MEGHE GROUP
VALUES REDEFINED

# DTEL (Department for Technology Enhanced Learning)

The Centre for Technology enabled Teaching & Learning , N Y S S, India

**Teaching Innovation - Entrepreneurial - Global**

## DEPARTMENT OF COMPUTER TECHNOLOGY

IV-SEMESTER

# COMPUTER ARCHITECTURE AND ORGANIZATION(CT2201)

## Unit II
## Basic Processing Unit

# UNIT II:- SYLLABUS

1 Addressing Modes

2 Some Fundamental Concepts

3 Stacks Subroutine.

4 Processing Unit: Some fundamental concepts.

5 Execution of a complete instruction: Single,

6 Two,

7 three bus organization, Sequencing of control Signals.

# CHAPTER-1 SPECIFIC OBJECTIVE / COURSE OUTCOME

**The student will be able to:**

1. Write various steps in a complete Instruction.
2. Write Control sequence for the given instruction

LECTURE 1:-

## Addressing Modes

- Implied
  - AC is implied in “ADD M[AR]” in “One-Address” instr.
  - TOS is implied in “ADD” in “Zero-Address” instr.
- Register
  - Indicate which register holds the operand

ADD R1,R2 R1=10, R2=20

- Absolute:

Indicate which Location holds the operand

MOVE LOC,R0 LOC=10

- Immediate

Move 20immediate, R1

Move #10,R1

  - The use of a constant in “MOV #5, R1”, i.e. R1 ← 5

## Addressing Modes

- Register Indirect
  - Indicate the register that holds the number of the register that holds the operand

    MOV (R2), R1

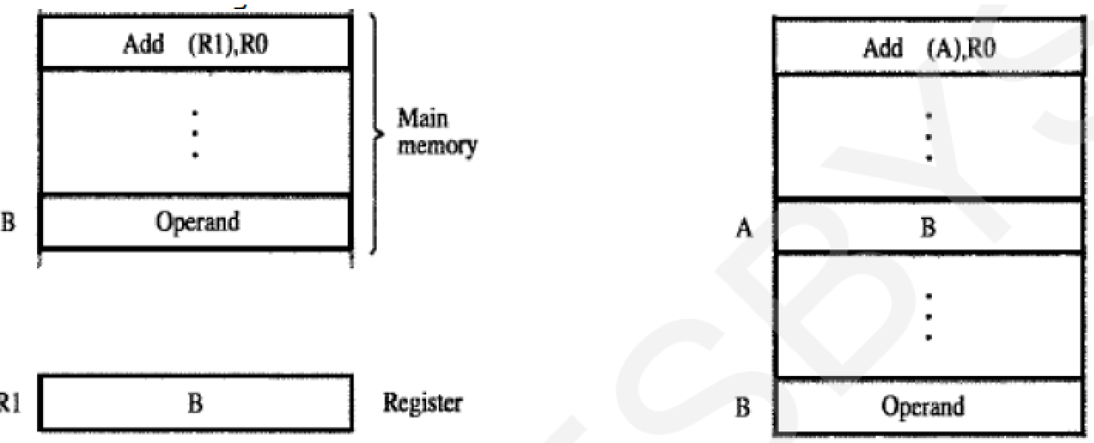
    MOV (A),R1
- Autoincrement / Autodecrement
  - Access & update in 1 instr.
- Direct Address
  - Use the given address to access a memory location

## Addressing Modes

- Register Indirect
  - Indicate the register that holds the number of the register that holds the operand

    MOV (R2), R1

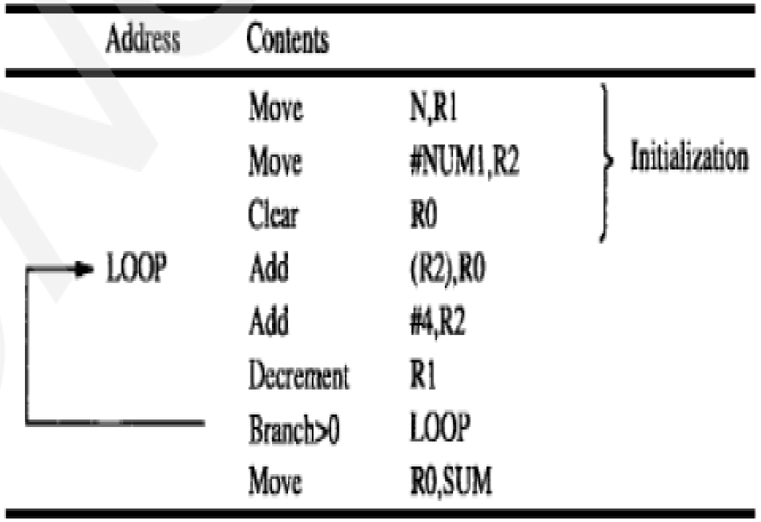
    MOV (A),R1



(a) Through a general-purpose register

(b) Through a memory location

Figure 2.11 Indirect addressing.



```
Address     Contents

            Move        N,R1       }
            Move        #NUM1,R2   } Initialization
            Clear       R0         }
  → LOOP    Add         (R2),R0
            Add         #4,R2
            Decrement   R1
            Branch>0    LOOP
            Move        R0,SUM
```

Figure 2.12 Use of indirect addressing in the program of Figure 2.10.

# Addressing Modes

## Indexing and Arrays

- Index mode – the effective address of the operand is generated by adding a constant value to the contents of a register.
- Index register

$X(R_i)$: $EA = X + [R_i]$

- The constant X may be given either as an explicit number or as a symbolic name representing a numerical value.
- If X is shorter than a word, sign-extension is needed.

## Addressing Modes

In general, the Index mode facilitates access to an operand whose location is defined relative to a reference point within the data structure in which the operand appears.

Several variations:

$(R_i, R_j)$: EA = $[R_i] + [R_j]$

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

Eg: Add 20(R1), R2

EA=>1000+20=1020

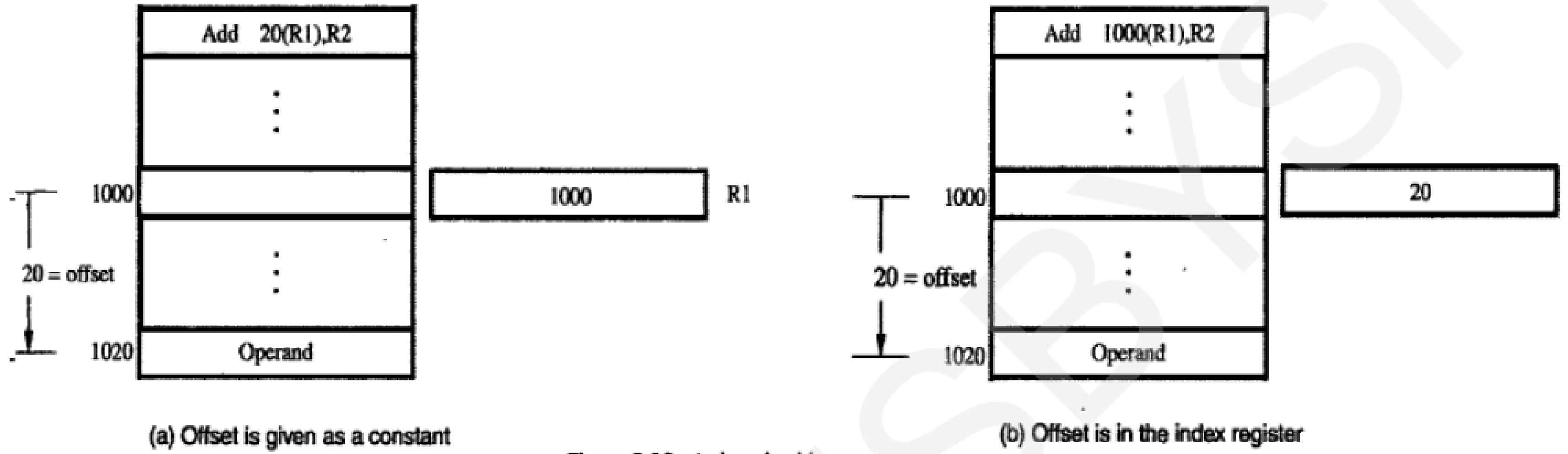


Figure 2.13 Indexed addressing.

## Addressing Modes

Example:

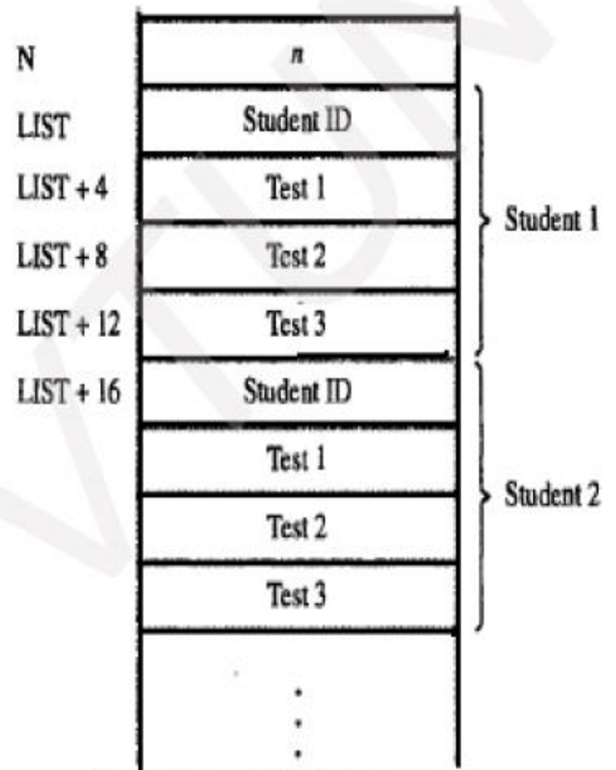
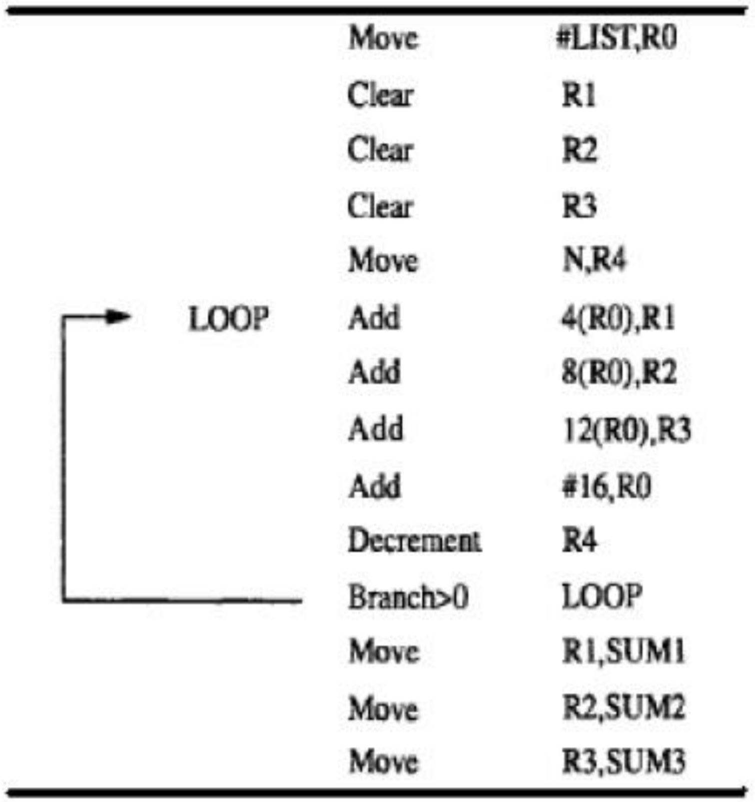


Figure 2.14 A list of students' marks.



```
                Move        #LIST,R0
                Clear       R1
                Clear       R2
                Clear       R3
                Move        N,R4
       LOOP     Add         4(R0),R1
                Add         8(R0),R2
                Add         12(R0),R3
                Add         #16,R0
                Decrement   R4
                Branch>0    LOOP
                Move        R1,SUM1
                Move        R2,SUM2
                Move        R3,SUM3
```

Figure 2.15 Indexed addressing used in accessing test scores in the list in Figure 2.14.

LECTURE 1:- Addressing Modes

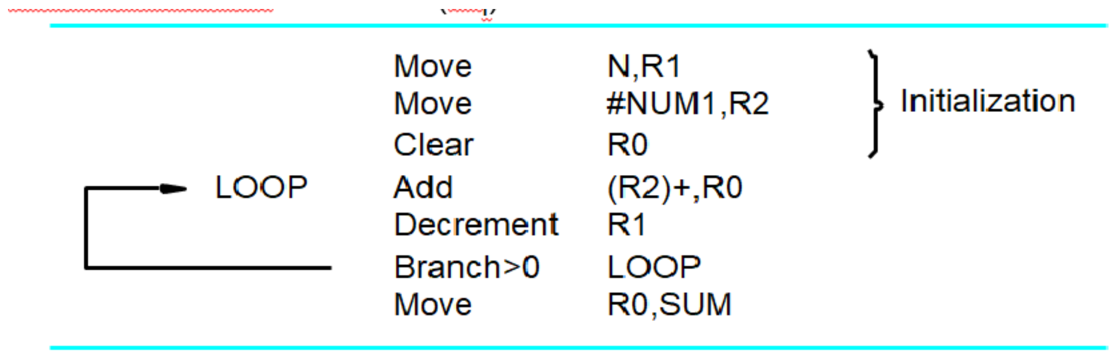
- Relative mode – the effective address is determined by the Index mode using the program counter in place of the general-purpose register.
- X(PC) – note that X is a signed number

Branch>0 LOOP

- This location is computed by specifying it as an offset from the current value of PC.
- Branch target may be either before or after the branch instruction, the offset is given as a singed num.

## Addressing Modes

- Autoincrement mode – the effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list.
- $(R_i)+$. The increment is 1 for byte-sized operands, 2 for 16-bit operands, and 4 for 32-bit operands.
- Autodecrement mode: $-(R_i)$ – decrement first



```
              Move         N,R1       }
              Move         #NUM1,R2   }  Initialization
              Clear        R0         }
       LOOP   Add          (R2)+,R0
              Decrement    R1
              Branch>0     LOOP
              Move         R0,SUM
```

re 2.16. The Autoincrement addressing mode used in the program of Figure 2.12.

# Assembly Language

- **Assembly Language:**

We generally use symbolic-names to write a program.

A complete set of symbolic-names and rules for their use constitute an **Assembly Language**.

The set of rules for using the mnemonics in the specification of complete instructions and programs is called the **Syntax** of the language.

Programs written in an assembly language can be automatically translated into a sequence of machine instructions by a program called an **Assembler.**

The user program in its original alphanumeric text formal is called a **Source Program**, and the assembled machine language program is called an **Object Program**.

For example:

*MOVE R0,SUM* ;The term MOVE represents OP code for operation performed by instruction.

*ADD #5,R3* ;Adds number 5 to contents of register R3 & puts the result back into registerR3.

ADDI 5,R3

# LECTURE 1:- Assembly Language

- **Assembler Directives:**

**Directives** are the assembler commands to the assembler concerning the program being assembled.

These commands are not translated into machine opcode in the object-program.

**EQU** informs the assembler about the value of an identifier (Figure: 2.18).

Ex: *SUM EQU 200* ;

**ORIGIN** tells the assembler about the starting-address of memory-area to place the data block.

Ex: ORIGIN 204 ;Instructs assembler to initiate data-block at memory-locations starting from 204.

**DATAWORD** directive tells the assembler to load a value into the location.

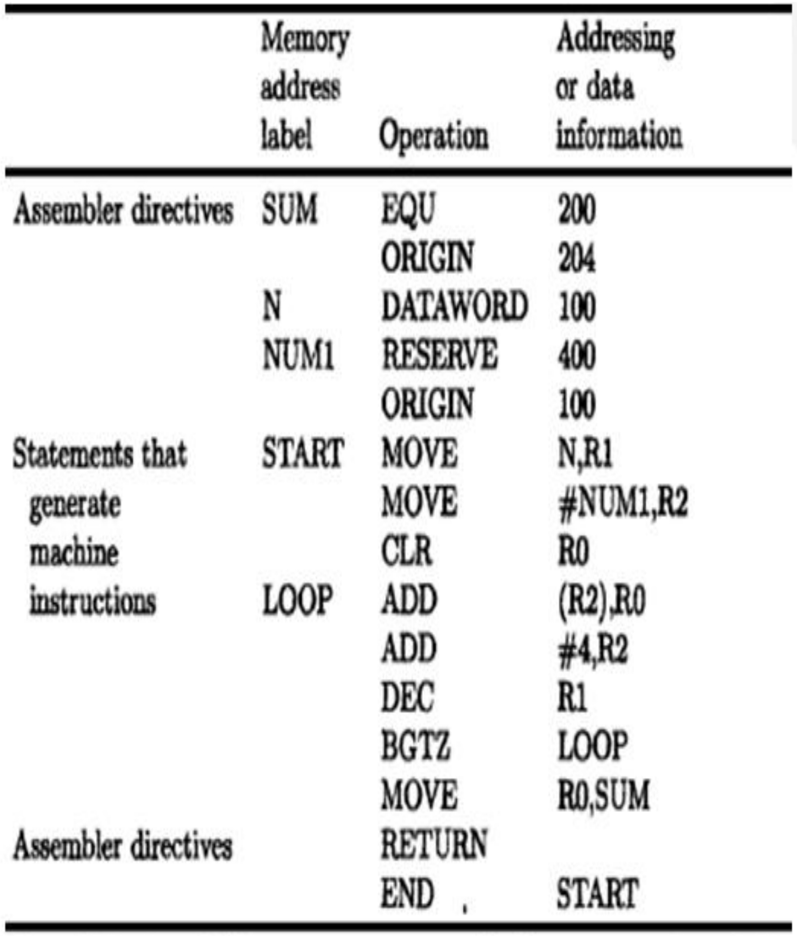
Ex: *N DATAWORD 100 ;*Informs the assembler to load data 100 into the memory-location N(204).

**RESERVE** directive is used to reserve a block of memory.

Ex: *NUM1 RESERVE 400 ;*declares a memory-block of 400 bytes is to be reserved for data.

**END** directive tells the assembler that this is the end of the source-program text.

**RETURN** directive identifies the point at which execution of the program should be terminated.



| | Memory address label | Operation | Addressing or data information |
|---|---|---|---|
| Assembler directives | SUM | EQU | 200 |
| | | ORIGIN | 204 |
| | N | DATAWORD | 100 |
| | NUM1 | RESERVE | 400 |
| | | ORIGIN | 100 |
| Statements that generate machine instructions | START | MOVE | N,R1 |
| | | MOVE | #NUM1,R2 |
| | | CLR | R0 |
| | LOOP | ADD | (R2),R0 |
| | | ADD | #4,R2 |
| | | DEC | R1 |
| | | BGTZ | LOOP |
| | | MOVE | R0,SUM |
| Assembler directives | | RETURN | |
| | | END | START |

**Figure 2.18** Assembly language representation for the program in Figure 2.17.

# Assembly Language

**ASSEMBLY AND EXECUTION OF PRGRAMS**

Programs written in an assembly language are automatically translated into a sequence of machine instructions by the **Assembler**.

**Assembler Program**

→ replaces all symbols denoting operations & addressing-modes with binary-codes used in machine instructions.
→ replaces all names and labels with their actual values.
→ assigns addresses to instructions & data blocks, starting at address given in ORIGIN directive
→ inserts constants that may be given in DATAWORD directives.
→ reserves memory-space as requested by RESERVE directives.

**Two Pass Assembler** has 2 passes:

**First Pass:** Work out all the addresses of labels.

As the assembler scans through a source-program, it keeps track of all names of numerical-values that correspond to them in a symbol-table.

**Second Pass:** Generate machine code, substituting values for the labels.

When a name appears a second time in the source-program, it is replaced with its value from the table.

The assembler stores the object-program on a magnetic-disk. The object-program must be loaded into the memory of the computer before it is executed. For this, a **Loader Program** is used.

**Debugger Program** is used to help the user find the programming errors.

Debugger program enables the user

→ to stop execution of the object-program at some points of interest &
→ to examine the contents of various processor-registers and memory-location.

# Stack & Subroutine

**SUBROUTINES**

A subtask consisting of a set of instructions which is executed many times is called a **Subroutine**.

A Call instruction causes a branch to the subroutine (Figure: 2.16).

At the end of the subroutine, a return instruction is executed

Program resumes execution at the instruction immediately following the subroutine call

The way in which a computer makes it possible to call and return from subroutines is referred to as its **Subroutine Linkage** method.

The simplest subroutine linkage method is to save the return-address in a specific location, which may be a register dedicated to this function. Such a register is called the **Link Register**.

When the subroutine completes its task, the Return instruction returns to the calling-program by branching indirectly through the link-register.

The **Call Instruction** is a special branch instruction that performs the following operations:

→ Store the contents of PC into link-register.

→ Branch to the target-address specified by the instruction.

The **Return Instruction** is a special branch instruction that performs the operation:

→ Branch to the address contained in the link-register.

## Stack & Subroutine

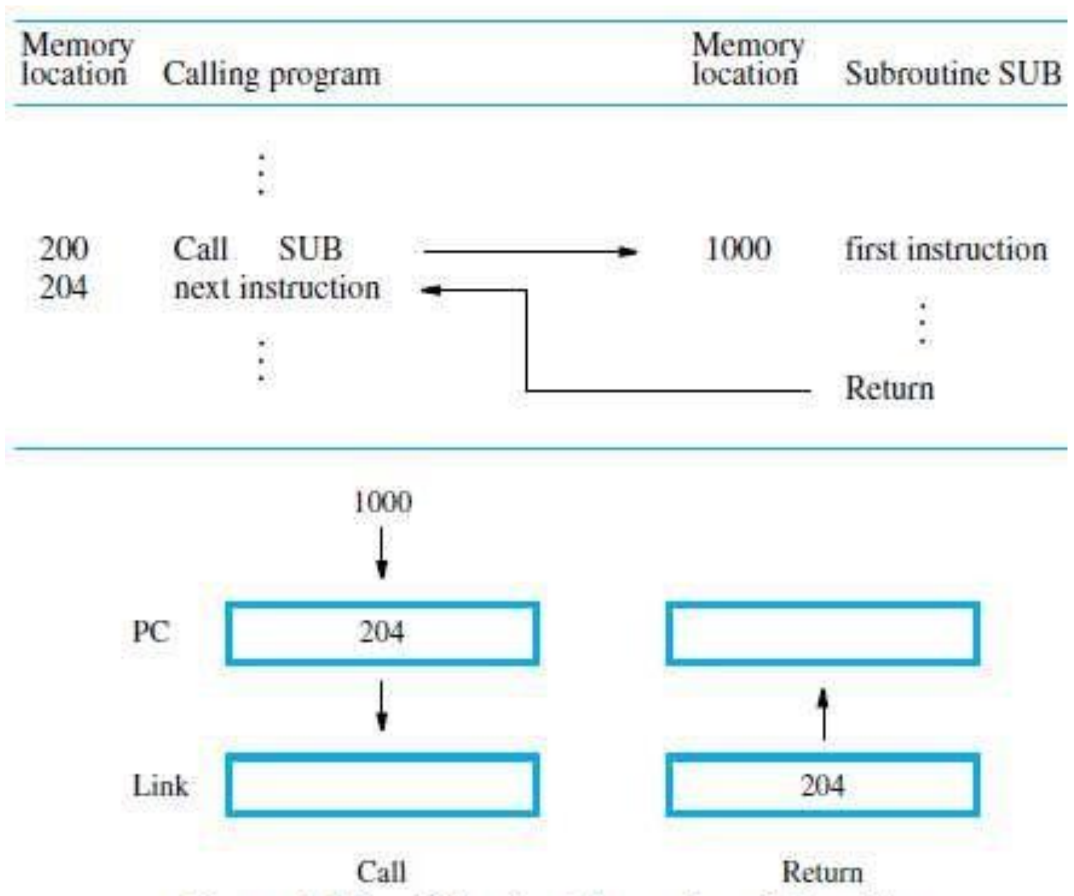


**Figure 2.16** Subroutine linkage using a link register.

LECTURE 1:-

## Assembly Language

**Program to illustrate stack frames for nested subroutines**

```
Memory
location          Instructions                  Comments

Main program
                  ⋮
2000              Move         PARAM2,–(SP)     Place parameters on stack.
2004              Move         PARAM1,–(SP)
2008              Call         SUB1
2012              Move         (SP),RESULT      Store result.
2016              Add          #8,SP            Restore stack level.
2020              next instruction
                  ⋮
First subroutine

2100   SUB1       Move         FP,–(SP)         Save frame pointer register.
2104              Move         SP,FP            Load the frame pointer.
2108              MoveMultiple R0–R3,–(SP)      Save registers.
2112              Move         8(FP),R0         Get first parameter.
                  Move         12(FP),R1        Get second parameter.
                  ⋮
                  Move         PARAM3,–(SP)     Place a parameter on stack.
2160              Call         SUB2
2164              Move         (SP)+,R2         Pop SUB2 result into R2.
                  ⋮
                  Move         R3,8(FP)         Place answer on stack.
                  MoveMultiple (SP)+,R0–R3      Restore registers.
                  Move         (SP)+,FP         Restore frame pointer register.
                  Return                        Return to Main program.
```

**Second subroutine**

```
3000   SUB2       Move         FP,–(SP)         Save frame pointer register.
                  Move         SP,FP            Load the frame pointer.
                  MoveMultiple R0–R1,–(SP)      Save registers R0 and R1.
                  Move         8(FP),R0         Get the parameter.
                  ⋮
                  Move         R1,8(FP)         Place SUB2 result on stack.
                  MoveMultiple (SP)+,R0–R1      Restore registers R0 and R1.
                  Move         (SP)+,FP         Restore frame pointer register.
                  Return                        Return to Subroutine 1.
```

**Figure 2.28** Nested subroutines.

LECTURE 1:- Assembly Language

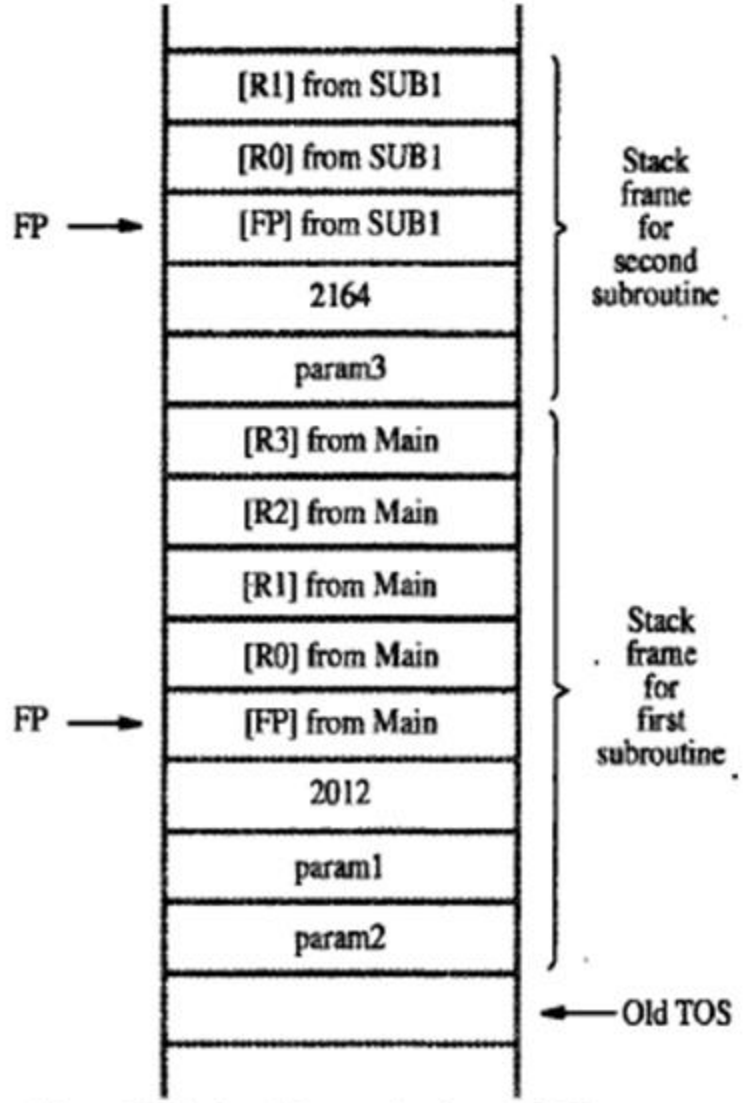


Figure 2.29 Stack frames for Figure 2.28.

## Some Fundamental Concepts

### Fundamental Concepts

- Processor fetches one instruction at a time and perform the operation specified.
- Instructions are fetched from successive memory locations until a branch or a jump instruction is encountered.
- Processor keeps track of the address of the memory location containing the next instruction to be fetched using Program Counter (PC).
- Instruction Register (IR)

## Some Fundamental Concepts

### Executing an Instruction

- Fetch the contents of the memory location pointed to by the PC. The contents of this location are loaded into the IR (fetch phase).

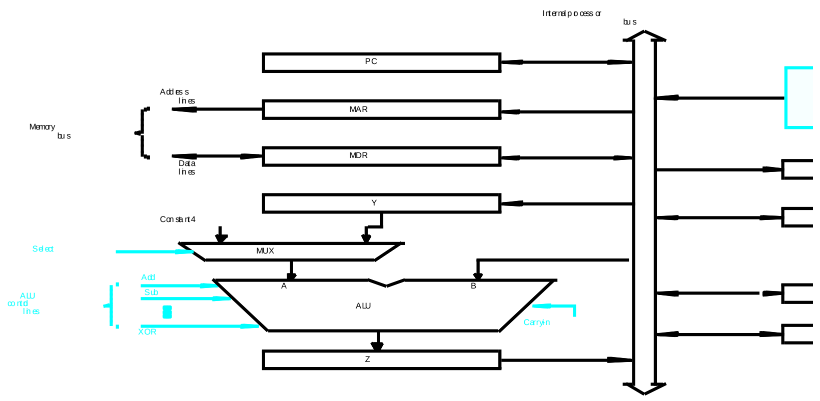
$$IR \leftarrow [[PC]]$$

- Assuming that the memory is byte addressable, increment the contents of the PC by 4 (fetch phase).

$$PC \leftarrow [PC] + 4$$

- Carry out the actions specified by the instruction in the IR (execution phase).

## Processor Organization



gure 7.1. Single-bus organization of the datapath inside a processor.

MDR HAS TWO INPUTS AND TWO OUTPUTS

Datapath

Textbook Page 413

## Some Fundamental Concepts

### Executing an Instruction

- Transfer a word of data from one processor register to another or to the ALU.
- Perform an arithmetic or a logic operation and store the result in a processor register.
- Fetch the contents of a given memory location and load them into a processor register.
- Store a word of data from a processor register into a given memory location.

# Some Fundamental Concepts

## Register Transfers

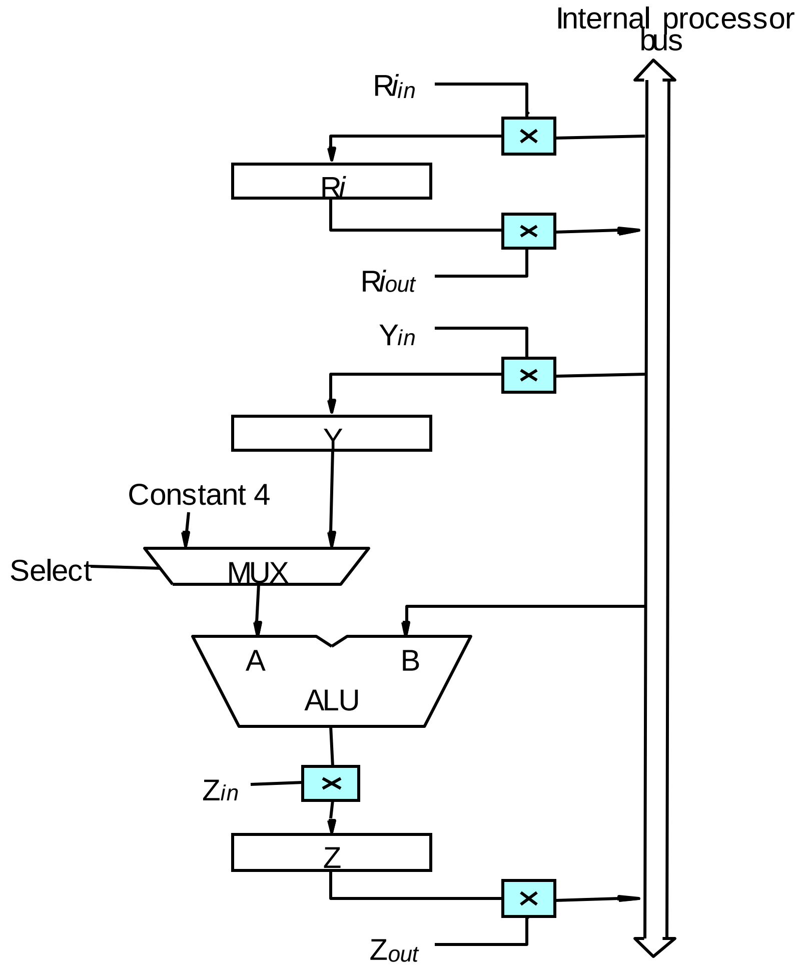


Figure 7.2. Input and output gating for the registers in Figure 7.1.

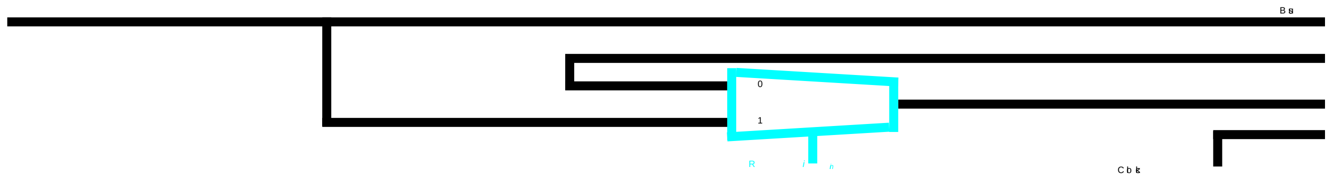
## Some Fundamental Concepts

Steps:To transfer the contents of register R1 to register R4

1. Enable the output of register R1 by setting $R1_{out}$ to 1. This places the contents of R1 on the processor bus.
2. Enable the input of register R4 by setting $R4_{in}$ to1. This loads data from the processor bus into register R4.

## Register Transfers (Cont.)

- All operations and data transfers are controlled by the processor clock.



F g u e 3 . h p t a d o p u g

## Performing an Arithmetic or Logic Operation

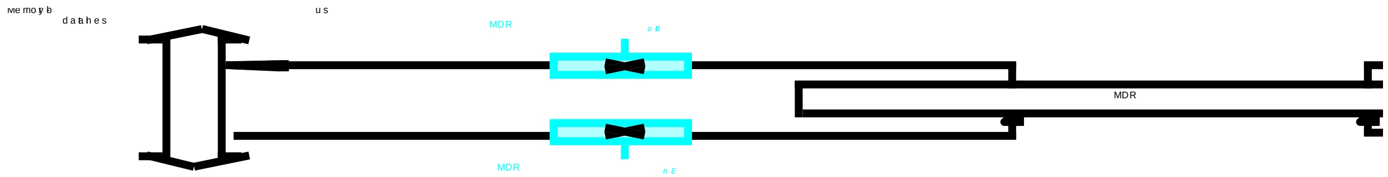
- The ALU is a combinational circuit that has no internal storage.
- ALU gets the two operands from MUX and bus. The result is temporarily stored in register Z.
- What is the sequence of operations to add the contents of register R1 to those of R2 and store the result in R3?

1. R1out, Yin
2. R2out, SelectY, Add, Zin
3. Zout, R3in

## Some Fundamental Concepts

Fetching a Word from Memory

- Address into MAR; issue Read operation; data into MDR.



Fgue 7.4. Conecbnandont bsgnasf or e

## Fetching a Word from Memory (Cont.)

- The response time of each memory access varies (cache miss, memory-mapped I/O,…).
- To accommodate this, the processor waits until it receives an indication that the requested operation has been completed (Memory-Function-Completed, MFC).
- Move (R1), R2

➢ MAR ← [R1] 1. R1out, MARin,Read

➢ Start a Read operation on the memory bus 2. MDRinE, WMFC

➢ Wait for the MFC response from the memory3. MDRout,R2in

➢ Load MDR from the memory bus

➢ R2 ← [MDR]

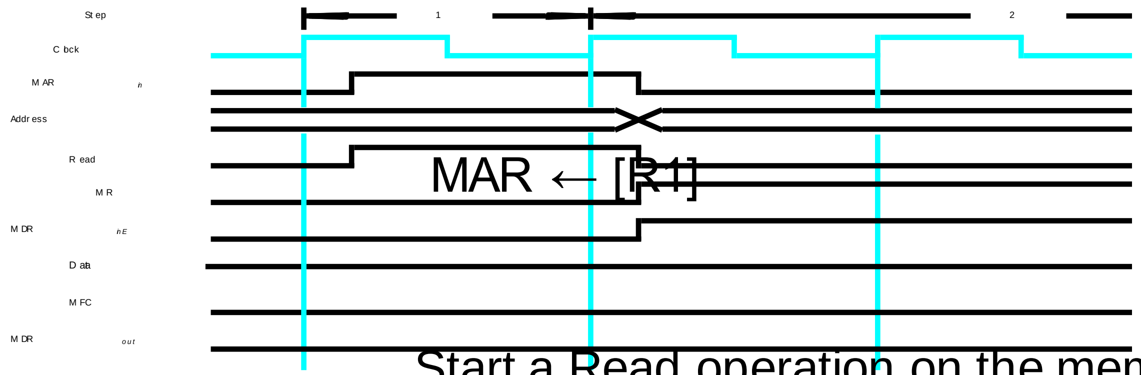
Move R2,(R1)

1. R1out,MARin,
2. R2out, MDRin,Write
3. MDRoutE,WMFC

# Some Fundamental Concepts

## Timing

Assume MAR
is always available
on the address lines
of the memory bus.



MAR ← [R1]

Start a Read operation on the memory bus

Figure 7.5. Timing of a memory Read operation.

Wait for the MFC response from the memory

Load MDR from the memory bus

R2 ← [MDR]

THANK YOU

## Execution of a Complete Instruction

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

# Execution of a Complete Instruction

## Architecture

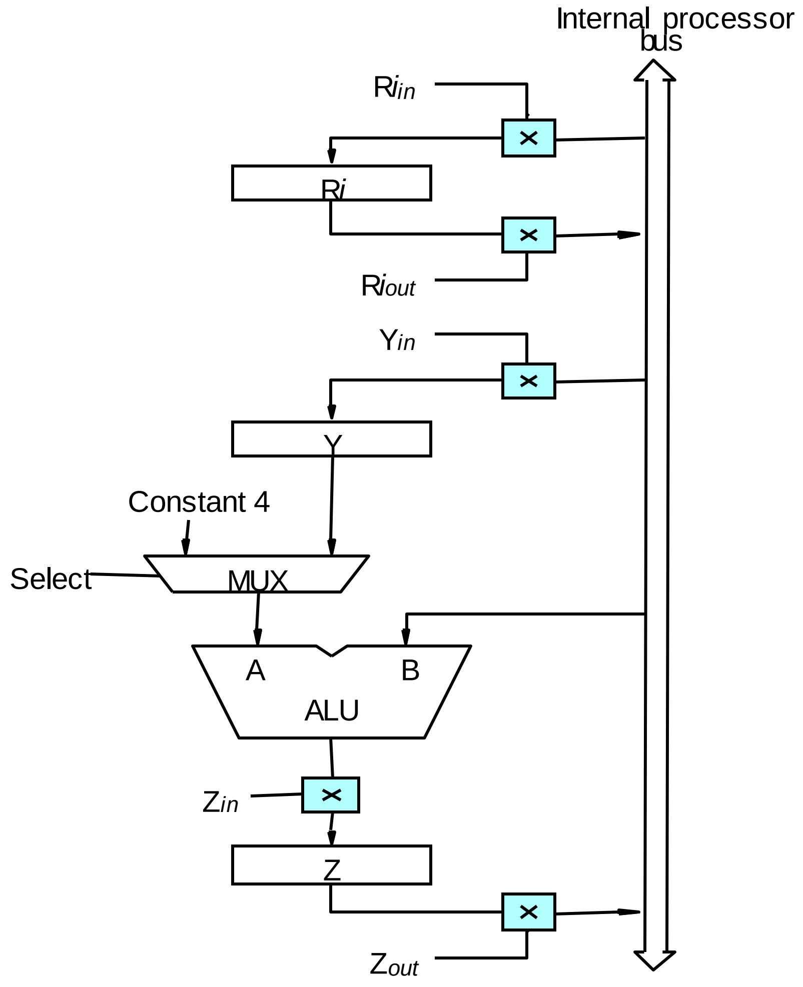


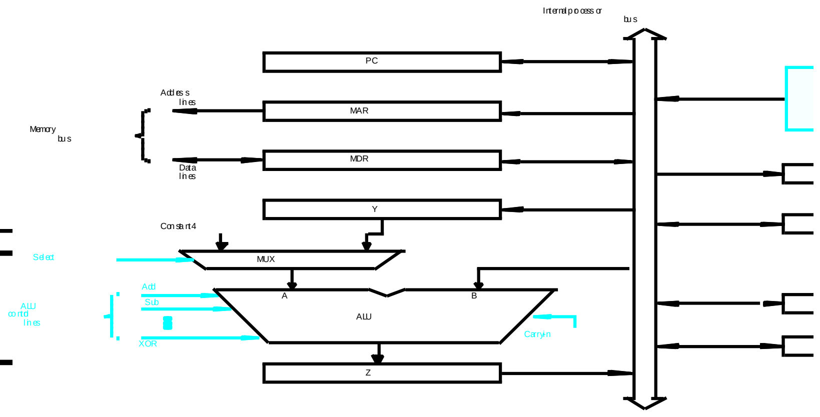
Figure 7.2. Input and output gating for the registers in Figure 7.1.

## Execution of a Complete Instruction

Add (R3), R1

| St p | Act o |
|---|---|
| 1 | $PC_{out}$, $MAR_{in}$, Read |
| 2 | $Z_{out}$, $PC_{in}$, $Y_{in}$, W MF |
| 3 | $MDR_{out}$, $R_{in}$ |
| 4 | $R3_{out}$, $MAR_{in}$, Read |
| 5 | $R1_{out}$, $Y_{in}$, W MF C |
| 6 | $MDR_{out}$, SelectY, Add, |
| 7 | $Z_{out}$, $R1_{in}$, End |

Figue 76. Con t ol seqene br exeut o



Figur e7.1. Single-bus organization of the datapath inside a processor.

Execution of a Complete Instruction

## Execution of Branch Instructions

- A branch instruction replaces the contents of PC with the branch target address, which is usually obtained by adding an offset X given in the branch instruction.
- The offset X is usually the difference between the branch target address and the address immediately following the branch instruction.
- Conditional branch

# LECTURE 2:-Execution of a Complete Instruction

## Execution of Branch Instructions (Cont.)

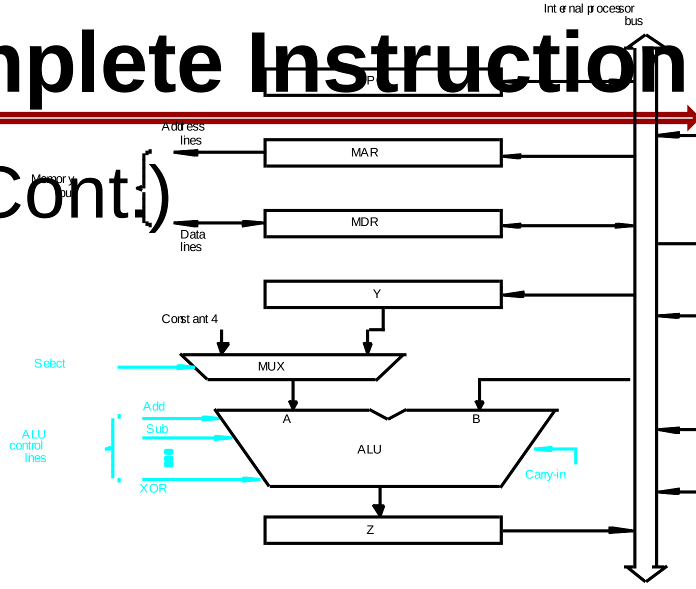


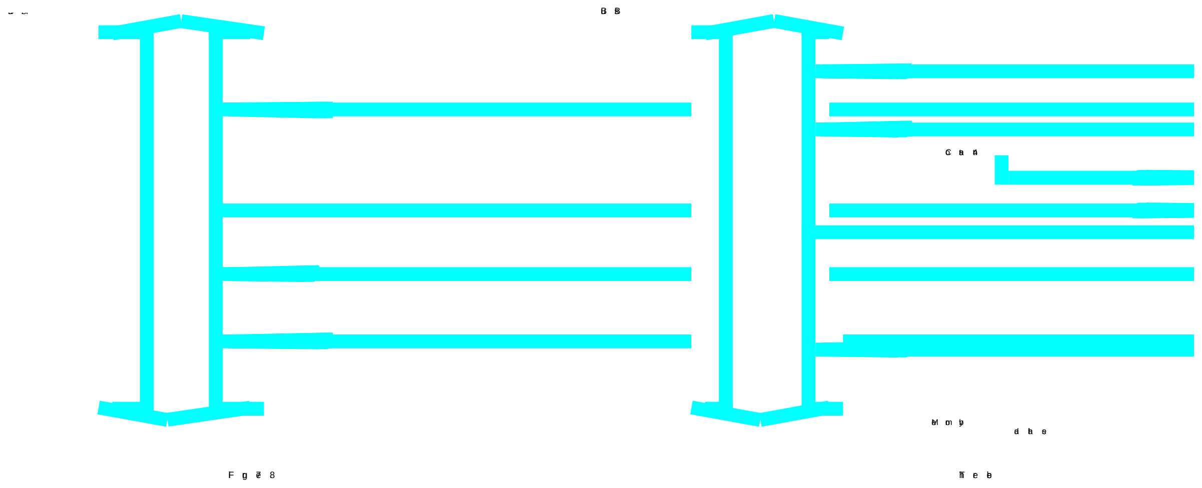
Figure 7.1. Single-bus organization of the datapath inside a processor.

| Step | Action |
|---|---|
| 1 | $PC_{out}$ , $MAR_{in}$ , Read, Select4,Add, $Z_{in}$ |
| 2 | $Z_{out}$ , $PC_{in}$ , $Y_{in}$ , WMFC |
| 3 | $MDR_{out}$ , $IR_{in}$ |
| 4 | Offset-field-of-$IR_{out}$ , Add, $Z_{in}$ |
| 5 | $Z_{out}$ , $PC_{in}$ , End |

4. Offset-field-of-IRout, Add, Zin, If N=0 then End

Figure 7.7. Control sequence for an unconditional branch instruction.

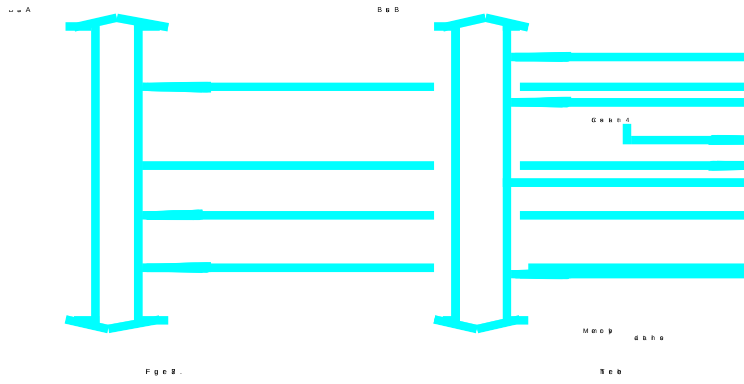
## Multiple-Bus Organization



# LECTURE 2:-Execution of a Complete Instruction

## Multiple-Bus Organization (Cont.)

- Add R4, R5, R6



| Step | Action |
|---|---|
| 1 | $PC_{out}$, R=B, $MAR_{in}$, Read, IncPC |
| 2 | WMFC |
| 3 | $MDR_{outB}$, R=B, $IR_{in}$ |
| 4 | $R4_{outA}$, $R5_{outB}$, SelectA, Add, $R6_{in}$, End |

Figure 7.9. Control sequence for the instruction.
Add R4,R5,R6 for the three-bus organization in Figure7.8.

# THANK YOU

## SimpleRISC

- SimpleRISC has 16 registers
  - Numbered r0 - r15
  - r14 is also referred as stack pointer(sp)
  - r15 is also referred as return address(ra)
- Special flags register-> Contains result of last comparisons
  - Flags.E=1(equality),flags.GT=1(greater than)

.

## LECTURE 2:- SimpleRISC

### mov instruction

| mov r1, r2 | R1<-r2 |
|---|---|
| mov r1, 3 | R1<-3 |

- Transfer the contents of one register to another
- Or, transfer the contents of an immediate to a register

.

# SimpleRISC

Arithmetic/Logical Instruction: SimpleRisc has 6 arithmetic instructions add, sub, mul, div, mod, cmp

| **add r1, r2, r3** | **r1<- r2 + r3** |
|---|---|
| add r1, r2, 10 | r1<- r2 + 10 |
| sub r1, r2, r3 | r1<- r2 - r3 |
| sub r1, r2, 10 | r1<- r2 - 10 |
| mul r1, r2, r3 | r1<- r2 * r3 |
| mul r1, r2, 10 | r1<- r2 * 10 |
| div r1, r2, r3 | r1<- r2 / r3(quotient) |
| div r1, r2, 10 | r1<- r2 / 10(quotient) |
| mod r1, r2, r3 | r1<- r2 mod r3 (remainder) |
| div r1, r2, 10 | r1<- r2 mod 10(remainder) |
| cmp r1, r2 | Sets flag : r1-r2=0 flags.E=1 |
| | r1 – r2>0 flags.GT=1 |

SimpleRISC

## Logical Instruction

| **and r1, r2, r3** | **r1<- r2 & r3** |
|---|---|
| and r1, r2, 10 | r1<- r2 & 10 |
| or r1, r2, r3 | r1<- r2 \| r3 |
| or r1, r2, 10 | r1<- r2 \| 10 |
| not r1, r2 | r1<-~r2 |
| not r1, 10 | r1<-~10 |

.

## SimpleRISC

Shift Instructions:

Logical shift left(lsl)(<<operator)

e.g. 0010 << 2 is equal to 1000

(<<n) is same as multiplying by $2^n$

Arithmetic shift right(asr)(>> operator)

e.g. 0010>>1 is equal to 0001

(>>n) same as dividing a signed number by $2^n$

.

## References (Book):

Computer Organization by Carl Hamacher, 5th Edition, McGraw Hill Education.

## References (Web):

Online Author PPTs (http://www.technolamp.co.in/2011/04/computer-organization-carl-hamacher.html)