

## UNIT-1

### Overview & instructions

#### Introduction

**Computer:** is an essential use of processing (Communication)

Any computing system will consists of a processor, Memory and I/O devices.

**Processor** (carry out processing)

**Memory** (used to store data)

**I/O device** (man to interact with the system)

#### Computer Architecture:

- It consist of set of hardware components technology and software specifications.
- It is a set of rules and methods that describe the functionality, organization and implementation of computer systems.
- It is a science and art of selecting and interconnecting that hardware components to create computers that means function, performance and cost goals

#### Comparison between building and computer architecture:

Building architecture (Structural design) studied by civil engineers

Computer architecture (Circuit design) studied by electronic engineers

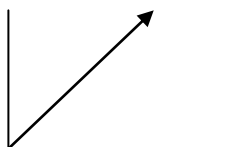
#### 1. Describe the eight ideas that leads to performance improvement (8)

Eight ideas that the computer architecture has been invented for computer design.

1. Design for moore's law
2. Use abstraction for design
3. Make common case fast
4. Performance via parallelism
5. Performance via pipelining
6. Performance via prediction
7. Hierarchy of memory
8. Dependability via redundancy

#### Design for moore's law :

- It states that integrated circuit resources (transistors) double every 18–24 months.
- The computer designer must predict the rapid change in IC capacity & design it accordingly.
- Moore's Law graph to represent designing for rapid change.



#### Use abstraction for design:

- Abstraction means freedom from representational quality
- A major productivity technique for hardware and software is to use abstractions.
- Use abstractions to represent the design at different levels of representation.
- Lower-level details are hidden to offer a simpler model at higher levels.

### **Make the Common Case Fast :**

- Make the common case fast to enhance performance better than optimizing the rare case.
- For this idea the common case has to be carefully identified and experimented.
- Example: increasing speed level for a sports car is very easier than to a minivan

### **Performance via parallelism:**

- Parallelism means simultaneous execution of source task on multiple processors in order to obtain the result faster
- Computer architects have offered designs that get more performance by performing operations in parallel.

Example: Dual-quad processor

### **Performance via pipelining :**

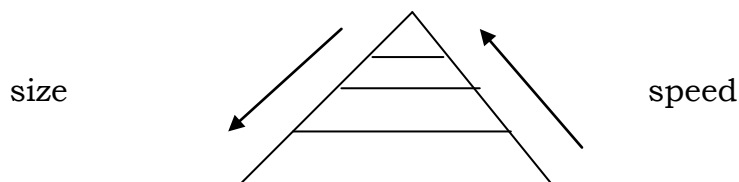
- A particular pattern of parallelism is called **pipelining**.
- In pipelining more than one instruction are executed at the same time to increase the performance and throughput.

### **Performance via prediction:**

- A statement about what will happen or might happen in the future.
- In some cases, based on prediction.
- It is better to start working based on prediction or average guess to make the performance faster rather than working until you know for sure.

### **Hierarchy of Memories :**

- Memory speed and size often plays a vital role in increasing the performance of the system, but due to the high cost of memory, the size of problem that can be solved is limited.
- To address this demand, hierarchy of memory has to be used.
- Memory to be faster, smallest and most expensive memory per bit at the top of the hierarchy and the slowest, largest and cheapest memory bit at the bottom of the hierarchy.



### **Dependability via Redundancy :**

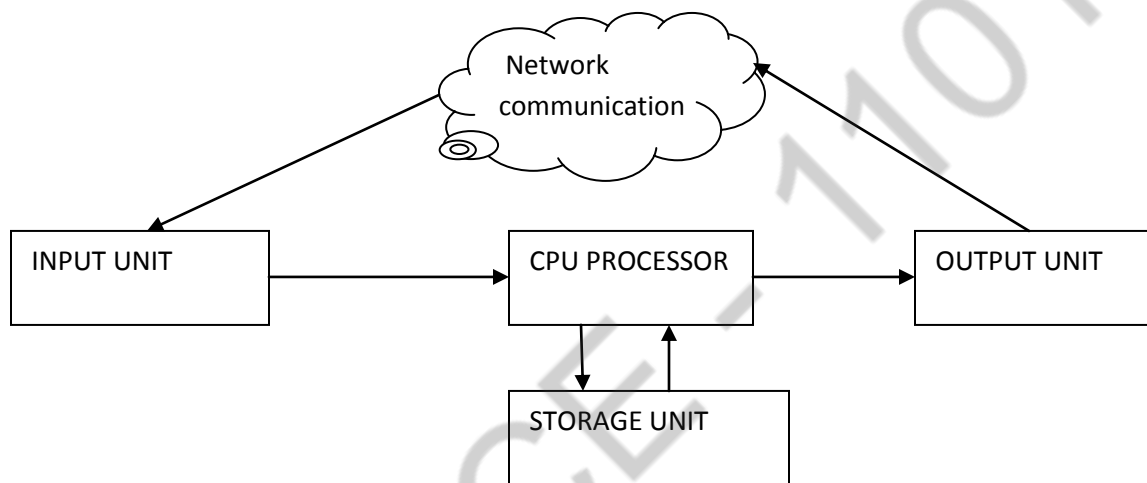
- Computers not only need to be fast, They need to be dependable by including redundant components.

- Since any physical device can fail, the systems has to be made dependable while including redundant components that can take over when a failure occurs *and* to help detect failures.

## 2. Components of a computer systems:

**MAY/ JUNE 2016, NOV/DEC 2015 , NOV/DEC 2014**

- A computer in its simplest form is a fast electronic machine.
- It accepts digitize information from the user processes it according to a sequence of instruction and provides the processor information to the user.
- Components of computer system are hardware and software



### **Hardware components are:**

1. Input unit 2. Output unit 3. Memory unit 4. Cpu

#### **1. Input unit:**

- It is used for entering data and programs from user to computer system for processing.
- Most commonly used input device are **keyboard and mouse [Expalin]**.

#### **2. Output unit:**

- It is used for displaying the results produced by the system after processing.
- Most commonly used output devices are **monitor, printer, plotter etc [Explain]**.

#### **3. Memory unit:**

- It is used for storing data and instruction before and after processing.
- It is divided into primary memory and secondary memory.**[explain Ram,Rom]**

### **Primary memory (volatile main memory):**

- It is fast semiconductor RAM.
- It loses instructions and data when power off.
- It used to hold program while they are running.

### **Secondary memory (non- volatile):**

- It is magnetic tapes, magnetic disk are used for the storage of larger amount of data.
- A form of memory that retains data even in the absence of a power source.
- It is permanent storage device.

### **4. Central processing unit (CPU):**

- Cpu is a brain of the system.
- Cpu takes data and instructions from the storage unit and makes all sorts of calculations based on the instructions gives, type of data provided.
- Cpu is divided into 2 sections namely:

#### **1. ALU: arithmetic and logical unit.**

- All arithmetic and logical operations are performed by the ALU.
- To perform these operations operands from the main memory are brought into internal registers of processor.
- After performing operation the result is either stored in the register or memory.

#### **2. Control unit:**

- It co-ordinates and controls all the activities among the functional units.
- A basic function of control unit is to fetch the instructions stored in main memory, identify the operations and devices involved in it and accordingly generate control signals to execute the desired operations.

### **Network communication:**

Network have becomes so popular that they are the backbone of current computer systems.

### **Networked computer have several major advantages:**

- **Communication:** Information is exchanged between computers at high speeds.
- **Resource sharing:** Rather than each computer having its own I/O devices, computers on the network can share I/O devices.
- **Nonlocal access:** By connecting computers over long distances, users need not be near the computer they are using.

### **Software components: Software is a collection of program**

#### **Computer software is divided into two broad categories:**

##### **1. System software    2. Application software**

#### **1. System software:**

- It is collection of programs which is needed in the creation, preparation and execution of other program.
- System software includes editor, assemblers, linker, loader, compilers, interpreters, debuggers and operating system

#### **2. Application software:**

- Allows to perform specific task on a computer using capabilities of computer.
- Application software to accomplish a task.
- Different application software are needed to perform different tasks.

### **Operating system:**

- OS Is a collection of routines that tells the computer what to do under a variety of conditions.
- It is used to control the sharing of and interaction among various computer units as they execute application programs.

### **3. Technology :**

- Technology shapes the computer for better performance.
- Technologies that have been used over time with relative performance per unit cost for each technology.

<b>Year</b>	<b>Technology used in computer</b>	<b>Relative performance/unit cost</b>
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit	900
1995	Very large – scale IC's	2400.00
2013	Ultra large-scale IC's	6,200,000.00

#### **Vacuum Tubes:**

- The first electronic computer, ENIAC (Electronic Numerical Integrator and computer)
- It was designed and constructed by Eckert and Mauchly.
- It was made up of more than 18000 vacuum tubes and 1500 relays.
- It was able to perform nearly 5000 additions or subtractions per second.
- It was a decimal rather than a binary machine.
- Weight-30 tones, area -15000 sq.ft , power consumption -140kW.
- Data memory consists of 20 accumulators, each capable of storing a ten digit decimal number.

#### **Transistors:**

- A transistor is simply an on/off switch controlled by electricity.
- Transistors are smaller, cheaper and low power consumption
- Greater speed, larger memory capacity and smaller size than first generation.
- CPU can handle both floating point and fixed point operation.
- Separate I/O processor having direct access to main memory to control I/O operations.
- It introduction of more complex arithmetic and logic unit and control units to support high level languages.

#### **Integrated circuit:**

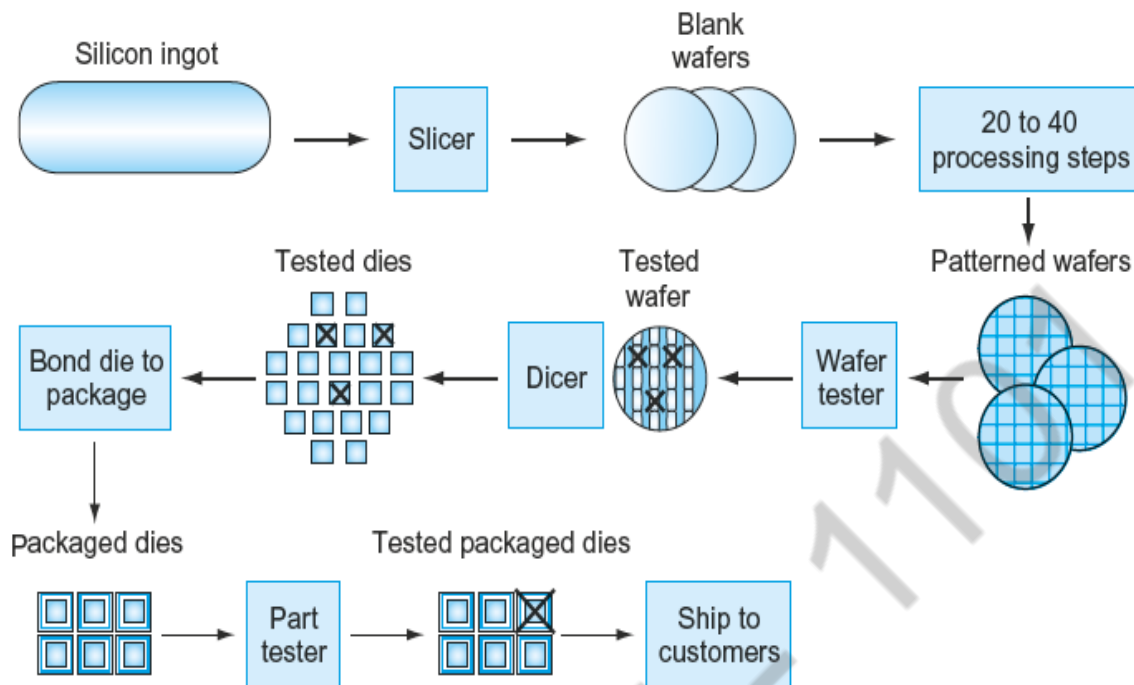
- It enabled lower cost, faster processors and development of memory chips.
- IC allowed to increase memory size and number of I/O port
- Magnetic core memories were replaced by integrated circuit memories.
- IC combined dozens to hundreds of transistors into a single chip.

#### **VLSI: very large-scale integrated (VLSI) circuit**

It consists of billions of combinations of conductors, insulators & switches manufactured in a small package.

- Excellent conductors of electricity (using either microscopic copper or aluminum wire)
- Excellent insulators from electricity (like plastic sheathing or glass)
- Areas that can conduct or insulate under special conditions (as a switch)

**Chip manufacturing process:** Today, most integrated circuits (ICs) are made of silicon



### Silicon :

- Silicon does not conduct electricity well
- Is a natural element that is a semi conductor.
- **Semiconductor:** A substance that does not conduct electricity well
- Silicon that allow tiny areas to transform into one of three devices: (conductors , insulator and switch)

### Silicon crystal ingot :

is a rod composed of a silicon crystal that is between 8 to 12 inches in diameter and about 12 to 24 inches long.

**Slicer:** These cylinders are sliced into thin

### Blank wafer:

These cylinders are highly polished wafers less than one-fortieth of an inch thick.

### 20 to 40 processing steps:

the wafers are exposed to a multiple-step photolithography process that is repeated once for each mask required by the circuit.

Each mask defines different parts of a transistor, capacitor, resistor, or connector composing the complete integrated circuit and defines the circuitry pattern for each layer on which the device is fabricated.

### Patterned wafers:

pattern on the wafer in the exact design of the mask

### Tester packaged dies:

**die** :The individual rectangular sections that are cut from a wafer, more informally known as chips.

**yield** :The percentage of good dies from the total number of dies on the wafer.

### Elaboration:

The cost of an integrated circuit can be expressed in three simple equations:

$$\text{cost per die} = \frac{\text{Cost per wafer}}{\text{dies per wafer} * \text{yield}}$$

$$\text{Diea per wafer} = \frac{\text{wafer area}}{\text{Dies area}}$$

$$\text{yield} = \frac{1}{\left(1 + \left(\text{Defect per area} * \text{Die} \frac{\text{area}}{2}\right)\right)^2}$$

### ULSI:

- Is the process of integrating or embedding millions of transistors on a single silicon semiconductor micro chip.
- It is a successor to large scale integration and very large scale integrating technology
- It was design to provide the greatest possible computational power from the smallest form factor of microchip or microprocessor dye.

## 4. Performance:

- Performance is an important attribute of a computer.
- It is an important criterion for selection of a computer.
- Performance of a computer can be measured in number of ways.

### Performance based on:

#### 1. Response time:

- How long it takes to do a task.
- It is also called execution time.
- It includes disk access, memory access, I/O activities.

#### 2. Throughput :

- Total amount of work done in a given time.
- It is also called bandwidth.

Performance of computer is directly related to the throughput and hence it is reciprocal of execution time.

$$\text{performance} = \frac{1}{\text{Executiontime}}$$

Evaluate two computers A & B. then performance of A is greater than B.

$$\text{Performance A} > \text{performance B}$$

$$\frac{1}{\text{Executiontime A}} > \frac{1}{\text{Executiontime B}}$$

$$\text{Execution time B} > \text{Execution A}$$

Execution time B greater than execution A so A is faster than B

A is n times faster as B to mean

$$\frac{\text{performance A}}{\text{performance B}} = n$$

$$\frac{\text{performance A}}{\text{performance B}} = \frac{1}{\text{Executiontime A}} / \frac{1}{\text{Executiontime B}}$$

$$\frac{\text{performance A}}{\text{performance B}} = \frac{\text{Execution B}}{\text{Executiontime A}} = n$$

**Problem:**

If a computer A runs a program in 10 second & B runs the same problem in 15 seconds how much faster is A than B?

$$\frac{\text{performance A}}{\text{performance B}} = \frac{\text{Execution time B}}{\text{Executiontime A}} = n$$

$$\frac{15}{10} = 1.5$$

A is therefore 1.5 times as fast as B

$$\frac{\text{performance A}}{\text{performance B}} = 1.5$$

$$\frac{\text{performance A}}{1.5} = \text{performance B}$$

**Measuring performance:**

- One of the important measures of a computer performance is a time.
- Program execution time is measured in seconds per program.
- Time can be divided in different ways

**CPU time:**

- It is also called CPU execution time
- The actual time the cpu spends for computing a specific task.
- CPU time is divided into user cpu time and system cpu time.

**User CPU time:** The cpu time spent in a program.



**System cpu time:** The cpu time spend in the operating system perform task on behalf of the program.

**Performance metrics:**

- Users and designers often examine performance using different metrics.
- All computers are constructed using a clock that determines when events take place in the hardware

**Most basic metrics are:**

**1.Clock cycle :** A clock cycle, or simply a "cycle," is a single electronic pulse of a [CPU](#). During each cycle, a CPU can perform a basic operation such as fetching an instruction, accessing memory, or writing data.

**2.Clock cycle time :**

*CPU execution time for a program = cpu clock cycle for a program X clock cycle time*

Alternatively because clock rate and clock cycle time are inverses:

$$CPU\ execution\ time\ for\ a\ program = \frac{CPU\ clock\ cycles\ for\ a\ program}{Clock\ rate}$$

**3.Clock rate:**

The clock rate of a computer is normally determined by the frequency of a crystal.

**Problems:**

Computer A run a program in 12 seconds with a 3 GHz clock. We have to design a computer B such that it can run the same program within 9 seconds. Determine the clock rate for computer B. Assume that due to increase in clock cycle rate , CPU design of computer b is affected and it requires 1.2 times as many clock cycles as computer A for execution this program.

**Solution:**

Clock rate A =  $3 * 10^9$  cycles/sec

CPU time A = 12 seconds

CPU time B = 9 seconds

We have:

$$CPU\ time\ A = \frac{CPU\ clock\ cycles\ A}{Clock\ rate\ A}$$

$$12\ seconds = \frac{CPU\ clock\ cycles\ A}{3 * 10^9\ cycles/sec}$$

CPU clock cycles A = 12 seconds \* 3 \* 10<sup>9</sup> cycle /sec = 36 \* 10<sup>9</sup> cycles

The cpu time for computer B can be given as

$$CPU\ time\ B = \frac{CPU\ clock\ cycles\ B}{Clock\ rate\ B}$$

$$CPU\ time\ B = \frac{1.2 * CPU\ clock\ cycles\ A}{Clock\ rate\ B}$$

$$9\ seconds = \frac{1.2 * 36 * 10^9\ cycles}{Clock\ rate\ B}$$

$$clock\ rate\ B = \frac{1.2 * 36 * 10^9\ cycles}{9\ second} = 4.8 * 10^9\ cycles = 4.8\ GHz$$

### **Instruction performance:**

The computer had to execute the instructions to run the program.

The execution time must depend on the number of instructions in a program.

**CPU clock cycles = instructions for a program \* Average clock cycles per instructions.**

### **Clock cycle per instructions (CPI):**

Average number of clock cycles per instructions for a program or program fragment

$$CPI = \frac{CPU\ clock\ cycles}{Instruction\ count}$$

### **Problem:**

Let us assume that two computers use same instruction set architecture. Computer A has a clock cycle time of 250ps and a CPI of 2.0 for some program and computer B has a clock cycle time of 500 ps and a CPI of 1.2 for the same program. Which computer is faster for this program and by how much?

**Answer:** Each computer executes the same number of instructions for the program call this number I. first find number of processor clock cycles for each computer:

1. CPU clock cycles = instructions for a program \* Average clock cycles per instructions.

$$CPU\ clock\ cycle\ A = I * 2.0$$

$$CPU\ clock\ cycle\ B = I * 1.2$$

2. Compute the cpu time for each computers

$$Cpu\ time\ A = cputime\ A * clock\ cycle\ time\ A$$

$$= I * 2.0 * 250\ ps$$

=500 Ips

Cpu time B = cpuclock cycle B \* clock cycle time B

= I \* 1.2 \* 500 ps

=600 Ips

3. Computer A is faster the amount faster is given by the ratio of the execution time

$$\frac{\text{performance A}}{\text{performance B}} = \frac{\text{Execution time B}}{\text{Executiontime A}} = n$$

$$\frac{600 \text{ Ips}}{500 \text{ Ips}} = 1.2$$

Computer A is 1.2 times faster than computer B for this program

**The classic cpu performance equation:**

**Cpu time = instruction count \* CPI \* clock cycle time**

Since the clock rate is the inverse of the clock cycle time.

$$\text{cpu time} = \frac{\text{instruction count} * \text{CPI}}{\text{clock rate}}$$

IC : the number of instructions executed by the program.

**problem:**

The table shows the two code sequence with number of instructions of different instruction classes within each code sequence respectively. The instruction are classified as A,B and C according to the CPI as shown in table.

- (i) Determine which code sequence executes the most instructions.
- (ii) Determine which code sequence will execute quickly.
- (iii) Determine the CPI for each code sequence

Code sequence	Instruction counts for each instruction class		
	A	B	C
1	4	2	4
2	8	2	2

**Solution :**

**(i) Code sequence execution:**

Code sequence 1 executes : 4+2+4 = 10 instructions

Code sequence 2 executes : 8+2+2 = 12 instructions

Therefore , code sequence 2 executes more instructions

**(ii) CPU clock cycles required to executes these code sequences is given as**

$$\text{CPU clock cycles 1} = (4*1) + (2*2) + (4*3) = 20 \text{ cycles}$$

CPU clock cycles 2 = (8\*1) + (2\*2) + (2\*3) = 18 cycles  
 Code sequence 2 is faster than code sequence 1.

### (iii) CPI for each code sequence

$$CPI = \frac{\text{CPU clock cycles}}{\text{Instruction count}}$$

$$CPI\ 1 = \frac{20}{10} = 2.0$$

$$CPI\ 2 = \frac{18}{12} = 1.5$$

Components of performance	Units of measure
Cpu execution time for a program	Seconds for the program
Instruction count	Instructions executed for the program
Clock cycles per instruction (CPI)	average no.of clock cycles per instruction
Clock cycle time	Seconds per clock cycle

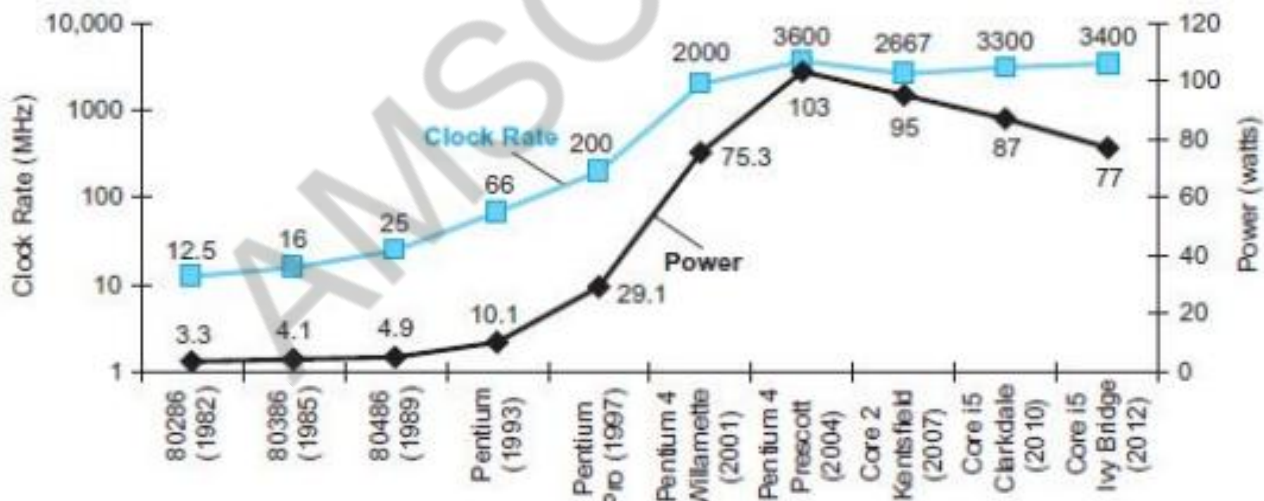
### 6.Power wall:

When processor runs at a high speed, it generates more & more power consumes.

When there is an increase clock rate there is increase in power consumed.

Power wall shows the increase in clock rate & power of eight generation of Intel microprocessor over 30 years.

### Clock rate and power for intel X86 processor



Pentium 4 made a dramatic jump in clock Rate and power but less in performance .Due to thermal problem.

Core 2 has simpler pipeline with lower clock rates and multiple processors per chip.

IC ( Integrated circuit) are called CMOS ( Complementary metal oxide semi conductor)

For CMOS the Primary source of energy consumption is called dynamic energy.

$$\text{Energy} = \text{capacity load} * \text{voltage}^2$$

The power consumed by a cpu is given by:

$$P = C V^2 f$$

Where C = Capacity loading, V = voltage applied, F= Running Frequency

**Problem:**

If a new processor has 85% of the capacitive load of old processor its supply voltage is reduced by 20% and new processor results in a 25% shrink in frequency. What is the impact on power consumption?

$$\begin{aligned} \frac{\text{Power (new Processor)}}{\text{Power (old Processor)}} &= \frac{(C * 0.85) * (V * 0.8)^2 * (F * 0.75)}{C V^2 f} \\ &= 0.85 * (0.8)^2 * 0.75 \\ &= 0.408 \end{aligned}$$

**The new processor uses only 40.8% of the power of the old processor**

**Power consumption can be addressed in the following ways:**

1. Lowering the power supply voltage to reduce power consumption
2. By using large cooling devices
3. Turning off part of chip that are not used in a given clock cycle

## **7. Uniprocessor to multiprocessors:**

The power limit has forced a dramatic change in the design of microprocessor.

To decrease the response time of a single program running on the single processor designer came up with multiple processors per chip.

The intention was to increase throughput rather than to decrease response time.

**Multicore microprocessor:**

- To avoid confusion between the word processor and microprocessors of such microprocessor are generally known as multicore microprocessor.
- Example: dual core, quad core microprocessors.
- Multicore processor imposed new challenges on the programming aspects.

**Writing programs to support parallelism is not a simple task for the following reasons:**

- It increases difficulty level of programming.
- Needs scheduling of subtask.
- Needs to synchronize tasks.
- Needs to maintain coordination between subtasks.

**Advantage:**

1. Improves cost/performance ratio.
2. System provides room for expansion.
3. Tasks are divided among the modules.
4. Reliability of the system.

**8. Instruction:**

- To command a computer hardware, you must speak in language.
- The words of a computer language called instruction.
- The collection of words is called instruction set.
- Machine instruction is in the form of binary codes.
- Each instruction of CPU has specific information field which are required to execute it.
- Such information fields of instruction is called element of instruction.

**Elements of instruction:**

1. Operation code: Specifies the operations to be performed.
2. Source/destination operand: Specifies the source/destination for the operand instruction.
3. Source operand address: Specified the instruction may require one or more source operands.
4. Destination operand address: The result stored in the destination operand
5. Next instruction address: To fetch the next instruction after completion of execution of current instruction

**Instruction types:**

- Data processing Instruction :transfer the data between memory and register.
- Arithmetic instruction performs arithmetic operation using numerical data.
- The logical instruction performs logical operation on the bits of a word.

**Data storage: memory instruction**

The data has transfer between memory and register.

**Data movement: data transfer instruction:**

The data has transfer between CPU register and I/O devices

**Control: test and branch instruction**

Test instruction tests the value of a data word. Branch instruction depends on decision made.

**Number of address:**

Computer may have instructions of different length containing varying number of address

**Three address:** Three address instruction can be represent 2 source operands and 1 destination operand

**Example:** Add A,B,C

**Two address:** Two address instruction can be represent 1 source operand and another operand act as a source as well as destination

**Example:** Add A,B

**One address:** One address instruction can be represent 1 source operand and accumulator act as a source as well as destination

**Example:** Add AC, B

**Zero address :** In this all the operands are defined implicitly.

**Example:**  $AC \leftarrow \overline{AC}$

**9. Operations and operand****Operations of computer hardware:**

- Instructions step by step instructions in a top down fashion.
- In MIPS processor (Advance Risk Machine (ARM)) , the declaration is always done by the register.
- It support 32 bit register.

**MIPS (ARM) Assembly language:**

Category	Instruction	Example	Meaning
Data transfer	LOAD WORD	Lw \$s1,20(\$s2)	\$s1= Memory [\$s2+20]
	STORE WORD	Sw \$s1, 20(\$s2)	Memory[\$s2+20]=\$s1
	LOAD HALF	Lh \$s1,20(\$s2)	\$s1= Memory [\$s2+20]
	STORE HALF	Sh \$s1, 20(\$s2)	Memory[\$s2+20]=\$s1
	LOAD BYTE	Lb \$s1,20(\$s2)	\$s1= Memory [\$s2+20]
	STORE BYTE	Sb \$s1, 20(\$s2)	Memory[\$s2+20]=\$s1
Arithmetic instructions	ADD	add \$s1, \$s2, \$s3	\$s1 = \$s2 + \$s3
	SUB	sub \$s1, \$s2, \$s3	\$s1 = \$s2 - \$s3
	Add immediate	addi \$s1, \$s2, 4	\$s1 = \$s2 + 4
Logical operations	And	and \$s1, \$s2, \$s3	\$s1 = \$s2 & \$s3
	Or	or \$s1, \$s2, \$s3	\$s1 = \$s2   \$s3
	Nor	nor \$s1, \$s2, \$s3	\$s1 = ~ (\$s2   \$s3)
	And immediate	andi \$s1, \$s2, 4	\$s1 = \$s2 & 4
	Or immediate	ori \$s1, \$s2, 4	\$s1 = \$s2   4
	Nor immediate	nori \$s1, \$s2, 4	\$s1 = ~ (\$s2   4)
	Shift left logical	sll \$s1, \$s2, 4	\$s1 = \$s2 << \$s3
	Shift right logical	srl \$s1, \$s2, 4	\$s1 = \$s2 >> \$s3
Conditional branch	Branch on equal	beq \$s1, \$s2, 5	If(\$s1 == \$s2) goto PC+25+100

	Branch on not equal	bne \$s1, \$s2, 5	If(\$s1 != \$s2) goto PC (procedure call)+25+100
	Set on less than	slt \$s1, \$s2, \$s3	If(\$s1 < \$s2) \$s1 = 1 else \$s1 = 0
	Set on less than immediate	slti \$s1, \$s2, 4	If(\$s1 < 4) \$s1 = 1 else \$s1 = 0
Unconditional branch	Jump	j 2500	goto 100
	Jump register	jr \$ ra	goto \$ra

### Operands of the computer hardware:

High level language, the operands of arithmetic instructions are restricted.

#### Three types of operands:

1. 32 register operand
2.  $2^{30}$  memory words operand
3. Constant or immediate operand

#### 32 register operand:

- Registers are primitives used in hardware design that are also visible to the programmers where the computer is completed.
- The size of register in the MIPS (ARM) architecture is 32 bits.

Register	0	1	2-3	4-7	8-15	16-23	24-25	26-27	28	29	30	31
Name	\$Zero	\$at	\$v0-\$v1	\$a0-\$a3	\$t0-\$t7	\$s0-\$s7	\$t8-\$t9	\$k0-\$k1	\$gp	\$sp	\$fp	\$ra

at – Reserved for assembler

v0 – v1 = value for results and expression evaluation

a0 – a3 = argument register

t0 – t7 = temporary register

s0 - s7 = saved register

t8 – t9 = more temporary register

k0 – k1 = reserved for operating system

gp = global pointer

sp = stack pointer

fp = frame pointer

ra = return address

#### Example: compiling a c assignment using register

f = (g + h) – (i + j)

add \$t0, \$s1, \$s2

add \$t1, \$s3, \$s4

sub \$s0, \$t0, \$t1

#### $2^{30}$ memory word operand:

- Accessed only by data transfer instructions. MIPS (ARM) uses byte addresses.
- So sequential word addresses differ by 4byte.
- Memory holds data, array, and spelled register.

Memory [0]	Memory[4]	.....	Memory [ 4294967292]
------------	-----------	-------	----------------------



### Example: compiling an assignment when an operand is in memory

$g = h + A[8]$

lw \$t0, 32(\$s2)      [ Effective address = base address + offset [offset = address \* 4 byte]]  
[ Effective address =  $0 + 8 * 4 = 32$ ]

Add \$s0, \$s1, \$t0

### Compiling using load and store:

$A[12] = h + A[8]$

LDR \$t0, 32(\$s1)

STR \$t0, 48(\$s1)

### Constant or immediate operands:

Program will use a constant in an operation.

In ARM arithmetic instructions have a constant as an operand.

### Example:

$a = b + 4$

addi \$s0, \$s1, 4

## 10. Representing instruction in the computer:

- Instructions are kept in computers as a series of high and low electric signals and represented as number.
- Each piece of an instruction can be considered as an individual number.
- Placing these number side by side forms the instruction.

### Instruction format:

A form of representation of an instruction composed of fields of binary numbers.

In MIPS ISA instructions fall into 3 categories

#### 1. R-format: register format

6	5	5	5	5	6
op	rs	Rt	Rd	Shamt	funct

Where:

Op: basic operation of the instruction, traditionally called the opcode

Rs: the first register source operand. Rs hold one of the argument of the operation

Rt: The second register source. Rt hold another arguments of the operation

Rd: The register destination operand. Rd stores the result of the operation.

Shamt: shift amount. Amount of bit to shift

Funct: function code. To specify the operation in addition to the opcode.

## 2. I-format: intermediate format

6	5	5	6
op	rs	rt	Constant or address

## 3. J – format: Jump format

6	36
op	Address

## Opcode and function code for each operation:

Operation	Opcode	Function code
add	0	32
Sub	0	34
Addi	8	
Lw	35	
Sw	43	
And	0	36
Or	0	37
Nor	0	39
Andi	12	
Ori	13	
Sll	0	0
Srl	0	2
Beq	4	
Bne	5	
Slt	42	
Slti	10	
Jump	2	

## Example: add \$t0, \$s1,\$s2 (R format)

R-forma	6	5	5	5	5	6
	op	rs	rt	Rd	shamt	funct

Decimal	0	17	8	25	0	32
---------	---	----	---	----	---	----

Binary	000000	10001	01000	11001	00000	100000
--------	--------	-------	-------	-------	-------	--------

## 2) lw \$t0, 32(\$s3) (I – format)

6	5	5	6
op	rs	rt	Constant or address

Decimal

35	19	8	32
----	----	---	----

Binary

100011	10011	01000	0000 0000 0010 0000
--------	-------	-------	---------------------

### 3.A[300] = h+A[300] (both R & I – formats)

Lw \$t0, 1200 (\$s2)

Add \$t1, \$s1, \$t0

Sw \$t1, 1200 (\$s2)

R-format	op	rs	rt	Address or constant		
I-format	op	rs	rt	rd	shamt	Funct
J-format	op	rs	rt	Address or constant		

Decimal representation:

35	18	8	1200		
0	17	8	9	0	32
43	9	18	1200		

Binary representation:

100011	10010	01000	0000010010110000		
000000	10001	01000	01001	00000	100000
101011	01001	10010	0000010010110000		

## 11. Logical Operation:

- An instruction in which the quantity being operated on and the result of the operation can have two values.
- Logic operations include any operations that manipulate *Boolean* values.
- Boolean values are either true or false. They can also be represented as 1 and 0. Normally, 1 represents true, and 0 represents false, but it could be the other way around.

### Types of logical operation:

1. AND operation,
2. OR operation
3. NOR operation
4. ANDI operation
5. SLL operation
6. SRL operation

**AND operation:** A logical bit by bit operation with two operands that calculation as 1 only if there is 1 in both the operation.

A	B	A ^ B
0	0	0
0	1	0
1	0	0
1	1	1

**Example: and \$s0,\$s1,\$s2 [ s1 =17 and s2 =35]**

S1 -> 0000 0000 0000 0000 0000 0000 0001 0001

S2 -> 0000 0000 0000 0000 0000 0000 0010 0011

-----

S0 -> 0000 0000 0000 0000 0000 0000 0000 0001

**OR operation:** A logical bit by bit operation with two operation with two operands that calculates 1 if there is a 1 in ether operand

A	B	A v B
0	0	0
0	1	1
1	0	1
1	1	1

**Example: and \$s0,\$s1,\$s2 [ s1 =17 and s2 =35]**

S1 -> 0000 0000 0000 0000 0000 0000 0001 0001

S2 -> 0000 0000 0000 0000 0000 0000 0010 0011

-----

S0 -> 0000 0000 0000 0000 0000 0000 0011 0011

**NOT OPERATION:** A logical bit by bit operation with one operand that inverts the bits, that is replaces every 1 with a 0 and every 0 with 1 .

A	A
0	1
1	0

**Example:**

**nor \$t0, \$t1, \$zero**

\$t1    0000 0000 0000 0000 0011 1100 0000 0000

\$t0    1111 1111 1111 1111 1100 0011 1111 1111

**Nor operation:** A logical bit by bit operation with two operations with two operands that calculates 1 if there is a 1 in ether operand

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

**Example: and \$s0,\$s1,\$s2 [ s1 =17 and s2 =35]**

S1 -> 0000 0000 0000 0000 0000 0000 0001 0001

S2 -> 0000 0000 0000 0000 0000 0000 0010 0011

-----  
S0 -> 1111 1111 1111 1111 1111 1111 1100 1100

**Andi operation:** A logical bit by bit operation with 1 operand and immediate value that calculation as 1 only if there is 1 in both the operation.

**Example: Andi \$s0,\$s1,9 [ s1 =17]**

S1 -> 0000 0000 0000 0000 0000 0000 0001 0001

9 -> 0000 0000 0000 0000 0000 0000 0000 1001

-----  
S0 -> 0000 0000 0000 0000 0000 0000 0000 0001

**SLL operation:** logical shift left. This instruction shift an operand by a number of the positions specified in a count operand in left side.

**Example: SLL \$s0,\$s1,6 [ s1 =18]**

S1 -> 0000 0000 0000 0000 0000 0000 0001 0010

S0 -> 0000 0000 0000 0000 0000 0100 1000 0000

$$18 * 2^i = 18 * 2^6 = 1152$$

**SRL operation:** logical shift right. This instruction shift an operand by a number of the positions specified in a count operand in right side.

**Example: SRL \$s0,\$s1,6 [ s1 =1152]**

S1 -> 0000 0000 0000 0000 0000 0100 1000 0000

S0 -> 0000 0000 0000 0000 0000 0000 0001 0010

$$1152 / 2^n = 18 / 2^6 = 18$$

## 12. Control operation:

### 1. Decision Making (branch instruction)

- A computer from a simple calculator is its ability to make decisions
- Decision making is commonly represented in programming languages using the *if* statement, sometimes combined with *go to* statements and labels.

**MIPS assembly language includes two decision-making instructions:**

#### 1. **beq register1, register2, L1**

If the value in register1 equals the value in register2 go to labeled statement L1. The mnemonic beq stands for *branch if equal*.

#### 2. **bne register1, register2, L1**

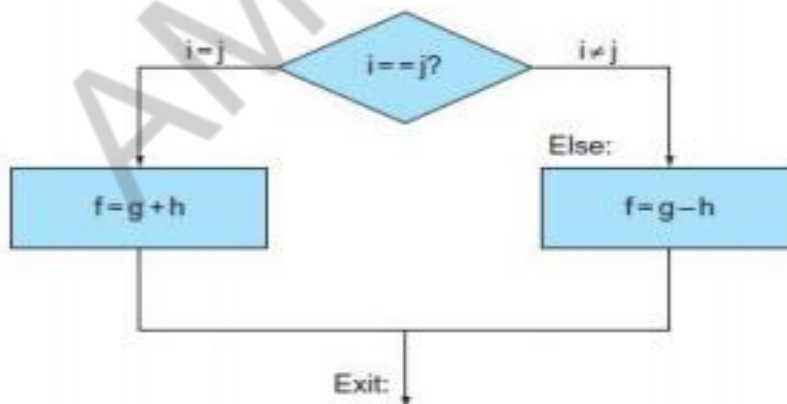
If the value in register1 does *not* equal the value in register2 go to the statement labeled L1. The mnemonic bne stands for *branch if not equal*.

These two instructions are traditionally called **conditional branches**.

### **Compiling if-then-else into Conditional Branches**

In the following code segment, f, g, h, i, and j are variables.

```
if (i == j)
    f = g + h;
else f = g - h;
```



If the five variables f through j correspond to the five registers \$s0 through \$s4, what is the compiled MIPS code for this C *if* statement?

In general, the code will be more efficient if we test for the opposite condition to branch over the code that performs the subsequent *then* part of the *if* (the label Else is defined below)

and so we use the branch if registers are *not* equal instruction (bne):

```
bne $s3,$s4,else      # go to Else if i ≠ j
add $s0, $s1, $s2      # f = g + h (skipped if i ≠ j)
j exit                # go to Exit
else: sub $s0, $s1, $s2  # f = g - h (skipped if i = j)
exit
```

### 3. LOOP :

Decisions are important both for choosing between two alternatives

1. found in *if* statements
2. found in loops.

The same assembly instructions are the building blocks for both cases.

#### Compiling a *while* Loop in C

```
while (save[i] == k)
    i += 1;
```

Assume that *i* and *k* correspond to registers \$s3 and \$s5 and the base of the array *save* is in \$s6. What is the MIPS assembly code corresponding to this C segment?

#### branch back to that instruction at the end of the loop :

```
Loop:    sll $t1,$s3,2      # Temp reg $t1 = i * 4
         add $t1,$t1,$s6    # $t1 = address of save[i]
         lw $t0,0($t1)      # Temp reg $t0 = save[i]
         bne $t0,$s5, Exit  # go to Exit if save[i] ≠ k

         addi $s3,$s3,1     # i = i + 1
         j Loop             # go to Loop
Exit:
```

#### Explanation:

1. To get the address of *save[i]*, we need to add \$t1 and the base of *save* in \$s6.
2. Now we can use that address to load *save[i]* into a temporary register:
3. The next instruction performs the loop test, exiting if *save[i]* ≠ *k*:
4. The next instruction adds 1 to *i*:
5. The end of the loop branches back to the *while* test at the top of the loop. We
6. just add the Exit label after it, and we're done

### 3. Case/Switch Statement

Most programming languages have a *case* or *switch* statement that allows the programmer to select one of many alternatives depending on a single value.

The simplest way to implement *switch* is via a sequence of conditional tests, turning the **switch** statement into a chain of *if-then-else* statements.

Sometimes the alternatives may be more efficiently encoded as a table of addresses of alternative instruction sequences, called a **jump address table** or **jump table**.

#### **Jump table:**

- The jump table is then just an array of words containing addresses that correspond to labels in the code.
- The program loads the appropriate entry from the jump table into a register.
- It then needs to jump using the address in the register.
- To support such situations, computers like MIPS include a *jump register* instruction (*jr*), meaning an unconditional jump to the address specified in a register. Then it jumps to the proper address using this instruction.
- **jump address table** Also called **jump table**. A table of addresses of alternative instruction sequences

**Example: j lable // Jump to lable**

**Jr \$s1 // Jump to address present in register**

### **13. Addressing modes: MAY/JUNE 2016, APR/MAY 2015, NOV/DEC 2015**

Addressing modes are the way of specifying an operand or memory address in an instruction.

The different ways in which the location of an operand is specified in an instruction are called address modes.

#### **Types of addressing modes:**

1. Register addressing mode
2. Immediate addressing mode.
3. Base or displacement addressing mode
4. Pc-relative addressing mode
5. Pseudo- direct addressing mode
6. In direct addressing mode
7. Auto increment addressing mode
8. Auto decrement addressing mode

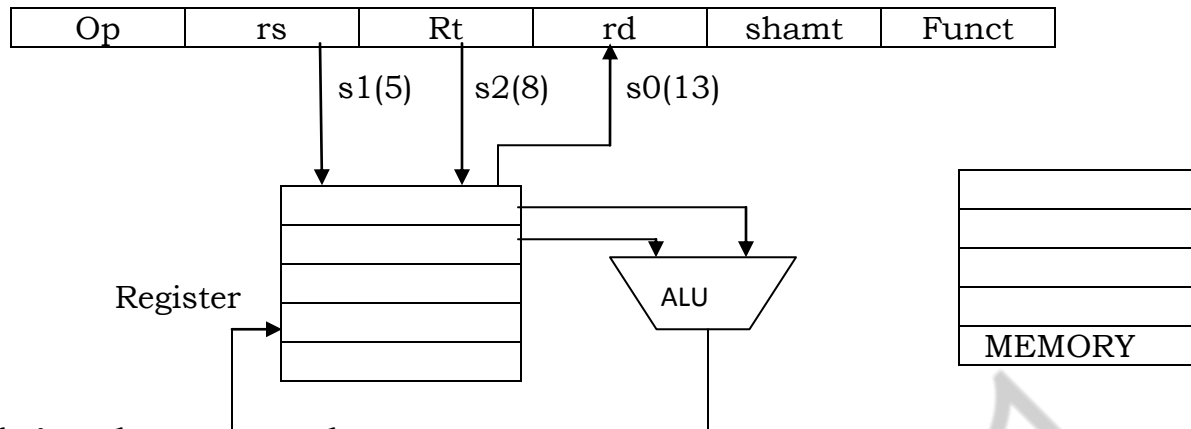
#### **Register addressing mode:**

- Is the considered the simplest addressing mode.
- Because the operands are in register.
- It allows the instructions to be executed much faster.



- It is a form of direct addressing.

Example: add \$s0, \$s1, \$s2                      where s1=5, s2=8

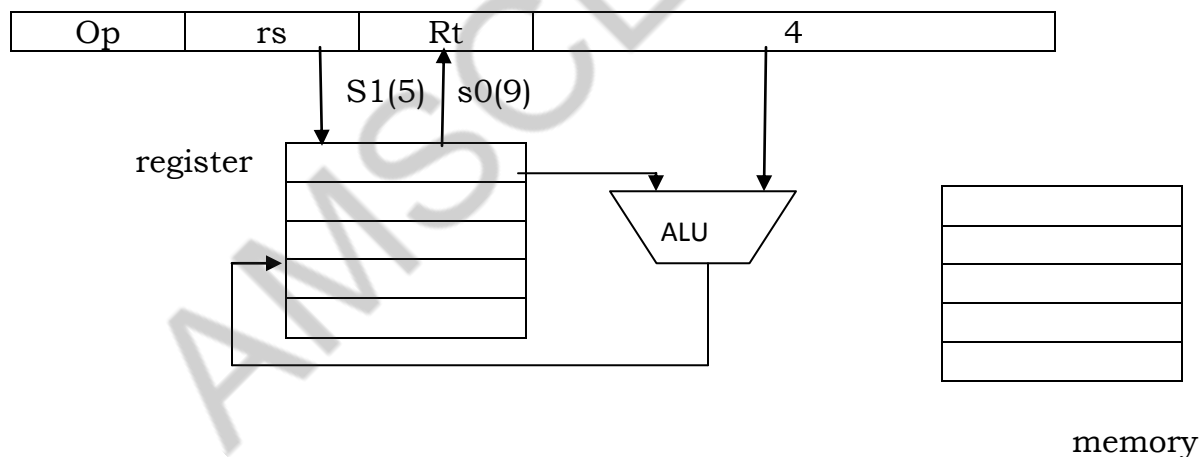


## Explain the example

### Immediate Addressing Mode:

- MIPS immediate addressing means that one operand is a constant within the instruction itself.
- The advantage of using it is that there is no need to have extra memory access to fetch the operand.
- But keep in mind that the operand is limited to 16 bits in size.

Example: add \$s0, \$s1, 4                      s1=5

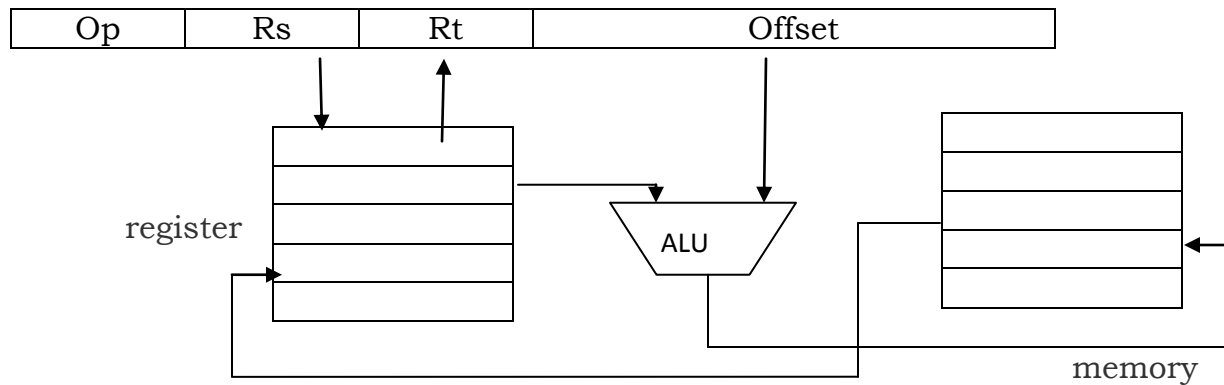


## Explain the example

### Base or displacement addressing mode:

- Base address is a data or instruction memory location is specified as a signed offset from a register.
- It is also known as indirect addressing; a register act as a pointer to an operand located at the memory location whose address is in the register.
- The address of the operand is the sum of the offset value and the base value.
- The size of the operand is limited to 16 bits.

Example: lw \$t0, 32(\$s1)



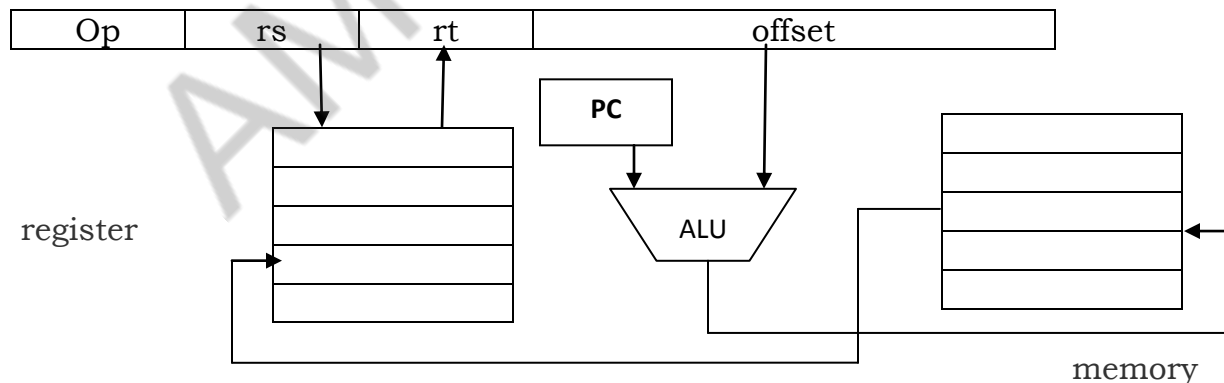
## Explain the example

### PC- relative addressing mode:

- It is also known as program counter addressing.
- It is a data or instruction memory location is specified as an offset relative to the incremented PC.
- It is usually used in conditional branches.
- Pc stores the address of next instruction to be fetched.
- It offset value can be an immediate value or an interpreted label value.  
It implements position independent code.

### Example: beq \$s0, \$s1, label

Address instruction  
 4008 addi \$s0,\$s1, 1 // Offset  
 4012 beq \$s0, \$s1, **Label** // this condition true move the control to the address (4024)  
 4016 sub \$s0, \$s1, \$s2 // Pc hold address of next instruction pc= 4016  
 4020 addi \$s2,\$s3, 1  
 4024 **Label** addi \$s1,\$s2 ,4 // EA = (pc + offset ) = 4016 + 4008 =4024.

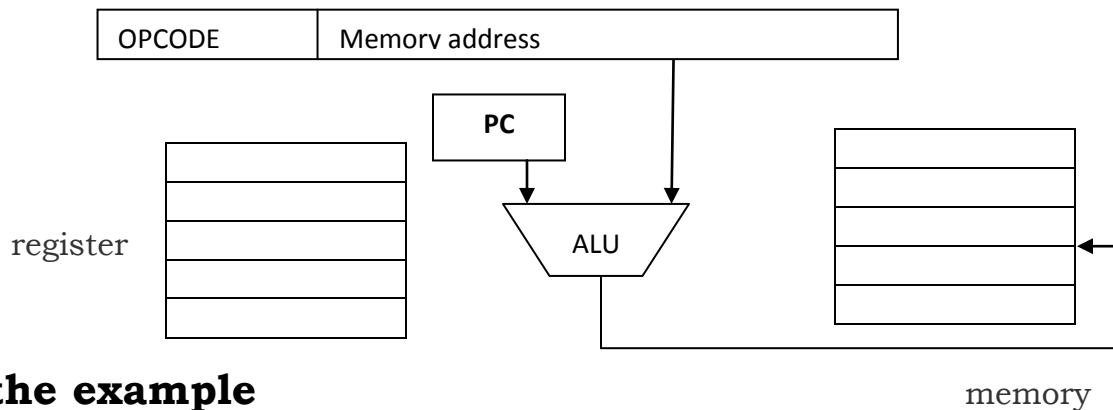


## Explain the example

### Pseudo direct Addressing mode:

- It is the memory address which (mostly) embedded in the instructions.
- It is specifically used for J-type instructions, j and jal.
- The instruction format is 6 bits of opcode and 26 bits for the immediate value.
- The effective address will always be a word aligned.

**Example: j label.**



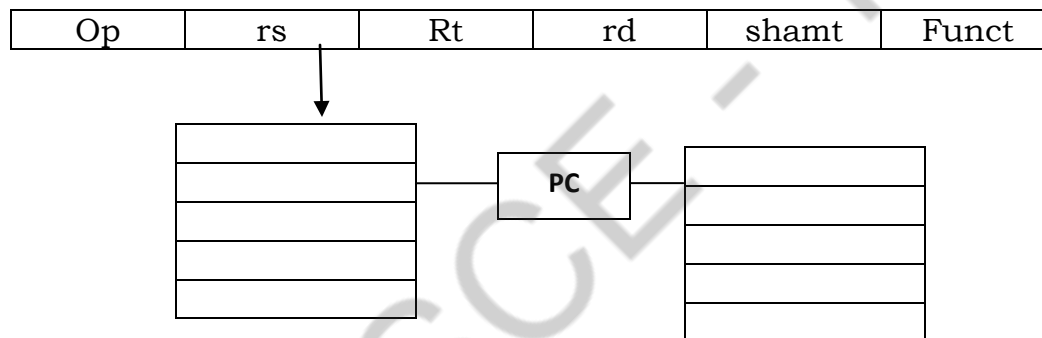
## Explain the example

### Indirect Addressing Mode:

It is also called register direct addressing mode

In this mode, the instruction contains the address of memory which refers the address of the operand.

**Example: j \$s1 // s1= 4008 (address)**



## Explain the example

### Auto increment addressing mode:

After accessing the operand, the content of this register are incremented to address the next location

**Example: Mov R0,(R2)+**

### Auto decrement addressing mode

The content of register specified in the instruction are first decremented and then used as an effective address of the operand

**Example : Mov – (R0),R2**

AMSCE - 1101

## UNIT-2

### Arithmetic operations

#### **1. Introduction:**

ALU is responsible for performing arithmetic operations such as add, subtract, division and multiplication and logical operation such as AND, OR, Inverting etc.

Arithmetic operation to be performed is based on data type.

#### **Two basic data types:**

1. Fixed point numbers
2. Floating point numbers

#### **Fixed point number:**

It allows the representation of number positive or negative integer numbers.

The length of 1, 2, 4 or more bytes.

#### **Floating point numbers:**

It allows the representation of number having both integer part and fractional part.

The length of single precision (4 bytes) or double precision (8 bytes).

#### **Big-Endian and Little-Endian Assignments:**

#### **Two ways that byte addresses can be assigned across words**

##### **1. Big-Endian Assignments:**

When lower byte addresses are used for the more significant bytes (leftmost bytes) of the word.

##### **2. Little-Endian Assignments:**

When lower byte addresses are used for the less significant bytes (rightmost bytes) of the word.

0 1 1 0 0 0 1 0

## 2. Addition and Subtraction:

We can relate addition and subtraction operations of numbers by the following relationship:

$$(\pm A) - (+B) = (\pm A) + (-B) \text{ and}$$

$$(\pm A) - (-B) = (\pm A) + (+B)$$

Therefore we can change subtraction operation to an addition operation by changing the sign of the subtrahend.

### Binary Addition:

Adding two single digit binary numbers

**Rules for binary additions are as follows:**

A	B	SUM(A+B)	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

**Example:** 10+3

$$\begin{array}{r} 10 \quad 1010 \\ 3 \quad 0011 \\ \hline 13 \quad 1101 \end{array}$$

The logical circuit which performs this operation is called a half adder.

The logical circuit which performs addition of 3 bits( 2 significant bits and a previous carry) is called full adder.

### Half adder:

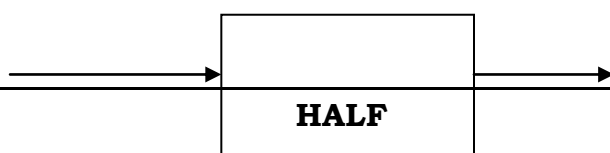
**Performs the most basic digital arithmetic operation, that is, the addition of two binary numbers.**

**The half-adder requires two outputs because the sum  $1 + 1$  is binary 10. The two inputs are called S (for sum) and C (for carry out).**

**Truth Table:**

A	B	SUM(A+B)	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

**Block diagram:**



**A**

**Sum = A+B**

**B**

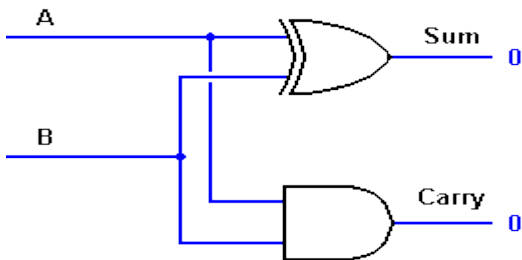
**Carry =AB**

**K-map**

**SUM =A  $\oplus$  B**

**CARRY= x • y**

**Logical circuit:**



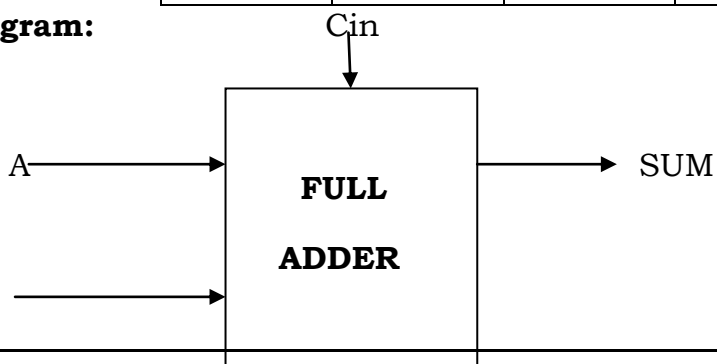
**FULL ADDER:**

When adding more than one bit, must consider the carry of the previous bit – full-adder has a “carry-in” input.

• **Truth Table:**

A	B	Cin	SUM(A+B)	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
1	0	0	1	0
0	1	1	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

**Block diagram:**





B

↓  $C_{out}$

## K-map simplification for carry and sum

For Carry ( $C_{out}$ )

$BC_{in}$	00	01	11	10
A				
0	0	0	1	0
1	0	1	1	1

$$C_{out} = AB + A C_{in} + B C_{in}$$

For Sum

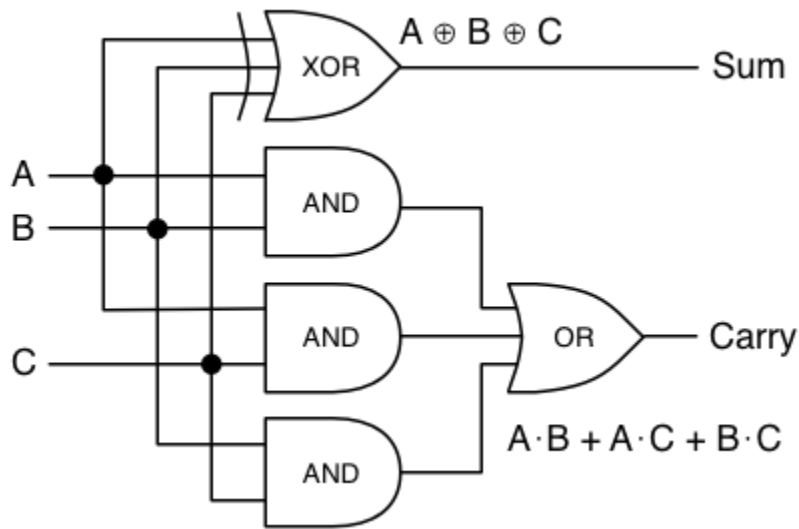
$BC_{in}$	00	01	11	10
A				
0	0	1	0	1
1	1	0	1	0

$$Sum = \bar{A} \bar{B} C_{in} + \bar{A} B \bar{C}_{in} + A \bar{B} \bar{C}_{in} + A B C_{in}$$

$$Sum = \bar{A} \bar{B} C_{in} + \bar{A} B \bar{C}_{in} + A \bar{B} \bar{C}_{in} + A B C_{in}$$

Cout:

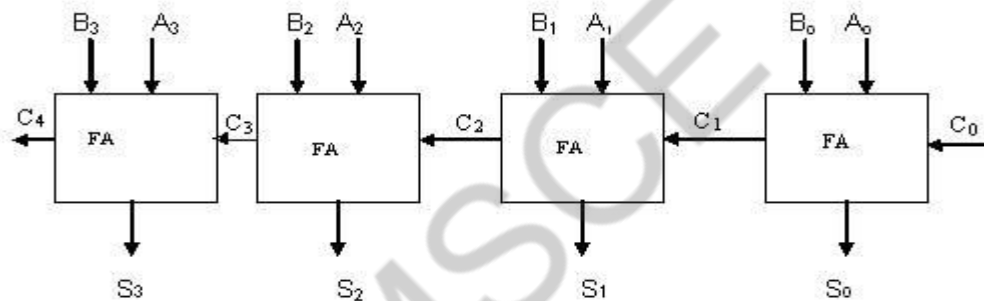
**Logical circuit:**



### Parallel Adder:

A n-bit parallel adder can be constructed using number of full adder circuits connected in parallel.

### Block Diagram:



The block diagram of the n-bit parallel adder using n number of full adder circuits connected in cascade.

The carry output of each adder is connected to the carry input of the next higher-order adder.

### Binary Subtraction:

Adding two single digit binary numbers

Rules for binary additions are as follows:

A	B	Diff	Borrow
0	0	0	0
0	1	1	1

1	0	1	0
1	1	0	0

### One's complement subtractions:

#### Case1: both numbers positive

$$\begin{array}{r}
 28 \quad 011100 \\
 +15 \quad 001111 \\
 \hline
 43 \quad 101011
 \end{array}$$

#### Case 2: Subtraction of smaller number from larger number

1. Determine the 1's complement of the smaller number
2. Add the 1's complement to the larger number
3. If end around carry is generated add it to the result

#### Example:

$$\begin{array}{r}
 28 \quad 011100 \\
 -15 \quad 110000 \quad (001111 \rightarrow 1's \text{ com}(110000)) \\
 \hline
 1 \mid 001100 \\
 \hline
 \quad \quad \quad 1 \\
 \hline
 13 \quad 001101
 \end{array}$$

#### Case 3: Subtraction of larger number from smaller number

1. Determine the 1's complement of the larger number.
2. Add the 1's complement to the smaller number.
3. If end around carry is generated discard the carry.
4. The result will be in 1's complement form .to get the result in true form, take the 1's complement of the result.

#### Example:

$$\begin{array}{r}
 15 \quad 001111 \\
 -28 \quad 100011 \quad (011100 \rightarrow 1's \text{ com}(100011)) \\
 \hline
 1 \mid 10010 \rightarrow 1's \text{ complement} \\
 \hline
 \quad \quad \quad
 \end{array}$$

-13    0 0 1 1 0 1

**Case 4:** Both negative

1. Determine the 1's complement of the both numbers.
2. Add the 1's complement to the both numbers.
3. If end around carry is generated add it to the result.
4. The result will be in 1's complement form .to get the result in true form, take the 1's complement of the result.

-28    1 0 0 0 1 1                    (011100-> 1's com ( 100011))

-15    1 1 0 0 0 0                    (001111-> 1's com( 110000))

-43    1 0 1 0 0 1 1

\_\_\_\_\_ 1

0 1 0 1 0 0    1's complement

1 0 1 0 1 1

**Two's complement subtractions:**

**Case1: both numbers positive**

28    0 1 1 1 0 0

+15    0 0 1 1 1 1

43    1 0 1 0 1 1

**Case 2: Subtraction of smaller number from larger number**

1. Determine the 2's complement of the smaller number
2. Add the 2's complement to the larger number
3. If end around carry is generated discard the carry

**Example:**

28    0 1 1 1 0 0

-15    1 0 0 0 0 1                    (001111-> 2's com ( 100001))

1 0 0 1 1 0 1

**Case 3:** Subtraction of larger number from smaller number

1. Determine the 2's complement of the larger number.
2. Add the 2's complement to the smaller number.
3. The result will be in 2's complement form .to get the result in true form, take the 2's complement of the result.

**Example:**

$$\begin{array}{r} 15 \quad 001111 \\ -28 \quad 111100 \quad (011100 \rightarrow 2's \text{ com } (111100)) \\ \hline \quad 1001011 \rightarrow 2's \text{ complement } (001011 \rightarrow 001101) \\ \hline -13 \quad 001101 \end{array}$$

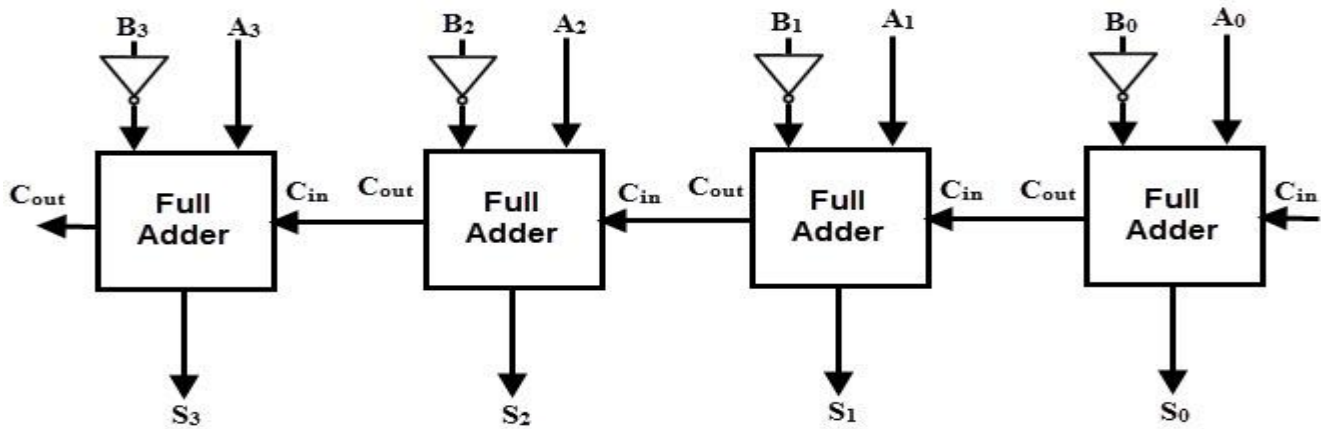
**Case 4:** Both negative

1. Determine the 2's complement of the both numbers.
2. Add the 2's complement to the both numbers.
3. If end around carry is generated discard the carry
4. The result will be in 2's complement form .to get the result in true form,

$$\begin{array}{r} -28 \quad 100011 \quad (011100 \rightarrow 1's \text{ com } (100011)) \\ -15 \quad 110000 \quad (001111 \rightarrow 1's \text{ com } (110000)) \\ \hline -43 \quad 1010011 \quad (\text{take } 2's \text{ complement}) \\ \hline \quad 101011 \end{array}$$

**Parallel Subtraction:**

- The subtraction  $A \rightarrow B$  can be done by taking the 2's complement of B and adding it to A.
- The 2's complement can be obtained by taking the 1's complement and adding one to the least significant pair of bits.
- The 1's complement can be implemented with inverters and a one can be added to the sum through the input carry to get 2's complement .



### Addition/Subtraction logical unit:

- Hardware implementation for integer addition and subtraction. IT consists of n-bit adder, 2's complement circuit, overflow detector logic circuit and AVF(overflow flag).
- Number a and b are the inputs for the n-bit adder.

### Addition operation:

- To add a and b, the add or subtract control line is set to Zero.
- Number b is given as one of the input to the n-bit adder along with the carry in signal  $C_0 = 0$  and added with number a.

**Result:  $R = a + b + 0$**

### Subtraction operation:

- To subtract a and b, the add or subtract control line is said to one.
- Number b is converted into 2's complement form (i.e) all bits of number b are complemented and added with carry in signal  $C_0 = 1$ .

**Result :  $R = a + \bar{b} + 1$**

### Overflow in integer Arithmetic:

Overflow can occur only when adding two numbers with same sign.

The carry bit from the MSB position is not sufficient indicator of overflow when adding signed numbers.

When both operands a and b have the same sign, an overflow occurs when the sign of result does not agree with sign of a and b.

**AVF** : Add overflow flip flop holds the overflow bit when A & B are added.

**The logical expression to detect overflow can be given as:**

$$\text{Overflow} = a_{n-1} b_{n-1} \bar{R}_{n-1} + \bar{a}_{n-1} \bar{b}_{n-1} R_{n-1}$$

Where:

$a_{n-1}$  = MSB of number a

$b_{n-1}$  = MSB of number b

$R_{n-1}$  = MSB of the result

**Overflow detector logic circuit:**

**Example:**

Both numbers positive:

+7    0 1 1 1

+3    0 0 1 1

1 0 1 0 (take 2's complement 1 0 1 0 = 0 1 1 0)

Result is -6, It is wrong due to overflow

**Flow Chart:**

Step 1: As & Bs are compared by a XOR gate.

Step 2 : if output =0 sign are identical, If Output =1 signs are different.

Step 3 : for addition operation same signs dictate addition of magnitudes.

Addition operation

Step 1: Magnitudes are added with a micro operation.  $EA \leftarrow A+B$  [ EA register combine A & E]

step 2: If E=1 overflow occurs and its transfer to AVF

Subtraction operation

Step1: Subtraction different sign dictate subtraction of magnitude.

Step 2: magnitudes are subtracted with a micro operation  $EA \leftarrow A-B+1$ . No overflow occurs so AVF =0.

Step 3 : if overflow occurs E=1 indicates  $A \geq B$  and the number in A is correct result E=0 indicates  $A < B$ . so we takes 2's complement of A

**Example : give any cases of addition and subtraction operation**

### **3. Multiplication:**

#### **Multiplication of unsigned (positive) Numbers**

- Multiplication is a complex operation than addition and subtraction.
- It can perform in hardware and software.
- The multiplication processor involves generation of partial products, one for each digit in the multiplier.
- When multiplier bit is 0 , the partial product is 0.
- When multiplier bit is 1 , the partial product is the multiplicand.
- Final product is produced by adding the partial products.

**Example:**

1011	Multiplicand (11 dec)
x 1101	Multiplier (13 dec)
<hr/>	
1011	Partial products
0000	Note: if multiplier bit is 1 copy



### Hardware implementation of unsigned binary multiplication:



Count =1

0

0 1 0 0

1 1 1 1

shift

Count=0

1

0 0 0 1

1 1 1 1

add

Shift

0

1 0 0 0

1 1 1 1

## Booth Multiplication Algorithm

MAY/JUNE 2016,NOV/DEC 2015,

APR/MAY2015

### **Signed(negative) multiplication - Booth's Algorithm :**

- A powerful algorithm for sign multiplication is a booth algorithm.
- This algorithm used to reduce number of operations required for multiplication by representing multiplier as a difference between 2 numbers.

### **Three Schemes used in Booth's Algorithm:**

1. Booth algorithm recording schemes
2. Hardware implementation of booth's algorithm
3. Bit pair recording schemes

### **Booth algorithm recording schemes:**

- +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.
- 0 times the shifted multiplicand is selected none of the above two cases.
- Implies 0 to right of the multiplier LSB.

### **Example:**

1 0 1 1 0 0    0 implied zero

-1 1 0 -1 0 0 (record multiplier using right shift)

### **Example: Multiply 0 1 1 1 0 (+14) and multiplier 1 1 0 1 1 (-5)**

1 1 0 1 1    0 (find record multiplier, apply implied 0 and shift the bit)

0 -1 1 0 -1    // record multiplier

Perform multiplication:

0 1 1 1 0 (+14)

0 -1 1 0 -1 (record multiplier of (-5))

1 1 1 1 | 1 0 0 1 0 ← 2's complement (-1 means take 2's complement of multiplicand)

0 0 0 | 0 0 0 0 0 X

0 0 | 0 1 1 1 0 X X

1 | 1 0 0 1 0 X X X ← 2's complement

0 0 0 0 0 X X X X

1 1 0 1 1 1 0 1 0 (-70)

**14 \* -5 = -70**

**-70 take 2's complement**

**256 128 64 32 16 8 4 2 1**

**0 0 1 0 0 0 1 0 0 ← 70**

**1 1 0 1 1 1 0 1 1 ← 1's complement**

**+1 ← 2's complement**

**1 1 0 1 1 1 0 1 0 (-70)**

**Hardware implementation of booth's algorithm:**

The Booth's algorithm can be implemented as shown. It consists of n-bit adder, shift, add subtract control logic and four registers A, B, Q, Q<sub>-1</sub>

Multiplier and multiplicand are loaded into register Q and register B, respectively.

Register A and Q<sub>-1</sub> are initially set by 0.

Sequence counter (SC) is set to number n equal to number of bits in the multiplier.

The n-bit adder performs addition of two inputs. One is A register and other is multiplicand.

**Addition operation:**

Add/Sub line is set 0, therefore cin = 0 and multiplicand is directly applied as a second input A register to the n-bit adder.

**Subtraction operation:**

Add/Sub line is set 1, therefore cin = 1 and multiplicand is complemented and then to the n-bit adder. The complement of multiplicand is added in the A register.

The shift , add and subtract control logic scans bits  $Q_0$  and  $Q_{-1}$  one at a time and generates the control signals

**Truth table for shift , add, and subtract control logic:**

$Q_0$	$Q_{-1}$	Add/Sub	Add/Sub Enable	Shift
0	0	X	0	1
0	1	0	1	1
1	0	1	1	1
1	1	X	0	1

**Flow chart: (algorithm)**

The sequence of events in booth's algorithm can be explained with the help of flowchart and algorithm.

Step 1: Load multiplicand and multiplier B and Q register and initially set zero in A &  $Q_{-1}$  register

Step 2: check the status of  $Q_0$  ,  $Q_{-1}$ ,

if  $Q_0 Q_{-1} = 10$  perform  $A \leftarrow A - B$

if  $Q_0 Q_{-1} = 01$  perform  $A \leftarrow A + B$

if  $Q_0 Q_{-1} = 00$  or  $11$  perform shift operation

Step 3: Arithmetic shift right operation perform from A, $Q$ , $Q_{-1}$  registers. And decrement Sequence Count (SC) by 1.

Step 4: check count. If count is zero end the process. Else repeat steps 2 and 3

Count = 0    

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

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

1
---

    Shift



1 1 0 0 0 1 X X X X  $\leftarrow$  (-1 : 2's complement)

1 1 0 1 1 0 1 0 1 0 (-150)

#### 4. Division: NOV/DEC 2015, NOV/DEC 2014

- The reciprocal operation of multiply is divide.
- The division process for binary numbers is similar to the decimal numbers.
- Divide's two operands called dividend and divisor and the results called quotient and remainder.

##### Formula:

$$\text{Dividend} = \text{Quotient} * \text{divisor} + \text{Remainder}$$

##### Division processor:

- The bit of divided are examined from left to right, until the set of bits examined represents a number greater than or equal to the divisor.
- The condition occurs, 0's are placed in the quotient from left to right.
- The condition is satisfied, 1 is placed in the quotient and the divisor is subtracted from partial dividend.
- The result is referred to as a partial remainder.

##### Example:

Handwritten binary division example:

Divisor  $\rightarrow$  1011

Dividend  $\rightarrow$  10010011

Quotient  $\rightarrow$  00001101

Partial Remainders  $\rightarrow$  001110, 001111

Remainder  $\rightarrow$  100

The diagram shows the long division process. The divisor 1011 is subtracted from the dividend 10010011. The quotient is 00001101. The partial remainders are 001110 and 001111. The final remainder is 100.

##### Types of division Algorithm:

1. Restoring Division Algorithm.



## 2. Non Restoring Division Algorithm.

### Restoring Division Algorithm:

- The hardware implementation for restoring division.
- It consists of  $n+1$  bit binary adder, shift, add and subtract control logic and registers A, B & Q
- Dividend and divisor are loaded into register B and Register Q.
- Register A is initially set to zero
- Division operation is carried out.
- After the division is completed, the  $n$  bit quotient is in register & the remainder is in register A.

### Flow chart:

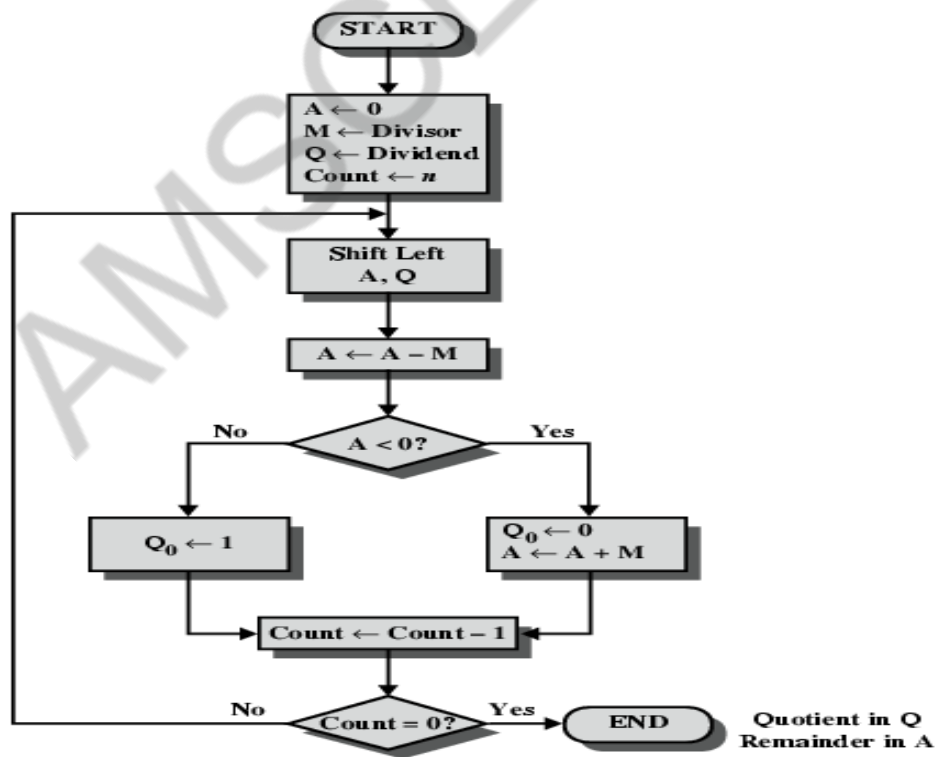
Step 1: Load dividend and divisor Q and B register and initially set zero in A register

Step 2: Shift A & Q left one binary position

Step 3: Subtract divisor (add 2's complement of divisor (B)) from A & place answer back in A ( $A \leftarrow A - B$ )

Step 4: In the sign bit of A is 1, set  $Q_0$  to 0 & add divisor back to A (that is , resorted); otherwise , set  $q_0$  to 1.

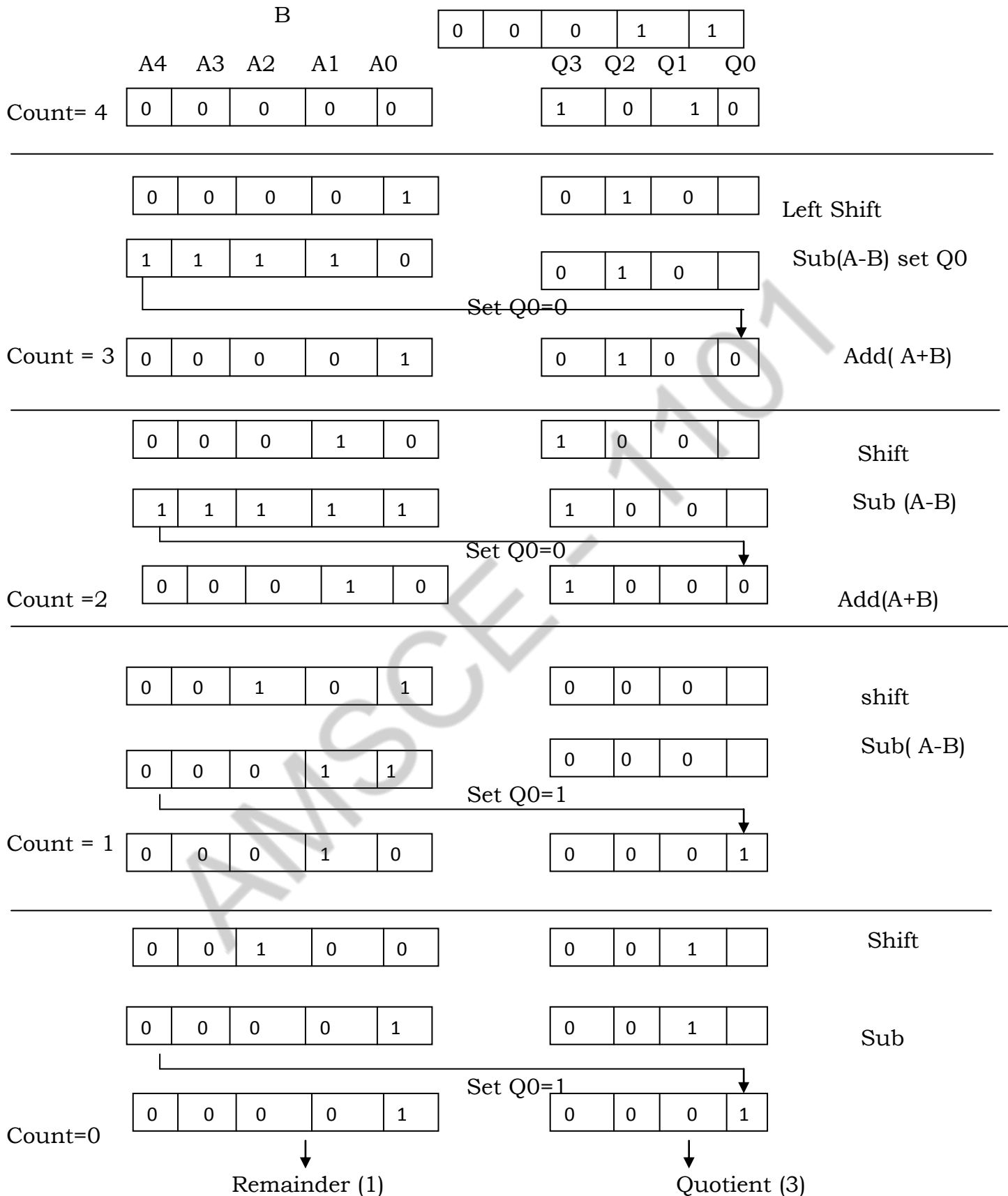
Step 5: Repeat steps 2 and 4  $n$  times.



### Example:

Dividend = 10 (1010)

Divisor = 3 (0011) ( if it is negative value take Take two complements ( 0 0 1 1 = 1 1 0 0 + 1 = 1 1 0 1 )



### Non –Restoring Division Algorithm:

- The hardware implementation for restoring division.
- It consists of  $n+1$  bit binary adder, shift, add and subtract control logic and registers A, B & Q.

### Draw Restoring Division algorithm diagram:

- Dividend and divisor are loaded into register B and Register Q.
- Register A is initially set to zero
- Division operation is carried out.
- After the division is completed, the  $n$  bit quotient is in register & the remainder is in register A.

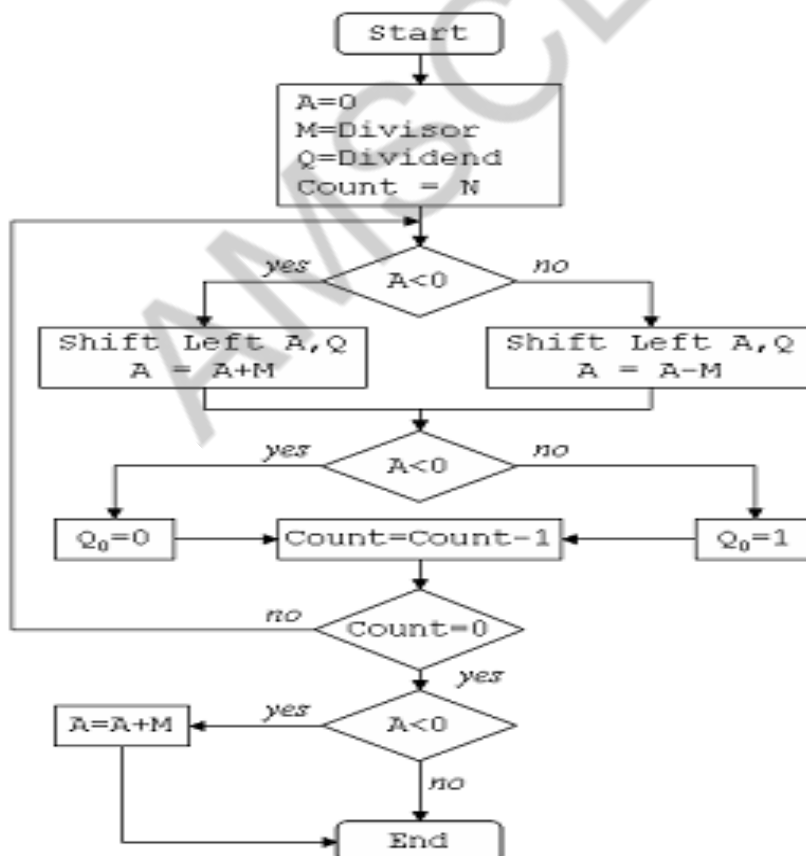
### Flow Chart:

Step 1: Load dividend and divisor Q and B register and initially set zero in A register

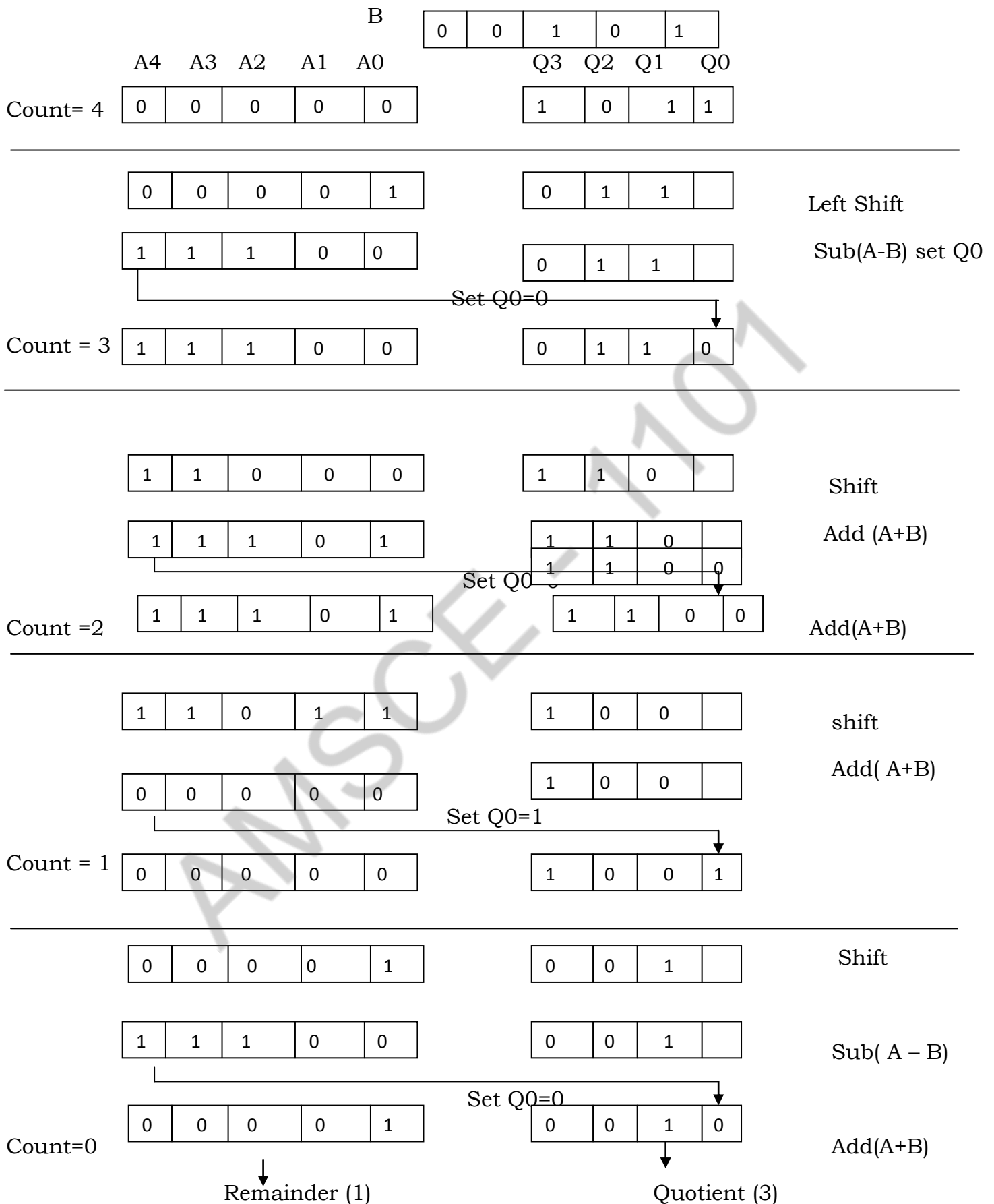
Step 2: If the sign bit of A is 0, shift A and Q left one bit position and subtract division from A; otherwise, shift A and Q left and add divisor to A. If the sign bit of A is 0 set  $Q_0$  to 1; otherwise set  $Q_0$  to 0. Shift A & Q left one binary position

Step 3: Repeat steps 1 and 2 for  $n$  times.

Step 4: In the sign bit of A is 1, add divisor to A.



**Example: dividend = 11 (1011) and divisor =5 (0101)**



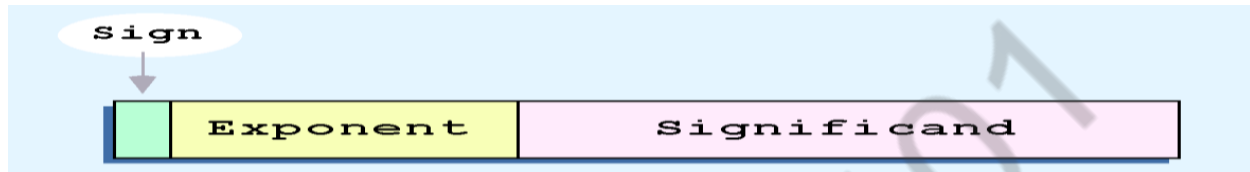
**Final product :** (11/ 5) remainder =1 and quotient = 3

## 5. Floating Point Representation:    APR/MAY 2016, APR/MAY 2015

The binary point is said to float and the numbers are called floating point numbers.

**It has represented 3 fields:**

- Computers use a form of scientific notation for floating-point representation
- Computer representation of a floating-point number consists of three fixed-size fields:



- Sign Field : The one-bit sign field is the sign of the stored value.
- Exponents Field: The size of the exponent field determines the range of values that can be represented.
- Significant Field : The size of the significant determines the precision of the representation

**Example:**

1.11101010110 X 2<sup>4</sup>

In this the

Sign field = 0

Mantissa field = 11101010110

Exponent field = 4

Scaling factor = 2

**IEEE Standard for floating point numbers:**

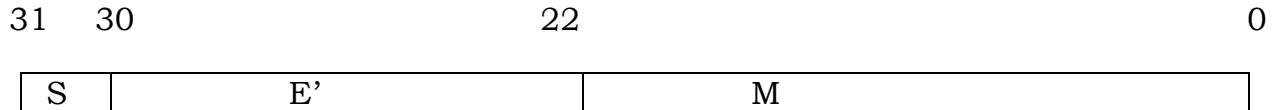
- The IEEE has established a standard for floating-point numbers.
- Institute of Electrical and Electronics Engineering
- IEEE format can be represented in two precisions.

**Two precisions:**

1. Single precision
2. Double precision

### Single precision:

- The IEEE-754 single precision floating point standard occupies a single 32 bit words.



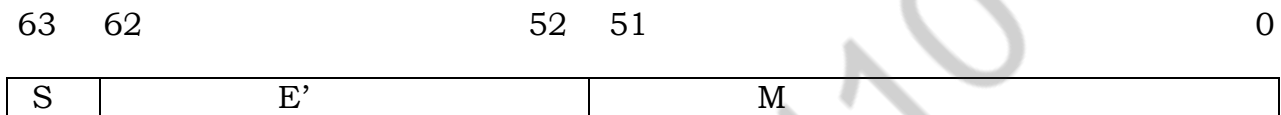
S : sign of number. 0- signifies +ve, 1- signifies -ve

E' : 8 bit signed exponent in excess-127 representation.

M : 23 bit mantissa fraction

### Double precision:

- The IEEE-754 double precision occupies a two 32 bit words.



S : sign of number. 0- signifies +ve, 1- signifies -ve

E' : 11 bit signed exponent in excess-1023 representation.

M : 52 bit mantissa fraction

**Example: Represent - 125.125<sub>10</sub> in single precision and double precision formats.**

**Step1:** convert decimal number in binary format.

2	125			0.125 * 2 = 0.250	LSD
2	62	- 1	↑ LSD	0.25 * 2 = 0.50	
2	31	- 0		0.50 * 2 = 1.00	
2	15	- 1		( 0 0 1) <sub>2</sub>	
2	7	- 1			
2	3	- 1			
	1	- 1	MSD		

( 1 1 1 1 1 0 1 )<sub>2</sub>

- 125.125<sub>10</sub> = ( 1 1 1 1 1 0 1 . 0 0 1 )<sub>2</sub>

**Step 2 :** Normalize the number

( 1 1 1 1 1 0 1 . 0 0 1 )<sub>2</sub> = 1.111101001 \* 2<sup>6</sup>

### Step 3 : single precision

$S = 1$  (-ve sign) ,  $E = 6$  ,  $M = 1\ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 1$

Bias for single precision format is = 127

$$E' = E + 127 = 6 + 127 = 133_{10}$$

Convert binary (133) =  $10001001_2$

1	1 0 0 0 1 0 0 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 0 0 1
sign	exponent	Mantissa

### Step 4:

$S = 1$  (-ve sign) ,  $E = 6$  ,  $M = 1\ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 1$

Bias for single precision format is = 1023

$$E' = E + 1023 = 6 + 1023 = 1029_{10}$$

Convert binary (133) =  $10000001001_2$

1	1 0 0 0 0 0 0 1 0 0 1	0 1 1 1 1 0 1 0 0 1
sign	exponent	Mantissa

### Exception:

IEEE standards, the processor sets flags if underflow and overflow

### Overflow:

- A situation in which a positive exponent becomes too large to fit in the exponent field.
- In single precision , if the number requires an exponent greater than +127
- In double precision , if the number requires an exponent greater than +1023

### underflow :

- A situation in which a negative exponent becomes too large to fit in the exponent field.
- In single precision , if the number requires an exponent less than -126
- In double precision , if the number requires an exponent less than -1022

### Floating point Addition:

Consider 2 floating point numbers

$A = m_1. R e_2$

$B = m_2. R e_2$

### Rules for addition and subtraction:

Step 1: select the number with smaller exponent & shift its mantissa right equal to the difference in exponent.

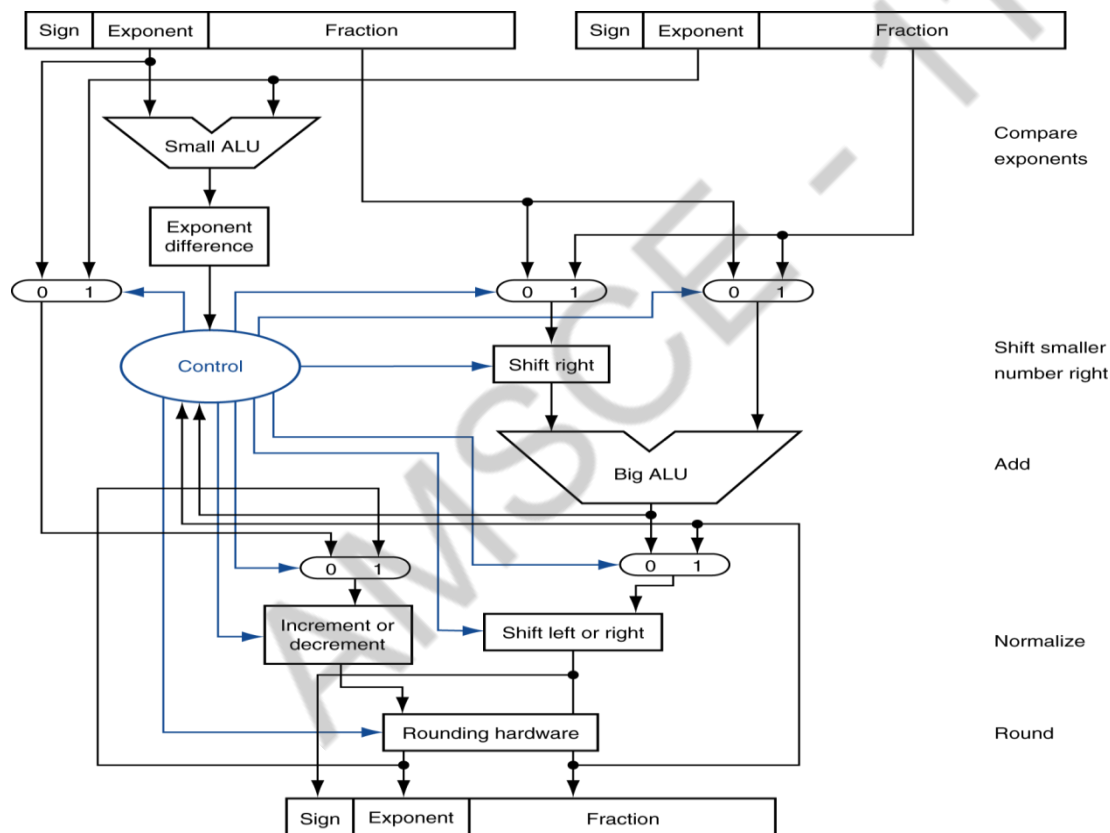
Step 2: set the exponent of the result equal to the larger exponent.

Step3: perform addition or subtraction on the mantissa and determine the sign of the result.

Step 4: Normalize the result if necessary.

Step 5: round the number ( 4 digits long)

### Flow chart:



### FP Adder Hardware:

The hardware implementation for the addition and subtraction of 32 bit floating point operation.

1 bit sign, 8 bits for exponent and 23 bits for mantissa.

- The steps of [flow chart](#) correspond to each block, from top to bottom.



- First, the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much.
- This difference controls the three multiplexors; from left to right, they select the larger exponent, the significand of the smaller number, and the significand of the larger number. The smaller significand is shifted right, and then the significands are added together using the big ALU.
- The normalization step then shifts the sum left or right and increments or decrements the exponent.
- Rounding then creates the final result, which may require normalizing again to produce the actual final result.

**Example:** Add the number  $1.75 \times 10^2$  and  $6.8 \times 10^4$

step 1:

$1.75 \times 10^2$	(select the smaller exponent)
<hr style="width: 100px; margin: 0;"/>	
$0.175 \times 10^3$	(shift the point right and increment the power by 1)
$0.0175 \times 10^4$	(shift the point right and increment the power by 1)

Step 2 : Addition of the significance ( mantissa)

0.0 1 7 5
6.8 0 0 0
<hr style="width: 100px; margin: 0;"/>
6.8 1 7 5
<hr style="width: 100px; margin: 0;"/>

Step 3: Normalize the result

$6.8 1 7 5 \times 10^4$

Step 4: round of the sum

$6.8 1 8 \times 10^4$

### Binary floating point addition

**Example:** Subtract the number  $0.5_{\text{ten}}$  and  $-0.4375_{\text{ten}}$

### Convert decimal to binary first:

$0.5 \times 2 = 1.0$

$0.1 \times 2^0$  (shift the point left and decrement the power by 1)

### **1.0 X 2<sup>-1</sup> // normalization**

$$\begin{array}{rcl} 0.4375 \times 2 & = & 0.8750 \\ 0.875 \times 2 & = & 1.750 \\ 0.75 \times 2 & = & 1.50 \\ 0.5 \times 2 & = & 1.0 \end{array}$$

- 0.0111 x 2<sup>0</sup> (shift the point left and decrement the power by 1)

### **- 1.11 X 2<sup>-2</sup> // normalization**

**step 1:**                      - 1.110 X 2<sup>-2</sup> (select the smaller exponent)

                                    - 0.111 X 2<sup>-1</sup> (shift the point right and increment the power by 1)

### **Step 2 : Addition of the significant ( mantissa)**

**1.000 X 2<sup>-1</sup> // normalization**

- 0.111 X 2<sup>-1</sup> // take 2's complement answer // 1.001)

Add the number:

$$\begin{array}{r} 1.000 \times 2^{-1} \\ 1.001 \times 2^{-1} \\ \hline 1\ 0.001 \times 2^{-1} \text{ [discard the carry]} \end{array}$$

### **Step 3: Normalize the result**

0.001 X 2<sup>-1</sup> (shift the point left and decrement the power by 1)

00.01 X 2<sup>-2</sup> (shift the point left and decrement the power by 1)

000.1 X 2<sup>-3</sup>

**1.00 X 2<sup>-4</sup>**

**Step 4: round of the sum: 1.00 X 2<sup>-4</sup>**

## **6. Floating point multiplication:**

Rules for Multiplication:

Step 1: Adding the exponent without bias and with bias. And subtract new exponents with bias and bias(127)

Step 2 : multiplication of significant.

Step 3: normalize the result

Step 4: round the product.

Step 5: place the sign in the final product

**Flow chart:**

**Example:** multiply the number  $1.110 \times 10^{10}$  and  $9.200 \times 10^{-5}$

step 1: Add two exponents without bias

$$10 + (-5) = 5$$

Add two exponents with bias (127)

$$10 + 127 = 137$$

$$\begin{array}{r} -5 + 127 = 122 \\ \hline \end{array}$$

$$\text{Add} \quad 259$$

Subtract both new exponents with bias and bias(127):

$$259 - 127 = 132$$

Step 2: Multiply the significance ( $1.110 \times 9.200$ )

1110

9200

---

0000

0000x

2220xx

9990xxx

10212000

**Product is  $10.212000 \times 10^5$**  (place the point and add the exponent)

**Step 3: normalize the result:**

$$10.212000 \times 10^5$$

$$1.0212000 \times 10^6$$

**Step 4: round the result:**

$$1.0212 \times 10^6$$

**Step 5: place the sign in the product:**

$$+ 1.0212 \times 10^6$$

**Binary floating point addition**

**Example: multiply the number  $0.5_{\text{ten}}$  and  $-0.4375_{\text{ten}}$**

**Convert decimal to binary first:**

$$0.5 \times 2 = 1.0$$

$$0.1 \times 2^0 \text{ (shift the point left and decrement the power by 1)}$$

**$1.0 \times 2^{-1}$  // normalization**

$$0.4375 \times 2 = \begin{array}{|c|} \hline 0 \\ \hline \end{array} .8750$$

$$0.875 \times 2 = \begin{array}{|c|} \hline 1 \\ \hline \end{array} .750$$

$$0.75 \times 2 = \begin{array}{|c|} \hline 1 \\ \hline \end{array} .50$$

$$0.5 \times 2 = \begin{array}{|c|} \hline 1 \\ \hline \end{array} .0$$

$$- 0.0111 \times 2^0 \text{ (shift the point left and decrement the power by 1)}$$

**-  $1.11 \times 2^{-2}$  // normalization**

step 1: Add two exponents without bias

$$-1 + (-2) = -3$$

Add two exponents with bias (127)

$$-1 + 127 = 126$$

$$-2 + 127 = \underline{125}$$

Add 251

Subtract both new exponents with bias and bias(127):

$$251 - 127 = 124$$

Step 2: Multiply the significance (1.0 \* 1.11)

1.11

10

**1.110**

**Product is  $1.110 \times 10^{-3}$**  (place the point and add the exponent)

**Step 3: normalize the result:**

$$1.110 \times 10^{-3}$$

**Step 4: round the result:**

$$1.110 \times 10^{-3}$$

**Step 5: place the sign in the product:**

$$- 1.110 \times 10^{-3}$$

## **7. Floating point Division:**

Step 1: Subtract the exponent without bias and with bias. And add new exponents with bias and bias (127)

Step 2 : Divide the significant.

Step 3: normalize the result

Step 4: round the product.

Step 5: place the sign in the final product

**Flow chart**

**Example:** multiply the number  $1.110 \times 10^7$  and  $9.200 \times 10^{-5}$

step 1: sub two exponents without bias

$$7 - (-5) = -12$$

sub two exponents with bias (127)

$$7 - 127 = -120$$

$$-5 - 127 = -132$$

$$\begin{array}{r} \text{Sub} \quad 12 \end{array} \quad (-120) - (-132) = (-120 + 132) = 12$$

Add both new exponents with bias and bias(127):

$$12 + 127 = 139$$

Step 2: Divide the significance ( $1.110 / 9.200$ )

After normal division the answer is  $= 0.1206 \times 10^{-12}$

**Product is  $10.212000 \times 10^5$**  (place the point and add the exponent)

**Step 3: normalize the result:**

$$0.1206 \times 10^{-12}$$

$$1.206 \times 10^{-13}$$

**Step 4: round the result:**

$$1.20 \times 10^{-13}$$

**Step 5: place the sign in the product:**

$$+1.20 \times 10^{-13}$$

### **CARRY LOOK AHEAD ADDER (CLA):**

- Carry look ahead adder or fast adder is a type of adder used in digital logic.
- It improves speed by reducing the amount of time required to determine carry bits
- The sum and carry output of any stage cannot be produced until the input carry occurs. This time delay is known as carry propagation delay.

### Design Issues:

In the carry-lookahead circuit we need to generate the two signals carry propagator(P) and carry generator(G),

$$P_i = A_i \oplus B_i$$

$$G_i = A_i \cdot B_i$$

The output sum and carry can be expressed as

$$Sum_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + (P_i \cdot C_i)$$

### Logical Circuit:

#### Carry vector: equation for

$$C_1 = G_0 + P_0 \cdot C_0$$

$$C_2 = G_1 + P_1 \cdot C_1 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0$$

$$C_3 = G_2 + P_2 \cdot C_2 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

$$C_4 = G_3 + P_3 \cdot C_3 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

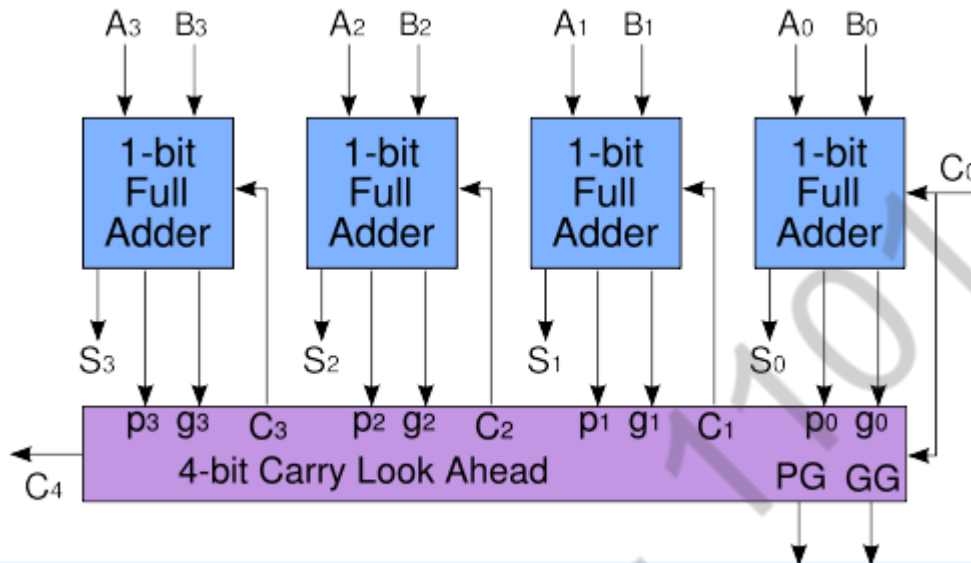
### Design of Carry Look ahead Adders:

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



The number of gate levels for the carry propagation can be found from the circuit of full adder.

The signal from input carry C<sub>in</sub> to output carry C<sub>out</sub> 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<sub>5</sub> would have 2 X 4 = 8 gate levels from C<sub>1</sub> to C<sub>5</sub>. For an n-bit parallel adder, there are 2n gate levels to propagate through.

## 8. SUBWORD PARALLELISM:

Partitioning the carry chains within the ALU can convert the 64 bit adder into 4 (16) bit adders or 8(8) bit adders

A single load can fetch multiple values and a single add instruction can perform multiple parallel additions refers to as **subword parallelism**.

It is also called vector or SIMD (single instruction and multiple Data)

It is also classified under the more general name of data level parallelism



## UNIT -3

### Processor and Control Unit

#### **1. BASIC MIPS IMPLEMENTATION:**

**NOV/DEC 2015, APR/MAY 2015 ( 16 MARKS )**

The basic MIPS implementation includes a subset of the core MIPS instruction set.

Every instructions are divided into 3 instruction classes

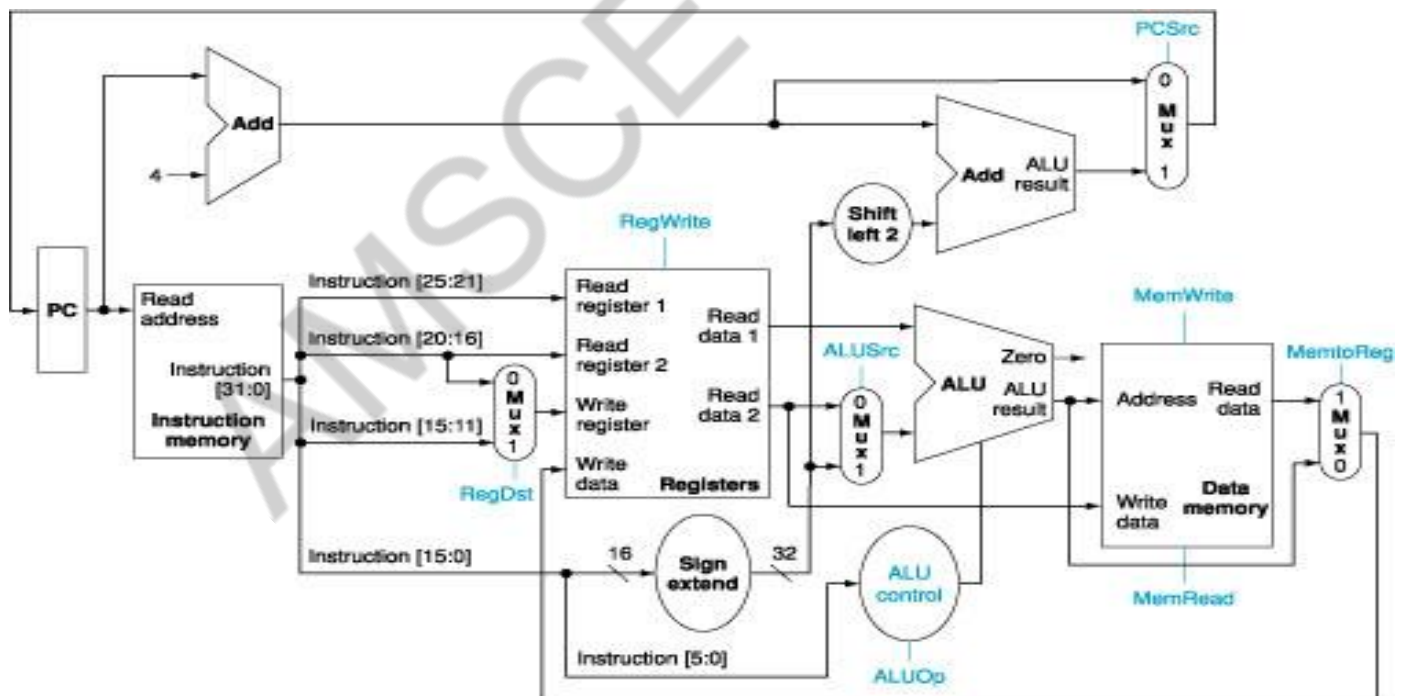
#### **Instruction classes:**

1. Memory Reference Instruction. [ load word and store word]
2. Arithmetic and logical instruction [ Add, Sub, mul , Or, And ect]
3. Branch instruction. [ jump and branch equal]

#### **Overview of the MIPS implementation:**

In every instruction, there are two steps which are identical

1. Fetch instruction: fetch the instruction from the memory
2. Fetch operand: select the registers to read



#### **Operation:**

**The program counter:** It supply instruction address to the instruction memory.

**Instruction memory:** After the instruction is fetched, the register operands required by an instruction are specified by fields of that instruction

**Register operand:** Once the register operands have been fetched, they can be used to compute three classes of instruction.

**1. Memory Reference Instruction:**

- It uses the ALU for an address calculation.
- After using ALU, memory reference instruction to access the memory either to read data for a load or write data for a store.

**2. Arithmetic and logical instruction:**

- It uses ALU for the operation execution.
- After completing the execution the Arithmetic and logical instruction must write the data from ALU or memory back into a register.

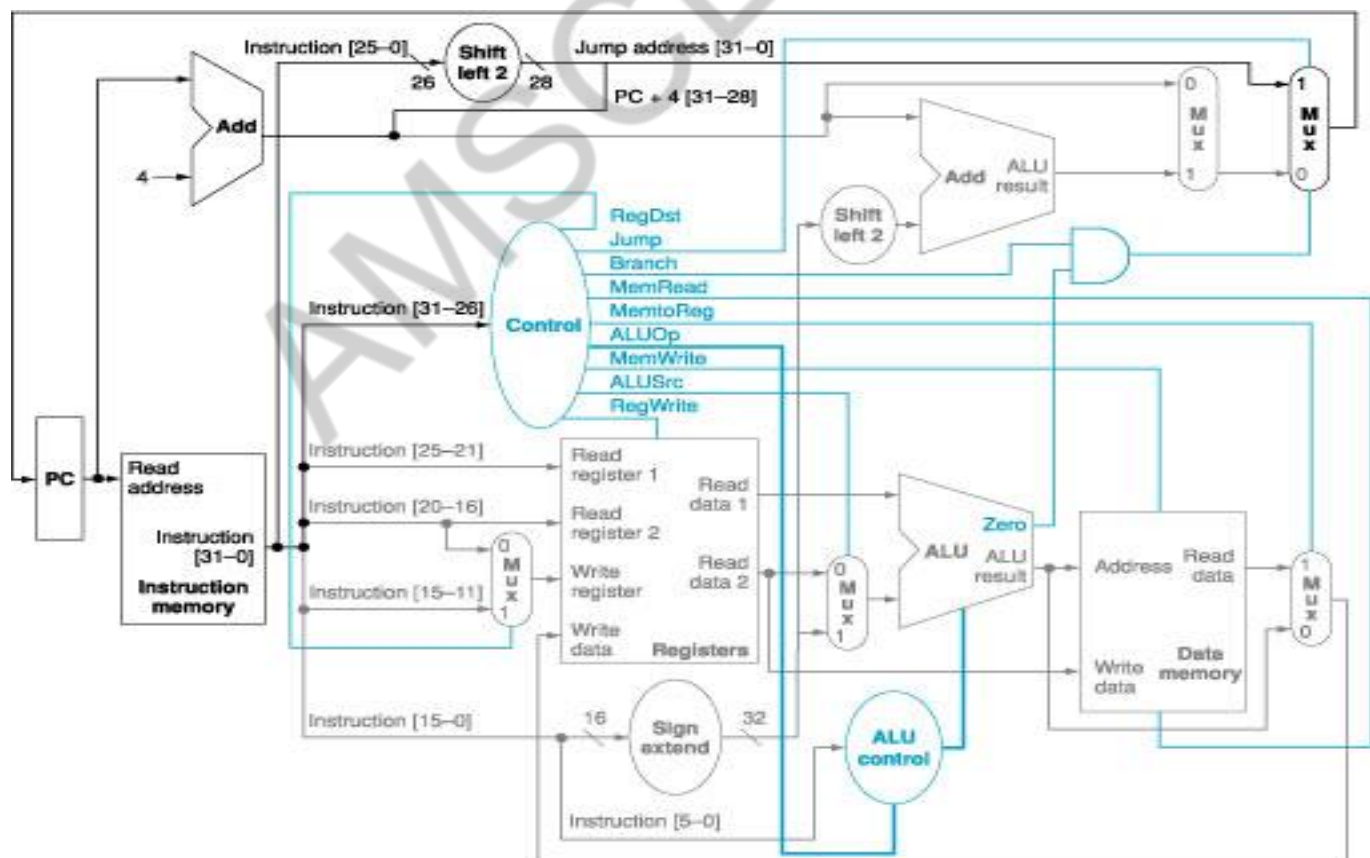
**3. Branch instruction:**

- It uses ALU for comparison.
- After comparison, need to change the next instruction address based on the comparison, otherwise Pc should be incremented by 4 to get the address of next instruction.

**Multiplexor:** The data going to a particular unit comes from two different sources. These data lines cannot be wired together, we must add a device that combine the multiple sources and sent to the destination. Such device called multiplexor (many inputs single output)

**Adder:** Increment the PC to the address of the next instruction.

**Basic implementation of MIPS with Control signals:**



The multiplexor selects from several inputs' based on the setting of its control lines.

The control lines are set Based on information taken from the instruction being execute.

### **Control unit:**

Control unit which has the instruction as an input is used to determine the control signals for the function unit and two of the multiplexors.

- The input to the control fields is the 6 bit opcode field from the instruction.
- The output of the control unit consist of three 1-bit signals are used to control multiplexors.
- 3 signals for controlling reads and writes in the register file and data memory.
- 1-bit control signal used in determining for branch
- 2-bit control signal for ALU.

### **Logic Design Conventions:**

In MIPS implementation consists of two different types of logic elements (**data path element**).

1. Combinational Element
2. State Element

### **Combinational Element:**

- The element that operates on data value such as AND gate or an ALU, which means the output depend only on the current inputs.

### **State Element:**

- A memory element such as register or a memory is called as state element.
- An element contains state if it has internal storage.
- Logical component that contains state are called sequential, because their output depend on both their inputs and the contents of the internal state.

### **Clocking Methodology:**

- It is used to determine when the data is valid and stable relative to the clock.
- It specifies the timing of read and writes.
- A clocking methodology is designed to ensure predictability.

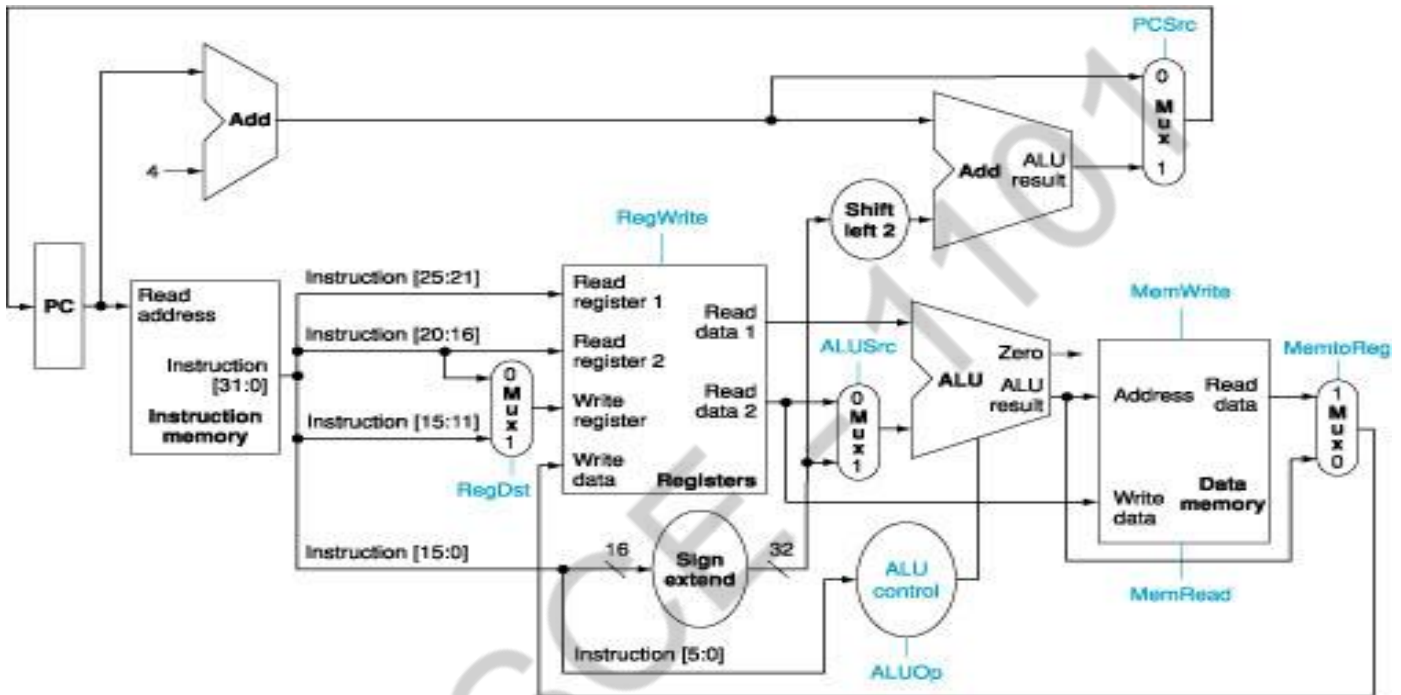
### **Edge – triggered clocking Methodology:**

- Any values stored in a sequential logic element are updated only on a clock edge.

## 2. BUILDING DATA PATH:

A data path element is used to operate on or hold data within a processor.

In MIPS implementation, the data path elements include the instruction and data memories, the register files, the ALU and adders.



**Instruction Memory:** A memory unit to store the instruction of a program and supply instructions gives an address.

**Program counter:** PC is register containing the address of the next instruction in the program.

**Adder:** Increment the Pc to the address of the next instruction.

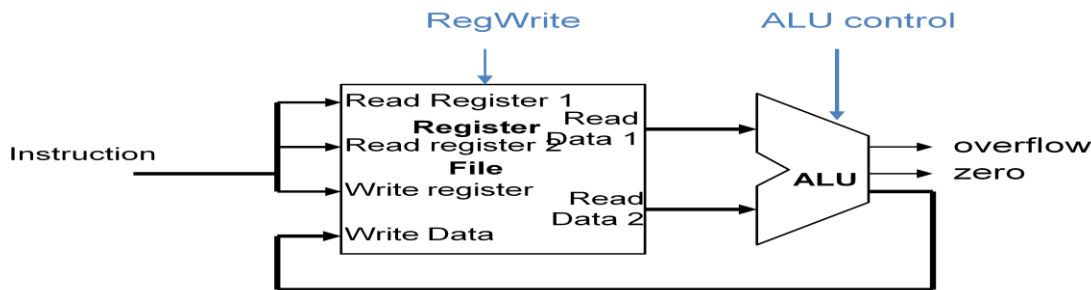
### Data segments:

There are three data Segments:

1. Data segment for Arithmetic and logical instruction.
2. Data segment for load word and store word instruction.
3. Data segment for branch instruction.

### 1.Data segment for Arithmetic and logic instruction:

- Arithmetic and logical instruction read operands from 2 registers, perform an Arithmetic and logical operation and write the result to the register.
- These instruction are also called R-format instructions.



### R- format instruction:

- R- format instruction have three register operands. 2 source operand and 1 destination operand.
- It include add, sub, AND, OR and slt.
- Example: OR \$t1, \$t2, \$t3

### The register files:

- In MIPS processor stores 32 general purpose register this structure called register file
- It is a collection of register.
- It contains the register state of computer.

### Read registers:

- To read data words, we need to specify the register number to the register file.

### Write Register: To write data words, we need two inputs.

1. To specify the register number to be written.
2. To supply the data to be written into the register.

### Register write:

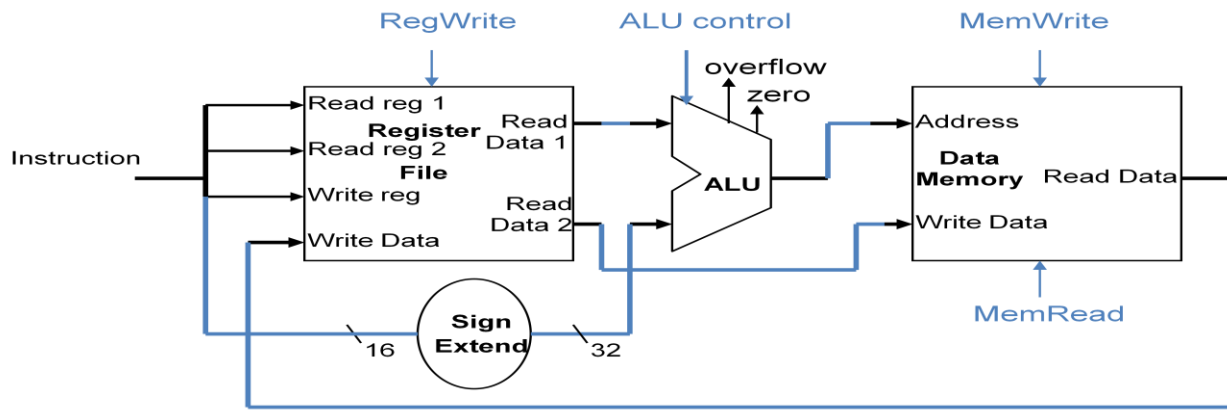
- It is a write control signal.
- It control write operation.
- If the signal is edge triggered, to perform write operation.
- If the signal is earlier clock cycle, to perform read operation.

### 2.Data Segment for load word and store word instruction:

The general form of load word and store word instruction in MIPS processor are

**Lw \$ t1, offset value(\$t2)**

**Sw \$t1, offset value (\$t2)**



These instructions compute memory address by adding the base register.

$$\text{Memory address} = \text{base register} + \text{offset value.}$$

**store value** : read from the Register File, written to the Data Memory

**load value** : read from the Data Memory, written to the Register File

**Sign extend** :

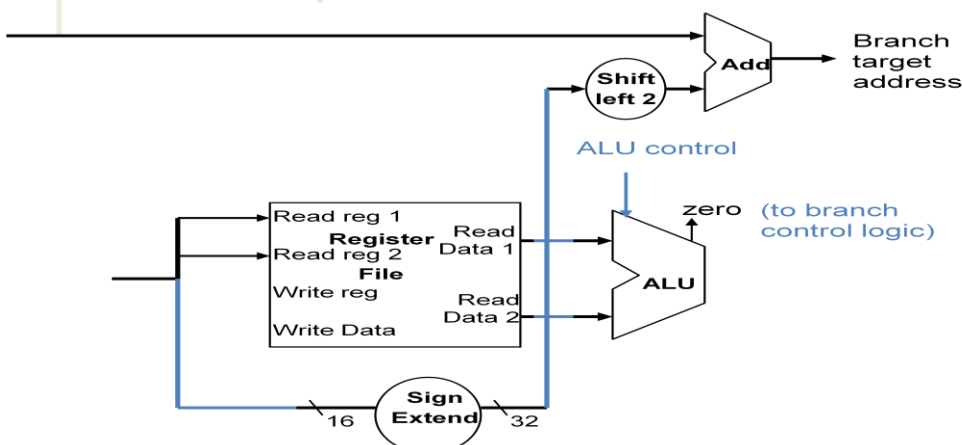
- To convert 16 bits offset field in the instruction to a 32 bit signed value.
- It is used to increase the size of the data item by replacing the high order sign bit of the original data item in the higher order bit of the larger destination data item.

**Data memory unit:**

- It has read and write control signal, an address input and an input for the data to be written into memory

### 3.Data Segment for branch instruction:

- The branch instruction has three operands, 2 register & 1 offset.
- 2 register are compared for equality (**zero** ALU output).
- 16 bit offset used to compute branch target address.
- Branch target address is an address specified in a branch which becomes the new program counter if the branch is taken.



**Example: beq \$t1, \$t2, offset**

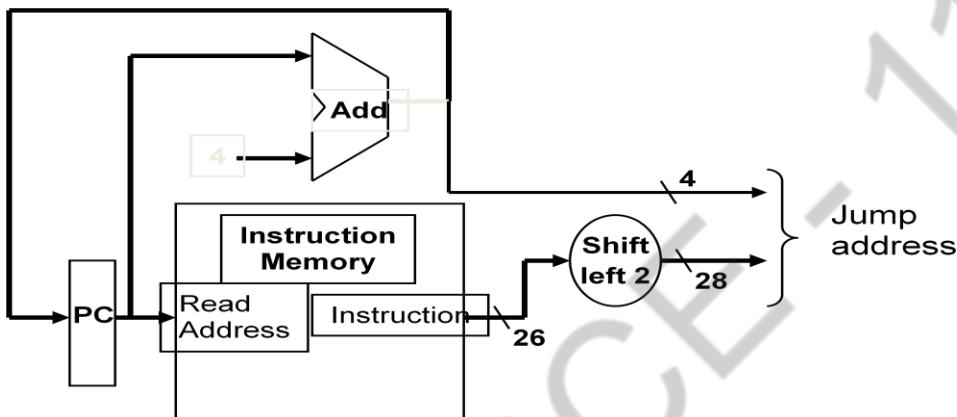
- If the condition is true ,the branch target address becomes new pc and the branch is taken.
- If the condition is false, incremented pc should replace the current pc and branch is not taken

**The branch data path must perform two operations:**

1. Compute the branch target address: the branch data path includes a sign extension unit, shifter and an adder
2. Compare the register content : used register file and the ALU.

**Jump operation involves:**

Replace the lower 28 bits of the PC with the lower 26 bits of the fetched instruction shifted left by 2 bits.

**Creating a single data path:**

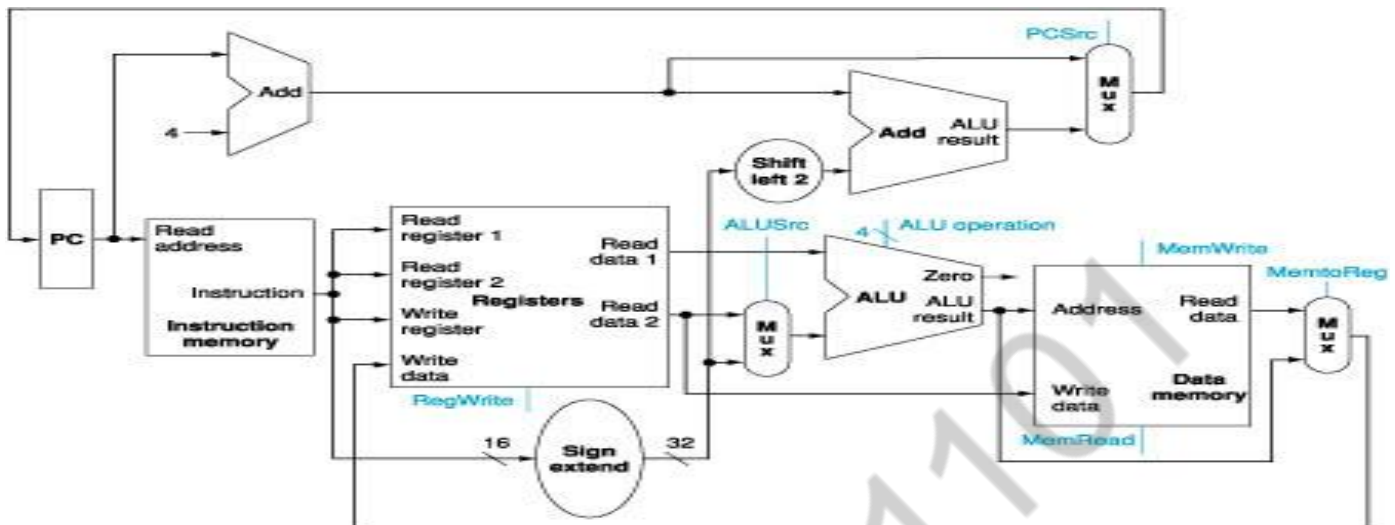
- The data path component for individual instruction classes, combine them into a single data path and add the control to complete the implementation.
- The simplest data path will attempt to execute all instruction in one clock cycle (fetch, decode and execute each instructions in one clock cycle)
- No data path resource can be used more than once per instruction.
- Multiplexors needed at the input of shared elements with control lines to do the selection
- Write signals to control writing to the Register File and Data Memory
- Cycle time is determined by length of the longest path.

**Problem:**

How to build a data path for the operational portion of the memory reference and arithmetic logical instructions that use a single register file and a single ALU to handle both types of instructions assign any necessary multiplexors.

**Answer:** to create a data path with only a single register file and a single ALU, we use two multiplexors. One is placed at the ALU input and another at the data input to the register file.

**Show how to build a datapath for arithmetic-logical ,memory reference and branch instructions.**



We can combine all the pieces to make a simple data path for the MIPS architecture by adding the datapath for instruction fetch, the data path from R- format and memory instruction and the data path for branches.

### **3. CONTROL IMPLEMENTATION SCHEME:**

A control implementation scheme by adding simple control functions to the existing data path.

**Different control implementation scheme:**

1. The ALU control
2. Designing the main control unit
3. Operation of the datapath

**The ALU control:**

There are 6 possible combinations of 4 control inputs. ALU will need to perform one of these function



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

**Load word and store word instruction:** use the ALU to compute the memory address by adding.

**R-type instruction:** uses ALU to perform one of the five actions.

**Branch instruction:** use ALU must perform a subtraction

**ALUOp control bits and different function codes:**

4 bit control input using a small control inputs the function field of the instruction and 2 bit control field.

Instruction opcode	ALUOp	Instruction operation	Func field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

**FIGURE 5.12** How the ALU control bits are set depends on the ALUOp control bits and the different function codes for the R-type instruction. The opcode, listed in the first column,

**Truth table:** It is representation of a logical operation by listing all the values of the inputs.

ALUOp		Func field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

**FIGURE 5.13** The truth table for the three ALU control bits (called Operation). The inputs

Truth table shows how the 4 bit ALU control is set depending on these 2 input fields.



Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

### Opcode fields of the instruction:

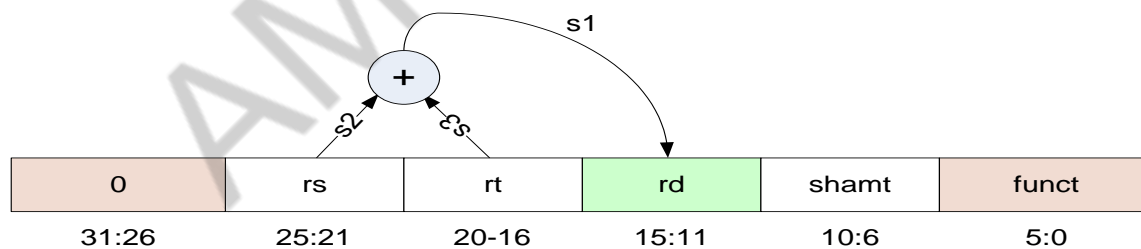
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

**FIGURE 5.18** The setting of the control lines is completely determined by the opcode fields of the instruction. The first row of

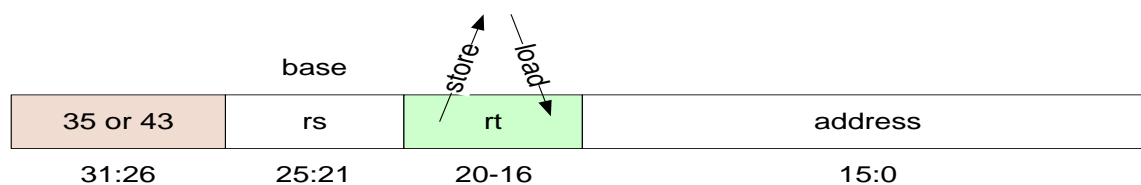
### Operation of the datapath:

3 instruction classes which help to understand how to connect the fields of an instruction to the data path.

1. Instruction format for R- format instruction, which all have an opcode of 0.
2. Instruction format for load (opcode = 35 ten) and store =43 ten) instruction.
3. Instruction format for branch equal (opcode =4)



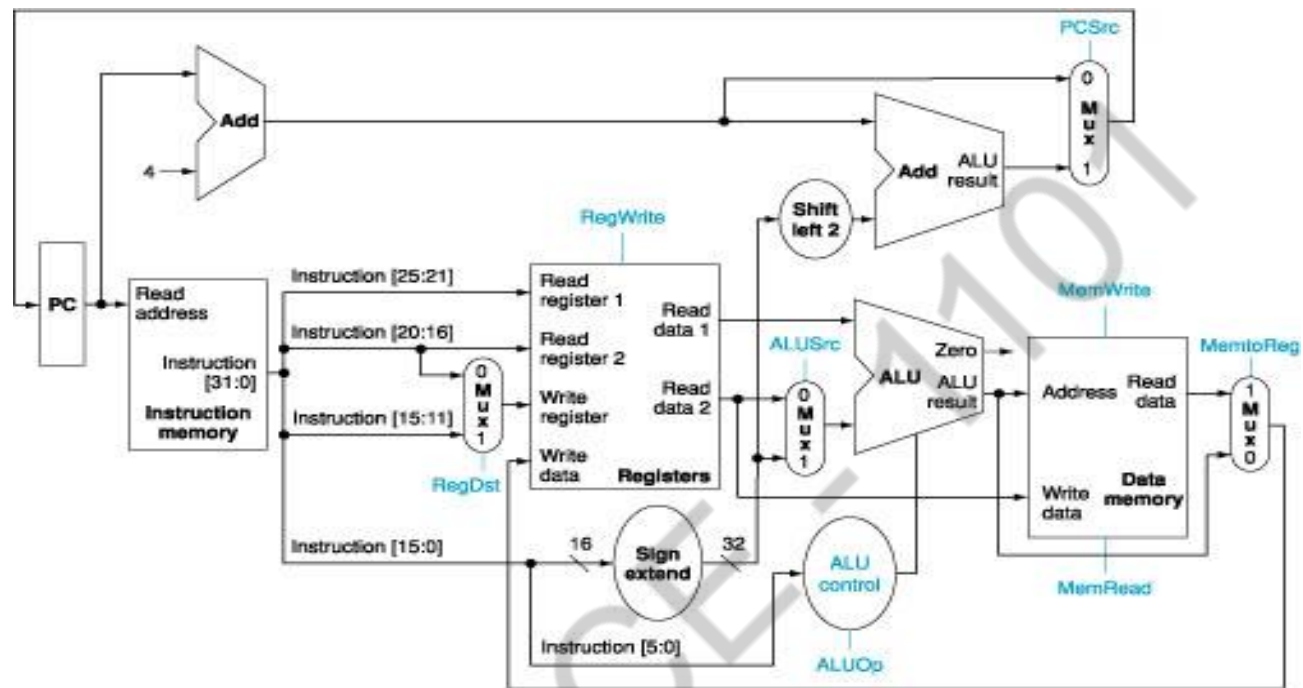
add \$s1, \$s2, \$s3



lw \$s1, 100(\$s2)

sw \$s1, 100(\$s2)

The destination register is in one or two place.



This figure shows the additions plus the ALU control block, the write signal for state elements, the read signal for the data memory and the control signals for the multiplexor.

1. The instruction is fetched and the Pc is incremented.
2. To registers, \$t2 and \$t3 are read from the register file and main control unit computes the setting of the control lines during this step.
3. The ALU operates on the data read from the register file, using the function code to generate the ALU function.
4. The result from the ALU is written into the register file using bits 15:11 of the instruction to select the destination register ( \$ t1)

## 2.Data path for an operation in a Load instruction:

Data path for an Load word instruction

LW \$t2, offsetvalu (\$t1)

Five steps needed to execute the instruction:

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. A register (\$t2) value is read form the register file.
3. The ALU computes the sum of the value read from the register file and the sign extended lower 16 bits of the instruction(offset).
4. The sum from the ALU is used as the address for the data memory.
5. The data from the memory units is written into the register files; the Register destination is given by bits 20:16 of the instruction (\$t1).

### 3.Data path for an operation in a branch –on –equal instruction:

Data path for beq instruction

Beq \$t1,\$t2,offset

Four steps needed to execute the instruction:

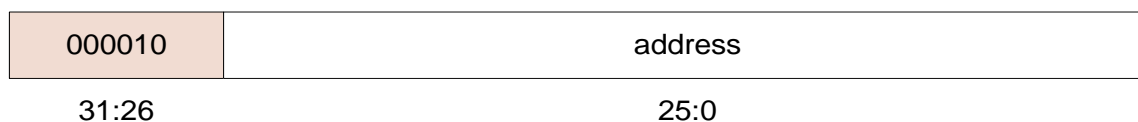
1. An instruction is fetched from the instruction memory and the Pc is incremented.
2. Two register \$t1,\$t2 are read from the register file.
3. The ALU performs a subtract on the data value read from the register file. The value PC+4 is added to the sign extended , lower 16 bits of the instruction shifted left by two, the result is the branch target address.
4. The Zero result from the ALU is used to decide which adder result to store into the PC.

### Implementing Jumps:

- The low-order 2 bits of a jump address are always 00<sub>two</sub>.
- The next lower 26 bits of this 32 bit address come from the 26 bit immediate field in the instruction.
- The upper 4 bits of the address replace the Pc come from Pc of the jump plus 4.

### Jump by storing into the PC the concatenation of:

1. The upper 4 bit of the current PC+4
2. The 26 bit immediate field of the jump instruction.
3. The bit 00<sub>two</sub>.





**4. PIPELINING: \_\_\_\_\_ MAY/JUNE 2016, NOV/DEC 2014 ( 16 MARKS )**  
**WHAT IS PIPELINING AND DISCUSS ABOUT PIPELINED DATA PATH AND CONTROL**

Pipelining is an implementation technique in which multiple instructions are overlapped in execution.

**Five steps in a MIPS instruction:**

1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register

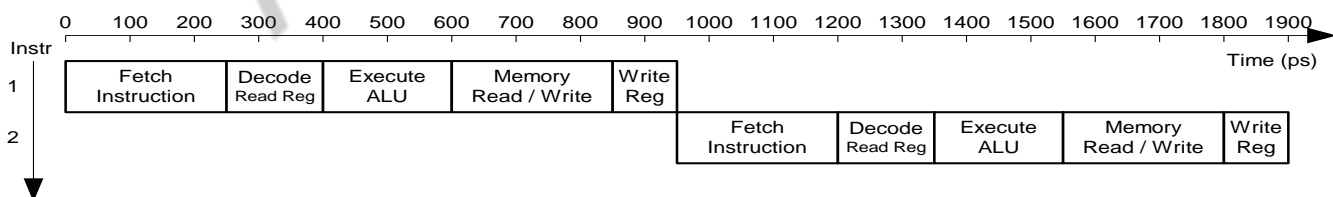
Example:

Assume time for stages is 100ps for register read or writev 200ps for other stages. Compare pipelined datapath with single-cyclev datapath

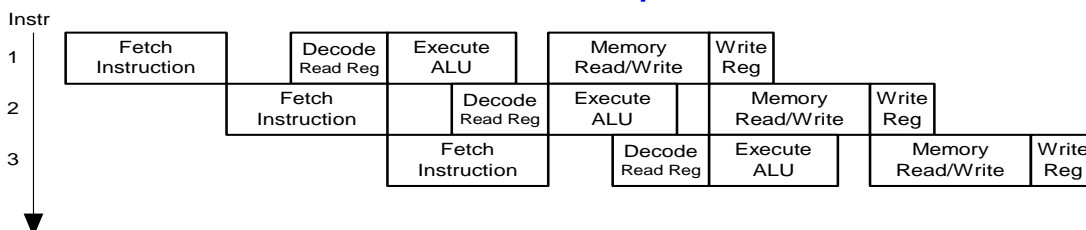
**Solution:**

Instruction class	Instruction fetch	Register read	ALU operation	Data Access	Register write	Total Time
LW	200ps	100ps	200ps	200ps	100ps	800ps
SW	200ps	100ps	200ps	200ps		700ps
R-format	200ps	100ps	200ps		100ps	600ps
Branch	200ps	100ps	200ps			500ps

**Single-Cycle**



**Pipelined**



### **Pipeline Speedup:**

If all stages are balanced i.e., all take the same time.

$$\text{Time between instructions pipeline} = \frac{\text{Time between instructions non pipeline}}{\text{Number of Pipe Stages}}$$

If not balanced, speedup is less. Speedup due to increased throughput Latency (time for each instruction) does not decrease

### **PIPELINED DATAPATH AND CONTROL:**

The datapath and control unit share similarities with both the single-cycle and multicycle implementations that we already saw.

An example execution highlights important pipelining concepts.

#### **4.1 PIPELINING DATAPATH:**

The single cycle data path with the pipeline stages identified. The division of an instruction into 5 stages means a five stage pipeline and the five stage are as follows:

IF: Instruction fetch

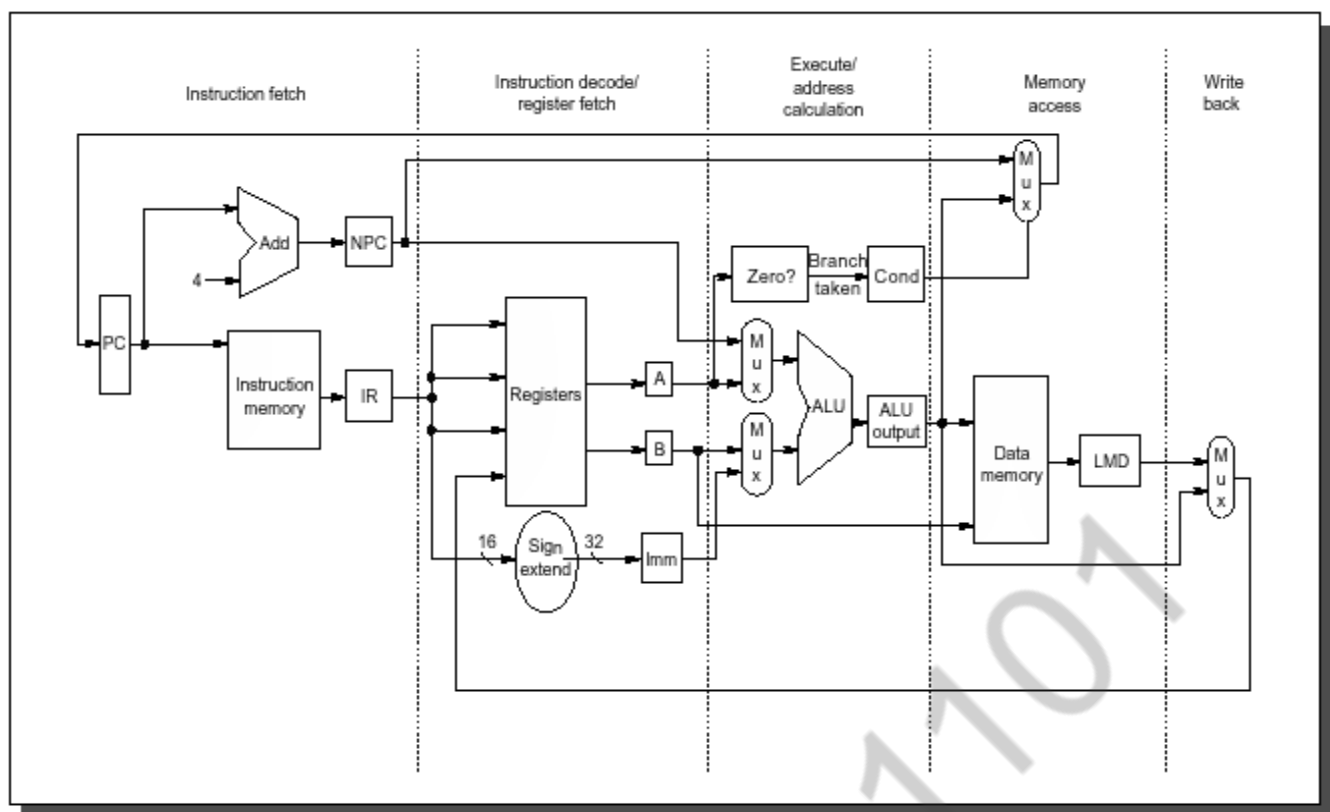
ID: Instruction decode and register file read.

Execution or address calculation

MEM: Data memory access

WB: Write back

**Five components correspond roughly to the way the data path:**

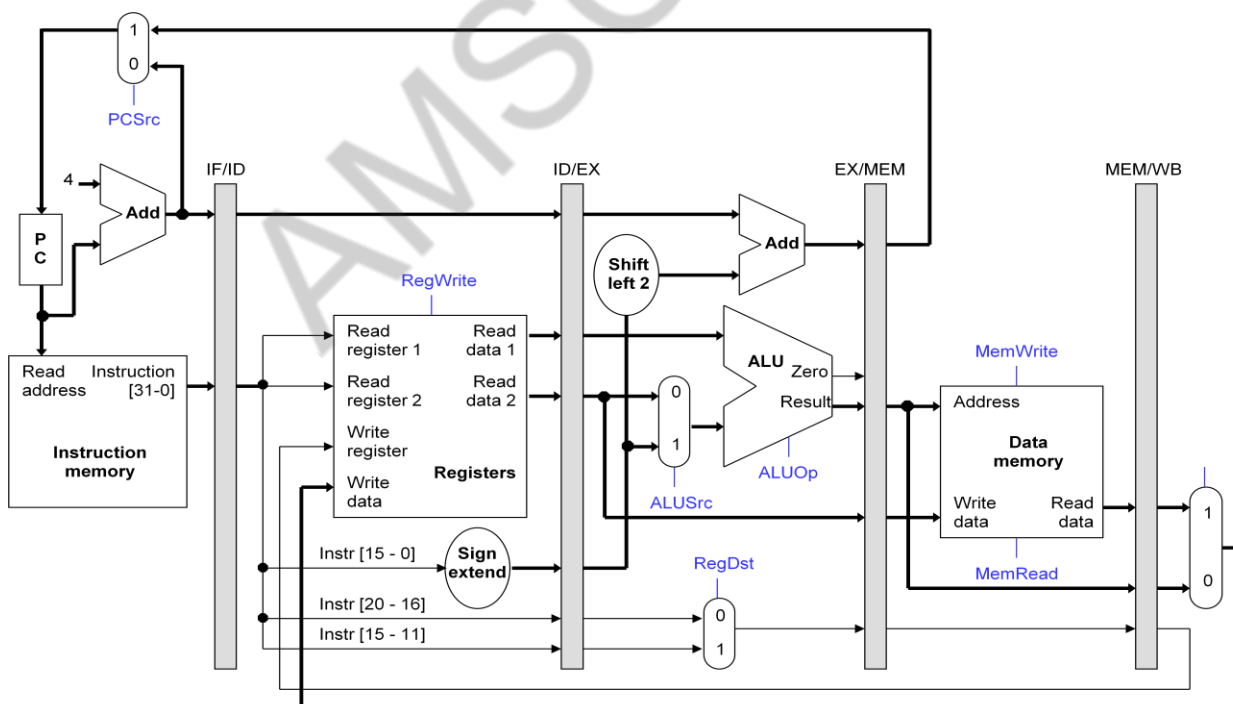


### Five stages as they complete execution. Returning to

The instructions and data move generally from left to right through the five stages .

Two exceptions to this left to right flow of instructions:

### Pipeline version of datapath:



### Pipelined Datapath highlighting the pipeline registers:



The registers are named for separating the 2 stages.

The register between IF & ID stages to separate instruction fetch and decode.

The register between ID & EXE to separate decode and ALU execution.

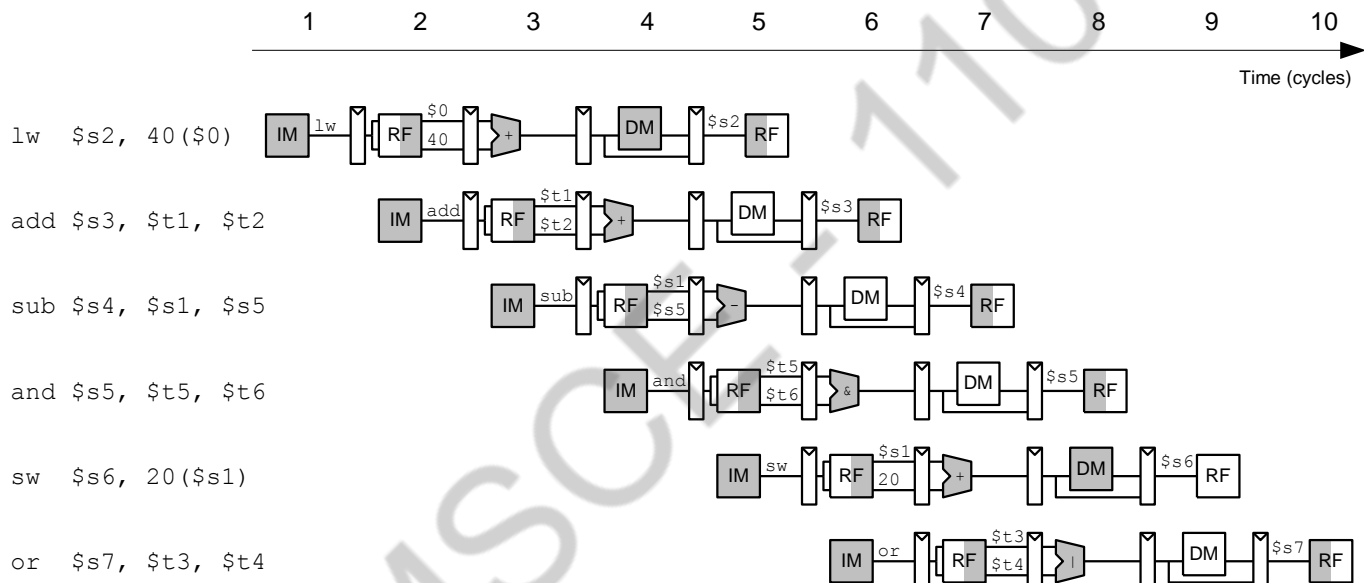
The register between EXE and MEM to separate ALU execution and data Memory

The register between MEM and WB to separate data memory and write data to register

**The register file is split into 2 local parts:**

1. Register read during register fetch(ID)
2. Register written during write back(WB)

**Instruction execution in a pipeline manner**



IM represent the instruction memory and the PC in the instruction fetch stage(IF)

Reg stand for the register file in the instruction decode/register file read stage

**Execution of Load /store instruction in a five stage pipeline:**

**Stage 1 : Instruction fetch:**

- The instruction being read from instruction memory using the address in PC and then storing the instruction in the IF/ID pipeline register.
- PC address is incremented by 4 and then write back into PC to read for next clock cycle.
- Incremented address is also saved in the IF/ID pipeline register.

**Stage 2 :Instruction decode & register file read:**

- The instruction portion of the IF/ID pipeline register.

- It supply the 16 bit immediate field, which is sign extended to 32 bit and the register numbers to read the 2 registers.
- All three values are stored in the ID/EX pipeline register.

### Stage 3: Execute and address calculation:

- The load instruction reads the contents of register 1 and the sign extended immediate from ID/EX pipeline register and add them using ALU.
- The sum is placed in EX/MEM pipeline register.

### Stage 4: Memory access:

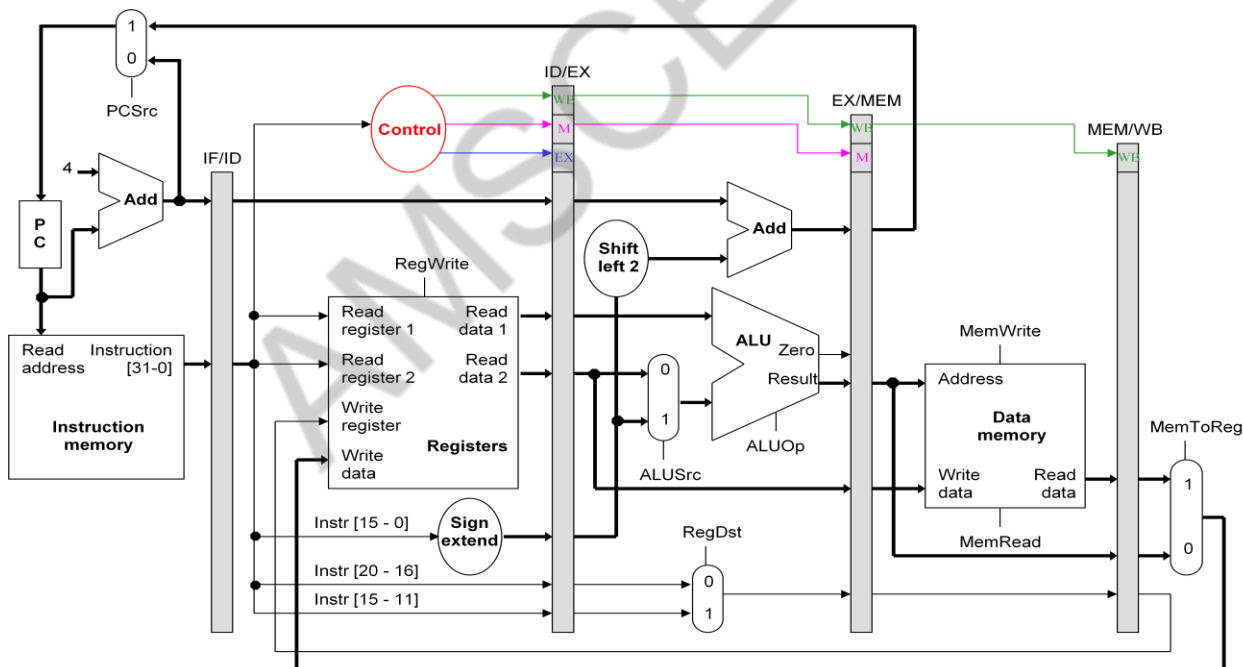
- The load instruction reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline registers.

### Stage 5: Write Back:

- Reading the data from the MEM/WB pipeline register and writing into the register file.

## 4.2 PIPELINED CONTROL:

- In pipeline control just add the control to the pipelined data path.
- Thus data path borrows the control logic for source, register destination number and ALU control.



- The control signals are generated in the same way as in the single-cycle processor—after an instruction is fetched, the processor decodes it and produces the appropriate control values.
- But just like before, some of the control signals will not be needed until some later stage and clock cycle.

- These signals must be propagated through the pipeline until they reach the appropriate stage. We can just pass them in the pipeline registers, along with the other data.

**Control signals can be categorized by the pipeline stage that uses them:**

Stage	Control signals needed		
<b>EX</b>	<b>ALUSrc</b>	<b>ALUOp</b>	<b>RegDst</b>
<b>MEM</b>	<b>MemRead</b>	<b>MemWrite</b>	<b>PCSrc</b>
<b>WB</b>	<b>RegWrite</b>	<b>MemToReg</b>	

### **Stage 1: Instruction fetch**

- The control signals to read instruction memory and to write PC are always asserted.

### **Stage 2 : Instruction decode/register file read:**

- As in the previous stage the same thing happens at every clock cycle, so there are no optional control line to set.

### **Stage 3: Execution/address calculation:**

- The signal to be set are RegDst,ALUOp and ALUSrc.
- The signals select the Result register, The ALU operation and either Read data 2 or a sign extended immediate for the ALU.

### **Write and draw ALU tabular column from (control implementation scheme\_)**

### **Stage 4: Memory access:**

- The control lines set in this stage are Branch, MEMRead and MEMwrite.
- These signals are set by branch equal,load and store instructions.
- PCSrc selects the next sequential address unless control asserts Branch and the ALU result was 0.

### **Stage 5: Write Back:**

- The two control lines are MemtoReg, which decides between sending the ALU result or the memory value to the register file and Reg-write, which writes the chosen value.

## **5. PIPELINE HAZARDS:**

**MAY/JUNE 2016, APR/MAY 2015, NOV/DEC 2014 (16 MARKS)**

A pipeline hazard refers to situations that prevent an instruction from entering the next stage is called hazard.

### **3 different types of hazard:**

1. Structural hazards
2. Data hazards
3. Control hazards

**Structural hazards** - Attempt to use the same resource by two or more instructions at the same time

**Control hazards** - attempt to make branching decisions before branch condition is evaluated

**Data hazards** – When wither the source or destination operands of an instruction are not available at time expected in the pipeline and as a result pipeline is stalled. This situation is a data hazard.

### **Let's look at a sequence with many dependences,:**

```
sub $2, $1, $3    # Register $2 written by sub
and $12, $2, $5   # 1st operand($2) depends on sub
or $13, $6, $2    # 2nd operand($2) depends on sub
add $14, $2, $2   # 1st($2) & 2nd($2) depend on sub
sw $15, 100($2)  # Base ($2) depends on sub
```

The last four instructions are all dependent on the result in register \$2 of the first instruction. If register \$2 had the value 10 before the subtract instruction and -20 afterwards, the programmer intends that -20 will be used in the following instructions that refer to register \$2.

### **5.1 HANDLING DATA HAZARD:**

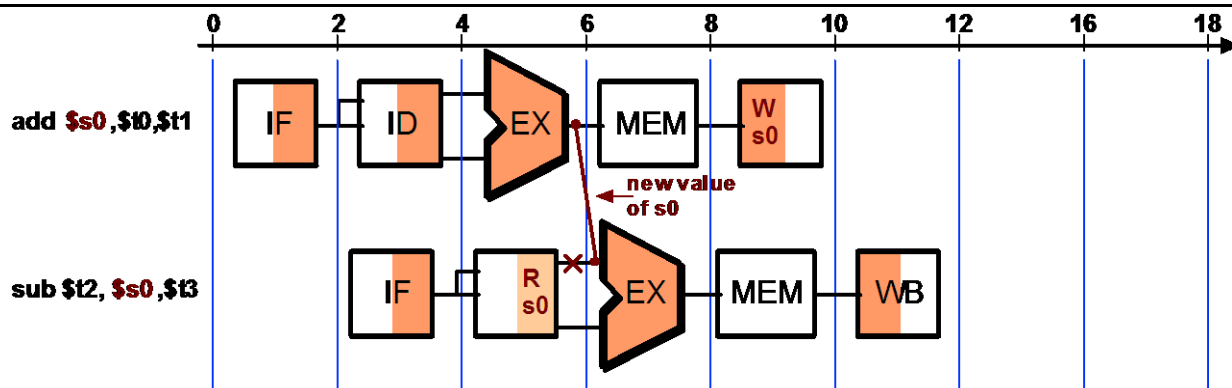
#### **Forwarding versus Stalling**

Data hazards occur when the pipeline changes the order of read/write accesses to operands that differs from the normal sequential order

#### **Forwarding (bypassing) with two instructions:**

- It is also known as bypassing.
- This is a method resolving a data hazard by retrieving the missing data element from the internal buffer rather than waiting for it to arrive from the memory.

#### **Graphical Representation of Forwarding:**



It show the connection to forward the value in \$s0 after the execution stage of the add instruction.

Forwarding paths are valid only if the destination stage is later in time than source stage.

Forwarding cannot prevent all pipeline stalls. They are also often referred to as *bubbles* in the pipeline.

### Forwarding versus stalling:

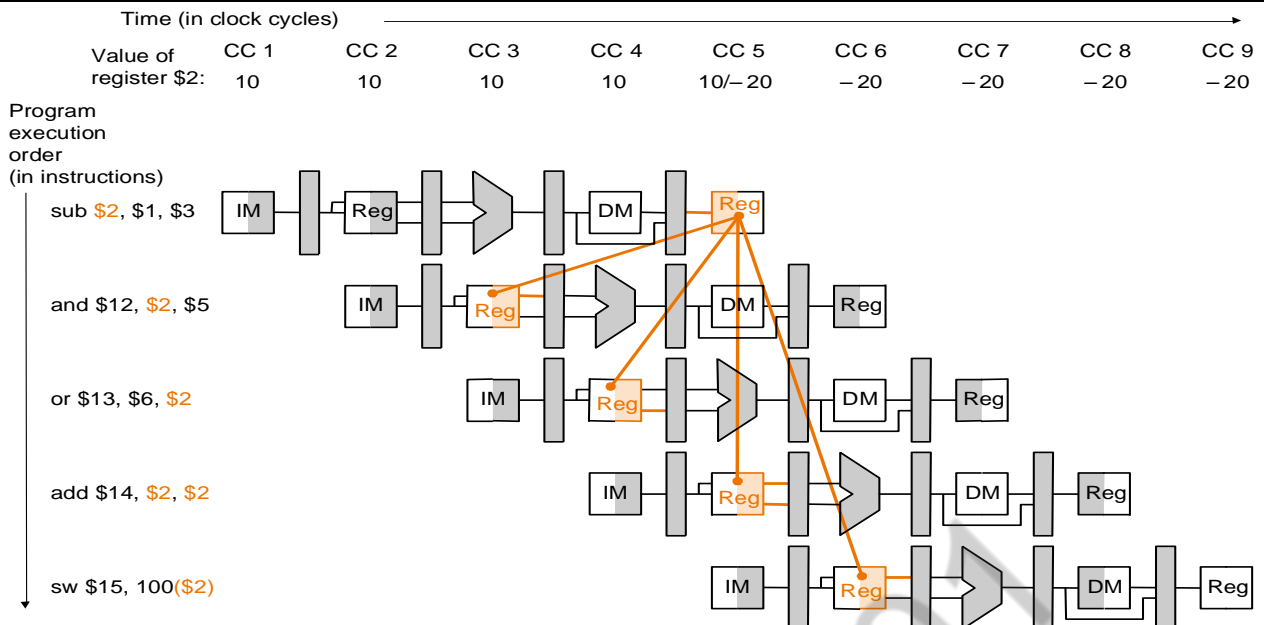
Data hazard with a sequence of many dependencies

Sub	\$2,	\$1,	\$3	#	register	\$2	written	by	sub
and	\$12,	\$2,	\$5	#	1 <sup>st</sup>	operand	(\$2)	depends	on
or	\$13,	\$6,	\$2	#	2 <sup>nd</sup>	operand	(\$2)	depends	on
add	\$14,	\$2,	\$2	#	1 <sup>st</sup>	(\$2)	&	2 <sup>nd</sup>	(\$2)
sw	\$15,	100(\$2)		#	base	(\$2)	depends	on	sub

The last four instructions are all dependent on the result in register \$2 of the first instruction.

If the register \$2 had the value 10 before the subtraction instruction and -20 afterwards, the programmer intends -20 will be used in the following instruction that refer to register \$2.

### Pipelined Dependences in a five instruction sequence:



This diagram illustrates the execution of these instructions using a multiple clock cycle pipeline representation.

### Hazard condition: HAZARD DETECTION UNIT

The two pairs of hazard conditions are

**1a: EX/MEM.RegisterRd = ID/EX.RegisterRs**

**1b: EX/MEM.RegisterRd = ID/EX.RegisterRt**

**2a: MEM/WB.RegisterRd = ID/EX.RegisterRs**

**2b: MEM/WB.RegisterRd = ID/EX.RegisterRt**

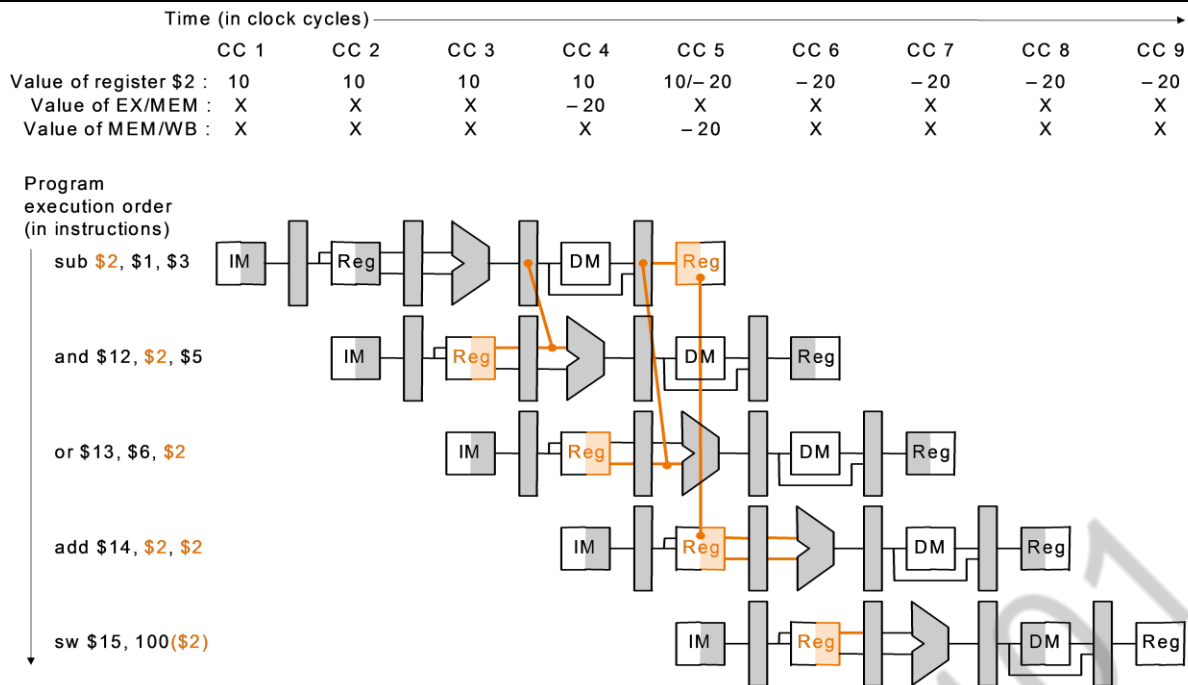
The first hazard in the sequence is one register \$2, between the result of sub \$2 \$1 \$3 and the first read operand of and \$12,\$2,\$5. This hazard can be detected when and instruction is the EX stage and the prior instruction is in the MEM stage

EX/MEM.Register RD= ID?EX.RegisreRs=\$2.

The sub-or is a type 2b hazard:

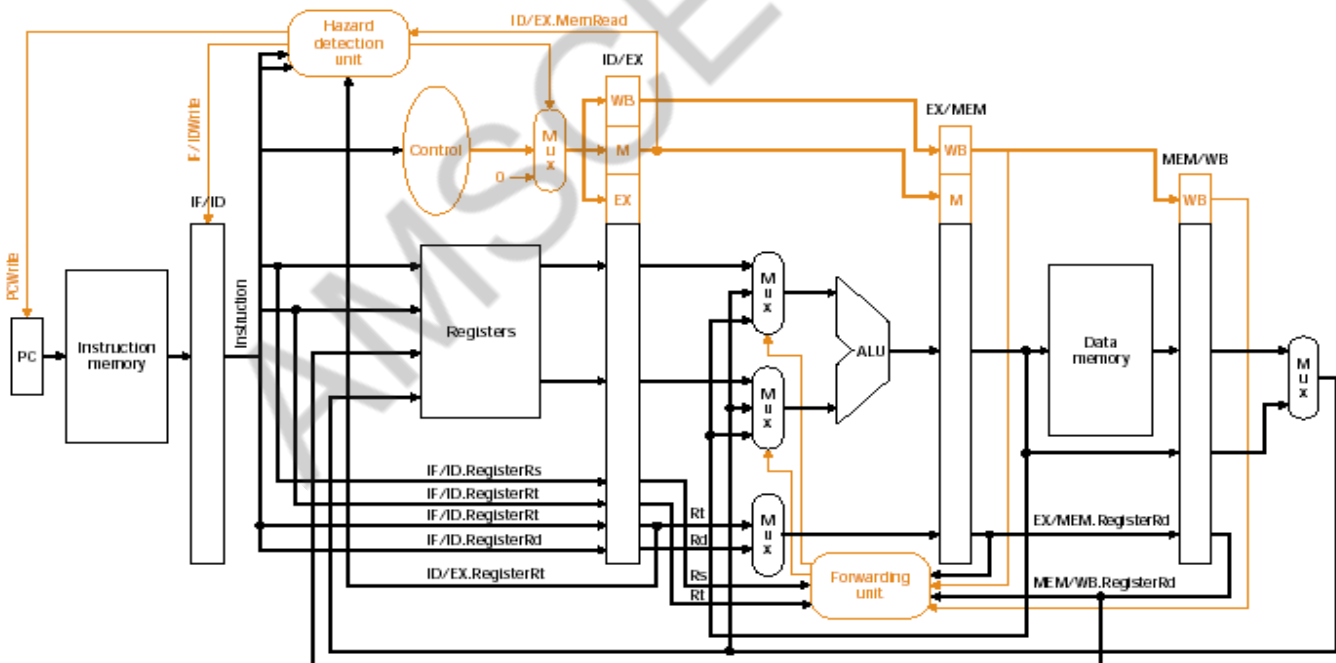
MEM/WB.RegisterRd = ID/EX.RegisterRt = \$2

- The two dependences on sub-add are not hazards because the register file supplies the proper data during the ID stage of add.
- There is no data hazard between sub and sw because sw reads \$2 the clock cycle *after* sub writes \$2.



Can forward only to the “or” and “add” instructions without stalling \$2 still unavailable in EX/MEM for “and”. When sub was the “writing” instruction, we forwarded from EX/MEM to the ALU for “and”

**The multiplexors have been expanded to add the forwarding paths, and we show the forwarding unit.**



The hardware necessary to support forwarding for operations that use results during the EX stage. Note that the EX/MEM.RegisterRd field is the register destination for either an ALU instruction (which comes from the Rd field of the instruction) or a load (which comes from the Rt field).

Forwarding can also help with hazards when store instructions are dependent on other instructions. Since they use just one data value during the MEM stage, forwarding is easy.

### The control values for the forwarding multiplexors:

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result
ForwardB = 00	ID/EX	ID/EX The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result

Let's now write both the conditions for detecting hazards and the control signals to resolve them: Example:

#### 1.Ex Hazard:

if (EX/MEM.RegWrite)

and (EX/MEM.RegisterRd  $\neq$  0)

and(EX/MEM.RegisterRd=ID/EX.RegisterRs))      ForwardA = 10

if (EX/MEM.RegWrite)

and (EX/MEM.RegisterRd  $\neq$  0)

and(EX/MEM.RegisterRd=ID/EX.RegisterRt))      ForwardB = 10

#### MEM Hazard:

if (MEM/WB.RegWrite)

and (MEM/WB.RegisterRd  $\neq$  0)

and(MEM/WB.RegisterRd=ID/EX.RegisterRs))      ForwardA = 01

if (MEM/WB.RegWrite)

and (MEM/WB.RegisterRd  $\neq$  0)

and(MEM/WB.RegisterRd=ID/EX.RegisterRt))      ForwardB = 01

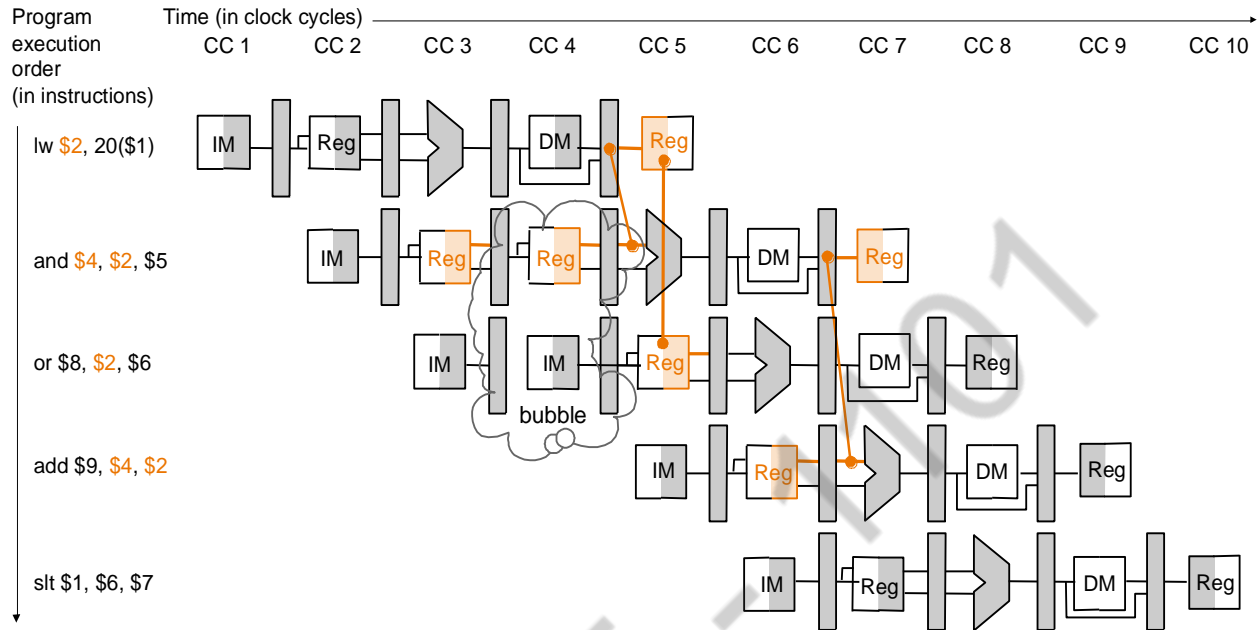
#### Data hazards and stalls:

- So far, we've only addressed "potential" data hazards, where the forwarding unit was able to detect and resolve them without affecting the performance of the pipeline.



- There are also “unavoidable” data hazards, which the forwarding unit cannot resolve, and whose resolution does affect pipeline performance.
- We thus add a (unavoidable) hazard detection unit, which detects them and introduces stalls to resolve them.

**Forwarding technique is used to minimize data hazard:**



In the load instruction the data is read from memory in clock cycle 4. While the ALU performs the operation for the following instruction, sometimes the stall the pipeline for the combination of load.

## 5.2 HANDLING CONTROL HAZARDS:

### Control hazard:

- It is also known as branch hazard.
- When the proper instruction cannot execute in the proper pipeline clock cycle, it is known as control branch.
- A *control hazard* is when we need to find the destination of a branch, and can't fetch any new instructions until we know that destination.

### Reducing the delay of branch:

One way to improve branch performance is to reduce the cost of the taken branch.

**There are two complicating factors:** A branch is either

1. **Taken:** If a branch is changing the PC to its target address, then it is a *taken* branch.

$$PC \leq PC + 4 + \text{Immediate}$$

2. **Not Taken:** If a branch doesn't change the PC to its target address, than it is a *not taken* branch.

$$PC \leq PC + 4$$

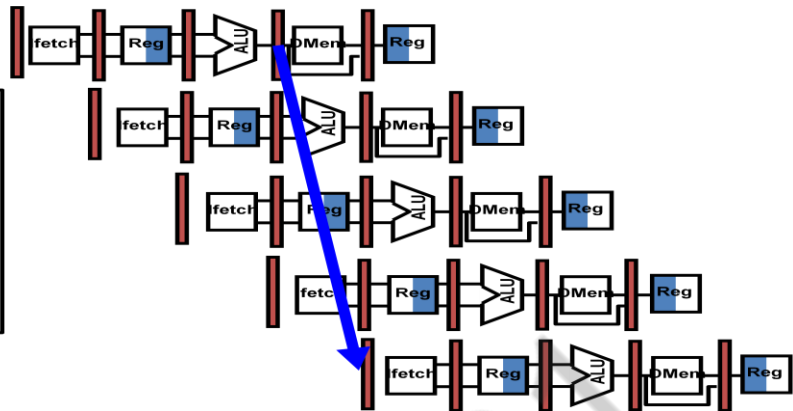
10: **beq r1,r3,36**

14: and r2,r3,r5

18: or r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11

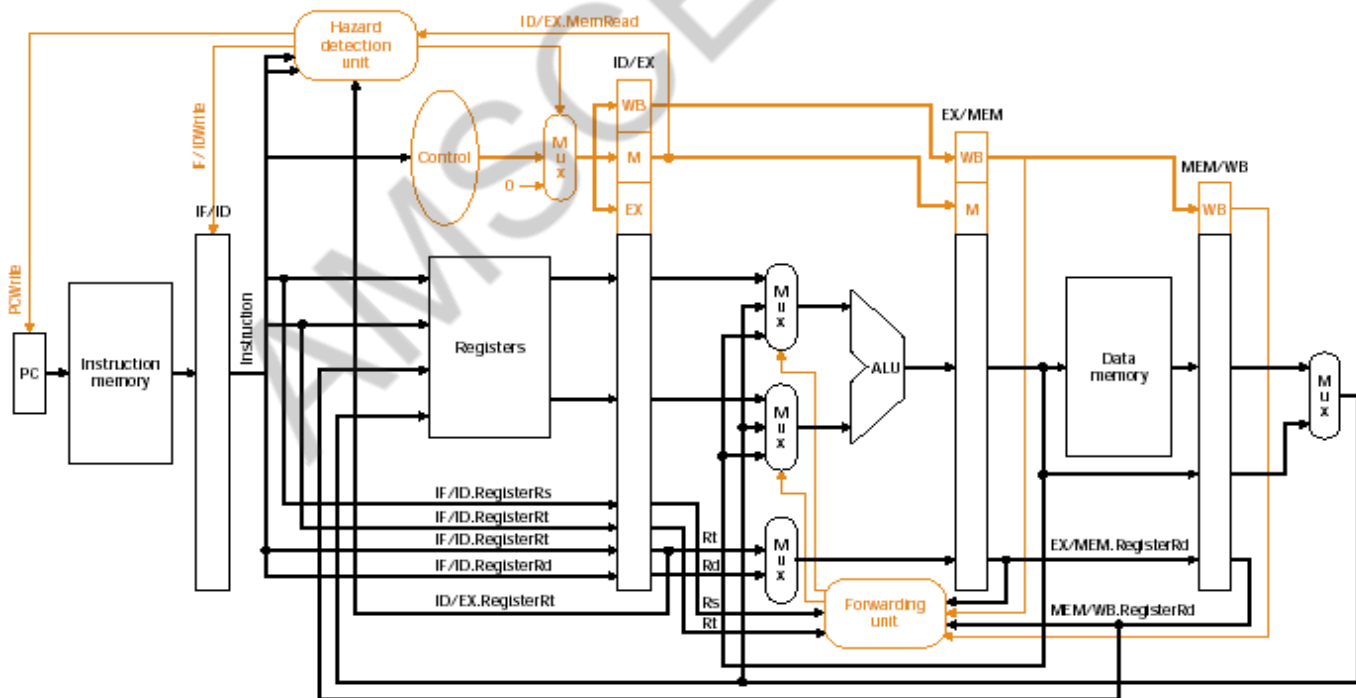


The branch instruction decided where to branch in MEM stage the clock cycle 4 for the beq instruction.

3 sequential instructions that follow the branch will be fetch and being execution.

3 following instruction begin execution beq branch to location 36.

There is delay in the proper instruction to fetch



**flush** To discard instructions in a pipeline, usually due to an unexpected event.

## **Handling control branch:**

Control hazard can be handle using branch prediction.

**Prediction means:** A statement of what will happen in the future

## **Branch predictor:**

- Branch prediction technique is used to handle branches.
- A branch predictor is a digital circuit that tries to guess which way a branch (e.g. an if-then-else structure) will go before this is known for sure.
- If the prediction is correct, avoid delay of the branch hazard
- If the prediction is incorrect, flush the wrong-path instructions from the pipeline & fetch the right path
- The purpose of the branch predictor is to improve the flow in the instruction pipeline

## **Two types of branch prediction:**

- The behavior of branch can be predicted both statically at compile time and dynamically by the hardware at execution time.
  1. Static branch prediction
  2. Dynamic branch prediction

## **Static branch prediction:**

Predicting a branch at compile time helps for scheduling data hazards.

## **Dynamic branch prediction:**

- the prediction determined at runtime is known as dynamic branch prediction
- The simplest dynamic branch-prediction scheme is a branch-prediction buffer or branch history table.

## **Branch-prediction buffer :**

- A branch-prediction buffer is a small memory (*cache*) indexed by the lower portion of the address of the branch instruction.
- The memory contains a bit that says whether the branch was recently taken or not.

## **Two branch prediction scheme:**

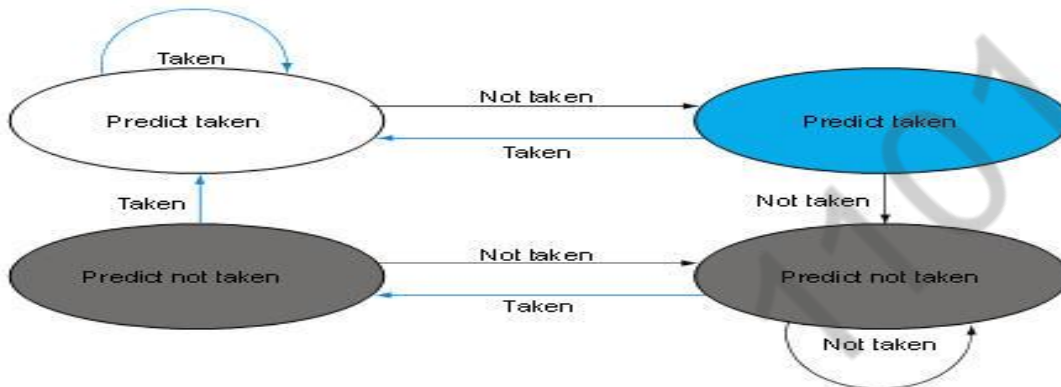
1. **one-bit prediction scheme**
2. **Two- bits prediction scheme**

## **1-bit prediction scheme:**

- If a branch is almost take, we can predict incorrectly twice otherwise it is not taken

### 2-bit prediction scheme:

- In a two bits prediction scheme must miss twice before it is changed
- In a two bits prediction scheme are used to encode the four states in the system.
- *One bit that predicts the direction of the current branch if the previous branch was not taken (PNT).*
- *One bit that predicts the direction of the current branch if the previous branch was taken (PT).*



- It is general instance of a counter-based predictor.
- Counter-based predictor is incremented when the prediction is accurate and decremented otherwise.
- The counters saturate at 00 or 11
- It has an n-bit saturating counter for each entry in the prediction buffer.
- With an n-bit counter, the counter can take on values between 0 and  $2^n - 1$ :
- When the counter is greater than or equal to one half of its maximum value ( $2^{n-1}$ ), the branch is predicted as taken; otherwise, it is predicted untaken.

### branch delay slot:

- The slot directly after a delayed branch instruction, which in the MIPS architecture is filled by an instruction that does not affect the branch.
- Compilers and assemblers try to place an instruction that always executes after the branch in the **branch delay slot**.
- The job of the software is to make the successor instructions valid and useful.

### Three ways in which the branch delay slot can be scheduled:

The top box in each pair shows the code before scheduling; the bottom box shows the scheduled code.

In (a), the delay slot is scheduled with an independent instruction from before the branch. This is the best choice. Strategies (b) and (c) are used when (a) is not possible.

In the code sequences for (b) and (c), the use of \$s1 in the branch condition prevents the add instruction (whose destination is \$s1) from being moved into the branch delay slot.

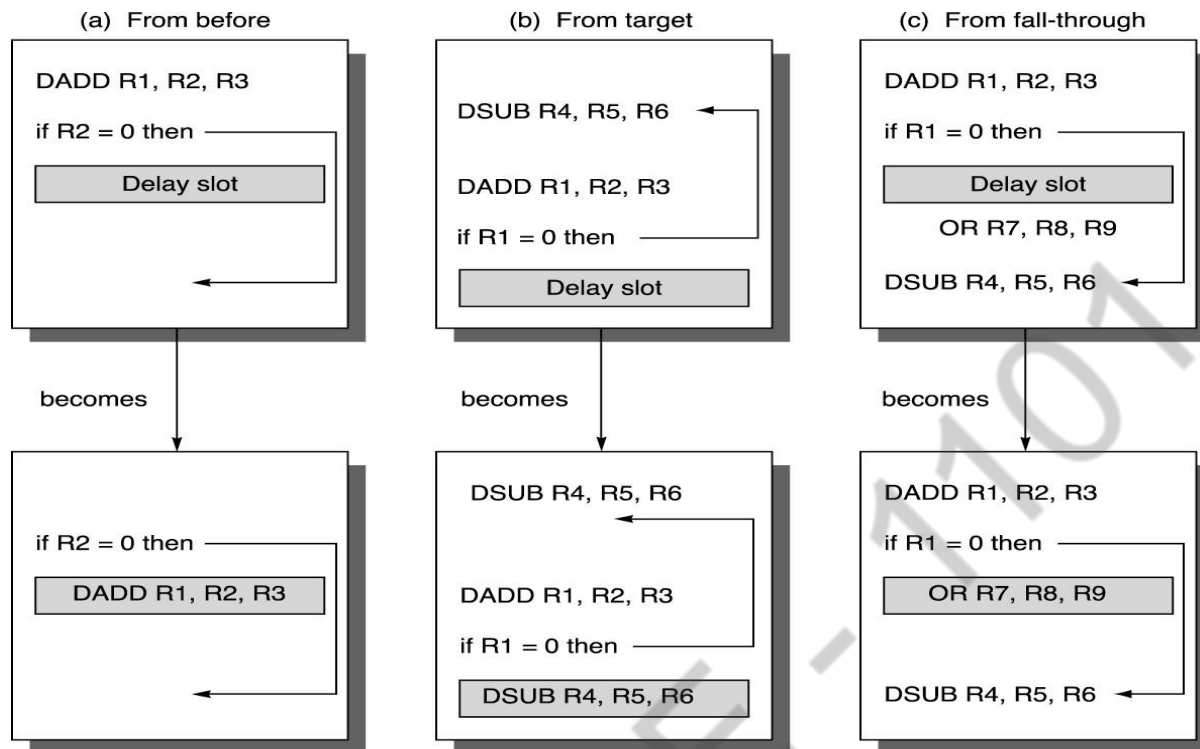
In (b) the branch delay slot is scheduled from the target of the branch; usually the target instruction will need to be copied because it can be reached by another path.

Strategy (b) is preferred when the branch is taken with high probability, such as a loop branch.

Finally, the branch may be scheduled from the not-taken fall-through as in (c).

To make this optimization legal for (b) or (c), it must be OK to execute the sub instruction when the branch goes in the unexpected direction.

By “OK” we mean that the work is wasted, but the program will still execute correctly.



© 2003 Elsevier Science (USA). All rights reserved.

## 6. EXCEPTIONS:

- One of the difficult parts of control is to implement exceptions and interrupts.
- Exceptions are generally generated unscheduled events that disrupt program execution and they have used to detect overflow.
- exception Also called interrupt.
- Interrupt comes from outside of the processor.

**An interrupts is an exception that comes from outside of the processor:**

Type of event	From where	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user	Internal	Exception
Arithmetic overflow	Internal	Exception

Hardware malfunction	Either	Exception interrupts	or
-------------------------	--------	-------------------------	----

## **Handling Exception in the MIPS architecture:**

### **Types of exception:**

1. Execution of an undefined instruction.
2. Arithmetic overflow in the instruction add \$1, \$2, \$1.

### **Responses to an exception:**

When an exception occurs the processor save the address of the offending instruction in the Exception Program Counter ( EPC) and transfer control to the OS at some specified address.

The OS take the appropriate action, which involve providing some service to the user program, taking some predefined action in response to an exception or stopping the execution of the program and reporting an error.

Methods used to communicate the reason for an exception:

To handling the exception it is must for the OS to know the reason for the exception.

### **Two main methods used to communicate the reason for an exception:**

1. **Status register method**
2. **Vectored interrupts method**

**Status register method:** the MIPS architecture uses a status register which holds a field that indicates the reason for the exception.

**Vectored interrupts method:** In a vectored interrupt the address to which control is transferred is determined by the cause of the exception.

### **Exception in a pipelined implementation:**

A pipelined implementation treats exceptions as another form of control hazard.

### **In pipeline computers interrupt and exceptions are further classified as:**

1. Imprecise interrupts or imprecise exceptions
2. Precise interrupts or precise exceptions

### **Imprecise interrupts or imprecise exceptions:**

In pipelined computers that are not associated with the exact instruction that war the cause of the interrupt or exception is called imprecise interrupts or imprecise exceptions

### **Precise interrupts or precise exceptions:**

An interrupt or exception that is always associated with the correct instruction in pipeline computer called precise interrupts or precise exceptions.

AMSCE-1101

## UNIT -4

### PARALLELISM

Parallel computing is a form of computation in which many calculations are carried out simultaneously.

Operation on the principle that are large problems can often be divided into smaller ones which are then solved concurrently.

#### **Different forms of parallel computing:**

**Bit level parallelism**

**Data parallelism**

**Instruction level parallelism**

**Task parallelism**

#### **1. INSTRUCTION LEVEL PARALLELISM:**

The potential overlap among instruction is called Instruction level parallelism(ILP).

This technique which is used to overlap the execution of instructions to improve performance.

Pipeline exploits the instruction level parallelism.

#### **Two primary method for increasing the potential amount of ILP:**

1. By increasing the depth of the pipeline to overlap more instructions.
2. By replicating the internal components of the computer so that it can launch multiple instructions in every pipeline stage.

Such multiple instructions are launched in one clock cycle is known as **multiple issue**.

#### **Implementation a multiple-issue processor:**

#### **Two ways to implementation a multiple issue processor:**

**1. Static multiple issue:** here the decisions of division of work are being made statically (at compile time)



**2. Dynamic multiple issue:** here the decisions of division of work are being made dynamically (at run time)

**There are two primary and distinct responsibilities that must be dealt with in a multiple-issue pipeline:**

1. Packaging instructions into issue slots the positions from which instructions could issue in a given clock cycle.
2. Dealing with data and control hazard.

**SPECULATION:**

is an approach that allows the compiler or processor to guess the properties of an instruction. So as to enable execution of other instructions that may depend on the speculation instruction.

**Example:** we might speculate the result of branch, so that the instruction after the branch could be executed earlier.

Need for speculation:

1. It is most important method for finding and exploiting more ILP.
2. Speculation may be wrong , It is necessary include a method to check if the guess was right or wrong and a method to undo the effects of the instruction that were executed speculatively this implementation of this undo capability add complexity.

**Speculation may be done in the compiler or by the hardware.**

1. Compiler can use speculation to reorder instruction.
2. The processor hardware can reorder instructions at runtime.

**Recovery mechanism:**

**In case of compiler speculation:** additional instructions are inserted to check the accuracy of the speculation and provide a recovery routine to use when the speculation is incorrect.

**In case of hardware speculation:** the processor usually use buffers to temporary store the speculative results.

**(i) If speculation is correct, the instructions as executed by allowing the contents of the buffers to be written to the registers or memory.**

**(ii) If speculation is incorrect the special hardware is used to flush the buffers and execute the correct instruction sequence.**

**Static multiple issue:**

Static multiple issue processors all use the compiler to assist with packaging instruction and handling hazards.

A static multiple issue processor usually launches many operations that are defined to be independent in a single wide instruction, typically with many separate opcode fields. Such a style of instruction set architecture is known as **very long instruction word (VLIW)**.

### **Loop Level Parallelism:**

The simplest and most common way to increase the amount of parallelism available among instructions is to exploit parallelism among iterations in a loop called LLP

#### **Example:**

**Addition of two 1000-elements arrays that is completely parallel**

```
For(i=0;i<=1000;i=i+1)
```

```
  x[i]=x[i]+y[i];
```

**Every iteration of loop can overlap with any other iteration.**

#### **Example 2:**

```
for (i=1; i<=100; i= i+1)
```

```
{
```

```
  a[i] = a[i] + b[i];      //s1
```

```
  b[i+1] = c[i] + d[i];    //s2
```

```
}
```

Statement s1 uses the value assigned in the previous iteration by statement s2, so there is a loop-carried dependency between s1 and s2.

Despite this dependency, this loop can be made parallel because the dependency is not circular:

- neither statement depends on itself;
- while s1 depends on s2, s2 does not depend on s1.

### **Dynamic multiple issue:**

Dynamic multiple-issue processors are also known as **superscalar** processors, or simply superscalar's.

#### **Superscalar processor:**

An advanced pipelining technique that enables the processor to execute more than one instruction per clock cycle by selecting them during execution.

#### **A simplest superscalar processor:**

Instructions issue in order, and the processor decides whether zero, one, or more instructions can issue in a given clock cycle.

### **Dynamic pipeline scheduling:**

- Many superscalar's extend the basic framework of dynamic issue decisions to include dynamic pipeline scheduling.
- Dynamic pipeline scheduling chooses which instructions to execute in a given clock cycle while trying to avoid hazards and stalls.

### **Let's start with a simple example of avoiding a data hazard.**

```
lw $t0, 20($s2)
addu $t1, $t0, $t2
sub $s4, $s4, $t3
slti $t5, $s4, 20
```

Even though the sub instruction is ready to execute, it must wait for the lw and addu to complete first, which might take many clock cycles if memory is slow.

Dynamic **pipeline** scheduling allows such hazards to be avoided either fully or partially.

### **The three primary units of a dynamically scheduled pipeline.**

- 1. Instruction fetch and decode unit**
- 2. Functional units**
- 3. Commit unit**

#### **Instruction fetches and decode unit**

The first unit fetches instructions, decodes them, and sends each instruction to a corresponding functional unit for execution.

#### **Functional units**

Each functional unit has buffers, called reservation stations.

**Reservation stations:** which hold the operands and the operation.

As soon as the buffer contains all its operands and the functional unit is ready to execute, the result is calculated.

When the result is completed, it is sent to any reservation stations waiting for this particular result as well as to the commit unit, which buffers the result until it is safe to put the result into the register file or, for a store, into memory.

#### **Commit unit:**

The unit in a dynamic or out-of-order execution pipeline that decides when it is safe to release the result of an operation to programmer visible registers and memory.

The buffer in the commit unit, often called the reorder buffer.

**Reorder buffer:** is also used to supply operands, in much the same way as forwarding logic does in a statically scheduled pipeline

#### **out-of-order execution:**

The processor then executes the instructions in some order that preserves the data flow order of the program. This style of execution is called an **out-of-order execution**, since the instructions can be executed in a different order than they were fetched.

### **In-order commit:**

A commit in which the results of pipelined execution are written to the programmer visible state in the same order that instructions are fetched.

To make programs behave as if they were running on a simple in-order pipeline, the instruction fetch and decode unit is required to issue instructions in order, which allows dependences to be tracked, and the commit unit is required to write results to registers and memory in program fetch order.

### **Various types of Dependences in ILP:**

- 1.Data dependences (also called true data dependences)
- 2.Name dependences
- 3.Control dependences

#### **1.Data dependences**

Instr<sub>J</sub> is data dependent on Instr<sub>I</sub>

Instr<sub>J</sub> tries to read operand before Instr<sub>I</sub> writes it



- or Instr<sub>J</sub> is data dependent on Instr<sub>K</sub> which is dependent on Instr<sub>I</sub>
- Caused by a “True Dependence” (compiler term)
- If true dependence caused a hazard in the pipeline, called a Read After Write (RAW)

### **Name Dependency:**

The name dependence occurs when two instructions use the same register or memory location, called a name, but there is no flow of data between the instructions associated with that name.

#### **There are two types of name dependences:**

1. **An anti dependence** between instruction i and instruction j occurs when instruction j writes a register or memory location that instruction i reads. The original ordering must be preserved to ensure that i reads the correct value.
2. **An output dependence** occurs when instruction i and instruction j write the same register or memory location. The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction j.

### **Register Renaming:**

- Renaming can be more easily done for register operands, where it is called register renaming.
- Register renaming can be done either statically by a compiler or dynamically by the hardware.

- Before describing dependences arising from branches, let's examine the relationship between dependences and pipeline data hazards.

**Hazards:** circumstances that would cause incorrect execution if next instruction were launched

- Structural hazards: Attempting to use the same hardware to do two different things at the same time
- Data hazards: Instruction depends on result of prior instruction still in the pipeline
- Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

**Data hazards:** Instruction depends on result of prior instruction still in the pipeline.

- Ignoring potential data hazards can result in [race conditions](#) (sometimes known as race hazards).

**There are three situations in which a data hazard can occur:**

- read after write (RAW), a *true dependency*
- write after read (WAR), an *anti-dependency*
- write after write (WAW), an *output dependency*

#### **Read After Write (RAW)**

Instr<sub>J</sub> tries to read operand before Instr<sub>I</sub> writes it



```


I:  add  r1, r2, r3
J:  sub  r4, r1, r3

```

Caused by a “Data Dependence” (in compiler nomenclature). This hazard results from an actual need for communication.

#### **Write After Read (WAR)**

Instr<sub>J</sub> writes operand before Instr<sub>I</sub> reads it Called an “anti-dependence” by compiler writers.



```

I:  sub  r4, r1, r3
J:  add  r1, r2, r3
K:  mul  r6, r1, r7

```

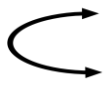
This results from reuse of the name “r1”.

- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and

- Reads are always in stage 2, and
- Writes are always in stage 5

### Write After Write (WAW)

Instr<sub>J</sub> writes operand before Instr<sub>I</sub> writes it is Called an “output dependence” by compiler writers



```

I:  sub  r1, r4, r3
J:  add  r1, r2, r3
K:  mul  r6, r1, r7

```

- This also results from the reuse of name “r1”.
- Can’t happen in MIPS 5 stage pipeline because:
- All instructions take 5 stages, and
- Writes are always in stage 5
- Will see WAR and WAW in later more complicated pipes

### Control dependence:

- Every instruction is control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program order

### Example:

```

if p1 {
    S1;
};
if p2 {
    S2;
}

```

- **S1 is control dependent on p1, and S2 is control dependent on p2 but not on p1.**

### In general, there are two constraints imposed by control dependences:

1. An instruction that is control dependent on a branch cannot be moved before the branch

**For example,** we cannot move the ‘then’ portion of an if-statement before the if-statement.

2. An instruction that is not control dependent on a branch cannot be moved after the branch

**For example,** we cannot take a statement before the if-statement and move it into the then-portion.

### ILP Architecture:

Information embedded in the program pertaining to available parallelism between instructions and operations in the program.

## **ILP architectures classifications:**

**1. Sequential Architecture:** The program is not executed to convey any explicit information regarding parallelism (superscalar processors)

**2. Dependency Architecture:** The program explicitly indicates the dependences that exist between operations (Dataflow processor)

**3. Independent Architecture:** The program provides information as to which operations are independent of one another.

	<b>Sequential Architecture</b>	<b>Dependence Architecture</b>	<b>Independence Architecture</b>
Additional info required in the program	None	Specification of dependence between operations	Minimally a partial list of independences .A complete specification of when and where each operation to be executed
Typical kind of ILP processor	Superscalar	Dataflow	VLIW
Dependences analysis	Performed by HW	Performed by compiler	Performed by compiler
Independences analysis	Performed by HW	Performed by HW	Performed by compiler
Scheduling	Performed by HW	Performed by HW	Performed by compiler
Role of compiler	Rearranges the code to make the analysis and scheduling HW more successful	Replaces some analysis HW	Replaces virtually all the analysis and scheduling HW

## **2. PARALLEL PROCESSING CHALLENGES      NOV/DEC 2014( 16 MARKS )**

Parallel processing will execute efficiently in performance and energy as the number of core per chip scales (multiprocessor).

Parallel processing challenges including scheduling, partitioning the task into parallel pieces, balancing the load evenly between the processors.

**Two major challenges:**

1. Limited parallelism
2. Long latency to remote memory.

### **1.Limited parallelism:**

- It is difficult to achieve good speedup in any parallel processing.
- Here using Amdahl's Law.

### **Amdhal's Law:**

- It calculates the performance improvement of a computer.
- It states that the performance improvement to be gained using faster mode of execution.

### **1.Speed up (performance improvement):**

It tells us how much faster a task can be executed using the machine with enhancement as compare to the original machine.

### **It defined as:**

$$\text{Speedup} = \frac{\text{performance for entire task using improved machine}}{\text{performance for entire task using old machine}}$$

**or**

$$\text{Speedup} = \frac{\text{execution time for entire task using old machine}}{\text{execution time for entire task using improved machine}}$$

$$\text{speedup} = \frac{ETO}{ETN}$$

**where :**

$$ETN = \frac{ETO}{ETO * [1 - Fe] + Fe / Se}$$

### **Equation:**

$$\text{Speedup} = \frac{1}{[1 - Fe] + Fe / Se}$$

**Where:**

**Se: Speedup enhanced:**

It tells how much faster the task would run if the enhancement mode was use for the entire program.



**Fe: Fraction enhanced:**

It is the fraction of the computation time in the original machine.

**Problem:**

What percentage of the original computation can be sequential to achieve a speedup of 80 times faster with 100 processors.

**Solution:**

Speedup = 80      speedup enhanced = 100    fraction=?

$$\text{speedup} = \frac{1}{[1 - Fe] + Fe/Se}$$

$$80 = \frac{1}{[1 - Fe] + \frac{Fe}{100}}$$

$$80 \times (1 - Fe) + \frac{Fe}{100} = 1$$

$$80 \times \frac{(100 - 100Fe) + Fe}{100} = 1$$

$$80 \times \frac{(100 - 99Fe)}{100} = 1$$

$$\frac{80}{100} (100 - 99Fe) = 1$$

$$0.8(100 - 99Fe) = 1$$

$$0.8 \times 100 - 0.8 \times 99Fe = 1$$

$$80 - 79.2Fe = 1$$

$$-79.2Fe = 1 - 80$$

$$-79.2Fe = -79$$

$$Fe = \frac{-79}{-79.2} = 0.9975$$

**2.Speedup challenges –increasing in problem size:**

Getting good speedup on a multiprocessor while keeping the problem size fixed is harder than getting good speedup by increasing the size of the problem.

**Problem:** we have to perform two sums. One is a sum of 20 scalar variables. One is a matrix sum of a pair of two dimensional array with dimension 20 by 20. let us assume only the

matrix sum is parallelizable. what speedup do we get with 10 versus 50 processors? Also calculate the speedups assuming the matrices grow to 40 by 40.

**Solution:**

**Single addition can performed in time t**

**20t addition do not benefit from parallel processors.**

**20 by 20 = 400t addition that do parallel processors.**

**Time required for single processor to perform all addition is 420 [ 400 + 20 additions]**

**The execution time for processor 10:**

*execution time after improvement* =  $\frac{\text{execution time affected by improvement}}{\text{Amount of improvement}} + \text{execution time affected}$

$$\text{execution time after improvement} = \frac{400 t}{10} + 20t = 60 t$$

**Speed up for processor 10:**

$$\text{speedup} = \frac{420 t}{60t} = 7$$

**The execution time for processor 50:**

$$\text{execution time after improvement} = \frac{400 t}{50} + 20t = 28 t$$

**Speed up for processor 50:**

$$\text{speedup} = \frac{420 t}{28t} = 15$$

**Problem size for 10 versus 50 processors:**

$$\frac{7}{10} * 100 = 70\% \text{ [ potential speed with 10 processor]}$$

$$\frac{15}{50} * 100 = 30\% \text{ [ potential speed with 50 processor]}$$

**Increasing matrix (40 by 40):**

**40 by 40 = 1600t addition that do parallel processors.**

**Time required for single processor to perform all addition is 1620[ 1600 + 20 additions]**

**The execution time for processor 10 versus 50:**

$$\text{execution time after improvement} = \frac{1600 t}{10} + 20t = 180 t$$

**Speed up for processor 10:**

$$speedup = \frac{1620t}{180t} = 9$$

$$speedup = \frac{420t}{60t} = 7$$

**The execution time for processor 50:**

$$execution\ time\ after\ improvement = \frac{1600t}{50} + 20t = 52t$$

**Speed up for processor 50:**

$$speedup = \frac{1620t}{52t} = 31.15$$

**Problem size for 10 versus 50 processors:**

$$\frac{9}{10} * 100 = 90\% \text{ [potential speed with 10 processor]}$$

$$\frac{31.15}{50} * 100 = 62.3\% \text{ [potential speed with 10 processor]}$$

**The above problem introduces two terms that describe ways to scale up.**

1. **Strong scaling:** speedup achieved on a multiprocessor without increasing the size of the problem.
2. **Weak scaling:** speedup achieved on a multiprocessor while increasing the size of the problem proportionally to the increase in the number of processors.

**3.Speedup challenge- Balancing Load**

**Problem:** we have achieved the speedup of 31.15 on the previous bigger problem size with 50 processors. In that problem we assumed the load was perfectly balanced. That is, each of the 50 processor performs 2% of the work. In this problem we have to calculate the impact on speedup if one processor's load is higher than all the rest. Calculate the impact on speedup if the hardest working processor's load is 4% and 10%. Also calculate the utilization of the rest of the processors?

**Solution:** we are taking 50 processors [split 50 processors has 1 processor and 49 processors]

**a. If one processor has 4 % of the parallel load, then it must do:**

$$4\% \times 1600 = 64t \text{ addition}$$

The other 49 processor will share the remaining 1536t addition [1600-64 = 1536]

**Calculate the execution time :**

$$\text{execution time after improvement} = \max\left[\frac{1536t}{49}, \frac{64t}{1}\right] + 20t = 84t$$

$$\text{speedup} = \frac{1620t}{84t} = 19.29$$

The speedup drops from 31.15 to 19.29

$$\text{Utilization of remaining 49 processors} = \frac{1536t}{49} = 31.35t$$

Thus we can say that the remaining 49 processors are utilized less than half the time as compared to 64t for hardest working processor.

**b. If one processor has 10 % of the parallel load, then it must do:**

$$10\% \times 1600 = 160t \text{ addition}$$

The other 49 processor will share the remaining 1440t addition [1600-160 = 1440]

**Calculate the execution time :**

$$\text{execution time after improvement} = \max\left[\frac{1440t}{49}, \frac{160t}{1}\right] + 20t = 180t$$

$$\text{speedup} = \frac{1620t}{180t} = 9$$

The speedup drops from 31.15 to 9

$$\text{Utilization of remaining 49 processors} = \frac{1440t}{49} = 29.39t$$

Thus we can say that the remaining 49 processors are utilized less than 20% of the time as compared to 160t for hardest working processor.

### **3. FLYNN'S CLASSIFICATION**

**( OR )**

**Discuss about SISD, MIMD, SIMD, SPMD and Vector systems**

**MAY/JUNE 2016, NOV/DEC 2015,**

**APR/MAY 2015 ( 16 MARKS )**

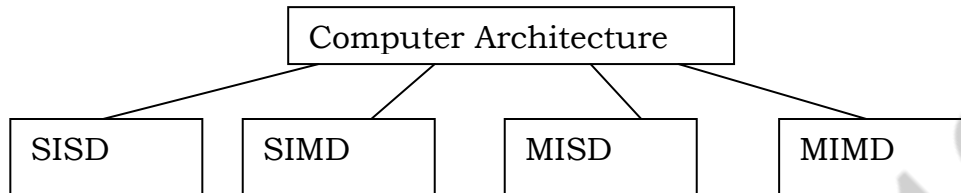
- In 1966, Michael Flynn proposed a classification for computer architectures based on the number of instruction streams and data streams (Flynn's Taxonomy).
- Flynn uses the stream concept for describing a machine's structure.
- A stream simply means a sequence of items (data or instructions).
- Processor unit operates by fetching instructions and operands from the main memory. Executing the instruction and placing the result in the main memory.

- The steps associated with the processing of instructions from an instruction cycle.
- The instruction can be stream flowing from main memory to processor.

### **Flynn's Taxonomy:**

- The classification of computer architectures based on the number of instruction streams and data streams (Flynn's Taxonomy).

Flynn looked at parallelism in the instruction and data stream called for by the instruction at the most constrained component of the multiprocessor and placed all computers into one of four categories.



1. SISD: Single instruction single data
2. SIMD: Single instruction multiple data
3. MISD: Multiple instructions single data
4. MIMD: Multiple instructions multiple data

### **Single Instruction stream, Single Data stream (SISD)**

- SISD corresponds to the traditional mono-processor (von Neumann computer). A single data stream is being processed by one instruction stream.
- A single-processor computer (uni-processor) in which a single stream of instructions is generated from the program.

In SISD computers instructions are executed sequentially but overlap in their execution stage.

They have more than one function unit but all functional units are control by a single control unit.

### **Single Instruction stream, Multiple Data stream (SIMD)**

- Each instruction is executed on a different set of data by different processors i.e multiple processing units of the same type process on multiple-data streams.
- This group is dedicated to array processing machines.
- Sometimes, vector processors can also be seen as a part of this group.
- They have multiple processing / execution unit and one control unit.
- All the processing /execution units are separated by the single control unit

### **Multiple Instruction stream, Single Data stream (MISD)**

- Each processor executes a different sequence of instructions.

- In case of MISD computers, multiple processing units operate on one single-data stream.
- In practice, this kind of organization has never been used

Until recently no processor that really fits this category

- “Streaming” processors; each processor executes a kernel on a stream of data
- Maybe VLIW?

### **Multiple Instruction stream, Multiple Data stream (MIMD)**

- Each processor has a separate program.
- An instruction stream is generated from each program.
- Each instruction operates on different data.
- This last machine type builds the group for the traditional multi-processors
- More than one processor unit having the ability to execute the several program simultaneously.
- In MIMD computer implies interactions among the multiple processor because all memory stream are derived from the same data space shared by all processors.

## **5. HARDWARE MULTITHREADING TECHNIQUES APR/MAY 2016& NOV/DEC 2014 (8 MARKS) , NOV/DEC 2015 & APR/MAY 2015 (16 MARKS)**

### **Multi-threading**

- ▶ The ability of an operating system to execute different parts of a program, called threads, simultaneously.
- ▶ The programmer must carefully design the program in such a way that all the threads can run at the same time without interfering with each other
- ▶ The multithreading paradigm has become more popular as efforts to further exploit instruction level parallelism have stalled.

### **Advantages of Multi-threading:**

- ▶ If a thread cannot use all the computing resources of the CPU, running another thread permits to not leave these idle.
- ▶ If several threads work on the same set of data, they can actually share its caching, leading to better cache usage or synchronization on its values.

### **Hardware multithreading:**

It allows multiple threads to share the functional units of a single processor in an overlapping fashion.

Increasing utilization of a processor by switching to another thread when one thread is stalled

### **Hardware Multithread done in 2 ways:**

#### **1. Fine-grained multi-threading :**

- Switches between threads on each instruction. causing the execution of multiple threads to be interleaved
- as the processor switches from one thread to the next, a thread that is currently stalled is skipped over
- CPU must be able to switch between threads at every clock cycle so that it needs extra hardware support

#### **Advantages**

- less susceptible to stalling situations
- throughput costs can be hidden because stalls are often unnoticed

#### **Disadvantages:**

- slows down execution of each thread
- requires a switching process that does not cost any cycles
- this can be done at the expense of more hardware (we will require at a minimum a PC for every thread)

#### **2. Coarse-grained multi-threading:**

- switches between threads only when current thread is likely to stall for some time (e.g., level 2 cache miss)
- the switching process can be more time consuming since we are not switching nearly as often and therefore does not need extra hardware support

#### **Advantages**

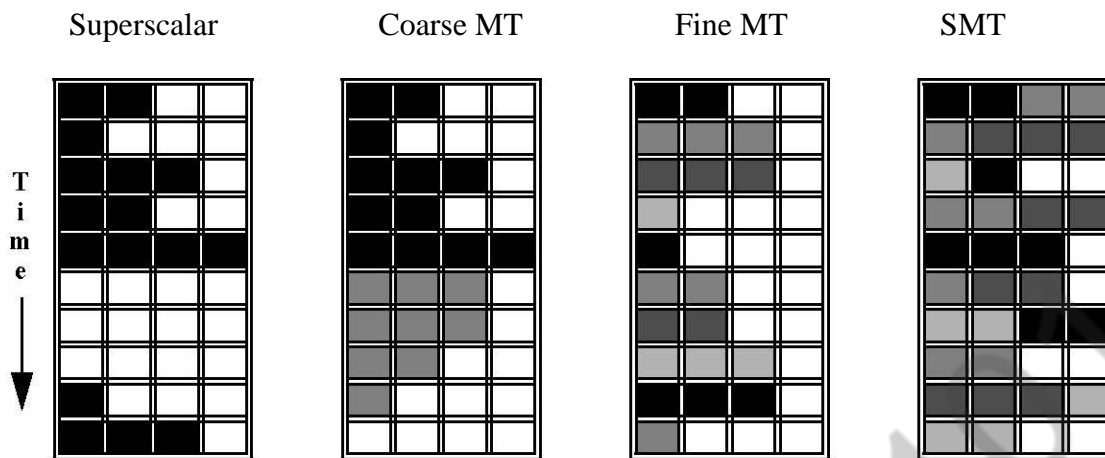
- more natural flow for any given thread
- easier to implement switching process
- can take advantage of current processors to implement coarse-grained, but not fine-grained

**Disadvantages:** limited in its ability to overcome throughput losses because of short stalling situations because the cost of starting the pipeline on a new thread is expensive (in comparison to fine-grained)

## Illustration:

The following diagram illustrates the differences in a processor's ability to exploit the resources of a superscalar for the following processor configuration

Issue Slots ►



**FIGURE 6.44** This illustration shows how these four different approaches use the issue slots of a superscalar processor. The horizontal dimension represents the instruction issue capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding issue slot is unused in that clock cycle. The shades of grey and black correspond to four different threads in the multithreading processors. Black is also used to indicate the occupied issue slots in the case of the superscalar without multithreading support.

- Horizontal dimension represents the instruction issue capability in each clock cycles.
- Vertical dimension represents a sequence of clock cycles.
- Empty slots indicate that the corresponding issue slots are unused in that clock cycles.

## Four Approaches:

### 1. Superscalar on a single thread (or) superscalar with no multithreading support

- The use of issue slots is limited by a lack of ILP.
- In addition a major stall, such as an instruction cache miss can leave the entire processor idle.

### 2. Superscalar with coarse-grained MT (c)

- the long stalls are partially hidden by switching to another that uses the resources of the processor.
- This reduces the number of completely idle clock cycle
- When there is a stall and the new thread has a start up period.

### 3. Superscalar with fine-grained MT (b)

- The interleaving of threads eliminates fully empty slots.



- Because only one thread issues instructions in a given clock cycle.
- However ILP limitations still lead to significant number of idle slots within individual clock cycles

#### 4. Superscalar with Simultaneous multithreading (d)

- In case, thread level parallelism and ILP are exploited simultaneously with multiple threads using the issue slots in a single clock cycle

### **6. MULTICORE PROCESSORS: APR/MAY 2016, NOV/DEC 2014 ( 8 MARKS )**

#### **Multicore:**

- A multi-core design in which a single physical processor contains the core logic of more than one processor.
- Multiprocessor systems allow true parallel execution; multiple threads or processes run simultaneously on multiple processors.
- An IC that has more than one core.
- All the core has their own functional unit.
- All the cores have their own L1 cache memory where L2 cache memory is shared among all the processor

#### **Multicore architecture are classified according to**

1. Number of processor
2. Approaches to processor-to -processor communication
3. Cache and memory implementations
4. Implementation of I/O bus and the Front Side Bus (FSB)

#### **Multicore configuration:**

- CPU manufacturers set about to develop multi-core configuration that allow multiple core to run in parallel on a single chip.

Configuration that support multiprocessing are:

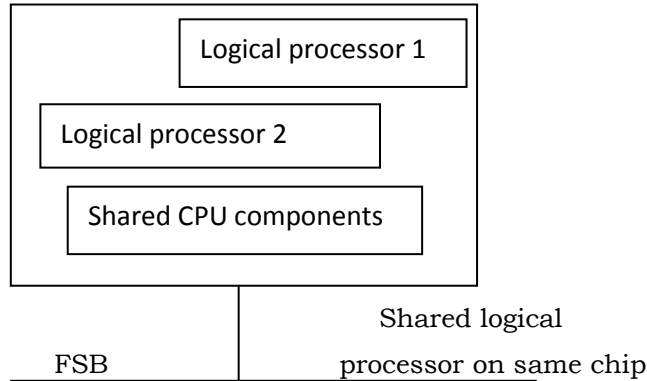
1. Hyper threading processing
2. Multiprocessor
3. Multi core

### Hyper threading processing:

With HT technology, parts of the one processor are shared between threads, while other parts are duplicated between them.

Ht technology literally interleaves the instructions in the execution pipeline.

- ♦ Two or more logical processors.
- ♦ Allows the scheduling logic maximum flexibility to fill execution slots.

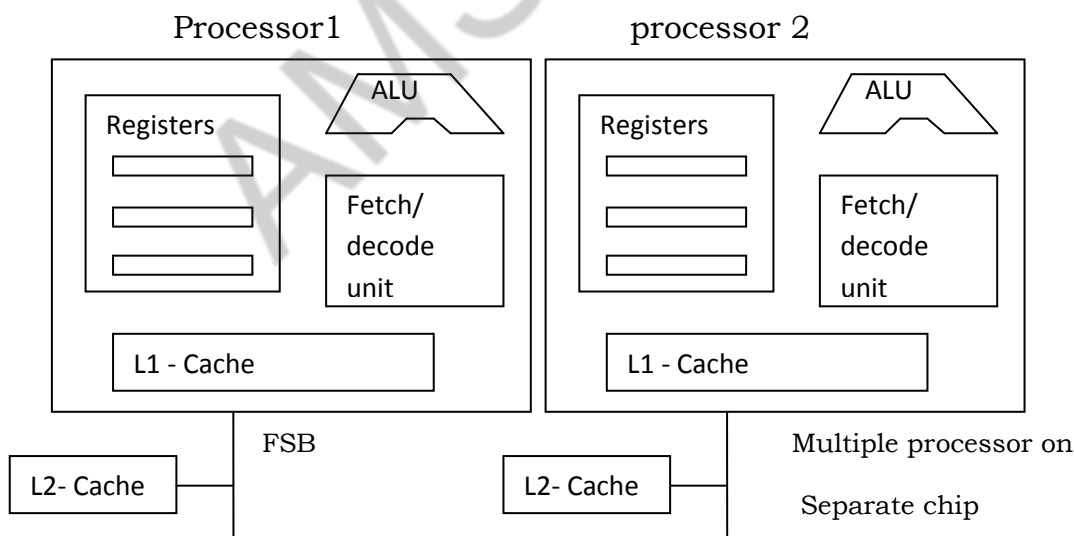


Hyper-threaded system uses a fraction of the resources and has a fraction of the waste of the SMP system.

- Provides more satisfactory solution
- Single physical processor is shared as two logical processors
- Each logical processor has its own architecture state
- Single set of execution units are shared between logical processors

### Multiprocessor:

A computer system in which two or more CPUs share full access to a common RAM



Continuous need for faster computers – shared memory model – message passing multiprocessor – wide area distributed system

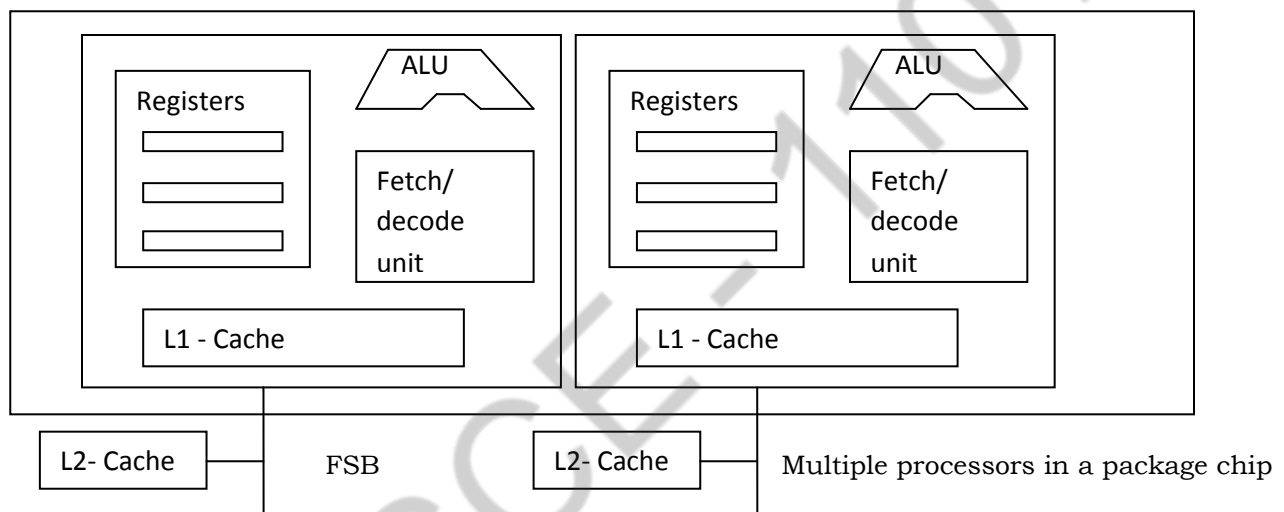
## Multicore :

- Chip-level multiprocessing(CMP or multicore): integrates two or more independent cores(normally a CPU) into a single package composed of a single integrated Circuit(IC), called a die, or more dies packaged, each executing threads independently.
- Every functional units of a processor is duplicated.
- Multiple processors, each with a full set of architectural resources, reside on the same die
- Processors may share an on-chip cache or each can have its own cache

Examples: HP Mako, IBM Power4

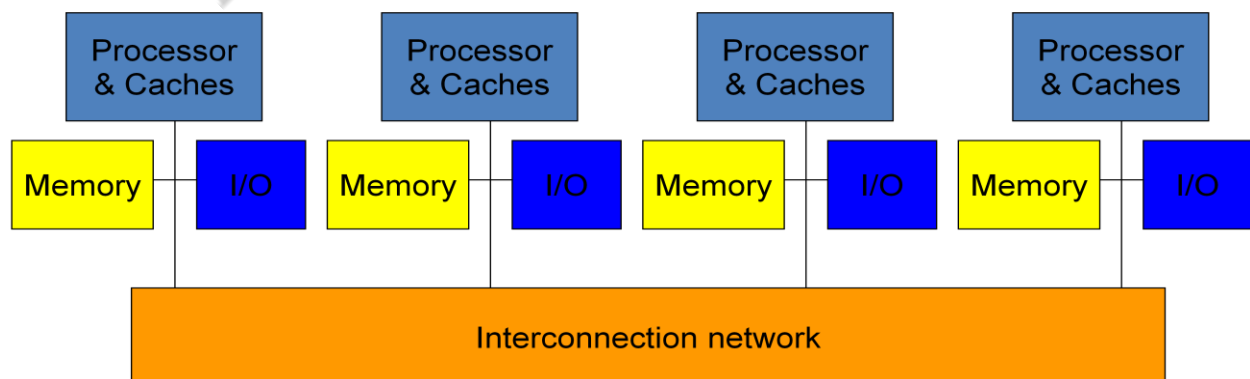
Challenges: Power, Die area (cost)

### Multicore(CMP)



## Shared Memory Architecture:

- A shared-memory multiprocessor (or just multiprocessor henceforth) is a computer system in which two or more CPUs share full access to a common RAM.
- A program running on any of the CPUs sees a normal (usually paged) virtual address space.



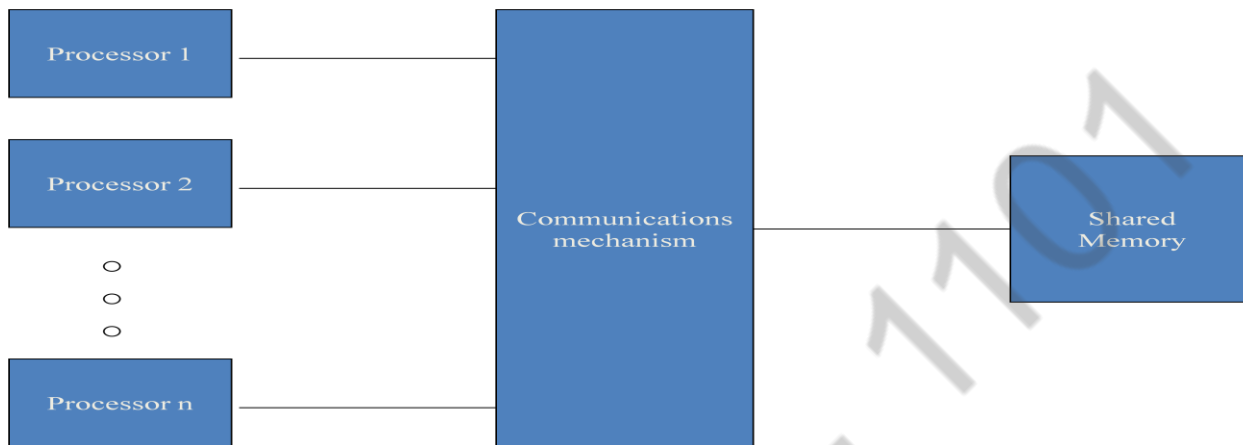
## Multiprocessor hardware:

Single physical address space multiprocessors are classified as

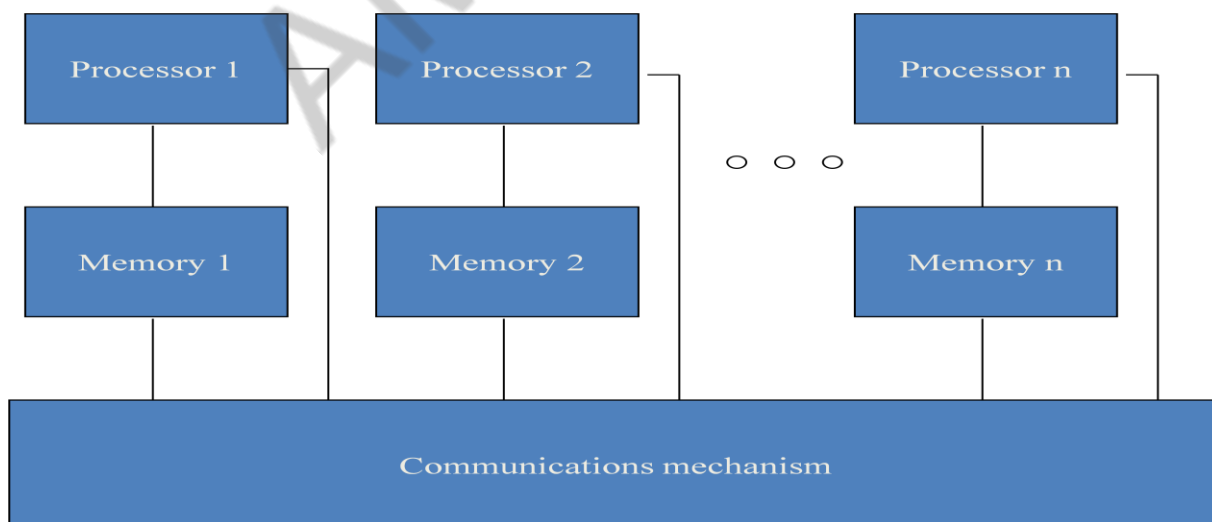
1. Uniform Memory Access (UMA)
2. Non-uniform Memory Access (NUMA)

### Uniform Memory Access (UMA)

- All memory addresses are reachable as fast as any other address.
- Time to access each memory word is the same
- Bus-based UMA
- CPUs connected to memory modules through switches



- The UMA is a type of symmetric multiprocessor, or SMP, that has two or more processors that perform symmetric functions.
- UMA gives all CPUs equal (uniform) access to all memory locations in shared memory.
- They interact with shared memory by some communications mechanism like a simple bus or a complex multistage interconnection network.
- **Non-uniform Memory Access (NUMA)**
- Some memory addresses are slower than others
- Memory distributed (partitioned among processors)
- Different access times for local and remote accesses



- NUMA architectures, unlike UMA architectures do not allow uniform access to all shared memory locations.
- This architecture still allows all processors to access all shared memory locations but in a non uniform way, each processor can access its local shared memory more quickly than the other memory modules not next to it.

AMSC-1101

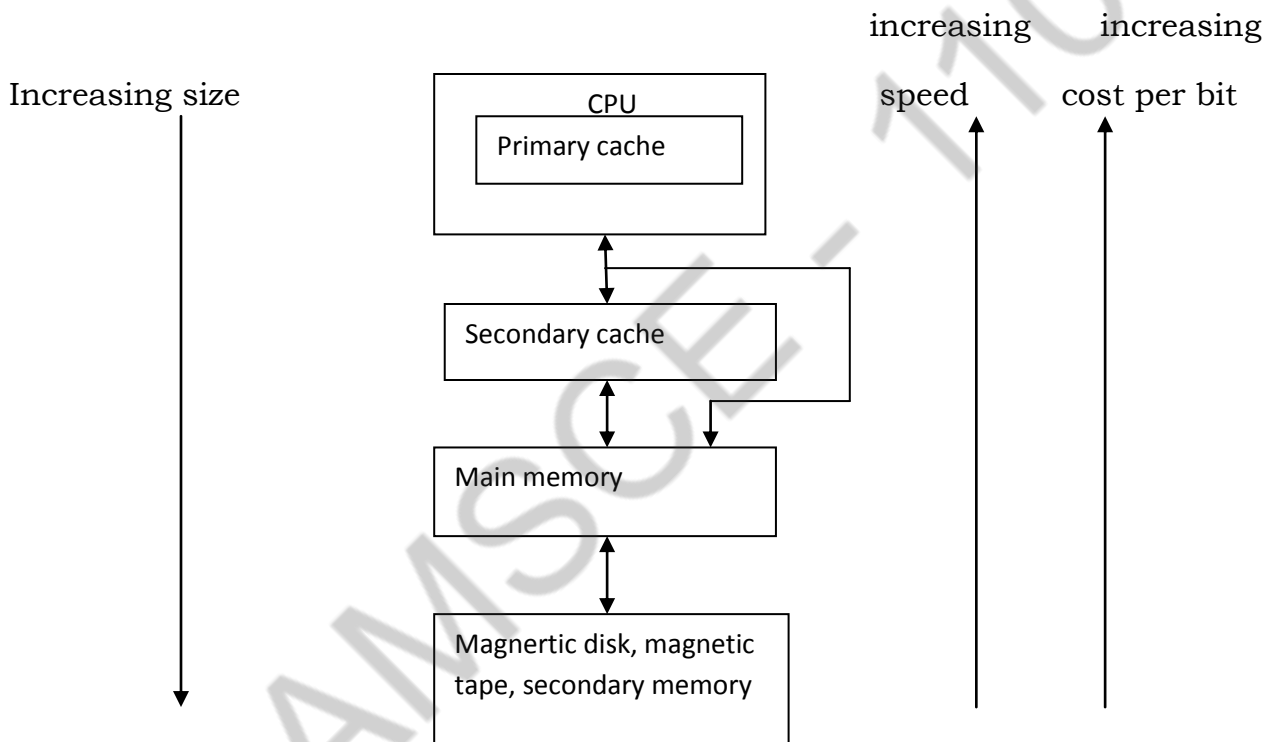
## Unit - 5

### Memory and I/O system

#### 1. MEMORY HIERARCHY:

- The memory unit is an essential component in any digital computer since it is needed for storing programs and data
- Computer memory should be fast , large and inexpensive. Unfortunately, it is impossible to meet all the three of these requirement
- The memory hierarchy system consists of all storage devices employed in a computer system from the slow by high-capacity auxiliary memory to a relatively faster main memory, to an even smaller and faster cache memory

#### Typical memory hierarchy:



#### main memory

- The memory units that directly communicate with CPU is called the *main memory*.
- The main memory occupies a central position by being able to communicate directly with the CPU.
- CPU logic is usually faster than main memory access time, with the result that processing speed is limited primarily by the speed of main memory.

#### Secondary memory:

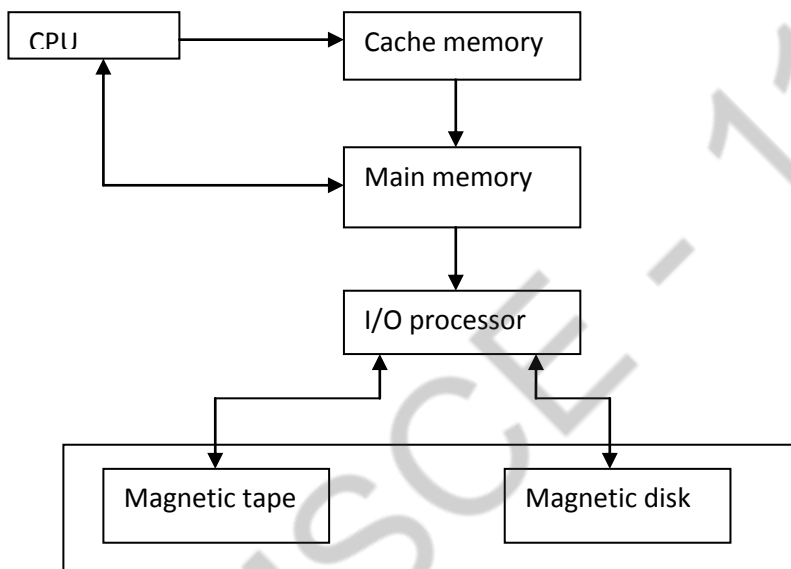
- Devices that provide backup storage are called secondary *memory*.

- Secondary memory devices through an I/O processor.
- Secondary memory access time is usually 1000 times that of main memory.

### Cache:

- A special very-high-speed memory called cache.
- It is used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate.
- The cache is used for storing segments of programs currently being executed in the CPU and temporary data frequently needed in the present calculations.
- The typical access time ratio between cache and main memory is about 1 to 7~10

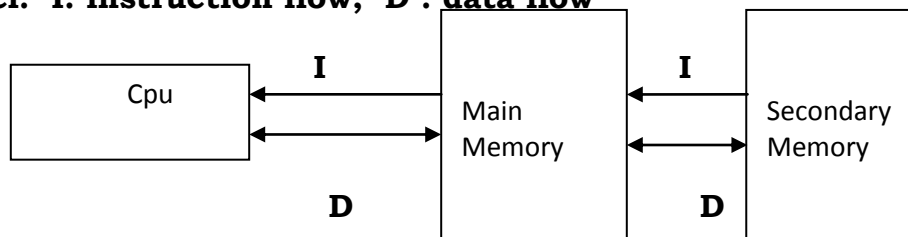
### The memory hierarchy can be employed in a computer system :



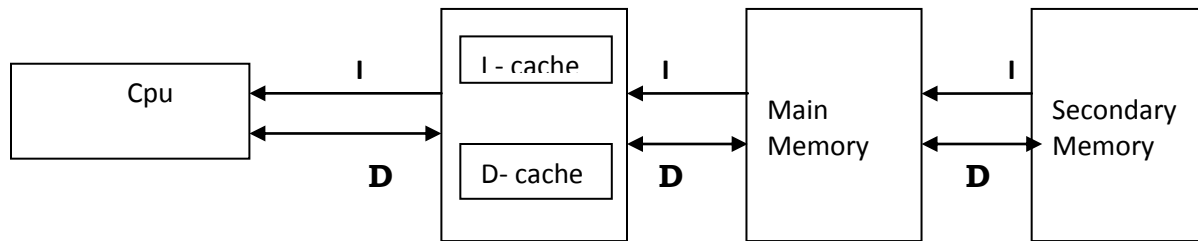
- Memory hierarchy in computer system magnetic tape and magnetic disk are used secondary memory. This is also known as auxiliary memory.
- The main memory occupies a central position by being able to communicate directly with cpu and with auxiliary memory device through an I/O processor.

### Memory hierarchy with two, three and four levels

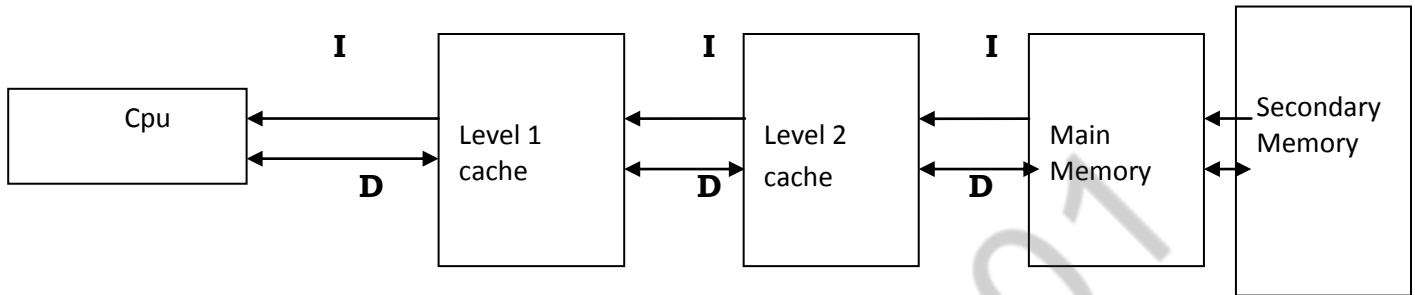
#### Two level: I: instruction flow, D : data flow



#### Three levels



#### Four level:



## 2. MEMORY TECHNOLOGY:

There are four primary technologies used today in memory hierarchies. Main memory is implemented from

1. DRAM (dynamic random access memory)
2. SRAM (static random access memory).
3. Flash Memory
4. Disk Memory

The fourth technology, used to implement the largest and slowest level in the hierarchy in servers, is magnetic disk

Memory Technology	Typical access time	\$per Gip in 2012
SRAM semiconductor memory	y 0.5–2.5 ns	\$500–\$1000
DRAM semiconductor memory	50–70 ns	\$10–\$20
Flash semiconductor memory	5,000–50,000 ns	\$0.75–\$1.00
Magnetic disk	5,000,000–20,000,000 ns	\$0.05–\$0.10

#### SRAM Technology:

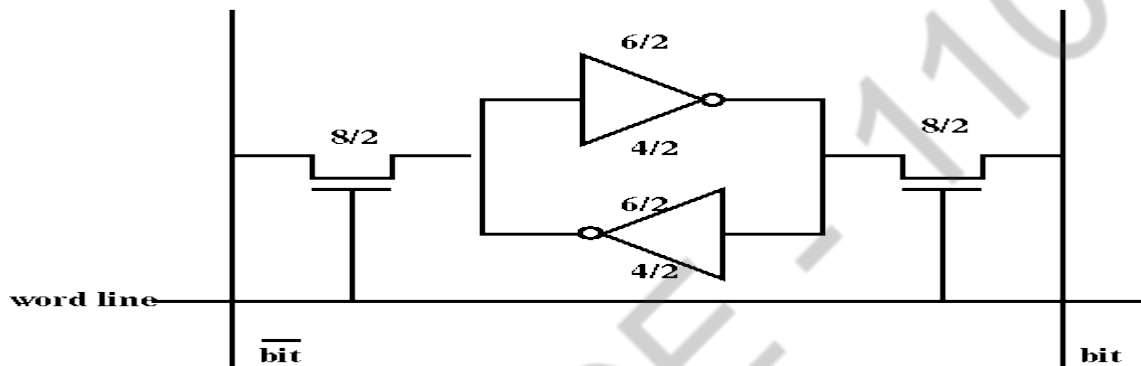
- SRAM = Static Random Access Memory
- Static: holds data as long as power is applied
- Volatile: cannot hold data if power is removed



- SRAMs are simply integrated circuits that are memory arrays with (usually) a single access port that can provide either a read or a write.
- SRAMs have a fixed access time to any datum, though the read and write access times may differ.

### SRAM CELL:

- It consists of 2 transistors act as a switch and 2 cross coupled inverters as a latch.
- The latch is connected to two bit lines by transistors T1 and T2.
- The word line controls the opening and closing of transistors T1 and T2.
- WL = 0, hold operation
- WL = 1, read or write operation



### 3 Operation States:

#### 1. Hold operation:

- word line = 0, access transistors are OFF. The data held in latch

#### 2. Write operation:

- word line = 1, access transistors are ON
- new data (voltage) applied to bit and  $\overline{\text{bit}}$
- data in latch overwritten with new value

#### 3. Read operation:

- word line = 1, access transistors are ON
- bit and  $\overline{\text{bit}}$  read by a sense amplifier

### Sense Amplifier

- basically a simple differential amplifier
- comparing the difference between bit and  $\overline{\text{bit}}$
- if bit >  $\overline{\text{bit}}$ , output is 1
- if bit <  $\overline{\text{bit}}$ , output is 0

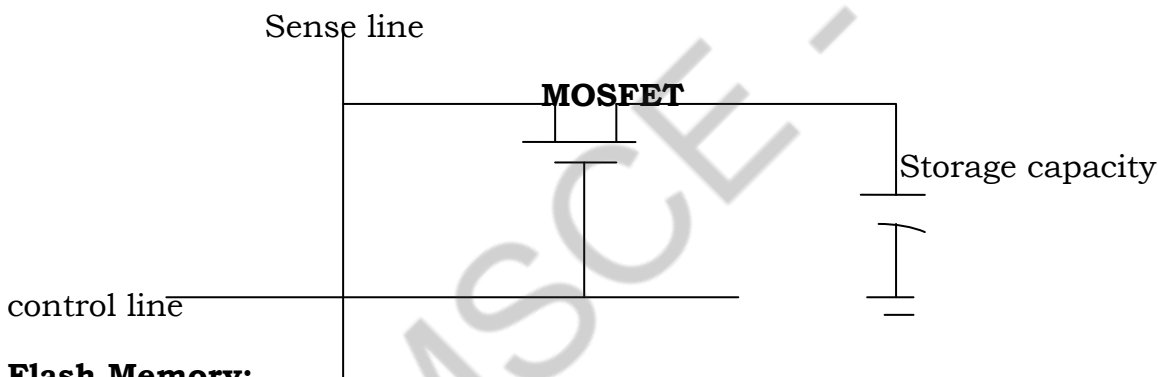
- allows output to be set quickly without fully charging/discharging bit line

### **DRAM Technology:** Dynamic Random Access Memory

- Dynamic: must be refreshed periodically
- Volatile: loses data when power is removed
- DRAM is less costly per bit than SRAM, although it is substantially slower.
- The price difference arises because DRAM uses significantly less area per bit of memory, and DRAMs thus have larger capacity for the same amount of silicon; the speed 0

### **DRAM CELL:**

- single access MOSFET (metal oxide semiconductor field effect transistor)
- storage capacitor (referenced to VDD or Ground)
- control input: word line, WL – data I/O: bit line
- RAM data is held on the storage capacitor
- temporary store data
- due to leakage currents which drain charge



### **Flash Memory:**

- Flash memory is a type of electrically erasable programmable read-only memory (EEPROM).
- Unlike disks and DRAM, but like other EEPROM technologies, writes can wear out flash memory bits.
- Most flash products include a controller to spread the writes by remapping blocks that have been written many times to less trodden blocks.
- This technique is called wear leveling.
- With wear leveling, personal mobile devices are very unlikely to exceed the write limits in the flash.
- Such wear leveling lowers the potential performance of flash, but it is needed unless higher level soft ware monitors block wear.
- Flash controllers that perform wear leveling can also improve yield by mapping out memory cells that were manufactured incorrectly.

### **Disk Memory:**

A magnetic hard disk consists of a collection of platters, which rotate on a spindle at 5400 to 15,000 revolutions per minute.

- The metal platters are covered with magnetic recording material on both sides, similar to the material found on a cassette or videotape.
- To read and write information on a hard disk,
- A movable *arm* containing a small electromagnetic coil called a *read-write head* is located just above each surface.
- The entire drive is permanently sealed to control the environment inside the drive, which, in turn, allows the disk heads to be much closer to the drive surface.

#### Track:

- Each disk surface is divided into concentric circles, called **tracks**.
- One of thousands of concentric circles that makes up the surface of a magnetic disk.

#### Sector:

One of the segments that make up a track on a magnetic disk; a sector is the smallest amount of information that is read or written on a disk.

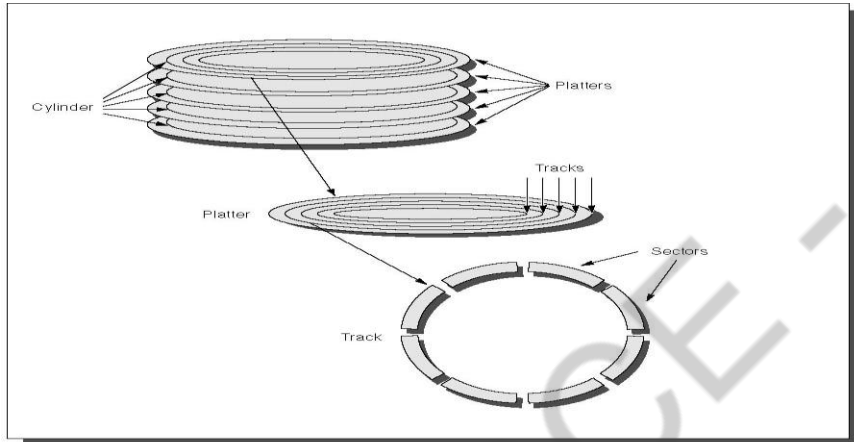


FIGURE 7.1 Disks are organized into platters, tracks, and sectors. Both sides of a platter are coated so that information can be stored on both surfaces. A cylinder refers to a track at the same position on every platter.

#### Seek

- To access data, the operating system must direct the disk through a three-stage process.
- The first step is to position the head over the proper track.
- This operation is called a **seek**, and the time to move the head to the desired track is called the *seek time*.

#### Different seek time:

Disk manufacturers report

- Minimum seek time
- Maximum seek time
- Average seek time in their manuals.

The first two are easy to measure, but the average is open to wide interpretation because it depends on the seek distance.

**Averages seek times:** Average seek times are usually advertised as 3 ms to 13 ms, but, depending on the application and scheduling of disk requests, the actual average seek time may be only 25% to 33% of the advertised number because of locality of disk references.

**Rotational latency or rotational delay:**

Once the head has reached the correct track, we must wait for the desired sector to rotate under the read/write head. This time is called the **rotational latency** or **rotational delay**. The average latency to the desired information is halfway around the disk. Disks rotate at 5400 RPM to 15,000 RPM.

The average rotational latency at 5400 RPM is

$$\text{Average rotational latency} = \frac{0.5 \text{ rotation RPM}}{5400 \text{ RPM}} = \frac{0.5 \text{ rotation RPM}}{5400 \frac{\text{RPM}}{60 \text{ seconds/minute}}} = 0.0056 \text{ seconds} \approx 5.6 \text{ ms}$$

**3. CACHE BASICS:**

**Q. Define Cache Memory? Explain the various techniques associated in cache memory**

**MAY/JUNE 2016, APR/MAY 2015, NOV/DEC 2014**

**Cache:**

- Cache is the fastest component in the memory hierarchy and the speed of CPU components.
- It is between in cpu and main memory .
- CPU access data from cache L1 and L2.
- L1 cache fabricated on CPU chip
- L2 cache between main memory and CPU.

**The following are some basic cache terms :**

- 1. Cache block** - The basic unit for cache storage. May contain multiple bytes/words of data.
  - 2. Cache line** - Same as cache block. Note that this is not the same thing as a “row” of cache.
  - 3. Cache set** - A “row” in the cache. The number of blocks per set is determined by the layout of the cache (e.g. direct mapped, set-associative, or fully associative).
  - 4. Tag** - A field in a table used for a memory hierarchy that contains the address information required to identify whether the associated block in the hierarchy corresponds to a requested word.
  - 5. Valid bit** - A bit of information that indicates whether the data in a block is valid (1) or not (0).
- The cache just before and after a reference to a word Xn that is not initially in the cache:**

Before the request, the cache contains a collection of recent references  $X_1, X_2, \dots, X_{n-1}$ , and the processor requests a word  $X_n$  that is not in the cache. This request results in a miss, and the word  $X_n$  is brought from memory into the cache.

**Before the reference to  $X_n$**

$X_4$
$X_1$
$X_{n-2}$
$X_{n-1}$
$X_2$
$X_3$

**After the reference to  $X_n$**

$X_4$
$X_1$
$X_{n-2}$
$X_{n-1}$
$X_2$
<b><math>X_n</math></b>
$X_3$

### Direct-mapped cache:

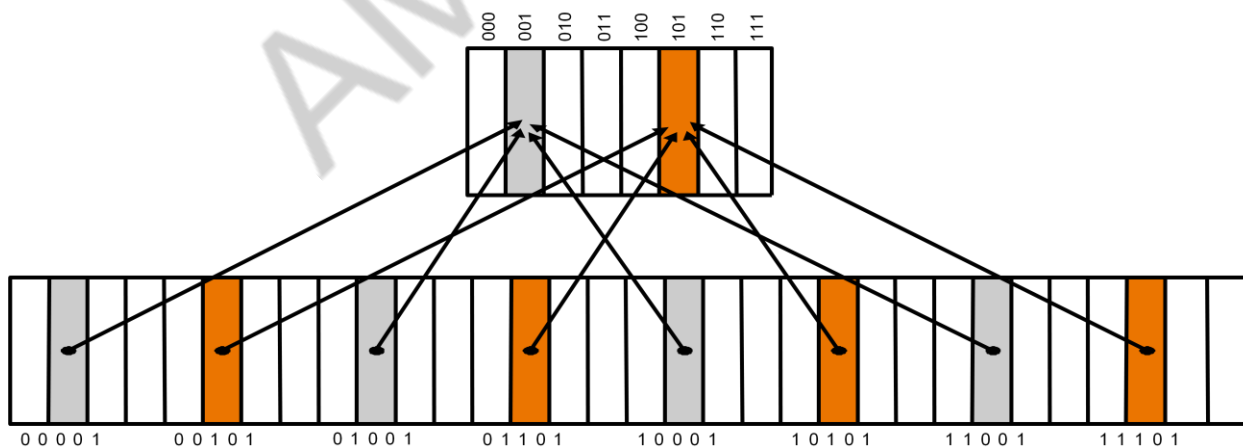
A cache structure in which each memory location is mapped directly to exactly one location in the cache.

**All direct-mapped caches use this mapping to find a block:**

(Block address) modulo (Number of blocks in the cache)

- If the number of entries in the cache is a power of 2
- Modulo can be computed simply by using the low-order  $\log_2$  (cache size in blocks) bits of the address.

**A direct-mapped cache with 8 entries showing the address of memory words between 0 and 31 that map to the same cache locations.**



Thus, addresses 00001two, 01001two, 10001two, and 11001two all map to entry 001two of the cache, while addresses 00101two, 01101two, 10101two, and 11101 two all map to entry 101two of the cache

### Accessing a cache:

The table shows how the contents of the cache change on each miss. Since there are eight blocks in the cache, the low-order three bits of an address give the block number:

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110two	miss (5.6b)	$(10\textcolor{teal}{110}\text{two} \bmod 8) = \textcolor{teal}{110}\text{two}$
26	11010two	miss (5.6c)	$(11\textcolor{teal}{010}\text{two} \bmod 8) = \textcolor{teal}{010}\text{two}$
22	10110two	hit	$(10\textcolor{teal}{110}\text{two} \bmod 8) = \textcolor{teal}{110}\text{two}$
26	11010two	hit	$(11\textcolor{teal}{010}\text{two} \bmod 8) = \textcolor{teal}{010}\text{two}$
16	10000two	miss (5.6d)	$(10\textcolor{teal}{000}\text{two} \bmod 8) = \textcolor{teal}{000}\text{two}$
3	00011two	miss (5.6e)	$(000\textcolor{teal}{11}\text{two} \bmod 8) = \textcolor{teal}{011}\text{two}$
16	10000two	hit	$(10\textcolor{teal}{000}\text{two} \bmod 8) = \textcolor{teal}{000}\text{two}$
18	10010two	miss (5.6f)	$(10\textcolor{teal}{010}\text{two} \bmod 8) = \textcolor{teal}{010}\text{two}$
16	10000two	hit	$(10\textcolor{teal}{000}\text{two} \bmod 8) = \textcolor{teal}{000}\text{two}$

The cache contents are shown after each reference request that misses, with the index and tag fields shown in binary for the sequence of addresses

a. **The initial state of the cache after power-on**

Index	Value	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

b. **After handling a miss of address (10110two)**

Index	Value	Tag	data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10two	Memory (10110two)
111	N		

c. **After handling a miss of address (11010two)**

Index	Value	Tag	Data
000	N		
001	N		
010	Y	11two	Memory(11010two)
011	N		

d. **After handling a miss of address (10000two)**

100	N		
101	N		
110	Y	10two	Memory (10110two)
111	N		

**g a miss of address (00011two)**

**After handling a miss of address (10010two)**

Index	Value	Tag	Data
000	Y	10two	Memory (10000two)
001	N		
010	Y	11two	Memory(11010two)
011	Y	00two	Memory (00011two)
100	N		
101	N		
110	Y	10two	Memory (10110two)
111	N		

00two (hit), 10010two (miss), and 10000two (hit).

The figures show the cache contents after each miss in the sequence has been handled. When address 10010two is referenced, the entry for address 11010two (26) must be replaced, and a reference to 11010two will cause a subsequent miss.

**cache for each possible address:**

The low-order bits of an address can be used to find the unique cache entry to which the address could map.

Figure shows how a referenced address is divided into

- **A tag field:** which is used to compare with the value of the tag field of the cache.
- **A cache index:** which is used to select the block 32 bit address.

■ The cache size is  $2^n$  blocks, so  $n$  bits are used for the index

Index	Value	Tag	data
000	y	10two	Memory (10000two)
001	N		
010	y	11two	Memory(11010two)
011	N		
100	N		
101	N		
110	Y	10two	Memory (10110two)
111	N		

**e.  
After  
handling**

The cache is initially empty, with all valid bits (V entry in cache) turned off (N).

**The processor requests the following addresses:**

10110two (miss), 11010two (miss), 10110two (hit), 11010two (hit), 10000two (miss), 00011two (miss),

100

Index	Value	Tag	data
000	y	10two	Memory (10000two)
001	N		
010	y	11two	Memory(11010two)
011	y	00two	Memory (00011two)
100	N		
101	N		
110	Y	10two	Memory (10110two)
111	N		

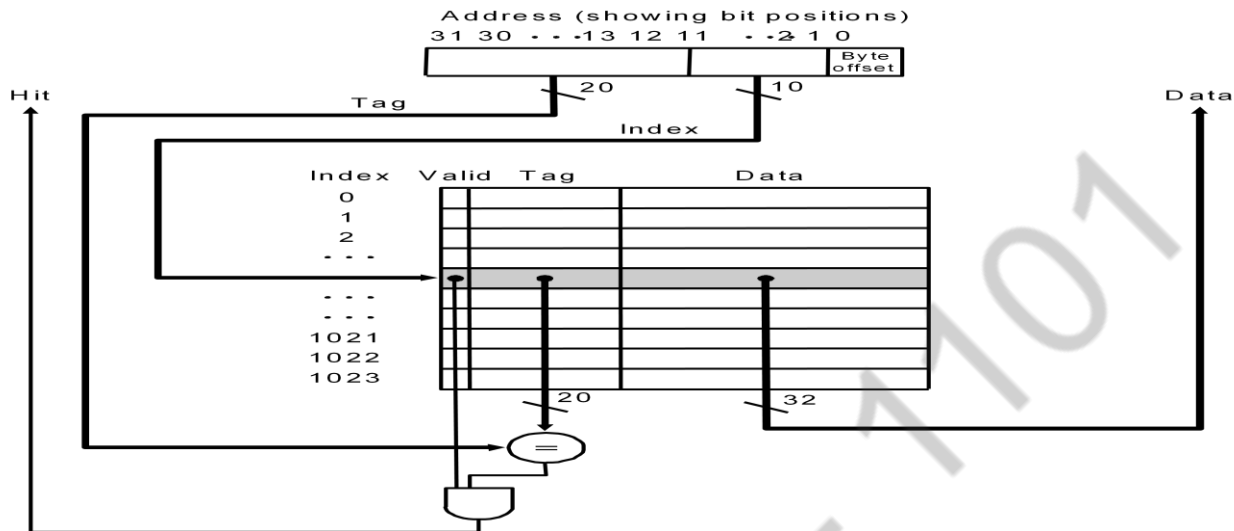
(18)

- The block size is  $2^m$  words ( $2^{m+2}$  bytes), so  $m$  bits are used for the word within the block, and two bits are used for the byte part of the address the size of the tag field is

$$32 - (n + m + 2).$$

The total number of bits in a direct-mapped cache is

$$2^n * (\text{block size} + \text{tag size} + \text{valid field size}).$$



- This cache holds 1024 words or 4 KiB. We assume 32-bit addresses in this chapter.
- The tag from the cache is compared against the upper portion of the address to determine whether the entry in the cache corresponds to the requested address.
- Because the cache has 210 (or 1024) words and a block size of one word, 10 bits are used to index the cache, leaving  $32 - 10 - 2 = 20$  bits to be compared against the tag.
- If the tag and upper 20 bits of the address are equal and the valid bit is on, then the request hits in the cache, and the word is supplied to the processor.
- Otherwise, a miss occurs.

### Handling Cache Misses:

#### cache miss:

- A request for data from the cache that cannot be filled because the data is not present in the cache.
- The control unit must detect a miss and process the miss by fetching the requested data from memory .

#### Cache Hit:

- If the cache reports a hit, the computer continues using the data as if nothing happened.



### **We can now define the steps to be taken on an instruction cache miss:**

1. Send the original PC value (current PC – 4) to the memory.
2. Instruct main memory to perform a read and wait for the memory to complete its access.
3. Write the cache entry, putting the data from memory in the data portion of the entry, writing the upper bits of the address (from the ALU) into the tag field, and turning the valid bit on.
4. Restart the instruction execution at the first step, which will re fetch the instruction, this time finding it in the cache.

The control of the cache on a data access is essentially identical: on a miss, we simply stall the processor until the memory responds with the data.

### **Handling Writes:**

**write-through:** The simplest way to keep the main memory and the cache consistent is always to write the data into both the memory and the cache. This scheme is called write-through.

**write buffer:** A queue that holds data while the data is waiting to be written to memory.

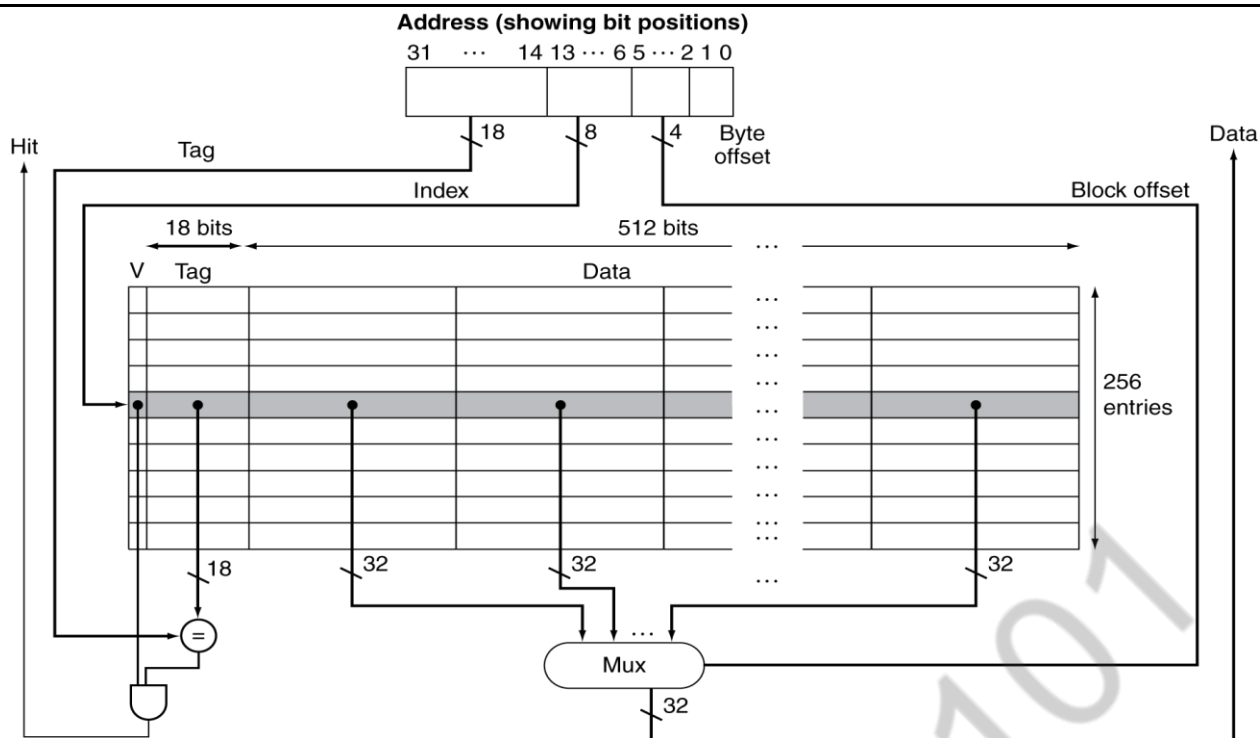
**write-back:** A scheme that handles writes by updating values only to the block in the cache, then writing the modified block to the lower level of the hierarchy when the block is replaced.

**Write Hit:** On data-write hit, could just update the block in cache, but then cache and memory would be inconsistent

### **Intrinsity FastMATH:**

- ◆ Embedded MIPS processor
  - 12-stage pipeline
  - Instruction and data access on each cycle
- ◆ Split cache: separate I-cache and D-cache
  - Each 16KB: 256 blocks × 16 words/block
  - D-cache: write-through or write-back
- ◆ SPEC2000 miss rates
  - I-cache: 0.4%
  - D-cache: 11.4%

Weighted average: 3.2%



#### 4. MEASURING AND IMPROVING CACHE

##### Two different techniques for improving cache performance:

- Reducing the miss rate by reducing the probability that two different memory blocks will contend for the same cache location.
- Reduces the miss penalty by adding an additional level to the hierarchy. This technique, called *multilevel caching*.

##### Miss Rate:

The fraction or percentage of accesses that result in a **miss** is called the **miss rate**.

$$\text{hit rate} + \text{miss rate} = 1.0 \text{ (100\%)}$$

##### Miss penalty:

The difference between lower level access time and cache access time is called the **miss penalty**.

##### CPU time:

CPU time can be divided into the clock cycles that the CPU spends executing the program and the clock cycles that the CPU spends waiting for the memory system.

##### Components of CPU time

## 1. Program execution cycles : Includes cache hit time

$$\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory-stall clock cycles}) \times \text{Clock cycle time}$$

## 2. Memory stall cycles :

Defined as the sum of the stall cycles coming from reads plus those coming from writes.

$$\text{Memory-stall clock cycles} = (\text{Read-stall cycles} + \text{Write-stall cycles})$$

$$\text{Read stall cycles} = \frac{\text{Reads}}{\text{program}} \times \text{Read miss rate} \times \text{Read Miss penalty}$$

$$\text{write stall cycles} = \frac{\text{writes}}{\text{program}} \times \text{write miss rate} \times \text{write Miss penalty}$$

$$\text{Memory stall cycles} = \frac{\text{Memory accesses}}{\text{program}} \times \text{miss rate} \times \text{Miss penalty}$$

We can also factor this as

$$\text{Memory stall cycles} = \frac{\text{Instructions}}{\text{program}} \times \frac{\text{Misses}}{\text{instruction}} \times \text{Miss penalty}$$

### Problem:

Assume the miss rate of an instruction cache is 2% and the miss rate of the data cache is 4%. If a processor has a CPI of 2 without any memory stalls and the miss penalty is 100 cycles for all misses, determine how much faster a processor would run with a perfect cache that never missed. Assume the frequency of all loads and stores is 36%.

#### ◆ Given

- I-cache miss rate = 2% [0.02]
- D-cache miss rate = 4% [0.04]
- Miss penalty = 100 cycles
- Base CPI (ideal cache) = 2
- Load & stores are 36% of instructions
- Miss cycles per instruction
- I-cache: Miss rate x miss penalty

$$0.02 \times 100 = 2$$

- D-cache: Miss rate x miss penalty

$$0.36 \times 0.04 \times 100 = 1.44$$

- Actual CPI is

Memory stall+ I-cache + D-cache

$$= 2 + 2 + 1.44 = 5.44$$

- Ideal CPU is

$$\diamond \frac{\text{Cpu time with stalls}}{\text{Cpu time with perfect cache}} = \frac{I \times \text{CPI stall} \times \text{clock cycle}}{I \times \text{CPI perfect} \times \text{clock cycle}} = \frac{\text{CPI stall}}{\text{CPI perfect}}$$

$$5.44/2 = 2.72 \text{ times faster}$$

### Average Access Time:

Hit time is also important for performance

### Hit Time:

### Average memory access time (AMAT):

Average memory access time is the average time to access memory considering both hits and misses and the frequency of different accesses

$$\text{AMAT} = \text{Hit time} + \text{Miss rate} \times \text{Miss Penalty}$$

### Problem:

Find the AMAT for a processor with a 1 ns clock cycle time, a miss penalty of 20 clock cycles, a miss rate of 0.05 misses per instruction, and a cache access time (including hit detection) of 1 clock cycle. Assume that the read and write miss penalties are the same and ignore other write stalls.

### Given:

CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%

AMAT = Time for a hit + Miss rate + Miss penalty

$$\text{AMAT} = 1 + 0.05 \times 20 = 2\text{ns}$$

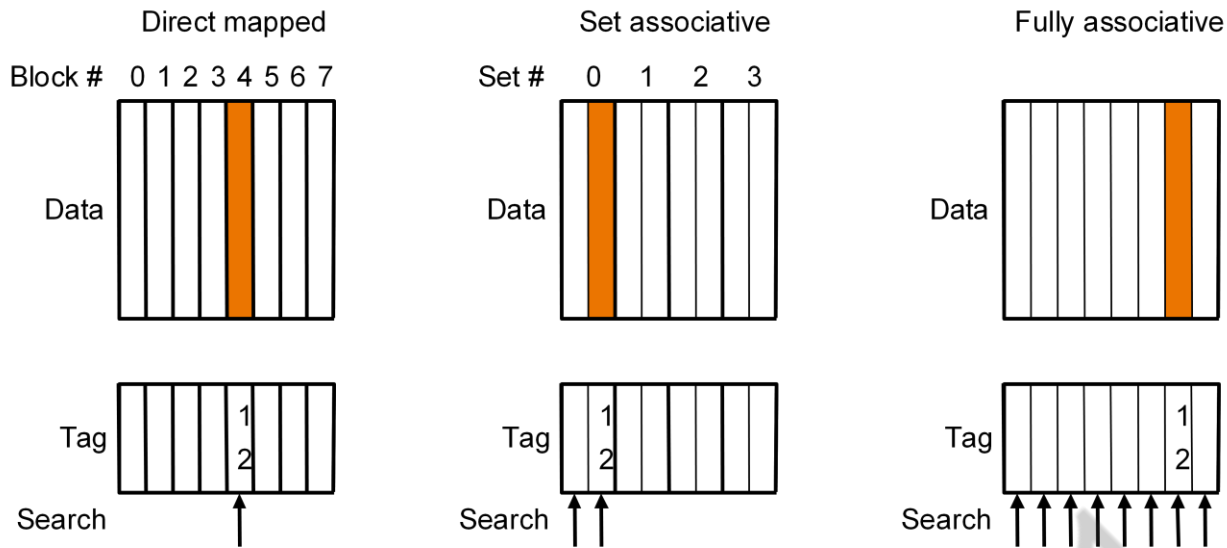
■ 2 cycles per instruction

### Reducing Cache Misses by More Flexible Placement of Blocks:

Allow more flexible block placement

### Commonly used methods:

1. Direct-Mapped Cache
2. fully associative cache :
3. Set-Associative Mapped Cache



**Direct mapped:** where a block can be placed in exactly one location, is at one extreme.

(Block number) modulo (Number of *blocks* in the cache)

**fully associative cache :** A cache structure in which a block can be placed in any location in the cache.

**Set-associative cache:** A cache that has a fixed number of locations (at least two) where each block can be placed.

(Block number) modulo (Number of *sets* in the cache)

Since the block may be placed in any element of the set, *all the tags of all the elements of the set* must be searched. In a fully associative cache, the block can go anywhere, and *all tags of all the blocks in the cache* must be searched.

**An eight-block cache configured as direct mapped, two way, four way and full way set associative:**

### One-way set associative (direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

### Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

### Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

### Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

The total size of the cache in blocks is equal to the number of sets times the associativity. Thus, for a fixed cache size, increasing the associativity decreases the number of sets while increasing the number of elements per set. With eight blocks, an eight-way set associative cache is the same as a fully associative cache.

### Locating a Block in the Cache:

In a direct-mapped cache, each block in a set-associative cache includes an address tag that gives the block address. The tag of every cache block within the appropriate set is checked to see if it matches the block address from the processor.

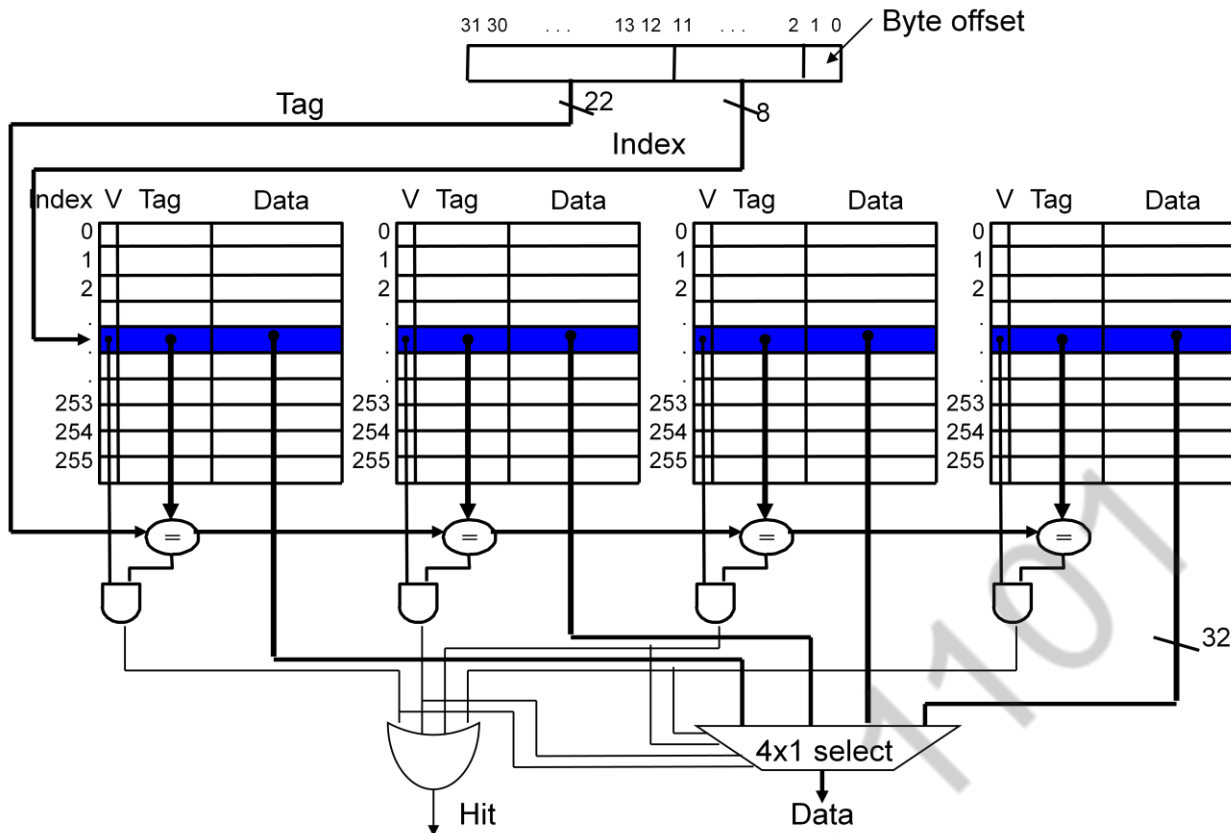
### The three portions of an address in a set-associative or direct-mapped cache:

Tag	Index	
-----	-------	--

The index is used to select the set, then the tag is used to choose the block by comparison with the blocks in the selected set. The block off set is the address of the desired data within the block.

### The implementation of a four-way set-associative cache requires four comparators and a 4-to-1 multiplexor:

In a direct-mapped cache, only a single comparator is needed, because the entry can be in only one block, and we access the cache simply by indexing.



The comparators determine which element of the selected set (if any) matches the tag. The output of the comparators is used to select the data from one of the four blocks of the indexed set, using a multiplexor with a decoded select signal. In some implementations, the Output enable signals on the data portions of the cache RAMs can be used to select the entry in the set that drives the output. The Output enable signal comes from the comparators, causing the element that matches to drive the data outputs. This organization eliminates the need for the multiplexor.

### Choosing Which Block to Replace:

#### ☐ When a miss occurs, which way's block do we pick for replacement?

**Least Recently Used (LRU):** the block replaced is the one that has been unused for the longest time

- Must have hardware to keep track of when each way's block was used relative to the other blocks in the set
- For 2-way set associative, takes one bit per set → set the bit when a block is referenced (and reset the other way's bit)

#### ☐ N-way set associative cache costs

- o N comparators (delay and area)

- MUX delay (set selection) before data is available
- Data available after set selection (and Hit/Miss decision). In a direct mapped cache, the cache block is available before the Hit/Miss decision
  - So its not possible to just assume a hit and continue and recover later if it was a miss

### **Reducing the Miss Penalty Using Multilevel Caches:**

#### **Use multiple levels of caches**

With advancing technology have more than enough room on the die for bigger L1 caches or for a second level of caches – normally a unified L2 cache (i.e., it holds both instructions and data) and in some cases even a unified L3 cache.

#### **Problem:**

CPI<sub>ideal</sub> of 2, 100 cycle miss penalty (to main memory), 36% load/stores, a 2% (4%) L1I\$ (D\$) miss rate, add a UL2\$ that has a 25 cycle miss penalty and a 0.5% miss rate

$$\text{CPI}_{\text{stalls}} = 2 + .02 \times 25 + .36 \times .04 \times 25 + .005 \times 100 + .36 \times .005 \times 100 = 3.54$$

(as compared to 5.44 with no L2\$)

#### **Design considerations for L1 and L2 caches are very different:**

- Primary cache should focus on minimizing hit time in support of a shorter clock cycle
  - Smaller with smaller block sizes
- Secondary cache(s) should focus on reducing miss rate to reduce the penalty of long main memory access times
  - Larger with larger block sizes
  - The miss penalty of the L1 cache is significantly reduced by the presence of an L2 cache – so it can be smaller (i.e., faster) but have a higher miss rate

#### **For the L2 cache, hit time is less important than miss rate:**

- The L2\$ hit time determines L1\$'s miss penalty
- L2\$ local miss rate >> than the global miss rate

### **5. Virtual Memory:**



**Q. What is virtual memory and explain virtual memory with neat diagram. NOV/DEC 2015, APR/MAY 2015**

- The main memory can act as a cache for the secondary storage, usually implemented with magnetic disks. This technique is called **virtual memory**.
- Main memory need contain only the active portions of the many virtual machines, just as a cache contains only the active portion of one program.
- The virtual machines sharing the memory change dynamically while the virtual machines are running.

**Two major motivations for virtual memory:**

1. to allow efficient and safe sharing of memory among multiple programs.
2. to allow a single user program to exceed the size of primary memory.

**Virtual memory mechanism:**

**Page:** A virtual memory block is called a **page**.

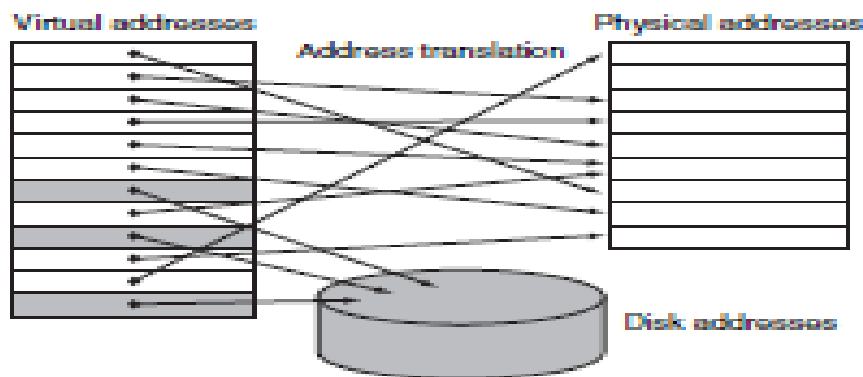
**Page fault:** An event that occurs when an access page is not present in main memory is called a **page fault**.

**Virtual address:** An address that corresponds to a location in virtual space and is translated by address mapping to a physical address when memory is accessed.

**Protection:** A set of mechanisms for ensuring that multiple processes sharing the processor, memory, or I/O devices cannot interfere, intentionally or unintentionally, with one another by reading or writing each other's data. These mechanisms also isolate the operating system from a user process.

**Address translation:** The virtually addressed memory with pages mapped to main memory. This process is called *address mapping* or **address translation**.

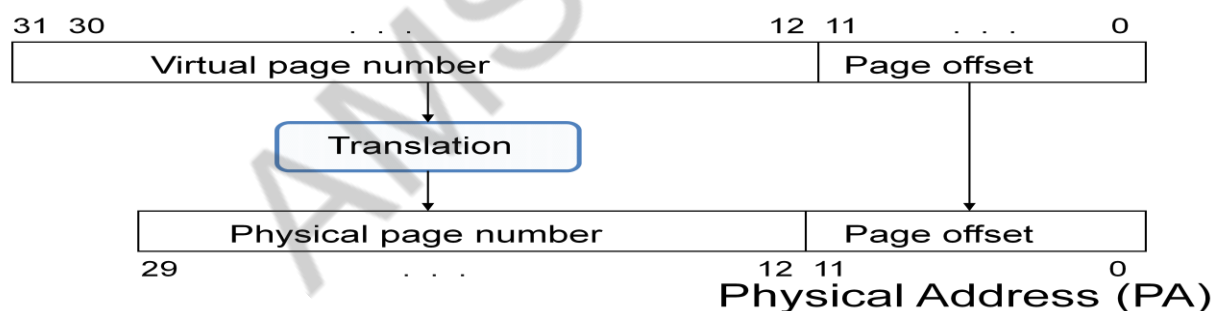
**In virtual memory, blocks of memory (called *pages*) are mapped from one set of addresses (called *virtual addresses*) to another set (called *physical addresses*).**



- The processor generates virtual addresses while the memory is accessed using physical addresses.
- Both the virtual memory and the physical memory are broken into pages, so that a virtual page is mapped to a physical page.
- It is also possible for a virtual page to be absent from main memory and not be mapped to a physical address; in that case, the page resides on disk.
- Physical pages can be shared by having two virtual addresses point to the same physical address.
- This capability is used to allow two different programs to share

#### **Mapping from a virtual to physical address:**

- A virtual address is translated to a physical address by a combination of hardware and software



- So each memory request *first* requires an address translation from the virtual space to the physical space
- A virtual memory miss (i.e., when the page is not in physical memory) is called a page fault
- The primary technique used here is to allow fully associative placement of pages in memory.

- Page faults can be handled in soft ware because the overhead will be small compared to the disk access time.

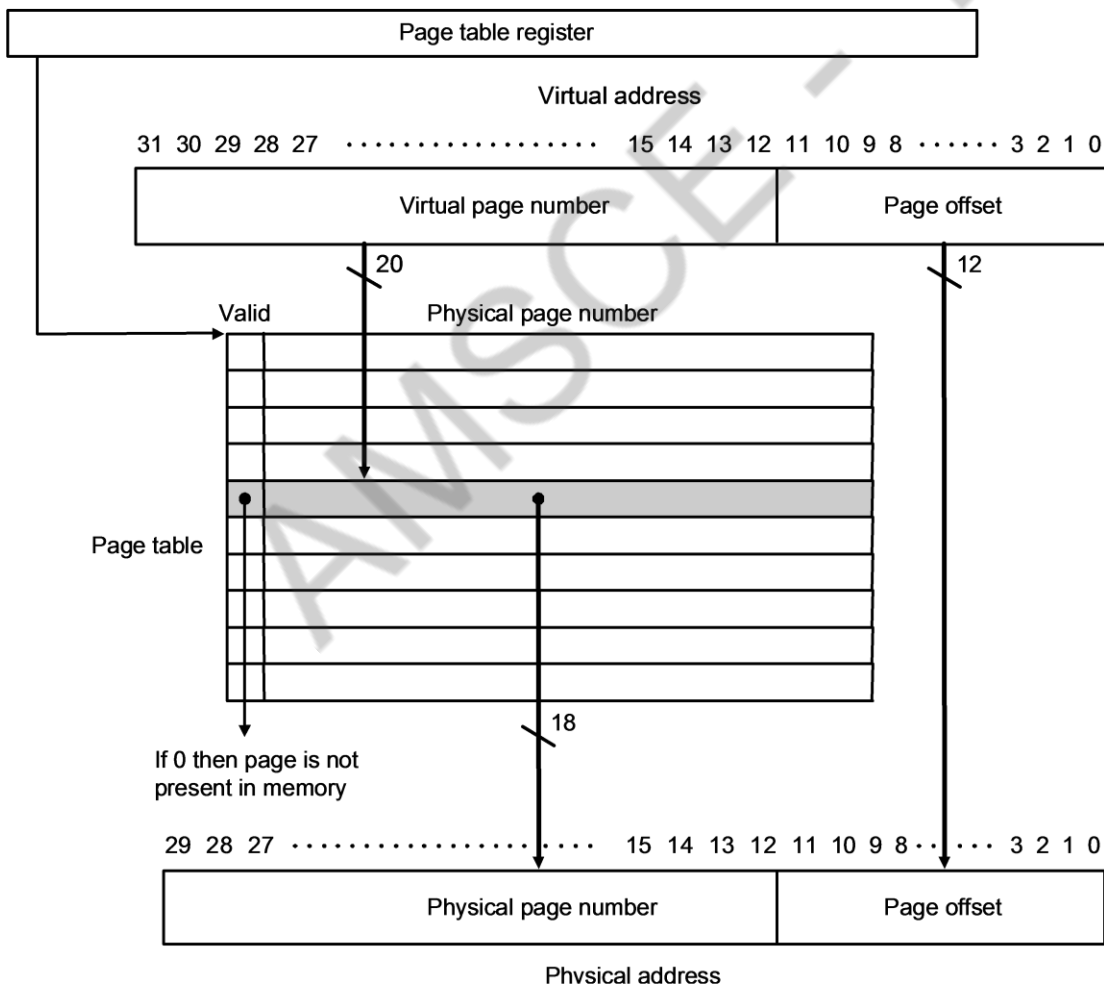
### Placing a Page and Finding It Again:

Because of the incredibly high penalty for a page fault, designers reduce page fault frequency by optimizing page placement. If we allow a virtual page to be mapped to any physical page, the operating system can then choose to replace any page it wants when a page fault occurs.

### Page table:

The table containing the virtual to physical address translations in a virtual memory system. The table, which is stored in memory, is typically indexed by the virtual page number; each entry in the table contains the physical page number for that virtual page if the page is currently in memory.

**The page table is indexed with the virtual page number to obtain the corresponding portion of the physical address:**



The location of the page table in memory, the hardware includes a register that points to the start of the page table; we call this the *page table register*.

A valid bit is used in each page table entry, just as we did in a cache.

If the bit is off, the page is not present in main memory and a page fault occurs.

If the bit is on, the page is in memory and the entry contains the physical page number.

Because the page table contains a mapping for every possible virtual page, no tags are required. In cache terminology, the index that is used to access the page table consists of the full block address, which is the virtual page number.

### Page Faults:

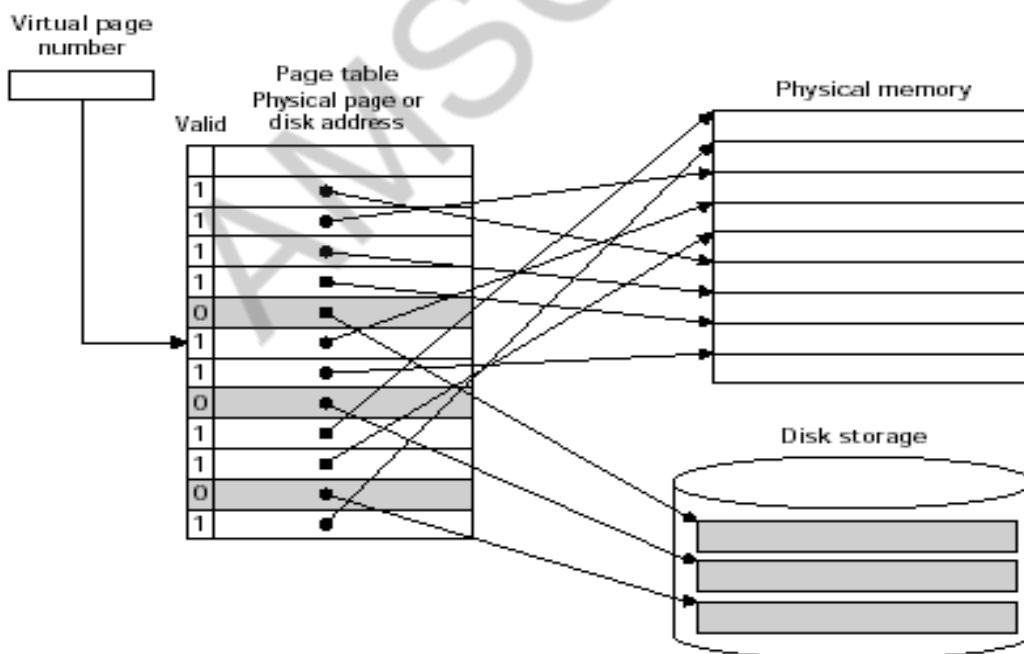
If the valid bit for a virtual page is off, a page fault occurs. The operating system must be given control. Once the operating system gets control, it must find the page in the next level of the hierarchy (usually flash memory or magnetic disk) and decide where to place the requested page in main memory.

### swap space :

The space on the disk reserved for the full virtual memory space of a process.

We want to minimize the number of page faults, most operating systems try to choose a page that they hypothesize will not be needed in the near future, using the past to predict the future. Operating systems follow the least recently used (LRU) replacement scheme.

**The page table maps each page in virtual memory to either a page in main memory or a page stored on disk, which is the next level in the hierarchy**

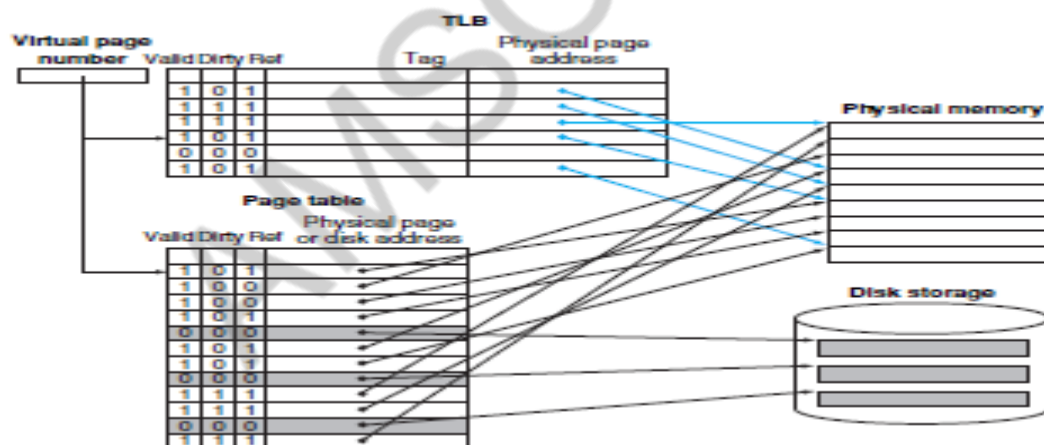


The virtual page number is used to index the page table. If the valid bit is on, the page table supplies the physical page number (i.e., the starting address of the page in memory) corresponding to the virtual page. If the valid bit is off, the page currently resides only on disk, at a specified disk address. In many systems, the table of physical page addresses and disk page addresses, while logically one table, is stored in two separate data structures. Dual tables are justified in part because we must keep the disk addresses of all the pages, even if they are currently in main memory. Remember that the pages in main memory and the pages on disk are the same size.

## 6. MAKING ADDRESS TRANSLATION FAST: THE TLB

A cache that keeps track of recently used address mappings to try to avoid an access to the page table.

The key to improving access performance is to rely on locality of reference to the page table. When a translation for a virtual page number is used, it will probably be needed again in the near future, because the references to the words on that page have both temporal and spatial locality. Modern processors include a special cache that keeps track of recently used translations. This special address translation cache is traditionally referred to as a **translation-lookaside buffer (TLB)**, although it would be more accurate to call it a translation cache.



The TLB contains a subset of the virtual-to-physical page mappings that are in the page table. The TLB mappings are shown in color. Because the TLB is a cache, it must have a tag field. If there is no matching entry in the TLB for a page, the page table must be examined. The page table either supplies a physical page number for the page (which can then be used to build a TLB entry) or indicates that the page resides on disk, in which case a page fault

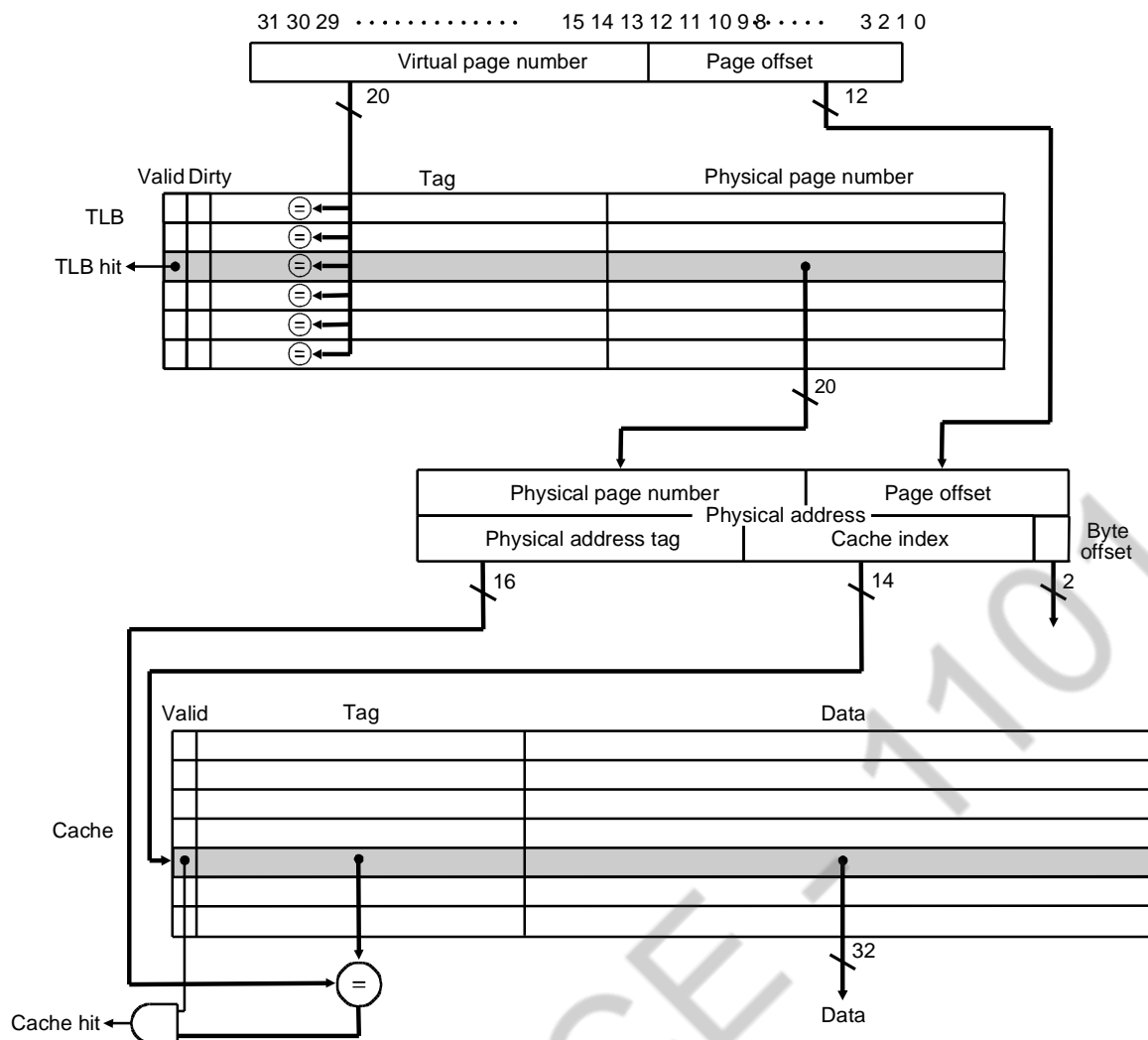
occurs. Since the page table has an entry for every virtual page, no tag field is needed; in other words, unlike a TLB, a page table is not a cache

### **Some typical values for a TLB might be**

- TLB size: 16–512 entries
- Block size: 1–2 page table entries (typically 4–8 bytes each)
- Hit time: 0.5–1 clock cycle
- Miss penalty: 10–100 clock cycles
- Miss rate: 0.01%–1%

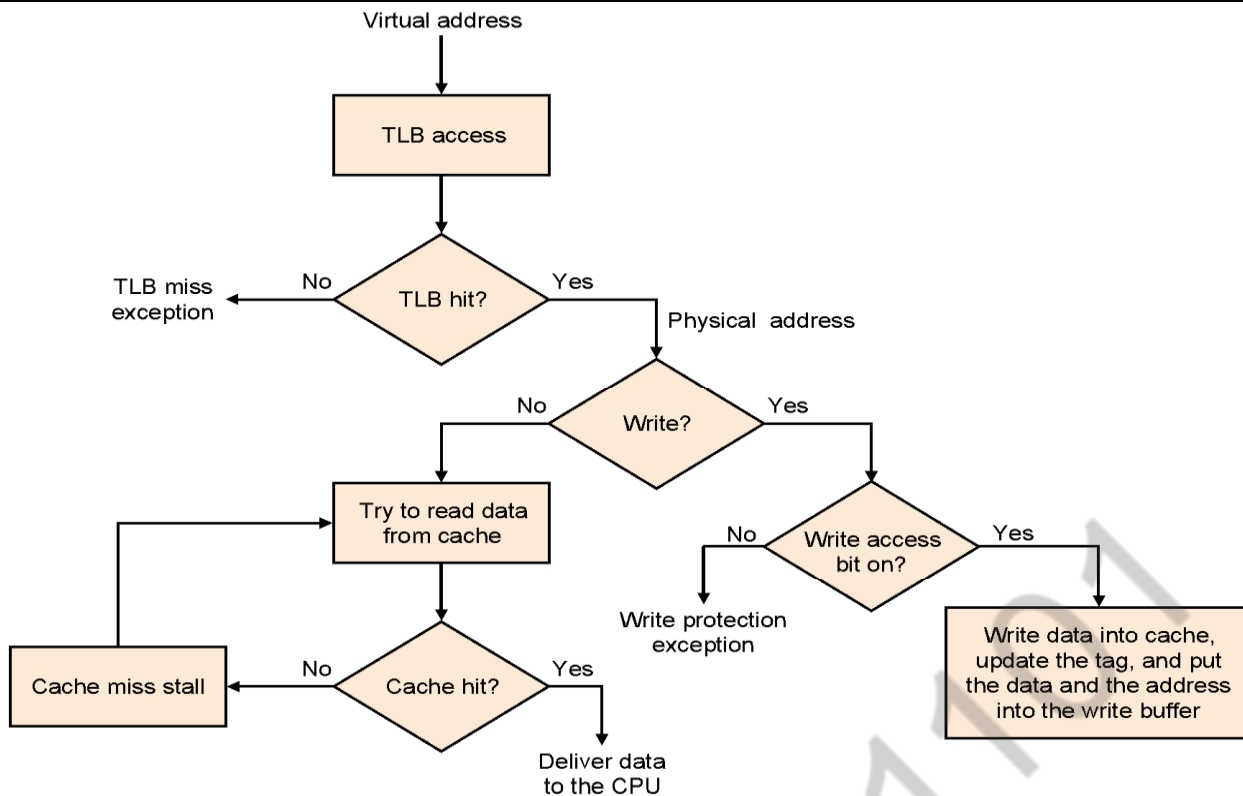
### **Integrating virtual memory, TLBs and caches**

Our virtual memory and cache systems work together as a hierarchy, so that data cannot be in the cache unless it is present in main memory. The operating system helps maintain this hierarchy by flushing the contents of any page from the cache when it decides to migrate that page to disk. At the same time, the OS modifies the page tables and TLB, so that an attempt to access any data on the migrated page will generate a page fault. Under the best of circumstances a virtual address is translated by TLB and sent to the cache where the appropriate data is found, retrieved and sent back to the processor. In worst case, a reference can miss in all three components of the memory hierarchy: The TLB, the page table and the cache.



### **DCEStation 3100 Write Through Write Not-Allocate Cache, sequence of events for read and write access**

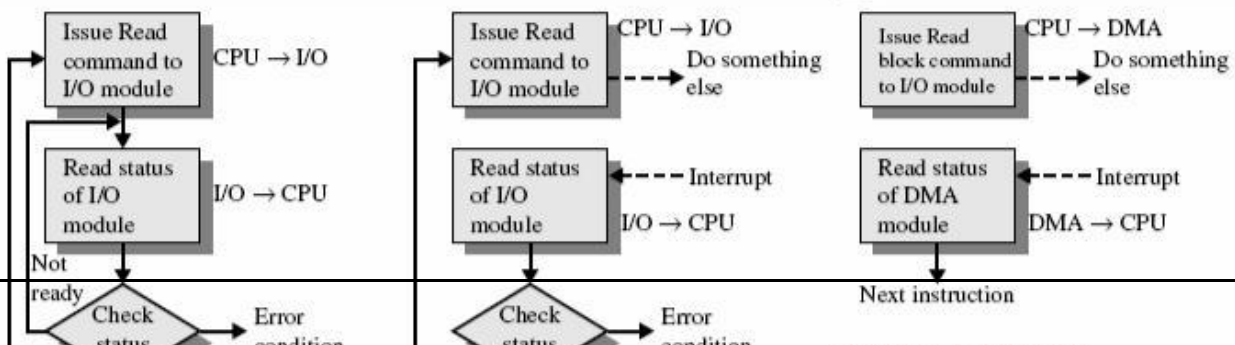
If the TLB generates a hit, the cache can be accessed with the resulting physical address. For a read, the cache generates a hit or miss and supplies the data or causes a stall while the data is brought from memory. If the operation is a write, a portion of the cache entry is overwritten for a hit and the data is sent to the write buffer if we assume write-through. A write miss is just like a read miss except that the block is modified after it is read from memory. Write-back requires writes to set a dirty bit for the cache block, and a write buffer is loaded with the whole block only on a read miss or write miss if the block to be replaced is dirty. Notice that a TLB hit and a cache hit are independent events, but a cache hit can only occur after a TLB hit occurs, which means that the data must be present in memory. The relationship between TLB misses and cache



## 7. PROGRAMMED I/O

### Overview of Programmed I/O

- Processor executes an I/O instruction by issuing command to appropriate I/O module
  - I/O module performs the requested action and then sets the appropriate bits in the I/O status register – I/O module takes no further action to alert the processor – it does not interrupt the processor
  - The processor periodically checks the status of the I/O module until it determines that the operation is complete
- I/O Commands The processor issues an address, specifying I/O module and device, and an I/O command. The commands are:
- **Control:** activate a peripheral and tell it what to do
  - **Test:** test various status conditions associated with an I/O module and its peripherals
  - **Read:** causes the I/O module to obtain an item of data from the peripheral and place it into an internal register
  - **Write:** causes the I/O module to take a unit of data from the data bus and





### Three Techniques for Input of a Block of Data

#### **I/O Instructions**

Processor views I/O operations in a similar manner as memory operations Each device is given a unique identifier or address

Processor issues commands containing device address – I/O module must check address lines to see if the command is for itself.

#### **I/O mapping**

##### **Memory-mapped I/O**

¾ Single address space for both memory and I/O devices

- Disadvantage – uses up valuable memory address space

¾ I/O module registers treated as memory addresses

¾ Same machine instructions used to access both memory and I/O devices

- Advantage – allows for more efficient programming

¾ Single read line and single write lines needed

¾ Commonly used

## • Isolated I/O

- ¾ Separate address space for both memory and I/O devices
- ¾ Separate memory and I/O select lines needed
- ¾ Small number of I/O instructions
- ¾ Commonly used

## 8. DMA AND INTERRUPTS

**Q. EXPLAIN BRIEFLY ABOUT DMA CONTROLLER WITH NEAT DIAGRAM. MAY/JUN 2016, NOV/DEC 2015 (16 MARKS) AND NOV/DEC 2014 (8 MARKS)**

### Interrupt-Driven I/O

- Overcomes the processor having to wait long periods of time for I/O modules
- The processor does not have to repeatedly check the I/O module status

### I/O module view point

- I/O module receives a READ command from the processor
- I/O module reads data from desired peripheral into data register
- I/O module interrupts the processor
- I/O module waits until data is requested by the processor
- I/O module places data on the data bus when requested

### Processor view point

- The processor issues a READ command
- The processor performs some other useful work
- The processor checks for interrupts at the end of the instruction cycle
- The processor saves the current context when interrupted by the I/O module
- The processor reads the data from the I/O module and stores it in memory
- The processor restores the saved context and resumes execution

### Design Issues

- a. How does the processor determine which device issued the interrupt
- b. How are multiple interrupts dealt with
- c. Device identification
- d. Multiple interrupt lines – each line may have multiple I/O modules
- e. Software poll – poll each I/O module
- f. Separate command line – TEST I/O
- g. Processor reads status register of I/O module Time consuming
- h. Daisy chain
- i. Hardware poll

- j. Common interrupt request line Processor sends interrupt acknowledge
- k. Requesting I/O module places a word of data on the data lines – —vector that uniquely identifies the I/O module – vectored interrupt

### • Bus arbitration

- ¾ I/O module first gains control of the bus
- ¾ I/O module sends interrupt request
- ¾ The processor acknowledges the interrupt request
- ¾ I/O module places its vector of the data lines

### Multiple interrupts

- The techniques above not only identify the requesting I/O module but provide methods of assigning priorities
- Multiple lines – processor picks line with highest priority
- Software polling – polling order determines priority
- Daisy chain – daisy chain order of the modules determines priority
- Bus arbitration – arbitration scheme determines priority

### Intel 82C59A Interrupt Controller

Intel 80386 provides

- Single Interrupt Request line – INTR
  - Single Interrupt Acknowledge line – INTA
  - Connects to an external interrupt arbiter, 82C59A, to handle multiple devices and priority structures
  - 8 external devices can be connected to the 82C59A – can be cascaded to 64 82C59A operation
- only manages interrupts

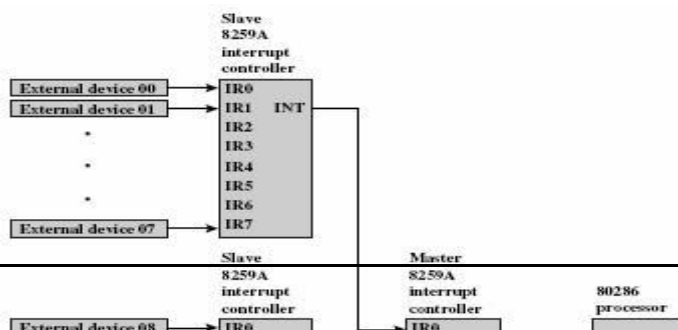
Accepts interrupt requests

- Determines interrupt priority
- Signals the processor using INTR
- Processor acknowledges using INTA
- Places vector information of data bus
- Processor process interrupt and communicates directly with I/O module

### 82C59A interrupt modes

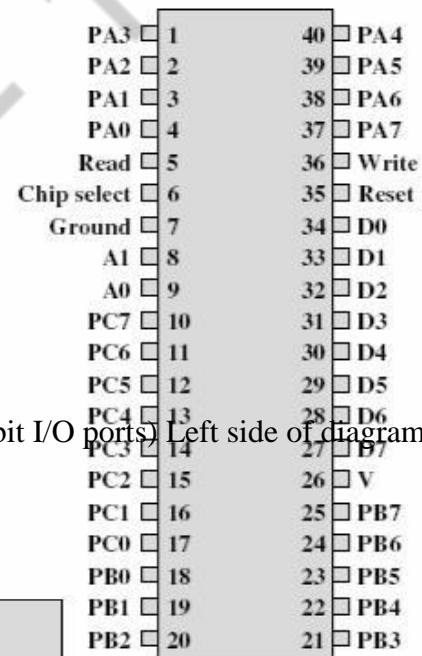
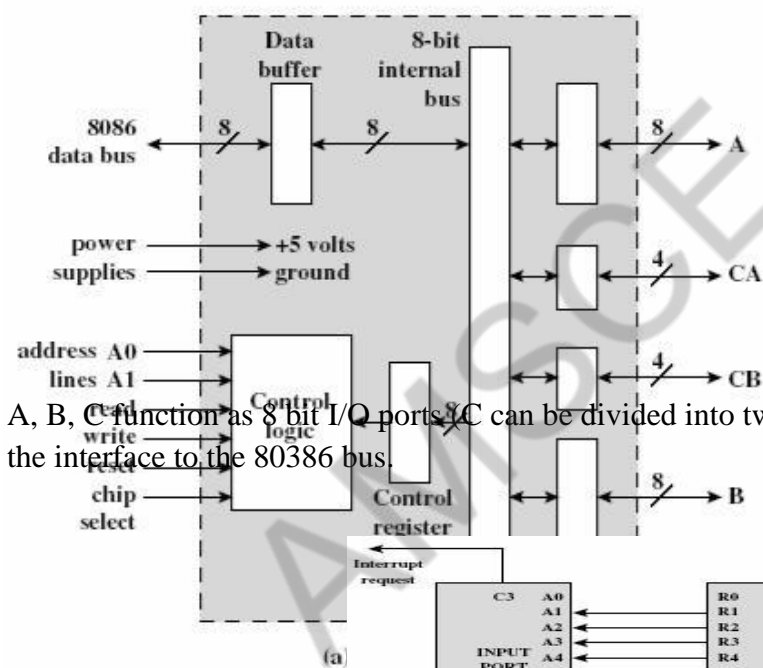
Fully nested – priority form 0 (IR0) to 7 (IR7)

Rotating – several devices same priority - most recently device lowest priority Special mask – processor can inhibit interrupts from selected devices.

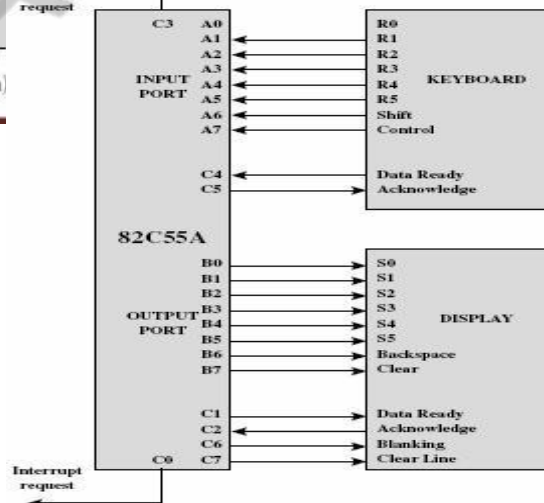


## Intel 82C55A Programmable Peripheral Interface

- ¾ Single chip, general purpose I/O module
- ¾ Designed for use with the Intel 80386
- ¾ Can control a variety of simple peripheral devices



(b) Pin layout



A, B, C function as 8 bit I/O ports. C can be divided into two 4 bit I/O ports. Left side of diagram show the interface to the 80386 bus.

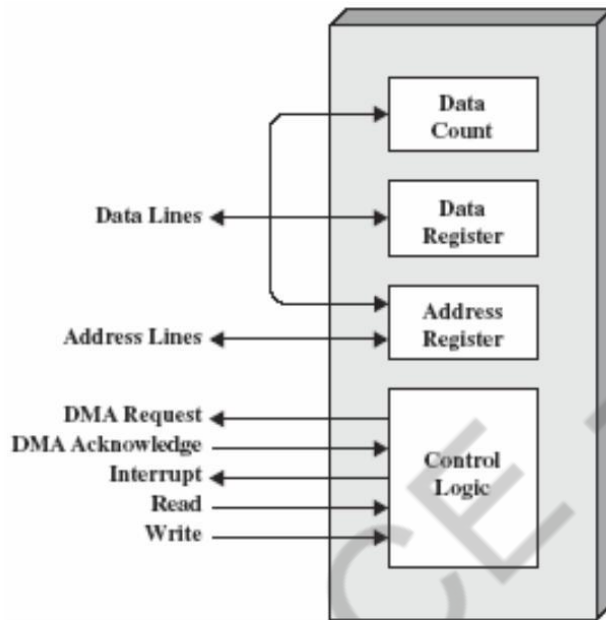
## Direct Memory Access

### Drawback of Programmed and Interrupt-Driven I/O

- I/O transfer rate limited to speed that processor can test and service devices
- Processor tied up managing I/O transfers

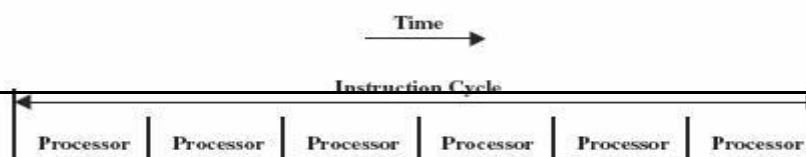
### MA Function

- DMA module on system bus used to mimic the processor.
- DMA module only uses system bus when processor does not need it.
- DMA module may temporarily force processor to suspend operations – cycle stealing.



### DMA Operation

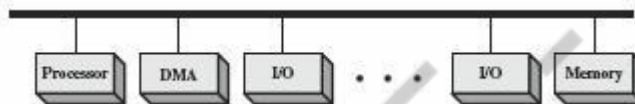
- x The processor issues a command to DMA module
- x Read or write
- x I/O device address using data lines
- x Starting memory address using data lines – stored in address register
- x Number of words to be transferred using data lines – stored in data register
- x The processor then continues with other work
- x DMA module transfers the entire block of data – one word at a time – directly to or from memory without going through the processor DMA module sends an interrupt to the processor when complete



## DMA and Interrupt Breakpoints during Instruction Cycle

- The processor is suspended just before it needs to use the bus.
- The DMA module transfers one word and returns control to the processor.
- Since this is not an interrupt the processor does not have to save context.
- The processor executes more slowly, but this is still far more efficient than either programmed or interrupt-driven I/O.

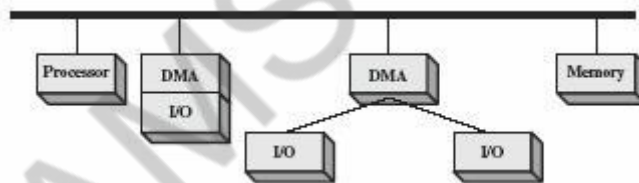
## DMA Configurations



Single bus – detached DMA module

Each transfer uses bus twice – I/O to DMA, DMA to memory

Processor suspended twice.

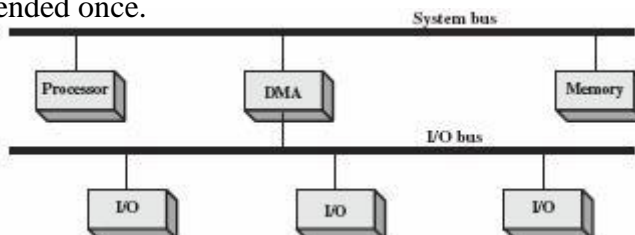


Single bus – integrated DMA module

Module may support more than one device

Each transfer uses bus once – DMA to memory

Processor suspended once.



Separate I/O bus

Bus supports all DMA enabled devices

Each transfer uses bus once – DMA to memory

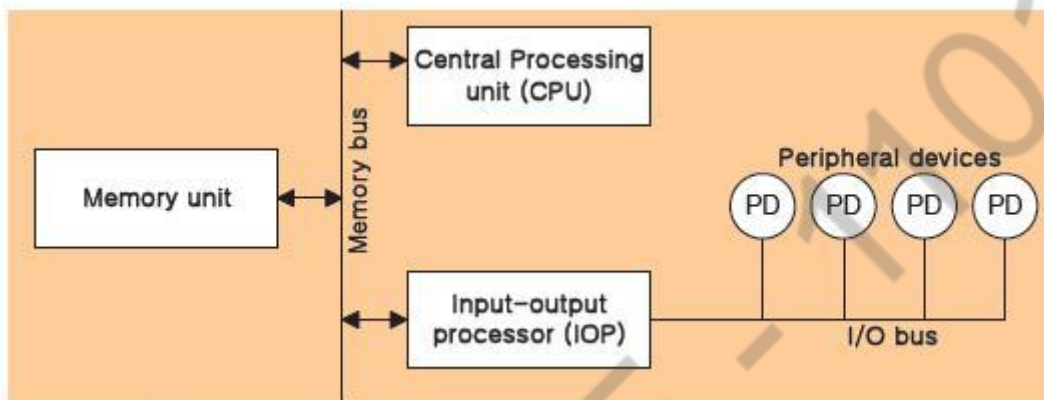
Processor suspended once.

## **9. INPUT-OUTPUT PROCESSOR (IOP)**

Communicate directly with all I/O devices

Fetch and execute its own instruction

IOP instructions are specifically designed to facilitate I/O transfer



### **Command**

Instruction *that are read form memory by an IOP*

Distinguish from instructions that are read by the CPU

Commands are prepared by experienced programmers and are stored in memory Command word  
= IOP program Memory

### **I/O Channels and Processors**

#### **The Evolution of the I/O Function**

1. Processor directly controls peripheral device
2. Addition of a controller or I/O module – programmed I/O
3. Same as 2 – interrupts added
4. I/O module direct access to memory using DMA
5. I/O module enhanced to become processor like – I/O channel
6. I/O module has local memory of its own – computer like – I/O processor
  - More and more the I/O function is performed without processor involvement.
  - The processor is increasingly relieved of I/O related tasks – improved performance.

### **Characteristics of I/O Channels**

- Extension of the DMA concept

- Ability to execute I/O instructions – special-purpose processor on I/O channel – complete control over I/O operations
- Processor does not execute I/O instructions itself – processor initiates I/O transfer by instructing the I/O channel to execute a program in memory
- Program specifies
  1. Device or devices
  2. Area or areas of memory
  3. Priority
  4. Error condition actions

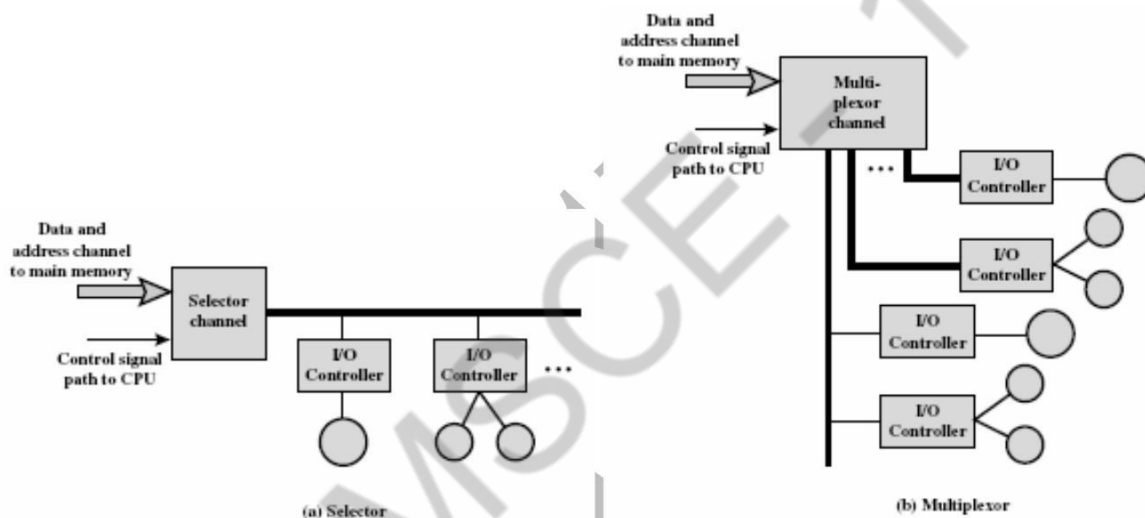
## Two type of I/O channels

- Selector channel
  - $\frac{3}{4}$  Controls multiple high-speed devices
  - $\frac{3}{4}$  Dedicated to the transfer of data with one of the devices
  - $\frac{3}{4}$  Each device handled by a controller, or I/O module
  - $\frac{3}{4}$  I/O channel controls these I/O controllers
- Multiplexor channel
 

Can handle multiple devices at the same time

Byte multiplexor – used for low-speed devices

Block multiplexor – interleaves blocks of data from several devices.

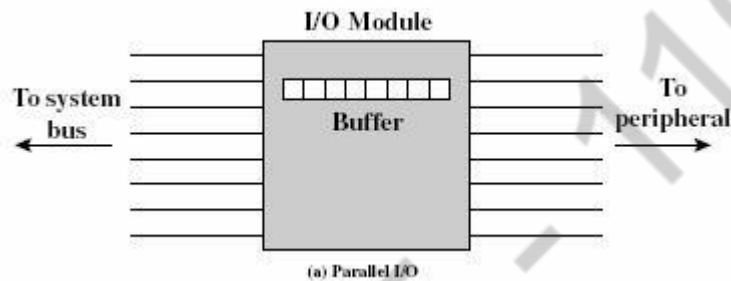
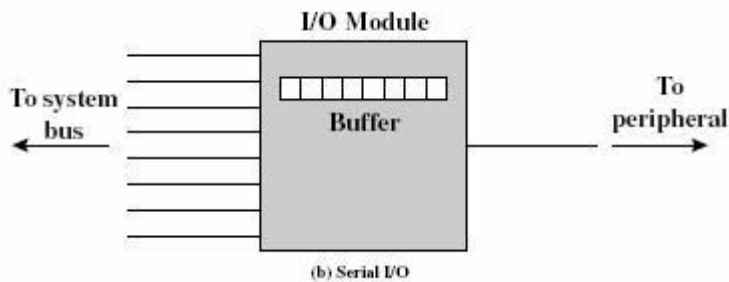




## The External Interface: FireWire and Infiniband

### Type of Interfaces

- Parallel interface – multiple bits transferred simultaneously
- Serial interface – bits transferred one at a time



### I/O module dialog for a write operation

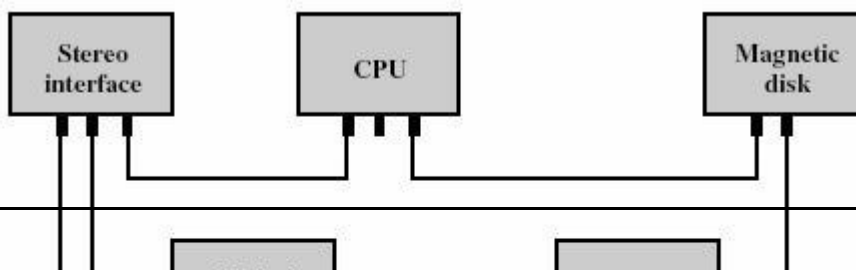
1. I/O module sends control signal – requesting permission to send data
2. Peripheral acknowledges the request
3. I/O module transfer data
4. Peripheral acknowledges receipt of data

### FireWire Serial Bus – IEEE 1394

- Very high speed serial bus
- Low cost
- Easy to implement
- Used with digital cameras, VCRs, and televisions

### FireWire Configurations

- Daisy chain
- 63 devices on a single port – 64 if you count the interface itself
- 1022 FireWire busses can be interconnected using bridges
- Hot plugging
- Automatic configuration
- No terminations
- Can be tree structured rather than strictly daisy chained



## FireWire three layer stack:Physical layer

Defines the transmission media that are permissible and the electrical and signaling characteristics of each 25 to 400 Mbps

Converts binary data to electrical signals

### Provides arbitration services

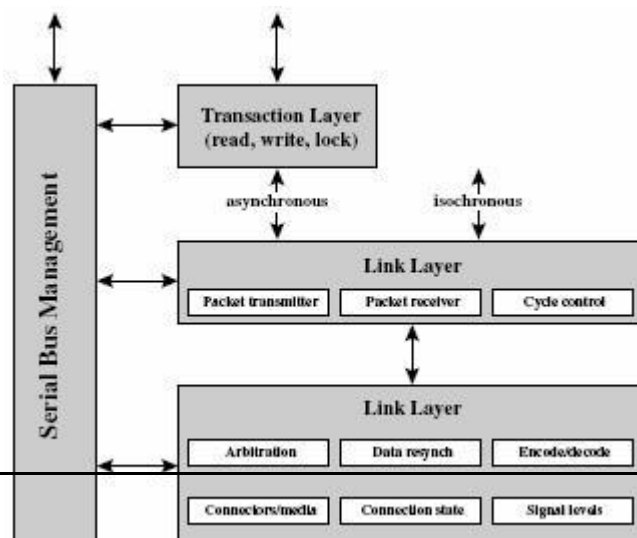
- ¾ Based on tree structure
- ¾ Root acts as arbiter
- ¾ First come first served
- ¾ Natural priority controls simultaneous requests – nearest root
- ¾ Fair arbitration
- ¾ Urgent arbitration

### Link layer

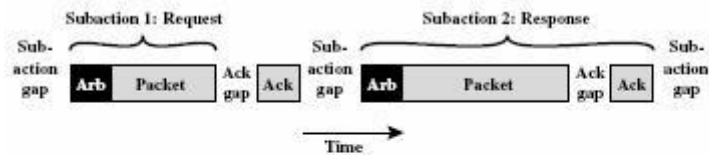
- Describes the transmission of data in the packets
- Asynchronous
  - o Variable amount of data and several bytes of transaction data transferred as a packet
  - o Uses an explicit address
  - o Acknowledgement returned
- Isochronous
  - o Variable amount of data in sequence of fixed sized packets at regular intervals
  - o Uses simplified addressing
  - o No acknowledgement

### Transaction layer

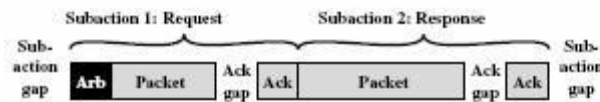
- Defines a request-response protocol that hides the lower-layer detail of FireWire from applications.



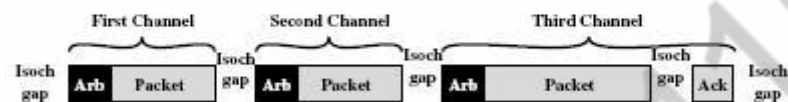
## FireWire Protocol Stack



(a) Example asynchronous subtraction



(b) Concatenated asynchronous subactions

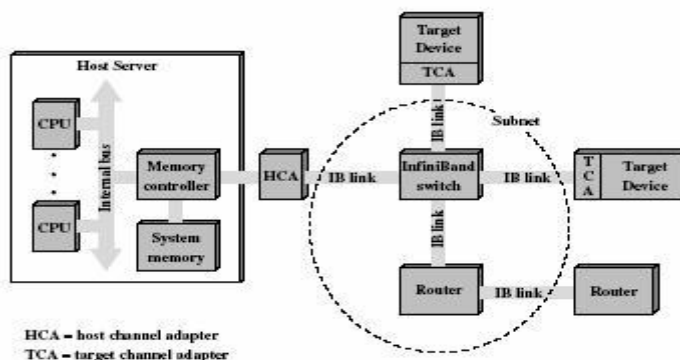


(c) Example isochronous subactions

## FireWire Subactions

## InfiniBand

- Recent I/O specification aimed at high-end server market
- First version released early 2001
- Standard for data flow between processors and intelligent I/O devices
- Intended to replace PCI bus in servers
- Greater capacity, increased expandability, enhanced flexibility
- Connect servers, remote storage, network devices to central fabric of switches and links
- Greater server density
- Independent nodes added as required
- I/O distance from server up to
  - o 17 meters using copper
  - o 300 meters using multimode optical fiber
  - o 10 kilometers using single-mode optical fiber
- Transmission rates up to 30 Gbps



## InfiniBand Operations

- 16 logical channels (virtual lanes) per physical link
- One lane for fabric management – all other lanes for data transport
- Data sent as a stream of packets
- Virtual lane temporarily dedicated to the transfer from one end node to another
- Switch maps traffic from incoming lane to outgoing lane

