Lecture No. 19

# Pipelined SRC

## Reading Material

Vincent P. Heuring & Harry F. Jordan — Chapter 5
Computer Systems Design and Architecture — 5.1.3

*Summary*

- Pipelined Version of the SRC: 5 stages of pipeline
- Adapting SRC instructions for Pipelined Execution
- Control Signals for Pipelined SRC: multiplexer

## Pipelined Version of the SRC

In this lecture, a pipelined version of the SRC is presented. The SRC uses a five-stage pipeline. Those five stages are given below:

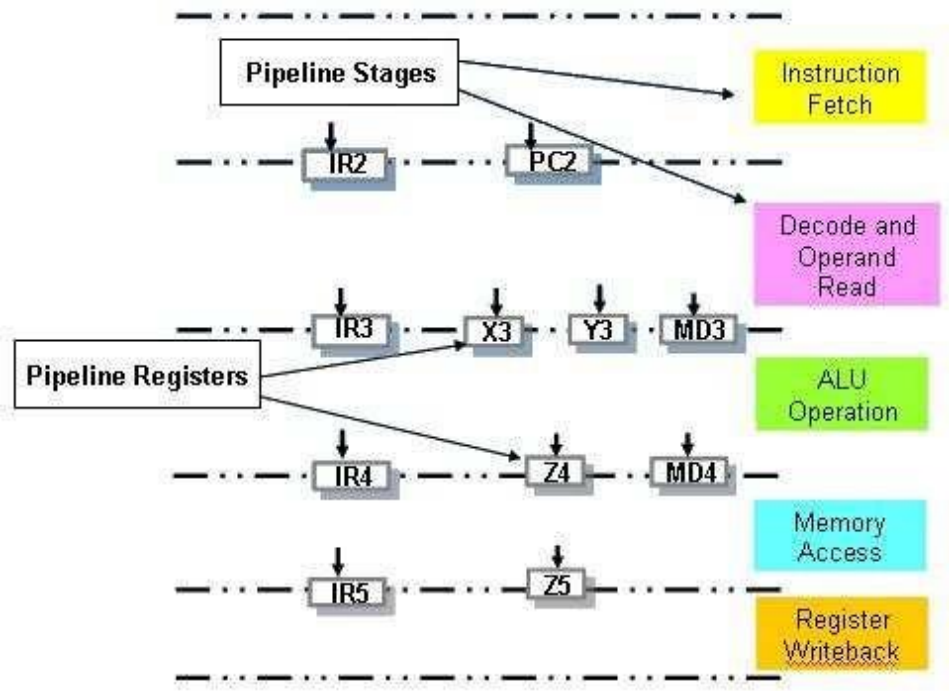
1. Instruction Fetch
2. Instruction decode/operand fetch
3. ALU operation
4. Memory access
5. Register write

As shown in the next diagram, there are several registers between each stage.

After the instruction has been fetched, it is stored in IR2 and the incremented value of the program counter is held in PC2. When the register values have been read, the first register value is stored in X3, and the second register value is stored in Y3. IR3 holds the opcode and ra. If it is a store to memory instruction, MD3 holds the register value to be stored.

After the instruction has been executed in the ALU, the register Z4 holds the result. The op-code and ra are passed on to IR4. During the write back stage, the register Z5 holds the value to be stored back into the register, while the op-code and ra are passed into IR5. There are also two separate memories and several multiplexers involved in the pipeline operation. These will be shown at appropriate places in later figures.

The number after a particular register name indicates the stage where the value of this register is used.




## Adapting SRC Instructions for Pipelined Execution

As mentioned earlier, the SRC instructions fall into the following three categories:

1. ALU Instructions
2. Load/Store instructions
3. Branch Instructions

We will now discuss how to design a common pipeline for all three categories of instructions.

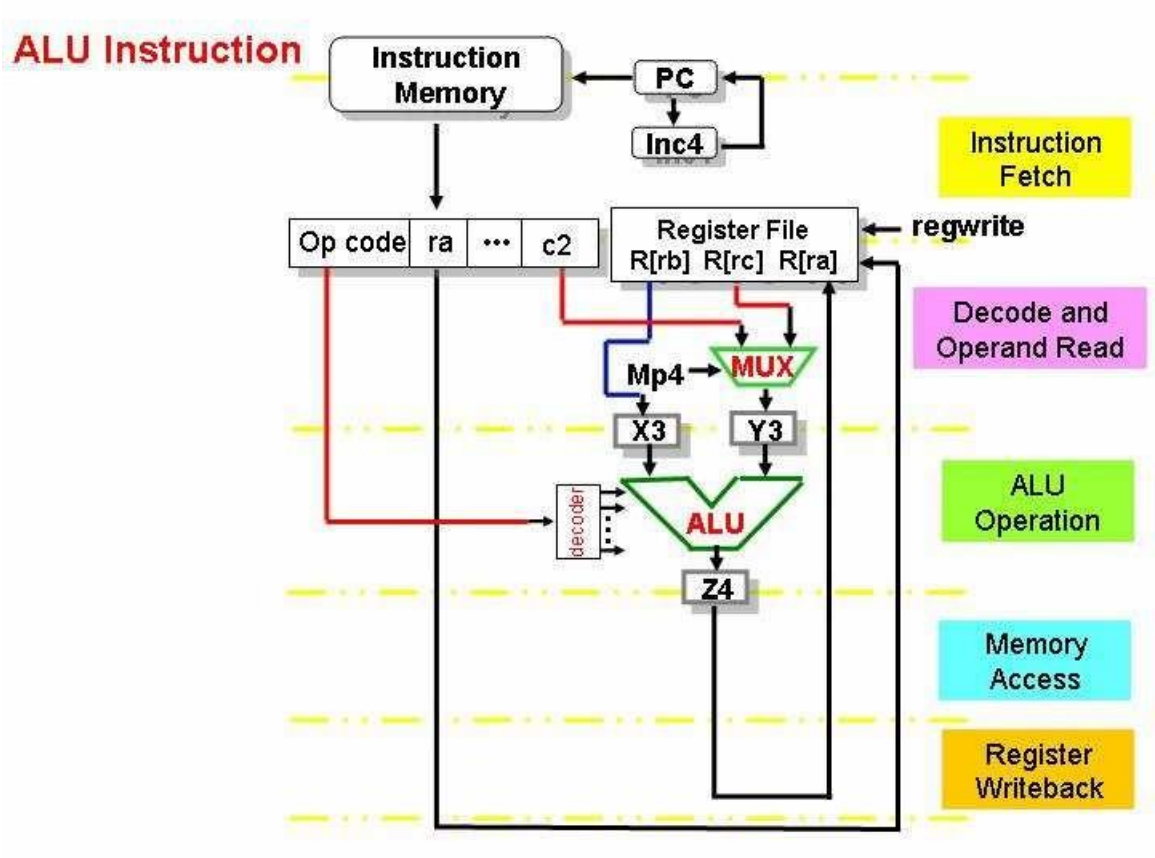
1. ALU instructions

ALU instructions are usually of the form:

op-code ra, rb, rc
or
op-code ra, rb, constant.

In the diagram shown, X3 and Y3 are temporary registers to hold the values between pipeline stages. X3 is loaded with operand value from the register file. Y3 is loaded with either a register value from the register file or a constant from the instruction. The operands are then available to the ALU. The ALU function is determined by decoding the op-code bits. The result of the ALU operation is stored in register Z4 and then stored in the destination register in the register write back stage. There is no activity in the memory access stage for ALU instructions. Note that Z5, IR3, IR4, and IR5 are not shown explicitly in the figure. The purpose of not including these registers is to keep the drawing simple. However, these registers will transfer values as

instructions progress through the pipeline. This comment also applies to some other figures in this discussion.

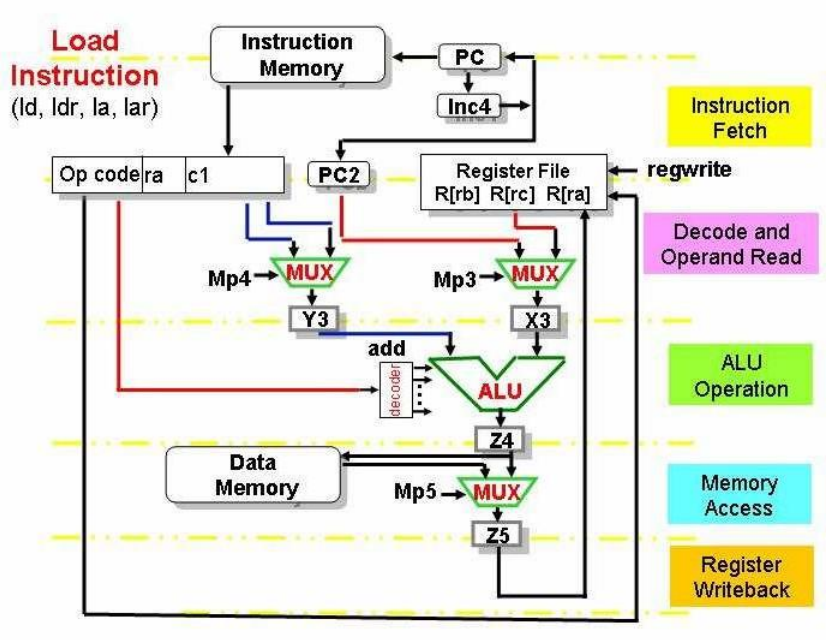



2. Load/Store instructions

Load/Store instructions are usually of the form:

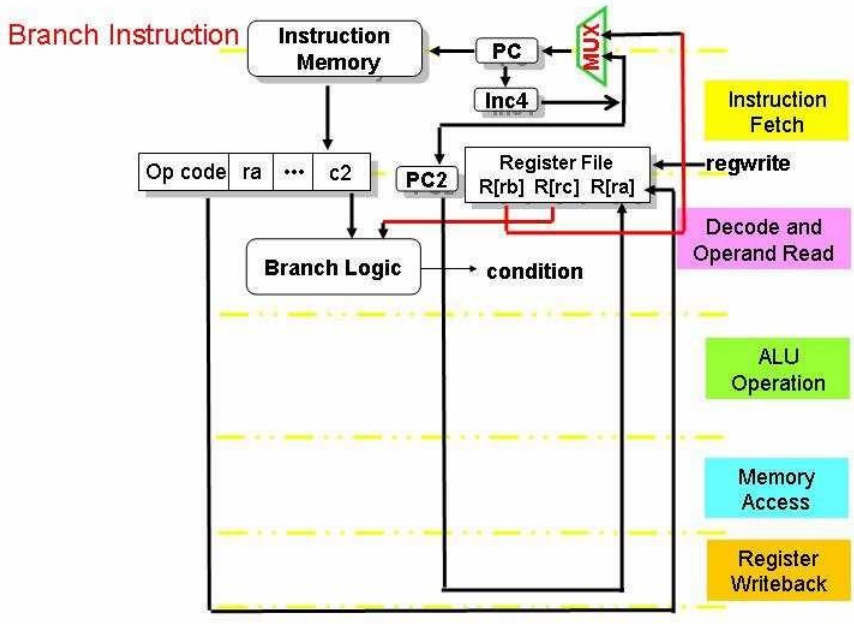
op-code ra, constant(rb)

The instruction is loaded into IR2 and the incremented value of the PC is loaded in PC2. In the next stage, X3 is loaded with the value in PC2 if the relative addressing mode is used, or the value in rb if the displacement addressing mode is used. Similarly, C1 is transferred to Y3 for the relative addressing mode, and c2 is transferred to Y3 for the displacement addressing mode. The store instruction is completed once memory access has been made and the memory location has been written to. The load instruction is completed once the loaded value is transferred back to the register file. The following figure shows the schematic for a load instruction. A similar schematic can be drawn for the store instruction.




3. Branch Instructions

Branch Instructions usually involve calculating the target address and evaluating a condition. The condition is evaluated based on the c2 field of the IR and by using the value in R[rc]. If the condition is true, the PC is loaded with the value in R[rb], otherwise it is incremented by 4 as usual. The following figure shows these details.

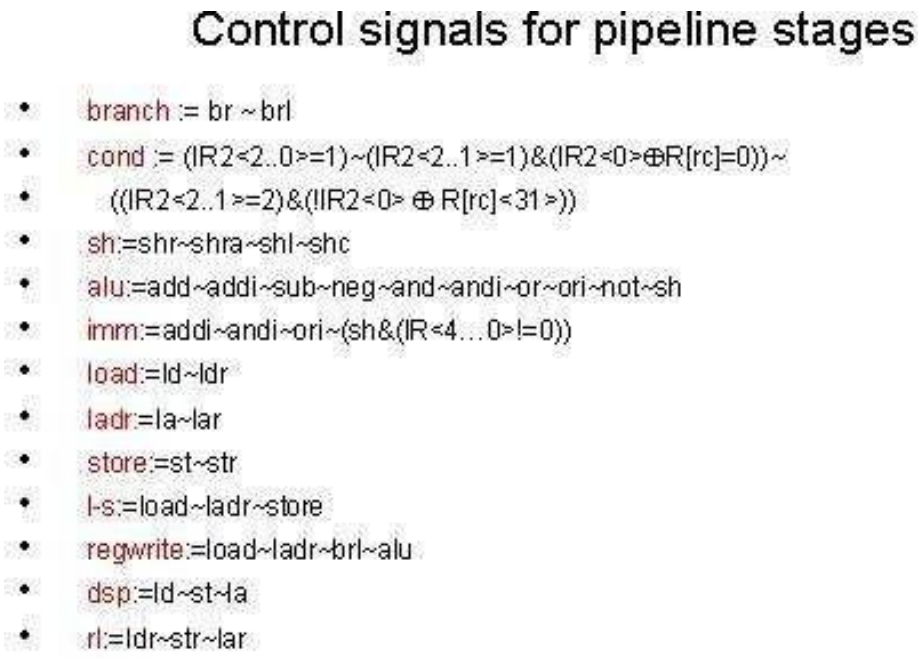



The complete pipelined data path

The pipelined data path implementation diagrams shown earlier for the three SRC instruction categories must be combined and refined to get a working system. These details get complicated very quickly. A detailed combined diagram is shown in Figure 5.7 of the text book.

## Control Signals for the Pipelined SRC

We define the following signals for the SRC by grouping similar op-codes:




In most cases, the signals defined above are used in the same stage where they are generated. If that is not the case, a number used after the signal name indicates the stage where the signal is generated.

Using these definitions, we can develop RTL statements for describing the pipeline activity as well as the equations for the multiplexer select signals for different stages of the pipeline. This is shown in the next diagram.

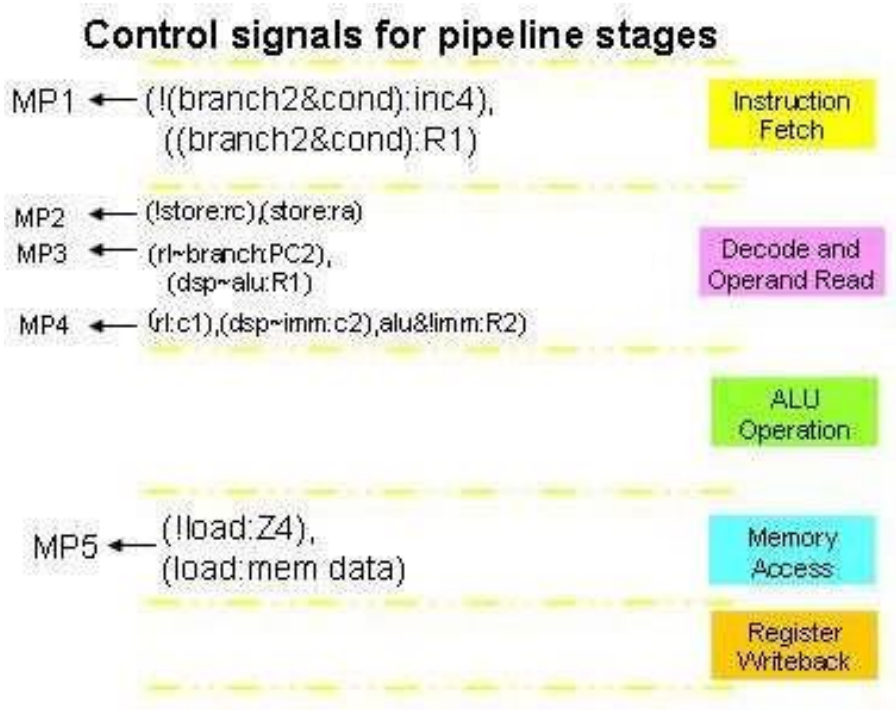
Control Signals for different pipeline Stages

Consider the RTL description of the Mp1 signal, which controls the input to the PC. It simply means that if the branch and cond signals are not activated, then the PC is incremented by 4, otherwise if both are activated then the value of R1 is copied in to the PC.

The multiplexer Mp2 is used to decide which registers are read from the register file. If the store signal is activated then R[rb] from the instruction bits is read from the register file so that its value may be stored into memory, otherwise R[rc] is read from the register file.

The multiplexer Mp3 is used to decide which registers are read from the register file for operand 2. If either rl or branch is activated then the updated value of PC2 is transferred to X3, otherwise if dsp or alu is activated, the value of R[ra] from the register file is transferred to the x3. In the same way, multiplexer Mp4 is used to select an input from Y3.

In the same way, multiplexer Mp4 is used to select an input for Y3.




The multiplexer MP5 s used to decide which value is transferred to be written back to the register file. If the load signal is activated data from memory is transferred to Z5, however if the load signal is not activated then data from Z4 (which is the result of ALU) is transferred to Z5 which is then written back to the register file.

Lecture No. 20

# Hazards in Pipelining

## Reading Material

Vincent P. Heuring & Harry F. Jordan Computer Systems Design and Architecture — Chapter 5, 5.1.5, 5.1.6

## Summary

- Structural RTL for Pipeline Stages
- Instruction Propagation Through the Pipeline
- Pipeline Hazards
- Data Dependence Distance
- Data Forwarding
- Compiler Solution to Hazards
- SRC Hazard Detection and Correction
- RTL for Hazard Detection and Pipeline Stall

## Structural RTL for Pipeline Stages

The Register Transfer Language for each phase is given as follows:

Instruction Fetch

IR2 ← M [PC];
PC2 ← PC+4;

Instruction Decode & Operand fetch

X3←l-s2:(rel2:PC2, disp2:(rb=0):?, (rb =0):R[rb]), brl2:PC2, alu2:R[rb],
Y3 ← l-s2:(rel2:c1, disp2:c2), alu2:(imm2:c2, !imm2:R[rc]),
MD3 ←store2:R[ra], IR3 ← IR2, stop2:Run ← 0,
PC ← !branch2:PC+4, branch2:(cond(IR2, R[rc]):R[rb], !cond(IR2, R[rc]):PC+4;

ALU operation

Z4 ← (I-s3: X3+Y3, brl3: X3, Alu3: X3 op Y3,
MD4 ← MD3,
IR4 ← IR3;

Memory access

Z5 ← (load4: M [Z4], ladr4~branch4~alu4:Z4),
store4: (M [Z4] ← MD4),
IR5 ←IR4;

Write back

regwrite5: (R[ra] ← Z5);

## Instruction Propagation through the Pipeline

Consider the following SRC code segment flowing through the pipeline. The instructions along with their addresses are

```
200:    add r1, r2, r3
204:    ld r5, [4(r7)
208:    br r6
212:    str r4, 56
…
400
```

We shall review how this chunk of code is executed.

### First Clock Cycle

Add instruction enters the pipeline in the first cycle. The value in PC is incremented from 200 to 204.

### Second Clock Cycle

Add moves to decode stage. Its operands are fetched from the register file and moved to X3 and Y3 at the end of clock cycle, meanwhile the Instruction ld r5, [4+r7] is fetched in the first stage and the PC value is incremented from 204 to 208.

### Third Clock Cycle

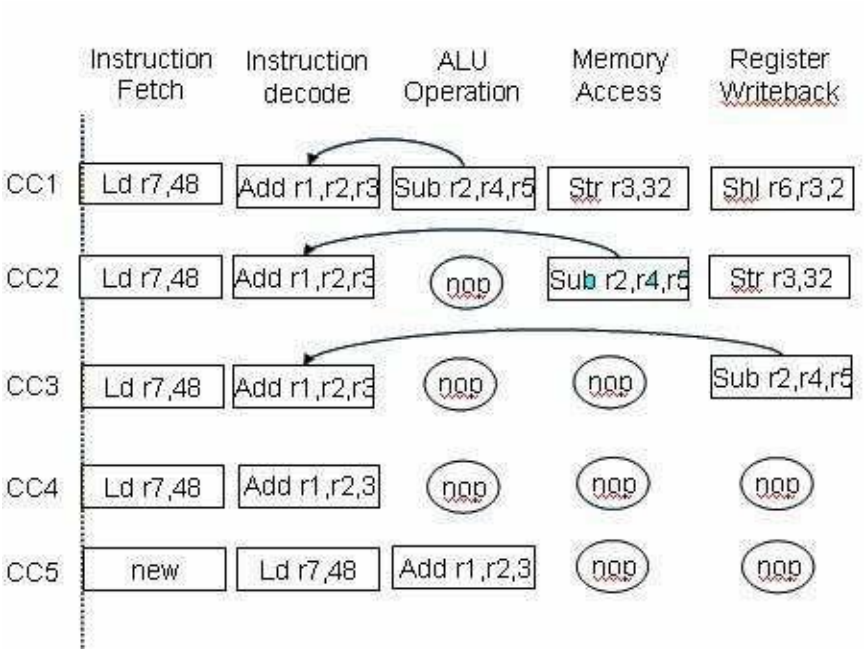
Add instruction moves to the execute stage, the results are written to Z4 on the trailing edge of the clock. Ld instruction moves to decode stage. The operands are fetched to calculate the displacement address. Br instruction enters the pipeline. The value in PC is incremented from 208 to 212.

### Fourth Clock Cycle

Add does not access memory. The result is written to Z5 at the trailing edge of clock. The address is being calculated here for ld. The results are written to Z4. Br is in the decode stage. Since this branch is always true, the contents of PC are modified to new address. Str instruction enters the pipeline. The value in PC is incremented from 212 to 216.

### Fifth Clock Cycle

The result of addition is written into register r1. Add instruction completes. Ld accesses data memory at the address specified in Z4 and result stored in Z5 at falling edge of clock. Br instruction just propagates through this stage without any calculation. Str is in the decode stage. The operands are being fetched for address calculation to X3 and Y3. The instruction at address 400 enters the pipeline. The value in PC is incremented from 400 to 404.




## Pipeline Hazards

The instructions in the pipeline at any given time are being executed in parallel. This parallel execution leads to the problem of instruction dependence. A hazard occurs when an instruction depends on the result of previous instruction that is not yet complete.

### Classification of Hazards

There are three categories of hazards

1. Branch Hazard
2. Structural Hazard
3. Data Hazard

### Branch hazards

The instruction following a branch is always executed whether or not the branch is taken. This is called the branch delay slot. The compiler might issue a nop instruction in the branch delay slot. Branch delays cannot be avoided by forwarding schemes.

### Structural hazards

A structural hazard occurs when attempting to access the same resource in different ways at the same time. It occurs when the hardware is not enough to implement pipelining properly e.g. when the machine does not support separate data and instruction memories.

### Data hazards

Data hazard occur when an instruction attempts to access some data value that has not yet been updated by the previous instruction. An example of this RAW (read after write) data hazard is;

```
200:    add r2, r3, r4
204:    sub r7, r2, r6
```

The register r2 is written in clock cycle 5 hence the sub instruction cannot proceed beyond stage 2 until the add instruction leaves the pipeline.

Data Hazard Detection & Correction

Data hazards can be detected easily as they occur when the destination register of an instruction is the same as the source register of another instruction in close proximity. To remedy this situation, dependent instructions may be delayed or stalled until the ones ahead complete. Data can also be forwarded to the next instruction before the current instruction completes, however this requires forwarding hardware and logic. Data can be forwarded to the next instruction from the stage where it is available without waiting for the completion of the instruction. Data is normally required at stage 2 (operand fetch) however data is earliest available at stage 3 output (ALU result) or stage 4 output (memory access result). Hence the forwarding logic should be able to transfer data from stage 3 to stage 2 or from stage 4 to stage 2

## Data Dependence Distance

Designing a data forwarding unit requires the study of dependence distances. Without forwarding, the minimum spacing required between two data dependent instructions to avoid hazard is four. The load instruction has a minimum distance of two from all other instructions except branch. Branch delays cannot be removed even with forwarding.

Table 5.1 of the text shows numbers related to dependence distances with respect to some important instruction categories.

## Compiler Solution to Hazards

Hazards can be detected by the compiler, by analyzing the instruction sequences and dependencies. The compiler can inserts bubbles (nop instruction) between two instructions that form a hazard, or it could reorder instructions so as to put sufficient distance between dependent instructions. The compiler solution to hazards is complex, expensive and not very efficient as compared to the hardware solution.

## SRC Hazard Detection and Correction

The SRC uses a hazard detection unit. The hazard can be resolved using either pipeline stalls or by data forwarding.

## Pipeline stalls

Consider the following sequence of instructions going through the SRC pipeline

```
200:    shl r6, r3, 2
204:    str r3, 32
208:    sub r2, r4,r5
212:    add r1,r2,r3
216:    ld r7, 48
```

There is a data hazard between instruction three and four that can be resolved by using pipeline stalls or bubbles

When using pipeline stalls, nop instructions are placed in between dependent instructions. The logic behind this scheme is that if opcode in stage 2 and 3 are both alu, and if ra in stage 3 is the same as rb or rc in stage 2, then a pause signal is issued to insert a bubble between stage 3 and 2. Similar logic is used for detecting hazards between stage 2 and 4 and stage 4 and 5.

## Data Forwarding

By adding data forwarding mechanism to the SRC data path, the stalls can be completely eliminated at least for the ALU instructions. The hazard detection is required between stages 3 and 4, and between stages 3 and 5. The testing and forwarding circuits employ wider IRs to store the data required in later stages. The logic behind this method is that if the ALU is activated for both 3 and 5 and ra in 5 is the same as rb in 3 then Z5 which hold the currently loaded or calculated result is directly forwarded to X3. Similarly, if both are ALU operations and instruction in stage 3 does not employ immediate operands then value of Z5 is transferred to Y3. Similar logic is used to forward data between stage 3 and 4.

## RTL for Hazard Detection and Pipeline Stall

The following RTL expression detects data hazard between stage 2 and 3, then stalls stage 1 and 2 by inserting a bubble in stage 3

alu3&alu2&((ra3=rb2)~((ra3=rc2)&!imm2)):
(pause2, pause1, op3←0)

## Meaning:

If opcode in stage 2 and 3 are both ALU, and if ra in stage 3 is same as rb or rc in stage 2, issue a pause signal to insert a bubble between stage 3 and 2.

Following is the complete RTL for detecting hazards among ALU instructions in different stages of the pipeline

| Data Hazard (between) | RTL for detection and stalling |
|---|---|
| Stage 2 and 3 | alu3&alu2&((ra3=rb2)~((ra3=rc2)&!imm2)):<br>(pause2, pause1, op3←0) |
| Stage 2 and 4 | alu4&alu2&((ra4=rb2)~((ra4=rc2)&!imm2)):<br>(pause2, pause1, op3←0) |
| Stage 2 and 5 | alu5&alu2&((ra5=rb2)~((ra5=rc2)&!imm2)):<br>(pause2, pause1, op3←0) |

# Lecture No. 21

Instruction Level Parallelism

## Reading Material

Vincent P. Heuring & Harry F. Jordan Computer Systems Design and Architecture — Chapter 5, 5.2

*Summary*

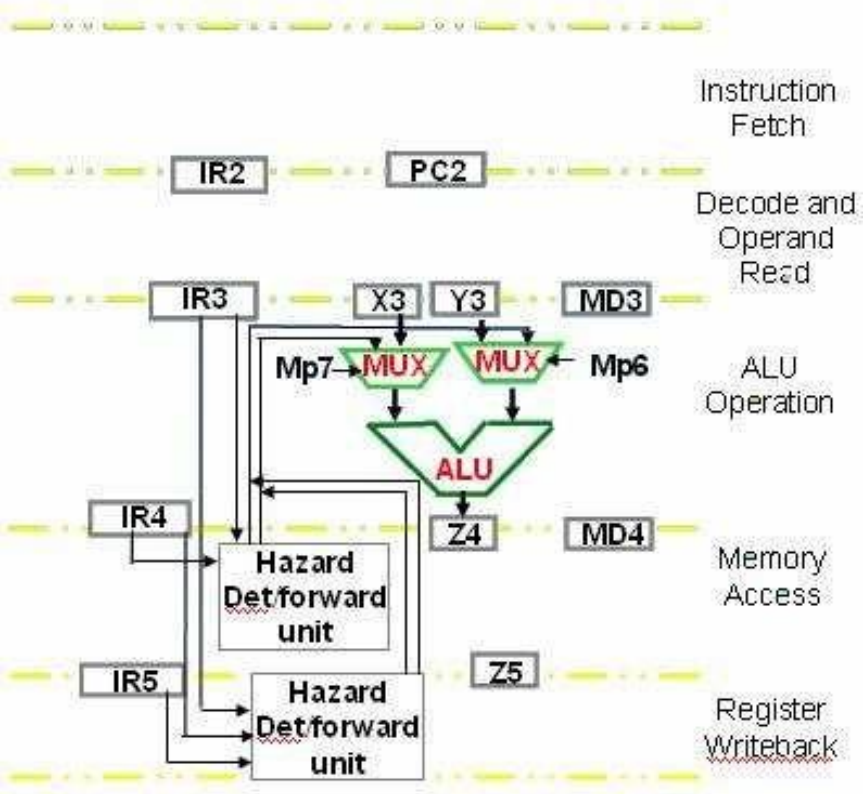
- Data Forwarding Hardware
- Instruction Level Parallelism
- Difference between Pipelining and Instruction-Level Parallelism
- Superscalar Architecture
- Superscalar Design
- VLIW Architecture

Maximum Distance between two instructions
Example

Read page no. 219 of Computer System Design and Architecture (Vincent P.Heuring, Harry F. Jordan)

## Data forwarding Hardware

The concept of data forwarding was introduced in the previous lecture.




RTL for data forwarding in case of ALU instructions

| Dependence | RTL |
|---|---|
| Stage 3-5 | alu5&alu3:((ra5=rb3):X←Z5, |
| | (ra5=rc3)&!imm3: Y ← Z5); |
| Stage 3-4 | alu4&alu3:((ra4=rb3):X←Z4, |
| | (ra4=rc3)&!imm3: Y ← Z4); |

## Instruction-Level Parallelism

Instruction-level parallelism (ILP) is the parallel execution of a sequence of instructions.

Increasing a processor's throughput

*Important*

There are two ways to increase the number of instructions executed in a given time by a processor

- By increasing the clock speed
- By increasing the number of instructions that can execute in parallel

Increasing the clock speed

- Increasing the clock speed is an IC design issue and depends on the advancements in chip technology.
- The computer architect or logic designer can not thus manipulate clock speeds to increase the throughput of the processor.

Increasing parallel execution of instructions

The computer architect cannot increase the clock speed of a microprocessor however he/she can increase the number of instructions processed per unit time. In pipelining we discussed that a number of instructions are executed in a staggered fashion, i.e. various instructions are simultaneously executing in different segments of the pipeline. Taking this concept a step further we have multiple data paths hence multiple pipelines can execute simultaneously. There are two main categories of these kinds of parallel instruction processors VLIW (very long instruction word) and superscalar.

The two approaches to achieve instruction-level parallelism are

- Superscalar Architecture
  A scalar processor that can issue multiple instructions simultaneously is said to be superscalar
- VLIW Architecture
  A VLIW processor is based on a very long instruction word. VLIW relies on instruction scheduling by the compiler. The compiler forms instruction packets which can run in parallel without dependencies.

## Difference between Pipelining and Instruction-Level Parallelism

| Pipelining | Instruction-Level Parallelism |
|---|---|
| Single functional unit | Multiple functional units |
| Instructions are issued sequentially | Instructions are issued in parallel |
| Throughput increased by overlapping the instruction execution | Instructions are not overlapped but executed in parallel in multiple functional units |
| Very little extra hardware required to implement pipelining | Multiple functional units within the CPU are required |

## Superscalar Architecture *Important*

A superscalar machine has following typical features

- It has one or more IUs (integer units) , FPUs (floating point units), and BPUs (branch prediction units)
- It divides instructions into three classes
  - Integer
  - Floating point
  - Branch prediction

The general operation of a superscalar processor is as follows

- Fetch multiple instructions
- Decode some of their portion to determine the class
- Dispatch them to the corresponding functional unit

As stated earlier the superscalar design uses multiple pipelines to implement instruction level parallelism.

### Operation of Branch Prediction Unit

- BPU calculates the branch target address ahead of time to save CPU cycles
- Branch instructions are routed from the queue to the BPU where target address is calculated and supplied when required without any stalls
- BPU also starts executing branch instructions by speculating and discards the results if the prediction turns out to be wrong

## Superscalar Design

The philosophy behind a superscalar design is

- to prefetch and decode as many instructions as possible before execution
- and to start several branch instruction streams speculatively on the basis of this decoding
- and finally, discarding all but the correct stream of execution

The superscalar architecture uses multiple instruction issues and uses techniques such as branch prediction and speculative instruction execution, i.e. it speculates on whether a particular branch will be taken or not and then continues to execute it and the following instructions. The results are not written back to the registers until the branch decision is confirmed. Most superscalar architectures contain a reorder buffer. The reorder buffer acts like an intermediary between the processor and the register file. All results are written onto the reorder buffer and when the speculated course of action is confirmed, the reorder buffer is committed to the register file.

## Superscalar Processors

Examples of superscalar processors

- PowerPC 601
- Intel P6
- DEC Alpha 21164

## VLIW Architecture

VLIW stands for "Very Long Instruction Word" typically 64 or 128 bits wide. The long instruction word carries information to route data to register files and execution units. The execution-order decisions are made at the compile time unlike the superscalar design where decisions are made at run time. Branch instructions are not handled very efficiently in this architecture . VLIW compiler makes use of techniques such as loop unrolling and code reordering to minimize dependencies and the occurrence of branch instructions.

Lecture No. 22

# Microprogramming

## Reading Material

Vincent P. Heuring & Harry F. Jordan — Chapter 5
Computer Systems Design and Architecture — 5.3

*Summary*

- Microprogramming
- Working of a General Microcoded Controller
- Microprogram Memory
- Generating Microcode for Some Sample Instructions
- Horizontal and Vertical Microcode Schemes
- Microcoded 1-bus SRC Design
- The SRC Microcontroller

## Microprogramming

In the previous lectures, we have discussed how to implement logic circuitry for a control unit based on logic gates. Such an implementation is called a hardwired control unit. In a micro programmed control unit, control signals which need to be generated at a certain time are stored together in a control word. This control word is called a microinstruction. A collection of microinstructions is called a microprogram. These microprograms generate the sequence of necessary control signals required to process an instruction. These microprograms are stored in a memory called the control store.

As described above microprogramming or microcoding is an alternative way to design the control unit. The microcoded control unit is itself a small stored program computer consisting of

- Micro-PC
- Microprogram memory
- Microinstruction word

Comparison of hardwired and microcoded control unit

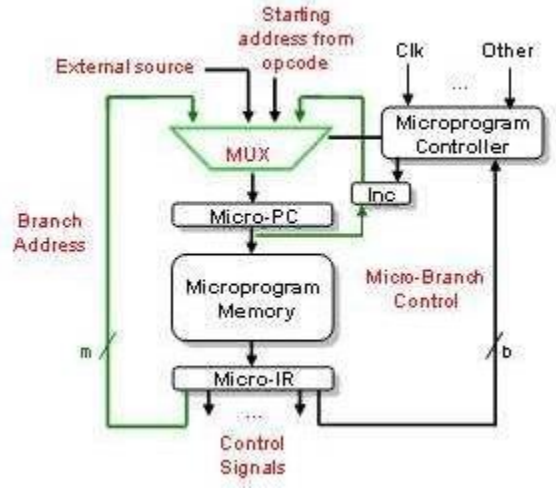
| Hardwired Control Unit | Microcoded Control Unit |
|---|---|
| The relationship between control inputs and control outputs is a series of Boolean functions. | The control signals here are stored as words in a microcode memory. |
| Hardwired control units are generally faster. | Microcode units simplify the computer logic but it is comparatively slower. |

Working of a general microcoded controller

A microcoded controller works in the same way as a small general purpose computer.

1. Fetch a micro-instruction and increment micro-PC.
2. Execute the instruction present in micro-IR.
3. Fetch the next instruction and so on…

The microcoded control unit is like a small computer in itself. It consists of a microprogram memory, which is read using a micro program counter. The micro PC is controlled by the microprogram controller. Values of the micro PC depends on a 4 to 1 MUX. The source may be the incremented micro PC value, or a calculated branch value, or a value derived by decoding an opcode for an instruction. The microprogram memory writes the control word at the chosen address into the micro instruction register. This control word is basically the set of all the control signals needed to execute the instruction at that particular instant.

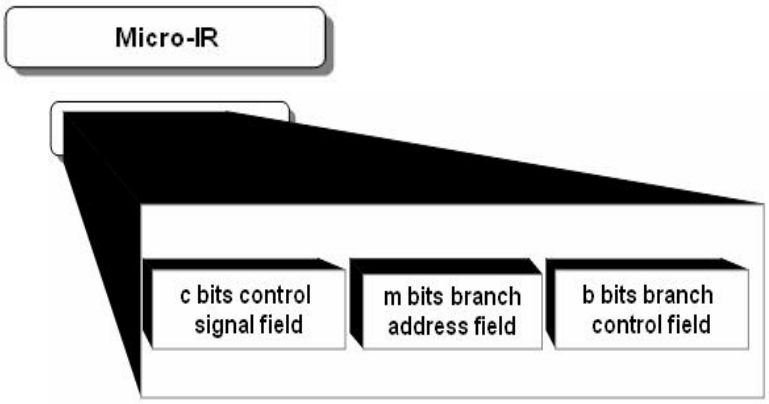



Fields in the micro instruction

C Bits
These form the control signal field

M Bits
These form the branch address field

B Bits
These form the branch control field.




Loading the micro-PC
The micro-PC can be loaded from one of the four possible sources

- Simple increment Steps sequentially from microinstruction to microinstruction
- Lookup table A lookup table maps the opcode field to the starting address of the microcode routine that generates control signals.
- External source Initializes micro-PC to begin an operation e.g. interrupts service, reset etc.
- Branch addresses Jumps anywhere in the microprogram memory on the basis of conditional or unconditional branch.

## Microprogram Memory

- This small memory contains micro routines for all the instructions in the ISA
- The micro-PC supplies the address and it returns the control word stored at that address
- It is much faster and smaller than a typical main memory

Layout of a typical microprogram memory

| Micro-Address | Memory Contents |
|---|---|
| 0 | Microcode for instruction fetch |
| | |
| … | Microcode for load instruction |
| | |
| … | Microcode for add instruction |
| | |
| … | Microcode for br instruction |
| | |
| $2^n$-1 | Microcode for reset instruction |

## Generating Microcode for Some Sample Instructions

- The control word for an instruction is used to generate the equivalent microcode sequence
- Each step in RTL corresponds to a microinstruction executed to generate the control signals.

Each bit in the control words in the microprogram memory represents a control signal.
The value of that bit decides whether the signal is to be activated or not.

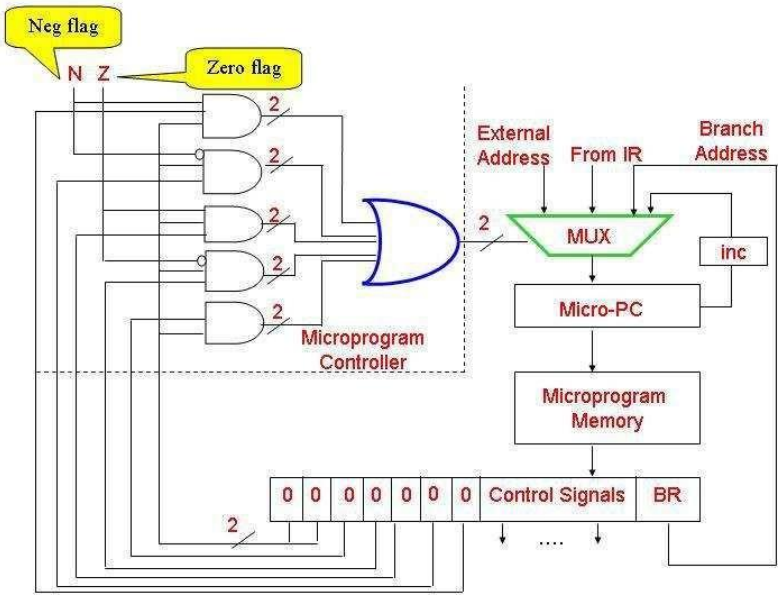
Example: Control Signals for the sub Instruction

The first three addresses from 100 to 102 represent microcode for instruction fetch and the last three addresses from 203 to 205 represent microcode for sub instruction. In the first cycle at address 100, the control signal PCout, LMAR, LC, and INC4 are activated and all other signals are deactivated. All these control signals are for the SRC processor. So, if the micro-PC contains 100, the contents of microprogram memory are copied into the micro IR. This corresponds to the structural RTL description of the T0 clock during instruction fetch phase. In the same way, the content of address 101 corresponds to T1, and the content of address 102 corresponds to T2.

| Address | Branch Control | PC out | Cout | Cout | R2Bus | LMAR | LC | LPC | LIR | LA | Bus2R | INC4 | Read | LMBR | MARout | SUB | RAE | RBE | RCE | END |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 100 | … | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 101 | … | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 102 | … | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 203 | … | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 204 | … | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 205 | … | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |

Microprogram Controller functions: Branching and looping

- Microprogram controller controls the sequence of the flow of microinstructions.
- The inputs to the microcontroller are from the branch control fields specified in the microcode word.
- Its output controls the 4 to 1 multiplexer inside the microcoded control unit.
- It implements conditional execution and both conditional and unconditional branch




If a branch instruction is encountered within the microprogram hardwired logic selects the branch address as the source of micro-PC using 4 to 1 mux. This hardwired logic caters for all branch instructions including branch if zero.

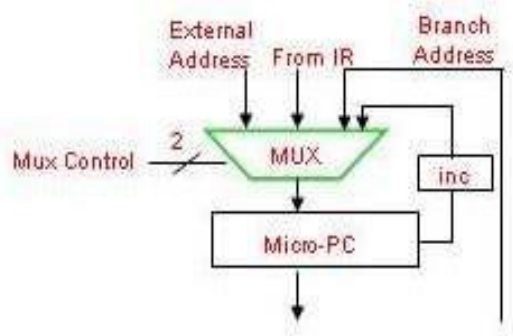
4-1 Multiplexer

The multiplexer supplies one of the four possible values to the micro-PC

The incremented value of the micro-PC is used when dealing with the normal flow of microinstructions.

The opcode from the instruction is used to set the micro-PC when a microroutine is initially being loaded.

| Mux Control | Select |
|---|---|
| 00 | Increment micro-PC |
| 01 | Opcode from IR |
| 10 | External address |
| 11 | Branch address |




External address is used when it is required to reset the microprogram controller.
Branch address is set into the micro-PC when a branch microinstruction is encountered.

How to form a branch

- A branch can be implemented by choosing one alternative from each of the following two lists.
- This scheme provides flexibility in choosing branches as we can form any combination of conditions and addresses.

| Condition |
|---|
| unconditional |
| not zero |
| zero |
| positive |
| negative |

| Address |
|---|
| From IR |
| External Address |
| Branch Address |

- Microcode Branching Examples

Following is an example of branch instructions in microcode

| Address | Mux COntrol | Branch | brnz | brz | brp | brn | Control Signals | Branch Address | Branching Action | Equivalent C construct |
|---|---|---|---|---|---|---|---|---|---|---|
| 400 | 00 | 0 | 0 | 0 | 0 | 0 | … | xxx | No branch,goto next address in sequence-401 | {…}; |
| 401 | 01 | 1 | 0 | 0 | 0 | 0 | … | xxx | To the address supplied by opcode | {…}; goto initial address; |
| 402 | 10 | 0 | 0 | 1 | 0 | 0 | … | xxx | To external address if Z flag is set | {…}; if Z then goto Ext. Add. |
| 403 | 11 | 0 | 0 | 0 | 0 | 1 | … | 200 | To 200 if N flag is set, else to 404 | {…}; if N then goto Label1; |
| 404 | 11 | 0 | 0 | 0 | 1 | 0 | 000 | 406 | To 406 if N is false, else to 405 | While (N) {...}; |
| 405 | 11 | 1 | 0 | 0 | 0 | 0 | … | 404 | Branch to 404 | While contd… |

Similarity between microcode and high level programs

- Any high level construct such as if-else, while, repeat etc. can be implemented using microcode
- A variety of microcode compilers similar to the high level compilers are available that allow easier programming in microcode
- This similarity between high level language and microcode simplifies the task of controller design.

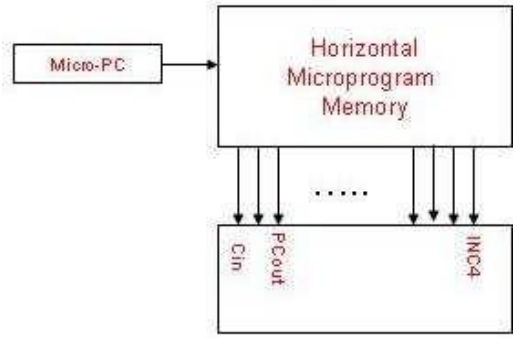
Horizontal and vertical microcode schemes

In horizontal microcode schemes, there are no intermediate decoders and the control word bits are directly connected to their destination i.e. each bit in the control word is directly connected to some control signal and the total number of bits in the control word is equal to the total number of control signals in the CPU.

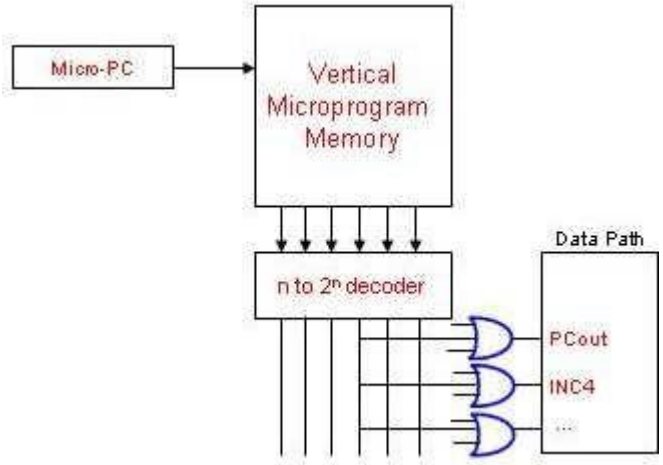
Vertical microcode schemes employ an extra level of decoding to reduce the control word width. From an n bit control word we may have $2^n$ bit signal values.

However, a completely vertical scheme is not feasible because of the high degree of fan out.

Horizontal Microcode Scheme




Vertical Microcode Scheme

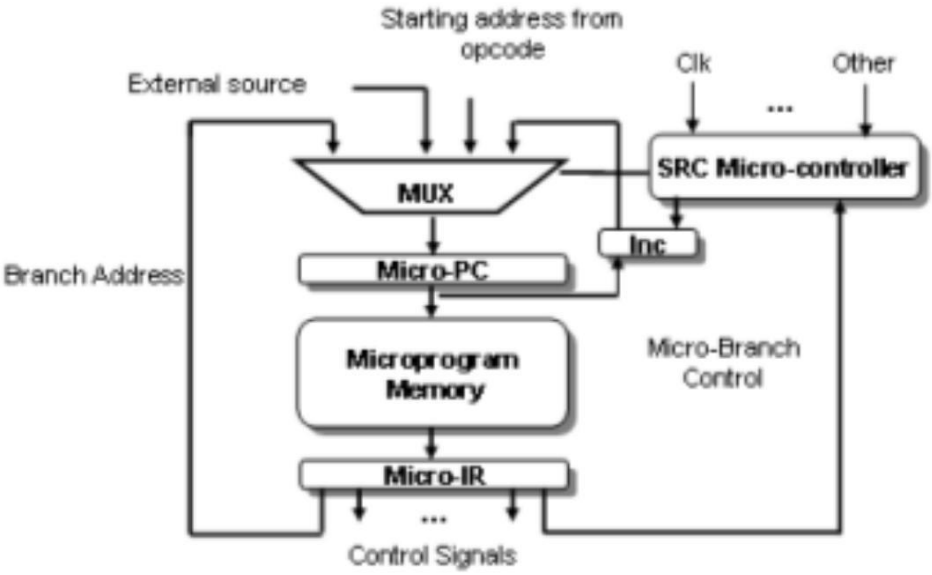


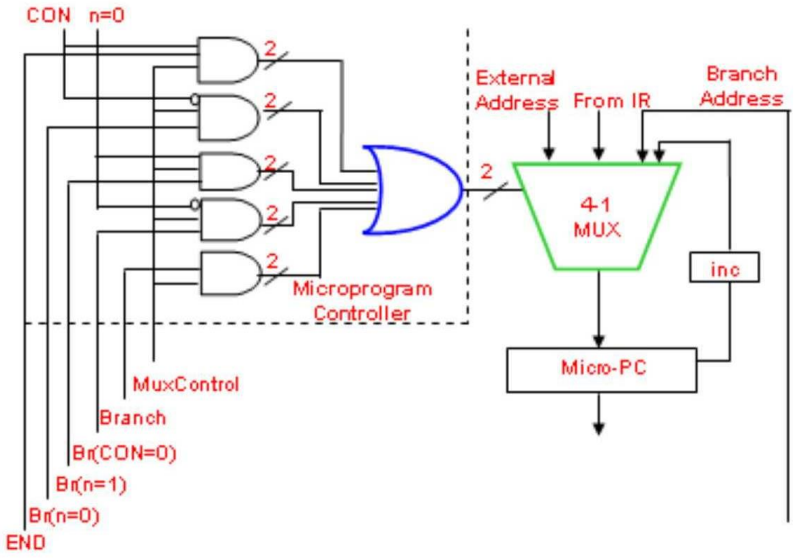

Microcoded 1-bus SRC design

In the SRC the bits from the opcode in the instruction register are decoded to fetch the address of the suitable microroutine from the microprogram memory. The microprogram controller for the SRC microcoded control unit employs the logic for handling exceptions and reset process. Since the SRC does not have any condition codes, we use the CON and n signals instead of N and Z flags to control branches in case of branch if equal to zero or branch if less than instructions.

The SRC Microprogram Controller

- The microprogram controller for the SRC microcoded control unit employs the logic for handling exceptions and reset process
- Since the SRC does not have any condition codes, we use the CON and n signals instead of N and Z flags to control branches




Microcode for some SRC instructions

| Address | MuxControl | Branch | Br(CON=0) | Br(n=1) | Br(n=0) | End | PCout | LMAR | Control Signals | Branch Address | RTL |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 300 | 00 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | … | xxx | MAR← PC: C ← PC + 4; |
| 301 | 00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | … | xxx | MBR← M[MAR]: PC ← C; |
| 302 | 01 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | … | xxx | IR,Micro-PC←MBR<31…27>; |
| 400 | 00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | … | xxx | A ← R[rb]; |
| 401 | 00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | … | xxx | C ← A + R[rc]; |
| 402 | 11 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | … | 300 | R[ra] ← C; Micro-PC ← 300; |

Assume the first control word at address 300. The RTL of this instruction is MAR PC combined with C PC+4. To facilitate these actions the PCout signal bit and the LMAR signal bit are set to one, so that the value of the PC may be written to the internal processor bus and written onto the MAR. The instructions at 300, 301 and 302 form the microcode for instructions fetch. If we examine the RTL we can see all the functionality of the fetch instruction. The value of PC is incremented, the old value of PC is sent to memory, the instruction from the sent address is loaded into memory buffer register. Then the opcode of the fetched instruction is used to invoke the appropriate microroutine.

Alternative approaches to microcoding *Important Question*

- Bit ORing
- Nanocoding
- Writable Microprogram Memory
- Subroutines in Microprogramming

# Lecture No. 23

## I/O Subsystems

**Reading Material**

Vincent P. Heuring & Harry F. Jordan — Chapter 8
Computer Systems Design and Architecture — 8.1, 8.2

***Summary***

- Introduction to I/O Subsystems
- Major Components of an I/O Subsystems
- Computer Interface
- Memory Mapped I/O versus Isolated I/O
- Considerations during I/O Subsystem Design
- Serial and Parallel Transfers
- I/O Buses

## Introduction to I/O Subsystems

This module is about the computer‟s input and output. As we have seen in the case of memory subsystems, that when we use the terms " read" and "write", then these terms are from the CPU‟s point of view. Similarly, when we use the terms "input" and "output" then these are also from the CPU‟s point of view. It means that when we are talking about an input cycle, then the CPU is receiving data from a peripheral device and the peripheral device is providing data. Similarly, when we talk about an output cycle then the CPU is sending data to a peripheral device and the peripheral device is receiving data. I/O Subsystems are similar to memory subsystems in many aspects. For example, both exchange bits or bytes. This transfer is usually controlled by the CPU. The CPU sends address information to the memory and the I/O subsystems. Then these subsystems decode the address and decide which device should be involved in the transfer. Finally the appropriate data is exchanged between the CPU and the memory or the I/O device.
Memory and I/O subsystems differ in the following ways:

1. Wider range of data transfer speed:
   I/O devices can be very slow such as a keyboard in which case the interval between two successive bytes (or keystrokes) can be in seconds. On the other extreme, I/O

devices can be very fast such as a disk drive sending data to the CPU or a stream of packets arriving over a network, in which case the interval between two successive bytes can be in microseconds or even nanoseconds. While I/O devices can have such a wide range of data transfer speed compared to the CPU‟s speed, the case of memory devices is not so. Even if a memory device is slow compared to the CPU, the CPU‟s speed can be made compatible by inserting wait states in the bus cycle.

2. Asynchronous activity:

Memory subsystems are almost always synchronous. This means that most memory transfers are governed by the CPU‟s clock. Generally this is not the case with I/O subsystems. Additional signals, called handshaking signals, are needed to take care of asynchronous I/O transfers.

3. Larger degradation in data quality:

Data transferred by I/O subsystems can carry more noise. As an example, telephone line noise can become part of the data transferred by a modem. Errors caused by media defects on hard drives can corrupt the data. This implies that effective error detection and correction techniques must be used with I/O subsystems.

4. Mechanical nature of many I/O devices:

Many I/O devices or a large portion of I/O devices use mechanical parts which inherently have a high failure rate. In case an I/O device fails, interruptions in data transfer will occur, reducing the throughput. As an example, if a printer runs out of paper, then additional bytes cannot be sent to it. The CPU‟s data should be buffered (or kept in a temporary place) till the paper is supplied to the printer, otherwise the CPU will not be able to do anything else during this time.

To deal with these differences, special software programs called device drivers are made a part of the operating system. In most cases, device drivers are written in assembly language.

You would recall that in case of memory subsystems, each location uses a unique address from the CPU‟s address space. This is generally not the case with I/O devices. In most cases, a group or block of contiguous addresses is assigned to an I/O device, and data is exchanged byte-by-byte. Internal buffers (memory) within the device store this data if needed.

In the past, people have paid a lot of attention to improve the CPU‟s performance, as a result of which the performance improvement of I/O subsystems was ignored. (I/O subsystems were even called the “orphans” of computer architecture by some people). Perhaps, many benchmark programs and metrics that were developed to evaluate computer systems focused on the CPU or the memory performance only. Performance of I/O subsystems is as important as that of the CPU or the memory, especially in today‟s world. For example, the transaction processing systems used in airline reservation systems or the automated teller machines in banks have a very heavy I/O traffic, requiring improved I/O performance. To illustrate this point, look at the following example.

Suppose that a certain program takes 200 seconds of elapsed time to execute. Out of these 200 seconds, 180 seconds is the CPU time and the rest is I/O time. If the CPU

performance improves by 40% every year for the next seven years because of developments in technology, but the I/O performance stays the same, let us look at the following table, which shows the situation at the end of each year. Remember that Elapsed time = CPU time + I/O time.
This gives us the I/O time = 200 – 180 = 20 seconds at the beginning, which is 10 % of the elapsed time.

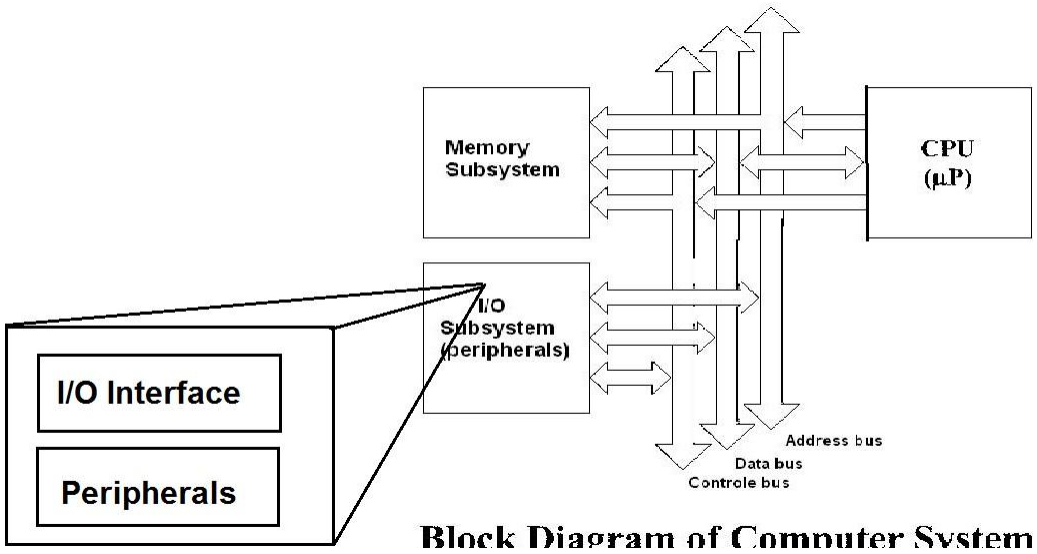
| Year # | CPU Time | I/O Time | Elapsed Time | I/O Time x100 % Elapsed Time |
|---|---|---|---|---|
| 0 | 180 | 20 | 200 | 10 % |
| 1 | 129 | 20 | 149 | 13.42 % |
| 2 | 92 | 20 | 112 | 17.85 % |
| 3 | 66 | 20 | 86 | 23.25 % |
| 4 | 47 | 20 | 67 | 29.85 % |
| 5 | 34 | 20 | 54 | 37.03 % |
| 6 | 24 | 20 | 44 | 45.45 % |
| 7 | 17 | 20 | 37 | 54.05 % |

It can be easily seen that over seven years, the I/O time will become more than 50 % of the total time under these conditions. Therefore, the improvement of I/O performance is as important as the improvement of CPU performance. I/O performance will also be discussed in detail in a later section.

## Major components of an I/O subsystem

I/O subsystems have two major parts:

- The I/O interface, which is the electronic circuitry that connects the CPU to the I/O device.
- Peripherals, which are the devices used to communicate with the CPU, for example, the keyboard, the monitor, etc.




Block Diagram of Computer System

## Computer Interface

A Computer Interface is a piece of hardware whose primary purpose is to connect together any of the following types of computer elements in such a way that the signal levels and the timing requirements of the elements are matched by the interface. Those elements are:

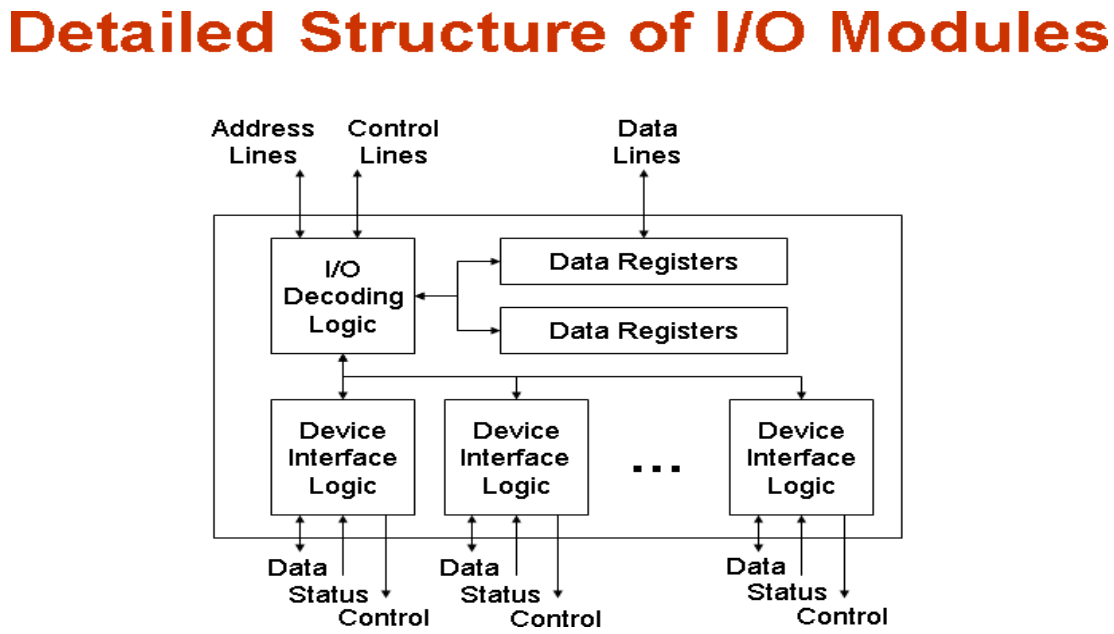
- The processor unit
- The memory subsystem(s)
- Peripheral (or I/O) devices
- The buses (also called "links")

In other words, an interface is an electronic circuit that matches the requirements of the two subsystems between which it is connected. An interface that can be used to connect the microcomputer bus to peripheral devices is called an I/O Port. I/O ports serve the following three purposes:

- Buffering (i.e., holding temporarily) the data to and from the computer bus.
- Holding control information that dictates how a transfer is to be conducted.
- Holding status information so that the processor can monitor the activity of the interface and its associated I/O element.

This control information is usually provided by the CPU and is used to tell the device how to perform the transfer, e.g., if the CPU wants to tell a printer to start a new page, one of the control signals from the CPU can be used for a paper advance command, thereby telling the printer to start printing from the top of the next page.

In the same way the CPU may send a control signal to a tape drive connected in the system asking it to activate the rewind mechanism so that the start of the tape is

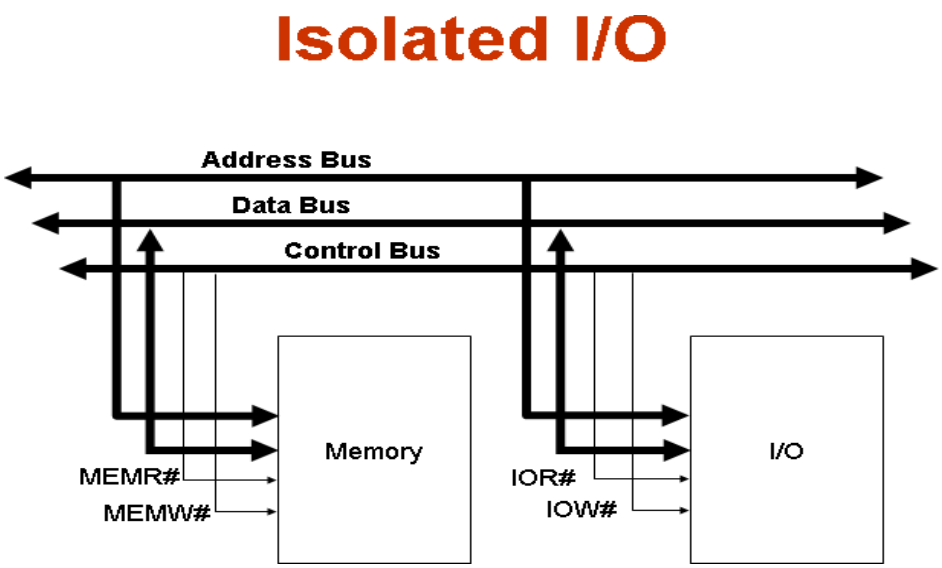



positioned for access by the CPU. Status information from various devices helps the CPU to know what is going on in the system. Once again, using the printer as an example, if the printer runs out of paper, this information should be sent to the CPU immediately. In the same way, if a hard drive in the system crashes, or if a sector is damaged and cannot be read, this information should also be conveyed to the CPU as soon as possible

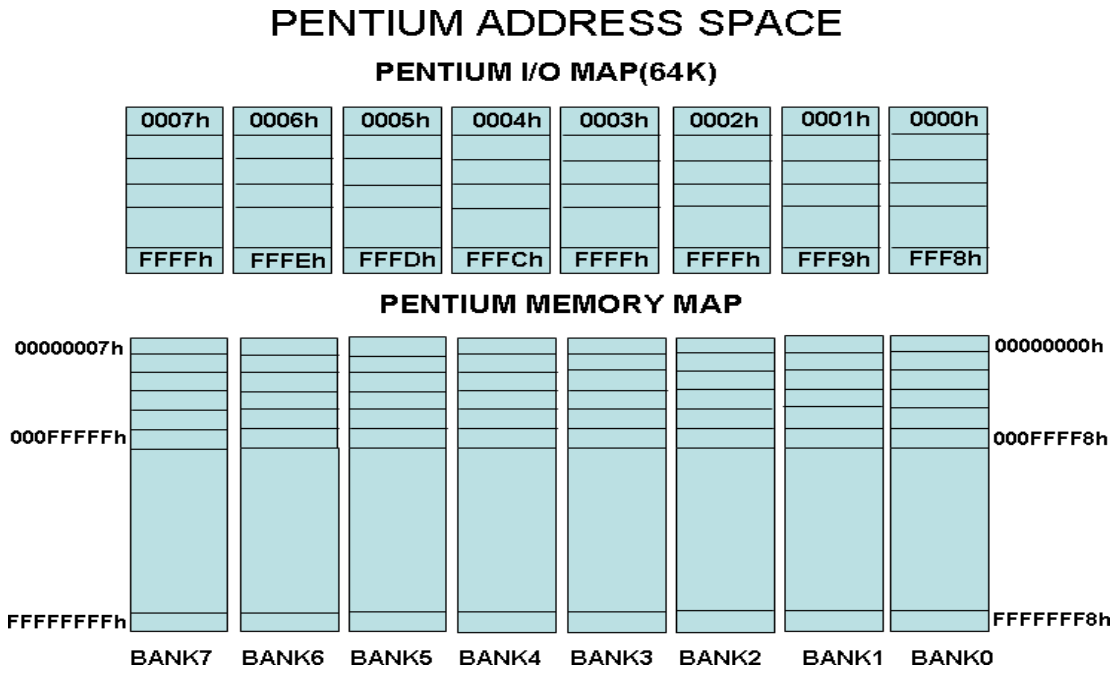
The term "buffer" used in the above discussion also needs to be understood. In most cases, the word buffer refers to I/O registers in an interface where data, status or control information is temporarily stored. A block of memory locations within the main memory or within the peripheral devices is also called a buffer if it is used for temporary storage. Special circuits used in the interfaces for voltage/current matching, at the input and the output, are also called buffers.

The given figure shows a block diagram of a typical I/O subsystem connected with the other components in a computer. The thick horizontal line is the system bus that serves as a back-bone in the entire computer system. It is used to connect the memory subsystems as well as the I/O subsystems together. The CPU also connects to this bus through a "bus interface unit", which is not shown in this figure. Four I/O modules are shown in the figure. One module is used to connect a keyboard and a mouse to the system bus. A second module connects a monitor to the system bus. Another module is used with a hard disk and a fourth I/O module is used by a modem. All these modules are examples of I/O ports. A somewhat detailed view of these modules is shown in the next figure.

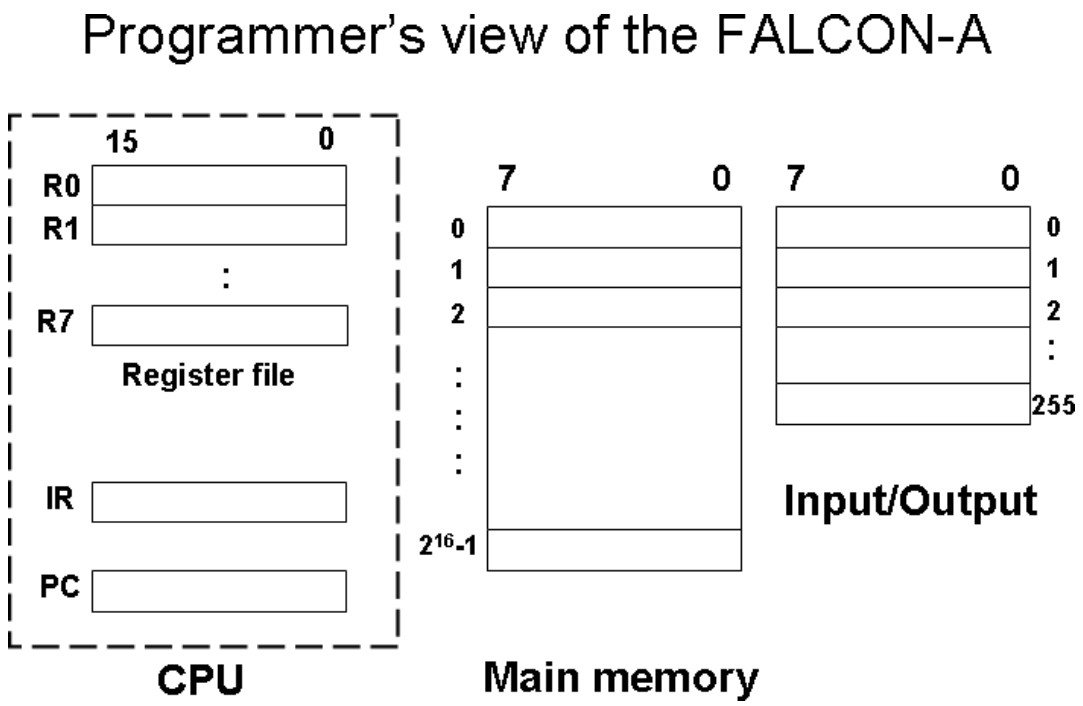
As we already know that the system bus actually consists of three buses, namely the




address bus, the data bus and the control bus. These three buses are being applied to the I/O module in this figure. At the bottom, we see a set of data, status and control lines from each "device interface logic" block. Each of these sets connects to a peripheral device. I/O decoding logic is also shown in this figure



**Memory Mapped I/O versus Isolated I/O**



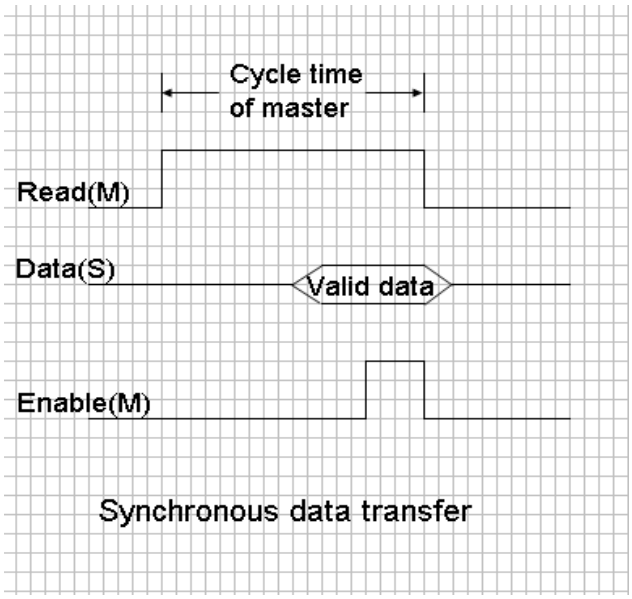

Although this concept was explained earlier as well, it will be useful to review it again in this context. In isolated I/O, a separate address space of the CPU is reserved for I/O operations. This address space is totally different from the address space used for memory devices. In other words, a CPU has two distinct address spaces, one for memory and one for input/output. Unique CPU instructions are associated with the I/O space, which means that if those instructions are executing on the CPU, then the accessed address space will be the I/O space and hence the devices mapped on the I/O space. The x86 family with the **in** and the **out** instructions is a well known example of this situation. Using the **in** instruction, the Pentium processor can receive information from a peripheral device, and using the **out** instruction, the Pentium processor can send information to a peripheral device. Thus, the I/O devices are mapped on the I/O space in case of the Pentium processor. In some processors, like the SRC, there is no separate I/O space. In this case, some address space out of the memory address space must be used to map I/O devices. The benefit will be that all the instructions which access memory can be used for I/O devices. There is no need for including separate I/O instructions in the ISA of the processor. However, the disadvantage will be that the I/O interface will become

complex. If partial decoding is used to reduce the complexity of the I/O interface, then a lot of memory addresses will be consumed. The given figure shows the memory address space as well as the I/O address space for the Pentium processor. The I/O space is of size 64 Kbytes, organized as eight banks of 8 Kbytes each.
A similar diagram for the FALCON-A was shown earlier and is repeated here for easy reference.
The next question to be answered is how the CPU will differentiate between these two address spaces. How will the system components know whether a particular transfer is meant for memory or an I/O device? The answer is simple: by using signals

from the control bus, the CPU will indicate which address space is meant during a particular transfer. Once again, using the Pentium as an example, if the **in** instruction is executing on the processor, the IOR# signal will become active and the MEMR# signal will be deactivated. For a **mov** instruction, the control logic will activate the MEMR# signal instead of the IOR# signal.

## Considerations during I/O Subsystem Design

Certain things must be taken care of during the design of an I/O subsystem.
Data location:
The designer must identify the device where the data to be accessed is available, the address of this device and how to collect the data from this device. For example, if a database needs to be searched for a record that is stored in the fourth sector of the second track of the third platter on a certain hard drive in the system, then this information is related to data location. The particular hard drive must be selected out of the possibly many hard drives in the system, and the address of this record in terms of platter number, track number and sector number must be given to this hard drive.




Data transfer:
he direction of transfer of data; whether it is out of the CPU or into the CPU, whether the data is being sent to the monitor or the hard drive, or whether it is being received from the keyboard or the mouse. It also includes the amount of data to be transferred and the rate at which it should be transferred. If a single mouse click is to be transferred to the CPU, then the amount of data is just one bit; on the other hand, a block of data for the hard drive may be several kilo bytes. Similarly, the rate of the transfer of data to a printer is very different from the

transfer rate needed for a hard drive.

Data synchronization:

This means that the CPU should input data from an input device only when the device is ready to provide data and send data to an output device only when it is ready to receive data.

There are three basic schemes which can be used for synchronization of an I/O data transmission:

- Synchronous transmission
- Semi-synchronous transmission
- Asynchronous transmission

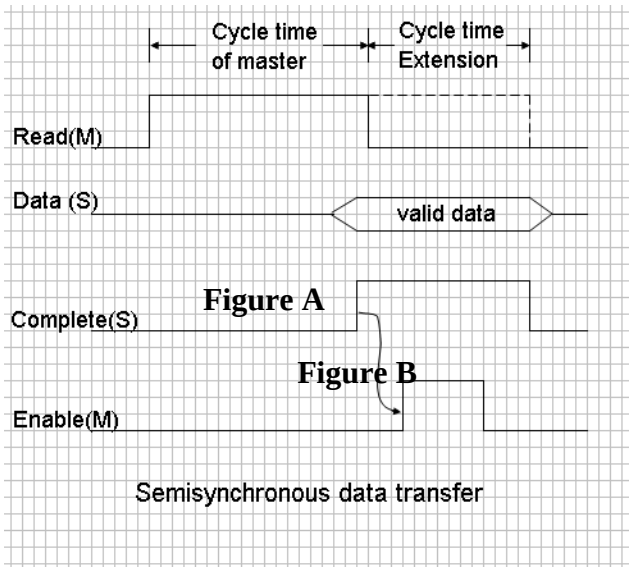
**Synchronous transmission:**

This can be understood by looking at the waveforms shown in Figure A.

s. The master and the slave are assumed to be permanently connected together, so that there is no need for the selection of the particular slave device out of the many devices that may be present in the system. It is also assumed that the slave device can perform the transfer at the speed of the master, so no handshaking signals are needed.

At the start of the transfer operation, the master activates the Read signal, which indicates to the slave that it should respond with data. The data is provided by the slave, and the master uses the Enable signal to latch it. All activity takes place synchronously with the system clock (not shown in the figure). A familiar example of synchronous transfer is a register-to-register transfer within a CPU.

**Figure A**




**Semi-synchronous transmission:**

Figure B explains this type of transfer. All activity is still synchronous with the system clock, but in some situations, the slave device may not be able to provide the data to the master within the allotted time. The additional time needed by the slave, can be provided by adding an integral number of clock periods to the master‟s cycle time.

**Figure A**

**Figure B**

The slave indicates its readiness by activating the complete signal. Upon receiving this signal, the master activates the Enable signal to latch the data provided by the slave. Transfers between the CPU and the main memory are examples of semi-synchronous transfer.

**Asynchronous transmission:**

This type of transfer does not require a common clock. The master and the slave operate at different speeds. Handshaking

signals are necessary in this case, and are used to coordinate the data transfer between the master and the slave as shown in the Figure C.

When the master wants to initiate a data transfer, it activates its Ready signal. The slave detects this signal, and if it can provide data to the master, it does so and also activates its Acknowledge signal. Upon receiving the Acknowledge signal, the master uses the Enable signal to latch the incoming data .The master then deactivates its Ready line, and in response to it, the slave removes its data and deactivates its Acknowledge line.

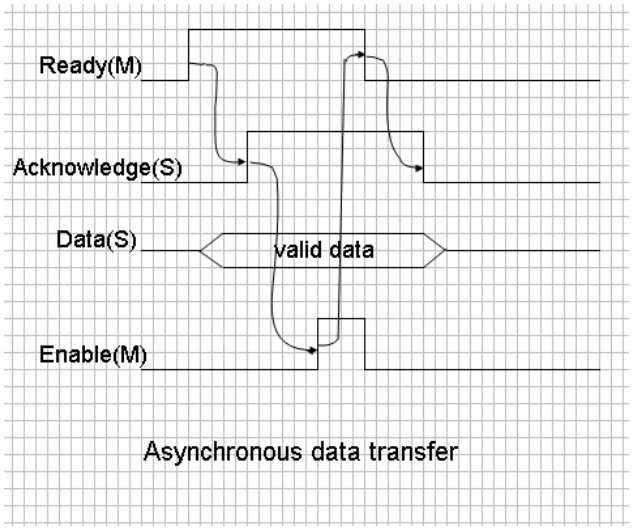



**Figure C**

In all the three cases discussed above, the waveforms correspond to an "input" or a "read" operation. A similar explanation will apply to an "output" or a "write" operation. It should also be noted that the latching of the incoming data can be done by the master either by using the rising edge of the Enable signal or by using its falling-edge. This will depend on the way the intermediate circuitry between the master and the slave is designed.

## Serial and Parallel Transfers

There are two ways in which data can be transferred between the CPU and an I/O device: serial and parallel.

***Serial Transfer,*** or serial communication of data between the CPU and the I/O devices, refers to the situation when all the data bits in a "piece of information", (which is a byte or word mostly), are transferred one bit at a time, over a single pair of wires.

Advantages:

- Easy to implement, especially by using UARTs[4] or USARTs[5].
- Low cost because of less wires.
- Longer distance between transmitter and receiver.

Disadvantages:

- Slow by its very nature.
- serial wave forms

***Parallel Transfer,*** or parallel communication of data between the CPU and the I/O devices, refers to the situation when all the bits of data (8 or 16 usually), are transferred over separate lines simultaneously, or in parallel.

Advantages:

[4] Universal Asynchronous Receiver Transmitter.

[5] Universal Synchronous Asynchronous Receiver Transmitter.

- Fast (compared to serial communication)

Disadvantages:

- High cost (because of more lines).
- Cost increases with distance.
- Possibility of interference (noise) increases with distance.

Remember that the terms "serial" and "parallel" are with respect to the computer I/O ports and not with respect to the CPU. The CPU always transfers data in parallel.

**Types of serial communication**

There are two types of serial communication:

Asynchronous:

- Special bit patterns separate the characters.
- "Dead time" between characters can be of any length.
- Clocks at both ends need not have the same frequency (within permissible limits).

Synchronous:

- Characters are sent back to back.
- Must include special "sync" characters at the beginning of each message.
- Must have special "idle" characters in the data stream to fill up the time when no information is being sent.
- Characters must be precisely spaced.
- Activity at both ends must be coordinated by a single clock. (This implies that the clock must be transmitted with data).

The "maximum information rate" of a synchronous line is higher than that of an asynchronous line with the same "bit rate", because the asynchronous transmission must use extra bits with each character. Different protocols are used for serial and parallel transfer. A protocol is a set of rules understood by both the sender and the receiver. In some cases, these protocols can be predefined for a certain system. As an alternate, some available standard protocols can be used.

**Error conditions related to serial communication**

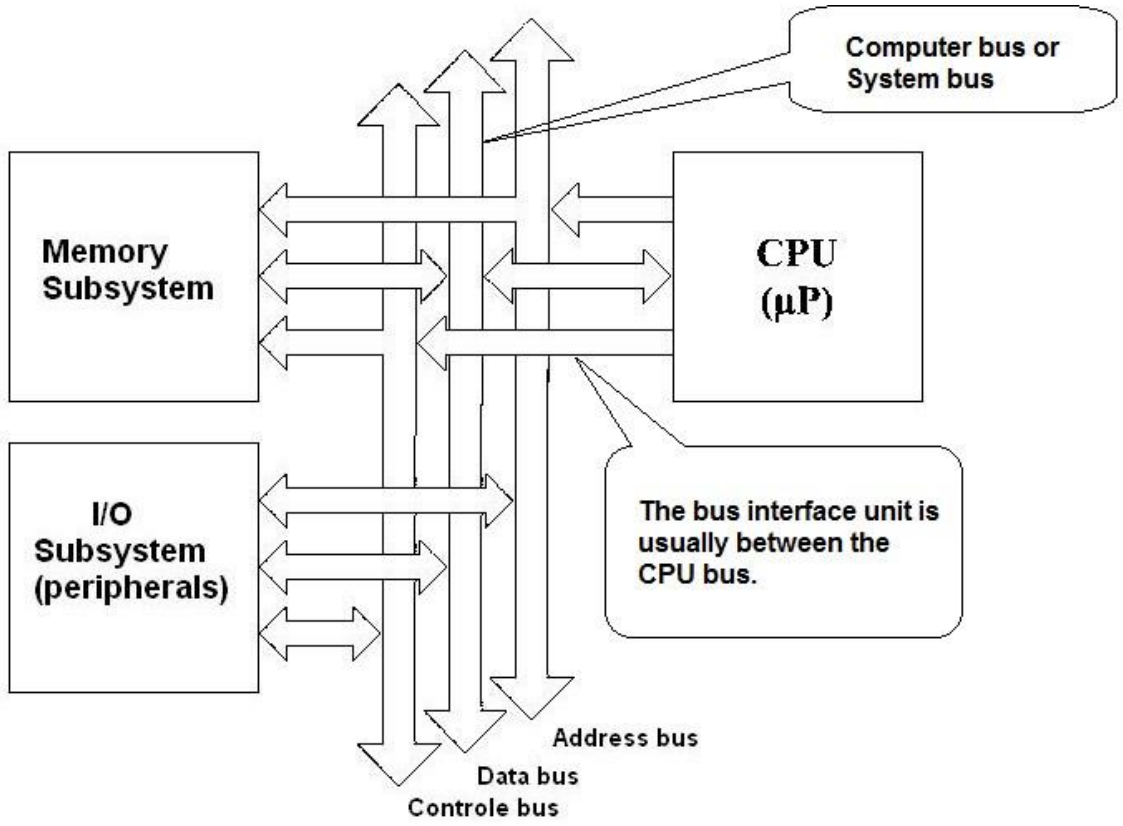
*(Some related to synchronous transmission, some to asynchronous, and some to both).*

- Framing Error: is said to occur when a 0 is received instead of a stop bit (which is always a 1). It means that after the detection of the beginning of a character with a start bit, the appropriate number of stop bits was not detected. [A]
- Parity Error: is said to occur when the parity* of the received data is not the same as it should be. [B] (PARITY is equivalent to the number of 1's; it is either EVEN or ODD. A PARITY BIT is an extra bit added to the data, for the purpose of error detection and correction. If even parity is used, the parity bit is set so that the total number of 1‟s, including the parity bit, is even. The same applies to odd parity.)
- Overrun Error: means that the prior character that was received, was not yet read from the USART's "receive data register" by the CPU, and is overwritten by the

new received character. Thus the first character was lost, and should be retransmitted. [A]

- Under-run Error: If a character is not available at the beginning of an interval, an he transmitter will insert an idle character till the end of the interval. [S]

The block diagram of a general purpose computer system that has been referred to repeatedly in this course has three buses in addition to the three most important blocks. These three buses are collectively referred to as the system bus or the computer bus[6]. The block diagram is repeated here for an easy reference in Figure 1.




**Block Diagram of Computer System**

**Figure 1**

Another organization that is used in modern computers is shown in Figure 2. It has a memory bus for connecting the CPU to the memory subsystem. This bus
is separate from the I/O bus that is used to connect peripherals and I/O devices to the

[6] In some cases, the external CPU bus is the same as the system bus, especially in the case of small, dedicated systems. However, for most systems, there is a "bus interface unit" between the CPU and the system bus. The bus interface unit is not shown in the figure.

system.
Examples of I/O buses include the PCI bus and the ISA bus. These I/O buses provide an "abstract interface" that can be used for interfacing a large variety of peripherals to the system with minimum hardware. It is also possible to standardize I/O buses, as done by several agencies, so that third party manufacturers can build add-on sub systems for existing architectures.
The location of these I/O buses may be different in different computers.Earlier generation computers used a single bus over which the CPU could communicate with the memory as well as the I/O devices. This meant that the bandwidth of the bus was shared between the memory and I/O devices. However, with the passage of time, computer architects drifted towards separate memory and I/O buses, thereby giving more flexibility to users wanting to upgrade their existing systems.

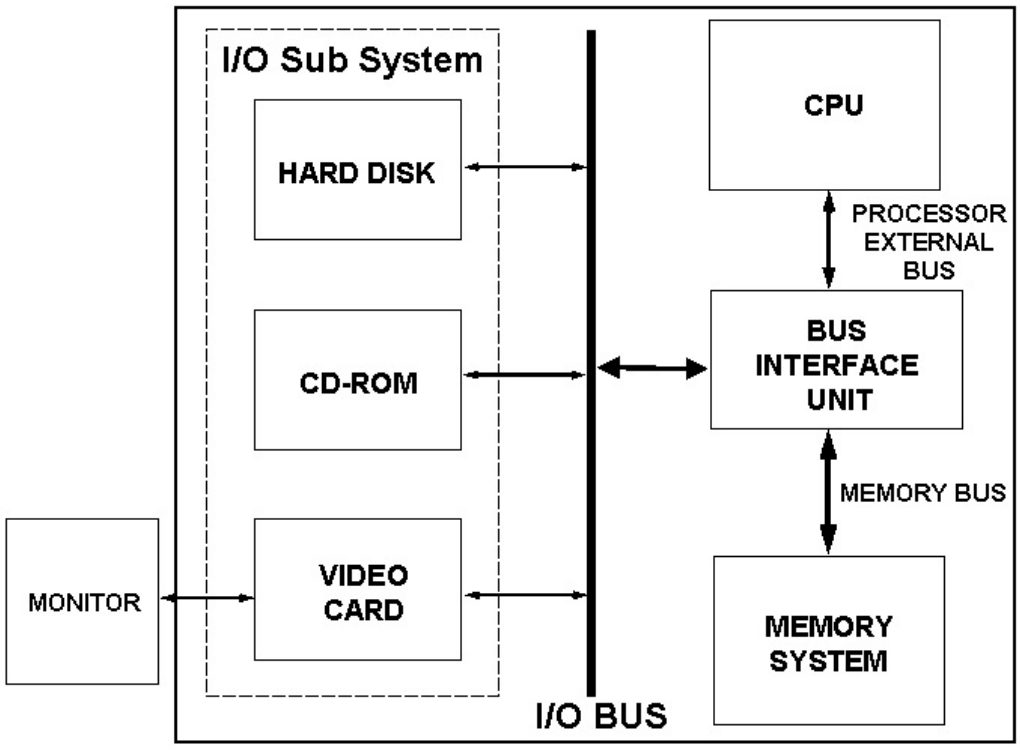



**Figure 2**

A main disadvantage of I/O buses (and the buses in general) is that every bus has a fixed bandwidth which is shared by all devices on the bus. Additionally, electrical constraints like transmission line effects and bus length further reduce the bandwidth. As a result of this, the designer has to make a decision whether to sacrifice interface simplicity (by connecting more devices to the bus) at the cost of bandwidth, or connect fewer devices to the bus and keep things simple to get a better bandwidth. This can be explained with the help of an example.

**Example # 1**

Problem statement:

Consider an I/O bus that can transfer 4 bytes of data in one bus cycle. Suppose that a designer is considering to attach the following two components to this bus:

Hard drive, with a transfer rate of 40 Mbytes/sec

Video card, with a transfer rate of 128 Mbytes/sec.

What will be the implications?

Solution:

The maximum frequency of the bus is 30 MHz[7]. This means that the maximum bandwidth of this bus is 30 x 4 = 120 Mbytes/sec. Now, the demand for bandwidth from these two components will be 128 + 40 =168 Mbytes/sec which is more than the 120 Mbytes/sec that the bus can provide. Thus, if the designer uses these two components with this bus, one or both of these components will be operating at reduced bandwidth.

**Bus arbitration:**

Arbitration is another issue in the use of I/O buses. Most commercially available I/O buses have protocols defining a number of things, for example how many devices can access the bus, what will happen if multiple devices want to access the bus at the same time, etc. In such situations, an "arbitration scheme" must be established. As an example, in the SCSI[8] specifications, every device in the system is assigned an ID which identifies the device to the "bus arbiter". If multiple devices send a request for the bus, the device with the highest priority will be given access to the bus first. Such a scheme is easy to implement because the arbiter can easily decide which device should be given access to the bus, but its disadvantage is that the device with a low priority will not be able to get access to the bus[9]. An alternate scheme would be to give the highest priority to the device that has been waiting for the longest time for the bus. As a result of this arbitration, the access time, or the latency, of such buses will be further reduced.

Details about the PCI and some other buses will be presented in a separate section.

**Example # 2**

Problem statement:

If a bus requires 10 nsec for bus requests, 10 nsec for arbitration and the average time to complete an operation is 15 nsec after the access to the bus has been granted, is it possible for such a bus to perform 50 million IOPS?

Solution:

For 50 million IOPS, the average time for each IOP is $1 / (50 \times 10^6) = 20$ nsec. Given the information about the bus, the sum of the three times is 10 + 10 + 15 = 35 nsec for a complete I/O operation. This means that the bus can perform a maximum of $1 / (35 \times 10^{-9}) = 28.6$ million IOPS.

Thus, it will not be able to perform 50 million IOPS.

[7] These numbers correspond to an I/O bus that is relatively old. Modern systems use much faster buses than this.

[8] Small Computer System Interface.

[9] Such a situation is called "starvation".

# Lecture No. 24

## Designing Parallel Input and Output Ports

**Reading Material**

Handouts Slides

***Summary***

- Designing Parallel I/O Ports
- Practical Implementation of the SAD
- NUXI Problem
- Variation in the Implementation of the Address Decoder
- Estimating the Delay Interval

## Designing Parallel I/O Ports

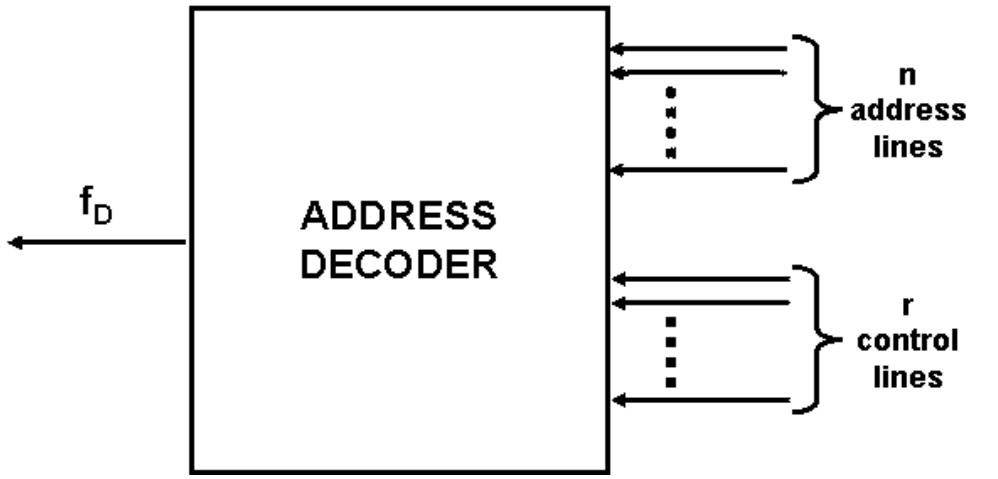
This section is about designing parallel input and output ports. As you already know from the previous discussion, an interface that is used to connect the computer bus with I/O devices is called an I/O port. This I/O port can be connected directly to the computer bus (also called the system bus) or through an intermediate bus called the I/O bus. This intermediate bus is also called the expansion bus or the peripheral bus. In any case, the following general information about I/O bus cycles on a typical CPU should be kept in mind: At the start of a particular bus cycle (which will be an I/O bus cycle in this case), the CPU places an address on its address bus. This address will identify the I/O device to be involved in the transfer. After some time the CPU will activate certain control signals, which will indicate whether the particular I/O bus cycle, is an I/O read or an I/O write cycle. Based on these control signals, in case of I/O read cycle, the CPU will be expecting data from the selected input device over the data bus, and for an I/O write cycle the CPU will provide data to the selected device over the data bus. At the end of this I/O bus cycle, the address (and data) information will be removed from the buses and the control signals will be reset. It can be easily understood from this discussion that we must match the timing requirements of the I/O ports to be designed with the timing parameters of the given CPU. Additionally, the voltage and current requirements of the I/O ports must be matched with the voltage and current specifications of the CPU. For simplicity, we ignore the voltage and current matching details in this discussion and only

focus on the logic levels and timing aspects of the design. Voltage and current related discussions are the topic of an electronics course.
Thus, there are two important functions which should be built into I/O ports.

1. Address decoding
2. Data isolation for input ports or data capturing for output ports.

**1. Address decoding:** Since every I/O port has a unique identifier associated with it, (which is called its ***address***, and no other port in the system should have the same address), by monitoring the system address bus, the I/O port knows when it is its turn to participate in a transfer. At this time, the address decoder within the I/O port generates an asserted output which can be applied to the enable input of tri-state buffers in input ports or the latch enable input of latches in output ports.

Block diagram of an address decoder




Our definition of an address decoder:
An "Address Decoder" is a combinational (logic) circuit with ***n*** + ***r*** inputs and a single output, where

***n*** = the number of address lines into the decoder, and
***r*** = the number of control lines into the decoder.

The output $f_D$ is active only when the corresponding address is present on the ***n*** address lines and the corresponding ***r*** control lines hold the "proper" (active or inactive) value. $f_D$ is inactive for all other situations.
Suggestions for address decoder design:

1.1 Start by thinking of the address decoder as a "big AND gate". We will call this a "skeleton address decoder" or SAD. The output of the SAD will be active only when the correct address is present on the system address bus and the relevant control bus signals hold the proper values. At all other times, the output of the SAD should be deactivated.
1.2 Always write the port address of the port to be designed in binary. Associate the CPU"s address lines with each bit. Those lines which are zero will be inverted before being fed into the "big AND gate"; other address lines will not be inverted.
1.3 List the relevant control signals for the system to which the port is to be attached. If the "proper" value of the signal is 0, it should be inverted before applying to the SAD, otherwise it is fed directly into the SAD.
1.4 Determine whether the decoder output should be active high or low. This will depend on the type of latch or buffer used in the design. If an active low decoder output is needed, invert the output from the "big AND gate".
1.5 Once the logic for the address decoder is established, the SAD can be implemented using any of the available methods of logic design. For example, HDL code in Verilog or VHDL can be generated and the address decoder can be implemented using PLDs. Alternately, the SAD can be implemented using SSI building blocks.
**2. Data isolation or capturing:** For input ports, the in coming data should be placed on the data bus only during the I/O read bus cycle. At all other times, this data should be isolated from the data bus otherwise it will cause "bus contention". Tri-state buffers are used for this purpose. Their input lines are connected to the peripheral device supplying data and their output lines are connected to the data bus. The common enable line of such buffers is driven with the output of the SAD. If this enable is active low, the output of the big AND gate in the SAD should be inverted, as described earlier.

For output ports, data is made available for the peripheral device at the data bus during the I/O write bus cycle. During other bus cycles, this data will be removed from the data bus by the processor. Latches (or registers) are used for this purpose. Their input lines are connected to the system data bus and their output lines are connected to the peripheral device receiving data. The common clock (or latch enable) line of such latches is driven with the output of the SAD. If this clock is active low, the output of the big AND gate in the SAD should be inverted.

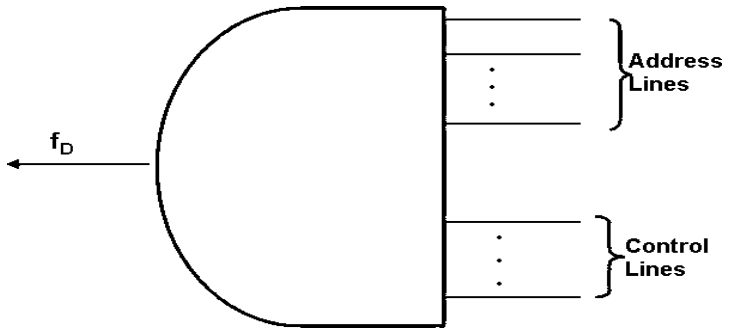
**Example # 1**

Problem Statement:

Design a 16-bit parallel output port mapped on address DEh of the I/O space of the FALCON-A CPU.

Solution:

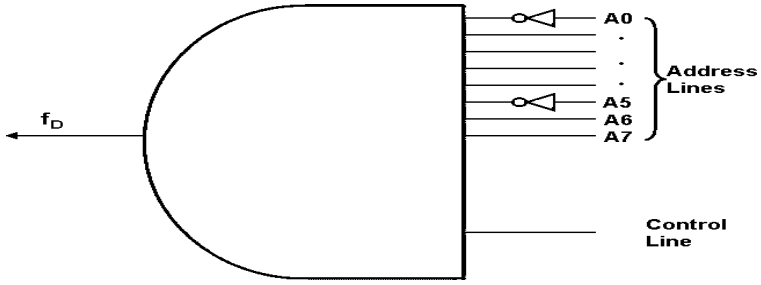
Using the guidelines mentioned above, we start with a "big AND gate" (SAD) and write the address to be decoded (DEh) in binary.



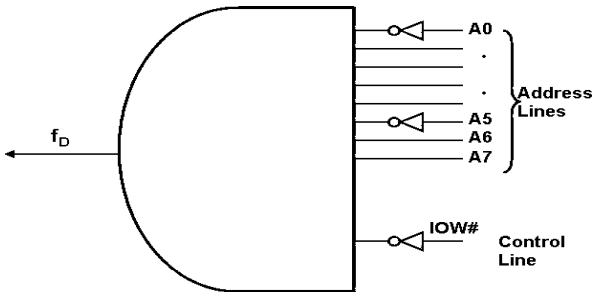

Thus, DEh → 1101 1110 b. Associating one CPU address line with each bit, we get A0 = 0, A1=1, etc as shown in the table below.
Because the I/O space on the FALCON-A is only 256 bytes, address lines A15 .. A8 are don‟t cares, and will not be used in this design.

| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| A7 | A6 | | A4 | A3 | A2 | A1 | |

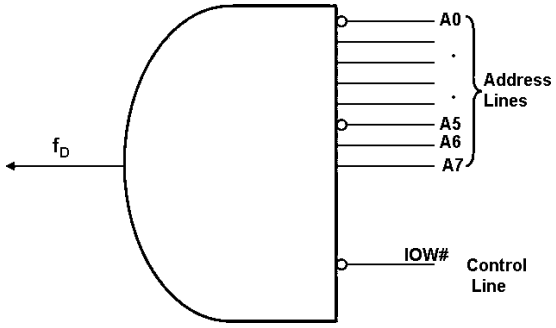
Thus, A0 and A5 will be applied to the "big AND gate" after inversion. The remaining address lines will be connected directly to the inputs of the SAD.




Next, we look at the relevant control signals. The only signal which should be used in this case is IOW#. A logic 0 (zero) on this line indicates that it is active. Thus, it should be inverted before being applied to the input of the SAD.
We can easily see that our SAD intuitively conforms to the way we defined an address decoder. Its output is a 1 only when the address (xxxx xxxx 1101 1110 b) is present on the FALCON-A‟s address bus during an I/O write cycle (By the way, this will take place when the instruction **out reg, addr** with **addr=DEh or 222d** is executing on the FALCON-A). At all other times, its output will be inactive.




To make things simple, we use a circle (or a bubble) to indicate an inverter, as shown .Since this is a 16-bit output port, we will use two 8-bit registers to capture data from the FALCON-A‟s data bus. The output of the SAD will be connected to the enable inputs of the two registers. The D-inputs of the registers will be connected to the data bus and the Q outputs of the registers will be connected to the peripheral device.

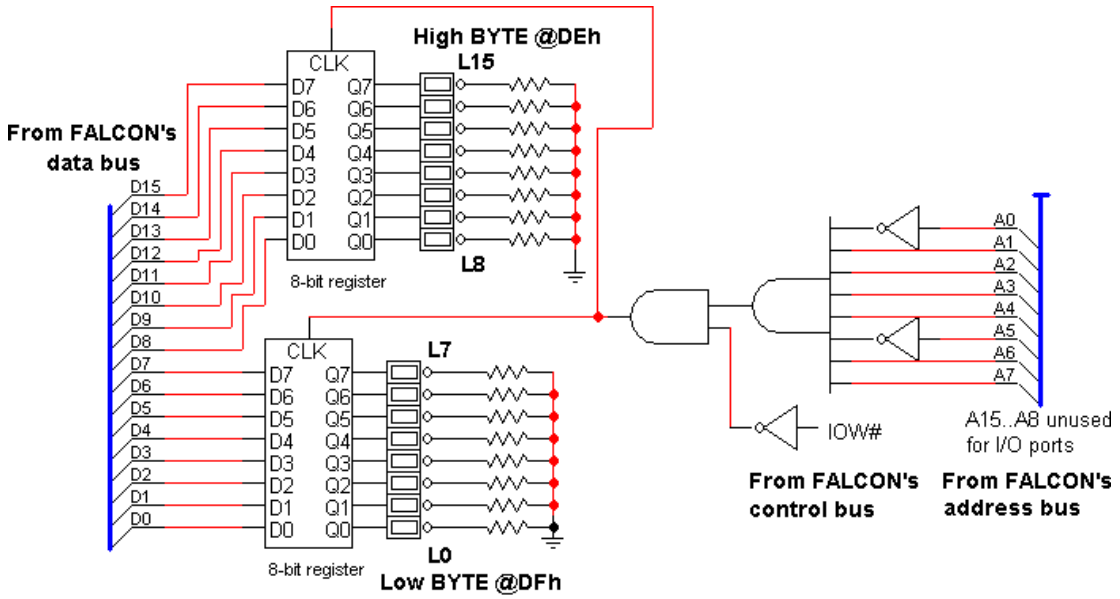



## Practical implementation of the SAD

Our SAD in this design is an AND gate with 9 inputs. Using SSI chips, we can implement this SAD using an 8-input AND gate and a 2-input AND gate as shown in the figure shown below.

**Displaying output data using LED branches:**

An “LED branch” is a combination of a resistor and a light emitting diode (LED) in series. Sixteen LED branches can be used to display the output data captured by the registers as shown in the figure below.




**A 16-bit parallel output port for the FALCON-A at address DEh and DFh**

**Example # 2**

Problem statement:

Given a 16-bit parallel output port attached with the FALCON-A CPU as shown in the figure. The port is mapped onto address DEh of the FALCON-A‟s I/O space. Sixteen LED branches are used to display the data being received from the FALCON-A‟s data bus. Every LED branch is wired in such a way that when a 1 appears on the particular data bus bit, it turns the LED on; a 0 turns it off.

Which LEDs will be ON when the instruction

**out r2, 222** [10]

[10] *Depending on the way the assembler is written, the syntax of the* ***out*** *instruction may allow only the decimal form of the port address, or only the hexadecimal form, or both. Our version of the assembler for the FALCON-A allows the decimal form only. It also requires that the port address be aligned on 16-bit "word boundaries", which means that every port address should be divisible by 2.*

executes on the CPU? Assume r2 contains 1234h.
Solution:
Since r2 contains 1234h, the bit pattern corresponding to this value will be sent out to the output port at address 222 (or DEh). This is the address of the output port in this example. Writing the bit pattern in binary will help us determine the LEDs which will be ON.

Now 1234h gives us the following bit associations with the data bus

| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | D | D12 | | 1 | D | D9 | | 8 | | D | D5 | D4 | | D2 | | 0 |
| MSB at address DEh | | | | | | | | LSB at address DFh | | | | | | | |

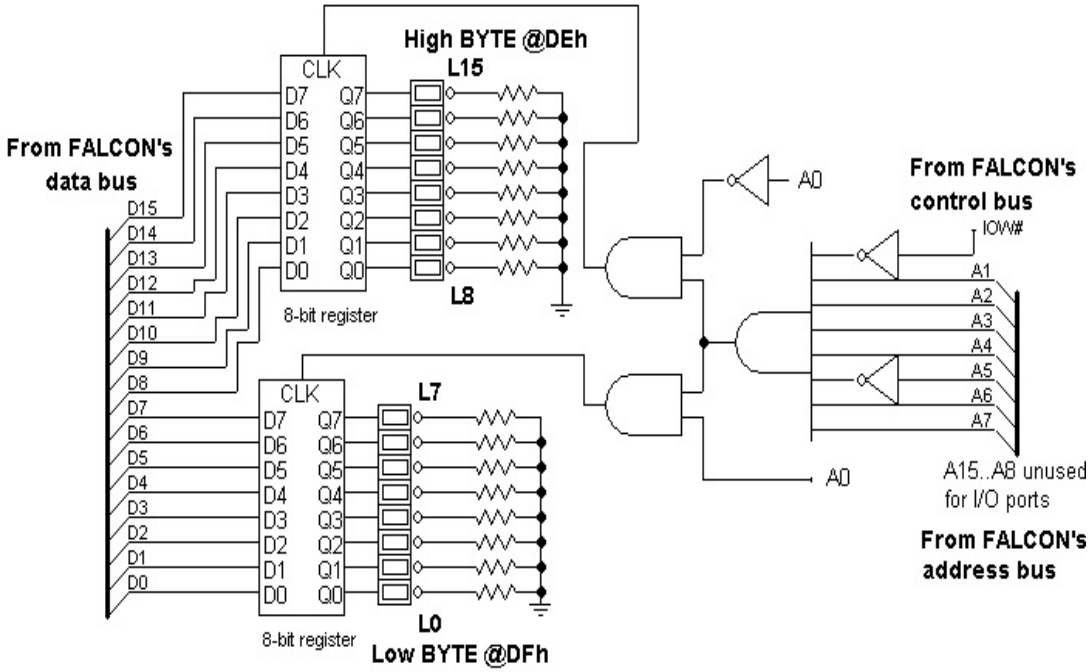
Note that the 8-bit register which uses lines D15 .. D8 of the FALCON-A‟s data bus is actually mapped onto address DEh of the I/O space. This is because the architect of the FALCON-A had chosen a “byte-wide” (i.e., x8) organization of the address space, a 16-bit data bus width, and the “big-endian” data format at the ISA design stage. Additionally, data bus lines D15...D8 will transfer the data byte of higher significance (MSB) using address DEh, and D7...D0 will transfer the data byte of lower significance (LSB) using address DFh. Thus the LEDs at L12, L9, L5, L4 and L2 will turn on.

## The NUXI Problem

It can be easily understood from the previous example that the big-endian format results in the least significant byte being transferred over the most significant side of the data bus, and vice versa. The situation will be exactly opposite when the little-endian format is used. In this case, the least significant byte will be transferred over the least side of the data bus. Now imagine a computer using the little-endian format exchanging data with a computer using the big-endian format over a 16-bit parallel port. (this may be the case when we have a network of different types of computer, for example). The data transmitted by one will be received in a “swapped” form by the other, eg., the string “UN” will be received as “NU” and the string “IX” will be received as “XI”. So UNIX changes to NUXI --- hence the name NUXI problem. Special software is used to resolve this problem.

## Variation in the Implementation of the Address Decoder

The implementation of the address decoder shown in Example #1(lec24) assumes that the FALCON-A does not allow the use of some part of its data bus during an I/O (or memory) transfer. Another restriction that was imposed by the assembler was that all port addresses should be divisible by 2. This implies that address line A0 will always be zero.
If the FALCON-A architect had allowed the use some of part of its data bus (eg, 8-bits) during a transfer, the situation would be different.
The logic diagram shown in the next figure is a 16-bit parallel output port at the same address (DEh) for the FALCON-A assuming that part of its data bus (D15..D8) or (D7..D0) can be used independently during an I/O transfer. Note that the enable inputs of the two 8-bit registers are not connected together in this case. Moreover, since the 16-bit port uses two addresses, address line A0 will be at a logic 0 for address DEh, and at a logic 1 for address DFh. This means that it cannot be used at the input of the big AND gate. So, A0 has been used in a different position with the two 2-input AND gates. The 2-input AND gate where A0 is applied after inversion will generate a 1 at its output when A0 = 0. Thus, this output will enable the 8-bit register mapped on the even address DEh. In case of the other AND gate, A0 is not inverted. So the corresponding 8-bit register will be mapped on the odd address DFh. The input that became available after removing A0 from its old position can be used for the IOW# control signal. The rest of the circuit is the same as it was in the previous figure.




A 16-bit parallel output port for the FALCON-A at address DEh and DFh

We can understand from the above discussion that the decisions made at the time of ISA design have a strong bearing on the implementation details and the working of the

computer. Suppose we assume that the assembler developer had decided not to restrict the port addresses to even values, then what will be the implications?

As an example, consider the execution of the instruction **out r2, 223** assuming r2 contains 1234h. This is a 16-bit transfer at address 223 (DFh) and 224 (E0h).

For the output port (shown in the first figure) where the CPU does not allow the use of some part of its data bus in a transfer, none of the registers will be enabled as a result of this instruction because the output of the 8-input AND gate will be a zero for both addresses DFh and E0h. Thus, that output port cannot be used.

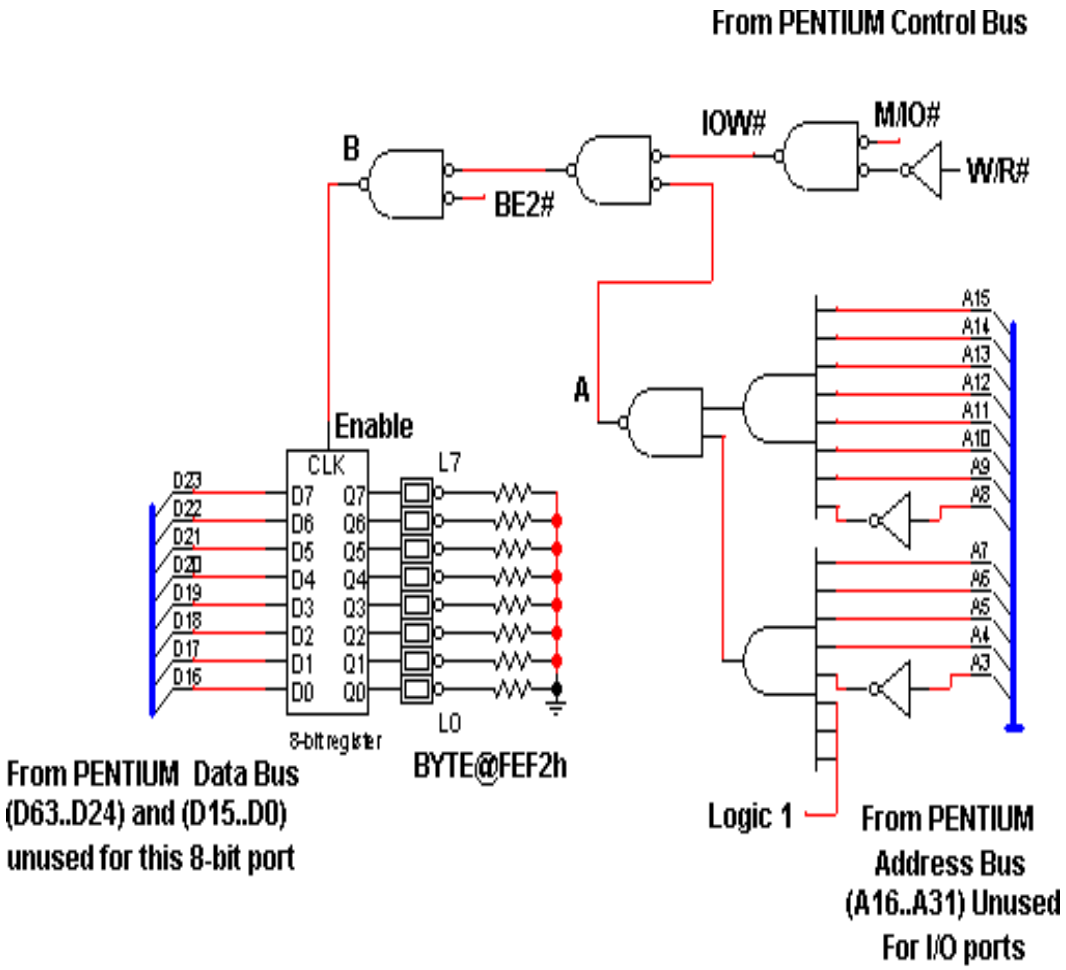
In the second figure, where the CPU has allowed to use a portion of its data bus in an I/O transfer, the register at the address DEh will not be enabled. The CPU will send the high data byte(12h) to the register at the address DFh (because it will be enabled at that time due to the address DFh) over data lines D7…D0. The fact that data lines D7…D0 should be used for the transfer of high byte, will be taken care of by the hardware, internal to the CPU.

Now the question is where the low data byte (i.e. 34h) present at D15…D8 data lines would be placed? If there exists an output port at address E0h in the system, then 34h will be placed there **(in the next bus cycle),** otherwise it will be lost. Again, it is the CPU‟s responsibility to check whether the next address in the system exists or not and if exists then enable that port so that the low byte of data can be placed there.

A possible option for the architect in this case would be to revisit the design steps and allow the use of part of the CPU registers (or at least for some of them) for I/O transfers.

The logic diagram shown below shows an 8-bit parallel output port at address FEF2h of the Pentium‟s I/O address space. Since the Pentium allows the use of some part of its data bus during a transfer, we can use the BE2# signal in the address decoder to enable the 8-bit register. The following instructions will access this output port.

```
mov dx, 0FEF2h
mov al, 12h
out dx, al
```




**An 8-bit Parallel Output Port for the PENTIUM Processor at address FEF2h of the I/O space**

The Pentium **does** allow the use of some part of its 32-bit accumulator register EAX. In case only 8-bits are to be transferred, register AL can be used, as shown in the program fragment above. The data byte 12h will be sent to the 8-bit register over lines D23..D16. Since 12h corresponds to 0001 0010 in binary, this will cause the LEDs L4 and L1 to turn on.

**Example # 3**

Problem statement:

Write an assembly language program to turn on the 16 LEDs one by one on the output port of Example #1(lec24). Each LED should stay on for a noticeable duration of time. Repeat from the first LED after the last LED is turned on.

Solution:

The solution is shown in the text box with a filename: Example_3.asmfa. The working of this program is explained below:

The first two instructions turn all the LEDs off by sending a 0 to each bit of the output port at address 222.

```
mov   r1,0
out r1,222
```

```
; filename: Example_3.asmfa;
ALL LEDS ARE turned Off initially
        movi r1,0
        out r1,222;
First LED will be turned on each time
start:
    movi r1,1
        out r1,222;
        movi r5,15;
;DELAY LOOP;
delay1: movi r2,0
again1: subi r2,r2,1
      jnz r2, [again1];
        movi r3,0          ;
TURN OFF ALL LEDS
        out r3,222;
delay2: movi r2,0
again2: subi r2,r2,1
         jnz r2, [again2];
        shiftl r1,r1,1;  next LED ON
        out r1,222
        subi r5,r5,1
        jnz r5, [delay1]
        jump [start]
        halt
```

Then a 1 is sent to L0 causing it to turn on, and the program enters a loop which executes 15 times to cause the other LEDs (L1 through L15) to turn on, one by one in sequence. Register r5 is being used as loop counter. The following three instructions introduce a delay between successive bit patterns sent to the output port, so that each LED stays on for a noticeable duration of time.

```
delay1:movi    r2,0
again1:subi r2,r2,1
        jnz r2,[again1]
```

Starting with a value of 0 in r2 [11], this value is decremented to FFFFh when the again1 loop is entered. The **jnz** instruction will cause r2 to decrement again and again; thereby executing the loop 65,535 times. An estimate of the delay interval is presented at the end of this section.

[11] *this is necessary because the immediate operand with the **movi** instruction of the FALCON-A has a range of 0h to FFh. This will not give us the large loop counter that we need here. So we use the above software trick. An alternate way would be to use nested loops, but that will tie up additional CPU registers.*

After this delay, all the LEDs are turned off, and a second delay loop executes. Finally, the next LED on the left, in sequence, is turned on by the following two instructions:

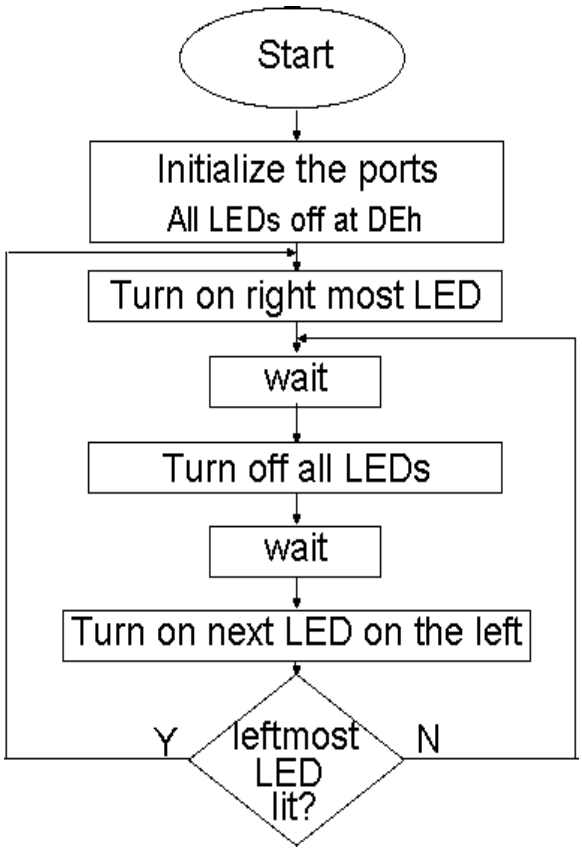
```
shiftl r1,r1,1
out r1, 222
```

After the left most LED is turned on, the process starts all over again because of the last **jump** instruction. The outermost loop executes indefinitely.

## Estimating the Delay Interval

To make things simple, assume that the FALCON-A is operating at a clock frequency of 1 MHz. Also, assume that the **subi** and the **jnz** instructions take 3 and 4 clock periods, respectively, to execute. Since these two instructions execute 65,535 times each, we can use the following formula to compute the execution time of this loop:

ET = CPI x IC x T = CPI x IC / f



where

CPI = clocks per instruction

T = time period of the clock, and

f = frequency of the clock.

Using the assumed values, we get

$ET = (3+4) \times 65535 / (1 \times 10^6) = 0.459 \text{ sec}$

Since the **movi r2, 0** instruction executes only once, the time it takes to execute is negligible and has been ignored in this calculation.

# Lecture No. 25

## Input Output Interface

**Reading Material**

Handouts Slides

***Summary***

- Designing a Parallel Input Port
- Memory Mapped I/O Ports
- Partial Decoding and the "wrap around" Effect
- Data Bus Multiplexing
- A generic I/O Interface
- The Centronics Parallel Printer Interface

## Designing a parallel input port

The following example illustrates a number of important concepts.

**Example # 1**

Problem statement:

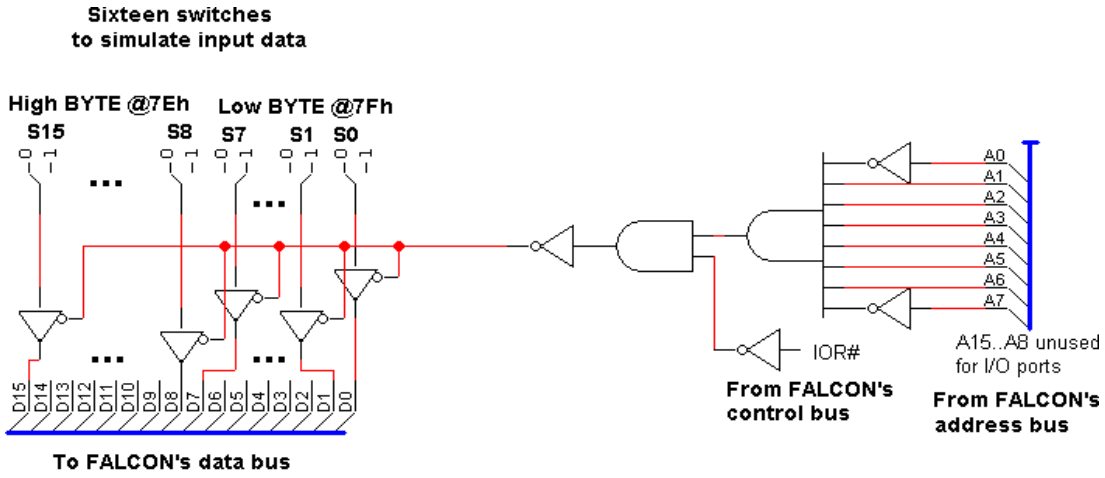
Design an 16-bit parallel input port mapped on address 7Eh of the I/O space of the FALCON-A CPU.

Solution:

The process of designing a parallel input port is very similar to the design of a parallel output port except for the following differences:

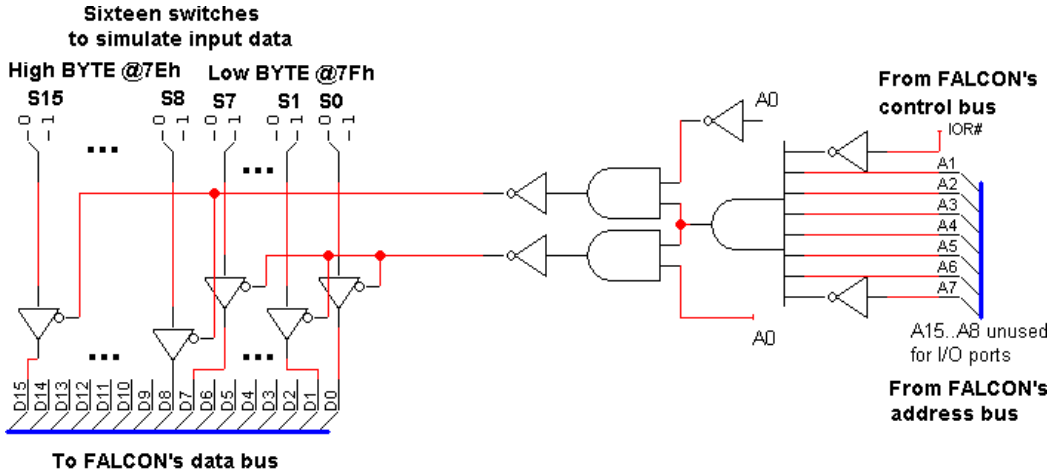
1. The address in this case is 7Eh, which is different from the previous value. Hence, the address decoder will have the inputs A7 and A0 inverted, while the other address lines at its input will not be inverted.
2. Control bus signal IOR# will be used instead of the signal IOW#.
3. A set of sixteen tri-state buffers will be used on. Their common enable line will be connected to the output of the big AND gate (in the figure, $f_D$ is being inverted because Enable is active low). The input of these buffers can be connected to the input device and the output is connected to the FALCON-A's data bus.

In this example, switches S15...S0 are used to simulate the input data. The complete logic circuit is shown in the next two figures.




A 16-bit parallel input port for the FALCON-A at address 7Eh and 7Fh

In the second figure, the CPU is assumed to allow the use of some part of its data bus during a transfer, while in the first figure it is not allowed.




A 16-bit parallel input port for the FALCON-A at address 7Eh and 7Fh

**Example # 2**

Problem statement:

Given a FALCON-A processor with a 16-bit parallel input port at address 7Eh and a 16-bit parallel output port at address DEh. Sixteen LED branches are used to display the data at the output port and sixteen switches are used to send data through the input port. Write an assembly language program to continuously monitor the input port and blink the

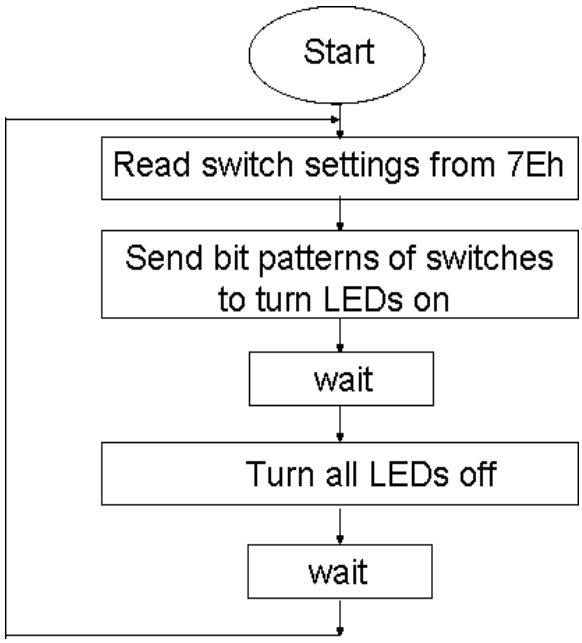
LED or LED(s) corresponding to the switch (es) set to logic 1. For example, if S0 and S2 are set to 1, then only the LEDs L0 and L2 should blink. If S7 is also set to logic 1 later, then L7 should also start blinking.

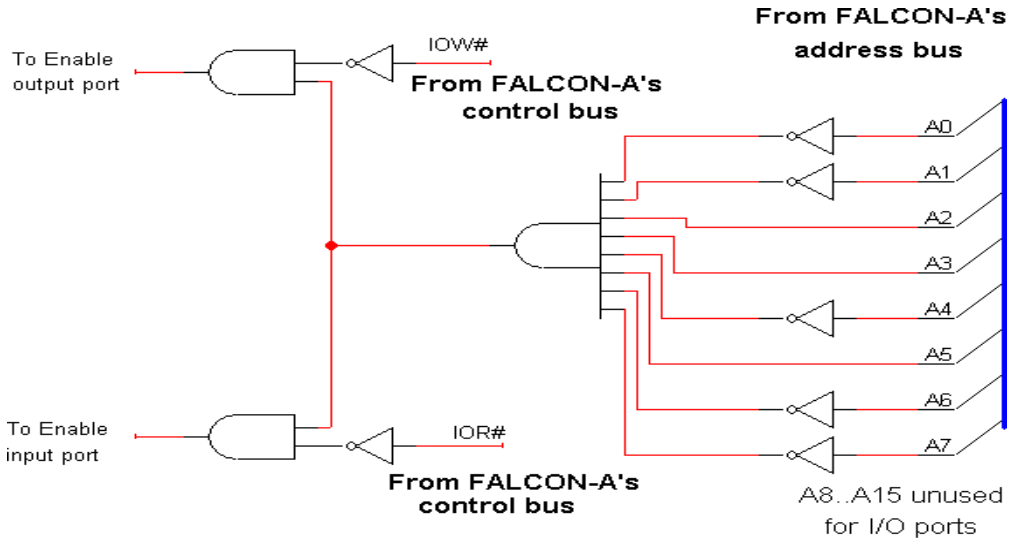
Solution:

The program is shown in the text box with filename: Example_2. It works as explained below:

The first two instructions read the input port at address 7Eh and send this bit pattern to the output port at address DEh. This will cause the LEDs corresponding to the switches that are set to a 1 to turn on. Next, the program waits for a suitable amount of time, and then turns all LEDs off and waits again.

```
;filename: Example_2.asmfa
;Notes:
;       r1 is used as an I/O register
;       r2 is used as a delay counter
;
start:  in r1, 126      ; 126d = 7Eh
        out r1, 222     ; 222d = DEh
;
        movi r2, 0
delay1: subi r2, r2, 1
        jnz r2, [delay1]
;
        movi r1, 0      ; all LEDs off
        out r1, 222
;
        movi r2, 0
delay2: subi r2, r2, 1
        jnz r2, [delay2]
;
        jump [start]
;
        halt
```

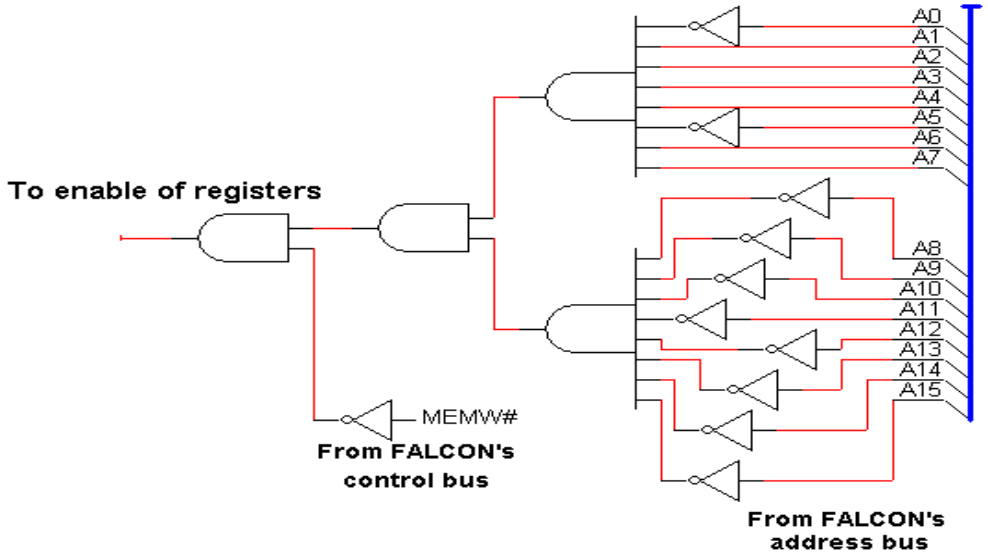



**FALCON-A's Address Decoder for an I/O Port at the Address 2Ch**

After the second wait, the program reads the input port again. The LEDs that will be turn on at the output port will now be according to the new switch settings at the input port. The process repeats indefinitely. Please see the flowchart also.

It is also possible to use a single address for both the input and the output port. The following diagram shows an address decoder for a 16-bit parallel input/output port at address 2Ch of the FALCON-A‟s I/O space. Note that the control bus lines IOW# and IOR# will differentiate between the register and the tri-state buffer




**Address Decoder for a memory mapped 16-bit parallel output port for the FALCON-A at address 00DEh and 00DFh**

.

## Memory mapped I/O ports

If it is desired to map the 16-bit output port of Example #1(lec24) on the memory space of the FALCON-A, the following changes would be needed.

1. Replace the IOW# signal with the MEMW# signal.
2. Use the entire CPU address bus at the input of the address decoder, as shown in the next figure. This address decoder uses the addresses 00DEh and 00DFh of the FALCON-A‟s memory space.
3. Use the instruction for sending data to the output port (for memory mapped input ports, use the **load** instruction instead of the **in** instruction).

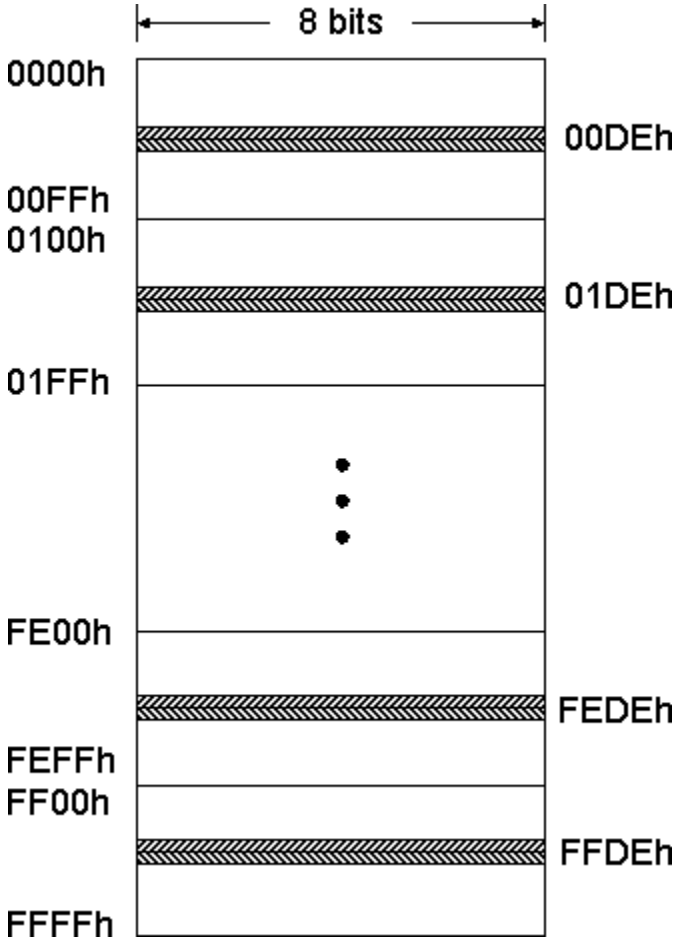
The program for Example #2(lec25) is rewritten for the case of a memory mapped output port, and is shown in the attached text box. The advantage will be that more than 256 ports are available, but the disadvantage is that the address decoder will become more complex, resulting in increased hardware costs.

To avoid the increase in hardware complexity, many architects use what is called “partial decoding”. This is explained in the next section.

```
;filename: Example_2MM.asmfa
;Notes:
;       For MEMORY MAPPED
;       output port at 00DEh
;
;       r6 holds the output address
;       r7 holds the input address
;
        movi r6, 111
        add r6, r6, r6
;
        movi r7, 126
;
;       r1 is used as an I/O register
;       r2 is used as a delay counter
;
start:  load r1,[r7]    ; 126d = 7Eh
        store r1, [r6]  ; 222d = DEh
;       movi r2, 0
delay1: subi r2, r2, 1
        jnz r2, [delay1]
;
        movi r1, 0      ; all LEDs off
        store r1, [r6]
;       movi r2, 0
delay2: subi r2, r2, 1
        jnz r2, [delay2]
;       jump [start]
;       halt
```

## Partial decoding and the “wrap around” effect

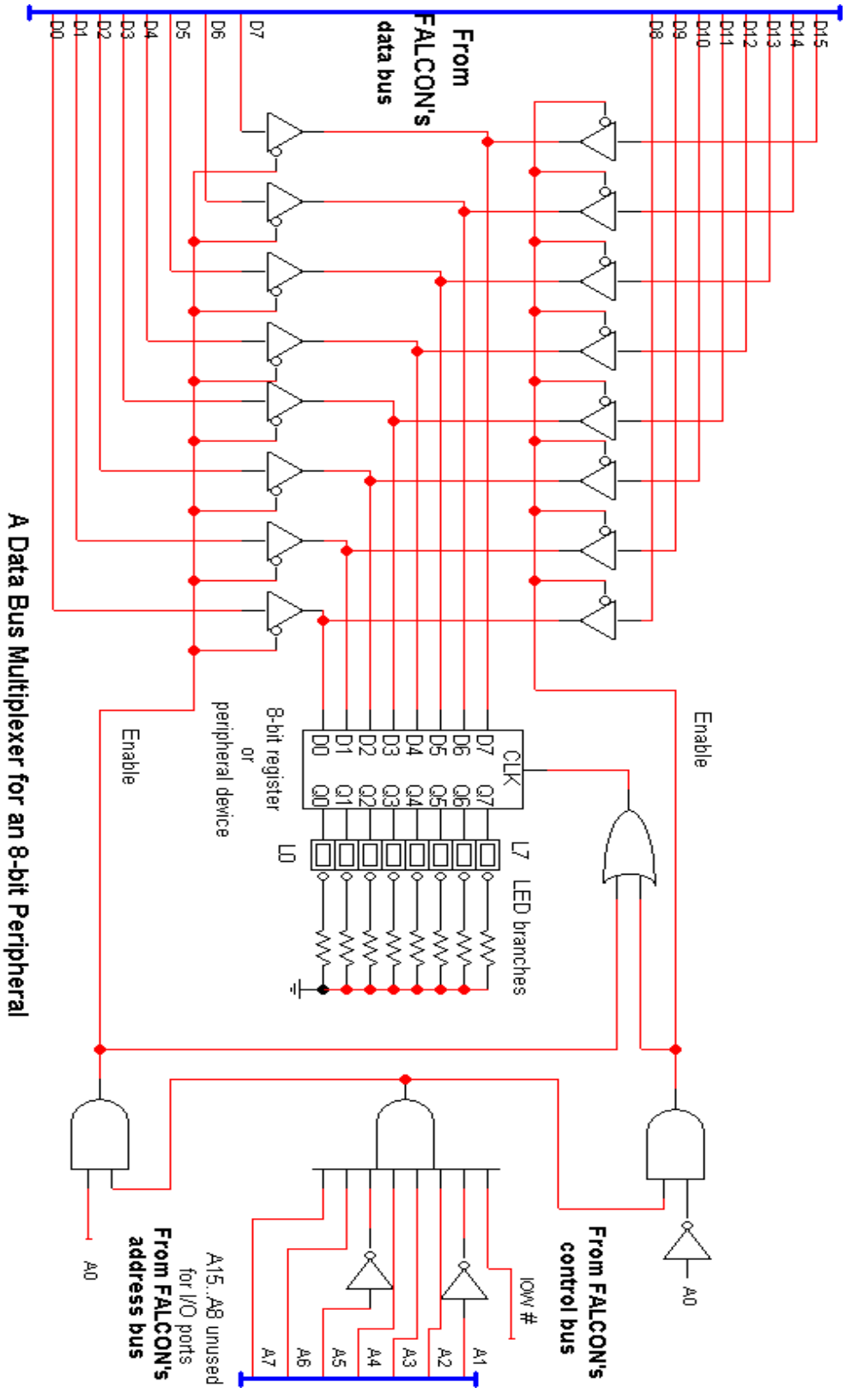
Partial decoding is a technique in which some of the CPU‟s address lines forming an input to the address decoder are ignored. This reduces the complexity of the address decoder, and also lowers the cost. As an example, if the address lines A8...A15 from the FALCON-A are not used in the address decoder of the previous figure, this will save eight inverters and two AND gates. Partial decoding is an attractive choice in small systems, where the size of the address space is large but most of the memory is unimplemented. However, partial decoding has its price as well. Consider the memory map for the




FALCON-A, shown again in the next figure. With 16 address lines, the total address space is $2^{16}$ = 64 Kbytes. When the upper eight address lines are unused, they become don‟t cares. The port shown in the previous figure will be accessed for address 00DEh. But, it will also be accessed for address 01DEh, 02DEh,....., FFDEh. In fact, the 64 Kbyte address space has been reduced to a 256 byte space. It “wrapped around” itself 256 times. If we only left 6 address lines, i.e., A15 ... A10, unconnected, then we will still have a “wrap around”, but of a different type. Now a 1 Kbyte (= $2^{10}$ ) address area will wrap around itself 64 times (= $2^{6}$ ).

## Data bus multiplexing

Data bus multiplexing refers to the situation when one part of the data bus is connected to the peripheral‟s data bus at one time and the second part of the data bus is connected to the peripheral‟s data bus at a different time in such a way that at one time, only one 8-bit portion of the data bus is connected to the peripheral.



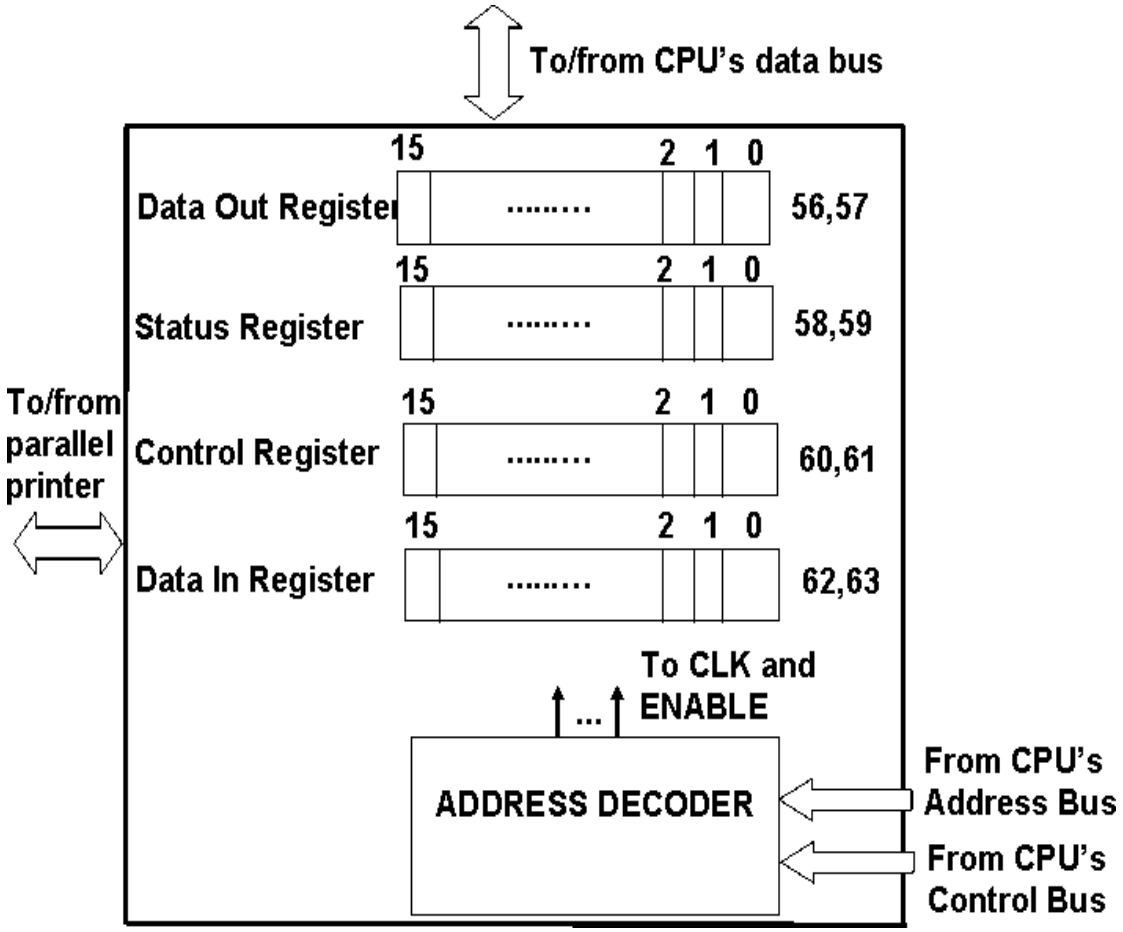

A Data Bus Multiplexer for an 8-bit Peripheral using address DCh and DDh.

Consider the situation where an 8-bit peripheral is to be interfaced with a CPU that has a 16-bit (or larger) data bus, but a byte-wide address space. Each byte transferred over the data bus will have a separate address associated with it. For such CPUs, data bus multiplexing can be used to attach 8-bit peripherals requiring a block of addresses. Tri-state buffers can be used for this

purpose as shown in the attached figure. The logic circuit shown is for an 8-bit parallel output port using addresses DCh and DDh of the FALCON‟s I/O address space. It is assumed that the CPU allows the use of a part of its data bus during a transfer, and that each 16-bit general purpose register can be used as two separate 8-bit registers, e.g., r1 can be split as r1L and r1H such that

**r1L<7..0> := r1<7..0>,** and
**r1H<7..0> := r1<15..8>**

The LED branches and the 8-bit register shown in the diagram serve as a place holder, and can be replaced by a peripheral device in actual practice. For an even address, A0=0, and the upper group of the tri-state buffers is enabled, thereby connecting D<15..8> of the CPU to the peripheral, while for an odd address from the CPU, A0=1, and the lower group of the tri-state buffers is enabled. This causes D<7..0> of the CPU to be connected with the peripheral device. In such systems the instruction **out r1H,220** will access the peripheral device using D<15..8>, while the instruction **out r1L,221** will access it using D<7..0>. The instruction **out r1,220** will send r1H to the peripheral and the contents of r1L will be lost. Why? This is left as an exercise for the student. The advantage of data bus multiplexing is that all addresses are utilized and none of them is wasted, while the disadvantage is the increased complexity and cost of the interface.




## A generic I/O interface

Most parallel I/O ports used with peripheral devices are mapped on a range of contiguous addresses. The following figure shows the block diagram of part of an interface that can be used with a typical parallel printer. It used eight consecutives addresses: address 56 to 63. A similar interface can be used with the FALCON-A. The registers shown within the interface are associated with some parallel device, and have some pre-defined functions. For example, the 16 bit register at addresses 56 and 57 can be used as a "data out" register for sending data bytes to the parallel device. In the same way, the register at addresses 60 and 61 can be used by the CPU to send control bits to the device. The double arrow shown at the top corresponds to the data bus connection of the interface with the CPU. The address decoder shown at the bottom receives address and control information from the CPU and generates enable signals for these registers. These abstract concepts are further explained in Example #3(lec25).

## The Centronics Parallel Printer Interface

The Centronics Parallel Printer Interface is an example of a real, industry standard, set of signal specifications used by most printer manufacturers. It was originally developed for Centronics printers and can be used by devices having a uni-directional, byte-wide parallel interface. Table 1 shows the important signals and their functions as defined by the Centronics standard. Note that the direction of the signals is with respect to the printer and not with respect to the CPU.

Typically, the printer (or any other similar device) is connected to the CPU via a cable which has a 25-pin connector at the CPU side and a 36-pin connector at the printer side. Every data bit in the 8-bit data bus D<7…0> uses a twisted pair for suppressing transmission-line effects, like radiation and noise. The return path of these pins should always be connected to signal ground. Additionally, the entire printer cable should be shielded, and connected to chassis ground on each side. The three signals STROBE#, BUSY and ACKNLG# form a set of handshaking signals. By using these signals, the CPU can communicate asynchronously with the printer, as shown in the accompanying timing waveforms. When the printer is ready for printing, the CPU starts data transfer to the printer by placing the 8-bit data (corresponding to the ASCII value of the character to be printed) on the printer‟s data bus (pin 2 through 9 on the 36-pin connector, as shown in Table 1). After this, a negative pulse of duration at least 0.5µs is applied to the STROBE# input (pin1) of the printer. The minimum set-up and hold times of the latches within the printer are specified as 0.5µs each, and these timing requirements must be observed by the CPU (the interface designer should make sure that these specifications are met). As soon as STROBE# goes low, the printer activates its BUSY line (pin 11) which is an indication to the CPU that additional bytes cannot be accepted. The CPU can monitor this status signal over an input port (a detailed assignment of these signals to I/O port bits is given in Table 2).

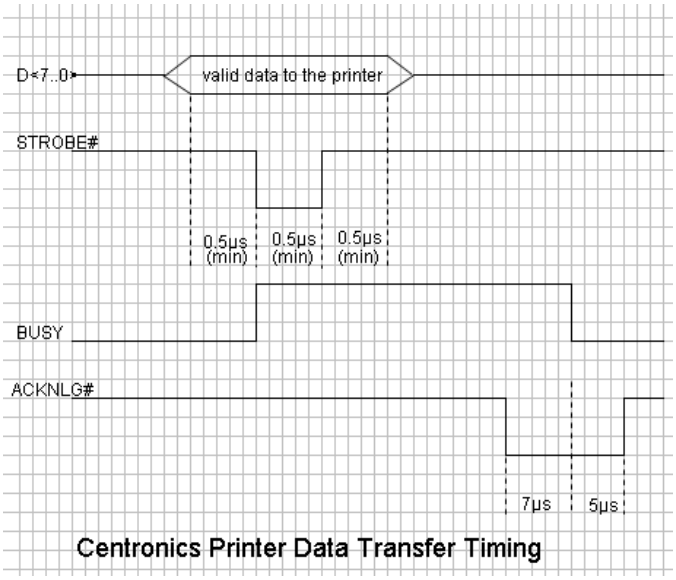
**Table 1: The Centronics Parallel Printer Interface**
**(power and ground signals are not shown)**

| Signal Name | Direction w.r.t. Printer | Function Summary | Pin# (25-DB) CPU side | Pin# (36-DB) Printer side |
|---|---|---|---|---|
| D<7..0> | Input | 8-bit data bus | 9,8,…,2 | 9,8,…,2 |
| STROBE# | Input | 1-bit control signal<br>High: default value.<br>Low: read-in of data is performed. | 1 | 1 |
| ACKNLG# | Output | 1-bit status signal<br>Low: data has been received and the printer is ready to | 10 | 10 |

| | | | | |
|---|---|---|---|---|
| | | accept new data.<br>High: default value. | | |
| BUSY | Output | 1-bit status signal<br>Low: default value<br>High: see note#1 | 11 | 11 |
| PE# | Output | 1-bit status signal<br>High: the printer is out of paper.<br>Low: default value. | 12 | 12 |
| INIT# | Input | 1-bit control signal<br>Low: the printer controller is reset to its initial state and the print buffer is cleared.<br>High: default value. | 16 | 31 |
| SLCT | Output | 1-bit status signal<br>High: the printer is in selected state. | 13 | 13 |
| AUTO FEED XT# | Input | 1-bit control signal<br>Low: paper is automatically fed after one line. | 14 | 14 |
| SLCT IN# | Input | 1-bit control signal<br>Low: data entry to the printer is possible.<br>High: data entry to printer is not Possible. | 17 | 36 |
| ERROR# | Output | 1-bit status signal<br>Low: see note#2.<br>High: default value. | 15 | 32 |

Note#1
The printer can not read data due to one of the following reasons:

1) During data entry
2) During data printing
3) In offline state
4) During printer error



Centronics Printer Data Transfer Timing

Note#2
When the printer is in one of the following states:

1) Paper end state
2) Offline state
3) Error state

When this character is completely received, the ACKNLG# signal (pin 10) goes low, indicating that the transfer is complete. Soon after this, the BUSY signal returns to logic zero, indicating that a new transfer can be initiated. The BUSY signal is more suitable for level-triggered systems, while the ACKNLG# signal is better for edge-triggered systems. The interface will typically use two eight bit parallel output ports of the CPU, one for the ASCII value of the character byte and the other for the control byte. It also specifies an 8-bit parallel input port for the printer‟s status information that can be checked by the CPU.

**Table 2: Centronics Bit Assignment For I/O Ports**

| **Logical Address** | **Description** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 8-bit output port for DATA | D<7> | D<6> | D<5> | D<4> | D<3> | D<2> | D<1> | D<0> |
| 1 | 8-bit input port for STATUS | BUSY | ACKNLG# | PE# | SLCT | ERROR# | Unused | Unused | Unused |
| 2 | 8-bit | Unus | Unused | DIR | IRQE | SLCT | INIT | Auto | STROB |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | output port for CONTROL | ed | | [12] | N | IN# | # | Feed XT# | E# |

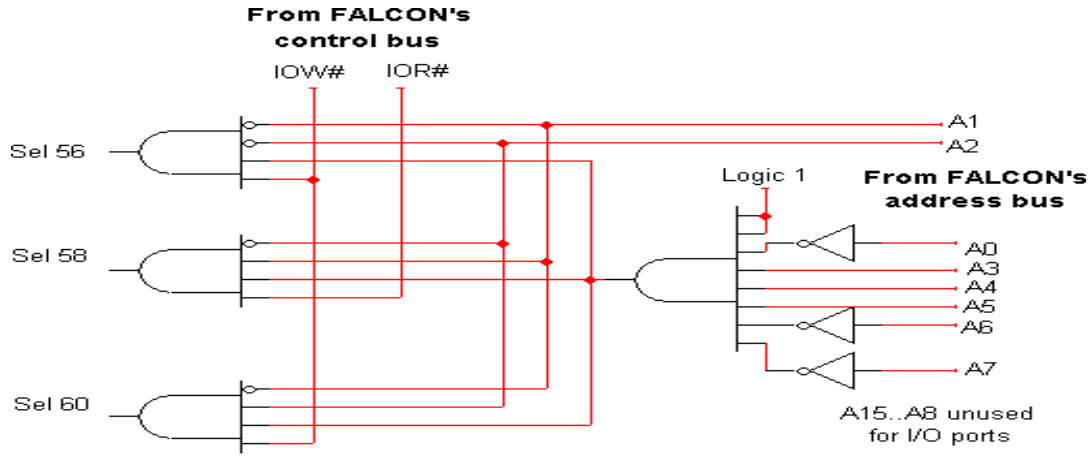
**Example # 3:**

Problem statement:

Design a Centronics parallel printer interface for the FALCON-A CPU. Map this interface starting at address 38h (56 decimal) of the FALCON-A‟s I/O address space.

Solution:

The Centronics interface requires at least three I/O addresses. However, since the FALCON-A has a 16-bit data bus, and since we do not want to implement data bus




**Address decoder for three parallel ports for the FALCON-A at addresses 38h, 3Ah, and 3Ch**

multiplexing (to keep things simple), we will use three contiguous even addresses, i.e., 38h, 3Ah and 3Ch for the address decoder design. This arrangement also conforms to the requirements of our assembler. Moreover, we will connect data bus lines D7...D0 of the FALCON-A to the 8-bit data bus of the printer (i.e. pins 9, 8, ... , 2 of the printer cable) and leave lines D15...D8 unconnected. Since the FALCON-A uses the big-endian format, this will make sure that the low byte of CPU registers will be transferred to the printer. (Recall that these bytes will actually be mapped on addresses 39h, 3Bh and 3Dh). The logic diagram of the address decoder for this interface is shown in the given figure.

[12] This bit, when set, enables the bidirectional mode.

# Lecture No. 26

## Programmed I/O

**Reading Material**

Vincent P. Heuring & Harry F. Jordan — Chapter 8
Computer Systems Design and Architecture — 8.2.2

**Summary**

- The Centronic Parallel Printer Interface(Cont.)
- Programmed Input/Output
- Examples of Programmed I/O for FALCON-A and SRC
- Comparisons of FALCON-A, SRC examples

## The Centronic Parallel Printer Interface (Cont.)

**(power and ground signals are not shown)**
(The explanation of this table is provided in lecture 25 also)

| Signal Name | Direction w.r.t. Printer | Function Summary | Pin# (25-DB) CPU side | Pin# (36-DB) Printer side |
|---|---|---|---|---|
| D<7..0> | Input | 8-bit data bus | 9,8,…,2 | 9,8,…,2 |
| STROBE# | Input | 1-bit control signal<br>High: default value.<br>Low: read-in of data is performed. | 1 | 1 |
| ACKNLG# | Output | 1-bit status signal<br>Low: data has been received and the printer is ready to accept new data.<br>High: default value. | 10 | 10 |
| BUSY | Output | 1-bit status signal<br>Low: default value<br>High: see note#1 | 11 | 11 |
| | | 1-bit status signal | | |

| | | | | |
|---|---|---|---|---|
| PE# | Output | High: the printer is out of paper.<br>Low: default value. | 12 | 12 |
| INIT# | Input | 1-bit control signal<br>Low: the printer controller is reset to its initial state and the print buffer is cleared.<br>High: default value. | 16 | 31 |
| SLCT | Output | 1-bit status signal<br>High: the printer is in selected state. | 13 | 13 |
| AUTO FEED XT# | Input | 1-bit control signal<br>Low: paper is automatically fed after one line. | 14 | 14 |
| SLCT IN# | Input | 1-bit control signal<br>Low: data entry to the printer is possible.<br>High: data entry to printer is not Possible. | 17 | 36 |
| ERROR# | Output | 1-bit status signal<br>Low: see note#2.<br>High: default value. | 15 | 32 |

**Table 2: Centronics Bit Assignment For I/O Ports**

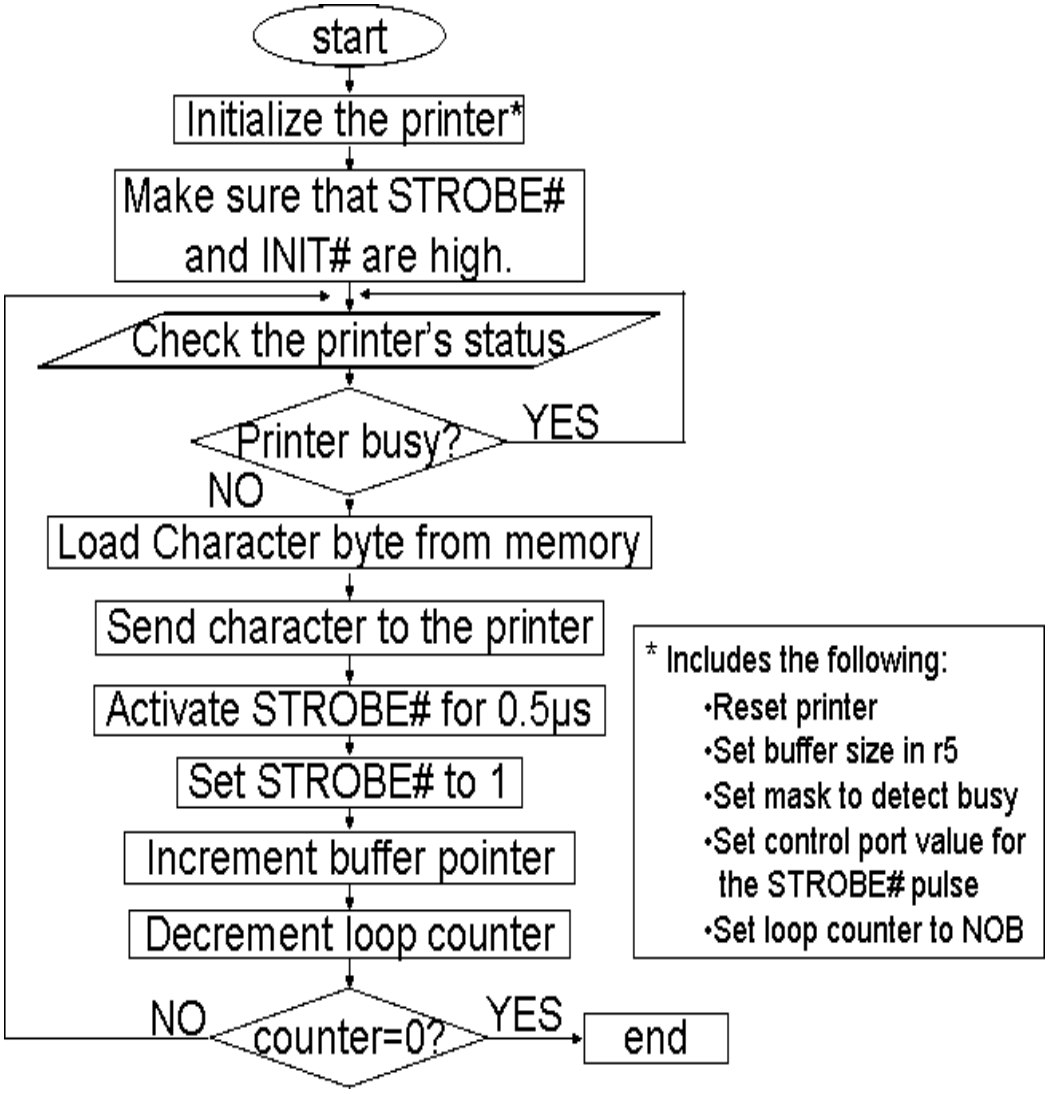
| **Logical Address** | **Description** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 8-bit output port for DATA | D<7> | D<6> | D<5> | D<4> | D<3> | D<2> | D<1> | D<0> |
| 1 | 8-bit input port for STATUS | BUSY | ACKNLG# | PE# | SLCT | ERROR# | Unused | Unused | Unused |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 8-bit output port for CONTROL | Unused | Unused | DIR[13] | IRQEN | SLCT IN# | INIT# | Auto Feed XT# | STROBE# |

[13] This bit, when set, enables the bidirectional mode.

**Example # 1**

Problem statement:




Problem statement:

Assuming that a Getronics parallel printer is interfaced to the FALCON-A processor, as shown in example 3 of lecture 25, write an assembly language program to send an 80 character line to the printer. Assume that the line of characters is stored in the memory starting at address 1024.

Solution:

The flowchart for the solution is shown in given figure and the program listing is shown in the textbox with filename: Example_1.

The first thing that needs to be done is the initialization of the printer. This means that a "reset" command should be sent to the printer. Using the information from Table 1, this

can be done by writing a 0 to bit 2 (i.e., INIT#) of the control register having logical address 2. In our example, this maps onto address 60 of the FALCON-A. (Remember to set this bit to logic 1 for normal operation of the printer). Then we make STROBE# high by placing logic 1 in bit 0 of the control register. Bit 1 and bit 3 should be 0 because we want to activate auto line feed and keep the printer in selected mode. Additionally, bit 4 and bit 5 should be 0 so that interrupts are disabled and the bi-directional mode is not selected. The complete control word is 0000 0001 and this value has been assigned to the variable reset in the program. The following instruction pair performs the reset operation:

```
movi  r1,  reset
out r1, controlp
```

As it is given that the starting address of the printer buffer is $1024^{14}$, so we place this address in r5. The mask to test the BUSY flag is placed in r3. The value for the mask is 80h. This corresponds to a logic 1 in bit 7 and logic zeros elsewhere for the status register having address 58 (logical address 1 in Table 1). Then the program enters a loop, called the polling loop, to test the status of the printer. If the printer is busy, the loop repeats. The following three instructions form the polling loop:

```
in  r1,  statusp
and r1, r1, r3
jnz r1, [again]
```

The status of the printer is placed in register r1, and bit 7 is tested for logic 0. If not so, the program repeats the status check operation.

When the printer is ready to accept a new character, it clears bit 7 (i.e., the BUSY bit) of the status register. At this time, the program picks the next character from the memory and sends it to the printer. The STROBE# line is activated and then it is deactivated to generate the necessary pulse on this input of the printer. Finally, the buffer pointer is advanced, the loop counter is decremented and the process repeats. When all the characters have been printed, the program halts.

A number of equates have been used in the program to make it flexible as well as easily readable. The program is shown on the next page.

[14] The **mul** instruction is used for this purpose because the 8-bit immediate operand in the **movi** instruction can only be within the range –128 and +127. Using the **mul** instruction in this way overcomes the limitation of the FALCON-A. Similarly, the **shiftl** instruction is used to bring 80h in register r3.

```
; filename: Example_1.asmfa
;
; This program sends an 80 character line
; to a FALCON-A parallel printer
;
; Notes:
; 1.  8-bit printer data bus connected to
;     D<7...0> of the FALCON-A (remember big-endian)
;     Thus, the printer actually uses addresses 57, 59 & 61
;
; 2.  one character per 16-bits of data xfered
;
;
      .org 400
;
NOB:        .equ  80
;
      movi r5, 32
      mul r5, r5, r5    ; r5 holds 1024 temporarily
;
      movi r3, 1
      shiftl r3, r3, 7  ; to set mask to 0080h
;
datap:      .equ 56
statusp:    .equ 58
controlp:   .equ 60
;
reset:            .equ 1
; used to set unidirectional, no interrupts,
; auto line feed, and strobe high
;
strb_H:           .equ 5
strb_L:           .equ 4
;
      movi r1 reset     ; use r1 for data xfer
      out r1, controlp
;
      movi r7, NOB      ; use r7 as character counter
;

again:      in r1, statusp
;
      and r1, r1, r3    ; test if BUSY = 1?
      jnz r1, [again]   ; wait if BUSY = 1
;
      load r1, [r5]
      out r1, datap
      movi r1, strb_L
      out r1, controlp
      movi r1, strb_H
      out r1, controlp
      addi r5, r5, 2
      subi r7, r7, 1
      jnz r7, [again]
      halt
```

## I/O techniques:

There are three main techniques using which a CPU can exchange data with a peripheral device, namely

- Programmed I/O
- Interrupt driven I/O
- Direct Memory Access (DMA)

In this section, we present the first one.

## Programmed Input/Output

Programmed I/O refers to the situation when all I/O operations are performed under the direct control of a program running on the CPU. This program, which usually consists of a "tight loop", controls all I/O activity, including device status sensing, issuing read or write commands, and transferring the data[15]. A subsequent I/O operation cannot begin until the current I/O operation to a certain device is complete. This causes the CPU to wait, and thus makes the scheme extremely inefficient. The solution to Example # 3(lec24), Example #2(lec25), and Example #1(lec26) are examples of programmed input/output. We will analyze the program for Example #1(lec26) to explain a few things related to the programmed I/O technique.

**Timing analysis of the program in Example # 1(lec26)**

The main loop of the program given in the solution to Example #1(lec26) executes 80 times. This is equal to the number of characters to be printed on one line. This portion of the program is shown again with the execution time of each instruction listed in brackets with it. The numbers shown are for a uni-bus CPU implementation. A complete list of execution times for all the FALCON-A"s instructions is given in Appendix A. A number of things can be noted now.

1. Assuming that the output at the hardware pins changes at the end of the (I/O write) bus cycle, the STROBE# signal will go from logic1

```
        movi r7, NOB          [2]
;
again:  in r1, statusp        [3]
        and r1 , r1, r3       [3]
        jnz  r1, [again]      [4]
;
        load r1, [r5]         [5]
        out r1, datap         [3]
        movi r1, strob_L      [2]
        out r1, controlp      [3]
        movi r1, s trob_H     [2]
        out r1, controlp      [3]
        addi r5, r5, 2        [3]
        subi r7, r7, 1        [3]
        jnz r7, [again]       [4]
        halt
```

[15] The I/O device has no direct access to the memory or the CPU, and transfer is generally done by using the CPU registers.

to logic 0 at the end of the instruction pair.

```
movi r1, strb_L     [2]
out r1, controlp    [3]
```

The execution time for these two instructions is 2+3 = 5 clock periods. Therefore, STROBE# stays at logic1 for at least 5 clock periods i.e., during these two instructions. For a 10MHz FALCON-A CPU, this will correspond to 5x100 = 500nsec = 0.5µsec. Since the data to the printer is being sent by the CPU using the two instructions (**load r1, [r5]** and **out r1, datap**) which are before the first **movi** instruction, the printer‟s data setup time requirement is satisfied as long as we do not increase the clock frequency beyond 10MHz.

After these two instructions, the next two instructions in the program cause STROBE# to go to logic 1 again.

```
movi r1, strb_H     [2]
out r1, controlp    [3]
```

These two instructions also take 5 clock periods, or 0.5µsec, to execute. Thus, the timing requirement of the STROBE# pulse width will also be satisfied as long as we do not increase the clock frequency beyond 10MHz. In case the frequency is greater than 10MHz, other instruction can be used in between these two pairs of instructions.

The printer‟s data hold time requirement is easily satisfied because there are a number of instructions after this **out** instruction which do not change the control port, and the character value is already present in the data register within the interface since the end of the **out r1, datap** instruction.

2. The three instructions given below:

```
again:  in r1, statusp   [3]
        and r1, r1, r3   [3]
        jnz r1, [again]  [4]
```

form what is called a “polling loop”. The process of periodically checking the status of a device to see if it is ready for the next I/O operation is called “polling”. It is the simplest way for an I/O device to communicate with the CPU. The device indicates its readiness by setting certain bits in a status register, and the CPU can read these bits to get information about the device. Thus, the CPU does all the work and controls all the I/O activities. The polling loop given above takes 10 clock periods. For a 10MHz FALCON-A CPU, this is 10x100=1µsec. One pass of the main loop takes a total of 3+3+4+5+3+2+3+2+3+3+3+4 = 38 clock periods which is 38x100 = 3.8µsec. This is the time that the CPU takes to send one character to the printer. If we assume that a 1000 character per second (cps) printer is connected to the CPU, then this printer has the

capability to print one character in every 1msec or every 1000µsec. So, after sending a character in 3.8µsec to the printer, the CPU will wait for about 996µsec before it can send the next character to the printer. This implies that the polling loop will be executed about 996 times for each character. This is indeed a very inefficient way of sending characters to the printer.

An improved way of doing this would be to include a memory of suitable size within the printer. This memory is also called a buffer, as explained earlier. The CPU can fill this buffer in a single "burst" at its own speed, and then do something else, while the printer picks up one character at a time from this buffer and prints it at its own speed. This is exactly the situation with today‟s printers. The task of generating the STROBE# pulse will also be done by the electronic circuits within the printer. In effect, a dedicated processor within the printer will do this job. However, if the buffer within the printer fills up, the CPU will still not be able to transfer additional data to it. A different handshaking scheme will then be needed to make the CPU to communicate asynchronously with the buffer in the printer, resulting in an inefficient operation again. This is explained below.

Assume that the printer has a FIFO type buffer of size 64 bytes that can be filled up without any delay at the time when the printer is not printing anything. When one or more character values are present in the buffer, the printer will pick up one value at a time and print it. Remember we have a 1000 cps printer, so it takes 1msec to print a character. The program for Example #1(lec26) is modified for this situation and is given below. All the assumptions are the same, unless otherwise mentioned.

```
again:    in r1, statusp [3]
          and r1, r1, r3 [3]
          jnz  r1, [again][4]
          load  r1,  [r5]  [5]
          out r1,  datap  [3]
          addi r5, r5, 2   [3]
          subi  r7,  r7,  1  [3]
          jnz r7, [again] [4]
```

Note that while the instructions for generating the STROBE# pulse have been eliminated, the polling loop is still there. This is necessary because the BUSY signal will still be present, although it will have a different meaning n now. In this case, BUSY =1 will mean that the buffer within the printer is full and it can not accept additional bytes.

The main loop shown in the program has an execution time of 28 clock periods, which is 2.8µsec for a 10MHz FALCON-A CPU. The polling loop still takes 10 clock periods or 1µsec. Assuming that this program starts when the buffer in the printer is empty, the outer loop will execute 64 times before the CPU encounters a BUSY=1 condition. After that the situation will be the same as in the previous case. The polling loop will execute for about 996 times before BUSY goes to logic 0. This situation will persist for the

remaining 16 characters (remember we are sending an 80 character line to the printer).

One can argue that the problem can be solved by increasing the buffer size to more than 80 bytes. Well, first of all, memory is not free. So, a large buffer will increase the cost of the printer. Even if we are willing to pay more for an improved printer, the larger buffer will still fill up whenever the number of characters is more than the buffer size. When that happens, we will be back to square one again.

A careful analysis of the situation reveals that there is something wrong with the scheme that is being used to send data to the printer. This problem of having a larger overhead of polling was recognized long ago, and therefore, interrupts were invented as an alternate to programmed I/O. Interrupt driven I/O will be the topic of the next lecture.

## Programmed I/O in SRC

In this section, we will discuss some more examples of programmed I/O with our example processor SRC which uses the memory mapped I/O technique.

### Program for Character Output

To understand how programmed I/O works in SRC, we will discuss a program which outputs the character to the printer. The first instruction loads the branch target and the second instruction loads the character into lower 8 bits of register r2. The 2-instruction loop reads the status register and tests the ready signal by checking its sign bit. It executes until the ready signal becomes logic one. On exit from the loop, the character is written to the device data register by the store instruction.

```
        lar r3, wait
        ldr r2, char
wait:   ld r1, COSTAT
        brpl r3, r1
        st r2, COUT
```

A 10 MIPS, SRC would execute 10,000 instructions waiting for a 1,000 character/sec printer.

### Program Fragment to Print 80-Character Line

The next example for the SRC is of a program which sends an 80-character line to a line printer with a command register. There are two nested loops starting at label wait. The two instruction inner loop, which waits for ready and the outer seven instruction loop which performs the following tasks.

- Outputs a character
  er
- Decrement the register containing the number of characters left to print
  haracters left to send.

The last two instructions issue the command to print the line.
The next example discussed from the book is of a driver program for 32-character input devices (Figure 8.10, Page 388).

## Comparisons of the SRC and FALCON-A Examples

The FALCON-A and SRC programmed I/O examples discussed are similar with some differences. In the first example discussed for the SRC (i.e. Character output), the control signal responsible for data transfer by the CPU is the ready signal while for FALCON-A Busy (active low)signal is checked. In the second example for the SRC, the instruction set, address width and no. of lines on address is different.

Although different techniques have been used to increase the efficiency of the programmed I/O, overheads due to polling can not be completely eliminated.

# Lecture No. 27

## Interrupt Driven I/O

**Reading Material**

Vincent P. Heuring & Harry F. Jordan — Chapter 8
Computer Systems Design and Architecture — 8.2.2

**Summary**

- Programmed I/O Driver for SRC
- Interrupt Driven I/O

## Programmed I/O Driver for SRC

Please refer to Figure 8.10 of the text and its associated explanation.

## Interrupt Driven I/O:

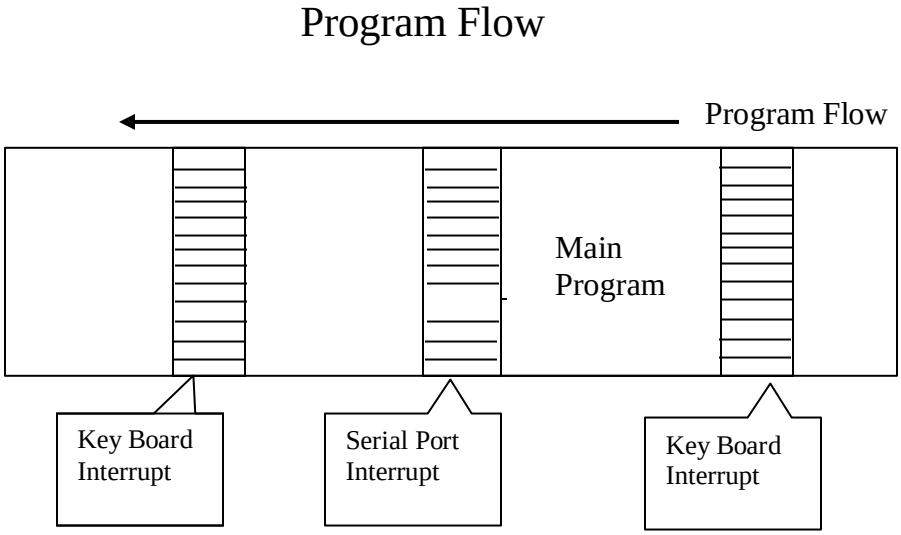
**Introduction:**

An interrupt is a request to the CPU to suspend normal processing and temporarily divert the flow of control through a new program. This new program to which control is transferred is called an Interrupt Service Routine or ISR. Another name for an ISR is an Interrupt Handler.

- Interrupts are used to demand attention from the CPU.
- Interrupts are asynchronous breaks in program flow that occur as a result of events outside the running program.
- Interrupts are usually hardware related, stemming from events such as a key or button press, timer expiration, or completion of a data transfer.

The basic purpose of interrupts is to divert CPU processing only when it is required. As an example let us consider the example of a user typing a document on word-processing software running on a multi tasking operating system. It is up to the software to display a character when the user presses a key on the keyboard. To fulfill this responsibility the processor can repeatedly poll the keyboard to check if the user has pressed a key. However, the average user can type at most 50 to 60 words in a minute. The rate of input is much slower than the speed of the processor. Hence, most of the polling messages that the processor sends to the keyboard will be wasted. A significant fraction of the

processor‟s cycles will be wasted checking for user input on the keyboard. It should also be kept in mind that there are usually multiple peripheral devices such as mouse, camera, LAN card, modem, etc. If the processor would poll each and every one of these devices for input, it would be wasting a large amount of its time. To solve this problem, interrupts are integrated into the system. Whenever a peripheral device has data to be exchanged with the processor, it interrupts the processor; the processor saves its state and then executes an interrupt handler routine (which basically exchanges data with the device).




After this exchange is completed, the processor resumes its task. Coming back to the keyboard example, if it takes the average user approximately 500 ms to press consecutive keys a modern processor like the Pentium can execute up to 300,000,000 instructions in these 500 Ms. Hence, interrupts are an efficient way to handle I/O compared to polling.

**Advantages of interrupts:**

- Useful for interfacing I/O devices with low data transfer rates.
- CPU is not tied up in a tight loop for polling the I/O device.

**Program Flow for an interrupt driven interface:**

The attached figure shows the program flow executing on a processor with interrupts enabled. As we can see, the program is interrupted in several locations to service various types of interrupts.

**Types of Interrupts:**

The general categories of interrupts are as follows:

- Internal Interrupts
- External Interrupts
  - Hardware Interrupts
  - Software Interrupts

**Internal Interrupts:**

- Internal interrupts are generated by the processor.
- These are used by processor to handle the exceptions generated during instruction execution

Internal interrupts are generated to handle conditions such as stack overflow or a divide-by-zero exception. Internal interrupts are also referred to as traps. They are mostly used for exception handling. These types of interrupts are also called exceptions and were discussed previously.

**External Interrupts:**
External interrupts are generated by the devices other than the processor. They are of two types.

- Hardware interrupts are generated by the external hardware.
- Software interrupts are generated by the software using some interrupt instruction.

As the name implies, external interrupts are generated by devices external to the CPU, such as the click of a mouse or pressing a key on a keyboard. In most cases, input from external sources requires immediate attention. These events require a quick service by the software, e.g., a word processing software must quickly display on the monitor, the character typed by the user on the keyboard. A mouse click should produce immediate results. Data received from the LAN card or the modem must be copied from the buffer immediately so that pending data is not lost because of buffer overflow, etc.

**Hardware interrupts:**
Hardware interrupts are generated by external events specific to peripheral devices. Most processors have at least one line dedicated to interrupt requests. When a device signals on this specific line, the processor halts its activity and executes an interrupt service routine. Such interrupts are always asynchronous with respect to instruction execution, and are not associated with any particular instruction. They do not prevent instruction completion as exceptions like an arithmetic overflows does. Thus, the control unit only needs to check for such interrupts at the start of every new instruction. Additionally, the CPU needs to know the identification and priority of the device sending the interrupt request.

There are two types of hardware interrupt:

- Maskable Interrupts
- Non-maskable Interrupts

**Maskable Interrupts:**

- These interrupts are applied to the INTR pin of the processor.
- These can be blocked by resetting the flag bit for the interrupts.

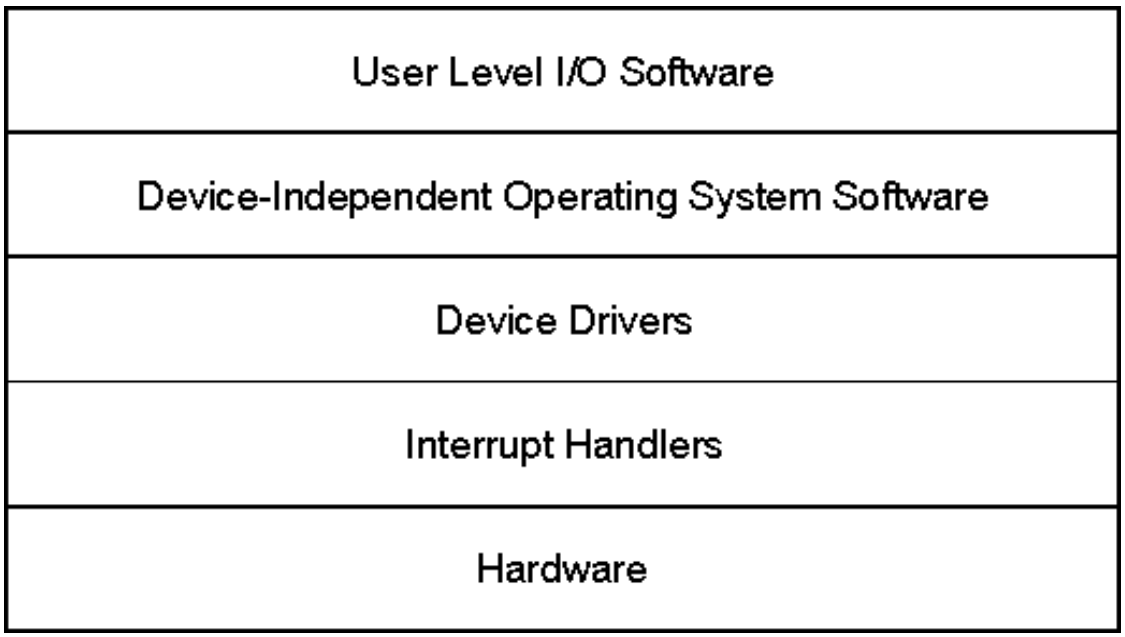
**Non-maskable Interrupts:**

- These interrupts are detected using the NMI pin of the processor.
- These can n
- Reserved for catastrophic event in the system.

**Software interrupts:**

Software interrupts are usually associated with the software. A simple output operation in a multitasking system requires software interrupts to be generated so that the processor may temporarily halt its activity and place the data on its data bus for the peripheral device. Output is usually handled by interrupts so that it appears interactive and asynchronous. Notification of other events, such as expiry of a software timer is also handled by software interrupts. Software interrupts are also used with system calls. When the operating system switches from user mode to supervisor mode it does so through software interrupts. Let us consider an example where a user program must delete a file. The user program will be executing in the user mode. When it makes the specific system call to delete the file, a software interrupt will be generated, this will cause the processor to halt its current activity (which would be the user program) and switch to supervisor mode. Once in supervisor mode, the operating system will delete the file and then control will return to the user program. While in supervisor mode the operating system would need to decide if it could delete the specified file with out harmful consequences to the systems integrity, hence it is important that the system switch to supervisor mode at each system call.

**I/O Software System Layers:**




The above diagram shows the various software layers related to I/O. At the bottom lies the actual hardware itself, i.e. the peripheral device. The peripheral device uses the hardware interrupts to communicate with the processor. The processor responds by executing the interrupt handler for that particular device. The device drivers form the bridge between the hardware and the software. The operating system uses the device drivers to communicate with the device in a hardware independent fashion, e.g., the

operating system need not cater for a specific brand of CRT monitors, or keyboards, the specific device driver written for that monitor or keyboard will act as an intermediary between the operating system and the device. It would be clear from the previous statement that the operating system expects certain common functions from all brands of devices in a category. Actually implementing these functions for each particular brand or vendor is the responsibility of the device driver. The user programs run at top of the operating system.

**Interrupt Service Routine (ISR):**

- It is a routine which is executed when an interrupt occurs.
- Also known as an Interrupt Handler.
- Deals with low-level events in the hardware of a computer system, like a tick of a real-time clock.

As it was mentioned earlier, an interrupt once generated must be serviced through an interrupt service routine. These routines are stored in the system memory ready for execution. Once the interrupt is generated, the processor must branch to the location of the appropriate service routine to execute it. The branch address of the ISR is discussed next.

**Branch Address of the ISR:**

There are two ways used to choose the branch address of an Interrupt Service Routine.

- Non-vectored Interrupts
- Vectored Interrupts

**Non-vectored Interrupts:**

In non-vectored interrupts, the branch address of the interrupt service routine is fixed. The code for the ISR is loaded at fixed memory location. Non-vectored interrupts are very easy to implement and not flexible at all. In this case, the number of peripheral devices is fixed and may not be increased. Once the interrupt is generated the processor queries each peripheral device to find out which device generated the interrupt. This approach is the least flexible for software interrupt handling.

**Vectored Interrupts:**

Interrupt vectors are used to specify the address of the interrupt service routine. The code for ISR can be loaded anywhere in the memory. This approach is much more flexible as the programmer may easily locate the interrupt vector and change its addresses to use custom interrupt servicing routines. Using vectored interrupts, multiple devices may share the same interrupt input line to the processor. A process called daisy chaining is then used to locate the interrupting device.

**Interrupt Vector:**

Interrupt vector is a fixed size structure that stores the address of the first instruction of the ISR.

**Interrupt Vector Table:**

- All of the interrupt vectors are stored in the memory in a special table called Interrupt Vector Table.
- Interrupt Vector Table is loaded at the memory location 0 for the 8086/8088.

**Interrupts in Intel 8086/8088:**

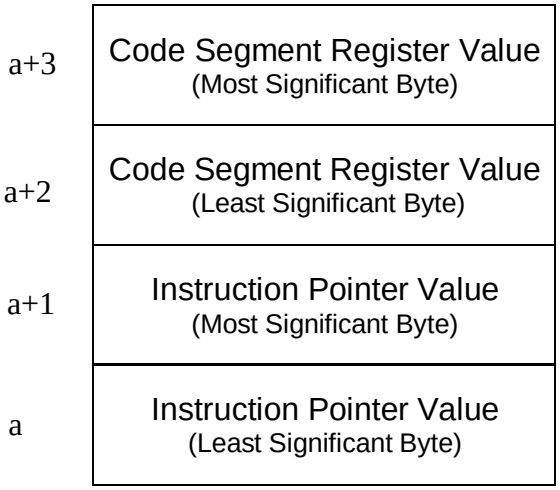
- Interrupts in 8086/8088 are vector interrupts.
- Interrupt vector is of 4 bytes to store IP and CS.
- Interrupt vector table is loaded at address 0 of main memory.
- There is provision of 256 interrupts.

**Branch Address Calculation:**

- The number of interrupt is the number of interrupt vector in the interrupt vector table.
- Since size of each vector is 4 bytes and interrupt vector starts from address 0, therefore, the address of interrupt vector can be calculated by simply multiplying the number by 4.

**Interrupt Vector Example:**

In 8086/8088 machines the size of interrupt vector is 4 bytes that holds IP and CS of ISR.




**Returning from the ISR:**

Every ISA should have an instruction, like the **IRET** instruction, which should be ates. This means that the **IRET** instruction should be the last instruction of every ISR. This is, in effect, a FAR RETURN in that it restores a number of registers, and flags to their value before the ISR was called. Thus the previous environment is restored after the servicing of the interrupt is completed.

**Interrupt Handling:**

The CPU responds to the interrupt request by completing the current instruction, and then

storing the return address from PC into a memory stack. Then the CPU branches to the ISR that processes the requested operation of data transfer. In general, the following sequence takes place.

**Hardware Interrupt Handling:**

- Hardware issues interrupt signal to the CPU.
- CPU completes the execution of current instruction.
- CPU acknowledges interrupt.
- Hardware places the interrupt number on the data bus.
- CPU determines the address of ISR from the interrupt number available on the data bus.
- CPU pushes the program status word (flags) on the stack along with the current value of program counter.
- The CPU starts executing the ISR.
- After completion of the ISR, the environment is restored; control is transferred back to the main program.

**Interrupt Latency:**

Interrupt Latency is the time needed by the CPU to recognize (not service) an interrupt request. It consists of the time to perform the following:

- Finish executing the current instruction.
- Perform interrupt-acknowledge bus cycles.
- Temporarily save the current environment.
- Calculate the [illegible]s and transfer control to the ISR.

If wait states are inserted by either some memory module or the device supplying the interrupt type number, the interrupt latency will increase accordingly.

Interrupt Latency for external interrupts depends on how many clock periods remain in the execution of the current instruction.

On the average, the longest latency occurs when a multiplication, division or a variable-bit shift or rotate instruction is executing when the interrupt request arrives.

**Response Deadline:**

It is the maximum time that an interrupt handler can take between the time when interrupt was requested and when the device must be serviced.

**Expanding Interrupt Structure:**

When there is more than one device that can interrupt the CPU, an Interrupt Controller is used to handle the priority of requests generated by the devices simultaneously.

**Interrupt Precedence:**

Interrupts occurring at the same time i.e. within the same instruction are serviced

according to a pre-defined priority.

- In general, all internal interrupts have priority over all external interrupts; the single-step interrupt is an exception.
- **NMI** has priority over **INTR** if both occur simultaneously.
- The above mentioned priority structure is applicable as far as the recognition of (simultaneous) interrupts is concerned. As far as servicing (execution of the related ISR) is concerned, the single-step interrupt always gets the highest priority, then the **NMI**, and finally those (hardware or software) interrupts that occur last. If IF is not 1, then **INTR** is ignored in any case. Moreover, since any ISR will clear **IF**, **INTR** has lower "service priority" compared to software interrupts, unless the ISR itself sets **IF**=1.

**Simultaneous Hardware Interrupt Requests**:
The priority of the devices requesting service at the same time is resolved by using two ways:

- Daisy-Chained Interrupt
- Parallel Priority Interrupt

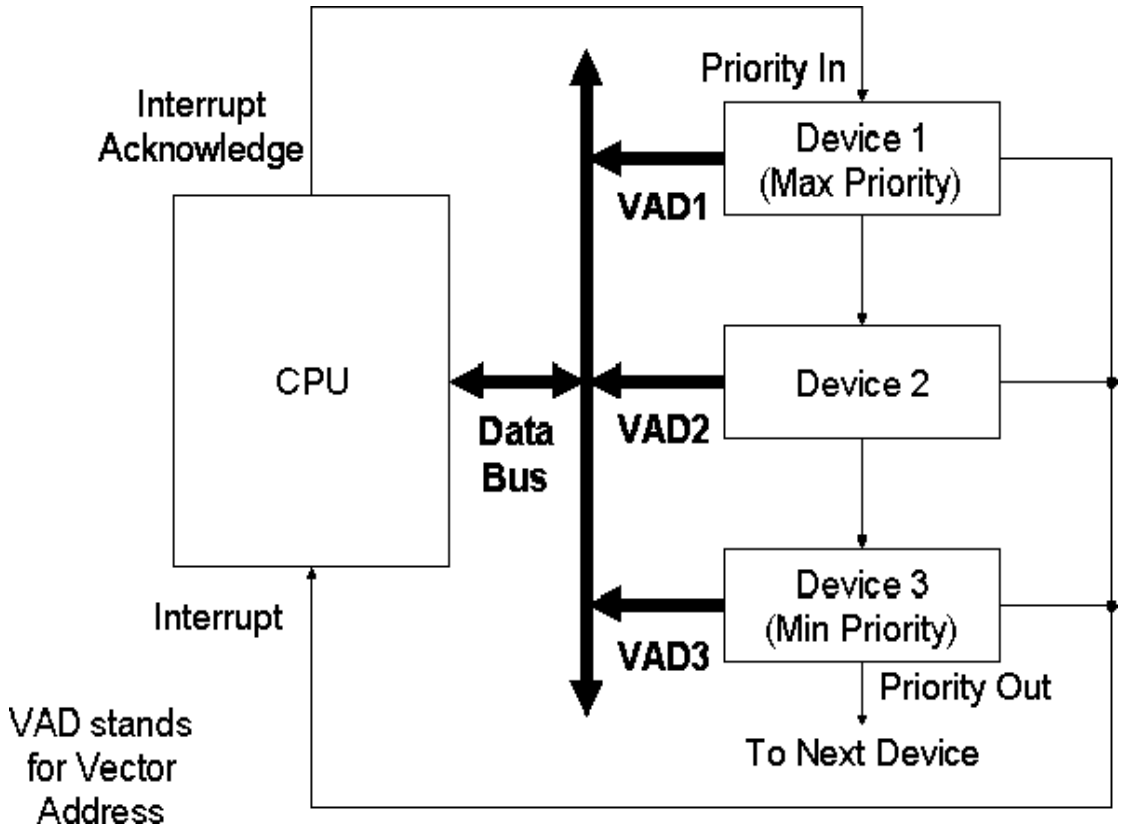
**Daisy-Chaining Priority:**

- The daisy-chaining method to resolve the priority consists of a series connection of the devices in order of their priority.
- Device with maximum priority is placed first and device with least priority is placed

**Daisy-Chain Priority Interrupt**

- The devices interrupt the CPU.
- The CPU sends acknowledgement to the maximum priority device.
- If the interrupt was generated by the device, the interrupt for the device is serviced.
- Otherwise the acknowledgement is passed to the next device.

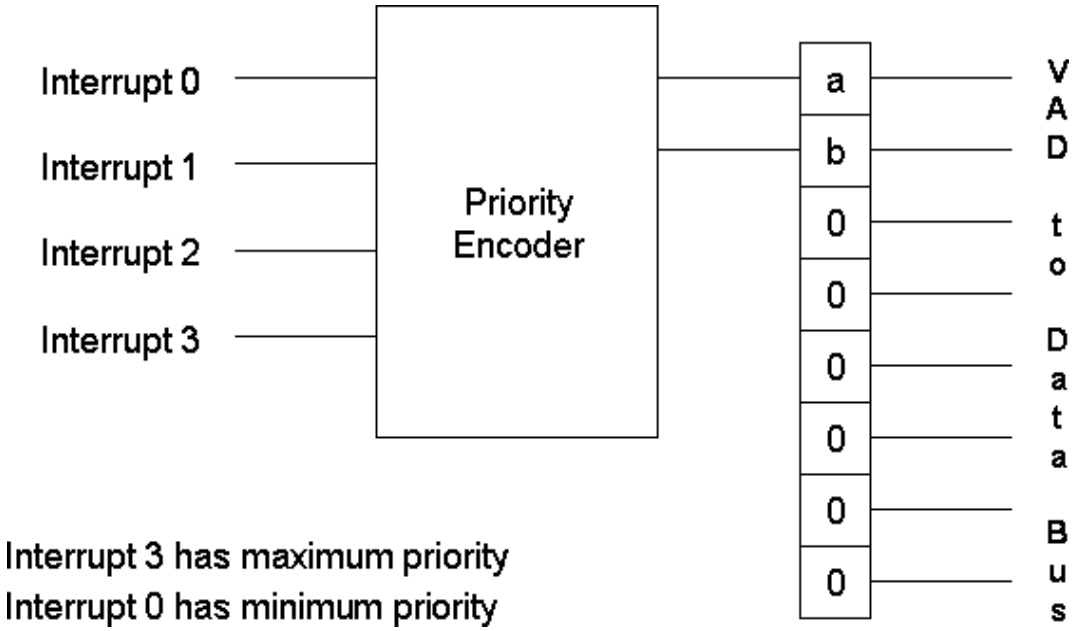
If the higher priority devices are going to interrupt continuously then the device with the lower priority is not serviced. So some additional circuitry is also needed to introduce fairness.




**Parallel Priority:**

- Parallel priority method for resolving the priority uses individual bits of a priority
- The priority of the device is determined by position of the input of the encoder used for the interrupt.

**Parallel Priority Interrupt:**




Interrupt 3 has maximum priority
Interrupt 0 has minimum priority

# Lecture No. 28

## Interrupt Hardware and Software

**Reading Material**

Vincent P. Heuring & Harry F. Jordan — Chapter 8
Computer Systems Design and Architecture — 8.3

### *Summary*

- Comparison of Interrupt driven I/O and Polling
- Design Issues
- Interrupt Handler Software
- Interrupt Hardware
- Interrupt Software

## Comparison of Interrupt driven I/O and Polling

Interrupt driven I/O is better than polling. In the case of polling a lot of time is wasted in questioning the peripheral device whether it is ready for delivering the data or not. In the case of interrupt driven I/O the CPU time in polling is saved.

Now the design issues involved in implementation of the interrupts are twofold. There would be a number of interrupts that could be initiated. Once the interrupt is there, how the CPU does know which particular device initiated this interrupt. So the first question is evaluation of the peripheral device or looking at which peripheral device has generated the interrupt. Now the second important question is that usually there would be a number of interrupts simultaneously available. So if there are a number of interrupts then there should be a mechanism by which we could just resolve that which particular interrupt should be serviced first. So there should be some priority mechanism.

## Design Issues

There are two design issues:

1. Device Identification
2. Priority mechanism

**Device Identification**
In this issue different mechanisms could be used.

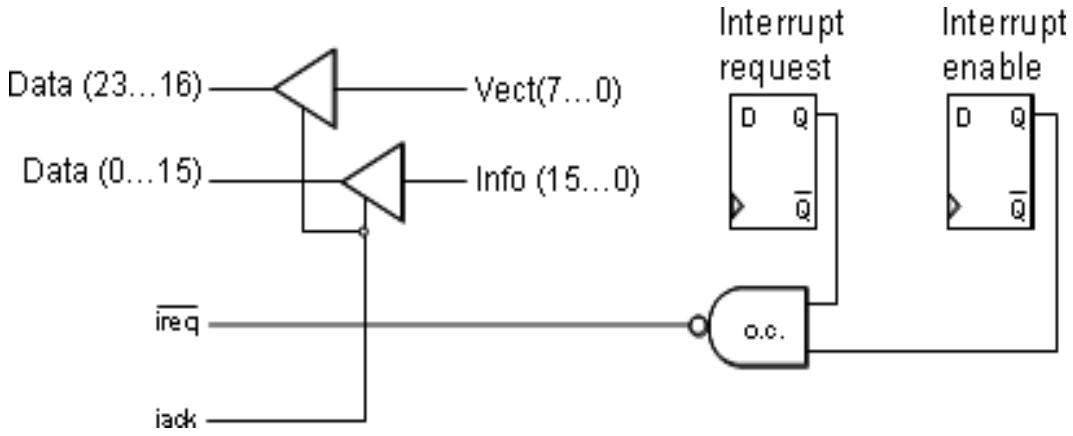
- Multiple interrupt lines
- Software Poll
- Daisy Chain

**1. Multiple Interrupt Line**

This is the most straight forward approach, and in this method, a number of interrupt lines are provided between the CPU and the I/O module. However, it is impractical to dedicate more than a few bus lines or CPU pins to interrupt lines. Consequently, even if multiple lines are used, it is likely that each line will have multiple I/O modules attached to it. Thus on each line, one of the other technique would still be required.

**2. Software Poll**

CPU polls to identify the interrupting module and branches to an interrupt service routine on detecting an interrupt. This identification is done using special commands or reading the device status register. Special command may be a test I/O. In this case, CPU raises test I/O and places the address of a particular I/O module on the address line. If I/O module sets the interrupt then it responds positively. In the case of an addressable status register, the CPU reads the status register of each I/O module to identify the interrupting




module. Once the correct module is identified, the CPU branches to a device service routine which is specific to that particular device.

**Simplified Interrupt Circuit for an I/O Interface**

For above two techniques the implementation might require some hardware. The hardware would be specific to the processor which is being used. For example, for the case of SRC, simple hardware machanism is indicated. Now the basic technique is

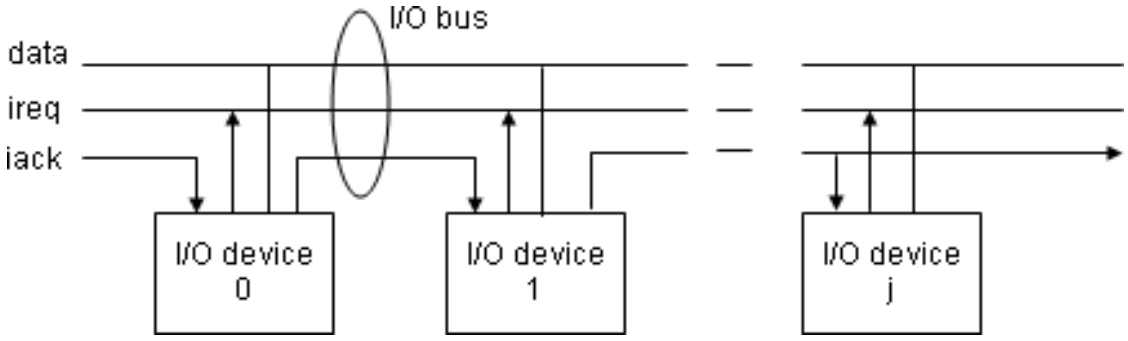
handshaking and in this case of handshaking, the peripheral device would initiate an interrupt. This interrupt needs to be enabled. We will have a mechanism of ANDing the two signals. One is interrupt enable and other is interrupt request. Now these two requests would be passed on the CPU. The CPU passes on the acknowledge signal to the device. The acknowledge signal is shared and it goes on to different devices.
The information about interrupt vector is given in 8-bits, from bit 0 to 7, which is translated to bit 16 to 23 on the data bus. Now the other 16-bits, from 0 to 15 are mapped to the data lines from 0 to 15. Now both of these are available through the tri-state buffers, which would be enabled through interrupt acknowledge.

**3. Daisy Chain**

The wired or interrupt signal allows several devices to request interrupt simultaneously. However, for proper operation one and only one requesting device must receive an acknowledge signal, otherwise if we have more than one devices, we would have a data bus contention and the interrupt information would not be resolved. The usual solution is called a daisy chain. Assuming that if we have jth devices requesting for interrupt then first device 0 would receive the acknowledge signal, so therefore, iack0=iack.

The next device would only receive an acknowledge i.e., the jth device would receive an acknowledge if the previous device that means j-1 does not have an enabled interrupt request, that means interrupt was not initiated by the previous device. Now the figure shows this concept in the form of a connection from device 0 to 1. From 0, we see the acknowledge is generated for device 1, device 1 generates acknowledge for device2 and so on. So this signal propagates from one device to other device. Logically we could write it in the form of equation:

$$iack_j = iack_{j-1} \wedge (req_{j-1} \wedge enb_{j-1})$$




As we said that the previous device should not have generated an interrupt, that means its interrupt was not enabled and therefore, it passes on the acknowledge signal from its output to he next device.

**Disadvantages of Software Poll and Daisy Chain**

The software poll has a disadvantage is that it consumes a lot of time, while the daisy

chain is more efficient. The daisy chain has the disadvantage that the device nearest to the CPU would have highest priority. So, usually those devices which require higher priority would be connected nearer to the CPU. Now in order to get a fair chance for other devices, other mechanisms could be initiated or we could say that we could start instead of device 0 from that device where the CPU finishes the last interrupt and could have a cyclic provision to different devices.

## Interrupt Handler Software

**Example using SRC**

(Read from Book, Jordan page395)

**Example using FALCON-A**

As an example of interrupt-driven I/O, consider an output device, such as a parallel printer connected to the FALCON-A CPU. Now suppose that we want to print a document while using an application program like a word processor or a spread sheet. In this section, we will explain the important aspects of hardware and software for implementing an interrupt driven parallel printer interface for the FALCON-A. During this discussion, we will also explain the differences and similarities between this interface and the one discussed earlier. To make things simple, we have made the assumption that only one interrupt pin is available on the FALCON-A, and only one interrupt is possible at a given time with this CPU. Implications of allowing only one interrupt at a time are that

- No NMI is possible
- No nesting of interrupts is possible
- No priority structure needed for multiple devices
- No arbitration needed for simultaneous interrupts
- No need for vectored interrupts, therefore, no need of interrupt vectors and interrupt vector tables
- Effect of software initiated interrupts and internal interrupts (exceptions) has to be ignored in this discussion

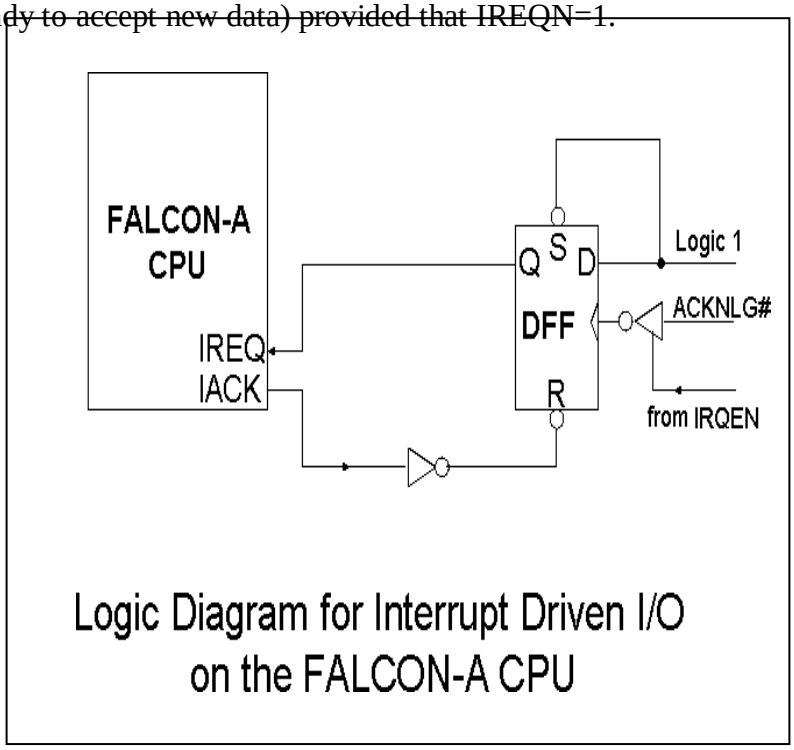
Along with the previous assumption, the following assumptions have also been used:

- Hardware sets and clears the interrupt flag, in addition to handling other things like saving PC, etc.
- The address of the ISR is stored at absolute address 2 in memory.
- The ISR will set up a stack in the memory for saving the CPU‟s environment

- One ASCII character stored per 16-bit word in the FALCON-A‟s memory and one character transferred during a 16-bit transfer.
- The calling program will call the ISR for printing the first character through the printer driver.
- Printer will activate ACKNLG# only when not BUSY.

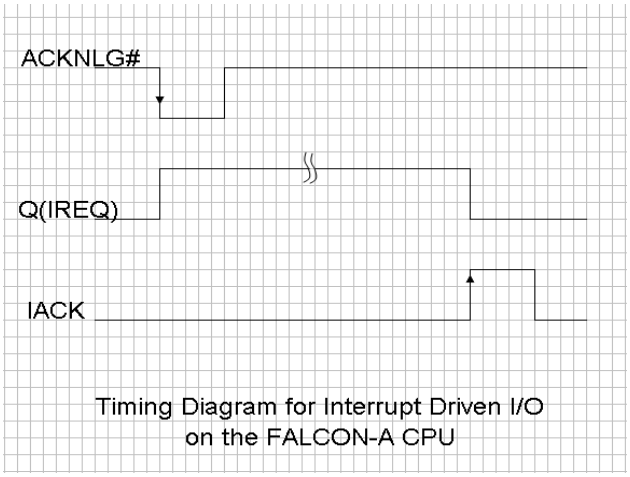
## Interrupt Hardware:

The logic diagram for the interrupt hardware is shown in the Figure. The interrupt request is synchronized by handshaking signals, called IREQ and IACK. The timing diagram for the handshaking signals used in the interrupt driven I/O is shown in the next Figure. The printer will assert IREQ as soon as the ACKNLG# signal goes low (i.e. as soon as the printer is ready to accept new data) provided that IREQN=1.




Logic Diagram for Interrupt Driven I/O on the FALCON-A CPU

The processor will complete the current instruction and respond by executing the interrupt service routine. The inverting tri-state buffer at the clock input of the D flip flop is enabled by IRQEN. This will make sure that after the current print job is complete, additional requests on IREQ are disabled. This can happen as a result of the printer being available even through the user may not have requested a print operation. The IACK line from the CPU is connected to the asynchronous reset, R, of the D flip flop so that the same interrupt request from the printer is not presented again to the CPU. The asynchronous set input of the D flip flop, labeled S in the diagram, is permanently connected to logic 1. This will make sure that the flip flop will never be set asynchronously. The D input is also permanently connected to logic 1, as a result of which the flip flop will always be set synchronously in response to ACKNLG# provided IRQEN=1. Recall that IRQEN is bit 4 on the centronics control port at logical address 2, and this is mapped onto address 60 of the FALCON-A‟s I/O space. The rest of the hardware is case of the same as in the case of the programmed I/O example.



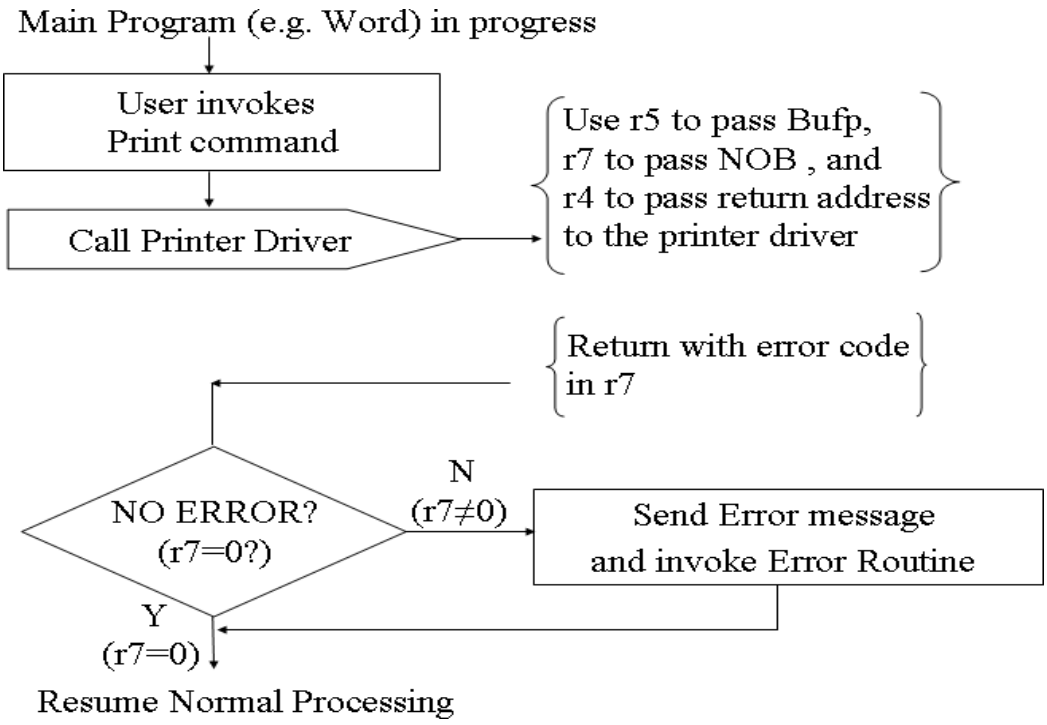
Timing Diagram for Interrupt Driven I/O on the FALCON-A CPU

## Interrupt Software:

Our software for the interrupt driven printer example consists of three parts:

1). Dummy calling program
2). Printer Driver
3). ISR

We are assuming that normal processing is taking place[16] e.g., a word processor is executing. The user wants to print a document. This




document is placed in a buffer by the word processor. This buffer is usually present somewhere else in the memory. The responsibility of the calling program is to pass the number of bytes to be printed and the starting address of the buffer where these bytes are stored to the printer driver. The calling program can also be called the main program.
Suppose that the total number of bytes to be printed are 40. (They are placed in a buffer having the starting address 1024.) When the user invokes the print command, the calling program calls the printer driver and passes these two parameters in r7 and r5 respectively. The return address of the calling program is stored in r4. A dummy calling program code is given below.
Bufp, NOB, PB, and temp are the spaces reserved in memory for later use in the program. The first instruction is **jump [main].** It is stored at absolute memory address 0 by using the **.org 0** directive. It will transfer control to the main program. The first instruction of the main program is placed at address "**main**", which is the entry point in this example. Note that the entry point is different in this case from the reset address, which is address 0 for the FALCON-A. Also note that the address of the first instruction in the printer driver is stored at address "**a4PD"** using the **.sw** directive. This value is then brought into r6.

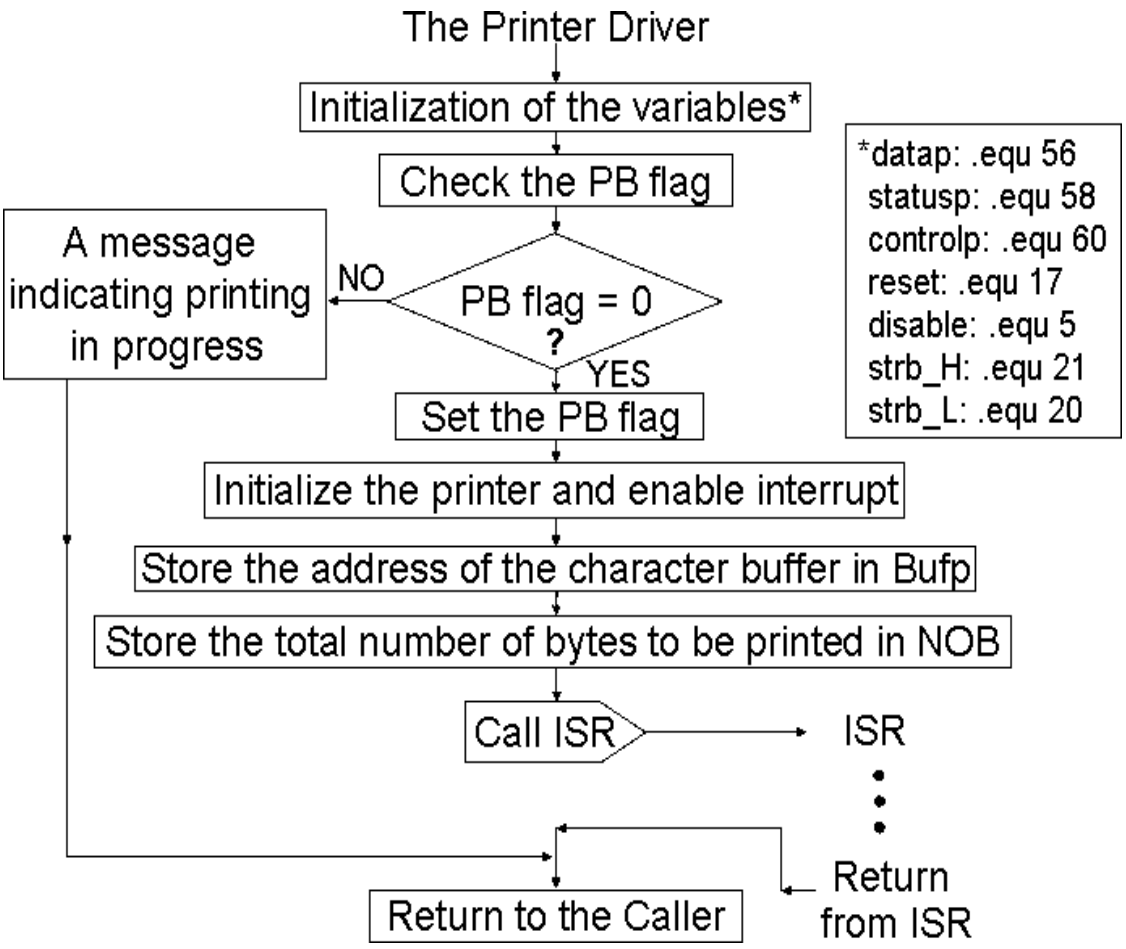
[16] Since only one interrupt is possible, a question may arise about the way the print command is presented to the word processor. It can be assumed that polling is used for the input device in this case.

The main program calls the printer driver by using the instruction **call r4, r6.** In an actual program, after returning from the printer driver, the normal processing resumes and if there are any error conditions, they will be handled at this point. Next, consider the code for the printer driver, shown in the attached text box.

```
; filename: Example_Falcon-A .asmfa
;This program sends a single character
;to a FALCON-A parallel printer
;using an interrupt driven I/O interface
;
; Notes:
; 1. 8-bit printer data bus connected to
;    D<7..0> of the FALCON-A (remember big-endian)
;    Thus, the printer actually uses addresses 57, 59 & 61
;
; 2. one character per 16-bits of data xfered ;
;
    .org 0
    jump [main]
a4ISR:    .sw beginISR
a4PD:     .sw  Pdriver
dv1:              .sw 1024
dv2:              .sw 40
Bufp:     .dw  1
NOB:      .dw  1
PB:       .dw 1
temp:     .dw 6
;
; Dummy Calling Program, e.g., a word processor
;
    .org 32
main:     load r6, [a4PD]        ;r6 holds address of printer driver
;
; user invokes print command here
;
    load r5, [dv1]          ;Prepare registers for passing
    load r7, [dv2]          ; information about print buffer.
;
;
; call printer driver
;
    call r4, r6
; Handle error conditions, if any , upon return.
```

```
; Normal processing resumes
;
    halt
```

The printer driver is loaded at address 50. Initialization of the variables includes setting of port addresses, variables for the STROBE# pulse, initializing the printer and enabling its IRQEN. The variables can be defined anywhere in the program because they reserve no memory space. When the printer driver starts, the PB flag is tested to make sure that a previous print job is not in progress. If so, the ISR is not invoked and a message is returned to the main program indicating that printing is in progress. This may display a "printer busy" icon on the user"s screen, or cause some other appropriate action. If the printer is available, it is initialized by the driver.




The following activities are also performed by the driver (see the attached flow chart also).

- Set port addresses

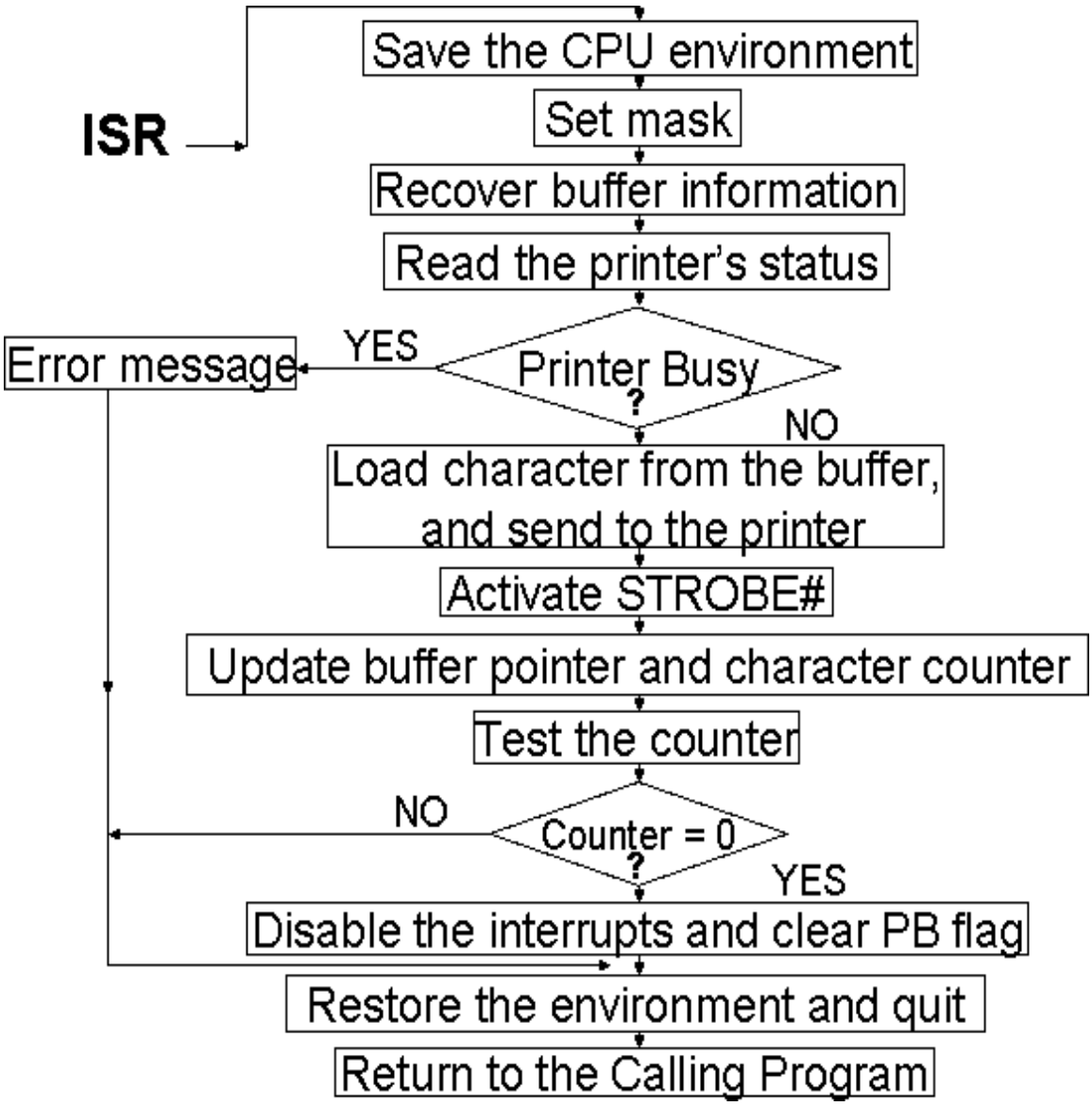
- Set up variables for the STROBE# puls
- Initialize printer and enable its IRQEN.
- Set up printer ISR by pointing to the buffer and initializing counter
- Make sure that the previous print job is not in progress
- Set PB flag to block further print jobs till current one is complete
- Invoke ISR for the first time
- Pass error message to main program if ISR reports an error
- Return to main program

The code and flow chart for the interrupt service routine (ISR) are discussed in the next few paragraphs.
We have assumed that the address of the ISR is stored at absolute memory address 2 by

```
; Printer driver
;
        .org 50                 ; starting address of Printer driver
;
datap:          .equ 56
statusp:        .equ 58
controlp:       .equ 60
;
reset:          .equ 17         ; or 11h
; used to set unidirectional, enable interrupts,
; auto line feed, and strobe high
disable:        .equ 5
;
strb_H:         .equ 21         ; or 15h
strb_L:         .equ 20         ; or 14h
;
; check PB flag first, if set,
; return with message.
;
Pdriver: load r1, [PB]
        jnz r1, [message]
        movi r1, 1
        store r1, [PB]          ; a 1 in PB indicates Print In Progress
        movi r1, reset          ; use r1 for data xfer
        out r1, controlp
        store r5, [Bufp]
        store r7, [NOB]
;
;
        int
;
        jump [finish]
message: nop                    ; in actual situation, put a message routine here
                                ;to indicate print in progress
finish:  ret r4
;
```

the operating system. One way to do that is by using the **.sw** directive (as done in the dummy calling program). The symbol **sw** stands for “storage of word”. It enables the user

to identify storage for a constant, or the value of a variable, an address or a label at a fixed memory location during the assembly process.




These values become part of the binary file and are then loaded into the memory when the binary file is loaded and executed. In response to a hardware interrupt or the software interrupt **int,** the control unit of the FALCON-A CPU will pick up the address of the first instruction in the ISR from memory location 2, and transfer control to it. This effectively means that the behavioral RTL of the **int** instruction will be as shown below:

**int** IPC← PC, PC ← M[2], IF ← 0

The IPC register in the CPU is a holding place for the current value of the PC. It is invisible to the programmer. Since the **iret** instruction should always be the last instruction in every ISR, its behavior RTL will be as shown below:

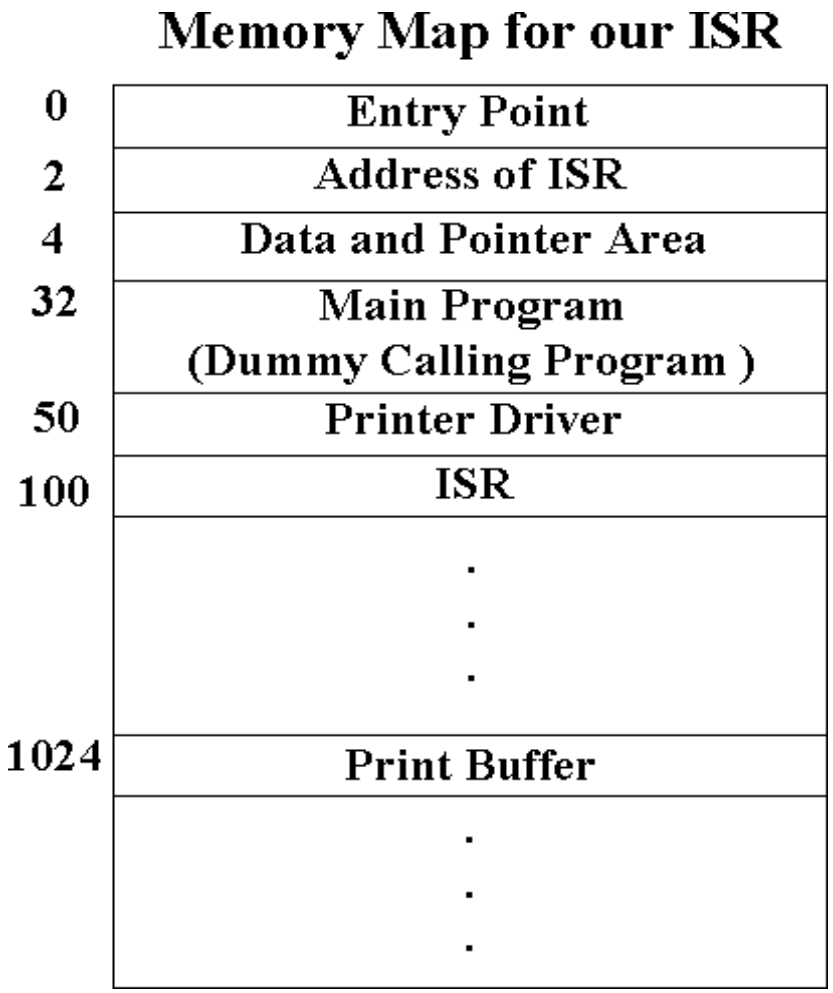
**iret** PC ← IPC, IF ← 1

The saving and restoring of the other elements of the CPU environment like the general purpose registers should be done within the ISR. The five **store** instructions at the beginning are used to save these registers into the memory block starting at address **temp**, and the five **load** instructions at the end are used to restore these registers to their original values.

```
;  ISR starts here
      .org 100
beginISR: movi r6, temp
      store r1, [r6]
      store r3, [r6+2]
      store r4, [r6+4]
      store r5, [r6+6]
      store r7, [r6+8]
      movi r3, 1
      shiftl r3,r3,7          ; to set mask to 0080h
      load r5, [Bufp]         ; not necessary to use r5 & r7 here
      load r7, [NOB]          ; using r7 as character counter
      in r1, statusp
      and r1,r1,r3            ; test if BUSY = 1 ?
      jnz r1, [error]         ; error if BUSY = 1
      load r1, [r5]           ; get char from printer buffer
      out r1, datap
      movi r1, strb_L
      out r1, controlp
      movi r1, strb_H
      out r1, controlp
      addi r5, r5, 2
      store r5, [Bufp]        ; update buffer pointer
      subi r7, r7, 1          ; update character counter
      store r7, [NOB]
      jz r7, [suspend]
      jump [last]
suspend: store r7, [PB]       ; clear PB flag
      movi r1, disable        ; disable future interrupts till
      out r1, controlp        ; printer driver called again
      jump [last]
error: movi r7, -1            ; error code in r7
```

```
; other error codes go here
;
last: load r1, [r6]
      load r3, [r6+2]
      load r4, [r6+4]
      load r5, [r6+6]
      load r7, [r6+8]
      iret
      .end
```

After setting the mask to 80h in r3, the current value of the buffer pointer and the number of bytes to be printed are brought from the memory into r5 and r7 respectively. After a byte is printed, these values are updated in the memory for use by the ISR when it is invoked again. The rest of the code in the ISR is the same as it was in case of the programmed I/O example. Note that we are testing the printer‟s BUSY flag within the ISR also. However, the difference here is that this testing is being done for a different reason, and it is done only once for each call to the ISR.




The memory map for this program is as shown in the Figure. The point to be noted here is that the ISR can be loaded anywhere in the memory but its address will be present at memory location 2 i.e. M[2].

# Lecture No. 29

## FALSIM

**Reading Material**

Handouts Slides

***Summary***

- Introduction to FALSIM
- Preparing source files for FALSIM
- Using FALSIM
- FALCON-A assembly language techniques

**Introduction to FALSIM:**

FALSIM is the name of the software application which consists of the FALCON-A assembler and the FALCON-A simulator. It runs under Windows XP.

**FALCON-A Assembler:**

Figure 1 shows a snapshot of the graphical user interface (GUI) for the FALCON-A Assembler. This tool loads a FALCON-A assembly file with a (.asmfa) extension and parses it. It shows the parsed results in an error log, lets the user view the assembled file‟s contents in the file listing and also provides the features of printing the machine code, an Instruction Table and a Symbol Table to a FALCON-A listing file. It also allows the user to run the FALCON-A Simulator.

The FALCON-A Assembler source code has two main modules, the 1st-pass module and the 2nd-pass module. The 1st-pass module takes an assembly file with a (.asmfa) extension and processes the file contents. It then generates a Symbol Table which corresponds to the storage of all program variables, labels and data values in a data structure at the implementation level. The Symbol Table is used by the 2nd-pass module. Failures of the 1st-pass are handled by the assembler using its exception handling mechanism.

The 2nd-pass module sequentially processes the .asmfa file to interpret the instruction op-codes, register op-codes and constants using the Symbol Table. It then produces a list file with a .lstfa extension independent of successful or failed pass. If the pass is successful a binary file with a .binfa extension is produced which contains the machine code for the program contained in the assembly file.

**FALCON-A Simulator**:

Figure 6 shows a snapshot of the GUI for the FALCON-A Simulator. This tool loads a FALCON-A binary file with a (.binfa) extension and presents its contents into different areas of the simulator. It allows the user to execute the program to a specific point within a time frame or just executes it, line by line. It also allows the user to view the registers, I/O port values and memory contents as the instructions execute.

**FALSIM Features**:

The FALCON-A Assembler provides its user with the following features:

*Select Assembly File:* Labeled as "**1**" in Figure 1, this feature enables the user to choose a FALCON-A assembly file and open it for processing by the assembler.

*Assembler Options:* Labeled as "**2**" in Figure 1.

- *Print Symbol Table*

This feature, if selected, writes the Symbol Table (produced after the execution of the 1st-pass of the assembler) to a FALCON-A list file with an extension of (.lstfa). The Symbol Table includes variables, addresses and labels with their respective values.

- *Print Instruction Table*

This feature, if selected, writes the FALCON-A instructions along with their op-codes at the end of the list file.

*List File:* Labeled as "**3**", in Figure 1, the List File feature gives a detailed insight of the FALCON-A listing file, which is produced as a result of the execution of the 1st and 2nd-pass. It shows the Program Counter value in hexadecimal and decimal formats along with the machine code generated for every line of assembly code. These values are printed when the 2nd-pass is completed.

*Error Log:* The Error Log is labeled as "**4**" in Figure 1. It informs the user about the errors and their respective details, which occurs in any of the two passes of the assembler. The size of this window can be changed by dragging the boundary line up or down.

*Highlight:* This feature is labeled as "**5**" in Figure 1 and helps the user to search for a certain input with the options of searching with "**match whole**" and "**match any**" parts of the string. The search also has the option of checking with/without considering "**case-sensitivity**". It searches the List File area and highlights the search results using the yellow color. It also indicates the total number of matches found.

*Start Simulator:* This feature is labeled as "**6**" in Figure 1. The FALCON-A Simulator is run using the FALCON-A Assembler"s "Start Simulator" option. Its features are detailed as follows:

*Load Binary File:* The button labeled as "**11**" in Figure 6, allows the user to choose and open a FALCON-A binary file with a (.binfa) extension. When a file is being loaded into the simulator all the register, constants (if any) and memory values are set.

*Registers:* The area labeled as "**12**" in Figure 6. enables, the user to see values present in different registers before, during and after execution.

*Instruction:* This area is labeled as "**13**" in Figure 6 and contains the value of PC, address of an instruction, its representation in Assembly, the Register Transfer Language, the op-code and the instruction type.

*I/O Ports:* I/O ports are labeled as "**14**" in Figure 6. These ports are available for the user to enter input operation values and visualize output operation values whenever an I/O operation takes place in the program. The input value for an input operation is given by the user before an instruction executes. The output values are visible in the I/O port area once the instruction has successfully executed.

*Memory:* The memory is divided into two areas and is labeled as "**15**" in Figure 6, to facilitate the view of data stored at different memory locations before, during and after program execution.

*Processor's State:* Labeled as "**16**" in Figure 6, this area shows the current values of the Instruction Register and the Program Counter while the program executes.

*Highlight:* The highlight option for the FALCON-A simulator is labeled as "**17**" in Figure 6. This feature is similar to the way the highlight feature of the FALCON-A Assembler works. It offers to highlight the search string which is entered as an input, with the "All " and " Part " option. The results of the search are highlighted using the yellow color. It also indicates the total number of matches.

The following is a description of the options available on the button panel labeled as "**18**" in Figure 6.

*Single Step:* "Single Step" lets the user execute the program, one instruction at a time. The next instruction is not executed unless the user does a "single step" again. By default, the instruction to be executed will be the one next in the sequence. It changes if the user specifies a different PC value using the Change PC option (explained below).

*Change PC:* This option lets the user change the value of PC (Program Counter). By changing the PC the user can execute the instruction to which the specified PC points. The value in the PC must be an even address.

*Execute:* By choosing this button, the user is able to execute the loaded program with the options of execution with/without breakpoint insertion. In case of breakpoint insertion, the user has the option to choose from a list of valid breakpoint values. It also has the option to set a limit on the time for execution. This "Max Execution Time" option restricts the program execution to a time frame specified by the user.

*Change Register:* Using the Change Register feature, the user can change the value present in a particular register.

*Change Memory Word:* This feature enables the user to change values present at a particular memory location.

*Display Memory:* Display Memory shows an updated memory area, after a particular memory location other than the pre-existing ones is specified by the

user.

*Change I/O:* Allows the user to give an I/O port value if the instruction to be executed requires an I/O operation. Giving in the input in any one of the I/O ports areas before instruction execution, indicates that a particular I/O operation will be a part of the program and it will have an input from some source. The value given by the user indicates the input type and source.

*Display I/O:* Display I/O works in a manner similar to Display Memory. Here the user specifies the starting index of an I/O port. This features displays the I/O ports stating from the index specified.

**2. Preparing Source Files for FALSIM:**

In order to use the FALCON-A assembler and simulator, FALSIM, the source file containing assembly language statements and directives should be prepared according to the following guidelines:

- The source file should contain ASCII text only. Each line should be terminated by a carriage return. The extension **.asmfa** should be used with each file name. After assembly, a list file with the original filename and an extension **.lstfa,** and a binary file with an extension **.binfa** will be generated by FALSIM.
- Comments are indicated by a semicolon (;) and can be placed anywhere in the source file. The FALSIM assembler ignores any text after the semicolon.
- Names in the source file can be of one of the following types:
- Variables: These are defined using the **.equ** directive. A value must also be assigned to variables when they are defined.
- Addresses in the "data and pointer area" within the memory: These can be defined using the **.dw** or the **.sw** directive. The difference between these two directives is that when **.dw** is used, it is not possible to store any value in the memory. The integer after **.dw** identifies the number of memory words to be reserved starting at the current address. (The directive **.db** can be used to reserve bytes in memory.) Using the **.sw** directive, it is possible to store a constant or the value of a name in the memory. It is also possible to use pointers with this directive to specify addresses larger than 127. Data tables and jump tables can also be set up in the memory using this directive.
- Labels: An assembly language statement can have a unique label associated with it. Two assembly language statements cannot have the same name. Every label should have a colon (:) after it.
- Use the **.org 0** directive as the first line in the program. Although the use of this line is optional, its use will make sure that FALSIM will start simulation by picking up the first instruction stored at address 0 of the memory. (Address 0 is called the reset address of the processor). A **jump [first]** instruction can be placed at address 0, so that control is transferred to the first executable statement of the main program. Thus, the label **first** serves as the identifier of the "entry point" in the source file. The **.org** directive can also be used anywhere in the source file to force code at a particular address in the memory.

- Address 2 in the memory is reserved for the pointer to the Interrupt Service Routine (ISR). The **.sw** directive can be used to store the address of the first instruction in the ISR at this location.
- Address 4 to 125 can be used for addresses of data and pointers[17]. However, the main program must start at address 126 or less[18], otherwise FALSIM will generate an error at the **jump [first]** instruction.
- The main program should be followed by any subprograms or procedures. Each procedure should be terminated with a **ret** instruction. The ISR, if any, should be placed after the procedures and should be terminated with the **iret** instruction.
- The last line in the source file should be the **.end** directive.
- The **.equ** directive can be used anywhere in the source file to assign values to variables.
- It is the responsibility of the programmer to make sure that code does not overwrite data when the assembly process is performed, or vice versa. As an example, this can happen if care is not exercised during the use of the **.org** directive in the source file.

**3. Using FALSIM:**

- To start FALSIM (the FALCON-A assembler and simulator), double click on the FALSIM icon. This will display the assembler window, as shown in the Figure 1.
- Select one or both assembler options shown on the top right corner of the assembler window labeled as **"2"**. If no option is selected, the symbol table and the instruction table will not be generated in the list (.lstfa) file.
- Click on the select assembly file button labeled as **"1"**. This will open the dialog box as shown in the Figure 2.
- Select the path and file containing the source program that is to be assembled.
- Click on the open button. FALSIM will assemble the program and generate two files with the same filename, but with different extensions. A list file will be generated with an extension .lstfa, and a binary (executable) file will be generated with an extension .binfa. FALSIM will also display the list file and any error messages in two separate panes, as shown in Figure 3.
- Double clicking on any error message highlights and displays the corresponding erroneous line in the program listing window pane for the user. This is shown in Figure 4. The highlight feature can also be used to display any text string, including statements with errors in them. If the assembler reported any errors in the source file, then these errors should be corrected and the program should be assembled again before simulation can be done. Additionally, if the source file had been assembled correctly at an earlier occasion, and a correct binary (.binfa)

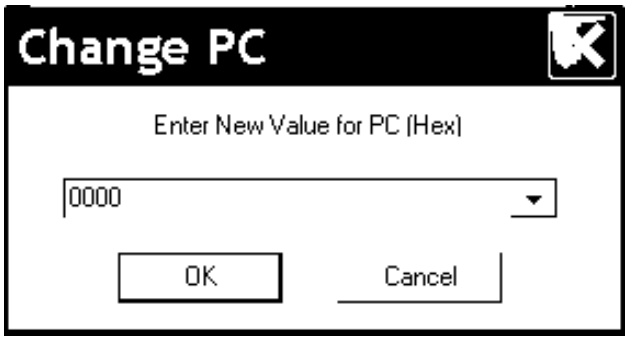
[17] Any address between 4 and 14 can be used in place of the displacement field in load or store instructions. Recall that the displacement field is just 5 bits in the instruction word.

[18] This restriction is because of the fact that the immediate operand in the **movi** instruction must fit an 8-bit field in the instruction word.

file exists, the simulator can be started directly without performing the assembly process.

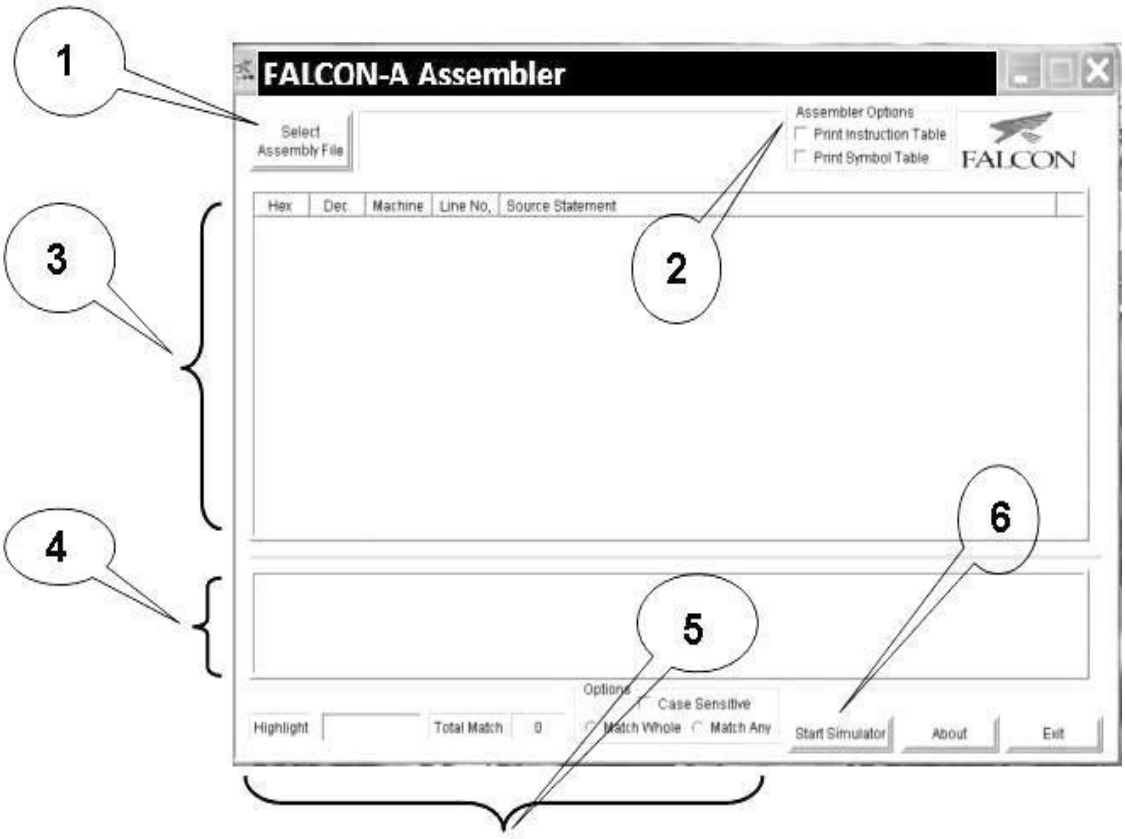
- To start the simulator, click on the start simulation button labeled as "**6"**. This will open the dialog box shown in Figure 6.
- Select the binary file to be simulated, and click **Open** as shown in Figure 7. (It is also possible to open the file by double clicking on the file name in the "Open" window).
- This will open the simulation window with the executable program loaded in it as shown in Figure 8. The details of the different panes in this window were given in section 1 earlier. Notice that the first instruction at address 0 is ready for execution. All registers are initialized to 0. The memory contains the address of the ISR (i.e., 64h which is 100 decimal) at location 2 and the address of the printer driver at location 4. These two addresses are determined at assembly time in our case. In a real situation, these addresses will be determined at execution time by the operating system, and thus the ISR and the printer driver will be located in the memory by the operating system (called re-locatable code). Subsequent memory locations contain constants defined in the program.
- Click single step button labeled as "**19**". FALSIM will execute the **jump [main]** instruction at address 0 and the PC will change to 20h (32 decimal), which is the address of the first instruction in the main program (i.e., the value of main).
- Although in a real situation, there will be many instructions in the main program, those instructions are not present in the dummy calling program. The first useful instruction is shown next. It loads the address of the printer driver in r6 from the pointer area in the memory. The registers r5 and r7 are also set up for passing the starting address of the print buffer and the number of bytes to be printed. In our dummy program, we bring these values in to these registers from the data area in the memory, and then pass these values to the printer driver using these two registers. Clicking on the single step button twice, executes these two instructions.
- The execution of the call instruction simulates the event of a print request by the user. This transfers control to the printer driver. Thus, when the **call r4, r6** instruction is single stepped, the PC changes to 32h (50 decimal) for executing the first instruction in the printer driver.
- Double click on memory location 000A, which is being used for holding the PB (printer busy) flag. Enter a 1 and click the change memory button. This will store a 0001 in this location, indicating that a previous print job is in progress. Now click single step and note that this value is brought from memory location 000E into register r1. Clicking single step again will cause the **jnz r1, [message]** instruction to execute, and control will transfer to the message routine at address 0046h. The **nop** instruction is used here as a place holder.
- Click again on the single step button. Note that when the **ret r4** instruction executes, the value in r4 (i.e., 28h) is brought into the PC. The blue highlight bar is placed on the next instruction after the **call r4, r6** instruction in the main program. In case of the dummy calling program, this is the **halt** instruction.

- Double click on the value of the PC labeled as "**20**". This will open a dialog box shown below. Enter a value of the PC (i.e., 26h) corresponding to the **call r4, r6** instruction, so




- that it can be executed again. A "list" of possible PC values can also be pulled down using, and 0026h can be selected from there as well.
- Click single step again to enter the printer driver again.
- Change memory location 000A to a 0, and then single step the first instruction in the printer driver. This will bring a 0 in r1, so that when the next **jnz r1, [message]** instruction is executed, the branch will not be taken and control will transfer to the next instruction after this instruction. This is **movi r1, 1** at address 0036h.
- Continue single stepping.
- Notice that a 1 has been stored in memory location 000A, and r1 contains 11h, which is then transferred to the output port at address 3Ch (60 decimal) when the **out r1, controlp** instruction executes. This can be verified by double clicking on the top left corner of the I/O port pane, and changing the address to 3Ch. Another way to display the value of an I/O port is to scroll the I/O window pane to the desired position.
- Continue single stepping till the **int** instruction and note the changes in different panes of the simulation window at each step.
- When the **int** instruction executes, the PC changes to 64h, which is the address of the first instruction in the ISR. Clicking single step executes this instruction, and loads the address of **temp** (i.e., 0010h) which is a temporary memory area for storing the environment. The five **store** instructions in the ISR save the CPU environment (working registers) before the ISR change them.
- Single step through the ISR while noting the effects on various registers, memory locations, and I/O ports till the **iret** instruction executes. This will pass control back to the printer driver by changing the PC to the address of the **jump [finish]** instruction, which is the next instruction after the **int** instruction.
- Double click on the value of the PC. Change it to point to the **int** instruction and click single step to execute it again. Continue to single step till the **in r1, statusp** instruction is ready for execution.

- Change the I/O port at address 3Ah (which represents the status port at address 58) to 80 and then single step the **in r1, statusp** instruction. The value in r1 should be 0080.
- Single step twice and notice that control is transferred to the **movi r7, FFFF**[19] instruction,which stores an error code of –1 in r1.




[19] The instruction was originally **movi r7, -1.** Since it was converted to machine language by the assembler, and then reverse assembled by the simulator, it became **movi r7, FFFF.** This is because the machine code stores the number in 16-bits after sign-extension. The result will be the same in both cases.

Figure 1

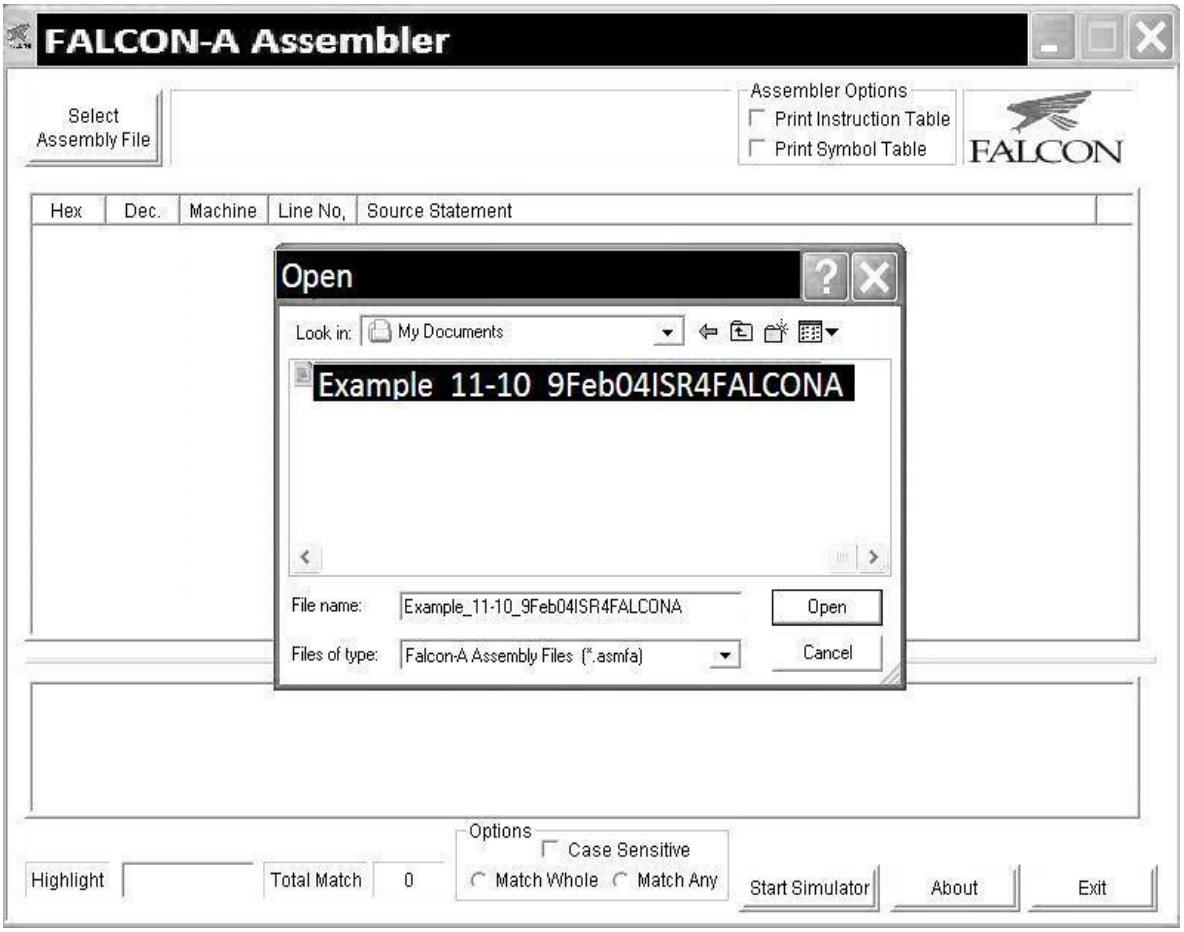



Figure 2


FALCON-A Assembler
Select Assembly File
Assembler Options
Print Instruction Table
Print Symbol Table
FALCON
Hex
Dec.
Machine
Line No,
Source Statement
0000 0000 0000 1 ; filename: Example_11-10
0000 0000 0000 2 ;This program sends a single character
0000 0000 0000 3 ;to a FALCON-A parallel printer
DlgFalconA_7Feb04
1 Error(s) During Second Pass
See C:\Documents and Settings\javaria\Desktop\Example_11-10_7Feb04ISR4FALCONA.lstfa
OK
Error: Line 62: Undefined variable `r2'
Options
Case Sensitive
Match Whole
Match Any
Highlight
Total Match
0
Start Simulator
About
Exit

Figure 4

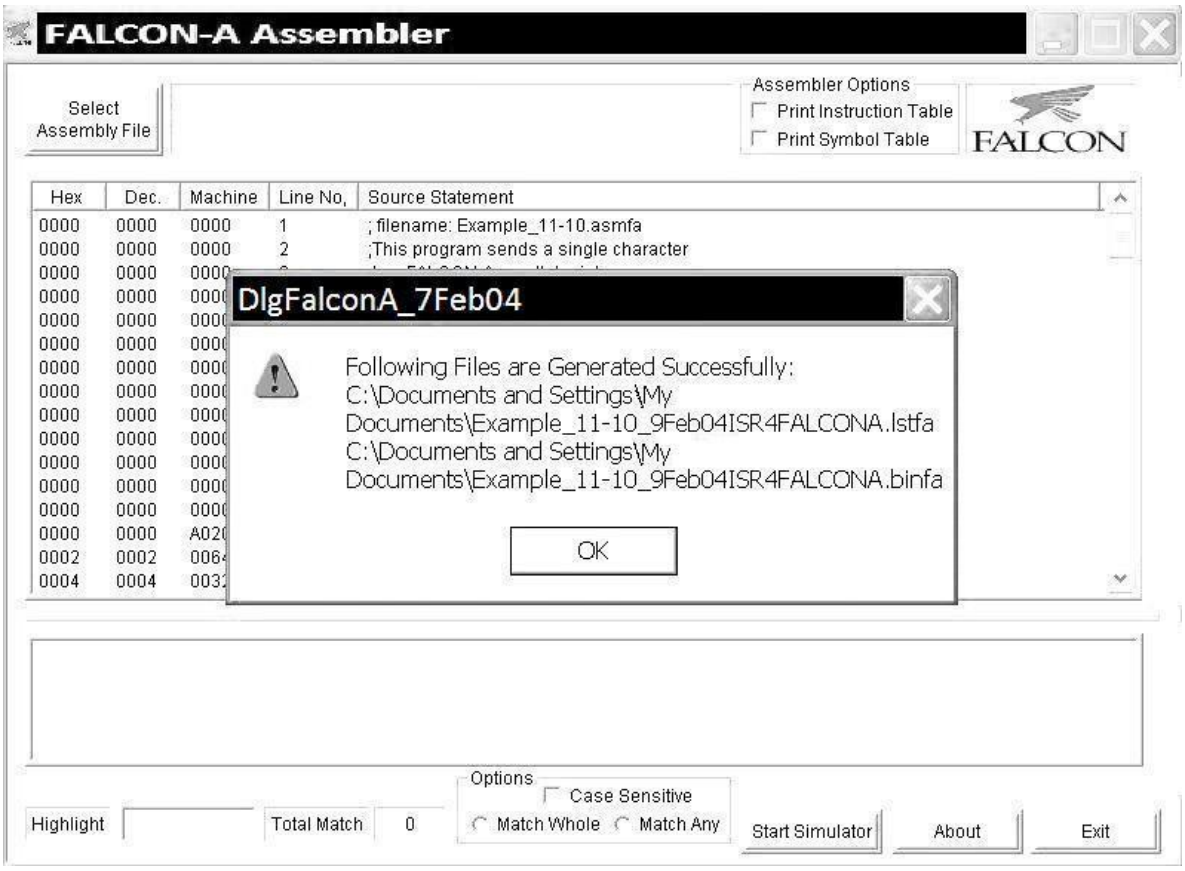

FALCON-A Assembler
Select Assembly File
Assembler Options
Print Instruction Table
Print Symbol Table
FALCON
Hex
Dec.
Machine
Line No,
Source Statement
0000 0000 0000 1 ; filename: Example_11-10.asmfa
0000 0000 0000 2 ;This program sends a single character
DlgFalconA_7Feb04
Following Files are Generated Successfully:
C:\Documents and Settings\My Documents\Example_11-10_9Feb04ISR4FALCONA.lstfa
C:\Documents and Settings\My Documents\Example_11-10_9Feb04ISR4FALCONA.binfa
OK
Options
Case Sensitive
Match Whole
Match Any
Highlight
Total Match
0
Start Simulator
About
Exit

Figure 5

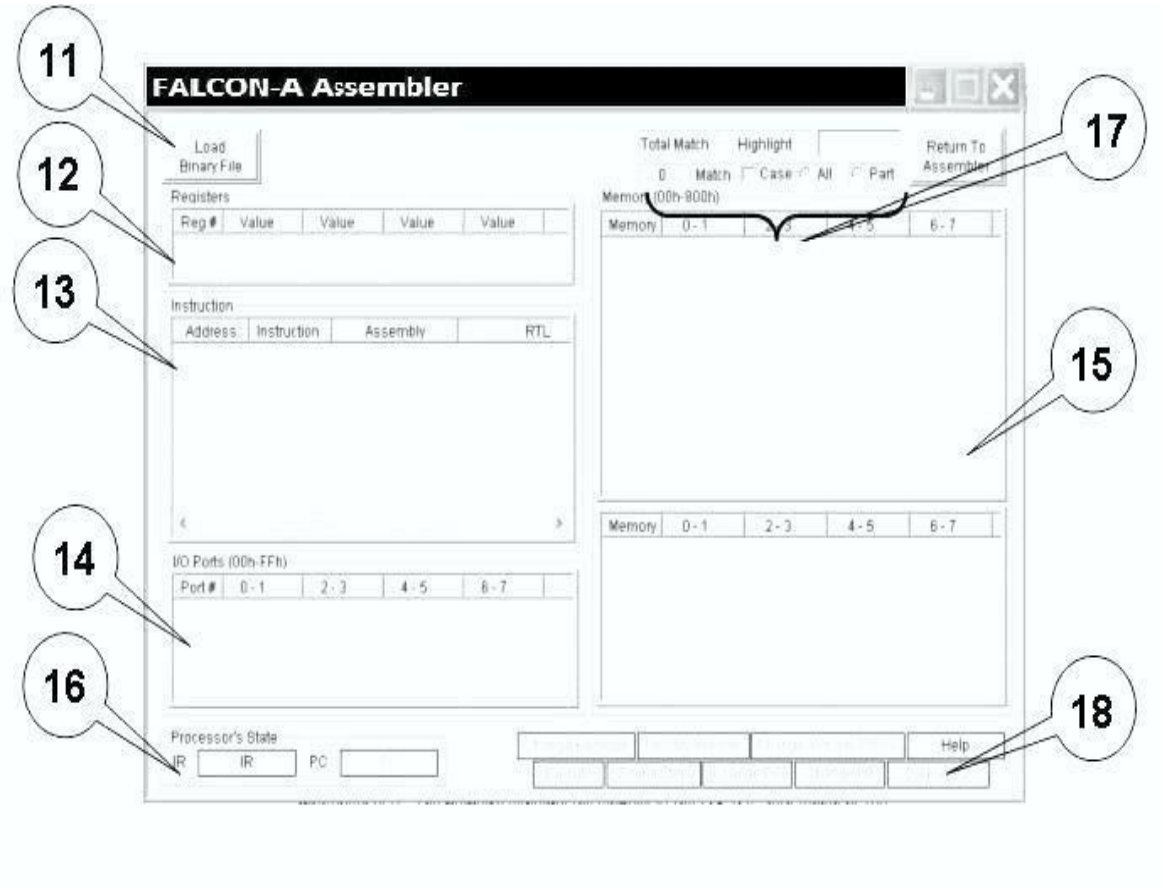

11
12
13
14
15
16
17
18
FALCON-A Assembler
Load Binary File
Total Match
Highlight
0 Match
Case
All
Part
Return To Assembler
Registers
Reg # Value Value Value Value
Memory (00h-800h)
Memory 0 - 1 2 - 3 4 - 5 6 - 7
Instruction
Address Instruction Assembly RTL
Memory 0 - 1 2 - 3 4 - 5 6 - 7
I/O Ports (00h-FFh)
Port # 0 - 1 2 - 3 4 - 5 6 - 7
Processor's State
IR
IR
PC
Help

Figure 6

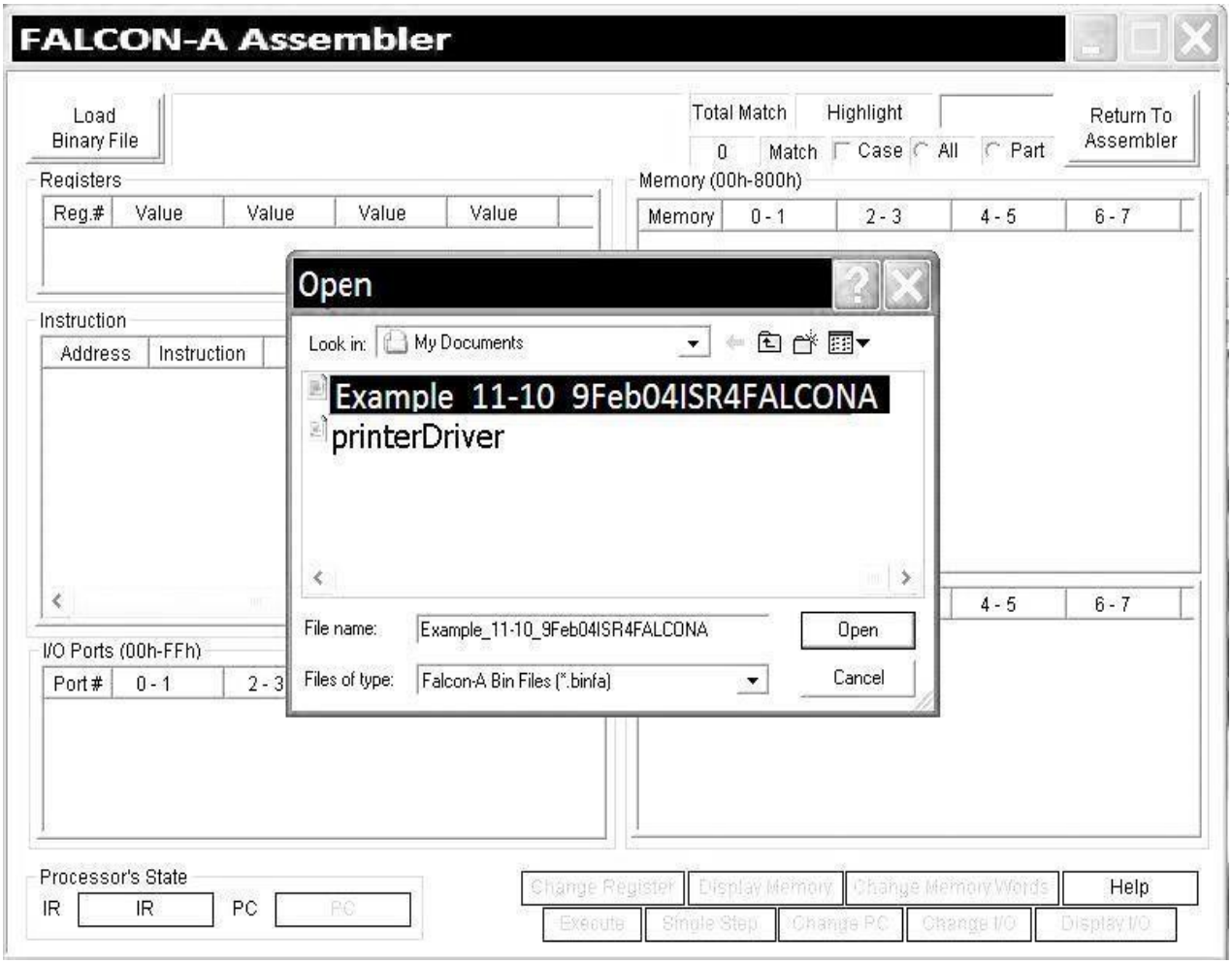
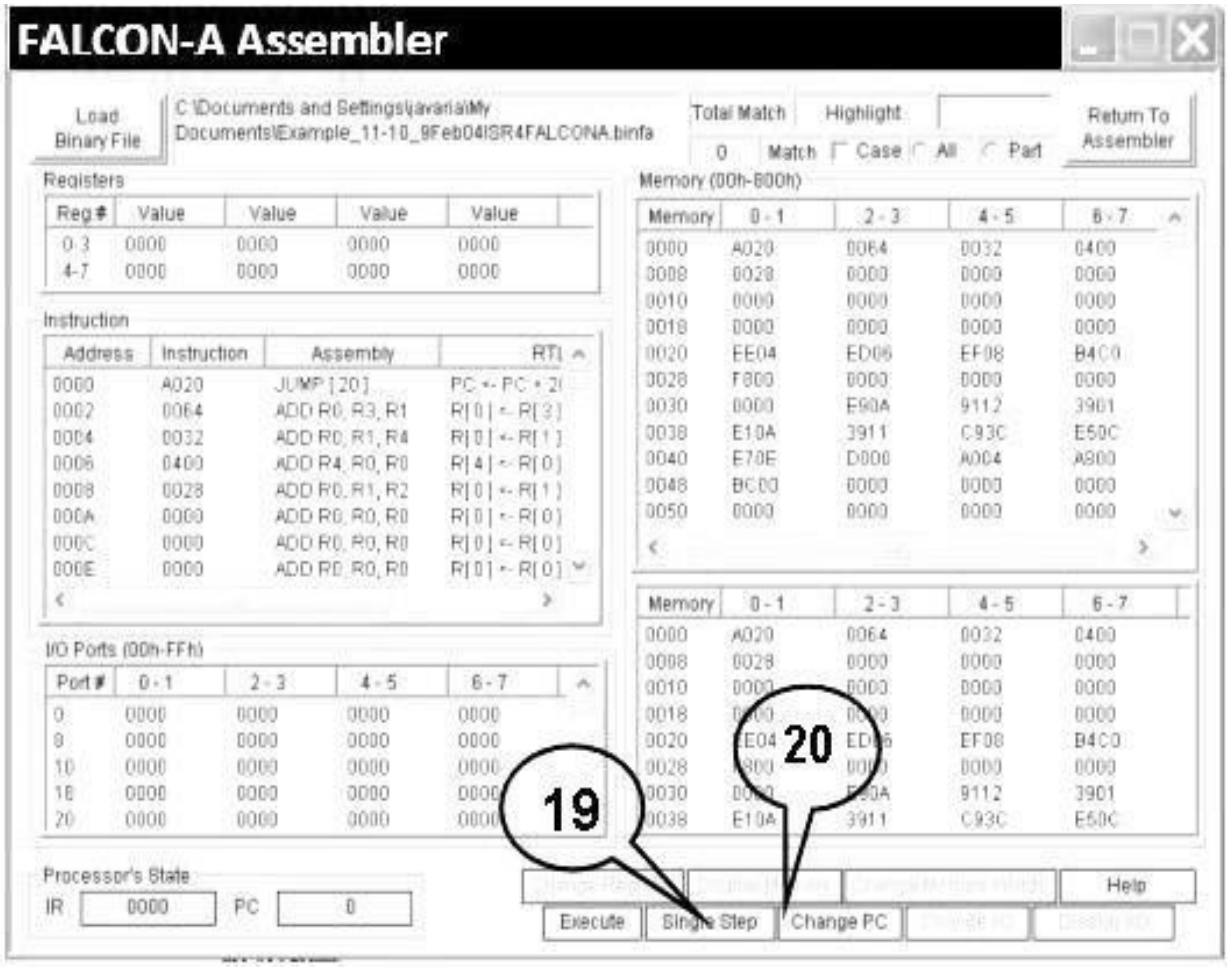

FALCON-A Assembler
Load Binary File
Total Match
Highlight
0
Match
Case
All
Part
Return To Assembler
Registers
Reg.#
Value
Value
Value
Value
Memory (00h-800h)
Memory
0 - 1
2 - 3
4 - 5
6 - 7
Instruction
Address
Instruction
Open
Look in:
My Documents
Example_11-10_9Feb04ISR4FALCONA
printerDriver
File name:
Example_11-10_9Feb04ISR4FALCONA
Open
Files of type:
Falcon-A Bin Files (*.binfa)
Cancel
I/O Ports (00h-FFh)
Port #
0 - 1
2 - 3
Processor's State
IR
PC
Change Register
Display Memory
Change Memory Words
Help
Execute
Single Step
Change PC
Change I/O
Display I/O

Figure 7




**4. FALCON-A assembly language programming techniques:**

- If a signed value, x, cannot fit in 5 bits (i.e., it is outside the range -16 to +15), FALSIM will report an error with a **load r1, [x]** or a **store r1, [x]** instruction. To overcome this problem, use **movi r2, x** followed by **load r1, [r2]**.

- If a signed value, x, cannot fit in 8 bits (i.e., it is outside the range -128 to +127), even the previous scheme will not work. FALSIM will report an error with the **movi r2, x** instruction. The following instruction sequence should be used to overcome this limitation of the FALCON-A. First store the 16-bit address in the memory using the **.sw** directive. Then use two load instructions as shown below:

```
a:    .sw    x
      load r2, [a]
      load r1, [r2]
```

  This is essentially a "memory-register-indirect" addressing. It has been made possible by the **.sw** directive. The value of **a** should be less than 15.
- A similar technique can be used with immediate ALU instructions for large values of the immediate data, and with the transfer of control (call and jump) instructions for large values of the target address.
- Large values (16-bit values) can also be stored in registers using the **mul** instruction combined with the **addi** instruction. The following instructions bring a 201 in register r1.

```
movi r2, 10
movi r3, 20
mul r1, r2, r3        ; r1 contains 200 after this instruction
addi r1, r1, 1        ; r1 now contains 201
```

- Moving from one register to another can be done by using the instruction **addi r2, r1, 0.**
- Bit setting and clearing can be done using the logical (and, or, not, etc) instructions.
- Using shift instructions (shiftl, asr, etc.) is faster that **mul** and **div,** if the multiplier or divisor is a power of 2.

# Lecture No. 30

## Interrupt Priority and Nested Interrupts

**Reading Material**

Vincent P. Heuring & Harry F. Jordan — Chapter 8
Computer Systems Design and Architecture — 8.3.3, 8.4

### *Summary*

- Nested Interrupts
- Interrupt Mask
- DMA

### Nested Interrupts

(Read from Book, Jordan Page 397)

### Interrupt Mask

(Read from Book, Jordan Page 397)

### Priority Mask

(Read from Book, Jordan Page 398)

### Examples

**Example # 1**[20]

Assume that three I/O devices are connected to a 32-bit, 10 MIPS CPU. The first device is a hard drive with a maximum transfer rate of 1MB/sec. It has a 32-bit bus. The second device is a floppy drive with a transfer rate of 25KB/sec over a 16-bit bus, and the third device is a keyboard that must be polled thirty times per second. Assuming that the polling operation requires 20 instructions for each I/O device, determine the percentage of CPU time required to poll each device.

[20] Adopted from [H&P org]

**Solution:**
The hard drive can transfer 1MB/sec or 250 K 32-bit words every second. Thus, this hard drive should be polled using at least this rate.

Using $1K=2^{10,}$ the number of CPU instructions required would be

$250 \times 2^{10} \times 20 = 5120000$ instructions per second.

Percentage of CPU time required for polling is

$(5.12 \times 10^6)/ (10 \times 10^6) = 51.2\%$

The floppy disk can transfer $25K/2= 12.5 \times 2^{10}$ half-words per second. It should be polled with at least this rate. The number of CPU instructions required will be $12.5 \times 2^{10} \times 20 = 256{,}000$ instructions per second.

Therefore, the percentage of CPU time required for polling is

$(0.256 \times 10^6)/ (10 \times 10^6) = 2.56\%$.

For the keyboard, the number of instructions required for polling is

$30 \times 20 = 600$ instructions per second.

Therefore, the percentage of CPU time spent in polling is

$600 / (10 \times 10^6) = 0.006\%$

It is clear from this example that while it is acceptable to use polling for a keyboard or a floppy drive, it is very risky to use polling for the hard drive. In general, for devices with a high data rate, the use of polling is not adequate.

**Example # 2**[2]

a. What should be the polling frequency for an I/O device if the average delay between the time when the device wants to make a request and the time when it is polled, is to be at most 10 ms?
b. If it takes 10,000 cycles to poll the I/O device, and the processor operates at 100MHz, what % of the CPU time is spent polling?
c. What if th[21]e system wants to provide an average delay of 1msec?

**Solution:**

a. Assuming that the I/O requests are distributed evenly in time, the average time that a device will have to wait for the processor to poll is half the time between

[21] Adopted from [Schaum]

polling attempts. Therefore, to provide an average delay of 10 ms, the processor will have to poll every 20 ms, or 50 times per second.

b. If each polling attempt takes 10,000 cycles, then the processor will spend 500,000 cycles polling each second. The % of CPU time spent in polling is then $(0.5x10^6)/(100x10^6)=0.5\%$
c. To provide an average delay of 1ms, the polling frequency must be increased. The processor will have to poll every 2ms, or 500 times per second. This will consume 5,000,000 cycles for polling. The % of CPU time spent polling then becomes 5/100=5%.

**Example # 3**[22]

What percentage of time will a 20MIPS processor spend in the busy wait loop of an 80-character line printer when it takes 1 msec to print a character and a total of 565 instructions need to be executed to print an 80 character line. Assume that two instructions are executed in the polling loop.

**Solution:**

Out of the total 565 instructions executed to print a line, 80x2=160 are required for polling. For a 20MIPS processor, the execution of the remaining 405 instructions takes $405/ (20x10^6) = 20.25\mu sec$. Since the printing of 80 characters takes 80ms, (80-0.02025) =79.97msec is spent in the polling loop before the next 80 characters can be printed. This is 79.97/80=99.96% of the total time.

**Example # 4**[23]

Consider a 20 MIPS processor with several input devices attached to it, each running at 1000 characters per second. Assume that it takes 17 instructions to handle an interrupt. If the hardware interrupt response takes 1μsec, what is the maximum number of devices that can be handled simultaneously?

**Solution:**

A service for one character requires $17/ (20x10^6) +1\mu sec=1.85\mu sec$. Since each device runs at 1000 characters per second, 1.85 ms of handling time is required by each device every second. Therefore the maximum number of devices that can be handled is $1/ (1.85x10^{-3}) = 540$.

**Example # 5**[24]

Assume that a floppy drive having a transfer rate of 25KB per second is attached to a 32 bit, 10MIPS CPU using an interrupt driven interface. The drive has a 16-bit data bus.

[22] Adopted from [H&J]
[23] Adopted from [H&J]
[24] Adopted from [H&P org]

Assume that the interrupt overhead is 20 instructions. Calculate the fraction of CPU time required to service this drive when it is active.

**Solution:**
Since the floppy drive has a 16-bit data bus, it can transfer two bytes at one time. Thus its transfer rate is 25/2 = 12.5K half-words (16-bits each) per second. This corresponds to an overhead of 20 instructions or 12.5K x 20 = 12.5 x $2^{10}$ x 20 = 256000 instructions per second.

**Example # 6[25]**
A processor with a 500 MHz clock requires 1000 clock cycles to perform a context switch and start an ISR. Assume each interrupt takes 10,000 cycles to execute the ISR and the device makes 200 interrupt requests per second. Also, assume that the processor polls every 0.5msec during the time when there are no interrupts. Further assume that polling an I/O device requires 500 cycles. Compute the following:

a. How many cycles per second does the processor spend handling I/O from the device if only interrupts are used?
b. What fraction of the CPU time is used in interrupt handling for part (a)?
c. How many cycles per second are spent on I/O if polling is also used with interrupts?
d. How often should the processor poll so that polling incurs the same overhead as interrupts?

**Solution:**

a. The device makes 200 interrupt requests per second, each of which takes
   10,000 + 2x1000 (context switching to the ISR and back from it)
   = 12,000 cycles.

   Thus, a total of 200x12,000=2,400,000 cycles per second are spent handling I/O using interrupts.

b. The percentage of the processor time used in interrupt handling is 2,400,000/(500x$10^6$) or 0.48%.

c. There are 200 interrupt requests per second, or one interrupt request every 5 ms. Every interrupt consumes a total of 12,000 cycles, as calculated in part (a). For a 500 MHz CPU, this is

   12000/(500 x $10^6$ ) = 24 microseconds.

   For 200 interrupts per second, this is 4.8 msec.

[25] Adopted from [Schaum]

This leaves 1000 - 4.8 = 995.2 msec for polling.

Since the processor polls once every 0.5 msec during the time when there is no interrupt, this corresponds to

995/0.5 = 1990 times per second.

The total number of cycles required for polling is

1990 x 500 = 995,000 cycles per second.

Thus, the total time spent on I/O when using polling with interrupts is

2,400,000 + 995,000 = 3,395,000 cycles per second.

d. The interrupt overhead is 1000 cycles per second for a context switch to the ISR and 1000 cycles per second back from it. This is a total of 2 x 1000 cycles per second. With 200 interrupts per second, this is
200 x 2000 = 400,000 cycles per second.

The polling overhead is 500 cycles per second. Thus, for the same overhead as interrupts, the polling operation should be performed
400,000 / 500 = 800 times per second,
or 1/800 = every 1.25 msec.

## Direct Memory Access (DMA)

Direct memory access is a technique, where by the CPU passes its control to the memory subsystem or one of its peripherals, so that a contiguous block of data could be transferred from peripheral device to memory subsystem or from memory subsystem to peripheral device or from one peripheral device to another peripheral device.

**Advantage of DMA**

The transfer rate is pretty fast and conceptually you could imagine that through disabling the tri-state buffers, the system bus is isolated and a direct connection is established between the I/O subsystem and the memory subsystem and then the CPU is free. It is idle at that time or it could do some other activity. Therefore, the DMA would be quite useful, if a large amount of data needs to be transferred, for example from a hard disk to a printer or we could fill up the buffer of a printer in a pretty short time.
As compared to interrupt driven I/O or the programmed I/O, DMA would be much faster. What is the consequence? The consequence is that we need to have another chip, which is a **DMA controller.** "
total activity and synchronize the transfer of data". DMA could be considered as a technique of transferring data from I/O to memory and from memory to I/O without the intervention of the CPU. The CPU just sets up an I/O module or a memory

subsystem, so that it passes control and the data could be passed on from I/O to memory or from memory to I/O or within the memory from one subsystem to another subsystem without interaction of the CPU. After this data transfer is complete, the control is passed from I/O back to the CPU.
Now we can illustrate further the advantage of DMA using following example.

**Example of DMA**
If we write instruction load as follows:

```
load [2], [9]
```

This instruction is illegal and not available in the SRC processor. The symbols [2] and [9] represent memory locations. If we want to have this transfer to be done then two steps would be required. The instruction would be:

```
load  r1, [9]
store r1, [2]
```

Thus it is not possible to transfer from one memory location to another without involving the CPU. The same applies to transfer between memory and peripherals connected to I/O ports. For example we cannot have:

```
out [6], datap
```

It has to be done again in two steps:

```
load r1, [6]
out  r1, datap
```

Similar comments apply to the "in" instruction. Thus the real cause of the limited transfer rate is the CPU itself. It acts as an unnecessary middle man. The example illustrates that in general, every data word travels over the system bus twice and this is not necessary, and therefore, the DMA in such cases is pretty useful.

**DMA Approach**
The DMA approach is to turn off i.e. through tri-state buffers and therefore, electrically disconnect from the system bus, the CPU and let a peripheral device or a memory subsystem or any other module or another block of the same module communicate directly with the memory or with another peripheral device. This would have the advantage of having higher transfer rates which could approach that of limited by the memory itself.

**Disadvantage of DMA**
The disadvantage however, would be that an additional DMA controller would be required, that could make the system a bit more complex and expensive. Generally, the DMA requests have priority over all other bus activities including interrupts. No interrupts may be recognized during a DMA cycle.

# Lecture No. 31

## Direct Memory Access (DMA)

**Reading Material**

Vincent P. Heuring & Harry F. Jordan — Chapter 8
Computer Systems Design and Architecture — 8.4

***Summary***

- Direct Memory Access (DMA):

## Direct Memory Access (DMA):

**Introduction**

Direct Memory Access is a technique which allows a peripheral to read from and/or write to memory without intervention by the CPU. It is a simple form of bus mastering where the I/O device is set up by the CPU to transfer one or more contiguous blocks of memory. After the transfer is complete, the I/O device gives control back to the CPU.
The following DMA transfer combinations are possible:

- Memory to memory
- Memory to peripheral
- Peripheral to memory
- Peripheral to peripheral

The DMA approach is to "turn off" ( .e., tri-state and electrically disconnect from the system buses) the CPU and let a peripheral device (or memory - another module or another block of the same module) co municate direct y with the mem ry (or another peripheral).
ADVANTAGE: Higher transfer rates (approaching that of the memory) can be achieved.
DISADVANTAGE: A DMA Contro ler, or a DMAC, is needed, making the system complex and expensive
Generally, DMA requests have priority over all other bus activities, including interrupts. No interrupts may be recognized during a DMA cycle.

**Reason for DMA:**

The instruction **load [2], [9]** is illegal. The symbols [2] and [9] represent memory locations. This transfer has to be done in two steps:

- **load r1,[9]**
- **store r1,bx**

Thus, it is not possible to transfer from one memory location to another without involving the CPU. The same applies to transfer between memory and peripherals connected to I/O ports. e.g., we cannot have **out [6], datap.** It has to be done in two steps:

- **load r1,[6]**
- **out r1, datap**

Similar comments apply to the **in** instruction.

***Thus, the real cause of the limited transfer rate is the CPU itself. It acts as an unnecessary "middleman". The above discussion also implies that, in general, every data word travels over the system bus twice.***

## Some Definitions:

- **MASTER COMPONENT:** A component connected to the system bus and having control of it during a particular bus cycle.
- **SLAVE COMPONENT**: A component connected to the system bus and with which the master component can communicate during a particular bus cycle. Normally the CPU with its bus control logic is the master component.
- **QUALIFICATIONS TO BECOME A MASTER:** A Master must have the capability to place addresses on the address bus and direct the bus activity during a bus cycle.
- **QUALIFIED COMPONENTS:**
  - Processors with their associated bus control logic.
  - DMA controllers
- **CYCLE STEALING**: Taking control of the system bus for a few bus cycles.

**Data Transfer using DMA:**

Data transfer using DMA takes place in three steps.

**1st Step:**

in this step when the processor has to transfer data it issues a command to the DMA controller with the following information:

- Operation to be performed i.e., read or write operation.
- Address of I/O device.
- Address of memory block.
- Size of data to be transferred.

After this, the processor becomes free and it may be able to perform other tasks.

**2nd Step:**

In this step the entire block of data is transferred directly to or from memory by the DMA controller.

**3rd Step:**

In this, at the end of the transfer, the DMA controller informs the processor by sending an

interrupt signal.

See figure 8.18 on the page number 400 of text book.

**The DMA Transfer Protocol:**

Most processors have a separate line over which an external device can send a request for DMA. There are various names in use for such a line. HOLD, RQ, or Bus Request (BR), etc. are examples of these names.

The DMA cycle usually begins with the alternate bus master requesting the system bus by activating the associated Bus Request line and, of course, satisfying the setup and hold times. The CPU completes the current bus cycle, in the same way as it does in case of interrupts, and responds by floating the address, data and control lines. A Bus Grant pulse is then output by the CPU to the same device from where the request occurred. After receiving the Bus Grant pulse, and waiting for the "float delay" of the CPU, the requesting device may drive the system bus. This precaution prevents bus contention. To return control of the bus to the CPU, the alternate bus master relinquishes bus control and issues a release pulse on the same Bus Request line. The CPU may drive the system bus after detecting the release pulse. The alternate bus master should be tri-stated off the local bus and have other CPU interface circuits re-enabled within this time.

**DMA has priority over Interrupt driven I/O:**

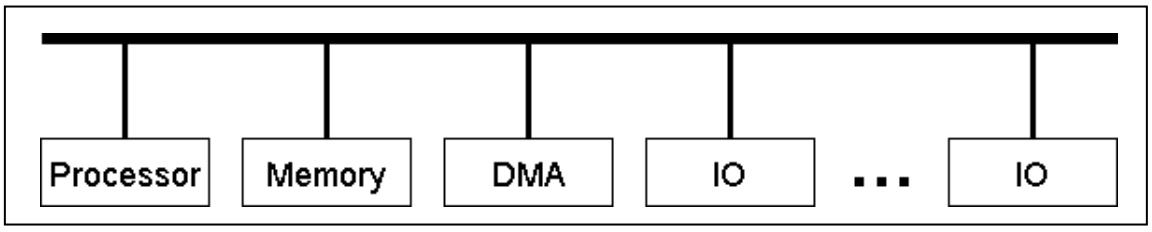
In interrupt driven I/O the I/O transfer depends upon the speed at which the processor tests and service a device. Also, many instructions are required for each I/O transfer. These factors become bottleneck when large blocks of data are to be transferred. While in the DMA technique the I/O transfers take place without the intervention by the CPU, rather CPU pauses for one bus cycle. So DMA technique is the more efficient technique for I/O transfers.

**DMA Configurations:**

- Single Bus Detached DMA
- Single Bus Integrated DMA
- I/O Bus

**Single Bus Detached DMA**

In the example provided by the above diagram, there is a single bidirectional bus connecting the processor, the memory, the DMA module and all the I/O modules.

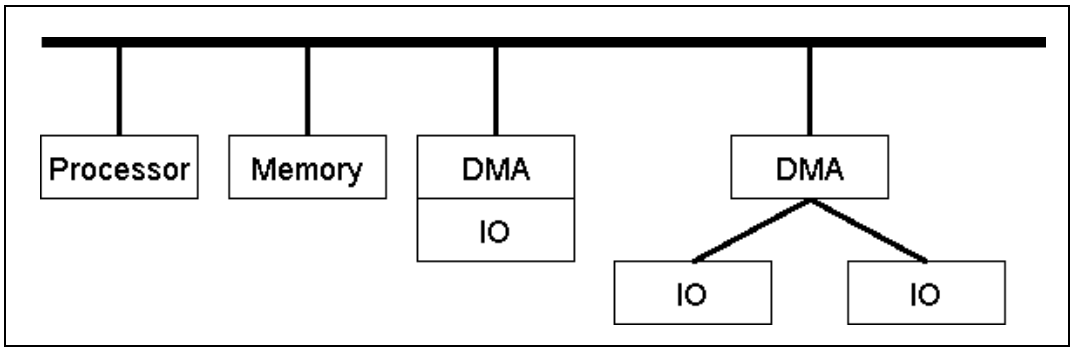



When a particular I/O module needs to read or write large amounts contiguous data it requests the processor for direct memory access. If permission is granted by the processor, the I/O module sends the read or write address and the size of data needed to

be read or written to the DMA module. Once the DMA module acknowledges the request, the I/O module is free to read or write its contiguous block of data from or onto main memory. Even though in this situation the processor will not be able to execute while the transfer is going on (as there is a just a single bus to facilitate transfer of data), DMA transfer is much faster then having each word of memory being read by the processor and then being written to its location.

**Single Bus Integrated DMA**

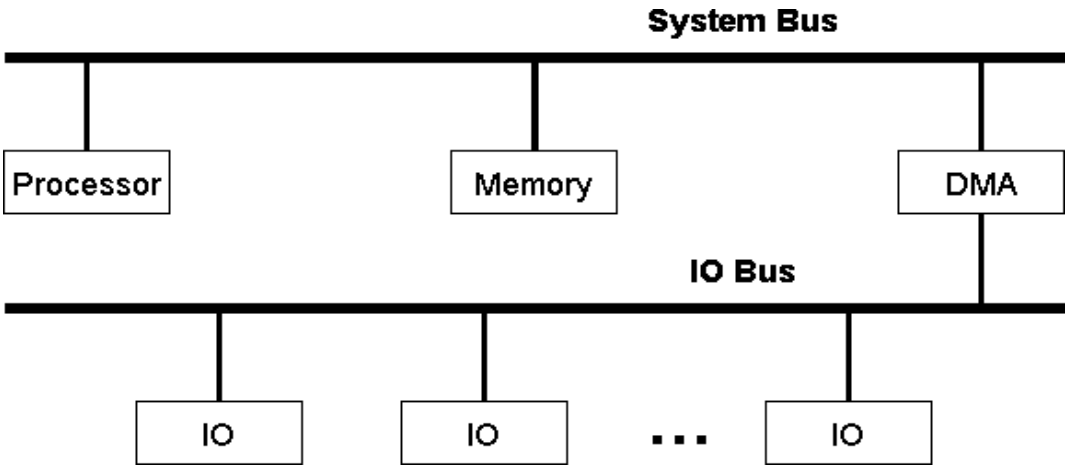
In this configuration the DMA and one or more I/O modules are integrated without the




inclusion of system bus functioning as the part of I/O module or may be as a separate module controlling the I/O module.

**IO Bus**

In this configuration we integrate the DMA and I/O modules through an I/O bus.




So it will cut the number of I/O interfaces required between DMA and I/O module.

Example

An I/O device transfers data at a rate of 10MB/s over a 100MB/s bus. The data is transferred in 4KB blocks. If the processor operates at 500MHz, and it takes a total of

5000 cycles to handle each DMA request, find the fraction of CPU time handling the data transfer with and without DMA.

Solution
Without DMA

The processor here copies the data into memory as it is sent over the bus. Since the I/O device sends data at a rate of 10MB/s over the 100MB/s bus, 10 % of each second is spent transferring data. Thus 10% of the CPU time is spent copying data to memory.

With DMA

Time required in handling each DMA request is 5000 cycles. Since 2500 DMA requests are issued (10MB/4KB) the total time taken is 12,500,000 cycles. As the CPU clock is 500MHZ, the fraction of CPU time spent is 12,500,000/(500x10$^6$) or 2.5%.

Example
A hard drive with a maximum transfer rate of 1Mbyte/sec is connected to a 32-bit, 10MIPS CPU operating at a clock frequency of 100 MHz. Assume that the I/O interface is DMA based and it takes 500 clock cycles for the CPU to set-up the DMA controller. Also assume that the interrupt handling process at the end of the DMA transfer takes an additional 300 CPU clock cycles. If the data transfer is done using 2 KB blocks, calculate the percentage of the CPU time consumed in handling the hard drive.

Solution
Since the hard drive transfers at 1MB/sec, and each block size is 2KB, there are

1000/2= 500 blocks transferred/sec

Every DMA transfer uses 500+300=800 CPU cycles. This gives us

800x500 = 400,000 = 400x10$^3$ cycles/sec

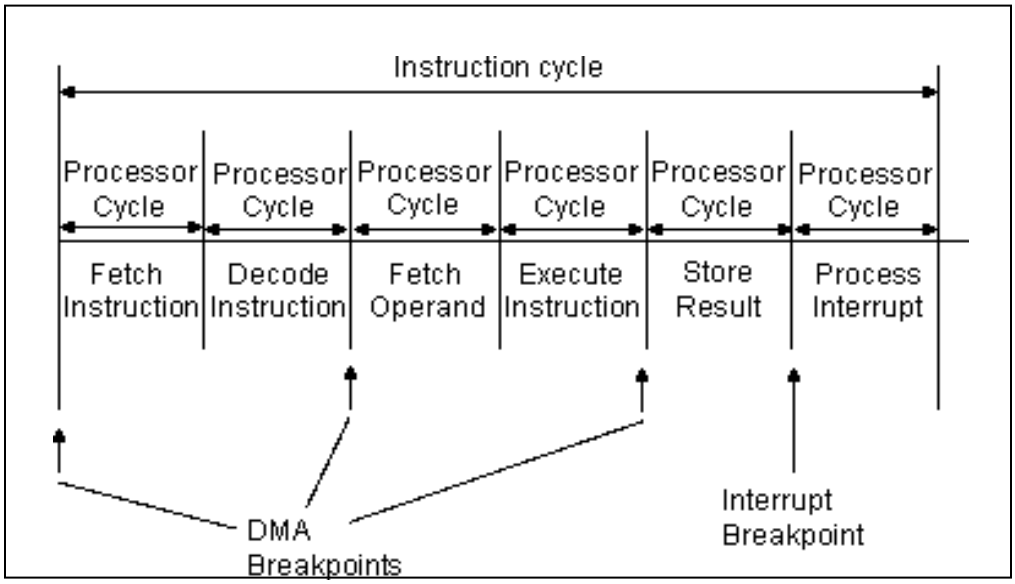
For the 100 MHz CPU, this corresponds to

(400x10$^3$) / (100x10$^6$)= 4x10$^{-3}$ = 0.4%

This would be the case when the hard drive is transferring data all the time. In actual situation, the drive will not be active all the time, and this number will be much smaller than 0.4%.
Another assumption that is implied in the previous example is that the DMA controller is the only device accessing the memory. If the CPU also tries to access memory, then either the DMAC or the CPU will have to wait while the other one is actively accessing the memory. If cache memory is also used, this can free up main memory for use by the DMAC.

**Cycle Stealing**
The DMA module takes control of the bus to transfer data to and from memory by

forcing the CPU to temporarily suspend its operation. This approach is called Cycle Stealing because in this approach DMA steals a bus cycle.

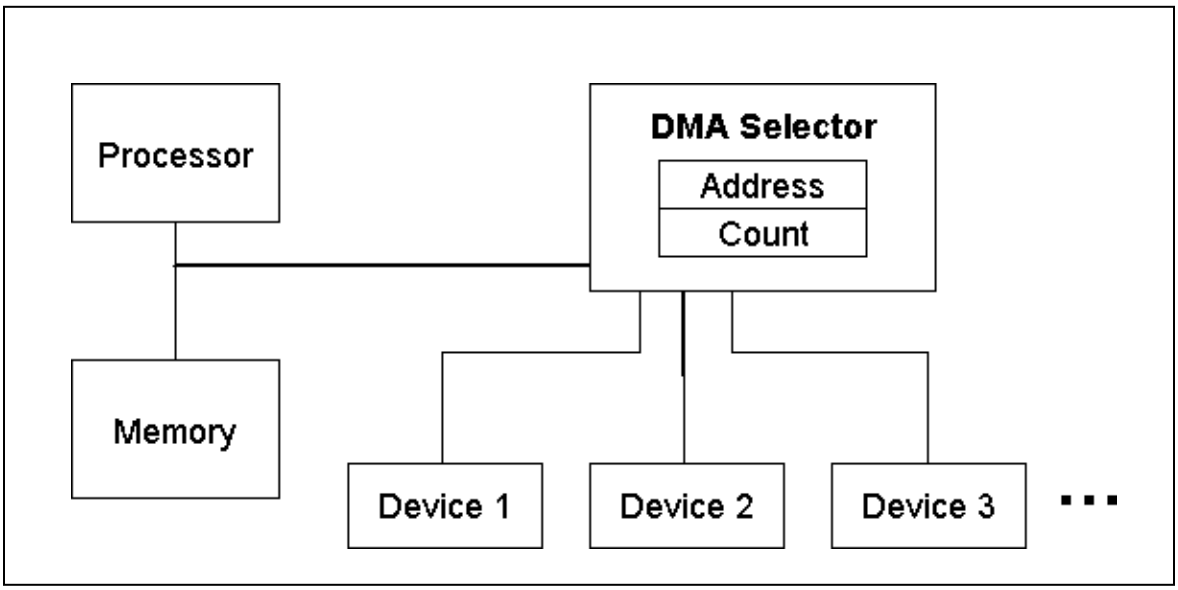



**DMA and Interrupt breakpoints during an instruction cycle**
The figure shows that the CPU suspends or pauses for one bus cycle when it needs a bus cycle, transfers the data and then returns the control back to the CPU.

**I/O processors**
When I/O module has its own local memory to control a large number of I/O devices without the involvement of CPU is called I/O processor.

**I/O Channels**
When an I/O module has a capability of executing a specific set of instructions for specific I/O devices in the memory without the involvement of CPU is called I/O channel.

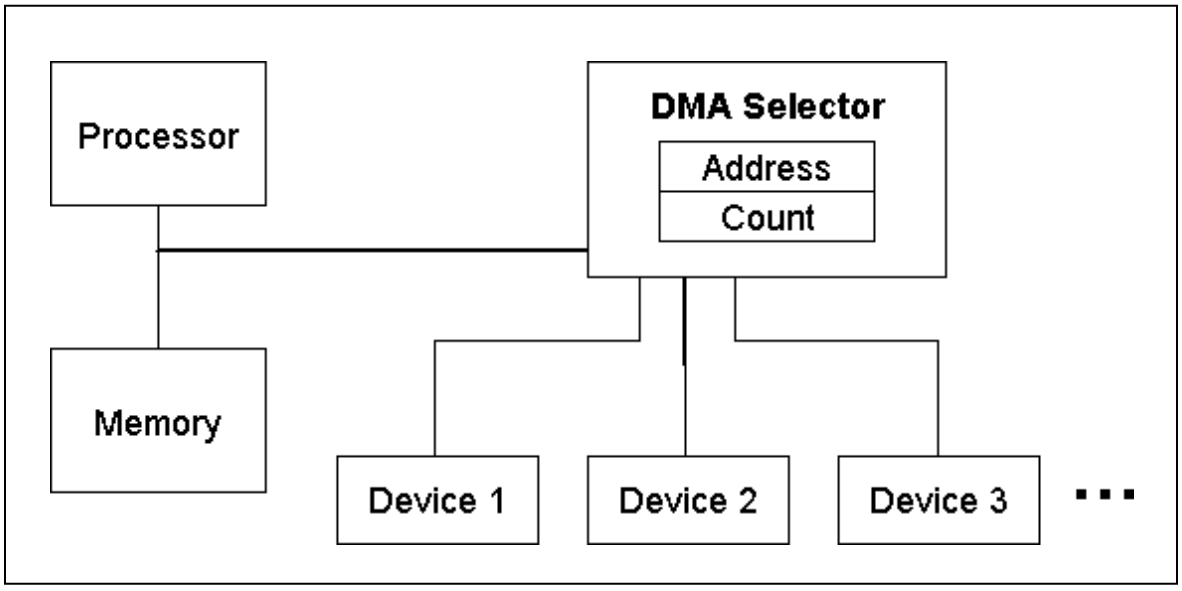
**I/O channel architecture:**




**Types of I/O channels:**

**Selector Channel**
It is the DMA controller that can do block transfers for several devices but only one at a time.

**Multiplexer Channel**




It is the DMA controller that can do block transfers for several devices at once.

**Types of Multiplexer Channel**

- Byte Multiplexer
- Block Multiplexer

**Byte Multiplexer**

- Byte multiplexer accepts or transmits characters.
- Interleaves bytes from several devices.
- Used for low speed devices.

**Block Multiplexer**

- Block multiplexer accepts or transmits block of characters.
- Interleaves blocks of bytes from several devices.
- Used for high speed devices.

**Virtual Address:**

Virtual address is generated be the logical by the memory management unit for translation.

**Physical Address:**

Physical address is the address in the memory.

**DMA and memory system**

DMA disturbs the relationship between the memory system and CPU.

**Direct memory access and the memory system**

Without DMA, all memory accesses are handled by the CPU, using address translation and cache mechanism. When DMA is implemented into an I/O system memory accesses can be made without intervening the CPU for address translation and cache access. The problems created by the DMA in virtual memory and cache systems can be solved using hardware and software techniques.

**Hardware Software Interface**

One solution to the problem is that all the I/O transfers are made through the cache to ensure that modified data are read and updated in the cache on the I/O write. This method can decrease the processor performance because of infrequent usage of the I/O data.

Another approach is that the cache is invalidated for an I/O read and for an I/O write, write-back (flushing) is forced by the operating system. This method is more efficient because flushing of large parts of cache data is only done on DMA block accesses.

Third technique is to flush the cache entries using a hardware mechanism, used in multiprogramming system to keep cache coherent.

**SOME clarifications:**

- The terms "serial" and "parallel" are with respect to the computer I/O ports --- not with respect to the CPU. The CPU always transfers data in parallel.

- The terms "programmed I/O", "interrupt driven I/O" and "DMA" are with respect to the CPU. Each of these terms refers to a way in which the CPU handles I/O, or the way data flow through the ports is controlled.
- The terms "simplex" and "duplex" are with respect to the transmission medium or the communication link.
- The terms "memory mapped I/O" and "independent I/O" are with respect to the mapping of the interface, i.e., they refer to the CPU control lines used in the interface.

# Lecture No. 32

## Magnetic Disk Drives

Reading Material

Vincent P. Heuring & Harry F. Jordan — Chapter 9
Computer Systems Design and Architecture — 9.1

*Summary*

- **Hard Disk**
- Static and Dynamic Properties
- Examples
- Mechanical Delays and Flash Memory
- Semiconductor Memory vs. Hard Disk

### Hard Disk

Peripheral devices connect the outside world with the central processing unit through the I/O modules. One important feature of these peripheral devices is the variable data rate. Peripheral devices are important because of the function they perform.
A hard disk is the most frequently used peripheral device. It consists of a set of platters. Each platter is divided into tracks. The track is subdivided into sectors. To identify each sector, we need to have an address. So, before the actual data, there is a header and this header consisting of few bytes like 10 bytes. Along with header there is a trailer. Every sector has three parts: a header, data section and a trailer.

### Static Properties

The storage capacity can be determined from the number of platters and the number of tracks. In order to keep the density same for the entire surface, the trend is to use more number of sectors for outer tracks and lesser number of sectors for inner tracks.

### Dynamic Properties

When it is required to read data from a particular location of the disk, the head moves towards the selected track and this process is called seek. The disk is constantly rotating at a fixed speed. After a short time, the selected sector moved under the head. This interval is called the rotational delay. On the average, the data may be available after half

a revolution. Therefore, the rotational latency is half revolution.
The time required to seek a particular track is defined by the manufacturer. Maximum, minimum and average seek times are specified. Seek time depends upon the present position of the head and the position of the required sector. For the sake of calculations, we will use the average value of the seek time.

- **Transfer rate**

When a particular sector is found, the data is transferred to an I/O module. This would depend on the transfer rate. It would typically be between 30 and 60 Mbytes/sec defined by the manufacturer.

- **Overhead time**

Up till now, we have assumed that when a request is made by the CPU to read data, then hard disk is available. But this may not be the case. In such situation we have to face a queuing delay. There is also another important factor: the hard disk controller, which is the electronics present in the form of a printed circuit board on the hard disk. So the time taken by this controller is called over head time
The following examples will clarify some of these concepts.

**Example 1**
Find the average rotational latency if the disk rotates at 20,000 rpm.

**Solution**
The average latency to the desired data is halfway round the disk so
Average rotational latency =0.5/(20,000/60)
=1.5ms

**Example 2**
A magnetic disk has an average seek time of 5 ms. The transfer rate is 50 MB/sec. The disk rotates at 10,000 rpm and the controller overhead is 0.2 msec. Find the average time to read or write 1024 bytes.

**Solution**
Average Tseek=5ms
Average Trot 0.5*60/10,000=3 ms
Ttransfer=1KB/50MB=0.02ms
Tcontroller=0.2ms
The total time taken= Tseek +Trot+ Ttsfr +Tctr
=5+3+0.02+0.2
=8.22 ms

**Example 3**
A hard disk with 5 platters has 1024 tracks per platter,512 sectors per track and 512 bytes/sector. What is the total capacity of the

disk?

**Solution**
512 bytes x 512
sectors=0.2MB/track
0.2 MB x 1024 tracks=0.2GB/platter
Therefore the hard disk has the total capacity of 5 x 0.2=1GB

**Example 4**
How many platters are required for a 40GB disk if there are 1024 bytes/sector, 2048 sectors per track and 4096 tracks per platter

**Solution**
The capacity of one platter
= 1024 x 2048 x 4096
= 8GB
For a 40GB hard disk, we need 40/8
= 5 such platters.

**Example 5**
Consider a hard disk that rotates at 3000 rpm. The seek time to move the head between adjacent tracks is 1 ms. There are 64sectors per track stored in linear order.
Assume that the read/write head is initially at the start of sector 1 on track 7.

a. How long will it take to transfer sector 1 on track 7 to sector 1 on track 9?
b. How long will it take to transfer all the sectors on track 12 to corresponding sectors on track 13?

**Solution**
Time for one revolution=60/3000=20ms
a. Total transfer time=sector read time+head movement time+rotational delay+sector write time

Time to read or write on sector=20/64=0.31ms/sector

Head movement time from track 7 to track 9=1msx2=2ms

After reading sector 1 on track 7, which takes .31ms, an additional 19.7 ms of rotational delay is needed for the head to line up with sector 1 again.
The head movement time of 2 ms gets included in the19.7 ms. Total transfer time=0.31ms+19.7ms+0.31ms=20.3ms

b. The time to transfer all the sectors of track 12 to track 13 can be computed in the similar way. Assume that the memory buffer can hold an entire track. So the time to read or write an entire track is simply the rotational delay for a track, which is 20 ms. The head movement time is 1ms, which is also the time for 1/0.3=3.3≈ 4 sectors to pass under the head. Thus after reading a track and repositioning the head, it is now on track 13, at four sectors past the initial sector that was read on track 12. (Assuming track 13 is written starting at sector 5)
therefore total transfer time= 20+1+20=41ms.
If writing of track 13 start at the first sector, an additional 19 ms should be added, giving a total transfer time= 60 ms

**Example 6**

Calculate time to read 64 KB (128 sectors) for the following disk parameters.
–180 GB, 3.5 inch disk
–12 platters, 24 surfaces
–7,200 RPM; (4 ms avg. latency)
–6 ms avg. seek (r/w)
–64 to 35 MB/s (internal)
–0.1 ms controller time

**Solution**

Disk latency = average seek time + average rotational delay + transfer time + controller overhead

= 6 ms + 0.5 x 1/(7200 RPM) /(60000ms/M)) + 64 KB / (64 MB/s) + 0.1 ms
= 6 + 4.2 + 1.0 + 0.1 ms = 11.3 ms

**Mechanical Delay and Flash Memory**

Mechanical movement is involved in data transfer and causes mechanical delays which are not desirable in embedded systems. To overcome this problem in embedded systems, flash memory is used. Flash memory can be thought of a type of electrically erasable PROM. Each cell consists of two MOSFET and in between these two transistors, we have a control gate and the presence/absence of charge tells us that it is a zero or one in that location of memory.
The basic idea is to reduce the control overheads, and for a FLASH chip, this control overhead is low. Furthermore flash memory has low power dissipation. For embedded devices, flash is a better choice as compared to hard disk. Another important feature is that read time is small for flash However the write time may be significant. The reason is

that we first have to erase the memory and then write it. However in embedded system, number of write operations is less so flash is still a good choice.

**Example 7**
Calculate the time to read 64 KB for the previous disk, this time using 1/3 of quoted seek time, 3/4 of internal outer track bandwidth

**Solution**

Disk latency = average seek time + average rotational delay + transfer time + controller overhead
= (0.33* 6 ms) + 0.5 * 1/(7200 RPM)
+ 64 KB / (0.75* 64 MB/s) + 0.1 ms
= 2 ms + 0.5 /(7200 RPM/(60000ms/M))
+ 64 KB / (48 KB/ms) + 0.1 ms
= 2 + 4.2 + 1.3+ 0.1 ms = 7.6 ms

**Semiconductor Memory vs. Hard Disk**
At one time developers thought that development of semiconductor memory would completely wipe out the hard disk. There are two important features that need to be kept in mind in this regard:

1. **Cost**
   It is low for hard disk as compared to semi-conductor memory.
2. **Latency**
   Typically latency of a hard disk is in milliseconds. For SRAM, it is $10^5$ times lower as compared to hard disk.

# Lecture No. 33

## Error Control

Reading Material

William Stallings 6th edition
Computer Organization and Architecture

*Summary*

- Operating System Interface
- Error Control
- RAID

### Operating System Interface

The Operating system interface plays an important role for disk operation. Operating system would define a logic block telling the controller about the track, sector, etc. There are different ways to define logic blocks. For example, we can define 5 bytes containing this information such that: the first 4 bits contain disk number(in case of a system having more than one disk), the next 4 bits contain the address of a particular track followed by a sector number and at the end, the number of bytes to transferred. So this defines a logical block transferred by the controller. Along this, we have additional information about control and status of the controller. The operating system essentially insulates the users from the hardware details of the disk.

### Error Control

There are two main issues in error control:

1. Detection of Error
2. Correction of Error

For detection of error, we just need to know that there exists an error. When the error is detected then the next step is to ask the so rce to resend that information. This process is called automatic request for repeat. In some cases there is also possibility that redundancy is enough and we reconstruct and find out exactly which particular bits are in error. This is called error correction.

There are three schemes commonly used for error control.

1. P arity code
2. Hamming code
3. CRC mechanism

**1. Parity code**

Along with the information bits, we add up another b t, which is called the parity bit. The objective is the total number of 1"s as even or odd. If the parity at the receiving end is different, an error is indicated. Once error is found, CPU may request to repeat that data. The concept of parity bit could be enhanced. In such a case, we would like to increase the distance between different code words. Consider a code word consists of four bits, 0000, and second code word consists of 1111. The distance between two codes is four. So the distance between the two codes would be the number of bits in which they differ from each other. So the concept of introducing redundancy is increase this distance. Larger the distance, higher will be the capacity of the code. For single parity, the distance is two, we can only detect the parity. But if the distance is three, we could also correct these single errors.

If D= minimum distance between two code words then D-1 errors could be detected and D/2 errors could be corrected.

**2. Hamming code**

Hamming code is an example of block code. We have an encoder which could be a program or a hardware device. We feed k inputs to it. These are k information input bits. We also feed some extra bits. Let r be the number of redundant bits. So at output we have r+k = m bits. As an example, for parity bit, we have k=7 and r=1 and m=8. So for 7 bits we get eight output bits.

For any positive integer m<=3, a Hamming code with following parameters exists:

- Code Length:
  $n=2^m-1$
- Number of information symbols:
  $k = 2^m-1-m$
- Number of parity-check symbols:
  $n – k = m$

**3. CRC**

The basic principle for CRC is very simple. We divide a particular code word and make it divisible by a prime number, and if it is divisible by a prime number then it is a valid code word.

CRC does not support error correction but the CRC bits generated can be used to detect multi-bit errors. At the transmitter, we generate extra CRC bits, which are appended to the data word and sent along. The receiving entity can check for errors by re computing the CRC and comparing it with the one that was transmitted.

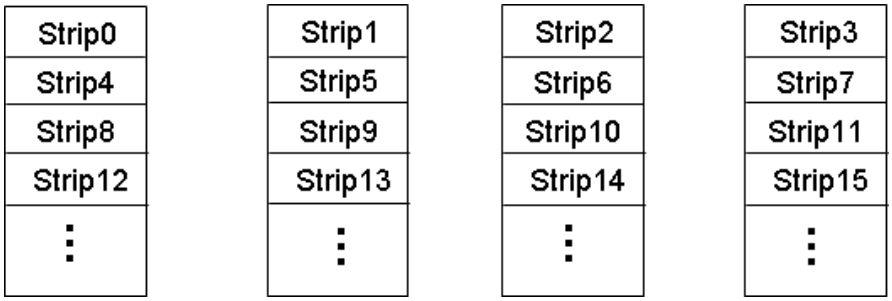
CRC has lesser overhead as compared to Hamming code. It is practically quite simple to implement and easy to use.

**RAID**

The main advantage of having an array of disks is that we could have a simultaneous I/O

request. Latency could also be reduced..

**RAID Level 0**



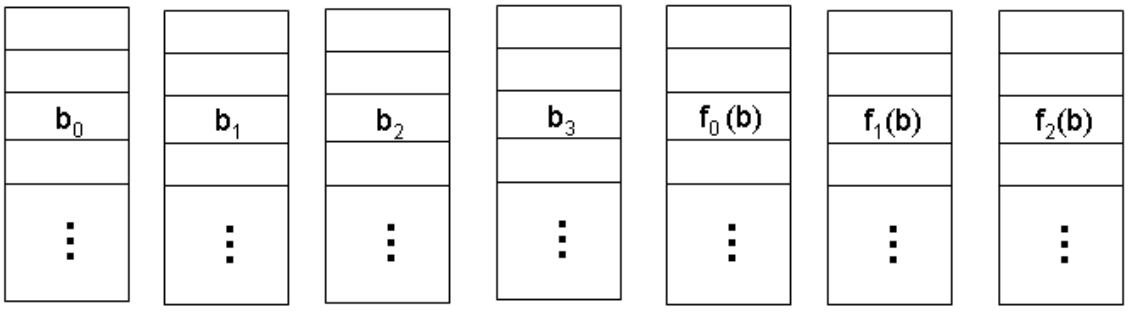

- Not a true member of the RAID family.
- Does not include redundancy to improve performance.
- In few applications, capacity and performance are primary concerns than improved reliability. So RAID level 0 is used in such applications.
- The user and system data are distributed across all the disks in the array.
- Notable advantage over the use of a single large disk.
- Two requests can be issued in parallel, reducing the I/O queuing time.

**Performance of RAID Levels**
Performance of RAID Levels depends upon two factors:

- Request pattern of the host system
- Layout of the data

**Similarities between RAID Levels 2 and 3**

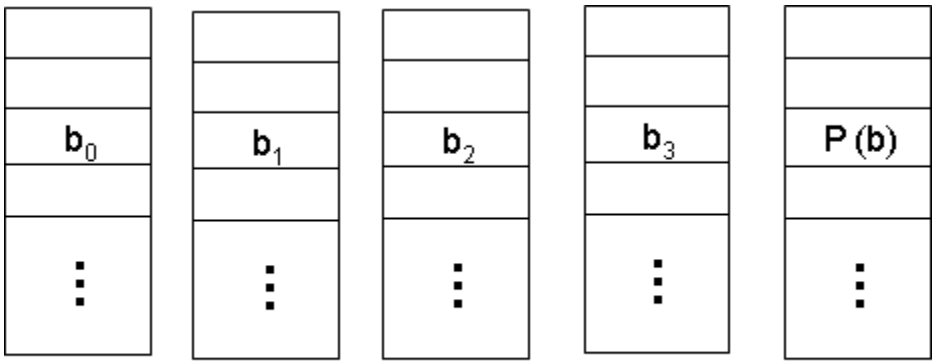
- Make use of parallel access techniques.
- All member disk participate in execution of every request.
- Spindles of the individual drives are synchronized
- Data striping is used
- Strips are as small as a single byte or word.




RAID Level 2

**Differences between RAID2 and RAID 3**

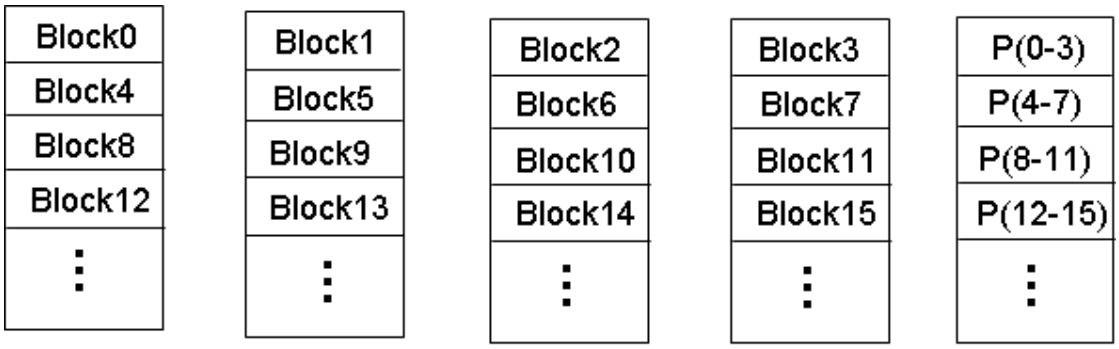
- In RAID 2, error-correcting code is calculated across corresponding bits on each data disk.
- RAID 3 requires only a single redundant disk.
- Instead of an error-correcting code, a simple parity bit is computed for the set of individual bits in RAID 3




RAID Level 3

**RAID Level 4**

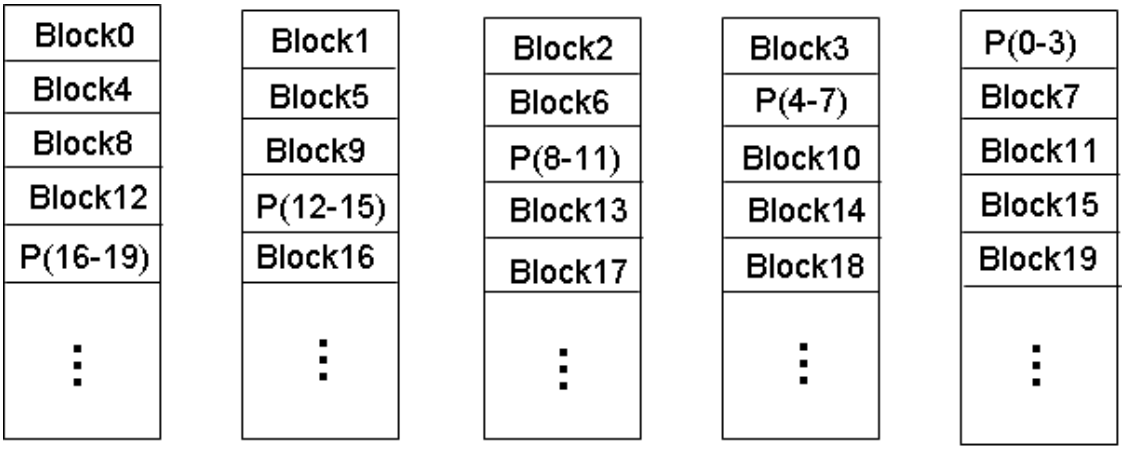
- Make use of independent access technique.
- Data str ping is use .
- A bit-by-bit parity strip is calculated across corresponding strip on each data disk.
- Involves a write penalty when an I/O write request of small size is performed.
- To calculate the new parity, the array management software must read the old user parity strip.




RAID Level 4

**RAID Level 5**

- Organized in a similar fashion to RAID 4
- The only difference is that RAID 5 distributes the parity strips across all disks.




RAID Level 5

# Lecture No. 34

## Number Systems and Radix Conversion

Reading Material

Vincent P. Heuring & Harry F. Jordan — Chapter 6
Computer Systems Design and Architecture — 6.1, 6.2

*Summary*

- Introduction to ALSU
- Radix Conversion
- Fixed Point Numbers
- Representation of Numbers
- Multiplication and Division using Shift Operation
- Unsigned Addition Operation

### Introduction to ALSU [26]

ALSU is a combinational circuit so inside an ALSU, we have AND, OR, NOT and other different gates combined together in different ways to perform addition, subtraction, and, or, not, etc. Up till now, we consider ALSU as a "black box" which takes two operands, a and b, at the input and has c at the output. Control signals whose values depend upon the opcode of an instruction were associated with this black box.

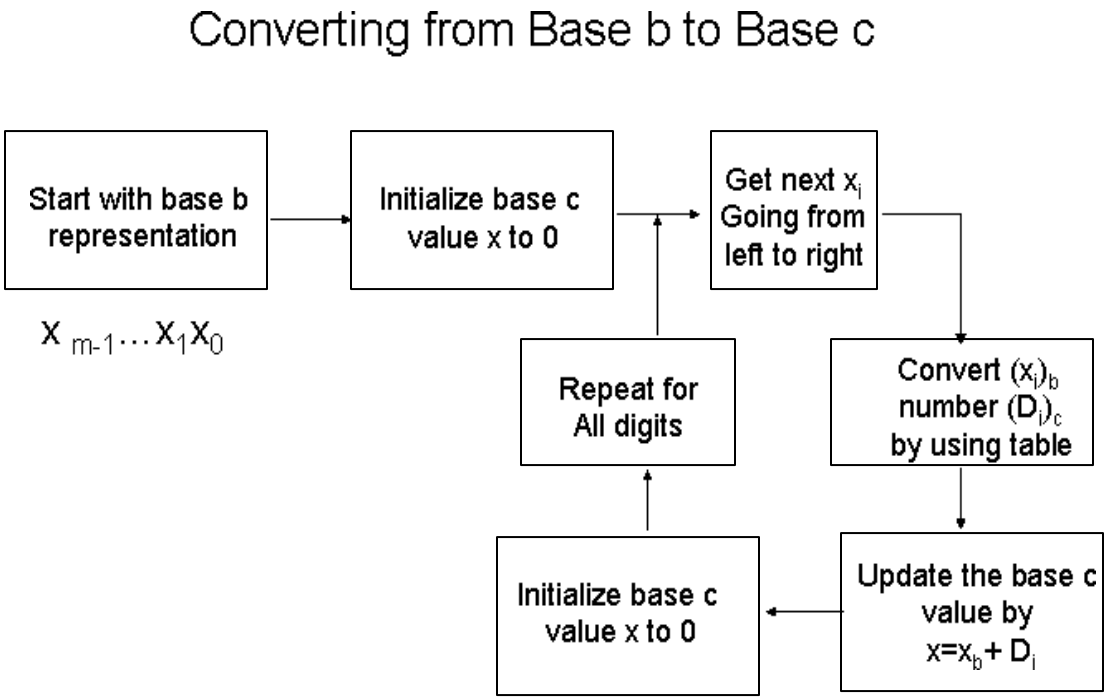
In order to understand the operation of the ALSU, we need to understand the basis of the representation of the numbers. For example, a designer needs to specify how many bits are required for the source operands and how many will be needed for the destination operand after an operation to avoid overflow and truncation.

### Radix Conversion

Now we will consider the conversion of numbers from a representation in one base to another. As human works with base 10 and computers with base 2, this radix conversion operation is important to discuss here. We will use base c notion for decimal representation and base b for any other base. The following figure shows the algorithm of

[26] In our discussion we have used ALU and ALSU for the same thing. We use ALSU when the shift aspect also needs to be emphasized.

converting from base b to base c:



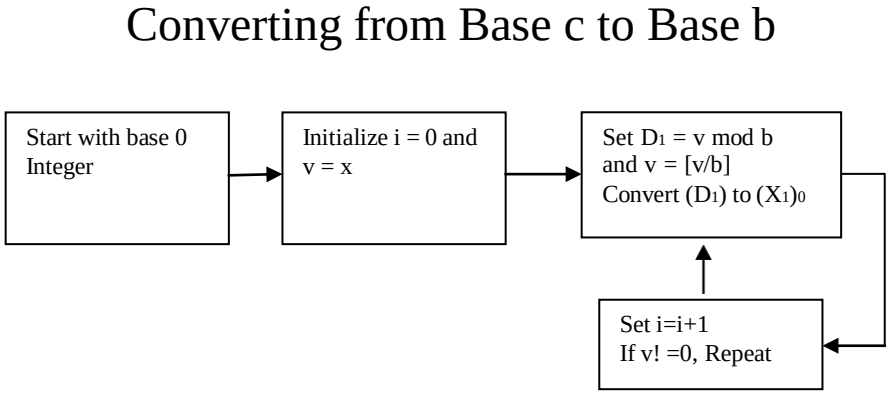

**Example 1**

Convert the hexadecimal number $B3_{16}$ to base 10.

**Solution**

According to the above algorithm,
X=0
X= x+B (=11) =11
X=16*11+3= 179
Hence $B3_{16}=179_{10}$

The following figure shows the algorithm of converting from base c to base b:



**Example 2**

Convert $390_{10}$ to base 16.

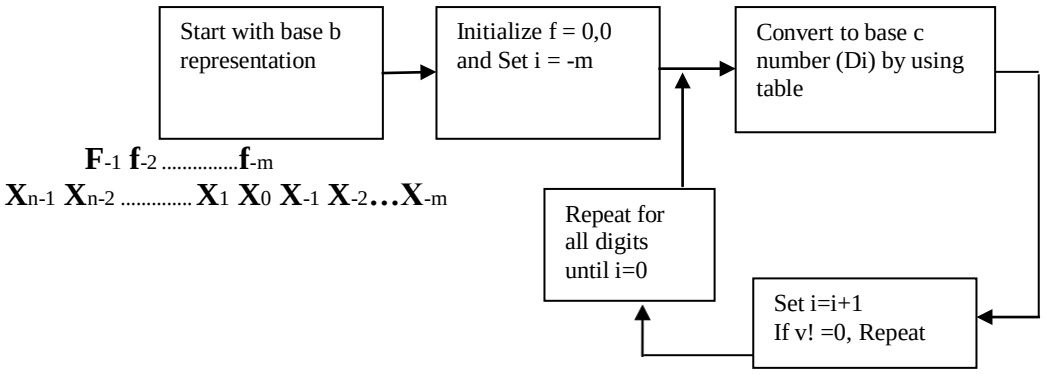
**Solution**

According to the above algorithm
390/16 =24( rem=6), x0=6
24/16= 1(rem=8), x1=8, x2=1
Thus $390_{10}=186_{16}$

## Fixed Point Numbers

Suppose we have a number with a radix point. For example, in 16.12, there are two digits on the left side and two digits on the right of the decimal point. In this case, the radix point is a decimal point because we suppose that given number is a decimal number.
If we have an integer, then this decimal point will be on the right most position i.e. 1612.0 and if it is in fraction then decimal will be at the left most position i.e. 0.1612
There are situations when we shift the position of the radix point. Shifting of the radix point towards left or right is called scaling and we could have multiplication with a base or division by a base respectively.
The following figure shows the algorithm of converting a base b fraction to base c:

Converting from Base c to Base b



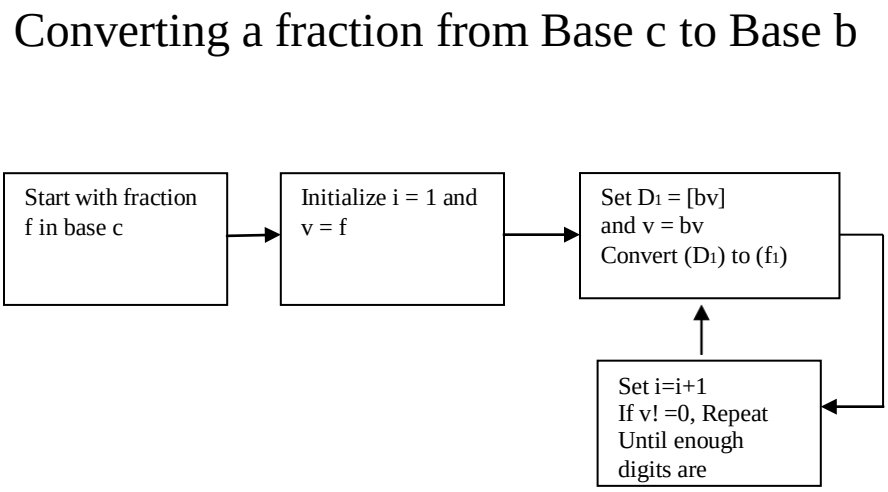
**Example 3**

Convert $(.4cd)_{16}$ to Base 10.

**Solution**

F=0
F=(0+13)/16=0.8125
F=(0.8125+12)/16=0.80078125
F=(0.80078125+4)/16=$(0.3000488)_{10}$

The following figure shows the algorithm of converting fraction from base c to base b:



**Example 4**

Convert $0.24_{10}$ to base 2.

**Solution**

0.24*2=0.48, $f_{-1}$=0
0.48*2=0.96, $f_{-2}$=0
0.96*2=1.92, $f_{-3}$=1
0.92*2=1.84, $f_{-4}$=1
0.84*2=1.68, $f_{-5}$=1,…
Thus $0.24_{10}$ =$(0.00111)_2$

## Representation of Numbers

There are four possibilities to represent integers.

1. Sign magnitude form
2. Radix complement form
3. Diminished radix complement form
4. Biased representation

**Sign magnitude form**

- This is the simplest form for representing a signed number
- A symbol representing the sign of the number is appended to the left of the number
- This representation complicates the arithmetic operations

**Radix complement form**

- This is the most common representation.
- Given an m-digit base b number x, the radix complement of x is

$$x^c = ( b^m – x) \bmod b^m$$

- This representation makes the arithmetic operations much easier.

**Diminished radix complement form**

- The diminished radix complement of an m-digit number x is

$$x^{c\epsilon\epsilon}=b^m -1- x$$

- This complement is easier to compute than the radix complement.
- The two complement operations are interconvertible, as

$$x^c= ( x^{c\epsilon\epsilon}+1)\bmod b^m$$

Table 6.1 of the text book shows the complement representation of negative numbers for radix complement and diminished radix complement form:
Table 6.2 of the text book shows the base 2 complement representation for 8-bit 2‟s and 1‟s complement numbers.

**Example 5**
The following table shows the decimal values in 2‟s complement, 1‟s complement, sign magnitude, 16‟s complement and in unsigned form:

| Decimal | 2's complement | 1's complement | Sign-magnitude | 16's complement | Unsigned |
|---|---|---|---|---|---|
| 27 | 011011 | 011011 | 011011 | 1B | 11011 |
| .17 | 0.00101011 | 0.00101011 | 0.00101011 | 0.2B | 0.00101011 |
| -26 | 100110 | 100101 | 111010 | E6 | - |
| -0.57 | 1.01101110 | 1.01101101 | 1.10010010 | F.6E | - |

## Multiplication and Division using Shift Operation

Shift left and shift right are two important operations used for various purposes. One typical example could be multiplication or division by base b. The following examples explain multiplication and division by using shift operation.

**Example 6**

- 6x4
  $00110_2 \times 4_{10} = 11000_2 = 24_{10}$

Overflow would occur if we will use 4 bits instead of 5 bits here.

- 60/16
  $0111100_2/16_{10} = 0000011_2 = 3_{10}$

The fractional portion of the result is lost.

**Example 7**

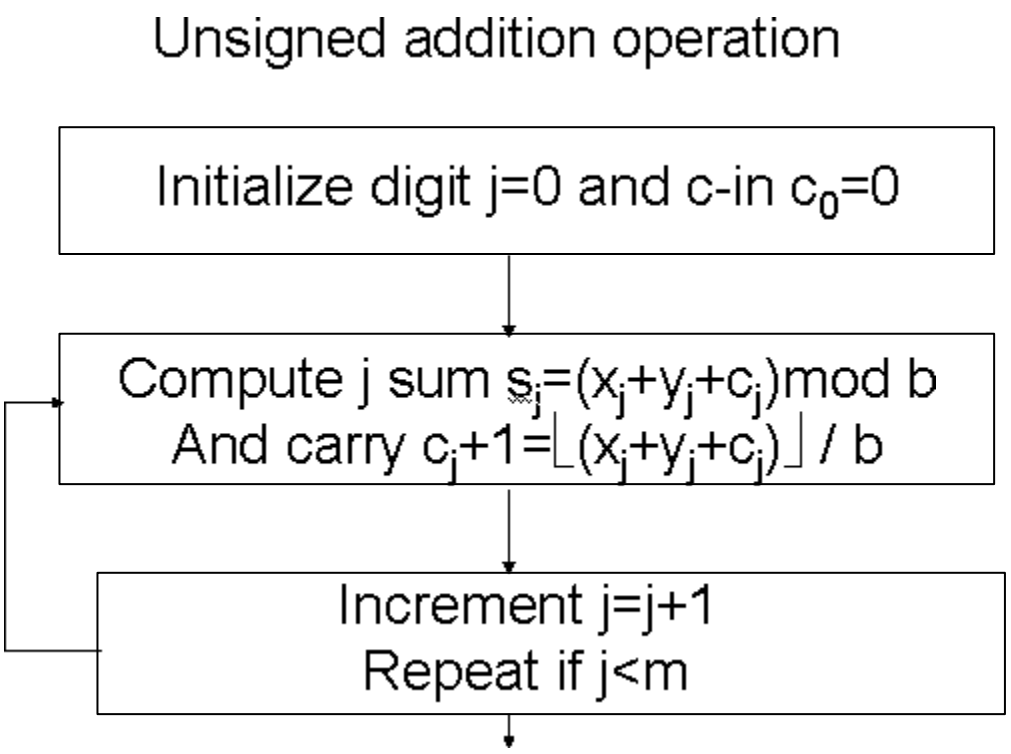
- -6x4
  $-6 = (11010)_2$
  $-6x4 = (01000)_2 = 8$ which is wrong!

using less no. of bits might change sign
So, -6 = $(111010)_2$
-6x4 = $(101000)_2$ = -24

**Example 8**

Multiplication and division of negative numbers

**Solution**




-24x2
-24= $(101000)_2$
-24x2= $(010100)_2$ = 20
-24x2= $(110100)_2$ = -12
Changing the size of the number,
24= 011000 (n=6) to 00011000 (n=8)
-24= 101000 (n=6) to 11101000 (n=8)

## Unsigned Addition Operation

The following diagram shows the digit wise procedure for adding m-digit base b numbers, x and y:

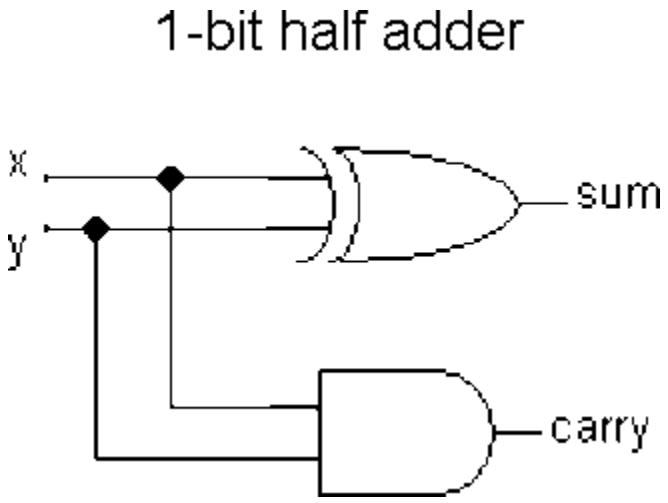
**Example 9**

Unsigned addition in base 2 and base16.

**Solution**

| **Base 16 addition** | **Base 2 addition** |
|---|---|
| A B 4 2 $_{16}$<br>+ 3 1 C 1 $_{16}$<br>carry 0 1 0 0<br>sum D D 0 3 $_{16}$ | 100011 $_2$<br>+ 011011 $_2$<br>carry 000110<br>sum 111110 $_2$ |

The following diagram shows the logic circuit for 1-bit half adder. It takes two 1-bit inputs x and y and as a result, we get a 1-bit sum and a 1-bit carry. This circuit is called a half adder because it does not take care of input carry. In order to take into account the effect of the input carry, a 1-bit full adder is used which is also shown in the figure. We




can add two m-bit numbers by using a circuit which is made by cascading m 1-bit full adders.

The situation, when addition of unsigned m-bit numbers results in an m+1 bit number, is

called overflow. Overflow is treated as exception in some processors and the overflow flag is used to record the status of the result.

# Lecture No. 35

## Multiplication and Division of Integers

**Reading Material**

Vincent P. Heuring & Harry F. Jordan — Chapter 6
Computer Systems Design and Architecture — 6.3, 6.4

*Summary*

- Overflow
- Different Implementations of the adder
- Unsigned and Signed Multiplication
- Integer and Fraction Division
- Branch Architecture

**Overflow**

When two m-bit numbers are added and the result exceeds the capacity of an m-bit destination, this situation is called an overflow. The following example describes this condition:

**Example 1**

Overflow in fixed point addition:

| Base 2 | Base 8 | Base 16 |
|---|---|---|
| 1011.1 | 7.25 | C36.A |
| +1110.0 | +6.76 | +72B.3 |
| c 1100 | c 11 .1 | c 1010 |
| s 11001.1 | s 16.23 | s 1361.D |

In these three cases, the fifth position is not allowed so this results in an overflow.

**Different Implementations of the Adder**

For a binary adder, the sum bit is obtained by following equation:

$$s_j = x_j y_j c_j + x_j \bar{y}_j \bar{c}_j + \bar{x}_j y_j \bar{c}_j + \bar{x}_j \bar{y}_j c_j$$

and the equation for carry bit is

$$c_{j+1} = x_j y_j + x_j c_j + y_j c_j$$

where x and y are the input bits.
The sum can be computed by the two methods:

- Ripple Carry Adder
- Carry Look ahead Adder

**Ripple Carry Adder**
In this adder circuit, we feed carry out from the previous stage to the next stage and so on. For 64 bit addition, 126 logic levels are required between the input and output bits. The logic levels can be reduced by using a higher base (Base 16). This is a relatively slow process.

**Complement Adder/Subtractor**

We can perform subtraction using an unsigned adder by

- Complement the second input
- Supply overflow detection hardware

**2"s Complement Adder/Subtractor**
A combined adder/subtractor can be built using a mux to select the second adder input. In this case, the mux also determines the carry-in to the adder. The equation for mux output is :

$$q_j = y_j \bar{r} + \bar{y}_j r$$

**Carry Look ahead Adder**
The basic idea in carry look ahead is to speed up the ripple carry by determining whether the carry is generated at the j position after addition, regardless of the carry-in at that stage or the carry is propagated from input to output in the digit.
This results in faster addition and lesser propagation delay of the carry bits. It divides the carry into two logical variables $G_j$ (generate) and $P_j$ (propagate). These variables are defined as:

$$G_j = x_j y_j$$
$$P_j = x_j + y_j$$

Hence the carry out will be

$$c_{j+1} = G_j + P_j c_j$$

Here the G and P each require one gate, and the sum bit needs two more gates in the full adder. This results in a less complexity i.e. log(m) which is much less as compare to

ripple carry adder where complexity is m (m is the number of bits of a digit to be added). Ripple carry and look ahead schemes are can be mixed by producing a carry-out at the left end of each look ahead module and using ripple carry to connect modules at any level of the look ahead tree.

**Unsigned Multiplication**

The general schema for unsigned multiplication in base b is shown in Figure 6.5 of the text book.

**Parallel Array Multiplier**

Figure 6.6 of the text book shows the structure of a fully parallel array multiplier for base b integers. All signal lines carry base b digits and each computational block consists of a full adder with an AND gate to form the product $x_i y_j$. In case of binary, $m^2$ full adders are required and the signals will have to pass through almost 4m gates.

**Series parallel Multiplier**

A combination of parallel and sequential hardware is used to build a multiplier. This results in a good speed of operation and also saves the hardware.

**Signed Multiplication**

The sign of a product is easily computed from the sign of the multiplier and the multiplicand. The product will be positive if both have same sign and negative if both have different sign. Also, when two unsigned digits having m and n bits respectively are multiplied, this results in a (m+n) –bit product, and (m+n+1)-bit product in case of sign digits. There are three methods for the multiplication of sign digits:

1. 2‟s complement multiplier
2. Booth recoding
3. Bit-Pair recoding

**2‟s complement Multiplication**

If numbers are represented in 2‟s complement form then the following three modifications are required:

1. Provision for sign extension
2. Overflow prevention
3. Subtraction as well as addition of the partial product

**Booth Recoding**

The Booth Algorithm makes multiplication simple to implement at hardware level and speed up the procedure. This procedure is as follows:

1. Start with LSB and for each 0 of the original number, place a 0 in the recorded number until a 1 in indicated.
2. Place a 1 for 1in the recorded table and skip any succeeding 1‟s until a 0 is encountered.
3. Place a 0 with 1 and repeat the procedure.

**Example 2**

Recode the integer 485 according to Booth procedure.

**Solution**
Original number:
00111100101=256+128+64+32+4+1=485
Recoded Number:

$0100\bar{1}01\bar{1}\bar{1}1$=+512-32+8-4+2-1=485

**Bit-Pair Recoding**

Booth recoding may increase the number of additions due to the number of isolated 1s. To avoid this, bit-pair recoding is used. In bit-pair recoding, bits are encoded in pairs so there are only n/2 additions instead of n.

**Division**

There are two types of division:

- Integer division
- Fraction division

**Integer division**

The following steps are used for integer division:

1. Clear upper half of dividend register and put dividend in lower half. Initialize quotient counter bit to 0
2. Shift dividend register left 1 bit
3. If difference is +ve, put it into upper half of dividend and shift 1 into quotient. If –ve, shift 0 into quotient
4. If quotient bits<m, goto step 2

5. m-bit quotient is in quotient register and m-bit remainder is in upper half of dividend register

**Example 3**

Divide $47_{10}$ by $5_{10}$.

**Solution**

D=000000 101111, d=000101

| | | | |
|---|---|---|---|
| D | 000001 011110 | | |
| d | 000101 | | |
| Diff(-) | | q | 0 |
| D | 000010 111100 | | |
| d | 000101 | | |
| Diff(-) | | q | 00 |
| D | 000101 111000 | | |
| d | 000101 | | |
| Diff(+) | | q | 001 |
| D | 000001 110000 | | |
| d | 000101 | | |
| Diff(-) | | q | 0010 |
| D | 000011 100000 | | |
| d | 000101 | | |
| Diff(-) | | q | 00100 |
| D | 000111 000000 | | |
| d | 000101 | | |
| Diff(+) | 000010 | q | 001001 |

Hence remainder = $(000010)_2 = 2_{10}$
Quotient = $(001001)_2 = 9_{10}$

**Fraction Division**

The following steps are used for fractional division:

1. Clear lower half of dividend register and put dividend in upper half. Initialize quotient counter bit to 0
2. If difference is +ve, report overflow
3. Shift dividend register left 1 bit
4. If difference is +ve, put it into upper half of dividend and shift 1 into quotient. If negative, shift 0 into quotient
5. If quotient bits<m, go to step 3

6. m-bit quotient has decimal at the left end and remainder is in upper half of dividend register

**Branch Architecture**

The next important function perform by the ALU is branch. Branch architecture of a machine is based on

1. condition codes
2. conditional branches

**Condition Codes**

Condition Codes are computed by the ALU and stored in processor status register. The „comparison" and „branching" are treated as two separate operations. This approach is not used in the SRC. Table 6.6 of the text book shows the condition codes after subtraction, for signed and unsigned x and y. Also see the SRC Approach from text book.

Usually implementation wi h flags is easier however it requires status registers. In case of branch instructions, decision is based on the branch itself.

Note: For more information on this topic, please see chapter 6 of the text book.

# Lecture No. 36

## Floating-Point Arithmetic

Reading Material

| | |
|---|---|
| Vincent P. Heuring & Harry F. Jordan | Chapter 6 |
| Computer Systems Design and Architecture | 6.3.2, 6.4, 6.4.1 |
| | 6.4.2, 6.4.3 |

*Summary*

- NxN Crossbar Design for Barrel Rotator
- Barrel Shifter with Logarithmic Number of Stages
- ALU Design
- Floating-Point Representations
- IEEE Floating-Point Standard
- Floating-Point Addition and Subtraction
- Floating-Point Multiplication
- Floating-Point Division

### NxN Crossbar Design for Barrel Rotator

Figure 6.11 of the text book
The figure shows an NxN crossbar design for barrel rotator. x indicates the input. So x0,x1,…,xn-1 are applied to the rows. The vertical lines are indicated by y1, y2,…yn-1 where y shows the output. So this forms a cross of x and y and the number of cross points are NxN. There is also a connection between each input and output using a tri-state buffer. At the input, we have a decoder which is used to select the shift count. Each output from the decoder is connected diagonally to the tri-state buffers. This arrangement requires N2 gates.

### Barrel Shifter with Logarithmic Number of Stages

Another alternate to an NxN crossbar barrel rotator is a logarithmic barrel shifter. This design is time-space trade-off. In this case, the number of shifts required is eight, and then there will be three stages for this purpose. Now a word is passed as input to the shifter. There are two possibilities. First the input word is passed to the next stage without any shift. This process is called bypass and second option is shift. The word is passed to the next stage after shift.

For the first stage, we have 1-bit right shift, for second stage, 2-bit right shift and so on. There is also a shift count unit which controls the number of shifts. For example, if 1-bit shift is required then only s0 will be one and other signals from shift count will be zero. If we want a 3-bit shift, then s0 and s1 will be 1 and all other signals will be zero.
The figure also shows one shift/bypass cell which is a combinational logic circuit. A shift/bypass signal decides whether the input word should be shifted or bypassed. This design requires only O (NlogN) switches but propagation delay has increased i.e. from O(1) to O(logN).

Figure 6.12 of the text book

## ALU Design

ALU is a combination of arithmetic, logic and shifter unit along with some multiplexers and control unit. T e idea is that based on the op-code of an instruction, appropriate control signals are activated to perform required ALU operation.
Figure 6.13 of the text book
The diagram shows two inputs x and y and one output z. All these are of n-bits. The inputs x and y are simultaneously provided to arithmetic, logic and shifter unit. There is a control unit which accepts op-code as input. Based on the op-code, it provides control signals to arithmetic, logic and shifter unit. The control unit also provides control signals to the two multiplexers. One mux has three inputs; each from arithmetic, logic and shifter unit and its output is z. The second mux provides status output corresponding to condition codes.

## Floating Point Representations

**Example**

-0.5 × 10-3
Sign = -1
Significand= 0.5
Exponent= -3
Base = 10= fixed for given type of representation

Significand is also called mantissa.
In computers, floating-point representation uses binary numbers to encode significant, exponent and their sign in a single word.
The diagram on Page 293 of the text shows an m-bit floating point number where s represents the sign of the floating point number. If s = 1 then the floating-point number will be a positive number; if s= 0 then it will be a negative number. The e field shows the value of exponent. To represent the exponent, a biased representation is used. So we represent e^ instead of e to show biased representation. In this technique, a number is added to the exponent so that the result is always positive. In general floating point numbers are of the form.
(-1)s × f × 2e

**Normalization**

A normalized, non zero floating point number has a significand whose left-most digit is non- zero and is a single number.
Example

0.56 × 10-3… ..... (Not normalized)
5.6 × 10-3… ..... (Normalized form)

Same is the case for binary.

## IEEE Floating-Point Standard

IEEE floating -point standard has the following features.

**Single-Precision Binary Floating Point Representation**

- 1-bit sign
- 8-bit exponent
- 23-bit fraction
- A bias of 127 is used.

Figure 6.15 of the text book

**Double precision Binary Floating Point Representation**

- 1-bit sign
- 11-bit exponent
- 52-bit fraction
- Exponent bias is 1023

Figure 6.16 of the text book.

**Overflow**

In table 6.7 of the text book, e^= 255, denotes numbers with no numeric value including + ∞ and - ∞ and called Not-a-Number or NaN. In computers, a floating-point number ranges from $1.2 \times 10^{-38} \leq x \leq 3.4 \times 1038$ can be represented. If a number does not lie in this range, then overflow can occur.
Overflow occurs when the exponent is too large and can not be represented in the exponent field.

## Floating –Point Addition and Subtraction

The following are the steps for floating-point addition and subtraction.

- Unpack sign , exponent and fraction fields
- Shift the significand
- Perform addition
- Normalize the sum
- Round off the result
- Check for overflow

Figure 6.17 of the text book.

**Example 1**
Perform addition of the following floating-point numbers.
0.510 , -0.437510
Binary:
$0.5_{10} = 1/2_{10} = 0.1_2 = 1.000 \times 2\text{-}1$
$-0.4375_{10} = -7/16_{10} = -7/24 = -0.0111_2 = -1.110 \times 2\text{-}2$

Align: $-1.110 \times 2\text{-}2 \rightarrow -0.111 \times 2\text{-}1$

Addition: $1.000 \times 2\text{-}1 + (-0.111 \times 2\text{-}1) = 0.001 \times 2\text{-}1$

Normalization of Sum:

$0.001_2 \times 2\text{-}1 = 0.010_2 \times 2\text{-}2$

$= 1.000_2 \times 2\text{-}4$

**Hardware Structure for Floating-Point Add and Subtract**
Figure 6.17 of the text book.

## Floating-Point Multiplication

The floating-point multiplication uses the following steps:

- Unpack sign, exponent and significands
- Apply exclusive-or operation to signs, add exponents and then multiply significands.
- Normalize, round and shift the result.
- Check the result for overflow.
- Pack the result and report exceptions.

## Floating-Point Division

The floating-point division uses the following steps:

- Unpack sign, exponent and significands
- Apply exclusive-or operation to signs, subtract the exponents and then divide the significands.
- Normalize, round and shift the result.
- Check the result for overflow.
- Pack the result and report exceptions.

# Lecture No. 37

## Components of memory Systems

**Reading Material**

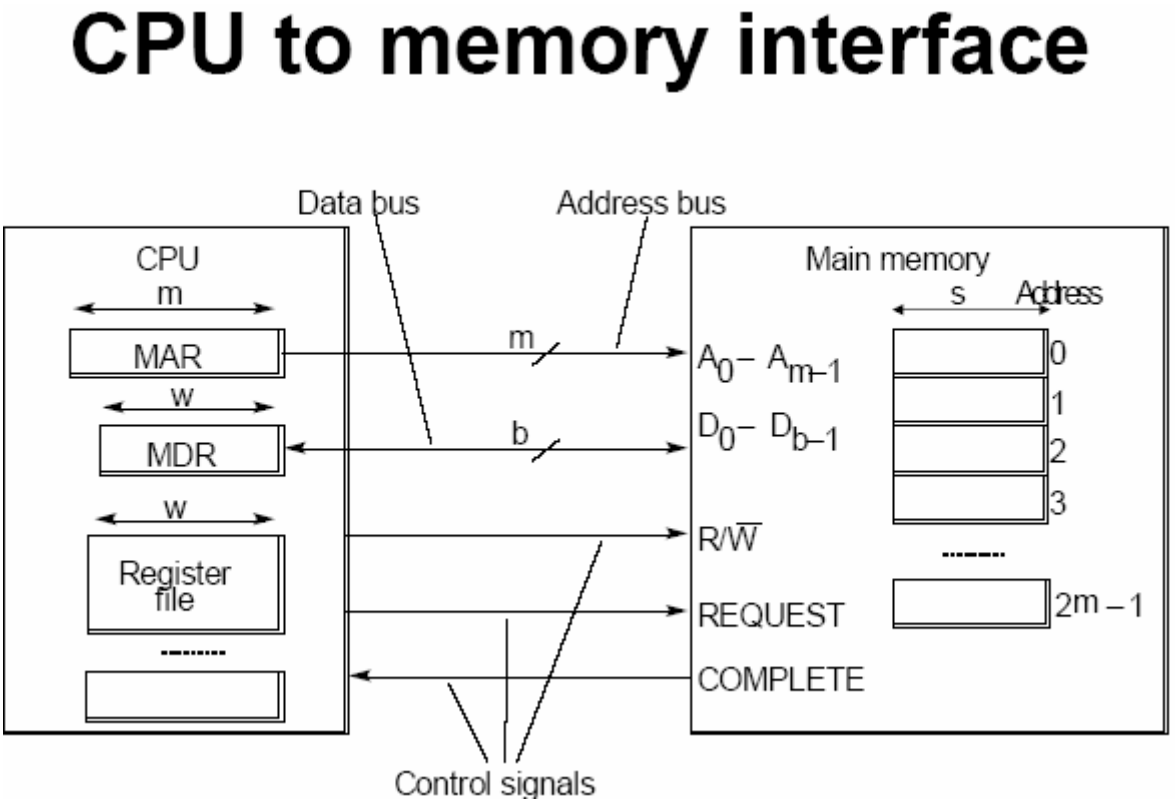
Vincent P. Heuring & Harry F. Jordan — Chapter 7
Computer Systems Design and Architecture — 7.1, 7.2

### *Summary*

- CPU to Memory Interface
- Static RAM cell Organization and Operation
- One & two Dimensional Memory Cells
- Matrix and Tree Decoders
- Dynamic RAM

## CPU to Memory Interface

The memory address register (MAR) is m-bits wide and contains memory address generated by the CPU directly connected to the m-bit wide address bus. The memory buffer register (MBR) is w-bit wide and contains a data word, directly connected to the data bus which is b-bit wide. The register file is a collection of 32, 32-bit wide registers used for data transfer between memory and the CPU. Memory address ranges from 0 to $2^m$-1.There also exist three control signals: $R/\overline{W}$ , REQUEST, and COMPLETE. When $R/\overline{W}$ signal is high, this would correspond to a read operation equivalent to having an input data to the CPU and output from the memory. If this signal is low then it would be a write operation and data would come from the CPU as an output and it would be written into a portion in the memory. In this case, the REQUEST signal coming from the CPU telling the memory that some interaction is required between the CPU and memory. As a result of this request (either read/write), along with the signal on the control and the address on the address bus, we might have the corresponding data on the data bus for a read operation and after the operation is complete, the memory would issue a control signal which corresponds in this case to COMPLETE.
Figure 7.1 of the text book.

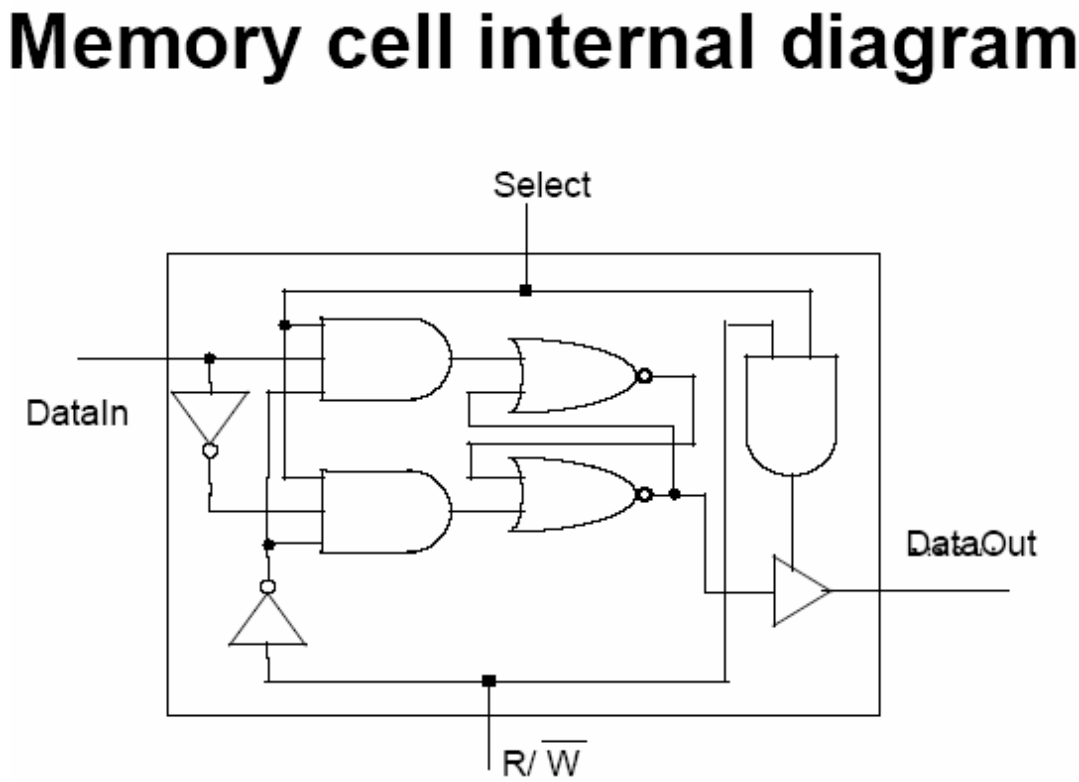



## Static RAM Cell Organization and Operation

**A Typical Memory Cell**

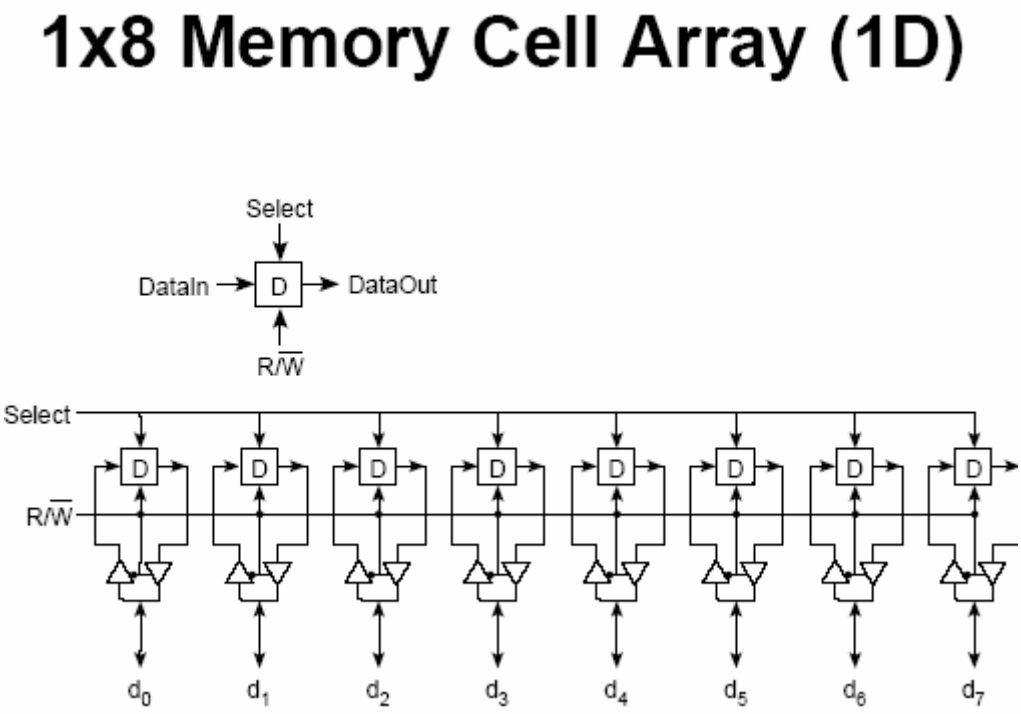
A memory cell provides four functions: Select, DataIn, DataOut, and Read/Write. DataIn means input and DataOut means output. The select signal would be enabled to get an operation of Read/Write from this cell.

Figure 7.3 of the text book.



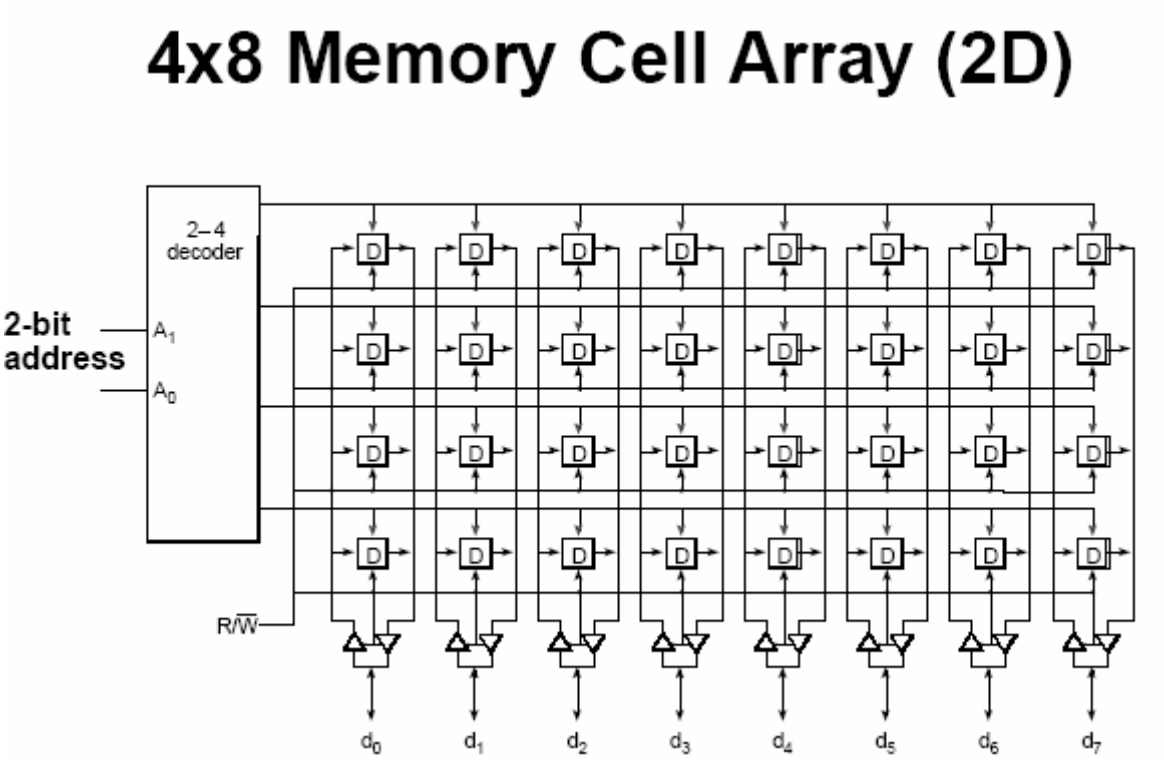

**1×8 Memory Cell Array (1D)**

In this arrangement, each block is connected through a bi-directional data bus implemented with 2 tri-state buffers. $R/\overline{W}$ and Select signals are common to all these cells. This 1-dimentional memory array could not be very efficient, if we need to have a very large memory.



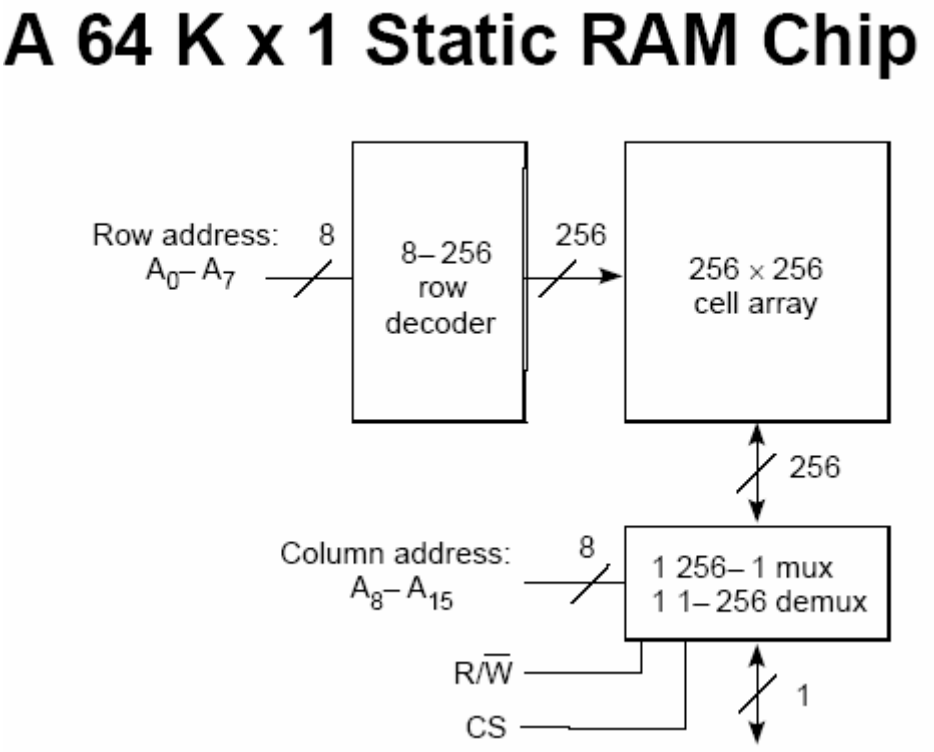

**4×8 Memory Cell Array (2D)**

In this arrangement, 4×8 memory cell array is arranged in 2-dimensions. At the input, we have a 2×4 decoder. Two address bits at the input A0 and A1 would be decoded into 4 select lines. The decoder selects one of four rows of cells and then $R/\overline{W}$ signal specifies whether the row will be read or written.



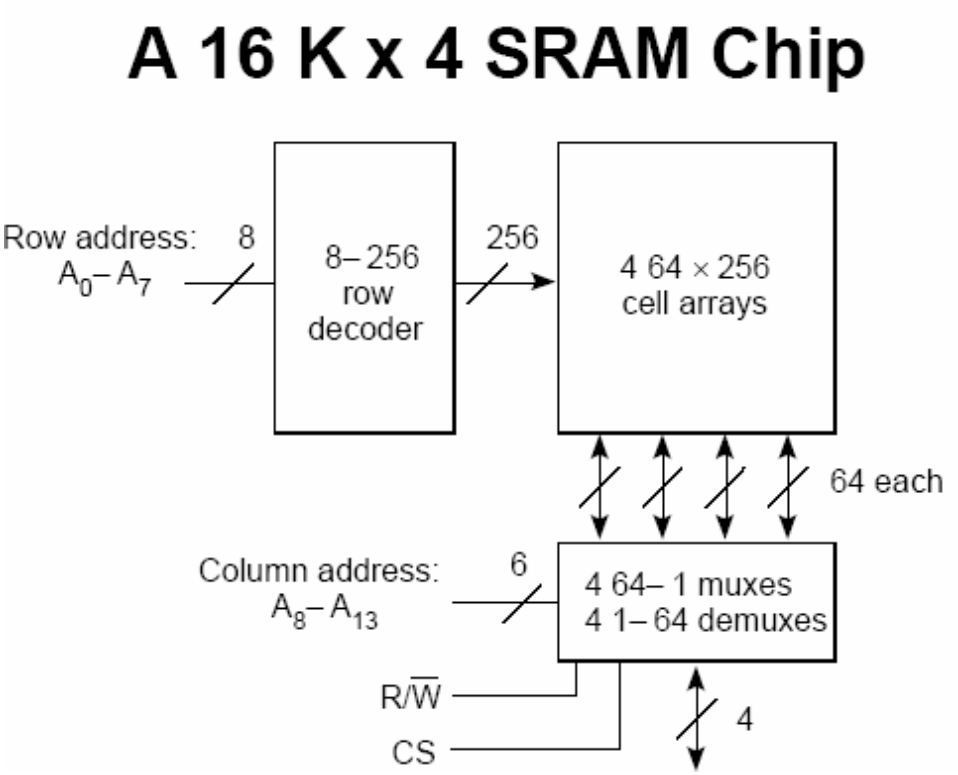

**A 64k×1 Static RAM Chip**

The cell array is indicated as 256 × 256. So, there would be 256 rows and 256 columns. A 64k × 1 cell array requires 16 address lines, a read/write line, $R/\overline{W}$, a chip select line, CS, and only a single data line. The lower order 8-address lines select one of the 256 rows using an 8-to-256 line row decoder. Thus the selected row contains 256 bits. The higher order 8-address lines select one of those 256 bits. The 256 bits in the row selected flow through a 256-to-1 line multiplexer on a read. On a memory write, the incoming bit flows through a 1-to-256 line demultiplexer that selects the correct column of the 256 possible columns.




**A 16k×4 Static RAM Chip**

In this case, memory is arranged in the form of four 64×256 memory cells. Four bits can be read and written at a time. For this, we use one 8-256 row decoder, four 64-1 muxes and four 1-64 de muxes. The lower address lines (A0-A7) are decoded into $2^8$ lines, $2^6$ lines from these $2^8$ are used to select row from one of the four 64×256 cell array and the remaining $2^2$ lines are used to select one of the 64×256 cell array. Now the upper address lines (A8-A13) are input into the 4 muxes and their output is used to select the required column from the four 64×256 cell arrays. Control lines read/write, $R/\overline{W}$, chip select, CS, are just similar to previous arrangement.




## Matrix and Tree Decoders

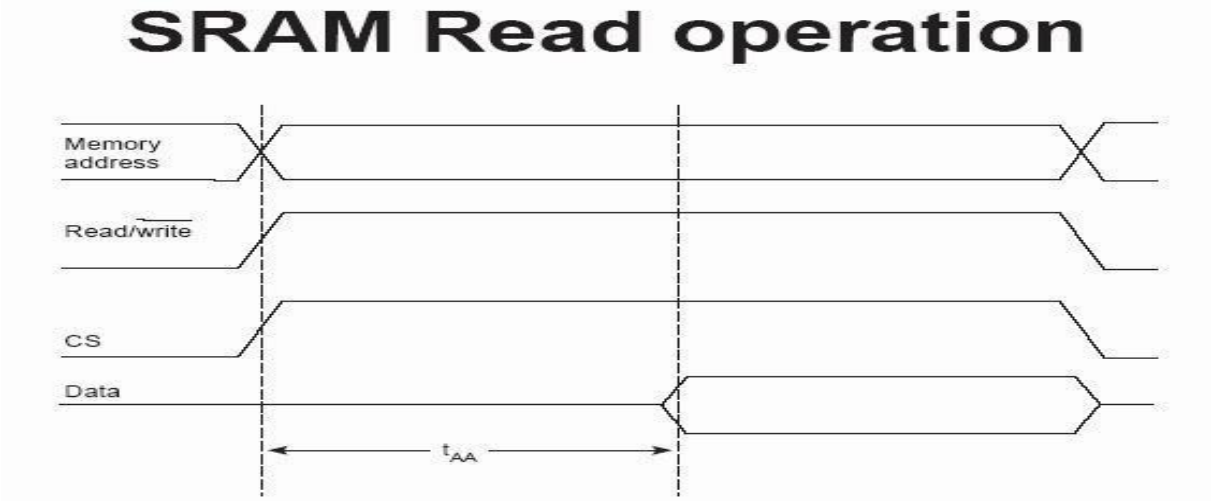
A typical one level decoder has n inputs and $2^n$ output, using one level of gates, each with a fan-in of n. Two level decoders are limited in size because of high gate fan-in. In order to reduce the gate fan-in to a value of 8 or 6, tree and matrix decoders are utilized.

### Six Transistor SRAM Cell

In this arrangement, the cross connection is through inverters to make the latch, the basic storage cell. This implementation uses six transistor cells. One transistor is used to implement each of the two inverters, two transistors are used to control access to the inverters for reading and writing, and two are used as active loads .

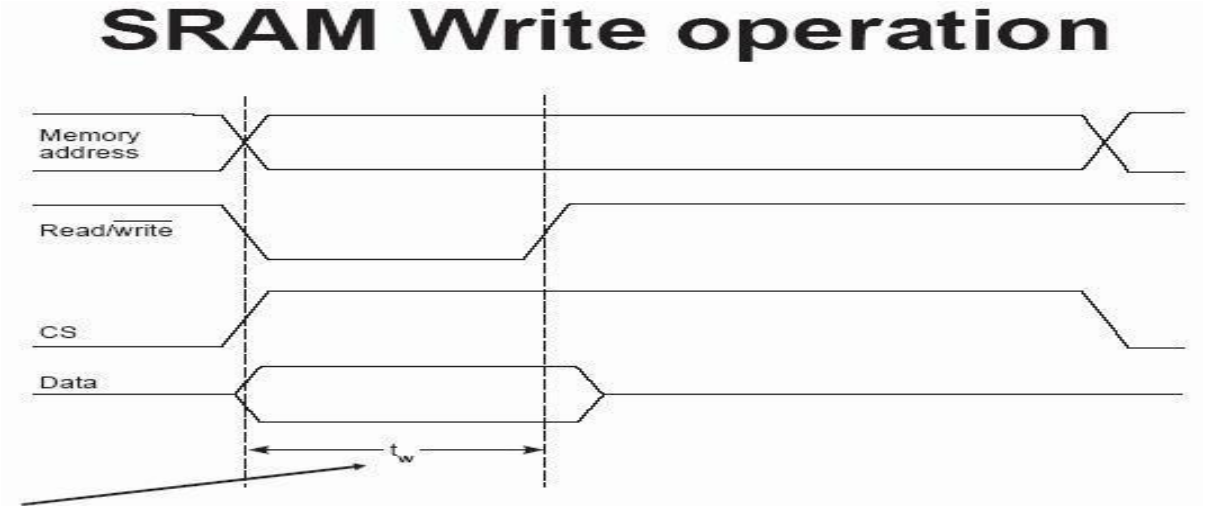
### SRAM Read Operation

First of all, the CPU provides the address on the external address bus. The read/write signal becomes active high. After time "$t_{AA}$", the data becomes available on the data bus. The chip retains this data on the data lines until the control signals are de asserted.



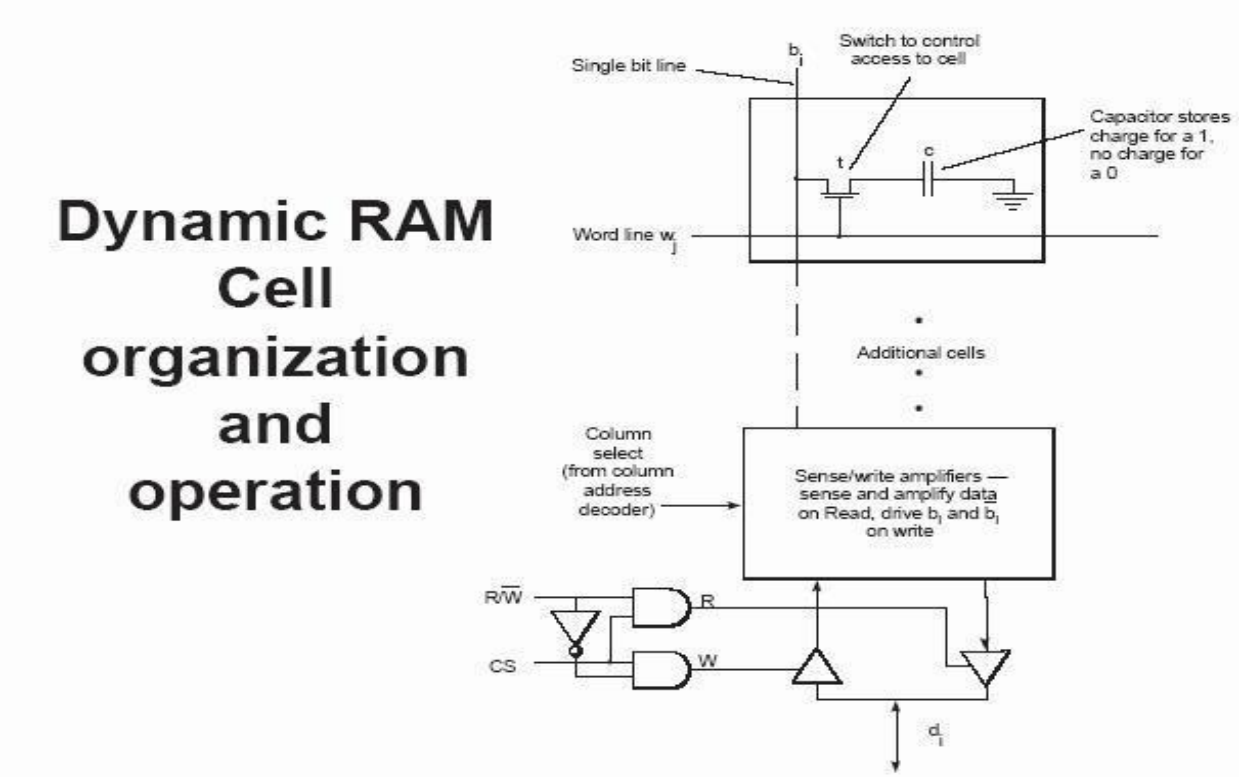

**SRAM Write Operation**

In the case of write cycle, the major di ference is that along with the address the CPU has also provided the data on the data bus. The chip select, CS, is immediately provided and write signal is made low. The $R/\overline{W}$ line must be held valid for a minimum time interval $t_w$ , the write time, until data, address, and control information have been propagated to the cell and strobe into it. During this period the data lines must be driven with the data to be written.




## Dynamic RAM

As an alternate to the SRAM cell, the data can be stored in the form of a charge on a capacitor (a charging/discharging transistor that can become a valid memory element), and this type of memory is called dynamic memory. The capacitor has to be refreshed and recharged to avoid data loss.




**Dynamic RAM Cell Operation**

In a DRAM cell, the storage capacitor will discharge in around 4-15ms. Refreshing the capacitor by reading or sensing the value on bit line, amplifying it, and placing it back on to the bit line is required. The need to refresh the DRAM cell complicates the DRAM system design.

For details, refer to Chapter 7 of the text book.

# Lecture No. 38

## Memory Modules

**Reading Material**

Vincent P. Heuring & Harry F. Jordan — Chapter 7
Computer Systems Design and Architecture — 7.2.6, 7.3

### *Summary*

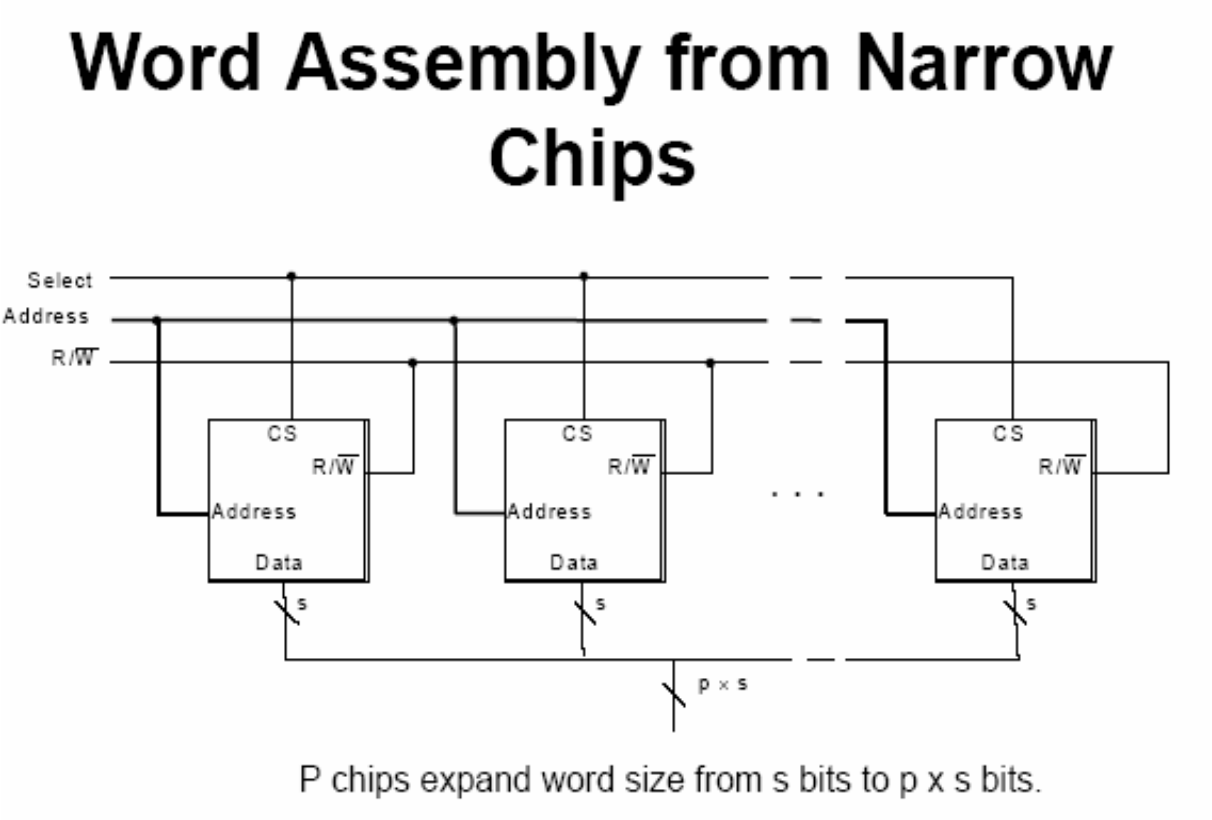
- Memory Modules
- Read Only Memory (ROM)
- Cache

## Memory Module

Static RAM chips can be assembled into systems without changing the timing characteristics of a memory access. Dynamic RAM chips, however, have enough timing complexity that a memory module built from dynamic RAM chips will have complex control. The cause of timing complexity is the time-multiplexed row and column addresses, and the refresh operation.

### Word Assembly from Narrow Chips

Chips can be combined to expand the memory word size while keeping the same number of words. Address, chip select, and R/W signals are connected in parallel to all the chips. Only the data signals are kept separate, with those from each chip supplying different bits of the wider word. For high capacity memory chips, narrow words are used. This is because adding a data pin to a chip with $2^m$ words of s bits increases the number of bits it can store by only a factor of (s+1)/s, while adding an address pin always doubles the capacity.




## Dynamic RAM Module with Refresh Control

For Dynamic RAM chips the total address is divided into row and column address. Row address strobe signal RAS and a column strobe signal CAS are used to differentiate between these two signals.

## Read Only Memory (ROM)

ROM is the read-only memory which contains permanent pattern of data that cannot be changed. ROM is nonvolatile i.e. it retains the information in it when power is removed from it. Different types of ROMs are discussed below.

### PROM

The PROM stands for Programmable Read only Memory. It is also nonvolatile and may be written into only once. For PROM, the writing process is performed electrically in the field. PROMs provide flexibility and convenience.

### EPROM

Erasable Programmable Read-only Memory or EPROM chips have quartz windows and by applying ultraviolet light erase the data can be erased from the EPROM. Data can be restored in an EPROM after erasure. EPROMs are more expensive than PROMs and are generally used for prototyping or small-quantity, special purpose work.

**EEPROM**

EEPROM stands for Electrically Erasable Programmable Read-only Memory. This is a read-mostly memory t at can be written into at any time without erasing prior contents; only the byte or bytes addressed are updated. The write operation takes considerably longer than the read operation. It is more expensive than EPROM.

**Flash Memory**

An entire flash memory can be erased in one or a few seconds, which is much faster than EPRO . In addition, it is possible to erase just blocks of memory rather than an entire chip.

## Cache

Cache by definition is a place for safe storage and provides the fastest possible storage after the registe rs. The cache contains a copy of portions of the main memory. When the CPU attempts to read a word from memory, a check is made to determine if the word is in the cache. If so, the word is delivered to the CPU. If not, a block of the main memory, consisting of some fixed number of words, is read into the cache and then the word is delivered to the CPU.

**Spatial Locality**

This would mean that in a part of a program, if we have a particular address being accessed then it is highly probable that the data available at the next address would be highly accessed.

**Temporal Correlation**

In this case, we say that at a particular time, if we have utilized a particular part of the memory then we might access the adjacent parts very soon.

**Cache Hit and Miss**

When the CPU needs some data, it communicates with the cache, and if the data is available in the cache, we say that a cache hit has occured. If the data is not available in the cache then it interacts with the main memory and fetches an appropriate block of data. This is a cache miss.

# Lecture No. 39

## The Cache

**Reading Material**

Vincent P. Heuring & Harry F. Jordan — Chapter 7
Computer Systems Design and Architecture — 7.4, 7.5

**Summary**

- Cache Organization and Functions
- Cache Controller Logic
- Cache Strategies

### Cache Organization and Functions:

The working of the cache is based on the principle of locality which has two aspects.
**Spatial Locality**: refers to the fact when a given address has been referenced, the next address is highly probable to be accessed within a short period of time.
**Temporal Locality** refers to the fact that once a particular data item is accessed, it is likely that it will be referenced again within a short period of time.
To exploit these two concepts, the data is transferred in blocks between cache and the main memory. For a request for data, if the data is available in the cache it results in a cache hit. And if the requested data is not present in the cache, it is called a cache miss. In the given example program segment, spatial locality is shown by the array ALPHA, in which next variable to be accessed is adjacent to the one accessed previously. Temporal locality is shown by the reuse of the loop variable 100 times in For loop instruction.

```
Int ALPHA [100], SUM;
SUM=0;
For (i=0; i<100; i++)
{SUM= SUM+ALPHA[i];}
```

**Cache Management**

To manage the working of the cache, cache control unit is implemented in hardware, which performs all the logic operations on the cache. As data is exchanged in blocks between main memory and cache, four important cache functions need to be defined.

- Block Placement Strategy
- Block Identification

- Block Replacement
- Write Strategy

**Block Diagram of a Cache System**

In the figure, the block diagram of a system using cache is shown. It consists of two components.

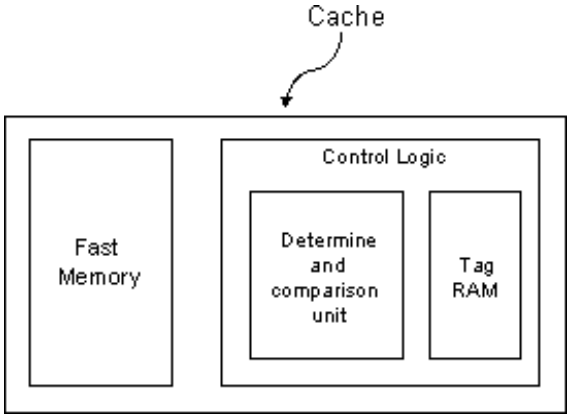
- Fast Memory
- Control Logic Uni

Control logic is further divided into two parts.

**Determine and Comparison Unit:** For determining and comparisons of the different parts of the address and to evaluate hit or miss.

**Tag RAM**: Second part consists of tag memory which stores the part of the memory address (called tag) of the information (block) placed in the data cache. It also contains additional bits used by the cache management logic.

**Data Cache:** is a block of fast memory which stores the copies of data and instructions frequently accessed by the CPU.




## Cache Strategies

In the next section we will discuss various cache functions, and strategies used to implement these functions.

**Block Placement**

Block placement strategy needs to be defined to specify where blocks from main memory will be placed in the cache and how to place the blocks. Now various methods can be used to map main memory blocks onto the cache .One of these methods is the associative mapping explained below.

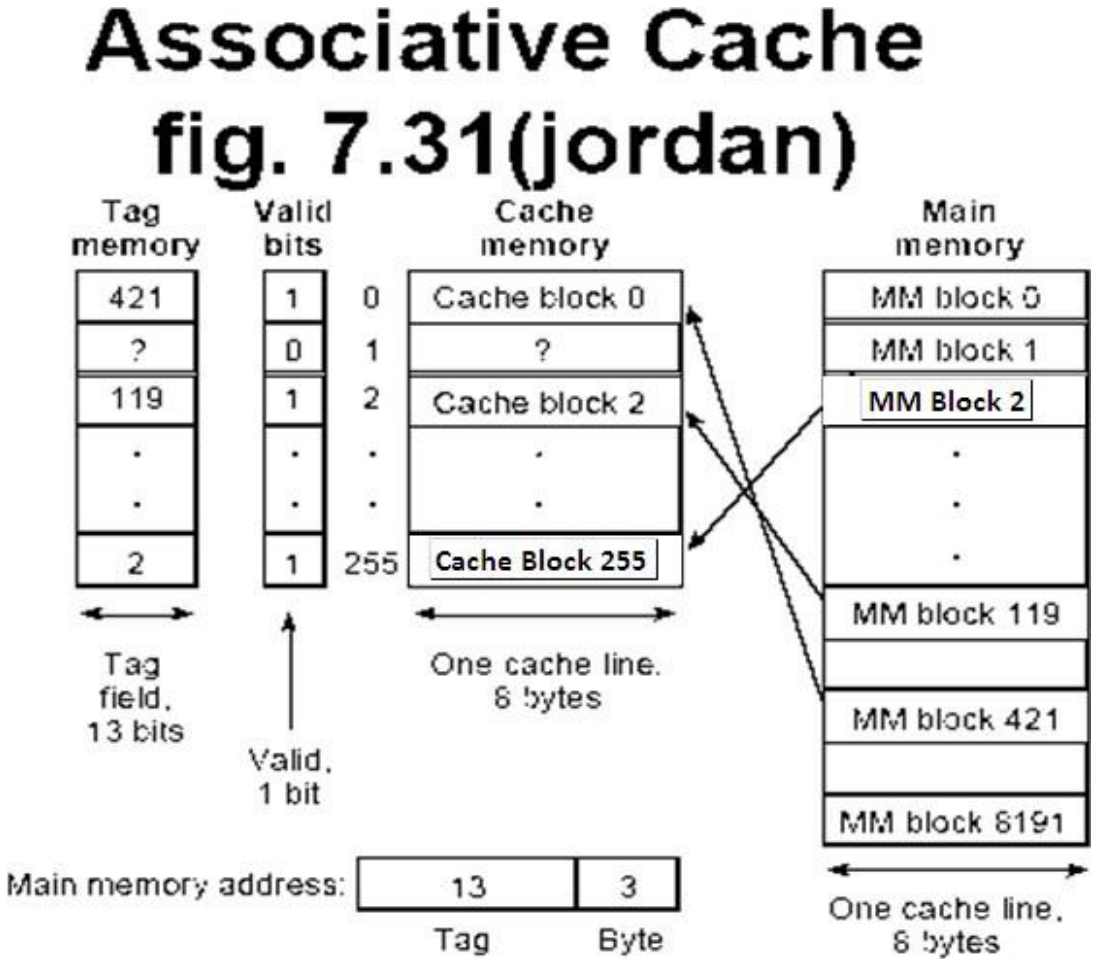
**Associative Mapping**

In this technique, block of data from main memory can be placed at any location in the cache memory. A given block in cache is identified uniquely by its main memory block

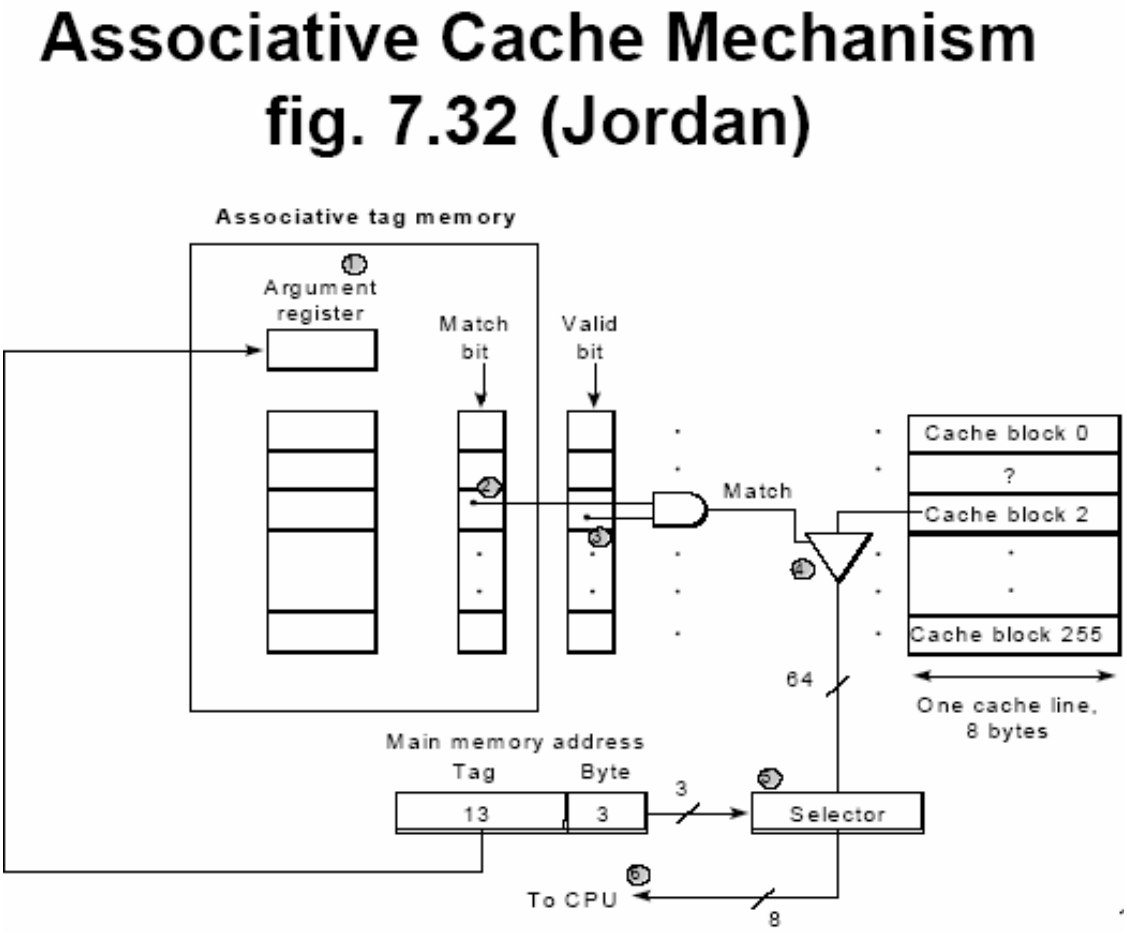
number, referred to as a tag, which is stored inside a separate tag memory in the cache. To check the validity of the cache blocks, a valid bit is stored for each cache entry, to verify whether the information in the corresponding block is valid or not.
Main memory address references have two fields.

- The word field becomes a "cache address" which specifies where to find the word in the cache.
- The tag field which must be compared against every tag in the tag memory.

**Associative Mapping Example**
Refer to Book Ch.7 Section (7.5) Figure 7.31(page 350-351) for detailed explanation.




**Mechanism of the Associative Cache Operation**
For details refer to book Ch.7, Section 7.5, Figure 7.32 (Page 351-352).




**Direct Mapping**

In this technique, a particular block of data from main memory can be placed in only one location into the cache memory. It relies on principle of locality.

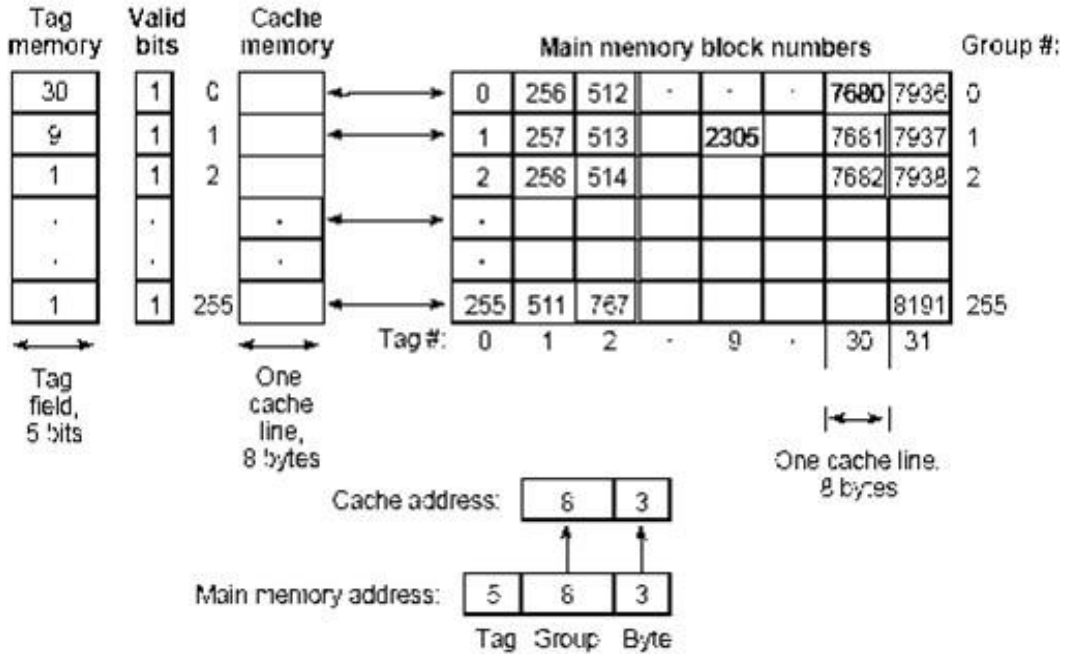
Cache address is composed of two fields:

- Group field
- Word field

Valid bit specifies that the information in the selected block is valid.

For a direct mapping example, refer to the book Ch.7, Section 7.5, Figure 7.33 (page 352 – 353).

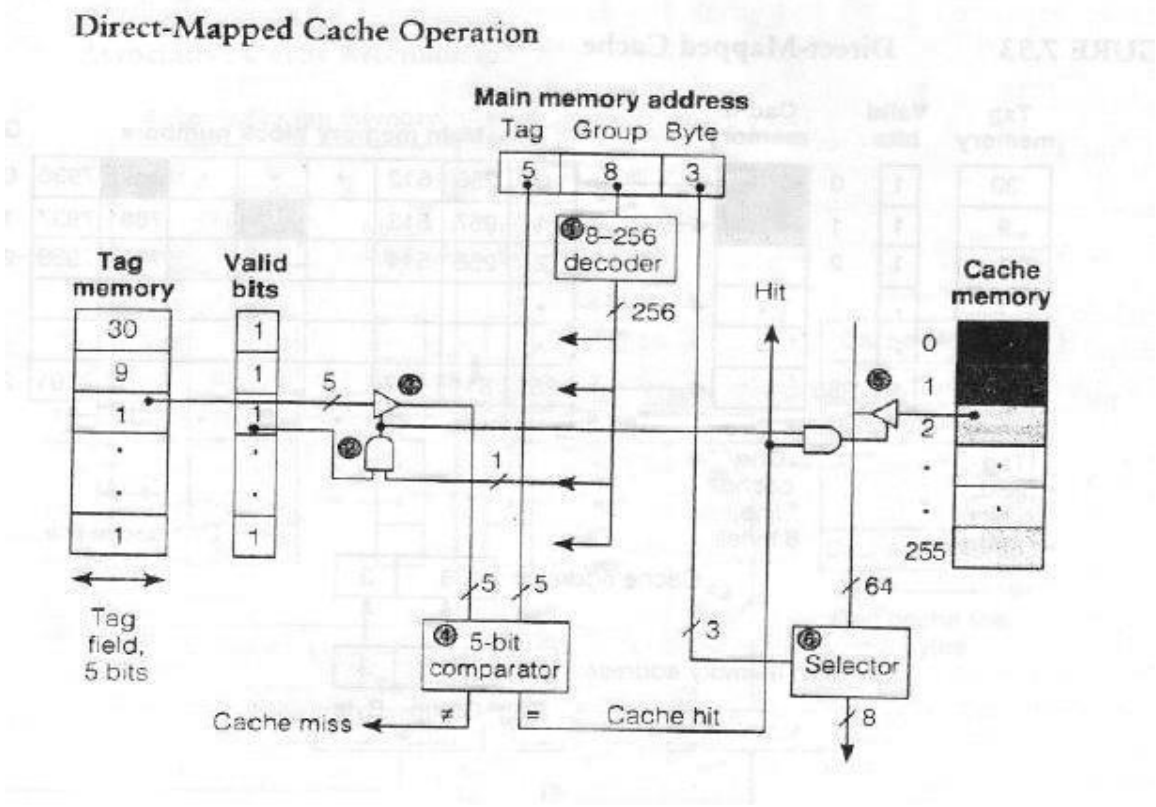
# Direct mapped cache
# fig. 7.33 (Jordan)




**Logic Implementation of the Controller for Direct Mapping**

Logic design for the direct mapping is simpler as compared to the associative mapping. Only one tag entry needs to be compared with the part of the address called group field.

**Tasks Required For Direct Mapping Cache:**
For details refer to the book Ch. 7, Section 7.5, Figure 7.34 (Page 353-354).




**Cache Design: Direct Mapped Cache**
To understand the principles of cache design, we will discuss an example of a direct mapped cache.
The size of the main memory is 1 MB. Therefore 20 address bits needs to be specified. Assume that the block size is 8 bytes. Cache memory is assumed to be 8 KB organized as 1 K lines of cache memory. Cache memory addresses will range from 0 up to 1023. Now we have to specify the number of bits required for the tag memory. The least significant three bits will define the block. The next 10 bits will define the number of bits required for the cache. The remaining 7 bits will be the width of the tag memory.
Main memory is organized in rectangular form in rows and columns. Number of rows would be from 0 up to 1023 defined by 10 bits. Number of rows in the main memory will be the same as number of lines in the cache. Number of columns will correspond to 7 bits address of the tag memory. Total number of columns will be 128 starting from 0 up to 127. With direct mapping, out of any particular row only one block could be mapped into the cache. Total number of cache entries will be 1024 each of 8 bytes.

**Advantage:**

Simplicity

**Disadvantage:**

Only a single block from a given group is present in cache at any time. Direct map Cache imposes a considerable amount of rigidity on cache organization.

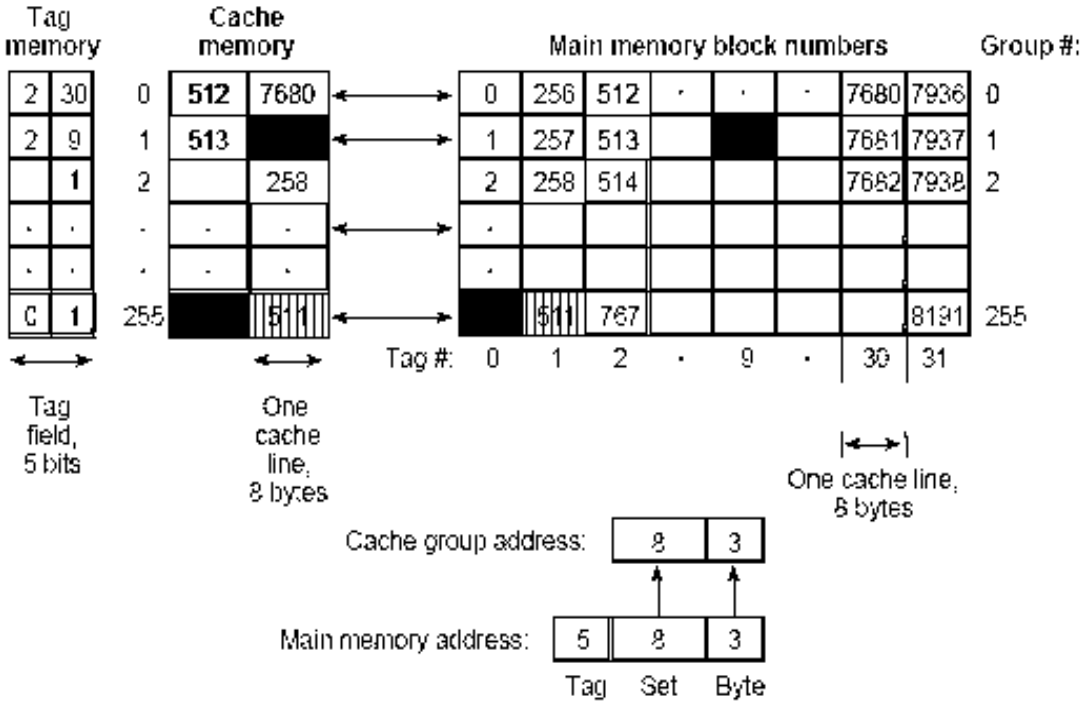
**Set Associative Mapping**

In this mapping scheme, a set consisting of more than one block can be placed in the cache memory.

The main memory address is d vided into two fields. The Set field is decoded to select the correct group. After that the tags in th selected groups are searched. Two possible places in which a block can reside must be searched associatively. Cache group address is the same as that of the direct-mapped cache.

For details of the Set associative mapping example, refer to the book Ch.7, Section 7.5, Figure 7.35 (Page 354-355).

# 2-Way Set-Associative Cache fig. 7.35(Jordan)




**Replacement Strategy**

For a cache miss, we have to replace a cache block with the data coming from main

memory. Different methods can be used to select a cache block for replacement.
**Always Replacement**: For Direct Mapping on a miss, there is only one block which needs replacement called always replacement.
For associative mapping, there are no unique blocks which need replacement .In this case there are two options to decide which block is to be replaced.

- **Random Replacement**: To randomly select the block to be replaced
- **LFU:** Based on the statistical results, the block which has been least used in the recent past, is replaced with a new block.

**Write Strategy**
When a CPU command to write to a memory data will come into cache, the writing into the cache requires writing into the main memory also.
**Write Through:** As the data is written into the cache, it is also written into the main memory called Write Through. The advantages are:

- Read misses never result in writes to the lower level.
- Easy to implement than write back

**Write Back:** Date resides in the cache, till we need to replace a particular block then the data of that particular block will be written into the memory if that needs a write, called write back. The advantages are:

- Write occurs at the speed of the cache
- Multiple writes with in the same block requires only one write to the lower memory.
- This strategy uses less memory bandwidth, since some writes do not go to the lower level; useful when using multi processors.

**Cache Coherence**
Multiple copies of the same data can exist in memory hierarchy simultaneously. The Cache needs updating mechanism to prevent old data values from being used. This is the problem of cache coherence. Write policy is the method used by the cache to deal with and keep the main memory updated.
**Dirty bit** is a status bit which indicates whether the block in cache is dirty (it has been modified) or clean (not modified). If a block is clean, it is not written on a miss, since lower level contains the same information as the cache. This reduces the frequency of writing back the blocks on replacement.
Writing the cache is not as easy as reading from it e.g., modifying a block can not begin until the tag has been checked, to see if the address is a hit. Since tag checking can not occur in parallel with the write as is the case in read, therefore write takes longer time.
**Write Stalls**: For write to complete in Write through, the CPU has to wait. This wait state is called write stall.
**Write Buffer**: reduces the write stall by permitting the processor to continue as soon as the data has been written into the buffer, thus allowing overlapping of the instruction execution with the memory update.
**Write Strategy on a Cache Miss**

On a cache miss, there are two options for writing.
**Write Allocate**: The block is loaded followed by the write. This action is similar to the read miss. It is used in write back caches, since subsequent writes to that particular block will be captured by the cache.
**No Write Allocate**: The block is modified in the lower level and not loaded into the cache. This method is generally used in write through caches, because subsequent writes to that block still have to go to the lower level.

# Lecture No. 40

## Virtual Memory

**Reading Material**

Vincent P. Heuring & Harry F. Jordan — Chapter 7
Computer Systems Design and Architecture — 7.6

**Summary**

- Virtual Memory
- Virtual Memory Organization

**Virtual Memory**

**Introduction**

Virtual memory acts as a cache between main memory and secondary memory. Data is fetched in advance from the secondary memory (hard disk) into the main memory so that data is already available in the main m mory when needed. The benefit is that the large access delays in reading data from hard disk are avoided.

Pages are formulated in the secondary memory and brought into the main memory. This process is managed both in hardware (Memory Management Unit) and the software (The operating systems is responsible for managing the memory resources).

The block diagram shown (Book Ch.7, Section 7.6, and figure 7.37) specifies how the data interchange takes place between cache, main memory and the disk. The Memory Management unit (MMU) is located between the CPU and the physical memory. Each memory reference issued by the CPU is translated from the logical address space to the physical address space, guided by operating system controlled mapping tables. As address translation is done for each memory reference, it must be performed by the hardware to speed up the process. The operating system is invoked to update the associated mapping tables.

**Memory Management and Address Translation**

The CPU generates the logical address. During program execution, effective address is generated which is an input to the MM , whic generates the virtual address. The v rtual address is divided into two fields. First field represents the page number and the second field is the word field. In the next step, the MMU translates the virtual address into the physical address which indicates the location in the physical memory.

**Advantages of Virtual Memory**

- Simplified addressing scheme: the programmer does not need to bother about the exact locations of variables/instructions in the physical memory. It is taken care of by the operating system.
- For a programmer, a large virtual memory will be available, even for a limited physical memory.
- Simplified access control.

## Virtual Memory Organization

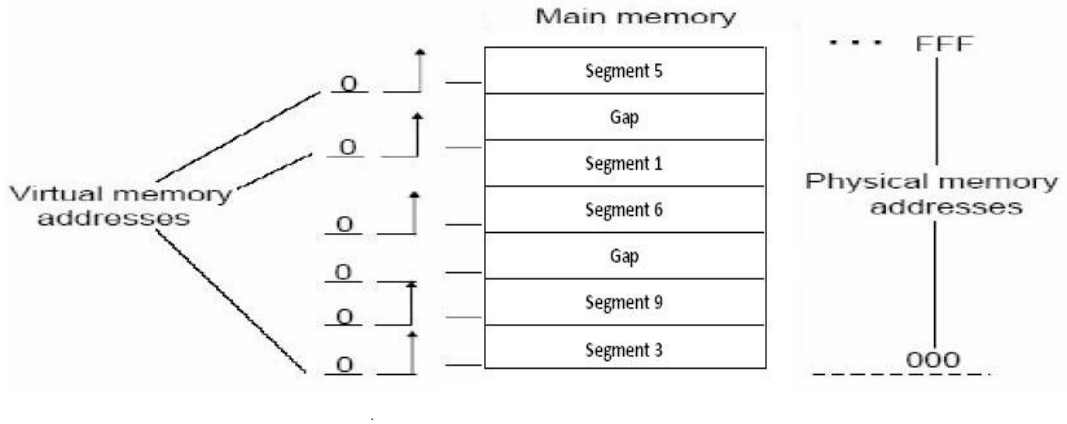
Virtual memory can be organized in different ways. This first scheme is segmentation.

**Segmentation:**

In segmentation, memory is divided into segments of variable sizes depending upon the requirements. Main memory segments identified by segments numbers, start at virtual address 0, regardless of where they are located in physical memory.

In pure segmented systems, segments are brought into the main memory from the secondary memory when needed. If segments are modified and not required any more, they are sent back to secondary memory. This invariably results in gap between segments, called external fragmentation i.e. less efficient use of memory. Also refer to Book Ch.7 , Section 7.6, Figure 7.38.

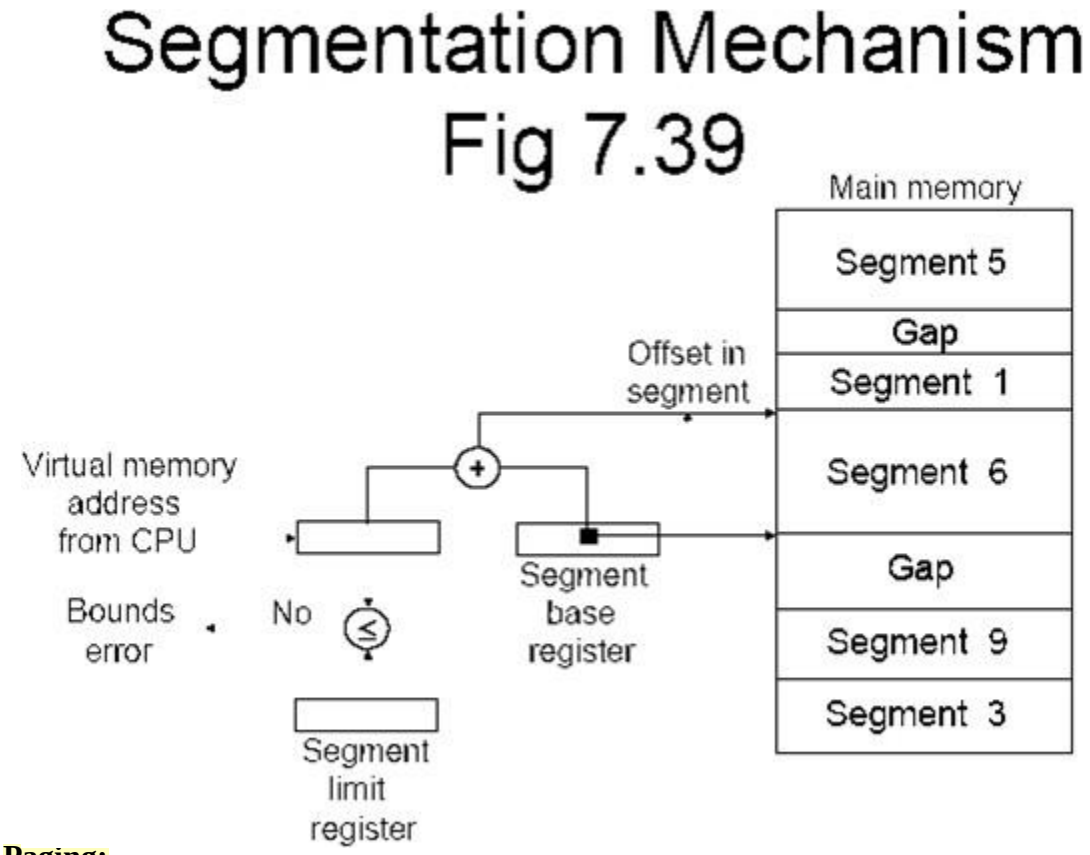
Memory Management by Segmentation
Fig 7.38




**Addressing of Segmented Memory**

The physical address is formed by adding each virtual address issued by the CPU to the contents of the segment base register in the MMU. Virtual address may also be compared with the segment limit register to keep track and avoiding the references beyond the specified limit. By maintaining table of segment base and limit registers, operating system can switch processes by switching the contents of the segment base and limit register. This concept is used in multiprogramming. Refer to book Ch.7, Section 7.6, and

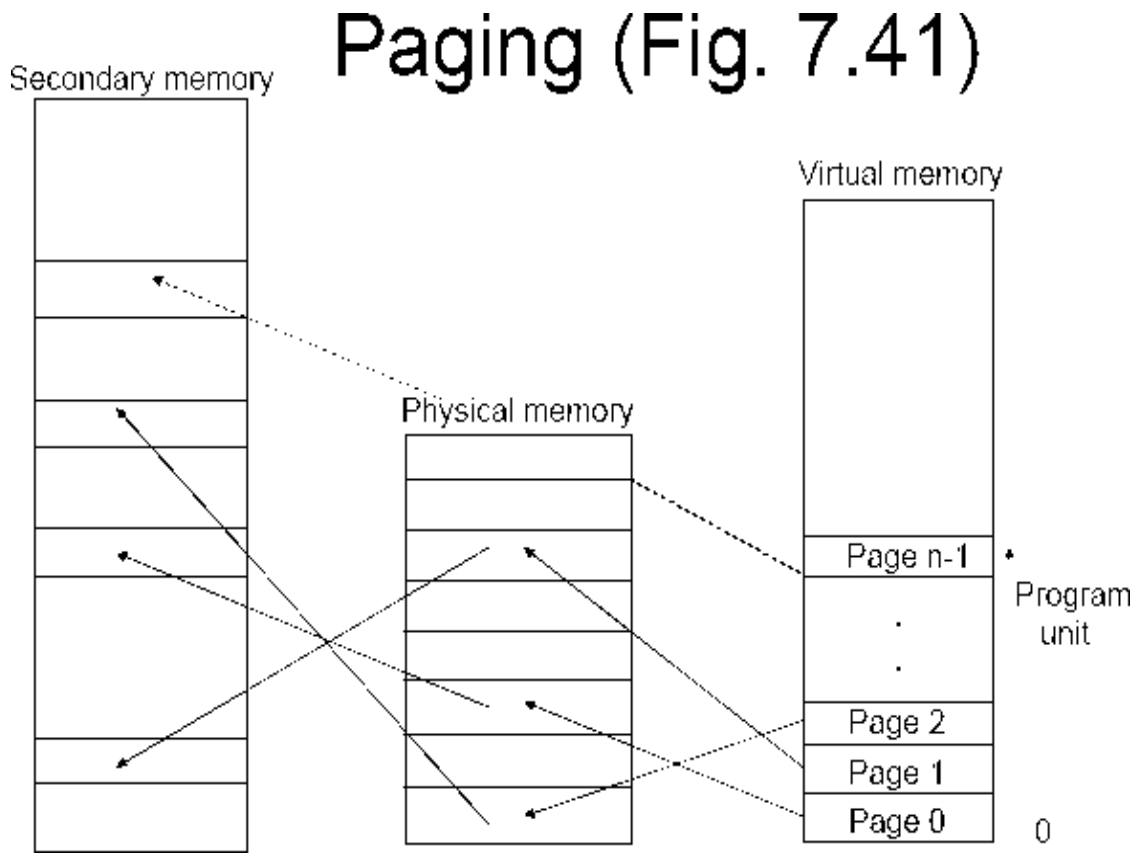
Figure 7.39




**Paging:**

In this scheme, we have pages of fixed size. In demand paging, pages are available in secondary memory and are brought into the main memory when needed.

Virtual addresses are formed by concatenating the page number with the word number. The MMU maps these pages to the pages in the physical memory and if not present in the physical memory, to the secondary memory. (Refer to Book Ch.7, Section 7.6, and Figure 7.41)




**Page Size**: A very large page size results in increased access time. If page size is small, it may result in a large number of accesses.
The main memory address is divided into 2 parts.

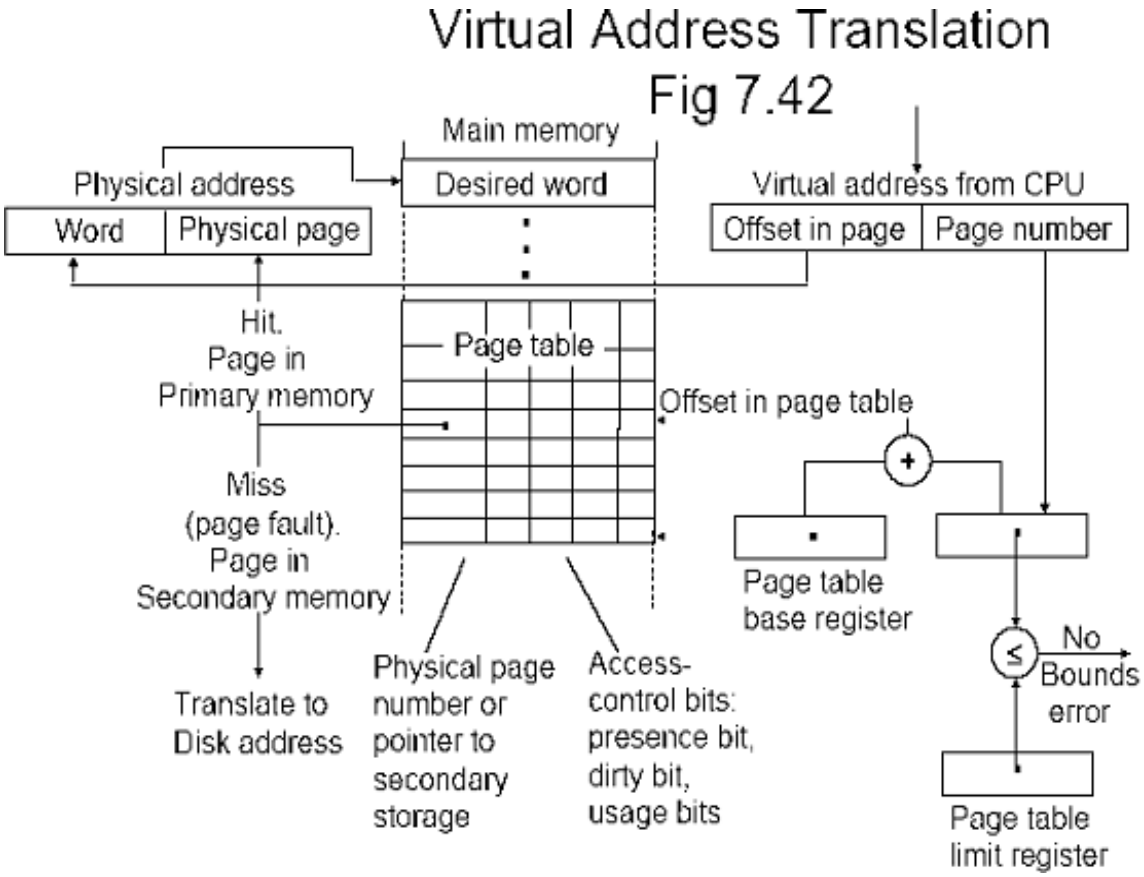
- Page number: For virtual address, it is called virtual page number.
- Word Fiel

**Virtual Address Translation in a Paged MMU:**
Virtual address composed of a page number and a word number, is applied to the MMU. The virtual page number is limit checked to verify its availability within the limits given in the table. If it is available, it is added to the page table base address which results in a page table entry. If there is a limit check fault, a bound exception is raised as an interrupt to the processor.

**Pa e Table**
The page table entry for each page has two fields.

- Page field
- Control Field: This includes the following bits.
  - Access control bits: These bits are used to specify read/write, and execute permissions.
  - Presence bits: Indicates the availability of page in the main memory.
  - Used bits: These bits are set upon a read/ write.

If the presence bit indicates a hit, then the page field of the page table entry contains the physical page number. It is concatenated with the word field of the virtual address to form a physical address.
Page fault occurs when a miss is indicated by the presence bit. In this case, the page field of the page table entry would contain the address of the page in the secondary memory. Page miss results in an interrupt to the processor. The requesting process is suspended until the page is brought in the main memory by the interrupt service routine.
Dirty bit is set on a write hit CPU operation. And a write miss CPU operation causes the MMU to begin a write allocate (previously discussed) process. (Refer to book Ch.7, Section 7.6, and Figure 7.42)




**Fragmentation:**
Paging scheme results in unavoidable internal fragmentations i.e. some pages (mostly last pages of each process) may not be fully used. This results in wastage of memory.

**Processor Dispatch -Multiprogramming**
Consider the case, when a number of tasks are waiting for the CPU attention in a multiprogramming, shared memory environment. And a page fault occurs. Servicing the page fault involves these steps.

1. Save the state of suspended process

2. Handle page fault
3. Resume normal execution

**Scheduling:** If there are a number of memory interactions between main memory and secondary memory, a lot of CPU time is wasted in controlling these transfers and number of interrupts may occur.

To avoid this situation, Direct Memory Access (DMA) is a frequently used technique. The Direct memory access scheme results in direct link between main memory and secondary memory, and direct data transfer without attention of the CPU. But use of DMA in virtual memory may cause coherence problem. Multiple copies of the same page may reside in main memory and secondary memory. The operating system has to ensure that multiple copies are consistent.

**Page Replacement**

On a page miss (page fault), the needed page must be brought in the main memory from the secondary memory. If all the pages in the main memory are being used, we need to replace one of them to bring in the needed page. Two methods can be used for page replacement.

**Random Replacement**: Randomly replacing any older page to bring in the desired page.

**Least Frequently Used**: Maintain a log to see which particular page is least frequently used and to replace that page.

**Translation Lookaside buffer**

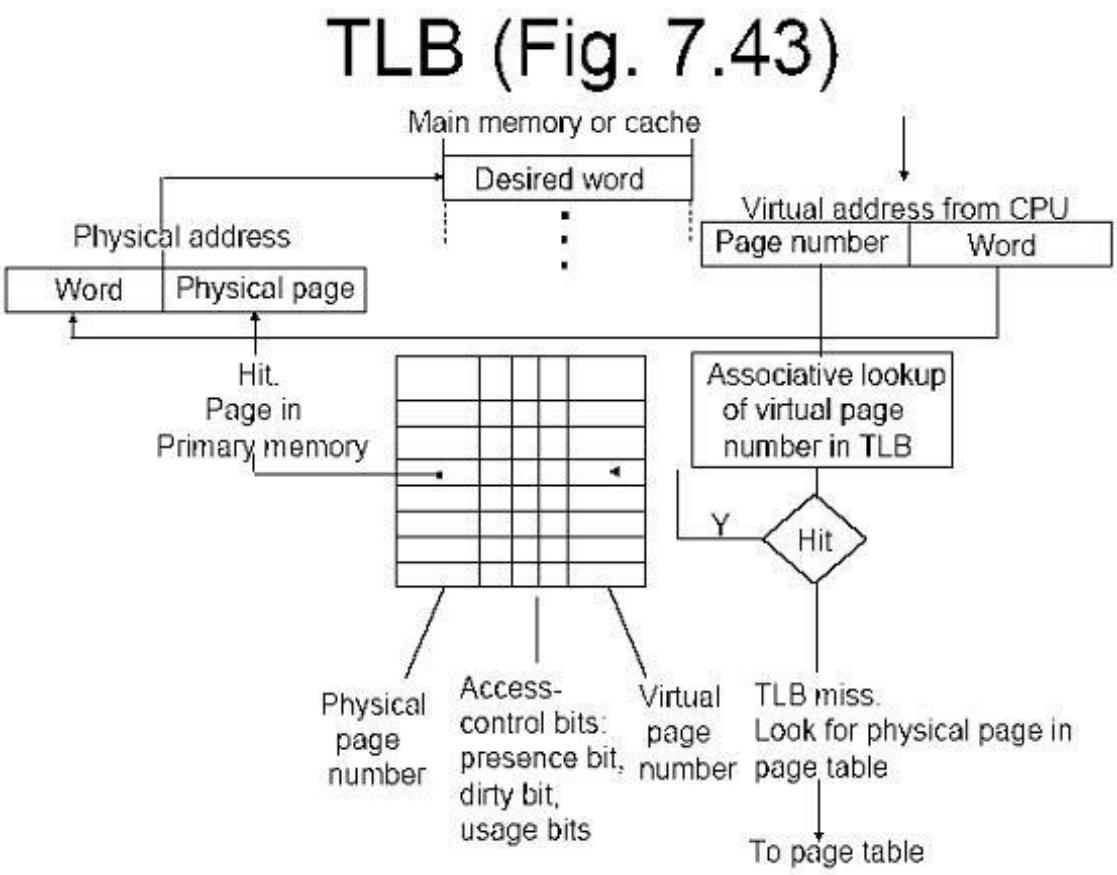
Identifying a particular page in the virtual memory requires page tables (might be very large) resulting in large memory space to implement these page tables. To speed up the process of virtual address translation, translation Lookaside buffer (TLB) is implemented as a small cache inside the CPU, which stores the most recent page table entry reference made in the MMU. It contents include

- A mapping from virtual to physical address
- Status bits i.e. valid bit, dirty bit, protection bit

It may be implemented using a fully associative organization

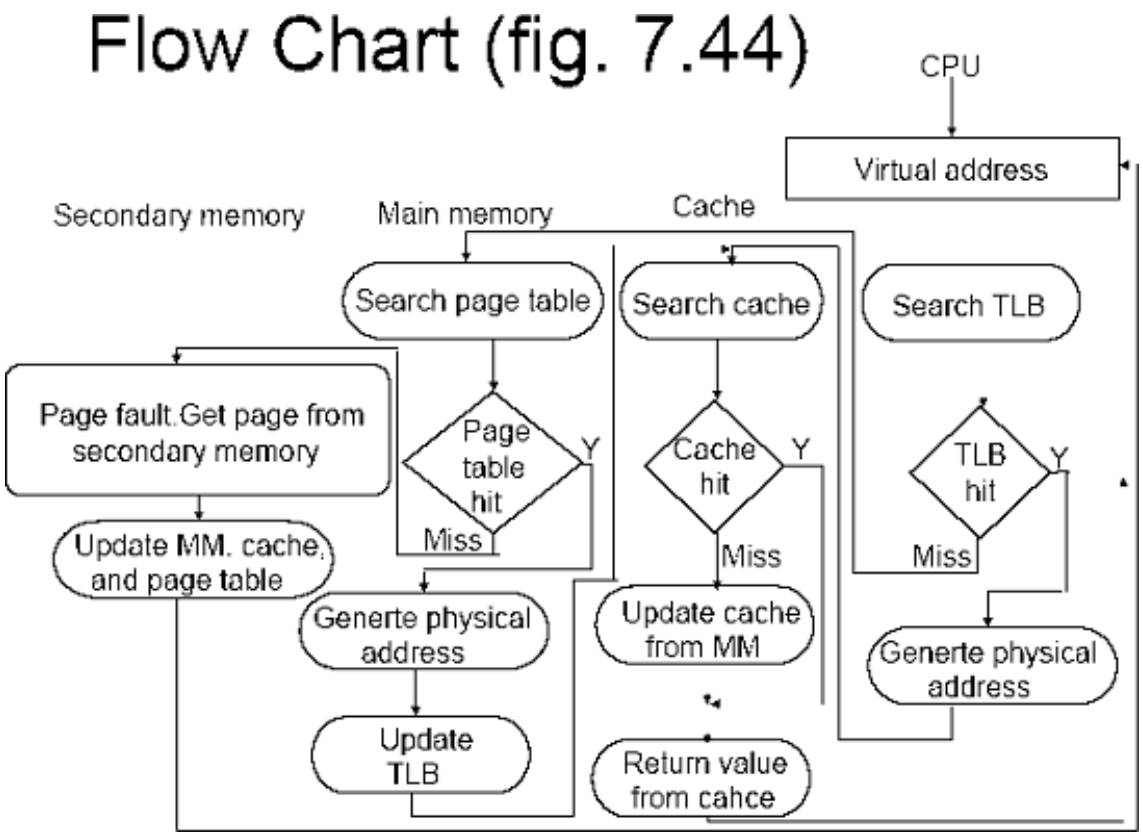
**Operation of TLB**

For each virtual address reference, the TLB is searched associatively to find a match between the virtual page number of the memory reference and the virtual page number in the TLB. If a match is found (TLB hit) and if the corresponding valid bit and access control bits are set, then the physical page mapped to the virtual page is concatenated. (Refer to Book Ch.7, Section 7.6, and Figure 7.43)




**Working of Memory Sub System**

When a virtual address is issued by the CPU, all components of the memory subsystem interact with each other. If the memory reference is a TLB hit, then the physical address is applied to the cache. On a cache hit, the data is accessed from the cache. Cache miss is processed as described previously. On a TLB miss (no match found) the page table is searched. On a page table hit, the physical address is generated, and TLB is updated and cache is searched. On a page table miss, desired page is accessed in the secondary memory, and main memory, cache and page table are updated. TLB is updated on the next access (cache access) to this virtual address. (Refer to Book Ch.7, Section 7.6, and Figure 7.44).




To reduce the work load on the CPU and to efficiently use the memory sub system, different methods can be used. One method is separate cache for data and instructions.

**Instruction Cache:** It can be implemented as a Translation Lookaside buffer.

**Data Cache:** In data cache, to access a particular table entry, it can be implemented as a TLB either in the main memory, cache or the CPU.

# Lecture No. 41

## Numerical Examples of DRAM and Cache

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

*Summary*

Numerical Examples related to

- DRAM
- Pipelining, Pre-charging and Parallelism
- Cache
- Hit Rate and Miss Rate
- Access Time

**Example 1**

If a DRAM has 512 rows and its refresh time is 9ms, what should be the frequency of row refresh operation on the average?

**Solution**
Refresh time= 9ms
Number of rows=512
Therefore we have to do 512 row refresh operations in a 9 ms interval, in other words one row refresh operation every $(9x10^{-3})/512 = 1.76x10^{-5}$seconds.

**Example 2**

Consider a DRAM with 1024 rows and a refresh time of 10ms.
a. Find the frequency of row refresh operations.
b. What fraction of the DRAM‟s time is spent on refreshing if each refresh takes 100ns.

**Solution**

Total number of rows = 1024

Refresh period = 10ms
One row refresh takes place after every
10ms/1024=9.7micro seconds
Each row refresh takes 100ns, so fraction of the DRAM‟s time taken by row refreshes is,
100ns/9.7 micro sec= 1.03%

**Example 3**

Consider a memory system having the following specifications. Find its total cost and cost per byte of memory.

| Memory type | Total bytes | Cost per byte |
|---|---|---|
| SRAM | 256 KB | 30$ per MB |
| DRAM | 128 MB | 1$ per MB |
| Disk | 1 GB | 10$ per GB |

**Solution**

Total cost of system
256 KB( ¼ MB) of SRAM costs = 30 x ¼ = $7.5
128 MB of DRAM costs= 1 x 128= $128
1 GB of disk space costs= 10 x 1=$10
Total cost of the memory system
= 7.5+128+10=$145.5
Cost per byte
Total storage= 256 KB + 128 MB + 1 GB
= 256 KB + 128x1024KB + 1x1024x1024KB
=1,179,904 KB
Total cost = $145.5
Cost per byte=145.5/(1,179,904x1024)
= $1.2x10$^{-7}$$/B

**Example 4**

Find the average access time of a level of memory hierarchy if the hit rate is 80%. The memory access takes 12ns on a hit and 100ns on a miss.

**Solution**

Hit rate =80%

Miss rate=20%
$T_{hit}$=12 ns
$T_{miss}$=100ns
Average $T_{access}$=(hit rate*$T_{hit}$)+(miss rate*$T_{miss}$)
=(0.8*12ns)+(0.2*100ns)
= 29.6ns

**Example 5**

Consider a memory system with a cache, a main memory and a virtual memory. The access times and hit rates are as shown in table. Find the average access time for the hierarchy.

| | Main memory | cache | virtual memory |
|---|---|---|---|
| Hit rate | 99% | 80% | 100% |
| Access time | 100ns | 5ns | 8ms |

**Solution**

Average access time for requests that reach the main memory
= (100ns*0.99)+(8ms*0.01)
= 80,099 ns
Average access time for requests that reach the cache
=(5ns*0.8)+(80,099ns*0.2)
=16,023.8ns

**Example 6**

Given the following memory hierarchy, find the average memory access time of the complete system

| Memory type | Average access time | Hit rate |
|---|---|---|
| SRAM | 5ns | 80 % |

| | | |
|---|---|---|
| DRAM | 60ns | 80% |
| Disk | 10ms | 100% |

**Solution**

For each level, average access time=( hit rate x access time for that level) + ((1-hit rate) x average access time for next level)
Average access time for the complete system
= (0.8x5ns) + 0.2 x((0.8x60ns) + (0.2)(1x10ms))
= 4 + 0.2(48+2000000)
=4 + 400009.6
= 400013.6 ns

**Example 7**

Find the bandwidth of a memory system that has a latency of 25ns, a pre charge time of 5ns and transfers 2 bytes of data per access.

**Solution**

Time between two memory references
=latency + pre charge time
= 25 ns+ 5ns
= 30ns
Throughput = 1/30ns
$=3.33x10^7$ operations/second
Bandwidth $= 2x\ 3.33x10^7$
$= 6.66x10^7$ bytes/s

**Example 8**

Consider a cache with 128 byte cache line or cache block size. How many cycles does it take to fetch a block from main memory if it takes 20 cycles to transfer two bytes of data?

**Solution**
The number of cycles required for the complete transfer of the block
=20 x 128/2
= 1280 cycles

Using large cache lines decreases the miss rate but it increases the amount of time a

program takes to execute as obvious from the number of clock cycles required to transfer a block of data into the cache.

**Example 9**

Find the number of cycles required to transfer the same 128 byte cache line if page-mode DRAM with a CAS-data delay of 8 cycles is used for main memory. Assume that the cache lines always lie within a single row of the DRAM, and each line lies in a different row than the last line fetched.

**Solution**

Memory requests to fetch each cache line=128/2= 64
Only the first fetch require the complete 20 cycles, and the other 63 will take only 8 clock cycles. Hence the no. of cycles required to fetch a cache line
=20 + 8 x 63
= 524

**Example 10**

Consider a 64KB direct-mapped cache with a line length of 32 bytes.

a. Determine the number of bits in the address that refer to the byte within a cache line.
b. Determine the number of bits in the address required to select the cache line.

**Solution**
Address breakdown

$n=\log_2$ of number of bytes in line
$m=\log_2$ of number of lines in cache

a. For the given cache, the number of bits in the address to determine the byte within the line= $n = \log_2 32 = 5$

b. There are 64K/32= 2048 lines in the given cache. The number of bits required to select the required line = $m = \log_2 2048 = 11$

Hence n=5 and m=11 for this example.

**Example 11**

Consider a 2-way set-associative cache with 64KB capacity and 16 byte lines.

a. How many sets are there in the cache?

b. How many bits of address are required to select a set in the cache?
c. Repeat the above two calculations for a 4-way set-associative cache with same size.

**Solution**

a. A 64KB cache with 16 byte lines contains 4096 lines of data. In a 2-way set associative cache, each set contains 2 lines, so there are 2048 sets in the cache.

b. $Log_2(2048)=11$. Hence 11 bits of the address are required to select the set.

c. The cache with 64KB capacity and 16 byte line has 4096 lines of data. For a 4-way set associative cache, each set contains 4 lines, so the number of sets in the cache would be 1024 and $Log_2(1024)=10$. Therefore 10 bits of the address are required to select a set in the cache.

**Example 12**
Consider a processor with clock cycle per instruction (CPI) = 1.0 when all memory accesses hit in the cache. The only data accesses are loads and stores, and these constitute 60% of all the instructions. If the miss penalty is 30 clock cycles and the miss rate is 1.5%, how much faster would the processor be if all instructions were cache hits?

**Solution**

Without any misses, the computer performance is
CPU execution time = (CPU clock cycles + Memory stall cycles) x Clock cycle
=(IC x CPI+ 0)x Clock cycle = IC x 1.0 x Clock cycle
Now for the computer with the real cache, first we compute the number of memory stall cycles:
Memory accesses = IC x Instruction x Miss Rate x Miss Penalty
Memory stall cycles

= IC x (l + 0.6) x 0.015 x 30
= IC x 0.72

where the middle term (1 + 0.6) represents one instruction access and 0.6 data accesses per instruction. The total performance is thus

CPU execution time cache = (IC x 1.0 + IC x 0.72) x Clock cycle
= 1.72 x IC x Clock cycles

The performance ratio is the inverse of the execution times

CPU execution time cache = 1.72 x IC x clock cycle
CPU execution time 1.0 x IC x clock cycle

The computer with no cache misses is 1.72 times faster

**Example 13**

Consider the above example but this time assume a miss rate of 20 per 1000 instructions. What is memory stall time in terms of instruction count?

**Solution**

Re computing the memory stall cycles:
Memory stall cycles=Number of misses x Miss penalty
=IC * Misses * Miss penalty
Instruction

=IC / 1000 * Misses * Miss penalty
Instruction * 1000
=IC / 1000 * 20 * 30
= IC /1000 * 600= IC * 0.6

**Example 14**

What happens on a write miss?

**Solution**

The two options to handle a write miss are as follows:
**Write Allocate**
The block is allocated on a write miss, followed by the write hit actions. This is just like read miss.
**No-Write Allocate**
Here write misses do not affect the cache. The block is modified only in the lower level memory.

**Example 15**

Assume a fully associative write-back cache with many cache entries that starts empty. Below is a sequence of five memory operations (the address is in square brackets):

Write Mem[300];

Write Mem[300];
Read Mem[400];
Write Mem[400];
WriteMem[300];

What is the number of hits and misses when using no-write allocate versus write allocate?

**Solution**

For no-write allocate, the address 300 is not in the cache, and there is *no* allocation on write, so the first two writes will result in misses. Address 400 is also not in the cache, so the read is also a miss. The subsequent write to address 400 is a hit. The last write to 300 is still a miss. The result for no-write allocate is four misses and one hit.
For write allocate, the first accesses to 300 and 400 are misses, and the rest are hits since 300 and 400 are both found in the cache. Thus, the result for write allocate is two misses and three hits.

**Example 16**

Which has the lower miss rate?
a 32 KB instruction cache with a 32 KB data cache or a 64 KB unified cache?
Use the following Miss per 1000 instructions.

| size | Instruction cache | Data cache | Unified cache |
|---|---|---|---|
| 32 KB | 1.5 | 40 | 42.2 |
| 64 KB | 0.7 | 38.5 | 41.2 |

**Assumptions**

- The percentage of instruction references is about 75%.
- Assume 40% of the instructions are data transfer instructions.
- Assume a hit takes 1 clock cycle.
- The miss penalty is 100 clock cycles.
- A load or store hit takes 1 extra clock cycle on a unified cache if there is only one cache port to satisfy two simultaneous requests.
- Also the unified cache might lead to a structural hazard.
- Assume write-through caches with a write buffer and ignore stalls due to the write buffer.

What is the average memory access time in each case?
**Solution**

First let's convert misses per 1000 instructions into
miss rates.

$$\text{Miss rate} = \frac{\frac{\text{Misses}}{\text{1000 Instructions}}}{\frac{\text{Memory accesses}}{\text{Instruction}}}$$

Since every instruction access has exactly one memory access to fetch the instruction, the instruction miss rate is

$$\text{Miss rate32 KB instruction} = \frac{1.5/1000}{1.00} = 0.0015$$

Since 40% of the instructions are data transfers, the data miss rate is

$$\text{Miss Rate 32 kb data} = \frac{40/1000}{0.4} = 0.1$$

The unified miss rate needs to account for instruction and data accesses:

$$\text{Miss Rate 64 kb unified} = \frac{42.2/1000}{1.00+0.4} = 0.031$$

As stated above, about 75% of the memory accesses are instruction references. Thus, the overall miss rate for the split caches is
(75% x 0.0015) + (25% x 0.1) = 0.026125
Thus, a 64 KB unified cache has a slightly lower effective miss rate than two 16 KB caches. The average memory access time formula can be divided into instruction and data accesses:
Average memory access time
= % instructions x (Hit time + Instruction miss rate x Miss Penalty) + % data x (Hit time + Data miss rate x Miss Penalty)

Therefore, the time for each organization is:

Average memory access time split
= 75%x(l +0.0015x 100) + 25%x(l +0.1x100)
= (75% x 1.15) + (25% x 11)
= 0.8625+2.75= 3.61
Average memory access time unified
= 75% x (1+0.031 x 100) +25% x (1 + 1+0.031 x 100)
= (75% x 4.1) + (25% x 5.1) = 3.075+1.275
= 4.35
Hence split caches have a better average memory access time despite having a worse effective miss rate. Split cache also avoids the problem of structural hazard present in a unified cache.

# Lecture No. 42

## Performance of I/O Subsystems

**Reading Material**

Patterson, D.A. and Hennessy, J.L. Chapter 8
Computer Architecture -A Quantitative Approach
*Summary*

- Introduction
- Performance of I/O Subsystems
- Loss System
- Single Server Model
- Little‟s Law
- Server Utilization
- Poisson distribution
- Benchmarks programs
- Asynchronous I/O and operating system

**Introduction**

Consider a producer-server model. A buffer (or queue) is present between them. Tasks are being received and when one task is finished (i.e. served) then the second task is taken up by the server. Now latency and the response time depend upon how many tasks are present in the queue and how quickly they are served. If there is no task, ahead in the queue the latency would be low and response time would be shorter.
Through put depends upon the average number of calls and the service time taken by a particular server.

**Performance of I/O Subsystems**

There are three methods to measure I/O subsystem performance:

- **Straight away calculations using execution time**
- **Simulation**
- **Queuing Theory**

**Loss System**

Loss system is a simple system having no buffer so it does not have any provision for the

queuing. In a loss system, provision is time in term of how many switches we do need, then provide some redundancy how many individuals I/O controllers we do need, then how many CPUs are there. It is also called dimension of a loss system.

**Delay System**

This system provides additional facilities. If we find some call party busy, we can have provision of call waiting. If we have more than one call waiting, then once we finish the first call, we may receive the second call.

**Single Server Model**

Consider a black box. Suppose it represents an I/O controller. At the input, we have arrival of different tasks. As one task is done, we have a departure at the output. So in the black box, we have a server. Now if we expand and open-up the black box, we could see that incoming calls are coming into the buffer and the output of the buffer is connected to the server. This is an example of "single server model".

**Little"s Law**

For a system with multiple independent requests for I/O service and input rate equal to output rate, we use Little"s law to find the mean number of tasks in the system and Time sys such that

Mean number of tasks = Arrival Rate x Mean Response time
and
$Time_{sys} = Time_q + Time_s$
where
$Time_s$ = Average time to serve task
$Time_q$ = Average time per task in the queue
$Time_{sys}$ = Aver time /task
Arrival Rate = $\lambda$ = Average number of arriving tasks
$Length_s$ = Average number of task in service
$Length_q$ = Average length of queue
and
$Length_{sys} = Length_q + Length_s$

**Server Utilization**

Server Utilization = Arrival Rate x $Time_q$

Server utilization is also called traffic intensity and its value must be between 0 and 1. Server utilization depends upon two parameters:

1. Arrival Rate
2. Average time required to serve each task

So, we can say that it depends on the I/O bandwidth and arrival rate of calls into the system.

**Example 1**

Suppose an I/O system with a single disk gets (on average) 100 I/O requests/second. Assume that average time for a disk to service an I/O request is 5ms. What is the utilization of the I/O system?

**Solution**

Time for an I/O request = 5ms
=0.005sec
Server utilization = 100 x 0.005
= 0.5

**Poisson distribution**

In order to calculate the response time of an I/O system, we make the following assumptions:

1. Arrival is random
2. System is memory less. It means that incoming calls are not correlated.

For characterize random events, according to above two assumptions, we use Poisson distribution:

Probability (k)= $(e^{-k} \times a^{k}) / k!$

a= Rate of events x Elapsed time
= Arrival rate x t

also

$$C^2 = \frac{\text{Variance}}{(\text{Arithmetic mean time})^2}$$

and

Average Residual Service Time = ½ x weighted mean time x $(1+C^2)$

**Example 2**

For the system of previous example having server utilization of 0.5, what is the mean number of I/O requests in the queue?

**Solution**

$$\text{Length}_q = \frac{(\text{Server utilization})^2}{(1- \text{Server utilization})}$$

$\text{Length}_q = (0.5)^2 / (1-0.5) = 0.5$

**Assumptions about Queuing Model**

1. Poisson distribution is assumed
2. The system is in equilibrium
3. The length of the queue is infinity
4. The system has only one server
5. The server will start the next task after finishing the previous one.

**Example 3**

Suppose a processor sends 10 disks I/O per second, these requests are exponentially distributed, and the average service time of an older disk is 10ms. Answer the following questions:

- What is the number of requests in the queue?
- What is the average time a spent in the queue?
- What is the average response time for a disk request?

**Solution**

Average number of arriving tasks/second = 20
Average disk time = 10ms = 0.01sec
Sever utilization = 20 x 0.01=0.2
$Time_q$ = 10ms x 0.2/(1-0.2) = 2.5ms
Average response time = 2.5+10=22.5ms

**M/M/m model of queuing theory**
A system which has multiple servers is called M/M/m model.
The following formulas are used for M/M/m model:

$$\text{Utilization} = \frac{\text{Arrival Rate} \times Time_s}{N_s}$$

$$Length_s = \text{Arrival Rate} \times Time_q$$

$$Time_q = \frac{(Time_s \times (P_{tasks} >= N_s))}{N_s \times (1 - \text{utilization})}$$

$$Prob_{tasks} >= N_s = \frac{N_s \times \text{utilization}}{N_s! \times (1-\text{utilization})} \times Prob_{0tasks}$$

**Example#4**

Suppose instead of a new, faster disk, we add a second slow disk, and duplicate the data so that read can be serviced by either disk. Let"s assume that the requests are all reads. Recalculate the answers to the earlier questions, this time using an M/M/m queue.

**Solution**

The average utilization of the two disks is given as;

$$\text{Server utilization} = \frac{\text{Arrival rate x Time}_s}{N_s}$$

$$= (20 \times 0.01) / 2$$

$$= 0.1$$

$$\text{Prob}_{0\text{tasks}} = \left[ 1 + \frac{(2 \times \text{utilization})^2}{2! \times (1 - \text{utilization})} + \frac{(2 \times \text{utilization})^n}{n!} \right]^{-1}$$

$$\text{Prob}_{0\text{tasks}} = \left[ 1 + \frac{(2x\ 0.1)^2}{2! \times (1 - 0.1)} + (2 \times 0.1) \right]^{-1}$$

$$= (1 + .022 + 0.2)^{-1}$$

$$= 1.222^{-1}$$

$$\text{Prob}_{\text{tasks}} >= N_s = \frac{(2 \times \text{utilization})^2}{2! \times (1 - \text{utilization})} \times \text{Prob}_{0\text{tasks}}$$

$$= \frac{(2x\ 0.1)^2}{2! \times (1 - 0.1)} 1.222^{-1}$$

$$= 0.018$$

$$\text{Time}_q = \text{Time}_s \times \frac{\text{Prob}_{\text{tasks}} >= N_s}{N_s \times (1 - \text{utilization})}$$

$$= 0.01 \times 0.018 / (2 \times 0.9)$$

$$= 0.1\text{msec}$$

Average response time = 10msec + 0.1msec
= 10.01msec

**Benchmarks programs**

In order to measure the performance of real systems and to collect the values of parameters needed for prediction, Benchmark programs are used.

**Types of Benchmark programs**

Two types of benchmark programs are used:

TPC-C

SPEC

**Asynchronous I/O and operating system**

In order to improve the I/O performance, parallelism is used.

For this, two approaches are available:

- Synchronous I/O
- Asynchronous I/O

**Synchronous I/O**

In this approach, operating system requests data and switches to another process. Until the desire data arrived. Then the operating system switches back to the requesting process.

**Asynchronous I/O**

This model is of the process to continue after making a request and it is not blocked until it tries to read requested data.

**Bus versus switches**

Consider a LAN, using bus topology. If we replace the bus with a switch, the speed of the data transfer will be improved to a great extent.

# Lecture No. 43

## Networks

**Reading Material**

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Patterson, D.A. and Hennessy, J.L. Chapter 8
Computer Architecture - A Quantitative Approach

**Summary**

- Introduction to computer network
- Difference between distributed computing and computer networks
- Classification of networks
- Interconnectivity in WAN
- Performance Issues
- Effective bandwidth versus Message size
- Physical Media

**Introduction to Computer Networks**

A computer architect should know about computer networks because of the two main reasons:

**1. Connectivity**

Connection of components with in a single computer follows the same principles used for the connection of different computers. It is important for the computer architect to know about connectivity for better sharing of bandwidth

**Sharing of resources**

Consider a lab with 50 computers and 2 printers using a network, all these 50 computers can share these 2 printers.

**Protocol**

A set of rules followed by different components in a network. These rules may be defined for hardware and software.

**Host**

It is a computer with a modem, LAN card and other network interfaces. Hosts are also

called nodes or end points. Each node is a combination of hardware and software and all nodes are interconnected by means of some physical media.

**Difference between Distributed Computing and Computer Networks**

In distributed computing, all elements which are interconnected operate under one operating system. To a user, it appears as a virtual uni-processor system.
In a computer network, the user has to specify and log in on a specific machine. Each machine on the network has a specific address. Different machines communicate by using the network which exists among them.

**Classification of Networks**
We can classify a network based on the following two parameters:

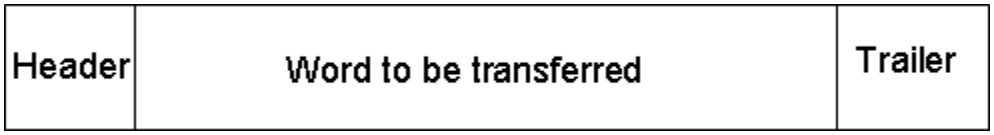
- The number and type of machines to be interconnected
- The distance between these machines

Based on these two parameters, we have the following type of networks:
**SAN (System/Storage Area Network)**
It refers to a cluster of machines where large disk arrays are present. Typical distances could be tens of meters.
**LAN (Local Area Network)**
It refers to the interconnection of machines in a building or a campus. Distances could be in Kilometers.
**WAN (Wide Area Network)**
It refers to the interconnection between LANs.

**Interconnectivity in WAN**

Two methods are used to interconnect WANs:

1. **Circuit switching**
   It is normally used in a telephone exchange. It is not an efficient way.
2. **Packet switching**
   A block (an appropriate number of bits) of data is called a packet. Transfer of data in the form packets through different paths in a network is called packet switching. Additional bits are usually associated with each packet. These bits contain information about the packet. These additional bits are of two types: header and trailer. As an example, a packet may have the form shown below:




If we use a 1- bit header, we may have the following protocol:
Header = 0, it means it is a request

Header = 0, Reply
By reading these header bits, a machine becomes able to receive data or supply data.
To transfer data by using packets through hardware is very difficult. So all the transfer is done by using software. By using more number of bits, in a header, we can send more messages. For example if n bits are used as header then $2^n$ is the number of messages that can be transmitted over a network by using a single header.
For a 2 bit header: we may have 4 types of messages:

00= Request
01= Reply
10= Acknowledge request
11= Acknowledge reply

**Error detection**

The trailer can be used for error detection. In the above example, a 4 bit checksum can be used to detect any error in the packet. The errors in the message could be due to the long distance transmission. If the error is found in some message, then this message will be repeated. For a reliable data transmission, bit error rate should be minimum.

**Software steps for sending a message:**

- Copy data to the operating system buffer.
- Calculate the checksum, include in trailer and star timer.
- Send data to the hardware for transmission.

**Software steps for message reception:**

- Copy data to the operating system buffer.
- Calculate the checksum; if same, send acknowledge and copy data to the user area otherwise discard the message.

**Response of the sender to acknowledgment:**

- If acknowledgment arrives, release copy of message from the system buffer.
- When timer expires, resend data and restart the time.

**Performance Issues**

1. **Bandwidth**
   It is the maximum rate at which data could be transmitted through networks. It is measured in bits/sec.
2. **Latency**
   In a LAN, latency (or delay) is very low, but in a WAN, it is significant and this is due to the switches, routers and other components in the network

3. **Time of flight**
   It is the time for first bit of the message to arrive at the receiver including delays. Time of the flight increases as the distance between the two machines increases.
4. **Transmission time**
   The time for the message to pass through the network, not including the time of flight.
5. **Transport latency**
   Transport latency= time of flight + transmission time
6. **Sender overhead**
   It is the time for the processor to inject message in to the network.
7. **Receiver overhead**
   It is the time for the processor to pull the message from the network.
8. **Total latency**
   Total latency = Sender overhead + Time of flight + Message size/Bandwidth + Receiver overhead
9. **Effective bandwidth**
   Effective bandwidth = Message size/Actual Bandwidth
   Actual bandwidth may be larger than the effective bandwidth.

**Example#1**

Assume a network with a bandwidth of 1500Mbits/sec. It has a sending overhead of 100μsec and a receiving overhead of 120μsec. Assume two machines connected together. It is required to send a 15,000 byte message from one machine to the other (including header), and the message format allows 15, 00 bytes in a single message. Calculate the total latency to send the message from one machine to another assuming they are 20m apart (as in a SAN). Next, perform the same calculation but assume the machines are 700m apart (as in a LAN). Finally, assume they are 1000Km apart (as in a WAN).
Assume that signals propagate at 66% of the speed of light in a conductor, and that the speed of light is 300,000Km/sec.

**Solution**

By using the assumption, we get:

$$\text{Time of flight} = \frac{\text{Distance between two machines in Km}}{2/3 \times 300{,}000\text{Km/sec}}$$

Total Latency = Sender overhead + Time of flight + Message size/bandwidth
+ Receiver overhead

**For SAN**:

Total latency = 100µsec
+ (0.020Km/(2/3 x 300,000Km/sec))
+ 15,000bytes/ 1500Mbits/sec
+ 120µsec
= 100µsec + 0.1µsec + 80µsec + 120µsec
= 300.1µsec

**For LAN**

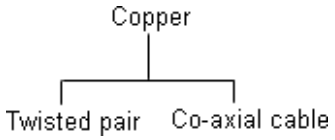
Total latency = 100µsec
+ (0.7Km/(2/3 x 300,000Km/sec))
+ 15,000bytes/ 1500Mbits/sec + 120µsec
= 100µsec + 3.5µsec + 80µsec + 120µsec
= 303.1µsec

**For WAN**

Total latency = 100µsec
+ (1000Km/(2/3 x 300,000Km/sec))
+ 15,000bytes/ 1500Mbits/sec
+ 120µsec
= 100µsec + 5000µsec + 80µsec + 120µsec
= 5300µsec

**Effective bandwidth versus Message size**
Effective bandwidth is always less than the raw bandwidth. If the effective bandwidth is closer to the raw bandwidth, the size of the message will be larger. If the message size is larger then network will be more effective.
If large number of the messages are present then a queue will be formed, and the user has to face delay. To minimize the delay, it is better to use packets of small size.

**Physical Media**




Twisted pair does not provide good quality of transmission and has less bandwidth. To get high performance and larger bandwidth, we use co-axial cable. For increased performance, better performance, we use fiber optic cables, which are usually made of glass. Data transmits through the fiber in the form of light pulses. Photo diodes and sensors are used to produce and receive electronic pulses.

# Lecture No. 44

## Communication Medium and Network Topologies

**Reading Material**

Patterson, D.A. and Hennessy, J.L. Chapter 8
Computer Architecture- A Quantitative Approach

***Summary***

- Physical Media (Continued)
- Shared Medium
- Switched Medium
- Connection Oriented vs. Connectionless Communication
- Network Topologies
- Seven-layer OSI Model
- Internet and Packet Switching
- Fragmentation
- Routing

**Modem**

To interconnect different computers by using twisted pair copper wire, an interface is used which is called modem. Modem stands for modulation/demodulation. Modems are very useful to utilize the telephone network (i.e. 4 KHz bandwidth) for data and voice transmission.

**Quality of Telephone Line**

Data transfer rate depends upon the quality of telephone line. If telephone line is of fine quality, then data transfer rate will be sufficiently high. If the phone line is noisy then data transfer rate will be decreased.

**Classification of Fiber Optic Cables**

Fiber optic cables can be classified into the following types.

**Multimode fiber**

This fiber has large diameter. When light is injected, it disperses, so the effective data rate decreases.

**Mono mode Fiber**

Its diameter is very small. So dispersion is small and data rate is very high.

**Wavelength –Division Multiplexing (WDM)**
Waves of different wavelengths are simultaneously sent through fiber. So as a result, throughput increases.

## Wireless Transmission

This is another effective medium for data transfer. Data is transferred in the form of electromagnetic waves. It has the following features:

- Data rate is in Mbits/Sec.
- Very effective because of flexibility.
- Band width is much less than fiber.

**Example 1**

Suppose we have 20 magnetic tapes, each containing 40GB. Assume that there are enough tape readers to keep any network busy. How long will it take to transmit the data over a distance of 5Km? The choices are category 5 twisted-pair wires at 100Mbits/sec, multimode fiber at 1500Mbits/sec and single-mode fiber at 3000Mbits/sec. (Adapted from CA3: H&P)

**Solution**

The total amount of data
= total no. of mag. tapes x capacity of each tape
= 20 x 40GB= 800GB

The time for each medium:
Twisted pair = 800GB/100Mbits/sec
= 65536 sec = 18.2 hr
Multimode Fiber = 800GB/1500Mbits/sec
= 4369.06sec = 1.213 hr

Single mode Fiber = 800GB/3000Mbits/sec
= 2184.55sec
= 0.66hr

Car = time to load car + transport time + time to unload car
= 250sec + 5Km/30Kph + 250sec
= 500.16 sec = 0.13hr

## Shared/Switched Medium

**Shared Medium**
If a number of computers are connected with a single physical medium (i.e. coaxial or fiber), this situation is called shared medium. Because of many computers, collision takes place and affects the data transfer rate. As the number of machines on a physical medium increases, the data transfer rate decreases.

**Switched Medium**
To increase the throughput, a switched medium is used.
**Example 2**

Compare 20 nodes connected in three different ways: a single 100Mbits/sec shared medium; a switch connected via cat5, each segment running at 100Mbits/sec; and a switch connected via optical fiber, each segment running at 1500Mbits/sec. The shared medium is 700m long, and the average length of each segment to a switch is 55m. Both switches can support full bandwidth. Assume each switch adds 6μsec to the latency, and the average message size is 200bytes. Ignore the overhead of sending or receiving a message and contention for the network.

**Solution**

First we will calculate the aggregate bandwidth:
For shared medium

Aggregate bandwidth = 100Mbits/sec
For switched twisted pair

Aggregate bandwidth = 20 x 100Mbits/sec
= 2000Mbits/sec
For switched optical fiber

Aggregate bandwidth = 20 x 1500Mbit/sec
= 30,000Mbits/sec

Transport time = Time of flight + (message size/BW)

$$\text{Transport time shared} = \frac{(700/1000)\text{Km}}{(2/3 \times 300{,}000)\text{Km}} \times 10^6 \mu\text{sec} + (200 \times 8\text{bits} / 100\text{Mbits/sec})$$

$$= 3.5\mu\text{sec} + 16\mu\text{sec} = 19.5\mu\text{sec}$$

For the switches, the distance is twice the average segment. We must also add latency for the switch.

$$\text{Transport time switch} = 2x\frac{(55/1000)\text{Km}}{(2/3 \times 300{,}000)\text{Km}} \times 10^6\mu s + 6\mu sec + (200 \times 8\text{bits} / 100\text{Mbits/sec})$$

$$= 0.55\mu sec + 6\mu sec + 16\mu sec$$
$$= 22.55\mu sec$$

$$\text{Transport time fiber} = 2x\frac{(55/1000)\text{Km}}{(2/3 \times 300{,}000)\text{Km}} \times 10^6\mu s + 6\mu sec + (200 \times 8\text{bits} / 1500\text{Mbits/sec})$$

$$= 0.55\mu sec + 6\mu sec + 1.06\mu sec$$
$$= 7.61\mu sec$$

Although the bandwidth of the switch is many times that of the shared medium, the latency for unloaded networks is comparable.

## Connection Oriented vs. Connection less Communication

**Connection Oriented Communication**

- In this method, same path is always taken for the transfer of messages.
- It reserves the bandwidth until the transfer is complete. So no other server could use that path until it becomes free.
- Telephone exchange and circuit switching is the example of connection oriented communication.

**Connection less Communication**

- Here message is divided into packets with each packet having destination address.
- Each packet can take different path and reach the destination from any route by looking at its address.
- Postal system and packet switching are examples of connection less communication.

## Network Topologies

Computers in a network can be connected together in different ways. The following three topologies are commonly used:

- Bus topology
- Star topology

- Ring topology

**Bus Topology**
In this arrangement, computers are connected via a single shared physical medium.

**Star topology**
Computers are connected through a hub. All messages are broad cast because the hub is not an intelligent device.

**Ring Topology**
All computers are connected through a ring. Only one computer can transmit data at one time, having a pass called "Token".

## Seven Layer OSI Model

There are seven layers in this model.

1. Physical Layer
2. Data Layer
3. Network Layer
4. Transport Layer
5. Session Layer
6. Presentation Layer
7. Application Layer

**OSI Model Characteristics**

- An interface is present between any two layers.
- A layer may use the data present in another layer.
- Each layer is abstracted from other layers.
- The service provided by one layer can be used by the other layer.
- Two layers can provide same service e.g. Check Sum calculated at different layers.
- On two machines, six layers are logically connected except the physical layer. The physical layers of two machines are physically connected.

## Internet and Packet Switching

Internet works on the concept of packet switching. Application layer passes data to the lower layer and that lower layer passes data to the next lower layer and on so on. In this data passing process through different layers, different headers are attached with the data which shows the source and destination addresses, number of data bytes in packet, type of message etc. At physical layer, this packet is transmitted into the network. At reception, reverse procedure is adopted.

## Fragmentation

When a packet is lost in the network, it is re-transmitted. If the size of the packet is large

then retransmission of packet is wastage of resources and it also increases the delay in the network. To minimize this delay, a large packet is divided into small fragments. Each fragment contains a separate header having destination address and fragment number. This fragmentation effectively reduces the queuing delay. At destination, these fragments are re-assembled and data is sent to the application layer.

## Routing

Routing works on store-and-forward policy. There are three methods used for routing:

- Source-based routing
- Virtual Circuit
- Destination-based routing

## TCP/IP

Internet uses TCP/IP protocol. In the TCP/IP model, session and presentation layers are not present, so Store-Forward routing is used.