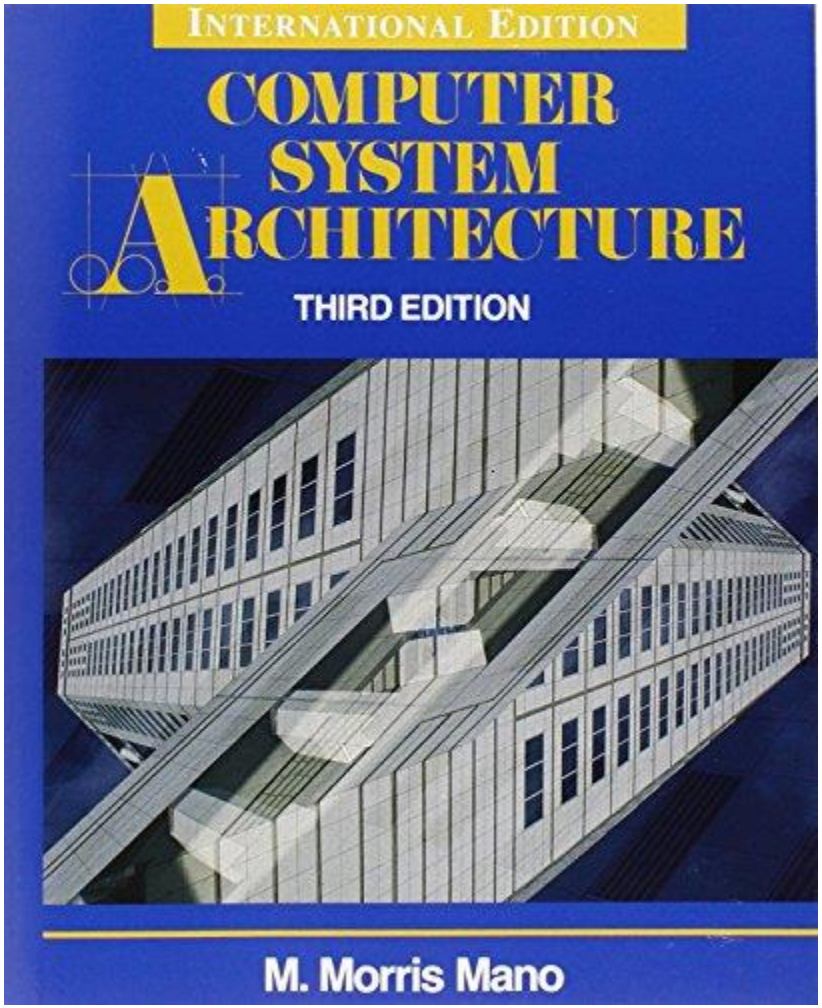
UNIT - II

CS304PC Computer Organization and Architecture

Shafakhatullah Khan Mohammed

Computer System Architecture

3rd edition (International Edition)



Unit # 2

Ch#7 Microprogrammed Control

Ch#8 Central Processing Unit

Chapter – 7

- **Title: Microprogrammed Control**
- **Topics:**
  - Control Memory
  - Address Sequencing
  - Micro Program Example
  - Design of Control Unit

Microprogramming

- The control unit is responsible for initiating the sequence of microoperations that comprise instructions.
  - When these control signals are generated by hardware, the control unit is hardwired.
  - When these control signals originate in data stored in a special unit and constitute a program on the small scale, the control unit is microprogrammed.

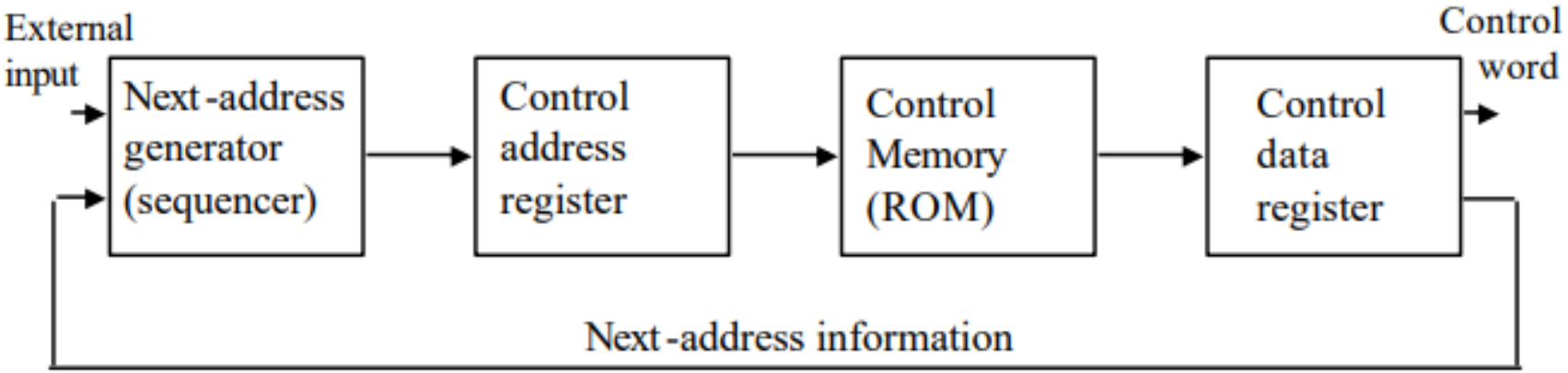
Controlled Memory

- The control function specifying a microoperation is a binary variable whose active state could be either 1 or 0.
  - In the variable's active state, the microoperation is executed.
  - The string of control variables which control the sequence of microoperations is called a control word.
- The microoperations specified in a control word is called a microinstruction.
  - Each microinstruction specifies one or more microoperations that is performed.
- The control unit coordinates stores microinstruction in its own memory (usually ROM) and performed the necessary steps to execute the sequences of microinstructions (called microprograms).

The Microprogrammed Control Unit

- In a microprogrammed processor, the control unit consists of:
  - Control address register: contains the address of the next microinstruction to be executed.
  - Control data register: contains the microinstruction to be executed.
  - The sequencer: determines the next address from within control memory
  - Control memory: where microinstructions are stored.

Microprogrammed Control Organization



Sequencer

- The sequencer generates a new address by:
  - incrementing the CAR
  - loading the CAR with an address from control memory.
  - transferring an external address

  or
  - loading an initial address to start the control operations.

Address Sequencing

- Microinstructions are usually stored in groups where each group specifies a routine, where each routine specifies how to carry out an instruction.
- Each routine must be able to branch to the next routine in the sequence.
- An initial address is loaded into the CAR when power is turned on; this is usually the address of the first microinstruction in the instruction fetch routine.
- Next, the control unit must determine the effective address of the instruction.

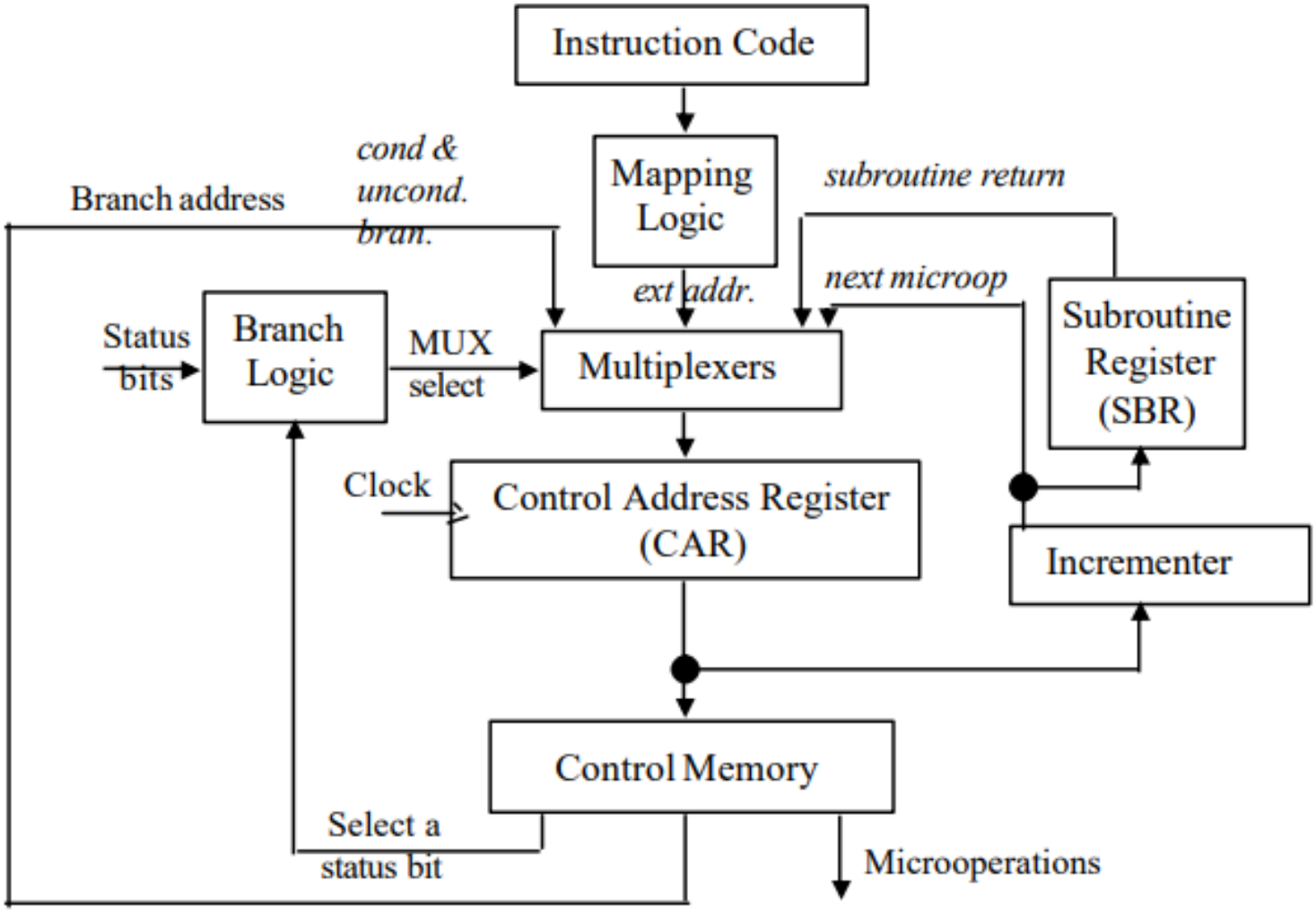
Mapping

- The next step is to generate the microoperations that executed the instruction.
  - This involves taking the instruction's opcode and transforming it into an address for the instruction's microprogram in control memory. This process is called mapping.
  - While microinstruction sequences are usually determined by incrementing the CAR, this is not always the case. If the processor's control unit can support subroutines in a microprogram, it will need an external register for storing return addresses.

Address Sequencing (Cont.…)

- When instruction execution is finished, control must be return to the fetch routine. This is done using an unconditional branch.
- Addressing sequencing capabilities of control memory include:
  - Incrementing the CAR
  - Unconditional and conditional branching (depending on status bit).
  - Mapping instruction bits into control memory addresses
  - Handling subroutine calls and returns.

Selection of Sequence for Control Memory



Conditional Branching

- Status bits
  - provide parameter information such as the carry-out from the adder, sign of a number, mode bits of an instruction, etc.
  - control the conditional branch decisions made by the branch logic together with the field in the microinstruction that specifies a branch address.

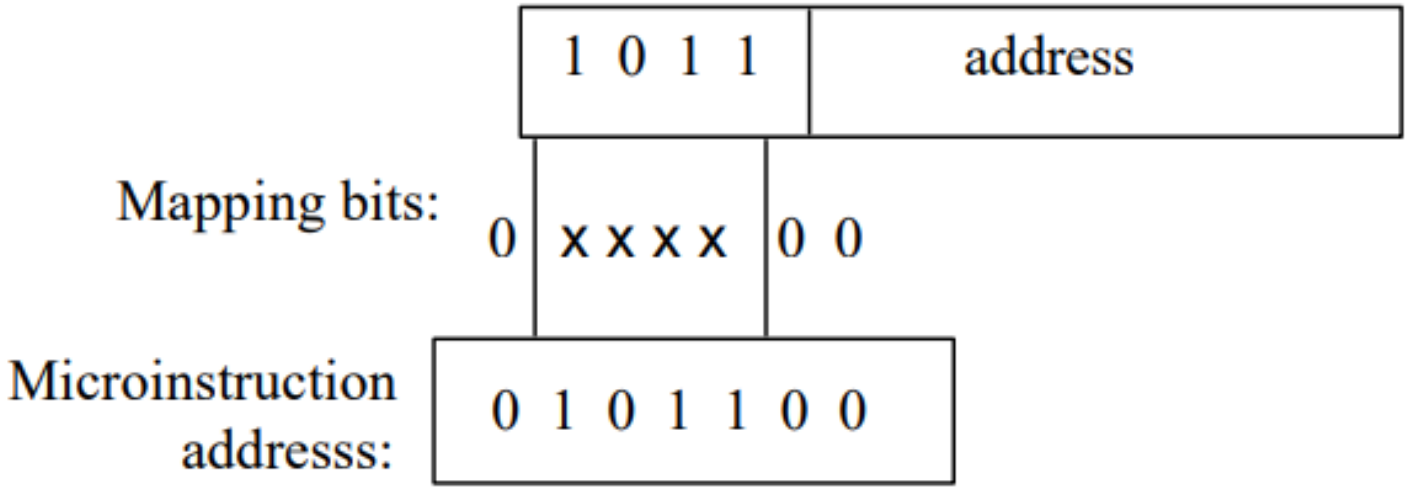
Branch Logic

- Branch Logic - may be implemented in one of several ways: –
  - The simplest way is to test the specified condition and branch if the condition is true; else increment the address register.
  - This is implemented using a multiplexer:
    - If the status bit is one of eight status bits, it is indicated by a 3-bit select number.
    - If the select status bit is 1, the output is 0; else it is 0.
    - A 1 generates the control signal for the branch; a 0 generates the signal to increment the CAR.
- Unconditional branching occurs by fixing the status bit as always being 1.

Mapping of Instruction

- Branching to the first word of a microprogram is a special type of branch. The opcode of the instruction indicates the branch.
- The mapping scheme shown in the figure allows for four microinstructions as well as overflow space from 1000000 to 1111111.

Mapping from Instruction code to Microoperation Address

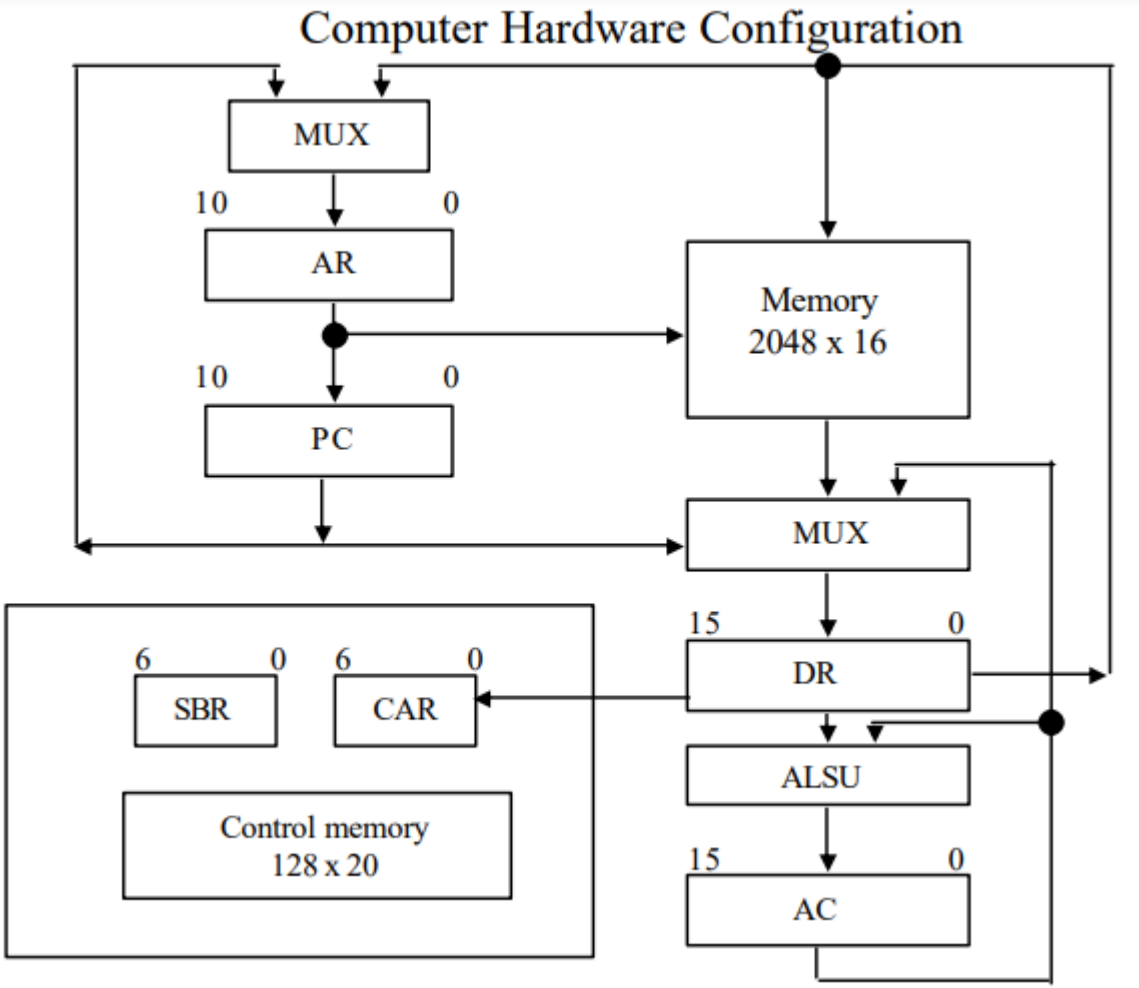


Subroutines

- Subroutine calls are a special type of branch where we return to one instruction below the calling instruction.
  - Provision must be made to save the return address since it cannot be written into ROM.

Subroutines

- Subroutine calls are a special type of branch where we return to one instruction below the calling instruction.
  - Provision must be made to save the return address since it cannot be written into ROM.

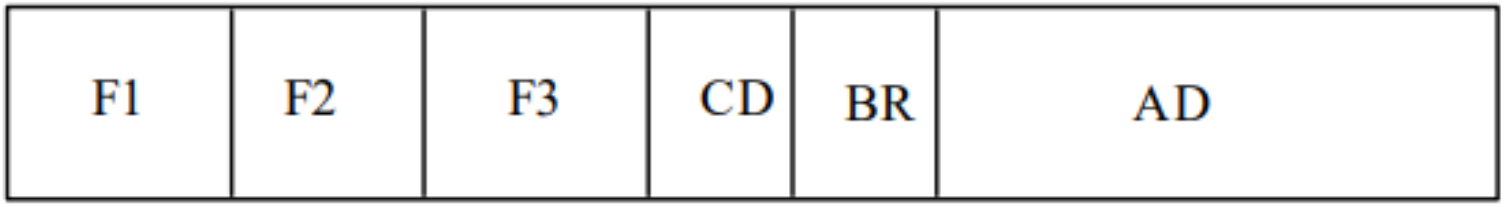


Computer Instructions

Computer Instructions

Symbol	Opcode	Description
ADD	0000	AC← AC + M[EA]
BRANCH	0001	IF (AC > 0) THEN PC ← EA
STORE	0010	M[EA] ← AC
EXCHANGE	0011	AC ← M[EA], M[EA]← AC

Microinstruction Code Format (20bits)



F1, F2, F3 : Microoperation Field

CD: Condition For Branching

BR: Branch Field

AD: Address Field

Symbols and Binary code for Microinstruction Fields

F1	Microoperation	Symbol
000	None	NOP
001	AC ← AC + DR	ADD
010	AC ← 0	CLRAC
011	AC ← AC + 1	INCAC
100	AC ← DR	DRTAC
101	AR ← DR(0-10)	DRTAR
110	AR ← PC	PCTAR
111	M[AR] ← DR	WRITE

Symbols and Binary code for Microinstruction Fields

F2	Microoperation	Symbol
000	None	NOP
001	AC ← AC- DR	SUB
010	AC ← AC ∨ DR	OR
011	AC ← AC ∧ DR	AND
100	DR ← M[AR]	READ
101	DR ← AC	ACTDR
110	DR ← DR + 1	INCDR
111	DR(0-10) ← PC	PCTDR

Symbols and Binary code for Microinstruction Fields

F3	Microoperation	Symbol
000	None	NOP
001	AC ← AC ⊕ DR	XOR
010	AC ← AC'	COM
011	AC ← shl AC	SHL
100	AC ← shr AC	SHR
101	PC ← PC + 1	INCPC
110	PC ← AR	ARTPC
111	Reserved	

Symbols and Binary code for Microinstruction Fields

CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional Branch
01	DR(15)	I	Indirect Address bit
10	AC(15)	S	Sign bit of AC
11	AC = 0	Z	Zero value in AC

Symbols and Binary code for Microinstruction Fields

BR	Symbol	Function
00	JMP	CAR ←AR if condition = 1 CAR←CAR + 1 if condition = 0
01	CAL	CAR ←AR, SBR ← CAR + 1 if cond. = 1 CAR←CAR + 1 if condition = 0
10	RET	CAR ← SBR (return from subroutine)
11	MAP	CAR(2-5) ← DR(11-14), CAR(0, 1, 6) ← 0

Symbolic Microinstructions

- It is possible to create a symbolic language for microcode that is machine-translatable to binary code.
- Each line defines a symbolic microinstruction with each column defining one of five fields:
  - Label – Either blank or a name followed by a colon (indicates a potential branch)
  - Microoperations – One, Two, Three Symbols, separated by commas (indicates that the microoperation is being performed)
  - CD - Either U, I, S or Z (indicates condition)
  - BR - One of four two-bit numbers
  - AD - A Symbolic Address, NEXT (address), RET, MAP (both of these last two converted to zeros by the assembler) (indicates the address of the next microinstruction)
- We will use the pseudo-instruction ORG to define the first instruction (or origin) of a microprogram, e.g., ORG 64 begins at 1000000.

Partial Symbolic Microprogram

Label	Microoperations	CD	BR	AD
	ORG 0			
ADD:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
	ORG 4			
BRANCH:	NOP	S	JMP	OVER
	NOP	U	JMP	FETCH
OVER:	NOP	I	CALL	INDRCT
	ARTPC	U	JMP	FETCH
	ORG 8			
STORE:	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
	WRITE	U	JMP	FETCH

Partial Symbolic Microprogram

```
            ORG 12
EXCHANGE:   NOP             I     CALL    INDRCT
            READ            U     JMP     NEXT
            ARTDR, DRTACU         JMP     NEXT
            WRITE           U     JMP     FETCH

            ORG 64
FETCH:      PCTAR           U     JMP     NEXT
            READ, INCPC     U     JMP     NEXT
            DRTAC           U     MAP
INDRCT:     READ            U     JMP     NEXT
            DRTAC           U     RET
```

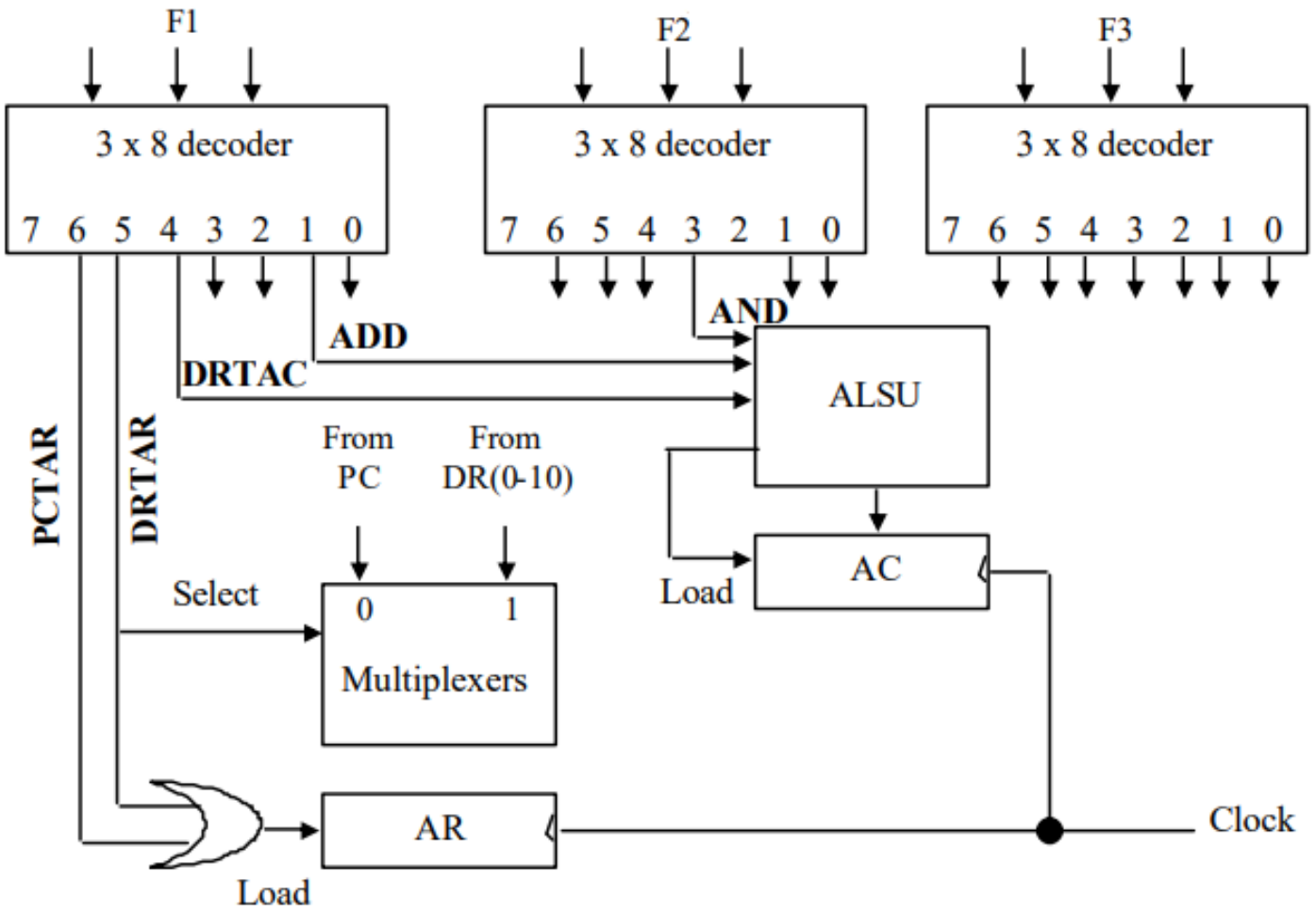
Partial Binary Microprogram

	Address		Binary Microinstruction					
Micro-Routine	Decimal	Binary	F1	F2	F3	CD	BR	AD
ADD	0	0000000	000	000	000	01	01	1000011
	1	0000001	000	100	000	00	00	0000010
	2	0000010	001	000	000	00	00	1000000
	3	0000011	000	000	000	00	00	1000000
BRANCH	4	0000100	000	000	000	10	00	0000110
	5	0000101	000	000	000	00	00	1000000
	6	0000110	000	000	000	01	01	1000011
	7	0000111	000	000	110	00	00	1000000
STORE	8	0001000	000	000	000	01	01	1000011
	9	0001001	000	101	000	00	00	0001010
	10	0001010	111	000	000	00	00	1000000
	11	0001011	000	000	000	00	00	1000000
EXCHANGE	12	0001100	000	000	000	01	01	1000011
	13	0001101	001	000	000	00	00	0001110
	14	0001110	100	101	000	00	00	0001111
	15	0001111	111	000	000	00	00	1000000
FETCH	64	1000000	000	000	000	00	00	1000001
	65	1000001	000	100	000	00	00	1000010
	66	1000010	000	000	000	00	11	0000000
INDRCT	67	1000011	000	100	000	00	00	1000100
	68	1000100	000	000	000	00	10	0000000

Control Unit Design

- Each field of k bits allows for 2k microoperations.
- The number of control bits can be reduced by grouping mutually exclusive microoperations together.
- Each field requires its own decoder to produce the necessary control signals.

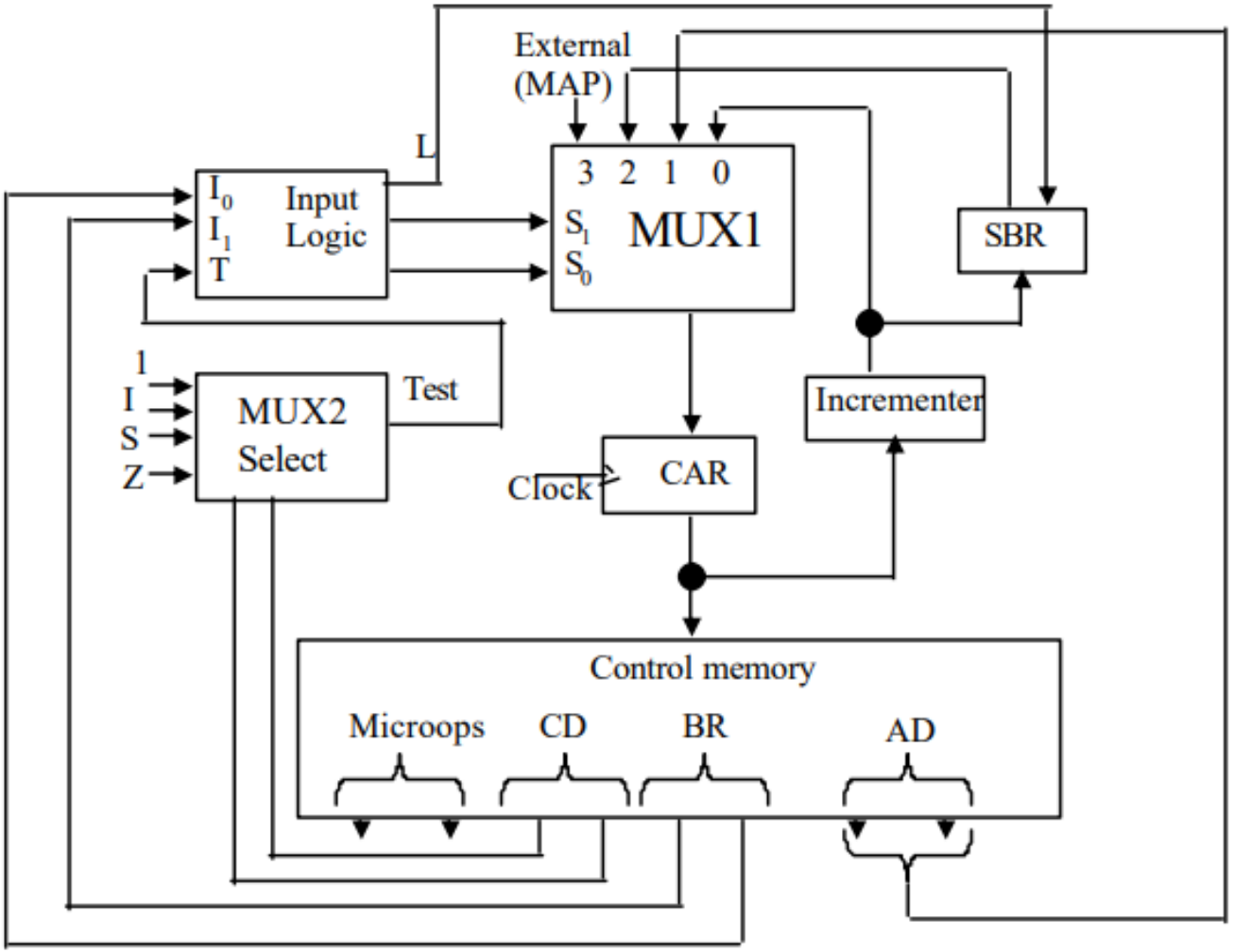
Decoding of Micro Fields



Microprogram Sequencer

- The microprogram sequencer selects the next address in control memory from which a microinstruction is to be fetched.
- Depending on the condition and on the branching type, it will be:
  - an external (mapped) address
  - the next microinstruction
  - a return from a subroutine
  - the address indicated in the microinstruction.

Microprogram Sequencer for a controlled memory



Input Logic Truth Table For A Microprogrammed Sequencer

BR Field		Input			MUX 1		Load SBR	
		I_1	I_0	T	S_1	S_0	L	
0	0	0	0	0	0	0	0	Next address
0	0	0	0	1	0	1	0	Specified addr.
0	1	0	1	0	0	0	0	
0	1	0	1	1	0	1	1	
1	0	1	0	x	1	0	0	Subroutine ret.
1	1	1	1	x	1	1	0	Ext. addr.

Chapter - 8

- **Title: Central Processing Unit**
- **Topics:**
  - General Register Organization
  - Instruction Formats
  - Addressing modes
  - Data Transfer and Manipulation
  - Program Control

Central Processing Unit - Introduction

- 3 major parts of CPU :
  - 1) Register Set
  - 2) ALU
  - 3) Control Design

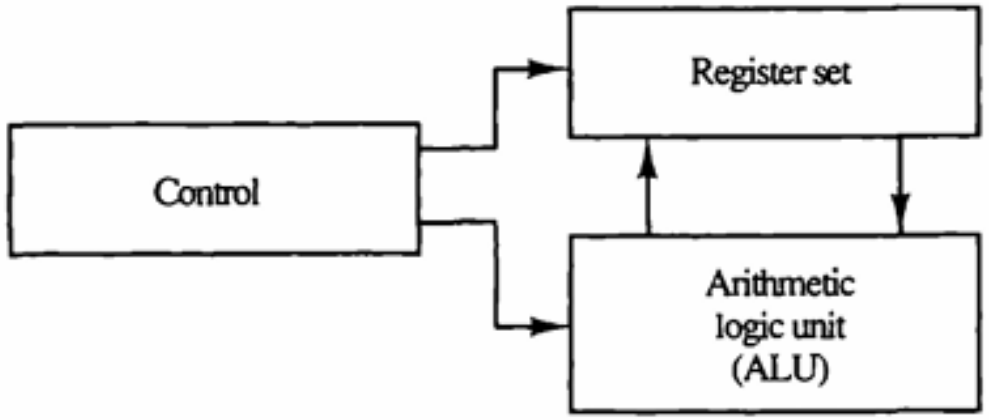
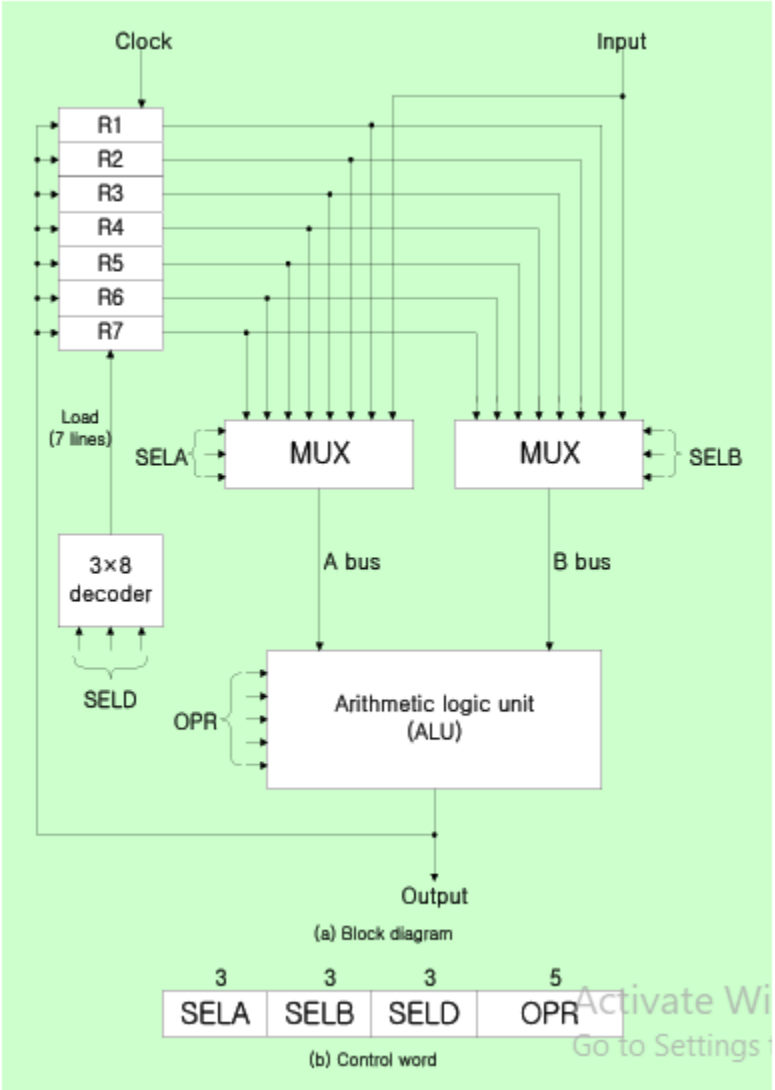


Figure 8-1 Major components of CPU.

- Examples of simple CPU
  - Hardwired Control
  - Microprogrammed Control

General Register Organization

- Necessity of Register:
  - Memory locations are needed for storing pointers, counters, return address, temporary results, and partial products during multiplication
  - Memory access is the most time-consuming operation in a computer.
  - More convenient and efficient way is to store intermediate values in processor registers.



General Register Organization

- Bus organization for 7 CPU registers:
  - 2 MUX: select one of 7 register or external data input by SELA and SELB
  - BUS A and BUS B: form the inputs to a common ALU
  - ALU: OPR determine the arithmetic or logic microoperation » The result of the microoperation is available for external data output and also goes into the inputs of all the registers
  - 3 X 8 Decoder: select the register (by SELD) that receives the information from ALU

General Register Organization

- Binary selector input :
  - 1) MUX A selector (SELA): to place the content of R2 into BUS A
  - 2) MUX B selector (SELB): to place the content of R3 into BUS B
  - 3) ALU operation selector (OPR): to provide the arithmetic addition R2 + R3.
  - 4) Decoder selector (SELD): to transfer the content of the output bus into R1.

General Register Organization

- Control Word 14-bit control word (4 fields): ***Fig. 8-2(b)***
  - SELA(3 bits): Select a source register for the A input of the ALU
  - SELB(3 bits): Select a source register for the B input of the ALU
  - SELD(3 bits): Select a destination register using the 3 X 8 decoder
  - OPR(5 bits): Select one of the operations in the ALU

TABLE 8-1 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

General Register Organization

- Encoding of Register Selection Fields: Tab. 8-1
  - SELA or SELB = 000 (Input): MUX selects the external input data.
  - SELD = 000 (None): no destination register is selected but the contents of the output bus are available in the external output.

TABLE 8-2 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

General Register Organization

TABLE 8-3 Examples of Microoperations for the CPU

Microoperation	Symbolic Designation				Control Word
	SELA	SELB	SELD	OPR	
$R1 \leftarrow R2 - R3$	R2	R3	R1	SUB	010 011 001 00101
$R4 \leftarrow R4 \vee R5$	R4	R5	R4	OR	100 101 100 01010
$R6 \leftarrow R6 + 1$	R6	—	R6	INCA	110 000 110 00001
$R7 \leftarrow R1$	R1	—	R7	TSFA	001 000 111 00000
Output $\leftarrow R2$	R2	—	None	TSFA	010 000 000 00000
Output $\leftarrow$ Input	Input	—	None	TSFA	000 000 000 00000
$R4 \leftarrow$ shl $R4$	R4	—	R4	SHLA	100 000 100 11000
$R5 \leftarrow 0$	R5	R5	R5	XOR	101 101 101 01100

- TSFA (Transfer A) : $R7 \leftarrow R1$,
  External Output $\leftarrow R2$,
  External Output $\leftarrow$ *External Input*
- XOR : $R5 \leftarrow 0$ (XOR $R5 \oplus R5$)

Instruction Formats

- Fields in Instruction Formats
  - 1) Operation Code Field: specify the operation to be performed
  - 2) Address Field: designate a memory address or a processor register
  - 3) Mode Field: specify the operand or the effective address (Addressing Mode)
- 3 types of CPU organizations
  - 1) Single AC Org. : **ADD X** ($AC \leftarrow AC + M[X]$)
  - 2) General Register Org. : **ADD R1, R2, R3** ($R1 \leftarrow R2 + R3$)
  - 3) Stack Org. : **PUSH X** ($TOS \leftarrow M[X]$)

Instruction Formats

- The influence of the number of addresses on computer instruction
- Example X = (A + B)*(C + D)
  - 4 arithmetic operations: ADD, SUB, MUL, DIV
  - 1 transfer operation to and from memory and general register: MOV
  - 2 transfer operation to and from memory and AC register: STORE, LOAD
  - Operand memory addresses: A, B, C, D
  - Result memory address: X
- 1) Three-Address Instruction

```
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
```

  - Each address fields specify either a processor register or a memory operand
  - Short program
  - Require too many bits to specify 3 address

Instruction Formats

- The influence of the number of addresses on computer instruction
- Example X = (A + B)*(C + D)
  - 4 arithmetic operations: ADD, SUB, MUL, DIV
  - 1 transfer operation to and from memory and general register: MOV
  - 2 transfer operation to and from memory and AC register: STORE, LOAD
  - Operand memory addresses: A, B, C, D
  - Result memory address: X
- 1) Three-Address Instruction

```
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
```

  - Each address fields specify either a processor register or a memory operand
  - Short program
  - Require too many bits to specify 3 address

Instruction Formats

- 2) Two-Address Instruction

```
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
```

  - The most common in commercial computers
  - Each address fields specify either a processor register or a memory operand

- 3) One-Address Instruction
  - All operations are done between the AC register and memory operand

```
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
```

Instruction Formats

- Zero-Address Instruction

```
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
```

  - Stack-organized computer does not use an address field for the instructions ADD, and MUL
  - PUSH, and POP instructions need an address field to specify the operand
  - Zero-Address: absence of address (ADD, MUL)

- RISC Instruction
  - Only use LOAD and STORE instruction when communicating between memory and CPU
  - All other instructions are executed within the registers of the CPU without referring to memory.

Instruction Formats

- RISC architecture will be explained in ***Unit-V of CS304PC***
- Program to evaluate X = (A + B) * (C + D)

```
LOAD    R1, A          R1 ← M[A]
LOAD    R2, B          R2 ← M[B]
LOAD    R3, C          R3 ← M[C]
LOAD    R4, D          R4 ← M[D]
ADD     R1, R1, R2     R1 ← R1 + R2
ADD     R3, R3, R2     R3 ← R3 + R4
MUL     R1, R1, R3     R1 ← R1* R3
STORE   X, R1          M[X] ← R1
```

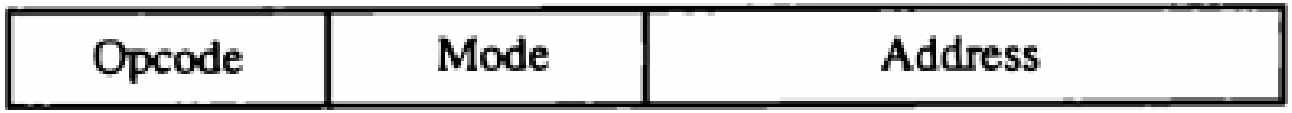
Addressing Modes

- Addressing Mode
  - 1) To give programming versatility to the user
    - pointers to memory, counters for loop control, indexing of data,….
  - 2) To reduce the number of bits in the addressing field of the instruction
- Instruction Cycle
  - 1) Fetch the instruction from memory and PC + 1
  - 2) Decode the instruction
  - 3) Execute the instruction
- Program Counter (PC)
  - PC keeps track of the instructions in the program stored in memory
  - PC holds the address of the instruction to be executed next
  - PC is incremented each time an instruction is fetched from memory

Addressing Modes

- Addressing Mode of the Instruction
- 1) Distinct Binary Code
  - A **Distinct Binary Code** implies that different binary patterns or codes are used to represent different instructions, memory addresses, or operand types.
- 2) Single Binary Code
  - A **Single Binary Code** implies that a single, uniform code is used for specifying an address or operation across multiple instructions.

Figure 8-6 Instruction format with mode field.



Addressing Modes

- Most addressing modes modify the address field of the instruction, two modes need no address field at all. These are the implied and immediate modes.
- **Implied Addressing Mode**: It is also known as Implicit Addressing Mode. It's an addressing mode where the operand is implicitly specified by the instruction itself.
  - Examples
    - COM: Complement Accumulator Operand in AC is implied in the definition of the instruction
    - PUSH: The stack push Operand is implied to be on top of the stack
- **Immediate Addressing Mode**: The operand is specified directly in the instruction itself.
  - Example: ADD R1, #5 (Add the value 5 to the contents of register R1)

Addressing Modes

- The following is the list of addressing modes, which modifies the address field of the instruction.
- **Register Addressing Mode**: The operand is stored in a register, and the instruction specifies the register.
  - Example: ADD R1, R2 ($R1 \leftarrow R2$).
- **Direct Addressing Mode** (Absolute Addressing): The effective address of the operand is given explicitly in the instruction.
  - Example: ADD R1, [2000] ($R1 \leftarrow M[2000]$).
- **Indirect Addressing Mode**: The address of the operand is found in a register or memory location specified in the instruction.
  - Example: ADD R1, [R2] ($R1 \leftarrow R2[X]$).

Addressing Modes

- **Relative Addressing Mode**: The effective address is determined by adding an offset (immediate value) to the current program counter (PC).
  - Example: JMP [PC + 4] (Jump to the instruction located at the current PC plus 4).
- **Indexed Addressing Mode**: The effective address is calculated by adding an index (often stored in a register) to a base address.
  - Example: ADD R1, [R2 + X] (Add the contents of memory location R2 + X to register R1, where X is an index value).
- **Base-Register Addressing Mode**: Similar to indexed addressing, the base address is stored in a specific register, and the effective address is calculated by adding a constant offset to the base register.
  - Example: ADD R1, [Base + 4] (Add the contents of the memory location that is Base register + 4 to register R1).

Addressing Modes

- **Auto-Increment Addressing Mode**: The effective address is the content of a register, but after the operand is accessed, the register is automatically incremented.
  - *Example:* ADD R1, [R2]+ (Add the contents of the memory location in R2 to R1, then increment R2).
- **Auto-Decrement Addressing Mode**: The effective address is the content of a register, but the register is automatically decremented before the operand is accessed.
  - *Example:* ADD R1, -[R2] (Decrement R2, then add the contents of the memory location pointed to by R2 to register R1).

Addressing Modes

- **Stack Addressing Mode**: The operand is implicitly on the top of the stack, with the stack pointer used as the reference.
  - Example: PUSH R1 (Push the contents of R1 onto the stack).
- **Register Indirect with Displacement**: The effective address is the sum of a base register and a displacement value provided in the instruction.
  - Example: MOV R1, [R2 + 8] (Move the contents of memory address R2 + 8 into R1).

Numerical Example

TABLE 8-4 Tabular List of Numerical Example

Addressing Mode	Effective Address	Content of *AC*
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

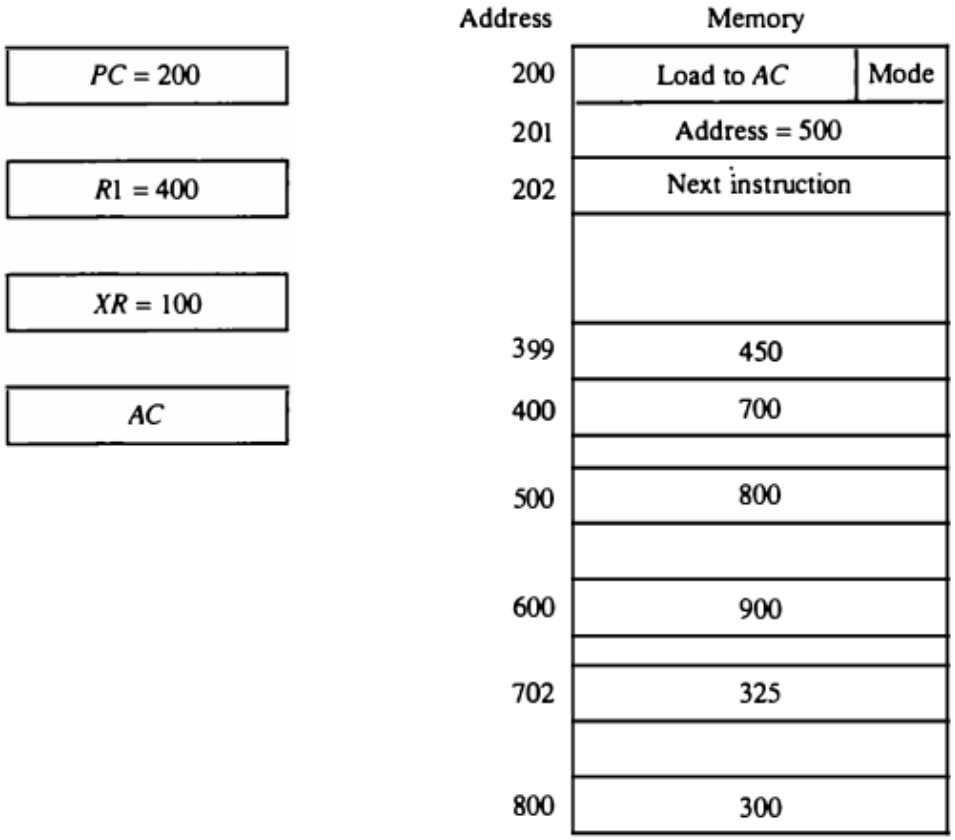


Figure 8-7 Numerical example for addressing modes.

Data Transfer and Manipulation

- **Instruction Set**: A collection of instructions that allow users to perform computational tasks.
  - **Key Differences**:
    - **Operands**: How they are determined via addressing and modes.
    - **Operation Codes**: Binary code assignments differ across computers.
    - **Symbolic Names**: Assembly language notation can vary between systems.
- **Categories of Computer Instructions**:
  - **Data Transfer Instructions**: Moving data between registers, memory, and I/O.
  - **Data Manipulation Instructions**: Performing arithmetic, logic, and shift operations.
  - **Program Control Instructions**: Directing program flow (e.g., jumps, subroutines).

Data Transfer

- **Data Transfer**: Movement of data between registers, memory locations, and I/O devices.
- **Types of Data Transfer**:
  - Between registers
  - Between register and memory
  - Between I/O devices and CPU
  - Between CPU and memory
- Efficient data transfer is critical for the overall performance of the system.

Data Transfer

- **Types of Data Transfer**:
  - Memory to Register Transfer
    - *Example*: MOV R1, [1000]
  - Register to Memory Transfer
    - *Example*: MOV [2000], R1
  - Register to Register Transfer
    - *Example*: MOV R1, R2
  - I/O Transfers: Communication between the CPU and external devices.
    - *Example*: IN (Input) and OUT (Output)

Data Transfer Instructions

- **Data Transfer Instructions**: Instructions that move data from one location to another.
  - Examples:
    - MOV: Move data between registers or memory.
    - LD (Load): Load data from memory to a register.
    - ST (Store): Store data from a register to memory.
    - PUSH/POP: Transfer data to/from the stack.
- Addressing Modes Used: Immediate, Direct, Indirect, Register.

TABLE 8-5 Typical Data Transfer Instructions

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

Addressing Modes in Data Transfer

TABLE 8-6 Eight Addressing Modes for the Load Instruction

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

Data Manipulation

- **Data Manipulation Instructions**: Instructions that operate on data (arithmetic, logical, bitwise, shift/rotate).
- Examples of Manipulation Instructions:
  - Arithmetic Operations (Addition, Subtraction)
  - Logical Operations (AND, OR, NOT, XOR)
  - Shift Operations (Left, Right)
  - Rotate Operations (Rotate Left, Rotate Right)

Data Manipulation

- **Basic Arithmetic Instructions**:
  - Addition (ADD)
    - Example: ADD R1, R2
  - Subtraction (SUB)
    - Example: SUB R1, R2
  - Multiplication (MUL)
    - Example: MUL R1, R2
  - Division (DIV)
    - Example: DIV R1, R2

TABLE 8-7 Typical 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

Data Manipulation

- **Basic Logical Instructions:**
  - AND Operation:
    - Example: AND R1, R2 (Performs bitwise AND on contents of R1 and R2)
  - OR Operation:
    - Example: OR R1, R2
  - XOR Operation:
    - Example: XOR R1, R2
  - NOT Operation:
    - Example: NOT R1 (Inverts the bits of R1)

TABLE 8-8 Typical Logical and Bit Manipulation Instructions

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

Data Manipulation

- **Basic Shift Instructions**: Move bits in a register left or right, filling with zeros or ones.
- Types:
  - Logical Shift: Shifts bits left or right and fills the shifted-out bit with 0.
    - Example: SHL R1, 1 (Shift bits in R1 left by 1)
  - Arithmetic Shift: Preserves the sign bit (for signed numbers).
    - Example: SAR R1, 1 (Arithmetic shift right by 1)

TABLE 8-9 Typical Shift Instructions

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 through carry	RORC
Rotate left through carry	ROLC

Data Manipulation

- **Basic Rotate Operations**: Rotates the bits in a register left or right, with the bits that "fall off" re-entering on the opposite side.
- Types:
  - Rotate Left (ROL): Bits shifted out of the left end are reintroduced at the right.
  - Rotate Right (ROR): Bits shifted out of the right end are reintroduced at the left.
    - Example: ROL R1, 1 (Rotate bits in R1 left by 1)

Program Control

- **Sequential Instruction Execution**:
  - Instructions are stored in consecutive memory locations.
  - The CPU fetches and executes instructions from memory in sequence.
  - After each fetch, the program counter (PC) is incremented to point to the next instruction.
- For data transfer or manipulation instructions, the PC continues sequentially.
- **Program control instructions** can alter the flow by changing the PC value, breaking the instruction sequence.

Program Control

- **Program control instructions -** instructions that alter the program counter to control the flow of execution.
- Types:
  - ***Branch Instructions***: Change the flow based on conditions.
  - ***Jump Instructions***: Alter the flow without conditions.
  - ***Skip Instructions***: Skip instructions based on a condition.
  - ***Call/Return Instructions***: Used with subroutines.
  - ***Compare/Test Instructions***: Set conditions for conditional branching.
- Control instructions break the sequence of instruction execution and enable branching to different program segments.

Program Control

- **Typical Program Control Instructions:**

TABLE 8-10 Typical Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TST

Program Control

- **Branch Instruction:**
  - One-address instruction: Written as BR ADR (ADR = symbolic address).
  - Transfers control to a new location by loading the PC with the branch address.
  - Example
    - BR ADR [Branch to location Address]
- **Unconditional Branch**: Always branches to the specified address without any conditions.
- **Conditional Branch**: Branches only if a certain condition is met (e.g., branch if zero, branch if positive). If the condition is not met, execution continues sequentially.
  - Example:
    - CMP R1, #0 ; Compare R1 with 0
    - BEQ ADR ; Branch to ADR if zero

Program Control

- **Jump Instructions**: Jump instructions are used to transfer control to a different part of the program, altering the normal sequential flow of execution.
  - Enable branching, loops, and control structures in programming.
  - The program counter (PC) is updated with the target address specified by the jump instruction.
- **Unconditional Jumps**: Always transfer control to the specified address without any condition.
  - Example: JMP LABEL ; Jump to the instruction at LABEL
- **Conditional Jumps**: Transfer control only if a specified condition is met (based on status flags).
  - Examples:
    - Branch if Zero (BEQ): Jumps if the Zero flag is set.
      - BEQ LABEL ; Jump to LABEL if Zero flag (Z) = 1
    - Branch if Not Zero (BNE): Jumps if the Zero flag is not set.
      - BNE LABEL ; Jump to LABEL if Zero flag (Z) = 0

Program Control

- **Skip Instructions**: A **zero-address instruction** that skips the next instruction if a condition is met.
  - **Operation:** Increments the PC during the **execute phase** to skip the next instruction.
  - **Conditional Skip**: if the condition is not met, an **unconditional branch** is used to manage control flow.
    - Example:
      - SKIPNZ ; Skip the next instruction if the result is not zero
      - JMP ADR ; Jump to address ADR
- **Call Instruction**:
  - Used to transfer control to a subroutine while saving the return address.
  - The subroutine is a block of reusable code, and after it executes, the control returns to the calling program.
  - Example: CALL SUBROUTINE ; Call a subroutine

Program Control

- **Return Instruction**: Transfers control back to the instruction following the call.
  - Example: RET ; Return from subroutine
- **Compare Instruction**:
  - Performs a subtraction but does not store the result.
  - Sets **status flags** (e.g., carry, zero, sign) based on the result of the operation.
  - **Example**: CMP R1, R2 ; Compare R1 and R2
- **Test Instruction**: Performs a **logical AND** between two operands and sets the status flags without changing the operands.
  - **Example**: TEST R1, R2 ; Test R1 AND R2

Program Control

- **Status Flags**: **Carry**, **Zero**, **Sign**, and **Overflow** flags are updated and used in subsequent conditional branch instructions.
- **Conditional Branching**: Based on the result of the **compare** or **test** instruction, the program can branch based on the updated status flags.
  - **Example**:
    - CMP R1, #0 ; Compare R1 with 0
    - BEQ ZERO_LABEL ; Branch to ZERO_LABEL if Zero Flag is set
- **Common Conditions**:
  1. **Branch if Zero (BEQ)**
  2. **Branch if Not Zero (BNE)**
  3. **Branch if Positive (BGT)**
  4. **Branch if Negative (BLT)**

Program Control

- **Status bit conditions** (or flags) are special bits in the status register of a CPU that store the outcome of arithmetic or logical operations.
- They provide information about the result of an operation, enabling conditional branching, looping, and decision-making in programs.
- **Common Status Bits**:
  - Bit C (carry) is set to 1 if the end carry C_8 is 1. It is cleared to 0 if the carry is 0.
  - Bit S (sign) is set to 1 if the highest-order bit F, is 1. It is set to 0 if the bit is 0.
  - Bit Z (zero) is set to 1 if the output of the ALU contains all 0's. !t is cleared to 0 otherwise. In other words, Z = 1 if the output is zero and Z = 0 if the output is not zero.
  - Bit V (overflow) is set to 1 if the exclusive-OR of the last two carries is equal to 1, and cleared to 0 otherwise. This is the condition for an overflow when negative numbers are in 2's. For the 8-bit ALU, V = 1 if the output is greater than + 127 or less than -128.

Program Control

- Status Register Bits

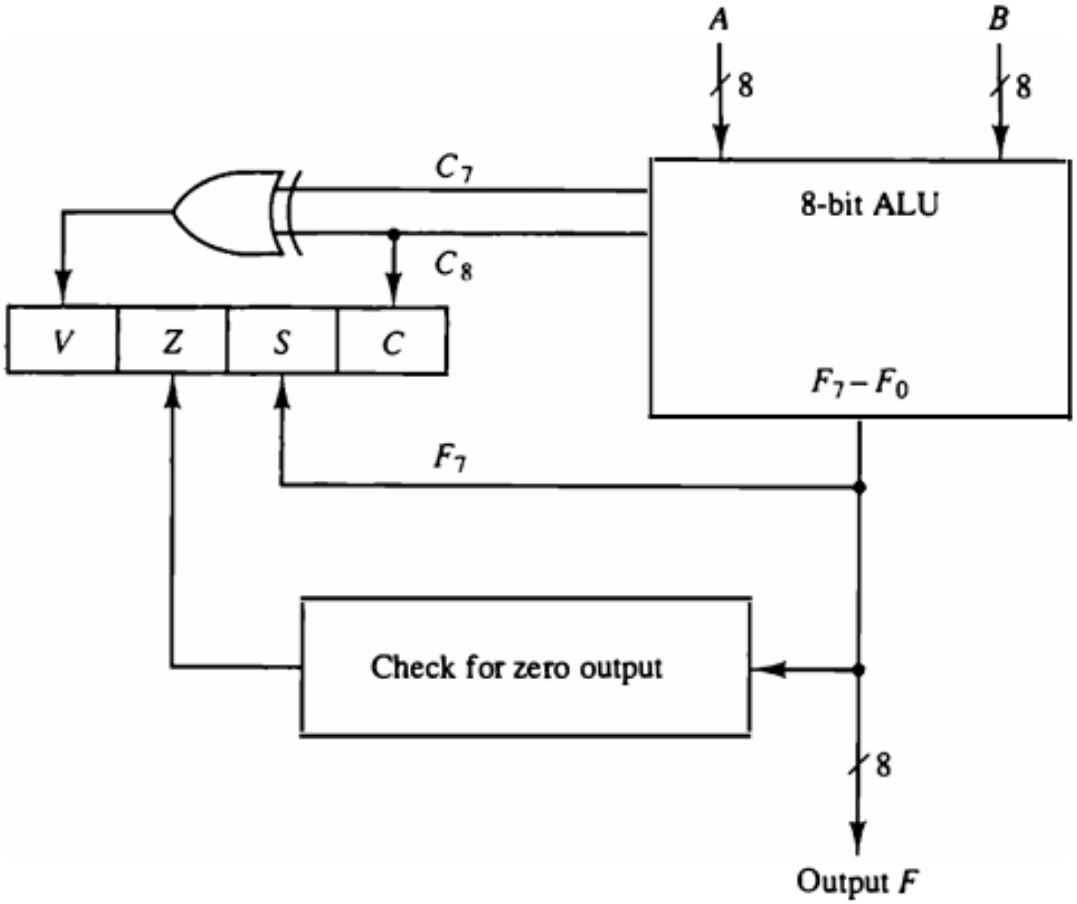


Figure 8-8 Status register bits.

Program Control

TABLE 8-11 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	$A \geq B$
BLO	Branch if lower	$A < B$
BLOE	Branch if lower or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$
Signed compare conditions ($A - B$)		
BGT	Branch if greater than	$A > B$
BGE	Branch if greater or equal	$A \geq B$
BLT	Branch if less than	$A < B$
BLE	Branch if less or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$

Program Control

- **Program Interrupt**
  - Transfer program control from a currently running program to another service program as a result of an external or internal generated request
  - Control returns to the original program after the service program is executed
- Differences between Interrupt Service Program and Subroutine Call:
  - 1) An interrupt is initiated by an internal or external signal (except for software interrupt)
    - A subroutine call is initiated from the execution of an instruction (CALL)
  - 2) The address of the interrupt service program is determined by the hardware
    - The address of the subroutine call is determined from the address field of an instruction
  - 3) An interrupt procedure stores all the information necessary to define the state of the CPU
    - A subroutine call stores only the program counter (Return address)

Program Control

- The state of the CPU at the end of the execute cycle (when the interrupt is recognized) is determined from:
  - 1. The content of the program counter
  - 2. The content of all processor registers
  - 3. The content of certain status conditions
- The collection of all status bit conditions in the CPU is sometimes called a **program status word or PSW**. The PSW is stored in a separate hardware register and contains the status information that characterizes the state of the CPU.
- Two CPU Operating Modes
  - Supervisor (System) Mode: Executing Privileged Instruction
    - When the CPU is executing a program that is part of the operating system
  - User Mode: Executing User program

Program Control

- **Types of Interrupts**
  - 1) ***External Interrupts***: come from an I/O device, from a timing device, from a circuit monitoring the power supply, or from any other external source.
  - 2) ***Internal Interrupts or TRAP***: caused by register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation.
  - 3) ***Software Interrupts***: initiated by executing an instruction (INT or RST) and is used by the programmer to initiate an interrupt procedure at any desired point in the program.

Program Control

- Storage information holds PC, CPU registers and Status Condition.