Aspects of Computer Architecture

T V Atkinson, Ph D
Senior Academic Specialist
Department of Chemistry
Michigan State University
East Lansing, MI 48824

Table of Contents

List of Tables

List of Tables

List of Figures

List of Figures

1. Introduction

1.1. Why should Chemists care about this mateial?

1. Typically, the chemistry professionals will encounter many different computer environments during their careers.
2. We (want to, need to, have to) use computers to do our work and have fun.

1.2. How can we characterize the people who use computers?

1.2.1. By the Type of Use

1. Application user
2. Operator
3. System manager
4. Application programmer
5. System programmer
6. Hardware developer
7. Software maintainer
8. Hardware maintainer
9. Hardware and software documentor
10. User support

1.2.2. By frequency of use of a particular program or facility

1. Occasional
2. Frequent

1.2.3. By level of expertise for a given program or facility

1. Novice
2. Versed
3. Expert/wizard/guru

2. Number Systems

An integer is represented in our system of writing by a string of symbols, digits, (d_i) from the set {0, 1, 2, … b-1} as shown below where "b" is the base of the representation.

$$\text{number} \Rightarrow d_n\, d_{n-1} \cdots d_2\, d_1\, d_0{}_{\mathrm{b}}$$

Numerically, the above notation represents the following sum.

$$\text{number} = \sum_{i=0}^{n} d_i \cdot b^i$$

As an example, the following are different representations of the same number.

$$\text{number} = 1010001110110010_2 = 121662_8 = 41906_{10} = A3B2_{16}$$

Fractional numbers can also be represented.

$$\text{number} \Rightarrow d_{-1}\, d_{-2} \cdots d_{-m+1}\, d_{-m}$$

$$\text{number} = \sum_{i=-1}^{-m} d_i b^i$$

The following are different representations of the same numbers.

$$\text{number} = 0.1_2 = 0.4_8 = 0.5_{10} = 0.8_{16}$$
$$\text{number} = 0.01_2 = 0.2_8 = 0.25_{10} = 0.4_{16}$$
$$\text{number} = 0.11_2 = 0.7_8 = 0.75_{10} = 0.\mathrm{C}_{16}$$

The general notation is as follows where "s" is the sign of the number and may be though of as being either +1 or -1.

$$\text{number} \Rightarrow s d_2\, d_1\, d_0 \,.\, d_{-1}\, d_{-2} \cdots d_{-m+1}\, d_{-m}$$

$$number = s \cdot \sum_{i=n}^{-m} d_i \cdot b^i$$

Scientific notation can be generalized as follows. In the following "s_m" is the sign of the mantissa, "s_e" is the sign of the exponent, and "B" is a symbol characteristic of the base.

$$\text{number} \Rightarrow s d_2\, d_1\, d_0 \,.\, d_{-1}\, d_{-2} \cdots d_{-m+1}\, d_{-m}\, B s_e\, e_n\, e_{n-1} \cdots e_2\, e_1 e_0$$

$$number = \left(s_m \cdot \sum_{i=-m}^{n} d_i \cdot b^i \right) \cdot \left(b^{s_e} \cdot \sum_{j=0}^{n} e_j \cdot b^j \right)$$

2.1. Range of Numbers

A given modulus, b, and a fixed number, n, of digits can express b^n numbers that range from 0 to $b^n - 1$. For example, in base 10, 5 digits can represent 100000 numbers from 0 to 99999. For base 2, 16 digits can represent 65536 (2^{16}) numbers from 0 to 65535.

2.2. Converting Between Different Moduli

Conversion of a number from one power of two modulus to another power of two modulus is fairly simple and very useful. The following discussion will assume unsigned integers that can be expressed in 16 binary digits (bits).

$$number \Rightarrow b_{15}\, b_{14}\, b_{13}\, b_{12}\, b_{11}\, b_{10}\, b_9\, b_8\, b_7\, b_6\, b_5\, b_4\, b_3\, b_2\, b_1\, b_0$$

$$\begin{aligned} number = {} & b_{15}\cdot 2^{15} + b_{14}\cdot 2^{14} + b_{13}\cdot 2^{13} + b_{12}\cdot 2^{12} + b_{11}\cdot 2^{11} + b_{10}\cdot 2^{10} + b_9\cdot 2^9 + b_8\cdot 2^8 \\ & + b_7\cdot 2^7 + b_6\cdot 2^6 + b_5\cdot 2^5 + b_4\cdot 2^4 + b_3\cdot 2^3 + b_2\cdot 2^2 + b_1\cdot 2^1 + b_0\cdot 2^0 \end{aligned}$$

2.2.1. Binary to/from Hexadecimal

Notice that the terms can be grouped in subsets of 4 as follows.

$$\begin{aligned} number = {} & \left(b_{15}\cdot 2^{15} + b_{14}\cdot 2^{14} + b_{13}\cdot 2^{13} + b_{12}\cdot 2^{12}\right) + \left(b_{11}\cdot 2^{11} + b_{10}\cdot 2^{10} + b_9\cdot 2^9 + b_8\cdot 2^8\right) \\ & + \left(b_7\cdot 2^7 + b_6\cdot 2^6 + b_5\cdot 2^5 + b_4\cdot 2^4\right) + \left(b_3\cdot 2^3 + b_2\cdot 2^2 + b_1\cdot 2^1 + b_0\cdot 2^0\right) \end{aligned}$$

Various powers of two can be factored out of the individual groups of 4 terms.

$$\begin{aligned} number = {} & \left(b_{15}\cdot 2^3 + b_{14}\cdot 2^2 + b_{13}\cdot 2^1 + b_{12}\cdot 2^0\right)\cdot 2^{12} + \left(b_{11}\cdot 2^3 + b_{10}\cdot 2^2 + b_9\cdot 2^1 + b_8\cdot 2^0\right)\cdot 2^8 \\ & + \left(b_7\cdot 2^3 + b_6\cdot 2^2 + b_5\cdot 2^1 + b_4\cdot 2^0\right)\cdot 2^4 + \left(b_3\cdot 2^3 + b_2\cdot 2^2 + b_1\cdot 2^1 + b_0\cdot 2^0\right) \end{aligned}$$

Realizing that $2^4 = 16$, the above can be transformed as follows.

$$\begin{aligned} number = {} & \left(b_{15}\cdot 2^3 + b_{14}\cdot 2^2 + b_{13}\cdot 2^1 + b_{12}\cdot 2^0\right)\cdot 16^3 + \left(b_{11}\cdot 2^3 + b_{10}\cdot 2^2 + b_9\cdot 2^1 + b_8\cdot 2^0\right)\cdot 16^2 \\ & + \left(b_7\cdot 2^3 + b_6\cdot 2^2 + b_5\cdot 2^1 + b_4\cdot 2^0\right)\cdot 16^1 + \left(b_3\cdot 2^3 + b_2\cdot 2^2 + b_1\cdot 2^1 + b_0\cdot 2^0\right)\cdot 16^0 \end{aligned}$$

The traditional decimal number system (base = 10) has the ten symbols (0, 1, 2, 3,4, 5, 6, 7, 8, 9). The hexadecimal system (base = 16) does not have a corresponding traditional set of 16 symbols. The set (0, 1, 2, 3,4, 5, 6, 7, 8, 9,A ,B, C, D, E, F) has been adopted.

Binary	Hexadecimal	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	5
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15
10000	10	16

The factors of the powers of 16 can be replaced with the symbols h_i

$$number = h_3 \cdot 16^3 + h_2 \cdot 16^2 + h_1 \cdot 16^1 + h_0 \cdot 16^0$$

$$\text{number} \Rightarrow h_3\, h_2\, h_1\, h_0$$

Thus, the binary number has been converted to the corresponding hexadecimal number with out any excessive arithmetic. Conversion of a hexadecimal number to the corresponding binary number is the inverse process.

As an example the following binary number will be converted to hexadecimal. First, the binary number is grouped into sets of 4 bits.

$$\text{number} = 000100001010101100011100_2$$

$$\text{number} = 0001\ 0000\ 1010\ 1011\ 0001\ 1100_2$$

Then, each set of four bits is replaced with the corresponding hexadecimal symbol. The number can then be regrouped.

$$\text{number} = 1\ 0\ A\ B\ 1\ C_{16}$$

$$\text{number} = 10AB1C_{16}$$

As a second example the following hexadecimal number will be converted to binary.

$$\text{number} = C730_{16}$$

$$\text{number} = C\ 7\ 3\ 0_{16}$$

Each hexadecimal symbol is then replaced with the corresponding set of four binary bits. The number can then be regrouped.

$$\text{number} = 1100\ 0111\ 0011\ 0000_2$$

$$\text{number} = 1100011100110000_2$$

2.2.2. Binary to/from Octal

In this case the terms in the binary representation are grouped into groups of three terms.

$$\begin{aligned} number = {} & \left(0 \cdot 2^{17} + 0 \cdot 2^{16} + b_{15} \cdot 2^{15}\right) + \left(b_{14} \cdot 2^{14} + b_{13} \cdot 2^{13} + b_{12} \cdot 2^{12}\right) \\ & + \left(b_{11} \cdot 2^{11} + b_{10} \cdot 2^{10} + b_9 \cdot 2^9\right) + \left(b_8 \cdot 2^8 + b_7 \cdot 2^7 + b_6 \cdot 2^6\right) \\ & + \left(b_5 \cdot 2^5 + b_4 \cdot 2^4 + b_3 \cdot 2^3\right) + \left(b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0\right) \end{aligned}$$

Now factor the appropriate power of two out of each group.

$$\begin{aligned} number = {} & \left(0 \cdot 2^2 + 0 \cdot 2^1 + b_{15} \cdot 2^0\right) \cdot 2^{15} + \left(b_{14} \cdot 2^2 + b_{13} \cdot 2^1 + b_{12} \cdot 2^0\right) \cdot 2^{12} \\ & + \left(b_{11} \cdot 2^2 + b_{10} \cdot 2^1 + b_9 \cdot 2^0\right) \cdot 2^9 + \left(b_8 \cdot 2^2 + b_7 \cdot 2^1 + b_6 \cdot 2^0\right) \cdot 2^6 \\ & + \left(b_5 \cdot 2^2 + b_4 \cdot 2^1 + b_3 \cdot 2^0\right) \cdot 2^3 + \left(b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0\right) \cdot 2^0 \end{aligned}$$

Given that $2^3 = 8$, the following transformation is made.

$$\begin{aligned} number = {} & \left(0\cdot 2^2 + 0\cdot 2^1 + b_{15}\cdot 2^0\right)\cdot 8^5 + \left(b_{14}\cdot 2^2 + b_{13}\cdot 2^1 + b_{12}\cdot 2^0\right)\cdot 8^4 \\ & + \left(b_{11}\cdot 2^2 + b_{10}\cdot 2^1 + b_9\cdot 2^0\right)\cdot 8^3 + \left(b_8\cdot 2^2 + b_7\cdot 2^1 + b_6\cdot 2^0\right)\cdot 8^2 \\ & + \left(b_5\cdot 2^2 + b_4\cdot 2^1 + b_3\cdot 2^0\right)\cdot 8^1 + \left(b_2\cdot 2^2 + b_1\cdot 2^1 + b_0\cdot 2^0\right)\cdot 8^0 \end{aligned}$$

The coefficients of the powers of 8 can be replaced by the octal symbols defined below.

Binary	Octal	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	5
0101	5	5
0110	6	6
0111	7	7
1000	10	8

$$\text{number} = o_5\cdot 8^5 + o_4\cdot 8^4 + o_3\cdot 8^3 + o_2\cdot 8^2 + o_1\cdot 8^1 + o_0\cdot 8^0$$

$$\text{number} \Rightarrow o_4\, o_3\, o_2\, o_1\, o_0$$

As an example the following binary number will be converted to octal. First, the binary number is grouped into sets of three bits.

$$\text{number} = 0001000001010101100011100_2$$

$$\text{number} = 000\ 100\ \ 001\ 010\ 101\ 100\ \ 011\ 100_2$$

Then each set of three bits is replaced with the corresponding octal symbol. The number can then be regrouped.

$$\text{number} = 0\ 4\ 1\ 2\ 5\ 4\ 3\ 4_8$$

$$\text{number} = 04125434_8$$

The inverse operation will serve as a second example. The following octal number will be converted to binary. Begin by separating the octal symbols.

$$\text{number} = 143460_8$$

$$\text{number} = 1\ 4\ 3\ 4\ 6\ 0_8$$

Each octal symbol is then replaced with the corresponding set of three binary bits. The number can then be regrouped.

$$\text{number} = 001\ 100\ 011\ 100\ 110\ 000_2$$

$$\text{number} = 1100011100110000_2$$

2.3. Signed Integers

A set of n binary bits, b_{n-1}, b_{n-2}, …, b_1, b_0, may also be used to represent a signed integer. Three different representations have been used.

2.3.1. Sign/Magnitude

This representation uses one bit to represent the sign of the number. The most significant bit is used for the sign bit. $b_s = b_{n-1} = 0$ for a positive number. $b_s = b_{n-1} = 1$ for a negative number. The absolute value of the number is placed in the remaining bits as an unsigned integer.

Examples, using 16 bit numbers:

Signed	Sign/Magnitude Representation			
Number	Binary	Oct	Dec	Hex
1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	1	1	1
-1	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	10001	32769	8001
32767	0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	77777	32767	7FFF
-32767	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	177777	65535	FFFF
0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	000000	0	0000

2.3.2. One's Complement

This representation again uses the most significant bit to represent the sign of the number. $b_{n-1} = b_s = 0$ for a positive number. $b_{n-1} = b_s = 1$ for a negative number. The absolute value of the number to be represented has to be less than $2^{n-1} - 1$. Convert the absolute value of the number to a binary number of n bits. Since the number is less than $2^{n-1} - 1$, the sign bit, $b_s = b_{n-1}$, will be zero. If the number being converted is negative, invert all n bits. Notice that the sign bit will be appropriate.

As an example convert the number 60_{10} to one's complement.

$0000\ 0000\ 0011\ 1100_2 = 60_{10} = 3C_{16}$ Convert absolute value to binary.

Finished.

Now convert -60_{10} to one's complement.

$0000\ 0000\ 0011\ 1100_2 = 60_{10} = 3C_{16}$ Convert absolute value to binary.

$1111\ 1111\ 1100\ 0011_2 = 65475_{10} = FFC3_{16}$ Invert all bits. Done

Further examples, using 16 bit numbers:

Signed	One's Complement Representation			
Number	Binary	Oct	Dec	Hex
1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	000001	1	0001
-1	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0	177776	65534	FFFE
32767	0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	077777	32767	7FFF
-32767	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	100000	32768	8000
0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	000000	0	0000
-0	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	177777	65535	FFFF

2.3.3. Two's Complement

This representation again uses the most significant bit to represent the sign of the number. $b_s = b_{n-1} = 0$ for a positive number. $b_s = b_{n-1} = 1$ for a negative number. Again, the absolute value of the number to be represented has to be less than $2^{n-1} - 1$. Convert the absolute value of the number to a binary number of n bits. Since the number is less than $2^{n-1} - 1$, the sign bit, $b_s = b_{n-1}$, will be zero. If the number being converted is negative, invert all n bits. Then add one to the resultant. Notice that the sign bit will be appropriate. This representation avoids the problem of +0 and -0 of the one's complement representation. This is now the typical representation used.

As an example convert the number 60_{10} to two's complement.

$0000\ 0000\ 0011\ 1100_2 = 60_{10} = 3C_{16}$ Convert absolute value to binary.

Finished.

Now convert -60_{10} to two's complement.

$0000\ 0000\ 0011\ 1100_2 = 60_{10} = 3C_{16}$ Convert absolute value to binary.

$1111\ 1111\ 1100\ 0011_2 = 65475_{10} = FFC3_{16}$ Invert all bits.

1_2 Add one.

$1111\ 1111\ 1100\ 0100_2 = 65476_{10} = FFC4_{16}$ Done.

Further examples, using 16 bit numbers:

Signed	Two's Complement			
Number	Binary	Oct	Dec	Hex
1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	1	1	1
-1	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	177777	65535	FFFF
32766	0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0	77776	32766	7FFE
32767	0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	77777	32767	7FFF
-32766	1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	100002	32770	8002
-32767	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	100001	32769	8001
-32768	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	100000	32768	8000

At this point, a few simple aritmetic examples might be useful. These examples use two's complement arithmetic. First there is the binary addition table for adding two binary single bit numbers (A + B). Multiple bit additions are performed bit by bit with the adding in of any carry from the previous position

A	B	A+B	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Add the binary equivalents of -60_{10} and +60_{10}.

1 1111 1111 1111 100 $_2$					Carry
0000 0000 0011 1100_2	=	60_{10}	=	$3C_{16}$	First number
1111 1111 1100 0100_2	=	-60_{10}	=	$FFC4_{16}$	Second number
0000 0000 0000 0000_2		0_{10}		0_{16}	Sum

2.4. Floating Point Numbers

This section summarizes the formats of several of the data types supported by Intel.[1] The value of a floating point number is given by the following

$$\text{number} = (-1)^S \left(2^{E-Bias}\right) F$$

where "S" is the sign bit, "E" is the exponent, "F" is the fractional mantissa, and "BIAS" is an integer that varies with representation and is listed below for these particular representations.

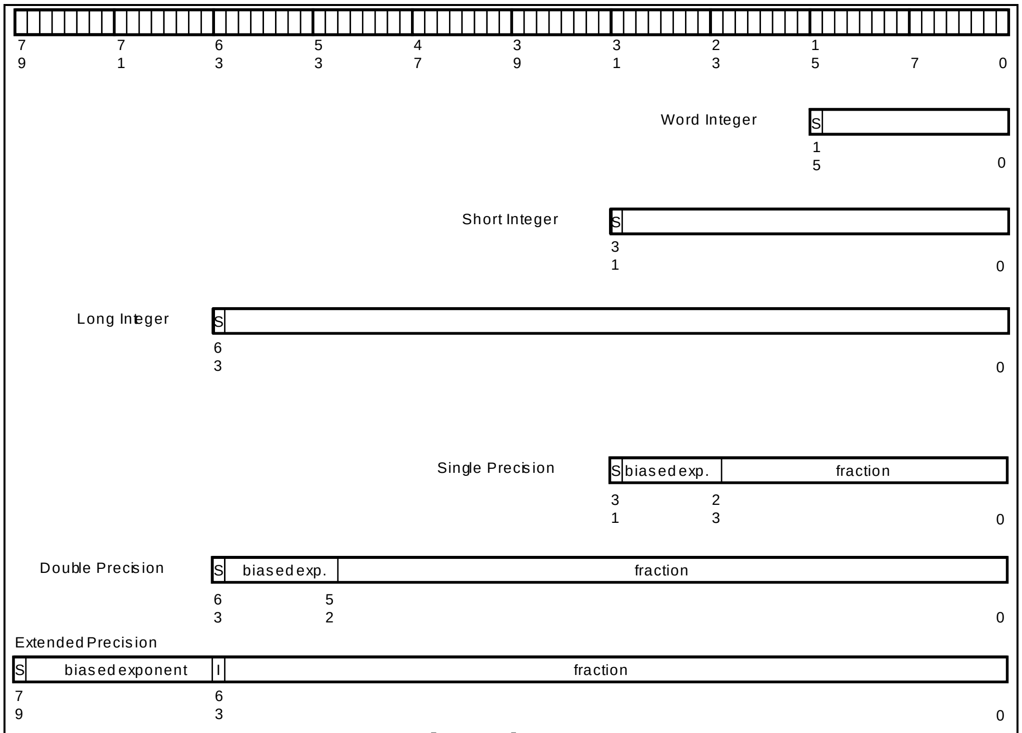


Figure 1 Intel Number Representations

[1] "Microprocessors," page 4-509, Intel Corporation, Literature Sales, PO Box 7641, Mt. Prospect IL 60056-7641, 1990.

Table 1 Number Formats

Data Type	Bits	Significant Digits	Range
Word Integer	16	4	$-32768 \le X \le 32767$
Short Integer	32	9	$-2x10^{9} \le X \le 2\ x10^{9}$
Long Integer	64	18	$-9x10^{18} \le X \le 9\ x10^{18}$
Single Precision	32	6-7	$8.43x10^{-37} \le \lvert X \rvert \le 3.37\ x10^{38}$
Double Precision	64	15-16	$4.19x10^{-307} \le \lvert X \rvert \le 1.67\ x10^{308}$
Extended Precision	80	19	$3.4x10^{-4932} \le \lvert X \rvert \le 1.2\ x10^{4932}$

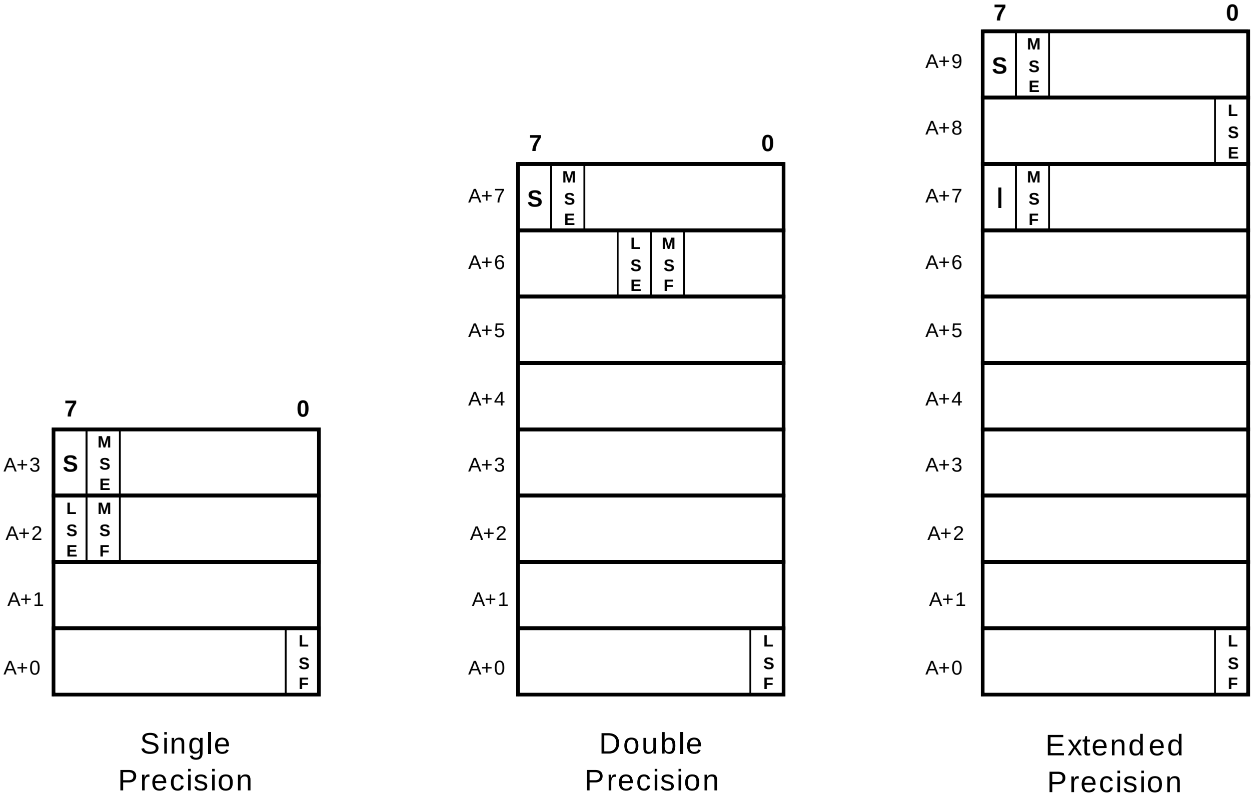


Figure 2 Intel Floating Point Storage

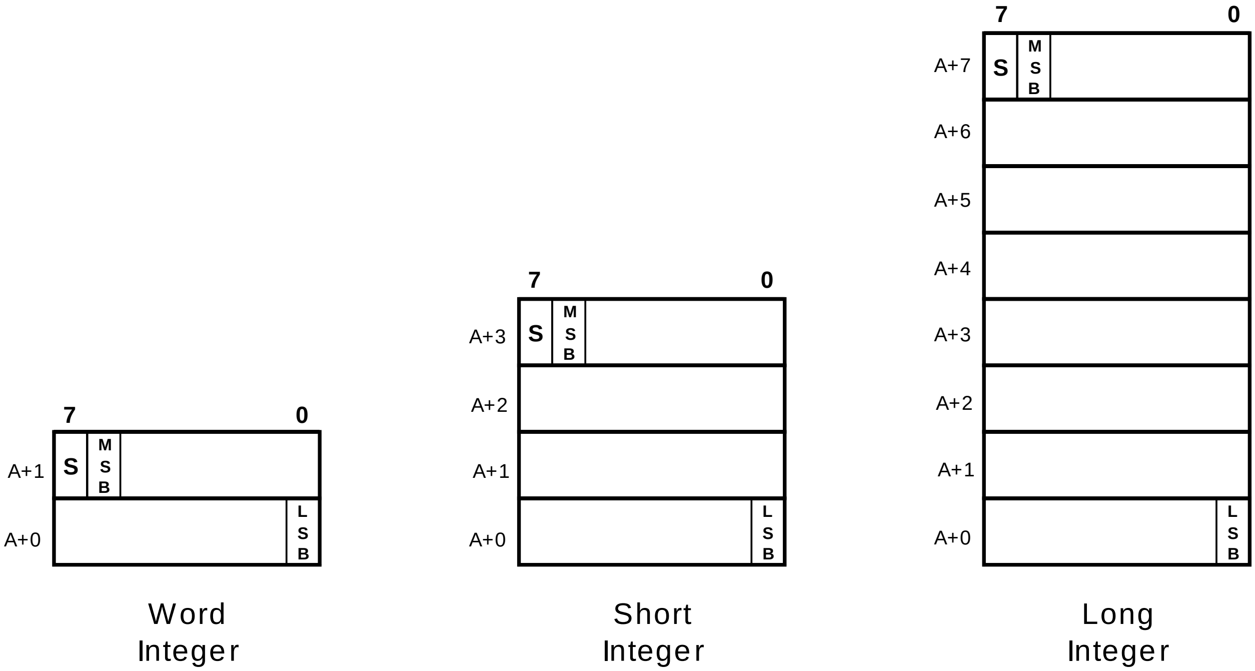


Figure 3 Intel Integer Storage

Table 2 Symbol Definitions

Symbol	Description
A	Base address of the stored number
S	Sign of the number (0 = positive, 1 = negative)
MSB	Most Significant Bit of integer
LSB	Least Significant Bit of Integer
MSF	Most Significant Bit of fraction
LSF	Least Significant Bit of fraction
MSE	Most Significant Bit of the exponent
LSE	Least Significant Bit of the exponent
Bias	Single 127 ($7F_{16}$)
	Double 1023 ($3FF_{16}$)
	Extended 16383 ($3FFF_{16}$)

2.5. Useful Tables of Numbers

2.5.1. Powers of Two

Table 3 Powers of 2

n	DEC	OCT	HEX	Common Name
0	1	1	1	
1	2	2	2	
2	4	4	4	
3	8	10	8	
4	16	20	10	
5	32	40	20	
6	64	100	40	
7	128	200	80	
8	256	400	100	
9	512	1000	200	
10	1024	2000	400	1K
11	2048	4000	800	2K
12	4096	10000	1000	4K
13	8192	20000	2000	8K
14	16384	40000	4000	16K
15	32768	100000	8000	32K
16	65536	200000	10000	64K
17	131072	400000	20000	128K
18	262144	1000000	40000	256K
19	524288	2000000	80000	512K
20	1048576	4000000	100000	1M or 1Meg
21	2097152	1000000	200000	2M or 2Meg
22	4194304	20000000	400000	4M or 4Meg
23	8388608	40000000	800000	8M or 8Meg
24	16777216	100000000	1000000	16M or 16Meg
25	33554432	200000000	2000000	32M or 32Meg
26	67108864	400000000	4000000	64M or 64Meg
27	134217728	1000000000	8000000	128M or 128Meg
28	268435456	2000000000	10000000	256M or 256Meg
29	536870912	4000000000	20000000	512M or 512Meg
30	1073741824	10000000000	40000000	1G or 1Gig
31	2147483648	20000000000	80000000	2G or 2Gig
32	4294967296	40000000000	100000000	4G or 4Gig

The above table contains the values of the first 32 powers of 2 expressed in base 10 (decimal or DEC), base 8 (octal or Oct), and base 16 (hexadecimal or Hex). The right most column of the table contains the common names often given to the corresponding quantities. This nomenclature is an artifact of the computer industry which early on chose to use the short hand name “one K” to represent the much longer and more appropriate name “One thousand twenty four,” etc.

Table 4 Counting in Different Moduli

DEC	Binary	OCT	HEX	Items being counted
0	0000000000000000	0	0	
1	0000000000000001	1	1	*
2	0000000000000010	2	2	**
3	0000000000000011	3	3	***
4	0000000000000100	4	4	****
5	0000000000000101	5	5	*****
6	0000000000000110	6	6	******
7	0000000000000111	7	7	*******
8	0000000000001000	10	8	********
9	0000000000001001	11	9	*********
10	0000000000001010	12	A	**********
11	0000000000001011	13	B	***********
12	0000000000001100	14	C	************
13	0000000000001101	15	D	*************
14	0000000000001110	16	E	**************
15	0000000000001111	17	F	***************
16	0000000000010000	20	10	****************
17	0000000000010001	21	11	*****************
18	0000000000010010	22	12	******************
19	0000000000010011	23	13	*******************
20	0000000000010100	24	14	********************
21	0000000000010101	25	15	*********************
22	0000000000010110	26	16	**********************
23	0000000000010111	27	17	***********************
24	0000000000011000	30	18	************************
25	0000000000011001	31	19	*************************
26	0000000000011010	32	1A	**************************
27	0000000000011011	33	1B	***************************
28	0000000000011100	34	1C	****************************
29	0000000000011101	35	1D	*****************************
30	0000000000011110	36	1E	******************************
31	0000000000011111	37	1F	*******************************
32	0000000000100000	40	20	********************************
33	0000000000100001	41	21	*********************************

Table 5 0 to 65536 in Multiple Moduli

DEC	BIN	BIN	OCT	BIN	HEX	DEC
0	0000000000000000	0 000 000 000 000 000	0	0000 0000 0000 0000	0	0
1	0000000000000001	0 000 000 000 000 001	1	0000 0000 0000 0001	1	1
2	0000000000000010	0 000 000 000 000 010	2	0000 0000 0000 0010	2	2
3	0000000000000011	0 000 000 000 000 011	3	0000 0000 0000 0011	3	3
4	0000000000000100	0 000 000 000 000 100	4	0000 0000 0000 0100	4	4
5	0000000000000101	0 000 000 000 000 101	5	0000 0000 0000 0101	5	5
6	0000000000000110	0 000 000 000 000 110	6	0000 0000 0000 0110	6	6
7	0000000000000111	0 000 000 000 000 111	7	0000 0000 0000 0111	7	7
8	0000000000001000	0 000 000 000 001 000	10	0000 0000 0000 1000	8	8
9	0000000000001001	0 000 000 000 001 001	11	0000 0000 0000 1001	9	9
10	0000000000001010	0 000 000 000 001 010	12	0000 0000 0000 1010	A	10
11	0000000000001011	0 000 000 000 001 011	13	0000 0000 0000 1011	B	11
12	0000000000001100	0 000 000 000 001 100	14	0000 0000 0000 1100	C	12
13	0000000000001101	0 000 000 000 001 101	15	0000 0000 0000 1101	D	13
14	0000000000001110	0 000 000 000 001 110	16	0000 0000 0000 1110	E	14
15	0000000000001111	0 000 000 000 001 111	17	0000 0000 0000 1111	F	15
16	0000000000010000	0 000 000 000 010 000	20	0000 0000 0001 0000	10	16
17	0000000000010001	0 000 000 000 010 001	21	0000 0000 0001 0001	11	17
18	0000000000010010	0 000 000 000 010 010	22	0000 0000 0001 0010	12	18
19	0000000000010011	0 000 000 000 010 011	23	0000 0000 0001 0011	13	19
20	0000000000010100	0 000 000 000 010 100	24	0000 0000 0001 0100	14	20
21	0000000000010101	0 000 000 000 010 101	25	0000 0000 0001 0101	15	21
22	0000000000010110	0 000 000 000 010 110	26	0000 0000 0001 0110	16	22
23	0000000000010111	0 000 000 000 010 111	27	0000 0000 0001 0111	17	23
24	0000000000011000	0 000 000 000 011 000	30	0000 0000 0001 1000	18	24
25	0000000000011001	0 000 000 000 011 001	31	0000 0000 0001 1001	19	25
26	0000000000011010	0 000 000 000 011 010	32	0000 0000 0001 1010	1A	26
27	0000000000011011	0 000 000 000 011 011	33	0000 0000 0001 1011	1B	27
28	0000000000011100	0 000 000 000 011 100	34	0000 0000 0001 1100	1C	28
29	0000000000011101	0 000 000 000 011 101	35	0000 0000 0001 1101	1D	29
30	0000000000011110	0 000 000 000 011 110	36	0000 0000 0001 1110	1E	30
31	0000000000011111	0 000 000 000 011 111	37	0000 0000 0001 1111	1F	31
32	0000000000100000	0 000 000 000 100 000	40	0000 0000 0010 0000	20	32
33	0000000000100001	0 000 000 000 100 001	41	0000 0000 0010 0001	21	33
34	0000000000100010	0 000 000 000 100 010	42	0000 0000 0010 0010	22	34
35	0000000000100011	0 000 000 000 100 011	43	0000 0000 0010 0011	23	35
36	0000000000100100	0 000 000 000 100 100	44	0000 0000 0010 0100	24	36
37	0000000000100101	0 000 000 000 100 101	45	0000 0000 0010 0101	25	37
38	0000000000100110	0 000 000 000 100 110	46	0000 0000 0010 0110	26	38
39	0000000000100111	0 000 000 000 100 111	47	0000 0000 0010 0111	27	39
40	0000000000101000	0 000 000 000 101 000	50	0000 0000 0010 1000	28	40

DEC	BIN	BIN	OCT	BIN	HEX	DEC
41	0000000000101001	0 000 000 000 101 001	51	0000 0000 0010 1001	29	41
42	0000000000101010	0 000 000 000 101 010	52	0000 0000 0010 1010	2A	42
43	0000000000101011	0 000 000 000 101 011	53	0000 0000 0010 1011	2B	43
44	0000000000101100	0 000 000 000 101 100	54	0000 0000 0010 1100	2C	44
45	0000000000101101	0 000 000 000 101 101	55	0000 0000 0010 1101	2D	45
46	0000000000101110	0 000 000 000 101 110	56	0000 0000 0010 1110	2E	46
47	0000000000101111	0 000 000 000 101 111	57	0000 0000 0010 1111	2F	47
48	0000000000110000	0 000 000 000 110 000	60	0000 0000 0011 0000	30	48
49	0000000000110001	0 000 000 000 110 001	61	0000 0000 0011 0001	31	49
50	0000000000110010	0 000 000 000 110 010	62	0000 0000 0011 0010	32	50
51	0000000000110011	0 000 000 000 110 011	63	0000 0000 0011 0011	33	51
52	0000000000110100	0 000 000 000 110 100	64	0000 0000 0011 0100	34	52
53	0000000000110101	0 000 000 000 110 101	65	0000 0000 0011 0101	35	53
54	0000000000110110	0 000 000 000 110 110	66	0000 0000 0011 0110	36	54
55	0000000000110111	0 000 000 000 110 111	67	0000 0000 0011 0111	37	55
56	0000000000111000	0 000 000 000 111 000	70	0000 0000 0011 1000	38	56
57	0000000000111001	0 000 000 000 111 001	71	0000 0000 0011 1001	39	57
58	0000000000111010	0 000 000 000 111 010	72	0000 0000 0011 1010	3A	58
59	0000000000111011	0 000 000 000 111 011	73	0000 0000 0011 1011	3B	59
60	0000000000111100	0 000 000 000 111 100	74	0000 0000 0011 1100	3C	60
61	0000000000111101	0 000 000 000 111 101	75	0000 0000 0011 1101	3D	61
62	0000000000111110	0 000 000 000 111 110	76	0000 0000 0011 1110	3E	62
63	0000000000111111	0 000 000 000 111 111	77	0000 0000 0011 1111	3F	63
64	0000000001000000	0 000 000 001 000 000	100	0000 0000 0100 0000	40	64
65	0000000001000001	0 000 000 001 000 001	101	0000 0000 0100 0001	41	65
66	0000000001000010	0 000 000 001 000 010	102	0000 0000 0100 0010	42	66
67	0000000001000011	0 000 000 001 000 011	103	0000 0000 0100 0011	43	67
68	0000000001000100	0 000 000 001 000 100	104	0000 0000 0100 0100	44	68
69	0000000001000101	0 000 000 001 000 101	105	0000 0000 0100 0101	45	69
70	0000000001000110	0 000 000 001 000 110	106	0000 0000 0100 0110	46	70
71	0000000001000111	0 000 000 001 000 111	107	0000 0000 0100 0111	47	71
72	0000000001001000	0 000 000 001 001 000	110	0000 0000 0100 1000	48	72
73	0000000001001001	0 000 000 001 001 001	111	0000 0000 0100 1001	49	73
74	0000000001001010	0 000 000 001 001 010	112	0000 0000 0100 1010	4A	74
75	0000000001001011	0 000 000 001 001 011	113	0000 0000 0100 1011	4B	75
76	0000000001001100	0 000 000 001 001 100	114	0000 0000 0100 1100	4C	76
77	0000000001001101	0 000 000 001 001 101	115	0000 0000 0100 1101	4D	77
78	0000000001001110	0 000 000 001 001 110	116	0000 0000 0100 1110	4E	78
79	0000000001001111	0 000 000 001 001 111	117	0000 0000 0100 1111	4F	79
80	0000000001010000	0 000 000 001 010 000	120	0000 0000 0101 0000	50	80

DEC	BIN	BIN	OCT	BIN	HEX	DEC
81	0000000001010001	0 000 000 001 010 001	121	0000 0000 0101 0001	51	81
82	0000000001010010	0 000 000 001 010 010	122	0000 0000 0101 0010	52	82
83	0000000001010011	0 000 000 001 010 011	123	0000 0000 0101 0011	53	83
84	0000000001010100	0 000 000 001 010 100	124	0000 0000 0101 0100	54	84
85	0000000001010101	0 000 000 001 010 101	125	0000 0000 0101 0101	55	85
86	0000000001010110	0 000 000 001 010 110	126	0000 0000 0101 0110	56	86
87	0000000001010111	0 000 000 001 010 111	127	0000 0000 0101 0111	57	87
88	0000000001011000	0 000 000 001 011 000	130	0000 0000 0101 1000	58	88
89	0000000001011001	0 000 000 001 011 001	131	0000 0000 0101 1001	59	89
90	0000000001011010	0 000 000 001 011 010	132	0000 0000 0101 1010	5A	90
91	0000000001011011	0 000 000 001 011 011	133	0000 0000 0101 1011	5B	91
92	0000000001011100	0 000 000 001 011 100	134	0000 0000 0101 1100	5C	92
93	0000000001011101	0 000 000 001 011 101	135	0000 0000 0101 1101	5D	93
94	0000000001011110	0 000 000 001 011 110	136	0000 0000 0101 1110	5E	94
95	0000000001011111	0 000 000 001 011 111	137	0000 0000 0101 1111	5F	95
96	0000000001100000	0 000 000 001 100 000	140	0000 0000 0110 0000	60	96
97	0000000001100001	0 000 000 001 100 001	141	0000 0000 0110 0001	61	97
98	0000000001100010	0 000 000 001 100 010	142	0000 0000 0110 0010	62	98
99	0000000001100011	0 000 000 001 100 011	143	0000 0000 0110 0011	63	99
100	0000000001100100	0 000 000 001 100 100	144	0000 0000 0110 0100	64	100
101	0000000001100101	0 000 000 001 100 101	145	0000 0000 0110 0101	65	101
102	0000000001100110	0 000 000 001 100 110	146	0000 0000 0110 0110	66	102
103	0000000001100111	0 000 000 001 100 111	147	0000 0000 0110 0111	67	103
104	0000000001101000	0 000 000 001 101 000	150	0000 0000 0110 1000	68	104
105	0000000001101001	0 000 000 001 101 001	151	0000 0000 0110 1001	69	105
106	0000000001101010	0 000 000 001 101 010	152	0000 0000 0110 1010	6A	106
107	0000000001101011	0 000 000 001 101 011	153	0000 0000 0110 1011	6B	107
108	0000000001101100	0 000 000 001 101 100	154	0000 0000 0110 1100	6C	108
109	0000000001101101	0 000 000 001 101 101	155	0000 0000 0110 1101	6D	109
110	0000000001101110	0 000 000 001 101 110	156	0000 0000 0110 1110	6E	110
111	0000000001101111	0 000 000 001 101 111	157	0000 0000 0110 1111	6F	111
112	0000000001110000	0 000 000 001 110 000	160	0000 0000 0111 0000	70	112
113	0000000001110001	0 000 000 001 110 001	161	0000 0000 0111 0001	71	113
114	0000000001110010	0 000 000 001 110 010	162	0000 0000 0111 0010	72	114
115	0000000001110011	0 000 000 001 110 011	163	0000 0000 0111 0011	73	115
116	0000000001110100	0 000 000 001 110 100	164	0000 0000 0111 0100	74	116
117	0000000001110101	0 000 000 001 110 101	165	0000 0000 0111 0101	75	117
118	0000000001110110	0 000 000 001 110 110	166	0000 0000 0111 0110	76	118
119	0000000001110111	0 000 000 001 110 111	167	0000 0000 0111 0111	77	119
120	0000000001111000	0 000 000 001 111 000	170	0000 0000 0111 1000	78	120

DEC	BIN	BIN	OCT	BIN	HEX	DEC
121	0000000001111001	0 000 000 001 111 001	171	0000 0000 0111 1001	79	121
122	0000000001111010	0 000 000 001 111 010	172	0000 0000 0111 1010	7A	122
123	0000000001111011	0 000 000 001 111 011	173	0000 0000 0111 1011	7B	123
124	0000000001111100	0 000 000 001 111 100	174	0000 0000 0111 1100	7C	124
125	0000000001111101	0 000 000 001 111 101	175	0000 0000 0111 1101	7D	125
126	0000000001111110	0 000 000 001 111 110	176	0000 0000 0111 1110	7E	126
127	0000000001111111	0 000 000 001 111 111	177	0000 0000 0111 1111	7F	127
128	0000000010000000	0 000 000 010 000 000	200	0000 0000 1000 0000	80	128
129	0000000010000001	0 000 000 010 000 001	201	0000 0000 1000 0001	81	129
130	0000000010000010	0 000 000 010 000 010	202	0000 0000 1000 0010	82	130
65530	1111111111111010	1 111 111 111 111 010	177772	1111 1111 1111 1010	FFFA	65530
65531	1111111111111011	1 111 111 111 111 011	177773	1111 1111 1111 1011	FFFB	65531
65532	1111111111111100	1 111 111 111 111 100	177774	1111 1111 1111 1100	FFFC	65532
65533	1111111111111101	1 111 111 111 111 101	177775	1111 1111 1111 1101	FFFD	65533
65534	1111111111111110	1 111 111 111 111 110	177776	1111 1111 1111 1110	FFFE	65534
65535	1111111111111111	1 111 111 111 111 111	177777	1111 1111 1111 1111	FFFF	65535
65536	0000000000000000	0 000 000 000 000 000	0	0000 0000 0000 0000	0	65536
65537	0000000000000001	0 000 000 000 000 001	1	0000 0000 0000 0001	1	65537
65538	0000000000000010	0 000 000 000 000 010	2	0000 0000 0000 0010	2	65538
65539	0000000000000011	0 000 000 000 000 011	3	0000 0000 0000 0011	3	65539
65540	0000000000000100	0 000 000 000 000 100	4	0000 0000 0000 0100	4	65540

3. Character Codes

3.1. Six Bit Character Codes

These codes were used in the early days of computing when memory and bandwidth was very expensive. Notice that there are only upper case characters.

Table 6 Six Bit Character Codes

Char	Octal	Dec	Hex	Char	Octal	Dec	Hex
@	0	0	0	space	40	32	20
A	1	1	1	!	41	33	21
B	2	2	2	"	42	34	22
C	3	3	3	#	43	35	23
D	4	4	4	$	44	36	24
E	5	5	5	%	45	37	25
F	6	6	6	&	46	38	26
G	7	7	7	'	47	39	27
H	10	8	8	(	50	40	28
I	11	9	9	)	51	41	29
J	12	10	A	*	52	42	2A
K	13	11	B	+	53	43	2B
L	14	12	C	,	54	44	2C
M	15	13	D	-	55	45	2D
N	16	14	E	.	56	46	2E
O	17	15	F	/	57	47	2F
P	20	16	10	0	60	48	30
Q	21	17	11	1	61	49	31
R	22	18	12	2	62	50	32
S	23	19	13	3	63	51	33
T	24	20	14	4	64	52	34
U	25	21	15	5	65	53	35
V	26	22	16	6	66	54	36
W	27	23	17	7	67	55	37
X	30	24	18	8	70	56	38
Y	31	25	19	9	71	57	39
Z	32	26	1A	:	72	58	3A
[	33	27	1B	;	73	59	3B
\	34	28	1C	<	74	60	3C
]	35	29	1D	=	75	61	3D
^	36	30	1E	>	76	62	3E
	37	31	1F	?	77	63	3F

3.2. ASCII Character Codes

Table 7 ASCII Character Codes

Character	Octal	Dec	Hex	Char	Octal	Dec	Hex	Char	Octal	Dec	Hex	Char	Octal	Dec	Hex
<NULL>	0	0	0		40	32	20	@	100	64	40	`	140	96	60
<SOH>	1	1	1	!	41	33	21	A	101	65	41	a	141	97	61
<STX>	2	2	2	"	42	34	22	B	102	66	42	b	142	98	62
<ETX>	3	3	3	#	43	35	23	C	103	67	43	c	143	99	63
<EOT>	4	4	4	$	44	36	24	D	104	68	44	d	144	100	64
<ENQ>	5	5	5	%	45	37	25	E	105	69	45	e	145	101	65
<ACK>	6	6	6	&	46	38	26	F	106	70	46	f	146	102	66
<BEL>	7	7	7	'	47	39	27	G	107	71	47	g	147	103	67
<BS>	10	8	8	(	50	40	28	H	110	72	48	h	150	104	68
<HT>	11	9	9	)	51	41	29	I	111	73	49	i	151	105	69
<LF>	12	10	A	*	52	42	2A	J	112	74	4A	j	152	106	6A
<VT>	13	11	B	+	53	43	2B	K	113	75	4B	k	153	107	6B
<FF>	14	12	C	,	54	44	2C	L	114	76	4C	l	154	108	6C
<CR>	15	13	D	-	55	45	2D	M	115	77	4D	m	155	109	6D
<SO>	16	14	E	.	56	46	2E	N	116	78	4E	n	156	110	6E
<SI>	17	15	F	/	57	47	2F	O	117	79	4F	o	157	111	6F
<DLE>	20	16	10	0	60	48	30	P	120	80	50	p	160	112	70
<DC1>	21	17	11	1	61	49	31	Q	121	81	51	q	161	113	71
<DC2>	22	18	12	2	62	50	32	R	122	82	52	r	162	114	72
<DC3>	23	19	13	3	63	51	33	S	123	83	53	s	163	115	73
<DC4>	24	20	14	4	64	52	34	T	124	84	54	t	164	116	74
<NAK>	25	21	15	5	65	53	35	U	125	85	55	u	165	117	75
<SYN>	26	22	16	6	66	54	36	V	126	86	56	v	166	118	76
<ETB>	27	23	17	7	67	55	37	W	127	87	57	w	167	119	77
<CAN>	30	24	18	8	70	56	38	X	130	88	58	x	170	120	78
<EM>	31	25	19	9	71	57	39	Y	131	89	59	y	171	121	79
<SUB>	32	26	1A	:	72	58	3A	Z	132	90	5A	z	172	122	7A
<ESC>	33	27	1B	;	73	59	3B	[	133	91	5B	{	173	123	7B
<FS>	34	28	1C	<	74	60	3C	\	134	92	5C	|	174	124	7C
<GS>	35	29	1D	=	75	61	3D	]	135	93	5D	}	175	125	7D
<RS>	36	30	1E	>	76	62	3E	^	136	94	5E	~	176	126	7E
<US>	37	31	1F	?	77	63	3F	_	137	95	5F	DEL	177	127	7F

Table 8 ASCII Control Characters

<NUL>	Null
<SOH>	Start of heading
<STX>	Start of text
<ETX>	End of text
<EOT>	End of transmission
<ENQ>	Enquiry
<ACK>	Acknowledge
<BEL>	Bell (audible signal)
<BS>	Backspace
<HT>	Horizontal Tabulation
<LF>	Line Feed - go to new line
<VT>	Vertical tabulation
<FF>	Form Feed - go to new page
<CR>	Carriage return - return to left margin
<SO>	Shift out
<SI>	Shift in
<DLE>	Data link escape
<DC1>	Device Control 1 - XON
<DC2>	Device Control 2
<DC3>	Device Control 3 - XOFF
<DC4>	Device Control 4
<NAK>	Negative Acknowledge
<SYN>	Synchronous idle
<ETB>	End of transmission block
<CAN>	Cancel
<EM>	End of medium
<SUB>	Substitute
<ESC>	Escape
<FS>	File Separator
<GS>	Group Separator
<RS>	Record Separator
<US>	Unit Separator
<DEL>	Delete

3.3. ANSI Character Codes

Table 9 ANSI Character Set

The ANSI character set consists of the ASCII character set plus the set of characters in this table.

Char	Octal	Dec	Hex	Char	Octal	Dec	Hex	Char	Octal	Dec	Hex	Char	Octal	Dec	Hex
	200	128	80		240	160	A0	À	300	192	C0	à	340	224	E0
	201	129	81	¡	241	161	A1	Á	301	193	C1	á	341	225	E1
‚	202	130	82	¢	242	162	A2	Â	302	194	C2	â	342	226	E2
ƒ	203	131	83	£	243	163	A3	Ã	303	195	C3	ã	343	227	E3
„	204	132	84	¤	244	164	A4	Ä	304	196	C4	ä	344	228	E4
…	205	133	85	¥	245	165	A5	Å	305	197	C5	å	345	229	E5
†	206	134	86	¦	246	166	A6	Æ	306	198	C6	æ	346	230	E6
‡	207	135	87	§	247	167	A7	Ç	307	199	C7	ç	347	231	E7
ˆ	210	136	88	¨	250	168	A8	È	310	200	C8	è	350	232	E8
‰	211	137	89	©	251	169	A9	É	311	201	C9	é	351	233	E9
Š	212	138	8A	ª	252	170	AA	Ê	312	202	CA	ê	352	234	EA
‹	213	139	8B	«	253	171	AB	Ë	313	203	CB	ë	353	235	EB
Œ	214	140	8C	¬	254	172	AC	Ì	314	204	CC	ì	354	236	EC
	215	141	8D	-	255	173	AD	Í	315	205	CD	í	355	237	ED
	216	142	8E	®	256	174	AE	Î	316	206	CE	î	356	238	EE
	217	143	8F	¯	257	175	AF	Ï	317	207	CF	ï	357	239	EF
	220	144	90	°	260	176	B0	Ð	320	208	D0	ð	360	240	F0
‘	221	145	91	±	261	177	B1	Ñ	321	209	D1	ñ	361	241	F1
’	222	146	92	²	262	178	B2	Ò	322	210	D2	ò	362	242	F2
“	223	147	93	³	263	179	B3	Ó	323	211	D3	ó	363	243	F3
”	224	148	94	´	264	180	B4	Ô	324	212	D4	ô	364	244	F4
•	225	149	95	µ	265	181	B5	Õ	325	213	D5	õ	365	245	F5
–	226	150	96	¶	266	182	B6	Ö	326	214	D6	ö	366	246	F6
—	227	151	97	·	267	183	B7	×	327	215	D7	÷	367	247	F7
˜	230	152	98	¸	270	184	B8	Ø	330	216	D8	ø	370	248	F8
™	231	153	99	¹	271	185	B9	Ù	331	217	D9	ù	371	249	F9
š	232	154	9A	º	272	186	BA	Ú	332	218	DA	ú	372	250	FA
›	233	155	9B	»	273	187	BB	Û	333	219	DB	û	373	251	FB
œ	234	156	9C	¼	274	188	BC	Ü	334	220	DC	ü	374	252	FC
	235	157	9D	½	275	189	BD	Ý	335	221	DD	ý	375	253	FD
	236	158	9E	¾	276	190	BE	Þ	336	222	DE	þ	376	254	FE
Ÿ	237	159	9F	¿	277	191	BF	ß	337	223	DF	ÿ	377	255	FF

3.4. Unicode Character Codes

In order to deal with the many character sets used in the written languages of the world, the Unicode Character Codes were developed over the last decade. The standard allows 8, 16, or 32 bit definitions of the characters. The following URL contains the details, and there are many, of the Unicode effort.

http://www.unicode.org/

The table below contains the lay out of the Unicode Character Sets.

`http://www.unicode.org/Public/UNIDATA/Blocks.txt`

```
# Unicode Character Database
# Copyright (c) 1991-2004 Unicode, Inc.
# For terms of use, see http://www.unicode.org/terms_of_use.html
# For documentation, see UCD.html
```

Table 10 UNICODE Character Codes

0000	007F	Basic Latin
0080	00FF	Latin-1 Supplement
0100	017F	Latin Extended-A
0180	024F	Latin Extended-B
0250	02AF	IPA Extensions
02B0	02FF	Spacing Modifier Letters
0300	036F	Combining Diacritical Marks
0370	03FF	Greek and Coptic
0400	04FF	Cyrillic
0500	052F	Cyrillic Supplement
0530	058F	Armenian
0590	05FF	Hebrew
0600	06FF	Arabic
0700	074F	Syriac
0780	07BF	Thaana
0900	097F	Devanagari
0980	09FF	Bengali
0A00	0A7F	Gurmukhi
0A80	0AFF	Gujarati
0B00	0B7F	Oriya
0B80	0BFF	Tamil
0C00	0C7F	Telugu
0C80	0CFF	Kannada
0D00	0D7F	Malayalam
0D80	0DFF	Sinhala
0E00	0E7F	Thai
0E80	0EFF	Lao
0F00	0FFF	Tibetan
1000	109F	Myanmar
10A0	10FF	Georgian
1100	11FF	Hangul Jamo
1200	137F	Ethiopic
13A0	13FF	Cherokee
1400	167F	Unified Canadian Aboriginal Syllabics
1680	169F	Ogham
16A0	16FF	Runic
1700	171F	Tagalog
1720	173F	Hanunoo
1740	175F	Buhid
1760	177F	Tagbanwa
1780	17FF	Khmer
1800	18AF	Mongolian
1900	194F	Limbu
1950	197F	Tai Le
19E0	19FF	Khmer Symbols
1D00	1D7F	Phonetic Extensions
1E00	1EFF	Latin Extended Additional
1F00	1FFF	Greek Extended
2000	206F	General Punctuation
2070	209F	Superscripts and Subscripts
20A0	20CF	Currency Symbols
20D0	20FF	Combining Diacritical Marks for Symbols
2100	214F	Letterlike Symbols
2150	218F	Number Forms
2190	21FF	Arrows
2200	22FF	Mathematical Operators
2300	23FF	Miscellaneous Technical
2400	243F	Control Pictures
2440	245F	Optical Character Recognition
2460	24FF	Enclosed Alphanumerics
2500	257F	Box Drawing
2580	259F	Block Elements
25A0	25FF	Geometric Shapes
2600	26FF	Miscellaneous Symbols
2700	27BF	Dingbats
27C0	27EF	Miscellaneous Mathematical Symbols-A
27F0	27FF	Supplemental Arrows-A
2800	28FF	Braille Patterns
2900	297F	Supplemental Arrows-B
2980	29FF	Miscellaneous Mathematical Symbols-B
2A00	2AFF	Supplemental Mathematical Operators
2B00	2BFF	Miscellaneous Symbols and Arrows
2E80	2EFF	CJK Radicals Supplement
2F00	2FDF	Kangxi Radicals
2FF0	2FFF	Ideographic Description Characters
3000	303F	CJK Symbols and Punctuation
3040	309F	Hiragana
30A0	30FF	Katakana
3100	312F	Bopomofo
3130	318F	Hangul Compatibility Jamo
3190	319F	Kanbun
31A0	31BF	Bopomofo Extended
31F0	31FF	Katakana Phonetic Extensions
3200	32FF	Enclosed CJK Letters and Months
3300	33FF	CJK Compatibility
3400	4DBF	CJK Unified Ideographs Extension A
4DC0	4DFF	Yijing Hexagram Symbols
4E00	9FFF	CJK Unified Ideographs
A000	A48F	Yi Syllables
A490	A4CF	Yi Radicals
AC00	D7AF	Hangul Syllables
D800	DB7F	High Surrogates
DB80	DBFF	High Private Use Surrogates
DC00	DFFF	Low Surrogates
E000	F8FF	Private Use Area
F900	FAFF	CJK Compatibility Ideographs
FB00	FB4F	Alphabetic Presentation Forms
FB50	FDFF	Arabic Presentation Forms-A
FE00	FE0F	Variation Selectors
FE20	FE2F	Combining Half Marks
FE30	FE4F	CJK Compatibility Forms
FE50	FE6F	Small Form Variants
FE70	FEFF	Arabic Presentation Forms-B
FF00	FFEF	Halfwidth and Fullwidth Forms
FFF0	FFFF	Specials
10000	1007F	Linear B Syllabary
10080	100FF	Linear B Ideograms
10100	1013F	Aegean Numbers
10300	1032F	Old Italic
10330	1034F	Gothic
10380	1039F	Ugaritic
10400	1044F	Deseret
10450	1047F	Shavian
10480	104AF	Osmanya
10800	1083F	Cypriot Syllabary
1D000	1D0FF	Byzantine Musical Symbols
1D100	1D1FF	Musical Symbols
1D300	1D35F	Tai Xuan Jing Symbols
1D400	1D7FF	Mathematical Alphanumeric Symbols
20000	2A6DF	CJK Unified Ideographs Extension B
2F800	2FA1F	CJK Compatibility Ideographs Supplement
E0000	E007F	Tags
E0100	E01EF	Variation Selectors Supplement
F0000	FFFFF	Supplementary Private Use Area-A
100000	10FFFF	Supplementary Private Use Area-B

4. Logic

4.1. Single Bit Logic Truth Tables

Table 11 Logic Truth Tables

A	B	$\overline{A}$	$\overline{B}$	$A \bullet B$ A.AND.B	$A + B$ A.OR.B	$\overline{A} \bullet \overline{B}$ $\overline{\text{A}}$.AND.$\overline{\text{B}}$	$\overline{A} + \overline{B}$ $\overline{\text{A}}$.OR.$\overline{\text{B}}$	$\overline{A \bullet B}$ $\overline{\text{A.AND.B}}$	$\overline{A + B}$ $\overline{\text{A.OR.B}}$
0	0	1	1	0	0	1	1	1	1
0	1	1	0	0	1	0	1	1	0
1	0	0	1	0	1	0	1	1	0
1	1	0	0	1	1	0	0	0	0

Notice that the following are true.

$$\overline{A} \bullet \overline{B} = \overline{A + B}$$

$$\overline{A} + \overline{B} = \overline{A \bullet B}$$

4.2. Multibit Logic Examples

Table 12 Logic Examples

Expression	Binary	Decimal	Octal	Hexidecimal
A	0000000000000001	1	1	1
B	0000000000000001	1	1	1
A.AND.B	0000000000000001	1	1	1
A.OR.B	0000000000000001	1	1	1
.NOT.A	1111111111111110	65534	177776	FFFE
.NOT.B	1111111111111110	65534	177776	FFFE

Expression	Binary	Decimal	Octal	Hexidecimal
A	1010101111001101	43981	125715	ABCD
B	0000000000000000	0	0	0
A.AND.B	0000000000000000	0	0	0
A.OR.B	1010101111001101	43981	125715	ABCD
.NOT.A	0101010000110010	21554	52062	5432
.NOT.B	1111111111111111	65535	177777	FFFF

Expression	Binary	Decimal	Octal	Hexidecimal
A	1010101111001101	43981	125715	ABCD
B	1111111111111111	65535	177777	FFFF
A.AND.B	1010101111001101	43981	125715	ABCD
A.OR.B	1111111111111111	65535	177777	FFFF
.NOT.A	0101010000110010	21554	52062	5432
.NOT.B	0000000000000000	0	0	0

Expression	Binary	Decimal	Octal	Hexidecimal
A	1010101111001101	43981	125715	ABCD
B	0000111111110000	4080	7760	FF0
A.AND.B	0000101111000000	3008	5700	BC0
A.OR.B	1010111111111101	45053	127775	AFFD
.NOT.A	0101010000110010	21554	52062	5432
.NOT.B	1111000000001111	61455	170017	F00F

Expression	Binary	Decimal	Octal	Hexidecimal
A	0000000011111111	255	377	FF
B	1000100110011000	35224	104630	8998
A.AND.B	0000000010011000	152	230	98
A.OR.B	1000100111111111	35327	104777	89FF
.NOT.A	1111111100000000	65280	177400	FF00
.NOT.B	0111011001100111	30311	73147	7667

Expression	Binary	Decimal	Octal	Hexidecimal
A	0000000011111111	255	377	FF
B	0000000000010011	19	23	13
A.AND.B	0000000000010011	19	23	13
A.OR.B	0000000011111111	255	377	FF
.NOT.A	1111111100000000	65280	177400	FF00
.NOT.B	1111111111101100	65516	177754	FFEC

Expression	Binary	Decimal	Octal	Hexidecimal
A	0000000011111111	255	377	FF
B	0011001100110011	13107	31463	3333
A.AND.B	0000000000110011	51	63	33
A.OR.B	0011001111111111	13311	31777	33FF
.NOT.A	1111111100000000	65280	177400	FF00
.NOT.B	1100110011001100	52428	146314	CCCC

Expression	Binary	Decimal	Octal	Hexidecimal
A	1111111100000000	65280	177400	FF00
B	1000100110011000	35224	104630	8998
A.AND.B	1000100100000000	35072	104400	8900
A.OR.B	1111111110011000	65432	177630	FF98
.NOT.A	0000000011111111	255	377	FF
.NOT.B	0111011001100111	30311	73147	7667

Expression	Binary	Decimal	Octal	Hexidecimal
A	1111111100000000	65280	177400	FF00
B	0000000000010011	19	23	13
A.AND.B	0000000000000000	0	0	0
A.OR.B	1111111100010011	65299	177423	FF13
.NOT.A	0000000011111111	255	377	FF
.NOT.B	1111111111101100	65516	177754	FFEC

Expression	Binary	Decimal	Octal	Hexidecimal
A	1111111100000000	65280	177400	FF00
B	0011001100110011	13107	31463	3333
A.AND.B	0011001100000000	13056	31400	3300
A.OR.B	1111111100110011	65331	177463	FF33
.NOT.A	0000000011111111	255	377	FF
.NOT.B	1100110011001100	52428	146314	CCCC

5. Gates and Latches

Error! Reference source not found. **and Table 13 Generic Gate, Switch, Latch - Definitions**

define three generic devices, which may be either analog or digital devices. The devices are three port devices with two inputs, e.g. $\mathbf{e_{in}}$ and a control signal $\mathbf{e_{GC}}$, $\mathbf{e_{SC}}$, or $\mathbf{e_{LC}}$, and one output, $\mathbf{e_{out}}$. The devices have two states. The control signal determines in which of the two states the device is at a particular time.

GateSwitchLatch_01.cdr 7-Oct-2004

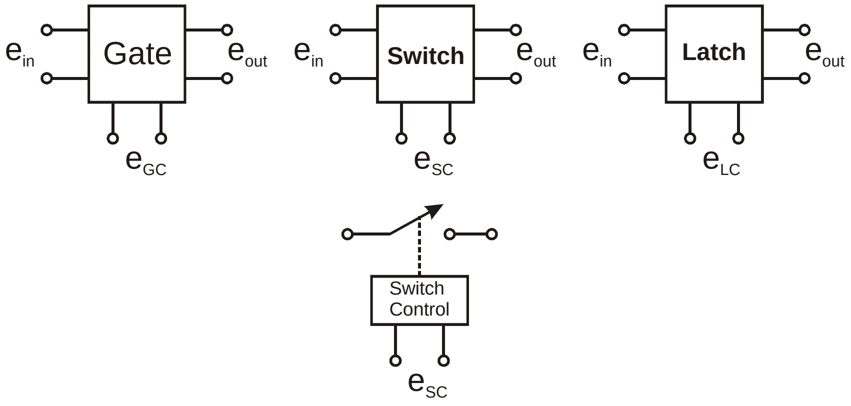


Figure 4 Generic Gate, Switch, and Latch

The gate nomenclature comes from the barnyard gate, i.e. when the gate is open, the animals can go through the gate; when the gate is closed then animals can not go through the gate. The latch is basically a camera, i.e. it captures a snapshot of the value of $\mathbf{e_{in}}$ at the time of the transition of $\mathbf{e_{LC}}$ and holds it for later inspection.

Table 13 Generic Gate, Switch, Latch - Definitions

Device	State	Control Signal	Behavior
Gate	Open	**e_{GC} = Open**	**e_{out}** = **e_{in}**
	Closed	**e_{GC} = Closed**	**e_{out}** = constant (also may be disconnected)
Switch	Closed	**e_{SC} = Closed**	**e_{out}** = **e_{in}**
	Open	**e_{SC} = Open**	**e_{out}** = constant
Latch	Follow	**e_{LC} = Follow**	**e_{out}** = **e_{in}**
	Latched	**e_{LC} = Latch**	**e_{out}** = **e_{in}** (t = Follow ↑ Latch)

Error! Reference source not found. illustrates a derivative combination, the tri-state gate which has the characteristics shown in Table 14 Tri-State Gate - Definition Definition

. This device derives its name from the fact that there are essentially three states: high, low, and disconnected. Such devices have great utility when constructing a “bus,” i.e. a “party line” or shared communication facility.

TriStateGate_01.cdr 10-Oct-2004

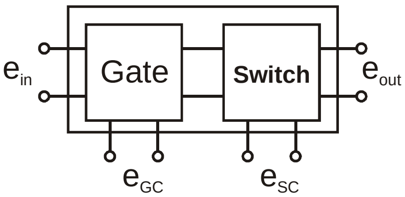


Figure 5 Tri-State Gate

Table 14 Tri-State Gate - Definition Definition

Switch Control	Gate Control	Behavior
e_{SC} = Closed	**e_{GC} = Open**	**e_{out}** = **e_{in}**
e_{SC} = Closed	**e_{GC} = Closed**	**e_{out}** = constant
e_{SC} = Open	**e_{GC} = Open**	Device is disconnected from the following circuitry.
e_{SC} = Open	**e_{GC} = Closed**	Device is disconnected from the following circuitry.

These generic concepts have widespread application in both digital and analog electronics. The remainder of this document will explore how these devices are implemented and applied in the digital domain.

6. Simple Computer

A common model of a simple computer is the "Von Neumann" model shown in Figure 6. This model consists of three types of functional units, Central Processing Unit (CPU), Memory, and I/O units of which there can be varying numbers in any real application. The CPU contains four subsystems, the Command Decoder, CD, arithmetic logical unit, ALU, Control Panel, and the CPU register set. The CPU is the engine that does the computational work of the system. The Command Decoder fetches, interprets, and causes the execution of the program instruction steps. The ALU performs, under the control of the CPU, the integer arithmetic and logical operations required by the program instructions. The I/O units provide the interface between the computing system and the outside world. The Control Panel allows the operator of the system to perform certain basic operations such as starting and stopping operation, and examining and changing aspects of the system. The functional units are connected by the I/O bus, a communication facility that allows information to be moved among the various functional units.

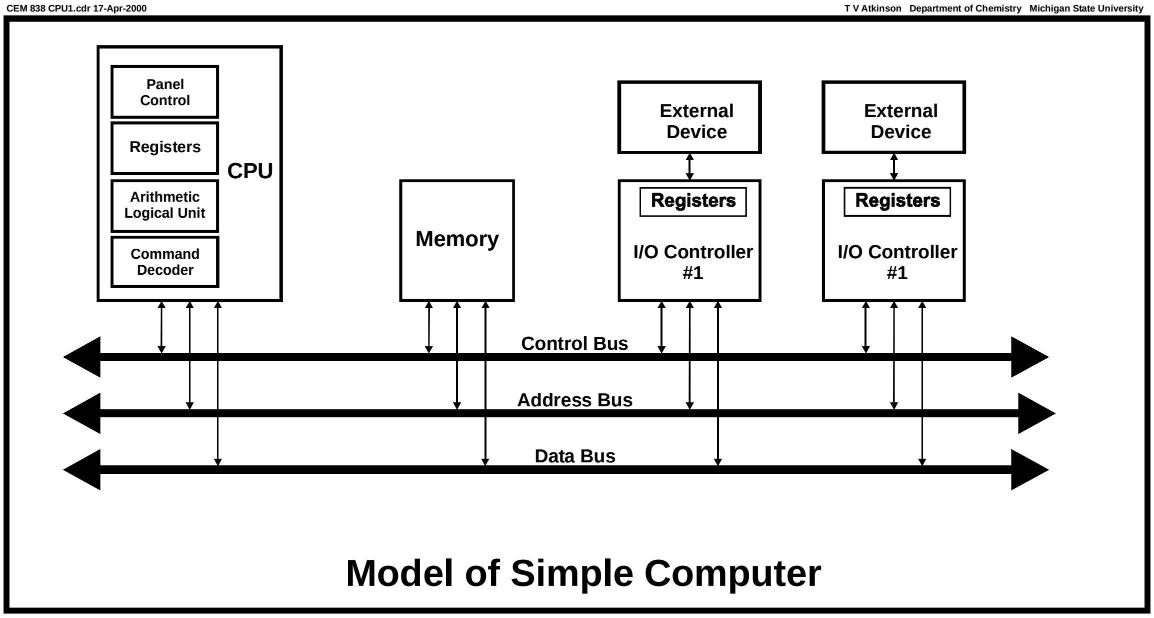


Figure 6 Von Neumann Model of Computer

6.1. Digital Buses

I/O buses are actually collections of parallel digital (binary) electrical signals that are simultaneously observed and/or manipulated by multiple functional units. Hence, a bus is an example of a "party-line" communication channel with the connected subsystems being peers on the channel. Information is typically moved between two of the participants on the bus. The I/O bus is usually considered to consist of three sub buses (Control, Address, and Data) (See Figure 6). The Data Bus is a collection of signals that contain the data being moved from one subsystem to the other. The Address Bus allows the participants on the bus to identify which subsystem is sending the information and which subsystem is receiving the information. The Control Bus is the collection of signals required to affect the transfer of information from one participant to the other.

The states of the control signals, i. e. the Control Bus, are defined one of the subsystems called the Bus Master. In simple computers, only the CPU can be master. In more complicated architectures, other functional units can be bus master. There have been many computer buses, e.g. Unibus, Qbus, New Bus, VMEbus, XT bus, ISA (ATBUS), SCSI, EISA, Micro Channel Architecture (MCA) bus, VESA, PCI, IEEE 488. One bus varies from another in the following ways.

1. Collection of signals (number and definition)
2. Technology used to implement electronics connected to the bus, e.g. TTL, CMOS, ECL, Optical elements (for optical fiber buses).
3. Physical implementation: connectors, conductors, etc.
4. Speed and timing relationships
5. Sequences of events required to effect transfers of information.

6.1.1. A Simple Example

The simple logic devices, i.e. And, Or, Nor, Nand, gates and latches, discussed in the section on logic are the atomic elements of digital devices. Real digital devices, e.g. computers, are, in essence, collections of such elements. As mentioned above, these collections are typically organized into subsystems. One issue is how information is moved from one subsystem of the device to another.

Figure 8 and Figure 9 illustrates one such mechanism, a simple digital bus that connects three single bit devices (registers) **A**, **B**, and **C**. In this example, $\mathbf{GC_a}$, $\mathbf{LC_a}$, $\mathbf{GC_b}$, $\mathbf{LC_b}$, $\mathbf{GC_c}$, and $\mathbf{LC_c}$ are control signals. $\mathbf{L_a}$, $\mathbf{L_b}$, and $\mathbf{L_c}$ are the contents of the registers. As an example, the following steps are performed to move the contents of Device **A** ($\mathbf{L_a}$) to Device **C** ($\mathbf{L_c}$).

1. All control signals are in the “off” state, i.e. $\mathbf{LC_i}$ are in the LATCHED state and $\mathbf{GC_i}$ are in the CLOSED state. The bus is idle.
2. Assert $\mathbf{GC_a}$ ($\mathbf{GC_a}$ = OPEN). BUS is now equal to $\mathbf{L_a}$
3. Strobe $\mathbf{LC_c}$ as in Figure 7. $\mathbf{L_c}$ now equals $\mathbf{L_a}$. Transfer is complete.

4. Deassert $\mathbf{GC_a}$ (Gate A is CLOSED.) Bus is now idle.

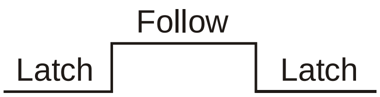


Figure 7 Strobe

Figure 10 shows the timing of events that would be required to move the contents of Register **A** into Register **C**. The contents of **A** is assumed to be 1 at the beginning, the contents of **B** and **C** are 0. The timing sequences for the control signals $\mathbf{GC_a}$, $\mathbf{LC_a}$, $\mathbf{GC_c}$, and $\mathbf{LC_c}$ are generated by some outside intelligence (often called the Bus Master). Figure 11 is a similar example with the contents of **A** being 0.

BusA1.cdr 23-Oct-2002 T V Atkinson Department of Chemistry Michigan State University

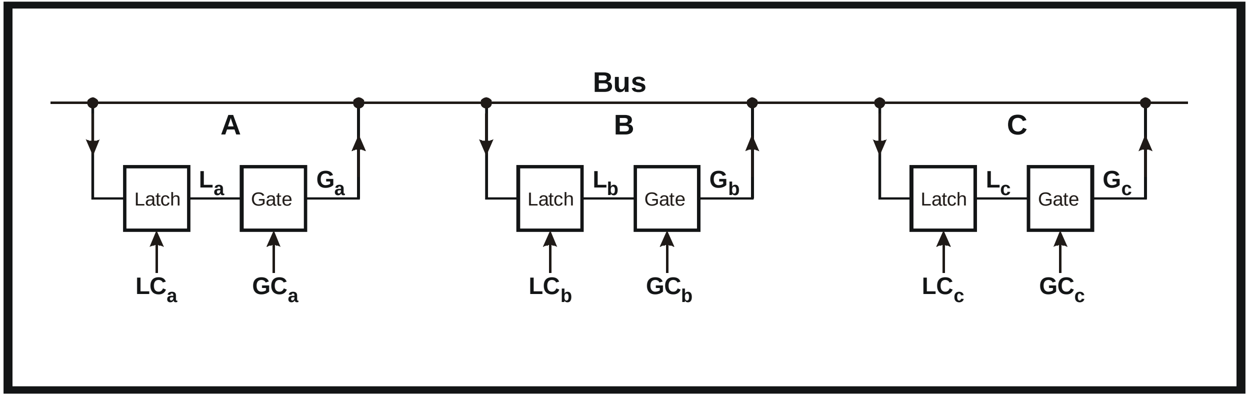


Figure 8 A Digital Bus with Three Devices

BusA1.cdr 21-Oct-2002 T V Atkinson Department of Chemistry Michigan State University

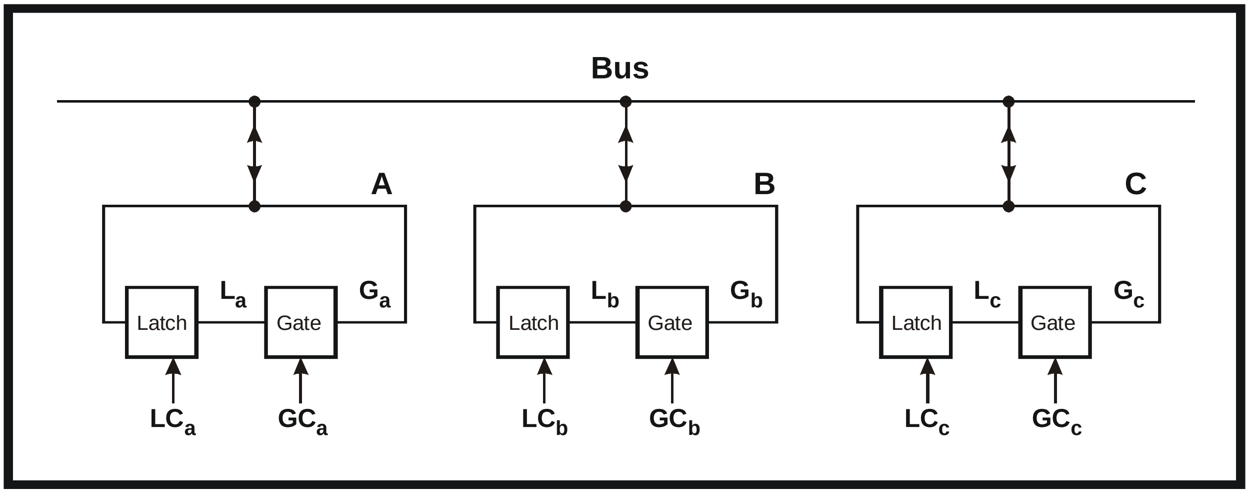


Figure 9 A 1-Bit Bus with Three Devices (Equivalent Schemat)

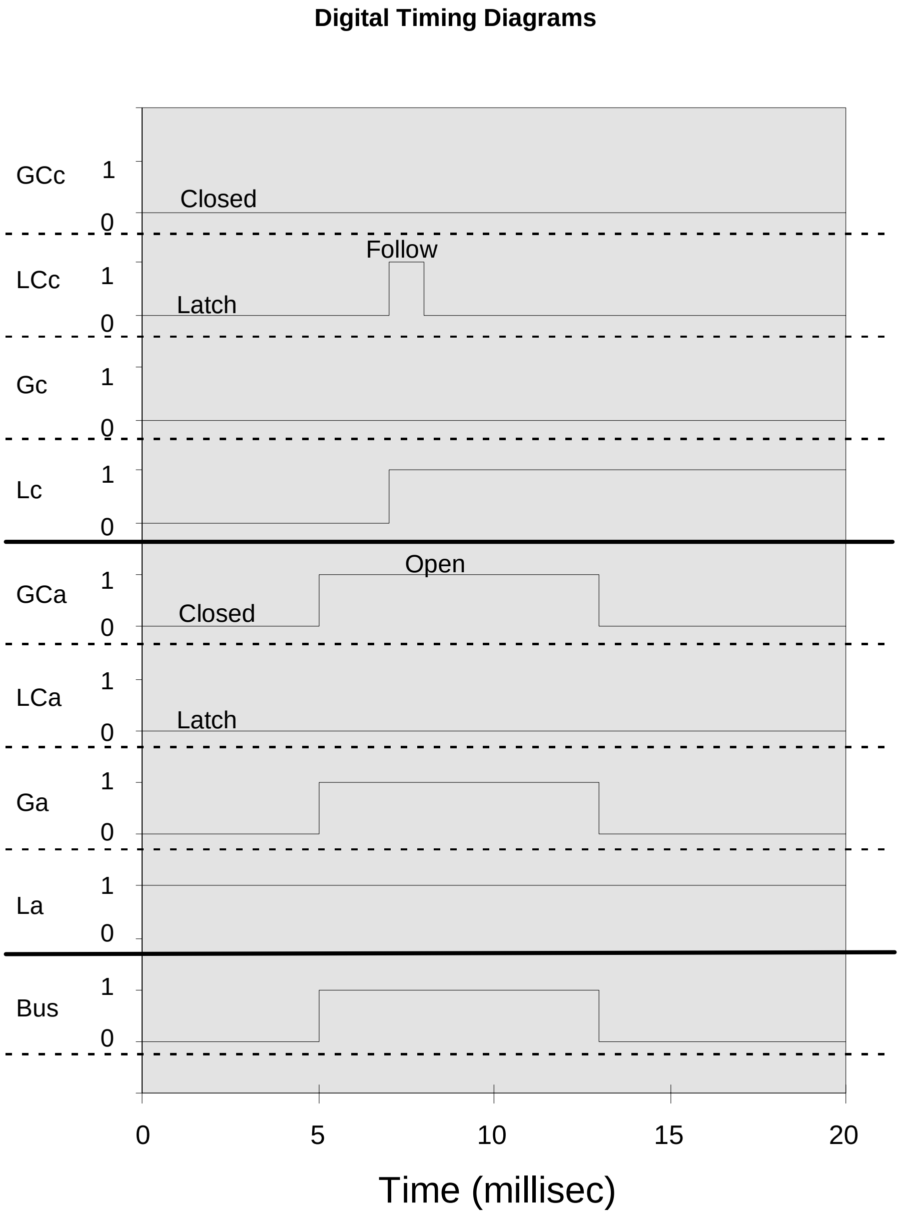


Figure 10 Timing - Transfer Contents of A (1) to C

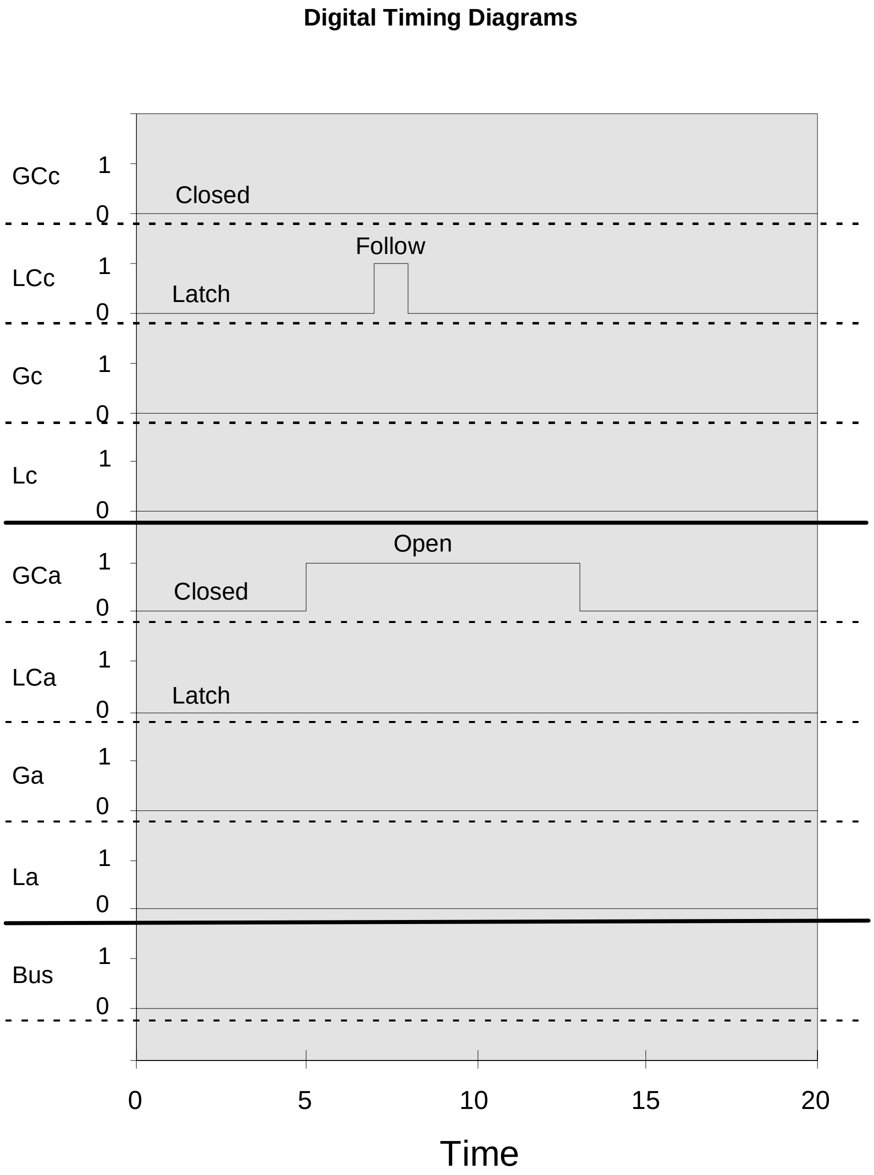


Figure 11 Timing - Transfer Contents of A (0) to C

6.1.2. A 4-Bit Bus

Figure 12 illustrates how three 4-bit devices would be connected with the simple bus discussed above. Notice that the control signals are the same for all bits of a device. Thus all four bits of information are eached moved as described above at the same time.

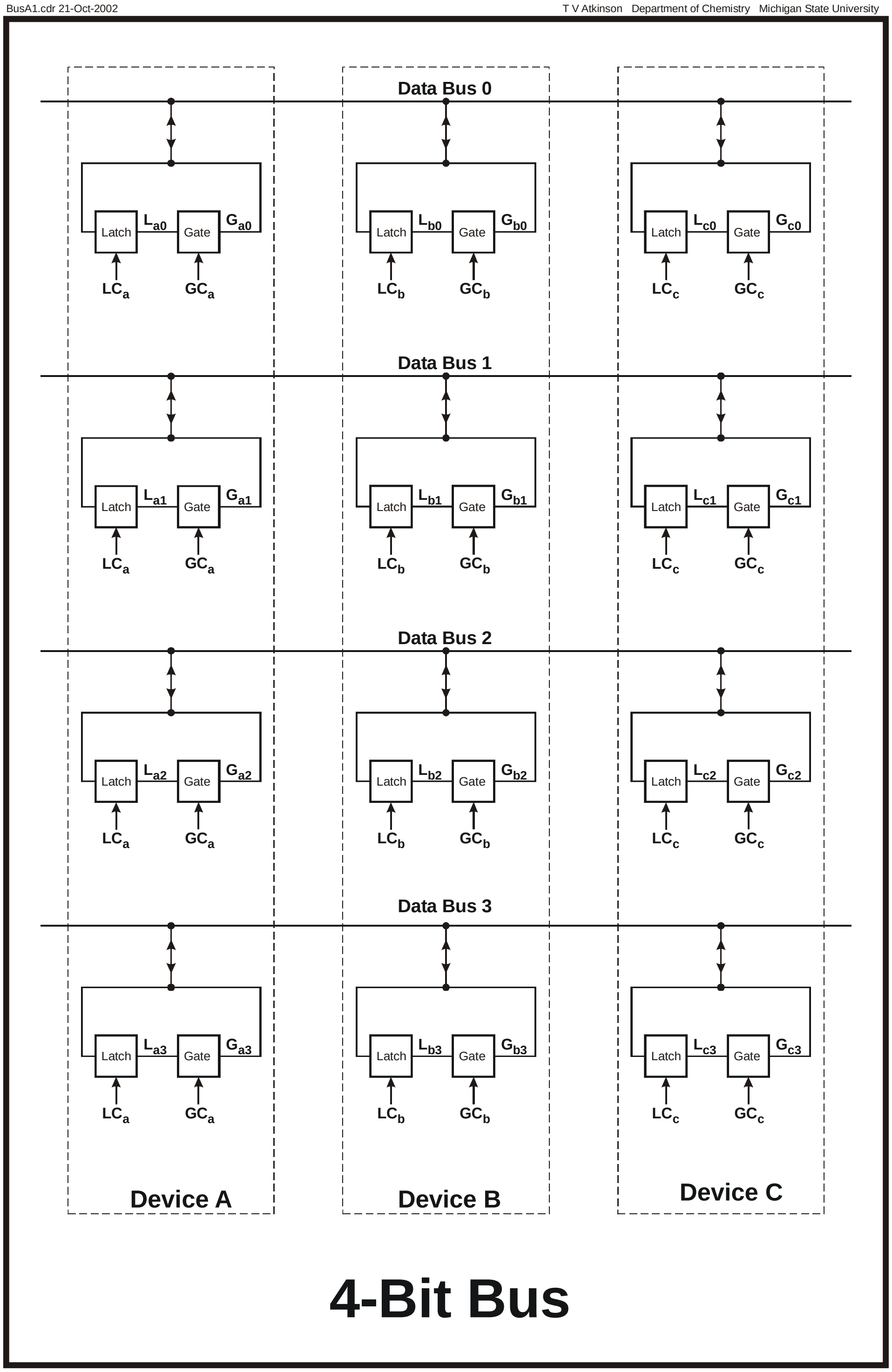

BusA1.cdr 21-Oct-2002
T V Atkinson Department of Chemistry Michigan State University
Data Bus 0
Data Bus 1
Data Bus 2
Data Bus 3
Latch
Gate
L_{a0} G_{a0} L_{b0} G_{b0} L_{c0} G_{c0}
L_{a1} G_{a1} L_{b1} G_{b1} L_{c1} G_{c1}
L_{a2} G_{a2} L_{b2} G_{b2} L_{c2} G_{c2}
L_{a3} G_{a3} L_{b3} G_{b3} L_{c3} G_{c3}
LC_a GC_a LC_b GC_b LC_c GC_c
Device A
Device B
Device C
4-Bit Bus

Figure 12 4-Bit Bus

6.1.3. An 8-Bit Bus System

Figure 13 and Figure 14 illustrate the next level of complexity. Here, there are two types of devices, i. e. the Master Register and any number of Slave Registers. All registers are 8-bit devices. In this system, the Master Register is involved in all transfers. Reading a register is defined as a transfer that copies information from that register to another. Writing a register is defined as a transfer that copies information from another to that register. The Address Bus is a set of signals that identify which Slave Register is involved in the transfer. Decoder is a function that monitors the Address Bus and goes true when the address of that Slave Register is on the Address Bus. The Control Bus, i.e. **STROBE** and **WRITE** are signals generated by the Bus Controller which is described here.

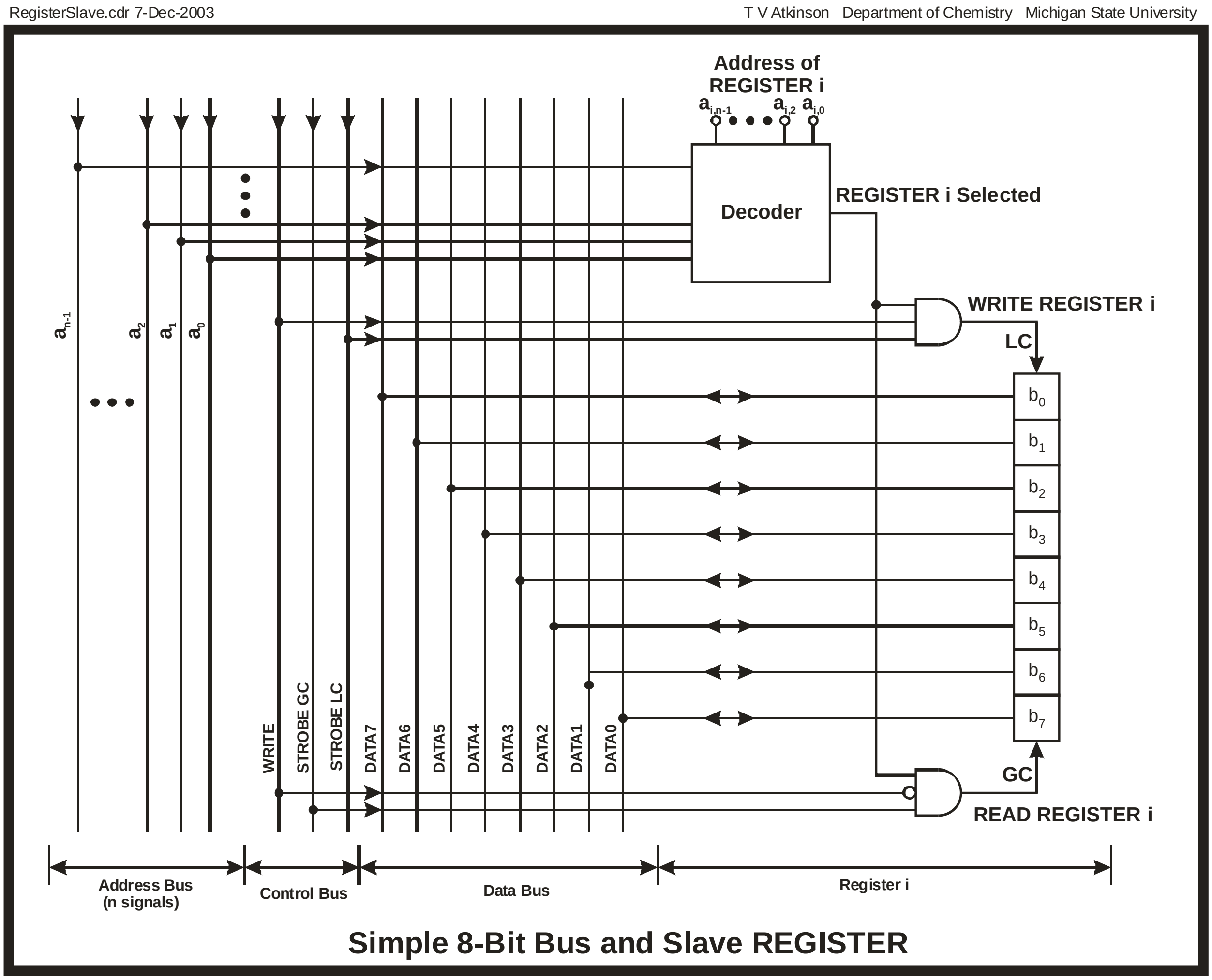


Figure 13 - Simple 8-Bit Bus and Slave Register

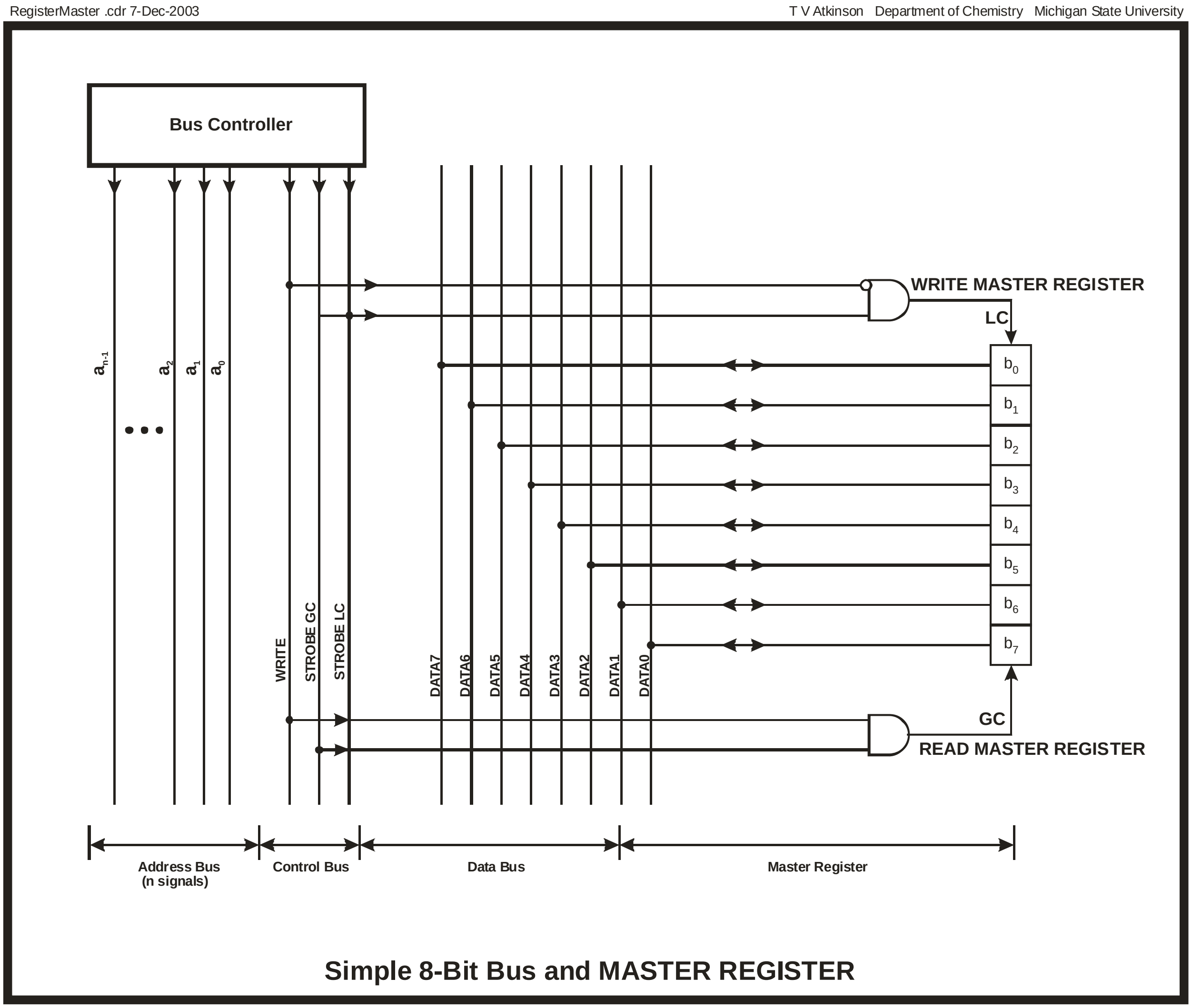


Figure 14 - Simple 8-Bit Bus and Master Register

Transfers of information from Slave Register *i* to the Master Register are accomplished with the following sets of steps.

1. The address of Slave Register$_i$ is placed on the Address Bus. The output of Decoder$_i$ goes true.
2. **WRITE** is set low.
3. **STROBE LC** and **STROBE GC** are strobed as indicated in Figure 15. This gates the contents of Slave Register$_i$ onto the Data Bus. The contents of the Data Bus are latched into Master Register.

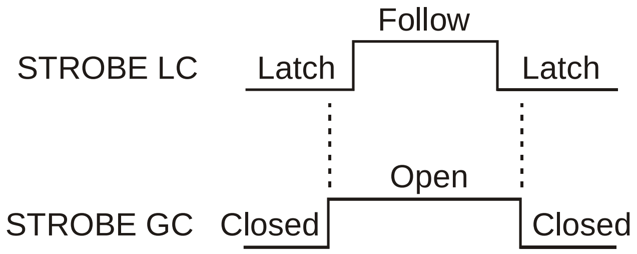


Figure 15 - Simple 8-Bit Bus and Master Register

6.1.4. A Simple Input/Output System

Figure 16 illustrates how one could implement a one bit register that outputs information from the digital device and a one bit register that would input information to the device from the outside world.

BusIO .cdr 21-Oct-2002 T V Atkinson Department of Chemistry Michigan State University

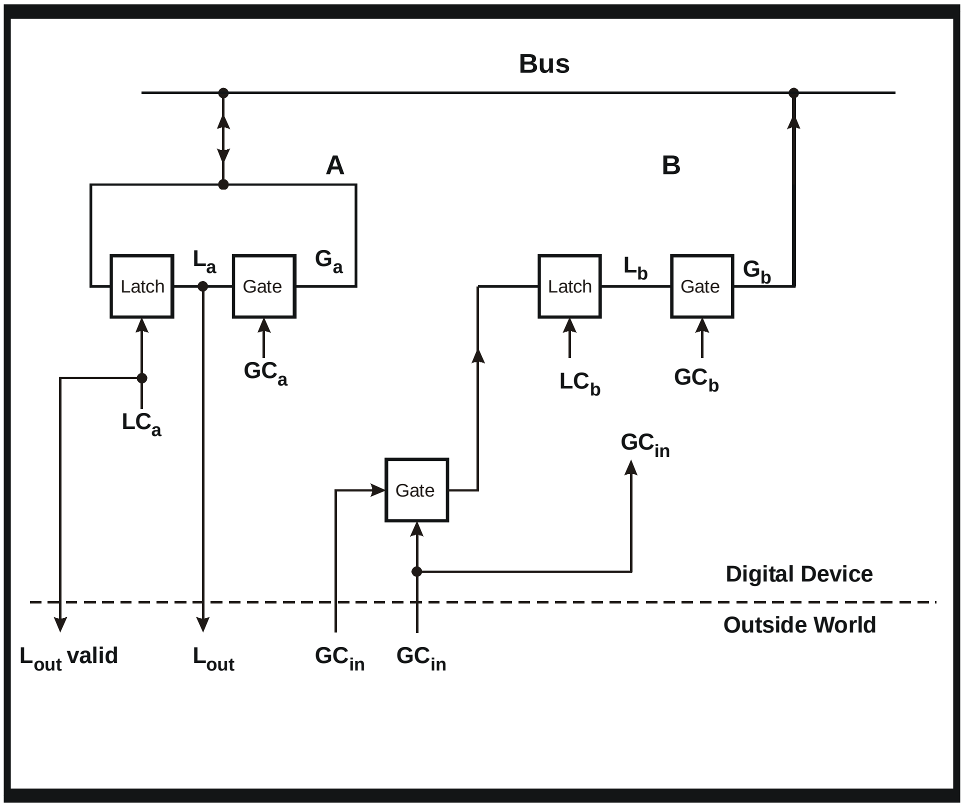


Figure 16 Simple 8-Bit Bus and Master Register

6.1.5. A More Complete I/O Bus Architecture

Figure 17, Figure 18, and Figure 19 depict a fairly simple but more complete bus architecture that illustrates a number of points. This particular architecture was constructed for illustration and does not match any particular computer system.

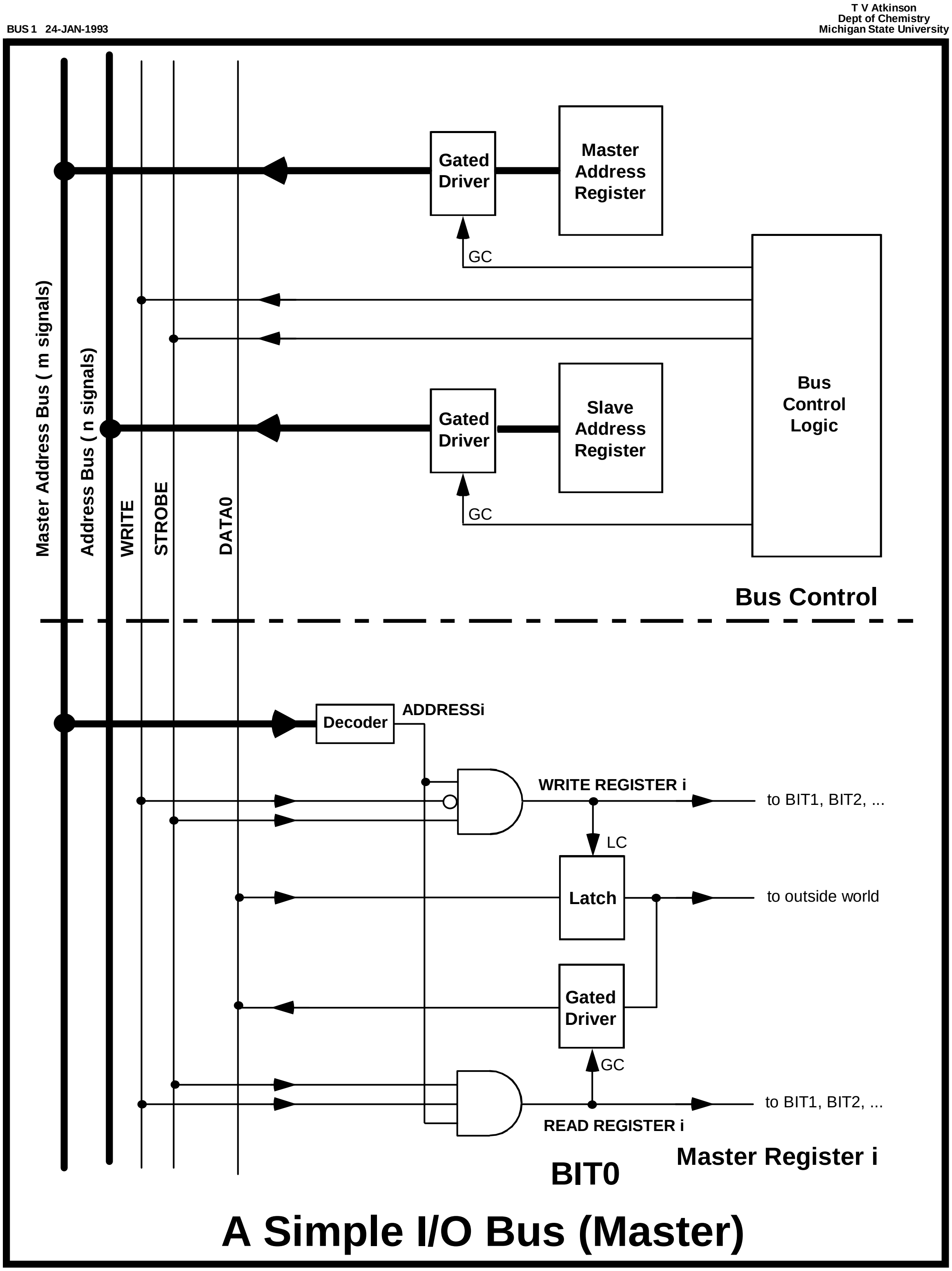


Figure 17 Simple I/O Bus: Bus Master

Figure 18 illustrates a slave device on the bus.

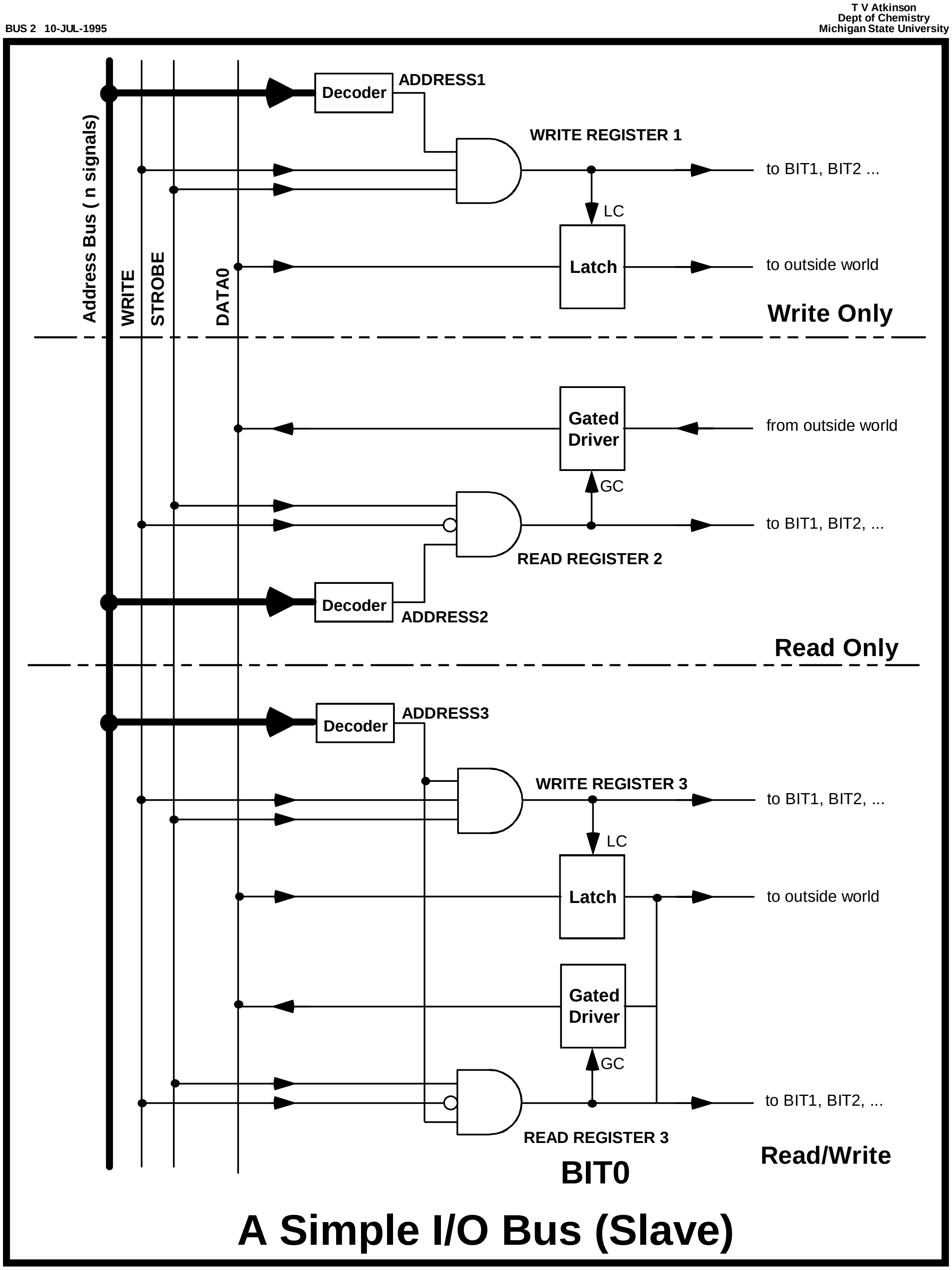


Figure 18 Simple I/O Bus: Slave

Figure 19 shows the remaining bits of the representative registers forming the slave devices.

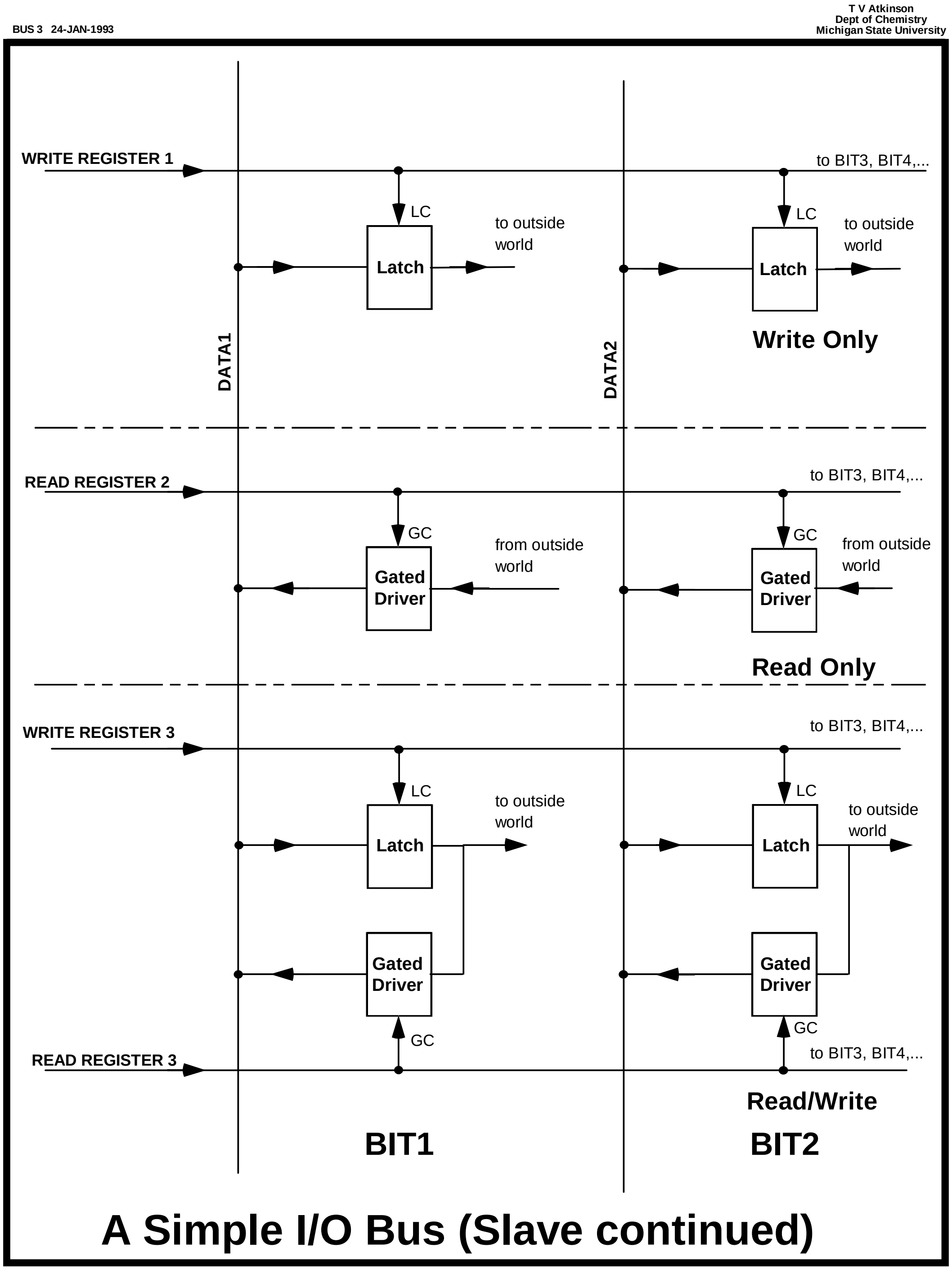


Figure 19 Simple I/O Bus: Slave (Continued)

6.1.5.1. Reads

For this simple architecture, a "read" is the transfer of information from the slave register to the master register.

1. The idle state of the bus consists of **STROBE** being 0. As a result, all **WRITE REGISTER j** and **READ REGISTER j** AND gates will have an output of 0. The states of all other signals are of no consequence.

2. Bus Master Bus Control Logic gates the address of the referenced master register onto the Master Address Bus. The output of the decoder for the appropriate Master Register will now change to 1, indicating selection of that register.

3. Bus Master Bus Control Logic gates the address of the referenced slave register onto the Address Bus. The output of the decoder for the appropriate slave register will now change to 1, indicating selection of that register.

4. Bus Master Bus Control Logic gates a 0, i.e. do a read, onto the WRITE line. This will set up the gating of the selected slave register onto the Data bus and the latches of the selected master register to follow the state of the Data bus.

5. A time delay occurs that allows all the above signals to settle and the various decoding to take place.

6. The Bus Master Bus Control Logic gates a 1 onto the signal **STROBE**. With **STROBE** now one, all inputs will be one for the **WRITE REGISTER i** AND gate of the addressed master register and the **READ REGISTER j** AND gate for the addressed slave register. Thus, these two signals will go to 1. All other AND gates will have at least one 0, resulting in outputs of those gates remaining at 0. Thus, the gate control signal (**GC**) for the gated driver for each of the bits of the addressed slave register will go to 1 and these signals will be gated onto the Data bus. Simultaneously, the latch control signal (**LC**) for the latches for each of the bits of the addressed master register will go to 1 and these latches will begin to follow the corresponding bits on the Data bus.

7. A time delay occurs that allows all the above signals to settle.

8. The Bus Master Bus Control Logic gates a 0 onto the signal **STROBE**. As a result, all **WRITE REGISTER j** and **READ REGISTER j** AND gates have an output of 0. Thus, the latches for the selected master register change to the latched state, freezing the contents of the selected slave register into the master register. The signals from the slave register are also removed from the Data bus. The bus is now in the idle state.

6.1.5.2. Writes

For this simple architecture, a "write" is the transfer of information from the master register to the slave register.

1. The idle state of the I/O bus consists of **STROBE** being 0. As a result, all **WRITE REGISTER j** and **READ REGISTER j** AND gates will have an output of 0. The states of all other signals are of no consequence.

2. Bus Master Bus Control Logic gates the address of the referenced master register onto the Master Address Bus. The output of the decoder for the appropriate master register will now change to 1, indicating selection of that register.

3. Bus Master Bus Control Logic gates the address of the referenced slave register onto the Address Bus. The output of the decoder for the appropriate slave register will now change to 1, indicating selection of that register.

4. Bus Master Bus Control Logic gates a 1, i.e., do a write, onto the **WRITE** line. This will set up the gating of the selected master register onto the Data bus and the latches of the selected slave register to follow the state of the Data bus.

5. A time delay occurs that allows all the above signals to settle and the various decoding to take place.

6. The Bus Master Bus Control Logic gates a 1 onto the signal **STROBE**. With **STROBE** now one, all inputs will be one for the **WRITE REGISTER i** AND gate of the addressed slave register and the **READ REGISTER i** AND gate for the addressed master register. Thus, these two signals will go to 1. All other AND gates will have at least one 0, resulting in outputs of those gates remaining at 0. Thus, the gate control signal (**GC**) for the gated driver for each of the bits of the addressed master register will go to 1 and the master register contents will be gated on to the Data bus. Simultaneously, the latch control signal (**LC**) for the latches for each of the bits of the addressed slave register will go to 1 and these latches will begin to follow the corresponding bits on the Data bus.

7. A time delay occurs that allows all the above signals to settle.

8. The Bus Master Bus Control Logic gates a 0 onto the signal **STROBE**. As a result, all **WRITE REGISTER j** and **READ REGISTER j** AND gates have an output of 0. Thus, the latches for the selected slave register change to the latched state, freezing the contents of the selected master register into the slave register. The signals from the master register are also removed from the DATA bus. The bus is now in the idle state.

6.2. Post Office (Programmers) Model of Computing

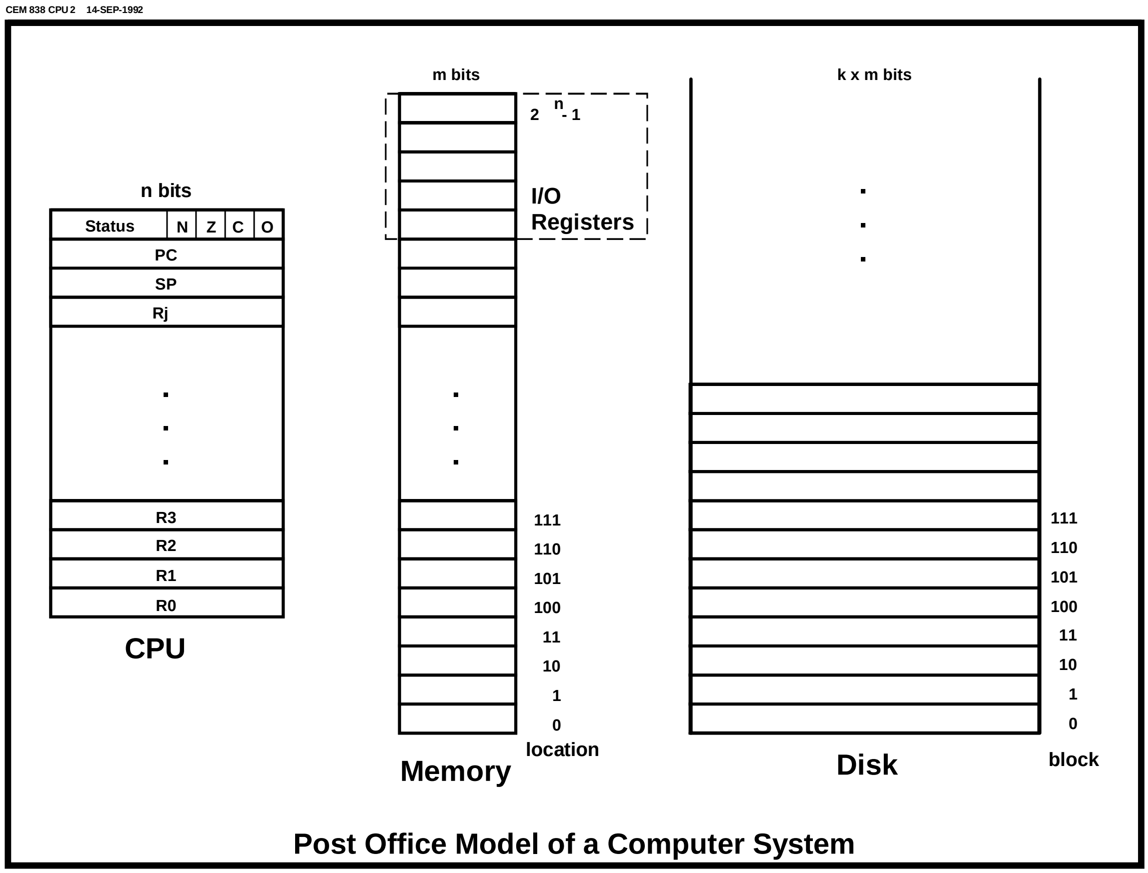


Figure 20 Post Office Model of Computing

The simple computer can also be modeled as three ranks of different size pigeon holes as illustrated in Figure 20. As in a post office, each of the boxes is identified by a unique label. For those boxes in the CPU, these labels are PC, R0, R1, … In the case of Memory and Disk, the identifiers are binary numbers. Each box depicted in that figure is capable of containing a single ordered collection of binary bits each of which can have two states (1 or 0). For a collection of n bits, there can be 2^n unique combinations of zeros and ones. Many different uses can be made of such collections of binary bits. In fact, most of the boxes described here can have different meanings at different times. Only a few locations, such as the status register and PC, have specific meanings impressed on the collection of bits at all times by the hardware.

6.3. Uses of collections of n binary bits

1. Logical: Each bit can represent a logical variable and the contents of the bit will represent a true or false.

2. Flags: Each bit will represent the state of some device or functional unit. Examples: Flags in the CPU Status Register indicate if the last operand is zero, if the last operation resulted in an overflow, etc. Status bits in device registers indicate the state of devices, e.g. is a valve open or closed, is a floppy disk mounted in the drive.

3. Control bits in device registers: Bits are connected to hardware devices and cause something to happen in the device, e.g. begin the conversion of a ADC, open/close a valve, fire a laser.

4. Character: Codes representing symbols such as the ASCII character sets.

5. Unsigned binary integer numbers - if each bit is considered to be a coefficient of a power of two and the collection of bits is considered to be ordered (see the section on number systems). The 2^n numbers will range from 0 to 2^n-1.

6. Unsigned binary fractions - if each bit is considered to be a coefficient of a negative power of two and the collection of bits is considered to be ordered (see the section on number systems). The 2^n numbers will range from 0 to 1 - 2^{-n}.

7. Addresses

8. Sign Magnitude signed binary integer numbers

9. One's complement signed binary integer numbers

10. Two's complement signed binary integer numbers

11. Floating Point numbers

[SEE: the figure titled 8087 Numeric Data Processor]

12. Instructions

6.4. Instruction Sets

A computer is a machine that performs a sequential set of recipes or instructions on one, two or three operands. The instructions describe exactly what is to be done for each step. The operands are the collections of bits located in the CPU registers, memory locations, and/or device registers. Instruction formats and sizes vary from machine to machine. In many machines different instructions can be of different sizes, usually in multiples of bytes. The instructions will contain two main parts. The first is a code identifying the particular instruction. The second is information about which locations in the computer contain the operands upon which the operations will be executed.

1. Moves (2 Operands)
   - 1.1. Register to Register
   - 1.2. Memory to Register (Load)
   - 1.3. Register to Memory (Store)
   - 1.4. Memory to Memory
   - 1.5. Register or Memory to a stack (PUSH)
   - 1.6. Stack to Register or Memory (POP)
   - 1.7. Clear operand
2. Logical operations (2 or 3 operands)

Operation	Result		Operand 2	Operator	Operand 1
Inverse	O_2	<---		.not.	O_1
AND	O_3	<---	O_2	.and.	O_1
OR	O_3	<---	O_2	.or.	O_1
XOR	O_3	<---	O_2	.xor.	O_1

3. Arithmetic +, -, *, /, negate (2 or 3 Operands)
4. Shifts (1 Operand)
   - 4.1. One bit shifts (left or right)
   - 4.2. Multiple bits shifts (left or right)
   - 4.3. Single register, multiple register
   - 4.4. Simple, circular, arithmetic (pull sign bit along)
5. Test/compare
6. Branches (1 or 2 Operands)
   - 6.1. Unconditional: jumps or branches

6.2. Conditional

6.3. Subroutine call

6.4. Subroutine return

6.5. Traps

6.6. Interrupt returns

7. NOOP (0 Operands)

8. Halt, Pause, Wait (0 Operands)

9. I/O Instructions (Only in the cases where the I/O registers are not part of the memory space).

10. Other special instructions.

6.5. Addressing

Typically, instruction sets deal with three types of operands, CPU registers, memory locations, and device registers. In many architectures, the memory and device registers are incorporated into one address space. Each instruction has to specify which operand(s) are to be used in the execution of that instruction. The instructions have 0 (NOOP, halt, …), 1, 2, or 3 operands. Except in the cases with 0 operands, one operand will be the destination operand and receives the results of the operation. Often the destinations will be one of the input operands and the instructions effectively have 1 or 2 operands. In all such cases, the contents of the destination operand are changed as the result of the instruction. Only those source operands that are also destinations are changed.

6.6. Operation

Any computer does useful work by executing a program, or ordered collection of instructions.

1. Reset the computer to a known initial condition.

2. Deposit the instructions and any operands into appropriate locations within memory.

3. Deposit the starting point (entry point) of the program into the PC.

4. Press the "GO" button.

5. CPU puts the contents of the PC onto the address bus and causes the contents of the memory location with that address to be fetched to the command decoder.

6. The contents of the PC are incremented by one.

7. The command decoder interprets the instruction code.

8. Any additional bytes of the instruction are fetched. PC is incremented accordingly.

9. Any operands are fetched.

10. The operation is executed by the ALU.

11. The results of the operation are written into the destination operand (register or memory).

12. As a result of the various increments, the PC now points to the next instruction and the process loops back to step 5 and the next instruction is fetched, then executed. This process continues until the CPU is halted or a HALT instruction is executed.

6.7. An Example Computer

The goal of this example is to illustrate the basic operation of a computer and to impress on you the **simplicity** of each aspect of computing, the **beauty** (it's like a puzzle with a very large number of pieces that fit together) of the low level workings of computing, and, most importantly, the **tedium** involved in doing computing on this level. Professionally, scientists can not afford to do this level of computing any more. Typically, you buy existing hardware and software to do these jobs.

This is a description of a simple computer architecture viewed from the software perspective. This architecture is defined for simplicity not efficiency. No such computer exists, but this example can be used to illustrate a number of concepts.

6.7.1. Registers

1. PC - 32 bits wide ; Program Counter

2. SP - 32 bits wide ; Stack Pointer

3. R5 - 32 bits wide ; General Registers

4. R4 - 32 bits wide

5. R3 - 32 bits wide

6. R2 - 32 bits wide

7. R1 - 32 bits wide

8. R0 - 32 bits wide

CEM 924 BitByte 23-JAN-1993

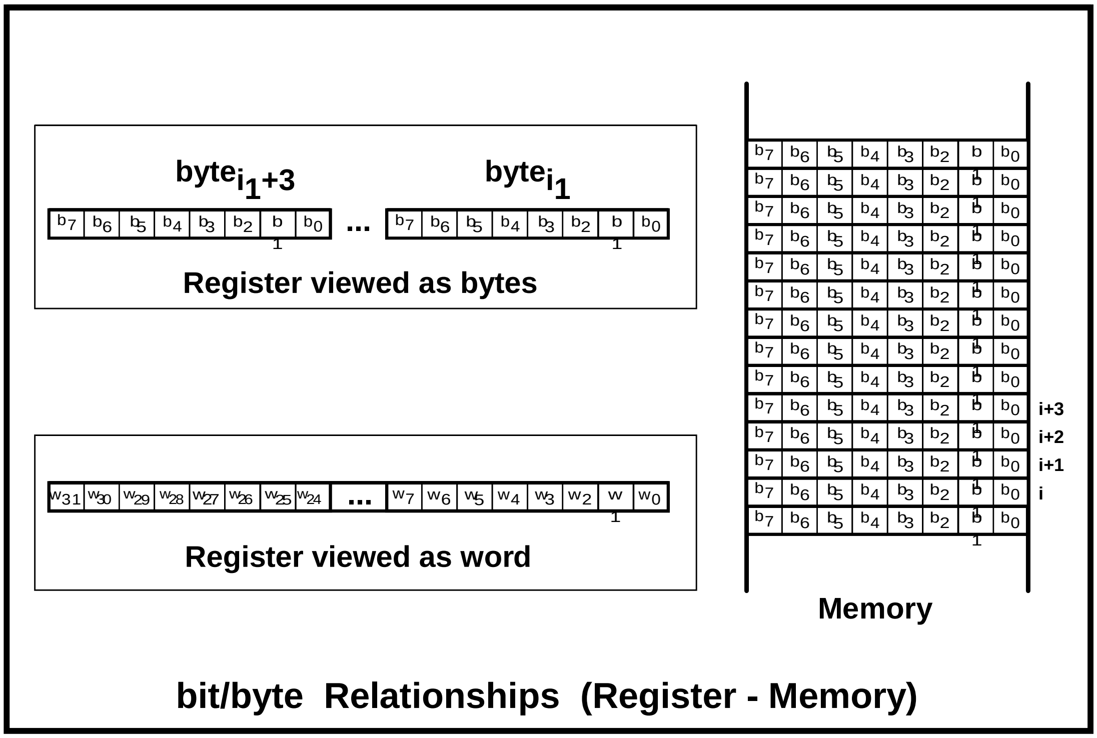

byte$_{i_1+3}$
byte$_{i_1}$
b7 b6 b5 b4 b3 b2 b1 b0 ... b7 b6 b5 b4 b3 b2 b1 b0
Register viewed as bytes
w31 w30 w29 w28 w27 w26 w25 w24 ... w7 w6 w5 w4 w3 w2 w1 w0
Register viewed as word
b7 b6 b5 b4 b3 b2 b1 b0
i+3
i+2
i+1
i
Memory
bit/byte Relationships (Register - Memory)

Figure 21 Bit Byte Word Relationship

6.7.2. Instruction format

Op code	Register(R_a)	Register(R_b)	Direct/Indirect Addr	Operand
B_{31} to B_{28}	B_{27} to B_{25}	B_{24} to B_{22}	B_{21}	B_{20} to B_0

B_{21} is a flag that indicates whether addressing for a given instruction is direct or indirect. If B_{21} of an instruction is 0, the operand field of the instruction contains the address of the operand. If B_{21} of an instruction is 1, R_b contains the address of the operand.

6.7.3. Instruction Set

Op Code	Mnem	I/D?	Description
0	HALT	N	Halt CPU operation
1	NOOP	N	Do nothing for one instruction cycle
2	LOAD	Y	Load contents of Operand (i.e. Memory Location) into Register R_a
3	STORE	Y	Store contents of Register R_a into Operand (i.e. Memory Location)
4	MOVE	N	Move the contents of register R_a to register R_b
5	AND	N	The logical **AND** of the contents of registers R_a and R_b is calculated and stored in Register R_a
6	OR	N	The logical **OR** of the contents of registers R_a and R_b is calculated and stored in Register R_a
7	INV	N	Invert the contents of Register R_a
8	NEG	N	Negate (take two's complement of) the contents of Register R_a
9	BNE	N	Branch to the address of the operand if Register R_a is not equal to zero
A	BEQ	Y	Branch to the address of the operand if Register R_a is equal to zero
B	JMP	Y	Branch to the address contained in the Operand
C	RET	Y	Branch to location after last CALL instruction
D	CALL	N	Branch to the address contained in the Operand and store next address for return
E	ADD	N	Contents of Register R_a is added to the contents of R_b and the result is stored in Register R_a
F	MUL	N	Contents of Register R_a is multiplied by the contents of R_b and the result is stored in Register R_a

6.7.4. An Example Program

This section will investigate a simple program using the example architecture. This program will do the simple calculation shown below and produce a resultant integer A given integers B and C.

$$A = (1+B)*37_{10} - C$$

The program would be developed and executed using the following steps:

1. Create the symbolic (assembly language) version of the program.
2. Assign memory locations to all the variables and constants (A, B, C, 1, 37_{10}).
3. Assign a memory location for the first word of the program.
4. Translate the assembly language into machine language (binary) form that the computer will actually execute.
5. Deposit the machine language program and constants into memory according to the memory assignments made.
6. Deposits values for B and C into the appropriate memory locations.
7. Deposit the number 1000_{16} into the PC.
8. "GO". Program executes, halts with the PC containing 1028_{16}.
9. Result is now in location 2010_{16}.

Location	Contents (word)	Label	Op Code	I/D	R_a	R_b	Operand	Comments
00001000	22 00 20 00	START:	LOAD	D	R1		ONE	;Load 1 into R1
00001004	24 00 20 08		LOAD	D	R2		B	;Load B into R2
00001008	E2 80 00 00		ADD	D	R1	R2		;R1 = (1+B)
0000100C	24 00 20 04		LOAD	D	R2		THSEV	;Load 37 into R2
00001010	F2 80 00 00		MUL		R1	R2		;R1 = (1+B)*37
00001014	24 00 20 0C		LOAD	D	R2		C	;Load C into R2
00001018	84 00 00 00		NEG		R2			;R2 = -C
0000101C	E2 80 00 00		ADD		R1	R2		;R1 = (1+B)*37 - C
00001020	32 00 20 10		STORE	D	R1		A	;Save Results
00001024	00 00 00 00	END:	HALT					;End of the Program, stop

Example translations from assembly language to machine language (binary).

LOAD D R1 ONE

Instr.	Op Code				R_a			R_b			I/D	Operand																				
Bit Number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Contents	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Hex	2				2				0				0				2				0				0				0			

LOAD D R2 B

Instr.	Op Code				R_a			R_b			I/D	Operand																				
Bit Number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Contents	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0
Hex	2				4				0				0				2				0				0				8			

ADD D R1 R2

Instr.	Op Code				R_a			R_b			I/D	Operand																				
Bit Number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Contents	1	1	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Hex	E				2				8				0				0				0				0				0			

View of Memory before Execution of Program

Location	Contents	Logical Name
00002018		
00002014		
00002010		A
0000200C		C
00002008		B
00002004	00 00 00 25	THSEV (37_{10})
00002000	00 00 00 01	One
00001FFC		
0000102C		
00001028		
00001024	00 00 00 00	END
00001020	34 00 20 10	
0000101C	E2 80 00 00	
00001018	86 00 00 00	
00001014	26 00 20 0C	
00001010	F2 80 00 00	
0000100C	26 00 20 04	
00001008	E2 80 00 00	
00001004	26 00 20 08	
00001000	24 00 20 00	START
00000FFC		

6.7.5. Example Program 2

This section will show how subprograms can be used to repeat multiple occurrences of the same procedure applied to different data. The formula from Example Program 1 is to be applied to three sets of data (B1, C1), (B2, C2), and (B3, C3) yielding three results A1, A2, and A3.

Location	Contents (word)	Label	Op Code	I/D	R_a	R_b	Operand	Comments
								; Upon entering the subprogram:
								; R3 will contain B
								; R4 will contain C
								; Upon return from the subprogram
								; R1 will contain the results
00001000	22 00 20 00	SUBPROG:	LOAD	D	R1		ONE	;Load 1 into R1
00001004	E2 C0 00 00		ADD	D	R1	R3		;R1 = (1+B)
00001008	24 00 20 04		LOAD	D	R2		THSEV	;Load 37 into R2
0000100C	F2 80 00 00		MUL		R1	R2		;R1 = (1+B)*37
00001010	88 00 00 00		NEG		R4			;R2 = -C
00001014	E3 00 00 00		ADD		R1	R4		;R1 = (1+B)*37 - C
00001018	C0 00 00 00		RET					;Return to calling program
00001100	26 00 20 10	START:	LOAD	D	R3		B1	;Load B into R3
00001104	28 00 20 1C		LOAD	D	R4		C1	;Load C into R4
00001108	D0 00 10 00		CALL	D			SUBPROG	;Go to subprogram
0000110C	32 00 20 28		STORE	D	R1		A1	;Save Results
00001110	26 00 20 14		LOAD	D	R3		B2	;Load B into R3
00001114	28 00 20 20		LOAD	D	R4		C2	;Load C into R4
00001118	D0 00 10 00		CALL	D			SUBPROG	;Go to subprogram

Location	Contents (word)	Label	Op Code	I/D	R_a	R_b	Operand	Comments
0000111C	32 00 20 2C		STORE	D	R1		A2	;Save Results
00001120	26 00 20 18		LOAD	D	R3		B3	;Load B into R3
00001124	28 00 20 1C		LOAD	D	R4		C3	;Load C into R4
00001128	D0 00 10 00		CALL	D			SUBPROG	;Go to subprogram
0000112C	32 00 20 30		STORE	D	R1		A3	;Save Results
00001130	00 00 00 00	END:	HALT					;End of the Program, stop

View Of Memory Before Execution of Program

Location	Contents	Logical Name
00002038		
00002034		
00002030		A3
0000202C		A2
00002028		A1
00002024		C3
00002020		C2
0000201C		C1
00002018		B3
00002014		B2
00002010		B1
00002008		
00002004	00 00 00 25	THSEV (37_{10})
00002000	00 00 00 01	One
00001FFC		

Location	Contents	Logical Name
00001138		
00001134		
00001130	00 00 00 00	END
0000112C	32 00 20 30	
00001128	D0 00 10 00	
00001124	28 00 20 1C	
00001120	26 00 20 18	
0000111C	32 00 20 2C	
00001118	D0 00 10 00	
00001114	28 00 20 20	
00001110	26 00 20 14	
0000110C	32 00 20 28	
00001108	D0 00 10 00	
00001104	28 00 20 1C	
00001100	26 00 20 00	START
000010FC		
00001020		

Location	Contents	Logical Name
0000101C		
00001018	C0 00 00 00	
00001014	E3 00 00 00	
00001010	88 00 00 00	
0000100C	F2 80 00 00	
00001008	24 00 20 04	
00001004	E2 C0 00 00	
00001000	22 00 20 00	SUBPROG
00000FFC		

6.7.6. An Example Application of Hardware and Software

Consider the example chemical experimental set up shown in Figure 22 where a laser beam is used to excite a sample. After the laser beam has been extinguished, the sample relaxes back to the ground state by emitting light. A monochromator and detector can be used to record the decay of the emitted light of a given wavelength.

LaserExp .cdr 21-Oct-2002

T V Atkinson Department of Chemistry Michigan State University

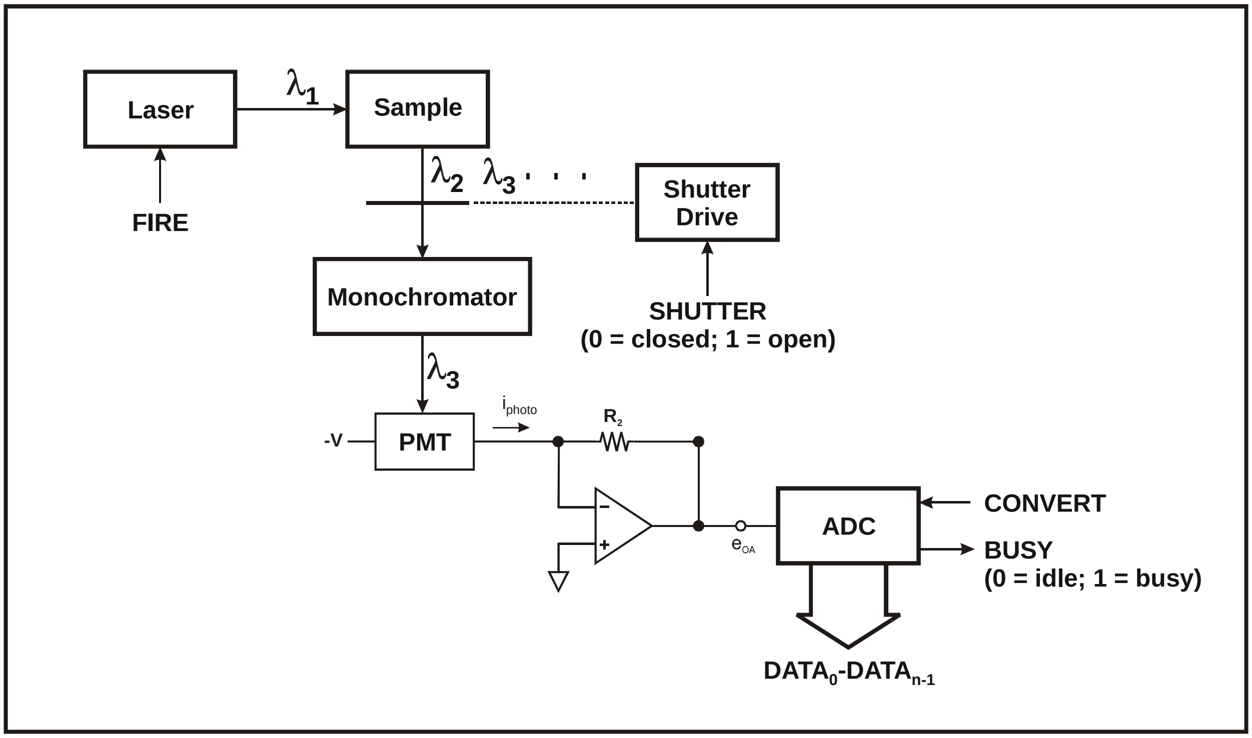


Figure 22 Laser Experiment

The sequence of events in the experiment are detailed below.

1. Shutter is closed. Fire laser.
2. Wait for Laser beam to dissipate.
3. Open Shutter.
4. Start ADC.
5. Wait until ADC has finished conversion.
6. Read ADC data register.
7. Store Number in the next element of the data array.
8. Decrement the count of data points taken.
9. If more points are needed, delay for the amount of time between acquiring points and then go back to Step 4

The interface to this experiment would look like Figure 23 to the software. Figure 24 show the first several hundred microseconds of the time course of the experiment using the program.

Implementation Steps:

1. Design
    a. Design the details of the experiment.
    b. Identify signals to be interfaced.
    c. Identify the steps required for execution of the experiment.
    d. Build the interface. Decide on the actual physical addresses of the various registers in the interface.
2. Program Development
    a. Decide how the CPU registers are to be utilized by the program.
    b. Lay out the logical flow of the program. Generate the actual program steps in nmemonic form.
    c. Layout the utilization of memory.
        i. Decide where to put the arrays to receive the data.
        ii. Decide where the constants are to be stored.
        iii. Decide where the first instruction of the program is to be stored.
        iv. Determine the address of each instruction of the program.
    d. Translate the mnemonic instructions into machine language (binary). Insert into the instructions the actuasl physical addresses of operands and branch points.
3. Set Up
    a. Assemble all experimental equipment.
    b. Install any interface hardware.
    c. Make all connections between the interface and the experimental apparatus.
    d. Load memory by depositing the machine language form into the appropriate locations.
4. Debug the facility
    a. Deposit the address of the logical entry point of the program into the CPU PC register.
    b. Assert GO.
    c. Observe operation during and after the run to insure the facility is working correctly. You may have to use a known sample to facilitate this operation.
    d. Modify the program and repeat the debug process until correct.
5. Production
    a. Place the appropriate sample in the apparatus. Select the appropriate wave lenth to be observed
    b. Execute the program
    c. Transfer the data to the appropriate place(s) for long term storage, analysis, and presentation.

LaserExpReg.cdr 21-Oct-2002

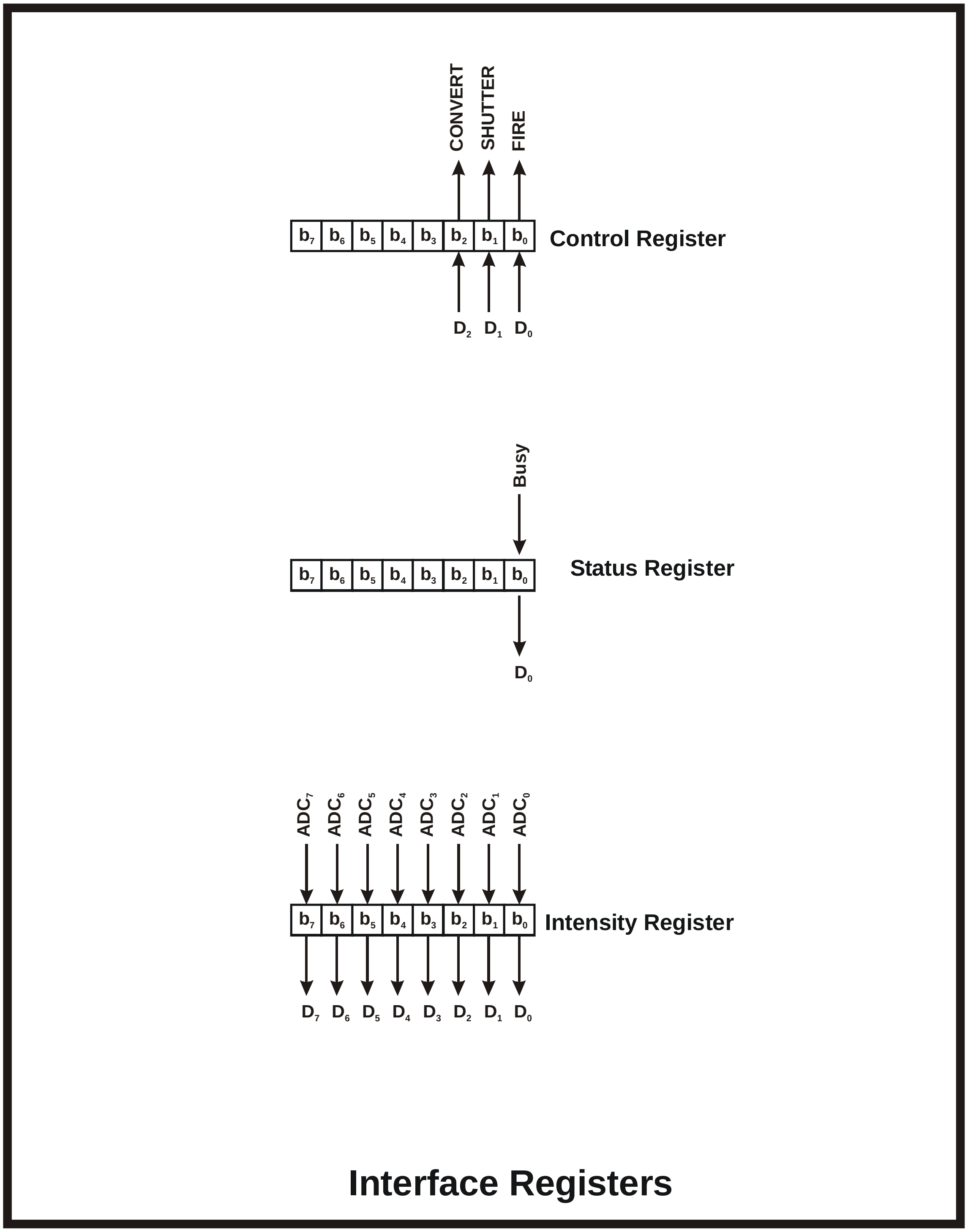

CONVERT
SHUTTER
FIRE
b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
Control Register
D_2 D_1 D_0
Busy
b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
Status Register
D_0
ADC_7 ADC_6 ADC_5 ADC_4 ADC_3 ADC_2 ADC_1 ADC_0
b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
Intensity Register
D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0
Interface Registers

Figure 23 Laser Experiment Interface (software View)

There are three device controller (interface) registers associated with this interface.

6.7.7. Sample Program

Notice that the program makes the following use of the CPU registers. Assume that each instruction takes 2.0 microseconds to execute.

Reg	Use
R1	Control word for interface.
R2	Number of points yet to acquire.
R3	Next storage location in the array to hold the data points.
R4	ADC value
R5	Delay counter

Location	Contents (word)	Label	Op Code	I/D	R_a	R_b	Operand	Comments
								; --- Initialization ---
00001000	24 00 A0 18	START:	LOAD	D	R2		NUMPNT	;Set number of points to acquire
00001004	26 00 A0 20		LOAD	D	R3		POINT	;Set storage pointer
								; --- Excite the sample ---
00001008	22 00 A0 0C		LOAD	D	R1		#1	;Fire Laser
0000100C	32 1F FF EC		STORE	D	R1		CONTROL	
00001010	10 00 00 00		NOOP					;Kill 8 micro seconds
00001014	10 00 00 00		NOOP					; while laser beam dies
00001018	10 00 00 00		NOOP					
0000101C	10 00 00 00		NOOP					
00001020	22 00 A0 10		LOAD	D	R1		#2	;Open shutter
00001024	32 1F FF EC		STORE	D	R1		CONTROL	
00001028	10 00 00 00		NOOP					;Kill 4 microsec while shutter settles
0000102C	10 00 00 00		NOOP					
								; --- Acquire Data ---
00001030	22 00 A0 14	LOOP1:	LOAD	D	R1		#6	
00001034	32 1F FF EC		STORE	D	R1		CONTROL	;Start ADC, keep shutter open
00001038	22 00 A0 10		LOAD	D	R1		#2	;Reset ADC trigger, keep shutter open
0000103C	32 1F FF EC		STORE	D	R1		CONTROL	
00001040	32 1F FF F0	LOOP2:	LOAD	D	R1		STATUS	;Wait until ADC is done
00001044	92 00 10 40		BNE		R1		LOOP2	
00001048	28 1F FF F4		LOAD	D	R4		DATA	;Get ADC value
0000104C	38 E0 00 04		STORE	I	R4	R3		;Store the value in data array
00001050	E6 00 A0 0C		ADD	D	R3		#1	;Increment the pointer
00001054	2A 00 A0 00		LOAD	D	R5		DELAY	;Wait to take the next point
00001058	EA 00 A0 14	LOOP3:	ADD	D	R5		#-1	;Decrement delay counter. Kill time
0000105C	9A 00 10 58		BNE		R5		LOOP3	;Branch if delay time is not over
00001060	E4 00 A0 14		ADD	D	R2		#-1	;Decrement the counter

Location	Contents (word)	Label	Op Code	I/D	R_a	R_b	Operand	Comments
00001064	94 00 10 30		BNE		R2		LOOP1	;Branch if not done
00001068	00 00 00 00		HALT					;All done, stop

View Of Memory Before Execution

Location	Contents	Logical Name
001FFFFC		Last word
001FFFF8		
001FFFF4		DATA
001FFFF0		STATUS
001FFFEC		CONTROL
001FFFE8		
001FFFE4		
001FFFE0		

0000A03C		
0000A038		
0000A034		
0000A030		
0000A02C		
0000A028		
0000A024		
0000A020		data array
0000A01C		
0000A018	00 00 03 E8	NUMPNT (100_{10})
0000A014	00 00 00 06	6
0000A010	00 00 00 02	2
0000A00C	00 00 00 01	1
0000A008	FF FF FF FF	-1
0000A004	00 00 A0 20	POINT
0000A000	00 00 00 0F	DELAY (15_{10})
00009FFC		
00009FF8		

Location	Contents	Logical Name
00001070		
0000106C		
00001068	00 00 00 00	
00001064	94 00 10 30	
00001060	E4 00 A0 14	
0000105C	9A 00 10 58	
00001058	EA 00 A0 14	
00001054	2A 00 A0 00	
00001050	E6 00 A0 0C	Loop3
0000104C	38 E0 00 00	
00001048	28 1F FF F4	
00001044	92 00 10 40	
00001040	32 1F FF F0	LOOP2
0000103C	32 1F FF EC	
00001038	22 00 A0 10	
00001034	32 1F FF EC	
00001030	22 00 A0 14	LOOP1
0000102C	10 00 00 00	
00001028	10 00 00 00	
00001024	32 1F FF EC	
00001020	22 00 A0 10	
0000101C	10 00 00 00	
00001018	10 00 00 00	
00001014	10 00 00 00	
00001010	10 00 00 00	
0000100C	32 1F FF EC	
00001008	22 00 A0 0C	
00001004	26 00 A0 20	
00001000	24 00 A0 18	START
00000FFC		

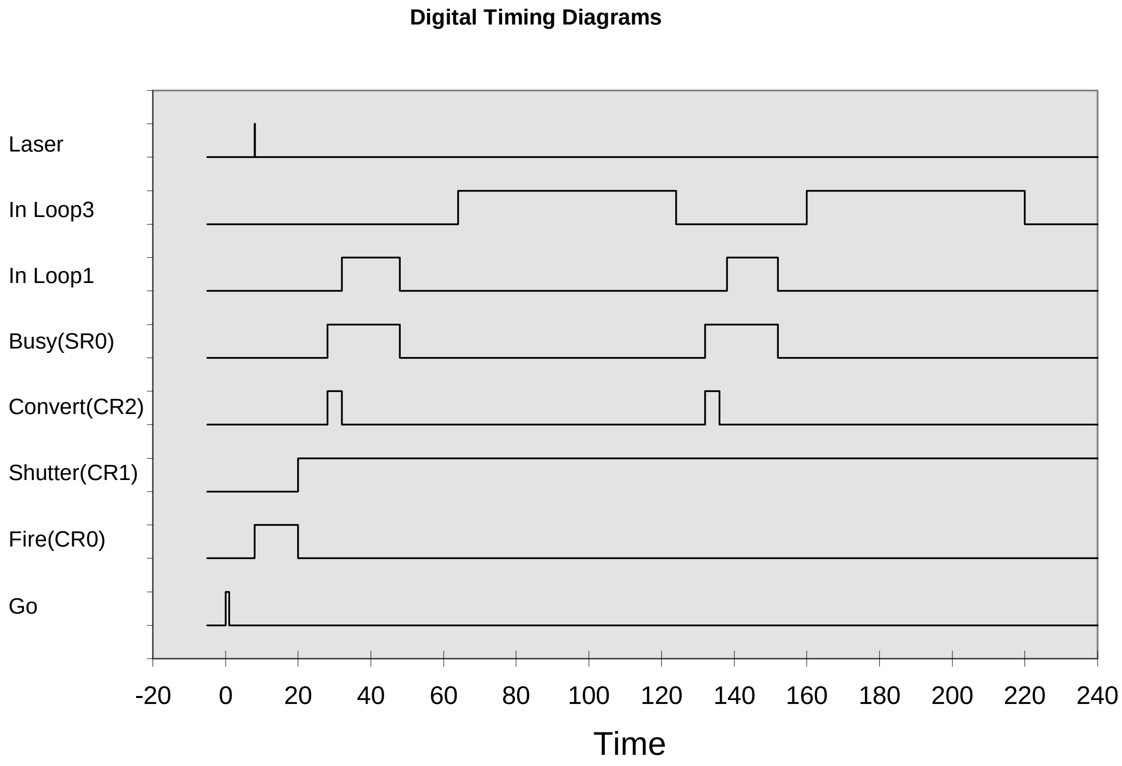


Figure 24 Laser Experiment Timing (microseconds)

7. Computer Architecture Taxonomy

7.1. Special Buses

CEM924 CPU 3 4-APR-1992

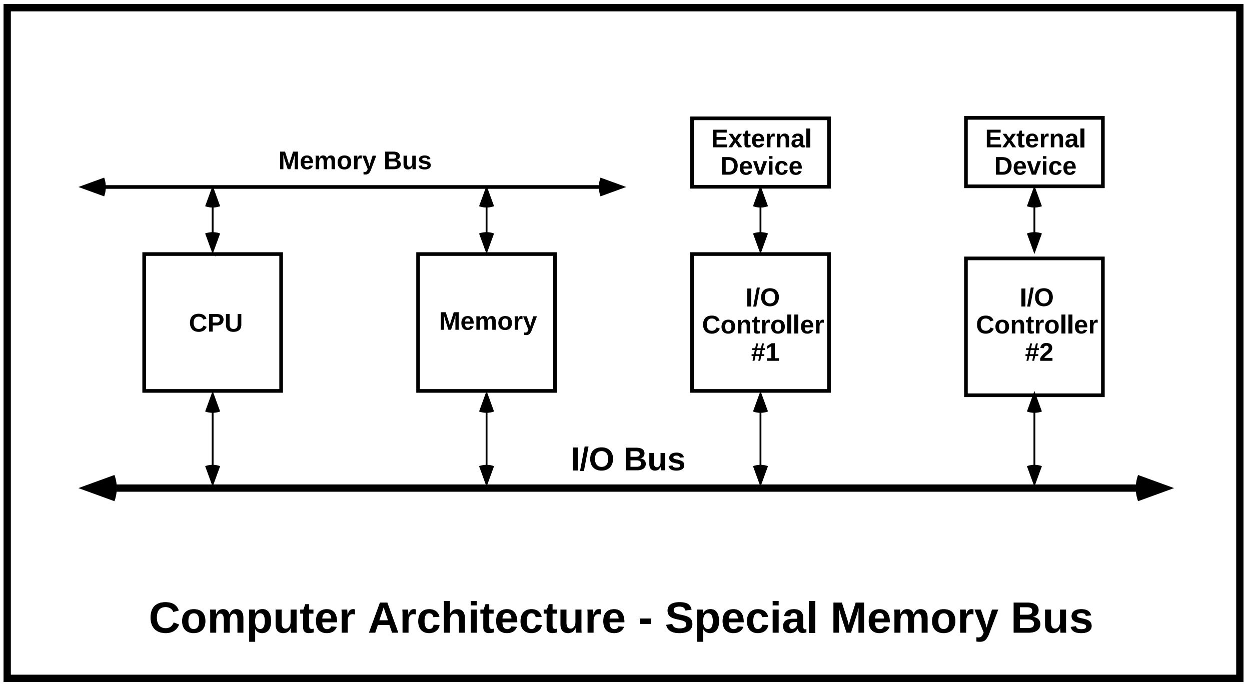
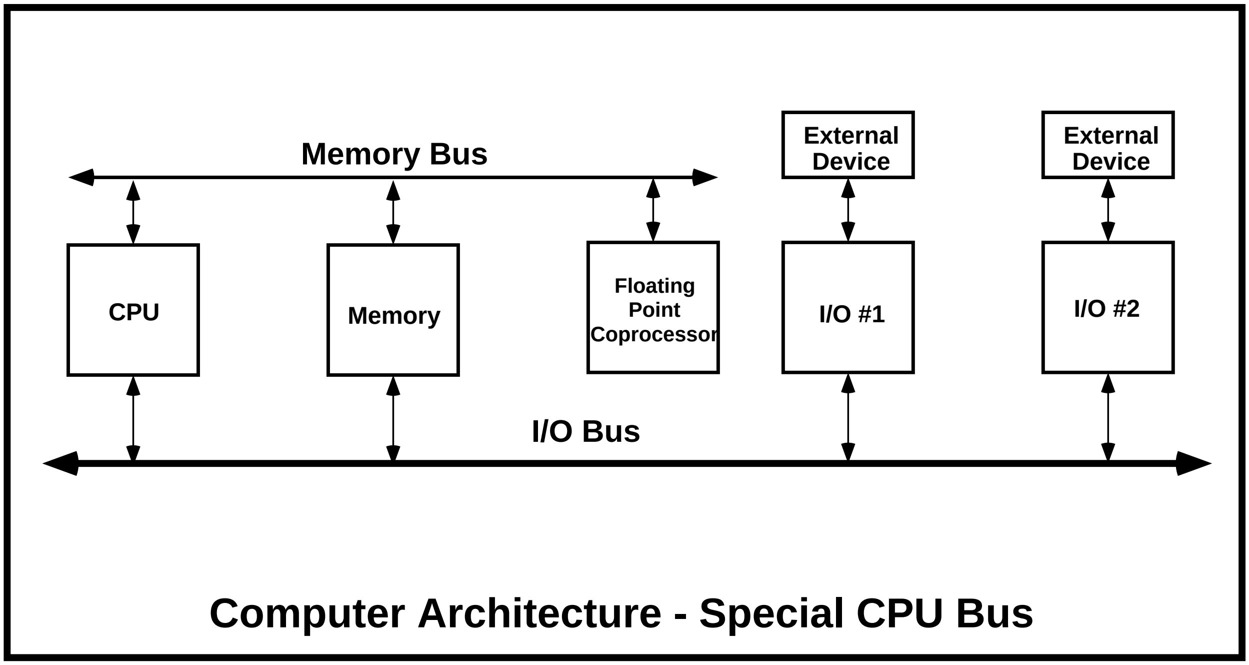


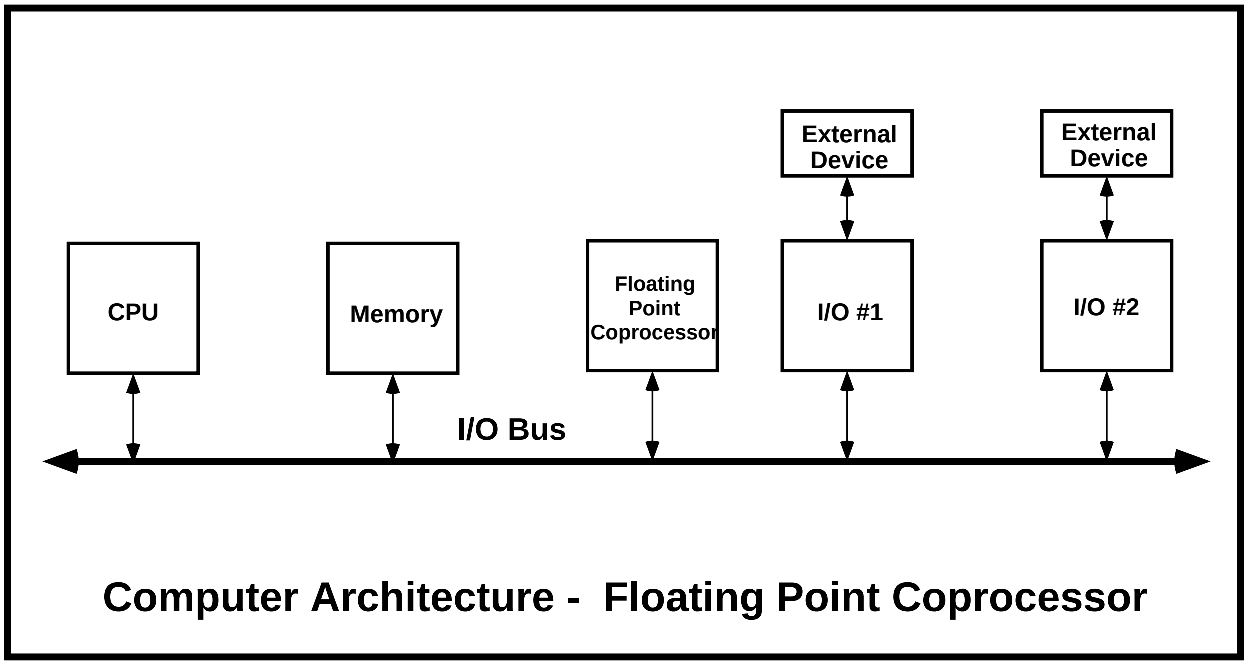
Figure 25 Special Memory Bus

7.2. Coprocessors



CEM 924 CPU 4 4-APR-1992

Figure 26 Special CPU Bus


External Device
External Device
CPU
Memory
Floating Point Coprocessor
I/O #1
I/O #2
I/O Bus
Computer Architecture - Floating Point Coprocessor

CEM 924 CPU 5 4-APR-1992

Figure 27 Floating Point CoProcessor I

7.3. Multiple I/O buses

CEM 924 CPU 6 4-APR-1992

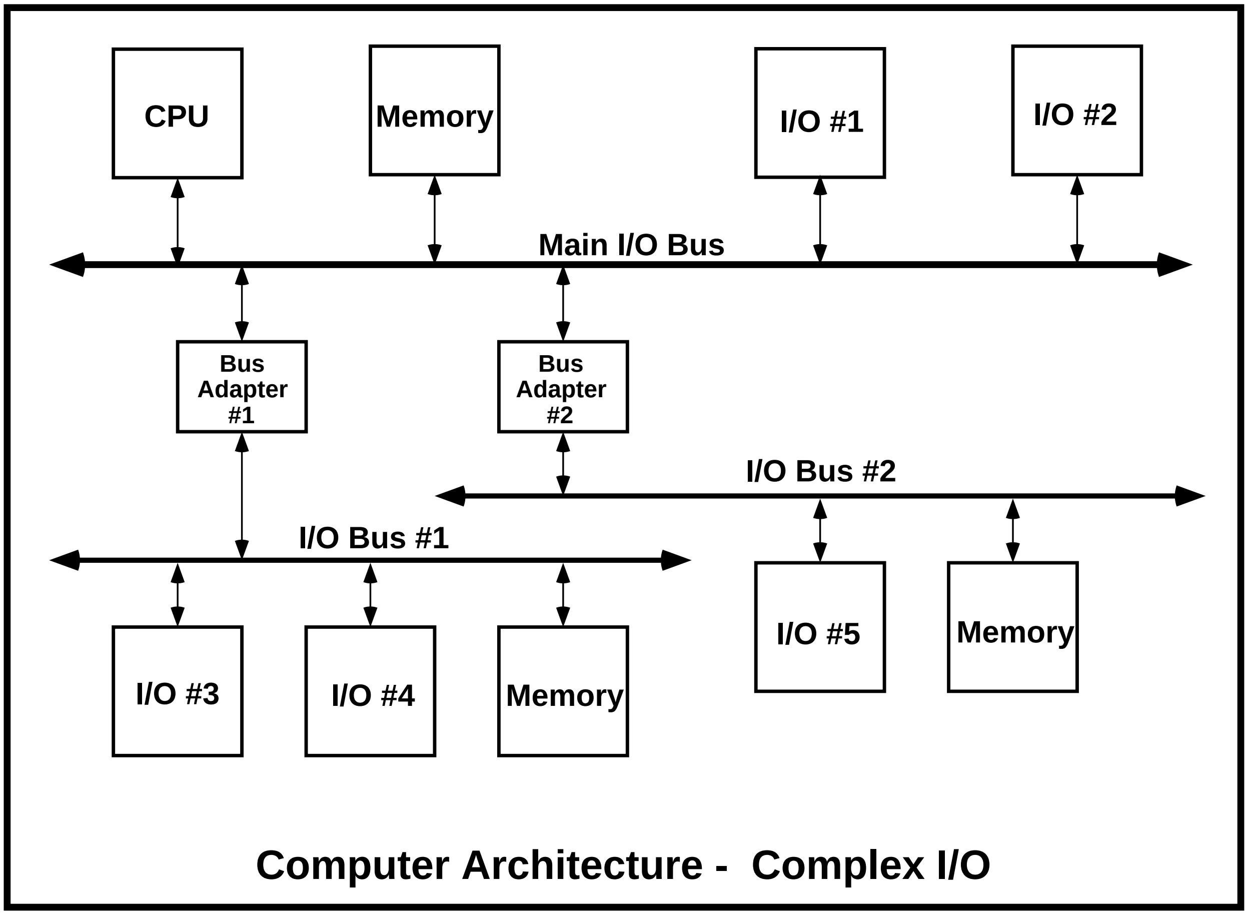
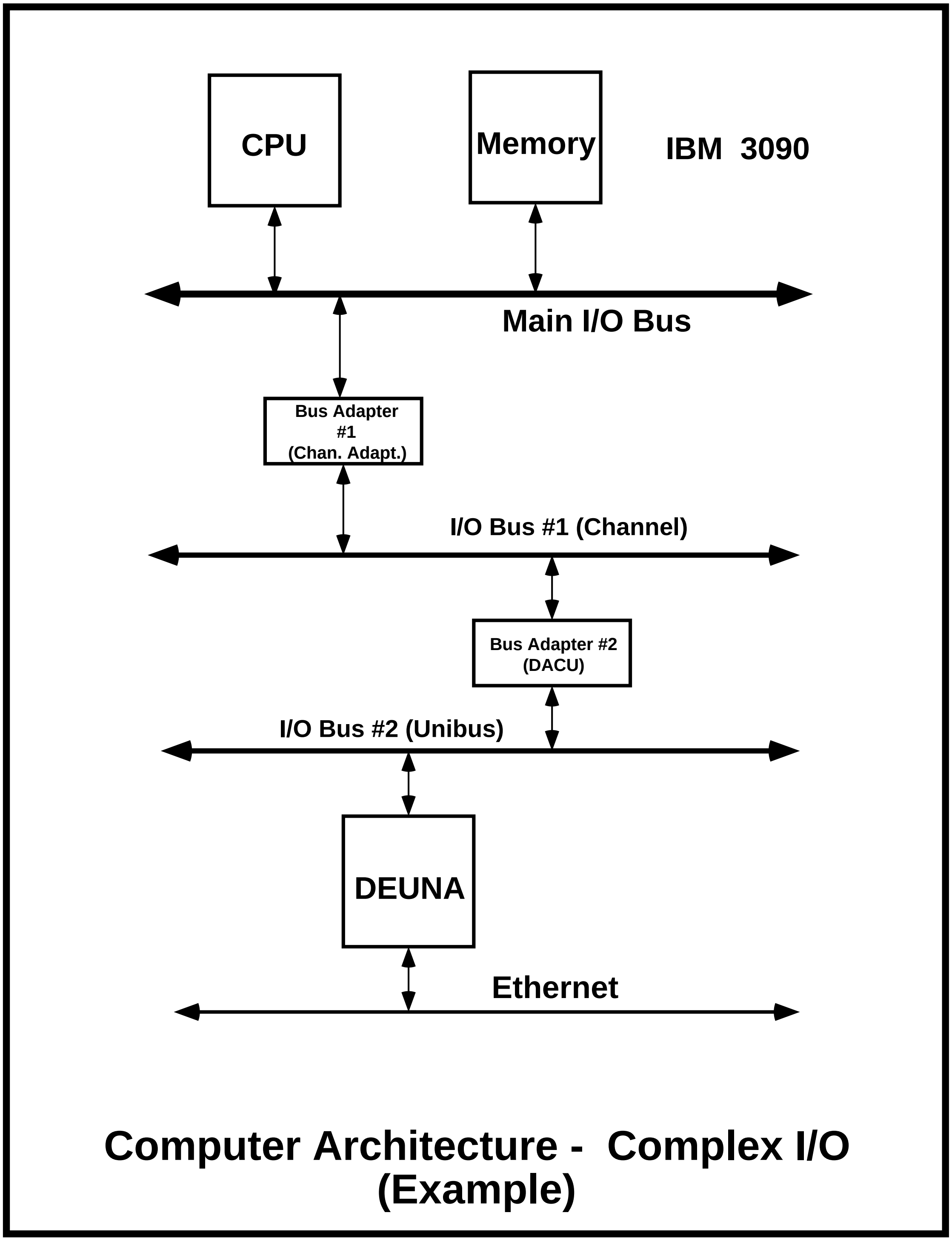


Figure 28 Complex I/O

CEM 924 CPU 7 4-APPR-1992



Computer Architecture - Complex I/O (Example)

Figure 29 Complex I/O: An Example

7.3.1.Motivations

1. Performance -

2. Compatibility with existing equipment
3. Compatibility with other vendors

7.3.1. Problems

1. Complexity of Hardware
2. Timing Delays
3. Extra Software

7.3.2. Examples

1. DEC PDP8: PDP8 <--> PDP8I <--> PDP8E
2. IBM PC: XT <--> AT <--> EISA, MCA, PCI, or Local Bus
3. DEC VAX: SBI (11/780) <--> MASSBUS
4. DEC VAX: SBI (11/780) <--> UNIBUS
5. DEC VAX: SBI (11/780) <--> QBUS
6. IBM PC – ISA
7. IBM PC – PCI
8. SCSI
9. IEEE 488

8. Multiple Processors

CEM924 CPU 8 4-APR-1992

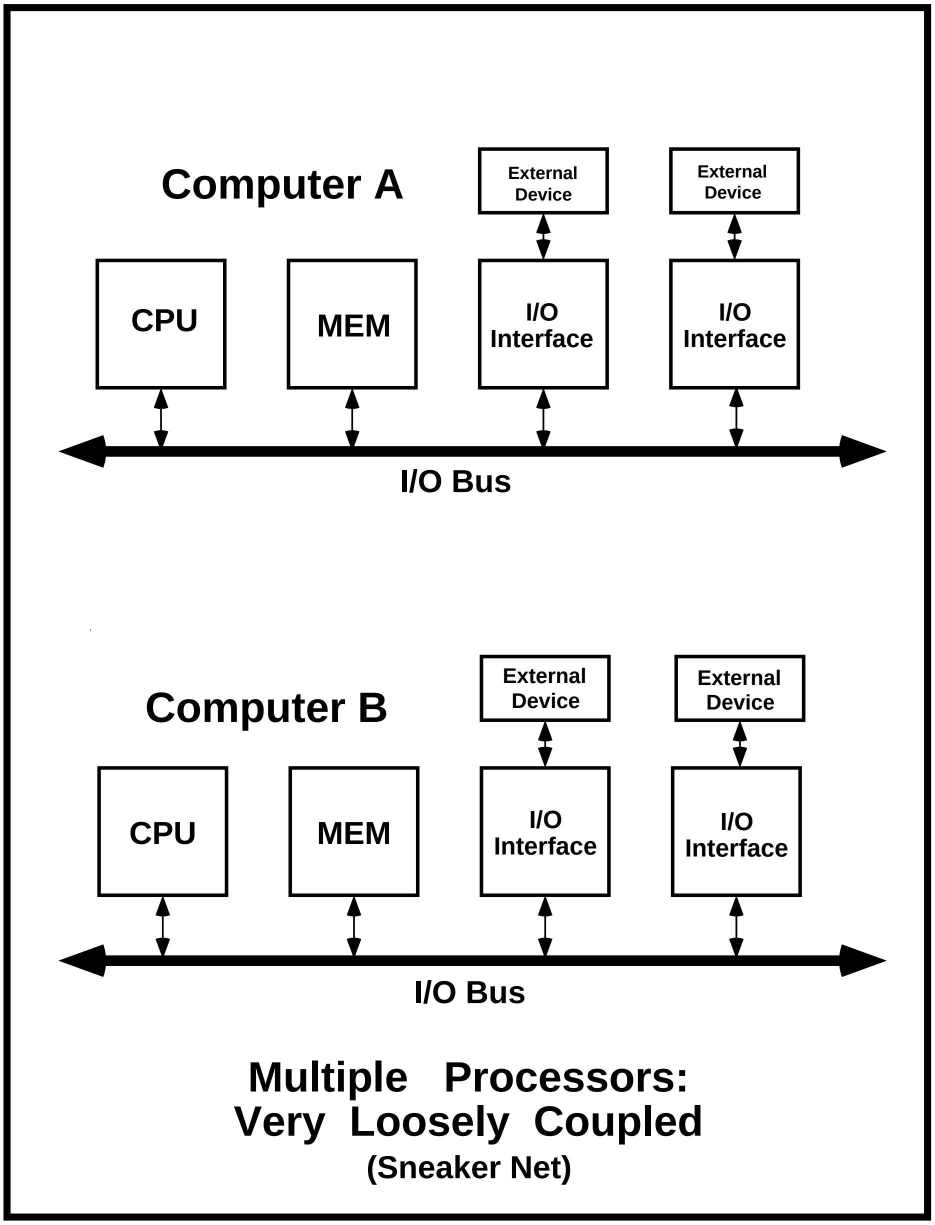


Figure 30 Multiple Processors: Very Loosely Coupled

CEM924CPU 9   4-APR-1992

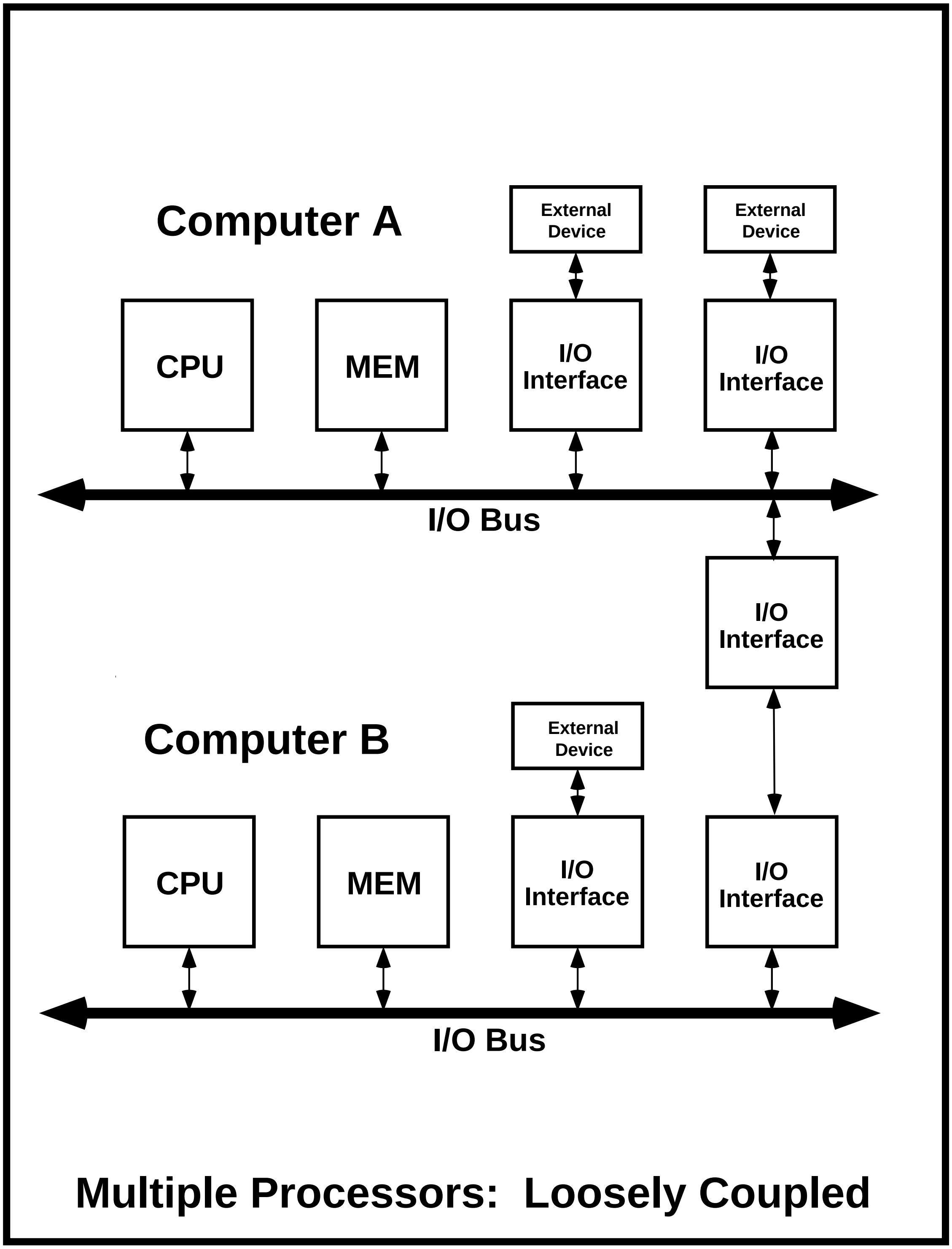


Figure 31 Multiple Processors: Loosely Coupled

CEM924 CPU 10 4-APR-1992

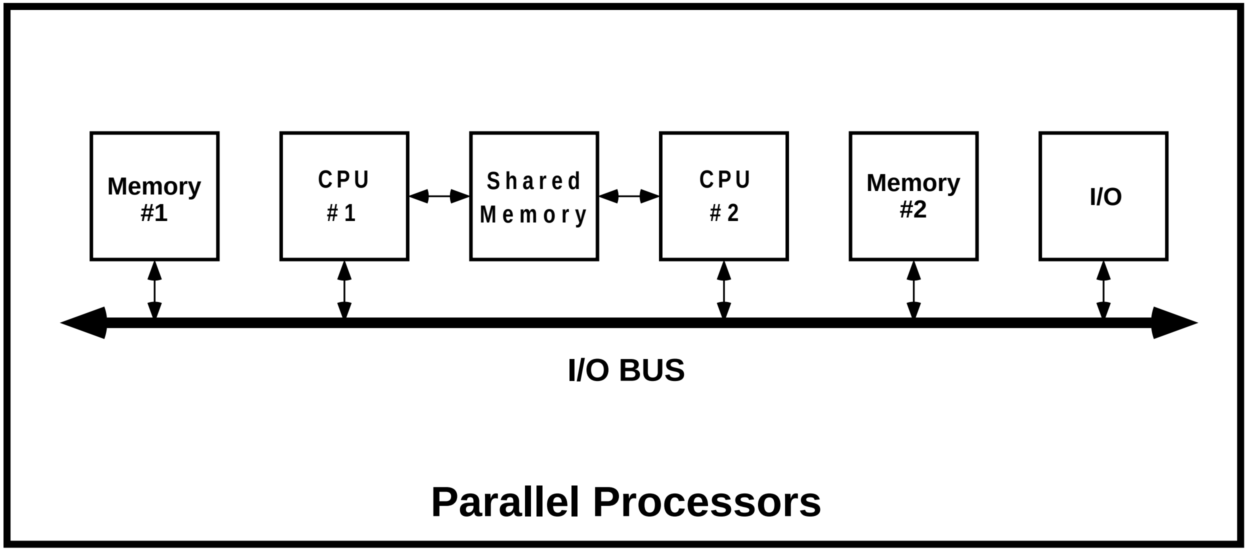


Figure 32 Multiple Processors: Parallel

Net Topology 4-APR-1992
T V Atkinson
Department of Chemistry
Michigan State University

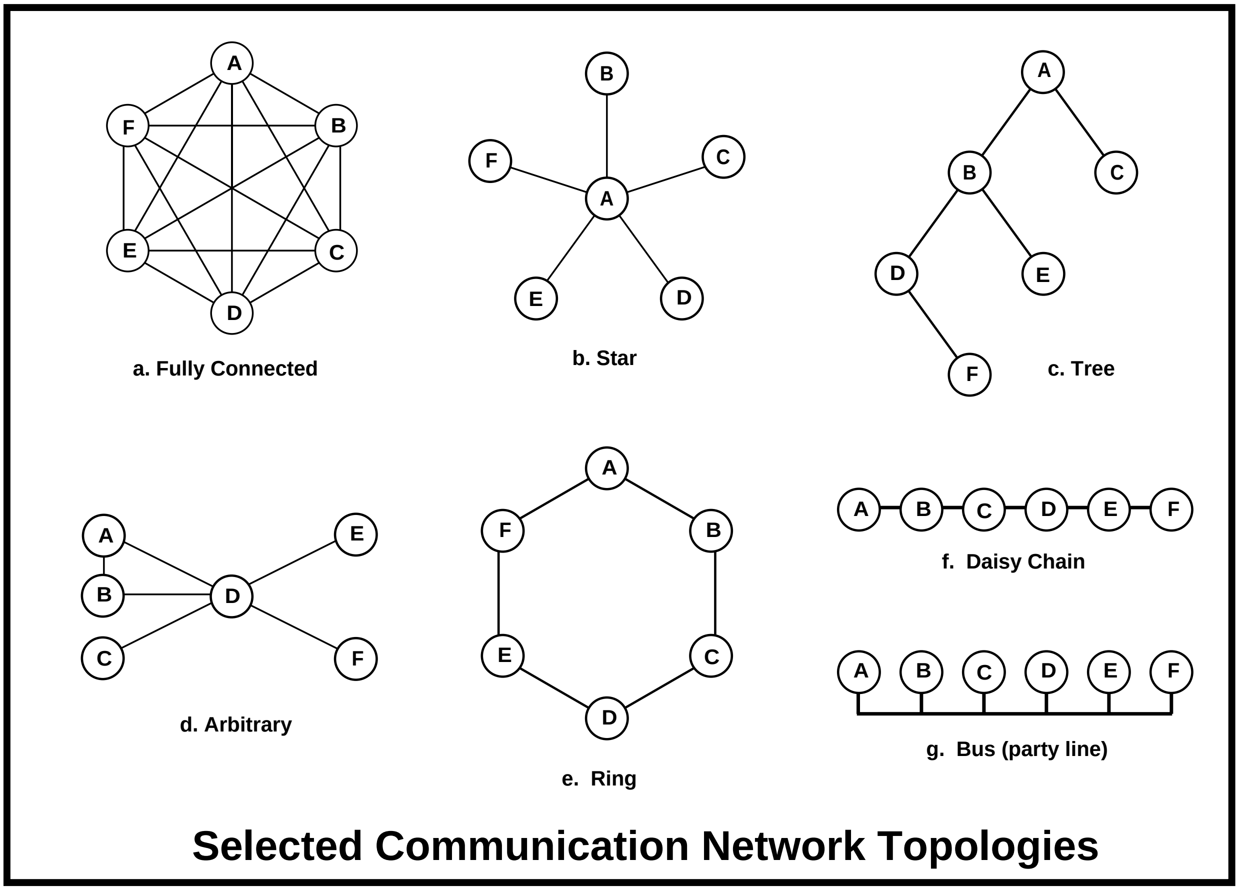


Figure 33 Multiple Processors: Connection Topologies

9. Disk Drives

9.1. General Architecture

Figure 34, Figure 35, and Figure 36 are a generalized depiction of a modern disk drive. This particular drive has two platters, four surfaces, and 8 heads. In actual practice, a drive may have one or more platters. One or both surfaces maybe used to contain data. Each surface may have one, two, or even more heads. Only one head is active at any given time. The head positioner places the heads over the track to be read or written. Various physical techniques are used to change (write) or sense (read) the magnetization of small domains of magnetic oxide within a track. Each bit of information will be encoded into one of these domains.

CEM 924 DISK1A 4-APR-1992

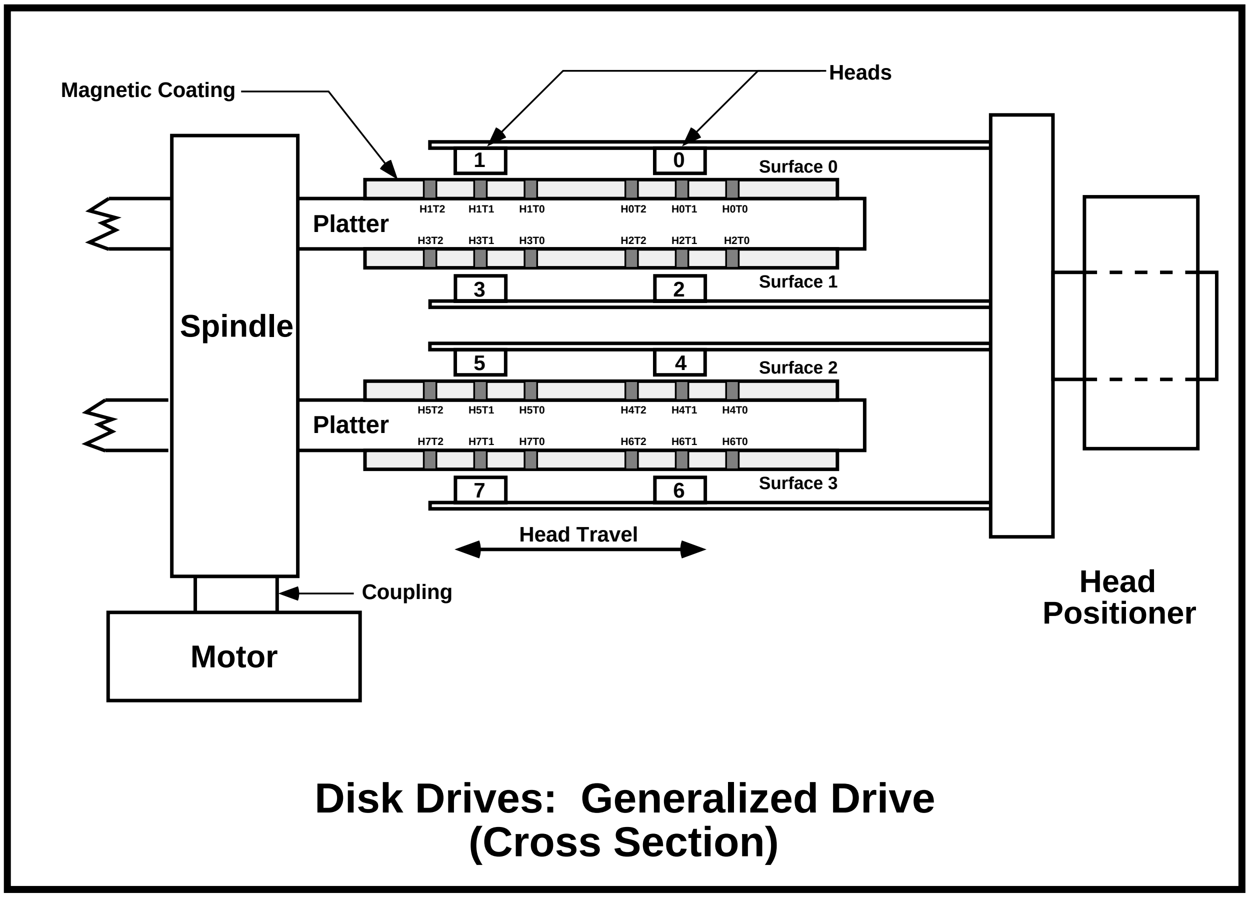
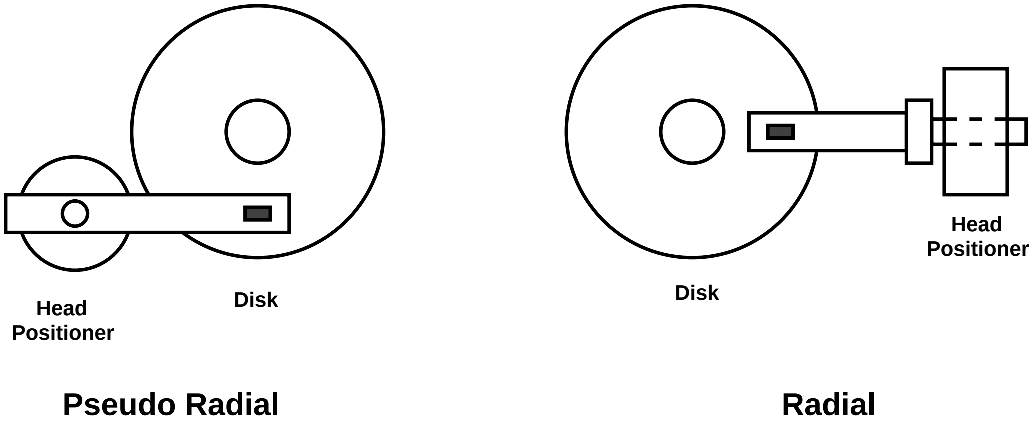


Figure 34 Generalized Drive (Cross Section

CEM 924 DISK1E 5-APR-1992



**Disk Drives: Head Positioners
(Top View)**

Figure 35 Head Positioners

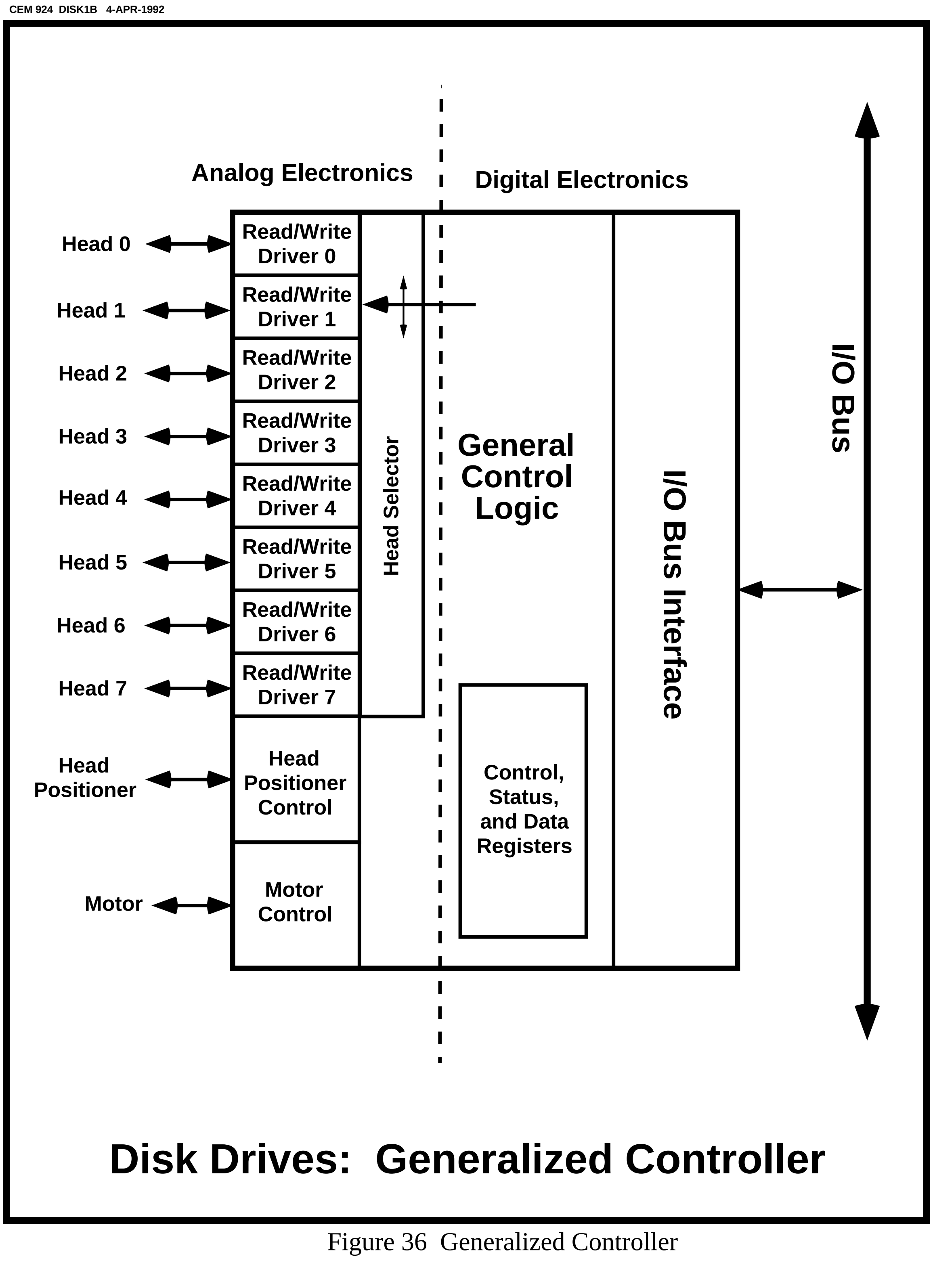


Figure 36 Generalized Controller

9.2. Disk Format

1. Sectors
   Preamble – track address, sector address (overhead)
   Data – the user’s data
   Postamble - redundancy or error detection data (overhead)
2. Tracks
3. Cylinders
4. Partitions

CEM 924 Disk 1C 5-APR-1992

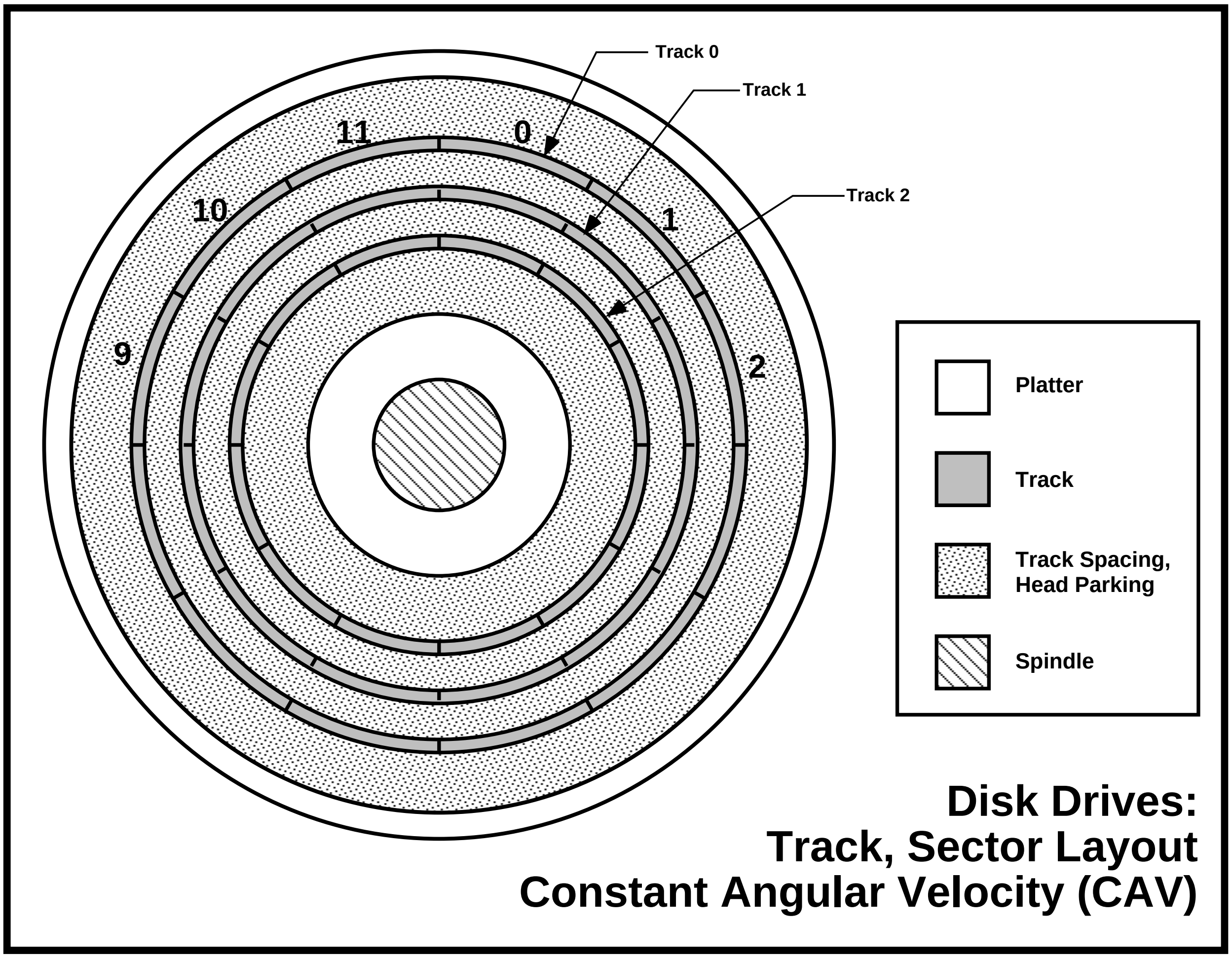
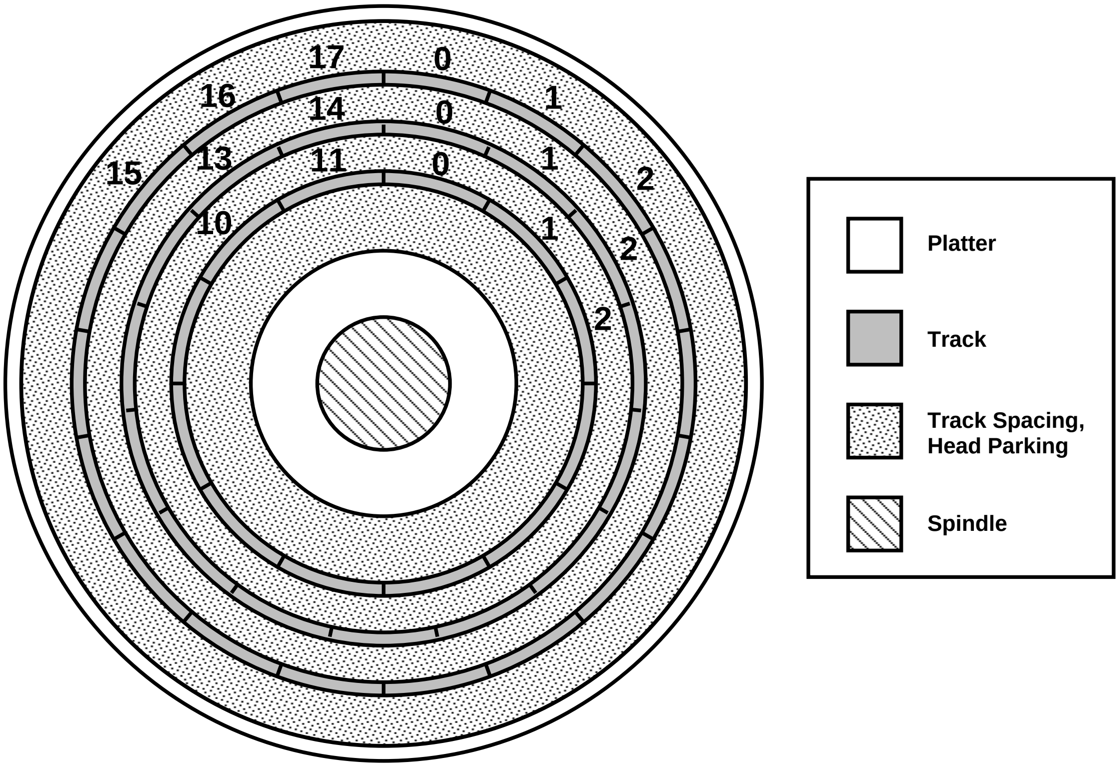


Figure 37 Track Sector Layout: CAV

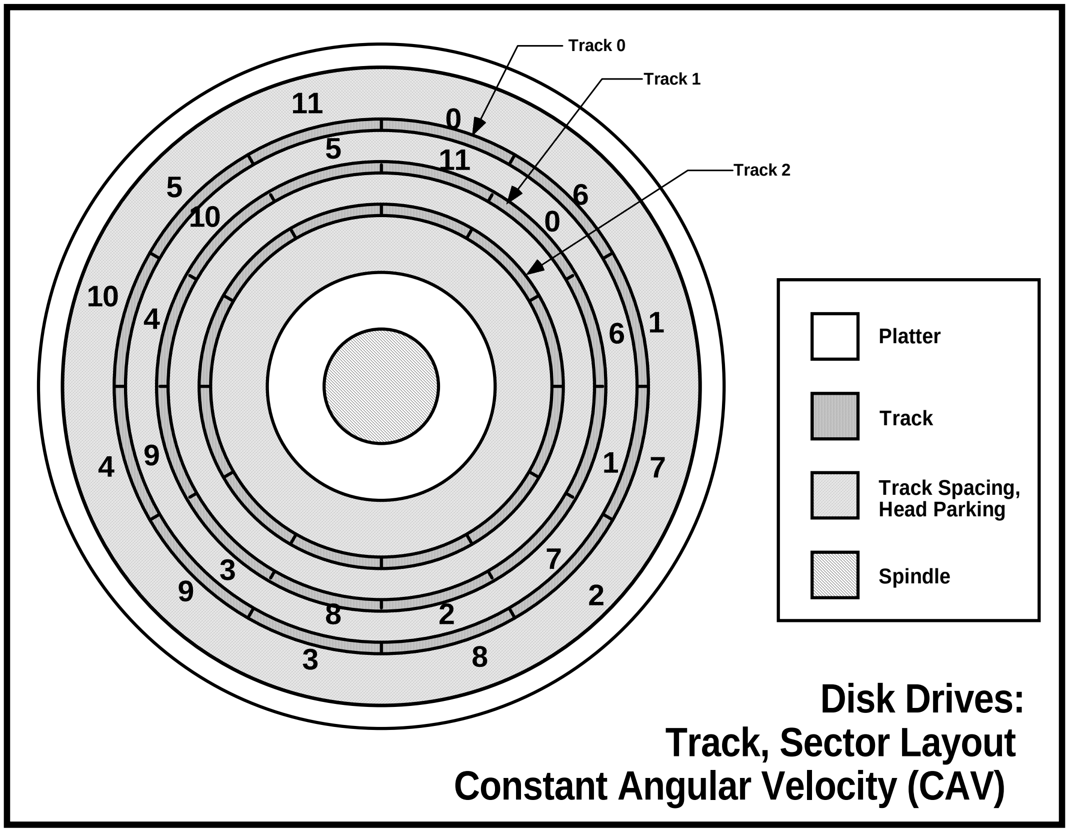
CEM 924 Disk 1D 5-APR-1992



Disk Drives: Track Sector Layout
Constant Linear Velocity (CLV)

Figure 38 Track Sector Layout: CLV

CEM 924 Disk 1C 5-APR-1992



Disk Drives:
Track, Sector Layout
Constant Angular Velocity (CAV)

Figure 39 Track Skew - Interleave

9.3. Mapping Sectors into Logical Blocks

A method of increasing performance is to interleave sectors and to skew tracks. Interleaving sectors changes the formating of the track so that logically ajacent sectors are actually separated physically, this provides time for the computer to digest one sector of information and get ready for the next before the next sector arrives below the head. The penalty for not being ready is waiting for a complete revolution of the disk before the desired sector again appears under the head.

Track skew is a similar technique. The disk is formatted so that sector 0 of the next track is located some angle (number of sectors) from the angular position of sector 0 of the previous track.

Both techniques are based on the fact that most disk read or write operations involve a number of logically consecutive sectors.

Table 15 Disks: Mapping Physical Sectors into Logical Blocks

Interleave		0		1		2		1
Track Skew		0		0		0		1
Logical Block	Track	Sector	Track	Sector	Track	Sector	Track	Sector
0	0	0	0	0	0	0	0	0
1	0	1	0	2	0	3	0	2
2	0	2	0	4	0	6	0	4
3	0	3	0	6	0	9	0	6
4	0	4	0	8	0	1	0	8
5	0	5	0	10	0	4	0	10
6	0	6	0	1	0	7	0	1
7	0	7	0	3	0	10	0	3
8	0	8	0	5	0	2	0	5
9	0	9	0	7	0	5	0	7
10	0	10	0	9	0	8	0	9
11	0	11	0	11	0	11	0	11
12	1	0	1	0	1	0	1	1
13	1	1	1	2	1	3	1	3
14	1	2	1	4	1	6	1	5
15	1	3	1	6	1	9	1	7
16	1	4	1	8	1	1	1	9
17	1	5	1	10	1	4	1	11
18	1	6	1	1	1	7	1	0
19	1	7	1	3	1	10	1	2
20	1	8	1	5	1	2	1	4
21	1	9	1	7	1	5	1	6
22	1	10	1	9	1	8	1	8
23	1	11	1	11	1	11	1	10
24	2	0	2	0	2	0	2	0
25	2	1	2	2	2	3	2	2

9.4. Figures of Merit for a Disk

FCI - Flux changes per inch. Density of flux changes along a track.

BPI - Bits per inch along a track.

TPI - Tracks per inch along the radius of the disk.

Areal Density - Density of data bits per square area.

Rotational Speed

Track to adjacent track seek time

Seek time (average) - The average time required to seek a given sector. This is the sum of one half the number of tracks times the track to track seek time plus one half of the time for one rotation.

Table 16 Disks: Example Drives

Attribute	Units	Kennedy 5380	Kennedy 7300	Seagate ST-12550W	DEC Rx02 (Floppy)
Platter diameter	inches	14 in	8 in	3.5	8
Number of platters		3	3	10	1
Number of data surfaces		5	5	19	1
Number of heads		5	5	19	1
Number of cylinders		823		2707	77
Bits per inch		6330	9420	52187	
Tracks per inch		430	480	3047	48
Tracks per surface		411	411		77
Capacity (unformatted)	Bytes	82M	41.4M	2572M	512K
Head flying heighth	μin	20	15		
Track to adjacent track seek	msec	10	6	0.6	
Track to track seek (max)	Msec	65	55	18	
Track to track (average)	Msec	35	30	8.9	
Spindle Rotation rate	Rpm	3000	3600	7200	360
Rotation times	Msec	20	16.7	8.33	166.7
Transfer Rate	Bits per sec	9.67M		35.3M	62.5K
Power consumption	Watts		75	13	
Mean time between failures (MTBF)	Hrs	10000	10000	500000	
Drive Size	in	7x17x25	4.6x8.5x14.25	1.6x4.0x6	
Drive weight		75 lbs	20 lbs	2.3	
Date of information		1982	mid 1980’s	1994	mid 1970’s

9.5. Combinations of Disks

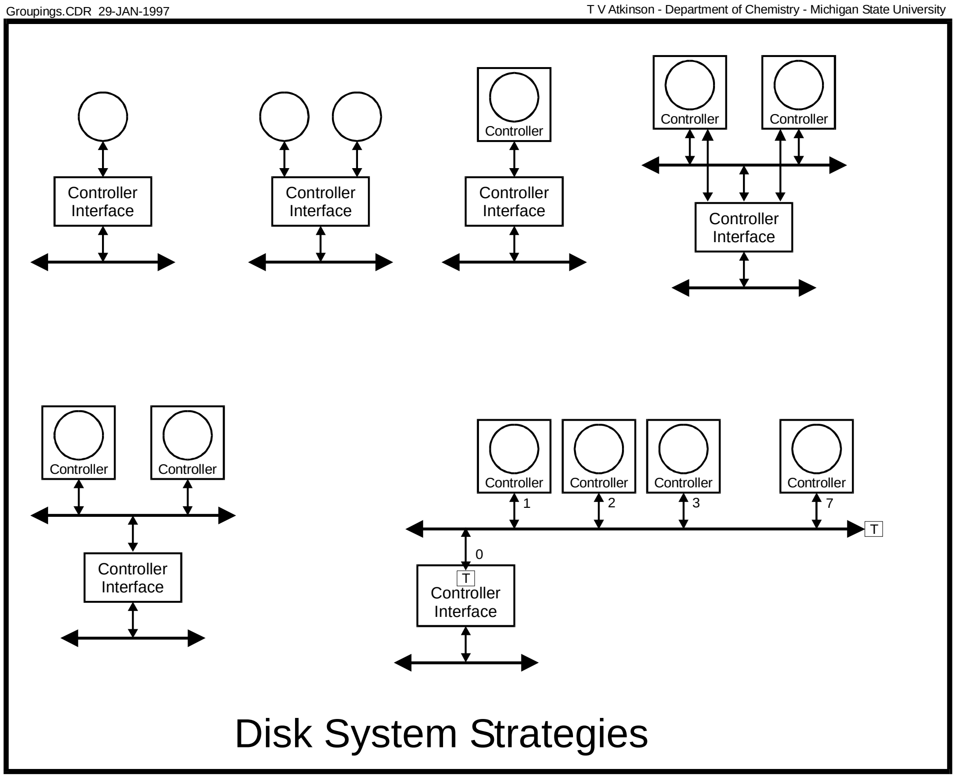

Groupings.CDR 29-JAN-1997
T V Atkinson - Department of Chemistry - Michigan State University
Controller Interface
Controller
1
2
3
7
0
T
Disk System Strategies

Figure 40 Disk System Strategies

9.5.1. Combinations of Simple Disks

The computer industry has typicaly sought three goals (performance, reliability, availability, and low cost) that are often at odds with one another. This is true for complete systems as well as particular subsystems. This section examines some of the developments centered on using more than one of the simpler disk drives. Many of these techniques are embodied in a formailization called Redundant Arrays of Inexpensive Disks (RAID)[2].

RAID is a set of techniques to provide higher performance and highly available disk systems using a number of drives and/or controllers in concert. The original intent was to use a combination of inexpensive disks to achieve the performance and functionality of large expensive disks. A number of taxonomies have been identified. Patterson, Gibson, and Katz of UC Berkeley first proposed RAID in 1987. RAID-1, 3, and 5 have been the most popular so far.

The array of disks appears as one logical drive. A given file is distributed over the drives in a defined manner. Redundancy is added to allow for recovery of data in the case of failures. Redundancy is extra data (overhead) that is stored with the data to enhance the probability that when reading the data back off the disk that two things will happen. First, that any errors in

[2] “An Introductions to RAID, Redundant Arrays of Inexpensive Disks,” Pete McLean, April 24, 1991, Digital Equipment Corporation.

reading the data will be detected. Second, if errors occur, the original data can be reconstructed using the redundant information. Performance is increased since each drive in the RAID set can be seeking and reading various pieces of the requested set of data independently and simultaneously of the other drives.

RAID-0: Simple disk striping where a file is divided into chunks. Each successive chunk is stored on the same block of the next disk of the set. When the last disk of the set has been used, the next chunk goes in the next available block on the first drive, … There is actually no redundancy in this case.

RAID-1: The original example and is also called disk shadowing or mirroring. As each block of a file is written to the disk system, a copy of the block is written on each drive of the RAID set. In case of disk failure, data can be retrieve from the other drives of the set.

RAID-2: Similar to RAID-3 except that a Hamming code is used to generate a number of redundancy chunks per subset of data chunks.

RAID-3: As in RAID-0, the file is divided into chunks and stored on n-1 disks of the RAID set. The nth disk of the RAID set contains a redundancy chunk, i.e., the Xor of the subset of data chunks, stored on the data disks in the corresponding blocks.

RAID-4: Similar to RAID-5 except that the redundancy chunks are all on one disk.

RAID-5: A redundancy chunk is used as in RAID-3 but any given drive contains both redundancy blocks and data blocks.

RAID-6: A more complicated redundancy algorithm is used, producing two chunks of redundancy information for each set of n-2 data chunks. As in RAID-5, data and redundancy chunks are distributed over all drives.

RAID0.CDR 29-JAN-1997 T V Atkinson - Department of Chemistry - Michigan State University

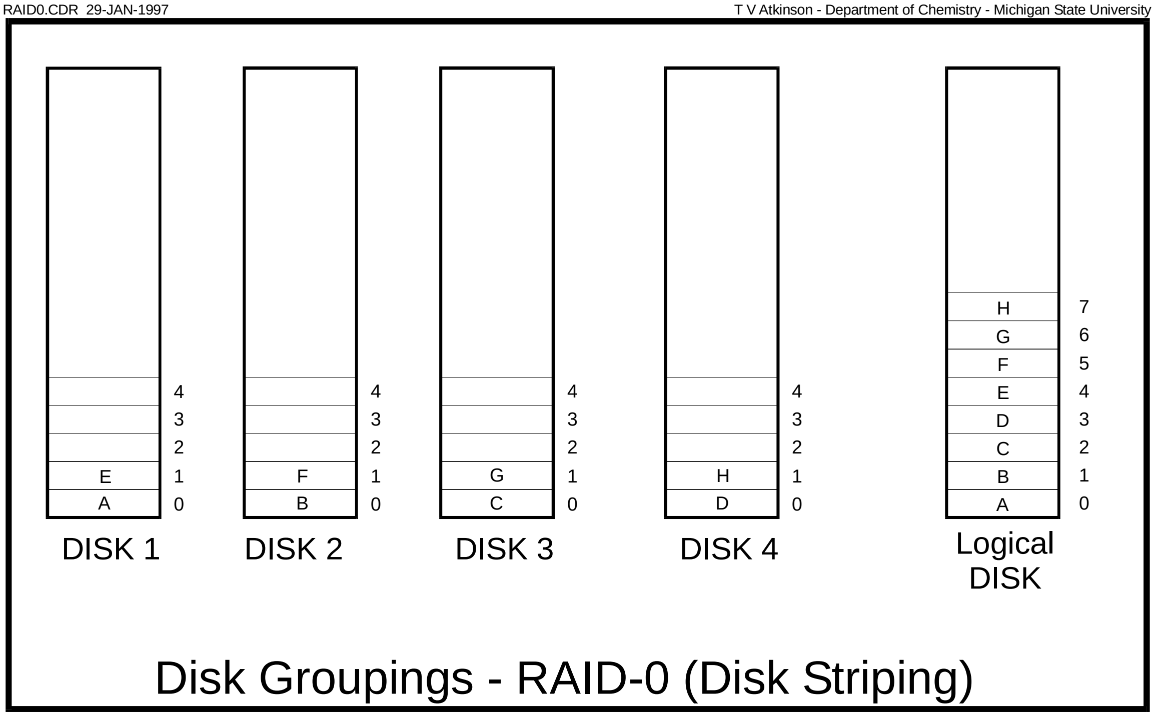


Figure 41 Disk Groupings - Raid 0

RAID1a.CDR 29-JAN-1997 T V Atkinson - Department of Chemistry - Michigan State University

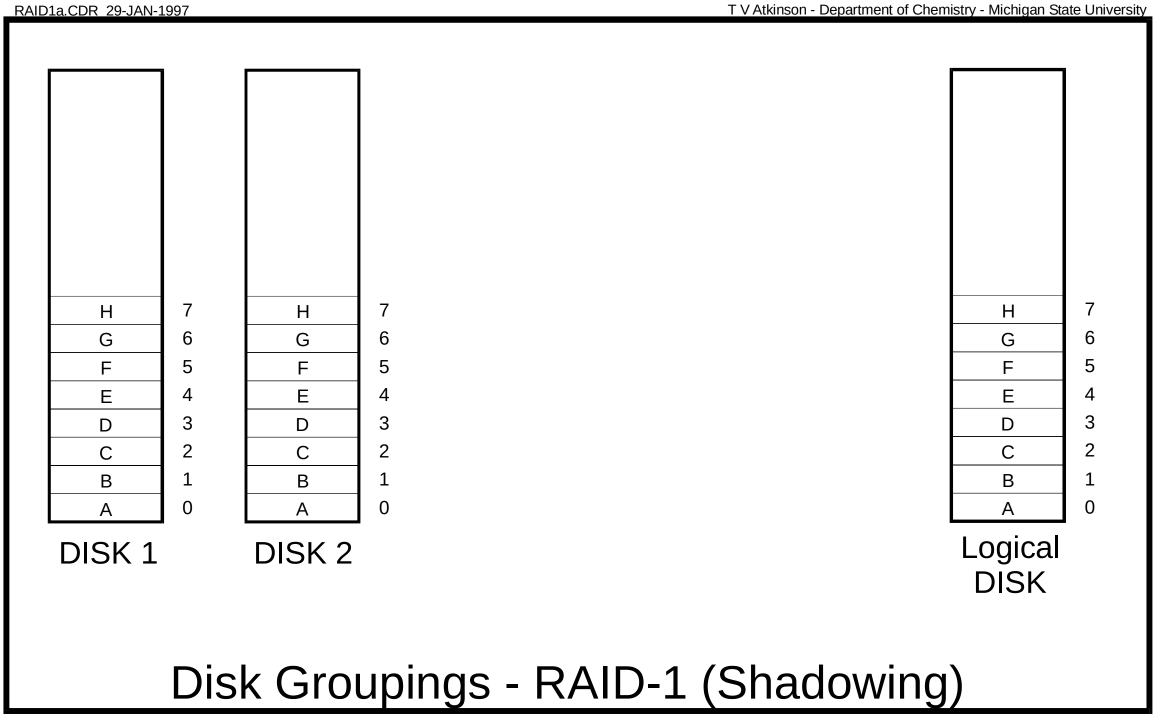


Figure 42 Disk Groupings - Raid 1

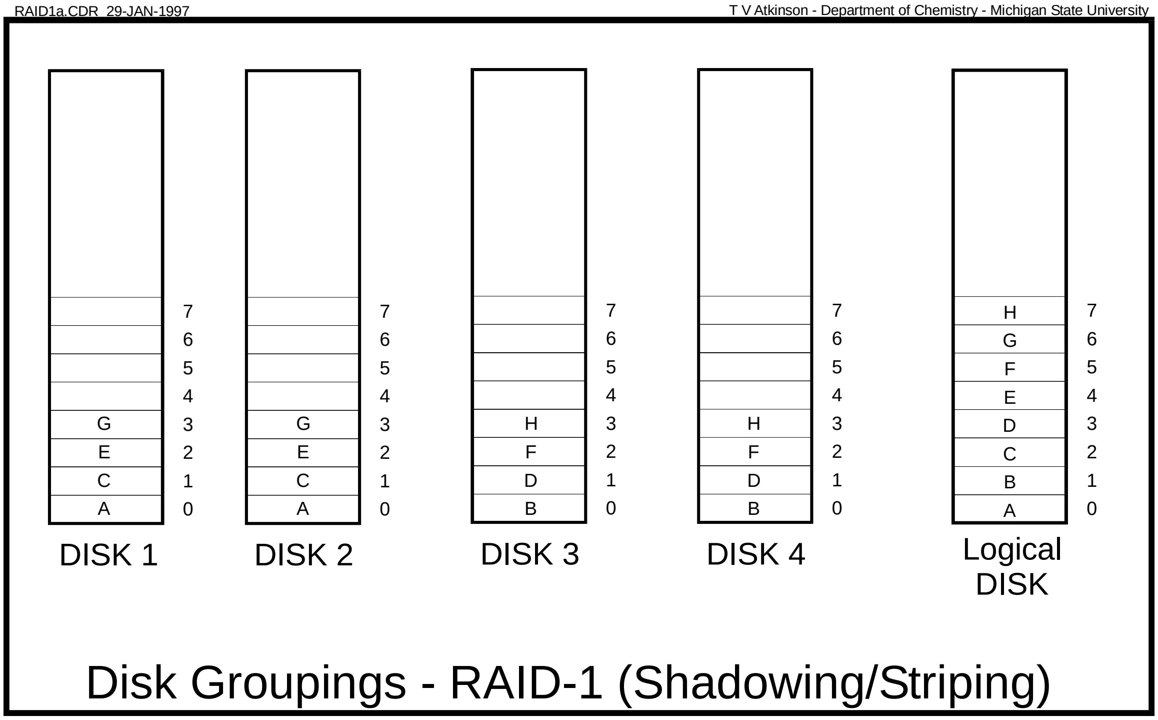

RAID1a.CDR 29-JAN-1997
T V Atkinson - Department of Chemistry - Michigan State University
G
E
C
A
7
6
5
4
3
2
1
0
DISK 1
G
E
C
A
DISK 2
H
F
D
B
DISK 3
H
F
D
B
DISK 4
H
G
F
E
D
C
B
A
Logical
DISK
Disk Groupings - RAID-1 (Shadowing/Striping)

Figure 43 Disk Groupings - Raid 1 Alternative

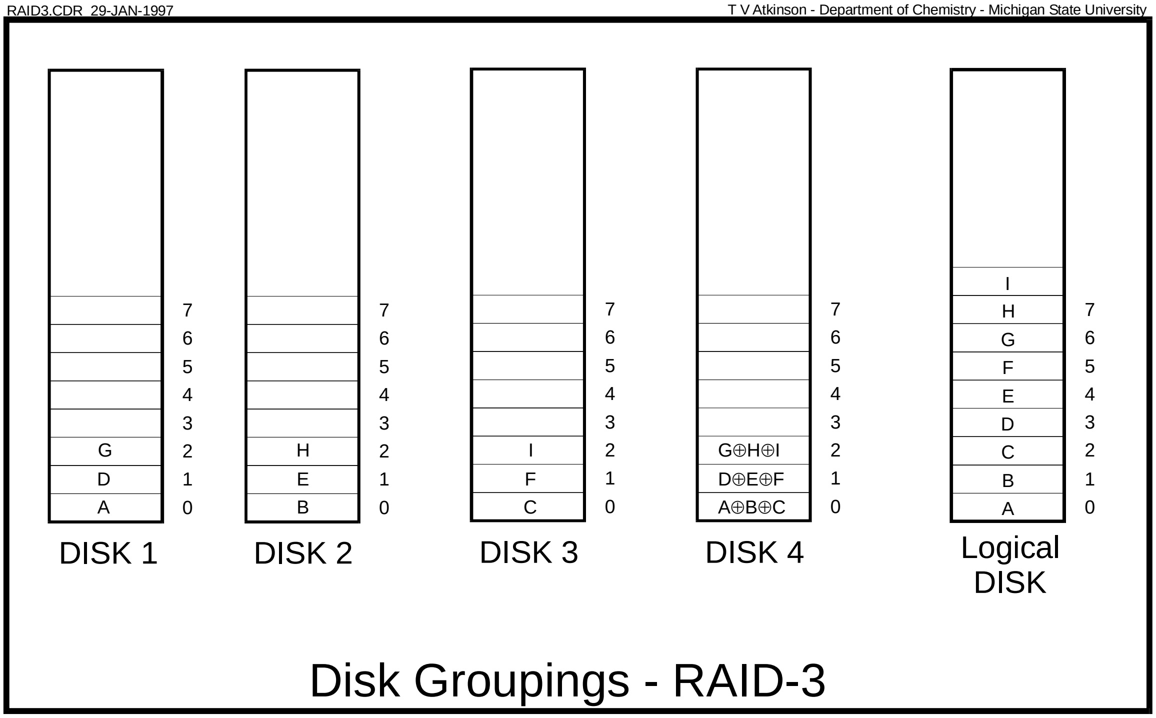


Figure 44 Disk Groupings - Raid 3

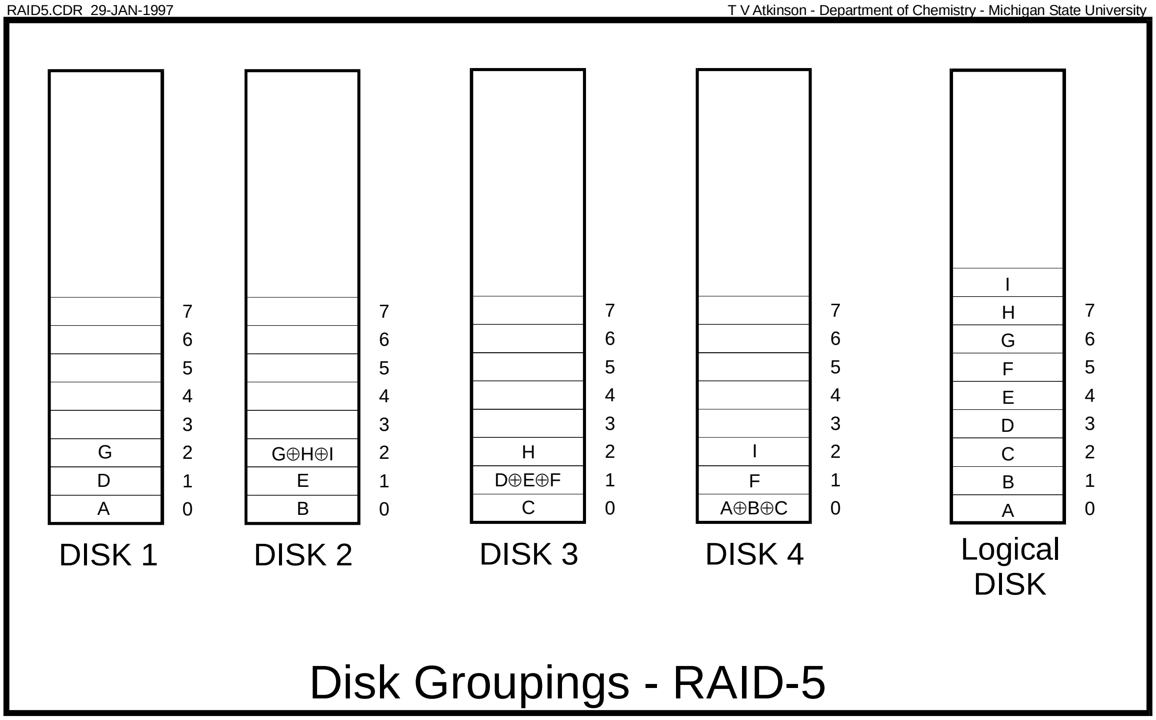


Figure 45 Disk Groupings - Raid 5

10. Memory Utilization

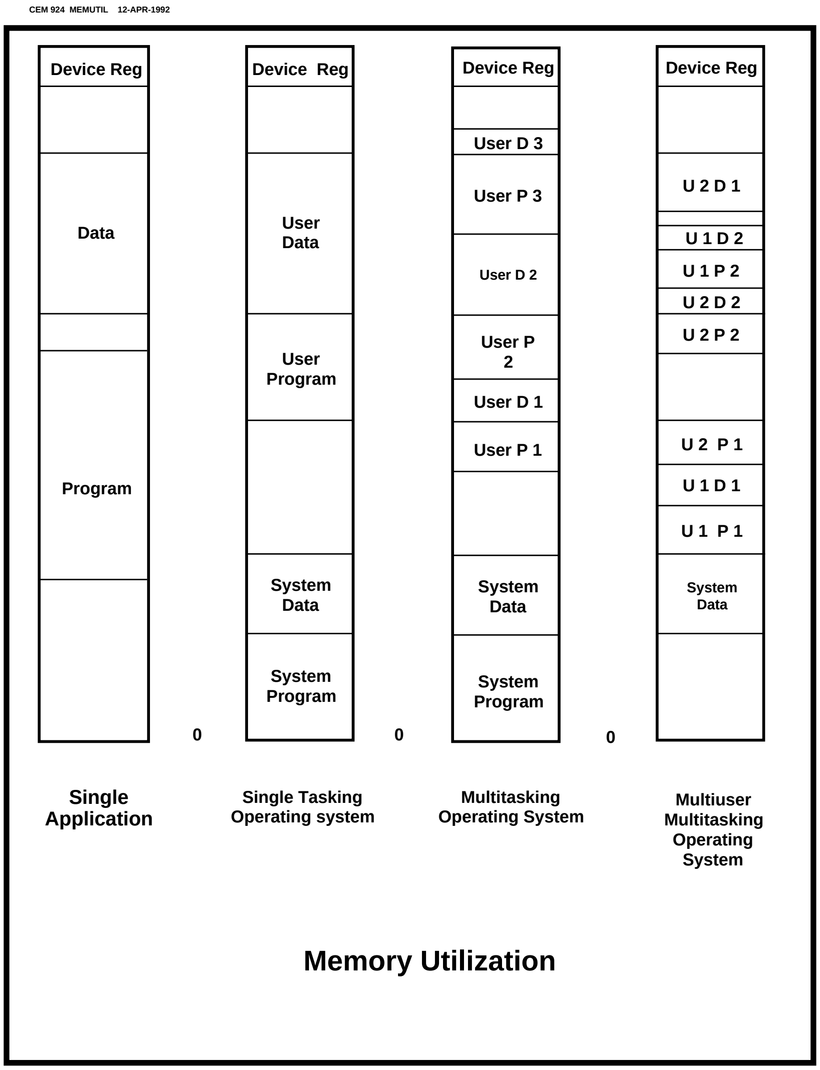
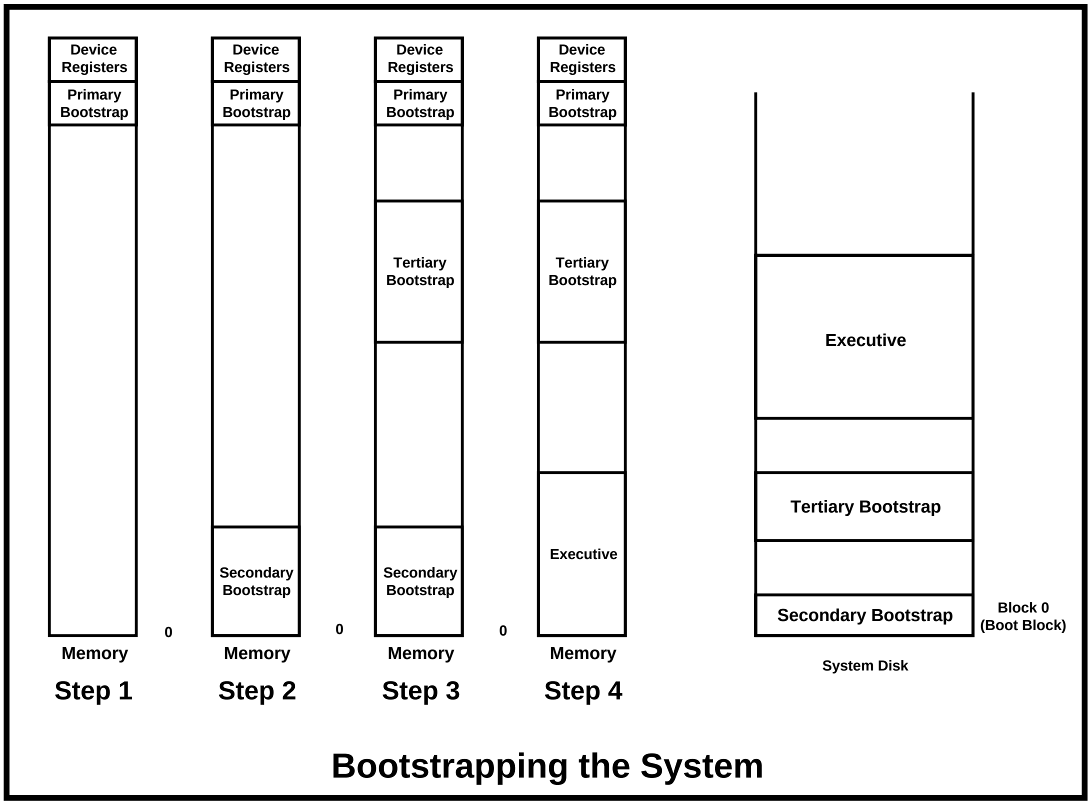


Figure 46 Memory Utilization

11. Boot Straps

CEM924 Boot1 13-APR-1992



Bootstrapping the System

Figure 47 Boot Strapping

11.1. Simple

1. Enter the Primary Bootstrap program into memory. (See Figure 47)
2. Enter the address of the entry point of the Primary Bootstrap Program into the PC.
3. Press Reset
4. Press Run
5. Primary Bootstrap reads the boot block of the system disk into memory, usually starting at address 0. (See Figure 47 Step 2)
6. Jump to the address of the entry point of the Secondary Bootstrap.
7. Secondary Bootstrap reads the Tertiary Bootstrap into a portion of memory that will not interfere with the loading of the Executive. (See Figure 47 Step 3)
8. Jump to the address of the entry point of the Tertiary Bootstrap.

9. Load the Executive into memory from the System Disk. (See Figure 47 Step 4)

10. Jump to the entry point of the Executive.

11. Executive will do further initialization of system data tables and load in any system overlays that are appropriate for this point of operation.

12. Executive executes any System Manager controlled startup scripts. These scripts perform additional initialization that is specific to the particular installation.

    12.1. PC/MS-DOS: \AUTOEXEC.BAT, \CONFIG.SYS

    12.2. VMS: SYS$MANAGER:SYSTARTUP_V5.COM

    12.3. SYS$STARTUP:SYLOGICALS.COM

    12.3. UNIX: /etc/rc*

13. Accept commands from the user(s)

11.2. Typical of Modern Machines with a Volatile Executive

CEM 924 18-JAN-1991 BOOT 3

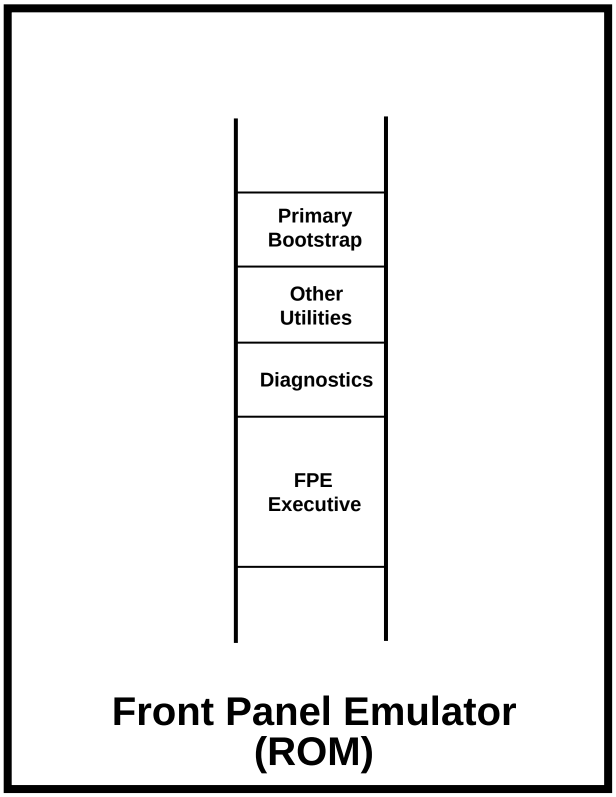


Figure 48 Front Panel Emulator

1. A simple resident Executive, Front Panel Emulator", which contains the Primary Bootstrap program is permanently installed in ROM in the CPU's address space. (See Figure 48)

2. Invoke the Primary Bootstrap by one of the following.

    2.1. Power Up

    2.2. Press Reset

    2.2. "Reset" The system (e.g. <CTRL><ALT><DEL>)

    2.3. Start execution at the entry point of the Primary Bootstrap (Not usually done)

3. Typically the Front Panel Emulator will include simple diagnostics that will be run at this point. These are programs that exercise the hardware and detect some forms of aberrant behavior. If errors are detected, the boot process stops.

4. In some cases, the Front Panel Emulator will engage in a dialog with the User at this point, allowing the running of additional diagnostics, disk formatting and/or other simple chores. The user may be able to specify which of several system disks will be booted in the next steps.

5. In some cases, a sniffer boot will occur. The Primary Bootstrap will try the following steps on each of a list of candidate system disks. As soon as one is found that has an intact boot block, the booting process will continue on that disk.

6. Primary Bootstrap reads the boot block of the system disk into memory, usually starting at address 0. (See Figure 49 Step 2)

7. Jump to the address of the entry point of the Secondary Bootstrap.

8. From this point on, everything proceeds as in the simple bootstrap.

11.3. Machines with a ROM based Operating System

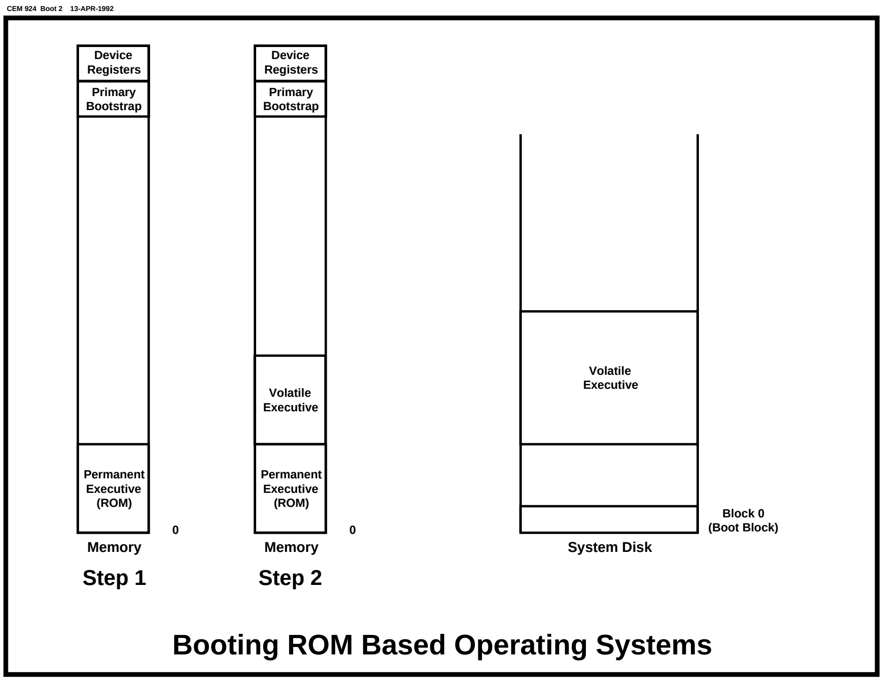


Figure 49 Booting a ROM Based OS

1. The Resident part of the Operating System Executive is permanently installed in ROM in the CPU's address space.

2. Invoke the Primary Bootstrap portion of the Resident Executive by one of the following.

   2.1. Power Up

   2.2. Press Reset

   2.2. "Reset" The system (e.g. <CTRL><ALT><DEL>)

   2.3. Start execution at the entry point of the Primary Bootstrap (Not usually done)

3. Typically simple diagnostics will be run at this point.

4. Executive will do further initialization of system data tables and load in any system overlays that are appropriate for this point of operation.

5. Executive executes any System Manager controlled startup scripts. These scripts perform additional initialization that is specific to the particular installation.

6. Accept commands from the user(s)

12. Memory Systems

Table 17 Powers of 2 (Abbreviated)

n	DEC	OCT	HEX	a.k.a.
10	1024	2000	400	1K
11	2048	4000	800	2K
12	4096	10000	1000	4K
13	8192	20000	2000	8K
14	16384	40000	4000	16K
15	32768	100000	8000	32K
16	65536	200000	10000	64K
17	131072	400000	20000	131K
18	262144	1000000	40000	256K
19	524288	2000000	80000	512K
20	1048576	4000000	100000	1M
21	2097152	1000000	200000	2M
22	4194304	20000000	400000	4M
23	8388608	40000000	800000	8M
24	16777216	100000000	1000000	16M
25	33554432	200000000	2000000	32M
26	67108864	400000000	4000000	64M
27	134217728	1000000000	8000000	128M
28	268435456	2000000000	10000000	256M
29	536870912	4000000000	20000000	512M
30	1073741824	10000000000	40000000	1G
31	2147483648	20000000000	80000000	2G
32	4294967296	40000000000	100000000	4G

Table 18 Representative Examples of DRAM Chips

Part Number	Num Bits	Org	Power Of 2	Pins	Data Lines	Add lines	Access	Data Book
MCM4027AC-2	4K	4Kx1	12	16	1/1	6	150	Motorola 1980
MCM4516C12	16K	16Kx1	14	16	1/1	7	120	Motorola 1980
MCM6632L15	32K	32Kx1	15	16	1/1	8	150	Motorola 1980
MCM6664L15	64K	64Kx1	16	16	1/1	8	150	Motorola 1980
HM48416A-12	64K	16Kx4	14	18	4	8	120	Hitachi 1984
HM50256-12	256K	256Kx1	18	16	1/1	9	120	Hitachi 1984
MCM511000A-70	1M	1Mx1	20	26	1	10	70	Motorola 1994
MCM514256A-70	1M	256Kx4	18	26	4	9	70	Motorola 1994
MCM44100B-60	4M	4Mx1	22	26	1	11	60	Motorola 1994
MCM44400B-60	4M	1Mx4	20	26	4	10	60	Motorola 1994
MCM54800A-70	4M	512Kx8	19	28	8	8	70	Motorola 1994
MCM54170B-70	4M	256Kx16	18	40	16	8	70	Motorola 1994
MCM516160A-60	16M	1Mx16	24	42	16	8	60	Motorola 1994
MCM54190B-70	16M	256Kx18	18	40	18	8	70	Motorola 1994
MCM516180A-60	16M	1Mx18	20	42	18	8	60	Motorola 1994
MCM516100-60	16M	16Mx1	24	28	1/1	12	60	Motorola 1994
MCM516400-60	16M	4Mx4	22	28	4	10	60	Motorola 1994

Table 19 Representative Examples of SIMMS

									Main Chip			Second Chip	
Part Number	Num Bits	Org	Pins	Size	Style	# Chips	Data Lines	Add lines	Num	Chip	Org	Num	Org
MCM81000	8M	1Mx8	30x1		SIMM	8	8	10	8	511000	1Mx1		
MCM81430	8M	1Mx8	30x1		SIMM	2	8	10	2	54400	1Mx4		
MCM84000	32M	4Mx8	30x1		SIMM	8	8	11	8	54100A	4Mx1		
MCM81600	64M	16Mx8	30x1		SIMM	4	8	12	4	517400	16Mx1		
MCM91000	9M	1Mx9	30x1		SIMM	9	8	10	9	511000	1Mx1		
MCM91430	9M	1Mx9	30x1		SIMM	3	8	10	2	54400AN	1Mx4	1	1Mx1
MCM94000	36M	4Mx9	30x1		SIMM	9	9	10	9	54100A	4Mx1		
MCM91600	144M	16Mx9	30x1		SIMM	9	9	12	9	517400	16Mx1		
MCM32100	32M	1Mx32	72x2	S	DIMM	8	32	10	8	54400AN	1Mx4		
MCM32130	32M	1Mx32	72x1	L	SIMM	8	32	10	8	54400	1Mx4		
MCM32230	64M	2Mx32	72x1	L	SIMM	8	32	10	8	54400	1Mx4		
MCM32400	128M	4Mx32	72x1	L	SIMM	8	32	11	8	517400	16Mx1		
MCM32400D	128M	4Mx32	72x2	S	DIMM	8	32	12	8	516400	4Mx4		
MCM32800	256M	8Mx32	72x1	L	SIMM	8	32	11	8	517400	16Mx1		
MCM36100	36M	1Mx36	72x1	L	SIMM	12	36	10	8	54400	1Mx4	4	1Mx1
MCM36104	36M	1Mx36	72x1	L	SIMM	9	36	10	9	54400	1Mx4		
MCM36200	72M	2Mx36	72x1	L	SIMM	24	36	10	16	54400	1Mx4	8	1Mx1
MCM36204	72M	2Mx36	72x1	L	SIMM	18	36	10	18	54400	1Mx4		
MCM36400	144M	4Mx36	72x1	L	SIMM	12	36	10	8	54400	1Mx4	4	4Mx1
MCM36800	258M	8Mx36	72x1	L	SIMM	24	36	11	16	517400	4Mx4	8	4Mx1
MCM40100	40M	1Mx40	72x1	L	SIMM	10	40	10	10	54400	4Mx4		
MCM40200	80M	2Mx40	72x1	L	SIMM	20	40	10	20	54400	4Mx4		
MCM40400	160M	4Mx40	72x1	L	SIMM	10	40	11	10	517400	4Mx4		
MCM40800	320M	8Mx40	72x1	L	SIMM	20	40	11	20	517400	4Mx4		
MCM64100	64M	1Mx64	84x2		DIMM	16	64	11	16	54400	1Mx4		
MCM64400	128M	4Mx64	84x2		DIMM	16	71(64)	12	16	516400	4Mx4		
WPD8M72	256M	8Mx72	84x2		DIMM	36	72	12	36		4Mx4		

13. Increasing Performance

1. Improve program (operating system or application)

2. Improve physical implementation (CPU, Memory, and/or Peripherals). This would entail rebuilding the same architecture with faster components. For instance, using transistors rather than vacuum tubes, TTL rather than RTL, ECL rather than TTL, or just be more careful so that things can run faster without errors.

3. Improve architecture (hardware or software)

4. Add concurrency (actually an example of above)

    4.1. Pipe lining (instruction fetch-decode-execution, floating point operations, vector operations)

    4.2. Branch prediction (to optimize instruction pre-fetch)

    4.2. Cache (Memory, instruction, data, and disk)

    4.3. Memory Interleaving

    4.4. Disk interleaving (sector interleaving, track skewing)

    4.5. Parallel Processing

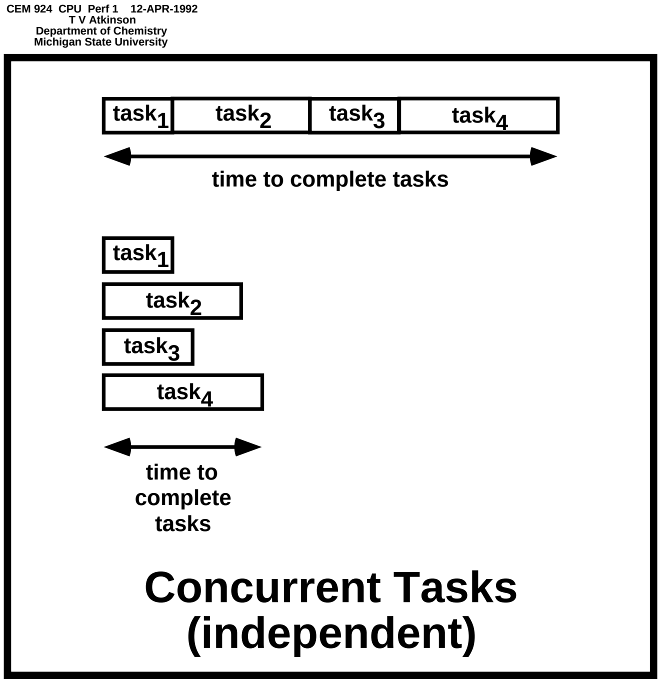
    4.6. Coprocessors - math, graphics, vector, array

    4.7. Multiple CPU's

    4.8. DMA device controllers

13.1. Concurrent Tasks

13.1.1. Tasks are completely independent.

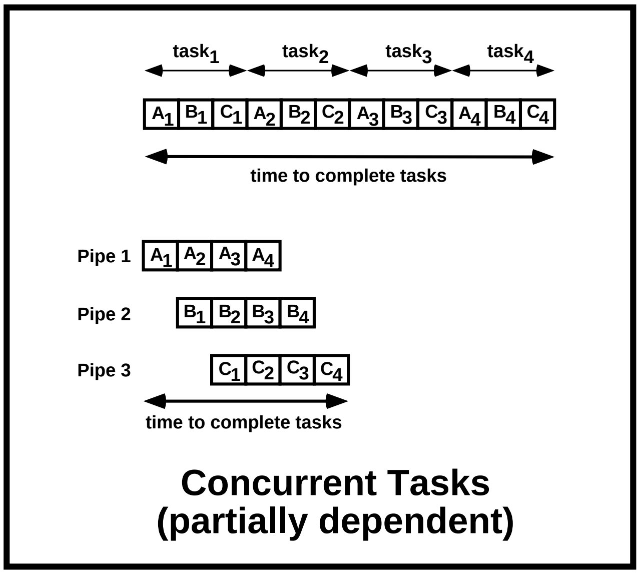

CEM 924 CPU Perf 1 12-APR-1992
T V Atkinson
Department of Chemistry
Michigan State University

Concurrent Tasks (independent)

Figure 50 Concurrent Tasks

13.1.2. Pipelines (Tasks are somewhat independent)

CEM 924 CPU Perf 2 12-APR-1992
T V Atkinson
Department of Chemistry
Michigan State University

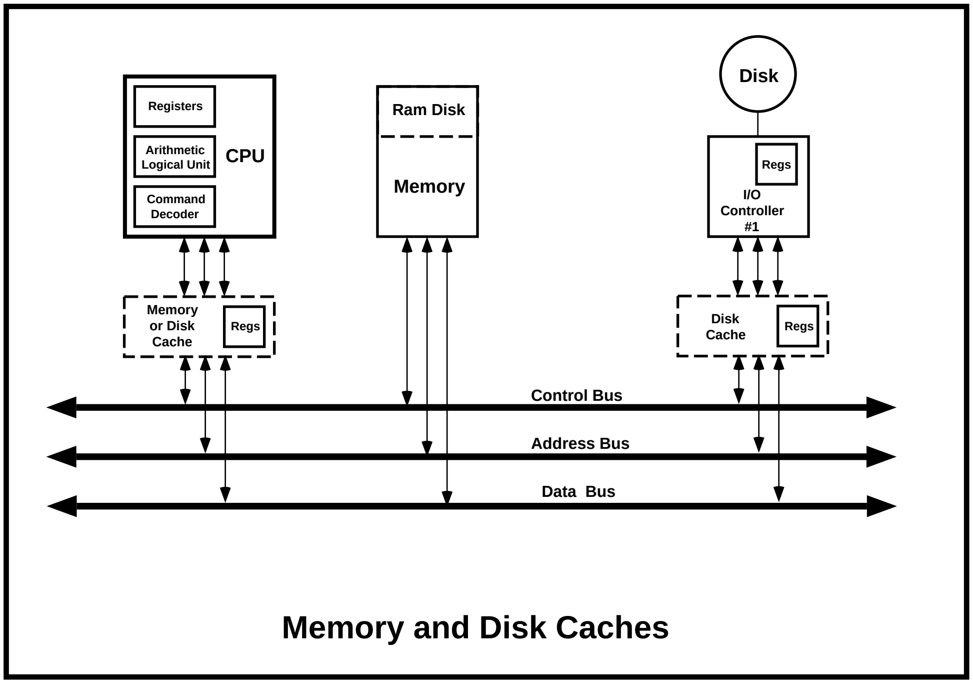


Concurrent Tasks
(partially dependent)

Figure 51 Concurrent Tasks (Partial Dependence)

13.1.3. Cache

CEM 924 Cache 1 13-APR-1992
T V Atkinson
Department of Chemistry
Michigan State University



Memory and Disk Caches

Figure 52 Cache and RAM Disk

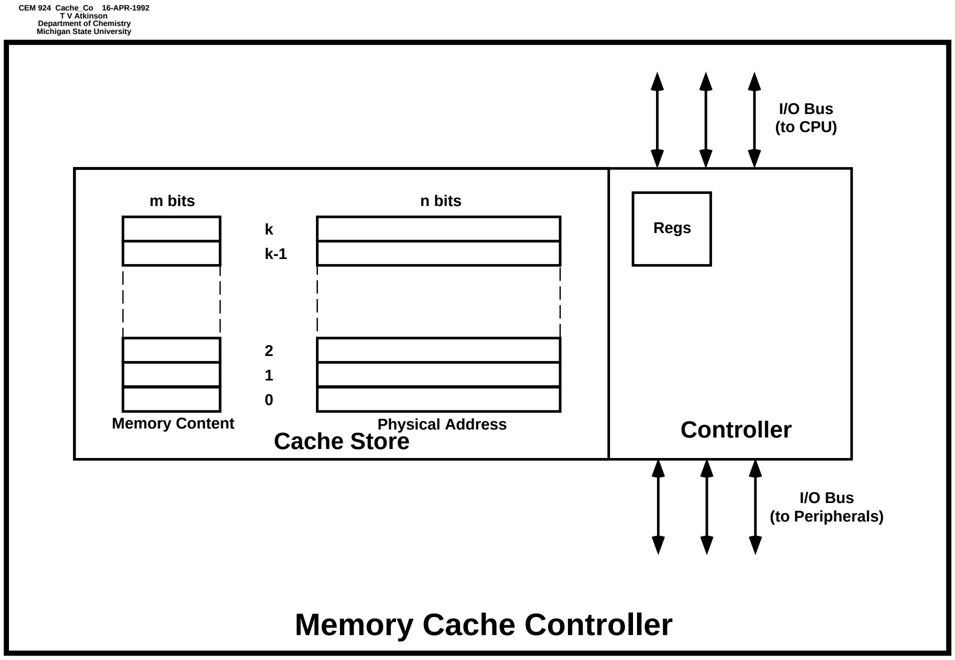


Figure 53 Memory Cache Controller

The following examples describe the operation of a generalized memory cache. In these examples, A, B, C, D, E, and F symbolically refer to specific locations within the physical memory space of the system.

1. CPU issues a read instruction to fetch the contents of memory location A. A copy of A is not currently being held in the cache. The cache store is not full.

   1.1. Cache controller passes reference through to memory subsystem.

   1.2. Memory returns the contents of memory location A to the cache controller.

   1.3. Cache controller passes the contents of memory location A onto the CPU.

   1.4. Cache controller stores the address of A and the contents of A in the cache store.

2. CPU issues a read instruction to fetch the current contents of memory location B. A copy of B is currently being held in the cache. For this case, whether the cache is full or empty has no effect.

   2.1. Cache controller returns the contents of Memory Location B to the CPU using the copy located in the cache store.

3. CPU issues a read instruction to fetch the contents of memory location C. A copy of C is not currently being held in the cache. The cache store is full.

   3.1. Cache controller decides which of the existing set of copies of memory locations to discard in order to create space for the new reference.

   3.2. Cache controller passes the reference through to memory subsystem.

   3.3. Memory returns the contents of memory location C to cache controller.

   3.4. Cache controller passes the contents of memory location C onto the CPU.

   3.5. Cache controller stores the address of C and the contents of memory location C in the cache store in the newly freed slot.

4. CPU issues an instruction to write a new value into memory location D. A copy of D is currently not being held in the cache. The cache store is not full.

   4.1. Cache controller stores the address of D and the new contents of memory location D in the cache store.

   4.2. Cache controller passes the address of memory location D and the new contents of memory location D to the memory subsystem which stores the new value into location D.

5. CPU issues an instruction to write a new value into memory location E. A copy of E is not currently being held in the cache. The cache store is full.

   5.1. Cache controller decides which of the existing set of copies of memory locations to discard from cache store.

   5.2. Cache controller stores the address of memory location E and the contents of memory location E in the cache store in the newly freed slot.

   5.3. Cache controller passes the address of memory location E and the new contents of memory location E to the memory subsystem which stores the new value into memory location E.

6. CPU issues an instruction to write a new value into memory location F. A copy of memory location F is currently being held in the cache. For this case, whether the cache is full or empty has no effect.

   6.1. Cache controller stores the address of F and the contents of memory location F in the cache store in an empty slot.

6.2. Cache controller passes the address of memory location F and the new contents for memory location F to the memory subsystem which stores the new value into memory location F.

13.1.4. Direct Memory Access (DMA)

DMA is an example of asymmetrical parallel processing where the disk controller is doing work while the CPU is doing other tasks. The device controller has enough intelligence to manage the transfer of information to (from) memory from (to) a disk drive once the transfer has been set up by the CPU (i. e. software). This section is a simplified example of such a device. To begin, however, the section illustrates the simpler, non-parallel programmed I/O technique of controlling a device such as a disk. In addition, this section will investigate some simple examples of interrupt structures, a necessary part of DMA operations.

13.1.4.1. Programmed I/O (Example: reading a block of data)

CEM 924 IO 1 12-APR-1992
T V Atkinson
Department of Chemistry
Michigan State University

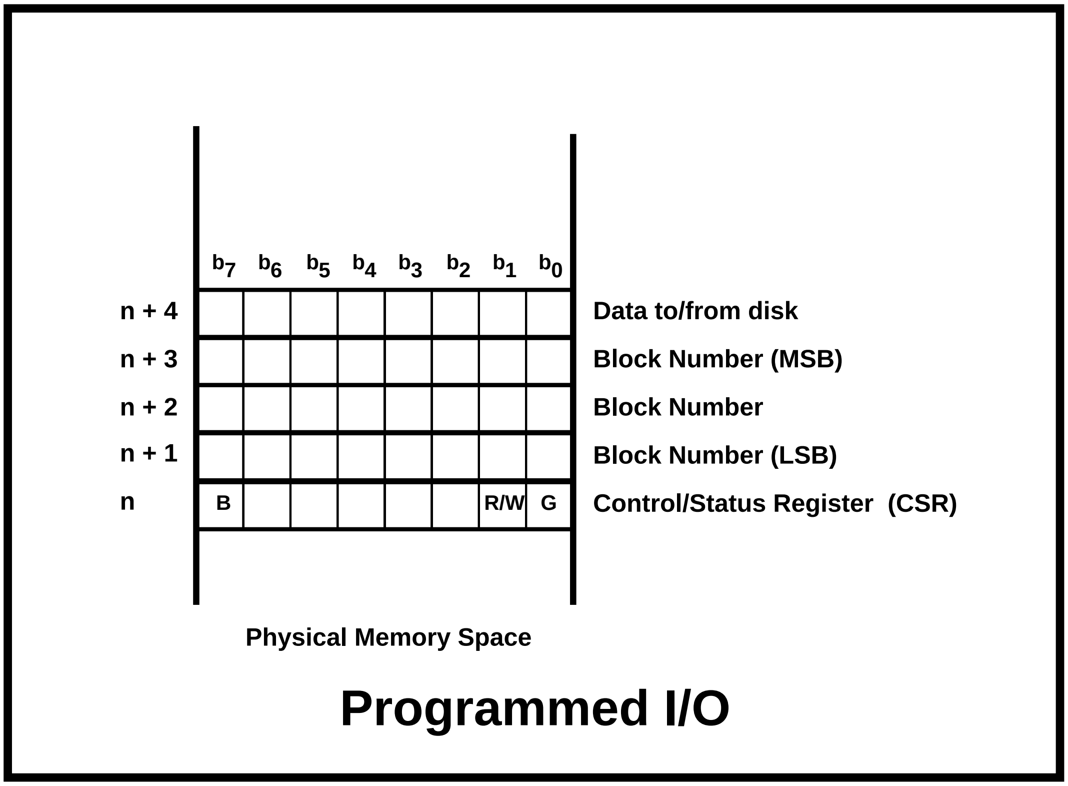


Figure 54 Program I/O

Figure 54 is a programmer's model of a simple interface to a disk. In this example "G" is the go bit [=0 stop disk controller, =1 find block and begin transfer], "R/W" is the direction bit [=0 read (information is transferred from disk to cpu), =1 write (information is transferred from the CPU to the disk)], "B" is the busy bit [=1 busy (next byte is being sought), =0 not busy (byte is ready to be transferred from controller to CPU), and "n" is the base address of the register set for the interface.

1. Write the number of the block on the disk to be fetched into the Block Number registers (n+1), (n+2), (n+3). Using three successive writes to the three byte registers will do this.

2. Write a "1" into the GO bit and a "0" into the R/W bit

3. LOOP: Read CSR(n)

4. IF CSR is negative, go to LOOP

5. Read the Data Register (n+4); Get next byte from disk

6. Put the byte away in memory

7. If there are more bytes to get, go to LOOP

8. If there are no more bytes to get, Write a "0" into GO bit of CSR (n); Stop the controller

13.1.4.2. Asynchronous I/O (Interrupt Structures)

As the next section will illustrate, if two independent devices, e. g. the CPU and the I/O device, are to operate asynchronously there must be mechanisms for the two entities to signal each other when necessary. The spinning-on-a-bit technique illustrated in the laser experiment is one example. Spinning-on-a-bit is very simple but inefficient; the CPU does nothing but watch the flag bit, waiting for the device, in that case the ADC, to finish.

The ability of an I/O device to interrupt the processing of the CPU provides another way for this necessary signaling to occur. In such cases, the program running in the CPU sets up the I/O controller for the I/O operation. The program then gives the I/O controller the command to begin the operation. At this point, the CPU is no longer needed and may proceed with other processing. The I/O device continues asynchronously until the operation I/O is completed. Upon completion, the I/O device must signal the program, so that appropriate actions may be taken, for example set up the next I/O operation. To achieve this signaling the I/O device “pulls an interrupt.” This section examines simplified versions of two strategies of doing interrupts.

13.1.4.2.1. Interrupt Structure 1

Figure xx illustrates this technique. Both the CPU and the I/O controllers have additional logic to implement the interrupt structure. Two explicit signals, **Interrupt Request** and **Interrupt Grant** are added to the Control Bus. Notice that **Interrupt Request** is a single signal bus, but **Interrupt Grant** is actually a “daisy chain,” the signal is generated in the CPU and sent to the first device which then has to repeat the process and sent the signal to the second device.

The main program sets up the I/O device for the desired operation, e. g. read a particular block, or write a block. The last step of this part of the operation is to enable the I/O device to do interrupts.

When the I/O operation is complete, the I/O device asserts **Interrupt Request**. The CPU interrupt handler determines if CPU interrupts are enabled. If so, it interrupts CPU operation at the end of the next instruction. If not, it waits until the current program decides to allow interrupts and enables them.

The CPU saves the current context, every thing that defines the current state of the program, in memory. At the minimum, the contents of the PC must be saved so that the program can be restarted later. Other registers may also be saved at this point. This is all achieved by logic within the CPU.

The CPU then asserts **Interrupt Grant**. This signal is passed down the daisy chain until reaching the first device that has an interrupt pending. That device does not pass the signal down the daisy chain. Thus priority of interrupt service is determined by the place on the bus.

The I/O controller interrupt handler puts an interrupt service address unique to that device onto the Address Bus. This interrupt service address has been “hard wired” into the I/O controller at the time of installation, often with jumpers.

The CPU interrupt handler reads the interrupt service address and get the contents of the memory location with that address and loads that number into the PC. In simpler terms, CPU program execution jumps to the location of the Interrupt Service Routine for that specific I/O device.

The Interrupt Service Routine saves any additional context of the interrupted program and then does any processing that is appropriate at this point for the I/O device. At some point the Interrupt Service Routine will reset the I/O controller and thus, clear the interrupt. When done the Interrupt Service Routine restores any of the context of the interrupted program and executes an interrupt return.

The CPU finishes restoring the context of the interrupted program. The last step of this is loading the PC with the address of the next instruction of the interrupted program. Execution then resumes.

CEM 924 IO 2 12-APR-1992
T V Atkinson
Department of Chemistry
Michigan State University

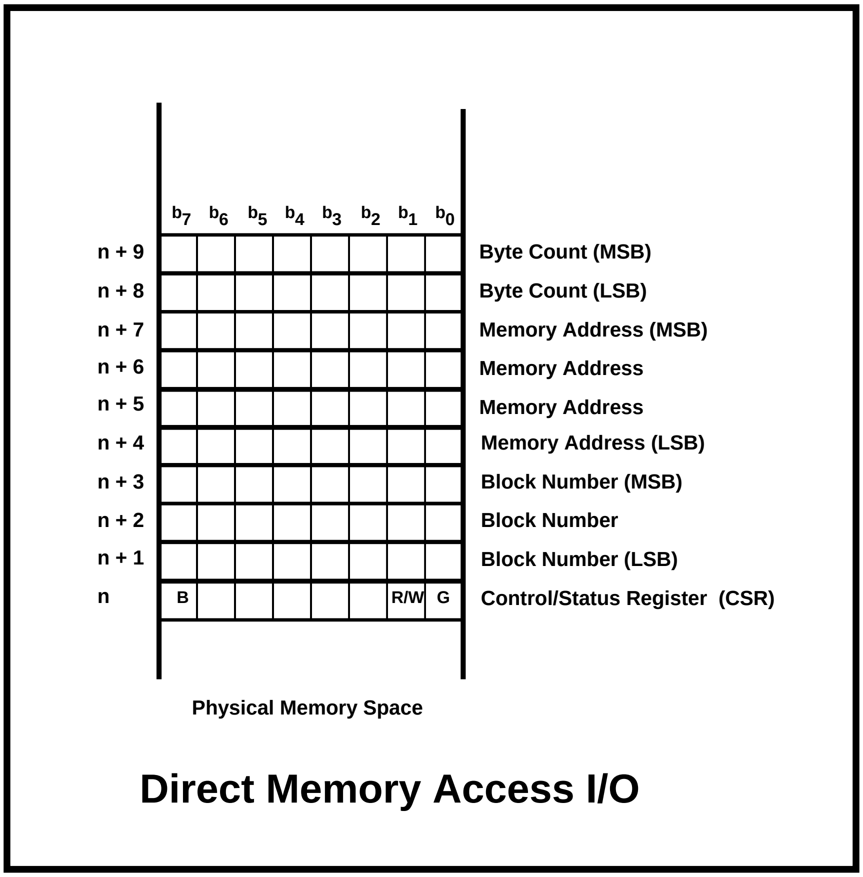


Figure 55 DMA Example

13.1.4.3. DMA I/O (Example: Write a block to disk)

1. Write the number of the block on the disk that will receive the information into the Block Number registers (n+1), (n+2), (n+3).

2. Write the physical memory address of the information to be written onto the disk into the memory address register registers (n+4), (n+5), (n+6), (n+7).

3. Write the number of the bytes to be transferred into the Byte Count Register (n+8), (n+9).

4. Write a "1" into the GO bit and a "1" into the R/W bit

5. Continue the program from this point

6. When the block has been transferred, the disk controller will signal (interrupt) the CPU and the CPU will stop executing the current program and execute any code that is required finish the transfer and then resume processing the interrupted program at the point of the interruption.

14. Memory Management

14.1. Introduction

The matching of the size of a program and the size of the available memory has always been an important concern. Often the logical image of a program is larger than the amount of physical memory that is available to contain it (See Figure 56). Several factors govern the amount of physical memory available to contain a given program.

1. Size of the CPU memory address space as defined by the size of the Program Counter (PC) Register.

2. Size of the physical memory address space as defined by the size of the memory address bus. The size of the physical memory address space can be less than, equal to, or greater than the size of the CPU memory address space. That is, the number of address signals constituting the Address Bus can be smaller than, equal to, or larger than the number of bits in the PC. In addition, the actual amount of physical memory can be less than or equal to the size of the physical memory address space. The actual amount of physical memory can be less than, equal to, or greater than the size of the CPU memory address space.

3. The amount of physical memory required by the operating system.

4. The amount of physical memory that any given process can expect to enjoy in a multitasking/user environment where resources are divided among the various tasks and/or users.

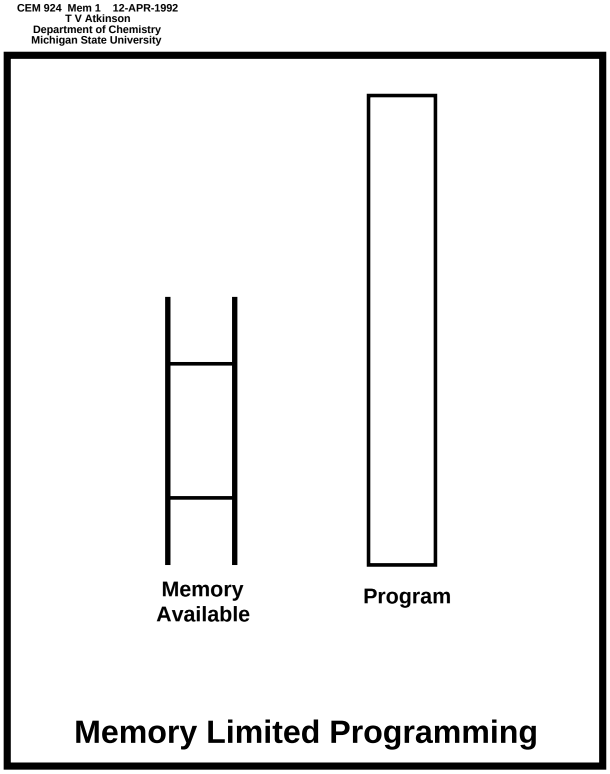

CEM 924 Mem 1 12-APR-1992
T V Atkinson
Department of Chemistry
Michigan State University
Memory
Available
Program
Memory Limited Programming

Figure 56 Program Exceeds Memory Available

A number of possible solutions exist to the problem of a program, code and data, being larger than the memory available to run it.

1. Trivial (from a programmer's point of view)

   1.1. Buy more memory

   1.2. Buy a new computer with a larger address space

2. Software solutions

   2.1. Rewrite the program to reduce the size.

   2.2. Chain

   2.3. Overlay

3. Hardware/Software Solutions (Address translations)

   3.1. Bank switching

   3.2. Segmentation

3.3. Paging

3.4. Virtual memory

14.2. Motivations for Memory Management

1. Expand CPU address space
2. Facilitates flexible assignment of memory to process(s). Allows segmentation of a process.
3. Assists in having multiple tasks in memory (multitasking)
4. Protection of one task from another
5. Allows sharing of data and code among tasks.
6. Augments virtual memory implementation.

14.3. Software solutions

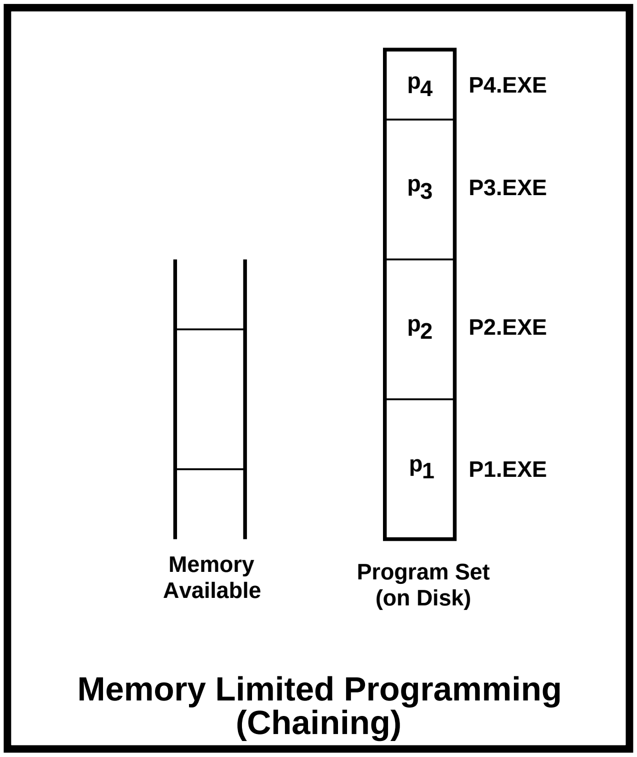
14.3.1. Chaining

This approach requires that the original program be subdivided into a number of smaller stand-alone programs, each of which is small enough to fit into the available physical memory (See Figure 57). Operationally, the user invokes p_1. At the end of the execution of p_1, p_2 is invoked either manually or automatically if the operating system allows. At the end of the execution of p_2, p_3 is invoked, etc. Each stand alone program segment is located in a separate disk file.

Communication among the programs is achieved via reading and writing disk files or, perhaps, by sharing a section of common physical memory.

Gaussian 86 is an example of such a program. Advantages of this approach: Very large programs can be built. Disadvantages: More work for the programmer. Some programs may not be easily segmented.

CEM 924 Mem 2 12-APR-1992
T V Atkinson
Department of Chemistry
Michigan State University

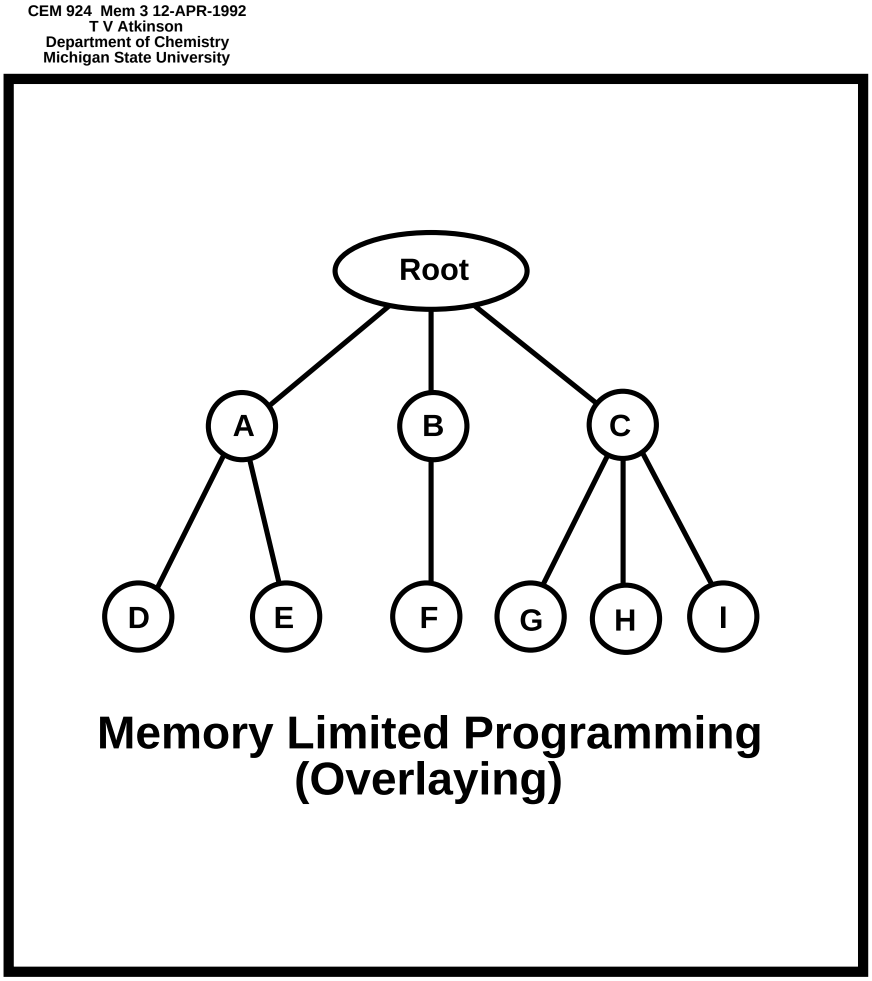


Memory Limited Programming (Chaining)

Figure 57 Memory Limited Programming (Chaining)

14.3.2. Overlaying

The program is again segmented (See Figure 59), but this time into a set of hierarchical subroutines as shown below (See Figure 58). In this three layer example ROOT calls A, B, and C. A calls D and E, etc. When the executing program requires a particular module, a subroutine call is made for that module. The operating system determines if that module is already in memory. If so, execution immediately branches to the entry point of that module. If not, the operating system reads that module from disk into the appropriate segment in physical memory. Execution then branches to the entry point of the newly loaded module. All program segments are located in a single disk file.



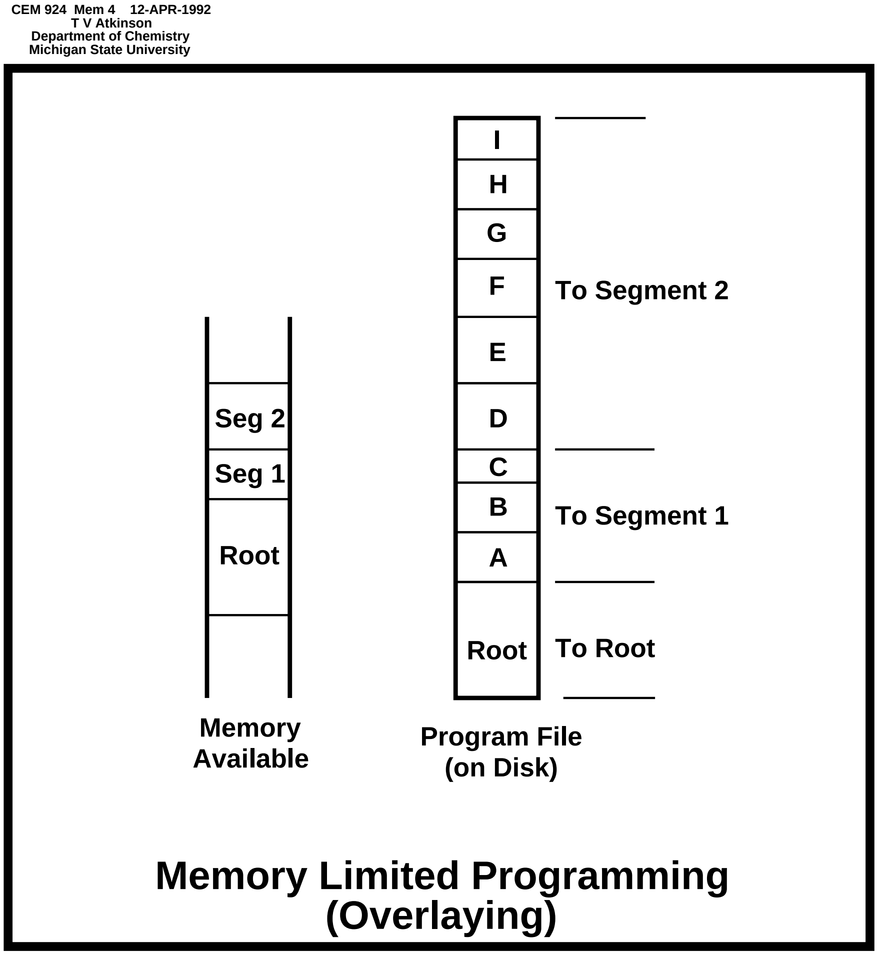
CEM 924 Mem 3 12-APR-1992
T V Atkinson
Department of Chemistry
Michigan State University

Memory Limited Programming (Overlaying)

Figure 58 Memory Limited Programming (Overlaying)

At any one time memory contains one of the following combinations.

1. ROOT, A, D
2. ROOT, A, E
3. ROOT, B, F
4. ROOT, C, G
5. ROOT, C, H
6. ROOT, C, I



CEM 924 Mem 4 12-APR-1992
T V Atkinson
Department of Chemistry
Michigan State University

Memory Limited Programming
(Overlaying)

Figure 59 Overlaying (Memory Layout)

14.4. Hardware/Software Solutions

The introduction of an additional hardware sub-system, i.e. memory management unit (MMU) (See Figure 60), allows various hardware approaches to solving the problem of memory space. In addition, other facilities such as memory protection and virtual memory can be included. The MMU will consist of a set of registers that are accessible to the CPU and some logic. The MMU translates the memory addresses output from the CPU (logical addresses) during the instruction and operand fetches stages of instruction execution into the addresses that are actually placed on the memory address bus (physical addresses).

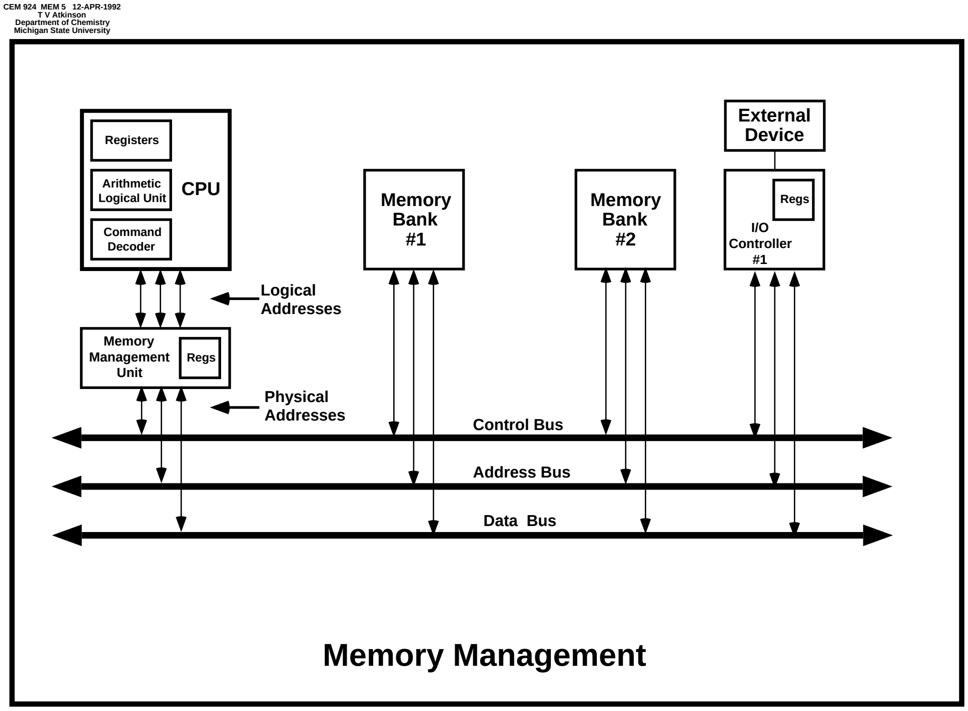


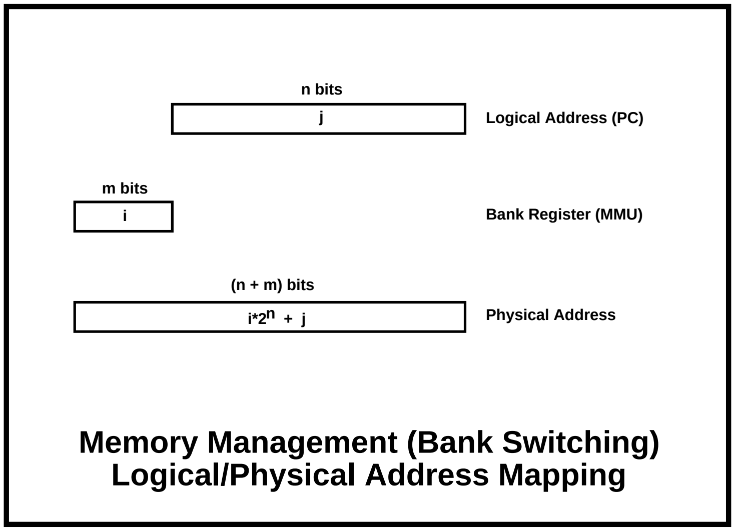
Figure 60 Memory Management

14.4.1. Bank Switching

In this type of implementation, the physical address space is divided into a group of equal size banks (See Figure 62). The MMU contains an m bit Bank Register (See Figure 61). The contents of this Bank Register is concatenated to the left side of the logical address to produce a (m+n) bit physical address, [i:j] or $i*2^n + j$.

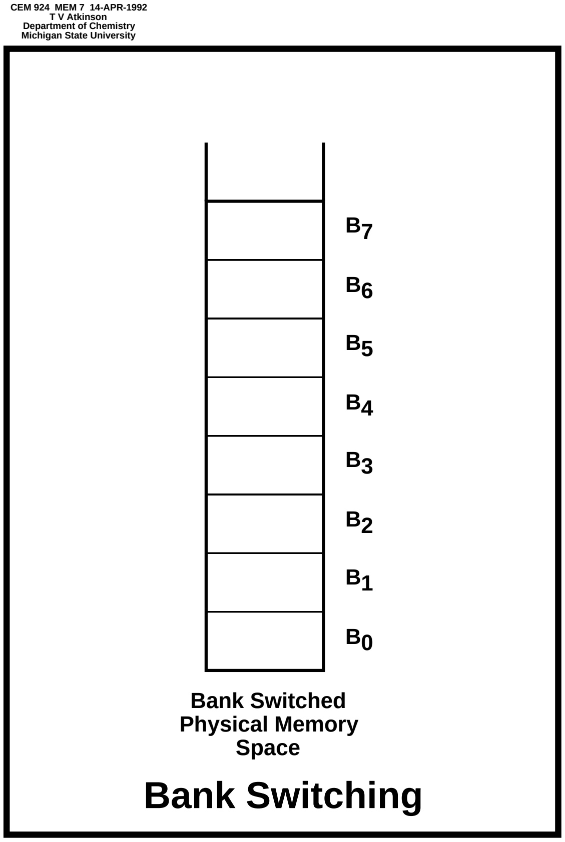
As an example, if n = 16 and m = 4, the logical (CPU) address space would be 65536. With the bank switching, the physical addresses on the memory address bus can now be 20 bits and support a physical memory address space of 1048576. At any one time, the program is operating in one of the sixteen physical banks of memory ($B_0, \dots, B_{15}$) of length 65536. The program switches between the banks by changing the contents of the Bank Register in the MMU. Changing the Bank Register will typically take several instruction times to affect. An enhancement of this approach would be to have two Bank Registers in the MMU. One would be used to map addresses of instruction (code). The second would be used to map addresses of data. This would allow the program to split the code and data into separate banks. In all cases, overhead would be required to switch the bank registers from one bank to another.

CEM 924 MEM 6 12-APR-1992
T V Atkinson
Department of Chemistry
Michigan State University



Memory Management (Bank Switching)
Logical/Physical Address Mapping

Figure 61 Bank Switching: Mapping



CEM 924 MEM 7 14-APR-1992
T V Atkinson
Department of Chemistry
Michigan State University

Bank Switching

Figure 62 Bank Switching: Memory Space

14.4.2. Bank Switching (Partial)

A more useful approach is to bank switch only a portion of the CPU memory space (See Figure 64). As before, multiple banks of physical memory of equal size are switched in and out of the CPU memory space. The physical banks of memory are all, in turn, switched into the same window of addresses within the CPU logical address space. In this type implementation, there are two types of memory systems, regular and bank switched, found on the bus as illustrated in Figure 63.

CEM 924 MEM 5a 18-APR-1992
T V Atkinson
Department of Chemistry
Michigan State University

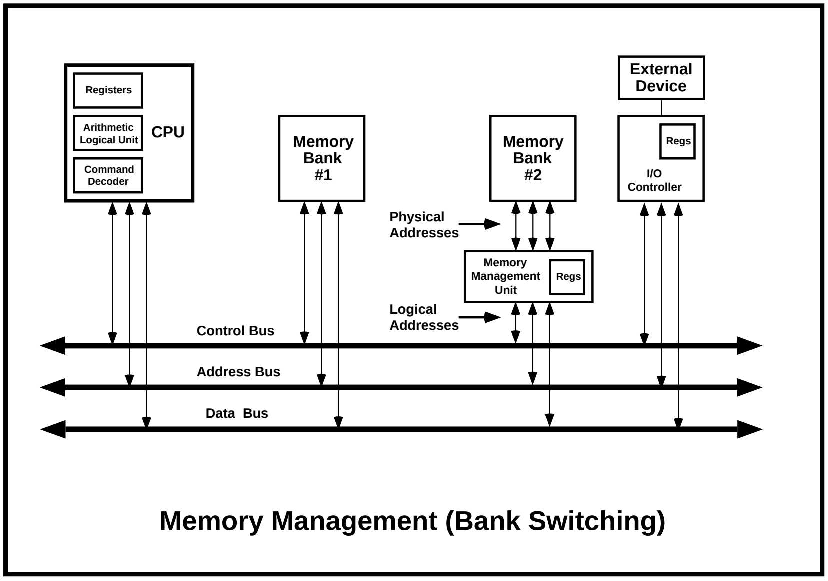


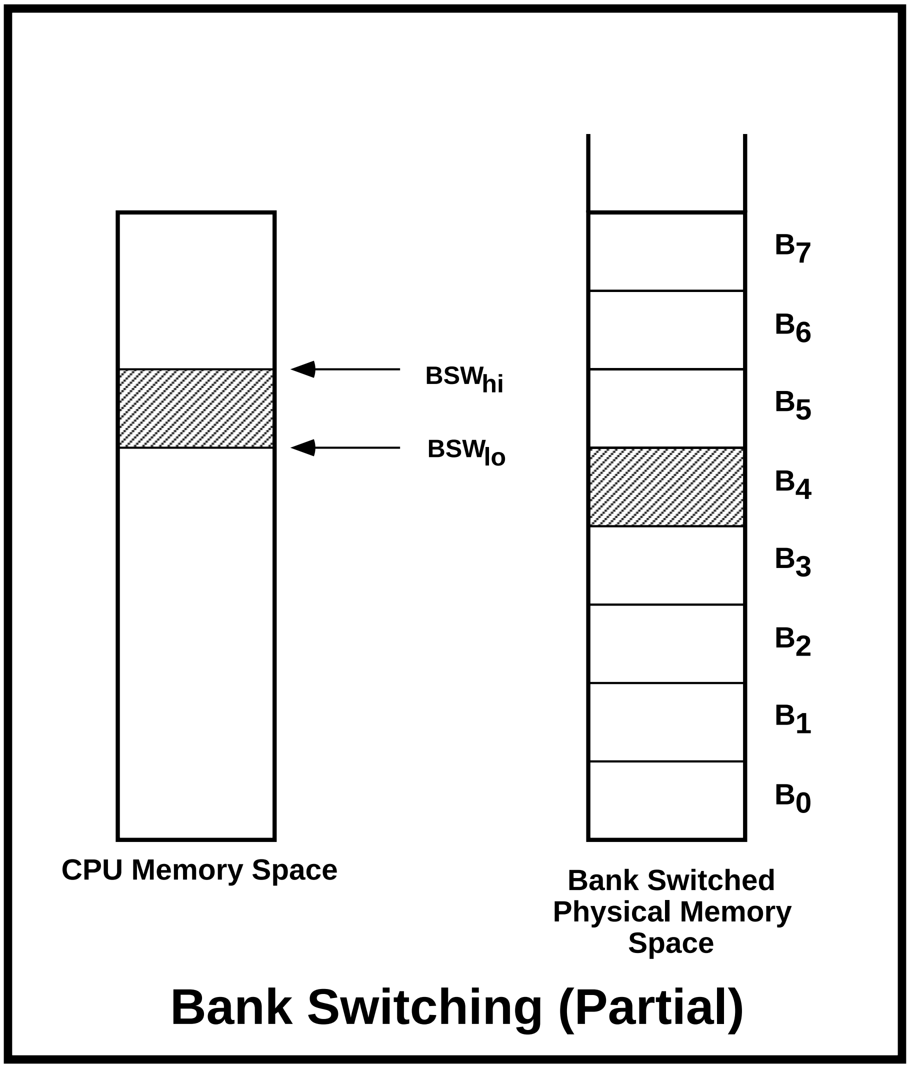
Figure 63 Bank Switching (Partial)

The bank switched memory subsystem has an internal memory address bus that is isolated from the regular address bus by the MMU. The CPU memory space is defined by the size of the PC register (n bits, n = k + l). The regular address bus is n bits wide. The Bank Switched memory subsystem address bus is m + l bits wide. A portion of the CPU address space between BSW_{lo}, and BSW_{hi} is set aside to receive the bank switched memory. Therefore, the regular memory system must not answer to addresses with in this range. Typically, the bank switched window will be set to be a integral power of two memory locations wide and the boundaries will also be integral powers of two. For example:

$$BSW_{hi} - BSW_{lo} = 2^{l}$$

The MMU contains a m bit wide register that is a regular I/O device register, i.e. the CPU can read/write numbers into it. During operation, the MMU takes each logical address, [i:j], on the regular address bus and partitions it into the most significant k bits, CPU Space window number, and the least significant l bits (See Figure 65). The CPU Space Window Number is compared with the most significant l bits of "Bank Switch (lo)". If these two numbers do not match, the address location of memory is not within the bank switched memory window and a regular memory or a device register will have to respond to the memory reference. If the two numbers do match, then the address is within the window and the banked switched memory has to handle the memory reference. The MMU then concatenates the m bits of the Bank Switch Register to the left of the l bits of the address within the window to form the m + j bit Bank Switched Space address, [h:j]. This address is then placed on the internal address bus of the Bank Switched Memory sub system and the appropriate location answers.

CEM 924 MEM 8 14-APR-1992
T V Atkinson
Department of Chemistry
Michigan State University



Bank Switching (Partial)

Figure 64 Bank Switching (Partial): Memory Spaces

Using this technique, the cpu memory space can be expanded by $(2^{m})*(2^{l})$ memory locations. As with regular memory, the Bank Switched Memory Space would not have to be fully populated with actual memory.

As an example, if $k = 4$, and $l = 16$, the logical (CPU) address space would be 1048576. The Bank Switch Window would be 65536 (64K). If the Bank Register were such that $m = 9$, the Bank Switched Space would be $2^{(9+16)} = 33554432$ (32M). Thus, by switching in the different banks of memory, the CPU could command 32M of memory. As with other techniques described here, there is the penalty of time required to switch the Bank Register. If the program must switch often among the various banks of memory, performance of the program would be severely decreased.

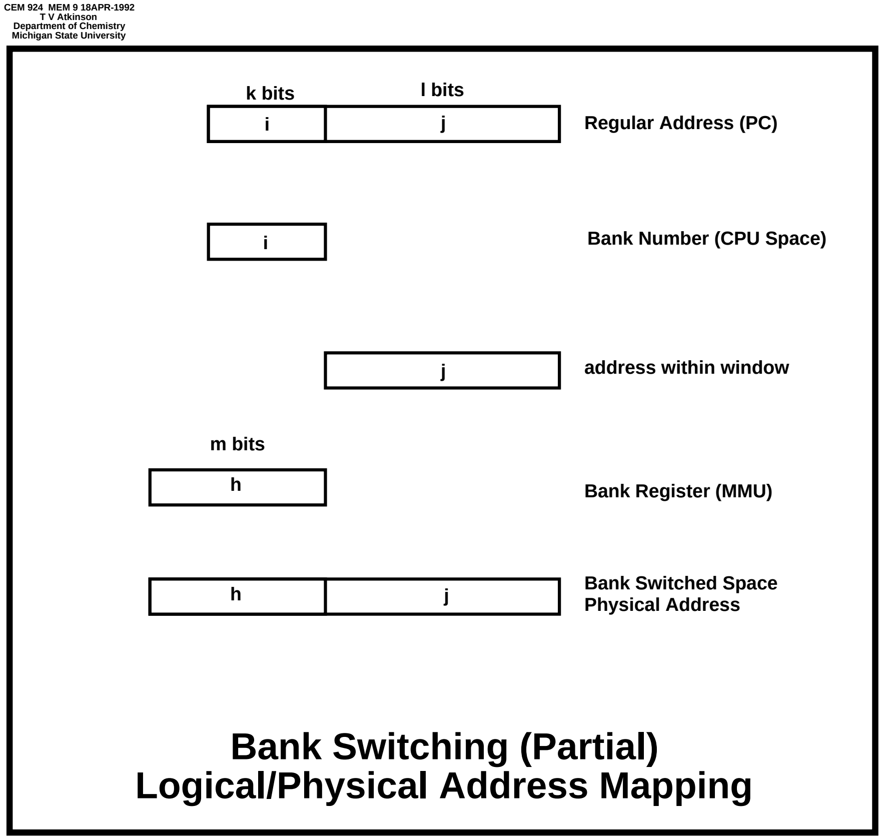


Figure 65 Bank Switching(Partial): Mapping

14.4.3. Segmentation

The MMU contains a Segment Register of m bits(See Figure 66). The MMU receives a memory reference from the CPU that includes a logical memory address [j]. The contents of the Segment Register, [i], is shifted to the left by k bits and added to the logical address, [j], to form a (n+k) physical address,[h], (See Figure 66). Thus, the physical address is $i*2^k + j$. Figure 67 illustrates how the memory spaces appear for the segmentation case. Figure 67 shows two example segments, each corresponding to a particular value in the segment register. While the Segment Register is set to i_1, the CPU could reference any memory location in physical memory that was within Segment 1. While the Segment Register is set to i_2, the CPU could reference any memory location in physical memory that was within Segment 2.

Taking the example of n = 16, m = 16, and k = 4, the Logical (CPU) address space would again be 65536. With segmentation, the physical addresses on the memory address bus can now be 20 bits and support a physical memory address space of 1048576. Now however, the physical memory space can be divided into a large number of segments, 2^m = 65536 actually, which may be overlapping. Each segment will be of length 65536. At any one time, the program is operating in one of the segments. The program switches between the segments by changing the contents of the Segment Register in the MMU. This process of switching the CPU context takes several instruction times to affect. An enhancement of this approach would be to have two segment registers in the MMU. One would be used to map addresses of instruction (code). The second would be used to map addresses of data. This would allow the program to split the code and data into separate segments. In all cases overhead is required to switch the segment registers from one to another.

CEM 924 MEM 10 14-APR-1992
T V Atkinson
Department of Chemistry
Michigan State University

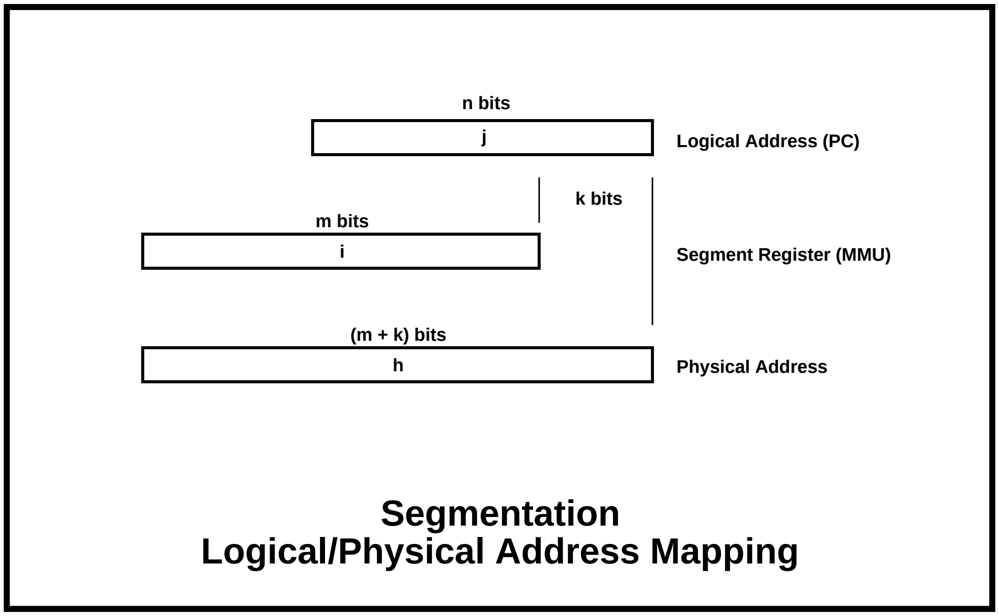


Figure 66 Segmentation: Mapping

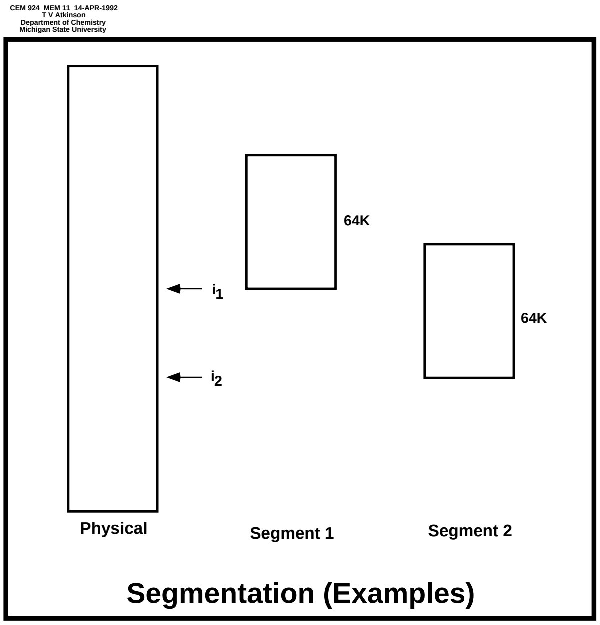


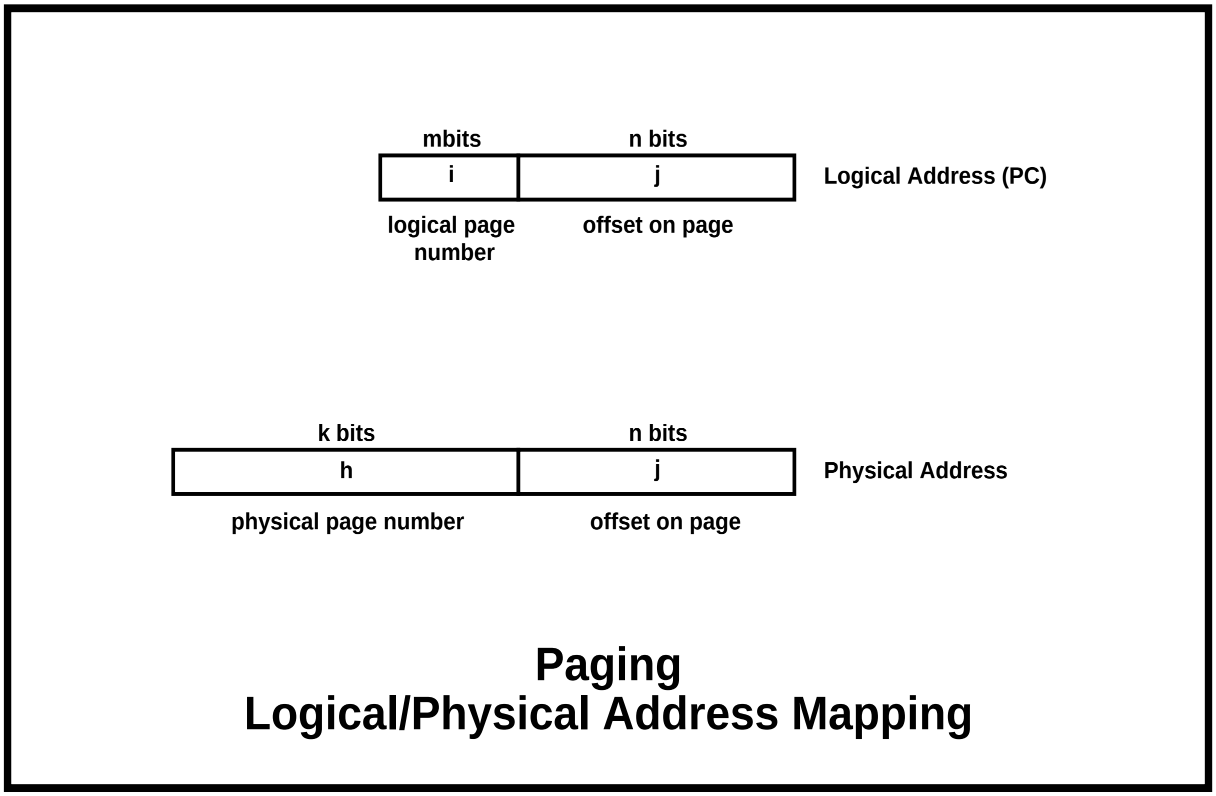
Figure 67 Segmentation: Memory Spaces

14.4.4. Paging

This section describes a simplified paging mechanism. In this paging system (See Figure 68, 69, 70), the logical memory space is divided into 2^m pages (LP_0, LP_1, …, LP_r) of size 2^n. Physical memory space is divided into 2^k pages (PP_0, PP_1, … , PP_p) of size 2^n (i.e. the same size as logical pages). The map of the transformation of logical pages into physical pages is contained in a set of registers called the Page Table that is located in the MMU (See Figure 70). These Page Table registers are located in the I/O space of the CPU and their contents are maintained by system software. Each entry in the Page Table contains a one bit write enable "W" register [W = 0, page is read only. W = 1, page is read/write.] and a k bit physical page number. For example in Figure 70, logical page LP_2 would actually be located on physical page PP_c.

Thus, to transform (See Figure 68) a given logical address [i:j], the contents of the i'th entry of the Page Table is concatenated with [j], the offset within the logical page, to form the physical address [h:j]. If the MMU receives a memory reference to write into a location on a page that is write protected (W = 0), an exception is declared and the operating system is notified and appropriate error handling occurs. The memory reference is aborted.

CEM 924 MEM 12 14-APR-1992
T V Atkinson
Department of Chemistry
Michigan State University



Paging
Logical/Physical Address Mapping

Figure 68 Paging: Mapping

The main problem with this paging mechanism is the size of the Page Table that would be required for modern computer systems. The usual techniques modify the mechanism described here so that only a portion of the Page Table is maintained in the MMU at any one time. Caching and other techniques make this possible.

CEM 924 MEM 13 18-APR-1992
T V Atkinson
Department of Chemistry
Michigan State University

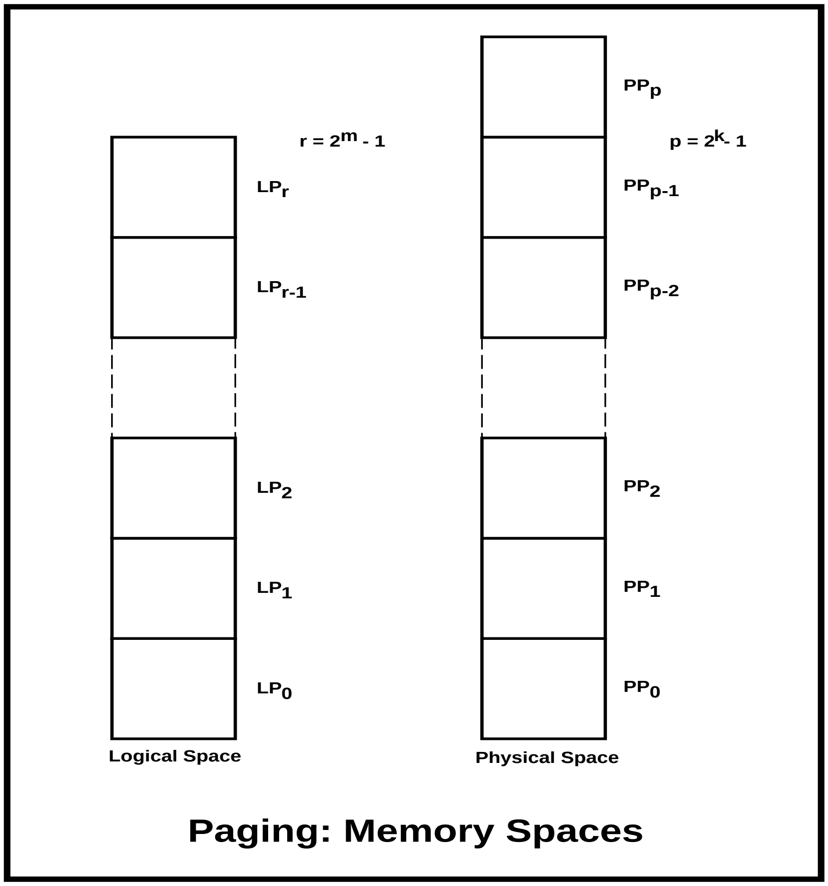
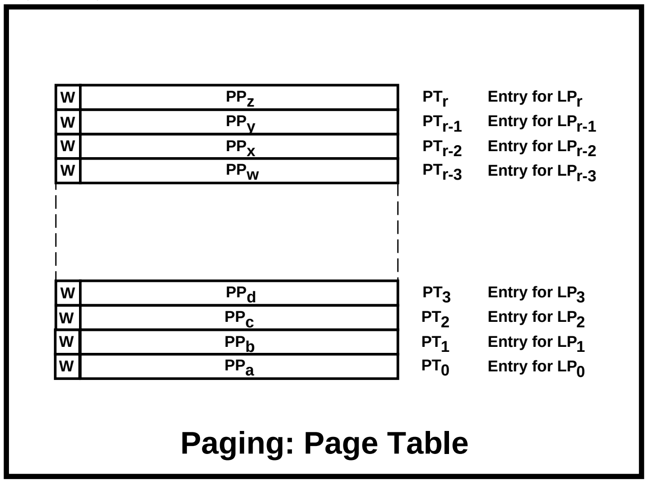


Figure 69 Paging: Memory Spaces

CEM 924 MEM 14   18-APR-1992
T V Atkinson
Department of Chemistry
Michigan State University



W	PP_z	PT_r	Entry for LP_r
W	PP_y	PT_{r-1}	Entry for LP_{r-1}
W	PP_x	PT_{r-2}	Entry for LP_{r-2}
W	PP_w	PT_{r-3}	Entry for LP_{r-3}
W	PP_d	PT_3	Entry for LP_3
W	PP_c	PT_2	Entry for LP_2
W	PP_b	PT_1	Entry for LP_1
W	PP_a	PT_0	Entry for LP_0

Paging: Page Table

Figure 70 Paging: Page Table

As an example of paging consider the two page program segment shown in Figure 71 where n = 9, making page sizes 512, m = 7 or 128 pages in the logical memory space, k = 11 or 2048 pages in the physical memory space. The size of the logical memory space is 2^{16} or 65536. The size of the physical memory space is $2^{(n + k)}$ or 1048576.

A and B are two memory locations within the program (See Figure 71a) and are used to illustrate details of the translation process that takes place for all memory references. The operating system in the process of loading the program into memory divides the program into logical pages (See Figure 71b). Note that locations within a page can be expressed relative to the individual page. The operating system assigns the program segment space in the logical memory (See Figure 71c). When the program is actually loaded into memory, the operating system finds the necessary free space within the physical memory and assigns the logical pages to physical pages (See Figure 71d) by making the appropriate entries in the Page Table (See Figure 71e). The actual pages of information can then be loaded into physical memory from the disk file containing the image of the program segment. Once the loading is complete, execution of the program segment can begin. Notice that this paging mechanism allows the logical pages to be distributed arbitrarily through the physical memory space allowing easier mapping for large collections of processes of different sizes that constantly change.

Address of Location A in various Memory Spaces

Memory Space	Binary	Octal	Decimal	Hex
Within Program	00000000000111011101	0000735	477	001DD
Within Page	00000000000111011101	0000735	477	001DD
Logical	00000001100111011101	0014735	6621	019DD
Physical	11100000001111011101	3401735	918493	E03DD

Address of Location B in various Memory Spaces

Memory Space	Binary	Octal	Decimal	Hex
Within Program	00000000001000110110	0000566	1066	00236
Within Page	00000000000000110110	0000066	54	00036
Logical	00000001110000110110	0016066	7222	01C36
Physical	00001111101000110110	0175066	64054	0FA36

CEM 924 MEM 14a 19-APR-1992
T V Atkinson
Department of Chemistry
Michigan State University

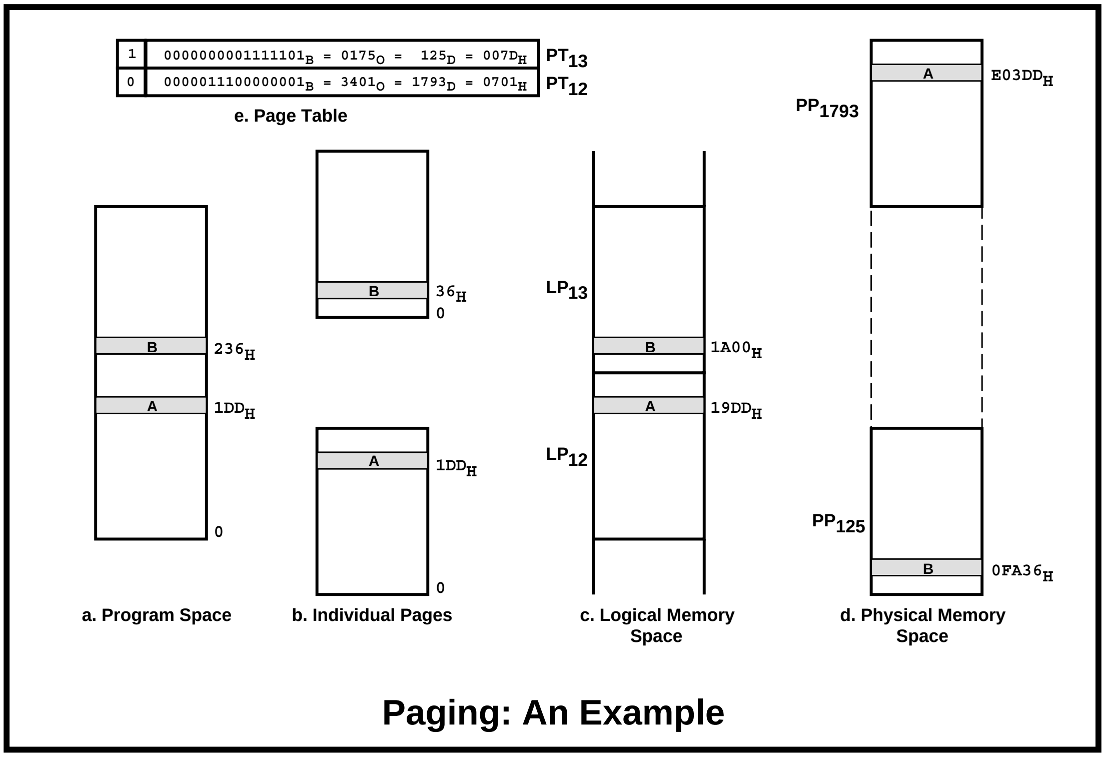


Figure 71 Paging: An Example

14.4.5. Virtual Memory

This section describes a simplified virtual memory system that is an extension of the above paging system. Each entry in the Page Table now contains three one bit registers (R, M, W) and a k bit physical page number register. The R register indicates whether the page is resident in physical memory. The M register indicates if the resident page has been modified while in physical memory. The W register indicates, as before, whether the page is to be written into by the program or not.

For each memory reference the following process is followed to map a logical address into a physical address.

1. Find the entry for the logical page in the Page Table.

2. If page is resident, form the physical address and place on the memory address bus.

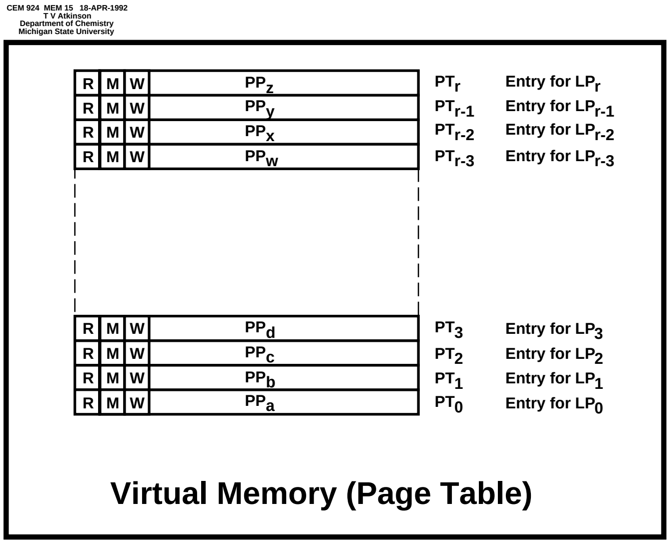

CEM 924 MEM 15 18-APR-1992
T V Atkinson
Department of Chemistry
Michigan State University
R M W PP_z PT_r Entry for LP_r
R M W PP_y PT_{r-1} Entry for LP_{r-1}
R M W PP_x PT_{r-2} Entry for LP_{r-2}
R M W PP_w PT_{r-3} Entry for LP_{r-3}
R M W PP_d PT_3 Entry for LP_3
R M W PP_c PT_2 Entry for LP_2
R M W PP_b PT_1 Entry for LP_1
R M W PP_a PT_0 Entry for LP_0
Virtual Memory (Page Table)

Figure 72 Virtual Memory: Page Table

3. If page is not resident, then do a Page Fault.

   3.1. If there is an empty entry in the Page Table indicating a free page of memory, assign the new logical page to the free physical page and make the appropriate entry in the Page Table.

      3.1.1. Read the requested logical page from disk into the reclaimed physical page.

      3.1.2. Form the physical address and place on the memory access bus.

   3.2. If there is not an empty entry in the Page Table, indicating a free page of memory, do the following

      3.2.1. Decide which physical page can be reclaimed.

      3.2.2. If the physical page that is to be reclaimed has been modified, write the physical page into the corresponding logical page on disk.

      3.2.3. Read the requested logical page from disk into the reclaimed physical page.

      3.2.4. Form the physical address and place on the memory access bus.

CEM 924 MEM 15a 20-APR-1992

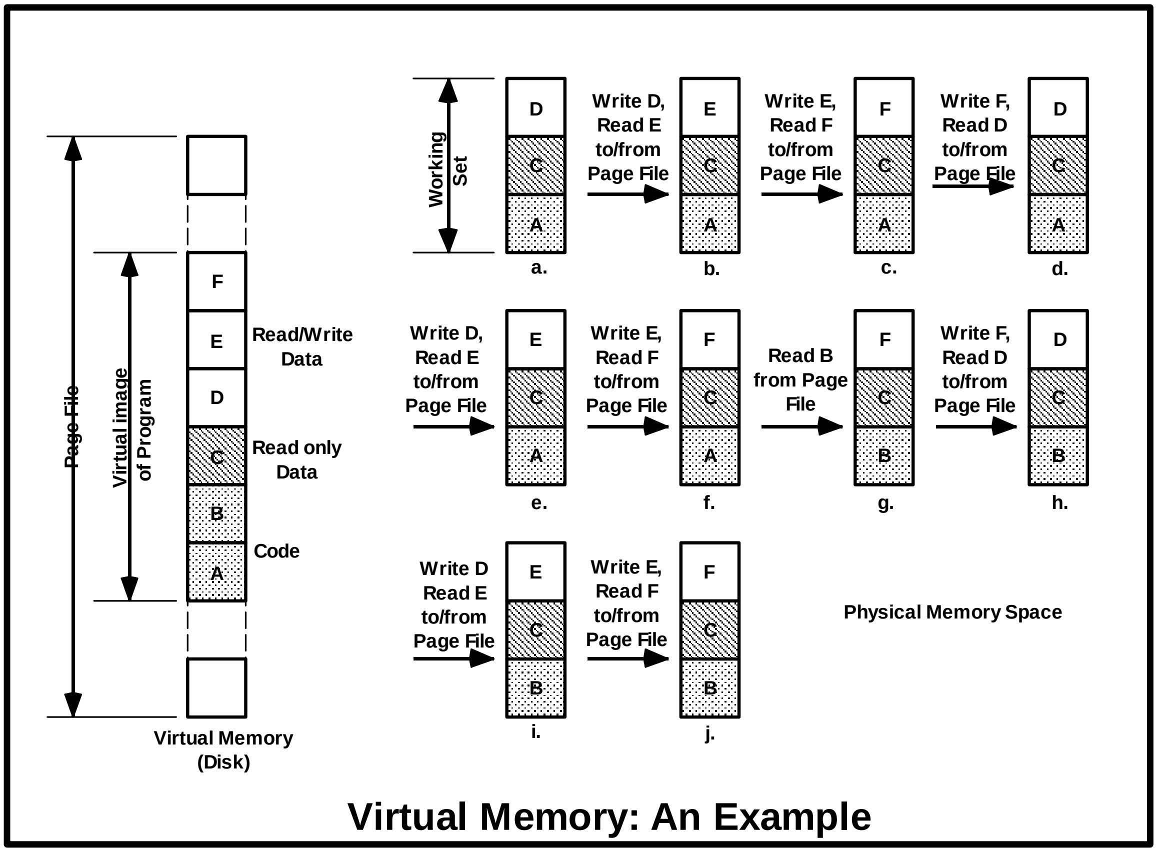

Page File
Virtual image of Program
F
E
D
C
B
A
Read/Write Data
Read only Data
Code
Virtual Memory (Disk)
Working Set
D
C
A
a.
Write D, Read E to/from Page File
E
C
A
b.
Write E, Read F to/from Page File
F
C
A
c.
Write F, Read D to/from Page File
D
C
A
d.
Write D, Read E to/from Page File
E
C
A
e.
Write E, Read F to/from Page File
F
C
A
f.
Read B from Page File
F
C
B
g.
Write F, Read D to/from Page File
D
C
B
h.
Write D Read E to/from Page File
E
C
B
i.
Write E, Read F to/from Page File
F
C
B
j.
Physical Memory Space
Virtual Memory: An Example

Figure 73 Virtual Memory: An Example

Figure 73 illustrates a simple program in a virtual memory system. The program consists of 6 pages of memory, labeled A, B …, F and. In this example, Page A contains code that loads an experimental spectrum into a data array that is contained in pages D, E, and F. The code in page A also steps through the data correcting the data using constants stored in page C. Page B contains programming that outputs the corrected spectrum to an output data file. In this example, the process is allowed three pages of physical memory, the working set. The execution of the program is as follows.

1. Page A is loaded and requests C and then begins trying to load the data, causing D to be loaded into the working set. Once D is loaded, the program in page A begins trying to load into page E. This causes a page fault.

2. The program in page A loads data into page E. Once E is loaded, the program in page A begins trying to load into page F. This causes a page fault.

3. The program in page A begins to branch to the program in page B. This causes a page fault.

4. The program in page B begins to process data on page D. This causes a page fault.

5. The program in page B processes the data on page D and then begins to process data on page E. This causes a page fault.

6. The program in page B processes the data on page E and then begins to process data on page F. This causes a page fault.

7. The program in page B processes the data on page F and then begins to write the data on page D to the output file. This causes a page fault.

8. The program in page B writes the data on page D to the output file and then begins to write the data on page E to the output file. This causes a page fault.

9. The program in page B writes the data on page E to the output file and then begins to write the data on page F to the output file. This causes a page fault.

10. The program in page B writes the data on page F to the output file.

14.4.5.1. Common Sections of memory

In multitasking operating systems, when more than one task has read only sections of data and/or code that are exactly like those of another task that is resident, it is possible for both(or more) tasks to share one copy of these sections rather than each task demanding a copy. This will, of course, save memory and perhaps the time needed to load the extra copies of the shared information into memory. A typical example of this is a run time library of standard math subroutines for languages like FORTRAN, C, BASIC, etc. This technique is made possible by paging.

CEM 924 MEM 16 14-APR-1992
T V Atkinson
Department of Chemistry
Michigan State University

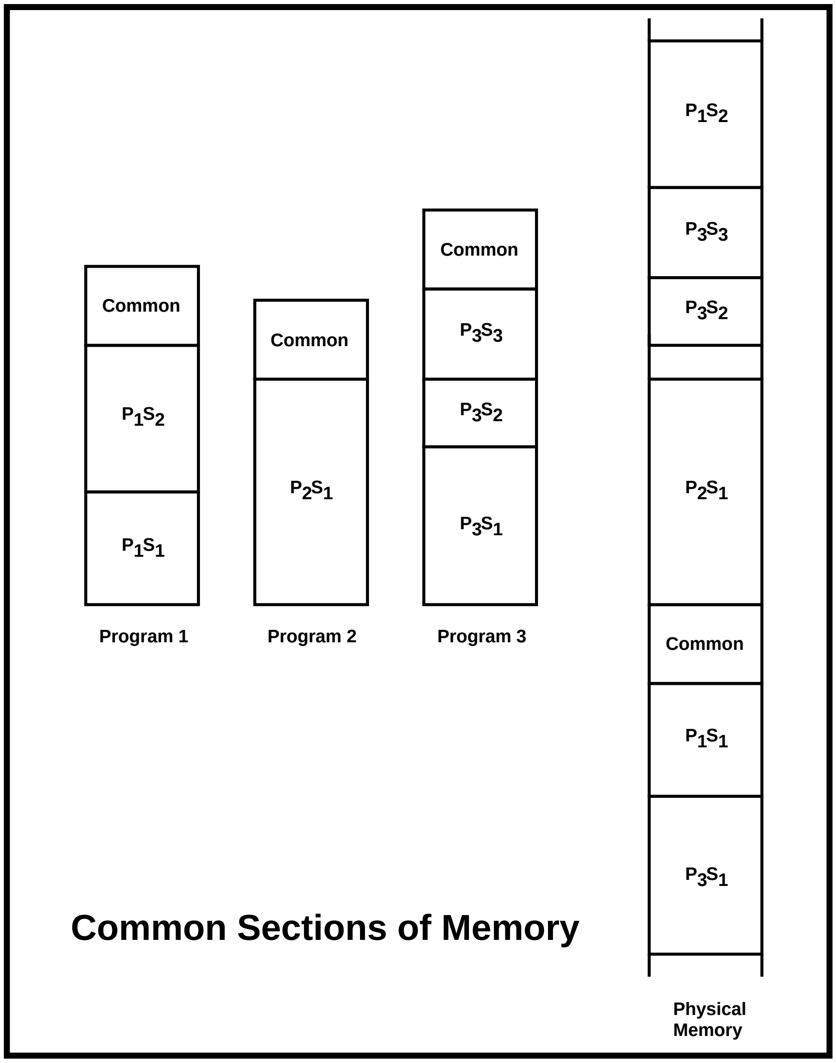


Figure 74 Common Sections

14.4.6. Memory Protection

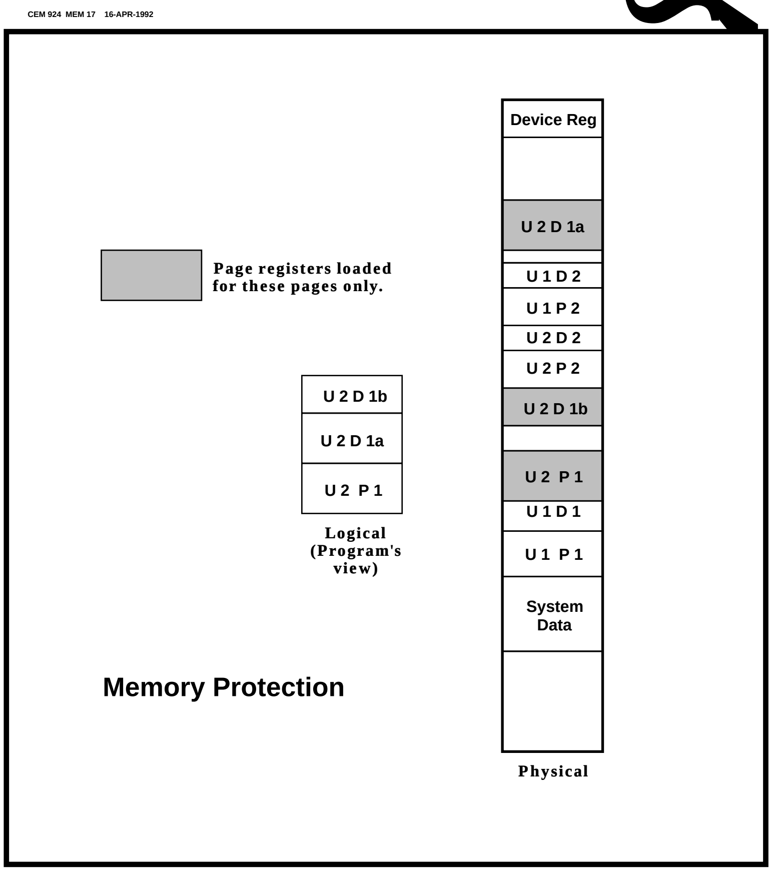


Figure 75 Memory Protection

15. Value of a Particular Computing Environment

1. Functionality
2. Compatibility with existing facilities
3. Performance
4. Cost to acquire
5. Cost to own
6. Reliability
7. Expandability

8. Ability to be Upgraded

9. Convenience

16. Measurement of Performance

A benchmark is typically a particular program with a given set of input data that is run in a given computer environment to measure the performance of that environment by comparing the results to those of other machines and environments. Generally trying to answer the questions:

1. Which computer or operating system or software should I buy?

2. What were the results of trying to improve a given environment with a given set of software or hardware changes?

16.1. Benchmarks

The following methods of determining the performance of a computing system is listed in the order of increasing desirability.

1. CPU Clock Speed (almost useless except when comparing examples of same architecture)

2. Instruction times: MIPS, MFLOPS, I/O rates, Graphic drawing rates, …

3. Standard single job benchmark: Linpack, Whetstone, Dhrystone, Livermoore Loops, Specmark, …

4. Your single job benchmark

5. Multiple standard jobs running simultaneously

6. Your mix of your jobs running in your environment

7. History of your system over an extended length of time

17. CISC vs RISC

Performance versus Memory

17.1. Main Attributes of RISC

1. Reduce the number of Instructions

2. Load and Store only Memory reference Instructions

3. ALU instructions occur in 1 CPU clock cycle

4. Cache !!!

5. Pipelines !!!

6. Lots of registers