1.35
E0h	INTC_ITR3		Section 6.5.1.36
E4h	INTC_MIR3		Section 6.5.1.37
E8h	INTC_MIR_CLEAR3		Section 6.5.1.38
ECh	INTC_MIR_SET3		Section 6.5.1.39

Offset	Acronym	Register Name	Section
F0h	INTC_ISR_SET3		Section 6.5.1.40
F4h	INTC_ISR_CLEAR3		Section 6.5.1.41
F8h	INTC_PENDING_IRQ3		Section 6.5.1.42
FCh	INTC_PENDING_FIQ3		Section 6.5.1.43
100h to 2FCh	INTC_ILR_0 to INTC_ILR_127		Section 6.5.1.44

APPENDIX F –SITARA AM335X TIMER REGISTERS

Offset	Acronym	Register Name	Section
0h	TIDR	Identification Register	Section 20.1.5.1
10h	TIOCP_CFG	Timer OCP Configuration Register	Section 20.1.5.2
20h	IRQ_EOI	Timer IRQ End-of-Interrupt Register	Section 20.1.5.3
24h	IRQSTATUS_RAW	Timer Status Raw Register	Section 20.1.5.4
28h	IRQSTATUS	Timer Status Register	Section 20.1.5.5
2Ch	IRQENABLE_SET	Timer Interrupt Enable Set Register	Section 20.1.5.6
30h	IRQENABLE_CLR	Timer Interrupt Enable Clear Register	Section 20.1.5.7
34h	IRQWAKEEN	Timer IRQ Wakeup Enable Register	Section 20.1.5.8
38h	TCLR	Timer Control Register	Section 20.1.5.9
3Ch	TCRR	Timer Counter Register	Section 20.1.5.10
40h	TLDR	Timer Load Register	Section 20.1.5.11
44h	TTGR	Timer Trigger Register	Section 20.1.5.12
48h	TWPS	Timer Write Posting Bits Register	Section 20.1.5.13
4Ch	TMAR	Timer Match Register	Section 20.1.5.14
50h	TCAR1	Timer Capture Register	Section 20.1.5.15
54h	TSICR	Timer Synchronous Interface Control Register	Section 20.1.5.16
58h	TCAR2	Timer Capture Register	Section 20.1.5.17

APPENDIX G – SITARA AM335X UART REGISTERS

Offset	Acronym	Register Name	Section
0h	THR	Transmit Holding Register	Section 19.5.1.1
0h	RHR	Receiver Holding Register	Section 19.5.1.2
0h	DLL	Divisor Latches Low Register	Section 19.5.1.3
4h	IER_IRDA	Interrupt Enable Register (IrDA)	Section 19.5.1.4
4h	IER_CIR	Interrupt Enable Register (CIR)	Section 19.5.1.5
4h	IER_UART	Interrupt Enable Register (UART)	Section 19.5.1.6
4h	DLH	Divisor Latches High Register	Section 19.5.1.7
8h	EFR	Enhanced Feature Register	Section 19.5.1.8
8h	IIR_UART	Interrupt Identification Register (UART)	Section 19.5.1.9
8h	IIR_CIR	Interrupt Identification Register (CIR)	Section 19.5.1.10
8h	FCR	FIFO Control Register	Section 19.5.1.11
8h	IIR_IRDA	Interrupt Identification Register (IrDA)	Section 19.5.1.12
Ch	LCR	Line Control Register	Section 19.5.1.13
10h	MCR	Modem Control Register	Section 19.5.1.14
10h	XON1_ADDR1	XON1/ADDR1 Register	Section 19.5.1.15
14h	XON2_ADDR2	XON2/ADDR2 Register	Section 19.5.1.16
14h	LSR_CIR	Line Status Register (CIR)	Section 19.5.1.17
14h	LSR_IRDA	Line Status Register (IrDA)	Section 19.5.1.18
14h	LSR_UART	Line Status Register (UART)	Section 19.5.1.19
18h	TCR	Transmission Control Register	Section 19.5.1.20
18h	MSR	Modem Status Register	Section 19.5.1.21
18h	XOFF1	XOFF1 Register	Section 19.5.1.22
1Ch	SPR	Scratchpad Register	Section 19.5.1.23
1Ch	TLR	Trigger Level Register	Section 19.5.1.24
1Ch	XOFF2	XOFF2 Register	Section 19.5.1.25
20h	MDR1	Mode Definition Register 1	Section 19.5.1.26
24h	MDR2	Mode Definition Register 2	Section 19.5.1.27
28h	TXFLL	Transmit Frame Length Low Register	Section 19.5.1.28
28h	SFLSR	Status FIFO Line Status Register	Section 19.5.1.29
2Ch	RESUME	RESUME Register	Section 19.5.1.30
2Ch	TXFLH	Transmit Frame Length High Register	Section 19.5.1.31
30h	RXFLL	Received Frame Length Low Register	Section 19.5.1.32
30h	SFREGL	Status FIFO Register Low	Section 19.5.1.33
34h	SFREGH	Status FIFO Register High	Section 19.5.1.34
34h	RXFLH	Received Frame Length High Register	Section 19.5.1.35
38h	BLR	BOF Control Register	Section 19.5.1.36
38h	UASR	UART Autobauding Status Register	Section 19.5.1.37
3Ch	ACREG	Auxiliary Control Register	Section 19.5.1.38
40h	SCR	Supplementary Control Register	Section 19.5.1.39
44h	SSR	Supplementary Status Register	Section 19.5.1.40
48h	EBLR	BOF Length Register	Section 19.5.1.41

Table 19-29. UART Registers (continued)

Offset	Acronym	Register Name	Section
50h	MVR	Module Version Register	Section 19.5.1.42
54h	SYSC	System Configuration Register	Section 19.5.1.43
58h	SYSS	System Status Register	Section 19.5.1.44
5Ch	WER	Wake-Up Enable Register	Section 19.5.1.45
60h	CFPS	Carrier Frequency Prescaler Register	Section 19.5.1.46
64h	RXFIFO_LVL	Received FIFO Level Register	Section 19.5.1.47
68h	TXFIFO_LVL	Transmit FIFO Level Register	Section 19.5.1.48
6Ch	IER2	IER2 Register	Section 19.5.1.49
70h	ISR2	ISR2 Register	Section 19.5.1.50
74h	FREQ_SEL	FREQ_SEL Register	Section 19.5.1.51
80h	MDR3	Mode Definition Register 3	Section 19.5.1.52
84h	TX_DMA_THRESHOLD	TX DMA Threshold Register	Section 19.5.1.53