EEE440
Computer Architecture
Handouts

FasihurRehman
Assistant Professor

VCOMSATS
Learning Management System
Computer Architecture

Introduction:

This handout provides supporting material of computer architecture to accompany with the video lectures. The content is mainly delivered in the lectures and this handout will follow the same order as that of used in video lectures.

A computer can be defined as a device that takes input, processes it according to a pre – stored program and sends results to outside world.

Von Neumann Architecture:

Although the history of computers can be taken from Abacus, a calculating frame, from fourteenth century yet evolution of modern computers started in 1940s when Von Neumann gave this architecture that has following features.

- Instructions and data are stored in memory
- Memory is region with binary data and every location has a unique address
- Instruction can only be executed in a CPU register (Register is smallest piece of memory normally referred to as memory element inside processor)
- Each high level language instruction is translated in many machine instructions

This means that a computer must have five functional units as shown in the Figure 1. The units are

1. **Input unit**: Receives data from outside world
2. **Output Unit**: Provides results to the outside world
3. **Memory Unit**: Stores data or/and instructions
4. **Arithmetic and Logic Unit (ALU)**: Performs arithmetic or logic operations
5. **Control Unit**: Provides timing and control signals to coordinate all actions

![Figure 1: Functional units of a computer](image-url)
First two jointly called I/O Units while 4 & 5 are known as Central Processing Unit (CPU).

Execution of a single instruction is based on one of the following operations:

1. Transfer word of data from register to another or the ALU
2. Perform arithmetic or logical operation and store result in a register
3. Fetch contents of a memory location to a register
4. Store register to a memory location

The details of these operations will be discussed later in this document.

**Functional Units of a Computer:**

**Memory Unit**

Stores Programs and Data

Memories are divided into two categories, Primary and Secondary

- **Primary**
  - Fast
  - Holds program during execution
  - Semiconductor cells
  - Each location has a unique address
  - Processed in words
  - Random Access Memory (RAM)
  - Access time
  - Hierarchy

- **Secondary**
  - Larger
  - Cheaper

**Arithmetic and Logic Unit**

- Computer operations are executed in ALU of the processor.

- Load the operands into memory – bring them to the processor – perform operation in ALU – store the result back to memory or retain in the processor.

**Control Unit**

- All computer operations are managed by the control unit.

- The timing signals that govern the I/O transfers are also generated by the control unit.

- Control unit is usually distributed throughout the machine.

- How control signals can be generated
Input and Output:

Data processed by the computer is not necessarily in the memory. Data has to be exchanged among peripheral devices and outside world. The way I/O operations are performed have impact on performance of the computer

- Input devices: keyboard, mouse, graphic tablet etc.
- Output devices: Monitor, printer, plotter etc.

Basic I/O

I/O is the means by which data are transferred between the processor and the outside world. Devices operate at different speeds to the processor so handshaking is required

Figure 2: Simple Interface [1]

Figure 2 shows a simple Keyboard and Monitor interface with a processor. The keyboard and display are coordinated via software. Register (on device) assigned to the keyboard hardware

- DATAIN contains ASCII of last typed character
- SIN is the status control flag, normally 0. When a character typed, becomes 1. After the processor reads DATAIN, it is automatically set back to 0

Register (on device) assigned to the display hardware

- DATAOUT receives a character code
• SOUT is the status control flag. It is 1 when ready to receive a character, set to 0 when the character is being transferred

These registers form the respective device interface

Interconnection Network

Interconnect used for communication between the processor, memory and I/O devices is called a Bus. Multiple busses are used for better performance. In practice, three types of busses are used i.e. Address

![Bus Organization](image)

Figure 3: Bus Organization [1]

Bus, Data Bus and Control Bus. Various industrial standards of busses exist. Peripheral Component Interconnect (PCI) is more common these days.

Operational Concepts

Instructions (programs) govern the activities of computer. Appropriate list of instructions is stored in the memory to perform a task. Processor performs the operation of instruction by

• bringing the instruction in to processor
• decoding the instruction
• perform the task
• everything is coordinated by control unit that generates appropriate control and timing signals

All operands (data) are also stored in memory

Example:

Let us now an instruction (Add LOCA, R0) is executed. This instruction adds the operand at memory location LOCA to the operand in a register R0 in the processor.

• Place the sum into register R0.
• The original contents of LOCA are preserved.
• The original content of R0 is overwritten.

• Instruction is fetched from the memory into the processor – the operand at LOCA is fetched and added to the contents of R0 – the resulting sum is stored in register R0.

Software:

Another key component of a computer system is software. There are two main types of software and those are system software and application software. System software
– Receive and translate user inputs and/or commands
– Stores and retries (data/instructions) from secondary storage
– Executing standard applications e.g. word processors, spreadsheets, compilers, linkers
– Provides interface to I/O units e.g. printer
– An example of system software is the operating system

While an application software is
– Usually written in high – level languages such as Java, C, C++ or Fortran etc.
– Compiler translates the program to an executable file which runs on the computer.

Performance:

Performance of a computer system is measured by the ability of a computer system to execute a certain (test) program. Performance is affected by hardware design, instruction set and compilers.

Performance depends on following factors

• T: processor time required to execute a program that has been prepared in high-level language
• N: number of actual machine language instructions needed to complete the execution (note: loop)
• S: average number of basic steps needed to execute one machine instruction. Each step completes in one clock cycle
• R: clock rate

\[ T = \frac{N \times S}{R} \]

Performance can further be enhanced by using parallelism i.e. performing multiple tasks simultaneously. Parallelism can be achieved by

• Pipelining (Performing independent tasks in parallel)
• Multicores (Having multiple processor cores in a single processor, like dual core, quad core etc.)
• Multiprocessors (parallel processing, multiple processors in one system)

Parallelism is out of scope of this course

**Instruction Execution**

Instructions of a program are executed in the processor. An instruction is brought into the processor (action is called fetch). The instruction is then decoded in the processor (register transfers take place) and the instruction is then executed. The result are stored accordingly. This process is also called **fetch and execute cycle**.

**Memory and Memory Locations**

Memory consists of huge number of storage cells capable of storing one bit each. Data are usually stored in groups of n bit called words. Throughout this course we have assumed 32 – bit words (i.e. 4 – byte words) unless stated otherwise. Memory is organized as shown in the Fig 4 and 5

![Figure 4: Memory Words](image-url)
Memory can either be retrieved as one word or one byte (8-bit), addresses, however, for each location are needed. A $k$-bit address memory has $2^k$ memory locations 0 – $2^k-1$, memory space.

24-bit memory: $2^{24} = 16,777,216 = 16\text{M} (1\text{M}=2^{20})$

32-bit memory: $2^{32} = 4\text{G} (1\text{G}=2^{30})$

1K(kilo)=$2^{10}$

1T(tera)=$2^{40}$

Byte locations have addresses 0, 1, 2, ... For 32 bit word length, successive words addresses are 0, 4, 8,...The addresses that begin at byte addresses in multiples of the number bytes in a Word are said to be aligned in memory. For example
32-bit word: word addresses: 0, 4, 8,....

64-bit word: word addresses: 0, 8, 16,....

LOAD and STORE are two memory operations. Load actually reads the given location and content of the memory is unchanged. Store overwrites that particular location with new data.

**Branching**

During normal execution, consecutive instructions are executed but under certain circumstances, control is shifted to another location. This is called branching or jumping. There are both conditional and unconditional branches. In conditional branch, branching takes place subject to certain condition like zero, negative etc.

**Addressing Modes**

A computer uses following addressing modes for accessing a memory location

- **Implied:** “ADD M[AR]” in “One-Address” instruction
- **Immediate:** Use of a constant in “MOV R1, 6”, i.e. R1 ← 6
- **Register:** Indicate which register holds the operand
- **Register Indirect:** Indicate the register that holds the number of the register that holds the operand

\[
\text{MOV } R1, (R2)
\]

- **Autoincrement / Autodecrement:** Access & update in 1 instr.
- **Direct Address:** Use the given address to access a memory location
- **Indirect Address:** Indicate the memory location that holds the address of the memory location that holds the data
- **Relative Address:** EA = PC + Relative Addr
- **Indexed:** EA = Index Register + Relative Addr
- **Base Register:** EA = Base Register + Relative Addr

**Indexing and Arrays**

Index mode – the effective address of the operand is generated by adding a constant value to the contents of a register. A general purpose register is designated as Index register

\[
X(R_i): \text{EA} = X + [R_i] \text{ where } X \text{ may be a +ve or -ve constant representing a numerical value.}
\]
### Table 2.1: RISC-type addressing modes.

<table>
<thead>
<tr>
<th>Name</th>
<th>Assembler syntax</th>
<th>Addressing function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>#Value</td>
<td>Operand = Value</td>
</tr>
<tr>
<td>Register</td>
<td>Ri</td>
<td>EA = Ri</td>
</tr>
<tr>
<td>Absolute</td>
<td>LOC</td>
<td>EA = LOC</td>
</tr>
<tr>
<td>Register indirect</td>
<td>(Ri)</td>
<td>EA = [Ri]</td>
</tr>
<tr>
<td>Index</td>
<td>X(Ri)</td>
<td>EA = [Ri] + X</td>
</tr>
<tr>
<td>Base with index</td>
<td>(Ri,Rj)</td>
<td>EA = [Ri] + [Rj]</td>
</tr>
</tbody>
</table>

EA = effective address
Value = a signed number
X = index value

Figure 6: Some important Addressing modes [1]
General, the Index mode provides access to an operand whose location is specified relative to a reference point within the data structure in which the operand appears. Some variants:

\[(R_i, R_j): EA = [R_i] + [R_j]\]
\[X(R_i, R_j): EA = X + [R_i] + [R_j]\]

In relative mode – the effective address is computed by the Index mode using the program counter in place of the general-purpose register.

\[X(\text{PC}) \quad (X \text{ is a signed number})\]

Branch > 0 \quad \text{LOOP}

This location is computed by specifying it as an offset from the current value of PC.

(a) Offset is given as a constant

(b) Offset is in the index register

Figure 7: Relative Addressing [1]
Branch target may be either before or after the branch instruction, therefore, offset is given as a signed number.

**Stacks**

Stacks are data structures that are implemented in the memory; elements are added or removed only from one end of the list. Last In First Out policy is implemented.

Stacks can be implemented in several ways, one way is

- First element placed in BOTTOM
- Grows in direction of decreasing memory address
- Assume 32-bit data word
- Subtract #4, SP

Placing an element on Stack is called PUSH while removing an element from stack is called POP

```
Move NEWITEM,(SP) ; push
```
Move (SP),ITEM ; pop

Add #4,SP

With auto-increment and auto-decrement

Move NEWITEM,-(SP); push

**Basic Input and Output**

The data on which the program instructions operate are not necessarily available in memory. Data may be transferred between processor and peripheral devices (disk, keyboard, etc.). I/O operations are essential. The way I/O operations are performed can significantly affect the performance of the computer.

Multiple I/O devices may be connected to the processor and the memory via a bus. Bus consists of three sets of lines to carry address, data and control signals. Each I/O device is assigned a unique address. To access an I/O device, the processor places the address on the address lines. The device recognizes the address, and responds to the control signals.

![Figure 9: I/O Devices connected to processor](image-url)
I/O devices can be addressed in two ways.

Memory Mapped I/O

- Devices and the memory share the same address space
- Any machine instruction that can access memory can be used to transfer data to or from an I/O device.
- Simpler software.

Separate Address space for I/O

- Devices and the memory have different address spaces
- Special instructions to transfer data to and from I/O devices.
- I/O devices may have to deal with fewer address lines.
- I/O address lines need not be physically separate from memory address lines.
- In fact, address lines may be shared between I/O devices and memory, with a control signal to indicate whether it is a memory address or an I/O address.

I/O devices can either be polled by the processor periodically for any service or the devices may raise interrupt in case when the device is ready for service.

In case of polling, processor halts its normal execution and polls each and every peripheral device for any service, services the device if required and resumes normal execution again. This scheme is very simple but takes a of useful processor time even when none of the I/O devices require any service. The problem is rectified by the scheme called interrupts.

Interrupts:

While using interrupts, the device itself tells processor by sending a signal to the processor through the processor interrupt line. Processor thus, can utilize the time to perform useful task.

Processor is, say, executing the instruction located at address i when an interrupt occurs. Routine executed as a result of interrupt request is known as the interrupt-service routine (ISR). When an interrupt occurs, control must be transferred to the ISR. But before transferring control, the current contents of the PC (i+1), must be saved in a known location usually stack. This way the execution at i+1 will resume upon return from the interrupt.

Subroutine call and ISR call are somewhat similar. A subroutine performs a task that is required by the calling program. Interrupt-service routine may not have anything in common with the program it
Interrupts. Interrupt-service routine and the program that it interrupts may belong to different users. As a result, before branching to the interrupt-service routine, not only the PC, but other information such as condition code flags, and processor registers used by both the interrupted program and the interrupt service routine must be stored. This enables the interrupted program to resume execution upon return from interrupt service routine.

Saving the executing program status involves memory operations. Execution of Interrupt service routine may be delayed a bit. This delay is called interrupt latency. Most processors save minimal information in order to reduce interrupt latency. When a processor receives Interrupt, it must jump/branch to ISR. Interrupting device must also be informed that processor has received the request. This either is done by a proper acknowledge signal or by data transfer between processor and device. ISR alters the execution sequence of currently running program. Some times this may not be desirable. So processors generally have ability to enable or disable interrupts. Normally ISR starts with Interrupt Disable instruction to avoid interruption from the same device. Last instruction enables the interrupts again.

Sometimes need to ensure that interrupts cannot occur. We may need to assign priorities e.g. high priority interrupts cannot be interrupted by low priority ones. It needs to be ensured that an active request doesn’t lead to an infinite loop. Interrupt routines may need to access data structures and ensure they do not get interrupted while doing so. Infinite loops are avoided by disabling all interrupts in first instruction of ISR.

**Multiple Interrupts:**

So far we have not discussed how multiple interrupts are handled. To handle multiple interrupts to poll every device after interrupt has occurred can be a simple solution but will make the system slow. Alternate solution is vectored interrupts. Interrupting device sends a code over the data bus to identify itself. Processor jumps to a table of addresses, indexed by the interrupt-vector code. The code may be the address of the first instruction of ISR. Interrupts are usually assigned priorities. During execution of ISR, interrupts are only accepted from the devices having higher interrupt priority.

![Figure 10: Interrupt nesting [1]](image)

Special priority arbitration hardware is used to negotiate priorities.
Direct Memory Access (DMA):

A special control unit is provided to transfer a block of data directly between an I/O device and the main memory, without continuous intervention by the processor. The operation of DMA controller, however, must be under the control of the program executed by the processor which means the processor must initiate the DMA transfer. Normally, DMA is used to transfer large blocks of data at high speed.

![DMA Registers](image)

**Figure 11: DMA Registers [1]**

DMA controller connects network to bus. Fig 12 shows two separate devices connected to the computer system. Disk controller has 2 independent DMA channels to 2 independent disks. Controller starts DMA transfer by writing address and word count information to disk controller. DMA controller operates independent of processor (processor does other things, perhaps runs a different program in a multitasking system). Memory accesses by processor and DMA controller perhaps prioritized. When DMA completes transfer, sets done and IRQ bit. DMA accesses can by interwoven with processor accesses (cyclestealing) or take over the bus (called block or burst mode). Arbitration is required to control DMA controllers and processors accessing memory at the same time.

DMA controller connects a high-speed network to the computer bus. Disk controller, which controls two disks, also has DMA capability. It provides two DMA channels. It can perform two independent DMA operations, as if each disk has its own DMA controller. The registers to store the memory address, word count and status and control information are duplicated.
Interface Circuits:

To add a peripheral to a microprocessor, interface circuits are designed. This involves hooking up whenever is required to the processor’s bus. I/O interface include the following:

- Storage buffer
- Status flags that can be accessed by the processor
- Address decoding circuitry
- Appropriate timing signals
- Format conversions e.g. serial to parallel

On computer side, interface circuit has bus signals for Address, Data, and Control. The device has Datapath and associated controls to transfer data between the interface and the I/O device. This side is called as a port. Ports are classified as:

- Serial port
Parallel port transfer data in the form of a group of bits, conventionally, 8 or 16 or more bits to or from the device while Serial port facilitates data transfer bit by bit. Processor and device communicate in the same manner for both the type of ports. Conversion from serial to parallel or vice versa, if needed, takes place inside the interface circuit.

**Parallel Interface:**

Keyboard is connected to a processor using a parallel port. Processor is 32-bits and uses memory-mapped I/O. On the processor side of the interface we have:

- Data lines.
- Address lines.
- Control or R/W line.
- Master-ready signal and Slave-ready signal.

![Figure 13: A parallel Interface](image)

On the keyboard side of the interface:

- Encoder circuit which generates a code for the key pressed.
- Debouncing circuit which eliminates the effect of a key bounce (a single key stroke may appear as multiple events to a processor).
- Data lines contain the code for the key.
- Valid line changes from 0 to 1 when the key is pressed. This causes the code to be loaded into DATAIN and SIN to be set to 1.
Serial Port:

Serial port connects the processor to I/O device through “one” data line allowing transfer of only “one bit” at a time. However, Parallel to serial conversion is takes place within the interface circuit. Serial port transfers data in serial mode to or from the device while on bus side the communication has to be in parallel.

Input shift register accepts input one bit at a time from the I/O device. Once all the bits in a word are received, the contents of the input shift register are loaded in parallel into DATAIN register. Output data in the DATAOUT register are loaded into the output shift register. Bits are shifted out of the output shift register and sent out to the I/O device one bit at a time. As soon as data from the input shift reg. are loaded into DATAIN, it can start accepting another word of data bit – by – bit. Input shift register and DATAIN registers are both used at input so that the input shift register can start receiving another word from the input device after loading the contents to DATAIN, before the processor reads the contents of DATAIN. This is called as double-buffering. Fewer lines are needed in serial interface therefore serial interface is more feasible for devices that are at larger distance from the computer. Speed of transmission over serial link is called “bitrate”. Since devices are available in range of speed, a serial interface must be able to work with a range of clock speeds. Examples are Universal Asynchronous Receiver Transmitter, RS – 232 – C.

Buses:

Processor, main memory and I/O devices are interconnected by a common bus. Bus provides communications path for transfer of data. It includes signals to handle interrupts and arbitration. Bus protocol defines set of rules for transferring data on a bus. Three types of busses are common Data, Address and Control. It should be specified whether read or write, usually a single R/W signal, read when 1 write when 0 along with operand size. Normally processor is master and device being addressed is the slave. In terms of clock signals, a bus can be Synchronous or Asynchronous. Common clock used to derive timing in synchronous bus. Asynchronous bus requires handshaking between master and Slave.

Device allowed to initiate data transfers on bus transactions is called bus master. After using the bus, bus master relinquishes control of the bus so others can acquire it. Bus arbitration is required to decide who is master when more than one requests bus at the same time. Two types

- Centralized
  - Arbiter circuit ensures only one request is granted at any time and enforces a priority scheme. Rotating priority is also possible.

- Distributed
  - All devices waiting to use the bus have equal responsibility in carrying out bus arbitration without using a central arbiter.
Basic Processing Unit:

A typical program consists of a series of steps specified by a sequence of machine instructions. An instruction is executed by carrying out a sequence of one or more primitive (arithmetic or logic) operations. Processor fetches one instruction at a time and performs the specified operation. Processor fetches instructions from consecutive memory locations until a jump or a branch instruction is come across. Program counter, a processor register, is used to keep track of the address of the memory location containing the next instruction to be fetched.

Figure 14: Inside a processor [1]
A simplified architecture of a processor is shown in Fig 14. Let’s first get accustomed to various components of the processor.

**PC**: Program Counter, a register that holds the address of next instruction to be executed

**MAR**: Memory Address Register holds the address of memory location whose content is to be fetched in the processor or where processor wants to write

**MDR**: Memory Data Register content of memory is fetched in this register or the data is sent to outside world through this register

**Y**: A temporary register that holds one of the operands to be operated on by ALU

**Mux**: Multiplexer selects constant 4 or content of Y register to be fed into ALU

**ALU**: Arithmetic and Logic Unit performs arithmetic and logic operations on the input(s)

**Z**: Temporary Register temporarily holds the output of ALU

**TEMP**: Temporary Register another temporary register that can be used to store operands or intermediate results

**R0 – R(n – 1)**: n general purpose registers

**IR**: Instruction Register instruction being executed currently is stored here

**Instruction Decoder and Control Logic**: Decodes current instruction and generates necessary control signals

Program execution involves sequence of steps based on the following rudimentary operations

1. Transfer word of data from register to another or the ALU: **Register Transfer**
2. Perform arithmetic or logical operation and store result in a register: **Arithmetic or Logic Operation**
3. Fetch contents of a memory location to a register: **Memory Read**
4. Store register to a memory location: **Memory Write**

Program, as described earlier, is executed by fetch and execute cycle. IR gets content of memory location stored in PC i.e. IR ← [[PC]] and PC is incremented. PC ← PC+4 (4 is added as next word will now be read and each word is 4 bytes long).

**Register Transfer**: Input and output of register Ri controlled via switches Riin and Riout. R4 ← R1: i.e. Set R1out to 1 causes content of R1 to be placed on CPU bus and R4in to 1 (others all 0) enables R4 to receive data that is available on CPU bus. All transfers are synchronized to a processor clock. Riout is called tri-state buffer.
Arithmetic or Logic Operations: The ALU is a combinational circuit that has no internal storage. ALU gets the two operands from MUX and bus. The result is temporarily stored in register Z. The sequence of operations and control signals to add the contents of register R1 to those of R2 and store the result in R3, i.e., $R3 \leftarrow R1 + R2$, is

1. $R1_{out}$, $Y_{in}$ (content of R1 is available on bus and Y is accepting data so content of R1 is copied to Y)

2. $R2_{out}$, SelectY, Add, $Z_{in}$ (Content of R2 on bus so available to B input of ALU, Select Y brings content of Y to A input of ALU, Add signal to ALU causes two operands to get added and result is stored in Z as it is ready to receive data).

3. $Z_{out}$, $R3_{in}$ (Content of Z placed on the processor bus. Data is copied to R3 as it is ready to receive the data).

Fetching Word from Memory: MDR, MAR connections to bus controlled as in figure 16. For Move (R1), R2
1. MAR ← [R1]

2. Start read on memory bus

3. Wait for Memory Function Completed (MFC) from memory (this signal is received when memory has been read and data has been written into the MDR, in case of write operation, MFC will mean data in MDR has been copied to the desired memory location)

4. Load MDR from bus (MFC means data has been written in MDR)

5. R2 ← [MDR]

Associated Control Signals are

1. R1out, MARin, Read

2. MDRinE, WMFC (wait for MFC)

3. MDRout, R2in

Figure 16: Memory operations

Storing word is also a similar procedure with slight changes i.e. control signal for write will be generated.
**Execution of Complete Instruction:**

Let’s now see that how a complete instruction is executed in the processor described above. Add (R3), R1 will read the content of memory whose address is in R3, add this to R1 and result will be stored in R1. Following will happen:

- Fetch the instruction
- Fetch the first operand (the contents of the memory location pointed to by R3)
- Perform the addition
- Load the result into R1

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PC\textsubscript{out}, MAR\textsubscript{in}, Read, Select4, Add, Z\textsubscript{in}</td>
</tr>
<tr>
<td>2</td>
<td>Z\textsubscript{out}, PC\textsubscript{in}, Y\textsubscript{in}, WMFC</td>
</tr>
<tr>
<td>3</td>
<td>MDR\textsubscript{out}, IR\textsubscript{in}</td>
</tr>
<tr>
<td>4</td>
<td>R3\textsubscript{out}, MAR\textsubscript{in}, Read</td>
</tr>
<tr>
<td>5</td>
<td>R1\textsubscript{out}, Y\textsubscript{in}, WMFC</td>
</tr>
<tr>
<td>6</td>
<td>MDR\textsubscript{out}, SelectY, Add, Z\textsubscript{in}</td>
</tr>
<tr>
<td>7</td>
<td>Z\textsubscript{out}, R1\textsubscript{in}, End</td>
</tr>
</tbody>
</table>

*Figure 17: Control Sequence*

Increment PC: Step 1: PC loaded into MAR and on CPU bus as well consequently available to B input of ALU, read request to memory, MUX selects 4, added to B (PC) and PC + 4 is stored in Z.

Step 2: Z moved to PC while waiting for memory, and in Y register (to be used in branch instruction) and wait for memory function complete which mean the memory has been read into the MDR.

Memory FetchStep 3: Word fetched (instruction read) from memory and loaded into IR.

Instruction DecodeStep 4: figure out what the instruction should do and set control circuitry for steps 4-7. R3 (address of memory location to be read) transferred to MAR and memory read operation initiated.
Register Transfer Step 5: contents of R1 moved to Y and waiting for memory function completion. The moment MFC is received, means data has been read into MDR.

Addition step 6: read operation completed and MDRout places data on B input of ALU. Select Y to provide other operand input of ALU and addition performed, the sum goes Z register.

Completion Step 7: result is transferred to R1, End causes a goto step 1 i.e. new instruction and so on.

**Execution of Branch Instruction:**

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}</td>
</tr>
<tr>
<td>2</td>
<td>Z_{out}, PC_{in}, Y_{in}, WMFC</td>
</tr>
<tr>
<td>3</td>
<td>MDR_{out}, IR_{in}</td>
</tr>
<tr>
<td>4</td>
<td>Offset-field-of-IR_{out}, Add, Z_{in}</td>
</tr>
<tr>
<td>5</td>
<td>Z_{out}, PC_{in}, End</td>
</tr>
</tbody>
</table>

*Figure 18: Control sequence for branch instruction [1]*

Now let’s see how unconditional branch instruction is executed. Steps 1-3, instruction fetch is common in every instruction.

Step 4: Offset from the next address is provided by IR on bus as well as B input of ALU when instruction is decoded. Offset to the PC (as content of PC and Y is same). The new address is now in Z.

Step 5: update the PC

For conditional e.g. branch < 0, step 4 is replaced with

*Offset-field-of-IR_{out}, Add, Z_{in}, If N=0 then End*

**Multiple Bus Organization:**

So far, we have seen a processor with a single bus. Now we explore how processor with multiple buses works and organized. One disadvantage of our single bus scheme is that only one data item can be transferred over the bus per cycle. A possible solution is multiple internal buses. All registers combined into a register file with 3 ports. Figure 19 shows the multiple bus model.
Figure 19: Three bus model

Buses A and B allow simultaneous transfer of the two operands for the ALU. ALU is able to just pass one of its operands to R unaltered e.g. R=X.

Incrementer unit computes PC+4, means we don’t need the ALU for this. ALU still has a 4 input for other instructions such as post-increment e.g. writing a block into the memory

Execution of ADD R4 R5 R6:

Now we execute the above instruction which adds contents of R4 and R5 and saves result in R6. Control sequence is given in Figure 20.
Instruction is fetched in first step. Content of PC is placed on bus B that passes on to R the o/p of ALU and thus on bus C and MAR gets the content of PC (the address of instruction to be fetched). Read request is tendered and PC is incremented. WMFC means memory has been read into the MDR. Instruction is passed on to IR though ALU. Decoding the instruction generates control signals to provide contents of R4 (through Mux) and R5 to ALU where these added and result directly goes to R6.

Here we have seen that not only we have been able to reduces some blocks in processor (e.g. temp, Z etc.) but also gained in execution time. Addition in this case is accomplished in 4 steps (clock cycles) as compared to 7 steps in single bus architecture.

**Control Unit:**

CPU executes sequence of instructions. Each instruction may be executed in several steps. Each step is executed in several sub–tasks. Micro operations fall in one of the following

- Register Transfer
- Arithmetic or Logic operations
- Memory read operation
- Memory write operation

To perform these tasks, at least one control signals has to be generated. For example, in every operation (single bus architecture) fetch operation i.e. step 1 – 3 were common. If control signal for ‘Read’ is issued, Data will be available in MDR after one additional step. Each operation may be regarded as a micro-operation.
Several micro-operations can be performed in the same control step if they don’t conflict as in figure 17.

Control unit is driven by the processor clock. Generated control signal depends on

- The actual step to be executed
- The condition and status flag of the processor
- The actual instruction executed
- Any external signal received (such as interrupts)

To execute instructions, a processor must have arrangement to generate control signals in proper sequence. Control signals can be generated

- Hard-wired Control
- Micro-programmed Control
- Programmable Logic Array

**Hardwired Control:**

A hardwired control is also called a finite state machine that produces control signals at the right time using counter. Control signals are functions of the IR, external inputs and condition codes. Hardwired system can operate at high speed; but with little flexibility.

**Generation of Control Signals:**

Each step in sequence of execution is completed in one clock cycle. A counter (called step counter) is used to keep track of the control steps.
Step decoder provides a separate signal line for each step in the control sequence. The o/p of the instruction decoder comprises separate lines for each machine instruction. For any instruction loaded in Instruction Register, only one of the output line INS1 – INSm is HIGH (i.e. 1) all other lines will be LOW (i.e. 0). The decoder inputs are combined to generate individual control signals. For example, END can be implemented as (logic function and circuit);

\[
\text{End} = T_7 \cdot \text{ADD} + T_5 \cdot \text{BR} + (T_5 \cdot N + T_4 \cdot N) \cdot \text{BRN} + ... 
\]
Microprogrammed Control:

In 1951 Maurice Wilkes came up with idea of sequencing of control signals within the computer similar to the sequencing actions required in a regular program. A stored program to represent the sequences of control signals. He named it microprogramming. Machine instructions are divided into sub-instructions (microinstructions) that implement the instruction set of the machine. Full set of microinstructions made up the microprogram.

Control signals are generated by executing a program similar to machine language programs.
Control Word: is a word whose each bit represents a control signal. Each control word is a unique combination of 0’s and 1’s. **Controlword** is also called micro instruction. For example: SelectY = 1 while Select4 = 0. **MicroRoutine**: Collection of control words for a particular machine instruction.
Each micro-instruction is stored in a special memory called control store. Control unit sequentially reads that memory and generates control signal. A program counter called micro-programmed counter is used to read control words (micro-instructions) sequentially. Micro-programmed counter is incremented by each clock cycle. Control signals are thus generated using that particular CW.

A micro-program counter is used to read control words sequentially from control store. Every time a new instruction loaded into the IR, output of “Starting Address Generator” loaded into the μPC. μPC automatically incremented by clock causing successive microinstructions to be read from the control store. Control signals delivered to various parts of the processor in the correct sequence. This scheme is not able to change its sequence as a result of other inputs such as the condition code e.g. Branch < 0.
A microprogrammed control with external inputs is organized. Starting and branch address generator is included that loads new address into $\mu$PC when instructed and has condition codes and external inputs which can affect $\mu$PC. $\mu$PC incremented every cycle except

- When new instruction loaded into IR, $\mu$PC loaded with starting address of the micro-routine.
- For taken branches, $\mu$PC updated to branch address.
- For End microinstruction, $\mu$PC set to 0.

Microinstructions can simply be structured by assigning one bit position to each control signal as shown in Figure 23. This is very inefficient, however. The length can be reduced: most signals are not needed simultaneously, and many signals are mutually exclusive. All mutually exclusive signals are placed in the same group in binary coding. Having a separate microroutine for each machine instruction results in a large total number of microinstructions and a large control store. Moreover, execution takes longer because it takes more time to carry out the required branches.
The microprogram we discussed requires several branch microinstructions, which perform no useful operation in the datapath. A powerful alternative approach is to include an address field as a part of every microinstruction to indicate the location of the next microinstruction to be fetched.

- Pros: separate branch microinstructions are virtually eliminated; few limitations in assigning addresses to microinstructions.
- Cons: additional bits for the address field are required.

**Arithmetic Unit:**

Signed numbers (integers) are represented
- Signed Magnitude
- 1’s Complement
- 2’s Complement

In signed magnitude, MSB is used as sign bit 0 for positive and 1 for negative values. Remaining n – 1 bits represent magnitude of the number. This scheme has some flaws as it has a positive zero and a negative zero. Moreover, arithmetic operations don’t yield correct results.

In 1’s complement,

\[ -N = (2^n - 1) - N \]

Where: \( n \) is the number of bits per word, \( N \) is a positive integer, \(-N\) in 1’s complement (Negative Number)

For example with an 8-bit word and \( N = 6 \), we have:

\[ -N = (2^8 - 1) - 6 = 255 - 6 = 249 = 11111001 \]

Alternatively

\[ N = +6 = 00000110 \]

\[ (-N) = -6 = 11111001 \]

The values 00000000 and 11111111 both represent zero in eight bit. Moreover, arithmetic operations don’t yield correct results as they have to be manipulated.

An alternate and adopted solution is 2’s complement representation which not only has only one zero but also yields correct results after arithmetic.

\[ N^* = 2^n - N \]
Where: \( n \) is the number of bits per word, \( N \) is a positive integer and \( N^* \) is \(-N\) in 2's complement notation.

For example with an 8-bit word and \( N = 6 \), we have: \( N^* = 28 - 6 = 256 - 6 = 250 = 1111010 \).

Alternatively, to take 2's complement is to start at the right and complement each bit to the left of the first "1".

For example: \( N = +6 = 00000110 \)

\[ N^* = -6 = 1111010 \]

**Overflow:**

Overflow, in case of addition, can only occur when both the operands have same sign.

Overflow happens when (at MSB)

\[ \text{Carry in} \neq \text{Carry out} \]

Normally, overflow is referred to 'V' flag in status register.

**Addition:** Addition can be implemented using following truth table shown in Figure 26.

<table>
<thead>
<tr>
<th>( x_i )</th>
<th>( y_i )</th>
<th>Carry-in ( c_i )</th>
<th>Sum ( x_i )</th>
<th>Carry-out ( c_{i+1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ x_i = \overline{x_i} y_i c_i + \overline{x_i} y_i \overline{c_i} + x_i \overline{y_i} c_i + x_i y_i c_i = \overline{x_i} y_i \oplus c_i \]

\[ c_{i+1} = y_i c_i + x_i y_i + x_i \overline{y_i} \]

**Example:**

\[ \frac{X}{Y} = \frac{7}{13} = 0 \begin{array}{c} 1 \end{array} 0 \begin{array}{c} 1 \end{array} \]

\[ \begin{array}{c} c_{i+1} \end{array} \overline{y_i} \boxed{x_i} \]

**Legend for stage \( i \)**

**Figure 26: Logic for binary addition [2]**
Subtraction can be performed by taking 2’s complement. Figure 29 shows adder with subtraction capability.
This circuit takes 2’s complement of the input $y_i$ when Add/Sub control is 1.

Overflow has special significance in arithmetic so its detection has key importance. Overflows can only occur when both the operands have same sign. Overflow occurs if the sign bit of the result is different from the sign bits of the operands. MSB represents the sign. Overflow can be detected by the following logic expressions:

$$\text{Overflow} = C_{n-1}$$

**Determination of delays:**

**Delay** is defined as time interval between the instances when o/p appears after the input has been applied. Delay in the o/p are evaluated as

- Sum is available in 1 gate delay
- Carry is available in 2 gate delays

For $n$ – bit adder

- $s_{n-1}$ is available after $2n-1$ gate delays
• \(c_n\) is available after 2n gate delays.

• Overflow is available after 2n + 2 gate delays

Our design objective is to perform tasks in minimum possible time i.e. minimum delays. In order to reduce delays, we predict carry which is the main source of delay. This leads to design of fast adders. Since

\[
\begin{align*}
\begin{align*}
s_i &= x_i \oplus y_i \oplus c_i \\
c_{i+1} &= x_i y_i + x_i c_i + y_i c_i
\end{align*}
\end{align*}
\]

Which can be re-written as

\[
\begin{align*}
c_{i+1} &= x_i y_i + (x_i + y_i) c_i \\
c_{i+1} &= G_i + P_i c_i
\end{align*}
\]

where \(G_i = x_i y_i\) and \(P_i = x_i + y_i\)

\(G_i\) and \(P_i\) are called Generate and Propagate at \(i^{th}\) stage. \(G_i\) and \(P_i\) help in evaluating carry based on inputs.

\[
c_{i+1} = G_i + P_i c_i + P_i P_{i-1} c_{i-1}
\]

Continuing

\[
c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \cdots + P_i P_{i-1} \cdots P_i P_{0} G_0
\]

Thus all carries can be obtained 3 gate delays after the application of \(X, Y\) and \(c_0\).

• One gate delay for \(P_i\) and \(G_i\)

• Two gate delays in the AND-OR circuit for \(c_{i+1}\)

All sums can be obtained 1 gate delay after the carries are computed independent of \(n\), so \(n\)-bit addition requires only 4 gate delays. This is called Carry Lookahead adder.

**Multiplication:**

At the most, product of \(2n\) – bit numbers is \(2n\) – bit number. Multiplication of unsigned numbers can be regarded as addition of shifted versions of the multiplicand. Partial products are either added at the end or partial products will be added at each stage.
These Multipliers are also called combinatorial array multipliers. They highly inefficient and use a very high number of gates moreover not useful for signed numbers. Sequential multipliers are used.

For sequential multipliers, if the $i^{th}$ bit of the multiplier is 1, shift the multiplicand and add the shifted multiplicand to the current value of the partial product. Multiplicands are left shifted before adding to partial product. Equivalently, same result can be achieved by adding “un – shifted” multiplicand to right shifted partial product.
Multiplication of Signed Numbers:

Considering 2’s-complement signed operands, let's what happen to (-13)×(+11) if following the same method of unsigned multiplication.

```
  1 0 0 1 1  (-13)
× 0 1 0 1 1  (+11)
  1 1 1 1 1  1 0 0 1 1
  1 1 1 1 1  0 0 1 1
  0 0 0 0 0 0 0 0 0
  1 1 1 0 0 1 1
  0 0 0 0 0 0
_________________________
  1 1 0 1 1 1 0 0 0 1 1  (-143)
```
Sign extension is blue.

Multiplication of signed numbers (Positive multiplier and negative multiplicand) can be carried out by above algorithm if sign extension can be implemented. The case of positive multiplicand and negative multiplier can be handled by taking 2’s – complement of both the multiplier and multiplicand and using above algorithm for multiplication, as the product will remain unchanged.

**Booth’s Algorithm:**

Consider in a multiplication, the multiplier is positive 0011110, appropriately shifted versions of the multiplicand are added in a standard procedure.

\[
\begin{array}{ccccccccc}
0 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & +1 & +1 & +1 & +1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\
\end{array}
\]

We can write multiplier as

\[
\begin{array}{c}
0100000 \\
- 0000010 \\
\hline
0011110
\end{array}
\]

(32)   (2)   (30)
Generally, in the Booth’s algorithm, -1 times the shifted multiplicand is selected when moving from 0 to 1, and +1 times the shifted multiplicand is selected when moving from 1 to 0, as the multiplier is scanned from right to left.

![Booth recoding table](image)

- **Multiplier**
<table>
<thead>
<tr>
<th>Bit $i$</th>
<th>Bit $i-1$</th>
<th>Version of multiplicand selected by bit $i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$0 \times M$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$+1 \times M$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$-1 \times M$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$0 \times M$</td>
</tr>
</tbody>
</table>

- **Figure 32: Booth recoding table [2]**
Thus recoding can be done using above table. This handles both positive and negative numbers uniformly and achieves some efficiency in terms of number of additions required when multiplier has large blocks of 1’s.

In order to improve multiplication speed, following multiplier bit recoding is also used.

<table>
<thead>
<tr>
<th>Multiplier bit-pair</th>
<th>Multiplier bit on the right</th>
<th>Multiplicand selected at position $i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i + 1$  $i$</td>
<td>$i - 1$</td>
<td>$0 \times M$</td>
</tr>
<tr>
<td>0        0</td>
<td>0</td>
<td>$+1 \times M$</td>
</tr>
<tr>
<td>0        0</td>
<td>1</td>
<td>$+2 \times M$</td>
</tr>
<tr>
<td>0        1</td>
<td>0</td>
<td>$-1 \times M$</td>
</tr>
<tr>
<td>0        1</td>
<td>1</td>
<td>$-2 \times M$</td>
</tr>
<tr>
<td>1        0</td>
<td>0</td>
<td>$0 \times M$</td>
</tr>
<tr>
<td>1        0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1        1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1        1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Figure 33: Multiplier bit recoding table [2]

Division:

Division is carried out by longhand division method. For longhand division, position the divisor appropriately with respect to the dividend and performs a subtraction. If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed. If the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is repositioned for another subtraction. Arrangement shown below performs restoring division.

- Shift $A$ and $Q$ left one binary position
- Subtract $M$ from $A$, and place the answer back in $A$
- If the sign of $A$ is 1, set $q_0$ to 0 and add $M$ back to $A$ (restore $A$); otherwise, set $q_0$ to 1
- Repeat these steps $n$ times
A non-restoring division technique is as under.

- **Step 1:** (Repeat $n$ times)
  - If the sign of $A$ is 0, shift $A$ and $Q$ left one bit position and subtract $M$ from $A$; otherwise, shift $A$ and $Q$ left and add $M$ to $A$.
  - Now, if the sign of $A$ is 0, set $q_0$ to 1; otherwise, set $q_0$ to 0.

- **Step 2:** If the sign of $A$ is 1, add $M$ to $A$

**Floating point numbers:** Floating points are represented by IEEE standard for representation of floating point numbers as shown.

---

**Figure 34: Divisor [2]**

**Figure 35: IEEE representation [2]**
### Special Values

Special values are represented in table above [2]. Infinity can positive or negative depending upon sign bit. Denormal values are used to allow gradual underflow. Denormal value is smaller than the smallest value that can be represented. NaN means Not a Number: 0/0 or square root of a negative number.

Floating point numbers are represented in a normalized form. MSB of the mantissa is always equal to 1. We can represent numbers without storing the MSB.

#### Addition / Subtraction Rules:

Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents. Set the exponent of the result equal to the larger exponent. Perform addition/subtraction on the mantissas and determine the sign of the result. Normalize the resulting value, if necessary.

#### Multiplication Rules:

Add the exponents and subtract 127 to maintain the excess-127 representation. Multiply the mantissas and determine the sign of the result. Normalize the resulting value, if necessary.

#### Division Rules:

Subtract the exponents and add 127 to maintain the excess-127 representation. Divide the mantissas and determine the sign of the result. Normalize the resulting value, if necessary.

### Memory Systems:

Memory is integral part of a computer system. A key design issue is to provide a computer system with as large and fast memory as possible, within given cost. Maximum size of memory is determined by addressing scheme, e.g. 16-bit addresses can only address $2^{16} = 65536$ memory locations. Most machines are byte addressable but retrieve/store data in words.

- $1K = 2^{10}$
Handouts  

- $1M = 2^{20}$
- $1G = 2^{30}$
- $1T = 2^{40}$

Memory speed is measured by

- **Memory access time**: Time between start and finish of a memory request and
- **Memory cycle time**: minimum delay between successive memory operations

Each memory cell holds one bit of information. Memory cells are organized as a two-dimensional array. Each row is one memory word. All cells of a row are connected to a common line, known as the “word line”. Word line is connected to the address decoder. Sense/write circuits are connected to the data input/output lines of the memory chip.

**Random access memory** (RAM), any location can be accessed with comparable access time (e.g. CD vs. tape drive). Processor usually runs much faster than main memory. Use a cache memory to store data that is likely to be used.

**Types of RAM’s:**

Static RAM (SRAM) are fast but small. Two transistor inverters are cross connected to implement a basic flip-flop. The cell is connected to one word line and two bits lines by transistors T1 and T2. When word line is at ground level, the transistors are turned off and the latch retains its state. Read operation: In order to read state of SRAM cell, the word line is activated to close switches T1 and T2. Sense/Write circuits at the bottom monitor the state of b and $b'$.

![Figure 36: Basic Flip flop](image)

Figure 36: Basic Flip flop [2]
Circuits of SRAM are capable of retaining their state as long as the power is applied. So these are volatile memory: because contents are lost when power is switched OFF. Access times of static RAMs are in the range of few nanoseconds. SRAM are costly.

Dynamic RAM:

Dynamic RAMs store data as charge on a capacitor. Capacitors leaks away with time so must be refreshed. In return for this trouble, the density is much higher. The common type used today is called a synchronous DRAM as it uses a clock. It does not retain their state indefinitely even if power is kept ON. Contents must be periodically refreshed. DRAM may be refreshed while being accessed for reading.

Synchronous DRAM:

In SDRAMs Memory operation is synchronized with processor clock. The outputs of the sense circuits are connected to a latch. During a Read operation, the contents of the cells in a row are loaded onto the latches. The contents of the cells are refreshed (without changing) the contents of the latches while refreshing. Latches hold data that corresponds to the selected columns are transferred to the output. For a burst mode of operation, successive columns are selected using column address counter and clock. CAS signal need not be generated externally. A new data is placed during raising edge of the clock. Memory latency is the time it takes to transfer a word of data to or from memory. Memory bandwidth is the number of bits or bytes that can be transferred in one second.

DDR SDRAMs:

Cell array is organized in two banks. Double data rate SDRAM transfers data on both clock edges (normal circuits only operate on rising clock edge).

Read Only Memories ROMs:

RAMs are volatile. We need some memory to be retained even if the power is switched off, for example, booting a computer. So we need some arrangement or non-volatile memory to store some instructions that need to be executed at start up. Non-volatile memory is read in a same manner as the RAM and called Read only Memories. Special writing process is needed to write ROMs.

Types of ROMs:

ROMs are of many types.

ROMs Data are stored at the time of manufacturing these types of memories. Data can then be only read and cannot be modified.

Programmable ROM (PROM):

User is allowed to load data. Data once inserted cannot be modified.
**Erasable PROM:**

User is allowed to load data. Stored data can be erased and new data can be loaded. EPROM provides flexibility during development of digital system. A special EPROM eraser is required to erase the stored data. Erasing requires exposure to UV rays. EPROM has to be removed from the circuit for erasing the data.

**Electrically Erasable PROM:**

EEPROM removes the disadvantages of EPROM. Requires only Electrical signal to read or write the data.

**Flash Memory:**

Flash memory uses same approach as that of EEPROM. Read operation is only from one cell while block of data can be written simultaneously. Flash has higher density and consumes very little power. Single flash chip has smaller memory so for larger memories, flash cards and flash disks are used.

**Speed Size and Cost:**

It’s challenging to design a computer system that provides large memory with adequate speed at an affordable cost. Static RAM’s are fast but they are expensive because of their cell complexity. So it’s nearly impossible to fabricate large number of cells on a single chip. Dynamic RAM have simple circuits making them cheap but are slower. Magnetic Disks are even slower although have huge memories and very cheap. So memory hierarchy is used to in order to achieve the aforesaid design objective. All types of memories are used in this hierarchy.
Figure 37: Memory Hierarchy [2]

Registers are at the top of the memory hierarchy providing fastest access. Processor cache can be implemented on the processor chip. Two levels of cache.

- Level 1 (L1) cache is on the processor chip.
- Level 2 (L2) cache is in between main memory and processor.

Main memory (MM) is at next level and is implemented as SIMMs. MM is much larger and much slower as compared with cache memory.

Magnetic disks (Normally last level) have huge capacity and are inexpensive.
Cache Memories:

Processor runs at much faster speed than its main memory (MM). Processor will then have to wait for a while data and instruction is read from the memory. This degrades the performance of the computer system. Speed of MM is bounded by certain constraints. Cache memory uses principal of “Locality of Reference” which makes MM appear (to the processor) faster than it actually is.

Locality of Reference has both temporal and spatial aspects.

Temporal (in time) When information item (instruction or data) is first needed, brought into cache where it will hopefully be used again.

Spatial (in space) Rather than a single word, fetch data from adjacent addresses as well. Block refers to a set of contiguous addresses of given size (cache block also called cache line). Replacement algorithm is required to decide what to do when cache is full.

Cache is transparent to the processor. When processor issues a read request, a block of words (one word at a time) is transferred from MM to cache. Cache has subsequent blocks of word data as well now.

Read request: Contents of a block are read into the cache the first time. Subsequent accesses are (hopefully) from the cache (called a read hit). Number of cache entries is relatively small, need to keep most likely data in cache. When an un-cached block is required, need to employ a replacement algorithm to remove an old block and to create space for the new.

Write Operation: Normally either of two schemes is used.

Scheme 1: Cache and memory updated at the same time (write-through)

Scheme 2: Update cache location and mark as dirty. Main memory updated when cache block is removed (write-back).

Mapping functions determine how addresses are assigned to cache locations. Only a few blocks can be accommodated in the cache. In order to determine which block of MM is in cache, mapping functions are used.

When cache memory is full and more blocks have to be brought in the cache, replacement algorithms are required to determine which block is to be replaced. If data is in the cache it is called a read or write hit. Read Hit means the data is read from the cache and write Hit means cache has copy of the contents of memory. Contents of memory and cache may be updated using write through protocol or write back protocol.

Mapping Functions:
As we have seen above, mapping functions are required as only a few blocks can be accommodated in cache. Three types of Mappings viz. Direct Mapping, Associative Mapping, Set Associative Mapping functions are in use.

In direct mapping, memory address is divided into 3 fields. Block determines position of block in cache. Tag is used to keep track of which block is in cache (as many blocks can map to same position in cache) and word selects which word of the block to return for a read operation. Direct mapping scheme is simple but not very flexible.

In direct mapping, the block is restricted to reside in a given position in the cache. Associative mapping allows block to reside in an arbitrary cache location. In this example, all 128 tag entries must be compared with the address Tag in parallel so cost is higher.

Figure 38: Direct Mapping [2]
Combination of direct and associative makes set associative scheme of mapping. Blocks 0, 64, 128, ..., 4032 map to cache set 0 and can occupy either of 2 positions within that set. A cache with k-blocks per set is called a k-way set associative cache.
Replacement Algorithms:

In direct mapped cache position of each block fixed, no replacement algorithm needed.

For associative and set associative mappings, however, we need to decide which block to replace (keep ones likely to be used in cache). One strategy is least recently used (LRU) e.g. for a 4 block cache, use a 2-bit counter. Set =0 for block accessed, other blocks incremented. Block with count=3 replaced upon a miss. Another algorithm is random replacement (choose random block). Its advantage is its easy implementation at high speed.

Performance:

Performance of a computer depends on...
• The speed at which, instruction is fetched in the processor.
• Speed of the execution of the instruction.
• Instructions and Data are quickly accessible when referenced memory locations are present in cache.

Performance, therefore, depends upon cache miss or cache hit. The number of hits stated as fraction of all attempted accesses is called **HitRate**. **MissRate** is the number of misses stated as fraction of attempted accesses. The extra time needed to bring the desired information into the cache is called the **MissPenalty**.

Our goal is to have a memory system with speed of cache and size of a hard disk, i.e. high hit rates (> 90%) is essential. Miss penalty must also be reduced.

Hit rate can be improved by increasing block size without changing the cache size. For best results, block size has to be kept of optimal size; neither very large nor very small blocks yield good results. When loading new blocks into cache, if used, load-through approach can reduce miss penalty.

Load – through approach: In case of a read miss, the requested info may also be sent to the processor as soon as it is transferred to the cache without waiting for the transfer of the whole block.

Write buffers can be used to speed up writing in both write through and write back protocols.

**Virtual Memory:**

For 32-bit memory, every process assumes 4GB of address space available to it. Memory contents for each process are

• Program’s EXE image
• Any non-system DLL
• Program’s global data
• Program’s stack
• Dynamically allocated memory
• Memory mapped files
• Inter-process shared memory blocks
• Memory local to specific executing thread
• Special memory blocks like virtual memory tables
• OS kernel and DLL’s
So, it is assumed that every process requires huge memories (RAMs). So we use disk space instead and in 32-bit addressable space, every process assumes to have 4GB of memory. This concept is referred to as virtual memory.

Using cache memory, we got improvement in speed of the memory. Now we’ll discuss an architectural solution that enhances the effective size of the memory of a computer system. Number of address bits in a computer determines the size of the maximum addressable space. Generally, computers don’t have that much physical memory. Programs, these days are huge and don’t fit in MM but are stored in hard disks. Virtual memory creates illusion of this large memory by bring small pieces of programs to MM from secondary storage device.

Concept of Virtual memory is similar to cache memory. Cache bridges the speed gap between the processor and main memory and is implemented in hardware. Virtual Memory, on the other hand, bridges speed gap between main memory and secondary memory and partly implemented in software.

Physical memory is not as large as address space spanned by processor e.g. processor might address 32-bits (4G) but memory may only be 1G. Part of program not in main memory stored on hard disk and is brought into main memory (using DMA) as needed. This is done automatically by the OS; application program does not need to be aware of the existence of virtual memory (VM). Addresses generated by the processor are called virtual or logical addresses. Memory management unit (MMU) translates virtual addresses to physical addresses.
Address Translation:

Memory divided into pages of fixed size. A page is a block of words occupying contiguous addresses in MM. Page is a basic unit of information that is exchanged between MM and secondary storage. Commonly size ranges from 2KB to 16KB. Too small means too much time will be spent in getting pages from disk and too large means a large portion of the page may not be needed or reference. Processor generates virtual addresses. High order bits (most significant bits) are the virtual page number and low order bits (least significant bits) are the offset.

Information about where each page is stored is maintained in a software controlled data structure in main memory called the page table. Starting address of page table is called the page table base register. Area of the main memory that can hold a page is called page frame. Address in physical memory is obtained by indexing the virtual page number from the page table base register.
Figure 42: Address Translation [2]

Page table entries (PTE) also include control bits describe status of page such as whether page is actually in the memory, whether the page has been modified, does program have read permission and does program have write permission etc.

Page table is used by Memory Management Unit (MMU) for every read and write request. So where should we place the page table? Ideally, MMU will be ideal place for page table. Page table size is huge. Since MMU is part of the processor this much memory cannot be made part of the MMU. So page table is implemented on the main memory. To speed up the access, a copy of small portion of the page table can be implemented within the MMU. This small cached table is called Translation Lookaside
Buffer (TLB). TLB operates similar to cache memory. Normally associative or set associative scheme is used for TLB. Processor must keep TLB and page table information consistent.

Figure 43: Translation Lookaside Buffer [2]

High-order bits of the virtual address generated by the processor select the virtual page. These bits are compared to the virtual page numbers in the TLB. If there is a match, a hit occurs and the corresponding address of the page frame is read. If there is no match, a miss occurs and the page table within the main memory must be consulted. Commercial processors use set-associative mapped TLBs.

A pagefault is said to occur if a program generates a request to access a page that is not in the memory. MMU generates exception. Processing of active task is suspended. Operating System (OS) takes control. OS transfers the required page to MM from the secondary storage (SS). Suspended program gets the control back after the page has been copied to the MM. Since secondary storage is accessed, the page transfer may cause a significant delay.

A page has to be brought in MM from the disk but if MM is full, an approach similar to cache implementation can be used. Principal of Locality of Reference can be applied here and least recently used algorithm can be used.
A page may be modified during its stay in MM, if such a page (dirty page) is to be replaced, write back protocol is used, write through protocol will be too slow.

**Computer Communication:**

Transmission of data (voice, video or text) via electrical, optical or wireless link between computers or network processors is referred to as Data Communication. In computer communication, the data is generally digital. Mostly, the data that required to be communicated is in analog form. So an analog to digital converter (ADC) will be required to convert the analog data into digital data.

Digital data is preferred as it is easy to mix signals and data using digital techniques, is less noise prone and good processing techniques such as compression, error correction, equalization and encryption are available for digital data.

Components of a communication system are

- **Source (Transmitter):** Communication originates from here.
- **Medium (Communication Link):** The channel through which communication takes place.
- **Sink (Receiver):** Communication terminates here.
- **Communication networks:** Interconnection of components of the system.
- **Communication rules (protocols):** Set of rules that govern the network.

Transmission directions are described as under.

- **Simplex:** One way communication e.g. TV transmission
- **Half Duplex:** Two way communication but not simultaneously, e.g. walkie talkie
- **Full Duplex:** Simultaneous two way communication e.g. telephone.

**Multiplexing:**

More than one signals use same channel simultaneously for communication thus making it cost effective. Types of Multiplexing are

- **Frequency Division Multiplexing (FDM)**
- **Time Division Multiplexing (TDM)**
- **Wavelength Division Multiplexing**
- **Code Division Multiplexing**
Frequency Division Multiplexing:

Non-overlapping frequency ranges are assigned to each user or signal in a medium. Signals are transmitted simultaneously but use different frequencies. This is oldest multiplexing technique and is mostly used in analog domain. Unfortunately, it is more susceptible to noise. Examples are Broadcast Radio and Television, Cable TV, and AMPS cell phone system.

Time Division Multiplexing:

Available time of the channel is shared by the users. Digital signals mostly use TDM. It has two basic types: synchronous Time Division Multiplexing and statistical TDM. T1/E1 and Integrated Services Digital Network (ISDN) use TDM.

In Synchronous TDM, two sources may generate bits at different rates; multiplexor inserts sequence of 1s and 0s alternatively for the source that is not generating bits. It is very popular and T-1, ISDN and SONET use this technique. Bandwidth requirement is same as that of all sources.

In Statistical TDM, time slots are allocated to active nodes only. It makes efficient use of bandwidth and is good for low bandwidth application.

Wavelength Division Multiplexing:

Each message is given a different frequency and is used in Optical fiber.

Code Division Multiplexing:

Same frequencies are used at the same time but different code is used for different messages and is also called Code Division Multiple Access (CDMA). Mobile phone systems use this technique.

Layered Structure:

Layered structure is used to reduce the complexity of tasks. A Seven layered model Open System Interconnection (OSI) is a popular choice. Each layer has its own job and protocol.

Application Layer: is used to allow access to the network resources.

Presentation Layer enables translation, encryption, and compression of data.

Session Layer establishes, manages and terminates session.

Transport Layer provides reliable process to process message delivery and error recovery.

Network Layer moves packets from source to destination and provides internetworking.

Data Link Layer organizes bits into frames and provides hop to hop delivery.

Physical Layer transmits bits over a medium and provides mechanical and electrical specifications.
Error Detection and Correction:

Data transfer between two devices must be error free. Data can be corrupted during transmission. A reliable communication system must have capability to detect and correct errors. Error detection and correction is implemented at the data link or transport layer of the OSI model. Types of errors are

- Single bit errors
- Multiple bits errors
- Burst Errors

Redundancy is used to detect errors. Error correction may be carried out by re-transmitting or using error correcting codes.

References:


http://www.unimasr.net/ums/upload/files/2012/Dec/UniMsr.com_e9bde9be7734d89bb34f85d7d2c05a35.pdf

http://mediafire.com/?kmn34f2uqmd4kpz

Helpful websites

http://faculty.kfupm.edu.sa/COE/aimane/assembly/pagegen.aspx-themeid=1&m300_0.htm

http://faculty.kfupm.edu.sa/COE/aimane/assembly/pagegen.aspx-ThemeID=1&m010_0.htm

Glossary