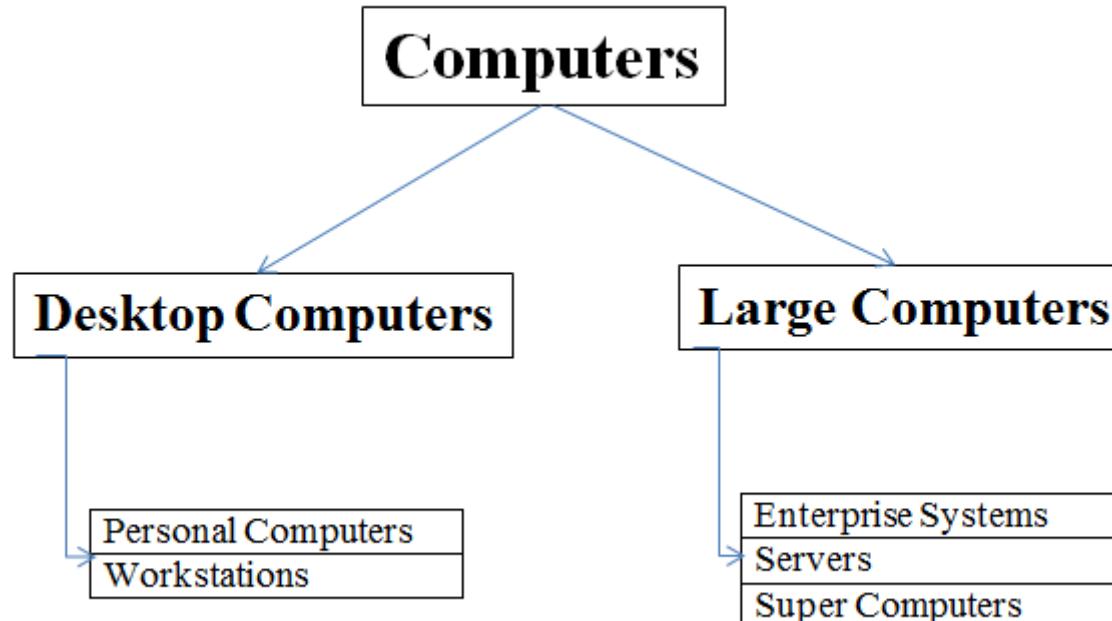


❖ *Computer Types:*

- ✓ **Definition of Computer:**
- ✓ Fast Electronic Calculating machine
- ✓ Input: Digitized info
- ✓ Process: Digitized info as per the stored instructions
- ✓ Output: Resulting information
- ✓ **Program:** The list of instructions that process Digitized info to produce the output
- ✓ **Memory:** Internal storage

Continued...

✓ Types:



Continued...

- ✓ Computers are classified into Desktop Computers and Large systems based on Size, Cost, Computational Power
- ✓ **Desktop Computers:**
- ✓ General Properties:
- ✓ They have Processing units, Storage units, Visual display, Audio O/P units, Keyboard
- ✓ Point(s) specific to Personal Computers:
- ✓ Most Common form of desktop computers
- ✓ Uses: Homes, Schools, Business offices
- ✓ Portable notebook computers are compact flavor of PCs i.e., all the components are packaged into an unit
- ✓ Point(s) specific to Work Stations:
- ✓ This has high resolution graphics I/O capabilities
- ✓ Computational Power(Work Stations)>Computational Power(PCs)
- ✓ Uses: Engineering applications

Continued...

- ✓ **Large Systems:**
- ✓ General Properties:
 - ✓ Large and very powerful than Desktop Computers
 - ✓ Multiple Processors, Technology like Parallel programming
 - ✓ Enterprise Systems, Servers at the lower end of the range, Super Computers at the higher end
 - ✓ Point(s) specific to Enterprise Systems
 - ✓ Enterprise Systems/Mainframes
 - ✓ Used in business data processing in medium to large corporations
 - ✓ Computing power and storage capacity is much more than workstations

Continued...

- ✓ Point(s) specific to Servers: (IRCTC, GOOGLE....)
- ✓ They contain database storage
- ✓ They are capable of handling large volumes of requests
- ✓ Uses: Education, Business, Personal user communities
- ✓ Requests and responses are transported over internet
- ✓ Internet communication happens via high speed fiber-optic links
- ✓ Point(s) specific to Super Computers:
- ✓ Used for Large scale numerical calculations
- ✓ Weather Forecasting applications, Satellite image processing applications, aircraft design and simulation

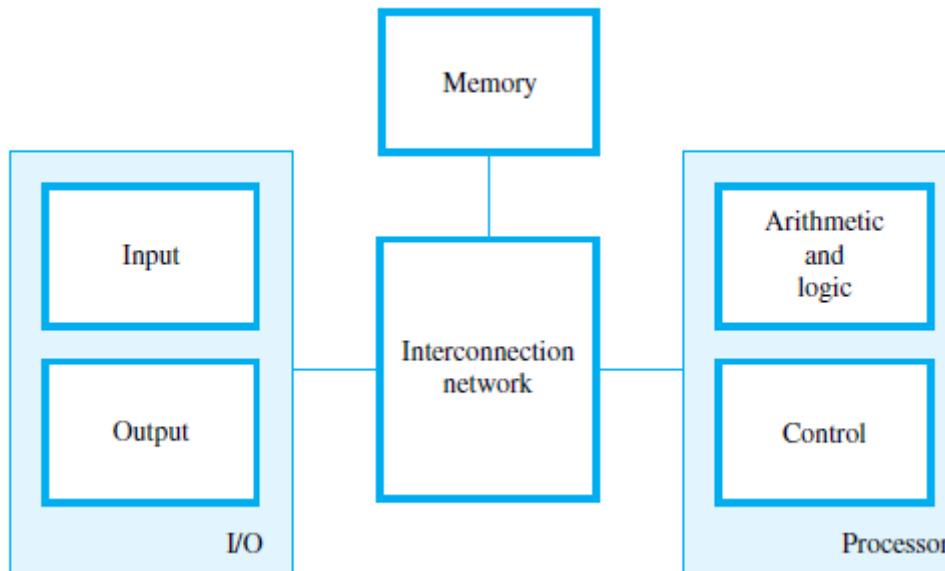
Continued...

❖ *Functional Units:*

- ✓ **Introduction:**
- ✓ A computer has 5 *functionally* independent parts
- ✓ They are input, output, memory, control, arithmetic and logic units
- ✓ Input and Output units are collectively called I/O
- ✓ ALU and Control units are collectively called Processor
- ✓ Next Slide

Continued...

Functional Units Diagram:



Continued...

- ✓ Input unit accepts coded info from electromechanical devices like keyboard , from other computers over digital communication links
- ✓ Memory stores the inputted data for later reference
- ✓ Inputted data may be given directly given to ALU and performs arithmetic/logic operations (Program stored in memory)
- ✓ Output unit sends the result to the outside world
- ✓ Control unit coordinates these actions
- ✓ There are several ways in which these units can be connected

Continued...

- ✓ Info handled by a computer:
- ✓ Instructions/Values/Addresses-----Data
- ✓ About instructions:
- ✓ Specify arithmetic, logical operations to be performed
- ✓ Govern transfer of info within computer, between computer and I/O devices
- ✓ Program is a list of instructions
- ✓ When a program is to be executed it must be in memory
- ✓ Processor fetches the instructions of the program to perform operations

Continued...

- ✓ Difference between values and addresses
- ✓ Values can be numbers/characters
- ✓ But addresses are numbers(integers in binary)
- ✓ Example wherein a program can be called data
- ✓ **Compiler**
- ✓ Information handled by a computer must be encoded in suitable form?
- ✓ Alphanumeric characters: **ASCII** (to denote a character 8 bits)

Continued...

- ✓ Input Unit:
- ✓ Computers accept coded info through input units
- ✓ Keyboard
- ✓ What happens during a Key press?
 - ✓ **When a key is pressed the ascii value for the key gets stored in a memory location**
- ✓ Others include joysticks, trackballs, mouse
- ✓ Graphic input devices

Continued...

- ✓ Memory Unit:
- ✓ To store programs and data
- ✓ RAM: (Primary memory/Main memory)
- ✓ Memory has a large no of **semiconductor storage cells** each of which will store one bit
- ✓ Individually they are rarely read/written
- ✓ Instead they are read/written in groups
- ✓ ?
- ✓ Word (16 to 64 bits and depends on ALU)
- ✓ Memory is organized in a way that a word can be read/written in an operation

Continued...

- ✓ Addresses are associated with words for easy retrieval
- ✓ Address and control command
- ✓ Memory capacity determines size of computer
- ✓ Smaller machines---tens of millions of words
- ✓ Larger--- hundreds of millions of words
- ✓ For memory word is the basic building block
- ✓ RAM---Random Access Memory
- ✓ Memory where any word can be accessed in the same amount of time
- ✓ Memory Access Time?
- ✓ Time required to access a word
- ✓ In modern RAMs it is few ns to 100ns
- ✓ Cache---Small, fast RAM unit(s)

Continued...

- ✓ ALU:
- ✓ Most of the operations are executed in ALU
- ✓ Addition of 2 numbers. How?
- ✓ Any other operation like mul, comparison can be done by bringing operands into processor and operation being performed by ALU
- ✓ Operands are stored in the registers of the processor
- ✓ CU and ALU are many times faster than other computer devices.
- ✓ That is how processor controls a number of devices

Continued...

- ✓ Output Unit:
- ✓ Counterpart
- ✓ Send processed results outside
- ✓ Printer
- ✓ Ink Jet printers, Laser printers (photocopying technique—Xerox machines) for printing
- ✓ We can have a printer that prints 10000 lines/min
- ✓ This speed may be great for mechanical device but nothing in front of processor speed

Continued...

- ✓ Control Unit:
- ✓ The operations of input, output, memory, ALU must be coordinated
- ✓ CU
- ✓ Its job is to send control signals to other units to get to know the states
- ✓ I/O transfers, input and output operations are controlled by instructions
- ✓ But then actual timing signals for transfers are generated by CU
- ✓ Timing signals determine when a given action has to take place
- ✓ Data transfers b/w processor and memory are also controlled by CU
- ✓ CU theoretically can be thought of as a unit that interacts with other parts
- ✓ In practice the circuitry is spread throughout the machine

❖ ***Basic Operational Concepts:***

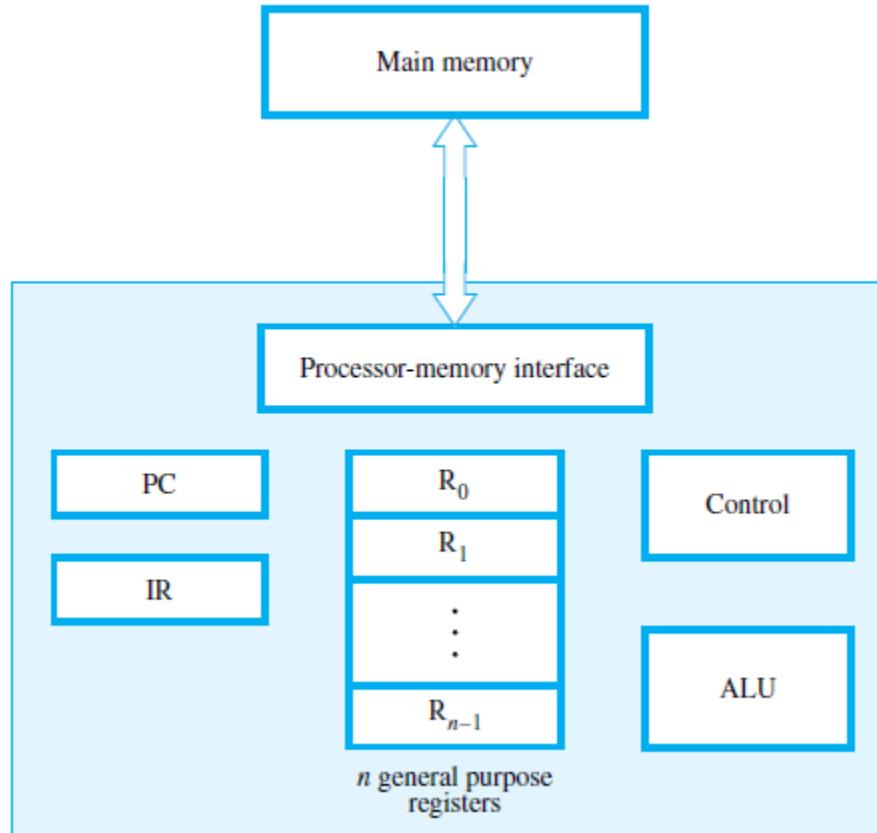
- ✓ Program in memory
- ✓ Individual instructions are brought from memory into processor (also data) to perform the operations
- ✓ Ex1: Load R2, LOC
 - ▶ This instruction reads the contents of a memory location whose address is represented symbolically by the label LOC and loads them into processor register R2.
 - ▶ The original contents of location LOC are preserved, whereas those of register R2 are overwritten.
- ✓ Ex2: Add R4, R2, R3
 - ▶ adds the contents of registers R2 and R3, then places their sum into register R4. The operands in R2 and R3 are not altered, but the previous value in R4 is overwritten by the sum.

Continued...

- ✓ Ex3: Store R4, LOC
 - This instruction copies the operand in register R4 to memory location LOC.
 - The original contents of location LOC are overwritten, but those of R4 are preserved.
 - PTO

Continued...

- ✓ Connections between memory and processor (one of the ways)



Continued...

- ✓ IR—Instruction Register
- ✓ It holds the instruction currently being executed
- ✓ PC—Program Counter
- ✓ It contains the address of the next instruction to be fetched and executed
- ✓ During the execution of an instruction PC is updated
- ✓ General purpose registers hold addresses/values (R0-Rn-1)

Continued...

- ✓ Only 2 registers communicate with memory
- ✓ MAR—Memory Address Register
- ✓ MDR—Memory Data Register
- ✓ MAR holds the address of location to be accessed
- ✓ MDR holds the data read from/written into the addressed location
- ✓ Scenario: **MAR, MDR, Control Signal**
 - ✓ 1. Instruction has to be fetched whose address is in PC
 - ✓ 2. Data has to be fetched from memory
 - ✓ 3. Result has to be stored in memory

Continued...

- ✓ Interrupts?
- ✓ A request from I/O devices
- ✓ Scenario: I/O device wants immediate attention of Processor
- ✓ Sends interrupt signals
- ✓ Processor before servicing the interrupt saves the PC, GPR of the current program
- ✓ Processor executes ISR for that interrupt
- ✓ Can be returned to the previously executing program

❖ *Numbers:*

- ✓ Computers operate on info represented by 2 electric signals 0 and 1
- ✓ 01000111001010—Bit/Binary digit
- ✓ What is the way to represent numbers and characters?
- ✓ String of bits
- ✓ Number representation:
- ✓ Unsigned integers
- ✓ A n-bit vector can represent unsigned integers in the range?
- ✓ 0 to $(2^n) - 1$

Continued...

- ✓ It is equally impt to represent signed numbers
- ✓ How?
- ✓ 3 Systems
 - ✓ 1. Sign and Magnitude
 - ✓ 2. 1's Complement
 - ✓ 3. 2's Complement
- ✓ Thing that is common to all 3 is for +ve numbers MSB is 0, for -ve numbers MSB is 1
- ✓ Positive values have identical representation in all 3 systems but negatives have different representations

Continued...

- ✓ Sign and Magnitude:
- ✓ In this system 1 bit (MSB) is reserved for sign and the rest of the bits to represent magnitude
- ✓ Ex: +5 (4-bit binary format)
0101
- ✓ Ex: -5 (4-bit binary format)
1101
- ✓ Just change the MSB from 0 to 1 if the negative value of a positive value needs to be represented
- ✓ Next Slide

Continued...

- ✓ 1's Complement:
- ✓ For +ve numbers:
- ✓ Normal Binary form (Unsigned number)
- ✓ For -ve numbers:
 - ✓ The binary rep of the corresponding +ve number is taken and all bits are complemented
 - ✓ Ex: +7 (4-bit format)
 - ✓ 0111
 - ✓ Ex: -7 (4-bit format)
 - ✓ Take binary rep of +7 and complement all bits
 - ✓ 1000

Continued...

- ✓ 2's Complement:
- ✓ Ex: +7 (4-bit binary form)
- ✓ 0111
- ✓ You subtract this from 10000
- ✓ What you get is 1001 which is the 2's complement for -7
- ✓ Table (PTO)

Continued...

✓ **B** Values represented

$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+ 7	+ 7	+ 7
0 1 1 0	+ 6	+ 6	+ 6
0 1 0 1	+ 5	+ 5	+ 5
0 1 0 0	+ 4	+ 4	+ 4
0 0 1 1	+ 3	+ 3	+ 3
0 0 1 0	+ 2	+ 2	+ 2
0 0 0 1	+ 1	+ 1	+ 1
0 0 0 0	+ 0	+ 0	+ 0
1 0 0 0	- 0	- 7	- 8
1 0 0 1	- 1	- 6	- 7
1 0 1 0	- 2	- 5	- 6
1 0 1 1	- 3	- 4	- 5
1 1 0 0	- 4	- 3	- 4
1 1 0 1	- 5	- 2	- 3
1 1 1 0	- 6	- 1	- 2
1 1 1 1	- 7	- 0	- 1

Continued...

- ✓ Sign and magnitude, 1's complement have diff representations for +0 and -0
- ✓ But 2's complement has only one representation
- ✓ -8 has representation only in 2's complement form
- ✓ 2's complement is the most efficient way to carry out arithmetic operations
- ❖ *Arithmetic Operations:*
 - ❖ Addition of Positive/ Unsigned numbers (4-bit binary format)
 - ❖ Adding 15 with 15 (30)
 - ❖ 1111 with 1111 (11110) Now the carry generated as a result of addition is considered overflow since overflow has occurred

Continued...

- ✓ Addition and Subtraction of Signed numbers
- ✓ When the 2 nos that are to be added are in the range of system and if after addition the result is in the range of the system then no overflow has occurred. **Carry generated if any in this case** will not be considered overflow ($+7+(-3)$)
- ✓ When the 2 nos that are to be added are in the range of system and if after addition the result is not in the range of the system then overflow has occurred. **Carry generated in this case** will be considered overflow $-7+(-2) = -9$
- ✓ **When 2 operands have same sign and the MSB of the result is of opposite sign then overflow is said to have occurred -** $7+(-2) = -9$

Continued...

- ✓ Addition rules:
- ✓ Add
- ✓ If the result is in the range, any carry generated is ignored
- ✓ If the result is out of the range, carry generated in that case is considered overflow
- ✓ Subtraction rules:
- ✓ $X - Y$
- ✓ Find 2's complement for Y and add it to X
- ✓ Addition rules apply here

Continued...

(a)
$$\begin{array}{r} 0010 \\ + 0011 \\ \hline 0101 \end{array}$$
 $\xrightarrow{\substack{(+2) \\ (+3)}}$

(c)
$$\begin{array}{r} 1011 \\ + 1110 \\ \hline 1001 \end{array}$$
 $\xrightarrow{\substack{(-5) \\ (-2) \\ (-7)}}$

(e)
$$\begin{array}{r} 1101 \\ - 1001 \\ \hline \end{array}$$
 $\xrightarrow{\substack{(-3) \\ (-7)}}$

(f)
$$\begin{array}{r} 0010 \\ - 0100 \\ \hline \end{array}$$
 $\xrightarrow{\substack{(+2) \\ (+4)}}$

(g)
$$\begin{array}{r} 0110 \\ - 0011 \\ \hline \end{array}$$
 $\xrightarrow{\substack{(+6) \\ (+3)}}$

(h)
$$\begin{array}{r} 1001 \\ - 1011 \\ \hline \end{array}$$
 $\xrightarrow{\substack{(-7) \\ (-5)}}$

(i)
$$\begin{array}{r} 1001 \\ - 0001 \\ \hline \end{array}$$
 $\xrightarrow{\substack{(-7) \\ (+1)}}$

(j)
$$\begin{array}{r} 0010 \\ - 1101 \\ \hline \end{array}$$
 $\xrightarrow{\substack{(+2) \\ (-3)}}$

(b)
$$\begin{array}{r} 0100 \\ + 1010 \\ \hline 1110 \end{array}$$
 $\xrightarrow{\substack{(+4) \\ (-6)}}$

(d)
$$\begin{array}{r} 0111 \\ + 1101 \\ \hline 0100 \end{array}$$
 $\xrightarrow{\substack{(+7) \\ (-3) \\ (+4)}}$

$\begin{array}{r} 1101 \\ + 0111 \\ \hline 0100 \end{array}$ $\xrightarrow{(+4)}$

$\begin{array}{r} 0010 \\ + 1100 \\ \hline 1110 \end{array}$ $\xrightarrow{(-2)}$

$\begin{array}{r} 0110 \\ + 1101 \\ \hline 0011 \end{array}$ $\xrightarrow{(+3)}$

$\begin{array}{r} 1001 \\ + 0101 \\ \hline 1110 \end{array}$ $\xrightarrow{(-2)}$

$\begin{array}{r} 1001 \\ + 1111 \\ \hline 1000 \end{array}$ $\xrightarrow{(-8)}$

$\begin{array}{r} 0010 \\ + 0011 \\ \hline 0101 \end{array}$ $\xrightarrow{(+5)}$

Continued...

- ✓ Sign Extension?
- ✓ Suppose if I want to store -8 in 16-bit register
- ✓ The sign bit is extended to the rest of the bit positions for the value to be retained
- ✓ 2's complement is used for signed numbers rep in modern computers

Continued...

❖ *Characters:*

- ✓ Computers should handle text also
- ✓ Representing characters, digits as characters, punctuations is impt
- ✓ ASCII character set

Performance

The most imp measure of the performance of a comp is how quickly it can execute prgms.

The speed with which a comp executes pgms is affected by the design of its instr set, its H/W and S/W including OS, and the technology in which the H/W is implemented.

- i) Technology
- ii) Parallelism

a) Instruction-level Parallelism: *pipelining*
overlap the exec of the steps of successive instrs.

Total exec time will be reduced.

b) Multicore Processors: *dual-core, quad-core, and octo-core* processors

c) Multiprocessors: *shared-memory multiprocessor message passing multicomp*

- ✓ We need a system that can accommodate very large integers and very small fractions and that system should take care of point because in FP nos point is not fixed its variable
- ✓ Floating point representations
- ✓ What constitute the representation?
- ✓ Sign, String of digits (Mantissa) and exponent constitute the representation
- ✓ Next Slide

Continued...

- ✓ IEEE standard for floating point nos:
- ✓ 3 standards/formats
- ✓ Single precision---32 bit format
- ✓ Double precision--- 64 bit format
- ✓ Single Precision:
 - ✓ In single precision format, 1 bit for sign, 8 bits for exponent, 23 bits for mantissa

Continued...



(a) Single format

Continued...

- ✓ Biased Exponent/ Exponent in Excess 127 representation
- ✓ ?
- ✓ To store the signed exponent as an unsigned integer
- ✓ If the original exponent is -100
- ✓ Then it will be biased to 27
- ✓ Which value is represented by the single precision format 0|00101000|001010 (Normalized version)
- ✓ ?
- ✓ Represent -23.75 in single precision format

Continued...

- ✓ Special Cases:
- ✓ Using 8 bits we can represent signed exponent in the range -128 to +127
- ✓ Exponents are biased
- ✓ The range of values of biased exponent if the exponent field has 8 bits is 0 to 255
- ✓ But 0 and 255 are special values
- ✓ The range is 1 to 254
- ✓ Hence the range of signed unbiased exponents that can be represented is -126 to +127
- ✓ 23 bits represent fractional part of mantissa
- ✓ It is called single precision since it occupies a single 32 bit word
- ✓ Scale factor of 10^{-38} to $+10^{+38}$ is possible here

Continued...

- ✓ Double precision:
- ✓ It occupies 2 32 bit words
- ✓ 1 bit for sign
- ✓ 11 bits for exponent
- ✓ 52 bits for mantissa
- ✓ Here also the exponent is biased and the biasing value is $1023 (2^{n-1}-1)$
- ✓ Range of biased exponents: 1 to 2046 (0, 2047 special values)
- ✓ Range of signed unbiased exponents: -1022 to +1023
- ✓ Scale factor: $10^{+308}, 10^{-308}$ is possible here

Continued...



(b) Double format

Continued...

- ✓ Special Values:
- ✓ $E' = 0 \ \&\& M=0$ (exact zero) or $E'=255 \ \&\& M=0$ (Infinity)
- ✓ The corresponding Es are -127, +128
- ✓ Now let's say I am adding 2 binary fractions that are in single precision whose exponents are -126, -126 and let's say the exponent of the result is -127 and M=0
- ✓ ?
- ✓ Underflow has occurred and the result is 0 (+ or -)
- ✓ If exponent is -127 and M!=0 then the result is a denormal number (smaller than normal)
- ✓ Similarly, let's say I am adding 2 binary fractions that are in single precision whose exponents are 127 and 127 and let's say the exponent of the result is +128 and M=0
- ✓ ?
- ✓ Overflow has occurred and the result is infinity (+ or -)

- ✓ Arithmetic operations:
- ✓ Addition
- ✓ Subtraction
- ✓ Multiplication
- ✓ Division
- ✓ Rules are given with respect to single precision format and these rules can be used for format that is of less or equal length of single precision
- ✓ Next Slide

Continued...

- ✓ Addition/Subtraction rules:
- ✓ 1. Convert the given decimal fractions to binary fractions
- ✓ 2. Normalize the converted binary fractions
- ✓ 3. Sign of the result: Sign of the number with the largest exponent is the sign of result
- ✓ 4. Find the biased exponents and represent the numbers
- ✓ 5. Take the number with smaller exponent and logical right shift its mantissa by difference in the exponents (Don't forget the 1 that is not represented). Now the exponent of this number would have become equal to the exponent of no with larger exponent
- ✓ 6. Add/Subtract the shifted mantissa of the no in step5 with the unshifted mantissa of the other number
- ✓ 7. Normalize the final answer, Represent it in single precision and check the answer for correctness

Continued...

- ✓ The numbers have to be represented in single precision and same is the result
- ✓ 1. $1.25+0.25$
- ✓ 2. $1.25-0.25$
- ✓ Multiplication Rules:
 - ✓ 1. Convert the given decimal fractions to binary fractions
 - ✓ 2. Normalized the converted binary fractions
 - ✓ 3. Sign of the result is the XOR of sign of 2 nos
 - ✓ 4. Find the biased exponents and represent the numbers
 - ✓ 5. Add the biased exponents and subtract 127
 - ✓ 6. Multiply mantissas: (Don't forget the hidden 1) and keep the result obtained
 - ✓ 7. Normalize the mantissa if required and add the generated exponent to the biased exponent obtained in step 5
 - ✓ 8. Represent the result in single precision
 - ✓ 9. Check the correctness of the result

Continued...

- ✓ Multiply : $-5.75 * 1.5 = -8.625$
- ✓ Division Rules:
 - ✓ 1. Convert the given decimal fractions to binary fractions
 - ✓ 2. Normalized the converted binary fractions
 - ✓ 3. Sign of the result is XOR of sign of 2 nos
 - ✓ 4. Find the biased exponents and represent them
 - ✓ 5. Subtract the biased exponents and add 127
 - ✓ 6. Divide mantissas: (Don't forget the hidden 1) and keep the result obtained
 - ✓ 7. Normalize the mantissa if required and add the generated exponent to the biased exponent obtained in step 5
 - ✓ 8. Represent the result in single precision
 - ✓ 9. Check the correctness of the result

Continued...

- ✓ Division: $-5.75/1.5 = -3.833333\dots$
- ✓ Guard Bits and Truncation:
- ✓ We have guard bits (3)
- ✓ First bit is guard bit, second bit is round bit, third bit is sticky bit
- ✓ When bit value 1 comes into sticky bit ,1 is retained
- ✓ This yields more accurate final result

Continued...

- ✓ Suppose if the final result is to be rounded off
- ✓ ?
- ✓ Rounding techniques are there
 - ✓ 1. Chopping
 - ✓ 2. Von Neumann Rounding
 - ✓ 3. Rounding
- ✓ 1. Chopping: Let's say I want to truncate from n bits to m bits
 - ✓ Simply the n-m bits from right to left are chopped

Continued...

- ✓ 2. Van Neumann rounding:
 - ✓ If the bits to be removed are all 0s they are chopped
 - ✓ If any one is 1, then the LSB of retained bits set to 1
- ✓ 3. Rounding:
 - ✓ If the MSB of the bits to be removed is 1, then 1 is added to the LSB of retained bits
- ✓ Best method

Continued...

- ✓ Round the final result **1.110110** from 6 bits to 3 bits in the fraction part using
 - ✓ **1. Chopping**
 - ✓ **2. Van Neumann**
 - ✓ **3. Rounding**
 - ✓ Guard bit, Round bit, Sticky bit + Rounding:
 - ✓ $G=1, R=1$ add 1 to LSB (Round UP)
 - ✓ $G=0, R=0||1$ No change (Round down)
 - ✓ $G=1, R=0$ Look at S
 - ✓ If $S=1$ Round Up
 - ✓ If $S=0$ Round to nearest even (If LSB is 0 leave it or else add 1 to LSB)

Continued...

- ✓ A=0 10001 011011
- ✓ B=0 01111 101010
- ✓ Do Subtract, Truncate the result using rounding

Floating point Arithmetic

- ▶ Addition
- ▶ Subtraction
- ▶ Multiplication
- ▶ Division
- ▶ IEEE 32-bit single precision
- ▶ IEEE 64-bit single precision

- ▶ ***FOR FLOATING POINT PORTION***

As per the syllabus floating point portion is from William stallings Text 2).

But it is covered from Hamacher Text 1.

9.7

9.7.1

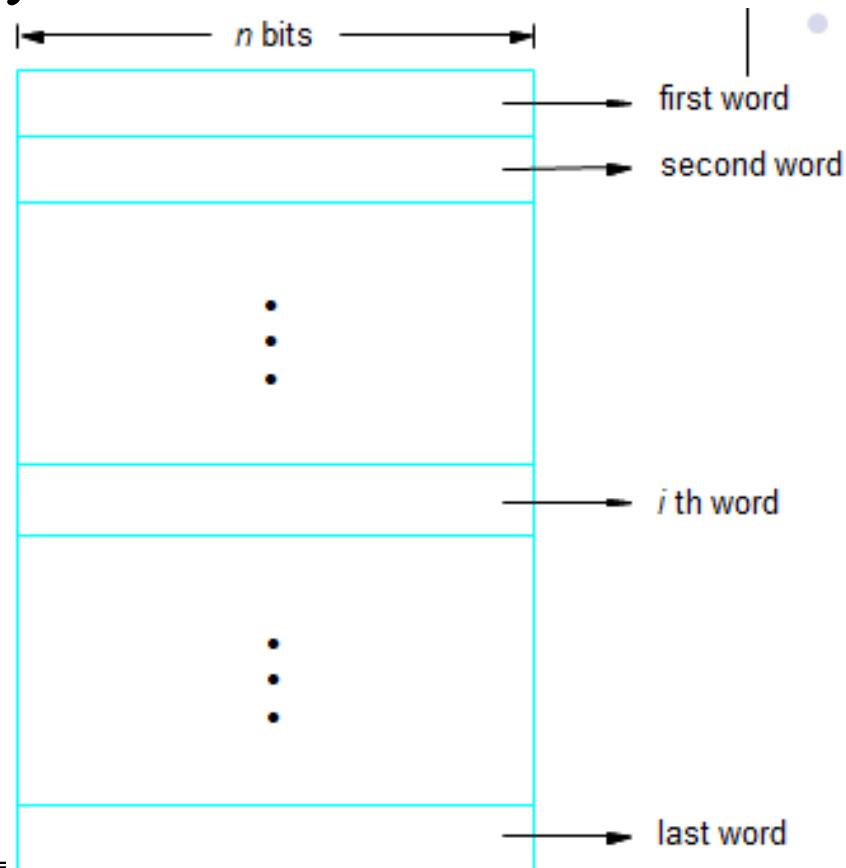
9.7.2

❖ ***Memory Locations and Addresses:***

- ✓ How memory is organized?
- ✓ Memory has millions of storage cells
- ✓ Each cell can store only a bit (Either 0 or 1)
- ✓ Bits are rarely handled individually
- ✓ Instead handled in groups of fixed size
- ✓ Words
- ✓ This can be stored or retrieved in a single operation
- ✓ Memory---Collection of words

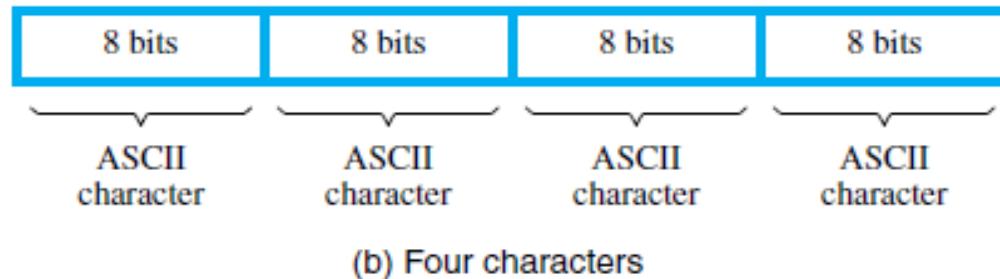
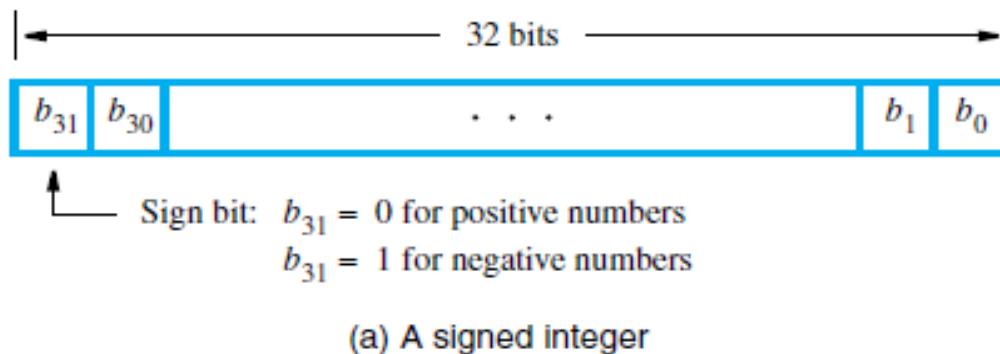
Continued...

✓ Memory Words



Continued...

- ✓ Word lengths vary from 8—64 bits
- ✓ ASCII--- 8 bits are used to represent a character
- ✓ A single word (length=32 bits) can store a 32-bit 2's complement number
- ✓ A single word (length=32 bits) can store 4 characters



Continued...

- ✓ Why do we need addresses?
- ✓ To access a word/or a byte we need distinct names or addresses
- ✓ _____ bit address is required to address 2^{24} locations
- ✓ Range of addresses? Address space?
- ✓ Suppose if the memory of a machine has 2147483648 words (1 word=2 bytes) then _____ bit address is required to address all the locations?
- ✓ $1 \text{ KiloBytes} = 1\text{K} = 1024 \text{ bytes} = 2^{10}$
- ✓ $1 \text{ MegaBytes} = 1\text{M} = 1024\text{K} = 2^{10} * 2^{10} = 2^{20}$
- ✓ $1 \text{ GigaBytes} = 1\text{G} = 1024\text{M} = 2^{10} * 2^{20} = 2^{30}$
- ✓ $1 \text{ TeraBytes} = 1\text{T} = 1024\text{G} = 2^{10} * 2^{30} = 2^{40}$

Continued...

- ✓ Byte Addressability:
- ✓ 1 byte = 8 bits always
- ✓ But not that 1 word=2 bytes always
- ✓ It depends on word length
- ✓ Word length—16
- ✓ 1 word=2 bytes
- ✓ Word length---32
- ✓ 1 word=4 bytes
- ✓ Word length—64
- ✓ 1 word= 8 bytes
- ✓ Rather addresses are assigned for every byte. It is called Byte addressability

Continued...

- ✓ Big Endian and Little Endian Assignment:
- ✓ Let's say 1 word=2 bytes i.e, 1 word=16 bits
- ✓ ALU is 16-bit and Registers—16 bits
- ✓ EX: 1
- ✓ How this ABCDH will be stored in memory in Big Endian?
- ✓ CD which is the lower order byte value is moved to the higher order byte address of that word and AB is moved to lower order byte address of that word
- ✓ CD is stored in address 0001 and AB in address 0000
- ✓ How this ABCDH will be stored in memory in Little Endian?
- ✓ AB in Higher order byte address and CD in lower order byte address
- ✓ AB in 0001 and CD in 0000
- ✓ In both addresses for words are not going to change

Continued...

✓ Big Endian and Little Endian Diagrams:

Word address	Byte address			
0	0	1	2	3
4	4	5	6	7
	•	•	•	•
$2^k - 4$	$2^k - 4$	$2^k - 3$	$2^k - 2$	$2^k - 1$

(a) Big-endian assignment

Byte address	3	2	1	0
0	3	2	1	0
4	7	6	5	4
	•	•	•	•
$2^k - 4$	$2^k - 1$	$2^k - 2$	$2^k - 3$	$2^k - 4$

(b) Little-endian assignment

Continued...

- ✓ Word Alignment:
- ✓ When a word is said to have aligned in memory?
- ✓ When they begin at addresses that are multiple of the no of bytes that make up a word
- ✓ If not they are unaligned
- ✓ Ex 1: 1 word=4 bytes
- ✓ Then words begin at addresses that are multiples of 4 i.e., 0,4,8,12...
- ✓ Word boundaries
- ✓ Ex 2: 1 word=8 bytes How about word boundaries
- ✓ 0,8,16.... (Multiple of 8)

Continued...

- ✓ Accessing numbers, characters, character strings:
- ✓ A number usually occupies a word
- ✓ By giving word address a number can be accessed
- ✓ A character usually occupies a byte
- ✓ By giving byte address a character can be accessed
- ✓ About character string?
- ✓ Beginning----- we can give the byte address of the first character
- ✓ Ending----- 1. It is better to have a control character after the actual string that says “End of String” (\$)
- ✓ 2. A separate loc that contains the length of string

Continued...

❖ *Memory Operations:*

- ✓ 2 basic operations: Load (Read/Fetch), Store(Write)
- ✓ Load operation: transfers copy of contents of ML to processor
- ✓ The Memory contents are not changed
- ✓ For load operation, processor sends address and read control signal
- ✓ Store operation: transfers an item from processor to ML
- ✓ The memory contents are changed
- ✓ For store operation, processor sends address ,data and write control signal

Continued...

❖ *Instructions and Instructions Sequencing:*

- ✓ A computer must have ins for performing 4 types of operations
- ✓ Data transfer between memory and processor registers
- ✓ A/L operations on data
- ✓ Program sequencing and control
- ✓ I/O transfers
- ✓ Some notations are required

Continued...

- ✓ Register Transfer notation:
- ✓ Possible locations in transfers may be memory loc, registers, registers in I/O
- ✓ How to refer to the content of a loc?
- ✓ By enclosing the name by square brackets
- ✓ Ex 1: $R1 <- [LOC]$
- ✓ Meaning?
- ✓ Ex 2: $R3 <- [R1] + [R2]$
- ✓ Meaning?
- ✓ This is Register Transfer notation

Continued...

- ✓ Assembly Language notation:
- ✓ We need another notation to represent machine ins
- ✓ Assembly Language notation
- ✓ Ex 1: Load R1, LOC
- ✓ Ex 2: Add R3, R1, R2

Difference between RISC and CISC Instruction sets

- ✓ RISC:
- ✓ Each instruction occupies exactly one word in memory
- ✓ All operands involved in an arithmetic or logic operation must either be in processor registers
- ✓ Memory operands are accessed only using Load and Store instructions
- ✓ CISC:
- ✓ Instructions may span more than one word of memory
- ✓ Not all operands involved in an arithmetic or logic operation must either be in processor registers
- ✓ Move in place of Load and Store instructions

Consider the statement

$$C = A + B$$

- ✓ RTN is $C \leftarrow [A] + [B]$
- ✓ RISC program:

Load R2, A

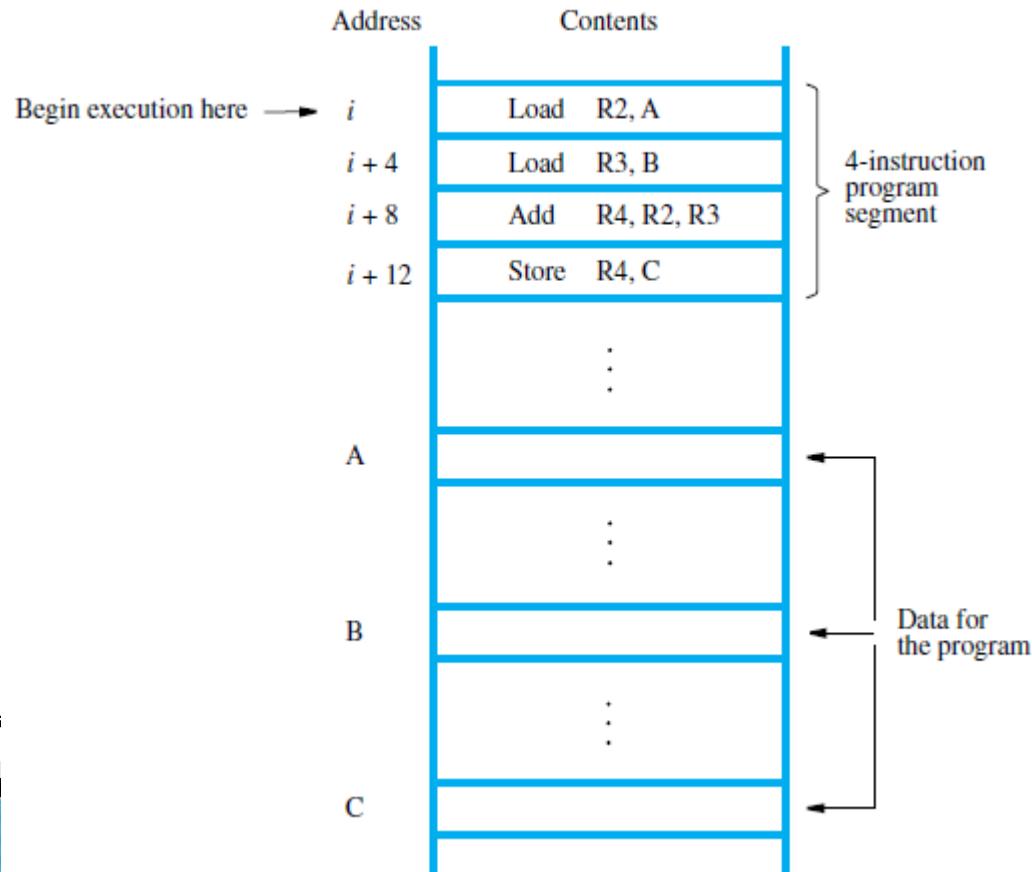
Load R3, B

Add R4, R2, R3

Store R4, C

Instruction Execution and Straight Line Sequencing

- ✓ Program segment for $C \leftarrow [A] + [B]$,



Continued...

- ✓ Addresses of 4 instructions are $i, i+4, i+8, i+12$
- ✓ Memory is byte addressable and 1 word=4 bytes.
- ✓ How the program is executed?
- ✓ To begin execution, the address of first instruction must be placed in PC
- ✓ With this instructions are fetched and executed one after the other
- ✓ Straight Line Sequencing
- ✓ During execution of every instruction PC is incremented by 4 to point to next instruction to be fetched and executed
- ✓ How an instruction is executed?

Continued...

- ✓ 2 phases
- ✓ Instruction fetch: Instruction is fetched from memory location and placed in IR of processor
- ✓ Instruction execution: Instruction is examined for operation and it is performed
- ✓ PTO

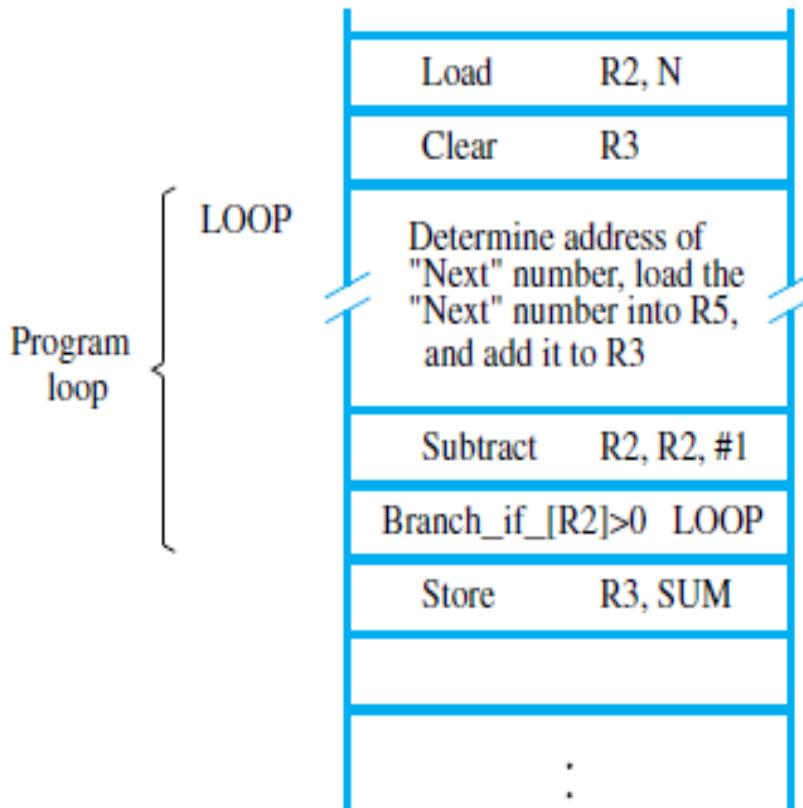
Continued...

- ✓ Branching:
- ✓ Problem: Task of adding a list of n numbers
- ✓ 1: Straight Line program

i	Load	R2, NUM1
$i + 4$	Load	R3, NUM2
$i + 8$	Add	R2, R2, R3
$i + 12$	Load	R3, NUM3
$i + 16$	Add	R2, R2, R3
⋮		
$i + 8n - 12$	Load	R3, NUM n
$i + 8n - 8$	Add	R2, R2, R3
$i + 8n - 4$	Store	R2, SUM
⋮		

Continued...

- ✓ Is this a good approach?
- ✓ 2: Instead of this long list, Let's put add instruction in a program loop



Continued...

- ✓ Comment---addresses of numbers, N, Sum
- ✓ Loop:
- ✓ Starts at LOOP
- ✓ Ends at Branch>0
- ✓ Each time thru loop address of the next number is determined , entry fetched and added to R3
- ✓ How ?
- ✓ Addressing modes
- ✓ N contains number of numbers to be added
- ✓ Moved to R2, R2 is used as counter that determines no of times loop is executed
- ✓ R2 is decremented by 1 each time in the loop
- ✓ Loop is repeated till R2 becomes 0

Continued...

- ✓ Branch >0 LOOP
- ✓ Branch instruction
- ✓ Makes the control go to branch target
- ✓ How?
- ✓ Up on branching to branch target, PC is loaded with the address of the first instruction of loop
- ✓ Branch >0 LOOP is a conditional branch
- ✓ If satisfied, PC is loaded with address of first instruction of loop
- ✓ If not, PC is incremented in normal way
- ✓ How long it repeats? (Branch >0 LOOP)

Continued...

- ✓ RISC Addressing modes:
- ✓ Immediate
- ✓ Register
- ✓ Absolute
- ✓ Register indirect
- ✓ Index: Index
- ✓ Base with Index
- ✓ Base with Index and Displacement

Continued...

- ✓ CISC Addressing modes:
- ✓ Immediate
- ✓ Register
- ✓ Absolute
- ✓ Register indirect
- ✓ Index: Index
- ✓ Base with Index
- ✓ Base with Index and Displacement
- ✓ Program Counter Relative
- ✓ Additional modes: Autoincrement and Autodecrement

Continued...

- ✓ Program operates on data in memory
- ✓ Data is organized into several structures---Data Structures
- ✓ Addressing modes: Diff ways in which location of an operand is specified
- ✓ PTO

Continued...

- ✓ Immediate mode:
- ✓ Operand is given explicitly
- ✓ When the value is not going to change
- ✓ Application: Constants are represented using this mode
- ✓ Syntax: #Value
- ✓ Ex:
- ✓ Add R4, R6, #200
- ✓ Register mode:
- ✓ Name of the register that has the operand is given
- ✓ Application: Variables are represented using this mode
- ✓ Syntax: Ri
- ✓ Effective address: Address from where operand can be determined
- ✓ EA: Ri
- ✓ Ex:
- ✓ Add R4, R2, R3

Continued...

- ✓ Absolute mode:
- ✓ Name of the memory location that has the operand is given
- ✓ Application: Variables are represented
- ✓ Syntax: LOC
- ✓ EA: LOC
- ✓ Ex:
- ✓ Load R2, NUM1

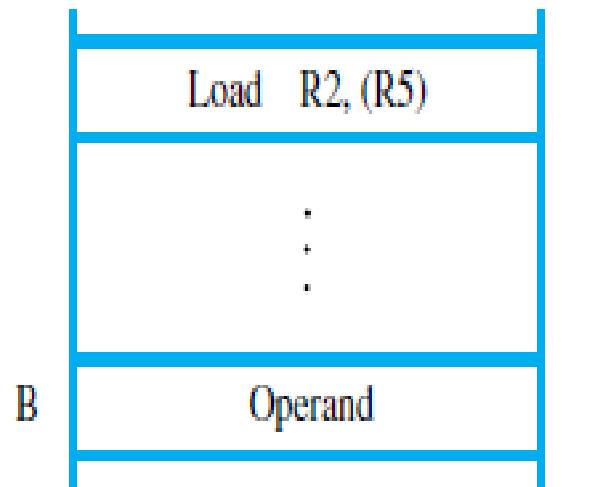
Continued...

- ✓ *Indirection and Pointers*: (Register indirect)
- ✓ Operand or its address is not given directly
- ✓ The address of operand is in a register
- ✓ Syntax: Register indirect: (Ri)
- ✓ EA: Register indirect: [Ri]
- ✓ Register or that has address is called pointer
- ✓ Ex: Next slide

Continued...



Main memory



Continued...

- ✓ Register indirect:
- ✓ First processor reads the contents of R1 and then uses B in R1 as address to obtain the operand
- ✓ APP: To add successive numbers in the list
- ✓ Next Slide

Continued...

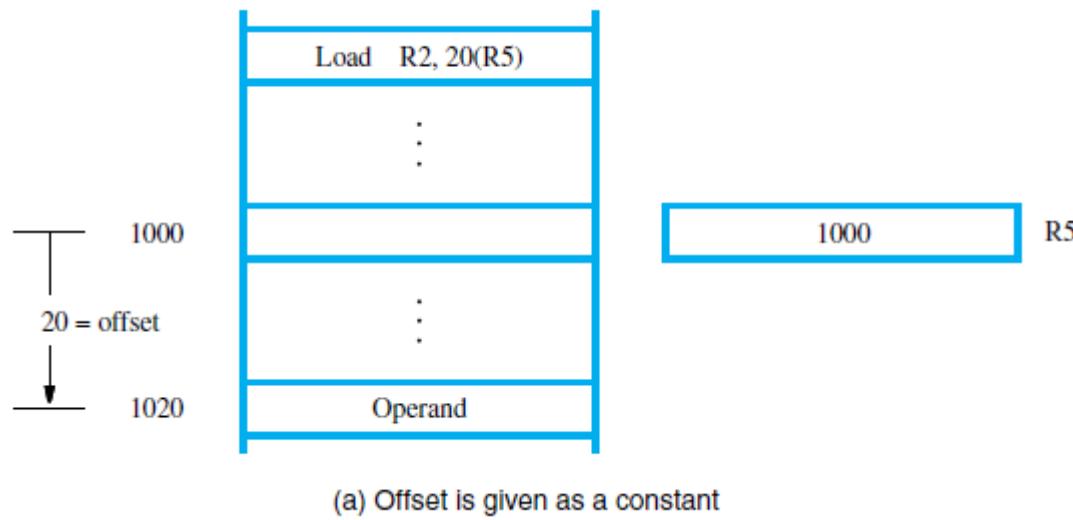


	Load	R2, N
	Clear	R3
	Move	R4, #NUM1
LOOP:	Load	R5, (R4)
	Add	R3, R3, R5
	Add	R4, R4, #4
	Subtract	R2, R2, #1
	Branch_if_[R2]>0	LOOP
	Store	R3, SUM

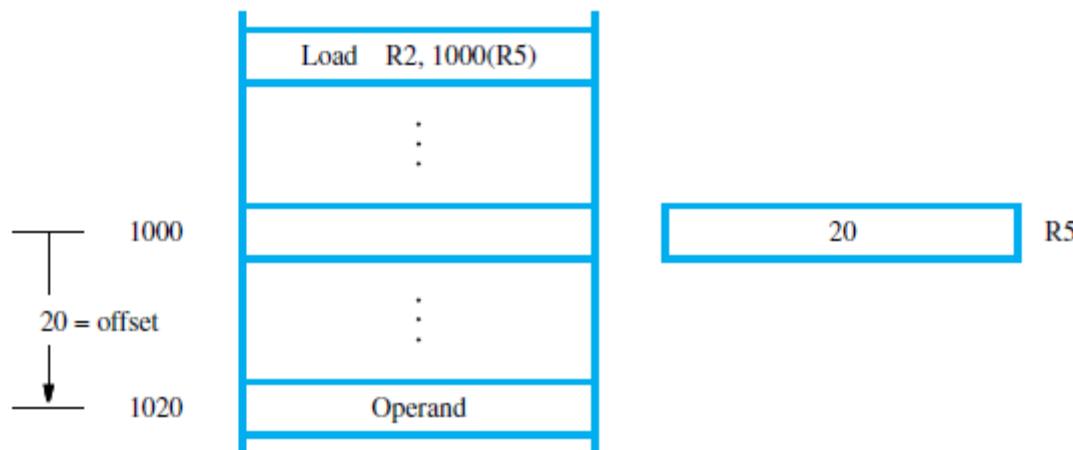
Continued...

- ✓ *Indexing and Arrays :*
- ✓ Operand or its address is not given directly
- ✓ The address of the operand is the content of register(s) together with a displacement value
- ✓ APP: To deal with arrays and lists
- ✓ Syntax: Simple Index: $X(R_i)$
- ✓ EA: $X+[R_i]$

Continued...



Continued...



Continued...

- ✓ PE: Calculating sum of Test Score 1, Test Score 2, Test Score 3 of n students and place results in Sum1,Sum2,Sum3

N
LIST
LIST+4
LIST+8
LIST+12
.....
.....
.....
.....

n
StudentID
Test1
Test2
Test3
StudentID
Test1
Test2
Test3

Continued...

- ✓ This should be thought of as 2D array
- ✓ Each row contains entry for a student
- ✓ Columns give IDs and Test scores
- ✓ Program in next slide

Continued...

	Move	R2, #LIST
	Clear	R3
	Clear	R4
	Clear	R5
	Load	R6, N
LOOP:	Load	R7, 4(R2)
	Add	R3, R3, R7
	Load	R7, 8(R2)
	Add	R4, R4, R7
	Load	R7, 12(R2)
	Add	R5, R5, R7
	Add	R2, R2, #16
	Subtract	R6, R6, #1
	Branch_if_[R6]>0	LOOP
	Store	R3, SUM1
	Store	R4, SUM2
	Store	R5, SUM3

Continued...

- ✓ Content of R4 is changing between records not within records
- ✓ In the example student ids are the reference points
- ✓ Base with Index:
- ✓ Address of operand is the content of Register Ri together with content of register Rj
- ✓ Syntax: (R_i, R_j)
- ✓ EA: $[R_i] + [R_j]$
- ✓ Actually, One will be an index and other will be a base register

Continued...

- ✓ Base index with displacement
- ✓ Address of operand is sum of contents of 2 registers together with a displacement
- ✓ Syntax: $X(R_i, R_j)$
- ✓ EA: $X + [R_i] + [R_j]$
- ✓ PTO

Continued...

- ✓ CISC Instruction Sets:
- ✓ Most arithmetic and logic instructions use the two-address format
- ✓ Operation destination, source
- ✓ Ex1:
 - ✓ Add B, A ; $B \leftarrow [A] + [B]$
- ✓ Ex2:
 - ✓ Move C, B
 - ✓ Add C, A ; $C \leftarrow [A] + [B]$
- ✓ The general form of the Move instruction is
- ✓ Move destination, source

Continued...

- ✓ *Relative:*
- ✓ EA of operand is relative to the content of index register
- ✓ Here the difference is EA of operand is relative to content of PC
- ✓ Syntax: $X(PC)$
- ✓ EA: $X+[PC]$
- ✓ X is a signed number because the branch target may be above or below branch instruction

Continued...

- ✓ *Additional Modes:*
- ✓ *Auto increment, Auto decrement modes*
- ✓ Autoincrement: EA of operand is the contents of a register
- ✓ After the operand is accessed, the content of register is automatically incremented by no of bytes that make up a word if I want successive words to be accessed
- ✓ If I want successive bytes to be accessed then the content of the register will be incremented by 1
- ✓ Syntax: $(R_i)^+$
- ✓ EA: $[R_i]$ and R_i is incremented

Continued...

- ✓ Autodecrement mode:
- ✓ EA: It is obtained by decrementing the content of register first and then taking the decremented value as address of the operand
- ✓ Words? Bytes?
- ✓ Syntax: -(Ri)
- ✓ EA: Decrement Ri and then EA=[Ri]
- ✓ These modes can be used to implement stack

Continued...

- ✓ Condition Codes:
- ✓ Info about results of various A/L operations must be kept track off
- ✓ Whether result has generated carry, overflow, result is zero/negative
- ✓ They are recorded in individual bits called condition code flags
- ✓ These flags are grouped in a special register called condition code register/flag register/status register
- ✓ These flags take values 0 or 1

Continued...

- ✓ Flags:
- ✓ N-----set to 1 if result is negative
- ✓ Z-----Set to 1 if result is 0
- ✓ V----- set to 1 if result has overflowed
- ✓ C----- Set to 1 if a carry results out of A/L operation
- ✓ These flags are affected as a result of A/L operations
- ✓ Branch>0 tests N and Z flags to cause a branch

Continued...

```
          Move      R2, N
          Clear     R3
          Move      R4, #NUM1
LOOP:   Add       R3, (R4) +
          Subtract  R2, #1
          Branch>0 LOOP
          Move      SUM, R3
```

Continued...

	Move	R2, #AVEC	R2 points to vector A.
	Move	R3, #BVEC	R3 points to vector B.
	Load	R4, N	R4 serves as a counter.
	Clear	R5	R5 accumulates the dot product.
LOOP:	Load	R6, (R2)	Get next element of vector A.
	Load	R7, (R3)	Get next element of vector B.
	Multiply	R8, R6, R7	Compute the product of next pair.
	Add	R5, R5, R8	Add to previous sum.
	Add	R2, R2, #4	Increment pointer to vector A.
	Add	R3, R3, #4	Increment pointer to vector B.
	Subtract	R4, R4, #1	Decrement the counter.
	Branch_if_[R4]>0	LOOP	Loop again if not done.
	Store	R5, DOTPROD	Store dot product in memory.

Figure 2.27 A RISC-style program for computing the dot product of two vectors.

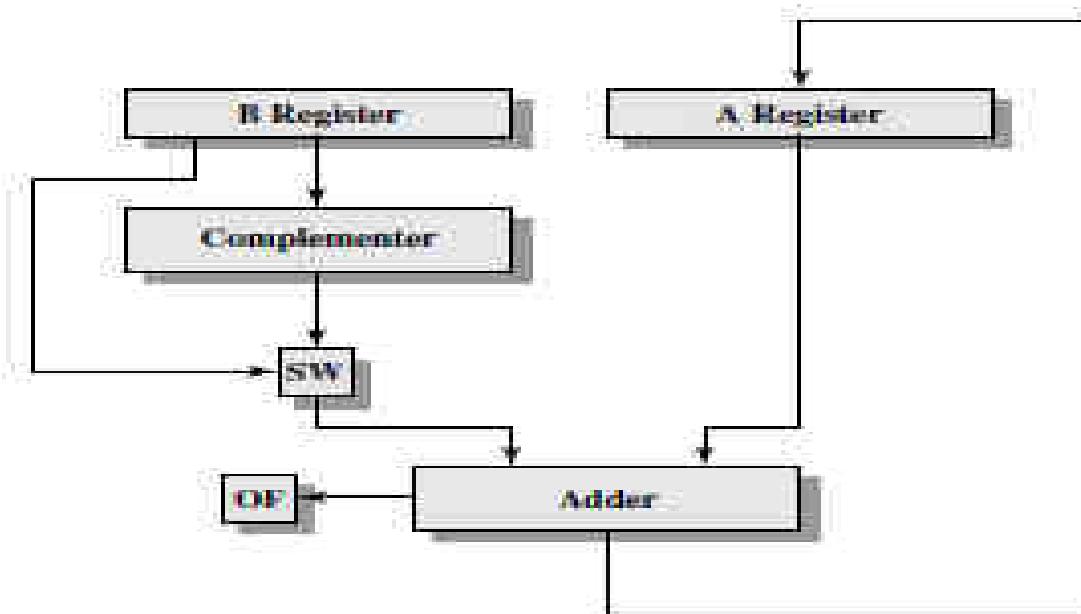
Continued...

	Move	R2, #AVEC	R2 points to vector A.
	Move	R3, #BVEC	R3 points to vector B.
	Move	R4, N	R4 serves as a counter.
	Clear	R5	R5 accumulates the dot product.
LOOP:	Move	R6, (R2)+	Compute the product of next components.
	Multiply	R6, (R3)+	
	Add	R5, R6	Add to previous sum.
	Subtract	R4, #1	Decrement the counter.
	Branch>0	LOOP	Loop again if not done.
	Move	DOTPROD, R5	Store dot product in memory.

Figure 2.28 A CISC-style program for computing the dot product of two vectors.

ARITHMETIC AND LOGIC UNIT

Hardware implementation for Addition and Subtraction:



OF — Overflow bit

SW — Switch (select addition or subtraction)

Figure 9.6 Block Diagram of Hardware for Addition and Subtraction

Multiplication

UNSIGNED INTEGERS

Multiplicand (11)

1011 X 1101

1011
0000
1011
1011

Multiplier (13)

10001111

Product (143)

C	A	Q	M	
D	0000	1101	1011	Initial values
D	1011	1101	1011	Add } First cycle
D	0101	1110	1011	Shift } Second cycle
D	1101	1111	1011	Add } Third cycle
D	0110	1111	1011	Shift } Fourth cycle
I	0001	1111	1011	Add }
D	1000	1111	1011	Shift } cycle

(b) Example from Figure 9.7 (product in A, Q)

Figure 9.8 Hardware Implementation of Unsigned Binary Multiplication

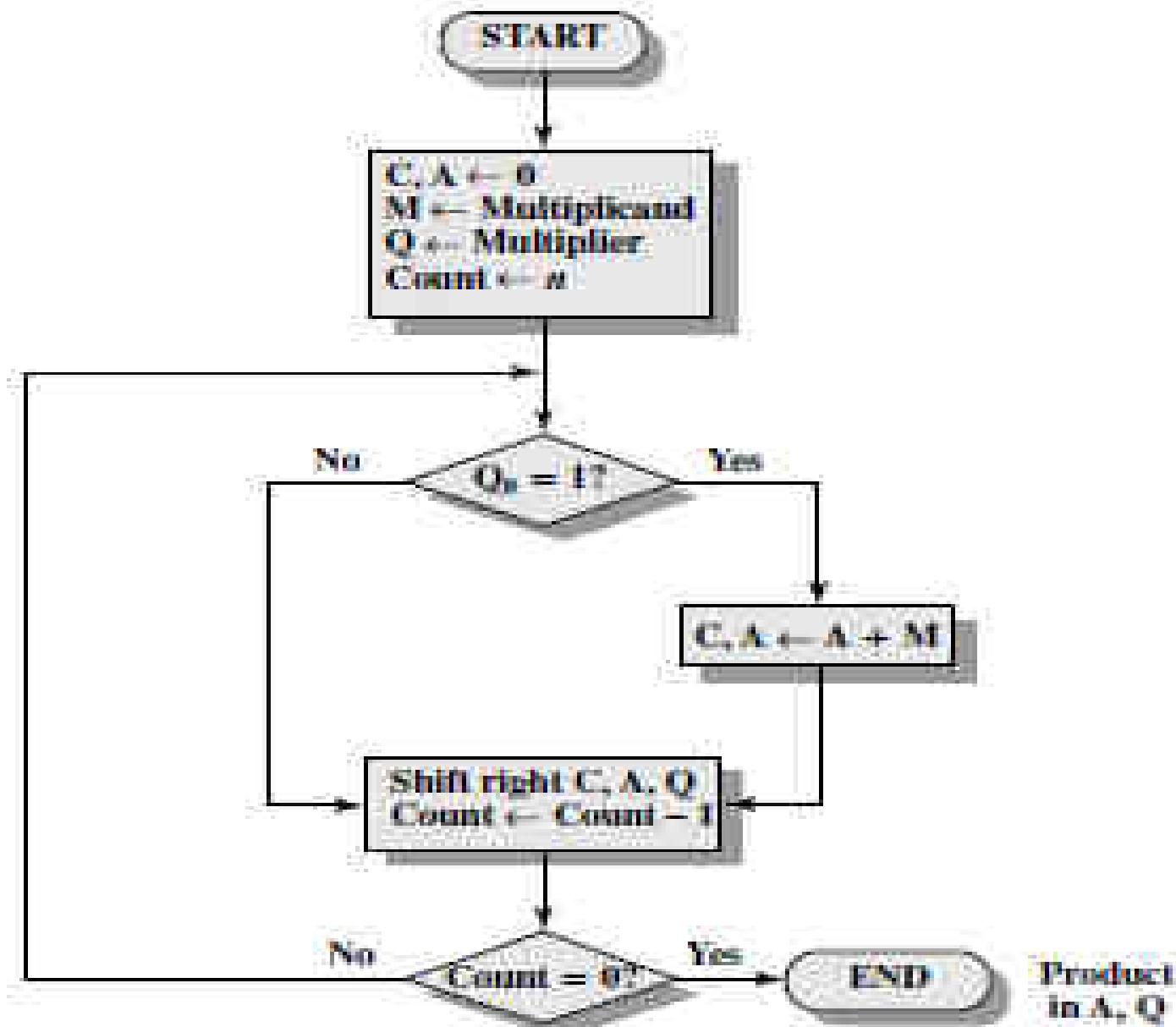
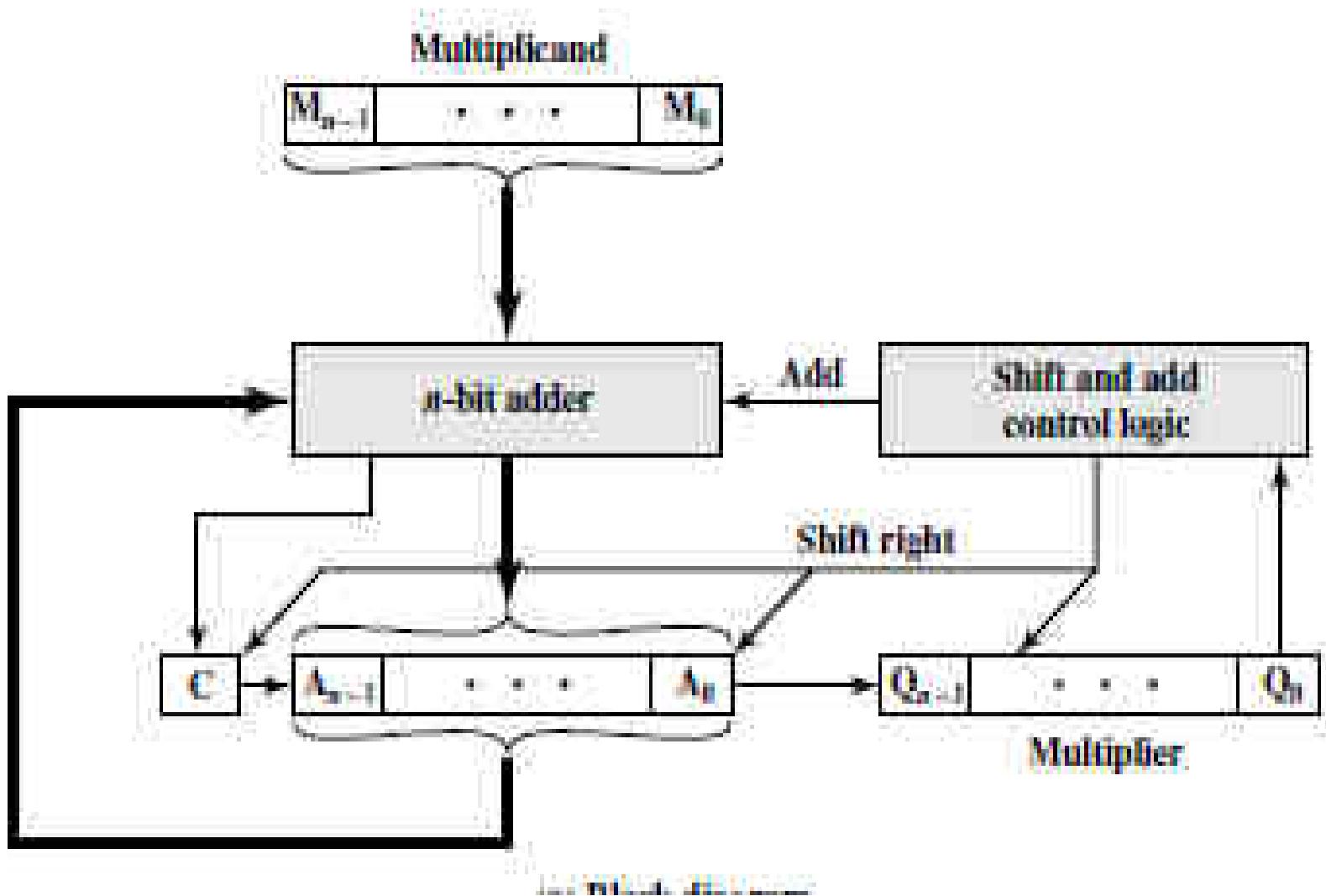


Figure 9.9 Flowchart for Unsigned Binary Multiplication



TWOS COMPLEMENT MULTIPLICATION

$$\begin{array}{r} 1011 \times 1101 \\ \hline 1011 \\ 0000 \\ 1011 \\ 1011 \\ \hline 10001111 \end{array}$$

1011	$\times 1101$	$1011 \times 1 \times 2^0$
00001011	$1011 \times 0 \times 2^1$	00000000
00101100	$1011 \times 1 \times 2^2$	00101100
01011000	$1011 \times 1 \times 2^3$	01011000
		10001111

Figure 9.10 Multiplication of Two Unsigned 4-Bit Integers Yielding an 8-Bit Result

Straightforward multiplication will not work if both the multiplicand and multiplier are negative.

In fact, it will not work if either the multiplicand or the multiplier is negative.

$\begin{array}{r} 1001 \quad (9) \\ \times 0011 \quad (3) \\ \hline 00001001 \quad 1001 \times 2^0 \\ 00010010 \quad 1001 \times 2^1 \\ \hline 00011011 \quad (27) \end{array}$	$\begin{array}{r} 1001 \quad (-7) \\ \times 0011 \quad (3) \\ \hline 11111001 \quad (-7) \times 2^0 = (-7) \\ 11110010 \quad (-7) \times 2^1 = (-14) \\ \hline 11101011 \quad (-21) \end{array}$
---	--

(a) Unsigned integers

(b) Twos complement integers

Figure 9.11 Comparison of Multiplication of Unsigned and 2's Complement Integers

Booth's algorithm:

Adv: Benefit of speeding up the multiplication process, relative to a more straightforward approach.

The multiplier in Reg Q and multiplicand in M are placed.

There is a 1-bit register Q_1 placed logically to the right of the least significant bit of the Q register.

The results of the mul will appear in A & Q regs.

A and Q_1 are initialized to 0.

A	Q	Q_1	M	
0000	0011	0	0111	Initial values
1001	0011	0	0111	$A \leftarrow A - M$
1100	1001	1	0111	Shift
1110	0100	1	0111	Shift
0101	0100	1	0111	$A \leftarrow A + M$
0010	1010	0	0111	Shift
0001	0101	0	0111	Shift

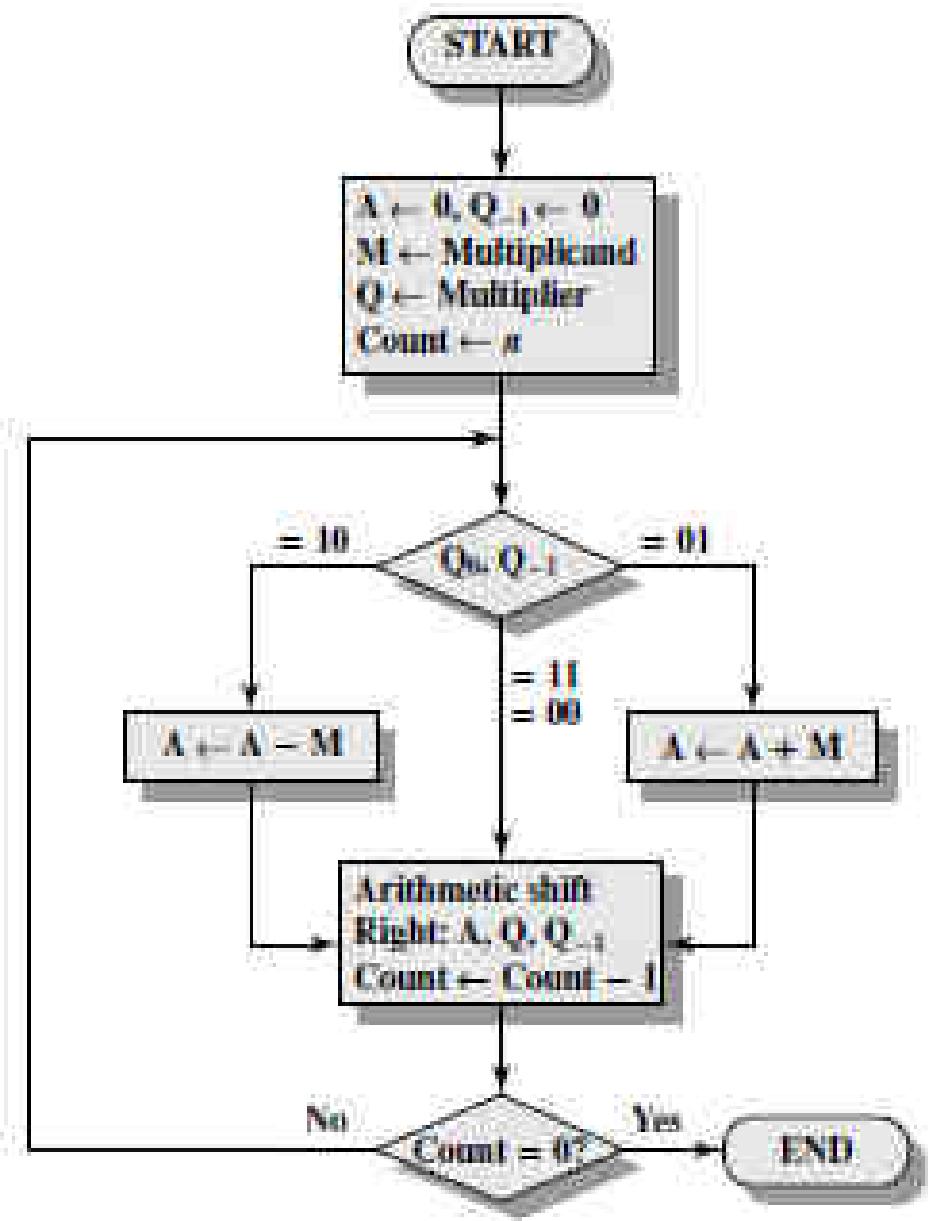


Figure 9.12 Booth's Algorithm for 2's Complement Multiplication

Consider a +ve multiplier $00011110 = 2^4 + 2^3 + 2^2 + 2^1 = 30$

Is also equal to $2^5 - 2^1$

The number of such operations can be reduced to two if we observe that

$$2^n + 2^{n-1} + \dots + 2^{n-k} = 2^{n+1} - 2^{n-k} \quad \text{----Eq 1}$$

So the product can be generated by one addition and one subtraction of the multiplicand.

This scheme extends to any number of blocks of 1s in a multiplier.

$$\begin{aligned} M \times (01111010) &= M \times (2^6 + 2^5 + 2^4 + 2^3 + 2^1) \\ &= M \times (2^7 - 2^6 + 2^5 + 2^3 - 2^1) \end{aligned}$$

The same scheme works for a negative multiplier.

Let X be a negative number in twos complement notation. The leftmost bit of X is 1, because X is negative:

$$\text{Representation of } X = \{1x_{n-1}x_{n-2} \dots x_1x_0\}$$

Then the value of X can be expressed as

$$X = -2^{n-1} + (x_{n-2} \times 2^{n-2}) + (x_{n-3} \times 2^{n-3}) + \dots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

Assume that the leftmost 0 is in the k th position. Thus, X is of the form

$$\text{Representation of } X = \{111 \dots 10x_{k-1}x_{k-2} \dots x_1x_0\}$$

Then the value of X is

$$X = -2^{n-1} + 2^{n-2} + \dots + 2^{k+1} + (x_{k-1} \times 2^{k-1}) + \dots + (x_0 \times 2^0) \quad \text{---2}$$

From Equation (1), we can say that

$$-2^{n-1} + 2^{n-2} + \dots + 2^{k+1} = 2^{n-1} - 2^{k+1}$$

Rearranging,

$$-2^{n-1} + 2^{n-2} + 2^{n-3} + \dots + 2^{k+1} = -2^{n-1} \quad \text{---3}$$

Substituting Eqn 3 in eqn 2 we have

$$X = -2^{n-1} + (x_{k-1} \times 2^{k-1}) + \dots + (x_0 \times 2^0)$$

Look at page 324, how -6 is handled with above principle.

DIVISION

Dividend = 147

Divisor = 11

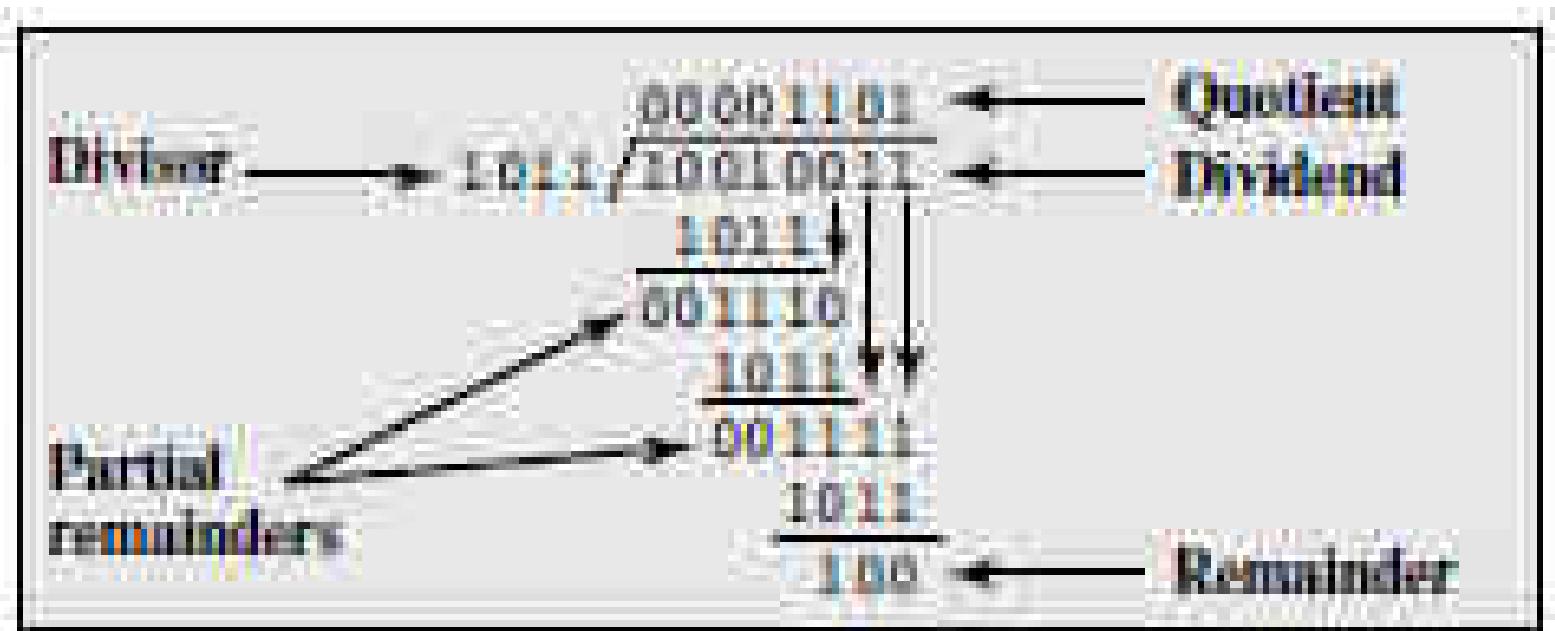


Figure 9.15 | Example of Division of Unsigned Binary Integers

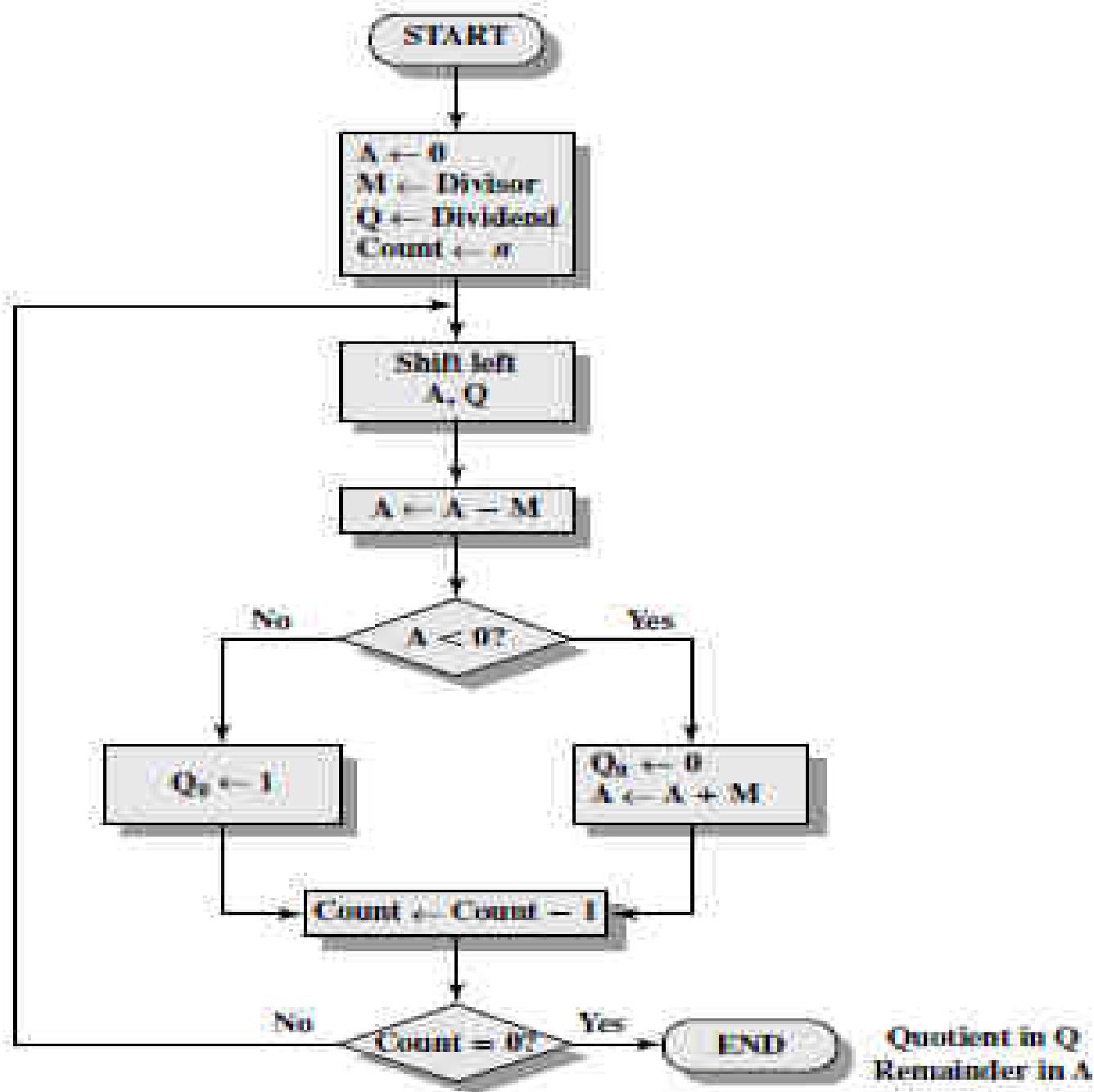


Figure 9.16: Flowchart for Unsigned Binary Division

A	Q	Initial Value
0000	0111	
0000	1110	Shift
1101		Use two's complement of 0011 for subtraction
1101		Subtract
0000	1110	Restore, set $Q_0 = 0$
0001	1100	Shift
1101		Subtract
1110		Subtract
0001	1100	Restore, set $Q_0 = 0$
0001	1000	Shift
1101		Subtract
0000	1001	Subtract, set $Q_0 = 1$
0001	0010	Shift
1101		Subtract
1110		Subtract
0001	0010	Restore, set $Q_0 = 0$

Figure 9.17 Example of Restoring Two's Complement Division (7/7)

To deal with -ve numbers, we recognize that

$$D = Q * V + R$$

Here D – dividend V – Divisor Q – Quotient R – Remainder

Consider eg. of int div with all possible combinations of signs of D and V:

$$D = 7 \quad V = 3 \quad Q = 2 \quad R = 1$$

$$D = 7 \quad V = -3 \quad Q = -2 \quad R = 1$$

$$D = -7 \quad V = 3 \quad Q = -2 \quad R = -1$$

$$D = -7 \quad V = -3 \quad Q = 2 \quad R = -1$$

Note that the signs of Q and R are easily derivable from the signs of D and V.
Specifically,

$$\text{sign}(R) = \text{sign}(D)$$

$$\text{sign}(Q) = \text{sign}(D) * \text{sign}(V)$$

Hence, 2's complement division is

- to convert the operands into unsigned values and,
- at the end, to account for the signs by complementation where needed.

CONTROL UNIT

Introduction

- CPU is viewed as a collection of two major components:
 - Processing section
 - Control Unit
- master clock.
- predefined sequence

Introduction (Contd..)

- Inputs are:
 - Master clock
 - Status info from processing section
 - Command signals from external agent (like RESET, ABORT)
- Outputs produced
 - Signals that drive the processing section and responses to an external environment.
- Control unit undertakes the following responsibilities:
 - **Instruction interpretation:** (CU read instr. , recognizes the instr type, gets operands and route to appropriate functional units of EU, necessary control signals are then issued to the EU to perform desired operation)
 - **Instruction sequencing:** CU det the address of next instruction to be executed and loads it on to PC.

Basic Concepts

- Basis for CU design are register transfer operations
 1. 8-bit info moved from Reg A to Reg B.

Such operation is described as $B \leftarrow A$

Declaring registers:

$A[8]$, $B[8]$, $PC[16]$

2. Reg can be defined as a portion of some other reg.

Assigning higher order byte of 16 bit PC

$PCH[8] = PC[15-8]$

3. Assigning individual bits

$B[0] = A[7]$ means msb of A is copied to lsb of B.

Basic Concepts (Contd..)

Normally two i/p's are associated with each reg:

E i/p or control i/p

Data i/p

The E i/p controls the data flow from A to B.

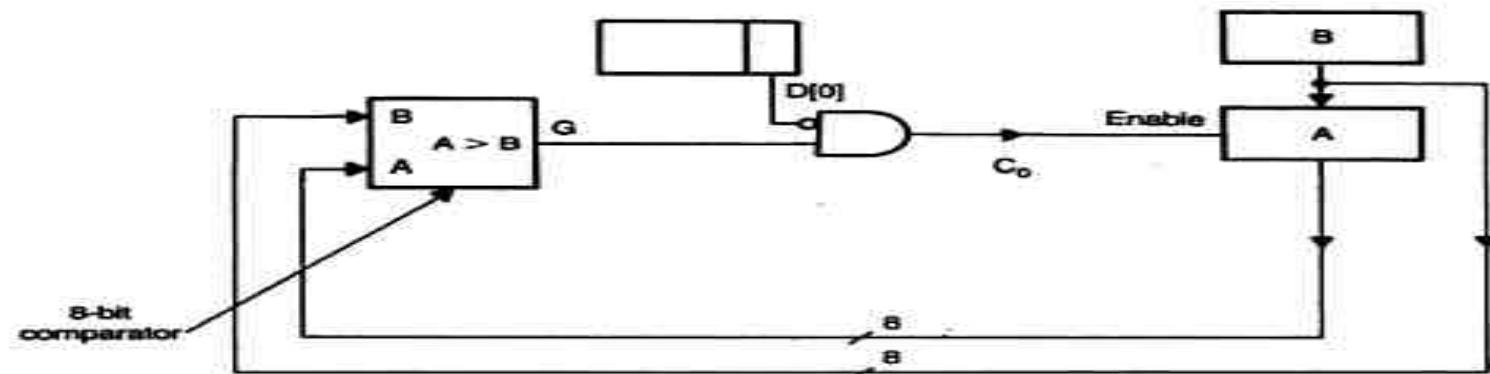
Reg B is loaded with A only when E is held high else contents of reg remain the same.

Such conditional transfer is expressed as

$$E: B \leftarrow A$$

4. Control i/p can be a fn of more than a variable.

IF $A > B$ and $D[0] = 0$ then $A \leftarrow B$



Basic Concepts (Contd..)

5. To perform reg transfer operation that involves selection.

If $x=0$ and $t=1$, then $A \leftarrow B$

else $A \leftarrow D$

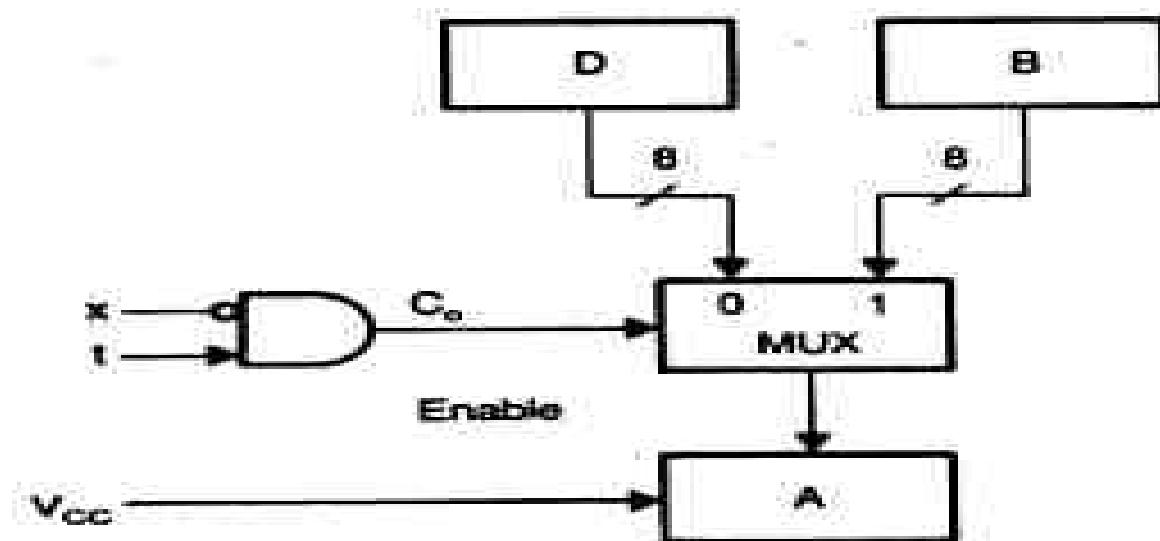
Such transfer is expressed as

$C_0 : A \leftarrow B$

$C_0' : A \leftarrow D$

Where $C_0 = x't$ and

$C_0' = (x't)' = x + t'$



MUX selects reg B if $C_0 = 1$; otherwise reg D is selected.

Basic Concepts (Contd..)

The other reg transfer operations are

$D \leftarrow A'$; Transfer the compl of A to D.

$A \leftarrow A+1$; Increment the content of A by 1.

$A \leftarrow A-1$; Decrement the content of A by 1.

$D \leftarrow AVB$; A OR B, store result in D

$D \leftarrow A \wedge B$; A AND B, store result in D

$LSR(A)$; Logical shift right

$ASR(A)$;

LSL, ASL, ROR, ROL

$A\$Q$ – used to concatenate A and Q

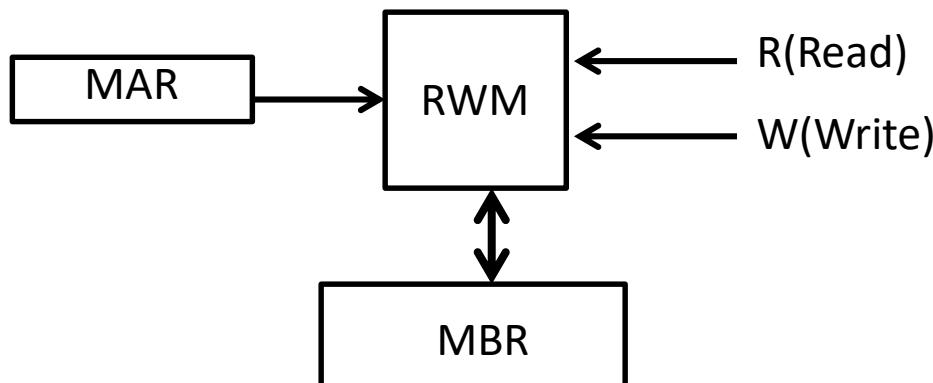
$ASR(A\$Q)$

Basic Concepts (Contd..)

- Whenever RWM is a part of processing section , MBR and MAR are associated with RWM unit.
MAR holds the address of desired mem word and MBR as buffer reg in all data transfer operations.

- R: $\text{MBR} \leftarrow M((\text{MAR}))$
- W: $M((\text{MAR})) \leftarrow \text{MBR}$

The line b/n RWM and MBR is bidirectional bus and it can be easily implemented using tristate buffers.



Basic Concepts (Contd..)

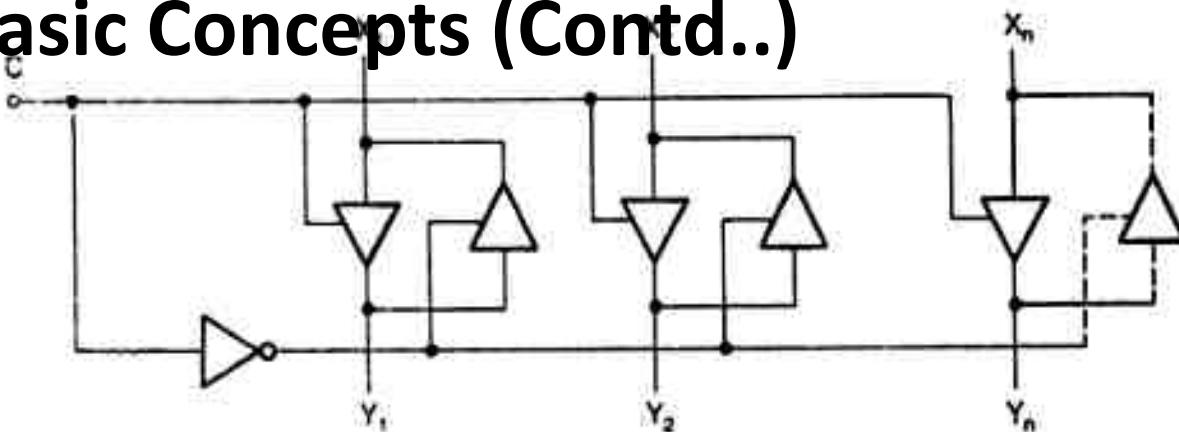


Fig: Bidirectional Data bus
When $C=1$, X to Y
When $C=0$, Y to X

6. i) Declare buses Inbus[4] and outbus[4] -- 4 bit buses

ii) $A = \text{inbus}$

means data of inbus is transferred to Reg A when next clock arrives

iii) $\text{Outbus} = B[7:4]$

Higher order 4 bits of 8 bit register B is made available on the outbus for one clock period.

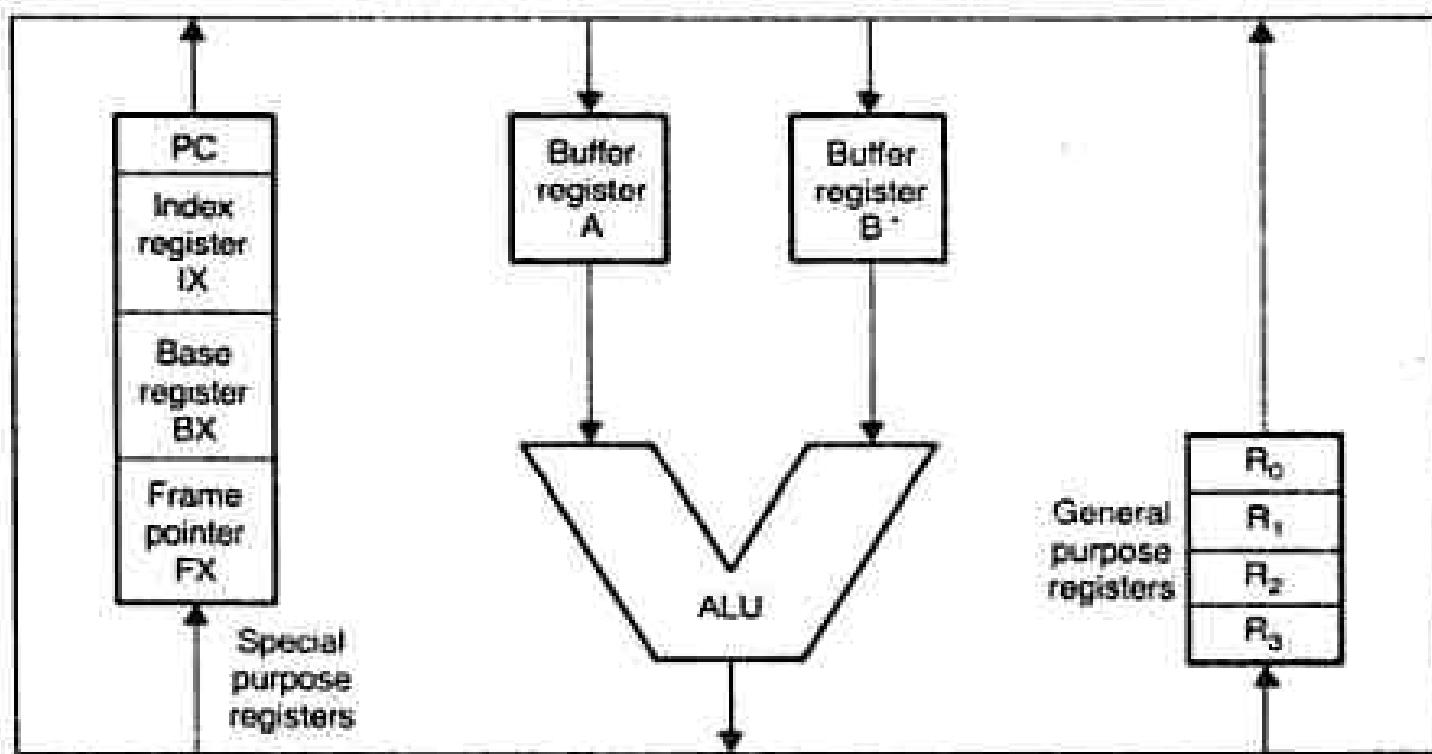
Rate at which computer performs operations (such as $A \leftarrow A+M$, $A \leftarrow A \wedge B$) is determined by bus structure.

Buses:

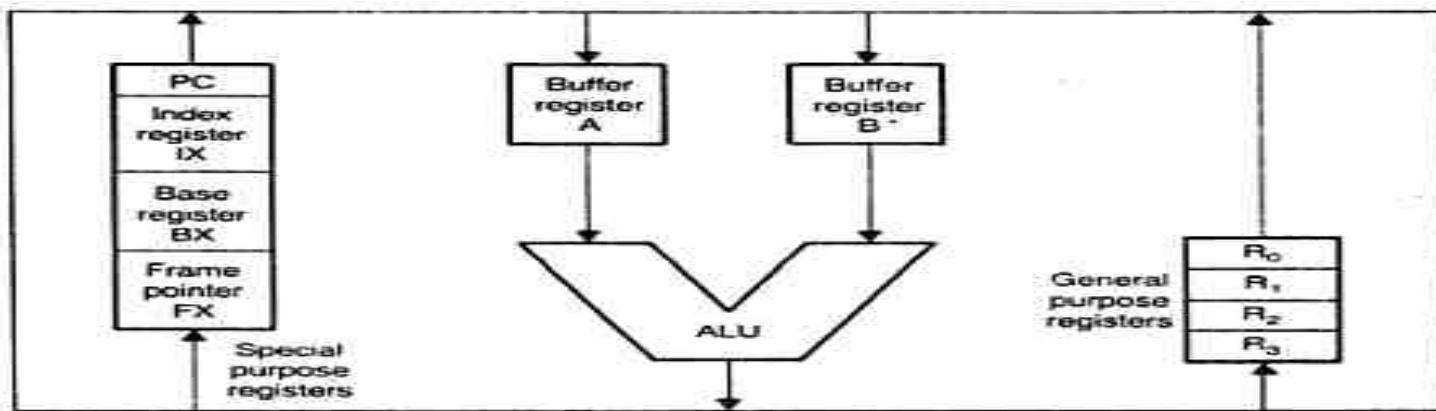
Several types of bus structure within the CPU:

- i) Single-bus oriented ALU
- ii) Two-bus oriented ALU
- iii) Three-bus oriented ALU

Single-bus oriented ALU



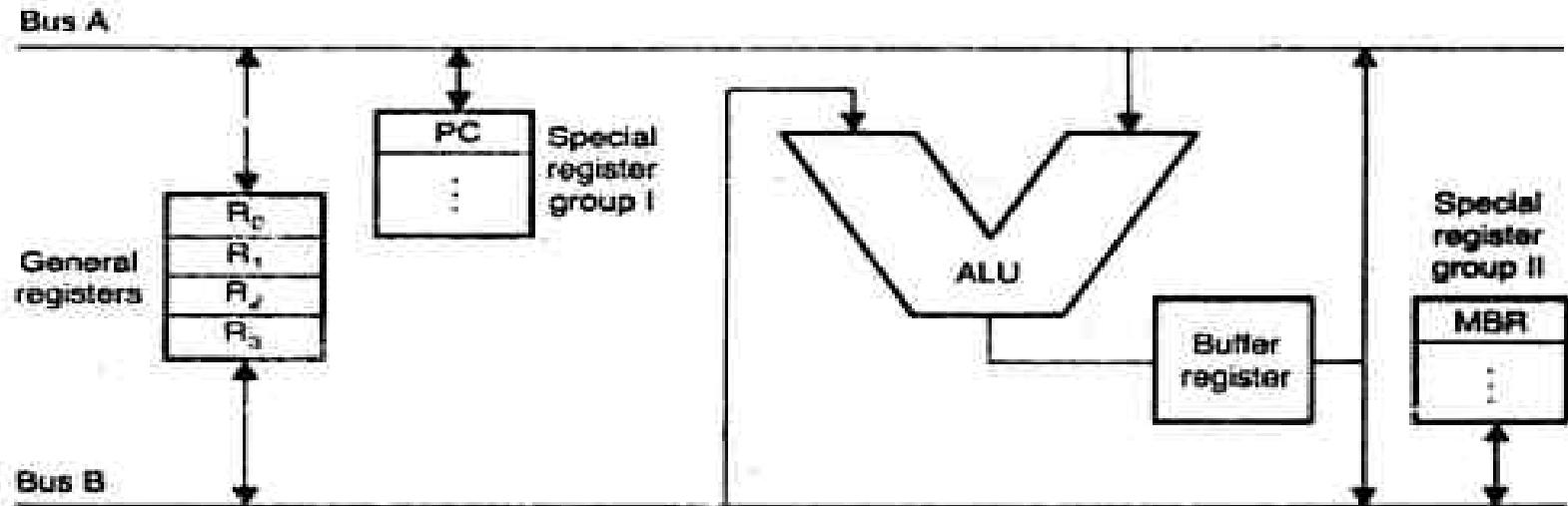
Single-bus oriented ALU (Contd..)



Disadvantages:

- Affects speed of execution of a typical 2 operand memory
- Each step given above increases the number of states in control logic. Hence more HW may be required to design control unit

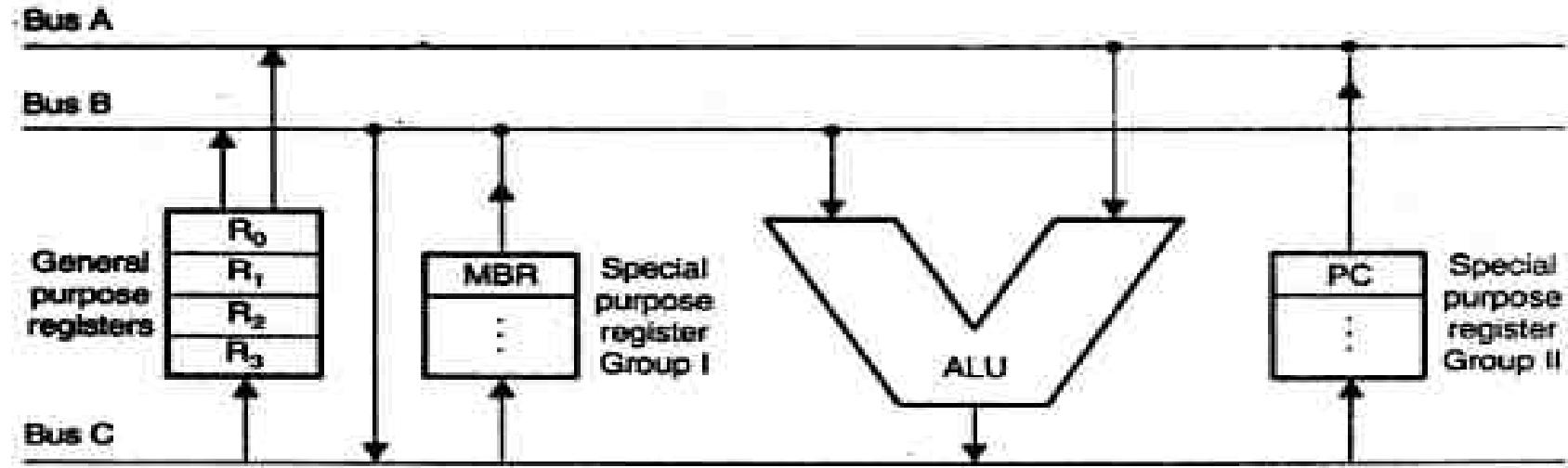
Two-bus oriented ALU



Two-bus oriented ALU (Contd..)

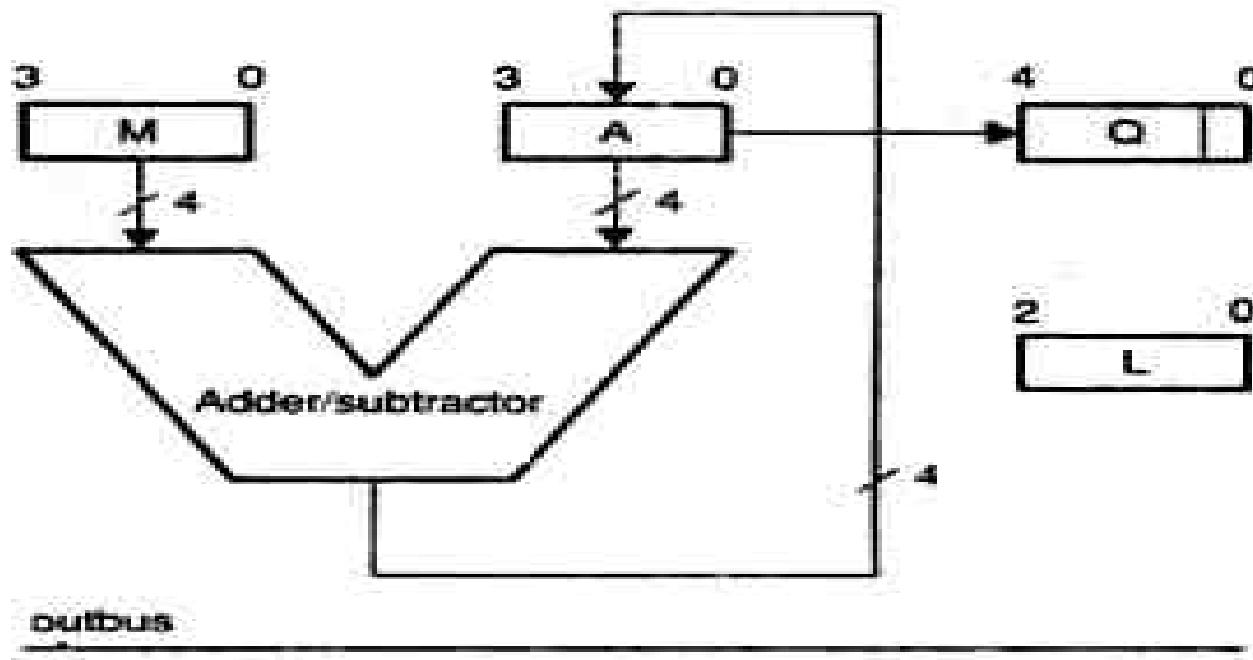
- Output Buffer register is used to prevent collision of the buses
- 1st cycle: loading operands and storing result in O/P buffer
- 2nd cycle: result in O/P buffer is pushed to bus(destination). The contents of buffer reg can be gated to either bus A or bus B.

Three-bus oriented ALU



4 X 4 two's complement Booth's multiplier

(Refiquzamman)



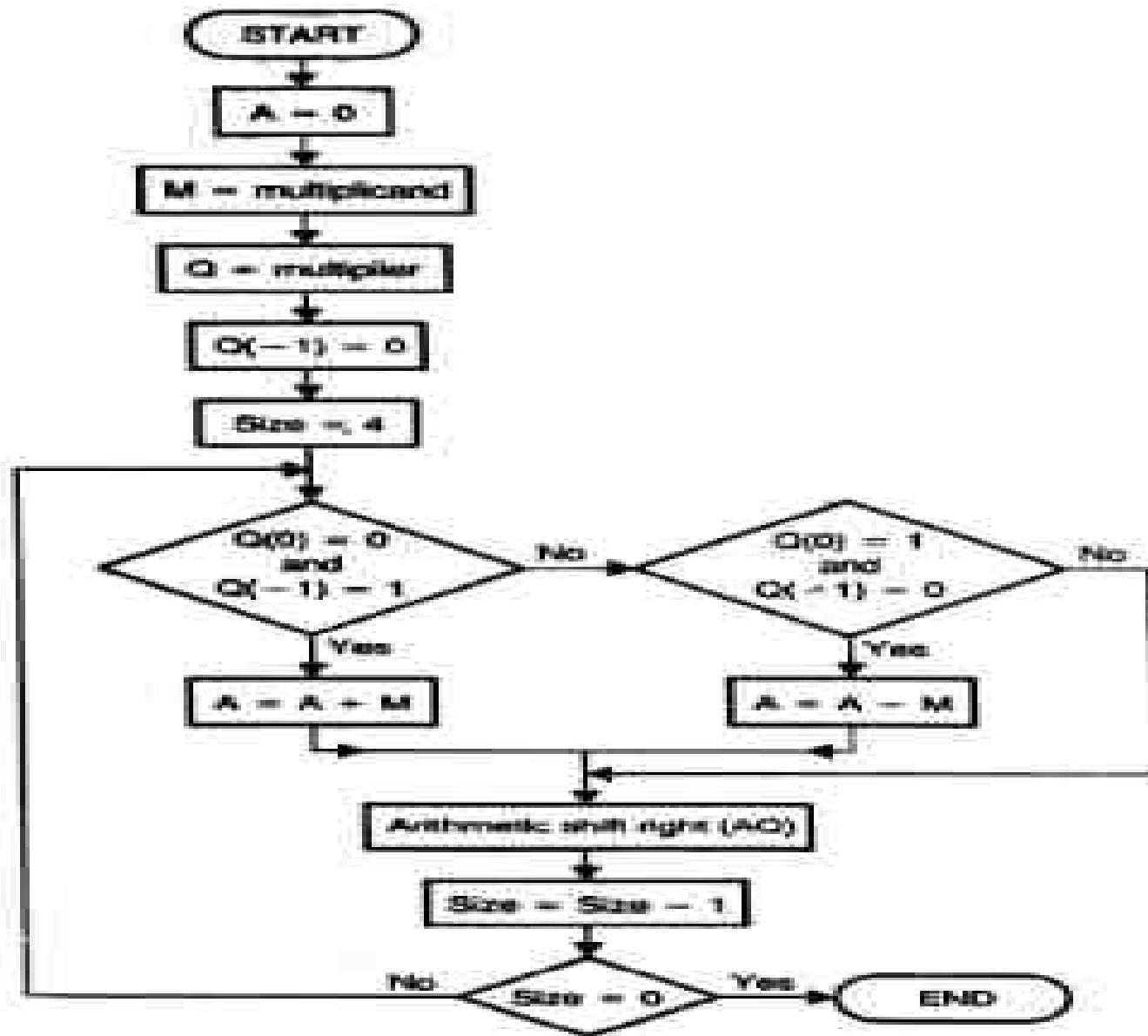
Multiplicand Bits Inspected $Q_{j-1} Q_j$	Recorded Digit Underneath Position j	Implied Action in Position j
0 0	0	None
0 1	1	Add M
1 0	1	Sub M
1 1	0	None

Figure 3-40 Booth's Recording Procedure

To see how this procedure works, the following trace is provided. Assume that $M = -4 = 1100$ and $Q = 7 = 0111$.

	M	A	Q	
Initial configuration	1100	0000	01110	$Q_{j-1} = \text{assumed } 0$ Bit pair inspected = 10
Iteration 1				
$A_1 = A - M$	1100	0100	01110	
Shift (AQ)	1100	0010	00111	Bit inspected = 11
Iteration 2				
Shift (AQ)	1100	0001	00011	Bit pair inspected = 11
Iteration 3				
Shift (AQ)	1100	0000	10001	Bit pair inspected = 01
Iteration 4				
$A_1 = A + M$	1100	1100	10001	
Shift (AQ)	1100	1110	01000	
				Product = 11100100
				= 100100
				= <u><u>-28</u></u>

4 X 4 two's complement Booth's multiplier (Contd..)



HARDWIRED APPROACH

- Control logic is a clocked sequential ckt.
- So conventional sequential ckt design procedure can be applied to build CU.
- Final circuit is obtained by physically connecting gates and flip flops.
- Cost of control logic increases with system complexity.

10 steps for hardwired control

- 1) Define task to be performed.
- 2) Propose a trial processing section.
- 3) Provide a reg tx descr algo based on processing section outlined.
- 4) Validate the algo by using trial data.
- 5) Describe the basic char of the HW elements to be used in the processing section.
- 6) Complete the design of the processing section by establishing necessary control points.
- 7) Propose the block diagram of the controller.
- 8) Specify state diagram of controller.
- 9) Specify the char of the HW elements to be used in the controller.
- 10) Complete the controller design and draw a logic diagram of final circuit.

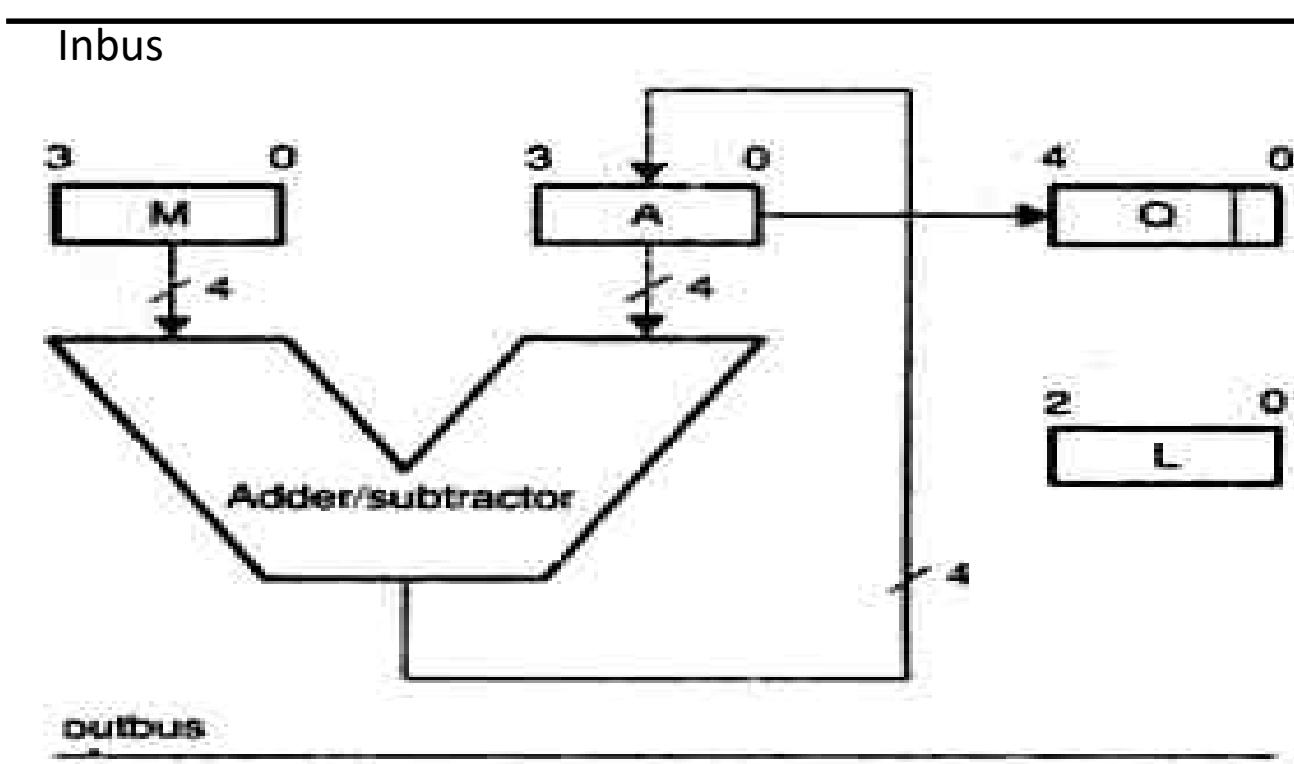
Step 1: Task definition.

Design a Booth's multiplier to multiply two 4-bit signed numbers.

Step 2: trial processing section.

$q_1 \ q_0$

- 0 0 → none
- 0 1 → add M
- 1 0 → sub M
- 1 1 → None



Step 3: register transfer description of Booth's multiplier procedure based on the processing section outlined in the previous step.

Declare registers A[4], M[4], Q[5], L[3]

Declare buses Inbus[4], outbus[4]

Start: $A \leftarrow 0$, $M \leftarrow \text{inbus}$, $L \leftarrow 4$; clear A and transfer M
 $Q[4:1] \leftarrow \text{inbus}$, $Q[0] \leftarrow 0$; transfer Q

Loop: if $Q[1:0] = 01$, then go to ADD
if $Q[1:0] = 10$, then go to SUB,
go to Rshift;

ADD: A←A+M;
 goto Rshift;

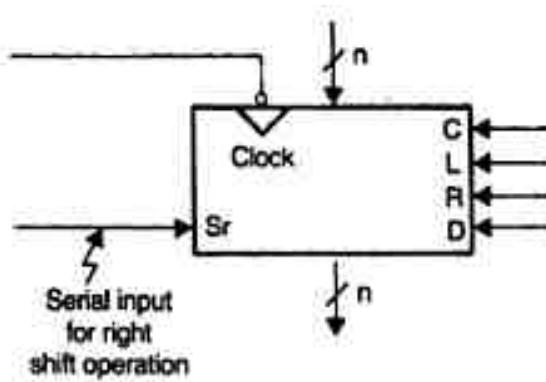
SUB: A←A-M;

Rshift: ASR(AQ), L \leftarrow L-1;
if L>0, then go to loop
outbus =A;
outbus=Q[4:1];
go to halt

Step 4: Validate the algo by using trial data.

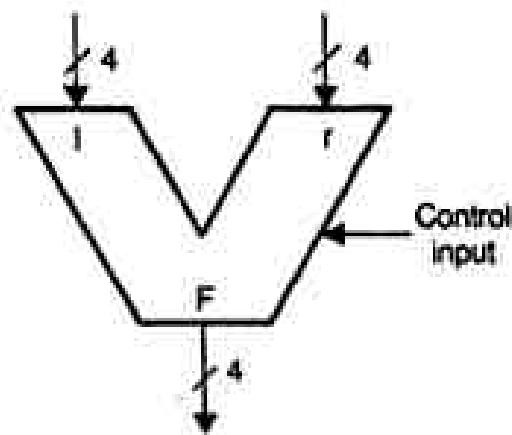
A	Q	Q_{-1}	M			
0000	0011	0	0111		Initial Values	
1001	0011	0	0111	A	A - M } First	
1100	1001	1	0111	Shift	Shift } Cycle	
1110	0100	1	0111	Shift	Shift } Second	
0101	0100	1	0111	A	A + M } Third	
0010	1010	0	0111	Shift	Shift } Cycle	
0001	0101	0	0111	Shift	Shift } Fourth	
					Cycle	

Step 5: Processing section includes GPRs, 4-bit adder / subtractor, Tristate buffers



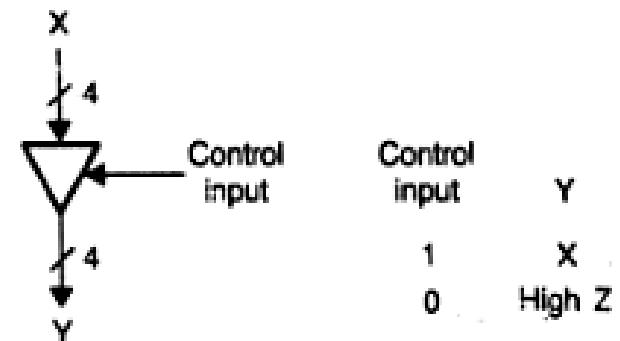
C	L	R	D	Clock	Action
1	0	0	0	+	Clear
0	1	0	0	+	Load external data
0	0	1	0	+	Right shift
0	0	0	1	+	Decrement by one
0	0	0	0	+	No change

a. Storage Register



b. Adder-subtractor

Control input	F
1	$l + r$
0	$l - r$



c. Tri-state Buffer

Figure 4.17 Characteristics of the Component Parts Used in the Processing Section of the Booth's Multiplier

Step 6: The complete design of processing section establishing control points.

$C_0: A \leftarrow 0$
 $C_1: M \leftarrow \text{Inbus}$
 $C_2: L \leftarrow 4$
 $C_3: Q[4:1] \leftarrow \text{Inbus}$
 $C_4: Q[0] \leftarrow 0$
 $F = 1 + r$
 $F = 1 - r$
 $A \leftarrow F$
 $\text{ASR}(A \text{ } S \text{ } O)$
 $L \leftarrow L - 1$
 $\text{Outbus} = A$
 $\text{Outbus} = Q[4:1]$

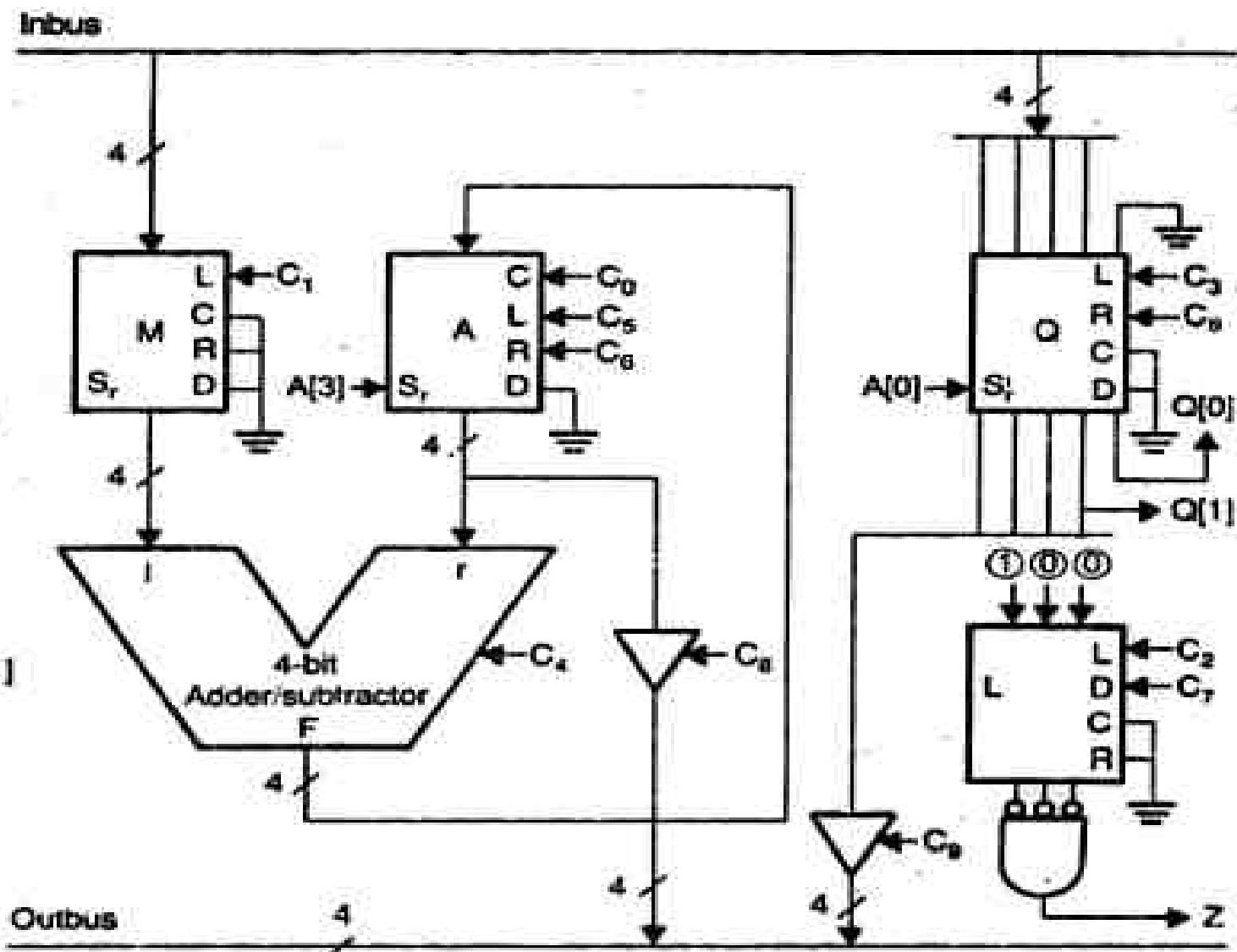


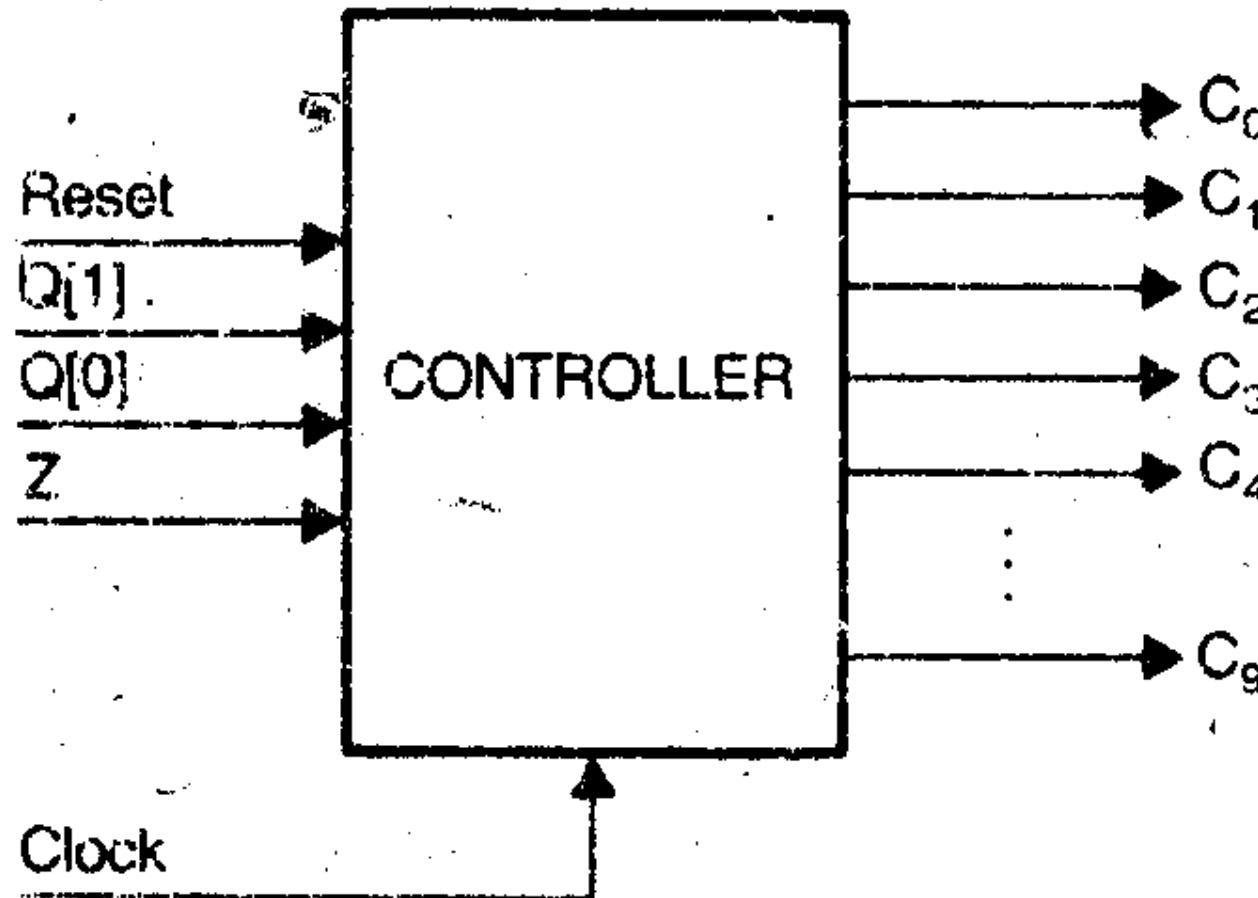
Figure 4.18 Processing Section of the Booth's Multiplier

Step 7: Block diagram of controller:

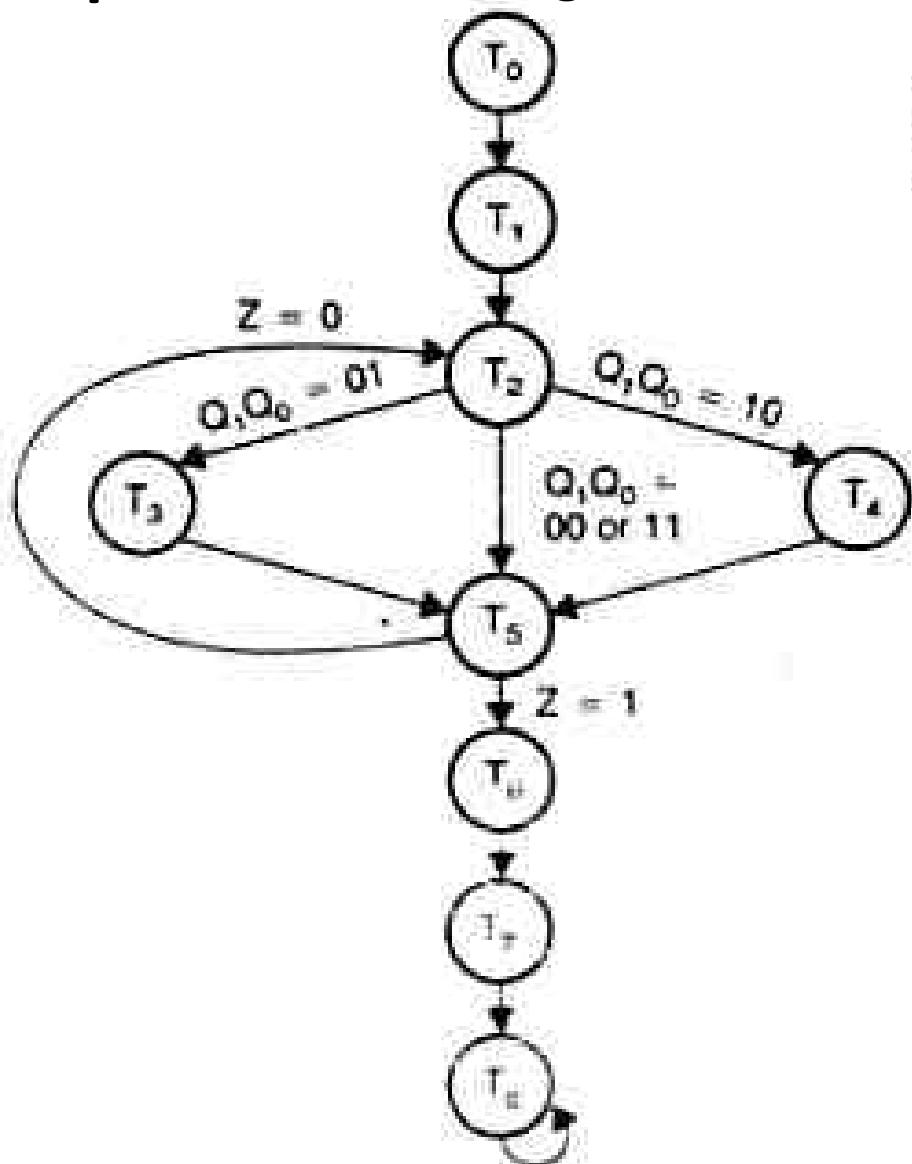
will have 5 i/p's and 10 o/p's.

RESET i/p is used to reset the controller so a new computation can begin.

CLK is used to synch the controller action for trailing edge of clock pulse.



Step 8: The state diagram of Booth's multiplier controller



a. State Diagram

CONTROL STATE	OPERATION PERFORMED	CONTROL SIGNALS TO BE ACTIVATED
T_0	$A \leftarrow 0, L \leftarrow 4,$ $M \leftarrow \text{Inbus}$	C_0, C_1, C_2
T_1	$Q [4:1] \leftarrow \text{Inbus},$ $Q [0] \leftarrow 0$	C_3
T_2	None	None
T_3	$A \leftarrow A + M$	C_4, C_5
T_4	$A \leftarrow A - M$	C_5 ($C_4 = 0$)
T_5	ASR (ASQ), $L \leftarrow L - 1$	C_6, C_7
T_6	$\text{Outbus} = A$	C_8
T_7	$\text{Outbus} =$ $Q [4:1]$	C_9
T_8	None	None

b. Controller Action

Step 9: The controller includes a mod -16 counter, a 4: 16 decoder, a sequence controller (SC).

Step 9: The controller includes a mod -16 counter, a 4: 16 decoder, a sequence controller (SC).

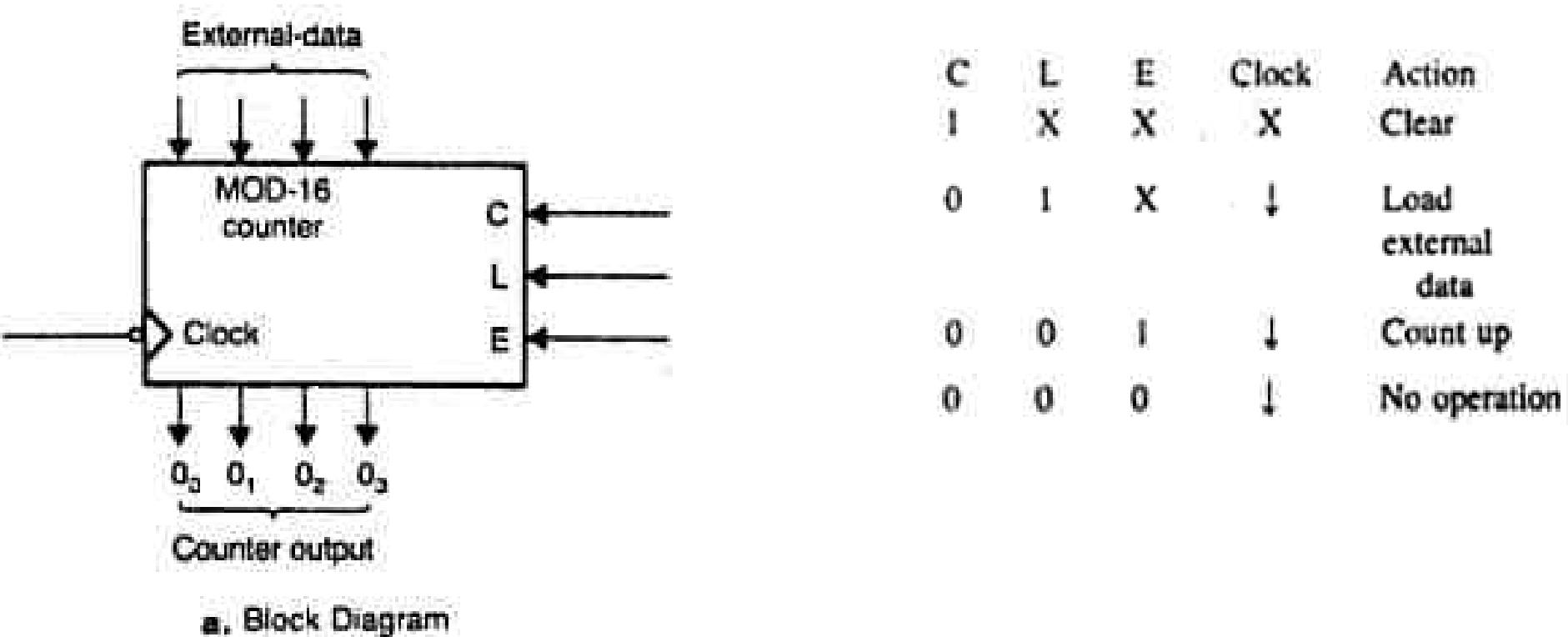


Figure 4.22 Characteristics of the Counter Used in the Controller Design

SC HW, which sequences the controller according to state diagram.

Hence TT for SC must be derived from the controller's state diagram.

Step 10: The multiplier controller with its logic diagram

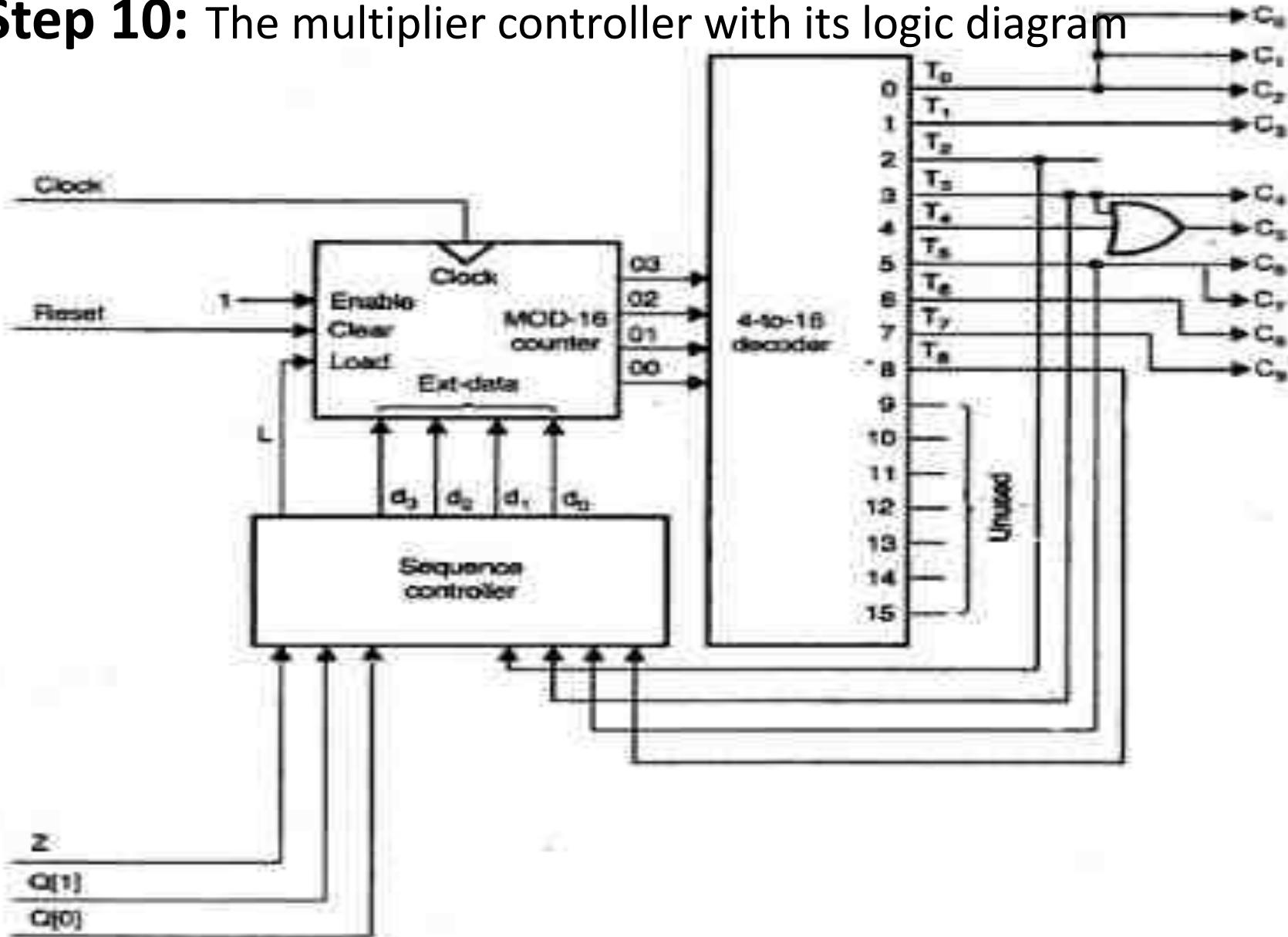


Figure 4.23 Logic Diagram of the Booth's Multiplier Controller

External data

	Q101	Q102	Q103	Q104	Q105	Q106	Q107
0	-	-	-	-	-	-	-
1	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-
3	-	-	-	-	-	-	-
4	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-
6	-	-	-	-	-	-	-
7	-	-	-	-	-	-	-
8	-	-	-	-	-	-	-
9	-	-	-	-	-	-	-
10	-	-	-	-	-	-	-

	Q101	Q102	Q103	Q104	Q105	Q106	Q107
0	X	X	X	X	X	X	X
1	X	X	X	X	X	X	X
2	X	X	X	X	X	X	X
3	X	X	X	X	X	X	X
4	X	X	X	X	X	X	X
5	X	X	X	X	X	X	X
6	X	X	X	X	X	X	X
7	X	X	X	X	X	X	X
8	X	X	X	X	X	X	X
9	X	X	X	X	X	X	X
10	X	X	X	X	X	X	X

Implementing SC using PLA:

$$P_0 = Q[1]^T Q[0]^T T_2$$

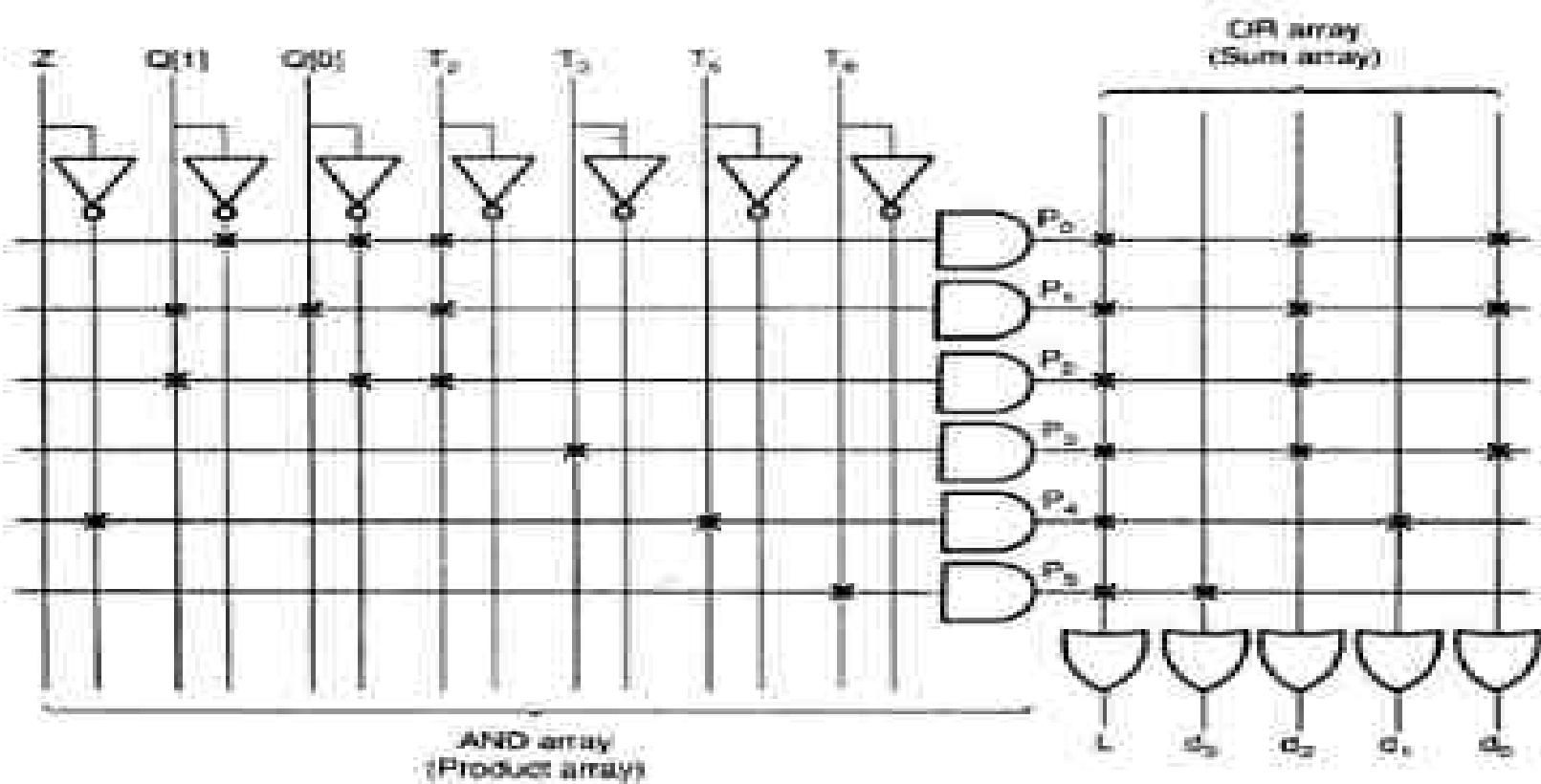
$$P_1 = Q[1] Q[0] T_2$$

$$P_2 = Q[1] Q[0]^T T_2$$

$$P_3 = T_3$$

$$P_4 = Z' T_5$$

$$P_5 = T_8$$



b. PLA Implementation

Figure 4.24 Sequence Controller Design

The PLA o/p's are summarized as

$$L = P_0 + P_1 + P_2 + P_3 + P_4 + P_5$$

$$d_3 = P_5$$

$$d_2 = P_0 + P_1 + P_2 + P_3$$

$$d_1 = P_4$$

$$d_0 = P_0 + P_1 + P_3$$

The controller design is completed by relating the control unit (T0-T8) with control i/p's C0-C9 as below:

$$C_0 = C_1 = C_2 = T_0$$

$$C_3 = T_1$$

$$C_4 = T_3$$

$$C_5 = T_3 + T_4$$

$$C_6 = C_7 = T_5$$

$$C_8 = T_6$$

$$C_9 = T_7$$

CU design using a PLA and D F/Fs

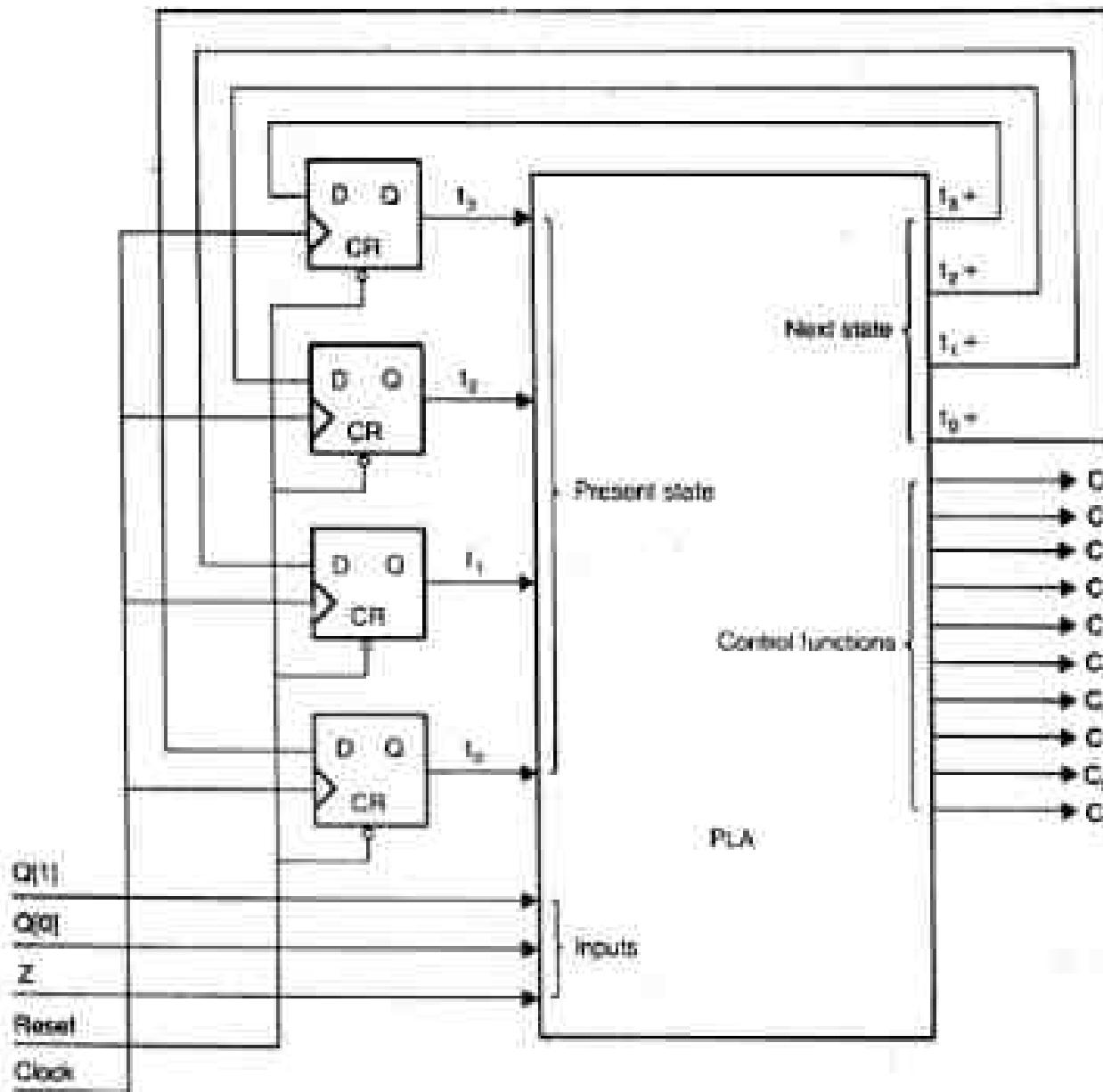


Figure 4.25 Organization of the PLA Control Unit for the Booth's Multiplier

Program clock #	PLA Input								PLA Output												
	Present state in binary				Inputs				Next state in binary				Control functions								
	R_1	R_2	R_3	R_4	Q_{1U}	Q_{1D}	Z	$I_1^{(1)}$	$I_2^{(1)}$	$I_3^{(1)}$	$I_4^{(1)}$	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	
T_0	0	0	0	0	X	X	X	0	0	0	0	1	1	1	1	0	0	0	0	0	0
T_1	0	0	0	1	X	X	X	0	0	0	0	0	0	0	0	1	0	0	0	0	0
T_2	0	0	1	0	0	1	X	0	0	1	1	1	0	0	0	0	0	0	0	0	0
T_3	0	0	1	0	0	0	X	0	1	0	1	0	0	0	0	0	0	0	0	0	0
T_4	0	0	1	0	1	1	X	0	1	0	1	0	0	0	0	0	0	0	0	0	0
T_5	0	0	1	0	1	0	X	0	1	0	0	0	0	0	0	0	0	0	0	0	0
T_6	0	0	1	0	1	0	X	0	1	0	0	0	0	0	0	0	0	0	0	0	0
T_7	0	0	1	1	X	X	X	0	1	0	0	0	0	0	0	0	0	0	1	1	0
T_8	0	1	0	0	X	X	X	0	1	0	1	0	0	0	0	0	0	0	1	0	0
T_9	0	1	0	1	X	X	X	0	0	0	1	0	0	0	0	0	0	0	1	0	0
T_{10}	0	1	0	1	X	X	X	0	0	1	0	0	0	0	0	0	0	0	1	0	0
T_{11}	0	1	1	0	X	X	X	0	1	1	0	0	0	0	0	0	0	0	0	1	0
T_{12}	0	1	1	1	X	X	X	1	0	0	0	0	0	0	0	0	0	0	0	0	1
T_{13}	1	0	0	0	X	X	X	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Counter's state table

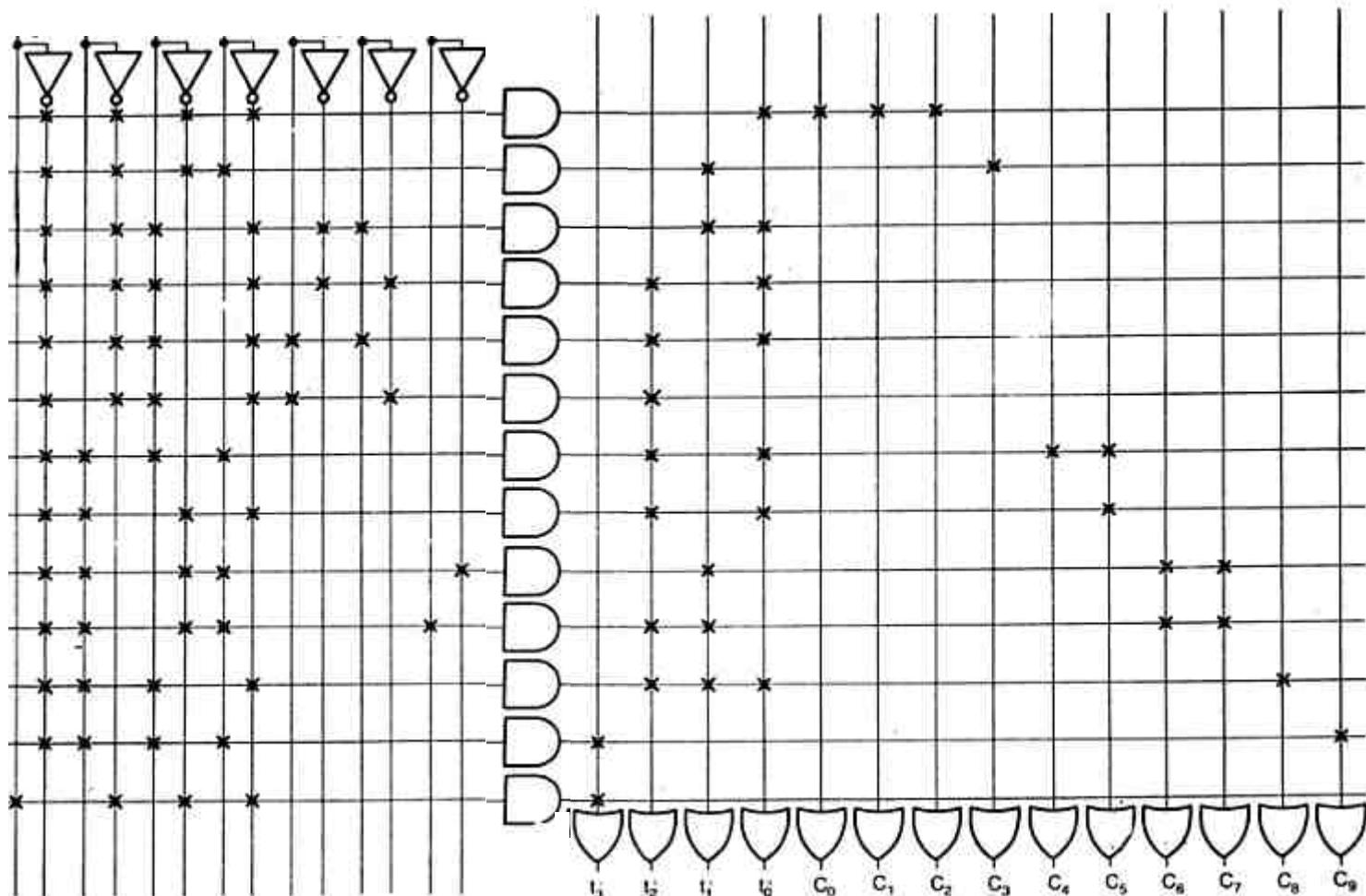


Figure 4.27 PLA Contents

Microprogrammed control unit:

Control word: all control signals that can be simultaneously activated are grouped to form the CW.

Micro operation: $A \leftarrow 0$, outbus=A, etc..

Each CW contains signals to activate one or more micro operations.

Control memory:

- Control words are held in separate memory called control memory (CM).
- Control words are fetched from CM and individual control fields are routed to various functional units to achieve desired task.

All microinstructions have 2 important fields:

- Control field

- Next address field

Purpose of control field is to indicate which control lines are to be activated.

Purpose of Next address field is to specify the address of the next microinstruction to be executed.

Wilke's design of microprogrammed control unit

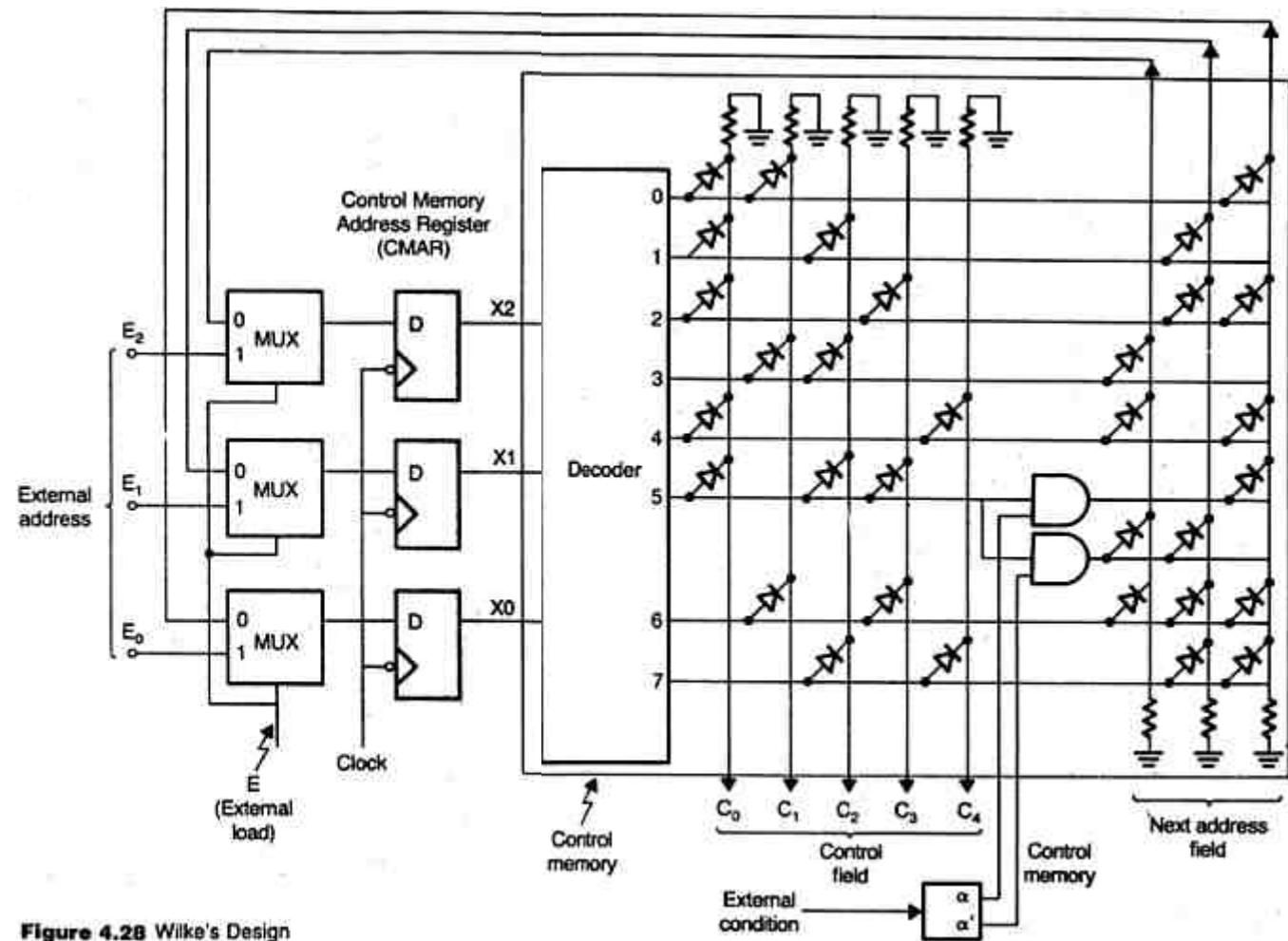


Figure 4.28 Wilke's Design

The length of the μ instruction is dependent on factors:

- How many μ operations will have to be activated simultaneously.
- The method by which the address of next μ instruction is specified.

All μ operations executed in parallel can be specified in a single μ instruction.

This allows short μ programs to be written.

If the degree of parallelism increases, then the length of μinstruction increases.

Very short μinstructions have limited capability in expressing parallelism. The overall length of μprogram will increase.

Various ways of organizing the control information:

Consider A, B, C, D each communicates with the outbus

C0: outbus=A;

C1: outbus=B;

C2: outbus=C;

C3: outbus=D;

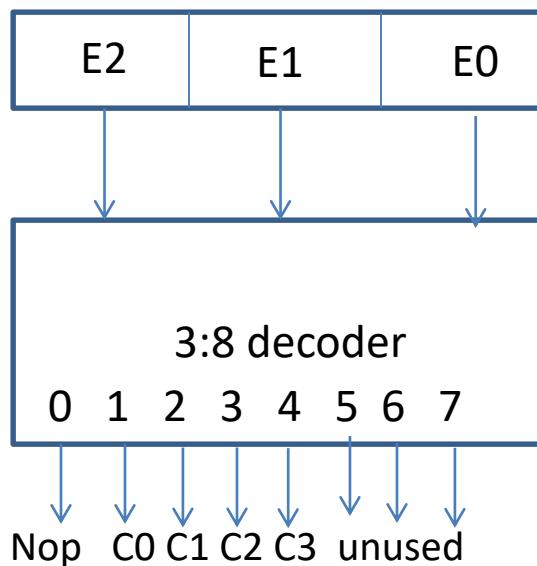
C0	C1	C2	C3
----	----	----	----

C0	C1	C2	C3	
1	0	0	0	Outbus=A
0	1	0	0	Outbus=B
0	0	1	0	Outbus=C
0	0	0	1	Outbus=D
0	0	0	0	NO operation

Here there is no need of decoding the control field.
 This method is known as unencoded format.

The above valid 5 binary patterns also can be represented as

E2	E1	E0	
0	0	0	No operation
0	0	1	Outbus=A
0	1	0	Outbus=B
0	1	1	Outbus=C
1	0	0	Outbus=D



HW: How a 15 control lines can be specified in unencoded and encoded format? What variations you can have?

What is Horizontal and vertical μ instr?
Features?

Hardwired approach v/s microprogrammed CU:

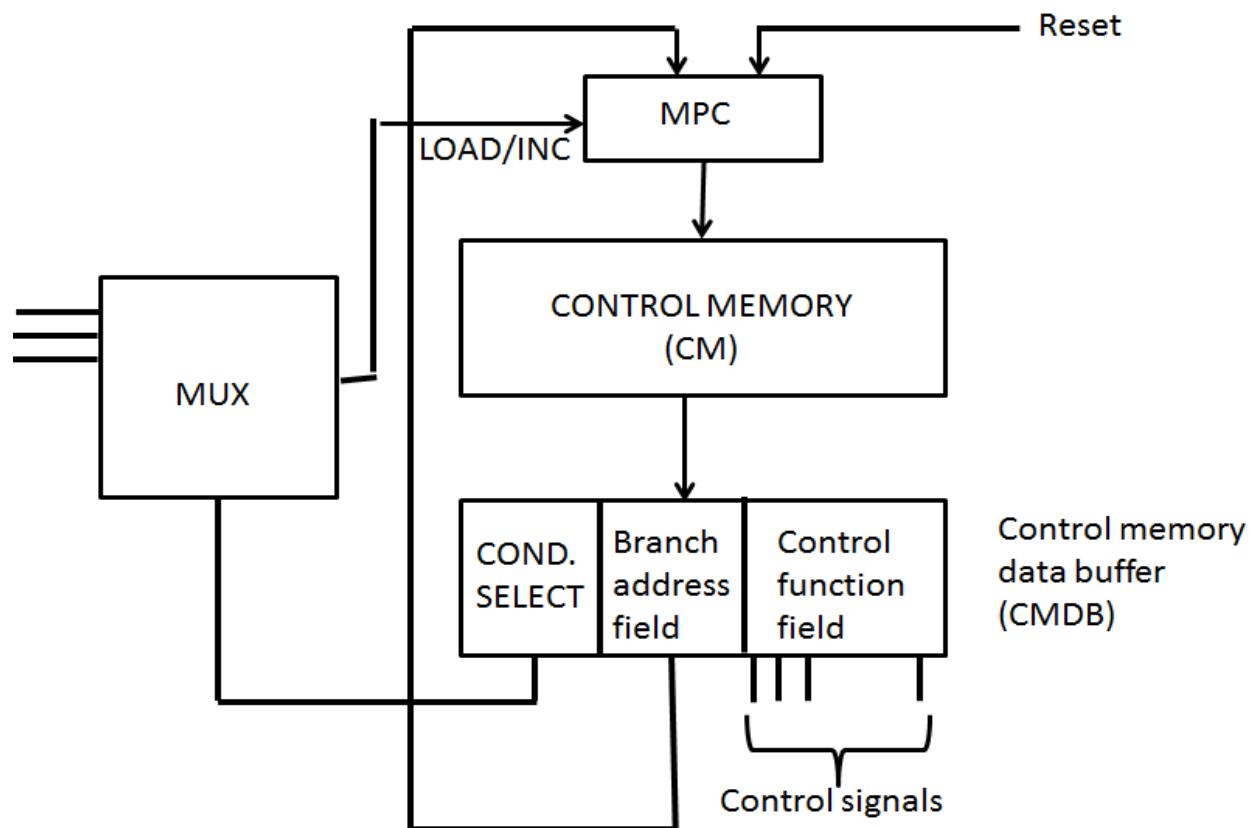
1. Microprogrammed approach is more expensive.
2. Control memory may reduce the overall speed of the machine, since microinstructions retrieval process takes significant amount of time.
3. Microprogramming provides a well structured control organization.
4. With Microprogramming, many additions and changes are made by simply changing the microprogram in Control memory, where as a small change in hardwired approach may lead to redesign the entire system.
5. Microprogrammed approach offers greater flexibility than hardwired since it provides an easy means for altering the contents of CM.

Architecture of modern microprogrammed control unit:

Next address field from the μ instruction can be eliminated. This pointer is referred as microprogram counter (MPC).

MPC is functionally identical to PC.

It points to the μ instruction to be executed next and incremented after each μ instruction fetch.



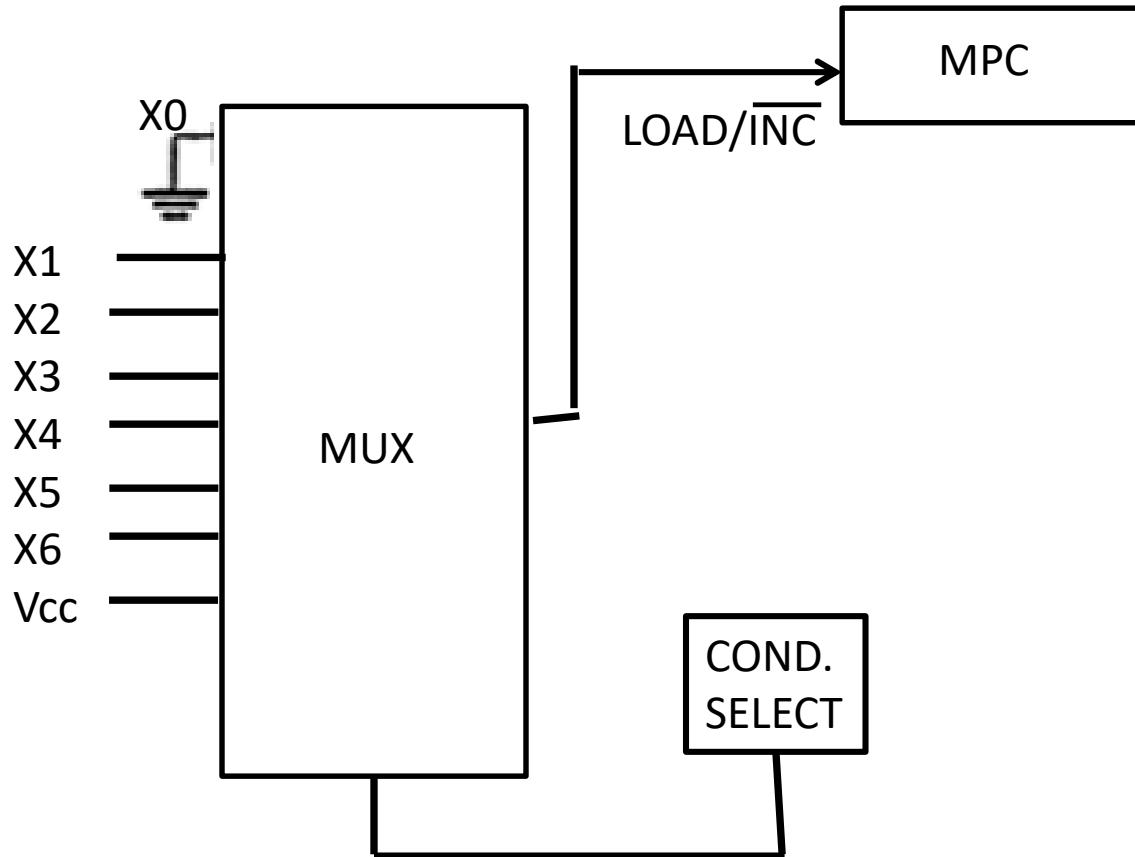
Suppose 6 external conditions X1, X2, X3, ..X6 are to be tested.

Cond select field and MUX can be organized as:

If cond sel 000 then MUX o/p 0. MPC incremented. No branch.

If cond sel 111 MUX o/p 1. Unconditional branching.

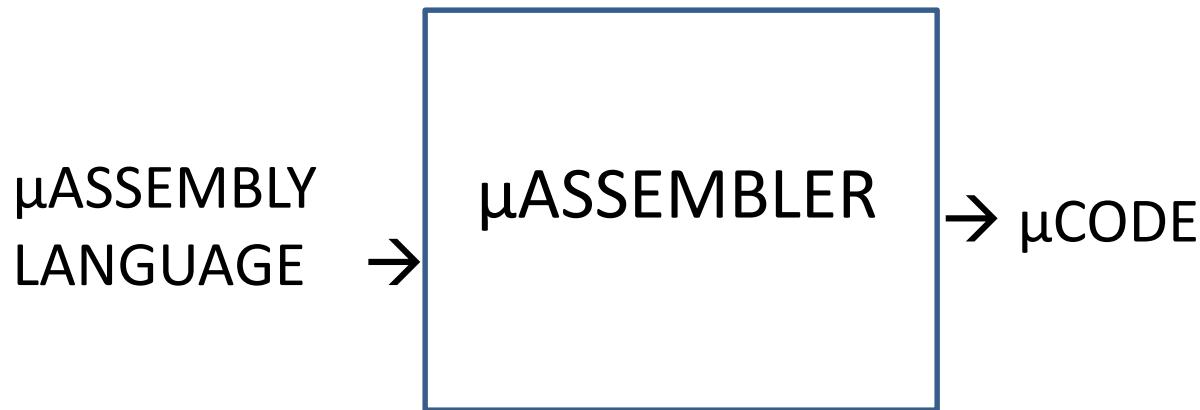
If cond sel 001 MUX o/p is same as value of X1. Now MPC will be loaded with branch addr when $X1=1$ else it is incremented.



Writing μ program is similar to writing ALP.

μ programmer must have more thorough knowledge about system archi

To speedup the development of μ code, \rightarrow μ assembly language.



These μ codes are held in CM.

Design of µprogrammed CU for 4 X 4 Booth's Multiplier:

STEP1: Write µprogram in a symbolic form.

Control Mem Addr

Control Word

0	START: A \leftarrow 0, M \leftarrow Inbus, L \leftarrow 4;
1	Q[4:1] \leftarrow Inbus, Q[0] \leftarrow 0;
2	LOOP: If Q[1:0]=01 then goto ADD;
3	If Q[1:0]=10 then goto SUB;
4	Goto RSHIFT;
5	ADD: A \leftarrow A+M;
6	Goto RSHIFT;
7	SUB: A \leftarrow A-M;
8	RSHIFT:ASR(A\$Q), L \leftarrow L-1;
9	If Z=0 then goto LOOP
10	outbus=A;
11	outbus=Q[4:1];
12	HALT: Goto HALT

CM holds 13 words, requiring a 4-bit branch address field.

STEP2: Q[1]Q[0]=01

Q[1]Q[0]=10 and Z=0 are checked. These cond are applied as i/ps to cond sel MUX.

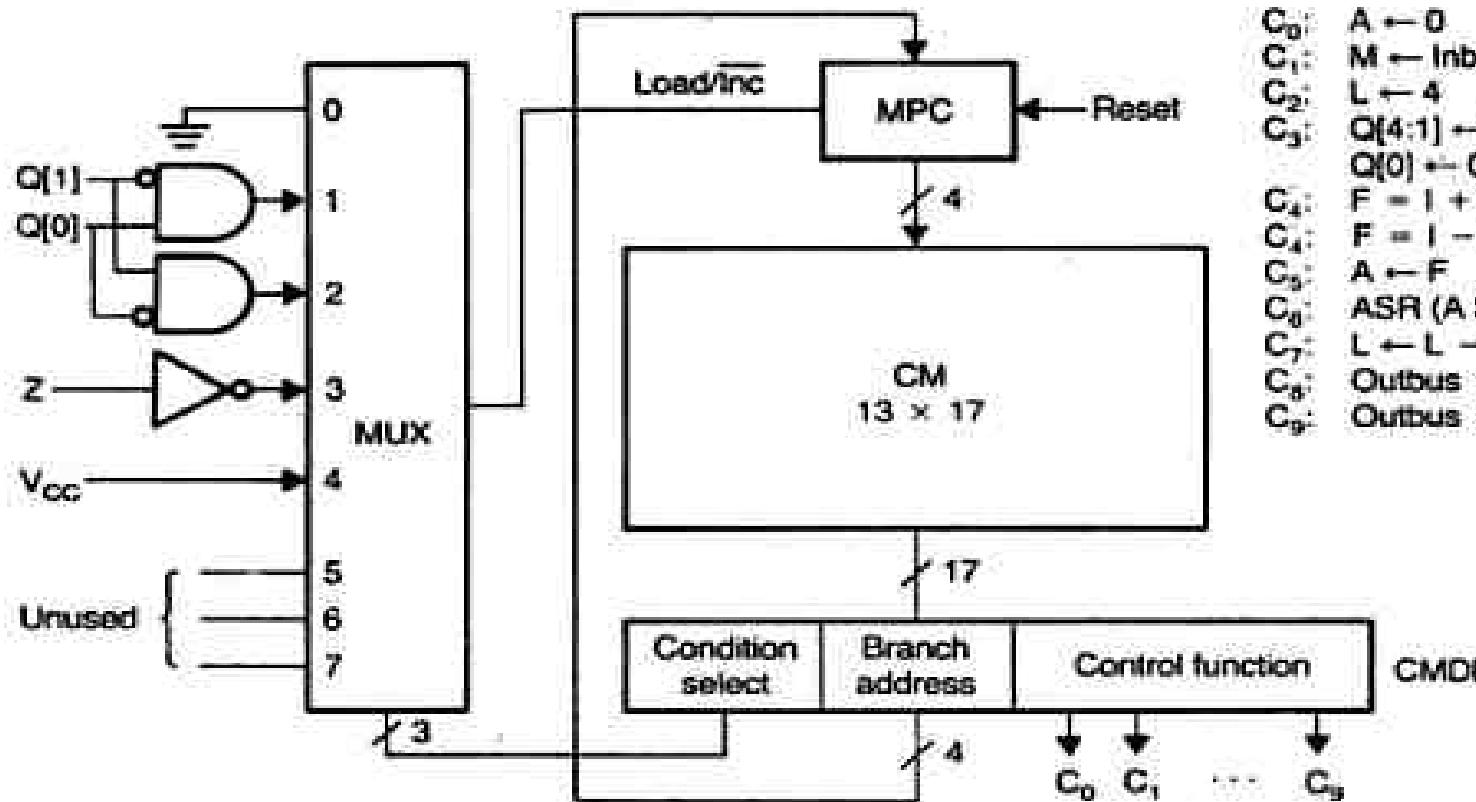
MUX must have atleast 5 data i/ps and 8:1 data selector.
3-bit cond sel field is used to encode 5 diff cond:

Cond Select Field	Action Taken
0 0 0	No branching
0 0 1	Branch if Q[1]Q[0]=01
0 1 0	Branch if Q[1]Q[0]=10
0 1 1	Branch if Z=0
1 0 0	Unconditional branch

Size of CW is

$$\begin{aligned} & \text{Size of} & \text{Size of} & \text{No of} \\ = & \text{Cond sel} & + \text{branch} & \text{functions} \\ & \text{field} & \text{field} & \\ = & 3 & + 4 & + 10 \\ = & 17 \text{ bits} & & \end{aligned}$$

Size of CMDB is 17 bits and CM is $13 \times 17 = 221$ bits

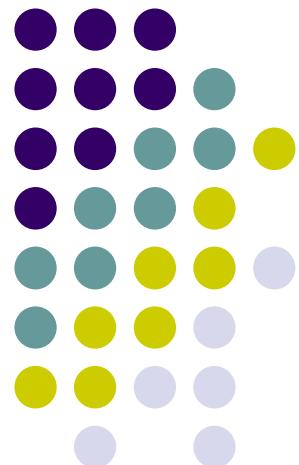


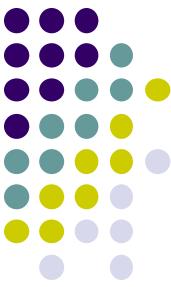
- | | |
|------------|----------------------------------|
| C_0 : | $A \leftarrow 0$ |
| C_1 : | $M \leftarrow \text{Inbus}$ |
| C_2 : | $L \leftarrow 4$ |
| C_3 : | $Q[4:1] \leftarrow \text{Inbus}$ |
| C_4 : | $Q[0] \leftarrow 0$ |
| C_5 : | $F = I + r$ |
| C_6 : | $F = I - r$ |
| C_7 : | $A \leftarrow F$ |
| C_8 : | $\text{ASR}(A, S, O)$ |
| C_9 : | $L \leftarrow L - 1$ |
| C_{10} : | $\text{Outbus} = A$ |
| C_{11} : | $\text{Outbus} = Q[4:1]$ |

ROM ADDRESS		CONTROL WORD											
In Dec	In Binary	COND SELECT	BRANCH ADDRESS	CONTROL FUNCTION									
				C0	C1	C2	C3	C4	C5	C6	C7	C8	C9
0	0 0 0 0	0 0 0	0 0 0 0	1	1	1	0	0	0	0	0	0	0
1	0 0 0 1	0 0 0	0 0 0 0	0	0	0	1	0	0	0	0	0	0
2	0 0 1 0	0 0 1	0 1 0 1	0	0	0	0	0	0	0	0	0	0
3	0 0 1 1	0 1 0	0 1 1 1	0	0	0	0	0	0	0	0	0	0
4	0 1 0 0	1 0 0	1 0 0 0	0	0	0	0	0	0	0	0	0	0
5	0 1 0 1	0 0 0	0 0 0 0	0	0	0	0	1	1	0	0	0	0
6	0 1 1 0	1 0 0	1 0 0 0	0	0	0	0	0	0	0	0	0	0
7	0 1 1 1	0 0 0	0 0 0 0	0	0	0	0	0	1	0	0	0	0
8	1 0 0 0	0 0 0	0 0 0 0	0	0	0	0	0	0	1	1	0	0
9	1 0 0 1	0 1 1	0 0 1 0	0	0	0	0	0	0	0	0	0	0
10	1 0 1 0	0 0 0	0 0 0 0	0	0	0	0	0	0	0	0	1	0
11	1 0 1 1	0 0 0	0 0 0 0	0	0	0	0	0	0	0	0	0	1
12	1 1 0 0	1 0 0	1 1 0 0	0	0	0	0	0	0	0	0	0	0

Binary listing of μprogram for 4 X 4 Booth's Multipier

The Memory System

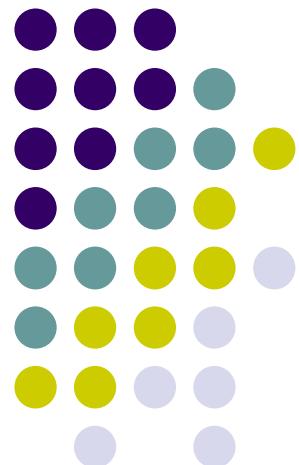




Overview

- Basic memory circuits
- Organization of the main memory
- Cache memory concept
- Virtual memory mechanism
- Secondary storage

Some Basic Concepts





Traditional Architecture

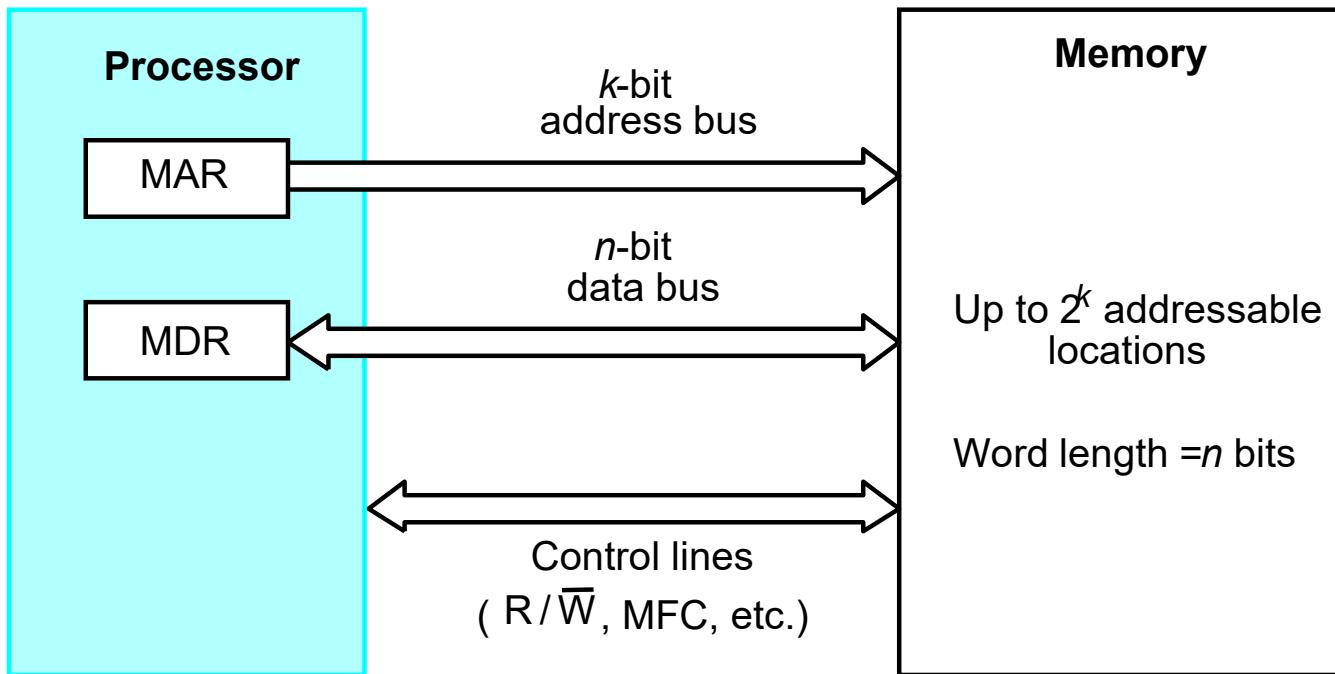
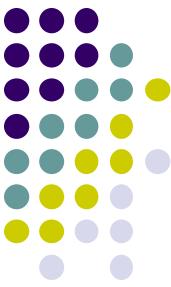


Figure 5.1. Connection of the memory to the processor.



Basic Concepts

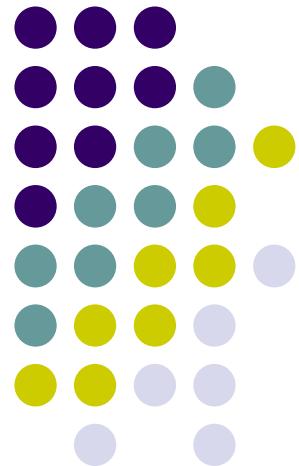
- “Block transfer” – bulk data transfer
- *Memory access time*-time that elapses between the initiation of an operation to transfer a word of data and the completion of that operation.
- *Memory cycle time*- the minimum time delay required between the initiation of two successive memory operations
- RAM – any location can be accessed for a Read or Write operation in some fixed amount of time that is independent of the location’s address.

Memory

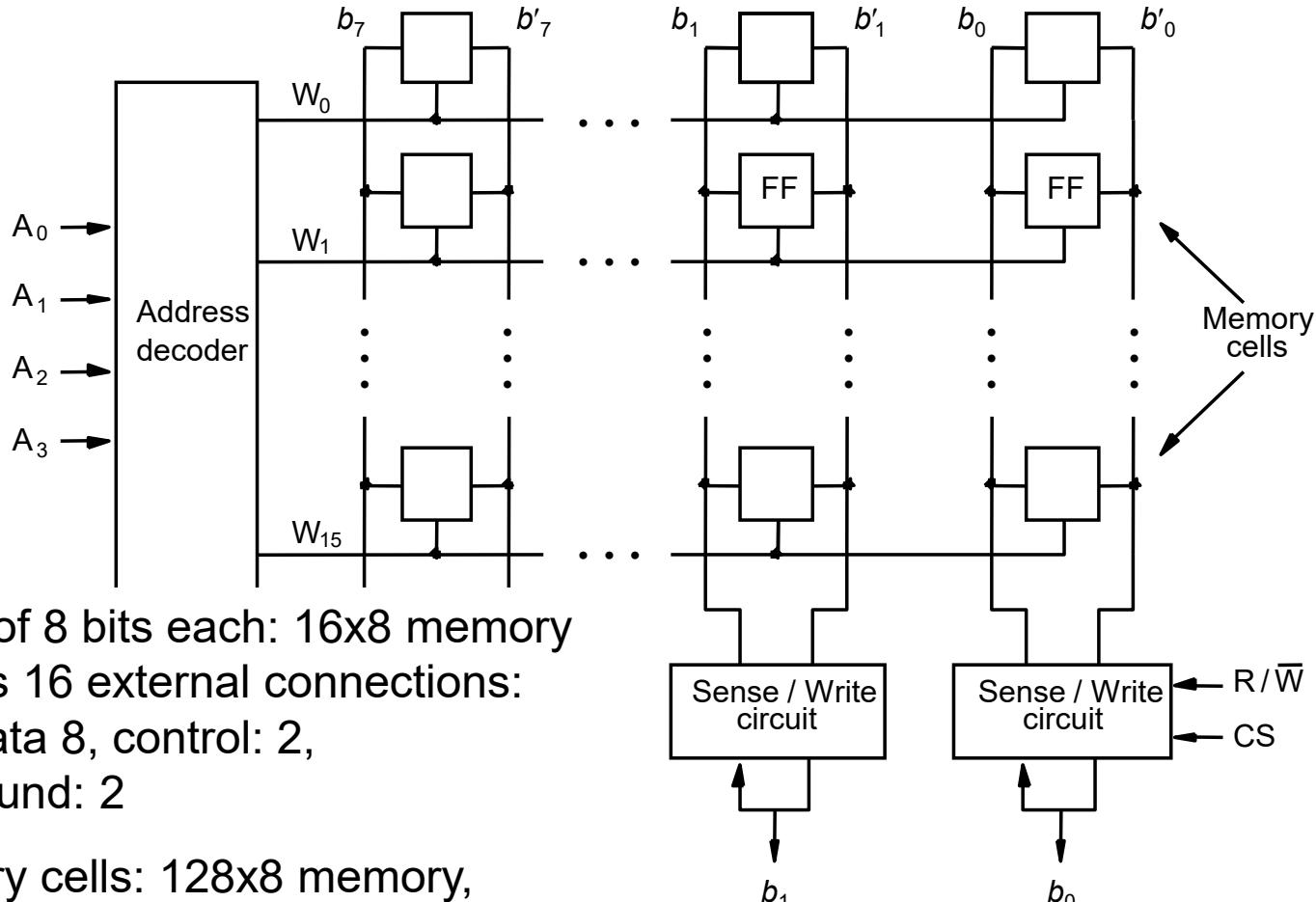
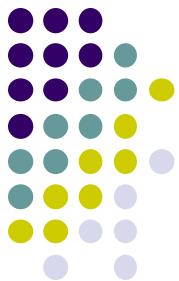


- **Cache memory**-a small, fast memory inserted between the larger, slower main memory and the processor.
 - It holds the currently active portions of a program and their data.
- **Virtual memory** -only the active portions of a program are stored in the main memory, and the remainder is stored on the much larger secondary storage device.
- Sections of the program are transferred back and forth between the main memory and the secondary storage device in a manner that is transparent to the application program.
- the application program sees a memory that is much larger than the computer's physical main memory

Semiconductor RAM Memories



Internal Organization of Memory Chips



Organization of bit cells in a memory chip.



A Memory Chip

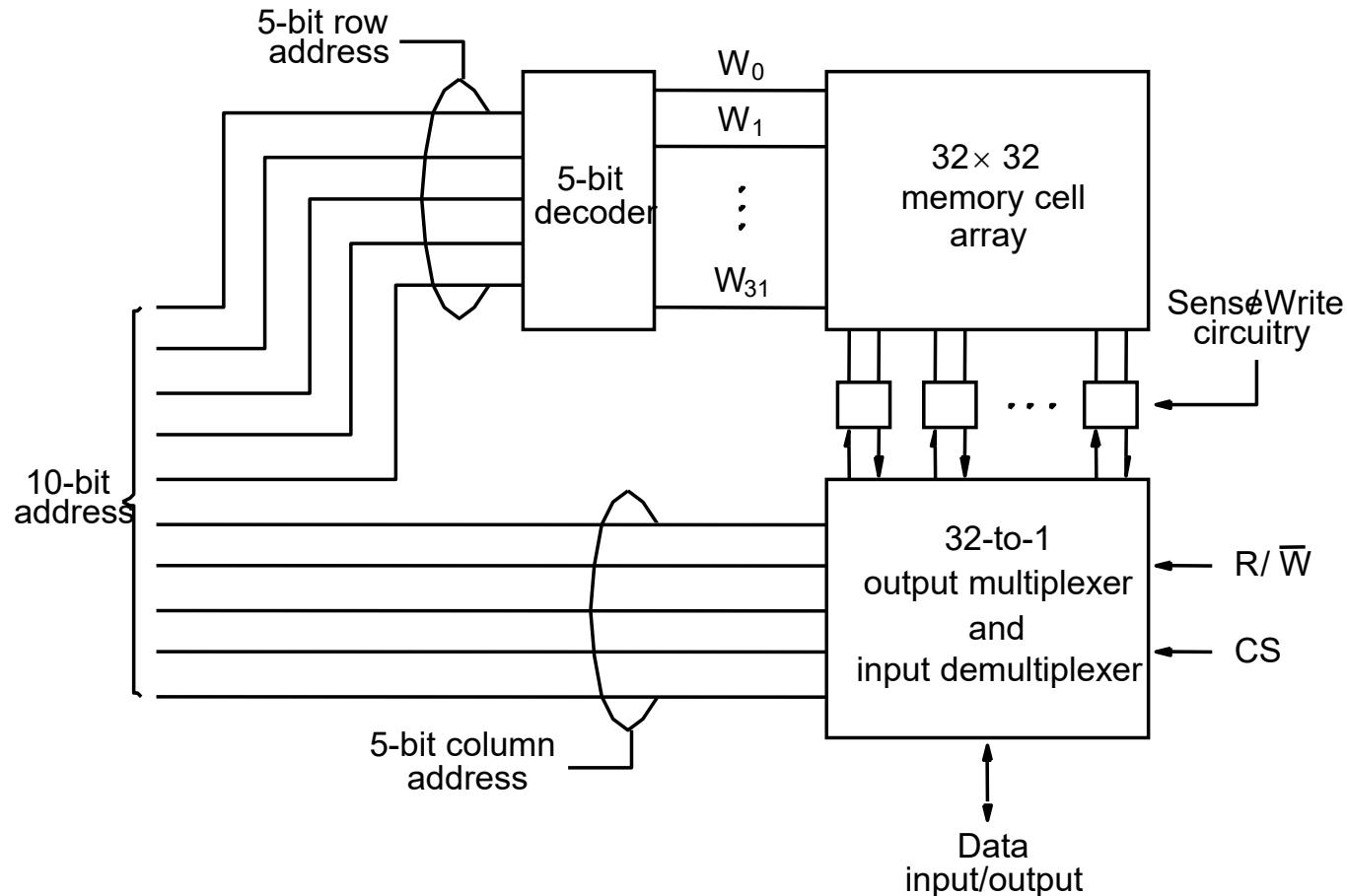


Figure 5.3. Organization of a $1K \times 1$ memory chip.

Structure of Larger Memories



21-bit address

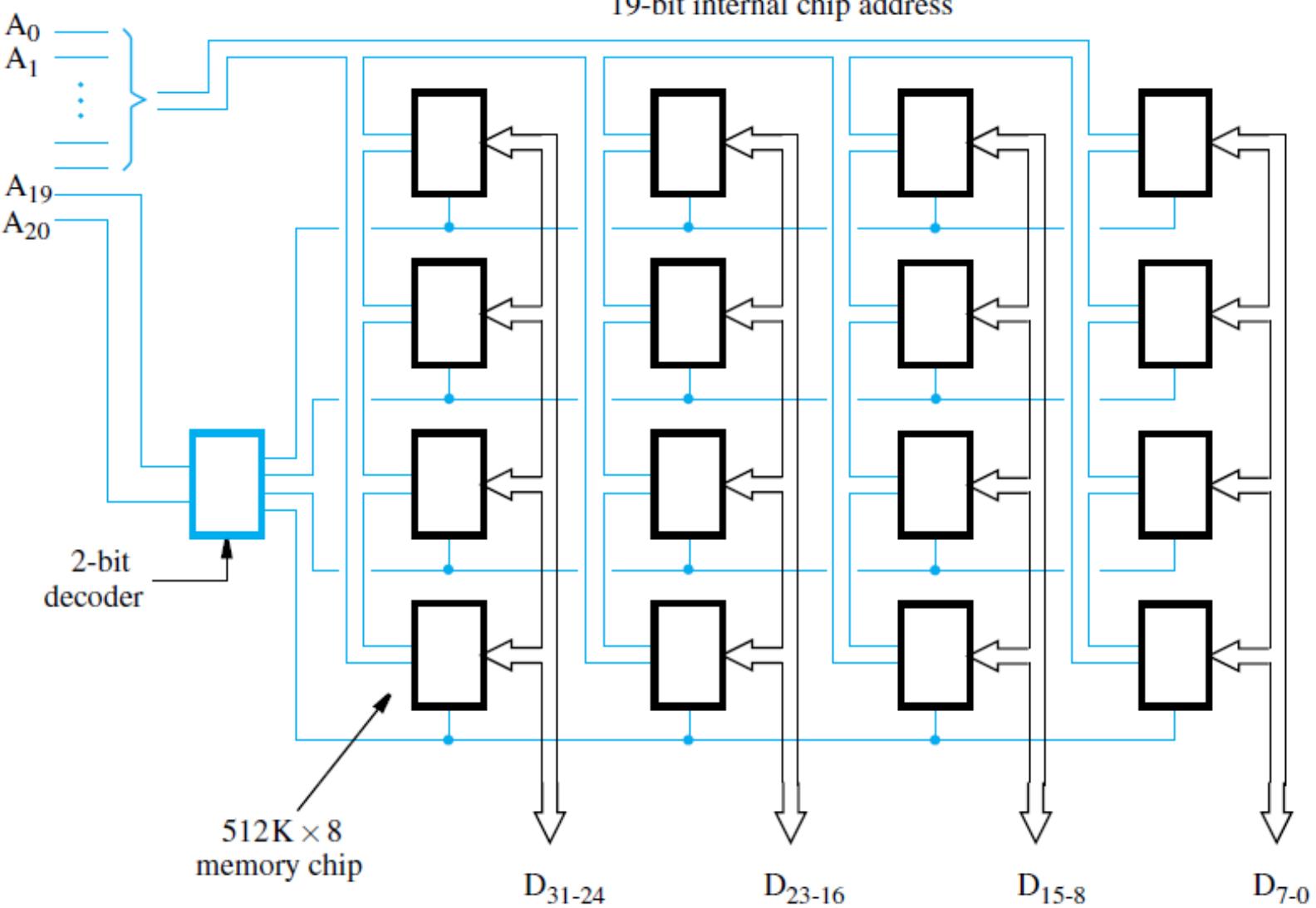
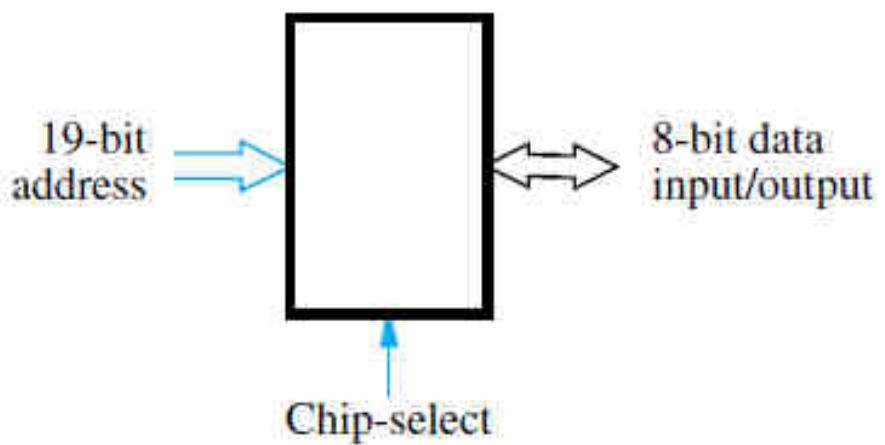


Figure 8.10 Organization of a 2M × 32 memory module using 512K × 8 static memory chips.



512K × 8 memory chip



Memory Hierarchy Speed, Size, and Cost

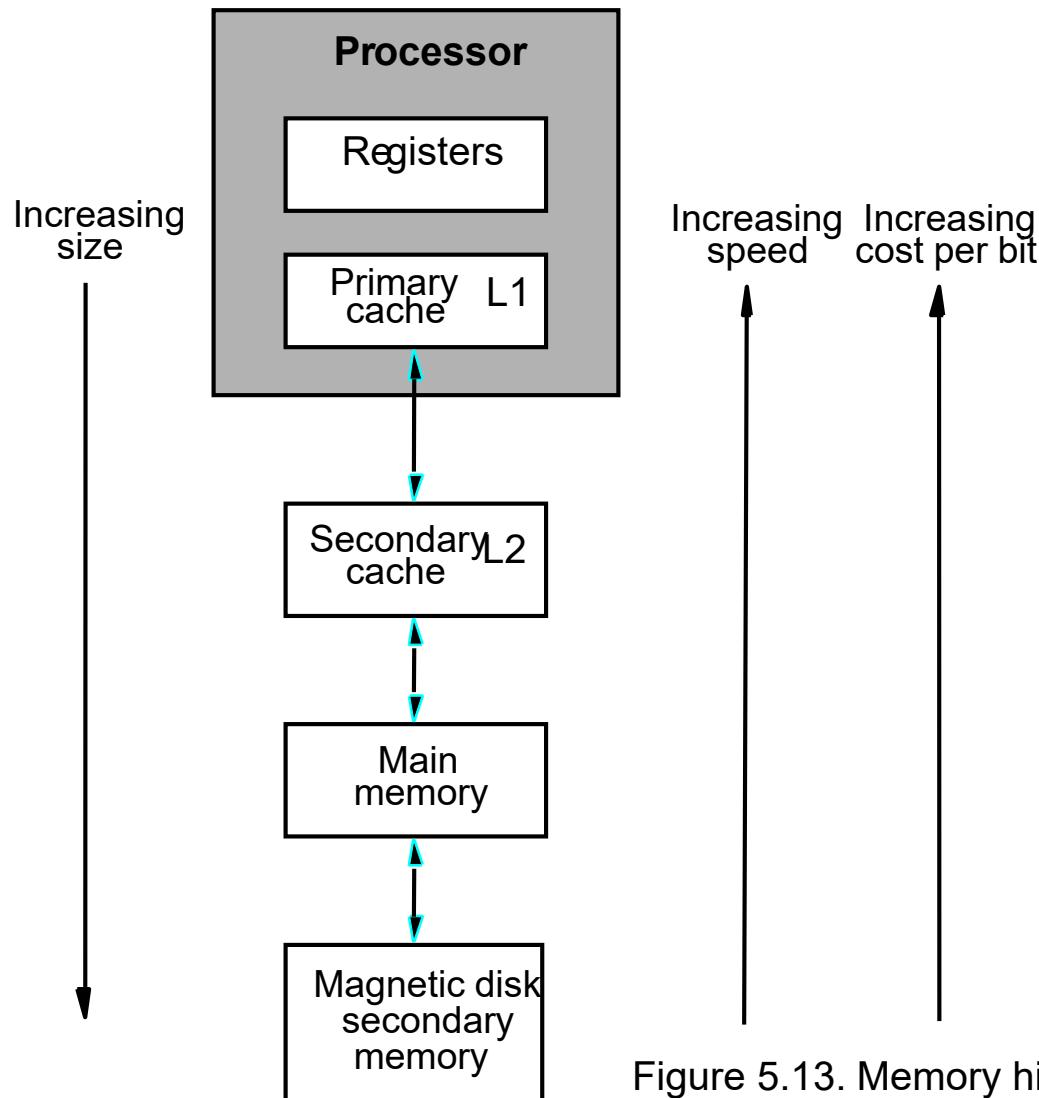
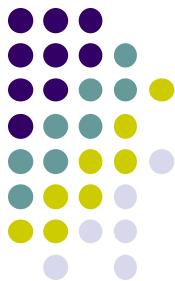
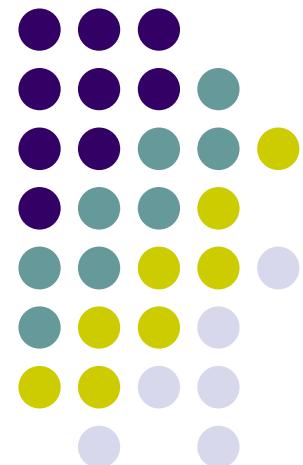


Figure 5.13. Memory hierarchy.

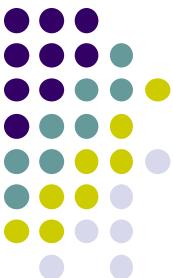
Cache Memories





Cache

- What is cache?
- Why we need it?
- Locality of reference (very important)
 - temporal- a recently executed instruction is likely to be executed again very soon. that.
 - spatial- instructions close to a recently executed instruction are also likely to be executed soon
- Cache block – *cache line*
 - A set of contiguous address locations of some size



Cache

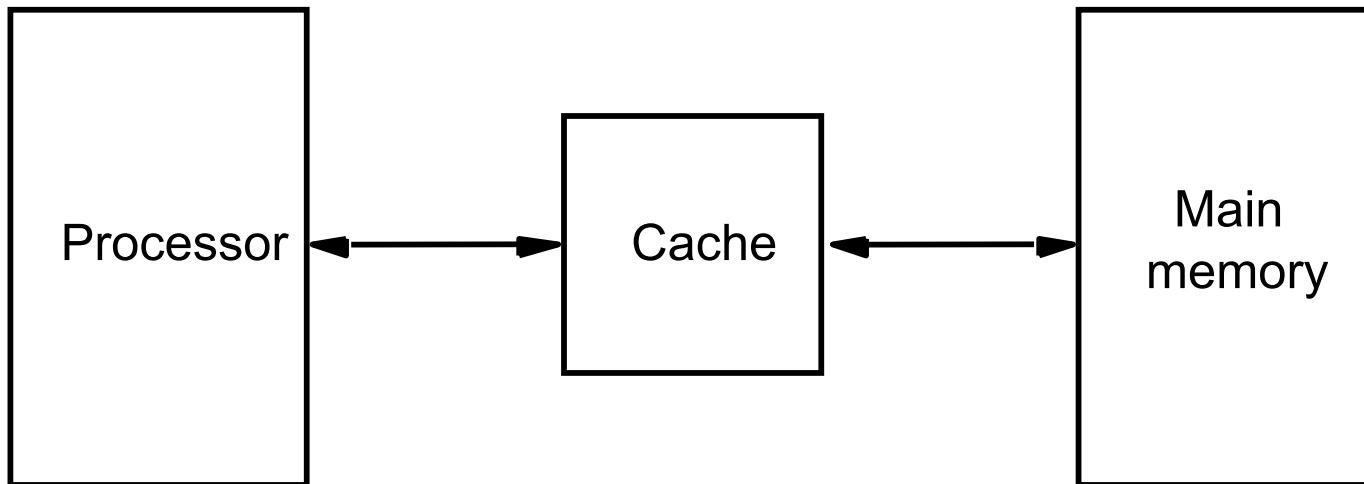
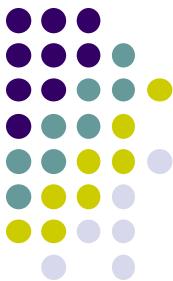


Figure 5.14. Use of a cache memory.

- Replacement algorithm
- Hit / miss
- Write-through / Write-back
- Load through



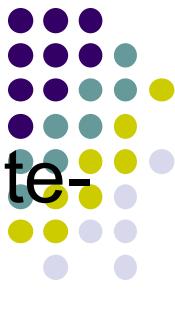
- The correspondence between the main memory blocks and those in the cache is specified by a *mapping function*.
- When the cache is full and a memory word (instruction or data) that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that contains the referenced word. The collection of rules for making this decision constitutes the cache's *replacement algorithm*.

Cache Hits



- The cache control circuitry determines whether the requested word currently exists in the cache.
- If it does, the Read or Write operation is performed on the appropriate cache location. -a ***read or write hit***.
 - main memory not involved when there is a cache hit in a Read operation
 - Write Operation –two ways
 - ***write-through protocol***, both the cache location and the main memory location are updated.
 - ***write-back, or copy-back***, protocol-update only the cache location and to mark the block containing it with an associated flag bit, often called the ***dirty or modified bit***. The main memory location of the word is updated later

Comparison

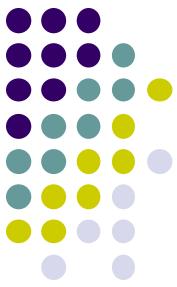


- The write-through protocol is simpler than the write-back protocol- but it results in unnecessary Write operations in the main memory when a given cache word is updated several times during its cache residency.
- The write-back protocol also involves unnecessary Write operations, because all words of the block are eventually written back, even if only a single word has been changed while the block was in the cache.
- The write-back protocol is used most often, to take advantage of the high speed with which data blocks can be transferred to memory chips.

Cache Misses



- Read operation for a word that is not in the cache constitutes a *Read miss*.
- causes the block of words containing the requested word to be copied from the main memory into the cache.
- After the entire block is loaded into the cache, the particular word requested is forwarded to the processor.
- Alternatively, this word may be sent to the processor as soon as it is read from the main memory. The latter approach, which is called ***load-through, or early restart***, reduces the processor's waiting time somewhat, at the expense of more complex circuitry.



- When a *Write miss* occurs in a computer that uses the write-through protocol, the information is written directly into the main memory.
- For the write-back protocol, the block containing the addressed word is first brought into the cache, and then the desired word in the cache is overwritten with the new information.



Direct Mapping

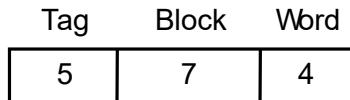
Block j of main memory maps onto block j modulo 128 of the cache

4: one of 16 words. (each block has $16=2^4$ words)

7: points to a particular block in the cache ($128=2^7$)

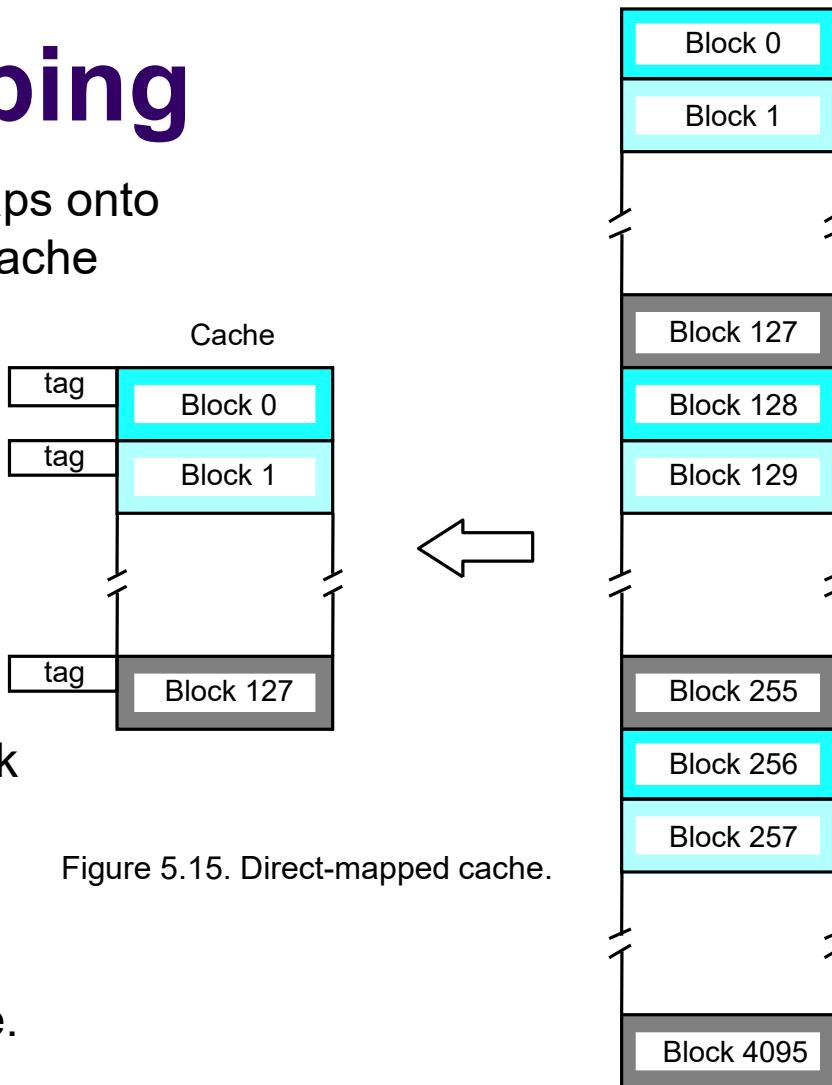
5: 5 tag bits are compared with the tag bits associated with its location in the cache.

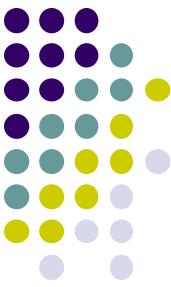
Identify which of the 32 blocks that are resident in the cache ($4096/128$).



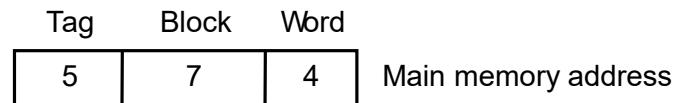
Main memory address

Figure 5.15. Direct-mapped cache.





Direct Mapping

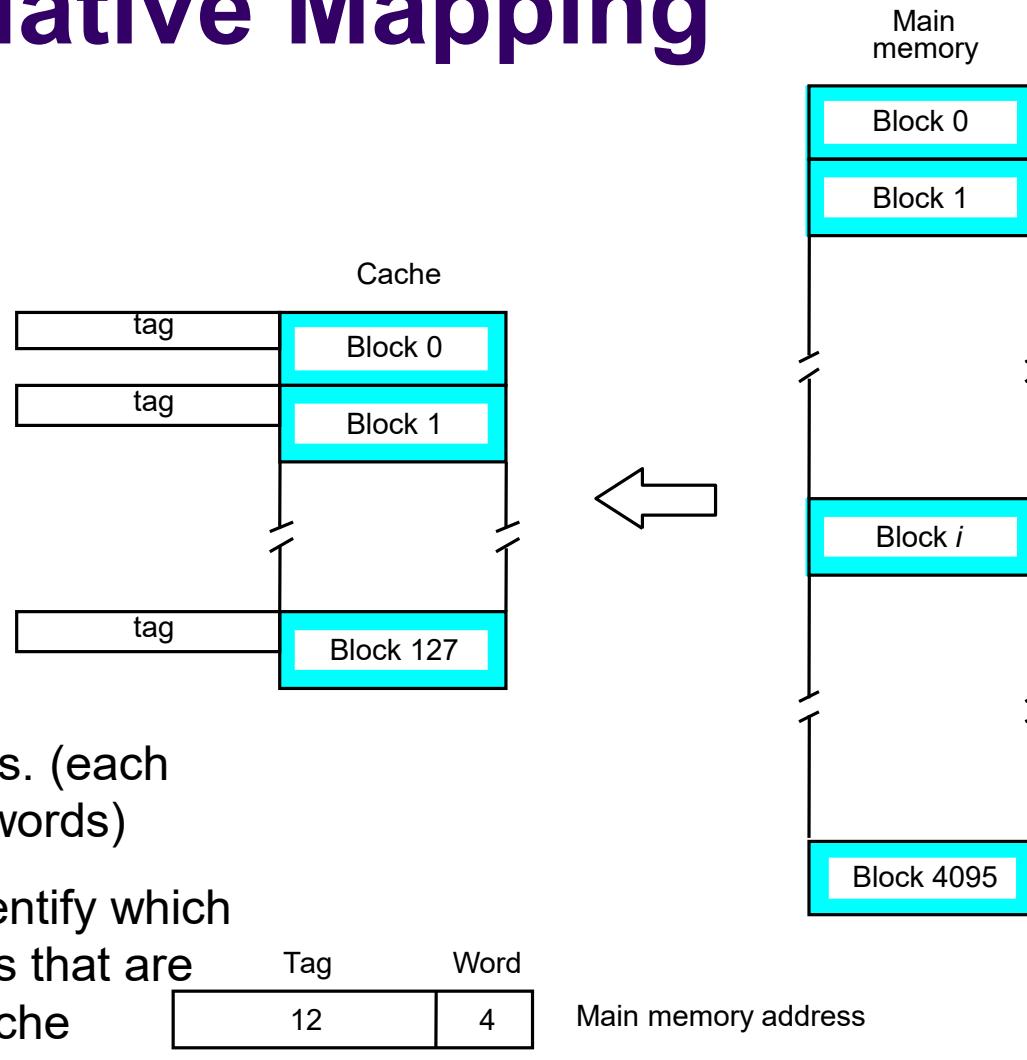


11101,1111111,1100

- Tag: 11101
- Block: $1111111=127$, in the 127th block of the cache
- Word: $1100=12$, the 12th word of the 127th block in the cache



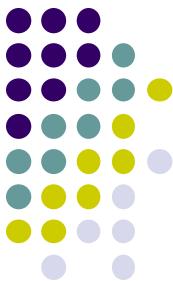
Associative Mapping



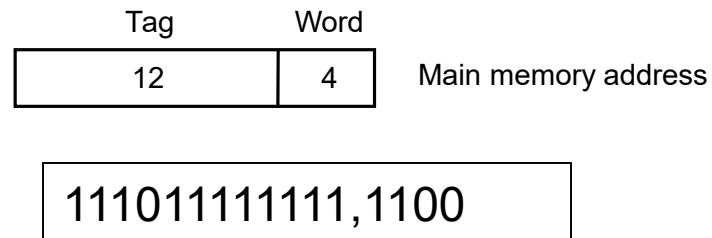
4: one of 16 words. (each block has $16=2^4$ words)

12: 12 tag bits Identify which of the 4096 blocks that are resident in the cache
 $4096=2^{12}$.

Figure 5.16. Associative-mapped cache.



Associative Mapping



- Tag: 111011111111
- Word: 1100=12, the 12th word of a block in the cache

Set-Associative Mapping

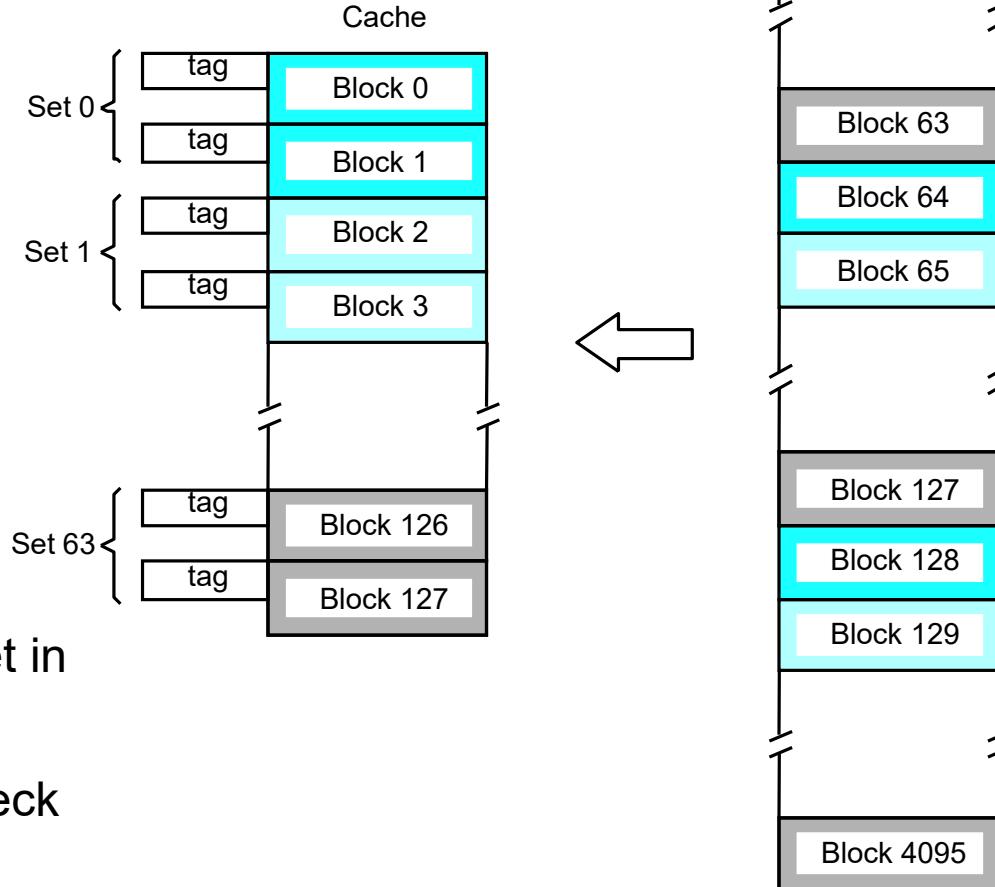


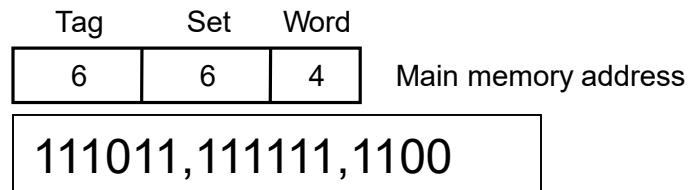
Figure 5.17. Set-associative-mapped cache with two blocks per set.

Tag	Set	Word
6	6	4

Main memory address



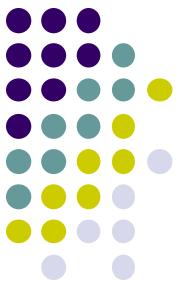
Set-Associative Mapping



- Tag: 111011
- Set: 111111=63, in the 63th set of the cache
- Word:1100=12, the 12th word of the 63th set in the cache



- A block-set-associative cache consists of a total of 64 blocks, divided into 4-block sets. The main memory contains 4096 blocks, each consisting of 32 words. Assuming a 32-bit byte-addressable address space, how many bits are there in each of the Tag, Set, and Word fields?



Solution:

Number of sets = $64/4 = 16$

Set bits = $4(2^4 = 16)$

Number of bytes = 128(Assuming 4 byte word)

Word bits = 7 bits ($2^7 = 128$)

- Tag=32-(4+7)=21



Problem:

A block-set-associative cache consists of a total of 64 blocks, divided into 4-block sets. The main memory contains 4096 blocks, each consisting of 128 words.

- a) How many bits are there in MM address?
- b) How many bits are there in each of the TAG, SET & word fields

Solution:



Number of sets = $64/4 = 16$

Set bits = 4 ($2^4 = 16$)

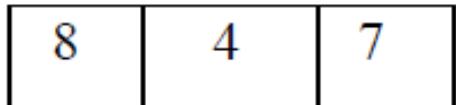
Number of words = 128

Word bits = 7 bits ($2^7 = 128$)

MM capacity : 4096×128 ($2^{12} \times 2^7 = 2^{19}$)

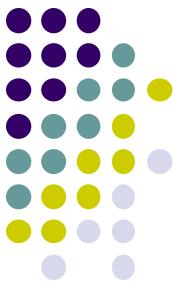
a) Number of bits in memory address = 19 bits

b)



TAG SET WORD

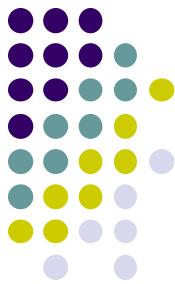
TAG bits = $19 - (7+4) = 8$ bits.



Problem:

A computer system has a MM capacity of a total of 1M 16 bits words. It also has a 4K words cache organized in the block set associative manner, with 4 blocks per set & 64 words per block. Calculate the number of bits in each of the TAG, SET & WORD fields of MM address format

Solution:



Capacity: 1M ($2^{20} = 1M$)

Number of words per block = 64

Number of blocks in cache = $4k/64 = 64$

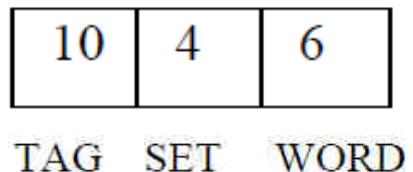
Number of sets = $64/4 = 16$

Set bits = 4 ($2^4 = 16$)

Word bits = 6 bits ($2^6 = 64$)

Tag bits = $20-(6+4) = 10$ bits

MM address format



- A 4×10 array of numbers, each occupying one word, is stored in main memory locations 7A00 through 7A27 (hex). The elements of this array, A, are stored in column order, as shown below. Assume the data cache has space for only eight blocks of data.

$$A(0, i) \leftarrow \frac{A(0, i)}{\left(\sum_{j=0}^9 A(0, j)\right) / 10} \quad \text{for } i = 0, 1, \dots, 9$$

```
SUM := 0
for j := 0 to 9 do
    SUM := SUM + A(0,j)
end
AVG := SUM/10
for i := 9 downto 0 do
    A(0,i) := A(0,i)/AVG
end
```

Figure 8.20 Task for example in Section 8.6.3.



	Memory address	Contents
(7A00)	0 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0	A(0,0)
(7A01)	0 1 1 1 1 0 1 0 0 0 0 0 0 0 0 1	A(1,0)
(7A02)	0 1 1 1 1 0 1 0 0 0 0 0 0 0 1 0	A(2,0)
(7A03)	0 1 1 1 1 0 1 0 0 0 0 0 0 0 1 1	A(3,0)
(7A04)	0 1 1 1 1 0 1 0 0 0 0 0 0 1 0 0	A(0,1)
	⋮	⋮
(7A24)	0 1 1 1 1 0 1 0 0 0 1 0 0 1 0 0	A(0,9)
(7A25)	0 1 1 1 1 0 1 0 0 0 1 0 0 1 0 1	A(1,9)
(7A26)	0 1 1 1 1 0 1 0 0 0 1 0 0 1 1 0	A(2,9)
(7A27)	0 1 1 1 1 0 1 0 0 0 1 0 0 1 1 1	A(3,9)

Diagram illustrating the memory organization:

- Tag for direct mapped: [Bit 11]
- Tag for set-associative: [Bit 10]
- Tag for associative: [Bit 9]

Figure 8.19 An array stored in the main memory.

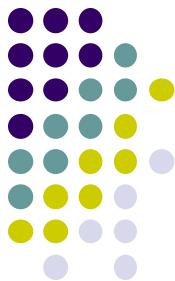
Direct Mapped Cache



Block position	Contents of data cache after pass:								
	$j = 1$	$j = 3$	$j = 5$	$j = 7$	$j = 9$	$i = 6$	$i = 4$	$i = 2$	$i = 0$
0	A(0,0)	A(0,2)	A(0,4)	A(0,6)	A(0,8)	A(0,6)	A(0,4)	A(0,2)	A(0,0)
1									
2									
3									
4	A(0,1)	A(0,3)	A(0,5)	A(0,7)	A(0,9)	A(0,7)	A(0,5)	A(0,3)	A(0,1)
5									
6									
7									

Figure 8.21 Contents of a direct-mapped data cache.

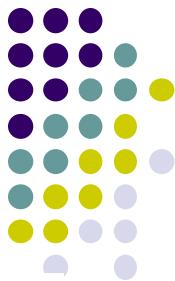
Associative Mapped Cache



		Contents of data cache after pass:				
Block position		$j = 7$	$j = 8$	$j = 9$	$i = 1$	$i = 0$
	0	A(0,0)	A(0,8)	A(0,8)	A(0,8)	A(0,0)
	1	A(0,1)	A(0,1)	A(0,9)	A(0,1)	A(0,1)
	2	A(0,2)	A(0,2)	A(0,2)	A(0,2)	A(0,2)
	3	A(0,3)	A(0,3)	A(0,3)	A(0,3)	A(0,3)
	4	A(0,4)	A(0,4)	A(0,4)	A(0,4)	A(0,4)
	5	A(0,5)	A(0,5)	A(0,5)	A(0,5)	A(0,5)
	6	A(0,6)	A(0,6)	A(0,6)	A(0,6)	A(0,6)
	7	A(0,7)	A(0,7)	A(0,7)	A(0,7)	A(0,7)

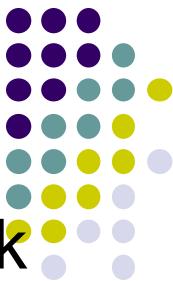
Figure 8.22 Contents of an associative-mapped data cache.

Set Associative Mapped Cache



Contents of data cache after pass:					
	$j = 3$	$j = 7$	$j = 9$	$i = 4$	$i = 2$
Set 0	A(0,0)	A(0,4)	A(0,8)	A(0,4)	A(0,4)
	A(0,1)	A(0,5)	A(0,9)	A(0,5)	A(0,5)
	A(0,2)	A(0,6)	A(0,6)	A(0,6)	A(0,2)
	A(0,3)	A(0,7)	A(0,7)	A(0,7)	A(0,3)

Figure 8.23 Contents of a set-associative-mapped data cache.



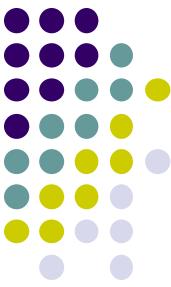
HomeWork

A byte-addressable computer has a small data cache capable of holding eight 32-bit words. Each cache block consists of one 32-bit word. When a given program is executed, the processor reads data sequentially from the following hex addresses:

200, 204, 208, 20C, 2F4, 2F0, 200, 204, 218, 21C, 24C, 2F4

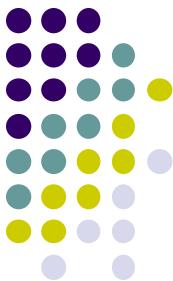
This pattern is repeated four times.

- (a) Assume that the cache is initially empty. Show the contents of the cache at the end of each pass through the loop if a direct-mapped cache is used, and compute the hit rate
- (b) Repeat part (a) for an associative-mapped cache that uses the LRU replacement algorithm.
- (c) Repeat part (a) for a four-way set-associative cache..



Replacement Algorithms

- Difficult to determine which blocks to be removed
- Least Recently Used (LRU) block
 - The cache controller tracks references to all blocks as computation proceeds.
 - Increase / clear track counters when a hit/miss occurs



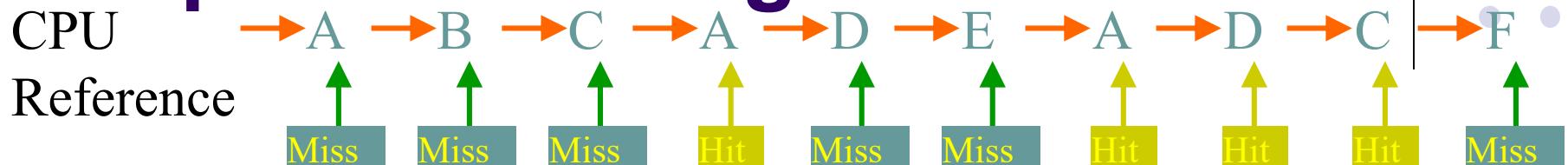
Replacement Algorithms

- For Associative & Set-Associative Cache

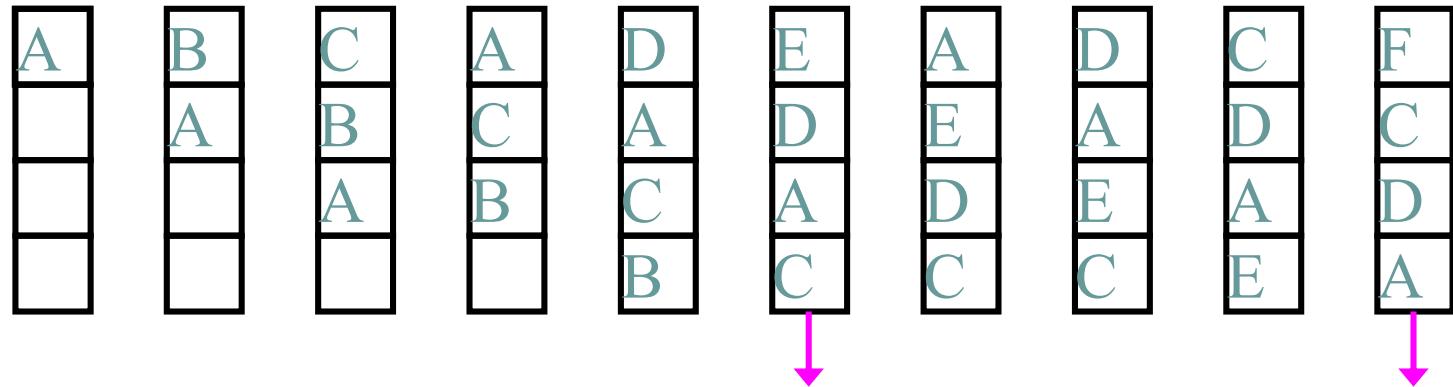
Which location should be emptied when the cache is full and a miss occurs?

- First In First Out (FIFO)
- Least Recently Used (LRU)
- Distinguish an *Empty* location from a *Full* one
 - Valid Bit

Replacement Algorithms



Cache
LRU ➔



$$\text{Hit Ratio} = 4 / 10 = 0.4$$

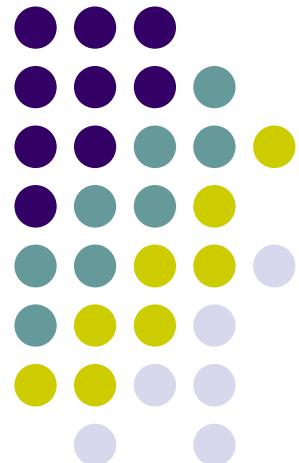


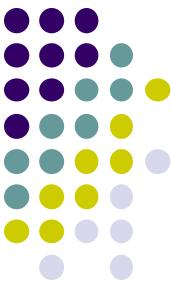
- The cache controller must track references to all blocks as computation proceeds.
- Suppose it is required to track the LRU block of a four-block set in a set-associative cache.
 - A 2-bit counter can be used for each block.
 - When a hit occurs, the counter of the block that is referenced is set to 0. Counters with values originally lower than the referenced one are incremented by one, and all others remain unchanged.
- When a *miss* occurs and the set is not full, the counter associated with the new block loaded from the main memory is set to 0, and the values of all other counters are increased by one.



- When a miss occurs and the set is full, the block with the counter value 3 is removed, the new block is put in its place, and its counter is set to 0.
- The other three block counters are incremented by one. It can be easily verified that the counter values of occupied blocks are always distinct

Virtual Memories

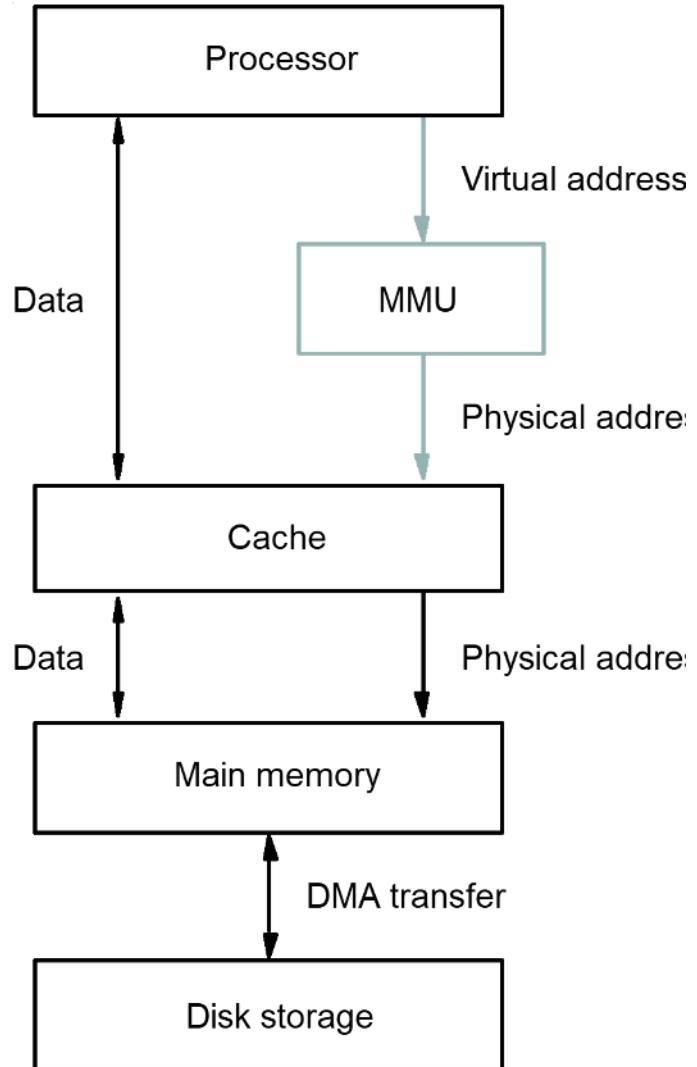




Overview

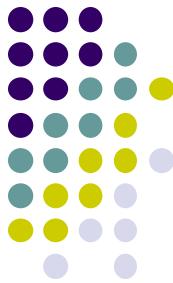
- Physical main memory is not as large as the address space spanned by an address issued by the processor.
 $2^{32} = 4 \text{ GB}$, $2^{64} = \dots$
- When a program does not completely fit into the main memory, the parts of it not currently being executed are stored on secondary storage devices.
- Techniques that automatically move program and data blocks into the physical main memory when they are required for execution are called virtual-memory techniques.
- Virtual addresses will be translated into physical addresses.

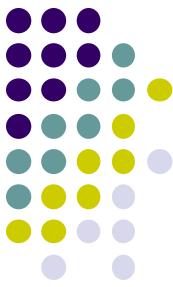
Overview



Memory
Management
Unit

Figure 5.26. Virtual memory organization





Address Translation

- All programs and data are composed of fixed-length units called pages, each of which consists of a block of words that occupy contiguous locations in the main memory.
- Page cannot be too small or too large.
- The virtual memory mechanism bridges the size and speed gaps between the main memory and secondary storage – similar to cache.



- Information about the main memory location of each page is kept in a ***page table***.
 - includes the main memory address where the page is stored and the current status of the page
 - Validity, modified
- An area in the main memory that can hold one page is called a ***page frame***.
- The starting address of the page table is kept in a ***page table base register***.
- By adding the virtual page number to the contents of this register, the address of the corresponding entry in the page table is obtained.
- The contents of this location give the starting address of the page if that page currently resides in the main memory.



Address Translation

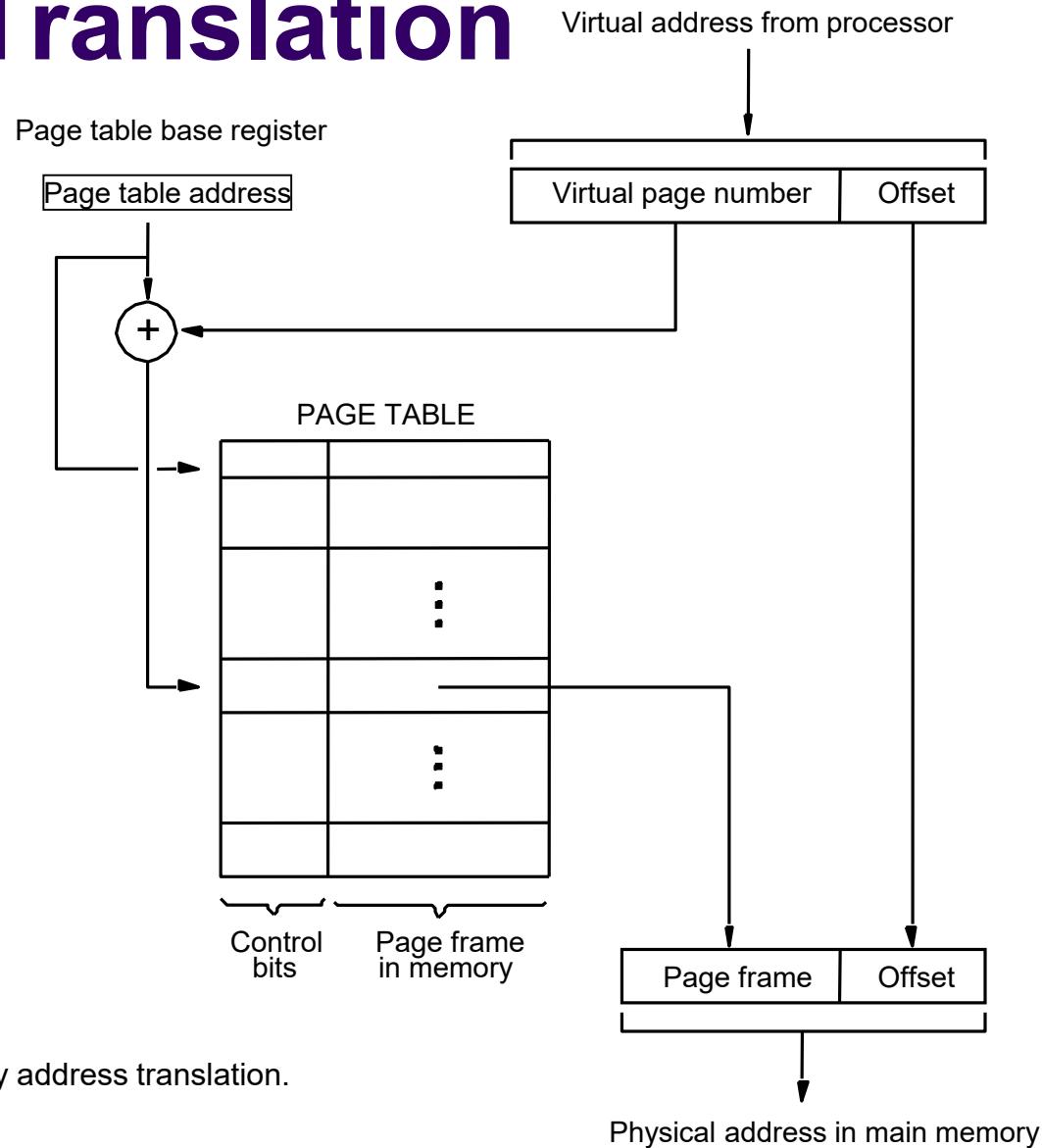
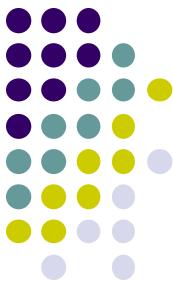


Figure 5.27. Virtual-memory address translation.

Physical address in main memory



Address Translation

- The page table information is used by the MMU for every access, so it is supposed to be with the MMU.
- Since MMU is on the processor chip and the page table is rather large, only small portion of it, which consists of the page table entries that correspond to the most recently accessed pages, can be accommodated within the MMU.
- Translation Lookaside Buffer (TLB)

TLB

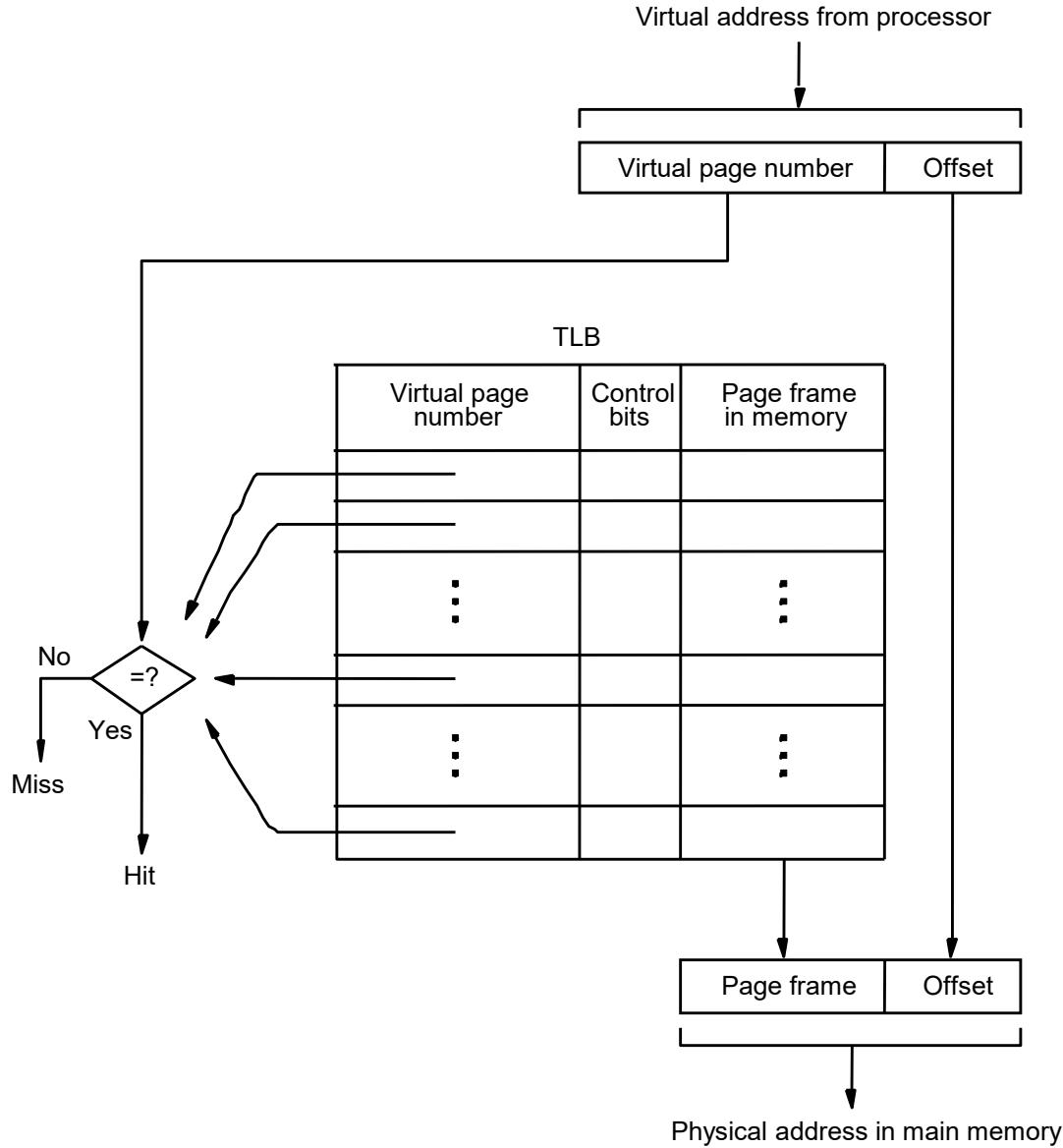
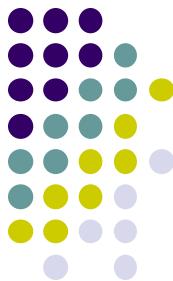
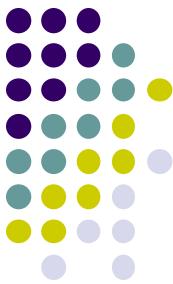


Figure 5.28. Use of an associative-mapped TLB.

TLB



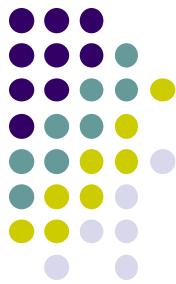
- The contents of TLB must be coherent with the contents of page tables in the memory
 - When the operating system changes the contents of a page table, it must simultaneously invalidate the corresponding entries in the TLB. One of the control bits in the TLB is provided for this purpose. When an entry is invalidated, the TLB acquires the new information from the page table in the memory as part of the MMU's normal response to access misses.
- Write-through is not suitable for virtual memory.
- Locality of reference in virtual memory



Translation procedure

- Given a virtual address, the MMU looks in the TLB for the referenced page. If the page table entry for this page is found in the TLB, the physical address is obtained immediately. If there is a miss in the TLB, then the required entry is obtained from the page table in the main memory and the TLB is updated.

Page fault

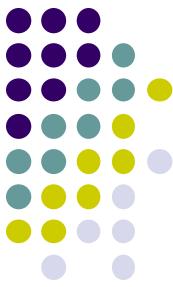


- When a program generates an access request to a page that is not in the main memory, a *page fault* is said to have occurred.
- The entire page must be brought from the disk into the memory before access can proceed. When it detects a page fault, the MMU asks the operating system to intervene by raising an exception (interrupt).

Page replacement



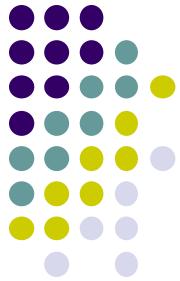
- If a new page is brought from the disk when the main memory is full, it must replace one of the resident pages.
- The problem of choosing which page to remove is just as critical here as it is in a cache, and the observation that programs spend most of their time in a few localized areas also applies.
- Concepts similar to the LRU replacement algorithm can be applied to page replacement, and the control bits in the page table entries can be used to record usage history



Page replacement – Contd..

- A modified page has to be written back to the disk before it is removed from the main memory. It is important to note that the write-through protocol, which is useful in the framework of cache memories, is not suitable for virtual memory. The access time of the disk is so long that it does not make sense to access it frequently to write small amounts of data

Performance Considerations



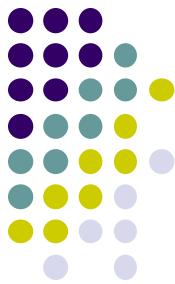
- Hit Rate and Miss Penalty
- Caches on the Processor Chip
- Other Enhancements

Hit Rate and Miss Penalty



- **hit rate** -number of hits stated as a fraction of all attempted accesses
- **miss rate**- number of misses stated as a fraction of attempted accesses
- the entire memory hierarchy would appear to the processor as a single memory unit that has **the access time of the cache on the processor chip** and the **size of the magnetic disk**
- the total access time seen by the processor when a miss occurs -***miss penalty***

Miss Penalty



- system with only one level of cache
 - the time to access a block of data in the main memory.
- Let h be the hit rate, M the miss penalty, and C the time to access information in the cache
- The average access time experienced by the processor is

$$t_{avg} = hC + (1 - h)M$$

Example



- Consider a computer that has the following parameters. Access times to the cache and the main memory are τ and 10τ , respectively. When a cache miss occurs, a block of 8 words is transferred from the main memory to the cache. It takes 10τ to transfer the first word of the block, and the remaining 7 words are transferred at the rate of one word every τ seconds. Assume that 30 percent of the instructions in a typical program perform a Read or a Write operation, which means that there are 130 memory accesses for every 100 instructions executed. Assume that the hit rates in the cache are 0.95 for instructions and 0.9 for data. Assume further that the miss penalty is the same for both read and write accesses
- miss penalty in this computer is given by:

$$M = \tau + 10\tau + 7\tau + \tau = 19\tau$$



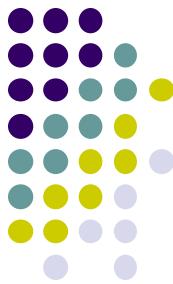
$$\frac{\text{Time without cache}}{\text{Time with cache}} = \frac{130 \times 10\tau}{100(0.95\tau + 0.05 \times 19\tau) + 30(0.9\tau + 0.1 \times 19\tau)} = 4.7$$

- cache makes the memory appear almost five times faster than it really is
- Let us consider how effective the cache of this example is compared to the ideal case in which the hit rate is 100 percent

$$\frac{\text{Time for real cache}}{\text{Time for ideal cache}} = \frac{100(0.95\tau + 0.05 \times 19\tau) + 30(0.9\tau + 0.1 \times 19\tau)}{130\tau} = 2.1$$

- 100% hit rate in the cache would make the memory appear twice as fast as when realistic hit rates are used.

How can the hit rate be improved?



- make the cache larger
 - increased cost
- increase the cache block size while keeping the total cache size constant, to take advantage of spatial locality
 - the performance of a computer is affected positively by increased hit rate and negatively by increased miss penalty
 - block size should be neither too small nor too large.
 - In practice, block sizes in the range of 16 to 128 bytes are the most popular choices
- miss penalty can be reduced if the load-through approach is used when loading new blocks into the cache.

Caches on the Processor Chip



- Most processor chips include at least one L1 cache. Often there are two separate L1 caches, one for instructions and another for data.
- In high-performance processors, two levels of caches are normally used, often implemented on the processor chip.
 - separate L1 caches for instructions and data –fast- 10's of KB
 - a larger L2 cache-slower but larger than L1- only affects the miss penalty of the L1 caches-100s of KB or MB



The average access time experienced by the processor in such a system is:

$$t_{avg} = h_1 C_1 + (1 - h_1)(h_2 C_2 + (1 - h_2)M)$$

where

h_1 is the hit rate in the L1 caches.

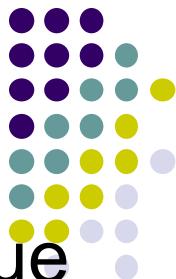
h_2 is the hit rate in the L2 cache.

C_1 is the time to access information in the L1 caches.

C_2 is the miss penalty to transfer information from the L2 cache to an L1 cache.

M is the miss penalty to transfer information from the main memory to the L2 cache.

Other Enhancements



- Each Write operation results in writing a new value into the main memory.
- If the processor must wait for the memory function to be completed, processor is slowed down by all Write requests.
- Processor does not need immediate access to the result of a Write operation; not necessary for it to wait for the Write request to be completed.
- *Write buffer* included for temporary storage of Write requests.

Write Buffer- write-through protocol (Contd..)



- Processor places each Write request into this buffer and continues execution of the next instruction.
- Sent to the main memory whenever the memory is not responding to Read requests.
- Read requests be serviced quickly, because the processor usually cannot proceed before receiving the data being read from the memory.
- These requests are given priority over Write requests



Write Buffer- write-through protocol (Contd..)

- The Write buffer may hold a number of Write requests.
- subsequent Read request may refer to data that are still in the Write buffer.
- To ensure correct operation, the addresses of data to be read from the memory are always compared with the addresses of the data in the Write buffer.
- In the case of a match, the data in the Write buffer are used.

Write Buffer- write-back protocol

- Write commands issued by the processor are performed on the word in the cache.
- When a new block of data is to be brought into the cache as a result of a Read miss, it may replace an existing block that has some dirty data.
- The dirty block has to be written into the main memory.
- If the required write-back is performed first, then the processor has to wait for this operation to be completed before the new block is read into the cache.



Write Buffer- write-back protocol(Contd..)

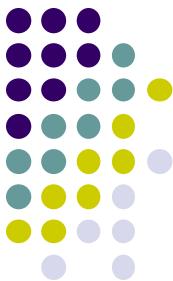
- It is more prudent to read the new block first.
- The dirty block being ejected from the cache is temporarily stored in the Write buffer and held there while the new block is being read.
- Afterwards, the contents of the buffer are written into the main memory.
- Write buffer also works well for the write-back protocol.

Prefetching



- new data are brought into the cache when they are first needed.
- Following a Read miss, the processor has to pause until the new data arrive, thus incurring a miss penalty.
- To avoid stalling the processor, it is possible to prefetch the data into the cache before they are needed
- Through
 - software -prefetch instruction
 - hardware- using circuitry that attempts to discover a pattern in memory references and prefetches data according to this pattern
- Prefetch-Executing this instruction causes the addressed data to be loaded into the cache, as in the case of a Read miss.

Prefetching-Contd..



- The hope is that prefetching will take place while the processor is busy executing instructions that do not result in a Read miss, thus allowing accesses to the main memory to be overlapped with computation in the processor
- Inserted either by programmer or compiler
- Overhead
 - Increases length of program
 - Data may not be used by instructions that follow-if the prefetched data are ejected from the cache by a Read miss involving other data



Lockup-Free Cache

- Software prefetching does not work well if the action of prefetching stops other accesses to the cache until the prefetch is completed
- While servicing a miss, the cache is said to be locked.
- modify the basic cache structure to allow the processor to access the cache while a miss is being serviced
- A cache that can support multiple outstanding misses is called ***lockup-free***.
- cache must include circuitry that keeps track of all outstanding misses

Problem



- Suppose that a computer has a processor with two L1 caches, one for instructions and one for data, and an L2 cache. Let τ be the access time for the two L1 caches. The miss penalties are approximately 15τ for transferring a block from L2 to L1, and 100τ for transferring a block from the main memory to L2. Assume that the hit rates are the same for instructions and data and that the hit rates in the L1 and L2 caches are 0.96 and 0.80, respectively.



- (a) What fraction of accesses miss in both the L1 and L2 caches, thus requiring access to the main memory?
- (b) What is the average access time as seen by the processor?
- (c) Suppose that the L2 cache has an ideal hit rate of 1. By what factor would this reduce the average memory access time as seen by the processor?
- (d) Consider the following change to the memory hierarchy. The L2 cache is removed and the size of the L1 caches is increased so that their miss rate is cut in half. What is the average memory access time as seen by the processor in this case?



(a) The fraction of memory accesses that miss in both the L1 and L2 caches is

$$(1 - h_1)(1 - h_2) = (1 - 0.96)(1 - 0.80) = 0.008$$

(b) The average memory access time using two cache levels is

$$\begin{aligned}t_{avg} &= 0.96\tau + 0.04(0.80 \times 15\tau + 0.20 \times 100\tau) \\&= 2.24\tau\end{aligned}$$

(c) With no misses in the L2 cache, we get:

$$t_{avg}(\text{ideal}) = 0.96\tau + 0.04 \times 15\tau = 1.56\tau$$

Therefore,

$$\frac{t_{avg}(\text{actual})}{t_{avg}(\text{ideal})} = \frac{2.24\tau}{1.56\tau} = 1.44$$

(d) With larger L1 caches and the L2 cache removed, the access time is

$$t_{avg} = 0.98\tau + 0.02 \times 100\tau = 2.98\tau$$



Reference

**Carl Hamacher, Zvonko Vranesic, Safwat Zaky and Naraig Manjikian,
“Computer Organization and Embedded Systems”, Sixth Edition, McGraw
Hill**

SECONDARY STORAGE:

The Semi-conductor memories do not provide all the storage capability.

The Secondary storage devices provide larger storage requirements.

Some of the Secondary Storage devices are,

- Magnetic Hard Disk
- Optical Disk
- Magnetic Tapes.

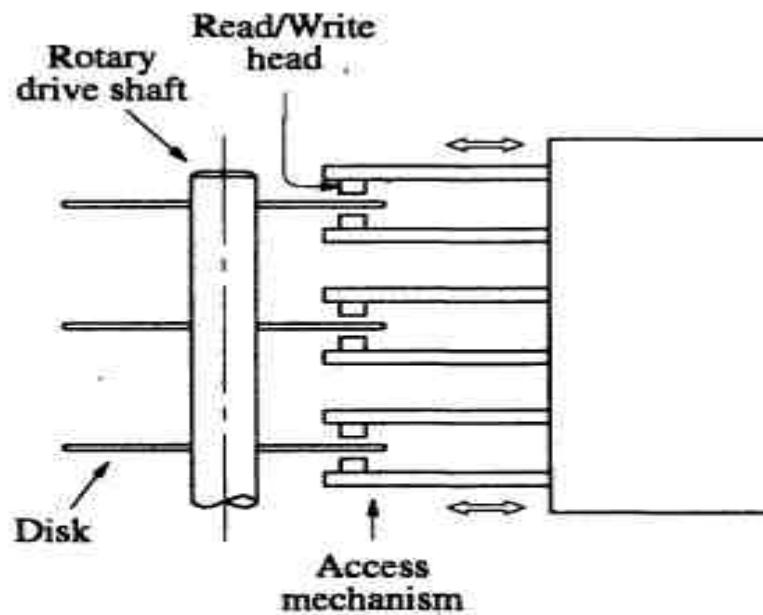
Magnetic Hard Disk:

One or more disk platters mounted on a common spindle.

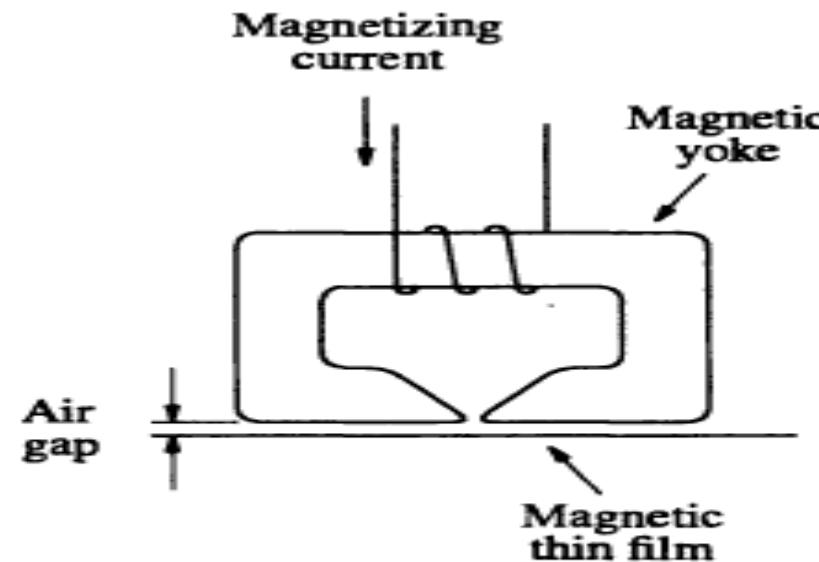
Magnetic film. Rotary drive.

Magnetized surfaces move in close proximity to read /write heads. Magnetic yoke and coil.

Info can be stored by applying the current pulse of suitable polarity to the magnetizing coil.



(a) Mechanical structure



(b) Read/Write head detail

Head consists of **magnetic yoke & magnetizing coil**.
Digital information can be stored by applying the current pulse.
Causes the magnetization of the film.
Same head can be used for reading.
Movement of the film relative to the yoke induce a voltage in the coil, a sense coil.
Very small distance to achieve high bit densities.

Winchester Technology:

In modern disk units, the disks and Read/Write heads are placed in a sealed, air –filtered enclosure called the Winchester Technology.

Read/write heads can operate closure to magnetic track surfaces and hence more densely the data can be packed.

Merits:

Winchester disks have a larger capacity.

Data intensity is high.

The read/write heads are movable.

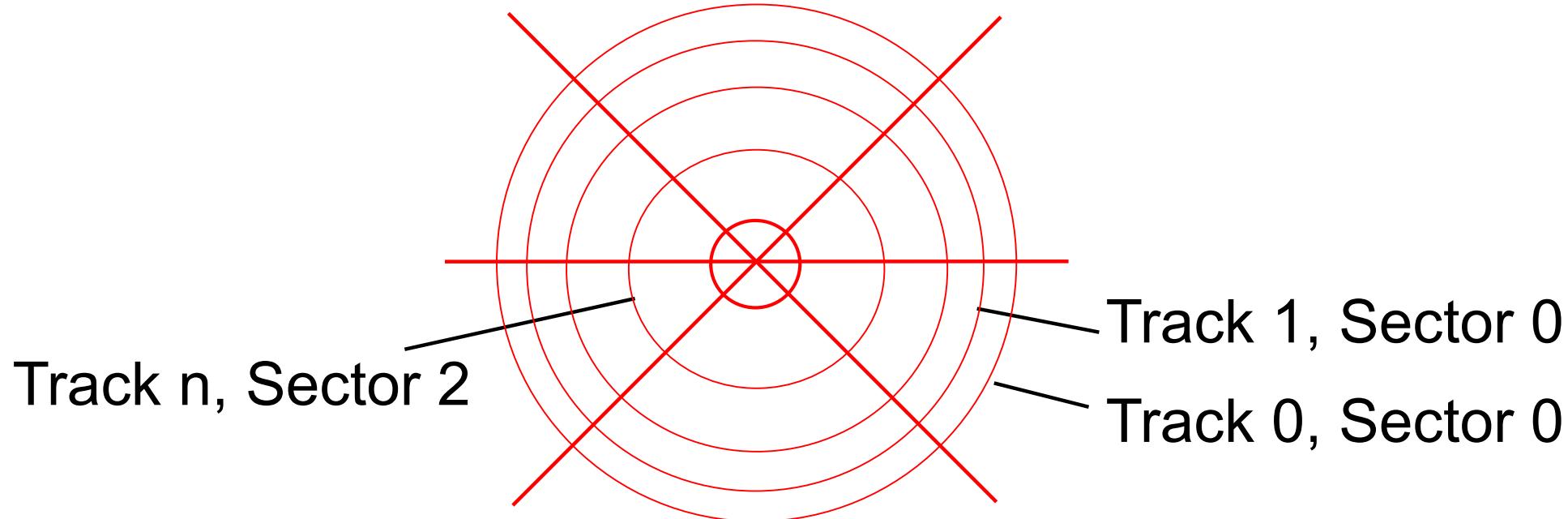
There is one head per surface.

All heads can move radially.

To read or write data on a given track, heads must first be positioned to that track.

The disk system consists of three key parts:

- Assembly of disk platters (disk)
- The electromechanical mechanism that spins the disk and moves the read/write heads (disk drive)
- The electronic circuitry that controls the operation of the system, (disk controller).



Organizing & Accessing the data on disk

Disk Platter

Disk Drive

Disk Controller

Tracks.

Sectors.

Logical cylinder.

The surface number, track number and the sector number.

Sector header , ECC (Error checking code)

Organization of data on a disk:

Each surface is concentric tracks, each track is divided into sectors.

The set of corresponding tracks on all surfaces of a stack of disks forms a logical cylinder.

The data on all tracks of a cylinder can be accessed without moving the read/write heads.

The data are accessed by specifying the surface number, the track number, and the sector number.

Data bits are stored serially on each track.

Each sector usually contains 512 bytes of data.

The data are preceded by a sector header.

Additional bits error-correcting code (ECC).

To distinguish sectors, there is a small intersector gap.
An unformatted disk has no information on its tracks.
The formatting information accounts for about 15%.
In a typical computer, the disk is subsequently divided into logical partitions.

Primary partition. number of additional partitions.
Each track has the same number of sectors.
So all tracks have the same storage capacity. Stored information is packed more densely on inner tracks than on outer tracks.
Possible to increase the storage density by placing more sectors on outer tracks, which have longer circumference.

Access time

There are 2 components involved:

They are,

Seek time – Time to move the RD/WR head to the proper track.

Latency – The amount of time that elapsed for the head to be positioned over the correct sector that passes under the read/write head.

Seek time + Latency = Disk access time

Typical disks

3.5 inch disk has the following parameter

Recording surface=20

Tracks=15000 tracks/surface

Sectors per track =400.

Each sector stores 512 bytes of data

$$\begin{aligned}\text{Capacity of formatted disk} &= 20 \times 15000 \times 400 \times 512 \\ &= 60 \times 10^9 = 60\text{GB}\end{aligned}$$

Seek time=6ms

Platter rotation=10000 rev/min

Latency=3ms and Internal transfer rate=34MB/s

Data Buffer / cache

That incorporates required SCSI circuit is SCSI drive.

The SCSI can transfer data at higher rate than the disk tracks.

Include a data buffer.

This buffer is a semiconductor memory.

The data buffer can provide cache mechanism for the disk
(i.e) when a read request arrives at the disk, then controller
first check if the data is available in the cache(buffer).

If the data is available in the cache, it can be accessed and
placed on SCSI bus . If it is not available then the data will be
retrieved from the disk.

Disk Controller

Operation of a disk drive is controlled by a disk controller. It provides interface between the disk drive and the bus that connects it to the rest of the computer system.

Figure shows a disk controller which controls two disk drives.

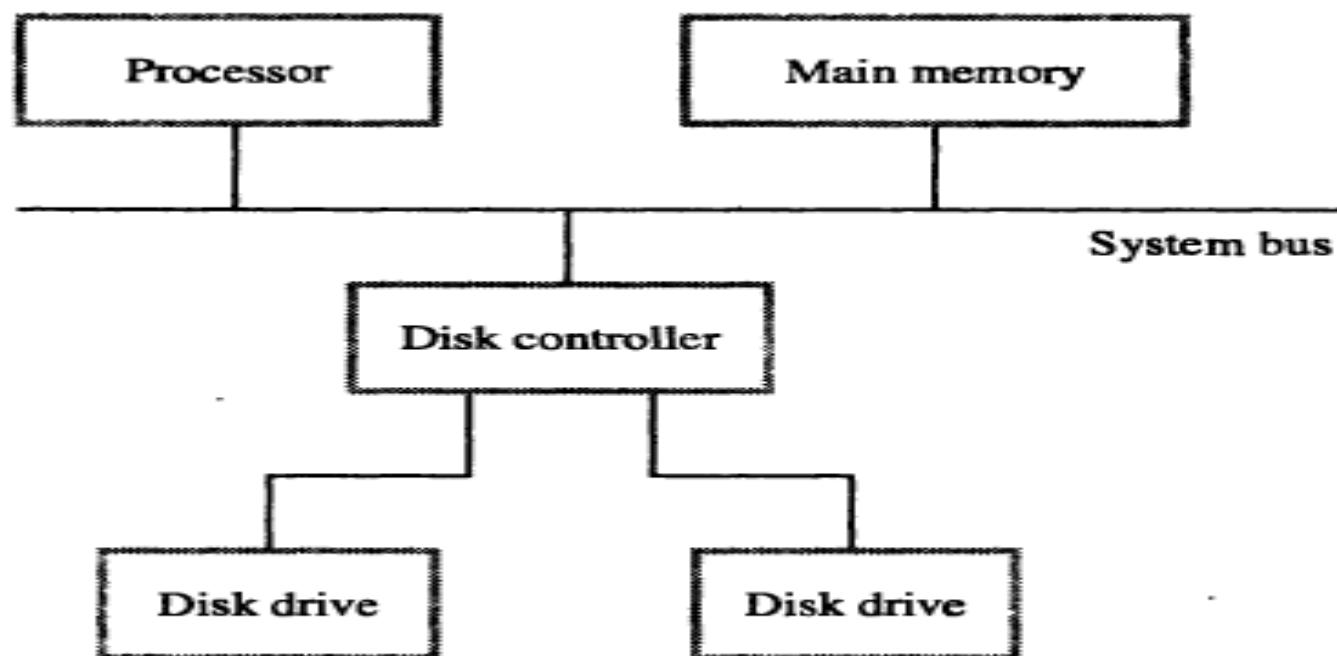


Figure 5.31 Disks connected to the system bus.

Disk Controller (Contd...)

The disk controller acts as interface between disk drive and system bus.

The disk controller uses DMA scheme to transfer data between disk and main memory.

When the OS initiates the transfer by issuing Read/Write request, the controller's register will load the following information:

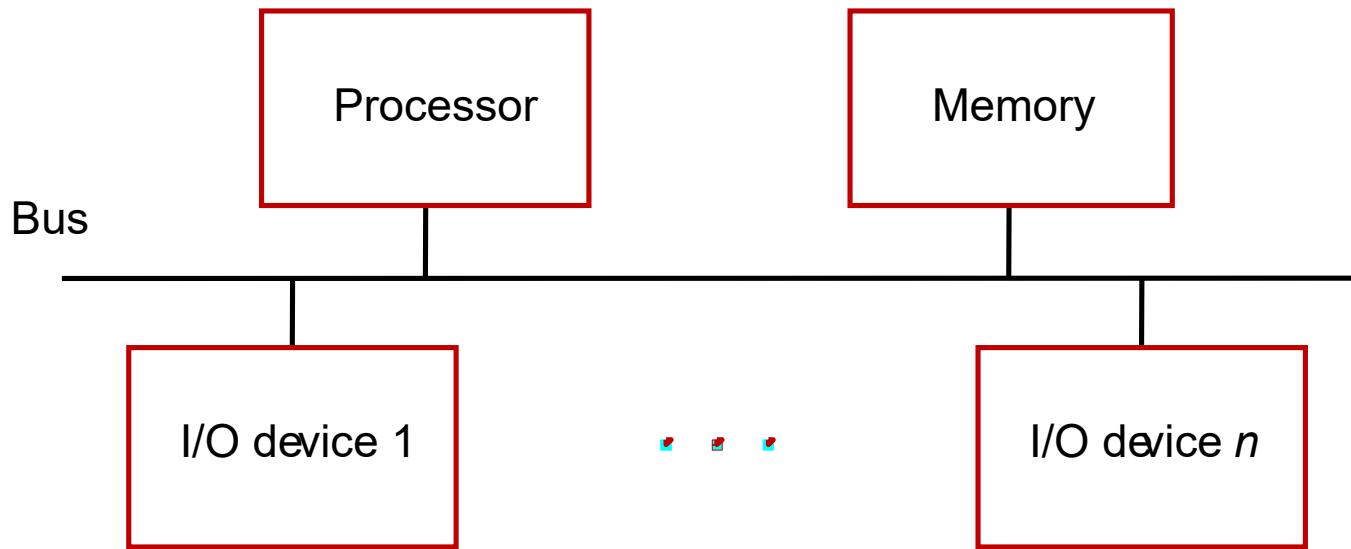
- Main memory address(address of first main memory location of the block of words involved in the transfer)
- Disk address(The location of the sector containing the beginning of the desired block of words)
- Word count (number of words in the block to be transferred).

The disk controller uses the DMA scheme to transfer data between the disk and the main memory. Actually, these transfers are from/to the data buffer.

INPUT/OUTPUT ORGANIZATION

Accessing I/O Devices

Accessing I/O devices



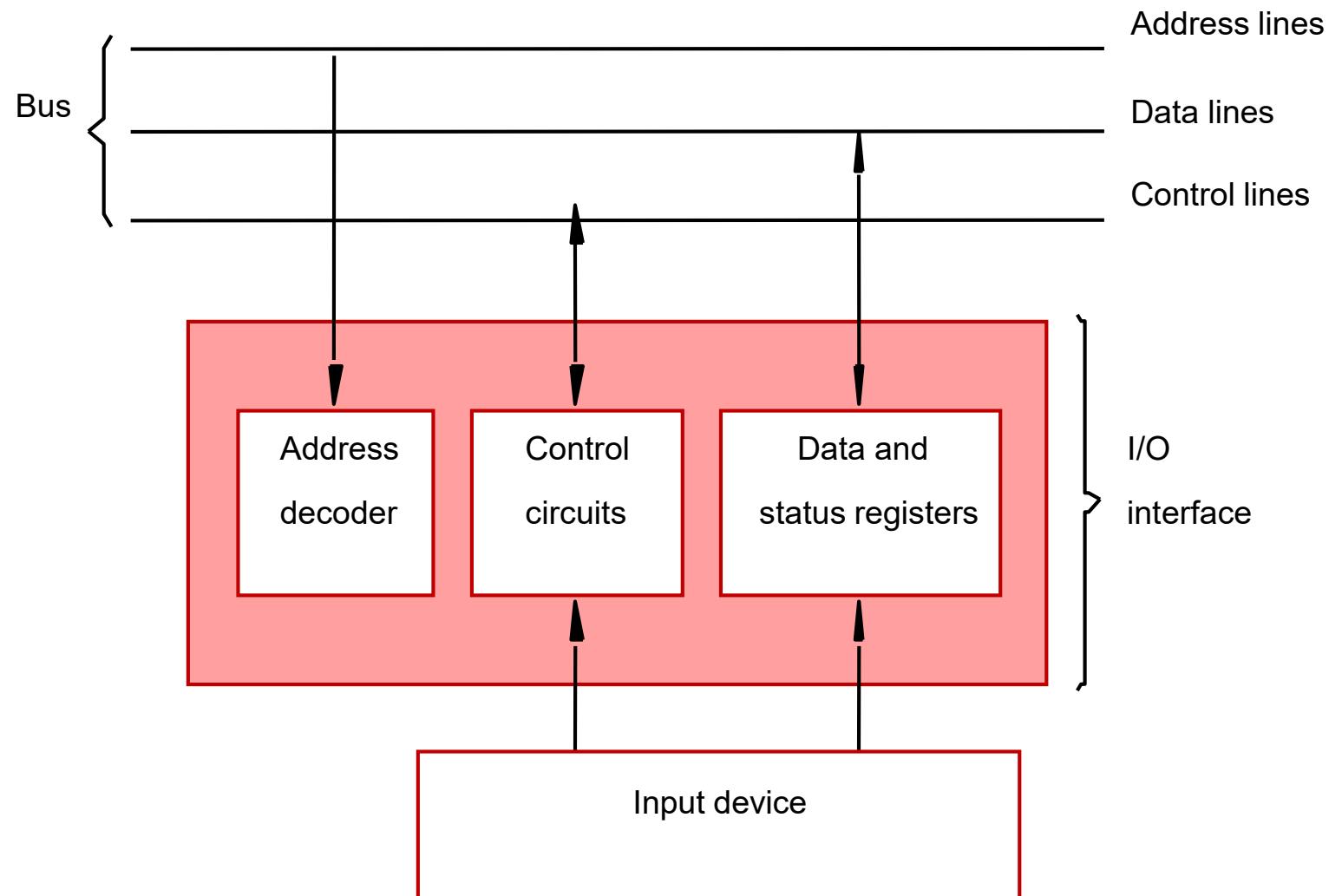
- Multiple I/O devices may be connected to the processor and the memory via a bus.
- Bus consists of three sets of lines to carry address, data and control signals.
- Each I/O device is assigned an unique address.
- To access an I/O device, the processor places the address on the address lines.
- The device recognizes the address, and responds to the control signals.

Accessing I/O devices (contd..)

- I/O devices and the memory may share the same address space:
 - **Memory-mapped I/O.**
 - Any machine instruction that can access memory can be used to transfer data to or from an I/O device.
 - Some addresses in the address space of the processor are assigned to these I/O locations-usually implemented as bit storage circuits (flip-flops) organized in the form of registers-*I/O registers*.
 - Load R2, DATAIN- reads the data from the DATAIN register and loads them into processor register R2.
 - Store R2, DATAOUT- sends the contents of register R2 to location DATAOUT, which is a register in an output device.

- **I/O devices and the memory may have different address spaces:**
 - Special instructions to transfer data to and from I/O devices.
 - I/O devices may have to deal with fewer address lines.
 - I/O address lines need not be physically separate from memory address lines.
 - In fact, address lines may be shared between I/O devices and memory, with a control signal to indicate whether it is a memory address or an I/O address.

Accessing I/O devices (contd..)



- I/O device is connected to the bus using an I/O interface circuit which has:
 - Address decoder, control circuit, and data and status registers.
- Address decoder decodes the address placed on the address lines thus enabling the device to recognize its address.
- Data register holds the data being transferred to or from the processor.
- Status register holds information necessary for the operation of the I/O device.
- Data and status registers are connected to the data lines, and have unique addresses.
- I/O interface circuit coordinates I/O transfers

Accessing I/O devices (contd..)

- The rate of transfer to and from I/O devices is slower than the speed of the processor. This creates the need for mechanisms to synchronize data transfers between them.
- **Program-controlled I/O:**
 - Processor repeatedly monitors a status flag to achieve the necessary synchronization.
 - Processor polls the I/O device.
- Two other mechanisms used for synchronizing data transfers between the processor and memory:
 - Interrupts.
 - Direct Memory Access.

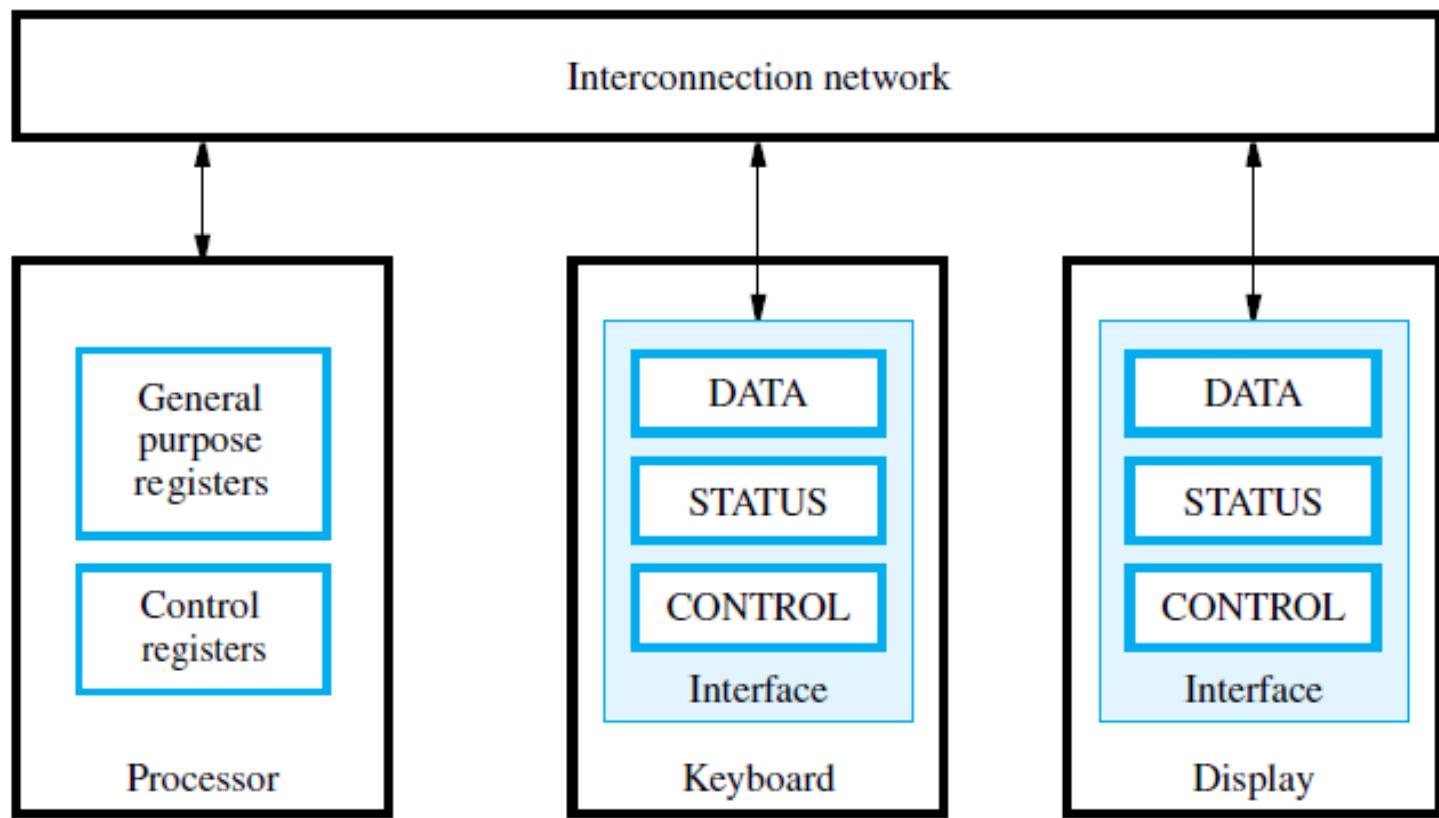


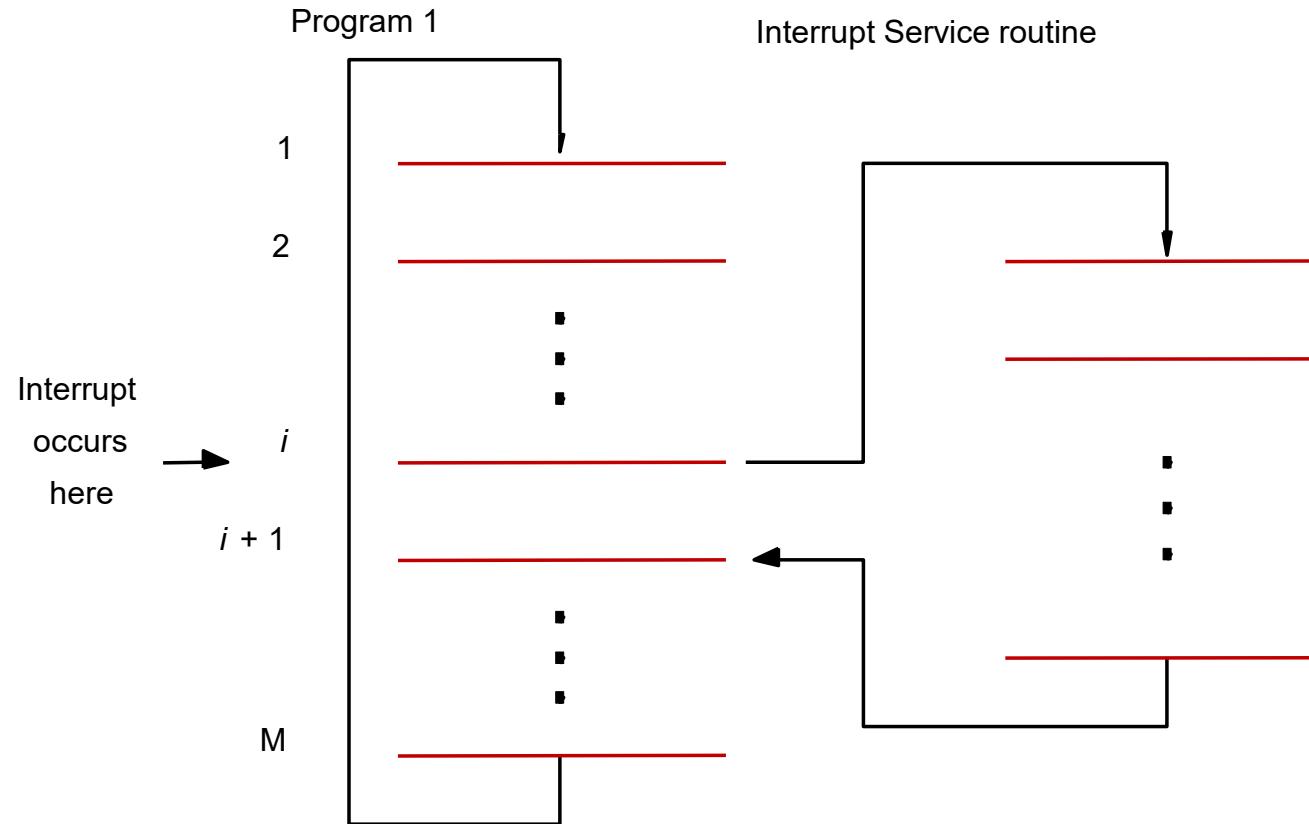
Figure 3.2 The connection for processor, keyboard, and display.

Interrupts

Interrupts

- In program-controlled I/O, when the processor continuously monitors the status of the device, it does not perform any useful tasks.
- **An alternate approach would be for the I/O device to alert the processor when it becomes ready.**
 - Do so by sending a hardware signal called an interrupt to the processor.
 - At least one of the bus control lines, called an interrupt-request line is dedicated for this purpose.
- **Processor can perform other useful tasks while it is waiting for the device to be ready.**

Interrupts (contd..)



Interrupts (contd..)

- Processor is executing the instruction located at address i when an interrupt occurs.
- Routine executed in response to an interrupt request is called the interrupt-service routine.
- When an interrupt occurs, control must be transferred to the interrupt service routine.
- But before transferring control, the current contents of the PC ($i+1$), must be saved in a known location.
- This will enable the return-from-interrupt instruction to resume execution at $i+1$.
- Return address, or the contents of the PC are usually stored on the processor stack.

Interrupts (contd..)

- Processor must inform the device that its request has been recognized so that it may remove its interrupt-request signal.
 - accomplished by means of a special control signal, called ***interrupt acknowledge***, which is sent to the device through the interconnection network.
 - the transfer of data between the processor and the I/O device interface
 - The execution of an instruction in the interrupt-service routine that accesses the status or data register in the device interface implicitly informs the device that its interrupt request has been recognized.

Interrupts (contd..)

- **Treatment of an interrupt-service routine is very similar to that of a subroutine.**
- **However there are significant differences:**
 - A subroutine performs a task that is required by the calling program.
 - Interrupt-service routine may not have anything in common with the program it interrupts.
 - Interrupt-service routine and the program that it interrupts may belong to different users.
 - Before branching to the interrupt-service routine, not only the PC, but other information such as condition code flags, and processor registers used by both the interrupted program and the interrupt service routine must be stored.
 - This will enable the interrupted program to resume execution upon return from interrupt service routine.

Interrupts (contd..)

- Saving and restoring information can be done automatically by the processor or explicitly by program instructions.
- Saving and restoring registers involves memory transfers:
 - Increases the total execution time.
 - Increases the delay between the time an interrupt request is received, and the start of execution of the interrupt-service routine. This delay is called interrupt latency.
- In order to reduce the interrupt latency, most processors save only the minimal amount of information:
 - This minimal amount of information includes Program Counter and processor status registers.
 - Any additional information that must be saved, must be saved explicitly by the program instructions at the beginning of the ISR
- In processors, with small number of registers, all registers are saved automatically by the processor hardware at the time an interrupt request is accepted and then restored to their respective registers as part of the execution of the Return-from-interrupt instruction

Interrupts (contd..)

- Some computers provide two types of interrupts
 - One saves all register contents, and the other does not.
- A particular I/O device may use either type, depending upon its response time requirements.
- Another interesting approach is to provide duplicate sets of processor registers.
 - a different set of registers can be used by the interrupt-service routine, thus eliminating the need to save and restore registers -*shadow registers*.

Enabling and Disabling of Interrupts

- Interrupt-requests interrupt the execution of a program, and may alter the intended sequence of events:
 - Sometimes such alterations may be undesirable, and must not be allowed.
 - For example, the processor may not want to be interrupted by the same device while executing its interrupt-service routine.
- Processors generally provide the ability to enable and disable such interruptions as desired.
- One simple way is to provide machine instructions such as *Interrupt-enable* and *Interrupt-disable* for this purpose.
- To avoid interruption by the same device during the execution of an interrupt service routine:
 - First instruction of an interrupt service routine can be Interrupt-disable.
 - Last instruction of an interrupt service routine can be Interrupt-enable.

Sequence of events involved in handling an interrupt request from a single device

1. The device raises an interrupt request.
2. The processor interrupts the program currently being executed and saves the contents of the PC and PS registers.
3. Interrupts are disabled by clearing the IE bit in the PS to 0.
4. The action requested by the interrupt is performed by the interrupt-service routine, during which time the device is informed that its request has been recognized, and in response, it deactivates the interrupt-request signal.
5. Upon completion of the interrupt-service routine, the saved contents of the PC and PS registers are restored (enabling interrupts by setting the IE bit to 1), and execution of the interrupted program is resumed.

Handling multiple devices

- Multiple I/O devices may be connected to the processor and the memory via a bus. Some or all of these devices may be capable of generating interrupt requests.
- Each device operates independently, and hence no definite order can be imposed on how the devices generate interrupt requests?
- How does the processor know which device has generated an interrupt?
- How does the processor know which interrupt service routine needs to be executed?
- When the processor is executing an interrupt service routine for one device, can other device interrupt the processor?
- If two interrupt-requests are received simultaneously, then how to break the tie?

Interrupts (contd..)

- Consider a simple arrangement where all devices send their interrupt-requests over a single control line in the bus.
- When the processor receives an interrupt request over this control line, how does it know which device is requesting an interrupt?
- This information is available in the status register of the device requesting an interrupt:
 - The status register of each device has an *IRQ* bit which it sets to 1 when it requests an interrupt.
- Interrupt service routine can poll the I/O devices connected to the bus. The first device with *IRQ* equal to 1 is the one that is serviced.
- Polling mechanism is easy, but time consuming to query the status bits of all the I/O devices connected to the bus.

Vectored Interrupts

- The device requesting an interrupt may identify itself directly to the processor.
 - Device can do so by sending a special code (4 to 8 bits) to the processor over the bus.
- The processor's circuits determine the memory address of the required interrupt-service routine.
- A commonly used scheme is to allocate permanently an area in the memory to hold the addresses of interrupt-service routines.
- These addresses are usually referred to as *interrupt vectors*, and they are said to constitute the *interrupt-vector table*

- When an interrupt request arrives, the information provided by the requesting device is used as a pointer into the interrupt-vector table, and the address in the corresponding interrupt vector is automatically loaded into the program counter

Nesting of Interrupts

- During the execution of an interrupt service routine of device, the processor does not accept interrupt requests from any other device.
- Since the interrupt service routines are usually short, the delay that this causes is generally acceptable.
- However, for certain devices this delay may not be acceptable.
 - Which devices can be allowed to interrupt a processor when it is executing an interrupt service routine of another device?

Interrupts (contd..)

- I/O devices are organized in a priority structure:
 - An interrupt request from a high-priority device is accepted while the processor is executing the interrupt service routine of a low priority device.
- A priority level is assigned to a processor that can be changed under program control.
 - or
- Priority level of a processor is the priority of the program that is currently being executed.
 - When the processor starts executing the interrupt service routine of a device, its priority is raised to that of the device.
 - If the device sending an interrupt request has a higher priority than the processor, the processor accepts the interrupt request.
 - Processor's priority is encoded in a few bits of the processor status register.

Simultaneous Requests

- The processor must have some means of deciding which request to service first.
- Polling the status registers of the I/O devices is the simplest such mechanism. In this case, priority is determined by the order in which the devices are polled.
- When vectored interrupts are used, we must ensure that only one device is selected to send its interrupt vector code. This is done in hardware, by using arbitration circuits

Direct Memory Access

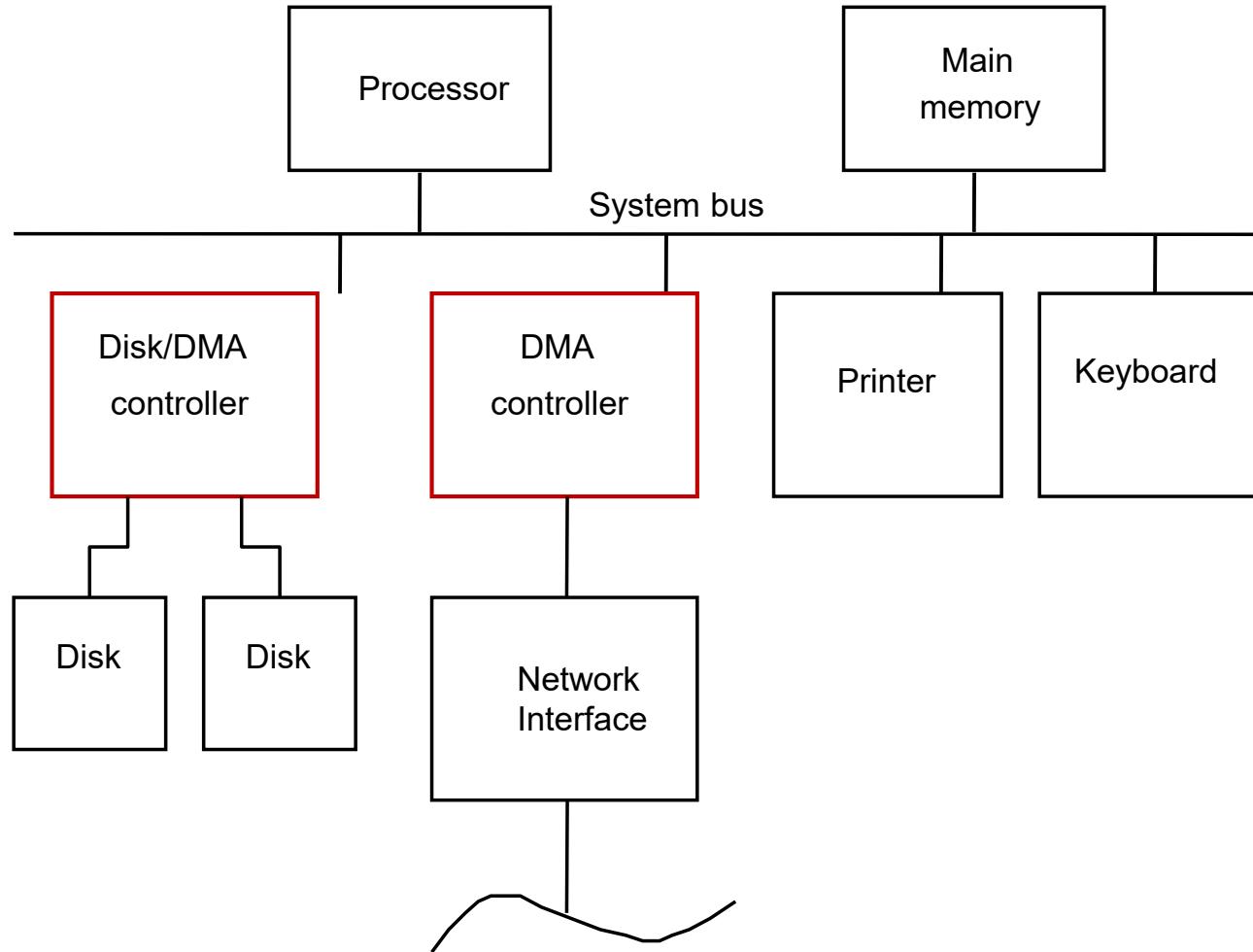
Direct Memory Access (contd..)

- Direct Memory Access (DMA):
 - A special control unit may be provided to transfer a block of data directly between an I/O device and the main memory, without continuous intervention by the processor.
- Control unit which performs these transfers is a part of the I/O device's interface circuit. This control unit is called as a DMA controller.
- DMA controller performs functions that would be normally carried out by the processor:
 - For each word, it provides the memory address and all the control signals.
 - To transfer a block of data, it increments the memory addresses and keeps track of the number of transfers.

Direct Memory Access (contd..)

- DMA controller can transfer a block of data from an external device to the processor, without any intervention from the processor.
 - However, the operation of the DMA controller must be under the control of a program executed by the processor. That is, the processor must initiate the DMA transfer.
- To initiate the DMA transfer, the processor informs the DMA controller of:
 - Starting address,
 - Number of words in the block.
 - Direction of transfer (I/O device to the memory, or memory to the I/O device).
- Once the DMA controller completes the DMA transfer, it informs the processor by raising an interrupt signal.

Direct Memory Access



Use of DMA controllers in a computer system

Use of DMA controllers in a computer system

- DMA controller connects a high-speed network to the computer bus.
- Disk controller, which controls two disks also has DMA capability.
- It provides two DMA channels.
- It can perform two independent DMA operations, as if each disk has its own DMA controller.
- The registers to store the memory address, word count and status and control information are duplicated.

Reference

- Carl Hamacher, Zvonko Vranesic, Safwat Zaky, Naraig Manjikian,
“Computer Organization And Embedded Systems” 6th Edition,
McGraw-Hill (Selected sections from Chapters 3 and 8)

Pipelining

Overview

- Pipelining is widely used in modern processors.
- Pipelining improves system performance in terms of throughput.
- Pipelined organization requires sophisticated compilation techniques.

Basic Concepts

Pipelined Execution

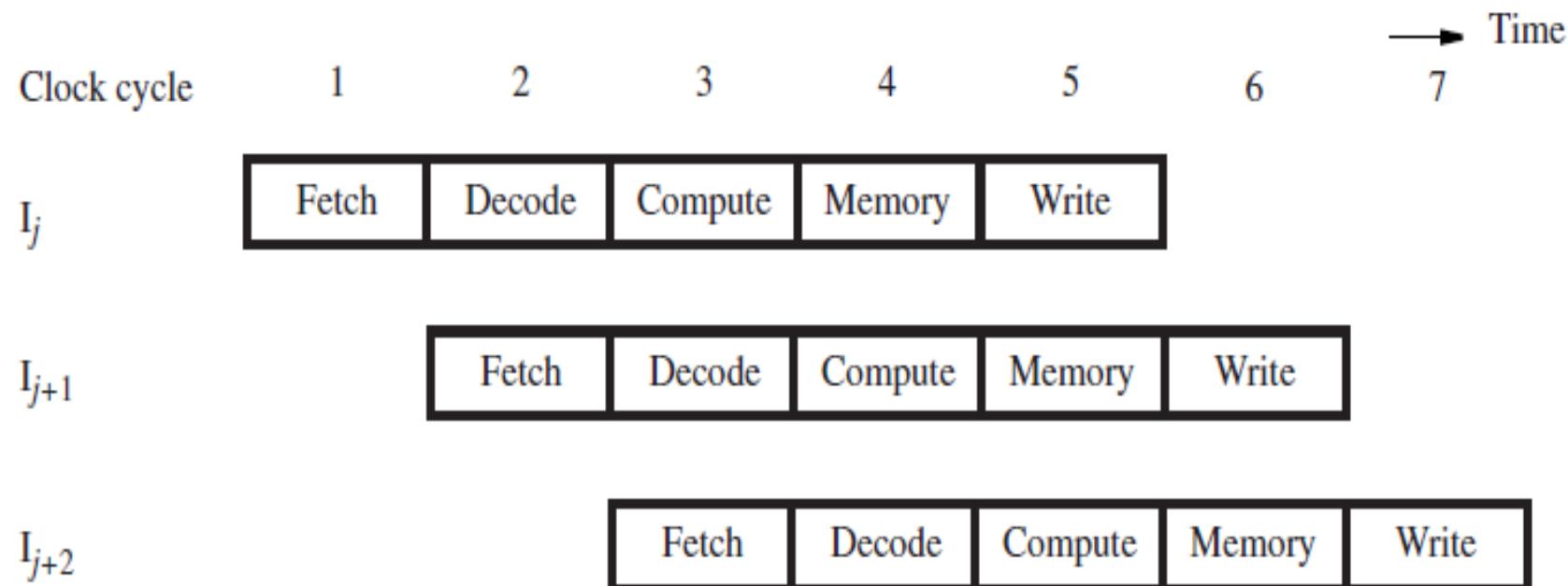


Figure 6.1 Pipelined execution—the ideal case.

Pipeline Organization

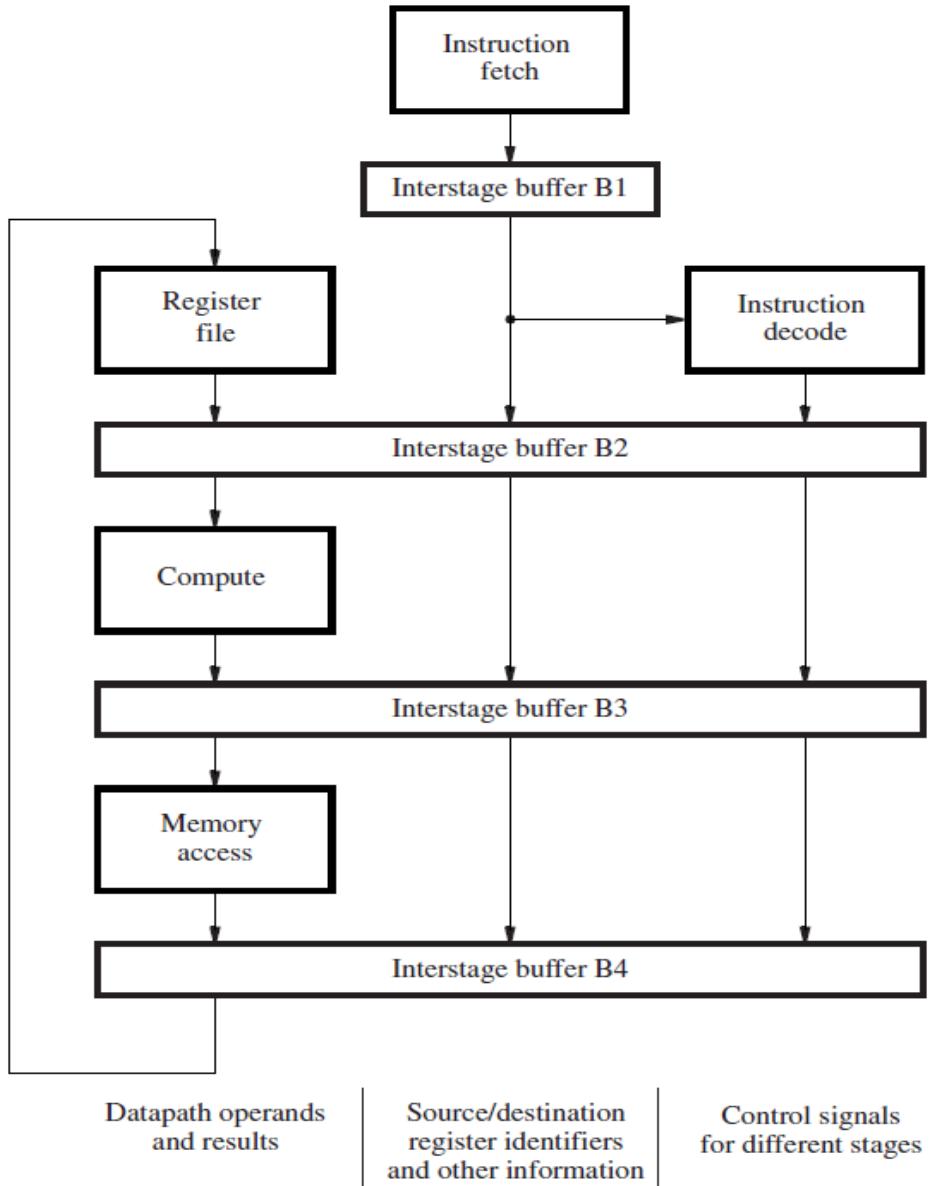


Figure 6.2 A five-stage pipeline.

The interstage buffers are used as follows

- Interstage buffer B1 feeds the Decode stage with a newly-fetched instruction.
- Interstage buffer B2 feeds the Compute stage with the operands read from the register file
- Interstage buffer B3 holds the result of the ALU operation, which may be data to be written into the register file or an address that feeds the Memory stage

In case of a write access to memory, buffer B3 holds the data to be written

- Interstage buffer B4 feeds the Write stage with a value to be written into the register file.

Pipelining Issues

- Consider the case of two instructions, I_j and I_{j+1} , where the destination register for instruction I_j is a source register for instruction I_{j+1} .
- To obtain the correct result it is necessary to wait until the new value is written into the register by instruction I_j .
- Hence, instruction I_{j+1} cannot read its operand until cycle 6, which means it must be *stalled* in the Decode stage for three cycles. While instruction I_{j+1} is stalled, instruction I_{j+2} and all subsequent instructions are similarly delayed. New instructions cannot enter the pipeline, and the total execution time is increased.

Hazard

- Any condition that causes the pipeline to stall is called a *hazard*.
 - Data Hazard
 - memory delays
 - branch instructions
 - resource limitations

Data Hazards

Add R2, R3, #100

Subtract R9, R2, #30

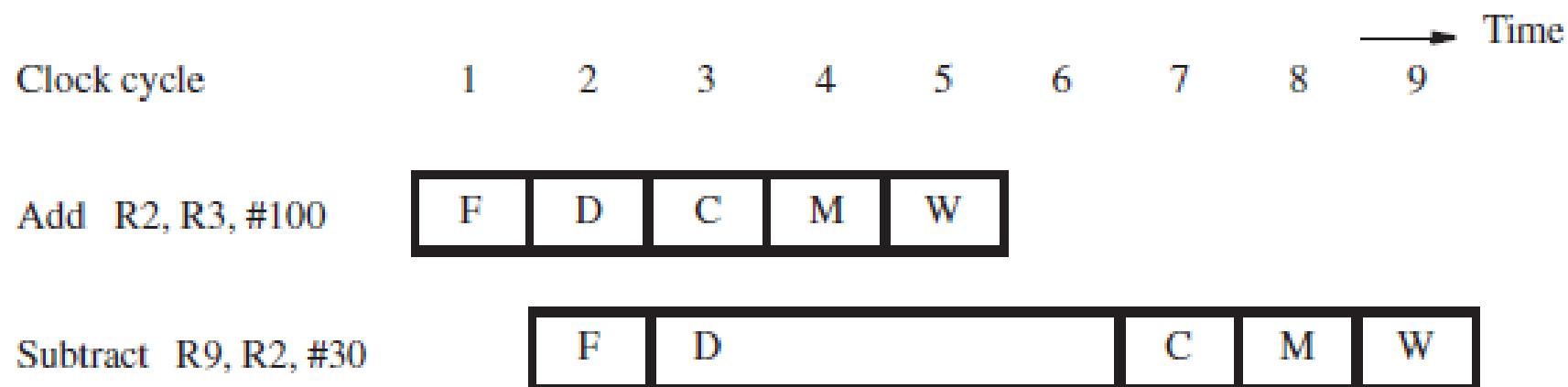


Figure 6.3 Pipeline stall due to data dependency.

Operand Forwarding

- Instead of from the register file, the second instruction can get data directly from the output of ALU after the previous instruction is completed.
- A special arrangement needs to be made to “forward” the output of ALU to the input of ALU.

Operand Forwarding (Contd..)

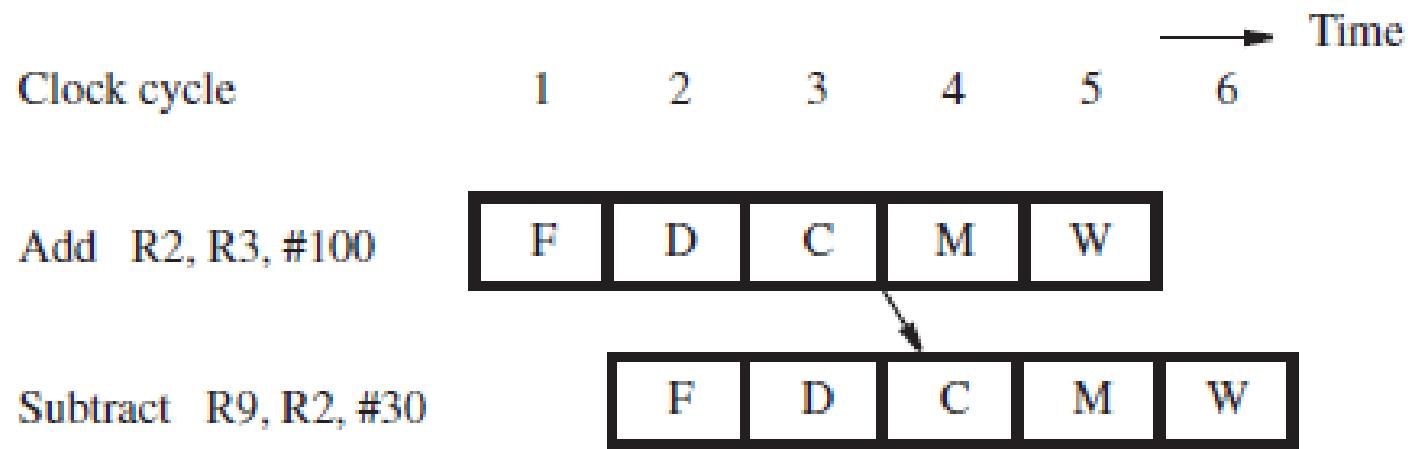


Figure 6.4 Avoiding a stall by using operand forwarding.

Handling Data Dependencies in Software

Add	R2, R3, #100
NOP	
NOP	
NOP	
Subtract	R9, R2, #30

(a) Insertion of NOP instructions for a data dependency

- compiler identifies a data dependency between two successive instructions I_j and I_{j+1} , it can insert three explicit NOP (No-operation) instructions between them.
- The NOPs introduce the necessary delay to enable instruction I_{j+1} to read the new value from the register file after it is written.
- The compiler can reorder the instructions to perform some useful work during the NOP slots

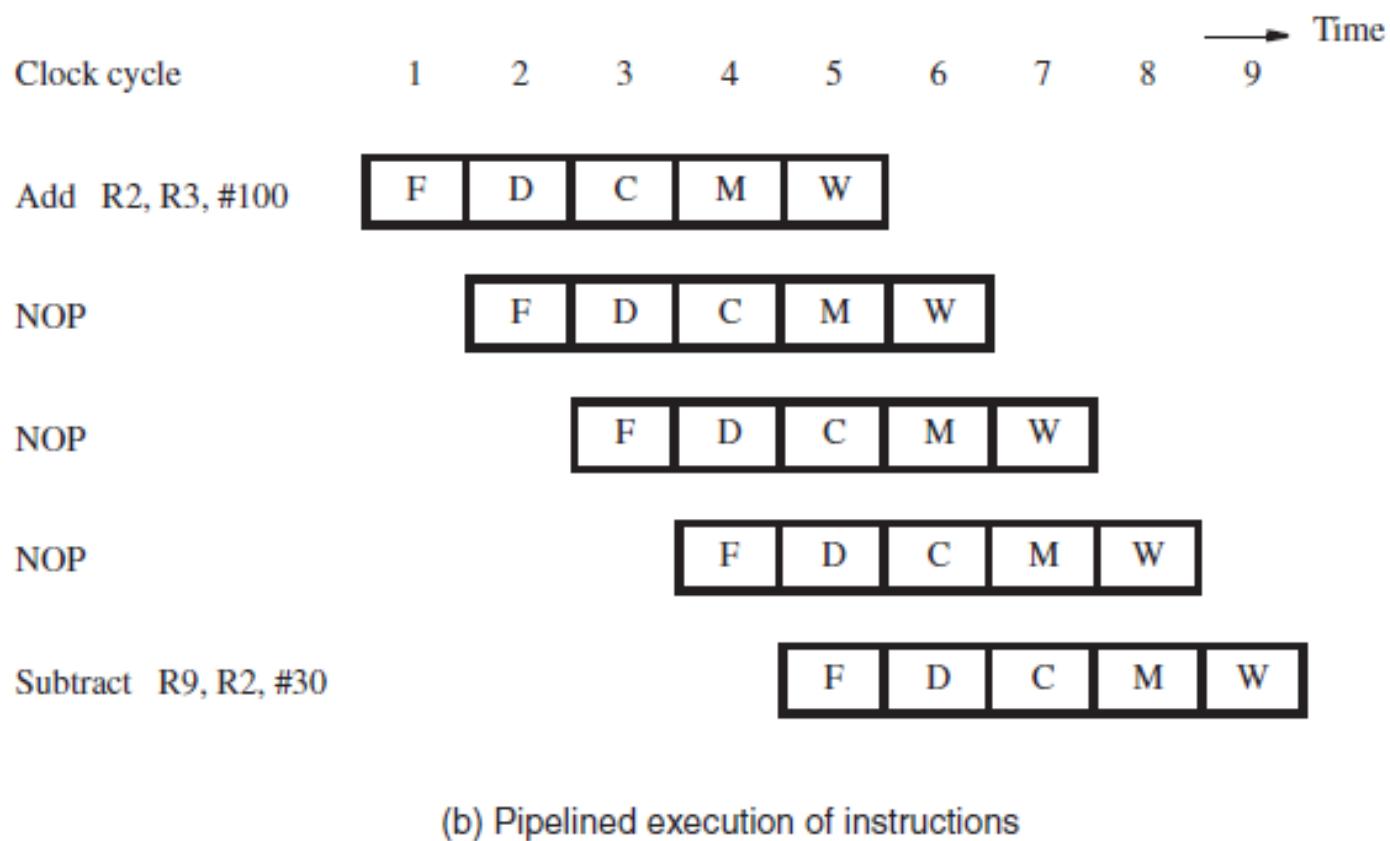


Figure 6.6 Using NOP instructions to handle a data dependency in software.

Memory Delays

- Delays arising from memory accesses are another cause of pipeline stalls.
- Load instruction may require more than one clock cycle to obtain its operand from memory.
- This may occur because the requested instruction or data are not found in the cache, resulting in a *cache miss*.
- A memory access may take ten or more cycles. For simplicity, the figure shows only three cycles.

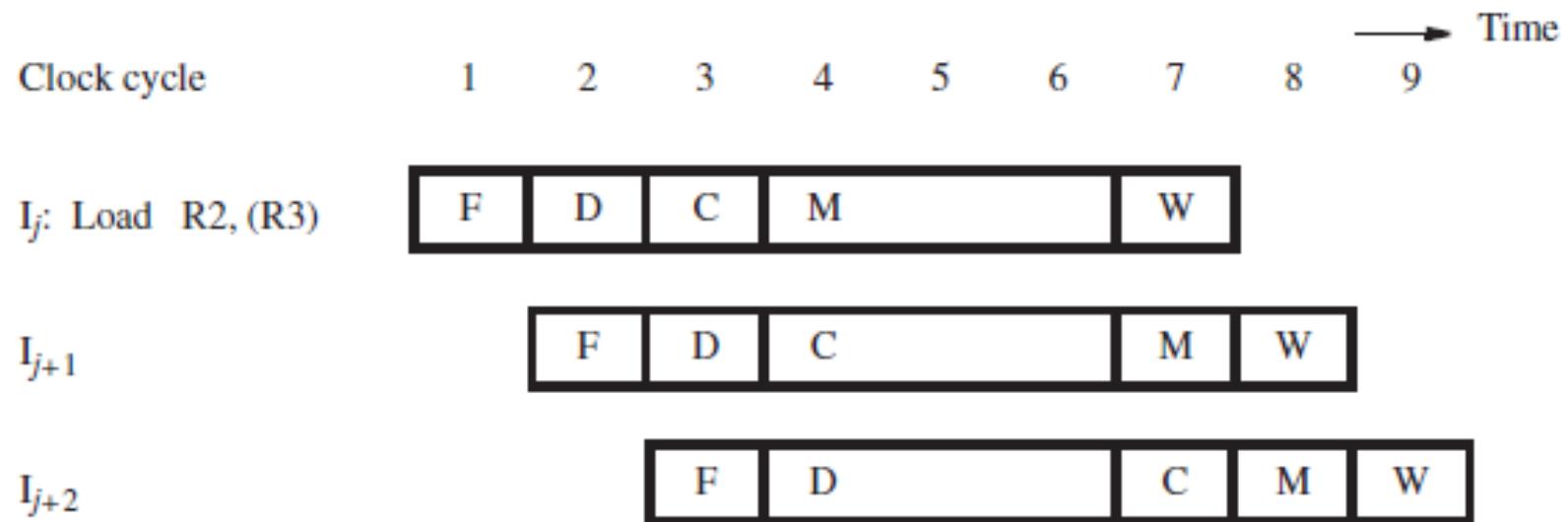


Figure 6.7 Stall caused by a memory access delay for a Load instruction.

Load R2, (R3)
Subtract R9, R2, #30

- Assume that the data for the Load instruction is found in the cache, requiring only one cycle to access the operand.
- The destination register R2 for the Load instruction is a source register for the Subtract instruction.
- Operand forwarding cannot be done because the data read from memory (the cache, in this case) are not available until they are loaded into register
- the Subtract instruction must be stalled for one cycle

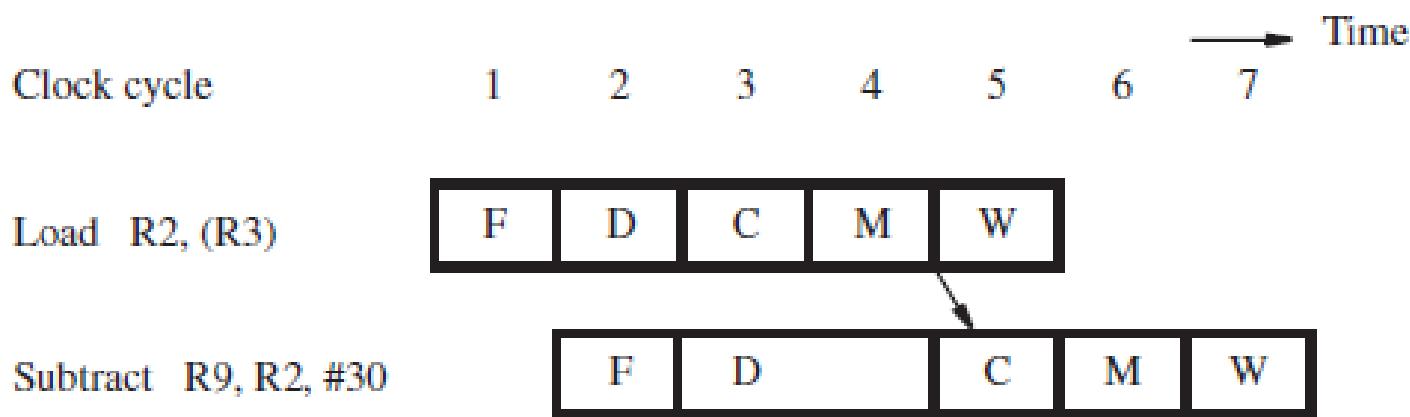


Figure 6.8 Stall needed to enable forwarding for an instruction that follows a Load instruction.

Instruction Hazards

Overview

- Whenever the stream of instructions supplied by the instruction fetch unit is interrupted, the pipeline stalls.
- Branch

Unconditional Branch

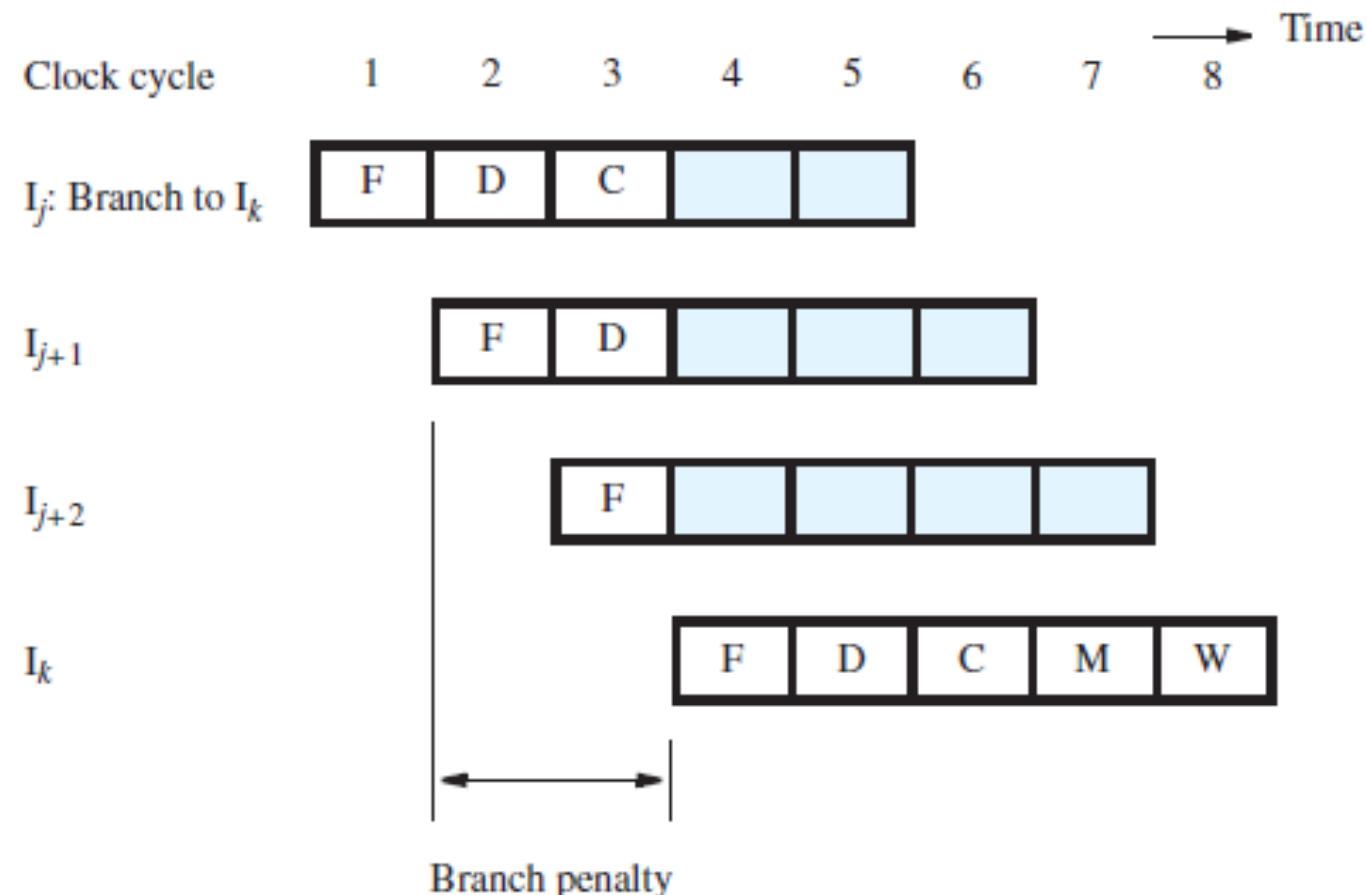


Figure 6.9 Branch penalty when the target address is determined in the Compute stage of the pipeline.

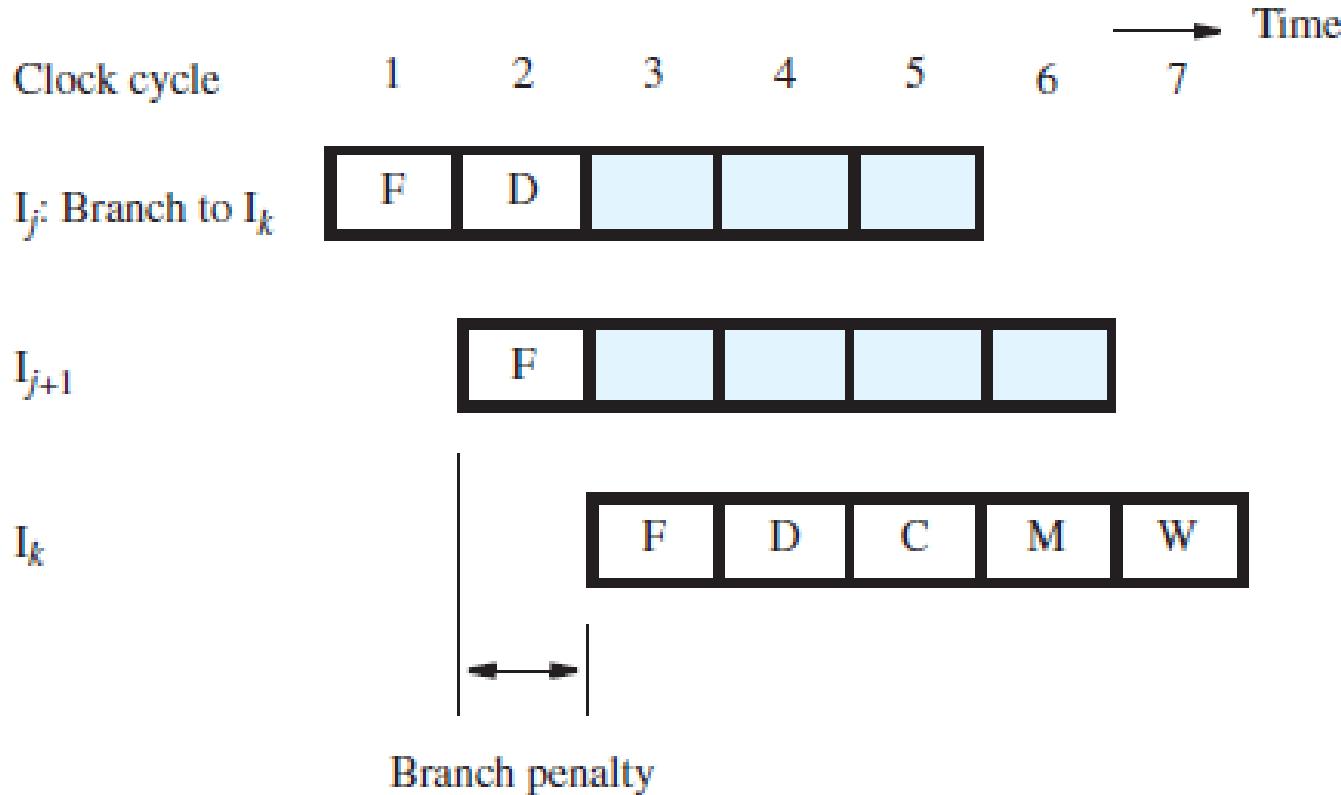


Figure 6.10 Branch penalty when the target address is determined in the Decode stage of the pipeline.

Conditional Branches

Branch_if_[R5]=[R6] LOOP

- A conditional branch instruction introduces the added hazard caused by the dependency of the branch condition on the result of a preceding instruction.
- The result of the comparison in the third step determines whether the branch is taken
- the branch condition must be tested as early as possible to limit the branch penalty.
- The decision to branch cannot be made until the execution of that instruction has been completed.

Conditional Branches Contd...

- the comparator that tests the branch condition can also be moved to the Decode stage, enabling the conditional branch decision to be made at the same time that the target address is determined.
- In this case, the comparator uses the values from outputs A and B of the register file directly.

Delayed Branch

- The location that follows a branch instruction is called the ***branch delay slot***
- The instructions in the delay slots are always fetched. Therefore, arrange for them to be fully executed whether or not the branch is taken.
- Place useful instructions in these slots.
- The effectiveness depends on how often it is possible to reorder instructions.
- branching takes place one instruction later than where the branch instruction appears in the instruction sequence. This technique is called ***delayed branching***.

The Branch Delay Slot

Add	R7, R8, R9
Branch_if_[R3]=0	TARGET
I_{j+1}	
\vdots	
TARGET: I_k	

(a) Original sequence of instructions containing
a conditional branch instruction

Branch_if_[R3]=0	TARGET
Add	R7, R8, R9
I_{j+1}	
\vdots	
TARGET: I_k	

(b) Placing the Add instruction in the branch delay
slot where it is always executed

Figure 6.11 Filling the branch delay slot with a useful instruction.

Reference

- Chapter 6 -Carl Hamacher, Zvonko Vranesic,Safwat Zaky, Naraig Manjikian, “*Computer Organization And Embedded Systems*” 6th Edition,, McGraw-Hill

Parallel Processing

Hardware Multithreading.

- Operating system (OS) software enables multitasking of different programs in the same processor by performing context switches among programs
- a program, together with any information that describes its current state of execution, is called a *process*
- Information about the memory and other resources allocated by the OS is maintained with each process.
- Each process has a corresponding thread, which is an independent path of execution within a program.

Hardware Multithreading -Threads

- the term *thread* is used to refer to a thread of control whose state consists of the contents of the program counter and other processor registers
- It is possible for multiple threads to execute portions of one program and run in parallel as if they correspond to separate programs.
- Two or more threads can be running on different processors, executing either the same part of a program on different data, or executing different parts of a program.
- Threads for different programs can also execute on different processors.
- All threads that are part of a single program run in the same address space and are associated with the same process

Hardware Multithreading-Contd..

- We focus on multitasking where two or more programs run on the same processor and each program has a single thread
- OS selects a process among those that are not presently blocked and allows this process to run for a short period of time.
- Only the thread corresponding to the selected process is active during the time slice.
- Context switching at the end of the time slice causes the OS to select a different process, whose corresponding thread becomes active during the next time slice.
- A timer interrupt invokes an interrupt-service routine in the OS to switch from one process to another

Hardware multithreading.

- Processor is implemented with several identical sets of registers, including multiple program counters.
- Each set of registers can be dedicated to a different thread.
- Thus, no time is wasted during a context switch to save and restore register contents.
- The processor is said to be using a technique called ***hardware multithreading***.

- With multiple sets of registers, context switching is simple and fast.
- All that is necessary is to change a hardware pointer in the processor to use a different set of registers to fetch and execute subsequent instructions.
- Switching to a different thread can be completed within one clock cycle.
- The state of the previously active thread is preserved in its own set of registers.

Coarse-grained multithreading

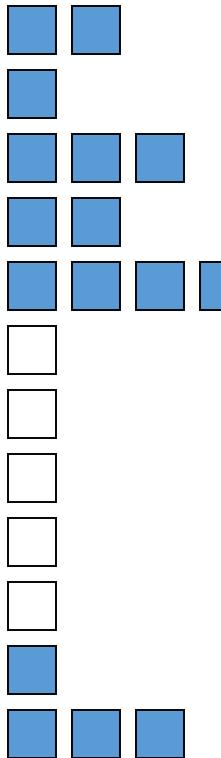
- Switching to a different thread may be triggered at any time by the occurrence of a specific event, rather than at the end of a fixed time interval
- a cache miss may occur when a Load or Store instruction is being executed for the active thread. Instead of stalling while the slower main memory is accessed to service the cache miss, a processor can quickly switch to a different thread and continue to fetch and execute other instructions.
- This is called ***coarse-grained*** multithreading because many instructions may be executed for one thread before an event such as a cache miss causes a switch to another thread

Fine-grained or interleaved multithreading

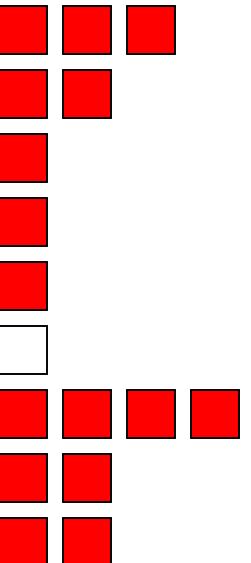
- Switch after every instruction is fetched.
- This is called ***fine-grained or interleaved*** multithreading.
- The intent is to increase the processor throughput.
- Each new instruction is independent of its predecessors from other threads.
- This should reduce the occurrence of stalls due to data dependencies.
- Throughput is increased by interleaving instructions from many threads
- It takes longer for a given thread to complete all of its instructions.

Conceptual Diagram

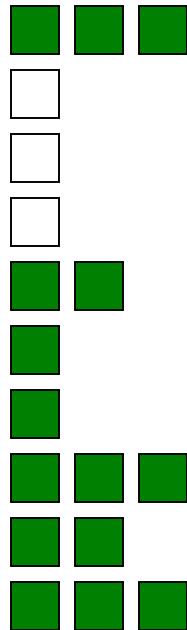
Thread A



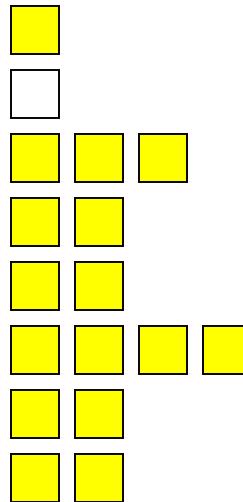
Thread B



Thread C

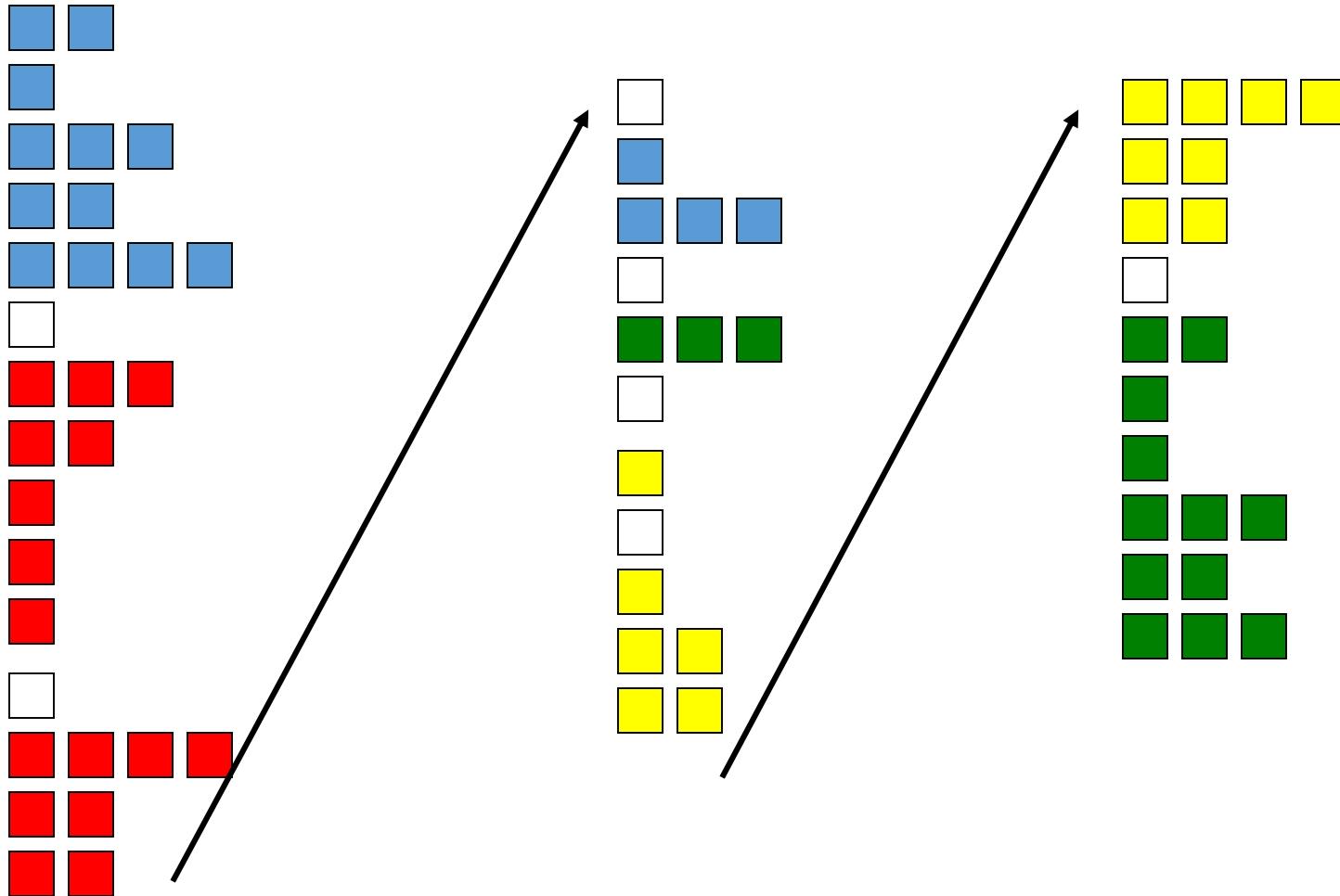


Thread D

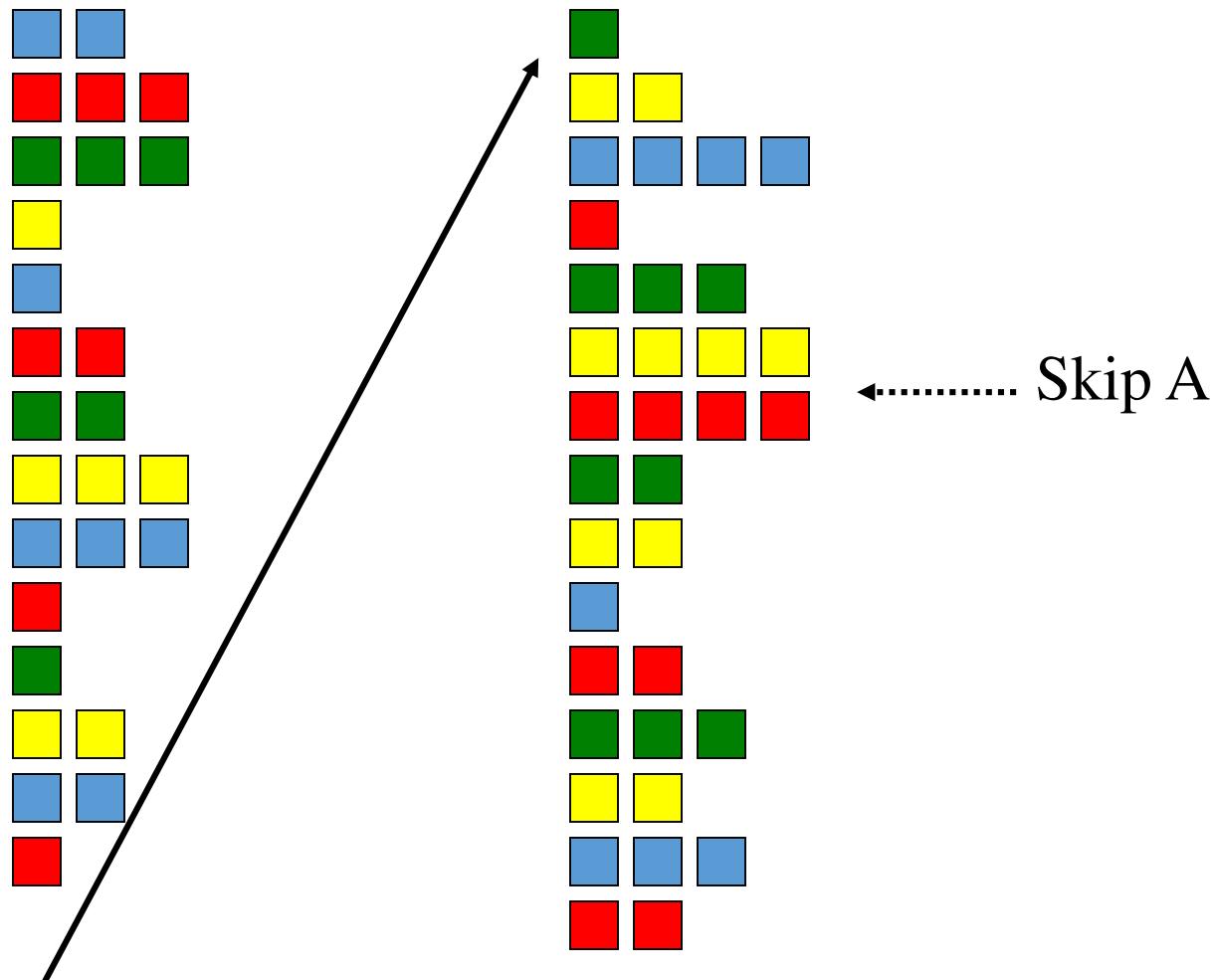


Coarse Multithreading

Stalls for A and C would be longer than indicated in previous slide
Assume long stalls at end of each thread indicated in previous slide



Fine Multithreading



Vector (SIMD) Processing

- programs use loops to perform operations on vectors of data, where a vector is an array of elements such as integers or floating-point numbers.
- When a processor executes the instructions in such a loop, the operations are performed one at a time on individual vector elements.
- Many instructions need to be executed to process all vector elements.

Vector (SIMD) Processing

- A processor can be enhanced with multiple ALUs.
- It is possible to operate on multiple data elements in parallel using a single instruction.
- Such instructions are called single-instruction multiple-data (SIMD) instructions. They are also called vector instructions.
- These instructions can only be used when the operations performed in parallel are independent. This is known as data parallelism.

Vector (SIMD) Processing

- The data for vector instructions are held in **vector registers**, each of which can hold several data elements. The number of elements, L, in each vector register is called the **vector length**.
- It determines the number of operations that can be performed in parallel on multiple ALUs.

Vectorization

- In a source program written in a high-level language, loops that operate on arrays of integers or floating-point numbers are vectorizable if the operations performed in each pass are independent of the other passes.
- Using vector instructions reduces the number of instructions that need to be executed and enables the operations to be performed in parallel on multiple ALUs.
- A vectorizing compiler can recognize such loops, if they are not too complex, and generate vector instructions.

Vector (SIMD) Processing

- The vector instruction

VectorAdd.S Vi, Vj, Vk

computes L sums using the elements in vector registers Vj and Vk, and places the resulting sums in vector register Vi.

- Special instructions are needed to transfer multiple data elements between a vector register and the memory. The instruction

VectorLoad.S Vi, X(Rj)

causes L consecutive elements beginning at memory location X + [Rj] to be loaded into vector register Vi. Similarly, the instruction

VectorStore.S Vi, X(Rj)

- causes the contents of vector register Vi to be stored as L consecutive locations in the memory.

Vectorization Example

- Consider vectorization of the loop given below

```
for (i = 0; i < N; i++)  
    A[i] = B[i] + C[i];
```

(a) A C-language loop to add vector elements

- Assume that the starting locations in memory for arrays A, B, and C are in registers R2, R3, and R4. Using conventional assembly-language instructions, the compiler may generate the loop.

LOOP:	Move	R5, #N	R5 is the loop counter.
	Load	R6, (R3)	R3 points to an element in array B.
	Load	R7, (R4)	R4 points to an element in array C.
	Add	R6, R6, R7	Add a pair of elements from the arrays.
	Store	R6, (R2)	R2 points to an element in array A.
	Add	R2, R2, #4	Increment the three array pointers.
	Add	R3, R3, #4	
	Add	R4, R4, #4	
	Subtract	R5, R5, #1	Decrement the loop counter.
	Branch_if_[R5]>0	LOOP	Repeat the loop if not finished.

(b) Assembly-language instructions for the loop

Vectorization Example Contd..

- The Load, Add, and Store instructions at the beginning of the loop are replaced by corresponding vector instructions that operate on L elements at a time.
- The vectorized loop requires only N/L passes to process all of the data in the arrays.
- With L elements processed in each pass through the loop, the address pointers in registers R2, R3, and R4 are incremented by $4L$, and the count in register R5 is decremented by L .

Vectorization Example Contd..

Vectorized form of the loop

	Move	R5, #N	R5 counts the number of elements to process.
LOOP:	VectorLoad.S	V0, (R3)	Load L elements from array B.
	VectorLoad.S	V1, (R4)	Load L elements from array C.
	VectorAdd.S	V0, V0, V1	Add L pairs of elements from the arrays.
	VectorStore.S	V0, (R2)	Store L elements to array A.
	Add	R2, R2, #4*L	Increment the array pointers by L words.
	Add	R3, R3, #4*L	
	Add	R4, R4, #4*L	
	Subtract	R5, R5, #L	Decrement the loop counter by L .
	Branch_if_[R5]> 0	LOOP	Repeat the loop if not finished.

(c) Vectorized form of the loop

GPU

- specialized chips called **Graphics Processing Units (GPUs)**.
- to accelerate the large number of floating-point calculations needed in high-resolution, three-dimensional graphics, such as in video games.
- a large GPU chip contains hundreds of simple cores with floating-point ALUs to perform them in parallel.

GPU Contd...

- A small program is written for the processing cores in the GPU chip.
- A large number of cores execute this program in parallel.
- The cores execute the same instructions, but operate on different data elements.
- A separate controlling program runs in the general-purpose processor of the host computer and invokes the GPU program when necessary.
- Before initiating the GPU computation, the program in the host computer must first transfer the data needed by the GPU program from the main memory into the dedicated GPU memory.
- After the computation is completed, the resulting output data in the dedicated memory are transferred back to the main memory.

GPU Contd..

- The processing cores in a GPU chip have a specialized instruction set and hardware architecture, which are different from those used in a general-purpose processor.
 - e.g *Compute Unified Device Architecture* (CUDA) that NVIDIA Corporation uses for the cores in its GPU chips
- To facilitate writing programs that involve a general-purpose processor and a GPU, an extension to the C programming language, called CUDA C, has been developed by NVIDIA
- This extension enables a single program to be written in C, with special keywords used to label the functions executed by the processing cores in a GPU chip.

GPU Contd..

- The compiler and related software tools automatically partition the final object program into the portions that are translated into machine instructions for the host computer and the GPU chip.
- Library routines are provided to allocate storage in the dedicated memory of a GPU-based video card and to transfer data between the main memory and the dedicated memory.
- An open standard called OpenCL has also been proposed by industry as a programming framework for systems that include GPU chips from any vendor

Shared-Memory Multiprocessors

- All processors have access to the same memory.
- Tasks running in different processors can access shared variables in the memory using the same addresses.
- The size of the shared memory is likely to be large.
- bottleneck when many processors make requests to access the memory simultaneously.
- Distributed across multiple modules so that simultaneous requests from different processors are more likely to access different memory modules, depending on the addresses of those requests.

Shared-Memory Multiprocessors-UMA

- Interconnection networks-enables any processor to access any module that is a part of the shared memory.
- All requests to access memory must pass through the network, which introduces latency.
- A system which has the same network latency for all accesses from the processors to the memory modules is called a Uniform Memory Access (UMA) multiprocessor.

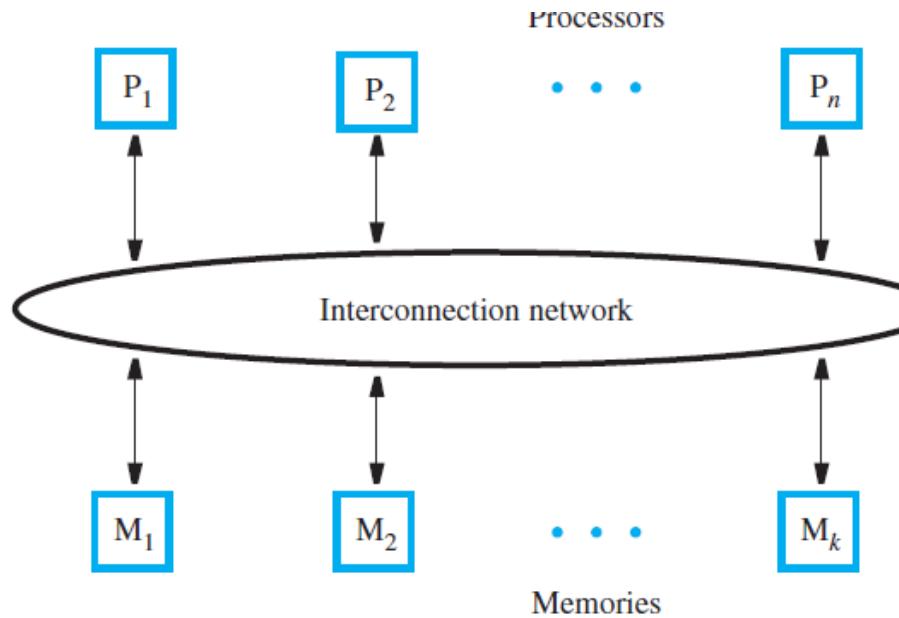


Figure 12.2 A UMA multiprocessor.

NUMA

- It is desirable to place a memory module close to each processor.
- The result is a collection of nodes, each consisting of a processor and a memory module.
- Nodes are then connected to the network, as shown in Figure 12.3.
- The n/w latency is avoided when a processor makes a request to access its local memory.
- a request to access a remote memory module must pass through the n/w
- Because of the difference in latencies for accessing local and remote portions of the shared memory, systems of this type are called Non-Uniform Memory Access (NUMA) multiprocessors.

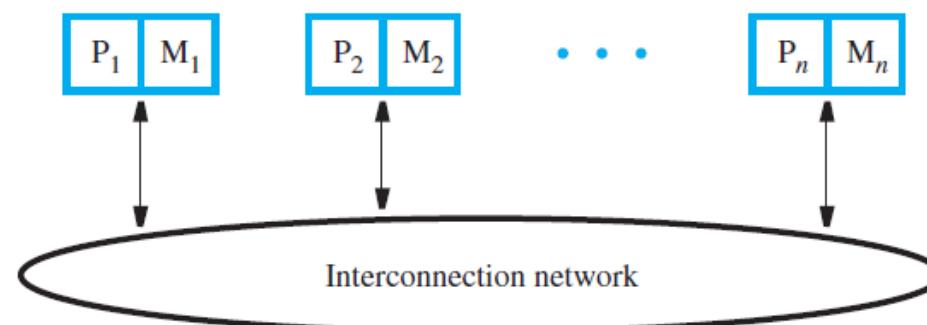


Figure 12.3 A NUMA multiprocessor.

Interconnection networks

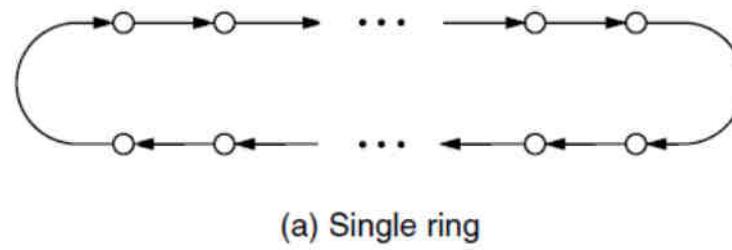
- must allow information transfer between any pair of nodes in the system.
- may also be used to broadcast information from one node to many other nodes.
- The traffic in the network consists of requests (such as read and write) and data transfers.
- The suitability of a particular network is judged in terms of cost, bandwidth, effective
- throughput, and ease of implementation.
- The term *bandwidth* refers to the capacity of a transmission link to transfer data and is expressed in bits or bytes per second.
- The *effective throughput* is the actual rate of data transfer.
 - This rate is less than the available bandwidth because a given link must also carry control information that coordinates the transfer of data

Bus

- Set of lines (wires) that provide a single shared path for information transfer
- Buses are most commonly used in UMA multiprocessors to connect a number of processors to several shared-memory modules.
- Arbitration is necessary to ensure that only one of many possible requesters is granted use of the bus at any time.
- The bus is suitable for a relatively small number of processors
 - contention for access to the bus
 - increased propagation delays when many processors are connected.

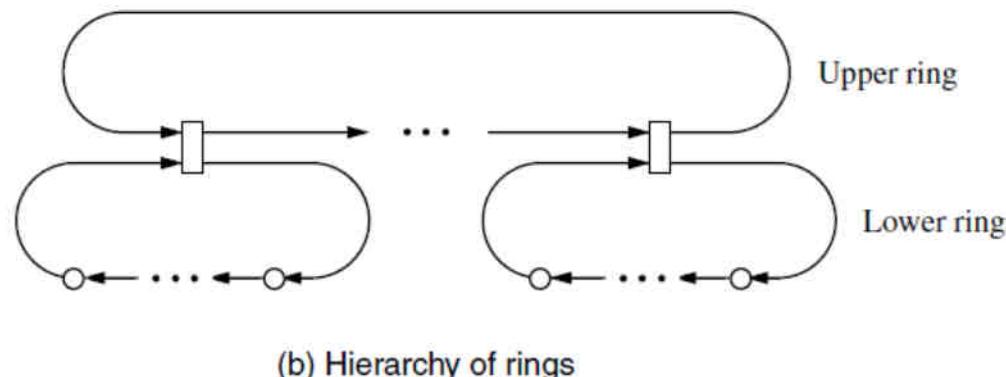
Interconnection Networks-Ring Based

- A ring network is formed with point-to-point connections between nodes
- A long single ring results in high average latency for communication between any two nodes.
- This high latency can be mitigated in two different ways.
- A second ring can be added to connect the nodes in the opposite direction. The resulting bidirectional ring halves the average latency and doubles the bandwidth. However, handling of communications is more complex



Hierarchy of rings

- A two-level hierarchy is shown
- The upper-level ring connects the lower-level rings. The average latency for communication between any two nodes on lower-level rings is reduced with this arrangement.
- Transfers between nodes on the same lower-level ring need not traverse the upper-level ring.
- Transfers between nodes on different lower-level rings include a traversal on part of the upper-level ring.
- The drawback of the hierarchical scheme is that the upper-level ring may become a bottleneck when many nodes on different lower-level rings communicate with each other frequently.



Crossbar Network

A crossbar is a network that provides a direct link between any pair of units connected to the network.

It is typically used in UMA multiprocessors to connect processors to memory modules. It enables many simultaneous transfers if the same destination is not the target of multiple requests.

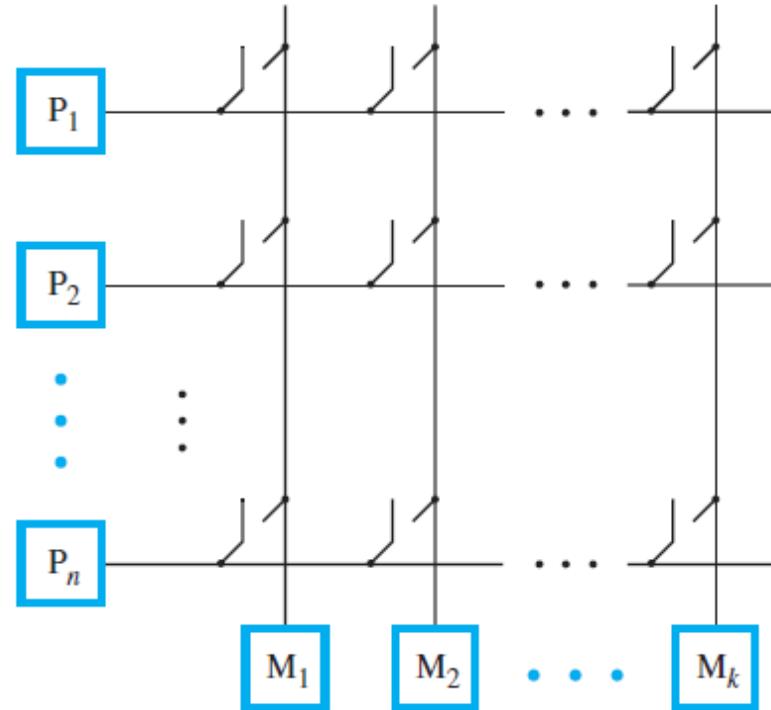


Figure 12.5 Crossbar interconnection network.

Mesh

- A natural way of connecting a large number of nodes is with a two-dimensional mesh
- Each internal node of the mesh has four connections, one to each of its horizontal and vertical neighbors.
- Nodes on the boundaries and corners of the mesh have fewer neighbors and hence fewer connections.
- To reduce latency for communication between nodes that would otherwise be far apart in the mesh, wraparound connections may be introduced between nodes at opposite boundaries of the mesh.
- A network with such connections is called a **torus**.
- All nodes in a torus have four connections. Average latency is reduced, but the implementation complexity for routing requests and responses through a torus is somewhat higher than in the case of a simple mesh

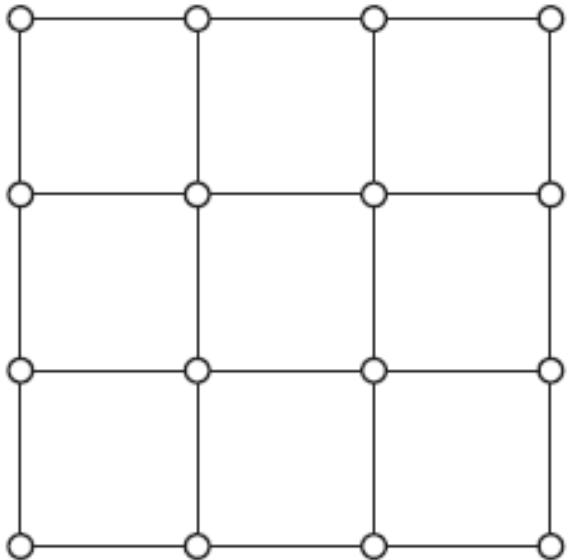


Figure 12.6 A two-dimensional mesh network.

Cache Coherence

- Possibility that copies of shared data may reside in several caches.
- When any processor writes to a shared variable in its own cache, all other caches that contain a copy of that variable will then have the old, incorrect value.
- They must be informed of the change so that they can either update their copy to the new value or invalidate it.
- This is the issue of maintaining ***cache coherence***, which requires having a consistent view of shared data in multiple caches.

Write-Through Protocol

Two ways

- updating the values in other caches
- invalidating the copies in other caches.

Updating the values in other caches

When a processor writes a new value to block of data in its cache, the new value is also written into the memory module containing the block being modified.

Since copies of this block may exist in other caches, these copies must be updated to reflect the change caused by the Write operation. The simplest way

- broadcast the written data to the caches of all processors in the system.
- As each processor receives the broadcast data, it updates the contents of the affected cache block if this block is present in its cache

Invalidating the copies in other caches

- When a processor writes a new value into its cache, this value is also sent to the appropriate location in memory, and all copies in other caches are invalidated.
- broadcasting can be used to send the invalidation requests throughout the system

Write-Back protocol

- based on the concept of ownership of a block of data in the memory.
- Initially, the memory is the owner of all blocks, and the memory retains ownership of any block that is read by a processor to place a copy in its cache.
- If some processor wants to write to a block in its cache, it must first become the exclusive owner of this block. To do so, all copies in other caches must first be invalidated with a broadcast request.
- The new owner of the block may then modify the contents at will without having to take any other action

- When another processor wishes to read a block that has been modified, the request for the block must be forwarded to the current owner. The data are then sent to the requesting processor by the current owner.
- The data are also sent to the appropriate memory module, which reacquires ownership and updates the contents of the block in the memory.
- The cache of the processor that was the previous owner retains a copy of the block. Hence, the block is now shared with copies in two caches and the memory.
- Subsequent requests from other processors to read the same block are serviced by the memory module containing the block.

- When another processor wishes to write to a block that has been modified, the current owner sends the data to the requesting processor.
- It also transfers ownership of the block to the requesting processor and invalidates its cached copy.
- Since the block is being modified by the new owner, the contents of the block in the memory are not updated.
- The next request for the same block is serviced by the new owner.

- The write-back protocol has the advantage of creating less traffic than the write-through protocol.
- Because a processor is likely to perform several writes to a cache block before this block is needed by another processor.
- With the write-back protocol, these writes are performed only in the cache, once ownership is acquired with an invalidation request.
- With the write-through protocol, each write must also be performed in the appropriate memory module and broadcast to other caches

Snoopy Caches

- In a single-bus system, all transactions between processors and memory modules occur via requests and responses on the bus.
- They are broadcast to all units connected to the bus.
- Suppose that each processor cache has a controller circuit that observes, or *snoops*, all transactions on the bus.
- scenarios for the write-back protocol

Scenario 1

- Consider a processor that has previously read a copy of a block from the memory into its cache.
- Before writing to this block for the first time, the processor must broadcast an *invalidation request* to all other caches, whose controllers accept the request and invalidate any copies of the same block.
- This action causes the requesting processor to become the new owner of the block.
- The processor may then write to the block and mark it as being modified.
- No further broadcasts are needed from the same processor to write to the modified block in its cache.

- if another processor broadcasts a *read request* on the bus for the same block, the memory must not respond because it is not the current owner of the block.
- The processor owning the requested block snoops the read request on the bus. Because it holds a modified copy of the requested block in its cache, it asserts a special signal on the bus to prevent the memory from responding.
- The owner then broadcasts a copy of the block on the bus, and marks its copy as clean (unmodified).
- The data response on the bus is accepted by the cache of the processor that issued the read request.

- The data response is also accepted by the memory to update its copy of the **block**. In this case, the memory reacquires ownership of the block, and the block is said to be in a shared state because copies of it are in the caches of two processors.
- Coherence is maintained because the two cached copies and the copy of the block in the memory contain the same data.
- Subsequent requests from any processor are serviced by the memory.

Scenario 2

- Two processors have copies of the same block in their respective caches, and both processors attempt to write to the same cache block at the same time.
- Since the block is in the shared state, the memory is the owner of the block.
- Both processors request the use of the bus to broadcast an invalidation message.
- One of the processors is granted the use of the bus first. That processor broadcasts its invalidation request and becomes the new owner of the block.
- Through snooping, the copy of the block in the cache of the other processor is invalidated.

- When the other processor is later granted the use of the bus, it broadcasts a *read-exclusive request*. This request combines a read request and an invalidation request for the same block.
- The controller for the first processor snoops the read-exclusive request, provides a data response on the bus, and invalidates the copy in its cache.
- Ownership of the block is therefore transferred to the second processor making the request.
- The memory is not updated because the block is being modified again.
- Since the requests from the two processors are handled sequentially, cache coherence is maintained at all times.
- This scheme is based on the ability of cache controllers to observe the activity on the bus and take appropriate actions. Such schemes are called *snoopy-cache* techniques

Performance

- the snooping function not interfere with the normal operation of a processor and its cache.
- Such interference occurs if the cache controller accesses the tags of the cache for every request that appears on the bus.
- In most cases, the cache would not contain a valid copy of the block that is relevant to a request.
- To eliminate unnecessary interference, each cache can be provided with a set of duplicate tags, which maintain the same status information about the blocks in the cache but can be accessed separately by the snooping circuitry.

Directory-Based Cache Coherence

- In systems using interconnection networks such as rings and meshes broadcasting every single request to the caches of all processors is inefficient
- A scalable, but more complex, solution to this problem uses *directories* in each memory module to indicate which nodes may have copies of a given block in the shared state.
- If a block is modified, the directory identifies the node that is the current owner.
- Each request from a processor must be sent first to the memory module containing the relevant block.
- The directory information for that block is used to determine the action that is taken.
- A read request is forwarded to the current owner if the block is modified. In the case of a write request for a block that is shared, individual invalidations are sent only to nodes that may have copies of the block in question.
- The cost and complexity of the directory-based approach for enforcing cache coherence limits its use to large systems.
- Small multiprocessors, including current multicore chips, use snooping.

Reference

- Carl Hamacher, Zvonko Vranesic, Safwat Zaky, Naraig Manjikian,
“Computer Organization And Embedded Systems”, Chapter 12, 6th
Edition, McGraw-Hill