# MODULE 3
# ADDRESSING MODES

MIPS uses five addressing modes

- **Register-only addressing modes**
- **Immediate addressing modes**
- **Base addressing modes**
- **PC relative addressing modes**
- **Pseudo-direct addressing modes**

The first three modes define modes of reading and writing operands.

The last two define modes of writing the program counter, PC.

- **Register-Only Addressing**
- Register-only addressing uses registers for all source and destination operands
- All R-type instructions use register-only addressing.
- **Immediate Addressing**
- Immediate addressing uses the 16-bit immediate along with registers as operands
- Add immediate (addi) and load upper immediate (lui), use immediate addressing
- **Base Addressing**
- Memory access instructions, such as load word (lw) and store word (sw), use base addressing
- The effective address of the memory operand is found by adding the base address in register rs to the sign-extended 16-bit offset found in the immediate field

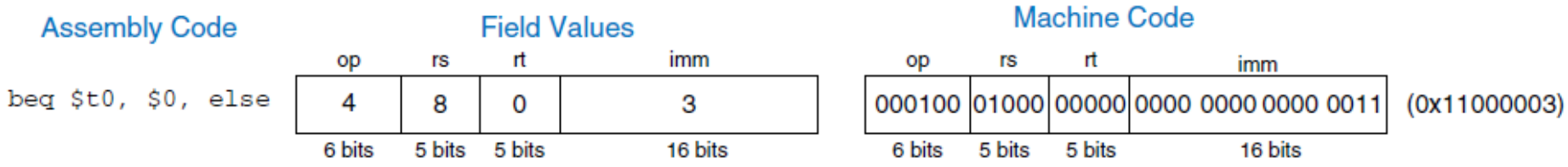
# PC-relative Addressing

- Conditional branch instructions use PC-relative addressing to specify the new value of the PC if the branch is taken
- The signed offset in the immediate field is added to the PC to obtain the new PC
- So the branch destination address is said to be relative to the current PC

# CALCULATING THE BRANCH TARGET ADDRESS

**MIPS Assembly Code**

```
0xA4            beq  $t0, $0, else
0xA8            addi $v0, $0, 1
0xAC            addi $sp, $sp, 8
0xB0            jr   $ra
0xB4      else: addi $a0, $a0, -1
0xB8            jal  factorial
```



# PC-relative Addressing

- The branch target address **(BTA)** is the address of the next instruction to execute if the branch is taken
- The beq instruction has a BTA of 0xB4
- The 16-bit immediate field gives the number of instructions between the BTA and the instruction after the branch instruction (the instruction at PC4)
- In this case, the value in the immediate field of beq is 3 because the BTA (0xB4) is 3 instructions past PC4 (0xA8)
- The processor calculates the BTA from the instruction by sign extending the 16-bit immediate, multiplying it by 4 (to convert words to bytes), and adding it to PC4.

Calculate the immediate field and show the machine code for the branch not equal (bne) instruction in the following program.

```
# MIPS assembly code
0x40 loop: add  $t1, $a0, $s0
0x44       lb   $t1, 0($t1)
0x48       add  $t2, $a1, $s0
0x4C       sb   $t1, 0($t2)
0x50       addi $s0, $s0, 1
0x54       bne  $t1, $0, loop
0x58       lw   $s0, 0($sp)
```

# Pseudo-Direct Addressing

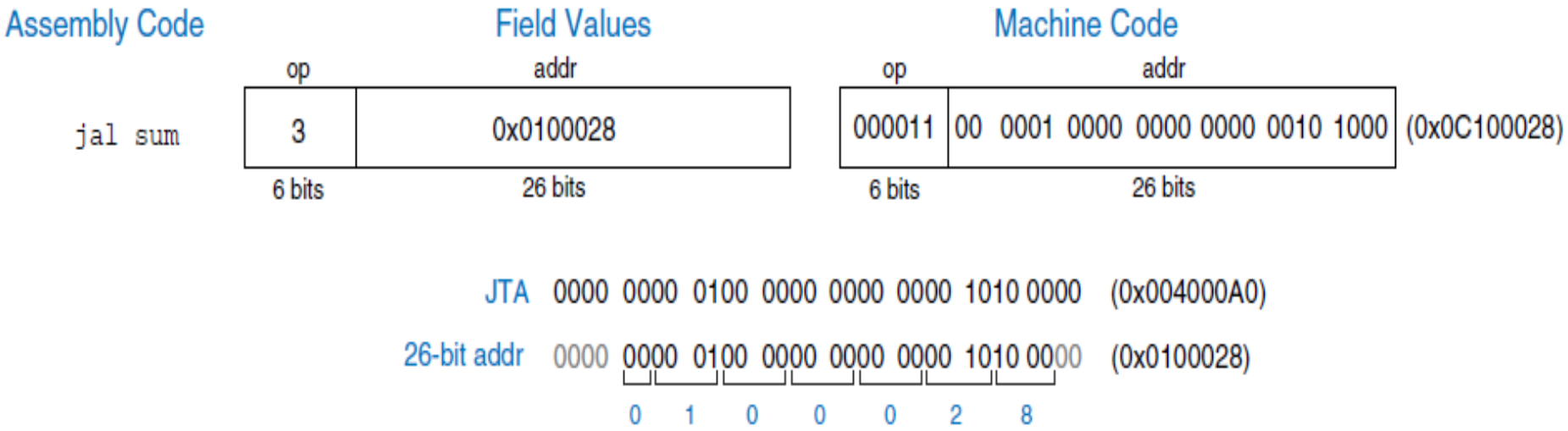
- In direct addressing, an address is specified in the instruction
- The jump instructions, j and jal, use direct addressing to specify a 32-bit jump target address (JTA)which indicate the instruction address to execute next
- The J-type instruction encoding does not have enough bits to specify a full 32-bit JTA
- Six bits of the instruction are used for the opcode, so only 26 bits are left to encode the JTA
- The two least significant bits, JTA1:0, should always be 0, because instructions are word aligned
- The next 26 bits, JTA27:2, are taken from the addr field of the instruction.
- The four most significant bits, JTA31:28, are obtained from the four most significant bits of PC+4

# CALCULATING THE JUMP TARGET ADDRESS

**MIPS Assembly Code**

```
0x0040005C          jal     sum
. . .
0x004000A0  sum:    add     $v0, $a0, $a1
```

- A jal instruction using pseudo-direct addressing
- The JTA of the jal instruction is 0x004000A0
- The top four bits and bottom two bits of the JTA are discarded
- The remaining bits are stored in the 26-bit address field (addr)
- The processor calculates the JTA from the J-type instruction by appending two 0's and prepending the four most significant bits of PC+4 to the 26-bit address field (addr)
- All J-type instructions, j and jal, use pseudo-direct addressing


Assembly Code
Field Values
Machine Code
op
addr
op
addr
jal sum
3
0x0100028
000011
00 0001 0000 0000 0000 0010 1000
(0x0C100028)
6 bits
26 bits
6 bits
26 bits
JTA 0000 0000 0100 0000 0000 0000 1010 0000 (0x004000A0)
26-bit addr 0000 0000 0100 0000 0000 0000 1010 0000 (0x0100028)
0 1 0 0 0 2 8

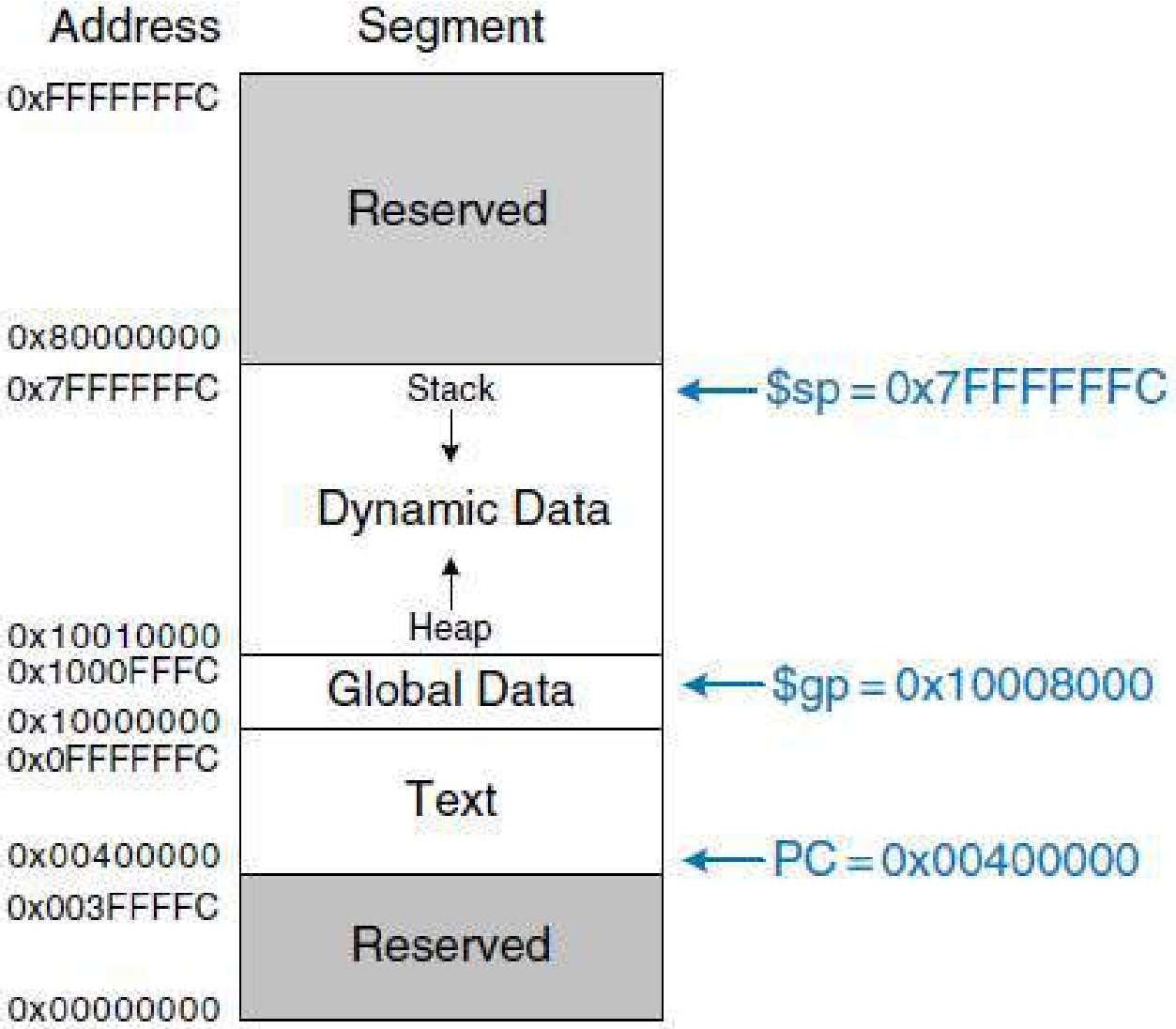
# COMPILING,ASSEMBLING, AND LOADING

- It describes how to compile and assemble a complete high-level program and how to load the program into memory for execution
- MIPS memory map, which defines where code, data, and stack memory are located

# The Memory Map

- With 32-bit addresses, the MIPS address space spans $2^{32}$ bytes= 4 gigabytes (GB)
- Word addresses range from 0 to 0xFFFFFFFC
- The MIPS architecture divides the address space into four parts or segments
  - the text segment
  - global data segment
  - dynamic data segment and
  - reserved segments

## MIPS memory map



- **Text Segment**
- The text segment stores the machine language program
- It is large enough to accommodate almost 256 MB of code
- The four most significant bits of the address in the text space are all 0, so the j instruction can directly jump to any address in the program
- **Global Data Segment**
- The global data segment stores global variables
- Global variables are defined at start-up, before the program begins executing
- These variables are declared outside the main procedure in a C program and can be accessed by any procedure
- The global data segment is large enough to store 64 KB of global variables

# Global Data Segment

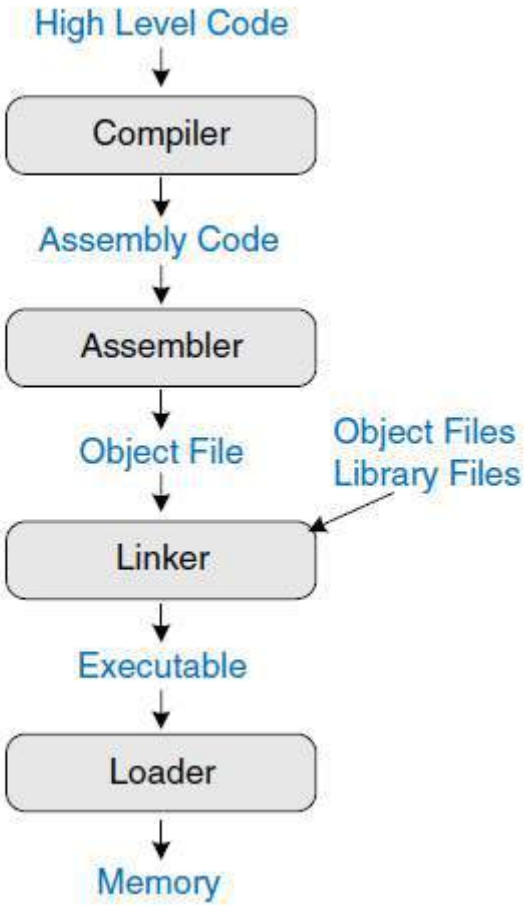
- Global variables are accessed using the global pointer ($gp), which is initialized to 0x100080000
- Like the stack pointer ($sp), $gp does not change during program execution
- Any global variable can be accessed with a 16-bit positive or negative offset from $gp
- The offset is known at assembly time, so the variables can be efficiently accessed using base addressing mode with constant offsets

# Dynamic Data Segment

- The dynamic data segment holds the stack and the heap
- The data in this segment are not known at start-up but are dynamically(run time) allocated and deallocated throughout the execution of the program
- This is the largest segment of memory used by a program, spanning almost 2 GB of the address space
- The stack is used to save and restore registers used by functions and to hold local variables
- The stack grows downward from the top of the dynamic data segment (0x7FFFFFFC) and is accessed in last-in-first-out (LIFO) order
- In C, memory allocations are made by the malloc function
- In C++ and Java, new is used to allocate memory
- The heap grows upward from the bottom of the dynamic data segment

- **Dynamic Data Segment**
- If the stack and heap ever grow into each other, the program's data can become corrupted
- The memory allocator tries to ensure that this never happens by returning an out-of-memory error if there is insufficient space to allocate more dynamic data
- **Reserved Segments**
- The reserved segments are used by the operating system and cannot directly be used by the program
- Part of the reserved memory is used for interrupts

# Translating and Starting a Program



# Translating and Starting a Program

- Figure shows the steps required to translate a program from a high level language into machine language and to start executing
- The high-level code is compiled into assembly code
- The assembly code is assembled into machine code in an object file
- The linker combines the machine code with object code from libraries and other files to produce an entire executable program
- Finally, the loader loads the program into memory and starts execution

# Step 1: Compilation

- Compiler translates high-level code into assembly language
- The .data and .text keywords are assembler directives that indicate where the text and data segments begin
- Labels are used for global variables f, g, and sum

**HIGH LEVEL CODE**

```
int f,g,sum//global variables
int main (void)
{
f=2;
g=3;
sum = f+g;
}
```

**ASSEMBLY CODE**

```
.data
f:
g:
sum:
.text
 main:
addi    $a0,$0,2
sw      $a0,f
addi    $a1,$0,3
sw      $a1,g
add     $v0,$a0,$a1
sw      $v0,sum
```

# Step 2: Assembling

- Assembler turns the assembly language code into an object file containing machine language code
- The assembler makes two passes through the assembly code
- On the first pass, the assembler assigns instruction addresses and finds all the symbols, such as labels and global variable names
- The code after the first assembler pass is shown here

```
0x00400000    addi    $a0,$0,2
0x00400004    sw      $a0,f
0x00400008    addi    $a1,$0,3
0x0040000c    sw      $a1,g
0x00400010    add     $v0,$a0,$a1
0x00400014    sw      $v0,sum
```

# Step 2: Assembling

- The names and addresses of the symbols are kept in a symbol table
- The symbol addresses are filled in after the first pass, when the addresses of labels are known
- Global variables are assigned storage locations in the global data segment of memory, starting at memory address 0x10000000
- On the second pass through the code, the assembler produces the machine language code
- Addresses for the global variables and labels are taken from the symbol table
- The machine language code and symbol table are stored in the object file

# Symbol table

| Symbol | Address |
|---|---|
| f | 0x10000000 |
| g | 0x10000004 |
| y | 0x10000008 |
| main | 0x00400000 |
| sum | 0x0040002C |

# Step 3: Linking

- Large  programs contain more than one file
- If the programmer changes only one of the files, it would be wasteful to recompile and reassemble the other files
- linker combines all the object files into one machine language file called executable file

| Executable file header | Text Size | Data Size |
|---|---|---|
| | 0x18 (24 bytes) | 0xC (12 bytes) |
| **Text segment** | **Address** | **Instruction** |
| | 0x00400000 | 0x20040002 |
| | 0x00400004 | 0xAF848000 |
| | 0x00400008 | 0x20050003 |
| | 0x0040000c | 0xAF858004 |
| | 0x00400010 | 0x00851020 |
| | 0x00400014 | 0xAF828008 |
| **Data segment** | **Address** | **Data** |
| | 0x10000000 | f |
| | 0x10000004 | g |
| | 0x10000008 | sum |

```
addi    $a0,$0,2
sw      $a0, 0x8000 ($gp)
addi    $a1,$0,3
sw      $a1, 0x8004 ($gp)
add     $v0,$a0,$a1
sw      $v0, 0x8008 ($gp)
```

- Figure shows the executable file
- It has three sections:

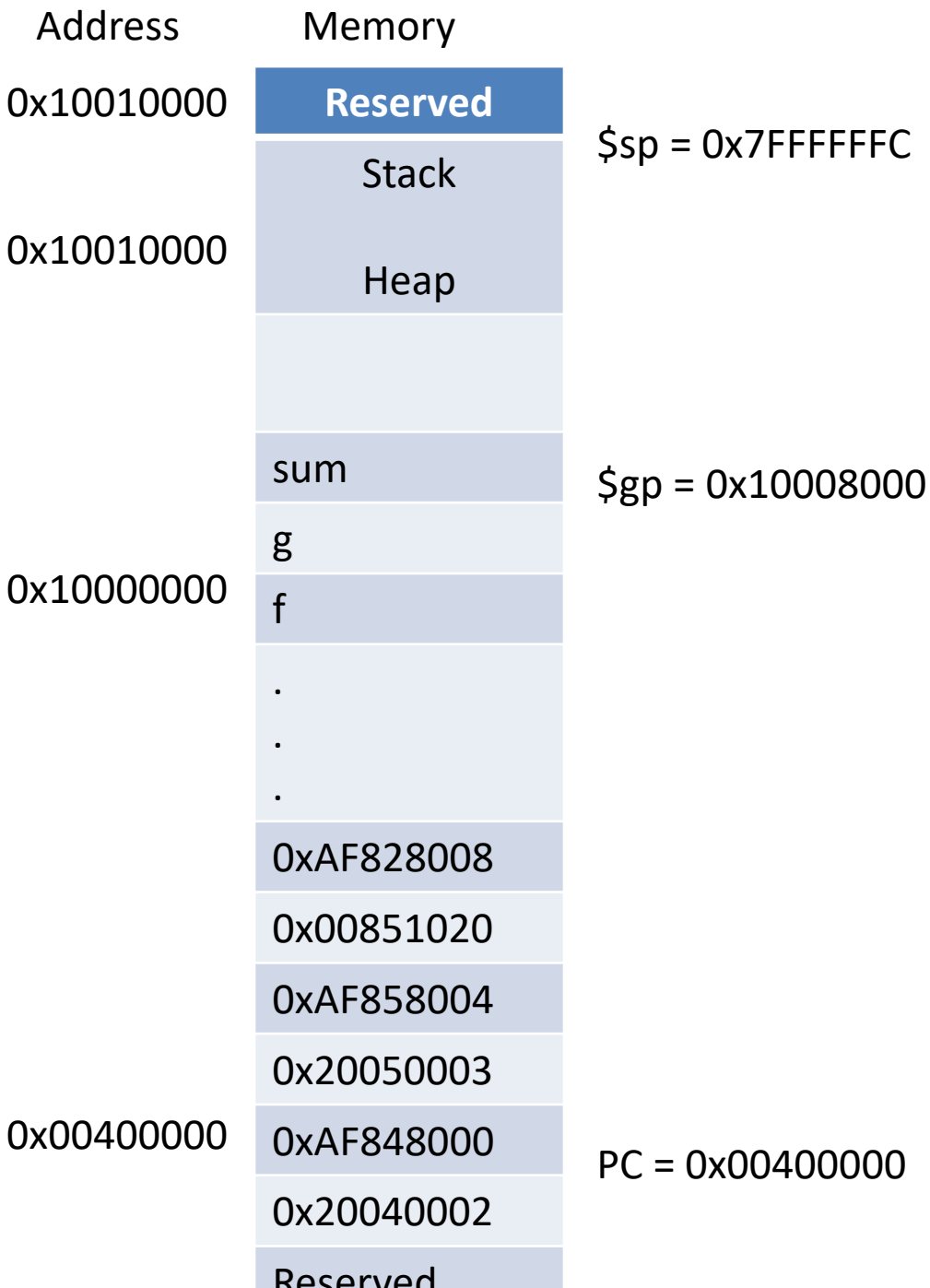
1. the executable file header
2. the text segment
3. the data segment

- The executable file header reports the text size (code size) and data size (amount of globally declared data)
- The text segment gives the instructions and the addresses where they are to be stored
- The executable file includes only machine instructions
- The data segment gives the address of each global variable
- The global variables are addressed with respect to the base address given by the global pointer, $gp

- sw $a0, 0x8000($gp), stores the value 2 to the global variable f, which is located at memory address 0x10000000
- The offset, 0x8000, is a 16-bit signed number that is sign-extended and added to the base address, $gp
- $gp + 0x8000 =0x10008000 + 0xFFFF8000 = 0x10000000, the memory address of variable f

# Step 4: Loading

- The operating system loads the program into memory
- The operating system sets $gp to 0x10008000 (the middle of the global data segment) and $sp to 0x7FFFFFFC (the top of the dynamic data segment)


Address
Memory
0x10010000
Reserved
$sp = 0x7FFFFFFC
Stack
0x10010000
Heap
sum
$gp = 0x10008000
g
0x10000000
f
.
.
.
0xAF828008
0x00851020
0xAF858004
0x20050003
0x00400000
0xAF848000
PC = 0x00400000
0x20040002
Reserved

# Pseudoinstructions

- If an instruction is not available in the MIPS it can be performed using one or more existing MIPS instructions
- MIPS is a reduced instruction set computer (RISC)
- MIPS defines pseudoinstructions that are not actually part of the instruction set
- These are commonly used by programmers and compilers
- When converted to machine code, pseudoinstructions are translated into one or more MIPS instructions

# Pseudoinstructions

| Pseudoinstruction | Corresponding MIPS Instructions |
|---|---|
| `li $s0, 0x1234AA77` | `lui $s0, 0x1234`<br>`ori $s0, 0xAA77` |
| `mul $s0. $s1. $s2` | `mult $s1. $s2`<br>`mflo $s0` |
| `clear $t0` | `add $t0, $0, $0` |
| `move $s1, $s2` | `add $s2, $s1, $0` |
| `nop` | `sll $0, $0, 0` |

## Pseudoinstruction using $at

| Pseudoinstruction | Corresponding MIPS Instructions |
|---|---|
| `beq $t2,` $imm_{15:0}$`, Loop` | `addi $at, $0,` $imm_{15:0}$<br>`beq $t2, $at, Loop` |

- In nop instruction PC is incremented by 4 upon its execution
- No other registers or memory values are altered
- The pseudoinstruction beq requires a temporary register in which to store the 16-bit immediate
- Assemblers use the assembler register, $at, for such purposes

# **Exception**s

- An exception is like an unscheduled procedure call that jumps to a new address
- Exceptions may be caused by hardware or software
- A hardware exception triggered by an input/output (I/O) is called an **inetrrupt**
- The program may encounter an error condition such as an **undefined instruction**
- Software exceptions are called **traps**
- Other causes of exceptions include division by zero
- attempts to read nonexistent memory
- hardware malfunctions
- debugger breakpoints
- arithmetic overflow etc

# **Exceptions**

- The processor records the cause of an exception and the value of the PC at the time the exception occurs
- It then jumps to the **exception handler procedure**
- The exception handler examines the cause of the exception and responds appropriately
- It then returns to the program that was executing before the exception took place
- In MIPS, the exception handler is always located at **0x80000180**

- The MIPS architecture uses a special-purpose register, called the Cause register, to record the cause of the exception
- Different codes are used to record different exception causes

## Exception cause codes

| Exception | Cause |
|---|---|
| hardware interrupt | 0x00000000 |
| system call | 0x00000020 |
| breakpoint/divide by 0 | 0x00000024 |
| undefined instruction | 0x00000028 |
| arithmetic overflow | 0x00000030 |

- MIPS uses another special-purpose register called the Exception Program Counter (EPC) to store the value of the PC at the time an exception
- The EPC and Cause registers are not part of the MIPS register file
- The mfc0 (move from coprocessor 0) instruction copies these
- Coprocessor 0 is called the MIPS processor control; it handles interrupts and processor diagnostics
- For example, mfc0 $t0, cause copies the Cause register into $t0

- The **syscall and break instructions** cause traps
- An exception causes the processor to jump to the exception handler
- The exception handler saves registers on the stack, then uses mfc0 to look at the cause and respond accordingly
- When the handler is finished, it restores the registers from the stack, copies the return address from EPC and returns

# Signed and Unsigned Instructions

- The MIPS architecture uses two's complement representation of signed numbers
- MIPS supports instructions that come in signed and unsigned flavors, including addition and subtraction, multiplication ,division, set less than, and partial word loads

# Addition and Subtraction

- Adding the following two huge positive numbers gives a negative result: 0x7FFFFFFF + 0x7FFFFFFF = 0xFFFFFFFE =- 2.
- This is called **arithmetic *overflow***
- The MIPS processor takes an exception on arithmetic over flow
- MIPS provides signed and unsigned versions of addition and subtraction.
- The signed versions are **add, addi, and sub**
- The unsigned versions are **addu, addiu, and subu**

# Multiplication and Division

- Multiplication and division behave differently for signed and unsigned numbers
- For example, as an unsigned number, 0xFFFFFFFF represents a large number, but as a signed number it represents -1
- 0xFFFFFFFF * 0xFFFFFFFF would equal FFFFFFFE00000001 if the numbers were unsigned but 0x0000000000000001 if the numbers were signed
- Therefore, multiplication and division come in both signed and unsigned flavors
- mult and div treat the operands as signed numbers. multu and divu treat the operands as unsigned numbers

# Set Less Than

- Set less than instructions can compare either two registers (slt) or a register and an immediate (slti)
- Set less than also comes in signed (slt and slti) and unsigned (sltu and sltiu) versions
- In a signed comparison,0x80000000 is less than any other number, because it is the most negative two's complement number
- In an unsigned comparison,0x80000000 is greater than 0x7FFFFFFF but less than 0x80000001,because all numbers are positive

# Loads

- Byte loads come in signed (lb) and unsigned (lbu) versions
- lb sign-extends the byte, and lbu zero-extends the byte to fill the entire 32-bit register
- Similarly, MIPS provides signed and unsigned half-word loads (lh and lhu), which load two bytes into the lower half and sign- or zero-extend the upper half of the word

# Floating-Point Instructions

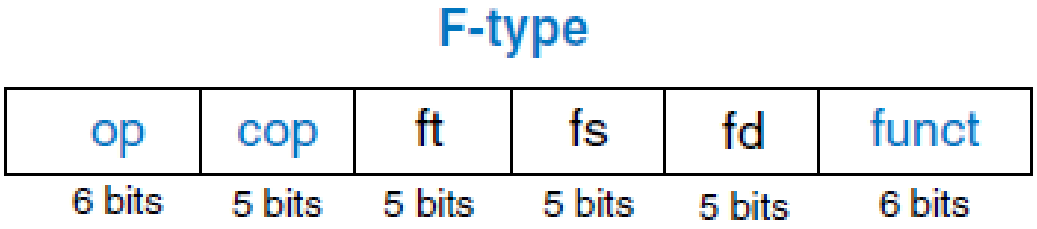
- The MIPS architecture supports a floating-point coprocessor
- MIPS defines 32 32-bit floating-point registers, $f0–$f31
- MIPS supports both single- and double-precision IEEE floating point arithmetic
- Double precision (64-bit) numbers are stored in pairs of 32-bit registers,
- So the 16 even-numbered registers ($f0, $f2, $f4, . . . , $f30) are used to specify double-precision operations

# MIPS floating-point register set

| Name | Number | Use |
|---|---|---|
| $fv0-$fv1 | 0, 2 | procedure return values |
| $ft0-$ft3 | 4, 6, 8, 10 | temporary variables |
| $fa0-$fa1 | 12, 14 | procedure arguments |
| $ft4-$ft5 | 16, 18 | temporary variables |
| $fs0-$fs5 | 20, 22, 24, 26, 28, 30 | saved variables |

# Floating-Point Instructions

- Floating-point instructions all have an opcode of 17 (10001)
- They require both a funct field and a cop (coprocessor) field to indicate the type of instruction
- MIPS defines the F-type instruction format for floating-point instructions
- cop=16(10000) for single-precision instructions or 17 (10001) for double precision instructions
- Instruction precision is indicated by .s and .d in the mnemonic



# Floating-Point Instructions

- Floating-point arithmetic instructions include
  - addition (add.s, add.d)
  - subtraction (sub.s, sub.d)
  - multiplication (mul.s, mul.d)
  - division (div.s, div.d)
  - negation (neg.s, neg.d)
  - absolute value (abs.s, abs.d)

# MICROARCHITECTURES

**Architecture:-** Specifies the programmer's view of the processor in terms of registers, instructions and memory.
Example- MIPS architecture, IA 32 architecture

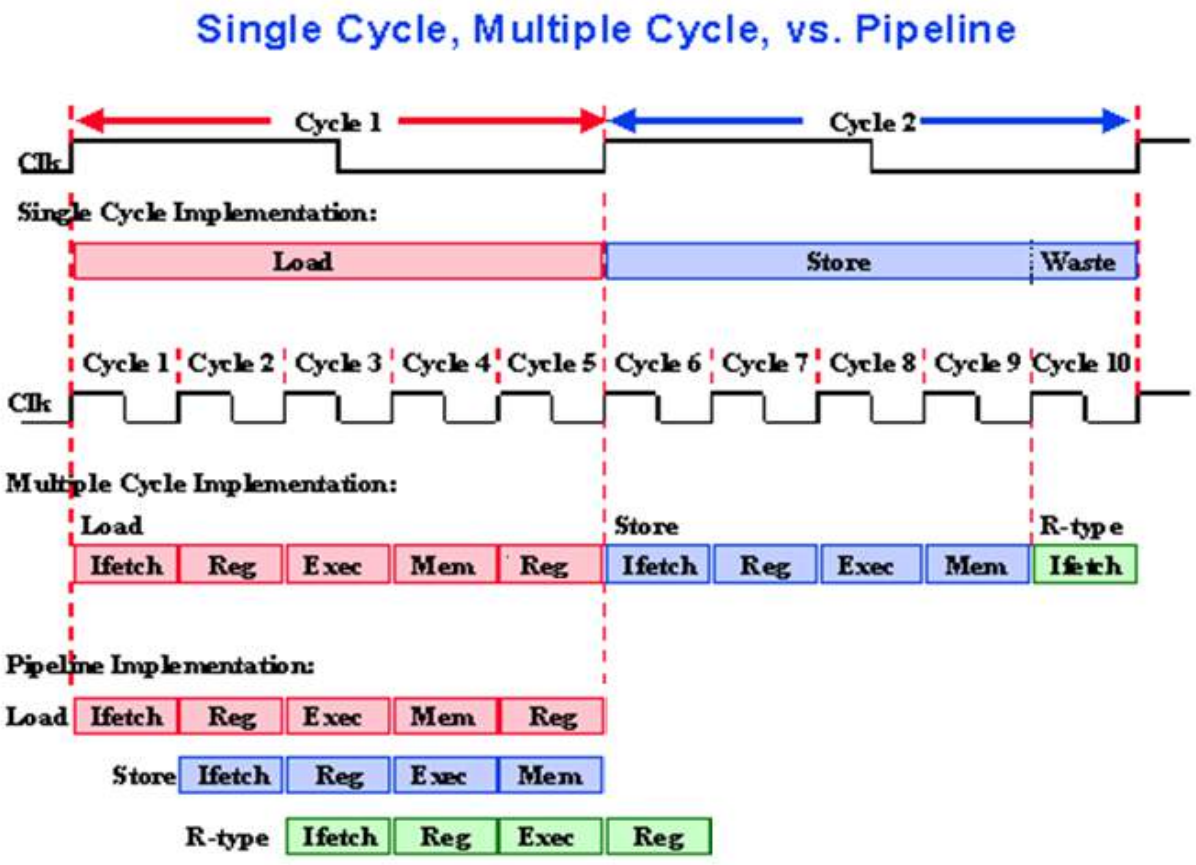
**Microarchitecture:-** Connection between logic and architecture.

It is the specific arrangement of registers, ALUs, Finite state machine (FSM), memories and other logic building blocks needed to implement architecture.

- Any architecture can have different microarchitectures each with different performance, cost and complexity.

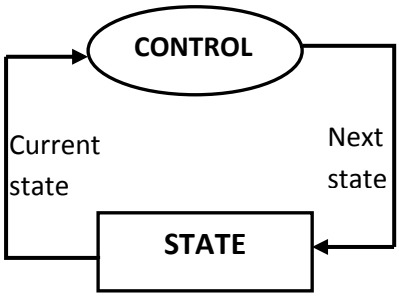
**Different microarchitectures of MIPS architecture**

1. √ Single cycle microarchitecture
   - All operations (fetch, decode, execute, write back) are performed in a single clock cycle
   - The instruction which need longest time decide the clock frequency
2. √ Multi cycle microarchitecture
   - All operations(fetch, decode, execute, write back) are performed in a multiple clock cycle
   - Each instruction takes different number of clocks.
3. Pipelined microarchitecture
   - Fetching of next instruction while decoding current instruction and executing previous instruction.



**State elements of MIPS processor -** A computer is just a big fancy state machine.

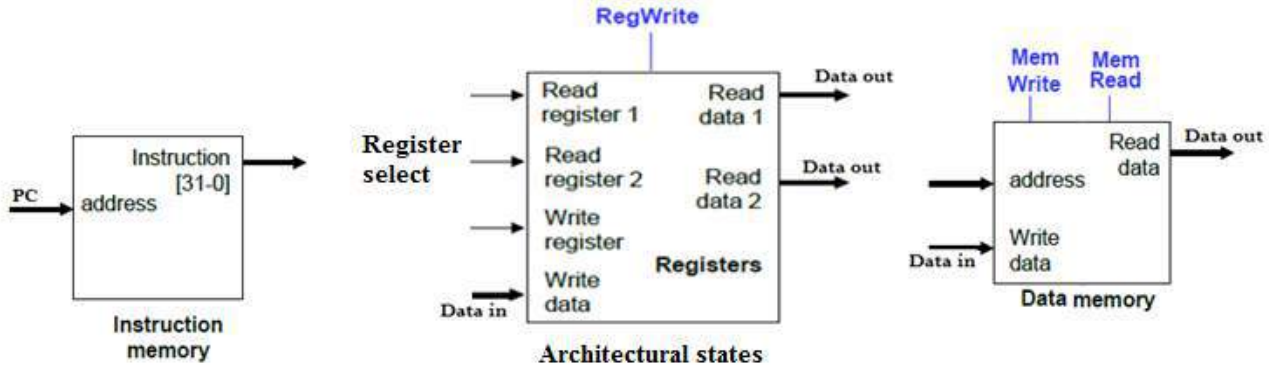
Registers and memory form the states and the processor keeps reading and updating the state, according to the instructions in some program.




Collection of state elements and their interconnection in a processor is called **Datapath** (or EU or DPU)

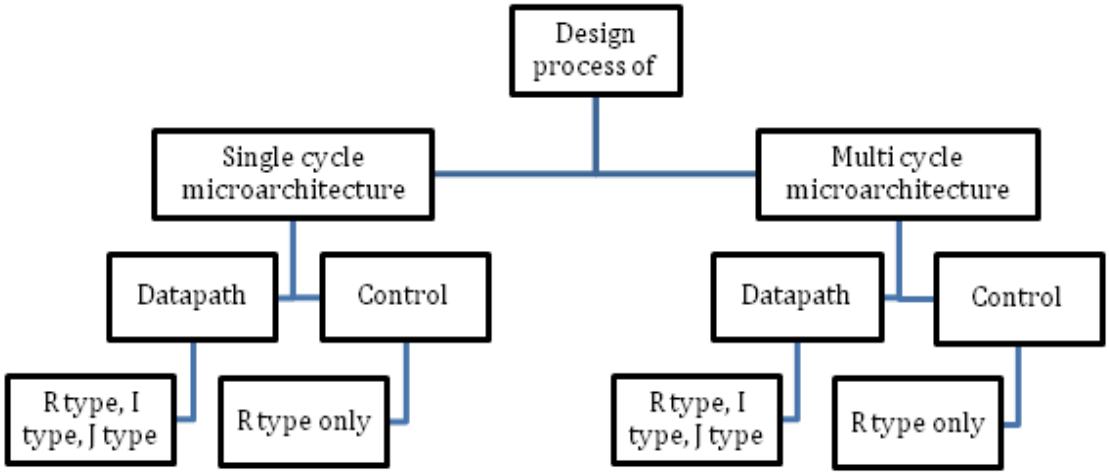
The **control unit** interprets the instruction and produces the control signals in proper sequence to carry out the operation.

1. Architectural state – consists of Program Counter (PC) and the 32 register set
2. Instruction memory
3. Data memory
4. Non architectural states –additional set of registers and circuits to either simplify the logic or improve the performance
   eg:- additional registers in multi cycle microarchitecture




- We can read from two registers at same time called **2 port memory**
- RegWrite should be 1 for a register write operation.

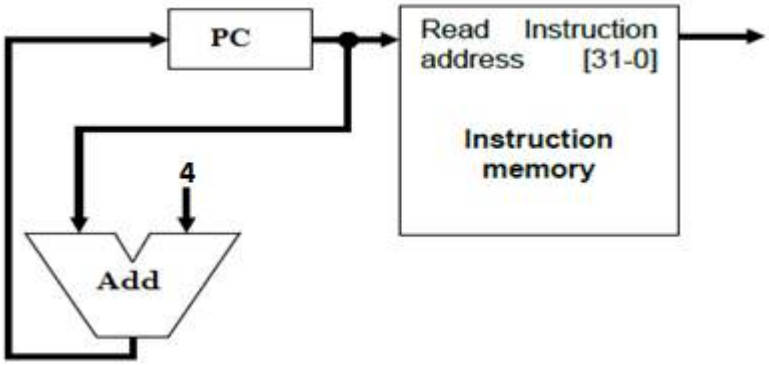
Topics in 4th module - Performance analysis and Design process of single cycle and multi cycle microarchitecture



**Design of single cycle microarchitecture *Datapath for***

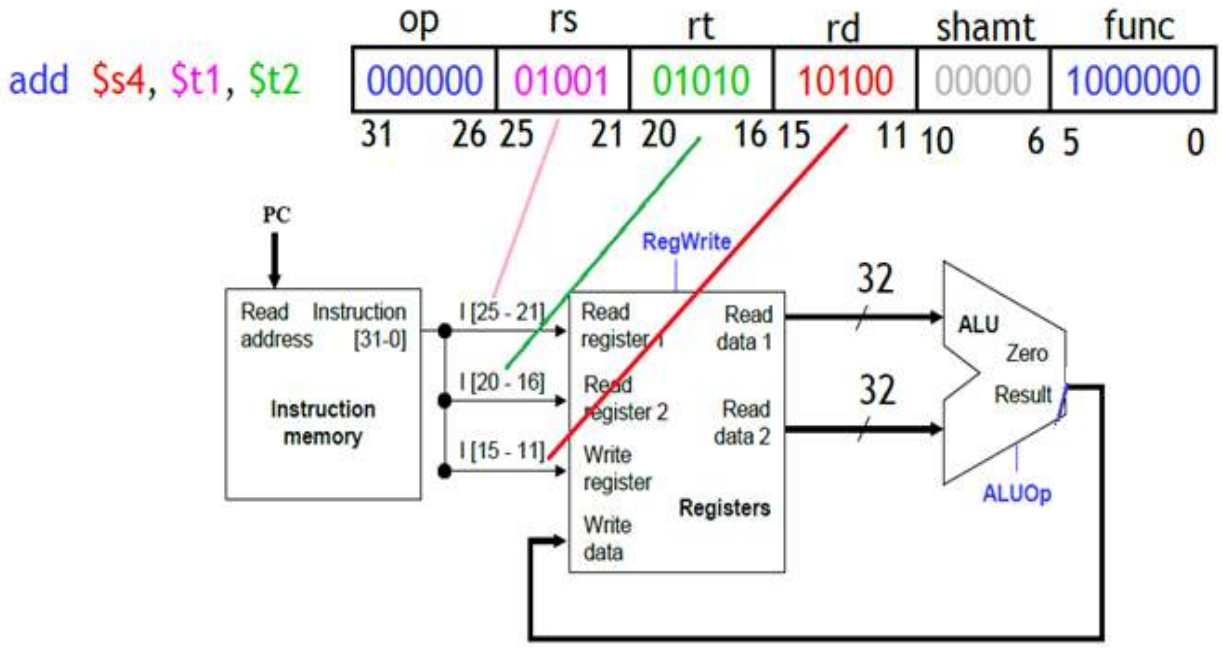
1. Datapath for Instruction fetching

   - The CPU is always in an infinite loop, fetching instructions from memory and executing them.
   - The program counter or PC register holds the address of the current instruction.
   - MIPS instructions are each four bytes long, so the PC should be incremented by four to read the next instruction in sequence.




2. Datapath for R type instruction

   - Register-to-register arithmetic instructions use the R-type format.
   - R-type instructions must access registers and an ALU.
   - Ther is thirty two 32-bits registers and each register specifier (address) is 5 bits long.
   - We can read from two registers at time called **2 port memoy**
   - RegWrite should be 1 for a register write operation.

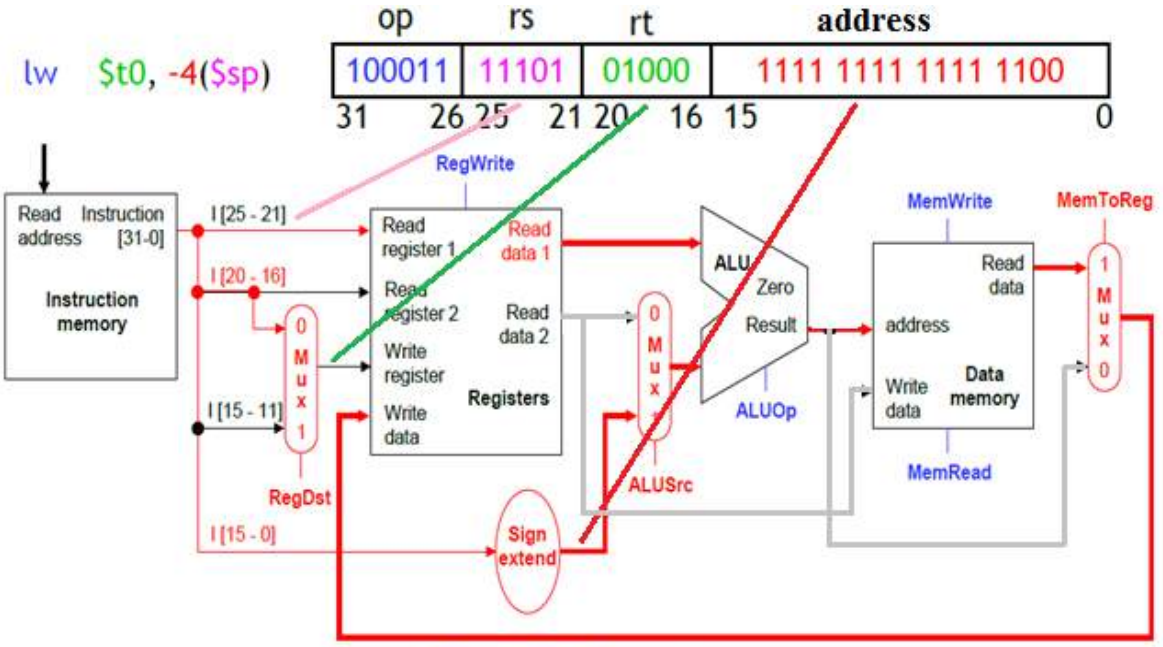



Datapath for R type Instruction

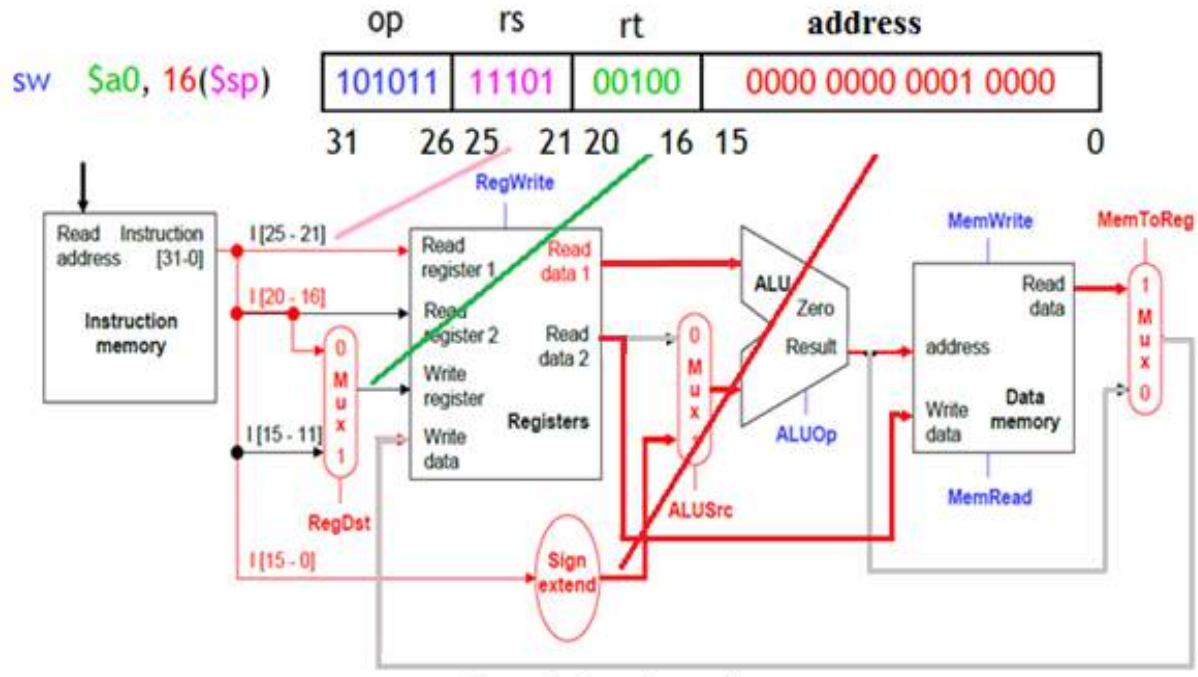
3. Datapath for I type instruction (lw, sw - Base addressing mode)

The lw, sw and beq instructions all use the I-type encoding.

- rt is the destination for lw, but a *source* for beq and sw.
- Address is a 16-bit signed constant.
- For an instruction like lw $t0, –4($sp), the base register $sp is added to the *sign-extended* constant to get a data memory address.
- This means the ALU must accept *either* a register operand for arithmetic instructions *or* a sign-extended immediate operand for lw and sw.
- We'll add a multiplexer, controlled by ALUSrc (ALU source), to select either a register operand (0) or a constant operand (1).



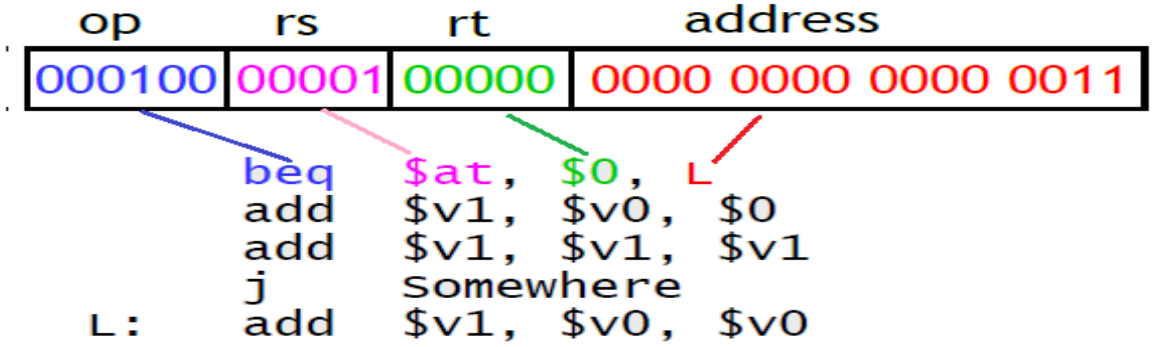

Datapath for lw instruction




Datapath for sw instruction

4. Datapath for I type instruction (beq – PC relative addressing mode)

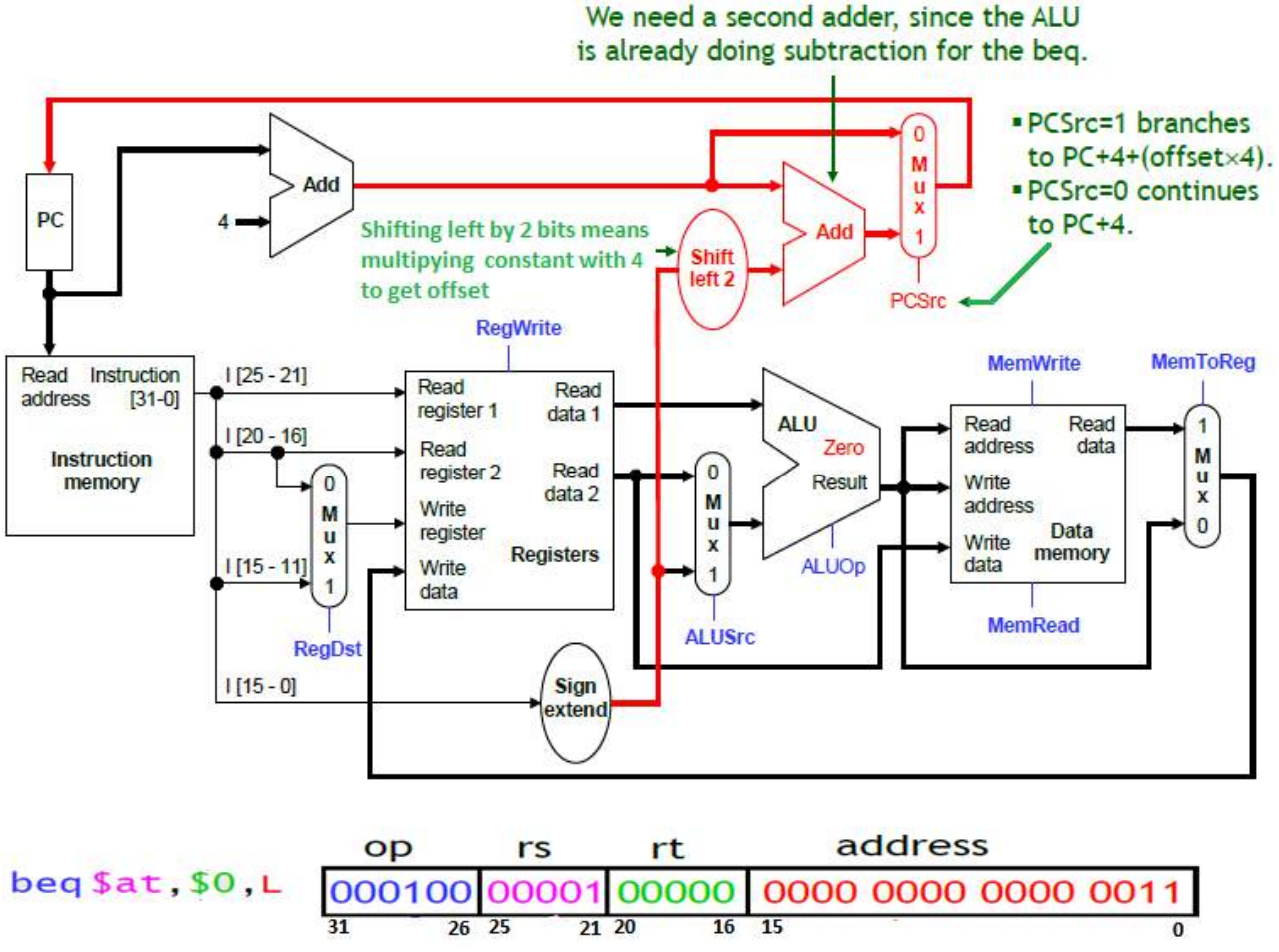
- For branch instructions, the constant is not an address but an *instruction offset (or word offset)* from the *next* program counter to the desired address.
- The target address L is three *instructions* past the first add, so the encoding of the branch instruction has 0000 0000 0000 0011 for the address field.




- Instructions are four bytes long, so the actual memory offset is 12 bytes.

**The steps in executing a beq**

1. Fetch the instruction, like beq $at, $0, offset, from memory.
2. Read the source registers, $at and $0, from the register file.
3. Compare the values by subtracting them in the ALU.
4. If the subtraction result is 0, the source operands were equal and the PC should be loaded with the target address, PC + 4 + (offset x 4). Zero flag will set to 1 if the result is zero. Zero flag is going as input to the control unit to take the decision.
5. Otherwise the branch should not be taken, and the PC should just be incremented to PC + 4 to fetch the next instruction sequentially.







**Design of single cycle microarchitecture *Control unit***

The **control unit** is responsible for setting all the control signals so that each instruction is executed properly.

**Generating control signals**

- The entire 32 instruction bits are not needed for the control unit to generate control signals.
- Most of the signals can be generated from the instruction opcode alone.
- The control unit needs 13 bits of inputs.
  - Six bits make up the instruction's opcode.
  - Six bits come from the instruction's func field.
  - It also needs the Zero flag of the ALU.

The figure below shows the final Datapath of single cycle microarchitecture. There are about 8 control signals, RegDst, RegWrite, ALUSrc, ALUControl, MemWrite, MemRead, PCSrc, MemToReg.

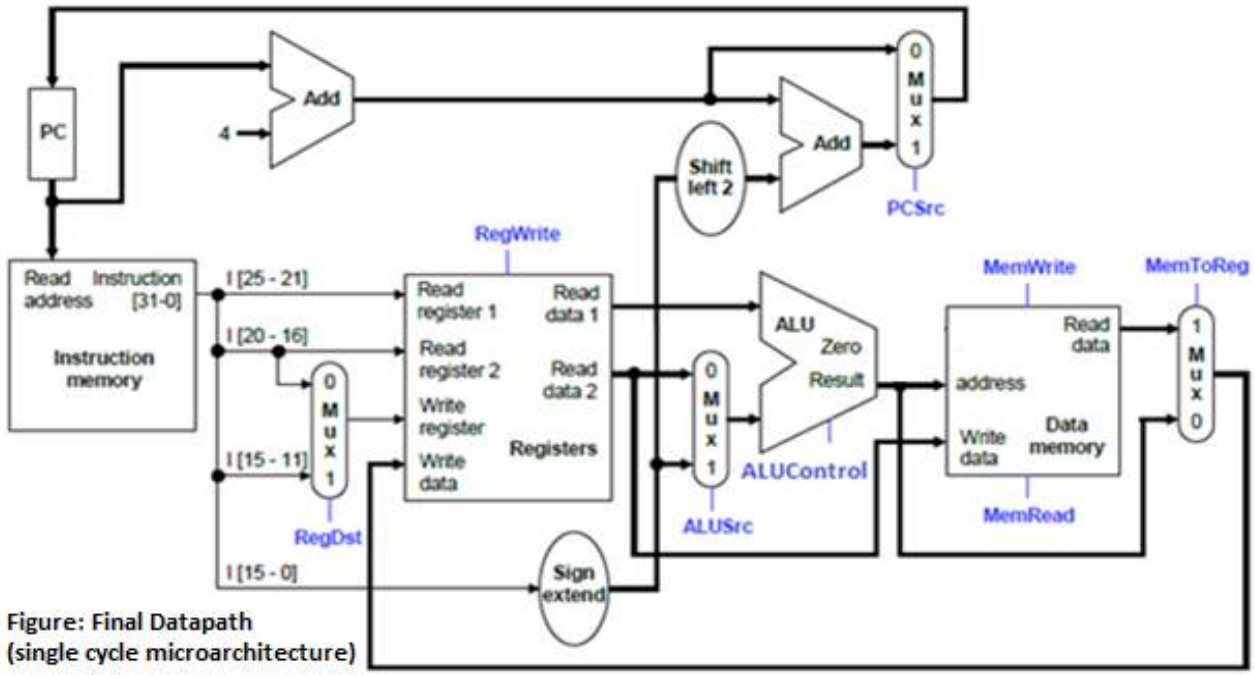



Figure: Final Datapath
(single cycle microarchitecture)

The control unit producing the control signals for this datapath can be built by combinational and sequential circuit design.

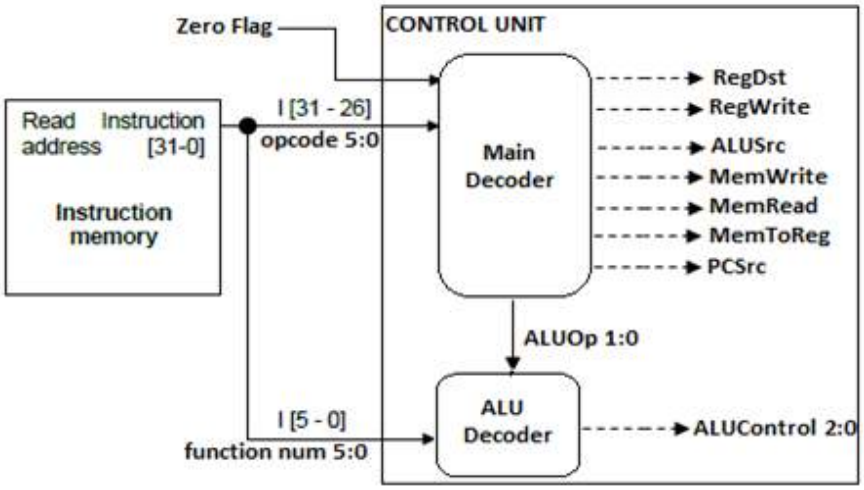



Figure: Control Unit
(Single cycle microarchitecture)

**Design of <u>Control unit for R Type Instructions (single cycle)</u>**

| Instruction | Control Signals | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | RegDst $C_8$ | ALUSrc $C_7$ | MemtoReg $C_6$ | RegWrite $C_5$ | MemRead $C_4$ | MemWrite $C_3$ | Branch $C_2$ | ALUOp $C_1C_0$ |
| R-Type | **1** | 0 | 0 | 1 | x | x | 0 | 10 |

| **Inputs** (Opcode field in the instruction) | | | | | | **Outputs** (control signals) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $Inst_{31}$ | $Inst_{30}$ | $Inst_{29}$ | $Inst_{28}$ | $Inst_{27}$ | $Inst_{26}$ | $C_8$ | $C_7$ | $C_6$ | $C_5$ | $C_4$ | $C_3$ | $C_2$ | $C_1$ | $C_0$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | x | x | 0 | 1 | 0 |

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

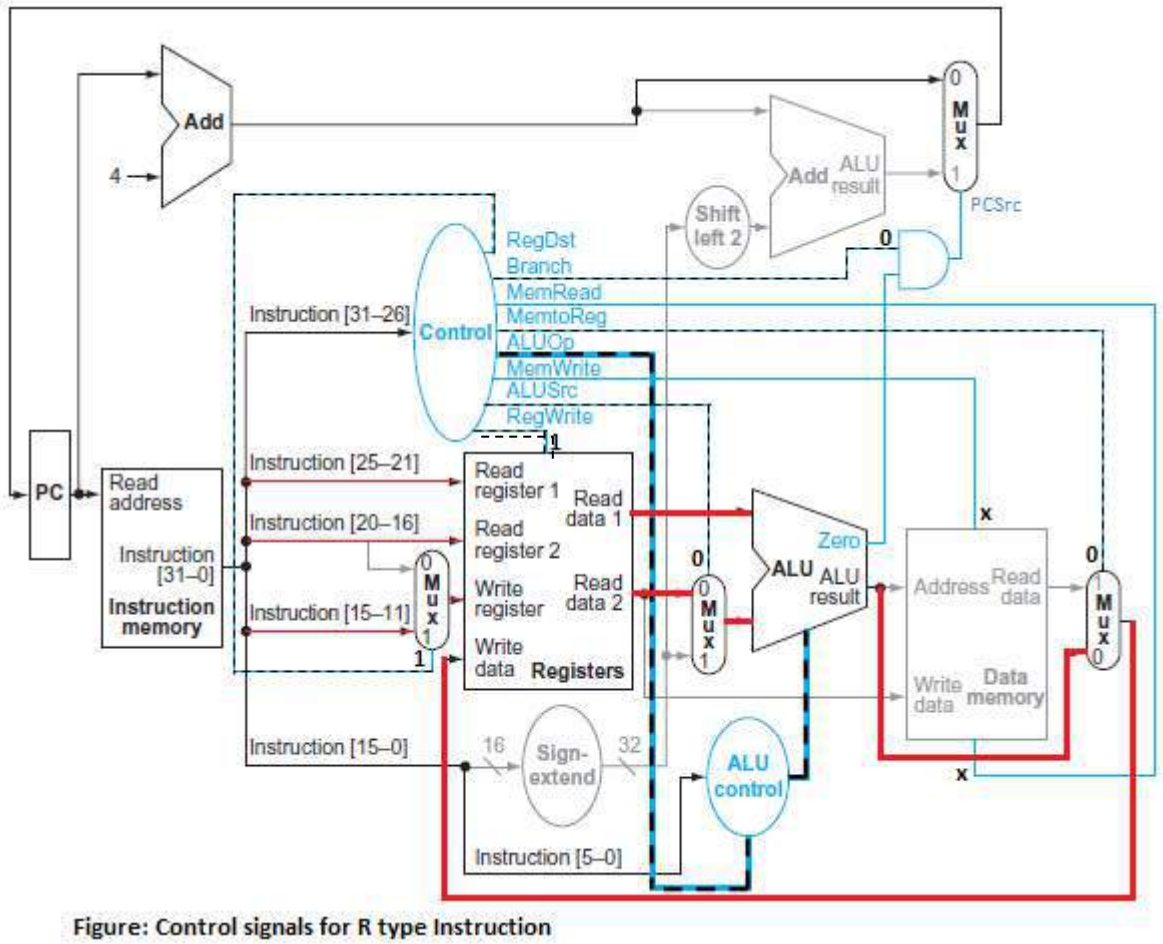



Figure: Control signals for R type Instruction

**Design of multi cycle microarchitecture *Datapath and Control unit***

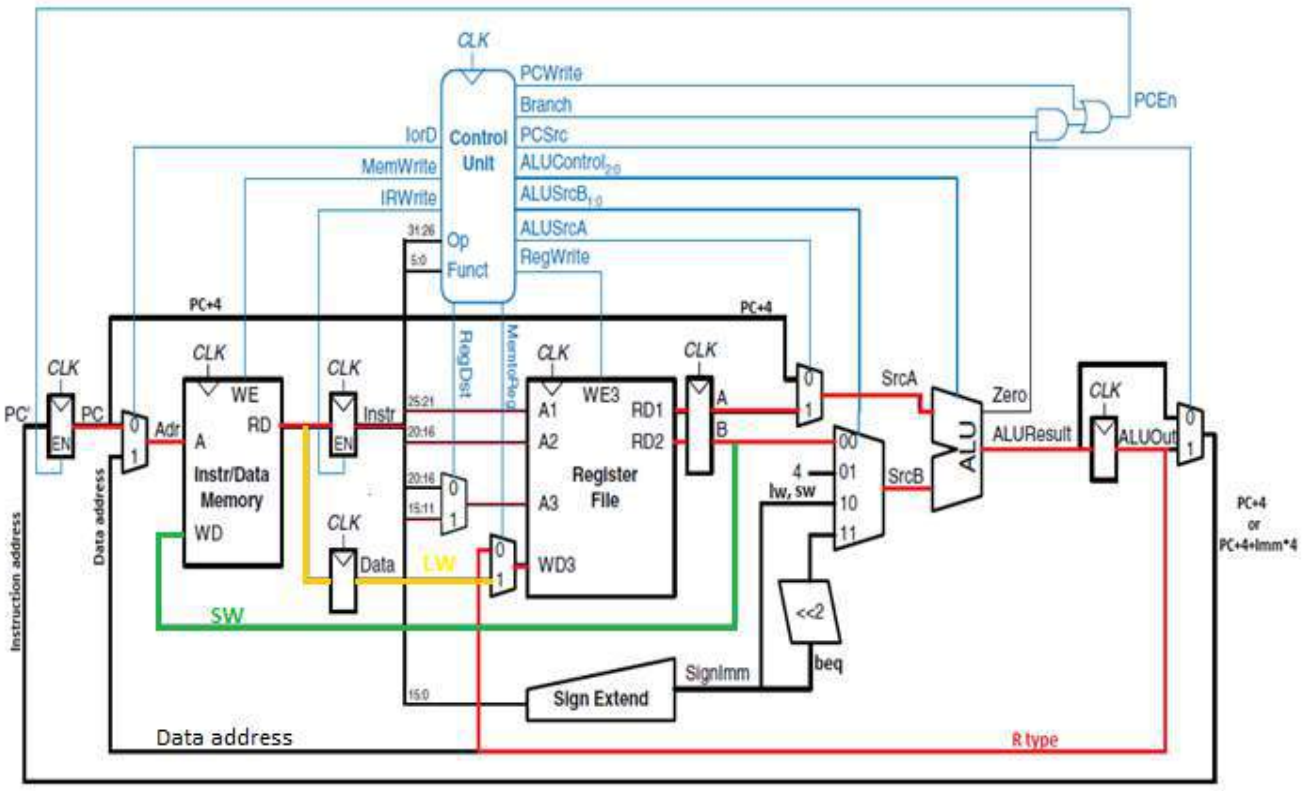
The single-cycle processor has three primary weaknesses.

1. It requires a clock cycle long enough to support the slowest instruction (lw), even though most instructions are faster.
2. It requires three adders (one in the ALU and two for the PC logic); adders are relatively expensive circuits, especially if they must be fast.
3. It has separate instruction and data memories, because both instruction and data accesses needed within same clock cycle. Most computers have a single large memory that holds both instructions and data and that can be read and written.

The multicycle processor addresses these weaknesses by

- Breaking an instruction into multiple shorter steps. In each short step, the processor can read or write the memory or register file or use the ALU. Different instructions use different numbers of steps, so simpler instructions can complete faster than more complex ones.
- The processor needs only one adder; this adder is reused for different purposes on various steps.
- The processor uses a combined memory for instructions and data. The instruction is fetched from memory on the first step, and data may be read or written on later steps.

In multi cycle implementation introduce **additional Registers** in every stages of the datapath to buffer each step. These "**non-architectural register**" are not accessible to programmer. In multi-cycle, same ALU used twice, in distinct cycles!

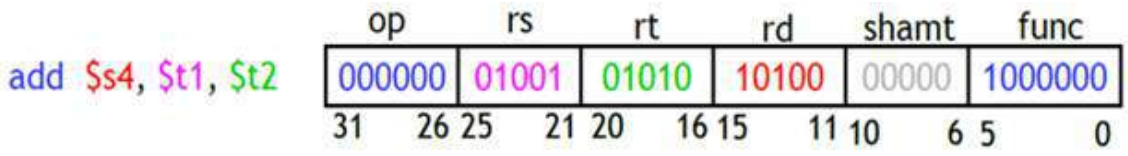


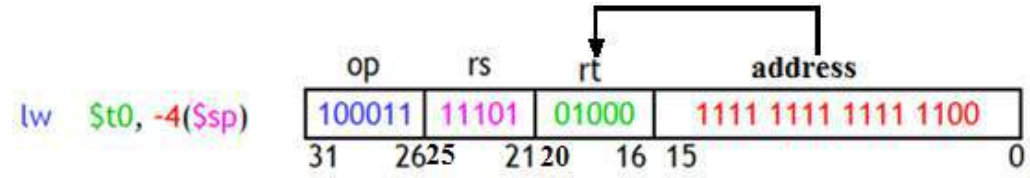

Above figure showing the control signals and Datapath (multicycle) for the

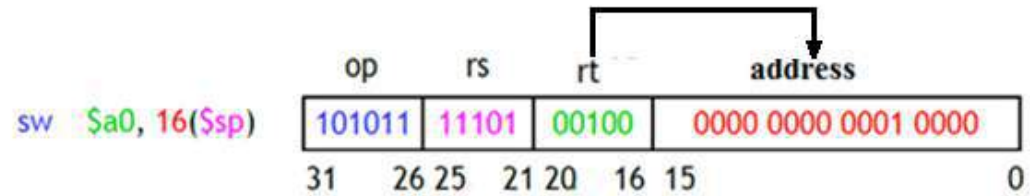
- Instructions lw, sw, R-type and beq
- Calculating instruction addresses
  i) (PC+4) - for normal ii) (PC+4+Imm$_{15:0}$*4) – for beq instruction

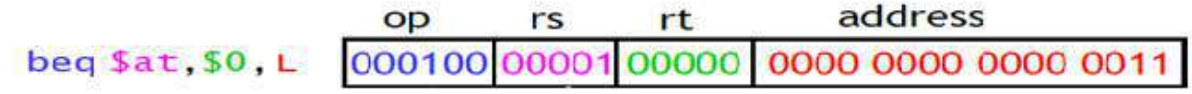
| State element | Input | R-Type instruction | lw instruction | Sw instruction | beq instruction | PC Address* calculation |
|---|---|---|---|---|---|---|
| Instr/data memory | Address in* | - | Data address | Data address | Instruction address | |
| | Data in (WD) | - | - | From 2nd source register (Register file) | - | |
| Register file | Source 1reg | $I_{25:21}$ | $I_{25:21}$ | $I_{25:21}$ | $I_{25:21}$ | |
| | Source 2 reg | $I_{20:16}$ | - | - | $I_{20:16}$ | |
| | Destn reg* | $I_{15:11}$ | $I_{20:16}$ | $I_{20:16}$ (source) | - | |
| | RegData in* | From ALUout | From Data memory | - | - | |
| ALU | Input1,SrcA* | Data out1 of register file | Data out1 of register file | Data out1 of register file | Data out1 of register file | PC+4 |
| | Input2,SrcB** | Data out2 of register file | $I_{15:0}$ (sign extnd) | $I_{15:0}$ (sign extnd) | Data out2 of register file | 4 (normal) or $Imm_{15:0}$*4 (beq) |
| | ALUresult | Result of the operation | Data memory address | Data memory address | Instruction memory address | |

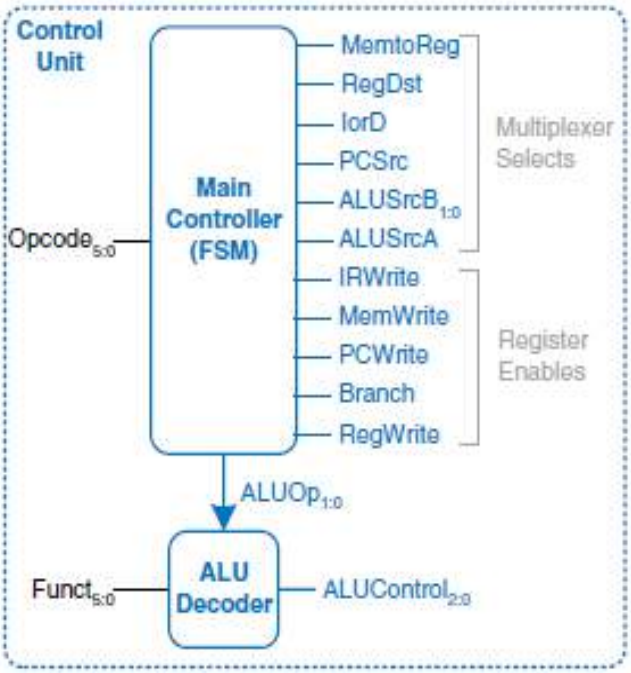
Note:- * need 2:1 mux – (address in, Dest reg, RegData in, ALUSrcA, PC address in)
** need 4:1 mux – (ALUSrcB)






**Control Unit** (multicycle microarchitecture)

MemtoReg =1, data is moving from memory to Reg (lw)
=0, data is moving from ALUresult to reg (R type)

RegDst =1, destination register is selected by the bits $I_{15:11}$
=0, by bits $I_{20:16}$

IorD =0, Instruction address
=1, Data address

PCSrc =0, PC+4 (normal)
=1, PC+4+$Imm_{15:0}$

ALUSrcA =0, PC+4
= 1, register dataOut1

ALUSrcB =00, register dataOut2
=01, 4
=10, signextImm$_{15:0}$
=11, signextImm$_{15:0}$*4

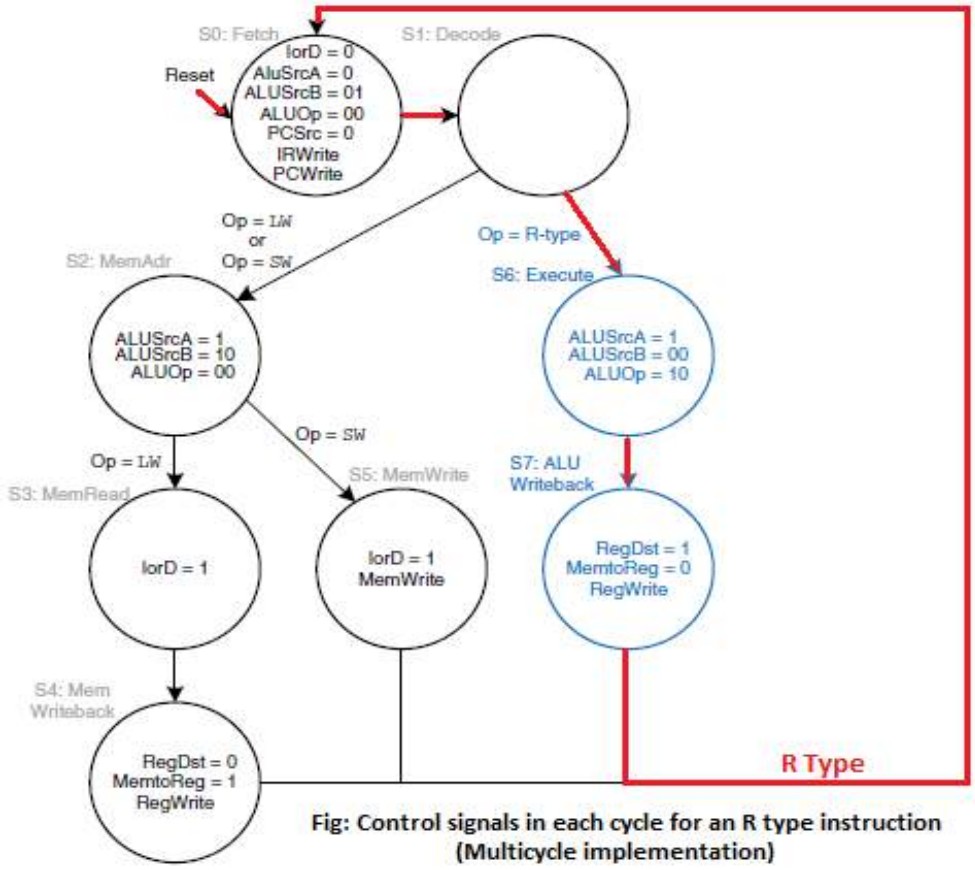



**Fig: Control signals in each cycle for an R type instruction (Multicycle implementation)**

**PERFORMANCE ANALYSIS**

- The cost depends on the amount of hardware required and the implementation technology.
- There are many ways to measure the performance of a computer; the only gimmick-free way to measure performance is by measuring the execution time of a program of interest to you. The computer that executes your program fastest has the highest performance. The next best choice is to measure the total execution time of a collection of programs that are similar to those you plan to run. Such collections of programs are called *benchmarks*, and the execution times of these programs are commonly published to give some indication of how a processor performs.
- The execution time of a program, measured in seconds, is given by

$$Execution\ time = (\#instructions)(\frac{Cycles}{Instructions})(\frac{seconds}{cycle})$$

The number of instructions in a program depends on the processor architecture.

Some architecture has complicated instructions that do more work per instruction, thus reducing the number of instructions in a program. However, these complicated instructions are often slower to execute in hardware.

The number of instructions also depends enormously on the cleverness of the programmer.

The number of cycles per instruction, often called ***CPI***, is the number of clock cycles required to execute an average instruction.

The number of seconds per cycle is the clock period, ***Tc.*** The clock period is determined by the critical path through the logic on the processor.

**Performance analysis of single cycle microarchitecture**

Each instruction in the single-cycle processor takes one clock cycle, so the CPI is 1. The critical path for the lw instruction.

$$T_c = t_{pcq\_PC} + t_{mem} + \max[t_{RFread}, t_{sext}] + t_{mux} + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$$

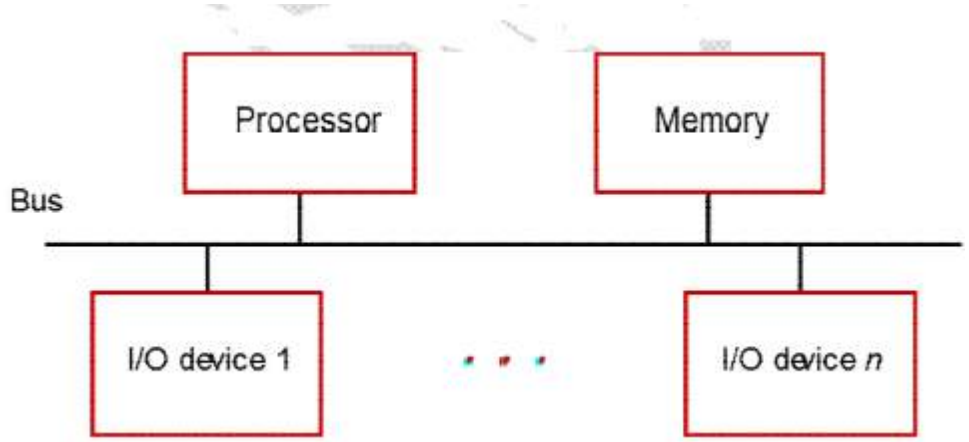
**Performance analysis of multi cycle microarchitecture**

The execution time of an instruction depends on both the number of cycles it uses and the cycle time. Whereas the single-cycle processor performed all instructions in one cycle, the multicycle processor uses varying numbers of cycles for the various instructions. However, the multicycle processor does less work in a single cycle and, thus, has a shorter cycle time.

The multicycle processor requires three cycles for beq and j instructions, four cycles for sw, addi, and R-type instructions, and five cycles for lw instructions. The CPI depends on the relative likelihood that each instruction is used.

**MODULE 5**

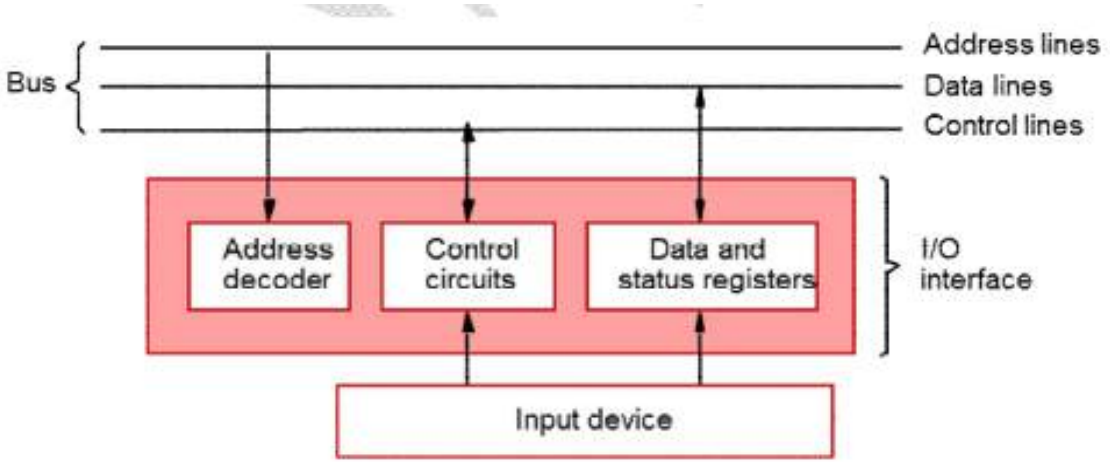
**Accessing I/O Devices**




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

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

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




• I/O device is connected to the bus using an I/O interface circuit which has: - Address decoder, control circuit, and data and status registers.
• Address decoder decodes the address placed on the address lines thus enabling the device to recognize its address.
• Data register holds the data being transferred to or from the processor.
• Status register holds information necessary for the operation of the I/O device.
• Data and status registers are connected to the data lines, and have unique addresses.
• I/O interface circuit coordinates I/O transfers.

**Modes Of Data Transfer**

The rate of transfer to and from I/O devices is slower than the speed of the processor. This creates the need for mechanisms to synchronize data transfers between them. Three techniques for this pupose are

- Programmed I/O
- Interrupt-driven I/O
- Direct memory access (DMA)

With programmed I/O, data are exchanged between the processor and the I/O module. The processor executes a program that gives it direct control of the I/O operation, including sensing device status, sending a read or write command, and transferring the data. When the processor issues a command to the I/O module, it must wait until the I/O operation is complete. If the processor is faster than the I/O module, this is wasteful of processor time. With interrupt driven I/O, the processor issues an I/O command, continues to execute other instructions, and is interrupted by the I/O module when the latter has completed its work. With both programmed and interrupt I/O, the processor is responsible for extracting data from main memory for output and storing data in main memory for input. The alternative is known as direct memory access (DMA).

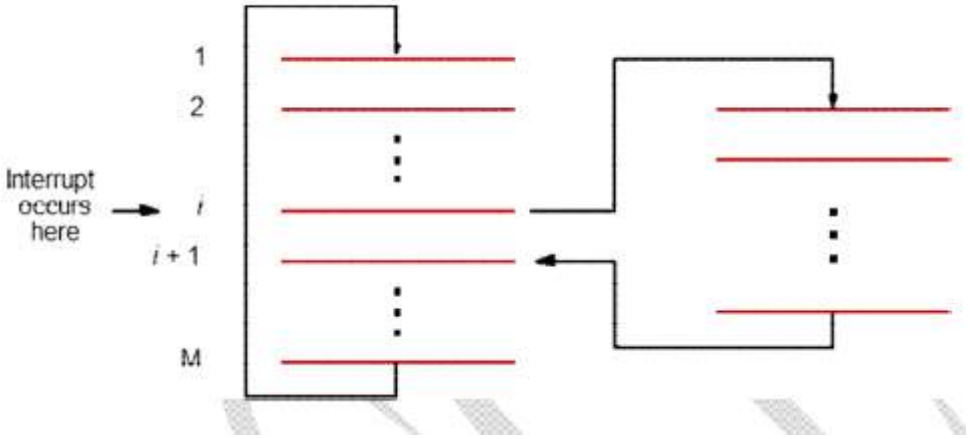
**Programmed I/O**

Processor repeatedly monitors a status flag to achieve the necessary synchronization. Processor polls the I/O device. When the processor is executing a program and encounters an instruction relating to I/O, it executes that instruction by issuing a command to the appropriate I/O module.

With programmed I/O, the I/O module will perform the requested action and then set the appropriate bits in the I/O status register it is the responsibility of the processor periodically to check the status of the I/O module until it finds that the operation is complete. There are four types of I/O commands that an I/O module may receive when it is addressed by a processor:
**Control:** Used to activate a peripheral and tell it what to do.
**Test:** Used to test various status conditions associated with an I/O module and its peripherals
**Read: C**auses the I/O module to obtain an item of data from the peripheral and place it in an internal buffer
**Write:** Causes the I/O module to take an item of data (byte or word) from the data bus and subsequently transmit that data item to the peripheral.

**Interrupt driven I/O**
In program-controlled I/O, when the processor continuously monitors the status of the device, it does not perform any useful tasks. An alternate approach would be for the I/O device to alert the processor when it becomes ready. This is done by sending a hardware signal called an interrupt to the processor. At least one of the bus control lines, called an interrupt-request line is dedicated for this purpose. Processor can perform other useful tasks while it is waiting for the device to be ready.

**Interrupt Processing**




• Processor is executing the instruction located at address i when an interrupt occurs.
• Routine executed in response to an interrupt request is called the interrupt-service routine.
• When an interrupt occurs, control must be transferred to the interrupt service routine.
• But before transferring control, the current contents of the PC (i+1), must be saved in a known location.
• This will enable the return-from-interrupt instruction to resume execution at i+1.
• Return address, or the contents of the PC are usually stored on the processor stack.
The occurrence of an interrupt triggers a number of events, both in the processor hardware and in software.
When an I/O device completes an I/O operation, the following sequence of hardware events occurs:
1. The device issues an interrupt signal to the processor.

2. The processor finishes execution of the current instruction before responding to the interrupt.
3. The processor tests for an interrupt, determines that there is one, and sends an acknowledgment signal to the device that issued the interrupt. The acknowledgment allows the device to remove its interrupt signal.
4. The processor now needs to prepare to transfer control to the interrupt routine. To begin, it needs to save information needed to resume the current program at the point of interrupt. The minimum information required is (a) the status of the processor, which is contained in a register called the program status word (PSW), and (b) the location of the next instruction to be executed, which is contained in the program counter. These can be pushed onto the system control stack
5. The processor now loads the program counter with the entry location of the interrupt-handling program that will respond to this interrupt. Once the program counter has been loaded, the processor proceeds to the next instruction cycle, which begins with an instruction fetch. Because the instruction fetch is determined by the contents of the program counter, the result is that control is transferred to the interrupt-handler program. The execution of this program results in the following operations:
6. At this point, the program counter and PSW relating to the interrupted program have been saved on the system stack. However, there is other information that is considered part of the "state" of the executing program. In particular, the contents of the processor registers need to be saved, because these registers may be used by the interrupt handler
7. The interrupt handler next processes the interrupt. This includes an examination of status information relating to the I/O operation or other event that caused an interrupt.
8. When interrupt processing is complete, the saved register values are retrieved from the stack and restored to the registers
9. The final act is to restore the PSW and program counter values from the stack. As a result, the next instruction to be executed will be from the previously interrupted program.

**Interrupt Hardware**

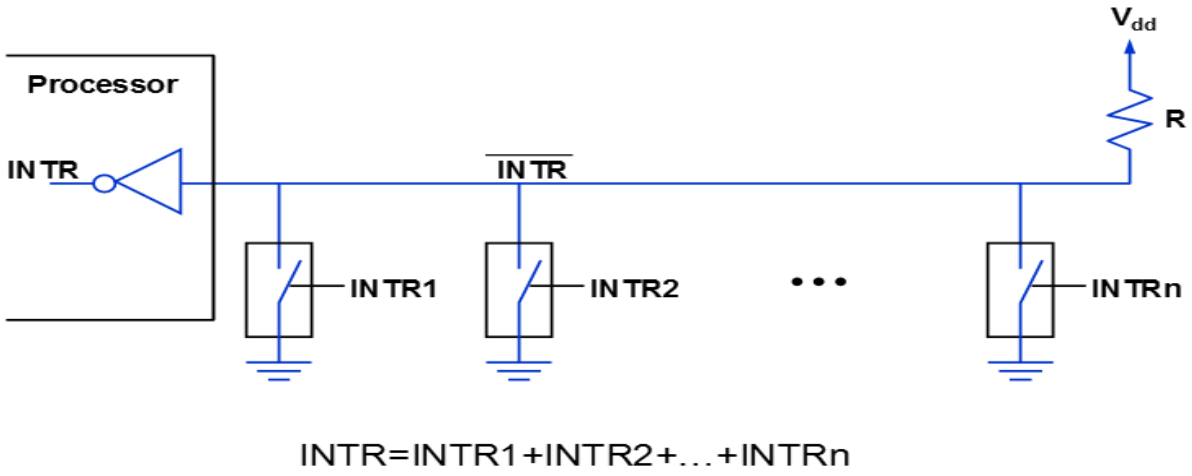



Fig: An equivalent circuit for an open-drain bus used to implement a common interrupt-request line

All the devices are connected to interrupt request line via switches to ground. To request an interrupt, a device closes its associated switch. Then the voltage on INTR drops to 0. If all the interrupts are inactive, the voltage on INTR line is VDD.

**Handling Multiple Devices**

Multiple I/O devices may be connected to the processor and the memory via a bus. Some or all of these devices may be capable of generating interrupt requests. The processor determines which device needs to be served based on one of the following approaches

**Polling scheme:**

• If the processor uses a polling mechanism to poll all of the I/O devices to determine which device is requesting an interrupt.
• In this case the priority is determined by the order in which the devices are polled.
• The first device with status bit set to 1 is the device whose interrupt request is accepted.

**Vectored Interrupts**

• Device requesting an interrupt identifies itself directly to the processor
• The device sends a special code to the processor over the bus.
• The code contains the
– Identification of the device,
– starting address for the ISR,
– Address of the branch to the ISR
• PC finds the IS R address from the code.

**Interrupt Nesting**

Multiple Priority Scheme

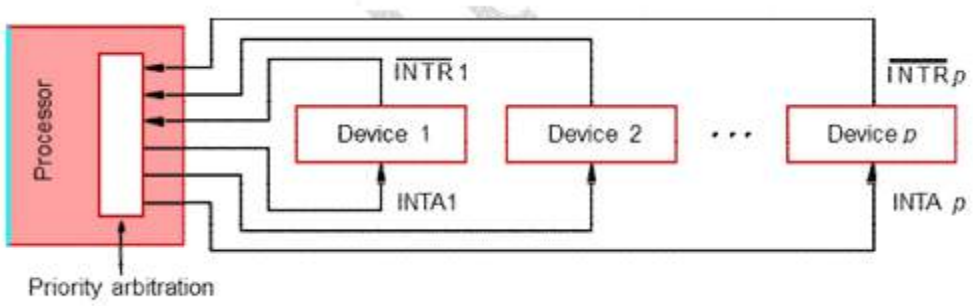


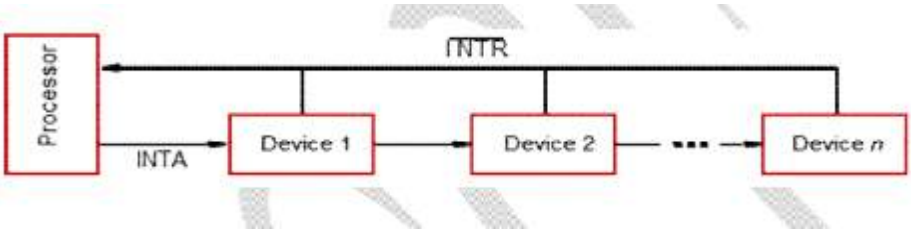

Fig: Implementation of Interrupt priority using individual interrupt-request and acknowledge lines.

• Each device has a separate interrupt-request and interrupt-acknowledge line.
• Each interrupt-request line is assigned a different priority level.
• Interrupt requests received over these lines are sent to a priority arbitration circuit in the processor.

• If the interrupt request has a higher priority level than the priority of the processor, then the request is accepted.

**Simultaneous Requests**

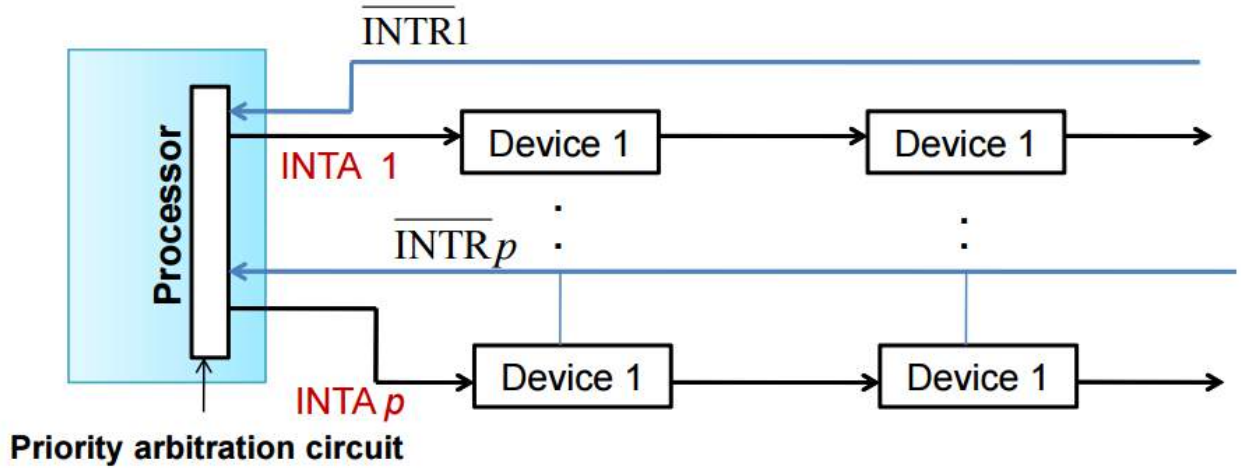
Daisy chain scheme:




• Devices are connected to form a daisy chain.
• Devices share the interrupt-request line, and interrupt-acknowledge line is connected to form a daisy chain.
• When devices raise an interrupt request, the interrupt-request line is activated.
• The processor in response activates interrupt-acknowledge.
• Received by device 1, if device 1 does not need service, it passes the signal to device 2.
• Device that is electrically closest to the processor has the highest priority.

Daisy Chaining with Priority Group

• Combining Daisy chaining and Interrupt nesting to form priority group
• Each group has different priority levels and within each group devices are connected in daisy chain way

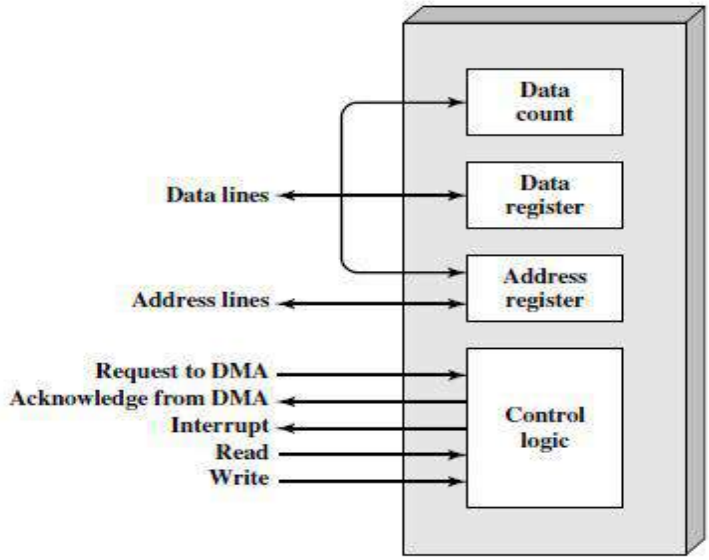



**Arrangement of priority groups**

**Direct Memory Access**

When large volumes of data are to be moved, a more efficient technique is required: direct memory access (DMA). The DMA module transfers the entire block of data, one word at a time, directly to or from memory, without going through the processor. A special control unit may be provided to transfer a block of data directly between an I/O device and the main memory, without continuous intervention by the processor. Control unit which performs these transfers is a part of the I/O device's interface circuit. This control unit is called as a DMA controller.DMA controller performs functions that would be normally carried out by the processor: For each word, it provides the memory address and all the control signals. To transfer a block of data, it increments the memory addresses and keeps track of the number of transfers**.**

DMA Function




When the processor wishes to read or write a block of data, it issues a command to the DMA module, by sending to the DMA module the following information:

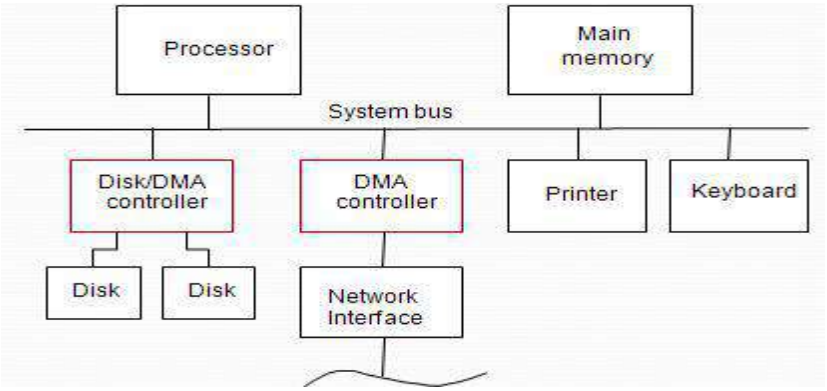
 whether a read or write is requested, using the read or write control line between the processor and the DMA module

 The address of the I/O device involved, communicated on the data lines

 The starting location in memory to read from or write to, communicated on the data lines and stored by the DMA module in its address register

 The number of words to be read or written, again communicated via the data lines and stored in the data count register

The DMA module transfers the entire block of data, one word at a time, directly to or from memory, without going through the processor. When the transfer is complete, the DMA module sends an interrupt signal to the processor. Thus, the processor is involved only at the beginning and end of the transfer.




Let us consider a memory organization with two DMA controllers. In this memory organization, a DMA controller is used to connect a high speed network to the computer bus.
Processor and DMA controllers both need to initiate data transfers on the bus and access main memory.

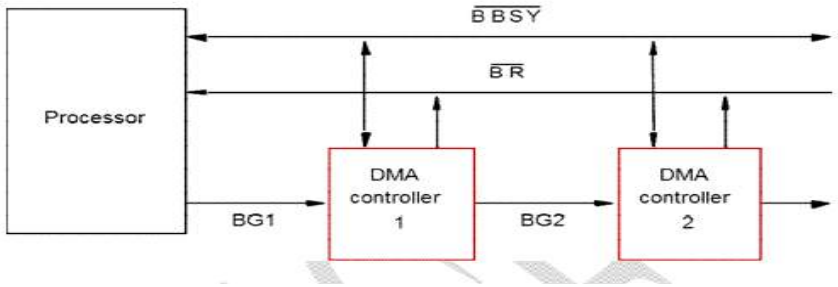
Cycle Stealing And Burst Mode

Memory access by the processor and DMA controllers are interwoven. Requests by DMA devices for using the bus are always given high priority than processor requests. Since the processor originates most memory access cycles, the DMA controller can be said to steal memory cycles from the processor and hence called cycle stealing.
The DMA controller is alternator given exclusive access to the main memory to transfer a block of data without interruption. This is called block or burst mode.

Bus Arbitration
The device that is allowed to initiate transfers on the bus at any given time is called the bus master. When the current bus master releases control of the bus, another device can acquire the status of the bus master. The process by which the next device to become the bus master is selected and bus mastership is transferred to it is called bus arbitration. There are two types of bus arbitration processes.
(1)Centralized arbitration and (2) distributed arbitration.

- In case of centralized arbitration, a single bus arbiter performs the arbitration.
- In case of distributed arbitration all devices which need to initiate data transfers on the bus participate or are involved in the selection of the next bus master.

Centralized Bus Arbitration



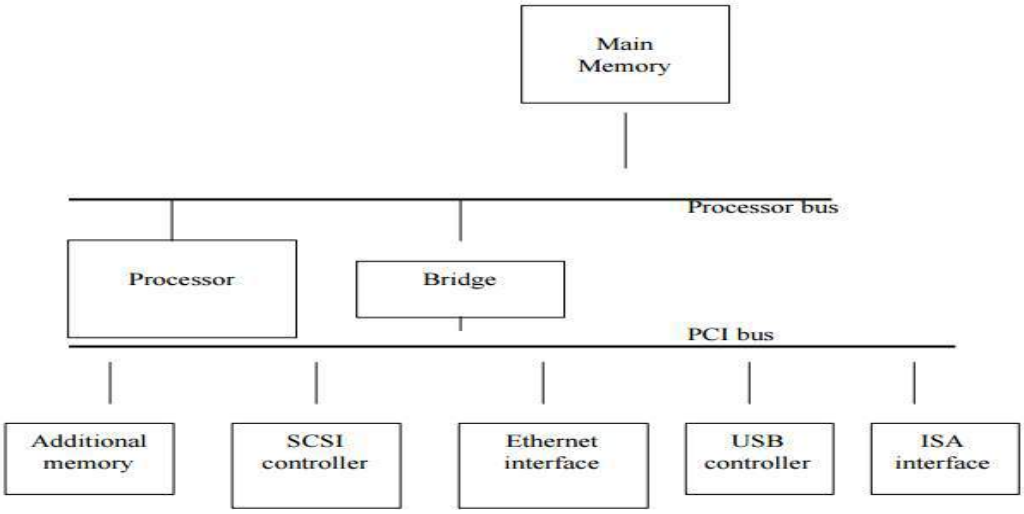
• Bus arbiter may be the processor or a separate unit connected to the bus.
• Normally, the processor is the bus master, unless it grants bus membership to one of the DMA controllers.
• DMA controller requests the control of the bus by asserting the Bus Request (BR) line.
• In response, the processor activates the Bus-Grant1 (BG1) line, indicating that the controller may use the bus when it is free.
• BG1 signal is connected to all DMA controllers in a daisy chain fashion.
• BBSY signal is 0, it indicates that the bus is busy. When BBSY becomes 1, the DMA controller which asserted BR can acquire control of the bus.

Distributed arbitration

All devices waiting to use the bus share the responsibility of carrying out the arbitration process. Arbitration process does not depend on a central arbiter and hence distributed arbitration has higher reliability. Each device is assigned a 4-bit ID number. All the devices are connected using 5 lines, 4 arbitration lines to transmit the ID, and one line for the Start-Arbitration signal. To request the bus a device: Asserts the Start-Arbitration signal. Places its 4-bit ID number on the arbitration lines. The request that has the highest ID number ends up having the highest priority.

**Standard I/O Interfaces-**

.



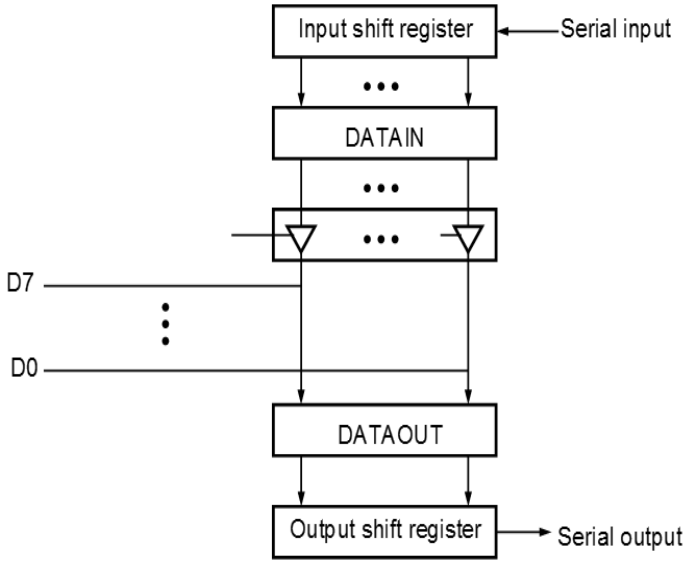

- The processor bus is the bus defined by the signals on the processor chip itself.
- Devices that require a very high speed connection to the processor, such as the main memory, may be connected directly to this bus.
- The motherboard usually provides another bus that can support more devices. The two buses are interconnected by a circuit, which we called a bridge that translates the signals and protocols of one bus into those of the other.

- It is impossible to define uniform standards for the processor bus. The structure of this bus is closely tied to the architecture of the processor.
  The expansion bus is not subject to these limitations, and therefore it can use a standardized signaling structure

**Serial Port**

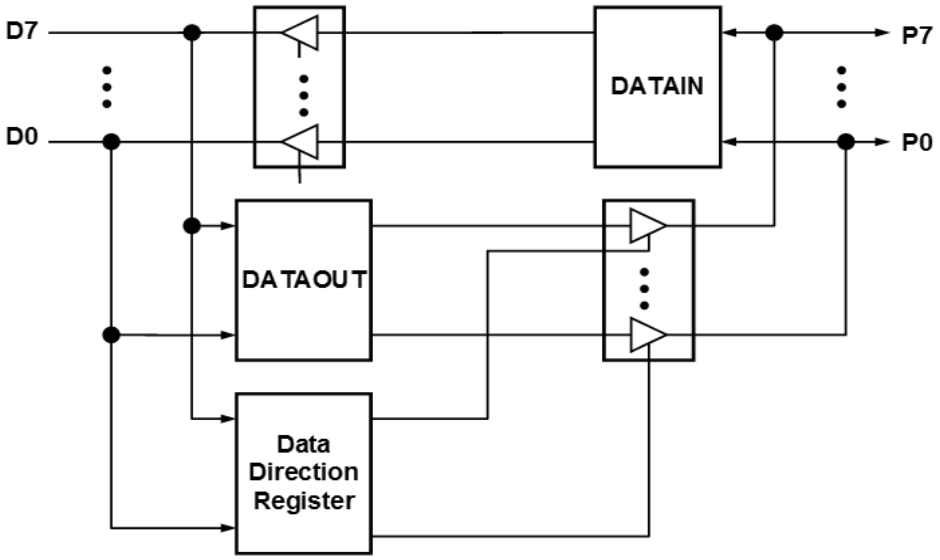
- A serial port is used to connect the processor to I/O devices that require transmission of data one bit at a time.
- The key feature of an interface circuit for a serial port is that it is capable of communicating in a bit-serial fashion on the device side and in a bit-parallel fashion on the bus side
- The transformation between the parallel and serial formats is achieved with shift registers that have parallel access capability.
- Serial transmission is convenient for connecting devices that are physically far away from computer.
- The speed of transmission is often expressed in bit rate-number of bits transferred /second

**A Serial Interface**



## Parallel port

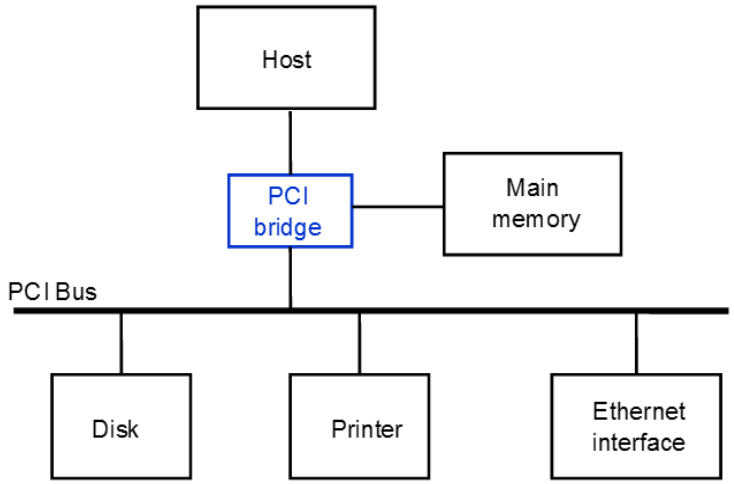
## A General 8-Bit Parallel Interface




A parallel port is a type of interface found on computers for connecting peripherals. The name refers to the way the data is sent, ie, parallel port send multiple bits of data at once as opposed to serial interfaces which send only one bit at a time. To do so, parallel port require multiple data lines in their cables and port connectors and tend to be larger than serial ports

## PCI(Peripheral Component Interconnect) Bus

- Use of a PCI bus in a computer system



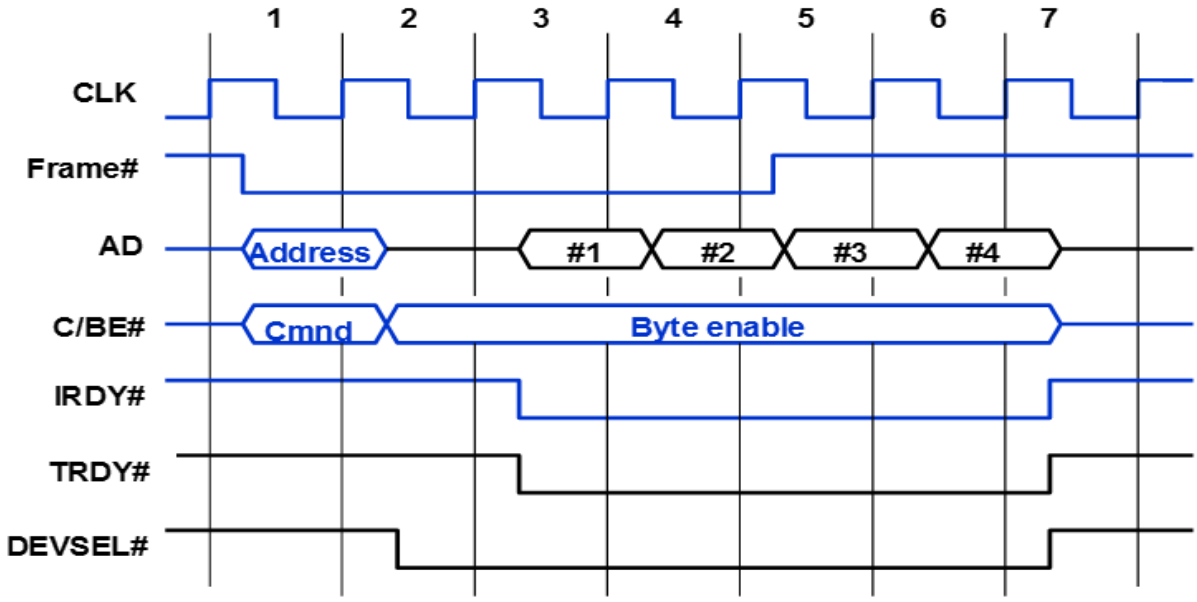

- The bus support three independent address spaces: memory, I/O, and configuration.
- The I/O address space is intended for use with processors, such Pentium, that have a separate I/O address space.

- However, the system designer may choose to use memory-mapped I/O even when a separate I/O
- Address space is available.
- The configuration space is intended to give the PCI its plug-and-play capability. A 4-bit command that accompanies the address identifies which of the three spaces is being used in a given data transfer operation.

**Data Transfer Signals on the PCI Bus**

| Name | Function |
|---|---|
| CLK | A 33-MHz or 66MHz clock |
| FRAME# | Sent by the initiator to indicate the duration of a transaction |
| AD | 32 address/data lines, which may be optionally increased to 64 |
| C/BE# | 4 command/byte-enable lines (8 for 64-bit bus) |
| IRDY#, TRDY# | Initiator-ready and Target-ready signals |
| DEVSEL# | A response from the device indicating that it has recognized its Address and is ready for a data transfer transaction |
| IDSEL# | Initialization Device Select |

A Read Operation on the PCI Bus



## SCSI Bus

The acronym SCSI stands for Small Computer System Interface. It refers to a standard bus defined by the American National Standards Institute (ANSI) under the designation X3.131. In the original specifications of the standard, devices such as disks are connected to a computer via a 50-wire cable, which can be up to 25 meters in length and can transfer data at rates up to 5 megabytes/s.

The SCSI bus standard has undergone many revisions, and its data transfer capability has increased very rapidly, almost doubling every two years. SCSI-2 and SCSI-3 have been defined, and each has several options. A SCSI bus may have eight data lines, in which case it is called a narrow bus and transfers data one byte at a time. Alternatively, a wide SCSI bus has 16 data lines and transfers data 16 bits at a time. There are also several options for the electrical signaling scheme used.

Devices connected to the SCSI bus are not part of the address space of the processor in the same way as devices connected to the processor bus. The SCSI bus is connected to the processor bus through a SCSI controller.

To illustrate the operation of the SCSI bus, let us consider how it may be used with a disk drive. Communication with a disk drive differs substantially from communication with the main memory.

A controller connected to a SCSI bus is one of two types – an initiator or a target. An initiator has the ability to select a particular target and to send commands specifying the operations to be performed. Clearly, the controller on the processor side, such as the SCSI controller, must be able to operate as an initiator. The disk controller operates as a target. It carries out the commands it receives from the initiator. The initiator establishes a logical connection with the intended target. Once this connection has been established, it can be suspended and restored as needed to transfer commands and bursts of data. While a particular connection is suspended, other device can use the bus to transfer information. This ability to overlap data transfer requests is one of the key features of the SCSI bus that leads to its high performance.

Data transfers on the SCSI bus are always controlled by the target controller. To send a command to a target, an initiator requests control of the bus and, after winning arbitration, selects the controller it wants to communicate with and hands control of the bus over to it. Then the controller starts a data transfer operation to receive a command from the initiator.

## Universal Serial Bus (USB)

The USB has been designed to meet several key objectives

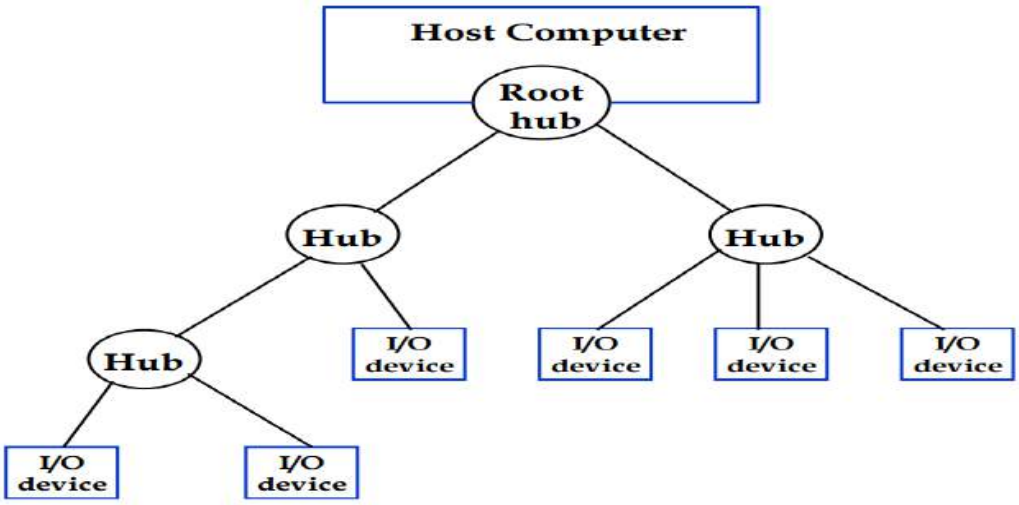
- Provide a simple, low-cost, and easy to use interconnection system that overcomes the difficulties due to the limited number of I/O ports available on a computer
- Accommodate a wide range of data transfer characteristics for I/O devices, including telephone and Internet connections
- Enhance user convenience through a “plug-and-play” mode of operation

### USB Structure

- A serial transmission format has been chosen for the USB because a serial bus satisfies the low-cost and flexibility requirements
- Clock and data information are encoded together and transmitted as a single signal
  - Hence, there are no limitations on clock frequency or distance arising from data skew

- To accommodate a large number of devices that can be added or removed at any time, the USB has the tree structure
  - Each node of the tree has a device called a hub, which acts as an intermediate control point between the host and the I/O device
  - At the root of the tree, a root hub connects the entire tree to the host computer

USB Tree Structure




USB Tree Structure

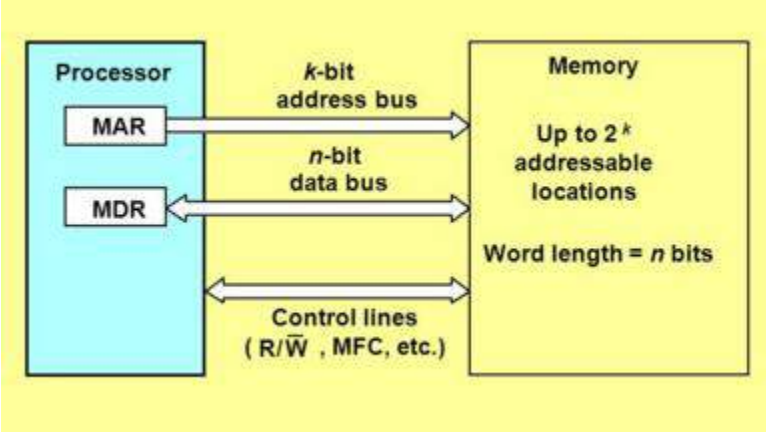
- The tree structure enables many devices to be connected while using only simple point-to-point serial links
- Each hub has a number of ports where devices may be connected, including other hubs
- In normal operation, a hub copies a message that it receives from its upstream connection to all its downstream ports
  - As a result, a message sent by the host computer is broadcast to all I/O devices, but only the addressed device will respond to that message
- A message sent from an I/O device is sent only upstream towards the root of the tree and is not seen by other devices
  - Hence, USB enables the host to communicate with the I/O devices, but it does not enable these devices to communicate with each other.

USB Protocols

- All information transferred over the USB is organized in packets, where a packet consists of one or more bytes of information
- The information transferred on the USB can be divided into two broad categories: control and data
  - Control packets perform such tasks as addressing a device to initiate data transfer, acknowledging that data have been received correctly, or indicating an error.
  - Data packets carry information that is delivered to a device. For example, input and output data are transferred inside data packets.

## Memory Systems

Memory is organized as a collection of cells. Each cell can store 1 bit of information- ‘0’ or ‘1’. The maximum size of the memory that can be used in any computer can be determined by the addressing scheme. Word length is the number of bits that can be transferred to or from the memory. It can be determined from the width of data bus.




The processor communicates with the memory system over a memory interface. The processor sends an address over the address bus to the memory system. For a read, Memwrite is 0 and the memory returns the data on the Read Data bus. For a Write, Memwrite is 1 and the processor sends data to memory on Write Data bus.

**Semiconductor Memories (RAM,ROM,EPROM)**

RAM

One distinguishing characteristic of RAM is that it is possible both to read data from the memory and to write new data into the memory easily and rapidly. Both the reading and writing are accomplished through the use of electrical signals. The other distinguishing characteristic of RAM is that it is volatile. A RAM must be provided with a constant power supply. If the power is interrupted, then the data are lost. Thus, RAM can be used only as temporary storage. The two traditional forms of RAM used in computers are: DRAM and SRAM

- DRAM
  - Low Cost
  - High Density
  - Medium Speed
- SRAM
  - High Speed
  - Ease of use
  - Medium Cost

ROM

A ROM is a form of semiconductor memory technology used where the data is written once and then not changed. In view of this it is used where data needs to be stored permanently, even when the power is removed - many memory technologies lose the data once the power is removed.

As a result, this type of semiconductor memory technology is widely used for storing programs and data that must survive when a computer or processor is powered down. For example the BIOS of a computer will be stored in ROM. As the name implies, data cannot be easily written to ROM. Depending on the technology used in the ROM, writing the data into the ROM initially may require special hardware. Although it is often possible to change the data, this gain requires special hardware to erase the data ready for new data to be written in.

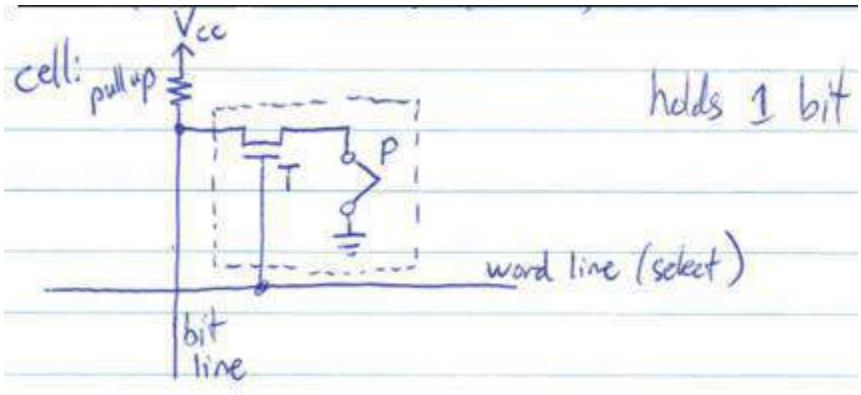



Fig: A ROM Cell

A logic value 0 is stored in the cell if the transistor is connected to ground at point p, otherwise a 1 is stored. To read the state of the cell, the word line is activated. Thus the transistor switch is

closed and voltage on the bit line drops to zero, if there is a connection between transistor and ground. If there is no connection, to ground the bit line remains at high voltage, indicating a 1.

EPROM

The term EPROM is an acronym that stands for 'Erasable Programmable Read Only Memory'. EPROMS were the first reprogrammable Read Only Memory (ROM) used in modern computers. ROMs have their contents 'burned in' and this content cannot be changed, even if there are errors. This makes updates harder with ROM because the contents cannot be changed once it has been 'burned'.

Like ROM, EPROMs are able to hold their value even when the electrical power is off. An EPROM can be erased by exposing it to ultraviolet light and then 'reprogrammed' with new instruction code and data. This makes an EPROM more flexible and much more useful, especially if the device that uses it needs to become smarter over time.

However, unlike ROM the contents of an EPROM can be changed by exposing the chip to higher than ultraviolet light. The EPROM is first programmed at the manufacturing plant by exposing the chip to ultraviolet light of a certain intensity that erases the chip. New program code can then be loaded into the chip, the voltage stepped up to 'burn' the values into the chip and then the chip is ready for use.
Examples of EPROMs you can hold in your hand include *compact flash, smart memory*, *memory sticks* etc.

Flash memory

In a flash device it is possible to read the contents of a single cell, but write the contents of an entire block of cells. Single flash chips are not sufficiently large, so larger memory modules are implemented using flash cards and flash drives.

Flash Cards

A card is simply plugged into a accessible slot. Flash cards come in a variety of memory sizes. Typical sizes are 8, 32 and 64 Mbytes.

Flash Drives

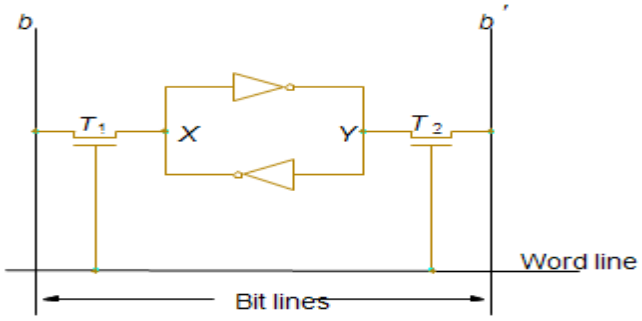
Large flash memory modules have been developed to replace hard disk drives. Flash drives are designed to emulate the hard disks. The capacity of flash drives is less than one gigabytes. Hard disk can store many gigabytes.

## Memory Cells-DRAM and SRAM

SRAM

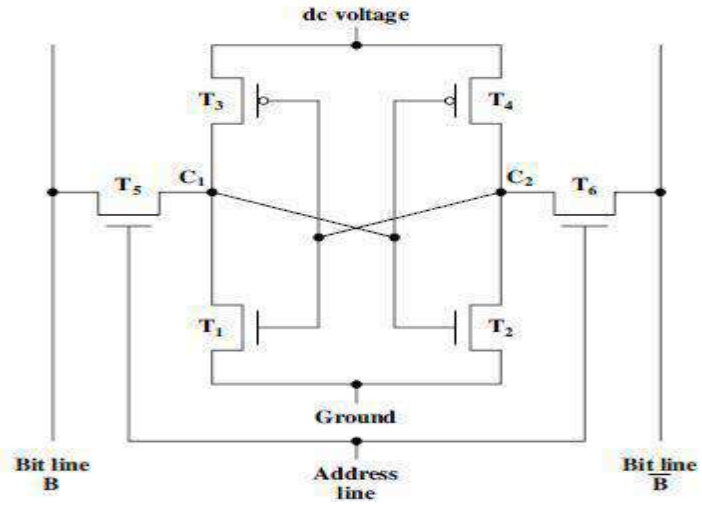
SRAM is a digital device. In a SRAM, binary values are stored using traditional flip-flop logic-gate configurations A static RAM will hold its data as long as power is supplied to it.

SRAM Cell




- Two transistor inverters are cross connected to implement a basic flipflop.
- The cell is connected to one word line and two bits lines by transistors T1and T2.
- When word line is at ground level, the transistors are turned off and the latch retains its state
- Read operation: In order to read state of SRAM cell, the word line is activated to close switches T1 and T2. Sense/Write circuits at the bottom monitor the state of b and b'.
- For a write operation, the desired bit value is applied to line b, while its complement is applied to b'line. The required signals are generated by sense/write circuitry. This forces the transistors into the proper state.

CMOS SRAM Cell is shown below




- Four transistors (T1,T2,T3,T4) are cross connected in an arrangement that produces a stable logic state.
- In logic state 1, point C1 is high and point C2 is low; in this state,T1 and T4 are off and T2 and T3 are on.
- In logic state 0, point C1 is low and point C2 is high; in this state,T1 and T4 are on and T2 and T3 are off.
- Both states are stable as long as the direct current (dc) voltage is applied.

- No refresh is needed to retain data. Advantage of CMOS SRAM-low power Consumption.

DRAM

A dynamic RAM (DRAM) is made with cells that store data as charge on capacitors. The presence or absence of charge in a capacitor is interpreted as a binary 1 or 0. Because capacitors have a natural tendency to discharge, dynamic RAMs require periodic charge refreshing to maintain data storage. The term dynamic refers to this tendency of the stored charge to leak away, even with power continuously applied.

- ❖ Do not retain their state indefinitely.
- ❖ Contents must be periodically refreshed.

DRAM CELL

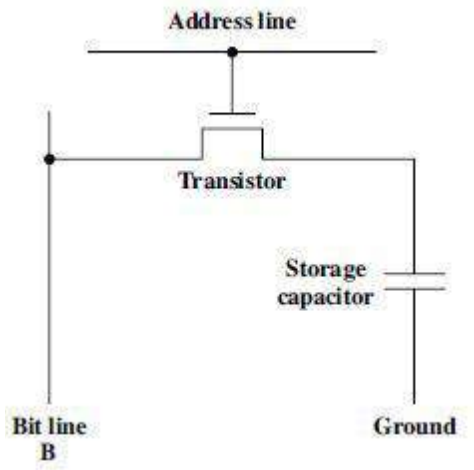



Figure shows a typical DRAM structure for an individual cell that stores1 bit. The address line is activated when the bit value from this cell is to be read or written. The transistor acts as a switch that is closed (allowing current to flow) if a voltage is applied to the address line and open (no current flows) if no voltage is present on the address line.

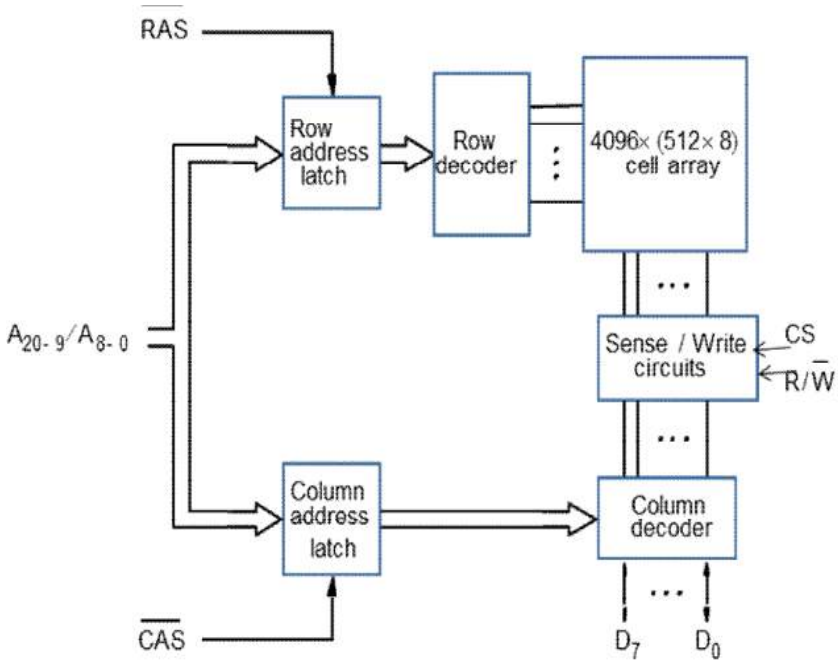
For the write operation, a voltage signal is applied to the bit line; a high voltage represents 1, and a low voltage represents 0. A signal is then applied to the address line, allowing a charge to be transferred to the capacitor.

For the read operation, when the address line is selected, the transistor turns on and the charge stored on the capacitor is fed out onto a bit line and to a sense amplifier. The sense amplifier compares the capacitor voltage to a reference value and determines if the cell contains a logic 1 or a logic 0.

Capacitor voltage is above the reference value, it drives the bit line to logic value 1Capacitor voltage is below the reference value, it drives the bit line to logic value 0.

Although the DRAM cell is used to store a single bit (0 or 1), it is essentially an analog device. The capacitor can store any charge value within a range; a threshold value determines whether the charge is interpreted as 1 or 0.

Asynchronous DRAMs



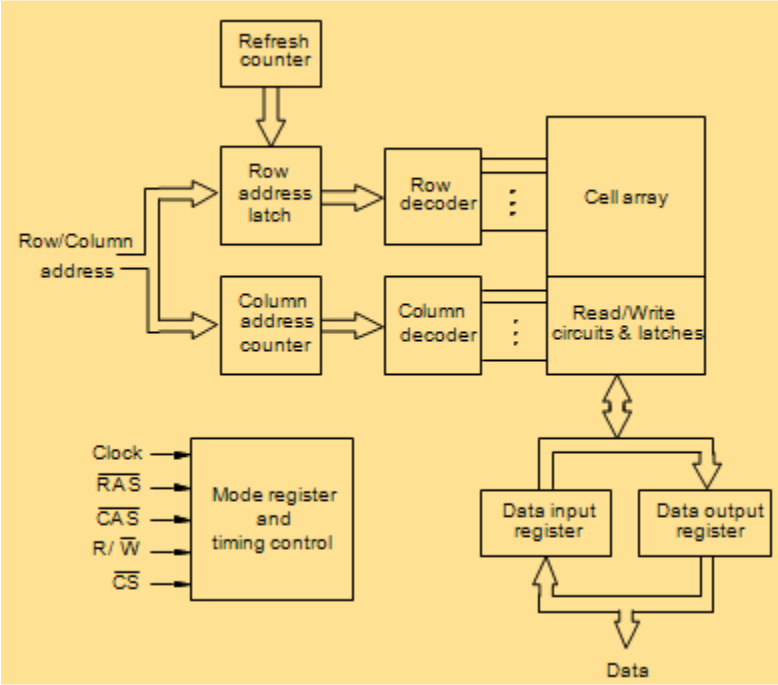

Each row can store 512 bytes. 12 bits to select a row, and 9 bits to select a group in a row. Total of 21 bits.

• First apply the row address; RAS signal latches the row address. Then apply the column address, CAS signal latches the address.

• Timing of the memory unit is controlled by a specialized unit which generates RAS and CAS. This is asynchronous DRAM

If R/W control signal indicates a Read operation, the output values are transferred to data line, $D_{7\text{-}0}$. If R/W control signal indicates a write operation, the information on data line, $D_{7\text{-}0}$ is transferred to the selected circuits.RAS and CAS control signals are active low so that they cause latching of addresses when they change from high to low.

Synchronous DRAMs




• Operation is directly synchronized with processor clock signal.
• During a Read operation, the contents of the cells in a row are loaded onto the latches.
• During a refresh operation, the contents of the cells are refreshed without changing the contents of the latches.
• Data held in the latches correspond to the selected columns are transferred to the output.
• For a burst mode of operation, successive columns are selected using column address counter and clock.CAS signal need not be generated externally. A new data is placed during raising edge of the clock.

**Internal organization of semiconductor RAM memories**

The basic element of a semiconductor memory is the memory cell. Memory cells share certain properties:
• They exhibit two stable (or semi stable) states, which can be used to represent binary 1 and 0.
• They are capable of being written into (at least once), to set the state.
• They are capable of being read to sense the state.

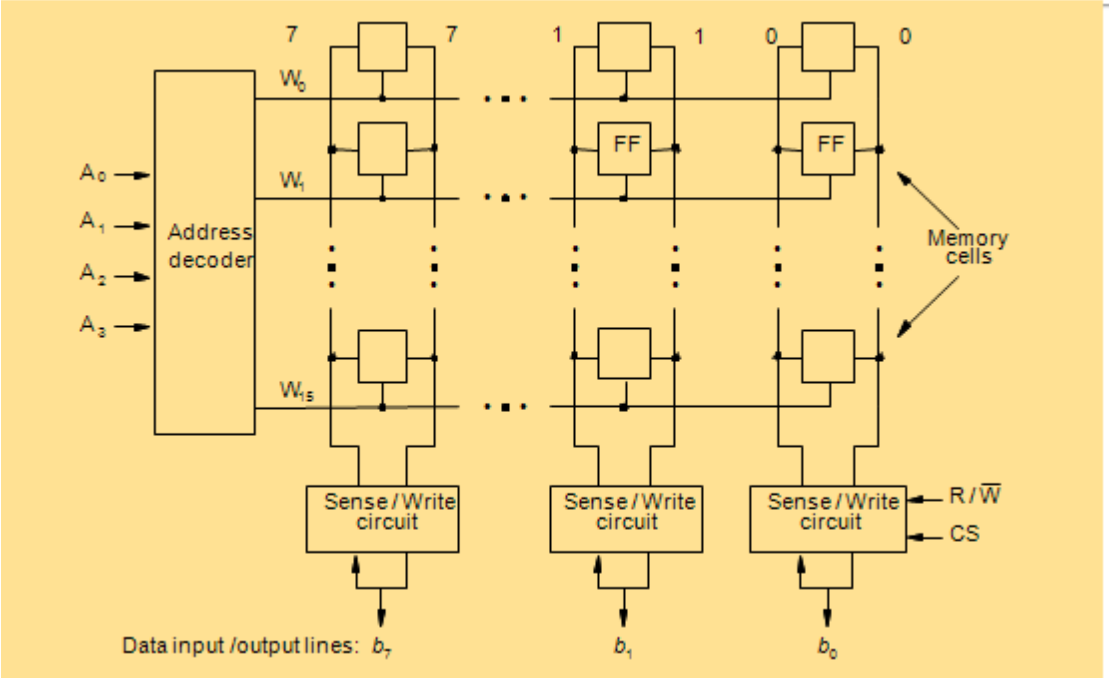


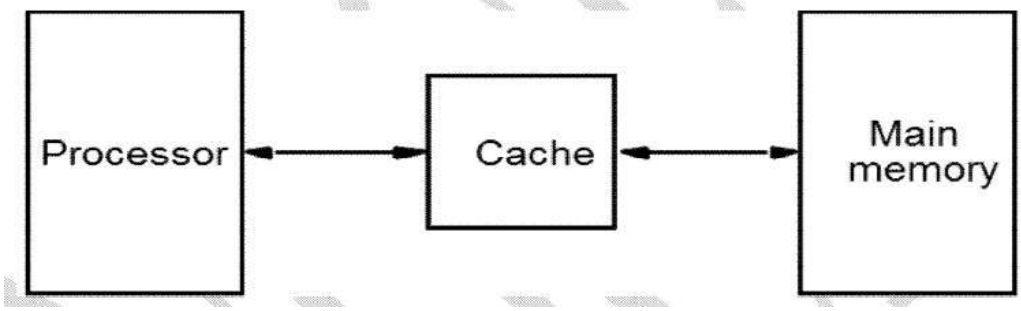

Fig: Organisation of bit cells in a memory chip

Each memory cell can hold one bit of information. Memory cells are organized in the form of an array. One row is one memory word. All cells of a row are connected to a common line, known as the "word line". Word line is connected to the address decoder. Sense/write circuits are connected to the data input/output lines of the memory chip.

**MODULE 6**

**Cache Memory**

Cache memory is an architectural arrangement which makes the main memory appear faster to the processor than it really is. The cache contains a copy of portions of main memory




.

- Cache memory is based on the property of computer programs known as "**locality of reference".**
- Analysis of programs indicates that many instructions in localized areas of a program are executed repeatedly during some period of time, while the others are accessed relatively less frequently. This is called "locality of reference".

  **Temporal locality of reference**:

  Recently executed instruction is likely to be executed again very soon.

  **Spatial locality of reference**:

  Instructions with addresses close to a recently instruction are likely to be executed soon.

Cache memory can store a reasonable no:of blocks at any given time. But this no:of blocks is small compared to main memory. The correspondence between the main memory blocks and those in the cache memory is specified by a mapping function. At any given time, only some blocks in the main memory are held in the cache, which blocks in the main memory are in the cache is determined by a "mapping function".

When the cache is full, and a block of words needs to be transferred from the main memory, some block of words in the cache must be replaced. This is determined by a "replacement algorithm".

If the data is in the cache it is called a Read or Write hit.

• Read hit:

The data is obtained from the cache.

• Write hit:

Cache has a replica of the contents of the main memory.

Contents of the cache and the main memory may be updated simultaneously. This is the **write-through protocol**.

Update the contents of the cache, and mark it as updated by setting a bit known as the dirty bit or modified bit.

The contents of the main memory are updated when this block is replaced. This is **write-back or copy-back protocol**

If the data is not present in the cache, then a Read miss or Write miss occurs.
• Read miss:
Block of words containing this requested word is transferred from the main memory into cache. After the block is transferred, the particular word requested is forwarded to the processor.
The desired word may also be forwarded to the processor as soon as it is transferred without waiting for the entire block to be transferred. This is called load-through or early-restart.
• Write-miss:
Write-through protocol is used, then the contents of the main memory are updated directly.
If write-back protocol is used, the block containing the addressed word is first brought into the cache. The desired word is overwritten with new information.

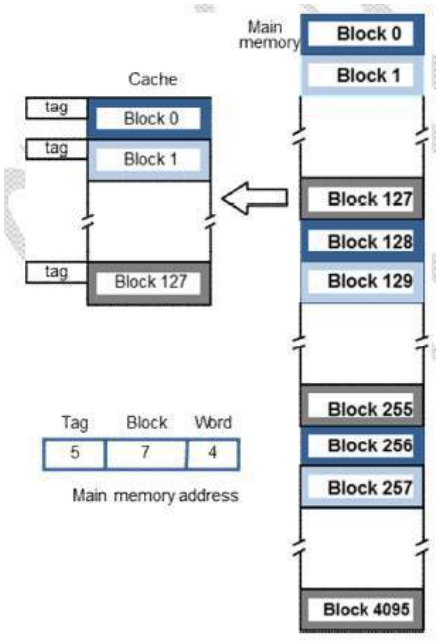
Cache Coherence Problem
A bit called as "valid bit" is provided for each block. If the block contains valid data, then the bit is set to 1, else it is 0. Data transfers between main memory and disk occur directly by passing the cache. When the data on a disk changes, the main memory block is also updated. If the data in the disk and main memory changes and the write-back protocol is being used, the data in the cache may also have changed and is indicated by the dirty bit.
The copies of the data in the cache, and the main memory are different. This is called the cache coherence problem.

**Mapping function**
Mapping functions determine how memory blocks are placed in the cache.

A simple processor example:
- Cache consisting of 128 blocks of 16 words each.
- Total size of cache is 2048 (2K) words.
- Main memory is addressable by a 16-bit address.
- Total size of main memory is 4K words
- Main memory has 4K blocks of 16 words each.
- Main memory has 64K words, view as 4K blocks of 16 words each.

**Three mapping functions:**
- Direct mapping
- Associative mapping
- Set-associative mapping.

**Direct mapping**




• Block j of the main memory maps to j modulo 128 of the cache. One of the main memory blocks 0,128,256... maps to 0, Blocks 1,129,257.... maps to 1,and so on.
• More than one memory block is mapped onto the same position in the cache.
• May lead to contention for cache blocks even if the cache is not full.
• Resolve the contention by allowing new block to replace the old block, leading to a trivial replacement algorithm.
Memory address is divided into three fields:

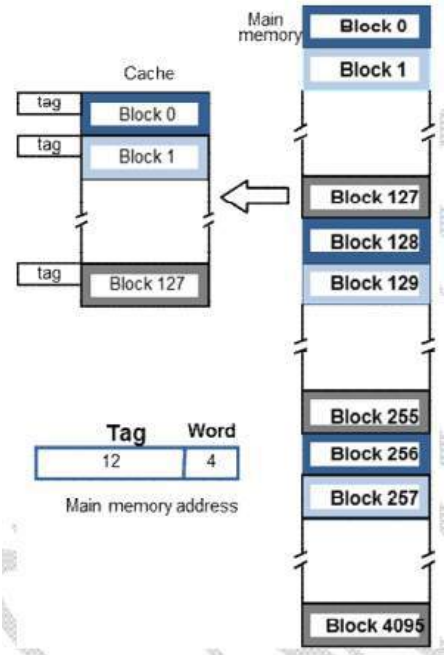
- Low order 4 bits determine one of the 16 words in a block.
- When a new block enters the cache, the next 7 bits determine the cache position in which this new block must be stored.
- High order 5 bits determine which of the possible 32 blocks(4096/128) is currently present in the cache. These are tag bits.

Simple to implement but not very flexible.

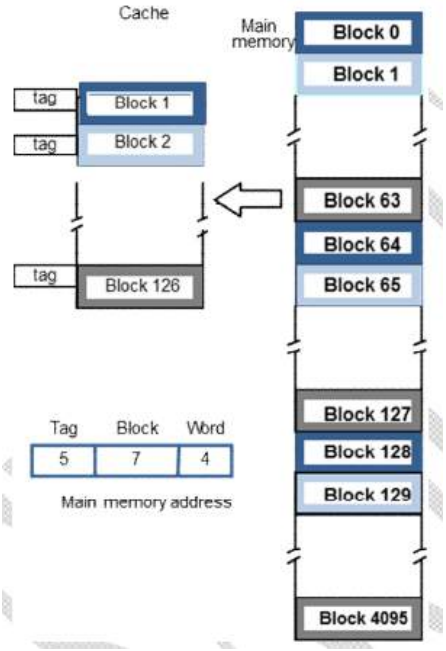
**Associative mapping**

• Main memory block can be placed into any cache position.

- Memory address is divided into two fields:
- Low order 4 bits identify the word within a block.
- High order 12 bits or tag bits identify a memory block when it is resident in the cache.
- Replacement algorithms can be used to replace an existing block in the cache when the cache is full.
- Flexible, and uses cache space efficiently.
- Cost is higher than direct-mapped cache because of the need to search all 128 patterns to determine whether a given block is in the cache.




**Set-Associative mapping**




- Set-associative mapping is a combination of direct and associative mapping.
- Blocks of cache are grouped into sets.
- Mapping function allows a block of the main memory to reside in any block of a specific set.
- Here a two way set associative mapping is implemented.
- Divide the cache into 64 sets, with two blocks per set.

- Memory block 0, 64, 128 etc. map to block 0, and they can occupy either of the two positions.
- Memory address is divided into three fields:
  - Lower order 4 bit field
  - 6 bit set field determines which set of the cache might contain the desired block.
  - 6 bit tag field are compared to the tag fields of the two blocks of the set to check if the desired block is present.

**Replacement algorithm**

Once the cache has been filled, when a new block is brought into the cache, one of the existing blocks must be replaced.

For direct mapping, there is only one possible line for any particular block, and no choice is possible.

For the associative and set associative techniques, a replacement algorithm is needed.

To achieve high speed, such an algorithm must be implemented in hardware. The most effective algorithm is **least recently used (LRU):**

**least recently used (LRU):** Replace that block in the set that has been in the cache longest with no reference to it.

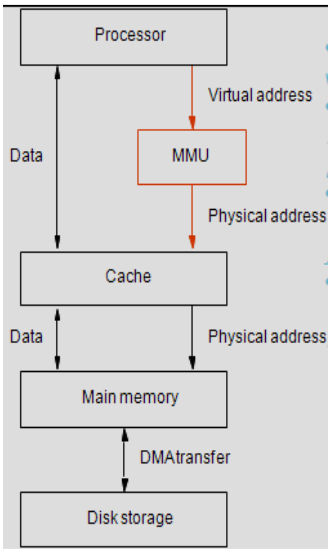
To use the LRU algorithm, the cache controller must track references to all blocks as computation proceeds. Suppose it is required to track the LRU block of a four-block set in asset associative cache. A 2 bit counter can be used for each block. When a hit occurs , the counter of the block that is referenced is set to 0. Counters with values originally lower than the referenced one are incremented by 1, and all others remain unchanged. When a miss occurs and set is not full, the counter associated with new block loaded to cache from main memory is set to 0, and the values of all other counters are increased by 1.When a miss occurs and set is full, the block with the counter value 3 (least referenced block) is removed, the new block is put in place and its counter is set to 0. The other blocks are implemented by one. It can be easily verified that the counter values of occupied blocks are always distinct.

Another possibility is **first-in-first-out (FIFO**): Replace that block in the set that has been in the cache longest. FIFO is easily implemented as a round robin or circular buffer technique.

## Virtual memory

- Virtual memory is an architectural solution to increase the effective size of the memory system.
- Physical main memory in a computer is generally not as large as the entire possible addressable space.
- Physical memory typically ranges from a few hundred megabytes to 1G bytes.
- Large programs that cannot fit completely into the main memory have their parts stored on secondary storage devices such as magnetic disks. Pieces of programs must be transferred to the main memory from secondary storage before they can be executed.
- Techniques that automatically move program and data between main memory and secondary storage when they are required for execution are called virtual-memory techniques.
- Programs and processors reference an instruction or data independent of the available physical main memory space. The binary addresses that the processor issues for either instructions or data are called logical or virtual addresses. Virtual addresses are translated into physical addresses by a combination of hardware and software subsystems.
- If virtual address refers to a part of the program that is currently in the main memory, it is accessed immediately.
- If the address refers to a part of the program that is not currently in the main memory, it is first transferred to the main memory before it can be used.

## Virtual memory organization




- Memory management unit (MMU) translates virtual addresses into physical addresses. MMU causes the operating system to bring the data from the secondary storage into the main memory

- If the desired data or instructions are in the main memory they are fetched as described previously. If the desired data or instructions are not in the main memory, they must be transferred from secondary storage to the main memory.

**MMU(Memory Management Unit)**

- The Memory Management Unit (MMU) translates logical or virtual addresses into physical addresses.
- MMU uses the contents of the page table base register to determine the address of the page table to be used in the translation.
- Changing the contents of the page table base register can enable us to use a different page table, and switch from one space to another.
- At any given time, the page table base register can point to one page table. Thus, only one page table can be used in the translation process at a given time.
- Pages belonging to only one space are accessible at any given time.

**Address translation**

- A simple method for translating virtual addresses into physical addresses
- All programs and data are composed of fixed length units called pages, each of which consists of a block of words that occupy continuous locations in the main memory.
- Page is a basic unit of information that is transferred between secondary storage and main memory.
- Size of a page commonly ranges from 2K to 16K bytes.
- Pages should not be too small, because the access time of a secondary storage device is much larger than the main memory. Pages should not be too large, else a large portion of the page may not be used, and it will occupy valuable space in the main memory

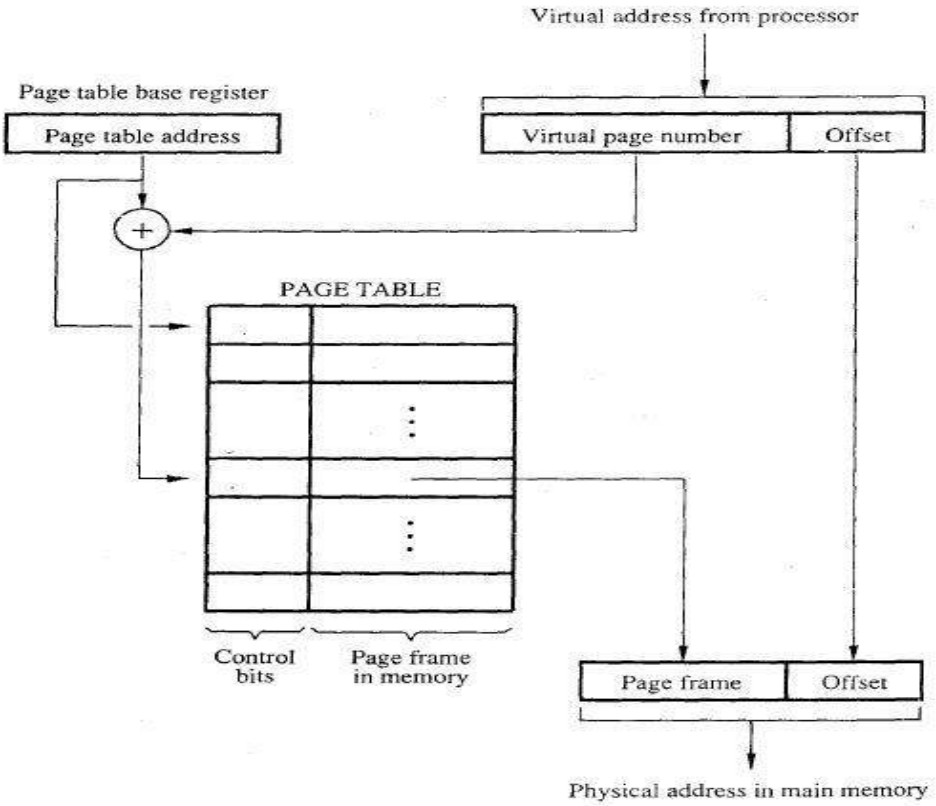



**Fig:Virtual memory address translation**

❖ **Memory segmentation**

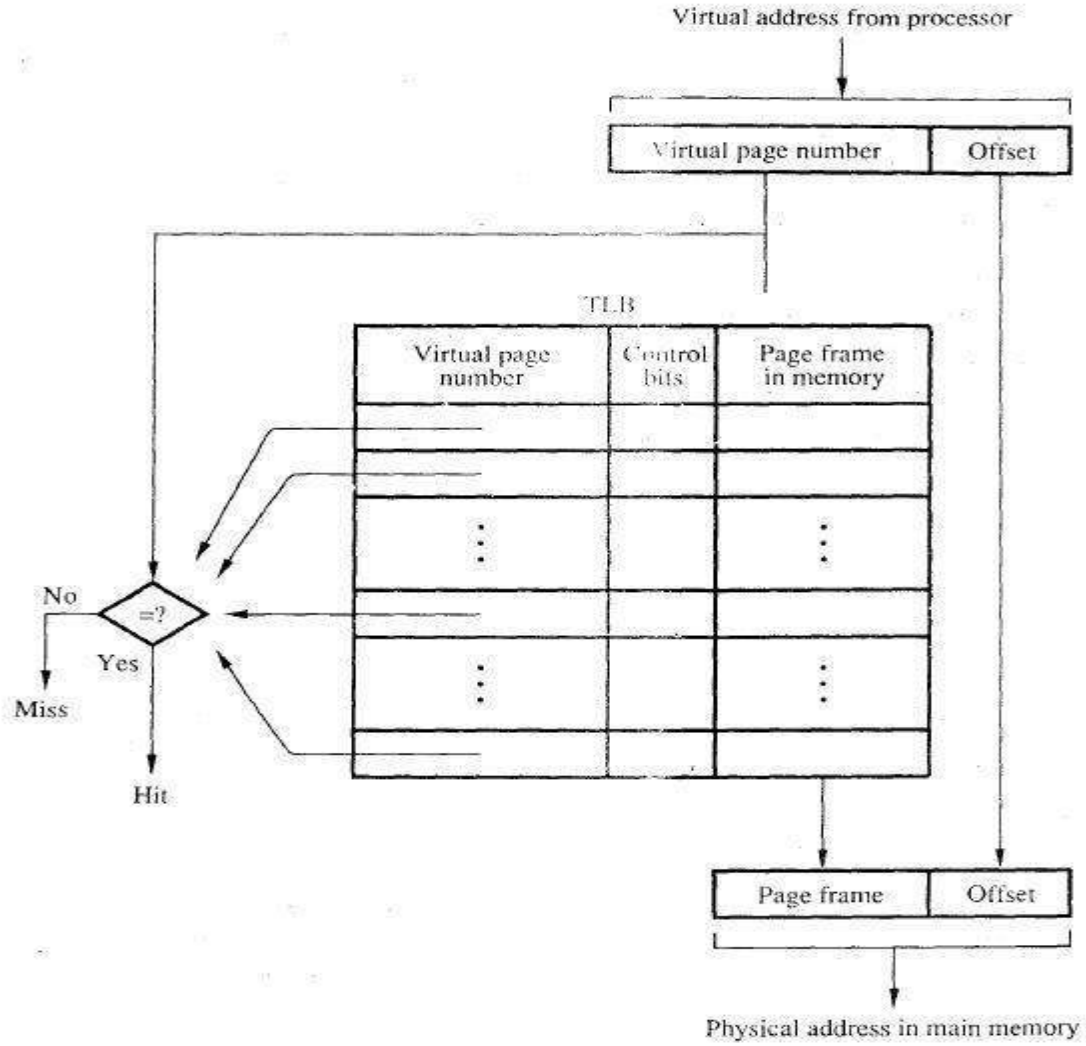
Memory segmentation is the division of a computer's primary memory into segments or sections. In a computer system using segmentation, a reference to a memory location includes a value that identifies a segment and an offset (memory location) within that segment.

Each virtual or logical address generated by a processor is interpreted as a virtual page number (high-order bits) plus an offset (low-order bits) that specifies the location of a word within that page.

❖ **Paging**

- Information about the main memory location of each page is kept in the **page table**.
- Information includes
  - Main memory address where the page is stored.
  - Current status of the page.
- Area of the main memory that can hold a page is called as page frame.
- Starting address of the page table is kept in a page table base register.
- Virtual page number generated by the processor is added to the contents of the page table base register. This provides the address of the corresponding entry in the page table. The contents of this location in the page table give the starting address of the page if the page is currently in the main memory.
- Page table entry for a page also includes some control bits which describe the status of the page while it is in the main memory.
- One bit indicates the validity of the page.Indicates whether the page is actually loaded into the main memory.
- One bit indicates whether the page has been modified during its residency in the main memory.
- Other control bits for various other types of restrictions that may be imposed. For example, a program may only have read permission for a page, but not write or modify permissions.
- The page table is used by the MMU for every read and write access to the memory. Ideal location for the page table is within the MMU. Page table is quite large.MMU is implemented as part of the processor chip. So it is impossible to include a complete page table on the chip. Page table is kept in the main memory. A copy of a small portion of the page table can be accommodated within the MMU. This portion consists of page table entries that correspond to the most recently accessed pages.A small cache called as **Translation Look aside Buffer (TLB)** is included in the MMU.
- TLB holds page table entries of the most recently accessed pages. Page table entry for a page includes: Address of the page frame where the page resides in the main memory, Some control bits are there. In addition to the above for each page, TLB must hold the virtual page number for each page.

**Associative-mapped TLB**




- High-order bits of the virtual address generated by the processor select the virtual page.
- These bits are compared to the virtual page numbers in the TLB.
- If there is a match, a hit occurs and the corresponding address of the page frame is read.
- If there is no match, a miss occurs and the page table within the main memory must be consulted.
- Set-associative mapped TLBs are found in commercial processors.
- A control bit is provided in the TLB to invalidate an entry. If an entry is invalidated, then the TLB gets the information for that entry from the page table.
- If a program generates an access to a page that is not in the main memory a page fault is said to occur. Whole page must be brought into the main memory from the disk, before the execution can proceed.
- Upon detecting a page fault by the MMU, following actions occur:

 MMU asks the operating system to intervene by raising an exception.
 Processing of the active task which caused the page fault is interrupted.
 Control is transferred to the operating system.
 Operating system copies the requested page from secondary storage to the main memory.

 Once the page is copied, control is returned to the task which was interrupted. Servicing of a page fault requires transferring the requested page from secondary storage to the main memory. This transfer may incur a long delay.

While the page is being transferred the operating system may:

 Suspend the execution of the task that caused the page fault.

 Begin execution of another task whose pages are in the main memory. Enables efficient use of the processor. To ensure that the interrupted task can continue correctly when it resumes execution ,there are two possibilities:

 Execution of the interrupted task must continue from the point where it was interrupted.

 The instruction must be restarted.

When a new page is to be brought into the main memory from secondary storage, the main memory may be full. Some page from the main memory must be replaced with this new page. How to choose which page to replace is similar to the replacement that occurs when the cache is full. The principle of locality of reference can also be applied here. A replacement strategy similar to LRU can be applied. Since the size of the main memory is relatively larger compared to cache, a relatively large amount of programs and data can be held in the main memory. This minimizes the frequency of transfers between secondary storage and main memory. A page may be modified during its residency in the main memory. Write-through protocol cannot be used, since it will incur a long delay each time a small amount of data is written to the disk.