Unit II: ALU Design

ALU Design

- An instruction code is a group of bits that instruct the computer to perform a specific operation. It is usually divided into parts, each having its own interpretation.
- The most basic part of an instruction code is its operation part.
- The operation code of an instruction is a group of bits that define such operations as add, subtract, multiply, shift, and complement.
- The number of bits required for the operation code of an instruction depends on the total number of operations available in the computer.

ALU Design

- The operation code must consist of at least n bits for a given 2^n (or less) distinct operations.
  - As an illustration, consider a computer with 64 distinct operations, one of them being an ADD operation. The operation code consists of six bits, with a bit configuration 110010 assigned to the ADD operation.
  - When this operation code is decoded in the control unit, the computer issues control signals to read an operand from memory and add the operand to a processor register.

ALU Design

- The operation part of an instruction code specifies the operation to be performed. This operation must be performed on some data stored in processor registers or in memory.
- An instruction code must therefore specify not only the operation but also the registers or the memory words where the operands are to be found, as well as the register or memory word where the result is to be stored.
- Memory words can be specified in instruction codes by their address. Processor registers can be specified by assigning to the instruction another binary code of k bits that specifies one of 2^k registers.

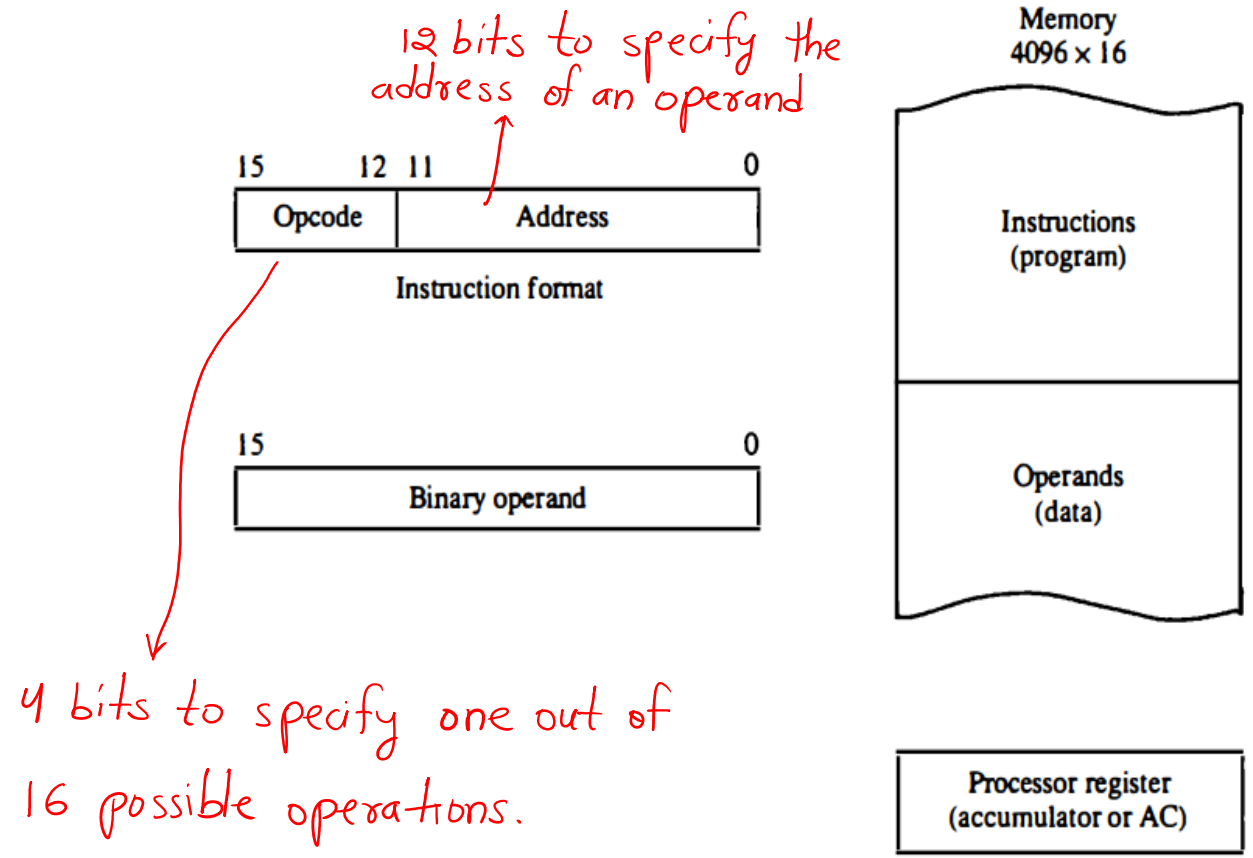
ALU Design

- There are many variations for arranging the binary code of instructions, and each computer has its own instruction code format.
- Instruction code formats are conceived by computer designers who specify the architecture of the computer.

ALU Design

Stored Program Organization

- The simplest way to organize a computer is to have one processor register and an instruction code format with two parts.
  - The first part specifies the operation to be performed and the second specifies an address.
  - The memory address tells the control where to find an operand in memory.



ALU Design

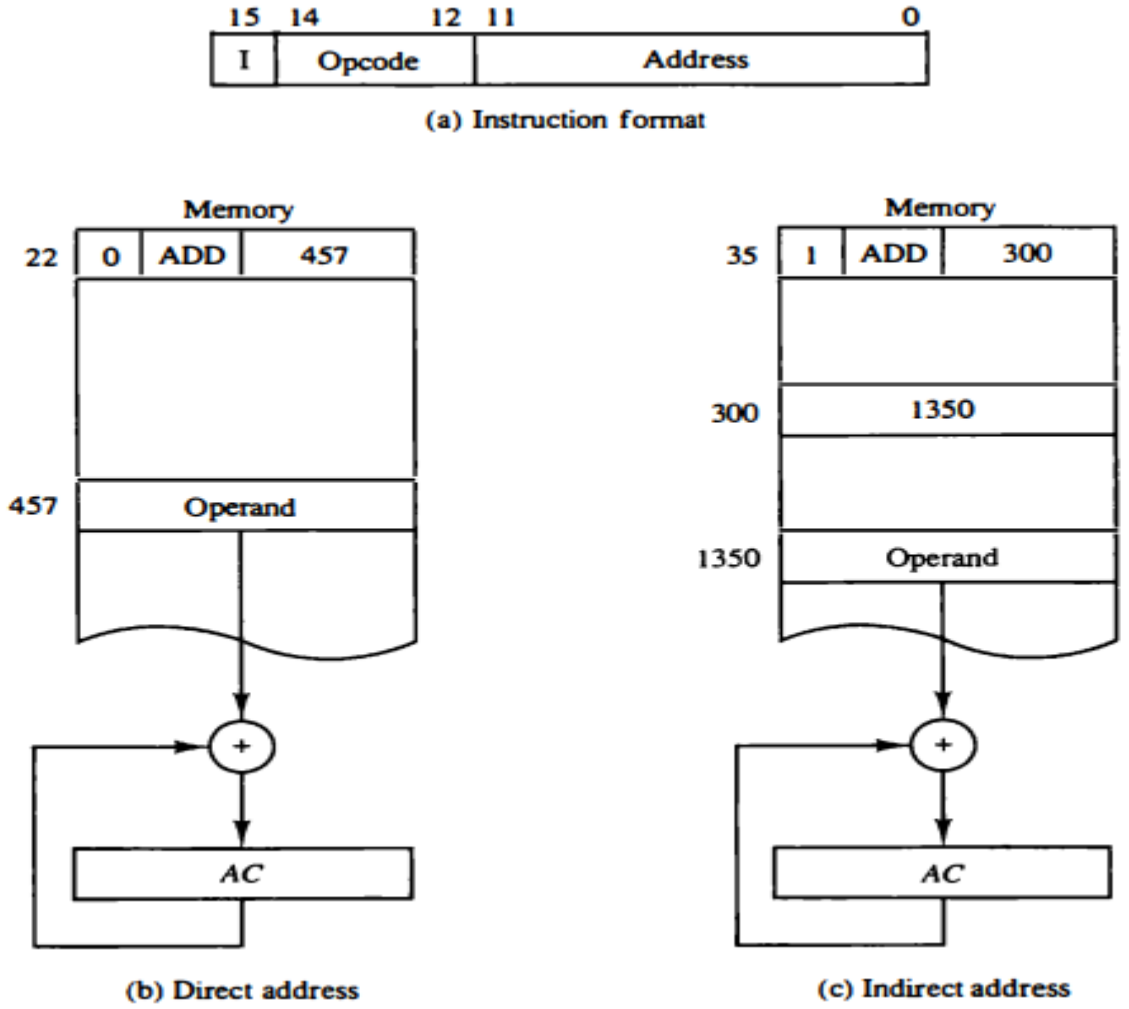
- It is sometimes convenient to use the address bits of an instruction code not as an address but as the actual operand.
  - When the second part of an instruction code specifies an operand, the instruction is said to have an immediate operand.
  - When the second part specifies the address of an operand, the instruction is said to have a direct address.
  - Indirect address: the bits in the second part of the instruction designate an address of a memory word in which the address of the operand is found.

ALU Design

- One bit of the instruction code can be used to distinguish between a direct and an indirect address.

Effective Address:

- The address of the operand in a computation-type instruction, or
- The target address in a branch-type instruction.



ALU Design

Computer Registers

- Computer instructions are normally stored in consecutive memory locations and are executed sequentially unless a branch instruction is encountered.
- The control reads an instruction from a specific address in memory and executes it. It then continues by reading the next instruction in sequence and executes it, and so on.
- This type of instruction sequencing needs a counter to calculate the address of the next instruction after execution of the current instruction is completed.

ALU Design

Computer Registers

- It is also necessary to provide a register in the control unit for storing the instruction code after it is read from memory.
- The computer needs processor registers for manipulating data and a register for holding a memory address.

ALU Design

Computer Registers

- Registers for the basic computer.

Register symbol	Number of bits	Register name	Function
DR	16	Data register	Holds memory operand
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
PC	12	Program counter	Holds address of instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds input character
OUTR	8	Output register	Holds output character

ALU Design

Computer Registers

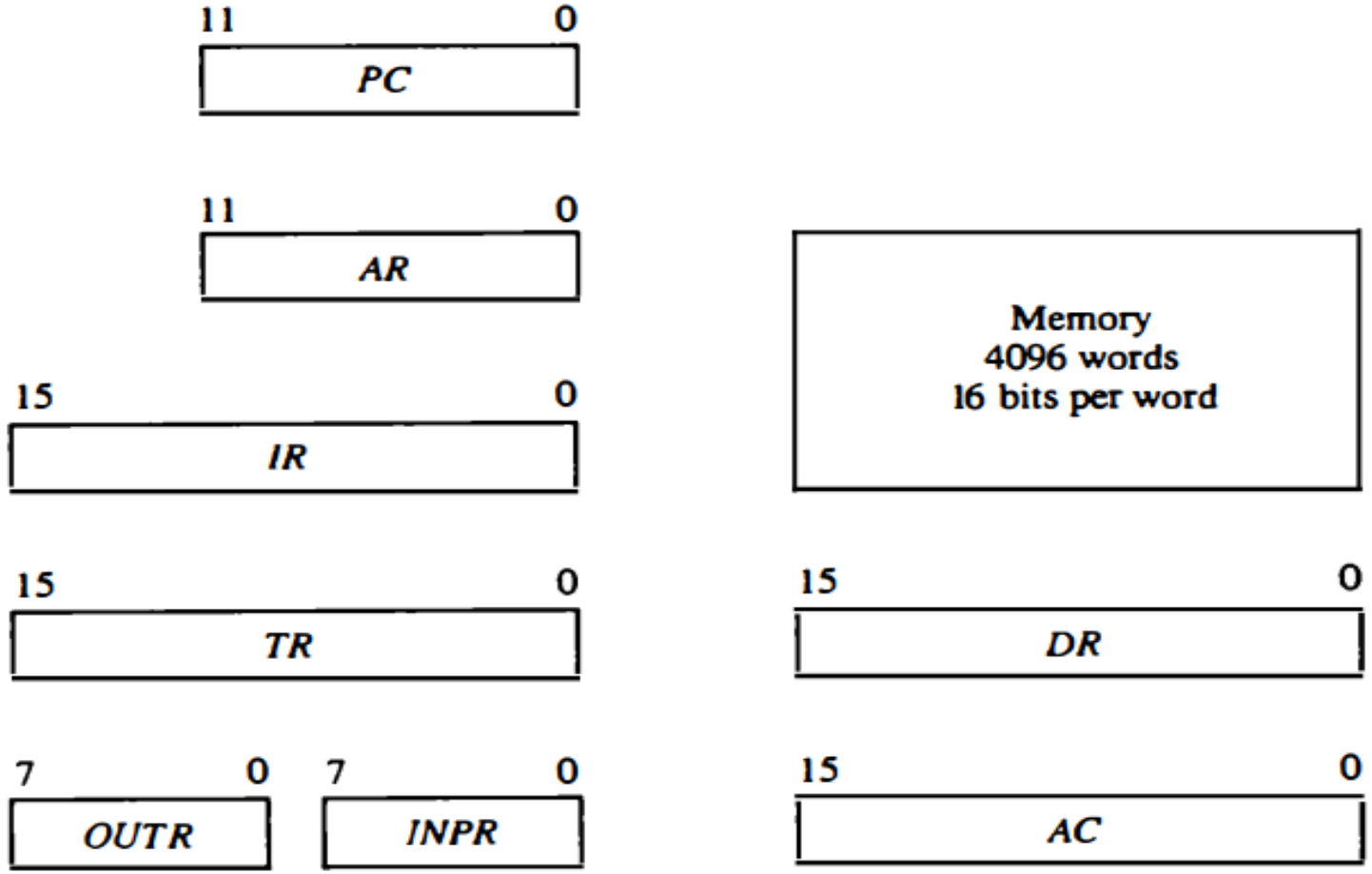


Figure: Basic computer registers and memory.

ALU Design

Computer Instructions

- The basic computer has three instruction code formats.
- Each format has 16 bits.

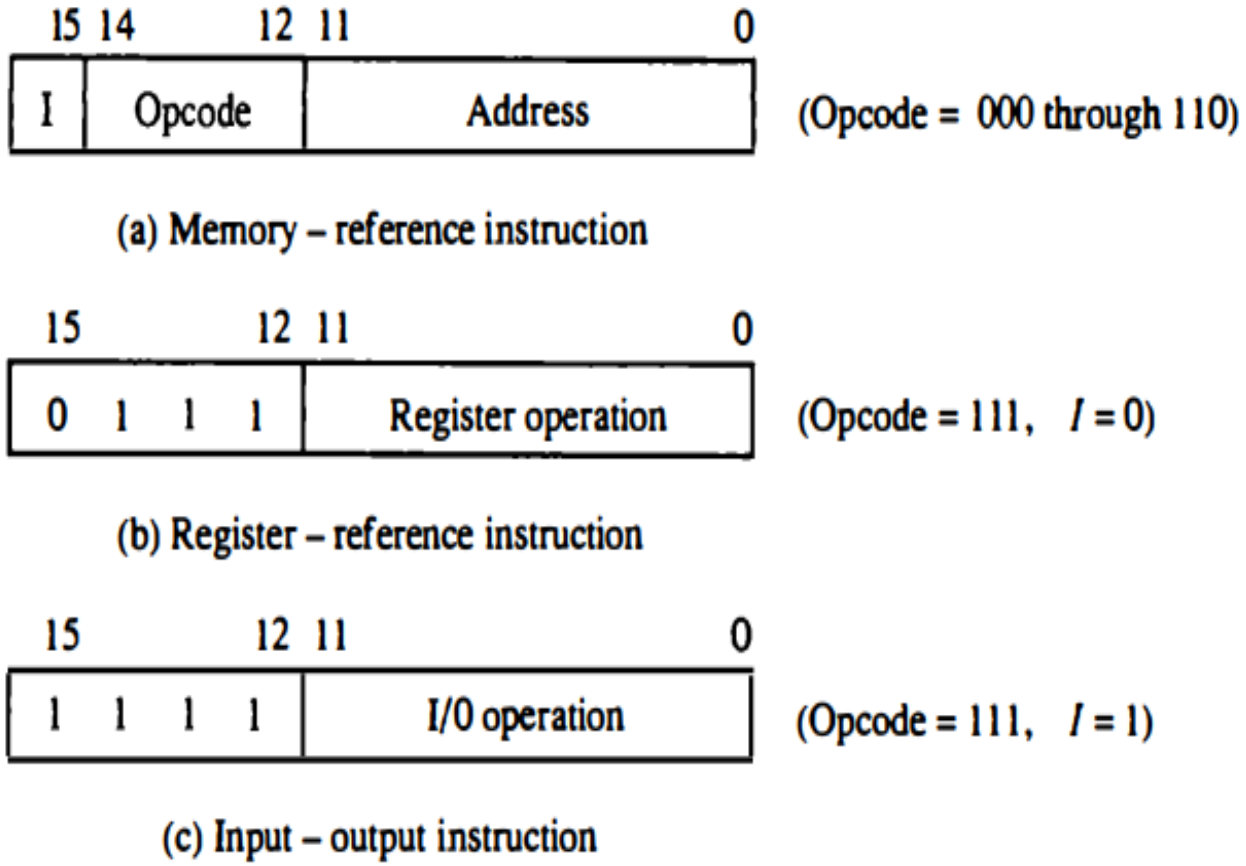


Figure: Basic computer instruction formats.

ALU Design

Computer Instructions

- The operation code (opcode) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered.
- A memory-reference instruction uses 12 bits to specify an address and one bit to specify the addressing mode I.
  - I is equal to 0 for direct address and to 1 for indirect address.
- The register reference instructions are recognized by the operation code 111 with a 0 in the leftmost bit (bit 15) of the instruction. An operand from memory is not needed; therefore, the other 12 bits are used to specify the operation or test to be executed.

ALU Design

Computer Instructions

- An input-output instruction does not need a reference to memory and is recognized by the operation code 111 with a 1 in the leftmost bit of the instruction. The remaining 12 bits are used to specify the type of input-output operation or test performed.

ALU Design

Computer Instructions

Symbol	Hexadecimal code $I = 0$	Hexadecimal code $I = 1$	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load memory word to AC
STA	3xxx	Bxxx	Store content of AC in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instruction if AC positive
SNA	7008		Skip next instruction if AC negative
SZA	7004		Skip next instruction if AC zero
SZE	7002		Skip next instruction if E is 0
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

Figure: Basic computer instructions.

ALU Design

Stack Organization

- A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved.
- The stack in digital computers is a memory unit with an address register. The register that holds the address for the stack is called a stack pointer (SP) and its value always points at the top item in the stack.
- The two operations of a stack are the insertion and deletion of items. The operation of insertion is called push (or push-down) and the operation of deletion is called pop (or pop-up).
  - These operations are simulated in computer stack by incrementing or decrementing the stack pointer register.

ALU Design

Register Stack

- A stack can be placed in a portion of a large memory, or it can be organized as a collection of a finite number of memory words or registers.
- The stack pointer register SP contains a binary number whose value is equal to the address of the word that is currently on top of the stack.

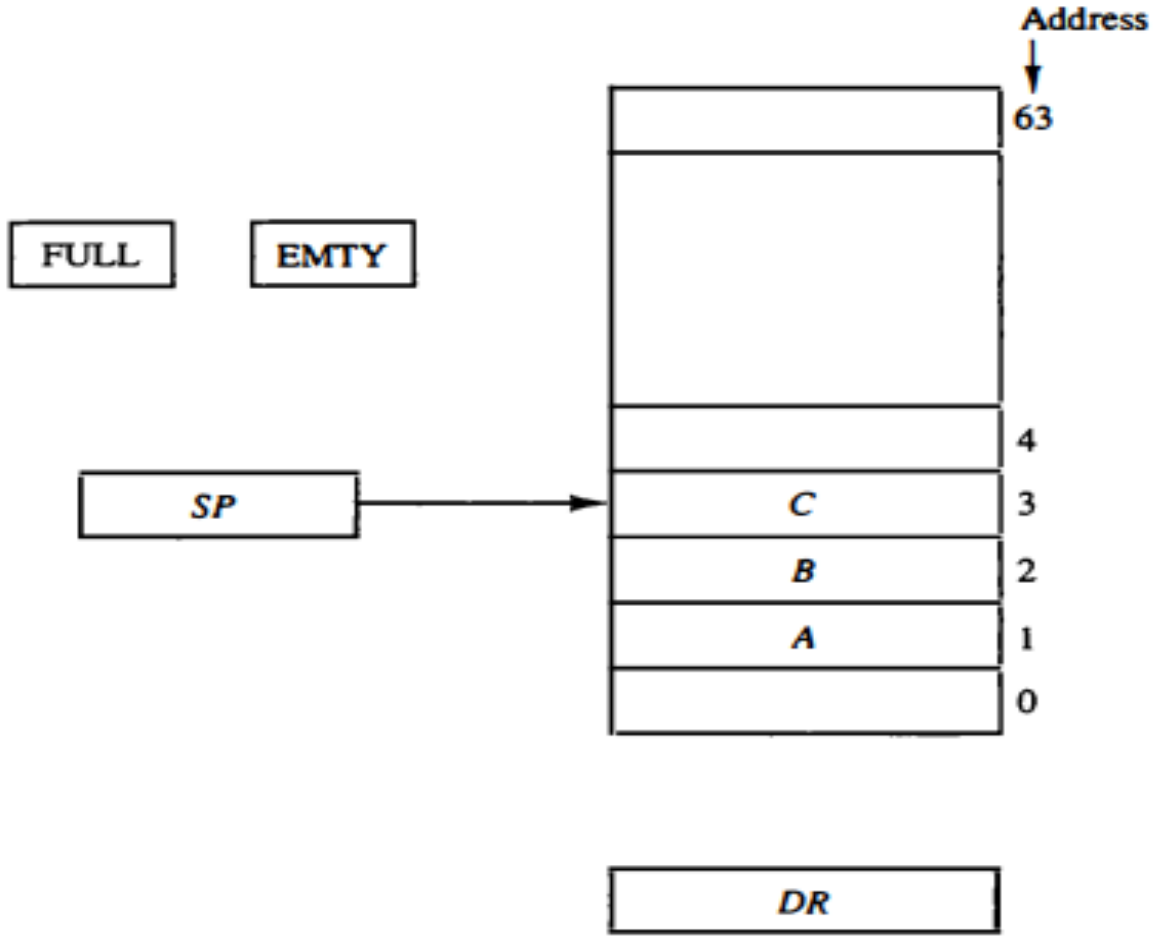


Figure: Block diagram of a 64-word stack.

ALU Design

Register Stack

- The one-bit register FULL is set to 1 when the stack is full, and the one-bit register EMTY is set to 1 when the stack is empty of items.
- DR is the data register that holds the binary data to be written into or read out of the stack.
- Initially, SP is cleared to 0, EMTY is set to 1, and FULL is cleared to 0, so that SP points to the word at address 0 and the stack is marked empty and not full. If the stack is not full (if FULL = 0), a new item is inserted with a push operation.

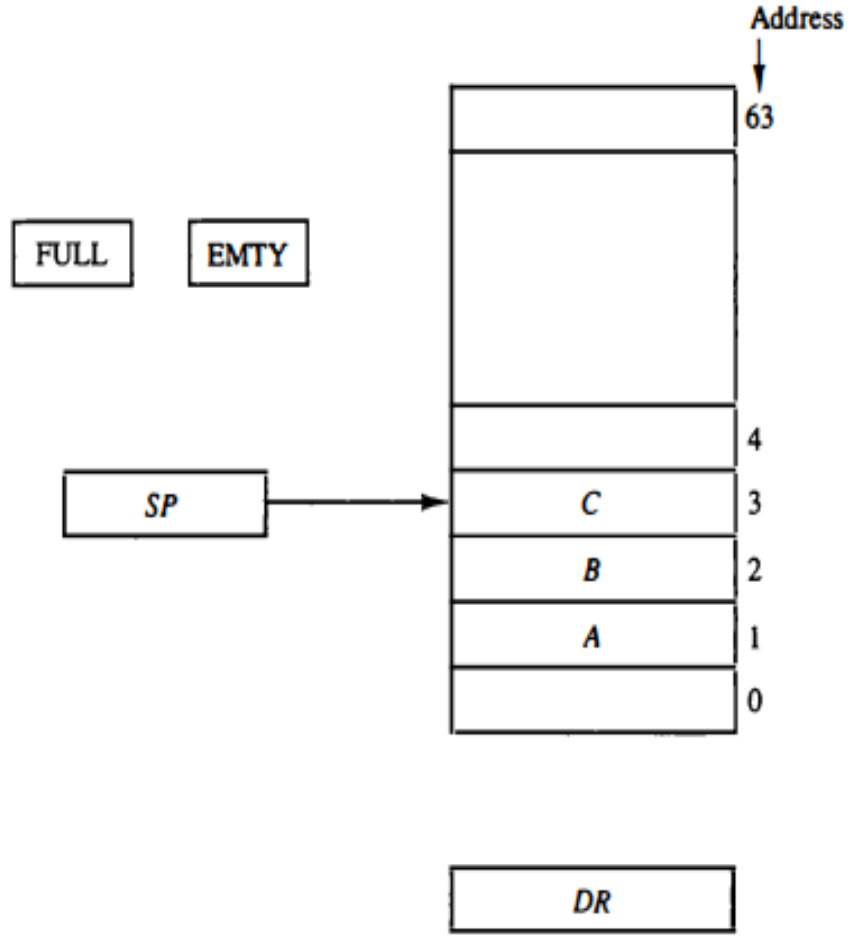


Figure: Block diagram of a 64-word stack.

ALU Design

Register Stack

- The push operation is implemented with the following sequence of microoperations:

$SP \leftarrow SP + 1$	Increment stack pointer
$M[SP] \leftarrow DR$	Write item on top of the stack
If $(SP = 0)$ then $(FULL \leftarrow 1)$	Check if stack is full
$EMTY \leftarrow 0$	Mark the stack not empty

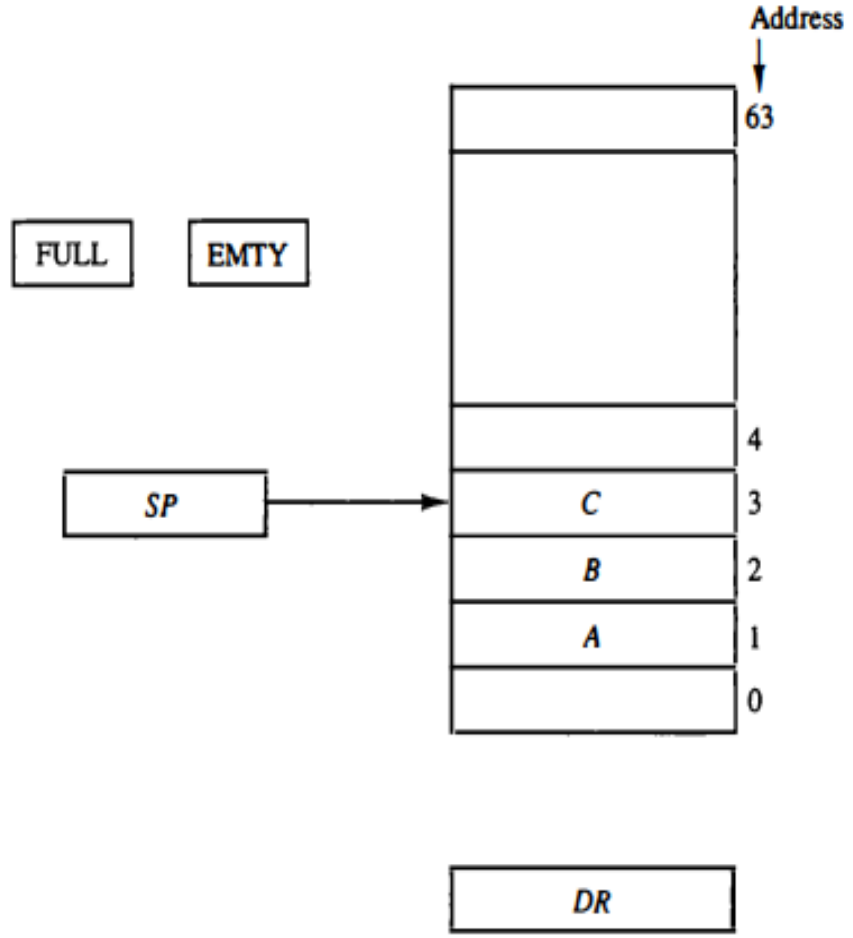


Figure: Block diagram of a 64-word stack.

ALU Design

Register Stack

- The pop operation consists of the following sequence of microoperations:

$DR \leftarrow M[SP]$	Read item from the top of stack
$SP \leftarrow SP - 1$	Decrement stack pointer
If $(SP = 0)$ then $(EMTY \leftarrow 1)$	Check if stack is empty
$FULL \leftarrow 0$	Mark the stack not full

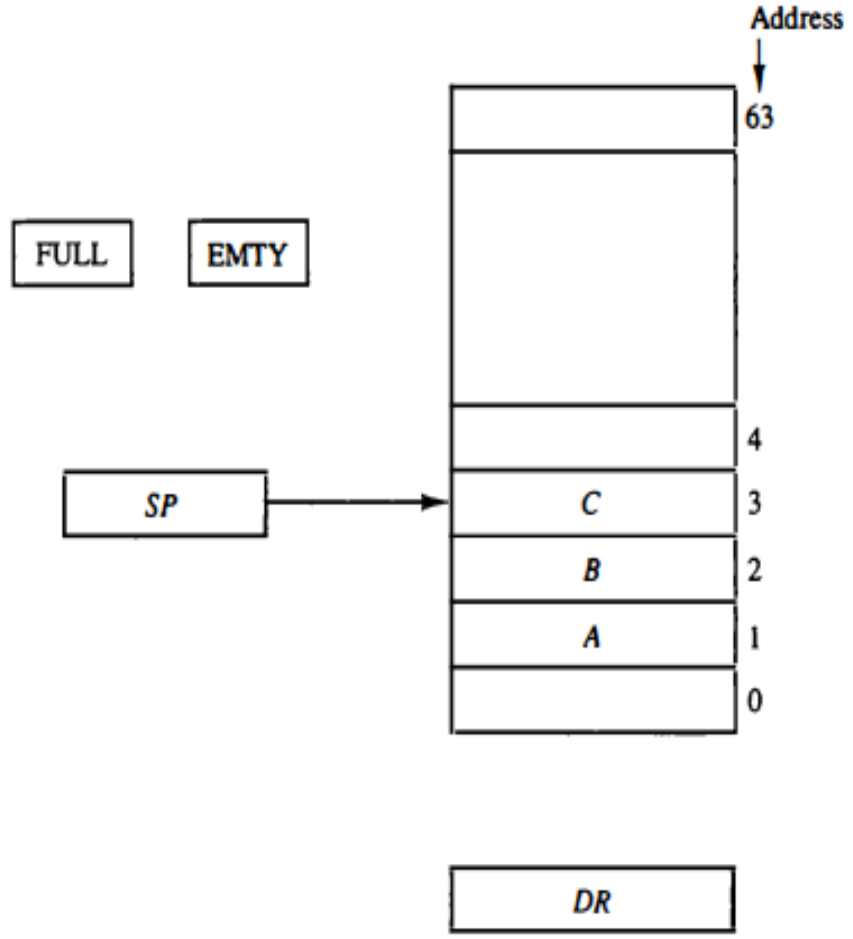


Figure: Block diagram of a 64-word stack.

ALU Design

Memory Stack

- The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer.
- The program counter PC points at the address of the next instruction in the program.
- The address register AR points at an array of data.
- The stack pointer SP points at the top of the stack.
- The three registers are connected to a common address bus, and either one can provide an address for memory.

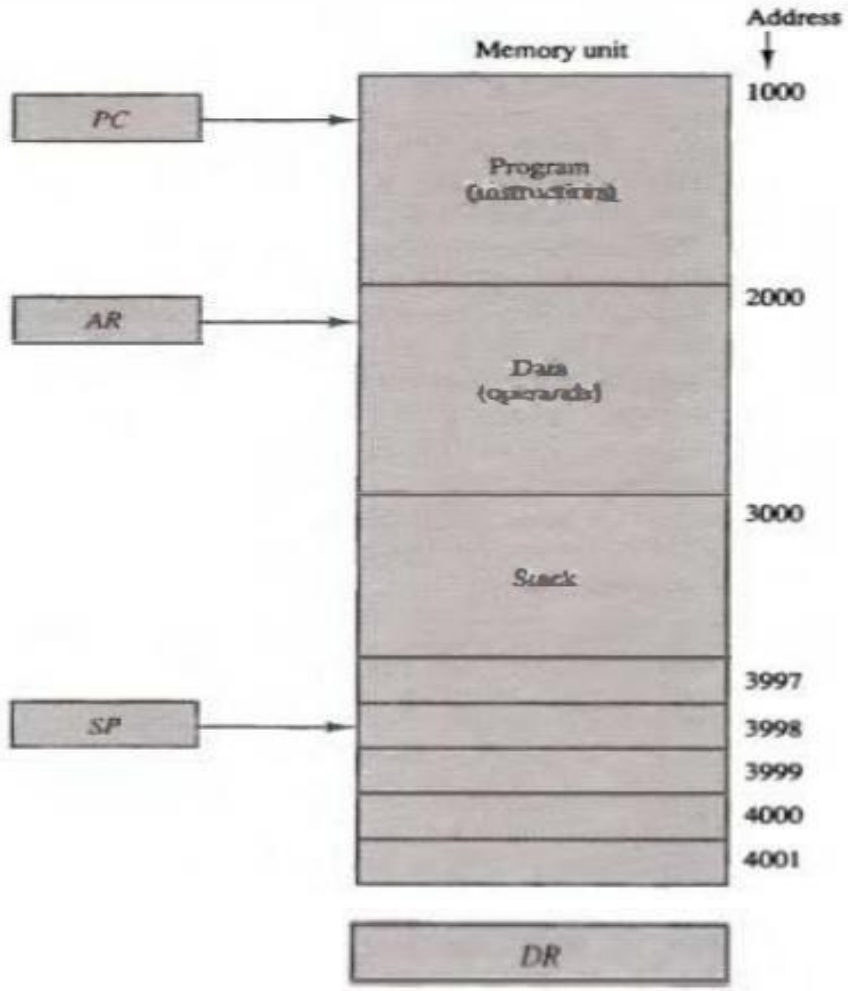


Figure: Computer memory with program, data, and stack segments.

ALU Design

Memory Stack

- A new item is inserted with the push operation as follows:

$$SP \leftarrow SP - 1$$

$$M[SP] \leftarrow DR$$

- A new item is deleted with a pop operation as follows:

$$DR \leftarrow M[SP]$$

$$SP \leftarrow SP + 1$$

- Most computers do not provide hardware to check for stack overflow (full stack) or underflow (empty stack).
  - The stack limits can be checked by using two processor registers: one to hold the upper limit (3000 in this case), and the other to hold the lower limit (4001 in this case).

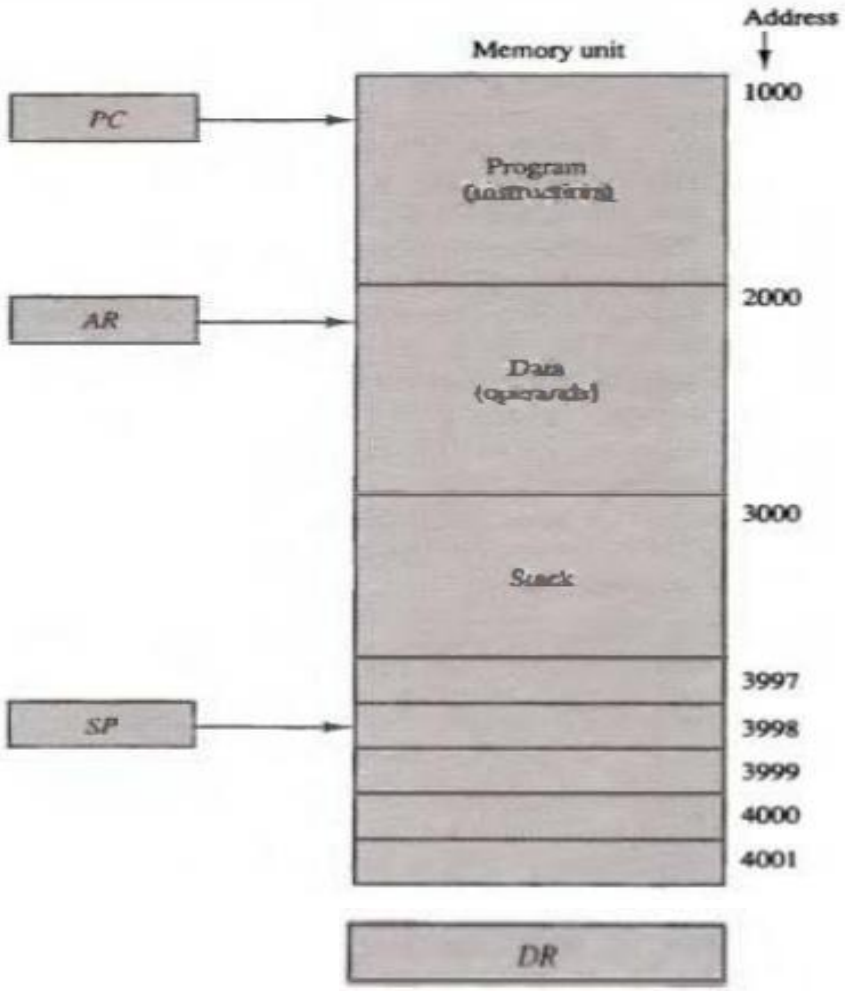


Figure: Computer memory with program, data, and stack segments.

ALU Design

Memory Stack

Advantage:

- CPU can refer to it without having to specify an address because the address is always available and automatically updated in the stack pointer.

ALU Design

Stack Organization

Reverse Polish Notation:

- A stack organization is very effective for evaluating arithmetic expressions. The common arithmetic expressions are written in *infix notations*.
- Consider the simple arithmetic expression: $A * B + C * D$
  - To evaluate this arithmetic expression, it is necessary to compute the product A * B, store this product while computing C * D, and then sum the two products.
  - From this example we see that to evaluate arithmetic expressions in infix notation it is necessary to scan back and forth along the expression to determine the next operation to be performed.

ALU Design

Stack Organization

Reverse Polish Notation:

- Consider the simple arithmetic expression: $(3*4) + (5*6)$
- Postfix or reverse polish notation: 34*56*+

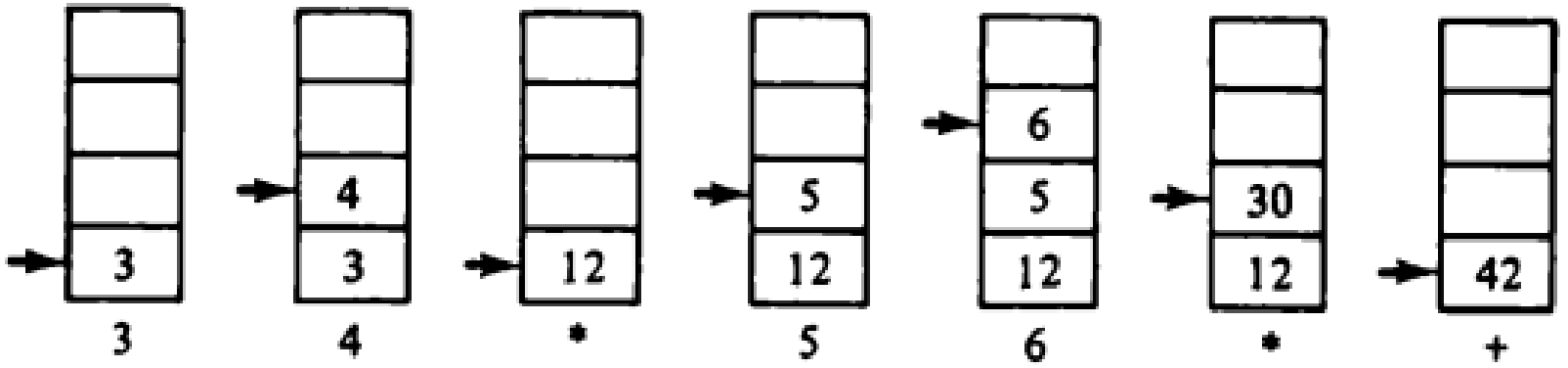


Figure: Stack operations to evaluate 3 * 4 + 5 * 6

ALU Design

Instruction Format

- The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are:
  - An operation code field that specifies the operation to be performed.
  - An address field that designates a memory address or a processor register.
  - A mode field that specifies the way the operand or the effective address is determined.

ALU Design

Instruction Format

- Computers may have instructions of several different lengths containing varying number of addresses.
- The number of address fields in the instruction format of a computer depends on the internal organization of its registers.
- Most computers fall into one of three types of CPU organizations:

  i. Single accumulator organization

  - All operations are performed with an implied accumulator register. The instruction format in this type of computer uses one address field. For example, ADD X.

ALU Design

Instruction Format

ii. General register organization

- Employ two or three address fields in their instruction format.

For example: ADD R1, R2, R3

ADD R1, R2

MOV R1, X

iii. Stack organization

- Computers with stack organization would have PUSH and POP instructions which require an address field. For example: PUSH X
- Operation-type instructions do not need an address field in stack-organized computers. For example: ADD

ALU Design

Instruction Format

- Influence of the number of addresses on the computer programs:
  - **Arithmetic statement**: $X = (A + B) * (C + D)$
  - **Assumption**: Operands are in memory addresses A, B, C, and D, and the result must be stored in memory at address X.
  - **Three-Address Instructions**

```
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
```

    - **Advantage:** results in short programs
    - **Disadvantage:** require too many bits to specify three addresses

ALU Design

Instruction Format

- Influence of the number of addresses on the computer programs:
  - **Arithmetic statement**: $X = (A + B) * (C + D)$
  - **Assumption**: Operands are in memory addresses A, B, C, and D, and the result must be stored in memory at address X.
  - **Two-Address Instructions**

```
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
```

ALU Design

Instruction Format

- Influence of the number of addresses on the computer programs:
  - **Arithmetic statement**: $X = (A + B) * (C + D)$
  - **Assumption**: Operands are in memory addresses A, B, C, and D, and the result must be stored in memory at address X.
  - **One-Address Instructions**

```
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
```

ALU Design

Instruction Format

- Influence of the number of addresses on the computer programs:
  - **Arithmetic statement**: $X = (A + B) * (C + D)$
  - **Assumption**: Operands are in memory addresses A, B, C, and D, and the result must be stored in memory at address X.
  - **Zero-Address Instructions**

```
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
```

ALU Design

Instruction Format

- Influence of the number of addresses on the computer programs:
  - **Arithmetic statement**: $X = (A + B) * (C + D)$
  - **Assumption**: Operands are in memory addresses A, B, C, and D, and the result must be stored in memory at address X.
  - **RISC Instructions**

```
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
```

ALU Design

Timing and Control

- The timing for all registers in the basic computer is controlled by a master clock generator.
- The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit.
- The clock pulses do not change the state of a register unless the register is enabled by a control signal.
- The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and microoperations for the accumulator.

ALU Design

Timing and Control

- There are two major types of control organization:
  1. Hardwired control
     - In the hardwired organization, the control logic is implemented with gates, flip-flops, decoders, and other digital circuits.
     - It has the advantage that it can be optimized to produce a fast mode of operation.
     - It requires changes in the wiring among the various components if the design has to be modified or changed.
  2. Microprogrammed control
     - In the microprogrammed organization, the control information is stored in a control memory.
     - The control memory is programmed to initiate the required sequence of microoperations.
     - In the microprogrammed control, any required changes or modifications can be done by updating the microprogram in control memory.

ALU Design

Block Diagram of Control Unit

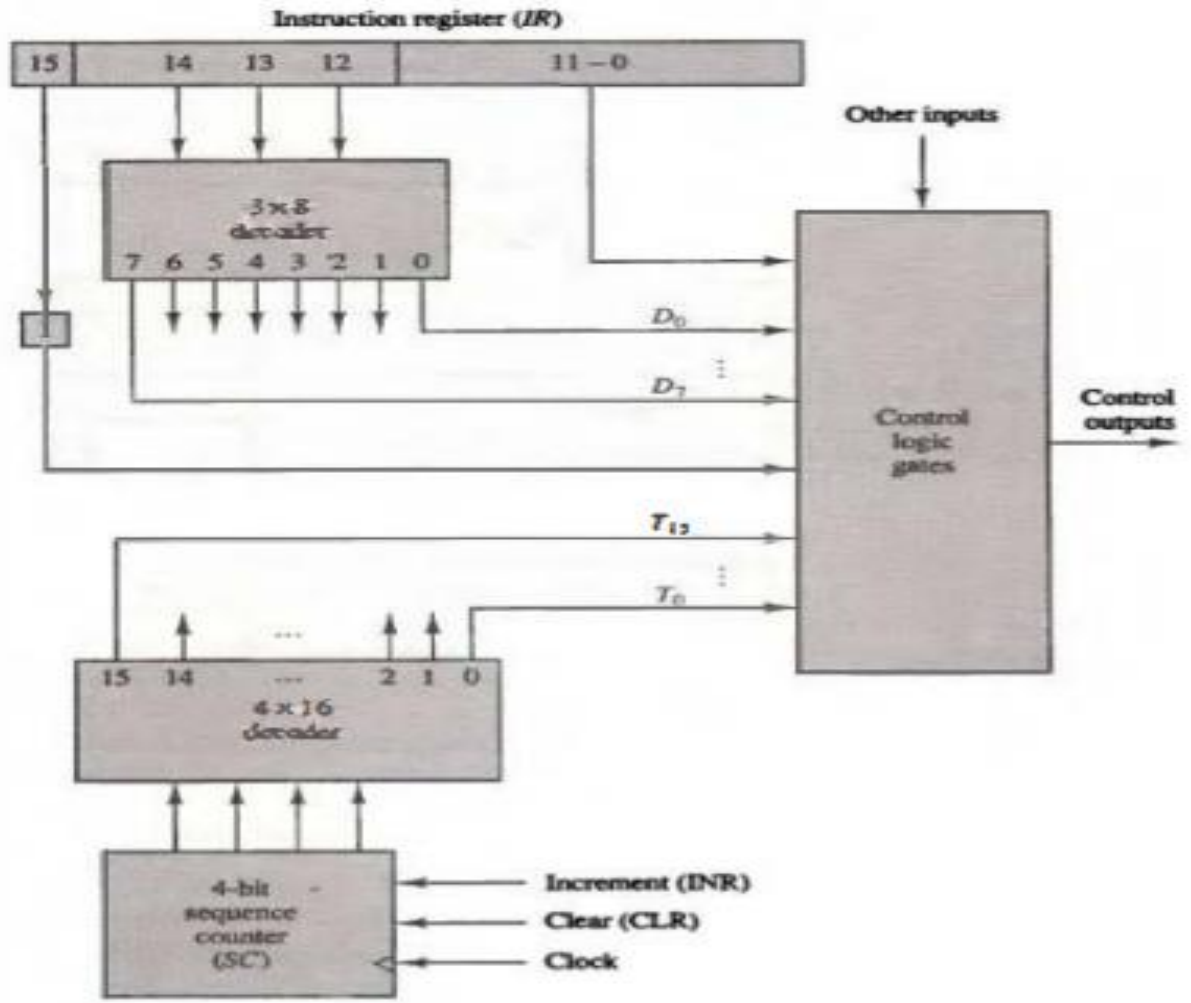


Figure: Control unit of Basic Computer

ALU Design

Control Timing Signals

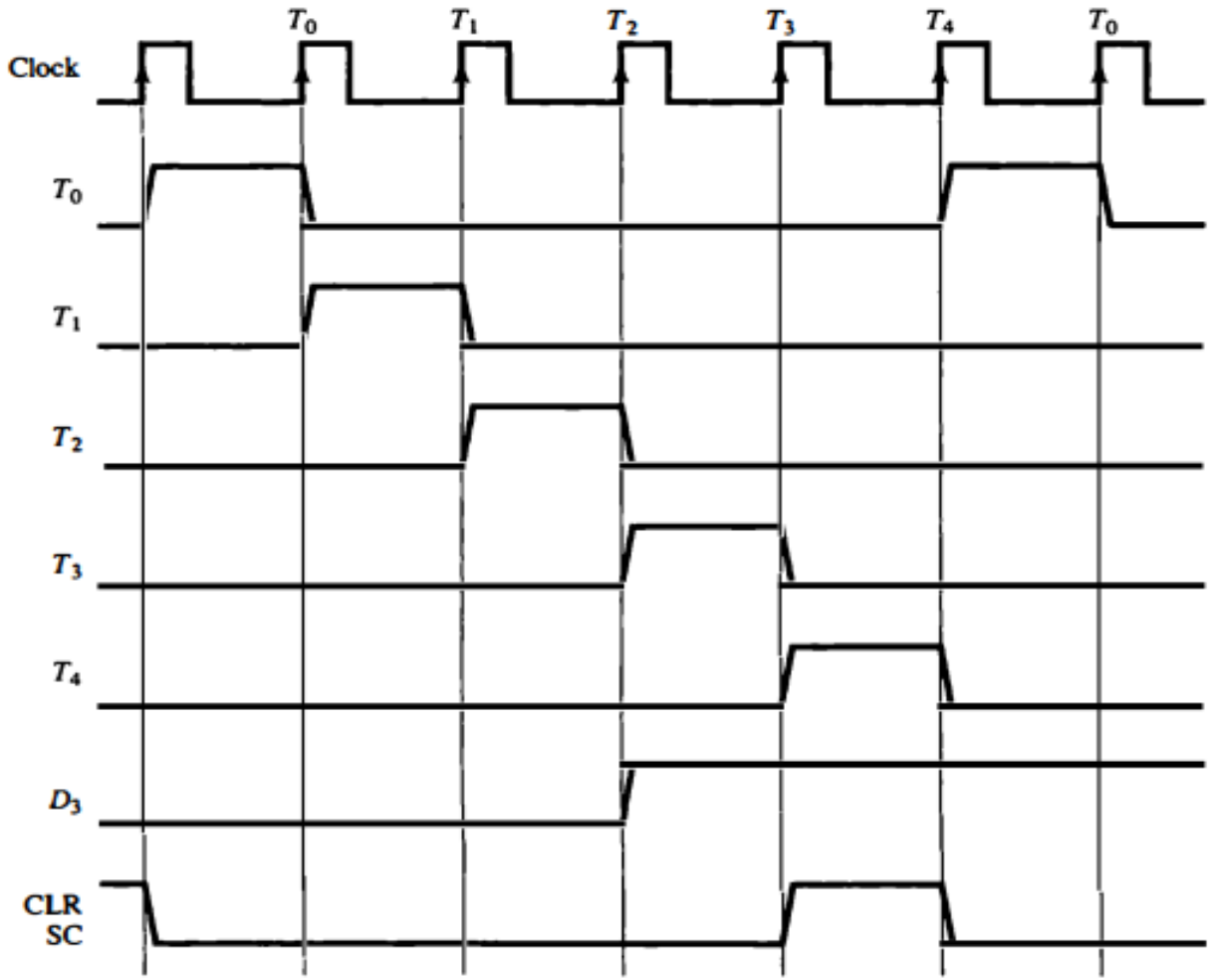


Figure: Example of control timing signals

ALU Design

Instruction Cycle

- A program consists of a sequence of instructions.
- The program is executed in the computer by going through a cycle for each instruction.
- Each instruction cycle consists of the following phases:
  1. Fetch an instruction from memory.
  2. Decode the instruction.
  3. Read the effective address from memory if the instruction has an indirect address.
  4. Execute the instruction.
- Upon the completion of step 4, the control goes back to step 1 to fetch, decode, and execute the next instruction. This process continues indefinitely unless a HALT instruction is encountered.

ALU Design

Instruction Cycle

- **Fetch and decode**
- Initially, the program counter PC is loaded with the address of the first instruction in the program. The sequence counter SC is cleared to 0, providing a decoded timing signal T_o.
- After each clock pulse, SC is incremented by one, so that the timing signals go through a sequence T_0, T_1, T_2, and so on.
- The microoperations for the fetch and decode phases can be specified by the following register transfer statements.

$$
\begin{aligned}
T_0&: \quad AR \leftarrow PC \\
T_1&: \quad IR \leftarrow M[AR], \quad PC \leftarrow PC + 1 \\
T_2&: \quad D_0, \ldots, D_7 \leftarrow \text{Decode } IR(12\text{–}14), \quad AR \leftarrow IR(0\text{–}11), \quad I \leftarrow IR(15)
\end{aligned}
$$

ALU Design

- **Instruction Fetch**
- To provide the data path for the transfer of PC to AR we apply timing signal T_0:
  1. Place the content of PC onto the bus by making the bus selection inputs $S_2S_1S_0$ equal to 010.
  2. Transfer the content of the bus to AR by enabling the LD input of AR.
- To implement T_1: $IR \leftarrow M[AR]$, $PC \leftarrow PC + 1$
  1. Enable the read input of memory.
  2. Place the content of memory onto the bus by making $S_2S_1S_0$ = 111.
  3. Transfer the content of the bus to IR by enabling the LD input of IR.
  4. Increment PC by enabling the INR input of PC.

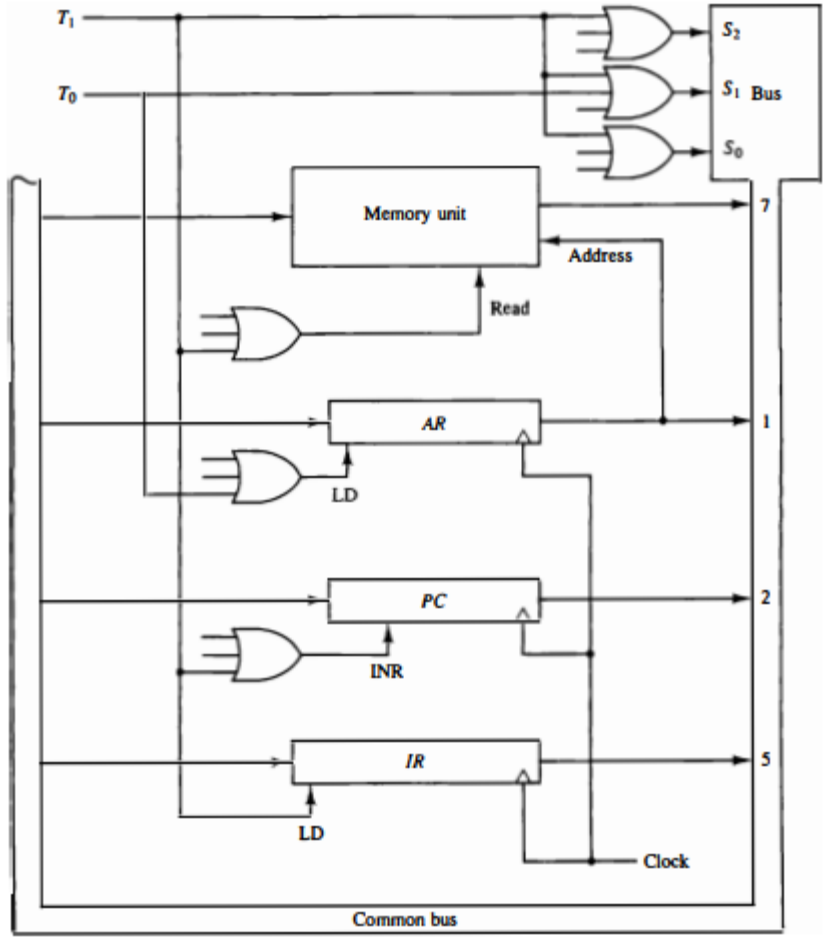


Figure: Register transfer for fetch phase.

ALU Design

- **Instruction Fetch and decode**

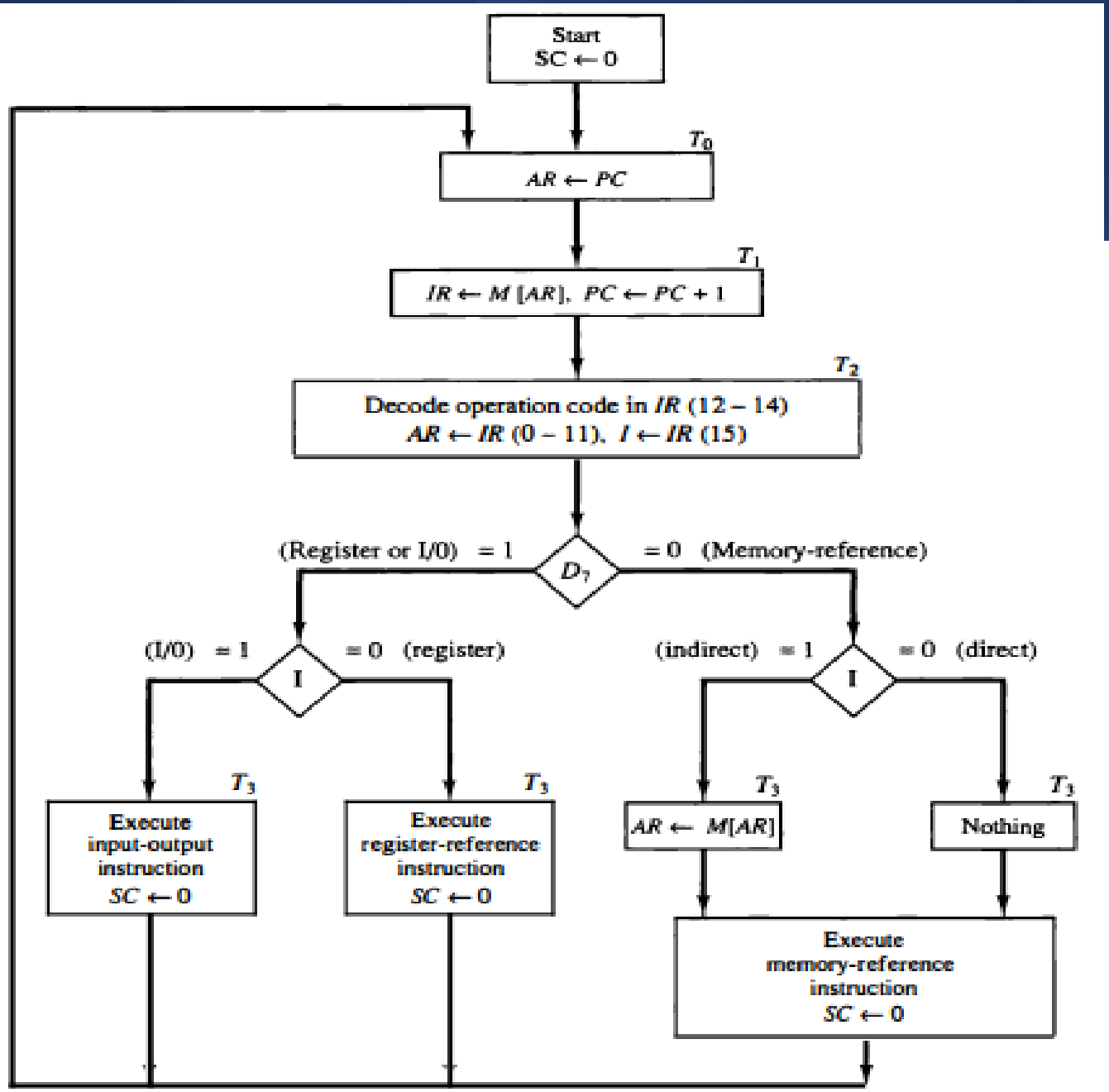


Figure: Flowchart for Instruction Cycle.

ALU Design

- **Memory-Reference Instructions**

Symbol	Operation decoder	Symbolic description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR], \quad E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC, \quad PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1,$ If $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$

ALU Design

Memory-Reference Instructions

- **AND**
- This instruction performs the AND logic operation on pairs of bits in AC and the memory word specified by the effective address. The result of the operation is transferred to AC.
- The microoperations that execute this instruction are:

$$D_0T_4: \quad DR \leftarrow M[AR]$$
$$D_0T_5: \quad AC \leftarrow AC \wedge DR, \quad SC \leftarrow 0$$

- The control function for this instruction uses the operation decoder D_0.
- Two timing signals are needed to execute the instruction:
  - The clock transition associated with timing signal T4 transfers the operand from memory into DR.
  - The clock transition associated with the next timing signal T5 transfers to AC the result of the AND logic operation between the contents of DR and AC. The same clock transition clears SC to 0, transferring control to timing signal T_0 to start a new instruction cycle.

ALU Design

Memory-Reference Instructions

- **ADD**
- This instruction adds the content of the memory word specified by the effective address to the value of AC.
- The microoperations needed to execute this instruction are:

$$D_1T_4: \quad DR \leftarrow M[AR]$$
$$D_1T_5: \quad AC \leftarrow AC + DR, \quad E \leftarrow C_{out}, \quad SC \leftarrow 0$$

- The control function for this instruction uses the operation decoder D_1.
- The sum is transferred into AC and the output carry C_{out} is transferred to the E (extended accumulator) flip-flop.

ALU Design

Memory-Reference Instructions

- **LDA: Load to AC**
- This instruction transfers the memory word specified by the effective address to AC.
- The microoperations needed to execute this instruction are:

$$D_2T_4: \quad DR \leftarrow M[AR]$$
$$D_2T_5: \quad AC \leftarrow DR, \quad SC \leftarrow 0$$

ALU Design

Memory-Reference Instructions

- **STA: Store from AC**
- This instruction stores the content of AC into the memory word specified by the effective address.
- The microoperation needed to execute this instruction is:

$$D_3T_4: \quad M[AR] \leftarrow AC, \quad SC \leftarrow 0$$

ALU Design

Memory-Reference Instructions

- **BUN: Branch Unconditionally**
- This instruction transfers the control to the instruction specified by the effective address.
- PC is incremented at time T_1 to prepare it for the address of the next instruction in the program sequence.
- The BUN instruction allows the programmer to specify an instruction out of sequence.
- The microoperation needed to execute this instruction is:

$$D_4T_4: \quad PC \leftarrow AR, \quad SC \leftarrow 0$$

ALU Design

Memory-Reference Instructions

- **BSA: Branch and Save Return Address**
- This instruction is useful for branching to a portion of the program called a subroutine or procedure.
- When executed -
  - The BSA instruction stores the address of the next instruction in sequence into a memory location specified by the effective address.
  - The effective address plus one is then transferred to PC to serve as the address of the first instruction in the subroutine.
- BSA instruction is executed with a sequence of two microoperations:

$$D_5T_4: \quad M[AR] \leftarrow PC, \quad AR \leftarrow AR + 1$$
$$D_5T_5: \quad PC \leftarrow AR, \quad SC \leftarrow 0$$

ALU Design

Memory-Reference Instructions

- **BSA**

D_5T_4: $M[AR] \leftarrow PC, \quad AR \leftarrow AR + 1$
D_5T_5: $PC \leftarrow AR, \quad SC \leftarrow 0$

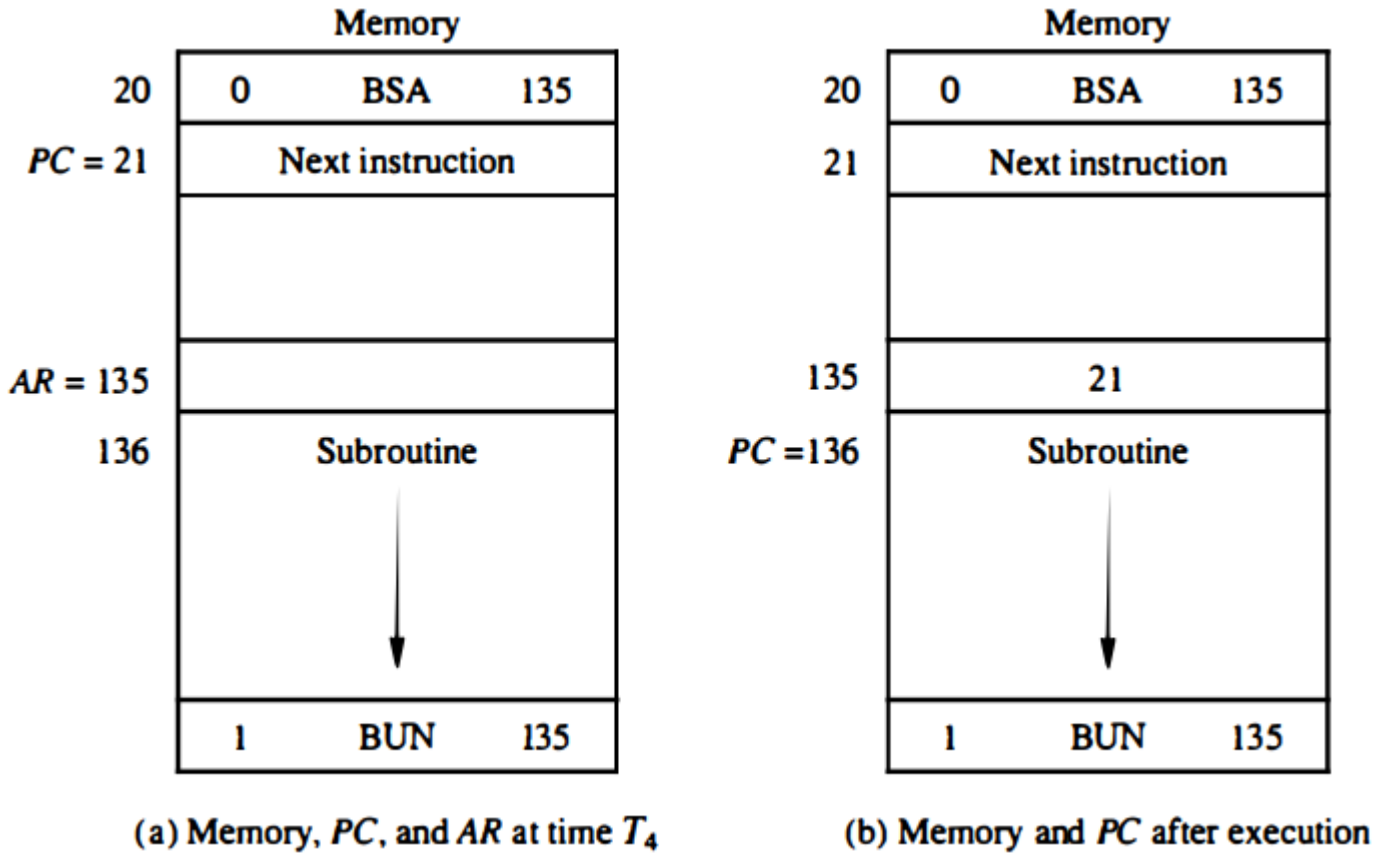


Figure: Example of BSA instruction execution.

ALU Design

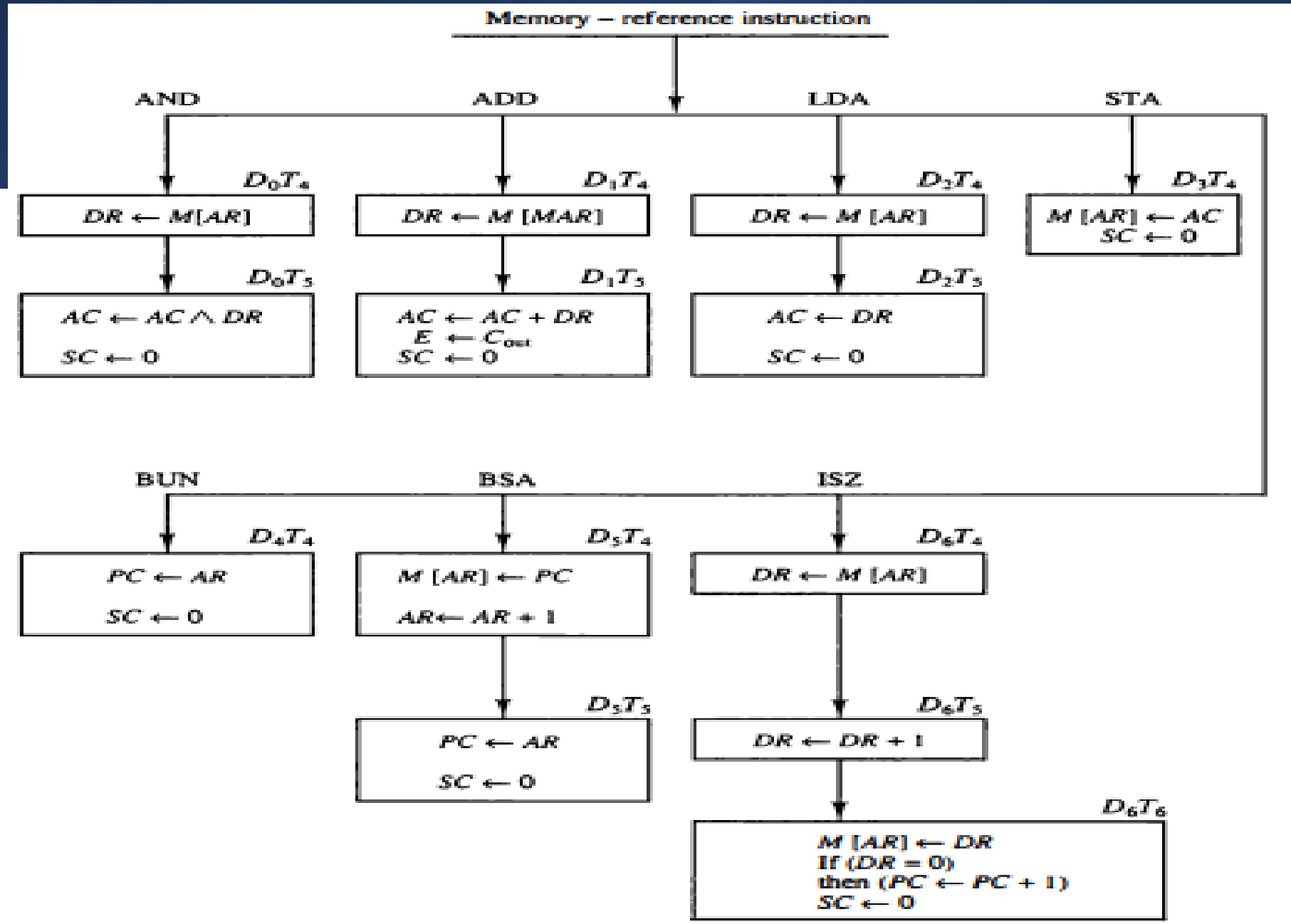
Memory-Reference Instructions

- **ISZ: Increment and Skip if Zero**
- This instruction increments the word specified by the effective address, and if the incremented value is equal to 0, PC is incremented by 1.
- This is done with the following sequence of microoperations:

$$D_6T_4: \quad DR \leftarrow M[AR]$$
$$D_6T_5: \quad DR \leftarrow DR + 1$$
$$D_6T_6: \quad M[AR] \leftarrow DR, \quad \text{if } (DR = 0) \text{ then } (PC \leftarrow PC + 1), \quad SC \leftarrow 0$$

ALU Design

Memory-Reference Instructions



ALU Design

Addressing Modes

- The operation field of an instruction specifies the operation to be performed. This operation must be executed on some data stored in computer registers or memory words.
- The way the operands are chosen during program execution is dependent on the addressing mode of the instruction.
- The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.

ALU Design

Addressing Modes

- Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:
  1. To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
  2. To reduce the number of bits in the addressing field of the instruction.
- Instructions may be defined with a variety of addressing modes.
- Addressing modes gives the flexibility for writing programs that are more efficient with respect to the number of instructions and execution time.

ALU Design

Addressing Modes

- Modes without an address field: Implied Mode and Immediate Mode.
- **Implied Mode:**
- Operands are specified implicitly in the definition of the instruction.
- For example, the instruction "complement accumulator" is an implied-mode instruction.
- All register reference instructions that use an accumulator are implied-mode instructions.
- Zero-address instructions in a stack-organized computer are implied-mode instructions since the operands are implied to be on top of the stack.

ALU Design

Addressing Modes

- Modes without an address field: Implied Mode and Immediate Mode.
- **Immediate Mode:**
- An immediate-mode instruction has an operand field rather than an address field.
- The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction.

ALU Design

Addressing Modes

- **Register Mode:**
- The operands are in registers that reside within the CPU.
- The particular register is selected from a register field in the instruction.
- A k-bit field can specify any one of 2^k registers.

ALU Design

Addressing Modes

- **Register Indirect Mode:**
- The instruction specifies a register in the CPU whose contents give the address of the operand in memory.
- Before using a register indirect mode instruction, place the memory address of the operand in the processor register.
- **Advantage:** the address field of the instruction uses fewer bits to select a register than to specify a memory address directly.

ALU Design

Addressing Modes

- **Autoincrement or Autodecrement Mode:**
- Similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory.
- When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table.

ALU Design

Addressing Modes

- **Direct Address Mode:**
- The effective address is equal to the address part of the instruction.
- The operand resides in memory and its address is given directly by the address field of the instruction.
- In a branch-type instruction the address field specifies the actual branch address.

ALU Design

Addressing Modes

- **Indirect Address Mode:**
- The address field of the instruction gives the address where the effective address is stored in memory.
- Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.

ALU Design

Addressing Modes

- **Relative Address Mode:**
- The content of the program counter is added to the address part of the instruction to obtain the effective address.
- For example, assume that the program counter contains the number 825, and the address part of the instruction contains the number 24.
- The instruction at location 825 is read from memory during the fetch phase and the program counter is then incremented by one to 826.
- The effective address computation for the relative address mode is 826 + 24 = 850.
- This is 24 memory locations forward from the address of the next instruction.

ALU Design

Addressing Modes

- **Indexed Address Mode:**
- The content of an index register is added to the address part of the instruction to obtain the effective address.
- The address field of the instruction defines the beginning address of a data array in memory.
- Each operand in the array is stored in memory relative to the beginning address.
- The distance between the beginning address and the address of the operand is the index value stored in the index register.

ALU Design

Addressing Modes

- **Indexed Address Mode:**
- Any operand in the array can be accessed with the same instruction provided that the index register contains the correct index value.
- The index register can be incremented to facilitate access to consecutive operands.

ALU Design

Addressing Modes

- **Base Register Addressing Mode:**
- The content of a base register is added to the address part of the instruction to obtain the effective address.
- A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address.
- The base register addressing mode is used in computers to facilitate the relocation of programs in memory.

ALU Design

Addressing Modes

- **Base Register Addressing Mode:**
- When programs and data are moved from one segment of memory to another, as required in multiprogramming systems, the address values of instructions must reflect this change of position.
- With a base register, the displacement values of instructions do not have to change. Only the value of the base register requires updating to reflect the beginning of a new memory segment.

ALU Design

Addressing Modes: Example

- Instruction: LDA

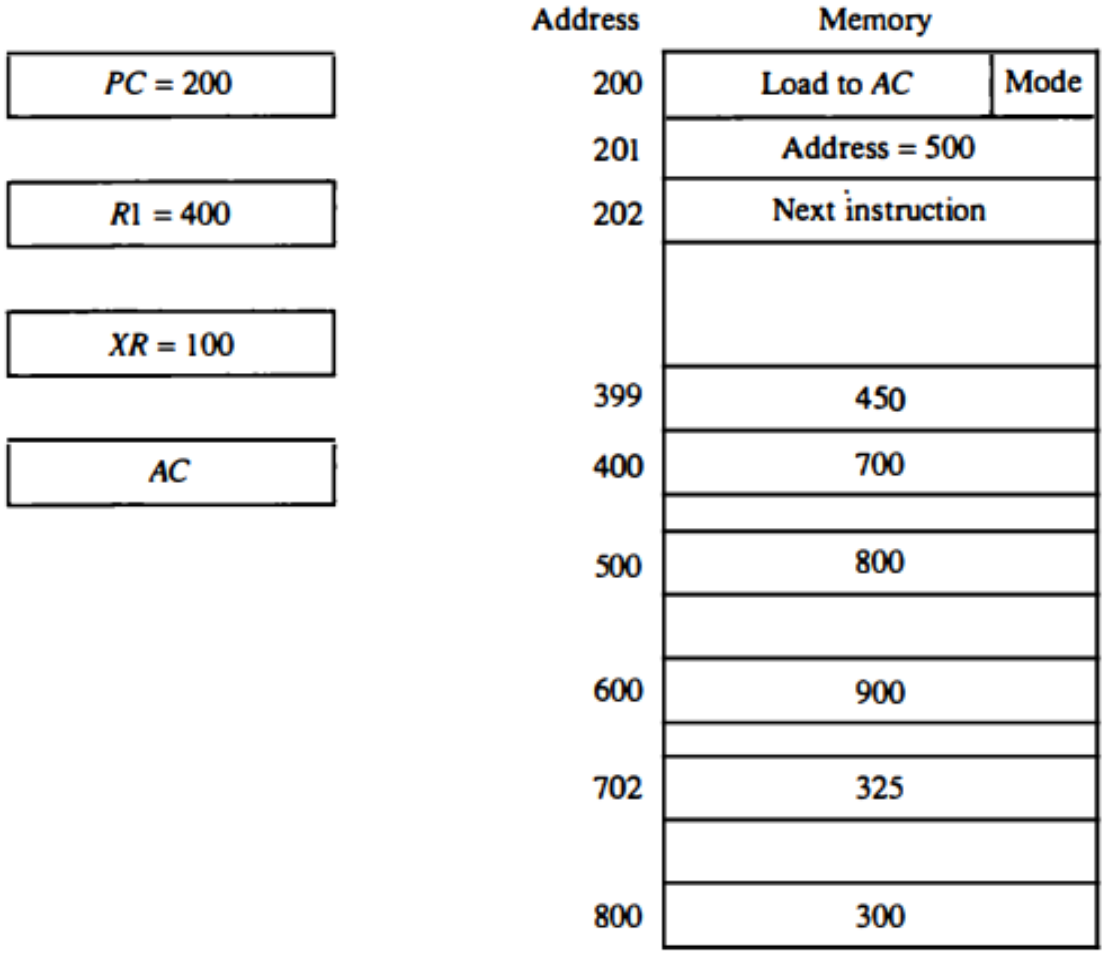


Figure: Numerical example for addressing modes.

ALU Design

Addressing Modes:

- For LDA instruction, the values of the effective address and the operand loaded into AC for the nine addressing modes.

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

Tabular list of Numerical example.

ALU Design

Program Control

- After the execution of a data transfer or data manipulation instruction, control returns to the fetch cycle with the program counter containing the address of the instruction next in sequence.
- Program control instructions specify conditions for altering the content of the program counter.
- The change in value of the program counter because of the execution of a program control instruction causes a break in the sequence of instruction execution.

ALU Design

Program Control

- Program control instructions:

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

ALU Design

Program Control

- An unconditional branch instruction causes a branch to the specified address without any conditions.
- The conditional branch instruction specifies a condition such as branch if positive or branch if zero.
  - If the condition is met, the program counter is loaded with the branch address and the next instruction is taken from this address.
  - If the condition is not met, the program counter is not changed, and the next instruction is taken from the next location in sequence.

ALU Design

Program Control

- The skip instruction does not need an address field and is therefore a zero-address instruction.
- A conditional skip instruction will skip the next instruction if the condition is met.
  - This is accomplished by incrementing the program counter during the execute phase in addition to its being incremented during the fetch phase.
- If the condition is not met, control proceeds with the next instruction in sequence.

ALU Design

Program Control

- The call and return instructions are used in conjunction with subroutines.
- The compare and test instructions do not change the program sequence directly.
- The compare instruction performs a subtraction between two operands, but the result of the operation is not retained. However, certain status bit conditions are set due to the operation.
- The test instruction performs the logical AND of two operands and updates certain status bits without retaining the result or changing the operands.
- The status bits of interest are the carry bit, the sign bit, a zero indication, and an overflow condition.

ALU Design

Program Control

- **Status Bit Conditions**
- The bits are set or cleared according to an operation performed in the ALU.
  1. 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.
  2. Bit S (sign) is set to 1 if the highest-order bit F_7 is 1. It is set to 0 if the bit is 0.
  3. Bit Z (zero) is set to 1 if the output of the ALU contains all O's. It is cleared to 0 otherwise.
  4. 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.

ALU Design

Program Control

- **Status Bit Conditions**
- The bits are set or cleared according to an operation performed in the ALU.
- Let A = 11110000 and B = 00010100. Perform A-B.

ALU Design

Program Control

- **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	$V = 0$

ALU Design

Program Control

- **Subroutine Call and Return**
- A subroutine is a self-contained sequence of instructions that performs a given computational task.
- During the execution of a program, a subroutine may be called to perform its function many times at various points in the main program.
  - Each time a subroutine is called, a branch is executed to the beginning of the subroutine to start executing its set of instructions.
  - After the subroutine has been executed, a branch is made back to the main program.

ALU Design

Program Control

- **Subroutine Call and Return**
- A subroutine call is implemented with the following microoperations:

$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PC$	Push content of PC onto the stack
$PC \leftarrow$ effective address	Transfer control to the subroutine

- If another subroutine is called by the current subroutine, the new return address is pushed into the stack, and so on.
- The instruction that returns from the last subroutine is implemented by the microoperations:

$PC \leftarrow M[SP]$	Pop stack and transfer to PC
$SP \leftarrow SP + 1$	Increment stack pointer

ALU Design

Program Interrupt

- It refers to the transfer of 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.
- Similar to subroutine call except for three variations:

  i. The interrupt is usually initiated by an internal or external signal rather than from the execution of an instruction (except for software interrupt).
  ii. The address of the interrupt service program is determined by the hardware rather than from the address field of an instruction; and
  iii. An interrupt procedure usually stores all the information necessary to define the state of the CPU (i.e. content of PC, content of all process registers, and content of certain status conditions).

ALU Design

Program Interrupt

- The CPU responds to an interrupt at the end execution of current instruction.
- Just before going to the next fetch phase, control checks for any interrupt signals.
  - If an interrupt is pending, control goes to a hardware interrupt cycle. During this cycle, the contents of PC and PSW are pushed onto the stack.

    *PSW (Program Status Word): the collection of all status bit conditions in the CPU.
  - The branch address for the particular interrupt is then transferred to PC and a new PSW is loaded into the status register.

ALU Design

Program Interrupt

- Restoring CPU state and continuing the execution of the original program:
  - The last instruction in the service program is a return from interrupt instruction. When this instruction is executed, the stack is popped to retrieve the old PSW and the return address.
  - The PSW is transferred to the status register and the return address to the program counter.

ALU Design

Types of Interrupt

- Three major types of interrupts that cause a break in the normal execution of a program:
  i. External interrupts
  ii. Internal interrupts
  iii. Software interrupts

ALU Design

Types of Interrupt

- **External interrupts** – are generated by the external sources.
- Examples that cause external interrupts –
    i. I/O device requesting transfer of data,
    ii. I/O device finished transfer of data,
    iii. elapsed time of an event, or
    iv. power failure.

ALU Design

Types of Interrupt

- **Internal Interrupts (or traps):**
  - Arise form illegal or erroneous use of an instruction or data.
  - Examples: register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation.
- If the program is rerun, the internal interrupts will occur in the same place each time.
- External interrupts depend on external conditions that are independent of the program being executed at the time.

ALU Design

Types of Interrupt

- **Software Interrupts:**
  - Software interrupt is an instruction defined in the instruction set of the processor.
  - Examples: Supervisor call instruction.
- This instruction provides means for switching from a CPU user mode to the supervisor mode.
- A program written by a user must run in the user mode. When an input or output transfer is required, the supervisor mode is requested by means of a supervisor call instruction.
- This instruction causes a software interrupt that stores the old CPU state and brings in a new PSW that belongs to the supervisor mode.

ALU Design

Practice Question:

- A certain microprocessor support 4 interrupts I_1, I_2, I_3, and I_4. I_1 has the highest priority and I_4 has the least priority. The CPU responds to the interrupt in every 2.5 microseconds. The service time of the interrupts are 20 microseconds, 40 microseconds, 30 microseconds, and 25 microseconds, respectively. The interrupts may or may not occur simultaneously. What is the possible range of time required to execute the interrupt I_4.

ALU Design

Full Adder (FA):

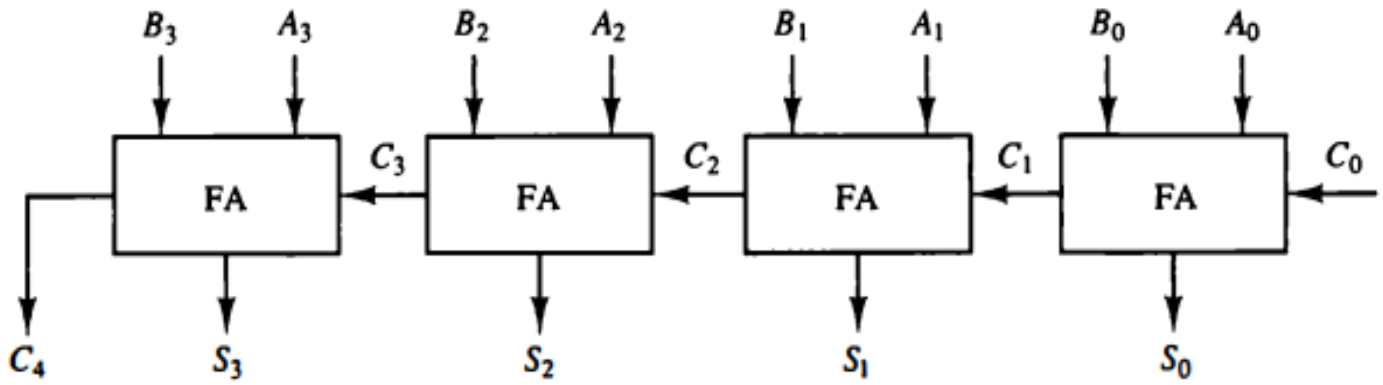


Figure: 4-bit Binary Adder.

- Truth table:

A	B	C	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$Sum = A \oplus B \oplus C$

$Carry = AB + BC + AC$

ALU Design

Adder-Subtractor:

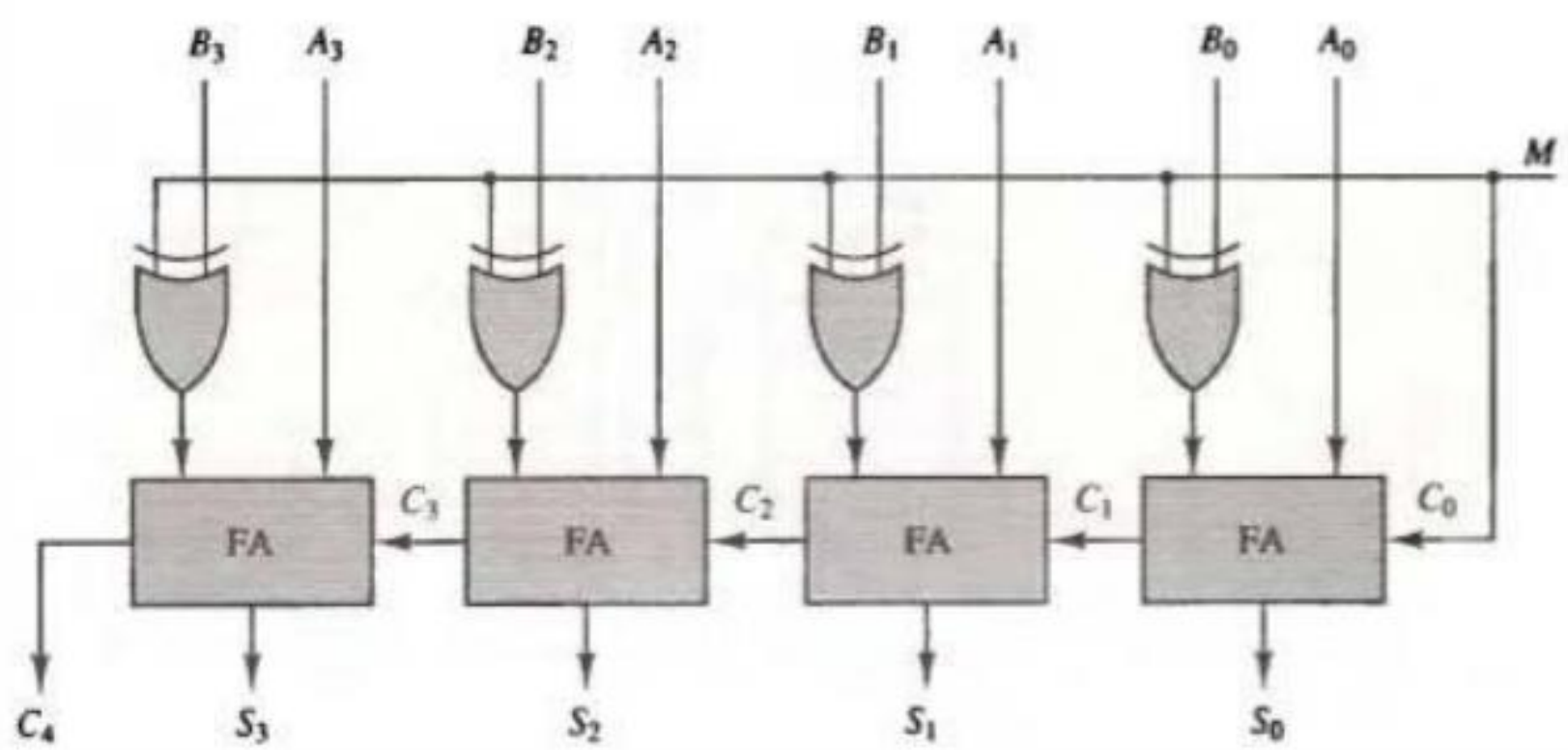


Figure: 4-bit adder-subtractor.

ALU Design

Adder-Subtractor:

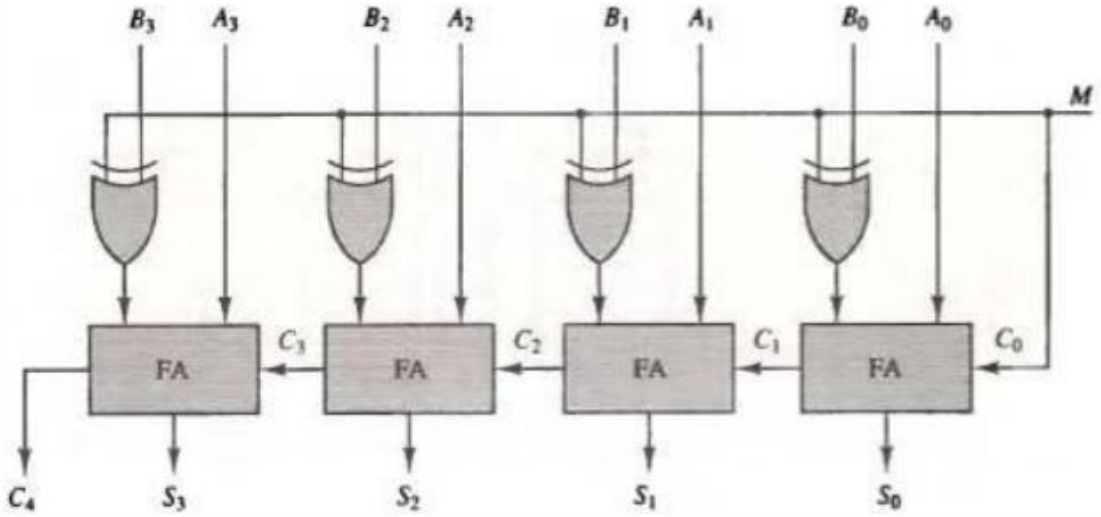


Figure: 4-bit adder-subtractor.

- When M = 0 the circuit is an adder and when M = 1 the circuit becomes a subtractor.
- For unsigned numbers, this gives A - B if A >= B or the 2's complement of (B - A) if A < B.
- For signed numbers, the result is A - B provided that there is no overflow.