CS8491 COMPUTER ARCHITECTURE

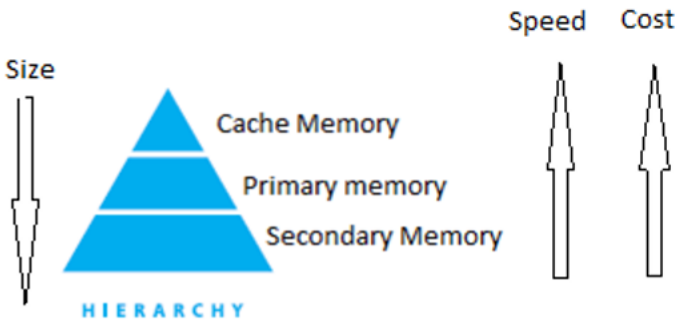
UNIT I BASIC STRUCTURE OF A COMPUTER SYSTEM

Functional Units – Basic Operational Concepts – Performance – Instructions: Language of the Computer – Operations, Operands – Instruction representation – Logical operations – decision making – MIPS Addressing.

EIGHT IDEAS

1. **Design for Moore's Law**
   - Moore's Law resulted from a 1965 prediction of such growth in IC capacity made by Gordon Moore, one of the founders of Intel.
   - The resources available per chip can easily double or quadruple between the start and finish of the project. i.e., it states that integrated circuit resources double every 18–24 months.
   - Computer architects must anticipate where the technology will be when the design finishes rather than design for where it starts.
   - We use an "up and to the right" Moore's Law graph to represent designing for rapid change.

2. **Use Abstraction to Simplify Design**
   - A major productivity technique for hardware and software is to use **abstractions** to represent the design at different levels of representation; lower-level details are hidden to offer a simpler model at higher levels.
3. **Make the Common Case Fast**
   - Making the **common case fast** will tend to enhance performance better than optimizing the rare case.
4. **Performance via Parallelism**
   - Computer architects have offered designs that get more performance by performing operations in parallel.
5. **Performance via Pipelining**
   - Pipeline icon is a sequence of pipes, with each section representing one stage of the pipeline.
6. **Performance via Prediction**
   - It can be faster on average to guess and start working rather than wait until you know for sure, assuming that the mechanism to recover from a misprediction is not too expensive and your prediction is relatively accurate.
7. **Hierarchy of Memories**
   - The fastest, smallest, and most expensive memory per bit at the top of the hierarchy and the slowest, largest, and cheapest per bit at the bottom.



8. Dependability via Redundancy

Computers not only need to be fast; they need to be dependable. Since any physical device can fail, we make systems **dependable** by including redundant components that can take over when a failure occurs *and* to help detect failures.

FUNCTIONAL UNITS

- A computer consists of five functionally independent main parts Input, Memory, Arithmetic and Logic unit (ALU), Output and Control unit.

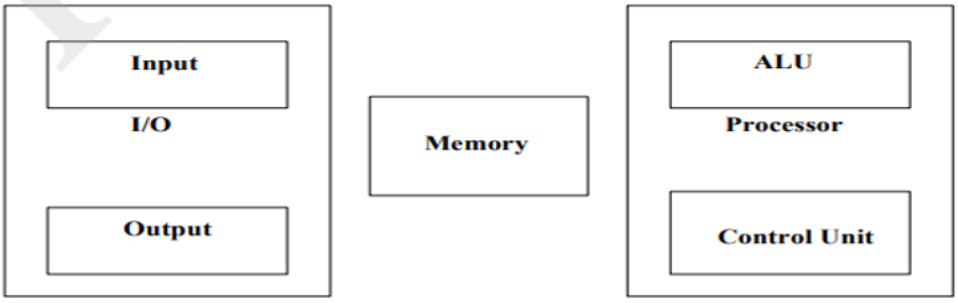
INPUT UNIT

- The source program/high level language program/coded information/simply data is fed to a computer through input devices keyboard is a most common type. Whenever a key is pressed, one corresponding word or number is translated into its equivalent binary code over a cable & fed either to memory or processor.
- Example:- Joysticks, trackballs, mouse, scanners etc are other input devices.

Mouse

- The original mouse was electromechanical and used a large ball that when rolled across a surface would cause an x and y counter to be incremented.
- Nowadays, the electromechanical mouse is replaced by optical mouse. The replacement of optical mouse, reduce cost and increase reliability.
- It includes Light emitting diode (LED) to provide lighting, a tiny black and white camera. The LED is underneath the mouse, the camera takes 1500 sample pictures/second.
- The sample pictures are sent to optical processor that compare the images and determine how far the mouse has moved.

CENTRAL PROCESSOR UNIT (CPU)



- CPU is also called processor. The active part of the computer, which contains the datapath and control units and which adds numbers, tests numbers, signals I/O devices to activate, and so on.

ALU or DATAPATH UNIT

- It performs the arithmetic operations.
- It is also called as brawn of the processor.
- The entire operation can be done with the help of registers.
- Registers are faster than memory.
- Registers can hold variables and intermediate results.
- Memory traffic is reduced, so program runs faster.

CONTROL UNIT

- It is also called as a brain of the processor.
- It fetches and analyses the instructions one-by-one and issue control signals to all other units to perform various operations.
- For a given instruction, the exact set of operations required is indicated by the control signals. The results of instructions are stored in memory.
- The component of the processor that commands the datapath, memory, and I/O devices according to the instructions of the program.

MEMORY UNIT

- Memory is nothing but a storage device. It stores the program and data.
- It is divided into 'n' number of cells. Each cell is capable of storing one bit information at a time.
- There are 2 classes of memory.1.Primary 2.Secondary.

Primary Memory

- It is made up of semiconductor material. So it is called Semiconductor memory
- Data storage capacity is less than secondary memory.
- Cost is too expensive than secondary memory.
- CPU can access data directly. Because it is an internal memory.
- Data accessing speed is very fast than secondary memory.
- Ex.RAM & ROM

RAM	ROM
Random Access Memory	Read Only Memory
Volatile memory	Non volatile memory
Data lost when the power turns off and that is used to hold data and program while they are running.	It retains data even in the absence of a power source and that is store programs between runs.
Temporary storage medium	Permanent storage medium
User perform both read and write operation	User can perform only read operation

RAM: There are two types of memory available namely, 1. SRAM 2. DRAM

SRAM	DRAM
Static RAM	Dynamic RAM
Information is stored in 1 bit cell called Flip Flop.	Information is represented as charge across a capacitor
Information will be available as long as power is available	It retains data for few ms based on the charge of capacitor even in the absence of a power
No refreshing is needed	Refreshing is needed

Less packaging density	High packaging density
More complex Hardware	Less complex hardware
More expensive	Less expensive
No random access	Random access is possible
Access time 10 ns	Access time 50 ns

Cache Memory: A small, fast memory that acts as a buffer for a slower, larger memory.

Secondary memory

- Secondary memory (Nonvolatile storage) is a form of storage that retains data
- even in the absence of a power source and that is used to store the programs between runs.
- It is made up of magnetic material. So it is called magnetic memory.
- Data storage capacity is high than primary memory.
- Cost is too low than primary memory.
- CPU cannot access data directly. Because it is an external memory.
- Data accessing speed is very slow than primary memory.
- Ex. Magnetic disk, Hard Disk, CD, DVD, Floppy Disk

Magnetic Disk

- It consists of a collection of platters, which rotate on a spindle at 5400 revolution/min.
- The metal platters are covered with magnetic recording material on both sides.
- Also called hard disk. A form of nonvolatile secondary memory composed of rotating platters coated with a magnetic recording material.
- Because they are rotating mechanical devices, access times are about 5 to 20 milliseconds and cost per gigabyte in 2012 was $0.05 to $0.10.

Optical Disk: Include both Compact Disk (CD) and Digital Video Disk(DVD).

Read-Only CD/DVD

- Data is recorded in a spiral fashion, with individual bits being recorded by burning small pit.
- The disk is read by shining a laser at the CD surface and determining by examining the reflected light whether there is a pit or flat surface.

Rewritable CD/DVD

- Use different recording surface that as a crystal line, reflective materials, pits are formed that are not reflective.

Erase CD/DVD

- The surface is heated and cooled slowly, allowing an annealing process to restore the surface recording layer to its crystalline structure.

Flash Memory

- A nonvolatile semiconductor memory. It is cheaper and slower than DRAM but more expensive per bit and faster than magnetic disks. Access times are about 5 to 50 microseconds and cost per gigabyte in 2012 was $0.75 to $1.00.

OUTPUT UNIT

- A mechanism that conveys the result of a computation to a user, such as a display, or to another computer.

Liquid Crystal Display

- A display technology using a thin layer of liquid polymers that can be used to transmit or block light according to whether a charge is applied.
- All laptops and desktop computers use Liquid Crystal Display (LCD) to get a thin, low-power display. A tiny transistor switch at each pixel to control current and make sharper images.
- The image is composed of a matrix of picture elements or pixels, which can be represented as a matrix of bits called **bitmap.**
- Depending on the size of the screen and the resolution, the display matrix ranges in size from 640*480 to 2560*1600.
- A red, green, blue (RGB) associated with each dot on the display determines the intensity of the three color components in the final image.
- **Pixel:** The smallest individual picture element. Screens are composed of hundreds of thousands to millions of pixels, organized in a matrix.

BASIC OPERATIONAL CONCEPTS

To perform a given task an appropriate program consisting of a list of instructions is stored in the memory. Individual instructions are brought from the memory into the processor, which executes the specified operations. Data to be stored are also stored in the memory.

Examples: Add LOCA, R0

This instruction adds the operand at memory location LOCA, to operand in register R0 & places the sum into register. This instruction requires the performance of several steps,

1. First the instruction is fetched from the memory into the processor
2. The operand at LOCA is fetched and added to the contents of R0
3. Finally the resulting sum is stored in the register R0

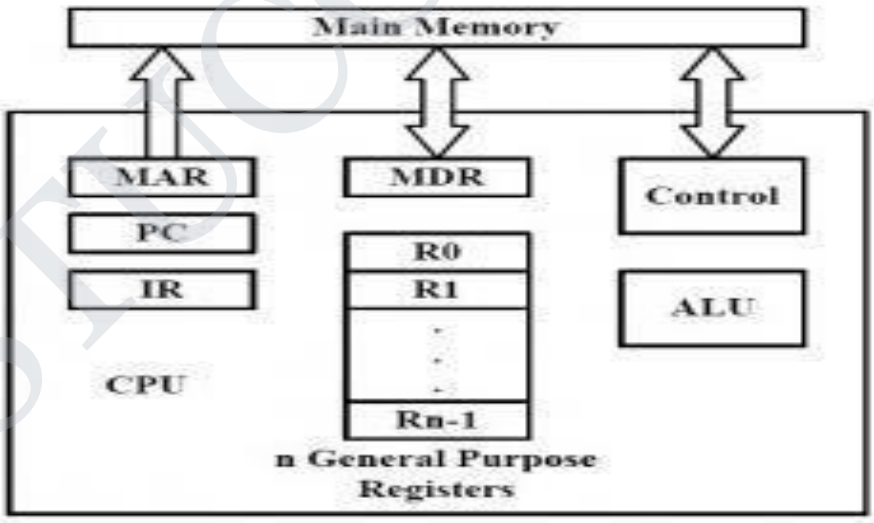


Fig: Connections between the processor and the memory

The fig shows how memory & the processor can be connected. In addition to the ALU & the control circuitry, the processor contains a number of registers used for several different purposes.

Instruction Register (IR)

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

Program Counter PC

- This is another specialized register that keeps track of execution of a program.
- It contains the memory address of the next instruction to be fetched and executed. Besides IR and PC, there are n-general purpose registers R0 through Rn-1. MAR PC IR MEMORY MDR R0 R1 …

The other two registers which facilitate communication with memory are:

1. MAR – (Memory Address Register):- It holds the address of the location to be accessed.

2. MDR – (Memory Data Register):- It contains the data to be written into or read out of the address location.

Operating steps are

1. Programs reside in the memory & usually get these through the I/P unit.
2. Execution of the program starts when the PC is set to point at the first instruction of the program.
3. Contents of PC are transferred to MAR and a Read Control Signal is sent to the memory.
4. After the time required to access the memory elapses, the address word is read out of the memory and loaded into the MDR.
5. Now contents of MDR are transferred to the IR & now the instruction is ready to be decoded and executed.
6. If the instruction involves an operation by the ALU, it is necessary to obtain the required operands.
7. An operand in the memory is fetched by sending its address to MAR & Initiating a read cycle.
8. When the operand has been read from the memory to the MDR, it is transferred from MDR to the ALU.
9. After one or two such repeated cycles, the ALU can perform the desired operation.
10. If the result of this operation is to be stored in the memory, the result is sent to MDR.
11. Address of location where the result is stored is sent to MAR & a write cycle is initiated.
12. The contents of PC are incremented so that PC points to the next instruction that is to be executed.

PERFORMANCE

The machine (or CPU) is said to be faster or has better performance running this program if the total execution time is shorter. When trying to choose among different computers, performance is an important attribute.

1. Throughput and Response Time

Response time

- Response time also called execution time. The total time required for the computer to complete a task, including disk accesses, memory accesses, I/O activities, operating system overhead, CPU execution time, and so on.

Throughput or bandwidth

- The total amount of work done in a given time. Decreasing response time almost always improves throughput.
- To maximize performance, we want to minimize response time or execution time for some task. Thus, we can relate performance and execution time for a computer X:

$$\text{Performance}_X = \frac{1}{\text{Execution time}_X}$$

This means that for two computers X and Y, if the performance of X is greater than the performance of Y, we have

$$\text{Performance}_X > \text{Performance}_Y$$

$$\frac{1}{\text{Execution time}_X} > \frac{1}{\text{Execution time}_Y}$$

$$\text{Execution time}_Y > \text{Execution time}_X$$

That is, the execution time on Y is longer than that on X, if X is faster than Y.

We will use the phrase "X is *n* times faster than Y"—or equivalently "X is *n* times as fast as Y"—to mean

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = n$$

If X is *n* times as fast as Y, then the execution time on Y is *n* times as long as it is on X:

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

Relative Performance

If computer A runs a program in 10 seconds and computer B runs the same program in 15 seconds, how much faster is A than B?

We know that A is *n* times as fast as B if

$$\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = n$$

Thus the performance ratio is

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

and A is therefore 1.5 times as fast as B.

In the above example, we could also say that computer B is 1.5 times slower than computer A

2. **Measuring Performance**

CPU execution time: Also called **CPU time**. The actual time the CPU spends computing for a specific task.

User CPU time: The CPU time spent in a program itself.

System CPU time: The CPU time spent in the operating system performing tasks on behalf of the program.

Clock cycle: Also called **tick**, **clock tick**, **clock period**, **clock**, or **cycle**. The time for one clock period, usually of the processor clock, which runs at a constant rate.

Clock period: This is the length of each clock cycle.

Clock rate*:* This is the inverse of the clock period.

3. **CPU Performance and Its Factors**
   - A simple formula relates the most basic metrics (clock cycles and clock cycle time) to CPU time.
   - This formula makes it clear that the hardware designer can improve performance by reducing the number of clock cycles required for a program or the length of the clock cycle.

$$\text{CPU execution time for a program} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

Alternatively, because clock rate and clock cycle time are inverses,

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

Example

Improving Performance

Our favorite program runs in 10 seconds on computer A, which has a 2 GHz clock. We are trying to help a computer designer build a computer, B, which will run this program in 6 seconds. The designer has determined that a substantial increase in the clock rate is possible, but this increase will affect the rest of the CPU design, causing computer B to require 1.2 times as many clock cycles as computer A for this program. What clock rate should we tell the designer to target?

Let's first find the number of clock cycles required for the program on A:

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

$$10 \text{ seconds} = \frac{\text{CPU clock cycles}_A}{2 \times 10^9 \frac{\text{cycles}}{\text{second}}}$$

$$\text{CPU clock cycles}_A = 10 \text{ seconds} \times 2 \times 10^9 \frac{\text{cycles}}{\text{second}} = 20 \times 10^9 \text{ cycles}$$

CPU time for B can be found using this equation:

$$\text{CPU time}_B = \frac{1.2 \times \text{CPU clock cycles}_A}{\text{Clock rate}_B}$$

$$6 \text{ seconds} = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{\text{Clock rate}_B}$$

$$\text{Clock rate}_B = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{6 \text{ seconds}} = \frac{0.2 \times 20 \times 10^9 \text{ cycles}}{\text{second}} = \frac{4 \times 10^9 \text{ cycles}}{\text{second}} = 4 \text{ GHz}$$

To run the program in 6 seconds, B must have twice the clock rate of A.

4. **Instruction Performance**

- Execution time is that it equals the number of instructions executed multiplied by the average time per instruction.
- Therefore, the number of clock cycles required for a program can be written as

$$\text{CPU clock cycles} = \text{Instructions for a program} \times \frac{\text{Average clock cycles}}{\text{per instruction}}$$

Clock cycles per Instruction

- The term clock cycles per instruction, which is the average number of clock cycles each instruction takes to execute, is often abbreviated as CPI.
- Since different instructions may take different amounts of time depending on what they do, CPI is an average of all the instructions executed in the program.

Example

Suppose we have two implementations of the same instruction set architecture.Computer A has a clock cycle time of 250 ps 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 sameprogram. Which computer is faster for this program and by how much?

We know that each computer executes the same number of instructions for the program; let's call this number I. First, find the number of processor clock cycles for each computer:

$$\text{CPU clock cycles}_A = I \times 2.0$$
$$\text{CPU clock cycles}_B = I \times 1.2$$

Now we can compute the CPU time for each computer:

$$\text{CPU time}_A = \text{CPU clock cycles}_A \times \text{Clock cycle time}$$
$$= I \times 2.0 \times 250 \text{ ps} = 500 \times I \text{ ps}$$

Likewise, for B:

$$\text{CPU time}_B = I \times 1.2 \times 500 \text{ ps} = 600 \times I \text{ ps}$$

Clearly, computer A is faster. The amount faster is given by the ratio of the execution times:

$$\frac{\text{CPU performance}_A}{\text{CPU performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{600 \times I \text{ ps}}{500 \times I \text{ ps}} = 1.2$$

We can conclude that computer A is 1.2 times as fast as computer B for this program.

5. **The Classic CPU Performance Equation**

We can now write this basic performance equation in terms of instruction count (the number of instructions executed by the program), CPI, and clock cycle time:

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

or, since the clock rate is the inverse of clock cycle time:

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}$$

$$\text{Time} = \text{Seconds/Program} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

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 number of clock cycles per instruction
Clock cycle time	Seconds per clock cycle

Example

Comparing Code Segments

A compiler designer is trying to decide between two code sequences for a particular computer. The hardware designers have supplied the following facts:

	CPI for each instruction class		
	A	B	C
CPI	1	2	3

For a particular high-level language statement, the compiler writer is considering two code sequences that require the following instruction counts:

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

Which code sequence executes the most instructions? Which will be faster? What is the CPI for each sequence?

Sequence 1 executes 2 + 1 + 2 = 5 instructions. Sequence 2 executes 4 + 1 + 1 = 6 instructions. Therefore, sequence 1 executes fewer instructions.

We can use the equation for CPU clock cycles based on instruction count and CPI to find the total number of clock cycles for each sequence:

$$\text{CPU clock cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{C}_i)$$

This yields

$$\text{CPU clock cycles}_1 = (2 \times 1) + (1 \times 2) + (2 \times 3) = 2 + 2 + 6 = 10 \text{ cycles}$$

$$\text{CPU clock cycles}_2 = (4 \times 1) + (1 \times 2) + (1 \times 3) = 4 + 2 + 3 = 9 \text{ cycles}$$

So code sequence 2 is faster, even though it executes one extra instruction. Since code sequence 2 takes fewer overall clock cycles but has more instructions, it must have a lower CPI. The CPI values can be computed by

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

$$\text{CPI}_1 = \frac{\text{CPU clock cycles}_1}{\text{Instruction count}_1} = \frac{10}{5} = 2.0$$

$$\text{CPI}_2 = \frac{\text{CPU clock cycles}_2}{\text{Instruction count}_2} = \frac{9}{6} = 1.5$$

Amdahl's Law

A rule stating that the performance enhancement possible with a given improvement is limited by the amount that the improved feature is used.

$$\text{Execution time after improvement} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$

Million Instructions per Second (MIPS)

- A measurement of program execution speed based on the number of millions of instructions.
- MIPS are computed as the instruction count divided by the product of the execution time. MIPS is easy to understand, and faster computers.

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}$$

- There are three problems with using MIPS as a measure for comparing computers.
- First, we cannot compare computers with different instruction sets using MIPS, since the instruction counts will certainly differ.
- Second, MIPS varies between programs on the same computer; thus, a computer cannot have a single MIPS rating.
- For example, by substituting for execution time, we see the relationship between MIPS, clock rate, and CPI:

$$\text{MIPS} = \frac{\text{Instruction count}}{\frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$

- Finally, and most importantly, if a new program executes more instructions but each instruction is faster, MIPS can vary independently from performance.

INSTRUCTION

The words of a computer‘s language are called instructions, and its vocabulary is called an instruction set. Instruction performs one of the following operations

- Data transfer between register and memory.
- ALU operation.
- Program control and sequencing.
- I/O transfer.

Stored program concept

The idea that instructions and data of many types can be stored in memory as numbers, leading to the stored program computer.

OPERATIONS OF THE COMPUTER HARDWARE

1. Arithmetic Operations

Add and Subtract instruction use three Operands - Two Sources and one Destination.

Example add $t0, $s1, $s2

2. Data Transfer Operations

These operations help in moving the data between memory and registers.

Example: 1

C code:

```
g = h + A[8];
```

- g in $s1, h in $s2, base address of A in $s3

Compiled MIPS code:

- Index 8 requires offset of 32
  - 4 bytes per word

```
lw  $t0, 32($s3)     # load word
add $s1, $s2, $t0
```

offset — base register

Example: 2

C code:

```
A[12] = h + A[8];
```

- h in $s2, base address of A in $s3

Compiled MIPS code:

- Index 8 requires offset of 32

```
lw  $t0, 32($s3)     # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)     # store word
```

3. Logical Operations

Logical Instructions are used for bitwise manipulation. It is useful for extracting and inserting groups of bits in a word

Operation	C	MIPS
Shift left	<<	sll
Shift right	>>	srl
Bitwise AND	&	and, andi
Bitwise OR	|	or, ori
Bitwise NOT	~	nor

4. Conditional Operations

1. Branch to a labeled instruction if a condition is true. Otherwise, continue sequentially.

   beq rs, rt, L1
   if (rs == rt) branch to instruction labeled L1;
   bne rs, rt, L1
   if (rs != rt) branch to instruction labeled L1;
   j L1

2. Unconditional jump to instruction labeled L1. Set result to 1 if a condition is true. Otherwise, set to 0.

   slt rd, rs, rt
   if (rs < rt) rd = 1; else rd = 0;
   slti rt, rs, constant
   if (rs < constant) rt = 1; else rt = 0;

3. Use in combination with beq, bne

   slt $t0, $s1, $s2 # if ($s1 < $s2)
   bne $t0, $zero, L # branch to L

Jump Instructions

Jump Address Table: Also called **jump table**. A table of addresses of alternative instruction sequences.

4. Procedure call: jump and link.

Jump-and-Link Instruction: An instruction that jumps to an address and simultaneously saves the address of the following instruction in a register($rain MIPS).

jal ProcedureLabel
Address of following instruction put in $ra.
Jumps to target address.

5. Procedure return: jump register

Return address: A link to the calling site that allows a procedure to return to the proper address in MIPS it is stored in register $ra.

jr $ra
Copies $ra to program counter.
Can also be used for computed jumps.
e.g., for case/switch statements.

Procedure: A stored subroutine that performs a specific task based on the parameters with which it is provided.

Caller: The program that instigates a procedure and provides the necessary parameter values.

Callee: A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller.

Program counter (PC): The register containing the address of the instruction in the program being executed.

Stack: A data structure for spilling registers organized as a last-in first-out queue.

Stack pointer: A valued noting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found. In MIPS, it is register $sp.

Push: Add element to stack.

Pop: Remove element from stack.

Global pointer: The register that is reserved to point to the static area.

OPERANDS OF THE COMPUTER HARDWARE

1. Register Operands

- Arithmetic instructions use register operands and MIPS has a 32 × 32-bit register file
- Mainly used for frequently accessed data
- Register numbered 0 to 31 and 32-bit data called a ―word
- $t0, $t1, …, $t9 for temporary values
- $s0, $s1, …, $s7 for saved variables

Word: The natural unit of access in a computer, usually a group of 32 bits; corresponds to the size of a register in the MIPS architecture.

Compiling a C Assignment Using Registers

Example: f = (g + h) – (i + j); The variables f, g, h, i, and j are assigned to the registers $s0, $s1, $s2, $s3, and $s4, respectively. What is the compiled MIPS code?

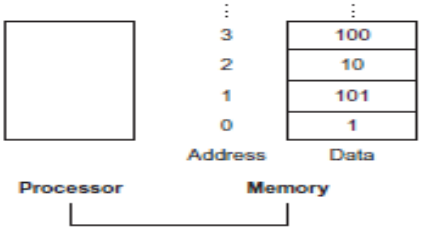
MIPS code

```
add $t0,$s1,$s2 # register $t0 contains g + h
add $t1,$s3,$s4 # register $t1 contains i + j
sub $s0,$t0,$t1 # f gets $t0 – $t1, which is (g + h)–(i + j)
```

2. Memory Operands

Data transfer instruction: A command that moves data between memory and registers.

Address: A value used to delineate the location of a specific data element within a memory array. For example, in the following figure, the address of the third data element is 2, and the value of Memory [2] is 10.



2.1 Load:

- The data transfer instruction that copies data from memory to a register is traditionally called load.
- The format of the load instruction is the name of the operation followed by the register to be loaded, then a constant and register used to access memory.
- The sum of the constant portion of the instruction and the contents of the second register forms the memory address.
- The actual MIPS name for this instruction is lw, standing for load word.

Compiling an Assignment When an Operand Is in Memory

Example: Let's assume that A is an array of 100 words and that the compiler has associated the variables g and h with the registers $s1 and $s2 as before. Let's also assume that the starting address, or base address, of the array is in$s3. Compile this C assignment statement:

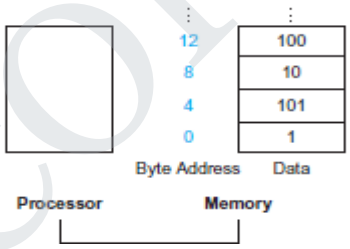
g = h + A[8];

Solution:

```
lw $t0,32($s3)      # Temporary reg $t0 gets A[8]
add $s1,$s2,$t0     # g = h + A[8]
```

The constant in a data transfer instruction (8) is called the off set, and the register added to form the address ($s3) is called the base register.

Alignment restriction: A requirement that data be aligned in memory on natural boundaries.



Little Endian Address: A lower byte address of the word is specified in least significant byte of the word is known as Little endian address.

Big Endian Address: A lower byte address of the word is specified in most significant byte of the word is known as big endian address.

2.2 Store:

- The instruction complementary to load is traditionally called store; it copies data from a register to memory. The format of a store is similar to that of a load.
- The actual MIPS name is sw, standing for store word.

Example:

Assume variable h is associated with register $s2 and the base address of the array A is in $s3. What is the MIPS assembly code for the C assignment statement below?

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

Solution:

```
lw $t0,32($s3)      # Temporary reg $t0 gets A[8]
add $t0,$s2,$t0     # Temporary reg $t0 gets h + A[8]
```

The final instruction stores the sum into A[12], using 48 (4×12) as the offset and register $s3 as the base register.

```
sw $t0,48($s3)      # Stores h + A[8] back into A[12]
```

3. Constant or Immediate Operands

- The constants would have been placed in memory when the program was loaded. For example, to add the constant 4 to register $s3.
- we could use the code

```
lw $t0, AddrConstant4($s1)          # $t0 = constant 4
add $s3,$s3,$t0                     # $s3 = $s3 + $t0 ($t0 == 4)
```

- This quick add instruction with one constant operand is called add immediate or addi. To add 4 to register $s3,We just write

```
addi $s3,$s3,4          # $s3 = $s3 + 4
```

- No subtract immediate instruction, Just use a negative constant

```
addi $s2,$s3,-1
```

REPRESENTING INSTRUCTIONS IN THE COMPUTER

- Since registers are referred to in instructions, there must be a convention to map register names into numbers.

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

Register numbers

$t0 – $t7 are reg‘s 8 – 15

$t8 – $t9 are reg‘s 24 – 25

$s0 – $s7 are reg‘s 16 – 23

Instruction format: A form of representation of an instruction composed of fields of binary numbers.

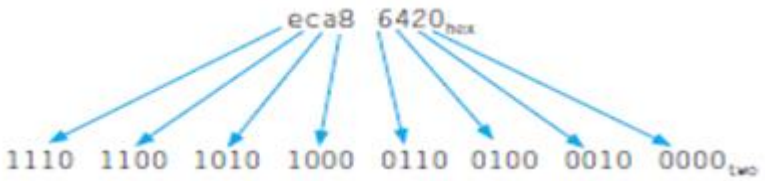
Machine language: Binary representation used for communication within a computer system.

Hexadecimal: Numbers in base 16. The following table shows Hexadecimal to binary conversion.

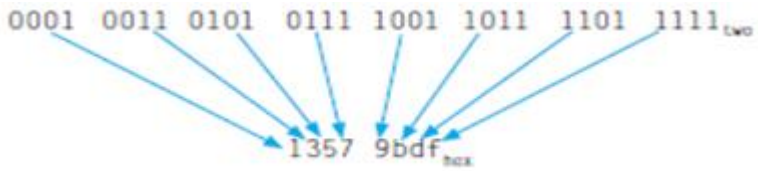
Example:

Convert the following hexadecimal and binary numbers into the other base:

1. **eca8 6420$_{hex}$**



2. $0001\ 0011\ 0101\ 0111\ 1001\ 1011\ 1101\ 1111_{two}$



Format Types:

1. R-Format
2. I-Format
3. J-Format

1. R-Type Format:

Opcode: The field that denotes the operation and format of an instruction.

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Instruction Fields:

op: operation code (opcode)
rs: first source register number
rt: second source register number
rd: destination register number
shamt: shift amount (00000 for now)
funct: function code (extends opcode)

Example:

```
add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

2. I-Type Format:

A second type of instruction format is called I-type (for immediate)or I-format and is used by the immediate and data transfer instructions. The fields of I-format are

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

the hardware knows whether to treat the last half of the instruction as three fields(R-type) or as a single field (I-type).

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

Example:

Translating MIPS Assembly Language into Machine Language

We can now take an example all the way from what the programmer writes to what the computer executes. If `$t1` has the base of the array `A` and `$s2` corresponds to `h`, the assignment statement

```
A[300] = h + A[300];
```

is compiled into

```
lw   $t0,1200($t1) # Temporary reg $t0 gets A[300]
add  $t0,$s2,$t0   # Temporary reg $t0 gets h + A[300]
sw   $t0,1200($t1) # Stores h + A[300] back into A[300]
```

What is the MIPS machine language code for these three instructions?

Solution:

Decimal Form:

Op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

Binary Form:

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

- The two MIPS instruction formats so far are R and I.
- The first 16 bits are the same: both contain an op field, giving the base operation; an rs field, giving one of the sources; and the rt field, which specifies the other source operand, except for load word, where it specifies the destination register.
- R-format divides the last 16 bits into an rd field, specifying the destination register; the shamt field, and the funct field, which specifies the specific operation of R-format instructions.
- I-format combines the last 16 bits into a single address field.

3. J-Format:

j target- J-type is short for "jump type". The format of an J-type instruction looks like:

Opcode	Address

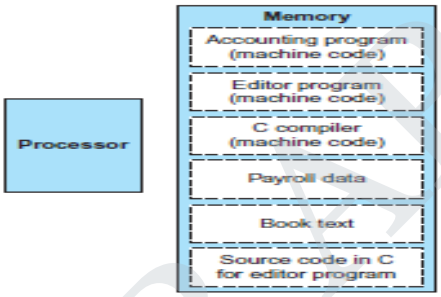
The semantics of the j instruction (j means jump) are:

PC <- PC31-28 IR25-0 00

where PC is the program counter, which stores the current address of the instruction being executed. You update the PC by using the upper 4 bits of the program counter, followed by the 26 bits of the target (which is the lower 26 bits of the instruction register), followed by two 0's, which creates a 32 bit address.

The stored-program concept:

- Stored programs allow a computer that performs by loading memory with programs and data and to begin executing at a given location in memory.



What MIPS instruction does this represent? Choose from one of the four options below.

op	rs	rt	rd	shamt	funct
0	8	9	10	0	34

1. sub $t0, $t1, $t2
2. add $t2, $t0, $t1
3. sub $t2, $t1, $t0
4. **sub $t2, $t0, $t1**

LOGICAL OPERATIONS

The packing and unpacking of bits into words. These instructions are called logical operations.

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

1. SHIFT:

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

shamt: how many positions to shift

Shift left logical

Shift left and fill with 0 bits

sll by i bits multiplies by 2i

Shift right logical

Shift right and fill with 0 bits

srl by i bits divides by 2i (unsigned only)

- For example, if register $s0contained

  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_{two} = 9_{ten}$

- and the instruction to shift left by 4 was executed, the new value would be:

  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_{two} = 144_{ten}$

- The dual of a shift left is a shift right. The actual name of the two MIPS shift instructions are called shift left logical (sll) and shift right logical (srl).
- The following instruction performs the operation above, assuming that the original value was in register $s0 and the result should go in register $t2:

  sll $t2,$s0,4 # reg $t2 = reg $s0 << 4 bits

- The encoding of sll is 0 in both the op and funct fields, rd contains 10 (register $t2), rt contains 16 (register $s0), and shamt contains 4. The rs field is unused and thus is set to 0.Shifting left by i bits gives the same result as multiplying by 2i.

2. AND

- A logical bit by-bit operation with two operands that calculates a 1 only if there is a 1 in both operands.

  Ex: and $t0, $t1, $t2

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0000 1100 0000 0000

3. OR

- A logical bit-by bit operation with two operands that calculates a 1 if there is a 1 in either operand.

  Ex: or $t0,$t1,$t2

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0011 1101 1100 0000

4. NOT

- A logical bit-by bit operation with one operand that inverts the bits; that is, it replaces every 1 with a 0, and every 0 with a 1.

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

5. NOR

- A logical bit-by bit operation with two operands that calculates the NOT of the OR of the two operands. That is, it calculates a 1 only if there is a 0 in both operands. If one operand is zero, then it is equivalent to NOT:

  A NOR 0= NOT (A OR 0) = NOT (A).

- If the register $t1 is unchanged from the preceding example and register $t3has the value 0, the result of the MIPS instruction \

  nor $t0,$t1,$t3 # reg $t0 = ~ (reg $t1 | reg $t3)

- register $t0

  $1111\ 1111\ 1111\ 1111\ 1100\ 0011\ 1111\ 1111_{two}$

INSTRUCTIONS FOR MAKING DECISIONS –CONTROL OPERATIONS

- It is ability to make decisions. Based on the input data and the values created during computation.
- The first instruction is **beq register1, register2, L1**
- This instruction means go to the statement labeled L1 if the value in register1 equals the value in register2. The mnemonic beq stands for branch if equal.
- The second instruction is **bne register1, register2, L1**
- It means go to the statement labeled L1 if the value in register1 does not equal the value in register2. 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 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?

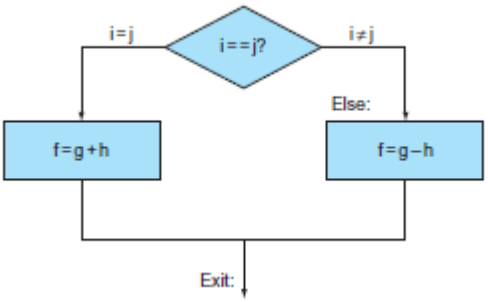
  if (i == j) f = g + h; else f = g – h;

MIPS Code

```
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:
```

Conditional Branch

- An instruction that requires the comparison of two values and that allows for a subsequent transfer of control to a new address in the program based on the outcome of the comparison.



Loops

- Decisions are important both for choosing between two alternatives—found in if statements and for iterating a computation found in loops. The same assembly instructions are the building blocks for both cases.

Compiling a while Loop in C

- Here is a traditional 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?

MIPS Code:

```
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:
```

Basic Block

- A sequence of instructions without branches (except possibly at the end) and without branch targets or branch labels (except possibly at the beginning).

Set on Less Than Instruction

- Compares two registers and sets a third register to 1 if the first is less than the second; otherwise, it is set to 0.
- The MIPS instruction is called set on less than, or slt. For example,

```
slt $t0, $s3, $s4            # $t0 = 1 if $s3 < $s4
```

- It means that register $t0 is set to 1 if the value in register $s3 is less than the value in register $s4; otherwise, register $t0 is set to 0.

- Constant operands are popular in comparisons, so there is an immediate version of the set on less than instruction. To test if register $s2 is less than the constant10, we can just write

  slti $t0,$s2,10 # $t0 = 1 if $s2 < 10

Signed versus Unsigned Comparison

- Suppose register $s0 has the binary number

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$ and that register $s1 has the binary number $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two}$

  What the values are of registers $t0 and $t1 after these two instructions?

Solution

slt $t0, $s0, $s1 # signed comparison

sltu $t1, $s0, $s1 # unsigned comparison

- The value in register $s0 represents -1_{ten} if it is an integer and $4,294,967,295_{ten}$ if it is an unsigned integer.
- The value in register $s1 represents 1_{ten} in either case. Then register $t0 has the value 1, since $-1_{ten} < 1_{ten}$, and register $t1 has the value 0, since $4,294,967,295_{ten} > 1_{ten}$.

MIPS ADDRESSING MODE SUMMARY

Multiple forms of addressing are generically called **addressing modes.** The following diagram shows how operands are identified for each addressing mode.

1**. Immediate addressing**, where the operand is a constant within the instruction itself

2. **Register addressing**, where the operand is a register

3. **Base or displacement addressing,** where the operand is at the memory location whose address is the sum of a register and a constant in the instruction

4. **PC-relative addressing**, where the branch address is the sum of the PC and a constant in the instruction

5. **Pseudo direct addressing**, where the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC

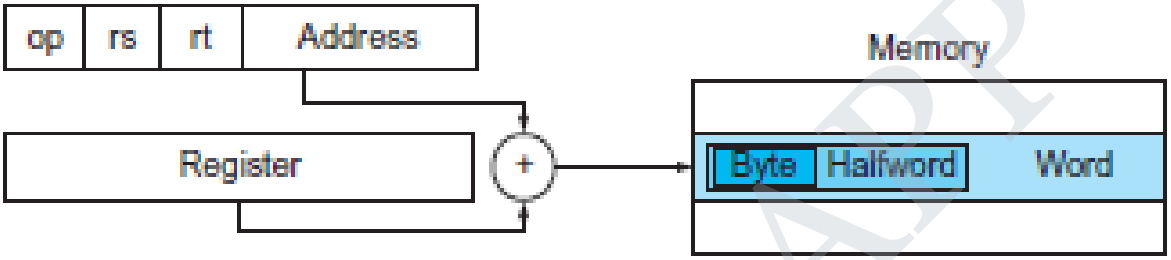
The following diagram shows all the MIPS instruction formats.

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

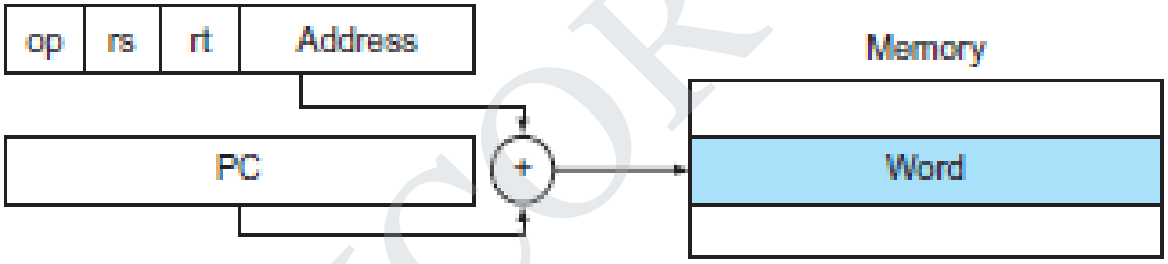
1. Immediate addressing

2. Register addressing

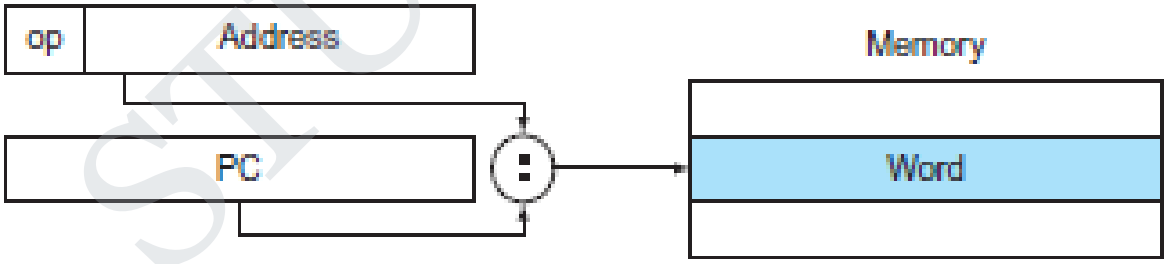
3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing

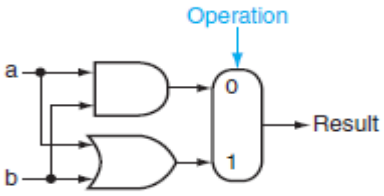


UNIT – II ARITHMETIC FOR COMPUTERS

Addition and Subtraction – Multiplication – Division – Floating Point Representation – Floating Point Operations – Subword Parallelism

ALU:

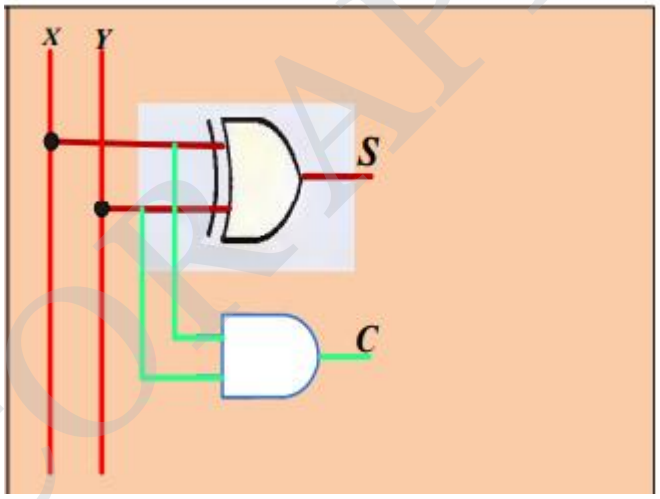
- Arithmetic Logic Unit (ALU). Hardware that performs addition, subtraction, and usually logical operations such as AND and OR.
- The arithmetic logic unit (ALU) is the brawn of the computer.



Half Adder

Half Adder: is a combinational circuit that performs the addition of two bits, this circuit needs two binary inputs and two binary outputs.

Inputs		Outputs	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0
Truth table			

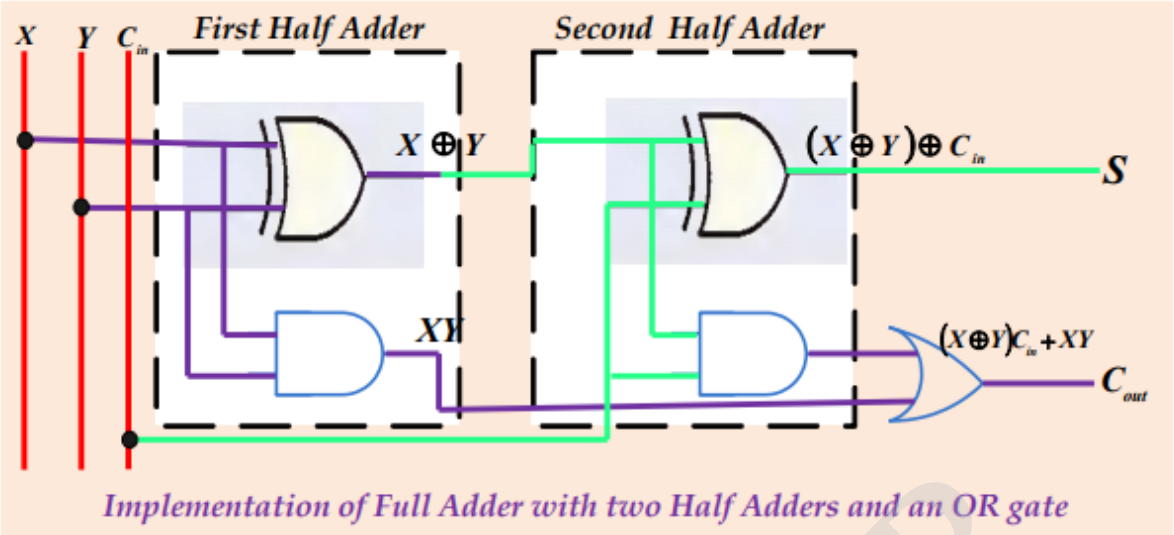


Where S is the sum and C is the carry.

$$\begin{cases} S = X \oplus Y \\ C = XY \end{cases} \quad 2 \} \text{ (Using XOR and AND Gates)}$$

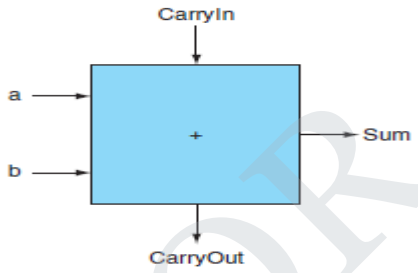
Full Adder

- Full Adder is a combinational circuit that performs the addition of three bits (two significant bits and previous carry).
- It consists of three inputs and two outputs, two inputs are the bits to be added, the third input represents the carry form the previous position.
- The full adder is usually a component in a cascade of adders, which add 8, 16, etc, binary numbers.
- An adder must have two inputs for the operands and a single-bit output for the sum and the second output to pass on the carry, called CarryOut.
- The CarryOut from the neighbor adder must be included as an input, we need a third input. This input is called CarryIn.



Implementation of Full Adder with two Half Adders and an OR gate

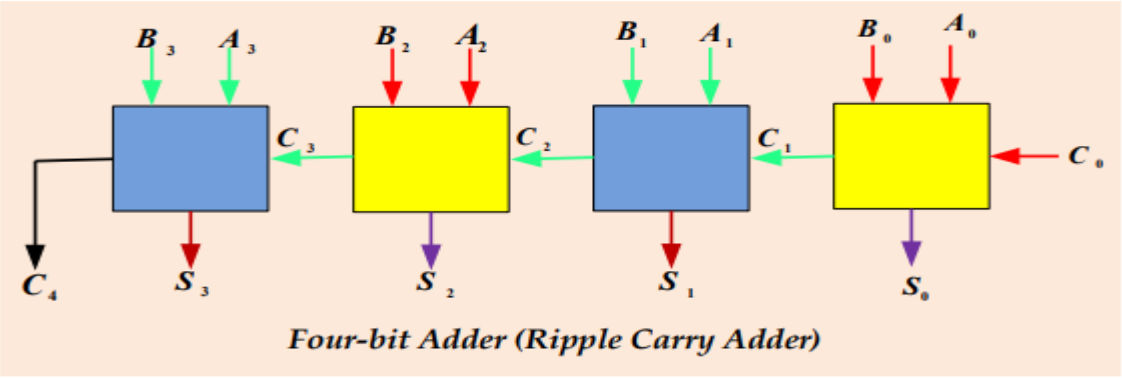
$$\begin{cases} S = C_{in} \oplus (X \oplus Y) \\ C_{out} = C_{in} \cdot (X \oplus Y) + XY \end{cases}$$



Inputs			Outputs		
a	b	CarryIn	CarryOut	Sum	Comments
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

Binary Adder (Asynchronous Ripple-Carry Adder)

- A binary adder is a digital circuit that produces the arithmetic sum of two binary numbers.
- A binary adder can be constructed with full adders connected in¬ cascade with the output carry form each full adder connected to the input carry of the next full adder in the chain.
- The four-bit adder is a typical example of a standard component .It can¬ be used in many application involving arithmetic operations.
- The input carry to the adder is and it ripples through the full adders to the output carry bit binary adder requires full adders



Four-bit Adder (Ripple Carry Adder)

CARRY-LOOK AHEAD ADDER

- Fast adder circuit must speed up the generation of carry signals. Carry look ahead logic uses the concepts of generating and propagating carries. Where Si is the sum and Ci+1 is the carry out.

➢ The ***carry propagation time*** is an important attribute of the adder because it limits the speed with which two numbers are added.

➢ To reduce the carry propagation delay time:

1) Employ faster gates with reduced delays.
2) Employ the principle of Carry Lookahead Logic.

Proof: ***(using carry lookahead logic)***

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$

The output sum and carry are:

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

✓ G_i-called a ***carry generate***, and it produces a carry of 1 when both A_i and B_i are 1.

✓ P_i-called a ***carry propagate***, it determines whether a carry into stage i will propagate into stage $i+1$.

✓ The ***Boolean function*** for the carry outputs of each stage and substitute the value of each C_i form the previous equations:

$$\left\{\begin{array}{l} C_0 = \textit{input carry} \\ C_1 = G_0 + P_0 C_0 \\ C_2 = G_1 + P_1 C_1 = G_1 + P_1(G_0 + P_0 C_0) \\ \quad = G_1 + P_1 G_0 + P_1 P_0 C_0 \\ C_3 = G_2 + P_2 C_2 = G_2 + P_2(G_1 + P_1 G_0 + P_1 P_0 C_0) \\ \quad = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0 \end{array}\right\}$$

➢ The three Boolean functions C_1, C_2 and C_3 are implemented in the ***carry lookahead generator.***

> ***The two level-circuit for the output carry C_4 is not shown, it can be easily derived by the equation.***

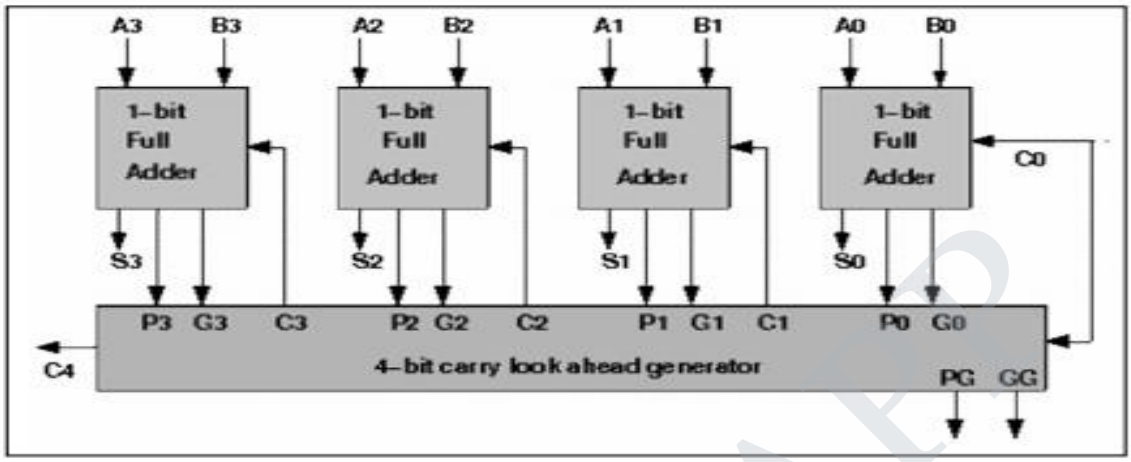
➢ C_3 does not have to wait for C_2 and C_1 to propagate, in fact C_3 is propagated at the same time as C_1 and C_2.

Adv:

1. Circuit is simplicity
2. Structure is slightly faster
3. Easy to understand
4. To eliminate inter stage carry delay.

Dadv:

1. Carry look-ahead is expensive

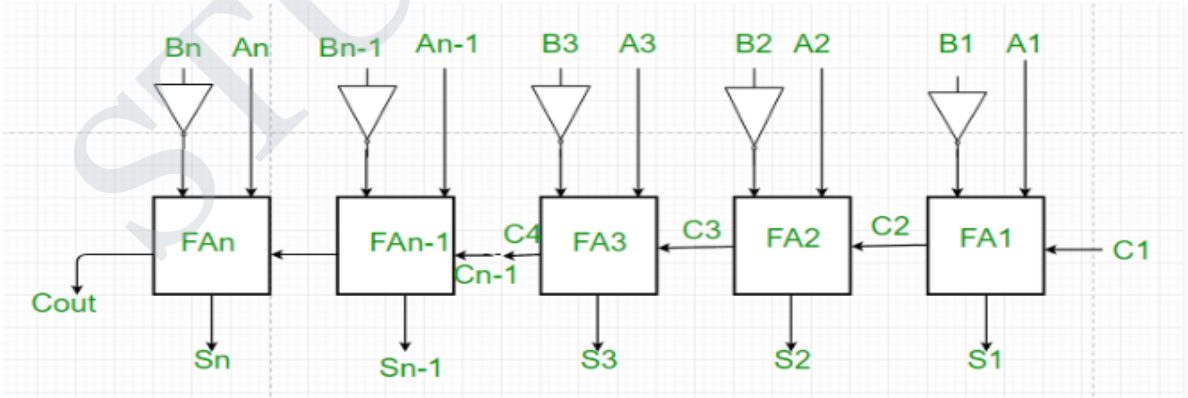


Carry Propagation Delay

The sum and carry output of any stage cannot be produced until the input carry occurs. This leads to a time delay in the addition process.

Parallel Subtractor

- A Parallel Subtractor is a digital circuit capable of finding the arithmetic difference of two binary numbers that is greater than one bit in length by operating on corresponding pairs of bits in parallel.
- The parallel subtractor can be designed in several ways including combination of half and full subtractors, all full subtractors or all full adders with subtrahend complement input.



Advantages of parallel Adder/Subtractor

- The parallel adder/subtractor performs the addition operation faster as compared to serial adder/subtractor.
- Time required for addition does not depend on the number of bits.
- The output is in parallel form i.e all the bits are added/subtracted at the same time.
- It is less costly.

Disadvantages of parallel Adder/Subtractor

- Each adder has to wait for the carry which is to be generated from the previous adder in chain.
- The propagation delay(delay associated with the travelling of carry bit) is found to increase with the increase in the number of bits to be added.

ADDITION AND SUBTRACTION:

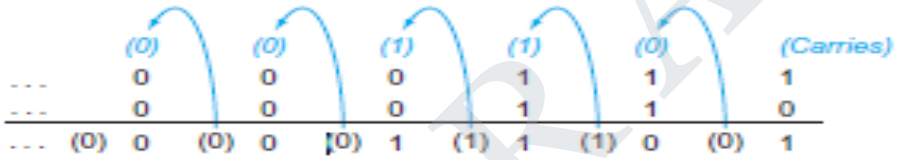
- Digits are added bit by bit from right to left, with carries passed to the next digit to the left.
- Subtraction uses addition. The appropriate operand is simply negated before being added.

Binary addition:

Let's try adding 6_{ten} to 7_{ten} in binary.

```
  0000 0000 0000 0000 0000 0000 0000 0111two = 7ten
+ 0000 0000 0000 0000 0000 0000 0000 0110two = 6ten
------------------------------------------------------
  0000 0000 0000 0000 0000 0000 0000 1101two = 13ten
```

The following figure shows the sums and carries. The carries are shown in parentheses.



- Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0.
- Hence, the operation for the second digit to the right is 0+1+1.
- This generates a 0 for this sum bit and a carry out of 1.
- The third digit is the sum of 1+1+1, resulting in a carry out of 1 and a sum bit of 1.
- The fourth bit is 1+0+0, yielding a 1 sum and no carry.

Binary subtraction:

- Subtracting 6_{ten} from 7_{ten} can be done directly

```
  0000 0000 0000 0000 0000 0000 0000 0111two = 7ten
- 0000 0000 0000 0000 0000 0000 0000 0110two = 6ten
------------------------------------------------------
  0000 0000 0000 0000 0000 0000 0000 0001two = 1ten
```

or via addition using the two's complement representation of −6:

```
  0000 0000 0000 0000 0000 0000 0000 0111two =  7ten
+ 1111 1111 1111 1111 1111 1111 1111 1010two = -6ten
------------------------------------------------------
  0000 0000 0000 0000 0000 0000 0000 0001two =  1ten
```

When overflow cannot occur in addition and subtraction?

Case: 1

- When adding operands with different signs, overflow cannot occur. The reason is the sum must be no larger than one of the operands. For example, -10+4=-6.
- Since the operands fit in 32 bits and the sum is no larger than an operand, the sum must fit in 32 bits as well. Therefore, no overflow can occur when adding positive and negative operands.

Case: 2

- When the signs of the operands are the same, overflow cannot occur. To see this, remember that c – a= c + (-a) because we subtract by negating the second operand and then add.
- Therefore, when we subtract operands of the same sign we end up by adding operands of different signs.

When overflow can occur in addition and subtraction?

Case: 1

- Overflow occurs when adding two positive numbers and the sum is negative

Case: 2

- Overflow occurs when adding two negative numbers and the sum is positive. This spurious sum means a carry out occurred into the sign bit.

Case: 3

- Overflow occurs in subtraction when we subtract a negative number from a positive number and get a negative result.

Case: 4

- When we subtract a positive number from a negative number and get a positive result. Such a ridiculous result means a borrow occurred from the sign bit.

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

- Add (add), add immediate (addi), and subtract (sub) cause exceptions on overflow.
- Add unsigned (addu), add immediate unsigned (addiu), and subtract unsigned (subu) do not cause exceptions on overflow.

Exception:

- Exception also called interrupt on many computers. An unscheduled event that disrupts program execution; used to detect overflow.

Interrupt:

- An exception that comes from outside of the processor.

EPC:

- MIPS include a register called the Exception Program Counter (EPC) to contain the address of the instruction that caused the exception.
- The instruction move from system control (mfc0) is used to copy EPC into a general-purpose register so that MIPS software has the option of returning to the off ending instruction via a jump register instruction.

MULTIPLICATION

- The first operand is called the multiplicand and the second the multiplier. The final result is called the product.
- If we ignore the sign bits, the length of the multiplication of an n-bit multiplicand and an m-bit multiplier is a product that is n+ m bits long.
- That is, n+ m bits are required to represent all possible products.
- For example, **Multiplying** 1000_{ten} by 1001_{ten}: **Multiplicand** 1000_{ten} **Multiplier** 1001_{ten}

$1000_{ten} \times 1001_{ten}$

1000
0000
0000
1000

Product 1001000_{ten}

Case: 1

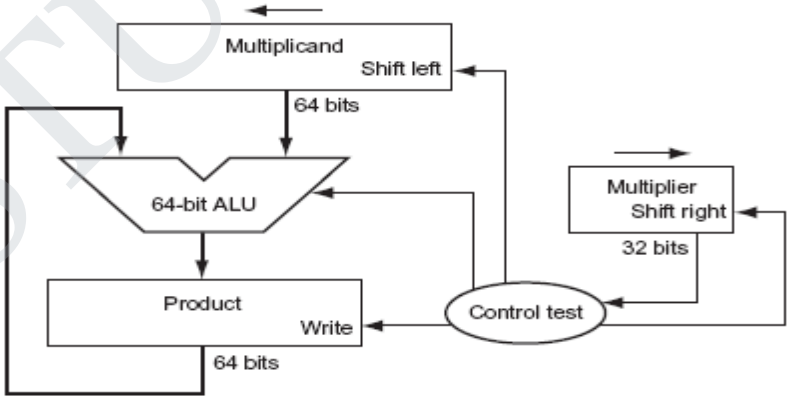
- Just place a copy of the multiplicand (1 x multiplicand) in the proper place if the multiplier digit is a 1.

Case: 2

- Place 0 (0 x multiplicand) in the proper place if the digit is 0.

FIRST VERSION OF THE MULTIPLICATION HARDWARE

- The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits.
- The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step.
- The multiplier is shifted in the opposite direction at each step.
- The algorithm starts with the product initialized to 0.
- Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.



Step: 1

- The least significant bit of the multiplier (Multiplier0) determines whether the multiplicand is added to the Product register.
- If the least significant bit of the multiplier is 1, add the multiplicand to the product.

Step: 2

- If not, go to the next step. Shift left the multiplicand register by 1 bit.

Step: 3

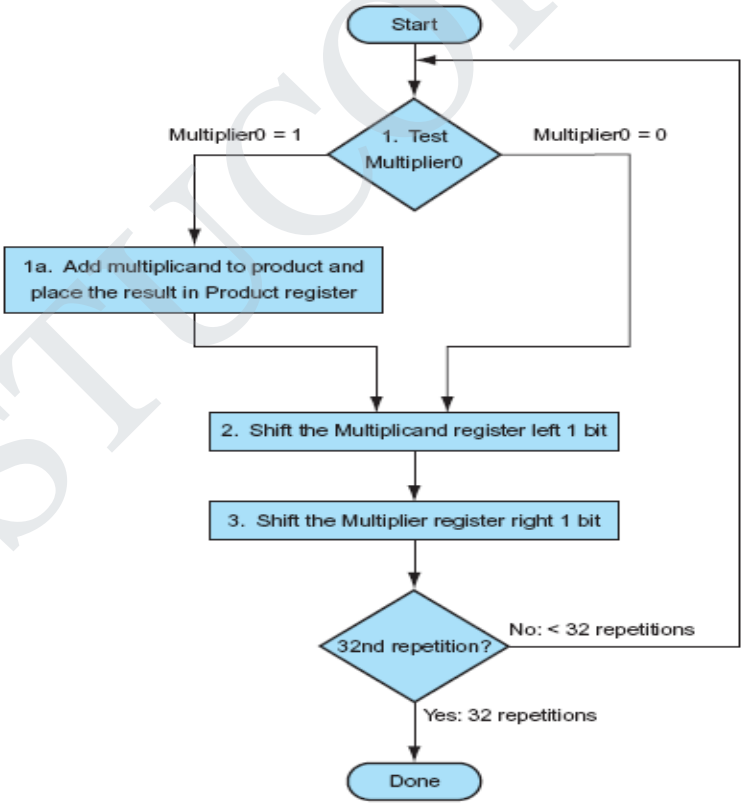
- Then shift right the multiplier register by 1 bit. These three steps are repeated 32 times to obtain the product.
- If each step took a clock cycle, this algorithm would require almost 100 clock cycles to multiply two 32-bit numbers.

Example:

Using 4-bit numbers to save space, multiply 2ten x 3ten, or 0010two x 0011two.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⇒ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⇒ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

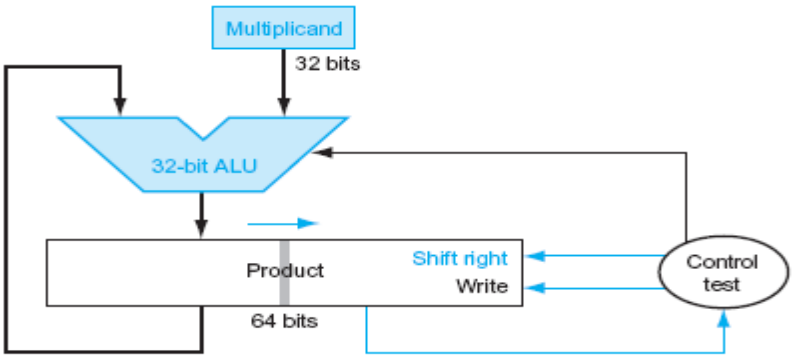
Flowchart:



Refined version of the multiplication hardware:

- Comparing with the first algorithm the Multiplicand register, ALU, and Multiplier register are all 32 bits wide, with only the Product register left at 64 bits.
- Now the product is shifted right. The separate Multiplier register also disappeared.

- The multiplier is placed instead in the right half of the Product register.

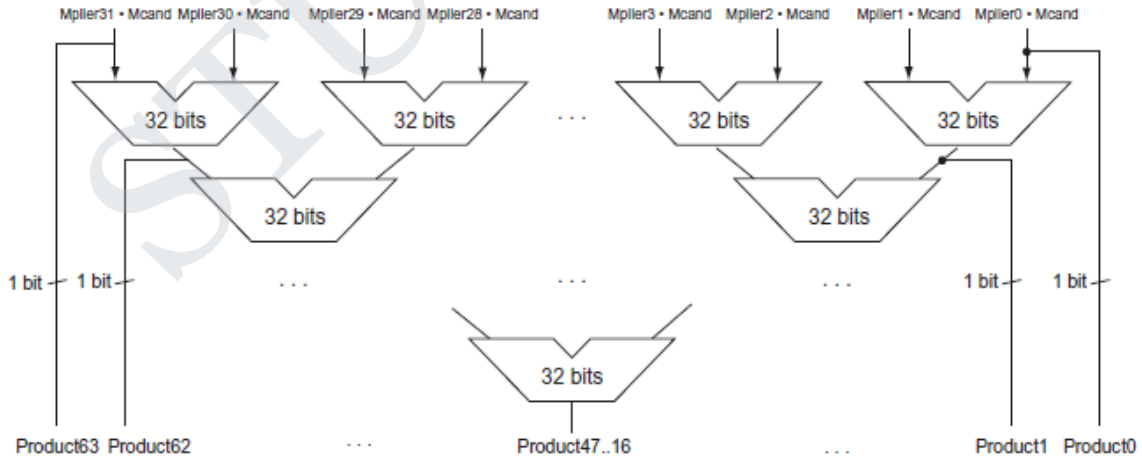


Signed Multiplication

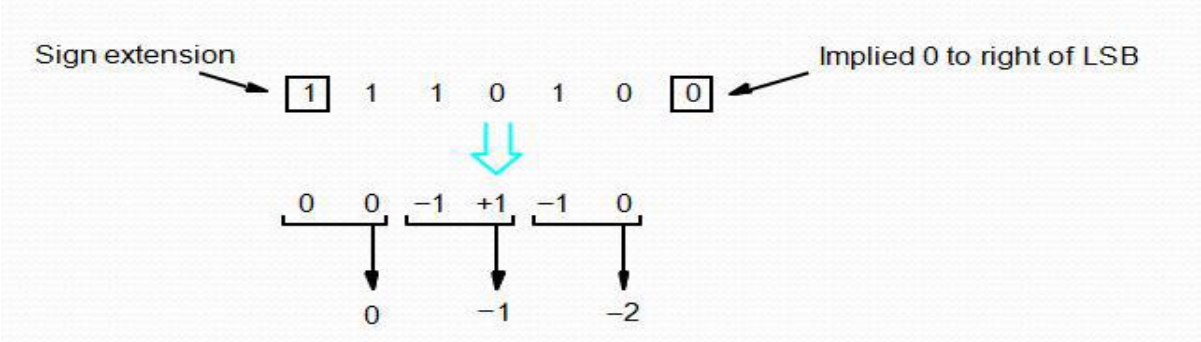
- First convert the multiplier and multiplicand to positive numbers and then remember the original signs.
- The algorithms should then be run for 31 iterations, leaving the signs out of the calculation.

Faster Multiplication

- Hardware designers can now build much faster multiplication hardware. Whether the multiplicand is to be added or not is known at the beginning of the multiplication by the 32 multiplier bits.
- Faster multiplications are possible by essentially providing one 32-bit adder for each bit of the multiplier:
- One input is the multiplicand ANDed with a multiplier bit, and the other is the output of a prior adder.
- To connect the outputs of adders on the right to the inputs of adders on the left, making a stack of adders 32 high.
- Rather than use a single 32-bit adder 31 times, this hardware "unrolls the loop" to use 31adders and then organizes them to minimize delay.



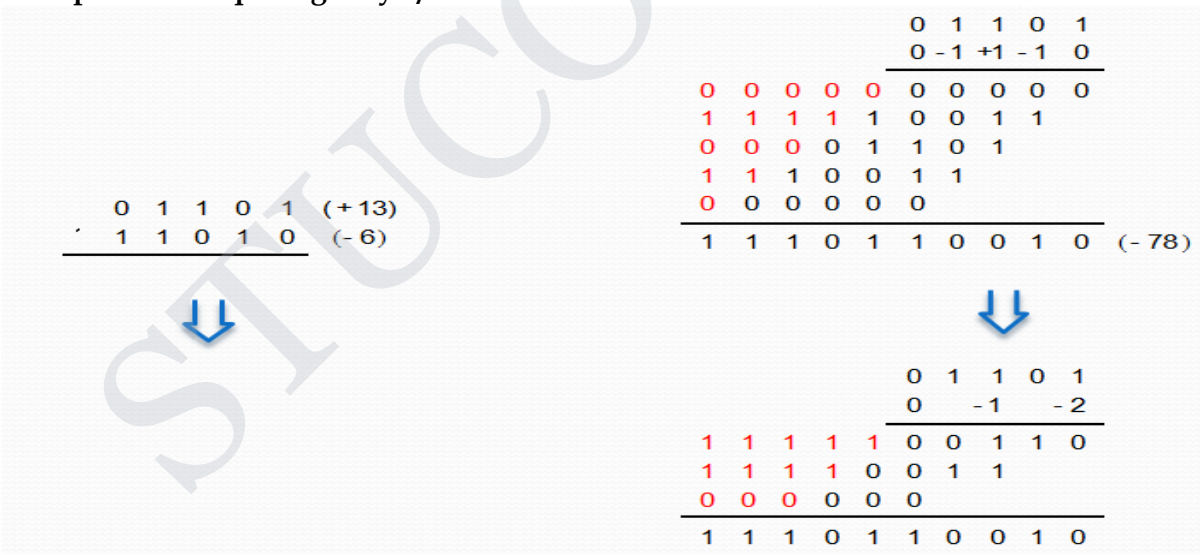
BOOTH'S BIT-PAIR RECODING OF THE MULTIPLIER.
A=+13 (Multiplicand) AND B= -6 (Multiplier)
Bit-pair recoding halves the maximum number of summands (versions of the multiplicand).
Example



Multiplicand Selection Decisions

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

Multiplication requiring only n/2 summands



BOOTH'S MULTIPLICATION ALGORITHM WITH SUITABLE EXAMPLE

Booth's Algorithm Principle:

- Performs additions and subtractions of the Multiplicand, based on the value of the multiplier bits.
- The algorithm looks at two adjacent bits in the Multiplier in order to decide the operation to be performed.

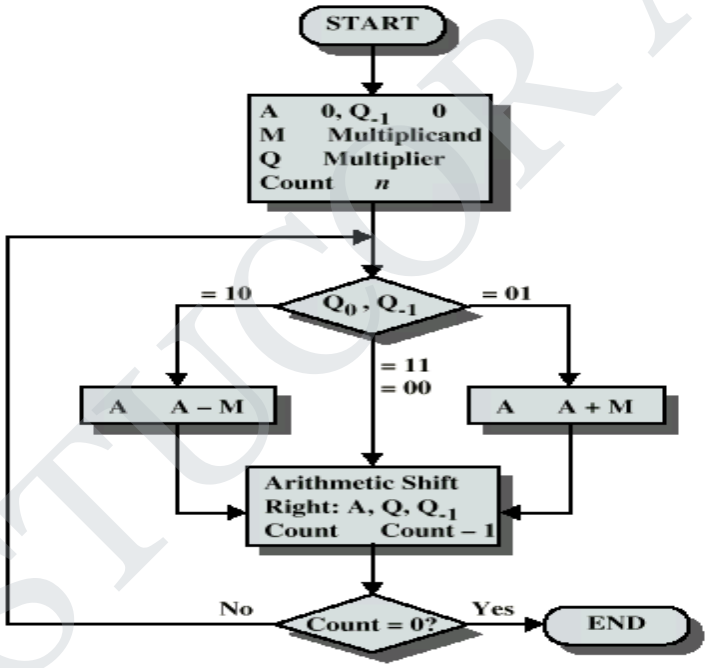
- The Multiplier bits are considered from the least significant bit (right-most) to the most significant bit; by default a 0 will be considered at the right of the least significant bit of the multiplier.
- If Multiplicand has Md bits and Multiplier has Mp bits, the result will be stored in a Md+Mp bit register and will be initialised with 0s
- As repeated operations and shifts are performed on partial results, the result register is the accumulator (A).
- Booth's algorithm gives a procedure for multiplying signed binary integer. It is based on the fact that strings of 0's in the multiplier require no addition but only shifting and a string of 1's in the multiplier require both operations.

Algorithm

The Q0 bit of the register Q and Q-1 is examined:

- If two bits are the same (11 or 00), then all of the bits of the A, Q and Q_1 registers are shifted to the right 1 bit. This shift is called arithmetic shift right.
- If two bits differ i.e., whether 01, then the multiplicand is adder or 10, then the multiplicand is subtracted from the register A. after that, right shift occurs in the register A, Q and Q_1.

Flowchart of Booth's Algorithm for 2's complement multiplication



Example of Booth's Algorithm (7*3=21)

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

A DIVISION ALGORITHM AND HARDWARE:

Dividend:

- A number being divided is called dividend.

Divisor:

- A number that the dividend is divided by is called divisor.

Quotient:

- It is called the primary result of a division.
- A number that when multiplied by the divisor and added to the remainder produces the dividend is known as quotient.

Remainder:

- It is the secondary result of a division.
- A number that when added to the product of the quotient and the divisor produces the dividend is known as remainder.

The example is dividing $1{,}001{,}010_{ten}$ by 1000_{ten}:

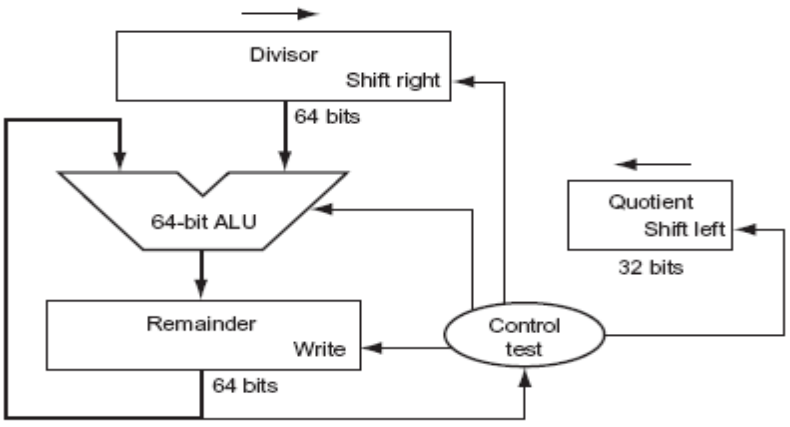
$$
\begin{array}{rll}
 & 1001_{ten} & \text{Quotient} \\
\text{Divisor } 1000_{ten} & \overline{)\,1001010_{ten}} & \text{Dividend} \\
 & -1000 & \\
 & \underline{\quad 10} & \\
 & 101 & \\
 & 1010 & \\
 & -1000 & \\
 & \underline{\quad 10_{ten}} & \text{Remainder}
\end{array}
$$

- Divide's two operands, called the dividend and divisor, and the result, called the quotient, are accompanied by a second result, called the remainder.
- Here is another way to express the relationship between the components:

Dividend=Quotient x Divisor + Remainder

Division Hardware:

- The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits.
- The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit each iteration.
- The remainder is initialized with the dividend.
- Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.



Divide algorithm:

Step: 1

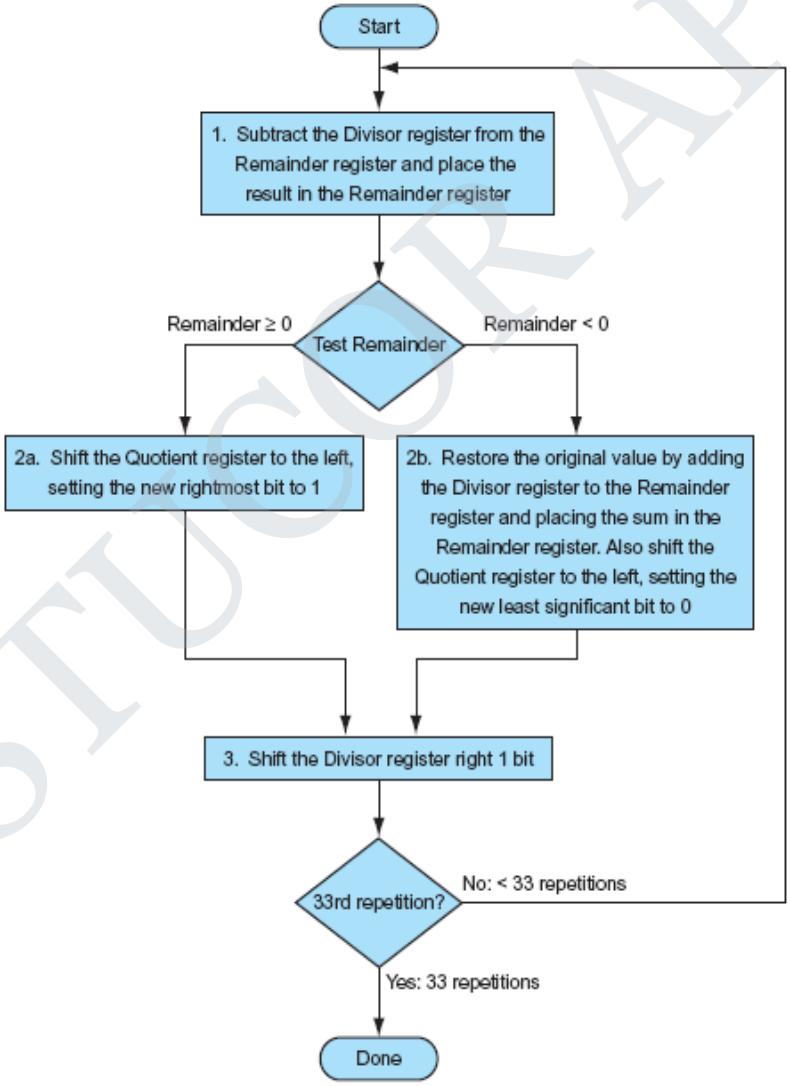
- It must first subtract the divisor register from the Remainder register and place the result in the Remainder register.

Step: 2

- Next we performed the comparison in the set on less than instruction.
- If the result is positive, the divisor was smaller or equal to the dividend, so shift the Quotient register to the left, setting the new rightmost bit to 1.
- If the result is negative, the next step Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register.
- Also shift the Quotient register to the left, setting the new least significant bit to 0

Step: 3

- The divisor is shifted right by 1 bit and then we iterate again.
- The remainder and quotient will be found in their registers after the iterations are complete.

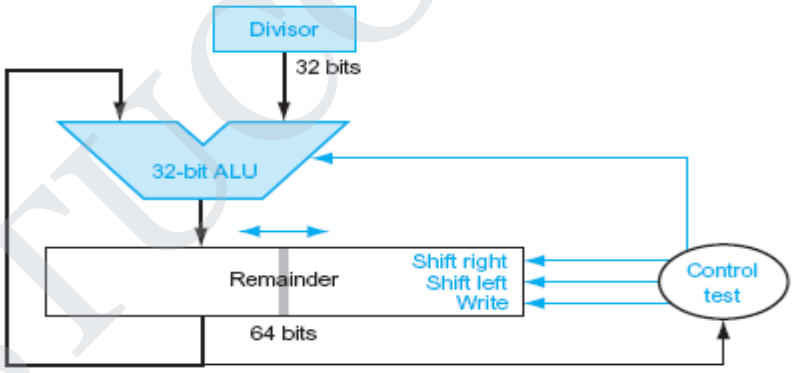


Using a 4-bit version of the algorithm to save pages, let's try dividing 7_{ten} by 2_{ten}, or 0000 0111two by 0010two.

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

An improved version of the division hardware:

- The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits.
- Compared to above division hardware, the ALU and Divisor registers are halved and the remainder is shifted left.
- This version also combines the Quotient register with the right half of the Remainder register.



Signed Division

- The simplest solution is to remember the signs of the divisor and dividend and then negate the quotient if the signs disagree.
- The one complication of signed division is that we must also set the sign of the remainder. Remember that the following equation must always hold:

$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

To understand how to set the sign of the remainder, let's look at the example of dividing all the combinations of $\pm 7_{ten}$ by $\pm 2_{ten}$. The first case is easy:

$$+7 \div +2\text{: Quotient} = +3, + \text{Remainder} = +1$$

Checking the results:

$$+7 = 3 \times 2 + (+1) = 6 + 1$$

If we change the sign of the dividend, the quotient must change as well:

$$-7 \div +2\text{: Quotient} = -3$$

Rewriting our basic formula to calculate the remainder:

$$\begin{aligned}\text{Remainder} &= (\text{Dividend} - \text{Quotient} \times \text{Divisor}) = -7 - (-3x + 2)\\ &= -7 - (-6) = -1\end{aligned}$$

So,

$$-7 \div +2\text{: Quotient} = -3, \text{Remainder} = -1$$

Checking the results again:

$$-7 = -3 \times 2 + (-1) = -6 - 1$$

The reason the answer isn't a quotient of −4 and a remainder of +1, which would also fit this formula, is that the absolute value of the quotient would then change depending on the sign of the dividend and the divisor! Clearly, if

Rule:

- The dividend and remainder must have the same signs, no matter what the signs of the divisor and quotient.

$$-(x \div y) \neq (-x) \div y$$

We calculate the other combinations by following the same rule:

$$+7 \div -2\text{: Quotient} = -3, \text{Remainder} = +1$$

$$-7 \div -2\text{: Quotient} = +3, \text{Remainder} = -1$$

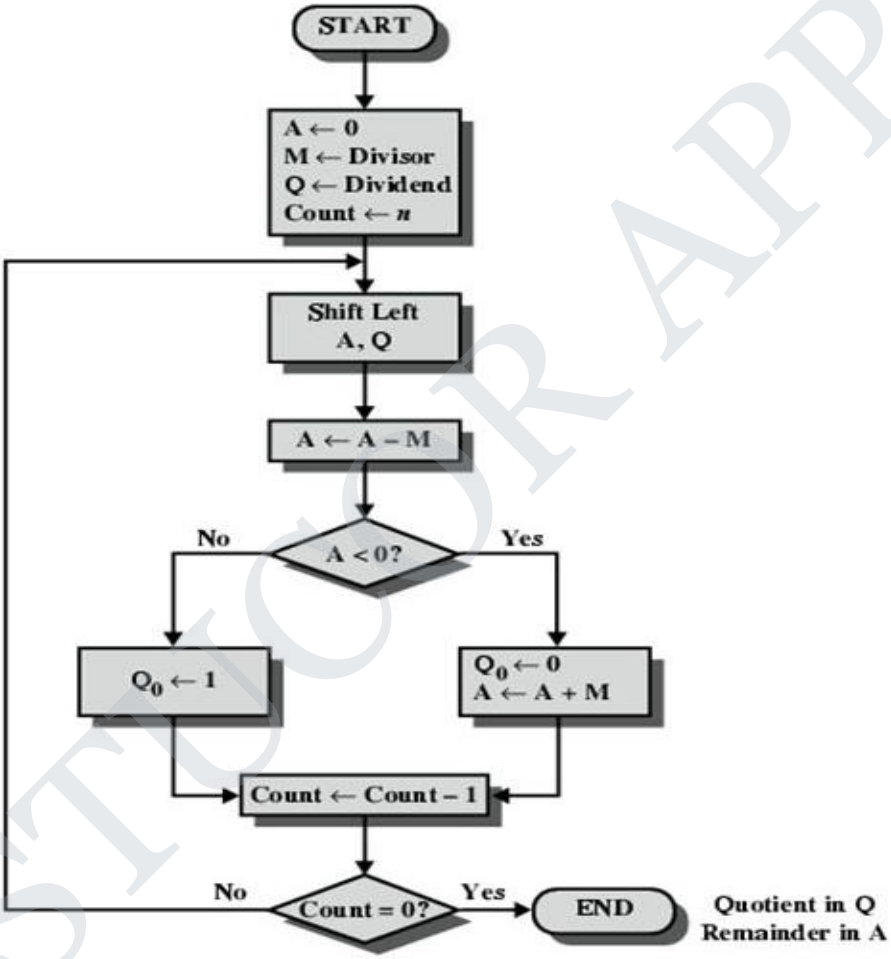
Faster Division

- There are techniques to produce more than one bit of the quotient per step.
- The SRT division technique tries to predict several quotient bits per step, using a table lookup based on the upper bits of the dividend and remainder.
- These algorithms use 6 bits from the remainder and 4 bits from the divisor to index a table that determines the guess for each step.

Category	Instruction	Example		Meaning	Comments
Arithmetic	multiply	mult	$s2,$s3	HI, Lo = $s2 × $s3	64-bit signed product in HI, Lo
	multiply unsigned	multu	$s2,$s3	HI, Lo = $s2 × $s3	64-bit unsigned product in HI, Lo
	divide	div	$s2,$s3	Lo = $s2 / $s3, HI = $s2 mod $s3	Lo = quotient, HI = remainder
	divide unsigned	divu	$s2,$s3	Lo = $s2 / $s3, HI = $s2 mod $s3	Unsigned quotient and remainder
	move from HI	mfhi	$s1	$s1 = HI	Used to get copy of HI
	move from Lo	mflo	$s1	$s1 = Lo	Used to get copy of Lo

Restoring Division Algorithm

- **Step-1:** First the registers are initialized with corresponding values (Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend)
- **Step-2:** Then the content of register A and Q is shifted right as if they are a single unit
- **Step-3:** Then content of register M is subtracted from A and result is stored in A
- **Step-4:** Then the most significant bit of the A is checked if it is 0 the least significant bit of Q is set to 1 otherwise if it is 1 the least significant bit of Q is set to 0 and value of register A is restored i.e the value of A before the subtraction with M
- **Step-5:** The value of counter n is decremented
- **Step-6:** If the value of n becomes zero we get of the loop otherwise we repeat fro step 2
- **Step-7:** Finally, the register Q contain the quotient and A contain remainder



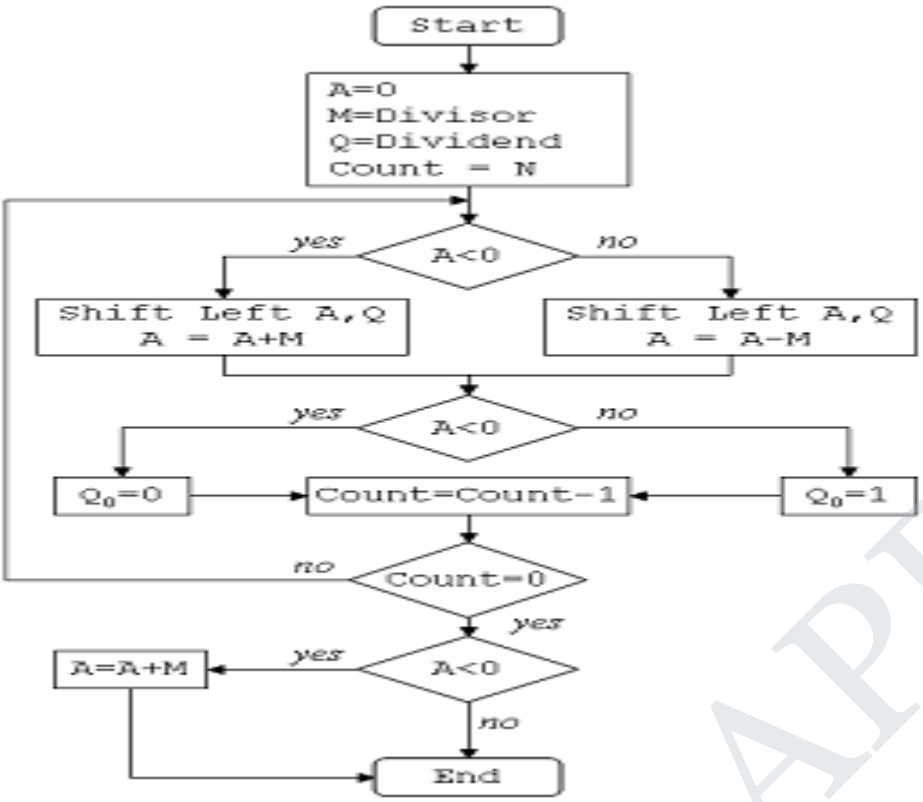
Example: 8 divide by 3=2 (2/3)

The quotient (0010)2 = 2 is in register Q, and the reminder (0010)2 = 2 is in register A.

	[M]	0011		
	[A]	0000	[Q]	1000
left shift A/Q		0001		000.
$A = [A] - [M]$	+	1101		
$A < 0$		1110		0000
$A = [A] + [M]$	+	0011		
		0001		0000
left shift A/Q		0010		000.
$A = [A] - [M]$	+	1101		
$A < 0$		1111		0000
$A = [A] + [M]$	+	0011		
		0010		0000
left shift A/Q		0100		000.
$A = [A] - [M]$	+	1101		
$A > 0$		0001		0001
left shift A/Q		0010		001.
$A = [A] - [M]$	+	1101		
$A < 0$		1111		0010
$A = [A] + [M]$	+	0011		
		0010		0010

Non-Restoring Division Algorithm

- **Step-1:** First the registers are initialized with corresponding values (Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend)
- **Step-2:** Check the sign bit of register A
- **Step-3:** If it is 1 shift left content of AQ and perform A = A+M, otherwise shift left AQ and perform A = A-M (means add 2's complement of M to A and store it to A)
- **Step-4:** Again the sign bit of register A
- **Step-5:** If sign bit is 1 Q[0] become 0 otherwise Q[0] become 1 (Q[0] means least significant bit of register Q)
- **Step-6:** Decrements value of N by 1
- **Step-7:** If N is not equal to zero go to **Step 2** otherwise go to next step
- **Step-8:** If sign bit of A is 1 then perform A = A+M
- **Step-9:** Register Q contain quotient and A contain remainder



	[M]	0011		
	[A]	0000	[Q]	1000
left shift A/Q		0001		000.
A=[A]-[M]	+	1101		
$A < 0$		1110		0000
left shift A/Q		1100		000.
$A = [A] + [M]$	+	0011		
$A < 0$		1111		0000
left shift A/Q		1110		000.
$A = [A] + [M]$	+	0011		
$A > 0$		0001		0001
left shift A/Q		0010		001.
$A = [A] - [M]$	+	1101		
$A < 0$		1111		0010
$A = [A] + [M]$	+	0011		
		0010		0010

FLOATING POINT

Normalized number:

- A number in floating-point notation that has no leading 0^s is known as normalized number. i.e., a number start with a single nonzero digit.
- For example, $1.0_{ten} \times 10^{-9}$ is in normalized scientific notation, but $0.1_{ten} \times 10^{-8}$ and $10.0_{ten} \times 10^{-10}$ are not.

Binary numbers in scientific notation:

- To keep a binary number in normalized form, we need a base that we can increase or decrease by exactly the number of bits the number must be shifted to have one nonzero digit to the left of the decimal point. $1.0_{two} \times 2^{-1}$

Floating-Point Representation

Floating point:

- Computer arithmetic that represents numbers in which the binary point is not fixed.

Fraction:

- The value, generally between 0 and 1, placed in the fraction field. The fraction is also called the mantissa.

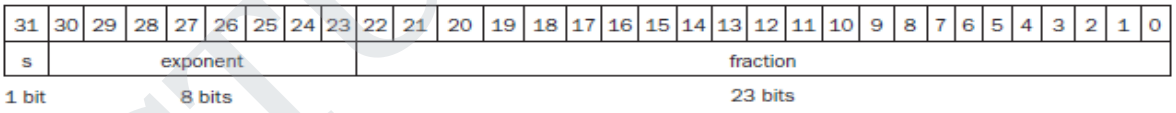
Exponent:

- In the numerical representation system of floating-point arithmetic, the value that is placed in the exponent field.

Single precision:

- A floating-point value represented in a single 32-bit word. Floating-point numbers are usually a multiple of the size of a word.
- Where s is the sign of the floating-point number (1 meaning negative), exponent is the value of the 8-bit exponent field (including the sign of the exponent), and fraction is the 23-bit number.
- F involves the value in the fraction field and E involves the value in the exponent field.

Format:



In general, floating-point numbers are of the form

$$(-1)^S \times F \times 2^E$$

Overflow:

- A situation in which a positive exponent becomes too large to fit in the exponent field is known as overflow.

Underflow:

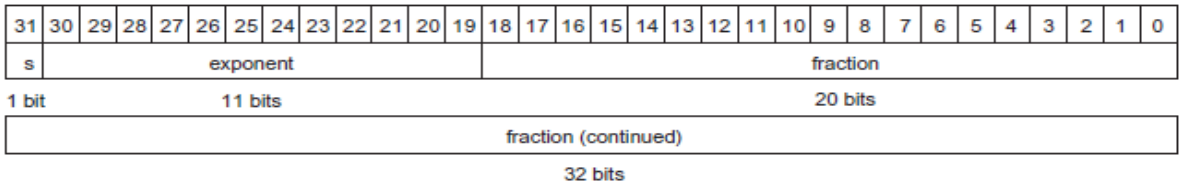
- A situation in which a negative exponent becomes too large to fit in the exponent field is known as underflow.

Double precision:

- One way to reduce chances of underflow or overflow is called double, and operations on doubles are called double precision floating-point arithmetic.
- It has a larger exponent. A floating-point value represented in two 32-bit words.

- Where s is still the sign of the number, exponent is the value of the 11-bit exponent field, and fraction is the 52-bit number in the fraction field.

Format:



IEEE 754 Format:

- MIPS double precision allows numbers almost as small as $2.0_{ten} \times 10^{+308}$ and almost as large as $2.0_{ten} \times 10^{-308}$.
- Although double precision does increase the exponent range.
- Its primary advantage is its greater precision because of the much larger fraction.
- IEEE 754 makes the leading 1-bit of normalized binary numbers implicit.
- Hence, the number is actually 24 bits long in single precision (implied 1 and a 23-bit fraction), and 53 bits long in double precision (1+52).

$$(-1)^S \times (1 + \text{Fraction}) \times 2^E$$

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

For example, $1.0_{two} \times 2^{-1}$ would be represented as

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

The value $1.0_{two} \times 2^{+1}$ would look like the smaller binary number

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

- The desirable notation must therefore represent the most negative exponent as 00 … 00_{two} and the most positive as 11 … 11_{two}.
- This convention is called biased notation, with the bias being the number subtracted from the normal, unsigned representation to determine the real value.
- IEEE 754 uses a bias of 127 for single precision, so an exponent of -1 is represented by the bit pattern of the value $-1+127_{ten}$, or 126_{ten}=0111 1110_{two}, and +1 is represented by 1+127, or 128_{ten} = 1000 0000_{two}.
- The exponent bias for double precision is 1023. Biased exponent means that the value represented by a floating-point number is really

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

The range of single precision numbers is then from as small as

$$\pm 1.00000000000000000000000_{two} \times 2^{-126}$$

to as large as

$$\pm 1.11111111111111111111111_{two} \times 2^{+127}.$$

Example: 1

Floating-Point Representation

Show the IEEE 754 binary representation of the number -0.75_{ten} in single and double precision.

The number -0.75_{ten} is also

$$-3/4_{ten} \text{ or } -3/2^2_{ten}$$

It is also represented by the binary fraction

$$-11_{two}/2^2_{ten} \text{ or } -0.11_{two}$$

In scientific notation, the value is

$$-0.11_{two} \times 2^0$$

and in normalized scientific notation, it is

$$-1.1_{two} \times 2^{-1}$$

The general representation for a single precision number is

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-127)}$$

Subtracting the bias 127 from the exponent of $-1.1_{two} \times 2^{-1}$ yields

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 000_{two}) \times 2^{(126-127)}$$

The single precision binary representation of -0.75_{ten} is then

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit | 8 bits | 23 bits

The double precision representation is

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}) \times 2^{(1022-1023)}$$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit | 11 bits | 20 bits

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

32 bits

Example: 2

Converting Binary to Decimal Floating Point

What decimal number is represented by this single precision float?

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

The sign bit is 1, the exponent field contains 129, and the fraction field contains $1 \times 2^{-2} = 1/4$, or 0.25. Using the basic equation,

$$\begin{aligned}(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})} &= (-1)^1 \times (1 + 0.25) \times 2^{(129-127)} \\ &= -1 \times 1.25 \times 2^2 \\ &= -1.25 \times 4 \\ &= -5.0\end{aligned}$$

FLOATING-POINT ADDITION

- Assume that we can store only four decimal digits of the significand and two decimal digits of the exponent.

Example: 1

Perform floating point addition for the following numbers.

$$\mathbf{9.999_{ten} \times 10^{1} + 1.610_{ten} \times 10^{-1}}$$

Step 1:

- Compare the exponent of both the operands.
- If it equal add the two operand (significand) .If it is not equal then increase the smaller exponent.
- i.e., shift the smaller number to the right until its exponent would match the larger exponent. As per our example

  $\mathbf{1.610_{ten} \times 10^{-1}} = 0.1610_{ten} \times 10^{0} = 0.01610_{ten} \times 10^{1}$
- But we can represent only four decimal digits so, after shifting, the number is really

  $0.016_{ten} \times 10^{1}$

Step 2:

Now add the Significand

$$\begin{array}{l} 9.999 \times 10^{1} \\ 0.016 \times 10^{1} \\ \hline 10.015 \times 10^{1} \end{array}$$

Step 3:

- Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent.
- This sum is not in normalized scientific notation, so we need to adjust it:

  $10.015 \times 10^{1} = 1.0015 \times 10^{2} = 1.0015 \times 10^{2}$
- Whenever the exponent is increased or decreased, we must check for overflow or underflow. i.e., we must make sure that the exponent still fits in its field.

Step 4:

- Round the significand to the appropriate number of bits.
- If the sum may no longer be normalized and we would need to perform step 3 again.

  $\mathbf{1.002_{ten} \times 10^{2}}$

Example: 2

Perform floating point addition for the following numbers.

0.5_{ten} and -0.4375_{ten}

Solution:

Assuming that we keep 4 bits of precision. 0.5_{ten}

Operand 1: Convert the operands to binary

0.5 x 2 =1.0

Scientific Notation

0.1×2^0

Normalizing the above value

1.0×2^{-1}

Operand 2: Convert the operands to binary -0.4375_{ten}

0.4375 x 2 = 0.8750

0.8750 x 2 = 1.7500

0.7500 x 2 = 1.5000

1.5000 x 2 = 1.0000

Scientific Notation

0.0111×2^0

Normalizing the above value

1.110×2^{-2}

Step 1:

The significand of the number with the lesser exponent ($-1.110_{two} \times 2^{-2}$) is shifted right until its exponent matches the larger number:

$$-1.110_{two} \times 2^{-2} = -0.111_{two} \times 2^{-1}$$

Step 2:

Add the significands

$$\begin{array}{rl} & 1.000 \times 2^{-1} \\ - & 0.111 \times 2^{-1} \text{ [Subtraction]} \\ \hline & 0.001 \times 2^{-1} \end{array}$$

Step 3: Normalize the sum and checking for overflow or underflow

$0.001 \times 2^{-1} = 1.0 \times 2^{-4}$

Step 4: Round the sum

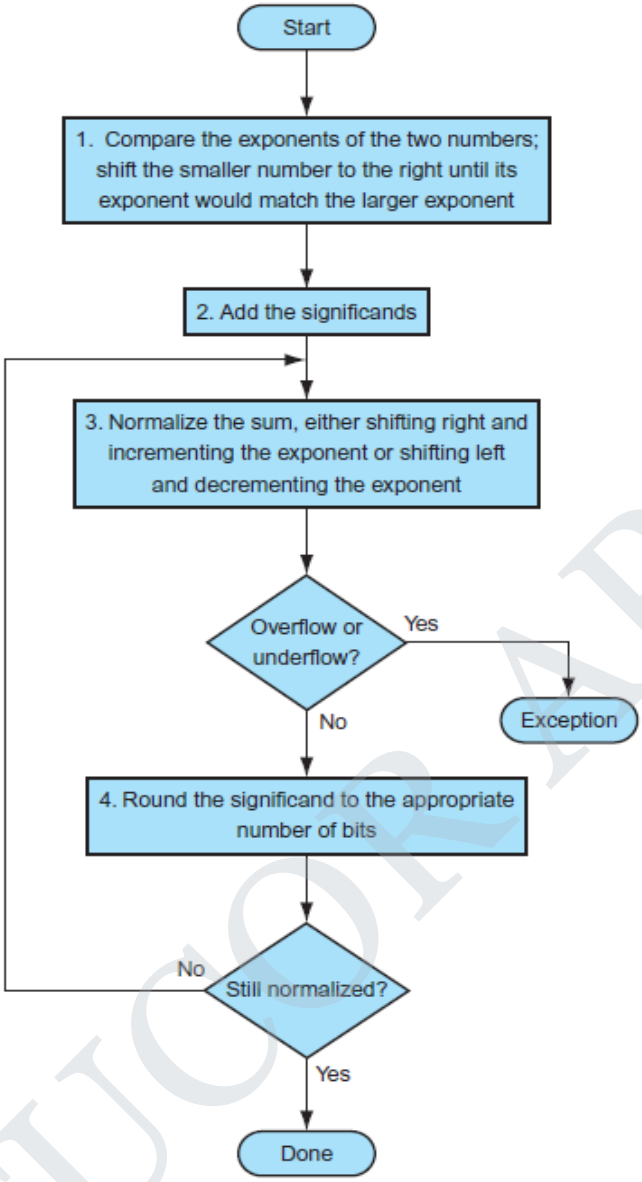
1.000×2^{-4}

Then convert the sum to decimal

$1.000 \times 2^{-4} = 0.0001_{two}$

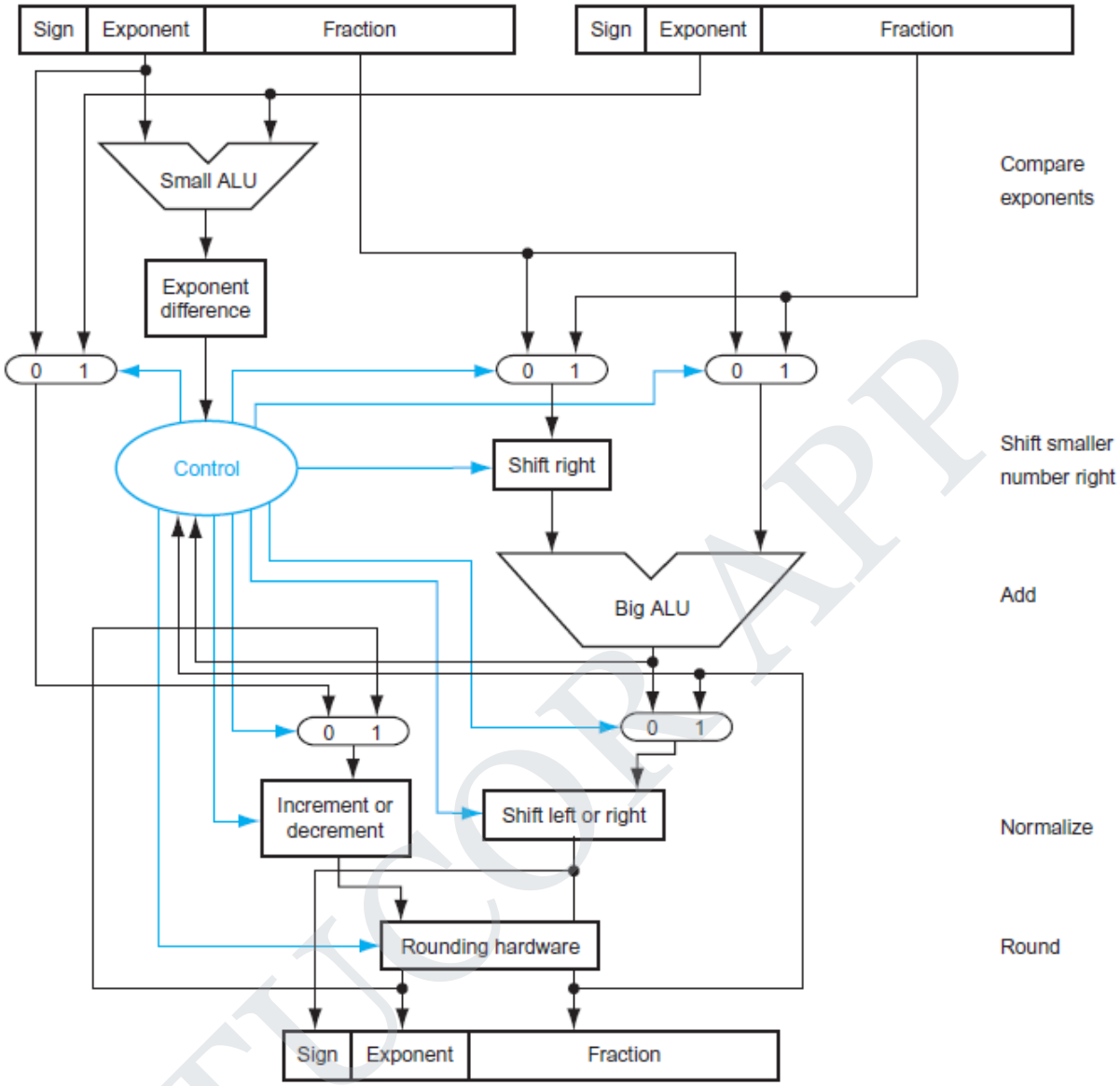
$1 / 2^4 = 1/16_{ten} = 0.0625_{ten}$

Flowchart:



- 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.

Block Diagram:



FLOATING-POINT MULTIPLICATION

Example: 1

Multiplying decimal numbers in scientific notation:

$$1.110_{ten} \times 10^{10} \times 9.200_{ten} \times 10^{-5}$$

Assume that we can store only four digits of the significand and two digits of the exponent.

Step 1:

- We calculate the exponent of the product by simply adding the exponents of the operands together:

  New exponent = 10 + (-5) = 5

- Let's do this with the biased exponents as well to make sure we obtain the same result:

  10 + 127 = 137, and -5 + 127 = 122, so New exponent = 137 +122=259

- This result is too large for the 8-bit exponent field.
- The problem is with the bias because we are adding the biases as well as the exponents.

- New exponent= (10+127)+(-5+127)=(5+2X127)=259
- To get the correct biased sum when we add biased numbers, we must subtract the bias from the sum.
- New exponent=137+122-127=259-127=132= (5+127) and 5 is indeed the exponent we calculated initially.

Step 2:

Next comes the multiplication of the significands:

$$\begin{array}{rl} & 1.110_{ten} \\ X & 9.200_{ten} \\ \hline & 0000 \\ & 0000 \\ & 2220 \\ & 9990 \\ \hline \end{array}$$

The product is 10212000_{ten}

- Assuming that we can keep only three digits to the right of the decimal point, the product is 10.212ten x 105

Step 3:

This product is unnormalized, so we need to normalize it:

$$10.212_{ten} \times 10^5 = 1.0212_{ten} \times 10^6$$

- After the multiplication, the product can be shifted right one digit and adding 1 to the exponent.
- At this point, we can check for overflow and underflow. Underflow may occur if both operands are small, that is, if both have large negative exponents.

Step 4:

Round of the Product

$$1.021_{ten} \times 10^6$$

Step 5:

- The sign of the product depends on the signs of the original operands.
- If they are both the same, the sign is positive; otherwise, it's negative. Hence, the product is

$$+1.021_{ten} \times 10^6$$

Example: 2

Multiple the numbers 0.5_{ten} and -0.4375_{ten}, using the steps in the above algorithm

- Binary equivalent of $0.5_{ten} = 1.000 \times 2^{-1}$ and $-0.4375_{ten} = -1.110 \times 2^{-2}$

Step 1 :

Adding the Exponents without bias

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

Or using the biased representation

$$(-1 + 127) + (-2 + 127) - 127 = (-1 - 2) + (127 + 127 - 127)$$
$$= -3 + 127 = 124$$

Step 2 : Multiplying the significands

$$
\begin{array}{r}
1.000_{two} \\
\times \quad 1.110_{two} \\
\hline
0000 \\
1000 \\
1000 \\
1000 \\
\hline
1110000_{two}
\end{array}
$$

- The Product is $1.110000_{two} \times 2^{-3}$,but we use 4 bits, so it is $1.110_{two} \times 2^{-3}$

Step 3:

Normalize the product, as per our example it is already normalized one

1.110×2^{-3}

Step 4:

Rounding the product no change

1.110 x 2-3

Step 5:

Since the signs of the original operands differ, make the sign of the product negative.

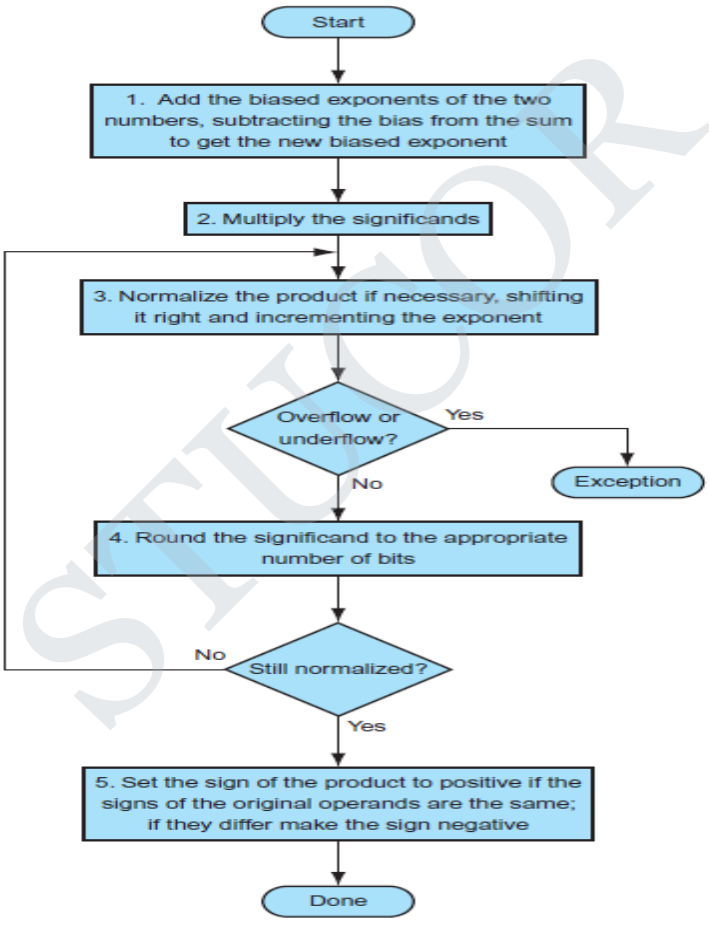
Hence, the product is

-1.110 x 2-3

Converting to decimal to check our results:

-1.110 x 2-3 = - 0.00111 = 1 /8 + 1 / 16 + 1 / 32 = - 0.21875

Flow Chart:



Guard Bit:

- Extra bits kept on the right during intermediate calculations of floating point numbers is called guard bit and it used to improve rounding accuracy.

Round:

- Method to make the intermediate floating-point result fit the floating-point format.
- The goal is typically to find the nearest number that can be represented in the format.

Sticky Bit:

- A bit used in rounding in addition to guard and round that is set whenever there are nonzero bits to the right of the round bit.

Subword Parallelism:

- By partitioning the 128-bit adder, a processor could use parallelism to perform simultaneous operations on short vectors of sixteen 8-bitoperands, eight 16-bit operands, four 32-bit operands, or two 64-bit operands.
- The cost of such partitioned adders was small.
- Given that the parallelism occurs within a wide word, the extensions are classified as subword parallelism.
- It is also classified under the more general name of data level parallelism.
- They have been also called vector or SIMD, for single instruction, multiple data.

UNIT-III

PROCESSOR AND CONTROL UNIT

Basic MIPS implementation – Building datapath – Control Implementation scheme – Pipelining – Pipelined datapath and control – Handling Data hazards & Control hazards – Exceptions.

A BASIC MIPS IMPLEMENTATION:

The implementation that includes a subset of the core MIPS instruction set:

- The memory-reference instructions **load word (lw)** and **store word (sw)**
- The arithmetic-logical instructions **add, sub, AND, OR, and slt**
- The instructions **branch equal (beq)** and **jump (j)**

An Overview of the Implementation:

For every instruction, the first two steps are identical:

1. Send the *program counter* (PC) to the memory that contains the code and fetch the instruction from that memory.

2. Read one or two registers, using fields of the instruction to select the registers to read. For the load word instruction, we need to read only one register, but most other instructions require reading two registers.

- After these two steps, the actions required to complete the instruction depend on the instruction class.
- For example, all instruction classes, except jump, use the arithmetic-logical unit (ALU) after reading the registers.
- The following diagram shows the high-level view of a MIPS implementation, focusing on the various functional units and their interconnection.

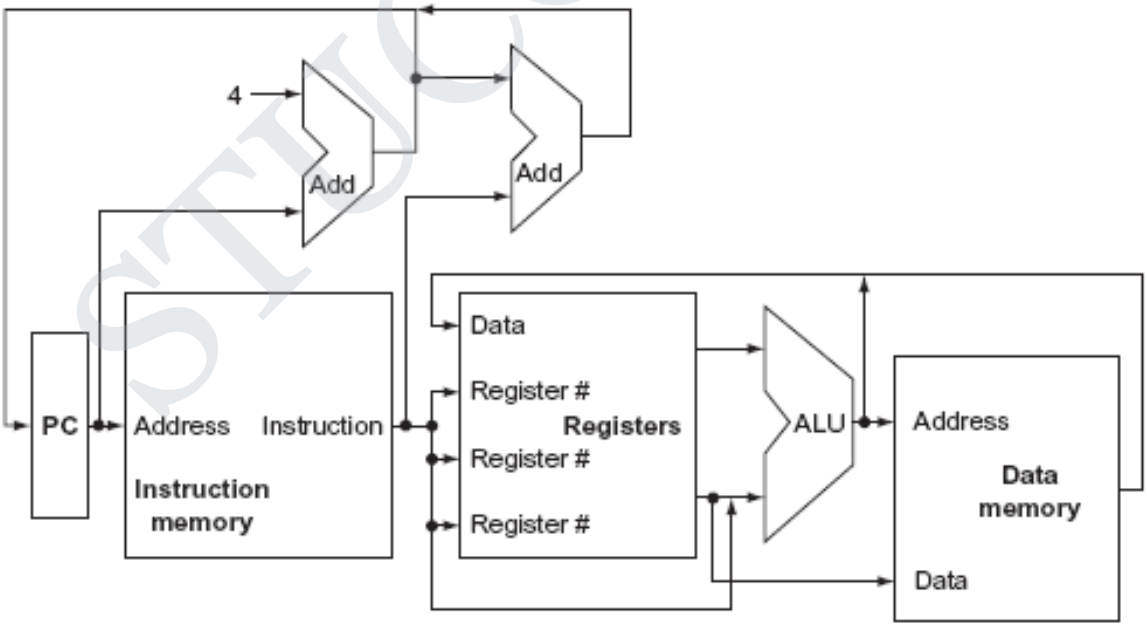


FIGURE 3.1 An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them.

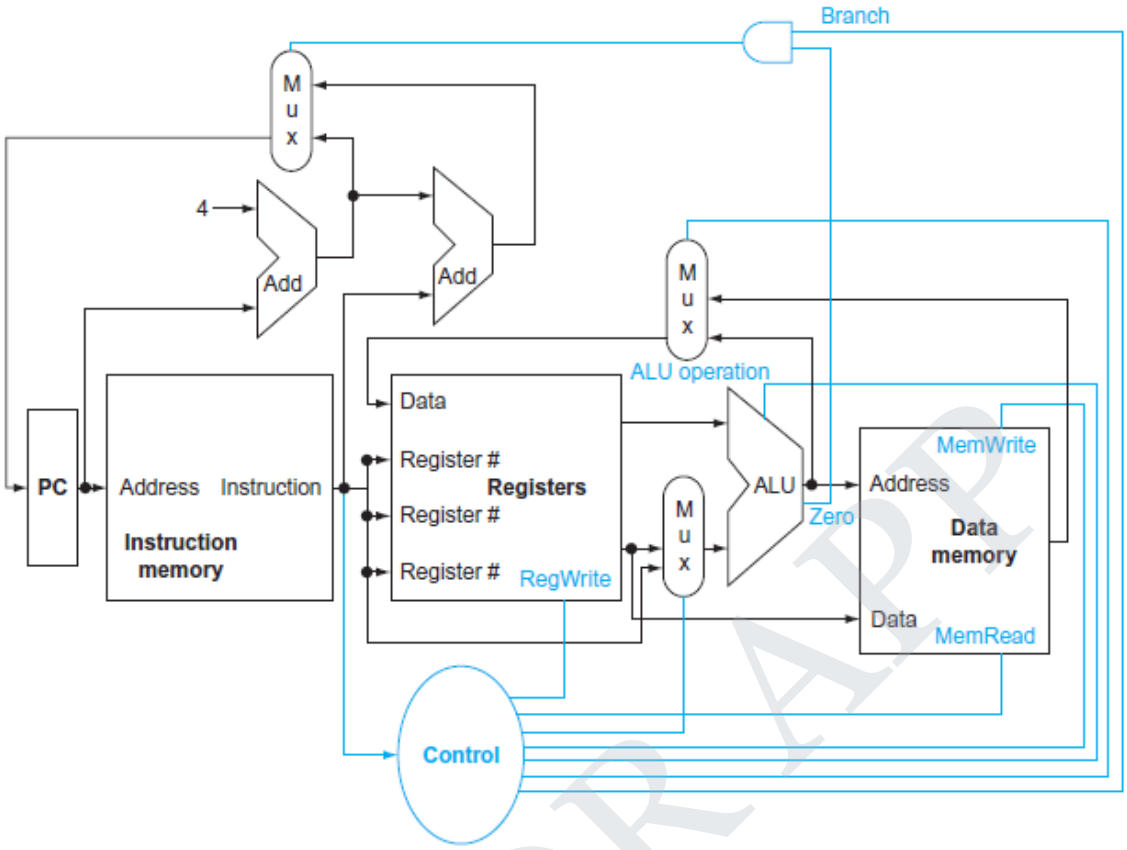
Operation:

- All instructions start by using the program counter to supply the instruction address to the instruction memory.
- After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction.
- Once the register operands have been fetched, all the instruction classes, except jump, use the ALU after reading the registers.
    - Memory reference instructions (load or store) use the ALU for an address calculation.
    - Arithmetic Logical instructions use the ALU for the operation execution.
    - Branches use the ALU for comparison.
- The second input to the ALU can come from a register or the immediate field of the instruction.
- After using the ALU, the actions required to complete various instruction classes are not same.
    - If the operation is a memory reference instruction a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register file.
    - If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register.
    - Branches require the use of the ALU output to determine the next instruction address, which comes either from the ALU (where the PC and branch off set are summed) or from an adder that increments the current PC by 4.

Basic implementation of MIPS with multiplexer:

- We must add a logic element that chooses from among the multiple sources and steers one of those sources to its destination. This selection is commonly done with a device called a **multiplexor**, although this device might better be called a **data selector** which selects from among several inputs based on the setting of its control lines.
- The control lines are set based primarily on information taken from the instruction being executed.
- The following figure shows the datapath with the three multiplexors added, as well as control lines for the major functional units.
- A control unit is used to determine how to set the control lines for the functional units and two of the multiplexors.
- The top multiplexor (“Mux”) controls what value replaces the PC (PC + 4 or the branch destination address); the multiplexor is controlled by the gate that “ANDs” together the Zero output of the ALU and a control signal that indicates that the instruction is a branch.
- The middle multiplexor, whose output returns to the register file, is used to steer the output of the ALU (in the case of an arithmetic-logical instruction) or the output of the data memory (in the case of a load) for writing into the register file.
- Finally, the bottommost multiplexor is used to determine whether the second ALU input is from the registers (for an arithmetic-logical instruction or a branch) or from the offset field of the instruction (for a load or store).

- The added control lines are straightforward and determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation.



Logic Design Conventions:

The datapath elements in the MIPS implementation consist of two different types of logic elements:

1. Combinational Elements:

- The elements that operate on data values are all **combinational**, which means that their outputs depend only on the current inputs.
- Given the same input, a combinational element always produces the same output.
- The ALU is an example of a combinational element. Given a set of inputs, it always produces the same output because it has no internal storage.

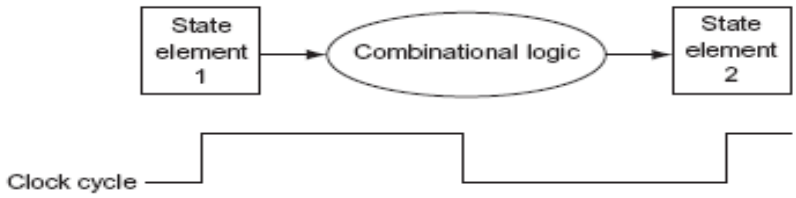
2. State Elements:

- It holds information about the state of the processor during the current clock cycle.
- An element contains state if it has some internal storage.
- All registers are state elements.
- A state element has at least two inputs and one output.
- The required inputs are the data value to be written into the element and the clock, which determines when the data value is written.
- The output from a state element provides the value that was written in an earlier clock cycle.

Clocking Methodology

- A **clocking methodology** defines when signals can be read and when they can be written.

- The approach used to determine when data is valid and stable relative to the clock.
- All state elements including memory, are assumed to be positive edge-triggered; that is, they change on the rising clock edge.



- Figure shows the two state elements surrounding a block of combinational logic, which operates in a single clock cycle.
- All signals must propagate from state element 1, through the combinational logic, and to state element 2 in the time of one clock cycle.
- The time necessary for the signals to reach state element 2 defines the length of the clock cycle.

Edge-triggered clocking methodology:

- An **edge-triggered clocking methodology** means that any values stored in a sequential logic element are updated only on a clock edge, which is a quick transition from low to high or vice versa.
- An edge-triggered methodology allows us to read the contents of a register, send the value through some combinational logic, and write that register in the same clock cycle.

Control signal

- A signal used for multiplexor selection or for directing the operation of a functional unit; contrasts with a data signal, which contains information that is operated on by a functional unit.

Asserted: The signal is logically high or true.
Deasserted: The signal is logically low or false.

BUILDING A DATAPATH

Datapath

- It is a collection of function units organized in a manner to execute each class of instruction.

Datapath elements

- A unit used to operate on or hold data within a processor is called **datapath element.**
- In the MIPS implementation, the datapath elements include the instruction and data memories, the register file, the ALU, and adders.

How to build a datapath:

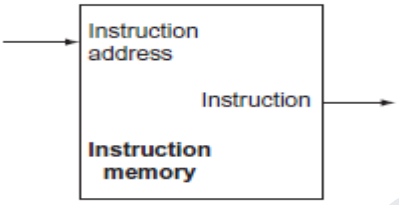
- Datapath design begins in examining the major components required to execute each class of MIPS instructions.
- First we have to know what the data path elements each instruction needs are, and also their control signals.

Stage: 1 [Datapath to fetch instruction and increment PC]

- The following diagram shows the datapath elements needed to fetch an instruction.
- The state elements are the **instruction memory, the program counter and adder.**

Instruction memory

- **Instruction memory** - a memory unit to store the instructions of a program and supply instructions given an address.
- The instruction memory need only provide read access because the datapath does not write instructions.

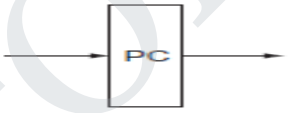


a. Instruction memory

- The output at any time reflects the contents of the location specified by the address input, and no read control signal is needed.

Program counter

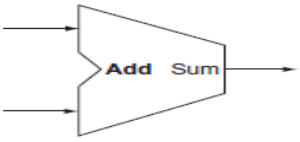
- The register containing the address of the instruction in the program being executed is called **program counter.**
- The program counter is a 32-bit register that is written at the end of every clock cycle and thus does not need a write control signal.



b. Program counter

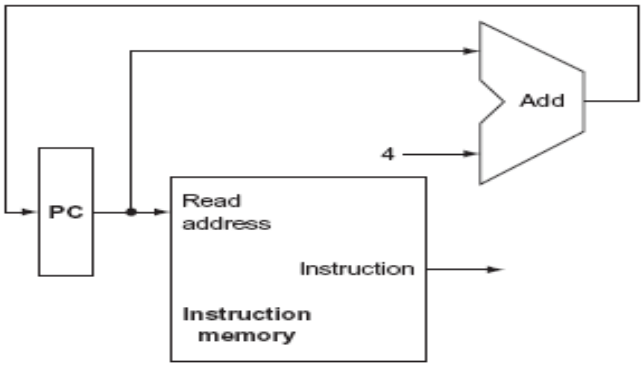
Adder

- **Adder** is used to increment the PC to the address of the next instruction.
- The adder is an ALU wired to always add its two 32-bit inputs and place the sum on its output.



c. Adder

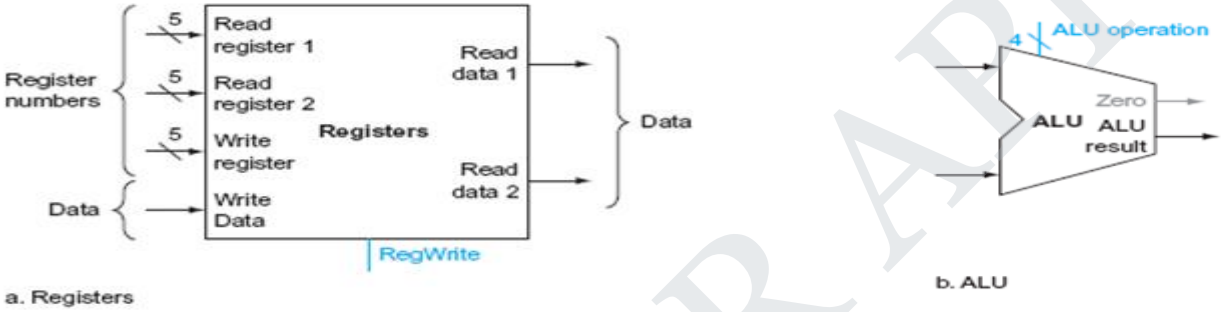
Combined all three elements into single stage



Stage: 2 [Datapath segment for multiport register file and the ALU]

Register File:

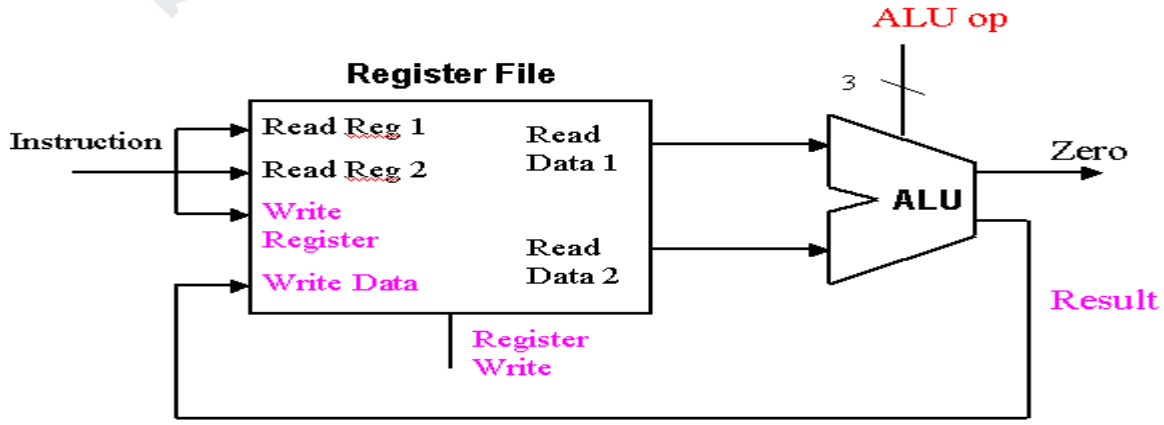
- A **register file** is a collection of registers in which any register can be read or written by specifying the number of the register in the file.
- R-format instructions have three register operands, so we will need to read two data words from the register file and write one data word into the register file for each instruction.
- For each data word to be read from the registers, we need an input to the register file that specifies the register number to be read and an output from the register file that will carry the value that has been read from the registers.
- To write a data word, we will need two inputs: one to specify the register number to be written and one to supply the data to be written into the register.
- The register number inputs are 5 bits wide to specify one of 32 registers ($32 = 2^5$), whereas the data input and two data output buses are each 32 bits wide.



ALU:

- ALU, which takes two 32-bit inputs and produces a 32-bit result, as well as a 1-bit signal if the result is 0. The inputs carrying the register number to the register file are all 5 bits wide, whereas the lines carrying data values are 32 bits wide.
- The operation to be performed by the ALU is controlled with the ALU operation signal, which will be 4 bits control signal.
- ALU provides an output signal that indicates whether the result was 0, we can send the two register operands to the ALU with the control set to do a subtract.
- If the Zero signal out of the ALU unit is asserted, we know that the two values are equal. We will be using it only to implement the equal test of branches.

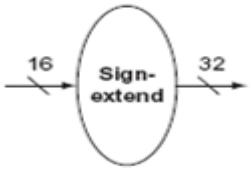
Combined two elements into single stage



Stage: 3 [Datapath segment for Branch Instruction]

Sign-extend

- To increase the size of a data item by replicating the high-order sign bit of the original data item in the high order bits of the larger, destination data item.



b. Sign extension unit

Branch

- A type of branch where the instruction immediately following the branch is always executed independent of whether the branch condition is true or false.

Branch taken

- A branch where the branch condition is satisfied and the program counter (PC) becomes the branch target. All unconditional jumps are taken branches.

Branch not taken or (untaken branch)

- A branch where the branch condition is false and the program counter (PC) becomes the address of the instruction that sequentially follows the branch.

Branch target address

- The address specified in a branch, which becomes the new program counter (PC) if the branch is taken.
- In the MIPS architecture the branch target is given by the sum of the offset field of the instruction and the address of the instruction following the branch.

Example:

- The beq instruction has three operands, two registers that are compared for equality, and a 16-bit off set used to compute the branch target address relative to the branch instruction address. **Ex: beq $t1,$t2,offset.**
- To implement this instruction, we must compute the branch target address by adding the sign-extended offset field of the instruction to the PC.

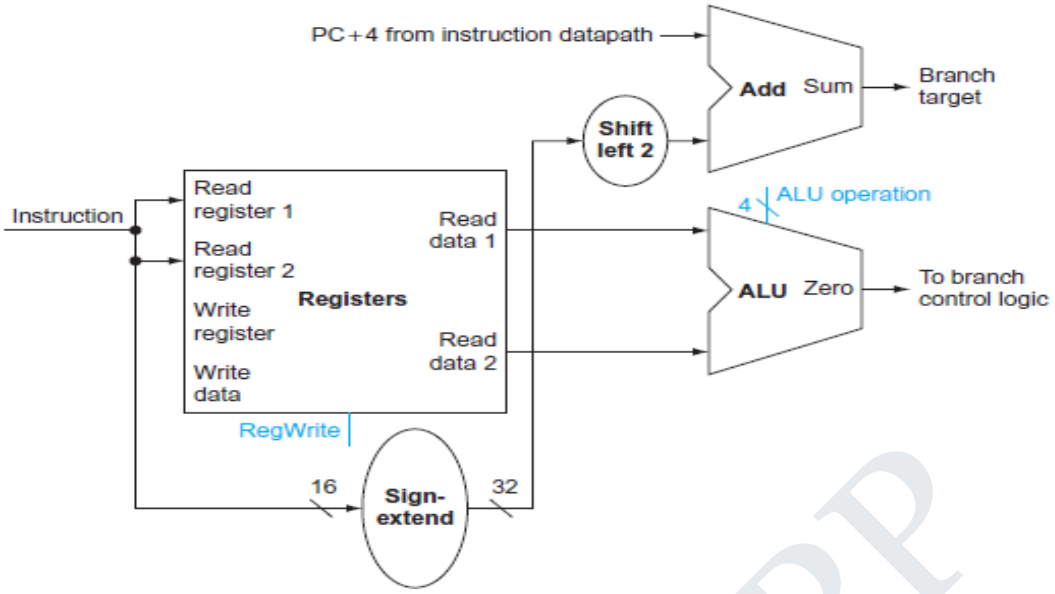
There are two details in the definition of branch instructions.

- The instruction set architecture specifies that the base for the branch address calculation is the address of the instruction following the branch i.e., PC+4 the address of the next instruction.
- The architecture also states that the offset field is shifted left 2 bits so that it is a word off set; this shift increases the effective range of the offset field by a factor of 4.

Branch Target Address = PC+4+offset (Shifted left 2 bits)

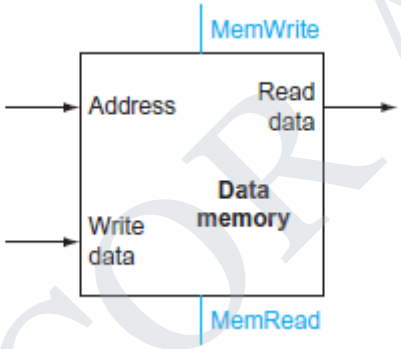
- The branch datapath must perform **two operations:** Compute the branch target address and compare the register contents.
- To compute the branch target address, the branch datapath includes a sign extension unit, shifter and an adder.
- Control logic is used to decide whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU.

Combined Diagram:



Stage: 4 [Datapath Segment for Load Word and Store Word Instructions]

Data Memory

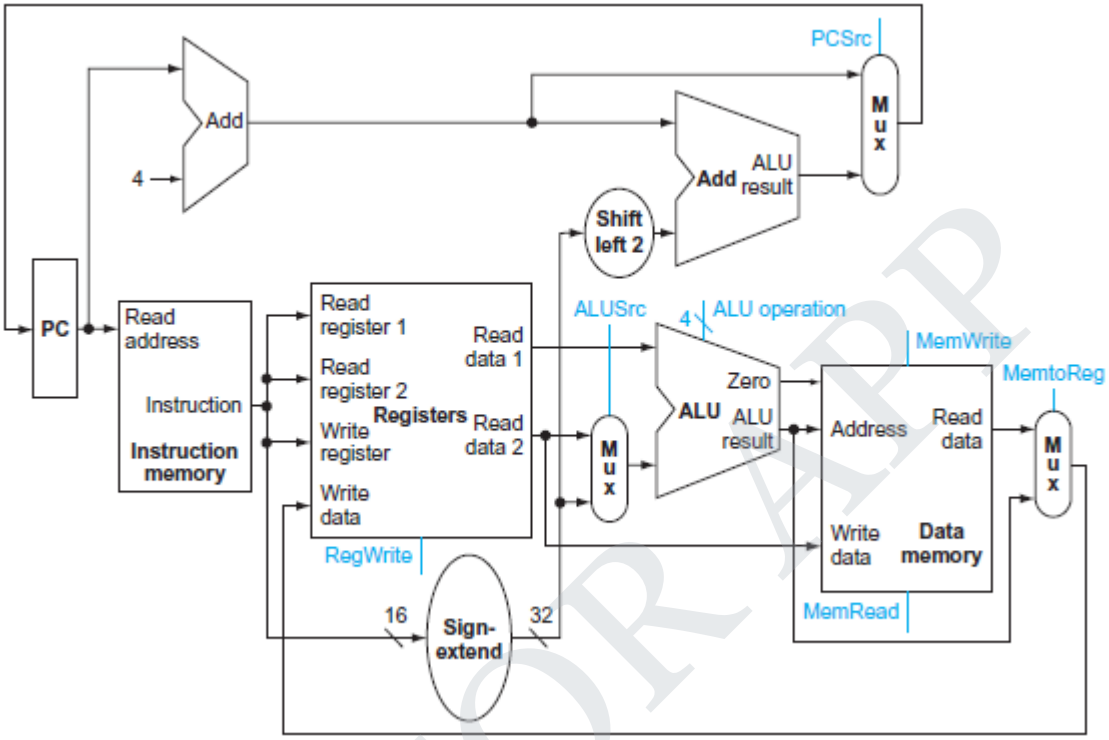


- The data memory unit is a state element with inputs for the address and the write data, and a single output for the read result. It has separate read and write controls to control the read and write operations.
- Although only one of these may be asserted on any given clock. The memory unit needs a read and write control signal.
- Consider the MIPS load word and store word instructions, which have the general form **Ex: lw $t1, offset_value ($t2)** or **sw $t1, offset_value ($t2).**
- These instructions compute a memory address by adding the base register, which is $t2, to the 16-bit signed off set field contained in the instruction.
- If the instruction is a store, the value to be stored must also be read from the register file where it resides in $t1.
- If the instruction is a load, the value read from memory must be written into the register file in the specified register, which is $t1.

Building a Datapath with all the stages:

- Now we can combine all the pieces to make a simple datapath for the core MIPS architecture by adding the datapath for instruction fetch, the datapath from R-type and memory instructions, and the datapath for branches.

- The following figure shows the datapath we obtain by composing the separate pieces. The branch instruction uses the main ALU for comparison of the register operands, so we must keep the adder for computing the branch target address.
- An additional multiplexor is required to select either the sequentially following instruction address (PC + 4) or the branch target address to be written into the PC.
- The control unit must be able to take inputs and generate a write signal for each state element, the selector control for each multiplexor, and the ALU control.



A CONTROL IMPLEMENTATION SCHEME

- This simple implementation covers load word (lw), store word (sw), branch equal (beq), and the arithmetic-logical instructions add, sub, AND, OR, and set on less than.

The ALU Control

- The MIPS ALU in defines the 6 following combinations of four control inputs:

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

- Depending on the instruction class, the ALU will need to perform one of these first five functions.
  - For load word and store word instructions, we use the ALU to compute the memory address by addition.
  - For the R-type instructions, the ALU needs to perform one of the five actions (AND, OR, subtract, add, or set on less than), depending on the value of the 6-bit funct (or function) field in the low-order bits of the instruction
  - For branch equal, the ALU must perform a subtraction.

- We can generate the 4-bit ALU control input using a small control unit that has as inputs the function field of the instruction and a 2-bit control field, which we call ALUOp.
- The 2 bits ALUOp is interpreted as shown in Table.

ALUOp	Action
00	loads and stores
01	subtract for beq
10	determined by the operation encoded in the funct field
11	--

- The following table shows how to set the ALU control inputs based on the 2-bit ALUOp control and the 6-bit function code.

Instruction opcode	ALUOp	Instruction operation	Funct 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

- Here multiple levels of decoding technique is used.

  Adv of using multiple levels of decoding:

  1. It reduces the size of the main control unit.
  2. Use of several smaller units may also increase the speed of the control unit.

Truth table

- From logic, a representation of a logical operation by listing all the values of the inputs and then in each case showing what the resulting outputs should be.

Don't-care term

- An element of a logical function in which the output does not depend on the values of all the inputs. Don't-care terms may be specified in different ways.

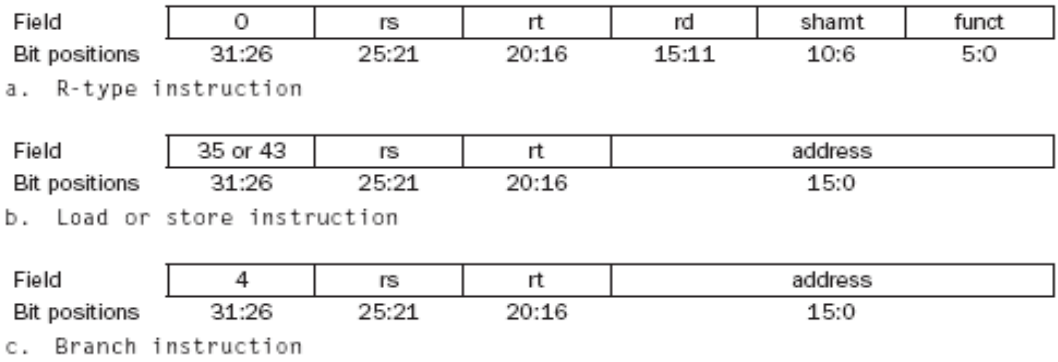
Opcode

- The field that denotes the operation and format of an instruction.
- The op field, is called the opcode, is always contained in bits 31:26. We will refer to this field as op[5:0].

Designing the Main Control Unit

- Designing other controls than ALU controls begins with identifying the fields of an instruction and the control lines that are needed for the datapath.
- There are three instruction classes: the R-type, branch, and load-store instructions. The following diagram shows these formats.
- The two registers to be read are always specified by the rs and rt fields, at positions 25:21 and 20:16. This is true for the R-type instructions, branch equal, and store.
- The 16-bit off set for branch equal, load, and store is always in positions 15:0.
- The destination register is in one of two places. For a load it is in bit positions 20:16 (rt), while for an R-type instruction it is in bit positions 15:11 (rd).

- Thus, we will need to add a multiplexor to select which field of the instruction is used to indicate the register number to be written.



ALUOp		Funct 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

- Instruction format for R-format instructions, which all have an opcode of 0. These instructions have three register operands: rs, rt, and rd. Fields rs and rt are sources, and rd is the destination. Ex: add, sub, AND, OR, and slt.
- The ALU function is in the funct field and is decoded by the ALU control design.
- Instruction format for load (opcode = 35ten) and store (opcode = 43ten) instructions.
- The register rs is the base register(25:21) that is added to the 16-bit address field to form the memory address. For loads, rt is the destination register for the loaded value. For stores, rt is the source register whose value should be stored into memory.
- Instruction format for branch equal (opcode =4). The registers rs and rt are the source registers that are compared for equality.
- The 16-bit address field is sign-extended, shifted, and added to the PC + 4 to compute the branch target address. The following table describes seven other control lines.
- These nine control signals (seven from above table and two for ALUOp) can now be set on the basis of six input signals to the control unit, which are the opcode bits 31 to 26.
- When the 1-bit control to a two way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input.

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.

- The following diagram shows the datapath with the control unit and the control signals.

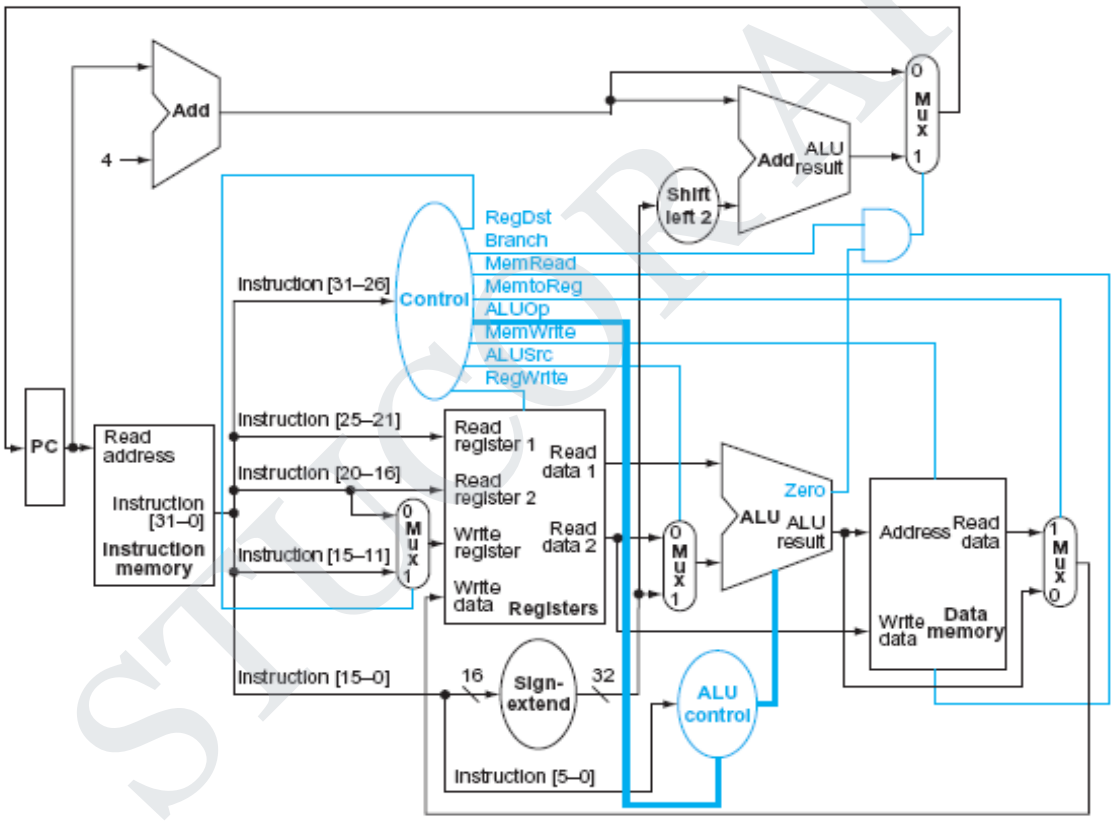


Figure: The simple datapath with the control unit

- The input to the control unit is the 6-bit opcode field from the instruction.
- The outputs of the control unit consist of three 1-bit signals that are used to control multiplexors (RegDst, ALUSrc, and MemtoReg).
- Three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite), a 1-bit signal used in determining whether to possibly branch (Branch), and a 2-bit control signal for the ALU (ALUOp).
- An AND gate is used to combine the branch control signal and the Zero output from the ALU; the AND gate output controls the selection of the next PC.

- Notice that PCSrc is now a derived signal, rather than one coming directly from the control unit.The control lines is completely determined by the opcode fields of the instruction as shown below

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

- The first row of the table corresponds to the R-format instructions (add, sub, AND, OR, and slt). For all these instructions, the source register fields are rs and rt, and the destination register field is rd; this defines how the ALUSrc and RegDst are set.
- An R-type instruction writes a register (Reg-Write = 1), but neither reads nor writes data memory.
- When the Branch control signal is 0, the PC is unconditionally replaced with PC + 4; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high.
- The ALUOp field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct field.
- The second and third rows of this table give the control signal settings for lw and sw. These ALUSrc and ALUOp fields are set to perform the address calculation.
- The MemRead and MemWrite are set to perform the memory access. Finally, RegDst and RegWrite are set for a load to cause the result to be stored into the rt register.
- The branch instruction is similar to an R-format operation, since it sends the rs and rt registers to the ALU. The ALUOp fi eld for branch is set for a subtract (ALU control = 01), which is used to test for equality.
- Thus, the entry MemtoReg in the last two rows of the table is replaced with X for don't care. Don't cares can also be added to RegDst when RegWrite is 0

Finalizing Control

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

- The top half of the table gives the combinations of input signals that correspond to the four opcodes, one per column, that determine the control output settings.
- The bottom portion of the table gives the outputs for each of the four opcodes. Thus, the output RegWrite is asserted for two different combinations of the inputs.

Single-cycle implementation: An implementation in which an instruction is executed in one clock cycle called single clock cycle implementation.

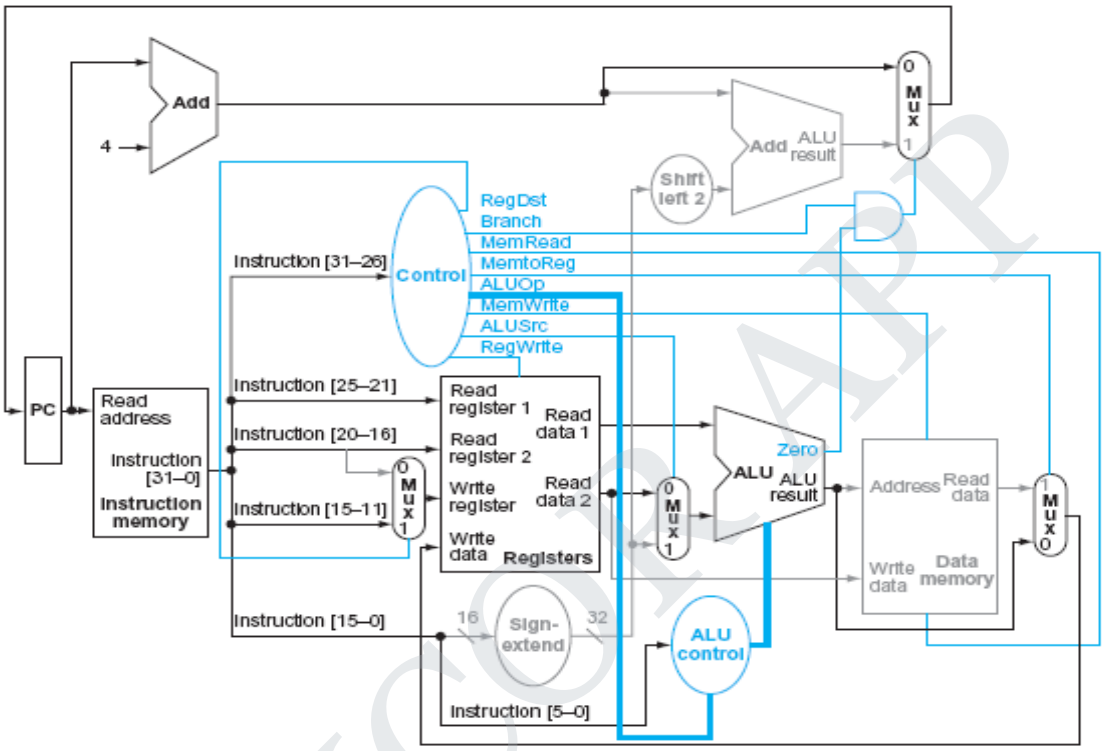
Operation of the Datapath **Example: add $t1,$t2,$t3**

Step:1 The instruction is fetched, and the PC is incremented.

Step:2 Two registers, $t2 and $t3, are read from the register fi le; also, the main control unit computes the setting of the control lines during this step.

Step:3 The ALU operates on the data read from the register fi le, using the function code (bits 5:0, which is the funct field, of the instruction) to generate the ALU function.

Step:4 The result from the ALU is written into the register fi le using bits 15:11 of the instruction to select the destination register ($t1).



. **Example:** **lw $t1, offset ($t2)**

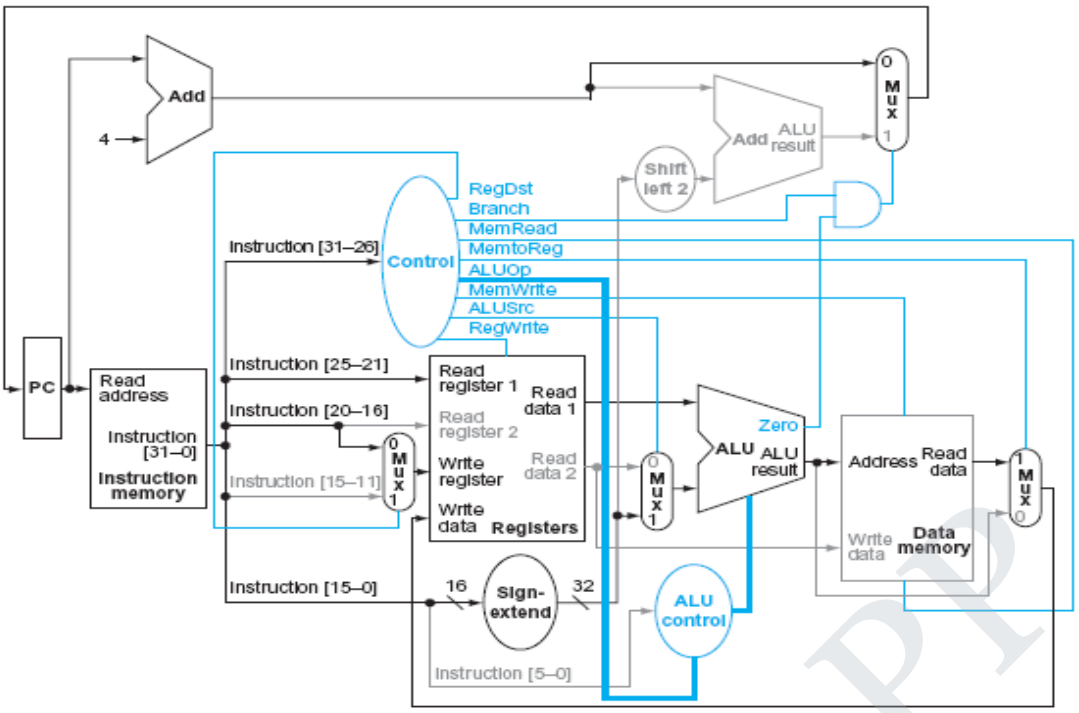
Step:1 An instruction is fetched from the instruction memory, and the PC is incremented.

Step:2 A register ($t2) value is read from the register file.

Step: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).

Step:4 The sum from the ALU is used as the address for the data memory.

Step:5 The data from the memory unit is written into the register file; the register destination is given by bits 20:16 of the instruction ($t1).



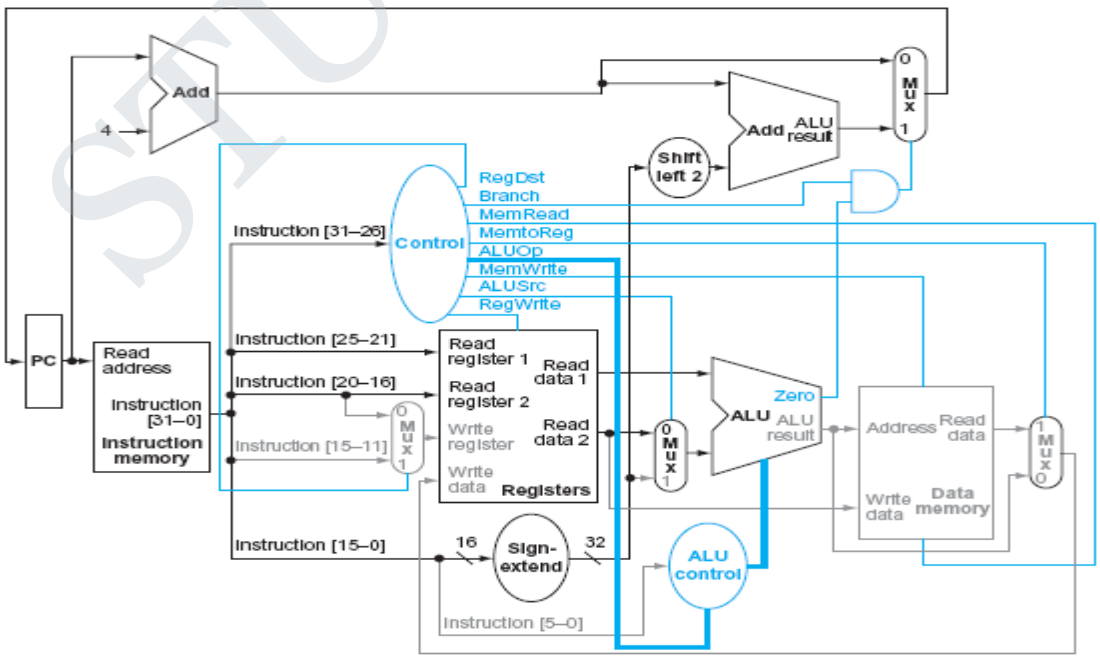
Example: **beq $t1, $t2, offset**

Step:1 An instruction is fetched from the instruction memory, and the PC is incremented.

Step:2 Two registers, $t1 and $t2, are read from the register fi le.

Step:3 The ALU performs a subtract on the data values read from the register file. The value of PC + 4 is addedto the sign-extended, lower 16 bits of the instruction (offset) shift ed left by two; the result is the branch target address.

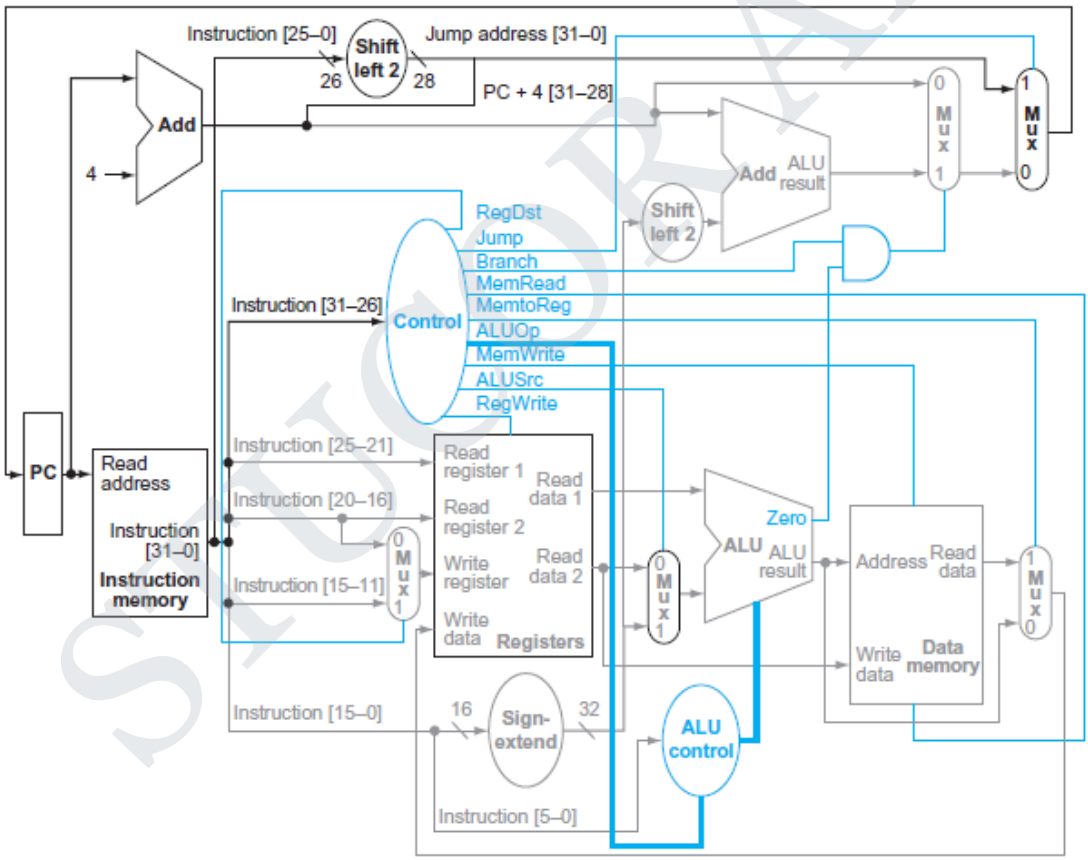
Step:4 The Zero result from the ALU is used to decide which adder result to store into the PC.



IMPLEMENTING JUMPS

Field	000010	address
Bit positions	31:26	25:0

- The jump instruction, looks somewhat like a branch instruction but computes the target PC differently and is not conditional.
- The upper 4 bits of the address that should replace the PC come from the PC of the jump instruction plus 4.
- Thus, we can implement a jump by storing into the PC the concatenation of the upper 4 bits of the current PC + 4 (these are bits 31:28 of the sequentially following instruction address)
- An additional multiplexor (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one.
- This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address.



AN OVERVIEW OF PIPELINING

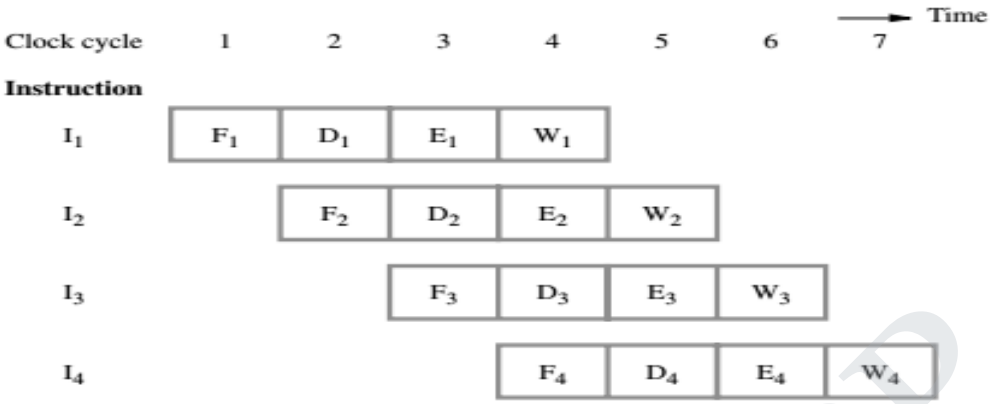
Pipelining:

An implementation technique in which multiple instructions are overlapped in execution is called pipeline. The different pipelining stages are,

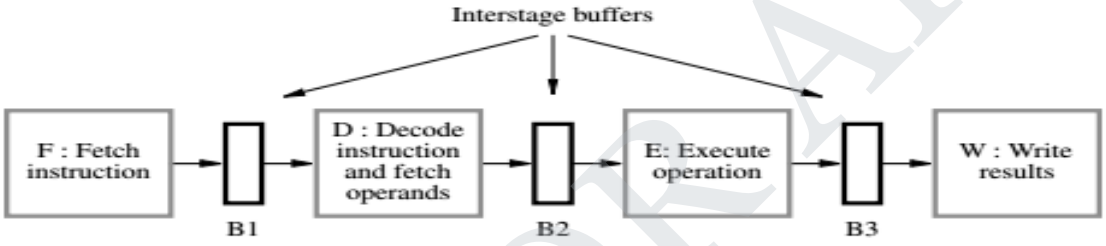
1. **Fetch** - Fetch instruction from memory.
2. **Decode** - Read registers while decoding the instruction. The regular format of MIPS instructions allow reading and decoding to occur simultaneously.

3. **Execute** - Execute the operation or calculate an address.
4. **Access** - Access an operand in data memory.
5. **Write** - Write the result into a register.

Four stage Instruction Pipelining



(a) Instruction execution divided into four steps



(b) Hardware organization

Hardware units are organized into stages:

- Execution in each stage takes exactly 1 clock period. Stages are separated by pipeline registers that preserve and pass partial results to the next stage.

Performance = complexity + cost.

- The pipeline approach brings additional expense plus its own set of problems and complications, called hazards.

Pipeline Performance (or) Speedup

- The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages.
- If all the stages take about the same amount of time and there is enough work to do, then the speed-up due to pipelining is equal to the number of stages in the pipeline.
- If the stages are perfectly balanced, then the time between instructions on the pipelined processor – assuming ideal conditions – is equal to

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instruction}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

- A pipelined processor allows multiple instructions to execute at once, and each instruction uses a different functional unit in the datapath. This increases throughput, so programs can run faster. One instruction can finish executing on every clock cycle, and simpler stages also lead to shorter cycle times.

Example – Single-Cycle versus Pipelined Performance

- Consider a simple program segment consists of eight instructions: lw, sw, add, sub, AND, OR, slt and beq. Compare the average time between instructions of a single-cycle implementation, in which all instructions take 1 clock cycle, to a pipelined implementation. The operation times for the major functional units in this example are 200 ps for memory access, 200 ps for ALU operation, and 100 ps for register file read or write.
- The following table shows the time required for each of the eight instructions. The single-cycle design must allow for the slowest instruction is lw – so the time required for every instruction is 800 ps. Thus, the time between the first and fourth instructions in the non-pipelined design is 3 x 800 ns or 2400 ps.
- Assume that following table shows the time taken by each and every stages of pipeline for different instruction

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

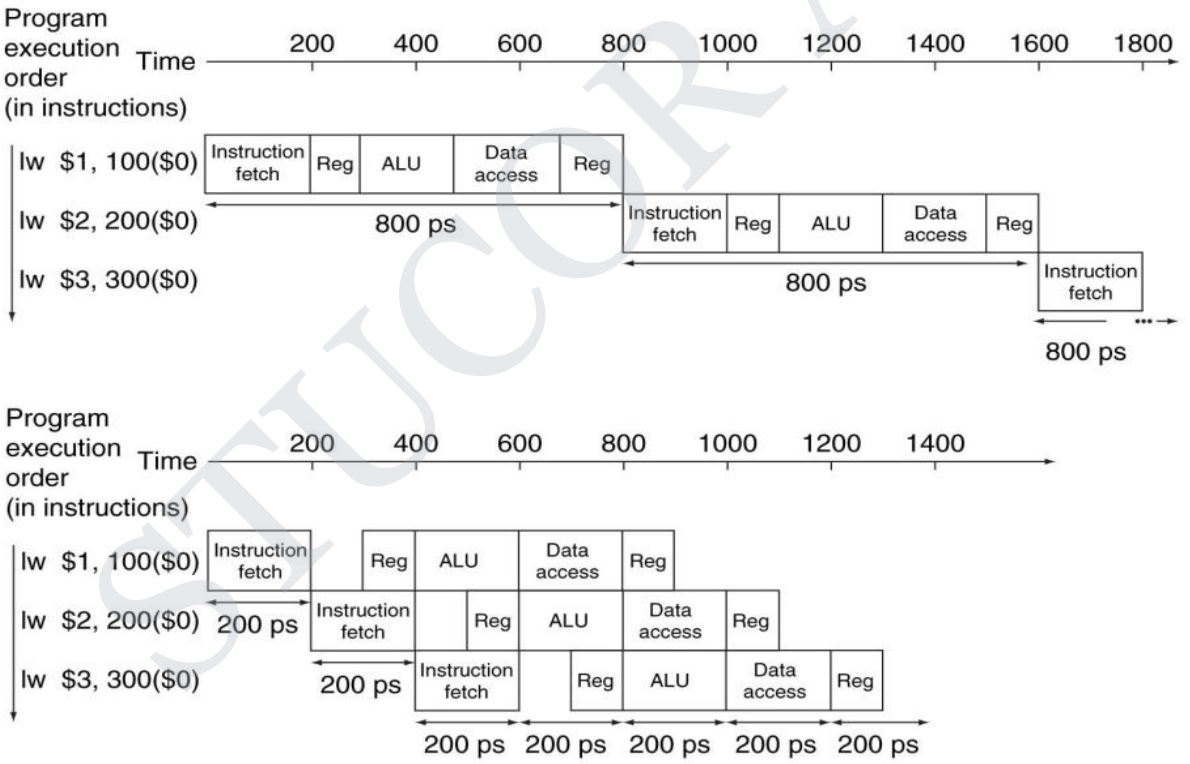


Figure: Single-Cycle, Non-Pipelined Execution in top versus Pipelined Execution in bottom.

- By comparing above two diagram, it is clear that pipeline process is best and it take reduce time to execute the instruction.
- Pipelining improves performance by increasing instruction throughput, as opposed to decreasing the execution time of an individual instruction.

Six stages in the pipeline:

1. Fetch instruction: Instructions are fetched from the memory into a temporary buffer before it gets executed.

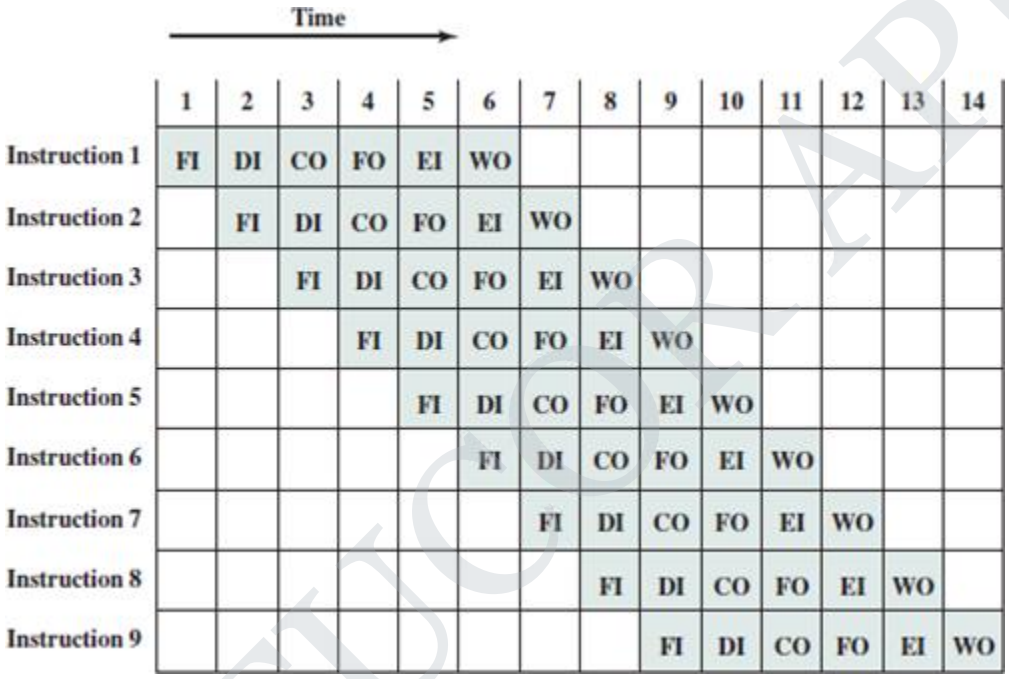
2. Decode instruction: The instruction is decoded by the CPU so that the necessary op codes and operands can be determined.

3. Calculate operand: Based on the addressing scheme used, either operands are directly provided in the instruction or the effective address has to be calculated.

4. Fetch Operand: Once the address is calculated, the operands need to be fetched from the address that was calculated. This is done in this phase.

5. Execute Instruction: The instruction can now be executed.

6. Write operand: Once the instruction is executed, the result from the execution needs to be stored or written back in the memory.



PIPELINED DATAPATH AND CONTROL

The division of an instruction into five stages means a five-stage pipeline, which in turn means that up to five instructions will be in execution during any single clock cycle.

1. IF: Instruction fetch
2. ID: Instruction decode and register file read
3. EX: Execution or address calculation
4. MEM: Data memory access
5. WB: Write back

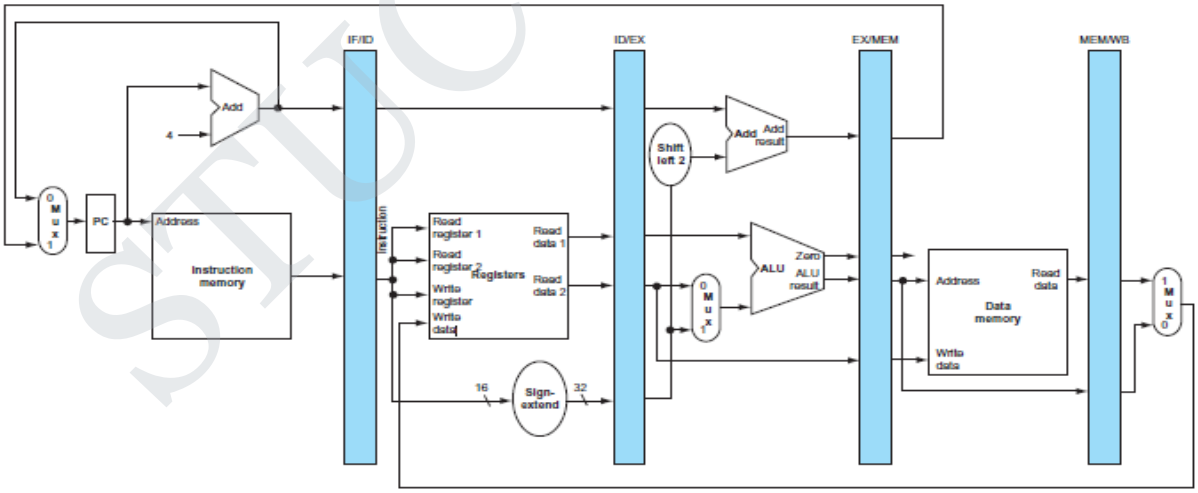
- Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the update of the PC and the write-back step, shown in color, which sends either the ALU result or the data from memory to the left to be written into the register file.
- There are, however, two exceptions to this left -to-right flow of instructions:

1. The write-back stage, which places the result back into the register file in the middle of the datapath.
2. The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage Data fl owing from right to left does not affect the current instruction;

- The first right-to-left flow of data can lead to data hazards and the second leads to control hazards.

The pipelined version of the datapath:

- The following diagram shows the pipelined datapath with the pipeline registers highlighted.
- All instructions advance during each clock cycle from one pipeline register to the next. The registers are named for the two stages separated by that register.
- For example, the pipeline register between the IF and ID stages is called IF/ID.
- Notice that there is no pipeline register at the end of the write-back stage. All instructions must update some state in the processor, the register file, memory, or the PC.
- For example, a load instruction will place its result in 1 of the 32 registers, and any later instruction that needs that data will simply read the appropriate register.
- The pipeline registers separate each pipeline stage. They are labeled by the stages that they separate; For example, the first is labeled IF/ID because it separates the instruction fetch and instructions decode stages.
- The registers must be wide enough to store all the data corresponding to the lines that go through them.
- For example, the IF/ID register must be 64 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address.

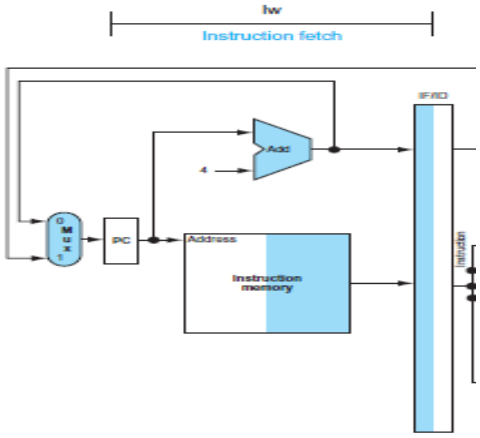


Example: Load Instruction (lw) **lw $s1, 100($s0)**

1. *Instruction fetch:*

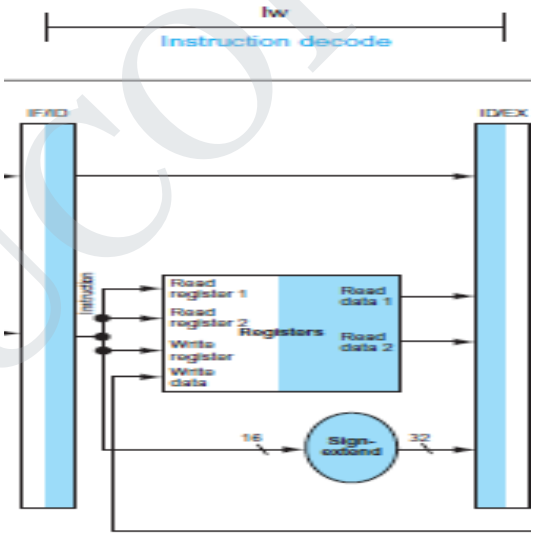
- The top portion of Figure shows the instruction being read from memory using the address in the PC and then being placed in the IF/ID pipeline register.
- The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle.

- This incremented address is also saved in the IF/ID pipeline register in case it is needed later for an instruction, such as beq.
- The computer cannot know which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline.



2. Instruction decode and register file read:

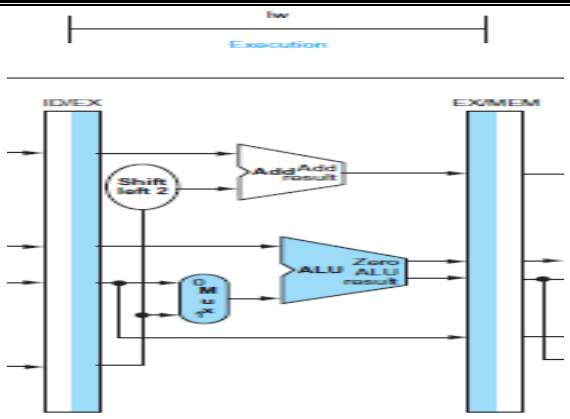
- The bottom portion of Figure shows the instruction portion of the IF/ID pipeline register supplying the 16-bit immediate field, which is sign-extended to 32 bits, and the register numbers to read the two registers.
- All three values are stored in the ID/EX pipeline register, along with the incremented PC address.
- We again transfer everything that might be needed by any instruction during a later clock cycle.



- There is no confusion when reading and writing registers, because the contents change only on the clock edge.
- Although the load needs only the top register in stage 2, the processor doesn't know what instruction is being decoded, so it sign-extends the 16-bit constant and reads both registers into the ID/EX pipeline register.
- We don't need all three operands, but it simplifies control to keep all three.

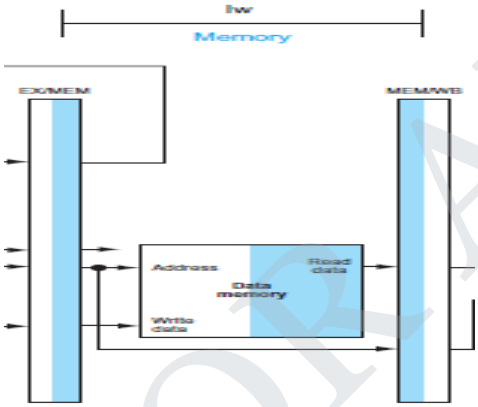
3. Execute or address calculation:

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



4. Memory access:

- The top portion of figure shows 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 register.



5. Write-back:

- The bottom portion of figure shows the final step: Data memory is read using the address in the EX/MEM pipeline registers, and the data is placed in the MEM/WB pipeline register.
- Next, data is read from the MEM/WB pipeline register and written into the register file in the middle of the datapath.

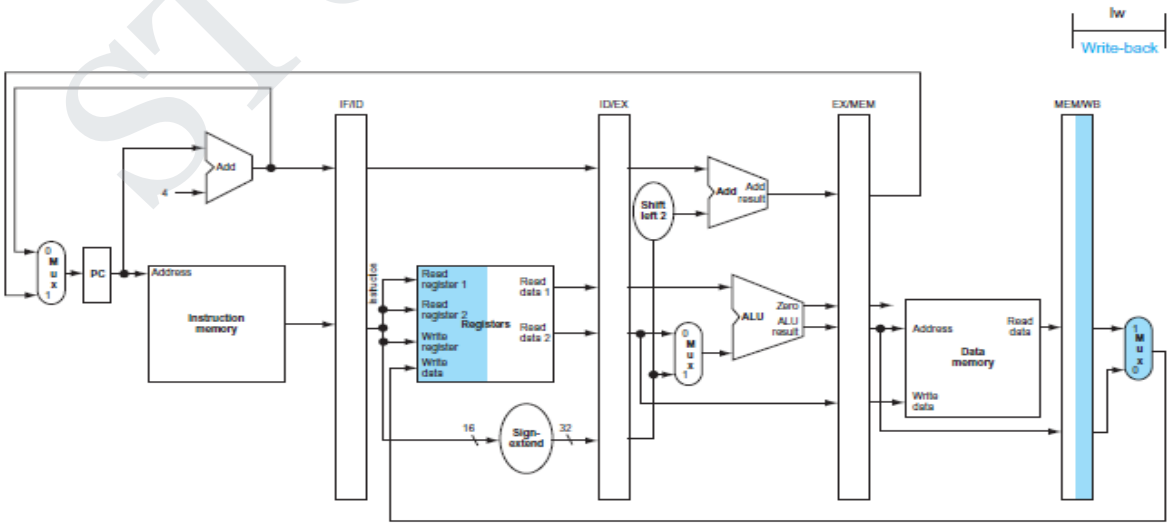
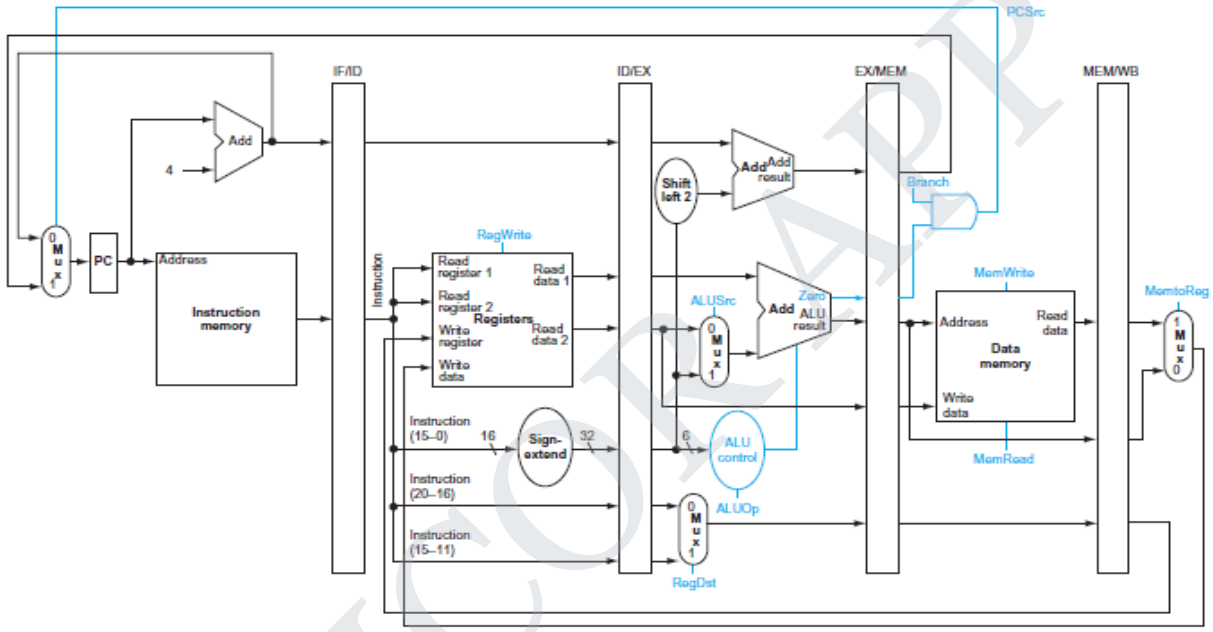


Figure: Combined Pipeline Datapath Diagram

PIPELINED CONTROL

- To specify control for the pipeline, we need only set the control values during each pipeline stage. Because each control line is associated with a component active in only a single pipeline stage, we can divide the control lines into five groups according to the pipeline stage.
- ***Instruction fetch:*** The control signals to read instruction memory and to write the PC are always asserted, so there is nothing special to control in this pipeline stage.
- ***Instruction decode/register file read:*** As in the previous stage, the same thing happens at every clock cycle, so there are no optional control lines to set.
- ***Execution/address calculation:*** The signals 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.



- This datapath borrows the control logic for PC source, register destination number, and ALU control. Note that we now need the 6-bit funct field (function code) of the instruction in the EX stage as input to ALU control, so these bits must also be included in the ID/EX pipeline register.
- Recall that these 6 bits are also the 6 least significant bits of the immediate field in the instruction, so the ID/EX pipeline register can supply them from the immediate field since sign extension leaves these bits unchanged.

Instruction opcode	ALUOp	Instruction operation	Function code	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

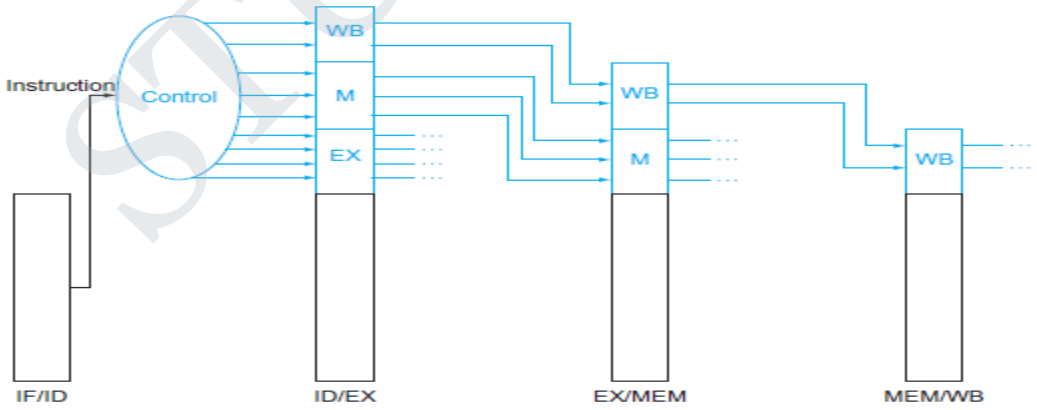
- The function of each of seven control signals is defined. The ALU control lines (ALUOp) are defined in the second column.

- When a 1-bit control to a 2-way multiplexor is asserted, the multiplexor selects the input corresponding to 1.
- Otherwise, if the control is deasserted, the multiplexor selects the 0 input.
- Note that PCSrc is controlled by an AND gate in if the Branch signal and the ALU Zero signal are both set, then PCSrc is 1; otherwise, it is 0. Control sets the Branch signal only during a beq instruction; otherwise, PCSrc is set to 0.

Signal name	Effect when deasserted (0)	Effect when asserted (1)
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.

Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

- **Memory access:** The control lines set in this stage are Branch, MemRead, and MemWrite. The branch equal, load, and store instructions set these signals, respectively. Recall that PCSrc selects the next sequential address unless control asserts Branch and the ALU result was 0.
- **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.



- The control lines for the final three stages. Note that four of the nine control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/ WB for use in the WB stage.

Pipeline Hazards

The condition that makes the pipeline to stall is called Hazards. The idle period in the pipeline execution is called Stall or Bubble.

Types of hazards:

1. Structural Hazard
2. Data Hazard
3. Control Hazard

1. Structural Hazard

- When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.

2. Data Hazards

- **Data hazards** occur when the pipeline must be stalled because one step must wait for another to complete.
- When a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.
- This is because of data dependence between the instructions that has been overlapped.

Consider the following example

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

- In the above instruction one of the operand ($s0) of the sub instruction will be fetched only after the add instruction store it result in the same register ($s0).
- So that sub instruction is stalled for some clock cycle which makes the pipeline process to waste the some clock cycle.

3. Control Hazards

- It is also called branch hazard. When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.

HANDLING DATA HAZARD:

Data hazard can be handled by using three methods.

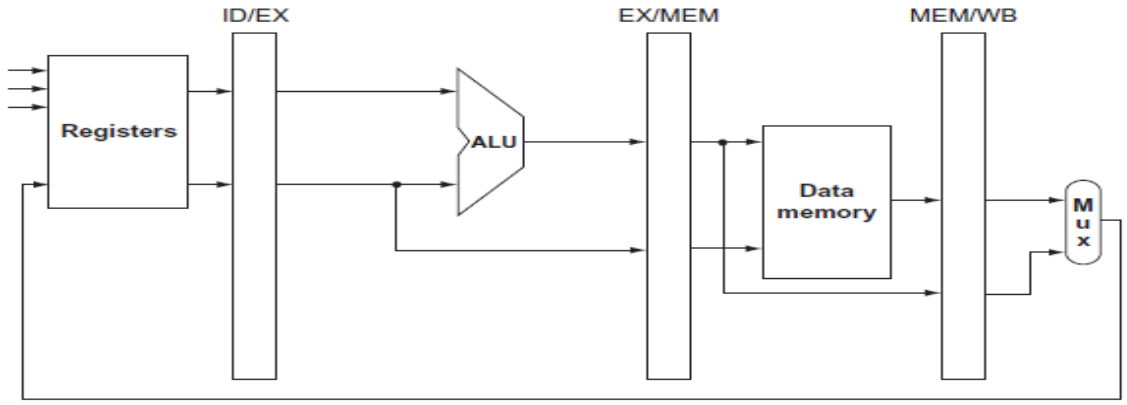
Solution to data hazard:

1. Operand forwarding(Hardware)
2. Reordering Code (software)
3. By using stall

1. Operand forwarding (Hardware):

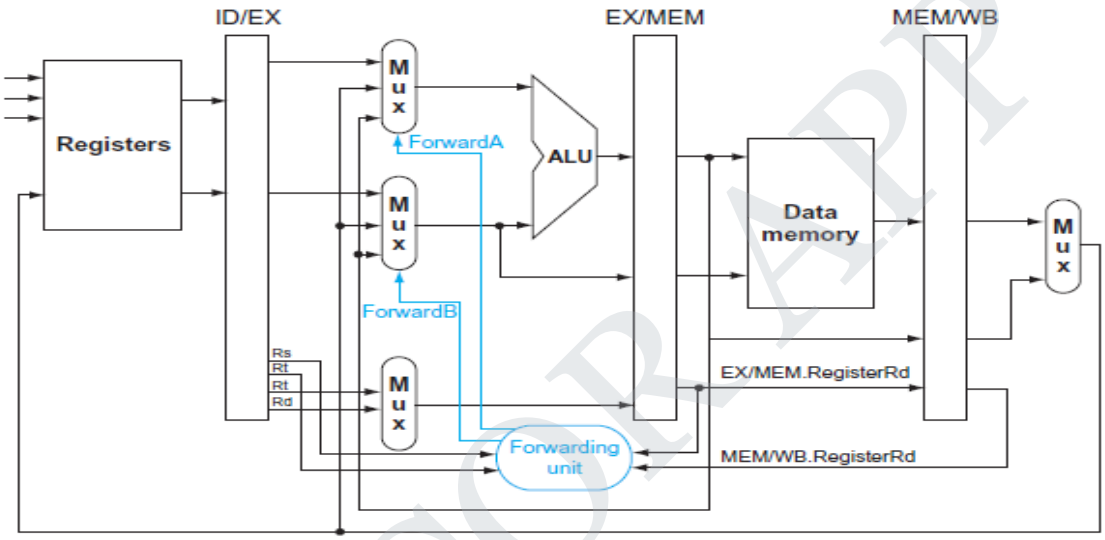
- The primary solution is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard.
- Forwarding Also called **bypassing.** A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer visible registers or memory.

No forwarding:



a. No forwarding

With Forwarding:



b. With forwarding

- On the top figure are the ALU and pipeline registers before adding forwarding. On the bottom figure, the multiplexors have been expanded to add the forwarding paths, and we show the forwarding unit.
- The new hardware is shown in color. This figure is a stylized drawing, however, leaving out details from the full datapath such as the sign extension hardware.
- Note that the ID/EX.Register Rt field is shown twice, once to connect to the Mux and once to the forwarding unit, but it is a single signal.

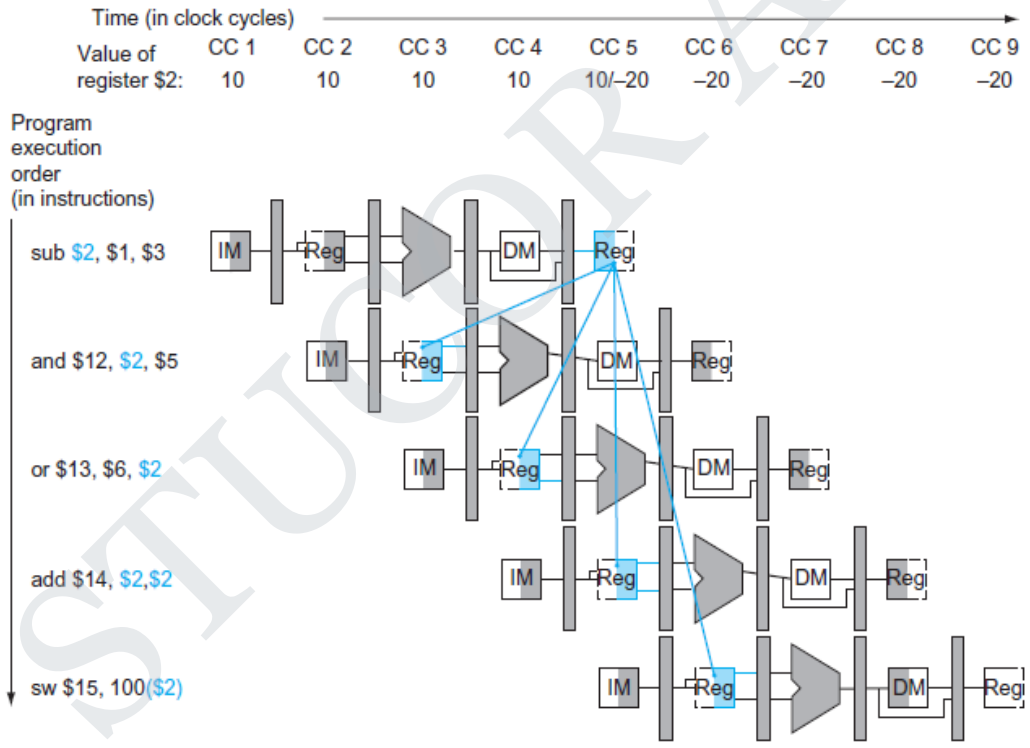
The control values for the forwarding multiplexors in the above diagram

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	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.

Pipeline datapath and control for data hazard:

- In the data is still being read from memory in clock cycle 4 while the ALU is performing the operation for the following instruction.
- Something must stall the pipeline for the combination of load followed by an instruction that reads its result.
- Hence, in addition to a forwarding unit, we need a hazard detection unit. It operates during the ID stage so that it can insert the stall between the load and its use.
- Pipelined dependences in a five-instruction sequence using simplified data paths to show the dependences
- All the dependent actions are shown in color, and "CC 1" at the top of the figure means clock cycle 1.
- The first instruction writes into $2, and all the following instructions read $2. This register is written in clock cycle 5, so the proper value is unavailable before clock cycle 5.
- The colored lines from the top datapath to the lower ones show the dependences. Those that must go backward in time are pipeline data hazards.

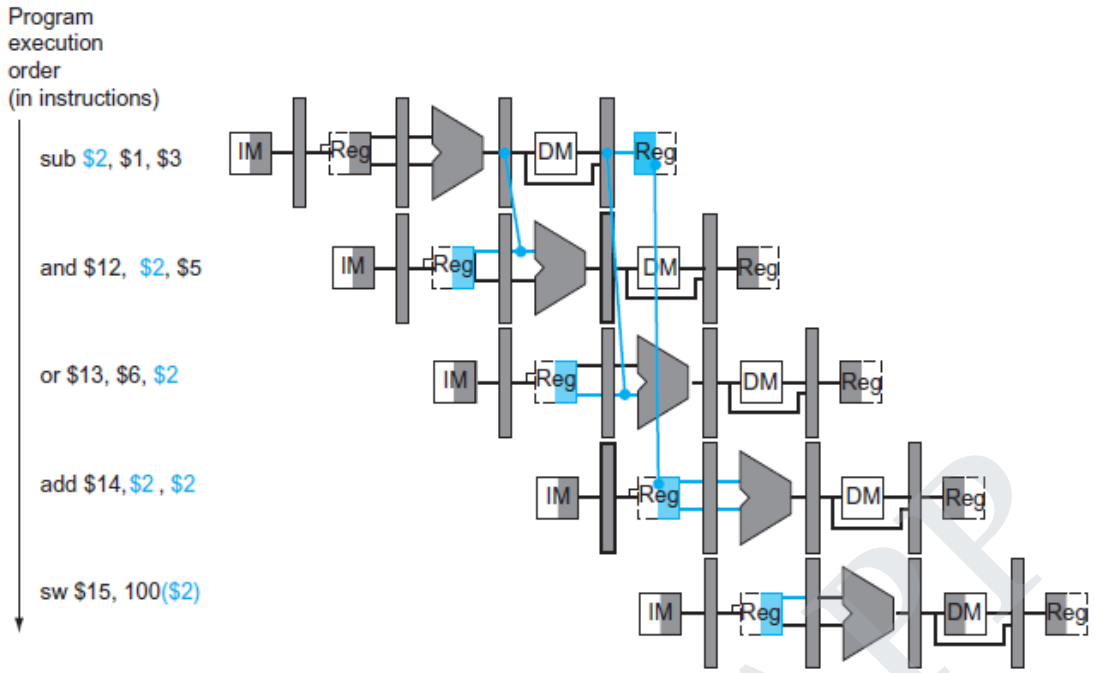
Data Dependences without data forwarding Technique:



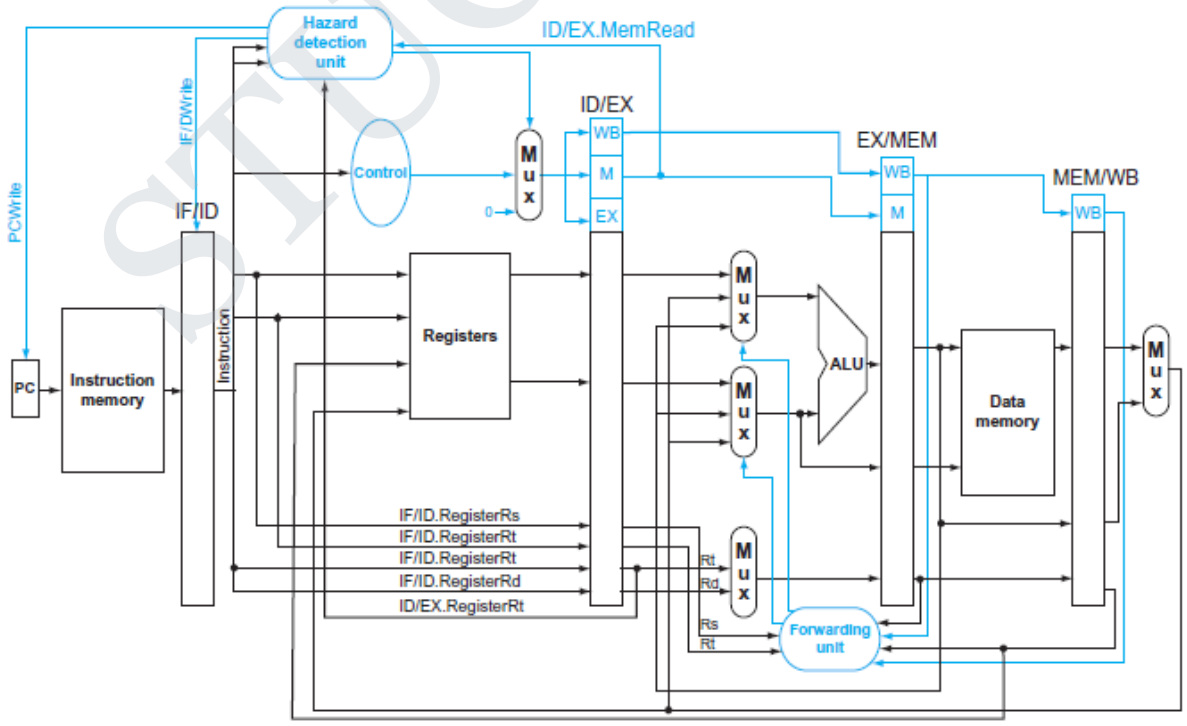
Data forwarding Technique:

- The dependences between the pipeline registers move forward in time, so it is possible to supply the inputs to the ALU needed by the AND instruction and OR instruction by forwarding the results found in the pipeline registers.
- The values in the pipeline registers show that the desired value is available before it is written into the register file.
- We assume that the register file forwards values that are read and written during the same clock cycle, so the add does not stall, but the values come from the register file instead of a pipeline register.

- Register file "forwarding" that is, the read gets the value of the write in that clock cycle.



- The following diagram highlights the pipeline connections for both the hazard detection unit and the forwarding unit.
- As before, the forwarding unit controls the ALU multiplexors to replace the value from a general-purpose register with the value from the proper pipeline register.
- The hazard detection unit controls the writing of the PC and IF/ID registers plus the multiplexor that chooses between the real control values and all 0s.
- The hazard detection unit stalls and deasserts the control fields if the load-use hazard test above is true.



2. Reordering Code to Avoid Pipeline Stalls:

Consider the following code segment in C:

a = b + e;

c = b + f;

Here is the generated MIPS code for this segment, assuming all variables are in memory and are addressable as off sets from $t0:

Before Reorder:

```
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1,$t2
sw $t3, 12($t0)
lw $t4, 8($t0)
add $t5, $t1,$t4
sw $t5, 16($t0)
```

Find the hazards in the preceding code segment and reorder the instructions to avoid any pipeline stalls.

- Both add instructions have a hazard because of their respective dependence on the immediately preceding lw instruction.
- Notice that bypassing eliminates several other potential hazards, including the dependence of the first adds on the first lw and any hazards for store instructions.
- Moving up the third lw instruction to become the third instruction eliminates both hazards:

After Reorder:

```
lw $t1, 0($t0)
lw $t2, 4($t0)
lw $t4, 8($t0)
add $t3, $t1,$t2
sw $t3, 12($t0)
add $t5, $t1,$t4
sw $t5, 16($t0)
```

3. Data hazard solved by using Stall

Pipeline stall

Pipeline stall also called bubble. A stall initiated in order to resolve a hazard.

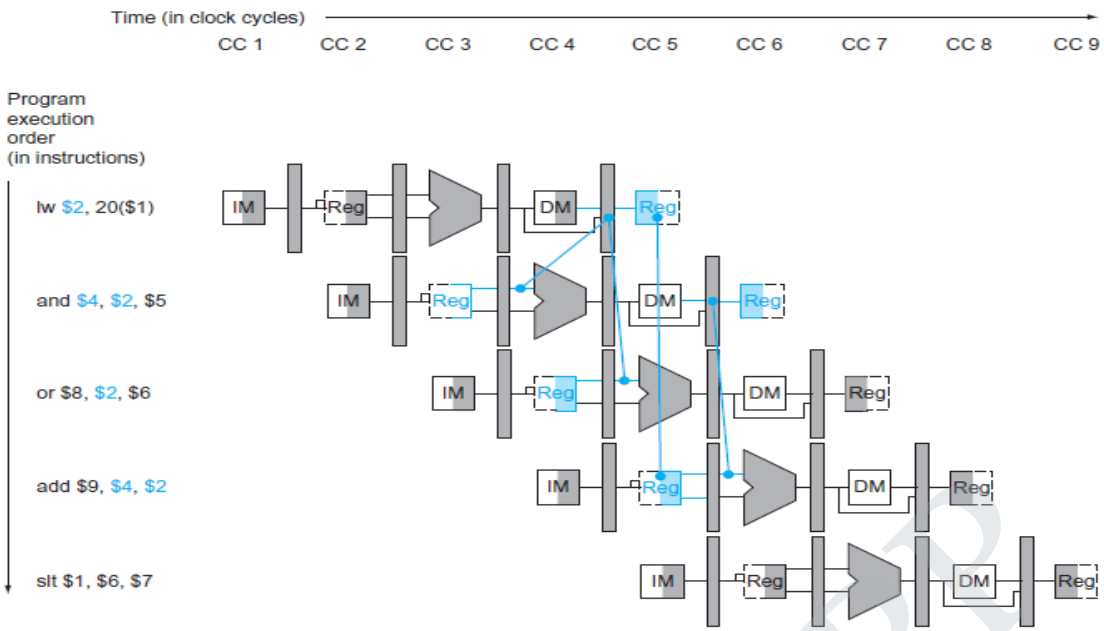
Load-use data hazard

A specific form of data hazard in which the data being loaded by a load instruction has not yet become available when it is needed by another instruction.

nop

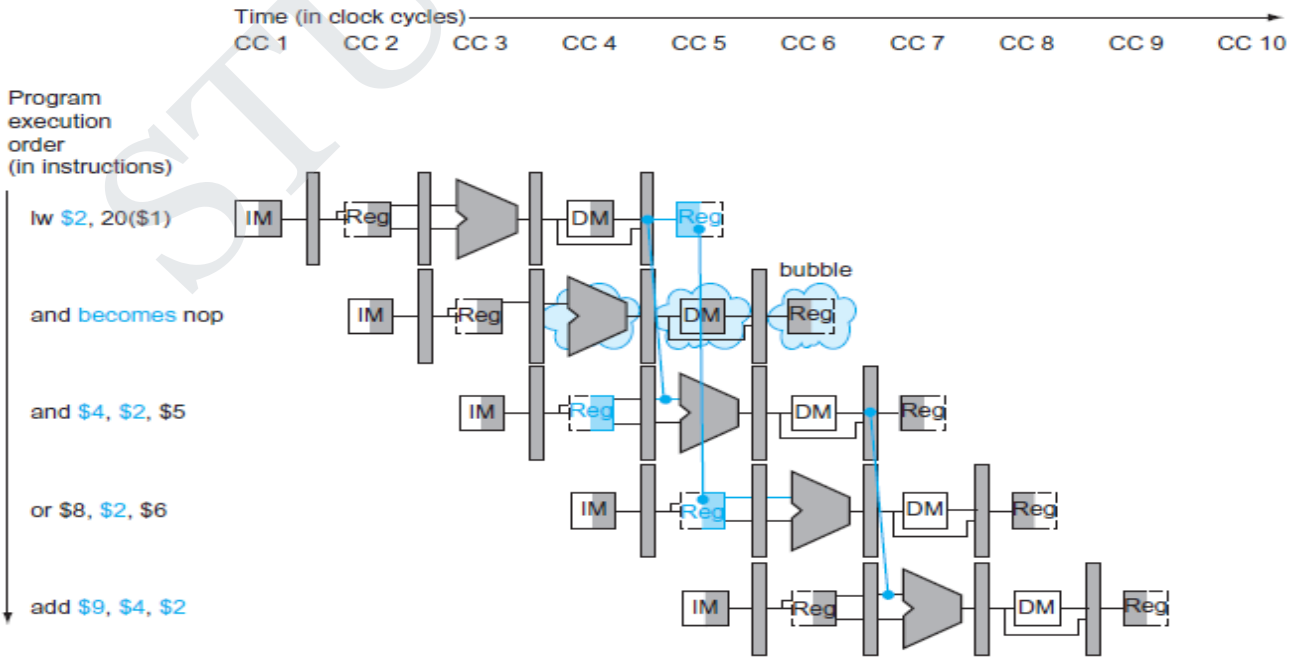
An instruction that does no operation to change state.

Data hazard without stall:



- The following diagram shows the AND instruction is turned into a nop and all instructions beginning with the AND instructions are delayed one cycle.
- In this example, the hazard forces the AND and OR instructions to repeat in clock cycle 4 what they did in clock cycle 3: AND reads registers and decodes, and OR is refetched from instruction memory.
- A bubble is inserted beginning in clock cycle 4, by changing the and instruction to a nop. Note that the and instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5.
- Likewise the OR instruction is fetched in clock cycle 3, but its ID stage is delayed until clock cycle 5. After insertion of the bubble, all the dependences go forward in time and no further hazards occur.

Data hazard with stall:

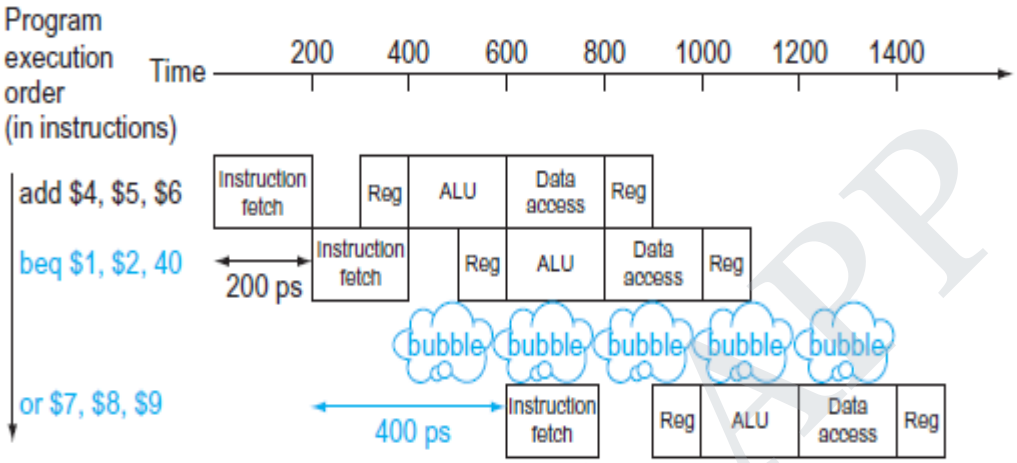


3. HANDLING CONTROL HAZARDS

- It is also called branch hazard. When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.

Performance of "Stall on Branch"

- Estimate the impact on the clock cycles per instruction (CPI) of stalling on branches. Assume all other instructions have a CPI of 1.



- This example assumes the conditional branch is taken, and the instruction at the destination of the branch is the OR instruction.
- There is a one-stage pipeline stall, or bubble, after the branch.

Two schemes for resolving control hazards

1. Branch prediction
2. Delayed branching

1. Branch Prediction

- Prediction techniques can be used to check whether a branch will be valid or not valid. These techniques reduce the branch penalty.
- A method of resolving a branch hazard that assumes a given outcome for the branch called branch prediction.
- The common prediction techniques are:
    - Predict Never Taken
    - Predict Always Taken
    - Predict By Opcode
    - Taken or Not Taken Branch
    - Branch History Table
- In the first two approaches if prediction is wrong a page fault or prediction violation error occurs. The processor then halts prefetching and fetches the instruction from the desired address.
- In the third approach, the prediction is based on the opcode of the branch instruction.
- The fourth and Fifth approaches are dynamic. They depend on history of the previously executed conditional branch instruction.

Branch prediction Strategies:

(i). Static Branch Prediction Strategy

(ii). Dynamic Branch Prediction Strategy

(i). Static Branch Prediction

- In this strategy branch can be predicted based on branch code types statically. This means that the probability of branch with respect to a particular branch type is used to predict the branch. This branch strategy may not produce accurate results every time.
- One improvement over branch stalling is to predict that the branch will not be taken and thus continue execution down the sequential instruction stream.
- If the branch is taken, the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target.
- If branches are untaken half the time, and if it costs little to discard the instructions, this optimization halves the cost of control hazards.
- Discarding instructions, then, means we must be able to flush instructions in the IF, ID, and EX stages of the pipeline.
- We already have the PC value and the immediate field in the IF/ID pipeline register, so we just move the branch adder from the EX stage to the ID stage;
- During ID, we must decode the instruction, decide whether a bypass to the equality unit is needed, and complete the equality comparison so that if the instruction is a branch, we can set the PC to the branch target address.
- Note that the bypassed source operands of a branch can come from either the ALU/MEM or MEM/WB pipeline latches.
- For example, if an ALU instruction immediately preceding a branch produces one of the operands for the comparison in the branch, a stall will be required, since the EX stage for the ALU instruction will occur after the ID cycle of the branch.

Example:

```
36 sub $10, $4, $8
40 beq $1, $3, 7 #      PC-relative branch to 40 + 4 + 7 * 4 = 72
44 and $12, $2, $5
48 or $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
. . .
72 lw $4, 50($7)
```

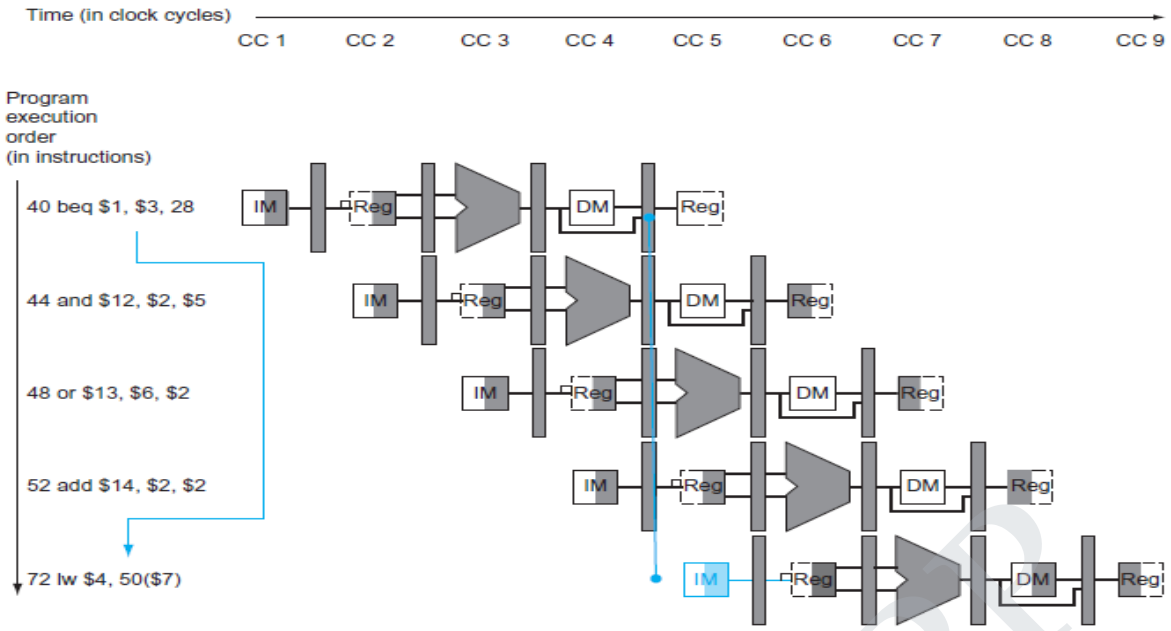
- The three sequential instructions that follow the branch will be fetched and begin execution. Without intervention, those three following instructions will begin execution before beq branches to lw at location 72.

Latency (pipeline)

- The number of stages in a pipeline or the number of stages between two instructions during execution.
- Pipelining does not reduce the time it takes to complete an individual instruction, also called the latency.

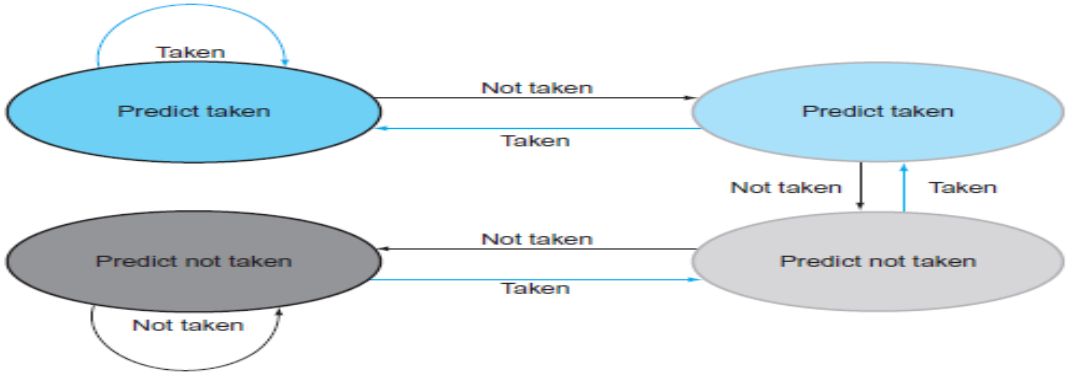
Flush

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



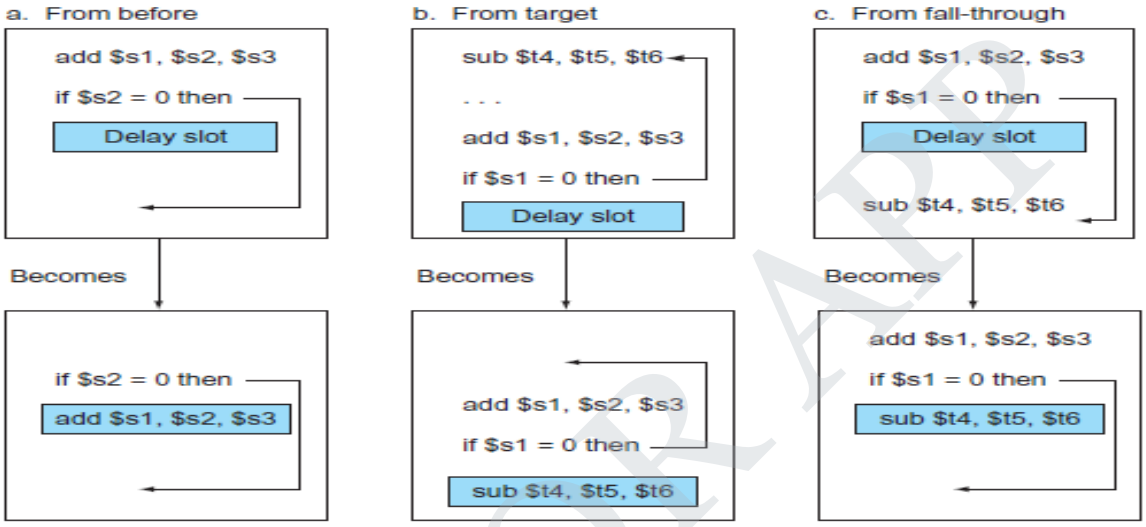
(ii). Dynamic Branch Prediction

- This strategy uses recent branch history during program execution to predict whether or not the branch will be taken next time when it occurs. It uses recent branch information to predict the next branch. This technique is called **dynamic branch prediction.**
- Prediction of branches at runtime using runtime information.
- **A branch prediction buffer** or **branch history table** is a small memory 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.
- This simple 1-bit prediction scheme has a performance shortcoming: even if a branch is almost always taken, we can predict incorrectly twice, rather than once, when it is not taken.
- To remedy this weakness, 2-bit prediction schemes are often used. In a **2-bit scheme**, a prediction must be wrong twice before it is changed.
- The following diagram shows the finite-state machine for a 2-bit prediction scheme. A branch prediction buffer can be implemented as a small, special buffer accessed with the instruction address during the IF pipe stage.
- If the instruction is predicted as taken, fetching begins from the target as soon as the PC is known; it can be as early as the ID stage. Otherwise, sequential fetching and executing continue. If the prediction turns out to be wrong, the prediction bits are changed.



2. Delayed branching

- The slot directly after a delayed branch instruction, which in the MIPS architecture is filled by an instruction that does not affect the branch.
- An instruction that always executes after the branch in the **branch delay slot**.
- The following figure shows the three ways in which the branch delay slot can be scheduled. The limitations on delayed branch scheduling arise from
  1. The restrictions on the instructions that are scheduled into the delay slots.
  2. Our ability to predict at compile time whether a branch is likely to be taken or not.
- Delayed branching was a simple and effective solution for a five-stage pipeline issuing one instruction each clock cycle.



- 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.
- This is the case, for example, if $t4 were an unused temporary register when the branch goes in the unexpected direction.

Branch target buffer:

- A structure that caches the destination PC or destination instruction for a branch. It is usually organized as a cache with tags, making it more costly than a simple prediction buffer.

Correlating predictor:

- A branch predictor that combines local behavior of a particular branch and global information about the behavior of some recent number of executed branches.

Tournament branch predictor

- A branch predictor with multiple predictions for each branch and a selection mechanism that chooses which predictor to enable for a given branch.

EXCEPTIONS

- **Exceptions** and **interrupts** events other than branches or jumps that change the normal flow of instruction execution.

Exception

- Exception also called interrupt. An unscheduled event that disrupts program execution and they are used to detect overflow.
- The two types of exceptions that our current implementation can generate are execution of an undefined instruction and an arithmetic overflow.

Interrupt

- It is an exception that comes from outside of the processor.
- We use the term *interrupt* only when the event is externally caused. Here are five examples showing whether the situation is internally generated by the processor or externally generated:

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

Handling Exception:

- The two types of exceptions can occur in the basic MIPS architecture implementation.
  1. Execution of an undefined instruction
  2. An arithmetic overflow.

Response to an Exception:

- When an exception occurs the processor saves the address of the ending instruction in the exception program counter (EPC) and then transfer control to the operating system at some specified address.
- The operating system then takes the appropriate action, which may involve providing some service to the user program, taking some predefined action in response to an overflow, or stopping the execution of the program and reporting an error.
- After performing whatever action is required because of the exception, the operating system can terminate the program or may continue its execution, using the EPC to determine where to restart the execution of the program.

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

- The first method used in the MIPS architecture is to include a **status register** (called the Cause register), which holds a field that indicates the reason for the exception.

- A second method is to use **vectored interrupts**. In a vectored interrupt, the address to which control is transferred is determined by the cause of the exception.

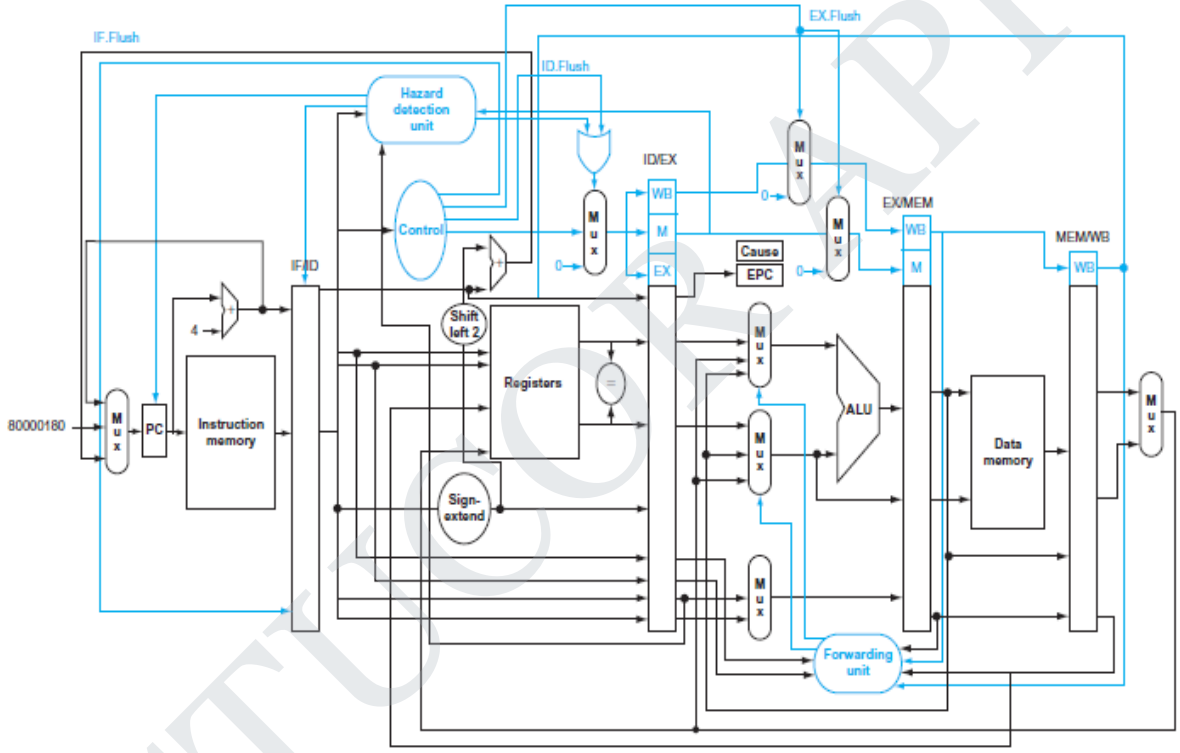
For example, to accommodate the two exception types listed above, we might define the following two exception vector addresses:

Exception type	Exception vector address (in hex)
Undefined instruction	$8000\ 0000_{hex}$
Arithmetic overflow	$8000\ 0180_{hex}$

Add two additional registers to our current MIPS implementation:

- ***EPC:*** A 32-bit register used to hold the address of the affected instruction.
- ***Cause:*** A register used to record the cause of the exception. In the MIPS architecture, this register is 32 bits, although some bits are currently unused.
- Assume there is a five-bit field that encodes the two possible exception sources mentioned above, with 10 representing an undefined instruction and 12 representing arithmetic overflow.

Exceptions in a Pipelined Implementation



Imprecise interrupt

- Imprecise interrupt also called imprecise exception. Interrupts or exceptions in pipelined computers that is not associated with the exact instruction that was the cause of the interrupt or exception.

Precise interrupt

- Precise interrupt also called precise exception. An interrupt or exception that is always associated with the correct instruction in pipelined computers.

UNIT IV

PARALLELISM

Parallel processing challenges – Flynn's classification – SISD, MIMD, SIMD, SPMD, and Vector Architectures - Hardware multithreading – Multi-core processors and other Shared Memory Multiprocessors - Introduction to Graphics Processing Units, Clusters, Warehouse Scale Computers and other Message-Passing Multiprocessors.

PARALLEL PROCESSING CHALLENGES:

- It is difficult to write software that uses multiple processors to complete one task is faster.
- Parallel processing will increase the performance of processor and it will reduce the utilization time to execute a task.
- By obtaining the parallel processing is not an easy task.
- The difficulty is not in hardware side it is in software side. We can understand that it is difficult to write parallel processing programs that are fast, especially as the number of processor increases.

Advantage:

1. To get better performance
2. It produce better energy efficiency

Performance:

If we are running a program on two different desktop computers, we will say that the faster one is the desktop computer that gets the job done first.

In some situation we cannot get parallel processing as faster than sequential programs. The reasons are,

1. **Scheduling:** It is done to load balance and share system resources effectively and achieve a target. It can be done by the following ways,
    1. Long term
    2. Medium Term
    3. Short Term
    4. Dispatcher
2. **Portioning the work into parallel pieces:** Divide the task equally to all the processor.
3. **Load Balancing:** Work load distribute evenly and amount of execution time is also equal.
4. **Time to Synchronize:** Here the throughput must be high.
5. **Overhead for Communication**

Strong scaling: Speedup achieved on a multiprocessor without increasing the size of the problem.

Weak scaling: Speedup achieved on a multiprocessor while increasing the size of the problem proportionally to the increase in the number of processors.

Speed-up Challenge

Suppose you want to achieve a speed-up of 90 times faster with 100 processors. What percentage of the original computation can be sequential? **[Dec'17]**

Amdahl's Law says

$$\text{Execution time after improvement} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$

We can reformulate Amdahl's Law in terms of speed-up versus the original execution time:

$$\text{Speed-up} = \frac{\text{Execution time before}}{(\text{Execution time before} - \text{Execution time affected}) + \frac{\text{Execution time affected}}{\text{Amount of improvement}}}$$

This formula is usually rewritten assuming that the execution time before is 1 for some unit of time, and the execution time affected by improvement is considered the fraction of the original execution time:

$$\text{Speed-up} = \frac{1}{(1 - \text{Fraction time affected}) + \frac{\text{Fraction time affected}}{\text{Amount of improvement}}}$$

Substituting 90 for speed-up and 100 for amount of improvement into the formula above:

$$90 = \frac{1}{(1 - \text{Fraction time affected}) + \frac{\text{Fraction time affected}}{100}}$$

Then simplifying the formula and solving for fraction time affected:

$$90 \times (1 - 0.99 \times \text{Fraction time affected}) = 1$$
$$90 - (90 \times 0.99 \times \text{Fraction time affected}) = 1$$
$$90 - 1 = 90 \times 0.99 \times \text{Fraction time affected}$$
$$\text{Fraction time affected} = 89/89.1 = 0.999$$

Thus, to achieve a speed-up of 90 from 100 processors, the sequential percentage can only be 0.1%.

Example: 2

Speed-up Challenge: Bigger Problem

Suppose you want to perform two sums: one is a sum of 10 scalar variables, and one is a matrix sum of a pair of two-dimensional arrays, with dimensions 10 by 10. For now let's assume only the matrix sum is parallelizable; we'll see soon how to parallelize scalar sums. What speed-up do you get with 10 versus 40 processors? Next, calculate the speed-ups assuming the matrices grow to 20 by 20.

If we assume performance is a function of the time for an addition, t, then there are 10 additions that do not benefit from parallel processors and 100 additions that do. If the time for a single processor is 110 t, the execution time for 10 processors is

$$\text{Execution time after improvement} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$

$$\text{Execution time after improvement} = \frac{100t}{10} + 10t = 20t$$

so the speed-up with 10 processors is $110t/20t = 5.5$. The execution time for 40 processors is

$$\text{Execution time after improvement} = \frac{100t}{40} + 10t = 12.5t$$

so the speed-up with 40 processors is $110t/12.5t = 8.8$. Thus, for this problem size, we get about 55% of the potential speed-up with 10 processors, but only 22% with 40.

Look what happens when we increase the matrix. The sequential program now takes $10t + 400t = 410t$. The execution time for 10 processors is

$$\text{Execution time after improvement} = \frac{400t}{10} + 10t = 50t$$

so the speed-up with 10 processors is $410t/50t = 8.2$. The execution time for 40 processors is

$$\text{Execution time after improvement} = \frac{400t}{40} + 10t = 20t$$

so the speed-up with 40 processors is $410t/20t = 20.5$. Thus, for this larger problem size, we get 82% of the potential speed-up with 10 processors and 51% with 40.

Speed-up Challenge: Balancing Load

Example:3

To achieve the speed-up of 20.5 on the previous larger problem with 40 processors, we assumed the load was perfectly balanced. That is, each of the 40 processors had 2.5% of the work to do. Instead, show the impact on speed-up if one processor's load is higher than all the rest. Calculate at twice the load (5%) and five times the load (12.5%) for that hardest working processor. How well utilized are the rest of the processors?

Solution:

If one processor has 5% of the parallel load, then it must do 5% × 400 or 20 additions, and the other 39 will share the remaining 380. Since they are operating simultaneously, we can just calculate the execution time as a maximum

$$\text{Execution time after improvement} = \text{Max}\left(\frac{380t}{39}, \frac{20t}{1}\right) + 10t = 30t$$

The speed-up drops from 20.5 to $410t/30t = 14$. The remaining 39 processors are utilized less than half the time: while waiting 20t for hardest working processor to finish, they only compute for $380t/39 = 9.7t$.

If one processor has 12.5% of the load, it must perform 50 additions. The formula is:

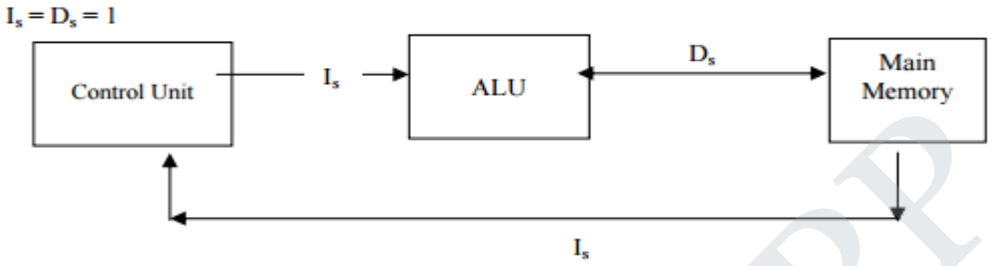
$$\text{Execution time after improvement} = \text{Max}\left(\frac{350t}{39}, \frac{50t}{1}\right) + 10t = 60t$$

The speed-up drops even further to $410t/60t = 7$. The rest of the processors are utilized less than 20% of the time ($9t/50t$). This example demonstrates the importance of balancing load, for just a single processor with twice the load of the others cuts speed-up by a third, and five times the load on just one processor reduces speed-up by almost a factor of three.

PROCESSOR ORGANAIZATION [FLYNN'S CLASSIFICATION]

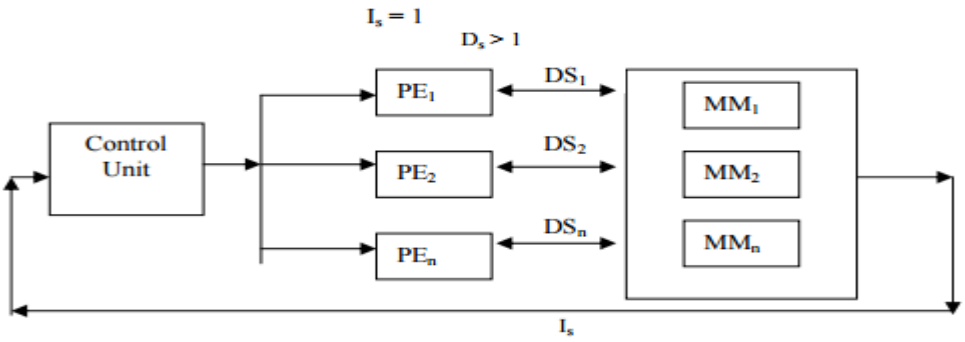
SISD

- Single Instruction stream, Single Data stream.
- Example of SISD is uniprocessor.
- It has a single control unit and producing a single stream of instruction.
- It has one processing unit and the processing has more than one functional unit these are under the supervision of one control unit.
- It has one memory unit.



SIMD

- It has one instruction and multiple data stream.
- It has a single control unit and producing a single stream of instruction and multi stream of data.
- It has more than one processing unit and each processing unit has its own associative data memory unit.
- In this organization, multiple processing elements work under the control of a single control unit.
- A single machine instruction controls the simultaneous execution of a number of processing element.
- Each instruction to be executed on different sets of data by different processor.
- The same instruction is applied to many data streams, as in a vector processor.
- All the processing elements of this organization receive the same instruction broadcast from the CU.
- Main memory can also be divided into modules for generating multiple data streams acting as a distributed memory as shown in figure.
- Therefore, all the processing elements simultaneously execute the same instruction and are said to be 'lock-stepped' together.
- Each processor takes the data from its own memory and hence it has on distinct data streams.
- Every processor must be allowed to complete its instruction before the next instruction is taken for execution. Thus, the execution of instructions is synchronous.
- Example of SIMD is Vector Processor and Array Processor.

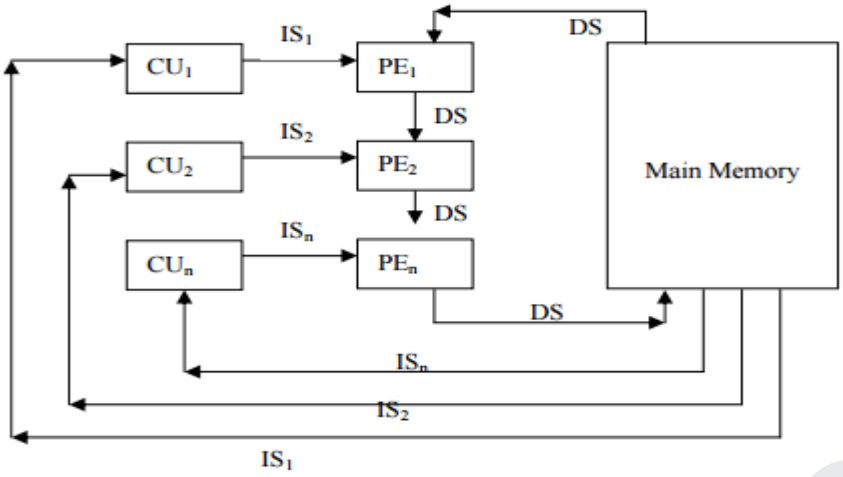


Advantage of SIMD:

- The original motivation behind SIMD was to amortize the cost of the control unit over dozens of execution units.
- Another advantage is the reduced instruction bandwidth and space.
- SIMD needs only one copy of the code that is being simultaneously executed while message-passing MIMDs may need a copy in every processor, and shared memory MIMD will need multiple instruction caches.
- SIMD works best when dealing with arrays in for loops because parallelism achieved by performing the same operation on independent data.
- SIMD is at its weakest in case or switch statements, where each execution unit must perform a different operation on its data, depending on what data it has. Execution units with the wrong data must be disabled so that units with proper data may continue.

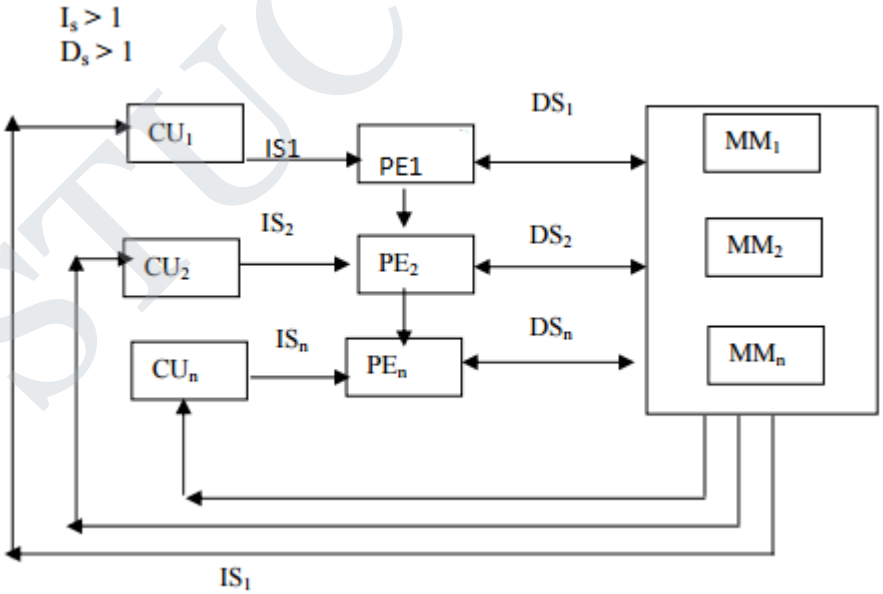
MISD

- Multiple Instruction and Single Data stream (MISD)
- In this organization, multiple processing elements are organized under the control of multiple control units.
- Each control unit is handling one instruction stream and processed through its corresponding processing element.
- But each processing element is processing only a single data stream at a time.
- Therefore, for handling multiple instruction streams and single data stream, multiple control units and multiple processing elements are organized in this classification.
- All processing elements are interacting with the common shared memory for the organization of single data stream as shown in figure.
- The only known example of a computer capable of MISD operation is the C.mmp built by Carnegie-Mellon University.



MIMD

- Multiple Instruction streams and Multiple Data streams (MIMD). In this organization, multiple processing elements and multiple control units are organized.
- Compared to MISD the difference is that now in this organization multiple instruction streams operate on multiple data streams.
- Therefore, for handling multiple instruction streams, multiple control units and multiple processing elements are organized such that multiple processing elements are handling multiple data streams from the main memory as shown in figure.
- The processors work on their own data with their own instructions. Tasks executed by different processors can start or finish at different times.
- They are not lock-stepped, as in SIMD computers, but run asynchronously.
- This classification actually recognizes the parallel computer. That means in the real sense MIMD organization is said to be a Parallel computer.



SIMD-VECTOR ARCHITECTURE [SPMD]

- SIMD is called vector architecture.
- It is also a great match to problems with lots of data-level parallelism.i.e. Parallelism achieved by performing the same operation on independent data.

- Rather than having 64 ALUs perform 64 additions simultaneously, like the old array processors.
- The vector architectures pipelined the ALU to get good performance at lower cost.
- The basic idea of vector architecture is to collect data elements from memory, put them in order into a large set of registers, operate on them sequentially in registers using pipelined execution units, and then write the results back to memory.
- A key feature of vector architectures is then a set of vector registers. Thus, vector architecture might have 32 vector registers, each with 64-bit elements.

Comparing Vector to Conventional Code

- Suppose we extend the MIPS instruction set architecture with vector instructions and vector registers. Vector operations use the same names as MIPS operations, but with the letter V appended.
- For example, addv.d adds two double-precision vectors. The vector instructions take as their input either a pair of vector registers (addv.d) or a vector register and a scalar register (addvs.d).
- The value in the scalar register is used as the input for all operations.
- The operation addvs.d will add the contents of a scalar register to each element in a vector register.
- The names lv and sv denote vector load and vector store, and they load or store an entire vector of double-precision data.
- One operand is the vector register to be loaded or stored; the other operand, which is a MIPS general-purpose register, is the starting address of the vector in memory.
- The conventional MIPS code versus the vector MIPS code for

$$Y = a \times X + Y$$

- Where X and Y are vectors of 64 double precision floating-point numbers, initially resident in memory, and a is a scalar double precision variable.
- This example is the so-called DAXPY loop that forms the inner loop of the DAXPY stands for double precision a × X plus Y.).
- Assume that the starting addresses of X and Y are in $s0 and $s1, respectively.

Here is the conventional MIPS code for DAXPY:

Here is the conventional MIPS code for DAXPY:

```
      l.d     $f0,a($sp)       :load scalar a
      addiu   $t0,$s0,#512     :upper bound of what to load
loop: l.d     $f2,0($s0)       :load x(i)
      mul.d   $f2,$f2,$f0      :a x x(i)
      l.d     $f4,0($s1)       :load y(i)
      add.d   $f4,$f4,$f2      :a x x(i) + y(i)
      s.d     $f4,0($s1)       :store into y(i)
      addiu   $s0,$s0,#8       :increment index to x
      addiu   $s1,$s1,#8       :increment index to y
      subu    $t1,$t0,$s0      :compute bound
      bne     $t1,$zero,loop   :check if done
```

Here is the vector MIPS code for DAXPY:

```
l.d      $f0,a($sp)       :load scalar a
lv       $v1,0($s0)       :load vector x
mulvs.d  $v2,$v1,$f0      :vector-scalar multiply
lv       $v3,0($s1)       :load vector y
addv.d   $v4,$v2,$v3      :add y to product
sv       $v4,0($s1)       :store the result
```

- The most dramatic is that the vector processor greatly reduces the dynamic instruction bandwidth, executing only 6 instructions versus almost 600 for the traditional MIPS architecture.
- This reduction occurs both because the vector operations work on 64 elements at a time and because the overhead instructions that constitute nearly half the loop on MIPS are not present in the vector code.

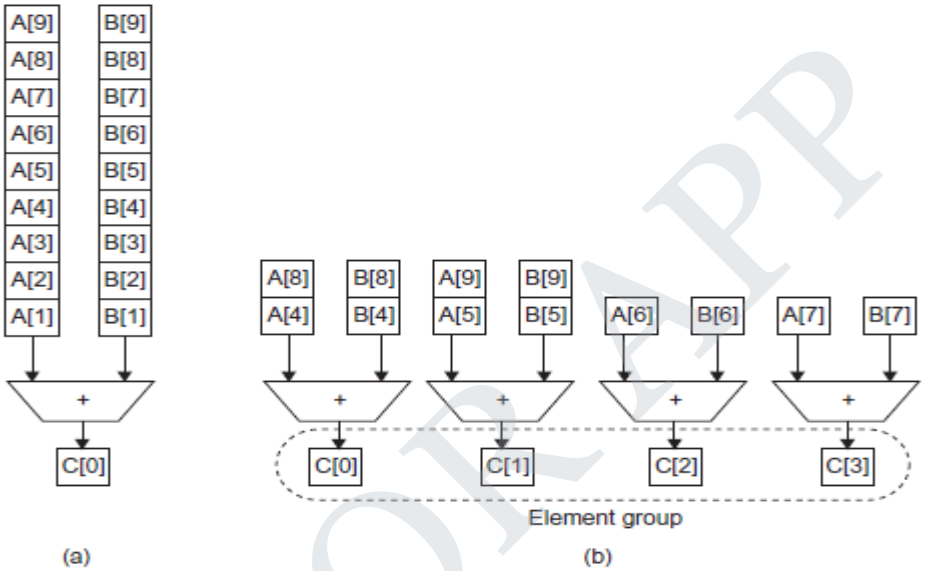
Vector versus Scalar

Vector	Scalar
The instruction fetch and decode bandwidth needed is dramatically reduced.	The instruction fetch and decode bandwidth needed is not reduced.
The compiler or programmer indicates that the computation of each result in the vector is independent of the computation of other results in the same vector,	The compiler or programmer indicates that the computation of each result is not independent of the computation of other results.
Hardware does not have to check for data hazards within a vector instruction.	Hardware to check for data hazards within a vector instruction.
Vector architectures and compilers have a reputation of making it much easier than when using MIMD multiprocessors to write efficient applications when they contain data-level parallelism.	Not easy to perform this operation
Hardware need only check for data hazards between two vector instructions once per vector operand, not once for every element within the vectors.	Hardware need only check for data hazards for every element within the array.
Save energy because of reduced checking	It does not save energy because it has more number of checking.
The cost of the latency to main memory is seen only once for the entire vector, rather than once for each word of the vector.	The cost of the latency to main memory is seen for each word of the scalar.
Efficient use of memory bandwidth and instruction bandwidth.	No efficient use of memory bandwidth and instruction bandwidth.
Entire loop behavior is predetermined	Entire loop behavior is not a predetermined

Vector Versus Multimedia Extensions

- Like multimedia extensions found in the x86 AVX instructions, a vector instruction specifies multiple operations.
- However, multimedia extensions typically specify a few operations while vector specifies dozens of operations.
- The number of elements in a vector operation is not in the opcode but in a separate register.
- This distinction means different versions of the vector architecture can be implemented with a different number of elements just by changing the contents of that register and hence retain binary compatibility.
- In contrast, a new large set of opcodes is added each time the vector length changes in the multimedia extension architecture of the x86: MMX, SSE, SSE2, AVX, AVX2 …
- Also unlike multimedia extensions, the data transfers need not be contiguous.
- Hardware finds the addresses of the items to be loaded in a vector register.

- Indexed accesses are also called gatherscatter, in that indexed loads gather elements from main memory into contiguous vector elements and indexed stores scatter vector elements across main memory.
- The following figure illustrates how to improve vector performance by using parallel pipelines to execute a vector add instruction.
- The figure using multiple functional units to improve the performance of a single vector add instruction, C = A + B.
- The vector processor (a) on the left has a single add pipeline and can complete one addition per cycle.
- The vector processor (b) on the right has four add pipelines or lanes and can complete four additions per cycle.



Vector Lane

- One or more vector functional units and a portion of the vector register file. Inspired by lanes on highways that increase traffic speed, multiple lanes execute vector operations simultaneously.
- Figure shows the structure of a four-lane vector unit. Thus, going to four lanes from one lane reduces the number of clocks per vector instruction by roughly a factor of four.
- The figure shows three vector functional units: an FP add, an FP multiply, and a load-store unit.
- For multiple lanes to be advantageous, both the applications and the architecture must support long vectors.
- The elements within a single vector add instructions are interleaved across the four lanes.
- The vector-register storage is divided across the lanes, with each lane holding every fourth element of each vector register.
- Each of the vector arithmetic units contains four execution pipelines, one per lane, which acts in concert to complete a single vector instruction.

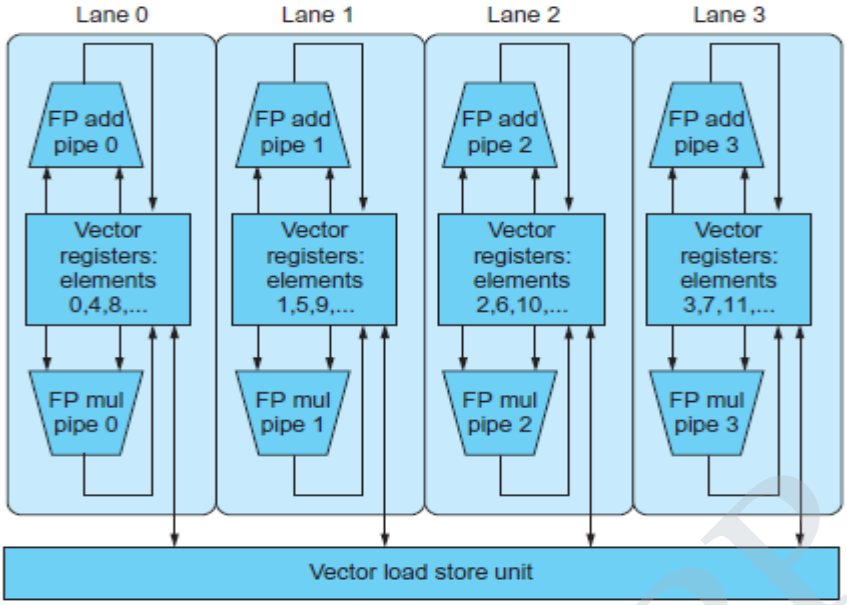


Figure: Structure of a vector unit containing four lanes.

HARDWARE MULTITHREADING

Multithreading

- A mechanism by which the instruction streams is divided into several smaller streams (threads) and can be executed in parallel is called multithreading.

Hardware Multithreading

- Increasing utilization of a processor by switching to another thread when one thread is stalled is known as hardware multithreading.

Thread

- A thread includes the program counter, the register state, and the stack. It is a lightweight process; whereas threads commonly share a single address space, processes don't.

Thread Switch

- The act of switching processor control from one thread to another within the same process. It is much less costly than a processor switch.

Process

- A process includes one or more threads, the address space, and the operating system state. Hence, a process switch usually invokes the operating system, but not a thread switch.

What are the approaches to hardware multithreading?

There are two main approaches to hardware multithreading.

1. Fine-grained Multithreading
2. Coarse-grained Multithreading

Fine-grained Multithreading

- A version of hardware multithreading that implies switching between threads after every instruction resulting in interleaved execution of multiple threads. It switches from one thread to another at each clock cycle.
- This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that clock cycle.

- To make fine-grained multithreading practical, the processor must be able to switch threads on every clock cycle.

Advantage

- Vertical waste is eliminated.
- Pipeline hazards cannot arise.
- Zero switching overhead
- Ability to hide latency within a thread i.e., it can hide the throughput losses that arise from both short and long stalls.
- Instructions from other threads can be executed when one thread stalls.
- High execution efficiency
- Potentially less complex than alternative high performance processors.

Disadvantage

- Clock cycles are wasted if a thread has little operation to execute.
- Needs a lot of threads to execute.
- It is expensive than coarse-grained multithreading.
- It slows down the execution of the individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from other threads.

Coarse-grained Multithreading

- Coarse-grained multithreading was invented as an alternative to fine-grained multithreading.
- A version of hardware multithreading that implies switching between threads only after significant events, such as a last-level cache miss.
- This change relieves the need to have thread switching be extremely fast and is much less likely to slow down the execution of an individual thread, since instructions from other threads will only be issued when a thread encounters a costly stall.

Advantage

- To have very fast thread switching.
- Doesn't slow down thread.

Disadvantage

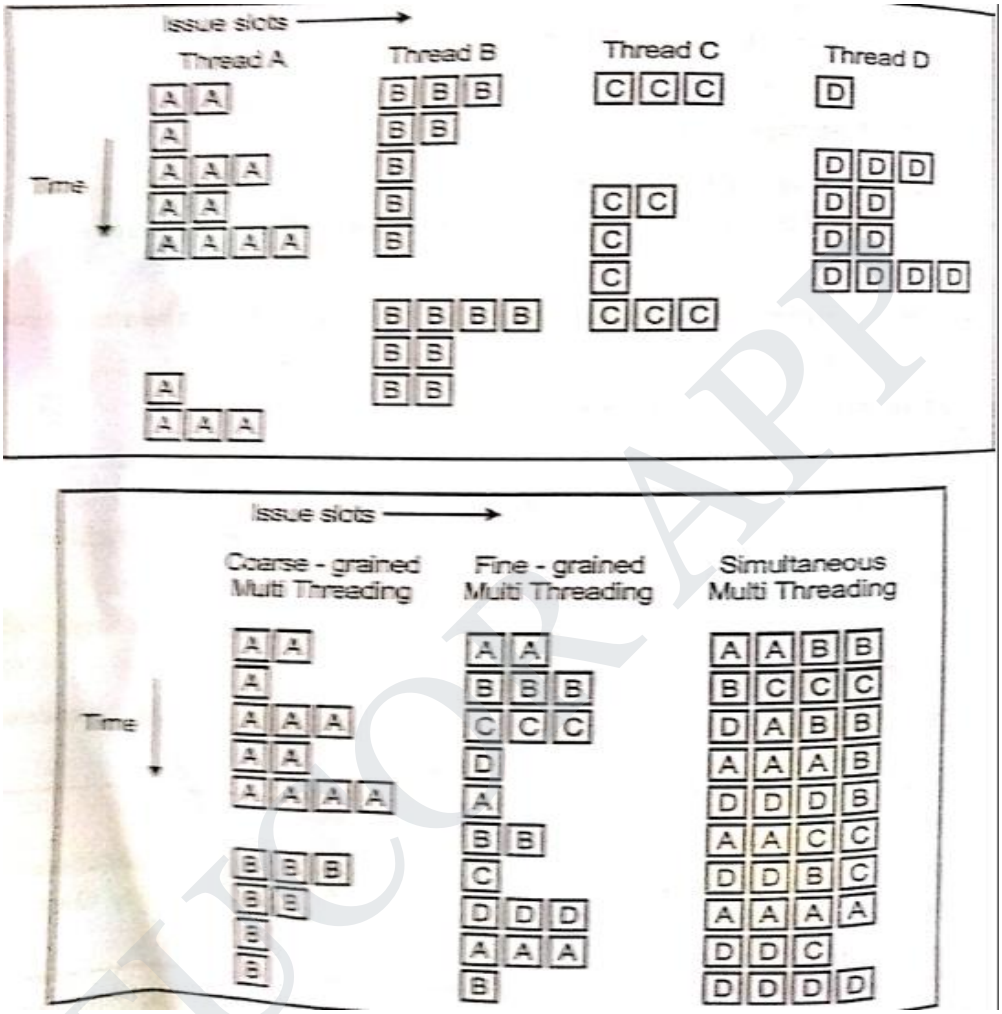
- It is hard to overcome throughput losses from shorter stalls, due to pipeline start-up costs.
- Since CPU issues instructions from 1 thread, when a stall occurs, the pipeline must be emptied.
- New thread must fill pipeline before instructions can complete.
- Due to this start-up overhead, coarse-grained multithreading is much more useful for reducing the penalty of high-cost stalls, where pipeline refill is negligible compared to the stall time.

Simultaneous multithreading (SMT)

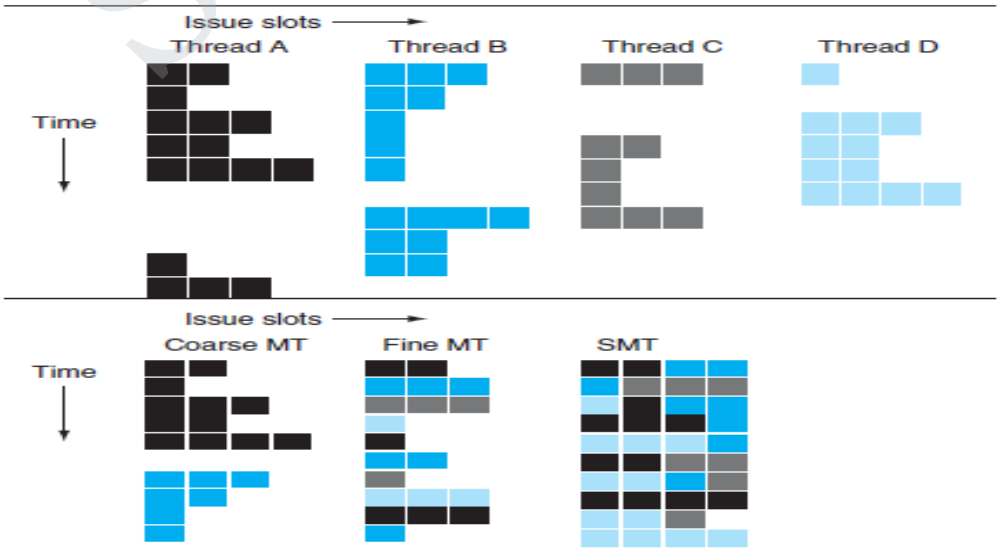
- It is a variation on hardware multithreading that uses the resources of a multiple-issue, dynamically scheduled pipelined processor to exploit thread-level parallelism at the same time it exploits instruction level parallelism.
- The key insight that motivates SMT is that multiple-issue processors often have more functional unit parallelism available than most single threads can effectively use.

- Since SMT relies on the existing dynamic mechanisms, it does not switch resources every cycle.
- Instead, SMT is always executing instructions from multiple threads, to associate instruction slots and renamed registers with their proper threads.

Figure: How four threads use the issue slots of a superscalar processor in different approaches?



- The four threads at the top show how each would execute running alone on a standard superscalar processor without multithreading support.
- The three examples at the bottom show how they would execute running together in



three multithreading options.

- 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 gray and color correspond to four different threads in the multithreading processors.
- The additional pipeline start-up effects for coarse multithreading, which are not illustrated in this figure, would lead to further loss in throughput for coarse multithreading.

Advantage

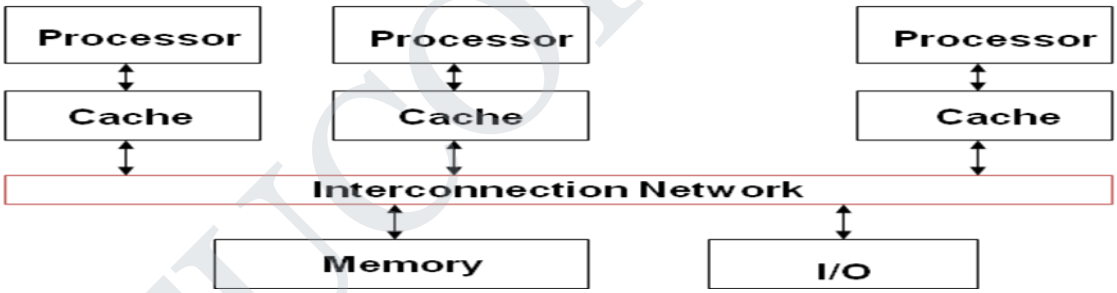
- It is ability to boost utilization by dynamically scheduling functional units among multiple threads.
- It increases hardware design facility.
- It produces better performance and add resources to a fine grained manner.

Disadvantage

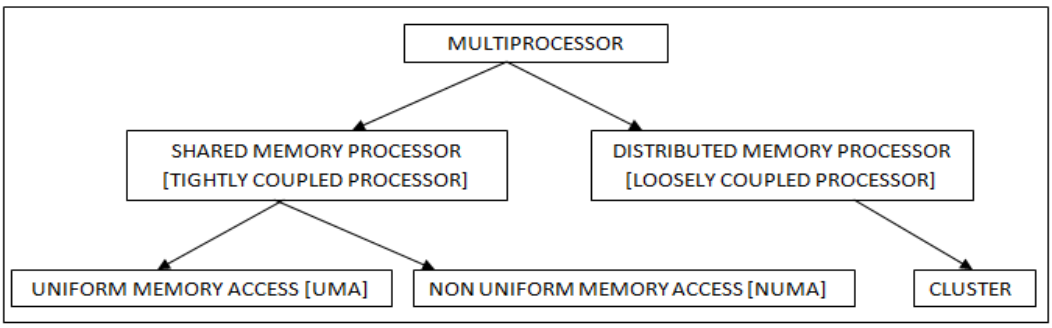
It cannot improve performance if any of the shared resources are the limiting bottlenecks for the performance.

MULTICORE AND OTHER SHARED MEMORY MULTIPROCESSORS

- **Multiprocessor:** A computer system with at least two processors
- **Multicore:** More than one processor available within a single chip.



- The conventional multiprocessor system used is commonly referred as shared memory multiprocessor system.
- **Shared Memory Multiprocessor (SMP)** is one that offers the programmer a single physical address space across all processors which is nearly always the case for multicore chips.
- Processors communicate through shared variables in memory, with all processors capable of accessing any memory location via loads and stores.
- Systems can still run independent jobs in their own virtual address spaces, even if they all share a physical address space.
- Use of shared data must be coordinated via synchronization primitives (locks) that allow access to data to only one processor at a time



Shared Memory Multiprocessor System.[Tightly coupled processor]

- The conventional multiprocessor system used is commonly referred as shared memory multiprocessor system.
- Single address space shared by all processors. Because every processor communicates through a shared global memory.
- For high speed real time processing, these systems are preferable as their throughput is high as compared to loosely coupled systems
- In tightly coupled system organization, multiple processors share a global main memory, which may have many modules.
- Tightly coupled systems use a common bus, crossbar, or multistage network to connect processors, peripherals, and memories.

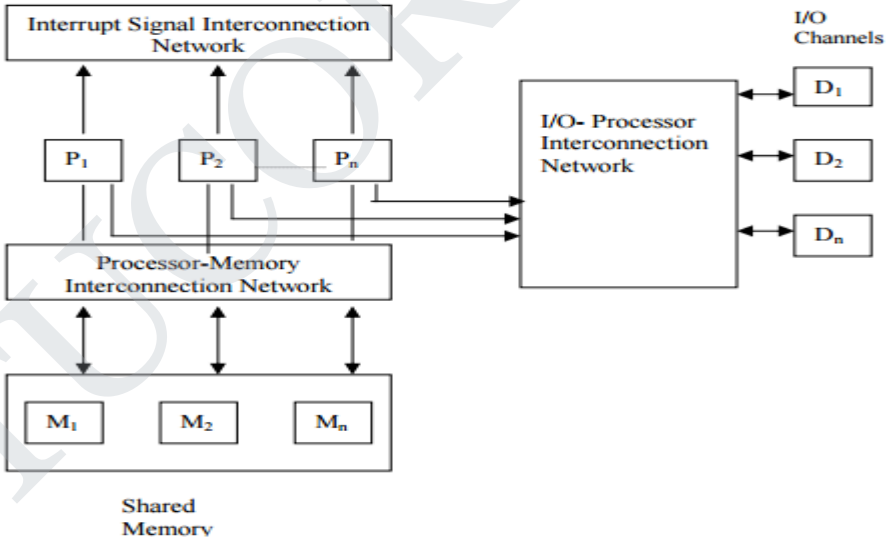
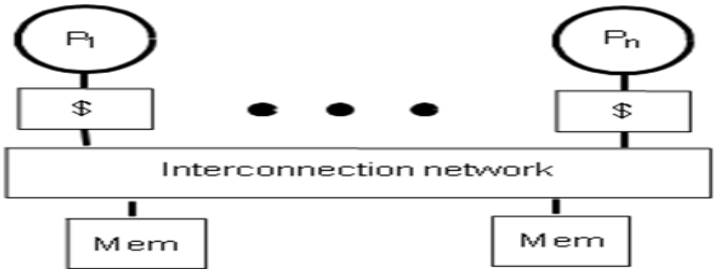


Figure : Tightly coupled system organization

Two common styles of implementing Shared Memory Multiprocessors (SMP) are,

Uniform memory access (UMA) multiprocessors

- In this model, main memory is uniformly shared by all processors in multiprocessor systems and each processor has equal access time to shared memory.
- This model is used for time-sharing applications in a multi user environment
- Tightly-coupled systems (high degree of resource sharing) suitable for general purpose and time-sharing applications by multiple users



- Physical memory uniformly shared by all processors, with equal access time to all words.
- Processors may have local cache memories. Peripherals also shared in some fashion.
- UMA architecture models are of two 20types,

Symmetric:

- All processors have equal access to all peripheral devices. All processors are identical.

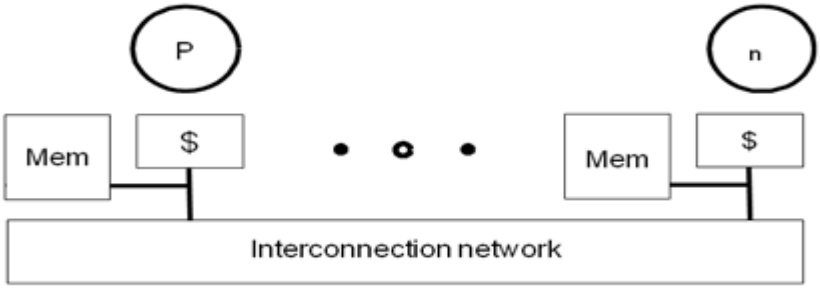
Asymmetric:

- One processor (master) executes the operating system other processors may be of different types and may be dedicated to special tasks.

Non Uniform Memory Access (NUMA) multiprocessors

- In shared memory multiprocessor systems, local memories can be connected with every processor. The collections of all local memories form the global memory being shared.
- In this way, global memory is distributed to all the processors. In this case, the access to a local memory is uniform for its corresponding processor as it is attached to the local memory.
- But if one reference is to the local memory of some other remote processor, then the access is not uniform.
- It depends on the location of the memory. Thus, all memory words are not accessed uniformly. All local memories form a global address space accessible by all processors
- Programming NUMAs are harder but NUMAs can scale to larger sizes and have lower latency to local memory
- Memory is common to all the processors. Processors easily communicate by means of shared variables.
- These systems differ in how the memory and peripheral resources are shared or distributed
- The access time varies with the location of the memory word.
- The shared memory is distributed among the processors as local memories, but each of these is still accessible by all processors (with varying access times).
- Memory access is fastest from the locally –connected processor, with the interconnection network adding delays for other processor accesses.
- Additionally, there may be global memory in a multiprocessor system, with two separate interconnection networks, one for clusters of processors and their cluster memories, and another for the global shared memories.
- Local memories are private with its own program and data. No memory contention so that the number of processors is very large

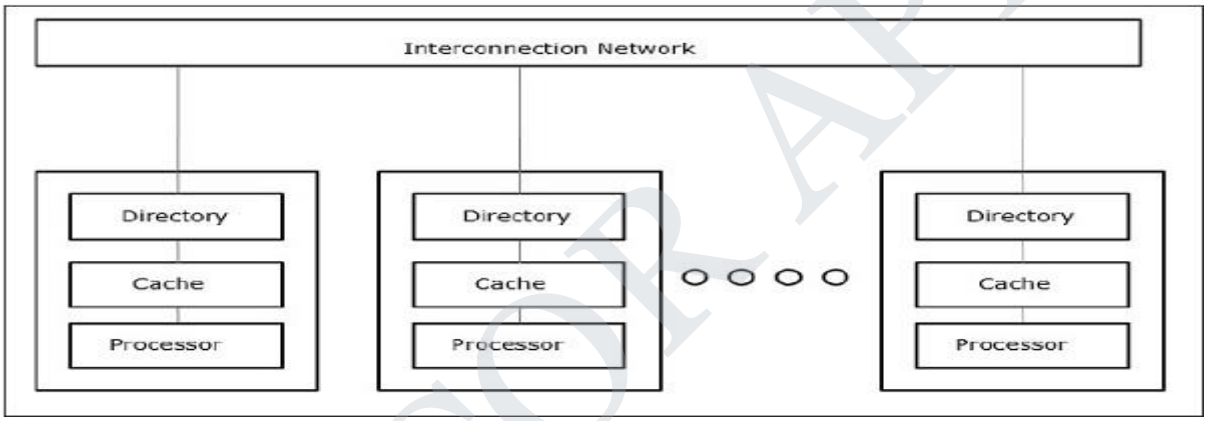
- The processors are connected by communication lines, and the precise way in which the lines are connected is called the topology of the multicomputer.



Distributed Memory (NUMA)

COMA:

- Cache Only Memory Architecture. The COMA model is a special case of the NUMA model. Here all the distributed memories are converted to cache memories.
- The local memories for the processor at each node are used as cache instead of actual



memory.

Distributed Memory [Loosely Coupled Systems]

- These systems do not share the global memory because shared memory concept gives rise to the problem of memory conflicts, which in turn slows down the execution of instructions.
- Therefore, to alleviate this problem, each processor in loosely coupled systems is having a large local memory (LM), which is not shared by any other processor.
- Thus, such systems have multiple processors with their own local memory and a set of I/O devices.
- This set of processor, memory and I/O devices makes a computer system.

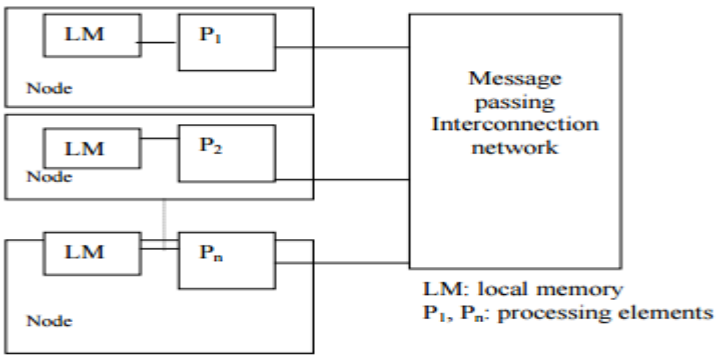


Figure 14: Loosely coupled system organisation

- Therefore, these systems are also called multi-computer systems. These computer systems are connected together via message passing interconnection network through which processes communicate by passing messages to one another.
- Since every computer system or node in multicomputer systems has a separate memory, they are called distributed multicomputer systems. These are also called loosely coupled systems.
- **Message passing:** Communicating between multiple processors by explicitly sending and receiving information.
- **Clusters:** Collections of computers connected via I/O over standard network switches to form a message-passing multiprocessor.
- **Send message routine:** A routine used by a processor in machines with private memories to pass a message to another processor.
- **Receive message routine:** A routine used by a processor in machines with private memories to accept a message from another processor.

MULTI-CORE COMPUTING

All computers are now parallel computers. Multi-core processors represent an important new trend in computer architecture. Decreased power consumption and heat generation. Minimized wire lengths and interconnect latencies. They enable true thread-level parallelism with great energy efficiency and scalability. To utilize their full potential, applications will need to move from a single to a multi-threaded model. Parallel programming techniques likely to gain importance. The difficult problem is not building multi-core hardware, but programming it in a way that lets mainstream applications benefit from the continued exponential growth in CPU performance. The software industry needs to get back into the state where existing applications run faster on new hardware.

Challenges resulting from Multicore:

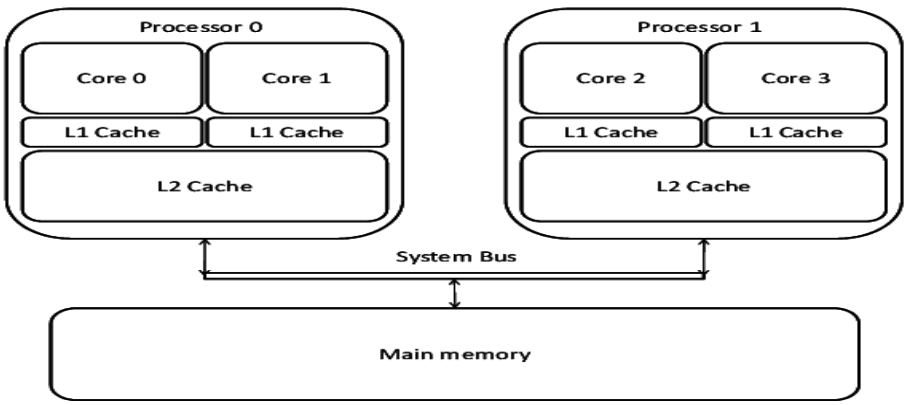
- Relies on effective exploitation of multiple-thread parallelism
  - Need for parallel computing model and parallel programming model
- Aggravates memory wall
- Memory bandwidth
  - Way to get data out of memory banks
  - Way to get data into multi-core processor array
- Memory latency
- Fragments L3 cache
- Pins become strangle point
  - Rate of pin growth projected to slow and flatten
  - Rate of bandwidth per pin (pair) projected to grow slowly
- Requires mechanisms for efficient inter-processor coordination
  - Synchronization
  - Mutual exclusion
  - Context switching

Advantages:

1. Cache coherency circuitry can operate at a much higher clock rate than is possible if the signals have to travel off-chip.
2. Signals between different CPUs travel shorter distances, those signals degrade less.
3. These higher quality signals allow more data to be sent in a given time period since individual signals can be shorter and do not need to be repeated as often.
4. A dual-core processor uses slightly less power than two coupled single-core processors.

Disadvantages

1. Ability of multi-core processors to increase application performance depends on the use of multiple threads within applications.
2. Most Current video games will run faster on a 3 GHz single-core processor than on a 2GHz dual-core processor (of the same core architecture.
3. Two processing cores sharing the same system bus and memory bandwidth limits the real-world performance advantage.
4. If a single core is close to being memory bandwidth limited, going to dual-core might only give 30% to 70% improvement.
5. If memory bandwidth is not a problem, a 90% improvement can be expected.



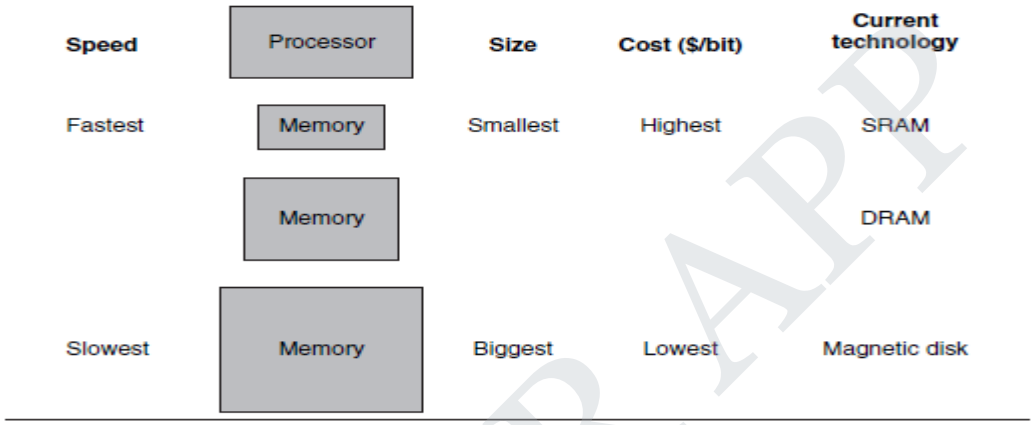
UNIT V

UNIT V MEMORY & I/O SYSTEMS **9**

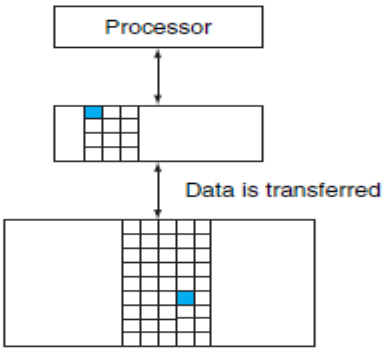
Memory Hierarchy - memory technologies – cache memory – measuring and improving cache performance – virtual memory, TLB's – Accessing I/O Devices – Interrupts – Direct Memory Access – Bus structure – Bus operation – Arbitration – Interface circuits - USB.

MEMORY HIERARCHY:

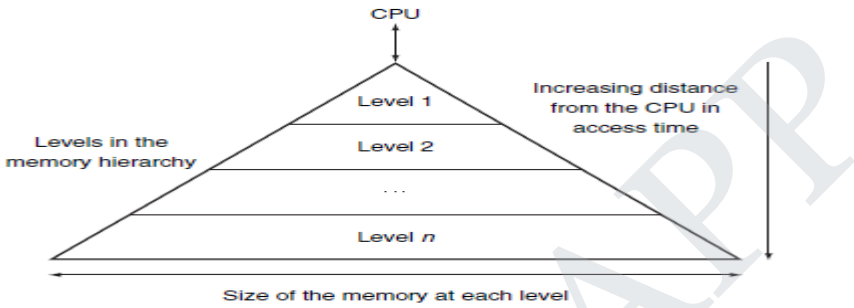
- A memory hierarchy consists of multiple levels of memory with different speeds and sizes.
- The faster memories are smaller and more expensive per bit than the slower memories.



- Figure shows the faster memory is close to the processor and the slower, less expensive memory is below it.
- A memory hierarchy can consist of multiple levels, but data is copied between only two adjacent levels at a time.
- The upper level is the one closer to the processor and smaller and faster than the lower level, since the upper level uses technology that is more expensive.
- **Block:** The minimum unit of information that can be either present or not present in the two-level hierarchy is called a **block or a line.**
- **Hit:** If the data requested by the processor appears in some block in the upper level, this is called a **hit**.
- **Miss:** If the data is not found in the upper level, the request is called a **miss.**
- The lower level in the hierarchy is then accessed to retrieve the block containing the requested data.
- **Hit rate or Hit ratio:** It is the fraction of memory accesses found in the upper level; it is often used as a measure of the performance of the memory hierarchy.
- **Miss rate**: Miss rate (1−hit rate) is the fraction of memory accesses not found in the upper level.
- **Hit time:** It is the time to access the upper level of the memory hierarchy, which includes the time needed to determine whether the access is a hit or a miss.
- **Miss penalty:** It is the time to replace a block in the upper level with the corresponding block from the lower level, plus the time to deliver this block to the processor.



Every pair of levels in the memory hierarchy can be thought of as having an upper and lower level.



- **Principle of locality:** It states that programs access a relatively small portion of their address space at any instant of time. There are two different types of locality:
- **Temporal locality:** The principle stating that if a data location is referenced then it will be likely to be referenced again soon.
- **Spatial locality:** The locality principle stating that if a data location is referenced, data locations with nearby addresses will be likely to be referenced soon.

MEMORY TECHNOLOGY:

There are four primary technologies used today in memory hierarchies.

1. DRAM (Dynamic Random Access Memory)
2. SRAM (Static Random Access Memory).
3. Flash memory.
4. Magnetic disk.

- Main memory is implemented from DRAM (Dynamic Random Access Memory).
- Levels closer to the processor (caches) use SRAM (Static Random Access Memory).
- DRAM is less costly per bit than SRAM, although it is significantly 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 third technology is flash memory. This nonvolatile memory is the secondary memory in Personal Mobile Devices.
- The fourth technology, used to implement the largest and slowest level in the hierarchy in servers, is magnetic disk.
- The access time and price per bit vary widely among these technologies, as the table shows, below.

Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	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

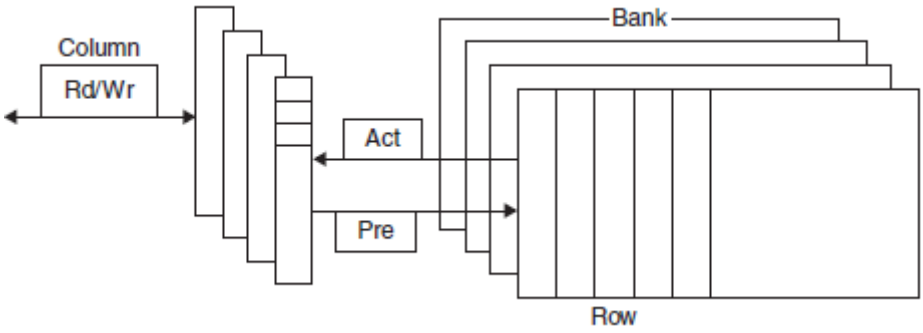
SRAM Technology

- 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.
- SRAMs don't need to refresh and so the access time is very close to the cycle time.
- SRAMs typically use six to eight transistors per bit to prevent the information from being disturbed when read.
- SRAM needs only minimal power to retain the charge in standby mode.
- All levels of caches are integrated onto the processor chip.
- In a SRAM, as long as power is applied, the value can be kept indefinitely.

DRAM Technology

- In a dynamic RAM (DRAM), the value kept in a cell is stored as a charge in a capacitor.
- A single transistor is then used to access this stored charge, either to read the value or to overwrite the charge stored there. Because DRAMs use only a single transistor per bit of storage, they are much denser and cheaper per bit than SRAM.
- As DRAMs store the charge on a capacitor, it cannot be kept indefinitely and must periodically be refreshed. That is why this memory structure is called dynamic, as opposed to the static storage in an SRAM cell.
- To refresh the cell, we merely read its contents and write it back. The charge can be kept for several milliseconds.
- Figure shows the internal organization of a DRAM, and shows the density, cost, and access time of DRAMs.
- Modern DRAMs are organized in banks, typically four for DDR3. Each bank consists of a series of rows.
- When the row is in the buffer, it can be transferred by successive column addresses at whatever the width of the DRAM is (typically 4, 8, or 16 bits in DDR3) or by specifying a block transfer and the starting address.
- To further improve the interface to processors, DRAMs added clocks and are properly called **Synchronous DRAMs or SDRAMs.**
- The advantage of **SDRAM**s is that the use of a clock eliminates the time for the memory and processor to synchronize.
- The speed advantage of synchronous DRAMs comes from the ability to transfer the bits in the burst without having to specify additional address bits.

- The fastest version is called **Double Data Rate (DDR) SDRAM.** The name means data transfers on both the rising and falling edge of the clock, thereby getting twice as much bandwidth as you might expect based on the clock rate and the data width.



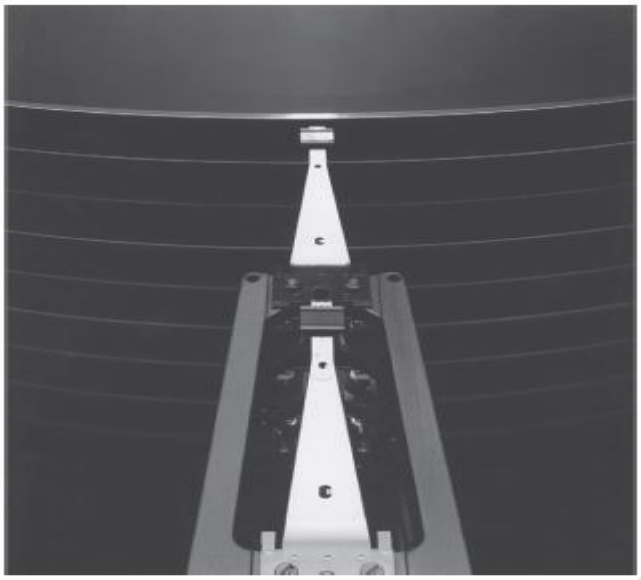
- Sending an address to several banks permits them all to read or write simultaneously. For example, with four banks, there is just one access time and then accesses rotate between the four banks to supply four times the bandwidth. This rotating access scheme is called **address interleaving.**

Flash Memory

- Flash memory is a type of electrically erasable programmable read-only memory (EEPROM).
- EEPROM technologies use flash memory bits for writing purpose.
- 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.**
- Personal mobile devices are very unlikely to exceed the write limits in the flash.

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 disk heads to be much closer to the drive surface.
- **Tracks:** Each disk surface is divided into concentric circles, called **tracks.** There are typically tens of thousands of tracks per surface.
- **Sector:** Each track is in turn divided into **sectors** that contain the information; each track may have thousands of sectors. Sectors are typically 512 to 4096 bytes in size.
- The sequence recorded on the magnetic media is a sector number, a gap, the information for that sector including error correction code.
- The disk heads for each surface are connected together and move in conjunction, so that every head is over the same track of every surface.
- The term cylinder is used to refer to all the tracks under the heads at a given point on all surfaces.



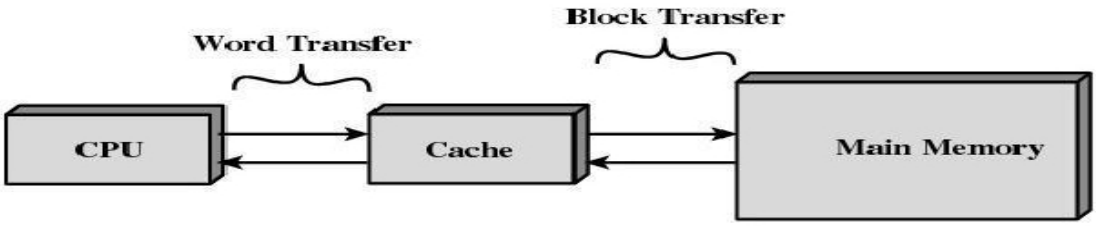
- **Seek Time:** To access data, the operating system must direct the disk through a three-stage process. The first step is to position the head over the proper track. This operation is called a seek, and the time to move the head to the desired track is called the **seek time.**
- Average seek times are usually advertised as 3 ms to 13 ms, but, depending on the application and scheduling of disk requests.
- **Rotational Latency:** 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 Average rotational latency 0.5 rotation.

$$\text{Average rotational latency} = \frac{0.5 \text{ rotation}}{5400 \text{ RPM}} = \frac{0.5 \text{ rotation}}{5400 \text{ RPM}/\left(60\frac{\text{seconds}}{\text{minute}}\right)}$$
$$= 0.0056 \text{ seconds} = 5.6 \text{ ms}$$

- The last component of a disk access, transfer time, is the time to transfer a block of bits.
- The transfer time is a function of the sector size, the rotation speed, and the recording density of a track.
- In summary, the two primary differences between magnetic disks and semiconductor memory technologies are that disks have a slower access time because they are mechanical devices flash is 1000 times as fast and DRAM is 100,000 times as fast.
- Magnetic disks memory is cheaper per bit because they have very high storage capacity at a modest cost disk is 10 to 100 times cheaper.
- Magnetic disks are nonvolatile like flash, but unlike flash there is no write wear-out problem. However, flash is much more rugged and hence a better match to the jostling inherent in personal mobile devices.

CACHE MEMORY

- Cache to represent the level of the memory hierarchy between the processor and main memory.



- If the number of entries in the cache is a power of 2, then modulo can be computed simply by using the low-order $\log_2$ (cache size in blocks) bits of the address.
- Because each cache location can contain the contents of a number of different memory locations.
- The tags contain the address information required to identify whether a word in the cache corresponds to the requested word.
- The tag needs only to contain the upper portion of the address, corresponding to the bits that are not used as an index into the cache. The lower 3-bit index field of the address selects the block.
- **Valid Bit:** The most common method is to add a valid bit a field in the tables of a memory hierarchy that indicates that the associated block in the hierarchy contains **valid data.** i.e., to indicate whether an entry contains a valid address. If the bit is not set, there cannot be a match for this block.

Accessing a Cache

- Below is a sequence of nine memory references to an empty eight-block cache, including the action for each reference.
- Figure shows how the contents of the cache change on each miss.

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	miss	$(11010_{two} \bmod 8) = 010_{two}$
22	10110_{two}	hit	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	hit	$(11010_{two} \bmod 8) = 010_{two}$
16	10000_{two}	miss	$(10000_{two} \bmod 8) = 000_{two}$
3	00011_{two}	miss	$(00011_{two} \bmod 8) = 011_{two}$
16	10000_{two}	hit	$(10000_{two} \bmod 8) = 000_{two}$
18	10010_{two}	miss	$(10010_{two} \bmod 8) = 010_{two}$
16	10000_{two}	hit	$(10000_{two} \bmod 8) = 000_{two}$

- Since there are eight blocks in the cache, the low-order three bits of an address give the block number:
- 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), and 10000two (miss), 00011two (miss), 10000two (hit), 10010two (miss), and 10000two (hit).
- The figures show the cache contents after each miss in the sequence has been handled. The tag field will contain only the upper portion of the address.

- A tag field, which is used to compare with the value of the tag field of the cache and a cache index, which is used to select the block.

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a. The initial state of the cache after power-on

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

b. After handling a miss of address (10110_{two})

Index	V	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

c. After handling a miss of address (11010_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

d. After handling a miss of address (10000_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

e. After handling a miss of address (00011_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	10_{two}	Memory (10010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

f. After handling a miss of address (10010_{two})

- The total number of bits needed for a cache is a function of the cache size and the address size, because the cache includes both the storage for the data and the tags.
- The size of the block above was one word, but normally it is several. For the following situation:

1. 32-bit addresses
2. A direct-mapped cache
3. The cache size is 2^n blocks, so n bits are used for the index
4. 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 × (block size + tag size+ 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 2^{10} (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.

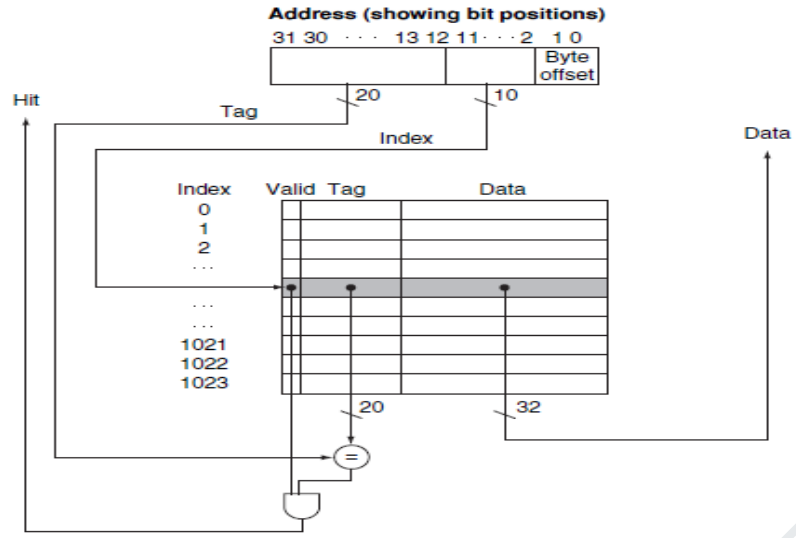


FIGURE For this cache, the lower portion of the address is used to select a cache entry consisting of a data word and a tag.

Example: 1

Mapping an Address to a Multiword Cache Block

Consider a cache with 64 blocks and a block size of 16 bytes. To what block number does byte address 1200 map?

We saw the formula on page 384. The block is given by

(Block address) modulo (Number of blocks in the cache)

where the address of the block is

$$\frac{\text{Byte address}}{\text{Bytes per block}}$$

Notice that this block address is the block containing all addresses between

$$\left\lfloor \frac{\text{Byte address}}{\text{Bytes per block}} \right\rfloor \times \text{Bytes per block}$$

and

$$\left\lfloor \frac{\text{Byte address}}{\text{Bytes per block}} \right\rfloor \times \text{Bytes per block} + (\text{Bytes per block} - 1)$$

Thus, with 16 bytes per block, byte address 1200 is block address

$$\left\lfloor \frac{1200}{6} \right\rfloor = 75$$

which maps to cache block number (75 modulo 64) = 11. In fact, this block maps all addresses between 1200 and 1215.

Example: 2

How many total bits are required for a direct-mapped cache with 16 KiB of data and 4-word blocks, assuming a 32-bit address?

Solution:

We know that 16 KiB is 4096 (2^{12}) words. With a block size of 4 words (2^2), there are 1024 (2^{10}) blocks. Each block has 4 * 32 or 128 bits of data plus a tag, which is (32-10-2-2) bits, plus a valid bit. Thus, the total cache size is

$$2^{10} \times (4 \times 32 + (32 - 10 - 2 - 2) + 1) = 2^{10} \times 147 = 147 \text{ Kibibits}$$

or 18.4 KiB for a 16 KiB cache. For this cache, the total number of bits in the cache is about 1.15 times as many as needed just for the storage of the data.

CACHE MEMORY MAPPING TECHNIQUES:

1. Direct Mapping

- **Direct-Mapped Cache:** A cache structure in which each memory location is mapped to exactly one location in the cache.

(Block address) modulo (Number of blocks in the cache)

The contention may occur,

- When the cache is full
- When more than one memory block is mapped onto a given cache block position.

The contention is resolved by allowing the new blocks to overwrite the currently resident block. Placement of block in the cache is determined from memory address. The memory address is divided into 3 fields. They are,

- **Low Order 4 bit field(word)->**Selects one of 16 words in a block.
- **7 bit cache block field->**When new block enters cache,7 bit determines the cache position in which this block must be stored.
- **5 bit Tag field->**The high order 5 bits of the memory address of the block is stored in 5 tag

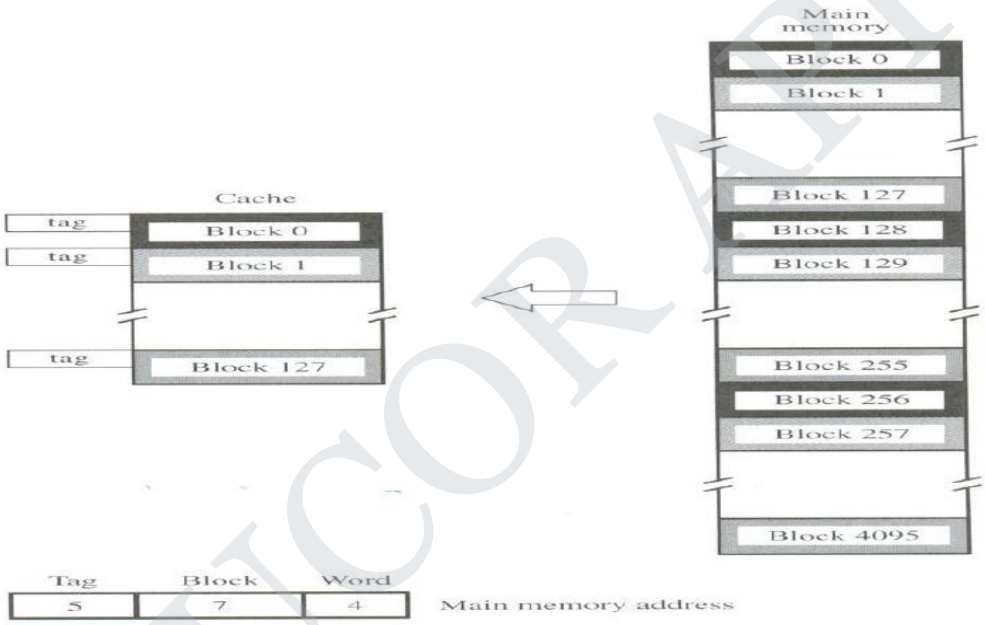


Fig: Direct Mapped Cache

Merit:
It is easy to implement
Demerit:
It is not very flexible.

2. Fully Associative:

- Where a block can be placed in any location in the cache. Such a scheme is called **fully associative.** It is also called associative Mapping method.
- To find a given block in a fully associative cache, all the entries in the cache must be searched because a block can be placed in any one.

Advantage: More Flexible than Direct Mapping

Disadvantage:

- Increase the hardware cost.
- It is suitable for caches with small numbers of blocks.

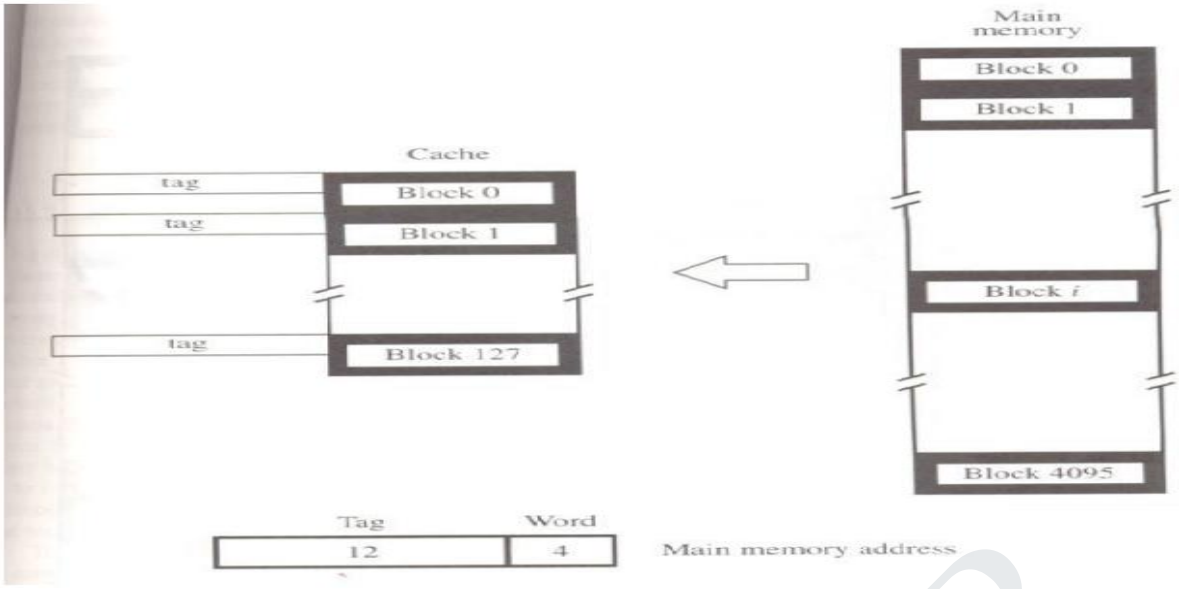


Fig: Associative Mapped Cache

3. Set Associative:

- The middle range of designs between direct mapped and fully associative is called **set associative.**
- In a set-associative cache, there are a fixed number of locations where each block can be placed.
- A set-associative cache with n locations for a block is called an n-way set-associative cache.

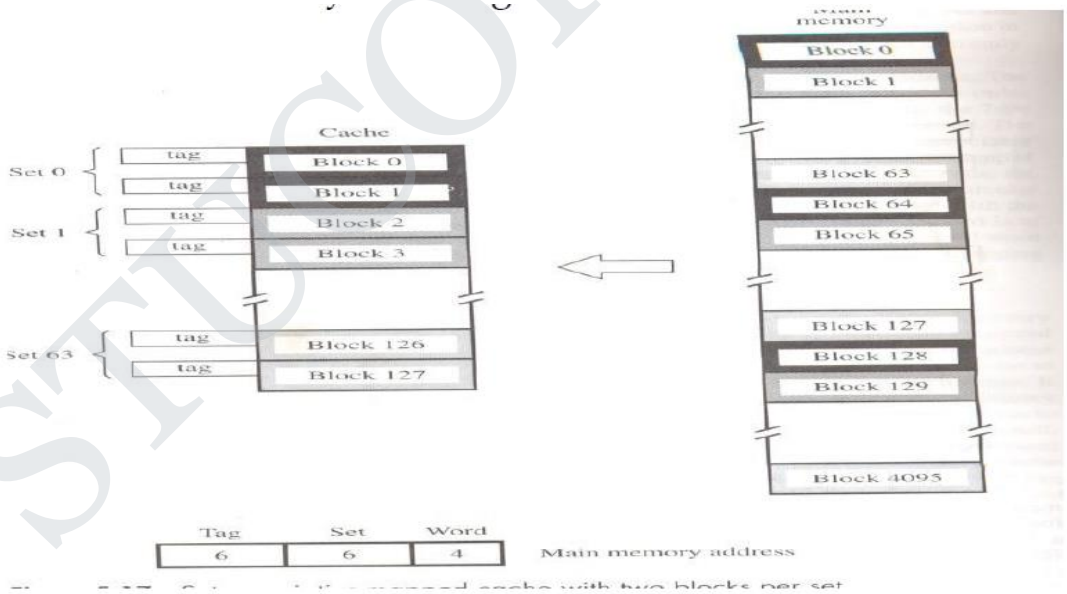


Fig: Set-Associative Mapping

- An n-way set-associative cache consists of a number of sets, each of which consists of n blocks.
- Each block in the memory maps to a unique set in the cache given by the index field, and a block can be placed in any element of that set.
- In a set-associative cache, the set containing a memory block is given by

(Block number) modulo (Number of *sets* in the cache)

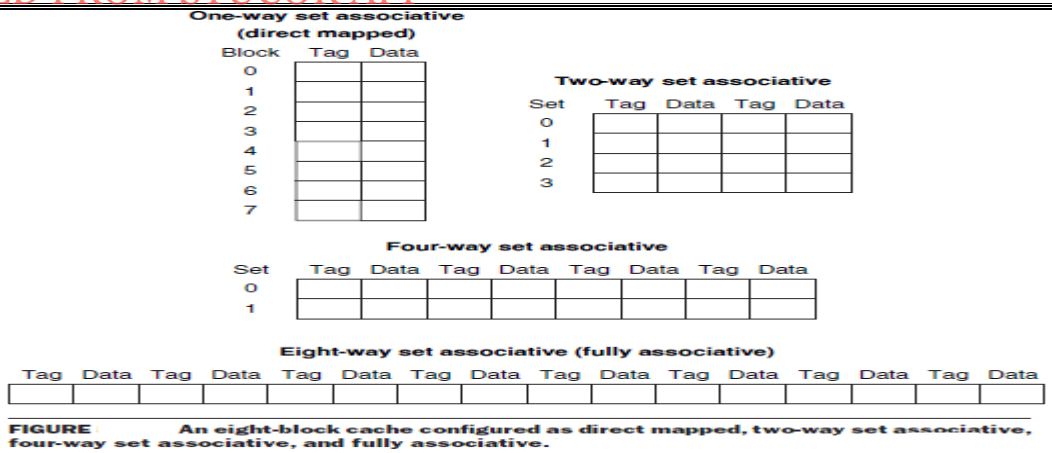


FIGURE An eight-block cache configured as direct mapped, two-way set associative, four-way set associative, and fully associative.

Example:1

Assume there are three small caches, each consisting of four one-word blocks. One cache is fully associative, a second is two-way set-associative, and the third is direct-mapped. Find the number of misses for each cache organization given the following sequence of block addresses: 0, 8, 0, 6, and 8.

Solution:

Direct Map: Number of blocks: 4

Block address	Cache block
0	(0 modulo 4) = 0
6	(6 modulo 4) = 2
8	(8 modulo 4) = 0

- The direct-mapped cache generates five misses for the five accesses.

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

Set Associative:

- The set-associative cache has two sets (with indices 0 and 1) with two elements per set. Let's first determine to which set each block address maps:

Block address	Cache set
0	(0 modulo 2) = 0
6	(6 modulo 2) = 0
8	(8 modulo 2) = 0

- Set-associative caches usually replace the least recently used block within a set;

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

- Notice that when block 6 is referenced, it replaces block 8, since block 8 has been less recently referenced than block 0. The two-way set-associative cache has four misses, one less than the direct-mapped cache.

Fully Associative:

- The fully associative cache has four cache blocks (in a single set); any memory block can be stored in any cache block.
- The fully associative cache has the best performance, with only three misses:

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

The implementation of a four-way set-associative cache requires four comparators and a 4-to-1 multiplexor.

- The following figure shows that in a four-way set-associative cache, four comparators are needed, together with a 4-to-1 multiplexor to choose among the four potential members of the selected set.
- 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 enables 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.

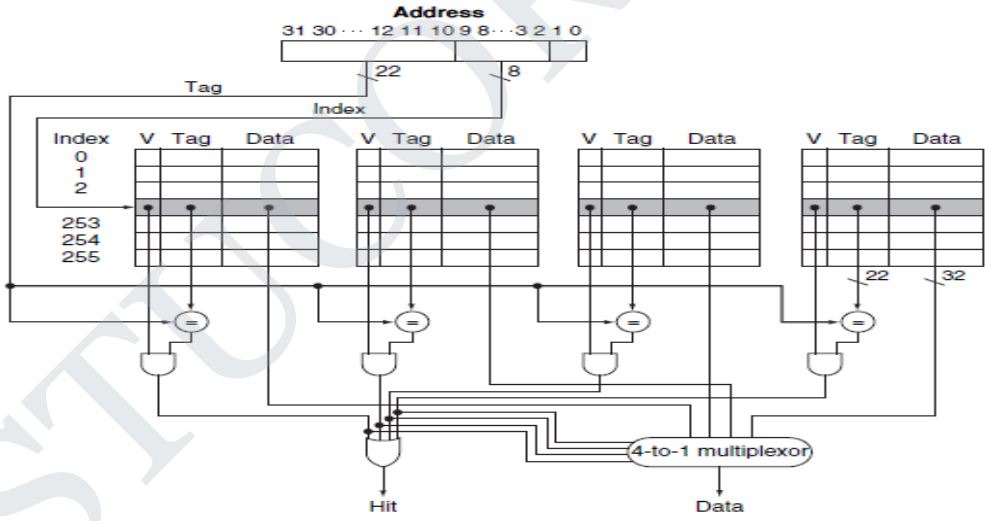


FIGURE The implementation of a four-way set-associative cache requires four comparators and a 4-to-1 multiplexor.

Example:

Increasing associativity requires more comparators and more tag bits per cache block. Assuming a cache of 4096 blocks, a 4-word block size, 16 bytes per block and a 32-bit address, find the total number of sets and the total number of tag bits for caches that are direct mapped, two-way and four-way set associative, and fully associative.

Solution:

- Number of bytes per block is : 16 [24]. So, 32-bit address yields 32-4=28 bits to be used for index and tag.

Direct Mapped cache:

- The direct-mapped cache has the same number of sets as blocks, and 12 bits of index (212=4096). Hence, the total number is (28-12) ×4096 =16 ×4096 =66 K tag bits.

Two-Way Set-Associative Cache:

- Thus, for a two-way set-associative cache, there are 2048 (211=2048) sets. The total number of tag bits is (28-11) × 2× 2048=34× 2048=70 Kbits.

Four-way set-associative cache:

- For a four-way set-associative cache, the total number of sets is 1024 (210=1024) sets. The total number is (28-10)×4×1024=72×1024=74 K tag bits.

Fully associative cache:

- For a fully associative cache, there is only one set with 4096 blocks, and the tag is 28 bits, leading to 28×4096×1=115 K tag bits.

Least Recently Used (LRU):

The algorithm which replaces the page that has not been used for the longest period of time is referred to as the *least recently used* (LRU) algorithm.

The result of applying LRU replacement to our example reference string is shown in Fig. The LRU algorithm produces 12 faults. It can be noticed that the first five faults are the same as the optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. The most recently used page is page 0, and just before that page 3 was used. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page 2, the LRU algorithm replaces page 3 since, of the three pages in memory {0, 3, 4}, page 3 is the least recently used. Despite these problems, LRU replacement with 12 faults is still much better than FIFO replacement with 15.

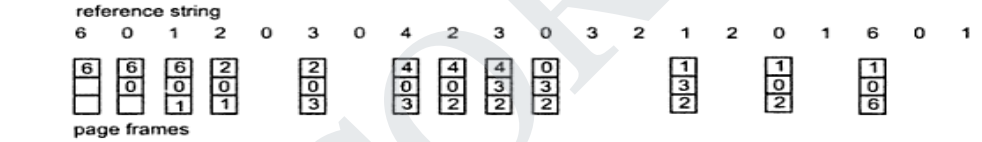


Fig. LRU page-replacement algorithm

First in First out (FIFO):

- A replacement scheme in which the block replaced is the one that has been entered first in the cache memory.

For our example reference string, our three frames are initially empty. The first three references (6, 0, 1) cause page faults, and and brought into these empty frames.

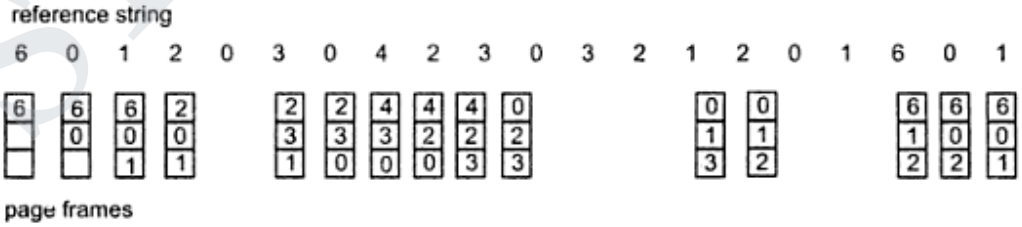


Fig. FIFO page-replacement algorithm

The next reference (2) replaces page 6, because page 6 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in page 0 being replaced, since it was the first of the three pages in memory (0, 1 and 2) to be brought in. This replacement means that the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Fig. 7.22. Every time a fault occurs, we show which pages are in out three frames. There are 15 faults algorithm.

MEASURING AND IMPROVING CACHE PERFORMANCE

- There are two different techniques for improving cache performance. The first one is **reducing the miss rate** by reducing the probability that two different memory blocks will contend for the same cache location.
- The second technique **reduces the miss penalty** by adding an additional level to the hierarchy. This technique, called multilevel caching.
- 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.
- Normally, we assume that the costs of cache accesses that are hits are part of the normal CPU execution cycles. Thus,

CPU time = (CPU execution clock cycles + Memory-stall clock cycles) × Clock cycle time

- The memory-stall clock cycles come primarily from cache misses. Memory-stall clock cycles can be defined as the sum of the stall cycles coming from reads plus those coming from write:

Memory-stall clock cycles = (Read-stall cycles + Write-stall cycles)

- The read-stall cycles can be defined in terms of the number of read accesses per program, the miss penalty in clock cycles for a read, and the read miss rate:

$$\text{Read-stall cycles} = \frac{\text{Reads}}{\text{Program}} \times \text{Read miss rate} \times \text{Read miss penalty}$$

- Writes are more complicated. For a write-through scheme, we have two sources of stalls:
- **Write Misses:** we fetch the block before continuing the write and write buffer stalls, which occur when the write buffer is full when a write occurs. Thus, the cycles stalled for writes equals the sum of these two:

$$\text{Write-stall cycles} = \left(\frac{\text{Writes}}{\text{Program}} \times \text{Write miss rate} \times \text{Write miss penalty}\right) + \text{Write buffer stalls}$$

- Because the write buffer stalls depend on the proximity of writes. Fortunately, in systems with a reasonable write buffer depth (e.g., four or more words) and a memory capable of accepting writes at a rate that significantly exceeds the average write frequency in programs (e.g., by a factor of 2), the write buffer stalls will be small, and we can safely ignore them.
- If a system did not meet these criteria, it would not be well designed; instead, the designer should have used either a deeper write buffer or a write-back organization.
- **Write-back**: schemes also have potential additional stalls arising from the need to write a cache block back to memory when the block is replaced. If we assume that the write buffer stalls are negligible, we can combine the reads and writes by using a single miss rate and the miss penalty:

$$\text{Memory-stall clock cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

We can also factor this as

$$\text{Memory-stall clock cycles} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Example:1

Calculating Average Memory Access Time

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.

The average memory access time per instruction is

$$\begin{aligned}\text{AMAT} &= \text{Time for a hit} + \text{Miss rate} \times \text{Miss penalty} \\ &= 1 + 0.05 \times 20 \\ &= 2 \text{ clock cycles}\end{aligned}$$

or 2 ns.

Handling Cache Miss

Cache miss

- It is a request for data from the cache that cannot be filled because the data is not present in the cache. 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 refetch the instruction, this time finding it in the cache.

Handling Writes

- **Inconsistent:** After the write into the cache, memory would have a different value from that in the cache. In such a case, the cache and memory are said to be **inconsistent.**
- **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 write buffer stores the data while it is waiting to be written to memory. After writing the data into the cache and into the write buffer, the processor can continue execution.

- When a write to main memory completes, the entry in the write buffer is freed. If the write buffer is full when the processor reaches a write, the processor must stall until there is an empty position in the write buffer.
- **Write Back:** In a write-back scheme, when a write occurs, the new value is written only to the block in the cache. The modified block is written to the lower level of the hierarchy when it is replaced.
- Write-back schemes can improve performance and more complex to implement than write-through.
- **Split cache:** A scheme in which a level of the memory hierarchy is composed of two independent caches that operate in parallel with each other, with one handling instructions and one handling data.

Example:2

Calculating Cache Performance

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%.

The number of memory miss cycles for instructions in terms of the Instruction count (I) is

$$\text{Instruction miss cycles} = I \times 2\% \times 100 = 2.00 \times I$$

As the frequency of all loads and stores is 36%, we can find the number of memory miss cycles for data references:

$$\text{Data miss cycles} = I \times 36\% \times 4\% \times 100 = 1.44 \times I$$

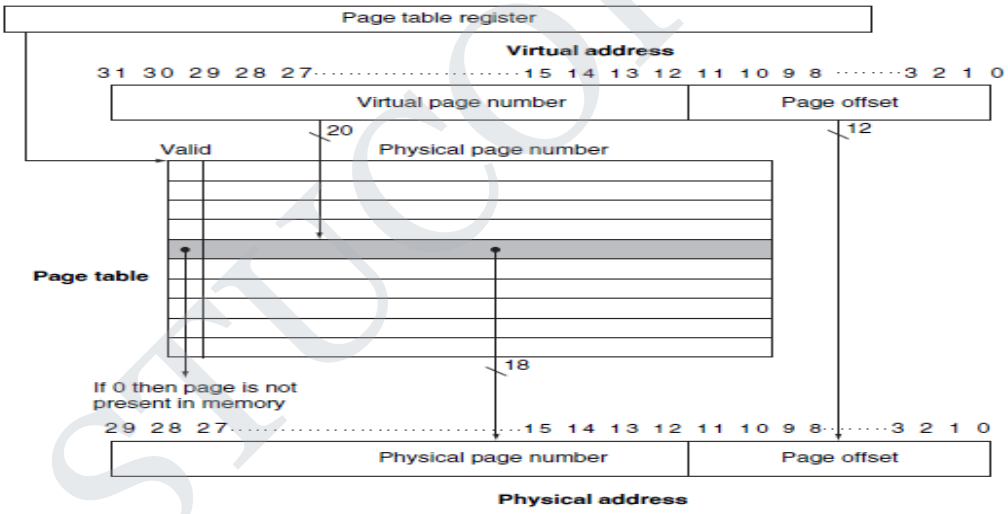
The total number of memory-stall cycles is 2.00 I + 1.44 I = 3.44 I. This is more than three cycles of memory stall per instruction. Accordingly, the total CPI including memory stalls is 2 + 3.44 = 5.44. Since there is no change in instruction count or clock rate, the ratio of the CPU execution times is

$$\frac{\text{CPU time with stalls}}{\text{CPU time with perfect cache}} = \frac{I \times \text{CPI}_{\text{stall}} \times \text{Clock cycle}}{I \times \text{CPI}_{\text{perfect}} \times \text{Clock cycle}} = \frac{\text{CPI}_{\text{stall}}}{\text{CPI}_{\text{perfect}}} = \frac{5.44}{2}$$

The performance with the perfect cache is better by $\frac{5.44}{2} = 2.72$.

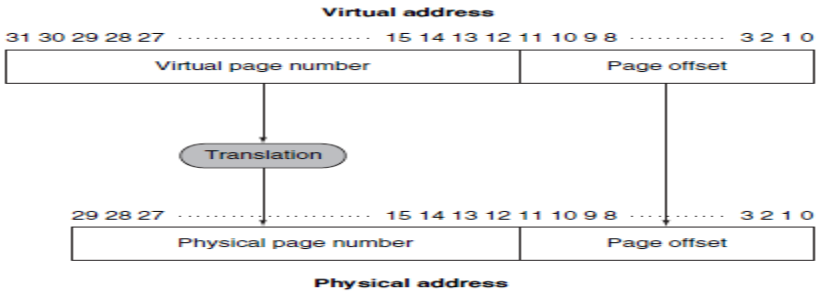
VIRTUAL MEMORY:

- **Virtual memory:** It is a technique that uses main memory as a "cache" for secondary storage.
- Virtual memory implements the translation of a program's address space to physical addresses. This translation process enforces protection of a program's address space from other virtual machines.
- **Protection**: It is 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.
- **Page fault:** It is an event that occurs when an accessed page is not present in main memory.
- **Address translation:** It is also called address mapping. The process by which a virtual address is mapped to an address used to access memory.
- In virtual memory, the address is broken into a virtual page number and a page offset. Figure shows the translation of the virtual page number to a physical page number.
- The physical page number constitutes the upper portion of the physical address, while the page off set, which is not changed, constitutes the lower portion.
- The number of bits in the page off set field determines the page size. The number of pages addressable with the virtual address need not match the number of pages addressable with the physical address.

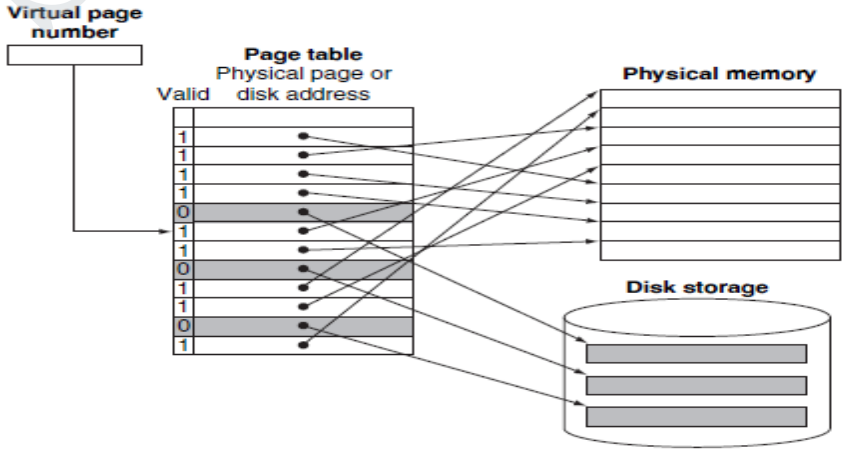


- **Pages:** Both the virtual memory and the physical memory are broken into pages, so that a virtual page is mapped to a physical page.
- Physical pages can be shared by having two virtual addresses point to the same physical address. This capability is used to allow two different programs to share data or code.
- **Segmentation**: It is a variable-size address mapping scheme in which an address consists of two parts: a segment number, which is mapped to a physical address, and a segment off set.
- **Page table**: It is 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.

Mapping from virtual address into physical address

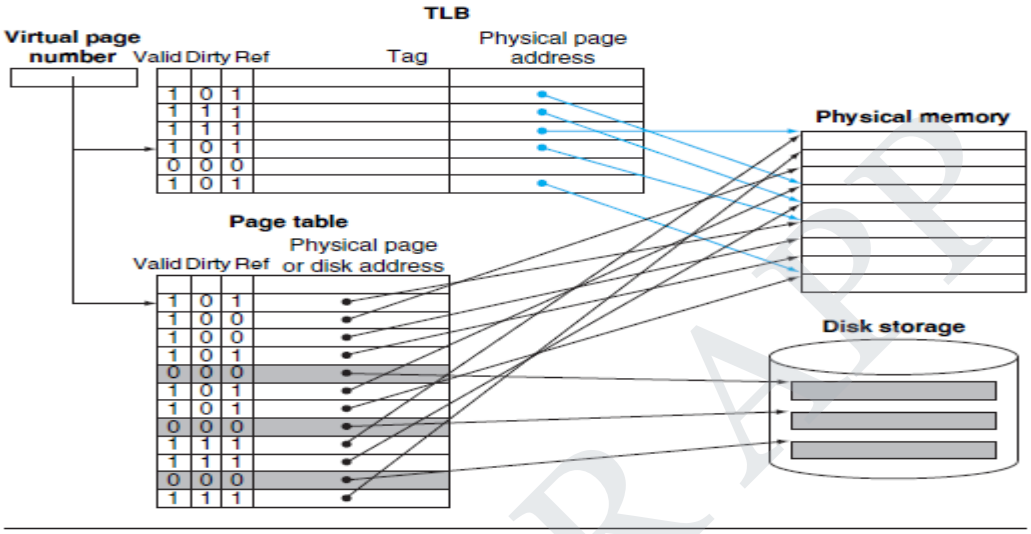


- The difficulty in using fully associative placement is in locating an entry, since it can be anywhere in the upper level of the hierarchy. A full search is impractical.
- In virtual memory systems, we locate pages by using a table that indexes the memory; this structure is called a page table, and it resides in memory.
- A page table is indexed with the page number from the virtual address to discover the corresponding physical page number. Each program has its own page table, which maps the virtual address space of that program to main memory.
- **Page Table Register:** To indicate 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**. Assume for now that the page table is in a fixed and contiguous area of memory.
- When a page fault occurs, if all the pages in main memory are in use, the operating system must choose a page to replace.
- Because 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 operating system searches for the least recently used page, assuming that a page that has not been used in a long time is less likely to be needed than a more recently accessed page. The replaced pages are written to swap space on the disk.
- **Swap space:** It is the space on the disk reserved for the full virtual memory space of a process.
- **Reference bit:** It is also called use bit. A field that is set whenever a page is accessed and that is used to implement LRU or other replacement schemes.



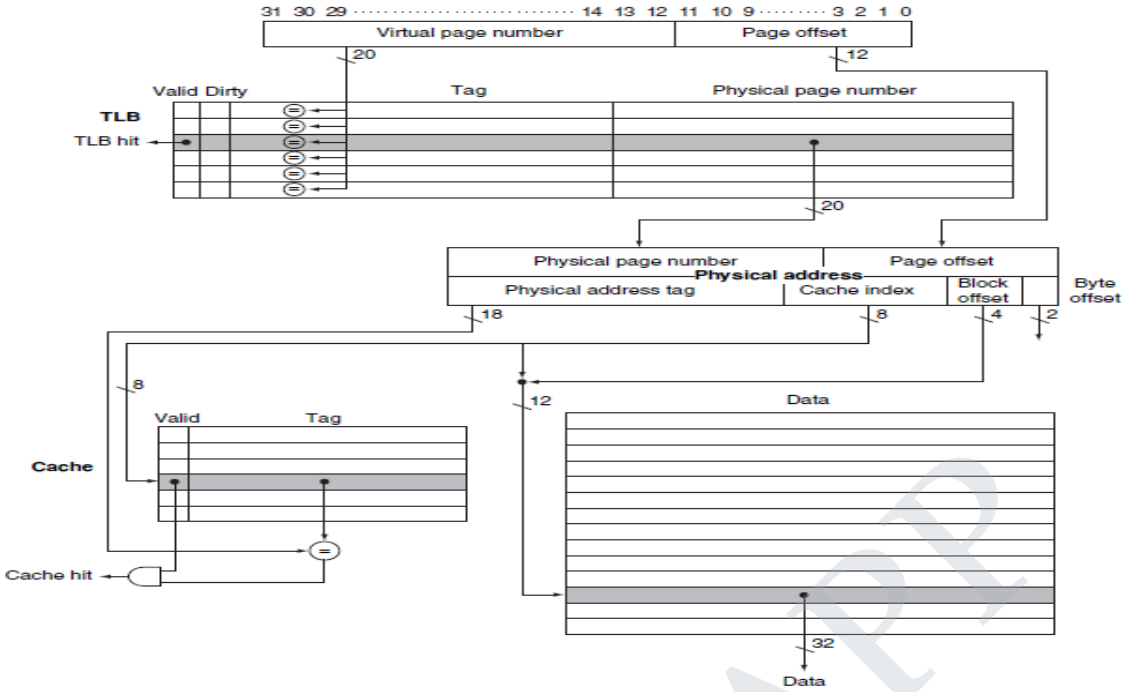
TRANSLATION-LOOKASIDE BUFFER (TLB):

- **Translation-lookaside buffer (TLB)** a cache that keeps track of recently used address mappings to try to avoid an access to the page table.
- Accordingly, 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.
- Figure shows that each tag entry in the TLB holds a portion of the virtual page number, and each data entry of the TLB holds a physical page number.

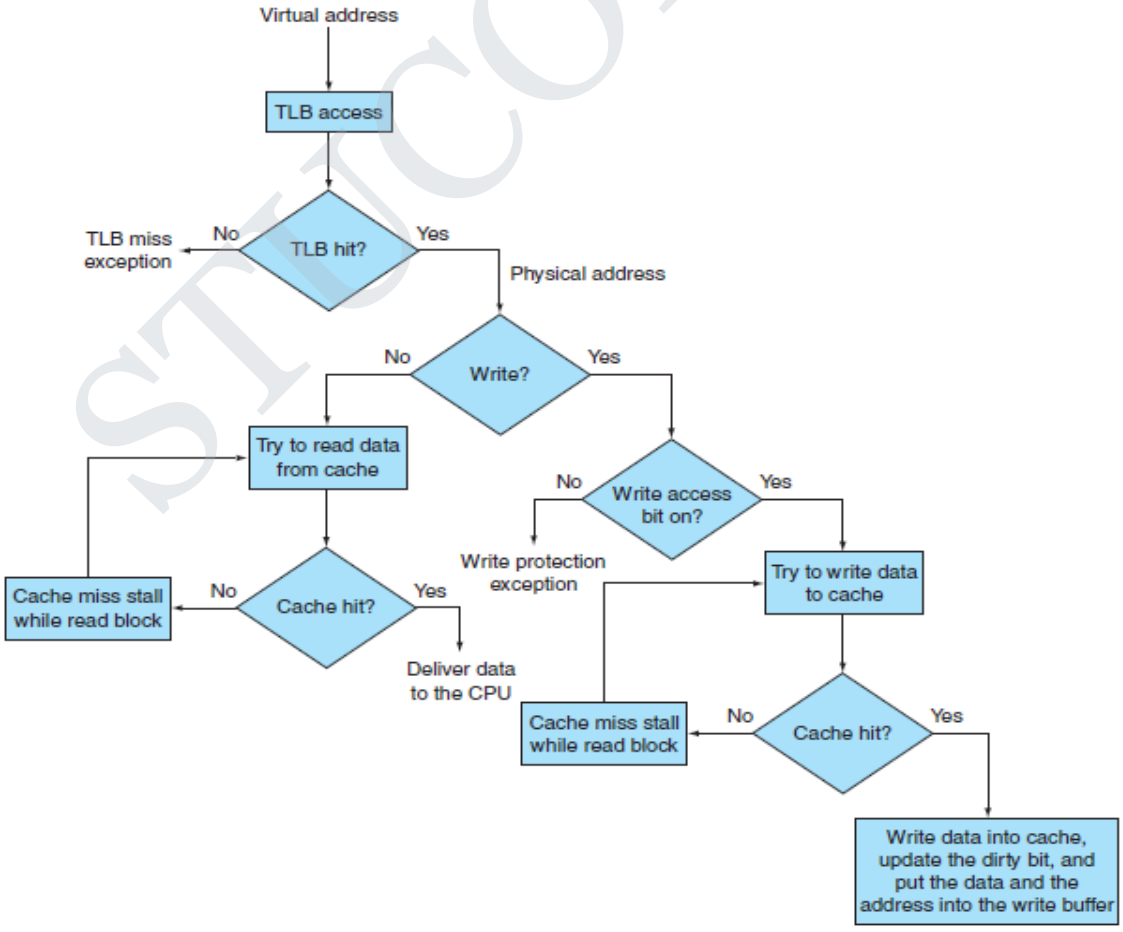


- Because we access the TLB instead of the page table on every reference, the TLB will need to include other status bits, such as the dirty and the reference bits. On every reference, we look up the virtual page number in the TLB.
- **Reference Bit:** If we get a hit, the physical page number is used to form the address, and the corresponding reference bit is turned on.
- **Dirty Bit:** If the processor is performing a write, the dirty bit is also turned on. If a miss in the TLB occurs, we must determine whether it is a page fault or purely a TLB miss.
- If the page exists in memory, then the TLB miss indicates only that the translation is missing. In such cases, the processor can handle the TLB miss by loading the translation from the page table into the TLB and then trying the reference again.
- **True page Fault:** If the page is not present in memory, then the TLB miss indicates a true page fault. In this case, the processor invokes the operating system using an exception. Because the TLB has many fewer entries than the number of pages in main memory, TLB misses will be much more frequent than true page faults.
- TLB misses can be handled either in hardware or in software.
- After a TLB miss occurs and the missing translation has been retrieved from the page table, we will need to select a TLB entry to replace.
- Because the reference and dirty bits are contained in the TLB entry, we need to copy these bits back to the page table entry when we replace an entry. These bits are the only portion of the TLB entry that can be changed.
- Using write-back—that is, copying these entries back at miss time rather than when they are written—is very efficient, since we expect the TLB miss rate to be small.

- Some systems use other techniques to approximate the reference and dirty bits, eliminating the need to write into the TLB except to load a new table entry on a miss.

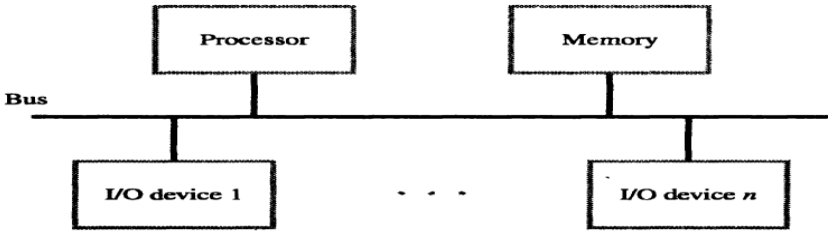


- **Virtually addressed cache** a cache that is accessed with a virtual address rather than a physical address.
- **Aliasing** a situation in which two addresses access the same object; it can occur in virtual memory when there are two virtual addresses for the same physical page.
- Physically addressed cache a cache that is addressed by a physical address.



ACCESSING I/O DEVICES

- A bus is a shared communication link, which uses one set of wires to connect multiple subsystems. Most modern computers use single bus arrangement for connecting I/O devices to CPU & Memory. The bus enables all the devices connected to it to exchange information.



Advantages

1. Versatility 2. Low cost.

Bus consists of 3 set of lines :

1. **Address:** Processor places a unique address for an I/O device on address lines.
2. **Data:** The data will be placed on Data lines. The data lines of the bus carry information between the source and the destination. This information may consist of data, complex commands, or addresses.
3. **Control:** The control lines are used to signal requests and acknowledgments, and to indicate what type of information is on the data lines. Processor requests for either Read / Write.

INPUT OUTPUT SYSTEM

1. Memory mapped I/O 2.Programmed I/O

1. Memory mapped I/O

- I/O devices and the memory share the same address space, the arrangement is called Memory-mapped I/O. In Memory-mapped I/O portions of address space are assigned to I/O devices and reads and writes to those addresses are interpreted as commands to the I/O device.

2. Programmed I/O

- Programmed I/O is a method included in every computer for controlling I/O operation. It is most useful in small, low speed system where hardware cost must be minimized.
- Programmed I/O requires all I/O operation can be executed under the direct control of the CPU.
- Generally data transfer takes place between two registers. One is CPU register and other register is attached in I/O device. I/O device does not have direct access to main memory.
- A data transfer from an I/O device to memory required CPU to execute several instructions.
- Input instruction to transfer a word from the I/O device to the CPU and store instruction to transfer the word from the CPU to memory.

Techniques:

1. I/O Addressing 2. I/O-Mapped I/O 3. I/O Instruction 4. I/O Interface Circuit

INTERRUPTS

- The word interrupt is used to cause a CPU to temporarily transfer control from its current program to another program.
- When I/O Device is ready, it sends the INTERRUPT signal to processor via a dedicated controller line. In response to the interrupt, the processor executes the Interrupt Service Routine (ISR).
- All the registers, flags, program counter values are saved by the processor before running ISR.
- The time required to save status & restore contribute to execution overhead is called **"Interrupt Latency".**
- Various internal and external sources generate interrupts to the CPU. IO interrupts are external requests to the CPU to initiate or terminate an operation.
- If a power supply failure occurs then interrupt can be generated and interrupt handler to save critical data about the system's state, it is a kind of **hardware interrupt.**
- An instruction to divide by zero is an examples of **software interrupt.**

Steps:

1. The processor completes its current instruction. No instruction is cut off in the middle of its execution.
2. The program counter's current contents are stored on the stack. Remember, during the execution of an instruction the program counter is pointing to the memory location for the next instruction.
3. The program counter is loaded with the address of an interrupt service routine.
4. Program execution continues with the instruction taken from the memory location pointed by the new program counter contents.
5. The interrupt program continues to execute until a return instruction is executed.
6. After execution of the RET instruction processor gets the old address (the address of the next instruction from where the interrupt service routine was called

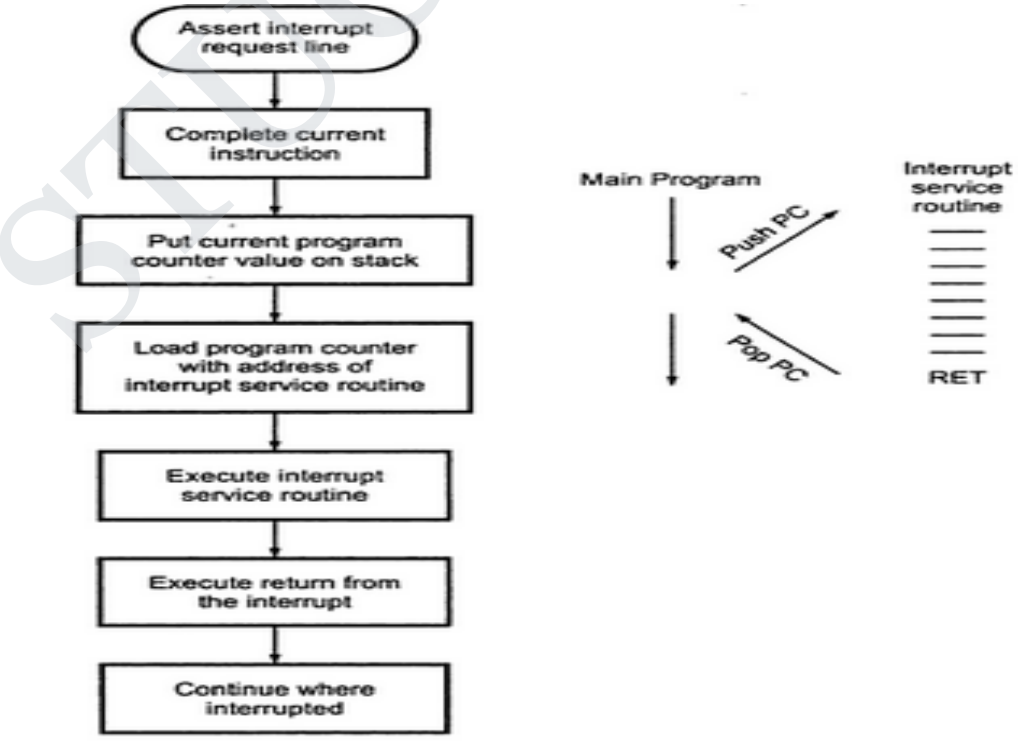


Fig. Response to an interrupt with the flowchart and diagram

Handling Multiple Devices

- Multiple devices can initiate interrupts. They uses the common interrupt request line Techniques are
    1. Maskable and Nonmaskable interrupt
    2. Single line interrupt(polling)
    3. Multilevel interrupt
    4. Vectored Interrupts
    5. Interrupt Nesting
    6. Daisy Chaining
    7. Daisy Chaining with priority group
    8. Pipeline Interrupt

1. Maskable and Nonmaskable interrupt

- Maskable interrupts are enabled and disabled under program control.
- By setting or resetting particular flip-flops in the processor, interrupts can be masked or unmasked.
- When masked processor does not respond to the interrupt even though the interrupt is activated. Most of the processor provides the masking facility.
- In the processor those interrupts which can be masked under software control are called **maskable interrupts.**
- In the processor those interrupts which cannot be masked under software control are called **nonmaskable interrupts.**
- Three methods of Controlling Interrupts (single device)
    1. Ignoring interrupt
    2. Disabling interrupts
    3. Special Interrupt request line

1. Ignoring Interrupts

- Processor hardware ignores the interrupt request line until the execution of the first instruction of the ISR completed. Using an interrupt disable instruction after the first instruction of the ISR. A return from interrupt instruction is completed before further interruptions can occur.

2. Disabling Interrupts

- Processor automatically disables interrupts before starting the execution of the ISR.
- The processor saves the contents of PC and PS (status register) before performing interrupt disabling. The interrupt-enable is set to 0 means no further interrupts allowed.
- When return from interrupt instruction is executed the contents of the PS are restored from the stack, and the interrupt enable is set to 1

3. Special Interrupt line

- Special interrupt request line for which the interrupt handling circuit responds only to the leading edge of the signal Edge –triggered.
- Processor receives only one request regardless of how long the line is activated.
- No separate interrupt disabling instructions.

2. Single Line Interrupt

- In a single interrupts there can be many interrupting devices. But all interrupt requests are made via a single input pin of the CPU.
- In single line interrupt system, all IO ports share a single INTERRUPT REQUEST line. On responding to an interrupt request, the CPU must scan all the IO devices to determine the source of the interrupt.
- This procedure requires activating an INTERRUPT ACKNOWLEDGE line and that is connected in daisy chain fashion to all IO devices.
- Once the interrupting I/O port is identified, the CPU will service it and then return to task it was performing before the interrupt.
- Polling software routine that checks the logic state of each device. The Interrupt service routine polls the I/O devices connected to the bus.
- Main advantage of polling is it allowing the interrupt priority to be programmed.

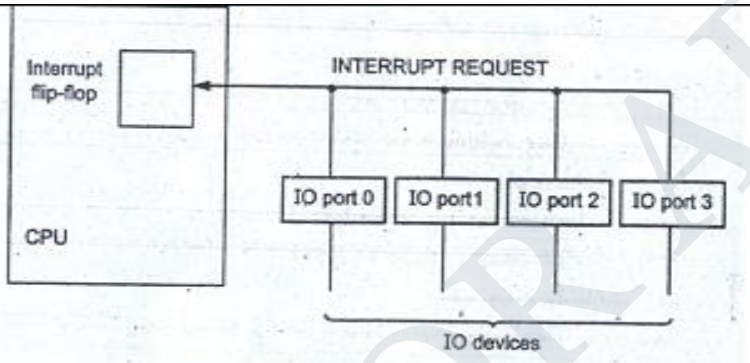


Fig. Single-line interrupt system

3. Multilevel Interrupt

- In multiple level interrupts, processor has more than one interrupt pins.
- The I/O devices are tied to the individual interrupt pins.
- The interrupts can be easily identified by the CPU upon receiving an interrupt request from it.
- This allows processor to go directly to that I/O device and service it without having to poll first.
- Advantage is saves the time in processing interrupt.

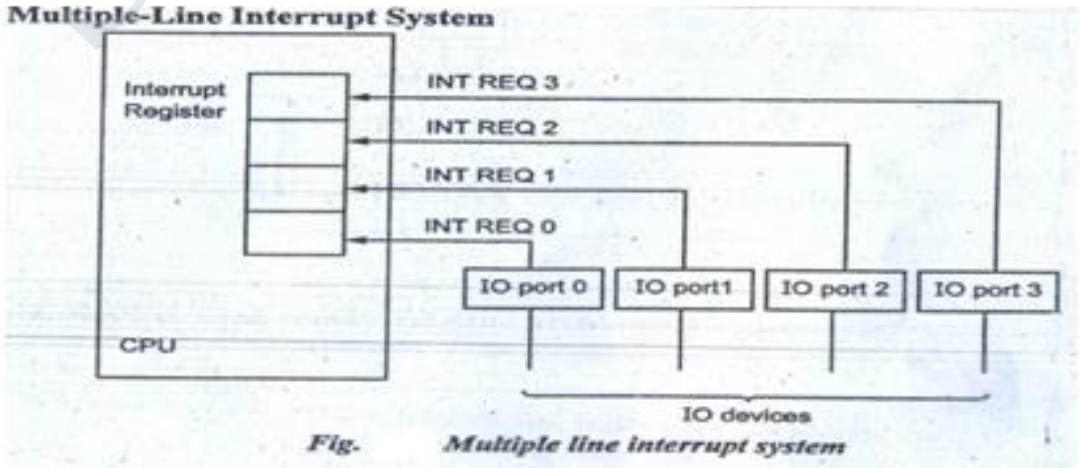


Fig. Multiple line interrupt system

4. Vectored Interrupts

- The following figure shows a basic way to derive interrupt vectors from multiple interrupt request lines.
- Each interrupt request line generates a unique fixed address and it is used to modify the CPU's program counter PC.
- Interrupt requests are stored on receipt in an interrupt register. The interrupt, mask register can disable any or all of the interrupt request lines under program control.
- The K masked interrupt signals are fed into a priority encoder that produces a [log2K] bit address and it is inserted into PC. PC finds the ISR address from the code.
- Device requesting an interrupt identifies itself directly to the processor. The device sends a special code to the processor over the bus.
- The code contains the
    1. Identification of the device
    2. Starting address for the ISR
    3. Address of the branch to the ISR

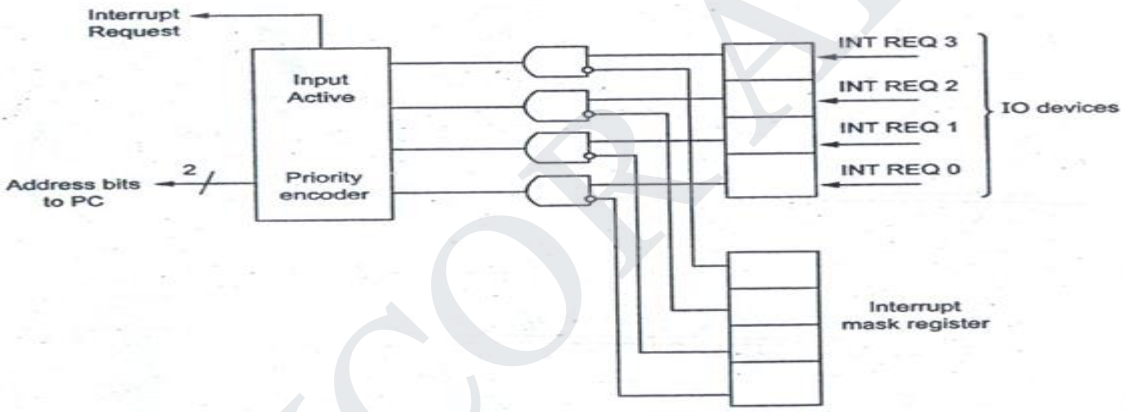
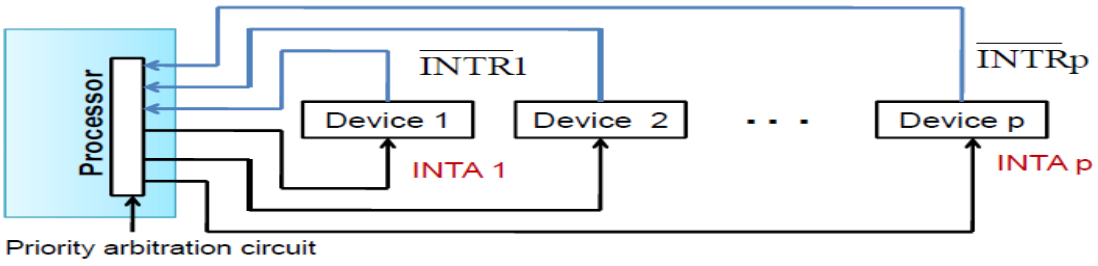


Fig. A vectored interrupt scheme

5. Interrupt Nesting

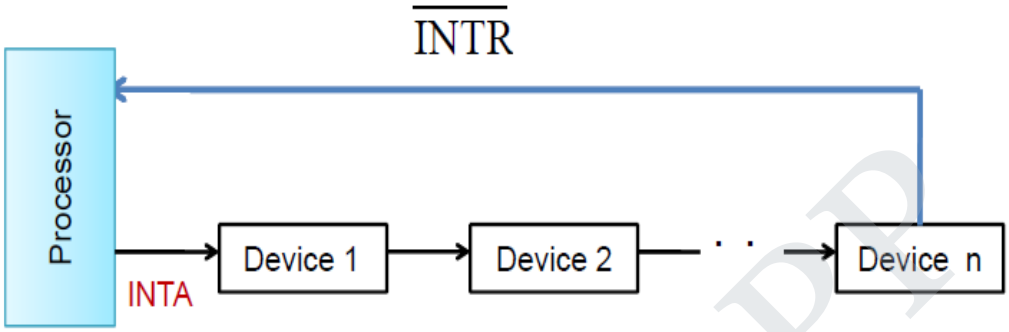
- Pre-Emption of low priority Interrupt by another high priority interrupt is known as Interrupt nesting.
- Need a priority of IRQ devices and accepting IRQ from a high priority device.
- The priority level of the processor can be changed dynamically.
- The privileged instruction write in the PS (processor status word), that encodes the processors priority. Organizing I/O devices in a prioritized structure.
- Each of the interrupt-request lines is assigned a different priority level. The processor is interrupted only by a high priority device.

Interrupt Nesting (contd.)



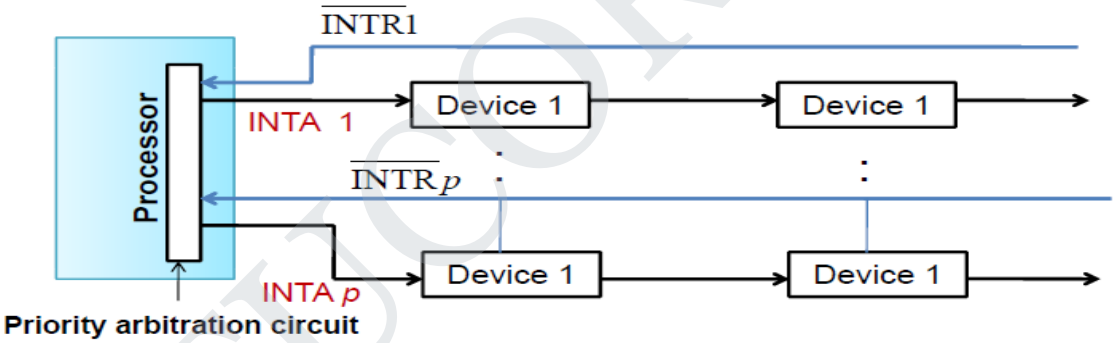
6. Daisy Chaining

- The interrupt request line INTR is common to all the devices.
- The interrupt acknowledgement line INTA is connected to devices in a DAISY CHAIN way.
- INTA propagates serially through the devices. Device that is electrically closest to the processor gets high priority.
- Low priority device may have a danger of STARVATION.



7. Daisy Chaining with Priority Group

- Combining Daisy chaining and Interrupt nesting to form priority group.
- Each group has different priority levels and within each group devices are connected in daisy chain way.



Arrangement of priority groups

8. Pipeline Interrupts

- In a pipelined processor, many instructions are executed at same time so it is difficult to find the interrupting instruction.
- In a pipelined process one instruction can finish sooner than another instruction that was issued earlier. Let's consider three different kinds of instructions floating point representation such as
  1. Multiply
  2. Add1
  3. Add2
- Multiply instruction has 7 clock cycles and Add 1 and 2 have four clock cycles.
- In Fig.(a) Add 1 instruction completes their execution before multiply instruction, even though multiply instruction starts their execution 1 cycle earlier than Add 1.

- Here no hazard occurs due to the data dependencies. Suppose add1 generates an interrupt due to a result overflows in its execution stage EX corresponding to cycle4.

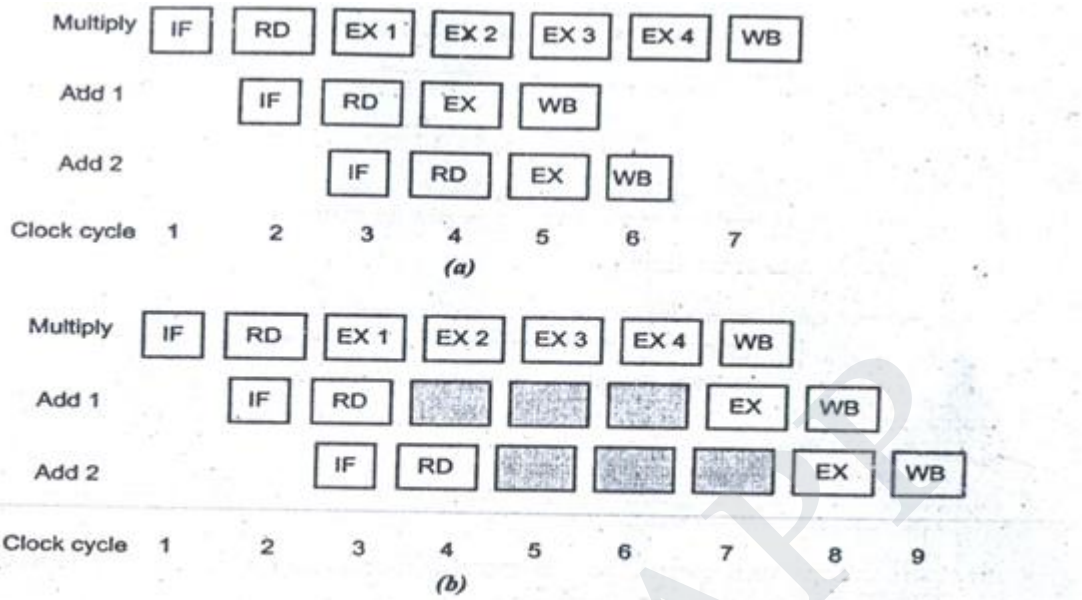


Fig. Instruction processing in a pipeline with (a) out-of-order completion (b) in-order completion

- Now we need transfer the control to an interrupt handler designed to service adder overflow. It is also possible that the ongoing multiply instruction will generate another interrupt in cycle6.
- This second interrupt can change the CPU's state to prevent proper processing of the first interrupt.
- Registers affected by add1 instruction can be further modified by the multiply interrupt so proper recovery from the add1 instruction may not be possible.
- In such kinds of situation the CPU state is said to have become **imprecise.**
- **Precise interrupt** is one where the system state information or correct transfer of control to the interrupt handler and correct return to the interrupting program is always preserved.

DIRECT MEMORY ACCESS (DMA)

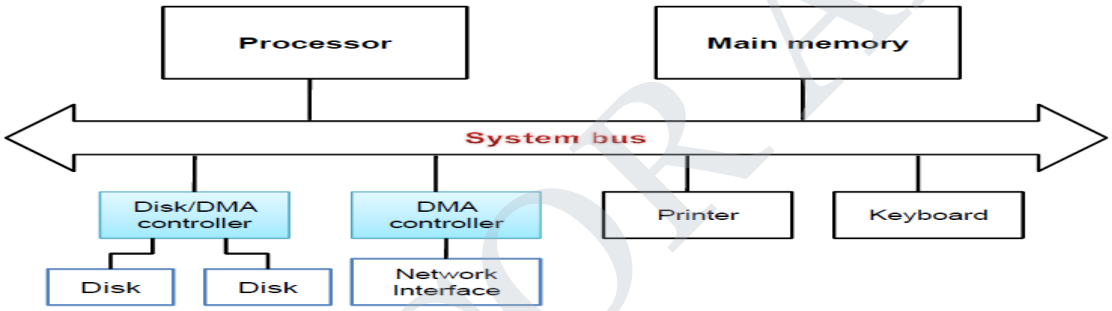
- To transfer large blocks of data at high Speed, between EXTERNAL devices & Main Memory, DMA approach is often used.
- DMA controller allows data transfer directly between I/O device and Memory, with minimal intervention of processor.
- DMA controller acts as a Processor, but it is controlled by CPU.
- To initiate transfer of a block of words, the processor sends the following data to controller
    - The starting address of the memory block
    - The word count
    - Control to specify the mode of transfer such as read or write
    - A control to start the DMA transfer

- DMA controller performs the requested I/O operation and sends a interrupt to the processor upon completion

	31	30		1	0
Status and Control	**IRQ**	**IE**		**R/W**	**Done**
Starting address					
Word count					

- In DMA interface First register stores the starting address, Second register stores Word count and Third register contains status and control flags.

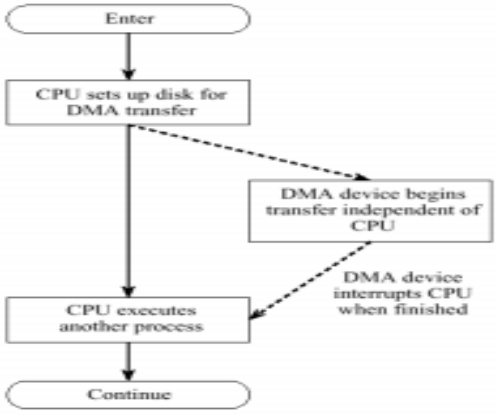
Bits and Flags	1	0
R/W	READ	WRITE
Done	Data transfer finishes	
IRQ	Interrupt request	
IE	Raise interrupt (enable) after Data Transfer	



Use of DMA Controller in a computer system

- Memory accesses by the processor and DMA Controller are interleaved.
- DMA devices have higher priority than processor over BUS control.
- **Cycle Stealing:** DMA Controller "steals" memory cycles from processor, though processor originates most memory access.
- **Block or Burst mode:** The DMA controller may give exclusive access to the main memory to transfer a block of data without interruption.

DMA Flowchart for a Disk Transfer



- **Conflicts in DMA:**
  1. Processor and DMA.
  2. Two DMA controllers, try to use the Bus at the same time to access the main memory.

The hardware needed to implement DMA is shown in the following Figure.

- The circuit assumes that all access to main memory is via a shared system bus.
- The IO device is connected to the system bus via a special interface circuit called DMA controller.
- It contains a data buffer register IODR, it also controls an address register IOAR and a data count register DC.
- These registers enable the DMA controller to transfer data to or from a contiguous region of memory.
- IOAR stores the address of the next word to be transferred. It is automatically incremented or decremented after each word transfer.
- The data counter DC stores the number of words that remain to be transferred.
- Data counter is automatically incremented or decremented after each transfer and tested for zero, when DC reaches zero the DMA transfer halls.

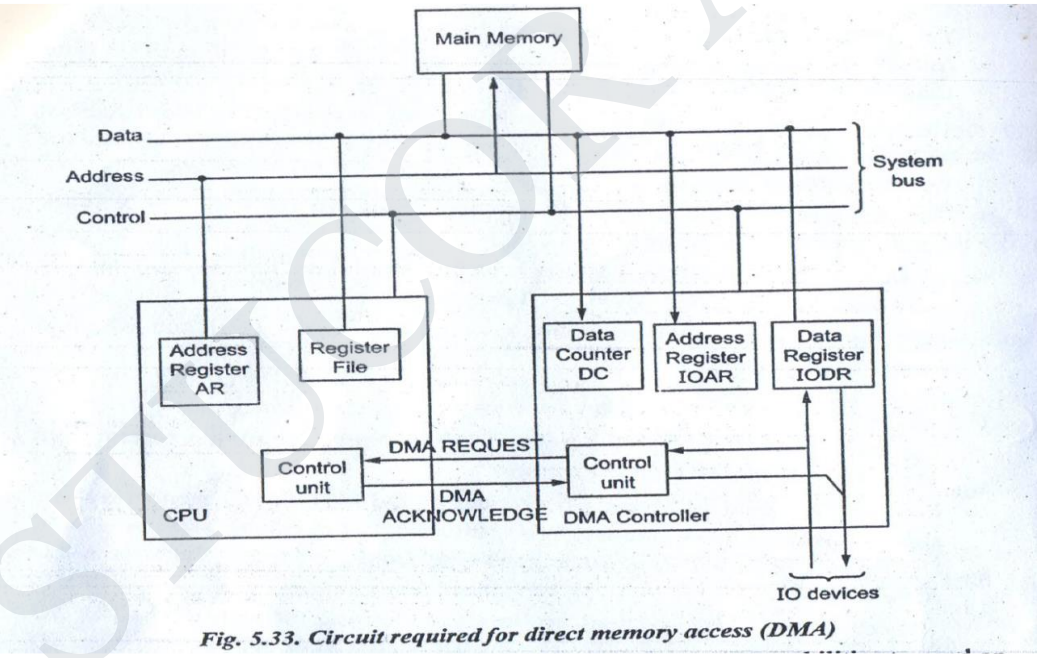


Fig. 5.33. Circuit required for direct memory access (DMA)

- The DMA controller is normally provided with interrupt capabilities to send an interrupt to the CPU to signal the end of the IO data transfer.
- A DMA controller can be designed to supervise DMA transfers involving several IO devices, each device has a different priority of access to the system bus. Under DMA control data can be transferred in several different ways.
- In a DMA block transfer a data word is a sequence of arbitrary length that is transferred in a single burst while the DMA controller is master of the memory bus.
- This DMA mode needs the secondary memories like disk drives, where data transmission cannot be stopped or slowed without loss of data.

- **Block DMA transfer** supports the fastest IO data transfer rates but it can make the CPU inactive for relatively long periods by tying up the system bus.
- **Cycle stealing:** An alternative technique called cycle stealing; it allows the DMA controller to use the system bus to transfer one word data after it must 'return control of the bus to the CPU.
- Cycle stealing reduces, the maximum IO transfer rate and also reduces the interference by the DMA controller in the CPU's memory access.
- **Transparent DMA:** It is possible to eliminate this interference completely by designing the DMA interfaces. CPU is not actually using the system bus. This is called transparent DMA.
- The above shown DMA circuit has the following DMA transfer process:

Step 1:

- The CPU executes two IO instructions, which load the DMA' registers IOAR and DC with their initial values.
- IOAR register should contain the base address of the memory region to be used in the data transfer.
- DC register should contain the number of words to be transferred to' or from that region.

Step 2:

- When the DMA controller is ready to transmit or receive data, it activates the DMA REQUEST line to the CPU.
- The CPU waits for the next DMA breakpoint.
- DMA controller relinquishes control of the data and address lines and activates DMA ACKNOWLEDGE.
- DMA REQUEST and DMA ACKNOWLEDGE are essentially used in BUS REQUEST and BUS GRANT lines for control of the system bus.
- DMA requests from several DMA controllers are resolved by bus priority control techniques.

Step 3:

- Now DMA controller transfers data directly to or from main memory. After transferring a word, IOAR and DC registers are updated.

Step 4:

- If DC has not yet reached zero but, the 10 device is not ready to send' or receive the next batch of data then the DMA-controller will release the system bus to the CPU by deactivating the DMA REQUEST line.
- The CPU responds by deactivating DMA ACKNOWLEDGE and resuming control of the system bus.

Step 5:

- If DC register is decremented to zero then the DMA controller again relinquished control of the system bus.
- It may also send an interrupt request signal to the CPU. The CPU responds by halting the IO device or by initiating a new DMA transfer.

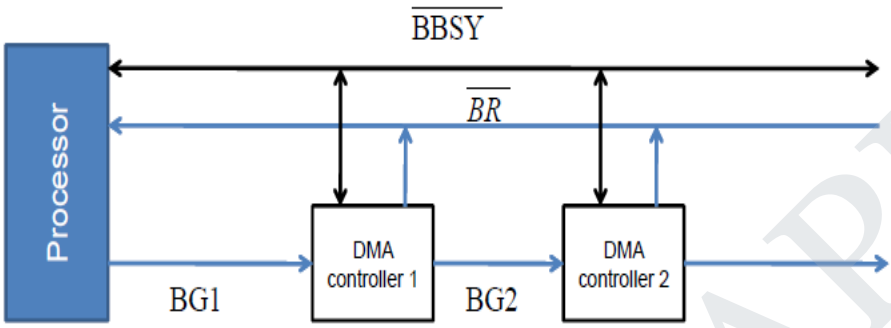
BUS ARBITRATION

- **Bus master:** Device that initiates data transfers on the bus.
- The next device can take control of the bus after the current master surrenders control.
- **Bus Arbitration:** Process by which the next device to become master is selected.

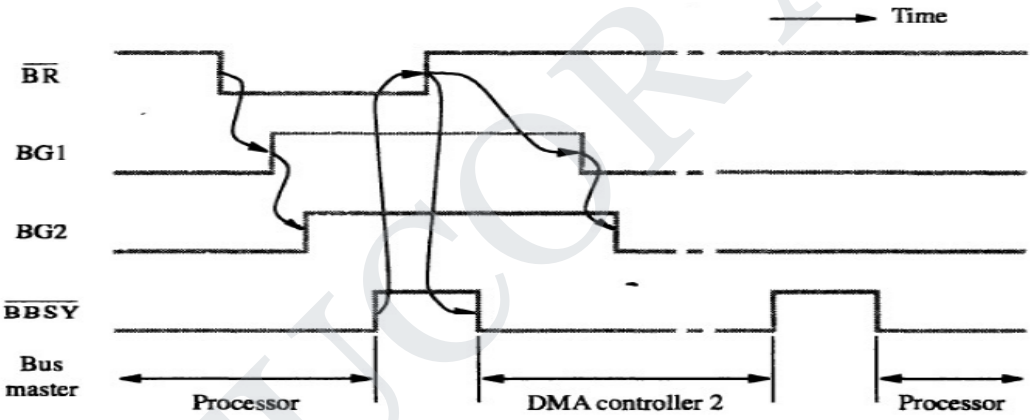
Types of bus arbitration:

1. Centralized and Distributed Arbitration
2. Distributed Arbitration

Centralized and Distributed Arbitration



A simple arrangement for bus arbitration using a daisy chain

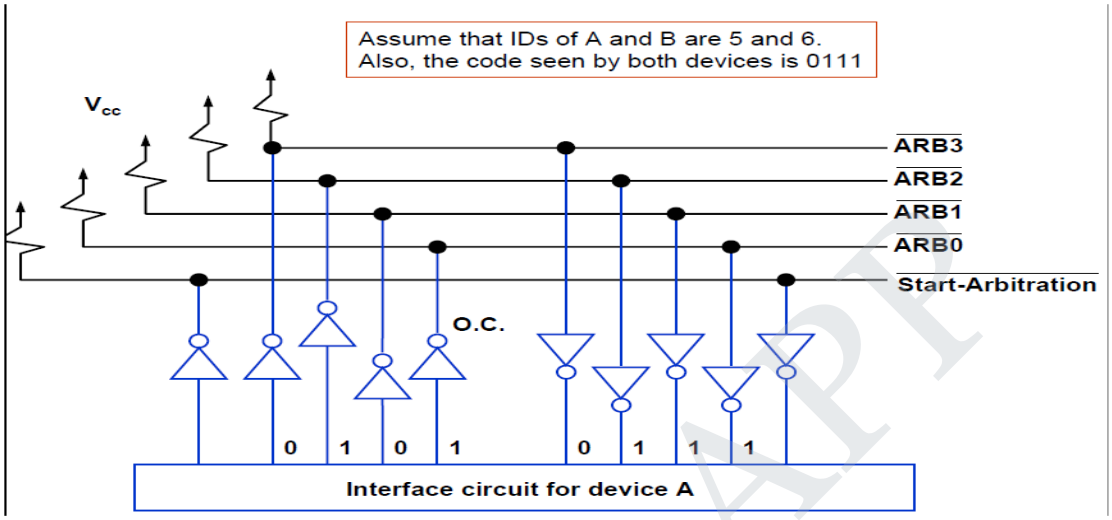


Sequence of signals during data transfer of bus mastership

- BR (Bus Request) line is the open drain line and the signal on this line is a logical OR of the bus request from all the DMA devices.
- BG (Bus Grant) line is processor activates this line indicating (acknowledging) to all the DMA devices (connected in daisy chain fashion) that the BUS may be used when it's free.
- BBSY (Bus Busy) line is an open collector line. The current bus master indicates devices that it is currently using the bus by signaling this line.
- Processor is normally the bus master, unless it grants bus mastership to DMA.
- For the timing/control, in the above diagram DMA controller 2 requests and acquires bus mastership and later releases the bus.
- During its tenure as the bus master, it may perform one or more data transfer operations, depending on whether it is operating in the cycle stealing or block mode.
- After it releases the bus, the processor resumes bus mastership.

Distributed Arbitration

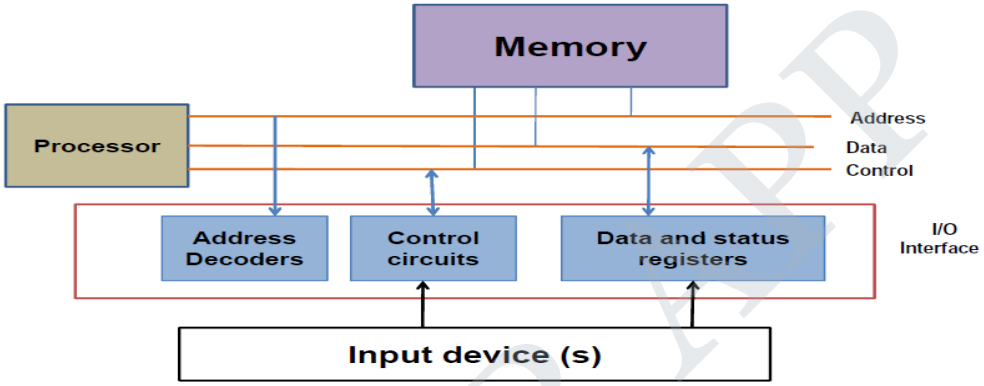
- All devices waiting to use the bus have to carry out the arbitration process and it has no central arbiter.
- Each device on the bus is assigned with a 4-bit identification number.
- One or more devices request the bus by asserting the start-arbitration signal and place their identification number on the four open collector lines.



- ARB0 through ARB3 are the four open collector lines.
- One among the four is selected using the code on the lines and one with the highest ID number.
- Assume that two devices, A and B, having ID numbers 5 and 6, respectively, are requesting the use of the bus.
- Device A transmits the pattern 0101, and device B transmits the pattern 0110. The code seen by both devices is 0111.
- Each device compares the pattern on the arbitration lines to its own ID, starting from the most significant bit.
- If it detects a difference at any bit position, it disables its drivers at that bit position and for all lower-order bits. It does so by placing a 0 at the input of these drivers.
- In the case of our example, device A detects a difference on line ARB I. Hence, it disables its drivers on lines ARB 1 and ARBO.
- This causes the pattern on the arbitration lines to change to 0110, which means that B has won the contention.

Interface Circuit

- The interface circuit contains the following units. Namely,
  1. Address Decoder
  2. Control Circuits
  3. Data registers and Status registers
- The register in I/O Interface is buffer and control. There are two flags in Status Registers, like SIN, SOUT and two Data Registers, like Data-IN, Data-OUT.
- Status register SIN, SOUT provides status information for keyboard and display unit.
- When the processor places the address and data on the memory bus, the memory system ignores the operation because the address indicates a portion of the memory space used for I/O.



- DATAIN is the address of the input buffer associated with the keyboard.

  Move DATAIN, R0; reads the data from DATAIN and stores them into processor registerR0

  Move R0, DATAOUT; sends the contents of register R0 to location DATAOUT

- The device controller, sees the operation, records the data, and transmits it to the device as a command.
- User programs are prevented from issuing I/O operations directly because the OS does not provide access to the address space assigned to the I/O devices and thus the addresses are protected by the address translation.