

CSE 4305
Computer Organization and Architecture

Lecture 2

A top-level view of CO

Sabrina Islam
Lecturer, CSE, IUT
Contact: +8801832239897
E-mail: sabrinaislam22@iut-dhaka.edu

- ❖ **How does a computer work?**
- ❖ **What are the main components that make the computer?**
- ❖ **How are these components organized?**
- ❖ **How do they communicate with each other?**

Most computers are designed based on the concepts developed by **John von Neumann**.

The 3 key concepts are:

- Ability to write/read data from memory
- Ability to execute instruction sequentially
- Ability to process inputs/outputs

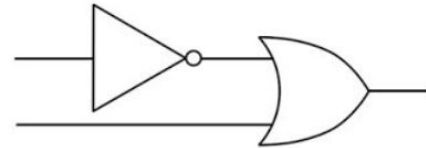
**But why are these concepts great and were adopted
while designing a computer?**

Consider a scenario:

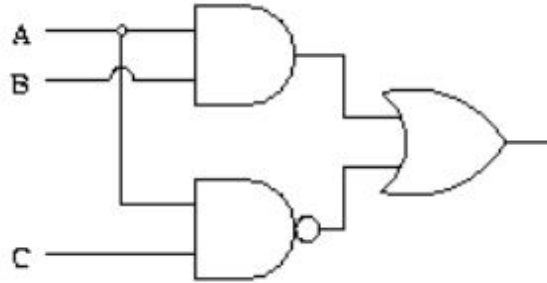
You want to solve a problem: $c = \bar{a} + b$

To solve this using digital circuits:

- We need to implement logic gates in a certain way.
- Employ some sort of memories.
- Execute the binary functions.



Now if you want to solve this problem: $c = \overline{a}c + ab$



This process of connecting the components in the desired configuration is called **hardwired programming**.

But do you see any problem?

Disadvantages of hardwired programming:

- Rigid design: Works only for a specific function.
- Requires generating a new circuit to accommodate additional functions
 - producing new truth tables.
 - algebraic simplification.
 - circuit design.

Now consider a new scenario where we have a different method

- A module capable of calculating arithmetic/logic functions:
 - Logic functions: AND, OR, NOT, NAND, NOR, XOR, XNOR
 - Arithmetic functions: addition, subtraction, multiplication, division, etc.
- A way to control such a module
 - Control signals indicating which operation to do
 - Data forwarding mechanisms: input/output

This is a general-purpose computer.

What are the advantages of such system?

- The system accepts data and control signals and produces results.
- No need for rewiring the hardware for each new program.
- The programmer merely needs to supply a new set of control signals that defines which functions need to be performed.

Now let's try to understand this system

- How shall the control signals be supplied?
- How do we put data and instructions into the system and how is the result reported back to us?
- How are the instructions and data stored while performing the operation if needed?

How shall the control signals be supplied?

- Program is a sequence of steps. At each step, an arithmetic/logical operation is performed on data
- Each instruction thus requires its own set of control signals.
- Hardware interprets each instruction and generates control signals.

This new method of programming is called **Software**.

So we need:

- An instruction interpreter capable of generating control signals.
- A general-purpose module for arithmetic/logic functions.

→ CPU

How do we put data and instructions into the system and how is the result reported back to us?

- We need a module that contains basic components for accepting data and instructions in some form and converting them into an internal form of signals usable by the system
- Another means to send back the produced results to us.

I/O module

How are the instructions and data stored while performing the operation if needed?

- There must be a place to store both instructions and data as operations on data may require access to more than just one element at a time in a predetermined sequence.
- Also we need to store information when a program jumps around instead of executing instructions sequentially.

Main memory

What components did we get so far?

- How shall the control signals be supplied? **CPU**
- How do we put data and instructions into the system and how is the result reported back to us? **I/O Modules**
- How are the instructions and data stored while performing the operation if needed? **Main memory**

Thus we found the main components of the **von Neumann** architecture:

- **CPU**
- **Main memory**
- **I/O modules**

These components together executes programs.

So as part of computer organization we need to look into details of these components and how they are interconnected to execute the programs.

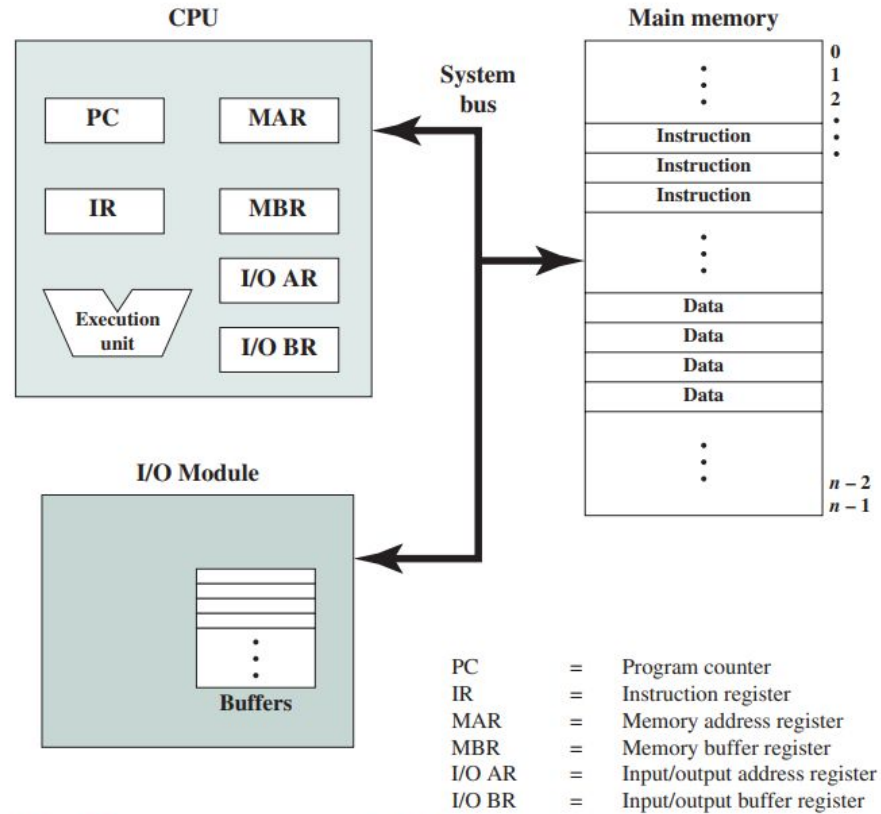
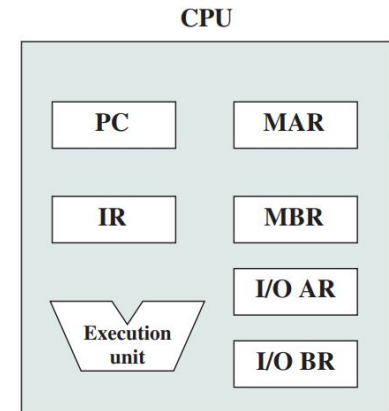


Fig: Top-level view of a computer

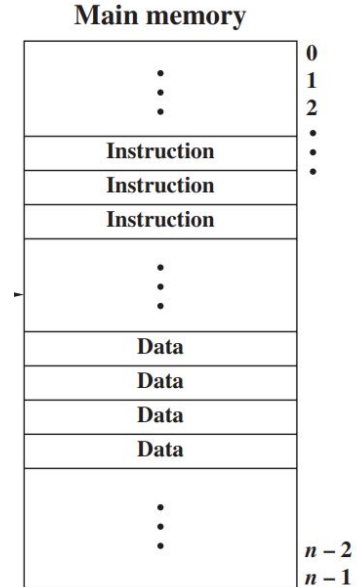
CPU has a set of internal registers:

- **Program Counter (PC)**
 - specifies the memory address of the next instruction to be executed.
- **Instruction Register (IR)**
 - holds the instruction currently being executed or decoded.
- **Memory address register (MAR)**
 - specifies memory address to be read/written.
- **Memory buffer register (MBR)**
 - contains the data to be written into memory or
 - receives the data read from memory
- **I/O address register (I/OAR)**
 - specifies a particular I/O device
- **I/O buffer (I/OBR) register**
 - used for the exchange of data between an I/O module and the CPU



Memory module

- Set of sequentially numbered addresses
- Each location contains binary information (word)
 - Instructions
 - Data



I/O module

- Transfers data from external devices to CPU and memory and vice versa.
- Containing internal buffers for temporarily holding data until they can be sent on.

**Now let's look at how a computer
functions!**

Computer basically executes instructions!

Program execution consists of:

- Instruction fetch
 - reads instructions from memory one at a time.
- Instruction execution.
 - executes each instruction fetched.

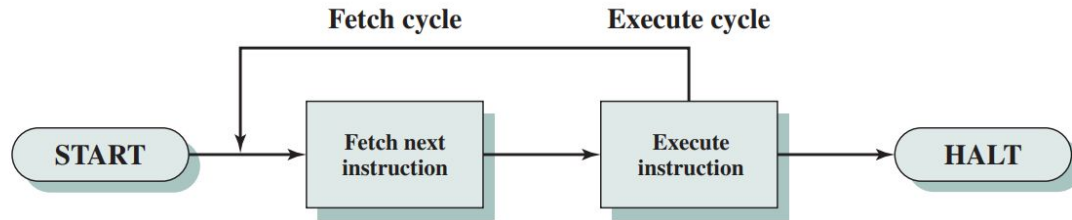


Fig: Basic Instruction Cycle

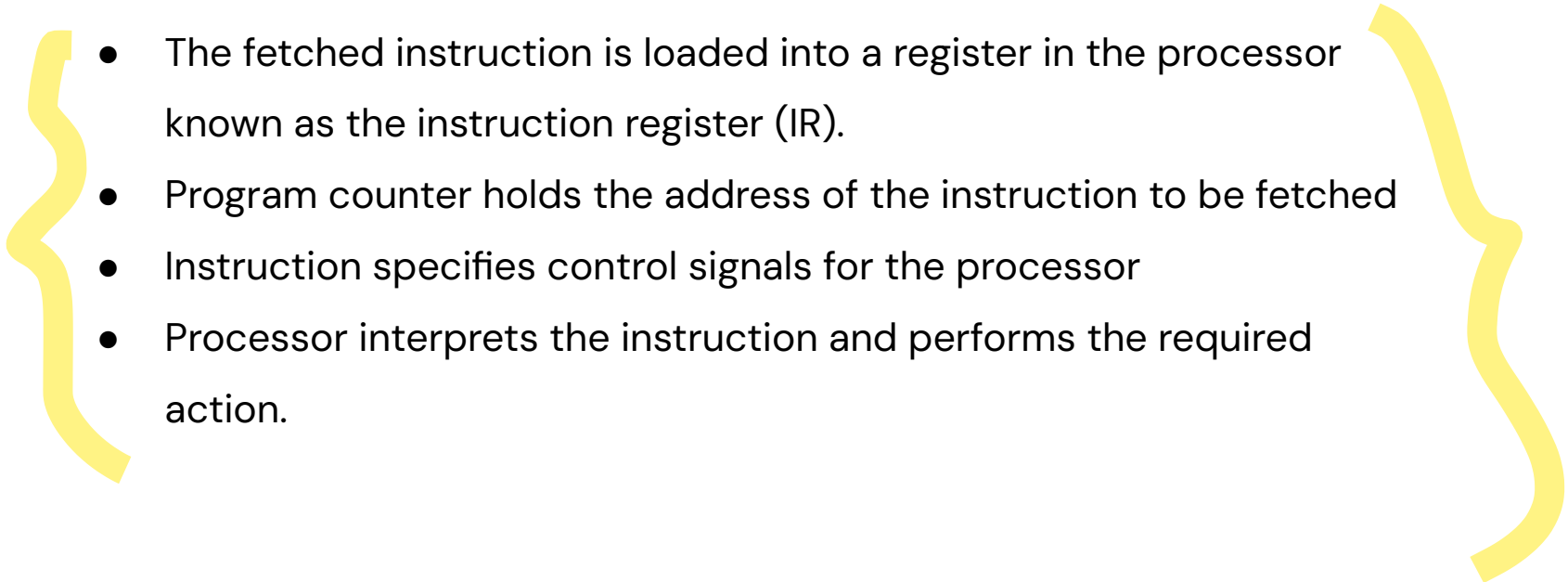
Instruction format

Suppose the instructions are 16 bit long.



Fig: Instruction format

- **Opcode**
 - the code of the operation to be executed e.g. add, multiply, etc.
 - here opcode has 4 bits - $2^4 = 16$ different combinations
- **Memory address**
 - 12 bits for memory - $2^{12} = 4096$ different memories can be addressed.

- 
- The fetched instruction is loaded into a register in the processor known as the instruction register (IR).
 - Program counter holds the address of the instruction to be fetched
 - Instruction specifies control signals for the processor
 - Processor interprets the instruction and performs the required action.

Processor interprets the instruction and performs the required action.

In general, these actions fall into **four** categories:

- **Processor-memory**
 - Data may be transferred from processor to memory or from memory to processor.
- **Processor-I/O**
 - Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module.
- **Data processing**
 - The processor may perform some arithmetic or logic operation on data.
- **Control**
 - An instruction may specify that the sequence of execution be altered.

Let's try to understand the execution of instructions with an example

Assume that the program counter is set to memory location 300, where the location address refers to a 16-bit word. The processor will next fetch the instruction at location 300. After the execution of this instruction, it will fetch instructions from locations 301, 302, 303, and so on.

The program adds the contents of the memory word at address 940 to the contents of the memory word at address 941 and stores the result in the latter location. What will be the steps?

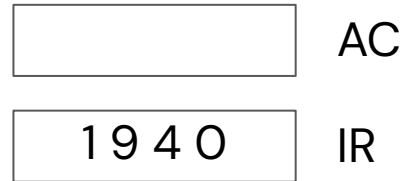
Solution

PC

| | |
|-----|---------|
| 300 | 1 9 4 0 |
| 301 | 5 9 4 1 |
| 302 | 2 9 4 1 |
| | . |
| | . |
| 940 | 0 0 0 3 |
| 941 | 0 0 0 2 |

Program counter (PC) = Address of instruction
Instruction register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

0001 = Load AC from memory
0010 = Store AC to memory
0101 = Add to AC from memory



Solution

PC

| | |
|-----|---------|
| 300 | 1 9 4 0 |
| 301 | 5 9 4 1 |
| 302 | 2 9 4 1 |
| | . |
| | . |
| 940 | 0 0 0 3 |
| 941 | 0 0 0 2 |

Program counter (PC) = Address of instruction
Instruction register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

0001 = Load AC from memory
0010 = Store AC to memory
0101 = Add to AC from memory

0 0 0 3

AC

1 9 4 0

IR

Solution

| | | |
|----|-----|---------|
| PC | 300 | 1 9 4 0 |
| | 301 | 5 9 4 1 |
| | 302 | 2 9 4 1 |
| | | . |
| | | . |
| | 940 | 0 0 0 3 |
| | 941 | 0 0 0 2 |

Program counter (PC) = Address of instruction
Instruction register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

0001 = Load AC from memory
0010 = Store AC to memory
0101 = Add to AC from memory

| | |
|---------|----|
| 0 0 0 3 | AC |
| 5 9 4 1 | IR |

Solution

| | | |
|----|-----|---------|
| PC | 300 | 1 9 4 0 |
| | 301 | 5 9 4 1 |
| | 302 | 2 9 4 1 |
| | | . |
| | | . |
| | 940 | 0 0 0 3 |
| | 941 | 0 0 0 2 |

Program counter (PC) = Address of instruction
Instruction register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

0001 = Load AC from memory
0010 = Store AC to memory
0101 = Add to AC from memory

| | |
|---------|----|
| 0 0 0 5 | AC |
| 5 9 4 1 | IR |

Solution

| | | |
|----|-----|---------|
| PC | 300 | 1 9 4 0 |
| | 301 | 5 9 4 1 |
| | 302 | 2 9 4 1 |
| | | . |
| | | . |
| | 940 | 0 0 0 3 |
| | 941 | 0 0 0 2 |

Program counter (PC) = Address of instruction
Instruction register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

0001 = Load AC from memory
0010 = Store AC to memory
0101 = Add to AC from memory

| | |
|---------|----|
| 0 0 0 5 | AC |
| 2 9 4 1 | IR |

Solution

| | | |
|----|-----|---------|
| PC | 300 | 1 9 4 0 |
| | 301 | 5 9 4 1 |
| | 302 | 2 9 4 1 |
| | | . |
| | | . |
| | 940 | 0 0 0 3 |
| | 941 | 0 0 0 5 |

Program counter (PC) = Address of instruction
Instruction register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

0001 = Load AC from memory
0010 = Store AC to memory
0101 = Add to AC from memory

| | |
|---------|----|
| 0 0 0 5 | AC |
| 2 9 4 1 | IR |

The steps:

1. The **PC** contains value **300**, the address of the first instruction.
 - a. This instruction (the value **1940** in hexadecimal) is loaded into the instruction register **IR**, and the PC is incremented
2. The first 4 bits (first hexadecimal digit) in the IR indicate that the AC is to be loaded. The remaining 12 bits (three hexadecimal digits) specify the address (940) from which data are to be loaded.
3. The next instruction (5941) is fetched from location 301, and the PC is incremented.
4. The old contents of the AC and the contents of location 941 are added, and the result is stored in the AC.
5. The next instruction (2941) is fetched from location 302, and the PC is incremented.
6. The contents of the AC are stored in location 941.

A more detailed look at the basic instruction cycle

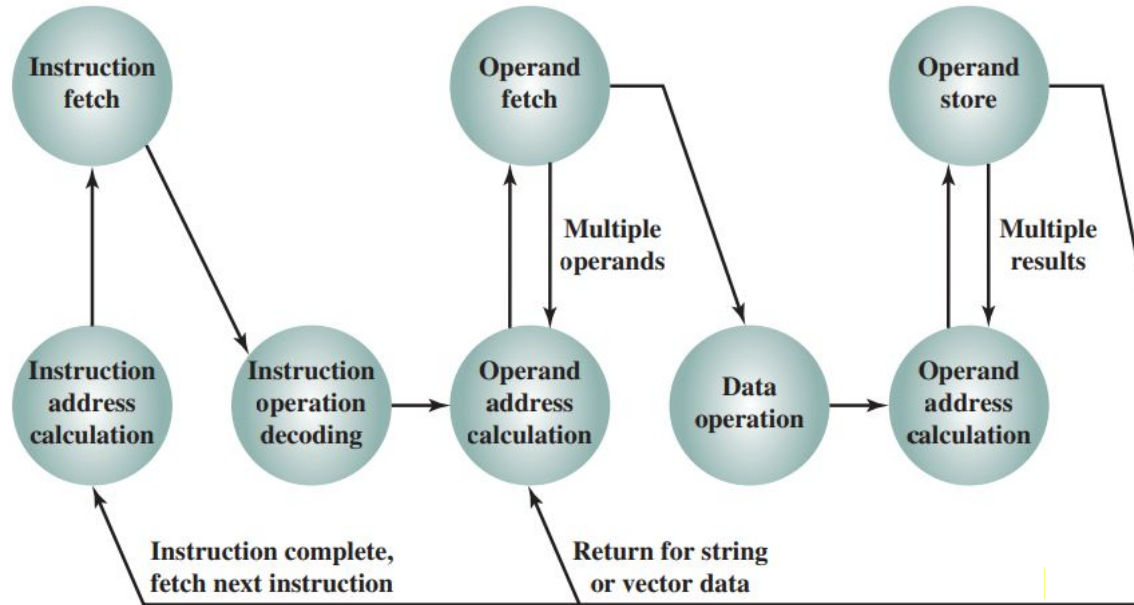


Fig: Instruction Cycle State Diagram

The states of an instruction cycle:

- Instruction address calculation (IAC):
 - Determine address of next instruction to be executed
- Instruction fetch (IF)
 - Read instruction from memory into the processor
- Instruction operation decoding (IOD)
 - Determine type of operation to be performed and operand(s) to be used.
- Operand address calculation (OAC)
 - Determine the address of the operand.
- Operand fetch (OF)
 - Fetch the operand from memory or read it in from I/O.
- Data operation (DO):
 - Perform the operation indicated in the instruction.
- Operand store (OS):
 - Write the result into memory or to I/O.

Example:

A processor includes an instruction, expressed symbolically as “ADD B,A”, that stores the sum of the contents of memory locations B and A into memory location A.

Solution:

- Fetch the ADD instruction.
- Read the contents of memory location A into the processor.
- Read the contents of memory location B into the processor.
- Add the two values.
- Write the result from the processor to memory location A.

sequence of states: iac, if, iod, oac, of, oac, of, do, oac, os.

Anything missing though?

Can you think of any other situations that aren't handled yet?

Consider the following scenarios:

1. An unexpected error occurred...
2. An input data is given from peripherals....
3. We want to share resources among multiple processes....

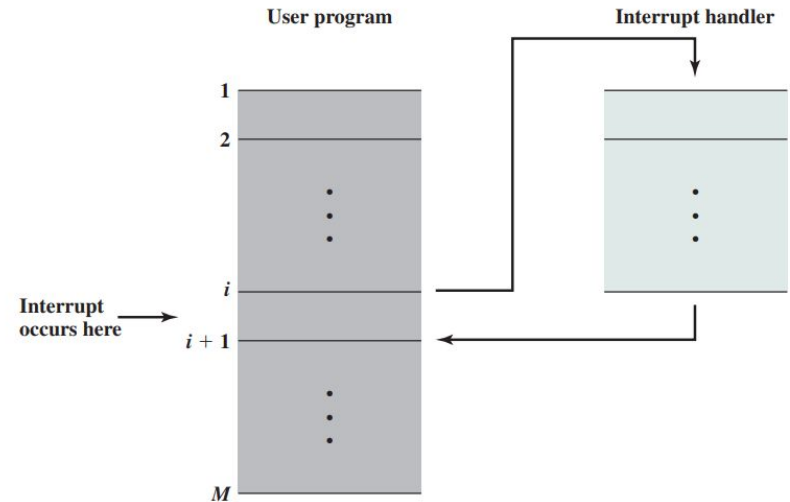
Situations that require computer to be interrupted:

- **Program**
 - arithmetic overflow, division by zero, segmentation fault, etc.
- **Timer**
 - execute something periodically.
- **I/O**
 - exchange communication between I/O devices and the processor.
- **Hardware**
 - generated by an hardware failure (power, memory parity error, etc).

That's why computer provides an interrupt mechanism.

Basic interruption mechanism:

- An interrupt request signal is sent to the processor.
- The processor responds by suspending current program execution
- The processor executes a routine capable of dealing with the interrupt (interrupt handler).
- After the handler finishes the program returns to the original execution.



Instruction cycle with interrupt cycle

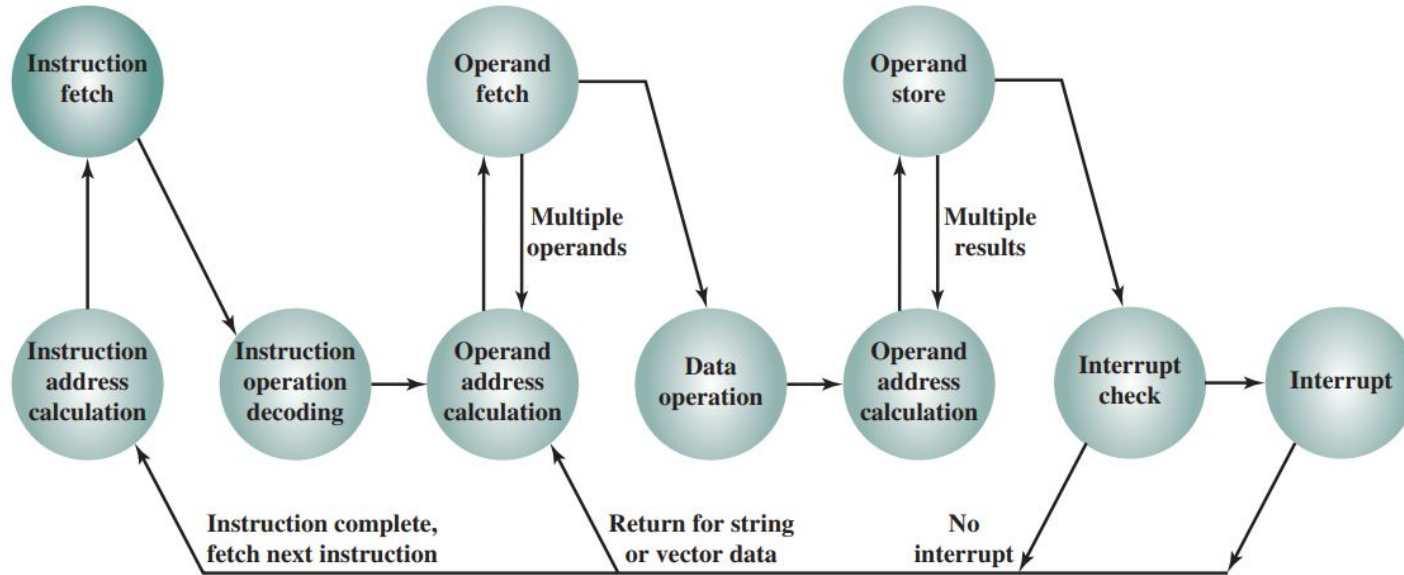


Fig: Instruction Cycle State Diagram, with Interrupts

Interruption Processing:

- Processor checks to see if any interrupts have occurred
 - indicated by the presence of an interrupt signal
- If no interrupts are pending
 - processor proceeds to the fetch cycle
 - fetches the next instruction of the current program.

Interrupt Processing (cont.):

- **If an interrupt is pending**, the processor does the following:
 - suspends execution of the program being executed, this entails:
 - Saving the PC
 - Saving the contents of the registers
 - **This is called saving the context of the program.**
 - sets the PC to the starting address of an interrupt handler routine
 - **The instruction cycle is restarted, but this time for the interrupt handler**
 - fetch the first instruction in the interrupt handler program, and so on.
 - Eventually the interrupt routine will terminate;
 - When the interrupt handler routine is completed:
 - Processor needs to resume execution of the original program
 - Processor restores contents of the previously saved registers
 - Continuing the execution of the previously saved PC
 - **This is called restoring the context of the program**

Interrupt Service Routine:

A specialized function in a computer's operating system that is designed to handle a specific interrupt request. When an interrupt occurs, the CPU transfers control to the corresponding ISR to process the interrupt.

- Determines the nature of interrupt.
- Performs the required actions.
- Context switching.

Interrupting mechanism is very useful!
Any disadvantages though?

Multiple interrupts:

Suppose at the same time:

- A key is pressed and
- Printer generates a signal after print operation....

How to handle them?

Approach 1:

Disable interrupts while an interrupt is being processed...

1. The processor will ignore interrupt request signals
 - a. Emphasis on the “ignore”, interrupt signals can still be generated.
2. After the interrupt handler routine completes:
 - a. Interrupts are enabled
 - b. the processor checks to see if any interrupts have occurred.

Any problem with this approach?

Approach 1:

Disadvantages:

1. Some interruptions may be more important than others but still gets ignored.
2. Data may be lost in the process.
3. Introduce additional latency in responding to time-critical events.
4. Loss of responsiveness in the system, especially in real-time or interactive applications.

Approach 2:

Allow each interruption to have a priority:

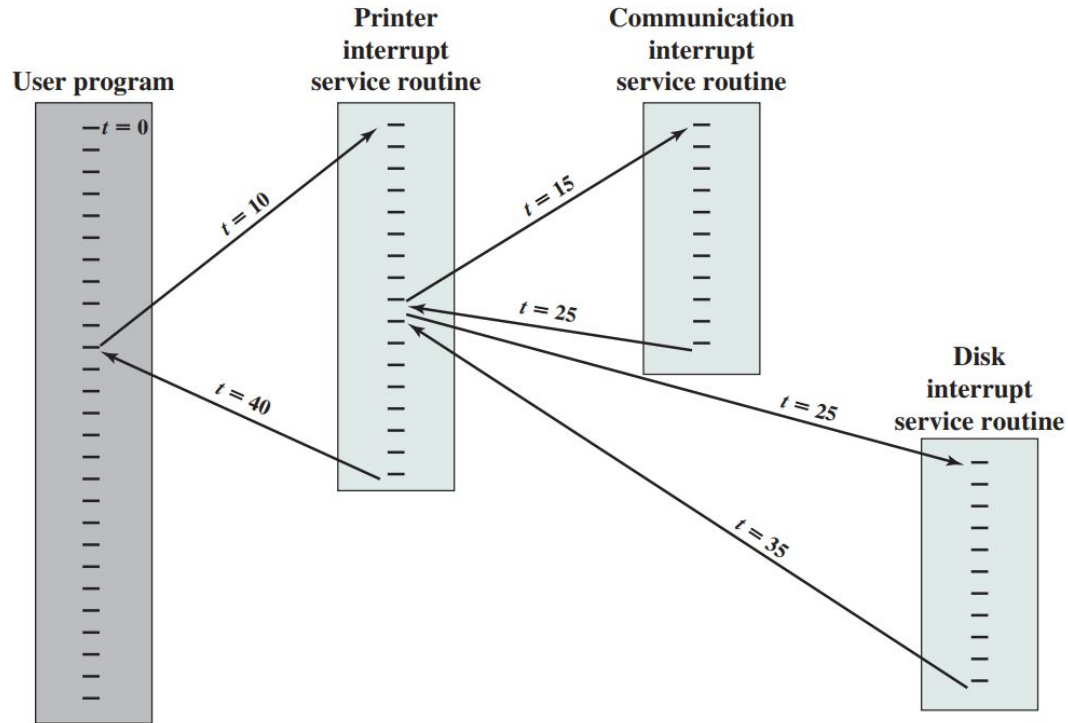
1. Higher priority interruptions can interrupt lower-priority interruptions.
2. This strategy is referred to as nested interrupt processing.

Example:

consider a system with three I/O devices: a printer, a disk, and a communications line, with increasing priorities of 2, 4, and 5, respectively. A user program begins at $t = 0$. At $t = 10$, a printer interrupt occurs. While this routine is still executing, at $t = 15$, a communications interrupt occurs. Again while this communication routine is executing, a disk interrupt occurs at $t = 20$.

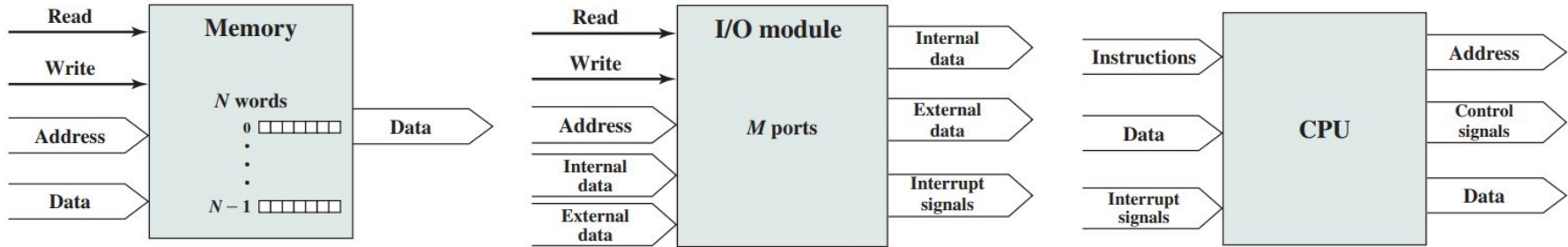
How will the interrupts be handled?

Example:



Interconnection Structure:

A computer is a network of basic modules. The collection of paths connecting the various modules is called the **interconnection structure**.



These modules need to communicate with each other in order to:

- Exchange data
- Exchange control signals

Interconnection Structure:

The most common interconnection structures are:

1. Bus Interconnection
2. Point-To-Point Interconnection

Bus Interconnection:

A bus is a communication pathway connecting two or more devices:

- Shared transmission medium.
- Connected devices can pick up the reception of all other devices.
- Only one device at a time can successfully transmit.

Bus Interconnection:

A bus consists of multiple **communication lines**:

- Each line transmits binary signals.
- A sequence of binary digits can be transmitted:
 - using a single line over time (i.e. sequentially).
 - several lines can be used (i.e. in parallel).

A bus connecting major computer components is called a **system bus**.

Bus Structure:

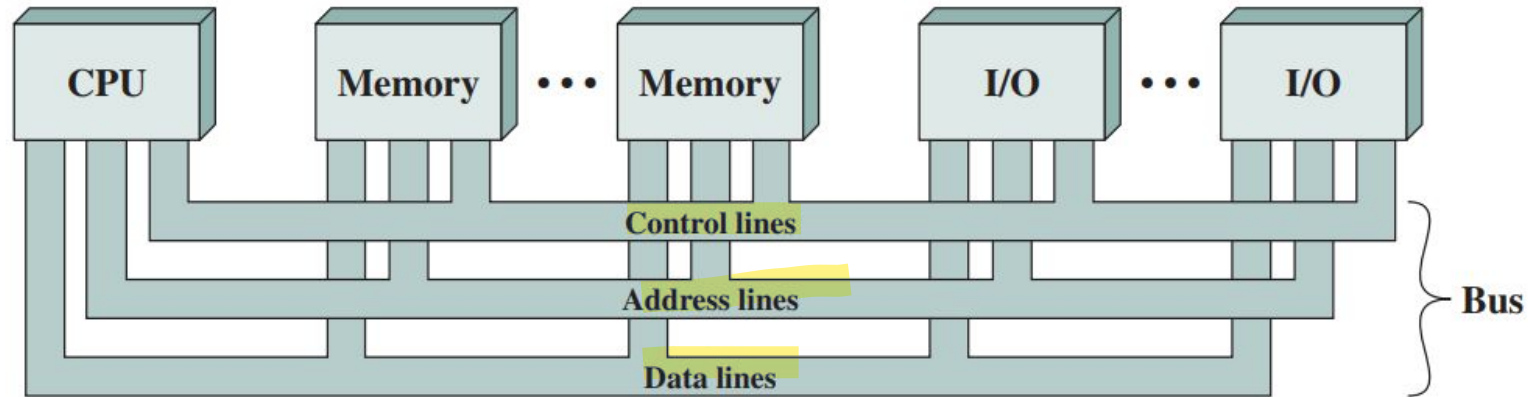


Fig: Instruction Cycle State Diagram, with Interrupts

Bus Structure:

Bus lines can be classified into three functional groups:

- Data:
 - for moving data among system modules.
- Address
 - for specifying the source or destination of the data.
- Control
 - for transmitting command information among the modules.

Data lines:

- The data bus may consist of 32, 64, 128, or even more separate lines:
 - a.k.a. width of the data bus;
- Each line can carry only 1 bit at a time:
 - the number of lines determines how many bits can be transferred at a time.
- Data bus width is key to system performance, e.g.:
 - if the data bus is 32 bits wide and each instruction is 64 bits long, each instruction requires two memory accesses.

Address Lines:

Used to designate the source or destination of the data on the data bus:

- The width of the address bus determines the maximum system memory;
- The address lines are generally also used to address I/O addresses;
 - Higher-order bits are used to select a particular module on the bus;
 - Lower-order bits select a memory location or I/O port within the module.

Control Lines:

- **Memory write:** Write bus data to a memory address.
- **Memory read:** read memory at memory address.
- **I/O write:** write bus data to an I/O address.
- **I/O read:** read data from an I/O address.
- **Bus request:** a module needs to gain control of the bus.
- **Bus grant:** a requesting module has been granted bus control.
- Many more control signals...

Any potential problems?

Performance issue...

If a great number of devices are connected, bus performance will suffer.

Main variables of bus performance:

- **Bus length:** the more devices attached to the bus, the greater the bus length and hence the greater the propagation delay.
- **Bus capacity:** bus has a certain data rate capacity.

Point-To-Point Interconnection:

Nowadays computers rely on P2P Interconnection instead of the bus structure for the following reasons:

- Electrical constraints:
 - encountered with increasing the frequency of wide synchronous buses.
- Difficulties of increasing bus data rate.

The point-to-point interconnection has **lower latency, higher data rate**, and **better scalability**.

Point-To-Point Interconnect Approach: **QPI** quick path interconnect

Main Characteristics:

- **Multiple Direct Connections**
 - Multiple components enjoy direct pairwise connections with other components – eliminates arbitration.
- **Layered Protocol Architecture**
 - Uses a layered protocol architecture rather than the simple use of signal.
- **Packetized Data Transfer**
 - Data are not sent in bit stream rather sent as a sequence of packets which include headers and error control codes.

Point-To-Point Interconnect Approach: QPI

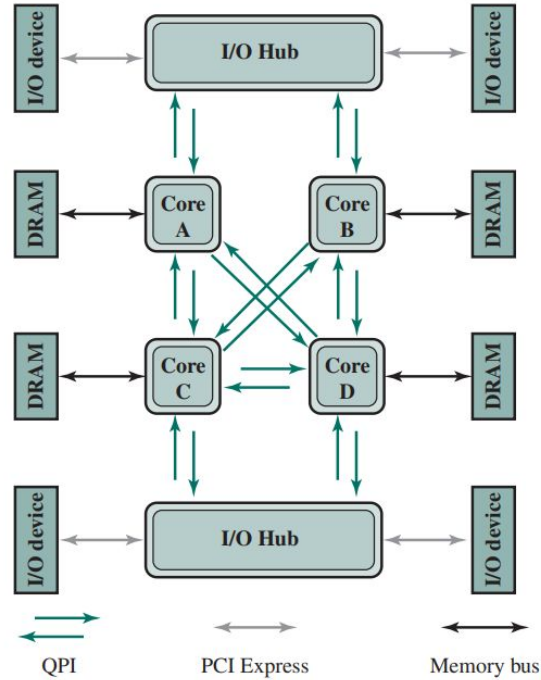


Fig: Multicore Configuration Using QPI

QPI Layers:

QPI is defined as a four-layer protocol architecture encompassing the following layers:

1. **Physical:** Consisting actual wires to carry signal as well as circuit and logic to support extra features to transmit and receive data.
2. **Link:** Responsible for reliable transmission and flow control.
3. **Routing:** Provides the framework for directing packets through the fabric.
4. **Protocol:** The high-level set of rules for exchanging packets of data between devices.

PCI Express:

Peripheral Component Interconnect Express, commonly abbreviated as **PCIe**. It is an interface standard for connecting high-speed input output (HSIO) components.

Key Features:

1. Faster data transfer rates compared to its predecessors.
2. A dedicated, point-to-point connection between the motherboard and the connected device.
 - a. reduces latency.
3. Allows the connection of various devices with different bandwidth requirements.

Reference:

1. Computer Organization and Architecture, by William Stallings, 10th edition.

Thank You!