Paper Name: Computer Architecture
Paper Code: CS401
Contact Hours/Week: 3
Credit: 3
Total Contact Hours: 36L

Objective(s)

- **To learn the basics of stored program concepts.**
- **To learn the principles of pipelining.**
- **To learn mechanism of data storage**
- **To distinguish between the concepts of serial, parallel, pipeline architecture.**

Outcome(s)

- **Learn pipelining concepts with a prior knowledge of stored program methods**
- **Learn about memory hierarchy and mapping techniques.**
- **Study of parallel architecture and interconnection network .**

Prerequisites:

1. Digital Logic
2. Computer organization
3. Computer Fundamentals
4. Programming Concept

Module – 1: [5L]

Introduction-

Introduction to basic computer architecture [1L]
Stored Program Concepts: Von Neumann & Havard Architecture [1L]
RISC VS CISC[1L]
Amdahl's law. [1L]
Performance Measure: MIPS, Benchmark Programs(SPECINT,SPECFP).[1L]

Module – 2: [6L]

Pipelining-
Pipelining: Basic concepts, Linear vs. Non Linear, Static vs. Dynamic, Unifunction vs. Multifunction [2L]
Instruction Pipeline [1L]
Arithmetic pipeline [1L]
Hazards: Data hazards, control hazards and structural hazards [1L]
Techniques for handling hazards [1L]

Module – 3: [4L]

Instruction-level parallelism-
Instruction-Level Parallelism: Basic Concepts [1L]
Techniques For Increasing ILP, Superscalar, Super Pipelined [1L]
VLIW Processor Architectures [1L]
Array and Vector Processors [1L]
Module – 4: [5L]

Memory Hierarchy: Internal Memory, Main Memory, Cache Memory, Secondary memory[2L]
Mapping Technique in cache memory: Direct, Full Associative and Set Associative[2L]
Performance Implementation in Cache Memory.[1L]

Module – 5: [16L]

Multiprocessor architecture-

Introduction to Parallel Architecture-Different Classification scheme, Performance of Parallel Computers, PRAM model(EREW,CREW,CRCW) [6L]
Interconnection Network(Omega,Baseline,Butterfly,Crossbar)[6L]
Multi-Core Processor with case study(INTEL)[2L]
Different Classification scheme:Serial Vs. Parallel, Pipeline vs. Parallel [2L]

Text Book:
1. Patterson D.A. and Hennessy , J.L. “Computer architecture a quantitative approach”, 2nd ed., Morgan Kaufman, 1996
2. Stone, H.S., “Advanced Computer”, Addison Wesley, 1989
3. Siegel, H.J., “Interconnection Network for Large Scale parallel Processing”, 2nd Ed., McGraw Hill, 1990

Reference Book:
1. Hwang & Briggs—Computer Architecture & Parallel Processing, TMH
2. Hayes J. P., “Computer Architecture & Organisation”, McGraw Hill

3. Design and Analysis of Prallel Algorithm-Schim

Lesson Plan for B.Tech Computer Science and Engineering Programme(Autonomy)
Paper Name: Computer Architecture
Paper Code: CS401
Contact Hours/Week: 3
Credit: 3
Total Contact Hours: 36L

Module No.	Course Content	Lecture Required	Reference / Text Books
1	Module – 1: [5L] Introduction- Introduction to basic computer architecture [1L] Stored Program Concepts: Von Neumann & Havard Architecture [1L] RISC VS CISC[1L] Amdahl's law. [1L] Performance Measure: MIPS, Benchman Programs(SPECINT,SPECFP).[1L]	5L	**Text Book:** 1. Patterson D.A. and Hennessy , J.L. "Computer architecture a quantitative approach", 2nd ed., Morgan Kaufman, 1996 **Reference Book:** 1. Hwang & Briggs—Computer Architecture & Parallel Processing, TMH
	Module – 2: [6L] Pipelining- Pipelining: Basic concepts, Linear vs. Non Linear, Static vs. Dynamic, Unifunction vs. Multifunction [2L] Instruction Pipeline [1L] Arithmetic pipeline [1L] Hazards: Data hazards, control hazards and structural hazards [1L] Techniques for handling hazards [1L]	6L	**Text Book:** 1. Patterson D.A. and Hennessy , J.L. "Computer architecture a quantitative approach", 2nd ed., Morgan Kaufman, 1996 **Reference Book:** 1. Hwang & Briggs—Computer Architecture & Parallel Processing, TMH
	Module – 4: [5L] Memory Hierarchy: Internal Memory, Main Memory, Cache Memory, Secondary memory[2L] Mapping Technique in cache memory:	5L	**Text Book:** 1. Patterson D.A. and Hennessy , J.L. "Computer architecture a quantitative approach",

	Direct, Full Associative and Set Associative[2L] Performance Implementation in Cache Memory.[1L]		2nd ed., Morgan Kaufman, 1996 **Reference Book:** 2. Hayes J. P., "Computer Architecture & Organisation", McGraw Hill
	<u>Module – 5:</u> [16L] Multiprocessor architecture- Introduction to Parallel Architecture-Different Classification scheme, Performance of Parallel Computers, PRAM model(EREW,CREW,CRCW) [6L] Interconnection Network(Omega,Baseline,Butterfly,Crossbar)[6L] Multi-Core Processor with case study(INTEL)[2L] Different Classification scheme:Serial Vs. Parallel, Pipeline vs. Parallel [2L]	16L	Text Book 3. Siegel, H.J., "Interconnection Network for Large Scale parallel Processing", 2nd Ed., McGraw Hill, 1990 **Reference Book:** 1. Hwang & Briggs—Computer Architecture & Parallel Processing, TMH 3. Design and Analysis of Prallel Algorithm-Schim G. Akl

MODULE 1

INTRODUCTION

LECTURE: 1

Introduction to Computer Architecture

A general-purpose computer has these parts:

1. processor: the ``brain" that does arithmetic, responds to incoming information, and generates outgoing information
2. primary storage (memory or RAM): the ``scratchpad" that remembers information that can be used by the processor. It is connected to the processor by a system bus (wiring).
3. system and expansion busses: the transfer mechanisms (wiring plus connectors) that connect the processor to primary storage and input/output devices.

A computer usually comes with several input/output devices: For input: a keyboard, a mouse; For output, a display (monitor), a printer; For both input and output: an internal disk drive, memory key, CD reader/writer, etc., as well as connections to external networks.

For reasons of speed, primary storage is connected ``more closely" to the processor than are the input/output devices. Most of the devices (e.g., internal disk, printer) are themselves primitive computers in the sense that they contain simple processors that help transfer information to/from the processor to/from the device.

Here is a simple picture that summarizes the above:

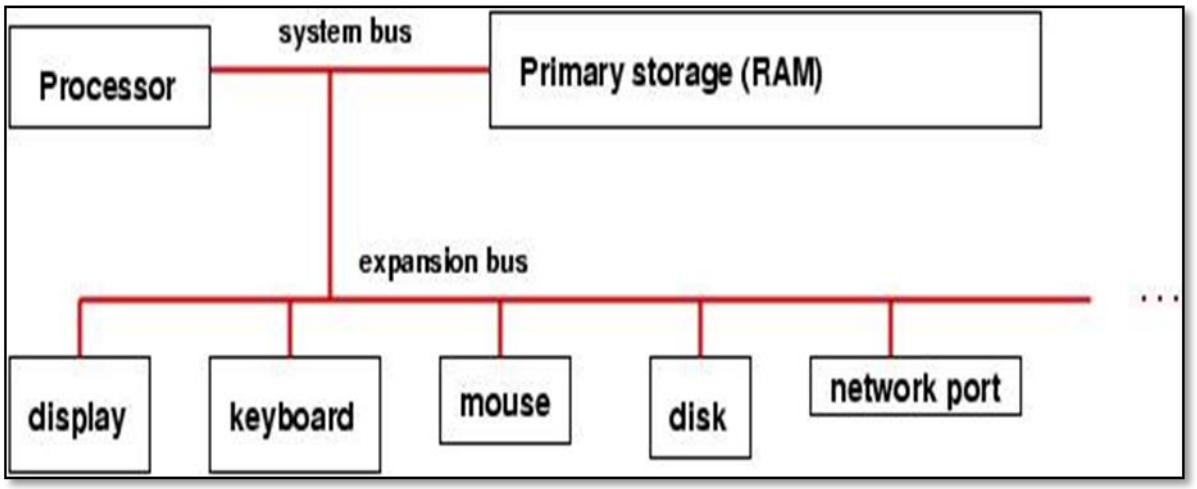


Fig: 1.1 Block Diagram of Computer System

Information and binary coding

For humans, information can be pictures, symbols, words, sounds, movements, and more. A typical computer has a keyboard and mouse so that words and movements can be sent to the processor as information. The information must be converted into electrical off-on (``0 and 1") pulses that travel on the bus and arrive to the processor, which can save them in primary storage.

It is premature to study precisely how numbers and symbols can be represented as off-on (0-1) pulses, but here is review of base-2 (*binary*) coding of numbers, which is the concept upon which computer information is based:

number binary coding
0 0000
1 0001
2 0010
3 0011
4 0100
5 0101
6 0110
7 0111
8 1000
...
14 1110
15 1111

and so on. It is possible to do arithmetic in base two, e.g. 3+5 is written:
0011
+0101

1000
The addition works like normal (base-10) arithmetic, where 1 + 1 = 10 (0 with a carry of 1). Subtraction, multiplication, etc., work this way, too, and it is possible to wire an electrical circuit that mechancially does the addition of the 0s and 1s. Indeed, a processor uses such a wiring, which operates on binary numbers held in registers, where a register is a sequence of bits (electronic ``flip-flops" each of which can remember a 0 or 1). Here is a picture of an 8-bit register that holds the number 9:
+--+--+--+--+--+--+--+--+
| 0| 0| 0| 0| 1| 0| 0| 1|
+--+--+--+--+--+--+--+--+

A processor has multiple such registers, and it can compute 3+5 by placing 3 (0000 0011) and 5 (0000 0101) into two registers and then using the wiring between the registers to compute the sum, which might be saved in a third register. A typical, modern register has 32 bits, called a *fullword*. Such a register can store a value in the approximate range of -2 billion to +2 billion.

When an answer, like 3+5 = 8, is computed, the processor might copy the answer to primary storage to save it for later use. Later, the processor can copy the number from storage back into a register and do more arithmetic with it.

Central processing unit

The processor is truly the computer --- it is wired to compute arithmetic and related operations on numbers that it can hold in its data registers. A processor is also called a Central Processing Unit (CPU).

Here is a simplistic picture of the parts of a processor:

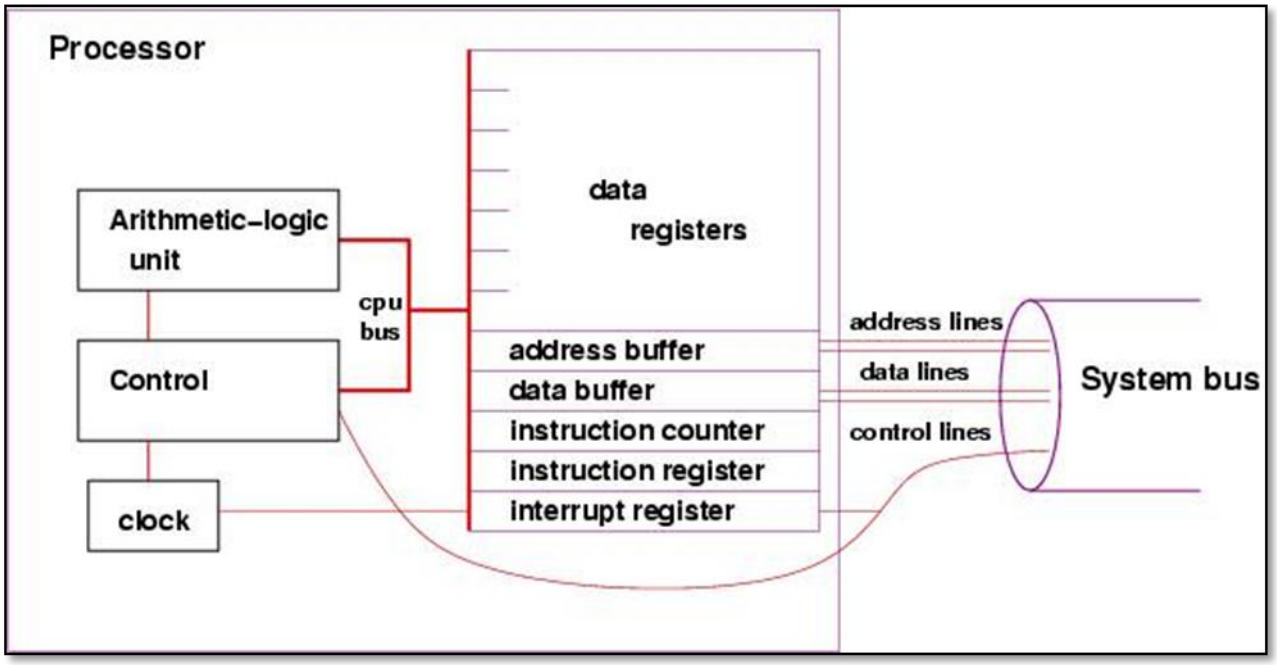


Fig: 1.2 Block Diagram of Parts of Processor

- The data registers hold numbers for computation, as noted earlier.
- There is a simple *clock* --- a pulse generator --- that helps the Control Unit do instructions in proper time steps.
- The *arithmetic-logic unit (ALU)* holds the wiring for doing arithmetic on the numbers held in the *data registers*. (Review the addition example above.)
- The *control unit* holds wiring that triggers the arithmetic operations in the ALU. How does the control unit know to request an addition or a subtraction? The answer is: it obtains instructions, one at a time, that have been stored in primary storage.
- The *instruction counter* is a register that tells the control unit where to find the instruction that it must do. (The details will be explained shortly.)
- The *instruction register* is where the instruction can be copied and held for study by the control unit,
- The *address buffer* and *data buffer* are two registers that are a ``drop-off" point when the processor wishes to copy information from a register to primary storage (or read information from primary storage to a register). We study them later.
- The *interrupt register* is studied much later.

A processor's speed is measured in Hertz (a kind of vibration speed) and is literally the speed of the computer's internal clock; the larger the Hertz number, the faster the processor.

Primary storage

Primary storage (also called *random-access memory --- RAM*) is literally a long sequence of fullwords, also called *cells*, where numbers can be saved for later use by the processor. (Recall that a fullword is 32 bits). Here is a simplistic picture:

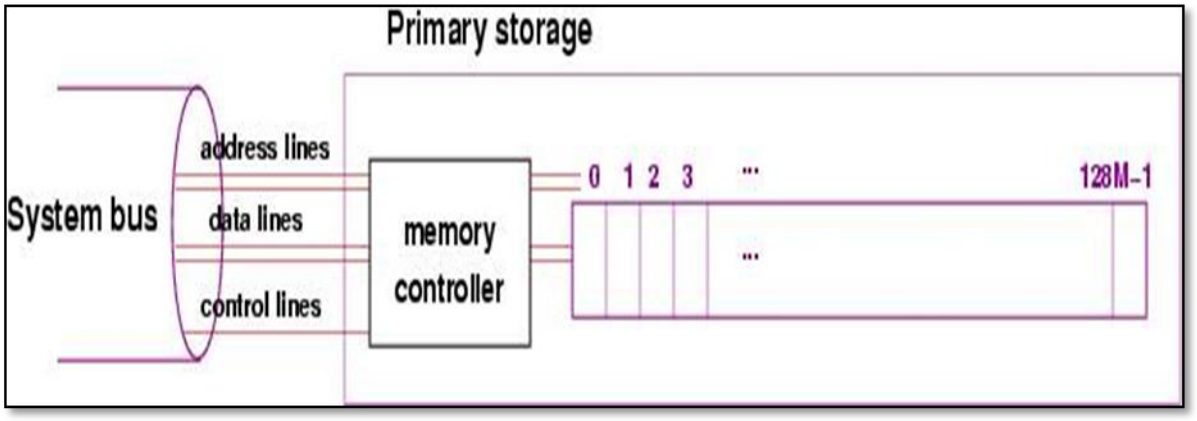


Fig: 1.3 System Bus and Memory Connection

The picture shows that each fullword (cell) is numbered by a unique *address* (analogous to street addresses for houses), so that information transferred from the processor can be saved at a specific cell's address and can be later retrieved by referring to that same address.

The picture shows an additional component, the *memory controller*, which is itself a primitive processor that can quickly find addresses and copy information stored in the addresses to/from the system bus. This works faster than if the processor did the work of reaching into storage to extract information.

When a number is copied from the processor into storage, we say it is *written*; when it is copied from storage into the processor, we say it is *read*.

As the diagram suggests, the *address lines* in the system bus are wires that transfer the bits that form the address of the cell in storage that must be read or written (the address is transmitted from the processor's address buffer --- see the previous section); the *data lines* are wires that transfer the information between the processor's data buffer and the cell in storage; and the control lines transmit whether the operation is a read or write to primary storage.

The tradition is to measure size of storage in *bytes*, where 8 bits equal one byte, and 4 bytes equal one full word. The larger the number, the larger the storage.

Stored programs

In the 1950's, John von Neumann realized that primary storage could hold not only numbers, but patterns of bits that represented *instructions* that could tell the processor (actually, tell the processor's control unit) what to do. A sequence of instructions was called a *program*, and this was the beginning of *stored-program, general purpose computers*, where each time a computer was started, it could receive a new program in storage, which told the processor what computations to do.

Here is a simplistic example of a stored program that tells the processor to compute the sum of three numbers held in primary storage at addresses, 64, 65, and 66 and place the result into the cell at address 67:

LOAD (read) the number at storage address 64 into data register 1
LOAD the number at storage address 65 into data register 2
ADD register 1 to register 2 and leave the sum in register 2
LOAD the number at address 66 to register 1
ADD register 1 to register 2 and leave the sum in register 2
STORE (write) the value in register 2 to storage address 67
instructions like LOAD, ADD, and STORE can be represented as bit patterns that are copied into the processor's instruction register.

Here is a simple coding of the six-instruction program, which is situated at addresses 1-6 of primary storage (and the numbers are at 64-66). The instructions are coded in bit patterns, and we assume that LOAD is 1001, ADD is 1010, and STORE is 1011. Registers 1 and 2 are 0001 and 0010. Storage addresses 64 -- 67 are of course 0100 0000 to 0100 0011.

The format of each instruction is: IIII RRRR DDDD DDDD, where IIII is the coding that states the operation required, RRRR is the coding of which data register to use, and DDDD DDDD is the data, which is either a storage address or another register number.

PRIMARY STORAGE

address: contents
------- --------
0: ...
1: 1001 0001 0100 0000
2: 1001 0010 0100 0001
3: 1010 0010 0000 0001
4: 1001 0001 0100 0010
5: 1010 0010 0000 0001
6: 1011 0010 0100 0011
7: ...
64: 0000 0000 0000 0100
65: 0000 0000 0000 0011
66: 0000 0000 0000 0001
67: ...
...

(Note: I have shortened the instructions to 16 bits, rather than use 32, because I got tired typing lots of zeros!)

The example is a contrived, but it should convince you that it is indeed possible to write instructions in terms of binary codings that a control unit can decode, disassemble, and execute.

It is painful for humans to read and write such codings, which are called *machine language*, and there are abbreviations, called *assembly language*, that use text forms. Here is a sample assembly-language version of the addition program:

```
LOAD R1 64
LOAD R2 65
ADD R2 R1
LOAD R1 66
ADD R2 R1
STORE R2 67
```

Instruction cycle

The *instructor cycle* are the actions taken by the processor to execute one instruction. Each time the processor's clock pulses (ticks) the control unit does these steps: (actually, modern processors do multiple instruction cycles for each clock pulse)

1. uses the number in the instruction counter to *fetch* an instruction from primary storage and copy it into the instruction register
2. reads the pattern of bits in the instruction register and *decodes* the instruction
3. based on the decoding, tells the ALU to *execute* the instruction, which means that the ALU manipulates the registers accordingly.
4. There is a fourth step in the instruction cycle, an *interrupt check*, that we study later.

Of course, the control unit is not alive, and it does not ``read'' or ``tell'' anything to anyone, but there is wiring between electrical components that propagate electrical 0-1 signals --- a kind of falling domino game--- that gives the appearance of conscious execution.

Here is a small example. Say that the clock has ``ticked'' (pulsed), and the instruction register holds 3. Say that address 3 in primary storage holds the coding of the instruction, ADD R2 R1. The instruction cycle might go like this:

1. Fetch: Consult the instruction counter; see it holds 0000 0011, that is, 3. Signal the memory controller to copy the contents of the cell at address 0000 0010 into the data buffer.

   When the instruction arrives, copy it from the data buffer into the instruction register.

   Increment the instruction counter to 4 (that is, 0000 0100).

2. Decode: Read the first (leading or high-order) bits and see that they indicate an ADD. Extract the bits that state the two registers to be added, here, R2 and R1.

3. Execute: Signal the ALU to add the values in registers 1 and 2 and place the result in register 2.

The previous description reads a bit tediously. This is OK, because the processor is incredibly fast. Nonetheless, modern processors can be made even faster, because while the ALU is doing the execution step, the controller can start the fetch-and-decode steps of the *next* instruction cycle. This form of speedup is called *pipelining* and is a topic intensively studied in computer architecture.

The forms of instruction that the processor can execute are called the *instruction set*.

There are these forms of instructions found in an instruction set:

1. data transfer between storage and registers (LOAD and STORE)
2. arithmetic and logic (ADD, SUBTRACT, ...)
3. control (test and branch) (the ALU perhaps resets the instruction counter)
4. input and output (the ALU sends a request on the system bus to an input/output device to read or write new information into storage)

Even small examples are painful to write in assembly language, and people quickly developed simpler notations that could be mechanically converted to assembly (which could itself be mechanically converted into base-2 codings).

FORTRAN (formula translator language) is a famous example, developed in the 1950's by John Backus. When a human writes a program using FORTRAN, she writes a set of mathematical equations that the computer executes. Instead of using specific numerical storage addresses, names from algebra (``variable names''), like x and y, can be used instead.

Here is an example, coded in FORTRAN, that places a value in a storage cell, named x, and then divides it by 2, saving the result again in the same cell:

```
x = 3.14159
x = x / 2
```

And here is an example that divides x by y, saving the answer in x's cell, *provided that y has a non-zero value*:

```
if ( y .NEQ. 0 )   x = x / y
```

(read this as ``if y not-equal-to 0, then compute x = x / y'')

With some work, one can write a program that mechanically translates FORTRAN programs into (long) sequences of machine code; such a program is called a *compiler*. There is another ``translation program,'' called an *interpreter*, which does not convert a program to machine code, but instead reads a program one line at a time and tells the processor to execute ``pre-fabricated'' sequences of instructions that match the program's lines. These concepts are developed in another lecture.

Languages like FORTRAN (and COBOL and LISP and C and Java and ...) are called *high-level programming languages*.

Secondary storage: disks

The previous section stated that programs and numbers can be saved in primary storage. But there is a limited amount of primary storage, and it is used to hold the program that the computer executes now. Programs and information that are saved for later use can be copied to *secondary storage*, such as the internal disk that is common to almost all computers.

Although it looks and operates differently than primary storage, it is perfectly fine to think of disk storage (and other forms of secondary storage, like a memory key or a CD), as a variant of primary storage, connected to the processor by means of the system bus, using its own controller to help read and write information. The main distinction is that secondary storage is *cheaper* (to buy) than primary storage, but it is *slower* to read and write information to and from it.

A typical computer uses disk secondary storage to hold a wide variety of programs that can be copied into primary storage for execution, as requested by the user. Secondary storage is also used to archive data files.

Secondary-storage devices are activated when the processor executes a READ or WRITE instruction. These instructions are not as simple to do as the LOAD and STORE instructions, because the secondary-storage devices are so slow, and the processor should not waste time, doing nothing, waiting for the device to finish its work.

The solution is: *The processor makes the request for a read or write and then proceeds to do other work*.

Consider how a processor might execute a WRITE instruction to the disk; here is how the instruction cycle might go:

1. Fetch: The control unit obtains the instruction from primary storage and places it in the instruction register, as usual.
2. Decode: The control unit reads the instruction and determines that it is a WRITE. It extracts that name of the device to be read (the disk), it extracts the address on the device where the information should be written, and it extracts the name of the register than holds the information to be written.
3. Execute: The control unit writes the address and data to the disk's *address buffer* and *data buffer*, which are two fullwords in primary storage. When these writes are finished, the controller signals the disk along the control lines of the system bus that there is information waiting for it in primary storage.

Now that the processor has initiated the disk-write, it proceeds to the next instruction to execute, and at the same time, the disk starts to spin, its own controller does a read of primary storage for the address and data information saved there, and finally, the data is written from primary storage to the disk.

Each secondary-storage device has its own ``buffers" reserved for it in primary storage --- this is simpler than wiring the processor for buffers for each possible storage device.

An important ``secondary storage" device (actually, it is an output device!) is the computer's display. A typical display is a huge grid of pixels (colored dots), each of which is defined by a trio of red-green-blue numerical values. The display has a huge buffer in primary storage, where there is one (or more) cell that describes the color of each pixel. A write instruction executed by the processor causes the display's buffer to be altered at the appropriate cells, and the display's controller (called the ``video controller") reads the information in the buffer and copies its contents to the display, thus repainting the display.

To summarize, here is a picture of a computer with buffers reserved for input/output devices in primary storage:

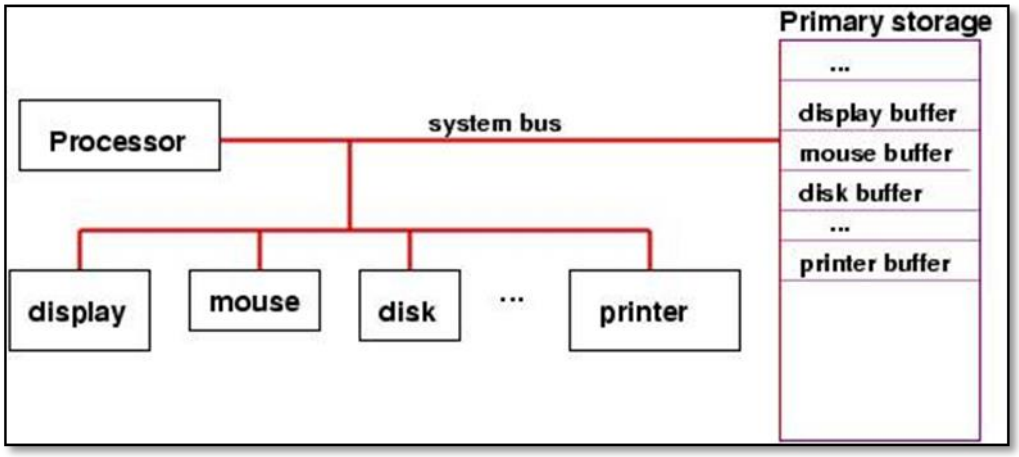


Fig: 1.4 Block Diagram of Buffer Reserved for Peripheral Devices

It is important to see in the picture that (the controllers in) the various storage devices can use the system bus to read/write from primary storage *without bothering the processor*. So, input and output can proceed at the same time that the processor executes instructions.

When a computer is connected to an outside network, the network can also be considered a kind of secondary-storage device that responds to read and write instructions, but the format of the reads and writes is far more complex --- they must include the address of the destination computer, the kind of data transmitted, the stage of interaction that is being done, etc. So, there are standardized patterns of bits, called *protocols*, that must be transmitted as ``reads" and ``writes" from the processor to the system bus to the port to the network. To accomplish a complete read or write, there might well be multiple transmissions from processor to bus to port to network. The design of protocols is a crucial issue to computer networks.

Interrupts

The previous section noted that a processor should not wait for a secondary-storage device to complete a write operation. But what if the processor asks the device to perform a read operation, how will the processor know when the information has been successfully read and deposited into the device's buffer in storage?

Here is a second, similar situation: A human presses the mouse's button, demanding attention from the processor (perhaps to start or stop a program or to provide input to the program that the processor is executing). How is the processor signalled about the mouse click?

To handle these situations, all processors are wired for interruption of their normal executions. Such an interruption is called an *interrupt*.

Recall the standard execution cycle:

1. fetch
2. decode
3. execute
4. check for interrupts

and recall the extra register, the *interrupt register*, that is embedded in the processor:

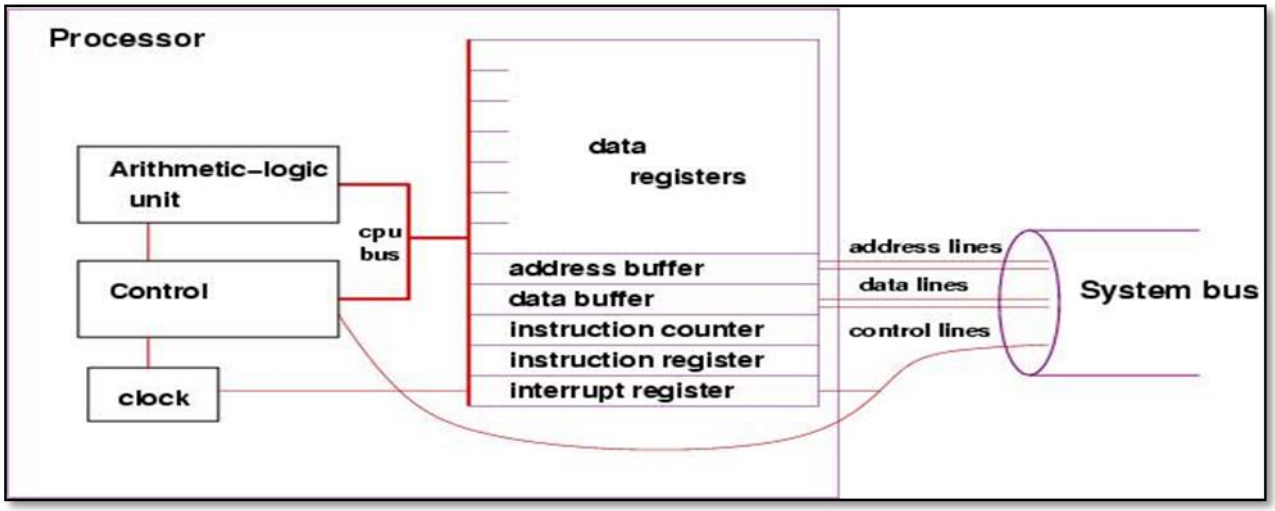


Fig: 1.5 Interrupt Register

The interrupt register is connected to the the system bus, so that when a secondary storage device has completed an action, it signals the control unit by setting to 1 one of the bits in the interrupt register.

Now, we can explain the final step of the execution cycle, the check for interrupts: After the execution step, the control unit examines the contents of the interrupt register, checking to see if any bit in the register is set to 1. If all bits are 0, then no device has completed an action, so the processor can start a new instruction.

But if a bit is set to 1, then there is an *interrupt* --- the processor must pause its execution and do whatever instructions are needed:

For example, perhaps the user has pressed the mouse button. The device controller for the mouse sends a signal on the system bus to set to 1 the bit for a ``mouse interrupt" in the interrupt register. When the control unit examines the interrupt register at the end of its current execution cycle, it sees that the bit for the mouse is set to 1. So, it resets the bit to 0 and *resets the instruction counter to the address of the program that must be executed whenever the mouse button is pressed*. Once the mouse-button program finishes, the processor can resume the work it was doing.

The mouse-button program is called an *interrupt handler*.

The previous story skipped a lot of details: Where does the processor find the interrupt-handler program for the mouse? What happens to the information resting in the registers if we must pause execution and start a new program, namely, the interrupt handler? What if more than one interrupt bit is set? What if a new interrupt bit gets set while the processor is executing the mouse-button program?

Some of the answers are a bit complex. Based on this picture, we can provide simplistic answers:

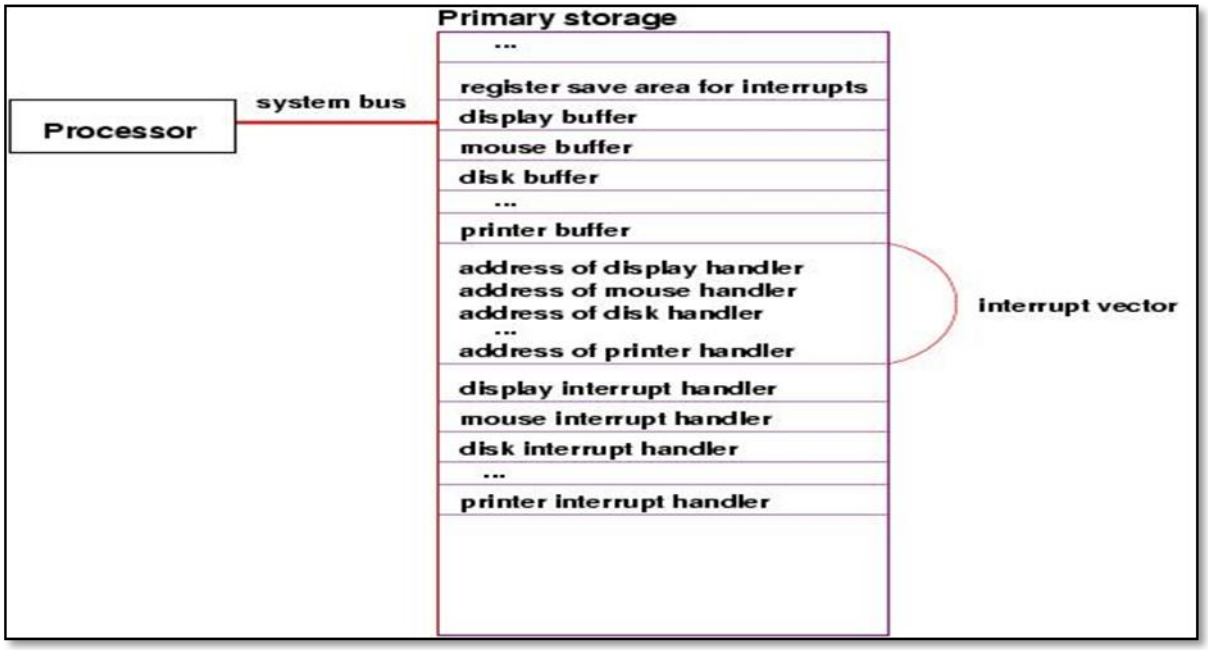


Fig: 1.6 Interrupt Vector

Cells in primary storage hold the addresses of the starting instructions for each of the interrupt handlers for the devices. The sequence of addresses is called an *interrupt vector*. The processor finds the address of the needed interrupt handler from the interrupt vector.

Before the processor starts executing an interrupt handler, it must copy the current values in all its registers to a *register-save area* in primary storage. When the interrupt handler is

finished, the values in the register-save area are copied back into the registers in the processor, so that the processor can resume what it was doing before the interrupt.

The case of multiple interrupts is not covered here, but the basic idea is that an executing interrupt handler can itself be interrupted and its own registers can be saved.

The Operating System

The startup- and manager-program is the *operating system*. When the computer is first started, the operating system is the program that executes first. As noted, it initializes the computer's storage as well as the controllers for the various devices. The interrupt handlers just discussed as considered parts of the operating system.In addition, the operating system helps the processor execute multiple programs ``simultaneously'' by executing each program a bit at a time. This technique, which is studied carefully in another lecture, is crucial so that a human user can start and use, say, a web browser and a text editor, at the same time.

The operating system is especially helpful at managing one particular output device --- the computer's display. The operating system includes a program called the *window manager*, which when executed, paints and repaints as needed the pixels in the display. The window manager must be executing ``all the time,'' even while the human user starts programs like a web browser, text editor, etc.The operating system lets the window manager repaint the display in stages: when the window-manager program repaints the display, it must execute a sequence of WRITE instructions. When the processor executes one of the WRITE instructions, this triggers the display's controller to paint part of the display. When the display controller finishes painting the part, it sets a bit in the interrupt register so that the interrupt handler for the display can execute and tell the processor to restart the window manager and continue repainting the display. In this way, the window manager is executing ``all the time,'' in starts and stops.Here is a revised picture of the computer's storage, which shows the inclusion of the operating system (``OS'') and the division of the remaining storage for the multiple user programs that are executing:

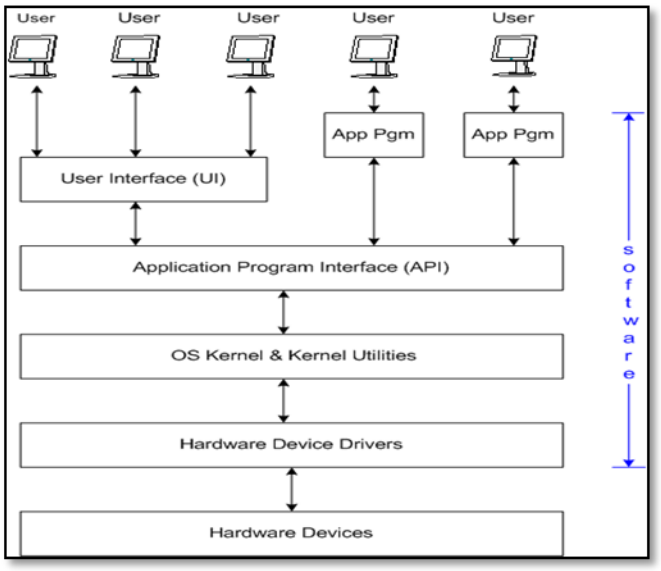


Fig: 1.7 Organisation of Operating System

VON NEUMANN ARCHITECTURE

LECTURE: 2

The von Neumann architecture, which is also known as the von Neumann model and Princeton architecture, is a computer architecturebased on that described in 1945 by the mathematician and physicist John von Neumann and others in the First Draft of a Report on the EDVAC. This describes a design architecture for an electronic digital computer with parts consisting of a processing unit containing an arithmetic logic unit and processor registers; a control unit containing an instruction register and program counter; a memory to store both data and instructions; external mass storage; and input and output mechanisms. The meaning has evolved to be any stored-program computer in which an instruction fetch and a data operation cannot occur at the same time because they share a common bus. This is referred to as the von Neumann bottleneck and often limits the performance of the system.

The design of a von Neumann architecture machine is simpler than that of a Harvard architecture machine, which is also a stored-program system but has one dedicated set of address and data buses for reading data from and writing data to memory, and another set of address and data buses for instruction fetching.

A stored-program digital computer is one that keeps its program instructions, as well as its data, in read-write, random-access memory (RAM). Stored-program computers were advancement over the program-controlled computers of the 1940s, such as the Colossus and the ENIAC, which were programmed by setting switches and inserting patch cables to route data and to control signals between various functional units. In the vast majority of modern computers, the same memory is used for both data and program instructions, and the von Neumann vs. Harvard distinction applies to the cache architecture, not the main memory (split cache architecture).

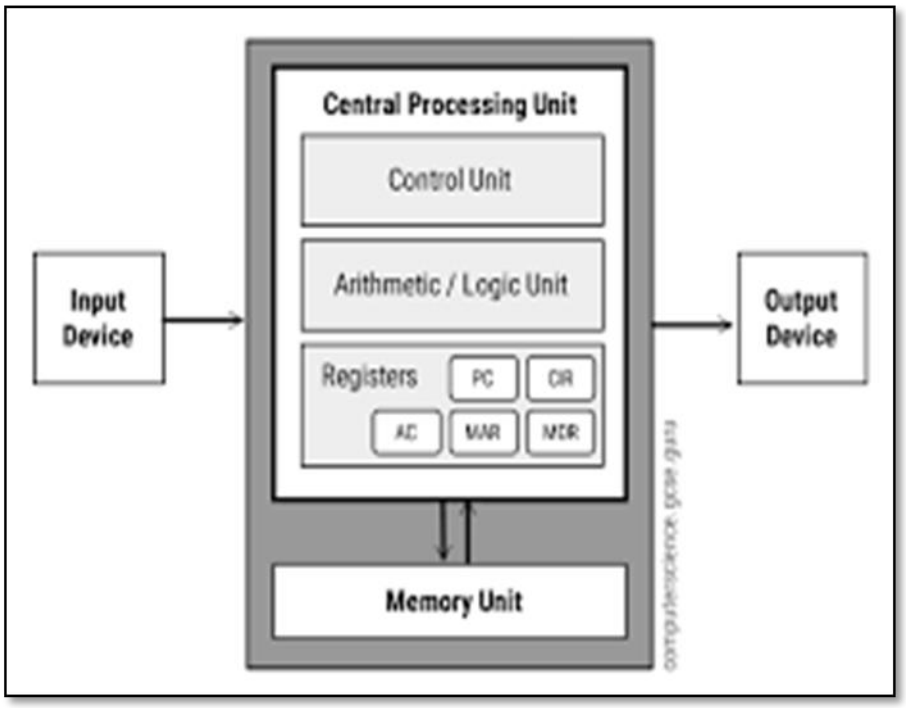


Fig: 1.8 Von Neumann Architecture

HARVARD ARCHITECTURE

The **Harvard architecture** is a computer architecture with physically separate storage and signal pathways for instructions and data. The term originated from the Harvard Mark I relay-based computer, which stored instructions on punched tape (24 bits wide) and data in electro-mechanical counters. These early machines had data storage entirely contained within the central processing unit, and provided no access to the instruction storage as data. Programs needed to be loaded by an operator; the processor could not initialize itself.

Today, most processors implement such separate signal pathways for performance reasons, but actually implement a modified Harvard architecture, so they can support tasks like loading a program from disk storage as data and then executing it.

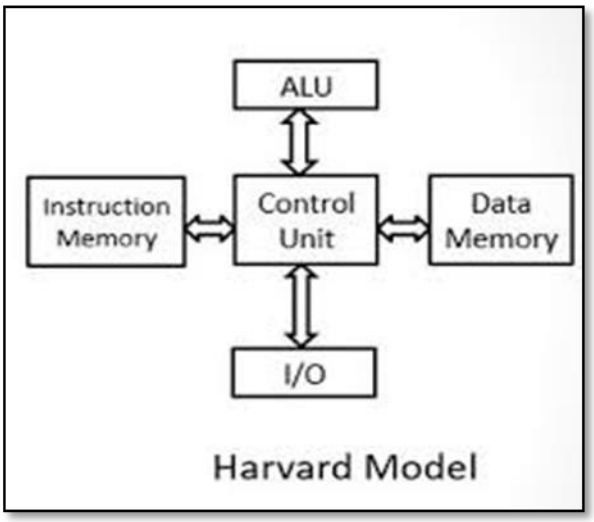


Fig: 1.9 Harvard Architecture

Memory Details

In Harvard architecture, there is no need to make the two memories share characteristics. In particular, the word width, timing, implementation technology, and memory address structure can differ. In some systems, instructions for pre-programmed tasks can be stored in read-only memory while data memory generally requires read-write memory. In some systems, there is much more instruction memory than data memory so instruction addresses are wider than data addresses.

Contrast with von Neumann architectures

Under pure von Neumann architecture the CPU can be either reading an instruction or reading/writing data from/to the memory. Both cannot occur at the same time since the instructions and data use the same bus system. In a computer using the Harvard architecture, the CPU can both read an instruction and perform a data memory access at the same time, even without a cache. A Harvard architecture computer can thus be faster for a given circuit complexity because instruction fetches and data access do not contend for a single memory pathway.

Also, a Harvard architecture machine has distinct code and data address spaces: instruction address zero is not the same as data address zero. Instruction address zero might identify a

twenty-four bit value, while data address zero might indicate an eight-bit byte that is not part of that twenty-four bit value.

Contrast with modified Harvard architecture

A modified Harvard architecture machine is very much like a Harvard architecture machine, but it relaxes the strict separation between instruction and data while still letting the CPU concurrently access two (or more) memory buses. The most common modification includes separate instruction and data caches backed by a common address space. While the CPU executes from cache, it acts as a pure Harvard machine. When accessing backing memory, it acts like a von Neumann machine (where code can be moved around like data, which is a powerful technique). This modification is widespread in modern processors, such as the ARM architecture, Power Architecture and x86 processors. It is sometimes loosely called a Harvard architecture, overlooking the fact that it is actually "modified".

Another modification provides a pathway between the instruction memory (such as ROM or flash memory) and the CPU to allow words from the instruction memory to be treated as read-only data. This technique is used in some microcontrollers, including the Atmel AVR. This allows constant data, such as text strings or function tables, to be accessed without first having to be copied into data memory, preserving scarce (and power-hungry) data memory for read/write variables. Special machine language instructions are provided to read data from the instruction memory, or the instruction memory can be accessed using a peripheral interface^ . (This is distinct from instructions which themselves embed constant data, although for individual constants the two mechanisms can substitute for each other.)

Speed:

In recent years, the speed of the CPU has grown many times in comparison to the access speed of the main memory. Care needs to be taken to reduce the number of times main memory is accessed in order to maintain performance. If, for instance, every instruction run in the CPU requires an access to memory, the computer gains nothing for increased CPU speed—a problem referred to as being memory bound.

It is possible to make extremely fast memory, but this is only practical for small amounts of memory for cost, power and signal routing reasons. The solution is to provide a small amount of very fast memory known as a CPU cache which holds recently accessed data. As long as the data that the CPU needs are in the cache, the performance is much higher than it is when the cache has to get the data from the main memory.

Internal vs. external design:

Modern high performance CPU chip designs incorporate aspects of both Harvard and von Neumann architecture. In particular, the "split cache" version of the modified Harvard architecture is very common. CPU cache memory is divided into an instruction cache and a data cache. Harvard architecture is used as the CPU accesses the cache. In the case of a cache miss, however, the data is retrieved from the main memory, which is not formally divided into separate instruction and data sections, although it may well have separate memory controllers used for concurrent access to RAM, ROM and (NOR) flash memory.

Thus, while a von Neumann architecture is visible in some contexts, such as when data and code come through the same memory controller, the hardware implementation gains the efficiencies of the Harvard architecture for cache accesses and at least some main memory accesses.

In addition, CPUs often have write buffers which let CPUs proceed after writes to non-cached regions. The von Neumann nature of memory is then visible when instructions are written as data by the CPU and software must ensure that the caches (data and instruction) and write buffer are synchronized before trying to execute those just-written instructions.

Modern uses of the Harvard architecture:

The principal advantage of the pure Harvard architecture—simultaneous access to more than one memory system—has been reduced by modified Harvard processors using modern CPU cache systems. Relatively pure Harvard architecture machines are used mostly in applications where trade-offs, like the cost and power savings from omitting caches, outweigh the programming penalties from featuring distinct code and data address spaces.

- Digital signal processors (DSPs) generally execute small, highly optimized audio or video processing algorithms. They avoid caches because their behavior must be extremely reproducible. The difficulties of coping with multiple address spaces are of secondary concern to speed of execution. Consequently, some DSPs feature multiple data memories in distinct address spaces to facilitate SIMD and VLIW processing. Texas Instruments TMS320 C55x processors, for one example, feature multiple parallel data buses (two write, three read) and one instruction bus.
- Microcontrollers are characterized by having small amounts of program (flash memory) and data (SRAM) memory, and take advantage of the Harvard architecture to speed processing by concurrent instruction and data access. The separate storage means the program and data memories may feature different bit widths, for example using 16-bit wide instructions and 8-bit wide data. They also mean that instruction prefetch can be performed in parallel with other activities. Examples include, PIC by Microchip Technology, Inc. and the AVR by Atmel Corp (now part of Microchip Technology).

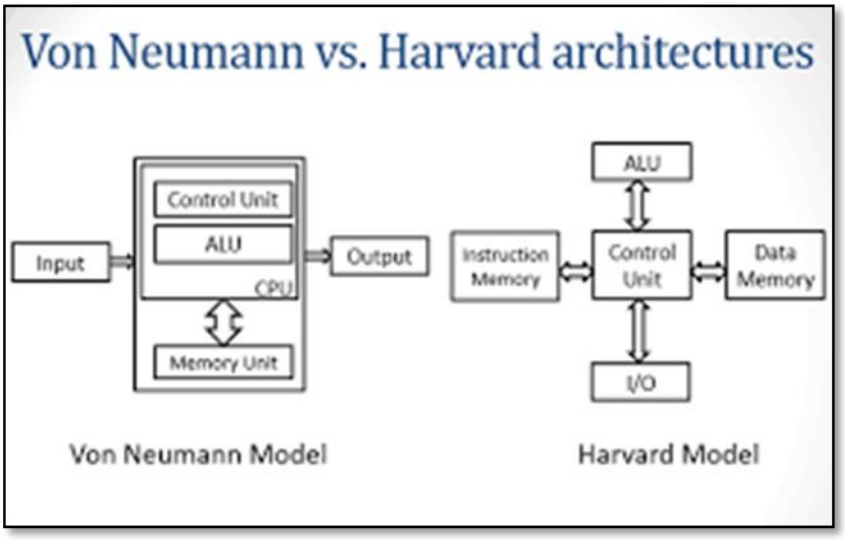


Fig: 1.10 Von Neumann VS Harvard Architecture

CISC VS RISC

LECTURE 3

CISC is a Complex Instruction Set Computer. It is a computer that can address a large number of instructions.

In the early 1980s, computer designers recommended that computers should use fewer instructions with simple constructs so that they can be executed much faster within the CPU without having to use memory. Such computers are classified as Reduced Instruction Set Computer or RISC.

Instruction set architecture (ISA)

Instruction set architecture (ISA) is the set of processor design techniques used to implement the instruction work flow on hardware. In more practical words, ISA tells you that how your processor going to process your program instructions.

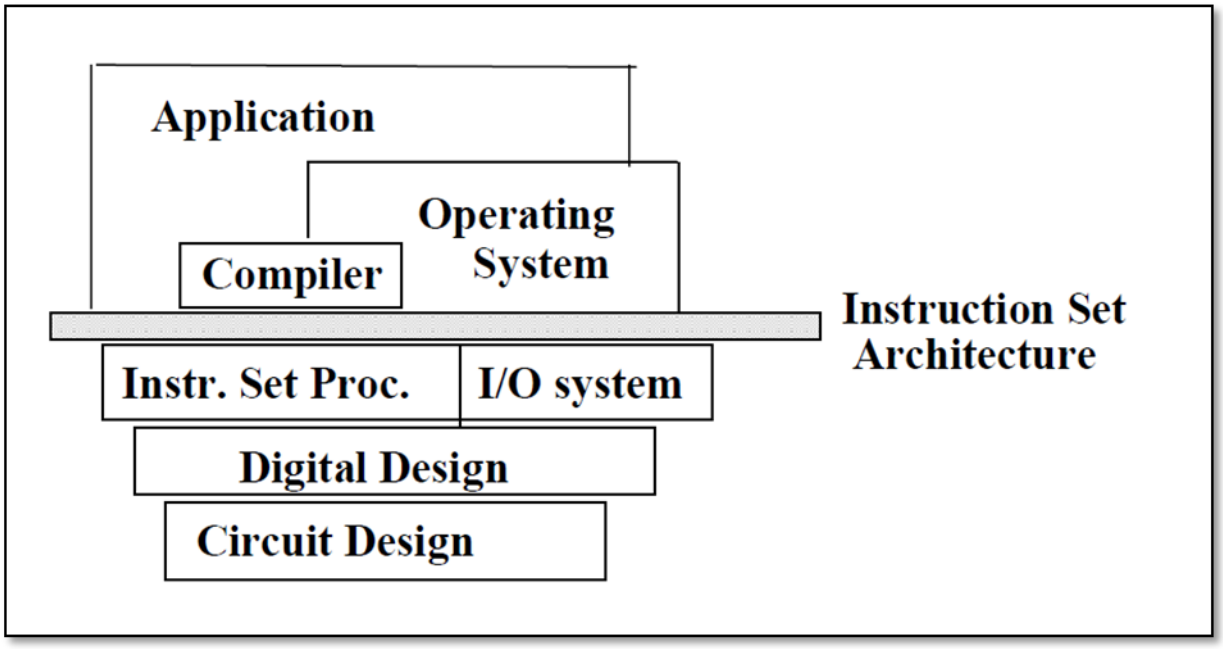


Fig: 1.11 Instruction set architecture (ISA)

There is no standard computer architecture accepting different types like CISC, RISC, etc.

Complex instruction set computer (CISC)

A complex instruction set computer (CISC /pronounce as 'sisk'/) is a computer where single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) or are capable of multi-step operations or addressing modes within single instructions, as its name suggest "COMPLEX INSTRUCTION SET".

Reduced instruction set computer (RISC)

A reduced instruction set computer (RISC /pronounce as 'risk'/) is a computer which only use simple instructions that can be divide into multiple instructions which perform low-level operation within single clock cycle, as its name suggest "REDUCED INSTRUCTION SET"

RISC & CISC architecture with example

Let we take an example of multiplying two numbers
A = A * B; <<<======this is C statement
The CISC Approach :- The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible. This is achieved by building processor hardware that is capable of understanding & executing a series of operations, this is where our CISC architecture introduced .

For this particular task, a CISC processor would come prepared with a specific instruction (we'll call it "MULT"). When executed, this instruction
Loads the two values into separate registers
Multiplies the operands in the execution unit
And finally third, stores the product in the appropriate register.
Thus, the entire task of multiplying two numbers can be completed with one instruction:
MULT A,B<<<======this is assembly statement
MULT is what is known as a "complex instruction." It operates directly on the computer's memory banks and does not require the programmer to explicitly call any loading or storing functions.

Advantage:-
Compiler has to do very little work to translate a high-level language statement into assembly
Length of the code is relatively short
Very little RAM is required to store instructions
The emphasis is put on building complex instructions directly into the hardware.
The RISC Approach :- RISC processors only use simple instructions that can be executed within one clock cycle. Thus, the "MULT" command described above could be divided into three separate commands:
"LOAD" which moves data from the memory bank to a register
"PROD" which finds the product of two operands located within the registers
"STORE" which moves data from a register to the memory banks.
In order to perform the exact series of steps described in the CISC approach, a programmer would need to code four lines of assembly:

```
LOAD R1, A      <<<======this is assembly statement
LOAD R2,B<<<======this is assembly statement
PROD A, B       <<<======this is assembly statement
STORE R3, A     <<<======this is assembly statement
```

At first, this may seem like a much less efficient way of completing the operation. Because there are more lines of code, more RAM is needed to store the assembly

level instructions. The compiler must also perform more work to convert a high-level language statement into code of this form.

Advantage:-

Each instruction requires only one clock cycle to execute, the entire program will execute in approximately the same amount of time as the multi-cycle "MULT" command.

These RISC "reduced instructions" require less transistors of hardware space than the complex instructions, leaving more room for general purpose registers. Because all of the instructions execute in a uniform amount of time (i.e. one clock)

Pipelining is possible.

LOAD/STORE mechanism:- Separating the "LOAD" and "STORE" instructions actually reduces the amount of work that the computer must perform. After a CISC-style "MULT" command is executed, the processor automatically erases the registers. If one of the operands needs to be used for another computation, the processor must re-load the data from the memory bank into a register. In RISC, the operand will remain in the register until another value is loaded in its place.

Example of RISC & CISC

Examples of CISC instruction set architectures are PDP-11, VAX, Motorola 68k, and your desktop PCs on intel's x86 architecture based too .

Examples of RISC families include DEC Alpha, AMD 29k, ARC, Atmel AVR, Blackfin, Intel i860 and i960, MIPS, Motorola 88000, PA-RISC, Power (including PowerPC), SuperH, SPARC and ARM too.

Which one is better?

We cannot differentiate RISC and CISC technology because both are suitable at its specific application. What counts are how fast a chip can execute the instructions it is given and how well it runs existing software. Today, both RISC and CISC manufacturers are doing everything to get an edge on the competition.

What's new?

You might thinking that RISC is now-a-days used in microcontroller application widely, so it's better for that particular application and CISC at desktop application. But reality is both are at threat position cause of a new technology called EPIC.

EPIC (Explicitly Parallel Instruction Computing) :-EPIC is a invented by Intel and is in a way, a combination of both CISC and RISC. This will in theory allow the processing of Windows-based as well as UNIX-based applications by the same CPU.

Intel is working on it under code-name Merced. Microsoft is already developing their Win64 standard for it. Like the name says, Merced will be a 64-bit chip.

If Intel's EPIC architecture is successful, it might be the biggest thread for RISC. All of the big CPU manufactures but Sun and Motorola are now selling x86-based products, and some are just waiting for Merced to come out (HP, SGI). Because of the x86 market it is not likely that CISC will die soon, but RISC may.

So the future might bring EPIC processors and more CISC processors, while the RISC processors are becoming extinct.

CISC	**RISC**
Larger set of instructions. Easy to program	Smaller set of Instructions. Difficult to program.
Simpler design of compiler, considering larger set of instructions.	Complex design of compiler.
Many addressing modes causing complex instruction formats.	Few addressing modes, fix instruction format.
Instruction length is variable.	Instruction length varies.
Higher clock cycles per second.	Low clock cycle per second.
Emphasis is on hardware.	Emphasis is on software.
Control unit implements large instruction set using micro-program unit.	Each instruction is to be executed by hardware.
Slower execution, as instructions are to be read from memory and decoded by the decoder unit.	Faster execution, as each instruction is to be executed by hardware.
Pipelining is not possible.	Pipelining of instructions is possible, considering single clock cycle.

AMDAHL'S LAW

LECTURE 4

The theory of doing computational work in parallel has some fundamental laws that place limits on the benefits one can derive from parallelizing a computation (or really, any kind of work). To understand these laws, we have to first define the objective. In general, the goal in large scale computation is to get as much work done as possible in the shortest possible time within our budget. We ``win'' when we can do a big job in less time or a bigger job in the same time and not go broke doing so. The ``power'' of a computational system might thus be usefully defined to be the amount of computational work that can be done divided by the time it takes to do it, and we generally wish to optimize power per unit cost, or cost-benefit.

Physics and economics conspire to limit the raw power of individual single processor systems available to do any particular piece of work even when the dollar budget is effectively unlimited. The cost-benefit scaling of increasingly powerful single processor systems is generally nonlinear and very poor - one that is twice as fast might cost four times as much, yielding only half the cost-benefit, per dollar, of a cheaper but slower system. One way to increase the power of a computational system (for problems of the appropriate sort) past the economically feasible single processor limit is to apply more than one computational engine to the problem.

This is the motivation for Beowulf design and construction; in many cases a Beowulf may provide access to computational power that is available in a alternative single or multiple processor designs, but only at a far greater cost.

In a perfect world, a computational job that is split up among N processors would complete in 1/N time, leading to an N -fold increase in power. However, any given piece of parallelized work to be done will contain parts of the work that *must* be done serially, one task after another, by a single processor. This part does *not* run any faster on a parallel collection of processors (and might even run more slowly). Only the part that can be parallelized runs as much as N-fold faster.

The ``speedup'' of a parallel program is defined to be the ratio of the rate at which work is done (the power) when a job is run on N processors to the rate at which it is done by just one. To simplify the discussion, we will now consider the ``computational work'' to be accomplished to be an arbitrary task (generally speaking, the particular problem of greatest interest to the reader). We can then define the speedup (increase in power as a function of N) in terms of the time required to complete this particular fixed piece of work on 1

Let T(N)be the time required to complete the task on N processors. The speedup S(N) is the ratio

to N processors.

In many cases the time T(1) has, as noted above, both a serial portion T_s and a parallelizable portion T_p. The serial time does not diminish when the parallel part is split up. If one is

"optimally" fortunate, the parallel time is decreased by a factor of 1/N. The speedup one can expect is thus

$$S(N) = \frac{T(1)}{T(N)} = \frac{T_s + T_p}{T_s + T_p/N}. \quad (2)$$

This elegant expression is known as *Amdahl's Law* and is usually expressed as an inequality. This is in almost all cases the *best* speedup one can achieve by doing work in parallel, so the real speed up $S(N)$ is less than or equal to this quantity.

Amdahl's Law immediately eliminates many, many tasks from consideration for parallelization. If the serial fraction of the code is not much smaller than the part that could be parallelized (if we rewrote it and were fortunate in being able to split it up among nodes to complete in less time than it otherwise would), we simply won't see much speedup no matter how many nodes or how fast our communications. Even so, Amdahl's law is still far too optimistic. It ignores the overhead incurred due to parallelizing the code. We must generalize it.

A fairer (and more detailed) description of parallel speedup includes at least two more times of interest:

$\mathbf{T_s}$ The original single-processor serial time.

$\mathbf{T_{is}}$ The (average) additional *serial* time spent doing things like interprocessor communications (IPCs), setup, and so forth in all parallelized tasks. This time can depend on N in a variety of ways, but the simplest assumption is that each system has to expend this much time, one after the other, so that the total additional serial time is for example $N * T_{is}$.

$\mathbf{T_p}$ The original single-processor parallelizeable time.

$\mathbf{T_{ip}}$ The (average) *additional* time spent by each processor doing just the setup and work that it does in parallel. This may well include idle time, which is often important enough to be accounted for separately.

It is worth remarking that generally, the most important element that contributes to T_{is} is the time required for communication between the parallel subtasks. This communication time is always there - even in the simplest parallel models where identical jobs are farmed out and run in parallel on a cluster of networked computers, the remote jobs must be begun and controlled with messages passed over the network. In more complex jobs, partial results developed on each CPU may have to be sent to all other CPUs in the beowulf for the calculation to proceed, which can be *very* costly in scaled time. As we'll see below, T_{is} in particular plays an extremely important role in determining the speedup scaling of a given

calculation. For this (excellent!) reason many beowulf designers and programmers are obsessed with communications hardware and algorithms.

It is common to combine T_{ip}, N and T_{is} into a single expression $T_o(N)$ (the ``overhead time") which includes any complicated $N$-scaling of the IPC, setup, idle, and other times associated with the overhead of running the calculation in parallel, as well as the scaling of these quantities with respect to the ``size" of the task being accomplished. The description above (which we retain as it illustrates the generic form of the relevant scalings) is still a *simplified* description of the times - real life parallel tasks can be much more complicated, although in many cases the description above is adequate.

Using these definitions and doing a bit of algebra, it is easy to show that an improved (but still simple) estimate for the parallel speedup resulting from splitting a particular job up between N nodes (assuming one processor per node) is:

$$S(N) = \frac{T_s + T_p}{T_s + N * T_{is} + T_p/N + T_{ip}}. \tag{3}$$

This expression will suffice to get at least a general feel for the scaling properties of a task that might be parallelized on a typical beowulf.

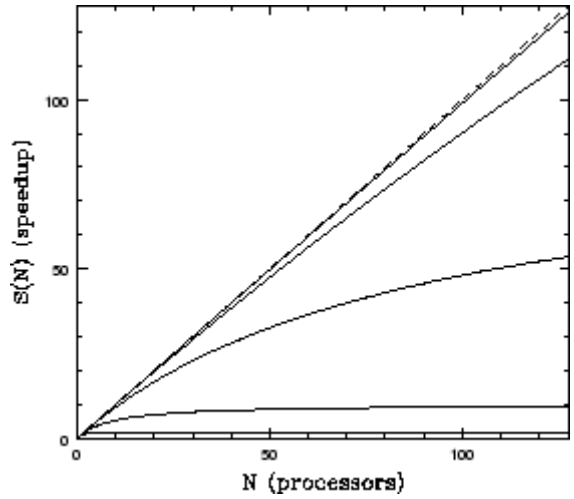


Figure 1.9: $T_{is} = 0$ and $T_p =$ 10, 100, 1000, 10000, 100000 (in increasing order).

It is useful to plot the dimensionless ``real-world speedup" (3) for various *relative* values of the times. In all the figures below, T_s = 10 (which sets our basic scale, if you like) and T_p = 10, 100, 1000, 10000, 100000 (to show the systematic effects of parallelizing more and more work compared to T_s).

MIPS

LECTURE 5

MIPS was one of the first RISC architectures. It was started about 20 years ago by John Hennessy.The architecture is similar to that of other recent CPU designs, including Sun's SPARC, IBMand Motorola's PowerPC, and ARM-based processors. MIPS designs are still used in many places today like Silicon Graphicsworkstations and servers,Various routers from Cisco,Game machines like the Nintendo 64and Sony Playstation 2.

Fig: 1.10 MIPS Computers

Components of the MIPS architecture:

- Major Components Of The Datapath
- Program Counter (PC)
- Instruction Register (IR)
- Register File
- Arithmetic And Logic Unit (ALU)
- Memory
- Control unit

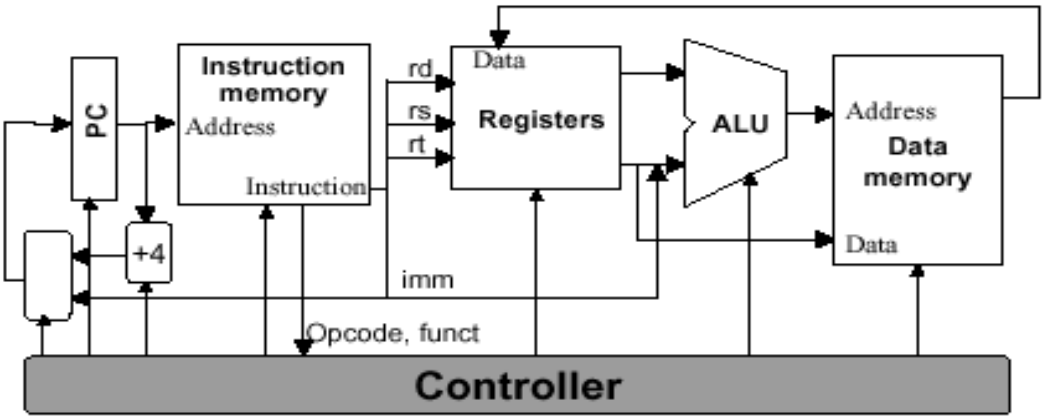


Fig: 1.12 Basic MIPS Architecture

In MIPS, programs are separated from data in memory

Text segment

- "instruction memory"
- part of memory that stores the program (machine code)
- **read only**

Data segment

- "data memory"
- part of memory that stores data manipulated by program
- **read/write**

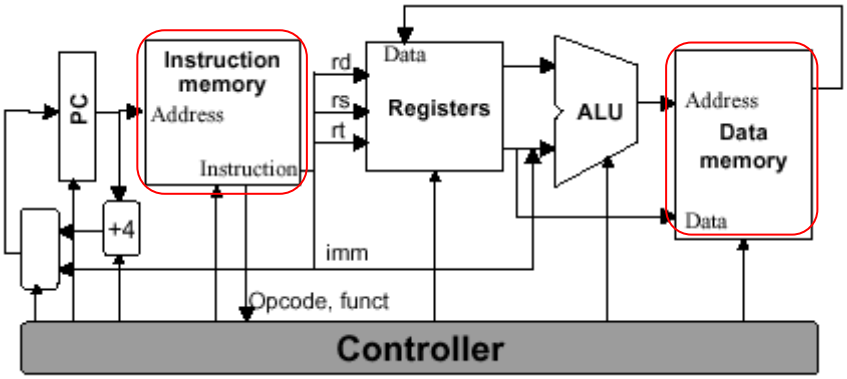


Fig: 1.13 Instruction Memories in MIPS Architecture

Distinction may or may not be reflected in the hardware:

- Von Neumann architecture – single, shared memory.
- Harvard architecture – physically separate memories

Program counter (PC)

Program: a sequence of machine instructions in the text segment

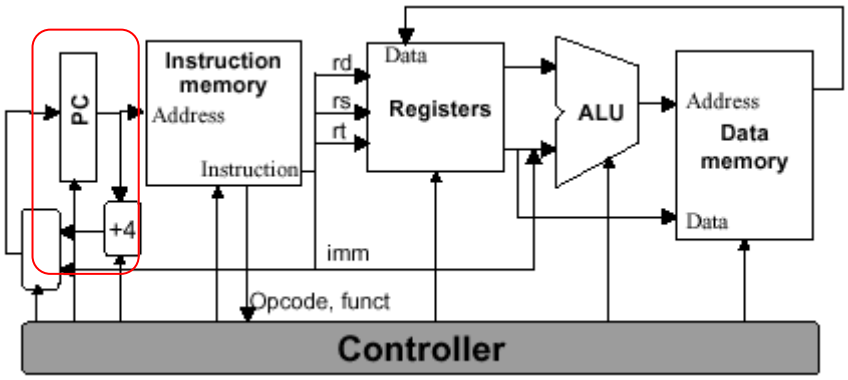


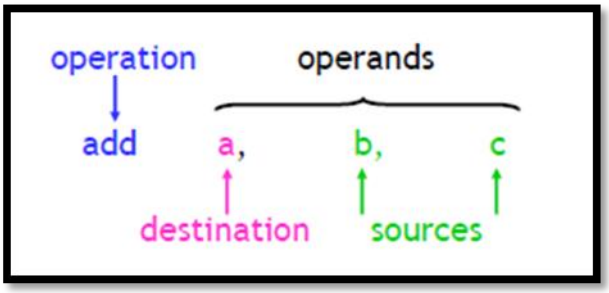
Fig: 1.14 PC in MIPS Architecture

Register that stores the **address** of the next instruction to fetch also called **the instruction pointer (IP)**

MIPS: Three Address:

MIPS uses three-address instructions for data manipulation. Each ALU instruction contains a destination and two sources.

For example, an addition instruction (a = b + c) has the form:



MIPS: Register-to-Register

MIPS is a register-to-register, or load/store, architecture.—The destination and sources must all be registers.—Special instructions, which we'll see later today, are needed to access main memory.

MIPS register file:

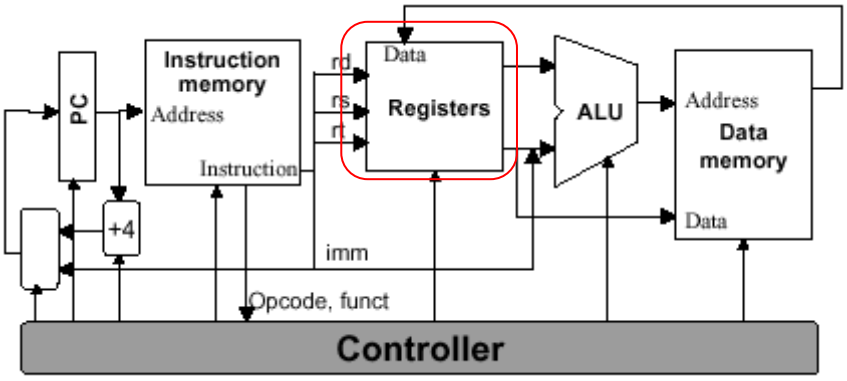


Fig: 1.15 MIPS Registers

MIPS processors have 32 registers, each of which holds a 32-bit value. Register addresses are 5 bits long. The data inputs and outputs are 32-bits wide .More registers might seem better, but there is a limit to the goodness. It's more expensive, because of the registers themselves as well as the decoders and muxes needed to select individual registers. Instruction lengths may be affected.

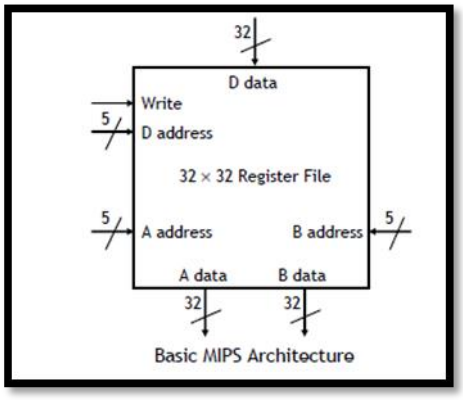


Fig: 1.16 MIPS Registers pin

MIPS register names begin with a $. There are two naming conventions:
By number:

```
$0   $1   $2   ...   $31
```

By (mostly) two-letter names, such as:

```
$a0-$a3   $s0-$s7   $t0-$t9   $sp   $ra
```

These registers are not general purpose registers. The basic integer arithmetic operations include the following:

add sub mul div

And here are a few logical operations:

and or xor

Remember that these all require three register operands; for example:

```
add  $t0, $t1, $t2      # $t0 = $t1 + $t2
mul  $s1, $s1, $a0      # $s1 = $s1 × $a0
```

MIPS Arithmetic and logic unit (ALU)

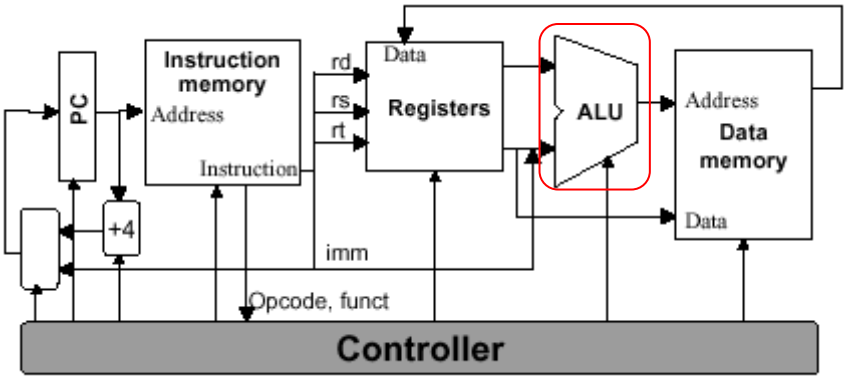


Fig: 1.17 MIPS ALU

Inputs:

- operands – 2 _ 32-bit
- operation – control signal

Outputs:

- result – 1 _ 64-bit (usually just use 32 bits of this)
- status – condition signals

Control Unit (CU)

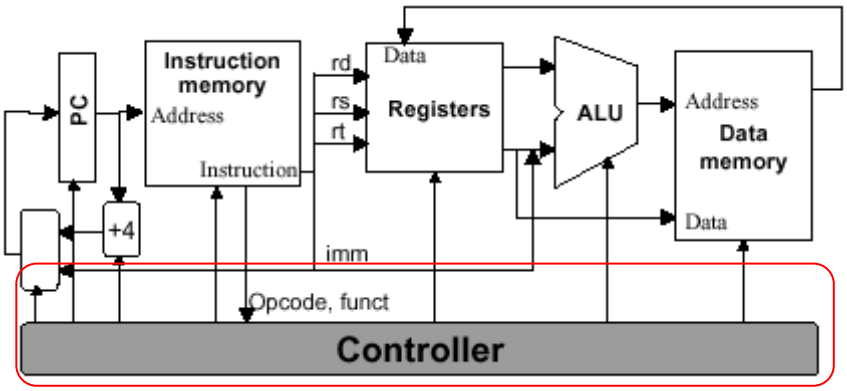


Fig: 1.18 MIPS CU

Controls components of datapath to implement FDX cycle

- **Inputs**: condition signals
- **Outputs**: control signals

MIPS Memory:

MIPS memory is byte-addressable, which means that each memory address references an 8-bit quantity. The MIPS architecture can support up to 32 address lines. This results in a 2^{32} x 8 RAM, which would be 4 GB of memory.Not all actual MIPS machines will have this much!

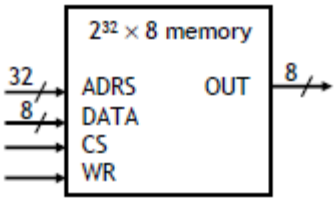


Fig: 1.19 MIPS Memory

Loading and storing bytes:

The MIPS instruction set includes dedicated load and store instructions for accessing memory. The main difference is that MIPS uses indexed addressing. The address operand specifies a signed constant and a register. These values are added to generate the effective address. The MIPS "load byte" instruction *lb* transfers one byte of data from main memory to a register.

```
lb $t0, 20($a0)     # $t0 = Memory[$a0 + 20]
```

The "store byte" instruction *sb* transfers the lowest byte of data from a register into main memory.

```
sb $t0, 20($a0)     # Memory[$a0 + 20] = $t0
```

Indexed addressing and arrays:

```
lb $t0, const($a0)
```

Indexed addressing is good for accessing contiguous locations of memory, like arrays or structures. The constant is the base address of the array or structure. The register indicates the element to access. For example, if $a0 contains 0, then

```
lb $t0, 2000($a0)
```

reads the first byte of an array starting at address 2000.

If $a0 contains 8, then the same instruction would access the ninth byte of the array, at address 2008. This is why array indices in C and Java start at 0 and not 1.

Loading and storing words:

You can also load or store 32-bit quantities a complete word instead of just a byte with the *lw* and *sw* instructions.

```
lw $t0, 20($a0)          # $t0 = Memory[$a0 + 20]
sw $t0, 20($a0)          # Memory[$a0 + 20] = $t0
```

Most programming languages support several 32-bit data types. Integers, Single-precision floating-point numbers, Memory addresses, or pointers, Unless otherwise stated, we'll assume words are the basic unit of data.

Memory alignment:

Keep in mind that memory is byte-addressable, so a 32-bit word actually occupies four contiguous locations of main memory.

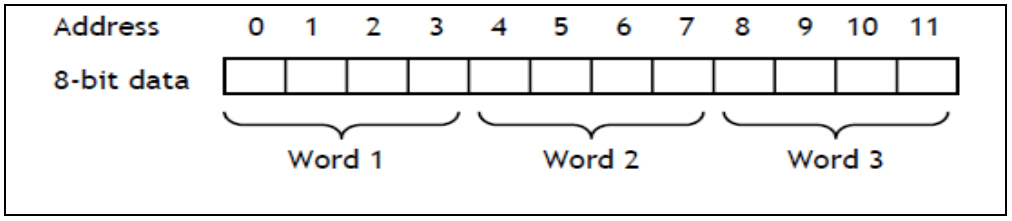


Fig: 1.20. Memory Alignment

The MIPS architecture requires words to be aligned in memory; 32-bit words must start at an address that is divisible by 4.

- 0, 4, 8 and 12 are valid word addresses.
- 1, 2, 3, 5, 6, 7, 9, 10 and 11 are not valid word addresses.
- Unaligned memory accesses result in a bus error, which you may have unfortunately seen before. This restriction has relatively little effect on high-level languages and compilers, but it makes things easier and faster for the processor.

MIPS data path with control signals

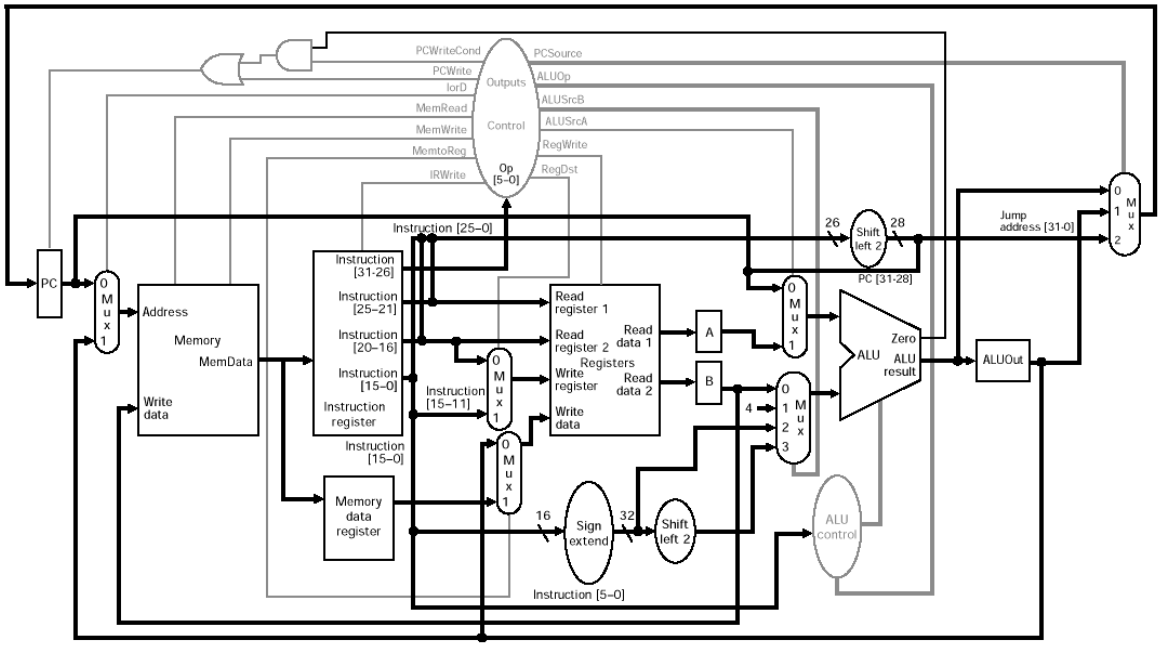


Fig: 1.21.MIPS data path with control signals

Benchmark Specification:

n computing, a benchmark is the act of running a computer program, a set of programs, or other operations, in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it.The term 'benchmark' is also mostly utilized for the purposes of elaborately designed benchmarking programs themselves.

Benchmarking is usually associated with assessing performance characteristics of computer hardware, for example, the floating point operation performance of a CPU, but there are circumstances when the technique is also applicable to software. Software benchmarks are, for example, run against compilers or database management systems.

Benchmarks provide a method of comparing the performance of various subsystems across different chip/system architectures.

Test suites are a type of system intended to assess the correctness of software.There are two common benchmarks program are

1. SPECint
2. SPECfp

SPECint

SPECint is a computer benchmark specification for CPU integer processing power. It is maintained by the Standard Performance Evaluation Corporation (SPEC). SPECint is the integer performance testing component of the SPEC test suite. The first SPEC test suite, CPU92, was announced in 1992. It was followed by CPU95, CPU2000, and CPU2006. The latest standard of SPECint is CINT2006 (aka SPECint2006).

CPU2006 is a set of benchmarks designed to test the CPU performance of a modern server computer system. It is split into two components, the first being CINT2006, the other being CFP2006 (SPECfp), for floating point testing.

SPEC defines a base runtime for each of the 12 benchmark programs. For SPECint2006, that number ranges from 1000 to 3000 seconds. The timed test is run on the system, and the time of the test system is compared to the reference time, and a ratio is computed. That ratio becomes the SPECint score for that test. (This differs from the rating in SPECINT2000, which multiplies the ratio by 100.)

As an example for SPECint2006, consider a processor which can run 400.perlbench in 2000 seconds. The time it takes the reference machine to run the benchmark is 9770 seconds.Thus the ratio is 4.885. Each ratio is computed, and then the geometric mean of those ratios is computed to produce an overall value.

Benchmark	Language	Category	Description
400.perlbench	C	Perl Programming Language	Derived from Perl V5.8.7. The workload includes SpamAssassin, MHonArc (an email indexer), and specdiff (SPEC's tool that checks benchmark outputs).
401.bzip2	C	Compression	Julian Seward's bzip2 version 1.0.3, modified to do most work in memory, rather than doing I/O.
403.gcc	C	C Compiler	Based on gcc Version 3.2, generates code for Opteron.
429.mcf	C	Combinatorial Optimization	Vehicle scheduling. Uses a network simplex algorithm (which is also used in commercial products) to schedule public transport.
445.gobmk	C	Artificial Intelligence: go playing	Plays the game of Go, a simply described but deeply complex game.
456.hmmer	C	Search Gene Sequence	Protein sequence analysis using profile hidden Markov models (profile HMMs)
458.sjeng	C	Artificial Intelligence: chess playing	A highly-ranked chess program that also plays several chess variants.
462.libquantum	C	Physics: Quantum Computing	Simulates a quantum computer, running Shor's polynomial-time factorization algorithm.
464.h264ref	C	Video Compression	A reference implementation of H.264/AVC, encodes a videostream using 2 parameter sets. The H.264/AVC standard is expected to replace MPEG2
471.omnetpp	C++	Discrete Event Simulation	Uses the OMNet++ discrete event simulator to model a large Ethernet campus network.
473.astar	C++	Path-finding Algorithms	Pathfinding library for 2D maps, including the well known A* algorithm.

SPECfp

SPECfp is a computer benchmark designed to test the floating point performance of a computer. It is managed by the Standard Performance Evaluation Corporation. SPECfp is the floating point performance testing component of the SPEC CPU testing suit. The first standard SPECfp was released in 1989[1] as SPECfp89. Later it was replaced by SPECfp92, then SPECfp95, then SPECfp2000, and finally SPECfp2006.

SPEC CPU2006 is a suite of benchmark applications designed to test the CPU performance. The suite is composed of two sets of tests. The first being CINT (aka SPECint) which is for evaluating the CPU performance in integer operations. The second set is CFP (aka SPECfp) which is for evaluating the CPU floating point operations performance.

The benchmark applications are programs that perform a strict set of operation that simulate real time situations, such as physical simulations, 3D graphics, and image processing. These applications are written in different programming languages, C, C++ and Fortran. Many SPECfp benchmark applications are derived from applications that are freely available to the public and each application is assigned a weight based on its importance.

Benchmark	Language	Category	Description
410.bwaves	Fortran	Fluid Dynamics	Simulates 3D transonic transient laminar viscous flow.
416.gamess	Fortran	Quantum Chemistry	Self-consistent field computations are performed using Restricted open-shell Hartree–Fock, the Restricted Hartree Fock method, and Multi-Configuration Self-Consistent Field
433.milc	C	Physics: Quantum Chromodynamics	A program that generates gauge field for lattice gauge theory programs with dynamical quarks.
434.zeusmp	Fortran	Physics/ CFD	A computational fluid dynamics program developed at NCSA(University of Illinois at Urbana-Champaign) for the simulation of astrophysical phenomena.
435.gromacs	C/ Fortran	Biochemistry/ Molecular Dynamics	Computes Newtonian equations of motion for hundreds to millions of particles. It simulates protein Lysozyme in a solution.
436.cactusADM	C/ Fortran	Physics/ General Relativity	Simulates the Einstein evolution equations using a staggered-leapfrog numerical method
437.leslie3d	Fortran	Fluid Dynamics	Computational Fluid Dynamics (CFD) using Large-Eddy Simulations with Linear-Eddy Model in 3D. Uses the MacCormack Predictor-Corrector time integration scheme.
444.namd	C++	Biology/ Molecular Dynamics	Simulates large biomolecular systems. The simulation has 92,224 atoms of apolipoprotein A - I.
447.dealII	C++	Finite Element Analysis	Computes adaptive finite elements and error estimation. The simulation solves a Helmholtz-type equation with non-constant coefficients.
450.soplex	C++	Linear Programming, Optimization	Solves a linear program using a simplex algorithm and sparse linear algebra. Test simulation include railroad planning and military airlift models.
453.povray	C++	Image Ray-tracing	The computation is a 1280x1024 anti-aliased image of a landscape with some abstract objects with textures using a Perlin noise function.
454.Calculix	C/ Fortran	Structural Mechanics	Computes finite element code for linear and nonlinear 3D structural applications. Uses the SPOOLES solver library.
459.GemsFDTD	Fortran	Computational Electromagnetics	Simulates the Maxwell equations in 3D using the finite-difference time-domain (FDTD) method.
465.tonto	Fortran	Quantum Chemistry	The simulation places a constraint on a molecular Hartree–Fock wave function calculation to better match experimental X-ray diffraction data.

Reference:

1. https://www.elsevier.com/books/computer-architecture/hennessy/978-0-12-383872-8
2. http://www.gatestudymaterial.com/study-material/computer%20organization%20and%20architecture/Advanced%20Computer%20Architecture%20by%20Kai%20Hwang.pdf
3. http://www.gatestudymaterial.com/study-material/computer%20organization%20and%20architecture/Advanced%20Computer%20Architecture%20by%20Kai%20Hwang.pdf

MCQ Questions:

1. The decoded instruction is stored in _____ .
a) IR
b) PC
c) Registers
d) MDR
2. The instruction -> Add LOCA,R0 does,
a) Adds the value of LOCA to R0 and stores in the temp register
b) Adds the value of R0 to the address of LOCA
c) Adds the values of both LOCA and R0 and stores it in R0
d) Adds the value of LOCA with a value in accumulator and stores it in R0
3. Which registers can interact with the secondary storage?
a) MAR
b) PC
c) IR
d) R0
4. During the execution of a program which gets initialized first?
a) MDR
b) IR
c) PC
d) MAR
5. Which of the register/s of the processor is/are connected to Memory Bus?
a) PC
b) MAR
c) IR
d) Both a and b
6. ISP stands for,
a) Instruction Set Processor
b) Information Standard Processing
c) Interchange Standard Protocol
d) Interrupt Service Procedure
7. The registers, ALU and the interconnection between them are collectively called as _____ .
a) Process route
b) Information trail
c) Information path
d) Data path

8. _____ is used to store data in registers.
a) D flip flop
b) JK flip flop
c) RS flip flop
d) None of these
9. The CISC stands for
a) Computer Instruction Set Compliment
b) Complete Instruction Set Compliment
c) Computer Indexed Set Components
d) Complex Instruction set computer
10. The Sun micro systems processors usually follow _____ architecture.
a) CISC
b) ISA
c) ULTRA SPARC
d) RISC
11. Out of the following which is not a CISC machine.
a) IBM 370/168
b) VAX 11/780
c) Intel 80486
d) Motorola A567
12. Pipe-lining is a unique feature of _____.
a) RISC
b) CISC
c) ISA
d) IANA
13. Which of the architecture is power efficient?
a) CISC
b) RISC
c) ISA
d) IANA
14. The control unit controls other units by generating ____ .
a) Control signals
b) Timing signals
c) Transfer signals
d) Command Signals
15. _____ bus structure is usually used to connect I/O devices .
a) Single bus
b) Multiple bus
c) Star bus
d) Rambus
16. The Input devices can send information to the processor,
a) When the SIN status flag is set
b) When the data arrives regardless of the SIN flag
c) Neither of the cases
d) Either of the cases

17. The collection of the above mentioned entities where data is stored is called as _____ .
a) Block
B) Set
c) Word
d) Byte
18. A complete microcomputer system consist of ……..
A) microprocessor
B) memory
C) peripheral equipment
D) all of the above
19. PC Program Counter is also called …….
A) instruction pointer
B) memory pointer
C) data counter
D) file pointer
20. In a single byte how many bits will be there?
A) 8
B) 16
C) 4
D) 32

Assignments:

1. Explain Amdahl's Law.
2. Differentiate between : RISC and CISC
3. What is ISA?
4. Differentiate between: Von- Neumann architecture Vs Harvard Architecture
5. Write down the full form: CISC.RISC.ISA,MIPS,PECTint.SPECfp
6. What is data path?
7. What do you mean by Instruction cycle?
8. What is interrupt? Why it is happened?
9. Write Short notes on: MIPS architecture/Component MIPS
10. What is SPECint and SPECfp Program?

Web/Video links:

1. https://www.youtube.com/watch?v=L2oWDL3Msaw&list=PLbMVogVj5nJRtXgdjQkYfYOHfsc-7Ar7Q
2. https://www.youtube.com/watch?v=9JJQ2MI-Y4A&list=PLbMVogVj5nJRtXgdjQkYfYOHfsc-7Ar7Q&index=3
3. https://www.youtube.com/watch?v=odlUDjraYa4&list=PLbMVogVj5nJRtXgdjQkYfYOHfsc-7Ar7Q&index=6
4. https://www.youtube.com/watch?v=TH9nd-KdVHs&list=PL73A9FB893089582E&index=1
5. https://www.youtube.com/watch?v=oQvsz3q0bYU
6. https://www.youtube.com/watch?v=RTC1ZCGBOu4
7. https://www.youtube.com/watch?v=YGSAWqQy9bI
8. https://www.youtube.com/watch?v=ibYYqvp9FmU
9. https://www.youtube.com/watch?v=56XG8Aw0fPk
10. https://www.youtube.com/watch?v=4DHHKXeDS-A
11. http://www.sanfoundry.com/computer-organization-mcqs-memory-locations-addresses/

MODULE 2

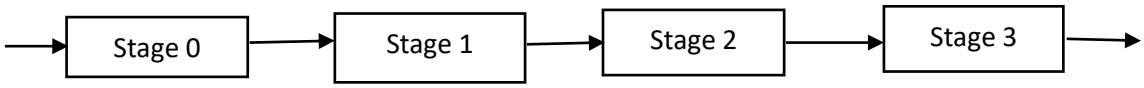
LECTURE 1

Pipelining

Pipelining is an implementation technique where multiple instructions are overlapped in execution. The computer pipeline is divided in stages. Each stage completes a part of an instruction in parallel. The stages are connected one to the next to form a pipe - instructions enter at one end, progress through the stages, and exit at the other end.

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.

Pipelining for instruction execution is similar to construction of factor assembly line for product manufacturing. The basic idea is to decompose the instruction execution process into a collection of smaller functions that can be independently performed by discrete subsystems in the processor implementation. An illustration of this decomposition into 4 parts is:



For pipelining, we will organized these discrete subsystems (which are called pipeline stages) implementing the instruction interpretation process into concurrently executing systems each operating on distinct instructions in the instruction stream (much like a factory assembly line).

Typical Non Pipelined Execution

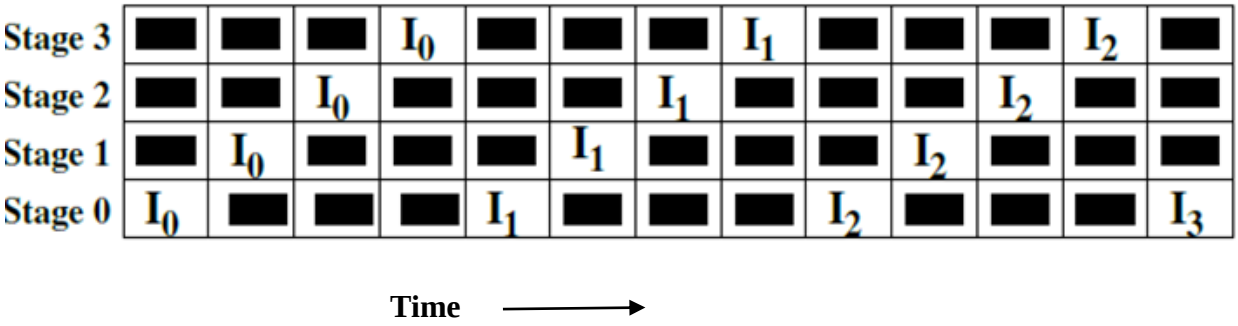


Fig: 2.1 Idealized Pipeline Executions

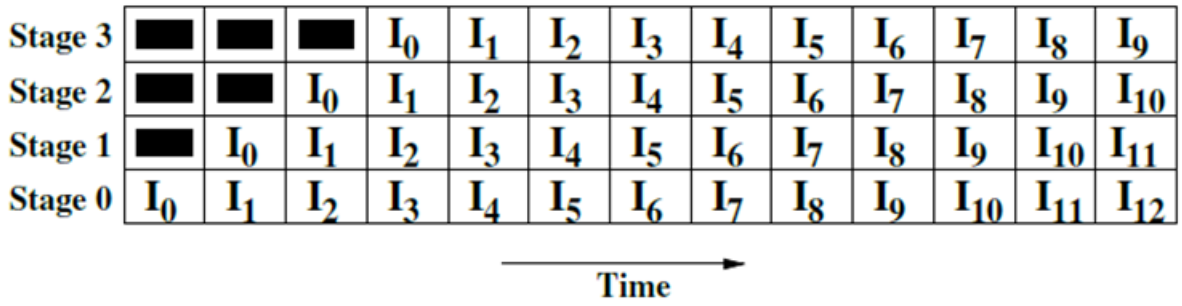


Fig: 2.2 Actual Pipeline Executions

Time to execute n instructions: (3+n)t.

Steady state : $\frac{Clock\ cycles\ for\ unpipelined\ execution}{Pipeline\ depth}$

Speedup , Efficiency and Throughput

Ideally, a linear pipeline ok k stages can process n tasks in $k + (n+1)$ clock cycles , where k cycles are needed to complete the execution of the very first task and the remaining n-1 tasks require n-1 cycles. Thus the total time required is:

$$T_k = [k + (n-1)]\tau$$

where τ is the clock period. Consider an equivalent function non-pipelined processor which has a *flow-through delay* of $k\tau$. The amount of time it takes to execute n tasks on this non pipelined processor is $T_1 = nk\tau$.

Speedup Factor

The speedup factor of a k-stage pipeline over an equivalent non-pipelined processor is defined as:

$$S_k = \frac{T_1}{T_K} = \frac{nk\tau}{k\tau+(n-1)\tau} = \frac{nk}{k+(n-1)}$$

Efficiency and Throughput

The efficiency E_k of a linear k-stage pipeline is defined as

$$E_k = \frac{Sk}{k} = \frac{n}{k+(n-1)}$$

Obviously, the efficiency approaches 1 when $n \to \infty$, and a lower bound on E_k is 1/k when n = 1. The pipeline throughput H_k is defined as the number of tasks (operations) per unit time :

$$H_k = \frac{n}{[k+(n-1)]\tau} = \frac{nf}{k+(n-1)}$$

The maximum throughput f occurs when Ek →1 and n→ ∞. This coincides with the speedup definition given in chapter 3. Note that $H_k = E_k$. $f = E_k/\tau = S_k/k\tau$.

Consider the numerical example,

let the time it takes to process a sub-operation in each segment be equal to t_p = 20 ns. Assume that the pipeline has k = 4 segments and execute n = 100 tasks in sequence. The pipeline system will take $(k + n - 1)t_p$= (4+ 99)x20= 2060 ns to complete.

Assuming that $t_{n=} kt_p$ = 4 x 20 = 80 ns ,

a non pipeline system requires nkt_p = 100 x 80 = 8000 ns to complete the 100 tasks. The speedup ratio is equal to 8000/2060 = 3.88. As the number of tasks increases , the speedup will approach 4 , which is equal to the number of segments in the pipeline. If we assume that t_n = 60 ns, the speedup becomes 60/20 = 3.

LECTURE 2

Linear vs Non-Linear,Static Vs Dynamic Vs Unifunction Vs Multifunction Pileline

A linear pipelining is a series of processing stages and memory access.

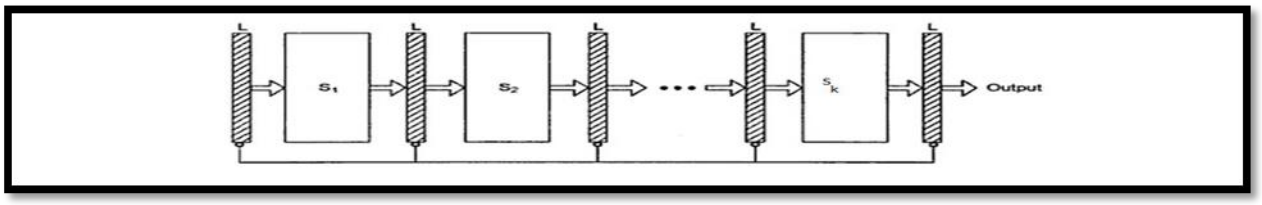


Fig: 2.3 Linear Pipeline

A non linear pipelining can be configured to perform various functions at different times. In a dynamic pipeline there is also feed forward or feedback connection. Non-linear pipeline also allows very long instruction words.

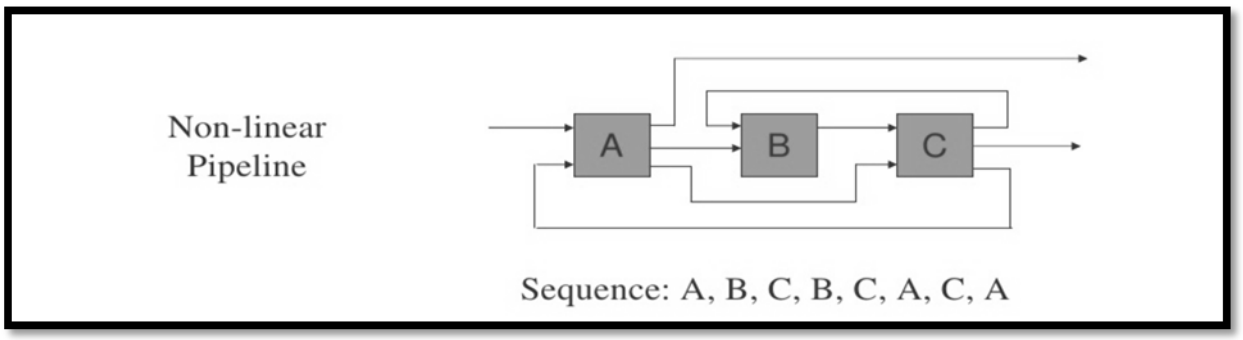


Fig: 2.4 Linear Pipeline

Linear Pipeline	Non-Linear Pipeline
Linear pipeline are static pipeline because they are used to perform fixed functions.	Non-Linear pipeline are dynamic pipeline because they can be reconfigured to perform variable functions at different times.
Linear pipeline allows only streamline connections.	Non-Linear pipeline allows feed-forward and feedback connections in addition to the streamline connection.
It is relatively easy to partition a given function into a sequence of linearly ordered sub functions.	Function partitioning is relatively difficult because the pipeline stages are interconnected with loops in addition to streamline connections.
The Output of the pipeline is produced from the last stage.	The Output of the pipeline is not necessarily produced from the last stage.

The reservation table is trivial in the sense that data flows in linear streamline.	The reservation table is non-trivial in the sense that there is no linear streamline for data flows.
Static pipelining is specified by single Reservation table.	Dynamic pipelining is specified by more than one Reservation table.
All initiations to a static pipeline use the same reservation table.	A dynamic pipeline may allow different initiations to follow a mix of reservation tables.

There are two types of pipelines: Static and Dynamic. A static pipeline can perform only one function at a time whereas a dynamic pipeline can perform more than one function at a time.

Static pipelining - it is composition of stages one after another means that the output of one stage is become input to the next stage we also called it linear pipelining. it is further divided in two types synchronous and asynchronous.

Dynamic pipelining- in it stages are connected in a liner fashion but this kind of pipelining used feed forward and feed backward connections as a input to the stages. It performs variable function but static perform fixed functions. In dynamic pipelining we can take intermediate outputs.

Static	Dynamic
It may assume only one functional configuration at a time	It permits several functional configurations to exist simultaneously
It can be either unifunctional or multifunctional	A dynamic pipeline must be multi-functional
Static pipelines are preferred when instructions of same type are to be executed continuously	The dynamic configuration requires more elaborate control and sequencing mechanisms than static pipelining

A pipeline unit with a fixed and dedicated function is called **unifunctional.**

A **multifunction pipe** may perform different functions, either at different times or at the same time.

Unifunctional Pipelines	**Multifunctional Pipelines**
A pipeline unit with fixed and dedicated function is called unifunctional.	A multifunction pipe may perform different functions either at different times or same time, by interconnecting different subset of stages in pipeline.
It has 12 unifunctional pipelines described in four groups: – Address Functional Units: • Address Add Unit • Address Multiply Unit	It has – one instruction processing unit – four memory buffer units and – four arithmetic units.
Example: CRAY1 (Supercomputer - 1976)	Example 4X-TI-ASC (Supercomputer - 1973)

LECTURE 3

Instruction Pipeline

A stream of instructions can be executed by a pipeline in an overlapped manner.

The Instruction Cycle is given below:

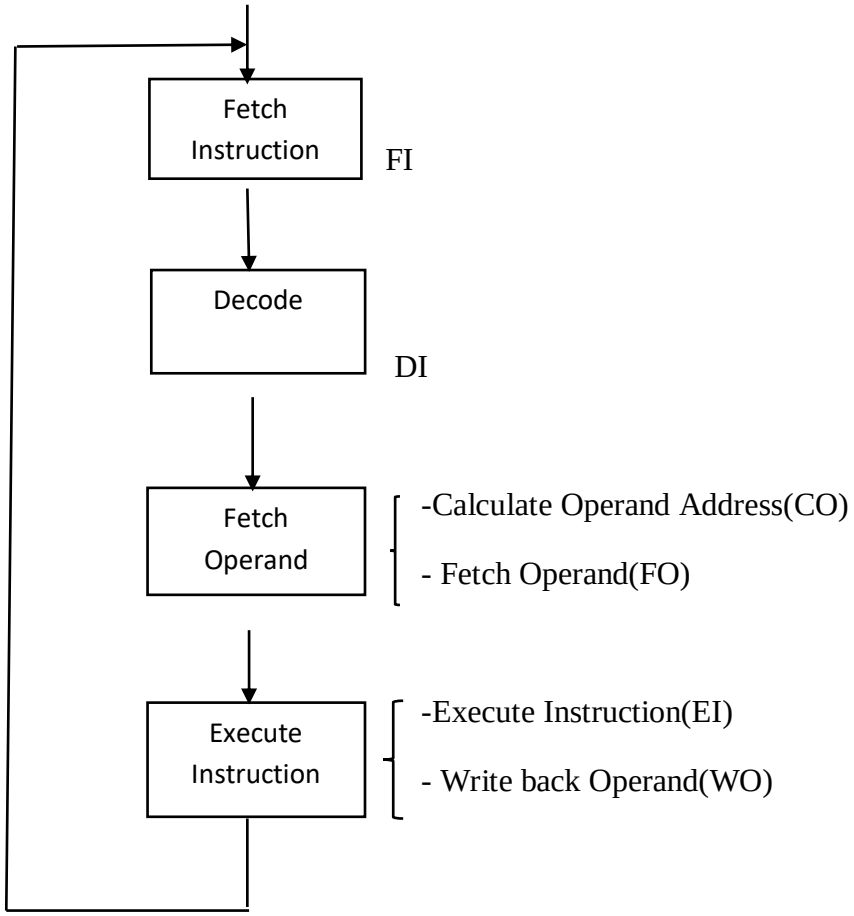


Fig: 2.5 Instruction Cycle

Instruction execution is extremely complex and involves several operations which are executed successively. This implies a large amount of hardware, but only one part of this hardware works at a given moment.

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution. This is solved without additional hardware but only by letting different parts of the hardware work for different instructions at the same time.

The pipeline organization of a CPU is similar to an assembly line: the work to be done in an instruction is broken into smaller steps (pieces), each of which takes a fraction of the time needed to complete the entire instruction. Each of these steps is a pipe stage (or a pipe segment).

The time required to execute a stage and move to the next is called a *machine cycle* (this is one or several clock cycles). The execution of one instruction takes several machine cycles as it passes through the pipeline.

The Four Segment Pipelining:

Four segment pipeline:
FI: fetch instruction
DA: decode instruction
FO: fetch operand
EX: execute instruction

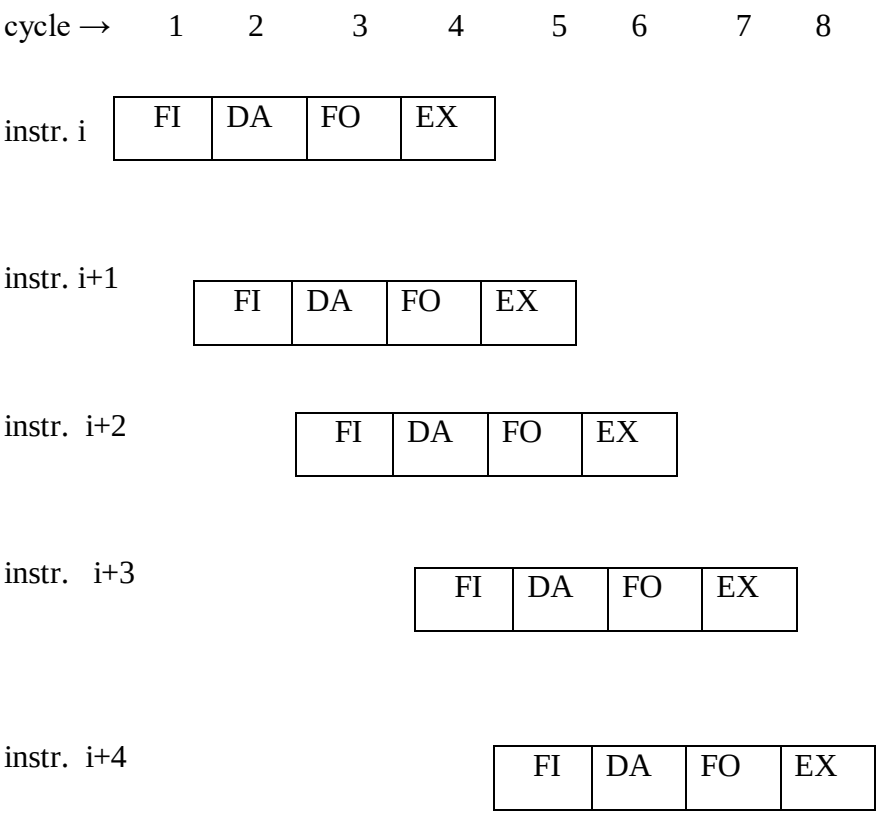


Fig: **2.6Pipelining by four Segments**

Acceleration by Pipelining Six Segments:

Six stage pipeline:
FI: fetch instruction FO: fetch operand
DI: decode instruction EI: execute instruction
CO: calculate operand address WO:write operand

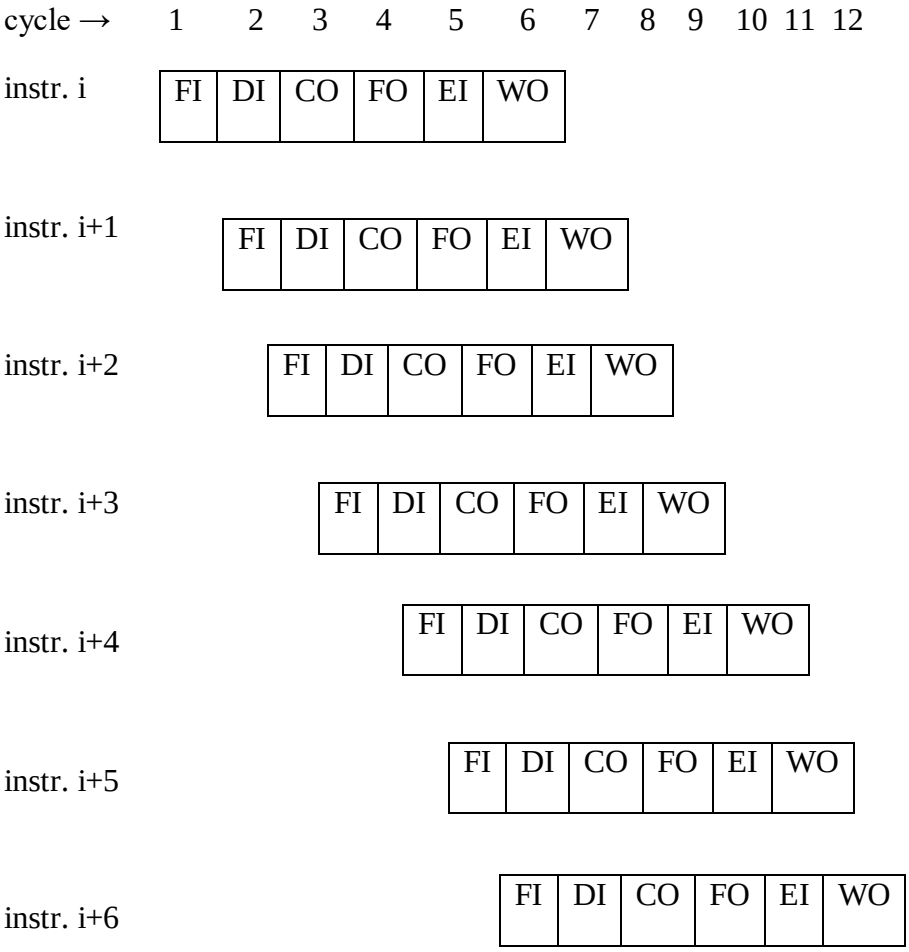


Fig: 2.7 **Pipelining by Six Segments**

Execution time for the 7 instructions, with pipelining:

$$(T_{ex}/6)\times 12 = 2\times T_{ex}$$

• Acceleration: $7\times T_{ex} / 2\times T_{ex} = 7/2$

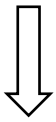
After a certain time (N-1 cycles) all the N stages of the pipeline are working: the pipeline is filled. Now, *theoretically*, the pipeline works providing maximal parallelism (N instructions are active simultaneously).
• τ: duration of one machine cycle
• n: number of instructions to execute

• k: number of pipeline stages
• $T_{k,n}$: total time to execute n instructions on a pipeline with k stages
• $S_{k,n}$: (theoretical) speedup produced by a pipeline with k stages when executing n instructions

$$T_{k,n} = [k + (n-1)] \times \tau$$

- The first instruction takes $k \times \tau$ to finish
- The following $n - 1$ instructions produce one result per cycle.

On a non-pipelined processor each instruction takes $k \times \tau$, and n instructions: $T_n = n \times k \times \tau$



$$S_{k,n} = \frac{Tn}{Tk,n} = \frac{nXkX\tau}{[k + (n-1)] \times \tau} = \frac{nXk}{k+(n-1)}$$

For large number of instructions ($n \to \infty$) the speedup approaches k (number of stages).
• Apparently a greater number of stages always provides better performance. However:

- a greater number of stages increases the overhead in moving information between stages and synchronization between stages.
- with the number of stages the complexity of the CPU grows.
- it is difficult to keep a large pipeline at maximum rate because of *pipeline hazards*.

ARITHMETIC PIPELINE

LECTURE 4

Pipeline arithmetic units are usually found in very high speed computers. They are used to implement floating point operations. We will now discuss the pipeline unit for the floating point addition and subtraction.
The inputs to floating point adder pipeline are two normalized floating point numbers.

$$X = A \times 2^a$$
$$Y = B \times 2^b$$

A and B are mantissas and a and b are the exponents.
The floating point addition and subtraction can be performed in four segments.

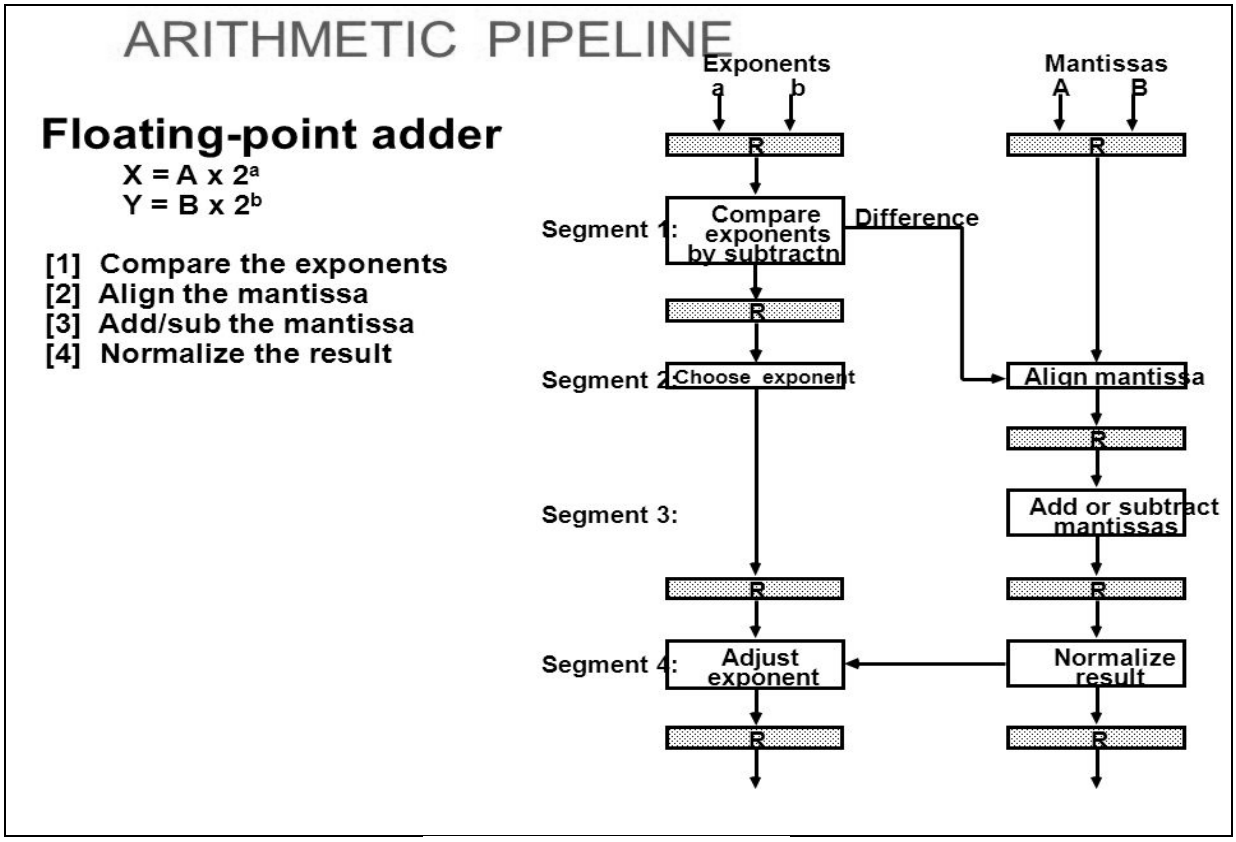


Fig: 2.8 Arithmetic Pipeline

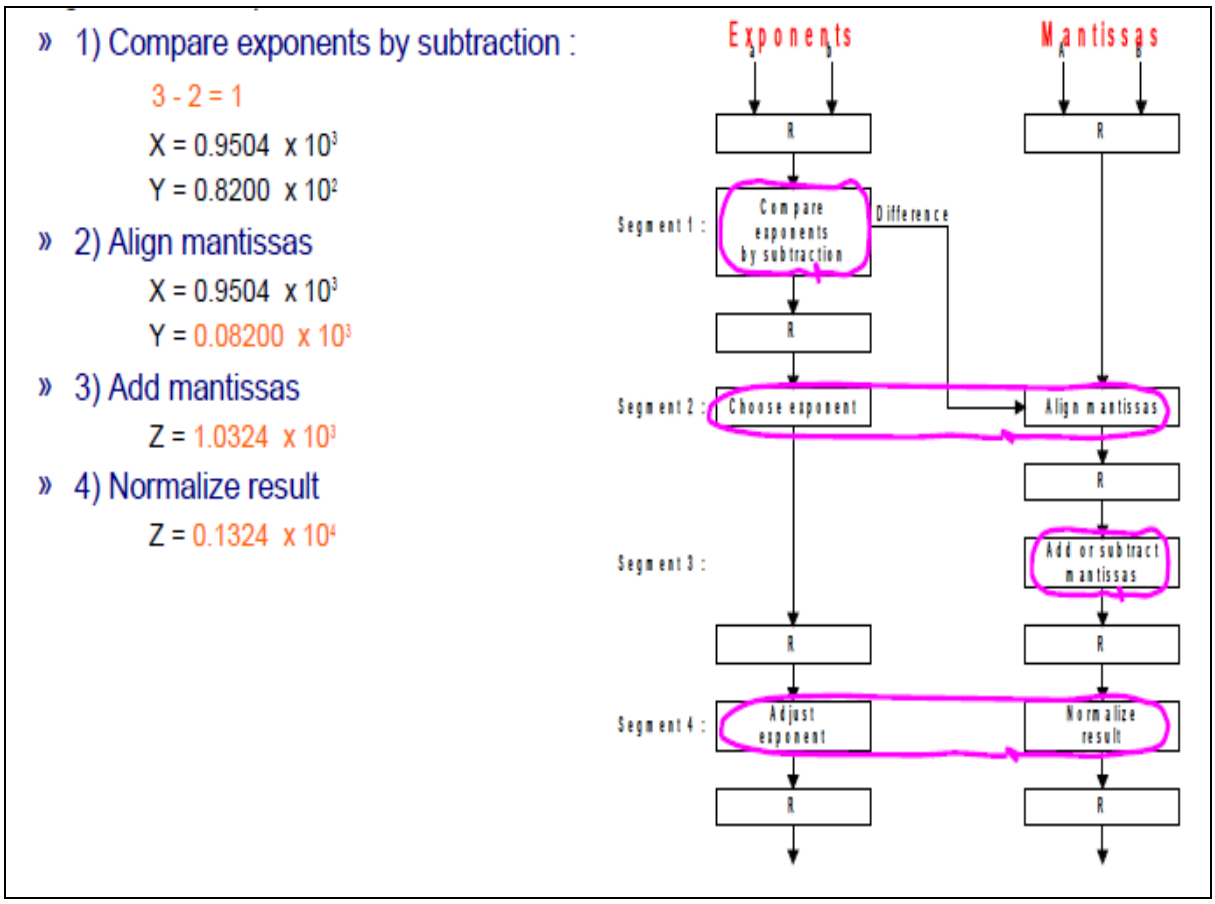


Fig: 2.9 Operation on Pipeline segments

PIPELINE HAZARDS

LECTURE 5

There are situations, called **hazards**, that prevent the next instruction in the instruction stream from being executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining.

There are three classes of hazards:

1.Structural Hazards. They arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution. **2.Data Hazards.** They arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline. **3.Control Hazards.**They arise from the pipelining of branches and other instructions that change the PC.

Hazards in pipelines can make it necessary to stall the pipeline. The processor can stall on different events:

A ***cache miss***. A cache miss stalls all the instructions on pipeline both before and after the instruction causing the miss.
A ***hazard in pipeline.*** Eliminating a hazard often requires that some instructions in the pipeline to be allowed to proceed while others are delayed. When the instruction is stalled, all the instructions issued *later* than the stalled instruction are also stalled. Instructions issued *earlier* than the stalled instruction must continue, since otherwise the hazard will never clear.

HAZARDS

Data hazards

Data hazards occur when instructions that exhibit data dependence modify data in different stages of a pipeline. Ignoring potential data hazards can result in race conditions (also termed race hazards). There are three situations in which a data hazard can occur:

1. read after write (RAW), a *true dependency*
2. write after read (WAR), an *anti-dependency*
3. write after write (WAW), an *output dependency*

Consider two instructions `i1` and `i2`, with `i1` occurring before `i2` in program order.

Read after write (RAW)

(`i2` tries to read a source before `i1` writes to it) A read after write (RAW) data hazard refers to a situation where an instruction refers to a result that has not yet been calculated or retrieved. This can occur because even though an instruction is executed after a prior instruction, the prior instruction has been processed only partly through the pipeline.

Example

For example:

```
i1. R2 <- R1 + R3
i2. R4 <- R2 + R3
```

The first instruction is calculating a value to be saved in register `R2`, and the second is going to use this value to compute a result for register `R4`. However, in a pipeline, when operands are fetched for the 2nd operation, the results from the first will not yet have been saved, and hence a data dependency occurs.

A data dependency occurs with instruction `i2`, as it is dependent on the completion of instruction `i1`.

Write after read (WAR)

(`i2` tries to write a destination before it is read by `i1`) A write after read (WAR) data hazard represents a problem with concurrent execution.

Example

For example:

```
i1. R4 <- R1 + R5
i2. R5 <- R1 + R2
```

In any situation with a chance that `i2` may finish before `i1` (i.e., with concurrent execution), it must be ensured that the result of register `R5` is not stored before `i1` has had a chance to fetch the operands.

Write after write (WAW)

(`i2` tries to write an operand before it is written by `i1`) A write after write (WAW) data hazard may occur in a concurrent execution environment.

Example

For example:

```
i1. R2 <- R4 + R7
i2. R2 <- R1 + R3
```

The write back (WB) of `i2` must be delayed until `i1` finishes executing.

Structural hazards:

A structural hazard occurs when a part of the processor's hardware is needed by two or more instructions at the same time. A canonical example is a single memory unit that is accessed both in the fetch stage where an instruction is retrieved from memory, and the memory stage where data is written and/or read from memory. They can often be resolved by separating the component into orthogonal units (such as separate caches) or bubbling the pipeline.

A structural hazard would for example result from memory access of instruction fetch and memory access of data, were it not for separate data and instruction caches:

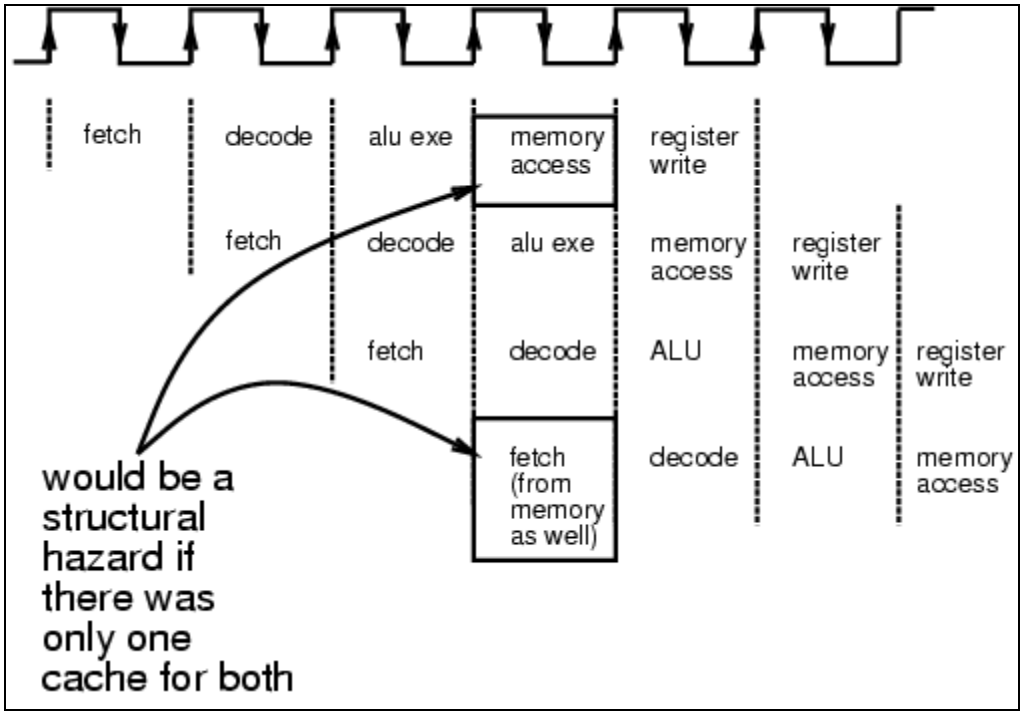


Fig: 2.10 Structural hazards due to instruction fetch and memory access of data

Another example of a structural hazard is when decoding (setting up input registers) makes reference to same register as a register write:

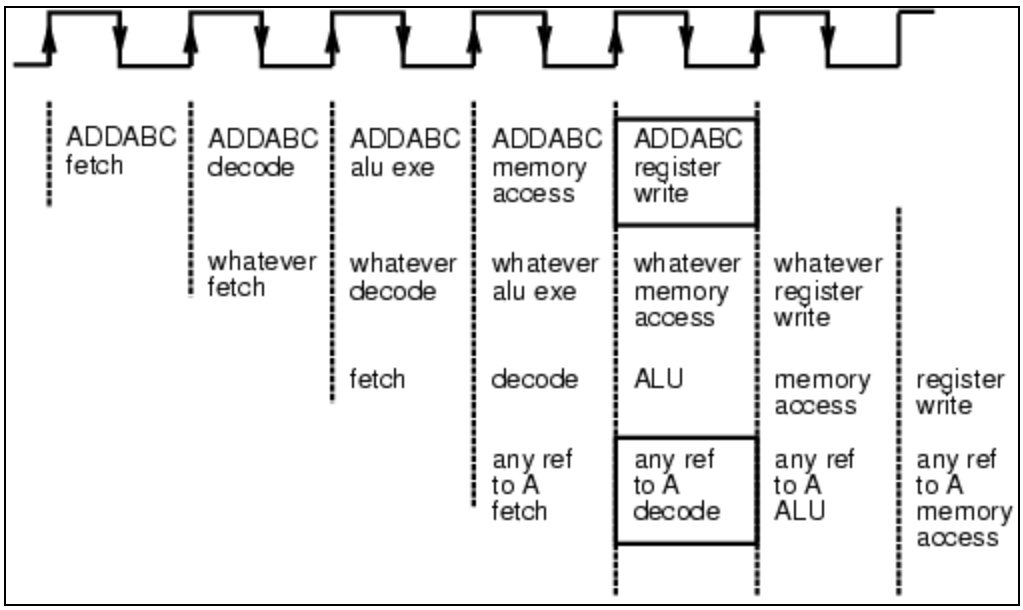


Fig: 2.11 Structural hazards due to reference to same register as a register write

Control hazards (branch hazards):

Branching hazards (also termed control hazards) occur with branches. On many instruction pipeline micro architectures, the processor will not know the outcome of the branch when it needs to insert a new instruction into the pipeline (normally the *fetch* stage).

To avoid control hazards micro architectures can:

- insert a pipeline bubble (discussed above), guaranteed to increase latency, or
- use branch prediction and essentially make educated guesses about which instructions to insert, in which case a pipeline bubble will only be needed in the case of an incorrect prediction

Pipeline bubble or Pipeline Stall:

Bubbling the pipeline, also termed a pipeline break or pipeline stall, is a method to preclude data, structural, and branch hazards. As instructions are fetched, control logic determines whether a hazard could/will occur. If this is true, then the control logic inserts no operations (NOPs) into the pipeline. Thus, before the next instruction (which would cause the hazard) executes, the prior one will have had sufficient time to finish and prevent the hazard. If the number of NOPs equals the number of stages in the pipeline, the processor has been cleared of all instructions and can proceed free from hazards. All forms of stalling introduce a delay before the processor can resume execution.

Flushing the pipeline occurs when a branch instruction jumps to a new memory location, invalidating all prior stages in the pipeline. These prior stages are cleared, allowing the pipeline to continue at the new instruction indicated by the branch.

In computing, a **bubble** or **pipeline stall** is a delay in execution of an instruction in an instruction pipeline in order to resolve a hazard.

During the decoding stage, the control unit will determine if the decoded instruction reads from a register that the instruction currently in the execution stage writes to. If this condition holds, the control unit will stall the instruction by one clock cycle. It also stalls the instruction in the fetch stage, to prevent the instruction in that stage from being overwritten by the next instruction in the program.

To prevent new instructions from being fetched when an instruction in the decoding stage has been stalled, the value in the PC register and the instruction in the fetch stage are preserved to prevent changes. The values are preserved until the bubble has passed through the execution stage.

The execution stage of the pipeline must always be performing an action. A bubble is represented in the execution stage as a NOP instruction, which has no effect other than to stall the instructions being executed in the pipeline.

Timeline

The following is two executions of the same four instructions through a 4-stage pipeline but, for whatever reason, a delay in fetching of the purple instruction in cycle #2 leads to a bubble being created delaying all instructions after it as well.

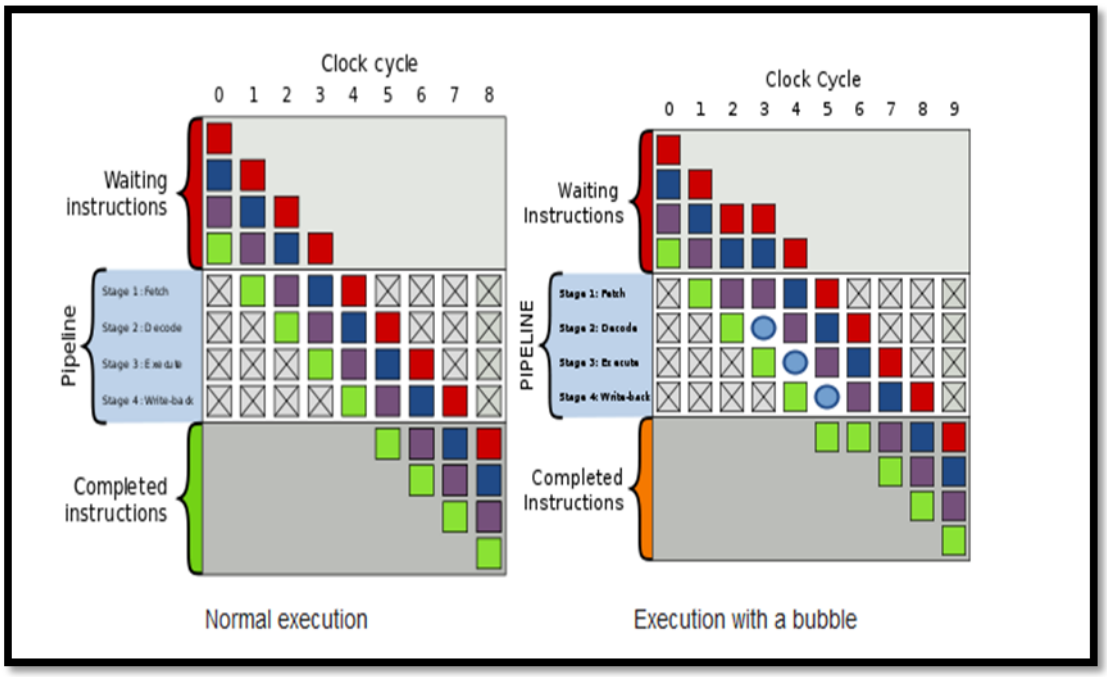


Fig: 2.12 Bubbles in Pipeline

Branch Prediction:

In computer science, predication is an architectural feature that provides an alternative to conditional branch instructions. Predication works by executing instructions from both paths of the branch and only permitting those instructions from the taken path to modify architectural state. The instructions from the taken path are permitted to modify architectural state because they have been associated (predicated) with a predicate, a Boolean value used by the instruction to control whether the instruction is allowed to modify the architectural state or not.

Most computer programs contain conditional code, which will be executed only under specific conditions depending on factors that cannot be determined beforehand, for example depending on user input. As the majority of processors simply execute the next instruction in a sequence, the traditional solution is to insert branch instructions that allow a program to conditionally branch to a different section of code, thus changing the next step in the sequence. This was sufficient until designers began improving performance by implementing instruction pipelining, a method which is slowed down by branches. For a more thorough description of the problems which arose, and a popular solution, see branch predictor.

Luckily, one of the more common patterns of code that normally relies on branching has a more elegant solution. Consider the following pseudo code:

if condition

do this

else

do that

On a system that uses conditional branching, this might translate to machine instructions looking similar to:[1]

branch if condition to label 1

do that

branch to label 2

label 1:

do this

label 2:

...

With predication, all possible branch paths are coded inline, but some instructions execute while others do not. The basic idea is that each instruction is associated with a predicate (the word here used similarly to its usage in predicate logic) and that the instruction will only be executed if the predicate is true. The machine code for the above example using predication might look something like this:

(condition) do this

(not condition) do that

Note that beside eliminating branches, less code is needed in total, provided the architecture provides predicated instructions. While this does not guarantee faster execution in general, it will if the do this and do that blocks of code are short enough.

Predication's simplest form is partial predication, where the architecture has conditional move or conditional select instructions. Conditional move instructions write the contents of one register over another only if the predicate's value is true, whereas conditional select instructions choose which of two registers has its contents written to a third based on the predicate's value. A more generalized and capable form is full predication. Full predication has a set of predicate registers for storing predicates (which allows multiple nested or sequential branches to be simultaneously eliminated) and most instructions in the architecture have a register specifier field to specify which predicate register supplies the predicate.

Advantages:

The main purpose of predication is to avoid jumps over very small sections of program code, increasing the effectiveness of pipelined execution and avoiding problems with the cache. It also has a number of more subtle benefits:

- Functions that are traditionally computed using simple arithmetic and bitwise operations may be quicker to compute using predicated instructions.
- Predicated instructions with different predicates can be mixed with each other and with unconditional code, allowing better instruction scheduling and so even better performance.
- Elimination of unnecessary branch instructions can make the execution of necessary branches, such as those that make up loops, faster by lessening the load on branch prediction mechanisms.
- Elimination of the cost of a branch miss prediction which can be high on deeply pipelined architectures.

Disadvantages:

Predication's primary drawback is in increased encoding space. In typical implementations, every instruction reserves a bit field for the predicate specifying under what conditions that instruction should have an effect. When available memory is limited, as on embedded

devices, this space cost can be prohibitive. However, some architecture such as Thumb-2 are able to avoid this issue (see below). Other detriments are the following:

- Predication complicates the hardware by adding levels of logic to critical paths and potentially degrades clock speed.
- A predicated block includes cycles for all operations, so shorter paths may take longer and be penalized.

Predication is most effective when paths are balanced or when the longest path is the most frequently executed, but determining such a path is very difficult at compile time, even in the presence of profiling information.

Pipeline Performance Analysis

1. CPI of a Pipeline Processor

Suppose an N-segment pipeline processes M instructions without stalls or penalties. We know that it takes N-1 cycles to load (setup) the pipeline, and M cycles to complete the instructions. Thus, the number of cycles is given by:

$$N_{cyc} = N + M - 1$$

The cycles per instruction are easily computed:

$$CPI = N_{cyc}/M = 1 + (N - 1)/M$$

2. Effect of Stalls

Now let us add some stalls to the pipeline processing scheme. Suppose that we have a N-segment pipeline processing M instructions, and we must insert K stalls to resolve data dependencies. This means that the pipeline now has a setup penalty of N-1 cycles, as before, a stall penalty of K cycles, and a processing cost (as before) of M cycles to process the M instructions. Thus, our governing equations become:

$$N_{cyc} = N + M + K - 1$$

and

$$CPI = N_{cyc}/M = 1 + (N + K - 1)/M$$

In practice, what does this tell us? Namely, that the stall penalty (and all the other penalties that we will examine) adversely impact CPI. Here is an example to show how we would analyze the problem of stalls in a pipelined program where the percentage of instructions that incur stalls versus non-stalls are specified.

3. Suppose that an N-segment pipeline executes M instructions, and that a fraction f_{stall} of the instructions require the insertion of K stalls per instruction to resolve data dependencies. The total number of stalls is given by $f_{stall} \cdot M \cdot K$ (fraction of instructions that are stalls, times the total number of instructions, times the average number of stalls per instruction). By substitution, our preceding equations for pipeline performance become:

$$N_{cyc} = N + M + (f_{stall} \cdot M \cdot K) - 1$$

and

$$CPI = N_{cyc}/M = 1 + (f_{stall} \cdot K) + (N - 1)/M$$

So, the CPI penalty due to the combined effects of setup cost and stalls now increases to fK + (N - 1)/M. If $f_{stall} = 0.1$, K = 3, N = 5, and M = 100, then CPI = 1 + 0.3 + 4/100 = 1.34, which is 34 percent larger than the fallacious assumption of CPI = 1.

3. Effect of Exceptions

For purposes of discussion, assume that we have M instructions executing on an N-segment pipeline with no stalls, but that a fraction f_{ex} of the instructions raise an exception in the EX stage. Further assume that each exception requires that (a) the pipeline segments before the EX stage be flushed, (b) that the exception be handled, requiring an average of H cycles per exception, then that (c) the instruction causing the exception and its following instructions be reloaded into the pipeline.

Thus, $f_{ex} \cdot M$ instructions will cause exceptions. In the MIPS pipeline, each of these instructions causes three instructions to be flushed out of the pipe (IF, ID, and EX stages), which incurs a penalty of four cycles (one cycle to flush, and three to reload) plus H cycles to handle the exception. Thus, the pipeline performance equations become:

$$N_{cyc} = N - 1 + (1 - f_{ex}) \cdot M + (f_{ex} \cdot M \cdot (H + 4))$$

which we can rewrite as

$$N_{cyc} = M + [N - 1 - M + (1 - f_{ex}) \cdot M + (f_{ex} \cdot M \cdot (H + 4))]$$

Rearranging terms, the equation for CPI can be expressed as

$$CPI = N_{cyc}/M = 1 + [1 - f_{ex} + (f_{ex} \cdot (H+4)) - 1 + (N - 1)/M]$$

After combining terms, this becomes:

$$CPI = N_{cyc}/M = 1 + [(f_{ex} \cdot (H+3)) + (N - 1)/M]$$

4. Effect of Branches

Branches present a more complex picture in pipeline performance analysis. Recall that there are three ways of dealing with a branch: (1) Assume the branch is not taken, and if the branch is taken, flush the instructions in the pipe after the branch, then insert the instruction pointed

to by the BTA; (2) the converse of 1); and (3) use a delayed branch with a branch delay slot and re-ordering of code (assuming that this can be done).

The first two cases are symmetric. Assume that an error in branch prediction (i.e., taking the branch when you expected not to, and conversely) requires L instruction to be flushed from the pipeline (one cycle for flushing plus L-1 "dead" cycles, since the branch target can be inserted in the IF stage). Thus, the cost of each branch prediction error is L cycles. Further assume that a fraction f_{br} of the instructions are branches and f_{be} of these instructions result in branch prediction errors.

The penalty in cycles for branch prediction errors is thus given by

$$branch\_penalty = f_{br} \cdot f_{be} \cdot M \text{ instructions} \cdot L \text{ cycles per instruction.}$$

The pipeline performance equations then become:

$$N_{cyc} = N - 1 + (1 - f_{br} \cdot f_{be}) \cdot M + (f_{br} \cdot f_{be} \cdot M \cdot L)$$

which we can rewrite as

$$N_{cyc} = M + [N - 1 - M + (1 - f_{br} \cdot f_{be}) \cdot M + (f_{br} \cdot f_{be} \cdot M \cdot L)$$

Rearranging terms, the equation for CPI can be expressed as

$$CPI = N_{cyc}/M = 1 + [(1 - f_{br} \cdot f_{be}) + (f_{br} \cdot f_{be} \cdot L) - 1 + (N - 1)/M].$$

After combining terms, this becomes:

$$CPI = N_{cyc}/M = 1 + [(f_{br} \cdot f_{be} \cdot (L-1)) + (N - 1)/M]$$

In the case of the branch delay slot, we assume that the branch target address is computed and the branch condition is evaluated at the ID stage. Thus, if the branch prediction is correct, there is no penalty. Depending on the method by which the pipeline evaluates the branch and fetches (or pre-fetches) the branch target, a maximum of two cycles penalty (one cycle for flushing, one cycle for fetching and inserting the branch target) is incurred for insertion of a stall in the case of a branch prediction error. In this case, the pipeline performance equations become:

$$N_{cyc} = N - 1 + (1 - f_{br} \cdot f_{be}) \cdot M + (f_{br} \cdot f_{be} \cdot 2M)$$

This implies the following equation for CPI as a function of branches and branch prediction errors:

$$CPI = N_{cyc}/M = 1 + [f_{br} \cdot f_{be} + (N - 1)/M]$$

Since $f_{br} << 1$ is usual, and f_{be} is, on average, assumed to be no worse than 0.5, the product $f_{br} \cdot f_{be}$, which represents the additional branch penalty for CPI in the presence of delayed branch and BDS, is generally small.

MCQ

1. The number of cycles required to complete n tasks in a k stage pipeline is :
    a) K + n – 1
    b) nk + 1
    c) k
    d) none of these
2. The performance of a pipelined processor suffers if
    a) The pipeline stages have different delays
    b) Consecutive instructions are dependent on each other
    c) The pipeline stages share hardware resources
    d) All of these
3. What will be the speed up for a four-stage linear pipeline when the number of instruction n=64?
    a) 4.5
    b) 7.1
    c) 6.5
    d) None of these
4. Dynamic pipeline allows
    a) Multiple functions to evaluate
    b) Only streamline connection
    c) To perform fixed function
    d) None of these
5. The division of stages of a pipeline into sub-stages is the basis for
    a) Pipelining
    b) Super-pipelining
    c) Superscalar
    d) VLIW processor
6. A pipeline stage
    a) Is sequential circuit
    b) Is combinational circuit
    c) Consists of both sequential and combinational circuits
    d) None of these

7. Simplest scheme to handle branches is to

1. Flush pipeline
2. Freezing pipeline
3. Depth of pipeline
4. Both a and b

8. A stall is commonly called a

1. Pipeline bubble
2. Bubble
3. Depth of pipeline
4. Both a and b

9. Load instruction has a delay or latency that cannot be eliminated by forwarding, other technique used is

1. Pipeline interlock
2. Deadlock
3. Stall interlock
4. Stall deadlock

10. When an instruction is stalled, all instructions issued later than stalled instruction and hence not as far along in pipeline, are also

1. Jumped
2. Stopped
3. Pipelined
4. Stalled

11. Control hazards can cause a greater performance loss for MIPS pipeline than do

1. Stall
2. Data hazard
3. Structural hazard
4. Branch hazard

12. Simplest dynamic branch-prediction scheme is a

1. Cancelling
2. Branch history buffer
3. Branch history table
4. Branch prediction table

13. Execution cycle with a branch, delay of one is

1. Delayed branch
2. Branch hazard
3. Structural hazard
4. Data hazard

14. Having load before store in a running program order, then interchanging this order, results in a

1. WAW hazards
2. Destination registers
3. WAR hazards
4. Registers

Short Answer Type Questions:

1. What are the different factors that can affect the performance of a pipelined system Differentiate between WAR and RAW with suitable a example.
2. Write a short note on Pipeline Hazards.
3. What are the different pipeline hazards and what are the remedies?
4. "Instruction execution throughput increases in proportion with the number of pipeline stages." Is it true? Justify your statement.
5. What do you mean by hazard?
6. What are different types of hazards?
7. What do you mean pipeline chaining?
7. What is pipeline stalls?
9. What is delayed branching?
10. What do you mean by Branch Prediction?

Assignments:

1. Consider a 4-stage pipeline that consists of Instruction Fetch (IF), Instruction Decode (ID), Execute (Ex) and Write Back WB stages. The times taken by these stages are 50 ns, 60 ns, 110 ns and 80 ns respectively. The pipeline registers are required after every pipeline stage, and each of these pipeline register consumes 10 ns delay. What is the speedup of the pipeline under ideal conditions compare to the corresponding non-pipelined implementation?

2. Define pipelining technique. Assume a 4 stage pipeline:
   Fetch : Read the instruction from the memory
   Decode : Decode the instruction
   Execute : Execute the ins ruction
   Write : Store the result in destination location
Draw the space - time diagram for pipelining.

Web Reference/Links:

1. https://www.youtube.com/watch?v=DoNqOcmlg9c
2. https://www.youtube.com/watch?v=_klfQh3dKTM
3. https://www.youtube.com/watch?v=q4fwx3h3mdg
4. https://www.tutorialspoint.com/computer_organization/instruction_pipeline_architecture.asp
5. http://www.studytonight.com/computer-architecture/pipelining
6. http://www.dauniv.ac.in/downloads/CArch_PPTs/CompArchCh06L01PipeLine.pdf
7. https://compas.cs.stonybrook.edu/course/cse502-s13/lectures/cse502-L4-pipelining.pdf
8. http://www2.cs.siu.edu/~cs401/Textbook/ch3.pdf
9. http://www.mhhe.com/engcs/electrical/hamacher/5e/graphics/ch08_453-510.pdf

MODULE 3
LECTURE 1

PARALLELISM

With the era of increasing processor speeds slowly coming to an end, computer architects are exploring new ways of increasing throughput. One of the most promising is to look for and exploit different types of parallelism in code.

TYPES OF PARALLELISM

There are three main types of parallelism:-

1. Instruction Level Parallelism

2. Data Level Parallelism

3. Thread Level Parallelism

Instruction Level Parallelism

Instruction-level parallelism (ILP) is a measure of how many of the instructions in a computer program can be executed simultaneously.

Pipelining can overlap the execution of instructions when they are independent of one another. This potential overlap among instructions is called instruction-level parallelism (ILP) since the instructions can be evaluated in parallel.

(ILP) is a measure of how many of the operations in a computer program can be performed simultaneously. Consider the following program:

1. e = a + b

2. f = c + d

3. g = e * f

Operation 3 depends on the results of operations 1 and 2, so it cannot be calculated until both of them are completed. However, operations 1 and 2 do not depend on any other operation, so they can be calculated simultaneously. If we assume that each operation can be completed in one unit of time then these three instructions can be completed in a total of two units of time, giving an ILP of 3/2.

A goal of compiler and processor designers is to identify and take advantage of as much ILP as possible. ILP allows the compiler and the processor to overlap the execution of multiple instructions or even to change the order in which instructions are executed.

How much ILP exists in programs is very application specific. In certain fields, such as graphics and scientific computing the amount can be very large. However, workloads such as cryptography exhibit much less parallelism.

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

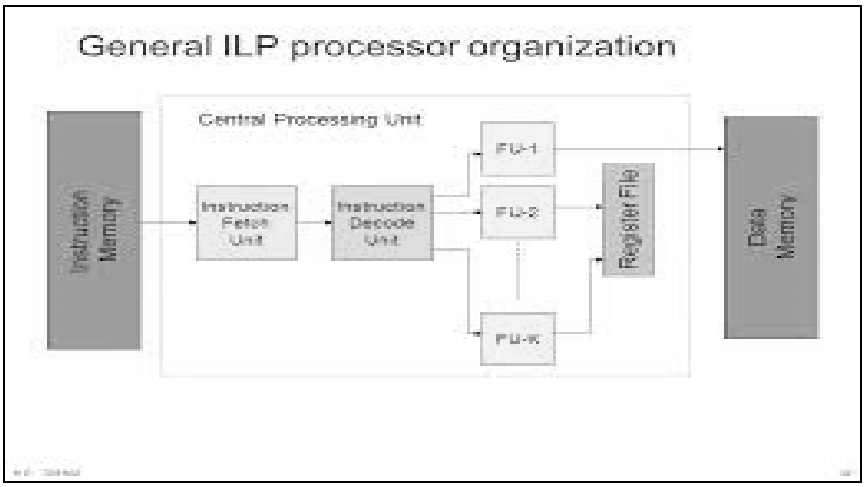


Fig: 3.1 ILP Processor

LECTURE 2

Superscalar processor

A superscalar processor is a CPU that implements a form of parallelism called instruction-level parallelism within a single processor**.**

In contrast to a scalar processor that can execute at most one single instruction per clock cycle, a superscalar processor can execute more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to different execution units on the processor. It therefore allows for more throughput (the number of instructions that can be executed in a unit of time) than would otherwise be possible at a given clock rate. Each execution unit is not a separate processor (or a core if the processor is a multi-core processor), but an execution resource within a single CPU such as an arithmetic logic unit.

The superscalar technique is traditionally associated with several identifying characteristics (within a given CPU):

1.Instructions are issued from a sequential instruction stream

2.The CPU dynamically checks for data dependencies between instructions at run time (versus software checking at compile time)

3.The CPU can execute multiple instructions per clock cycle.

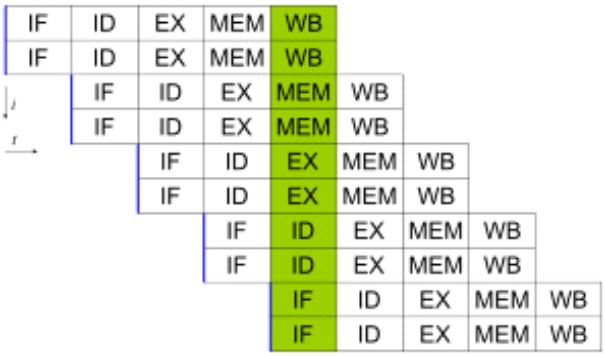


fig:- Simple superscalar pipeline. By fetching and dispatching two instructions at a time, a maximum of two instructions per cycle can be completed. (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back, i = Instruction number, t = Clock cycle [i.e., time])

Fig:3.2 Superscalar Execution

Super pipelined Processors

Traditional pipelined architectures have a single pipeline stage for each of instruction cycle stage: instruction fetch, instruction decode, memory read, ALU operation and memory write.

A super pipelined processor has a pipeline where each of these logical steps may be sub divided into multiple pipeline stages.

In contrast to a superscalar processor, a super pipelined one has split the main computational pipeline into more stages. Each stage is simpler (does less work) and thus the clock speed can be increased. However the *latency*, measured in clock cycles, for any instruction to complete has increased from 4 cycles in early RISC processors to 8 or more.

Benefit

The major benefit of super pipelining is the increase in the number of instructions which can be in the pipeline at one time and hence the level of parallelism.

Drawbacks

The larger number of instructions "in flight" (*ie* in some part of the pipeline) at any time, increases the potential for data dependencies to introduce stalls. Simulation studies have suggested that a pipeline depth of more than 8 stages tends to be counter-productive.

Superscalar vs. Super pipelined

- Superscalar machines can issue several instructions per cycle. Super pipelined machines can issue only one instruction per cycle, but they have cycle times shorter than the time required for any operation.
- Both of these techniques exploit instruction-level parallelism, which is often limited in many applications. Super pipelined machines are shown to have better performance and less cost than superscalar machines.

LECTURE - 3

VLIW (Very Long Instruction Word)

Very long instruction word (VLIW) describes a computer processing architecture in which a language compiler or pre-processor breaks program instruction down into basic operations that can be performed by the processor in parallel (that is, at the same time). These operations are put into a very long instruction word which the processor can then take apart without further analysis, handing each operation to an appropriate functional unit.

VLIW is sometimes viewed as the next step beyond the reduced instruction set computing (RISC) architecture, which also works with a limited set of relatively basic instructions and can usually execute more than one instruction at a time (a characteristic referred to as *superscalar*).

The VLIW architecture is generalized from two well-established concepts: horizontal microcoding and superscalar processing. A typical VLIW (very long instruction word) machine has instruction words hundreds of bits in length. As illustrated in fig. 4.14a, multiple functional units are used concurrently in a VLIW processor. All functional units share the use of a common large register file. The operations to be simultaneously executed by the functional units are synchronised in a VLIW instruction, say , 256 or 1024 bits per instruction word, as implemented in the Multi-flow computer models .

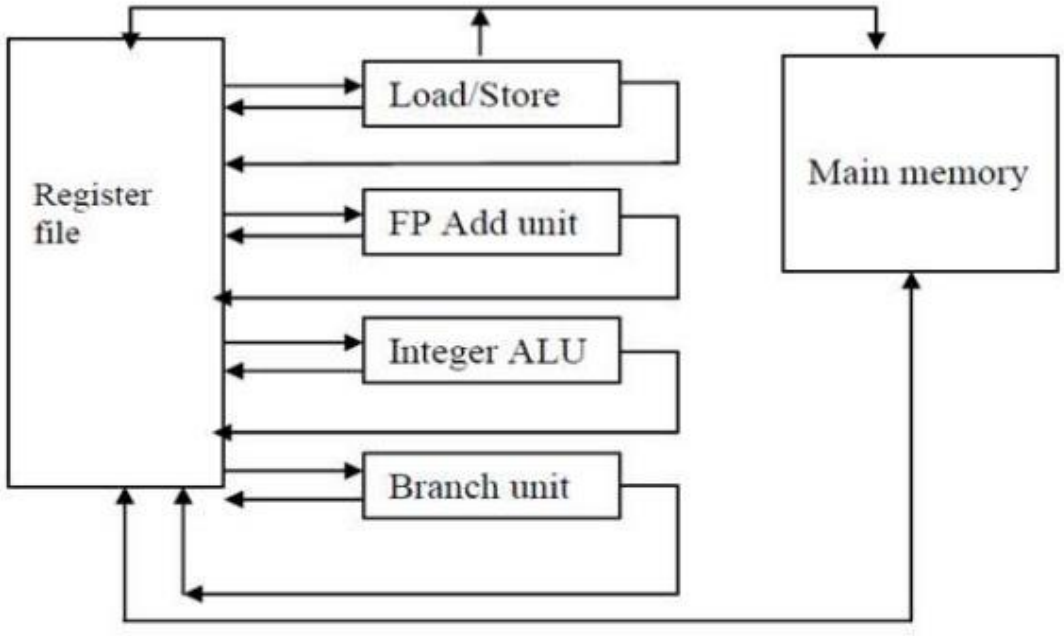


Fig: 3.3VLIW Architecture

The VLIW concept is borrowed from horizontal micro coding. Different fields of the long instruction word carry the opcodes to be dispatched to different functional units. Programs written in conventional short instruction words (say 32 bits) must be compacted together to form the VLIW instructions. This code compaction must be done by a compiler which can predict branch outcomes using elaborate heuristics or run-time statistics.

The main advantage of VLIW processors is that complexity is moved from the hardware to the software, which means that the hardware can be smaller, cheaper, and require less power to operate. The challenge is to design a compiler or pre-processor that is intelligent enough to decide how to build the very long instruction words. If dynamic pre-processing is done as the program is run, performance may be a concern.

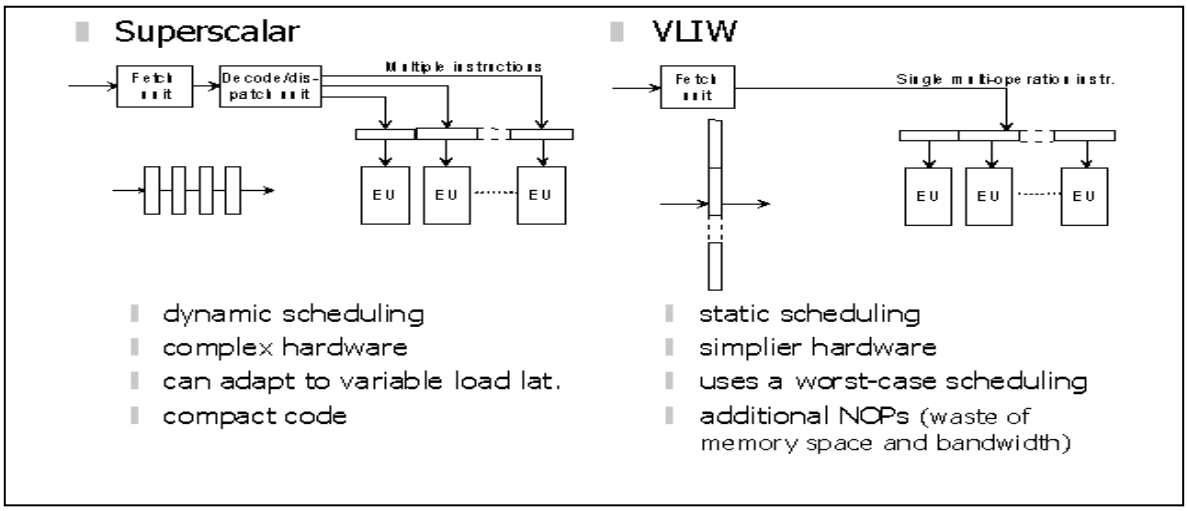


Fig: 3.4 Superscalar Vs VLIW Architecture

Super Pipelined:
In contrast to a superscalar processor, a super pipelined one has split the main computational pipeline into more stages. Each stage is simpler (does less work) and thus the clock speed can be increased. However the latency, measured in clock cycles, for any instruction to complete has increased from 4 cycles in early RISC processors to 8 or more.

The major benefit of superpipelining is the increase in the number of instructions which can be in the pipeline at one time and hence the level of parallelism.

.

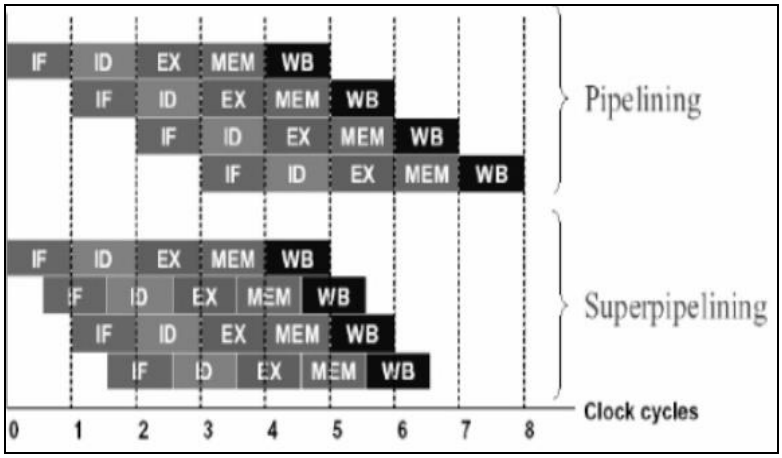


Fig: 3.5 Pipelining Vs Super pipelining Architecture

The larger number of instructions "in flight" (ie in some part of the pipeline) at any time, increases the potential for data dependencies to introduce stalls. Simulation studies have suggested that a pipeline depth of more than 8 stages tends to be counter-productive

Super pipelining is based on dividing the stages of a pipeline into sub-stages and thus increasing the number of instructions which are supported by the pipeline at a given moment. By dividing each stage into two, the clock cycle period t will be reduced to the half, t/2; hence, at the maximum capacity, the pipeline produces a result every t/2 s. For a given architecture and the corresponding Instruction set there is an optimal number of pipeline stages; increasing the number of stages over this limit reduces the overall performance. A solution to further improve speed is the Superscalar architecture.

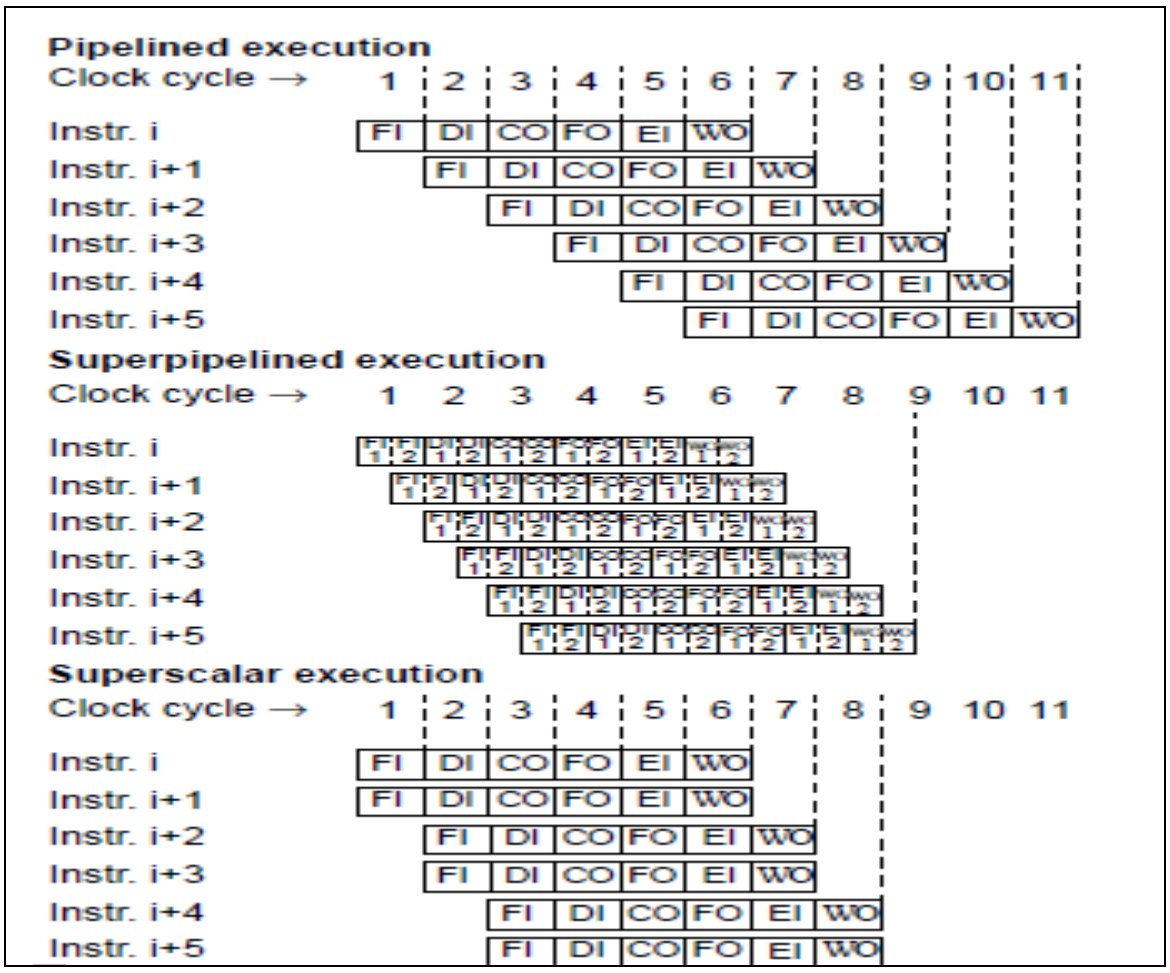


Fig: 3.6 Pipeline Vs Super Pipeline Vs Superscalar

LECTURE - 4

Array Processor and its Types

A computer/processor that has an architecture especially designed for processing arrays (e.g. matrices) of numbers. The architecture includes a number of processors (say 64 by 64) working simultaneously, each handling one element of the array, so that a single operation can apply to all elements of the array in parallel. To obtain the same effect in a conventional processor, the operation must be applied to each element of the array sequentially and so consequently much more slowly.

An array processor may be built as a self-contained unit attached to a main computer via an I/O port or internal bus; alternatively, it may be a distributed array processor where the processing elements are distributed throughout, and closely linked to, a section of the computer's memory.

Array processors are very powerful tools for handling problems with a high degree of parallelism. They do however demand a modified approach to programming. The conversion of conventional (sequential) programs to serve array processors is not a trivial task, and it is sometimes necessary to select different (parallel) algorithms to suit the parallel approach.

Types:

There are basically two types of array processors:

Attached Array Processors

An attached array processor is a processor which is attached to a general purpose computer and its purpose is to enhance and improve the performance of that computer in numerical computational tasks. It achieves high performance by means of parallel processing with multiple functional units.

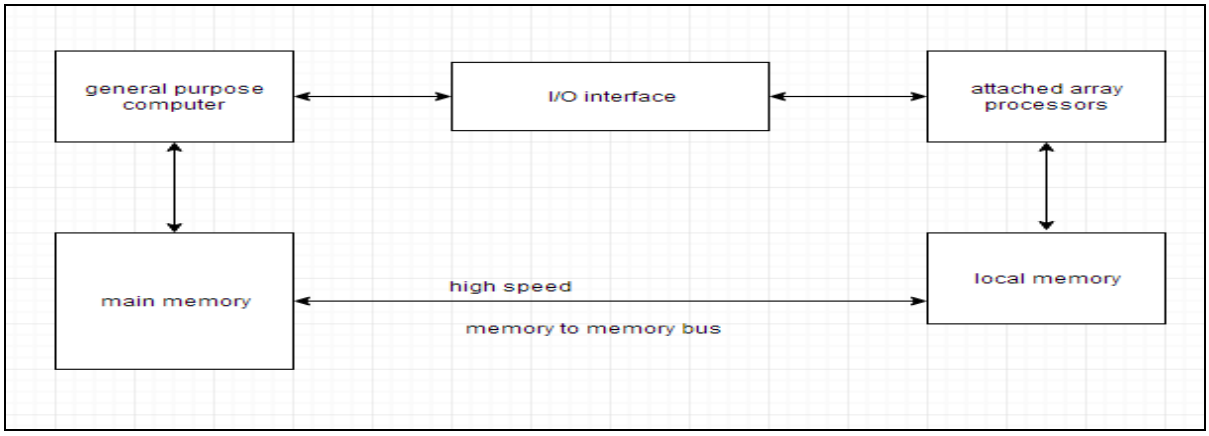


Fig: 3.5 Attached Array Processors

SIMD Array Processors

SIMD is the organization of a single computer containing multiple processors operating in parallel. The processing units are made to operate under the control of a common control unit, thus providing a single instruction stream and multiple data streams.

A general block diagram of an array processor is shown below. It contains a set of identical processing elements (PE's), each of which is having a local memory M. Each processor element includes an **ALU**and **registers**. The master control unit controls all the operations of the processor elements. It also decodes the instructions and determines how the instruction is to be executed.

The main memory is used for storing the program. The control unit is responsible for fetching the instructions. Vector instructions are send to all PE's simultaneously and results are returned to the memory.

The best known SIMD array processor is the **ILLIAC IV** computer developed by the **Burroughs corps**. SIMD processors are highly specialized computers. They are only suitable for numerical problems that can be expressed in vector or matrix form and they are not suitable for other types of computations.

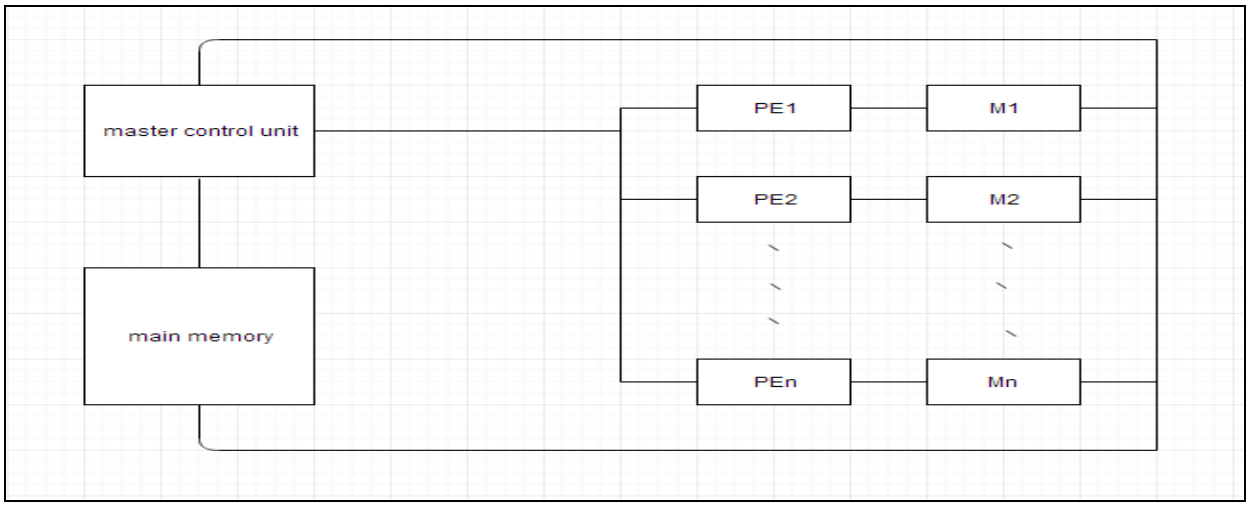


Fig: 3.6 SIMD Array Processors

Why use the Array Processor

- Array processors increase the overall instruction processing speed.
- As most of the Array processors operate asynchronously from the host CPU, hence it improves the overall capacity of the system.
- Array Processors has its own local memory, hence providing extra memory for systems with low memory.

VECTOR PROCESSOR

Vector Operations

- Arithmetic operations on large arrays of numbers
- Conventional scalar processor

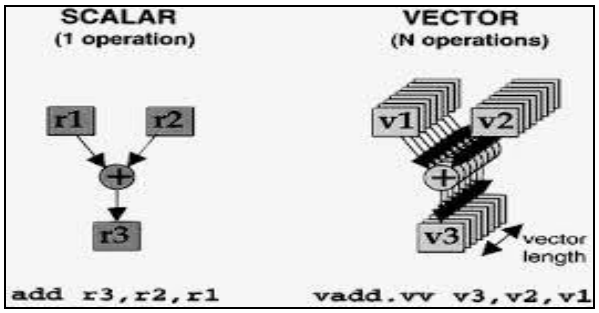


Fig: 3.7

» Machine language

```
    Initialize I = 0
20  Read A(I)
    Read B(I)
    Store C(I) = A(I) + B(I)
    Increment I = I + 1
    If I ≤ 100 go to 20
    Continue
```

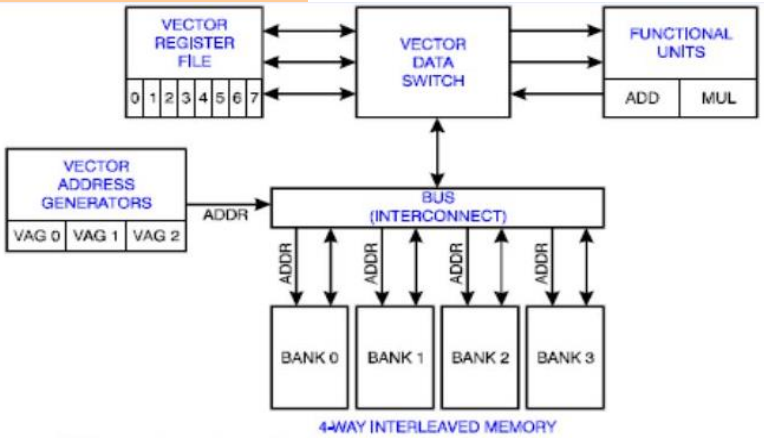
» Fortran language

```
    DO 20 I = 1, 100
20  C(I) = A(I) + B(I)
```

Vector processor

- » Single vector instruction

```
C(1:100) = A(1:100) + B(1:100)
```



- Vector Address Generators drive strided fetches/stores
- Interconnect/Bus supports pipelined+interleaved loads and stores
- Vector Register File holds collated data sets
- Functional units perform pipelined operations on vector register contents

Fig: 3.8 Vector Processor Architecture

Vector Instruction Format

Operation code	Base address source 1	Base address source 2	Base address destination	Vector length
ADD	A	B	C	100

- Matrix Multiplication

3 x 3 matrices multiplication : $n^2 = 9$ inner product

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

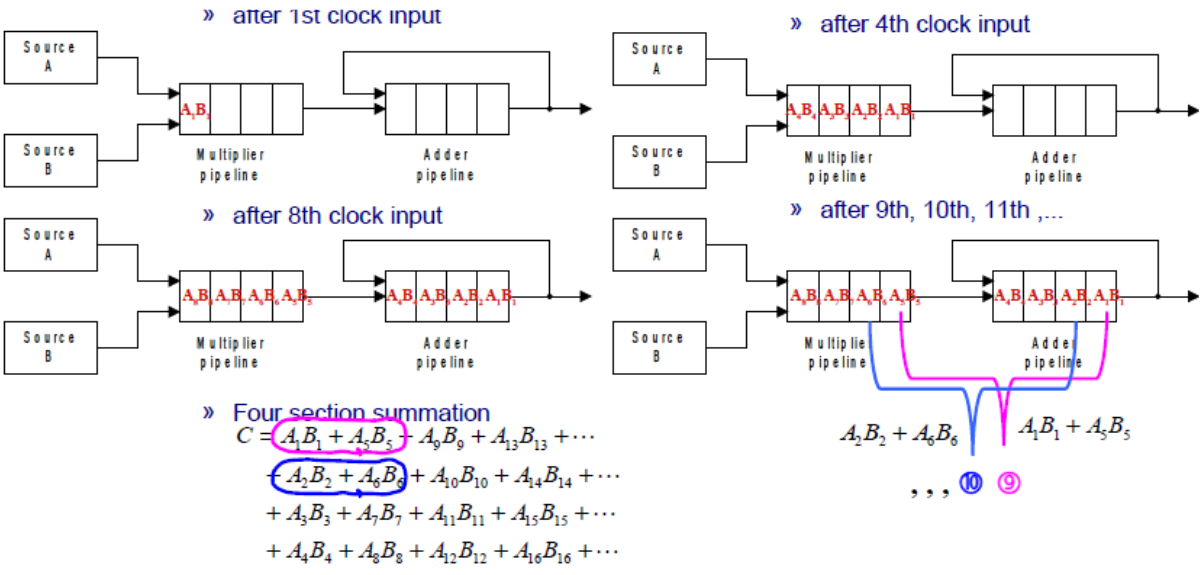
» $c_{11} = a_{11} b_{11} + a_{12} b_{21} + a_{13} b_{31}$

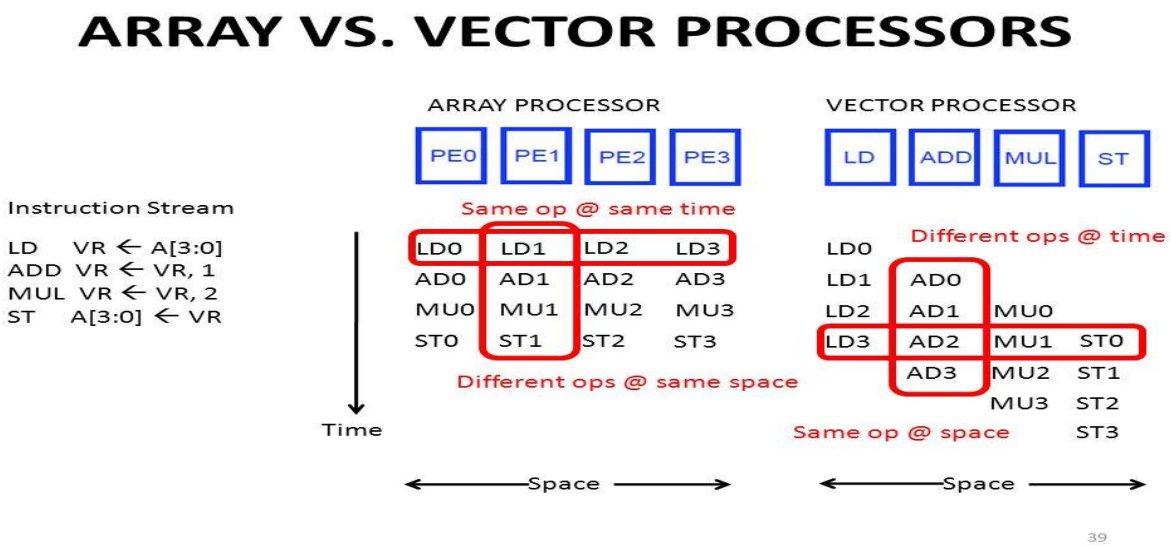
Cumulative multiply-add operation

$c = c + a \times b$

» $c_{11} = c_{11} + a_{11} b_{11} + a_{12} b_{21} + a_{13} b_{31}$; $n^3 = 27$ multiply-add

$$C = A_1B_1 + A_2B_2 + A_3B_3 + \cdots + A_kB_k$$





References:

1. https://www.cc.gatech.edu/~milos/Teaching/CS6290F07/3_ILP.pdf

2. https://www.scribd.com/doc/33700101/Instruction-Level-Parallelism

3. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.141.2892&rep=rep1&type=pdf

4. https://www.cs.ucf.edu/~dcm/Teaching/CDA5106-Fall2015/Slides/CH3.pdf

MCQ:

I. The vector stride value is required
   a) To deal with the length of vectors
   b) To find the parallelism in vectors
   c) To access the elements in multi dimensional vectors
   d) To execute vector instruction

II. Superscalar processors have CPI of
   a) Less than 1
   b) Greater than 1
   c) More than 2
   d) Greater than 3

III. Portability is definitely an issue for which of the following architectures?
   a) VLIW processor
   b) Super Scalar processor
   c) Super pipelined
   d) None of these

IV. Array process is present in
a) MIMD
b) MISD
c) SISD
d) SIMB

V. Basic difference between vector processor and array processor is :
a) Pipelining
b) Interconnection network
c) Register
d) None of these

VI. Array processor is put under which of these categories?
a) SISD
b) SIMD
c) MISD
d) MIMD

VII. Array processors perform computations to exploit
a) Temporal parallelism
b) Spatial parallelism
c) Sequential behaviour of programs
d) Modularity of programs

VIII. Primary challenge for every multiple-issue processors is trying to exploiting large amount of

1. IP
2. FLP
3. FP
4. ILP

IX. VLIW stands for
a) Very Long Instruction Word
b) Very Long Instruction Width
c) Very Large Instruction Word
d) Very Long Instruction Width

X. The important feature of the VLIW is _____.
a) ILP
b) Cost effectiveness

c) Performance
d) None of the mentioned

XI. The main difference between the VLIW and the other approaches to improve performance is
a) Cost effectiveness.
b) Increase in performance
c) Lack of complex hardware design
d) All of the above

XII. In VLIW the decision for the order of execution of the instructions depends on the program itself.
a) True
b) False

XIII. The parallel execution of operations in VLIW is done according to the schedule determined by _____.
a) Task scheduler
b) Interpreter
c) Compiler
d) Encoder

XIV. The VLIW processors are much simpler as they don not require of _____.
a) Computational register
b) Complex logic circuits
c) SSD slots
d) Scheduling hardware
XV. The VLIW architecture follows _____ approach to achieve parallelism.
a) MISD
b) SISD
c) SIMD
d) MIMD

XVI. The following instruction is allowed in VLIW:
f12 = f0 * f4, f8 = f8 + f12, f0 = dm(i0, m3), f4 = pm(i8, m9);
a) True
b) False
XVII. To compute the direction of the branch the VLIW uses _____.
a) Seekers
b) Heuristics
c) Direction counter
d) Compass
XVIII. EPIC stands for
a) Explicitly Parallel Instruction Computing
b) External Peripheral Integrating Component
c) External Parallel Instruction Computing
d) None of the mentioned

Short Answer Type Questions:

1. Write short notes on Array Processor.
2. Compare superscalar , super-pipeline and VLIW techniques.
3. Discuss about strip mining and vector stride in vector processors.
4. Explain the concept of strip mining used in vector processors. Why do vector processors use memory banks ?
5. Briefly describe the VLIW processor architecture. What are the limitations of VLIW ?
6. Discuss different types of vector instructions.

Web Reference/Link:

1. https://www.youtube.com/watch?v=Ri-9vA2Xltg
2. https://www.youtube.com/watch?v=T4t6vCPSeYs
3. https://www.youtube.com/watch?v=zBn2YGPTL_Q
4. https://www.youtube.com/watch?v=CjR9bqdIfZM
5. https://www.youtube.com/watch?v=T9B_DtYTKCc
6. https://www.youtube.com/watch?v=jn637mopME8

MODULE 4

LECTURE 1

Memory Hierarchy

The memory technology and storage organization at each level are characterized by five parameters: the access time, memory size, cost per byte, transfer bandwidth and unit of transfer.

There are four major storage levels:

1. Internal - Processor registers and cache
2. Main - the system RAM
3. Mass storage - Secondary storage

Storage devices such as registers, caches, main memory, disk devices and tape units are often organized as a hierarchy as in the given figure.

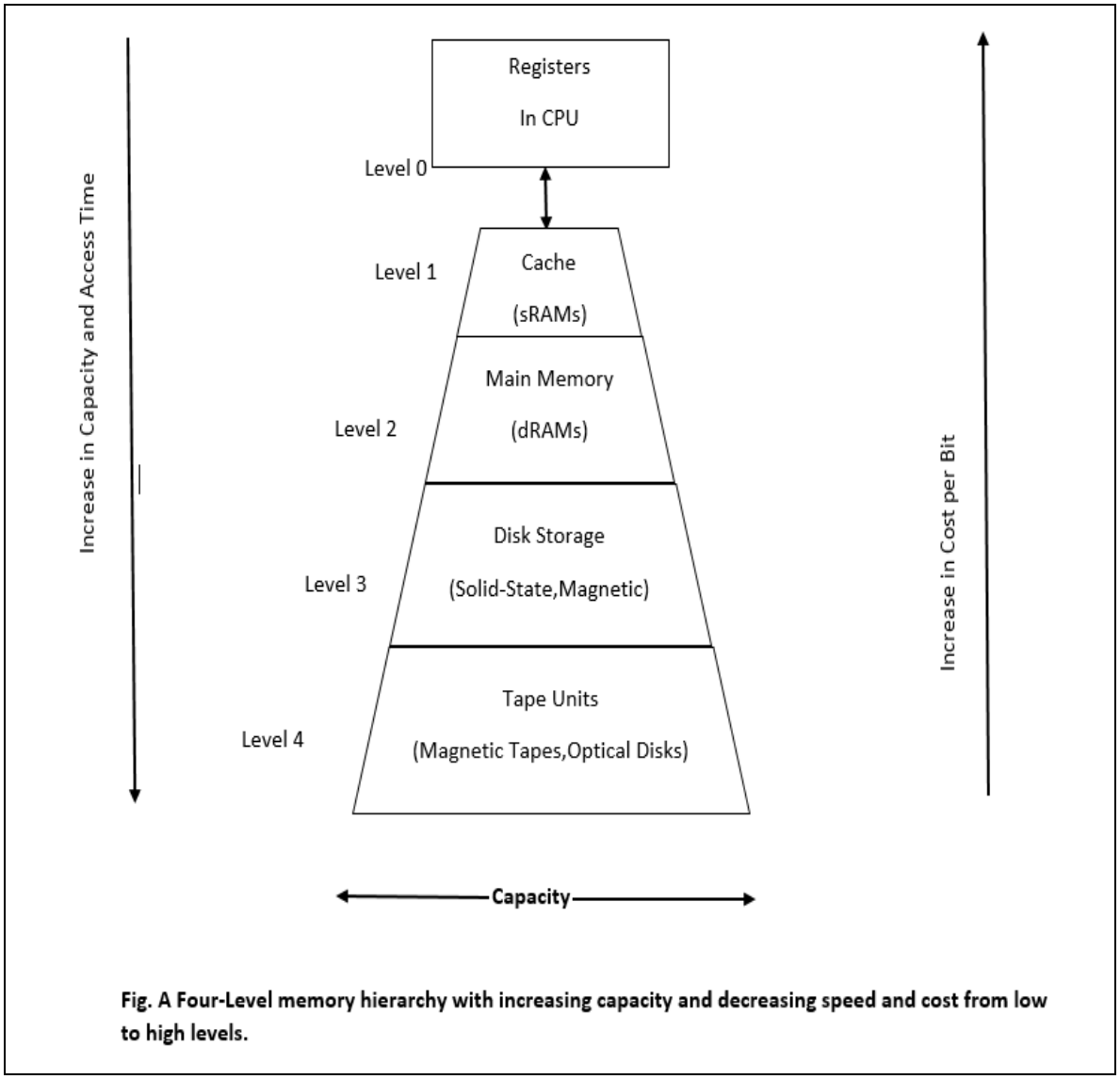


Fig: 4.1Memory Hierarchy

LECTURE 2

Registers and caches

The register and the cache are parts of the processor complex, built either on the processor chip or on the processor board. Register assignment is often made by the compiler. Register transfer operations are directly controlled by the processor after instructions are decoded. Register transfer is conducted at processor speed , usually in on clock cycle.

Therefore, many designers would not consider registers a level of memory. We list them here for comparison purposes. The cache is controlled by the MMU and is programmer-transparent. The cache can also be implemented at one or multiple levels, depending on the speed and application requirements.

Main Memory

The main memory of the computer is also known as RAM, standing for Random Access Memory. It is constructed from integrated circuits and needs to have electrical power in order to maintain its information. It can be directly accessed by the CPU. The access time to read or write any particular byte are independent of where about in the memory that byte is, and currently is approximately 50 nanoseconds (a thousand millionth of a second). This is broadly comparable with the speed at which the CPU will need to access data. Main memory is expensive compared to external memory so it has limited capacity. The capacity available for a given price is increasing all the time. For example many home Personal Computers now have a capacity of 16 megabytes (million bytes), while 64 megabytes is commonplace on commercial workstations. The CPU will normally transfer data to and from the main memory in groups of two, four or eight bytes, even if the operation it is undertaking only requires a single byte.

Secondary Memory

Secondary memory is where programs and data are kept on a long-term basis. Common secondary storage devices are the hard disk and optical disks.

- The hard disk has enormous storage capacity compared to main memory.
- The hard disk is usually contained inside the case of a computer.
- The hard disk is used for long-term storage of programs and data.
- Data and programs on the hard disk are organized into files.
- A file is a collection of data on the disk that has a name.

A hard disk might have a storage capacity of 500 gigabytes (room for about 500 x 10^9 characters). This is about 100 times the capacity of main memory. A hard disk is slow compared to main memory. If the disk were the only type of memory the computer system would slow down to a crawl. The reason for having two types of storage is this difference in speed and capacity.

Large blocks of data are copied from disk into main memory. The operation is slow, but lots of data is copied. Then the processor can quickly read and write small sections of that data in main memory. When it is done, a large block of data is written to disk.

Often, while the processor is computing with one block of data in main memory, the next block of data from disk is read into another section of main memory and made ready for the processor. One of the jobs of an operating system is to manage main storage and disks this way.

LECTURE 3

Mapping Technique in cache memory

The memory system has to quickly determine if a given address is in the cache

There are three popular methods of mapping addresses to cache locations

–Fully Associative – Search the entire cache for an address

– Direct – Each address has a specific place in the cache

–Set Associative – Each address can be in any of a small set of cache locations

Direct Mapping

Each block of main memory maps to only one cache line
—i.e. if a block is in cache, it must be in one specific place
i = j modulo m

i = cache line number j = main memory block number m = number of lines in cache

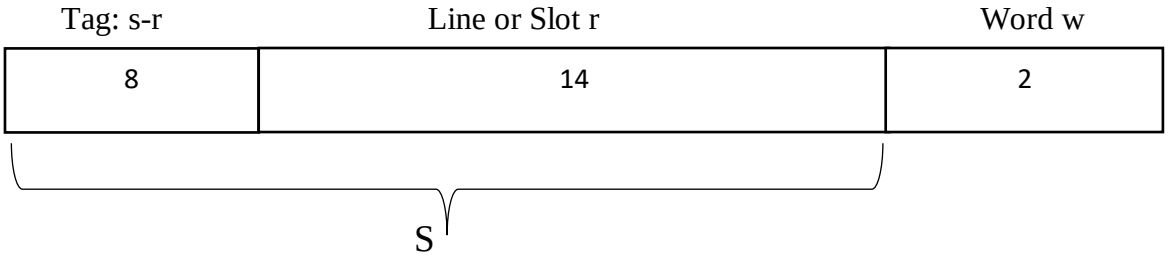
•Address is in two parts

•Least Significant w bits identify unique word

•Most Significant s bits specify one memory block

•The MSBs are split into a cache line field r and a tag of s-r (most significant)

Direct Mapping Address Structure



•24 bit address (224 = 16M main memory)

•2 bit word identifier (22 = 4 byte block)

•22 bit block identifier (s)

—8 bit tag (s-r =22-14)

—14 bit slot or line

•No two blocks in the same line have the same Tag field

•Check contents of cache by finding line and checking Tag

Direct Mapping Cache Line Table

•Cache line Main Memory blocks held

•0 0, m, 2m, 3m…2s-m

•1 1,m+1, 2m+1…2s-m+1

•m-1 m-1, 2m-1,3m-1…2s-1

Direct Mapping Cache Organization

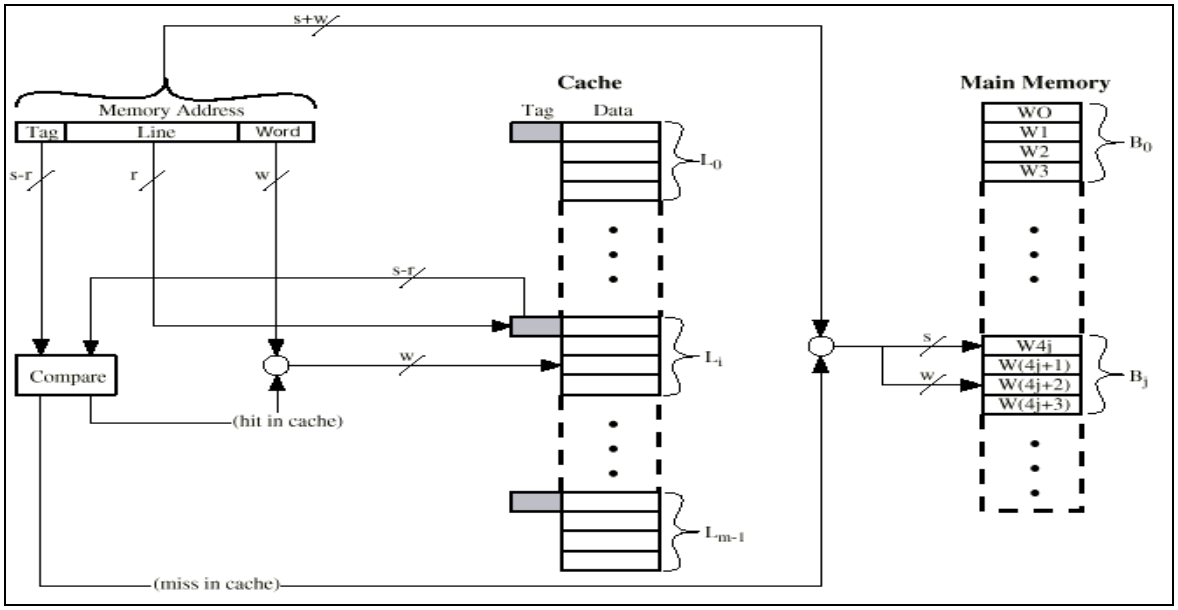


Fig: 4.2 Direct Mapping Examples

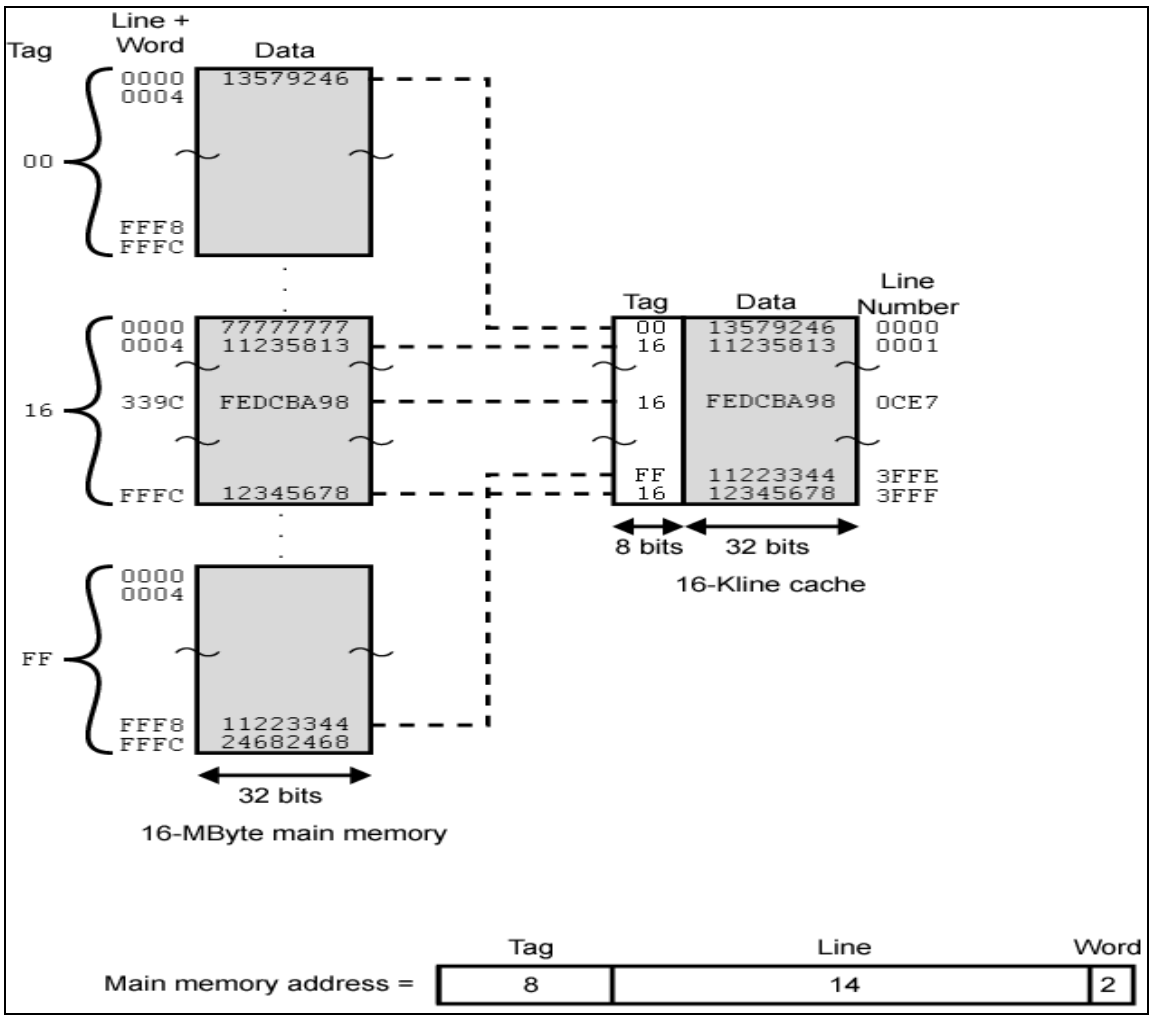


Fig: 4.3 Direct Mapping Example

Direct Mapping Summary

•Address length = (s + w) bits

•Number of addressable units = 2s+w words or bytes

•Block size = line size = 2w words or bytes

•Number of blocks in main memory =

•2s+ w/2w = 2s

•Number of lines in cache = m = 2r

•Size of tag = (s – r) bits

LECTURE 4

Associative Mapping

•A main memory block can load into any line of cache

•Memory address is interpreted as tag and word

•Tag uniquely identifies block of memory

•Every line's tag is examined for a match

•Cache searching gets expensive

Fully Associative Cache Organization

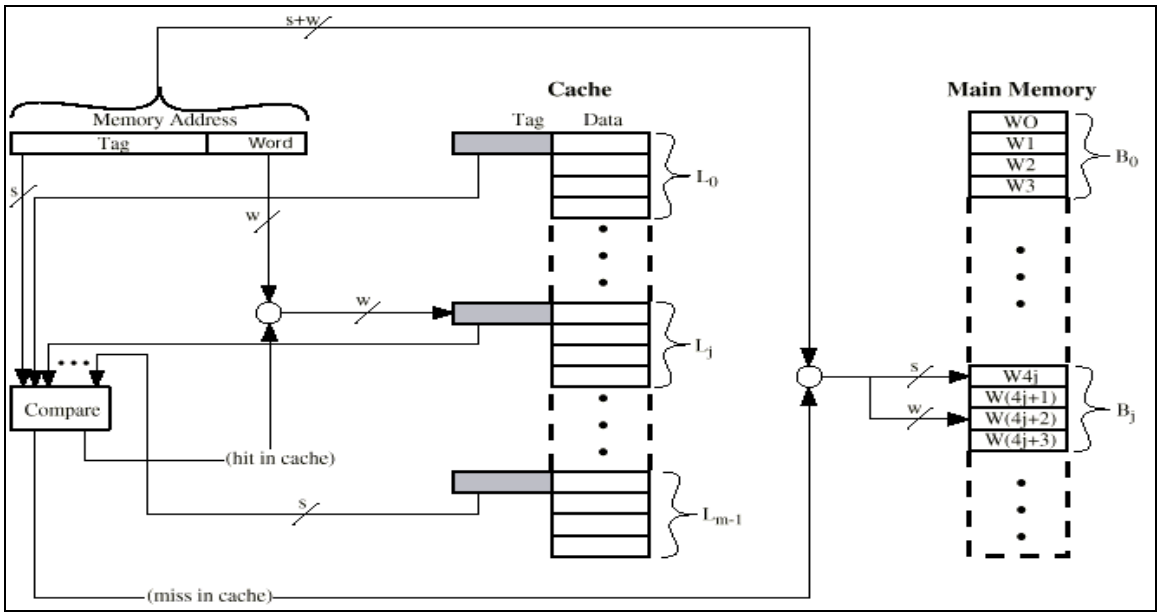


Fig: 4.4 Fully Associative Cache Organizations

Associative Mapping Example

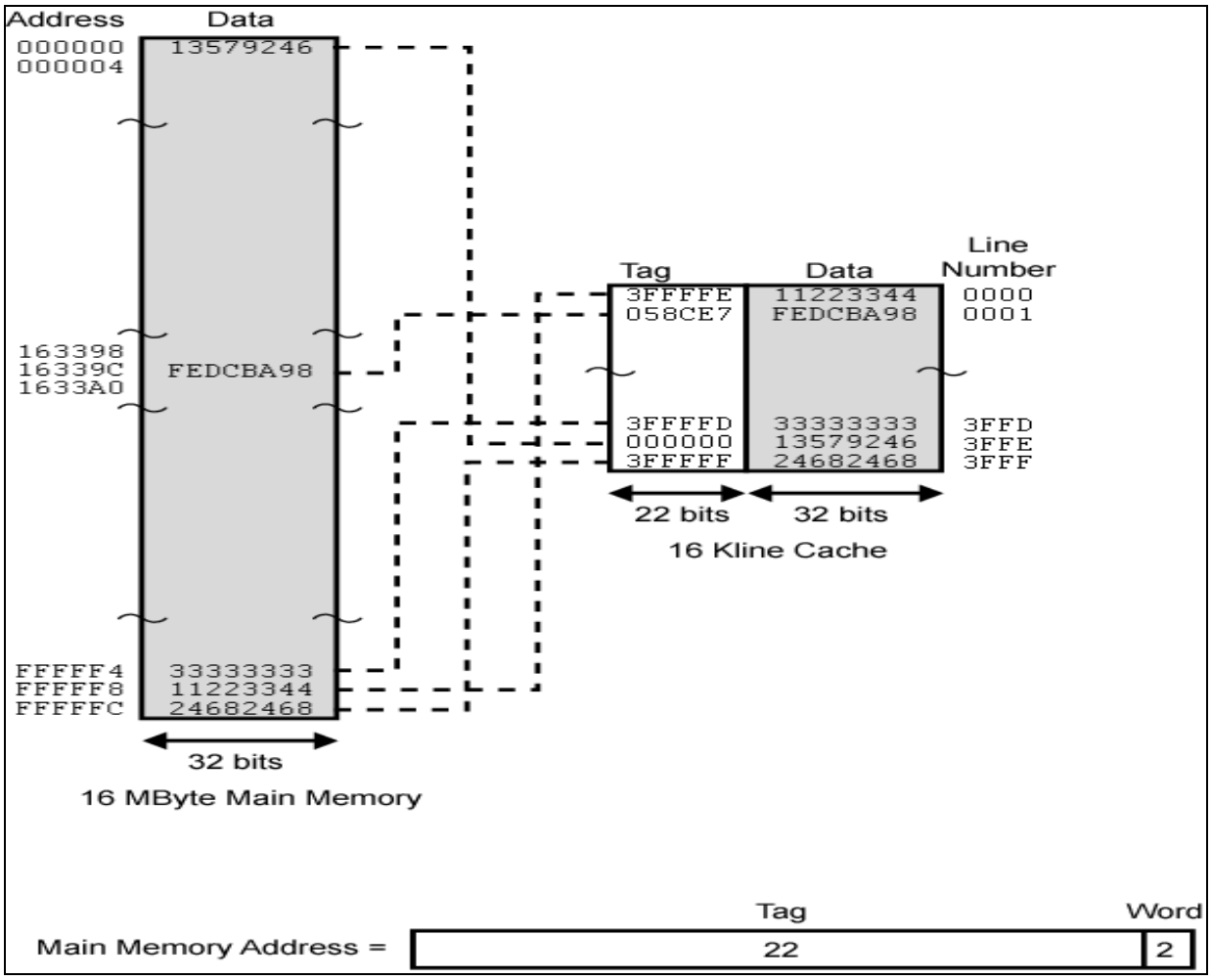


Fig : 4.5 Associative Mapping Example

Associative Mapping Address Structure

S	W
Tag: 22 bit	Word 2 bit

•22 bit tag stored with each 32 bit block of data

•Compare tag field with tag entry in cache to check for hit

•Least significant 2 bits of address identify which 8 bit word is required from 32 bit data block

•e.g.

—Address Tag Data Cache line

—FFFFFC 3FFFFF 24682468 3FFF

Associative Mapping Summary

•Address length = (s + w) bits

•Number of addressable units = 2s+w words or bytes

•Block size = line size = 2w words or bytes

•Number of blocks in main memory = 2s+w/2w = 2s

•Number of lines in cache = not determined by the address format

•Size of tag = s bits

Set Associative Mapping

•Cache is divided into a number of sets (v)

•Each set contains a number of lines (k)

•A given block maps to any line in a given set

—e.g. Block B can be in any line of set I

—m = k v

—i = j modulo v

i = cache line number
j = main memory block number
m = number of lines in cache

•e.g. 2 lines per set

—2 way associative mapping

—A given block can be in one of 2 lines in only one set

Set Associative Mapping Summary

•Address length = (s + w) bits

•Number of addressable units = 2s+w words or bytes

•Block size = line size = 2w words or bytes

•Number of blocks in main memory = 2s+w/2w = 2s

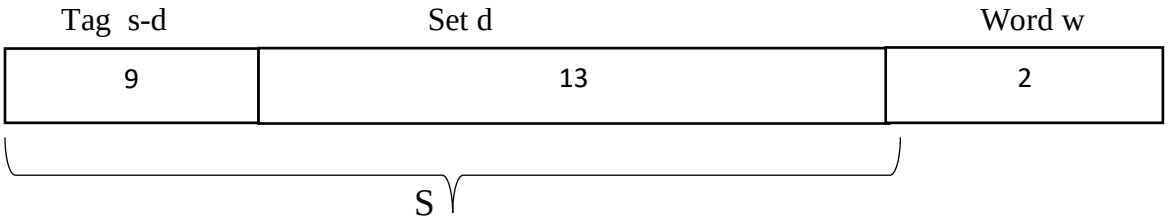
•Number of lines in set = k (k-way set associative mapping)

•Size of set field = d bits

•Number of sets = v = 2d

•Number of lines in cache = kv = k 2d

•Size of tag = (s – d) bits



Set Associative Mapping Example
•13 bit set number

•Block number in main memory is modulo 213

•000000, 008000, ..., FF8000 map to same set

k - Way Set Associative Cache Organization

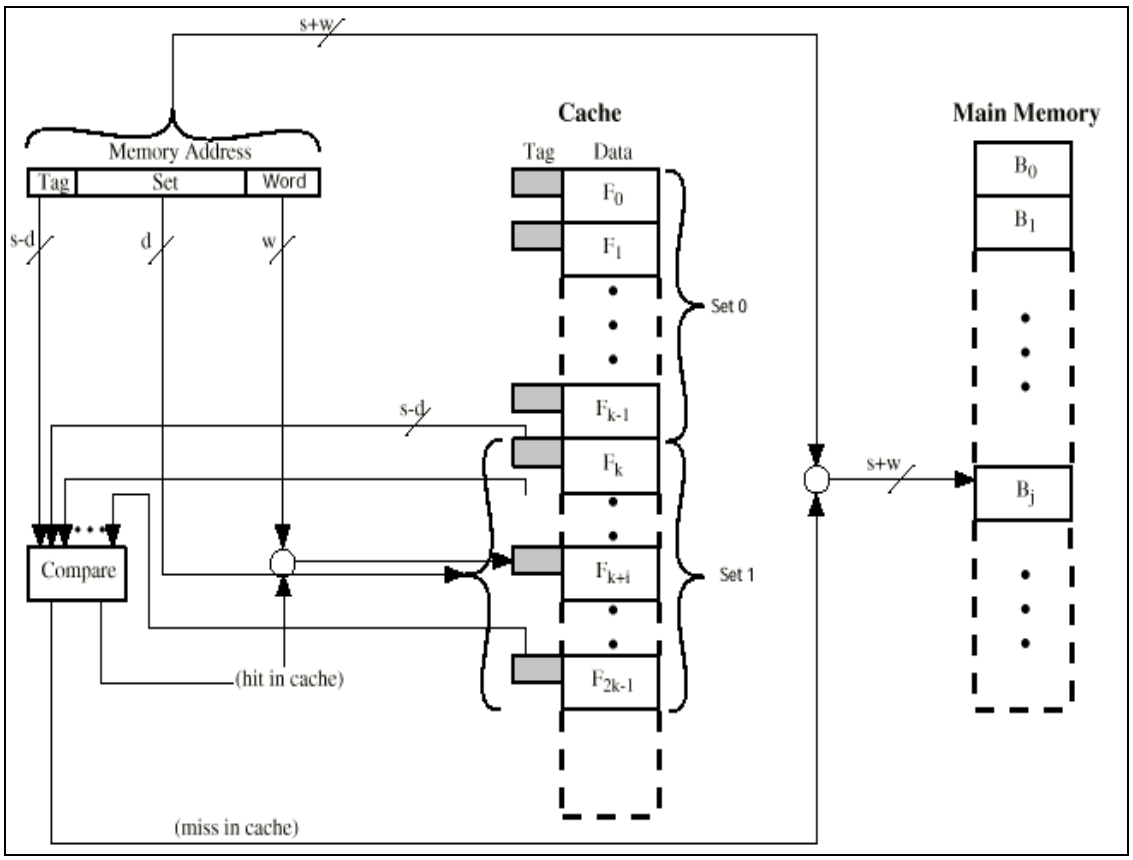


Fig: 4.6 k - Way Set Associative Cache Organization

Set Associative Mapping Address Structure

Tag 9 bit	Set 13 bit	Word 2 bit

•

•Compare tag field to see if we have a hit

•e.g

— Address Tag Data Set number
— 1FF7FFC 1FF 24682468 1FFF
— 02C7FFC 02C 12345678 1FFF

Two Way Set Associative Mapping Example

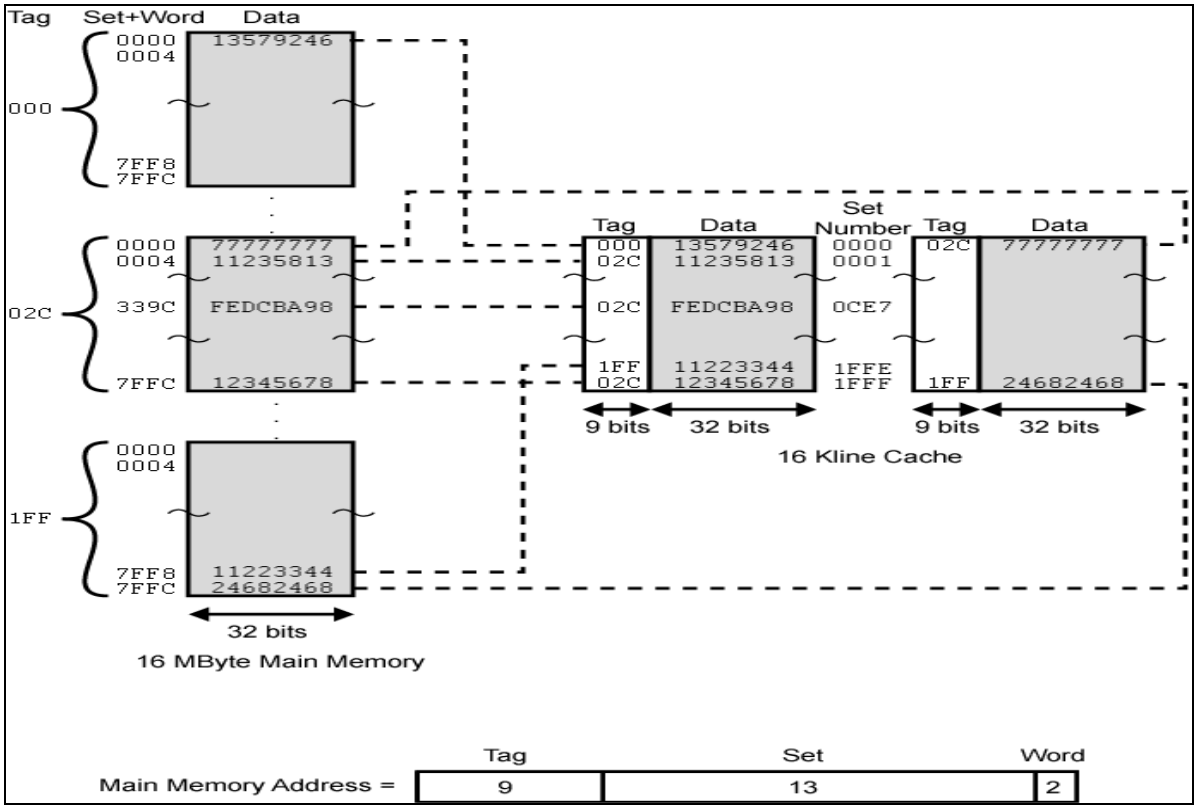


Fig: 4.7 Two Way Set Associative Mapping Example

In the extreme case of:
v = m, k = 1
Set associative mapping reduces to direct mapping
and for:
v = 1, k = m
it reduces to fully associative mapping
2-way organization is the most common set associative organization (v = m/2, k = 2).
4-way organization (v = m/4, k = 4) makes a little improvement for a relatively small additional cost

LECTURE 5

Performance implementation in Cache Memory

The performance of a cache design concerns two related aspects: the *cycle count* and the *hit ratio.* The cycle count refers to the number of basic machine cycles needed for cache access , update and coherence control. The hit ratio determines how effectively the cache can reduce the overall memory-access time.

Cycle Counts

The cache speed is affected by the underlying static or dynamic RAM technology , the cache organization and the cache hit ratios. The total cycle should be predicated with appropriate cache hit ratios.

The cycle counts are not credible unless detailed simulation of all aspects of a memory hierarchy is performed. The write-through or write-back policies also affect the cycle count.

Hit Ratios

The cache hit ratio is affected by the cache size and by the block size in different ways.

When the cache size approaches infinity , a 100% hit ratio should be expected. However , this will never happen because the cache size is always bounded by a limited budget. The initial cache loading and changes in locality also prevent such an ideal performance.

The cache misses are categorized into following three groups:

- **Compulsory**: The very first access to a block cannot be in a cache , so the block must be brought into the cache. These are also called cold start misses.
- **Capacity**: If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur because of blocks being discarded and later retrieved.
- **Conflict**: If the block placement strategy is set associative or direct mapped , conflict misses (in addition to compulsory and capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. These are also called collision misses.

Some techniques to reduce the cache miss rate are described below:

- **Large Block Size**: The simplest way to reduce miss rate is to increase block size. Large block sizes will reduce compulsory misses. Large block sizes may increase conflict misses and even capacity misses if cache is small. So , it is the task of cache designer to choose the block sizes in such a way that all types of cache miss rates minimized.

- **Higher Associativity** : Increased associativity of set associative cache will reduce the cache miss rate. That means that 8-way set assosciative cache will experience less number of cache misses than that of 4-way or 2-way set assosciative cache. But higher way set associative cache will increase the cost of the memory.
- **Use of Victim Cache**: To reduce the conflict misses without impairing clock rate , one small fully assosciative cache called victim cache between a cache and its refill path. This victim cache contains only blocks (victims) that are discarded recently from a cache because of a miss and are checked on a miss to see if they have the desired data before going to the next lower-level memory. If it is found there, the victim block and cache block are swapped.

MCQ

1) Assuming a Main memory of size 32k x 12 , cache memory of size 512 x 12 and block size of 1 word , the addressing relationships using direct mapping would be
    a) tag field – 6 bits , index field – 9 bits
    b) tag field – 9 bits , index field – 6 bits
    c) tag field – 7 bits , index field – 8 bits
    d) none of these
2) Consider the high speed 40 ns memory cache with a successful hit ratio of 80%. The regular memory has an access time of 100 ns. What is the efffective access time for CPU to access memory?
    a) 52 ns
    b) 60 ns
    c) 70 ns
    d) 80 ns
3) A computer with a cache access time of 100 ns , a memory access time of 1000 ns , and a hit ratio of 0.9 produces an average access time of
    a) 250 ns
    b) 200 ns
    c) 190 ns
    d) None of these
4) Associative memory is a
    a) Pointer addressable memory
    b) Very cheap memory
    c) Content addressable memory
    d) Slow memory
5) How many address bits are required for a 512 x 4 memory?
    a) 512
    b) 4
    c) 9
    d) $A_0 – A_6$
6) Assume a system where main memory is of size 16 K × 12and cache memory is of size 1K × 12. For a direct mapping system which statement is correct?
    a) Tag field is 9 bits and index field is 6 bits
    b) Tag field is 4 bits and index field is 10 bits

c) Tag field is 7 bits and index field is 8 bits
d) None of these

7) A direct mapped cache memory with n blocks is nothing but which of the following set associative cache memory organizations?
   a) 0-way set associative
   b) 1-way set associative
   c) 2-way set associative
   d) n-way set associative

8) In which type of memory mapping there will be conflict miss?
   a) Direct mapping
   b) Set associative mapping
   c) Associative mapping
   d) Both (a) & (b).

9) Which is not the property of a memory module?
   a) Inclusion
   b) Consistency
   c) Capability
   d) Locality

10) During transfer of data between the processor and memory we use _____ .
11) a) Cache
12) b) TLB
13) C) Buffers
14) d) Registers

Short Answer Type Questions:

1. Consider the performance of a main memory organization , when a cache miss has occurred as
   i) 4 clock cycles to send the address
   ii) 24 clock cycles for the access time per word
   iii) 4 clock cycles to send a word of data
   Estimate:
   a) The miss penalty for a cache block of 4 words.
   b) The miss penalty for a 4 way interleaved main memory with a cache block of 4 words.
2. What is the cache coherence problem? Suggest one method to solve this problem.
3. What is the limitation of direct mapping method? Explain with example how it can be improved in set-associative mapping.
4. How is a block chosen for replacement in set-associative cache to resolve a cache miss?

Web References/Links:

1.http://www.csie.nuk.edu.tw/~kcf/course/ComputerOrganization/ComputerOrganization_Chapter7_Memory.pdf
2. https://csapp.cs.cmu.edu/2e/ch6-preview.pdf
3. http://www.inf.ed.ac.uk/teaching/courses/inf2c-cs/11-12/lectures/lec12-slides.pdf
4. http://www.cse.iitd.ernet.in/~dheerajb/Memory_cache.pdf
5. http://compsci.hunter.cuny.edu/~sweiss/course_materials/csci360/lecture_notes/chapter_05.pdf
6. https://www.youtube.com/watch?v=6F6NP1lrRpc
7. https://www.youtube.com/watch?v=dGhNzDOhJ2w
8. https://www.youtube.com/watch?v=_kZY4orPQW0
9. https://www.youtube.com/watch?v=-rqGUUOGJoQ
10. https://www.youtube.com/watch?v=WW-_i8LmgvE
11. https://www.youtube.com/watch?v=u_B_pCkAgak
12. https://www.youtube.com/watch?v=PpB5D5yI1QY
13. https://www.youtube.com/watch?v=240GcMH4kUQ
14. https://www.youtube.com/watch?v=3NJWyBRSwng
15. https://www.youtube.com/watch?v=fDjIPN5qAdk

MODULE: 5

MULTIPROCESSOR ARCHITECTURE

LECTURE: 1

Introduction to parallel Architecture:

Parallel computing:
Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel"). There are several different forms of parallel computing: bit-level, instruction level, data, and task parallelism. Parallelism has been employed for many years, mainly in high-performance computing, but interest in it has grown lately due to the physical constraints preventing frequency scaling. As power consumption (and consequently heat generation) by computers has become a concern in recent years,parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multi-core processors. Parallel computers can be roughly classified according to the level at which the hardware supports parallelism, with multi-core and multi-processor computers having multiple processing elements within a single machine, while clusters, MPPs, and grids use multiple computers to work on the same task. Specialized parallel computer architectures are sometimes used alongside traditional processors, for accelerating specific tasks. Parallel computer programs are more difficult to write than sequential ones, because concurrency introduces several new classes of potential software bugs, of which race conditions are the most common. Communication and synchronization between the different subtasks are typically some of the greatest obstacles to getting good parallel program performance.

Fig: 5.1 IBM's Blue Gene/P massively parallel supercomputer

Flynn's Classical Taxonomy:

Among mentioned above the one widely used since 1966, is Flynn's Taxonomy. This taxonomy distinguishes multi-processor computer architectures according two independent dimensions of *Instruction stream* and *Data stream*. An instruction stream is sequence of instructions executed by machine. And a data stream is a sequence of data including input, partial or temporary results used by instruction stream. Each of these dimensions can have only one of two possible states: *Single* or *Multiple*. Flynn's classification depends on the distinction between the performance of control unit and the data processing unit rather than its operational and structural interconnections. Following are the four category of Flynn classification and characteristic feature of each of them.

1. Single instruction stream, single data stream (SISD)

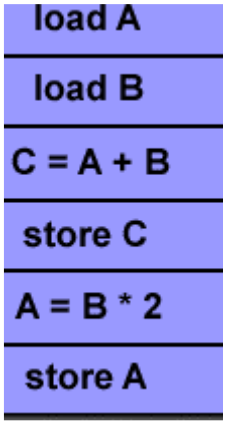


Figure 5.2 Execution of instruction in SISD processors

The figure 1.1 is represents an organization of simple SISD computer having one control unit, one processor unit and single memory unit.

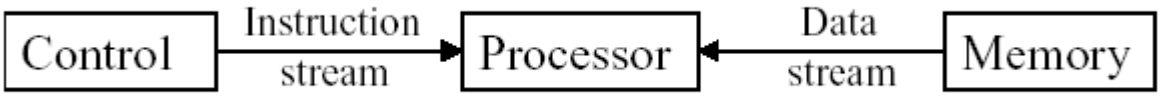


Figure 5.3 SISD processor organizations

- They are also called scalar processor i.e., one instruction at a time and each instruction have only one set of operands.
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle.
- Single data: only one data stream is being used as input during any one clock cycle.

- Deterministic execution.
- Instructions are executed sequentially.
- This is the oldest and until recently, the most prevalent form of computer.
- Examples: most PCs, single CPU workstations and mainframes.

b) Single instruction stream, multiple data stream (SIMD) processors

- A type of parallel computer.
- Single instruction: All processing units execute the same instruction issued by the control unit at any given clock cycle as shown in figure 5.4 where there are multiple processor executing instruction given by one control unit.

  - Multiple data: Each processing unit can operate on a different data element as shown if figure below the processor are connected to shared memory or interconnection network providing multiple data to processing unit.

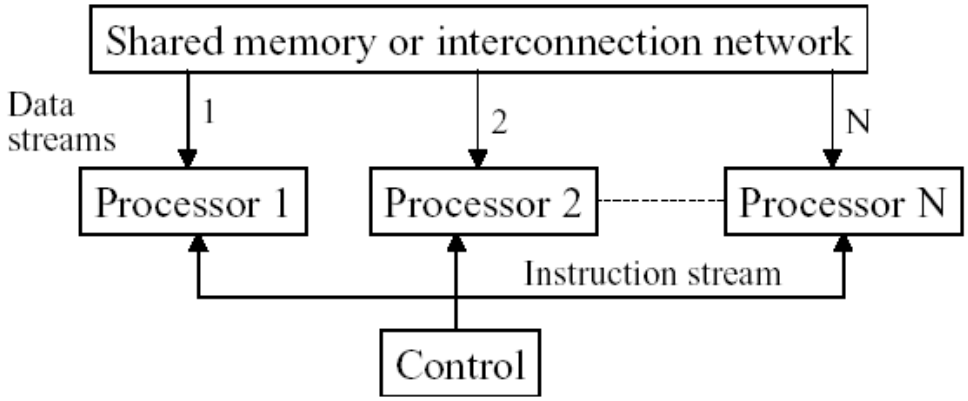


Figure 5.4 SIMD processor organizations

- This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units.
- Thus single instruction is executed by different processing unit on different set of data as shown in figure 5.4
- Best suited for specialized problems characterized by a high degree of regularity, such as image processing and vector computation.
- Synchronous (lockstep) and deterministic execution.

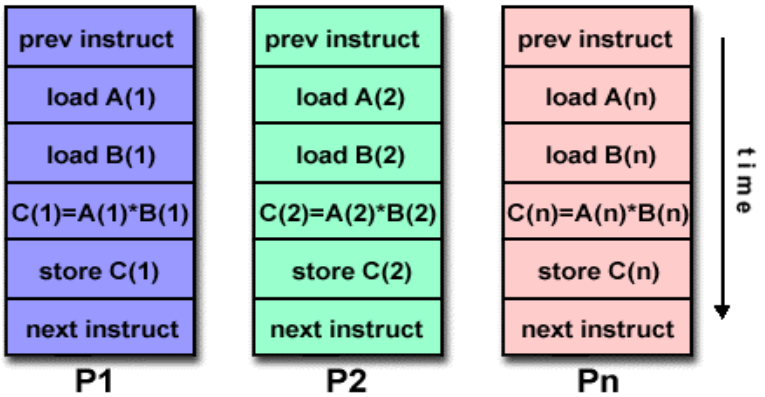


Figure 5.5 Execution of instructions in SIMD processors

c) Multiple instruction streams, single data stream (MISD)

- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction streams as shown in figure 5.6 a single data stream is forwarded to different processing unit which are connected to different control unit and execute instruction given to it by control unit to which it is attached.

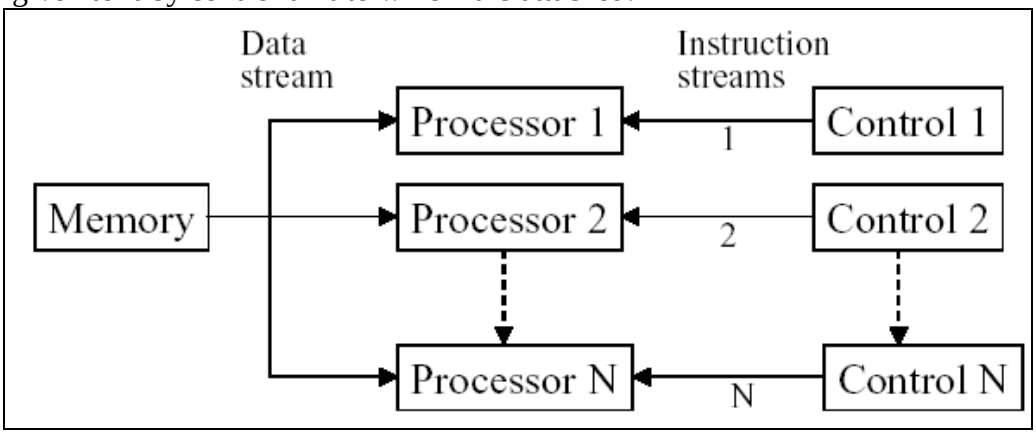


Figure 5.6 MISD processor organizations

- Thus in these computers same data flow through a linear array of processors executing different instruction streams as shown in figure 5.6
- This architecture is also known as systolic arrays for pipelined execution of specific instructions.
- Few actual examples of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon C.mmp computer (1971).
- Some conceivable uses might be:

1. Multiple frequency filters operating on a single signal stream
2. Multiple cryptography algorithms attempting to crack a single coded message.

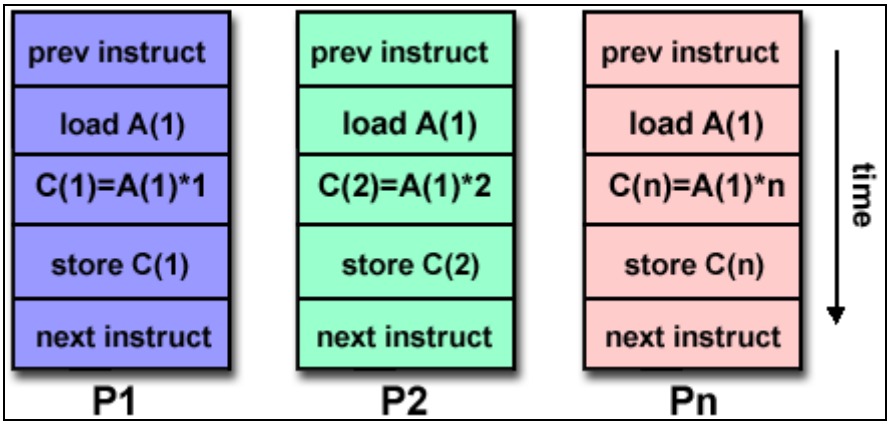


Figure 5.7 Execution of instructions in MISD processors

d) Multiple instruction stream, multiple data stream (MIMD)

- Multiple Instructions: every processor may be executing a different instruction stream

- Multiple Data: every processor may be working with a different data stream as shown in figure 5.8 multiple data stream is provided by shared memory.
- Can be categorized as loosely coupled or tightly coupled depending on sharing of data and control.
- Execution can be synchronous or asynchronous, deterministic or non-deterministic.

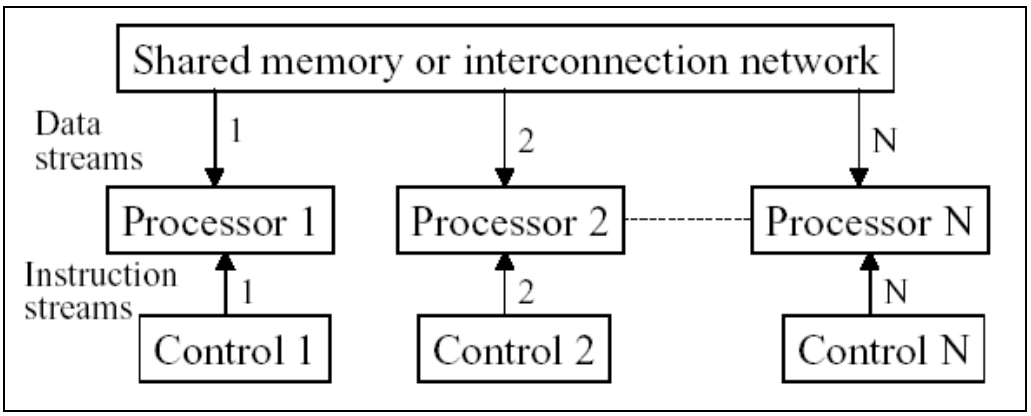


Figure 5.8 MIMD processor organizations

- As shown in figure 5.8 there are different processor each processing different task.
- Examples: most current supercomputers, networked parallel computer "grids" and multi-processor SMP computers - including some types of PCs.

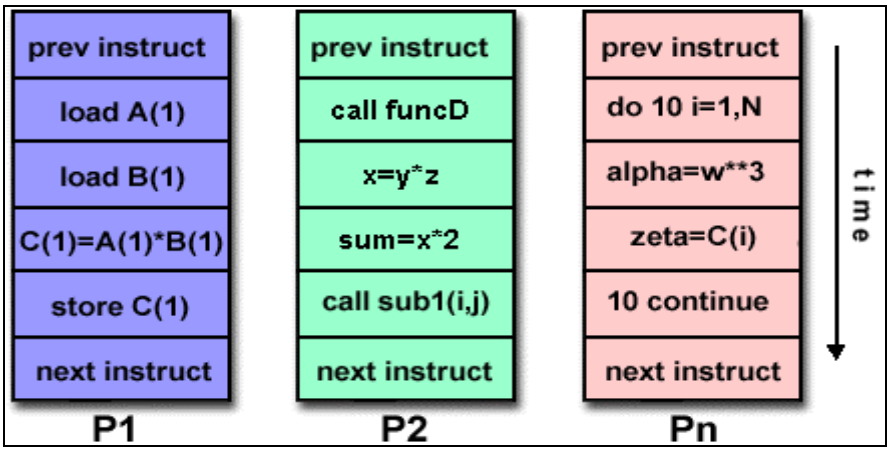


Figure 5.9 Execution of instructions MIMD processors

Here the some popular computer architecture and there types
SISD IBM 701, IBM 1620, IBM 7090, PDP VAX11/ 780
SISD (With multiple functional units) IBM360/91 (3); IBM 370/168 UP
SIMD (Word Slice Processing) Illiac – IV; PEPE
SIMD (Bit Slice processing) STARAN; MPP; DAP
MIMD (Loosely Coupled) IBM 370/168 MP; Univac 1100/80
MIMD (Tightly Coupled) Burroughs- D – 825

LECTURE: 2

Types of parallelism:

Bit-level parallelism

From the advent of very-large-scale integration (VLSI) computer-chip fabrication technology in the 1970s until about 1986, speed-up in computer architecture was driven by doubling computer word size—the amount of information the processor can manipulate per cycle. Increasing the word size reduces the number of instructions the processor must execute to perform an operation on variables whose sizes are greater than the length of the word. For example, where an 8-bit processor must add two 16-bit integers, the processor must first add the 8 lower-order bits from each integer using the standard addition instruction, then add the 8 higher-order bits using an add-with-carry instruction and the carry bit from the lower order addition; thus, an 8-bit processor requires two instructions to complete a single operation, where a 16-bit processor would be able to complete the operation with a single instruction. Historically, 4-bit microprocessors were replaced with 8-bit, then 16-bit, then 32-bit microprocessors. This trend generally came to an end with the introduction of 32-bit processors, which has been a standard in general-purpose computing for two decades. Not until recently (c. 2003–2004), with the advent of x86-64 architectures, have 64-bit processors become commonplace.

Instruction-level parallelism

A computer program is in essence, a stream of instructions executed by a processor. These instructions can be re-ordered and combined into groups which are then executed in parallel without changing the result of the program. This is known as instruction-level parallelism. Advances in instruction-level parallelism dominated computer architecture from the mid-1980s until the mid-1990s.

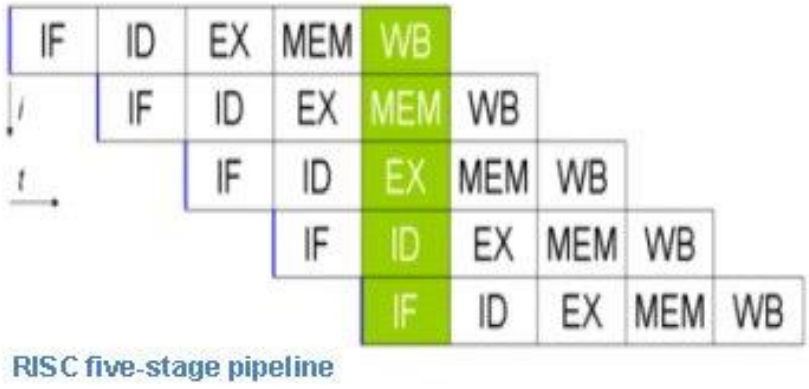


Figure: 5.10 Instruction-level parallelism

Modern processors have multi-stage instruction pipelines. Each stage in the pipeline corresponds to a different action the processor performs on that instruction in that stage; a processor with an N-stage pipeline can have up to N different instructions at different stages of completion. The canonical example of a pipelined processor is a RISC processor, with five

stages: instruction fetch, decode, execute, memory access, and write back. The Pentium 4 processor had a 35-stage pipeline.
A five-stage pipelined superscalar processor, capable of issuing two instructions per cycle. It can have two instructions in each stage of the pipeline, for a total of up to 10 instructions (shown in green) being simultaneously executed. In addition to instruction-level parallelism from pipelining, some processors can issue more than one instruction at a time. These are known as superscalar processors. Instructions can be grouped together only if there is no data dependency between them. Scoreboarding and the Tomasulo algorithm (which is similar to scoreboarding but makes use of register renaming) are two of the most common techniques for implementing out-of-order execution and instruction-level parallelism.

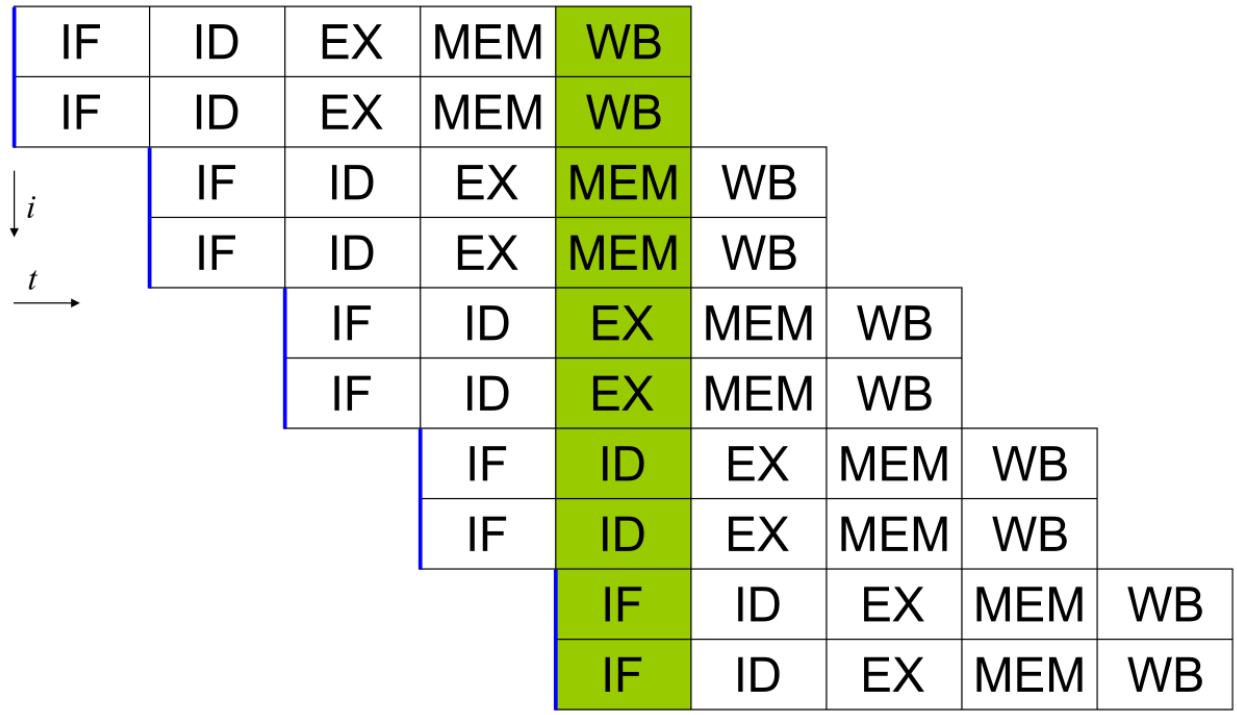


Figure: 5.11 a five-stage pipelined superscalar processor.

.

Classes of parallel computers

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism. This classification is broadly analogous to the distance between basic computing nodes. These are not mutually exclusive; for example, clusters of symmetric multiprocessors are relatively common.

Multicore computing
A multicore processor is a processor that includes multiple execution units ("cores") on the same chip. These processors differ from superscalar processors, which can issue multiple instructions per cycle from one instruction stream (thread); in contrast, a multicore processor can issue multiple instructions per cycle from multiple instruction streams. IBM's Cell microprocessor, designed for use in the Sony PlayStation 3, is another prominent multicore processor. Each core in a multicore processor can potentially be superscalar as well—that is,

on every cycle, each core can issue multiple instructions from one instruction stream. Simultaneous multithreading (of which Intel's Hyper Threading is the best known) was an early form of pseudo-multicoreism. A processor capable of simultaneous multi- threading has only one execution unit ("core"), but when that execution unit is idling (such as during a cache miss), it uses that execution unit to process a second thread.

Symmetric multiprocessing
A symmetric multiprocessor (SMP) is a computer system with multiple identical processors that share memory and connect via a bus. Bus contention prevents bus architectures from scaling. As a result, SMPs generally do not comprise more than 32 processors. "Because of the small size of the processors and the significant reduction in the requirements for bus bandwidth achieved by large caches, such symmetric multiprocessors are extremely cost-effective, provided that a sufficient amount of memory bandwidth exists."

Distributed computing
A distributed computer (also known as a distributed memory multiprocessor) is a distributed memory computer system in which the processing elements are connected by a network. Distributed computers are highly scalable.

Cluster computing
A cluster is a group of loosely coupled computers that work together closely, so that in some respects they can be regarded as a single computer. Clusters are composed of multiple standalone machines connected by a network. While machines in a cluster do not have to be symmetric, load balancing is more difficult if they are not. The most common type of cluster is the Beowulf cluster, which is a cluster implemented on multiple identical commercial off-the-shelf computers connected with a TCP/IP Ethernet local area network. Beowulf technology was originally developed by Thomas Sterling and Donald Becker. The vast majority of the TOP500 supercomputers are clusters.

Massive parallel processing
A massively parallel processor (MPP) is a single computer with many networked processors. MPPs have many of the same characteristics as clusters, but MPPs have specialized interconnect networks (whereas clusters use commodity hardware for networking). MPPs also tend to be larger than clusters, typically having "far more" than 100 processors. In a MPP, "each CPU contains its own memory and copy of the operating system and application. Each subsystem communicates with the others via a high-speed interconnect. A cabinet from Blue Gene/L, ranked as the fourth fastest supercomputer in the world according to the 11/2008 TOP500 rankings. Blue Gene/L is a massively parallel processor. Blue Gene/L, the fifth fastest supercomputer in the world according to the June 2009 TOP500 ranking, is a MPP

Grid computing
Distributed computing is the most distributed form of parallel computing. It makes use of computers communicating over the Internet to work on a given problem. Because of the low bandwidth and extremely high latency available on the Internet, distributed computing typically deals only with embarrassingly parallel problems. Most grid computing applications use middleware, software that sits between the operating system and the application to manage network resources and standardize the software interface. The most common distributed computing middleware is the Berkeley Open Infrastructure for Network

Computing (BOINC). Often, distributed computing software makes use of "spare cycles", performing computations at times when a computer is idling.

LECTURE: 3

Multiprocessor:

A multiprocessor system is a computer system comprising of two or more processor. An interconnection network links this processor. The primary objective of multiprocessor system is to enhance the performance by means of parallel processing. It falls under MIMD architecture.

Besides providing high performance, the multiprocessor also offers the following benefits:

1. Fault tolerance and graceful degradation.
2. Scalability and modular growth.

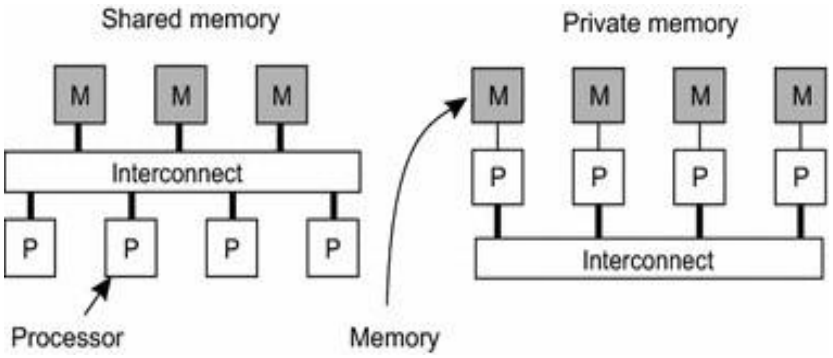


Figure 5.12 Multiprocessor systems.

Classification:

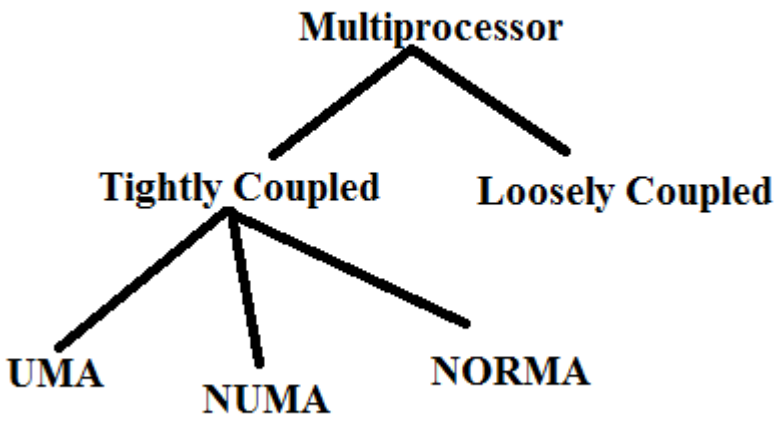


Figure 5.13 Classification of Multiprocessor

Tightly Coupled Multiprocessor System: In tightly coupled multiprocessor; the multiple processor share information by a common memory (Global Memory).Hence, this type is also known as shared memory multiprocessor system. Beside sharing the global memory dedicated to its which cannot be accessed by other processors in the system.

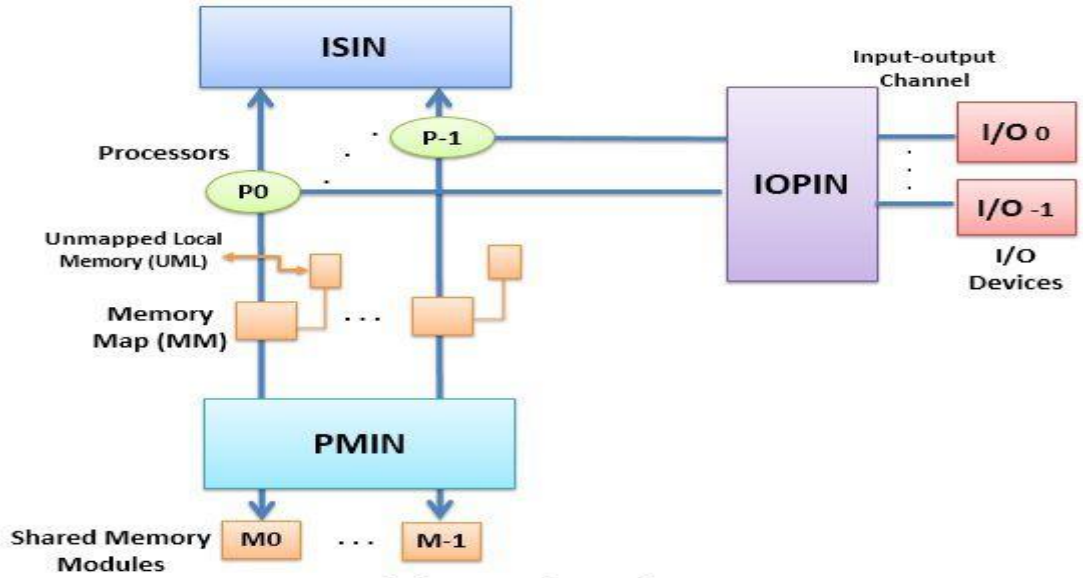


Figure: 5.14

Loosely Coupled Multiprocessor System: In loosely coupled multiprocessor system memory is not shared and each processor has its own memory. This type of a system is known as distributed memory multiprocessor system. The information is exchanged network by a common message passing protocol.

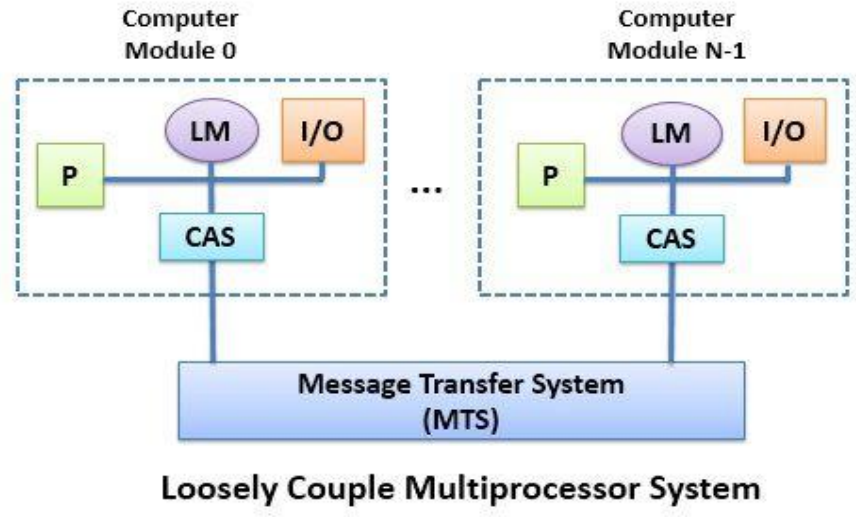


Figure: 5.15 Loosely Coupled Multiprocessor System

Uniform Memory Access:

Uniform memory access (UMA) is a shared memory architecture used in parallel computers. All the processors in the UMA model share the physical memory uniformly. In UMA architecture, access time to a memory location is independent of which processor makes the request or which memory chip contains the transferred data. Uniform memory access computer architectures are often contrasted with non-uniform memory access (NUMA) architectures. In the UMA architecture, each processor may use a private cache. Peripherals are also shared in some fashion. The UMA model is suitable for general purpose and time sharing applications by multiple users. It can be used to speed up the execution of a single large program in time critical applications.

In a uniform memory access system the access time of memory is equal for all processor. A symmetric multiprocessor is UMA multiprocessor system with identical processors, equally capable of performing similar function in a identical manner. All the processors have equal access time for the memory and I/O resources.

Types of UMA architectures:

1. UMA using bus-based symmetric multiprocessing (SMP) architectures.
2. UMA using crossbar switches.
3. UMA using multistage interconnection networks.

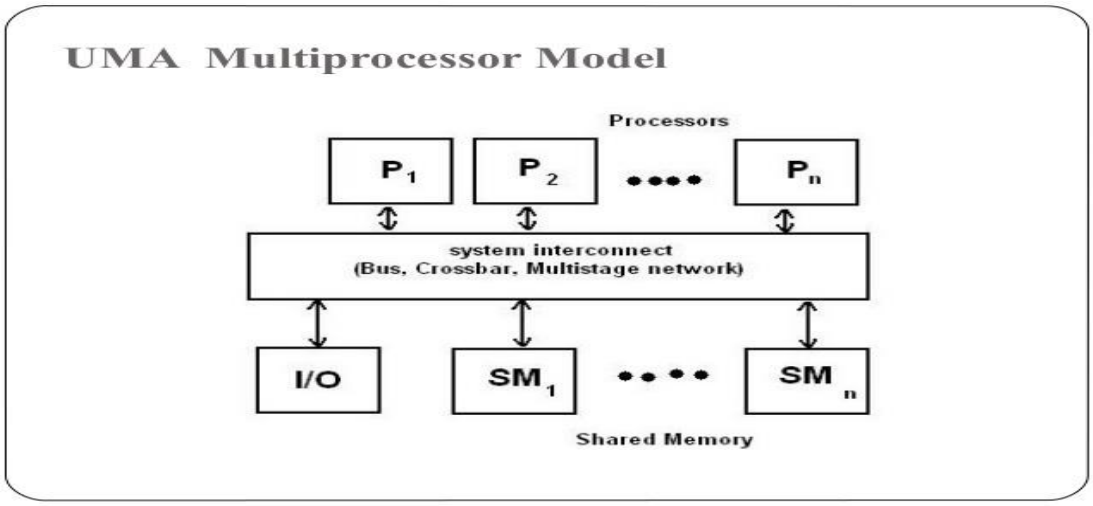


Figure: 5.16 UMA architectures

LECTURE: 4

Non-Uniform Memory Access:

Non-uniform memory access (NUMA) is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to the processor. Under NUMA, a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors). The benefits of NUMA are limited to particular workloads, notably on servers where the data are often associated strongly with certain tasks or users.

NUMA architectures logically follow in scaling from symmetric multiprocessing (SMP) architectures. They were developed commercially during the 1990s by Burroughs (later Unisys), Convex Computer (later Hewlett-Packard), Honeywell Information Systems Italy (HISI) (later Groupe Bull), Silicon Graphics (later Silicon Graphics International), Sequent Computer Systems (later IBM), Data General (later EMC), and Digital (later Compaq, now HP). Techniques developed by these companies later featured in a variety of Unix-like operating systems, and to an extent in Windows NT. The first commercial implementation of a NUMA-based UNIX system was the Symmetrical Multi Processing XPS-100 family of servers, designed by Dan Gielan of VAST Corporation for Honeywell Information Systems Italy.

Modern CPUs operate considerably faster than the main memory they use. In the early days of computing and data processing, the CPU generally ran slower than its own memory. The performance lines of processors and memory crossed in the 1960s with the advent of the first supercomputers. Since then, CPUs increasingly have found themselves "starved for data" and having to stall while waiting for data to arrive from memory. Many supercomputer designs of the 1980s and 1990s focused on providing high-speed memory access as opposed to faster processors, allowing the computers to work on large data sets at speeds other systems could not approach.

Limiting the number of memory accesses provided the key to extracting high performance from a modern computer. For commodity processors, this meant installing an ever-increasing amount of high-speed cache memory and using increasingly sophisticated algorithms to avoid cache misses. But the dramatic increase in size of the operating systems and of the applications run on them has generally overwhelmed these cache-processing improvements. Multi-processor systems without NUMA make the problem considerably worse. Now a system can starve several processors at the same time, notably because only one processor can access the computer's memory at a time. NUMA attempts to address this problem by providing separate memory for each processor, avoiding the performance hit when several processors attempt to address the same memory. For problems involving spread data(common for servers and similar applications), NUMA can improve the performance over a single shared memory by a factor of roughly the number of processors (or separate memory banks). Another approach to addressing this problem, utilized mainly by non-NUMA systems, is the multi-channel memory architecture; multiple memory channels are increasing the number of simultaneous memory accesses.

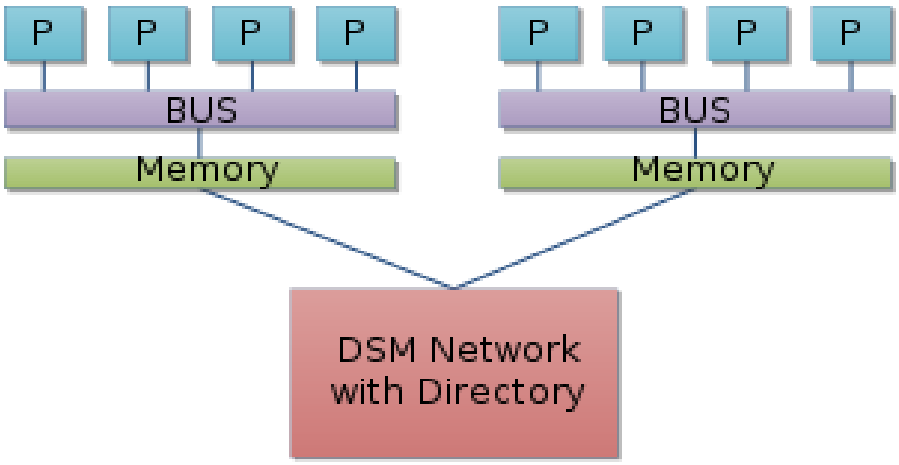


Figure: 5.17 Architecture of a NUMA system.

No-Remote Memory Access

No Remote Memory Access (NORMA) is a computer memory architecture for multiprocessor system.

In NORMA architecture, the address space globally is not unique and the memory is not globally accessible by the processor.

Accesses to remote memory modules are only indirectly possible by message through the interconnection network to other processors, which in turn possibly deliver the desired data in a reply message.

Two categories of parallel computers are discussed below namely shared common memory or unshared distributed memory.

Shared memory multiprocessors

Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.

- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Shared memory machines can be divided into two main classes based upon memory access times: ***UMA, NUMA and COMA.***

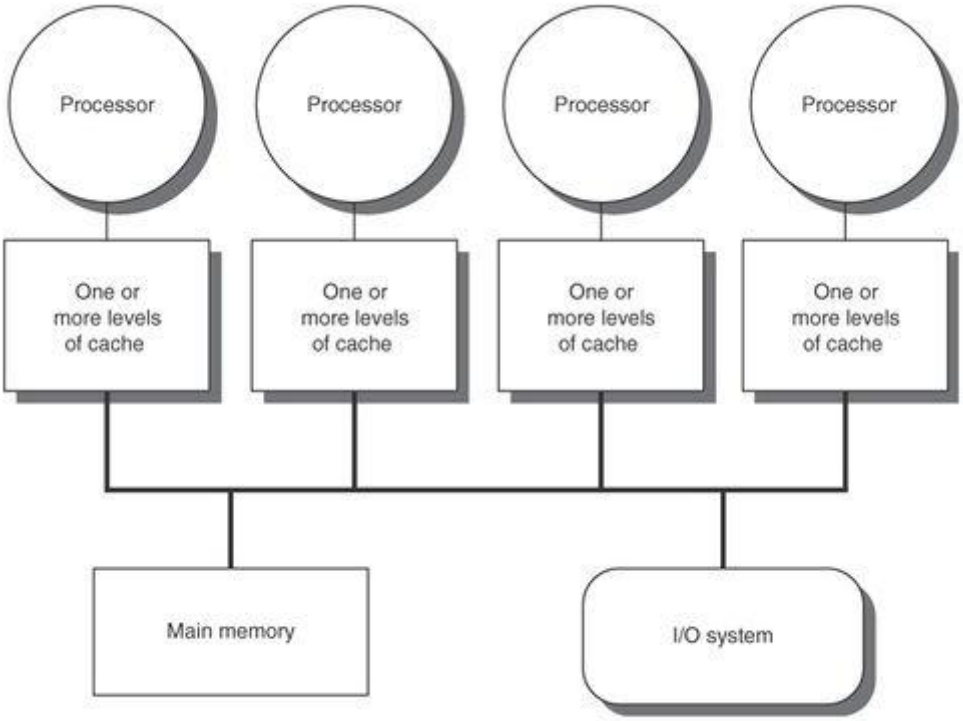


Figure: 5.18 Shared memory multiprocessors

Advantages:

- Global address space provides a user-friendly programming perspective to memory.
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs.

Disadvantages:

- Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-

LECTURE: 5

Parallel random-access machine:

In computer science, a **parallel random-access machine** (**PRAM**) is a shared-memory abstract machine. As its name indicates, the PRAM was intended as the parallel-computing analogy to the random-access machine (RAM). In the same way that the RAM is used by sequential-algorithm designers to model algorithmic performance (such as time complexity), the PRAM is used by parallel-algorithm designers to model parallel algorithmic performance (such as time complexity, where the number of processors assumed is typically also stated). Similar to the way in which the RAM model neglects practical issues, such as access time to cache memory versus main memory, the PRAM model neglects such issues as synchronization and communication, but provides any (problem-size-dependent) number of processors. Algorithm cost, for instance, is estimated using two parameters O (time) and O (time × processor_number).

Random Access Machine is a favourite model of a sequential computer. Its main features are:

1. Computation unit with a user defined program.
2. Read-only input tape and write-only output tape.
3. **Unbounded** number of local memory cells.
4. Each memory cell is capable of holding an integer of unbounded size.
5. Instruction set includes operations for moving data between memory cells, comparisons and conditional branches, and simple arithmetic operations.
6. Execution starts with the first instruction and ends when a HALT instruction is executed.
7. All operations take unit time regardless of the lengths of operands.
8. **Time complexity** = the number of instructions executed.
9. **Space complexity** = the number of memory cells accessed.

Parallel Random Access Machines (PRAM) is a model, which is considered for most of the parallel algorithms. Here, multiple processors are attached to a single block of memory. A PRAM model contains −

- A set of similar type of processors.
- All the processors share a common memory unit. Processors can communicate among themselves through the shared memory only.
- A memory access unit (MAU) connects the processors with the single shared memory.

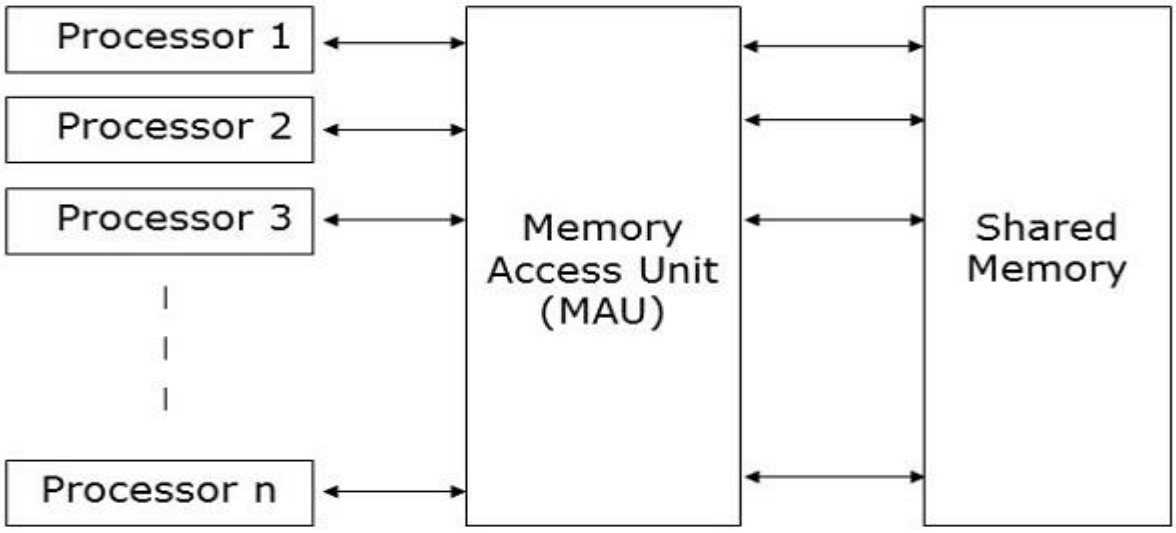


Figure: 5.19 PRAM

Here, **n** number of processors can perform independent operations on **n** number of data in a particular unit of time. This may result in simultaneous access of same memory location by different processors.

To solve this problem, the following constraints have been enforced on PRAM model −

- **Exclusive Read Exclusive Write (EREW)** − Here no two processors are allowed to read from or write to the same memory location at the same time.
- **Exclusive Read Concurrent Write (ERCW)** − Here no two processors are allowed to read from the same memory location at the same time, but are allowed to write to the same memory location at the same time.
- **Concurrent Read Exclusive Write (CREW)** − Here all the processors are allowed to read from the same memory location at the same time, but are not allowed to write to the same memory location at the same time.
- **Concurrent Read Concurrent Write (CRCW)** − All the processors are allowed to read from or write to the same memory location at the same time.

There are many methods to implement the PRAM model, but the most prominent ones are −

- Shared memory model.
- Message passing model.
- Data parallel model.

LECTURE: 6

Shared Memory Model:

Shared memory emphasizes on control parallelism than on data parallelism. In the shared memory model, multiple processes execute on different processors independently, but they share a common memory space. Due to any processor activity, if there is any change in any memory location, it is visible to the rest of the processors.

As multiple processors access the same memory location, it may happen that at any particular point of time, more than one processor is accessing the same memory location. Suppose one is reading that location and the other is writing on that location. It may create confusion. To avoid this, some control mechanism, like **lock / semaphore,** is implemented to ensure mutual exclusion.

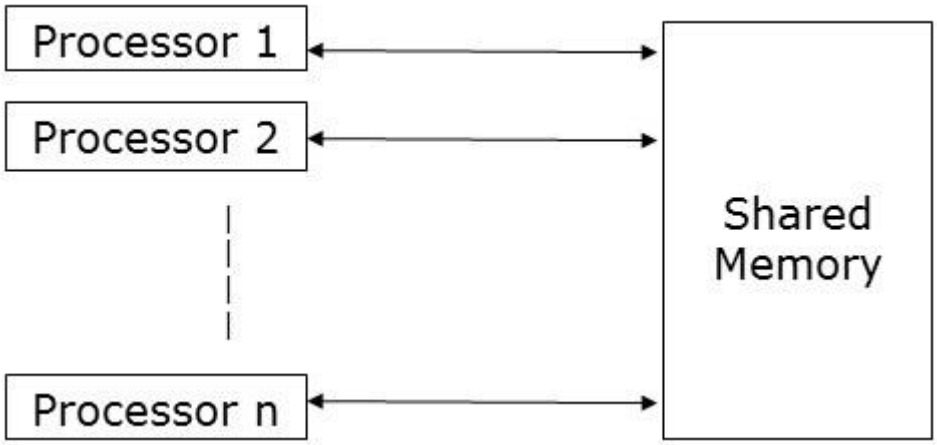


Figure: 5.20 Shared Memory.

Shared memory programming has been implemented in the following −

- **Thread libraries** − The thread library allows multiple threads of control that run concurrently in the same memory location. Thread library provides an interface that supports multithreading through a library of subroutine. It contains subroutines for
    - Creating and destroying threads.
    - Scheduling execution of thread.
    - Passing data and message between threads.
    - Saving and restoring thread contexts.

Examples of thread libraries include – Solaris-TM threads for Solaris, POSIX threads as implemented in Linux, Win32 threads available in Windows NT and Windows 2000, and Java-TM threads as part of the standard Java-TM Development Kit (JDK).

- **Distributed Shared Memory (DSM) Systems** − DSM systems create an abstraction of shared memory on loosely coupled architecture in order to implement shared

memory programming without hardware support. They implement standard libraries and use the advanced user-level memory management features present in modern operating systems.

- **Program Annotation Packages** − This is implemented on the architectures having uniform memory access characteristics. The most notable example of program annotation packages is OpenMP. OpenMP implements functional parallelism. It mainly focuses on parallelization of loops.

The concept of shared memory provides a low-level control of shared memory system, but it tends to be tedious and erroneous. It is more applicable for system programming than application programming.

Merits of Shared Memory Programming.

- Global address space gives a user-friendly programming approach to memory.
- Due to the closeness of memory to CPU, data sharing among processes is fast and uniform.
- There is no need to specify distinctly the communication of data among processes.
- Process-communication overhead is negligible.
- It is very easy to learn.

Demerits of Shared Memory Programming.

- It is not portable.
- Managing data locality is very difficult.

LECTURE: 7

Interconnection Network:

An **interconnection network** in a parallel machine transfers information from any source node to any desired destination node. This task should be completed with as small latency as possible. It should allow a large number of such transfers to take place concurrently. Moreover, it should be inexpensive as compared to the cost of the rest of the machine.

The network is composed of links and switches, which helps to send the information from the source node to the destination node. A network is specified by its topology, routing algorithm, switching strategy, and flow control mechanism.

Organizational Structure

Interconnection networks are composed of following three basic components −

- **Links** − A link is a cable of one or more optical fibers or electrical wires with a connector at each end attached to a switch or network interface port. Through this, an analog signal is transmitted from one end, received at the other to obtain the original digital information stream.
- **Switches** − A switch is composed of a set of input and output ports, an internal "cross-bar" connecting all input to all output, internal buffering, and control logic to effect the input-output connection at each point in time. Generally, the number of input ports is equal to the number of output ports.
- **Network Interfaces** − The network interface behaves quite differently than switch nodes and may be connected via special links. The network interface formats the packets and constructs the routing and control information. It may have input and output buffering, compared to a switch. It may perform end-to-end error checking and flow control. Hence, its cost is influenced by its processing complexity, storage capacity, and number of ports.

Classification of Interconnection network:

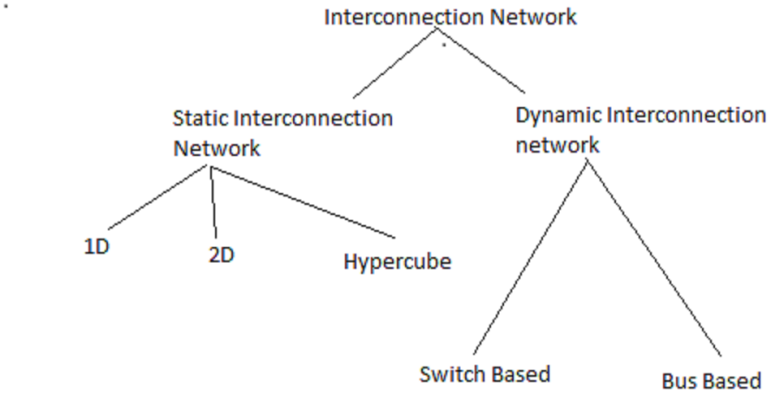
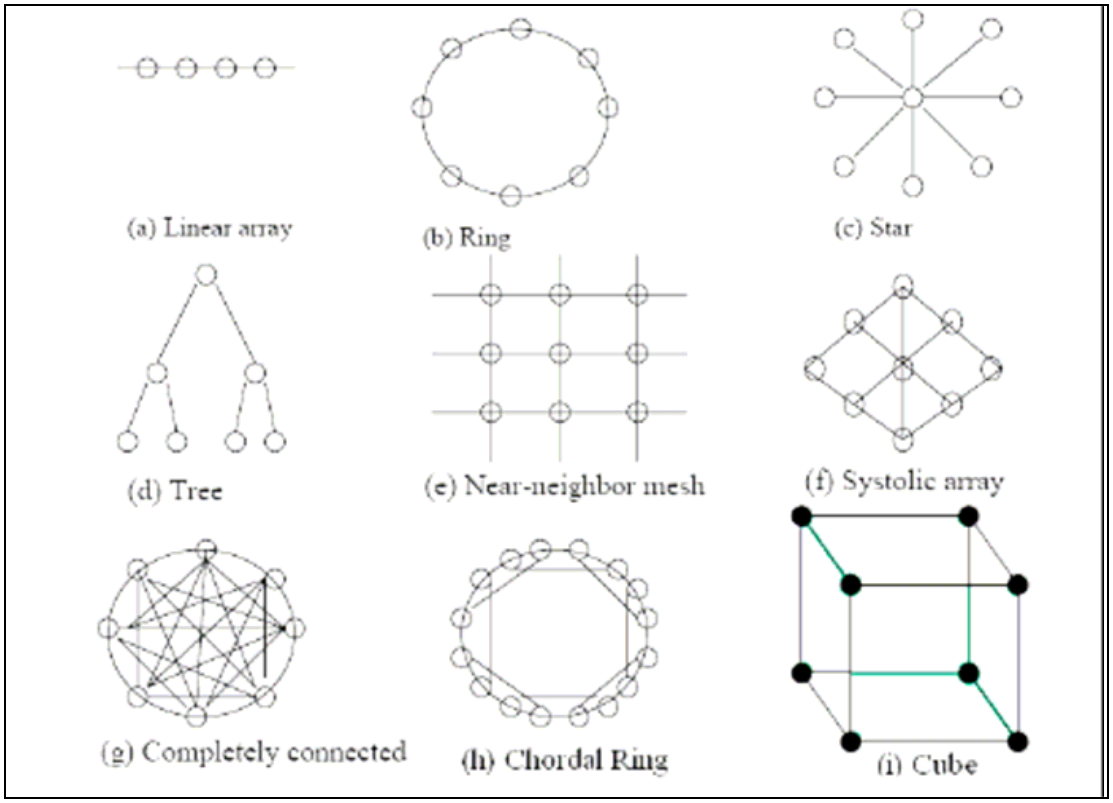


Figure: 5.21 Classification of Interconnection network

Static Interconnection Networks:
Static interconnection networks for elements of parallel systems (ex. processors, memories) are based on fixed connections that can't be modified without a physical re-designing of a system. Static interconnection networks can have many structures such as a linear structure (pipeline), a matrix, a ring, a torus, a **complete connection** structure, a tree, a star, a **hyper-cube**.



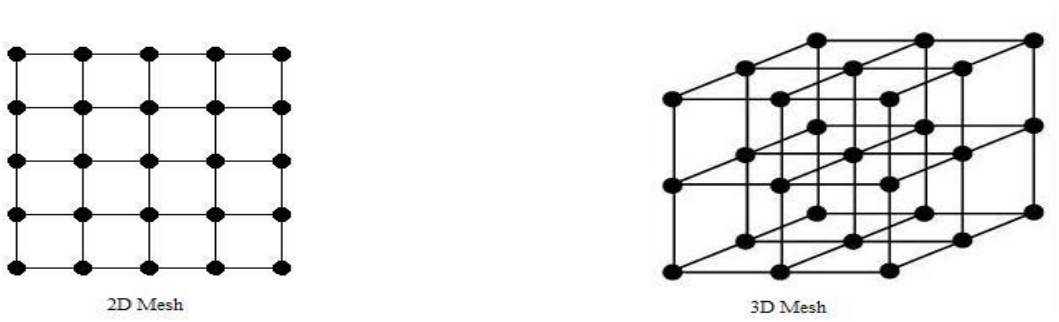


Figure: 5.22 Static interconnection network topologies

Hypercube Interconnection Network:

In a hypercube structure, processors are interconnected in a network, in which connections between processors correspond to edges of a n-dimensional cube. The hypercube structure is very advantageous since it provides a low **network diameter** equal to the degree of the cube. The network diameter is the number of edges between the most distant nodes. . The network diameter determines the number in intermediate transfers that have to be done to send data between the most distant nodes of a network. In this respect the hyper cubes have very good properties, especially for a very large number of constituent nodes. Due to this hyper cubes are popular networks in existing parallel systems.

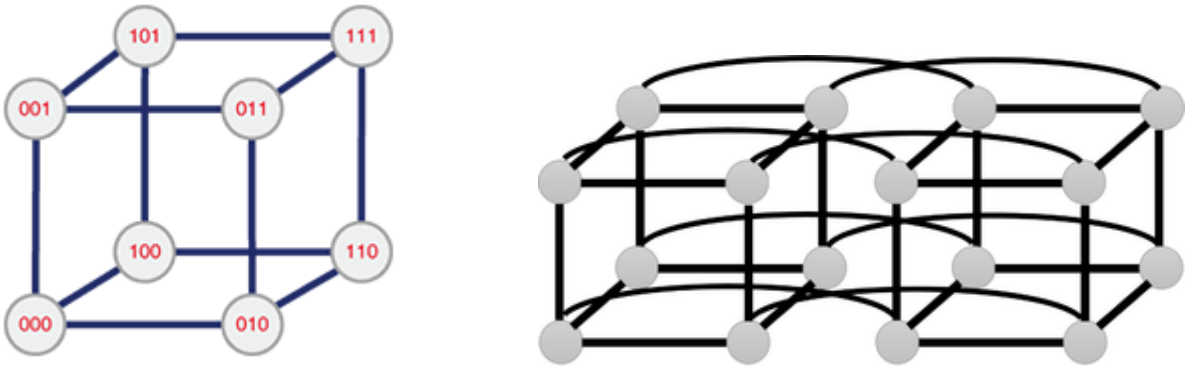


Figure: 5.23 Hypercube Network.

LECTURE: 8

Dynamic Interconnection Networks:

Dynamic interconnection networks between processors enable changing (reconfiguring) of the connection structure in a system. It can be done before or during parallel program execution.

Interconnection Network:

Interconnection networks are composed of switching elements. Topology is the pattern to connect the individual switches to other elements, like processors, memories and other switches. A network allows exchange of data between processors in the parallel system.

- **Direct Connection Networks –** Direct networks have point-to-point connections between neighboring nodes. These networks are static, which means that the point-to-point connections are fixed. Some examples of direct networks are rings, meshes and cubes.

- **Indirect connection networks** − Indirect networks have no fixed neighbors. The communication topology can be changed dynamically based on the application demands. Indirect networks can be subdivided into three parts: bus networks, multistage networks and crossbar switches.

    - **Bus networks –** A bus network is composed of a number of bit lines onto which a number of resources are attached. When busses use the same physical lines for data and addresses, the data and the address lines are time multiplexed. When there are multiple bus-masters attached to the bus, an arbiter is required.

    - **Multistage networks –** A multistage network consists of multiple stages of switches. It is composed of ‘axb’ switches which are connected using a particular inter stage connection pattern (ISC). Small 2x2 switch elements are a common choice for many multistage networks. The number of stages determines the delay of the network. By choosing different inter stage connection patterns, various types of multistage network can be created.

    - **Crossbar switches** − A crossbar switch contains a matrix of simple switch elements that can switch on and off to create or break a connection. Turning on a switch element in the matrix, a connection between a processor and a memory can be made. Crossbar switches are non-blocking, that is all communication permutations can be performed without blocking.

Bus Networks:

- A bus is the simplest type dynamic interconnection networks. It constitutes a common data transfer path for many devices. Depending on the type of implemented transmissions we have **serial busses** and **parallel busses.** The devices connected to a bus can be processors, memories, I/O units, as shown in the figure below.

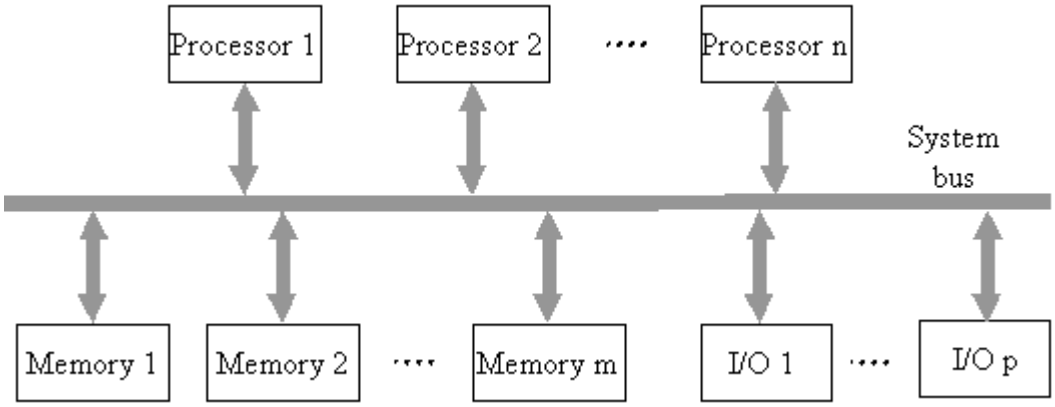


Figure: 5.24 a diagram of a system based on a single bus

Only one device connected to a bus can transmit data. Many devices can receive data. In the last case we speak about a **multicast transmission**. If data are meant for all devices connected to a bus we speak about a **broadcast transmission**. Accessing the bus must be synchronized. It is done with the use of two methods: a **token method** and a **bus arbiter method.** With the token method, a token (a special control message or signal) is circulating between the devices connected to a bus and it gives the right to transmit to the bus to a single device at a time. The bus arbiter receives data transmission requests from the devices connected to a bus. It selects one device according to a selected strategy (ex. using a system of assigned priorities) and sends an acknowledge message (signal) to one of the requesting devices that grants it the transmitting right. After the selected device completes the transmission, it informs the arbiter that can select another request. The receiver (s) address is usually given in the header of the message. Special header values are used for the broadcast and multicasts. All receivers read and decode headers. These devices that are specified in the header, read-in the data transmitted over the bus.

The throughput of the network based on a bus can be increased by the use of a **multi-bus network** shown in the figure below. In this network, processors connected to the busses can transmit data in parallel (one for each bus) and many processors can read data from many busses at a time.

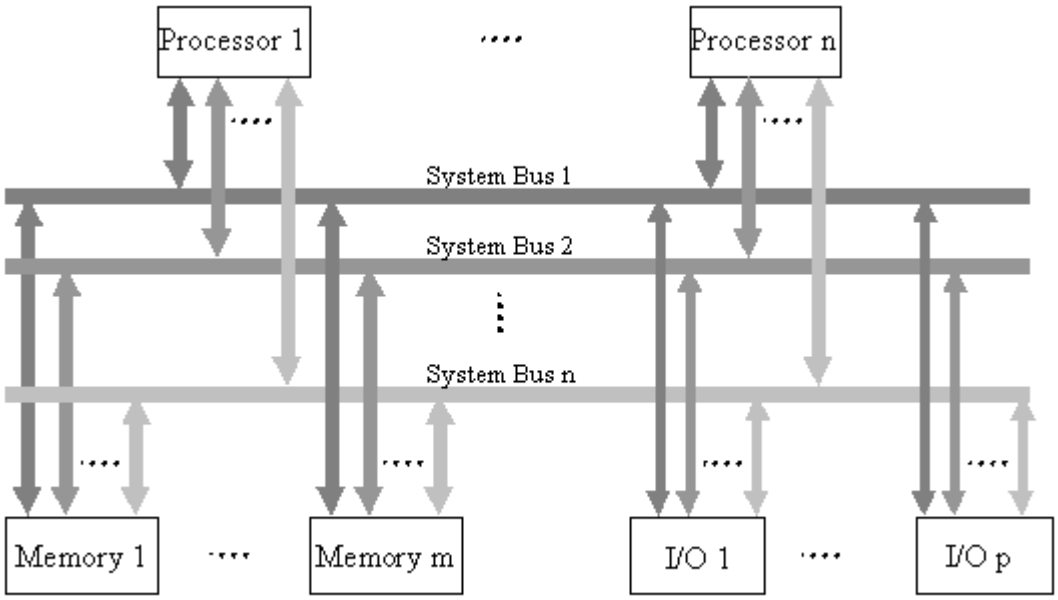


Figure: 5.25 a diagram of a system based on a multi- bus.

LECTURE: 9

Crossbar switches:

A crossbar switch is a circuit that enables many interconnections between elements of a parallel system at a time. A crossbar switch has a number of input and output data pins and a number of control pins. In response to control instructions set to its control input, the crossbar switch implements a stable connection of a determined input with a determined output. The diagrams of a typical crossbar switch are shown in the figure below.

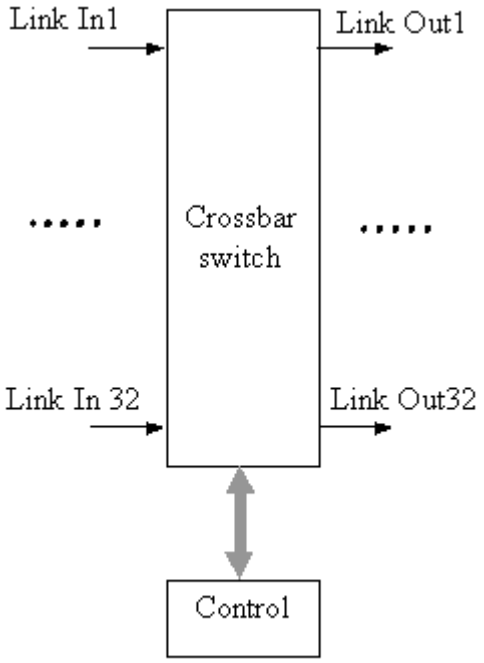


Figure: 5.26 Crossbar Switch, general scheme.

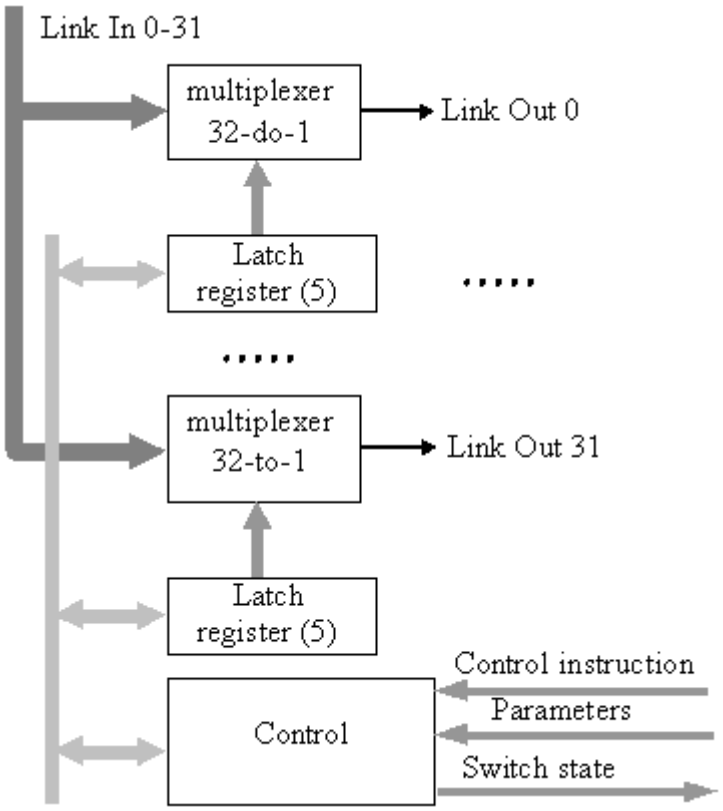


Figure: 5.27 Crossbar switch, internal structure

Control instructions can request reading the state of specified input and output pins i.e. their current connections in a crossbar switch. Crossbar switches are built with the use of multiplexer circuits, controlled by latch registers, which are set by control instructions. Crossbar switches implement direct, single **non-blocking connections**, but on the condition that the necessary input and output pins of the switch are free. The connections between free pins can always be implemented independently on the status of other connections. New connections can be set during data transmissions through other connections. The non-blocking connections are a big advantage of crossbar switches. Some crossbar switches enable broadcast transmissions but in a blocking manner for all other connections. The disadvantage of crossbar switches is that extending their size, in the sense of the number of input/output pins, is costly in terms of hardware. Because of that, crossbar switches are built up to the size of 100 input/output pins. The crossbar switches that contain hundreds of pins are implemented using the technique of multistage interconnection networks that is discussed in the next section of the lecture.

LECTURE: 10

Multistage Interconnection (Omega) Networks:

Multistage connection networks are designed with the use of small elementary crossbar switches (usually they have two inputs) connected in multiple layers. The elementary crossbar switches can implement 4 types of connections: straight, crossed upper broadcast and lower broadcast. All elementary switches are controlled simultaneously. The network like this is an alternative for crossbar switches if we have to switch a large number of connections, over 100. The extension cost for such a network is relatively low.

In such networks, there is no full freedom in implementing arbitrary connections when some connections have already been set in the switch. Because of this property, these networks belong to the category of so called **blocking networks.**

However, if we increase the number of levels of elementary crossbar switches above the number necessary to implement connections for all pairs of inputs and outputs, it is possible to implement all requested connections at the same time but statically, before any communication is started in the switch. It can be achieved at the cost of additional redundant hardware included into the switch. The block diagram of such a network, called the Benes network, is shown in the figure below.

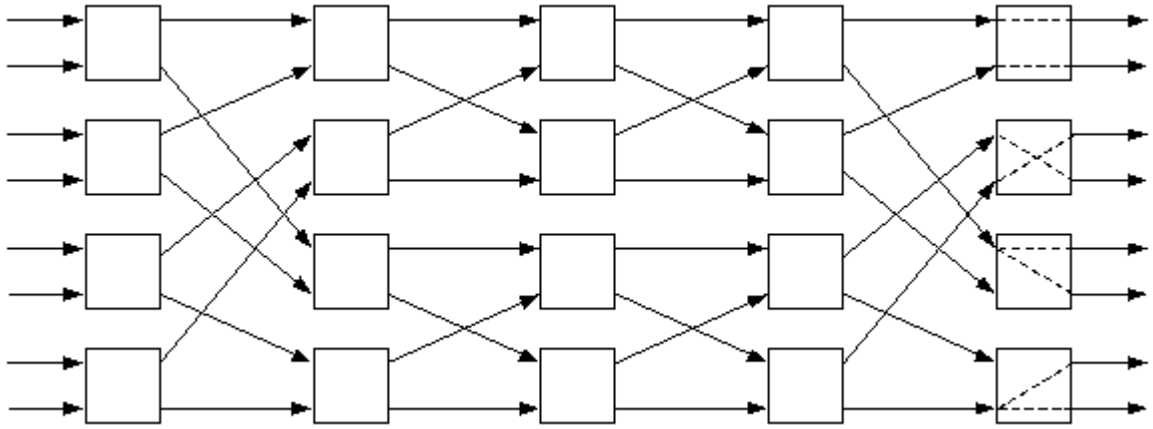


Figure: 5.28 Multistage Connection Network For Parallel Systems.

To obtain non blocking properties of the multistage connection network, the redundancy level in the circuit should be much increased. To build a non blocking multistage network ***n x n***, the elementary two-input switches have to be replaced by 3 layers of switches ***n x m***, ***r x r*** and ***m x n***, where ***m , 2n - 1*** and ***r*** is the number of elementary switches in the layer 1 and 3. Such a switch was designed by a French mathematician Clos and it is called the **Clos network.** This switch is commonly used to build large integrated crossbar switches. The block diagram of the Clos network is shown in the figure below.

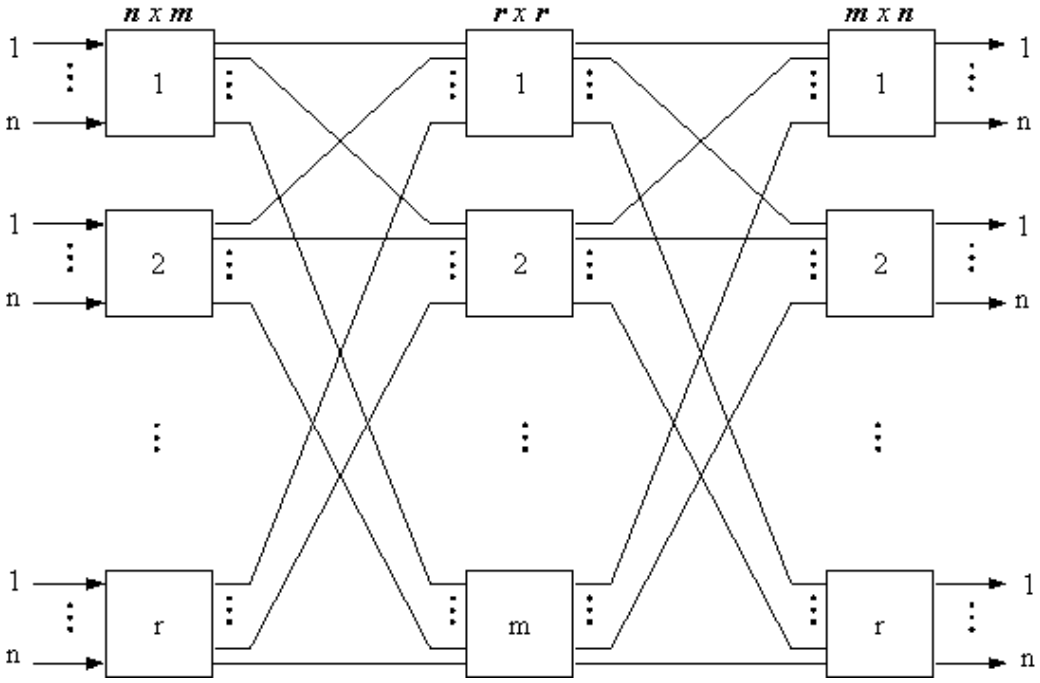


Figure: 5.29 A non-blocking Clos interconnection network

LECTURE: 11

Baseline Network:

Baseline network is one of the important interconnection networks employed in parallel computing systems. Baseline network is a type of permutation network, which connects an equal number of inputs and outputs and realizes a set of permutations. In the Baseline network, the maximum number of allowable permutations is 2n *N/2, where n is the number of switching stages (n = log2N) and each switch has two inputs and two outputs. Fig. 5.30 depict Baseline networks.

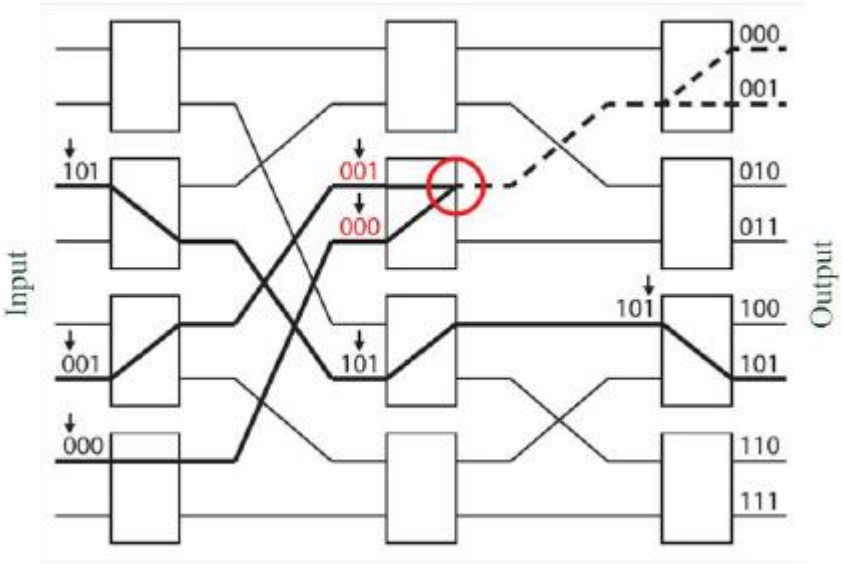


Figure: 5.30 8x8 Baseline network

Butterfly Network:

A **butterfly network** is a computer science technique to link multiple computers into a high-speed computing network. This form of multistage interconnection network topology can be used to connect different nodes in a multiprocessor system. The interconnect network for a shared memory multiprocessor system must have low latency and high bandwidth compared to other network systems, like local area networks (LANs) or internet. Multiprocessor systems must have low latency and high bandwidth for three reasons: (1) Messages are relatively short as most messages consist of coherence protocol requests and responses without data. (2) Messages are generated frequently because each read or write miss generates messages to every node in the system to ensure coherence. Read or write misses occur when the requested data is not in the processor's cache and must

be fetched from either memory or another processor's cache. (3) Messages are generated frequently, therefore rendering it difficult for processors to hide the communication delay.

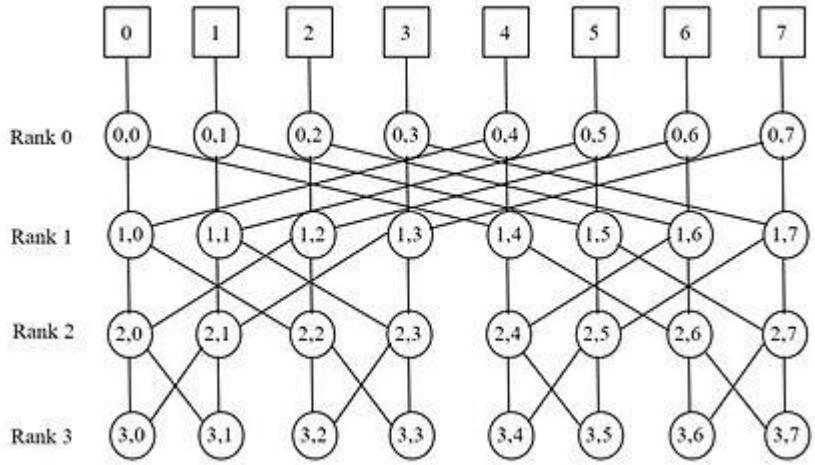


Figure: 5.31: Butterfly Network for 8 processors.

The major components of an interconnect network are:

- Processor Nodes which consist of one or more processors along with their caches, memories and communication assist.
- Switching Nodes (Router) which connect communication assist of different processor nodes in a system. In multistage topologies, higher level switching nodes connect to lower level switching nodes as shown in figure 1, where switching nodes in rank 0 connect to processor nodes directly while switching nodes in rank 1 connect to switching nodes in rank 0.
- Links which are physical wires between two switching nodes (routers). They can be uni-directional or bi-directional.

These multistage networks have lower cost than a cross bar but still obtain lower contention than a bus. The ratio of switching nodes to processor nodes is greater than one in a butterfly network. Such topology where the ratio of switching nodes to processor nodes is greater than one is called an indirect topology.

The network derived its name from connections between nodes in two adjacent ranks (as shown in figure 5.31), which resembles a butterfly. When top and bottom ranks are merged into a single rank, it is called a *Wrapped Butterfly Network*. In figure 5.31, if rank 3 nodes are connected back to respective rank 0 nodes, then it becomes a wrapped butterfly network.

BBN Butterfly, a massive parallel computer built by Bolt, Beranek and Newman in the 1980s, used a butterfly interconnect network. Later in 1990, Cray Research's machine Cray C90, used a butterfly network to communicate between its 16 processors and 1024 memory banks.

Butterfly network building:

For a butterfly network with 'p' processor nodes, there needs to be p ($\log_2 p + 1$) switching nodes. Figure 5.31 shows a network with 8 processor nodes, which means there are 32 switching nodes. It also represents each node as N (rank, column number). For example, node at column 6 in rank 1 is represented as (1, 6) and node at column 2 in rank 0 is represented as (0, 2).

For any 'i' greater than zero, a switching node N (i,j) gets connected to N(i-1, j) and N(i-1, m), where 'm' is obtained by flipping the i^{th} most significant bit of j. For example, consider the node N (1,6): i equals 1 and j equals 6, therefore m is obtained by flipping the first most significant bit of 6.

Variable	Binary representation	Decimal Representation
j	110	6
m	010	2

Table 5.1

As a result, the nodes connected to N (1,6) are :-

N(i,j)	N(i-1,j)	N(i-1,m)
(1,6)	(0,6)	(0,2)

Table 5.2

Thus, N (0,6), N(1,6), N(0,2), N(1,2) form a butterfly pattern. Several butterfly patterns exist in the figure and therefore, this network is called a Butterfly Network.

CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.

- Programmer responsibility for synchronization constructs that insure "correct" access of global memory.
- Expense: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

LECTURE: 12

Distributed Memory:

Like shared memory systems, distributed memory systems vary widely but share a common characteristic. Distributed memory systems require a communication network to connect inter-processor memory.

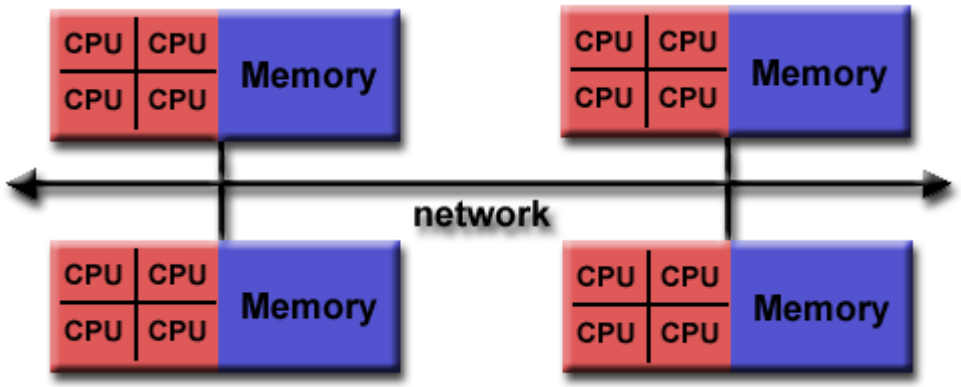


Figure: 5.32 Distributed memory systems

Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.

- Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility

Modern multicomputer use hardware routers to pass message. Based on the interconnection and routers and channel used the multi-computers are divided into generation

I. 1st generation: based on board technology using hypercube architecture and software controlled message switching.

II. 2nd Generation: implemented with mesh connected architecture, hardware message routing and software environment for medium distributed –grained computing.

III. 3rd Generation: fine grained multicomputer like MIT J-Machine.

- The network "fabric" used for data transfer varies widely, though it can be as simple as Ethernet.

Advantages:

- Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
- Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
- Cost effectiveness: can use commodity, off-the-shelf processors and networking.

Disadvantages:

- The programmer is responsible for many of the details associated with data communication between processors.
- It may be difficult to map existing data structures, based on global memory, to this memory organization.
- Non-uniform memory access (NUMA) times.

Multi-core Processor Architecture

A multi-core processor is a single computing component with two or more independent actual processing units (called "cores"), which are units that read and execute program instructions.[1] The instructions are ordinary CPU instructions (such as add, move data, and branch), but the single processor can run multiple instructions on separate cores at the same time, increasing overall speed for programs amenable to parallel computing.[2] Manufacturers typically integrate the cores onto a single integrated circuit die (known as a chip multiprocessor or CMP), or onto multiple dies in a single chip package.

A multi-core processor implements multiprocessing in a single physical package. Designers may couple cores in a multi-core device tightly or loosely. For example, cores may or may not share caches, and they may implement message passing or shared-memory inter-core communication methods. Common network topologies to interconnect cores include bus, ring, two-dimensional mesh, and crossbar. Homogeneous multi-core systems include only identical cores; heterogeneous multi-core systems have cores that are not identical (e.g. big.LITTLE have heterogeneous cores that share the same instruction set, while AMD Accelerated Processing Units have cores that don't even share the same instruction set). Just as with single-processor systems, cores in multi-core systems may implement architectures such as VLIW, superscalar, vector, or multithreading.

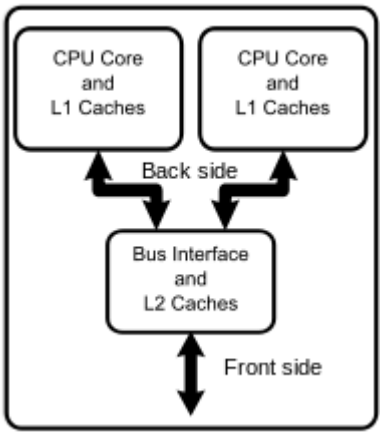


Figure: 5.33 Multi-core Processor Architecture

Multi-core processors are widely used across many application domains, including general-purpose, embedded, network, digital signal processing (DSP), and graphics (GPU).

Case Study 1: An Intel Core

Intel Core is a line of mid-to-high end consumer, workstation, and enthusiast central processing units (CPU) marketed by Intel Corporation. These processors displaced the existing mid-to-high end Pentium processors of the time, moving the Pentium to the entry level, and bumping the Celeron series of processors to low end. Identical or more capable versions of Core processors are also sold as Xeon processors for the server and workstation markets.

Fig: 5.34 INTEL Processor

Core Duo

Intel Core Duo (product code 80539) consists of two cores on one die, a 2 MB L2 cache shared by both cores, and an arbiter bus that controls both L2 cache and FSB (front-side bus) access.

Core 2 Duo

The majority of the desktop and mobile Core 2 processor variants are Core 2 Duo with two processor cores on a single Merom, Conroe, Allendale, Penryn, or Wolfdale chip. These come in a wide range of performance and power consumption, starting with the relatively slow ultra-low- The mobile Core 2 Duo processors with an 'S' prefix in the name are produced in a smaller µFC-BGA 956 package, which allows building more compact laptops.

Within each line, a higher number usually refers to a better performance, which depends largely on core and front-side bus clock frequency and amount of second level cache, which are model-specific. Core 2 Duo processors typically use the full L2 cache of 2, 3, 4, or 6 MB available in the specific stepping of the chip, while versions with the amount of cache reduced during manufacturing are sold for the low-end consumer market as Celeron or Pentium Dual-Core processors. Like those processors, some low-end Core 2 Duo models disable features such as Intel Virtualization Technology.

Core i3

Intel intended the Core i3 as the new low end of the performance processor line from Intel, following the retirement of the Core 2 brand.

The first Core i3 processors were launched on January 7, 2010.

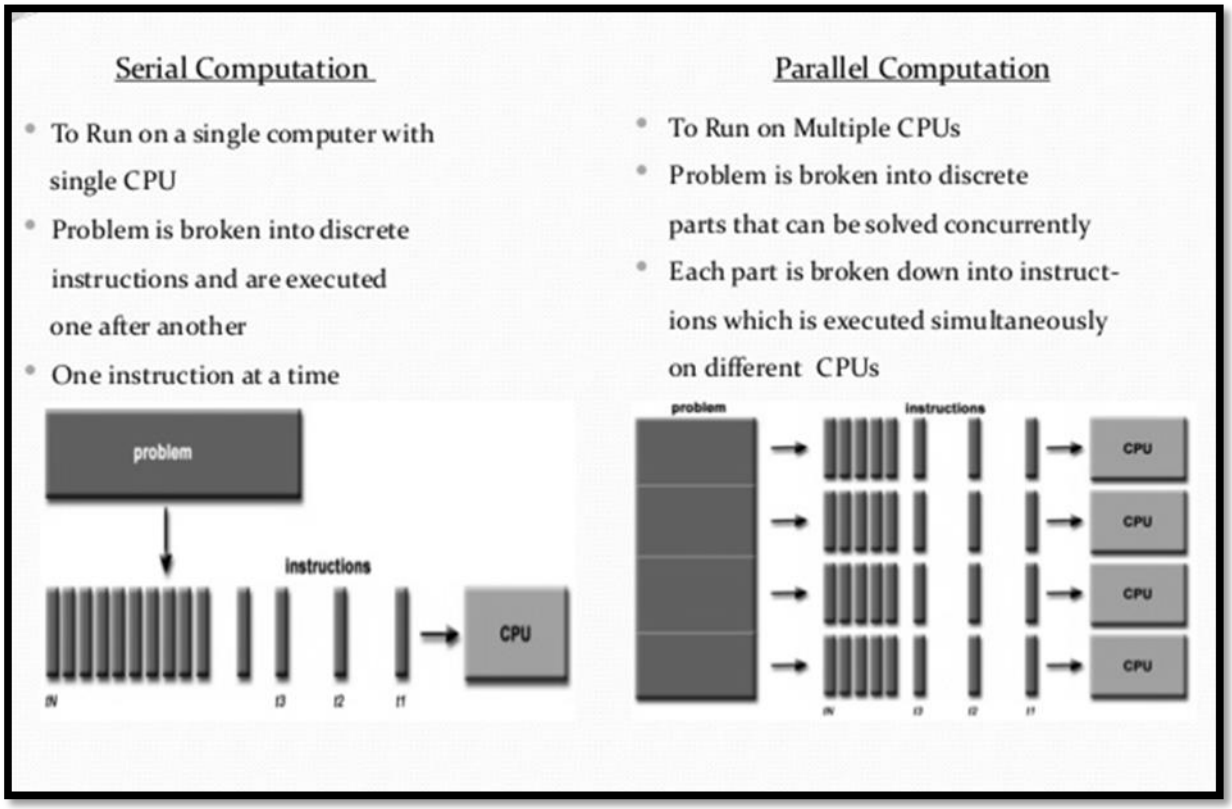
The first Nehalem based Core i3 was Clarkdale-based, with an integrated GPU and two cores.The same processor is also available as Core i5 and Pentium, with slightly different configurations.

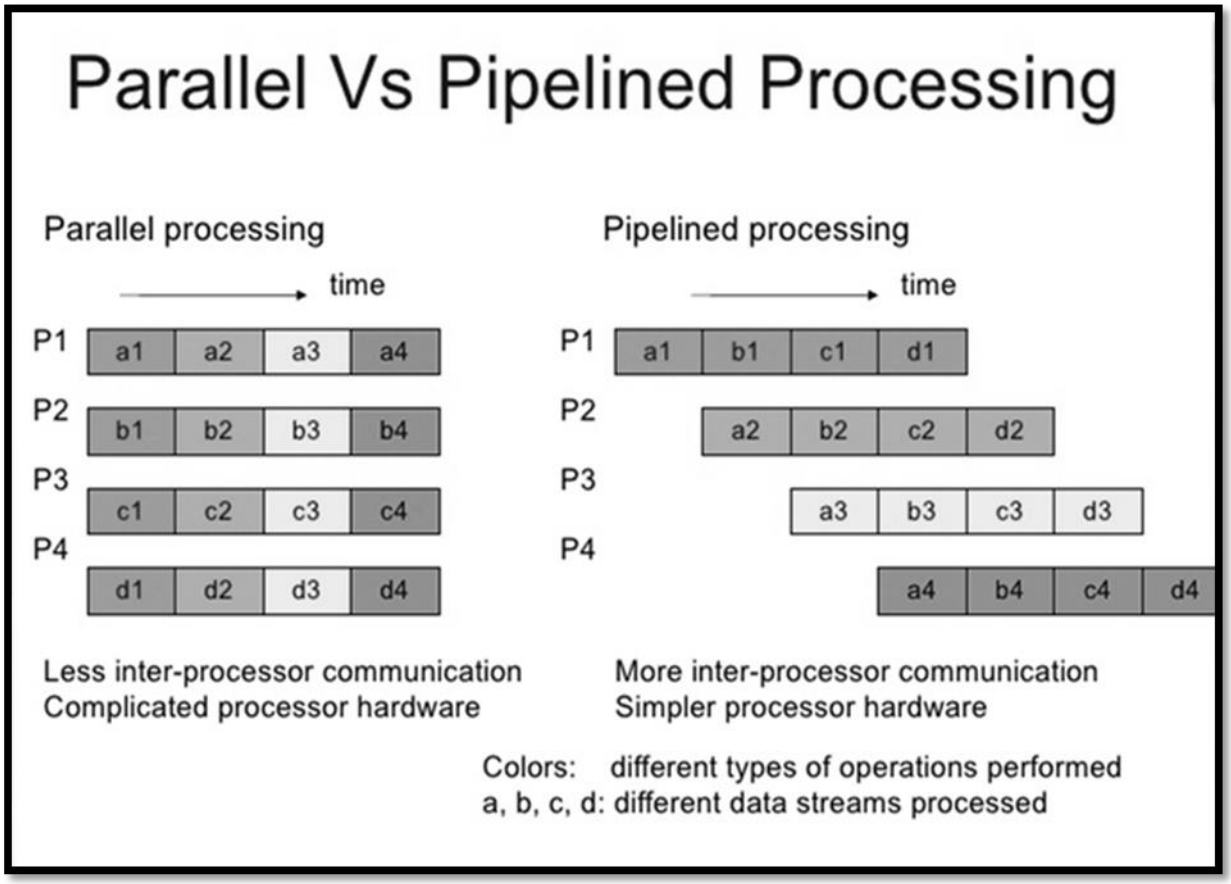
The Core i3-3xxM processors are based on Arrandale, the mobile version of the Clarkdale desktop processor. They are similar to the Core i5-4xx series but running at lower clock speeds and without Turbo Boost. A limited number of motherboards by other companies also support ECC with Intel Core ix processors; the Asus P8B WS is an example, but it does not support ECC memory under Windows non-server operating systems.

Core i5

The first Core i5 using the Nehalem microarchitecture was introduced on September 8, 2009, as a mainstream variant of the earlier Core i7, the Lynnfield core. Lynnfield Core i5 processors have an 8 MB L3 cache, a DMI bus running at 2.5 GT/s and support for dual-channel DDR3-800/1066/1333 memory and have Hyper-threading disabled. The same processors with different sets of features (Hyper-Threading and other clock frequencies) enabled are sold as Core i7-8xx and Xeon 3400-series processors, which should not be confused with high-end Core i7-9xx and Xeon 3500-series processors based on Bloomfield. A new feature called Turbo Boost Technology was introduced which maximizes speed for demanding applications, dynamically accelerating performance to match the workload.

The Core i5-5xx mobile processors are named Arrandale and based on the 32 nm Westmere shrink of the Nehalem microarchitecture. Arrandale processors have integrated graphics capability but only two processor cores. They were released in January 2010, together with Core i7-6xx and Core i3-3xx processors based on the same chip. The L3 cache in Core i5-5xx processors is reduced to 3 MB, while the Core i5-6xx uses the full cache and the Core i3-3xx does not support for Turbo Boost.[32] Clarkdale, the desktop version of Arrandale, is sold as Core i5-6xx, along with related Core i3 and Pentium brands. It has Hyper-Threading enabled and the full 4 MB L3 cache.


Serial Computation
To Run on a single computer with single CPU
Problem is broken into discrete instructions and are executed one after another
One instruction at a time
problem
instructions
CPU
Parallel Computation
To Run on Multiple CPUs
Problem is broken into discrete parts that can be solved concurrently
Each part is broken down into instruct-ions which is executed simultaneously on different CPUs
problem
instructions
CPU
CPU
CPU
CPU



References:

[1] http://www.dauniv.ac.in/downloads/CArch_PPTs/CompArchCh12L01MultProcArch.pdf

[2] http://www2.cs.dartmouth.edu/~dfk/papers/kotz:pioarch.pdf

[3] http://www.cs.vu.nl/~ast/books/mos2/sample-8.pdf

[4] http://compsci.hunter.cuny.edu/~sweiss/course_materials/csci360/lecture_notes/chapter_07.pdf

[5] https://www.cs.fsu.edu/~engelen/courses/HPC-adv-2008/PRAM.pdf

[6] http://people.cs.aau.dk/~adavid/teaching/MVP-08/02b-MVP08.pdf

[7]http://www.secs.oakland.edu/~ganesan/old/courses/CSE%20664%20W08/CSE%20664%20Parallel%20Architectures%201.pdf

[8] https://pdfs.semanticscholar.org/ca10/9a629de0dce2bce4a891e43e7bbb1a056da6.pdf

[9] http://www.cs.cmu.edu/afs/cs/academic/class/15418-s12/www/lectures/18_interconnects.pdf

[10] http://www2.cs.siu.edu/~cs401/Textbook/ch5.pdf

MCQ Questions:

i. Multiprocessor system has advantage of

A. Increased Throughput
B. Expensive hardware
C. operating system
D. both a and b

ii. Octa-core processor is processors of computer system that contains

A. 2 processors
B. 4 processors
C. 6 processors
D. 8 processors

iii. Symmetric multiprocessing in computer system does not use

A. master relationship
B. slave relationship
C. master slave relationship
D. serial processing

iv. System containing only one processor is called

A. multiprocessor
B. single processor
C. dual processor
D. specific processor

v. Multiprocessing provided by computer system has a type of

A. symmetric multiprocessor
B. asymmetric multiprocessing
C. symmetric multiprocessing
D. both b and c

vi. Interconnection networks are also called

A. Communication subnets
B. Communication subsystems
C. Cellular telecommunication
D. Both a and b

vii Algorithm that defines which network path, or paths, are allowed for each packet, is known as

A. Routing algorithm
B. Switching algorithm
C. Blocking algorithm
D. Networking algorithm

viii Address and data information is typically referred to as the

A. Request payload
B. Link
C. Tailer
D. Message payload

ix When number of switch ports is equal to or larger than number of devices, this simple network is referred to as

A. Crossbar
B. Crossbar switch
C. Switching
D. Both a and b

x All nodes in each dimension form a linear array, in the

A. Mesh topology
B. Bus topology
C. Star topology
D. Torus topology

Short Answer Type Questions:

1. What is the significance of interconnection network in the multiprocessor architecture?
2. What do you mean by multiprocessor system?
3. Give the architecture for a typical MIMD processor?
4. What is multistage switching network?
5. What is Uniform memory access (UMA)?
6. Distinguish between tightly and loosely coupled multiprocessor systems.
7. What is omega network? Explain with an example.
8. What is PRAM?
9. What is hypercube network?
10. What is shared memory multiprocessor system?

Assignment:

1. Describe different types of interconnection networks in computer system? What is multistage switching network?
2. Short note on cluster computer?
3. What are the similarities and dissimilarities between the multiprocessor system and multiple computer system?
4. What are the different architectural models for multiprocessors? Explain each of them with example.
5. What is the main difference and similarities between multicomputer and Multiprocessor?

6. What are the common data routing functions among the Processing Elements and how are they implemented?

7. Explain the main factors that can influence the Performance of interconnection networks.

8. What are the different types of Multi-stage interconnection networks?
9. Describe different types of interconnection networks in computer system?
10. What is the significance of interconnection network in the multiprocessor architecture?

Web/Video links:

[1] https://www.youtube.com/watch?v=WdqdebPmPuQ

[2] https://www.youtube.com/watch?v=Y8rhSNFG0AU

[3]http://www.dauniv.ac.in/downloads/CArch_PPTs/CompArchCh12L01MultProcArch.pdf

[4] https://www.youtube.com/watch?v=93CVEOXM3T4

[5] https://www.youtube.com/watch?v=LDXyqkxoE8w

[6] https://www.youtube.com/watch?v=WKXbvhkzBUo

[7] https://www.youtube.com/watch?v=rjobxf1Qs_o

[8] https://www.youtube.com/watch?v=rss_LriYLMw

[9] https://www.youtube.com/watch?v=EoONr6VZExA

[10] https://chetsarena.files.wordpress.com/2012/10/1-3-interconnection-network.pdf