# UNIT-2

# Chapter 2: Central Processing Unit

# CENTRAL PROCESSING UNIT

* Introduction
* General Register Organization
* Stack Organization
* Instruction Formats
* Addressing Modes
* Data Transfer and Manipulation
* Program Control
* Reduced Instruction Set Computer (RISC)

# MAJOR COMPONENTS OF CPU

Storage Components:

Registers

Flip-flops

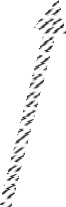
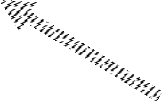
Execution (Processing) Components:

Arithmetic Logic Unit (ALU):

Arithmetic calculations, Logical computations, Shifts/Rotates

Transfer Components:

Bus



**ALU**

**Control Unit**

**Register File**

Control Components:

Control Unit

3

# GENERAL REGISTER ORGANIZATION

**Clock**



**Load (7 lines)**

**SELA {**

**SELD**

**OPR**

**Output**

**ALU**

**R7**

**R6**

**R5**

**R4**

**R3**

**R2**

**R1**

**Input**

**} SELB**

|  |  |
| --- | --- |
| **MUX** | |
|  | **A bus** |

|  |  |
| --- | --- |
| **MUX** | |
|  | **B bus** |



**3 x 8 decoder**

4

# OPERATION OF CONTROL UNIT

The control unit directs the information flow through ALU by:

* Selecting various *Components* in the system
* Selecting the *Function* of ALU

**Example: R1 <- R2 + R3**

1. MUX A selector (SELA): BUS A  R2
2. MUX B selector (SELB): BUS B  R3
3. ALU operation selector (OPR): ALU to ADD
4. Decoder destination selector (SELD): R1  Out Bus

**3 3 3 5**

Control Word

|  |  |  |  |
| --- | --- | --- | --- |
| **SELA** | **SELB** | **SELD** | **OPR** |

Encoding of register selection fields

|  |  |  |  |
| --- | --- | --- | --- |
| **Binary Code** | **SELA** | **SELB** | **SELD** |
| **000** | **Input** | **Input** | **None** |
| **001** | **R1** | **R1** | **R1** |
| **010** | **R2** | **R2** | **R2** |
| **011** | **R3** | **R3** | **R3** |
| **100** | **R4** | **R4** | **R4** |
| **101** | **R5** | **R5** | **R5** |
| **110** | **R6** | **R6** | **R6** |
| **111** | **R7** | **R7** | **R7** |

5

***Control***

# ALU CONTROL

Encoding of ALU operations

|  |  |  |
| --- | --- | --- |
| **OPR**  **Select** | **Operation** | **Symbol** |
| **00000** | **Transfer A** | **TSFA** |
| **00001** | **Increment A** | **INCA** |
| **00010** | **ADD A + B** | **ADD** |
| **00101** | **Subtract A - B** | **SUB** |
| **00110** | **Decrement A** | **DECA** |
| **01000** | **AND A and B** | **AND** |
| **01010** | **OR A and B** | **OR** |
| **01100** | **XOR A and B** | **XOR** |
| **01110** | **Complement A** | **COMA** |
| **10000** | **Shift right A** | **SHRA** |
| **11000** | **Shift left A** | **SHLA** |

Examples of ALU Microoperations

6

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Symbolic Designation**  **Microoperation SELA SELB SELD OPR Control Word** | | | | | |
| **R1  R2 - R3 R2 R3 R1** | **SUB** | **010** | **011** | **001** | **00101** |
| **R4  R4  R5 R4 R5 R4** | **OR** | **100** | **101** | **100** | **01010** |
| **R6  R6 + 1 R6 - R6** | **INCA** | **110** | **000** | **110** | **00001** |
| **R7  R1 R1 - R7** | **TSFA** | **001** | **000** | **111** | **00000** |
| **Output  R2 R2 - None** | **TSFA** | **010** | **000** | **000** | **00000** |
| **Output  Input Input - None** | **TSFA** | **000** | **000** | **000** | **00000** |
| **R4  shl R4 R4 - R4** | **SHLA** | **100** | **000** | **100** | **11000** |
| **R5**p****e 2**0**52: Comput**R**er**5** Orga**R**n**5**izatio**R**n**5** | **XOR** | **101** | **101** | **101** | **01100** |

# REGISTER STACK ORGANIZATION

Stack

* Very useful feature for nested subroutines, nested loops control
* Also efficient for arithmetic expression evaluation
* Storage which can be accessed in LIFO
* Pointer: SP
* Only PUSH and POP operations are applicable

|  |
| --- |
|  |
|  |
|  |
| **C** |
| **B** |
| **A** |
|  |

Register Stack

Push, Pop operations

**Flags**

**FULL**

**EMPTY**

**Stack pointer**



**SP**

**stack**

**Address**

**63**

**4**

**3**

**2**

**1**

**0**

**DR**

**/\* Initially, SP = 0, EMPTY = 1, FULL = 0 \*/**

**PUSH POP**

**SP  SP + 1 DR  M[SP]**

**M[SP]  DR SP  SP - 1**

**If (SP = 0) then (FULL  1) If (SP = 0) then (EMPTY  1)**

**EMPTY  0 FULL  0**

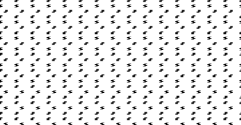
# MEMORY STACK ORGANIZATION

|  |  |
| --- | --- |
| **Program (instructions)** | **1000** |
| **Data (operands)** |  |
| **stack** | **3000** |
|  |  |
|  | **3997** |
|  | **3998** |
|  | **3999** |
|  | **4000** |
|  | **4001** |
|  |  |
| **DR** |  |

Memory with Program, Data,

**PC**

and Stack Segments



**AR**

**SP**

**-** A portion of memory is used as a stack with a processor register as a stack pointer

* PUSH: SP  SP - 1

M[SP]  DR

* POP: DR  M[SP]

SP  SP + 1

* Most computers do not provide hardware to check stack overflow (full stack) or underflow(empty stack)

# REVERSE POLISH NOTATION

Arithmetic Expressions: A + B

A + B Infix notation

+ A B Prefix or Polish notation

A B + Postfix or reverse Polish notation

- The reverse Polish notation is very suitable for stack

manipulation

Evaluation of Arithmetic Expressions

Any arithmetic expression can be expressed in parenthesis-free

Polish notation, including reverse Polish notation

**(3 \* 4) + (5 \* 6) ** **3 4 \* 5 6 \* +**

|  |
| --- |
|  |
|  |
|  |
| **12** |

|  |
| --- |
|  |
|  |
| **5** |
| **12** |

|  |
| --- |
|  |
| **6** |
| **5** |
| **12** |

|  |
| --- |
|  |
|  |
| **30** |
| **12** |

|  |
| --- |
|  |
|  |
|  |
| **42** |



|  |
| --- |
|  |
|  |
|  |
| **3** |

|  |
| --- |
|  |
|  |
| **4** |
| **3** |

**3 4 \* 5 6 \* +**

*Instruction Format*

# INSTRUCTION FORMAT

Instruction Fields

OP-code field - specifies the operation to be performed

Address field - designates memory address(s) or a processor register(s)

Mode field - specifies the way the operand or the

effective address is determined

The number of address fields in the instruction format

depends on the internal organization of CPU

- The three most common CPU organizations:

**Single accumulator organization:**

ADD X /\* AC  AC + M[X] \*/

**General register organization:**

|  |  |  |
| --- | --- | --- |
| ADD | R1, R2, R3 | /\* R1  R2 + R3 \*/ |
| ADD | R1, R2 | /\* R1  R1 + R2 \*/ |
| MOV | R1, R2 | /\* R1  R2 \*/ |
| ADD | R1, X | /\* R1  R1 + M[X] \*/ |

**Stack organization:**

PUSH X /\* TOS  M[X] \*/

ADD

# THREE, and TWO-ADDRESS INSTRUCTIONS

## Three-Address Instructions:

Program to evaluate X = (A + B) \* (C + D) :

|  |  |  |  |
| --- | --- | --- | --- |
| ADD | R1, A, B | /\* R1  M[A] + M[B] | \*/ |
| ADD | R2, C, D | /\* R2  M[C] + M[D] | \*/ |
| MUL | X, R1, R2 | /\* M[X]  R1 \* R2 | \*/ |

- Results in short programs

- Instruction becomes long (many bits)

## Two-Address Instructions:

Program to evaluate X = (A + B) \* (C + D) :

MOV R1, A /\* R1  M[A] \*/

ADD R1, B /\* R1  R1 + M[B] \*/

MOV R2, C /\* R2  M[C] \*/

ADD R2, D /\* R2  R2 + M[D] \*/ MUL R1, R2 /\* R1  R1 \* R2 \*/ MOV X, R1 /\* M[X]  R1 \*/

# ONE, and ZERO-ADDRESS INSTRUCTIONS

## One-Address Instructions:

* Use an implied AC register for all data manipulation
* Program to evaluate X = (A + B) \* (C + D) :

LOAD A /\* AC  M[A] \*/

|  |  |  |  |
| --- | --- | --- | --- |
| ADD | B | /\* AC  AC + M[B] | \*/ |
| STORE | T | /\* M[T]  AC | \*/ |
| LOAD | C | /\* AC  M[C] | \*/ |
| ADD | D | /\* AC  AC + M[D] | \*/ |
| MUL | T | /\* AC  AC \* M[T] | \*/ |
| STORE | X | /\* M[X]  AC | \*/ |

## Zero-Address Instructions:

* Can be found in a stack-organized computer
* Program to evaluate X = (A + B) \* (C + D) :

PUSH A /\* TOS  A \*/

PUSH B /\* TOS  B \*/

ADD /\* TOS  (A + B) \*/

PUSH C /\* TOS  C \*/

PUSH D /\* TOS  D \*/

ADD /\* TOS  (C + D) \*/

MUL /\* TOS  (C + D) \* (A + B) \*/

POP X /\* M[X]  TOS \*/

# ADDRESSING MODES

## Addressing Modes:

* Specifies a rule for interpreting or modifying the address field of the instruction (before the operand is actually referenced)
* Variety of addressing modes
  + to give programming flexibility to the user
  + to use the bits in the address field of the

instruction efficiently

# TYPES OF ADDRESSING MODES

## Implied Mode

Address of the operands are specified implicitly

in the definition of the instruction

* + No need to specify address in the instruction
  + EA = AC, or EA = Stack[SP**], EA: Effective Address.**

## Immediate Mode

Instead of specifying the address of the operand, operand itself is specified

* + - No need to specify address in the instruction
      * However, operand itself needs to be specified
      * Sometimes, require more bits than the address
      * Fast to acquire an operand

## Register Mode

Address specified in the instruction is the register address

* + - * Designated operand need to be in a register
      * Shorter address than the memory address
      * Saving address field in the instruction
      * Faster to acquire an operand than the memory addressing
      * EA = IR(R) (IR(R): Register field of IR)

# TYPES OF ADDRESSING MODES

## Register Indirect Mode

Instruction specifies a register which contains

the memory address of the operand

* Saving instruction bits since register address is shorter than the memory address
* Slower to acquire an operand than both the

register addressing or memory addressing

* EA = [IR(R)] ([x]: Content of x)

## Auto-increment or Auto-decrement features:

Same as the Register Indirect, but:

* When the address in the register is used to access memory, the value in the register is incremented or decremented by 1 (after or before the execution of the instruction)

# TYPES OF ADDRESSING MODES

**Direct Address Mode**

Instruction specifies the memory address which

can be used directly to the physical memory

* Faster than the other memory addressing modes
* Too many bits are needed to specify the address for a large physical memory space
  + EA = IR(address), (IR(address): address field of IR)

## Indirect Addressing Mode

The address field of an instruction specifies the address of a memory location that contains the address of the operand

* + When the abbreviated address is used, large physical memory can be addressed with a relatively small number of bits
  + Slow to acquire an operand because of an additional memory

access

* + EA = M[IR(address)]

# TYPES OF ADDRESSING MODES

## Relative Addressing Modes

The Address fields of an instruction specifies the part of the address

(abbreviated address) which can be used along with a designated register to calculate the address of the operand

PC Relative Addressing Mode(R = PC)

- EA = PC + IR(address)

* + - Address field of the instruction is short
    - Large physical memory can be accessed with a small number of address bits

## Indexed Addressing Mode

XR: Index Register:

* + - * EA = XR + IR(address)

## Base Register Addressing Mode

BAR: Base Address Register:

* + - * EA = BAR + IR(address)

# ADDRESSING MODES - EXAMPLES

**Address 200**

**PC = 200**

|  |  |
| --- | --- |
| **Load to AC** | **Mode** |
| **Address = 500** | |
| **Next instruction** | |
|  | |
| **450** | |
| **700** | |
|  | |
| **800** | |
|  | |
| **900** | |
|  | |
| **325** | |
|  | |
| **300** | |

**201**

**202**

**R1 = 400**

**399**

**XR = 100**

**400**

**Memory**

**AC**

**500**

**600**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Addressing Effective Mode Address** | |  |  | **Content**  **of AC** |
| **Direct address** | **500** | **/\* AC  (500)** | **\*/** | **800** |
| **Immediate operand** | **-** | **/\* AC  500** | **\*/** | **500** |
| **Indirect address** | **800** | **/\* AC  ((500))** | **\*/** | **300** |
| **Relative address** | **702** | **/\* AC  (PC+500)** | **\*/** | **325** |
| **Indexed address** | **600** | **/\* AC  (XR+500)** | **\*/** | **900** |
| **Register** | **-** | **/\* AC  R1** | **\*/** | **400** |
| **Register indirect** | **400** | **/\* AC  (R1)** | **\*/** | **700** |
| **Autoincrement** | **400** | **/\* AC  (R1)+** | **\*/** | **700** |
| **Autodecrement** | **399** | **/\* AC  -(R)** | **\*/** | **450** |

**702**

**800**

# DATA TRANSFER INSTRUCTIONS

**Typical Data Transfer Instructions**

|  |  |
| --- | --- |
| **Name** | **Mnemonic** |
| Load | LD |
| Store | ST |
| Move | MOV |
| Exchange | XCH |
| Input | IN |
| Output | OUT |
| Push | PUSH |
| Pop | POP |

**Data Transfer Instructions with Different Addressing Modes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Mode** | **Assembly Convention** | | **Register Transfer** |
| **Direct address** | **LD** | **ADR** | **AC  M[ADR]** |
| **Indirect address** | **LD** | **@ADR** | **AC  M[M[ADR]]** |
| **Relative address** | **LD** | **$ADR** | **AC  M[PC + ADR]** |
| **Immediate operand** | **LD** | **#NBR** | **AC  NBR** |
| **Index addressing** | **LD** | **ADR(X)** | **AC  M[ADR + XR]** |
| **Register** | **LD** | **R1** | **AC  R1** |
| **Register indirect** | **LD** | **(R1)** | **AC  M[R1]** |
| **Autoincrement** | **LD** | **(R1)+** | **AC  M[R1], R1  R1 + 1** |
| **Autodecrement** | **LD** | **-(R1)** | **R1  R1 - 1, AC  M[R1]** |

# DATA MANIPULATION INSTRUCTIONS

**Three Basic Types: Arithmetic instructions**

**Logical and bit manipulation instructions**

**Shift instructions**

**Arithmetic Instructions**

|  |  |
| --- | --- |
| **Name** | **Mnemonic** |
| **Increment** | **INC** |
| **Decrement** | **DEC** |
| **Add** | **ADD** |
| **Subtract** | **SUB** |
| **Multiply** | **MUL** |
| **Divide** | **DIV** |
| **Add with Carry** | **ADDC** |
| **Subtract with Borrow** | **SUBB** |
| **Negate(2’s Complement)** | **NEG** |

**Logical and Bit Manipulation Instructions Shift Instructions**

:

**Set carry SETC Complement carry COMC Enable interrupt EI**

**Disable interrupt DI** cpe 252

**CLR COM AND OR XOR CLRC**

**Clear Complement AND**

**OR**

**Exclusive-OR Clear carry**

**Mnemonic**

**Name**

|  |  |
| --- | --- |
| **Name** | **Mnemonic** |
| **Logical shift right** | **SHR** |
| **Logical shift left** | **SHL** |
| **Arithmetic shift right** | **SHRA** |
| **Arithmetic shift left** | **SHLA** |
| **Rotate right** | **ROR** |
| **Rotate left** | **ROL** |
| **Rotate right thru carry** | **RORC** |
| **Rotate left thru carry** | **ROLC** |

Computer Organization 20

# PROGRAM CONTROL INSTRUCTIONS

**+1**

 **In-Line Sequencing**

**(Next instruction is fetched from the**

**PC**

**next adjacent location in the memory)**

**Address from other source; Current Instruction, Stack, etc**

**Branch, Conditional Branch, Subroutine, etc**

**Program Control Instructions**

|  |  |
| --- | --- |
| **Name** | **Mnemonic** |
| **Branch** | **BR** |
| **Jump** | **JMP** |
| **Skip** | **SKP** |
| **Call** | **CALL** |
| **Return** | **RTN** |
| **Compare(by - )** | **CMP** |
| **Test (by AND)** | **TST** |

**Status Flag Circuit A B**

**8-bit ALU**

**F7 - F0**

**8 8**

**\* CMP and TST instructions do not retain their results of operations(- and AND, respectively). They only set or clear certain Flags.**



**c7**

**c8**

|  |  |  |  |
| --- | --- | --- | --- |
| **V** | **Z** | **S** | **C** |

**8**



**F7**

**Check for**

**zero output**

# CONDITIONAL BRANCH INSTRUCTIONS

|  |
| --- |
| **Mnemonic Branch condition Tested condition** |
| **BZ Branch if zero Z = 1**  **BNZ Branch if not zero Z = 0**  **BC Branch if carry C = 1**  **BNC Branch if no carry C = 0**  **BP Branch if plus S = 0**  **BM Branch if minus S = 1**  **BV Branch if overflow V = 1 BNV Branch if no overflow V = 0** |
| ***Unsigned* compare conditions (A - B) BHI Branch if higher A > B BHE Branch if higher or equal A  B BLO Branch if lower A < B BLOE Branch if lower or equal A  B BE Branch if equal A = B**  **BNE Branch if not equal A  B** |
| ***Signed* compare conditions (A - B)**  **BGT Branch if greater than A > B BGE Branch if greater or equal A  B BLT Branch if less than A < B BLE Branch if less or equal A  B BE Branch if equal A = B**  **BNE Branch if not equal A  B** |

SUBROUTINE CALL AND RETURN

## SUBROUTINE CALL

Call subroutine Jump to subroutine Branch to subroutine

Branch and save return address

**Two Most Important Operations are Implied;**

**\*** Branch to the beginning of the Subroutine

- Same as the Branch or Conditional Branch

* Save the Return Address to get the address of the location in the Calling Program upon exit from the Subroutine

**CALL**

**SP  SP - 1 M[SP]  PC PC  EA**

**RTN**

**PC  M[SP] SP  SP + 1**

* + Locations for storing Return Address:
    - Fixed Location in the subroutine(Memory)
    - Fixed Location in memory
    - In a processor Register
    - In a memory stack

- most efficient way

## Types of Interrupts:

PROGRAM INTERRUPT

**External interrupts**

External Interrupts initiated from the outside of CPU and Memory

* I/O Device -> Data transfer request or Data transfer complete
* Timing Device -> Timeout
* Power Failure

## Internal interrupts (traps)

Internal Interrupts are caused by the currently running program

* Register, Stack Overflow
* Divide by zero
* OP-code Violation
* Protection Violation

## Software Interrupts

Both External and Internal Interrupts are initiated by the computer Hardware. Software Interrupts are initiated by texecuting an instruction.

* Supervisor Call -> Switching from a user mode to the supervisor mode

-> Allows to execute a certain class of operations

which are not allowed in the user mode

# INTERRUPT PROCEDURE

## Interrupt Procedure and Subroutine Call

* + **The interrupt is usually initiated by an internal or an external signal rather than from the execution of an instruction (except for the software interrupt)**
  + **The address of the interrupt service program is determined by the hardware rather than from the address field of an instruction**
  + **An interrupt procedure usually stores all the information necessary to define the state of CPU rather than storing only the PC.**

**The state of the CPU is determined from;**

**Content of the PC**

**Content of all processor registers**

**Content of status bits**

**Many ways of saving the CPU state depending on the CPU architectures**

# RISC: REDUCED INSTRUCTION SET

**Historical Background**

**IBM System/360, 1964**

# COMPUTERS

* + - **The real beginning of modern computer architecture**
    - **Distinction between *Architecture* and *Implementation***
    - **Architecture: The abstract structure of a computer**

**seen by an assembly-language programmer**

**High-Level Language**

**Compiler -program Instruction**

**Set**

**Hardware**

**Architecture Implementation**

**Continuing growth in semiconductor memory and microprogramming**

**-> A much richer and complicated instruction sets**

**=> CISC(Complex Instruction Set Computer)**

* **Arguments advanced at that time**

**Richer instruction sets would simplify compilers**

**Richer instruction sets would alleviate the software crisis**

* + **move as much functions to the hardware as possible**
  + **close *Semantic Gap* between machine language and the high-level language**

**Richer instruction sets would improve the *architecture quality***

COMPLEX INSTRUCTION SET COMPUTERS: CISC

High Performance General Purpose Instructions

Characteristics of CISC:

1. A large number of instructions (from 100-250 usually)
2. Some instructions that performs a certain tasks are not used frequently.
3. Many addressing modes are used (5 to 20)
4. Variable length instruction format.
5. Instructions that manipulate operands in memory.

# PHYLOSOPHY OF RISC

**Reduce the semantic gap between**

**machine instruction and microinstruction**

* 1. **Cycle instruction**

**Most of the instructions complete their execution**

**in 1 CPU clock cycle - like a microoperation**

* + - **Functions of the instruction (contrast to CISC)**
      * **Very simple functions**
      * **Very simple instruction format**
      * **Similar to microinstructions**

**=> No need for microprogrammed control**

* + - **Register-Register Instructions**
      * **Avoid memory reference instructions except**

**Load and Store instructions**

* + - * **Most of the operands can be found in the**

**registers instead of main memory**

**=> Shorter instructions**

**=> Uniform instruction cycle**

**=> Requirement of large number of registers**

* + - **Employ instruction pipeline**

# CHARACTERISTICS OF RISC

**Common RISC Characteristics**

* **Operations are register-to-register, with only LOAD and STORE accessing memory**
* **The operations and addressing modes are reduced Instruction formats are simple**

# CHARACTERISTICS OF RISC

**RISC Characteristics**

* + **Relatively few instructions**
  + **Relatively few addressing modes**
  + **Memory access limited to load and store instructions**
  + **All operations done within the registers of the CPU**
  + **Fixed-length, easily decoded instruction format**
  + **Single-cycle instruction format**
  + **Hardwired rather than microprogrammed control**

**More RISC Characteristics**

-**A relatively large numbers of registers in the processor unit.**

-**Efficient instruction pipeline**

-**Compiler support: provides efficient translation of high-level language**

**programs into machine language programs.**

**Advantages of RISC**

* **VLSI Realization**
* **Computing Speed**
* **Design Costs and Reliability**
* **High Level Language Support**