Supplementary Notes for ECE252

Version 1.55

Dr. Sol Rosenstark, B.E.E, M.E.E, Ph.D., P.E.

Professor Emeritus
of the
Electrical and Computer Engineering Department
New Jersey Institute of Technology
Newark, New Jersey

Contents

Topics Covered in the ECE μP Course

1. The register and chip architecture of the 68000 CPU.
2. Assembly and emulation of 68000 assembly language files.
    (a) Demonstration of the above. Dumping memory, tracing, breakpointing, register and flag display.
    (b) The V-flag from the supplementary notes.
3. The ASCII table. The landmarks are: 'A' = 41H, 'a' = 61H, '0' = 30H, SP = 20H, CR = 0DH, LF = 0AH.
4. Interpretation of Motorola HEX S-files. Consult the supplementary notes.
5. Hand assembly.
6. Addressing modes, sign extension must be considered. Examples:
    (a) Immediate: MOVE.W #$4586,D0.
    (b) Memory direct: MOVE.B $4586,D0.
    (c) Memory direct: LEA $8600,A3, sign extension depending on how it's assembled.
    (d) Address register indirect:
        i. MOVE.L (A1),D1
        ii. With predecrement MOVE.L −(A1),D1. Decrement depends on .B, .W or .L.
        iii. With posticrement MOVE.W (A1)+,D1. Increment depends on .B, .W or .L.
        iv. With displacement MOVE.B d_{16}(A2),D1. Note size of displacement.
        v. With displacement and indexing MOVE.W d_8(A2,A3.W),D2. Note size of displacement.
7. Conditional branches. Differences between arithmetic and logic branches.
8. Shifts, ROLs for bit manipulations.
9. ANDing and ORing with MASKs.
10. Distinctions between ASCII, BCD and packed BCD data.
11. The stack. PUSHes, POPs and subroutine calls and returns. Stack diagrams are very helpful and should be used to avoid confusion.
    (a) A registers get stored in higher memory, D registers in lower memory.

   (b) Higher number registers go into higher memory, lower number registers into lower memory.

12. The exception table. It contains longwords (vectors) which are the addresses of the service routines for the individual exceptions. The first two vectors are used by the CPU to initialize the SSP and the PC after reset.

13. Interrupts. There are two varieties.

    (a) Autovectoring is signaled when the $\overline{\text{VPA}}$ pin on the M68000 CPU is pulled low. (This is done with the $\overline{\text{AVEC}}$ on the M68EC000 CPU.) The $\overline{\text{IPL}}_2 \ldots \overline{\text{IPL}}_0$ CPU pins define the autovector number.

    (b) User vectoring is signaled when the $\overline{\text{DTACK}}$ pin on the CPU is pulled low. The CPU then reads the user vector off the lowest 8 pins of the data bus.

14. Base conversions and IEEE floating point format as covered in the supplementary notes.

15. Address decoding.

    (a) X's are variables and are used to denote address pins going directly to the memory or I/O chips.

    (b) Y's are don't cares and are used to denote address pins which are unaccounted for, and determine the number of address ranges for the decoding system.

16. The use $\overline{\text{LDS}}$ and $\overline{\text{UDS}}$ with 16-bit memory systems.

    (a) $\overline{\text{LDS}}$ is used to read the LSB to data pins $D_7 - D_0$ at odd addresses.

    (b) $\overline{\text{UDS}}$ is used to read the MSB to data pins $D_{15} - D_8$ at even addresses.

17. Timing diagrams and the significance of using $\overline{\text{DTACK}}$ for creating wait states.

18. Serial port transmission. Waveforms illustrated in the supplementary notes. Studied in connection with the UART on the single board computer schematic.

19. Parameter passing for reentrant subroutines and the LINK and UNLK instructions as covered in the supplementary notes.

20. Programs, programs, programs, but usually short ones.

The ASCII Chart

The code that is used for the transfer of information between computers and their peripheral devices is the American Standard Code for Information Exchange, commonly referred to as ASCII code. A tabulation of the code is shown in table 1.1. Below is a summary of the salient points of the ASCII chart.

The portion of the ASCII code in the range \$20 to \$7E represents readable characters corresponding to the Roman alphabet in both upper and lower case, the numbers from 0 to 9, special symbols such as (, }, \$, &, as well as punctuation marks. Thus a \$50 sent to a computer monitor or a printer will cause a letter *P* to appear on the peripheral. If the character *w* is typed on a keyboard, then the code \$77 is sent to the computer. Some landmarks in the ASCII table worth noting are the capital letters, which begin with \$41 representing the letter *A*, and ascending in proper order. The lower case letters begin with \$61 representing the letter *a*. The lower case and upper case letters differ from each other by \$20. It is also notworthy that \$20 is the code for a space.

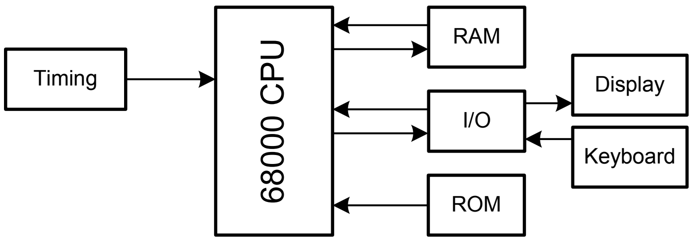
The ASCII code in the range 0 to \$1F represents control characters. As an example, they are the codes sent to a computer by the keyboard when a letter key is struck while the CTRL key is held down. To determine the HEX code for this operation, simply subtract \$40 from the ASCII value of the capital letter. Thus CTRL-I produces HT, a horizontal tab. A CTRL-H sends the code for a backspace, whereas CTRL-L produces a form feed which can be used to clear a screen or to have a printer eject a sheet of paper.

Typing CTRL-S produces a DC3, which is also known as an XOFF. It can be used in most programs to stop screen scrolling. The complementary action is obtained by typing CTRL-Q, which produces a DC1, also known as an XON. This can be used in most programs to restart screen scrolling. The ACK is often used in serial MODEM communication systems to acknowledge receipt of a block of valid data, whereas NAK can be used to as a negative acknowledge. And when the computer sent to the terminal a CTRL-G, which is a BEL, we used to get a ring but now get a beep. We should not omit that a carriage return (CR) is a \$0D, and is used to bring the screen cursor to the beginning of the current line. In addition a line feed (LF) is a \$0A, used to bring the screen cursor to the next line. Both are needed at the end of a line to make an orderly transition from one line to the next.

Table 1.1: The ASCII Code Chart.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	NUL	32	20	SP	64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	´	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	−	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	83	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	ˆ	126	7E	˜
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

The 68EC000 CPU Architecture



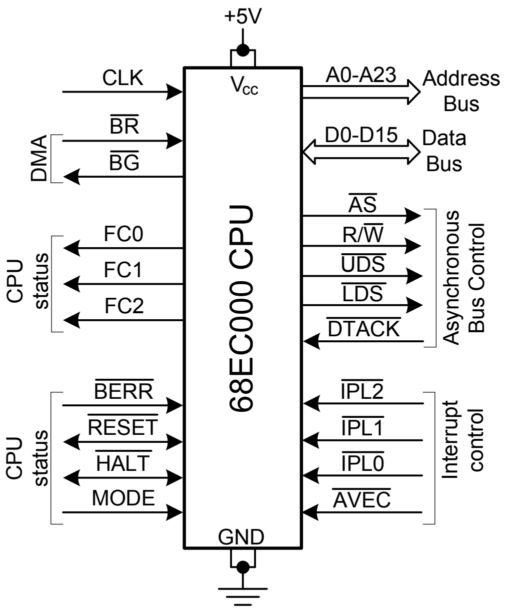


Figure 1.1: The block diagrams for the 68EC000 CPU.

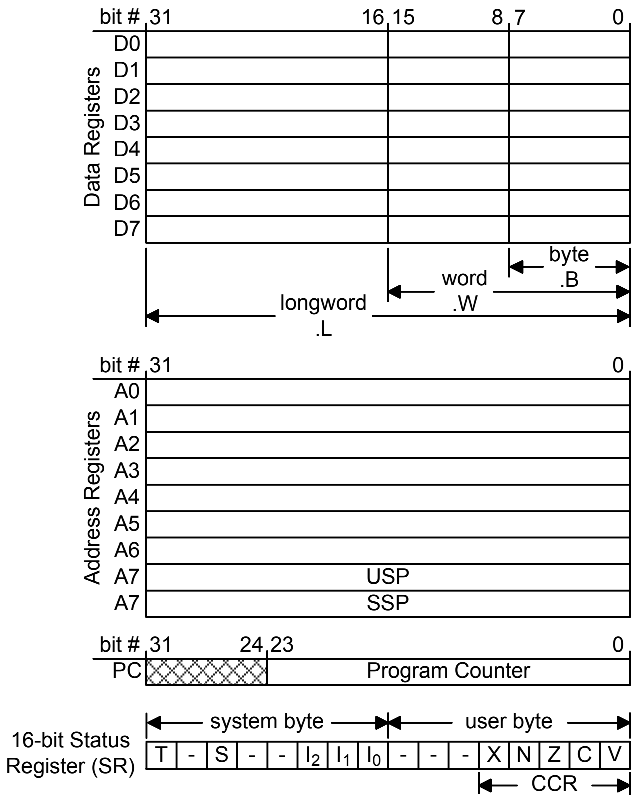
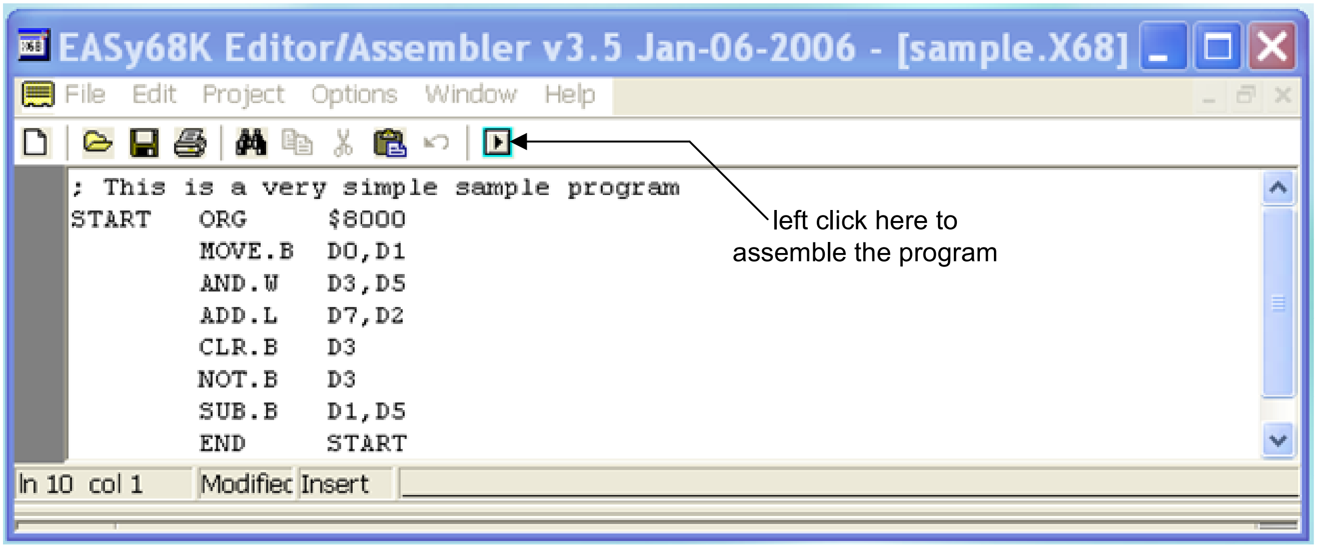


Figure 1.2: The register architectute of the 68EC000 CPU.

How to Use the Tools for this Course

The EASy68K Development System


EASy68K Editor/Assembler v3.5 Jan-06-2006 - [sample.X68]
File Edit Project Options Window Help
left click here to assemble the program
ln 10 col 1
Modified Insert

```
; This is a very simple sample program
START   ORG     $8000
        MOVE.B  D0,D1
        AND.W   D3,D5
        ADD.L   D7,D2
        CLR.B   D3
        NOT.B   D3
        SUB.B   D1,D5
        END     START
```

Figure 1.3: Creating the source file SAMPLE.X68

The assembler and emulator that is offered by Antonakos is a perfectly workable development system. The assembler ASM68K and the emulator EMU68K have to be executed in DOS. Most students have difficulty mastering the intricacies of the DOS operating system. Accordingly it is recommended that students download the freely distributed and open EASy68K Development System. This system is windows oriented and it should be easy to become familiar with it.

When you open EASy68K, you have the editor available to you. It is shown in figure 1.3. Some instructions have been typed in to create the source file. The fields are separated by TABs to keep the file columns aligned neatly. Once the file typing is complete it can be saved in a convenient folder, in this case using the name SAMPLE.X68. This step creates the source file.

Left clicking on the arrow icon shown in figure 1.3 causes the file to be assembled. This creates the two additional files SAMPLE.L68 and SAMPLE.S68. The latter is a HEX file and will be discussed later on. The file SAMPLE.L68 is the list file and shows the the memory addresses and OP-codes assigned to each instruction. The listing of this file appears below.

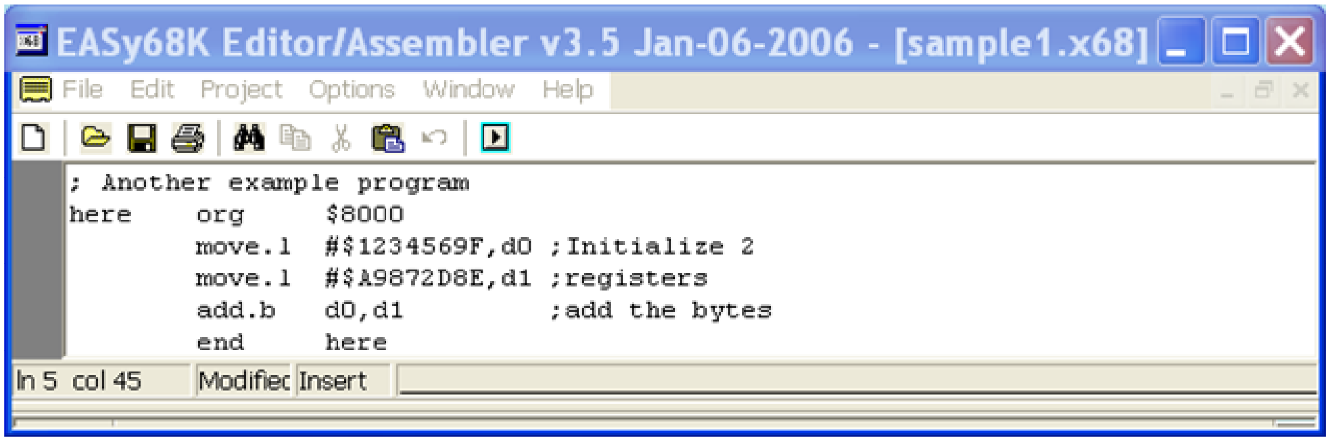
```
00008000 Starting Address
Assembler used: EASy68K Editor/Assembler v3.5 Jan-06-2006
Created On: 9/11/2006 2:03:20 PM

00000000           1  ; This is a very simple sample program
00008000           2  START   ORG     $8000
00008000  1200     3          MOVE.B  D0,D1
00008002  CA43     4          AND.W   D3,D5
00008004  D487     5          ADD.L   D7,D2
00008006  4203     6          CLR.B   D3
00008008  4603     7          NOT.B   D3
0000800A  9A01     8          SUB.B   D1,D5
0000800C           9          END     START

No errors detected
No warnings generated
```

The two last lines indicate that there were no problems associated with the source file. The first line in the source file has a leading semicolon (;) and is considered a comment by the assembler and does not produce any OP-code. The ORG in the second line is a directive telling the assembler that we will want the file to reside (originate) starting in memory location $8000. Accordingly all the instructions that follow are assigned addressed starting with $8000.

In this particular case, the instructions typed into this program all assemble into one word OP-codes. The instruction MOVE.B $D0, D1$ will be loaded into memory location $8000. It has the opcode $1200 which, when executed, will copy a byte of data from data register D0 into data register D1. Similarly for the other instructions.

An instructive example

In figure 1.4 we have typed the source file for another example. In the first two lines we initialize the registers D0 and D1 with longwords. The # signs preceding the hex numbers indicate that these are *immediate* data operations.


EASy68K Editor/Assembler v3.5 Jan-06-2006 - [sample1.x68]
File Edit Project Options Window Help
ln 5 col 45
Modified
Insert

```
; Another example program
here    org     $8000
        move.l  #$1234569F,d0 ;Initialize 2
        move.l  #$A9872D8E,d1 ;registers
        add.b   d0,d1         ;add the bytes
        end     here
```

Figure 1.4: The emulation window for the file SAMPLE1.X68

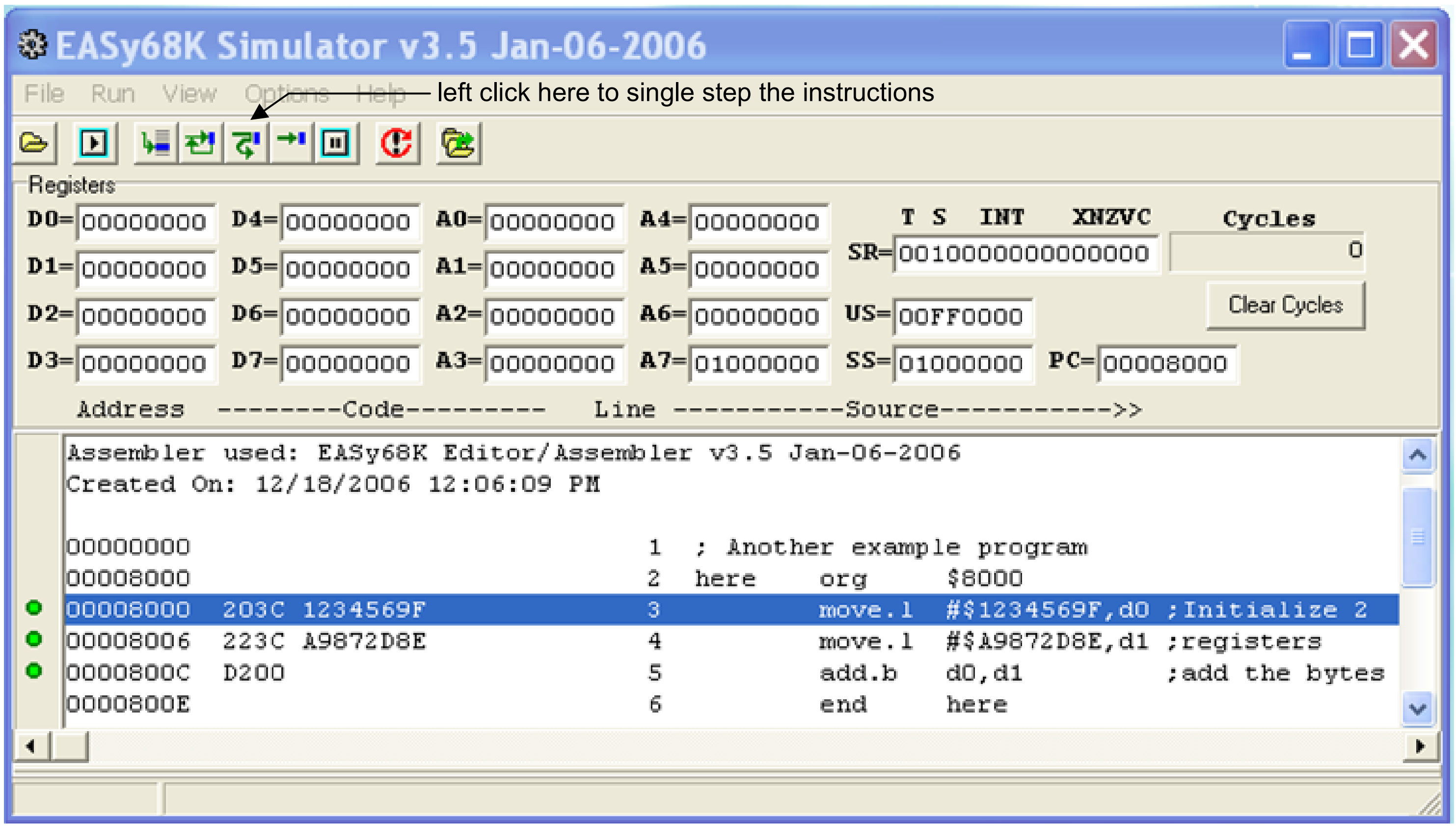
The data to be moved into the registers consists of the hex numbers immediately following the # signs. It is not necessary to clear the registers before moving data into them. The data will overwrite the existing data in the registers.

In the third line the CPU will add the byte in register D0 to the byte in register D1, and the result will be stored in the byte portion of the D1 register.

Once the file is assembled (consult figure 1.3 to see how that is done) the window shown in figure 1.5 appears on the screen. To *single step* through the program you can left click on the icon indicated or simply type F8. This procedure is called *tracing*.

After we trace the first instruction we see that indeed D0 = $1234569F. The Z-flag and N-flag are both zero indicating that the value in D0 is not 0, and that it is not negative. After the second instruction is traced we observe that indeed D1 = $A9872D8E. This time the N-flag equals one, indicating that the longword in D1 is negative. This is indeed the case since the longword has a binary 1 in the most significant bit position.

The third instruction should only affect the byte portion of D1. After tracng it we find that D1 contains the longword $A9872D2D, confirming that prediction. The sum of the HEX bytes $9F and $8E is $12D. Since this operation was restricted to one byte, the 1 in the result $12D could not possibly spill into the word portion of D1. It was stored in the C-flag which has changed to 1. In this instruction the X-flag always copies the C-flag so it is also equal to 1. Since two negative bytes $9F and $8E were added, we should have gotten a negative result. But the sum $2D in the byte position of the D1 register is positive, contrary to what was expected, so the oveflow V-flag is set to 1.


EASy68K Simulator v3.5 Jan-06-2006
File Run View Options Help
left click here to single step the instructions
Registers
Clear Cycles

```
D0=00000000  D4=00000000  A0=00000000  A4=00000000      T S  INT   XNZVC       Cycles
D1=00000000  D5=00000000  A1=00000000  A5=00000000  SR=0010000000000000           0
D2=00000000  D6=00000000  A2=00000000  A6=00000000  US=00FF0000
D3=00000000  D7=00000000  A3=00000000  A7=01000000  SS=01000000  PC=00008000

  Address   --------Code---------   Line -----------Source----------->>
Assembler used: EASy68K Editor/Assembler v3.5 Jan-06-2006
Created On: 12/18/2006 12:06:09 PM

00000000                             1  ; Another example program
00008000                             2  here    org     $8000
00008000  203C 1234569F              3          move.l  #$1234569F,d0 ;Initialize 2
00008006  223C A9872D8E              4          move.l  #$A9872D8E,d1 ;registers
0000800C  D200                       5          add.b   d0,d1         ;add the bytes
0000800E                             6          end     here
```

Figure 1.5: Emulation of the program SAMPLE1

A portion of the list file, sample1.L68, appears below. The first program line is a comment and produces no OP-code. The assembler was instructed in the second line that this program must be originated at at memory address $8000. Therefore it assigns the value $8000 to the label **here**. The symbol table at the end of the list file confirms this. The ORG is a directive to the assembler and produces no OP-code. The third line contains the opcode $203C, which instructs the CPU to move a longword of immediate data into the D0 register. The data, $1234569F, immediately follows the OP-code, hence the name immediate data.

```
00000000                            1  ; Another example program
00008000                            2  here    org     $8000
00008000  203C 1234569F             3          move.l  #$1234569F,d0 ;Initialize 2
00008006  223C A9872D8E             4          move.l  #$A9872D8E,d1 ;registers
0000800C  D200                      5          add.b   d0,d1         ;add the bytes
0000800E                            6          end     here

No errors detected
No warnings generated


SYMBOL TABLE INFORMATION
Symbol-name         Value
-------------------------
HERE                8000
```

It is recommended that you try a few sample programs using this system in order to become acquainted with it.

The emulator in this system has a number of functions built in. In order to make the transition from the ECE252 lecture course to the ECE395 laboratory seamless, I've taken the trouble to build many of these functions into the monitor program for the ECE395 single board computer (SBC). Table 1.2 shows all the functions of Sim68K that the single board computer of ECE395 will recognize, so students in ECE252 should confine themselves to these.

The Antonakos Development System

If you know DOS quite well then you can use this system without difficulty. Otherwise it is recommended that you go back to the preceding section.

Working in a DOS window, create a subdirectory \ASM68K on your hard drive. Copy to that subdirectory the files of ASM68K.EXE and EMU68K.EXE from the compact disk which comes with Antonakos's book. Consult the website `http://www.sunybroome.edu/~antonakos_j`, if you want to download later versions of these files. These will undoubtedly contain corrections and updates.

Table 1.2: The features available in the Sim68K simulator and on the SBC. Put the task number in D0.B and use TRAP #15 for all the tasks.

Task	The function and its requirements.
0	Display string at (A1), D1.W long (max 255) with CR, LF.
1	Display string at (A1), D1.W long (max 255) without CR, LF.
2	Read string from keyboard and store at (A1), length in D1.W (max 80).
4	Read a number from the keyboard into D1.L.
5	Read single character from the keyboard into D1.B.
6	Display the character in D1.B on the screen.
7	Set D1.B to 1 if keyboard input is pending, otherwise clear it. Use task 5 to read it.
9	Terminate the program gracefully.
12	Controls keyboard echo. D1.B = 0 to turn it off, D1.B ≠ 0 to turn it on. Echo is restored on 'Reset' or when a new file is loaded.
13	Display a null terminated string at (A1) with CR, LF.
14	Display a null terminated string at (A1) without CR, LF.

Make sure that you have a path to a **non-document type of editor** that produces pure ASCII files. WinWord is distinctly not to be used but NOTEPAD and EDIT will do the job. The latter is less desirable because it makes a mess out of TABs.

On page 13 of Antonakos's book he discusses the Hello program. Create the source file HELLO.ASM using a non-document type of editor. The source file should look something like this:

```
cr      equ     $d
lf      equ     $a
        org     $8000
enter   lea     hmsg,a3 ;point A3 to the message
        trap    #3      ;Send the string to the screen
        trap    #9      ;Do a graceful exit
hmsg    dc.b    'Hello!',cr,lf,0
        end     enter
```

Note the use of equates at the beginning of the program. In general we use equates to set the values of parameters that we might want to change sometime in the future.

Assemble the resultant HELLO.ASM program by typing ASM68K HELLO. If errors are signaled when the assembly is finished then read the file HELLO.LST to see where the errors are and correct them. Reassemble the program and have a look at the resultant HELLO.LST and HELLO.HEX files.

The file HELLO.LST lists the assembled instructions and it is largely self explanatory. A slightly edited version of the file is shown below:

```
<0D>                    cr      equ     $d
<0A>                    lf      equ     $a
008000                          org     $8000
008000  47F9 0000 800A  enter   lea     hmsg,a3 ;point A3 to the message
008006  4E43                    trap    #3      ;Send the string to the screen
008008  4E49                    trap    #9      ;Do a graceful exit
00800A  4865 6C6C 6F21  hmsg    dc.b    'Hello!',cr,lf,0
008010  0D0A 00
008013                          end     enter
8 lines processed.
0 warnings.
0 fatals.
```

To the left of each line is the loading address of the code and to the right of that you find the OPCODE (e.g. 47F9 in the first line) followed by from 1 to 4 words of data needed by the opcode.

Emulate the HELLO program by typing EMU68K HELLO.

Creating a Report in DOS

When operating in a DOS window, open the .LST file with NOTEPAD. Emulate the program in a DOS window. Right click on the top DOS bar, then on EDIT, then on MARK and then highlight the emulation. Hitting ENTER copies it to the DOS buffer. Now paste the emulation into the NOTEPAD containing the .LST file. This is a way of creating a .PRN file showing the .LST file and the emulation on one page.

An example of such a solution is shown below. This procedure should be carried out with every assigned programming problem. All programs submitted for grading should be in this format.

```
<0D>                    cr      equ     $d
<0A>                    lf      equ     $a
008000                          org     $8000
008000  47F9 0000 800A  enter   lea     hmsg,a3 ;point A3 to the message
008006  4E43                    trap    #3      ;Send the string to the screen
008008  4E49                    trap    #9      ;Do a graceful exit
00800A  4865 6C6C 6F21  hmsg    dc.b    'Hello!',cr,lf,0
008010  0D0A 00
008013                          end     enter
8 lines processed.
0 warnings.
0 fatals.
```

```
c:\ECE252\ASMfiles>emu68k hello
68000 Emulator V4.2, 1/10/05
53926 (0xD2A6) bytes allocated for emulator memory.
S008000068656C6C6FE3
S116800047F90000800A4E434E4948656C6C6F210D0A004B
S90380007C
Starting address: 8000
hello.hex loaded into emulator memory.
Enter '?' for help
-?
Author       A
Breakpoint   B number (1-4) address (0 to disable)
Dump         D [address] [lines]
Fill         F start-address stop-address pattern-byte
Enter        E [address] (use <cr> to skip over a byte
                    and any illegal key to exit)
Go           G [address] , [break address 1] , [break address 2]
               Ex: G 8200 (begin execution at $8200)
               Ex: G 8200, 8212 (begin execution at $8200 with
                   temporary breakpoint at $8212)
               Ex: G 8200, 8212, 8348  (begin execution at $8200 with
                   temporary breakpoints at $8212 and $8348)
               Ex: G , 8212, 8348 (begin execution at current PC with)
                   temporary breakpoints at $8212 and $8348)
Help         ?
Hex          H number1 number2
Load         L filename
Quit         Q
Register     R [register]
Trace        T [address] , [lines]
               Ex: T 400   (begin trace at $400)
               Ex: T 400,5 (trace 5 inst. starting at $400)
               Ex: T ,5    (trace next 5 inst. at current pc)
Unassemble   U [address] , [lines]
               Ex: Same syntax as Trace command
-g
Hello!
Program exit at address 00008008
-q
c:\ECE252\ASMfiles>
```

Deciphering of Motorola HEX S-files

Another file that is produced during the assembly of the source file is the HEX file. It contains address information, data and executable code for the assembled source file. It is the file that is used for the emulation of the program and it may also be downloaded for execution on the single board computer (SBC) that students use in the microprocessor laboratory. In the case of the HELLO program this file takes the form:

```
S008000068656C6C6FE3
S116200047F90000200A4E434E4948656C6C6F210D0A000B
S9032000DC
```

The EASy68K assembler creates files with the extension .S68. Antonakos's assembler creates files with the extension .HEX. Both types are Motorola HEX S-files.

A Motorola HEX S-file consists of pure ASCII characters. It means that it can be printed on a computer screen, as well as on a printer, without producing any smileys or aces of spades. Although the explanation below deals with the file HELLO.HEX shown above, it applies equally to all HEX S-files.

The `S0` in the first line indicates this is a header line. For convenience it is repeated below along with the explanation.

```
S008000068656C6C6FE3
08 means there are 8 bytes that follow.
0000 is an address, which is meaningless in the header.
68 65 6C 6C 6F are ASCII byte codes for the filename HELLO.
E3 is the check byte used to obtain the hex check sum as below:
hex check sum = 08 + 00 + 00 + 68 + 65 + 6C + 6C + 6F + E3 = FF
```

The hex check sum value of `$FF` attests to the integrity of this line of data.

It is noteworthy that the .S68 HEX-files created by the EASy68K assembler are always given the name `68KPROG   11CREATED BY EASY68K` irrespective of the name of the source file.

The HEX file can have any number of lines beginning with `S1`. These lines contain code, data, and a 16-bit load address. In this particular example we have:

```
S116200047F90000200A4E434E4948656C6C6F210D0A000B
This line contains code and data. The 16, following the S1, says
there are $16, or 22D, bytes that follow.
2000 means the bytes start loading at this 16-bit HEX address.
The rest, with the exception of 0B, are the bytes to load.
0B is the hex check byte as explained previously.
```

Some assemblers produce lines beginning with `S2`. These lines contain code, data, and a 24-bit load address. As an example we have:

```
S208AF800032004281D3
08 says there are $8 bytes in this line.
AF8000 means the bytes load at this 24-bit HEX address.
The rest, with the exception of D3, are the bytes to load.
D3 is the hex check byte as always.
```

The line beginning with `S9` indicates that this is a file-terminating footer-line with a 16-bit program entry-address. In our example it we have:

```
The line S9032000DC means this line contains a 16-bit
program entry-address, in this case 2000. The byte
count is 03 and DC is the hex check byte.
```

Some assemblers produce a footer line beginning with `S8`. This line is in all respects identical to the previously discussed footer line but it has a 24-bit program entry-address.

```
The line S804AF8000CC means this line contains a
24-bit program entry-address, in this case AF8000. The
byte count is 04 and CC is the hex check byte.
```

Some Possible Short Practice Programs

In the beginning of his book Antonakos's book he discusses the Hello program. This is as simple an example of assembly language programming as you'll ever find and it can be used to become familiar with the assembler and the emulator which are used with this course.

Create the source file HELLO.ASM using a non-document type of editor. Assemble the resultant HELLO.ASM program by typing ASM68K HELLO. If errors are signaled read the file HELLO.LST to see where the errors are and correct them. Reassemble the program and have a look at the resultant HELLO.LST and HELLO.HEX files.

Emulate the program using EMU68K HELLO. Become familiar with the HELP menu of the emulator by typing ?. Contrary to what most students think, there is more to this emulator than just the tracing option.

Make the change necessary to make sure that instead of getting the terrible looking final result:

```
Hello!Program exit at address 00002106
-
```

you get the rather more attractive result:

```
Hello!

Program exit at address 00002106
-
```

The above program can also be used with the EASy68K system. But the TRAP #3 used in Antonakos's sytem will have to be replaced with either task 13 or task 14.

Now that you have warmed up using the HELLO program you are ready for something more interesting. Write and emulate the following short programs.

1. ADD2.ASM

```
; Write and test a short program which will add a
; byte in D0 to a byte in D1. Use the emulator to
; test the program for various values. Look at the
; various flags and see how they are affected by
; your choice of values. Repeat for subtraction.
          org     $2000
  start   ---
          ---
          trap    #9
          end     start
```

2. TESTIT.ASM

```
; Write a short program which will test a byte
; in D0 to see if it is greater than $70 and
; less than $FA. If it is, then return with D0.B
; set to $FF. If not then D0.B is to be reset to
; zero. (NOTE: Be careful to distinguish between
; arithmetic and logic branches.) Use the emulator
; to test the program for enough values.
          org     $2000
  start   ---
          ---
          trap    #9
          end     start
```

3. FACTOR.ASM

```
; Calculate 5! The result is to appear in D1.L.
; Initialize a loop counter as well as the
; starting value in D1.L. Then do the looping.
          org     $2000
  factor  ---
          ---
  again   ; Some looping will be required.
          ---
          trap    #9
          end     factor
```

4. FINDX.ASM

```
; Put the displacement of 'x' from the
; start of MESG in D1.B. If none found
; then D1.B = $FF. This ASCII string is
;_not_ terminated with a NULL.
        org     $2000
findx
; Initialize the source, destination and loop counter
; registers. Get assembler to calculate the message
; length for the loop counter. Do the searching.
; The comparison can be made with #'x'.
succes  trap    #9
        org     $2020
mesg    dc.b    'This is an easy exercise.'
mesge
        end     findx
```

5. STRCNT.ASM

```
; Write a routine to count the bytes in the ASCII
; string at STRING. The NULL is not counted.
        org     $2000
; Initialize the byte counter and pointer
strcnt
;then do the counting by looping.
loop
        --
exit    trap    #9
        org     $2100
string  dc.b    'All men are created equal. '
        dc.b    'They are etc.',0
        end     strcnt
```

6. MOVENUL.ASM

```
; To move a message from memory location LOC1 to
; memory location LOC2. The NULL is also moved.
        org     $2000
; Initialize the pointers A0 and A1.
movenul --
        --
; Use a loop to do the moving and remember that the
; MOVE instruction sets the Z-flag properly.
```

```
loop    --
        --
quit    trap    #9
        org     $2100
loc1    dc.b    'This is a fantastic course!',0
        org     $2200
loc2    ds.b    50 ; 50 uninitialized bytes
        end     movenul
```

7. MOVEFF.ASM

```
; To move a message from memory location LOC1 to
; memory location LOC2, but first prefill 200 bytes
; of the target memory with FFs. The message below
; is terminated with a NULL.
        org     $2000
; First prefill the target space with FFs.
moveff
; Now do the move of the message.
        trap    #9
;
        org     $2100
loc1    dc.b    'This course is fantastic!',0
        org     $2200
loc2    ds.b    200
        end     moveff
```

8. MOVIT.ASM

```
; To move a message from memory location MESG to memory
; location DEST.
; This message is absolutely not terminated with a NULL.
        org     $2000
; Initialize the source, destination and loop counter
; registers. Get assembler to calculate the message
; length for the loop counter. Do the moving.
movit   --
        trap    #9
        org     $2020
mesg    dc.b    'This course is terrific!'
mesge
        org     $2040
dest    ds.b    50   ;50 uninitialized bytes
        end     movit
```

9. CONCAT.ASM

```
; Use of a subroutine is emphasized. Program to
; concatenate two parts of a string located
; at ST1 and ST2 and put the result at DEST.
; For the 1st part of the message initialize
; registers and call the subroutine MOVALL.
; Repeat the above for 2nd part of message.
        org     $2000
concat  ---
        trap    #9
; Subroutine for doing the moving
movall  --
        rts
        org     $2040
st1     dc.b    'George Washington '
st1e
        org     $2060
st2     dc.b    'slept here.'
st2e
        org     $2080
dest    ds.b    50
        end     concat
```

10. CRYPT.ASM

```
; The oldest and simplest encryption method is the
; Caesar cipher. The encryption method consists of
; changing the letters of the Roman alphabet by a
; fixed offset. We wish to encrypt the message stored
; at SOURCE into the message stored at MESSAGE. In our
; case we wish to change 'a' to 'h', 'b' to 'i', and
; so on. This should make the message impossible to
; read by only the very stupidest of enemies. The NULL
; is not crypted.
        org     $2000
aich    equ     'h'    ;Change as required
eh      equ     'a'
; Initialize registers
crypt   --
; Do the moving and adding
loop    --
        --
quit    trap    #9      ;exit gracefully
```

```
        org     $2100
source  dc.b    'Alert all centurions!',0
        org     $2140
message ds.b    $40
        end     crypt
```

11. ENCRYPT.ASM

```
; In a naive cryptography scheme we take the string at
; MESG and add 1 to the first letter, 2 to the second,
; and so on. Put the encrypted message at CRYPTED. The
; NULL is not to be encrypted.
        org     $2000
; Initialize registers
encrypt --
        --
; Do the crypting
loop    --
        --
exit    trap    #9
        org     $2100
mesg    dc.b    'Tell all the spies to lay low '
        dc.b    'else they run the risk etc.',0
        org     $2400
crypted ds.b    70
        end     encrypt
```

12. KBDIN.ASM

```
; At KBDBUF we have some arbitrary string of 5 ASCII
; digits, the most significant is first. Convert this
; string of 5 ASCII digits to a binary number at BINNUM.
; The ASCII digits are changed to BCD first. Then the
; number is calculated using nesting, like this:
; NUM = {[(num1*10 + num2)*10+num3]*10+num4}*10+num5
        org     $2000
kbdin   --
        --
loop    --
        --
exit    trap    #9
        org     $2100
kbdbuf  dc.b    '56394' ;What's the biggest we can have?
        org     $2110
```

```
binnum  ds.l    1
        end     kbdin
```

13. BINASC.ASM

```
; Subroutine for taking a 16 bit binary number at
; HEXNUM and converting to ASCII decimal digits.
; For example, 23AFH in HEX is converted using
; 23AFH / 10D = 391H rem 5. The 5 is the least
; significant digit of the original number. We OR
; it with 30H to get the ASCII 35H. Continue with
; 391 / 10D = 5BH rem 3, etc. The result for the above
; number should be '9135' in ASCII. All registers
; used are saved and restored.
        org     $2000
; Point A0 to charsto+6 and initialize other registers.
binasc  --
        --
; HINT: DIVU produces a 16-bit quotient in the LSW of
; the destination. The remainder is in the MSW of the
; destination. The destination must have the MSW = 0
; at the outset.
here    --
        --
        trap    #9
; Storage areas
        org     $2030
hexnum  dc.w    $23AF   ;binary number here
        org     $2040
charsto dc.b    '0'
        ds.b    5
        end     binasc
```

14. PACK.ASM

```
; Routine for packing 8 ASCII numbers
; found at CHARSTO. The resulting numbers
; are stored at PACKED. All registers used
; are saved and restored.
        org     $2000
pack
; Save registers, fix up pointers and loop counter.
loop
; Do everything on a byte basis, pick up ASCII byte,
; mask out 30H, and move into position.
        trap  #9
; Storage areas
        org   $8030
charsto dc.b  '95465783' ;ASCII input
        org   $8040
packed  ds.b  4     ;Packed BCD output
        end   pack
```

15. ADDBCD.ASM

```
; To add 6 packed BCD bytes (12 decimal digits)
; stored at STRING1 and STRING2 and put the
; result at STRING3.
        org   $2000
addbcd
; Make sure that pointers point to end of each string
; of numbers, because addition is done from right to
; left. Also clear the X-flag before you start.
; Fix up loop counter too.
loop
; Do the addition in a loop.
        --
        trap  #9
; Storage areas
        org   $8030
string1 dc.b  $45,$32,$78,$97,$56,$64
        org   $8040
string2 dc.b  $51,$56,$71,$89,$57,$28
        org   $8050
string3 ds.b  6
        end   addbcd
```

Branch Insructions Simplified

In figure **??** we have a line representing all HEX bytes.

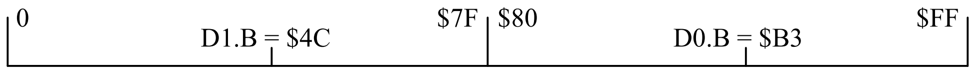


Figure 1.6: The number line in HEX bytes.

Suppose D1.B = $4C and D0.B = $B3. In the logic sense the numbers are treated in a continuous line. So D0.B $>$ D1.B, because $B3 is to the right of $4C. But that does not hold true in the arithmetic sense, because the numbers from 0 to $7F are positive, whereas those from $80 to $FF are negative. In the arithmetic sense D0.B $<$ D1.B.

The Bcc, which are the branch on condition instructions, exist in both arithmetic and logic versions. Let us assume that we just executed the instruction CMP.B D1,D0. The arithmetic branches rely on a signed comparison and the logic branches rely on an unsigned comparison. Assume that the CMP instruction is immediately followed by one of the branch instruction listed in the tables below.

Warning: The CMP.B D1,D0 instruction sets flags on the basis of the comparison between D0 and D1. If an instruction is inserted between the CMP and the branch instruction, then the inserted instruction will most likely modify the flags, and the expected branch will not take place as expected.

You can look at table 3.2 in Antonakos's book to see how the flags affect the different branch instructions. For example, for the BGT branch to take place, the condition $Z + (N \oplus V) = 0$ must be satisfied. That requires too much thinking. Instead of doing that, we consult tables 1.3 and 1.4.

Table 1.3: Arithmetic Branches.

Signed Comparison	Branch	Condition
D0 $>$ D1	BGT	Branch if Greater Than
D0 $\geq D1$	BGE	Branch if Greater or Equal
D0 $=$ D1	BEQ	Branch if Equal
D0 $\neq D1$	BNE	Branch if Not Equal
D0 $\leq D1$	BLE	Branch if Less or Equal
D0 $<$ D1	BLT	Branch if Less Than

Table 1.4: Logic Branches.

Unsigned Comparison	Branch	Condition
D0 $>$ D1	BHI	Branch if Higher
D0 $\geq D1$	BCC	Branch if Carry Clear
D0 $=$ D1	BEQ	Branch if Equal
D0 $\neq D1$	BNE	Branch if Not Equal
D0 $\leq D1$	BLS	Branch if Lower or Same
D0 $<$ D1	BCS	Branch if Carry Set

And finally there is the unconditional branch: BRA.

Shift and Roll Instruction Diagrams

The easiest way to understand the shift and roll instructions is to view them in the diagrams in figure 1.7. A close examination will reveal that the logic instructions do not try to preserve the sign bit. The arithmetic instructions, on the other hand, do preserve the sign bit where possible.

If, for example, you execute ASL #2,D0.B, then the byte in D0 will get multiplied by 4, because 2 binary zeros will be added on the right side of the number. Conversely, shifting to the right, by executing ASR #2,D0.B, will divide the number by 4, and the sign will be preserved. This is because the most significant bit gets recopied into the most significant position as shown in the diagram.

Suppose D0.B contains 10010010B = \$92. Because it has a binary 1 in the most significant position we know that the number is negative. To find its value we take the two's complement of this number, by complementing all its bits and adding 1, to determine that it is −\$6E. To verify the last result we add the two numbers to find that \$92 + \$6E = \$100. Executing the instruction ASR #2,D0.B, we get the result that D0.B = 11100100B = \$E4. The twos complement of this, found by the alternative method \$100 −\$E4, tells us that it is −\$1C.

To verify, we multiply −\$1C by 4 to get −\$70. This is not quite equal to the orginal −\$6E. We have an error of 2. This is because in right shifting we discarded the 10B on the right side of the number. The discarding of the 1 in the second bit position represents a round-off error of 2.

Examine the ROXL and the ROXR instructions. They roll the bits through the X-flag. These are not trivial instructions. They become useful if for some reason we decide to turn around the binary digits in a register, in a new cryptography scheme. Think of a program to do this.

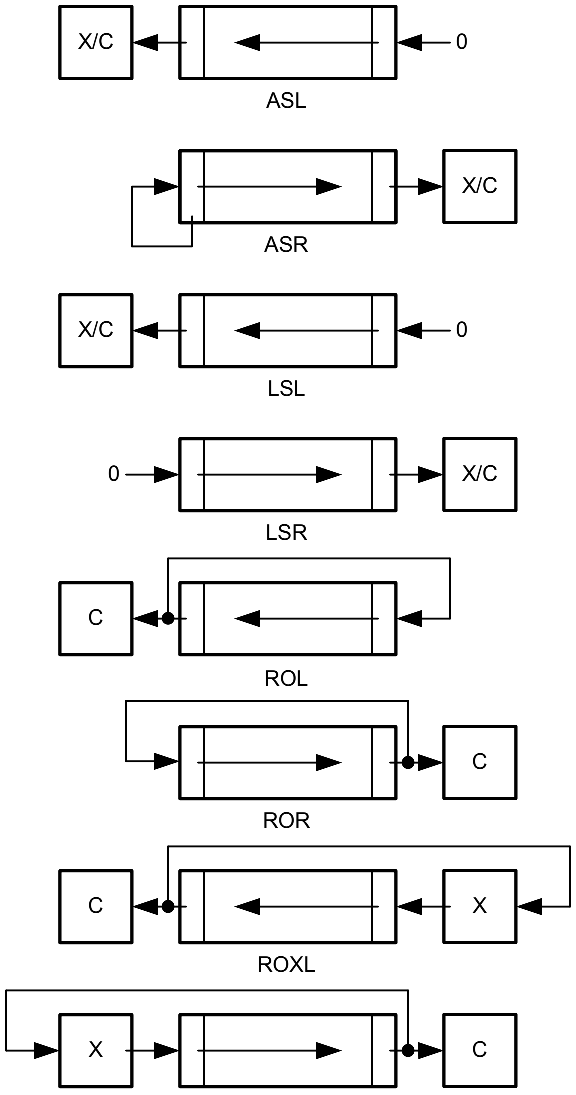


Figure 1.7: The number line in HEX bytes.

Binary Coded Decimal (BCD) Numbers

When typing numbers into a computer, they are entered in ASCII format. An examination of the ASCII table reveals that the digits from 0 to 9 go in as \$30 to \$39. So, if the digits 9, 5, 7, 3, 6 are typed. they could appear in the buffer:

```
ASCSTOR      DC.B       $39,$35,$37,$33,$36
```

The 3 has to be stripped away to convert the numbers to binary coded decimal (BCD) numbers for use in arithmetic operations. This is easily done by picking up the numbers into, e.g, the D1.B register and then using ANDI.B #\$F,D1. Then the numbers can be stored in the buffer:

```
BCDSTOR      DC.B       9,5,7,3,6
```

If it is desired to get the decimal value of the BCD digits into a form that can be used for calculation, we have to write a short routine to compute

$$\text{number} = \{[(9 \cdot 10 + 5) \cdot 10 + 7] \cdot 10 + 3\} \cdot 10 + 6$$

This could appear in the D1.L as \$175F8, and can be used for arithemetic purposes.

Question: What is the biggest decimal number that can be stored in the D1.L register?

If, for some reason, there is a shortage of memory space, BCD digits can be packed into *packed* BCD. Observe that the decimal digits 0 to 9 require only 4 bits (or a nibble) for their representation. From the BCDSTOR table, we can pick up the 6 in D0.B, then pick up the 3 in D1.B. We left shift D1.B by 4, and then OR.B D1,D0. We can then store the packed BCD numbers in the buffer:

```
PAKSTOR      DC.B       $09,$57,$36
```

In this manner we can reduce the storage requirements by a factor of 2.

Addition of Packed BCD Numbers

Adding packed BCD numbers, with a CPU that's capable of adding only HEX numbers, is not a trivial matter. Suppose we want to add the numbers \$08 and \$04. After hex addition the result is \$0C. This is not a packed decimal number. The rule for correcting it is to add 6 if the result of the one digit addition is greater than 9, or if a carry was generated. Here the result is greater than 9 and no carry was generated. Adding 6 to \$C produces the correct BCD result \$12.

As another example, consider adding \$98 to \$59. $8 + 9 = \$11$. The 1 in the left position is a *half-carry* (HC-flag) from the addition in the least significant half of the byte. Since a carry occurred, we must add 6 to the \$11, to get the result \$17. So the result in the least significant nibble is 7. Adding the HC-flag, which is 1, to the 9 and the 5, we get \$F. Since this exceeds 9 we add 6 to it to get \$15.The final result is \$57 and the carry-flag as well as the X-flag are both 1. Indeed 98D + 59D = 157D.

The above manipulations are very tedious. Fortunately for us, Motorola implemented the packed BCD addition instruction ABCD. It lets us add packed BCD bytes between two D registers, or between two memory locations. It adds the two bytes and the X-flag. The resultant carry is copied into the X-flag. It's important to null the X-flag at the start of the addition.

If you have downloaded the file "programs.zip Practice software" from my website: `http://web.njit.edu/~rosensta/`, then you should have the program file ADDBCD.ASM at your disposal. To understand how the ABCD instruction works, you are encouraged to assemble and emulate this short program.

Exception Processing

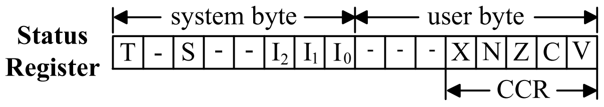
Modes of Operation of the 68EC000 CPU

The MC68000 CPUs have two modes of operation. They consist of the supervisor and the user modes. The S bit of the status register (see figure 1.8) determines the mode of operation. If it is 1 then the CPU is in supervisor mode, if it is 0 then it is in user mode. We saw previously in figure 1.2 that the CPU has two stack pointers (registers A7), the SSP for the supervisor mode, and the USP for the user mode.

Certain instructions can be executed in (the privileged) supervisor mode and not in (the less privileged) user mode. They are STOP, RESET, RTE, and any instructions that deal with the full SR. As an example, in user mode you may perform ANDI.B #$10,CCR. By contrast, the instruction ANDI.W #$10,SR can only be executed in supervisor mode. If an attempt is made to execute the priveleged instructions in user mode, then a PRIVILEGE VIOLATION exception will take place.

Exception processing begins when the CPU detects a condition which does not permit it to continue normal excution of a program. There are software generated exceptions and there are hardware generated ones. As an example of a software generated exception consider the program:

```
      -----
      CLR.W   D5
      DIVU    D5,D3 ;divides the longword in D3 by the word in D5
NEXT  ----
```



Exception vector assignments

	Address		
Vector #	**Dec**	**Hex**	**Assignment**
0	0	000	Reset: Initial SSP
	4	004	Reset: Initial PC
2	8	008	Bus error
3	12	00C	Address error
4	16	010	Illegal instruction
5	20	014	Zero divide
6	24	018	CHK instruction
7	28	01C	TRAPV instruction
8	32	020	Privilege violation
9	36	024	Trace
10	40	028	Line 1010 emulator
11	44	02C	Line 1111 emulator
12	48	030	(Unassigned, reserved)
13	52	034	(Unassigned, reserved)
14	56	038	Format error
15	60	03C	Unitialized interrupt vector
16–23	64	040	Unassigned, reserved
	95	05F	–
24	96	060	Spurious interrupt
25	100	064	Level 1 interrupt autovector
26	104	068	Level 2 interrupt autovector
27	108	06C	Level 3 interrupt autovector
28	112	070	Level 4 interrupt autovector
29	116	074	Level 5 interrupt autovector
30	120	078	Level 6 interrupt autovector
31	124	07C	Level 7 interrupt autovector
32–47	128	080	TRAP instruction vectors
	191	0BF	–
48–63	192	0C0	Unassigned, reserved
	255	0FF	–
64–255	256	100	User interrupt vectors
	1023	3FF	–

Figure 1.8: The 68000 exception vector table.

Before performing the division, the CPU checks the divisor in register D5.W. If it is zero then ZERO DIVIDE (vector 5) exception processing proceeds as follows:

1. The Program Counter (PC) is pointing to the instruction at NEXT when the CPU is executing the DIVU instruction. The CPU PUSHes this longword PC address onto the supervisor stack.

2. The CPU then PUSHes the word length Status Register (SR) onto the supervisor stack.

3. The CPU shifts to supervisor mode. It then multiplies the exception vector by 4. So $5 \times 4 = 20_{10} = \$14$.

4. The CPU reads a longword at address $14. This longword is the address of the service routine for the ZERO DIVIDE exception. The CPU loads this address into the PC, hence execution proceeds at this new address.

5. If the exception is fatal, as is the case here, then the function of the service routine is to print a message saying what problem was encountered, and to terminate execution of the program. No attempt is made to go back to the offending program so the CPU is left in supervisor mode. In the case of the Single Board Computer (SBC), used in the μP lab, it waits for another command (e.g. register dump) to be typed.

6. If the exception is not fatal then the exception servicing routine ends in an RTE instruction. Look up what this does and you'll see why. **Question:** Why is this different from an ordinary RTS ?

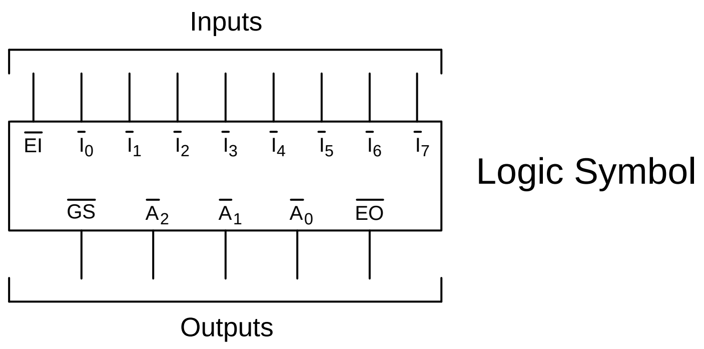
There is a complete table of exceptions shown in figure 1.8. As another example let us look at the Line 1111 exception. There are no Motorola instructions that begin with $F in the most significant nibble. This was reserved for those who want to create their own instructions. Any instruction that begins with $F will cause a Line 1111 exception.

The CPU will run through steps 1 and 2 outlined above. Since this is vector 11, then in step 3 the CPU multiplies 11 by 4 to obtain 44 decimal or 02C HEX. It looks at this address to find the address of the Line 1111 exception processing routine and then branches there to execute it. (The routine might simply multiply the word in D0.W by 16.) When this routine terminates in an RTE instruction, the CPU picks up where it left off.

The 74148, 8 to 3 line Priority Encoder

It is worthwhile to become familiar with the 74148, 8 to 3 line priority encoder chip, before proceeding with the discussion. of interrupts. Figure 1.9 is presented to help the students better understand the functioning of this chip.

74148 8-Line to 3-Line Priority Encoder


Inputs
$\overline{EI}$ $\overline{I}_0$ $\overline{I}_1$ $\overline{I}_2$ $\overline{I}_3$ $\overline{I}_4$ $\overline{I}_5$ $\overline{I}_6$ $\overline{I}_7$
$\overline{GS}$ $\overline{A}_2$ $\overline{A}_1$ $\overline{A}_0$ $\overline{EO}$
Outputs
Logic Symbol

	Inputs									Outputs				
Line	$\overline{EI}$	$\overline{I}_0$	$\overline{I}_1$	$\overline{I}_2$	$\overline{I}_3$	$\overline{I}_4$	$\overline{I}_5$	$\overline{I}_6$	$\overline{I}_7$	$\overline{GS}$	$\overline{A}_2$	$\overline{A}_1$	$\overline{A}_0$	$\overline{EO}$
1	H	X	X	X	X	X	X	X	X	H	H	H	H	H
2	L	H	H	H	H	H	H	H	H	H	H	H	H	L
3	L	X	X	X	X	X	X	X	L	L	L	L	L	H
4	L	X	X	X	X	X	X	L	H	L	L	L	H	H
5	L	X	X	X	X	X	L	H	H	L	L	H	L	H
6	L	X	X	X	X	L	H	H	H	L	L	H	H	H
7	L	X	X	X	L	H	H	H	H	L	H	L	L	H
8	L	X	X	L	H	H	H	H	H	L	H	L	H	H
9	L	X	L	H	H	H	H	H	H	L	H	H	L	H
10	L	L	H	H	H	H	H	H	H	L	H	H	H	H

Truth Table

Figure 1.9: Logic symbol and truth table for the 74148, 8 to 3 priority encoder.

Autovectored Interrupts

Autovectoring is very simple to implement, but is limited to only 7 levels of interrupts. The level 0 autovector indicates that no interrupt is requested. Interrupts can be disabled (masked out) by setting the bits (I_2,I_1,I_0) in the system byte of the status register (see figure 1.8). If, for example, the mask is set at level 4 then interrupts of level 4 or lower are masked out.

In the diagram of figure 1.10 it is assumed that all input pins of the 74148 have pullup resistors. Assume that the $\overline{I}_4$ pin is pulled down. The 74148 chip outputs are then as shown. The CPU recognizes that a level 4 interrupt is requested, it finishes executing the current instruction, and checks the interrupt mask. If the mask is set at level 3 or lower then it proceeds to service the interrupt.

The CPU acknowledges that it is servicing the interrupt by outputting (1,1,1) on the (FC2,FC1,FC0) pins. This, in conjunction with the $\overline{\text{AS}}$ pin can be used to generate an $\overline{\text{INTACK}}$ signal. It also lets us know that it is servicing a level 4 interrupt by putting out a (1,0,0) on the (A3,A2,A1) address pins. It then checks to see if the $\overline{\text{AVEC}}$ pin or the $\overline{\text{DTACK}}$ pin is pulled down. If the $\overline{\text{AVEC}}$ pin is pulled down then the CPU understands that it is dealing with autovectoring. If, on the other hand, the $\overline{\text{DTACK}}$ pin is pulled down then the CPU proceeds with user vectoring.

NOTE: The $\overline{\text{AVEC}}$ pin on the 68EC000 CPU corresponds to the $\overline{\text{VPA}}$ pin on the now defunct 68000 CPU.

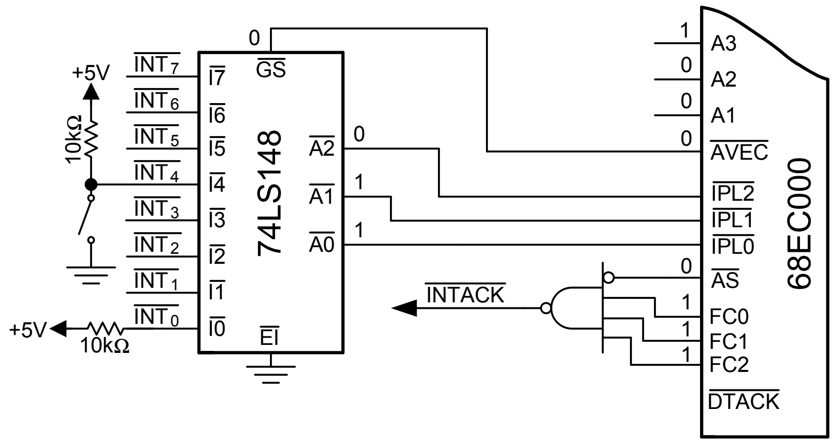


Figure 1.10: Example of autovectored interrupts.

User Vectored Interrupts

User vectoring is somewhat more complicated, but it makes it possible to implement 192 levels of interrupts. In this case the $\overline{\text{DTACK}}$ pin is pulled down and the CPU proceeds with user vectoring. This means that as a final step it reads the lowest 8 pins of the data bus to determine the user vector number, which can range from \$40 to \$FF.

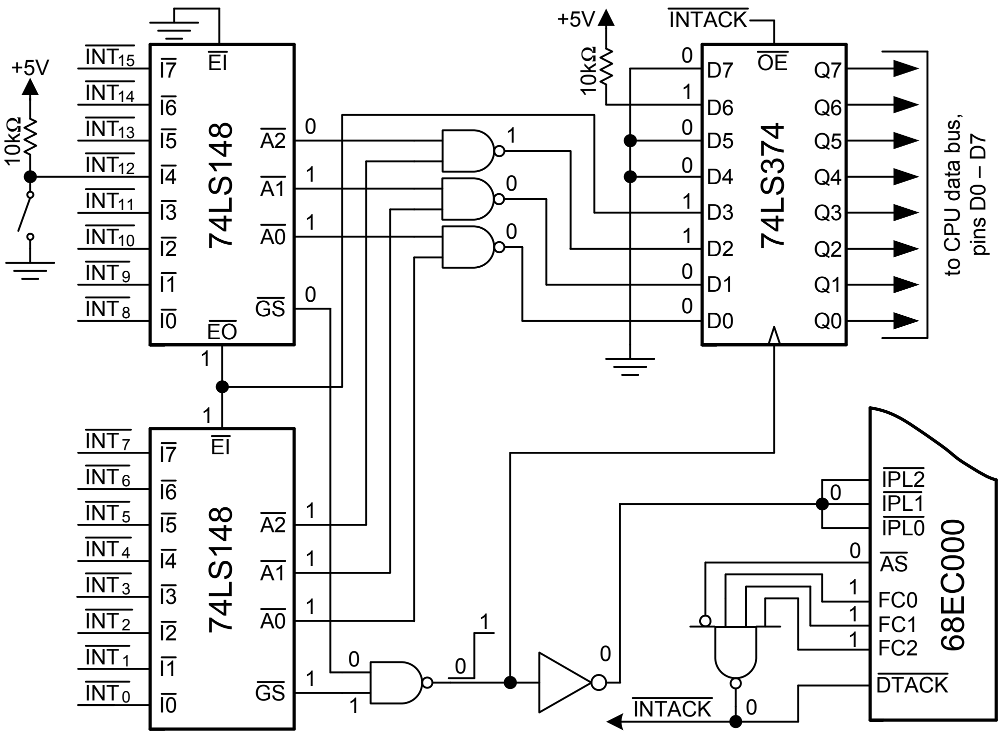


Figure 1.11: Example of user vectored interrupts.

Base Conversion and the IEEE Floating-Point Number Standard

The Positional Number Notation

The decimal number system uses the digits from 0 to 9 to represent numbers of any desired magnitude. The position of the digits from right to left determines their contribution to the overall number. It should come as no great surprise that the interpretation of the decimal number 982.34 is

$$982.34 = 9 \times 100 + 8 \times 10 + 2 \times 1 + 3 \times \frac{1}{10} + 4 \times \frac{1}{100} \tag{1.1}$$

Using powers of 10 we get the more useful interpretation

$$982.34 = 9 \times 10^2 + 8 \times 10^1 + 2 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2} \tag{1.2}$$

Converting from any base to decimal is quite an easy task. It is obvious from (1.2) that the base r number $(abcd.fg)_r$ has the decimal value

$$(abcd.fg)_r = (a \times r^3 + b \times r^2 + c \times r^1 + d \times r^0 + f \times r^{-1} + g \times r^{-2})_{10} \tag{1.3}$$

The Hexadecimal Number System

We know full well the binary number system is used in digital computers. We are also aware of the fact that the notation for binary numbers is awkward in that even small numbers are represented by long strings of zeros and ones. The hexadecimal number system has been adopted to shorten the representation of binary numbers and make their display more manageable. In the last equation we saw how to convert from any base to the decimal system. Now we have to learn to convert from the decimal number system to the hexadecimal number system. The methods are so general that these conversions can be used to convert from decimal to any other base.

Conversion of Decimal Integers to Hexadecimal

In spite of the fact that nowadays we have calculators to do the job, it is still worthwhile to learn how to convert numbers from one base to another. Because we are accustomed to doing arithmetic in the decimal number system, it will be observed that the conversion process in going from any base to decimal is not at all the same as going in the other direction.

If we want to see the individual digits comprising an integer decimal number, we need only divide that number successively by 10 and look at the remainders. For example, to see the digits comprising the integer decimal number 982, we do the following:

$$\begin{aligned} 982 \div 10 &= 98 \quad \text{rem } 2 \\ 98 \div 10 &= 9 \quad \text{rem } 8 \\ 9 \div 10 &= 0 \quad \text{rem } 9 \end{aligned} \tag{1.4}$$

The remainders on the right are the digits contained in the decimal number 982, with the most significant digit on the bottom and the least significant on top. The algorithm is finished when the quotient is zero, as on the last line above.

The above method is applicable to any base system. For illustration purposes, the decimal number 102973 will be converted to hexadecimal form by the same method.

$$\begin{aligned} 102973 \div 16 &= 6435 \quad \text{rem } 13 \\ 6435 \div 16 &= 402 \quad \text{rem } 3 \\ 402 \div 16 &= 25 \quad \text{rem } 2 \\ 25 \div 16 &= 1 \quad \text{rem } 9 \\ 1 \div 16 &= 0 \quad \text{rem } 1 \end{aligned} \tag{1.5}$$

The remainders on the right, read from bottom to top in hexadecimal form, are the desired result. We conclude that

$$102973_{10} = \$1923\text{D} \tag{1.6}$$

Exercises - Convert the decimal numbers below to the base indicated by using the procedure illustrated above.

1. Convert 548 to HEX and verify by converting back.
2. Convert 65933 to HEX and verify.
3. Convert 4702 to base 5 (quinary) and verify.
4. Convert 97255 to base 12 (duodecimal) and verify.

Conversion of Decimal Fractions to Hexadecimal

If we want to see the individual digits comprising a decimal fraction, we need only multiply that number successively by 10 and look at the digit before the decimal point. For example, to see the digits comprising the decimal fraction 0.598, we do the following:

$$\begin{aligned} 0.598 \times 10 &= 5 + 0.98 \\ 0.98 \times 10 &= 9 + 0.8 \\ 0.8 \times 10 &= 8 + 0 \end{aligned} \tag{1.7}$$

The integers on the right are the digits contained in the decimal fraction 0.598, with the most significant digit on top and the least significant on the bottom. The algorithm is finished when the fraction is zero, or the specified number of digits behind the decimal point have been obtained.

The above method is applicable to any base system. For illustration purposes, the decimal fraction 0.857 will be converted to hexadecimal form, with 4 digits behind the radix point by the same method.

$$\begin{aligned} 0.857 \times 16 &= 13 + 0.712 \\ 0.712 \times 16 &= 11 + 0.392 \\ 0.392 \times 16 &= 6 + 0.272 \\ 0.272 \times 16 &= 4 + 0.352 \end{aligned} \tag{1.8}$$

The integers on the right, read from top to bottom in hexadecimal form, are the desired result. We conclude that

$$0.857_{10} = \$0.\text{DB64} \tag{1.9}$$

Exercises - Convert the decimal numbers below to the base indicated, to 4 places behind the radix point, with proper roundoff, by using the procedure illustrated above. Proper roundoff means that you have to calculate a additional digit to see which way to round.

1. Convert 0.548 to HEX and verify by converting back.
2. Convert 0.659 to HEX and verify.
3. Convert 0.548 to base 5 (quinary) and verify.
4. Convert 0.548 to base 12 (duodecimal) and verify.

The IEEE Format of Floating-Point Numbers

In the late 1970s, the IEEE set up a committee to come up with a standard for the representation and of floating-point numbers. This would determine how they would be stored in a computer and make it possible to exchange this data between computers of different brands. This ultimately resulted in IEEE Standard 754, to which conform the coprocessors of most computer manufacturers. The system is designed to get the most out of the memory space which is used for the storage of the numbers.

There are three IEEE standards for floating-point numbers. First, there is a 32-bit single precision or *short-reals* format. Secondly, there is a 64-bit double precision or *long-reals* format. Finally there is an 80-bit extended precision format. The latter is used inside floating-point math coprocessors to reduce roundoff errors during calculations. The extended precision format will not be discussed further.

Both the short-reals and long-reals numbers come in *normalized*, *denormalized* and some other special versions. All of those will be discussed below.

Normalized Floating-Point Numbers in IEEE Format

It is easiest to explain the standard through a simple example. Consider the decimal number

$$x = 9.25_{10} = \$9.4 \tag{1.10}$$

First write the HEX number in the binary form

$$x = 1001.01 \tag{1.11}$$

Now move the radix point until there is only a single 1 preceding it, to obtain the binary form

$$x = 1.00101 \times 2^3 \tag{1.12}$$

The above number has the *mantissa* (1.00101). Since every finite number will be rewritten with a 1 preceding the radix point then we can safely discard this one bit. This leaves us with the *significand*

$$\text{significand} = 00101000000... \tag{1.13}$$

We also have the binary exponent

$$\exp = +03 \tag{1.14}$$

There two basic systems for storing the numbers in memory of interest to us are the less accurate short-reals form and the more accurate long-reals form. The two systems are shown in tables 1.5 and 1.6. Read the captions in the two figures carefully as they discuss restrictions on the *exrads*, which are the biased exponents.

Table 1.5: Short-reals 32-bit normalized number storage. Exrads of 0 or \$FF are not permitted for these *normalized* numbers.

bit	31	$30 \longleftrightarrow 23$	$22 \longleftrightarrow 0$
	sign	8 bit	Significand.
	$0 \rightarrow +$	exrad =	23 bits with an
	$1 \rightarrow -$	exp + \$7F	implied 24*th* bit

Table 1.6: Long-reals 64-bit normalized number storage. Exrads of 0 or \$7FF are not permitted for *normalized* normalized numbers.

bit	63	$62 \longleftrightarrow 52$	$51 \longleftrightarrow 0$
	sign	11 bit	Significand.
	$0 \rightarrow +$	exrad =	52 bits with an
	$1 \rightarrow -$	exp + \$3FF	implied 53*rd* bit

For the number in our example, the IEEE short real format is

$$\overbrace{0}^{+}\;\overbrace{1\,0\,0\,0\,0\,0\,1\,0}^{\$82\,=\,3+\$7F}\overbrace{0\,0\,1\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0}^{\text{23 bit significand}} \tag{1.15}$$

which represents the bytes

$$\overbrace{0\,1\,0\,0\,0\,0\,0\,1}^{\$41}\overbrace{0\,0\,0\,1\,0\,1\,0\,0}^{\$14}\overbrace{0\,0\,0\,0\,0\,0\,0\,0}^{\$00}\overbrace{0\,0\,0\,0\,0\,0\,0\,0}^{\$00} \tag{1.16}$$

In Intel based computers (which are little-endian) these bytes are stored in memory with the least significant byte in the lower memory address. In Motorola based computers (which are big-endian) they are stored with the most significant byte in lower address memory, exactly as they appear in the last equation.

The reason an offset is used for the exponents is to make the range of exponents go continuously from the smallest to the largest. This way numbers can be compared without a need to examine the mantissa.

We can determine the largest and the smallest numbers that this system can handle. We'll do it only for the short-reals and leave the problem for the long reals as an exercise.

Since normalized numbers may not have exrads of all zeros or all ones, the range for exrads is

$$1 \leq \text{exrad} \leq \$\text{FE} = 254_{10} \tag{1.17}$$

Since the bias is \$7F $= 127_{10}$, we readily determine the range of exponents to be

$$-126_{10} \leq \exp \leq +127_{10} \tag{1.18}$$

Table 1.7: Short-reals denormalized numbers.

bit	31	$30 \longleftrightarrow 23$	$22 \longleftrightarrow 0$
	sign	8 bit	Significand.
	$0 \rightarrow +$	exrad = 0	23 bits but with no
	$1 \rightarrow -$	hence exp = $-\$7F$	implied $24th$ bit

Table 1.8: Long-reals denormalized numbers.

bit	63	$62 \longleftrightarrow 52$	$51 \longleftrightarrow 0$
	sign	11 bit	Significand.
	$0 \rightarrow +$	exrad = 0	52 bits but with no
	$1 \rightarrow -$	hence exp = $-\$3FF$	implied $53rd$ bit

For short-reals the mantissa can be as small as $1.0000\ldots = 1$ or as large as $1.1111\ldots \approx 2$, we conclude that

$$1 \times 2^{-126} \leq \text{normalized numbers} \leq 2 \times 2^{127} \tag{1.19}$$

Expressing the above in decimal form we finally conclude that the range of numbers that normalized short-reals format can handle is, in decimal form

$$1.175 \times 10^{-38} \leq \text{normalized numbers} \leq 3.403 \times 10^{38} \tag{1.20}$$

Denormalized IEEE Floating-Point Numbers

For short reals as well as for long-reals it was desired to have another range of numbers between the smallest normalized number and zero. This gap was filled by the denormalized numbers. This number system is shown in tables 1.7 and 1.8.

To implement this range the exrad is always zero hence the exponent of 2 is $-\$7F$ for short-reals and $-\$3FF$ for long reals. The significand can have any value but zero, and it has no implied 1. So, for short-reals, the mantissa corresponds to the significand and ranges in value from 2^{-23} to $0.1111\ldots \approx 1$.

We conclude that

$$2^{-23} \times 2^{-127} \leq \text{denormalized numbers} \leq 1 \times 2^{-127} \tag{1.21}$$

Which expressed in decimal form gives us the range

$$7.006 \times 10^{-46} \leq \text{normalized numbers} \leq 5.877 \times 10^{-39} \tag{1.22}$$

IEEE Floating-Point Standard for Special Cases

In the IEEE scheme of floating point numbers, zero is represented by a significand which is zero along with an exrad which is zero. It can have either a positive or a negative sign.

Infinity is represented by a significand which is zero and an exrad which is maximum, namely $FF for short-reals and $7FF for long-reals. The infinity can be either positive or negative.

Finally there is the *Not-a-Number*, or NAN, which represents the result of an operation that has no mathematical significance, such as dividing infinity by infinity. These numbers have the same exrads as the infinities and the significand must have any non-zero value. The NAN can be either positive or negative.

Exercises

1. The bytes 3F 40 00 00 are stored in memory in a Motorola based computer. What decimal number do those bytes represent ?

2. How will 9.25_{10} appear in memory as a long real in a Motorola based computer ?

3. How will 2.5_{10} appear in memory as a long real in a Motorola based computer ?

4. In the short-reals format, how many different normalized mantissas are there ?

5. In the short-reals format, how many different normalized exponents are there ? 's'or all ones is not part of normalized numbers.

6. On the basis of the last 2 questions, how many different numbers can be represented in short-reals format ?

7. Determine the decimal range of numbers that the normalized long-reals format can handle. A result similar to (1.20) is what is desired.

8. Determine the decimal range of numbers that the denormalized long-reals format can handle. A result similar to (1.22) is what is desired.

9. The decimal precision of a number is defined by the quantity of permissible decimal digits in the mantissa of the number. Determine the decimal precision that is available in the short-reals as well as the long reals number representation.

References

1. G. W. Gorsline, *16 Bit Modern Microcomputers,* Prentice Hall, 1985, Page 200.

2. A. S. Tanenbaum, *Structured Computer Organization,* 4*th* Edition, Prentice Hall, 1999, Appendix B.

Subroutines and Parameter Handling

The orderly passing of parameters to subroutines is a fairly complicated subject. One of the best explanations of the most sensible manner for accomplishing this task is discussed in detail in *68000 Assembly Language,* by Alan Clements, PWS publishing Company, 1994. The material that follows is based largely on the examples and explanations found in that book.

Subroutine Calls and Returns Using a Stack

Consider the short program below which demonstrates how the stack is used to access subroutines.

```
main    ---
        bsr     sub1 ;Address N is pushed onto the stack
N       ---          ;and execution proceeds at sub1
;
sub1    ---
        bsr     sub2 ;Address M is pushed onto the stack
M       ---          ;and execution proceeds at sub2
        rts          ;Address N is popped from the stack
                     ;and put into PC
sub2    ---
        rts          ;Address M is popped from the stack
                     ;and put into PC
```

The above shows nested subroutines. The long-word return-addresses are automatically *pushed* onto the stack by the BSR opcode and the same long-word return-addresses are automatically *popped* from the stack by the RTS opcode. The 68000 microprocessor series use address register A7 as the default stack pointer for subroutine calls and returns.

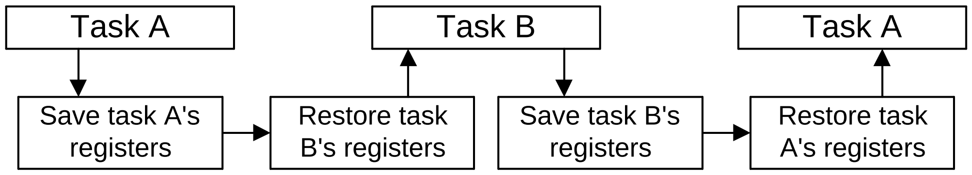


Figure 1.12: Task switching in a multitask environment.

Subroutine nesting is generally four or five deep. It can go much deeper when *recursion* is used. In that case care should be taken that the stack not run out of memory space.

For best programming readability, specific functions should be carried out by subroutines. The comments above the subroutine should contain an explanation of what the subroutine does and what registers are affected. The main program simply calls subroutine after subroutine to get the job done.

Task Switching in a Multitasking Environment

In a personal computer the EPROM BIOS controls various hardware functions of the computer. In the Windows environment we can have a number of tasks taking place seemingly simultaneously. In fact the CPU allocates a segment of time to each task and switches tasks according to some predetermined protocol. These tasks make use of the hardware interface routines which reside in the BIOS.

Let us suppose that the CPU is executing two tasks and that task A is executing a subroutine which task B needs to use as well. Before the switch takes place the CPU saves the registers of task A, including the status register but not the stack pointer. It then restores the registers of task B which it had saved earlier and transfers control to task B. Before control is transferred back to task A the registers of task B are saved and those of task A restored. This procedure is shown in figure 1.12.

Reentrancy in a Multitasking Environment

A reentrant process is one whose execution can be suspended before its completion, and which can then be used by another task without any harm being done to the suspended process. An example of such a process is the reading of a book.

Let us suppose that you are reading a book and a friend asks to borrow it. You mark the page where you left off and lend him the book. When the book is returned you continue to read where you left off. This process is reentrant provided your friend does not rip pages out of the book.

For a subroutine to be reentrant it is essential that all parameters, (meaning data), used by it be stored in such a way that they are not harmed when the task switching takes place. This holds true no matter where in the subroutine's execution the task switching takes place. The next task which uses the subroutine must not in any way alter the parameters used by the first task. If a subroutine stores parameters in absolute memory locations, then these variables will be corrupted by the next task that uses the subroutine. Take as an example the routine shown below.

```
main    bsr     getchar      ;Get a KBD char
        move.b  charsto,d1   ;put it in D1
        ---
getchar bsr     instat       ;Get status
        beq     getchar      ;0 -> no char, so loop
        move.b  dreg,charsto ;Get char
        rts
charsto ds.b    1            ;Store KBD character here
```

The above GETCHAR subroutine is distinctly not reentrant. If a second task were to borrow this subroutine before the first task executes the RTS code then it would store its own keyboard character at CHARSTO thus overwriting the character stored there by the first task. Compare the above code with that below.

```
main    bsr     getchar
        ---
getchar bsr     instat
        beq     getchar
        move.b  dreg,d1      ;Get KBD char into D1
        rts
```

In this case the KBD character was kept in a register. All registers are protected during task switching so the above routine is reentrant.

Subroutine Parameter Passing

Parameter Passing by Value

The simplest method of passing parameters to a subroutine is to put them into registers before the subroutine is called. The program below demonstrates the passing of one parameter to a subroutine.

```
main    move.b  #'G',d0
        bsr     char_out
        ---
```

The above method is called *passing parameters by value*. The number of parameters passed is limited by the number of registers available. This kind of parameter passing makes the subroutine reentrant.

Parameter Passing by Reference

If we have a field of data, then a subroutine can be given access to it by passing the starting address of the field as well as the size of the field in registers. In the example below, register A3 points to the start of an ASCII string and D0 holds the byte count of the string. The parameter in D0 is passed by value whereas the address pointer in A3 is *passed by reference*.

```
main    lea     mesg,a3          ;Point A3 to MESG
        move.w  #(mesge-mesg),d0 ;Put length into D0
        bsr     print_string
        ---
mesg    dc.b    'A very longwinded message ...'
mesge
```

The above two methods allow for a limited number of parameters to be passed since the number of CPU registers is limited. A more general way of performing this task is to pass parameters on the stack.

Parameter Passing by Using the Stack

The following short program demonstrates parameter passing using the stack.

```
main    pea     strbeg  ;Push start and end addresses
        pea     strend  ;of string to be searched
        pea     wordst  ;Push start and end addresses
        pea     wordend ;of the word to be found
        bsr     strsrch
        lea     16(a7),a7 ;tidy up the stack
        ;more code
strsrch lea     4(a7),a0    ;Point A0 past return address
        movem.l (a0)+,a3-a6 ;Pop parameters off the stack
        ;additonal code
        rts
strbeg  dc.b    'A very longwinded message ...'
strend
wordst  dc.b    'mess'
wordend
        end     main
```

The subroutine STRSRCH has to find if the word substring 'mess' exists in the sentence 'A very longwinded message ...' and report this fact back to the

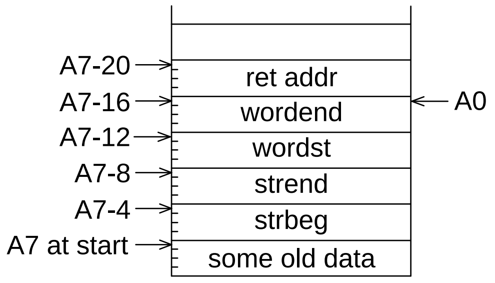


Figure 1.13: Memory map of the stack for the substring matching program.

routine MAIN. The parameters for the start and end of the sentence, as well as for the start and end of the substring, are pushed onto the stack as shown in figure 1.13.

On entering the subroutine the register A0 is initialized to point 4 bytes above the current stack pointer to the parameter WORDEND. The register A0 is used in the second line of code to pop the passed parameters into the work registers A3 to A6 where they will be used to accomplish the substring search.

Creating a Stack Frame

If reentrancy is to be assured, then new memory should be assigned for subroutine storage each time the subroutine is called. This is called *dynamic memory allocation*. There is no better way of doing it than by utilizing the stack.

Suppose that in the previous example we wanted to pass parameters to the subroutine as before and we also needed 18 bytes of memory for temporary storage of the subroutine's calculated values. Then an improved method of subroutine storage is the creation of a stack frame. This is demonstrated in the program below.

The register A6 is traditionally used as a frame pointer although any other address register can do the job. It is well possible that STRSRCH was called by another subroutine which itself was using A6 as a frame pointer. So the first order of business when the subroutine STRSRCH is entered is to push the old frame pointer register A6. The next step is to load the frame pointer A6 with the stack pointer value in A7. It is essential not to modify A6 after this step so that the old stack pointer A7 can be recovered later. At this point the stack pointer is decremented by 18 bytes to create the required subroutine work space. This is the last step in creating the stack frame and now the task of coding the remainder of the subroutine can be undertaken.

The new stack pointer, still the address register A7, is now used to save the registers to be used in the work of the subroutine. The frame pointer, register A6, is used to retrieve the passed parameters and to store any variables in the

18 byte workspace. All frame access should be done using displacements relative to address register A6 so that the value in A6 is never modified. This is done so that when the stack frame is collapsed at the end of the subroutine, the old stack pointer can be reloaded with its old value without problems.

```
main    pea     strbeg  ;Push start and end addresses
        pea     strend  ;of string to be searched
        pea     wordst  ;Push start and end addresses
        pea     wordend ;of the word to be found
        bsr     strsrch
        lea     16(a7),a7 ;tidy up the stack
; additional lines of code go here
strsrch move.l  a6,-(a7)    ;protect the old frame pointer
        movea.l a7,a6       ;create a new frame pointer
        lea     -18(a7),a7  ;leave 18 bytes free
        movem.l d0/a1-a4,-(a7) ;protect work registers
        movea.l 8(a6),a4    ;wordend in A4
        movea.l 12(a6),a3   ;wordst  in A3
        movea.l 16(a6),a2   ;strend  in A2
        movea.l 20(a6),a1   ;strbeg  in A1
; Do the necessary work but never modify the frame pointer A6.
; Before returning from the subroutine do the following:
        movem.l (a7)+,d0/a1-a4 ;restore work registers
        movea.l a6,a7       ;collapse the stack frame
        movea.l (a7)+,a6    ;restore old frame pointer
        rts
strbeg  dc.b    'A very longwinded message ...'
strend
wordst  dc.b    'mess'
wordend
        end     main
```

The memory map of the functioning of the frame can be easily followed by consulting figure 1.14. Observe that A0 points to the stack address where the old frame pointer is stored. To store a long word in register D0, at the first available location in the work space, we would use the instruction MOVE.L D0,-4(A6). To subsequently store a word in D1 we would have to use MOVE.W D1,-6(A6). The value of A6 must remain intact for the entire subroutine operation.

At the subroutine's end, the instruction MOVEA.L A6,A7 is used to restore the old value of A7 thus collapsing the stack frame. It now remains to pop the old value of A6 by using MOVEA.L (A7)+,A6 in order to restore the frame pointer of the routine that called the subroutine STRSRCH.

The above example shows a relatively laborious way of establishing and collapsing a frame. Fortunately the designers of the Motorola 68000 series chips

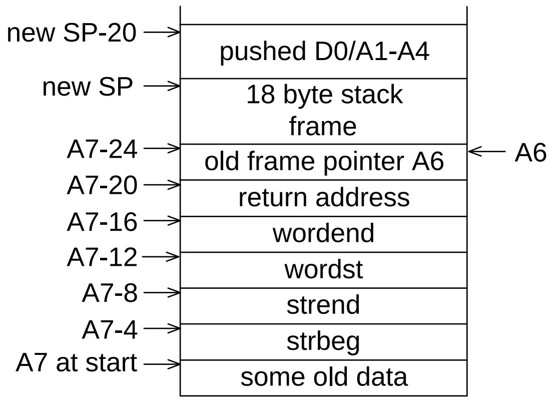


Figure 1.14: Memory map of the stack for the substring matching program.

considered the concept of frame usage of sufficient significance that they furnished two instructions to automate the entire process.

Use of LINK and UNLK Instructions for Stack Frames

To facilitate the frame creation process we use the LINK instruction. To facilitate its collapse we use the UNLK instruction. The last short program becomes even shorter when these two instructions are used.

```
main    pea     strbeg  ;Push start and end addresses
        pea     strend  ;of string to be searched
        pea     wordst  ;Push start and end addresses
        pea     wordend ;of the word to be found
        bsr     strsrch
        lea     16(a7),a7 ;tidy up the stack
; additional lines of code go here
strsrch link    a6,#-18
        movem.l d0/a1-a4,-(a7) ;protect work registers
        movea.l 8(a6),a4    ;wordend in A4
        movea.l 12(a6),a3   ;wordst  in A3
        movea.l 16(a6),a2   ;strend  in A2
        movea.l 20(a6),a1   ;strbeg  in A1
; Do the necessary work but never modify the frame pointer A6.
; Before returning from the subroutine do the following:
        movem.l (a7)+,d0/a1-a4 ;restore work registers
        unlk    a6          ;collapse the stack frame
        rts
;
```

```
strbeg  dc.b    'A very longwinded message ...'
strend
wordst  dc.b    'mess'
wordend
        end     main
```

Reading the above program we see that the LINK instruction

```
        link    a6,#-18
```

performs the work of the three instructions

```
        move.l  a6,-(a7)    ;protect the old frame pointer
        movea.l a7,a6       ;create a new frame pointer
        lea     -18(a7),a7  ;leave 18 bytes free
```

We also note that the UNLK instruction

```
        unlk    a6
```

performs the task of the two instructions

```
        movea.l a6,a7       ;collapse the stack frame
        movea.l (a7)+,a6    ;restore old frame pointer
```

The LINK and UNLK instructions would have undoubtedly been dispensed with in a RISC (reduced instruction set computer) because they can be constructed from other available code. This makes the job of compiler writers, who must use assembly language, very difficult. But the dearth of instructions makes the RISC computer perform most tasks faster. The designers of CISC (complete instruction set computer) machines want to make the CPU user friendly, so they create a wealth of instructions to make the assembly language look almost like a higher level language. Hence the luxury of finding complicated instructions for use in stack-frame management.

An Example of Stack Frame Management

We will now try to illustrate the parameter passing principles discussed thus far in the example that follows.

```
        ORG     $8000
main    move.w  #19,d0
        move.w  #11,d1
        move.w  d0,-(a7)    ;Push P on stack
        move.w  d1,-(a7)    ;Push Q on stack
        pea     R           ;Push address of R
        bsr     calc        ;Call subroutine
        lea     8(a7),a7    ;Tidy up
        trap    #9          ;Exit gracefully

calc    link    a0,#-14     ;Establish frame
        movem.l d6/a6,-(a7) ;Don't trash registers
        move.w  14(a0),d6   ;Get P
        mulu    d6,d6       ;Find P^2
        move.l  d6,-4(a0)   ;Save P^2 on stack frame
        move.w  12(a0),d6   ;Get Q
        mulu    d6,d6       ;Find Q^2
        move.l  d6,-8(a0)   ;Save Q^2 on stack frame
        add.l   -4(a0),d6   ;Q^2+P^2
        move.l  d6,-12(a0)  ;Store it
        move.l  -4(a0),d6   ;Get P^2
        sub.l   -8(a0),d6   ;Sub Q^2
        move.w  d6,-14(a0)  ;Store it
        move.l  -12(a0),d6  ;Get numerator
        divu    -14(a0),d6  ;Divide by denominator
        movea.l  8(a0),a6   ;Get R reference
        move.w  d6,(a6)+    ;Store ratio
        swap    d6
        move.w  d6,(a6)     ;Store remainder
        movem.l (a7)+,d6/a6 ;Restore registers
        unlk    a0          ;Collapse stack frame
        rts
        org     $8100
r       ds.w    2
        end
```

In this short program we wish to find the ratio $(P^2 + Q^2)/(P^2 - Q^2)$. The parameters P and Q are passed by value. The resultant quotient and remainder are to be stored at the address designated by R. The latter is passed by reference.

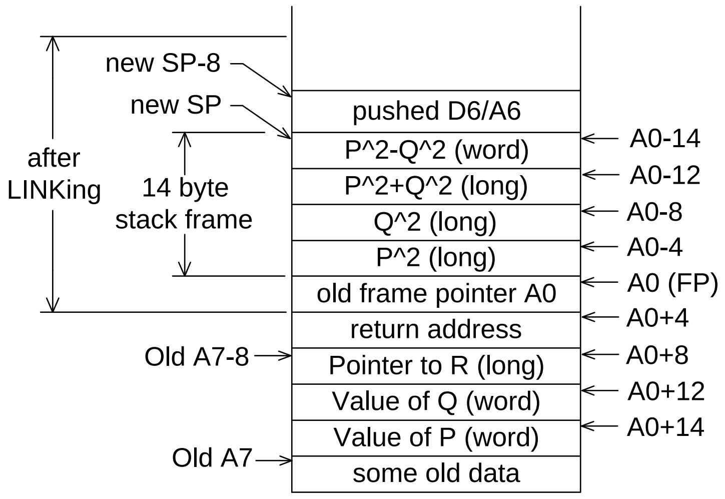


Figure 1.15: Memory map of the stack for ratio program.

To see how the parameters are handled we consult the memory map for this program shown in figure 1.15. The first step performed in the subroutine is the establishment of the frame through the LINK A0,#-14 instruction. Once that is done it is possible to push registers, used for performing the work in the subroutine, using the instruction MOVEM.L D6/A6,-(A7). In the next line of code MOVE.W 14(A0),D6 is used to get the parameter P. The above data is accessed using the addressing mode *address register indirect with displacement.* This method is used to access both the stack for the passed parameters and for storing of intermediate results. An example of the latter is the use of MOVE.L D6,-4(A0) to save P^2 on the stack frame.

In order to determine the displacements necessary for getting and saving parameters on the stack it is necessary to draw a diagram similar to the one shown in figure 1.15.

The RS-232 Serial Interface

A unidirectional 8-bit parallel interface requires at the very least 8 wires for data and one wire for ground. By contrast a unidirectional serial interface requires only 2 wires. One for the data and one for the ground. Such a cable is much cheaper to run, particularly where substantial distances are involved.

The current-loop interface existed in the early days of computers and handled data in serial form. It was designed specifically to interface with the now largely defunct (and very noisy) electromechanical teletypes. The RS-232 serial interface is a logical consequence of the current loop interface, and it has become fairly standard on modern personal computers. It should be mentioned that most RS-232 interfaces are bidirectional and, as a consequence, require at least 3 wires for a minimal connection.

An RS-232 communication interface requires a universal asynchronous receiver transmitter (UART) to transform outgoing data from parallel to serial form, and incoming data from serial to parallel form. Our single board computer uses the fast Intel 8251A (programmable communication interface) PCI as the UART. In the following material we will refer to figure 1.16a.

When the UART has no new data to transmit, it is in the IDLE state, and it outputs a continuous $+5\,\mathrm{V}$ on its T$\times$D pin. When a byte of data is loaded into the UART in parallel form, the UART immediately gets busy sending it out serially. It starts by sending a start bit at a zero volt level. This is then followed by the data in the byte, with the least significant bit (lsb) first. The data may consist of 7 or 8 bits, although the latter is the most accepted format these days. The most significant bit (msb) is sent last.

The data may then be followed by a parity bit, if this option is specified. If even parity is used, a bit is added so that the sum of the bits in the data and parity, modulo 2, will be 0. If using odd parity, a bit is added so that the sum of the bits in the data and parity, modulo 2, will be 1. In the example in figure 1.16a the number of 1's in the data is even, so a unity bit must be added to produce odd parity.

The parity bit is then followed by 1 or 2 stop bits, which are at a level of $+5\,\mathrm{V}$. The utilization of the 2 stop bits fell out of use when the 300 bit/second communication speed became largely obsolete. If there is no other data that needs to be transmitted, then the UART output remains at the idle level of

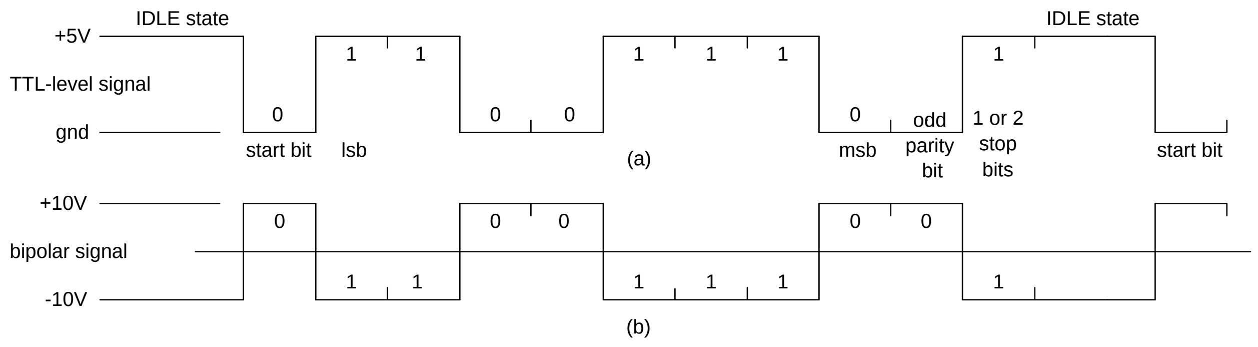


Figure 1.16: Serial form of the letter ‘s’ (ASCII value = 73H). TTL-level signal (a) and bipolar signal (b).

+5 V. Otherwise a new start bit is transmitted, followed by data.

The UART deals with data using TTL compatible voltages, that is 0 and +5 V. The RS-232 interface requires that the TTL signals be inverted and converted to a bipolar form with voltages ranging from ±3 V to ±15. Ordinarily the very popular 1488 and 1489 chips are used for this purpose. Their disadvantage is that they require the use of additional +12 V and −12 V power supplies. The MAX233A TTL-RS232 interface chip does away with that need in that it generates the +10 and −10 voltages internally. The signal that one actually sees on the wire is shown in figure 1.16b.

Most UARTS use a clock signal that is 16 times (16×) the actual bit rate of the RS-232 interface. The I8251A is no exception. So, for example, to get a bit rate of 38400 bits/sec requires the use of a UART clock of 614.4 kHz. A 16× clock is used so that the UART will have an opportunity to properly synchronize to the incoming signal. When the UART sees the falling edge of the start bit, it counts 8 clock sycles (one half of 16) then it verifies again that the voltage is still at 0. Then it samples the other incoming bits in the middle of their pulse, so as to minimize the possibility of sampling at a signal transition. If a framing, or a parity error, occur then the UART stores this in its status register. This can then be consulted when the data is read from the UART.

The Overflow Flag in Signed Number Arithmetic

The overflow flag has significance only for signed number arithmetic. In that case $V = 1$ indicates that the result of an arithmetic operation could not be stored correctly within the allocated space.

To facilitate the discussion, let us suppose a 4-bit CPU. The registers can hold 4-bit signed numbers. The significance of the signed binary (2's complement) representation is shown in the table below.

Table 1.9: The 4-bit signed numbers.

Binary	Hex	Binary	Hex
0000	0	1000	−8
0001	1	1001	−7
0010	2	1010	−6
0011	3	1011	−5
0100	4	1100	−4
0101	5	1101	−3
0110	6	1110	−2
0111	7	1111	−1

In the above signed binary representation the leftmost (high-order) bit is the *sign bit*. It is to be noted that in this system there are seven positive numbers and eight negative ones. Below are some examples of binary addition.

$$\begin{array}{llll}
\text{(a)} & \text{(b)} & \text{(c)} & \text{(d)} \\
0000 & 1111 & 0110 & 1000 \\
\quad 0011 = 3 & \quad 1101 = -3 & \quad 0011 = 3 & \quad 1110 = -2 \\
+\underline{\;0100 = 4} & +\underline{\;1011 = -5} & +\underline{\;0110 = 6} & +\underline{\;1001 = -7} \\
\quad 0111 = 7 & \quad 1000 = -8 & \quad 1001 = -7 & \quad 0111 = 7
\end{array}$$

There is no possibility of overflow occurring when two numbers of opposite sign are added since the magnitude of the result is always smaller than the largest magnitude of the two original numbers. If the signs of the two numbers added are the same then overflow can occur. It depends on the magnitudes of the numbers. The magnitudes of the numbers in examples (a) and (b) are small enough to produce no overflow. This is not the case in examples (c) and (d), where the results appearing in the 4-bit storage locations are incorrect.

The overflow bit can be determined from the carry-in to the sign bit, C_{is}, and the carry-out of the sign bit, C_{os}. The overflow flag V can be computed by exclusive ORing the above two bits, namely using $V = C_{is} \oplus C_{os}$. This is how most CPUs do it. Applying this rule to the above example we get $V = 0$ for cases (a) and (b) and $V = 1$ for cases (c) and (d).

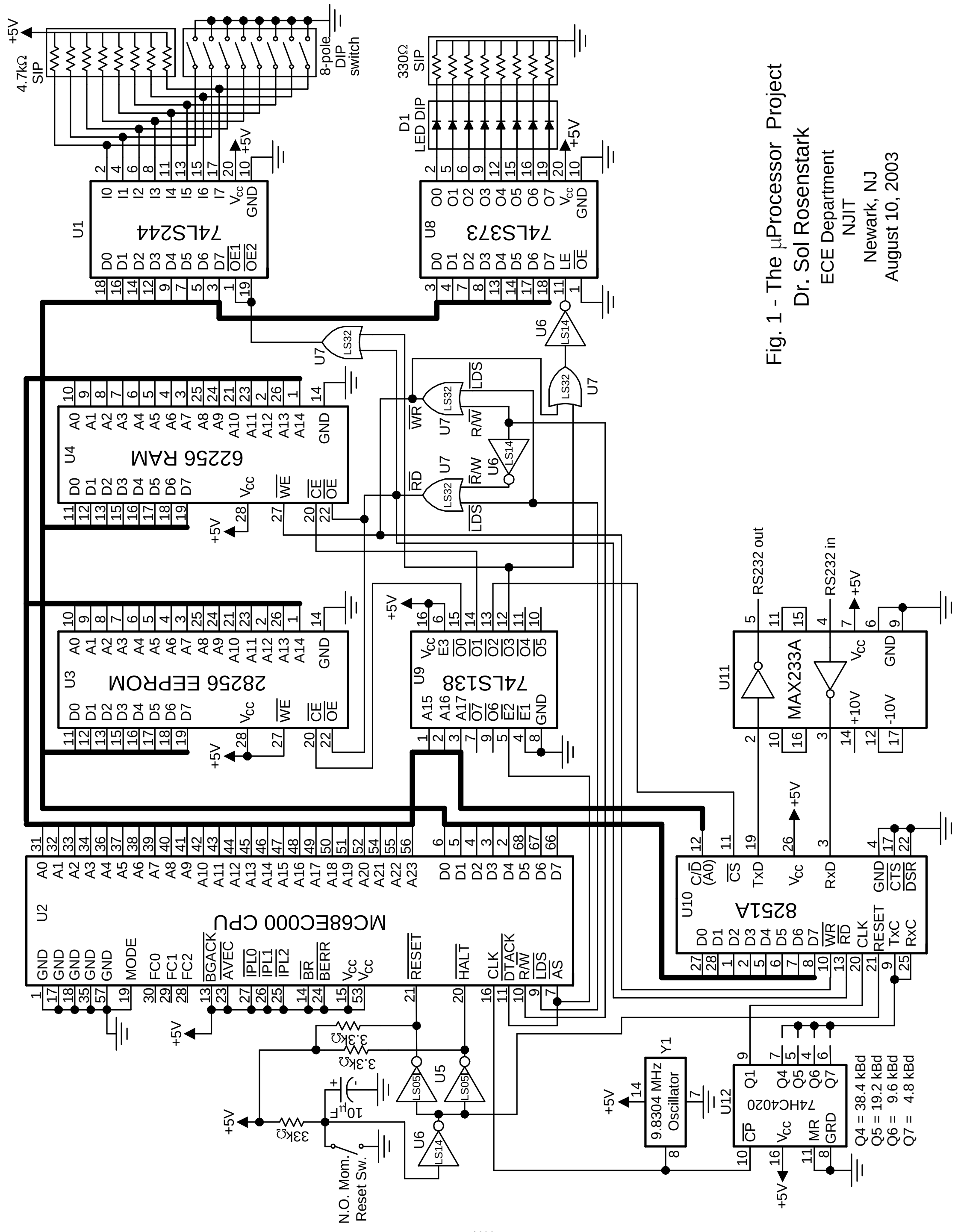


Fig. 1 - The μProcessor Project

Dr. Sol Rosenstark

ECE Department

NJIT

Newark, NJ

August 10, 2003

Memory Address Decoding and Timing Diagrams

There are two types of CPUs available. Those referred to as big-endian and those referred to as little-endian processors. Motorola CPUs are big-endian. They store the MSB in the even memory address, and the LSB in the next (higher, odd) memory address.

For example, in the EMU68K memory dump below, we see that the word stored at address \$2020 is \$5468. The MSB is \$54, which sits at the even address \$2020, and the LSB is \$68, and it sits at memory address \$2021. The longword stored there is \$54686973. In a big-endian CPU memory dump, what you see is what you get. The data can be interpreted effortlessly. Contrast this with the situaton for little-endian CPUs.

Intel processors are little-endian. For the memory dump below, the word at address \$2020 is \$6854. They store the LSB in the even memory address, and the MSB in the next (higher, odd) memory address. The long word stored at \$2020 is \$73696854. You have to go through some contortions to read memory dumps, but you soon get used to it.

There is no consensus as to which system is better, but Motorola memory dumps are easier to interpret.

```
-d 2020 2
002020  5468 6973 2069 7320 6120 6661 6E74 6173
002030  7469 6320 636F 7572 7365 2100 0000 0000
```

If you examine the pin-out of the MC68000 CPU, you will not find an A_0 pin. This function is served by the $\overline{\text{UDS}}$ and the $\overline{\text{LDS}}$ pins.

To reduce the pin count, most ROM memory chips have an 8-bit data bus. Since the MC68000 CPU has a 16-bit data bus, you need two ROM chips side by side to meet this requirement. In figure 1.17 we see just such an arrangement. The left chip supplies data to the CPU's $D_0 - D_7$ pins and the right chip delivers the data to the $D_8 - D_{15}$ pins. If bytes are read, then the MSB is read by making the $\overline{\text{UDS}}$ signal active, the LSB is read by making the $\overline{\text{LDS}}$ signal active. In

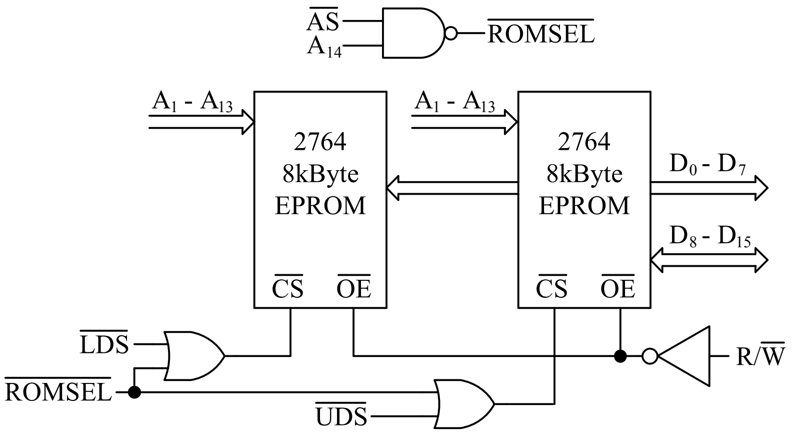


Figure 1.17: 8 kiloword EPROM memory.

reading a word, both $\overline{\mathrm{UDS}}$ and $\overline{\mathrm{LDS}}$ are active. To get longwords, two successive word reads are required.

Note that in figure 1.17, only pins $A_1 - A_{14}$ are utilized. Pins $A_{15} - A_{23}$ are unaccounted for. The memory shown is non-uniquely decoded, since the unaccounted for pins are don't-cares, so any 0 and 1 pattern of those can be used to access this memory.

To elaborate on this topic further, let us examine figure 1.18. Here is a case of complete or unique address decoding. All address lines $A_1 - A_{23}$ are accounted for, and the memory has only one address range. The lowest byte can be read at address \$480000, and the highest byte at address \$48FFFF. This contrasts with incomplete, or non-unique decoding found in figure 1.19.

In this case we have the two address lines, A_{19} and A_{18} unaccounted for in the diagram. They are don't-cares, and can be 00, 01, 10 or 11, without affecting the reading of the memory. As a consequence we see 4 ranges for this memory system. The byte in the lowest memory location can be read at \$600000, or at \$640000. It will be the same byte that is found at \$680000 and \$6C0000. This is because these chips are not uniquely decoded.

Finally let us look at the EPROM memory shown in figure 1.20. A 74LS139 2-4 decoder has been used to select 4 banks of EPROMS. The decoding diagram explains how the chips are addressed.

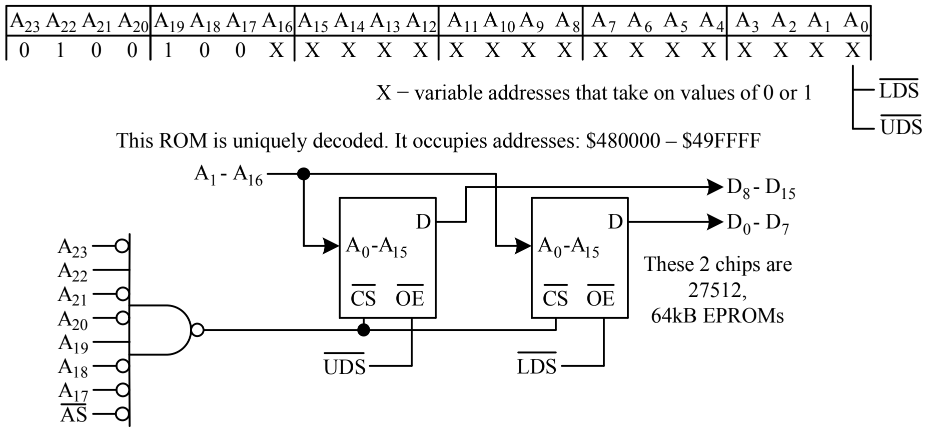

A23 A22 A21 A20 A19 A18 A17 A16 A15 A14 A13 A12 A11 A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0
0 1 0 0 1 0 0 X X X X X X X X X X X X X X X X X
X – variable addresses that take on values of 0 or 1
LDS
UDS
This ROM is uniquely decoded. It occupies addresses: $480000 – $49FFFF
A1 - A16
D8 - D15
D0 - D7
D
A0-A15
CS OE
These 2 chips are 27512, 64kB EPROMs
A23
A22
A21
A20
A19
A18
A17
AS
UDS
LDS

Figure 1.18: A uniquely decoded memory.

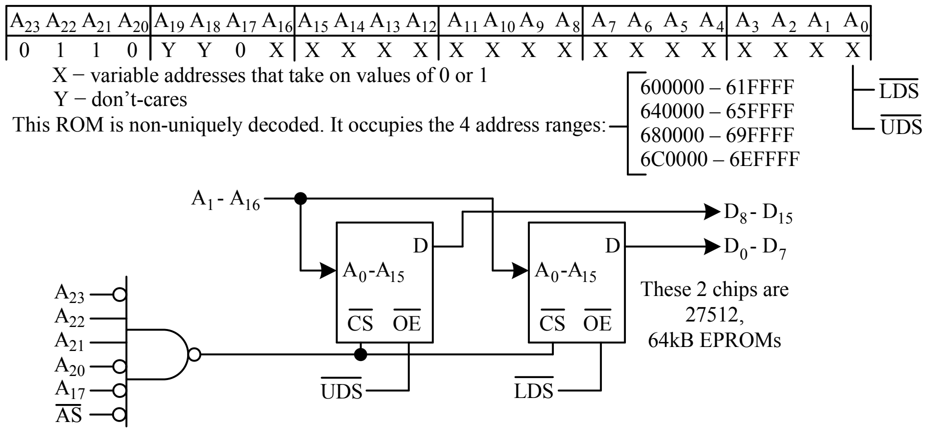
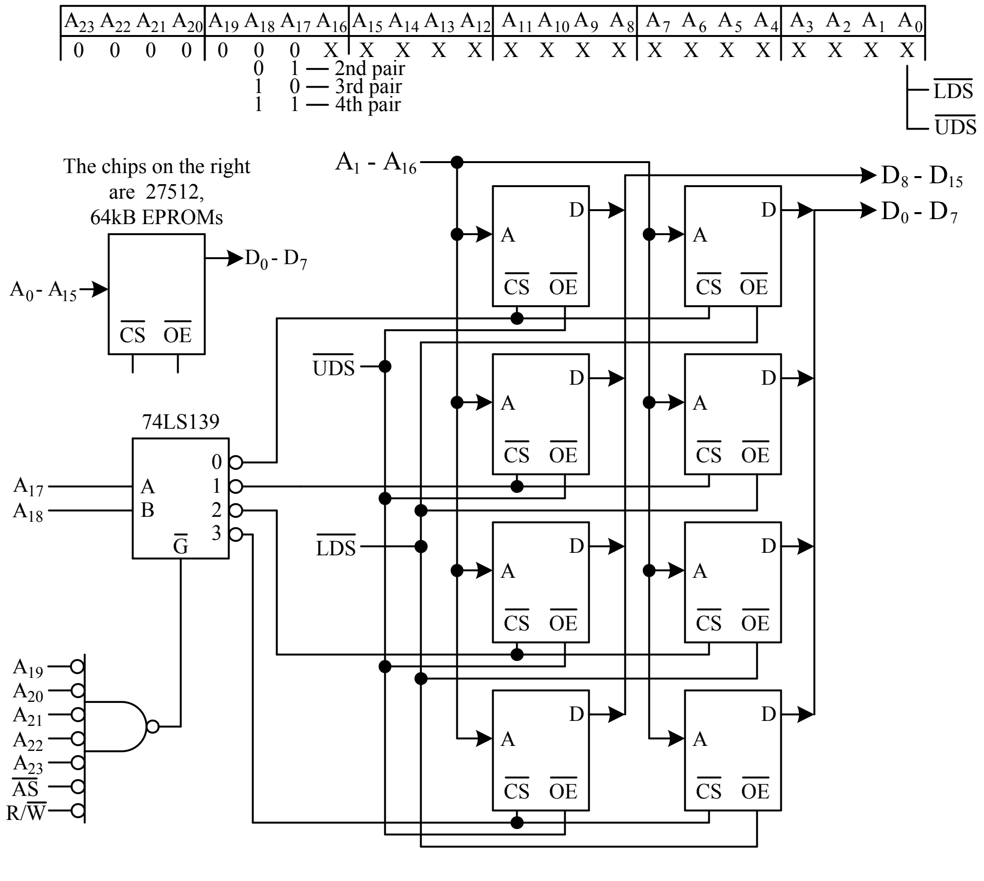
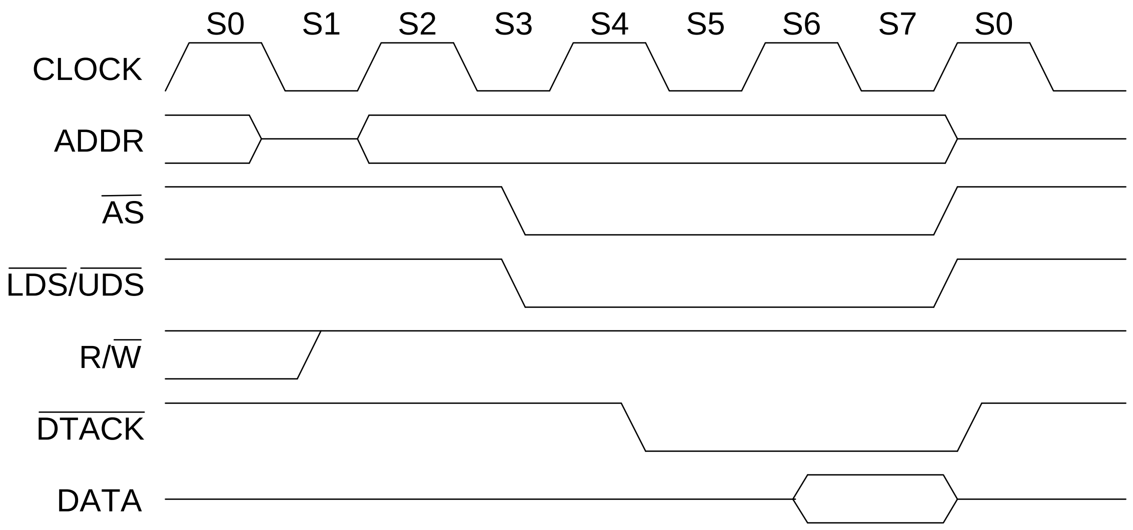

A23 A22 A21 A20 A19 A18 A17 A16 A15 A14 A13 A12 A11 A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0
0 1 1 0 Y Y 0 X X X X X X X X X X X X X X X X X
X – variable addresses that take on values of 0 or 1
Y – don't-cares
This ROM is non-uniquely decoded. It occupies the 4 address ranges:
600000 – 61FFFF
640000 – 65FFFF
680000 – 69FFFF
6C0000 – 6EFFFF
LDS
UDS
A1 - A16
D8 - D15
D0 - D7
D
A0-A15
CS OE
These 2 chips are 27512, 64kB EPROMs
A23
A22
A21
A20
A17
AS
UDS
LDS

Figure 1.19: A non-uniquely decoded memory.

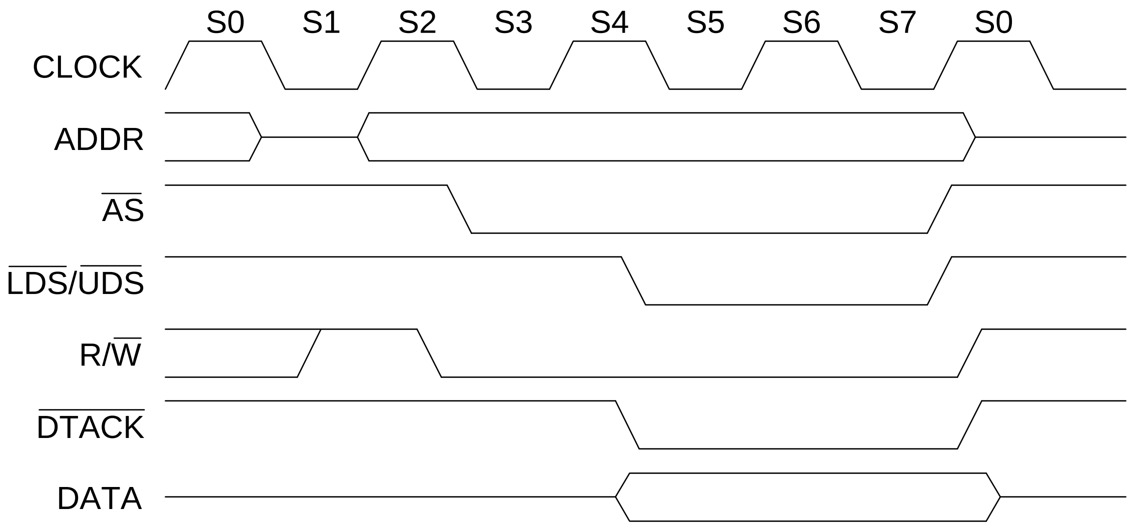


A_{23} A_{22} A_{21} A_{20} A_{19} A_{18} A_{17} A_{16} A_{15} A_{14} A_{13} A_{12} A_{11} A_{10} A_9 A_8 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0
0 0 0 0 0 0 0 X X X X X X X X X X X X X X X X X
0 1 — 2nd pair
1 0 — 3rd pair
1 1 — 4th pair
$\overline{LDS}$
$\overline{UDS}$
The chips on the right are 27512, 64kB EPROMs
A_0 - A_{15}
D_0 - D_7
$\overline{CS}$ $\overline{OE}$
74LS139
A_{17}
A_{18}
A B $\overline{G}$ 0 1 2 3
A_{19} A_{20} A_{21} A_{22} A_{23} $\overline{AS}$ R/$\overline{W}$
A_1 - A_{16}
$\overline{UDS}$
$\overline{LDS}$
D_8 - D_{15}
D_0 - D_7
A D $\overline{CS}$ $\overline{OE}$

Figure 1.20: 512kB of uniquely decoded memory.


S0
S1
S2
S3
S4
S5
S6
S7
S0
CLOCK
ADDR
$\overline{\text{AS}}$
$\overline{\text{LDS}}/\overline{\text{UDS}}$
$\text{R}/\overline{\text{W}}$
$\overline{\text{DTACK}}$
DATA

Synchronous Read Cycle


S0
S1
S2
S3
S4
S5
S6
S7
S0
CLOCK
ADDR
$\overline{\text{AS}}$
$\overline{\text{LDS}}/\overline{\text{UDS}}$
$\text{R}/\overline{\text{W}}$
$\overline{\text{DTACK}}$
DATA

Synchronous Write Cycle

Figure 1.21: The timing diagrams appearing in Antonakos's book were inaccurate in previous editions. These are in agreement with those presented by Motorola.