

UNIT 1

OVERVIEW AND INSTRUCTION

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

COMPUTER ARCHITECTURE

- **Computer Architecture** deals with designing and implementation of instruction set, information format and memory addressing techniques of a computer.
- **Computer Organization** refers to the operational units and their interconnections that describe the function and design of various units of a computer.
- A **Computer Architect** performs instruction set design, and memory addressing modes.

1.1 EIGHT IDEAS

In the last 60 years of computer design, computer architects have proposed 8 great ideas. They are,

1. Design for Moore's Law
2. Use Abstraction to Simplify Design
3. Make the Common Case Fast
4. Performance via Parallelism
5. Performance via Pipelining
6. Performance via Prediction
7. Hierarchy of Memories
8. Dependability via Redundancy

1.1.1 Design for Moore's law

- Developed by Gordon E. Moore, co-founder of Intel.
- The design of a computer takes many years.

- The design of a system may start with an existing technology
- At the end of the product, the technology may grow and the product has to be reworked.
- Hence, computer architects must imagine about the future technology (technology at the finish time of the project) rather than designing with the existing one.
- Moore's Law graph is given by



- The graph represents the concept: “up and to the right”, which means that the technology changes rapidly.

1.1.2 Use Abstraction to Simplify Design

- Computer architects and programmers use abstractions (Generalization of concepts) to represent the design at several levels.
- The detail represented at each level hides the details of lower levels.
- This may improve productivity since abstraction simplifies design and thus the design time decreases.
- This provides a simpler design model due to abstraction.
- Example
 - Operating systems hide the details involved in handling input and output devices.
 - High-level languages hide the details of the sequence of instructions need to accomplish a task.

1.1.3 Make the common case fast

- The performance shall be improved by improving the common case rather than developing the rare case.
- This makes the design process simpler and faster.
- The concept is often called the Amdahl's law.
- Example
 - It is easier to design a sports car having a capacity of one / two passengers than to design a minivan with a capacity of six /seven.



COMMON CASE FAST

1.1.4 Performance via parallelism

- Parallelism is a process of performing multiple jobs simultaneously.
- A processor engages in several activities in the execution of an instruction.
- Each instruction is executed at the same time to increase the performance.
- Larger problems are often subdivided into smaller units and are solved concurrently through parallelism.



PARALLELISM

1.1.5 Performance via pipelining

- Pipelining is an extension of the idea of parallelism.
- Pipeline is a set of jobs connected in series, where output of one element is the input of the next one.
- Here, the independent elements are executed in parallel to improve performance.
- Rather than processing each instruction sequentially, every instruction is split up into a sequence of steps so that different steps can be executed concurrently and in parallel, to improve performance.



PIPELINING

1.1.6 Performance via prediction

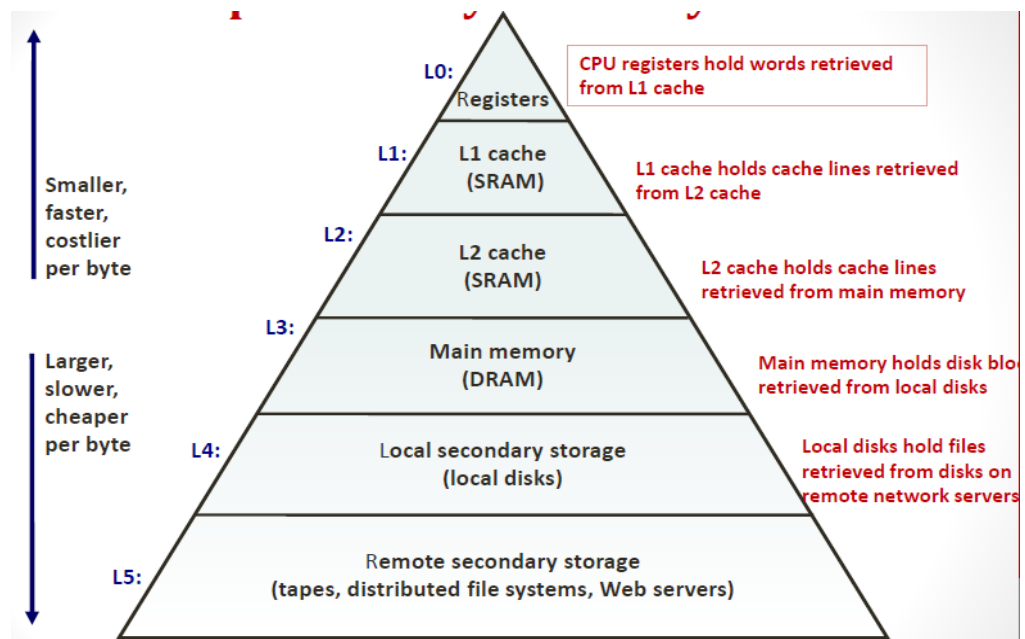
- Branch statements may cause unconditional wait, reducing the performance.

- This can be reduced by using branch predictor that guesses the path taken by a branch statement before it is actually known.
- The branch predictor improves the flow of execution in the instruction pipeline.
- It is performed by predicting the outcome of the condition test and then start executing the indicated instruction rather than waiting for correct answer.
- Performance is improved if the guesses are reasonably accurate and the penalty of wrong guesses is not too severe.



1.1.7 Hierarchy of memories

- Users need the memory to be very fast, large, and cheap.
- Computer has a range of memory units with cache and register memories being fast and small and secondary storage memories being slow and large.
- Cache memory is a small high-speed memory that holds recently accessed data.
- The memory hierarchy is given by,



1.1.8 Dependability via redundancy

- Computers need to be dependable since any device can fail.
- Hence several redundant modules (copies of data) can be maintained that helps the user to recover data when a failure occurs.
- One of the finest ideas in data storage is the RAID concept (Redundant Array of Inexpensive Disks).
- Data is stored redundantly on multiple disks that services us to recover them back.



DEPENDABILITY

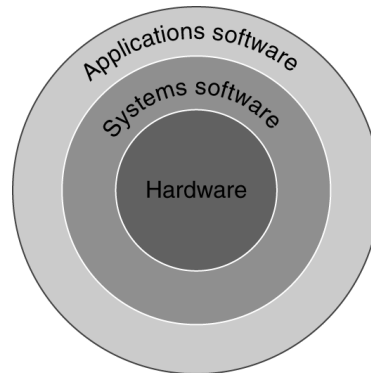
1.2 COMPONENTS OF A COMPUTER SYSTEM

Concept

- A computer is an information processing machine.
- It consists of a number of interrelated components that work together to convert data into information.
- The processing is carried out electronically, usually with no intervention from a human user.
- **Input unit** accepts the information from the user using input devices.
- The information received is either stored in the **computer's memory** for later reference or immediately used by the arithmetic and logic unit to perform the desired operations.
- The information is processed using the **instructions** (software) stored in the computer.
- The results are sent back to the user through the **output unit**.
- All the above actions are coordinated by the **control unit**.
- The list of instructions that performs a task is called a **program**, which is stored in the **memory**.

Components of a computer

1. Hardware component
2. Software component



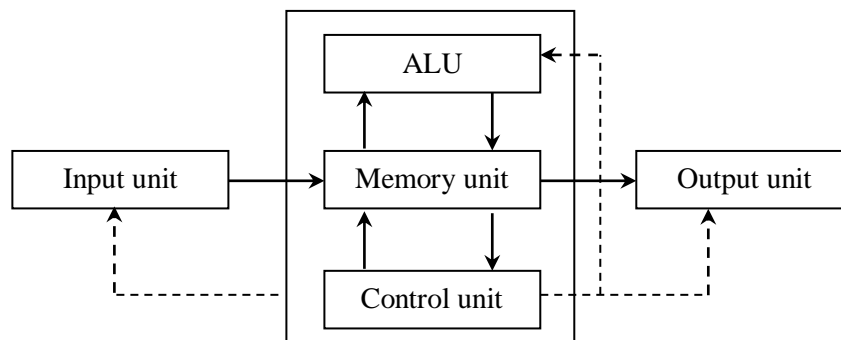
1.2.1 Hardware component

The electronic components interconnected in the computer system constitute the hardware components of a system.

A computer consists of the following functional units.

- Input Unit
- Central Processing Unit
 - Memory Unit – Primary memory, secondary memory, Cache memory, Registers
 - Arithmetic and logic Unit
 - Control Unit
- Output Unit

Functional units of a Computer



1.2.1.1 Input Unit

- They are electromechanical devices that allow the user to provide information into the computer for analysis and storage inside the CPU.
- Input device captures information and translates it into a form that can be processed by the CPU.
- Computer accepts input in two ways. They are,
 - Manual entry → the information is entered using keyboard or mouse.
 - Direct entry → the information is fed into the computer automatically from a source document like barcode.
- Examples for Input devices: Keyboard, pointing devices like Mouse, Joystick.
- Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code
- It is then transmitted over a cable to either the computer memory or the processor.

1.2.1.2 Central Processing Unit (CPU)

- It is referred as **‘the brain of a computer system’**.
- It converts data (input) into meaningful information (output).
- It is a highly complex, extensive set of electronic circuitry which executes stored program instructions called software.
- It controls all internal and external devices and performs arithmetic and logic operations
- It controls the usage of main memory to store data and instructions and controls the sequence of operations.
- It consists of three main subsystems.
 - **Arithmetic and Logic Unit (ALU)**
 - **Memory Unit**
 - **Control Unit (CU)**

1.2.1.3 Arithmetic Logic Unit (ALU)

It contains the electronic circuitry that executes all arithmetic and logical operations on the data. ALU comprises of two units

Arithmetic Unit

- Contains the circuitry that is responsible for performing the actual computing and carrying out the arithmetic calculations (+, -, *, /).
- To perform these operations, operands from the main memory is brought into processor
- After performing the operation results are stored in the memory location
- It can perform these operations at very high speed.

Logic Unit

- It enables the CPU to perform logical operations based on the instructions provided to it.
- Example: Logical comparison between data. (Logical Operations : =, <, > conditions)

1.2.1.4 Memory Unit

- Memory refers to the electronic holding place for instructions and data.
- Memory also stores the intermediate results and output.
- Memory is mainly classified into two categories: primary and secondary.

Primary Memory

- It is also known as main memory/ internal memory/ in-built memory.
- It stores data and instructions for processing.
- It is an integral component of CPU.
- It is a fast memory that operates at electronic speeds.
- The memory contains large no of semiconductor storage cells.
- Each cell carries 1 bit of information.
- The Cells are processed in a group of fixed size called Words.
- To provide easy access to any word in a memory, a distinct address is associated with each word location.
- Addresses are numbers that identify successive locations.
- The number of bits in each word is called the word length.
- The word length ranges from 16 to 64 bits.
- It can be classified as random access memory (RAM) and Read only memory (ROM).

Differences between RAM and ROM

Features	RAM	ROM
Stands for	Random Access Memory	Read-only memory
Volatility	RAM is volatile i.e. its contents are lost when the device is powered off.	It is non-volatile i.e. its contents are retained even when the device is powered off.
Types	The two main types of RAM are static RAM and dynamic RAM.	The types of ROM include PROM, EPROM and EEPROM.
Use	RAM allows the computer to read data quickly to run applications. It allows reading and writing.	ROM stores the program required to initially boot the computer. It only allows reading.
Definition	Random Access Memory or RAM is a form of data storage that can be accessed randomly at any time, in any order and from any physical location.	Read-only memory or ROM is also a form of data storage that cannot be easily altered or reprogrammed.

Secondary Memory

- It is also known as auxiliary memory or external memory.
- It is used for storing software programs and data.
- It is less expensive and stores huge volume of data than primary memory.
- The data and instructions stored on such devices are permanent in nature.
- It can be removed only if the user wants or if the device is destroyed.
- Example: Pen drive, Floppy Disk, Compact Disk, External hard disk, etc.

Differences between primary and secondary memory

Sl. No.	Primary Memory	Secondary Memory
1.	It is also called as Main or Internal or Built-in memory.	It is also known as Secondary or Auxiliary or External memory.
2.	It is present inside the computer.	It is present external to the computer that can be connected.

3.	It is smaller in size and holds limited data.	It is larger in size and holds massive amount of data.
4.	Due to its locality, it transfers data faster.	Since it is outside the CPU, it is slower when compared to primary memory.
5.	It is costlier than secondary memory.	It is cheaper than main memory.
6.	Example: RAM, ROM	Example: Disk drives, optical disks, magnetic tape drives.

Cache memory

- Cache is a high speed memory located in between RAM and the CPU.
- It increases the speed of processing since it holds the most frequently used data.
- It is highly expensive and smaller in size.
- It is present in two or three forms in a system – L1, L2 L3 cache memories.
- The memory ranges from 256 KB to 2 MB.

Register memory

- Registers are special-purpose, high speed temporary memory units
- It holds various types of information such as data, instructions, addresses and the intermediate result of calculations.
- It holds the information that the CPU is currently working on.
- It is said to be “**CPU’s working memory**” or an additional storage location that offers the advantage of speed.
- Registers are of two types: general purpose and special purpose registers.
 - **General purpose registers**
 - These are a set of registers that store temporary data and addresses by the programmer.
 - It includes floating point registers, constant registers, vector registers etc.
 - **Special purpose registers**
 - These are registers that are meant for performing special purposes by the CPU.

- They include Program counter (PC), accumulator (ACC), instruction register (IR), memory address register (MAR), memory buffer register (MBR), memory data register (MDR) etc.
- Program counter contains the address of the next instruction to be processed.
- Accumulator stores the result of various arithmetic and logical operations.
- Instruction register holds the current instruction that is fetched.
- MAR contains the address of the next location in the memory that is to be accessed.
- MBR stores the temporary data and MDR stores the operands of the expression processed.

1.2.1.5 Control Unit

- This unit checks the correctness of sequence of operations.
- It fetches the instructions from the primary memory, interprets them and ensures correct execution of the program.
- It also controls the input and output devices
- Directs the overall functioning of the other units of the computer.
- Control Unit
 - Supervises and controls the path of information that runs over the processor.
 - Organizes the various activities of those units that lie inside it.
 - Guides the flow of data through the different parts of the computer.
 - Interprets the instructions.
 - Regulates the time controls of the processor.
 - Sends and receives control signals from various peripheral devices.

1.2.1.6 Output Devices

- Output Devices take the machine-coded output results from the CPU and convert them into a form that is easily readable by human beings.

- The output can be obtained in two forms: hardcopy and softcopy.
 - The physical form of the output is called as hardcopy.
 - The output which resides in the memory is called as softcopy.
- Example: Monitors, Printers, Plotters and audio response etc.

1.2.2 Software Component

- Software is the collection of instructions written in a computer language.
- It is responsible for controlling, integrating and managing the hardware components of a computer and to accomplish a specific task.
- Software instructs the hardware to perform the desirable task to be done.

Types

- System Software
- Application Software

1.2.2.1 System Software

- System software is a program that manages and supports the computer resources and operations of a computer system.
- It executes various tasks such as processing data and information, controlling hardware components, and allowing users to use application software.
 - It is more transparent and less noticed by the users.
 - They usually interact with the hardware or the applications.
 - Basic functionality includes file management, visual display, keyboard input, etc.
- Example: Operating systems, device drivers, language translators, text editors, utilities, loaders, linkers, etc.

Operating System

- An **operating system (OS)** is a collection of software that manages computer hardware resources and provides common services for computer programs.
- The operating system is a vital component of the system software in a computer system. Application programs require an operating system to function.

1.13 Computer Architecture

- The functions of OS include Disk Access, Memory Management, Task Scheduling, and User Interfacing.
- Provides a software platform on top of which other programs run.
- Example: MS –DOS, WINDOWS, LINUX, UNIX.

Device Drivers

- These are system programs which are responsible for proper functioning of devices.
- Whenever a new device is added to a computer system the driver must be installed before the device is used.
- It acts as a translator between the device and the program that uses the device.
- It is not an independent program; it assists or is assisted by the OS for proper functioning.
- Example: printer, monitor, mouse, keyboard.

Programming Language Translators

- The language translators transform the instructions prepared by programmers in a high level language into the form which can be understood by the computer.
- Translators are divided into 3 categories: compiler, interpreter, and assembler.
- **Compiler**
 - Compiler translates the high level programming language into machine language.
 - It translates source code into object code.
 - It can be used for larger applications.
 - Example: C, C++, PASCAL compilers.
- **Interpreter**
 - Interpreters translate the source code into object code in line-by-line manner, without looking at the entire program.
 - Programs produced by compilers run much faster than interpreter.
 - It is easier to modify the source code.
 - Example: Basic.

- **Assembler**
 - Assemblers translate assembly language program into machine language.

System Utility

- These programs perform day-to-day tasks related to the maintenance of the computer system.
- They are used to support, enhance and secure existing programs and data in the computer system.
- They are generally small programs having specific task to perform.

1.2.2.2 Application Software

- This is the most used software by the users.
- It is used to accomplish specific tasks.
- Application software consists of a single program (Ex. Notepad) or a collection of programs (software package) (Ex. Microsoft Office Suite).

Some of the most commonly used application software are as follows.

Word Processors

- A word processor is software used to compose, format, edit and print electronic documents.
- We can include pictures, graphs, and charts and allows changes in alignments, margins, font, and color and also allows spell checking.
- Example: Microsoft word, word perfect

Spreadsheets

- A spreadsheet application is a rectangular grid, which allows text, numbers and complex functions to be entered into a matrix of thousands of individual cells.
- Applications include payroll processing, financial record maintenance.
- Example: Microsoft Excel, Lotus 1-2-3.

Image Editors

- Image editor programs are designed specifically for capturing, creating, editing and manipulating images.

1.15 Computer Architecture

- The programs provide a variety of special features for creating and altering images.
- They also enable the user to create and superimpose layers, import and export graphic files, adjust an image and improve its appearance
- Example: Adobe Photoshop, Adobe Corel Draw.

Database Management Systems

- Database Management software is a collection of computer programs that supports structuring of the database in a standard format
- It provides tools for data input, verification, storage, retrieval, query and manipulation in an efficient manner.
- It controls the security and integrity of the database from unauthorized access.
- Example: Oracle, FoxPro.

Presentation Applications

- A Presentation is a means of assessment, which requires presentation providers to present their work orally in the presence of an audience.
- It combines both visual and verbal elements.
- Presentation software allows the user to create presentations by producing slides/hand-outs for the presentation of projects.
- Example: Microsoft PowerPoint.

Desktop Publishing Software

- The desktop publishing is a technique of using a personal computer to design images and pages, and assemble type and graphics, then using a printer to output the assembled pages onto paper.
- This software is used for creating magazines, books etc.
- Example: Adobe PageMaker, Quark Express.

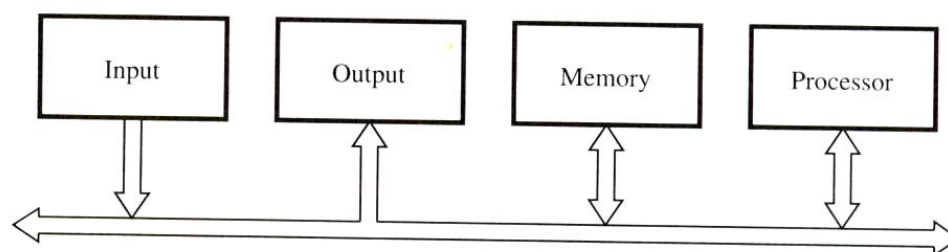
1.3 BUS STRUCTURE

- A group of lines that serves as the connecting path from one device to another is called a Bus.
- A Bus may be defined as a set of communication lines/ data paths that carry the data, address or control signal among various units of a CPU.

- A memory/ system bus interconnects the processor with memory units and I/O units.
- When a word of data is transferred between units, all bits are transferred in parallel.
- The bits are transferred simultaneously over many wires, or lines, with one bit per line.
- There are 2 types of Bus structures. They are,
 - Single Bus Structure
 - Multiple Bus Structure

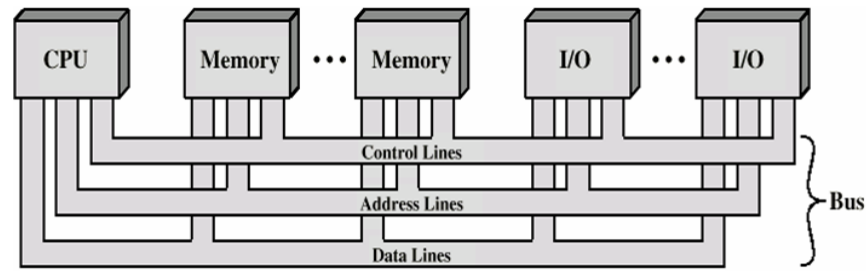
Single Bus Structure

- All the components (I/O, memory and processor) are connected to a common bus.
- It allows only one transfer at a time because only two units can actively use the bus at any given time.
- Advantage of using single bus structure is
 - Low cost
 - High flexibility in attaching peripheral devices.
- The only problem is that the performance is low due to slow data transfer.



Multiple Bus Structure

- To achieve high performance, multiple bus structures are used.
- This allows two or more transfers to be carried out at the same time, concurrently.
- This provides a better performance but at an increased cost.



- The devices connected to a bus vary widely in their speed of operation.
- Hence buffer registers are used to hold information during transfers.
- Thus, the buffer register prevents a high speed processor from being locked to a slow I/O device during data transfer.
- This allows the processor to switch rapidly from one device to another.

1.4 TECHNOLOGIES FOR BUILDING PROCESSORS AND MEMORY

- A computer architect must plan the design based on the future technology changes
- The designer must be aware of rapid changes in implementation technology.
- There are four implementation technologies to be considered. They are,
 - Integrated Circuit logic technology
 - Semiconductor DRAM
 - Magnetic disk technology
 - Network technology

Integrated circuit logic technology

- A transistor is a small electronic device made of semiconductor material that carries current and amplify.
- It was used in II generation computers and was very slow.
- Due to its low capacity, Integrated circuits(IC) were introduced.
- The IC technology emerged with the fabrication of transistors on a single chip.
- The Small Scale Integration (SSI) technology involved few transistors (< 100) on a silicon chip.
- Then emerged the Medium Scale Integration (MSI) composing hundreds of transistors on a chip.

- The III generation computers relied on SSI and MSI technology.
- The IV generation computers used Large Scale Integration (LSI) and Very Large Scale Integration (VLSI) technology that was composed of thousands of transistors on a single chip.
- The V generation computers being dependent on the Ultra Large Scale Integration (ULSI) technology that possesses several millions of transistors on a chip.
- The transistor density increases by about 35% per year.

Year	Technology used in computers	Relative performance/unit cost
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit	900
1995	Very large-scale integrated circuit	2,400,000
2005	Ultra large-scale integrated circuit	6,200,000,000

Semiconductor DRAM (dynamic random-access memory)

- DRAM is a semiconductor memory device that stores each bit of data in a separate passive electronic component like capacitor.
- Capacity increases by about 40% per year every two years.

Magnetic disk technology

- Magnetic storage is the storage of data on a magnetized medium.
- The density increased by about 30% per year before 1990, and increased to 100% per year in 1996.
- It has again dropped back to 30% per year since 2004.
- But still, disks are still 50 –100 times cheaper per bit than DRAM.

Network technology

- In order to provide communication from one computer to another, LAN (Local Area Network) like Ethernet, MAN (Metropolitan Area Network), WAN

(Wide Area Network) like internet and Wireless Network like Wi-Fi, Bluetooth were introduced.

- Networking technologies allow data sharing, communication
- Although technology improves continuously, the impact of these improvements can be in distinct leaps.

1.5 PERFORMANCE

- The prime factors of the success of a computer are the speed and cost.
- Performance depends on how fast machine instructions can be brought into the processor for execution and how fast they can be executed.
- The performance of a computer is dependent on
 - the design of the compiler
 - the machine instruction set
 - the hardware

Terminologies

Response time

- It is the time between the start and completion of a task.
- It refers the execution time of a set of instructions.
- The faster execution of instructions leads to reduction in response time, thus increases throughput.

Throughput

- It is the total work done in a unit time period.

Elapsed Time

- The total time required to execute the program is called the elapsed time.
- It depends on all the units in computer system.

Processor Time

- The period in which the processor is active is called the processor time.
- It depends on hardware involved in the execution of the machine instruction.

Clock

- The Processor circuits are controlled by a timing signal called a clock.

Clock Cycle

- The clock defines a regular time interval called clock cycle.

Bandwidth

- The amount of data that can be transferred from one point to another in a given time period is called bandwidth.
- It is expressed in bits per second (bps).

Clock cycles per instruction (CPI)

- Average number of clock cycles per instruction for a program or program fragment.

CPU clock cycles = Instructions for a program \times Average clock cycles per instruction

Execution of an Instruction

- At the start of execution all program instructions and the required data are stored in the main memory.
- As execution proceeds, instructions are fetched one by one over the bus into the processor, and a copy is placed in the cache.
- When the execution of an instruction calls for data located in the main memory, the data are fetched and a copy is placed in the cache.
- A Program will be executed faster if the movement of instruction and data between the main memory and the processor is minimized, which is achieved by using the Cache.
- To execute a machine instruction, the processor divides the action to be performed into a sequence of basic steps; each step can be completed in one clock cycle.

Clock Rate, $R = 1/P$ (measured in cycles per second)

where, $P \rightarrow$ Length of one clock cycle

Basic Performance Equation

Let

$T \rightarrow$ the processor time required to execute a program.

$N \rightarrow$ the actual number of instruction executions

$S \rightarrow$ Average number of basic steps needed to execute one machine instruction

If the **clock rate** is **R** cycles/second, the program execution time is given by

$$T = (N \cdot S) / R$$

where, $T \rightarrow$ Performance Parameter

$R \rightarrow$ Clock Rate in cycles/sec

$N \rightarrow$ Actual number of instruction executions

$S \rightarrow$ Average number of basic steps needed to execute one machine instruction

To achieve high performance, the computer designer must reduce the value of T , which means reducing N and S , increasing R .

$$N, S < R$$

The value of N is reduced if the source program is compiled into fewer machine instructions. The value of S is reduced if instructions have a smaller number of basic steps to perform or if the execution of instructions is overlapped.

Pipelining

A considerable improvement in performance can be achieved by overlapping the execution of successive instructions. This technique is called pipelining.

Superscalar Operation

Multiple instruction pipelines can be implemented in the processor that allows several instructions can be executed in parallel by creating parallel paths. This mode of operation is called the Superscalar execution.

Clock Rate

There are 2 possibilities to increase the clock rate(R). They are,

- Improving the integrated Chip(IC) technology makes logic circuits faster, which reduces the time needed to complete a basic step. This allows the clock period, P , to be reduced and the clock rate, R , to be increased.
- Reducing the amount of processing done in one basic step also helps to reduce the clock period P .

Performance Improvement

- To maximize the performance, minimize the response time or execution time for some task.

- The performance of the computer is directly related to performance and execution time for computer, X.

$$Performance_x = \frac{1}{ExecutionTime_x}$$

- For two computers X and Y, the performance of X is greater than Y then we have

$$Performance_x > Performance_y$$

$$\frac{1}{ExecutionTime_x} > \frac{1}{ExecutionTime_y}$$

$$ExecutionTime_y > ExecutionTime_x$$

Example

Time taken to run a program = 10s on A, 15s on B

Relative performance = Execution Time_B / Execution Time_A

$$= 15s / 10s$$

$$= 1.5$$

So A is 1.5 times faster than B

Measuring Performance

- Measured in terms of seconds per program
- Defined as the total time taken to complete a task. The task includes
 - disk access
 - memory access
 - I/O activities
 - Execution of Instructions
- This time taken is known as wall-clock time / response time.

CPU Time

- Time the CPU spends computing for particular task and does not include the time waiting for I/O.
- This is also called as CPU execution time

Formula

$$CPU\ time = \frac{CPU\ time\ spent\ in\ the\ program(\textbf{user CPU time})}{CPU\ time\ spent\ in\ the\ operating\ system(\textbf{system CPU time})}$$

Example

Let

- User CPU time = 90.7 seconds
- System CPU time = 12.9 seconds
- Elapsed time = 2 minutes and 39 seconds (159 seconds)
- CPU time

$$CPU\ Time = \frac{90.7 + 12.9}{159} = 0.65$$

Performance Equation I

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

Clock rate is inverse of clock cycle time

$$\text{Clock Rate} = \frac{1}{\text{Clock cycle time}}$$

- CPU Execution time

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

- Performance is improved by reducing the length of the clock cycle or number of clock cycle required for a program
- Execution time depends on the number of instructions in a program
- Number of clock cycles

$$CPU\ clock\ cycles = Instructions\ for\ a\ program \times Average\ clock\ cycles\ per\ instruction$$

- Clock cycles per instruction (CPI)
 - Average number of clock cycles in which each instruction takes to execute

Performance Equation II

- CPU Execution Time in terms of instruction count, CPI and clock cycle time

CPU Execution time = Instruction count \times CPI \times Clock cycle time

(OR)

$$\text{CPU Execution Time} = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

1.6 POWER WALL

- Processors run at high clock speed and it generates more heat and consumes high power to improve performance.
- If clock rate increases, power consumption also gets increased.

Power Issues

- Power has to be provided to the processor and it has to be distributed around the chip.
- Power consumed by a device dissipates it in terms of heat and it must be removed.

Power Consumption

- In CMOS chips, dynamic power refers to the dominant energy consumption in switching transistors.
- The power required per transistor is proportional to the product of the load capacitance of the transistor, the square of the voltage, and the frequency of switching.
- Power consumed by CPU is given by

$$P_{\text{dynamic}} = CV^2F$$

where,

- P is Power
- C is capacitive loading
- V is voltage applied
- F is frequency running

- Mobile devices care about battery life more than power, so energy is the proper metric to be considered.

- Energy is measured in joules.

$$\text{Energy}_{\text{dynamic}} = \text{Capacitive load} \times \text{Voltage}^2$$

- In CMOS, static power is becoming an important issue because leakage current flows even when a transistor is off.
- Thus, increasing the number of transistors increases power even if they are turned off, and leakage current increases in processors with smaller transistor sizes.
- As a result, very low power systems are even gating the voltage to inactive modules to control loss due to leakage.

$$\text{Power}_{\text{static}} = \text{Current}_{\text{static}} \times \text{Voltage}$$

1.7 UNIPROCESSORS TO MULTIPROCESSORS

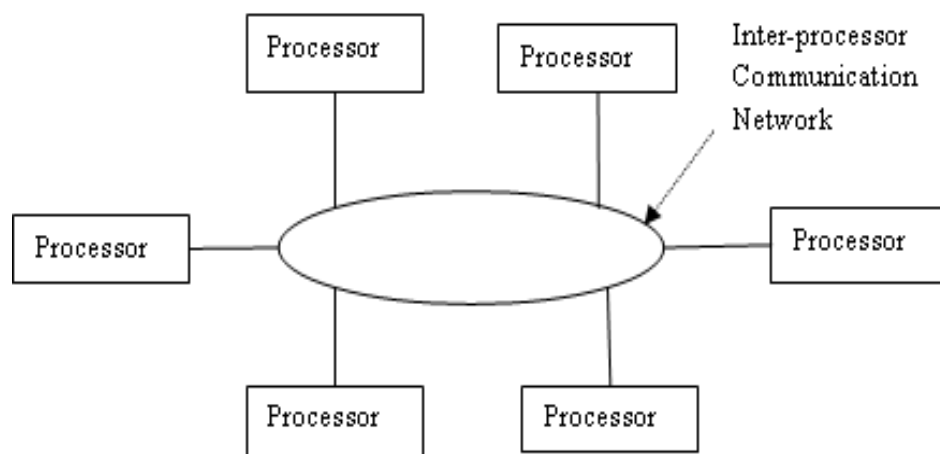
- Because of limitation forced by power consumption there is a change in the design of microprocessor.
- Rather than continuing to decrease the response time of a single program running on the single processor, designer came up with multiple processors per chip.
- Intention is to reduce the throughput rather than decreasing the response time.
- To reduce the confusion between the words processor and microprocessor, processors are referred as core such microprocessors are known as "multicore microprocessor".
- Example
 - Dual core microprocessor is a chip that contains two processors or cores.
 - Quad core microprocessor is a chip that contains four processors or cores.
- In previous days, programmers rely on hardware, architecture and compiler to double their performance of the program.
- Now-a-days, programmers rewrite their programs to support parallelism.

1.7.1 Multi-Processor

- Computers that contain several processor units are called multiprocessor system.
- These systems either execute a number of different application tasks in parallel or execute subtasks of a single large task in parallel.
- All processors have access to all memory locations and they are called shared memory multiprocessor systems.
- The high performance of these systems comes with much increased complexity and cost.

1.7.2 Advantage of Multiprocessor System

- Improves cost or performance ratio of the system.
- Tasks are divided among several modules/processors.
- If failure occurs, it is cheaper and easier to find and replace the malfunctioning processor.
- If fault occurs in one processor, the others processor can take the responsibility of performing the task of failure processor.



1.8 INSTRUCTIONS

- An instruction is a piece of a program that performs an operation issued by the computer processor.
- Every instruction is defined by the instruction set of the processor.

1.8.1 Instruction Set

- A list of all the instructions with all their variants that can be executed by a processor is called instruction set.
- It is a group of commands defined by the processor in machine understandable language.

Example

- Arithmetic instructions → Add, Subtract, Multiply and Divide
- Logic instructions → And (Conjunction) , Or (Disjunction), Not (negation)
- Control flow instructions → Goto, if ... goto, call, and return
- Data handling and memory instructions → Read, Write, Copy, Set, Load, Store

Instruction Sets are differentiated based on

- Operand storage in the CPU (data can be stored in a stack structure or in registers)
- Number of explicit operands per instruction (zero, one, two, and three address)
- Operand location (instructions can be classified as register-to-register, register-to-memory or memory-to-memory)
- Operations
- Type and size of operands (operands can be addresses, numbers, or even characters)

Format

An instruction has three fields, namely-

- **Operation code** (Opcode) specifies which type of operation to be performed.
- **Mode Field** specifies the way the operand or effective address is determined.
- **Address Field** specifies memory address or a processor register.

Opcode	Mode field	Address field
--------	------------	---------------

1.8.2 Types of Instruction format

0 – Operand instruction

- These instructions have no address fields.
- They are also called as Zero address instruction or Stack instruction.
- They do not have source / destination addresses. The address is implicit.

- All the operations are done using stack data structure.
- The operands are present on the top of the stack.
- In other words, the absolute address of the operand is held in a special register that is automatically incremented (or decremented) to point to the location of the top of the stack.
- Syntax

Stack_Operation / Operation

- Example

To perform $C = A + B$, the instructions are,

PUSH A	//Inserts the data A onto the stack
PUSH B	//Inserts the data B onto the stack
ADD	//Adds the value of A and B
POP C	//Gets the added value from the stack

- **Advantages**
 - It is a simple model of expression evaluation.
 - The instructions are short.
- **Disadvantages**
 - A stack can't be randomly accessed.
 - This makes it hard to generate effective code.
 - Since the same stack is used for every operation, it creates a bottleneck.

1 – Operand instruction

- These instructions contain one address field.
- They are also known as one address instruction or Accumulator instruction.
- Accumulator (ACC) register is used for manipulation of data.
 - All the operations are carried out between the accumulator register and a memory operand.
- Syntax

Operation Destination_Location

- **Example**

ADD A is equivalent to $ACC \leftarrow ACC + A$

Where, $A \rightarrow$ Destination operand

The Arithmetic Operation, $C = A + B$ is performed as,

LOAD A

ADD B

STORE C

- **Advantage**

- The instructions are short.

- **Disadvantage**

- The accumulator is only a temporary storage so memory traffic is the highest for this approach also.

2 - Operand Instructions

- These instructions contain two address fields namely, the source and the destination.
- Each address field specifies either a processor register or a memory.
- They are also called as two – Address instructions or general purpose register instructions.
- Syntax

Operation Destination_Location, Source_Location

- **Example**

To perform $C = A + B$, an intermediate register R1 is used as,

LOAD R1, A ADD R1, B STORE C, R1	or	LOAD R1, A LOAD R2, B ADD R1, R2 STORE C, R1
--	----	---

- **Advantage**
 - Makes code generation easy.
- **Disadvantage**
 - All operands must be named leading to longer instructions.

3 - Operand instruction

- These instructions contain three address fields.
- They are also called as three address instructions or general purpose register instruction.
- Register address field may be a processor register or a memory operand.
- **Syntax**

Operation Source1_Location, Source2_location,
Destination_Location

- **Example:** To perform $C = A + B$, the code is

ADD A, B, C	or	MOVE R1, A ADD C, R1, B	or	LOAD R1, A LOAD R2, B ADD R3, R1, R2 STORE C, R3
-------------	----	----------------------------	----	---

- **Advantage**
 - Makes code generation easy.
- **Disadvantage**
 - All operands must be named leading to longer instructions.

1.8.3 Instruction Execution

The four phases in instruction execution are

- Fetch the Instruction from memory - the instruction is fetched from the memory location whose address is in Program Counter (PC) and is placed in the instruction register.

1.31 Computer Architecture

- Decode the Instruction.
- Execute the Instruction - the operands are fetched from the memory or processor registers, and the operation is performed.
- Store the result in the destination location.

1.8.4 MIPS Instruction Formats

- **R - Format**

Field	opcode	rs	rt	rd	shamt	funct
Bit Positions	31-26	25-21	20-16	15-11	10-6	5-0

- Opcode = 0
- Three register operands: rs, rt, and rd
 - rs and rt - sources
 - rd - destination
- shamt field - used only for shifts
- funct field - The ALU function (add, sub, and, or, and slt) and is decoded by the ALU control design

ALU control lines	Function
000	AND
001	OR
010	Add
110	Subtract
111	Set on less than

- **I - Format**

Field	opcode	rs	rt	address
Bit Positions	31-26	25-21	20-16	15-0

- *For load and store instructions*
 - Opcode = 35(for load) and Opcode = 43(for store)
 - rs - the base register

- *rt* is
 - For loads, the destination register for the loaded value
 - For stores, the source register whose value should be stored into memory
- The memory address is computed as

Memory address = base register + 16-bit address field

○ *For branch instructions*

- Opcode = 4
- *rs* and *rt* are the source registers that are compared for equality
- The branch target address is computed as

Target address = PC + (signed-extended 16-bit offset address << 2)

• **J – Format**

Field	opcode	address
Bit Positions	31-26	25-0

- Opcode = 2
- The destination address is computed as

Target address = PC [31-28] || (offset address << 2)

1.9 LOGICAL INSTRUCTIONS

- Instructions that perform logical operations and manipulate Boolean values are called as logical instructions.
- They include Logical AND, OR, and NOT.

1.9.1 AND instruction

- It contains three register operands.
- These instructions perform bitwise AND operation between the source registers and stores the result in the destination register.
- It is also called as conjunction operation.

1.33 Computer Architecture

- Syntax

Operation destination, source1, source2

- Example

AND R3, R1, R2 //Equivalent to $R3 = R1 \& R2$

1.9.2 OR instruction

- OR instruction contains three register operands.
- It performs bitwise OR operation between the source registers and stores the result in the destination register.
- This is also called as disjunction operation.

- Syntax

Operation destination, source1, source2

- Example

OR R3, R1, R2 //Equivalent to $R3 = R1 | R2$

1.9.3 NOR instruction

- These instructions have three register operands.
- It performs bitwise NOR operation (OR operation followed by NOT) between two source registers and stores the result in the destination register.

- Syntax

Operation destination, source1, source2

- Example

NOR R1, R2, R3 // Equivalent to $R1 = \sim (R2 | R3)$

1.9.4 AND Immediate (ANDI) instruction

- This instruction contains three register operands.
- It performs bitwise AND operation between a source register and specified immediate value and stores the result in the destination register.

- Syntax

Operation destination, source1, Immediate_Value

- Example

AND R1, R2, Immediate_Value // Equivalent to $R1 = R2 \& \text{Imm_Val}$

1.9.5 OR Immediate (ORI) instruction

- This instruction has three register operands.
- It perform bitwise OR operation between a source registers and specified immediate value and finally stores the result in the destination register.
- Syntax
Operation destination, source1, Immediate_Value

- Example

OR \$1, \$2, immediate_Value //Equivalent to $R1 = R2 \mid Imm_Val$

1.9.6 Shift Left Logical instruction

- This instruction contains three register operands.
- It shifts the given register value left by the shift amount listed in the instruction and stores the result in a third register.
- Syntax
Operation destination, source1, constant

- Example

SLL R1, R2, 10 // Equivalent to $R1 = R2 \ll 10$

1.9.7 Shift Right Logical instruction

- This instruction has three register operands.
- It shifts the specified register value right by the shift amount listed in the instruction and stores the result in a third register.
- Syntax
Operation destination, source1, Constant

- Example

SRL R1, R2, 10 // Equivalent to $R1 = R2 \gg 10$

1.9.8 Shift Right Arithmetic instruction

- This instruction possesses three register operands.
- It shifts a register value right by the shift amount listed in the instruction and places the result in a third register.

- SRA R1, R2, 10 // Equivalent to $R1 = R2 \gg 10$

- Control statements are those that take a decision out of or without a condition.
- These instructions perform a test by evaluating a logical condition.
- Depending on the outcome of the condition, it modifies the PC to take the branch or continue to the next instruction.
- Control operations are of two types.
 - Conditional branch → Performs branching based on a condition
 - Unconditional branch → Performs branching without any condition

An instruction that directs the computer to another part of the program based on the results of a comparison is called conditional branching.

- BEQ stands for branch on equal.
- The instruction checks if the two register values are equal. If so, it branches to the specified offset.
- Syntax
Operation source1, source2, offset
- Example

```
BEQ R1, R2, OFFSET           // Equivalent to
                               if (R1==R2) goto L1;
                               a=b+c;
                               L1: a=b-c;
```

1.10.1.2 BNE Instruction

- BNE stands for branch on not equal.
- The instruction performs branching if the two registers values are not equal.
- Syntax
Operation source1, source2, offset

- Example

BNE R1, R2, OFFSET

// Equivalent to

if (R1 != R2) goto L1;

a=b+c;

L1: a=b-c;

1.10.2 Unconditional Branch

An instruction that directs the computer to another part of the program without any test/condition operation is called unconditional branching.

1.10.2.1 J Instruction

- J stands for jump.
- It Jumps to the specified address.
- Syntax
Operation offset
- Example

J Target_Address;

1.10.2.2 JAL Instruction

- JAL stands for Jump and link.
- The instruction jumps to the specified address and stores the return address.
- Syntax
Operation offset
- Example
JAL Target_Address;

1.10.2.3 JR Instruction

- JR stands for Jump Register.
- It jumps to the address contained in the specified register R.
- Syntax

Operation source

- Example

JAL R1;

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

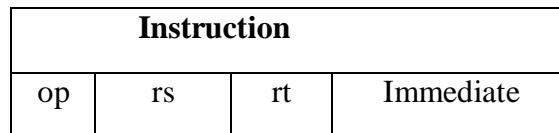
FIGURE 2.20 MIPS instruction formats.

1.11 MIPS ADDRESSING AND ADDRESSING MODES

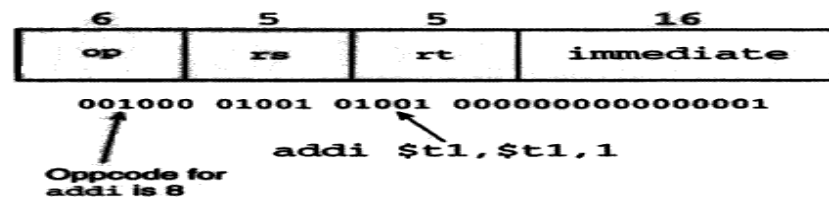
- There are various ways to specify the address of the operands for any given operations such as load, add or branch. The different ways of determining the address of the operands specified in an instruction are called addressing modes. In other words, addressing mode specifies how address is determined which may be either memory or register.
- MIPS addressing modes are as follows:
 - **Immediate addressing:** The operand is a constant within the instruction itself
 - **Register addressing:** The operand is a register
 - **Base or displacement addressing:** The operand is at the memory location whose address is the sum of a register and a constant in the instruction
 - **PC-relative addressing:** The branch address is the sum of the PC and a constant in the instruction
 - **Pseudo-direct addressing:** The jump address is the 26 bits of the instruction concatenated with the upper bits of the PC

1.11.1 Immediate Addressing

- The instruction contains an immediate operand that has a constant value or an expression.



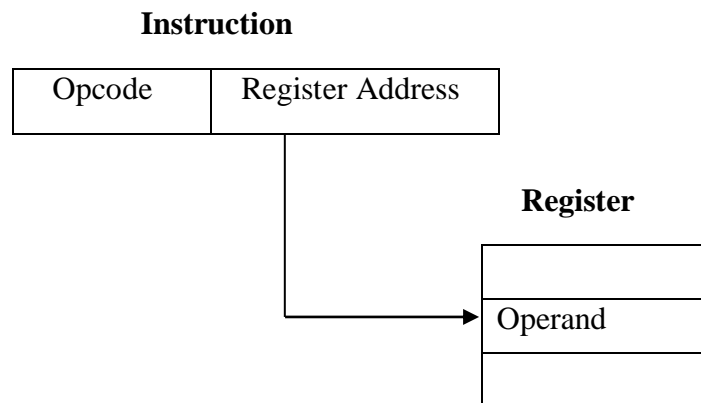
- The operand is embedded inside the instruction
- Since the instruction does not require an extra memory access to fetch the operand, it executes faster.
- Example:



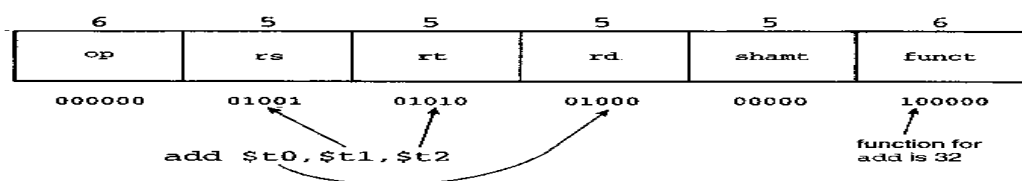
1.11.2 Register Addressing

- It is the simplest addressing modes of all. The instruction works on register operands. It works much faster than other addressing modes because it does not involve with memory access.
- The register address is specified as a part of the instruction.

Effective Address, EA = Register Address, A

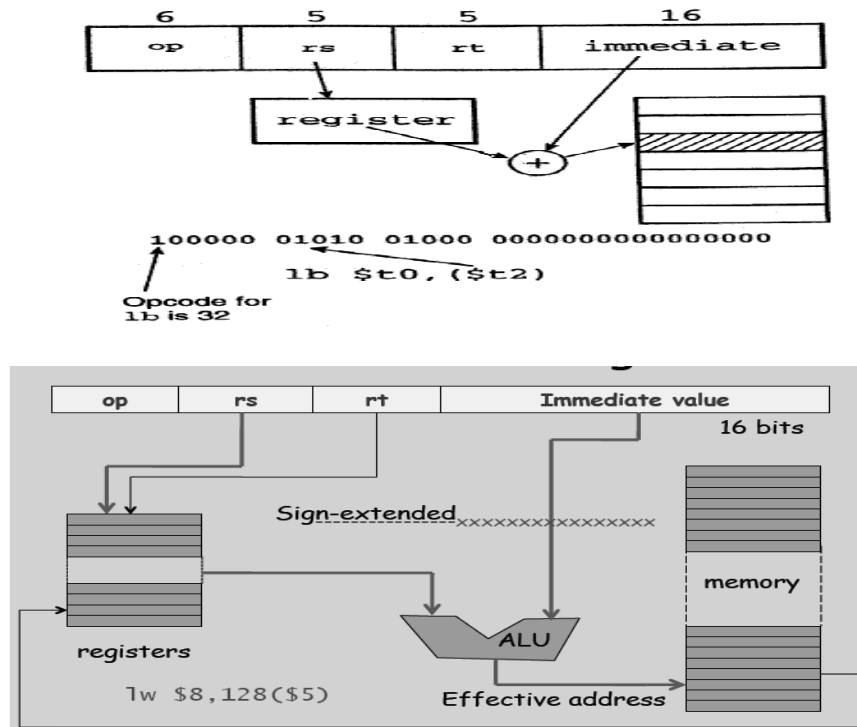


- Example



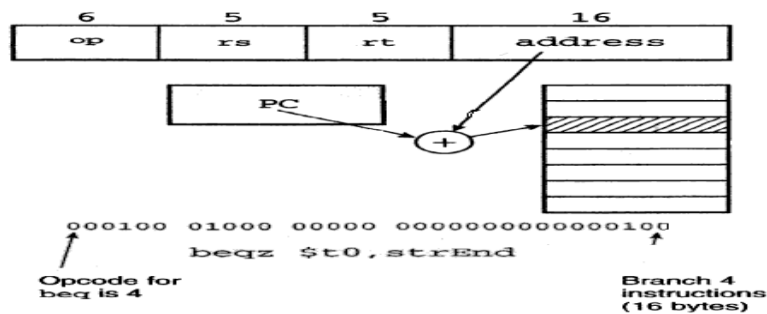
1.11.3 Base Addressing

- The address of the operand is the sum of the immediate and the value in a register (rs). 16-bit immediate is a two's complement number
- Effective Address = $A + (R)$
- Examples:



1.11.4 PC Relative Addressing

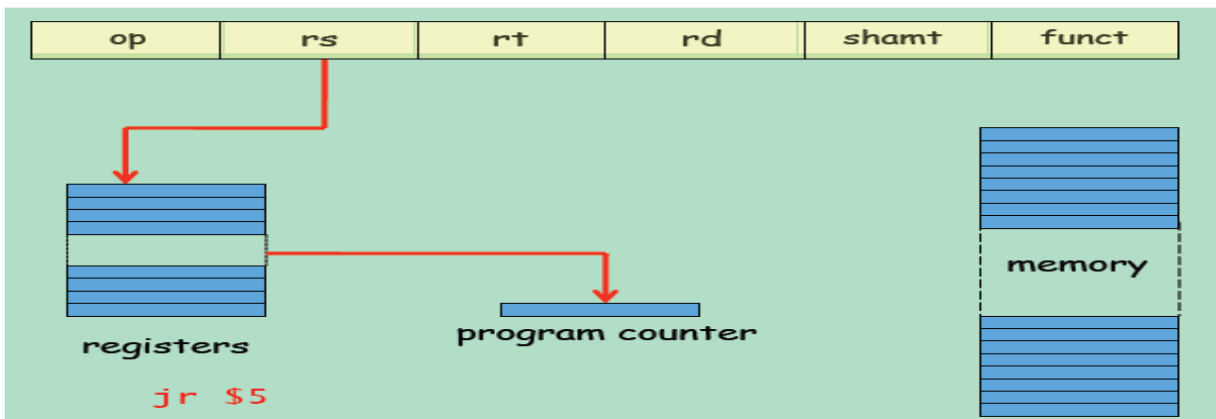
- For relative addressing, the implicitly referenced register is the program counter (PC).
- PC-relative addressing is used for conditional branches. The address is the sum of the program counter and a constant in the instruction. Example



1.11.5 Pseudo Direct Addressing

In Register Direct Addressing, the value the (memory) effective address is in a register. It is also called “Indirect Addressing”. Special case of base addressing where offset is 0. Used with the jump register instructions.

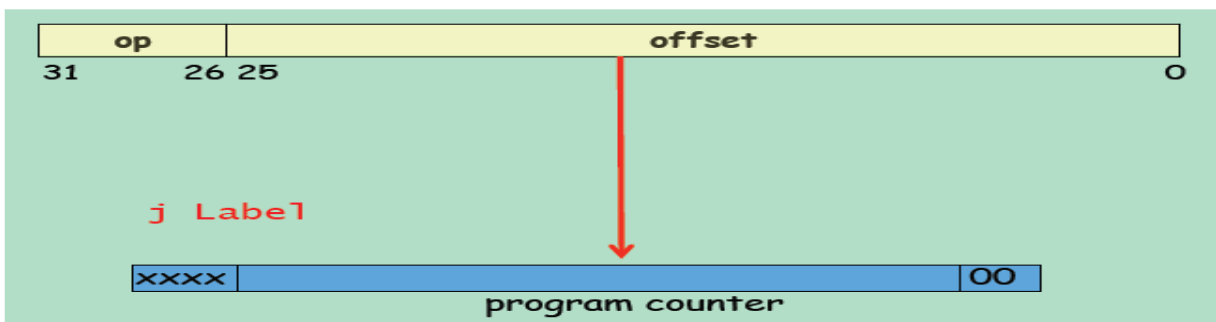
Example: jr \$31



Direct Addressing: the address is “the immediate”. 32-bit address cannot be embedded in a 32-bit instruction.

Pseudodirect addressing: 26 bits of the address is embedded as the immediate, and is used as the instruction offset within the current 256MB (64MWord) region defined by the MS 4 bits of the PC.

Example: j Label



UNIT

ARITHMETIC OPERATIONS

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

2.1 ALU

- An **arithmetic logic unit (ALU)** is a digital circuit that performs arithmetic and logical operations.
- The ALU is a fundamental building block of the central processing unit (CPU)/processor of a computer.
- The processors are composed of very powerful and very complex ALUs.
- The ALU was proposed by the famous mathematician, John von Neumann in 1945.
- The ALU works on two types of numbers
 1. Fixed point numbers
 2. Floating point numbers
- The basic operations are implemented in hardware level. ALU is having collection of two types of operations, namely -
 1. Arithmetic operations
 2. Logical operations

Consider an ALU having 4 arithmetic operations and 4 logical operations.

- To identify any one of these four logical operations or four arithmetic operations, two control lines are needed.
- Also to identify the any one of these two groups- arithmetic or logical, another control line is needed.
- So, with the help of three control lines, any one of these eight operations can be identified.
- Arithmetic operations include addition, subtraction, multiplication and division.

- The four logical operations include OR, AND, NOT & EX-OR.
- We need three control lines to identify any one of these operations.
- The input combination of these control lines are shown below.
- Control line C_2 is used to identify the group: logical or arithmetic,
 - $C_2=0$: arithmetic operation
 - $C_2=1$: logical operation.
- Control lines C_0 and C_1 are used to identify any one of the four operations in a group. One possible combination is given here.

C_1	C_0	Arithmetic $C_2 = 0$	Logical $C_2 = 1$
0	0	Addition	OR
0	1	Subtraction	AND
1	0	Multiplication	NOT
1	1	Division	EX-OR

- A 3 x 8 decoder is used to decode the instruction.
- The block diagram of the ALU is shown.

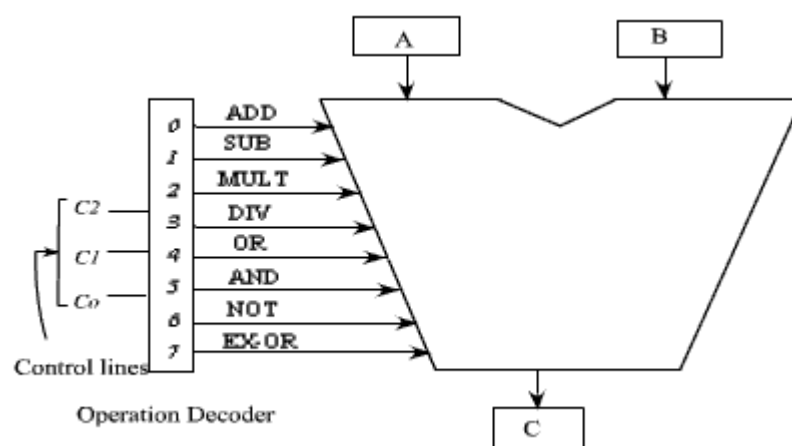


Figure 2.1: Block Diagram of the ALU

1.43 Computer Architecture

- The ALU has got two input registers named as A and B and one output storage register, named as C.
- It performs the operation as: $C = A \text{ operator } B$
- The input data are stored in A and B, and according to the operation specified in the control lines, the ALU perform the operation and put the result in register C.

Example

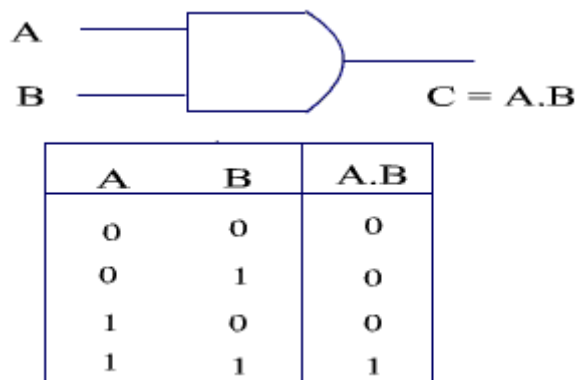
- If the contents of controls lines are, 000, then the decoder enables the addition operation and it activates the adder circuit and the addition operation is performed on the data that are available in storage register A and B.
- After the completion of the operation, the result is stored in register C.

2.1.1 Logic Gates

- There are several logic gates exists in digital logic circuit.
- These logic gates can be used to implement the logical operation.
- Some of the common logic gates are AND, OR, EX-OR etc.

AND gate

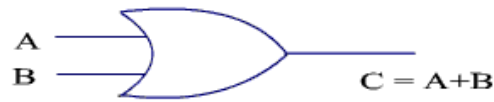
- The AND gate produces output is high (1) if both the inputs are high (1).
- The AND gate and its truth table are as shown.



AND gate and its truth table.

OR gate

- The output of the OR gate is high if any one of the input is high.
- The OR gate and its truth table are as shown.



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

OR gate and its truth table

EX-OR gate

- The output of the EX-OR gate is high if either of the input is high.
- The EX-OR gate and its truth table is given as follows.



A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

EX-OR gate and its truth table

2.1.2 Number System

- Every computer stores numbers, letters, and other special characters in coded form.
- The number system is classified based on the number of characters/symbols/numbers that it supports, which is called as the base or radix of a number system.
- The most commonly used number systems are
 - Decimal number system
 - Binary Number System

Decimal Number System

- The base of the decimal number system is 10.
- The digits start from 0 to 9.

- The successive positions, read from left to right represent units, tens, hundreds, thousands, and so on.
- Example: the decimal number 7516 (written as 7516_{10}) consists of the digit 6 in the units position, 1 in the tens position, 5 in the hundreds position, and 7 in the thousands position, and its value can be written as: $(7 \times 1000) + (5 \times 100) + (1 \times 10) + (6 \times 1)$ (or) $2000 + 500 + 80 + 6$ (or) 2586.

Binary Number System

- The base of the binary number system is 2.
- "Binary digit" is often referred to by the common abbreviation as 'bit'.
- A "bit" in computer terminology means either a 0 or a 1.
- Each position in a binary number represents a power of the base (2).

1 Bit	2 Bit	3 Bit	4 Bit	Decimal Value
1	0 0	0 0 0	0 0 0 0	0
0	0 1	0 0 1	0 0 0 1	1
	1 0	0 1 0	0 0 1 0	2
	1 1	0 1 1	0 0 1 1	3
		1 0 0	0 1 0 0	4
		1 0 1	0 1 0 1	5
		1 1 0	0 1 1 0	6
		1 1 1	0 1 1 1	7
			1 0 0 0	8
			1 0 0 1	9
			1 0 1 0	10
			1 0 1 1	11
			1 1 0 0	12
			1 1 0 1	13
			1 1 1 0	14
			1 1 1 1	15

2.1.3 Fixed Point (Integer) Representation

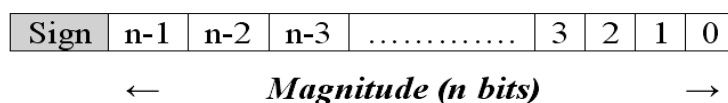
- A **fixed-point number** representation is a real data type for a number.
- It has a set of digits after and before the radix point ('.').
- Fixed-point numbers are useful for representing fractional values (decimal and binary numbers).
- It is used in low-cost embedded microprocessors since
 - the processor has no floating point unit (FPU).
 - it provides improved performance or accuracy.

2.1.4 Signed Numbers Representation

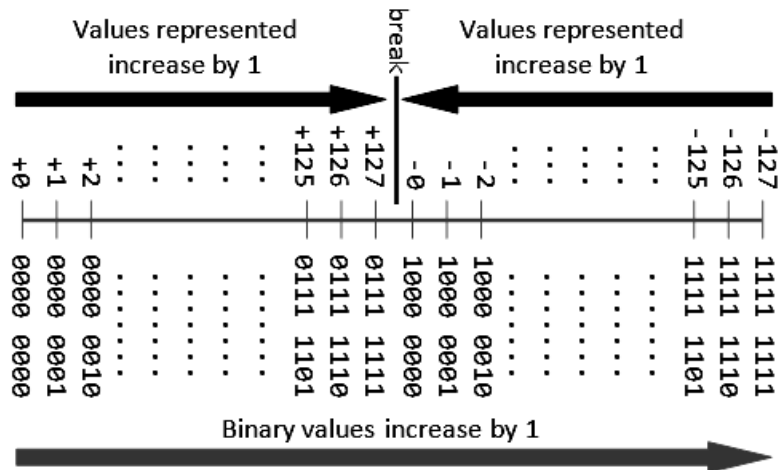
- This is used to represent zero, positive and negative numbers.
- Three representation schemes had been proposed for signed integers:
 1. Sign-Magnitude representation
 2. 1's Complement representation
 3. 2's Complement representation

1. Sign-Magnitude representation

- An n-bit signed binary number consists of two parts
 - Sign bit
 - It is the left most bit.
 - It is also called as the Most Significant Bit (MSB)
 - Sign Bit Value
 - 0 - Positive Integer or Zero
 - 1 - Negative Integer or Zero
 - Magnitude bits
 - Remaining n-1 bits - Magnitude



- **Sign Magnitude representation from -127 to +127 integer**



- **Example**

- Suppose that $n=8$ and the binary representation is $0\ 100\ 0001_2$

- Sign bit

- $0 \Rightarrow$ positive

- Absolute value

- $100\ 0001_2 = 65_{10}$

Hence, the integer is $+65_{10}$

- Suppose that $n=8$ and the binary representation is $1\ 100\ 0001_2$

- Sign bit

- $1 \Rightarrow$ Negative

- Absolute value

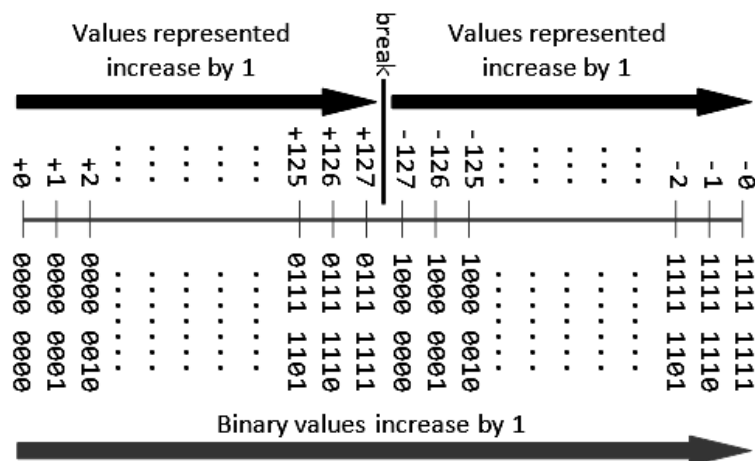
- $100\ 0001_2 = 65_{10}$

Hence, the integer is -65_{10}

- **Drawbacks**

- Two representations ($0000\ 0000_2$ and $1000\ 0000_2$) for the number zero, which could lead to inefficiency and confusion
- Positive and negative integers need to be processed separately

- An n-bit signed binary number consists of two parts
 - Sign bit
 - The left most bit, also called the Most Significant Bit (MSB)
 - Sign Bit Value
 - 0 - Positive Integer or Zero
 - 1 - Negative Integer or Zero
 - Magnitude
 - Remaining n-1 bits - Magnitude
 - Positive Integer
 - Absolute value of the integer is equal to “the magnitude of the (n-1)-bit binary pattern”
 - Negative Integer
 - Absolute value of the integer is equal to “the magnitude of the complement (inverse) of the (n-1)-bit binary pattern”
 - Hence called 1's complement
- **1's Complement representation from -127 to +127 integer**



- **Example**

- Suppose that $n=8$ and the binary representation is $0\ 100\ 0001_2$

- Sign bit

- $0 \Rightarrow$ positive

- Absolute value

- $100\ 0001_2 = 65_{10}$

Hence, the integer is $+65_{10}$

- Suppose that $n=8$ and the binary representation is $1\ 100\ 0001_2$

- Sign bit

- $1 \Rightarrow$ Negative

- Absolute value

- $100\ 0001_2 = 011\ 1110_2$ (1's Complement)

$$= 62_{10}$$

Hence, the integer is -62_{10}

- **Drawbacks**

- Two representations ($0000\ 0000_2$ and $1111\ 1111_2$) for the number zero, which could lead to inefficiency and confusion
- Positive and negative integers need to be processed separately

3. 2's Complement representation

- An n -bit signed binary number consists of two parts

- Sign bit

- The left most bit, also called the Most Significant Bit (MSB)

- Sign Bit Value

- 0 - Positive Integer or Zero
 - 1 - Negative Integer or Zero

-
- The diagram illustrates the wrap-around of a 4-bit counter. It is divided into two main sections: the top section shows the counter reaching its maximum value (+127) and then wrapping around to its minimum value (-128), and the bottom section shows the counter continuing from -128 and wrapping around to its maximum value (+127).
- Top Section: Values represented increase by 1**
- The top section shows the counter values from +125 to +127, followed by a 'break' symbol, and then values from -128 to -1. The binary values are shown below the decimal values, increasing by 1.
- | Values represented increase by 1 | Binary values increase by 1 |
|----------------------------------|-----------------------------|
| +125 | 0111 1101 |
| +126 | 0111 1110 |
| +127 | 0111 1111 |
| -128 | 1000 0000 |
| -127 | 1000 0001 |
| -126 | 1000 0010 |
| -125 | 1000 0011 |
| -124 | 1000 0100 |
| -123 | 1000 0101 |
| -122 | 1000 0110 |
| -121 | 1000 0111 |
| -120 | 1000 1000 |
| -119 | 1000 1001 |
| -118 | 1000 1010 |
| -117 | 1000 1011 |
| -116 | 1000 1100 |
| -115 | 1000 1101 |
| -114 | 1000 1110 |
| -113 | 1000 1111 |
| -112 | 1001 0000 |
| -111 | 1001 0001 |
| -110 | 1001 0010 |
| -109 | 1001 0011 |
| -108 | 1001 0100 |
| -107 | 1001 0101 |
| -106 | 1001 0110 |
| -105 | 1001 0111 |
| -104 | 1001 1000 |
| -103 | 1001 1001 |
| -102 | 1001 1010 |
| -101 | 1001 1011 |
| -100 | 1001 1100 |
| -99 | 1001 1101 |
| -98 | 1001 1110 |
| -97 | 1001 1111 |
| -96 | 1010 0000 |
| -95 | 1010 0001 |
| -94 | 1010 0010 |
| -93 | 1010 0011 |
| -92 | 1010 0100 |
| -91 | 1010 0101 |
| -90 | 1010 0110 |
| -89 | 1010 0111 |
| -88 | 1010 1000 |
| -87 | 1010 1001 |
| -86 | 1010 1010 |
| -85 | 1010 1011 |
| -84 | 1010 1100 |
| -83 | 1010 1101 |
| -82 | 1010 1110 |
| -81 | 1010 1111 |
| -80 | 1011 0000 |
| -79 | 1011 0001 |
| -78 | 1011 0010 |
| -77 | 1011 0011 |
| -76 | 1011 0100 |
| -75 | 1011 0101 |
| -74 | 1011 0110 |
| -73 | 1011 0111 |
| -72 | 1011 1000 |
| -71 | 1011 1001 |
| -70 | 1011 1010 |
| -69 | 1011 1011 |
| -68 | 1011 1100 |
| -67 | 1011 1101 |
| -66 | 1011 1110 |
| -65 | 1011 1111 |
| -64 | 1100 0000 |
| -63 | 1100 0001 |
| -62 | 1100 0010 |
| -61 | 1100 0011 |
| -60 | 1100 0100 |
| -59 | 1100 0101 |
| -58 | 1100 0110 |
| -57 | 1100 0111 |
| -56 | 1100 1000 |
| -55 | 1100 1001 |
| -54 | 1100 1010 |
| -53 | 1100 1011 |
| -52 | 1100 1100 |
| -51 | 1100 1101 |
| -50 | 1100 1110 |
| -49 | 1100 1111 |
| -48 | 1101 0000 |
| -47 | 1101 0001 |
| -46 | 1101 0010 |
| -45 | 1101 0011 |
| -44 | 1101 0100 |
| -43 | 1101 0101 |
| -42 | 1101 0110 |
| -41 | 1101 0111 |
| -40 | 1101 1000 |
| -39 | 1101 1001 |
| -38 | 1101 1010 |
| -37 | 1101 1011 |
| -36 | 1101 1100 |
| -35 | 1101 1101 |
| -34 | 1101 1110 |
| -33 | 1101 1111 |
| -32 | 1110 0000 |
| -31 | 1110 0001 |
| -30 | 1110 0010 |
| -29 | 1110 0011 |
| -28 | 1110 0100 |
| -27 | 1110 0101 |
| -26 | 1110 0110 |
| -25 | 1110 0111 |
| -24 | 1110 1000 |
| -23 | 1110 1001 |
| -22 | 1110 1010 |
| -21 | 1110 1011 |
| -20 | 1110 1100 |
| -19 | 1110 1101 |
| -18 | 1110 1110 |
| -17 | 1110 1111 |
| -16 | 1111 0000 |
| -15 | 1111 0001 |
| -14 | 1111 0010 |
| -13 | 1111 0011 |
| -12 | 1111 0100 |
| -11 | 1111 0101 |
| -10 | 1111 0110 |
| -9 | 1111 0111 |
| -8 | 1111 1000 |
| -7 | 1111 1001 |
| -6 | 1111 1010 |
| -5 | |

- **Example**
 - Suppose that $n=8$ and the binary representation is $0\ 100\ 0001_2$
 - Sign bit
 - $0 \Rightarrow$ positive

- Absolute value

- $100\ 0001_2 = 65_{10}$

Hence, the integer is $+65_{10}$

- Suppose that $n=8$ and the binary representation is $1\ 100\ 0001_2$

- Sign bit

- $1 \Rightarrow \text{Negative}$

- Absolute value

- $100\ 0001_2 = 011\ 1110_2$ (1's Complement)
 $= 011\ 1111_2$ (2's Complement)
 $= 63_{10}$

Hence, the integer is -63_{10}

2.1.5 Unsigned Numbers

- An n -bit unsigned binary number contains only magnitude part
 - All n bits – Magnitude
 - Represent integers from 0 to $2^n - 1$

n	Minimum	Maximum
8	0	$2^8 - 1 = 255$
16	0	$2^{16} - 1 = 65,535$
32	0	$2^{32} - 1 = 4,294,967,295$ (9+ digits)
....		

- Represent zero and positive integers, but not negative integers

Floating-point Number

- The fractional binary numbers are represented by considering the binary point.
- If binary point is assumed to the right of the sign bit, we represent the fractional binary numbers as,

$$B = (b_0 * 2^0 + b_1 * 2^{-1} + b_2 * 2^{-2} + \dots + b_{(n-1)} * 2^{-(n-1)})$$

2.2 ADDITION AND SUBTRACTION OF SIGNED NUMBERS

2.2.1 Addition of numbers

- The ALU performs addition by adding each bit of the addend with every bit of the augend from right to left.
- At the end of every bit by bit addition, the sum bit is noted and the carry bit is passed to the immediate digit to the left.

Terminologies

Carry

This represents the overflow result while performing addition of two or more binary numbers.

Example

Carry →	0	1	1	-
Augend →	1	0	1	1
Addend →	0	0	1	1
Operator →	+			

Resultant →	1	1	1	0

Least Significant Bit (LSD)

The leftmost bit of every binary number is the LSD of the binary number.

Most significant Bit (MSD)

The rightmost bit of every binary number is the MSD of the binary number.

Example

1	1	0	0
↑			↑
MSD			LSD

- Binary addition always starts from LSD and move towards MSD with or without having carry bit.
- The table below shows the binary addition rules which are followed while performing addition of two binary numbers.

1.53 Computer Architecture

Let the general expression be: **operand1 + operand2**.

Operand1	Operand2	Result	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

For the expression that involves three operands, the general expression shall be:
operand1 + operand2 + operand3.

Operand1	Operand2	Operand3	Result	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Steps involved in performing addition of two binary numbers are,

Step 1: Start from the LSD of both the numbers by applying the rules of binary addition.

Step 2: The result of adding the individual bits of each column is noted in the result row, with the carry bit carried over to the preceding column.

Step 3: The carry bit is considered while adding the preceding column bits.

Step 4: Repeat the above steps until MSD is processed.

Example**1. Add $(101)_2$ with $(1110)_2$.**

$$\begin{array}{r}
 \text{Carry} \rightarrow 1 \quad 1 \quad 0 \quad 0 \quad - \\
 \\
 \text{Augend} \rightarrow 1 \quad 0 \quad 1 \\
 \text{Addend} \rightarrow \downarrow 1 \quad 1 \quad 1 \quad 0 \\
 \text{Operator} \rightarrow + \\
 \hline
 \text{Sum} \rightarrow 1 \quad 0 \quad 0 \quad 1 \quad 1
 \end{array}$$

Result = $(10011)_2$ **2. Add $(100)_2$ with $(11)_2$.**

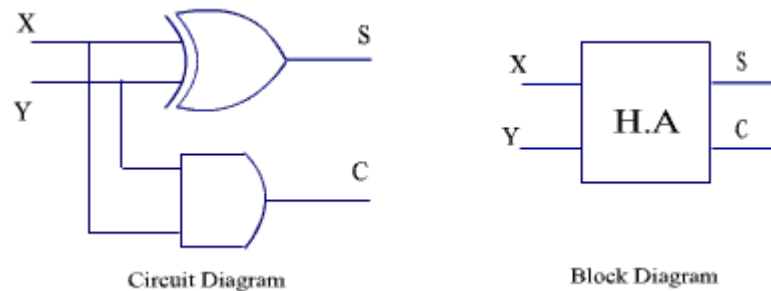
$$\begin{array}{r}
 \text{Carry} \rightarrow 0 \quad 0 \quad - \\
 \\
 \text{Augend} \rightarrow 1 \quad 0 \quad 0 \\
 \text{Addend} \rightarrow 0 \quad 1 \quad 1 \\
 \text{Operator} \rightarrow + \\
 \hline
 \text{Sum} \rightarrow 1 \quad 1 \quad 1
 \end{array}$$

Result = $(111)_2$ **3. Add $(1110)_2$ with $(1101)_2$.**

$$\begin{array}{r}
 \text{Carry} \rightarrow 1 \quad 1 \quad 0 \quad 0 \quad - \\
 \\
 \text{Augend} \rightarrow 1 \quad 1 \quad 1 \quad 0 \\
 \text{Addend} \rightarrow \downarrow 1 \quad 1 \quad 0 \quad 1 \\
 \text{Operator} \rightarrow + \\
 \hline
 \text{Sum} \rightarrow 1 \quad 1 \quad 0 \quad 1 \quad 1
 \end{array}$$

Result = $(11011)_2$

The circuit is implemented as follows.



Circuit diagram and Block diagram of Half Adder

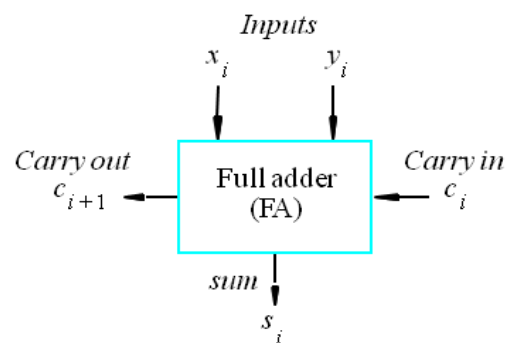
This circuit cannot handle the carry input, so it is termed as half adder.

2.2.2.2 Full Adder circuit

- A full adder is a combinational circuit that forms the arithmetic sum of three bits.
- It consists of three inputs and two outputs.
- Two of the input variables, denoted by x and y, represent the two bits to be added.
- The third input Z, represents the carry from the previous lower position.

Block Diagram

The complete circuit for single stage of addition is given



At the i^{th} stage:

- Input:
 - x_i , the first input
 - y_i , the second input
 - c_i , the carry-in
- Output:
 - s_i is the sum
 - c_{i+1} carry-out to $(i+1)^{\text{st}}$ state

Truth Table

x_i	y_i	Carry-in C_i	Sum S	Carry-out C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The simplified expression for Sum, S and C are

$$S = x'y'z + x'yz' + xy'z + xyz$$

$$C = xy + xz + yz$$

$$= xy + xy'z + x'yz$$

The above expressions can be written as follows.

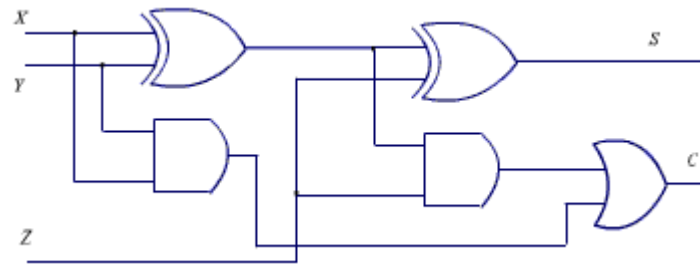
$$S = z \oplus (x \oplus y)$$

$$= z'(xy' + x'y) + z(xy' + x'y)$$

$$= z'(xy' + x'y) + z(xy + x'y')$$

$$= xy'z' + x'yz' + xyz + x'y'z$$

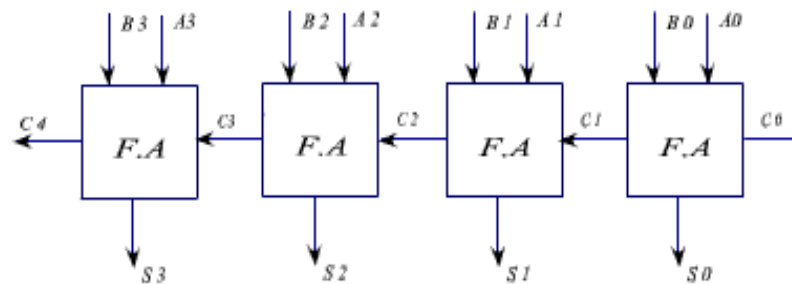
$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$

Circuit diagram - full adder

Full Adder

2.2.2.3 4 – bit Adder circuit

- To get the four bit adder, we have to use 4 full adder blocks.
- The carry output the lower bit is used as a carry input to the next higher bit.



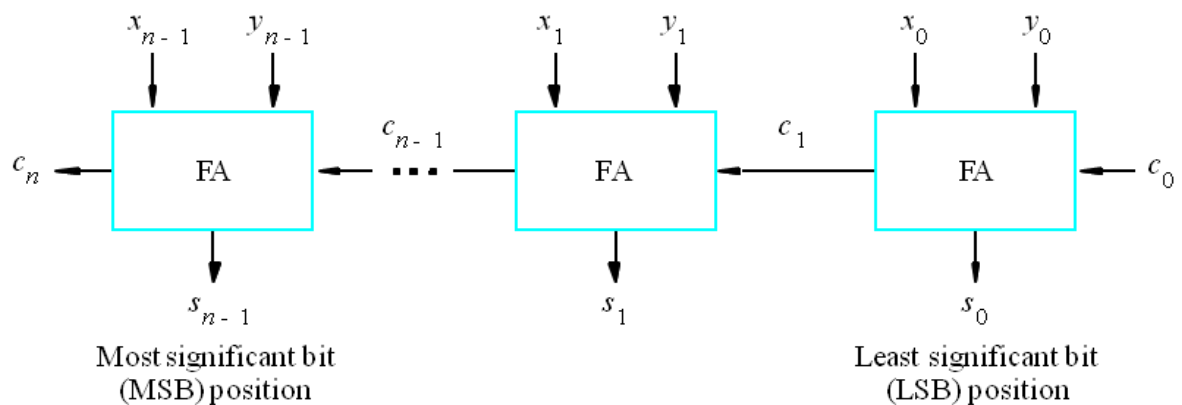
4-bit adder circuit

2.2.2.4 n-bit ripple-carry adder

- Cascade n full adder (FA) block to form a n-bit adder.
- Each full adder inputs a C_{in} , which is the C_{out} of the previous adder
 - This adder is called as a n-bit ripple carry adder
 - Each carry bit “ripples” or “propagates” to the next full adder

Block Diagram

- Used to add two input X and Y
- x_{n-1} and y_{n-1} – Sign bits



- Carry-in c_0 into the LSB position provides a convenient way to perform subtraction

Advantage

- Allows for fast design time

Disadvantage

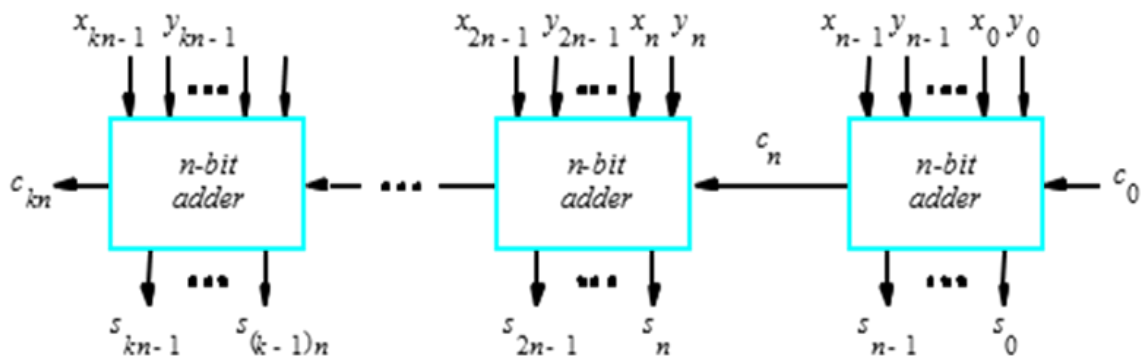
- Relatively slow
 - Each full adder must wait for the carry bit to be calculated from the previous full adder

2.2.2.5 Cascade of k n-bit Adders

- Circuit to add K n-bit numbers by cascading k n-bit adders
- Each n-bit adder forms a block, so this is cascading of blocks
- Carries ripple or propagate through blocks, Blocked Ripple Carry Adder

Block Diagram

- The carry-in, c_0 , into the least-significant-bit (LSB) position provides a convenient means of adding 1 to a number
- Forming the 2's-complement of a number involves adding 1 to the 1's complement of the number

**2.2.3 Subtraction of numbers**

- The ALU performs subtraction by subtracting each bit of the minuend with every bit of the subtrahend from right to left.
- At the end of every bit by bit subtraction, the difference bit is noted and the borrow bit is passed to the immediate digit to the left.

Terminologies

Borrow

This represents the underflow value while performing subtraction of two or more binary numbers.

Borrowing occurs whenever a smaller bit (0) is subtracted by a larger bit (1).

Example

$$\begin{array}{r}
 \text{Borrow} \rightarrow \quad - \quad 1 \quad - \\
 \quad \quad \quad 0 \quad 10 \\
 \quad \quad \quad \uparrow \quad \uparrow \\
 \text{Minuend} \rightarrow \quad 1 \quad 0 \quad 1 \\
 \text{Subtrahend} \rightarrow \quad 0 \quad 1 \quad 1 \\
 \text{Operator} \rightarrow \quad - \\
 \hline
 \text{Difference} \rightarrow \quad 0 \quad 1 \quad 0
 \end{array}$$

- Binary subtraction starts from LSD and move towards MSD with or without borrowing a bit from the preceding bit.
- The table below shows the binary addition rules which are followed while performing subtraction of two binary numbers.

Let the general expression be: **operand1 - operand2**.

Operand1	Operand2	Result	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Steps involved in performing subtraction of two binary numbers

- Step 1: Start from the LSD of both the numbers by applying the rules of binary subtraction.
- Step 2: The difference of each column is noted, with the borrow bit taken from the preceding column, if required.
- Step 3: The borrow bit is considered while subtracting the preceding column bits.
- Step 4: Repeat the above steps until MSD is processed.
- Step 5: If the borrow bit of the MSD of the minuend is '0', the resultant difference is a positive value. Stop the operation.
- Step 6: If the borrow bit of the MSD of the minuend is '1', the resultant difference is a negative value. Hence perform two's complementation of the difference value and stop the operation.

Finding Two's complement

1. Take one's complement by inverting 0's by 1's and 1's by 0's.
2. Add '1' to the one's complement result.

Example: 1. Subtract $(110)_2$ by $(101)_2$.

Borrow →	0	0	1
		0	10
		↑	↑
Minuend →	1	1	0
Subtrahend →	1	0	1
Operator →	-		

Difference →	0	0	1

Result = $(001)_2$

Example: 2. Subtract $(100)_2$ by $(11)_2$.

Borrow \rightarrow	0	1	1
		1	
		\uparrow	
		10	10
		\uparrow	\uparrow
Minuend \rightarrow	4	0	0
Subtrahend \rightarrow	0	1	1
Operator \rightarrow	-		

Difference \rightarrow	0	0	1

Example: 3. Subtract $(101)_2$ by $(110)_2$.

Borrow \rightarrow	1	1	0
		0	10
		\uparrow	\uparrow
Minuend \rightarrow	4	0	1
Subtrahend \rightarrow	1	1	0
Operator \rightarrow	-		

Difference \rightarrow	1	1	1

Taking two's complement of $(111) \rightarrow 000 + 1 \rightarrow 001$.

Result = - $(001)_2$ [‘-’ is indicated since the borrow bit of the minuend's MSD is 1]

Example: 4. Subtract $(11)_2$ by $(100)_2$.

$$\begin{array}{r}
 \text{Borrow} \rightarrow \quad \quad \quad \mathbf{1} \quad \mathbf{0} \quad \mathbf{0} \\
 \quad \quad \quad \quad \quad \quad 10 \\
 \quad \quad \quad \quad \quad \quad \uparrow \\
 \text{Minuend} \rightarrow \quad \quad \quad 0 \quad 1 \quad 1 \\
 \text{Subtrahend} \rightarrow \quad \quad 1 \quad 0 \quad 0 \\
 \text{Operator} \rightarrow \quad \quad - \\
 \hline
 \text{Difference} \rightarrow \quad \quad \mathbf{1} \quad \mathbf{1} \quad \mathbf{1}
 \end{array}$$

Taking two's complement of $(111) \rightarrow 000 + 1 \rightarrow 001$.

Result = - $(001)_2$ ['-' is indicated since the borrow bit of the minuend's MSD is 1]

Other Examples

1. $00100101 - 00010001 = 00010100$

$$\begin{array}{r}
 \quad \quad \quad 0 \quad 1 \quad \quad \quad \text{borrows} \\
 0 \quad 0 \quad \cancel{1} \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 = 37_{(\text{base } 10)} \\
 - 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 1 = 17_{(\text{base } 10)} \\
 \hline
 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 = 20_{(\text{base } 10)}
 \end{array}$$

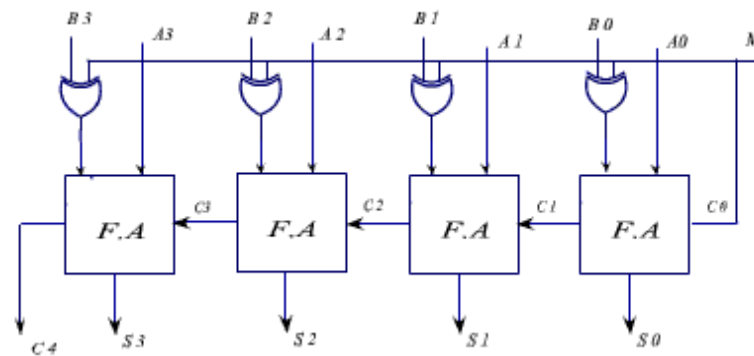
2. $00110011 - 00010110 = 00011101$

$$\begin{array}{r}
 \quad \quad \quad 1 \\
 \quad \quad \quad 0 \quad 0 \quad 1 \quad 1 \quad \quad \quad \text{borrows} \\
 0 \quad 0 \quad \cancel{1} \quad \cancel{1} \quad 0 \quad 0 \quad 1 \quad 1 = 51_{(\text{base } 10)} \\
 - 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 = 22_{(\text{base } 10)} \\
 \hline
 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 = 29_{(\text{base } 10)}
 \end{array}$$

2.2.4 Binary Subtractor circuit

- The subtraction operation can be implemented with the help of binary adder circuit, since $(A - B) = A + (-B)$.
- 2's complement representation of a number is treated as a negative number of the given number.
- 2's complement of a number shall be calculated by complementing each bit and adding 1 to it.

- The circuit for subtracting A-B consist of an added with inverter placed between each data input B and the corresponding input of the full adder.
- The input carry C_0 must be equal to 1 when performing subtraction.
- The operation thus performed becomes A, plus the 1's complement of B , plus 1.
- This is equal to A plus 2's complement of B.
- With this principle, a single circuit can be used for both addition and subtraction.



4-bit adder subtractor

- The circuit diagram of a 4-bit adder subtractor is shown above.
- Here, the mode (M) is the selection input line, which will determine the operation,
- If $M=0$, then addition, $A+B$ is performed.
- If $M=1$, then $(A - B) = A + (-B)$ is performed.

The operation of OR gate:

$$x \oplus 0 = x$$

$$x \oplus 1 = x'$$

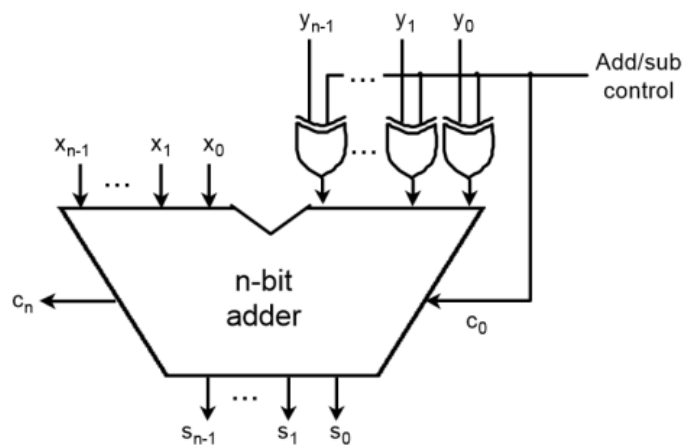
$$\text{If } M=0, \quad B_i \oplus 0 = B_i$$

$$B_i \oplus 1 = B_i'$$

2.2.5 Addition / Subtraction Logic Unit

- Perform subtraction operation, X-Y
 - Find 2's complement of Y
 - Add it with X

- Add/Sub control
 - Used to decide whether addition or subtraction is performed
 - 0 – Addition
 - Supply the Y-vector unchanged to one of the adder input
 - Carry-in signal value is 0
 - 1 - Subtraction
 - Supply the 1's complement of Y-vector
 - Carry-in signal value is 1



2.2.6 Overflow in Addition and Subtraction

- An overflow occurs when the result of the operation cannot be represented with the available hardware; here it is 32-bit word. While adding or subtracting two 32-bit numbers may yield a result that needs 33rd bit to be full expressed.
- Overflow cannot occur when adding operands with different signs. 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.
- Similarly in Subtraction we subtract by negating the second operand and then add i.e. $x - y = x + (-y)$. Therefore, when we subtract operands of the same sign we are actually adding operands of different signs.
- When can overflow occur then?
 - In addition, overflow occurs, when adding two positive numbers ends up in a negative result or vice versa. This means a carry out occurred in the sign bit.

- In Subtraction, overflow occurs when we subtract a negative number from a positive number and get a negative result, or vice versa. This means borrow occurred from the sign bit.
- Unsigned integers are commonly used for memory addresses where overflows are ignored.

Operation	Operand A	Operand B	Result indicating overflow
A+B	(+) ve	(+) ve	(-) ve
A+B	(-) ve	(-) ve	(+) ve
A-B	(+) ve	(-) ve	(-) ve
A-B	(-) ve	(+) ve	(+) ve

- The MIPS solution is to have two kinds of arithmetic instructions to recognize the two choices:
 - 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.
- MIPS detects overflow with an exception, also called an interrupt on many computers. An exception or interrupt is essentially an unscheduled procedure call. The address of the instruction that overflowed is saved in a register, and the computer jumps to a predefined address to invoke the appropriate routine for that exception. The interrupted address is saved so that in some situations the program can continue after corrective code is executed.
- MIPS includes a register called the exception program counter (EPC) to contain the address of the instruction that caused the exception

2.3 MULTIPLICATION

- Multiplication of two numbers is also performed on binary digits.
- The operand to be multiplied is called the multiplicand.
- The second operand that quantifies the multiplicand is called the multiplier.
- The final result is the product value that is obtained after multiplication.

2.3.1 Manual multiplication algorithm

- Multiplication process involves generation of partial product, one for each digit in multiplier
 - Partial products are added to produce the final product
- In binary system partial products are easily defined

- If multiplier bit is 0, the partial product is 0
- If multiplier bit is 1, the partial product is multiplicand
- Product of two n-digit numbers can be accommodated in **2n** digits
 - So, product of the two 4-bit numbers fits into 8 bits

Example

Multiplying 1101 by 1011

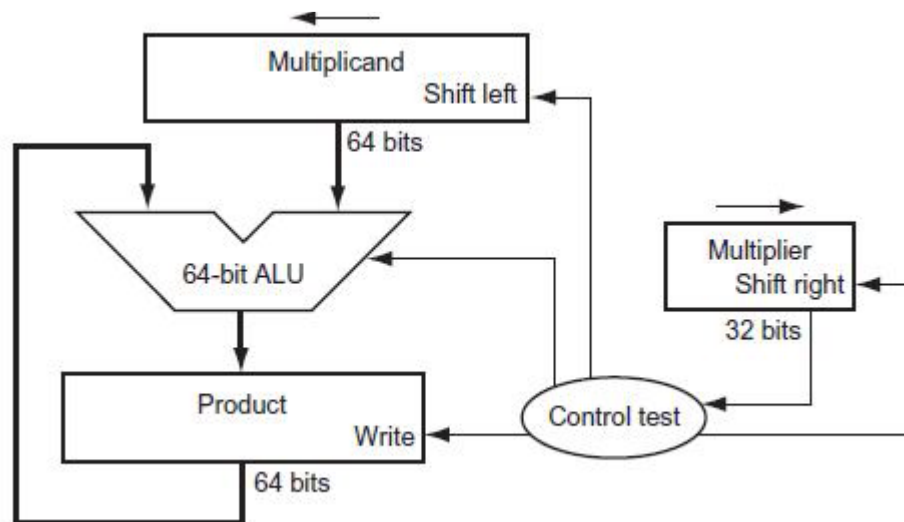
Multiplicand, M : (13)					1	1	0	1
Multiplier, Q : (11)				×	1	0	1	1
					<hr/>			
Partial products	{				1	1	0	1
				1	1	0	1	
			0	0	0	0		
		1	1	0	1			
					<hr/>			
Final Product, P : (143)		1	0	0	0	1	1	1

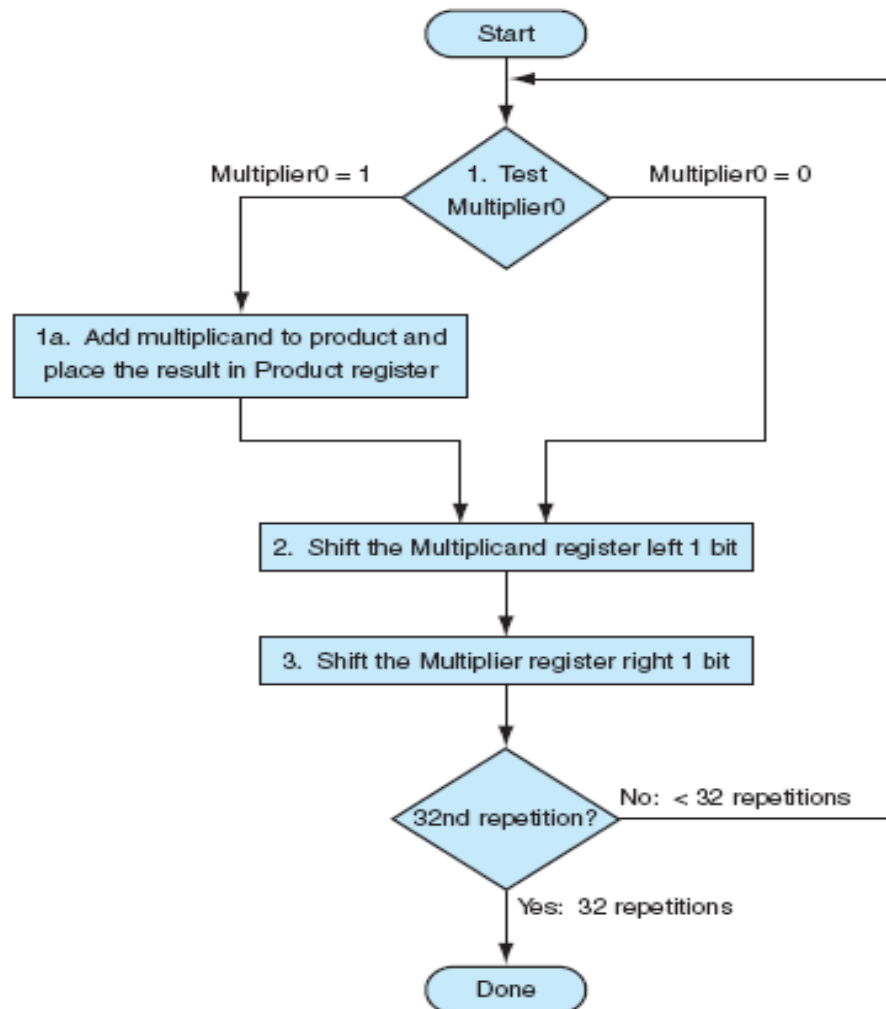
- First operand is called Multiplicand, second one is Multiplier and final result is called product
- Each successive partial product is shifted one position to the left relative to the next partial product
- Final product is produced by summing the partial products

2.3.1.1 Multiplication Hardware – First Version

- The hardware given below is used to multiply two binary data.
- The Multiplicand register, the ALU and the product register handle 64 bits whereas the multiplier register can handle 32 bits.
- The multiplicand is processed from right to left, by shifting left one bit on each step.
- The multiplier is shifted in the opposite direction at each step.
- The product register is initialized to 0. The value changes based on the control test decision.

- The control test component selects when to shift the multiplicand and the multiplier and when to write product bits in product register.

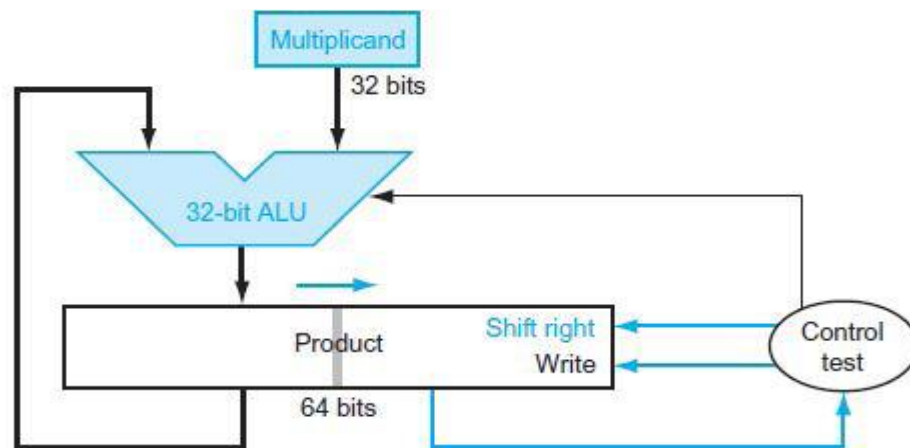
**Flowchart**



- The least significant bit of the multiplier (Multiplier0) determines whether the multiplicand is added to the Product register.
- The left shift in step 2 has the effect of moving the intermediate operands to the left, just as when multiplying by hand.
- The shift right in step 3 gives us the next bit of the multiplier to examine in the following iteration.
- 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.
- The relative importance of arithmetic operations like multiply varies with the program, but addition and subtraction may be anywhere from 5 to 100 times more popular than multiply.

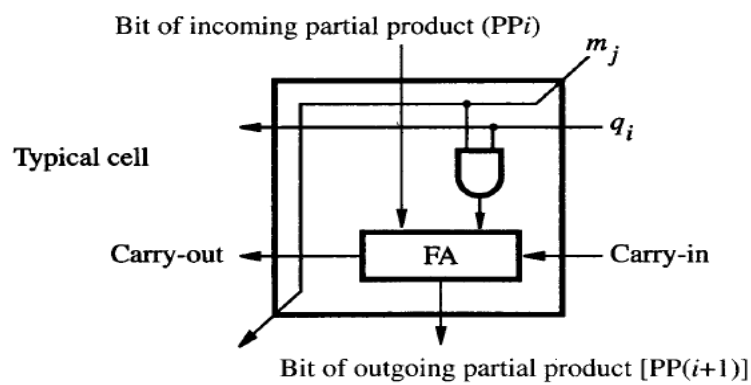
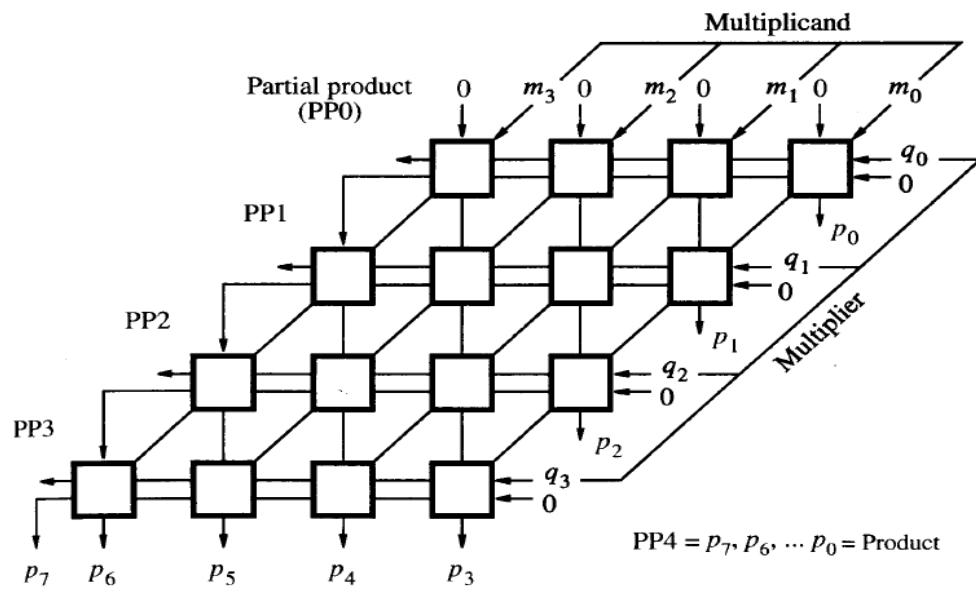
2.3.1.2 Multiplication Hardware - Refined Version

- The refined version has 32 bit Multiplicand register and ALU, with the product register having 64 bits long.
- The multiplier register is placed instead of the right half of the product register.
- Based on the decision made by the control test, the product register is shifted right one bit at each step.



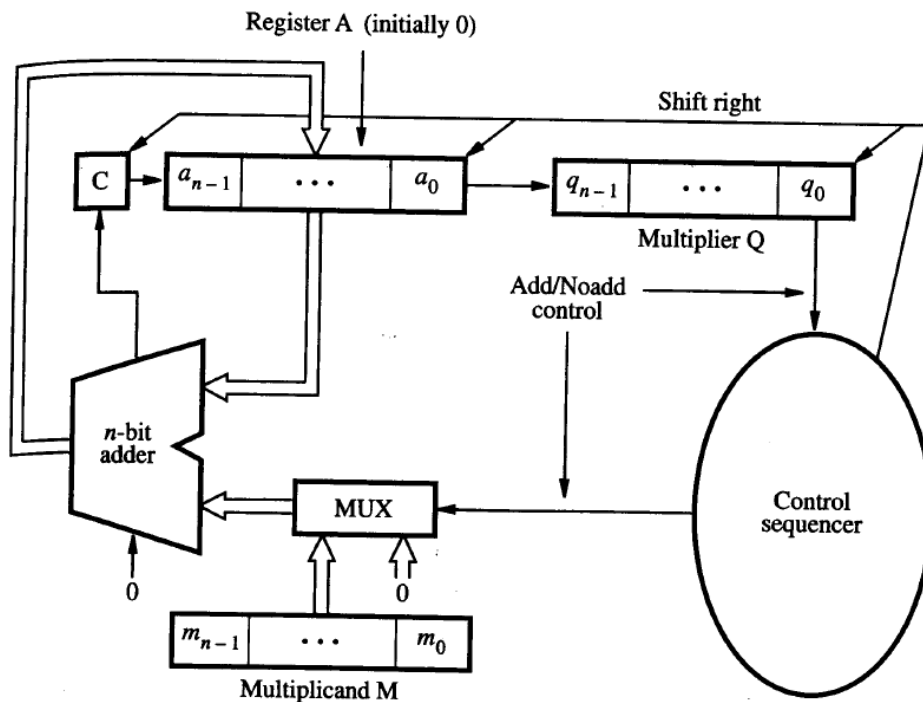
2.3.1.3 Array Implementation of positive binary operands

- Implemented in a combinational two-dimensional logic array
- Full Adder(FA)
 - Main component in each cell
- AND gate
 - Determines whether a multiplicand bit, m_j , is added to the incoming partial-product bit, based on the value of the multiplier bit, q_i .
- If $q_i=0$, PP_i is passed vertically downward unchanged.
- If $q_i=1$, PP_i is added with the multiplicand to generate $PP(i+1)$, for each row $0 \leq i \leq 3$.
- PP_0 is all 0's.
- PP_4 is the desired product.



2.3.1.4 Sequential Circuit Binary Multiplier

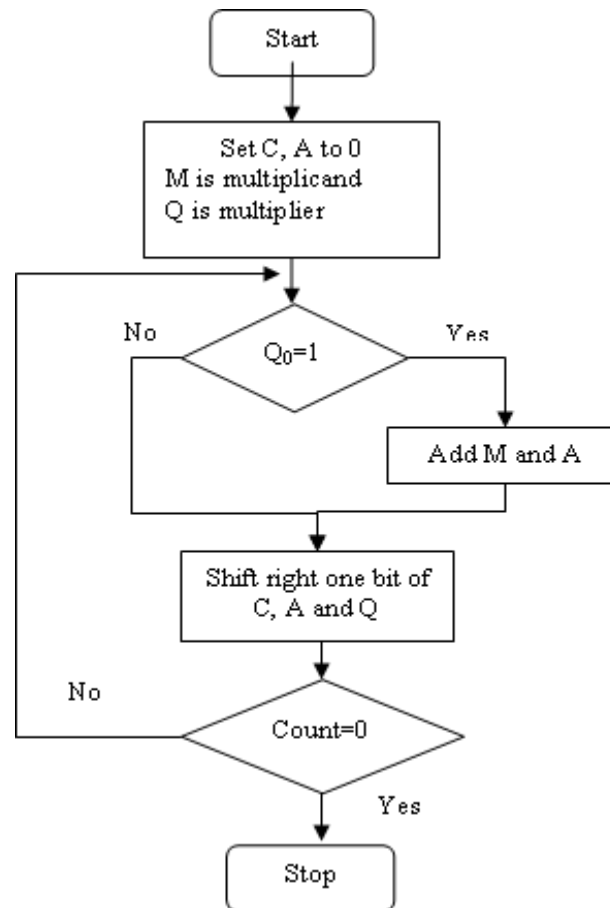
Register Configuration



Algorithm

- Multiplier is loaded into the register Q and multiplicand is loaded into the register B
- Register C and A is initially set to 0
- Check the whether q_0 of Q register bit is 0 or 1
- If q_0 bit is 1
 - Add multiplicand and partial product
 - Shift all bits of C,A and Q registers to the right one bit
 - C bit goes to A_{n-1} , A_0 goes to Q_{n-1} , Q_0 is lost
- If q_0 bit is 0
 - No need to perform addition
 - Shift all bits of C,A and Q registers to the right one bit
- Repeat the step to get the desired results in A and Q registers

Flowchart

**Example**

- Multiplicand (M) : 1101
- Multiplier (Q) : 1011

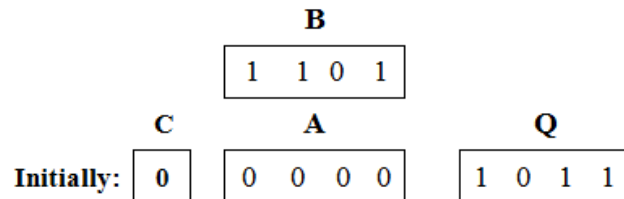
Longhand Multiplication

Multiplying 1101 by 1011

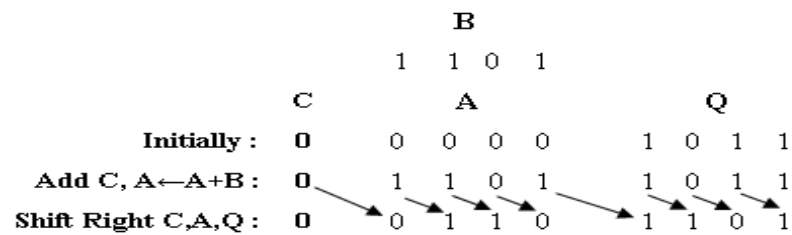
Multiplicand, M : (13)		1	1	0	1				
Multiplier, Q : (11)	×	1	0	1	1				
Partial products	{				1	1	0	1	
					1	1	0	1	
				0	0	0	0		
		1	1	0	1				
Final Product, P : (143)		1	0	0	0	1	1	1	1

Multiplication Process

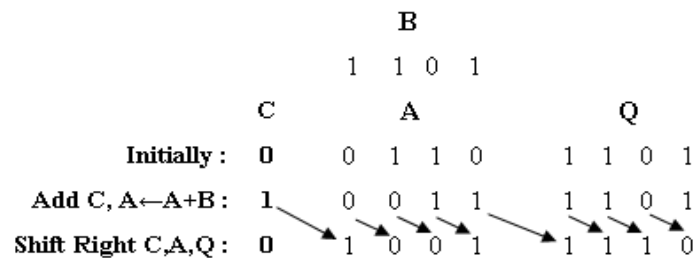
- Initially



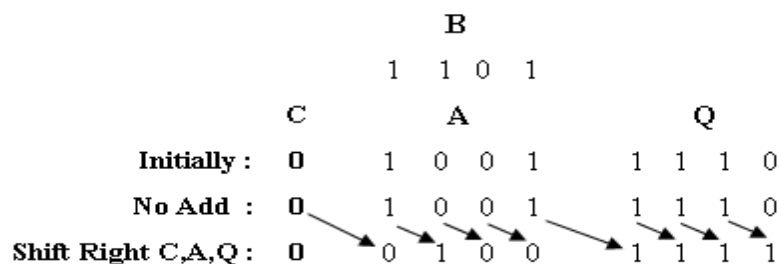
- First Cycle



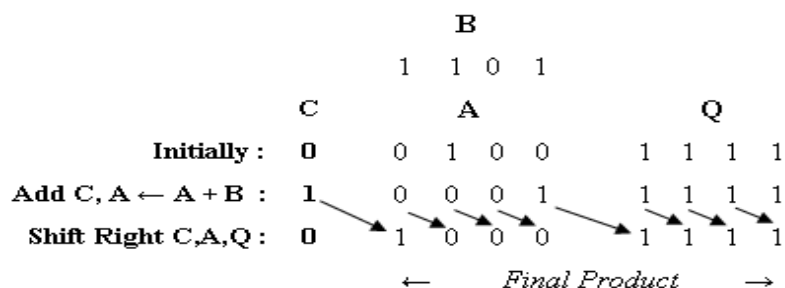
- Second Cycle



- Third Cycle



- Fourth Cycle



- For a negative multiplier, a straightforward solution is to form the 2's-complement of both the multiplier and the multiplicand and proceed as in the case of a positive multiplier
 - Possible because complementation of both operands does not change the value or the sign of the product

2.3.2 Booth's Algorithm

- A technique that works equally well for both negative and positive multipliers.
- Generates a 2n-bit product and treats both positive and negative 2's complement n-bit operands uniformly.
- Consider a multiplication, where the multiplier is positive 0011110 (30)

$$\begin{array}{r}
 \begin{array}{ccccccc}
 & & & & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
 & & & & 0 & 0 & +1 & +1 & +1 & +1 & 0 \\
 \hline
 & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & 0 & 1 & 0 & 1 & 1 & 0 & 1 & \\
 & & 0 & 1 & 0 & 1 & 1 & 0 & 1 & & \\
 & & & 0 & 1 & 0 & 1 & 1 & 0 & 1 & \\
 & & & & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
 & & & & & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & \\
 \hline
 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0
 \end{array}
 \end{array}$$

- To reduce the number of required operations
 - Represent the multiplier in terms of difference between two numbers(2n-bit product)

$$\begin{array}{cccccccc}
 64 & 32 & 16 & 8 & 4 & 2 & 1 \\
 0 & 0 & 1 & 1 & 1 & 1 & 0 \rightarrow 30
 \end{array}$$

- Represent 30 as $32(2^5) - 2(2^1)$

$$\begin{array}{r}
 \begin{array}{cccccccc}
 & & & & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\
 2^5 \rightarrow 32 & \rightarrow & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
 2^1 \rightarrow 2 & - \rightarrow & 0 & 0 & 0 & 0 & 0 & 1 & 0 & - \\
 \hline
 30 & & 0 & 0 & 1 & 1 & 1 & 1 & 0
 \end{array}
 \end{array}$$

								0	1	0	1	1	0	1		
								0 + 1	0	0	0	- 1	0			
								<hr/>								
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
1	1	1	1	1	1	1	0	1	0	0	0	1	1		← 2's complement of the multiplicand	
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	0	0	0	0						
0	0	0	1	0	1	1	0	1								
0	0	0	0	0	0	0	0									
<hr/>																
0	0	0	1	0	1	0	1	0	0	0	1	1	0			

- In the Booth scheme,
 - -1 times the shifted multiplicand is selected when moving from 0 to 1, and
 - +1 times the shifted multiplicand is selected when moving from 1 to 0, as the multiplier is scanned from right to left
- **Example**

$$\begin{array}{cccccccccc} \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ 0 & +1 & -1 & +1 & 0 & -1 & 0 & +1 & 0 & -1 & \end{array}$$

Multiplicand: (+13) \rightarrow 0 1 1 0 1 0 1 1 0 1
Multiplier: (-6) \rightarrow \times 1 1 0 1 0 0 -1 +1 -1 0

					0	1	1	0	1	
					0	-1	+1	-1	0	
					<hr/>					
0	0	0	0	0	0	0	0	0	0	
1	1	1	1	1	0	0	1	1		← 2's Complement of Multiplicand
0	0	0	0	1	1	0	1			← Multiplicand
1	1	1	0	0	1	1				← 2's Complement of Multiplicand
0	0	0	0	0	0					
<hr/>										
1	1	1	0	1	1	0	0	1	0	(- 78)

Booth multiplier Recoding Table

Multiplier		Version of multiplicand selected by bit i
Bit i	Bit i-1	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

16-bit Multiplier Types

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

Worst Multiplier :	Case	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
		↓															
		+1	-	+1	-1	+1	-	+1	-	+1	-1	+1	-	+1	-	+1	-
			1				1		1				1		1		1
Ordinary Multiplier :		1	1	0	0	0	1	0	1	1	0	1	1	1	1	0	0
		↓															
		0	-	0	0	+1	-	+1	0	-1	+1	0	0	0	-	0	0
			1				1								1		
Good Multiplier:		0	0	0	0	1	1	1	1	1	0	0	0	0	1	1	1
		↓															
		0	0	0	+1	0	0	0	0	-1	0	0	0	+1	0	0	-1

Advantage

- Used for both negative and positive integers
- Achieves more efficiency in number of addition

2.3.3 Fast Multiplication

The techniques used for speeding up the multiplication operation are

- Bit-Pair Recoding of Multipliers
- Carry-Save Addition of Summands

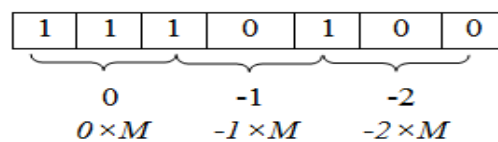
2.3.3.1 Bit-Pair Recoding of Multipliers

- Halves the maximum number of summands (versions of the multiplicand)
- Derived from the booth algorithm

Multiplicand selection decisions table

Multiplier bit-pair		Multiplier bit on the right i-1	Multiplicand selected at position i
i+1	i		
0	0	0	$0 \times M$
0	0	1	$+1 \times M$
0	1	0	$+1 \times M$
0	1	1	$+2 \times M$
1	0	0	$-2 \times M$
1	0	1	$-1 \times M$
1	1	0	$-1 \times M$
1	1	1	$0 \times M$

Example



Using Booth Recoding

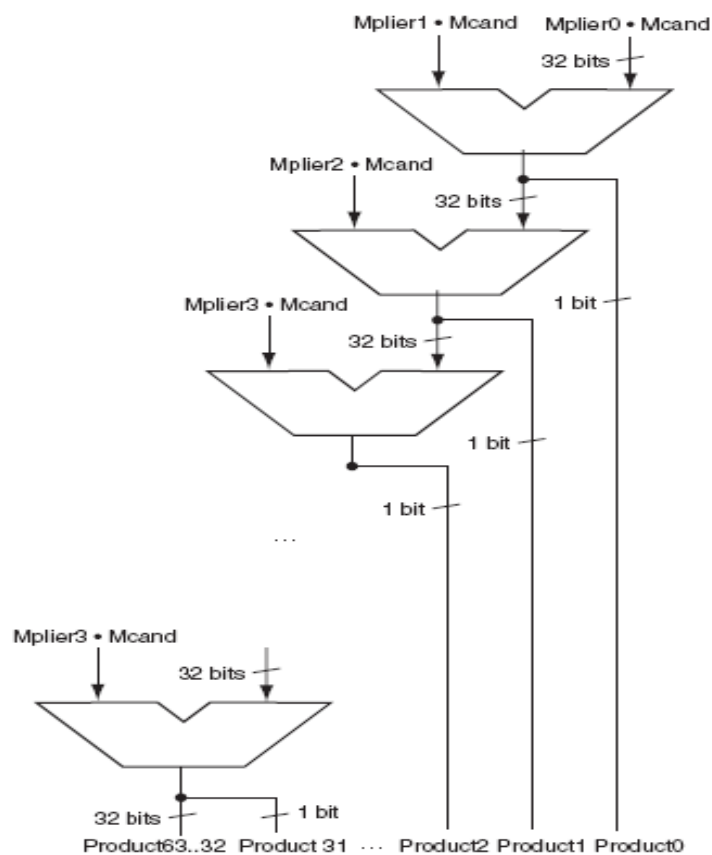
					0	1	1	0	1
					0	-1	+1	-1	0
					<hr/>				
0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	0	0	1	1 ← 2's Complement of Multiplicand
0	0	0	0	0	0	1	1	0	1 ← Multiplicand
1	1	1	1	1	1	0	0	1	1 ← 2's Complement of Multiplicand
0	0	0	0	0	0	0	0	0	
					<hr/>				
1	1	1	0	1	1	0	0	1	0

(- 78)

Using Bit-Pair Recoding

$$\begin{array}{r}
 0 \ 1 \ 1 \ 0 \ 1 \\
 0 -1 -2 \\
 \hline
 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \\
 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \\
 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\
 \hline
 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0
 \end{array}$$

Fast Multiplication hardware

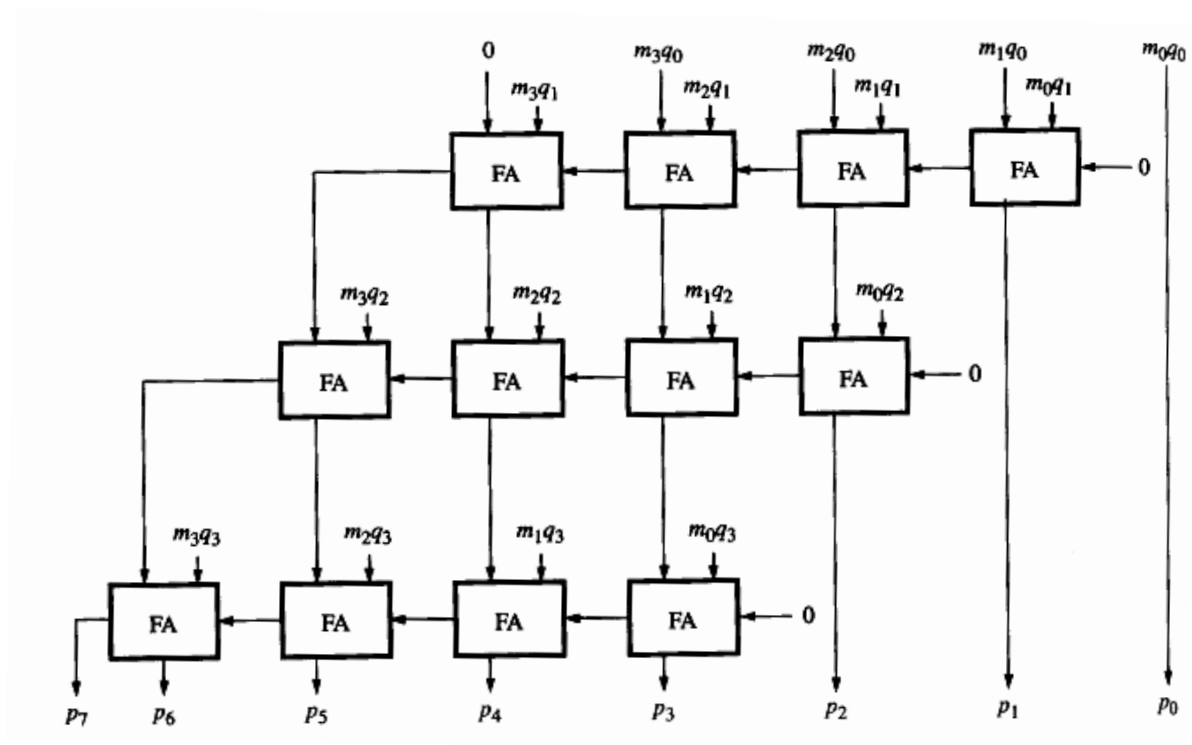


2.3.3.2 Carry-Save Addition (CSA) of Summands

- Reduces the time needed to add the summands and Speed up the addition process

Ripple-carry Array

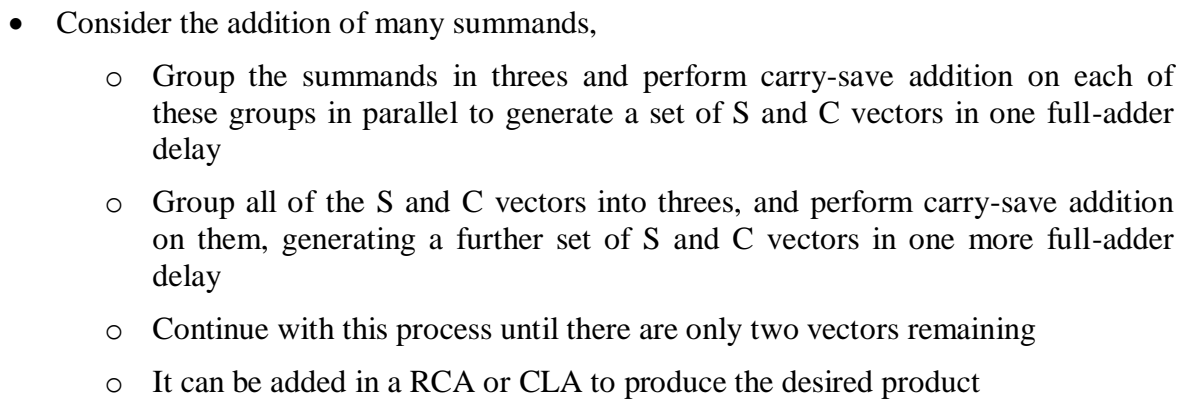
- Also called row ripple form
- Each row consists of AND gates that implement the bit products
 - First Row bit products: m_3q_0 , m_2q_0 , m_1q_0 and m_0q_0



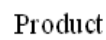
- Carries are ripple along the rows

Carry-save Array

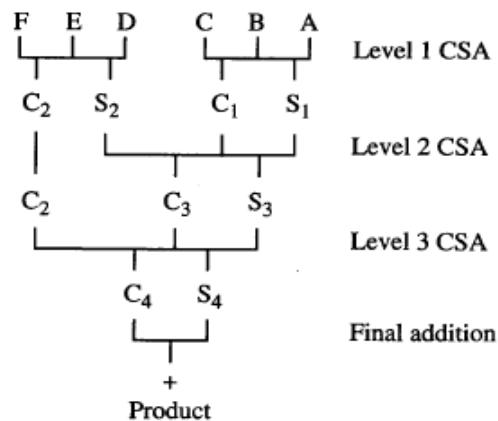
- Carries are saved and introduced into the next row
 - Frees up an input to three full adders in the first row
 - Inputs are introduced the third summand bit products m_2q_2, m_1q_2 and m_0q_2
 - Two inputs of each full adder in the second row are fed by sum and carry outputs from the first row
 - Third input is used to introduce the bit products m_2q_3, m_1q_3 and m_0q_3 of the fourth summand
 - High-order bit products m_3q_2 and m_3q_3 of the third and fourth summands are introduced into the remaining free inputs at the left end in the second and third rows
 - Saved carry bits and sum bits from the second row are added in the third row to produce final product bits
- The delay through the carry-save array is somewhat less than delay through the ripple-carry array. This is because the S and C vector outputs from each row are produced in parallel in one full-adder delay.



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



Schematic representation of the carry-save addition operations



- When the number of summands is large, the time saved is proportionally much greater.
- Some omitted issues
 - Sign-extension
 - Computation width of the final CLA/RCA
 - Bit-pair recoding

2.4 DIVISION

2.4.1 Longhand Division

Steps

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

Example

- **Decimal**

$$\begin{array}{r} \text{Divisor} \rightarrow 13 \overline{) 274} \\ \underline{26} \\ 14 \\ \underline{13} \\ 1 \end{array} \begin{array}{l} \leftarrow \text{Quotient} \\ \leftarrow \text{Dividend} \\ \\ \\ \\ \leftarrow \text{Remainder} \end{array}$$

- **Binary**

$$\begin{array}{r} \text{Divisor} \rightarrow 1101 \overline{) 100010010} \\ \underline{1101} \\ 10000 \\ \underline{1101} \\ 1110 \\ \underline{1101} \\ 1 \end{array} \begin{array}{l} \leftarrow \text{Quotient} \\ \leftarrow \text{Dividend} \\ \\ \\ \\ \\ \leftarrow \text{Remainder} \end{array}$$

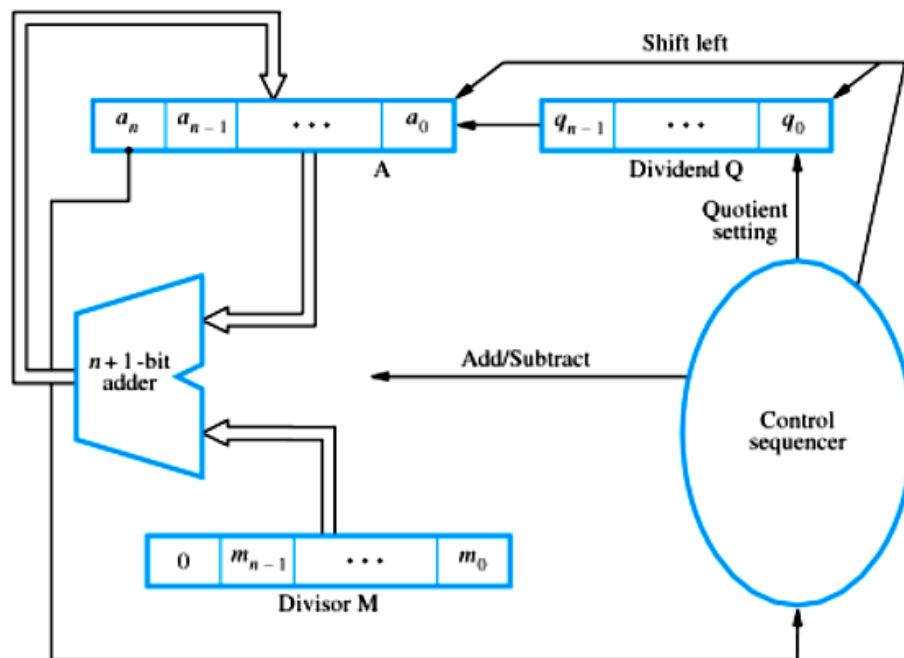
2.4.2 Restoring Division**Basic Operation**

- Initially
 - An n-bit positive-divisor is loaded into register M
 - An n-bit positive-dividend is loaded into register Q at the start of the operation
 - Register A is set to 0
- After division operation
 - An n-bit quotient is in register Q
 - Remainder is in register A

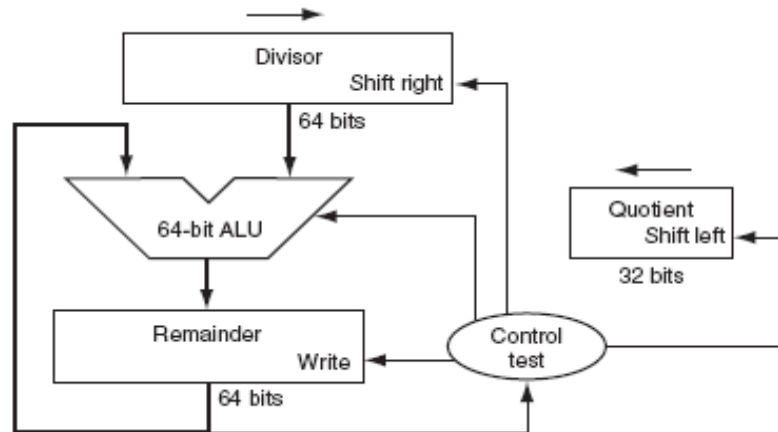
Steps in division operation

1. Shift A and Q left one binary position
2. Subtract M from A, and place the answer back in A
 - For subtraction, find 2's complement of M and add with A
 - $A - B = A + 2\text{'s complement}(B)$
3. If the sign of A is 1,
 - Set q_0 to 0 and add M back to A (restore A)
 Otherwise,
 - Set q_0 to 1
4. Repeat these steps n times

Logic Diagram



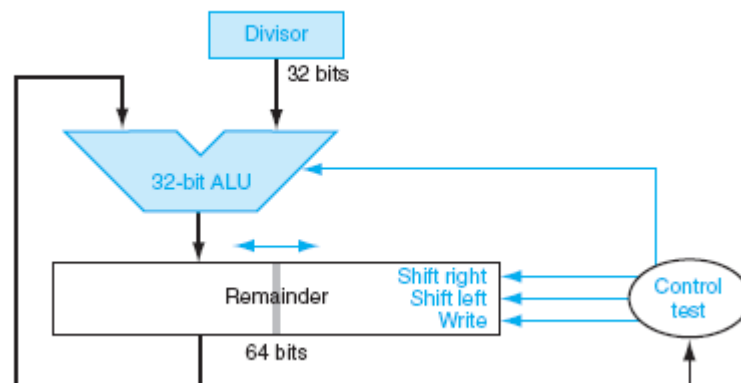
Implementation hardware



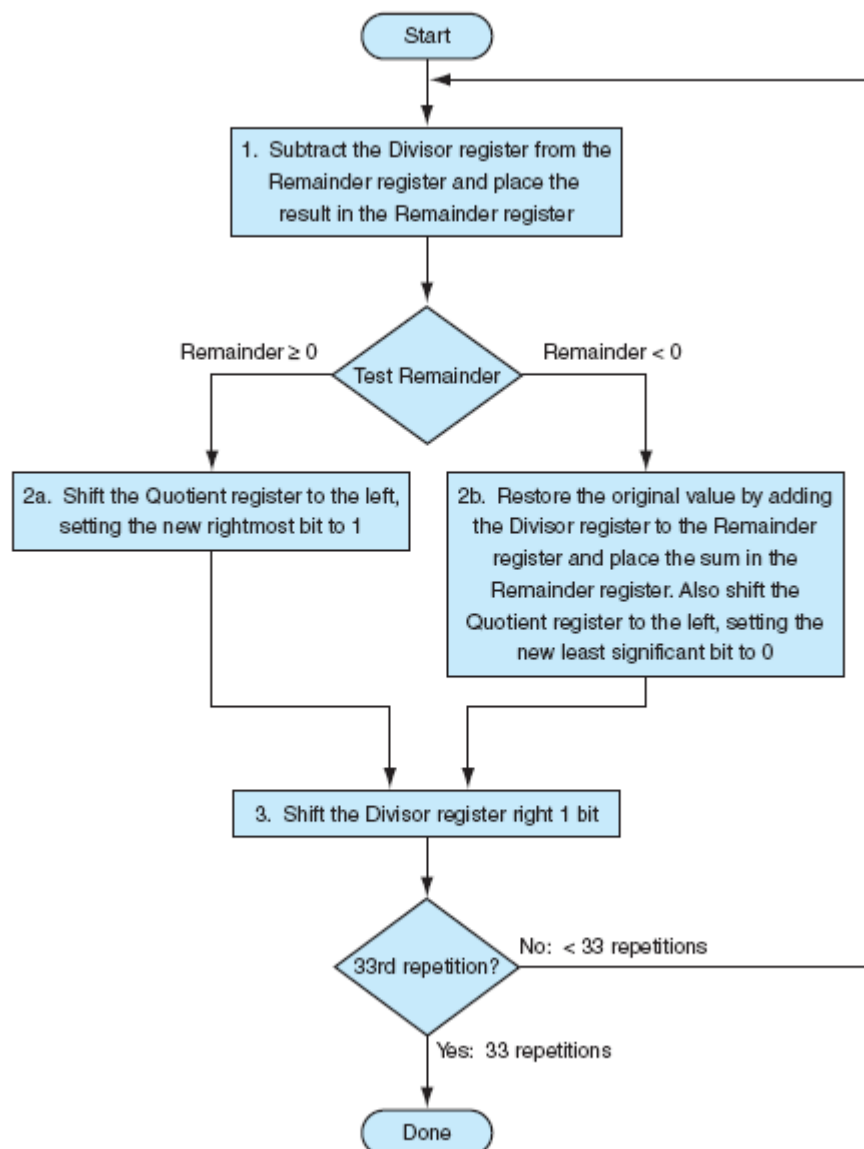
- The 32-bit Quotient register set to 0.
- Every iteration of the algorithm needs to move the divisor to the right one digit.
- We start with the divisor placed in the left half of the 64-bit Divisor register and shift it right 1 bit each step to align it with the dividend.
- The Remainder register is initialized with the dividend.
- The system first subtract the divisor in step 1
- If the result is positive, the divisor was smaller or equal to the dividend, so we generate a 1 in the quotient (step 2a).
- If the result is negative, the next step is to restore the original value by adding the divisor back to the remainder and generate a 0 in the quotient (step 2b).
- The divisor is shifted right and then we iterate again.
- The remainder and quotient will be found in their namesake registers after the iterations are complete.

Revised hardware

- The following figure shows the revised hardware for multiplication.
- The speedup comes from shifting the operands and the quotient simultaneously with the subtraction.
- This refinement halves the width of the adder and registers by noticing where there are unused portions of registers and adders.



Flowchart



Example

- Dividend (Q) : 1000
- Divisor (M) : 11

2's Complement of M

$$\begin{array}{rcl}
 \text{M:} & & \mathbf{0 \ 0 \ 0 \ 1 \ 1} \\
 \text{1's Complement:} & & 1 \ 1 \ 1 \ 0 \ 0 \\
 +1: & & 1 \ + \\
 \hline
 \text{2's Complement:} & & \underline{1 \ 1 \ 1 \ 0 \ 1}
 \end{array}$$

Longhand Division

$$\begin{array}{r}
 10 \\
 11 \overline{) 1000} \\
 \underline{11} \\
 10
 \end{array}$$

Restoring Division

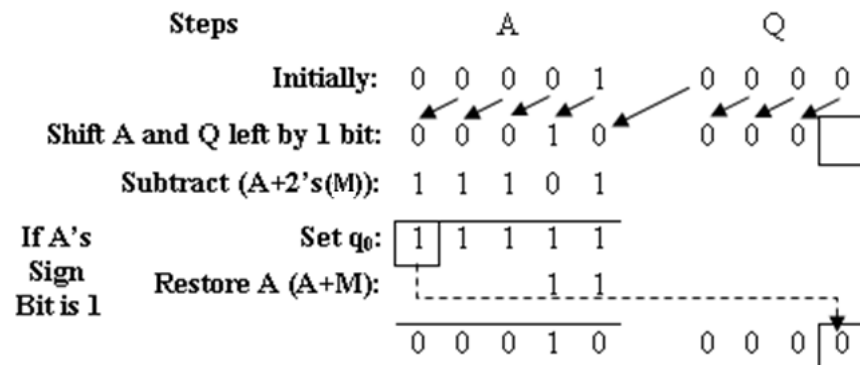
- **Initially**

$$\begin{array}{c}
 \text{A} \\
 \downarrow \\
 0 \ 0 \ 0 \ 0 \ 0 \\
 \text{M} \rightarrow 0 \ 0 \ 0 \ 1 \ 1
 \end{array}
 \qquad
 \begin{array}{c}
 \text{Q} \\
 \downarrow \\
 1 \ 0 \ 0 \ 0
 \end{array}$$

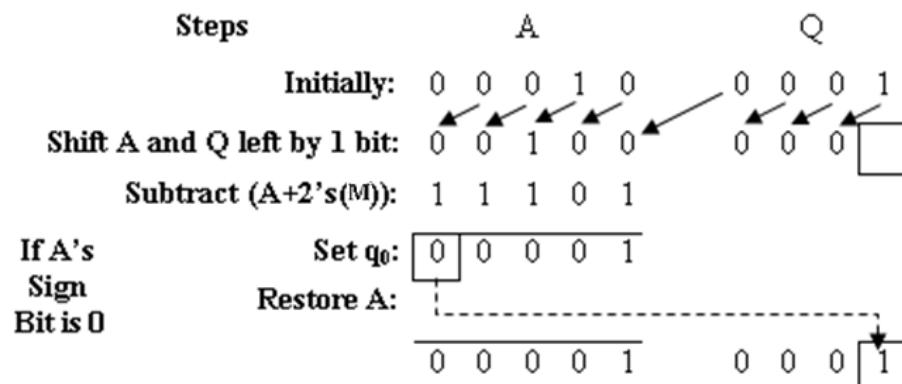
- **First Cycle**

Steps	A	Q
Initially:	0 0 0 0 0	1 0 0 0
Shift A and Q left by 1 bit:	0 0 0 0 1	0 0 0
Subtract (A+2's(M):	1 1 1 0 1	
If A's Sign Bit is 1	Set q_0 : 1 1 1 1 0	
Restore A (A+M):	<div style="display: flex; align-items: center;"> <div style="border-left: 1px dashed black; height: 10px; width: 10px; margin-right: 5px;"></div> <div style="border-bottom: 1px solid black; flex-grow: 1; position: relative;"> 1 1 </div> </div>	0 0 0 0 0

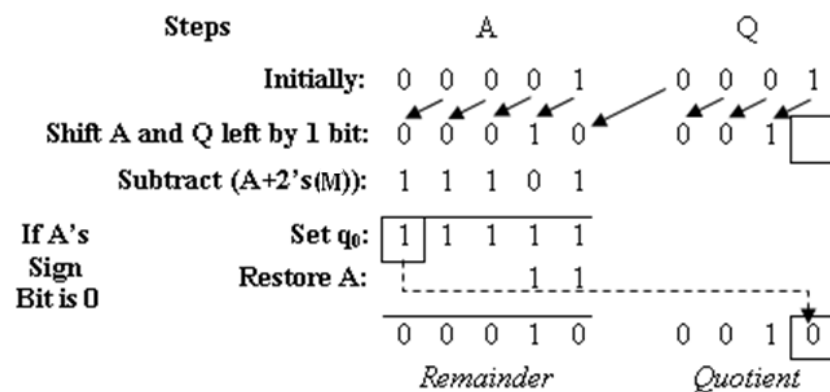
- Second Cycle



- Third Cycle



- Fourth Cycle



Illustration

2.4.3 Non-restoring Division

- Avoid the need for restoring A after an unsuccessful subtraction

Steps

1. Repeat n times
 - If the sign of A is 0,
 - Shift A and Q left one bit position and subtract M from A
 - Otherwise
 - Shift A and Q left and add M to A
 - Now, if the sign of A is 0, set q_0 to 1; otherwise, set q_0 to 0
2. If the sign of A is 1, add M to A

Example

- Dividend (Q) : 1000
- Divisor (M) : 11

2's Complement of M

$$\begin{array}{rcl}
 \text{M:} & & \mathbf{0 \ 0 \ 0 \ 1 \ 1} \\
 \text{1's Complement:} & & \mathbf{1 \ 1 \ 1 \ 0 \ 0} \\
 +1 : & & \mathbf{1 \ +} \\
 \hline
 \text{2's Complement:} & & \mathbf{1 \ 1 \ 1 \ 0 \ 1} \\
 \hline
 \end{array}$$

Longhand Division

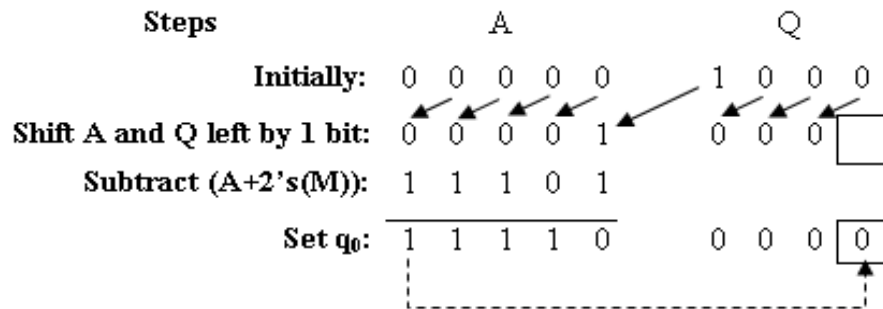
$$\begin{array}{r}
 10 \\
 11 \overline{) 1000} \\
 \underline{11} \\
 10
 \end{array}$$

Restoring Division

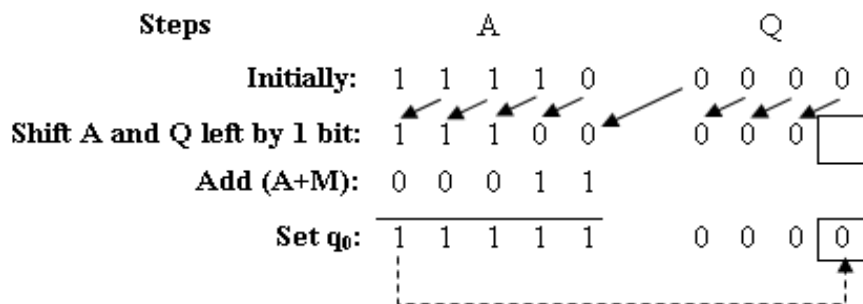
- **Initially**

$$\begin{array}{ccccccccc}
 & & & & \mathbf{A} & & & & \mathbf{Q} \\
 & & & & \downarrow & & & & \downarrow \\
 & 0 & 0 & 0 & 0 & 0 & & 1 & 0 & 0 & 0 \\
 \mathbf{M} \rightarrow & 0 & 0 & 0 & 1 & 1 & & & & &
 \end{array}$$

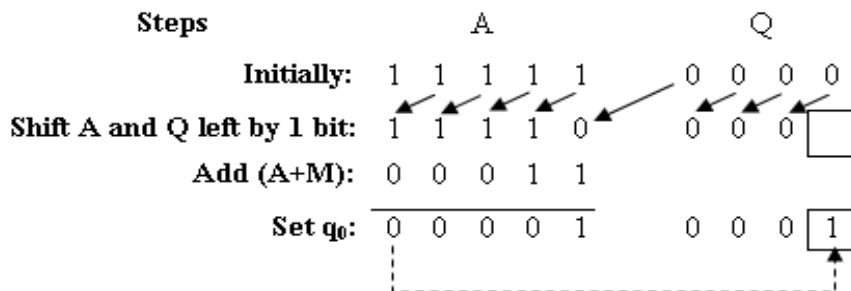
- First Cycle



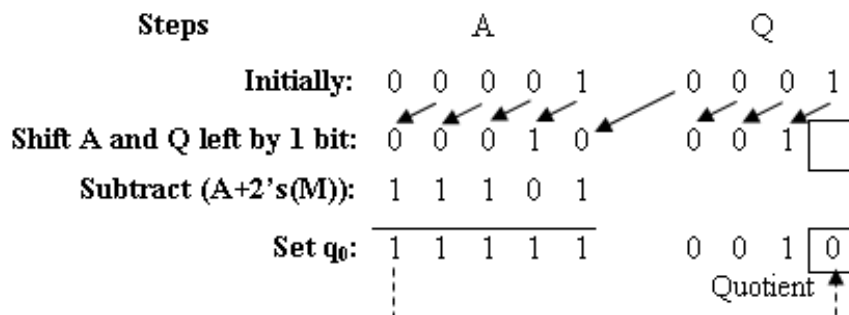
- Second Cycle



- Third Cycle

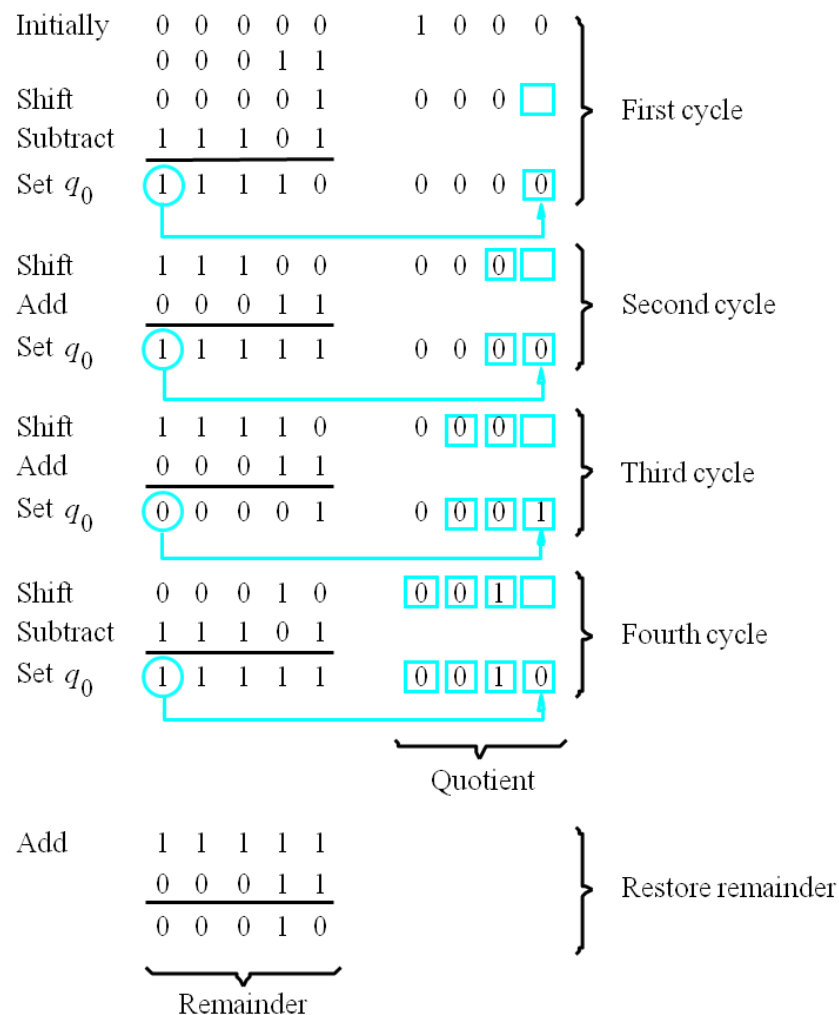


- Fourth Cycle



Restore Remainder

$$\begin{array}{r}
 \text{Add } 1 \ 1 \ 1 \ 1 \ 1 \\
 \text{M: } 0 \ 0 \ 0 \ 1 \ 1 \ + \\
 \hline
 0 \ 0 \ 0 \ 1 \ 0 \\
 \hline
 \text{Remainder}
 \end{array}$$

Illustration**2.5 FLOATING POINT REPRESENTATION****2.5.1 Floating Point Number**

A floating point is a computer arithmetic that represents numbers in which the binary point is not fixed.

The numerical value of a finite number can be represented by four integer components.

- Sign (s)
- Base (b)
- Significant or Fraction (m)
- Exponent (e)

Representation: $(-1)^s \times m \times b^e$

Example:

$$6.345 \times 10^{23}$$

$$-7.525 \times 10^{-12}$$

$$6.642 \times 10^{-32}$$

Terminologies

Scale factor

Scale factor indicates the position of the decimal point with respect to the significant digits.

For the above example, the scale factor is $10^{23}, 10^{-12}, 10^{-32}$

Fraction (m) is represented in the following format

$$i_m i_{m-1} \dots i_2 i_1 i_0. F_1 F_2 \dots F_{n-1} F_n$$

Where

i - integer parts

F - fraction parts

Overflow (floating-point)

Overflow is a situation in which a positive exponent becomes too large to fit in the exponent field.

Underflow (floating-point)

Underflow is a situation in which a negative exponent becomes too large to fit in the exponent field.

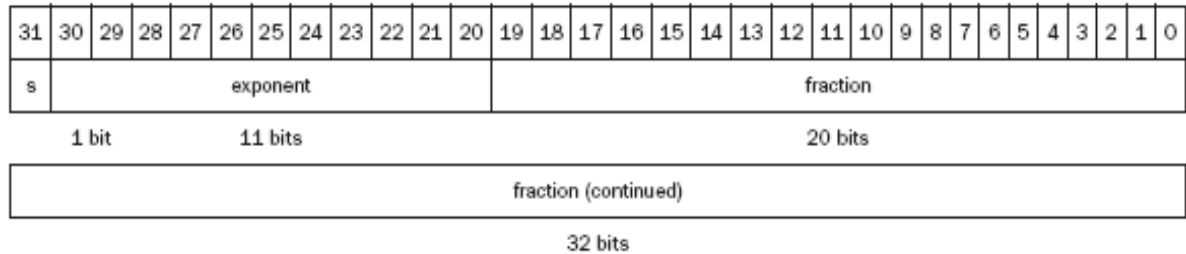
One way to reduce chances of underflow or overflow is to choose another format that has a larger exponent – double precision format.

Double precision

Double precision is a floating point value that is represented in two 32-bit words.

Single precision

Single precision is a floating point value which is represented in a single 32-bit word.



2.5.2 IEEE format for Single-Precision Floating-Point Numbers

In 32-bit single-precision floating-point representation:

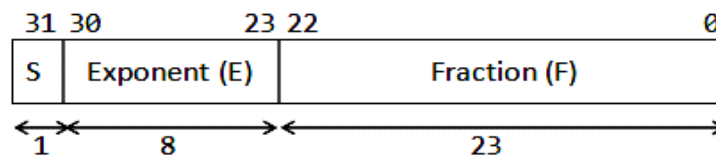
Most significant bit is the sign bit (S),

0 - Positive numbers

1 - Negative numbers

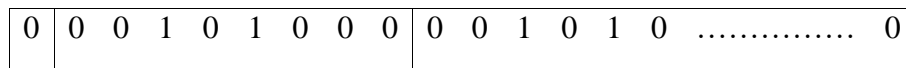
Next 8 bits represent exponent (E)

Remaining 23 bits represents fraction (F)



$$\text{Value represented} = \pm 1.F \times 2^{E-127}$$

Example



$$\text{Value represented} = \pm 1.001010 \dots 0 \times 2^{-87}$$

2.5.3 IEEE format for Double-Precision Floating-Point Numbers

In 64-bit double-precision floating-point representation:

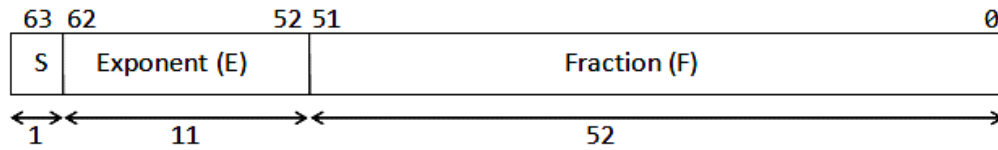
Most significant bit is the sign bit (S),

0 - Positive numbers

1 - Negative numbers

Next 11 bits represent exponent (E)

Remaining 52 bits represents fraction (F)



$$\text{Value represented} = \pm 1.F \times 2^{E-1023}$$

2.5.4 Special Values

- End values in E : 0 and 255
 - Used to represent special values
- Different Special Values

E Value	F Value	S Value	Represented Value
0	0	0	+0
		1	-0
255	0	0	$+\infty$
		1	$-\infty$
0	$\neq 0$		Denormal $\pm 0.F \times 2^{-126}$
255	$\neq 0$		NaN (Not a Number)

2.5.5 Exceptions

- In IEEE standard, processor set exception flags when the exception occurs
- Types of Exception
 - Underflow
 - Occurs when an number requires an exponent less than -126 (for single precision) or -1022(for double precision) to represent it in normalized form
 - Overflow

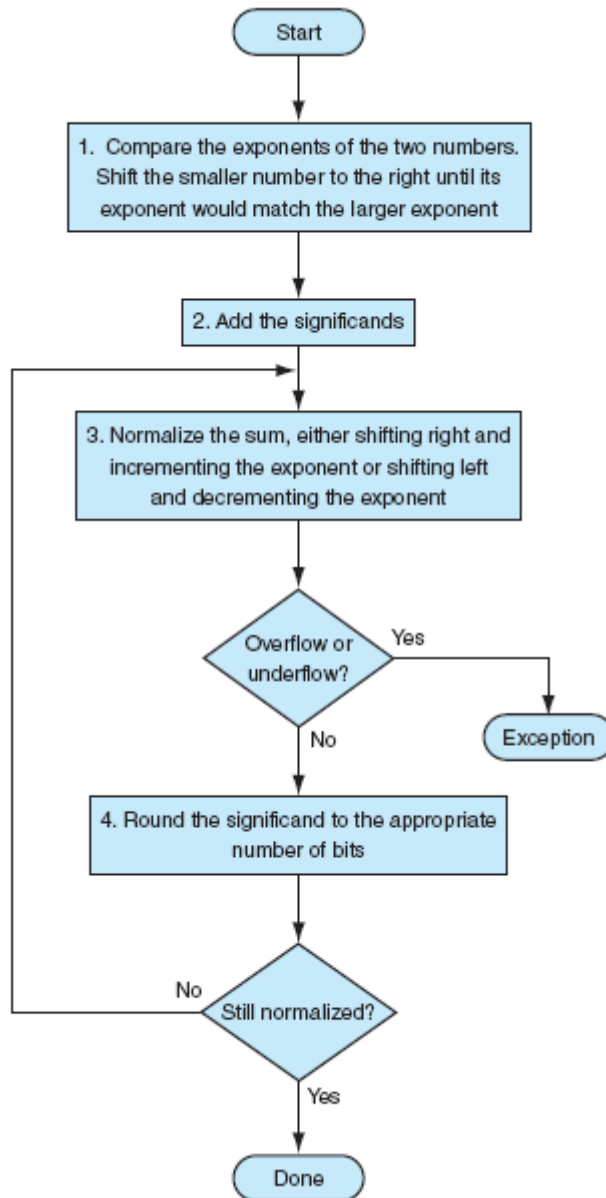
- Occurs when an number requires an exponent greater than +127 (for single precision) or +1023(for double precision) to represent it in normalized form
- Divide by Zero
 - Occurs when any number is divided by zero
- Inexact
 - Occurs when any result requires rounding in order to be represented in one of the normal formats
- Invalid
 - Occurs when the operations such as $\frac{0}{0}$ or $\sqrt{-1}$ are attempted

2.6 FLOATING POINT OPERATIONS

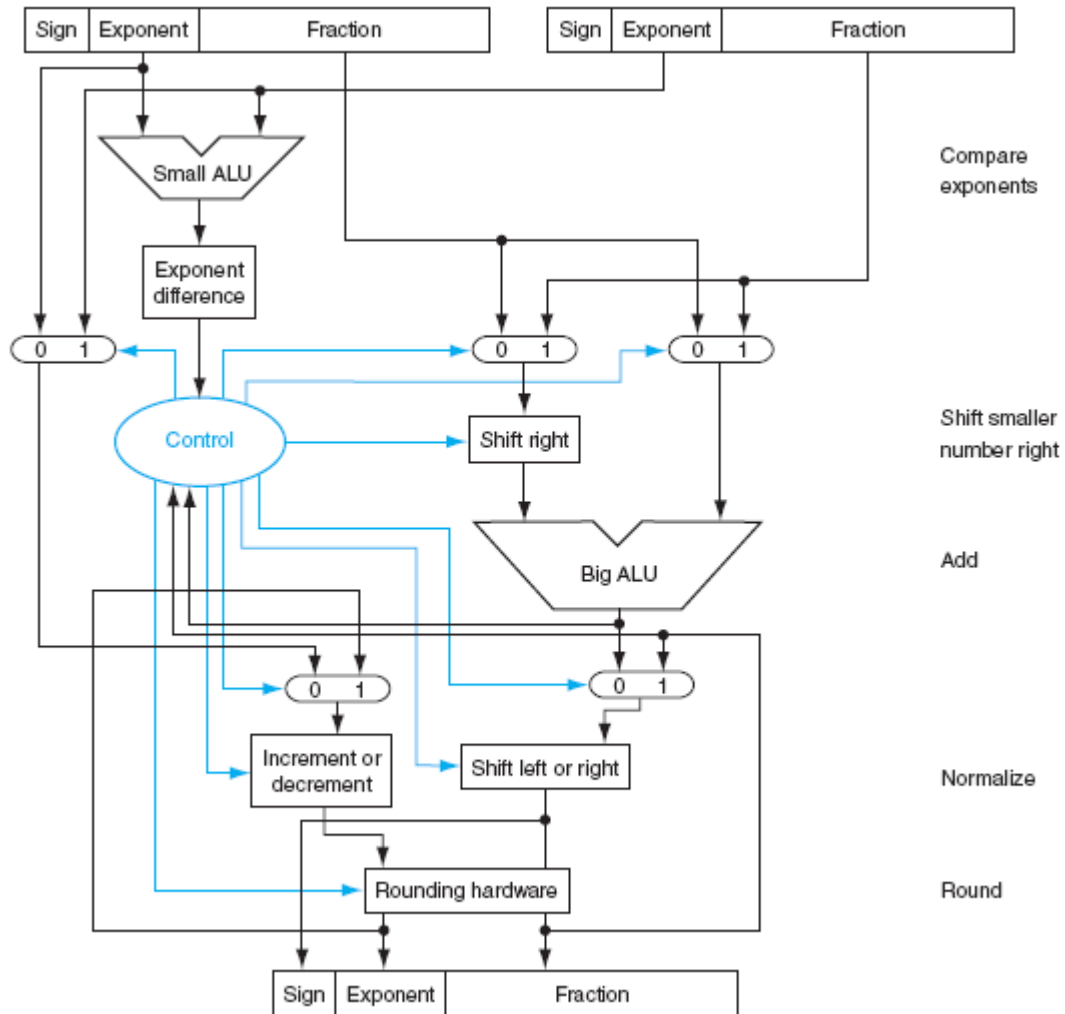
2.6.1 Floating-Point Addition

- First, convert the two representations to scientific notation. Thus, we explicitly represent the hidden 1.
- In order to add, we need the exponents of the two numbers to be the same.
- This is done by rewriting Y. This will result in Y being not normalized, but value is equivalent to the normalized Y.
- Add x - y to Y's exponent. Shift the radix point of the mantissa (significant) Y left by x - y to compensate for the change in exponent.
- Add the two mantissas of X and the adjusted Y together.
- If the sum in the previous step does not have a single bit of value 1, left of the radix point, and then adjust the radix point and exponent until it does.
- Convert back to the one byte floating point representation.

Flowchart

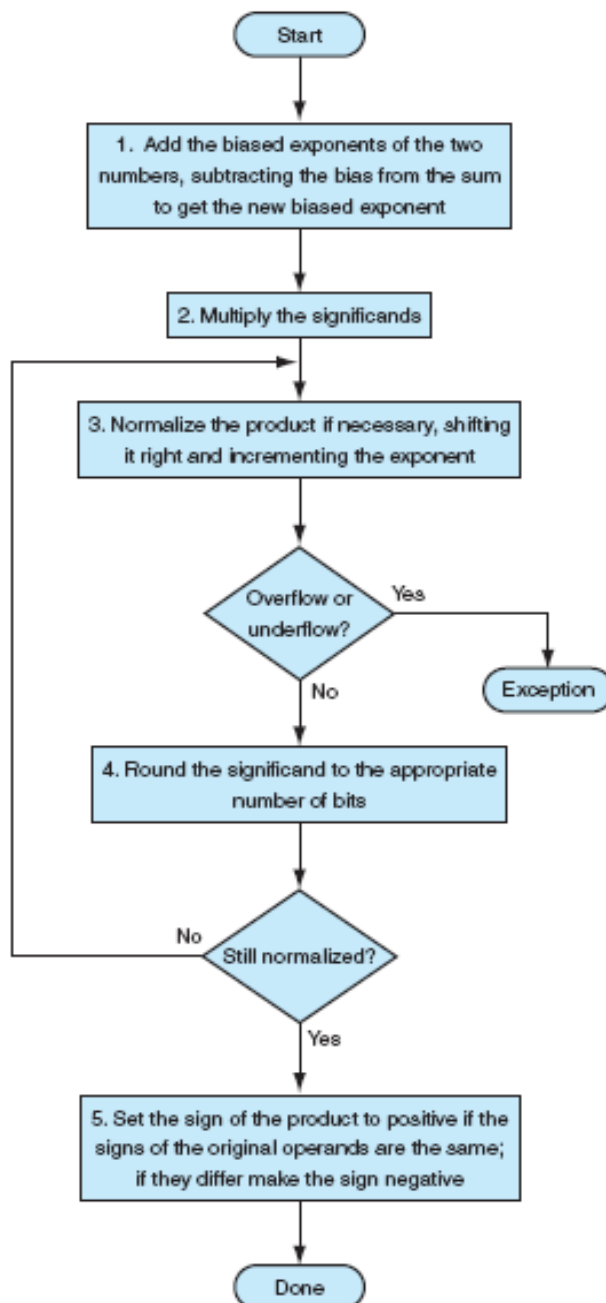


Hardware Implementation



2.6.2 Floating point Multiplication

- First, convert the two representations to scientific notation. Thus, we explicitly represent the hidden 1.
- Let x be the exponent of X . Let y be the exponent of Y . The resulting exponent (call it z) is the sum of the two exponents. z may need to be adjusted after the next step.
- Multiply the mantissa of X to the mantissa of Y . Call this result m .
- If m does not have a single 1 left of the radix point, and then adjust the radix point so it does, and adjust the exponent z to compensate.
- Add the sign bits, mod 2, to get the sign of the resulting multiplication.
- Convert back to the one byte floating point representation, truncating bits if needed.

Flowchart**2.7 ACCURATE ARITHMETIC**

- Unlike integers, which can represent exactly every number between the smallest and largest number, floating-point numbers are normally approximations for a number they can't really represent.

- IEEE 754 has two extra bits on the right during intermediate additions, called guard and round, respectively
- **Guard** - The first of two extra bits kept on the right during intermediate calculations of floating point numbers; 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.
- IEEE 754 has four rounding modes: always round up (toward $+\infty$), always round down (toward $-\infty$), truncate, and round to nearest even
- **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

2.8 SUBWORD PARALLELISM

- A Subword is a lower precision unit of data contained within a word.
- In subword parallelism, multiple subwords are packed into a word and then process whole words.
- With the appropriate subword boundaries, this technique results in parallel processing of subwords.
- Since the same instruction is applied to all subwords within the word - form of SIMD(Single Instruction Multiple Data) processing.
- Possible to apply subword parallelism to noncontiguous subwords of different sizes within a word
- Practical implementation is simple if subwords are same size and they are contiguous within a word
- The data parallel programs that benefit from subword parallelism tend to process data that are of the same size
- Also called data-level parallelism, vector parallelism, or Single Instruction, Multiple Data (SIMD)
- Subword parallelism is an efficient and flexible solution for media processing because algorithm exhibit a great deal of data parallelism on lower precision data.
- Useful for computations unrelated to multimedia that exhibit data parallelism on lower precision data

Example

Basic idea

Treat a 64-bit register as a vector of 2 32-bit or 4 16-bit or 8 8-bit values (short vectors)

Partition 64-bit datapaths to handle multiple narrow operations in parallel

- Graphics and audio applications can take advantage of performing simultaneous operations on short vectors
 - Example: 128-bit adder:
 - Sixteen 8-bit adds
 - Eight 16-bit adds
 - Four 32-bit adds

These are also called as data-level parallelism, vector parallelism, or Single Instruction, Multiple Data (SIMD).

UNIT 3

PROCESSOR AND CONTROL UNIT

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

3.1 INTRODUCTION

- The CPU performs all the operations that are required by the system for its smooth functioning.
- It works on stored program concept.
- The control unit issues control signals and provide direction to the compiler.
- The operation or task performed by the CPU to perform instruction execution are:
 - **Fetch Instruction:** The CPU reads an instruction from memory.
 - **Interpret/Decode Instruction:** The instruction is decoded to determine what action is required.
 - **Fetch Data:** The execution of an instruction may require reading data from memory or I/O module.
 - **Execute/ Process Instruction:** The execution of an instruction may require performing some arithmetic or logical operation on data.
 - **Write data:** The result of an execution may require writing data to memory or an I/O module.
- The major components of the CPU are an arithmetic and logic unit (ALU) and a control unit (CU).
- The ALU does the actual computation or processing of data.
- The CU controls the movement of data and instruction into and out of the CPU and controls the operation of the ALU.

Basic terminologies

- **Response time/ Execution time** is 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** is the total amount of work done in a given time.
- The **user CPU time** is the CPU time spent in a program itself.
- The **system CPU time** is the CPU time spent in the operating system performing tasks on behalf of the program.
- The **clock cycle**, also called tick/ clock tick/ clock period/ clock/ cycle is the time for one clock period, usually of the processor clock, which runs at a constant rate.
- The **clock period** is the length of each clock cycle.

Performance and Execution Time

- The response time of a CPU should decrease in order to increase the performance. This is also accomplished by increasing the throughput.
- The relationship between the performance and execution time of a computer, X is given by

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

- Consider two computers X and Y. If the performance of X is greater than the performance of Y, then

$$\text{Performance}_x > \text{Performance}_y$$

$$\frac{1}{\text{Execution time}_x} > \frac{1}{\text{Execution time}_y}$$

$$\text{Execution time}_x < \text{Execution time}_y$$

- The execution time on Y is longer than that on X, if X is faster than Y.

$$\frac{\text{Performance}_x}{\text{Performance}_y} = \eta$$

- If X is n times faster than Y, then the execution time on Y is n times longer than it is on X.

$$\frac{\text{Performance}_x}{\text{Performance}_y} = \frac{\text{Execution time}_y}{\text{Execution time}_x} \eta$$

- The performance of the CPU is measured by the following factors
 - Instruction count
 - Determined by Instruction Set Architecture (ISA) and the compiler
 - Cycles per Instruction (CPI) and Clock Cycle time
 - Determined by CPU hardware
- The basic performance equation is given as

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

(or)

$$\text{CPU time} = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

3.1.1 MIPS

- MIPS stands for Million Instructions Per Second (MIPS)
- Simplest metrics used to measure CPU performance
- MIPS is computed as the instruction count divided by the product of the execution time and 10^6 .

$$\text{MIPS} = \frac{\text{Instruction Count}}{\text{Execution Time} \times 10^6}$$

3.1.2 Types of Instruction

The MIPS instruction set includes the following

- Memory-reference instructions
 - load word (lw) and store word (sw)
- Arithmetic and logical instructions
 - add, sub, and, or, and slt
- Control Flow Instructions
 - Branch equal (beq) instructions
 - Jump (j) instruction

3.1.3 MIPS Instruction Execution

MIPS instructions classically take five steps:

1. Fetch instruction from memory

2. Read registers while decoding the instruction. The format of MIPS instructions allows reading and decoding to occur simultaneously
3. Execute the operation or calculate an address
4. Access an operand in data memory
5. Write the result into a register

3.1.4 MIPS Instruction Formats

- **R - Format**

Field	opcode	rs	rt	rd	shamt	funct
Bit Positions	31-26	25-21	20-16	15-11	10-6	5-0

- Opcode = 0
- Three register operands: rs, rt, and rd
 - rs and rt - sources
 - rd - destination
- shamt field - used only for shifts
- funct field - The ALU function (add, sub, and, or, and slt) and is decoded by the ALU control design

ALU control lines	Function
000	AND
001	OR
010	Add
110	Subtract
111	Set on less than

- **I - Format**

Field	opcode	rs	rt	address
Bit Positions	31-26	25-21	20-16	15-0

- For load and store instructions
 - Opcode = 35(for load) and Opcode = 43(for store)

- rs - the base register
- rt is
 - For loads, the destination register for the loaded value
 - For stores, the source register whose value should be stored into memory
- The memory address is computed as

$$\text{Memory address} = \text{base register} + 16\text{-bit address field}$$

- For branch instructions

- Opcode = 4
- rs and rt are the source registers that are compared for equality
- The branch target address is computed as

$$\text{Target address} = \text{PC} + (\text{signed-extended 16-bit offset address} \ll 2)$$

- **J – Format**

Field	opcode	address
Bit Positions	31-26	25-0

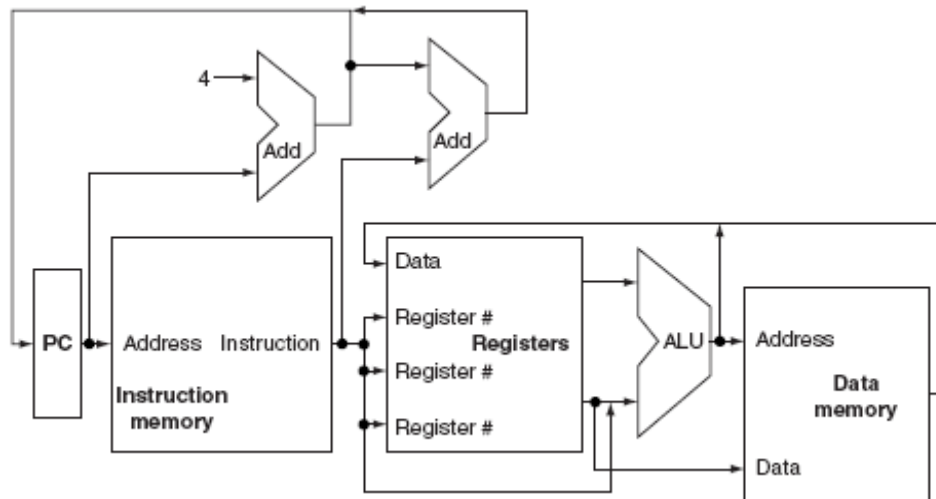
- Opcode = 2
- The destination address is computed as

$$\text{Target address} = \text{PC}[31-28] \parallel (\text{offset address} \ll 2)$$

3.2 BASIC MIPS IMPLEMENTATION

Overview of the CPU

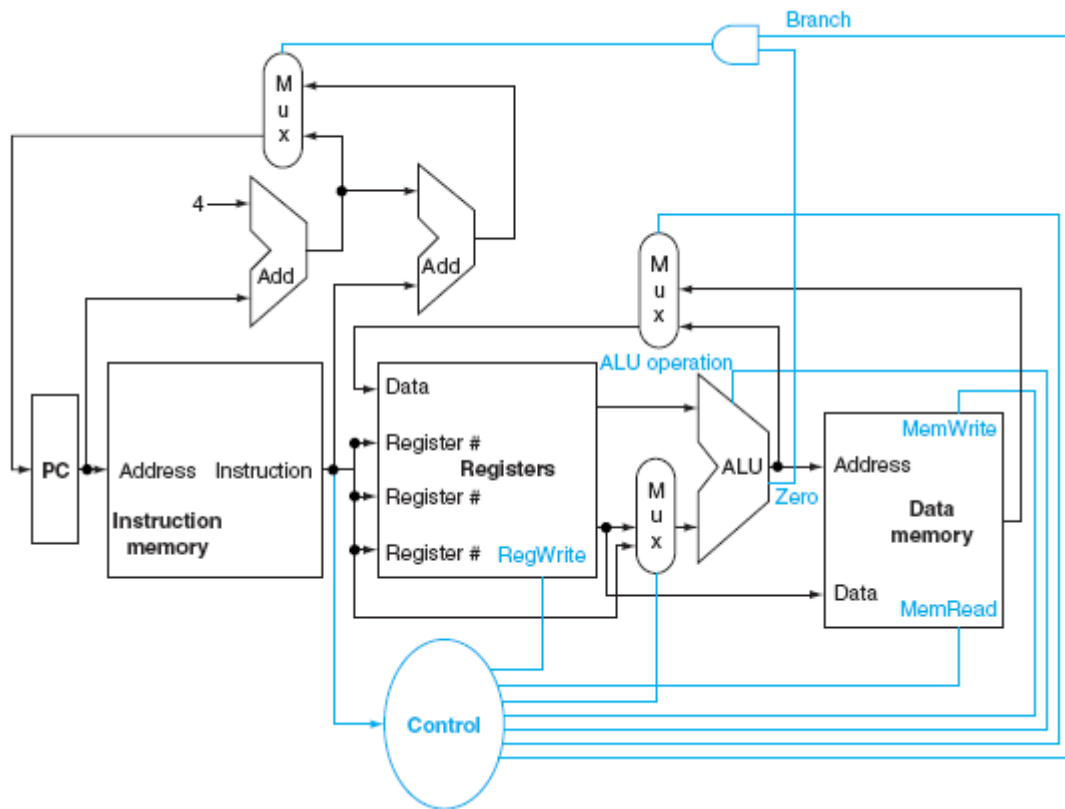
- The high-level view of MIPS implementation with the major functional units and the major connections between them is shown in the following diagram.
- A control unit has the instruction as an input.
- It is used to determine how to set the control lines for the functional units and two of the multiplexors.
- The third multiplexor, which determines whether $\text{PC} + 4$ or the branch destination address is written into the PC, is set based on the zero output of the ALU, which is used to perform the comparison of a BEQ instruction.



MIPS subset

- Program counter supplies the instruction address to the instruction memory.
- The instructions are fetched from the instruction memory.
- After the instruction is fetched, the register operands are fetched from the instruction fields.
- Once the register operands have been fetched, the operands are operated to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or a compare (for a branch).
- If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register.
- If the operation is 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 operation is branch,
 - the ALU output is used to determine the next instruction address, which comes from either the ALU (where the PC and branch offset are summed) or from an adder that increments the current PC by 4.
- The thick lines interconnecting the functional units represent buses, which consist of multiple signals.
- The arrows are used to guide the reader in knowing how information flows.

- Since signal lines may cross, we explicitly show when crossing lines are connected by the presence of a dot where the lines cross.



Implementation of MIPS

- The circuit consists of two multiplexor controls – topmost and bottommost.
 - The top one 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 multiplexor whose output returns to the register file is used to navigate the output of the ALU or the output of the data memory for writing into the register file.
 - The bottommost multiplexor is used to determine whether the second ALU input is from the registers or from the offset field of the instruction.
- The added control lines determine the operation performed at the ALU.
- They predict whether the data memory should read or write, and whether the registers should perform a write operation.

Finally, the single-cycle datapath must have separate instruction and data memories because

- the format of data and instructions is different in MIPS and hence different memories are needed
- having separate memories is less expensive
- the processor operates in one cycle and cannot use a single-ported memory for two different accesses within that cycle

3.3 BUILDING DATA PATH

3.3.1 Datapath

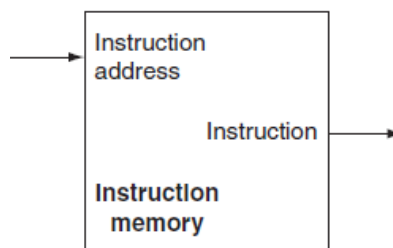
- The datapath is the pathway that the data takes through the CPU.
- As the data travels through the datapath, the control unit regulates interaction between the datapath and the data according to the instruction being executed.
- The datapath consists of functional units that perform data processing operations such as addition, subtraction, logical AND, OR, inverting, and shifting.

3.3.2 Datapath Elements

- A datapath element is a functional unit used to operate on or hold data within a processor.
- The datapath elements are
 - the instruction memory
 - the data memory
 - the register file
 - the arithmetic logic unit (ALU)
 - adders

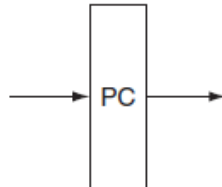
Instruction Memory

- A memory unit that is used to store the instructions of a program and supply instructions given an address

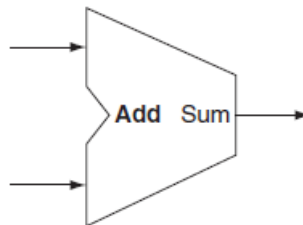


Program Counter (PC)

- PC is used to hold the address of the instruction in the program being executed.
- It is a 32-bit register.
- PC will be written at the end of every clock cycle and thus does not need a write control signal.

**Adder**

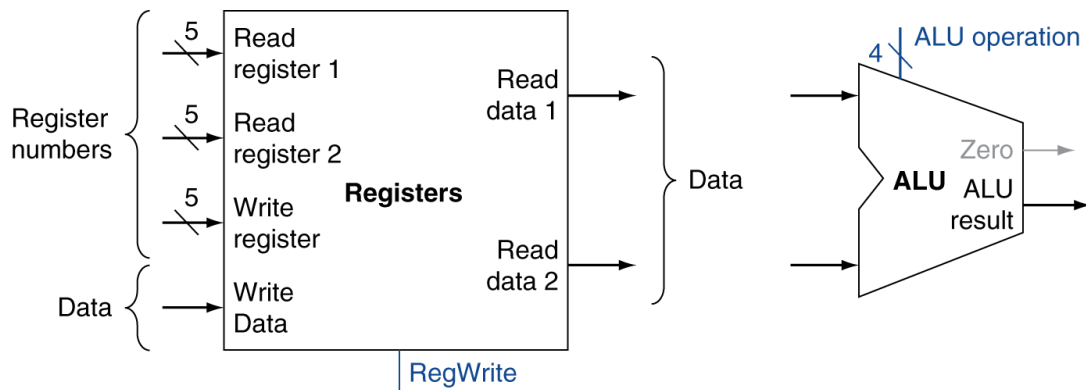
- Adders increment the PC to enable it to point the address of the next instruction.
- An ALU performs the addition of its two 32-bit inputs and place the result on its output.

**Registers**

- Registers are the data storage locations available in the processor.
- Register file is a structure that stores the processor's 32 general-purpose registers.
- A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed
- It holds a total of four inputs (three for register numbers and one for data).
 - two read ports - read two data words from the register file
 - One write port - write one data word into the register file
 - One write Data
- It possesses two outputs (both for data)
 - Two read data that carry the value that has been read from the registers
 - Register number inputs are 5 bits wide to specify one of 32 registers ($32 = 2^5$)
 - Data input and two data output buses are each 32 bits wide

ALU

- Takes two 32-bit inputs
- Produces a 32-bit result, as well as a 1-bit signal if the result is 0
- ALU operation signal controls the operation to be performed by the ALU
- 4 bits wide

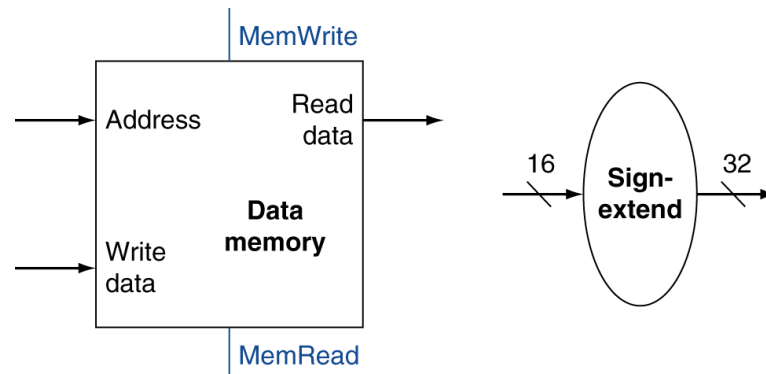


a. Registers

b. ALU

Data Memory unit

- It is a state element with two inputs for the address and the write data.
- A single output for the read result contains separate read and write controls.
- The sign-extension unit takes a 16-bit input that is sign-extended into a 32-bit result.
- The sign-extend is used to increase the size of a data item by replicating the high-order sign bit of the original data item in the highorder bits of the larger, destination data item.



a. Data memory unit

b. Sign extension unit

Mux/Multiplexer

- It is also called as data selector.
- It allows multiple connections to the input of an element and have a control signal select among the inputs.

Branch Instructions

- The beq instruction has three operands, two registers that are compared for equality, and a 16-bit offset used to compute the branch target address relative to the branch instruction address.
- The Branch target address is the address specified in a branch, which becomes the new program counter (PC) if the branch is taken.
- The branch target is given by the sum of the offset field of the instruction and the address of the instruction following the branch.
 - Format: beq R1,R2,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.
- It is necessary to determine whether the next instruction is the instruction that follows sequentially or the instruction at the branch target address.
- When the condition is true (i.e., the operands are equal), the branch target address becomes the new PC, called as the branch taken.
- **Branch taken** is a branch where the branch condition is satisfied and the program counter (PC) becomes the branch target. All unconditional branches are taken branches.
- If the operands are not equal, the incremented PC should replace the current PC (just as for any other normal instruction), which is stated as the branch not taken.
- **Branch not taken** is a branch where the branch condition is false and the program counter (PC) becomes the address of the instruction that sequentially follows the branch.

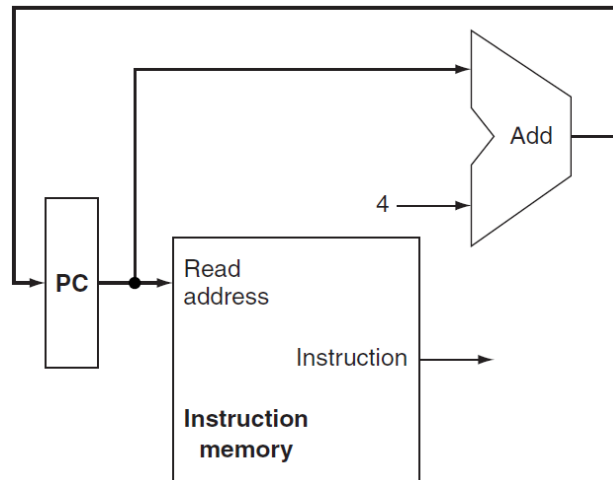
Delayed Branch

- Branches are delayed if the instruction immediately following the branch is always executed, independent of whether the branch condition is true or false.
- When the condition is false, the execution looks like a normal branch.
- When the condition is true, a delayed branch first executes the instruction immediately following the branch in sequential instruction order before jumping to the specified branch target address.

3.3.3 Building a Datapath

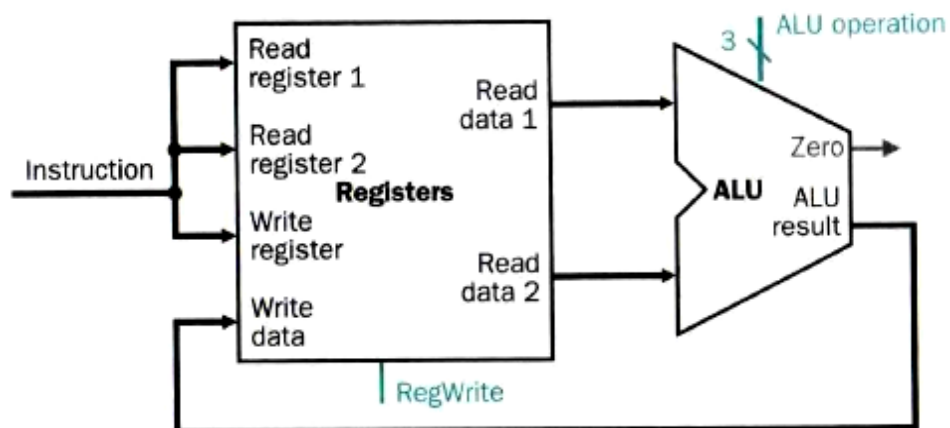
3.3.3.1 Fetching instructions

- To execute any instruction, the instructions have to be fetched from the memory.
- To prepare for executing the next instruction, the program counter is incremented by 4 bytes that points the next instruction to be executed.



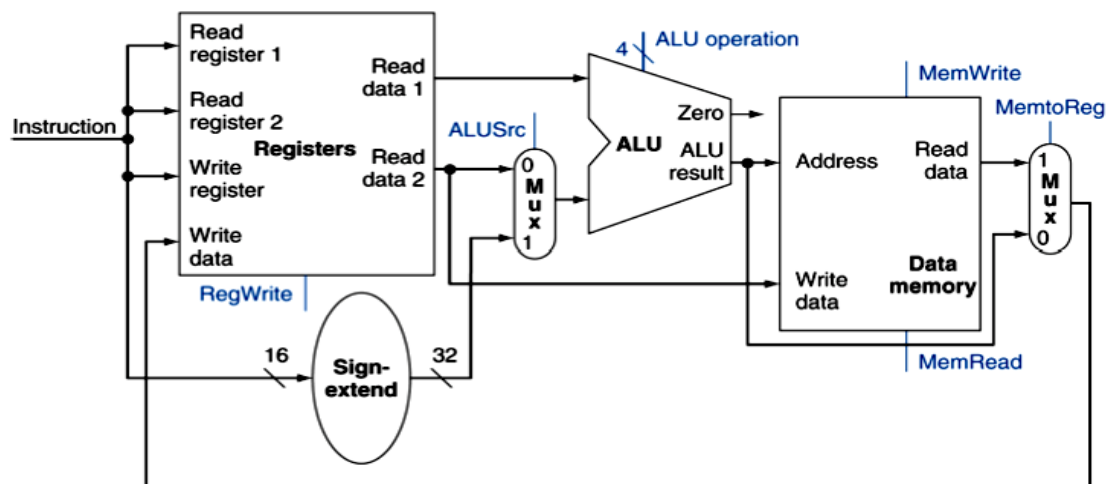
3.3.3.2 Datapath for R-type instructions

- The following additional components are needed for the implementation of the datapath for R-format instructions.
 - Register file
 - ALU
- The ALU accepts the input from the DataRead ports of the register file.
- The register file is written by the ALU in combination with the RegWrite signal.



3.3.3.3 Datapath for Load/Store instruction

- The following additional components are added to build the datapath for load and store instruction.
 - Data Memory unit
 - Sign Extension unit



- The register number inputs are read from the instruction field.
- Memory address is calculated based on the operands in the instruction field.
- For load instructions, the data at the memory address is read from data memory.
- For store instructions, the write data is written into the data memory at the memory address.

3.3.3.4 Datapath for branch instruction

Basic Operation

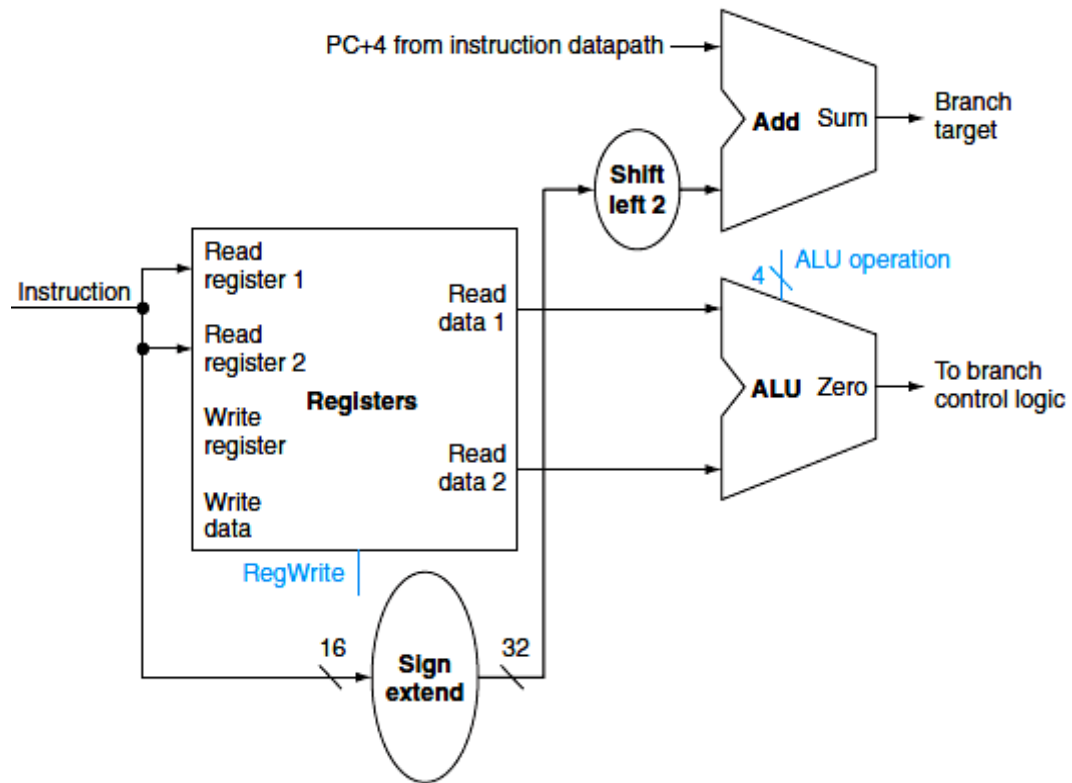
- Compute the branch target address.
 - Adder is used to compute the branch target address.
 - It is computed as,

Branch Target = Incremented PC + sign-extended,

Lower 16 bits of the instruction and,

Shifted left 2 bits

- Compare the register contents.
 - The ALU evaluates the branch condition.



3.3.4 Simple Implementation Scheme

3.3.4.1 Creating a single datapath

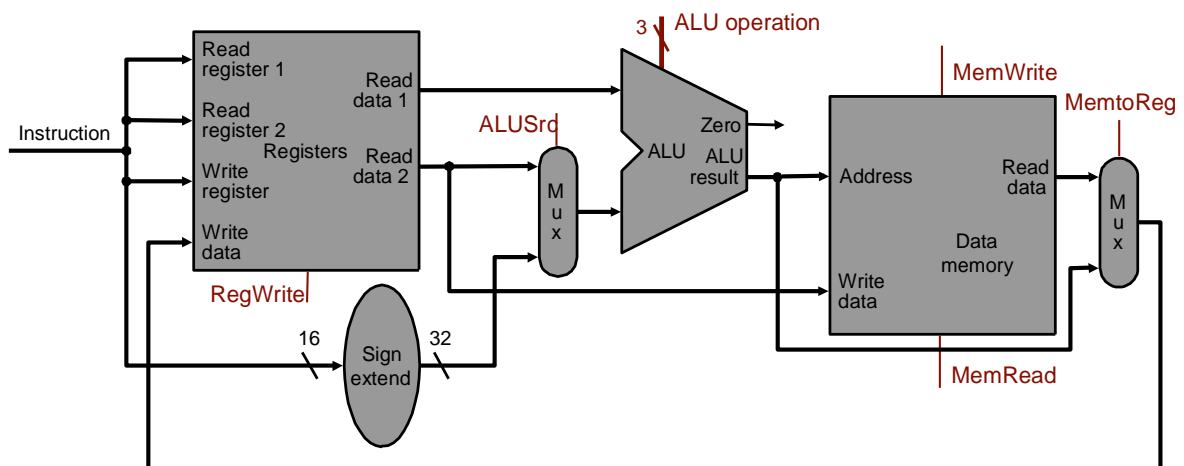
- Executing all instructions in a single clock cycle
 - No datapath resource can be used more than once in an instruction
 - Any element needed more than once must be duplicated
 - Instruction memory must be separated from data (separate datapath resources)
- Sharing datapath elements between different instruction classes
 - Need multiple connections to the input of an element, selected by a control signal
 - Commonly achieved by a multiplexor

3.3.4.2 Datapath for Arithmetic-logical (R-type) and Memory access (load/store)

- The main differences between the Arithmetic/Logical and Load/Store instruction execution are

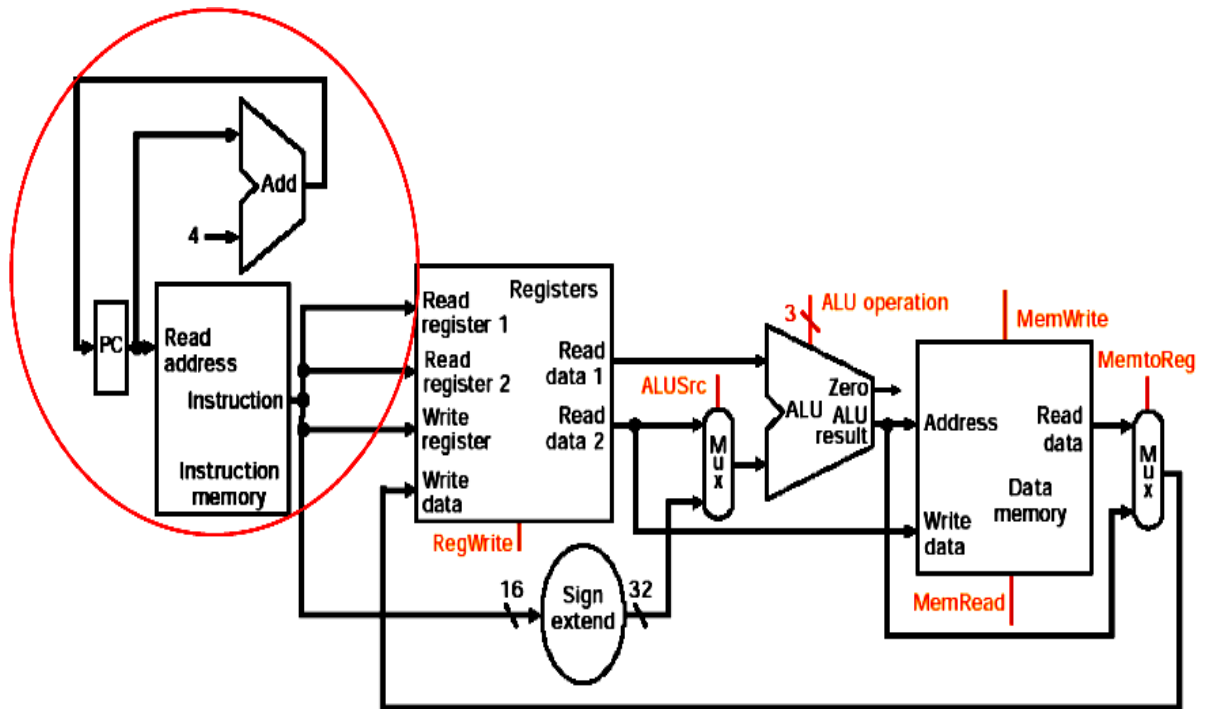
Feature	Arithmetic/Logical	Load/Store
Second operand input to ALU	Register contents for R-type	Sign-extended immediate value (offset)
Value stored in destination register	ALU output for R-type	Data memory value

- For combining the two datapaths
 - Select source of second ALU operand
 - Select source of data to write to register
- Use 2 MUXes with control inputs
 - ALUSrc for ALU
 - MemtoReg for register write



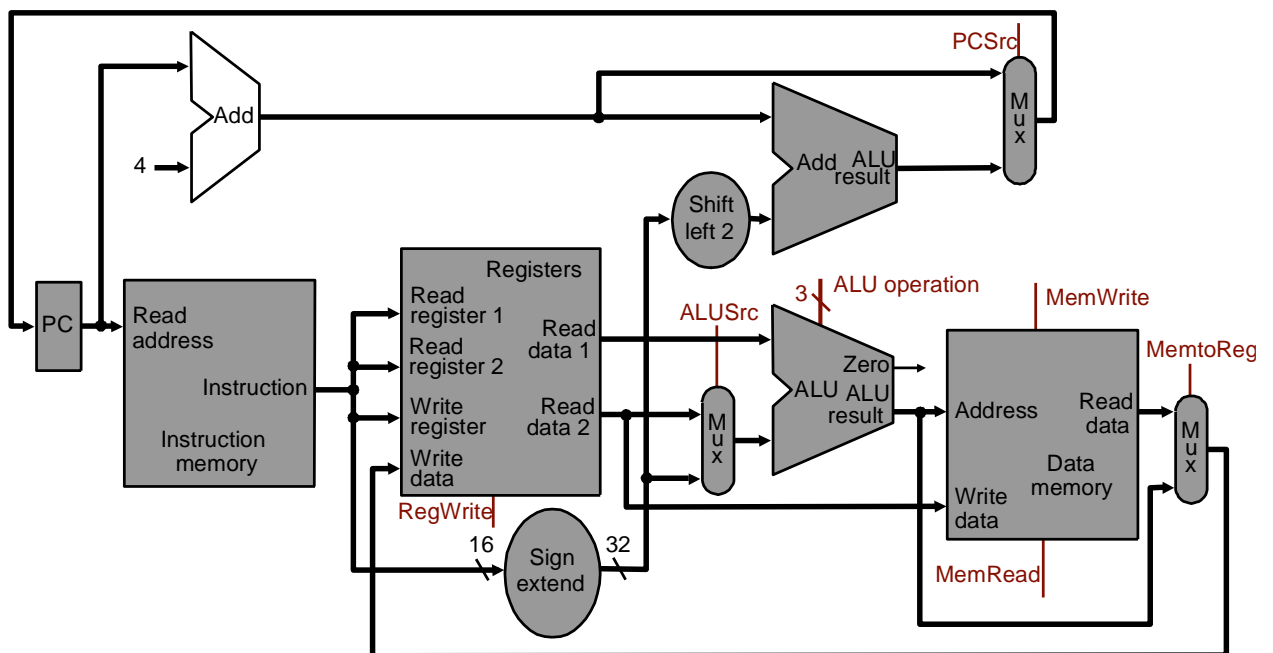
3.3.4.3 Combining datapaths: instruction fetch

- Need separate adder to
 - increment PC
 - perform ALU operation in same clock cycle



3.3.4.4 Combining datapaths: branch

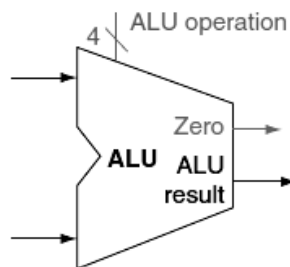
- Datapath can execute the following basic instructions in a single clock cycle
 - Load/store word
 - ALU operations
 - Branches



- An additional multiplexor(MUX) is needed to integrate branches
 - Uses PCSrc input to select:
 - incremented PC or (PC + immediate) from second adder
 - output goes to update PC
- Need to keep separate adder to compute branch address

3.3.4.5 The ALU Control

- Three control inputs (bits) - one for bnegate and two for operation



- Only five of possible eight combinations are used
- Depending on instruction class, ALU has to perform one of these five functions.

ALU Control Lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	Set on less than
1100	NOR

- For load and store word, ALU computes memory address by addition
- For R-type instructions, ALU does one of the five actions based on the value of 6-bit funct field in low-order bits of opcode
- For branch, ALU performs a subtraction
- Control unit for the 3-bit ALU control input
 - Input from funct field of opcode and a 2-bit control field called ALUOP
 - ALUOP indicates operation

Bits	Operation
00	Add for load and store
01	Subtract for beq
10	Determined by operation encoded in funct field

- Output of ALU control is a 3-bit signal (one of five combinations) to control the ALU

3.3.4.6 Setting of ALU control bits

- Setting ALU control inputs based on 2-bit ALUOP control and 6-bit funct field
 - Opcode determines the setting of the ALUOp bits
 - All the encodings are shown in binary
 - When the ALUOp code is 00 or 01, the desired ALU action does not depend on the function code field
 - The function code value is “don’t care” which is shown as xxxxxx
 - When the ALUOp value is 10, then the function code is used to set the ALU control input

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	xxxxxx	add	010
SW	00	store word	xxxxxx	add	010
Branch Equal	01	branch equal	xxxxxx	subtract	110
R-type	10	Add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	Set on less than	101010	set on less than	111

3.3.4.7 Truth table for the three ALU control bits

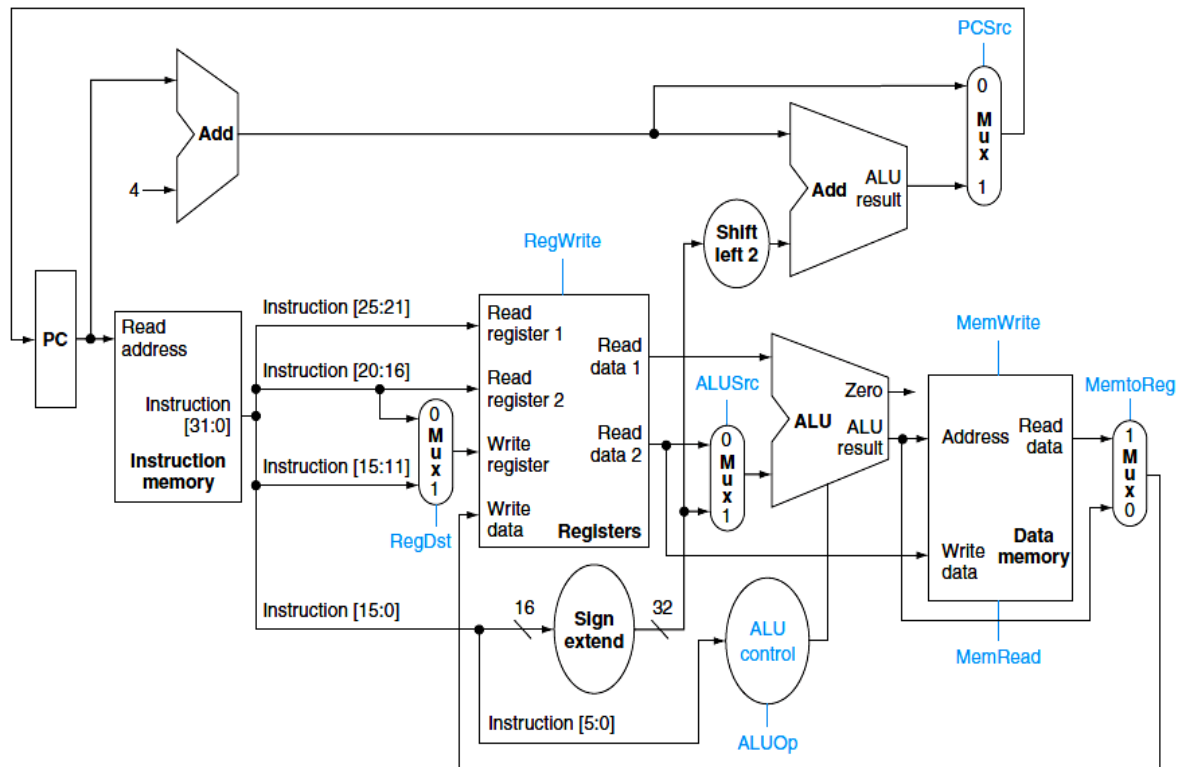
- Inputs: ALUOp and function code field
- Only the entries for which the ALU control is asserted are shown.
- Some don't-care entries have been added.
- Example
 - ALUOp does not use the encoding 11
 - so, truth table can contain entries 1X and X1, rather than 10 and 01.
 - Also, when the function field is used, the first two bits (F5 and F4) of these instructions are always 10
 - So, they are don't-care terms and are replaced with XX in the truth table

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

3.4 CONTROL IMPLEMENTATION SCHEME

3.4.1 Control Unit

- Instruction bit numbers for register numbers, opcode, function
- MUX to select destination register
 - RegDst: selects \$rd or \$rt to write data
- ALU control: uses function code and ALUOp to generate ALU operation selection
 - ALUOp: 2-bit code generated by main control



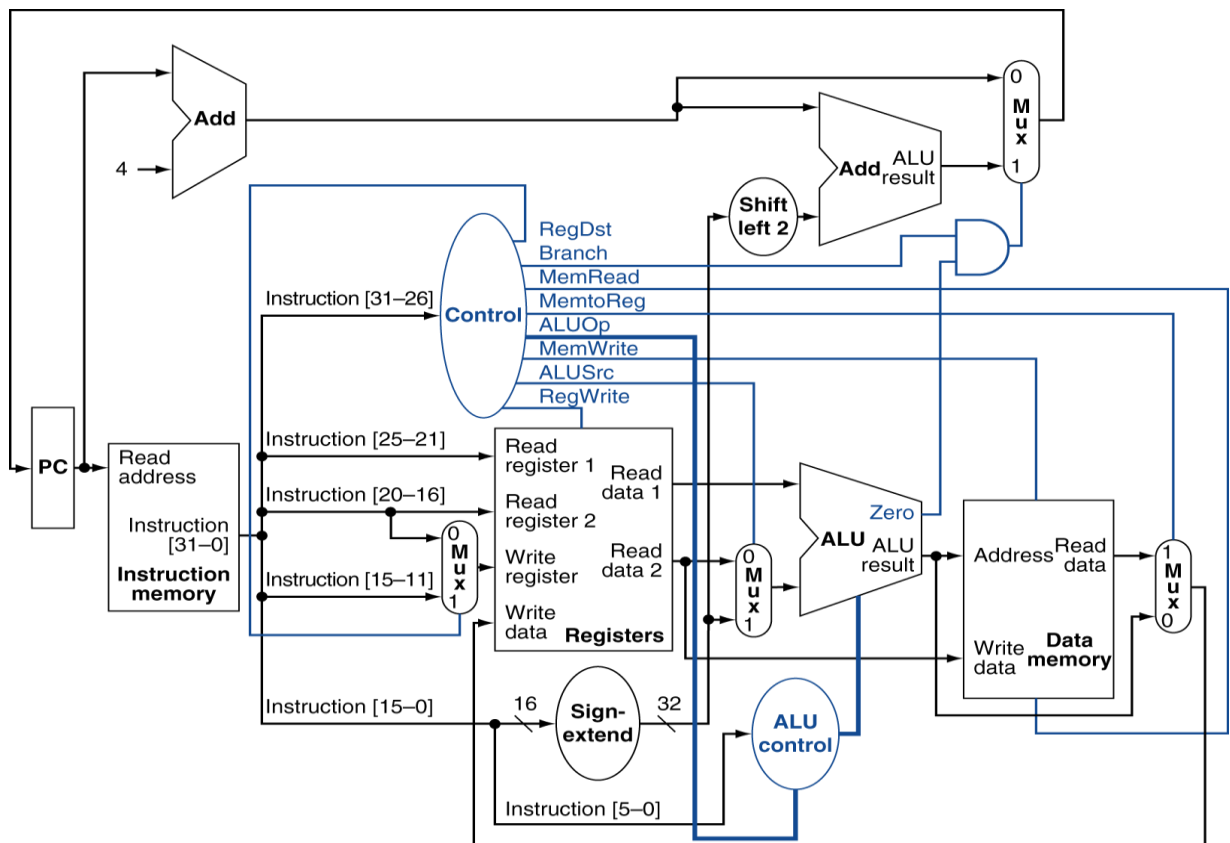
3.4.2 Effect of Control Signal

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

3.4.3 Simple Datapath with Control unit

- Input to the control unit is the 6-bit opcode field from the instruction
- 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
 - Used to combine the branch control signal and the Zero output from the ALU.
 - Output controls the selection of the next PC.
- PCSrc is now a derived signal, rather than one coming directly from the control unit.

3.4.4 Setting of control bits

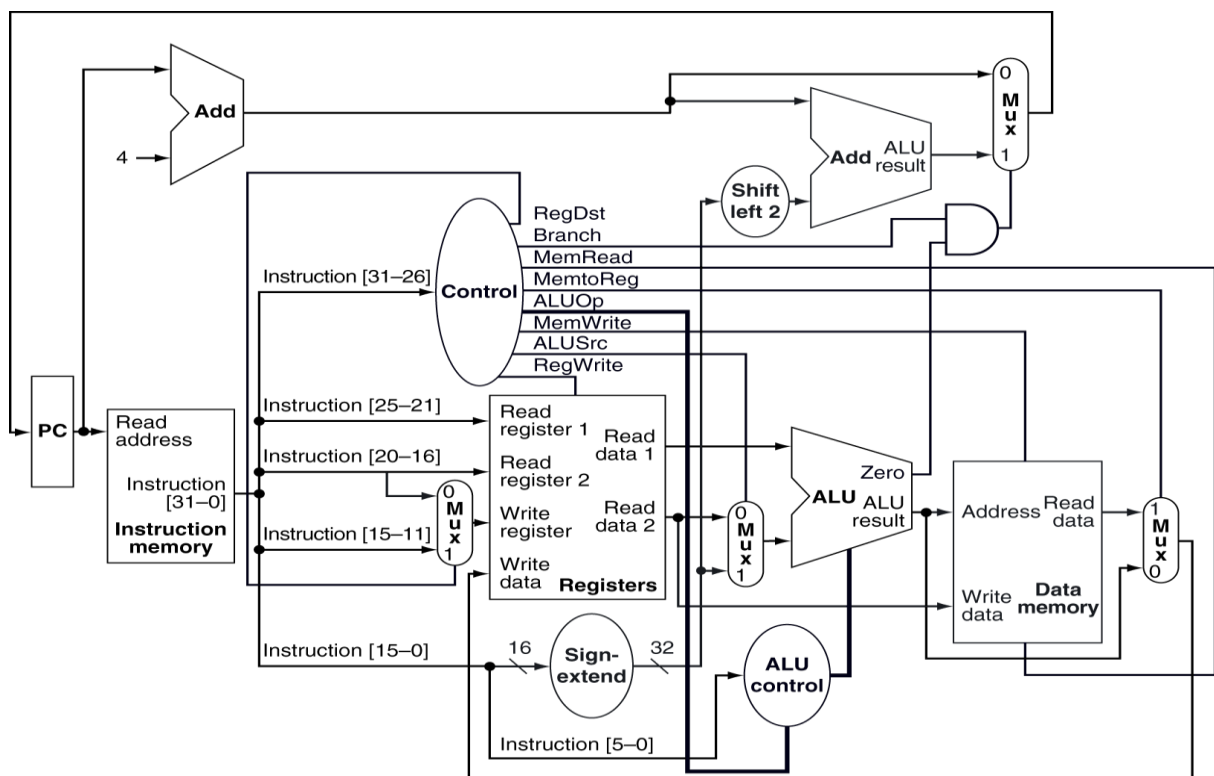
- R-format instructions (add, sub, and, or, and slt).
 - For all these instructions, the source register fields are rs and rt, and the destination register field is rd; this defines how the signals ALUSrc and RegDst are set.
 - Furthermore, an R-type instruction writes a register (RegWrite = 1), but neither reads nor writes data memory.
 - Branch = 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.
- lw and sw Instructions
 - ALUSrc and ALUOp fields are set
 - To perform the address calculation.
 - MemRead and MemWrite are set
 - To perform the memory access.
 - RegDst and RegWrite are set for a load
 - To cause the result to be stored into the rt register.
- Branch instruction
 - Sends the rs and rt registers to the ALU.
 - ALUOp field for branch is set for a subtract (ALU control = 01)
 - Used to test for equality.

- MemtoReg field is don't care when the RegWrite signal is 0
 - Since the register is not being written, the value of the data on the register data write port is not used.
- Don't cares can also be added to RegDst when RegWrite is 0

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

3.4.5 Datapath: R-type

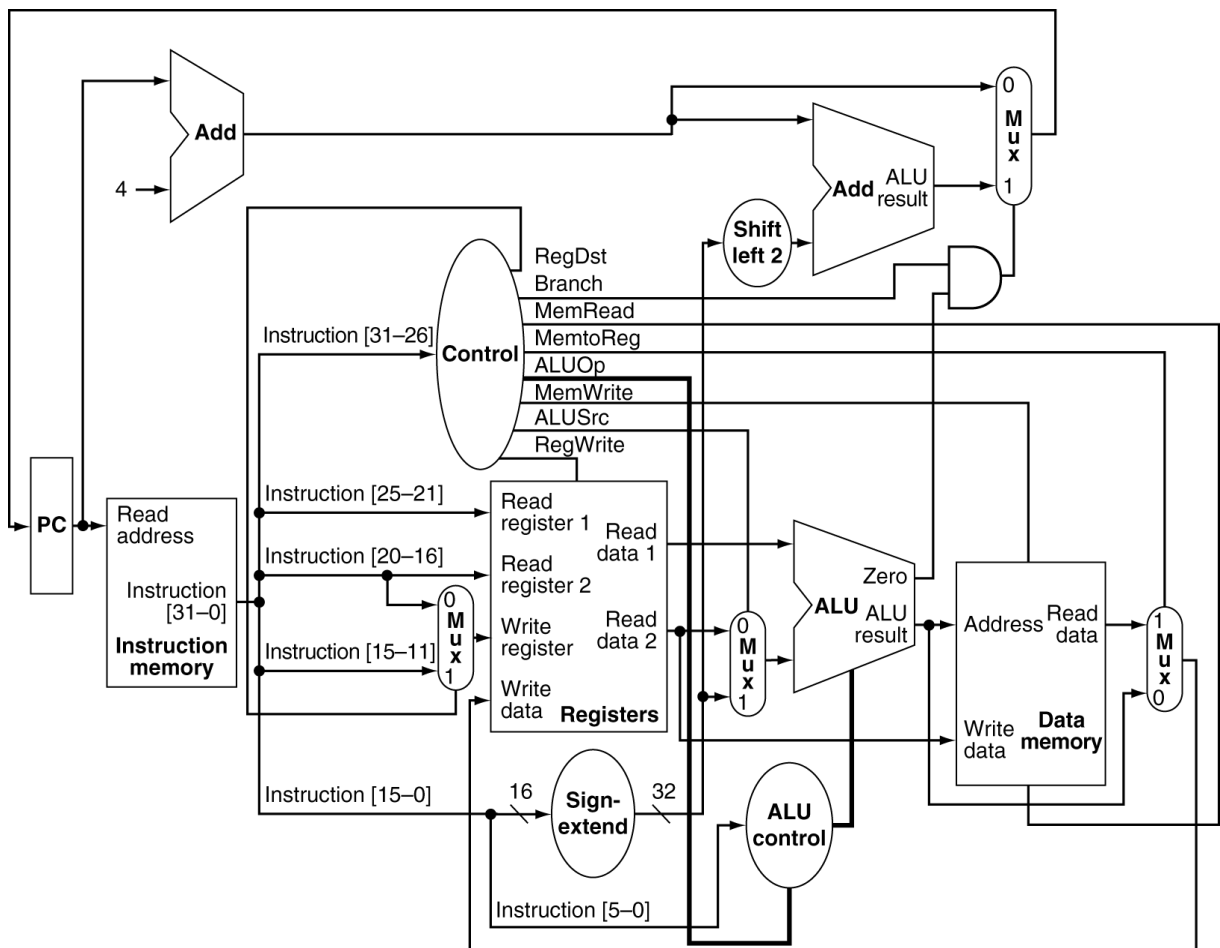
1. Fetch instruction and increment PC
2. Obtain operands from register file, based on source register numbers
3. Perform ALU operation, using ALU control to select, ALUSrc = 0
4. Select output from ALU using MemtoReg = 0
5. Write back to destination register (RegWrite = 1, RegDst = 1 for \$rd)



3.4.6 Datapath: Memory Access (load)

1. Fetch instruction and increment PC
2. Obtain base register operand (Read data 1) from register file
3. Perform addition of register value with sign-extended immediate operand in ALU, using ALU control to select operation, ALUSrc = 1 to select immediate
4. Use ALU result as address for data memory
5. Use MemtoReg = 1 to select Read data and write back to destination register

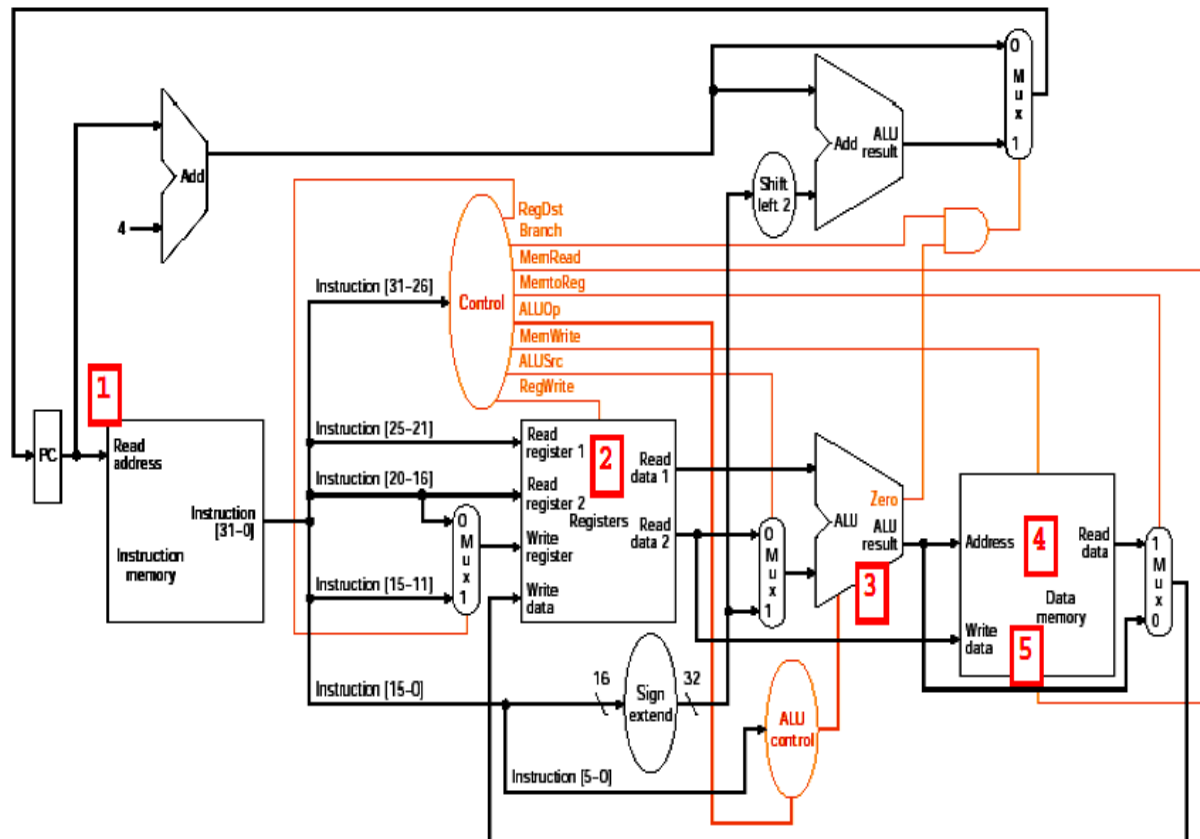
Controls: RegWrite = 1, RegDst = 0 for \$rt

**3.4.7 Datapath: memory access (store)**

1. Fetch instruction and increment PC
2. Obtain base register (Read data 1) and data (Read data 2) from register file

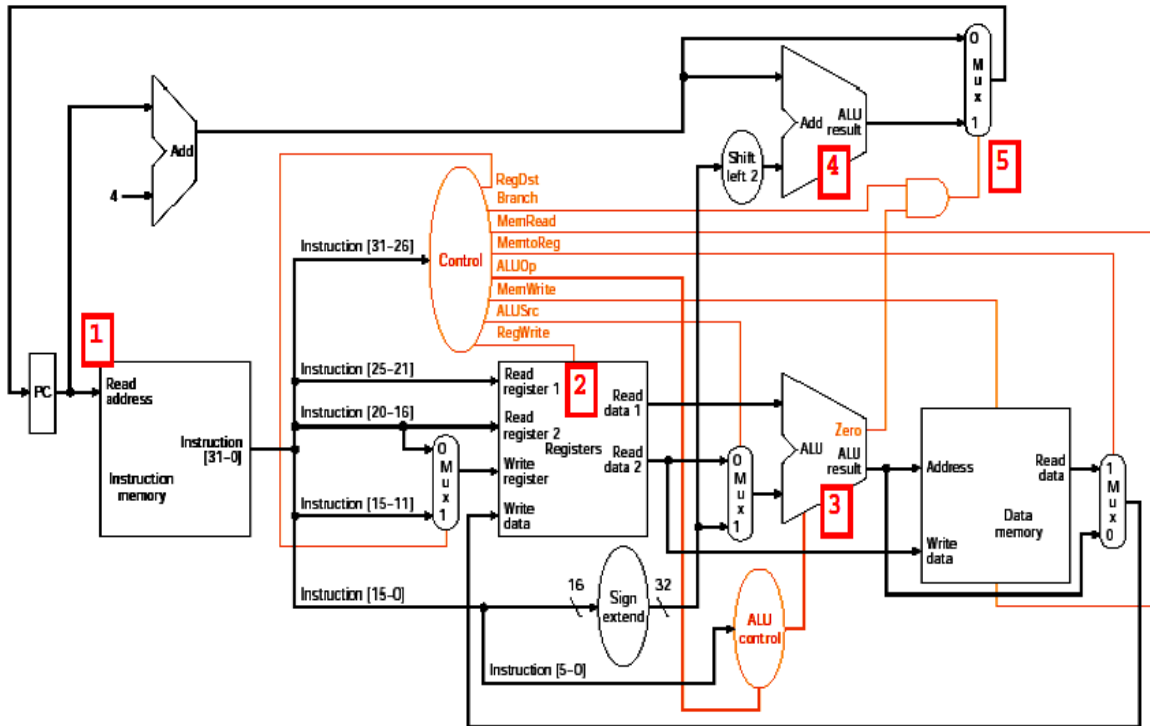
3. Perform addition of register value with sign-extended immediate operand in ALU, using ALU control to select operation, ALUSrc = 1 to select immediate
4. Use ALU result as address for data memory
5. Using MemWrite = 1, write data operand to memory address

Note that MemtoReg and RegDst are don't cares

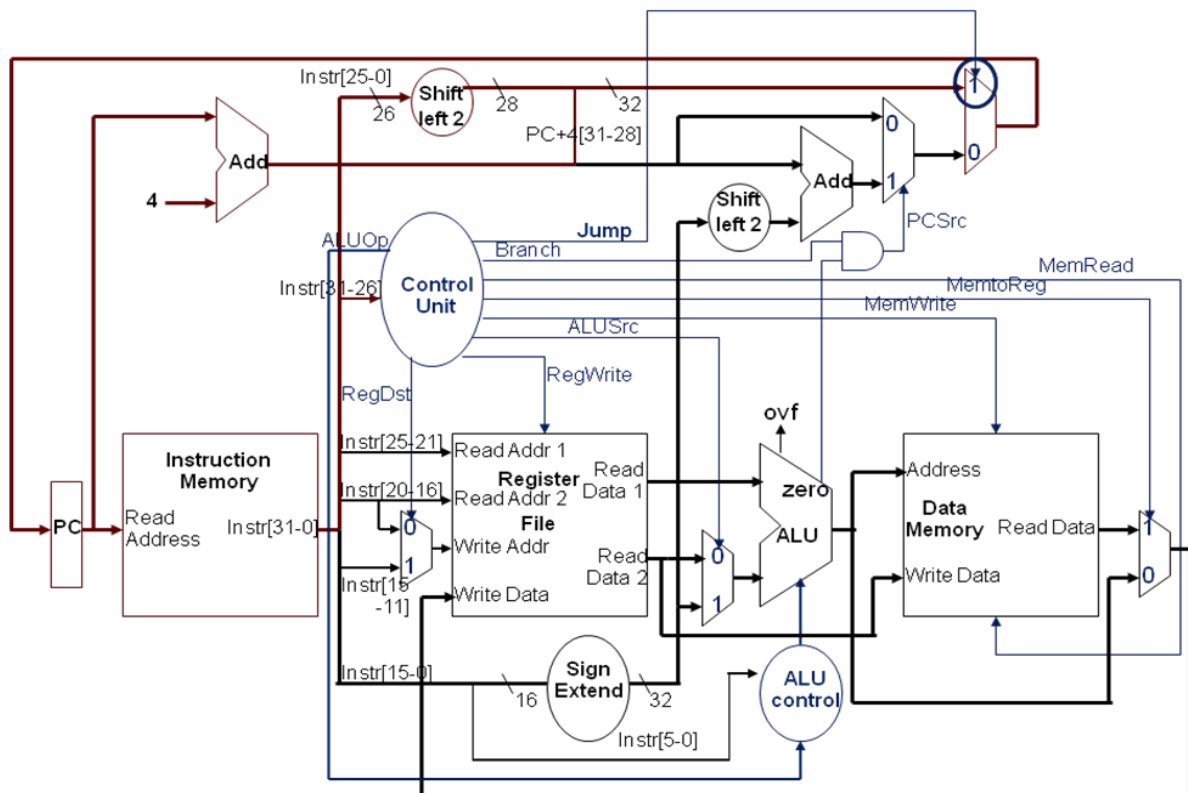


3.4.8 Datapath: branch

1. Fetch instruction and increment PC
2. Read 2 registers from register file for comparison
3. ALU subtracts data values, using ALU control to select operation and ALUSrc = 0
4. Generate branch address: add (PC + 4) to sign-extended offset, shifted left by 2
5. Use Zero output from ALU (and Branch control) to determine which result to use to update PC
 - If equal, use branch address
 - else use incremented PC



3.4.9 Datapath: jump



1. Shift instruction bits 25-0 left 2 bits to create 28 bit value Fig. 5.29
2. Combine with bits 31-28 of (PC + 4) to produce 32-bit jump address
3. Additional MUX uses Jump control to select instruction address
 - 0: Incremented PC or branch target
 - 1: Jump address

3.5 PIPELINING

- Pipelining is an implementation technique in which multiple instructions are overlapped in execution.
- This enables the processors to complete the tasks faster.
- Pipeline is divided into five stages.
- Each stage completes a part of an instruction in parallel.
- The stages are connected one to the next to form a pipe like structure.
- Instructions enter at one end, progress through the stages, and exit at the other end.

Pipelining and Performance

- Pipelining does not decrease the time for individual instruction execution.
- Instead, it increases instruction throughput.
- The throughput of the instruction pipeline is determined by how often an instruction exits the pipeline.
- Instruction pipeline is to overlap the operation of the different stages and maximize the throughput.

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

Single-Cycle vs. Pipeline

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	2ns	1ns	2ns	2ns	1ns	8ns
Store word (sw)	2ns	1ns	2ns	2ns		7ns
R-format (add, sub, and, or, slt)	2ns	1ns	2ns		1ns	6ns
Branch (beq)	2ns	1ns	2ns			5ns

Example – Need for Pipelining

Consider the following instructions which is executed with and without pipeline

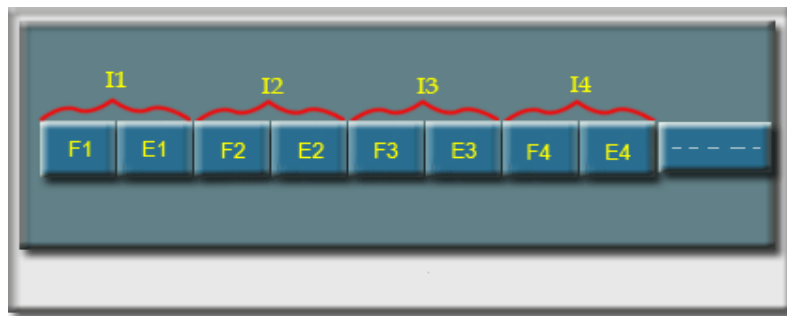
I1: lw \$1, 100(\$0)

I2: lw \$2, 200(\$0)

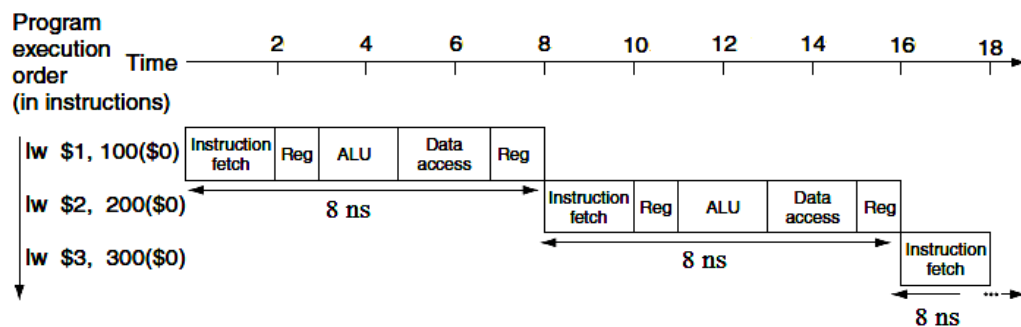
I3: lw \$3, 300(\$0)

Without Pipeline

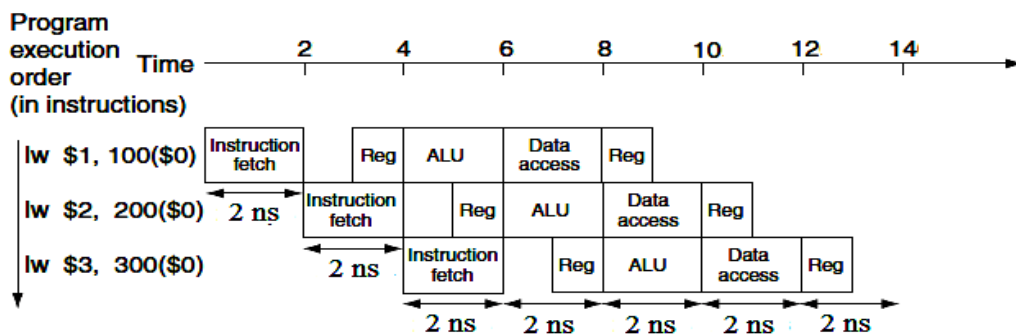
Execution of instructions without pipeline is called sequential pipeline. Every instruction is



The execution is done as



Total Clock Cycle = 24 ns

With Pipeline

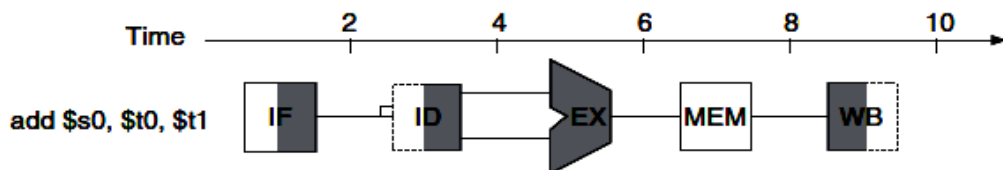
Total Clock Cycle = 14 ns

3.5.1 Stages of a pipeline

MIPS pipeline classically take the following five steps:

1. Fetch instruction from memory (IF)
2. Read registers while decoding the instruction (ID)
 - In MIPS implementation reading and decoding occur simultaneously.
3. Execute the operation or calculate an address(EX)
4. Access an operand in data memory(MEM)
5. Write back the result into a register(WB)

3.5.1.1 Graphical representation



- Time represents Clock Cycle

Symbols

- IF for the instruction fetch stage, with the box representing instruction memory
- ID for the instruction decode/register file read stage, with the drawing showing the register file being read
- EX for the execution stage, with the drawing representing the ALU
- MEM for the memory access stage, with the box representing data memory
- WB for the write back stage, with the drawing showing the register file being written

Shading

- Shading indicates the element is used by the instruction
- Non Shading indicates the element is not used by the instruction
 - Example: MEM has a white background because add does not access the data memory.

- Shading on the right half of the register file or memory means the element is read in that stage
 - Example: Right half of ID is shaded in the second stage because the register file is read
- Shading of the left half of the register file or memory means it is written in that stage
 - Example: Left half of WB is shaded in the fifth stage because the register file is written

3.5.2 Pipeline Hazards

- Any condition that causes the pipeline to stall is called a hazard.
- It prevents the next instruction in the instruction stream from being executing during its designated clock cycle.
- These events are called hazards, and there are three different types.
- They are
 - Data hazard
 - Control/ Instruction hazard
 - Structural hazard

Terminologies

Forwarding is also called bypassing. It is 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.

Load-use data hazard is a specific form of data hazard in which the data requested by a load instruction has not yet become available when it is requested.

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

3.5.2.1 Structural Hazards

- It occurs when two instructions use the same resource at the same time.
- It means that the hardware cannot support the combination of instructions that we want to execute in the same clock cycle.
- Structural hazard is an occurrence in which a planned instruction cannot execute in the proper clock cycle because the hardware cannot support the combination of instructions that are set to execute in the given clock cycle.

Program Execution order(in instructions) ↓	Time→	cc1	cc2	cc3	cc4	cc5	cc6	cc7	cc8
	lw \$1, 100(\$0)	IF	ID	EX	MEM	WB			
	lw \$2, 200(\$0)		IF	ID	EX	MEM	WB		
	lw \$3, 300(\$0)			IF	ID	EX	MEM	WB	
	lw \$4, 400(\$0)				IF	ID	EX	MEM	WB

- In cc4, First instruction is accessing data from memory, while Fourth instruction is fetching an instruction from that same memory.
 - Without two memories, our pipeline could have a structural hazard.

3.5.2.2 Data Hazards

- It occurs when the data are not available at the time expected in the pipeline.
- It is also called pipeline data hazard.
- It is an occurrence in which a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.
- A data hazard is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline.
- As a result some operation has to be delayed, and the pipeline stalls.

Classification

Consider two instructions I_1 and I_2 , with I_1 occurring before I_2 .

The possible data hazards are:

- RAW (read after write)
 - I_2 reads a source before I_1 writes it,
 - So, I_2 incorrectly gets the old value
- WAW (write after write)
 - I_2 tries to write an operand before it is written by I_1 .

1.135 Computer Architecture

- The writes end up being performed in the wrong order, leaving the value written by I_1 rather than the value written by I_2 in the destination.
- WAR (write after read)
 - I_2 tries to write a destination before it is read by I_1
 - So, I_1 incorrectly gets the new value

Handling Methods (Read-After-Write (RAW) Hazard)

- **Forwarding**

Consider the following instructions

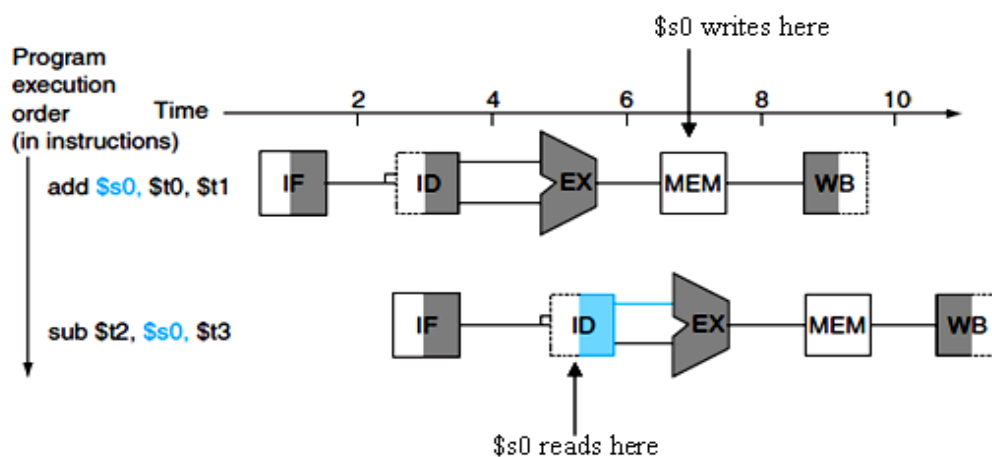
```
add  $s0, $t0, $t1
sub  $t2, $s0, $t3
```

It is equivalent to the following statement

$$\$s0 = \$t0 + \$t1$$

$$\$t2 = \$s0 + \$t3$$

Here, the add instruction followed immediately by a subtract instruction that uses the sum ($\$s0$) is represented



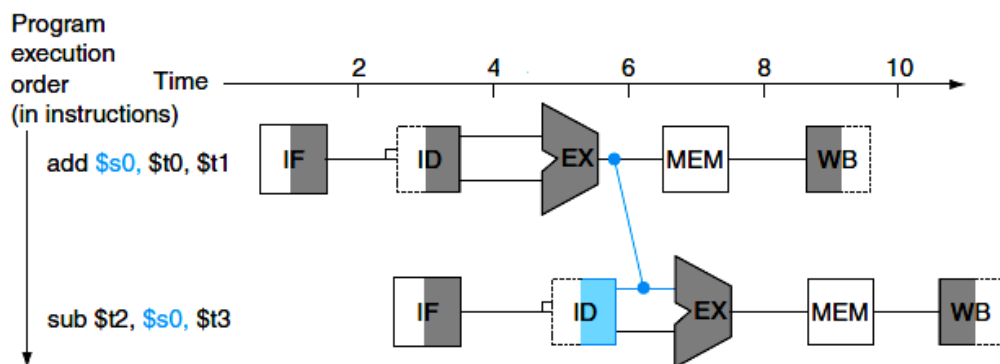
- Sub instruction will read an old value of $R0$
- Pipeline Stall sub in ID for 3 cycles until result written to $R0$.
- Solution

1. Insert bubbles
2. Use internal data forwarding
 - Also called bypassing
 - Adding extra hardware to retrieve the missing item early from the internal resources.

Note:

- “Forwarding” comes from the idea that the result is passed forward from an earlier instruction to a later instruction.
- “Bypassing” comes from passing the result by the register file to the desired unit.

Graphical representation of forwarding



- The connection shows the forwarding path from the output of the EX stage of add to the input of the EX stage for sub, replacing the value from register \$s0 read in the second stage of sub.

○ MEM Forwarding (for Load-use Data Hazard)

Consider the following instructions

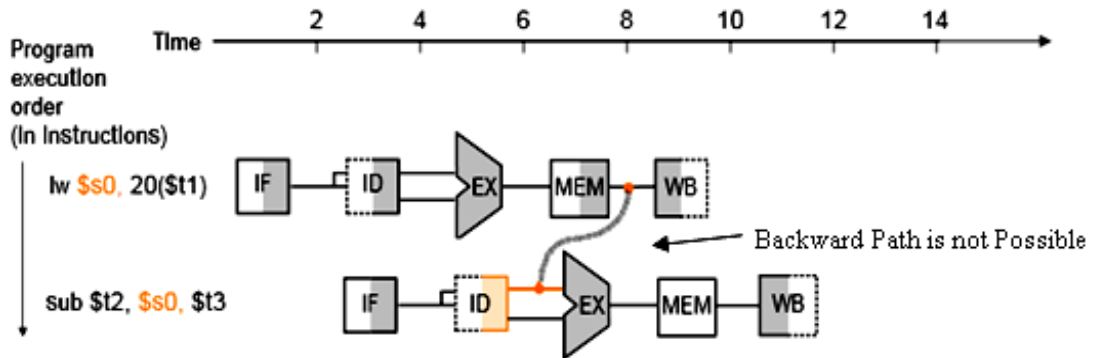
```
lw    $s0, 20($t1)
sub   $t2, $s0, $t3
```

It is equivalent to the following statement

$$\$s0 = \$t1$$

$$\$t2 = \$s0 - \$t3$$

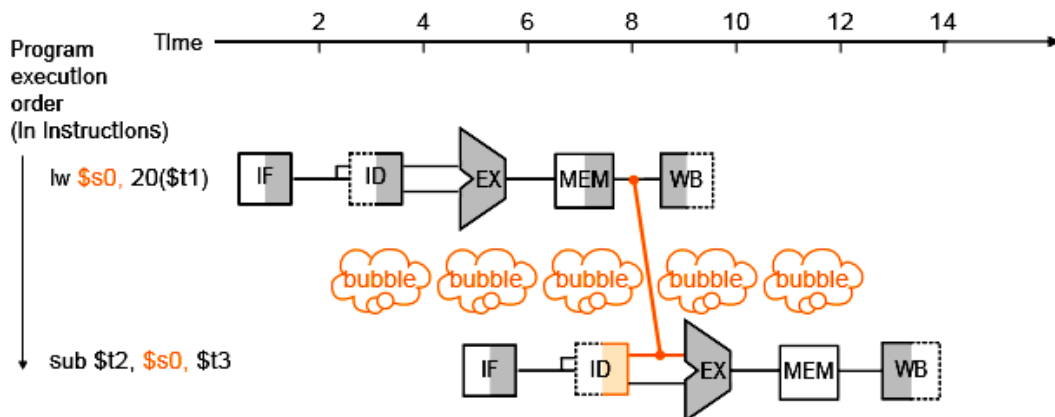
Here, the result from lw will not be available until after MEM stage.



- Path from memory access stage output to execution stage input would be going backwards in time

Solution

- One bubble need to be inserted.



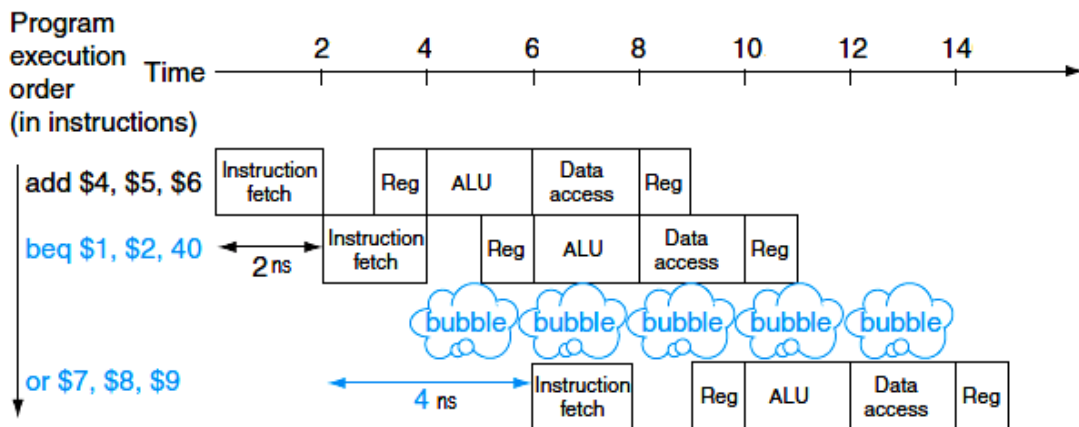
3.5.2.3 Control Hazards

- It is also called branch hazard or instruction hazard.
- It occurs when the branching decisions are made before branch condition is evaluated.
- It is an occurrence in which the proper instruction cannot execute in the proper clock cycle because the instruction that was fetched is not the one that is needed.
- The flow of instruction addresses is not what the pipeline expected.

Handling Methods

Stall the pipeline

- A one-stage pipeline stall, or bubble, after the branch

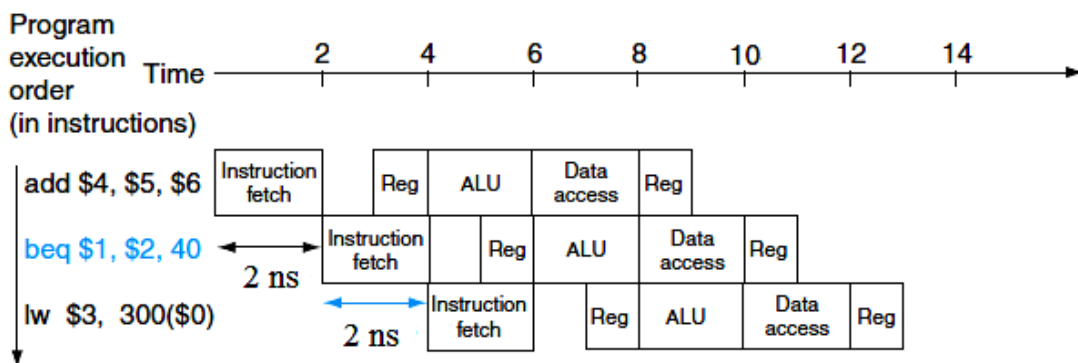


▪ Drawback:

- The cost is too high to use. So, predict method is used

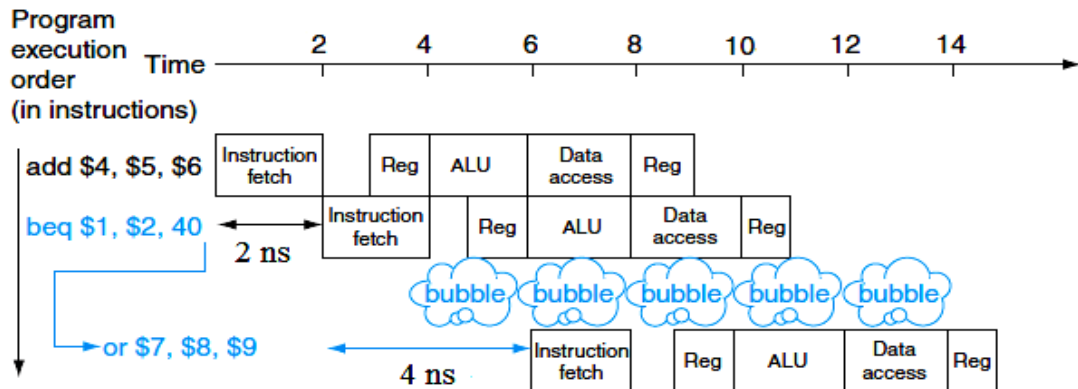
Predict branch not taken

- One simple approach to always predict that branches will fail (Not taken)



Predict branch taken

- Pipeline stall occur only when branches are taken
- A bubble is inserted at least during the first clock cycle immediately following the branch.
- This simplifies the tasks.

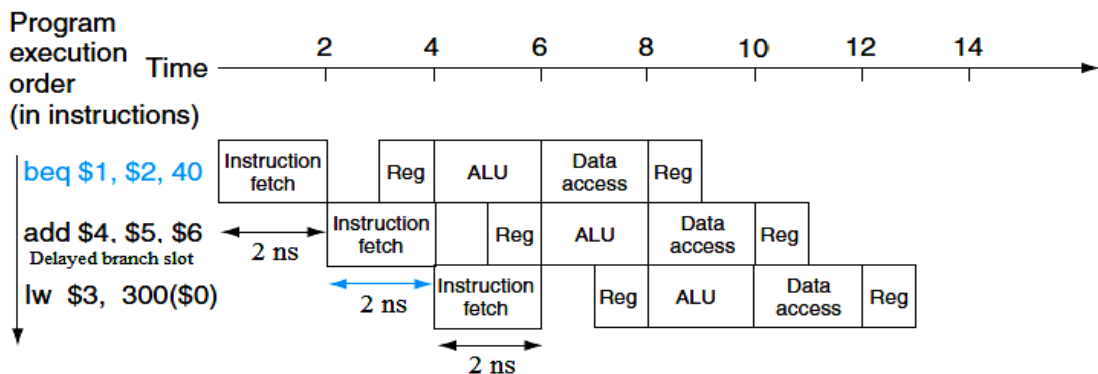


Branch Prediction

- A method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.

Delayed Branch

- Executes the next sequential instruction, with the branch taking place after that one instruction delay.



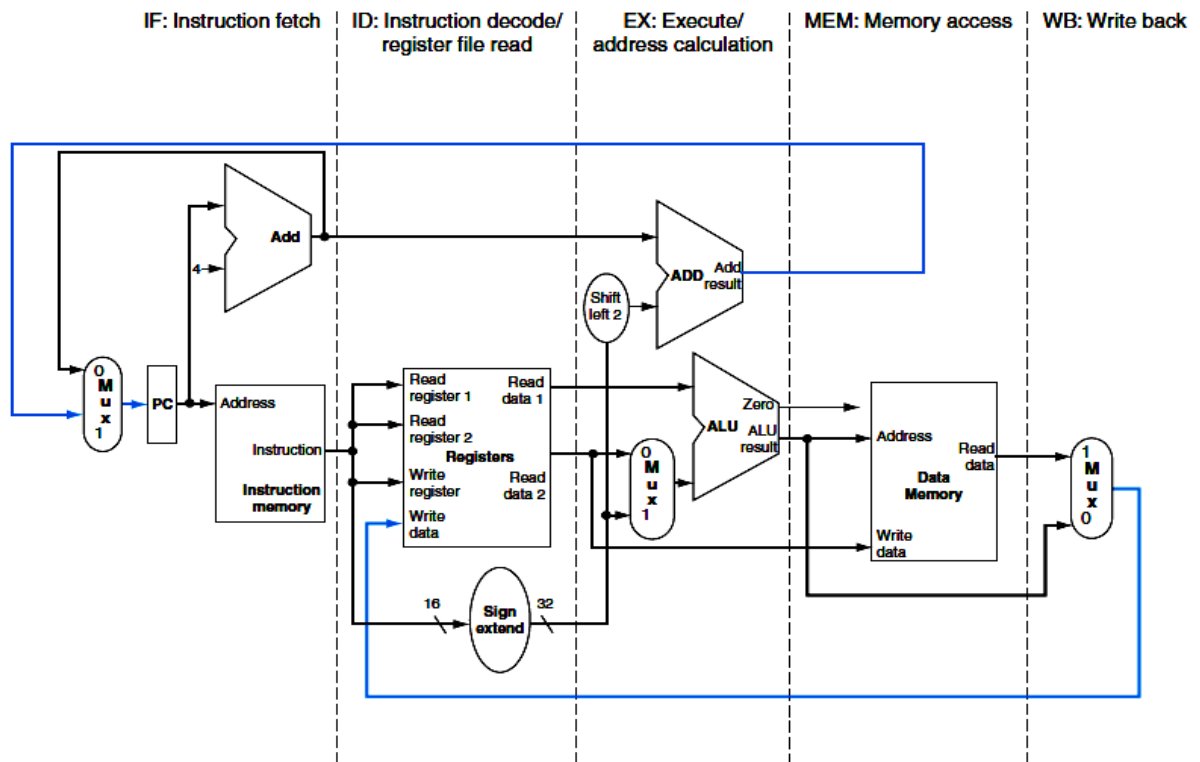
- Add instruction before the branch does not affect the branch.
- Can be moved to the delayed branch slot following the branch.

3.6 PIPELINED DATAPATH

MIPS pipeline classically take the following five steps:

- Fetch instruction from memory (IF)
- Read registers while decoding the instruction (ID)
- Execute the operation or calculate an address (EX)
- Access an operand in data memory (MEM)
- Write back the result into a register (WB)

3.6.1 Single Cycle Datapath

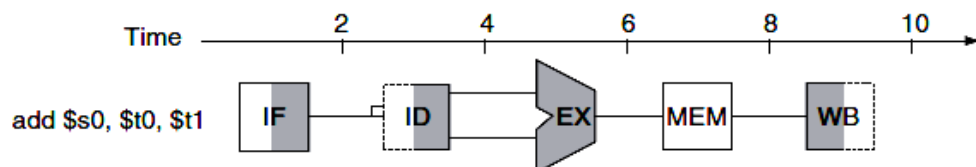


Exceptions

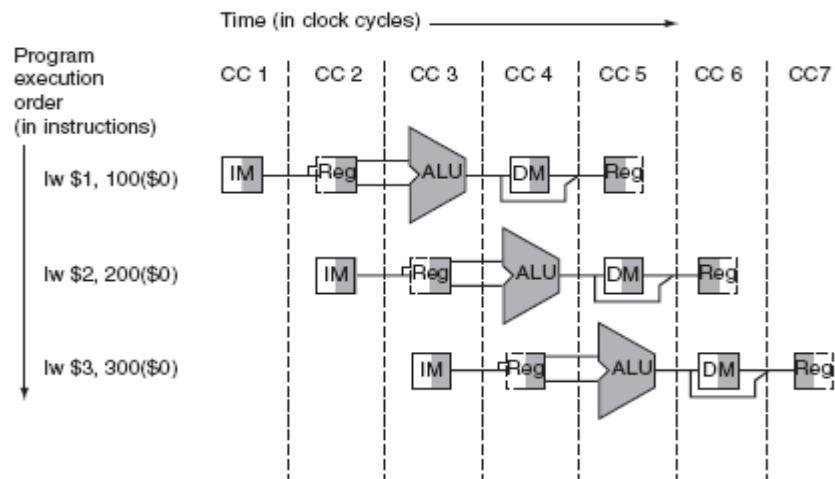
The following two exceptions arise when the instructions are flow from left-to-right:

- Write-back stage - places the result back into the register file in the middle of the datapath.
- Selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage.

3.6.2 Graphical representation



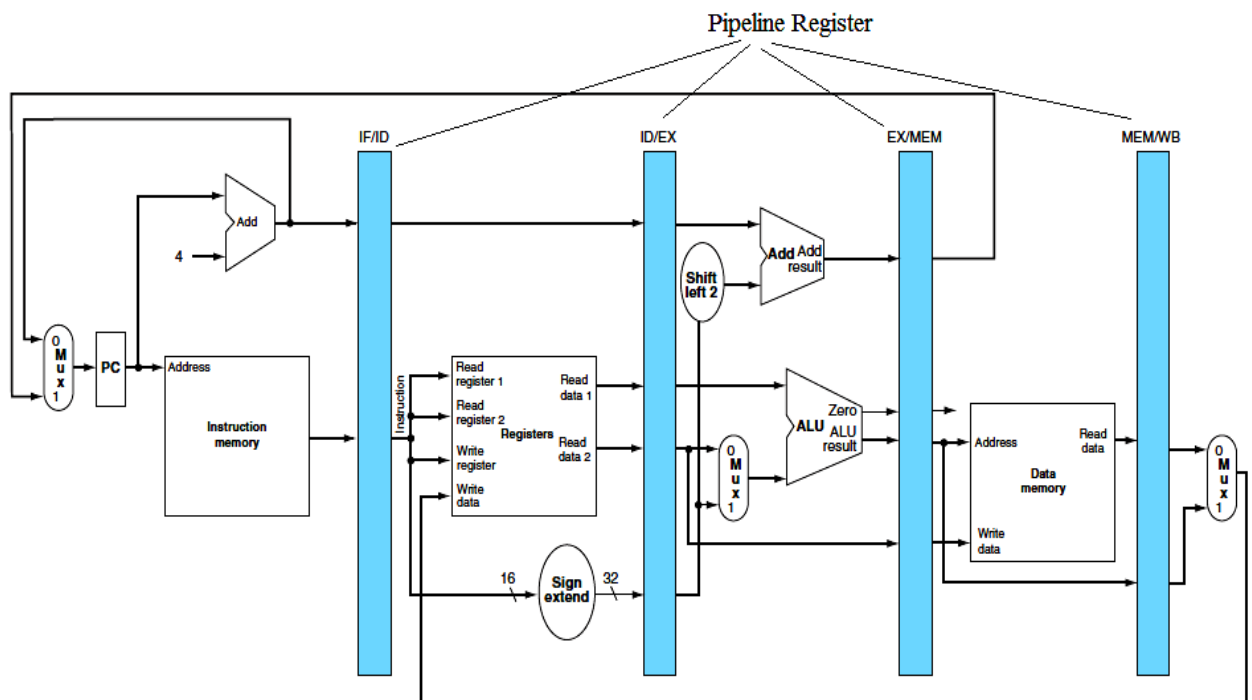
- Shows the execution of the instructions by displaying their private datapaths on a common time line.



Single clock cycle pipeline diagram

3.6.3 Pipelined version of the datapath

- Pipeline registers
 - Separate each pipeline stage
- No pipeline register is available at the end of the write-back stage
 - All instructions must update some state in the processor—the register file, memory, or the PC
 - So a separate pipeline register is redundant to the state that is updated



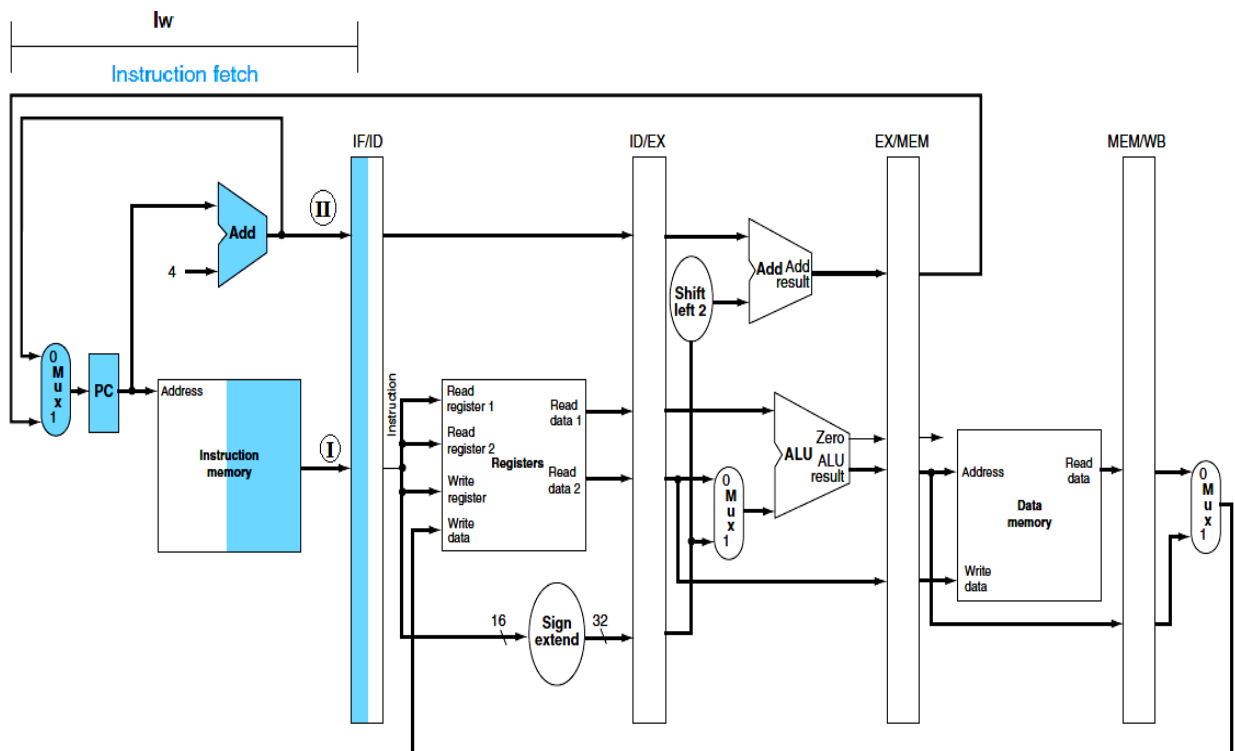
3.6.4 Demonstration

3.4.4.1 Load instruction

The active portions of the datapath highlighted as a load instruction goes through the following five stages of pipelined execution

1. Instruction fetch

- i. Instruction being read from memory using the address in the PC
 - a. Fetched instruction is placed in the IF/ID pipeline register.
- ii. The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle
 - a. Incremented PC address is also saved in the IF/ID pipeline register which is needed later for an instruction, such as beq

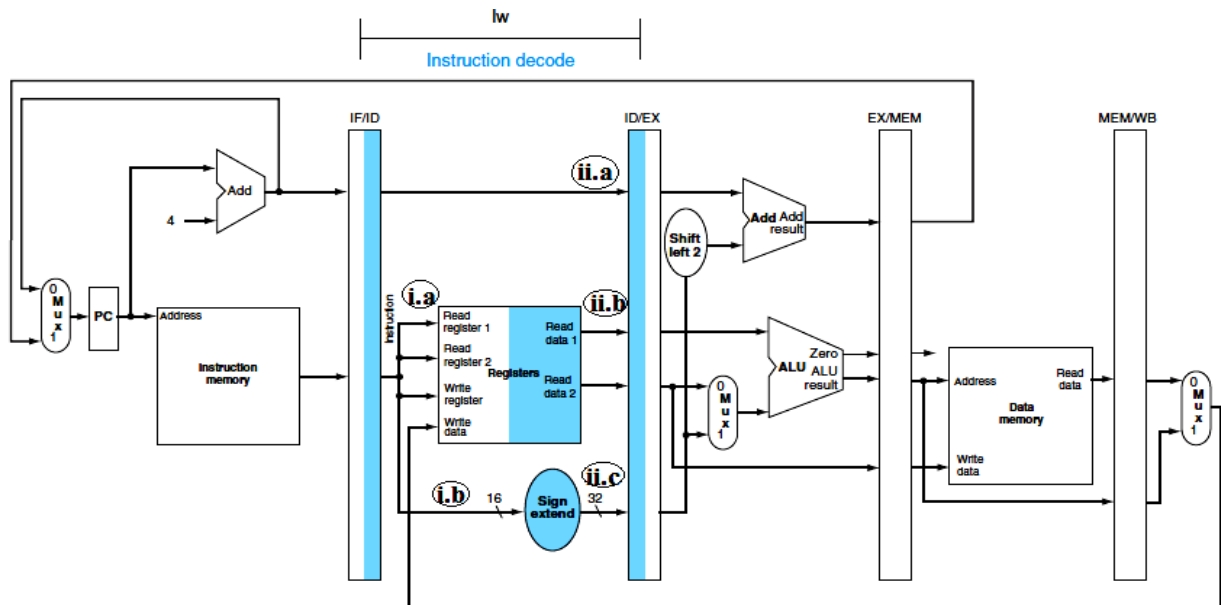


2. Instruction decode and register file read

- i. Instruction portion of the IF/ID pipeline register supplies
 - a. Register numbers to read the two registers.
 - b. 16-bit immediate field, which is sign-extended to 32 bits

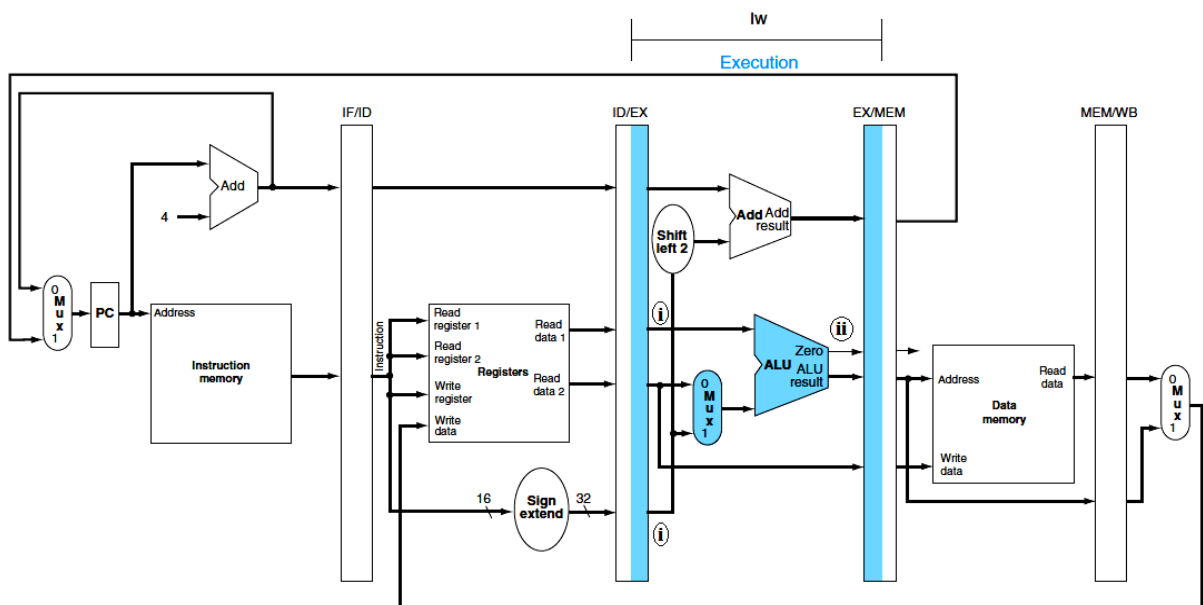
ii. The following values are stored in the ID/EX pipeline register

- Incremented PC address
- Data's read from the two registers
- 32-bit sign-extended value



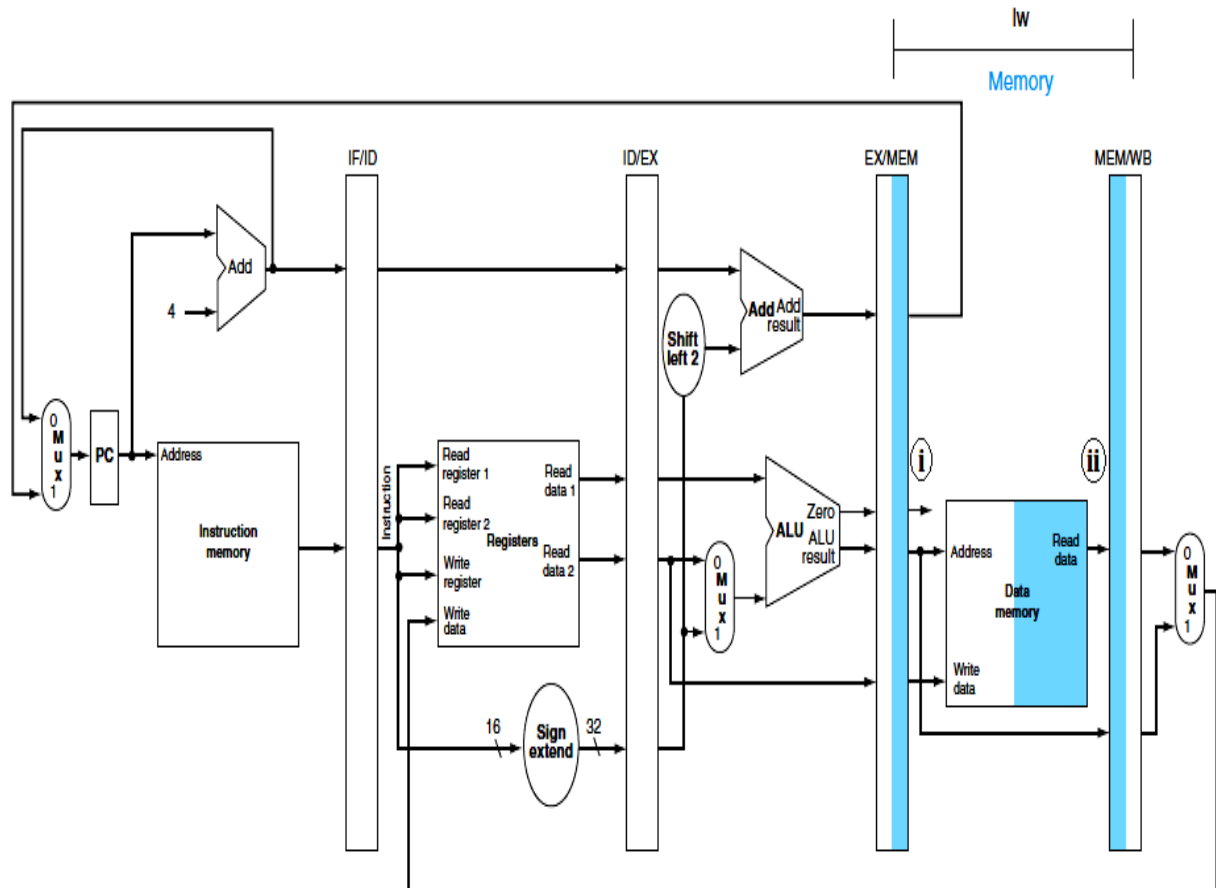
3. Execute or address calculation

- Reads the register1 contents and the sign-extended immediate from the ID/EX pipeline register.
- Adds them using the ALU and that sum is placed in the EX/MEM pipeline register.



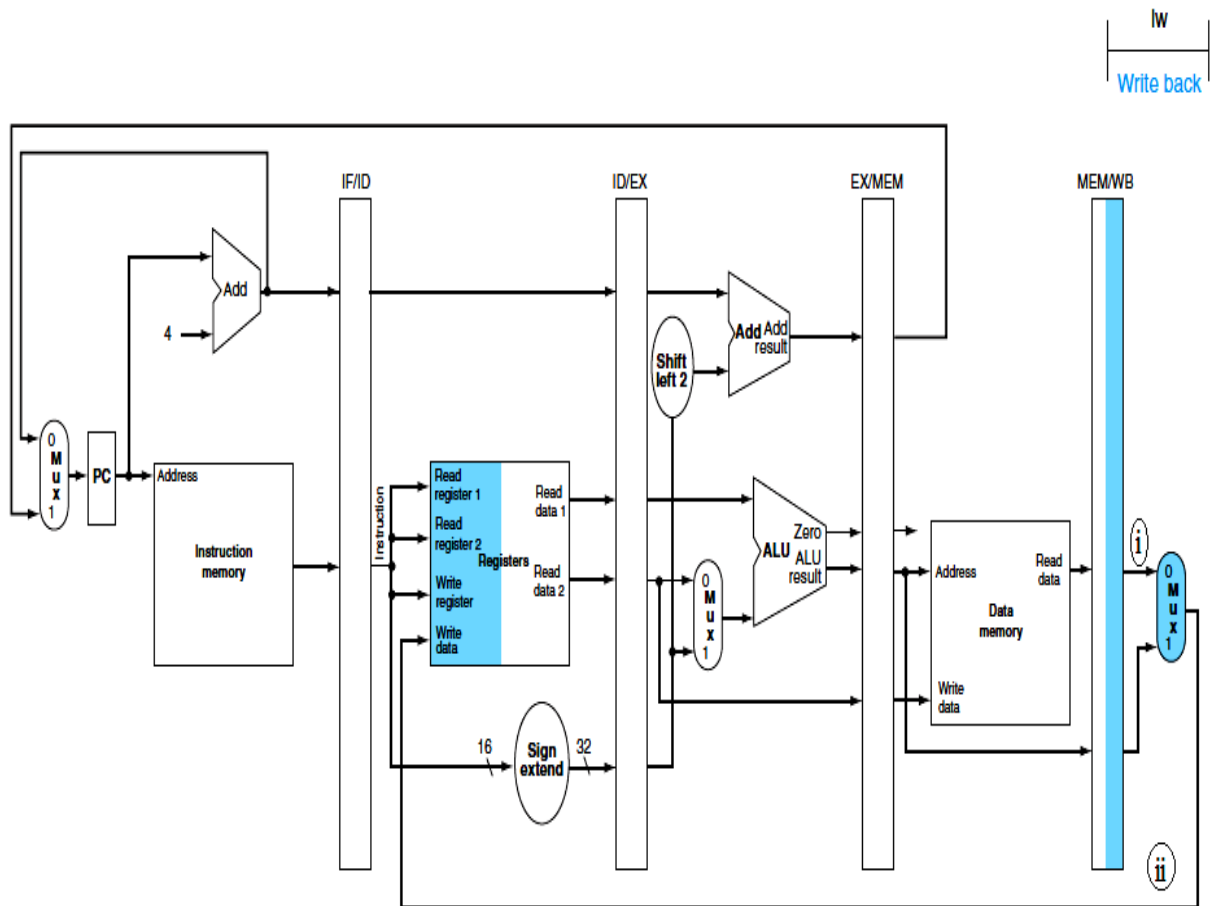
4. Memory access

- i. Read the data memory using the address from the EX/MEM pipeline register.
- ii. Load the data into the MEM/WB pipeline register.



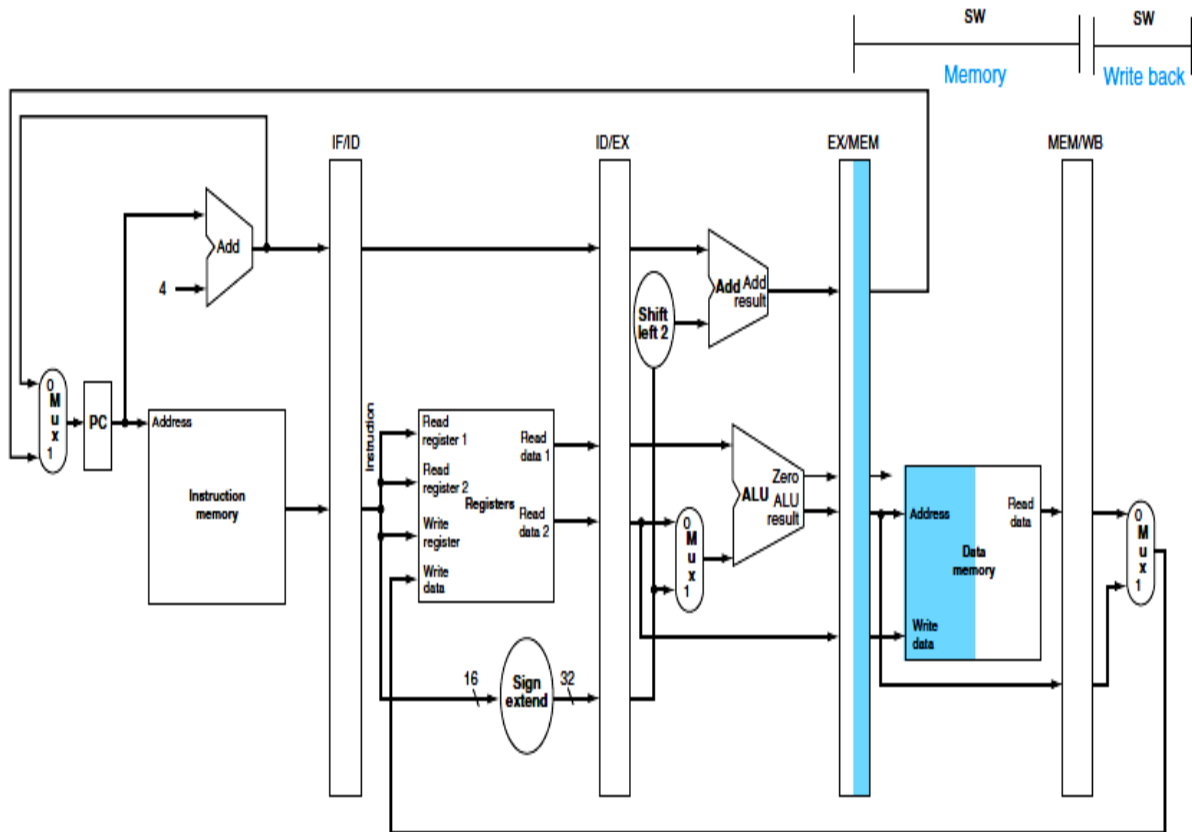
5. Write back

- i. Read the data from the MEM/WB pipeline register.
- ii. Write it into the register file in the middle of the figure.



3.4.4.2 Store Instruction

1. Instruction fetch
2. Instruction decode and register file read
3. Execute and address calculation
 - Second register value is loaded into the EX/MEM pipeline register to be used in the next stage.
4. Memory access
 - Register containing the data to be stored was read and stored in ID/EX
 - Only way to make the data available during the MEM stage is to place the data into the EX/MEM pipeline register in the EX stage.
5. Write back
 - No process is done



3.7 PIPELINED CONTROL

- To specify control for the pipeline, the control values are set during each pipeline stage.

3.7.1 Types of Control

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

3.7.2 Graphical representation of Pipelined Control

Control lines are divided into five groups according to the pipeline stage

1. Instruction fetch

- The control signals to read instruction memory and to write the PC are always asserted
 - No control to set

2. Instruction decode and register file read

- No optional control lines to set

3. Execute and address calculation

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

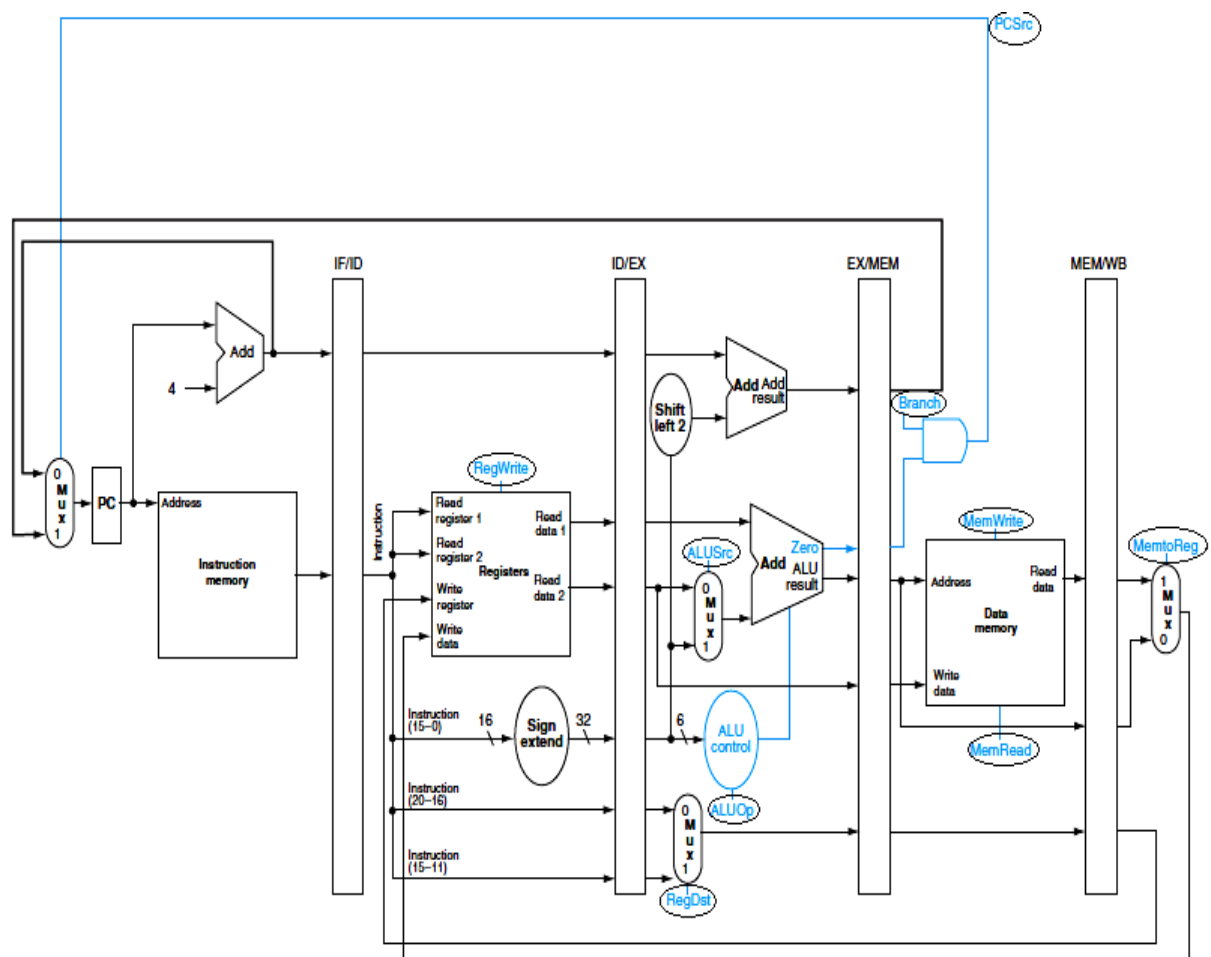
4. Memory Access

- The control lines set in this stage are
 - Branch - set by the branch equal
 - MemRead - set by the load instruction

- MemWrite - set by the store instruction.
- PCSrc - selects the next sequential address unless control asserts Branch and the ALU result was zero.

5. Write Back

- The two control lines are
 - MemtoReg - Decides between sending the ALU result or the memory value to the register file.
 - RegWrite - Writes the chosen value.



3.8 EXCEPTIONS

- Exceptions are also called as interrupts.
- These are unscheduled events that disrupt the execution of programs.

Example

Consider the following instruction

ADD R1, R2, R1

// Equivalent to the $R1 = R2 + R1$

- Arithmetic overflow has occurred
- Need to transfer control to the exception routine immediately after this instruction
 - Not to contaminate other registers or memory locations

Solution

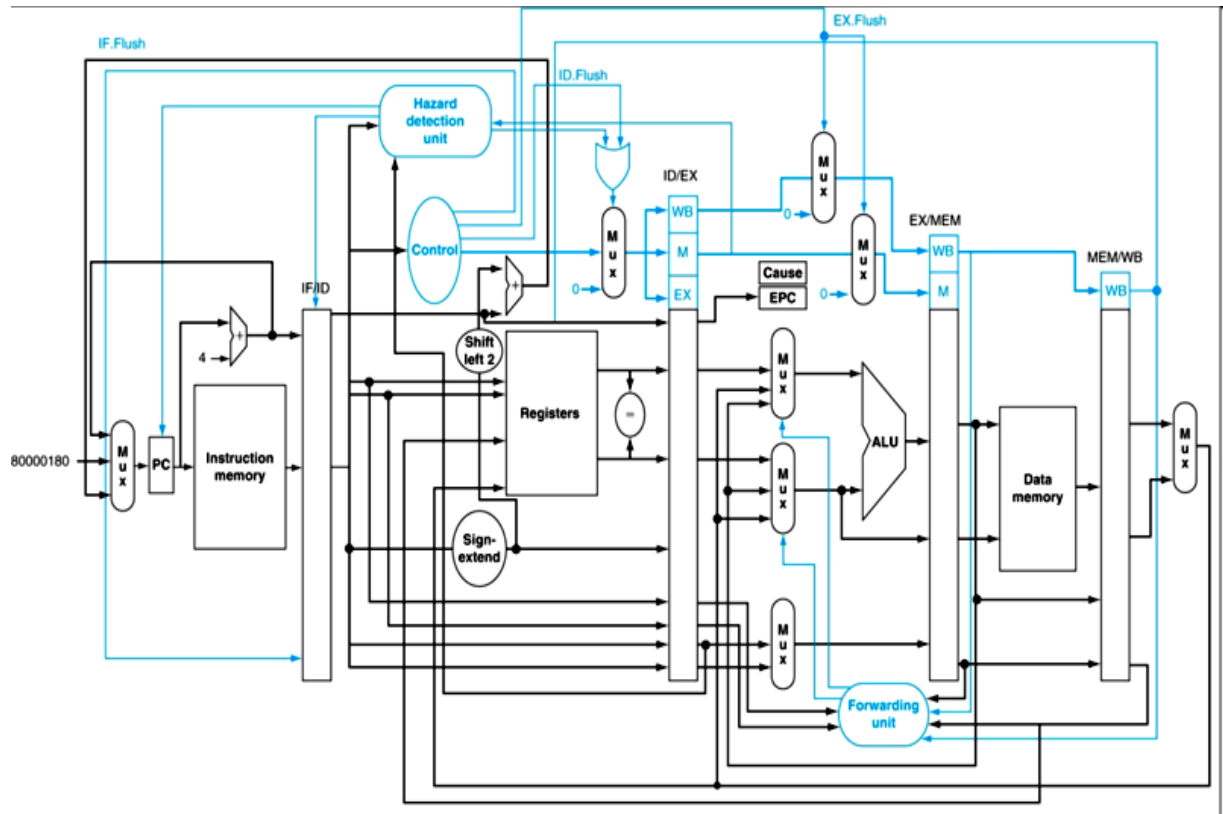
Flush the instructions that follow the add instruction from the pipeline and begin fetching instructions from the new address.

- Flush the instruction in the IF stage by turning it into a nop
- Flush the instructions in the ID stage, the multiplexor already in the ID stage is used that zeros control signals for stalls
 - A new control signal, called ID.Flush, is ORed with the stall signal from the Hazard Detection Unit to flush during ID.
- Flush the instructions in the EX phase.
 - A new signal called EX.Flush is used to cause new multiplexors to zero the control lines.
- Start fetching instructions from location $4000\ 0040_{\text{hex}}$ (exception location for an arithmetic overflow)
 - Add an additional input to the PC multiplexor that sends $4000\ 0040_{\text{hex}}$ to the PC

Limitation

- If the execution is stopped in the middle of the instruction, the programmer will not be able to see the original value of register R1 that helped cause the overflow because it will be hit as the destination register of the add instruction.
- In order to overcome,
 - Overflow exception is detected during the EX stage.
 - Hence, the EX.Flush signal is used to prevent the instruction in the EX stage from writing its result in the WB stage.

Datapath to handle exceptions



Exception Support

- EPC (Exceptional Program Counter)
 - A 32-bit register.
 - Hold the address of the offending instruction.
- Cause
 - A 32-bit register in MIPS (some bits are unused currently.)
 - Record the cause of the exception.

Exception Detection

- **Undefined instruction**
 - This exception is detected when no next state is defined from state 1 for the op value.
 - Handle this by defining the next state value for all op values other than lw, sw, 0 (R-type), jmp, and beq as a new state, “other”.

- **Arithmetic overflow**
 - It is detected with the Overflow signal out of the ALU.
 - Signal is used in the modified FSM to specify an additional possible next state.

UNIT

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.

4.1 INTRODUCTION

- A computer system that has two or more processors is called as a multiprocessor
- As power is an overriding issue in processors, the computer architects had replaced large inefficient processors with many smaller and efficient processors. This provides improved power efficiency and scalable performance.
- Consider a multiprocessor system with ‘n’ processors. If a processor fails, the system would continue to provide service with the remaining ‘n-1’ processors.
- Parallelism is a mode of operation in which a process is split into parts, which are executed simultaneously on different processors attached to the same computer.

Goals of Parallelism

- Parallelism speeds up the computer processing capability (or) it increases the computational speed.
- It increases throughput by making two or more ALUs in CPU can work concurrently. [Throughput is the amount of processing that can be accomplished during a given interval of time]
- It improves the performance of the computer for a given clock speed.

Types of Parallelism

- Instruction level parallelism
- Thread level or Task level Parallelism

- Bit-level Parallelism
- Data level parallelism
- Transaction level parallelism

4.2 INSTRUCTION-LEVEL-PARALLELISM

The technique used by overlapping the execution of instructions so as to improve the performance is called instruction level parallelism.

ILP is the principle that there are many instructions in code that don't depend on each other so it's possible to execute those instructions in parallel.

- Building compilers to analyze the code.
- Building hardware to be even smarter than that code.

There are two primary methods for increasing the performance of a system using instruction-level parallelism.

- By increasing the depth of the pipeline to overlap more instructions.
- By replicating the internal components of the computer so that it can launch multiple instructions in every pipeline stage. This technique is called as **multiple issue**.

Multiple Issue

- Multiple issue is a scheme whereby multiple instructions are launched in 1 clock cycle.
- There are two major ways to implement a multiple-issue processor
- The types differ in the division of work between the compiler and the hardware.
- They are
 - Static multiple issue-Compiler decides multiple issue before execution.
 - Dynamic multiple issue-Processor decides multiple issue before execution.

Static multiple issue is an approach to implementing a multiple-issue processor where many decisions are made by the compiler before execution.

Dynamic multiple issue is an approach to implementing a multiple-issue processor where many decisions are made during execution by the processor.

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

- Packaging instructions into issue slots:
 - It deals with the evaluation of the number of instructions and the type of instructions that can be issued in a given clock cycle.
 - This process is partially handled by the compiler in most of the static multiple issue processors.
 - In dynamic issue designs, it is normally dealt at the runtime by the processor.
- Dealing with data and control hazards:
 - In static issue processors, some or all of the consequences of data and control hazards are handled statically by the compiler.
 - Most of the dynamic issue processors attempt to ease at least some classes of hazards using hardware techniques operating at execution time.

Approaches to exploit ILP

The two separable approaches to exploit ILP are

- Dynamic, hardware intensive approach
 - Rely on hardware to help discover and exploit the parallelism dynamically.
 - Used in the desktop and server markets and in a wide range of processors

Example

- Pentium III and 4
- Athlon
- MIPS R10000/12000
- Sun Ultra SPARC-III
- PowerPC 603, G3, G4

- Alpha 21264
- Static, compiler intensive approach
 - Rely on software technology to find parallelism, statically at compile-time.
 - Broadly used in the embedded market than desktop/server market.

Example

- IA-64 architecture
- Intel's Itanium

Static Multiple Issue Processors:-

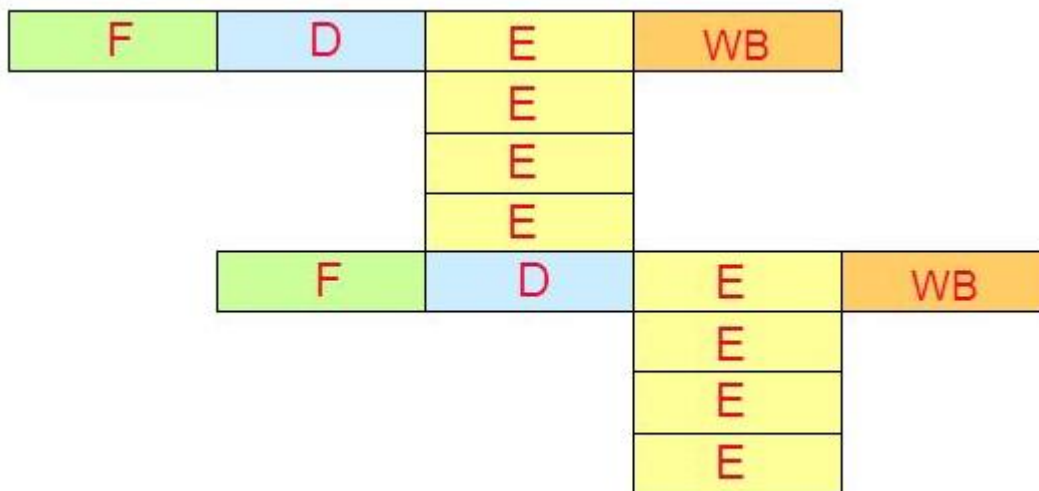
- Compiler specifies a set of instruction that executes together in a given clock cycle.
- Simple hardware, complex compiler.
- Issue Packet:-

The set of instructions that issues together in one clock cycle; the packet may be determined statically by the compiler or dynamically by the processor.

- Very Long Instruction Word (VLIW);-

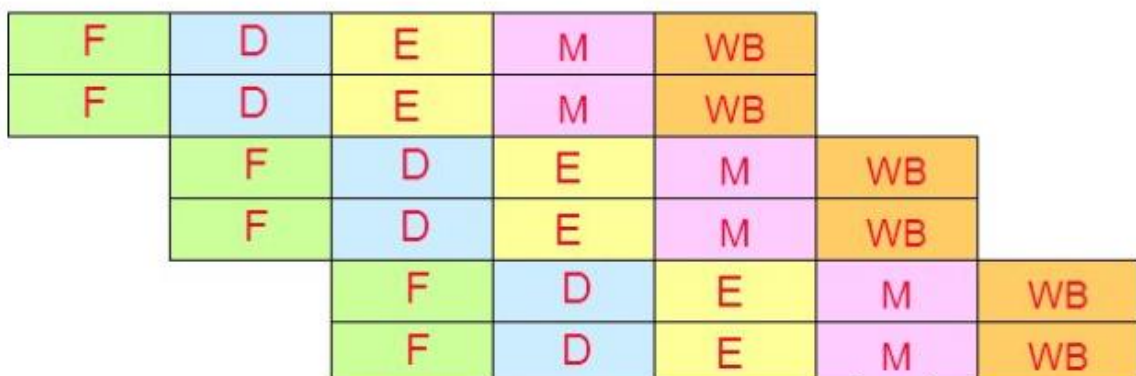
A style of instruction set architecture that launches many operations that are defined to be independent in a single wide instruction, typically with many separate opcode fields.

- Compiler detects and avoids hazards.



Dynamic multiple issue-Processors:-

- Hardware decides which instruction executes together.
- Complex hardware, simple compiler.
- The processor decides whether zero, one or more instructions can be issued in a given clock cycle.
- Dynamic multiple-issue processors are also known as superscalar processors, or simply superscalars.



- Many superscalars extend the basic framework of dynamic issue decisions to include dynamic pipeline scheduling.
- Dynamic pipeline scheduling chooses which instructions to execute in a given clock cycle while trying to avoid hazards and stalls.
- Let's start with a simple example of avoiding a data hazard. Consider the following code sequence:

```
lw      $t0, 20($s2)
addu    $t1, $t0, $t2
sub      $s4, $s4, $t3
slti     $t5, $s4, 20
```

- Even though the sub instruction is ready to execute, it must wait for the lw and addu to complete first, which might take many clock cycles if memory is slow.
- Dynamic pipeline scheduling allows such hazards to be avoided either fully or partially.

Dynamic pipeline scheduling:-

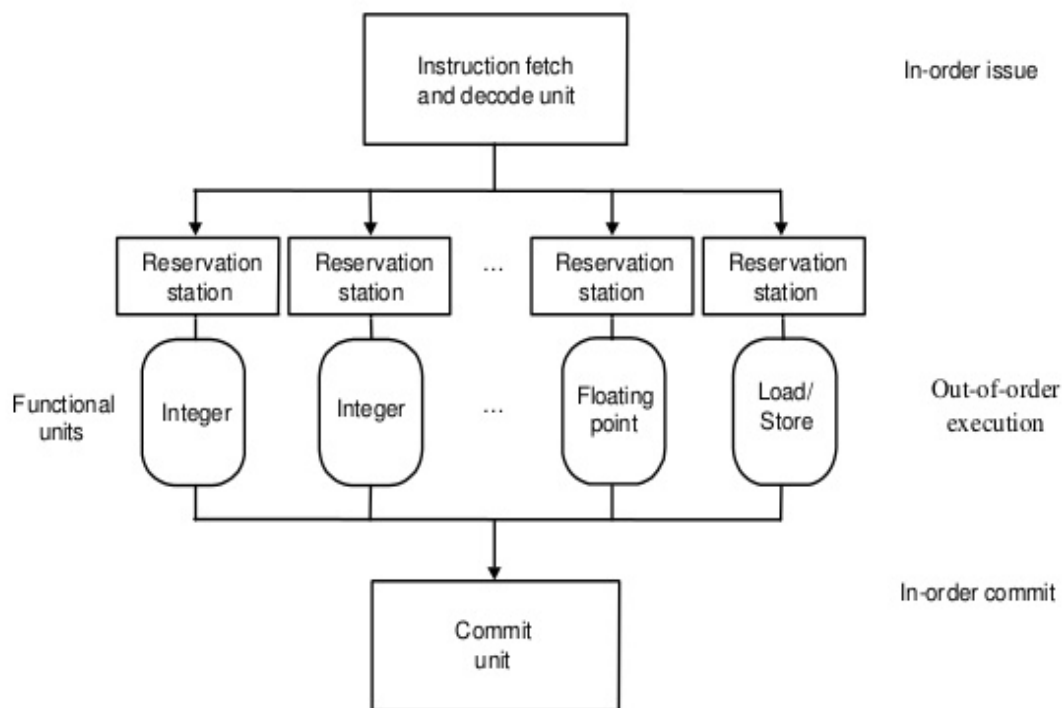
- It chooses which instruction to execute in a given clock cycle while trying to avoid hazards and stalls.
- It is divided into three major units,
 1. Instruction fetch and issue unit
 2. Multiple functional Units
 3. Commit unit
- The first unit fetches instructions, decodes them, and sends each instruction to a corresponding functional unit for execution.
- Each functional unit has buffers, called reservation stations, which hold the operands and the operation.

- As soon as the buffer contains all its operands and the functional unit is ready to execute, the result is calculated.
- When the result is completed, it is sent to any reservation stations waiting for this particular result as well as to the commit unit, which buffers the result until it is safe to put the result into the register file or, for a store, into memory.
- The buffer in the commit unit, often called the reorder buffer, is also used to supply operands, in much the same way as forwarding logic does in a statically scheduled pipeline. Once a result is committed to the register file, it can be fetched directly from there, just as in a normal pipeline.
- The combination of buffering operands in the reservation stations and results in the reorder buffer provides a form of register renaming
- When an instruction is issued,
 - It is copied to a reservation station for the appropriate functional unit. Any operands that are available in the register file or reorder buffer are also immediately copied into the reservation station. The instruction is buffered in the reservation station until all the operands and the functional units are available. For the issuing instruction, the register copy of the operand is no longer required, and if a write to that register occurred, the value could be overwritten.
 - If an operand is not in the register file or reorder buffer, it must be waiting to be produced by a functional unit. The name of the functional unit that will produce the result is tracked. When that unit eventually produces the result, it is copied directly into the waiting reservation station from the functional unit by-passing the registers.

These steps effectively use the reorder buffer and the reservation stations to implement register renaming.

The processor then executes the instructions in some order that preserves the data flow order of the program. This style of execution is called an out-of-order execution, since the instructions can be executed in a different order than they were fetched.

To make programs behave as if they were running on a simple in-order pipeline, the instruction fetch and decode unit is required to issue instructions in order, which allows dependences to be tracked, and the commit unit is required to write results to registers and memory in program fetch order. This conservative mode is called in-order commit.



4.2.1 ILP Terminology - Basic block

- Basic block is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit.
- The amount of parallelism available within a basic block is quite small.
- The instructions in Basic block are likely to depend on each other.

- The amount of overlap that can exploit within a basic block is likely to be much less than the average basic blocks size.

4.2.1.1 Methods to enhance performance of ILP

To obtain substantial performance enhancements, the ILP across multiple basic blocks are exploited.

Loop-level parallelism

- The simplest and most common way to increase the ILP is to exploit parallelism among iterations of a loop.
- This type of parallelism is often called loop-level parallelism.

Example

```
for (i=0; i<=999; i=i+1)
```

```
    x[i] = x[i] + y[i];
```

- Every iteration of the loop can overlap with any other iteration.
- There are a number of techniques for converting loop-level parallelism into instruction-level parallelism.
- Those techniques work by unrolling the loop either statically by the compiler or dynamically by the hardware.
- An important alternative method for exploiting loop-level parallelism is the use of SIMD in both vector processors and Graphics Processing Units (GPUs).

Techniques to convert LLP to ILP

There are a number of techniques for converting such loop-level parallelism into instruction-level parallelism.

Loop Unrolling:

- It is used for converting loop-level parallelism into instruction-level parallelism.
- Either the compiler or the hardware is able to exploit the parallelism inherent in the loop.

```
for(i=1; i<=1000; i=i+4)
```



```
{  
  
    x[i] = x[i] + y[i];  
  
    x[i+1] = x[i+1] + y[i+1];  
  
    x[i+2] = x[i+2] + y[i+2];  
  
    x[i+3] = x[i+3] + y[i+3];  
  
}
```

Such techniques work by unrolling the loop either

- Statically by the compiler or
- Dynamically by the hardware.

Vector instructions

- An important alternative method for exploiting loop-level parallelism is the use of **vector instructions**.
- A vector instruction operates on a sequence of data items.
- The above code sequence could execute in four instructions on some vector

Processors

- Two instructions to load the vectors x and y from memory.
- One instruction to add the two vectors.
- One instruction to store back the result vector.
 - Processors that exploit ILP have almost completely replaced vector-based processors.
 - Vector based processors are used in graphics, digital signal processing, and multimedia applications.

4.2.2 Data Dependences and Hazards

Determining how one instruction depends on another is critical to

- Determine how much parallelism exists in a program
- Determine how that parallelism can be exploited.

In order to exploit instruction-level parallelism, which instructions can be executed in parallel must be determined.

Parallel instructions

- If two instructions are parallel, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls, assuming the pipeline has sufficient resources.

Dependent instructions

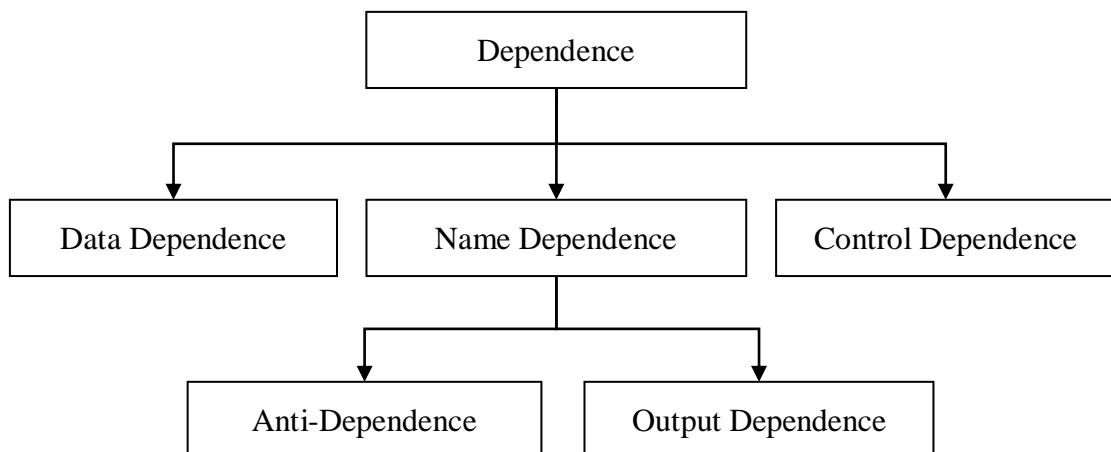
- If two instructions are dependent, they are not parallel and must be executed in order, although they may often be partially overlapped.

In both the cases, whether an instruction is dependent on another instruction must be determined.

Types of dependences

There are three different types of dependences:

1. Data dependences
2. Name dependences
3. Control dependences



4.2.2.1 Data dependences

Data dependences are also called true data dependences.

An instruction j is data dependent on instruction i if either of the following holds:

- Instruction i produces a result that may be used by instruction j,

Instruction i

Instruction j

or

- Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction I, which is called transitive dependence.

Instruction i

Instruction k

Instruction j



Example

Consider the following code sequence that increments a vector of values in memory (starting at 0(R1), and with the last element at 8(R2)), by a scalar in register F2.

```
Loop:    L.D      F0, 0(R1)      ; F0=array element
          ADD.D    F4, F0, F2     ; add scalar in F2
          S.D      F4, 0(R1)     ; store result
          DADDUI   R1, R1, #-8    ; decrement pointer 8 bytes
          BNE      R1, R2, LOOP   ; branch R1! =R2
```

The data dependence in the above code involve both the following part

- **Floating-point data part**

```
Loop:    L.D      F0,0(R1)      ;F0=array element
          
          ADD.D    F4,F0,F2     ;add scalar in F2
          
          S.D      F4,0(R1)     ;store result
```

- **Integer data part**

DADDIU	R1, R1,-8	; decrement pointer
	↓	
		; 8 bytes (per DW)
BNE	R1, R2, Loop	; branch R1! =R2

In both of the above dependent sequences, the arrows show

- Each instruction depending on the previous one
- Show the order that must be preserved for correct execution
- The arrow points from an instruction that must precede the instruction that the arrowhead points to.

Dependent Instructions

- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped.
- Executing the instructions simultaneously will cause a processor with pipeline interlocks to detect a hazard and stall, thereby reducing or eliminating the overlap.
- If data dependence caused a hazard in pipeline, then it is called a **Read After Write (RAW)** hazard.
- In a processor without interlocks that relies on compiler scheduling, the compiler cannot schedule dependent instructions in such a way that they completely overlap, since the program will not execute correctly.

Pipeline organization

- Dependences are a property of programs.
- The concern is to check whether a given dependence results in an actual hazard being detected and whether that hazard actually causes a stall are properties of the pipeline organization.

Property of Pipeline

The presence of dependence indicates potential for a hazard but the actual hazard and the length of any stall is a property of the pipeline.

Importance of the data dependencies

The importance of data dependence is that the dependence conveys the following three things

1. The possibility of a hazard
2. The order in which results must be calculated, and
3. An upper bound on how much parallelism can possibly be exploited.

Overcoming of data dependence

Dependence can be overcome in two different ways

- By maintaining the dependence but avoiding a hazard.
- By eliminating dependence by transforming the code.

Data Dependences through registers/memory

A data value may flow between instructions either through registers or through memory locations.

Dependences through registers

Dependences through registers are easy since the register names are fixed in the instructions

```
lw r10,10(r11)
```

```
add r12,r10,r8
```

just compare register names..

It gets more complicated when branches intervene and correctness concerns cause a compiler or hardware to be conservative.

Dependences through memory

Dependences that flow through memory locations are more difficult to detect. Since two addresses may refer to the same location but look different

For example, 100(R4) and 20(R6) may be identical memory addresses.

1.2.2.2 Name dependences

Name dependence occurs when two instructions use the same register or memory location, called a name, but there is no flow of data between the instructions associated with that name.

Since there is no value being transmitted between the instructions in name dependence is not a true dependence.

Types of name dependences

There are two types of name dependences between an instruction *i* that precedes instruction *j* in program order:

1. Antidependence

An Antidependence between instruction *i* and instruction *j* occurs when instruction *j* writes a register or memory location that instruction *i* reads.

The original ordering must be preserved to ensure that *i* reads the correct value.

2. Output dependence

An output dependence occurs when instruction *i* and instruction *j* write the same register or memory location.

The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction *j*.

Register renaming

Since a name dependence is not a true dependence, instructions involved in a name dependence can execute simultaneously or be reordered, if the name (register number or memory location) used in the instructions is changed so the instructions do not conflict.

The renaming that can be done for register operands, where it is called register renaming.

Register renaming can be done either statically by a compiler or dynamically by the hardware.

4.2.2.3 Data Hazards

A hazard is created whenever there is dependence between instructions, and they are close enough that the overlap caused by pipelining, or other reordering of instructions, would change the order of access to the operand involved in the dependence.

The hazards are named by the ordering in the program that must be preserved by the pipeline.

Instruction i

Instruction j

Program Order

Hardware and Software techniques must preserve program order, a order instructions would execute in if executed sequentially one at a time as determined by original source program

HW/SW goal

- To exploit parallelism by preserving program order only where it affects the outcome of the program.

Classification of Data Hazards

Data Hazards are classified into three types depending on the order of read and write accesses in the instructions.

They are,

1. **RAW (read after write)**
2. **WAW(Write after Write)**
3. **WAR(Write after Read)**

Consider the two instructions i and j with i occurring before j in program order.

1. RAW (read after write)

- j tries to read a source before i writes it, so j incorrectly gets the old value. This hazard is the most common type and corresponds to true data dependence.
- Program order must be preserved to ensure that j receives the value from i.
- In the simple common five-stage static pipeline a load instruction followed by an integer ALU instruction that directly uses the load result will lead to a RAW hazard.

2. WAW(Write after Write)

WAW: j tries to write an operand before it is written by i . The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by j in the destination.

WAW hazards are present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled.

The classic five-stage integer pipeline writes a register only in the WB stage and avoids this class of hazards

WAW hazards can also between a short integer pipeline and a longer floating-point pipeline.

Example

A floating point multiply instruction that writes F4, shortly followed by a load of F4 could yield a WAW hazard, since the load could complete before the multiply completed.

Instruction_J writes operand before Instruction_I writes it.

```

      ↩ I: sub r1, r4, r3
      J: add r1, r2, r3
      K: mul r6, r1, r7
  
```

This is called “output dependence” by compiler writers.

This also results from the reuse of name “r1”. If output -dependence caused a hazard in the pipeline, called a Write After Write (WAW) hazard.

3. WAR(Write after Read)


WAR: j tries to write a destination before it is read by i , so i incorrectly gets the new value. This hazard arises from antidependence.

WAR hazards cannot occur in most static issue pipelines even deeper pipelines or floating point pipelines because all reads are early (in ID) and all writes are late (in WB).

A WAR hazard occurs either when there are some instructions that write results early in the instruction pipeline, and other instructions that read a source late in the pipeline or when instructions are reordered.

Instr_J writes operand before Instr_I reads it


```
    I: sub r4, r1, r3
    J: add r1, r2, r3
    K: mul r6, r1, r7
```



It is called “anti-dependence” by compiler writers.

This results from reuse of the name “r1”. If anti-dependence caused a hazard in the pipeline, then it is called a Write After Read (WAR) hazard.

RAR (Read after Read)

RAR is not a hazard since reading any number of times simultaneously does not create issues.

4.2.2.4 Control dependences

A control dependence determines the ordering of an instruction, *i*, with respect to a branch instruction so that the instruction *i* is executed in correct program order and only when it should be.

Every instruction, except for those in the first basic block of the program, is control dependent on some set of branches, and, in general, these control dependences must be preserved to preserve program order.

Example

Consider the following code segment

```
    if p1 {
        S1;
    }
    if p2 {
        S2;
    }
```

S1 is control dependent on p1, and

S2 is control dependent on p2 but not on p1.

Control dependences constraints

The Two constraints imposed by control dependences are

1. An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch.

Example

We cannot take an instruction from the then portion of if-statement and move it before if-statement.

2. An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch.

Example

We cannot take a statement before if-statement and move it into the then portion.

Preserving Control dependence

Control dependence is preserved by two properties in a simple pipeline

1. Instructions execute in program order
 - Ensures that an instruction that occurs before a branch is executed before the branch.
2. Detection of control or branch hazards
 - Ensures that an instruction that is control dependent on a branch is not executed until the branch direction is known.

When processors preserve strict program order, they ensure that control dependences are also preserved.

Example:

DADDU R2, R3, R4

BEQZ R2, L1

LW R1, 0(R2)

L1:

It is easy to see that if we do not maintain the data dependence involving R2, we can change the result of the program.

Less obvious is the fact that if we ignore the control dependence and move the load instruction before the branch, the load instruction may cause a memory protection exception.

To allow us to reorder these instructions, ignore the exception when the branch is taken.

Ignoring Control Dependence

Control dependence need not be preserved; instead the following two properties are critical to program correctness

1. exception behavior
2. data flow

Preserving the exception behavior

This means that any changes in the ordering of instruction execution must not change how exceptions are raised in the program.

Example

Consider this code sequence:

```
DADDU    R2,R3,R4
BEQZ     R2,L1
LW       R1,0(R2)
```

L1:

Problem with moving LW before BEQZ

If the data dependence involving R2 is not maintained, the result of the program can be changed.

If we ignore the control dependence and move the load instruction before the branch, the load instruction may cause a memory protection exception.

Preserving the data flow

The **data flow** is the actual flow of data values among instructions that produce results and those that consume them. Branches make the data flow dynamic, since they allow the source of data for a given instruction to come from many points.

Example

Consider the following code fragment:

DADDU R1, R2, R3

BEQZ R4, L

DSUBU R1, R5, R6

L: ...

OR R7, R1, R8

- In the above example ,
 - The value of R1 used by OR instruction depends on whether the branch is taken or not.
 - OR instruction is data dependent on both DADDU and DSUBU
- If the branch is taken the value of R1 computed by DADDU will be used by OR.
- If the branch is not taken, the value of R1 computed by DSUBU will be used by OR.

Speculation

The violation of the control dependence cannot affect either the exception behavior or the data flows are determined by considering the following code sequence.

DADDU R1, R2, R3

BEQZ R12, skipnext

DSUBU R4, R5, R6

DADDU R5, R4, R9

Skipnext: OR R7, R8, R9

Liveness – The property of whether a value will be used by an upcoming instruction is called liveness.

If R4 was unused after the instruction labeled skipnext, then changing the value of R4 just before the branch would not affect the data flow since R4 would be dead in the code region after skipnext.

- Assume R4 is dead (rather than live) after skipnext.
- We can execute DSUBU before BEQZ since
 - R4 could not generate an exception.
 - The data flow cannot be affected.

- This type of code scheduling is called speculation.
 - The compiler is betting on the branch outcome. In this case, the bet is that the branch is usually not taken.

4.3 PARALLEL PROCESSING CHALLENGES

1. Concurrency

- Concurrency is a property of a system representing the fact that multiple activities can be executed at the same time
- If an algorithm cannot be divided into groups of operations that can execute concurrently, performance improvements due to parallelism cannot be achieved, and any processors after the first will be of limited use in accelerating the algorithm
- To a large extent, different problems inherently have differing amounts of concurrency. For most problems, developing an algorithm that achieves its maximal concurrency requires a combination of cleverness and experience from the programmer
- The three fundamental ways of improving the performance of the application using concurrency:
 1. **Reduce latency:** A unit of work is executed in shorter time by subdivision into parts that can be executed concurrently
 2. **Hide latency:**
 - Multiple long-running tasks are executed together by the underlying system
 - Effective when the tasks are blocked because of external resources they must wait upon, such as disk or network I/O operations
 3. **Increase throughput**
 1. By executing multiple tasks concurrently, the general system throughput can be increased
 2. Also speeds up independent sequential tasks that have not been specifically designed for concurrency yet.

2. Data Distribution

- Another challenge in parallel programming is the distribution of a problem's data
- Most conventional parallel computers have a notion of data locality

- Implies that some data will be stored in memory that is “closer” to a particular processor and can therefore be accessed much more quickly
- Data locality may occur
 - due to each processor having its own distinct local memory—as in a distributed memory machine
 - due to processor-specific caches as in a shared memory system.
- Due to the impact of data locality, a parallel programmer must pay attention to where data is stored in relation to the processors that will be accessing it
- The more local the values are, the quicker the processor will be able to access them and complete its work
- It should be evident that distributing work and distributing data are tightly coupled, and that an optimal design will consider both aspects together

3. Inter-process Communication

- Inter-process communication (IPC) is a set of programming interfaces that allow a programmer to coordinate activities among different program processes that can run concurrently in an operating system.
 - Allows a program to handle many user requests at the same time
- **Factors to Consider**
 - **Cost of communications**
 - Inter-processor communication virtually always implies overhead.
 - Machine cycles and resources that could be used for computation are instead used to package and transmit data.
 - **Latency vs. Bandwidth**
 - **Latency** is the time it takes to send a minimal (0 byte) message from point A to point B. Commonly expressed as microseconds.
 - **Bandwidth** is the amount of data that can be communicated per unit of time. Commonly expressed as megabytes/sec or gigabytes/sec.
 - Sending many small messages can cause latency to dominate communication overheads.
 - More efficient to package small messages into a larger message, thus increasing the effective communications bandwidth.

- **Visibility of communications**
 - With the Message Passing Model, communications are explicit and generally quite visible and under the control of the programmer.
 - With the Data Parallel Model, communications often occur transparently to the programmer, particularly on distributed memory architectures. The programmer may not even be able to know exactly how inter-task communications are being accomplished
- **Synchronous vs. asynchronous communications**
 - Synchronous communications are often referred to as **blocking** communications since other work must wait until the communications have completed
 - Asynchronous communications are often referred to as **non-blocking** communications since other work can be done while the communications are taking place
 - Interleaving computation with communication is the single greatest benefit for using asynchronous communications.
- **Scope of communications**
 - Knowing which tasks must communicate with each other is critical during the design stage of a parallel code.
- **Efficiency of communications**

4. Load Balancing

- Load balancing refers to the practice of distributing approximately equal amounts of work among tasks so that all tasks are kept busy all of the time.
- Load balancing is important to parallel programs for performance reasons.
- **Steps for achieving**
 - **Equally partition the work each task receives**
 - For array/matrix operations where each task performs similar work, evenly distribute the data set among the tasks.
 - For loop iterations where the work done in each iteration is similar, evenly distribute the iterations across the tasks.
 - If a heterogeneous mix of machines with varying performance characteristics is being used, be sure to use some type of

performance analysis tool to detect any load imbalances. Adjust work accordingly

- **Use dynamic work assignment**

- Certain classes of problems result in load imbalances even if data is evenly distributed among tasks:
 - Sparse arrays - some tasks will have actual data to work on while others have mostly "zeros"
 - Adaptive grid methods - some tasks may need to refine their mesh while others don't
 - N-body simulations - where some particles may migrate to/from their original task domain to another task's; where the particles owned by some tasks require more work than those owned by other tasks
- When the amount of work each task will perform is intentionally variable, or is unable to be predicted, it may be helpful to use a **scheduler - task pool** approach. As each task finishes its work, it queues to get a new piece of work
- It may become necessary to design an algorithm which detects and handles load imbalances as they occur dynamically within the code

5. Implementation and Debugging

- Programmers often implement parallel algorithms by creating a single executable that will execute on each processor
- The program is designed to perform different computations and communications based on the processor's unique ID to ensure that the work is divided between instances of the executable.
 - Referred to as the Single Program, Multiple Data (SPMD) model
 - Attractiveness stems from the fact that only one program must be written
 - Alternative is to use the Multiple Program, Multiple Data (MPMD) model
 - Several cooperating programs are created for execution on the processor set
 - In either case, the executables must be written to cooperatively perform the computation while managing data locality and communication
 - They must also maintain a reasonably balanced load across the processor set

- It should be clear that implementing such a program will inherently require greater programmer effort than writing the equivalent sequential program

6. Speed up Challenge

- To get good speed-up on a multiprocessor while keeping the problem size fixed is harder than getting good speed-up by increasing the problem size
- **Strong Scaling** means measuring speed-up while keeping the problem size fixed
- **Weak Scaling** means that the program size grows proportionally to the increase in the number of processors.

- **Amdahl's Law** says

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

- **Modified Amdahl's Law** in terms of speed-up versus the original execution time:

$$\text{Speed-up} = \frac{\text{Execution time before}}{(\text{Execution time before} - \text{Execution time affected}) + \frac{\text{Execution time affected}}{100}}$$

- Assuming Execution time before = 1 and
Execution time affected = fraction of time

$$\text{Speed-up} = \frac{\text{Execution time before}}{(1 - \text{Fraction time affected}) + \frac{\text{Fraction time affected}}{100}}$$

Example

1. How will you achieve a speed-up of 90 times faster with 100 processors? What percentage of the original computation can be sequential?

Solution:

Speed-up to be achieved = 90

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

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%.

2. 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. What speed-up do

you get with 10 versus 100 processors? Next, calculate the speed-ups assuming the matrices grow to 100 by 100.c

Solution:

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 $110t$, the execution time for 10 processors is

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

$$\text{Execution time affected improvement} = (100t/10) + 10t = 20t$$

So the speed-up with 10 processors is $110t/20t = 5.5$.

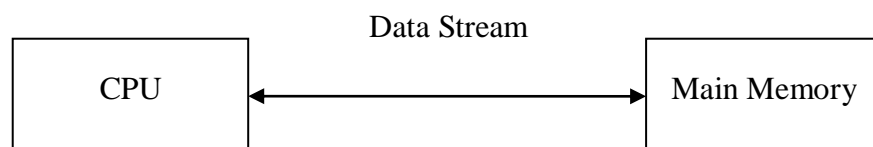
The execution time for 100 processors is

$$\text{Execution time after improvement} = (100t/100) + 10t = 11t$$

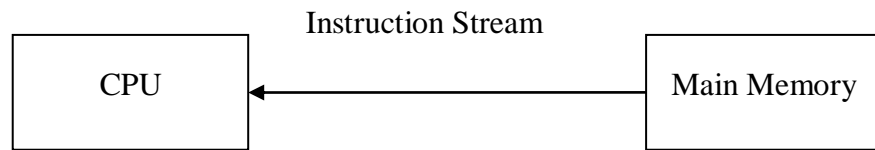
So the speed-up with 100 processors is $110t/11t = 10$.

4.4 FLYNN'S CLASSIFICATION

- First proposed by Michael J. Flynn in 1966
- Flynn uses the stream concept for describing a machine's structure
 - Stream – It refers to sequence or flow of either instructions or data operated on by the computer.
 - Types
 - Data Stream – The flow of operands between processor and memory in bi-directional manner.



- Instruction Stream – the flow of instructions from main memory to CPU



- Classification of parallel computer architectures that are based on the number of concurrent instruction and data streams

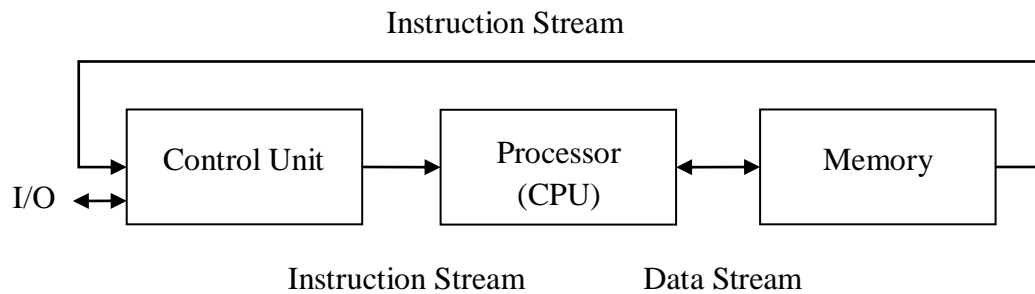
Categories

- SISD- Single **I**nstruction **S**ingle **D**ata Stream
- SIMD- Single **I**nstruction **M**ultiple **D**ata Stream
- MISD- **M**ultiple **I**nstruction **S**ingle **D**ata Stream
- MIMD- **M**ultiple **I**nstruction **M**ultiple **D**ata Stream

Data Stream	Instruction Stream		
		Single Instruction	Multiple Instruction
	Single Data	SISD	MISD
	Multiple Data	SIMD	MIMD

4.4.1 SISD- Single **I**nstruction **S**ingle **D**ata Stream

- A Single computer which uses no parallelism in either the instruction or data stream is called SISD.
- Single control unit fetches single instruction from memory then the control unit generates appropriate control signals to direct single processing unit to operate on single data stream.

**Example**

- IBM 704
- VAX
- CRAY - 1

Limitation

- Low level of parallelism

4.4.2 SIMD- Single Instruction Multiple Data Stream

- They have multiple processing/execution units and one control unit. Therefore all processing/execution units are supervised by single control unit. All processing element received same instruction from control unit but operate on different data streams.
- The concept of achieving parallelism by performing the same operation on independent data is called data level parallelism.
- Dedicated to array processing machines.

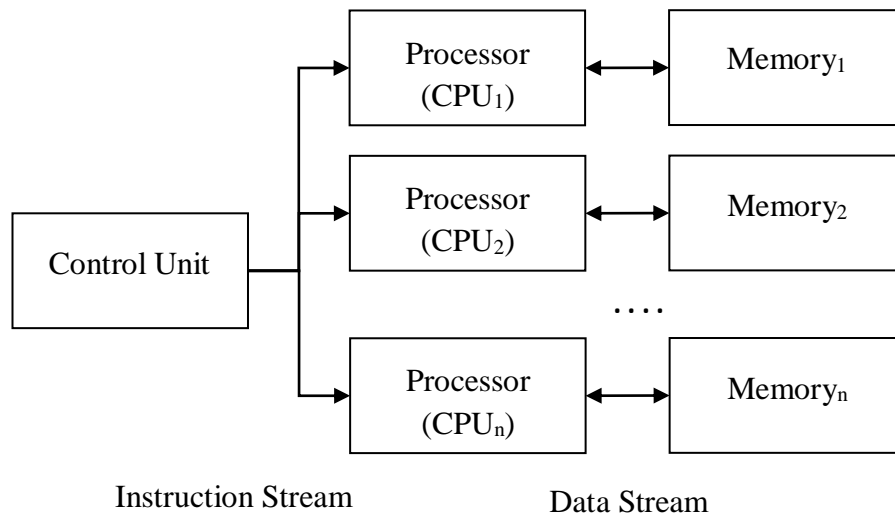
Characteristics of SIMD system:-

- Single machine instruction that controls simultaneous execution.
- Allow no of processing elements to perform in lock-step basis.
- Vectors or array processors are used for instruction execution.

Advantages of SIMD system:-

- Reduces the cost of Control unit bandwidth and space.
- It needs only one copy of code that is being simultaneously executed.

- This system in X86 is used for multimedia extension.



Example

- ILLIAC-IV
- MPP
- CM-2
- STARAN

4.4.3 MISD- Multiple Instruction Single Data Stream

- There are n number of processing units each receiving distinct instructions operating over the same data stream and its derivatives.
- The result of one processor becomes the input of next processor in micro pipe.
- All processing units are interacting with common shared memory for organization of single data stream.

Characteristics of MISD system:-

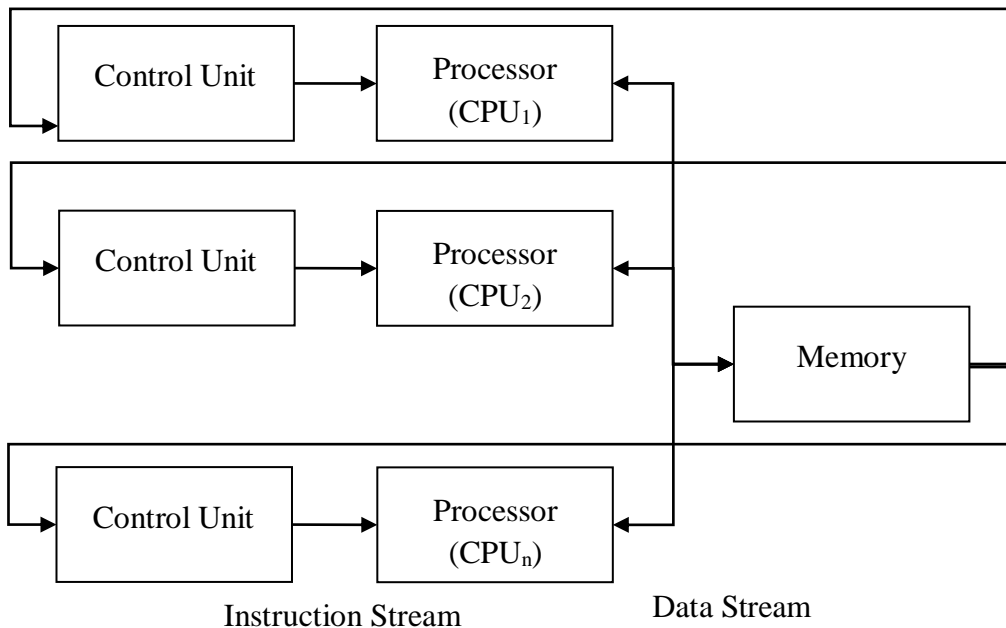
- Each processor executes a different sequence of instruction in single data stream.
- Each processing unit has associated data memory.

Example

- Pipeline Architecture

Limitation

- Low level of parallelism
- High bandwidth required
- High complexity



4.4.4 MIMD- Multiple Instruction Multiple Data Stream

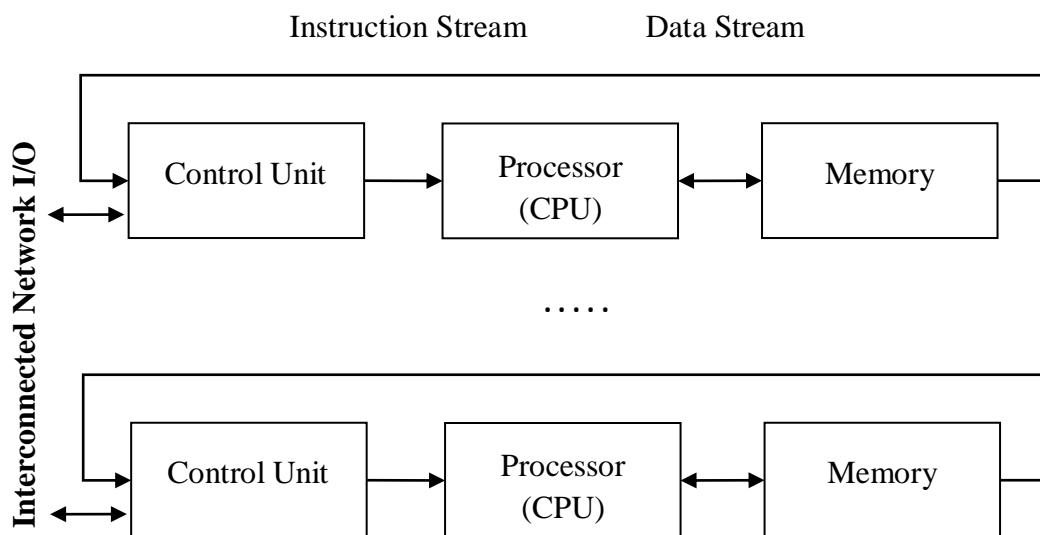
- MIMD Computer implies interactions among the multiple processors because all memory streams are derived from same data space shared by all processors.
- The Processors work on their own data with their own instructions.
- Tasks executed by different processors can start or finish at different time, but run asynchronously. This classification is actually recognizes parallel computer.

Characteristics of MIMD system:-

- Each instruction operates on different data.
- Each processing elements are associated data memory.

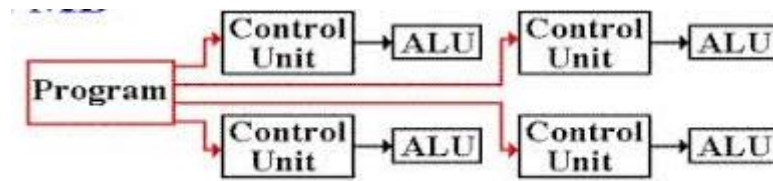
Example

- Distributed Computing Systems
- IBM 370/168M
- CRAY XMP



4.4.5 SPMD - Single Program Multiple Data

- SPMD was proposed first in 1983 by Michel Auguin and François Larbey in the OPSILA parallel computer. By the late 1980s, there were many distributed computers with proprietary message passing libraries. The first SPMD standard was PVM. The current de facto standard is MPI.
- In computing, SPMD (single program, multiple data) is a technique employed to achieve parallelism;
- It is a subcategory of MIMD. Tasks are split up and run simultaneously on multiple processors with different input in order to obtain results faster.
- SPMD is the most common style of parallel programming



Example

- Titanium
- MPI

Advantages of SPMD

- Data Locality
- Structured Parallelism
- Simple runtime implementation

Limitation

- Difficult to write divide and conquer problems
- Hard to get desired speed up

4.4.6 Vector Architecture

- An older interpretation of SIMD is called a vector architecture which has been closely identified with Cray Computers.
- It is again a great match to problems with lots of data-level parallelism.
- 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 philosophy 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, and then write the results back to memory.
- A key feature of vector architectures is a set of vector registers. Thus, a vector architecture might have 32 vector registers, each with 64 64-bit elements.

Properties of Vector instructions

Vector instructions have several important properties compared to conventional instruction set architectures, which are called scalar architectures in this context:

- A single vector instruction is equivalent to executing an entire loop. The time required for instruction fetch and decode bandwidth is dramatically reduced.
- In a vector instruction, the computation of each result in the vector is independent of the computation of other results in the same vector, so hardware does not have to check for data hazards within a vector instruction. This reduces power consumption too.
- Vector architectures and compilers help to write efficient applications which have data-level parallelism.
- 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.
- As an entire loop is replaced by a vector instruction whose behavior is predetermined, control hazards that would normally arise from the loop branch are non-existent.

For these reasons, vector operations can be made faster than a sequence of scalar operations on the same number of data items, and designers are motivated to include vector units if the application domain can use them frequently.

4.5 HARDWARE MULTITHREADING**Idea**

- Allows multiple threads to share the functional units of a single processor in an overlapped fashion. To allow this sharing the processor must duplicate the independent state of each thread. i.e) each thread will have separate copy of register file and program counter.

Use

- Promote utilization of existing hardware resources

Need

- To tolerate latency (initial motivation)
 - Latency of memory operations, dependent instructions, branch resolution
 - By utilizing processing resources more efficiently
- To improve system throughput

- By exploiting thread-level parallelism
- By improving superscalar processor utilization
- To reduce context switch penalty

Three implementations

- Coarse-grained Multi Threading
- Fine-grained Multi Threading
- Simultaneous Multi Threading (SMT)

4.5.1 Coarse-grained Multi Threading

Idea

It won't switch out the executing thread until it reaches a situation that triggers a switch. This situation occurs when instruction execution reaches either a long latency operation or explicit additional switch operation.

When a thread is stalled due to some event, switch to a different hardware context

- Switch-on-event multithreading

Advantages

- Reduces the penalty of high cost stalls.
- Less likely slow down the execution of individual thread.
- Relieves need to have very fast thread-switching
- Doesn't slow down thread, since instructions from other threads issued only when the thread encounters a costly stall.

Disadvantages

- Difficult to overcome throughput losses due to shorter stalls.
- When stall occurs, pipeline should be emptied.

4.5.2 Fine-grained Multi-Threading

Idea

- Switches between thread after every instruction. Such that no two instructions from the thread are in the pipeline concurrently.

- It makes interleaved execution of multiple threads at same time. The interleaving is done in round robin fashion on every clock cycle.
- Improves pipeline utilization by taking advantage of multiple threads

Advantages

- No need for dependency checking between instructions (only one instruction in pipeline from a single thread)
- No need for branch prediction logic
- Bubble cycles used for executing useful instructions from different threads
- Improved system throughput, latency tolerance, utilization
- It hides throughput losses due to short and long stalls.

Disadvantages

- Extra hardware complexity: multiple hardware contexts, thread selection logic
- Reduced single thread performance (one instruction fetched every N cycles)
- Resource contention between threads in caches and memory
- Dependency checking logic between threads remains (load/store)

4.5.3 Simultaneous Multi Threading (SMT)

SMT is a variation on multithreading that uses resources of a multiple-issue, dynamically scheduled processors to exploit TLP at the same time it exploits ILP i.e., convert thread-level parallelism into more ILP. It shares the functional units dynamically and flexibly between multiple threads so that no white boxes which are idle.

It exploits the following features of modern processors:

- Multiple functional units
 - Modern processors typically have more functional units available than a single thread can utilize
- Register renaming and dynamic scheduling
 - Multiple instructions from independent threads can co-exist and co-execute

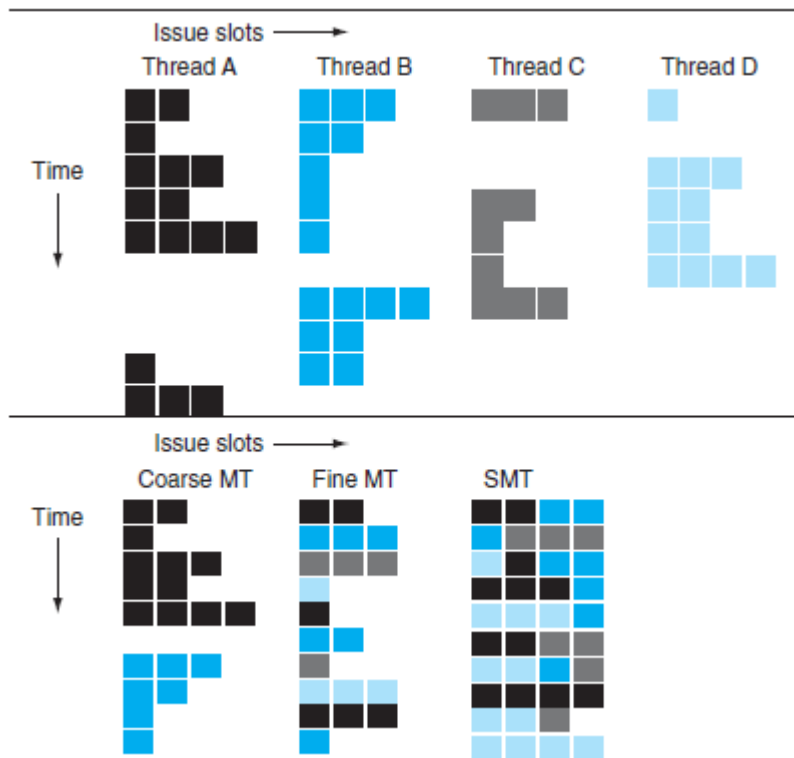
Advantages of SMT:-

- More functional unit, parallelism available then single threading.
- Register renaming and dynamic scheduling are used so that does not switch resource for every cycle.
- Multiple instructions from independent threads can be issued without regard to dependences among them.

Example:-

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.



4.5.3.1 Design Challenges in SMT

- Impact of fine-grained scheduling on single thread performance
 - A preferred thread approach sacrifices neither throughput nor single-thread performance?
 - Unfortunately, with a preferred thread, the processor is likely to sacrifice some throughput, when preferred thread stalls
 - Reason
 - Pipeline is less likely to have a mix of instructions from several threads resulting in greater probability that either empty slots or a stall will occur.

Design challenges

The Design challenges of SMT processor include the following

- Larger register file needed to hold multiple contexts

- Not affecting clock cycle time, especially in
 - Instruction issue - more candidate instructions need to be considered
 - Instruction completion - choosing which instructions to commit may be challenging
- Ensuring that cache and TLB conflicts generated by SMT do not degrade performance

Observation

The following two observations that are made with respect to these problems are

1. Potential performance overhead due to multithreading is small
 2. Efficiency of current superscalar is low with the room for significant improvement.
- Works well if
 - Number of compute intensive threads does not exceed the number of threads supported in SMT
 - Threads have highly different characteristics (e.g. one thread doing mostly integer operations, another mainly doing floating point operations)
 - Does not work well if
 - Threads try to utilize the same function units
 - Assignment problems are found as follows:
 - a dual processor system, each processor supporting 2 threads simultaneously (OS thinks there are 4 processors)
 - To compute intensive application processes might end up on the same processor instead of different processors (OS does not see the difference between SMT and real processors!)

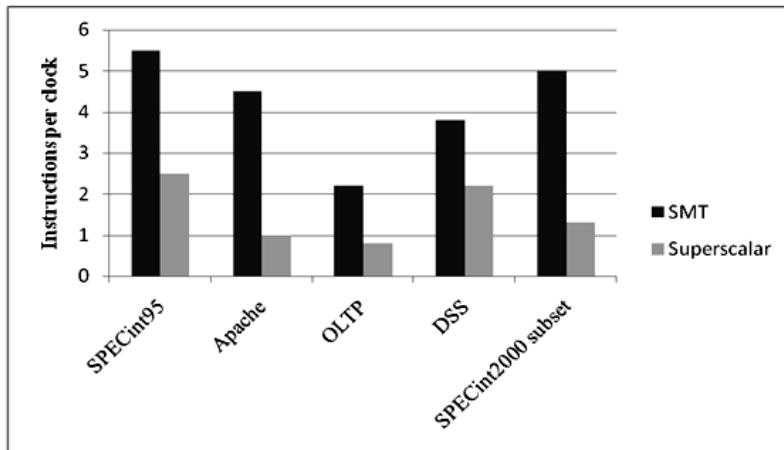
4.5.3.2 Potential performance advantage from SMT

The following table shows the features of an aggressive SMT design incorporated into an aggressive superscalar for the evaluation of an SMT extension starts with an aggressive superscalar that has roughly double the capacity of existing superscalar processor.

Processor characteristic	Capability
Integer functional units	6 (include 4 loads/stores per cycle)
Pipeline depth	9 stages
Floating-point functional units	4
Instruction queues	32
Renaming registers	100 each for integer and floating point
Commit capability	Up to 12 instructions per cycle
TLB	128 entries each for instructions and data
Primary instruction cache	128 KB, 2-way set associative, single ported, 2-cycle fill penalty, 32 outstanding misses
Primary data cache	128 KB, 2-way set associative, dual ported, 2-cycle fill penalty
L2 cache	16 MB, direct-mapped, 20-cycle latency, fully pipelined
L1-L2 bus/refill	256 bits wide, 2-cycle latency
Store buffer	32 entries
Memory System	90-cycle latency, fully pipelined
Branch-target buffer	1K entries, 4-way set associative
Hardware context for SMT	8
Fetch Policy	8 instructions per clock.

Performance advantage

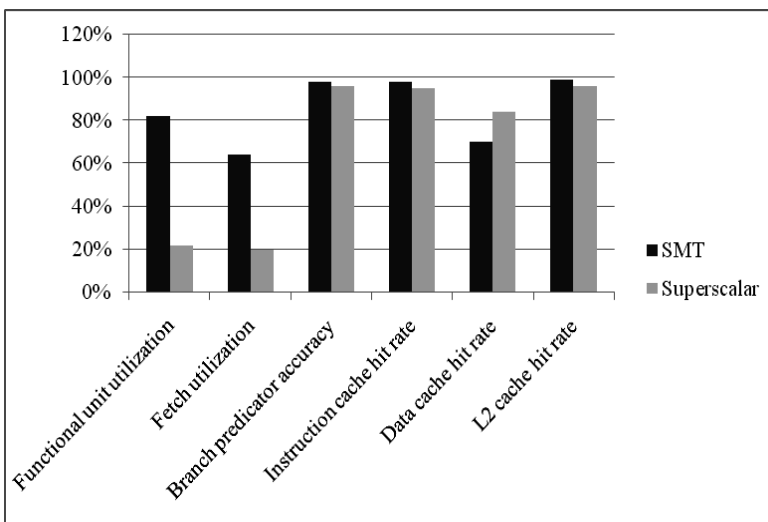
The following diagram shows the advantage in throughput, which is measured as instructions per clock, of SMT with eight contexts to the superscalar processor.



Comparison of the SMT processor to the base superscalar processor

The SMT processor are compared to the base superscalar processor in several key measures

- Utilization of functional units
- Utilization of fetch units
- Accuracy of branch predictor
- Hit rates of primary caches
- Hit rates of secondary caches



Comparison of the SMT processor to the base superscalar processor

Performance Improvement

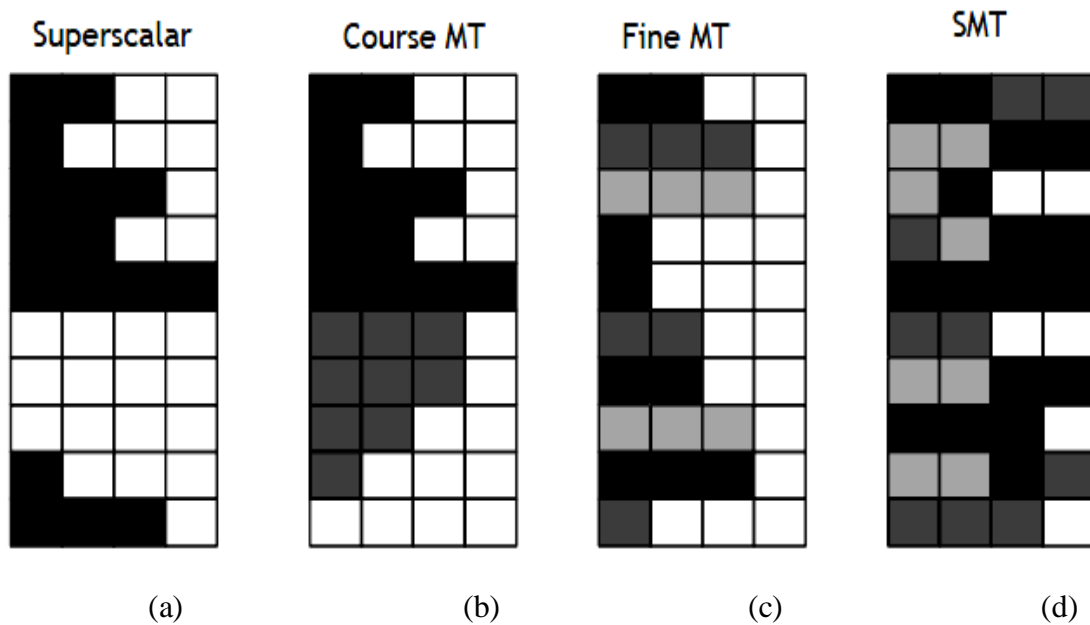
The key to maximize SMT performance is to share the following

- Issue slots
- Functional Units
- Renaming registers

4.5.4 Illustration

The following diagram illustrates the differences in a processor's ability to exploit the resources of a superscalar for the following processor configuration

- A superscalar processor with no multithreading support
- A superscalar processor with coarse-grain multithreading
- A superscalar processor with fine-grain multithreading
- A superscalar processor with simultaneous multithreading (SMT)



- Horizontal dimension represents the instruction issue capability in each clock cycle.
- Vertical dimension represents a sequence of clock cycles.
- Empty slots indicate that the corresponding issue slots are unused in that clock cycle.

Superscalar processor with no multithreading support

The use of issue slots is limited by a lack of ILP. Stalls such as an instruction cache miss leave the entire processor idle.

Superscalar processor with coarse-grain multithreading

The long stalls are partially hidden by switching to another thread that uses the resources of the processor.

- Reduced the number of completely idle clock cycles.
- But, the ILP limitations still lead to idle cycles.

Superscalar processor with fine-grain multithreading

The interleaving of threads eliminates fully empty slots.

The ILP limitations still lead to a significant number of time slots within individual clock cycles because only one thread issues instructions in a given clock cycle.

Superscalar processor with simultaneous multithreading (SMT)

The thread-level parallelism (TLP) and instruction-level parallelism (ILP) are exploited simultaneously with multiple threads using the issue slots in a single clock cycle.

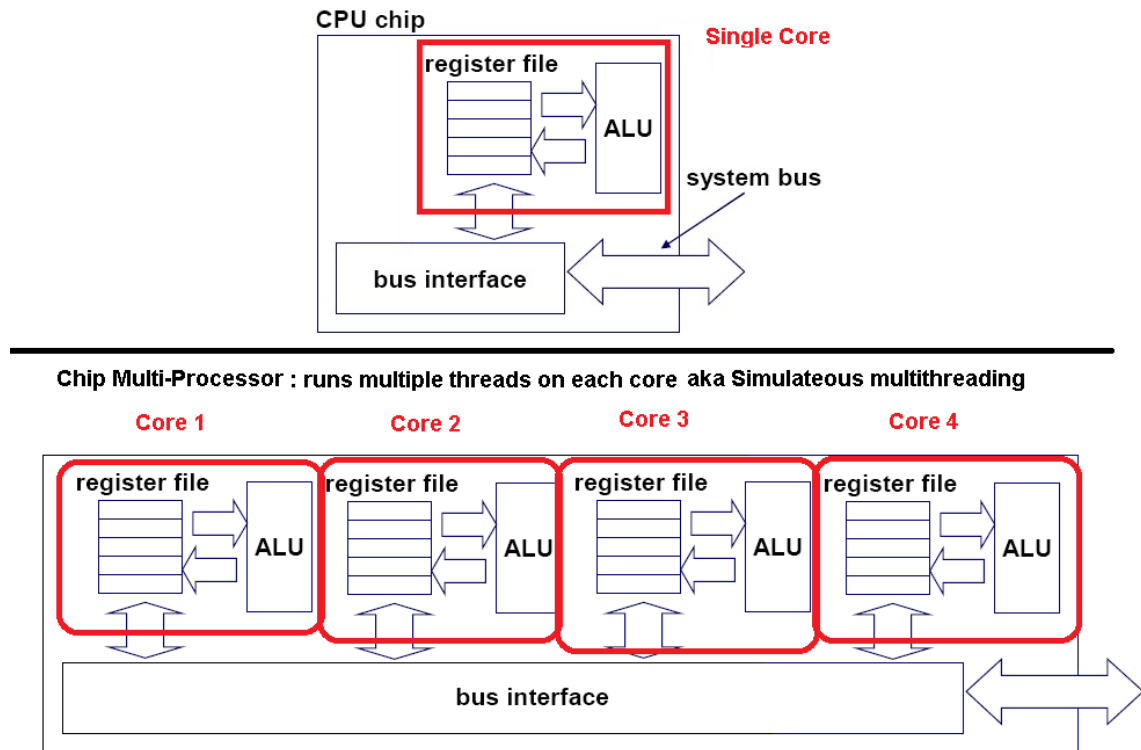
The issue slot usage is limited by the following factors

- Imbalances in the resource needs
- Resource availability over multiple threads
- Number of active threads considered
- Finite limitations of buffer
- Ability to fetch enough instruction from multiple threads
- Practical limitations of what instructions combinations can issue from one thread and multiple threads.

4.6 MULTICORE PROCESSORS AND OTHER SHARED MEMORY MULTIPROCESSORS

4.6.1 MULTICORE PROCESSORS:-

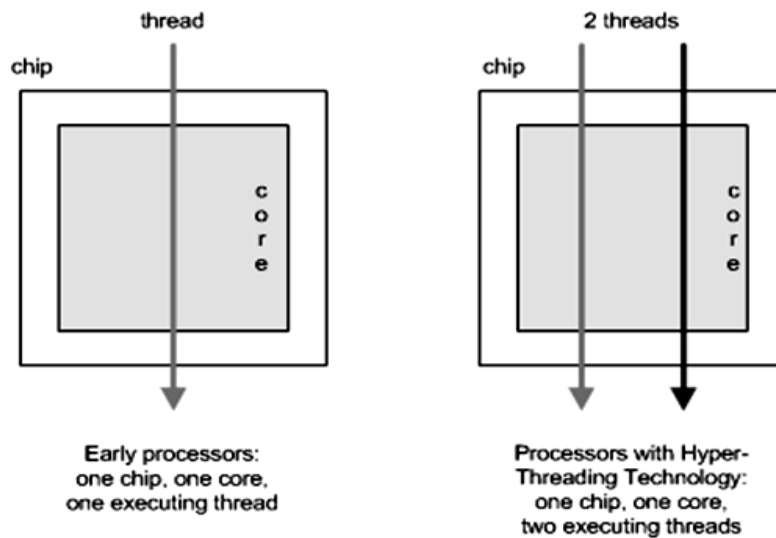
Single core processor vs. multicore processor:-



A multi-core processor is an integrated circuit to which two or more processor have attached for enhance performance and is is more efficient simultaneous processing of multiple tasks. i.e) single physical processor contains the core logic of two or more processors.

- Multiple cores can run multiple instructions at the same time, increasing overall speed
- Implements multiprocessing in a single physical package

The following figure shows how this appears in relation to previous technologies.



Multi-Core processors have multiple execution cores on a single chip

In this design, each core has its own execution pipeline. And each core has the resources required to run without blocking resources needed by the other software threads. It is classified according to number of processors it contains,

Multi-core processors may have

- Two cores
 - Dual-core CPUs
 - Example AMD Phenom II X2 and Intel Core Duo
- Three cores
 - Tri-core CPUs
 - Example AMD Phenom II X3
- Four cores
 - Quad-core CPUs
 - Example AMD Phenom II X4, Intel's i5 and i7 processors
- Six cores
 - Hexa-core CPUs
 - Example AMD Phenom II X6 and Intel Core i7 Extreme Edition 980X

- Eight cores
 - Octo-core CPUs
 - Example Intel Xeon E7-2820 and AMD FX-8350
- Ten cores
 - Example, Intel Xeon E7-2850) or more

Applications

Multi-core processors are widely used across many application domains including

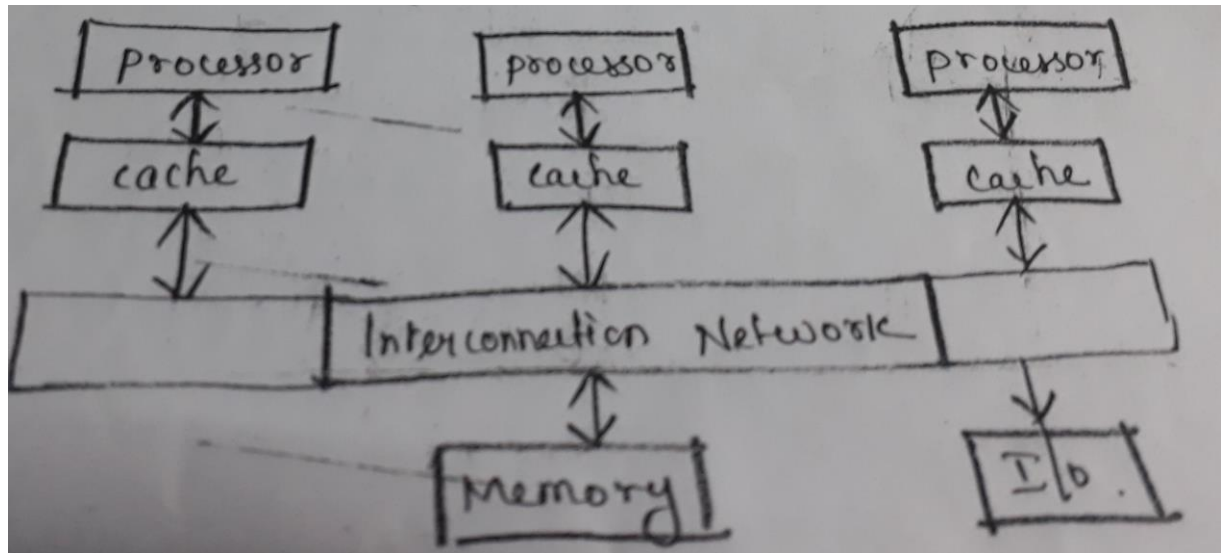
- General-purpose
- Embedded
- Network
- Digital Signal Processing (DSP)
- Graphics

Multicore processor implements multiprocessing in a single physical package that can be implemented by

- **Distributed shared memory intercore message passing method**
- **Symmetric shared memory intercore message passing method**

Symmetric shared memory intercore communication method:-

- A parallel processor with single physical address space is called shared memory multiprocessor.
- Processors communicate through shared variables in memory.
- All processors can access any memory location so that independent jobs can run in their own virtual address space, if they all share same physical address space.



Types of memory access for single address space:-

Two types of memory access

1. Uniform memory access (UMA)

Time taken to access a word from memory is uniform for all processors.
Latency to a word in memory does not depend on which processor asks for it.

2. Non Uniform memory access (NUMA)

Time taken to access a word from memory is different for different processors. Here main memory is divided and attached to different microprocessors.

Need of Synchronization:-

The process of co-coordinating the behavior of two or more processes, which may run on different processors is called synchronization.

Locks:-

A synchronization mechanism that allows access to data to only one processor at a time is called locks. Other processor must wait until the original process unlocks the variable.

Distributed shared memory intercore message passing method:-

An alternative approach to sharing an address space is for all the processor to have their own private physical address space. In this approach each processor will have their own private physical address space. Communication between multiple processors is done by explicitly sending and receiving information.

Routines for communication:-

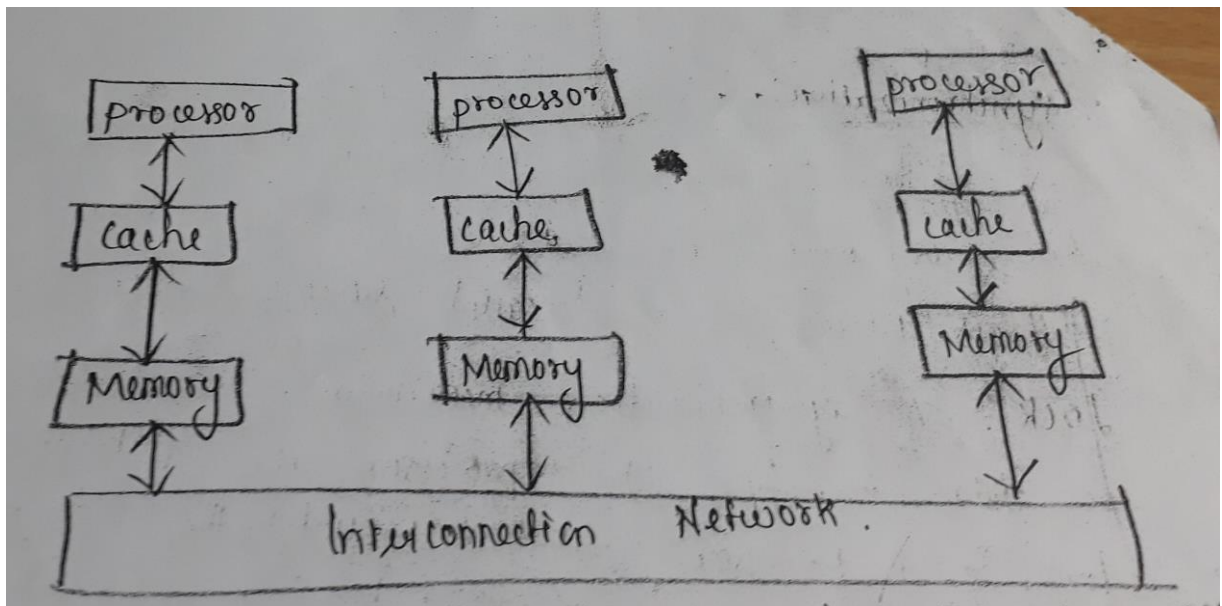
- **Send message routine:-**

It is used to send message to other processor in a machine.

- **Receive message routine:-**

It is used to receive message from other processor in a machine.

If sender processor needs confirmation the message has reached the receiver then the receiver processor can also send acknowledgement message back to the sender.



Advantage:-

Lower Latency which leads to better communication, more efficient and enhanced performance.

Disadvantages:-

- Cost for communication is very high.
- It n copies of operating system when memory is distributed.

Fundamental Theorem of Multi-Core Processors

Multi-core processors take advantage of a fundamental relationship between power and frequency.

By incorporating multiple cores, each core is able to run at a lower frequency, dividing among them the power normally given to a single core.

The result is a big performance increase over a single core processor.

This fundamental relationship between power and frequency can be effectively used to multiply the number of cores from two to four, and then eight and more, to deliver continuous increases in performance without increasing power usage.

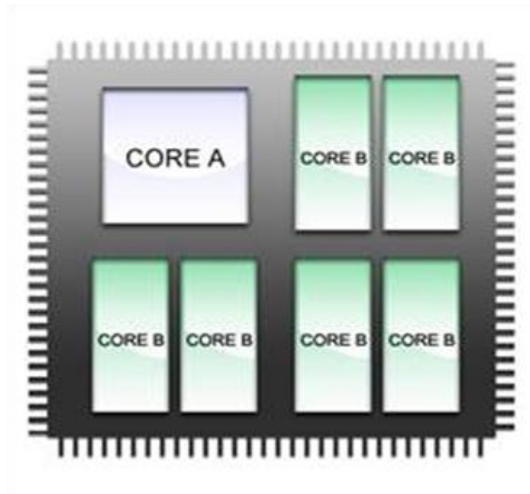
To do this though, there is much advancement that must be made that are only achievable by a company like Intel.

These include:

- **Continuous advances in silicon process technology** from 65 nm to 45 nm and to 32 nm) to increase transistor density. In addition, Intel is committed to continuing to deliver superior energy-efficient performance transistors.
- **Enhancing the performance of each core and optimizing it for multi-core** through the introduction of new advanced microarchitectures about every two years.
- **Improving the memory subsystem and optimizing data access** in ways that ensure data can be used as fast as possible among all cores. This minimizes latency and improves efficiency and speed.
- **Optimizing the interconnect fabric** that connects the cores to improve performance between cores and memory units.
- **Optimizing and expanding the instruction set** to enhance the capabilities of Intel® architecture and enable the industry to deliver advanced applications with greater performance and lower power requirements. Some of these instructions can effectively dedicate a core to deliver specific capabilities.
- **Continuing to grow Intel's commitment to developing multi-core software tools and programs** by working closely with developers, independent software vendors (ISVs), operating system vendors (OSVs) and academia. Through these efforts, Intel enables the industry to develop software that runs faster and better on our energy-efficient performance multi-core platforms.

Heterogeneous Multi-Core Processors

Heterogeneous Multi-Core Processor is a processor in which Multiple cores of different types are implemented in one CPU.



Heterogeneous Multi-Core Processor

Advantage

- Massive parallelism today
- Specialization of hardware for different tasks.

Disadvantage

- Developer productivity - use of the software tools requires special training.
- Portability - software written for GPUs will not run on other GPUs or on CPUs.
- Manageability - multiple GPUs and CPUs in a grid need their work allocated and balanced, and event-based systems need to be supported.

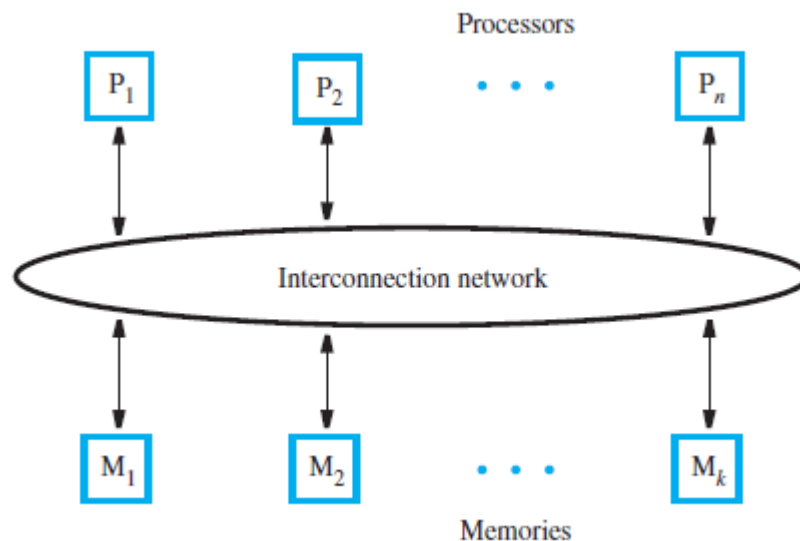
4.6.2 SHARED MEMORY MULTIPROCESSOR:-

- A multiprocessor system consists of a number of processors capable of simultaneously executing independent tasks.
- A task may encompass a few instructions for one pass through a loop, or thousands of instructions executed in a subroutine.
- In a shared-memory multiprocessor, all processors have access to the same memory.

- Tasks running in different processors can access shared variables in the memory using the same addresses. The size of the shared memory is likely to be large.
- Implementing a large memory in a single module would create a bottleneck when many processors make requests to access the memory simultaneously.
- An interconnection network enables any processor to access any module that is a part of the shared memory.
- When memory modules are kept physically separate from the processors, all requests to access memory must pass through the network, which introduces latency.

Uniform Memory Access (UMA) multiprocessor:-

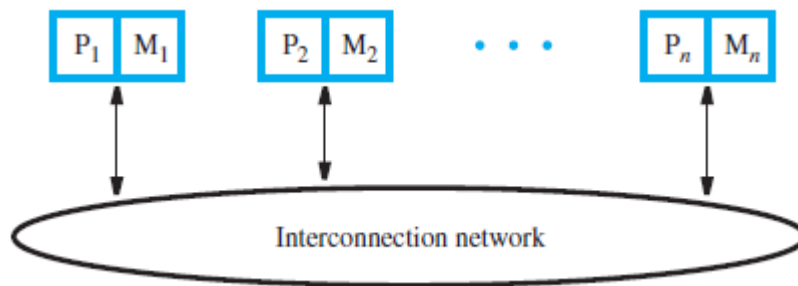
- A system which has the same network latency for all accesses from the processors to the memory modules is called a Uniform Memory Access (UMA) multiprocessor.
- Although the latency is uniform, it may be large for a network that connects many processors and memory modules.
- For better performance, it is desirable to place a memory module close to each processor. The result is a collection of *nodes*, each consisting of a processor and a memory module.



Uniform Memory Access (UMA) multiprocessor

Non-Uniform Memory Access (NUMA) multiprocessors:-

- The network latency is avoided when a processor makes a request to access its local memory. However, a request to access a remote memory module must pass through the network.
- Because of the difference in latencies for accessing local and remote portions of the shared memory, systems of this type are called Non-Uniform Memory Access (NUMA) multiprocessors.



NON Uniform Memory Access (NUMA) multiprocessor

4.6.1 Interconnection Networks:-

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

- a read request is likely to be a single packet sent from a processor to a memory module. The packet contains the node identifiers for the source and destination, the address of the location to be read, and a command field that indicates what type of read operation is required.
- A write request that writes one word in a memory module is also likely to be a single packet that includes the data to be written. On the other hand, a read response may involve an entire cache block requiring several packets for the data transfer.

The following are the interconnection networks that are commonly used in multiprocessors.

Bus

- A bus is a set of lines (wires) that provide a single shared path for information transfer.
- Buses are most commonly used in UMA multiprocessors to connect a number of processors to several shared-memory modules.
- Arbitration is necessary to ensure that only one of many possible requesters is granted use of the bus at any time.
- The bus is suitable for a relatively small number of processors because of the contention for access to the bus and the increased propagation delays caused by electrical loading when many processors are connected.
- Higher performance can be achieved by using a split-transaction bus, in which a request and its corresponding response are treated as separate events.

Consider a situation where multiple processors need to make read requests to the memory. Arbitration is used to select the first processor to be granted use of the bus for its request. After the request is made, a second processor is selected to make its request, instead of leaving the bus idle. Assuming that this request is to a different memory module, the two read accesses proceed in parallel. If neither module has finished with its access, a third processor is selected to make its request, and so on.

Ring

A *ring* network is formed with point-to-point connections between nodes.

Two approaches are used

1. Single Ring
2. Hierarchy of rings

Single ring:-

A long single ring results in high average latency for communication between any two nodes.

This high latency can be mitigated in two different ways.

A second ring can be added to connect the nodes in the opposite direction.

The resulting *bidirectional ring* halves the average latency and doubles the bandwidth. However, handling of communications is more complex.

hierarchy of rings:-

- It is a two-level hierarchy.
- The upper-level ring connects the lower-level rings.
- The average latency for communication between any two nodes on lower level rings is reduced with this arrangement.
- Transfers between nodes on the same lower-level ring need not traverse the upper-level ring. Transfers between nodes on different lower-level rings include a traversal on part of the upper-level ring.

Drawback of the Hierarchical Scheme:-

The upper-level ring may become a bottleneck when many nodes on different lower-level rings communicate with each other frequently.

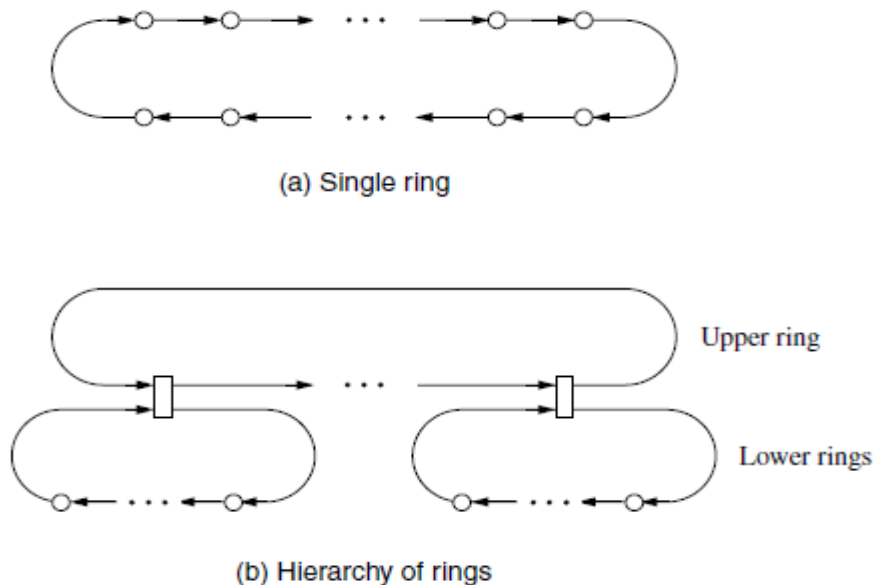


Figure Ring based inter connection network

Crossbar:-

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

- It is typically used in UMA multiprocessors to connect processors to memory modules. It enables many simultaneous transfers if the same destination is not the target of multiple requests.
- For example, we can implement the structure using a crossbar that comprises a collection of switches for n processors and k memories, $n \times k$ switches are needed.

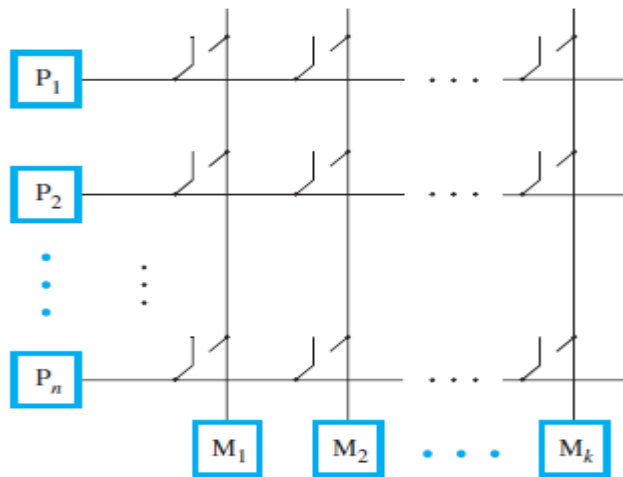


Figure Cross bar interconnection network

Mesh:-

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

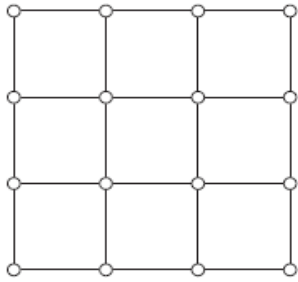


Figure Two Dimensional mesh network

4.7 INTRODUCTION TO GRAPHICS PROCESSING UNITS

- The increasing demands of processing for computer graphics has led to the development of specialized chips called *graphics processing units* (GPUs).
- The primary purpose of GPUs is to accelerate the large number of floating-point calculations needed in high-resolution three-dimensional graphics, such as in video games.
- Since the operations involved in these calculations are often independent, a large GPU chip contains hundreds of simple cores with floating-point ALUs to perform them in parallel.
- A GPU chip and a dedicated memory for it are included on a video card. Such a card is plugged into an expansion slot of a host computer using an interconnection standard such as the PCIe standard.
- A small program is written for the processing cores in the GPU chip and a large number of cores execute this program in parallel. The cores execute the same instructions, but operate on different data elements.
- A separate controlling program runs in the general-purpose processor of the host computer and invokes the GPU program when necessary.

-
- Before initiating the GPU computation, the program in the host computer must first transfer the data needed by the GPU program from the main memory into the dedicated GPU memory.
 - After the computation is completed, the resulting output data in the dedicated memory are transferred back to the main memory.
 - The processing cores in a GPU chip have a specialized instruction set and hardware architecture, which are different from those used in a general-purpose processor.
 - An example is the *Compute Unified Device Architecture* (CUDA) that NVIDIA Corporation uses for the cores in its GPU chips. To facilitate writing programs that involve a general-purpose processor and a GPU, an extension to the C programming language, called CUDA C, has been developed by NVIDIA . This extension enables a single program to be written in C, with special keywords used to label the functions executed by the processing cores in a GPU chip.
 - The compiler and related software tools automatically partition the final object program into the portions that are translated into machine instructions for the host computer and the GPU chip. Library routines are provided to allocate storage in the dedicated memory of a GPU-based video card and to transfer data between the main memory and the dedicated memory.
 - An open standard called Open CL has also been proposed by industry as a programming framework for systems that include GPU chips from any vendor .

4.7.1 Key characteristics as to how GPUs vary from CPUs

- GPUs are accelerators that supplement a CPU, so they do not need be able to perform all the tasks of a CPU. This role allows them to dedicate all their resources to graphics.
- It's fine for GPUs to perform some tasks poorly or not at all, given that in a system with both a CPU and a GPU, the CPU can do them if needed.

- The GPU problems sizes are typically hundreds of megabytes to gigabytes, but not hundreds of gigabytes to terabytes.

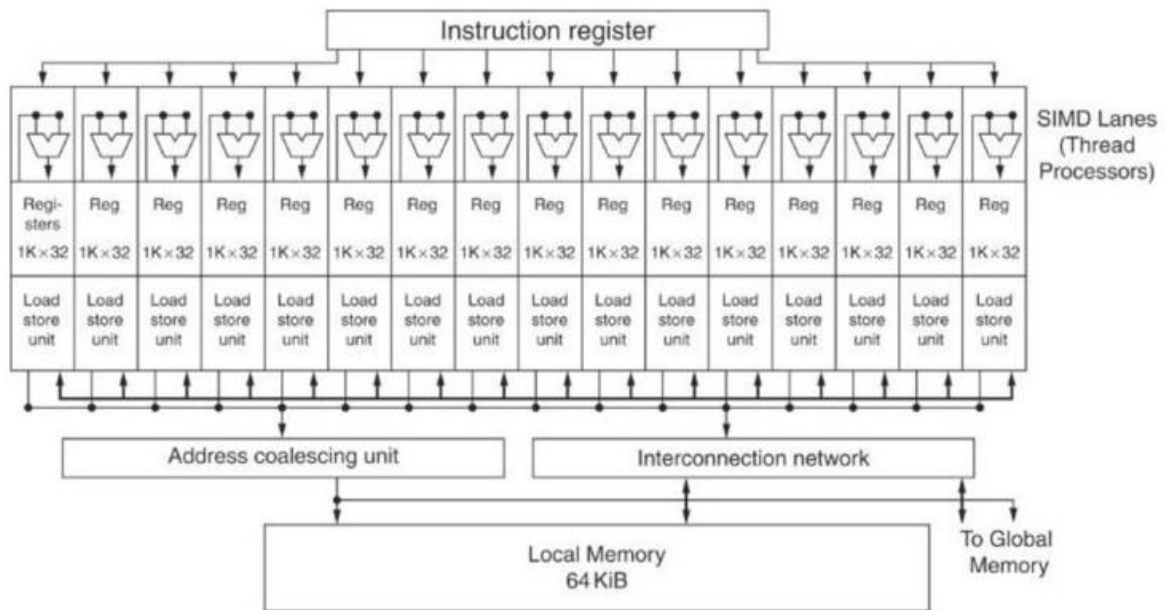
These differences led to different styles of architecture:

- Perhaps the biggest difference is that GPUs do not rely on multilevel caches to overcome the long latency to memory, as do CPUs.
- Instead, GPUs rely on hardware multithreading (Section 6.4) to hide the latency to memory.
- That is, between the time of a memory request and the time that data arrives, the GPU executes hundreds or thousands of threads that are independent of that request.
- The GPU memory is thus oriented toward bandwidth rather than latency.
- There are even special graphics DRAM chips for GPUs that are wider and have higher bandwidth than DRAM chips for CPUs. In addition, GPU memories have traditionally had smaller main memories than conventional microprocessors.
- In 2013, GPUs typically have 4 to 6 GiB or less, while CPUs have 32 to 256 GiB. Finally, keep in mind that for general-purpose computation, you must include the time to transfer the data between CPU memory and GPU memory, since the GPU is a coprocessor.
- Given the reliance on many threads to deliver good memory bandwidth, GPUs can accommodate many parallel processors (MIMD) as well as many threads.
- Hence, each GPU processor is more highly multithreaded than a typical CPU, plus they have more processors.

An Introduction to the NVIDIA GPU Architecture

- NVIDIA decided that the unifying theme of all these forms of parallelism is the CUDA Thread.

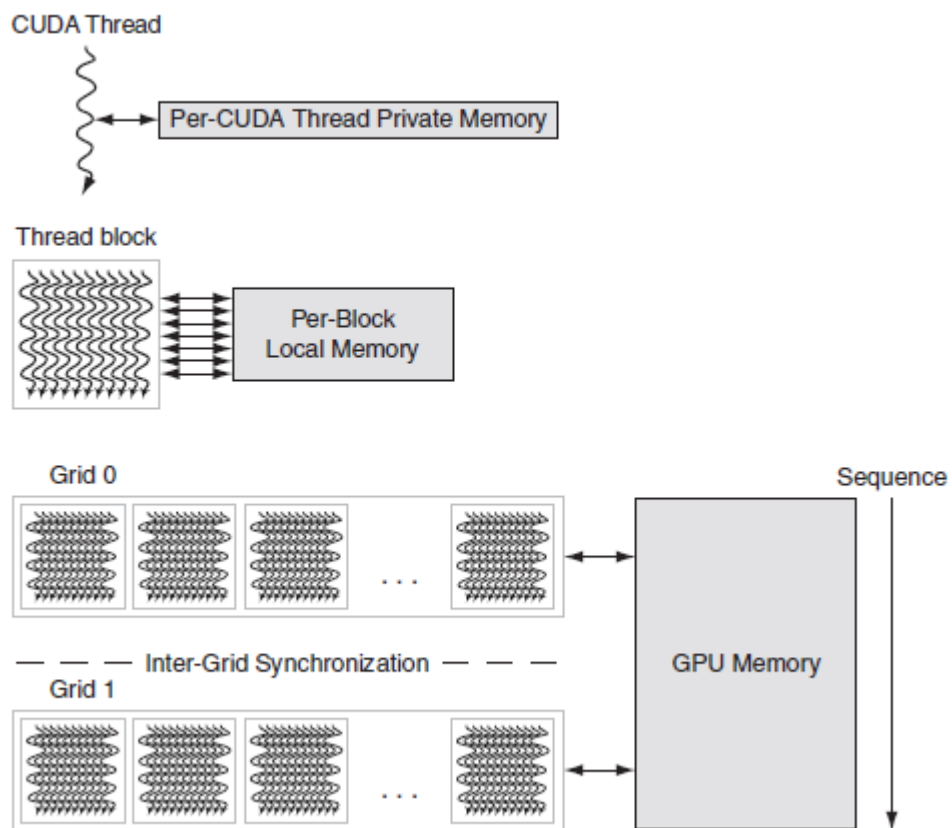
- Using this lowest level of parallelism as the Programming primitive, the compiler and the hardware can gang thousands of CUDA threads together to utilize the various styles of parallelism within a GPU: multithreading, MIMD, SIMD, and instruction-level parallelism.
- These threads are blocked together and executed in groups of 32 at a time.
- A multithreaded processor inside a GPU executes these blocks of threads, and a GPU consists of 8 to 32 of these multithreaded processors.
- A multithreaded SIMD processor is similar to a Vector Processor, but the former has many parallel functional units instead of just a few that are deeply pipelined, as does the latter.
- GPU contains a collection of multithreaded SIMD processors; that is, a GPU is a MIMD composed of multithreaded SIMD processors. For example, NVIDIA has four implementations of the Fermi architecture at different price points with 7, 11, 14, or 15 multithreaded SIMD processors.
- To provide transparent scalability across models of GPUs with differing number of multithreaded SIMD processors, the Thread Block Scheduler hardware assigns blocks of threads to multithreaded SIMD processors.



- Simplified block diagram of the data path of a multithreaded SIMD Processor. It has 16 SIMD lanes. The SIMD Thread Scheduler has many independent SIMD threads that it chooses from to run on this processor.
- Dropping down one more level of detail, the machine object that the hardware creates, manages, schedules, and executes is a *thread of SIMD instructions*, which also call a *SIMD thread*. It is a traditional thread, but it contains exclusively SIMD instructions. These SIMD threads have their own program counters and they run on a multithreaded SIMD processor.
- The *SIMD Thread Scheduler* includes a controller that lets it know which threads of SIMD instructions are ready to run, and then it sends them off to a dispatch unit to be run on the multithreaded SIMD processor. It is identical to a hardware thread scheduler in a traditional multithreaded processor, except that it is scheduling threads of SIMD instructions.

- Thus, GPU hardware has two levels of hardware schedulers:
 1. The *Thread Block Scheduler* that assigns blocks of threads to multithreaded SIMD processors, and
 2. the SIMD Thread Scheduler *within* a SIMD processor, which schedules when SIMD threads should run.
- The SIMD instructions of these threads are 32 wide, so each thread of SIMD instructions would compute 32 of the elements of the computation. Since the thread consists of SIMD instructions, the SIMD processor must have parallel functional units to perform the operation.

NVIDIA GPU Memory Structures



GPU Memory structures

1.213 Computer Architecture

- GPU Memory is shared by the vectorized loops. All threads of SIMD instructions within a thread block share Local Memory.
- The above diagram shows the memory structures of an NVIDIA GPU.
- On-chip memory that is local to each multithreaded SIMD processor *Local Memory*.
- It is shared by the SIMD Lanes within a multithreaded SIMD processor, but this memory is not shared between multithreaded SIMD processors.
- Off-chip DRAM shared by the whole GPU and all thread blocks *GPU Memory*.
- Rather than rely on large caches to contain the whole working sets of an application, GPUs traditionally use smaller streaming caches and rely on extensive multithreading of threads of SIMD instructions to hide the long latency to DRAM, since their working sets can be hundreds of megabytes. Thus, they will not fit in the last level cache of a multicore microprocessor.
- Given the use of hardware multithreading to hide DRAM latency, the chip area used for caches in system processors is spent instead on computing resources and on the large number of registers to hold the state of the many threads of SIMD instructions.

Similarities and differences between multicore with multimedia SIMD extensions and recent GPUs.

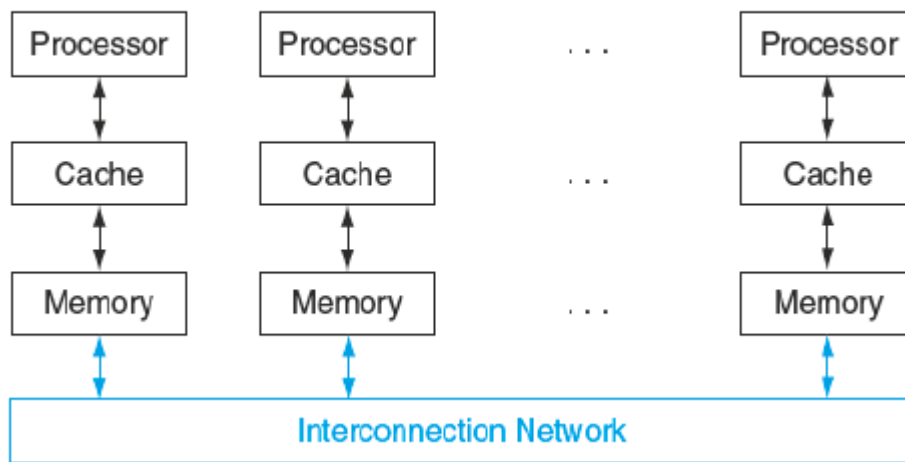
- At a high level, multicore computers with SIMD instruction extensions do share similarities with GPUs.

Feature	Multicore with SIMD	GPU
SIMD processors	4 to 8	8 to 16
SIMD lanes/processor	2 to 4	8 to 16
Multithreading hardware support for SIMD threads	2 to 4	16 to 32
Largest cache size	8 MIB	0.75 MIB
Size of memory address	64-bit	64-bit
Size of main memory	8 GiB to 256 GiB	4 GiB to 6 GiB
Memory protection at level of page	Yes	Yes
Demand paging	Yes	No
Cache coherent	Yes	No

- Both are MIMDs whose processors use multiple SIMD lanes, although GPUs have more processors and many more lanes.
- Both use hardware multithreading to improve processor utilization, although GPUs have hardware support for many more threads.
- Both use caches, although GPUs use smaller streaming caches and multicore computers use large multilevel caches that try to contain whole working sets completely.
- Both use a 64-bit address space, although the physical main memory is much smaller in GPUs. While GPUs support memory protection at the page level, they do not yet support demand paging.
- SIMD processors are also similar to vector processors. The multiple SIMD processors in GPUs act as independent MIMD cores, just as many vector computers have multiple vector processors.
- This view would consider the Fermi GTX 580 as a 16-core machine with hardware support for multithreading, where each core has 16 lanes.
- The biggest difference is multithreading, which is fundamental to GPUs and missing from most vector processors.

4.7.2 Clusters, Warehouse Scale Computers and other Message-Passing Multiprocessors.

The alternative approach to sharing an address space is for the processors to each have their own private physical address space.

**Message passing multiprocessor**

The Classic organization of a multiprocessor with multiple private address spaces, traditionally called a message-passing multiprocessor.

- This alternative multiprocessor must communicate via explicit message passing, which traditionally is the name of such style of computers.
- Provided the system has routines to send and receive messages, coordination is built in with message passing, since one processor knows when a message is sent, and the receiving processor knows when a message arrives.
- If the sender needs confirmation that the message has arrived, the receiving processor can then send an acknowledgment message back to the sender.
- There have been several attempts to build large-scale computers based on high-performance message-passing networks, and they do offer better

absolute communication performance than clusters built using local area networks.

- Indeed, many supercomputers today use custom networks. The problem is that they are much more expensive than local area networks like Ethernet.
- Few applications today outside of high performance computing can justify the higher communication performance, given the much higher costs.
- Computers that rely on message passing for communication rather than cache coherent shared memory are much easier for hardware designers to build. There is an advantage for programmers as well, in that communication is explicit, which means there are fewer performance surprises than with the implicit communication in cache-coherent shared memory computers.
- The downside for programmers is that it's harder to port a sequential program to a message-passing computer, since every communication must be identified in advance or the program doesn't work.
- Cache-coherent shared memory allows the hardware to figure out what data needs to be communicated, which makes porting easier.
- There are differences of opinion as to which is the shortest path to high performance, given the pros and cons of implicit communication, but there is no confusion in the marketplace today.
- Multicore microprocessors use shared physical memory and nodes of a cluster communicate with each other using message passing.
- Some concurrent applications run well on parallel hardware, independent of whether it offers shared addresses or message passing. In particular, task-level parallelism and applications with little communication-like Web search, mail servers, and file servers--do not require shared addressing to run well.

- As a result, clusters have become the most widespread example today of the message-passing parallel computer.
- Given the separate memories, each node of a cluster runs a distinct copy of the operating system.
- In contrast, the cores inside a microprocessor are connected using a high-speed network inside the chip, and a multichip shared memory system uses the memory interconnect for communication.
- The memory interconnect has higher bandwidth and lower latency, allowing much better communication performance for shared memory multiprocessors.
- The weakness of separate memories for user memory from a parallel programming perspective turns into a strength in system dependability. Since a cluster consists of independent computers connected through a local area network, it is much easier to replace a computer without bringing down the system in a cluster than in a shared memory multiprocessor.
- Fundamentally, the shared address means that it is difficult to isolate a processor and replace it without heroic work by the operating system and in the physical design of the server.
- It is also easy for clusters to scale down gracefully when a server fails, thereby improving dependability. Since the cluster software is a layer that runs on top of the local operating systems running on each computer, it is much easier to disconnect and replace a broken computer
- Given that clusters are constructed from whole computers and independent, scalable networks, this isolation also makes it easier to expand the system without bringing down the application that runs on top of the cluster.
- Their lower cost, higher availability, and rapid, incremental expandability make clusters attractive to service Internet providers, despite their poorer

communication performance when compared to large-scale shared memory multiprocessors.

- The search engines that hundreds of millions of us use every day depend upon this technology. Amazon, Facebook, Google, Microsoft, and others all have multiple datacenters each with clusters of tens of thousands of servers. Clearly, the use of multiple processors in Internet service companies has been hugely successful.

Warehouse-Scale Computers:-

- Internet services required the construction of new buildings to house, power, and cool 100,000 servers. Although they may be classified as just large clusters, their architecture and operation are more sophisticated.
- They act as one giant computer and cost on the order of \$150M for the building, the electrical and cooling infrastructure, the servers, and the networking equipment that connects and houses 50,000 to 100,000 servers. consider them a new class of computer, called **Warehouse-Scale Computers (WSC)**.
- The most popular framework for batch processing in a WSC is MapReduce and its open-source twin Hadoop.
- Inspired by the Lisp functions of the same name, Map first applies a programmer-supplied function to each logical input record. Map runs on thousands of servers to produce an intermediate result of key- value pairs.
- Reduce collects the output of those distributed tasks and collapses them using another programmer-defined function. With appropriate software support, both are highly parallel yet easy to understand and to use. Within 30 minutes, a novice programmer can run a MapReduce task on thousands of servers.

- For example, one MapReduce program calculates the number of occurrences of every English word in a large collection of documents. Below is a simplified version of that program, which shows just the inner loop and assumes just one occurrence of all English words found in a document:

```
map(String key, String value):  
    // key: document name  
  
    // value: document contents for each word w in value:  
  
    EmitIntermediate(w, "1"); // Produce list of all words  
reduce(String key, Iterator values):  
    // key: a word  
  
    // values: a list of counts int result = 0;  
  
    for each v in values:  
  
        result += ParseInt(v); // get integer from key-value pair  
Emit(AsString(result));
```

- The function `EmitIntermediate` used in the `Map` function emits each word in the document and the value one.
- Then the `Reduce` function sums all the values per word for each document using `ParseInt()` to get the number of occurrences per word in all documents.
- The MapReduce runtime environment schedules map tasks and reduce tasks to the servers of a WSC.
- It requires innovation in power distribution, cooling, monitoring, and operations, the WSC is a modern descendant of the 1970s supercomputers—making Seymour Cray the godfather of today’s WSC architects.
- His extreme computers handled computations that could be done nowhere else, but were so expensive that only a few companies could afford them.
- This time the target is providing information technology for the world instead of high performance computing for scientists and engineers.

- Hence, WSCs surely play a more important societal role today than Cray's supercomputers did in the past.

WSCs have three major distinctions:

1. Ample, easy parallelism:

- A concern for a server architect is whether the applications in the targeted marketplace have enough parallelism to justify the amount of parallel hardware and whether the cost is too high for sufficient communication hardware to exploit this parallelism. A WSC architect has no such concern.
- First, batch applications like MapReduce benefit from the large number of independent data sets that need independent processing, such as billions of Web pages from a Web crawl.
- Second, interactive Internet service applications, also known as Software as a Service (SaaS), can benefit from millions of independent users of interactive Internet services.
- Reads and writes are rarely dependent in SaaS, so SaaS rarely needs to synchronize. For example, search uses a read-only index and email is normally reading and writing independent information.
- We call this type of easy parallelism Request-Level Parallelism, as many independent efforts can proceed in parallel naturally with little need for communication or synchronization.

2. Operational Costs Count:

- Traditionally, server architects design their systems for peak performance within a cost budget and worry about energy only to make sure they don't exceed the cooling capacity of their enclosure.

1.221 Computer Architecture

- They usually ignored operational costs of a server, assuming that they pale in comparison to purchase costs.
- WSC have longer lifetimes—the building and electrical and cooling infrastructure are often amortized over 10 or more years—so the operational costs add up: energy, power distribution, and cooling represent more than 30% of the costs of a WSC over 10 years.

3. Scale and the Opportunities/Problems Associated with Scale:

- To construct a single WSC, you must purchase 100,000 servers along with the supporting infrastructure, which means volume discounts.
- Hence, WSCs are so massive internally that you get economy of scale even if there are not many WSCs.
- These economies of scale led to cloud computing, as the lower per unit costs of a WSC meant that cloud companies could rent servers at a profitable rate and still be below what it costs outsiders to do it themselves.
- The flip side of the economic opportunity of scale is the need to cope with the failure frequency of scale.
- Even if a server had a Mean Time To Failure of an amazing 25 years (200,000 hours), the WSC architect would need to design for 5 server failures every day. The annualized disk failure rate (AFR) was measured at Google at 2% to 4%. If there were 4 disks per server and their annual failure rate was 2%, the WSC architect should expect to see one disk fail every hour. Thus, fault tolerance is even more important for the WSC architect than the server architect.

