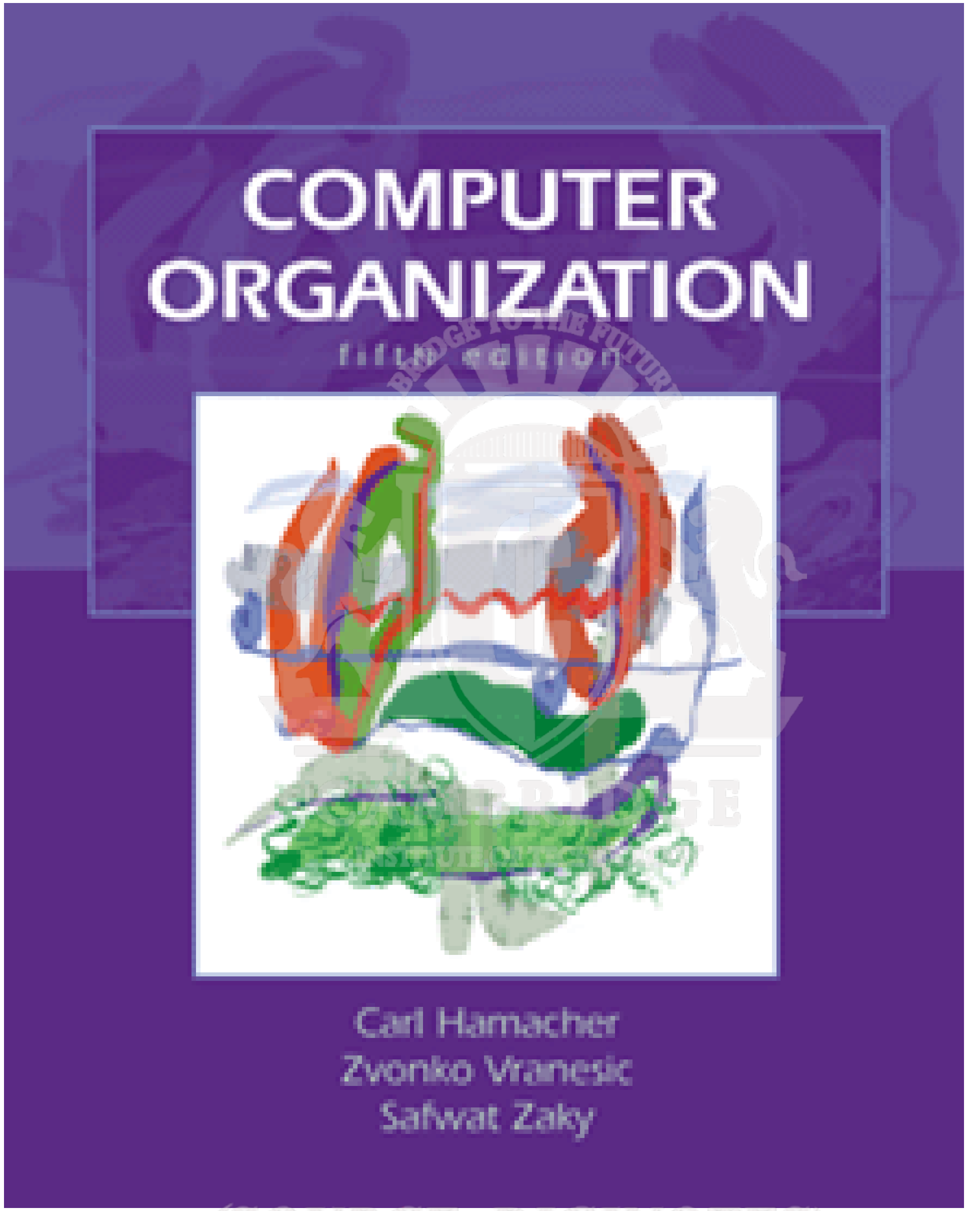
Subject code
15CS34



COMPUTER ORGANIZATION

fifth edition

Carl Hamacher
Zvonko Vranesic
Safwat Zaky

Sowmya K P
Asst. Professor
Dept of ISE
CITech

MODULE 1: Basic structure of computer & Machine instructions and programs

Courtesy: Text book: Carl Hamacher 5th Edition

Functional Units

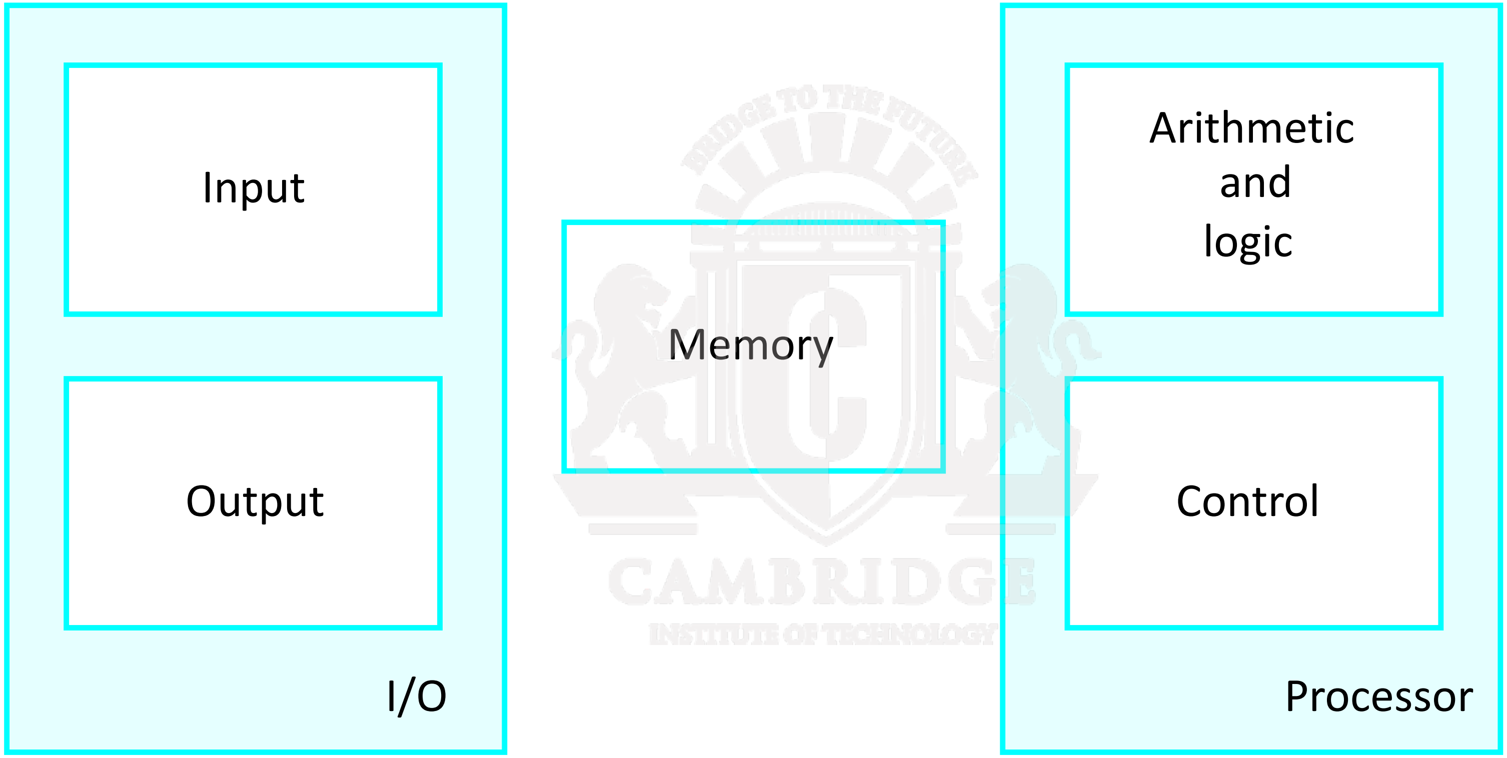


Figure 1.1. Basic functional units of a computer.

Basic operational concepts : A Typical Instruction

- Add LOCA, R0
- Add the operand at memory location LOCA to the operand in a register R0 in the processor.
- Place the sum into register R0.
- The original contents of LOCA are preserved.
- The original contents of R0 is overwritten.
- Instruction is fetched from the memory into the processor – the operand at LOCA is fetched and added to the contents of R0 – the resulting sum is stored in register R0.

Separate Memory Access and ALU Operation

- Load LOCA, R1
- Add R1, R0
- Whose contents will be overwritten?

Connection Between the Processor and the Memory

Registers

- **The instruction register (IR):- Holds the instructions that is currently being executed.**

- Its output is available for the control circuits which generates the timing signals that control the various processing elements in one execution of instruction.

- **The program counter PC:-**
- This is another specialized register that keeps track of execution of a program. It contains the memory address of the next instruction to be fetched and executed

- Besides IR and PC, there are n-general purpose registers R0 through Rn-1.

Registers

- The other two registers which facilitate communication with memory are: -
- **1. MAR – (Memory Address Register)**:- It holds the address of the location to be accessed.
- **2. MDR – (Memory Data Register):-** It contains the data to be written into or read out of the address location.

Typical Operating Steps

Programs reside in the memory through input devices

PC is set to point to the first instruction

The contents of PC are transferred to MAR

A Read signal is sent to the memory

The first instruction is read out and loaded into MDR

The contents of MDR are transferred to IR

Decode and execute the instruction

If the instruction involves an operation by the ALU, it is necessary to obtain the required operands.

Typical Operating Steps (Cont’)

An operand in the memory is fetched by sending its address to MAR & Initiating a read cycle.

When the operand has been read from the memory to the MDR, it is transferred from MDR to the ALU

After one or two such repeated cycles, the ALU can perform the desired operation.

If the result of this operation is to be stored in the memory, the result is sent to MDR.

Address of location where the result is stored is sent to MAR & a write cycle is initiated.

The contents of PC are incremented so that PC points to the next instruction that is to be executed.

Interrupt

- Normal execution of programs may be preempted if some device requires urgent servicing.
- An interrupt is a request signal from an I/O device for service by the processor.
- The processor provides the requested service by executing an appropriate interrupt service routine

Bus Structures

- There are many ways to connect different parts inside a computer together.
- A group of lines that serves as a connecting path for several devices is called a *bus*.
- Address/data/control
- Since the bus can be used for only one transfer at a time, only two units can actively use the bus at any given time.
- Bus control lines are used to arbitrate multiple requests for use of one bus.
- Single bus structure is
- Low cost
- Very flexible for attaching peripheral devices
- Multiple bus structure certainly increases, the performance but also increases the cost significantly.

Bus Structure

- Single-bus

Performance

- The most important measure of a computer is how quickly it can execute programs.
- Three factors affect performance:
- ***Hardware design***
- ***Instruction set***
- ***Compiler***
- The total time required to execute the program **is elapsed time** is a measure of the performance of the entire computer system. It is affected by the speed of the processor, the disk and the printer.
- The time needed to execute a instruction is called the processor time.
- Processor time to execute a program depends on the hardware involved in the execution of individual machine instructions

Performance

- ***The figure which includes the cache memory as part of the processor unit***

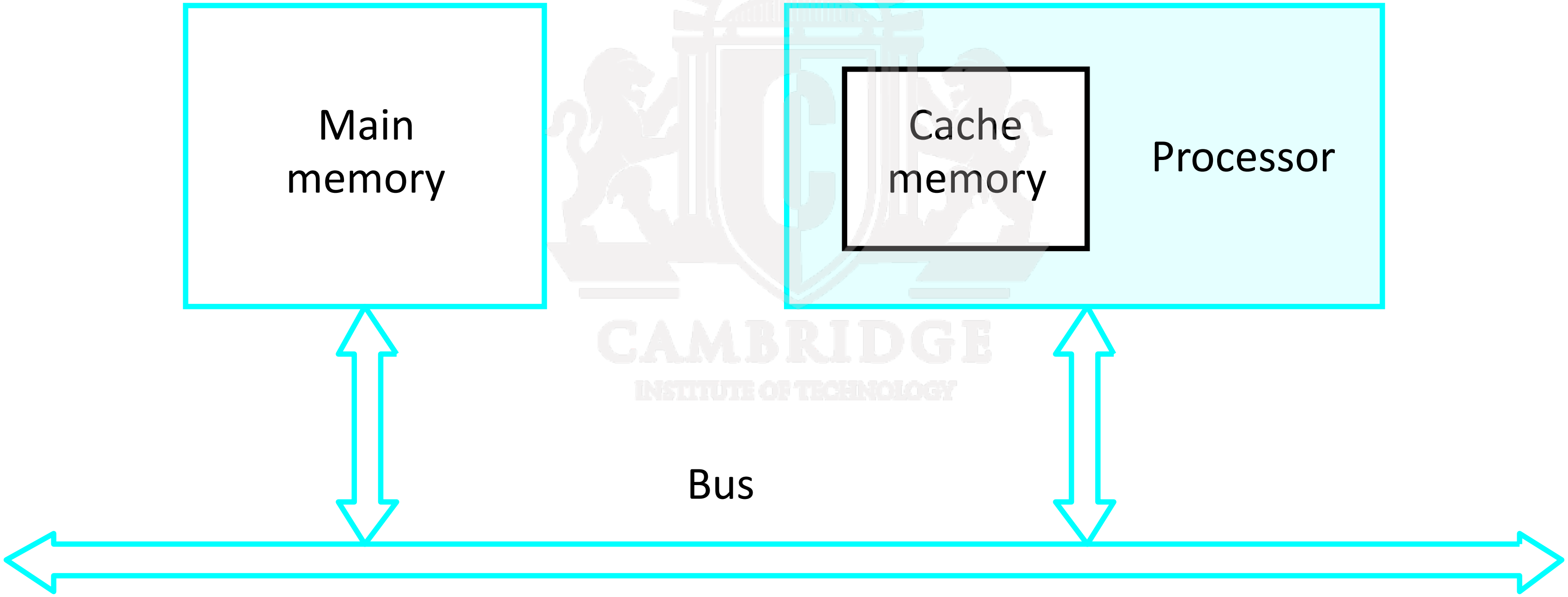


Figure 1.5. The processor cache.

Performance

- The processor and a relatively small cache memory can be fabricated on a single integrated circuit chip.
- Speed
- Cost
- Memory management

Performance

- Let us examine the flow of program instructions and data between the memory and the processor.
- At the start of execution, all **program instructions** and the **required data** are stored in the main memory.
- As the execution proceeds, instructions are fetched one by one over the bus into the processor, and **a copy is placed in the cache** later if the same instruction or data item is needed a second time, it is read directly from the cache

Processor Clock

- Processor circuits are controlled by a timing signal called ***clock***
- The clock defines the regular time intervals called **clock cycles**.
- To execute a machine instruction the processor divides the action to be performed into a sequence of basic steps that each step can be completed in one clock cycle.
- The length P of one clock cycle is an important parameter that affects the processor performance.
- Processor used in today's personal computer and work station have a clock rates that range from a few hundred million to over a billion cycles per second

Processor Clock

- The length P of one clock cycle is an important parameter that affects processor performance.
- Its inverse is the clock rate R=1/P.

- The term “cycles per second” is called *hertz*

- *The term “million” is denoted by the prefix Mega(M)*
- *and “billion” is denoted by the prefix Giga(G)*

Basic Performance Equation

- T – processor time required to execute a program that has been prepared in high-level language

- N – number of actual machine language instructions needed to complete the execution (note: loop)

- S – average number of basic steps needed to execute one machine instruction. Each step completes in one clock cycle

- R – clock rate

- Note: these are not independent to each other

$$T = \frac{N \times S}{R}$$

How to improve T?

Clock Rate

- These are two possibilities for increasing the clock rate 'R'.

  1. Improving the IC technology makes logical circuit faster, which reduces the time of execution of basic steps. This allows the clock period P, to be reduced and the clock rate R to be increased.

  2. Reducing the amount of processing done in one basic step also makes it possible to reduce the clock period P. however if the actions that have to be performed by an instructions remain the same, the number of basic steps needed may increase.

Performance Measurement

- The performance measure is the time taken by the computer to execute a given bench mark. Initially some attempts were made to create artificial programs that could be used as bench mark programs

- A non profit organization called SPEC- system performance evaluation corporation selects and publishes bench marks
- The program selected range from game playing, compiler, and data base applications to numerically intensive programs in astrophysics and quantum chemistry. In each case, the program is compiled under test, and the running time on a real computer is measured.
- The same program is also compiled and run on one computer selected as reference.
- The 'SPEC' rating is computed as follows.

Performance Measurement

$$SPEC\ rating = \frac{\text{Running time on the reference computer}}{\text{Running time on the computer under test}}$$

$$SPEC\ rating = (\prod_{i=1}^{n} SPEC_i)^{\frac{1}{n}}$$

If the SPEC rating = 50 Means that the computer under test is 50 times as fast as the ultra sparc 10.

This is repeated for all the programs in the SPEC suit, and the geometric mean of the result is computed.

Machine instructions and programs : Objectives

- Machine instructions and program execution, including branching and subroutine call and return operations.
- Number representation and addition/subtraction in the 2’s-complement system.
- Addressing methods for accessing register and memory operands.
- Assembly language for representing machine instructions, data, and programs.
- Program-controlled Input/Output operations.

Memory Location, Addresses, and Operation

- Memory consists of many millions of storage cells, each of which can store 1 bit.
- Data is usually accessed in *n*-bit groups. *n* is called word length.

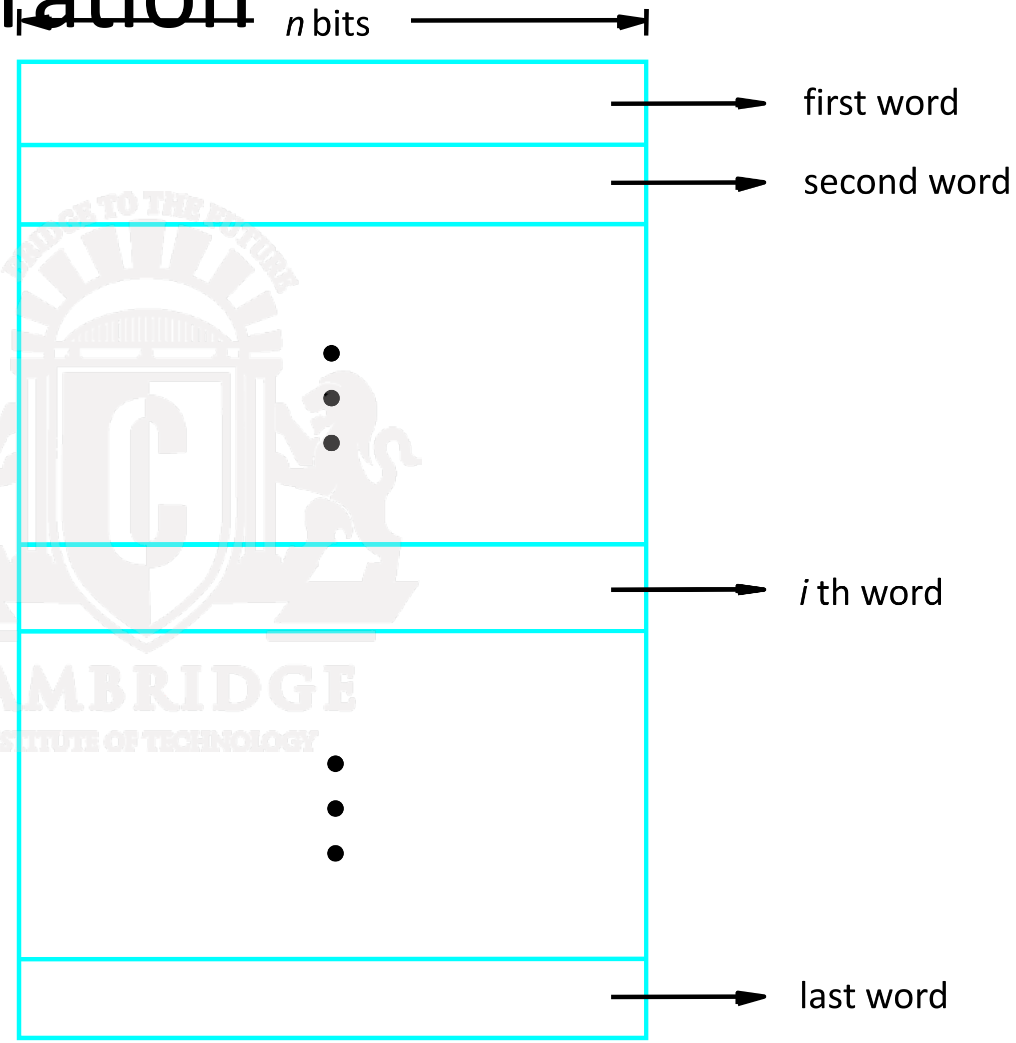
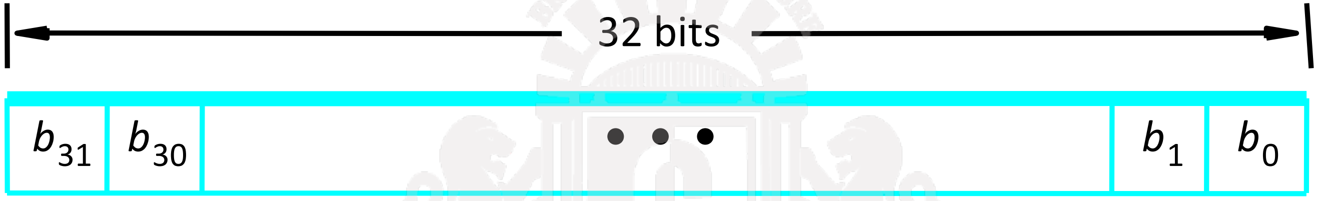


Figure 2.5. Memory words.

Memory Location, Addresses, and Operation

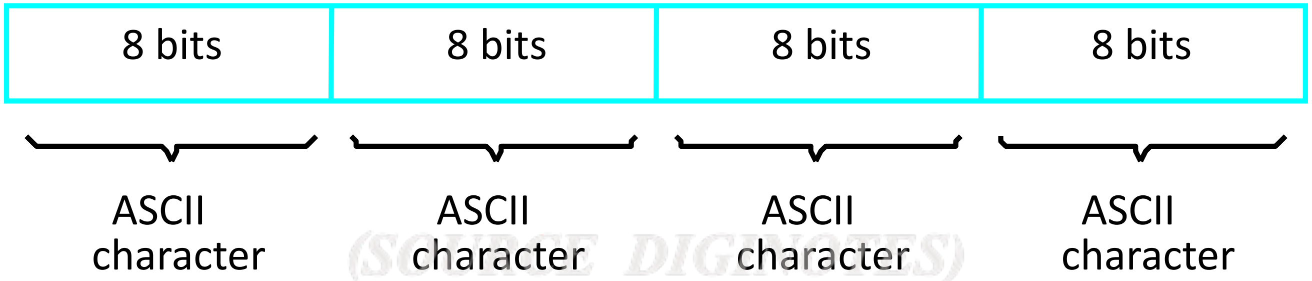
- 32-bit word length example



Sign bit: $b_{31} = 0$ for positive numbers

$b_{31} = 1$ for negative numbers

(a) A signed integer



(b) Four characters

Memory Location, Addresses, and Operation

- To retrieve information from memory, either for one word or one byte (8-bit), addresses for each location are needed.
- A *k*-bit address memory has 2^k memory locations, namely $0 – 2^k-1$, called memory space.
- 24-bit memory: 2^{24} = 16,777,216 = 16M (1M=2^{20})
- 32-bit memory: 2^{32} = 4G (1G=2^{30})
- 1K(kilo)=2^{10}
- 1T(tera)=2^{40}

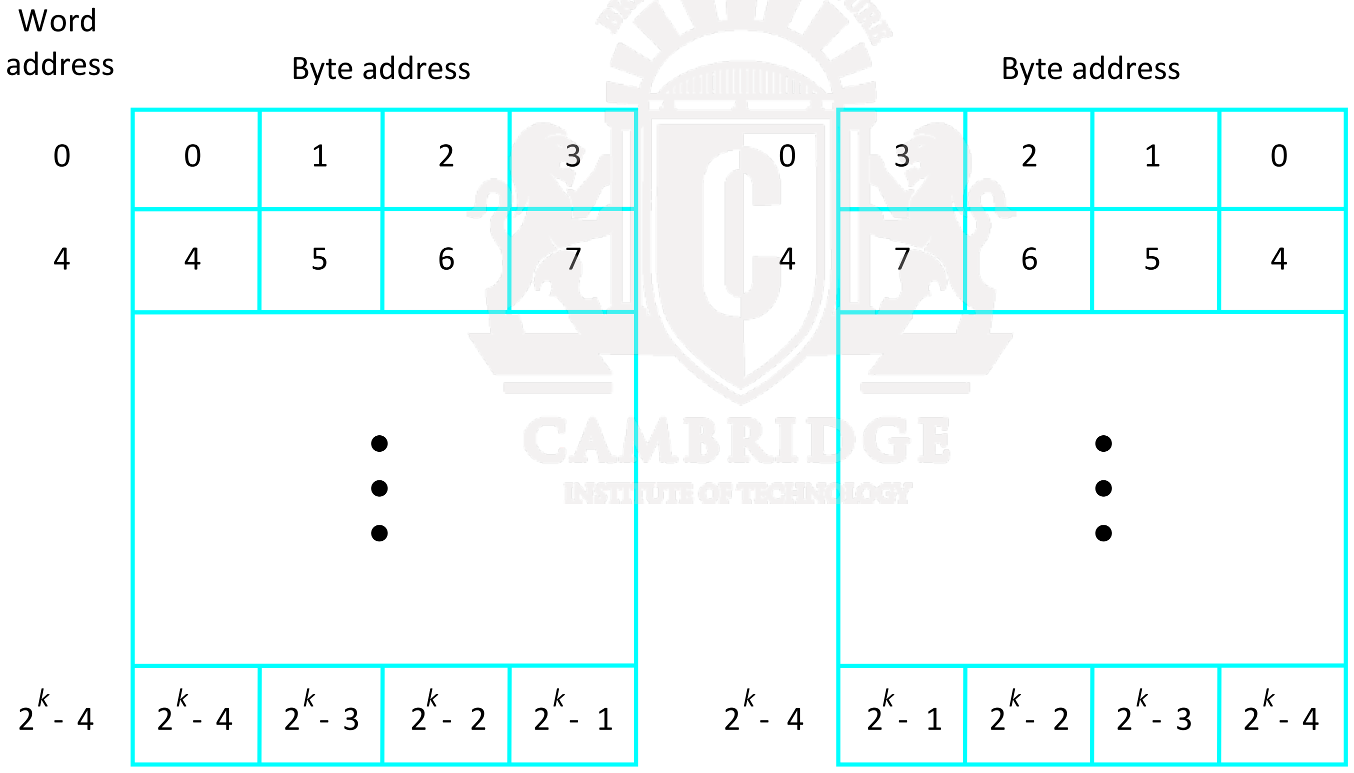
Memory Location, Addresses, and Operation

- It is impractical to assign distinct addresses to individual bit locations in the memory.
- The most practical assignment is to have successive addresses refer to successive byte locations in the memory – byte-addressable memory.
- Byte locations have addresses 0, 1, 2, … If word length is 32 bits, they successive words are located at addresses 0, 4, 8,…

Big-Endian and Little-Endian Assignments

Big-Endian: lower byte addresses are used for the most significant bytes of the word

Little-Endian: opposite ordering. lower byte addresses are used for the less significant bytes of the word



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			
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

Figure 2.7. Byte and word addressing.

Memory Location, Addresses, and Operation

- Address ordering of bytes
- Word alignment
  - – Words are said to be aligned in memory if they begin at a byte addr. that is a multiple of the num of bytes in a word.
    - 16-bit word: word addresses: 0, 2, 4,….
    - 32-bit word: word addresses: 0, 4, 8,….
    - 64-bit word: word addresses: 0, 8,16,….
- Access numbers, characters, and character strings

Memory Operation

- Load (or Read or Fetch)
  - ➢ Copy the content. The memory content doesn’t change.
  - ➢ Address – Load
  - ➢ Registers can be used
- Store (or Write)
  - ➢ Overwrite the content in memory
  - ➢ Address and Data – Store
  - ➢ Registers can be used

“Must-Perform” Operations

- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

Register Transfer Notation

- Identify a location by a symbolic name standing for its hardware binary address (LOC, R0,…)
- Contents of a location are denoted by placing square brackets around the name of the location (R1←[LOC], R3 ←[R1]+[R2])
- Register Transfer Notation (RTN)

Assembly Language Notation

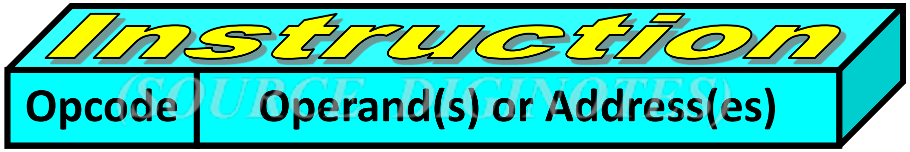
- Represent machine instructions and programs.
- Move LOC, R1 = R1←[LOC]
- Add R1, R2, R3 = R3 ←[R1]+[R2]

CPU Organization

- Single Accumulator
  - Result usually goes to the Accumulator
  - Accumulator has to be saved to memory quite often
- General Register
  - Registers hold operands thus reduce memory traffic
  - Register bookkeeping
- Stack
  - Operands and result are always in the stack

Instruction Formats

- Three-Address Instructions
  - ADD R1, R2, R3 R1 ← R2 + R3
- Two-Address Instructions
  - ADD R1, R2 R1 ← R1 + R2
- One-Address Instructions
  - ADD M AC ← AC + M[AR]
- Zero-Address Instructions
  - ADD TOS ← TOS + (TOS – 1)
- RISC Instructions
  - Lots of registers. Memory is restricted to Load & Store



Instruction Formats

Example: Evaluate (A+B) ∗ (C+D)

- Three-Address
  1. ADD R1, A, B ; R1 ← M[A] + M[B]
  2. ADD R2, C, D ; R2 ← M[C] + M[D]
  3. MUL X, R1, R2 ; M[X] ← R1 ∗ R2

Instruction Formats

Example: Evaluate (A+B) ∗ (C+D)

- Two-Address

1.	MOV	R1, A	; R1 ← M[A]
2.	ADD	R1, B	; R1 ← R1 + M[B]
3.	MOV	R2, C	; R2 ← M[C]
4.	ADD	R2, D	; R2 ← R2 + M[D]
5.	MUL	R1, R2	; R1 ← R1 ∗ R2
6.	MOV	X, R1	; M[X] ← R1

Instruction Formats

Example: Evaluate (A+B) ∗ (C+D)

- One-Address

```
1. LOAD   A        ; AC ← M[A]
2. ADD    B        ; AC ← AC + M[B]
3. STORE  T        ; M[T] ← AC
4. LOAD   C        ; AC ← M[C]
5. ADD    D        ; AC ← AC + M[D]
6. MUL    T        ; AC ← AC ∗ M[T]
7. STORE  X        ; M[X] ← AC
```

Instruction Formats

Example: Evaluate (A+B) ∗ (C+D)

- Zero-Address

1. PUSH A ; TOS ← A
2. PUSH B ; TOS ← B
3. ADD ; TOS ← (A + B)
4. PUSH C ; TOS ← C
5. PUSH D ; TOS ← D
6. ADD ; TOS ← (C + D)
7. MUL ; TOS ← (C+D)∗(A+B)
8. POP X ; M[X] ← TOS

Instruction Formats

Example: Evaluate (A+B) ∗ (C+D)

- RISC

```
1.  LOAD    R1, A                 ; R1 ← M[A]
2.  LOAD    R2, B                 ; R2 ← M[B]
3.  LOAD    R3, C                 ; R3 ← M[C]
4.  LOAD    R4, D                 ; R4 ← M[D]
5.  ADD     R1, R1, R2            ; R1 ← R1 + R2
6.  ADD     R3, R3, R4            ; R3 ← R3 + R4
7.  MUL     R1, R1, R3            ; R1 ← R1 ∗ R3
8.  STORE   X, R1                 ; M[X] ← R1
```

Using Registers

- Registers are faster
- Shorter instructions
  - The number of registers is smaller (e.g. 32 registers need 5 bits)
- Potential speedup
- Minimize the frequency with which data is moved back and forth between the memory and processor registers.

Instruction Execution and Straight-Line Sequencing

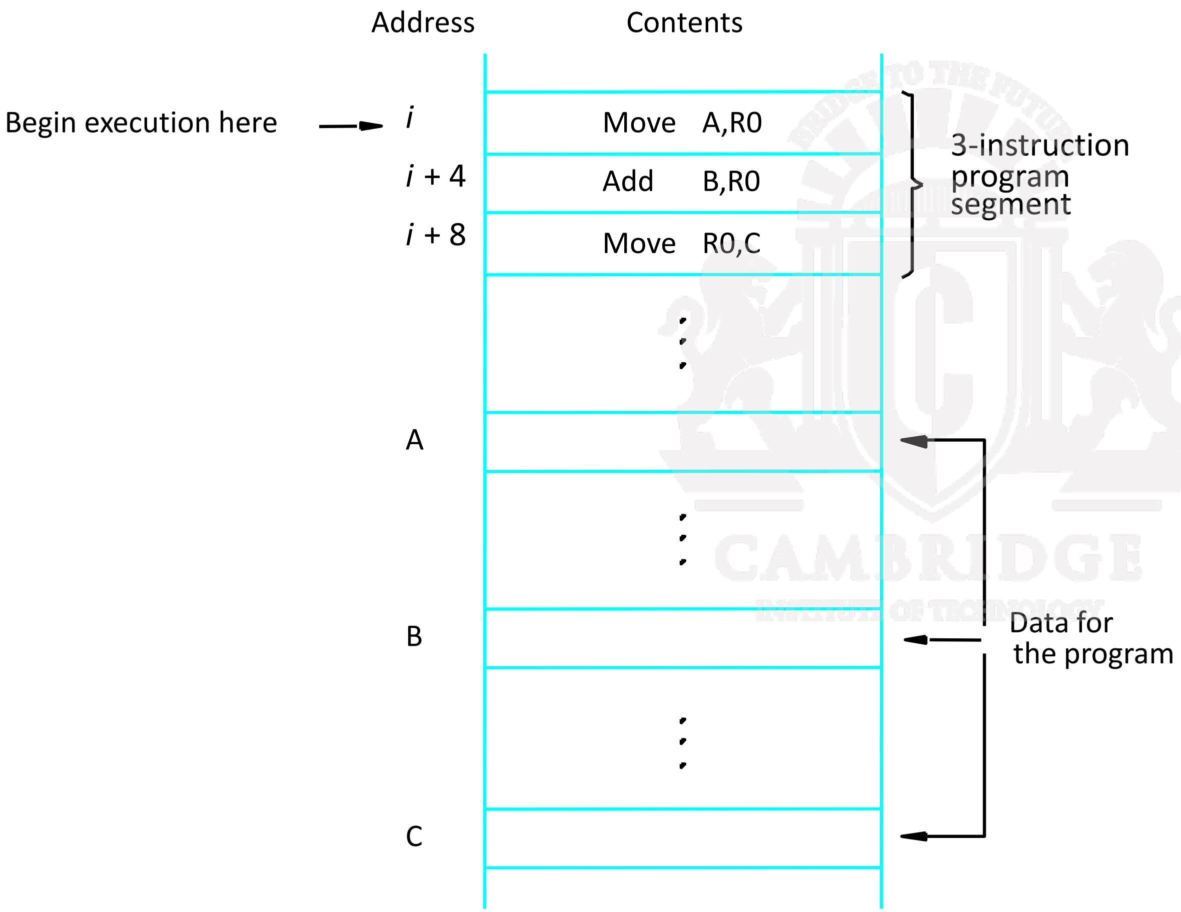


Figure 2.8. A program for C ← [A] + [B].

Assumptions:

- One memory operand per instruction
- 32-bit word length
- Memory is byte addressable
- Full memory address can be directly specified in a single-word instruction

Two-phase procedure

-Instruction fetch

-Instruction execute

Page 43

Branching

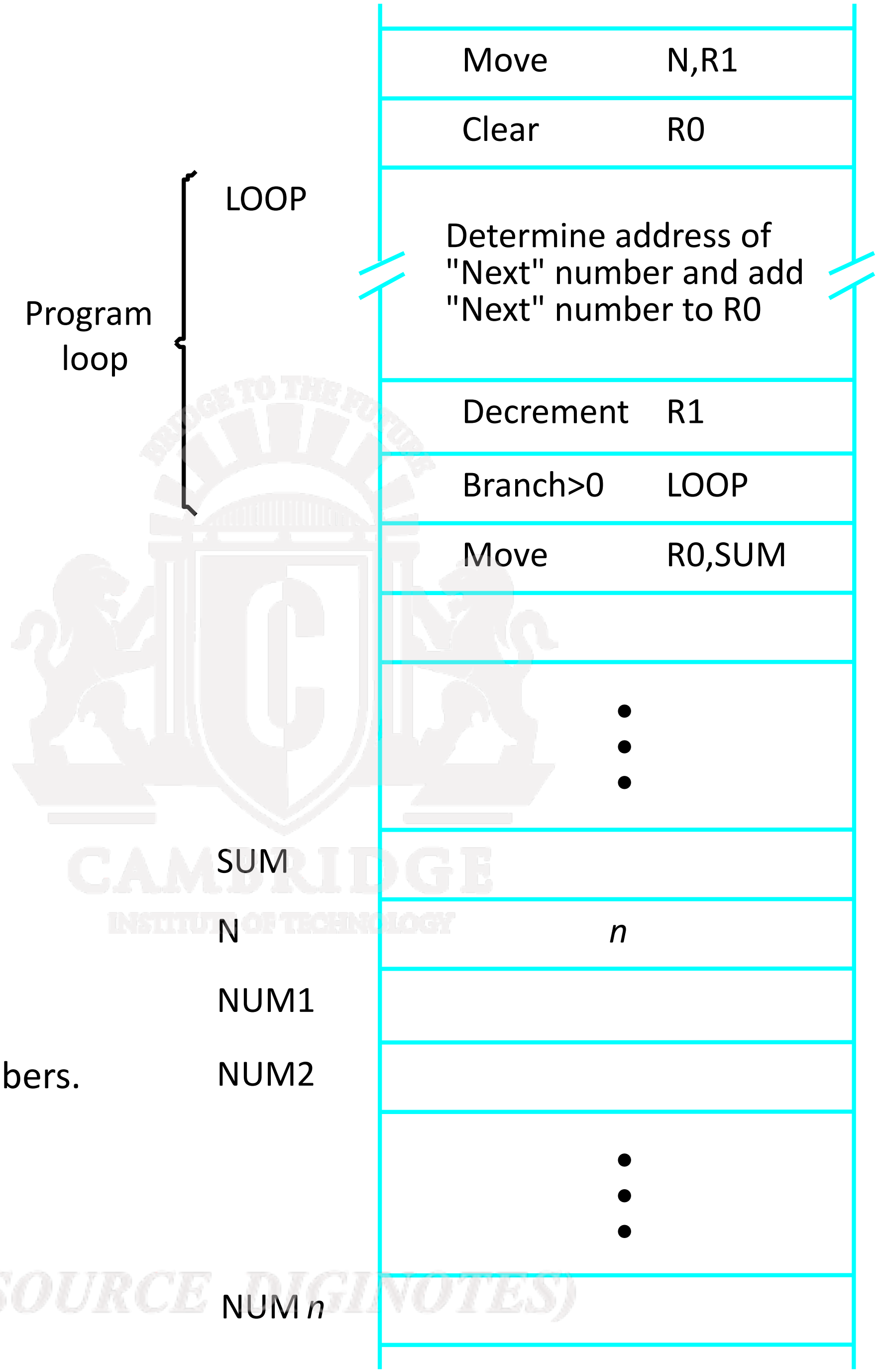
i	Move	NUM1,R0
$i + 4$	Add	NUM2,R0
$i + 8$	Add	NUM3,R0
	⋮	
$i + 4n - 4$	Add	NUMn,R0
$i + 4n$	Move	R0,SUM
	⋮	
SUM		
NUM1		
NUM2		
	⋮	
NUMn		

Figure 2.9. A straight-line program for adding n numbers.

Branching

Branch target

Conditional branch



	Move	N,R1
	Clear	R0
LOOP	Determine address of "Next" number and add "Next" number to R0	
	Decrement	R1
	Branch>0	LOOP
	Move	R0,SUM
	⋮	
SUM		
N	*n*	
NUM1		
NUM2		
	⋮	
NUM *n*		

Program loop

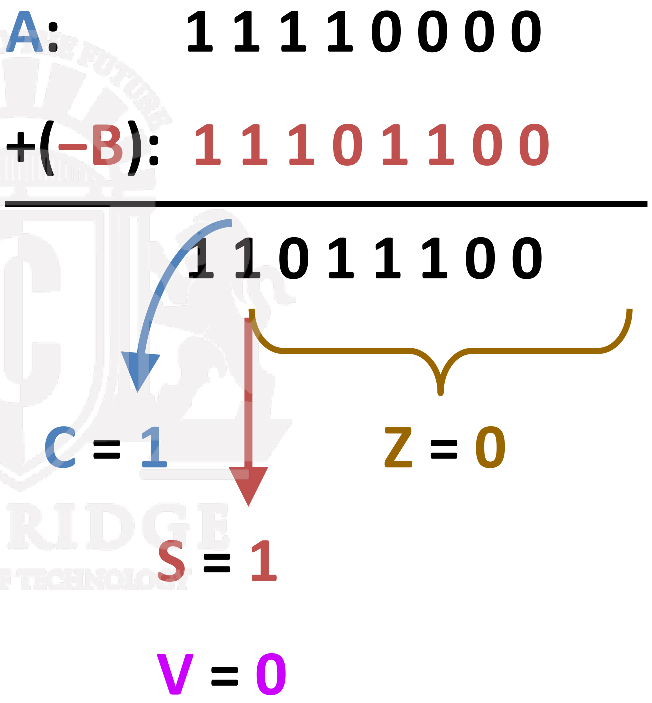
Figure 2.10. Using a loop to add *n* numbers.

Condition Codes

- Condition code flags
- Condition code register / status register
- N (negative)
- Z (zero)
- V (overflow)
- C (carry)
- Different instructions affect different flags

Conditional Branch Instructions

- Example:
  - A: 1 1 1 1 0 0 0 0
  - B: 0 0 0 1 0 1 0 0



A: 1 1 1 1 0 0 0 0

+(−B): 1 1 1 0 1 1 0 0

1 1 0 1 1 1 0 0

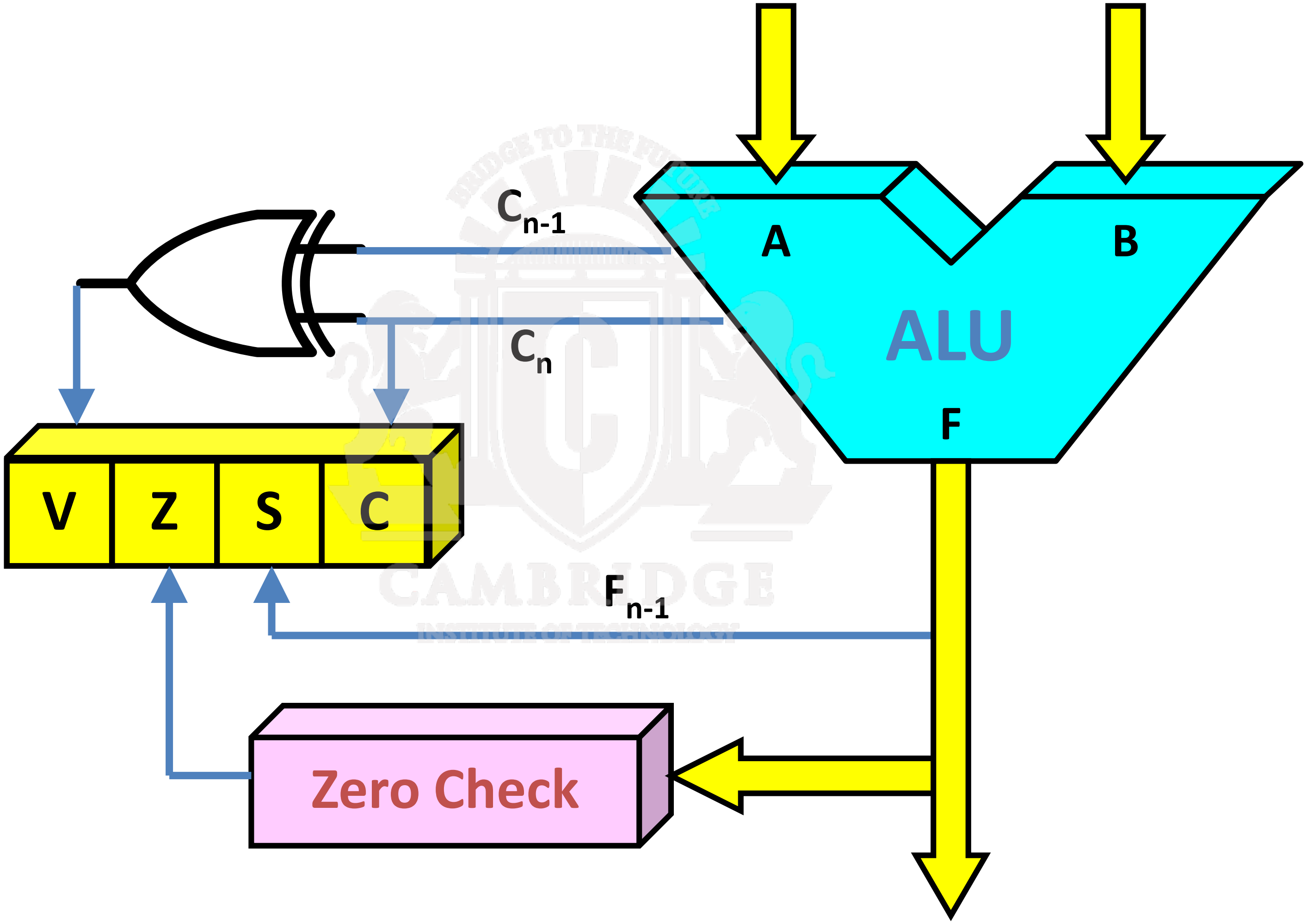
C = 1

Z = 0

S = 1

V = 0

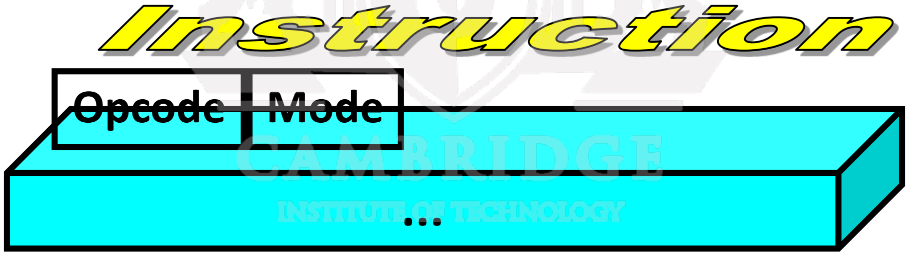
Status Bits



(SOURCE DIGINOTES)

Addressing Modes

Addressing modes:-The different ways in which the location of an operand is specified in an instruction are referred to as addressing modes.



CHAPTER 2: Machine instructions and programs

Courtesy: Text book: Carl Hamacher 5th Edition

Addressing Modes

- **Register mode** :-The operand is the content of a processor register; the name(address) is given in the processor.

  Ex:- Move R1,R2

- **Absolute mode**:-The operand is in a memory location; the address of this location is given explicitly in the instruction.

  Ex:- Move LOC,R2

Immediate mode:- The operand is given explicitly in the instruction(clearly immediate mode is to specify the value of a source operand)

Ex:- Move #200,R0

For ex:- A=B+6;

Addressing Modes

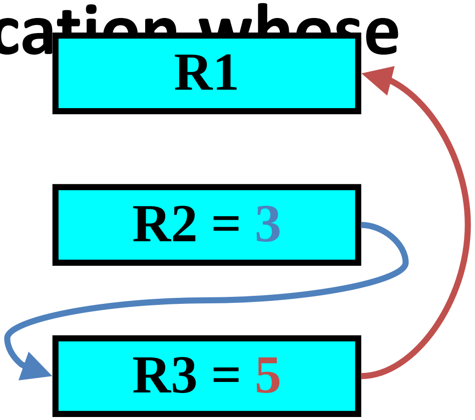
- **Indirect Address:**

  The effective address of the operand is the content of a register or memory whose address appears in the instruction

Ex:- 1) Add (R1) R0 2) Add (AR),R0

Addressing Modes

- **Indirect mode**
  - **The effective address of the operand is the contents of a register or memory location whose address appears in the instructions**

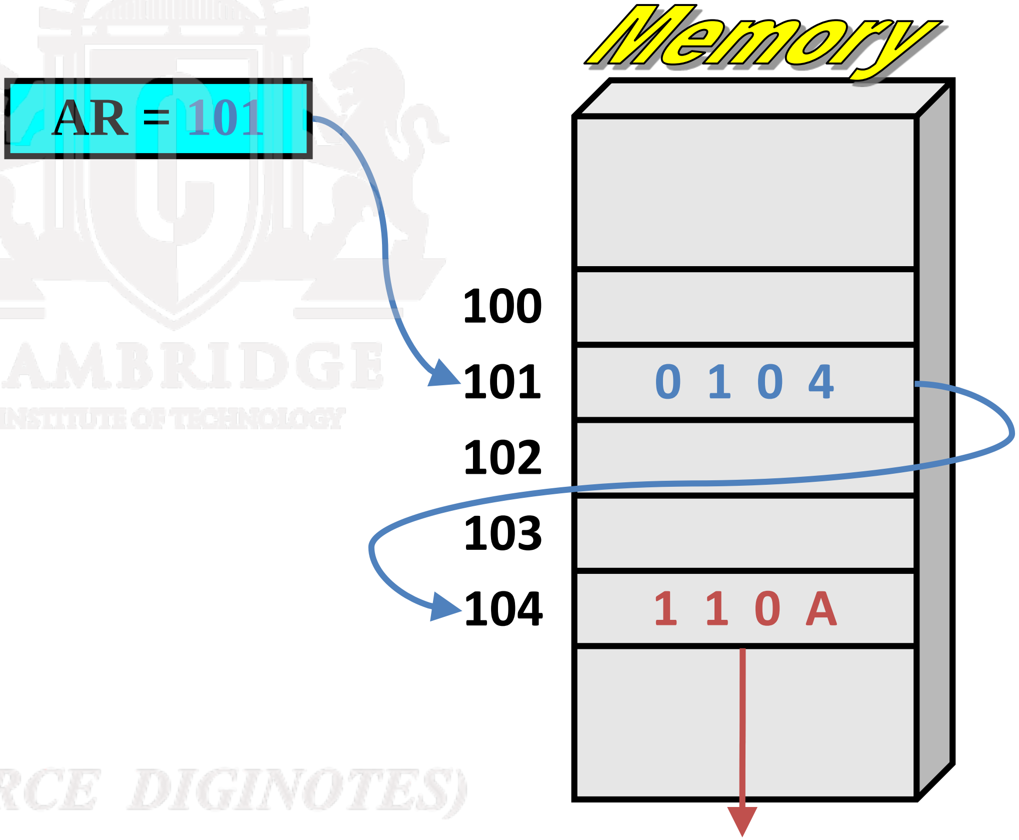


Ex:- Add R1, (R2);

Add (AR),R0;

Addressing Modes

- Indirect Address
  - – Indicate the memory location that holds the address of the memory location that holds the data



(SOURCE DIGINOTES)

Addressing Modes

- **Index mode:-** The effective address of the operand is generated by adding a constant value to the content of a register

Ex:- 1) Add 20(R1),R2 2) Add 1000(R1),R2

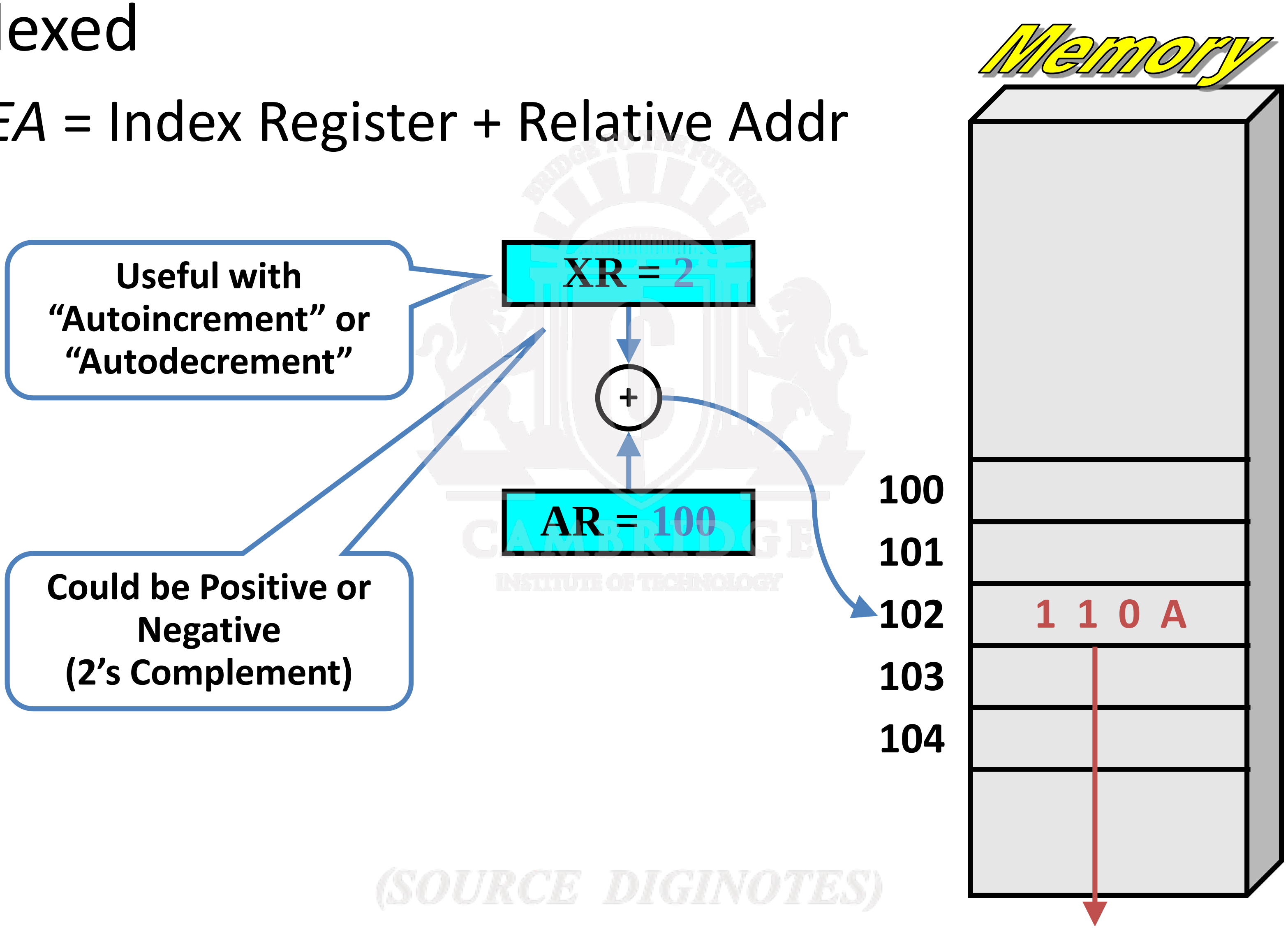
Index register

$X(R_i)$: $EA = X + [R_i]$

- The constant X may be given either as an explicit number or as a symbolic name representing a numerical value.
- If X is shorter than a word, sign-extension is needed.

Addressing Modes

- Indexed
  - *EA* = Index Register + Relative Addr



Useful with "Autoincrement" or "Autodecrement"

Could be Positive or Negative (2's Complement)

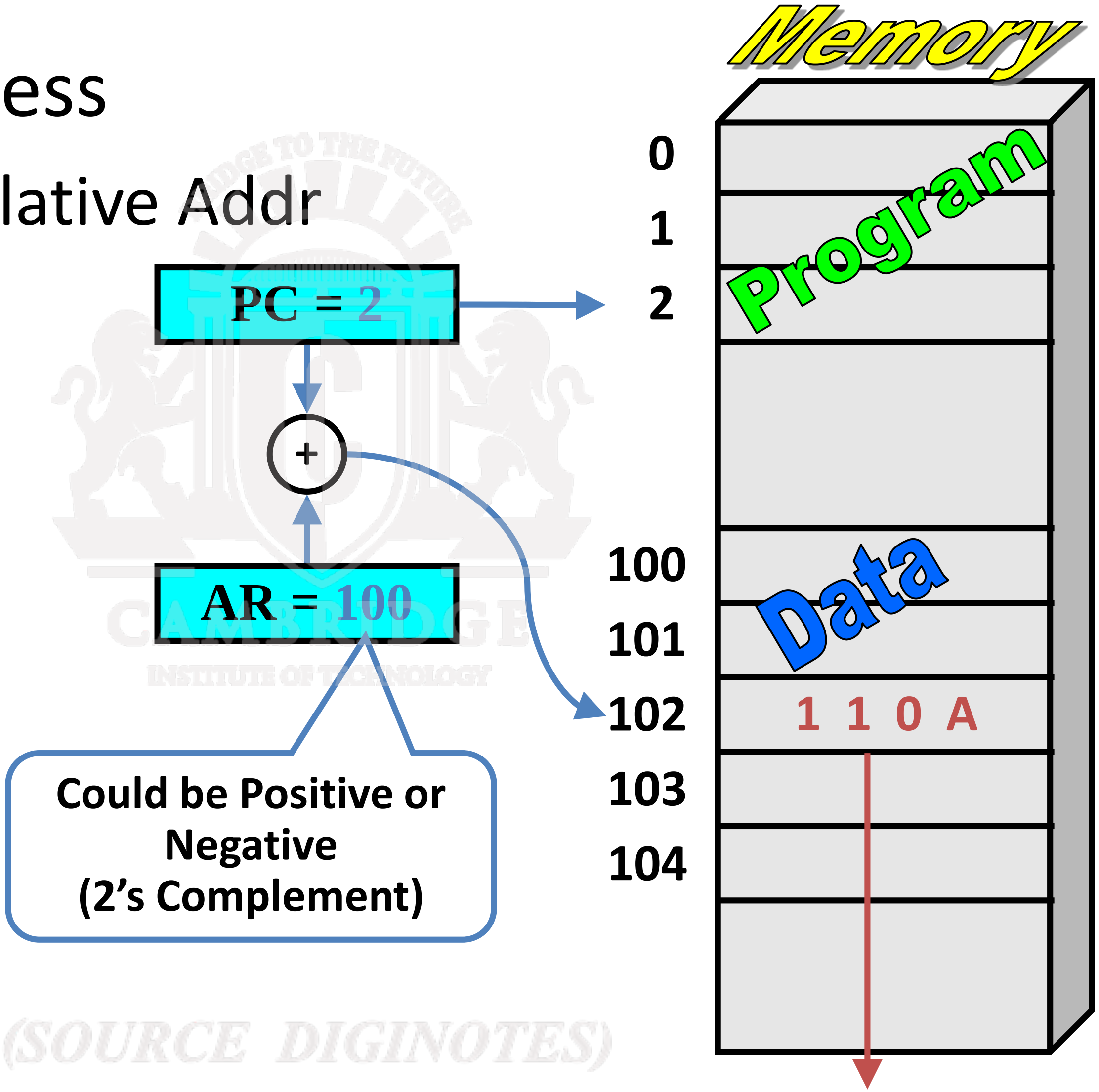
Addressing Modes

- **Relative mode** :-The effective address is determined by the Index mode using the program counter in place of the general purpose register R*i*.

- **Ex:- Add 20(PC),R0**

Addressing Modes

- Relative Address
  - *EA* = PC + Relative Addr



Additional Modes

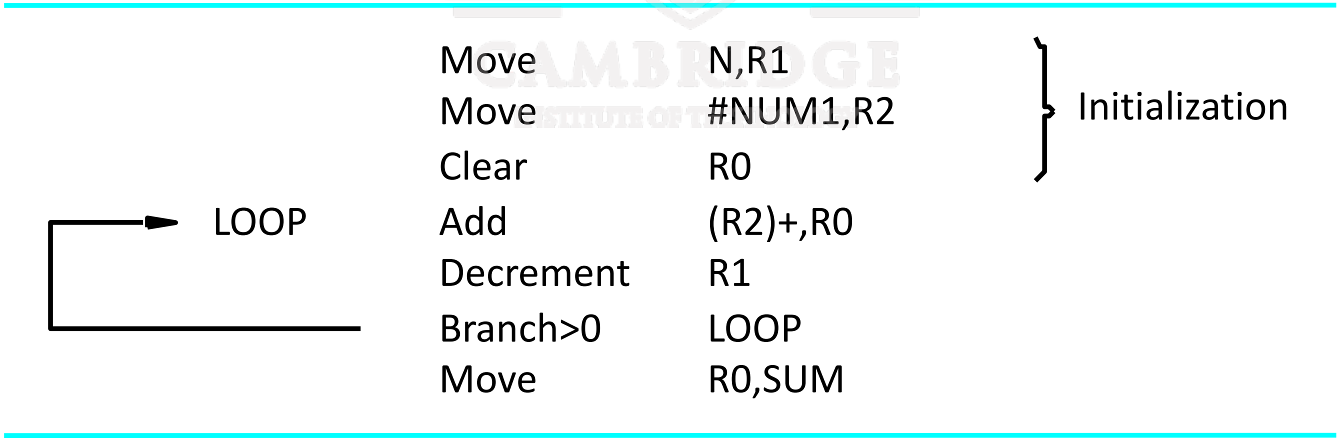
- **Auto increment mode** :-The effective address of the operand is the content of a register specified in the instruction determined. After accessing the operand, the contents of this registers are automatically incremented to point to the next item in a list.
- Ex:- (R*i*)+

- **Auto decrement mode** :-The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand.
- Ex:- -(R*i*)

Additional Modes

- Autoincrement mode – the effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list.

- Autodecrement mode: -(R_i) – decrement first



```
            Move        N,R1      }
            Move        #NUM1,R2  }  Initialization
            Clear       R0        }
LOOP        Add         (R2)+,R0
            Decrement   R1
            Branch>0    LOOP
            Move        R0,SUM
```

Figure 2.16. The Autoincrement addressing mode used in the program of Figure 2.12.

Addressing Modes

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (Direct)	LOC	EA = LOC

Indirect	(R*i*)
	(LOC)
		EA = [R*i*]
		EA = [LOC]

Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Base with index and offset	X(R*i*,R*j*)	EA = [R*i*] + [R*j*] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R*i*)+	EA = [R*i*] ; Increment R*i*
Autodecrement	−(R*i*)	Decrement R*i* ; EA = [R*i*]

Assembly Language

- Machine instructions are represented by patterns of 0's and 1's –awkward to deal
- So we use symbolic names to represent the patterns. so far we have used normal words, such as **Move, Add, and Branch,** for the instruction operation to represent the corresponding binary code patterns.
- When writing programs for a specific computer, such words are normally replaced by acronyms called ***mnemonics,*** such as **MOV,ADD,BR** .similarly we use the notation **R3** to refer to **register 3** and **LOC** to refer to a **memory location.**
- A complete set of names and rules for their use constitutes a programming language, generally referred to as an **assembly language**

Assembly Language

- The set of rules for using the mnemonics in the specification of complete instructions and programs is called the **syntax** of the language.

Assembly Language : Types of Instructions

- Data Transfer Instructions

Data value is not modified

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

Assembler DIrectives

- In addition to provide a mechanism for representing instructions in a program.
- The assembly language allows the programmer to specify other information needed to translate the source program into object program.
- Suppose that the name SUM is used to represent the value 200. This can be conveyed to the assembler through a statement as

SUM EQU 200

This statement simply informs the assembler that the SUM should be replaced by the value 200 where ever it appears in the program.

Such statements are **assembler directives**

Assembler Directives

	Address	Operation	Addressing
	Label		or Data operation
Ex:-	**SUM**	**EQU**	**200**
		ORIGIN	**204**
	N	**DATAWORD**	**100**
	NUM1	**RESERVE**	**400**

This **ORIGIN** directive tells the assembler where in the memory to place the data block that follows. In this case the location specified has the address 204.since this location is to be loaded with value 100.

- A **DATAWORD** directive is used to inform the assembler of this requirement .It states that the data value is to be placed in the memory word at address 204

Assembler Directives

- Any statement that results in instructions of data being placed in a memory location may be given a **memory address label.** This label is assigned a value equal to the address of that location.

- Because the **DATAWORD** statement is given the label N, the Name N is assigned the value 204.Whenever N is encountered in the rest of the program it will be replaced with this value.

- The **RESERVE** directive declares that a memory block of 400 bytes to be reserved for data and that the name NUM1 is to be associated with address 208

Assembler DIrectives

```
START     MOVE   N,R1
          MOVE  #NUM1,R2
          CLR R0
LOOP      ADD (R2),R0
           ADD #4,R2
           DEC R1
           BGTZ  LOOP
           MOVE R0,SUM
           RETURN
           END START
```

Assembler DIrectives

- The Last Statement in the source program is the assembler directive **END**, which tells that this is the end of source program text. The END directive includes the label **START**, which is the address of the location at which execution of the program is to begin.

- The **RETURN** assembler directive identifies the point at which execution of the program should be terminated.
- Most assembly languages require statements in a source program to be written in the form

Label Operation Operand(s) Comment;

These four fields are separated by an appropriate delimiter, typically one or more blank characters.

Number Notations

- Numbers can be specified as an operand in an instruction using different represent representations.

- **Ex:- ADD #93,R1 (Decimal).**
-
- **ADD #%01011101,R1 (Binary)**

- **ADD #$5D,R1 (Hexadecimal)**

Basic Input/Output Operations:Program-Controlled I/O

- Read in character input from a keyboard and produce character output on a display screen.

Rate of data transfer (keyboard, display, processor)

Difference in speed between processor and I/O device creates the need for mechanisms to synchronize the transfer of data.

A solution: on output, the processor sends the first character and then waits for a signal from the display that the character has been received. It then sends the second character. Input is sent from the keyboard in a similar way.

Program-Controlled I/O Example

- Registers
- Flags
- Device interface

Program-Controlled I/O Example

The keyboard and the display are separate device as shown in fig. the action of striking a key on the keyboard does not automatically cause the corresponding character to be displayed on the screen.

One block of instructions in the I/O program transfers the character into the processor, and another associated block of instructions causes the character to be displayed

Striking a key stores the corresponding character code in an 8-bit buffer register associated with the keyboard. Let us call this register **DATAIN,** as shown in fig

Program-Controlled I/O Example

To inform the processor that a valid character is in **DATAIN**, a status control flag, **SIN**, is set to 1. A program monitors **SIN**, and when **SIN** is set to 1, the processor reads the contents of **DATAIN**. When the character is transferred to the processor, **SIN** is automatically cleared to 0. If a second character is entered at the keyboard, **SIN** is again set to 1, and the processor repeat

An analogous process takes place when characters are transferred from the processor to the display. A buffer register, **DATAOUT**, and a status control flag, **SOUT**, are used for this transfer. When **SOUT** equals 1, the display is ready to receive a character.

Program-Controlled I/O Example

- Machine instructions that can check the state of the status flags and transfer data:

```
READWAIT  Branch to READWAIT if SIN = 0
          Input from DATAIN to R1

WRITEWAIT Branch to WRITEWAIT if SOUT = 0
          Output from R1 to DATAOUT
```

Program-Controlled I/O Example

- Memory-Mapped I/O – some memory address values are used to refer to peripheral device buffer registers. No special instructions are needed. Also use device status registers.

```
READWAIT  Testbit   #3, INSTATUS
          Branch=0  READWAIT
          MoveByte  DATAIN, R1
```

- **The write may be implemented as**

```
WRITEWAIT    Testbit   #3, OUTSTATUS
               Branch=0  WRITEWAIT
               MoveByte  R1,DATAOUT
```

Stacks

A stack is a small area in the memory of a computer used to store data elements. The main feature of the stack is that, the elements can be added or removed to one end only and the other is fixed.

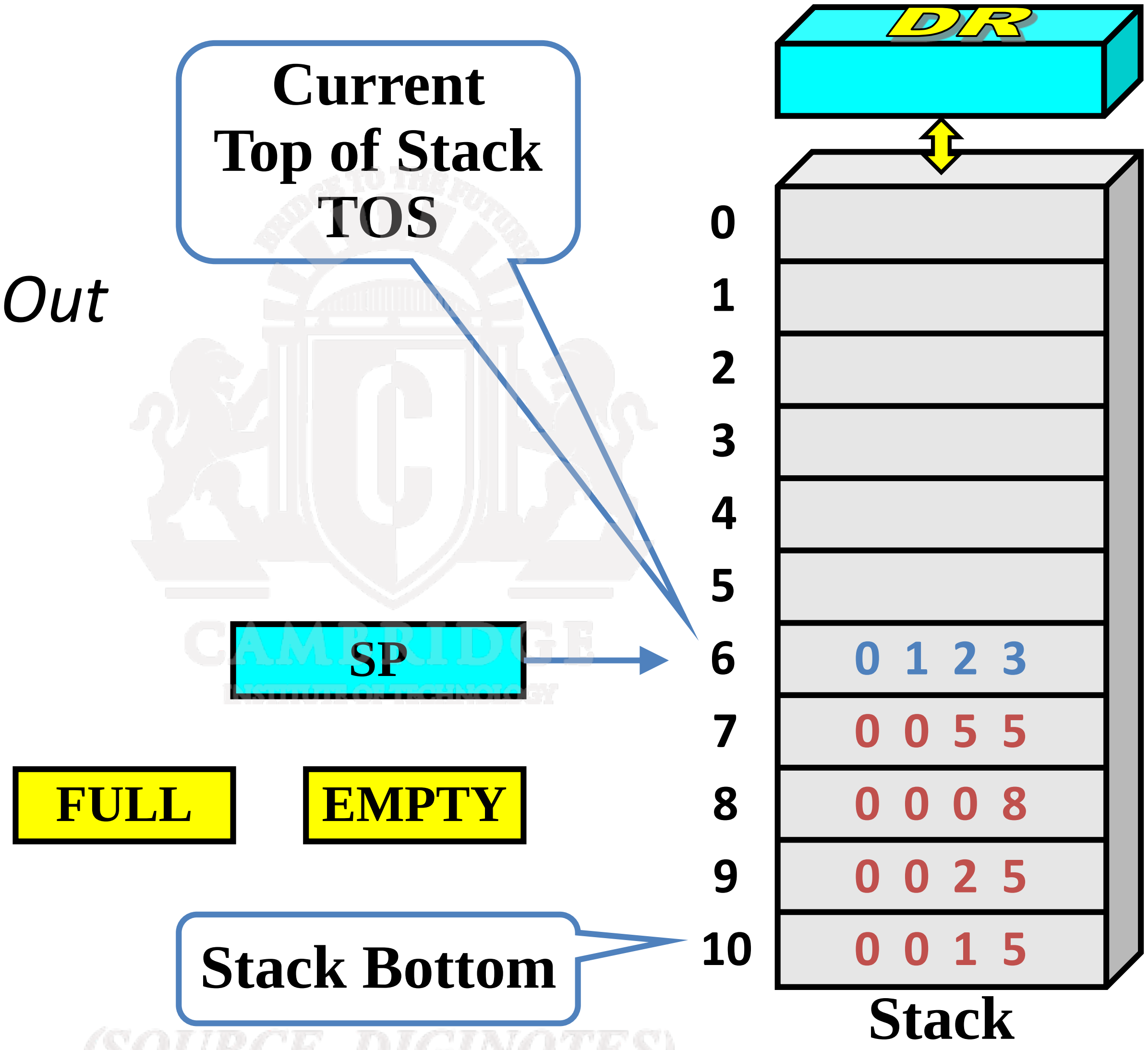
The open end is called top of the stack**(TOS),**and the fixed end is **bottom.** We use the term **Push** and **POP** to denote the operations on the stack.

When elements are pushed into the stack they are placed in successively **lower address locations**. Thus the stack **grows** in the direction of **decreasing** memory address

Stack : Stack Organization

- LIFO

  Last In First Out



DR

Current Top of Stack TOS

SP

FULL

EMPTY

Stack Bottom

0	
1	
2	
3	
4	
5	
6	0 1 2 3
7	0 0 5 5
8	0 0 0 8
9	0 0 2 5
10	0 0 1 5

Stack

Stack Organization

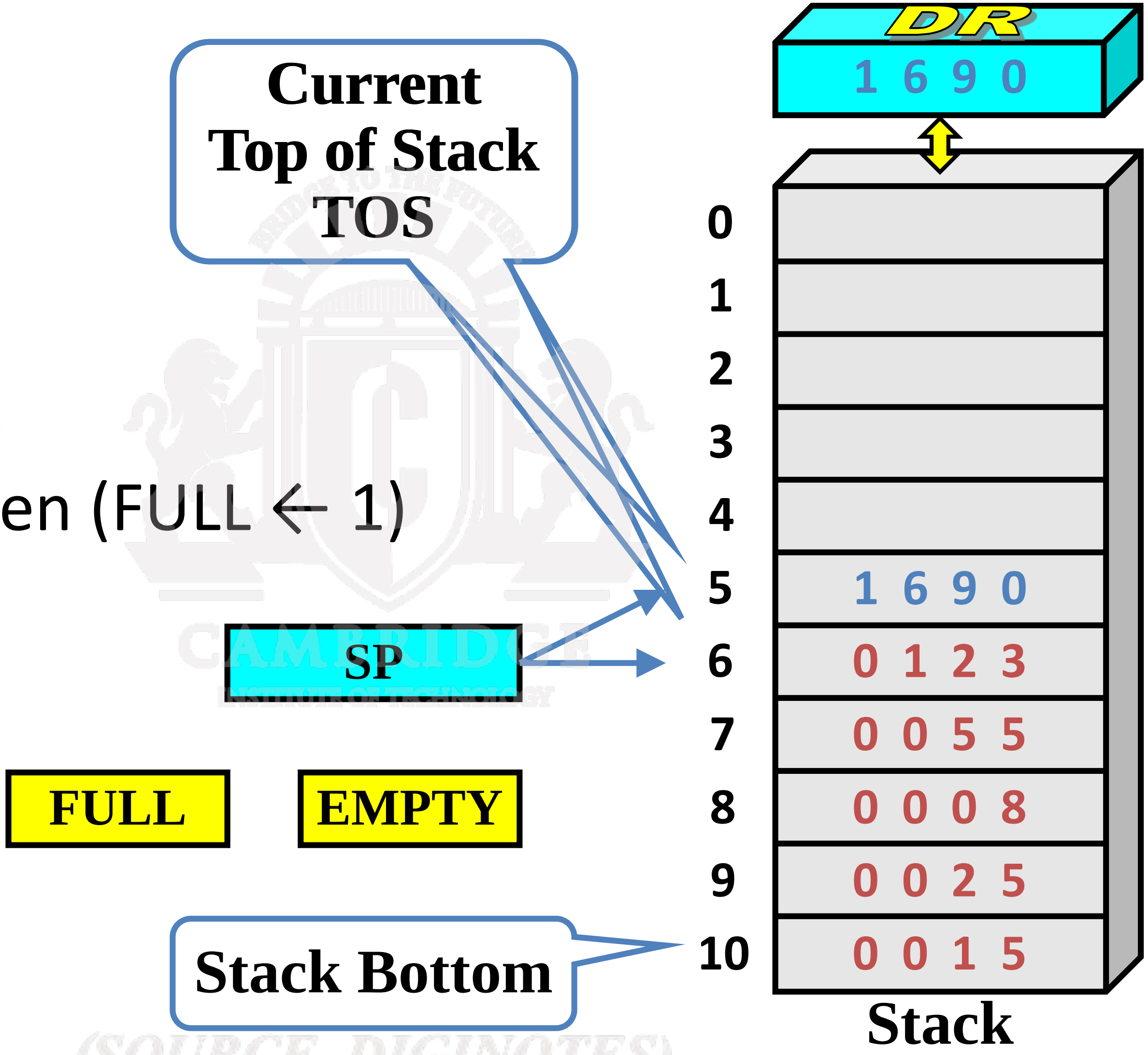
- PUSH

  SP ← SP – 1

  M[SP] ← DR

  If (SP = 0) then (FULL ← 1)

  EMPTY ← 0



Stack Organization

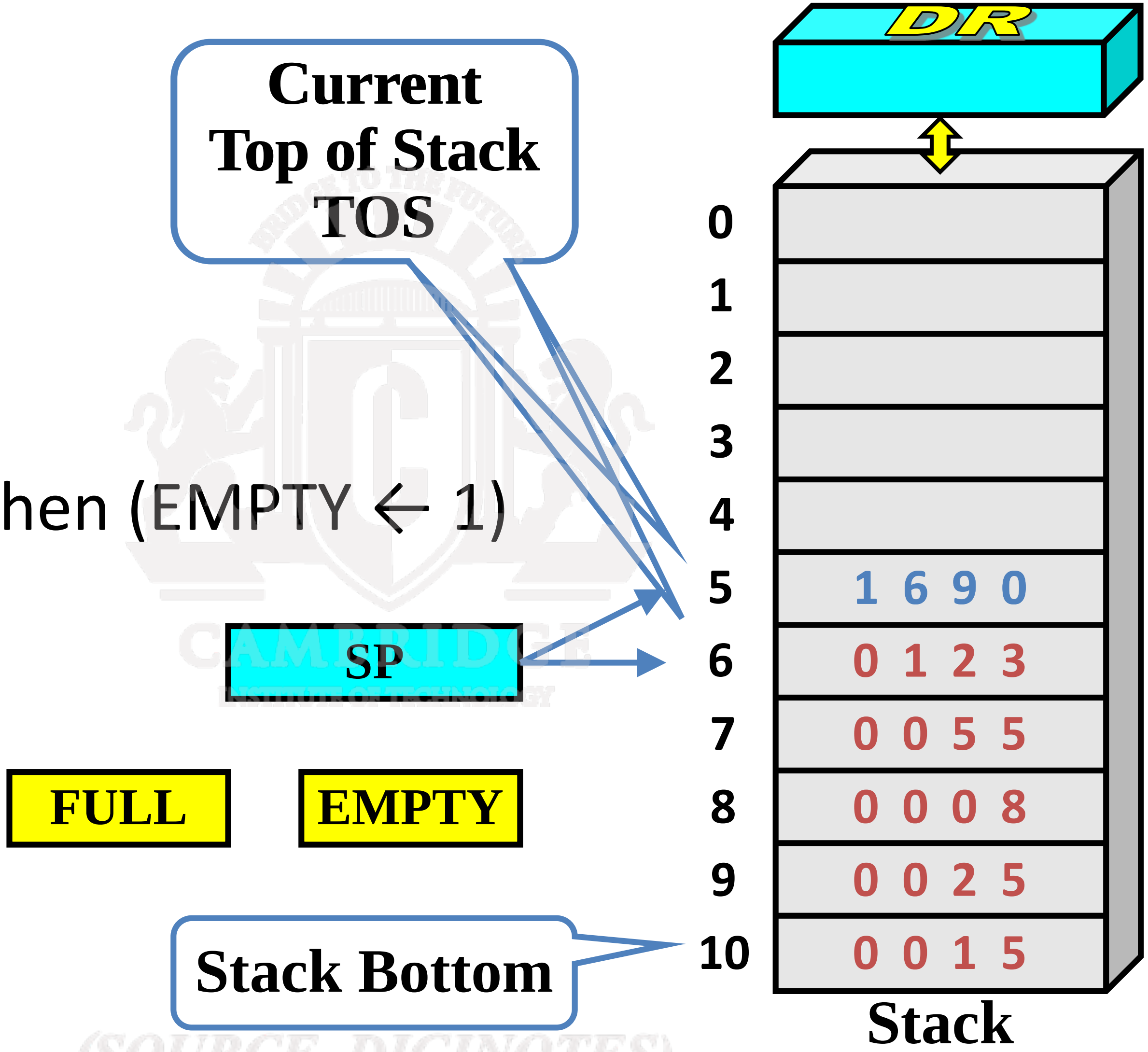
- POP

DR ← M[SP]

SP ← SP + 1

If (SP = 11) then (EMPTY ← 1)

FULL ← 0



Stack Organization

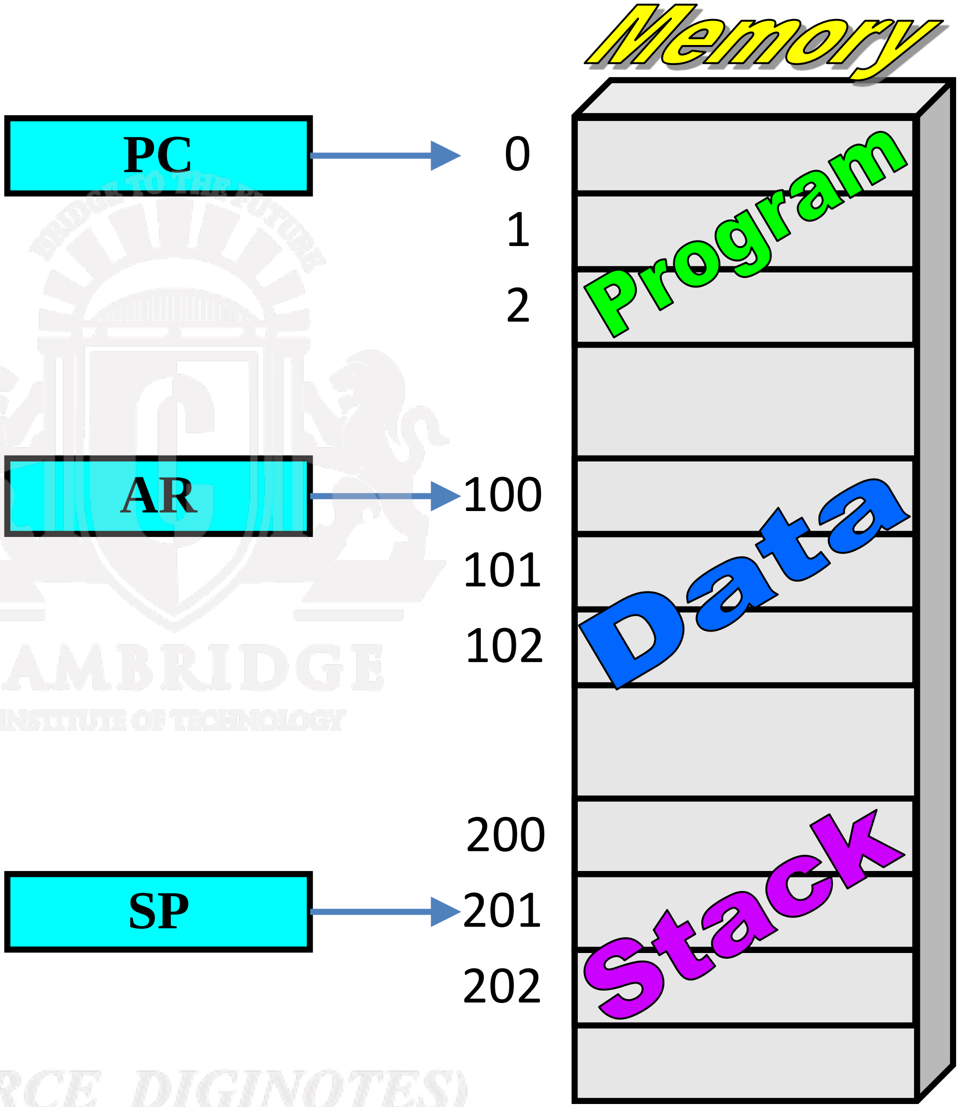
- Memory Stack
  - PUSH

    SP ← SP – 1

    M[SP] ← DR
  - POP

    DR ← M[SP]

    SP ← SP + 1



(SOURCE DIGINOTES)

Stack organization

- Assume a byte addressable memory and a word length of a data to be **32-bits**
- The push operation places a data item above the current top of the stack. i.e. the **SP** is to be decremented before data can be placed.
- So to transfer a data from a memory location NUM to the have the following instructions.
  - **Subtract #4,SP**
  - **Move NUM,(SP)**
- **To remove a data**
  - **Move (SP),NUM**
  - **Add #4,SP**

Reverse Polish Notation

- Infix Notation

  $A + B$

- Prefix or Polish Notation

  $+ A\ B$

- Postfix or Reverse Polish Notation (RPN)

  $A\ B\ +$

$$A * B + C * D \xrightarrow{\text{RPN}} A\ B * C\ D * +$$

$(2)\ (4) * (3)\ (3) * +$

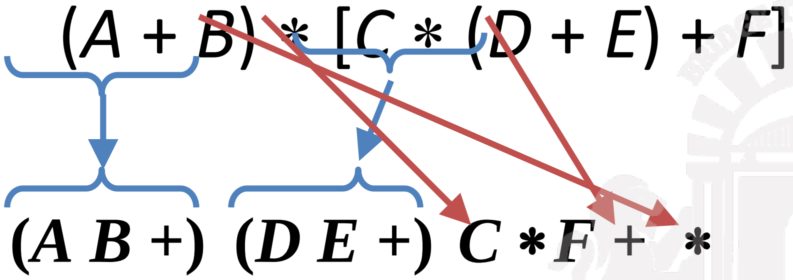
$(8)\ (3)\ (3) * +$

$(8)\ (9) +$

17

Reverse Polish Notation

- Example



$(A + B) * [C * (D + E) + F]$

$(A\ B\ +)\ (D\ E\ +)\ C * F +\ *$

Reverse Polish Notation

- Stack Operation

(3) (4) * (5) (6) * +

PUSH 3

PUSH 4

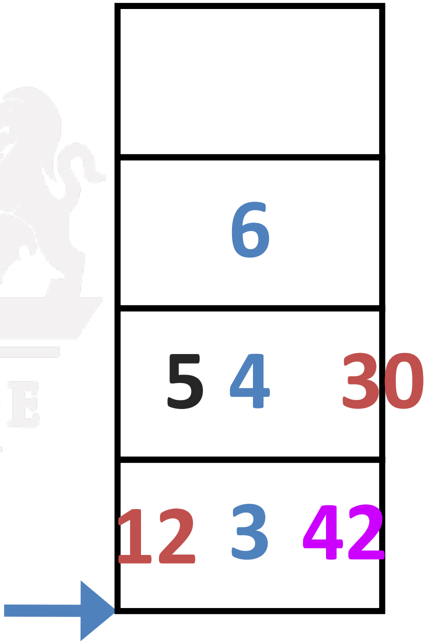
MULT

PUSH 5

PUSH 6

MULT

ADD



STACK:-POP and PUSH

- Suppose that a stack runs from location 2000(BOTTOM) down no further location 1500. The stack pointer is loaded initially with the address value 2004.
- SP is decremented by 4 before new data are stored onto the stack, hence a initial value of 2004 means that the first item pushed onto the stack will be at location 2000.
- To prevent either pushing an item on full stack or popping an item off an empty stack, the single-instruction push and pop operations can be replaced by the instruction sequences
- The compare instruction

Compare src, dst **performs** **[dst]-[src]**

Doesn't change the operand value

Stack: POP and PUSH

- SAFEPOP Compare #2000,SP
- Branch>0 EMPTYERROR
- Move (SP)+,ITEM.
- SAFEPUSH Compare #1500,SP
- Branch<=0 FULLERROR
- Move NEWITEM,-(SP).

Subroutines

- In a given program, it is often necessary to perform a particular subtask many times on different data-values. Such a subtask is usually called a subroutine.

For example,

- a subroutine may evaluate the sine function or sort a list of values into increasing or decreasing order.
- It is possible to include the block of instructions that constitute a subroutine at every place where it is needed in the program. However, to save space, only one copy of the instructions that constitute the subroutine is placed in the memory, and any program that requires the use of the subroutine simply branches to its starting location..

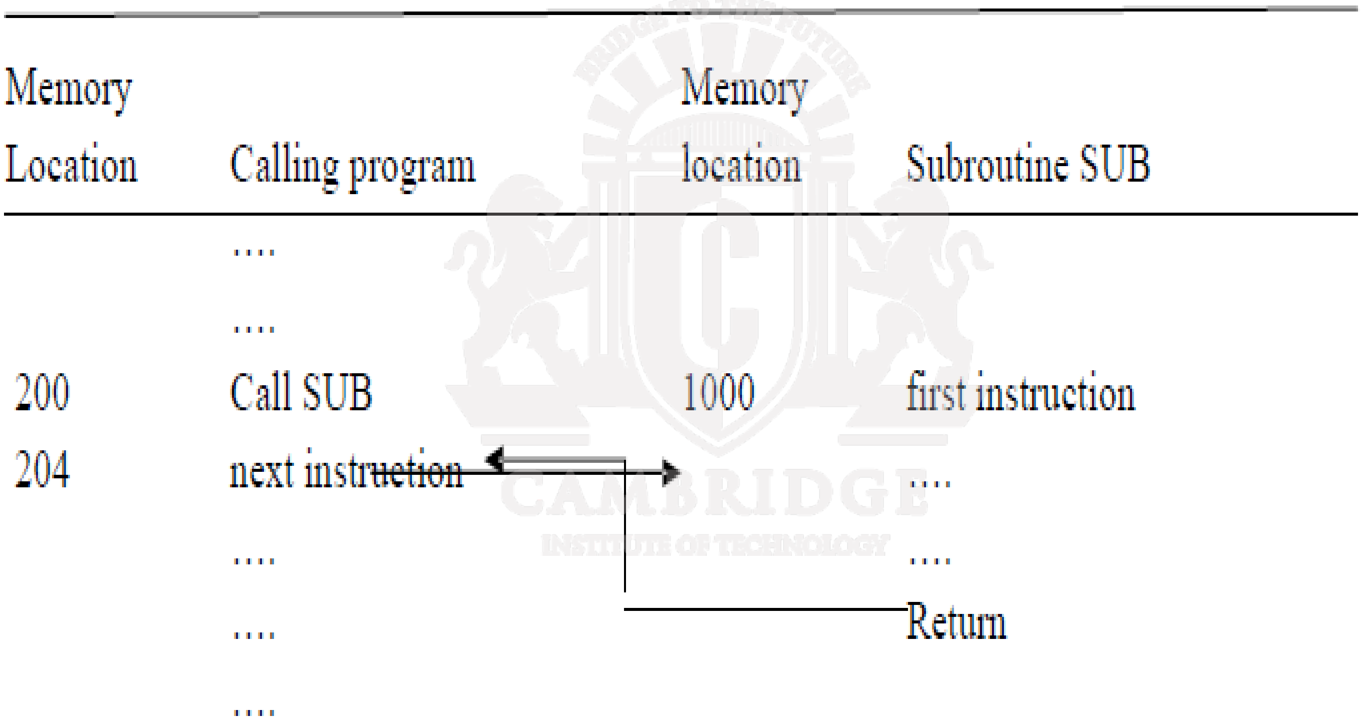
Subroutines

- When a program branches to a subroutine we say that it is calling the subroutine. The instruction that performs this branch operation is named a **Call instruction**

- After a subroutine has been executed, the calling program must resume execution, continuing immediately after the instruction that called the subroutine. The subroutine is said to return to the program that called it by executing a **Return instruction**

- The way in which a computer makes it possible to call and return from subroutines is referred to as its **subroutine linkage method**

Subroutines

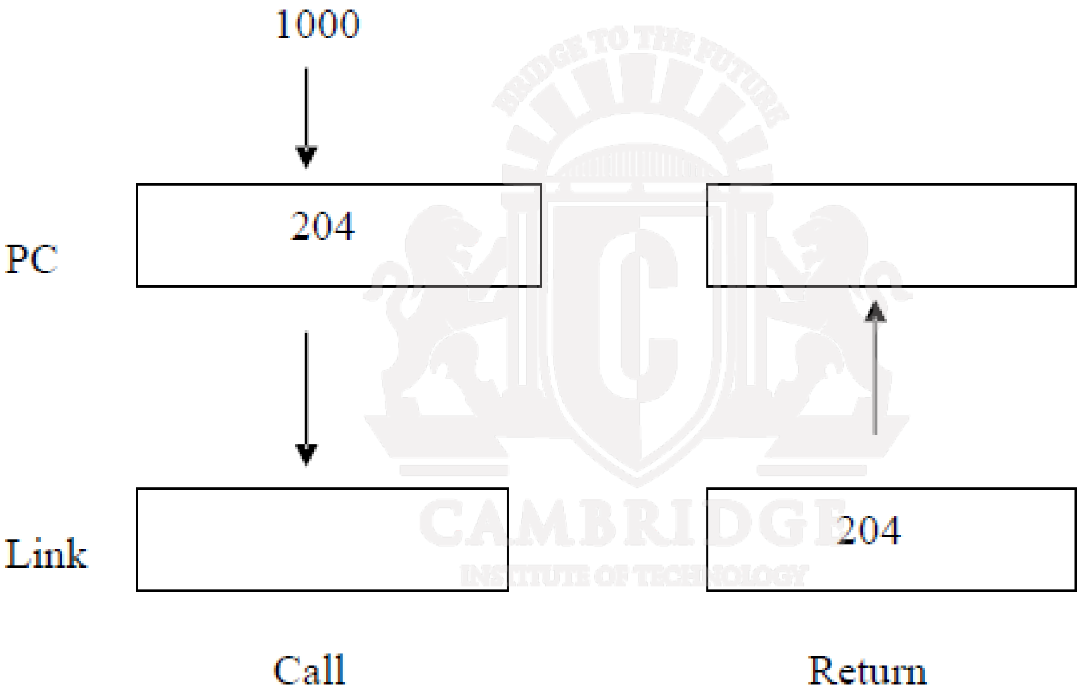


Memory Location	Calling program	Memory location	Subroutine SUB
	....		
	....		
200	Call SUB	1000	first instruction
204	next instruction		....
	....		....
	....		Return
	....		

Subroutines

- The simplest subroutine linkage method is to save the return address in a specific location, which may be a register dedicated to this function. Such a register is called the **link register.**
- When the subroutine completes its task, the Return instruction returns to the calling program by branching indirectly through the link register.
- The Call instruction is just a special branch instruction that performs the following operations
- • *Store the contents of the PC in the link register*
- Branch to the target address specified by the instruction
- The Return instruction is a special branch instruction the performs the operation
- • *Branch to the address contained in the link register*

Subroutines



Subroutine Nesting

- A common programming practice, called **subroutine nesting**, is to have one subroutine call another. In this case, the return address of the second call is also stored in the link register, destroying its previous contents. Hence, it is essential to **save the contents of the link register** in some other location before **calling another subroutine.**
- Otherwise, the **return address** of the first subroutine will be **lost**.
- This suggests that the return addresses associated with subroutine calls should be pushed onto a **stack**. A particular register is designated as the **stack pointer**, SP, to be used in this operation. The stack pointer points to a stack called the **processor stack**. The Call instruction pushes the contents of the PC onto the processor stack and loads the subroutine address into the PC. The Return instruction pops the return address from the processor stack into the PC.

Parameter Passing

- The exchange of information between a calling program and a subroutine is referred to as **parameter passing**. Parameter passing may be accomplished in several ways. The parameters may be placed in **registers** or in **memory locations**, where they can be accessed by the subroutine. Alternatively, the parameters may be placed on the **processor stack** used for saving the return address.

- The purpose of the subroutines is to add a list of numbers. Instead of passing the actual list entries, the calling program passes the address of the first number in the list.
- This technique is called **passing by reference.** The second parameter is **passed by value**, that is, the actual number of entries, n, is passed to the subroutine.

Parameter Passing

- The exchange of **information** between a **calling program** and a **subroutine** is referred to as **parameter passing**. Parameter passing may be accomplished in several ways. The parameters may be placed in **registers** or in **memory locations**, where they can be accessed by the subroutine. Alternatively, the parameters may be placed on the **processor stack** used for saving the return address.

- The purpose of the subroutines is to add a list of numbers. Instead of passing the actual list entries, the calling program passes the **address** of the first number in the list.
- This technique is called **passing by reference.** The second parameter is **passed by value**, that is, the actual number of entries, n, is passed to the subroutine.

Parameter Passing

```
              Move  N,R1
              Move  #NUM1,R2
              Clear  R0
              Call  ARRAYADD
              Move R0,SUM
              ........
              END
ARRAYADD        Add  (R2)+,R0
                Decrement  R1
                Branch>0  ARRAYADD
                Return
```

Calling (main) Program

Subroutine

Stack Frame

During execution of the subroutine, **six locations** at the top of the stack contain entries that are needed by the subroutine.

These locations constitute a ***private workspace*** for the subroutine, created at the time the subroutine is entered and freed up when the subroutine returns control to the calling program. Such space is called a **stack frame** .

Stack Frame

-
- Move #NUM1,-(SP)
- Move N,-(SP)
- **Call ARRAYADD**
- Move 4(SP),SUM
- Add #8,SP
- ……..
- END

Calling (main) Program

Stack Frame

- **ARRAYADD** Move Multiple R0-R2,-(SP)
- Move 16(SP),R1
- Move 20(SP),R2
-
- Clear R0
- **BACK** Add (R2)+,R0
- Decrement R1
- Branch>0 LOOP
- Move R0,20(SP)
- Move Multiple (SP)+,R0-R2
- **Return**

Subroutine

Stack Frame

- In addition to the stack pointer SP, it is useful to have another pointer register, called the **Frame pointer (FP**), for convenient access to the parameters passed to the subroutine and to the local memory variables used by the subroutine
- These **local variables** are only used within the **subroutine**, so it is appropriate to allocate space for them in the stack frame associated with the subroutine. We assume that **four parameters** are passed to the subroutine, **three local variables** are used within the subroutine, and registers **R0** and **R1** need to be saved because they will also be used within the subroutine.
- The pointers **SP** and **FP** are manipulated as the stack frame is built, used, and dismantled for a particular of the subroutine

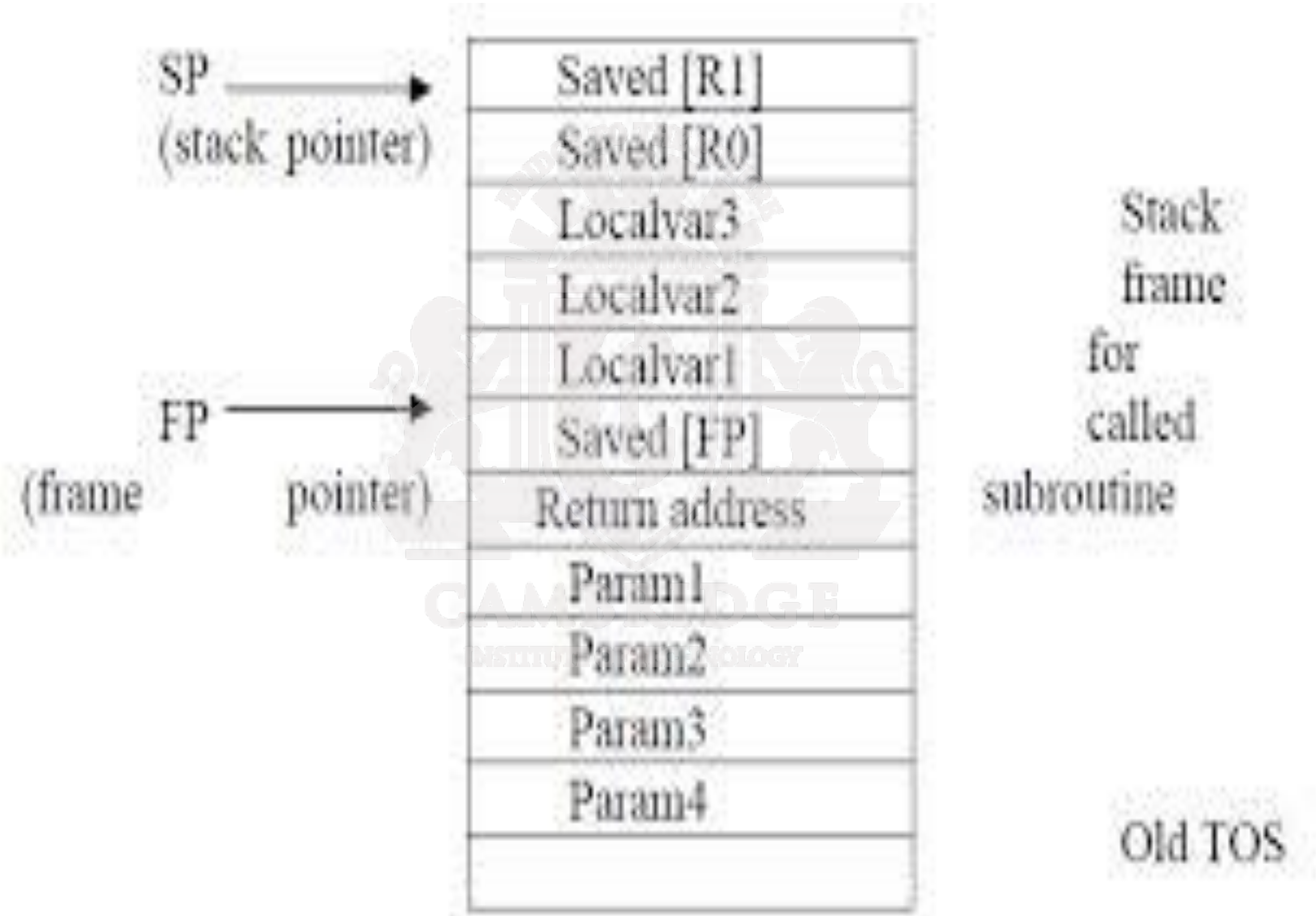
Stack Frame

- We begin by assuming that **SP** point to the old **top-of-stack (TOS)** element in fig b. Before the subroutine is called, the calling program pushes the **four parameters** onto the stack.

- The call instruction is then executed, resulting in the **return address** being pushed onto the stack. Now, **SP** points to this return address, and the first instruction of the subroutine is about to be executed. This is the point at which the frame pointer **FP** is set to contain the proper memory address.

- Since **FP** is usually a general-purpose register, it may contain **information** of use to the Calling program. Therefore, its contents are saved by pushing them onto the stack. Since the SP now points to this position, its contents are copied into FP

Stack Frame



SP (stack pointer)

FP (frame pointer)

Saved [R1]
Saved [R0]
Localvar3
Localvar2
Localvar1
Saved [FP]
Return address
Param1
Param2
Param3
Param4

Stack frame for called subroutine

Old TOS

(SOURCE DIGINOTES)

Queues

- Data structure similar to stack
- **First In First Out**
- Queue has two ends :**One entry** and **One exit.**
- Both the ends of a queue move to **higher addresses** as data are added at the back and removed from the front, so **two pointers** are needed to keep track of the queue operations.
- The queue would continuously move through the memory in the direction of higher address: may leads to a "**queue overflow**"
- Solution is to limit the queue to a fixed region in memory by using a **circular buffer.**

Queues

- Consider the memory addresses from **BEGINNING** to **END** are assigned to queue.
- The first element is entered into the location **BEGINNING** and successive entries are appended to the queue by entering them at successively higher addresses.
- By the time the back of the queue reaches **END** some items have been removed from the queue creating empty spaces.
- Hence the back pointer is reset to the value **BEGINNING** and the process continues.

Additional Instructions

LOGICAL INSTRUCTIONS:-

- The basic logic operations are: AND,OR,NOT, and XOR logic gates
- All the programming languages provide instructions to perform these operations on all bits of a byte or word independently.
- For ex:- **Not dst**
- **Complements all bits contained in the destination operand that are changed to 1’s and 0’s and 0’s to 1’s**

The **dst** may be a processor register or memory location

Additional Instructions

SHIFT AND ROTATE INSTRUCTIONS:-

- There are many applications that require the bits of an operand to be **shifted right** or **left** some specified number of bit positions.
- The details of how the shifts are performed depend on whether the operand is a signed number or some more general binary-coded information.
- For general operands, we use a **logical shift**. For a signed number, we use an **arithmetic shift**, which preserves the sign of the number

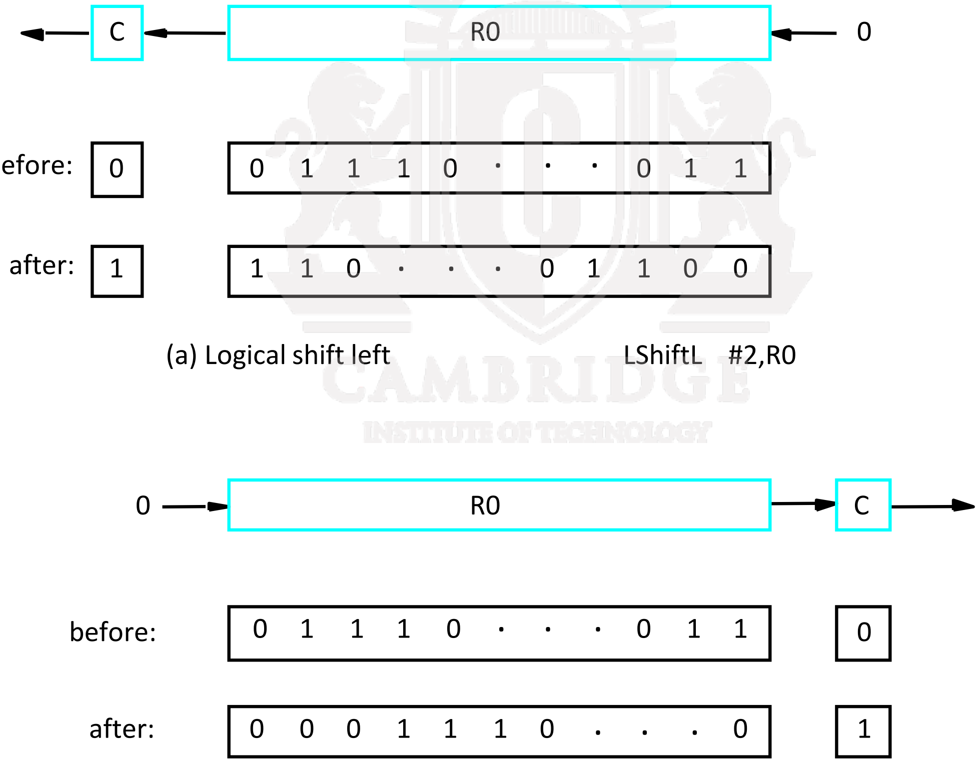
Additional Instructions

- **Logical shifts:-**
- Two logical shift instructions are needed, one for shifting left (LShiftL) and another for shifting right (LShiftR). These instructions shift an operand over a number of bit positions specified in a count operand contained in the instruction.
- The general form of a logical left shift instruction is
-
- LShiftL Count,R0

(a) Logical shift left LShiftL #2, R0

Additional Instructions : Logical Shifts

- Logical shift – shifting left (LShiftL) and shifting right (LShiftR)

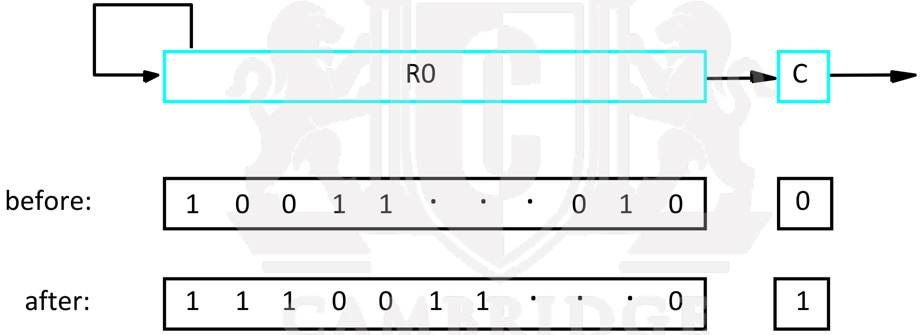


(a) Logical shift left LShiftL #2,R0

(b) Logical shift r ight LShiftR #2,R0

Arithmetic Shifts

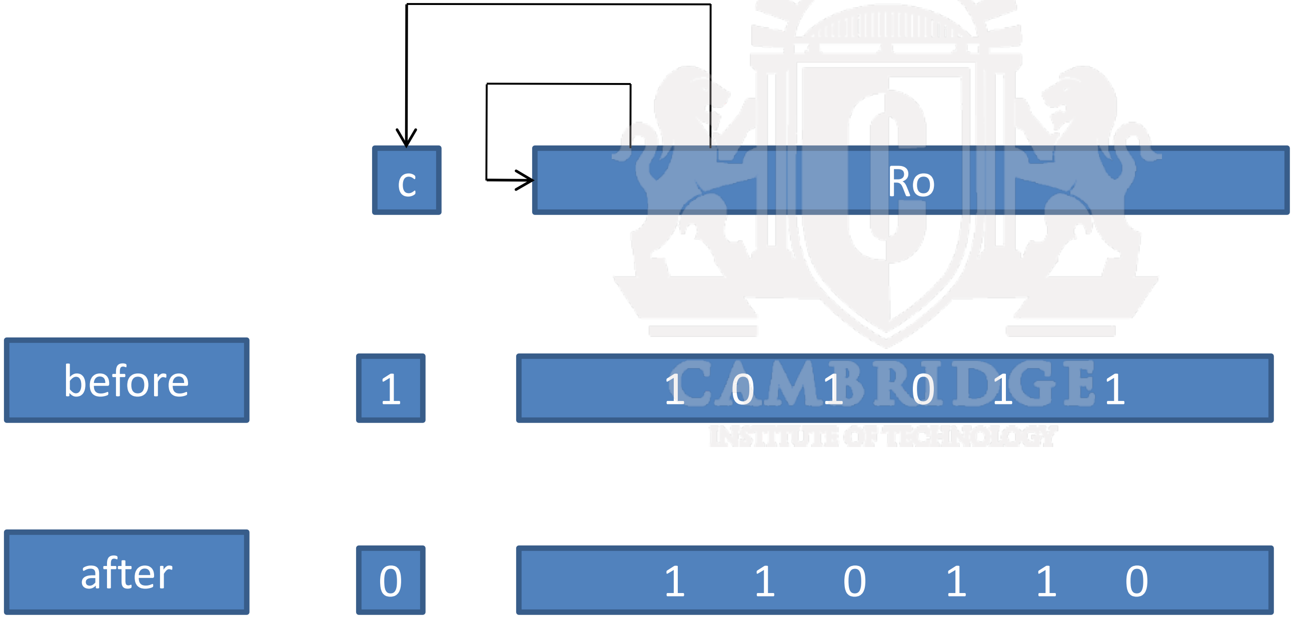
Arithmetic Shift Right :-In this case sign bit is repeated into the bit towards its right and it is also put back in the same position



(c) Ar ithmetic shift right AShiftR #2,R0

Arithmetic Shifts

Arithmetic Shift Left **:-In this case the bit next to the sign bit on the right side is shifted out into the carry bit**



c

Ro

before

1

1 0 1 0 1 1

after

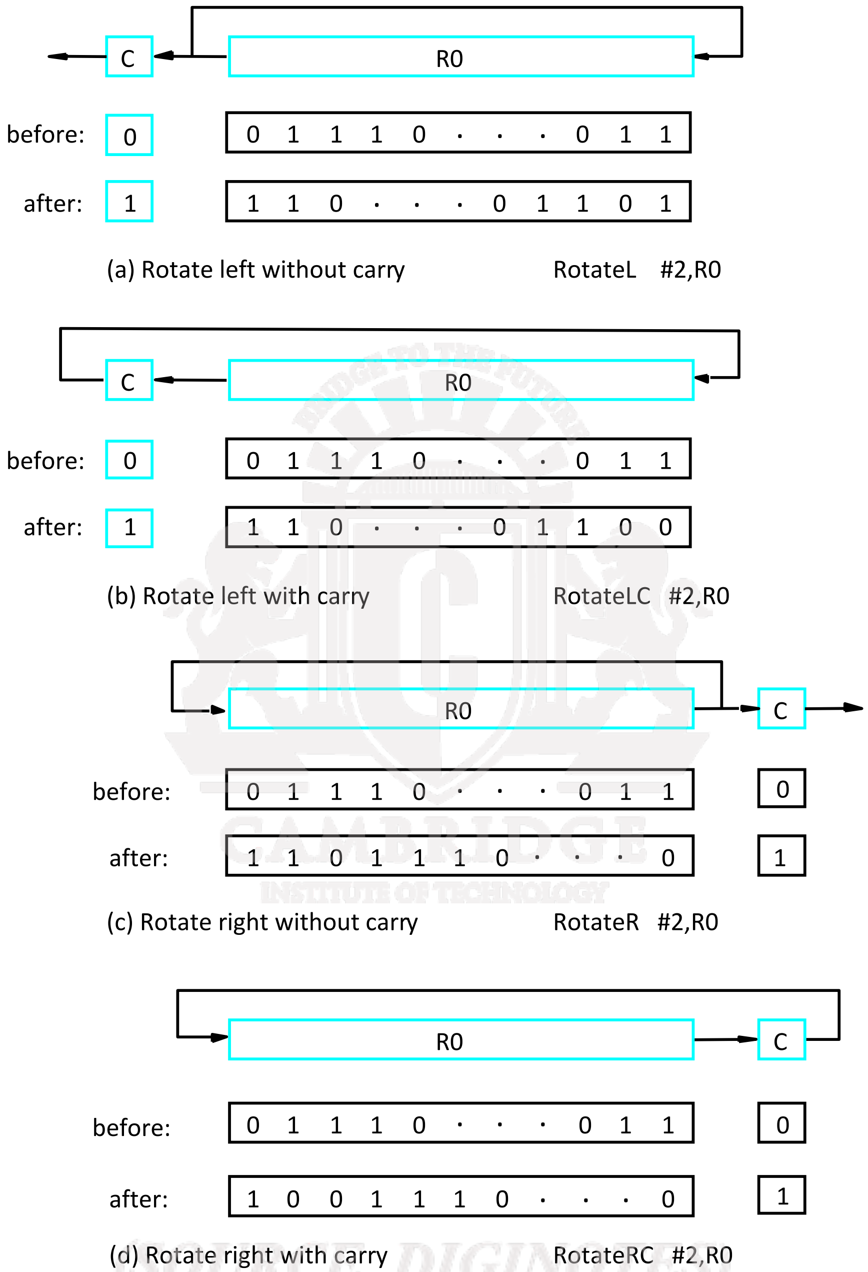
0

1 1 0 1 1 0

AshiftL #1,R0

Rotate Instructions

- In the shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is retained in the **Carry flag C.**
- To **preserve all bits**, a set of **rotate instructions** can be used. They **move the bits** that are shifted out of one end of the operand **back into the other end**.
- Two versions of both the left and right rotate are usually provided. In one version, the bits of the operand are **simply rotated**. In the other version, the rotation includes the **C flag** instructions.
- When carry flag is not involved in the rotation it contains the **last bit shifted out of the register.**
- **All four possibilities of rotate instructions are as shown**



(a) Rotate left without carry RotateL #2,R0

(b) Rotate left with carry RotateLC #2,R0

(c) Rotate right without carry RotateR #2,R0

(d) Rotate right with carry RotateRC #2,R0

Figure 2.32. Rotate instructions.

Encoding of Machine Instructions

- To be executed in a processor, an instruction must be encoded in a **compact binary pattern.** Such encoded instructions are properly referred to as machine instructions.
- The instructions that use symbolic names and acronyms are called **assembly language instructions**, which are converted into the **machine instructions** using the **assembler program**.
- We have seen instructions that perform operations such as **add, subtract, move, shift, rotate, and branch.**
- These instructions may use operands of different sizes, such as **32-bit and 8-bit numbers or 8-bit ASCII-encoded characters**.
- The type of operation that is to be performed and the type of operands used may be specified using an encoded binary pattern referred to as the **OP code** for the given instruction.

Encoding of Machine Instructions

- Suppose that **8** bits are allocated for this purpose, giving **256** possibilities for specifying different instructions. This leaves **24** bits to specify the rest of the required information.
- Let us examine some typical cases. The instruction
- **Add R1, R2**
- Has to specify the registers R1 and R2, in addition to the OP code. If the processor has **16 registers**, then **four bits** are needed to identify each register. **Additional bits** are needed to indicate that the **Register addressing mode** is used for each operand.

Encoding of Machine Instructions

- The instruction **Move 24(R0), R5**
- Requires **16** bits to denote the **OP code** and the **two registers**, and some bits to express that the source operand uses the **Index addressing mode** and that the **index value is 24**.
- In all these examples, the instructions can be encoded in a **32-bit** word. Depicts a possible format.
- There is an 8-bit Op-code field and two 7-bit fields for specifying the source and destination operands. The 7-bit field identifies the addressing mode and the register involved (if any). The "Other info" field allows us to specify the additional information that may be needed, such as an index value or an immediate operand.

(a) One-word instruction

Opcode	Source	Dest	Other info

(b) Two-Word instruction

Opcode	Source	Dest	Other info
Memory address/Immediate operand			

(c) Three-operand instruction

Op code	Ri	Rj	Rk	Other info

Encoding of Machine Instructions

- But, what happens if we want to specify a memory operand using the Absolute addressing mode? The instruction
    - **Move R2, LOC**
- Requires 18 bits to denote the OP code, the addressing modes, and the register.
- This leaves 14 bits to express the address that corresponds to LOC, which is clearly insufficient.
    - **And #$FF000000. R2**
- In which case the **second word** gives a full 32-bit immediate operand. If we want to allow an instruction in which two operands can be specified using the Absolute addressing mode, for example

(a) One-word instruction

Opcode	Source	Dest	Other info

(b) Two-Word instruction

Opcode	Source	Dest	Other info
Memory address/Immediate operand			

(c) Three-operand instruction

Op code	Ri	Rj	Rk	Other info

Encoding of Machine Instructions

- If we want to allow an instruction in which two operands can be specified using the Absolute addressing mode, for example
  - **Move LOC1, LOC2**
- Then it becomes necessary to use two additional words for the **32-bit addresses of the operands**.

MODULE 2 : Input / Output organization

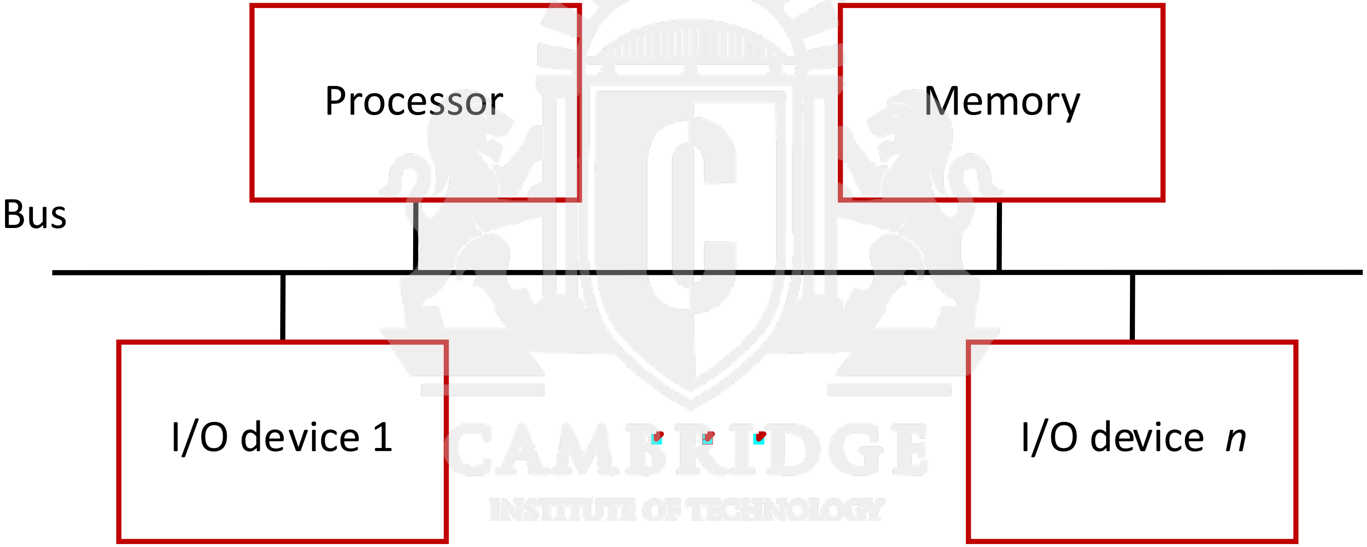
Courtesy: Text book: Carl Hamacher 5th Edition

Accessing I/O devices

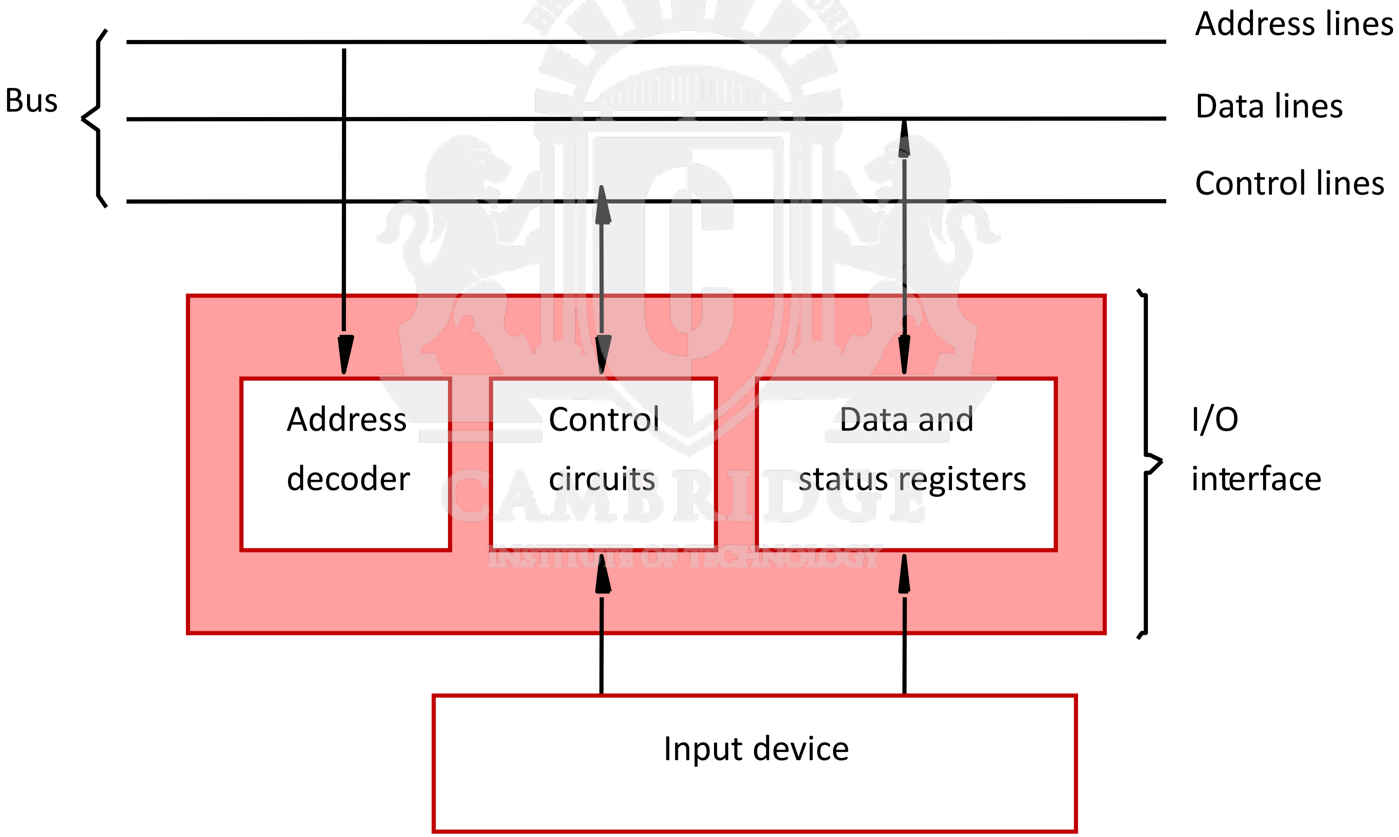
- Multiple I/O devices may be connected to the processor and the memory via a **bus**.
- Bus consists of three sets of lines to carry **address, data** and **control** signals.
- Each I/O device is assigned an **unique** address.
- To access an I/O device, the processor places the address on the address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

The processor requests either a read or a write operation, and the requested data are transferred over the data lines, when I/O devices and the memory share the same address space, the arrangement is called **memory-mapped I/O.**

Accessing I/O devices



Accessing I/O devices (contd..)



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.

Interrupts

- In **Program-controlled I/O**, where the processor repeatedly checks a status flag to achieve the required synchronization between the
- processor and an input or output device. We say that the processor **polls** the device. There are two other commonly used mechanisms for implementing I/O operations: **Interrupts** and **Direct Memory Access(DMA).**
- In the case of interrupts, synchronization is achieved by having the I/O device send a **special signal** over the bus whenever it is ready for a data **transfer operation**. Direct memory access is a technique used for **high-speed I/O devices**.
- It involves having the device interface transfer data directly to or from the memory, without continuous **involvement** by the processor.

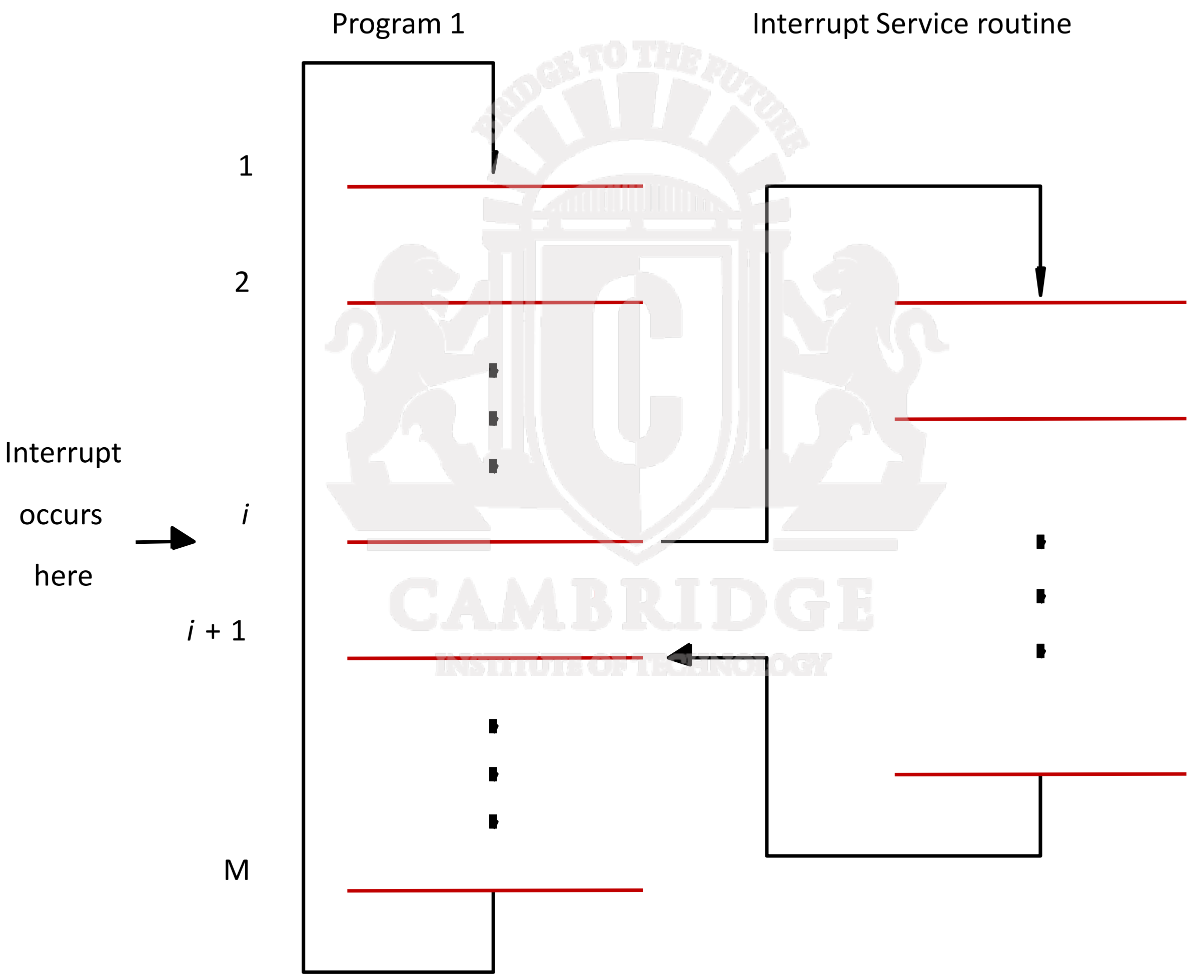
Interrupts

- The routine executed in response to an interrupt request is called the **interrupt- service routine**, which is the **PRINT** routine in our example. Interrupts bear considerable resemblance to subroutine calls. Assume that an interrupt request arrives during execution of instruction i in figure

- **In Interrupt** :- 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.

-

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..)

- Treatment of an interrupt-service routine is very similar to that of a subroutine.
- However there are **significant differences**:
  - 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.**
  - As a result, 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.

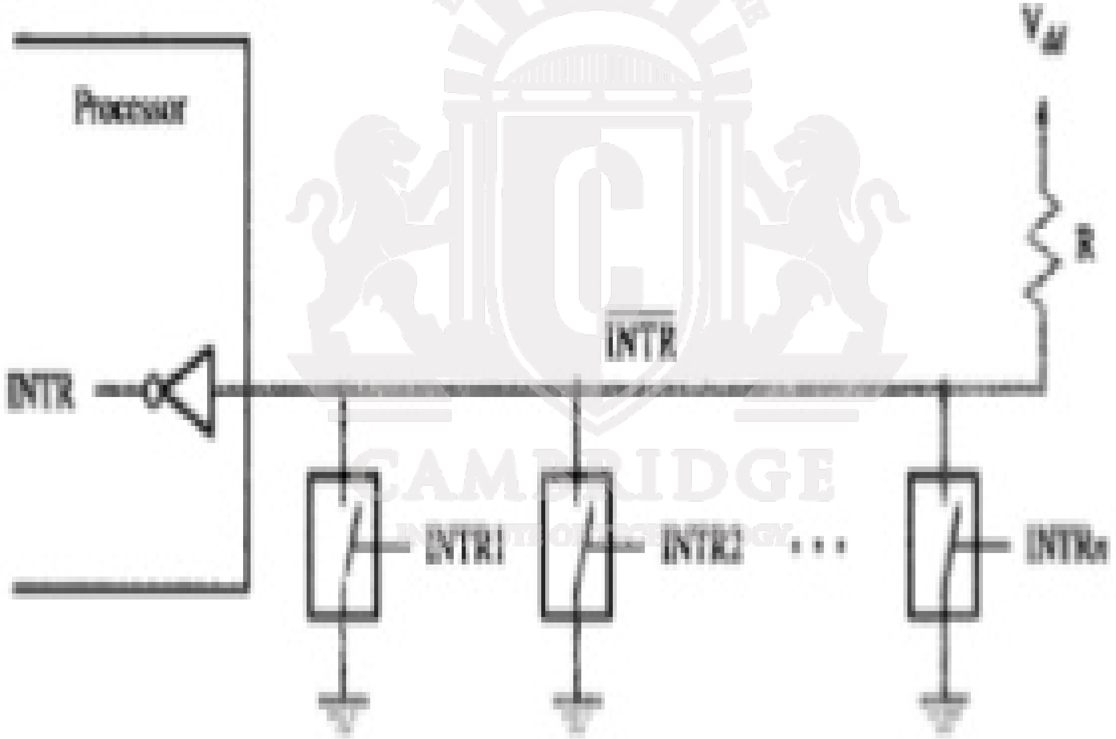
Interrupts (contd..)

- 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 interrupt service routine.

Interrupts Hardware

- We pointed out that an I/O device requests an interrupt by activating a bus line called **interrupt-request.**
- Most computers are likely to have several I/O devices that can request an interrupt. A single interrupt-request line may be used to serve n devices as depicted. All devices are connected to the line via **switches to ground**. To request an interrupt, a device closes its associated **switch.**
- Thus, if all interrupt-request signals **INTR1** to **INTRn** are inactive, that is, if all switches are open, the voltage on the interrupt- request line will be equal to **Vdd**
- This is the inactive state of the line

Interrupts Hardware



(SOURCE DIGINOTES)

Interrupts Hardware

- Since the closing of one or more switches will cause the line voltage to drop to **0,** the value of INTR is the logical OR of the requests from individual devices, that is,

  INTR = INTR1 + ………+INTRn

- It is customary to use the complemented form to name the interrupt-request signal on the common line, because this signal is active when in the low-voltage state

Interrupts (contd..)

Sometime , 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.**

Interrupts (contd..)

- When a processor receives an interrupt-request, it must branch to the **interrupt service routine.**
- It must also **inform** the device that it has **recognized** the interrupt request.

- This can be accomplished in two ways:
  - Some processors have an explicit **interrupt-acknowledge control signal** for this purpose.
  - In other cases, the **data transfer** that takes place between the device and the processor can be used to **inform** the device.

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.
- For example, bits **KIRQ** and **DIRQ** are the interrupt request bits for the keyboard and the display, respectively

Interrupts (contd..)

Vectored Interrupts :- The device requesting an interrupt may **identify itself** directly to the processor.

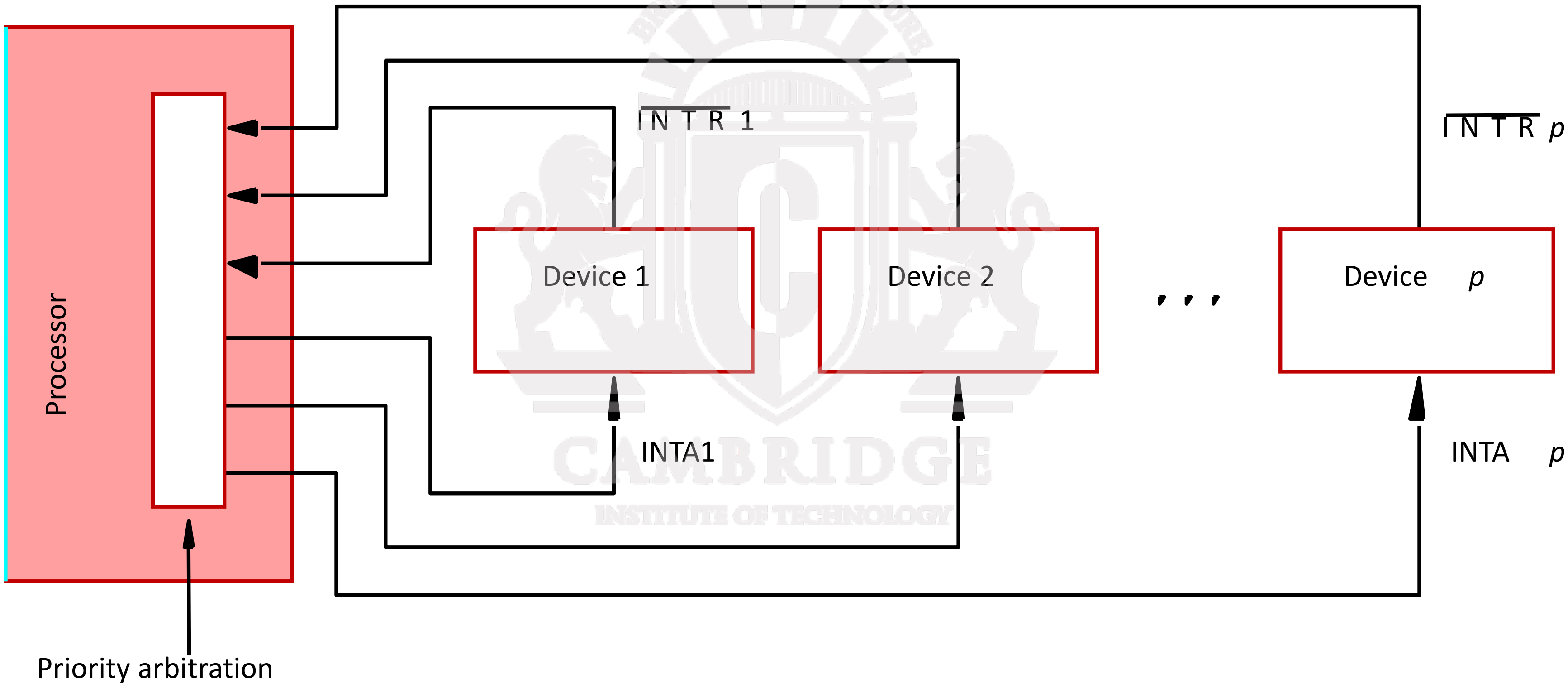
- Device can do so by sending a special code **(4 to 8 bits)** the processor over the bus.
- Code supplied by the device may represent a part of the **starting address** of the **interrupt-service routine.**
- The remainder of the starting address is obtained by the processor based on other information such as the range of memory addresses where interrupt service routines are located.

- Usually the location pointed to by the interrupting device is used to store the starting address of the interrupt-service routine.

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.
  - 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.

Controlling Multiple Devices:- **Priority method**



Controlling Multiple Devices:- **Priority method**

- Each device has a separate **interrupt-request** and **interrupt-acknowledge** line.

- Each interrupt-request line is assigned a different **priority level**.

- Interrupt requests received over these lines are sent to a priority **arbitration circuit** in the processor.

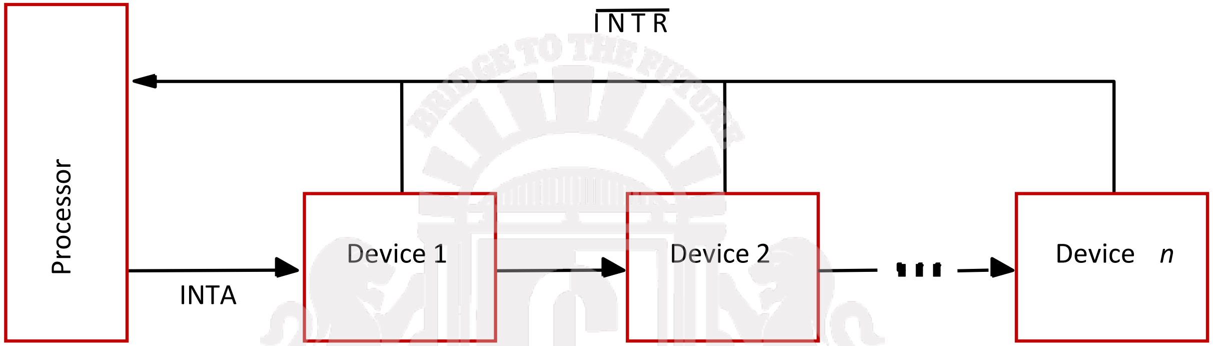
- If the interrupt request has a higher priority level than the priority of the processor, then the request is accepted.

Interrupts (contd..)

Polling scheme:

- If the processor uses a polling mechanism to poll the status registers of I/O devices to determine which device is requesting an interrupt.
- In this case the priority is determined by the order in which the devices are polled.
- The first device with status bit set to 1 is the device whose interrupt request is accepted.

Controlling Multiple Devices:- : **Daisy chain scheme**



- Devices are connected to form a **daisy chain.**
- Devices share the **interrupt-request line**, and **interrupt acknowledge** line is connected to form a daisy chain.
- When devices raise an **interrupt request,** the **interrupt-request line** is activated.
- The processor in response activates **interrupt-acknowledge**.
- Received by device 1, if device 1 does not need service, it passes the signal to device 2.
- Device that is **electrically closest** to the processor has the highest priority.

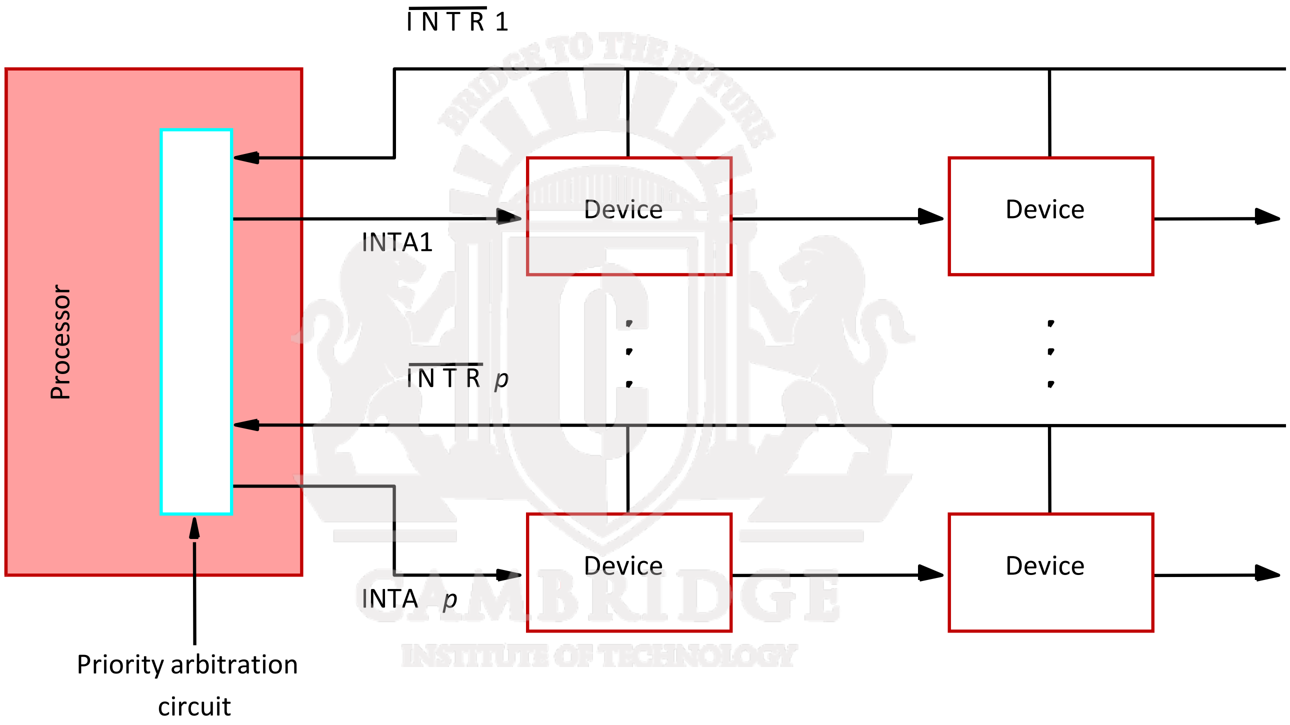
Interrupts (contd..)

- When I/O devices were organized into a **priority** structure, each device had its own **interrupt-request** and **interrupt-acknowledge line**.

- When I/O devices were organized in a **daisy chain** fashion, the devices shared an **interrupt-request line**, and the **interrupt-acknowledge** **propagated** through the devices.

- A **combination** of **priority structure** and **daisy chain** scheme can also used.

Interrupts (contd..)


$\overline{\text{INTR}}$ 1
Processor
Device
Device
INTA1
$\overline{\text{INTR}}$ p
Device
Device
INTA p
Priority arbitration circuit

- Devices are organized into **groups.**
- Each group is assigned a different **priority level.**
- All the devices within a single group share an **interrupt-request line**, and are connected to form a **daisy chain**.

Controlling Multiple Devices

- To control which devices are allowed to generate interrupt requests, the **interface circuit** of each I/O device has an **interrupt-enable bit**.
  - *If the **interrupt-enable bit** in the device interface is set to 1, then the device is allowed to generate an interrupt-request.*
- **Interrupt-enable bit** in the device's **interface circuit** determines whether the device is allowed to generate an interrupt request.
- **Interrupt-enable bit** in the **processor status register** or the priority structure of the interrupts determines whether a given interrupt will be accepted.

Controlling Multiple Devices

- The control needed is usually provided in the form of an interrupt-enable bit in the device’s interface circuit.
- The **Keyboard interrupt-enable, KEN**, and **Display interrupt-enable, DEN,** flags in register CONTROL perform this function.
- If either of these flags is set, the interface circuit generates an interrupt request whenever the corresponding status flag in register STATUS is set.
- At the same time, the interface circuit sets bit **KIRQ** or **DIRQ** to indicate that the keyboard or display unit, respectively, is requesting an interrupt. If an interrupt-enable bit is equal to 0, the interface circuit will not generate an interrupt request, regardless of the state of the status flag .

Exceptions

- Interrupts caused by interrupt-requests sent by I/O devices.
- Interrupts could be used in many other situations where the execution of one program needs to be suspended and execution of another program needs to be started.
- In general, the term **Exception** is used to refer to any event that causes an interruption.
  - Interrupt-requests from I/O devices is one type of an exception.
- Other types of exceptions are:
  - **Recovery from errors**
  - **Debugging**
  - **Privilege exception**

Exceptions (contd..)

- Many sources of errors in a processor. For example:
  - Error in the data stored.
  - Error during the execution of an instruction.
- When such errors are detected, exception processing is initiated.
  - Processor takes the **same steps** as in the case of I/O interrupt-request.
  - It suspends the execution of the current program, and starts executing an **exception-service routine.**
- Difference between handling I/O interrupt-request and handling exceptions due to errors:
  - In case of I/O interrupt-request, the processor usually completes the execution of an instruction in progress before branching to the interrupt-service routine.
  - In case of exception processing however, the execution of an instruction in progress usually **cannot be completed**.

Exceptions (contd..)

- Debugger uses exceptions to provide important features:
  - Trace,
  - Breakpoints.
- **Trace mode:**
  - Exception occurs after the execution of **every instruction.**
  - Debugging program is used as the exception-service routine.
- **Breakpoints:**
  - Exception occurs only at **specific points** selected by the user.
  - Debugging program is used as the exception-service routine.

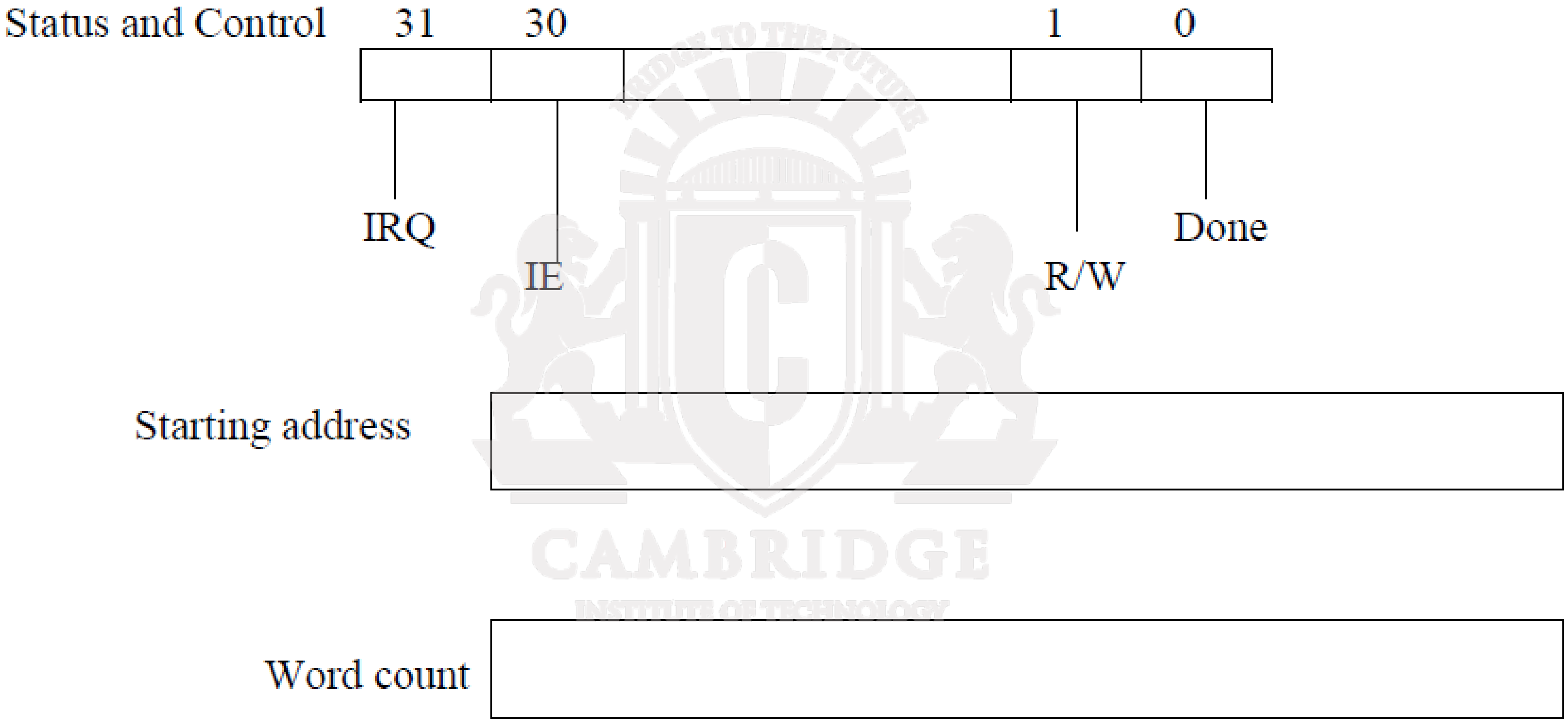
Exceptions (contd..)

- Certain instructions can be executed only when the processor is in the supervisor mode. These are called privileged instructions.
- If an attempt is made to execute a privileged instruction in the user mode, a privilege exception occurs.
- Privilege exception causes:
  - **Processor to switch to the supervisor mode,**
  - **Execution of an appropriate exception-servicing routine.**

Direct Memory Access

- **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.

Registers of **DMA controller**



Status and Control

31

30

1

0

IRQ

IE

R/W

Done

Starting address

Word count

Figure 4 Registers in DMA interface

Registers of **DMA controller**

DMA controller registers that are accessed by the processor to initiate transfer operations.

Two registers are used for storing the **Starting address** and the **word count.** The third register contains **status** and **control flags**.

The **R/W** bit determines the direction of the transfer. When this bit is set to 1 by a program instruction, the controller performs a read operation, that is, it transfers data from the memory to the I/O device. Otherwise, it performs a write operation.

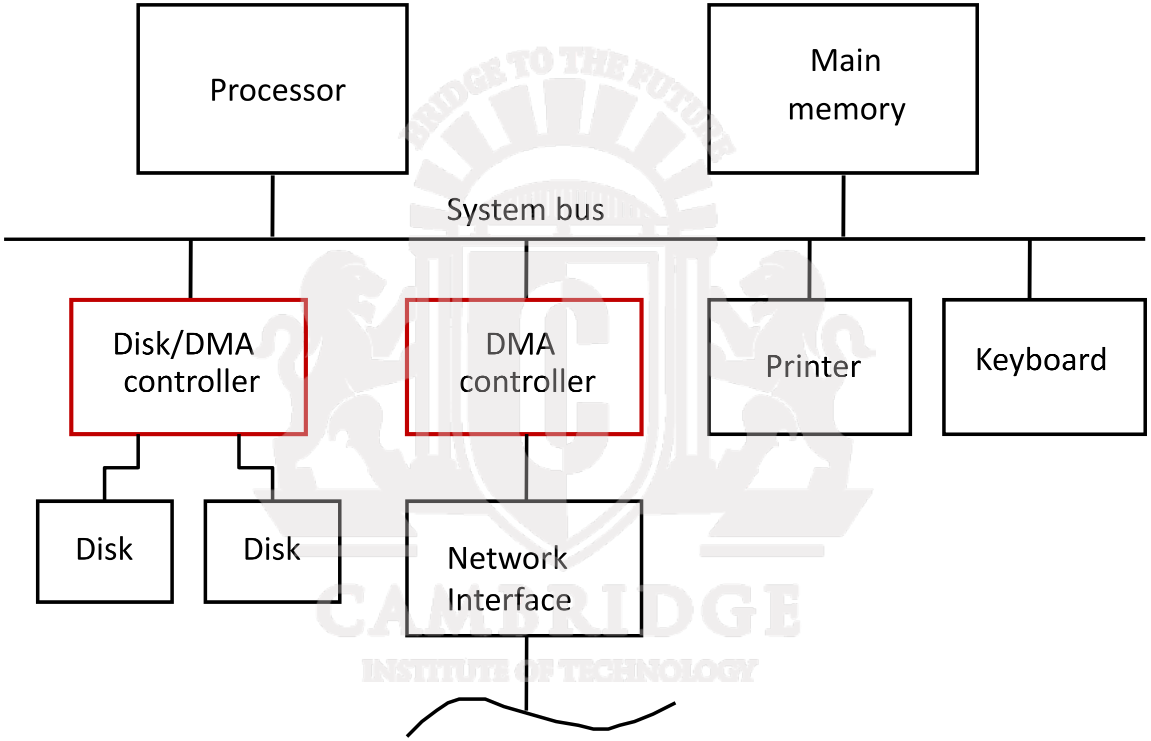
When the controller has completed transferring a block of data and is ready to receive another command, it sets the **Done flag to 1.** Bit 30 is the **Interrupt-enable flag, IE.**

When this flag is set to 1, it causes the controller to raise an interrupt after it has completed transferring a block of data. Finally, the controller sets the **IRQ bit to 1** when it has requested an interrupt.

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



- *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.*

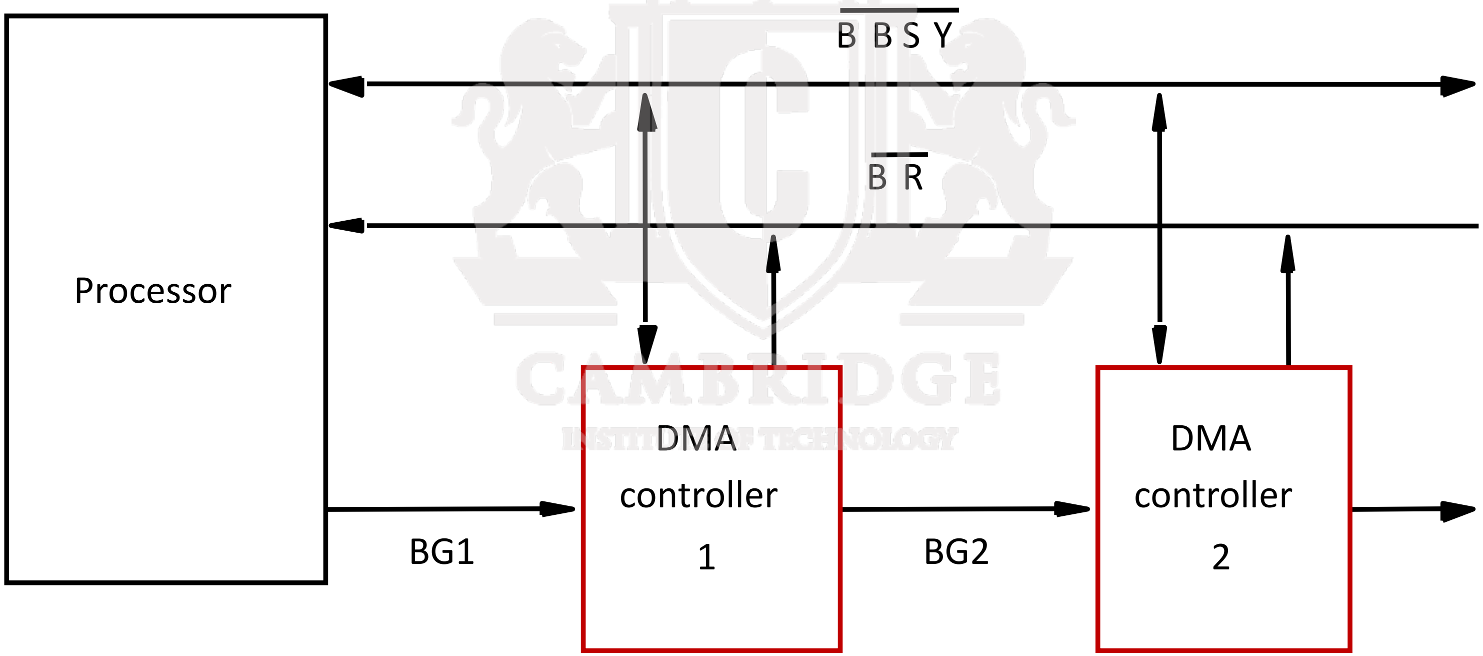
Direct Memory Access (contd..)

- Memory accesses by the processor and the DMA controller are interwoven. Requests by DMA devices for using the bus are always given **higher priority** than processor requests.
- Among different DMA devices, top priority is given to high-speed peripherals such as a **disk, a high-speed network interface, or a graphics display device**.
- Since the processor originates most memory access cycles, the DMA controller can be said to **"steal" memory cycles** from the processor. Hence, the interweaving technique is usually called **cycle stealing**.
- Alternatively, the DMA controller **may be given exclusive access** to the main memory to transfer a block of data without interruption. This is known as **block or burst mode**.

Bus arbitration

- **Processor and DMA controllers** both need to initiate data transfers on the bus and access main memory.
- The device that is allowed to initiate transfers on the bus at any given time is called the **bus master.**
- When the current bus master relinquishes its status as the bus master, another device can acquire this status.
    - The process by which the next device to become the bus master is selected and bus mastership is transferred to it is called **bus arbitration.**
- **Centralized arbitration:**
    - A single bus arbiter performs the arbitration.
- **Distributed arbitration:**
    - All devices participate in the selection of the next bus master.

Centralized Bus Arbitration



Centralized Bus Arbitration(cont.,)

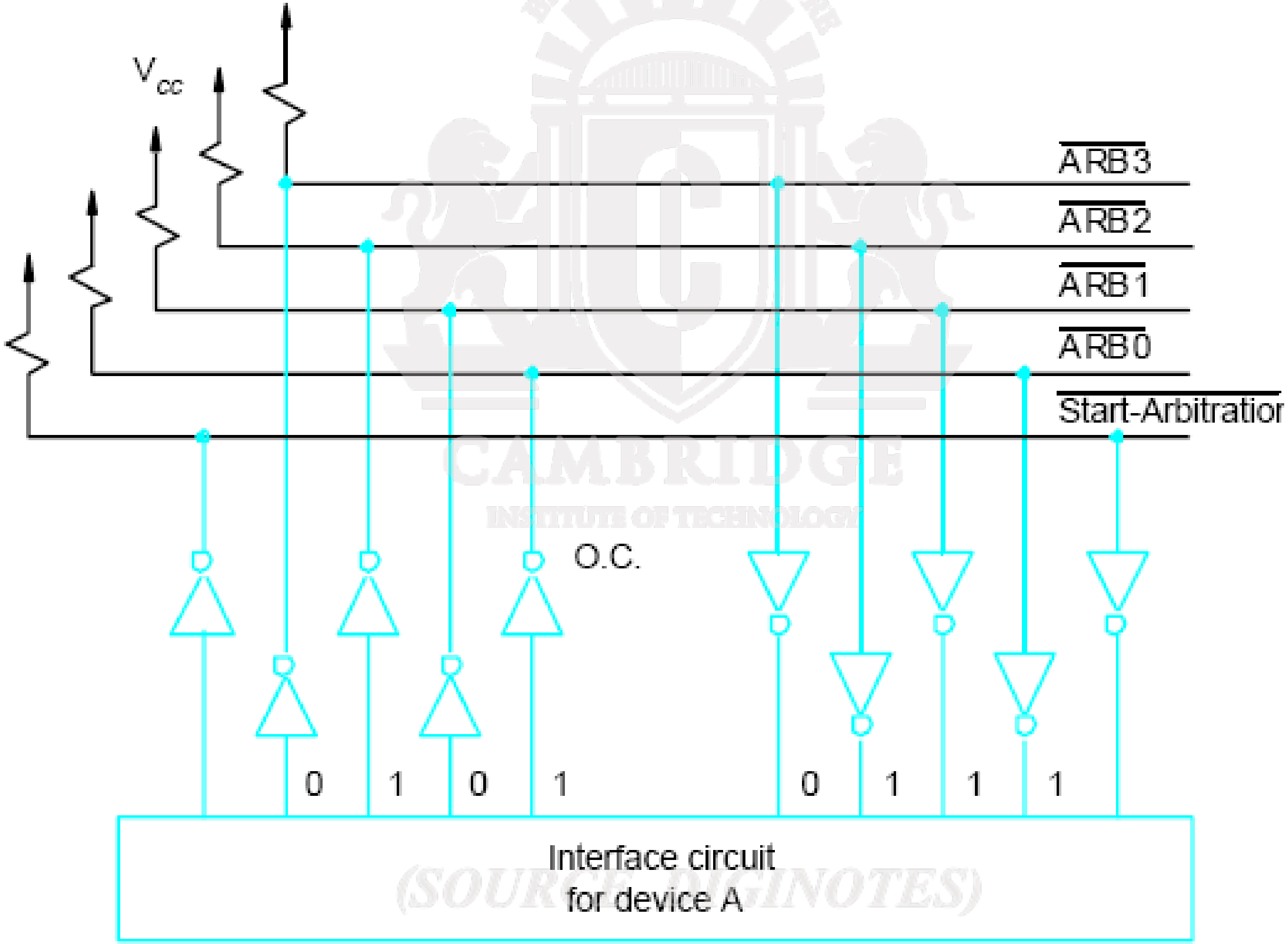
- Bus arbiter may be the processor or a separate unit connected to the bus.
- Normally, the processor is the bus master, unless it grants bus membership to one of the DMA controllers.
- DMA controller requests the control of the bus by asserting the **Bus Request (BR)** line.
- In response, the processor activates the **Bus-Grant1 (BG1**) line, indicating that the controller may use the bus when it is free.
- **BG1** signal is connected to all DMA controllers in a daisy chain fashion.
- **BBSY** signal is 0, it indicates that the bus is busy. When **BBSY** becomes **1**, the DMA controller which asserted **BR** can acquire control of the bus.

(SOURCE DIGINOTES)

Distributed arbitration

- All devices waiting to use the bus share the responsibility of carrying out the arbitration process.
  - Arbitration process does not depend on a central arbiter and hence distributed arbitration has higher reliability.
- Each device is assigned a 4-bit ID number.
- All the devices are connected using 5 lines, 4 arbitration lines to transmit the ID, and one line for the Start-Arbitration signal.
- To request the bus a device:
  - **Asserts the Start-Arbitration signal.**
  - **Places its 4-bit ID number on the arbitration lines**.
- The pattern that appears on the arbitration lines is the logical-OR of all the 4-bit device IDs placed on the arbitration lines.

Distributed arbitration



Distributed arbitration(Contd.,)

- ***Arbitration process:***
  - Each device compares the pattern that appears on the arbitration lines to its own ID, starting with MSB.
  - If it detects a difference, it transmits 0s on the arbitration lines for that and all lower bit positions.
  - The pattern that appears on the arbitration lines is the logical-OR of all the 4-bit device IDs placed on the arbitration lines.

Distributed arbitration (contd..)

- *Device A has the ID 5 and wants to request the bus:*
  - *Transmits the pattern 0101 on the arbitration lines.*
- *Device B has the ID 6 and wants to request the bus:*
  - *Transmits the pattern 0110 on the arbitration lines.*
- *Pattern that appears on the arbitration lines is the logical OR of the patterns:*
  - *Pattern 0111 appears on the arbitration lines.*

Arbitration process:

- *Each device compares the pattern that appears on the arbitration lines to its own ID, starting with MSB.*
- *If it detects a difference, it transmits 0s on the arbitration lines for that and all lower bit positions.*
- *Device A compares its ID 5 with a pattern 0101 to pattern 0111.*
- *It detects a difference at bit position 0, as a result, it transmits a pattern 0100 on the arbitration lines.*
- *The pattern that appears on the arbitration lines is the logical-OR of 0100 and 0110, which is 0110.*
- *This pattern is the same as the device ID of B, and hence B has won the arbitration.*

Buses

- **Processor, main memory, and I/O devices** are interconnected by means of a bus.

- Bus provides a communication path for the transfer of data.
  - **Bus also includes lines to support interrupts and arbitration.**

- A **bus protocol** is the set of rules that govern the behavior of various devices connected to the bus, as to when to **place information** on the bus, when to **assert control signals**, etc.

Buses (contd..)

- Bus lines may be grouped into three types:
  - Data
  - Address
  - Control
- Control signals specify:
  - Whether it is a read or a write operation.
  - Required size of the data, when several operand sizes (byte, word, long word) are possible.
  - Timing information to indicate when the processor and I/O devices may place data or receive data from the bus.
- Schemes for timing of data transfers over a bus can be classified into:
  - **Synchronous,**
  - **Asynchronous.**

Bus master and slaves

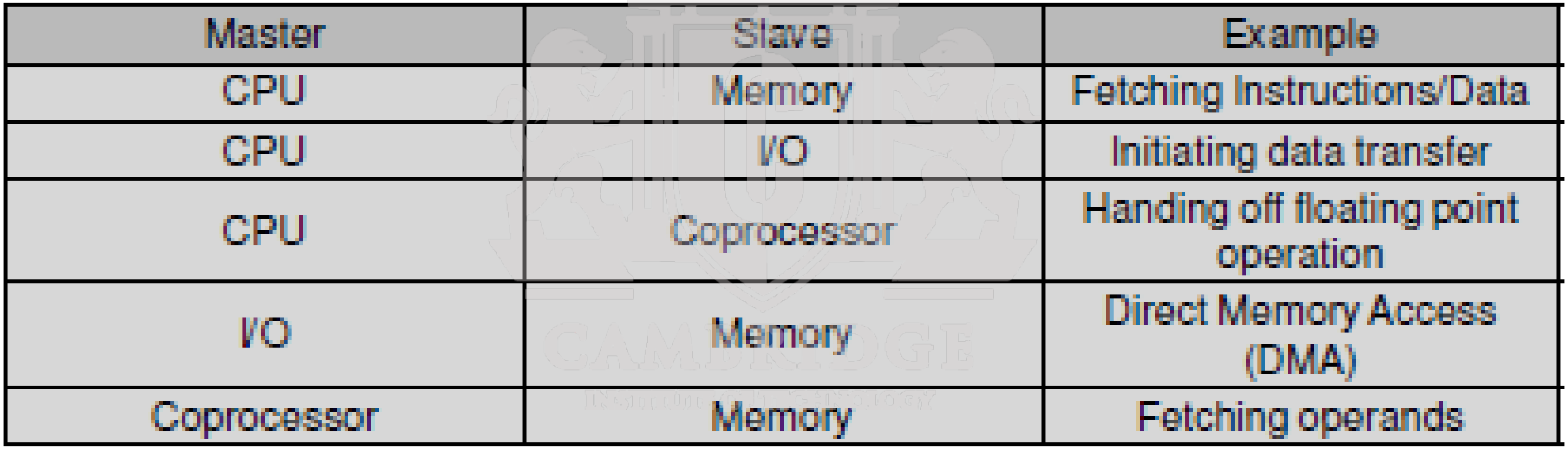
Active devices attached to the bus that can initiate bus transfers are called **masters**

Passive devices that wait for requests are called **slaves**

Some devices may act as slaves at some times and masters at others

Memory can **never** be a master device.

Examples of Bus master and slaves



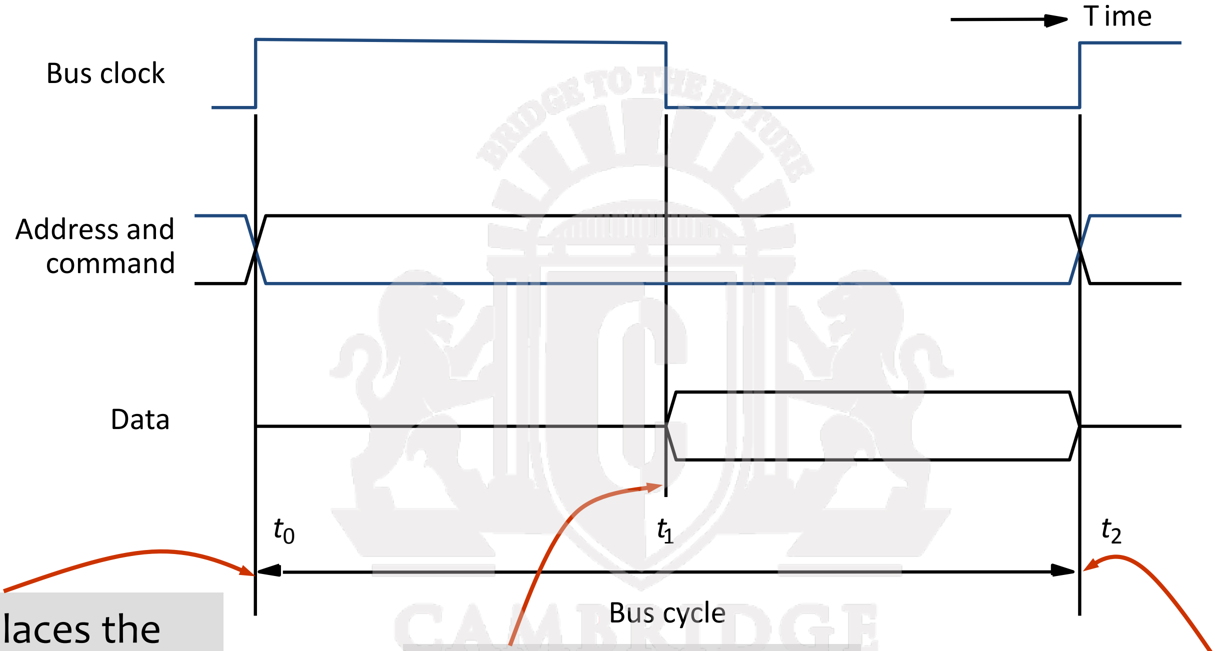
Master	Slave	Example
CPU	Memory	Fetching Instructions/Data
CPU	I/O	Initiating data transfer
CPU	Coprocessor	Handing off floating point operation
I/O	Memory	Direct Memory Access (DMA)
Coprocessor	Memory	Fetching operands

(SOURCE DIGINOTES)

Synchronous bus (contd..)

- In a **synchronous bus**, all devices derive timing information from a common clock line.
- Equally spaced pulses on this line define **equal time intervals**.
- In the simplest form of a synchronous bus, each of these intervals constitutes a **bus cycle** during which one data transfer can take place
- The address and data lines in this and subsequent figures are shown as **high and low at the same time**. This is a common convention indicating that some lines are high and some low, depending on the particular address or data pattern being transmitted.
- The **crossing points** indicate the times at which these patterns change.

Synchronous bus (contd..)



Master places the device address and command on the bus, and indicates that it is a Read operation.

Addressed slave places data on the data lines

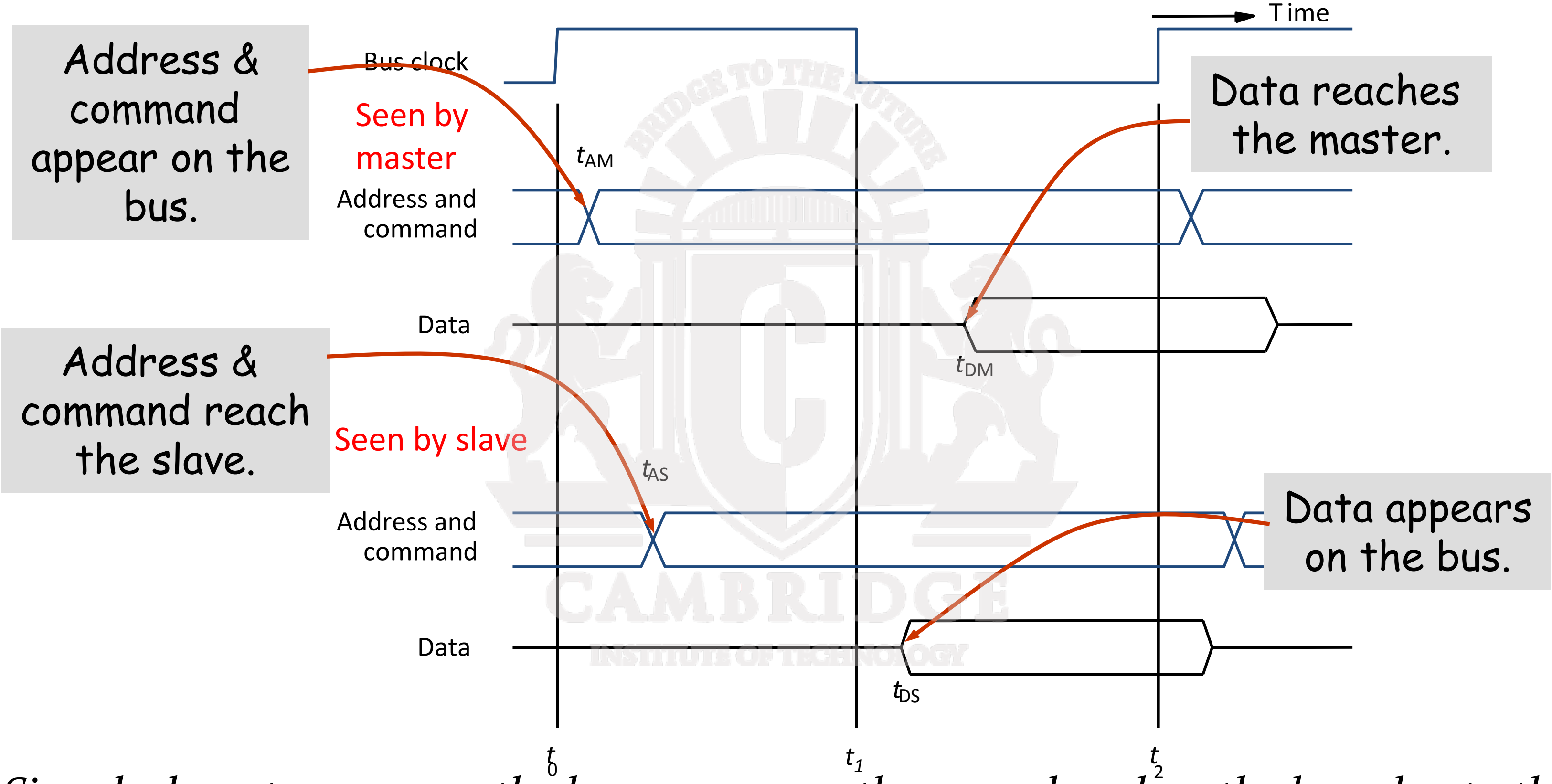
Master "strobes" the data on the data lines into its input buffer, for a Read operation.

- *In case of a Write operation, the master places the data on the bus along with the address and commands at time t_o.*
- *The slave strobes the data into its input buffer at time t_2.*

Synchronous bus (contd..)

- Once the master places the device address and command on the bus, it takes time for this information to propagate to the devices:
  - This time depends on the **physical and electric characteristics** of the bus.
- Also, all the devices have to be given enough time to **decode** the address and control signals, so that the addressed slave can place data on the bus.
- Width of the pulse $t_1 - t_0$ depends on:
  - Maximum propagation delay between two devices connected to the bus.
  - Time taken by all the devices to decode the address and control signals, so that the addressed slave can respond at time t_1.

Synchronous bus (contd..)



Address & command appear on the bus.

Seen by master

Data reaches the master.

Address & command reach the slave.

Seen by slave

Data appears on the bus.

- *Signals do not appear on the bus as soon as they are placed on the bus, due to the propagation delay in the interface circuits.*
- *Signals reach the devices after a propagation delay which depends on the characteristics of the bus.*
- *Data must remain on the bus for some time after t_2 equal to the hold time of the buffer.*

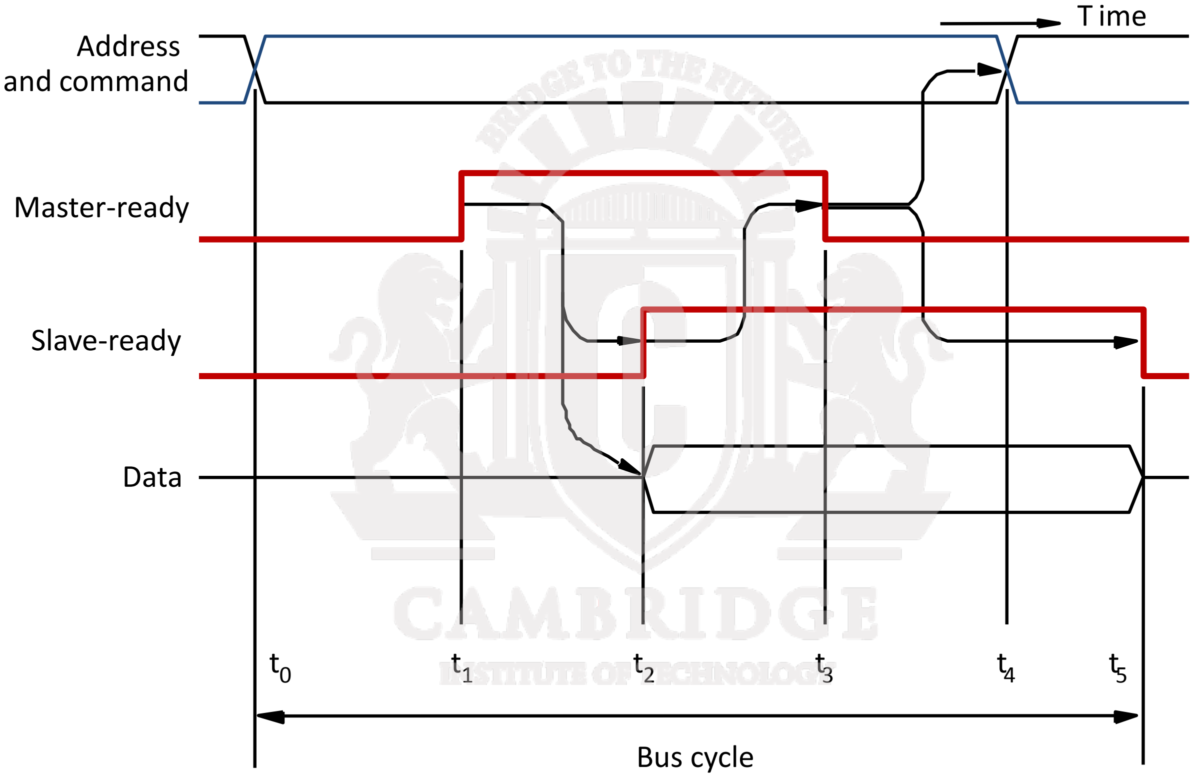
Asynchronous bus

- Data transfers on the bus is controlled by a **handshake protocol** between the master and the slave.
- Common clock in the synchronous bus case is replaced by two **timing control lines**:
  - Master-ready,
  - Slave-ready.
- Master-ready signal is asserted by the master to indicate to the slave that it is ready to participate in a data transfer.
- Slave-ready signal is asserted by the slave in response to the master-ready from the master, and it indicates to the master that the slave is ready to participate in a data transfer.

Asynchronous bus (contd..)

- Data transfer using the handshake protocol:
  - Master places the address and command information on the bus.
  - **Asserts** the **Master-ready signal** to indicate to the slaves that the address and command information has been placed on the bus.
  - All devices on the bus **decode the address**.
  - Address slave performs the required operation, and informs the processor it has done so by asserting the **Slave-ready signal.**
  - Master removes all the signals from the bus, once **Slave-ready is asserted.**
  - If the operation is a Read operation, Master also **strobes** the data into its input buffer.

Asynchronous bus (contd..)



***t_0* - Master places the address and command information on the bus.**

***t_1* - Master asserts the Master-ready signal. Master-ready signal is asserted at t_1 instead of t_0**

***t_2* - Addressed slave places the data on the bus and asserts the Slave-ready signal.**

***t_3* - Slave-ready signal arrives at the master.**

***t_4* - Master removes the address and command information.**

***t_5* - Slave receives the transition of the Master-ready signal from 1 to 0. It removes the data and the Slave-ready signal from the bus.**

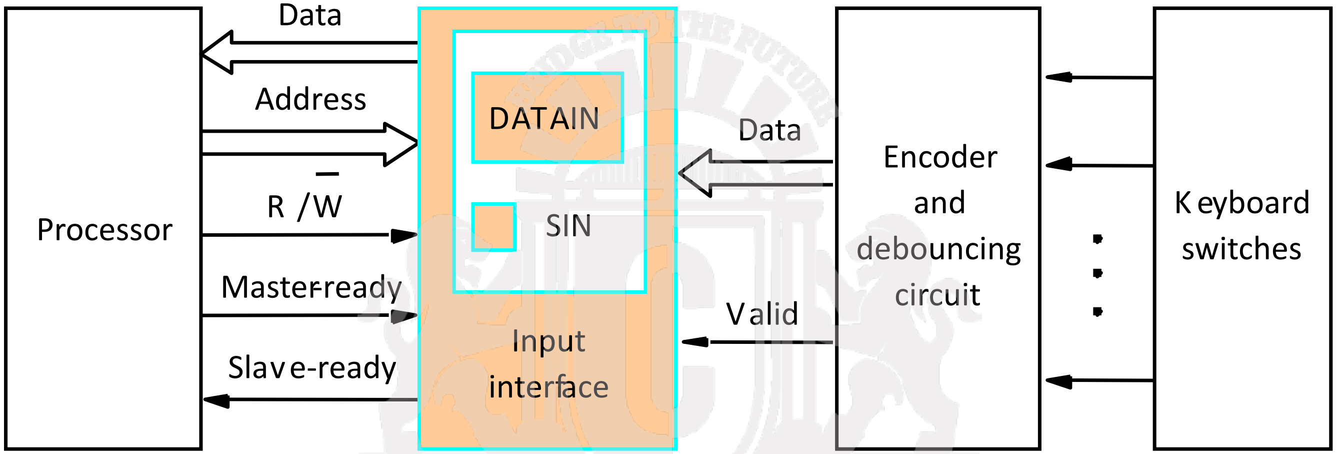
Interface circuits

- I/O interface consists of the circuitry required to connect an I/O device to a computer bus.
- Side of the interface which connects to the computer has bus signals for:
    - Address,
    - Data
    - Control
- Side of the interface which connects to the I/O device has:
    - Datapath and associated controls to transfer data between the interface and the I/O device.
    - This side is called as a <u>“port”.</u>
- Ports can be classified into two:
    - Parallel port,
    - Serial port.

Interface circuits (contd..)

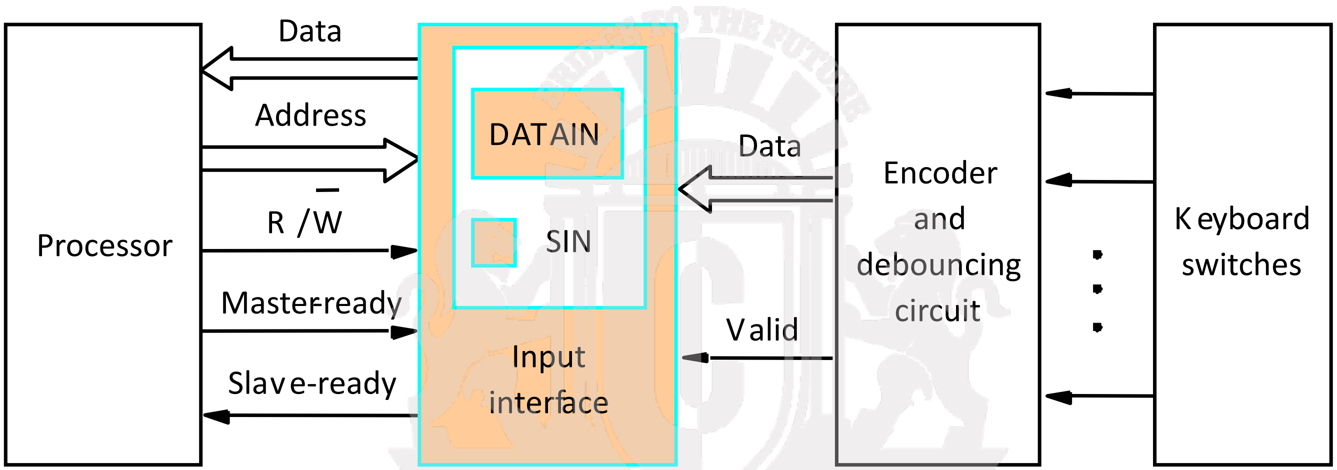
- Parallel port transfers data in the form of a number of bits, normally 8 or 16 to or from the device.
- Serial port transfers and receives data one bit at a time.
- Processor communicates with the bus in the same way, whether it is a parallel port or a serial port.
  - Conversion from the parallel to serial and vice versa takes place inside the interface circuit.

Parallel port



- *Keyboard is connected to a processor using a parallel port.*
- *Processor is 32-bits and uses memory-mapped I/O and the asynchronous bus protocol.*
- *On the processor side of the interface we have:*
  - *Data lines.*
  - *Address lines*
  - *Control or R/W line.*
  - *Master-ready signal and*
  - *Slave-ready signal.*

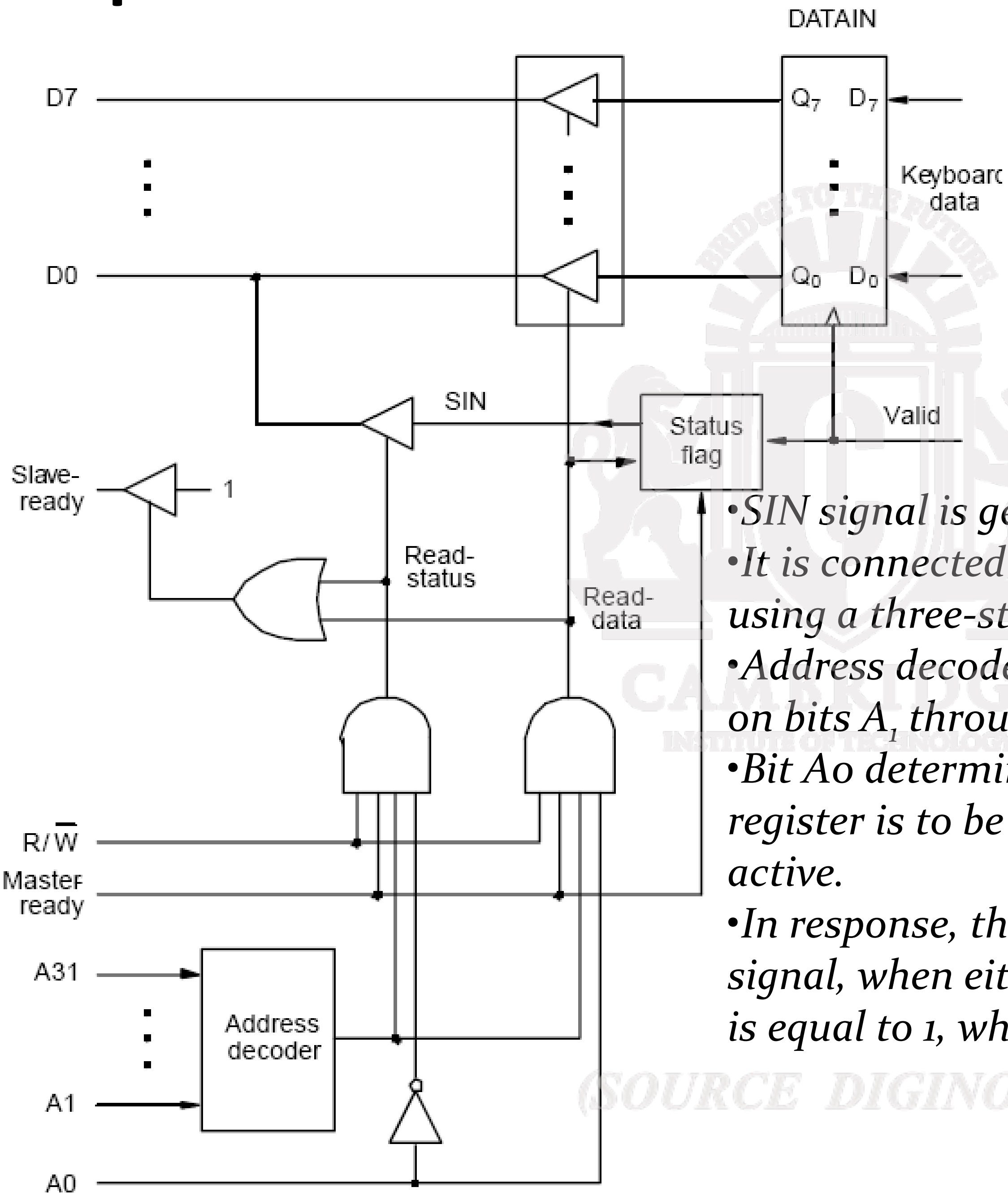
Parallel port (contd..)



•*On the keyboard side of the interface:*

- *Encoder circuit which generates a code for the key pressed.*
- *Debouncing circuit which eliminates the effect of a key bounce (a single key stroke may appear as multiple events to a processor).*
- *Data lines contain the code for the key.*
- *Valid line changes from 0 to 1 when the key is pressed. This causes the code to be loaded into DATAIN and SIN to be set to 1.*

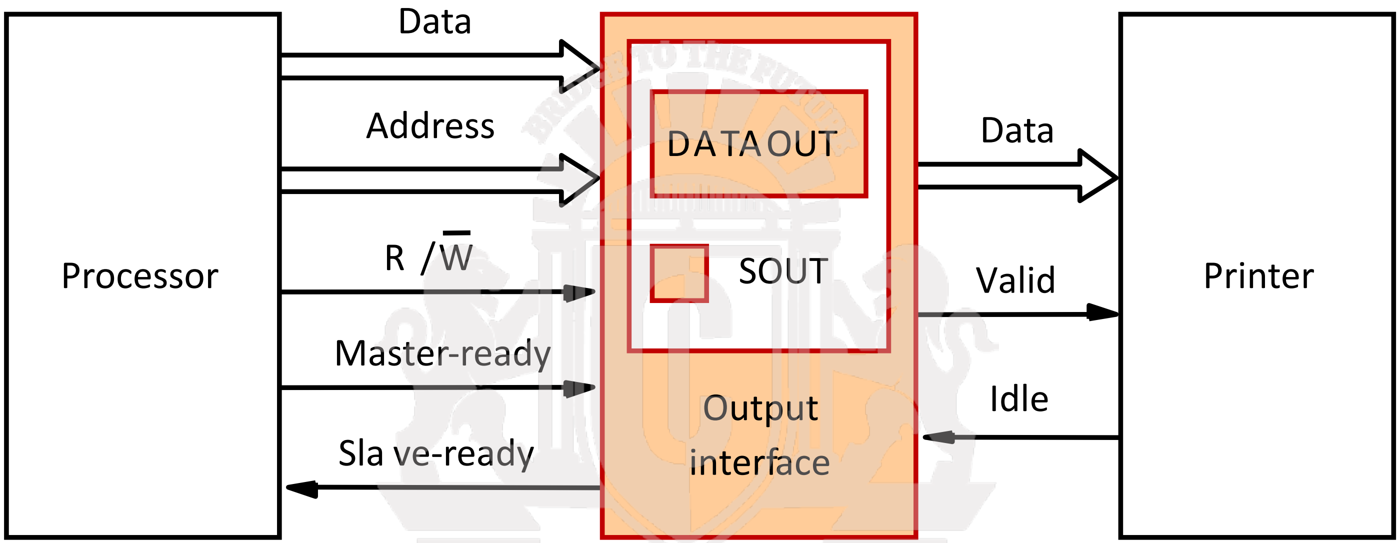
Input Interface Circuit



- Output lines of DATAIN are are connected to the data lines of the bus by means of 3 state drivers
- Drivers are turned on when the processor issues a read signal and the address selects this register.

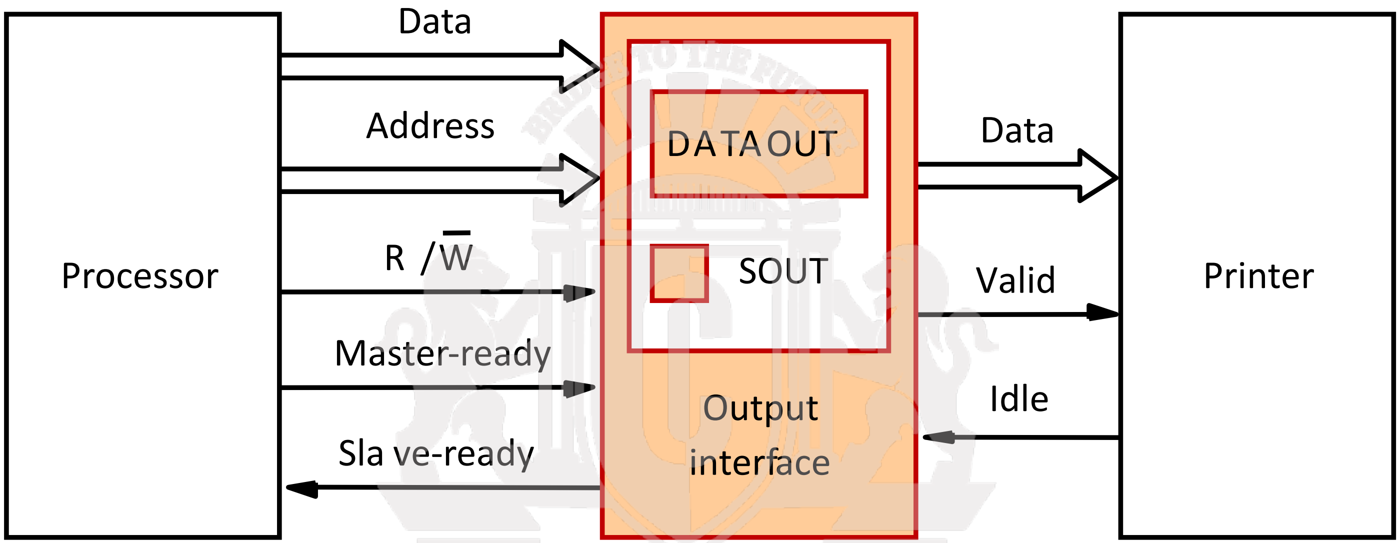
- SIN signal is generated using a status flag circuit.
- It is connected to line D_0 of the processor bus using a three-state driver.
- Address decoder selects the input interface based on bits A_1 through A_{31}.
- Bit Ao determines whether the status or data register is to be read, when Master-ready is active.
- In response, the processor activates the Slave-ready signal, when either the Read-status or Read-data is equal to 1, which depends on line A_0.

Parallel port (contd..)



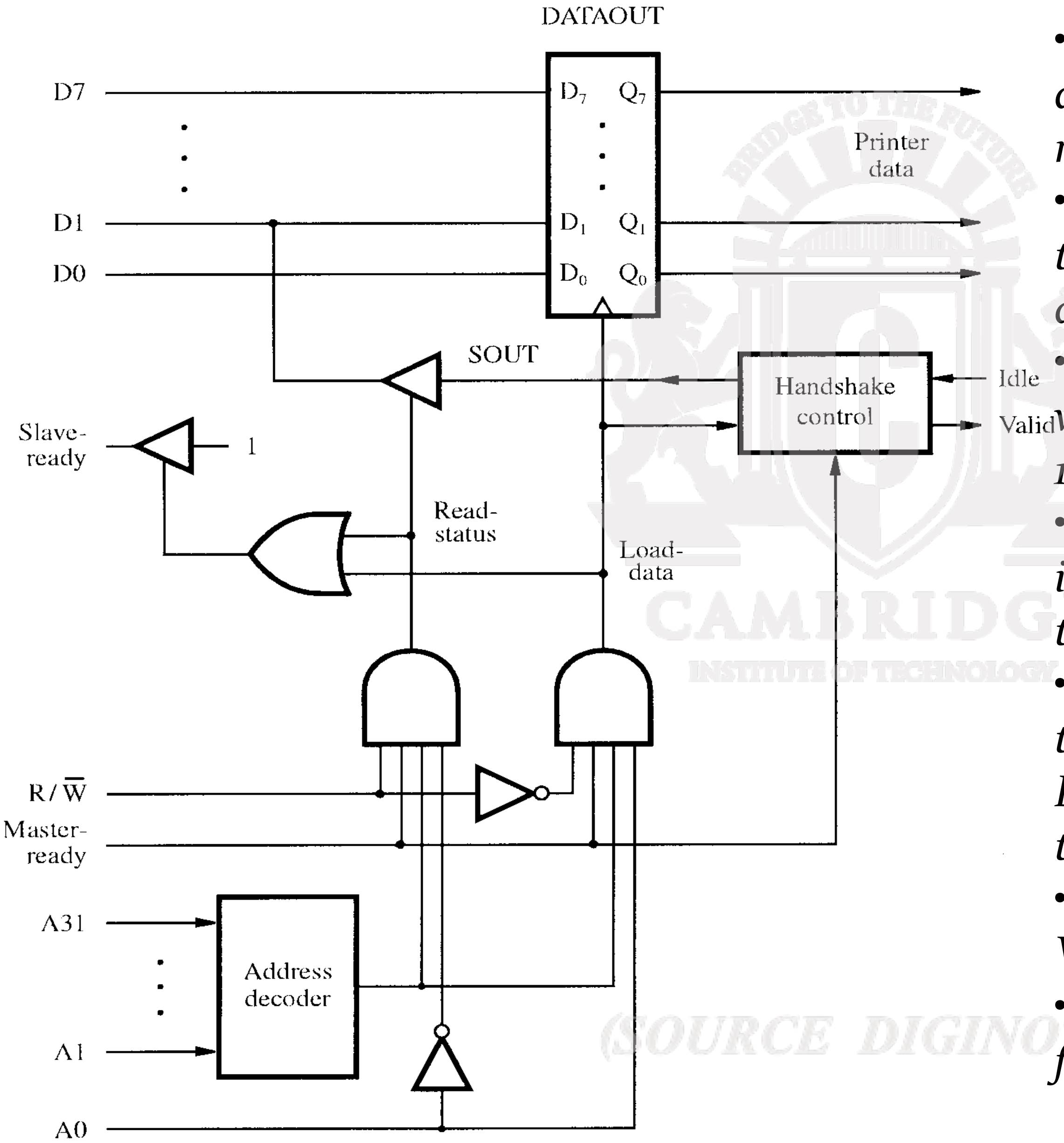
- *Printer is connected to a processor using a parallel port.*
- *Processor is 32 bits, uses memory-mapped I/O and asynchronous bus protocol.*
- *On the processor side:*
  - *Data lines.*
  - *Address lines*
  - *Control or R/W line.*
  - *Master-ready signal and*
  - *Slave-ready signal.*

Parallel port (contd..)

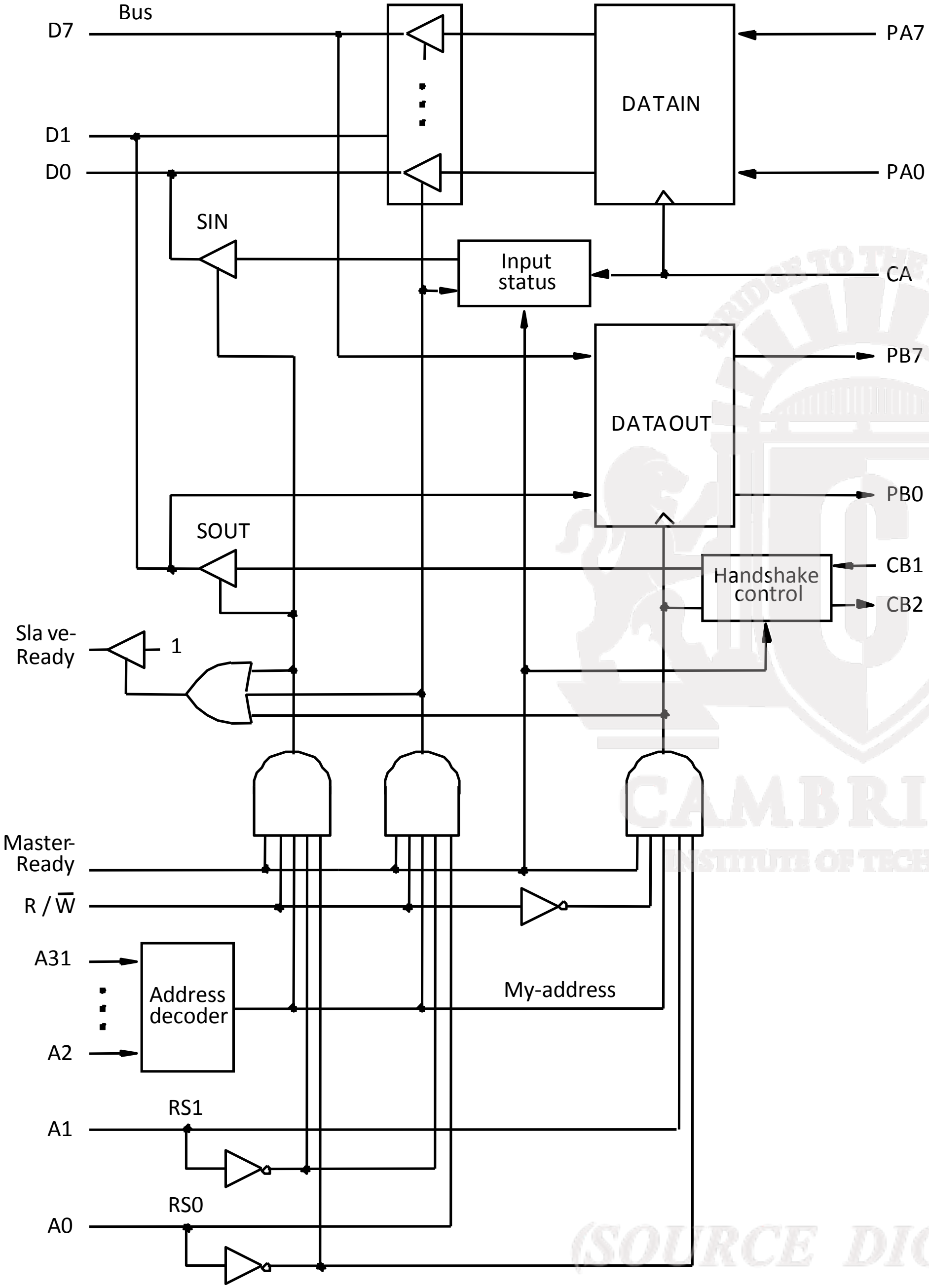


- *On the printer side:*
  - *- Idle signal line which the printer asserts when it is ready to accept a character. This causes the SOUT flag to be set to 1.*
  - *- Processor places a new character into a DATAOUT register.*
  - *- Valid signal, asserted by the interface circuit when it places a new character on the data lines.*

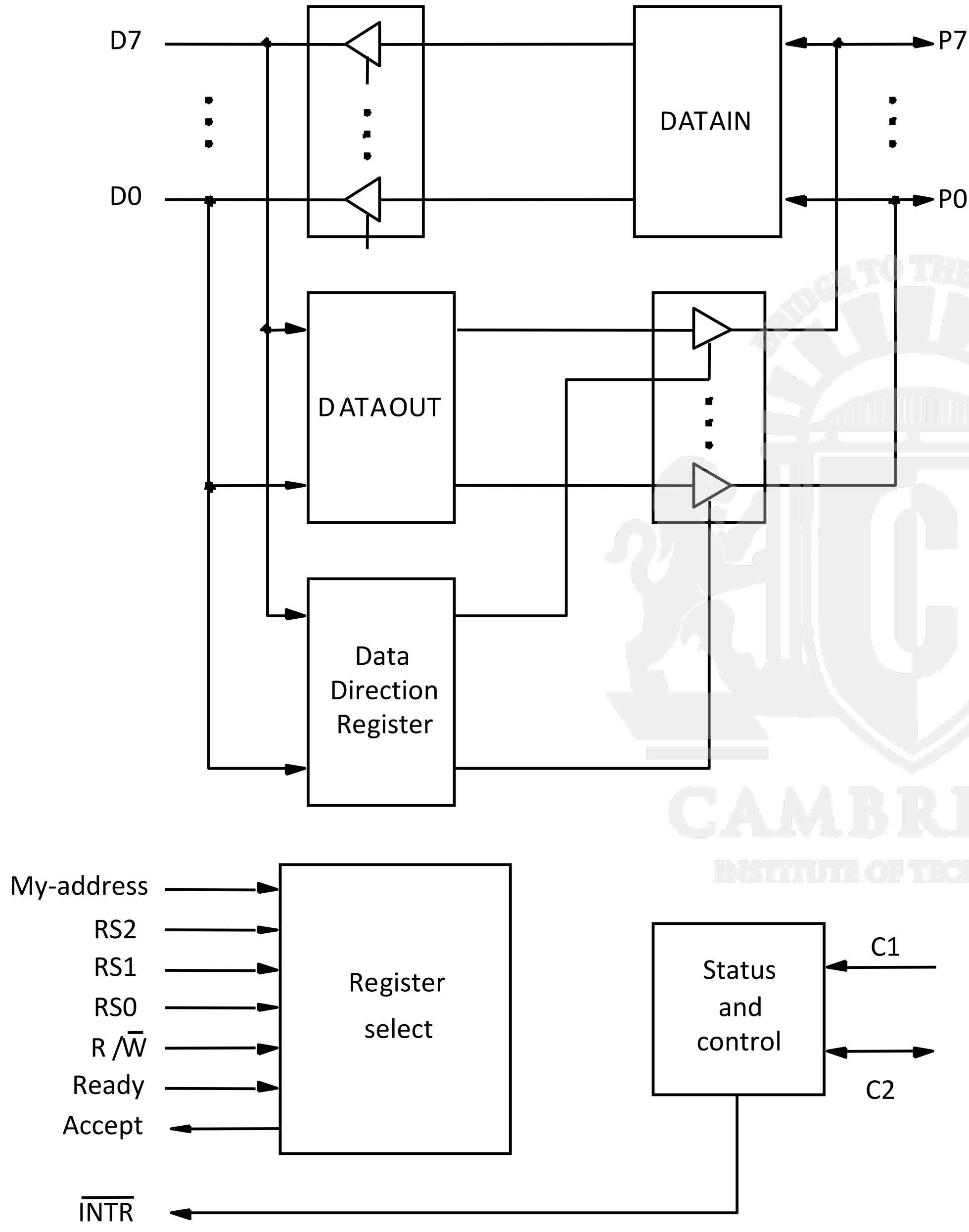
Output Interface Circuit



- *Data lines of the processor bus are connected to the DATAOUT register of the interface.*
- *The status flag SOUT is connected to the data line D1 using a three-state driver.*
- *The three-state driver is turned on, when the control Read-status line is 1.*
- *Address decoder selects the output interface using address lines A1 through A31.*
- *Address line A0 determines whether the data is to be loaded into the DATAOUT register or status flag is to be read.*
- *If the Load-data line is 1, then the Valid line is set to 1.*
- *If the Idle line is 1, then the status flag SOUT is set to 1.*



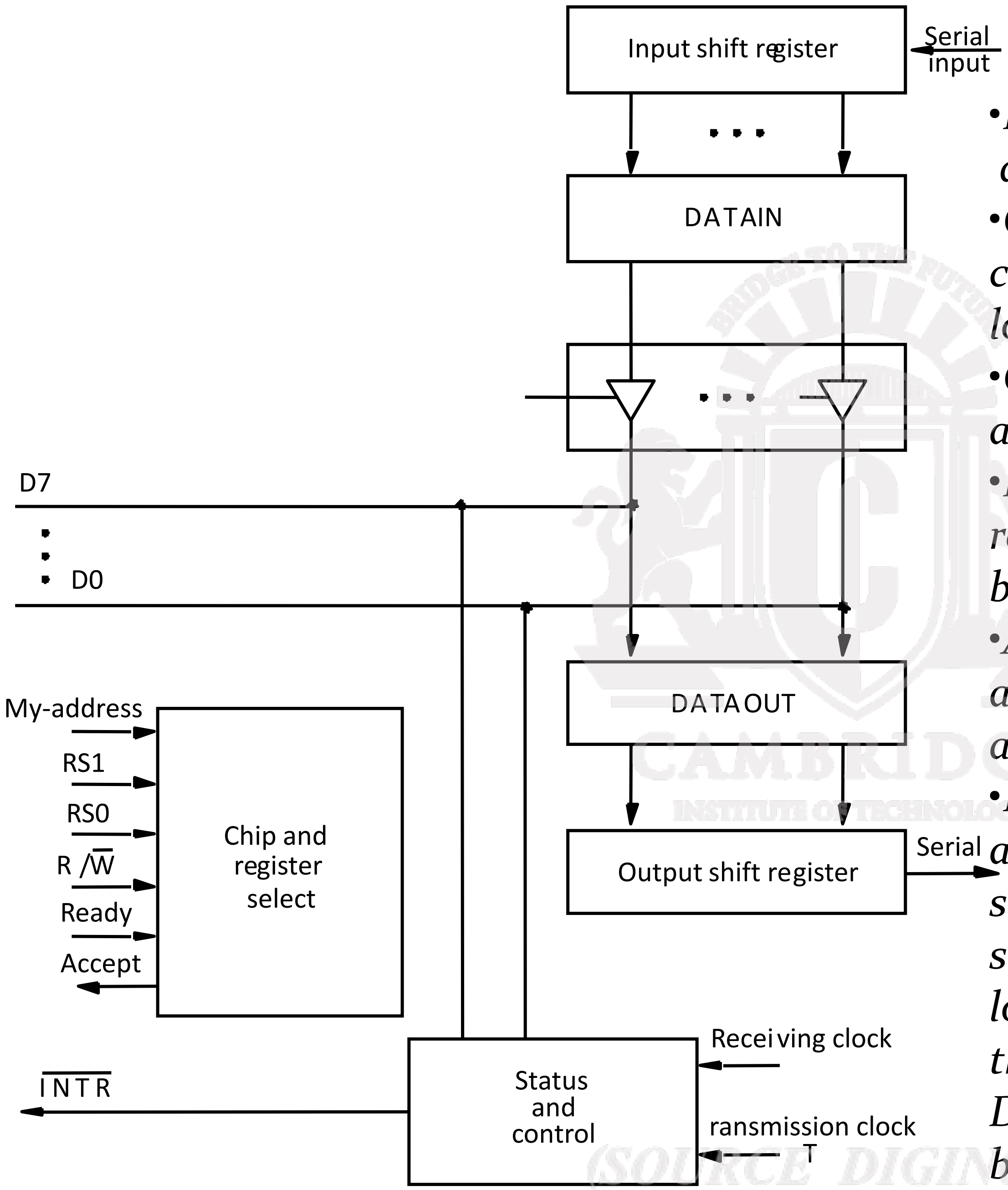
- Combined I/O interface circuit.
- Address bits A2 through A31, that is 30 bits are used to select the overall interface.
- Address bits A1 through A0, that is, 2 bits select one of the three registers, namely, DATAIN, DATAOUT, and the status register.
- Status register contains the flags SIN and SOUT in bits 0 and 1.
- Data lines PA0 through PA7 connect the input device to the DATAIN register.
- DATAOUT register connects the data lines on the processor bus to lines PB0 through PB7 which connect to the output device.
- Separate input and output data lines for connection to an I/O device.



- Data lines to I/O device are bidirectional.
- Data lines P7 through P0 can be used for both input, and output.
- In fact, some lines can be used for input & some for output depending on the pattern in the Data Direction Register (DDR).
- Processor places an 8-bit pattern into a DDR
- If a given bit position in the DDR is 1, the corresponding data line acts as an output line, otherwise it acts as an input line.
- C1 and C2 control the interaction between the interface circuit and the I/O devices.
- Ready and Accept lines are the handshake control lines on the processor bus side, and are connected to Master-ready & Slave-ready
- Input signal My-address is connected to the output of an address decoder.
- Three register select lines that allow up to 8 registers to be selected.

Serial port

- Serial port is used to connect the processor to I/O devices that require transmission of data one bit at a time.
- Serial port communicates in a bit-serial fashion on the device side and bit parallel fashion on the bus side.
  - Transformation between the parallel and serial formats is achieved with shift registers that have parallel access capability.



- *Input shift register accepts input one bit at a time from the I/O device.*
- *Once all the 8 bits are received, the contents of the input shift register are loaded in parallel into DATAIN register.*
- *Output data in the DATAOUT register are loaded into the output shift register.*
- *Bits are shifted out of the output shift register and sent out to the I/O device on bit at a time.*
- *As soon as data from the input shift reg. are loaded into DATAIN, it can start accepting another 8 bits of data.*
- *Input shift register and DATAIN register are both used at input so that the input shift register can start receiving another set of 8 bits from the input device after loading the contents to DATAIN, before the processor reads the contents of DATAIN. This is called as double-buffering.*

Serial port (contd..)

- Serial interfaces require fewer wires, and hence serial transmission is convenient for connecting devices that are physically distant from the computer.
- Speed of transmission of the data over a serial interface is known as the "bit rate".
  - Bit rate depends on the nature of the devices connected.
- In order to accommodate devices with a range of speeds, a serial interface must be able to use a range of clock speeds.
- Several standard serial interfaces have been developed:
  - Universal Asynchronous Receiver Transmitter (UART) for low-speed serial devices.
  - RS-232-C for connection to communication links.

Standard I/O interfaces

- I/O device is connected to a computer using an interface circuit.
- Do we have to design a different interface for every combination of an I/O device and a computer?
- A practical approach is to develop standard interfaces and protocols.
- A personal computer has:
  - A motherboard which houses the processor chip, main memory and some I/O interfaces.
  - A few connectors into which additional interfaces can be plugged.
- Processor bus is defined by the signals on the processor chip.
  - Devices which require high-speed connection to the processor are connected directly to this bus.

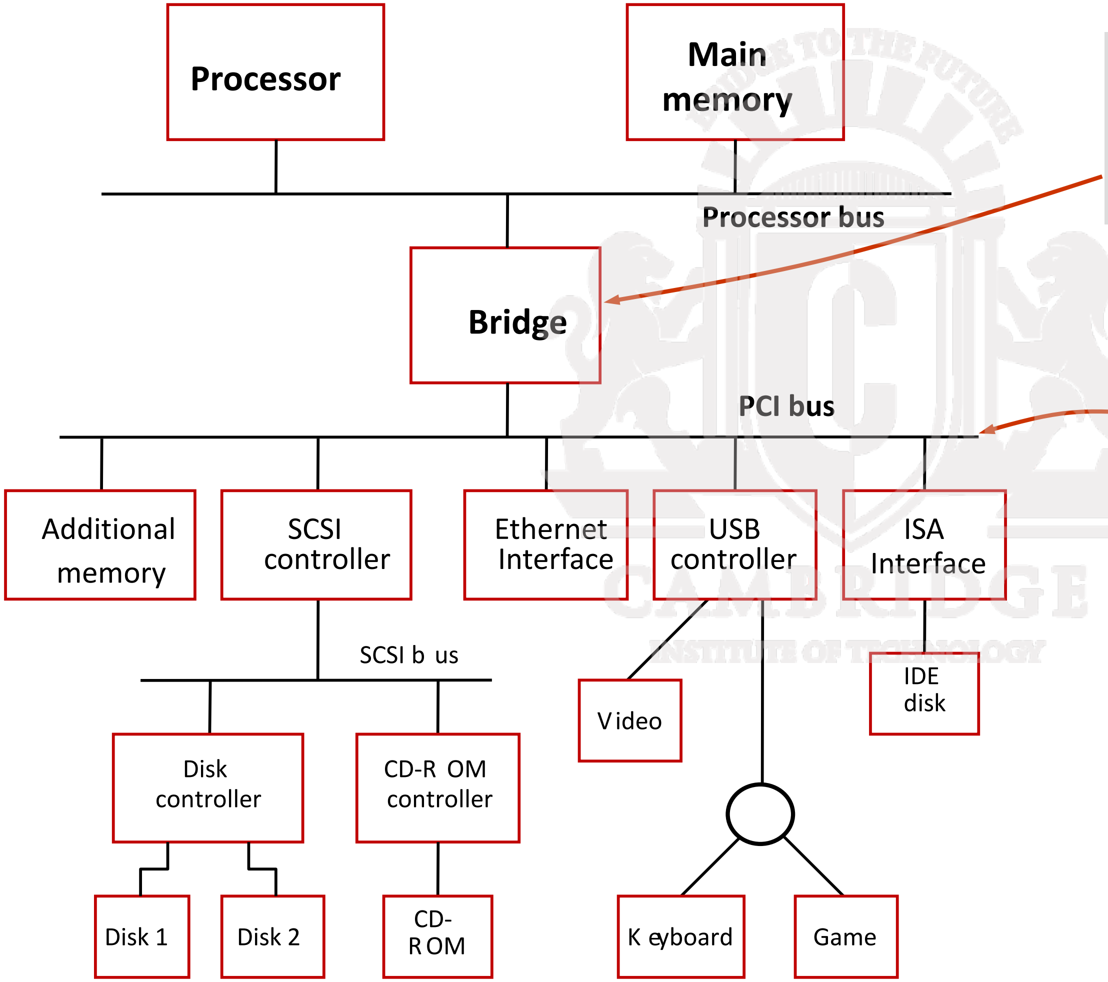
Standard I/O interfaces (contd..)

- Because of electrical reasons only a few devices can be connected directly to the processor bus.
- Motherboard usually provides another bus that can support more devices.
  - Processor bus and the other bus (called as expansion bus) are interconnected by a circuit called “bridge”.
  - Devices connected to the expansion bus experience a small delay in data transfers.
- Design of a processor bus is closely tied to the architecture of the processor.
  - No uniform standard can be defined.
- Expansion bus however can have uniform standard defined.

Standard I/O interfaces (contd..)

- A number of standards have been developed for the expansion bus.
  - Some have evolved by default.
  - For example, IBM's Industry Standard Architecture.
- Three widely used bus standards:
  - PCI (Peripheral Component Interconnect)
  - SCSI (Small Computer System Interface)
  - USB (Universal Serial Bus)

Standard I/O interfaces (contd..)



Bridge circuit translates signals and protocols from processor bus to PCI bus.

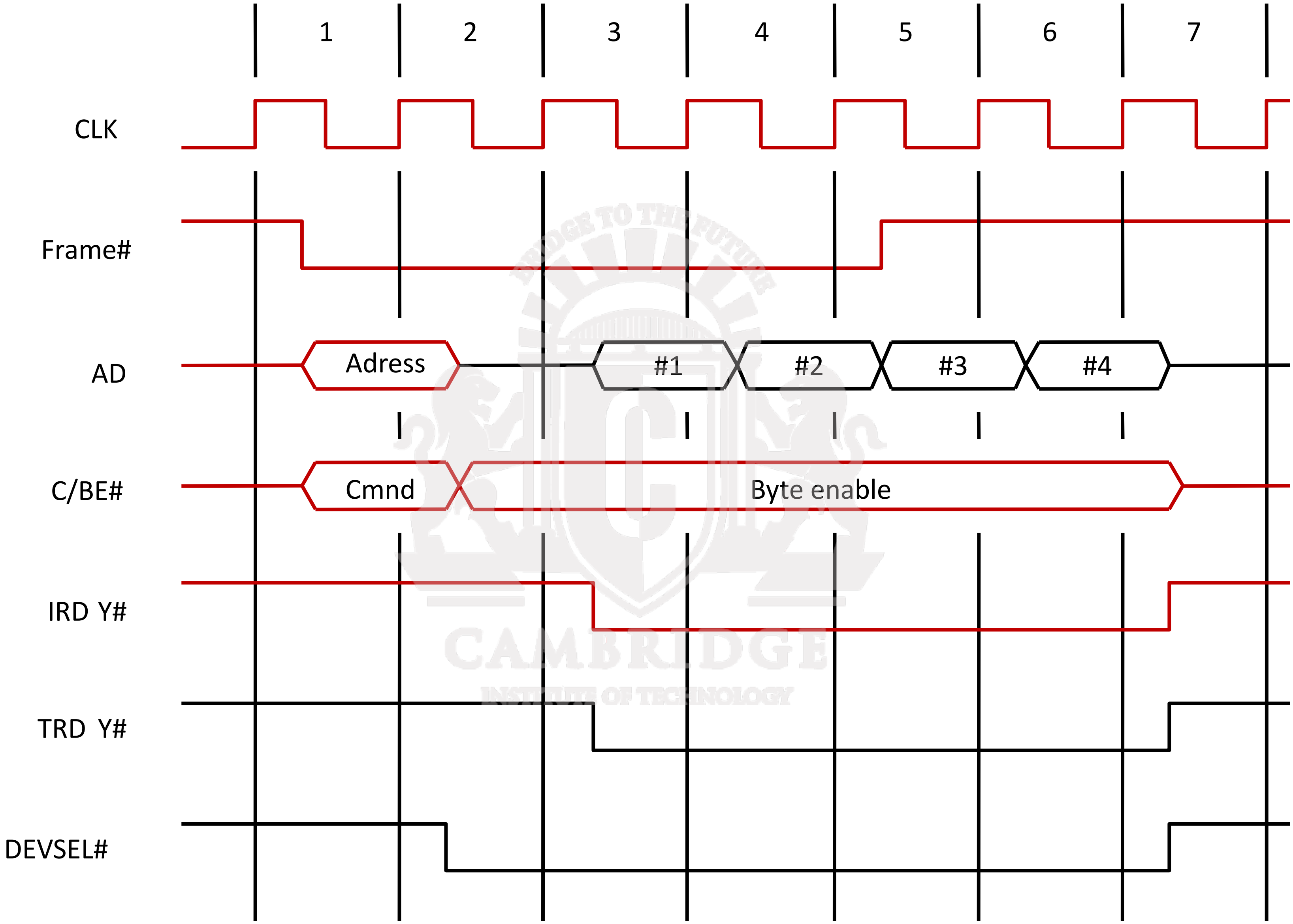
Expansion bus on the motherboard

PCI Bus

- *Peripheral Component Interconnect*
- Introduced in 1992
- Low-cost bus
- Processor independent
- Plug-and-play capability
- In today's computers, most memory transfers involve a burst of data rather than just one word. The PCI is designed primarily to support this mode of operation.
- The bus supports three independent address spaces: memory, I/O, and configuration.
- we assumed that the master maintains the address information on the bus until data transfer is completed. But, the address is needed only long enough for the slave to be selected. Thus, the address is needed on the bus for one clock cycle only, freeing the address lines to be used for sending data in subsequent clock cycles. The result is a significant cost reduction.
- A master is called an initiator in PCI terminology. The addressed device that responds to read and write commands is called a target.

Data transfer signals on the PCI bus.

Name	Function
CLK	A 33-MHz or 66-MHz clock.
FRAME#	Sent by the initiator to indicate the duration of a transaction.
AD	32 address/data lines, which may be optionally increased to 64.
C/BE#	4 command/byte-enable lines (8 for a 64-bit bus).
IRDY#, TRDY#	Initiator-ready and Target-ready signals.
DEVSEL#	A response from the device indicating that it has recognized its address and is ready for a data transfer transaction.
IDSEL#	Initialization Device Select.



A read operation on the PCI bus

Device Configuration

- When an I/O device is connected to a computer, several actions are needed to configure both the device and the software that communicates with it.
- PCI incorporates in each I/O device interface a small configuration ROM memory that stores information about that device.
- The configuration ROMs of all devices are accessible in the configuration address space. The PCI initialization software reads these ROMs and determines whether the device is a printer, a keyboard, an Ethernet interface, or a disk controller. It can further learn bout various device options and characteristics.
- Devices are assigned addresses during the initialization process.
- This means that during the bus configuration operation, devices cannot be accessed based on their address, as they have not yet been assigned one.
- Hence, the configuration address space uses a different mechanism. Each device has an input signal called Initialization Device Select, IDSEL#
- Electrical characteristics:
  - PCI bus has been defined for operation with either a 5 or 3.3 V power supply

SCSI Bus

- The acronym SCSI stands for Small Computer System Interface.
- It refers to a standard bus defined by the American National Standards Institute (ANSI) under the designation X3.131 .
- In the original specifications of the standard, devices such as disks are connected to a computer via a 50-wire cable, which can be up to 25 meters in length and can transfer data at rates up to 5 megabytes/s.
- The SCSI bus standard has undergone many revisions, and its data transfer capability has increased very rapidly, almost doubling every two years.
- SCSI-2 and SCSI-3 have been defined, and each has several options.
- Because of various options SCSI connector may have 50, 68 or 80 pins.

SCSI Bus (Contd.,)

- Devices connected to the SCSI bus are not part of the address space of the processor
- The SCSI bus is connected to the processor bus through a SCSI controller. This controller uses DMA to transfer data packets from the main memory to the device, or vice versa.
- A packet may contain a block of data, commands from the processor to the device, or status information about the device.
- A controller connected to a SCSI bus is one of two types – an initiator or a target.
- An initiator has the ability to select a particular target and to send commands specifying the operations to be performed. The disk controller operates as a target. It carries out the commands it receives from the initiator.
- The initiator establishes a logical connection with the intended target.
- Once this connection has been established, it can be suspended and restored as needed to transfer commands and bursts of data.
- While a particular connection is suspended, other device can use the bus to transfer information.
- This ability to overlap data transfer requests is one of the key features of the SCSI bus that leads to its high performance.

SCSI Bus (Contd.,)

- Data transfers on the SCSI bus are always controlled by the target controller.
- To send a command to a target, an initiator requests control of the bus and, after winning arbitration, selects the controller it wants to communicate with and hands control of the bus over to it.
- Then the controller starts a data transfer operation to receive a command from the initiator.

SCSI Bus (Contd.,)

- Assume that processor needs to read block of data from a disk drive and that data are stored in disk sectors that are not contiguous.
- The processor sends a command to the SCSI controller, which causes the following sequence of events to take place:

1. The SCSI controller, acting as an initiator, contends for control of the bus.
2. When the initiator wins the arbitration process, it selects the target controller and hands over control of the bus to it.
3. The target starts an output operation (from initiator to target); in response to this, the initiator sends a command specifying the required read operation.
4. The target, realizing that it first needs to perform a disk seek operation, sends a message to the initiator indicating that it will temporarily suspend the connection between them. Then it releases the bus.
5. The target controller sends a command to the disk drive to move the read head to the first sector involved in the requested read operation. Then, it reads the data stored in that sector and stores them in a data buffer. When it is ready to begin transferring data to the initiator, the target requests control of the bus. After it wins arbitration, it reselects the initiator controller, thus restoring the suspended connection.

SCSI Bus (Contd.,)

6. The target transfers the contents of the data buffer to the initiator and then suspends the connection again
7. The target controller sends a command to the disk drive to perform another seek operation. Then, it transfers the contents of the second disk sector to the initiator as before. At the end of this transfers, the logical connection between the two controllers is terminated.
8. As the initiator controller receives the data, it stores them into the main memory using the DMA approach.
9. The SCSI controller sends as interrupt to the processor to inform it that the requested operation has been completed

Operation of SCSI bus from H/W point of view

Category	Name	Function
Data	– DB(0) to – DB(7)	Data lines: Carry one byte of information during the information transfer phase and identify device during arbitration, selection and reselection phases
	– DB(P)	Parity bit for the data bus
Phase	– BSY	Busy: Asserted when the bus is not free
	–SEL	Selection: Asserted during selection and reselection
Information type	– C/D	Control/Data: Asserted during transfer of control information (command, status or message)
	– MSG	Message: indicates that the information being transferred is a message

Table 4. The SCSI bus signals.

Table 4. The SCSI bus signals.(*cont*.)

Category	Name	Function
Handshake	‾ REQ	Request: Asserted by a target to request a data transfer cycle
	‾ ACK	Acknowledge: Asserted by the initiator when it has completed a data transfer operation
Direction of transfer	‾ I/O	Input/Output: Asserted to indicate an input operation (relative to the initiator)
Other	‾ ATN	Attention: Asserted by an initiator when it wishes to send a message to a target
	‾ RST	Reset: Causes all device controls to disconnect from the bus and assume their start-upstate

Main Phases involved

- Arbitration
  - A controller requests the bus by asserting BSY and by asserting it's associated data line
  - When BSY becomes active, all controllers that are requesting bus examine data lines
- Selection
  - Controller that won arbitration selects target by asserting SEL and data line of target. After that initiator releases BSY line.
  - Target responds by asserting BSY line
  - Target controller will have control on the bus from then
- Information Transfer
  - Handshaking signals are used between initiator and target
  - At the end target releases BSY line
- Reselection

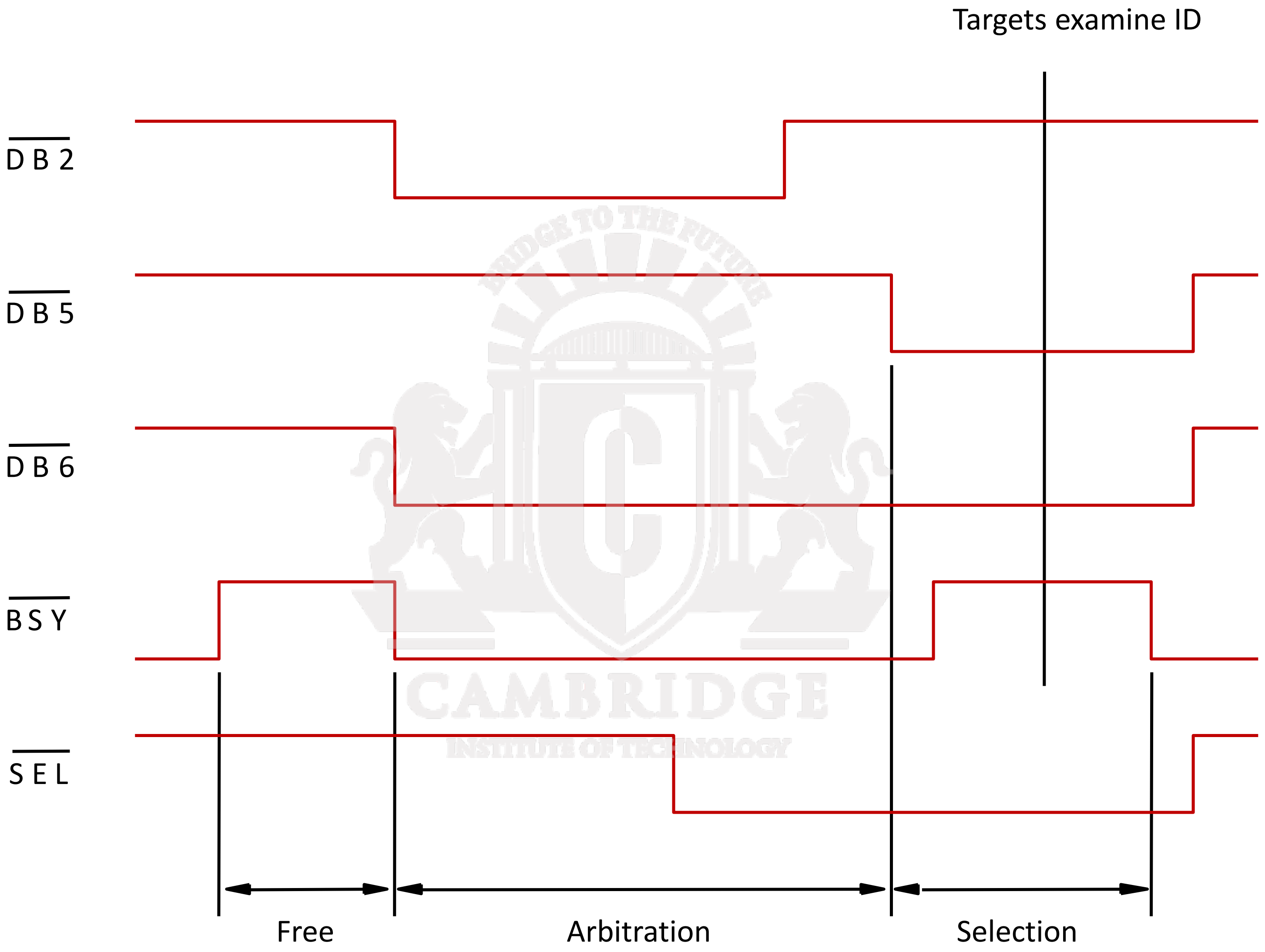
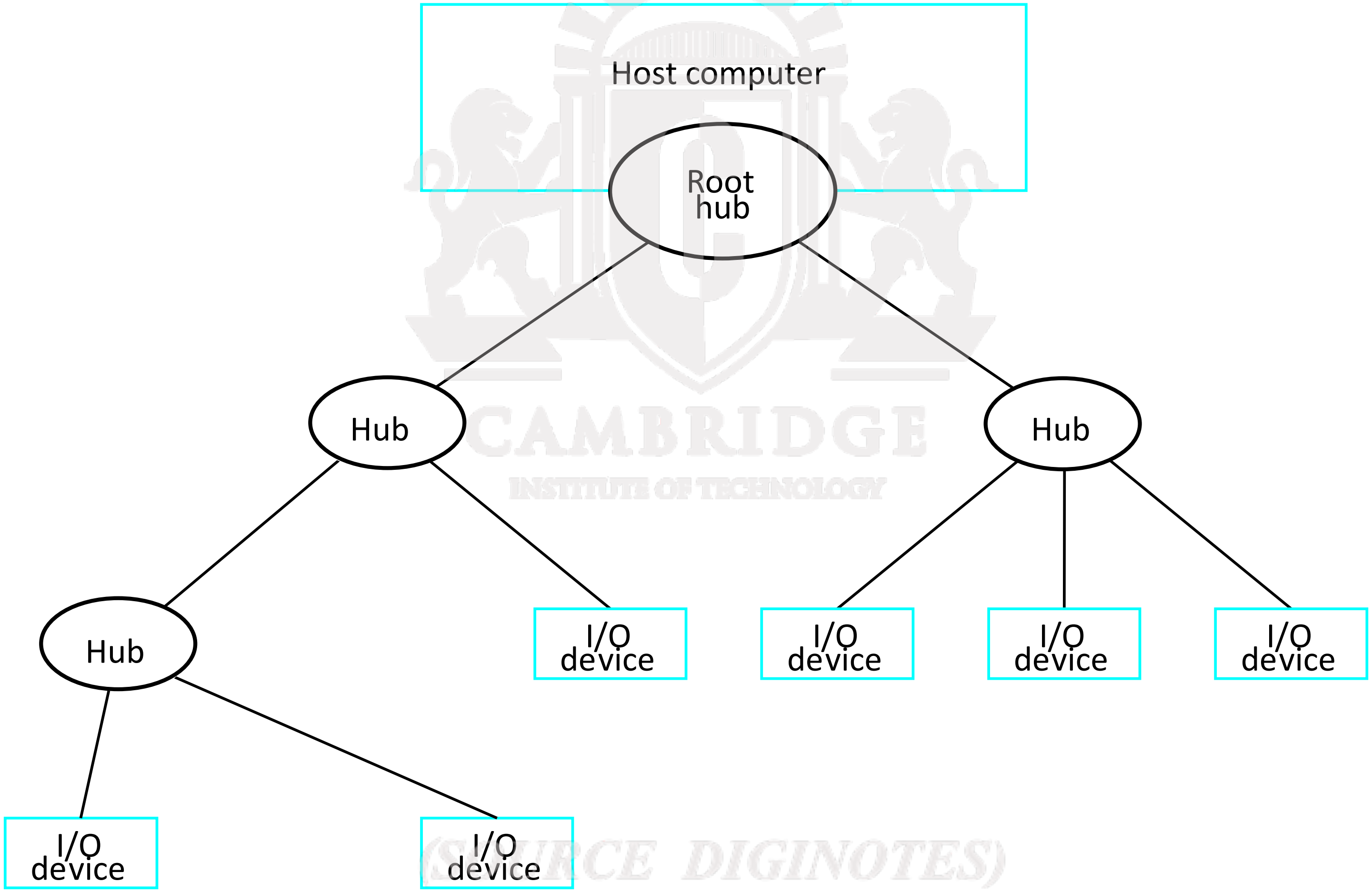


Figure 42. Arbitration and selection on the SCSI bus.
Device 6 wins arbitration and selects device 2.

USB

- Universal Serial Bus (USB) is an industry standard developed through a collaborative effort of several computer and communication companies, including Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, Nortel Networks, and Philips.
- Speed
    - Low-speed(1.5 Mb/s)
    - Full-speed(12 Mb/s)
    - High-speed(480 Mb/s)
- Port Limitation
- Device Characteristics
- Plug-and-play

Universal Serial Bus tree structure



Universal Serial Bus tree structure

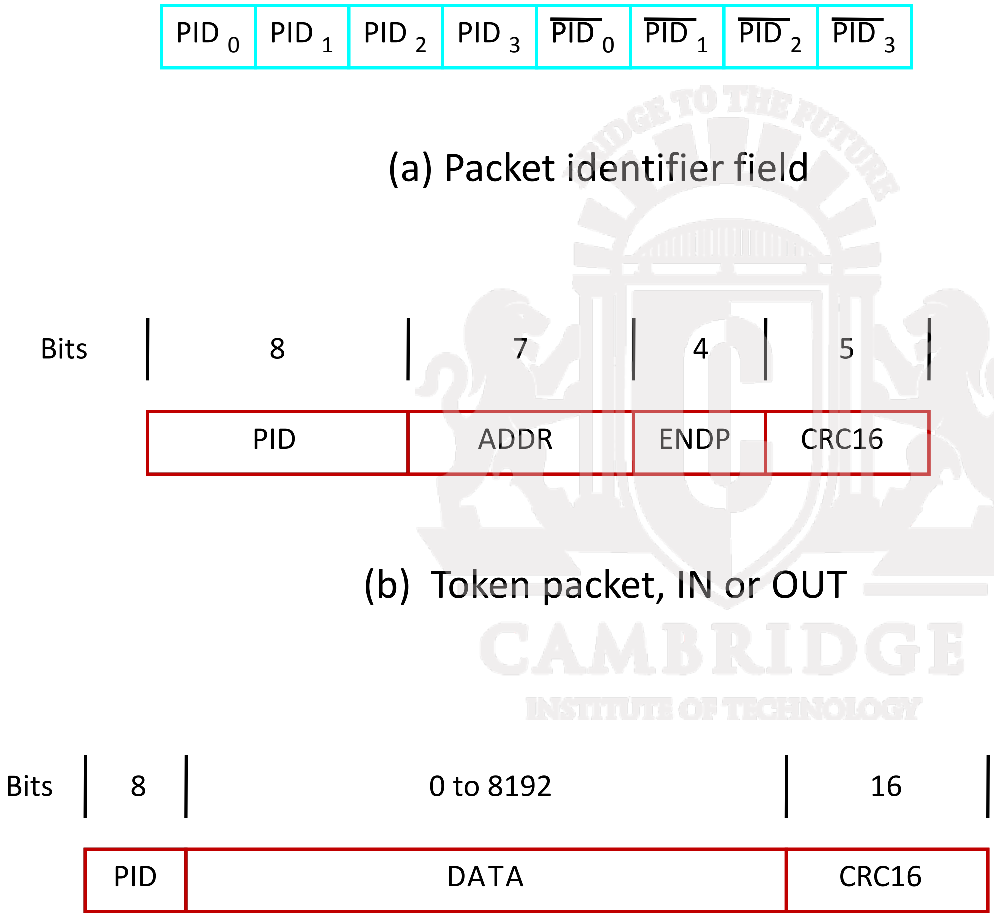
- To accommodate a large number of devices that can be added or removed at any time, the USB has the tree structure as shown in the figure.
- Each node of the tree has a device called a hub, which acts as an intermediate control point between the host and the I/O devices. At the root of the tree, a root hub connects the entire tree to the host computer. The leaves of the tree are the I/O devices being served (for example, keyboard, Internet connection, speaker, or digital TV)
- In normal operation, a hub copies a message that it receives from its upstream connection to all its downstream ports. As a result, a message sent by the host computer is broadcast to all I/O devices, but only the addressed device will respond to that message. However, a message from an I/O device is sent only upstream towards the root of the tree and is not seen by other devices. Hence, the USB enables the host to communicate with the I/O devices, but it does not enable these devices to communicate with each other.

Addressing

- When a USB is connected to a host computer, its root hub is attached to the processor bus, where it appears as a single device. The host software communicates with individual devices attached to the USB by sending packets of information, which the root hub forwards to the appropriate device in the USB tree.
- Each device on the USB, whether it is a hub or an I/O device, is assigned a 7-bit address. This address is local to the USB tree and is not related in any way to the addresses used on the processor bus.
- A hub may have any number of devices or other hubs connected to it, and addresses are assigned arbitrarily. When a device is first connected to a hub, or when it is powered on, it has the address 0. The hardware of the hub to which this device is connected is capable of detecting that the device has been connected, and it records this fact as part of its own status information. Periodically, the host polls each hub to collect status information and learn about new devices that may have been added or disconnected.
- When the host is informed that a new device has been connected, it uses a sequence of commands to send a reset signal on the corresponding hub port, read information from the device about its capabilities, send configuration information to the device, and assign the device a unique USB address. Once this sequence is completed the device begins normal operation and responds only to the new address.

USB Protocols

- All information transferred over the USB is organized in packets, where a packet consists of one or more bytes of information. There are many types of packets that perform a variety of control functions.
- The information transferred on the USB can be divided into two broad categories: control and data.
  - Control packets perform such tasks as addressing a device to initiate data transfer, acknowledging that data have been received correctly, or indicating an error.
  - Data packets carry information that is delivered to a device.
- A packet consists of one or more fields containing different kinds of information. The first field of any packet is called the packet identifier, PID, which identifies the type of that packet.
- They are transmitted twice. The first time they are sent with their true values, and the second time with each bit complemented
- The four PID bits identify one of 16 different packet types. Some control packets, such as ACK (Acknowledge), consist only of the PID byte.



(a) Packet identifier field

(b) Token packet, IN or OUT

(c) Data packet

Figure 45. USB packet format.

Control packets used for controlling data transfer operations are called token packets.

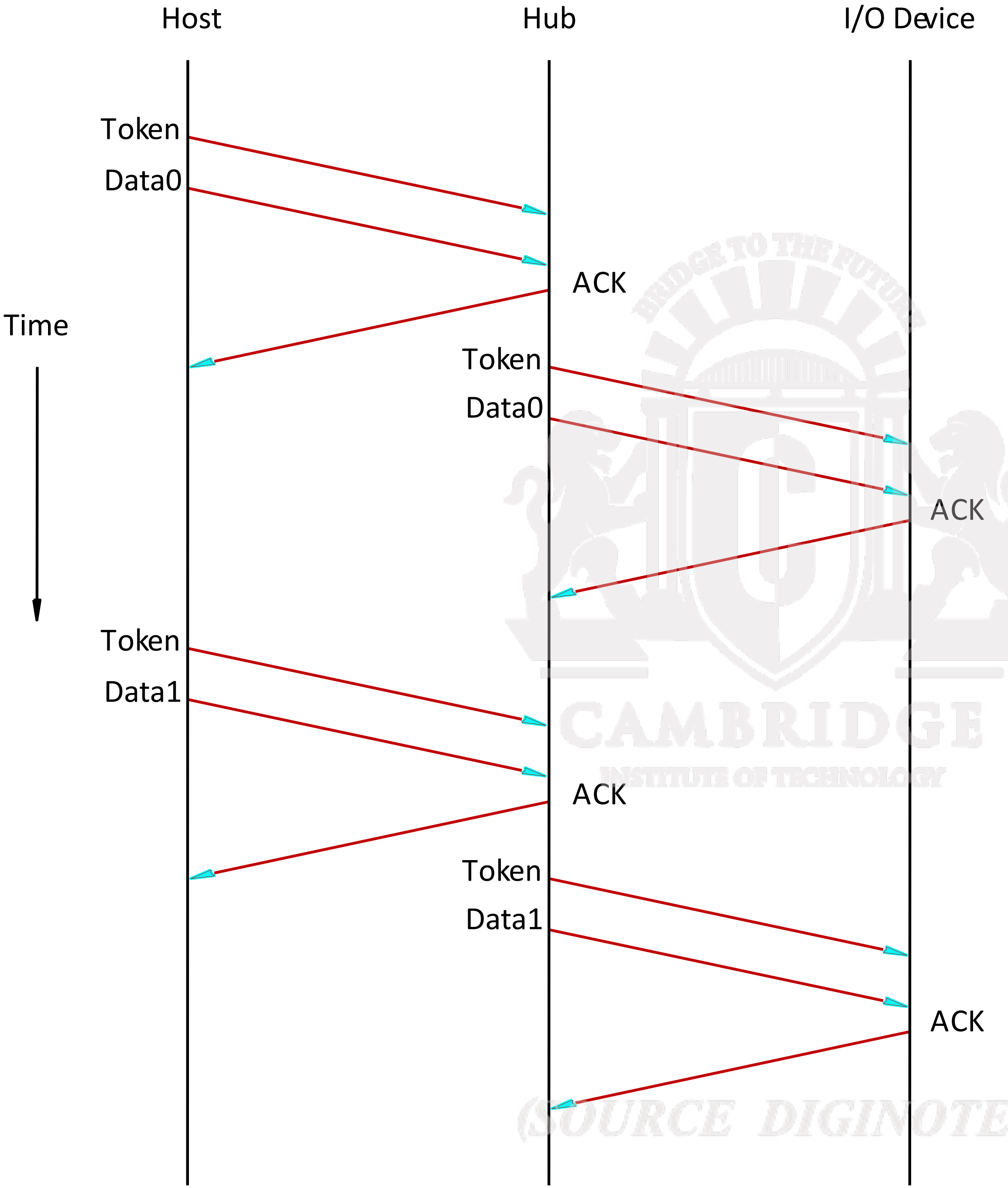


Figure: An output transfer

Isochronous Traffic on USB

- One of the key objectives of the USB is to support the transfer of isochronous data.
- Devices that generates or receives isochronous data require a time reference to control the sampling process.
- To provide this reference. Transmission over the USB is divided into frames of equal length.
- A frame is 1ms long for low-and full-speed data.
- The root hub generates a Start of Frame control packet (SOF) precisely once every 1 ms to mark the beginning of a new frame.
- The arrival of an SOF packet at any device constitutes a regular clock signal that the device can use for its own purposes.
- To assist devices that may need longer periods of time, the SOF packet carries an 11-bit frame number.
- Following each SOF packet, the host carries out input and output transfers for isochronous devices. This means that each device will have an opportunity for an input or output transfer once every 1 ms.

Electrical Characteristics

- The cables used for USB connections consist of four wires.
- Two are used to carry power, +5V and Ground.
  - Thus, a hub or an I/O device may be powered directly from the bus, or it may have its own external power connection.
- The other two wires are used to carry data.
- Different signaling schemes are used for different speeds of transmission.
  - At low speed, 1s and 0s are transmitted by sending a high voltage state (5V) on one or the other o the two signal wires. For high-speed links, differential transmission is used.

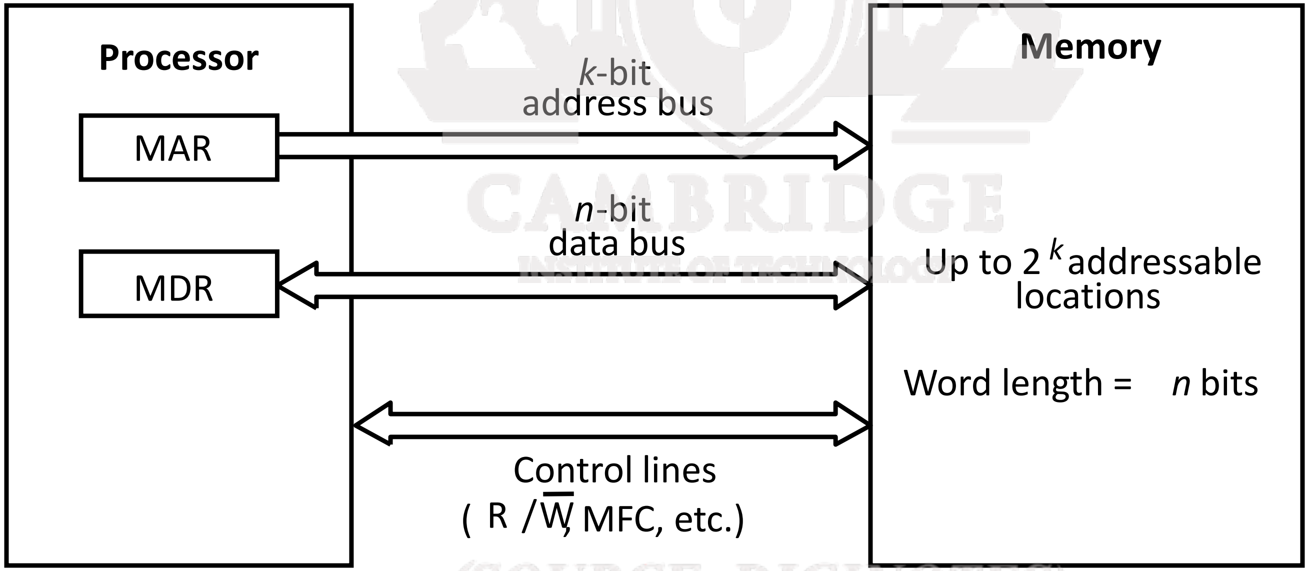
(SOURCE DIGINOTES)

Module 3:The Memory System

Courtesy: Text book: Carl Hamacher 5th Edition

Some basic concepts

- Maximum size of the Main Memory
- byte-addressable
- CPU-Main Memory Connection



Some basic concepts(Contd.,)

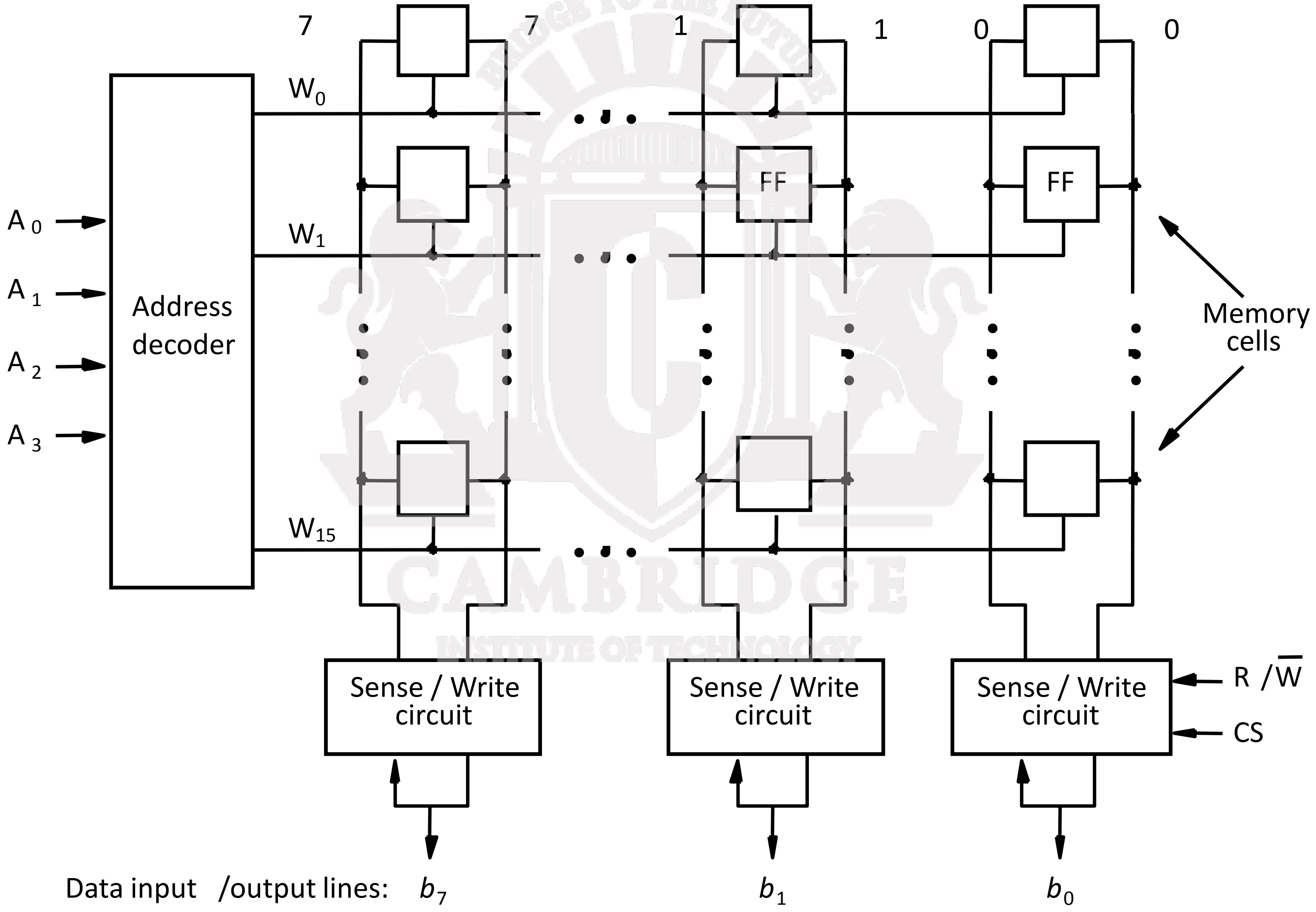
- Measures for the speed of a memory:
  - **memory access time.**
  - **memory cycle time.**
- An important design issue is to provide a computer system with as **large** and **fast** a memory as possible, within a given cost target.
- Several techniques to increase the effective size and speed of the memory:
  - **Cache memory (to increase the effective speed).**
  - **Virtual memory (to increase the effective size).**

Internal organization of memory chips

- **Each memory cell can hold one bit of information.**
- **Memory cells are organized in the form of an array.**
- **One row is one memory word.**
- **All cells of a row are connected to a common line, known as the "word line".**
- **Word line is connected to the address decoder.**
- **Sense/write circuits are connected to the data input/output lines of the memory chip.**

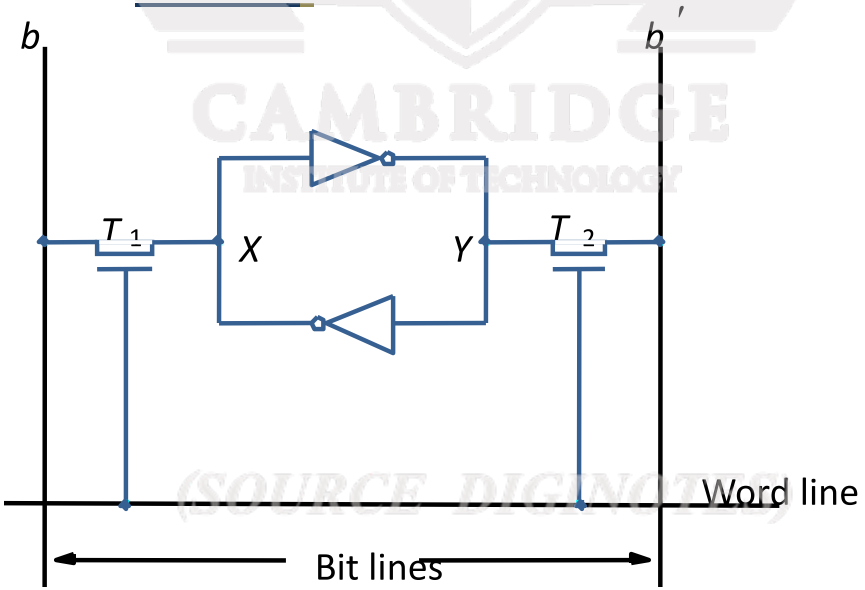
(SOURCE DIGINOTES)

Internal organization of memory chips (Contd.,)



SRAM Cell

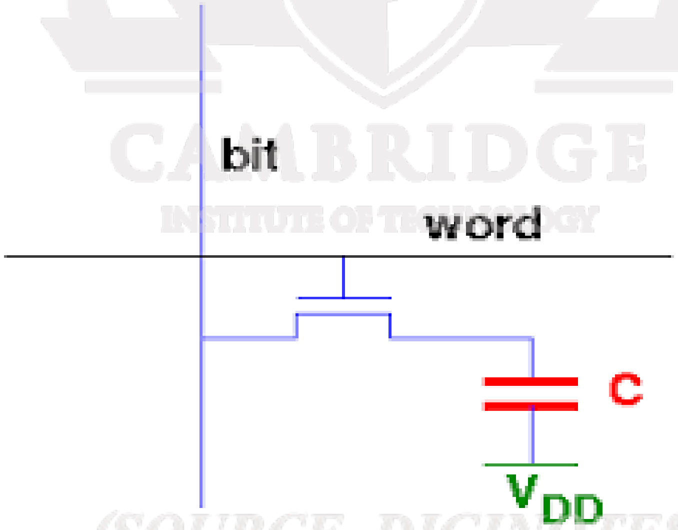
- **Two transistor inverters are cross connected to implement a basic flip-flop.**
- **The cell is connected to one word line and two bits lines by transistors T1 and T2**
- **When word line is at ground level, the transistors are turned off and the latch retains its state**
- **Read operation: In order to read state of SRAM cell, the word line is activated to close switches T1 and T2. Sense/Write circuits at the bottom monitor the state of b and b’**



DRAM Cell

- A single DRAM cell is shown in figure having a capacitor C and a Transistor T.
- To store information in this cell ,transistor T is turned on and an appropriate voltage is applied to the bit line and the capacitor gets charged. after the transistor is turned off, the capacitor begins to discharge.

- Hence the information can be retrieved correctly only if it is read before the charge on the capacitor drops below some threshold value.

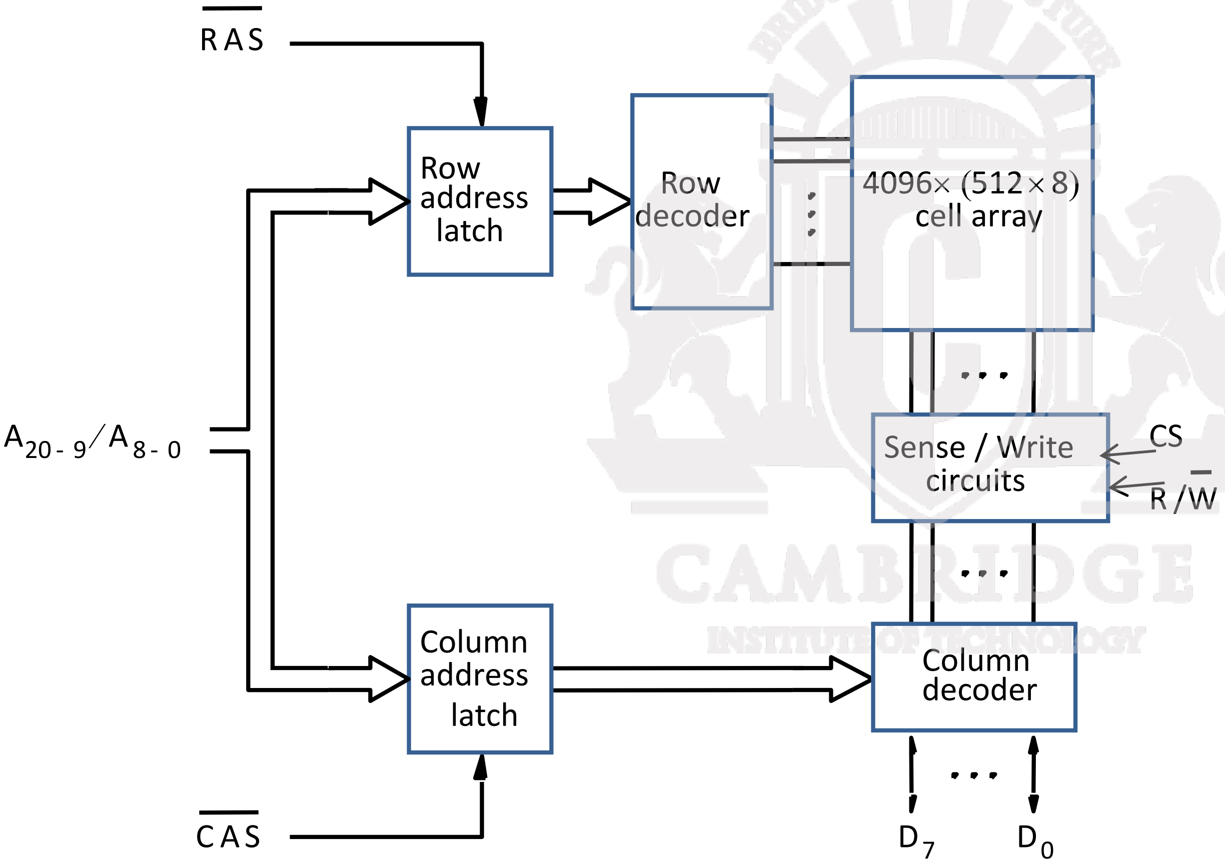


(SOURCE DIGINOTES)

Asynchronous DRAMs

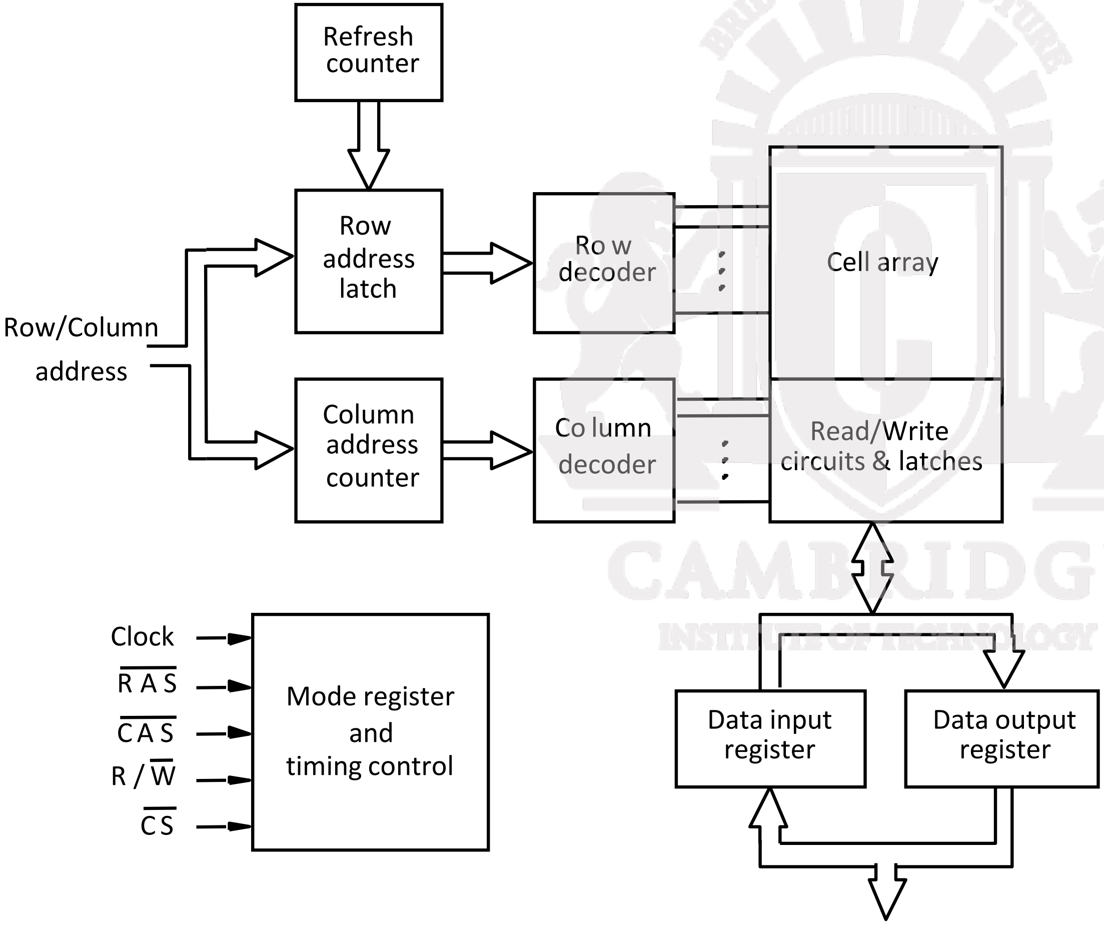
- **Static RAMs (SRAMs):**
  - Consist of circuits that are capable of retaining their state as long as the power is applied.
  - Volatile memories, because their contents are lost when power is interrupted.
  - Access times of static RAMs are in the range of few nanoseconds.
  - However, the cost is usually high.
- **Dynamic RAMs (DRAMs):**
  - Do not retain their state indefinitely.
  - Contents must be periodically refreshed.
  - Contents may be refreshed while accessing them for reading.

Asynchronous DRAMs



- Each row can store 512 bytes. 12 bits to select a row, and 9 bits to select a group in a row. Total of 21 bits.
- First apply the row address, RAS signal latches the row address. Then apply the column address, CAS signal latches the address.
- Timing of the memory unit is controlled by a specialized unit which generates RAS and CAS.
- This is asynchronous DRAM

Synchronous DRAMs



(SOURCE DIGINOTES)

Synchronous DRAMs

- Operation is directly synchronized with processor clock signal.
- The outputs of the sense circuits are connected to a latch.
- During a Read operation, the contents of the cells in a row are loaded onto the latches.
- During a refresh operation, the contents of the cells are refreshed without changing the contents of the latches.
- Data held in the latches correspond to the selected columns are transferred to the output.
- For a burst mode of operation, successive columns are selected using column address counter and clock.**CAS signal** need not be generated externally. A new data is placed during raising edge of the clock

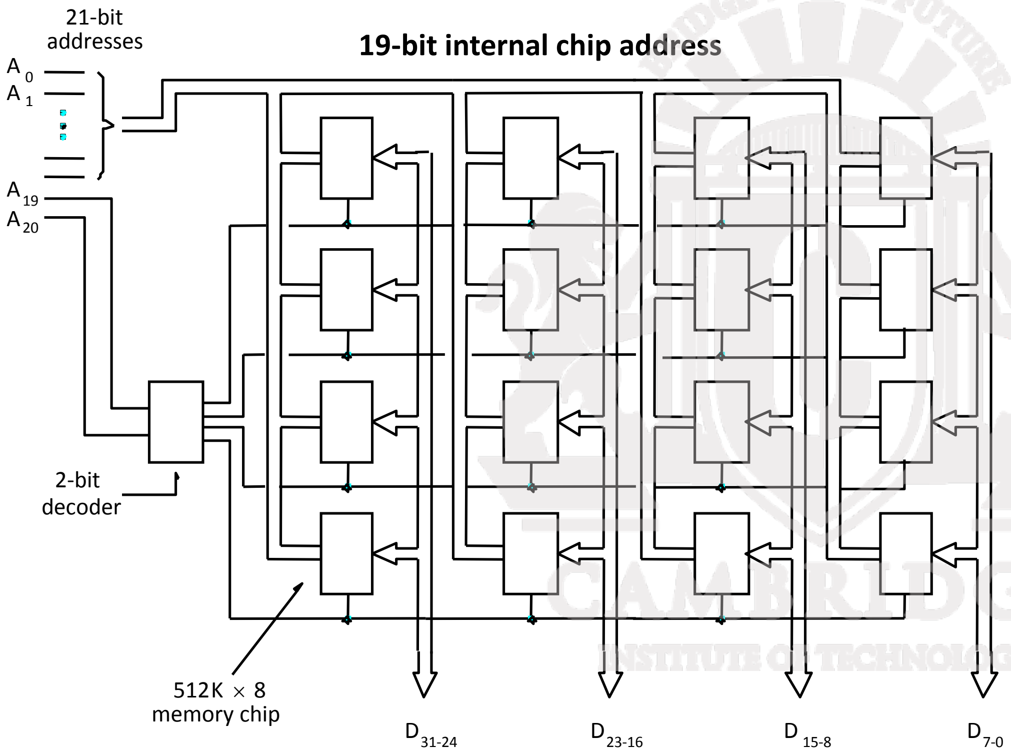
Fast Page Mode

- Suppose if we want to access the consecutive bytes in the **selected row**.
- This can be done without having to reselect the row.
  - **Add a latch at the output of the sense circuits in each row.**
  - **All the latches are loaded when the row is selected.**
  - **Different column addresses can be applied to select and place different bytes on the data lines.**
- Consecutive sequence of column addresses can be applied under the **control signal CAS**, without reselecting the row.
  - **Allows a block of data to be transferred at a much faster rate than random accesses.**
  - **A small collection/group of bytes is usually referred to as a block.**
- This transfer capability is referred to as the **fast page mode** feature.

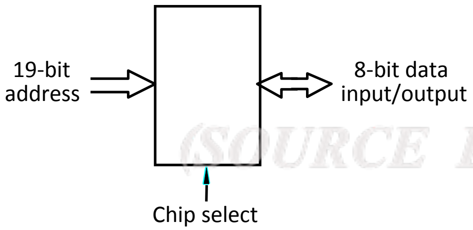
Latency, Bandwidth, and DDRSDRAMs

- **Memory latency** is the time it takes to transfer a word of data to or from memory
- **Memory bandwidth** is the number of bits or bytes that can be transferred in one second.

Static memories



512K × 8 memory chip



Implement a memory unit of 2M words of 32 bits each.
Use 512x8 static memory chips.
Each column consists of 4 chips.
Each chip implements one byte position.
A chip is selected by setting its chip select control line to 1.
Selected chip places its data on the data output line, outputs of other chips are in high impedance state.
21 bits to address a 32-bit word.
High order 2 bits are needed to select the row, by activating the four Chip Select signals.
19 bits are used to access specific byte locations inside the selected chip.

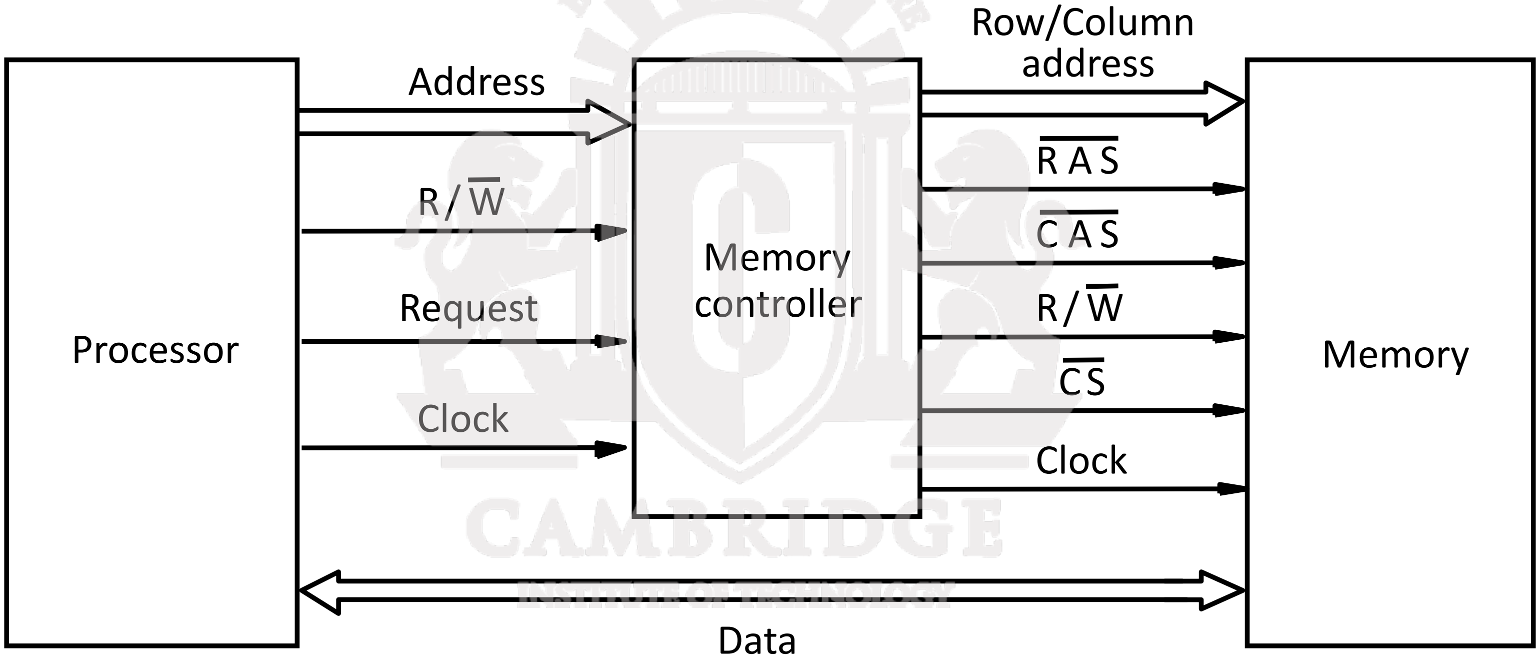
Dynamic memories

- Large dynamic memory systems can be implemented using DRAM chips in a similar way to static memory systems.
- Placing large memory systems directly on the motherboard will occupy a large amount of space.
  - Also, this arrangement is inflexible since the memory system cannot be expanded easily.
- Packaging considerations have led to the development of larger memory units known as **SIMMs (Single In-line Memory Modules)** and **DIMMs (Dual In-line Memory Modules).**
- Memory modules are an assembly of memory chips on a small board that plugs vertically onto a single socket on the motherboard.
  - Occupy less space on the motherboard.
  - Allows for easy expansion by replacement.

Memory controller

- Recall that in a dynamic memory chip, to reduce the number of pins, multiplexed addresses are used.
- Address is divided into two parts:
  - High-order address bits select a row in the array.
  - They are provided first, and latched using RAS signal.
  - Low-order address bits select a column in the row.
  - They are provided later, and latched using CAS signal.
- However, a processor issues all address bits at the same time.
- In order to achieve the multiplexing, memory controller circuit is inserted between the processor and memory.

Memory controller (contd..)



Read-Only Memories (ROMs)

- SRAM and SDRAM chips are volatile:
  - Lose the contents when the power is turned off.
- Many applications need memory devices to retain contents after the power is turned off.
  - For example, computer is turned on, the operating system must be loaded from the disk into the memory.
  - Store instructions which would load the OS from the disk.
  - Need to store these instructions so that they will not be lost after the power is turned off.
  - We need to store the instructions into a non-volatile memory.
- Non-volatile memory is read in the same manner as volatile memory.
  - Separate writing process is needed to place information in this memory.
  - Normal operation involves only reading of data, this type of memory is called Read-Only memory (ROM).

Read-Only Memories (Contd.,)

- Read-Only Memory:
  - Data are written into a ROM when it is manufactured.
- Programmable Read-Only Memory (PROM):
  - Allow the data to be loaded by a user.
  - Process of inserting the data is irreversible.
  - Storing information specific to a user in a ROM is expensive.
  - Providing programming capability to a user may be better.
- Erasable Programmable Read-Only Memory (EPROM):
  - Stored data to be erased and new data to be loaded.
  - Flexibility, useful during the development phase of digital systems.
  - Erasable, reprogrammable ROM.
  - Erasure requires exposing the ROM to UV light.

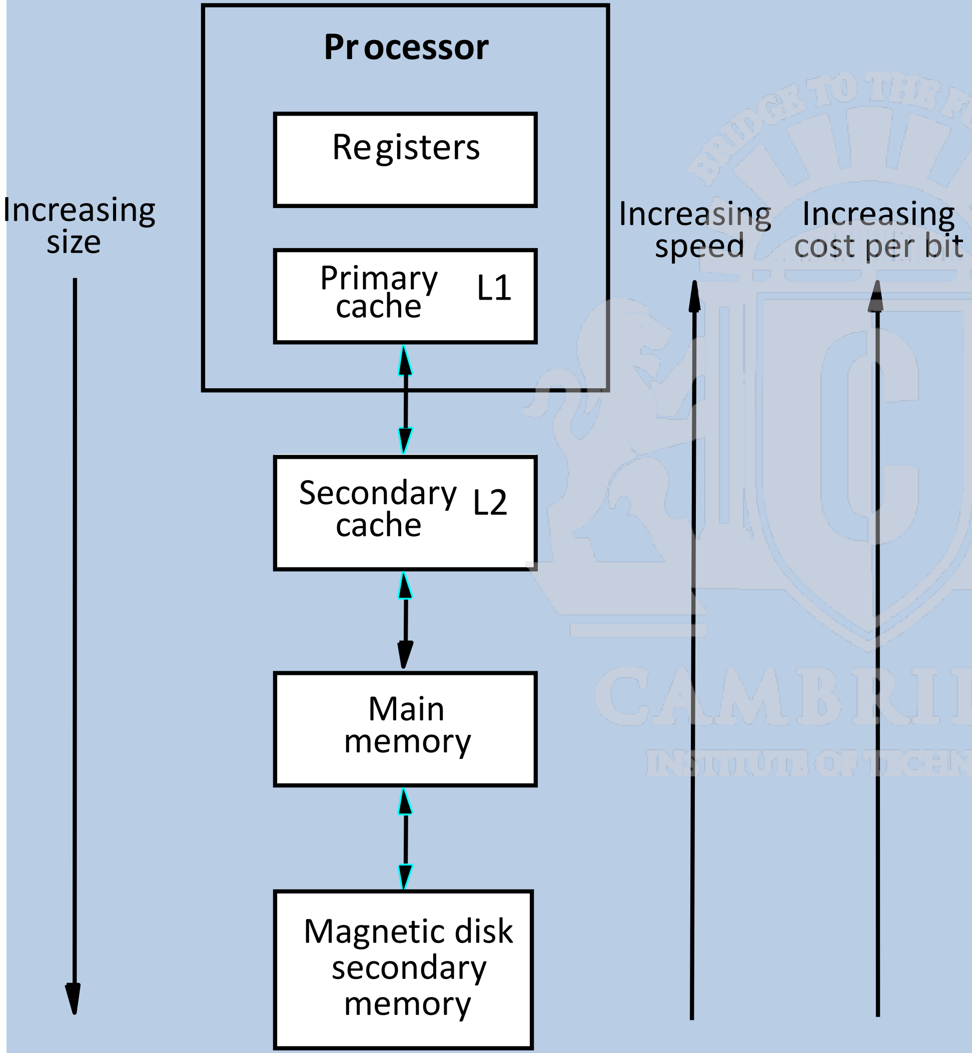
Read-Only Memories (Contd.,)

- Electrically Erasable Programmable Read-Only Memory (EEPROM):
  - To erase the contents of EPROMs, they have to be exposed to ultraviolet light.
  - Physically removed from the circuit.
  - EEPROMs the contents can be stored and erased electrically.
- Flash memory:
  - Has similar approach to EEPROM.
  - Read the contents of a single cell, but write the contents of an entire block of cells.
  - Flash devices have greater density.
    - Higher capacity and low storage cost per bit.
  - Power consumption of flash memory is very low, making it attractive for use in equipment that is battery-driven.
  - Single flash chips are not sufficiently large, so larger memory modules are implemented using flash cards and flash drives.

Speed, Size, and Cost

- A big challenge in the design of a computer system is to provide a sufficiently large memory, with a reasonable speed at an affordable cost.
- Static RAM:
  - Very fast, but expensive, because a basic SRAM cell has a complex circuit making it impossible to pack a large number of cells onto a single chip.
- Dynamic RAM:
  - Simpler basic cell circuit, hence are much less expensive, but significantly slower than SRAMs.
- Magnetic disks:
  - Storage provided by DRAMs is higher than SRAMs, but is still less than what is necessary.
  - Secondary storage such as magnetic disks provide a large amount of storage, but is much slower than DRAMs.

Memory Hierarchy



- *Fastest access is to the data held in processor registers. Registers are at the top of the memory hierarchy.*
- *Relatively small amount of memory that can be implemented on the processor chip. This is processor cache.*
- *Two levels of cache. Level 1 (L1) cache is on the processor chip. Level 2 (L2) cache is in between main memory and processor.*
- *Next level is main memory, implemented as SIMMs. Much larger, but much slower than cache memory.*
- *Next level is magnetic disks. Huge amount of inexepensive storage.*
- *Speed of memory access is critical, the idea is to bring instructions and data that will be used in the near future as close to the processor as possible.*

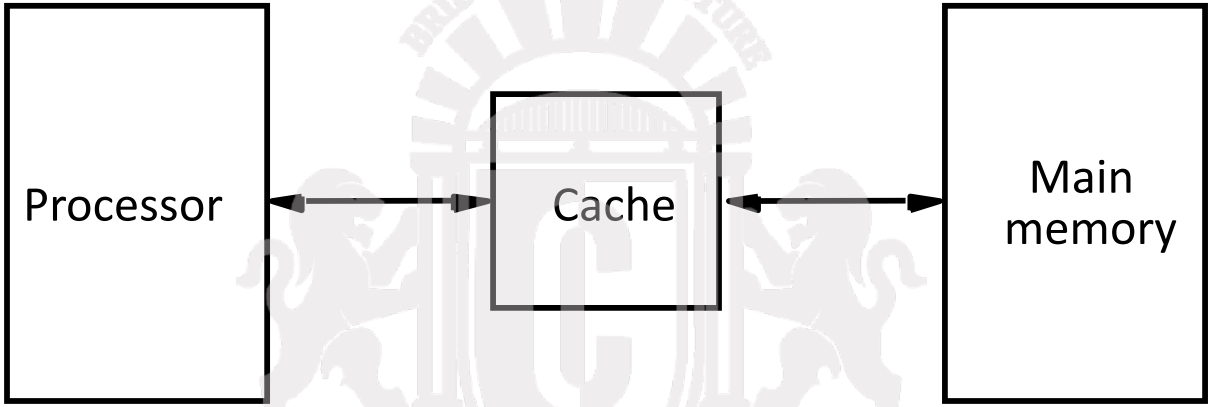
Cache Memories

- Processor is much faster than the main memory.
  - As a result, the processor has to spend much of its time waiting while instructions and data are being fetched from the main memory.
  - Major obstacle towards achieving good performance.
- Speed of the main memory cannot be increased beyond a certain point.
- Cache memory is an architectural arrangement which makes the main memory appear faster to the processor than it really is.
- Cache memory is based on the property of computer programs known as “locality of reference”.

Locality of Reference

- Analysis of programs indicates that many instructions in localized areas of a program are executed repeatedly during some period of time, while the others are accessed relatively less frequently.
  - These instructions may be the ones in a loop, nested loop or few procedures calling each other repeatedly.
  - This is called <u>"locality of reference".</u>
- Temporal locality of reference:
  - Recently executed instruction is likely to be executed again very soon.
- Spatial locality of reference:
  - Instructions with addresses close to a recently instruction are likely to be executed soon.

Cache memories



- Processor issues a Read request, a block of words is transferred from the main memory to the cache, one word at a time.
- Subsequent references to the data in this block of words are found in the cache.
- At any given time, only some blocks in the main memory are held in the cache. Which blocks in the main memory are in the cache is determined by a "mapping function".
- When the cache is full, and a block of words needs to be transferred from the main memory, some block of words in the cache must be replaced. This is determined by a "replacement algorithm".

Cache hit

- Existence of a cache is transparent to the processor. The processor issues Read and
  Write requests in the same manner.

- If the data is in the cache it is called a Read or Write hit.

- Read hit:
  - The data is obtained from the cache.

- Write hit:
  - Cache has a replica of the contents of the main memory.
  - Contents of the cache and the main memory may be updated simultaneously. This is the write-through protocol.
  - Update the contents of the cache, and mark it as updated by setting a bit known as the dirty bit or modified bit. The contents of the main memory are updated when this block is replaced. This is write-back or copy-back protocol.

Cache miss

- If the data is not present in the cache, then a Read miss or Write miss occurs.

- Read miss:
  - Block of words containing this requested word is transferred from the memory.
  - After the block is transferred, the desired word is forwarded to the processor.
  - The desired word may also be forwarded to the processor as soon as it is transferred without waiting for the entire block to be transferred. This is called load-through or early-restart.

- Write-miss:
  - Write-through protocol is used, then the contents of the main memory are updated directly.
  - If write-back protocol is used, the block containing the addressed word is first brought into the cache. The desired word is overwritten with new information.

Cache Coherence Problem

- A bit called as “valid bit” is provided for each block.
- If the block contains valid data, then the bit is set to 1, else it is 0.
- Valid bits are set to 0, when the power is just turned on.
- When a block is loaded into the cache for the first time, the valid bit is set to 1.

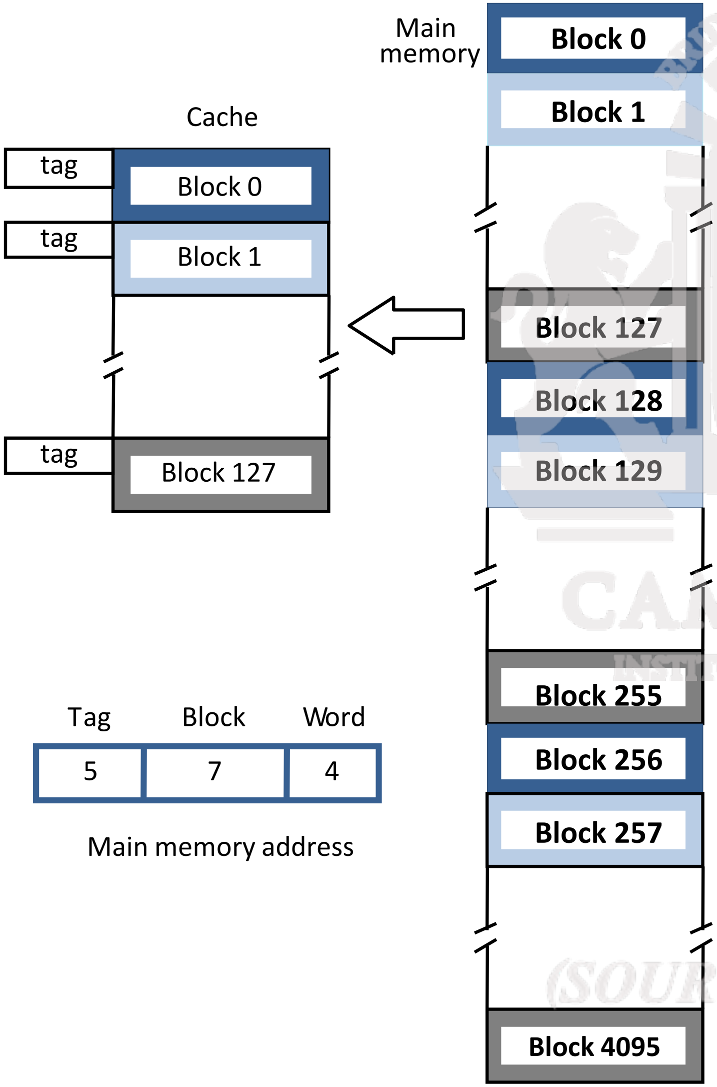
- Data transfers between main memory and disk occur directly bypassing the cache.
- When the data on a disk changes, the main memory block is also updated.
- However, if the data is also resident in the cache, then the valid bit is set to 0.

- What happens if the data in the disk and main memory changes and the write-back protocol is being used?
- In this case, the data in the cache may also have changed and is indicated by the dirty bit.
- The copies of the data in the cache, and the main memory are different. This is called the cache coherence problem.
- One option is to force a write-back before the main memory is updated from the disk.

Mapping functions

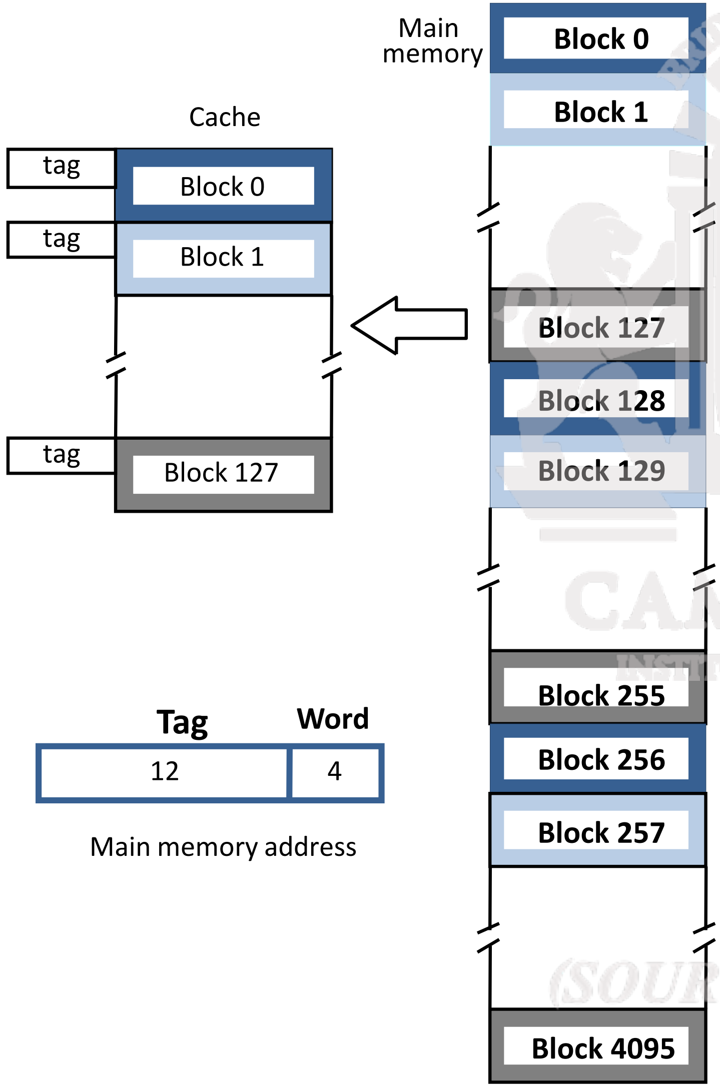
- Mapping functions determine how memory blocks are placed in the cache.
- A simple processor example:
  - Cache consisting of 128 blocks of 16 words each.
  - Total size of cache is 2048 (2K) words.
  - Main memory is addressable by a 16-bit address.
  - Main memory has 64K words.
  - Main memory has 4K blocks of 16 words each.
- Three mapping functions:
  - Direct mapping
  - Associative mapping
  - Set-associative mapping.

Direct mapping



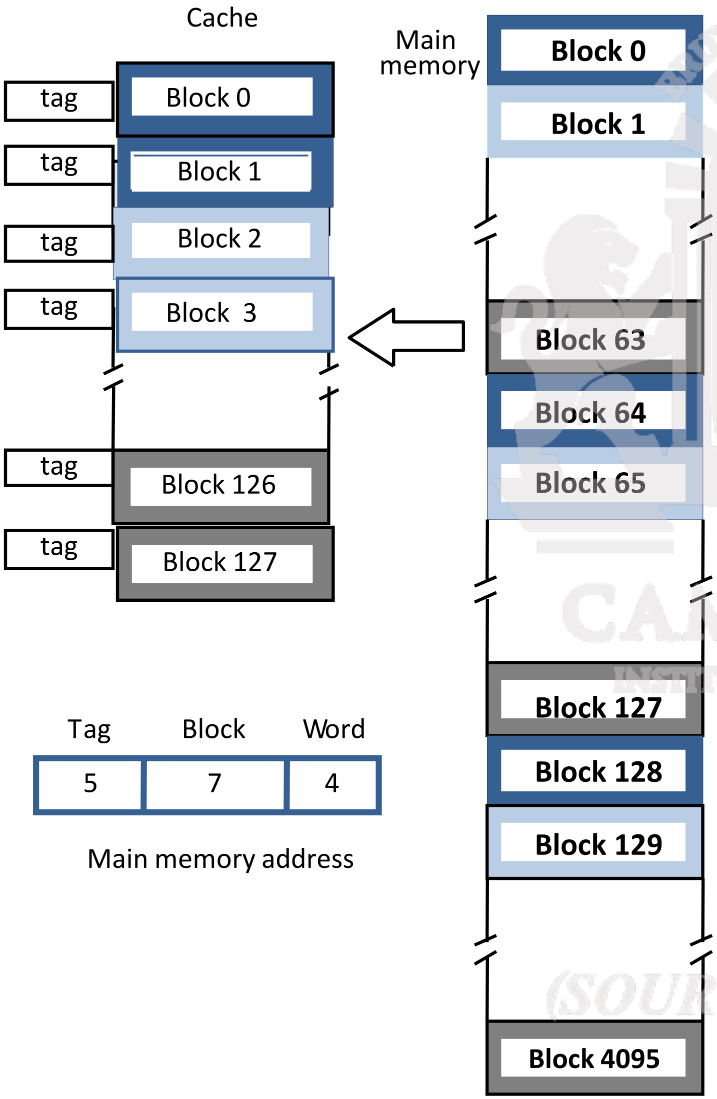
- Block j of the main memory maps to j modulo 128 of the cache. 0 maps to 0, 129 maps to 1.
- More than one memory block is mapped onto the same position in the cache.
- May lead to contention for cache blocks even if the cache is not full.
- Resolve the contention by allowing new block to replace the old block, leading to a trivial replacement algorithm.
- Memory address is divided into three fields:
  - Low order 4 bits determine one of the 16 words in a block.
  - When a new block is brought into the cache, the the next 7 bits determine which cache block this new block is placed in.
  - High order 5 bits determine which of the possible 32 blocks is currently present in the cache. These are tag bits.
- Simple to implement but not very flexible.

Associative mapping



- Main memory block can be placed into any cache position.
- Memory address is divided into two fields:
  - Low order 4 bits identify the word within a block.
  - High order 12 bits or tag bits identify a memory block when it is resident in the cache.
- Flexible, and uses cache space efficiently.
- Replacement algorithms can be used to replace an existing block in the cache when the cache is full.
- Cost is higher than direct-mapped cache because of the need to search all 128 patterns to determine whether a given block is in the cache.

Set-Associative mapping



Blocks of cache are grouped into sets.

Mapping function allows a block of the main memory to reside in any block of a specific set.

Divide the cache into 64 sets, with two blocks per set.

Memory block 0, 64, 128 etc. map to block 0, and they can occupy either of the two positions.

Memory address is divided into three fields:

- 6 bit field determines the set number.
- High order 6 bit fields are compared to the tag fields of the two blocks in a set.

Set-associative mapping combination of direct and associative mapping.

Number of blocks per set is a design parameter.

- One extreme is to have all the blocks in one set, requiring no set bits (fully associative mapping).
- Other extreme is to have one block per set, is the same as direct mapping.

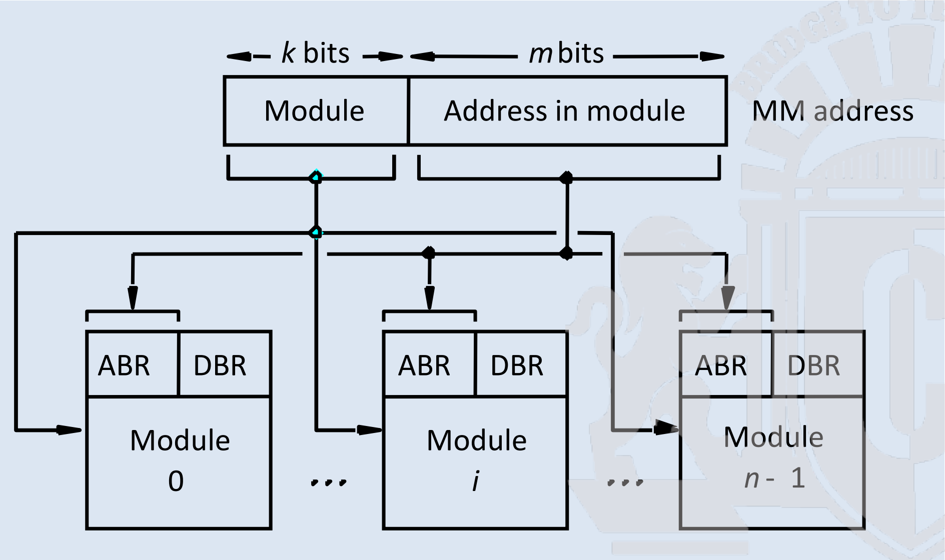
Performance considerations

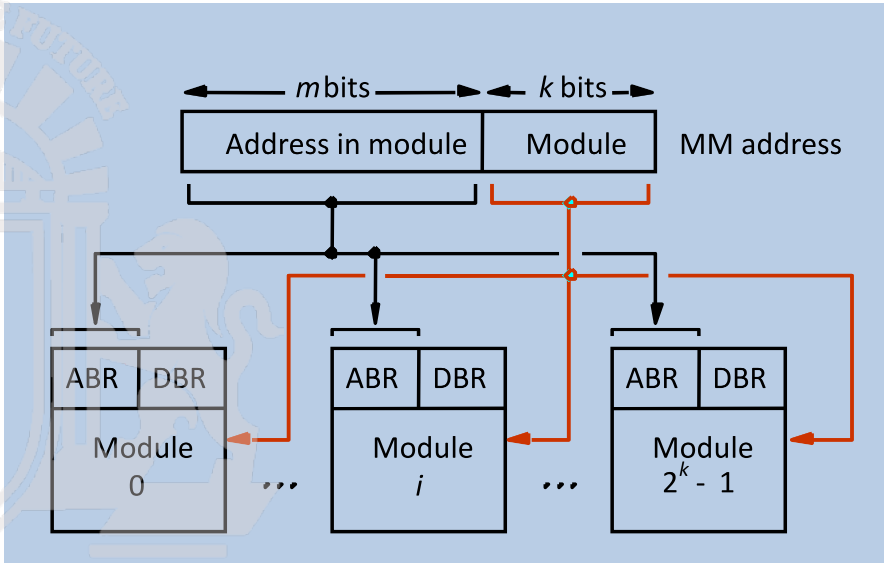
- A key design objective of a computer system is to achieve the best possible performance at the lowest possible cost.
  - Price/performance ratio is a common measure of success.
- Performance of a processor depends on:
  - How fast machine instructions can be brought into the processor for execution.
  - How fast the instructions can be executed.

Interleaving

- Divides the memory system into a number of memory modules. Each module has its own address buffer register (ABR) and data buffer register (DBR).
- Arranges addressing so that successive words in the address space are placed in different modules.
- When requests for memory access involve consecutive addresses, the access will be to different modules.
- Since parallel access to these modules is possible, the average rate of fetching words from the Main Memory can be increased.

Methods of address layouts





- *Consecutive words are placed in a module.*
- *High-order k bits of a memory address determine the module.*
- *Low-order m bits of a memory address determine the word within a module.*
- *When a block of words is transferred from main memory to cache, only one module is busy at a time.*

•Consecutive words are located in consecutive modules.

•Consecutive addresses can be located in consecutive modules.

•While transferring a block of data, several memory modules can be kept busy at the same time.

Hit Rate and Miss Penalty

- Hit rate
- Miss penalty
- Hit rate can be improved by increasing block size, while keeping cache size constant
- Block sizes that are neither very small nor very large give best results.
- Miss penalty can be reduced if load-through approach is used when loading new blocks into cache.

Caches on the processor chip

- In high performance processors 2 levels of caches are normally used.
- Avg access time in a system with 2 levels of caches is

$$T_{ave} = h1c1+(1-h1)h2c2+(1-h1)(1-h2)M$$

Other Performance Enhancements

Write buffer

- Write-through:
- Each write operation involves writing to the main memory.
- If the processor has to wait for the write operation to be complete, it slows down the processor.
- Processor does not depend on the results of the write operation.
- Write buffer can be included for temporary storage of write requests.
- Processor places each write request into the buffer and continues execution.
- If a subsequent Read request references data which is still in the write buffer, then this data is referenced in the write buffer.

- Write-back:
- Block is written back to the main memory when it is replaced.
- If the processor waits for this write to complete, before reading the new block, it is slowed down.
- Fast write buffer can hold the block to be written, and the new block can be read first.

Other Performance Enhancements (Contd.,)

Prefetching

- *New data are brought into the processor when they are first needed.*
- *Processor has to wait before the data transfer is complete.*
- *Prefetch the data into the cache before they are actually needed, or a before a Read miss occurs.*
- *Prefetching can be accomplished through software by including a special instruction in the machine language of the processor.*
  - ***Inclusion of prefetch instructions increases the length of the programs.***
- *Prefetching can also be accomplished using hardware:*
  - ***Circuitry that attempts to discover patterns in memory references and then prefetches according to this pattern.***

Other Performance Enhancements (Contd.,)

Lockup-Free Cache

- *Prefetching scheme does not work if it stops other accesses to the cache until the prefetch is completed.*
- *A cache of this type is said to be “locked” while it services a miss.*
- *Cache structure which supports multiple outstanding misses is called a lockup free cache.*
- *Since only one miss can be serviced at a time, a lockup free cache must include circuits that keep track of all the outstanding misses.*
- *Special registers may hold the necessary information about these misses.*

Virtual memories

- Recall that an important challenge in the design of a computer system is to provide a large, fast memory system at an affordable cost.
- Architectural solutions to increase the effective speed and size of the memory system.
- Cache memories were developed to increase the effective speed of the memory system.
- Virtual memory is an architectural solution to increase the effective size of the memory system.

Virtual memories (contd..)

- Recall that the addressable memory space depends on the number of address bits in a computer.
  - For example, if a computer issues 32-bit addresses, the addressable memory space is 4G bytes.
- Physical main memory in a computer is generally not as large as the entire possible addressable space.
  - Physical memory typically ranges from a few hundred megabytes to 1G bytes.
- Large programs that cannot fit completely into the main memory have their parts stored on secondary storage devices such as magnetic disks.
  - Pieces of programs must be transferred to the main memory from secondary storage before they can be executed.

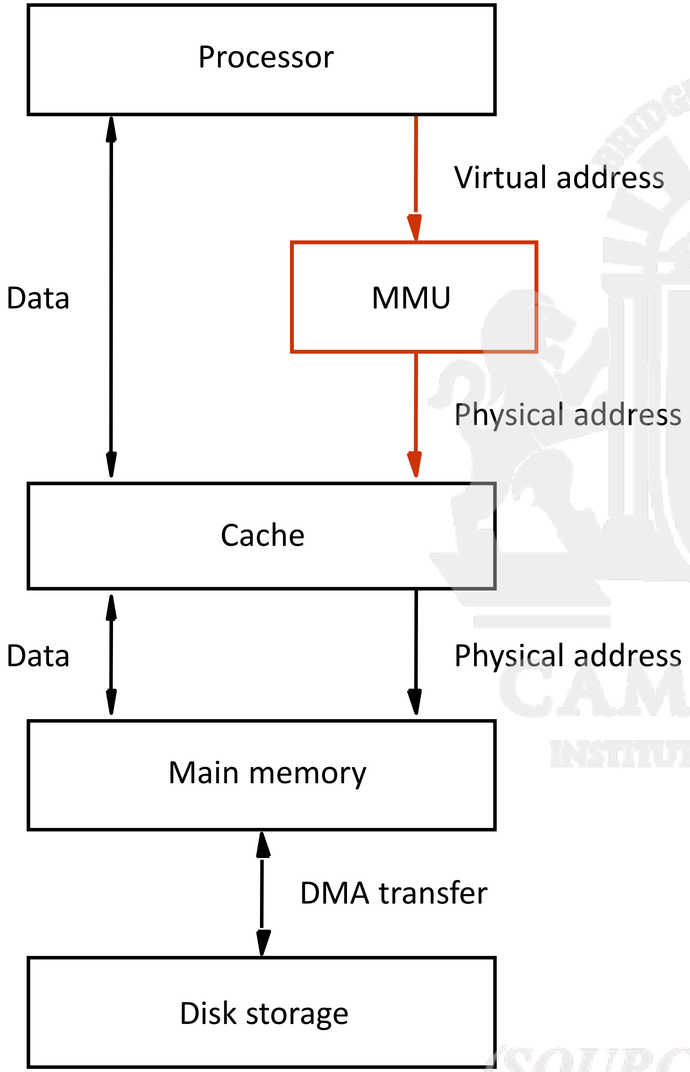
Virtual memories (contd..)

- When a new piece of a program is to be transferred to the main memory, and the main memory is full, then some other piece in the main memory must be replaced.
  - Recall this is very similar to what we studied in case of cache memories.
- Operating system automatically transfers data between the main memory and secondary storage.
  - Application programmer need not be concerned with this transfer.
  - Also, application programmer does not need to be aware of the limitations imposed by the available physical memory.

Virtual memories (contd..)

- Techniques that automatically move program and data between main memory and secondary storage when they are required for execution are called virtual-memory techniques.
- Programs and processors reference an instruction or data independent of the size of the main memory.
- Processor issues binary addresses for instructions and data.
  - These binary addresses are called logical or virtual addresses.
- Virtual addresses are translated into physical addresses by a combination of hardware and software subsystems.
  - If virtual address refers to a part of the program that is currently in the main memory, it is accessed immediately.
  - If the address refers to a part of the program that is not currently in the main memory, it is first transferred to the main memory before it can be used.

Virtual memory organization



- *Memory management unit (MMU) translates virtual addresses into physical addresses.*
- *If the desired data or instructions are in the main memory they are fetched as described previously.*
- *If the desired data or instructions are not in the main memory, they must be transferred from secondary storage to the main memory.*
- *MMU causes the operating system to bring the data from the secondary storage into the main memory.*

Address translation

- Assume that program and data are composed of fixed-length units called pages.
- A page consists of a block of words that occupy contiguous locations in the main memory.
- Page is a basic unit of information that is transferred between secondary storage and main memory.
- Size of a page commonly ranges from 2K to 16K bytes.
  - Pages should not be too small, because the access time of a secondary storage device is much larger than the main memory.
  - Pages should not be too large, else a large portion of the page may not be used, and it will occupy valuable space in the main memory.

(SOURCE DIGINOTES)

Address translation (contd..)

- Concepts of virtual memory are similar to the concepts of cache memory.
- Cache memory:
  - Introduced to bridge the speed gap between the processor and the main memory.
  - Implemented in hardware.
- Virtual memory:
  - Introduced to bridge the speed gap between the main memory and secondary storage.
  - Implemented in part by software.

Address translation (contd..)

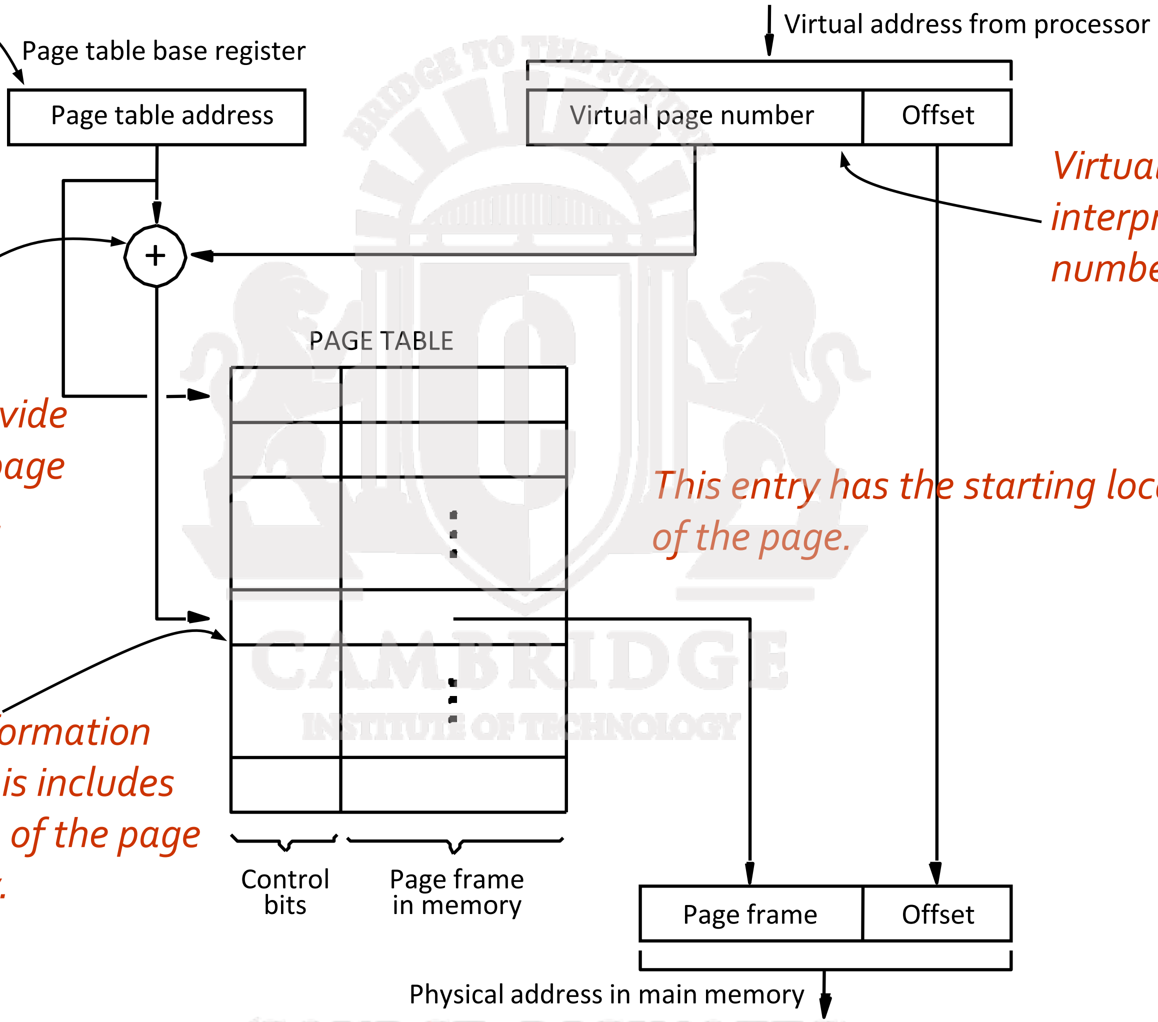
- Each virtual or logical address generated by a processor is interpreted as a virtual page number (high-order bits) plus an offset (low-order bits) that specifies the location of a particular byte within that page.
- Information about the main memory location of each page is kept in the page table.
  - Main memory address where the page is stored.
  - Current status of the page.
- Area of the main memory that can hold a page is called as page frame.
- Starting address of the page table is kept in a page table base register.

Address translation (contd..)

- Virtual page number generated by the processor is added to the contents of the page table base register.
  - This provides the address of the corresponding entry in the page table.
- The contents of this location in the page table give the starting address of the page if the page is currently in the main memory.

Address translation (contd..)

PTBR holds the address of the page table.



Virtual address is interpreted as page number and offset.

PTBR + virtual page number provide the entry of the page in the page table.

This entry has the starting location of the page.

Page table holds information about each page. This includes the starting address of the page in the main memory.

Address translation (contd..)

- Page table entry for a page also includes some control bits which describe the status of the page while it is in the main memory.
- One bit indicates the validity of the page.
  - Indicates whether the page is actually loaded into the main memory.
  - Allows the operating system to invalidate the page without actually removing it.
- One bit indicates whether the page has been modified during its residency in the main memory.
  - This bit determines whether the page should be written back to the disk when it is removed from the main memory.
  - Similar to the dirty or modified bit in case of cache memory.

Address translation (contd..)

- Other control bits for various other types of restrictions that may be imposed.
  - For example, a program may only have read permission for a page, but not write or modify permissions.

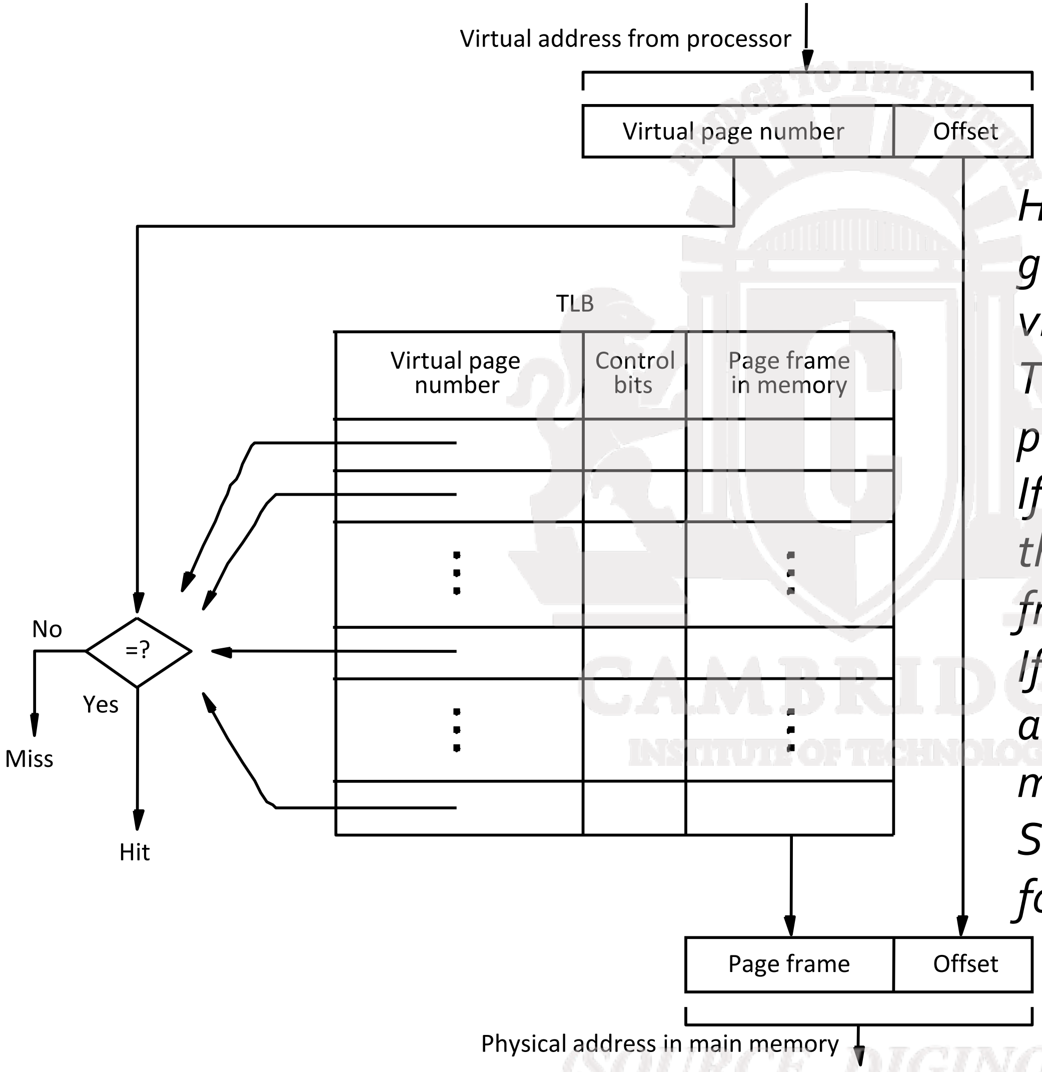
Address translation (contd..)

- Where should the page table be located?
- Recall that the page table is used by the MMU for every read and write access to the memory.
  - Ideal location for the page table is within the MMU.
- Page table is quite large.
- MMU is implemented as part of the processor chip.
- Impossible to include a complete page table on the chip.
- Page table is kept in the main memory.
- A copy of a small portion of the page table can be accommodated within the MMU.
  - Portion consists of page table entries that correspond to the most recently accessed pages.

Address translation (contd..)

- A small cache called as Translation Lookaside Buffer (TLB) is included in the MMU.
  - TLB holds page table entries of the most recently accessed pages.
- Recall that cache memory holds most recently accessed blocks from the main memory.
  - Operation of the TLB and page table in the main memory is similar to the operation of the cache and main memory.
- Page table entry for a page includes:
  - Address of the page frame where the page resides in the main memory.
  - Some control bits.
- In addition to the above for each page, TLB must hold the virtual page number for each page.

Address translation (contd..)



Associative-mapped TLB

High-order bits of the virtual address generated by the processor select the virtual page.
These bits are compared to the virtual page numbers in the TLB.
If there is a match, a hit occurs and the corresponding address of the page frame is read.
If there is no match, a miss occurs and the page table within the main memory must be consulted.
Set-associative mapped TLBs are found in commercial processors.

Address translation (contd..)

- How to keep the entries of the TLB coherent with the contents of the page table in the main memory?
- Operating system may change the contents of the page table in the main memory.
  - Simultaneously it must also invalidate the corresponding entries in the TLB.
- A control bit is provided in the TLB to invalidate an entry.
- If an entry is invalidated, then the TLB gets the information for that entry from the page table.
  - Follows the same process that it would follow if the entry is not found in the TLB or if a “miss” occurs.

Address translation (contd..)

- What happens if a program generates an access to a page that is not in the main memory?
- In this case, a page fault is said to occur.
  - Whole page must be brought into the main memory from the disk, before the execution can proceed.
- Upon detecting a page fault by the MMU, following actions occur:
  - MMU asks the operating system to intervene by raising an exception.
  - Processing of the active task which caused the page fault is interrupted.
  - Control is transferred to the operating system.
  - Operating system copies the requested page from secondary storage to the main memory.
  - Once the page is copied, control is returned to the task which was interrupted.

Address translation (contd..)

- Servicing of a page fault requires transferring the requested page from secondary storage to the main memory.
- This transfer may incur a long delay.
- While the page is being transferred the operating system may:
  - Suspend the execution of the task that caused the page fault.
  - Begin execution of another task whose pages are in the main memory.
- Enables efficient use of the processor.

Address translation (contd..)

- How to ensure that the interrupted task can continue correctly when it resumes execution?
- There are two possibilities:
  - Execution of the interrupted task must continue from the point where it was interrupted.
  - The instruction must be restarted.
- Which specific option is followed depends on the design of the processor.

Address translation (contd..)

- When a new page is to be brought into the main memory from secondary storage, the main memory may be full.
  - Some page from the main memory must be replaced with this new page.
- How to choose which page to replace?
  - This is similar to the replacement that occurs when the cache is full.
  - The principle of locality of reference (?) can also be applied here.
  - A replacement strategy similar to LRU can be applied.
- Since the size of the main memory is relatively larger compared to cache, a relatively large amount of programs and data can be held in the main memory.
  - Minimizes the frequency of transfers between secondary storage and main memory.

Address translation (contd..)

- A page may be modified during its residency in the main memory.
- When should the page be written back to the secondary storage?
- Recall that we encountered a similar problem in the context of cache and main memory:
  - Write-through protocol(?)
  - Write-back protocol(?)
- Write-through protocol cannot be used, since it will incur a long delay each time a small amount of data is written to the disk.

Memory management

- Operating system is concerned with transferring programs and data between secondary storage and main memory.
- Operating system needs memory routines in addition to the other routines.
- Operating system routines are assembled into a virtual address space called system space.
- System space is separate from the space in which user application programs reside.
  - This is user space.
- Virtual address space is divided into one

  system space + several user spaces.

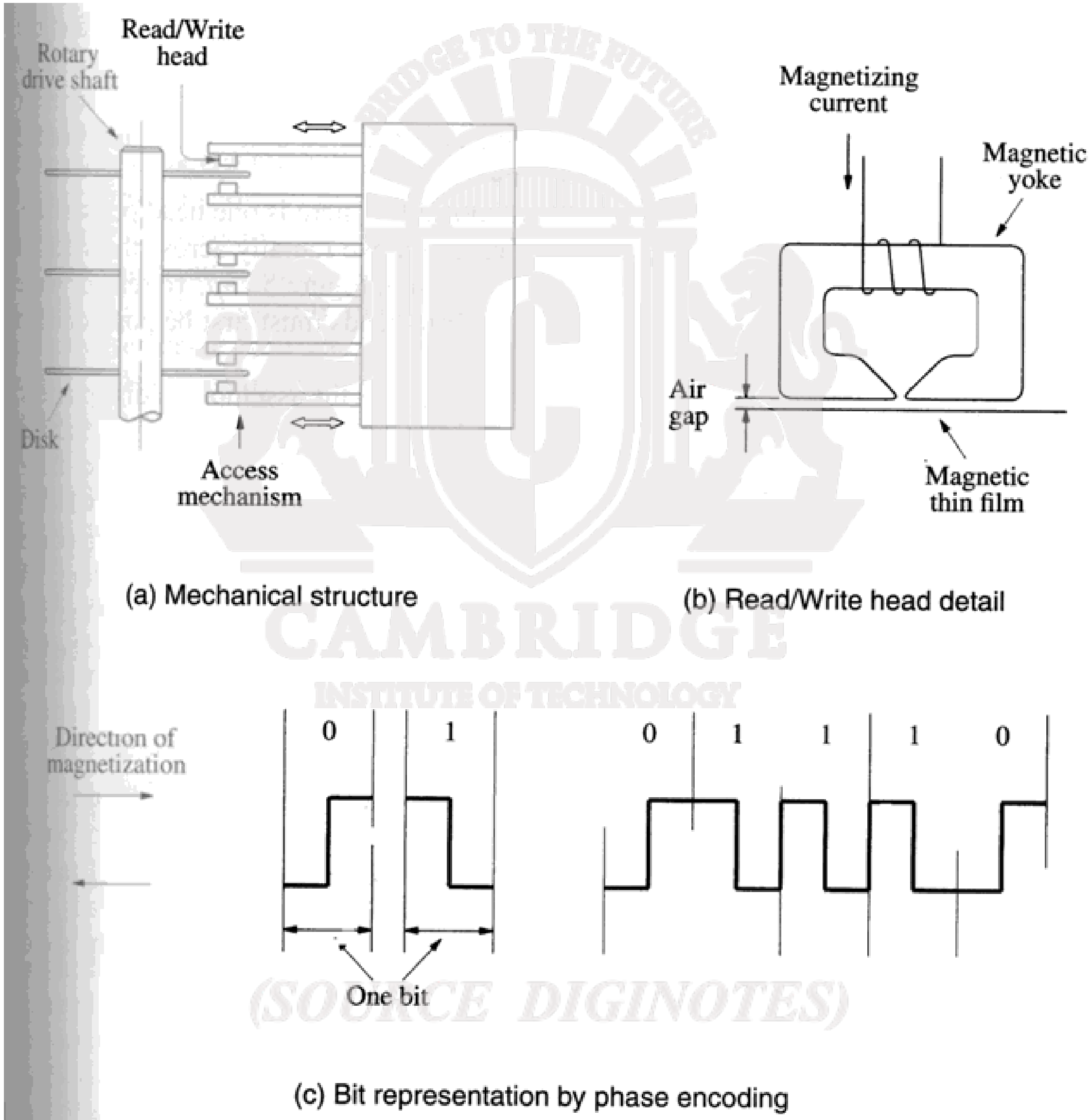
Memory management (contd..)

- Recall that the Memory Management Unit (MMU) translates logical or virtual addresses into physical addresses.
- MMU uses the contents of the page table base register to determine the address of the page table to be used in the translation.
  - Changing the contents of the page table base register can enable us to use a different page table, and switch from one space to another.
- At any given time, the page table base register can point to one page table.
  - Thus, only one page table can be used in the translation process at a given time.
  - Pages belonging to only one space are accessible at any given time.

Memory management (contd..)

- When multiple, independent user programs coexist in the main memory, how to ensure that one program does not modify/destroy the contents of the other?
- Processor usually has two states of operation:
  - Supervisor state.
  - User state.
- Supervisor state:
  - Operating system routines are executed.
- User state:
  - User programs are executed.
  - Certain privileged instructions cannot be executed in user state.
  - These privileged instructions include the ones which change page table base register.
  - Prevents one user from accessing the space of other users.

Magnetic Hard Disks



(a) Mechanical structure

(b) Read/Write head detail

(c) Bit representation by phase encoding

Disk

Disk drive

Disk controller

Organization of Data on a Disk

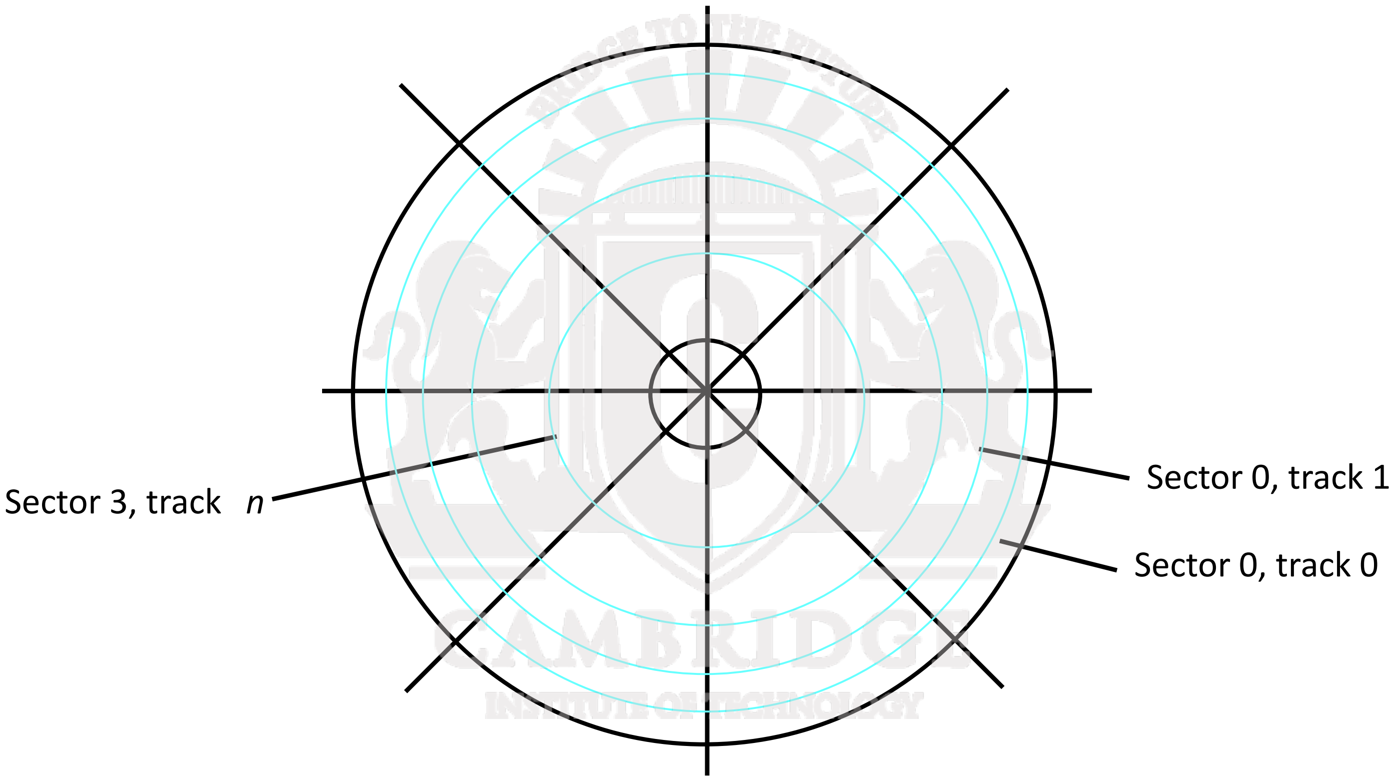


Figure 5.30. Organization of one surface of a disk.

Access Data on a Disk

- Sector header
- Following the data, there is an error-correction code (ECC).
- Formatting process
- Difference between inner tracks and outer tracks
- Access time – seek time / rotational delay (latency time)
- Data buffer/cache

Disk Controller

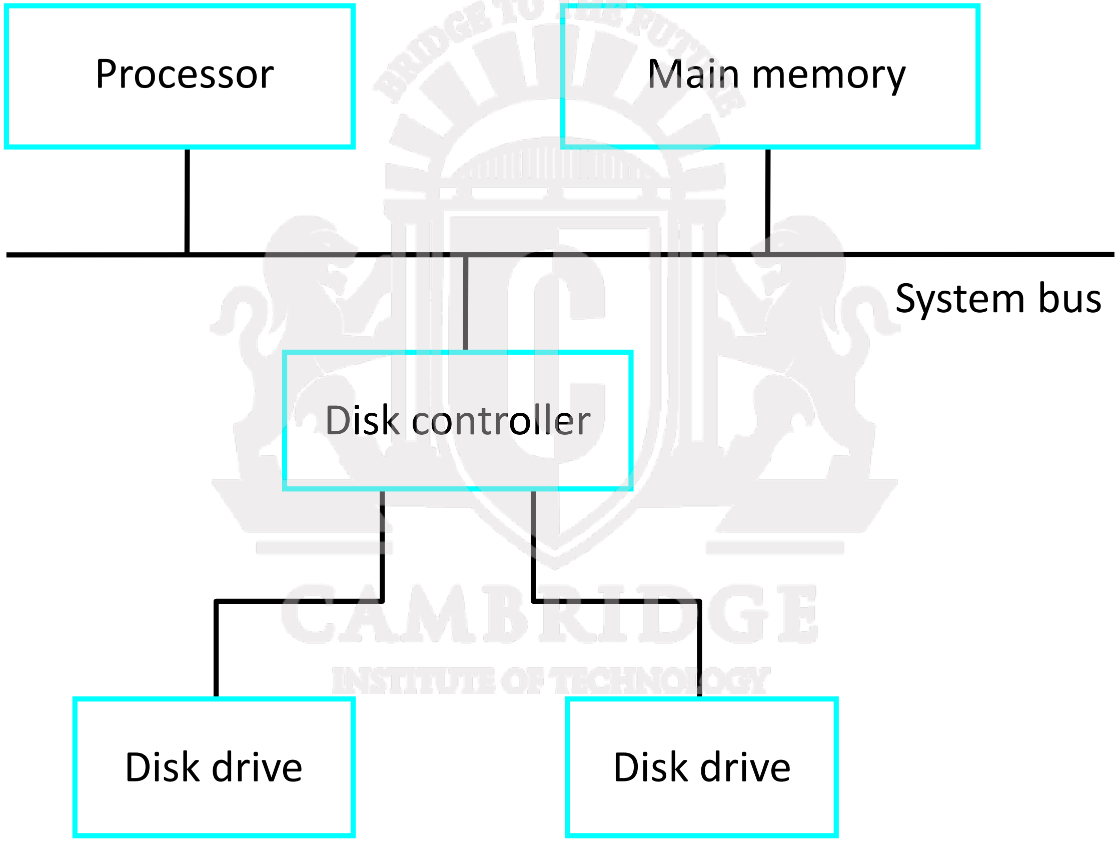


Figure 5.31. Disks connected to the system bus.

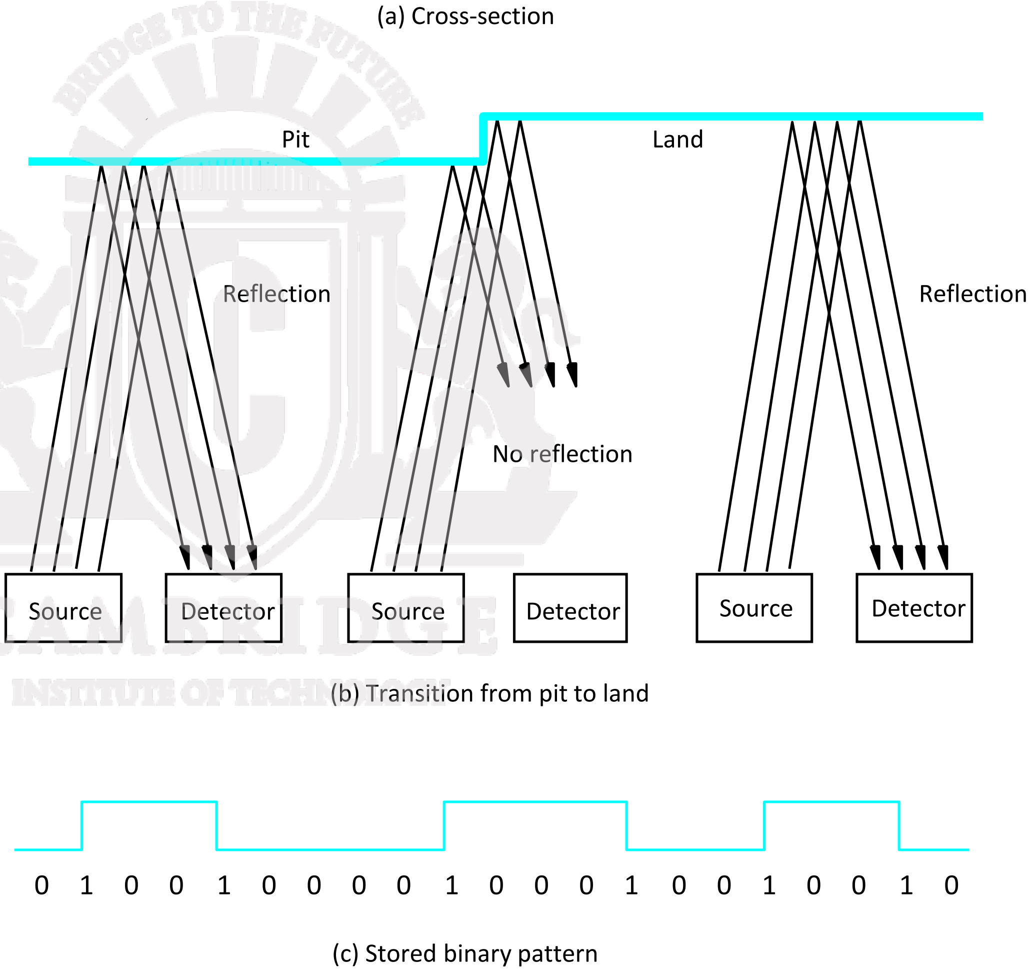
Disk Controller

- Seek
- Read
- Write
- Error checking

RAID Disk Arrays

- Redundant Array of Inexpensive Disks
- Using multiple disks makes it cheaper for huge storage, and also possible to improve the reliability of the overall system.
- RAID0 – data striping
- RAID1 – identical copies of data on two disks
- RAID2, 3, 4 – increased reliability
- RAID5 – parity-based error-recovery

Optical Disks



(a) Cross-section

(b) Transition from pit to land

(c) Stored binary pattern

Figure 5.32. Optical disk.

Optical Disks

- CD-ROM
- CD-Recordable (CD-R)
- CD-ReWritable (CD-RW)
- DVD
- DVD-RAM

Magnetic Tape Systems

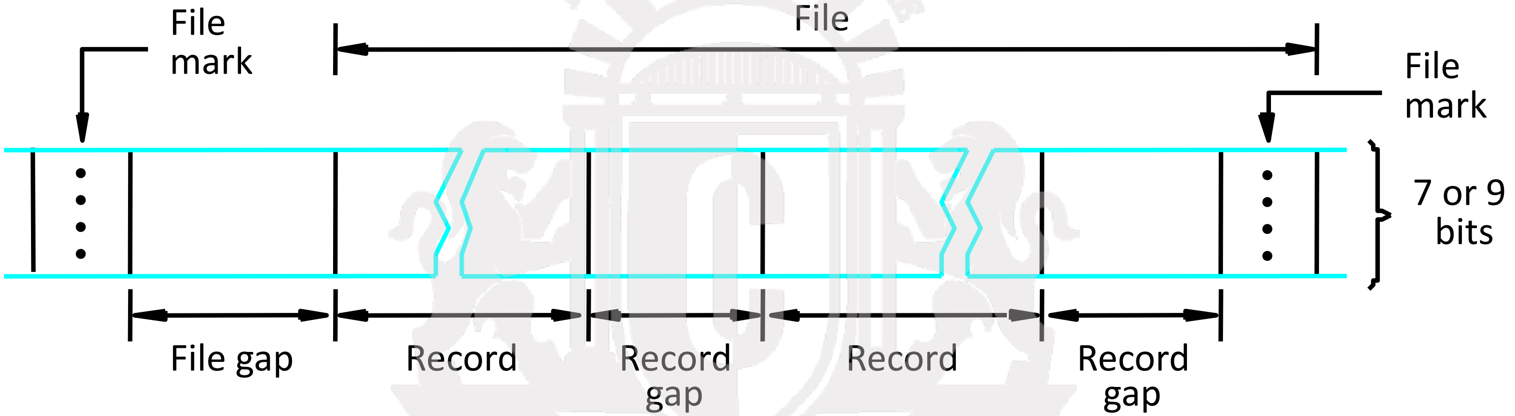


Figure 5.33. Organization of data on magnetic tape.

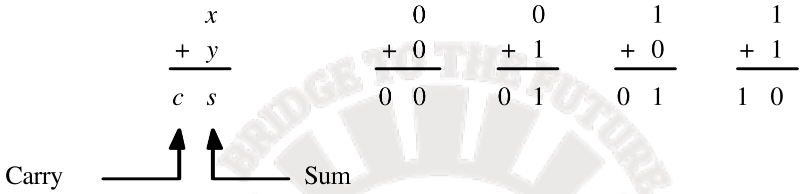
Arithmetic

A basic operation in all digital computers is the addition and subtraction of two numbers They are implemented, along with the basic logic functions such as AND,OR, NOT,EX-OR in the ALU subsystem of the processor. In this chapter we will study how to implement these operations by using different techniques.

Addition and Subtraction of Signed Numbers

Half Adder

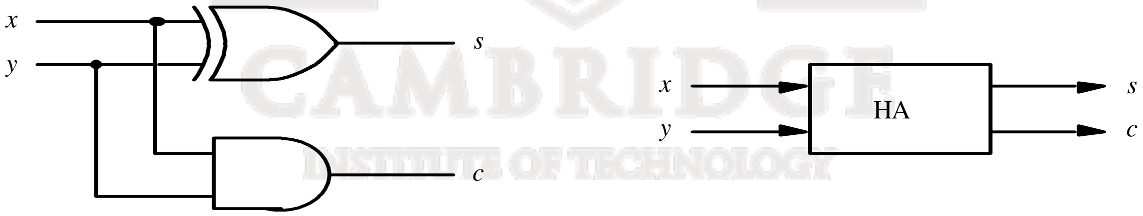
Figure 1(a),(b),(c),(d): Implementation of Half Adder



(a) The four possible cases

x	y	Carry c	Sum s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(b) Truth table



(c) Circuit

(d) Graphical symbol

Full Adder

The following figure 2 shows the logic truth table for the sum and carry-out functions for adding equally weighted bits x_i and y_i in two numbers X and Y. The figure also shows logic expressions for these functions, along with an example of addition of the 4-bit unsigned numbers 7 and 6.

x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$${}_i = \bar{x}_i\bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = y_i c_i \quad x_i c_i + x_i y_i$$

s

Example:

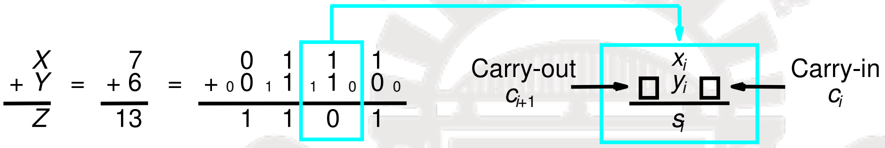


Figure 2

Legend for stage i

Implementation

The logic expression for S_i in figure 2 can be implemented with a 3-input XOR gate,used in figure 3(a)as a part of the logic required for a single stage of binary addition. The carry-out function, Ci+1, is implemented with a two level AND-OR logic circuit.

Using AND –OR gate

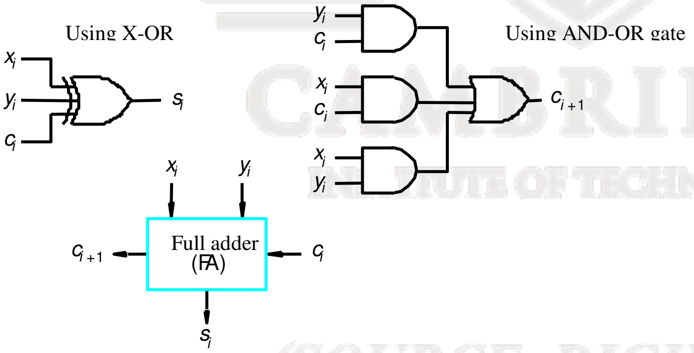


Figure 3(a) Logic For a single stage

A cascaded connection of n full adder blocks, as shown in figure 4(a), can be use to add two n-bit numbers. Since the carries must propagate, or ripple through this cascade, the configuration is called n-bit ripple carry adder.

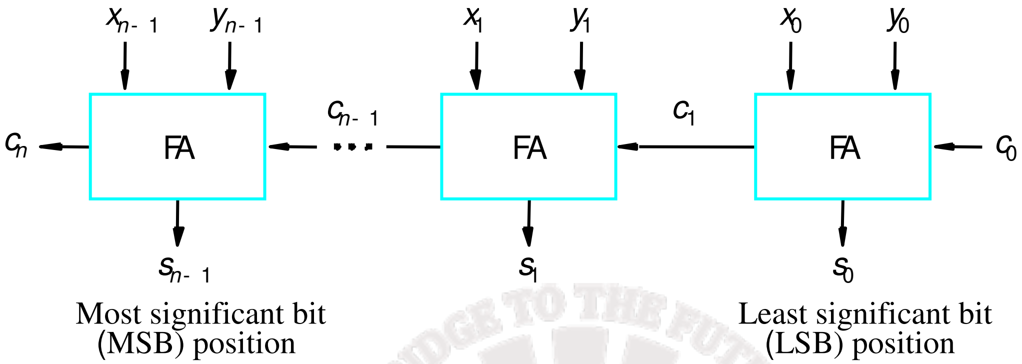


Figure 4(a) An n-bit ripple carry adder

Overflow – Overflow occurs in signed numbers having same signs, and sign of the result is different, and also it is shown that carry bits C_n and C_{n-1} are different. A circuit is added to detect overflow, eg. $C_{n-1} \oplus C_n$

In order to perform the subtract operation X-Y on 2's complement numbers X and Y, we form the 2s-complement of Y and add it to X. The logic circuit network shown in figure (5) can be used to perform either addition or subtraction based on the value applied to the Add/Sub input control line. This line is set to 0 for addition, applying the Y vector un changed to one of the adder inputs along with a carry-in signal,C0 of 0 . When Add/Sub control line is set to 1, the Y vector is 1's complemented by the XOR gates and C0 is set to 1 to complete the 2's complementation of Y. Remember that 2's complementing a negative number is done exactly same manner as for positive number. An XOR gate can be added to Figure(5) to detect the overflow condition $C_{n-1} \oplus C_n$

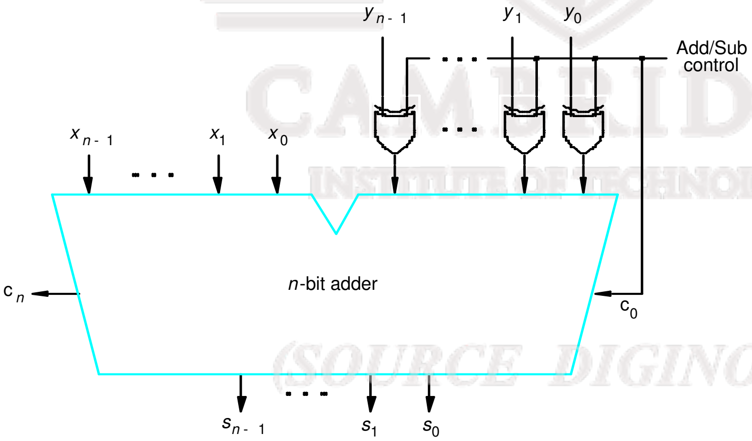


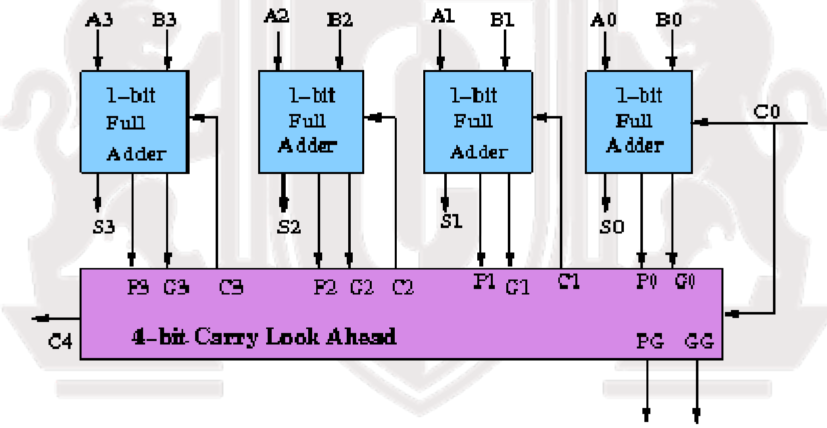
Figure 6.3. Binary addition-subtraction logic netw ork.

Design of Fast Adders

If an n-bit ripple carry adder is used in the addition /subtraction unit of Figure (3), it may have too much delay in developing its outputs, s_0 through s_{n-1} and $c_{n.}$ The delay through any combinational logic network constructed from gates in a particular technology is determined by adding up the number of logic gate delays along the longest signal propagation path through the network. In the case of n-bit ripple-carry adder, the longest path is from inputs x0,y0, and c0 at the LSB position to outputs c_n and s_{n-1} at the most-significant-bit(MSB) position.

Design of Carry Lookahead Adders

To reduce the computation time, there are faster ways to add two binary numbers by using carry lookahead adders. They work by creating two signals P and G known to be Carry Propagator and Carry Generator. The **carry propagator** is propagated to the next level whereas the **carry generator** is used to generate the output carry, regardless of input carry. The block diagram of a 4-bit Carry Lookahead Adder is shown here below -



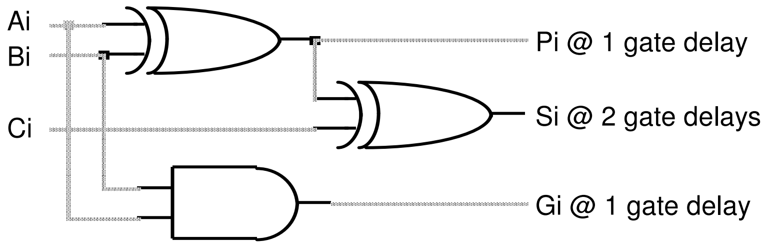
Let us consider the design of a 4 bit adder is shown in figure (6). The carries can be implemented as C1=G0+P0C0

$C_2=G_1+P_1G_0+P_1P_0C_0$

$C_3=G_2+P_2G_1+P_2P_1G_0+P_2P_1P_0C_0$

$C_4=G_3+P_3G_2+P_3P_1G_1+P_3P_2P_1G_0+P_3P_2P_1P_0C_0$

Each of the carry equations can be implemented in a two-level logic network.Variables are the adder inputs and carry in to next stage



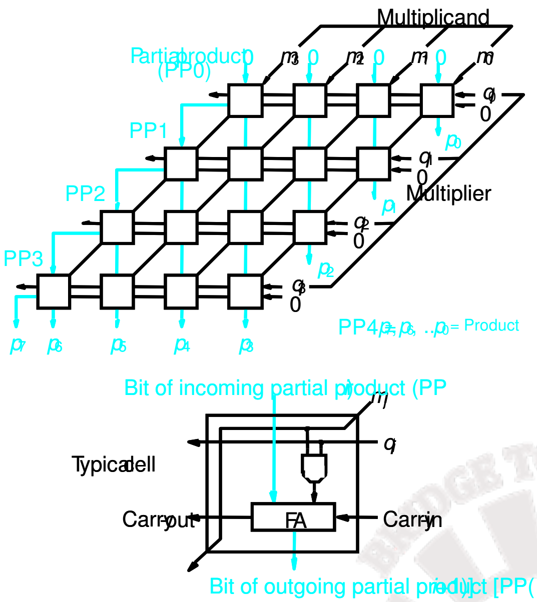
The number of gate levels for the carry propagation can be found from the circuit of full adder. The signal from input carry Cin to output carry Cout requires an AND gate and an OR gate, which constitutes two gate levels. So if there are four full adders in the parallel adder, the output carry C_5 would have 2 X 4 = 8 gate levels from C_1 to C_5. For an n-bit parallel adder, there are 2n gate levels to propagate through..

Multiplication of Positive numbers

The usual algorithm for multiplying integers by hand is illustrated in figure7(a) for the binary system.This algorithm applies to unsigned numbers and to positive signed numbers. The product of two n-digit numbers can be accommodated in 2n digits, so the product of the 4 bit numbers in this example fits into 8 bits.

```
                1   1   0   1          (13)
            ×   1   0   1   1          (11)
          -------------------
                1   1   0   1
            1   1   0   1
        0   0   0   0
    1   1   0   1
-----------------------------
1   0   0   0   1   1   1   1          (143)
```

Figure 7(a) Manual multiplication algorithm



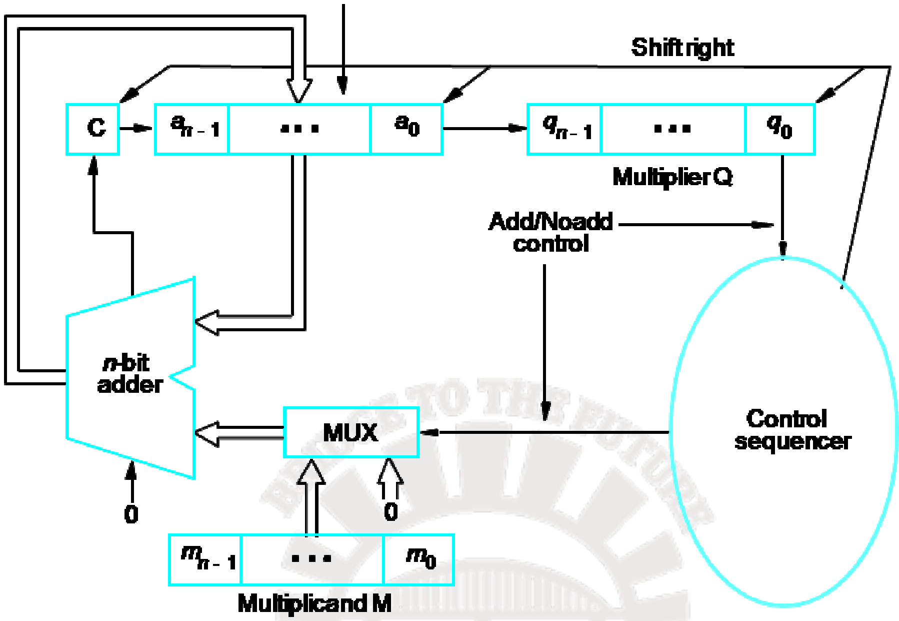
Figure

Binary multiplication of positive operands can be implemented in a combinational two dimensional logic array as shown in figure7(b). The main component in each cell is a full adder FA. The AND gate in each cell determines whether a multiplicand bit m_j , is added to the incoming partial product bit, based on the value of the multiplierbit qj. Each row I,where $0 \leq I \leq 3$, adds the multiplicand to the incoming partial product, PP_i to generate the outgoing partial product, $PP_{(i+1)}$, if q_i=1. If q_i=0, PP_i is passed vertically downward unchanged. PP_0 is all 0s, and PP_4 is the desired product. The multiplicand is shifted left one position per row by the diagonal signal path.

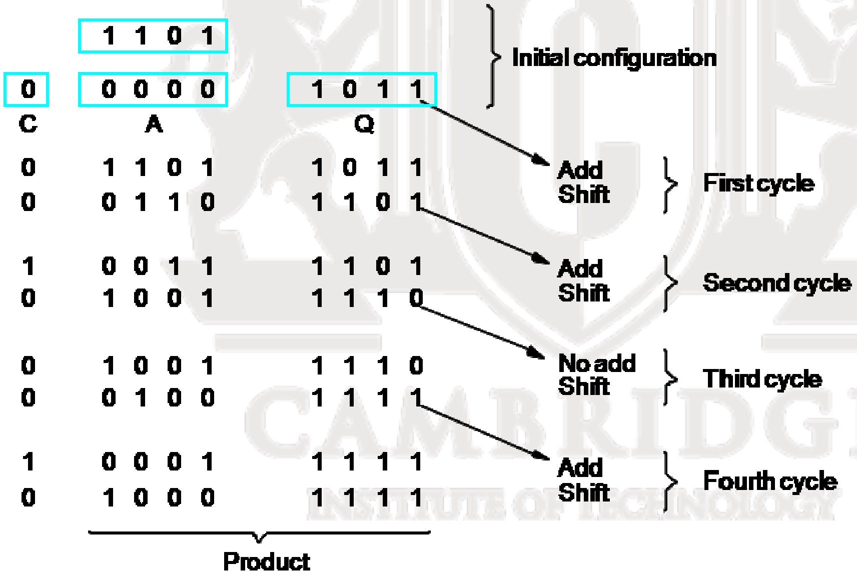
Worst case signal propagation delay path is from the upper right corner of the array to the higher order product bit output at the bottom left corner of the array.

Sequential Circuit Binary multiplier

Registers A and Q combined hold PP_i multiplier bit qi generates the signal Add/Noadd. This signal controls the addition of the multiplicand, M to PP_i to genertae $PP_{(i+1)}$. The product is computed in n cycles. The partial product grows in length by one bit per cycle from the initial vector,PP_0 of n 0s in register A. The carry-out from the adder is stored in flip-flop C, shown at the left end of register A. At the start, the multiplier is loaded into register Q, the multiplicand into register M, and C and A are cleared to 0. At the end of each cycle, C,A and Q are shifted right one bit position to allow for growth of the partial product as the multiplier is shifted out of register Q. Because of this shifting, multiplier bit qi appears at the LSB position Q to generate the Add/Noadd signal at the correct time, starting with q0 during the first cycle, q1 during the second cycle, and so on.
After they are used , the multiplier bits are discarded by the right shift operation. Note that the carry-out from the adder is the leftmost bit of $PP_{(i+1)}$, and must be held in the C flip-flop to be shifted right with the contents of A and Q. After n cycles, the high-order half of the product is held in register A and the low order half is in register Q.



(a) Register configuration



	1 1 0 1			
0	0 0 0 0	1 0 1 1		Initial configuration
C	A	Q		
0	1 1 0 1	1 0 1 1	Add	First cycle
0	0 1 1 0	1 1 0 1	Shift	
1	0 0 1 1	1 1 0 1	Add	Second cycle
0	1 0 0 1	1 1 1 0	Shift	
0	1 0 0 1	1 1 1 0	No add	Third cycle
0	0 1 0 0	1 1 1 1	Shift	
1	0 0 0 1	1 1 1 1	Add	Fourth cycle
0	1 0 0 0	1 1 1 1	Shift	

Product

(b) Multiplication example

Signed Multiplication

Booth Algorithm

A powerful algorithm for signed –number multiplication is a Booth’s algorithm which generates a 2n bit product and treats both positive and negative numbers uniformly. This algorithm suggest that we can reduce the number of operations required for multiplication by representing multiplier as a difference between two numbers.

For example, multiplier 0 0 1 1 1 0(14) can be represented as follows.

```
       0 1 0 0 0 0 (16)
   -   0 0 0 0 1 0 (2)
  -----------------------
       0 0 1 1 1 0 (14)
```

Therefore, the product can be computed by adding 24 times the multiplicand to the 2s complement of 21 times the multiplicand. In simple notations, we can describe the sequence of required operations be recoding the preceding multiplier as

0 + 1 0 0 -1 0

In general , For Booth’s algorithm recoding scheme can be given as

-1 times the shifted multiplicand is selected when moving from 0 to 1,+1 times the shifted multiplicand is selected when moving from 1 to 0, and 0 timesw the shifted muluiplicand is selected for none of the above case,as multiplier is scanned from right to left.

Fast Multiplication -- Booth's Algorithm

The Booth's algorithm serves two purposes:

1. Fast multiplication (when there are consecutive 0's or 1's in the multiplier).
2. Signed multiplication.

$98,765 \times 10,001$ $\qquad$ $98,765 \times 9,999$

First consider two decimal multiplications: and . It is obvious that If straight forward multiplication is used, the first one is easier than the second as only two single-digit multiplications are needed for the first while four are needed for the second. However, as we also realize that:

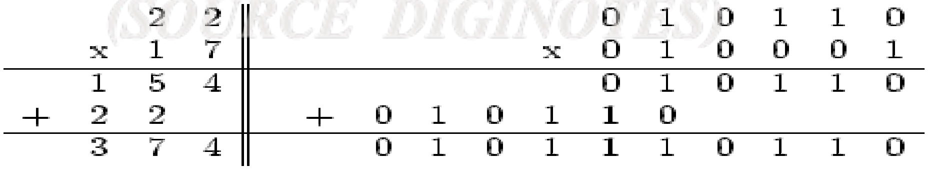
$$98765 \times 10001 = 98765 \times (10000 + 1) = 98765 \times 10000 + 98765$$

$$98765 \times 9999 = 98765 \times (10000 - 1) = 98765 \times 10000 - 98765$$

the two multiplications should be equally easy.

Example 1

If there is a sequence of 0's in the multiplier, the multiplication is easy as all 0's can be skipped.



```
        2  2 ||                     0  1  0  1  1  0
     x  1  7 ||                  x  0  1  0  0  0  1
   ----------||-------------------------------------
        1  5  4 ||                  0  1  0  1  1  0
   +  2  2      ||    +  0  1  0  1  1  0
   ----------||-------------------------------------
      3  7  4 ||         0  1  0  1  1  1  0  1  1  0
```

Example 2

However, it does not help if there is a sequence of 1's in the multiplier. We have to go through each one of them:

$$\begin{array}{cccc||ccccccccccc}
 & & 2 & 2 & & & & & & & 0 & 1 & 0 & 1 & 1 & 0 \\
 & \mathrm{x} & 1 & 4 & & & & & & \mathrm{x} & 0 & 0 & 1 & 1 & 1 & 0 \\
\hline
 & & 8 & 8 & & & & & & 0 & 1 & 0 & 1 & 1 & 0 & \\
| & 2 & 2 & & & & & & 0 & 1 & 0 & 1 & 1 & 0 & & \\
 & & & & + & 0 & 0 & 1 & 0 & 1 & 1 & 0 & & & & \\
\hline
 & 3 & 0 & 8 & & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0
\end{array}$$

How can we enjoy a sequence of 1's as well as a sequence of 0's? We first Realize that $001110 = 010000 - 000010$, or in general a string of 1's in the multiplier A can be written as:

$$\begin{array}{cccccccccccc}
 & d & d & \underline{0} & \underline{1} & 1 & \cdots & 1 & \underline{1} & \underline{0} & d & d \\
= & d & d & \underline{1} & \underline{0} & 0 & \cdots & 0 & 0 & 0 & d & d \\
- & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & \underline{1} & \underline{0} & 0 & 0
\end{array}$$

where d is ``don't care'' (either 0 or 1). If we define the first part of the above as $A_{leftend} = dd10\cdots00dd$ and the second part as $A_{rigtend} = 0000\cdots1000$, then the multiplication becomes

$$B \times A = B \times (A_{leftend} - A_{rightend}) = B \times A_{leftend} - B \times A_{rightend}$$

In other words, only the two ends of a string of 1's in the multiplier need to be taken care of. At the left end the multiplicand is added to the partial product, while at the right end the multiplicand is subtracted from the partial product. The above multiplication can therefore be written as:

$$\begin{array}{ccccccccccc||ccccccccccc}
 & & & & & 0 & 1 & 0 & 1 & 1 & 0 & & & & & & 0 & 1 & 0 & 1 & 1 & 0 \\
 & & & & \mathrm{x} & 0 & \underline{0} & \underline{1} & 1 & \underline{1} & \underline{0} & & & & & \mathrm{x} & 0 & \underline{0} & \underline{1} & 1 & \underline{1} & \underline{0} \\
\hline
 & 0 & 1 & 0 & 1 & 1 & 0 & & & & & & 0 & 1 & 0 & 1 & 1 & 0 & & & & \\
- & & & & 0 & 1 & 0 & 1 & 1 & 0 & & + & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & \\
\hline
 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0
\end{array}$$

On the right side above the subtraction is carried out by adding 2's complement. We observe that there is a sequence of 1's in the multiplier, only the two ends need to be taken care of, while all 1's in between do not require any operation. The Booth's algorithm for multiplication is based on this observation. To do a multiplication, $B \times A$

where

- $B = b_{n-1}b_{n-2}\cdots b_1b_0$ is the multiplicand
- $A = a_{n-1}a_{n-2}\cdots a_1a_0$ is the multiplier

we check every two consecutive bits in A at a time:

a_i	a_{i-1}	$a_{i-1} - a_i$	Operations
0	0	0	in middle of string of 0. No operation.
1	0	-1	beginning of string of 1. Subtract **B** from partial product
1	1	0	in middle of string of 1. No operation.
0	1	1	end of string of 1. Add **B** to partial product

where $i = 0, 1, \cdots, n-1$, and when $i = 0$, $a_{i-1} = a_{-1} \equiv 0$.

Why does it work? What we did can be summarized as the following

$$Product : \quad (a_{-1} - a_0) \times B \times 2^0 + (a_0 - a_1) \times B \times 2^1 + (a_1 - a_2) \times B \times 2^2 +$$

$$\cdots + (a_{n-2} - a_{n-1}) \times B \times 2^{n-1}$$

$$: \quad B \times [\, -a_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} a_i \times 2^i \,]$$

$$\overset{*}{=} B \times Val(A)$$

* Recall that the value of a signed-2's complement number (either positive or negative) can be found by:

$$Val(A = a_{n-1}\cdots a_0) = -a_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} a_i \times 2^i$$

Another Example:

Assume $n = 7$ bits available. Multiply $B = 22 = (0010110)_2$ by $A = -34 = -(0100010)_2$. First represent both operands and their negation in signed 2's complement:

$$\begin{array}{llll} 22: & 0010110, & -22: & 1101010 \\ 34: & 0100010, & -34: & 1011110 \end{array}$$

Then carry out the multiplication in the hardware:

		[M]	0010110			
q_iq_{i-1}	Action	[A]	0000000	[Q]	1011110	0
00	right shift		0000000		0101111	0
10	-B	+	1101010			
			1101010		0101111	0
	right shift		1110101		0010111	1
11	right shift		1111010		1001011	1
11	right shift		1111101		0100101	1
11	right shift		1111110		1010010	1
01	+B	+	0010110			
			0010100		1010010	1
	right shift		0001010		0101001	0
10	-B	+	1101010			
			1110100		0101001	0
	right shift		1111010		0010100	1

The upper half of the final result 1111010 0010100 is in register [A] while the lower half is in register [Q]. The product is given in signed 2's complement and its actual value is negative of the 2's complement:

$$B \times A = -\overline{11110100010100} = -00001011101100 = -748_{10}$$

Another Example

```
   0 1 1 0 1   (+13)                0  1  1  0  1
 × 1 1 0 1 0   (- 6)      ⟹         0 -1 +1     0
 -----------                       --------------
                      0  0  0  0  0  0  0  0  0  0
                      1  1  1  1  1  0  0  1  1
                      0  0  0  0  1  1  0  1
                      1  1  1  0  0  1  1
                      0  0  0  0  0  0
                      -----------------------------
                      1  1  1  0  1  1  0  0  1  0   (- 78)
```

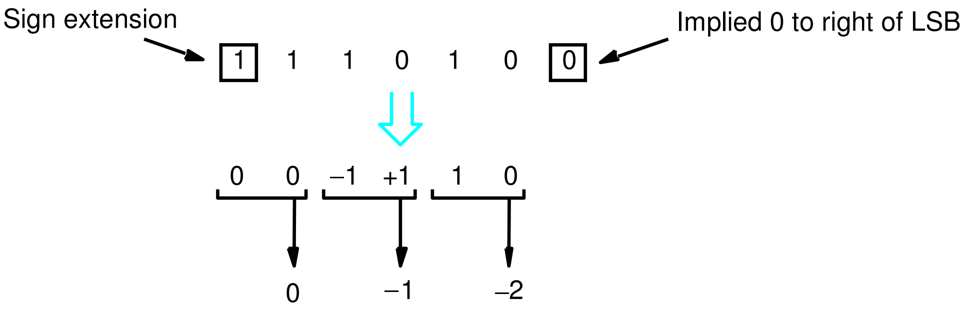
Also note that:

- As the operands are in signed 2's complement form, the *arithmetic shift* is used for the right shifts above, i.e., the MSB bit (sign bit) is always repeated while all other bits are shifted to the right. This guarantees the proper sign extension for both positive and negative values represented in signed 2's complement.
- When the multiplicand is negative represented by signed 2's complement, it needs to be complemented again for subtraction (when the LSB of multiplier is 1 and the extra bit is 0, i.e., the beginning of a string of 1's).

- Best case – a long string of 1's (skipping over 1s)
- Worst case – 0's and 1's are alternating

Bit-Pair Recoding of Multipliers

- Group the booth recoded multiplier bits in pairs, and can be observed, that, the pair (+1, -1) is same to the pair (0, +1), i.e., Instead of adding -1 x M at shift position i with +1 x M at i+1, it can be added with +1 x M at position i.
- Bit-pair recoding halves the maximum number of summands (versions of the multiplicand).



Multiplier bit-pair		Multiplier bit on the right	Multiplicand selected at position *i*
i + 1	*i*	*i* − 1	
0	0	0	0× M
0	0	1	+ 1× M
0	1	0	+ 1× M
0	1	1	+ 2× M
1	0	0	2× M
1	0	1	− 1× M
1	1	0	− 1× M
1	1	1	0× M

(b) Table of multiplicand selection decisions

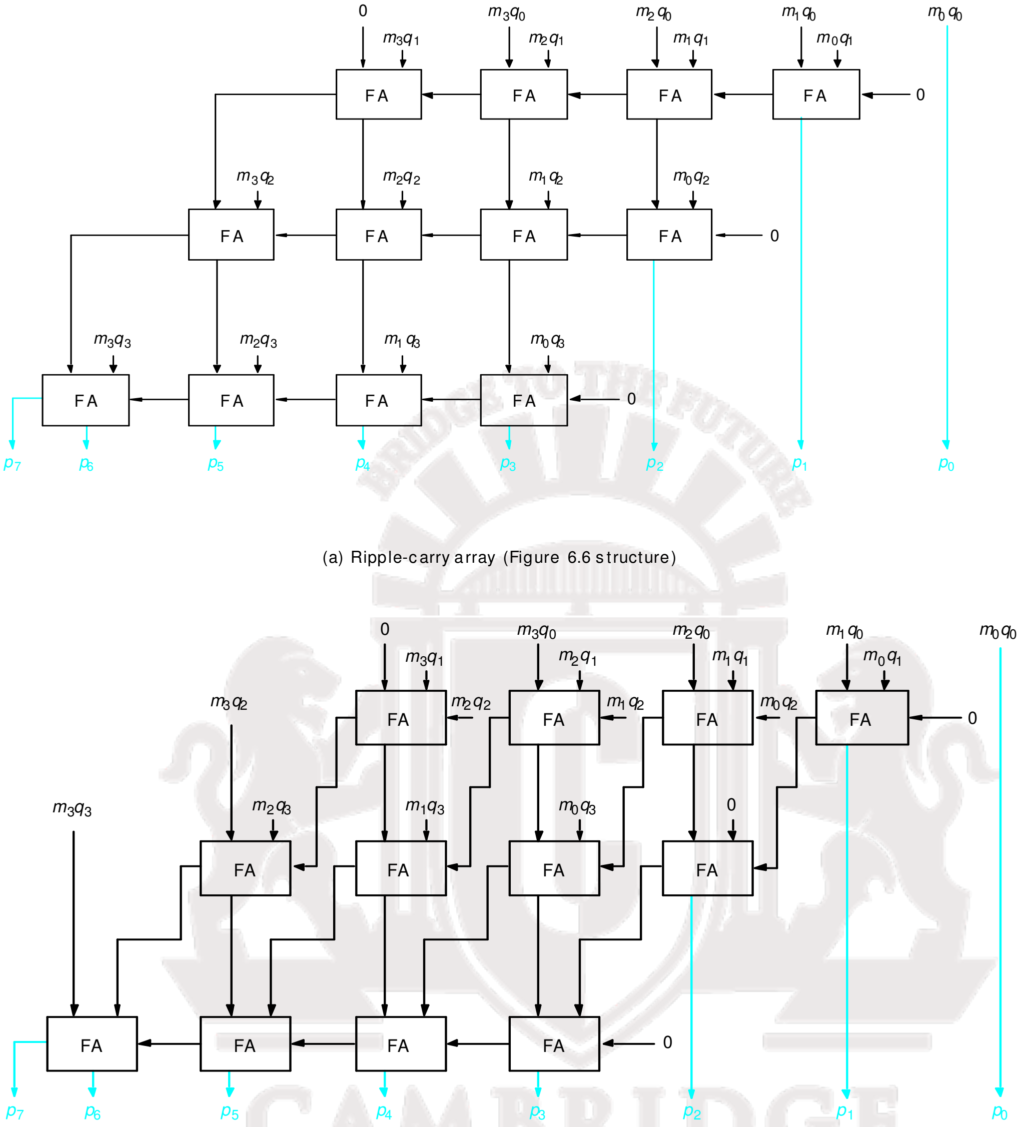
Example

```
               0  1  1  0  1   (+13)
               1  1  0  1  0   (-6)
            -----------------

                     ⇓

               0  1  1  0  1
               0 -1 +1 -1  0
               --------------
0  0  0  0  0  0  0  0  0  0
1  1  1  1  1  0  0  1  1
0  0  0  0  1  1  0  1
1  1  1  0  0  1  1
0  0  0  0  0  0
-----------------------------
1  1  1  0  1  1  0  0  1  0   (-78)

                     ⇓

               0  1  1  0  1
               0    -1    -2
               --------------
1  1  1  1  1  0  0  1  1  0
1  1  1  1  0  0  1  1
0  0  0  0  0  0
-----------------------------
1  1  1  0  1  1  0  0  1  0   (-78)
```

Carry-Save Addition (CSA) of Summands

Carry save addition speeds up the addition process. In CSA, instead of letting the carries ripple along the rows, they can be *saved* and introduced into next row, at correct weighted positions. The full adder is input with three partial bit products in the first row.

- Multiplication requires the addition of several summands.
- CSA speeds up the addition process.
- Consider the array for 4x4 multiplication shown in fig(1).
- First row consisting of just the AND gates that implement the bit products m3q0,m2q0,m1q0 and m0q0 .



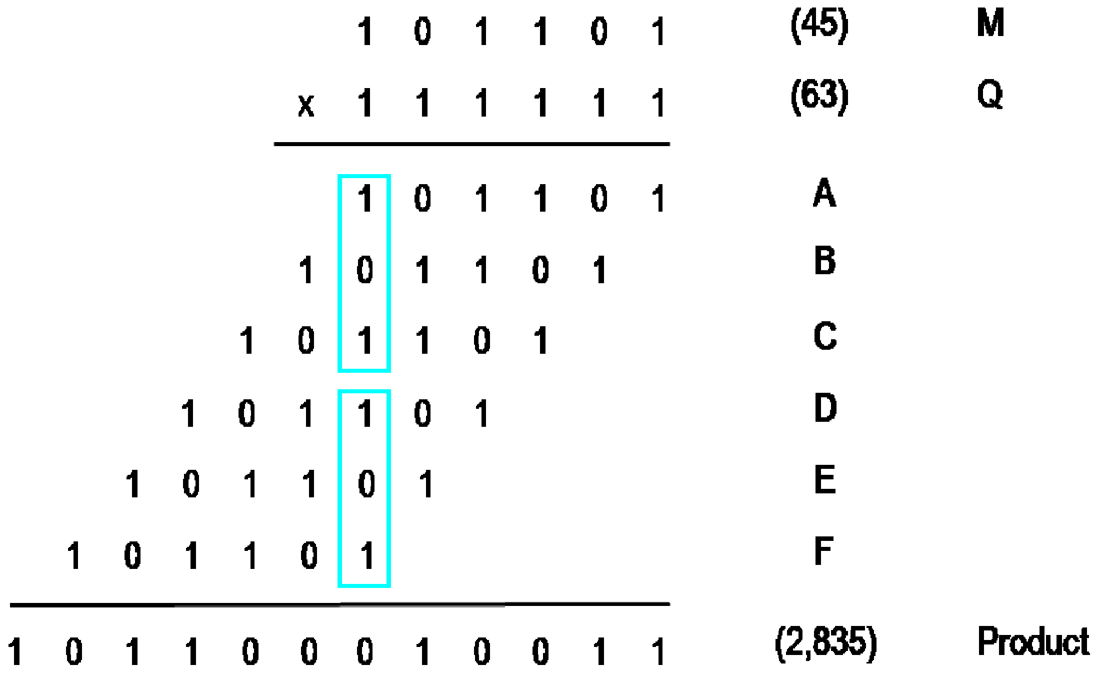
(a) Ripple-carry array (Figure 6.6 structure)

(b) Carry-save array

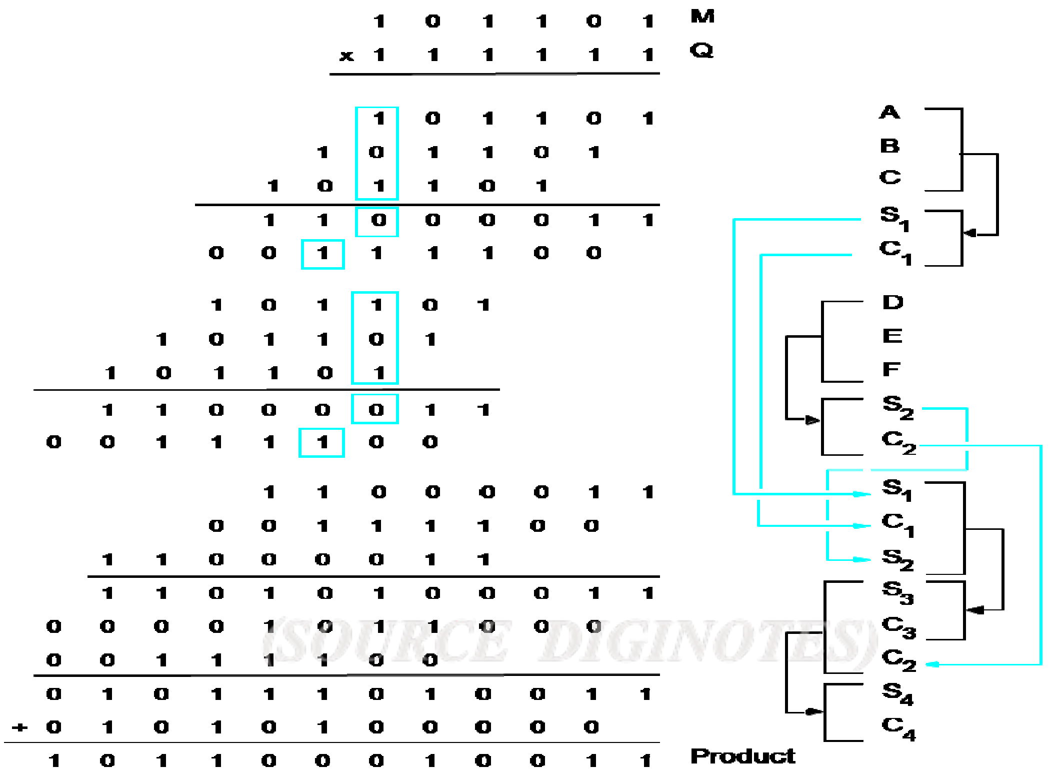
Figure 6.16. Ripple-carry and carry-save arrays for the multiplication operation M x Q = P for 4-bit operands.

- The delay through the carry-save array is somewhat less than delay through the ripple-carry array. This is because the S and C vector outputs from each row are produced in parallel in one full-adder delay.
- Consider the addition of many summands, we can:
- Group the summands in threes and perform carry-save addition on each of these groups in parallel to generate a set of S and C vectors in one full-adder delay
- Group all of the S and C vectors into threes, and perform carry-save addition on them, generating a further set of S and C vectors in one more full-adder delay
- Continue with this process until there are only two vectors remaining

- They can be added in a RCA or CLA to produce the desired product.



```
                           1 0 1 1 0 1    (45)     M
                        x  1 1 1 1 1 1    (63)     Q
                          ------------
                           1 0 1 1 0 1    A
                         1 0 1 1 0 1      B
                       1 0 1 1 0 1        C
                     1 0 1 1 0 1          D
                   1 0 1 1 0 1            E
                 1 0 1 1 0 1              F
                 -----------------------
                 1 0 1 1 0 0 0 1 0 0 1 1  (2,835)  Product
```



```
                 1 0 1 1 0 1    M
              x  1 1 1 1 1 1    Q
                 -----------
                 1 0 1 1 0 1
               1 0 1 1 0 1
             1 0 1 1 0 1
         ---------------
             1 1 0 0 0 0 1 1
           0 0 1 1 1 1 0 0

           1 0 1 1 0 1
         1 0 1 1 0 1
       1 0 1 1 0 1
     -----------------
       1 1 0 0 0 0 1 1
     0 0 1 1 1 1 0 0

             1 1 0 0 0 0 1 1
           0 0 1 1 1 1 0 0
       1 1 0 0 0 0 1 1
       ---------------------
       1 1 0 1 0 1 0 0 0 1 1
     0 0 0 0 1 0 1 1 0 0 0
     0 0 1 1 1 1 0 0
     -----------------------
     0 1 0 1 1 1 0 1 0 0 1 1
   + 0 1 0 1 0 1 0 0 0 0 0
     -----------------------
     1 0 1 1 0 0 0 1 0 0 1 1   Product
```

- When the number of summands is large, the time saved is proportionally much greater.

- Delay: AND gate + 2 gate / CSA level + CLA gate delay, Eg., 6 bit number require 15 gate delay, array 6x6 require 6(n-1)-1 = 29 gate D.
- In general CSA takes 1.7 *log*2 k -1.7 levels of CSA to reduce k summands

Integer Division

Manual Division

```
      21                          10101
    ______               ____________________________
13 ) 274              1101 ) 100010010
     26                      1101
    _____                    ________
     14                       10000
     13                        1101
    _____                    ________
      1                          1110
                                 1101
                               ________
                                   1
```

Longhand Division Steps

- Position the divisor appropriately with respect to the dividend and performs a subtraction.
- If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed.
- If the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is repositioned for another subtraction.

Restoring Division

- Similar to multiplication circuit
- N-bit positive divisor is loaded into register M and an n-bit positive dividend is loaded into register Q at the start of the operation.
- Register A is set to 0
- After the division operation is complete, the n-bit quotient is in register Q and the remainder is in register A.
- The required subtractions are facilitated by using 2's complement arithmetic.
- The extra bit position at the left end of both A and M accomodates the sign bit during subtraction.
- Shift A and Q left one binary position
- Subtract M from A, and place the answer back in A
- If the sign of A is 1, set q0 to 0 and add M back to A (restore A); otherwise, set q0 to 1

- Repeat these steps *n* times

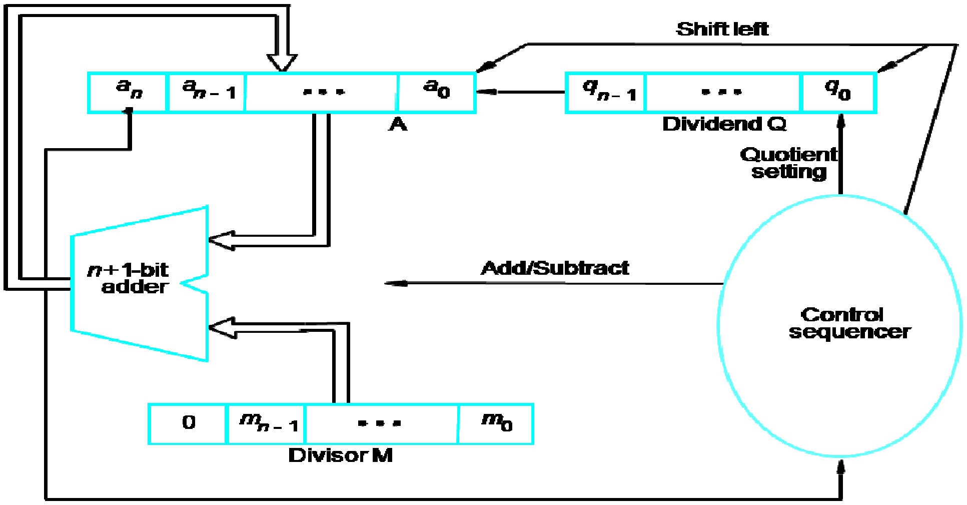
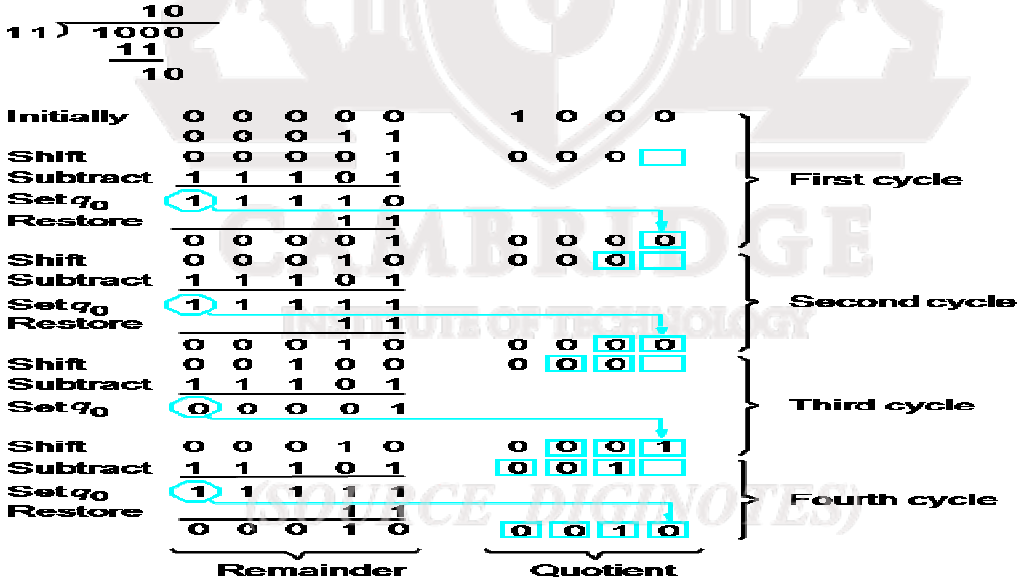


Figure 6.21. Circuit arrangement for binary division.

Example



```
          10
11 )    1000
         11
          10

Initially  0 0 0 0 0     1 0 0 0
           0 0 0 1 1
Shift      0 0 0 0 1     0 0 0 □
Subtract   1 1 1 0 1                 First cycle
Set q0     1 1 1 1 0
Restore          1 1
           0 0 0 0 1     0 0 0 0
Shift      0 0 0 1 0     0 0 0 □
Subtract   1 1 1 0 1                 Second cycle
Set q0     1 1 1 1 1
Restore          1 1
           0 0 0 1 0     0 0 0 0
Shift      0 0 1 0 0     0 0 0 □
Subtract   1 1 1 0 1                 Third cycle
Set q0     0 0 0 0 1

Shift      0 0 0 1 0     0 0 0 1
Subtract   1 1 1 0 1     0 0 1 □
Set q0     1 1 1 1 1                 Fourth cycle
Restore          1 1
           0 0 0 1 0     0 0 1 0
           Remainder     Quotient
```

Figure 6.22. A restoring-division example.

Non-restoring Division

- Restoring division algorithm can be improved by avoiding the need for restoring A after an unsuccessful subtraction
- Subtraction is said to be unsuccessful if the result is negative
- If A is positive, we shift left and subtract M that is we perform 2A-M.
- If A is negative, we restore it by performing A+M, and then we shift it left and subtract M.
- This is equivalent to performing 2A+M.
- Q0 is set to 0 or 1 after the correct operation has been performed.

Algorithm for Non Restoring Division

Step 1:(Repeat n times)

- If the sign of A is 0, shift A and Q left one bit position and subtract M from A; Otherwise , shift A and Q left and add M to A.
- Now if the sign of A id 0 set q0 to 1; otherwise , set q0 to 0

Step 2: If the sign of A is 1, add M to A

Example

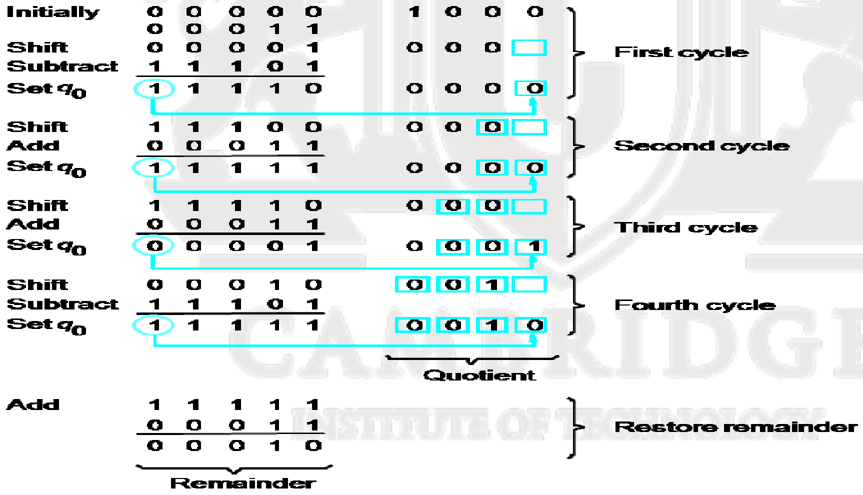


Figure 6.23. A nonrestoring-division example.

Comparision

- Needs restoring of reg A if the result of subtraction is –ve.

- In each cycle content of reg A is first shifted left and then divisor is subtracted from it

- Does not need restoring of remainder
- Slower algorithm

- Does not need restoring
- Needs restoring of remainder if remainder is –ve

- In each cycle the content of reg A is first shifted left and then the divisor is added or subtracted with the content of reg A depending on the sign of A
- Faster algorithm

Floating-Point Numbers and Operations

- So far we have dealt with fixed-point numbers and have considered them as integers.
- Floating-point numbers: the binary point is just to the right of the sign bit.
- In the 2’s complement system, the signed value F,represented n-bit binary fraction

  $B=b_0b_{-1}b_{-2}\ldots\ldots b_{-(n-1)}$

  $F(B)= -b_0\, x2^0+b_{-1}x2^{-1}+b_{-2}x2^{-2}\ldots\ldots\ldots b_{-(n-1)}x2^{-(n-1)}$

- Where the range of F is:$2^{-(n-1)} \leq F \leq 1-2^{-(n-1)}$
- The position of the binary point is variable and is automatically adjusted as computation proceeds.

- If n=32, then the value range is approximately

  $2^{(-31)} \leq F \leq 1-2^{-(31)}$ $(1-2.3283X10^{-10})$

- But this range is not sufficient to represent fractional numbers,
- To accommodate very large integers and very small fractions, a computer must be able to represent numbers and operate on them in such a way that the position of the binary point is variable and is automatically adjusted as computation proceeds.
- In this case the binary point is said to float, and the numbers are called floating point numbers.
- What are needed to represent a floating-point decimal number?
- It needs three fields
- Sign
- Mantissa (the significant digits)
- Exponent to an implied base (scale factor)

“Normalized” – the decimal point is placed to the right of the first (nonzero) significant digit

- Let us consider the number 111101.1000110 to be represented in floating point format.
- To represent the number in floating point format, first binary point is shifted to right of the first bit and the number is multiplied by the scaling factor to get the same value.
- The number is said to be Normalized form and is given as

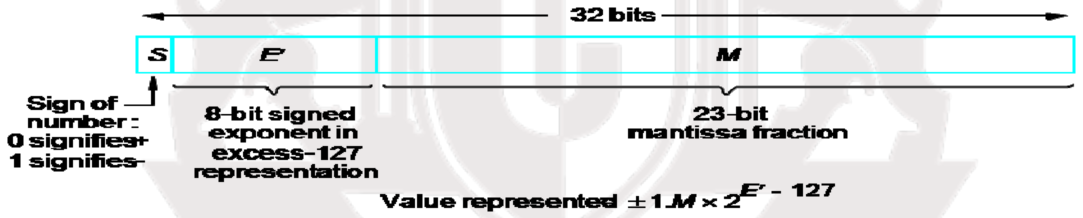
$$111101.1000110 \qquad \underbrace{1.11101100110}_{\text{Normalized form}} \times \underbrace{2^{\overset{\text{Exponent}}{5}}}_{\text{Scale factor}}$$

IEEE Standard for Floating-Point Numbers

Think about this number (all digits are decimal): $\pm X_1.X_2X_3X_4X_5X_6X_7 \times 10^{\pm Y1Y2}$.It is possible to approximate this mantissa precision and scale factor range in a binary representation that occupies 32 bits: 24-bit mantissa (1 sign bit for signed number), 8-bit exponent.

Instead of the signed exponent, E, the value actually stored in the exponent field is an unsigned integer E'=E+127, so called excess-127 format.

Single Precision


32 bits
S
E'
M
Sign of number : 0 signifies + 1 signifies -
8-bit signed exponent in excess-127 representation
23-bit mantissa fraction
Value represented ±1.M × 2^(E' - 127)

(a) Single precision

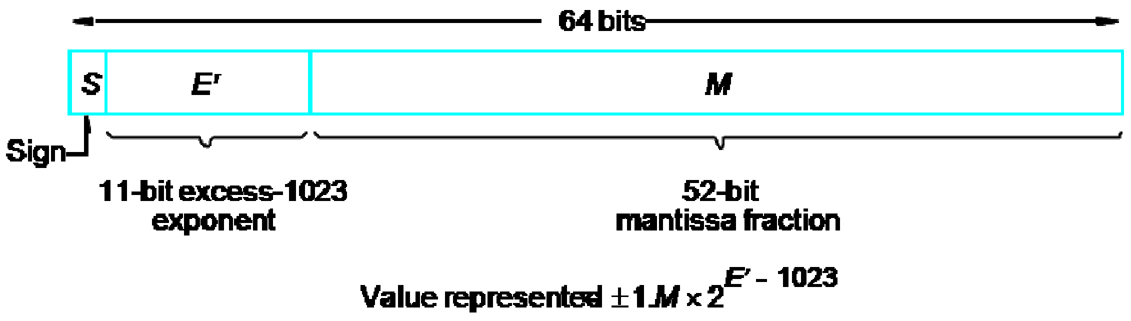
0	00101000	001010 - - - 0

Value represented $1.001010..0 \times 2^{-87}$

(b) Example of a single-precision number

$101000)_2 = 40_{10}$; $40 - 127 = -87$

Double Precision



64 bits
S
E'
M
Sign
11-bit excess-1023 exponent
52-bit mantissa fraction
Value represented ±1.M × 2^(E' - 1023)

(c) Double precision

Problem

1)Represent 1259.125_{10} in single precision and double precision formats

- Step 1 :Convert decimal number to binary format

  $1259_{(10)}$=$10011101011_{(2)}$

  Fractional Part

  0.125 $_{(10)}$=0.001
- Binary number = 10011101011+0.001

  =10011101011.001

Step 2:Normalize the number

10011101011.001=1.0011101011001 x 2^{10}

Step3:Single precision format:

For a given number S=0,E=10 and M=0011101011001

Bias for single precision format is = 127

E'=E+127=10+127=137 $_{(10)}$

=10001001 $_{(2)}$

- Number in single precision format

$$\underbrace{0}_{\text{Sign}}\ \underbrace{10001001}_{\text{Exponent}}\ \underbrace{0011101011001\ldots.0}_{\text{Mantissa(23 bit)}}$$

Step 4:Double precision format:

For a given number S=0,E=10 and M=0011101011001

Bias for double precision format is = 1023

E'=E+1023=10+1023=1033 $_{(10)}$

=10000001001 $_{(2)}$

- Number in double precision format is given as

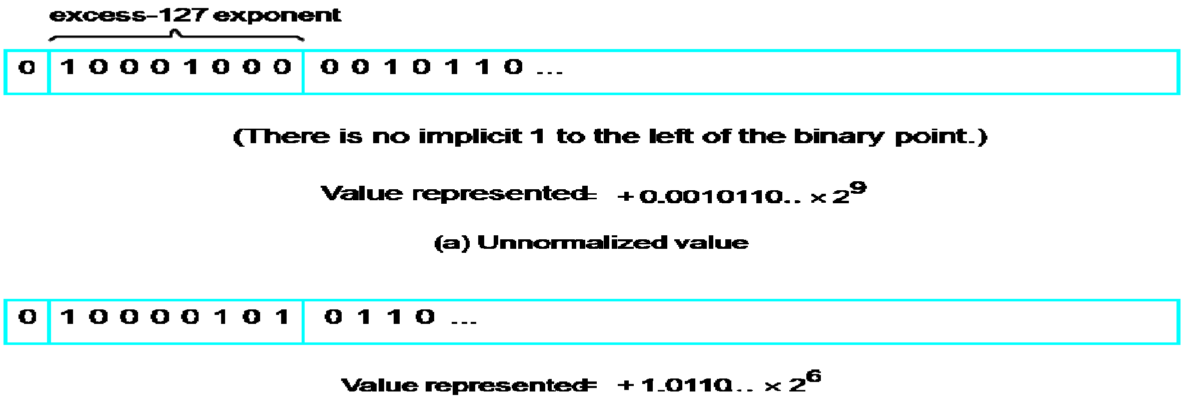
$$\underbrace{0}_{\text{Sign}}\ \underbrace{10001001}_{\text{Exponent}}\ \underbrace{0011101011001\ldots.0}_{\text{Mantissa(23 bit)}}$$

IEEE Standard

- For excess-127 format, 0 ≤ E’ ≤ 255. However, 0 and 255 are used to represent special value. So actually 1 ≤ E’ ≤ 254. That means -126 ≤ E ≤ 127.
- Single precision uses 32-bit. The value range is from 2^{-126} to 2^{+127}.
- Double precision used 64-bit. The value range is from 2^{-1022} to 2^{+1023}.

Normalization

- If a number is not normalized, it can always be put in normalized form by shifting the fraction and adjusting the exponent. As computations proceed, a number that does not fall in the representable range of normal numbers might be generated.
- In single precision, it requires an exponent less than -126 (underflow) or greater than +127 (overflow). Both are exceptions that need to be considered.


excess-127 exponent
0
1 0 0 0 1 0 0 0
0 0 1 0 1 1 0 ...
(There is no implicit 1 to the left of the binary point.)
Value represented = +0.0010110.. × 2^9
0
1 0 0 0 0 1 0 1
0 1 1 0 ...
Value represented = +1.0110.. × 2^6

(a) Unnormalized value

(b) Normalized version

Special Values

- The end value 0 and 255 are used to represent special values.
- When E’=0 and M=0, the value exact 0 is represented. (±0)
- When E’=255 and M=0, the value ∞ is represented. (± ∞)
- When E’=0 and M≠0, de normal numbers are represented. The value is ±0.M´2^{-126}. (allow for Gradual underflow)
- When E’=255 and M≠0, Not a Number (NaN).
- NaN is the result of performing an invalid operation, such as 0/0 or square root of -1.

Exceptions

- A processor must set exception flags if any of the following occur in performing operations: underflow, overflow, divide by zero, inexact, invalid.
- When exception occurs, the results are set to special values.

Arithmetic Operations on Floating-Point Numbers

Add/Subtract rule

1. Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.
2. Set the exponent of the result equal to the larger exponent.
3. Perform addition/subtraction on the mantissas and determine the sign of the result.
4. Normalize the resulting value, if necessary.

Subtraction of floating point numbers

- Similar process is used for subtraction
- Two mantissas are subtracted instead of addition
- Sign of greater mantissa is assigned to the result

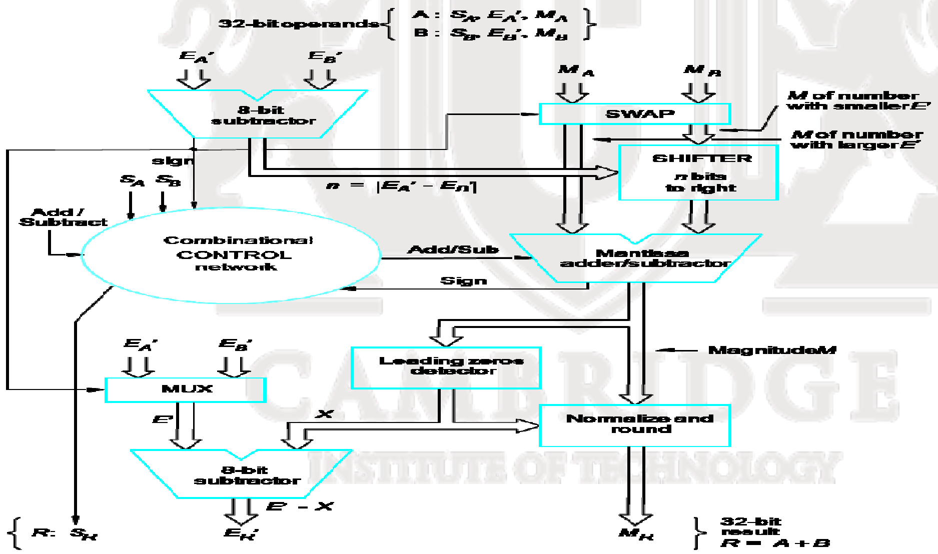


Figure 6.26. Floating-point addition-subtraction unit.

Step 1: Compare the exponent for sign bit using 8bit subtractor
Sign is sent to SWAP unit to decide on which number to be sent to SHIFTER unit.
Step2: The exponent of the result is determined in two way multiplexer depending on the sign bit from step1

Step3: Control logic determines whether mantissas are to be added or subtracted. Depending on sign of the operand. There are many combinations are possible here, that depends on sign bits, exponent values of the operand.
Step4: Normalization of the result depending on the leading zeros, and some special case like 1.xxxxx operands. Where result is 1x.xxx and X = -1, therefore will increase the exponent value.

Example

Add single precision floating point numbers A
and B, where A=44900000 H and B = 42A00000H.
Solution
Step 1 :Represent numbers in single precision format
A = 0 1000 1001 0010000….0
B = 0 1000 0101 0100000….0
Exponent for A = 1000 1001 =137
Therefore actual exponent = 137-127(Bias) =10
Exponent for B = 1000 0101 = 133
Therefore actual exponent = 133-127(Bias) = 6

With difference 4. Hence its mantissa is shifted right by 4 bits as shown below
Step 2:Shift mantissa
Shifted mantissa of B = 0 0 0 0 0 1 0 0…0
Step 3: Add mantissa
Mantissa of A = 00100000…0
Mantissa of B = 00000100…0
Mantissa of result = 00100100…0
As both numbers are positive, sign of the result is positive
Result =0100 0100 1001 0010 0…0
=44920000H

Multiply rule

- Add the exponents and subtract 127.
- Multiply the mantissas and determine the sign of the result.
- Normalize the resulting value, if necessary.

Divide rule

- Subtract the exponents and add 127.
- Divide the mantissas and determine the sign of the result.

Normalize the resulting value, if necessary

Guard Bits

- **During the intermediate steps, it is important to retain extra bits, often called guard bits, to yield the maximum accuracy in the final results.**
- **Removing the guard bits in generating a final result requires truncation of the extended mantissa.**

Truncation

- **Chopping – Remove the guard bits**

$0.b_{-1}b_{-2}b_{-3}000$ -- $0.b_{-1}b_{-2}b_{-3}111$à$0.b_{-1}b_{-2}b_{-3}$

Error ranges from 0 to 0.000111.

Chopping is biased because is not symmetrical

about 0, 0 to 1 at LSB.

- **Von Neumann Rounding - All 6-bit fractions with $b_{-4}b_{-5}b_6$ not equal to 000 are truncated to to $0.b_{-1}b_{-2}1$**
- **This truncation is unbiased, error ranges: -1 to +1 at LSB.**
- **unbiased rounding is better because positive error tend to offset negative errors as the computation proceeds.**

- **Rounding (A 1 is added to the LSB position of the bits to be retained if there is a 1 in the MSB position of the bits being removed) – unbiased, -½ to +½ at LSB.**

$0.b_{-1}b_{-2}b_{-3}1$ .... is rounded to $0.b_{-1}b_{-2}b_{-3}+ 0.001$

- **Round to the nearest number or nearest even number in case of a tie**

($0.b_{-1}b_{-2}0100$ -> $0.b_{-1}b_{-2}0$; $0.b_{-1}b_{-2}1100$ -> $0.b_1b_21+0.001$)

- **Best accuracy**
- **Most difficult to implement**

Addition and Subtraction

Floating point addition is analogous to addition using scientific notation. For example, to add 2.25x $10^0$to 1.340625x 10^2:

1. Shift the decimal point of the smaller number to the left until the exponents are equal. Thus, the first number becomes .0225x 10^2.
2. Add the numbers with decimal points aligned:

```
     0.0225      x 10²
 +   1.340625    x 10²
 ----------------------
     1.363125    x 10²
```

3. Normalize the result.

Once the decimal points are aligned, the addition can be performed by ignoring the decimal point and using integer addition.

The addition of two IEEE FPS numbers is performed in a similar manner. The number 2.25 in IEEE FPS is:

```
S      E                   M
0   1000 0000   (1) 001 0000 0000 0000 0000 0000
```

The number 134.0625 in IEEE FPS is:

```
S      E                   M
0   1000 0110   (1) 000 0110 0001 0000 0000 0000
```

1. To align the binary points, the smaller exponent is incremented and the mantissa is shifted right until the exponents are equal. Thus, 2.25 becomes:

```
S      E                   M
0   1000 0110   (0) 000 0010 0100 0000 0000 0000
```

2. The mantissas are added using integer addition:

```
     S      E                   M
     0   1000 0110   (0) 000 0010 0100 0000 0000 0000
 +   0   1000 0110   (1) 000 0110 0001 0000 0000 0000
 -----------------------------------------------------
     0   1000 0110   (1) 000 1000 0101 0000 0000 0000
```

3. The result is already in normal form. If the sum overflows the position of the hidden bit, then the mantissa must be shifted one bit to the right and the exponent incremented. The mantissa is always less than 2, so the hidden bits can sum to no more than 3 (11).

The exponents can be positive or negative with no change in the algorithm. A smaller exponent means more negative. In the bias-127 representation, the smaller exponent has the smaller value for E, the unsigned interpretation.

An important case occurs when the numbers differ widely in magnitude. If the exponents differ by more than 24, the smaller number will be shifted right entirely out of the mantissa field, producing a zero mantissa. The sum will then equal the larger number. Such *truncation errors* occur when the numbers differ by a factor of more than 2^{24}, which is approximately $1.6 \cdot 10^{7}$. The precision of IEEE single precision floating point arithmetic is approximately 7 decimal digits.

Negative mantissas are handled by first converting to 2's complement and then performing the addition. After the addition is performed, the result is converted back to sign-magnitude form.

When adding numbers of opposite sign, cancellation may occur, resulting in a sum which is arbitrarily small, or even zero if the numbers are equal in magnitude. Normalization in this case may require shifting by the total number of bits in the mantissa, resulting in a large loss of accuracy.

When the mantissa of the sum is zero, no amount of shifting will produce a 1 in the hidden bit. This case must be detected in the normalization step and the result set to the representation for 0, E = M = 0. This result does not mean the numbers are equal; only that their difference is smaller than the precision of the floating point representation.

Floating point subtraction is achieved simply by inverting the sign bit and performing addition of signed mantissas as outlined above.

Multiplication

The multiplication of two floating point numbers is analogous to multiplication in scientific notation. For example, to multiply 1.8x 10^{1}times 9.5x 10^{0}:

1. Perform unsigned integer multiplication of the mantissas. The decimal point in the sum is positioned so that the number of decimal places equals the sum of the number of decimal places in the numbers.
2. 1.8
3. x 9.5
4. -----

   17.10

5. Add the exponents:
6. 1
7. + 0
8. ---

   1

9. Normalize the result:

$$17.10 \cdot 10^{1} = 1.710 \cdot 10^{2}.$$

10. Set the sign of the result.

The multiplication of two IEEE FPS numbers is performed similarly. The number 18.0 in IEEE FPS format is:

```
S      E                 M
0   1000 0011   (1) 001 0000 0000 0000 0000 0000
```

The number 9.5 in IEEE FPS format is:

```
S      E                 M
0   1000 0010   (1) 001 1000 0000 0000 0000 0000
```

1. The product of the 24 bit mantissas produces a 48 bit result with 46 bits to the right of the binary point:

   $$01.01010110000000\ldots0000$$

   Truncated to 24 bits with the hidden bit in (), the mantissa is:

   $$(1).01010110000000000000000.$$

2. The biased-127 exponents are added. Addition in biased-127 representation can be performed by 2's complement with an additional bias of -127 since:

   $$\mathbf{E} = (\mathbf{e}_1 + \mathbf{e}_2) + 127 = (\mathbf{e}_1 + 127) + (\mathbf{e}_2 + 127) - 127 = (\mathbf{E}_1 + \mathbf{E}_2) - 127.$$

   The sum of the exponents is:

   ```
       E
     1000 0011 (4)
   + 1000 0010 (3)
   -----------
     0000 0101
   + 1000 0001 (-127)
   -----------
     1000 0110 (+7)
   ```

3. The mantissa is already in normal form. If the position of the hidden bit overflows, the mantissa must be shifted right and the exponent incremented.
4. The sign of the result is the xor of the sign bits of the two numbers.

When the fields are assembled in IEEE FPS format, the result is:

```
S      E                 M
0   1000 0110   (1) 010 1011 0000 0000 0000 0000
```

Rounding occurs in floating point multiplication when the mantissa of the product is reduced from 48 bits to 24 bits. The least significant 24 bits are discarded.

Overflow occurs when the sum of the exponents exceeds 127, the largest value which is defined in bias-127 exponent representation. When this occurs, the exponent is set to 128 (E = 255) and the mantissa is set to zero indicating + or - infinity.

Underflow occurs when the sum of the exponents is more negative than -126, the most negative value which is defined in bias-127 exponent representation. When this occurs, the exponent is set to -127 (E = 0). If M = 0, the number is exactly zero.

If M is not zero, then a *denormalized* number is indicated which has an exponent of -127 and a hidden bit of 0. The smallest such number which is not zero is 2^{-149}. This number retains only a single bit of precision in the rightmost bit of the mantissa.

Basic processing Unit

Chapter Objectives

- How a processor executes instructions
- Internal functional units and how they are connected
- Hardware for generating internal control signals
- The micro programming approach
- Micro program organization

Fundamental Concepts

- Processor fetches one instruction at a time, and perform the operation specified.
- Instructions are fetched from successive memory locations until a branch or a jump instruction is encountered.
- Processor keeps track of the address of the memory location containing the next instruction to be fetched using Program Counter (PC).
- Instruction Register (IR)

Executing an Instruction

- Fetch the contents of the memory location pointed to by the PC. The contents of this location are loaded into the IR (fetch phase).

  IR ← [[PC]]
- Assuming that the memory is byte addressable, increment the contents of the PC by 4 (fetch phase).

  PC ← [PC] + 4
- Carry out the actions specified by the instruction in the IR (execution phase).

Processor Organization

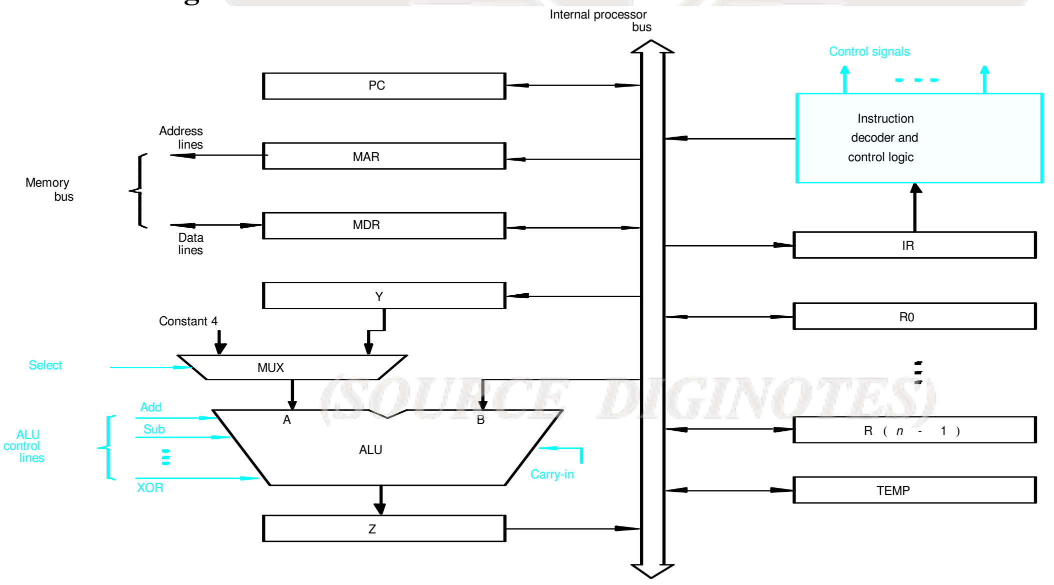


Figure 7.1. Single-bus organization of the datapath inside a processor.

- ALU and all the registers are interconnected via a single common bus.
- The data and address lines of the external memory bus connected to the internal processor bus via the memory data register, MDR, and the memory address register, MAR respectively.
- Register MDR has two inputs and two outputs.
- Data may be loaded into MDR either from the memory bus or from the internal processor bus.
- The data stored in MDR may be placed on either bus.
- The input of MAR is connected to the internal bus, and its output is connected to the external bus.
- The control lines of the memory bus are connected to the instruction decoder and control logic.
- This unit is responsible for issuing the signals that control the operation of all the units inside the processor and for increasing with the memory bus.
- The MUX selects either the output of register Y or a constant value 4 to be provided as input A of the ALU.
- The constant 4 is used to increment the contents of the program counter.

Register Transfers

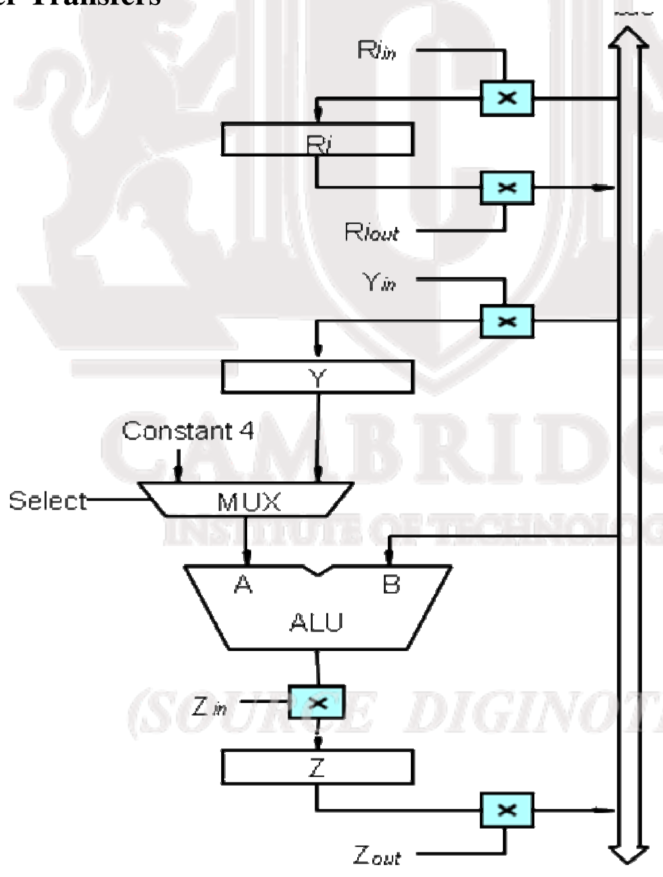


Figure 7.2. Input and output gating for the registers in Figure 7.1.

- Instruction execution involves a sequence of steps in which data are transferred from one register to another.
- For each register two control signals are used to place the contents of that register on the bus or to load the data on the bus into register.(in figure)
- The input and output of register Ri_{in} and Ri_{out} is set to 1, the data on the bus are loaded into R_i.
- Similarly, when Ri_{out} is set to 1, the contents of register Ri are placed on the bus.
- While Ri_{out} is equal to 0, the bus can be used for transferring data from other registers.

Example

- Suppose we wish to transfer the contents of register R1 to register R4. This can be accomplished as follows.
- Enable the output of registers R1 by setting R1out to 1. This places the contents of R1 on the processor bus.
- Enable the input of register R4 by setting R4out to 1. This loads data from the processor bus into register R4.
- All operations and data transfers with in the processor take place with in time periods defined by the processor clock.
- The control signals that govern a particular transfer are asserted at the start of the clock cycle.

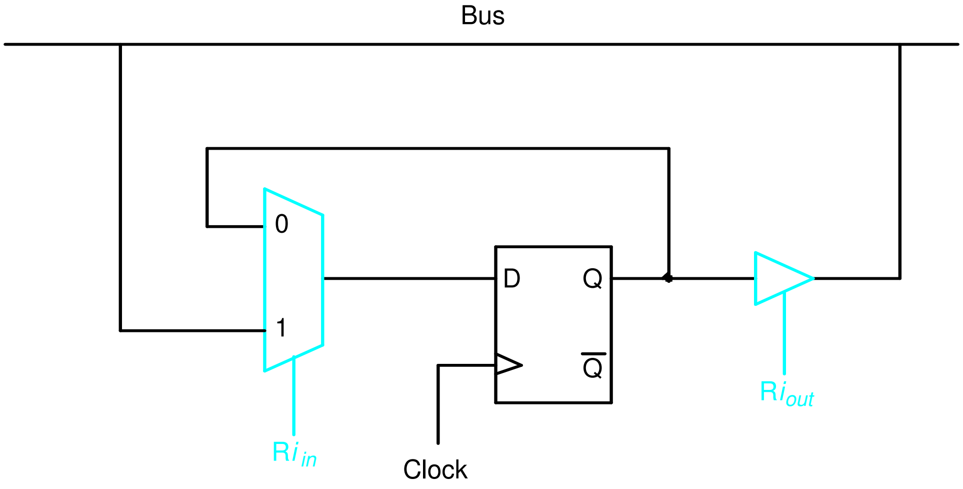


Figure 7.3. Input and output g ating for one register bit.

Performing an Arithmetic or Logic Operation

- The ALU is a combinational circuit that has no internal storage.
- ALU gets the two operands from MUX and bus. The result is temporarily stored in register Z.
- What is the sequence of operations to add the contents of register R1 to those of R2 and store the result in R3?
  - R1out, Yin

- R2out, SelectY, Add, Zin
- Zout, R3in

- All other signals are inactive.
- In step 1, the output of register R1 and the input of register Y are enabled, causing the contents of R1 to be transferred over the bus to Y.
- Step 2, the multiplexer's select signal is set to Select Y, causing the multiplexer to gate the contents of register Y to input A of the ALU.
- At the same time, the contents of register R2 are gated onto the bus and, hence, to input B.
- The function performed by the ALU depends on the signals applied to its control lines.
- In this case, the ADD line is set to 1, causing the output of the ALU to be the sum of the two numbers at inputs A and B.
- This sum is loaded into register Z because its input control signal is activated.
- In step 3, the contents of register Z are transferred to the destination register R3. This last transfer cannot be carried out during step 2, because only one register output can be connected to the bus during any clock cycle.

Fetching a Word from Memory

- The processor has to specify the address of the memory location where this information is stored and request a Read operation.
- This applies whether the information to be fetched represents an instruction in a program or an operand specified by an instruction.
- The processor transfers the required address to the MAR, whose output is connected to the address lines of the memory bus.

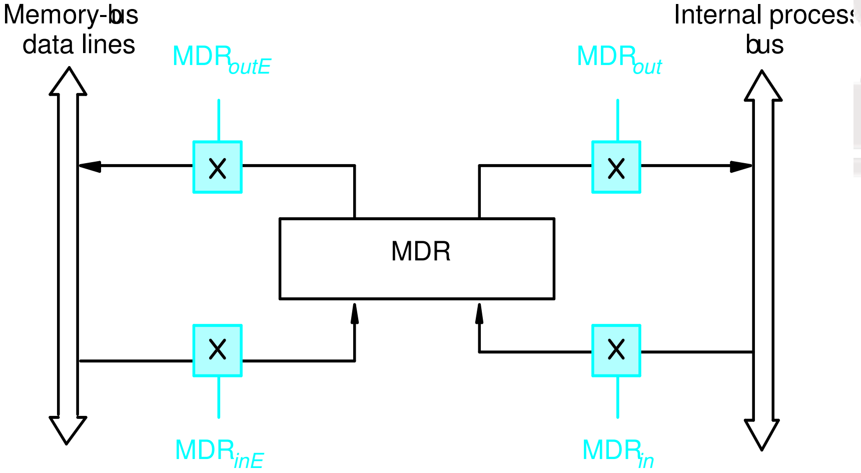


Figure 7.4. Connection and control signals for register MDR.

- At the same time , the processor uses the control lines of the memory bus to indicate that a Read operation is needed.
- When the requested data are received from the memory they are stored in register MDR, from where they can be transferred to other registers in the processor.

- The response time of each memory access varies (cache miss, memory-mapped I/O,…).
- To accommodate this, the processor waits until it receives an indication that the requested operation has been completed (Memory-Function-Completed, MFC).
- **Move (R1), R2**

  MAR ← [R1]
  Start a Read operation on the memory bus
  Wait for the MFC response from the memory
  Load MDR from the memory bus
  R2 ← [MDR]

- The output of MAR is enabled all the time.
- Thus the contents of MAR are always available on the address lines of the memory bus.
- When a new address is loaded into MAR, it will appear on the memory bus at the beginning of the next clock cycle.(in fig)
- A read control signal is activated at the same time MAR is loaded.
- This means memory read operations requires three steps, which can be described by the signals being activated as follows

  $R1_{out}$,MAR_{in},Read
  MDR_{inE},WMFC
  MDR_{out},$R2_{in}$

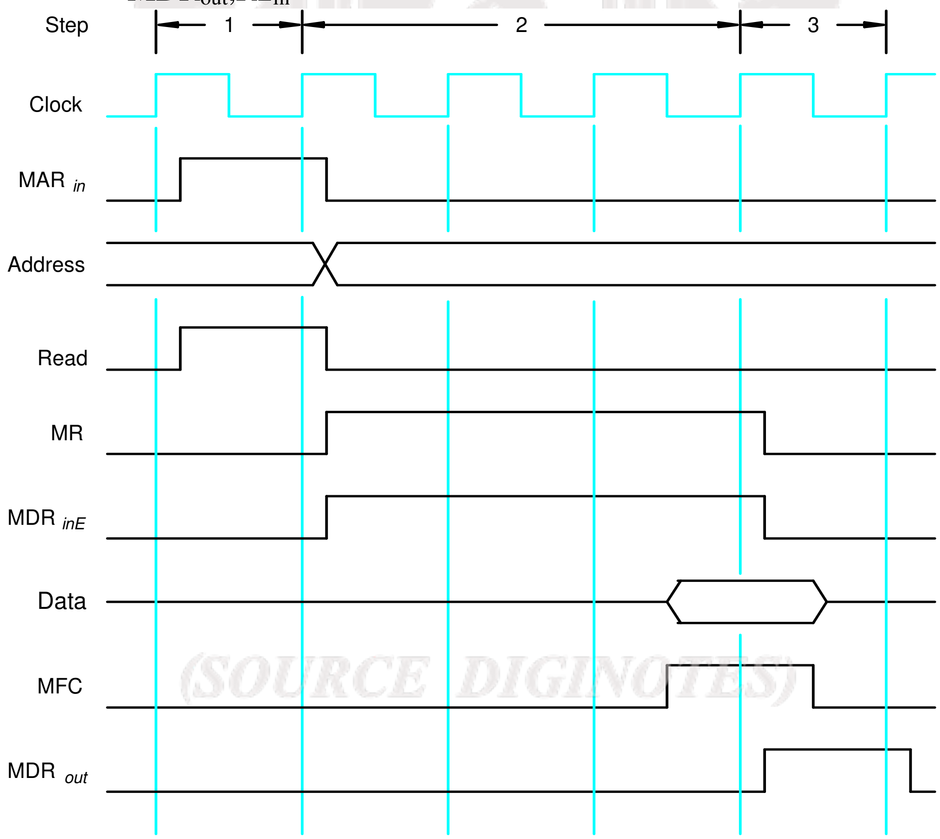


Figure 7.5. Timing of a memory Read operation.

Storing a word in Memory

- Writing a word into a memory location follows a similar procedure.
- The desired address is loaded into MAR.
- Then , the data to be written are loaded into MDR, and a write command is issued.

Example

- Executing the instruction
- Move R2,(R1) requires the following steps
  - **1 $R1_{out}$,MAR_{in}**
  - **2.$R2_{out}$,MDR_{in},Write**
  - **3.MDR_{outE},WMFC**

Execution of a Complete Instruction

- Add (R3), R1
- Fetch the instruction
- Fetch the first operand (the contents of the memory location pointed to by R3)
- Perform the addition
- Load the result into R1

Step	Action
1	PC_{out} , MAR_{in} , Read, Select4, Add, Z_{in}
2	Z_{out} , PC_{in} , Y_{in} , WMFC
3	MDR_{out} , IR_{in}
4	$R3_{out}$, MAR_{in} , Read
5	$R1_{out}$, Y_{in} , WMFC
6	MDR_{out} , SelectY, Add, Z_{in}
7	Z_{out} , $R1_{in}$, End

Figure 7.6. Control sequence for execution of the instruction Add (R3),R1.

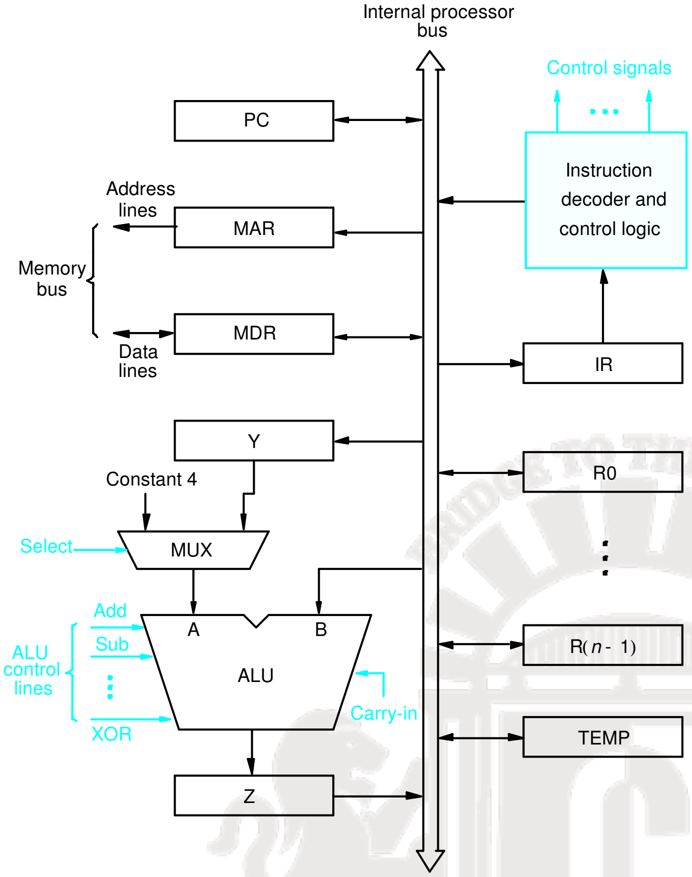


Figure 7.1. Single-bus organization of the datapath inside a proc

Execution of Branch Instructions

- A branch instruction replaces the contents of PC with the branch target address, which is usually obtained by adding an offset X given in the branch instruction.
- The offset X is usually the difference between the branch target address and the address immediately following the branch instruction.
- Conditional branch

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4,Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMF C
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, Add, Z_{in}
5	Z_{out}, PC_{in}, End

Figure 7.7. Control sequence for an unconditional branch instruction.

Multiple-Bus Organization

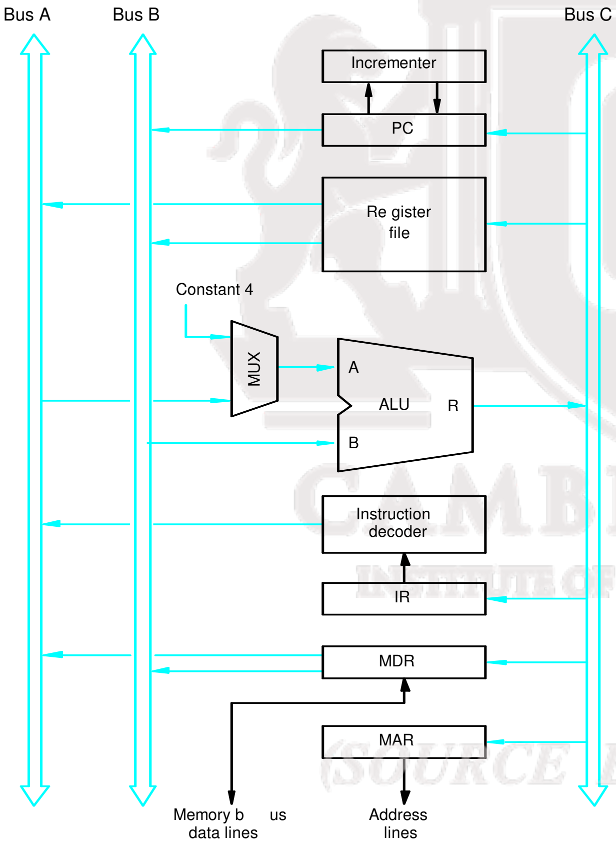
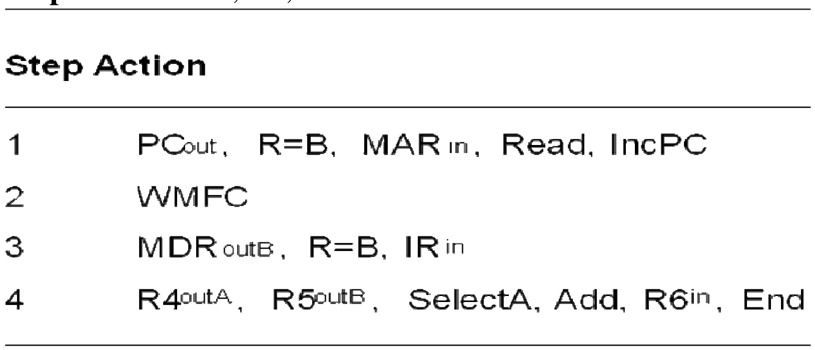


Figure 7.8. Three-b us or g anization of the datapath.

Example : Add R4, R5, R6



Step	Action
1	PC_{out}, R=B, MAR_{in}, Read, IncPC
2	WMFC
3	MDR_{outB}, R=B, IR_{in}
4	$R4_{outA}$, $R5_{outB}$, SelectA, Add, $R6_{in}$, End

Figure 7.9. Control sequence for the instruction. Add R4,R5,R6, for the three-bus organization in Figure 7.8.

Hardwired Control

- To execute instructions, the processor must have some means of generating the control signals needed in the proper sequence.
- Two categories: hardwired control and micro programmed control
- Hardwired system can operate at high speed; but with little flexibility.

Control Unit Organization

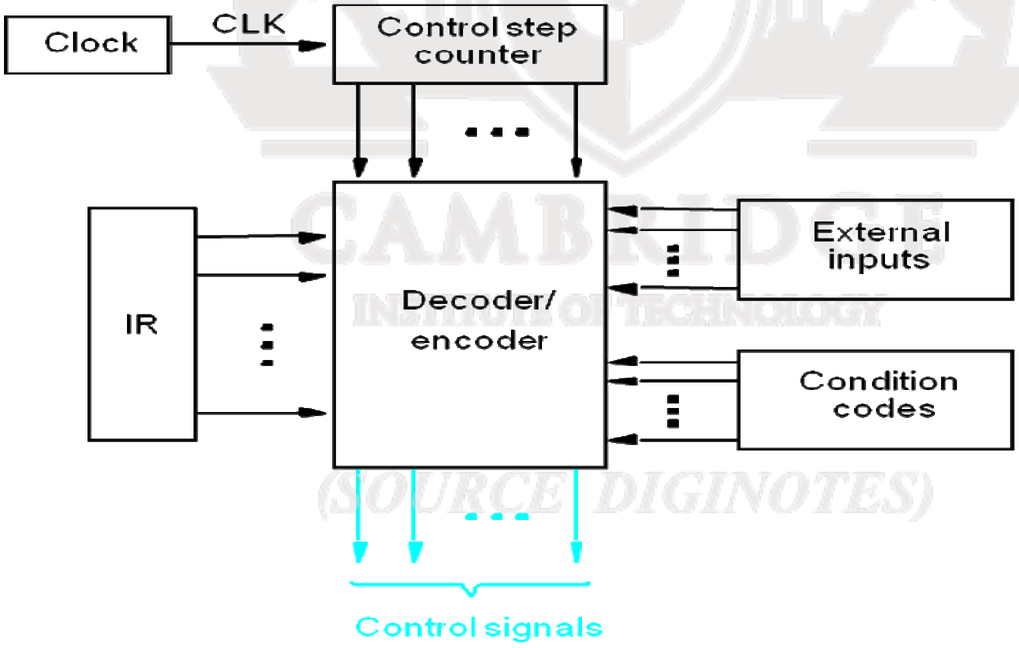


Figure 7.10. Control unit organization.

n

Detailed Control design

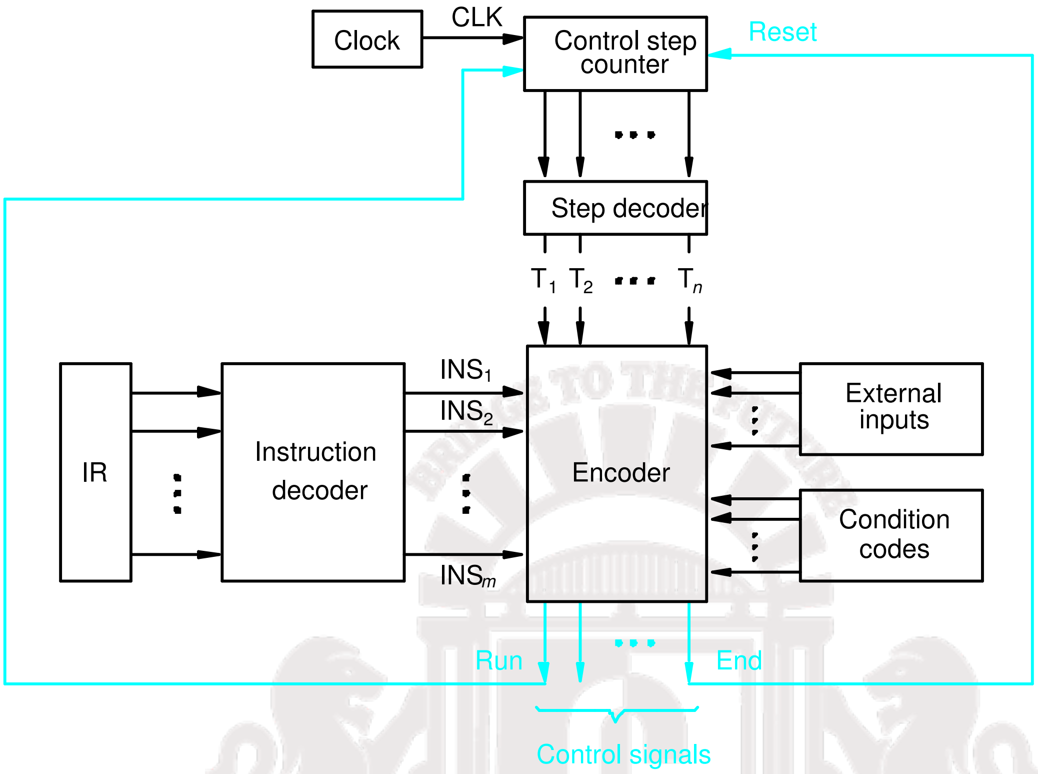


Figure 7.11. Separation of the decoding and encoding functio

Generating Z_{in}

- $Z_{in} = T_1 + T_6 \cdot ADD + T_4 \cdot BR + \ldots$

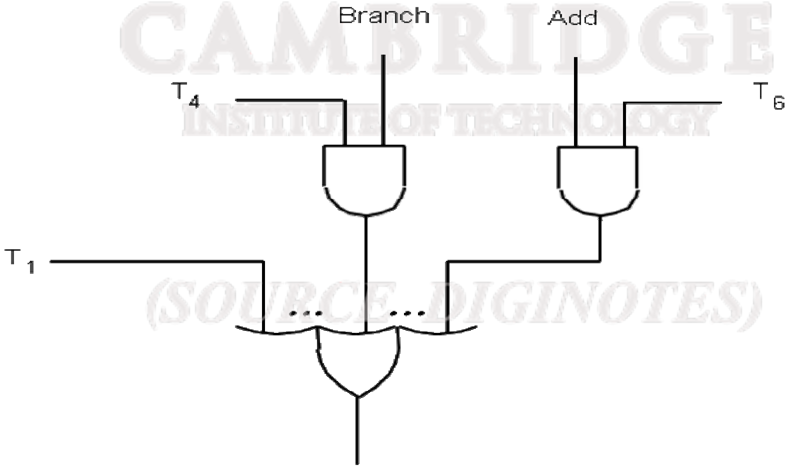


Figure 7.12. Generation of the Z_{in} control signal for the processor in Figure 7.1.

Generating End

- **End = T_7 • ADD + T_5 • BR + (T_5 • N + T_4 • N) • BRN +…**

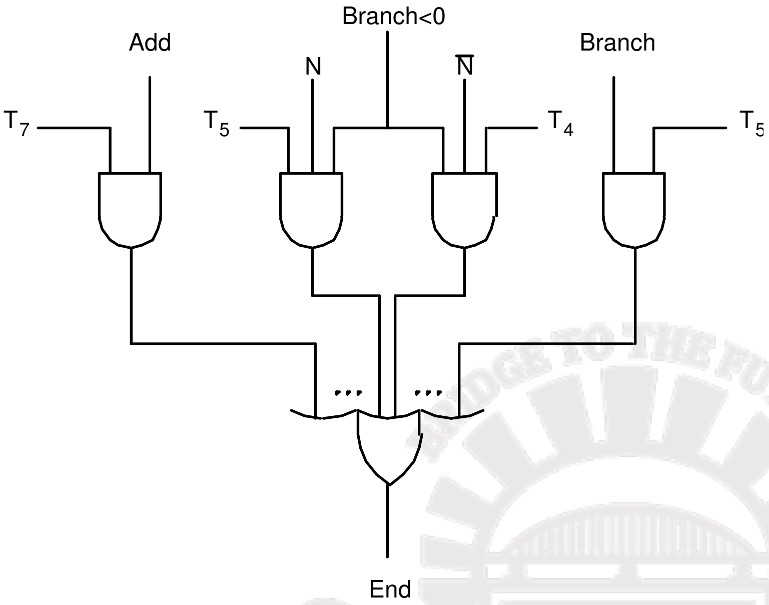


Figure 7.13. Generation of the End control signal.

A Complete Processor

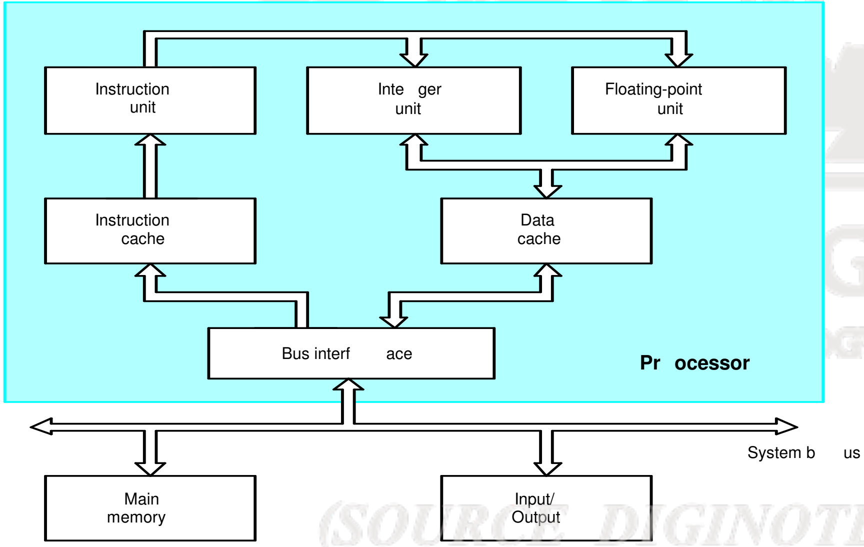


Figure 7.14. Block diagram of a complete processor .

Microprogrammed Control

- Control signals are generated by a program similar to machine language programs.
- Control Word (CW); microroutine; microinstruction

Micro - instruction	..	PC_{in}	PC_{out}	MAR_{in}	Read	MDR_{out}	IR_{in}	Y_{in}	Select	Add	Z_{in}	Z_{out}	$R1_{out}$	$R1_{in}$	$R3_{out}$	WMFC	End	..
1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	
2		1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	
4		0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	
5		0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	
6		0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	
7		0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	

Figure 7.15An example of microinstructions for Figure 7.6

Step	Action
1	PC_{out} , MAR_{in} , Read, Select4, Add, Z_{in}
2	Z_{out} , PC_{in} , Y_{in} , WMFC
3	MDR_{out} , IR_{in}
4	$R3_{out}$, MAR_{in} , Read
5	$R1_{out}$, Y_{in} , WMFC
6	MDR_{out} , SelectY, Add, Z_{in}
7	Z_{out} , $R1_{in}$, End

Figure 7.6. Control sequence for execution of the instruction Add (R3),R1.

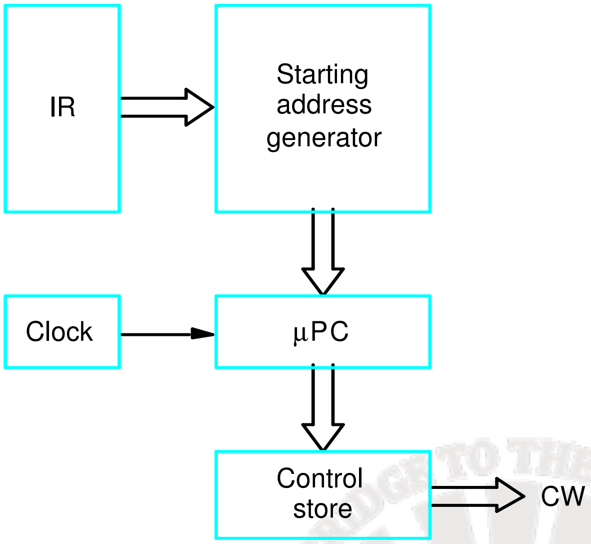
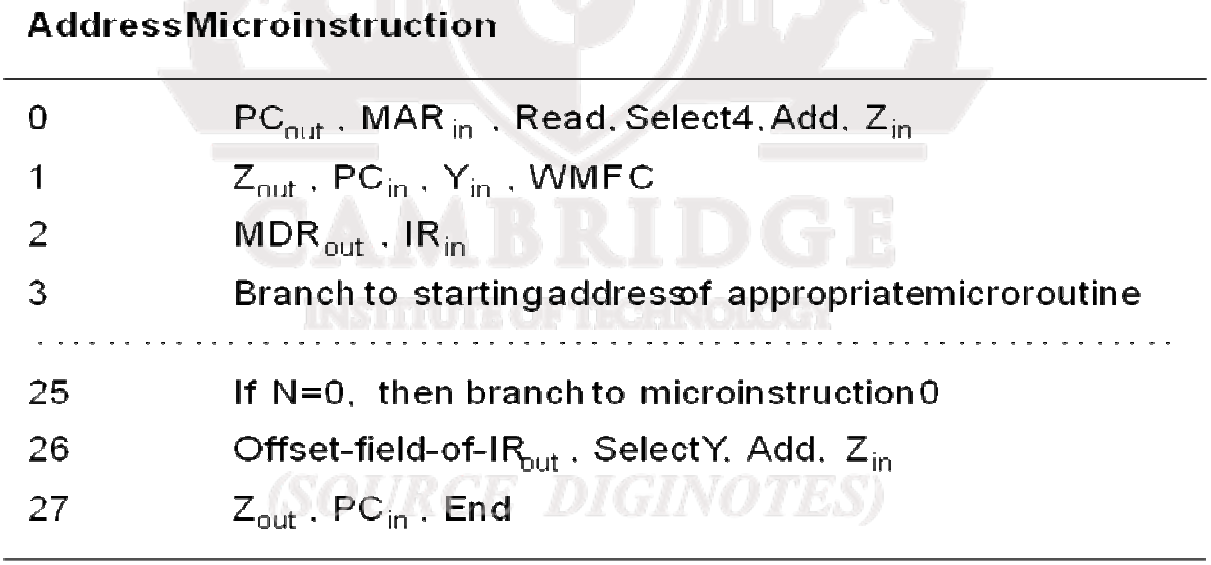


Figure 7.16. Basic organization of a microprogrammed control unit.

- The previous organization cannot handle the situation when the control unit is required to check the status of the condition codes or external inputs to choose between alternative courses of action.
- Use conditional branch microinstruction.



Address	Microinstruction
0	PC_{out} , MAR_{in} , Read, Select4, Add, Z_{in}
1	Z_{out} , PC_{in} , Y_{in} , WMFC
2	MDR_{out} , IR_{in}
3	Branch to starting address of appropriate microroutine
25	If N=0, then branch to microinstruction 0
26	Offset-field-of-IR_{out} , SelectY, Add, Z_{in}
27	Z_{out} , PC_{in} , End

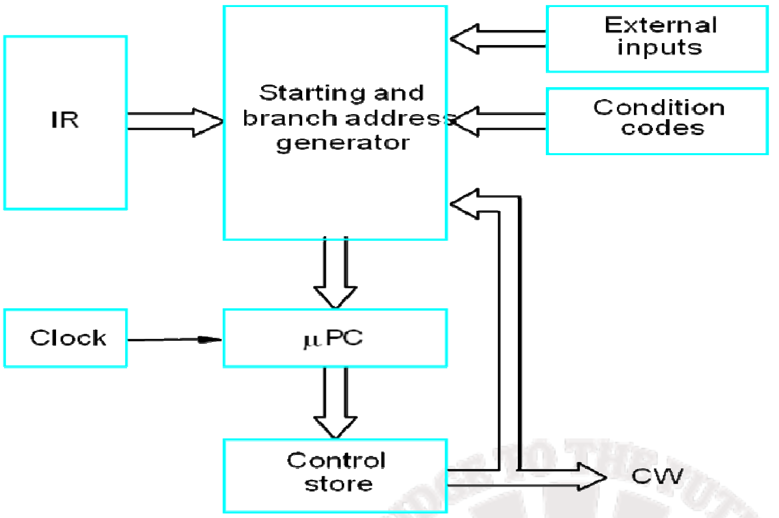


Figure 7.18. Organization of the control unit to allow conditional branching in the microprogram.

Microinstructions

- A straightforward way to structure microinstructions is to assign one bit position to each control signal.
- However, this is very inefficient.
- The length can be reduced: most signals are not needed simultaneously, and many signals are mutually exclusive.
- All mutually exclusive signals are placed in the same group in binary coding.

Microinstruction

F1	F2	F3	F4	F5

F1 (4 bits)	F2 (3 bits)	F3 (3 bits)	F4 (4 bits)	F5 (2 bits)
0000: No transfer	000: No transfer	000: No transfer	0000: Add	00: No action
0001: PC$_{out}$	001: PC$_{in}$	001: MAR$_{in}$	0001: Sub	01: Read
0010: MDR$_{out}$	010: IR$_{in}$	010: MDR$_{in}$	⋮	10: Write
0011: Z$_{out}$	011: Z$_{in}$	011: TEMP$_{in}$	1111: XOR	
0100: R0$_{out}$	100: R0$_{in}$	100: Y$_{in}$	16 ALU functions	
0101: R1$_{out}$	101: R1$_{in}$			
0110: R2$_{out}$	110: R2$_{in}$			
0111: R3$_{out}$	111: R3$_{in}$			
1010: TEMP$_{out}$				
1011: Offset$_{out}$				

F6	F7	F8	• • •

F6 (1 bit)	F7 (1 bit)	F8 (1 bit)
0: SelectY 1: Select4	0: No action 1: WMFC	0: Continue 1: End

Figure 7.19. An example of a partial format for field-encoded microinstructions.

Further Improvement

- Enumerate the patterns of required signals in all possible microinstructions. Each meaningful combination of active control signals can then be assigned a distinct code.
- Vertical organization
- Horizontal organization

Micro program Sequencing

- If all micro programs require only straightforward sequential execution of microinstructions except for branches, letting a μPC governs the sequencing would be efficient.
- However, two disadvantages:
  - Having a separate micro routine for each machine instruction results in a large total number of microinstructions and a large control store.
  - Longer execution time because it takes more time to carry out the required branches.
- Example: Add src, Rdst
- Four addressing modes: register, autoincrement, autodecrement, and indexed (with indirect forms).

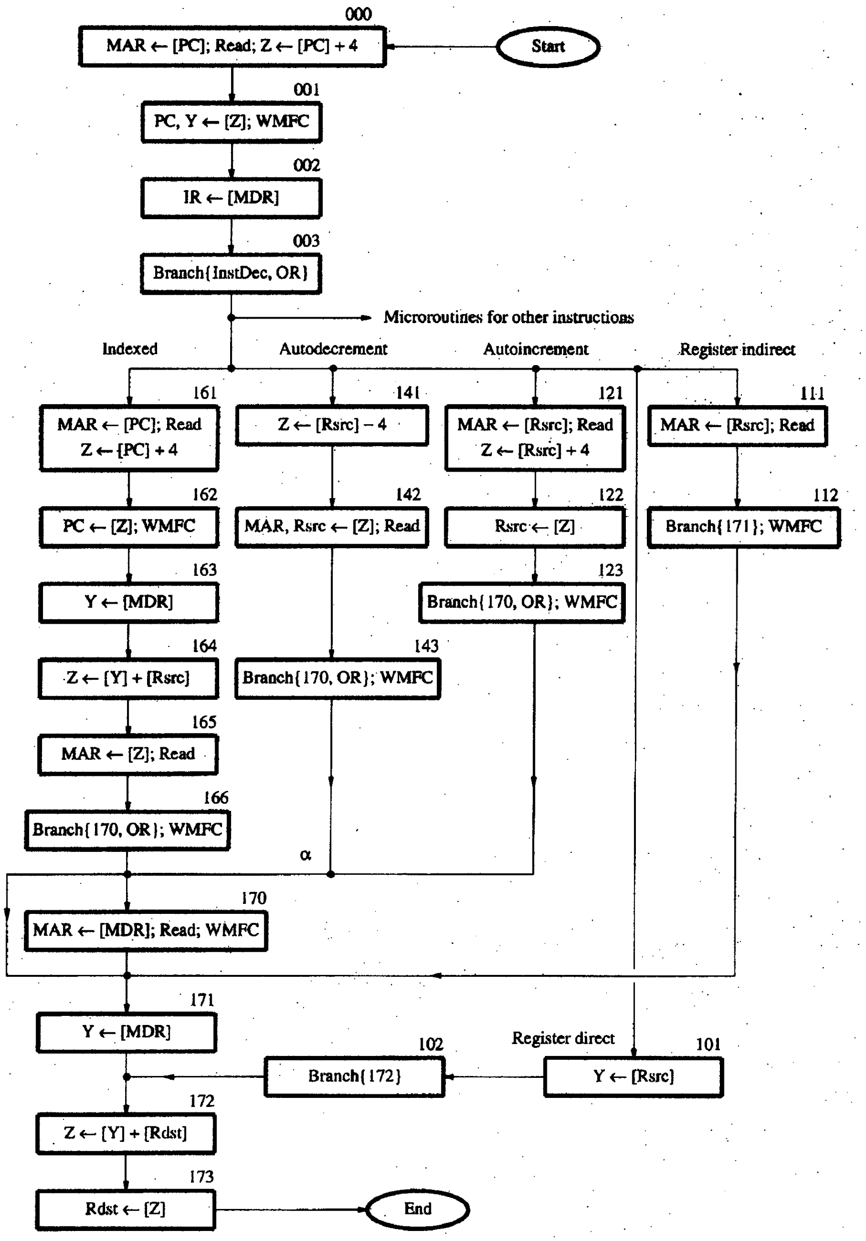


Figure 7.20. Flowchart of a microprogram for the Add src,Rdst instruction.

Address (octal)	Microinstruction
000	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
001	Z_{out}, PC_{in}, Y_{in}, WMFC
002	MDR_{out}, IR_{in}
003	μBranch {$\mu PC \leftarrow 101$ (from Instruction decoder); $\mu PC_{5,4} \leftarrow [IR_{10,9}]$; $\mu PC_{3} \leftarrow [\overline{IR_{10}}] \cdot [\overline{IR_9}] \cdot [IR_8]$}
121	$Rsrc_{out}$, MAR_{in}, Read, Select4, Add, Z_{in}
122	Z_{out}, $Rsrc_{in}$
123	μBranch {$\mu PC \leftarrow 170$; $\mu PC_{0} \leftarrow [\overline{IR_8}]$}, WMFC
170	MDR_{out}, MAR_{in}, Read, WMFC
171	MDR_{out}, Y_{in}
172	$Rdst_{out}$, SelectY, Add, Z_{in}
173	Z_{out}, $Rdst_{in}$, End

Figure 7.21. Microinstruction for Add (Rsrc)+,Rdst.

Microinstructions with Next-Address Field

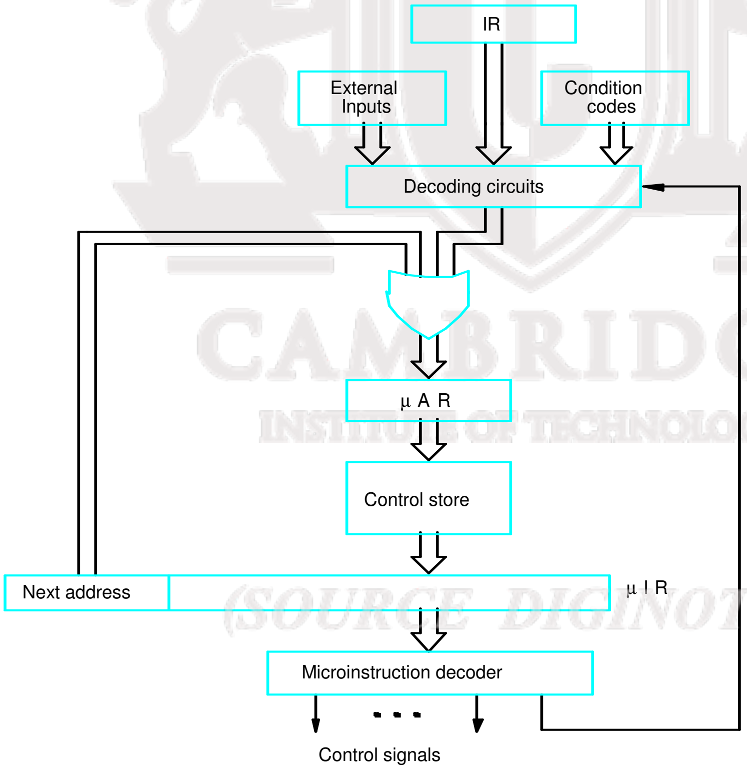


Figure 7.22. Microinstruction-sequencing organization.

- The microprogram we discussed requires several branch microinstructions, which perform no useful operation in the datapath.
- A powerful alternative approach is to include an address field as a part of every microinstruction to indicate the location of the next microinstruction to be fetched.
- Pros: separate branch microinstructions are virtually eliminated; few limitations in assigning addresses to microinstructions.
- Cons: additional bits for the address field (around 1/6)

Microinstruction

F0	F1	F2	F3

F0 (8 bits)	F1 (3 bits)	F2 (3 bits)	F3 (3 bits)
Address of next microinstruction	000: No transfer 001: PC$_{out}$ 010: MDR$_{out}$ 011: Z$_{out}$ 100: Rsrc$_{out}$ 101: Rdst$_{out}$ 110: TEMP$_{out}$	000: No transfer 001: PC$_{in}$ 010: IR$_{in}$ 011: Z$_{in}$ 100: Rsrc$_{in}$ 101: Rdst$_{in}$	000: No transfer 001: MAR$_{in}$ 010: MDR$_{in}$ 011: TEMP$_{in}$ 100: Y$_{in}$

F4	F5	F6	F7

F4 (4 bits)	F5 (2 bits)	F6 (1 bit)	F7 (1 bit)
0000: Add 0001: Sub ⋮ 1111: XOR	00: No action 01: Read 10: Write	0: SelectY 1: Select4	0: No action 1: WMFC

F8	F9	F10

F8 (1 bit)	F9 (1 bit)	F10 (1 bit)
0: NextAdrs 1: InstDec	0: No action 1: OR$_{mode}$	0: No action 1: OR$_{indsrc}$

Figure 7.23. Format for microinstructions in the example of Section 7

Implementation of the Microroutine

Octal address	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
0 0 0	0 0 0 0 0 0 0 1	0 0 1	0 1 1	0 0 1	0 0 0 0	0 1	1	0	0	0	0
0 0 1	0 0 0 0 0 0 1 0	0 1 1	0 0 1	1 0 0	0 0 0 0	0 0	0	1	0	0	0
0 0 2	0 0 0 0 0 0 1 1	0 1 0	0 1 0	0 0 0	0 0 0 0	0 0	0	0	0	0	0
0 0 3	0 0 0 0 0 0 0 0	0 0 0	0 0 0	0 0 0	0 0 0 0	0 0	0	0	1	1	0
1 2 1	0 1 0 1 0 0 1 0	1 0 0	0 1 1	0 0 1	0 0 0 0	0 1	1	0	0	0	0
1 2 2	0 1 1 1 1 0 0 0	0 1 1	1 0 0	0 0 0	0 0 0 0	0 0	0	1	0	0	1
1 7 0	0 1 1 1 1 0 0 1	0 1 0	0 0 0	0 0 1	0 0 0 0	0 1	0	1	0	0	0
1 7 1	0 1 1 1 1 0 1 0	0 1 0	0 0 0	1 0 0	0 0 0 0	0 0	0	0	0	0	0
1 7 2	0 1 1 1 1 0 1 1	1 0 1	0 1 1	0 0 0	0 0 0 0	0 0	0	0	0	0	0
1 7 3	0 0 0 0 0 0 0 0	0 1 1	1 0 1	0 0 0	0 0 0 0	0 0	0	0	0	0	0

Figure 7.24. Implementation of the microroutine of Figure 7.21 using a next-microinstruction address field. (See Figure 7.23 for encoded signals.)

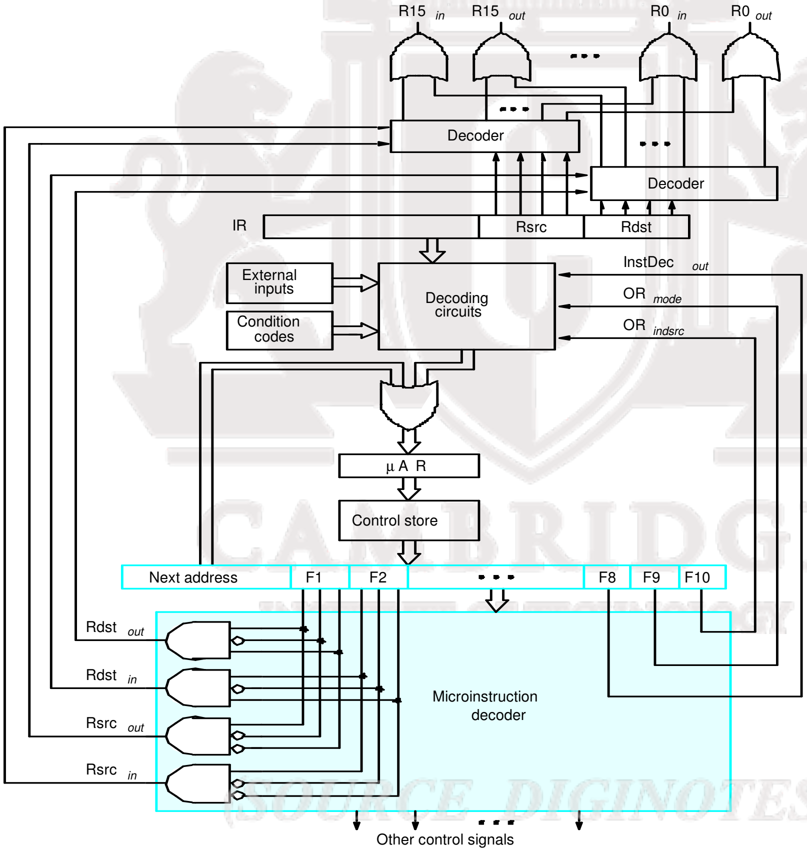


Figure 7.25. Some details of the control-signal-generating circuitry.

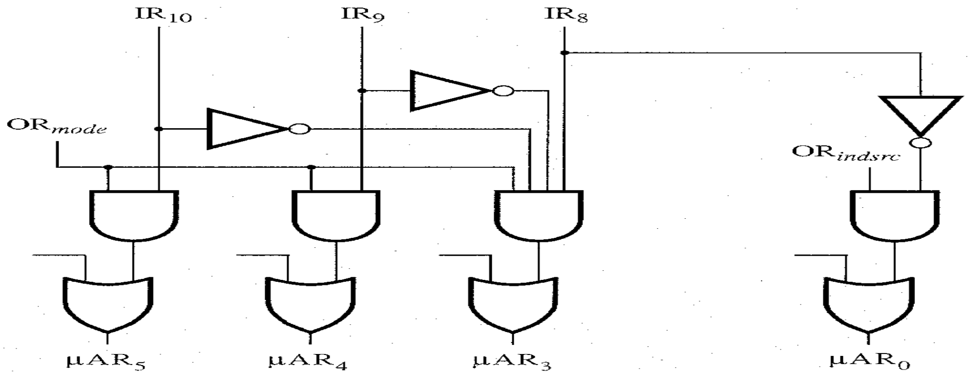


Figure 7.26. Control circuitry for bit-ORing (part of the decoding circuits in Figure 7.25).

Further Discussions

- Prefetching
- Emulation

MODULE 5(CONT.): EMBEDDED SYSTEMS & LARGE COMPUTER SYSTEMS

MICROWAVE OVEN

• Microwave-oven is one of the examples of embedded-system.

• This appliance is based on **magnetron** power-unit that generates the microwaves used to heat food.

• When turned-on, the magnetron generates its maximum power-output.

Lower power-levels can be obtained by turning the magnetron on & off for controlled time-intervals.

• **Cooking Options** include:

→ Manual selection of the power-level and cooking-time.
→ Manual selection of the sequence of different cooking-steps.
→ Automatic melting of food by specifying the weight.

• **Display (or Monitor)** can show following information:

→ Time-of-day clock.
→ Decrementing clock-timer while cooking.
→ Information-messages to the user.

• **I/O Capabilities** include:

→ Input-keys that comprise a 0 to 9 number pad.
→ Function-keys such as Start, Stop, Reset, Power-level etc.
→ Visual output in the form of a LCD.
→ Small speaker that produces the beep-tone.

• **Computational Tasks** executed are:

→ Maintaining the time-of-day clock.
→ Determining the actions needed for the various cooking-options.
→ Generating the control-signals needed to turn on/off devices.
→ Generating display information.

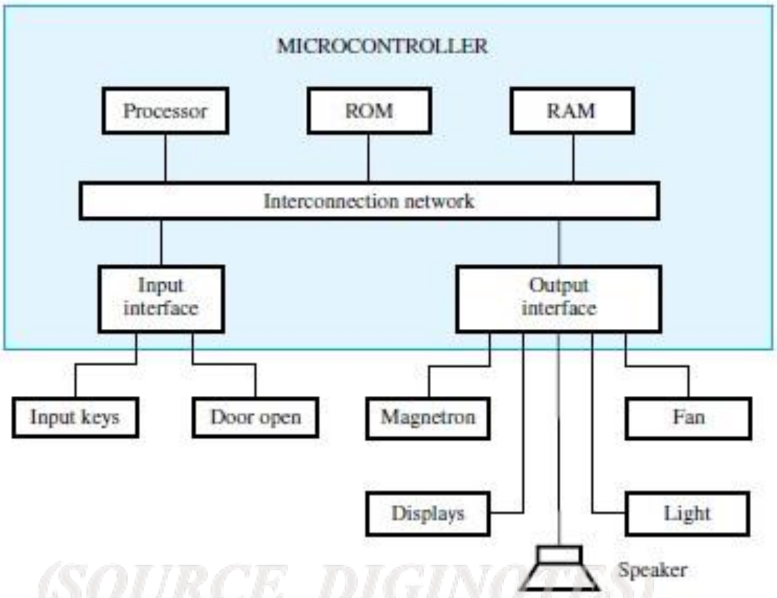


Figure 10.1 A block diagram of a microwave oven.

• **Non-volatile ROM** is used to store the program required to implement the desired actions.

So, the program will not be lost when the power is turned off (Figure 10.1).

• Most important requirement: The microcontroller must have sufficient I/O capability.

Parallel I/O Ports are used for dealing with the external I/O signals.

Basic I/O Interfaces are used to connect to the rest of the system.

DIGITAL CAMERA

• Digital Camera is one of the examples of embedded system.

• An array of **Optical Sensors** is used to capture images (Figure 10.2).

• The optical-sensors convert light into electrical charge.

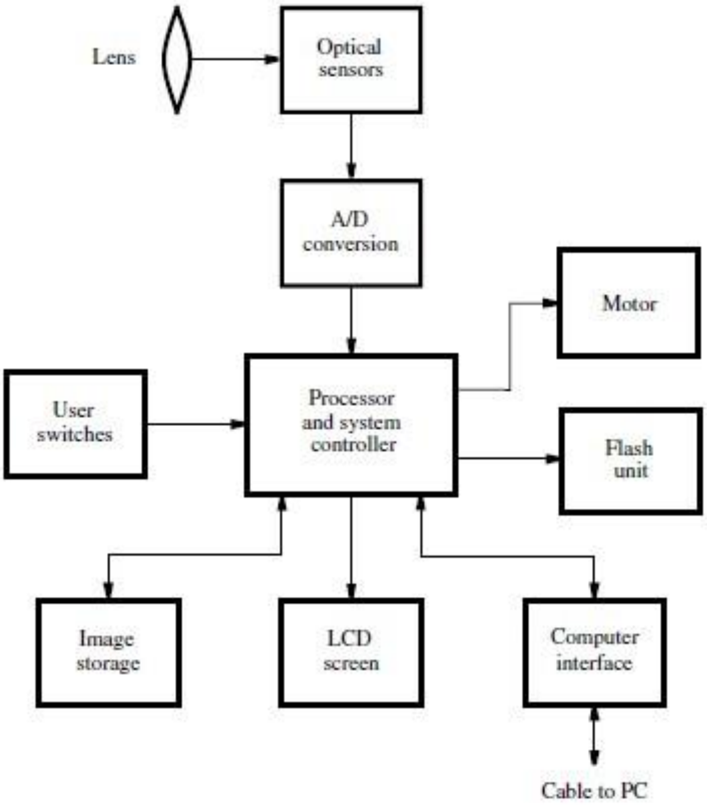


Figure 10.2 A simplified block diagram of a digital camera.

- Each sensing-element generates a charge that corresponds to one **pixel**.
  One pixel is one point of a pictorial image.
  The number of pixels determines the quality of pictures that can be recorded & displayed.
- **ADC** is used to convert the charge which is an analog quantity into a digital representation.
- **Processor**
  → manages the operation of the camera.
  → processes the raw image-data obtained from the ADCs to generate images.
- The images are represented in standard-formats, so that they are suitable for use in computers.
- Two standard-formats are:
  1) TIFF is used for uncompressed images &
  2) JPEG is used for compressed images.
- The processed-images are stored in a larger storage-device. For ex: Flash memory cards.
- A captured & processed image can be displayed on a LCD screen of camera.
- The number of saved-images depends on the size of the storage-unit.
- Typically, **USB Cable** is used for transferring the images from camera to the computer.
- **System Controller** generates the signals needed to control the
  operation of i) Focusing mechanism and
  ii) Flash unit.

(ADC → Analog-to-digital converter, LCD → liquid-crystal display)
(TIFF →Tagged Image File Format, JPEG →Joint Photographic Experts Group)

HOME TELEMETRY (DISPLAY TELEPHONE)

- Home Telemetry is one of the examples of embedded system.
- The display-telephone has an embedded processor which enables a remote access to other devices in the home.
- Display telephone can perform following functions:
  1) Communicate with a computer-controlled home security-system.
  2) Set a desired temperature to be maintained by an air conditioner.
  3) Set start-time, cooking-time & temperature for food in the microwave-oven.
  4) Read the electricity, gas, and water meters.
- All of this is easily implementable if each of these devices is controlled by a **microcontroller**.
- A link (wired or wireless) has to be provided between
  1) Device microcontroller & 2) Microprocessor in the telephone.
- Using signaling from a remote location to observe/control state of device is referred to as **telemetry**.

MICROCONTROLLER CHIPS FOR EMBEDDED APPLICATIONS

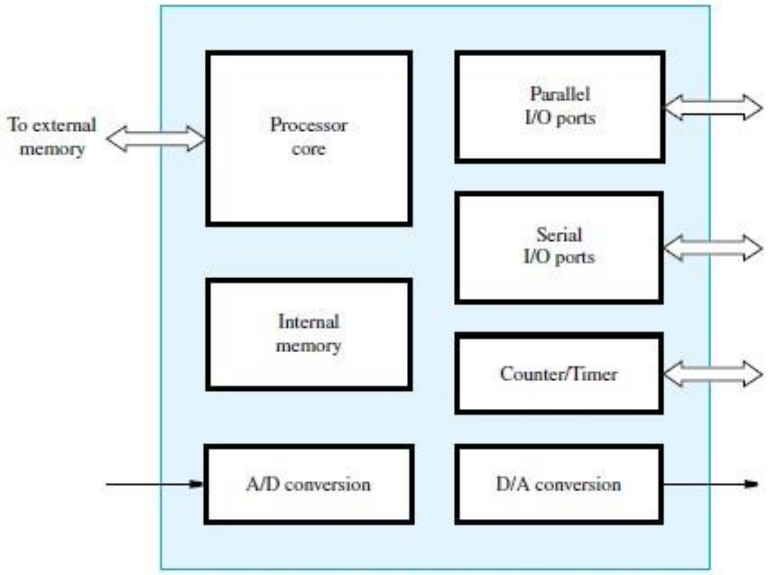


Figure 10.3 A block diagram of a microcontroller.

- **Processor Core** may be a basic version of a commercially available microprocessor (Figure 10.3).
- Well-known popular microprocessor architecture must be chosen. This is because, design of new products is facilitated by
  → numerous CAD tools
  → good examples &
  → large amount of knowledge/experience.
- **Memory-Unit** must be included on the microcontroller-chip.
- The memory-size must be sufficient to satisfy the memory-requirements found in small applications.
- Some memory should be of **RAM** type to hold the data that change during computations.
  Some memory should be of **Read-Only** type to hold the software.
  This is because an embedded system usually does not include a magnetic-disk.
- A field-programmable type of ROM storage must be provided to allow cost-effective use.
  For example: EEPROM and Flash memory.
- **I/O ports** are provided for both parallel and serial interfaces.
- **Parallel and Serial Interfaces** allow easy implementation of standard I/O connections.
- **Timer Circuit** can be used
  → to generate control-signals at programmable time intervals &
  → for event-counting purposes.
- An embedded system may include some **analog devices**.
- **ADC & DAC** are used to convert analog signals into digital representations, and vice versa.

PARALLEL I/O INTERFACE

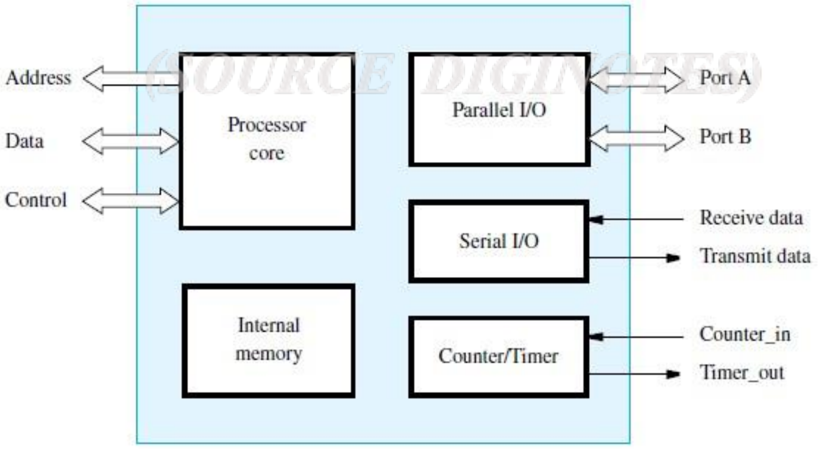


Figure 10.4 An example microcontroller.

- Each parallel port has an associated 8-bit DDR (Data Direction Register) (Figure 10.4).
- **DDR** can be used to configure individual data lines as either input or output.

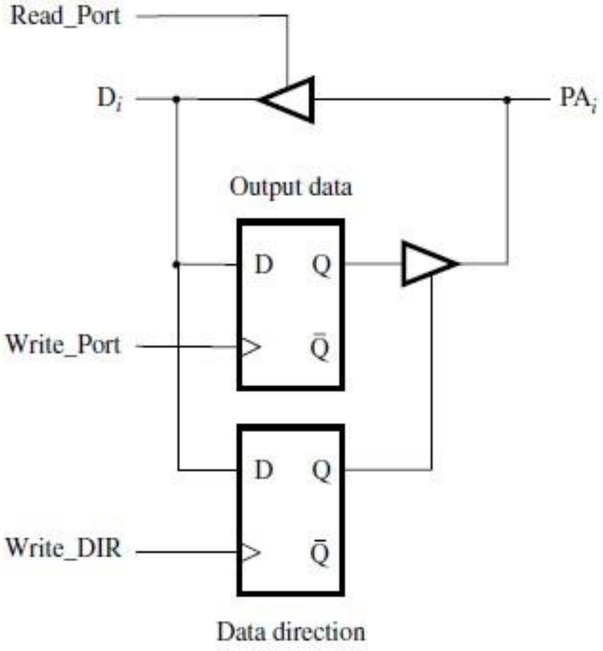


Figure 10.5 Access to one bit in port A in Figure 10.4.

- If the data direction flip-flop contains a 0, then Port pin **PA_i** is treated as an input (Figure 10.5).

  If the data direction flip-flop contains a 1, then Port pin PA_i is treated as an output.
- Activation of control-signal **Read_Port**, places the logic value on the port-pin onto the data line D_i.

  Activation of control-signal **Write_Port**, places value loaded into output data flip-flop onto port-pin.
- **Addressable Registers** are (Figure 10.6):

  1) Input registers (PAIN for port A, PBIN for port B)
  2) Output registers (PAOUT for port A, PBOUT for port B)
  3) Direction registers (PADIR for port A, PBDIR for port B)
  4) Status-register (PSTAT) &
  5) Control register (PCONT)

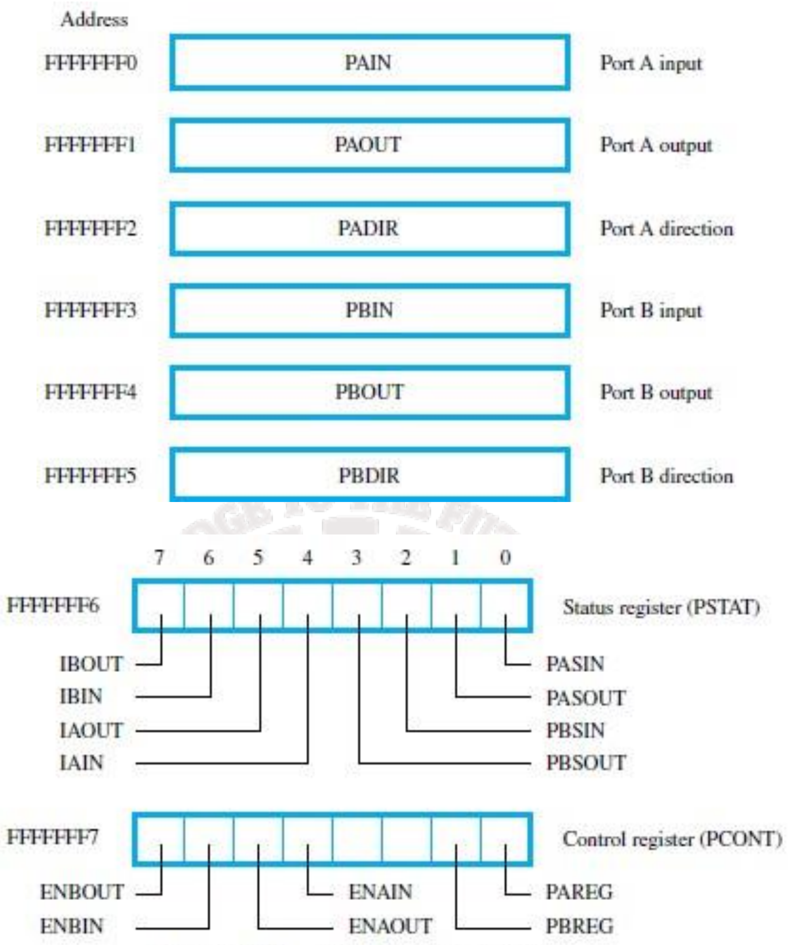


Figure 10.6 Parallel interface registers.

• **Status Register** provides information about the current status of

1) Input registers &

2) Output registers.

• PASIN =1 → When there are new data on port A (Figure 10.6).

PASIN =0 → When the processor accepts the data by reading the PAIN register.

• The interface uses a separate control line to indicate availability of new data to the connected-device.

• PASOUT = 1 → When the data in register PAOUT are accepted by the connected-device.

PASOUT = 0 → When the processor writes data into PAOUT.

• Like PASIN & PAOUT, the flags PBSIN and PBSOUT perform the same function on port B.

• The status register also contains four interrupt flags. They are IAIN, IAOUT, IBIN & IBOUT.

• IAIN = 1 → When interrupt is enabled and the corresponding I/O action occurs.

• The interrupt-enable bits are held in control register PCONT.

• ENAIN= 1 → when the corresponding interrupt is enabled.

• For ex: If ENAIN=1 & PASIN=1, then interrupt flag IAIN is set to 1 and an interrupt request is raised.

Thus,

IAIN = ENAIN * PASIN

• **Control Registers** is used for controlling data transfers to/from the devices connected to ports A/B.

• Port A has two control lines: CAIN and CAOUT.

• CAIN and CAOUT are be used to provide an automatic signaling mechanism b/w i) Interface and

ii) Attached device.

• PAREG and PBREG are used to select the mode of operation of inputs to ports A and B respectively.

• If PAREG =1;

Then, a register is used to store the input data. Otherwise, a direct path from the pins is used.

SERIAL I/O INTERFACE

- The serial interface provides the UART capability to transfer data (Figure 10.7). (UART → Universal Asynchronous Receiver/Transmitter).
- Double buffering is
  → used in both the transmit- and receive-paths.
  → needed to handle bursts in I/O transfers correctly.

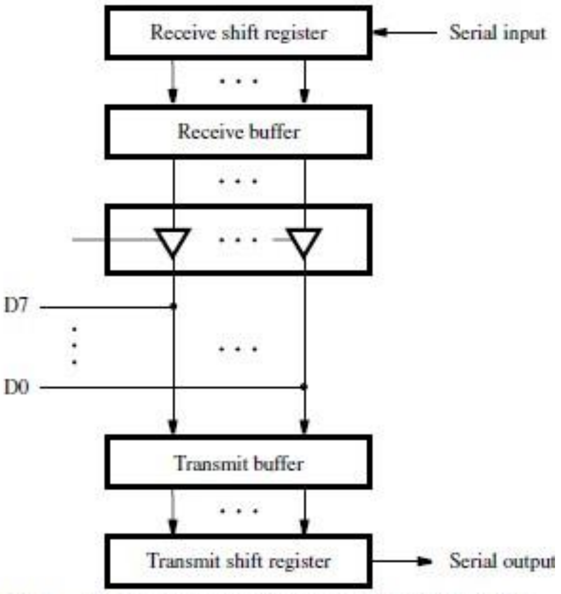


Figure 10.7 Receive and transmit structure of the serial interface.

- **Addressable Registers** are (Figure 10.8):
  1) Receive-buffer
  2) Transmit-buffer
  3) Status-register (SSTAT)
  4) Control register (SCONT) &
  5) Clock-divisor register (DIV).

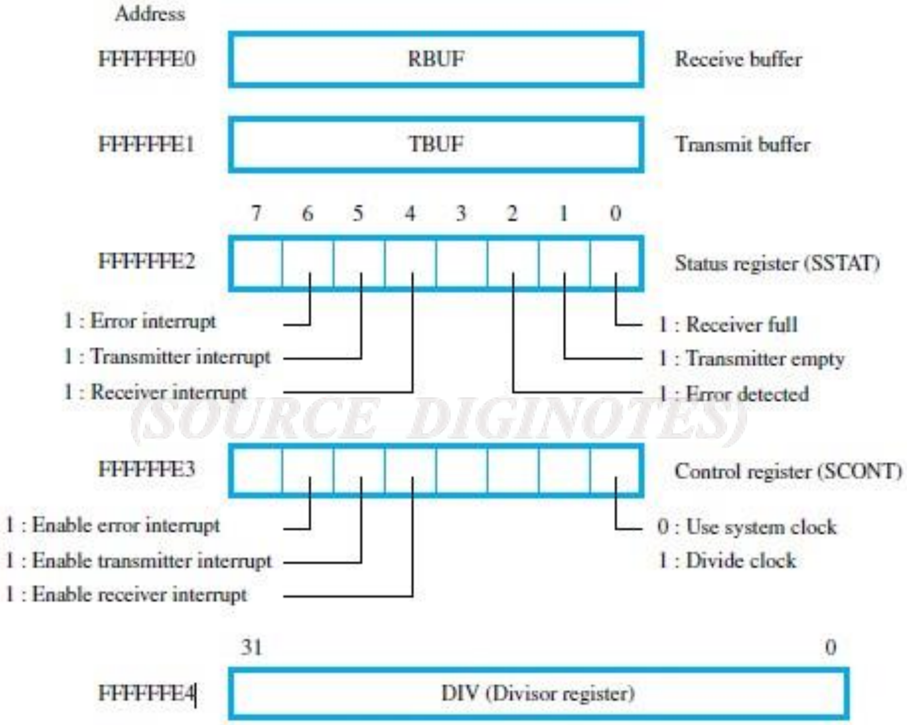


Figure 10.8 Serial interface registers.

- Input data are read from the **Receive-buffer**.
  Output data are loaded into the **Transmit-buffer**.
- **Status Register** (SSTAT) provides information about the current status of i) Receive-units and
  ii) Transmit-units.
- Bit SSTAT0 = 1 → When there are new data in the receive-buffer.
  Bit SSTAT0 = 0 → When the processor accepts the data by reading the receive-buffer.
- SSTAT1 = 1 → When the data in transmit-buffer are accepted by the connected-device.
  SSTAT1 = 0 → When the processor writes data into transmit-buffer. (SSTAT0 & SSTAT1 similar to SIN & SOUT)
- SSTAT2 = 1 → if an error occurs during the receive process.
- The status-register also contains the interrupt flags.
- SSTAT4 =1 → When the receive-buffer becomes full and the receiver-interrupt is enabled.
  SSTAT5 = 1 → When the transmit-buffer becomes empty & the transmitter-interrupt is enabled.
- **Control Register** (SCONT) is used to hold the interrupt-enable bits.
- If SCONT6−4 = 1.
  Then the corresponding interrupts are enabled. Otherwise, the corresponding interrupts are disabled.
- Control register also indicates how the transmit clock is generated
- If SCONT0 = 0.
  Then, the transmit clock is the same as the system (processor) clock.
  Otherwise, a lower frequency transmit clock is obtained using a clock-dividing circuit.
- **Clock-divisor register** (DIV) divides system-clock signal to generate the serial transmission clock.
- The counter generates a clock signal whose frequency is equal to

$$= \frac{\text{Frequency of system clock}}{\text{Contents of DIV register}}$$

COUNTER/TIMER

- A 32-bit down-counter-circuit is provided for use as either a counter or a timer.
- Basic operation of the circuit involves
  - → loading a starting value into the counter and
  - → then decrementing the counter-contents using either
    - i) Internal system clock or
    - ii) External clock signal.
- The circuit can be programmed to raise an interrupt when the counter-contents reach 0.

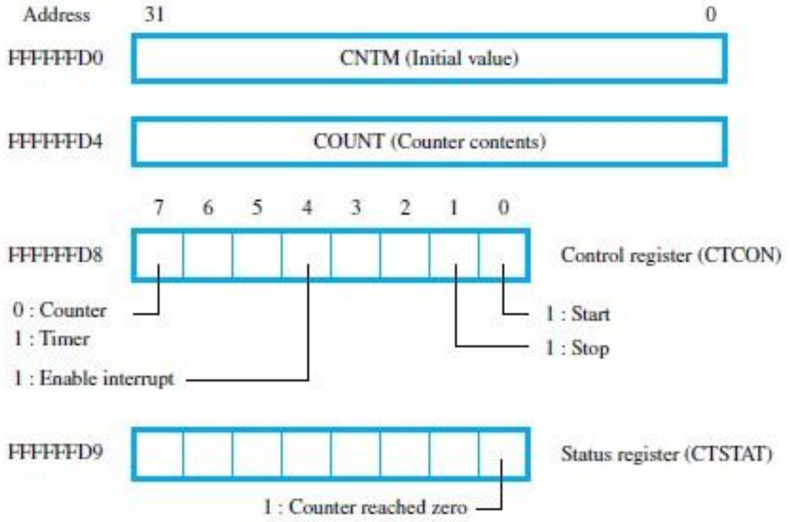


Figure 10.9 Counter/Timer registers.

- **Counter/Timer Register** (CNTM) can be loaded with an initial value (Figure 10.9).
- The initial value is then transferred into the counter-circuit.
- The current contents of the counter can be read by accessing memory-address FFFFFFD4.
- **Control Register** (CTCON) is used to specify the operating mode of the counter/timer circuit.
- The control register provides a mechanism for
  - → starting & stopping the counting-process &
  - → enabling interrupts when the counter-contents are decremented to 0.
- **Status Register** (CTSTAT) reflects the state of the circuit.
- There are 2 modes: 1) Counter mode 2) Timer mode.

Counter Mode

- CTCON7 = 0 → When the counter mode is selected.
- The starting value is loaded into the counter by writing it into register CNTM.
- The counting-process begins when bit CTCON0 is set to 1 by a program.
- Once counting starts, bit CTCON0 is automatically cleared to 0.
- The counter is decremented by pulses on the Counter.
- Upon reaching 0, the counter-circuit
  - → sets the status flag CTSTAT0 to 1 &
  - → raises an interrupt if the corresponding interrupt-enable bit has been set to 1.
- The next clock pulse causes the counter to reload the starting value.
- The starting value is held in register CNTM, and counting continues.
- The counting-process is stopped by setting bit CTCON1 to 1.

Timer Mode

- CTCON7 = 1 → When the timer mode is selected.
- This mode can be used to generate periodic interrupts.
- It is also suitable for generating a square-wave signal.
- The process starts as explained above for the counter mode.
- As the counter counts down, the value on the output line is held constant.
- Upon reaching 0, the counter is reloaded automatically with the starting value, and the output signal on the line is inverted.

- Thus, the period of the output signal is twice the starting counter value multiplied by the period of the controlling clock pulse.
- In the timer mode, the counter is decremented by the system clock.

MODULE 5(CONT.): THE STRUCTURE OF GENERAL-PURPOSE MULTIPROCESSORS

THE STRUCTURE OF GENERAL-PURPOSE MULTIPROCESSORS

1. UMA (Uniform Memory Access) Multiprocessor

• An interconnection-network permits n processors to access k memories (Figure 12.2). Thus, any of the processors can access any of the memories.

• The interconnection-network may introduce network-delay between

1) Processor &

2) Memory.

• A system which has the same network-latency for all accesses from the processors to the memory-modules is called a **UMA Multiprocessor**.

• Although the latency is uniform, it may be large for a network that connects

→ many processors &

→ many memory-modules.

• For better performance, it is desirable to place a memory-module close to each processor.

• **Disadvantage:**

➢ Interconnection-networks with very short delays are costly and complex to implement.

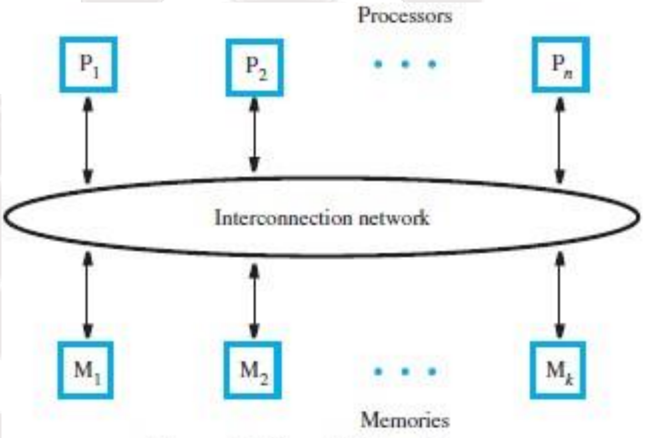


Figure 12.2 A UMA multiprocessor.

2. NUMA (Non-Uniform Memory Access) Multiprocessors

• Memory-modules are attached directly to the processors (Figure 12.3).

• The network-latency is avoided when a processor makes a request to access its local memory.

• However, a request to access a remote-memory-module must pass through the network.

• Because of the difference in latencies for accessing local and remote portions of the shared memory, systems of this type are called **NUMA** multiprocessors.

• **Advantage:**

➢ A high computation rate is achieved in all processors

• **Disadvantage:**

➢ The remote accesses take considerably longer than accesses to the local memory.

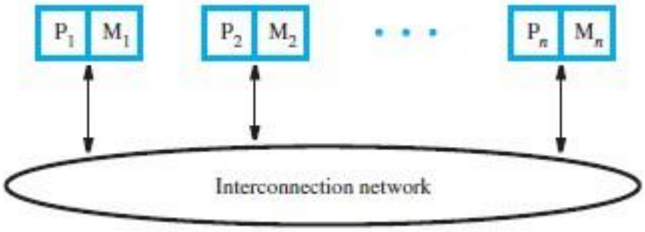


Figure 12.3 A NUMA multiprocessor.

3. Distributed Memory Systems

- All memory-modules serve as private memories for processors that are directly connected to them.
- A processor cannot access a remote-memory without the cooperation of the remote-
processor.
- This cooperation takes place in the form of messages exchanged by the processors.
- Such systems are often called **Distributed-Memory Systems** (Figure 12.4).

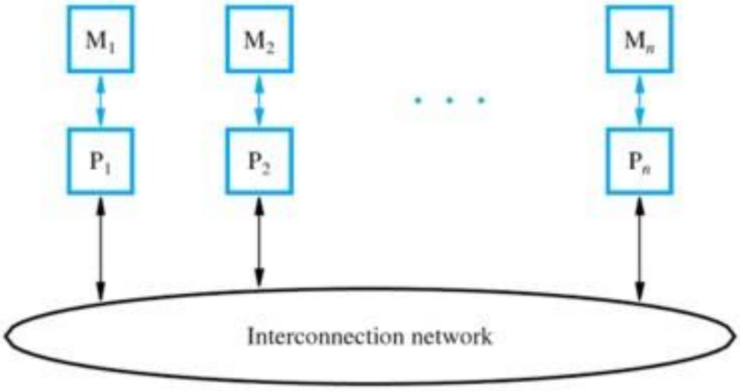


Figure 12.4 A distributed memory system.