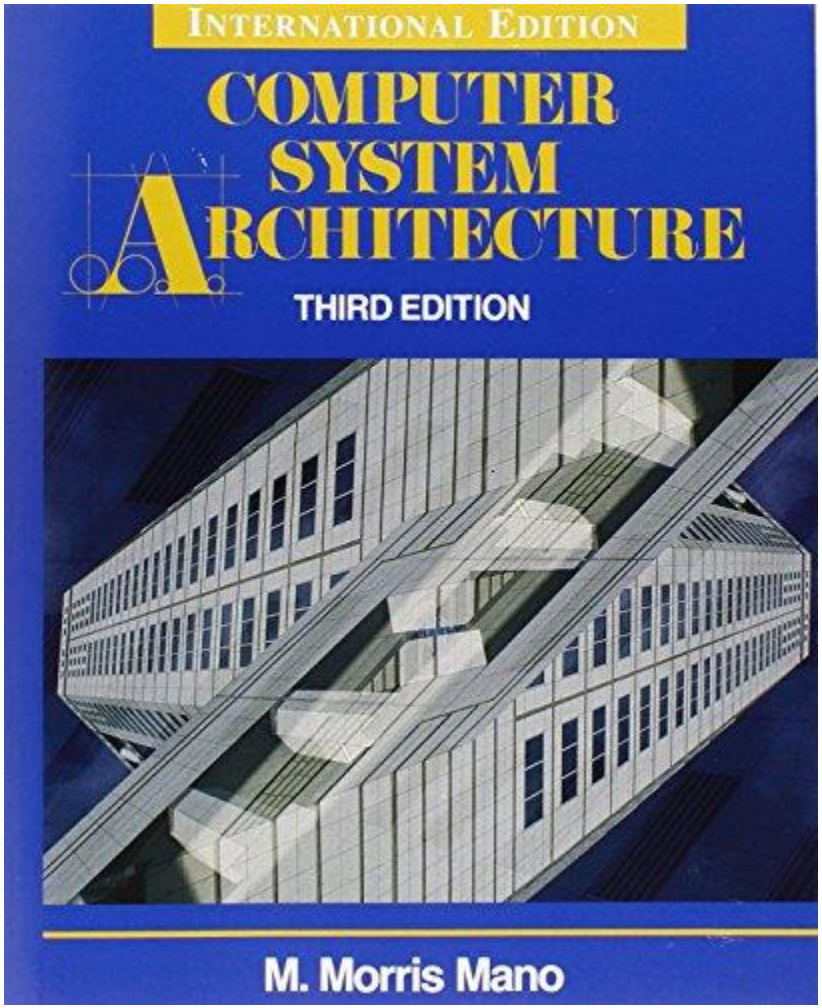
UNIT - I

CS304PC Computer Organization and Architecture

Shafakhatullah Khan Mohammed

Computer System Architecture

3rd edition (International Edition)



Unit # 1

Ch#1 Digital Computers

Ch#4 Register Transfer Language and Microoperations

Ch#5 Basic Computer Organization and Design

Chapter – I

- **Title: Digital Computers**
- **Topics:**
  - Introduction to Digital Computers
  - Block Diagram of Digital Computer
  - Definition of Computer Organization
  - Definition of Computer Design
  - Definition of Computer Architecture

Digital Computer - Introduction

- **Definition**: A digital computer is a digital system that performs various computational tasks.
- **Digital Representation**: Information is represented by variables with limited discrete values.
- **Discrete Values:** Decimal digits 0, 1, 2, ..., 9 provide 10 discrete values. (Base 10 System)

Generation	Time	Principal Technology	Examples
Zeroth	Late 1800's to 1940	Electro Mechanical Punch Cards	Tabulating and Sorting Machines
First	1940 to 1956	Vacuum Tubes	ENIAC UNIVAC I
Second	1956 to 1963	Transistors	Mainframes
Third	Late 1960's- 1970's	Integrated circuit	Mainframes Mini computers
Fourth	1971 to Present	Microprocessors	Mainframes Mini-computers Micro-computers
Fifth	Present and Beyond	Artificial Intelligence	In development

History of Computer (Generation of Computers)

Digital Computer - Introduction

- **Binary and Binary Number System**
- **Reliability**: Digital computers function more reliably with two states (binary – Base 2 System).
- **Human Logic**: Tends to be binary (true/false, yes/no).
- **Definition**: Digital computers use the binary number system (digits: 0 and 1).
- **Bit**: A binary digit is called a bit.
- **Information Representation**: Groups of bits represent binary numbers and other symbols (e.g., decimal digits, letters).
- **Binary to Decimal Conversion**
- **Example**: Binary number 1001011 = Decimal 75.

$$1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 75$$

- **Multiple Representations**: The same bits can represent different things (e.g., letter K, control code).

Digital Computer - Introduction

- **Computer System Components**
- **Functional Entities**: Hardware and Software.
- **Hardware**: Electronic components and electromechanical devices.
- **Software**: Instructions and data for data-processing tasks.
- **Program and Data Base**
- **Program**: A sequence of instructions for the computer.
- **Data Base**: Data manipulated by the program.
- **System Software**: Collection of programs for effective computer use (includes operating system).
- **Importance of System Software**
- **Indispensable**: Compensates for differences between user needs and hardware capabilities.
- **Customer Needs**: Hardware and necessary software for effective operation.

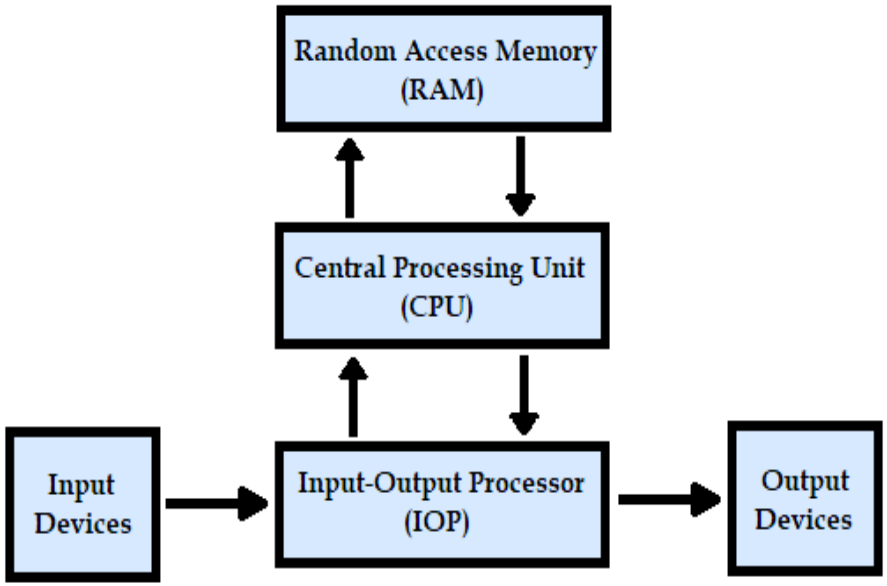
Digital Computer - Introduction

- **Application Program**: User-written programs for specific problems (e.g., high-level language programs).
- **System Program**: Programs that support the system (e.g., compilers).

Aspect	Application Programs	System Programs
Purpose	Help users perform specific tasks (e.g., writing, browsing).	Manage and operate hardware, enabling the system to function.
Examples	Word processors, games, browsers, media players.	Operating systems, device drivers, compilers, utilities.
User Interaction	Direct interaction with users to accomplish specific tasks.	Mostly run in the background, minimal direct user interaction.
Dependency	Depends on system programs to run.	Independent, forms the core for hardware management.
Installation/Uninstallation	Can be installed/uninstalled by the user.	Usually comes pre-installed, essential for system operation.
Focus	Solves user-specific problems or provides services.	Ensures system operation and hardware functionality.

Digital Computer - Block Diagram

- **Major Hardware Components**
- **Central Processing Unit (CPU)**:
  - Arithmetic and Logic Unit (ALU)
  - Registers for data storage
  - Control circuits for instruction execution
- **Memory**: Random Access Memory (RAM) for instructions and data.
- **Input/Output Processor (IOP)**:
  - Circuits for communication and information transfer.



Block Diagram of Digital Computer

Digital Computer - Computer H/W

- **Key Concepts in Computer Hardware**
- **i) Computer Organization**, ii) **Computer Design**, iii) **Computer Architecture**
- **Computer Organization**
- **Definition**: Focuses on how hardware components operate and connect.
- **Objective**: Investigate the organizational structure to ensure components function as intended.
- **Assumption**: Components are already in place.

Digital Computer - Computer H/W

- **Computer Design (Computer Implementation)**
- **Definition**: Concerned with the hardware design of the computer.
- **Process**: Formulate computer specifications.
    - Develop hardware for the system.
- **Focus**: Determine what hardware to use and how to connect parts.
- **Computer Architecture**
- **Definition**: Structure and behavior of the computer as perceived by the user.
- **Includes**:
    - a) Information formats, b) Instruction set c) Memory addressing techniques
- **Objective**: Specify functional modules (e.g., processors, memories) and their integration into a computer system.

Chapter - IV

- **Title: Register Transfer and Microoperations**
- **Topics:**
  - Register Transfer Language
  - Register Transfer
  - Bus and Memory Transfer
  - Arithmetic Microoperations
  - Logic Microoperations
  - Shift Microoperations
  - Arithmetic Logic Shift Unit.

Register Transfer Language

- **Digital Systems**: A digital system is an interconnection of digital hardware modules that accomplish specific information-processing tasks.
- **Variety**: Ranges from simple integrated circuits to complex interconnected digital computers.
- **Design Approach**: Modular design is used, with modules constructed from digital components.
- **Key Components of Digital Systems:**
  - Registers
  - Decoders
  - Arithmetic Elements
  - Control Logic
- **Interconnection**: Modules are interconnected with common data and control paths to form a digital computer system.

Register Transfer Language

- **Digital modules**: Digital modules are defined by the registers they contain and the operations performed on the data.
- **Microoperations**:
  - Defined as elementary operations on data stored in registers.
  - Examples include:
    - a) Shift b) Count c) Clear d) Load
- **Examples of Microoperations**
  - **Counter with Parallel Load**:
    - Capable of performing:
      - Increment and Load
  - **Bidirectional Shift Register**:
    - Capable of performing:
      - Shift Right and Shift Left

Register Transfer Language

- **Internal Hardware Organization**
  - Set of registers and their functions.
  - Sequence of microoperations on binary information in registers.
  - Control that initiates the sequence of microoperations.
- **Register Transfer Language (RTL)**: A symbolic notation to describe microoperation transfers among registers.
- **Functionality**:
  - Implies availability of hardware logic circuits for microoperations.
  - Facilitates the design process of digital systems.
- **RTL vs Other Languages**
- **Programming Language**: A procedure for writing symbols to specify computational processes.
- **Natural Language**: A system for writing symbols for communication.
- **Register Transfer Language**: A system for expressing microoperation sequences in digital modules.

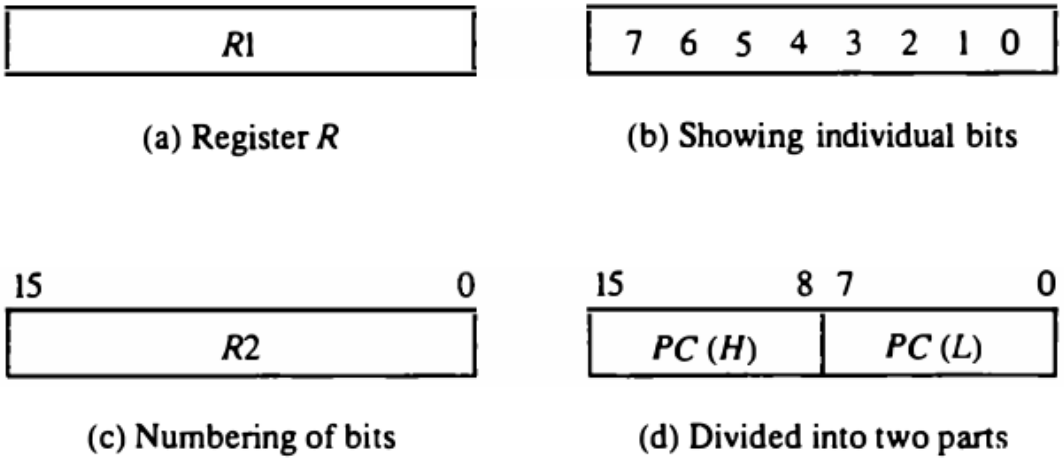
Register Transfer

- **Computer Register**: Computer registers are designated by capital letters (sometimes followed by numerals) to denote their function.
- **Examples**:
  - **MAR**: Memory Address Register
  - **PC**: Program Counter
  - **IR**: Instruction Register
  - **R1**: Processor Register
- **Structure of Registers**
- **Bit Numbering**:
  - Individual flip-flops in an n-bit register are numbered from 0 to n-1.
  - Numbering starts from the rightmost position (0) and increases to the left.
- **Representation**:
  - Commonly represented as a rectangular box with the register name inside.

Register Transfer

- **Block Diagram of Registers**

Figure 4-1 Block diagram of register.



- **Figure 4-1**:
  - (a) Basic representation of a register.
  - (b) Individual bits distinguished.
  - (c) Bit numbering in a 16-bit register.
  - (d) Partitioning of a 16-bit register into low (L) and high (H) bytes.

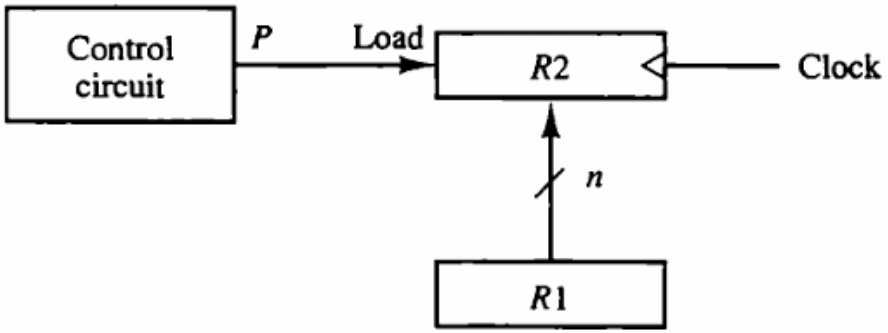
Register Transfer

- **Register Transfer Notation**
- **Transfer Operation:**
  - Symbolic representation: R2 <- R1
  - Indicates transfer of content from R1 to R2.
  - Content of R1 remains unchanged after the transfer.
- **Control Conditions for Transfers**
- **Conditional Transfer:**
  - **Example:** If (P = 1) then (R2 <- R1)
  - P is a control signal generated in the control section.
- **Control Function:**
  - A Boolean variable (P) that determines if the transfer occurs.
  - Notation: P: R2 <- R1

Register Transfer

- **Hardware Implementation**
- **Hardware Construction**:
  - Every register transfer statement implies a hardware design for the transfer.

Figure 4-2 Transfer from R1 to R2 when P = 1.

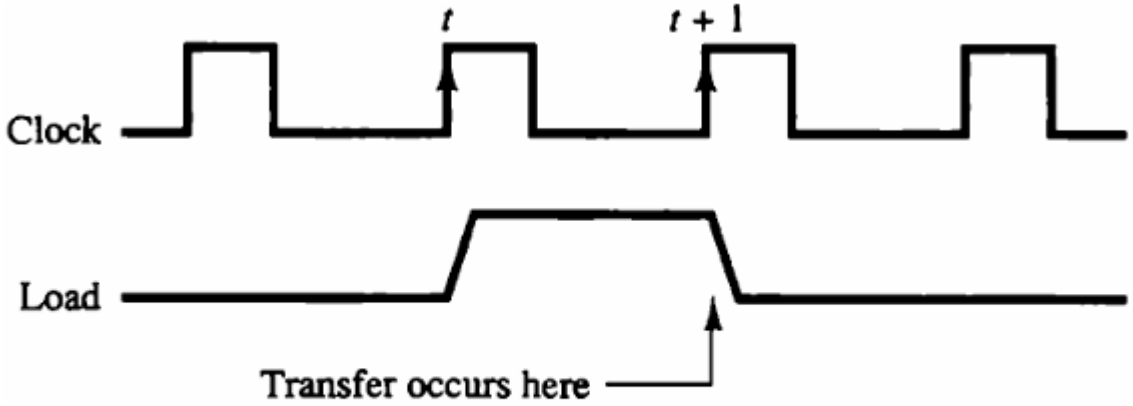


(a) Block diagram

- **Block Diagram**:
  - Shows connections between registers R1 and R2.
  - Control variable P activates the load input of R2.

Register Transfer

- **Timing and Control Signals**



(b) Timing diagram

- **Timing Diagram**:
  - P is activated by the rising edge of a clock pulse.
  - Data inputs of R2 are loaded in parallel during the next clock transition.
- **Clock Synchronization**:
  - Transfers occur during clock edge transitions.

Register Transfer

- **Basic Symbols in Register Transfer Notation**
- **Key Symbols:**
  - Registers: Denoted by capital letters (e.g., R1, R2).
  - Parentheses: Specify parts of a register (e.g., PC(0-7)).
  - Arrow: Indicates transfer direction.
  - Comma: Separates simultaneous operations (e.g., T: R2 <- R1, R1 <- R2).

TABLE 4-1 Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	*MAR*, *R2*
Parentheses ( )	Denotes a part of a register	*R2*(0–7), *R2*(*L*)
Arrow ←	Denotes transfer of information	*R2* ← *R1*
Comma ,	Separates two microoperations	*R2* ← *R1*, *R1* ← *R2*

Bus and Memory Transfer

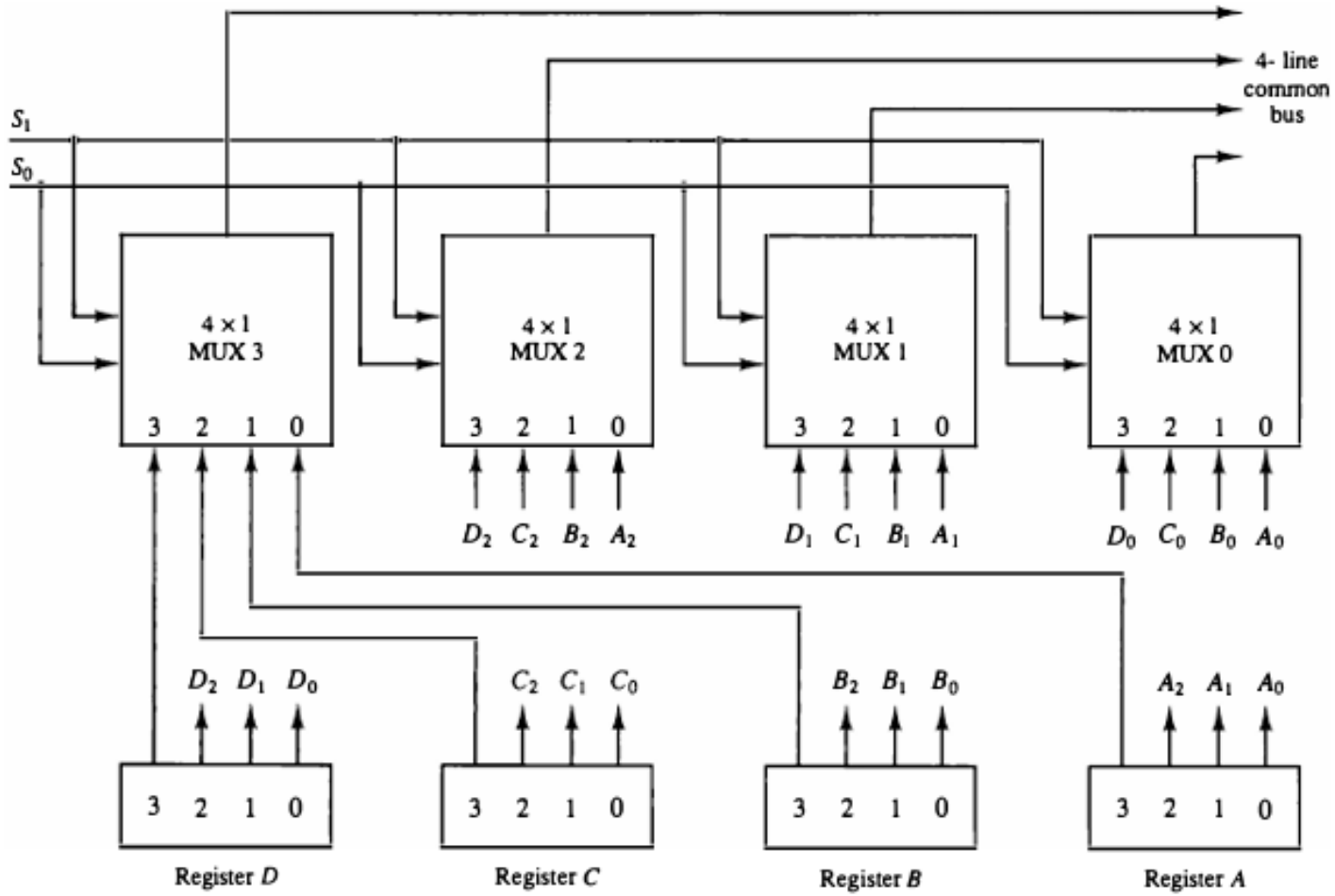
- **Registers in Digital Computers**:
  - A typical digital computer has many registers.
  - Efficient data transfer between registers is essential.
- **Challenge**:
  - Excessive wiring if separate lines are used for each register.
- **Common Bus System**:
  - A common bus system allows efficient data transfer between multiple registers.
- **Structure**:
  - Consists of a set of common lines (one for each bit) for binary information transfer.
- **Operation**:
  - Control signals determine which register is selected during data transfer.

Bus and Memory Transfer

- **Multiplexers in Bus Systems**
- **Role of Multiplexers**:
  - Select the source register whose data is placed on the bus.
- **Configuration**:
  - For four registers, a bus system can be constructed using multiplexers.
  - **Example**: Each register has four bits (numbered 0 through 3).
- **Figure 4-3**:
  - Illustrates the bus system for four registers.
- **Components**:
  - Four 4 x 1 multiplexers, each with four data inputs and two selection inputs (S_1 and S_0).
- **Connection**:
  - Outputs of registers connected to multiplexer inputs using labels (e.g., A_1).

Bus and Memory Transfer

Figure 4-3 Bus system for four registers.



Bus and Memory Transfer

- **Multiplexer Functionality**
- **Data Input Selection**:
  - MUX 0 multiplexes the 0 bits, MUX 1 the 1 bits, and so on.
- **Selection Lines**:
  - S_1 and S_0 control which register's bits are transferred to the bus.
- **Example**:
  - When S_1 S_0 = 00, the bus receives data from register A.

TABLE 4-2 Function Table for Bus of Fig. 4-3

S_1	S_0	Register selected
0	0	*A*
0	1	*B*
1	0	*C*
1	1	*D*

Bus and Memory Transfer

- **Generalization**:
  - A bus system can multiplex k registers of n bits each to produce an n-line common bus.
- **Multiplexer Requirements**
  - **Number of Multiplexers**:
    - Equal to the number of bits (n) in each register.
  - **Size of Multiplexers**:
    - Each multiplexer must be k x 1 to handle k data lines.
  - **Example**:
    - For eight registers of 16 bits, 16 multiplexers are needed.
- **Transfer Mechanism:**
  - Connect bus lines to inputs of destination registers.
  - Activate load control of the selected destination register.
- **Symbolic Representation:**
  - When bus is included: BUS <- C, R1 <- BUS, Implied bus transfer: R1 <- C

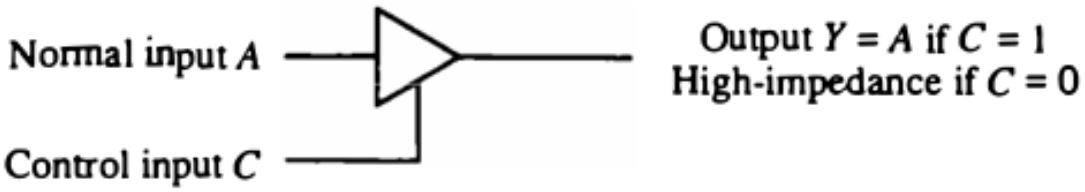
Three State Bus Buffers

- **Definition**:
  - A three-state gate is a digital circuit with three states: logic 1, logic 0, and high impedance.
- **High-Impedance State**:
  - Behaves like an open circuit, output is disconnected and has no logic significance.
- **Functionality of Three-State Gates**
- **Conventional Logic**:
  - Can perform any conventional logic function (e.g., AND, NAND).
- **Common Use**:
  - Most commonly used in bus system design is the **buffer gate**.

Three State Bus Buffers

- **Three-State Buffer Gate**
- **Graphic Symbol**:

Figure 4-4 Graphic symbols for three-state buffer.



- **Inputs**:
  - Normal input and control input.
- **Control Input Functionality**:
  - Control input = 1: Output enabled, behaves like a conventional buffer.
  - Control input = 0: Output disabled, enters high-impedance state.

Three State Bus Buffers

- **Advantages of High-Impedance State**
- **Special Feature**:
  - Allows multiple three-state gate outputs to connect to a common bus line without loading effects.
- **Bus System Construction**:
  - Demonstrated in Fig. 4-5 with outputs of four buffers connected to a single bus line.
- **Control of Three-State Buffers**
- **Active State Control**:
  - Only one buffer can be active at a time to prevent conflicts.
- **Control Mechanism**:
  - Control inputs determine which normal input communicates with the bus line.

Three State Bus Buffers

- **Using a Decoder for Control**
- **Decoder Functionality**:
  - Ensures only one control input is active at any time.
- **Operation**:
  - When the decoder enable input = 0: All outputs are 0, bus line is in the high-impedance state.
  - When enabled: One buffer is active based on select inputs.

Three State Bus Buffers

Comparison with Multiplexers

- **Multiplexer Functionality**:

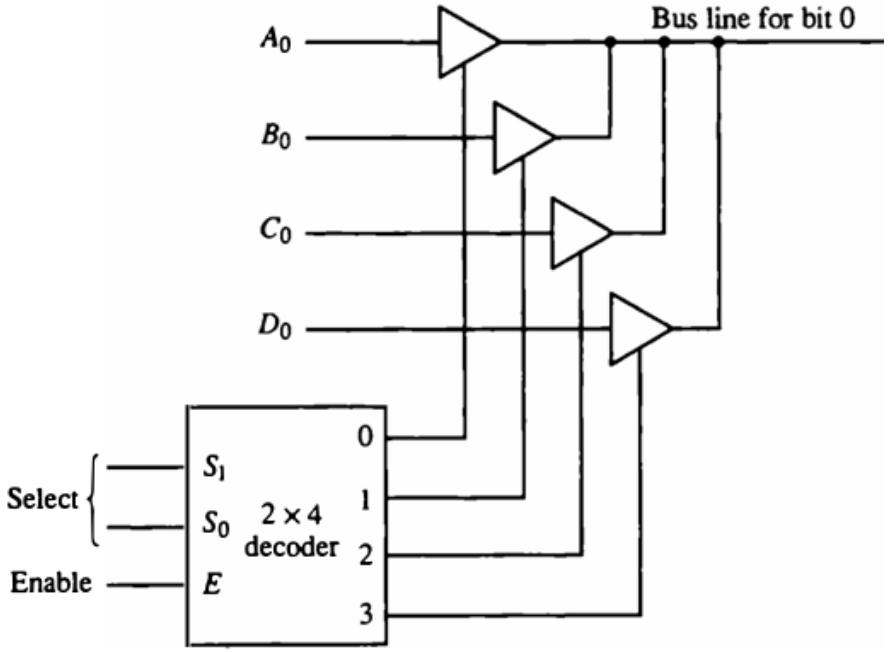


Figure 4-5 Bus line with three state-buffers.

- **Construction of Common Bus**:
  - For four registers of ***n*** bits, need ***n*** circuits with four buffers each.
- **Output Lines**:
  - Each common output produces one line for the common bus, totaling ***n*** lines.

Memory Transfer

- **Memory Operations:**
  - **Read Operation**: Transfer of information from memory to the outside environment.
  - **Write Operation**: Transfer of new information into memory.
- **Memory,** is denoted by the letter **M**.
- **Addressing**:
  - Each memory word is selected by its **memory address**.
  - Address of M is specified in square brackets: **M[address]**.
- **Registers in Memory Operations:**
- **Address Register (AR)**: Receives the address for memory access.
- **Data Register (DR)**: Holds data transferred from or to memory.

Memory Transfer

- **Read Operation Statement**:
  - **Read**: DR <- M[AR]
- **Functionality**:
  - Transfers information from the memory word M selected by the address in AR into DR.
- **Write Operation Statement**:
  - Transfers content from DR to the memory word M selected by the address in AR.
- **Example**:
  - Input data in register R1, operation: R3 <- R1 + $\overline{R2}$ + 1
  - $\overline{R2}$ represents the 1's complement of R2.
- **2's Complement Calculation**:
  - Adding 1 to the 1's complement of R2 produces the 2's complement.
  - The operation R1 - R2 can be achieved by adding R1 to the 2's complement of R2.

Microoperations Overview

- **Increment and Decrement**: Symbolized by plus-one and minus-one operations.
- **Implementation**: This can be implemented with a combinational circuit or a binary up-down counter.
- **Arithmetic Microoperations Operations**
- **Basic Microoperations**: Addition and subtraction are included in the basic set.
- **Multiplication and Division:** They are not listed in the basic set, but are valid operations implemented in digital systems using combinational circuits.
- **Implementation of Multiplication and Division**
- **Multiplication**: Typically implemented using a sequence of add and shift microoperations.
- **Division**: Implemented using a sequence of subtract and shift microoperations.

Arithmetic Microoperations

- Unlike register transfer microoperations, arithmetic microoperations change the information content.
- The basic arithmetic microoperations are:
  - Addition
  - Subtraction
  - Increment
  - Decrement
  - Shift
- The RTL statement: **R3 ← R1 + R2** indicates an add microoperation. We can similarly specify the other arithmetic microoperations.
- Multiplication and division are not considered microoperations.
  - A sequence of adds and shifts implements multiplication.
  - A sequence of subtracts and shifts implements division.

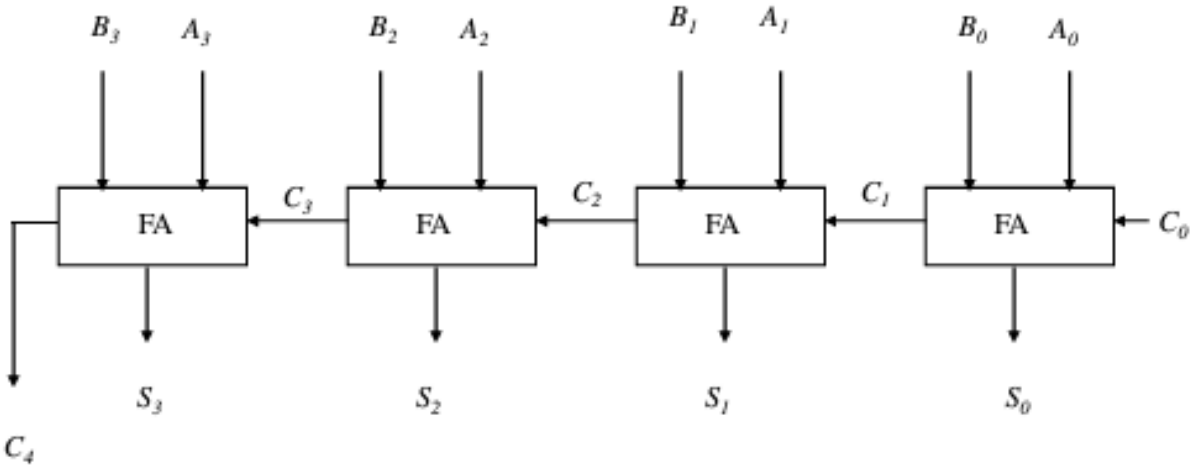
Arithmetic Microoperations

TABLE 4-3 Arithmetic Microoperations

Symbolic designation	Description
$R3 \leftarrow R1 + R2$	Contents of $R1$ plus $R2$ transferred to $R3$
$R3 \leftarrow R1 - R2$	Contents of $R1$ minus $R2$ transferred to $R3$
$R2 \leftarrow \overline{R2}$	Complement the contents of $R2$ (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement the contents of $R2$ (negate)
$R3 \leftarrow R1 + \overline{R2} + 1$	$R1$ plus the 2's complement of $R2$ (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of $R1$ by one
$R1 \leftarrow R1 - 1$	Decrement the contents of $R1$ by one

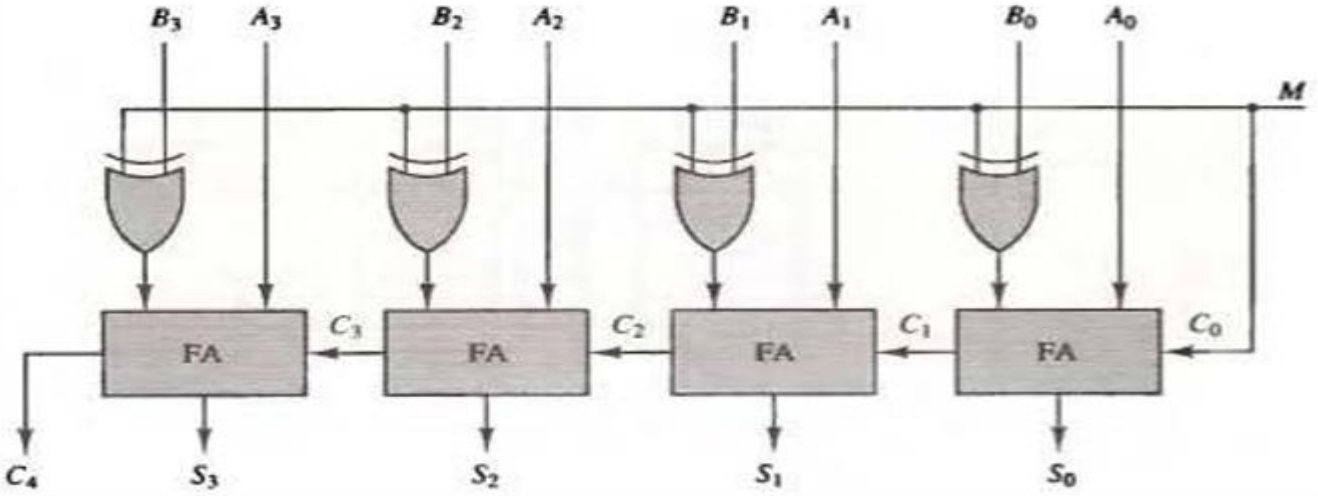
Binary Adder

- We implement a binary adder with registers to hold the data and a digital circuit to perform the addition (called a **binary adder**).
- The binary adders are constructed using full adders connected in cascade so that the carry produced by one full adder becomes an input for the next.
- Adding two *n*-bit numbers requires *n* full adders.
- The *n* data bits for **A** and **B** might come from **R1** and **R2** respectively



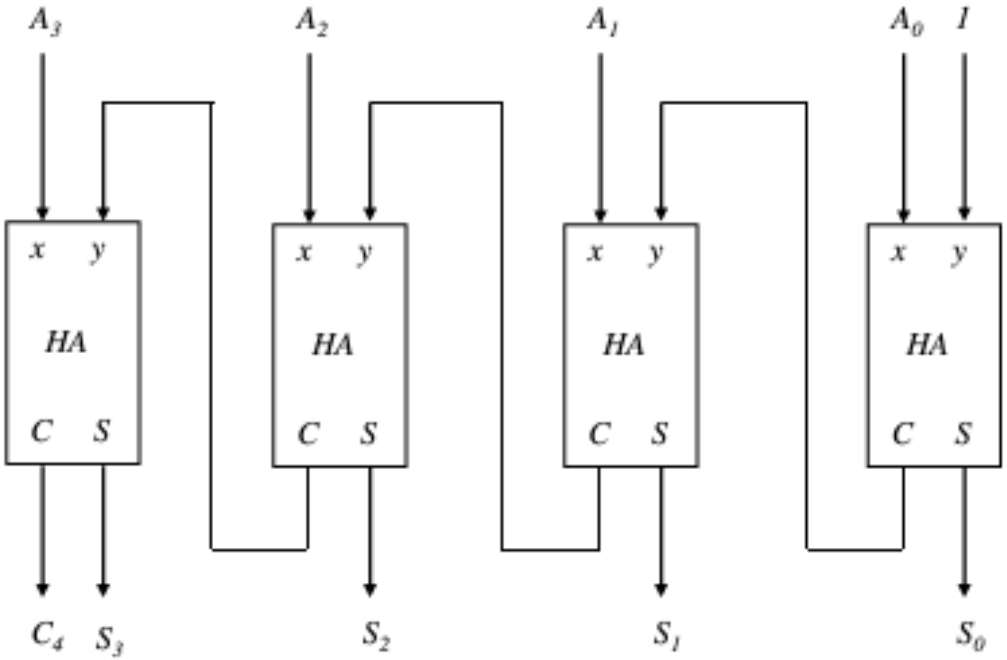
Adder - Subtracter

- Subtracting **A – B** is most easily done by adding $\overline{\boldsymbol{B}}$ to **A** and then adding 1.
- This makes it convenient to combine addition and subtraction into one circuit, called an adder-subtracter.
- M is the mode indicator
  - M = 0 indicates addition (B is left alone and C_0 is 0)
  - M = 1 indicates subtraction (B is complement and C_0 is 1).



Binary Increment

- The binary incrementer adds 1 to the contents of a register, e.g., a register storing 0101 would have 0110 in it after being incremented.
- There are times when we want incrementing done independent of a register. We can accomplish this with a series of cascading half-adders.



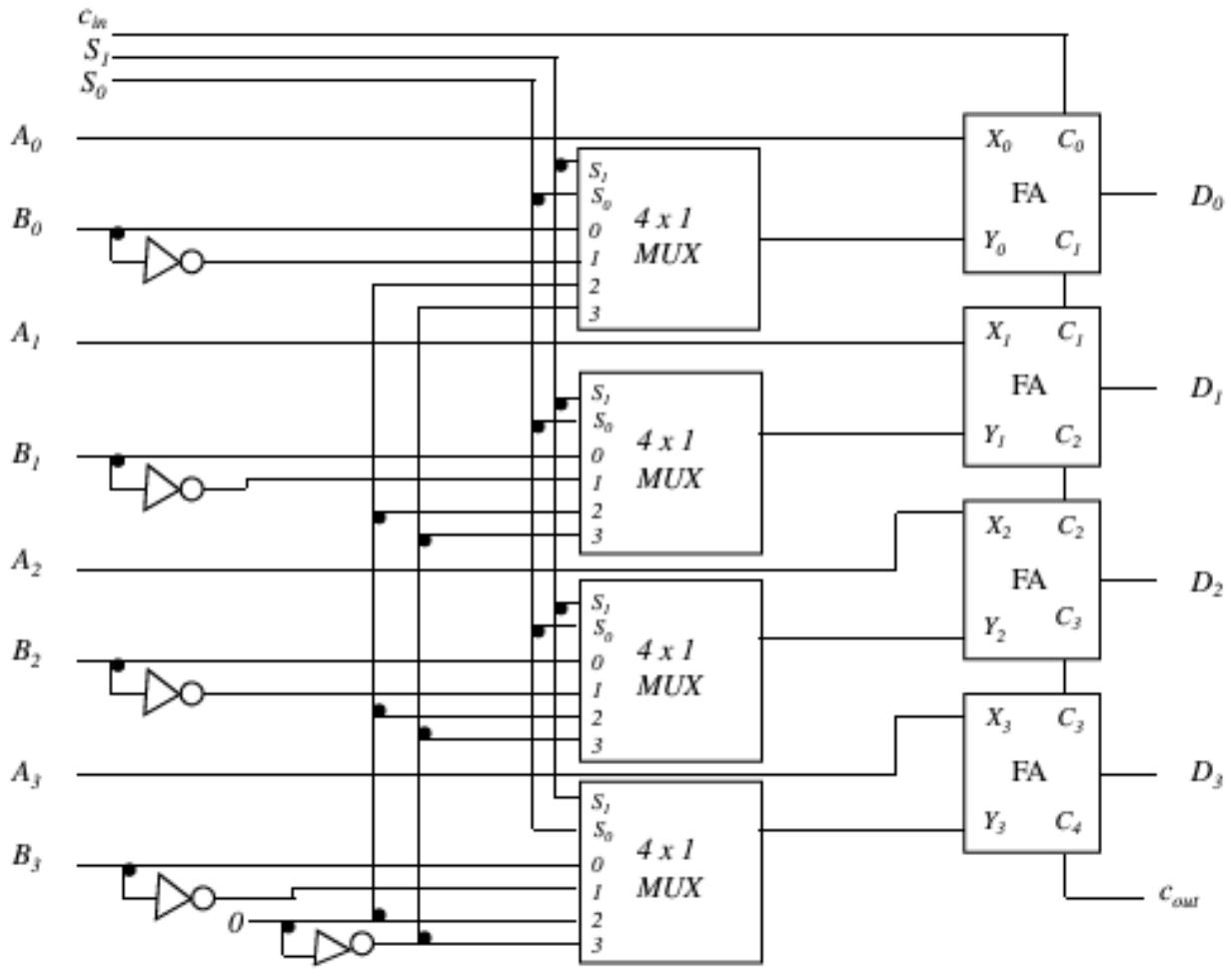
Arithmetic Circuit

- We can implement 7 arithmetic microoperations (add, add with carry, subtract, subtract with borrow, increment, decrement and transfer) with one circuit.
- We provide a series of cascading full adders with Ai and the output of a 4x1 multiplexer.
  - The multiplexer`s inputs are two selects, Bi, $\overline{Bi}$, logical 0 and logical 1.
  - Which of these four values we provide (together with the carry) determines which microoperation is performed.

Arithmetic Circuit Function Table

Select		Input	Output		
S_1	S_0	C_{in}	Y	$D = A + Y + C_{in}$	Microoperation
0	0	0	B	$D = A + B$	Add
0	0	1	B	$D = A + B + 1$	Add with Carry
0	1	0	$\overline{B}$	$D = A + \overline{B}$	Subtract with Borrow
0	1	1	$\overline{B}$	$D = A + \overline{B} + 1$	subtract
1	0	0	0	$D = A$	Transfer A
1	0	1	0	$D = A + 1$	Increment A
1	1	0	1	$D = A - 1$	Decrement A
1	1	1	1	$D = A$	Transfer A

4 bit Arithmetic Circuit



Microoperations of Arithmetic Circuit

- When S_1 S_0 = 00, the MUX provides B.
  - The result is Add (for C_{in}= 0) or Add With Carry (for C_{in} = 1).
- When S_1 S_0 = 01, the MUX provides B'.
  - The result is Subtract with Borrow (for C_{in} = 0) or Subtract (for C_{in} = 1).
- When S_1 S_0 = 10, the MUX provides 0.
  - The result is Transfer (for C_{in} = 0) or Increment (for C_{in} = 1).
- When S_1 S_0 = 11, the MUX provides 1.
  - The result is Decrement (for C_{in} = 0) or Transfer (for C_{in} = 1).

Logic Microoperations

- Logic microoperations are binary operations performed on corresponding bits of two-bit strings.
- Example: **P: R1 ←R1 ⊕R2**
  - 1010 Content of R1
  - 1100 Content of R2
  - 0110 Content of R1 after P = 1
- Special Symbols used for logic operations:
  - ∧ - AND, ∨ - OR, and ⊕ - XOR
  - This avoids confusing AND with multiplication, OR with addition, etc.

Logic Microoperations

Truth Tables for 16 2-Variable Function

x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

16 Logic Microoperations

Boolean Function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR

16 Logic Microoperations

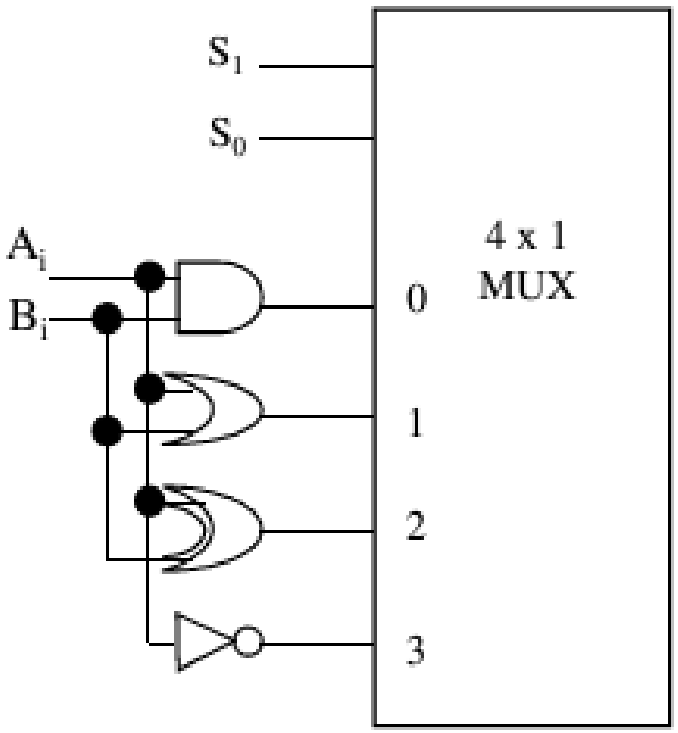
Boolean Function	Microoperation	Name
$F_8 = (x + y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \overline{B}$	Complement B
$F_{11} = x + y'$	$F \leftarrow A \vee \overline{B}$	
$F_{12} = x'$	$F \leftarrow \overline{A}$	Complement A
$F_{13} = x' + y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow$ all 1's	Set to all 1's

16 Logic Microoperations

Boolean Function	Microoperation	Name
$F_8 = (x + y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \overline{B}$	Complement B
$F_{11} = x + y'$	$F \leftarrow A \vee \overline{B}$	
$F_{12} = x'$	$F \leftarrow \overline{A}$	Complement A
$F_{13} = x' + y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow$ all 1's	Set to all 1's

One Stage of Logic Circuit

One Stage of Logic Circuit



S_1	S_0	Output	Operation
0	0	E = A ^ B	AND
0	1	E = A ∨ B	OR
1	0	E = A ⊕ B	XOR
1	1	E = A'	Complement

Logic Applications

- Logic Operations allow us to manipulate individual bits or portions of a word stored in a register.
- These applications include:
  - selective set
  - selective complement
  - select clear
  - Mask
  - insert and
  - clear

Selective set and Selective Complement

- **Selective-set** sets to 1 the bits in register A where there is a corresponding 1 in register B:
  - 1010 Content of A before
  - 1100 Content of B (logic operand)
  - 1110 Content of A after
- This is done using the logical-OR operation.
- **Selective-complement** complements the bits in register A where there is a corresponding 1 in register B:
  - 1010 Content of A before
  - 1100 Content of B (logic operand)
  - 0110 Content of A after
- This is done using the exclusive-OR operation.

Selective Clear and Mask

- **Selective-clear** clears to 0 the bits in register A where there is a corresponding 1 in register B:
  - 1010 Content of A before
  - 1100 Content of B (logic operand)
  - 0010 Content of A after
- This is done using the logical-AND operation and $\overline{B}$.
- **Mask** clears to 0 the bits in register A where there is a corresponding 0 in register B:
  - 1010 Content of A before
  - 1100 Content of B (logic operand)
  - 1000 Content of A after
- This is done using the logical-AND operation and B.

Insert

- **Insert:** inserts a new value into a set of bits in register A.
- First we mask out the upper four bits (in our 8-bit value):
  - 0110 1010 Content of A before
  - 0000 1111 Content of B (logic operand)
  - 0000 1010 Content of A after
- In the second step, we insert the new values:
  - 0000 1010 Content of A before
  - 1001 0000 Content of B (logic operand)
  - 1001 1010 Content of A after
- The masking is done using an AND and the insertion is done with an OR.

Shift Microoperations

- **Shift microoperations** are used for serial transfer of data. They are also used in conjunction with arithmetic and logic operations.
- The register contents can be shifted to the left or the right.
- There are three types of shift operations:
  - Logical shifts transfer 0 through the serial input, with all the bits involved in the shifting.
  - Arithmetic shifts multiplies (or divides) a signed number by 2.
  - Circular shifts circulates the bits of the register around the two ends with no loss of information.

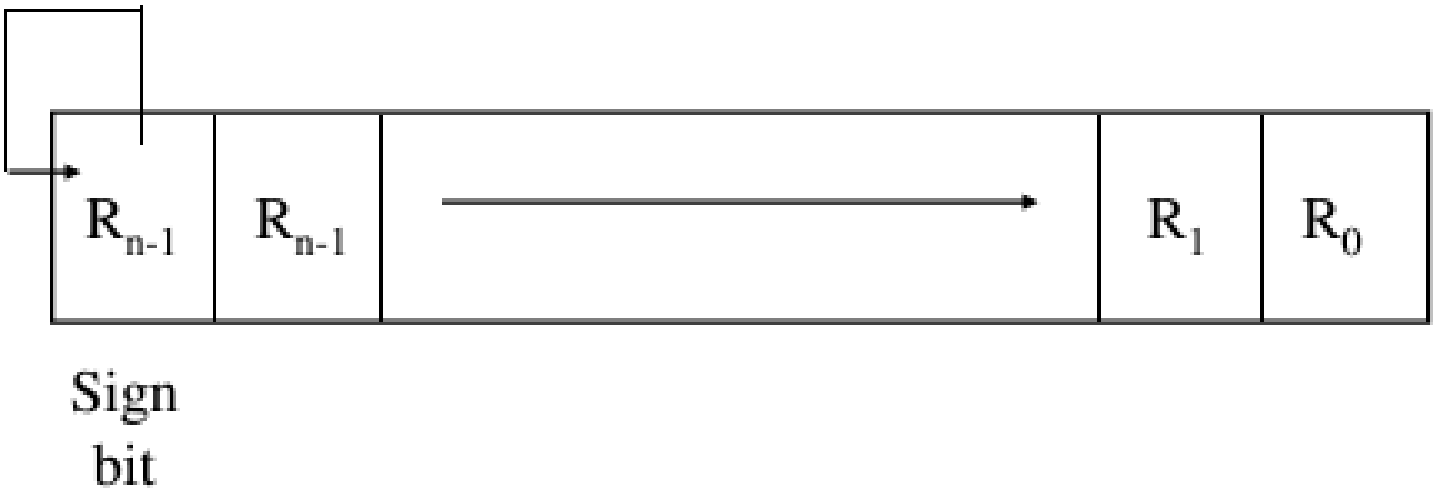
Shift Microoperations

Symbolic Designation	Description
R ← shl R	Shift-left register R
R ← shr R	Shift-right register R
R ← cil R	Circular shift-left register R
R ← cir R	Circular shift-right register R
R ← ashl R	Arithmetic Shift-left register R
R ← ashr R	Arithmetic Shift-right register R

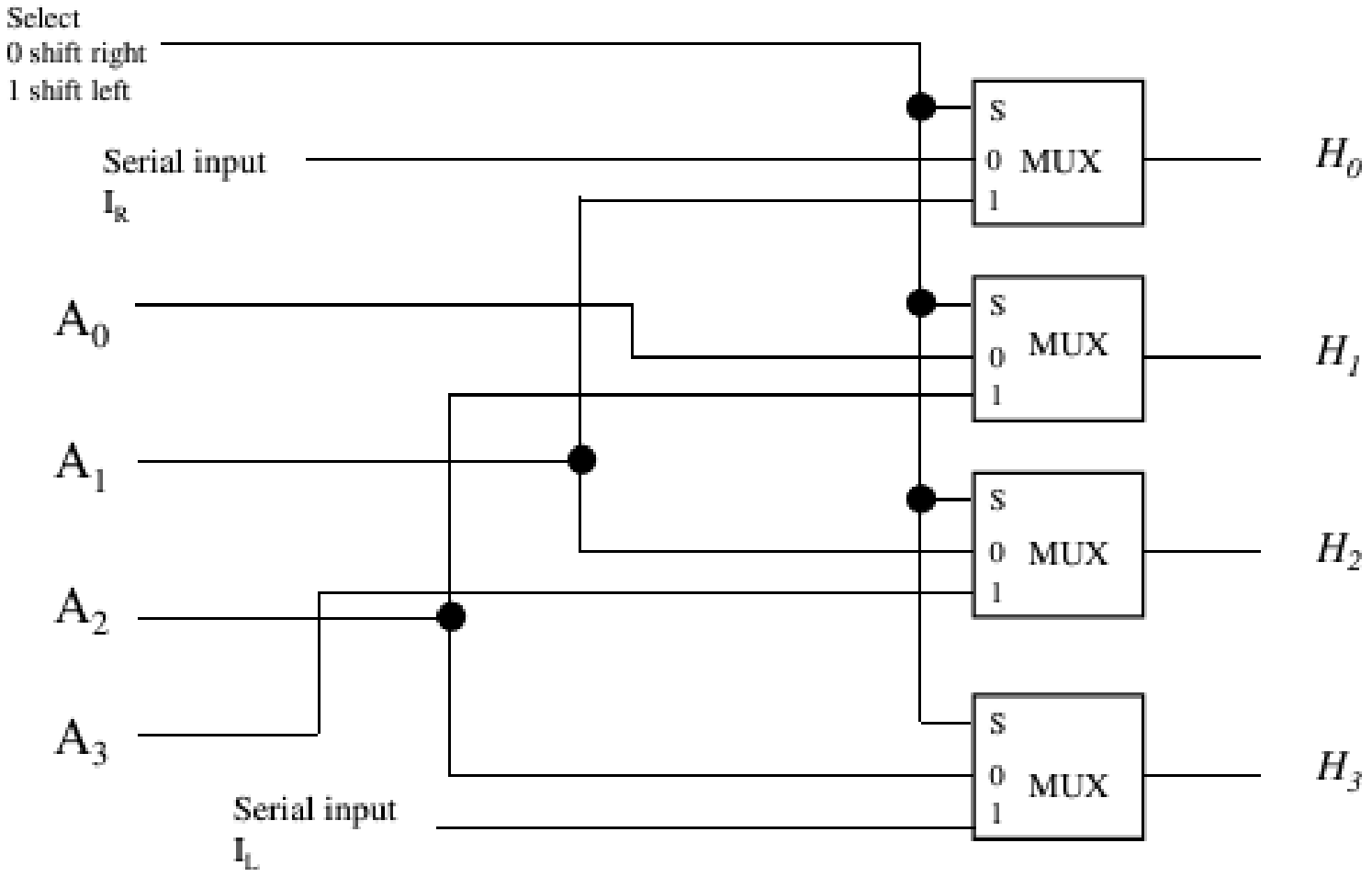
Shift Microoperations

Symbolic Designation	Description
R ← shl R	Shift-left register R
R ← shr R	Shift-right register R
R ← cil R	Circular shift-left register R
R ← cir R	Circular shift-right register R
R ← ashl R	Arithmetic Shift-left register R
R ← ashr R	Arithmetic Shift-right register R

Arithmetic Shift Right

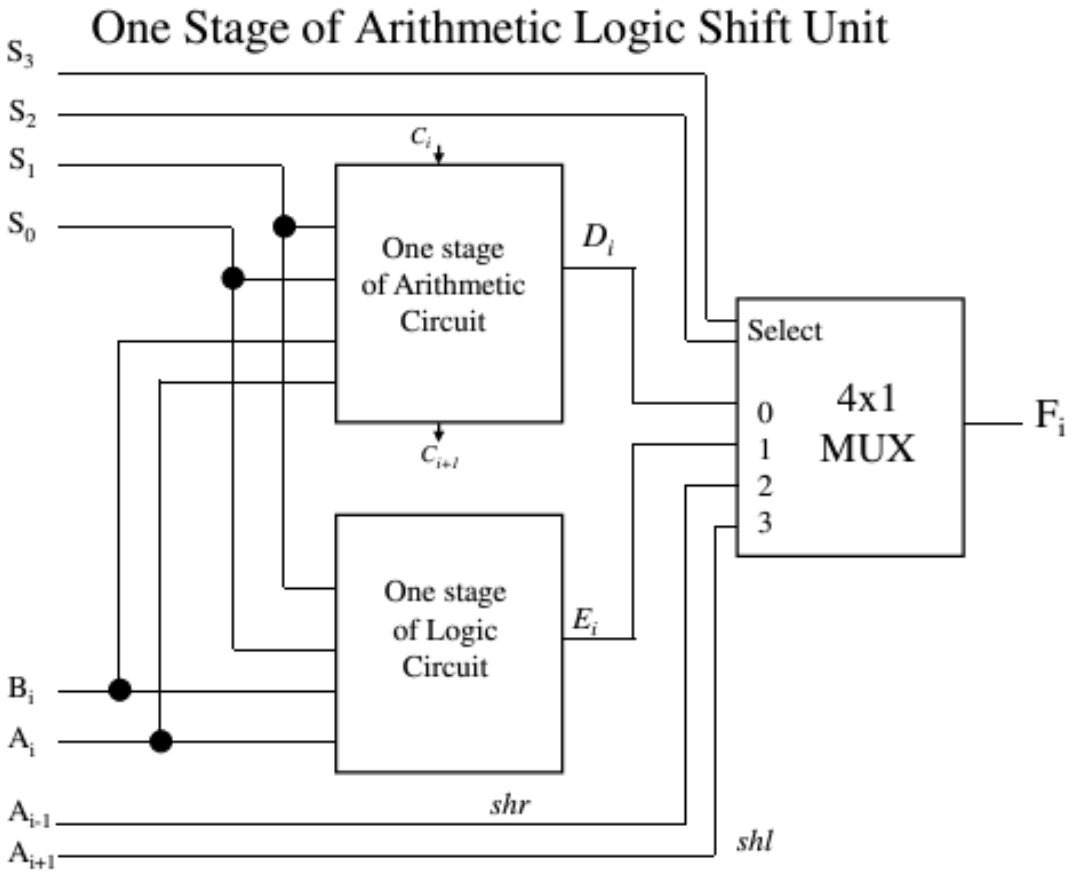


4 bit Combinational Circuit Shifter



Arithmetic Logic Shift Unit

- Instead of having individual registers performing the various microoperations, computers use an Arithmetic Logic Unit which combines these functions.



Function Table for Arithmetic Logic Shift Unit

Operation Select

S_3	S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	0	F =A	Transfer A
0	0	0	0	1	F = A + 1	Increment A
0	0	0	1	0	F = A + B	Addition
0	0	0	1	1	F = A + B + 1	Add with Carry
0	0	1	0	0	$F = A + \overline{B}$	Subtract with Borrow
0	0	1	0	1	$F = A + \overline{B} + 1$	Subtraction
0	0	1	1	0	F = A – 1	Decrement A
0	0	1	1	1	F = A	Transfer A
0	1	0	0	x	$F = A \wedge B$	AND
0	1	0	1	x	$F = A \vee B$	OR
0	1	1	0	x	$F = A \oplus B$	XOR
0	1	1	1	x	F = A	Complement A
1	0	x	x	x	F = shr A	Shift-Right A into F
1	1	x	x	x	F = shl A	Shift-Left A into F

Chapter – V

- **Title: Basic Computer Organization and Design**
- **Topics:**
  - Instruction Codes
  - Computer Registers
  - Computer Instructions
  - Timing and Control
  - Instruction Cycle
  - Memory Reference Instructions
  - Input – Output and Interrupt.

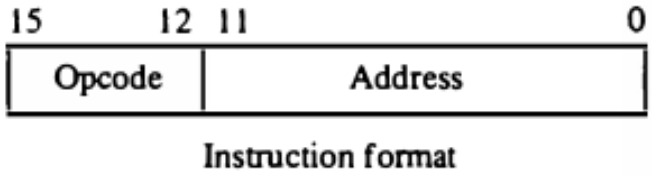
Instruction Codes

- An instruction code is a group of bits that instruct the computer to perform a specific operation.
- The operation code of an instruction is a group of bits that define operations such as addition, subtraction, shift, complement, etc.
- An instruction must also include one or more operands, which indicate the registers and/or memory addresses from which data is taken or to which data is deposited.
- Microoperations:
- The instructions are stored in computer memory like the data is stored.
- The control unit interprets these instructions and uses the operations code to determine the sequences of microoperations that must be performed to execute the instruction.

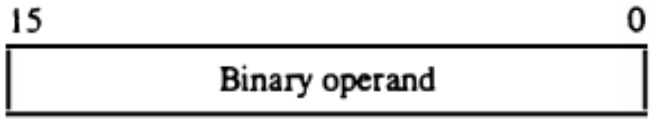
Instruction Codes

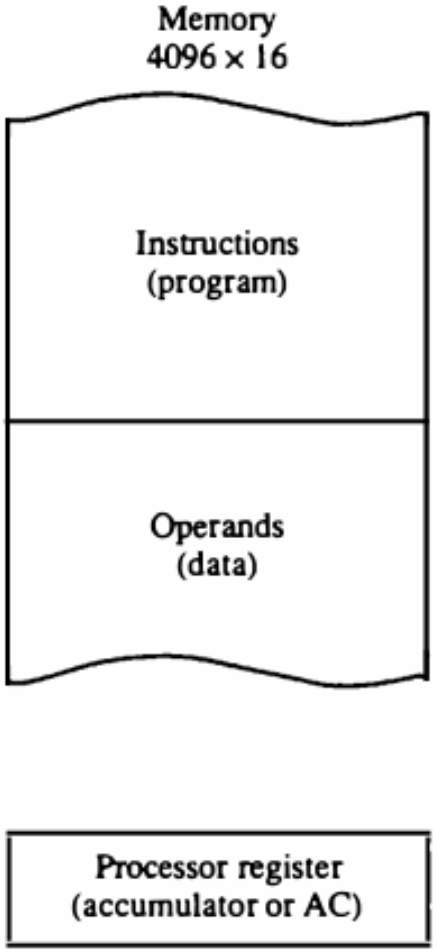
- **Store Program Organization**:
- The operands are specified by indicating the registers and/or memory locations in which they are stored.
  - k bits can be used to specify which of 2^k registers (or memory locations) are to be used.
- The simplest design is to have one processor register (called the ***accumulator***) and two fields in the instruction, one for the ***opcode*** and one for the operand.
- Any operation that does not need a memory operand frees the other bits to be used for other purposes, such as specifying different operations.

Instruction Codes



Instruction format

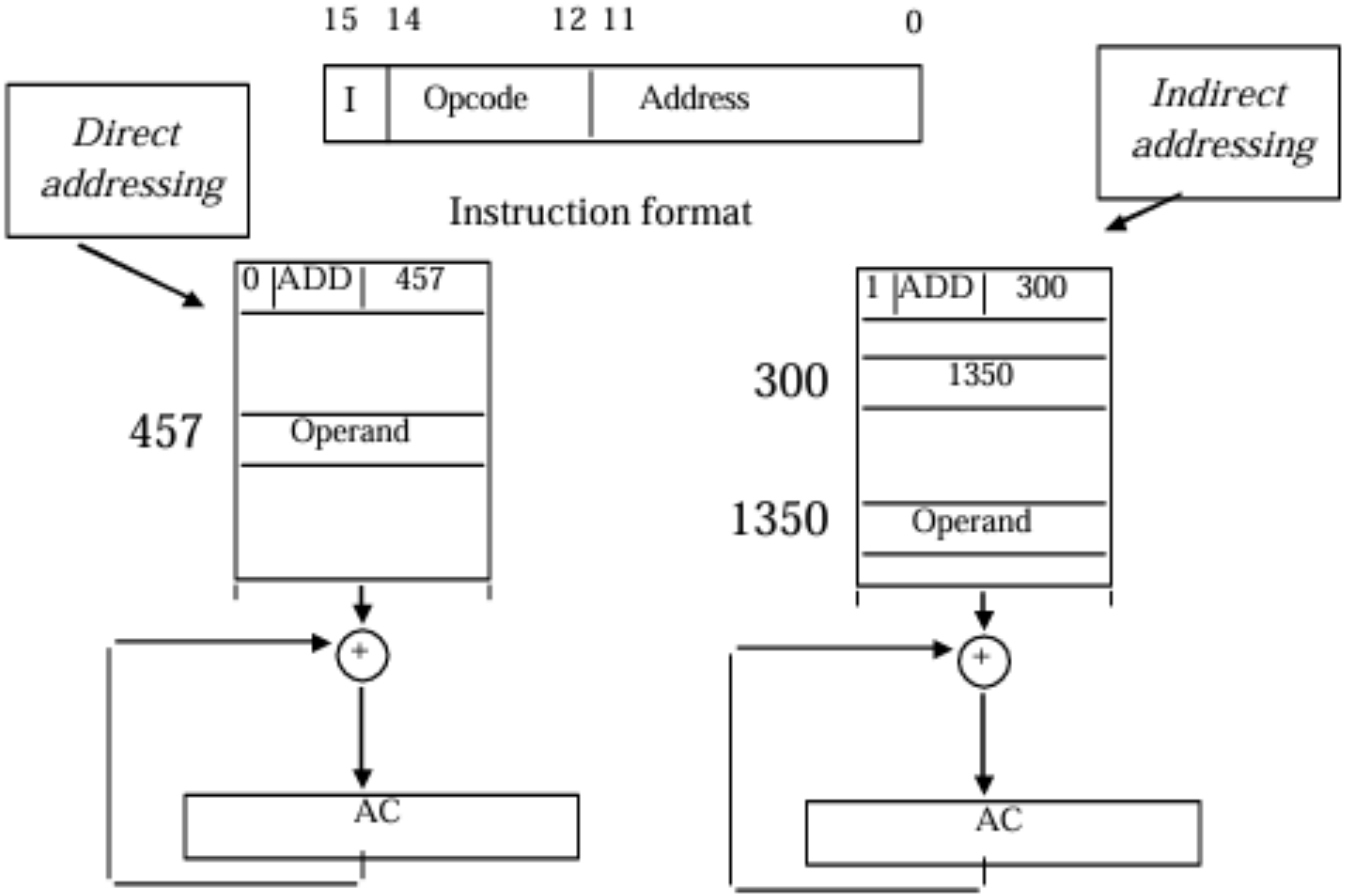




Instruction Codes

- **Addressing Modes**
- Four different types of operands can appear in an instruction:
  - Direct operand
    - an operand stored in the register or the memory location specified.
  - Indirect operand
    - an operand whose address is stored in the register or the memory location specified.
  - Immediate operand
    - an operand whose value is specified in the instruction.

Instruction Codes



Computer Registers

- Computer instructions are stored in consecutive locations and are executed sequentially; this requires a register which can stored the address of the next instruction; we call it the ***Program Counter***.
- We need registers which can hold the address at which a memory operand is stored as well as the value itself.
- We need a place where we can store
  - temporary data
  - the instruction being executed,
  - a character being read in
  - a character being written out.

Computer Registers

TABLE 5-1 List of Registers for the Basic Computer

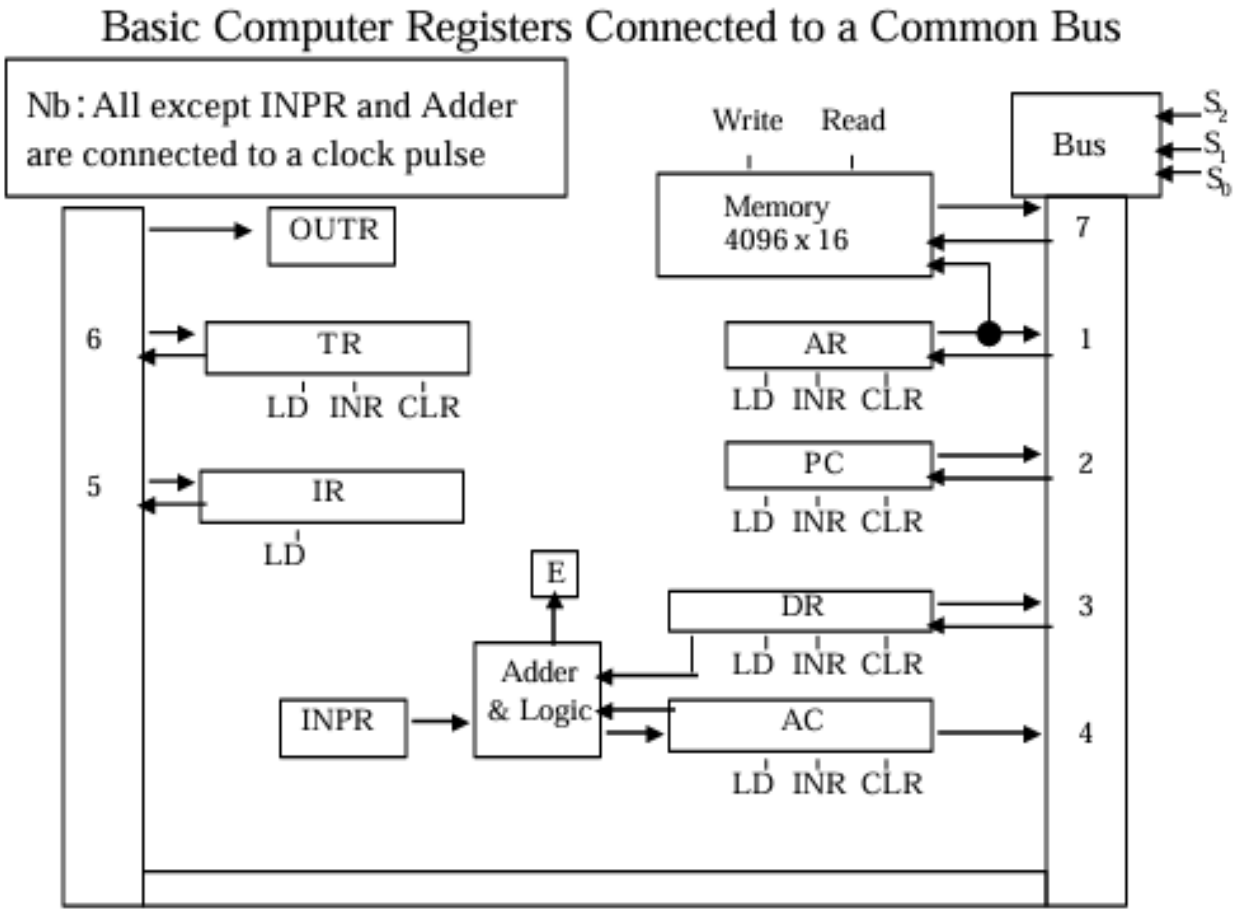
Register symbol	Number of bits	Register name	Function
DR	16	Data register	Holds memory operand
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
PC	12	Program counter	Holds address of instruction
TR	16	Temporary register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

Computer Registers

- **Common Bus System:**
- To avoid excessive wiring, memory and all the registers are connected via a common bus.
- The specific output that is selected for the bus is determined by $S_2\ S_1\ S_0$.
- The register whose LD (Load) is enabled receives the data from the bus.
- Registers can be incremented by setting the INR control input and can be cleared by setting the CLR control input.
- The Accumulator's input must come via the Adder & Logic Circuit. This allows the Accumulator and Data Register to swap data simultaneously.
- The address of any memory location being accessed must be loaded in the Address Register.

Computer Registers

- **Common Bus System:**

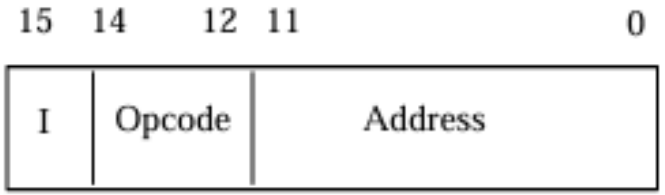


Computer Instructions

- The basic computer has three instruction code formats:
  - Memory-reference format: where seven 3-bit opcodes are followed by a 12-bit memory address and preceded by a bit which indicates whether direct or indirect addressing is being used.
  - Register-reference format: where 0111_2 is followed by 12 bits which indicate a register instruction.
  - Input-output format: where 1111_2 is followed by 12 bit which indicate an input-output instruction.
- In register-reference and I/O formats, only one of the lower 12 bits is set.

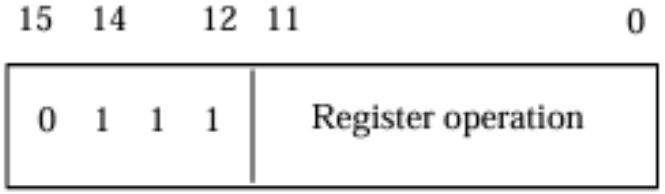
Computer Instructions

- The basic computer instruction formats:



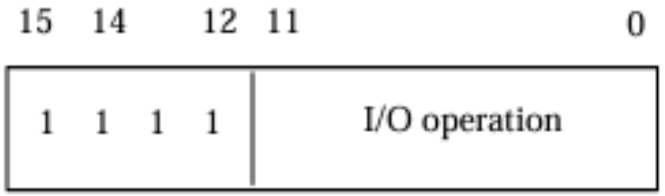
Memory-reference instruction

Opcode = 000 through 110



Register-reference instruction

Opcode = 111, I = 0



Input-output instruction

Opcode = 111, I = 1

Computer Instructions

- **Instruction Set Completeness:** A computer instruction set is said to be complete if the computer includes a sufficient number of instructions in each of these categories:
  - Arithmetic, logical and shift instructions
  - Instructions for moving data from registers to memory and memory to registers.
  - Program-control and status-checking instructions
  - Input and output instructions
- **Arithmetic, Logic and Shifting Completeness:** We have instructions for adding, complementing and incrementing the accumulator. Using these we can also subtract.
  - AND and complement provide NAND, from which all other logical operations can be constructed.
  - Using circular shift operations we can construct logical and arithmetic shifts.

Computer Instructions

- We can construct multiply and divide by adding, subtracting and shifting.
- While this is complete, it is not very efficient; it would be to our advantage to have subtract, multiply, OR and XOR.
- We can perform moves using the LDA and STA instructions.
- We have unconditional branches (BUN), subprogram calls (BSA) and conditional branches (ISZ).
- We also have all the instructions we need to perform input and output and handle the interrupt that they generate.

Computer Instructions

- **Basic Register-Reference Instructions:**

	Hexadecimal code		
Symbol	**I = 0**	**I = 1**	**Description**
AND	0xxx	8xxx	AND mem. Word to AC
ADD	1xxx	9xxx	ADD mem. Word to AC
LDA	2xxx	Axxx	Load mem. Word to AC
STA	3xxx	Bxxx	Store Content of AC in mem.
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero

Computer Instructions

- **Basic Register-Reference Instructions:**

Symbol	Hex. Code	Description
CLA	7800	Clear AC
CLE	7400	Clear E
CMA	7200	Complement AC
CME	7100	Complement E
CIR	7080	Circulate right AC & E
CIL	7040	Circulate left AC & E
INC	7020	Increment AC

Computer Instructions

- **Basic Register-Reference Instructions:**

Symbol	Hex. Code	Description
SPA	7010	Skip next instruction if AC is positive
SNA	7008	Skip next instruction if AC is negative
SZA	7004	Skip next instruction if AC is zero
SZE	7002	Skip next instruction if E is zero
HLT	7001	Halt computer

Computer Instructions

- **Basic Input-output Instructions:**

Symbol	Hex. Code	Description
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

Computer Instructions

- **Basic Input-output Instructions:**

Symbol	Hex. Code	Description
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

Timing and Control

- The timings for all the registers are controlled by a master clock generator.
  - Its pulses are applied to all flip-flops and registers, including in the control unit.
  - The control signals are generated in the control unit and provide control inputs for the bus's multiplexers and the processor registers respectively provide microoperations for the accumulator.
- There are two types of control:
  - *Hardwired–control* logic is implemented with gates, flip-flops, decoders and other digital circuits.
  - *Microprogrammed–control* information is stored in a control program, which is programmed to perform the necessary steps to implement instructions.

Timing and Control

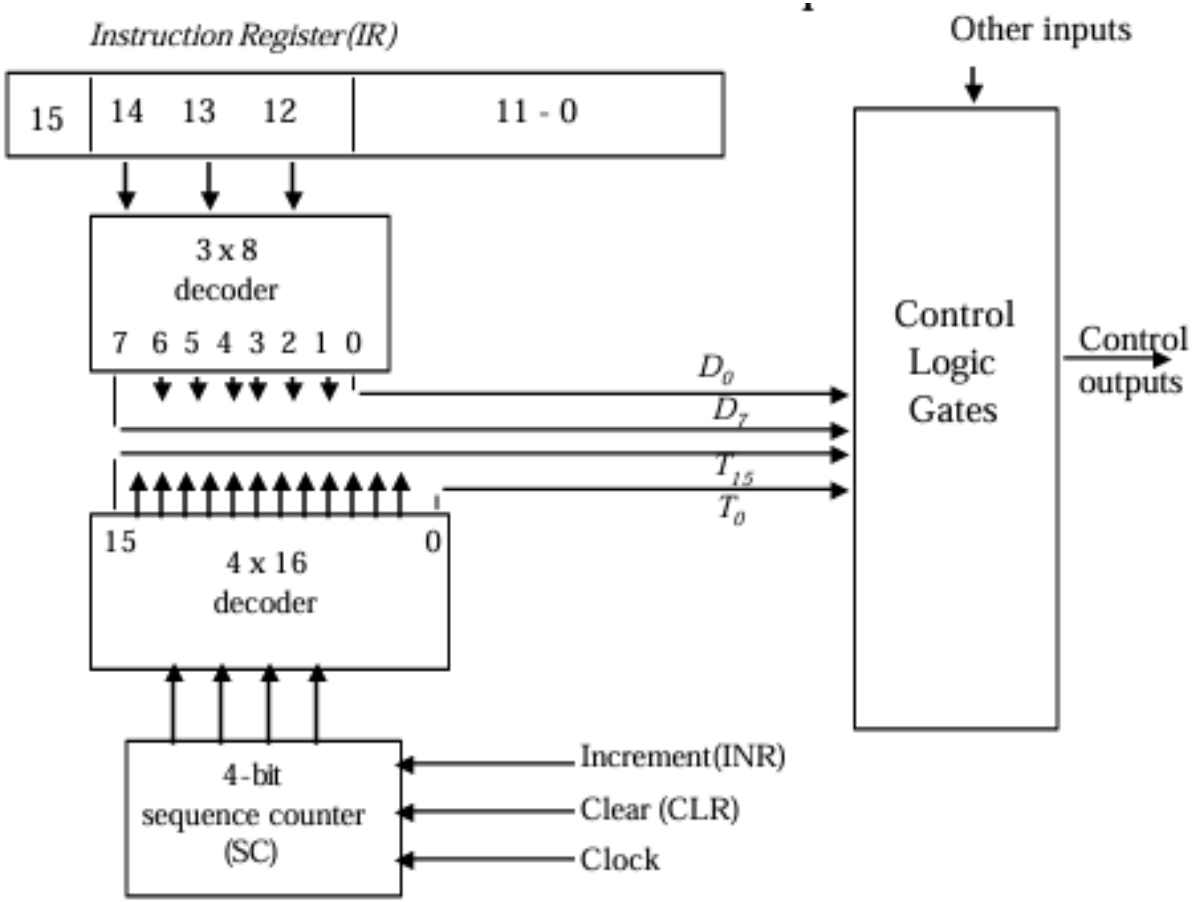
- **Timing Signals:**
- Timing signals are generated by the sequence counter (SC), which receives as inputs the clock pulse, increment and clear.
- The SC's outputs are decoded into 16 timing signals T_0 through T_{15} , which are used to control the sequence of operations.
- The RTL statement

$$D_3T_4: \ SC \leftarrow 0$$

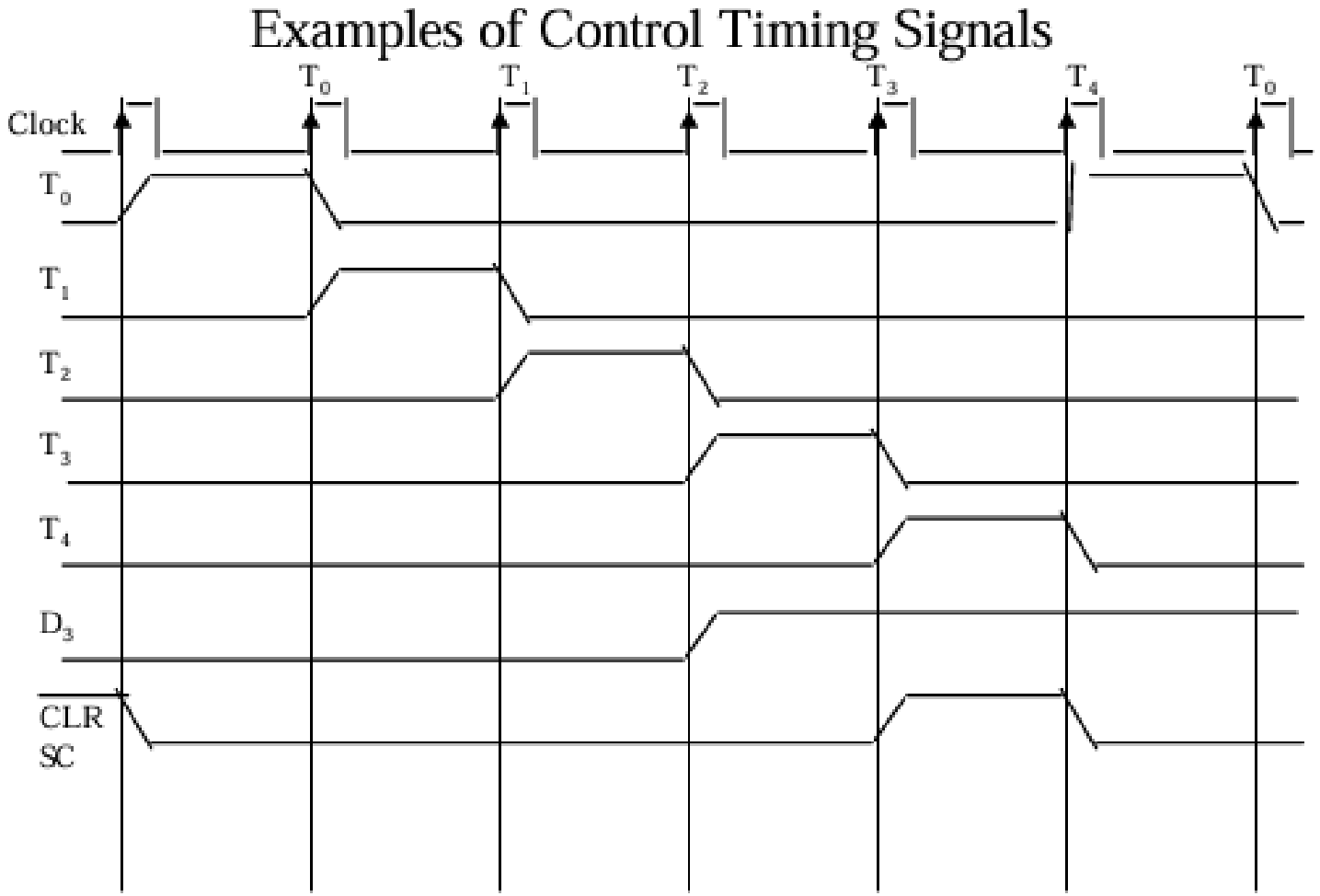
- resets the sequence counter to zero; the next timing signal is T_0

Timing and Control

- Control Unit of Basic Computer:



Timing and Control



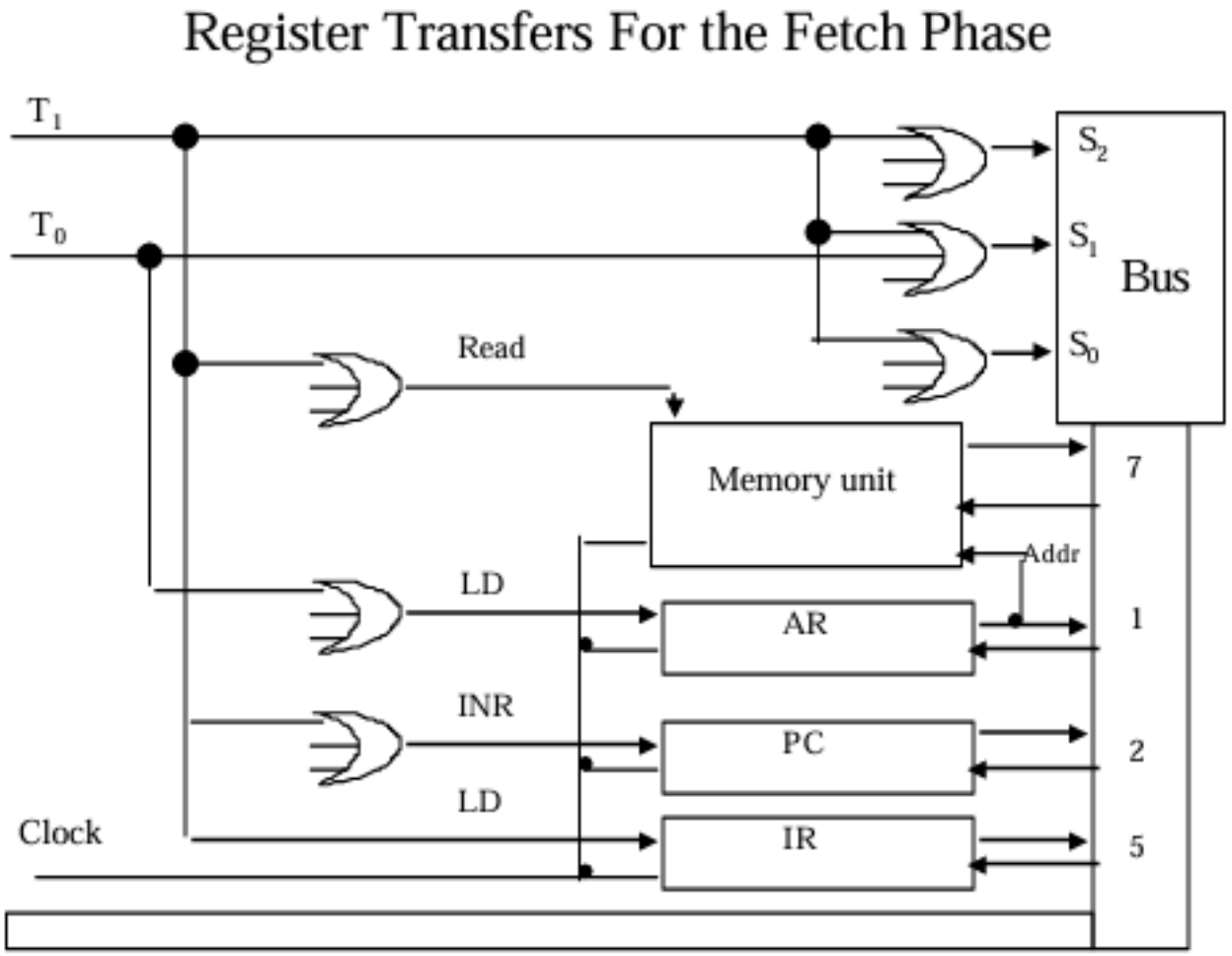
Instruction Cycle

- The instructions of a program are carried out by a process called the instruction cycle.
- The instruction cycle consists of these phases:
  - Fetch an instruction from memory
  - Decode the instruction
  - Read the effective address from memory if the operand has an indirect address.
  - Execute the instruction.
- ***Fetch and Decode:*** Initially, the PC has stored the address of the instruction about to be executed and the SC is cleared to 0.
- With each clock pulses the SC is incremented and the timing signals go through the sequence T_0, T_1, T_2, etc.
- It is necessary to load the AR with the PC's address (it is connected to memory address inputs): T_0 : AR ← PC

Instruction Cycle

- ***Fetch and Decode:***
- Subsequently, as we fetch the instruction to be executed, we must increment the program counter so that it points to the next instruction:
- T_1 : IR ←M[AR], PC ←PC + 1
- To carry out the instruction, we must decode and prepare to fetch the operand. In the event it is an indirect operand, we need to have the indirect addressing bit as well:
- T_2 : D_0, … D_7 ← Decode IR(12-14), AR ← IR (0-11), I ← IR(15)

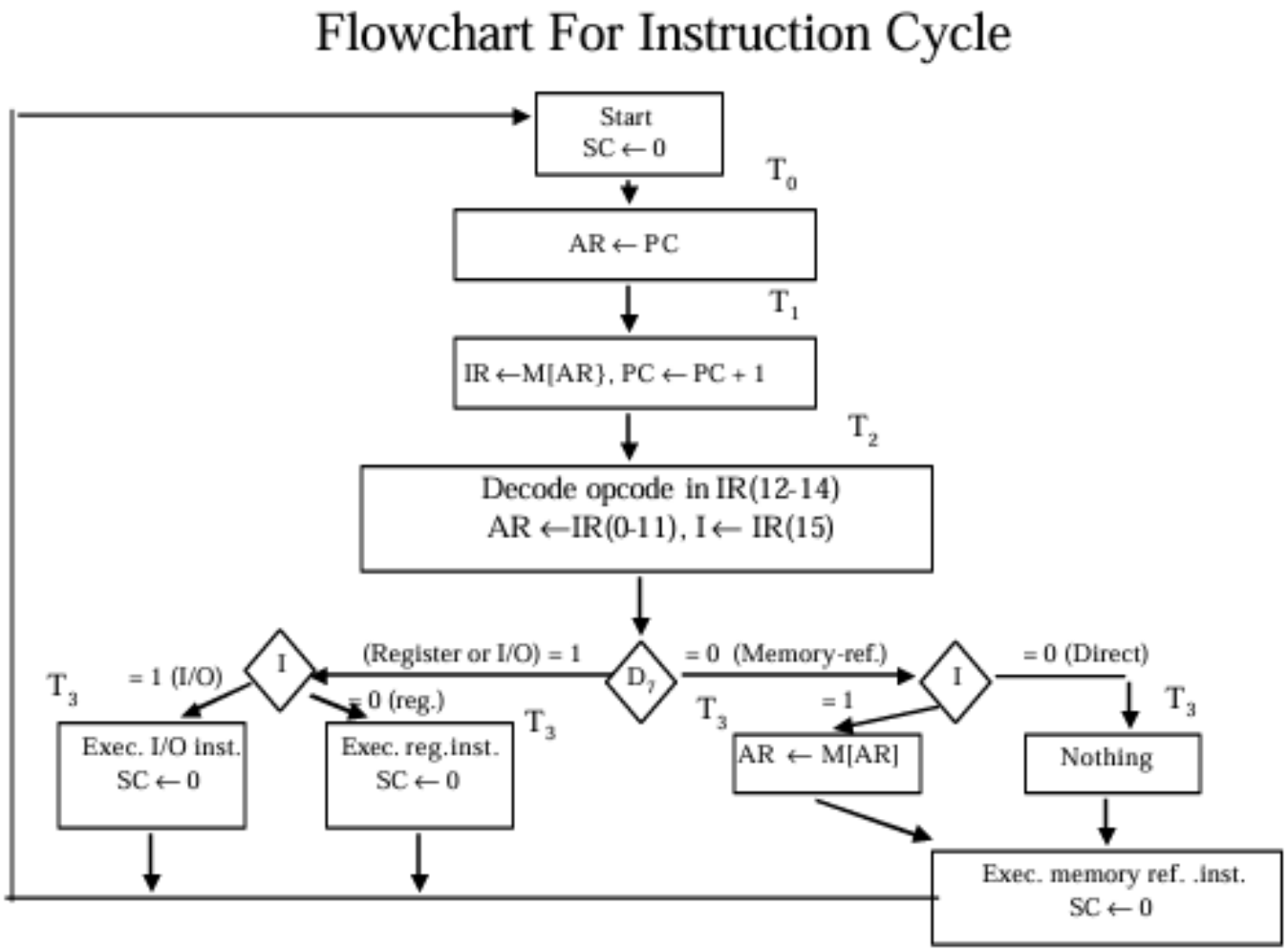
Instruction Cycle



Instruction Cycle

- **Type of Instruction and Addressing**
- During time T_3, the control unit determines if this is a memory-reference, register-reference or input/output instruction.
  - The latter two are distinguished by the I (indirect) bit.
  - If it is a memory-reference instruction, the I bit will determine direct or indirect addressing.
- The four separate paths are:
  - D_7'I: AR < M[AR]
  - D_7'I'T_3: Nothing
  - D_7 I'T_3: Execute a register-reference instruction
  - D_7 IT_3: Execute an input-output instruction

Instruction Cycle



Instruction Cycle

Execution of Register-Reference Instructions

$D_7I'T_3 = r$ (common to all register-reference instructions)
$IR(I) = B_i$ [bit in IR(0-11) that specifies the operation]

	r	SC ← 0	Clear SC
CLA	rB_{11}	AC ← 0	Clear AC
CLE	rB_{10}	E ← 0	Clear E
CMA	rB_9	AC ← AC'	Complement AC
CME	rB_8	E ← E'	Complement E
CIR	rB_7	AC ← shr AC, AC(15) ← E E ← AC(0)	Circulate right
CIL	rB_6	AC ← shl AC, AC(0) ← E E ← AC(15)	Circulate left
INC	rB_5	AC ← AC + 1	Increment AC

Instruction Cycle

Execution of Register-Reference Instructions

SPA	rB_4	If (AC(15) = 0) then PC ← PC + 1	Skip if positive
SNA	rB_3	If (AC(15) = 1) Then PC ← PC + 1	Skip if negative
SZA	rB_2	If (AC = 0) Then PC ← PC + 1	Skip if AC zero
SZE	rB_1	If (E = 0) Then PC ← PC + 1	Skip if E zero
HLT	rB_0	S ← 0 (S is a start-stop flip-flop)	Halt computer

Memory-Reference Instruction

Symbol	Op. Decoder	Symb. Desc.
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR]$, $E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC$ $PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1$ If $M[AR] + 1 = 0$ Then $PC \leftarrow PC + 1$

Memory-Reference Instruction

- All memory-reference instructions have to wait until T4 so that the timing is the same whether the operand is direct or indirect.
- AND, ADD and LDA must all be performed in two steps because AC can only be accessed via DR:
- AND:
  - D_0T_4: DR ←M[AR]
  - D_0T_5: AC ← AC ∧ DR, SC ← 0
- ADD:
  - D_1T_4: DR ←M[AR]
  - D_1T_5 : AC ← AC + DR, E ←C_{out}, SC ← 0
- LDA:
  - D_2T_4: DR ←M[AR]
  - D_0T_5 : AC ←DR, SC ←0

Memory-Reference Instruction

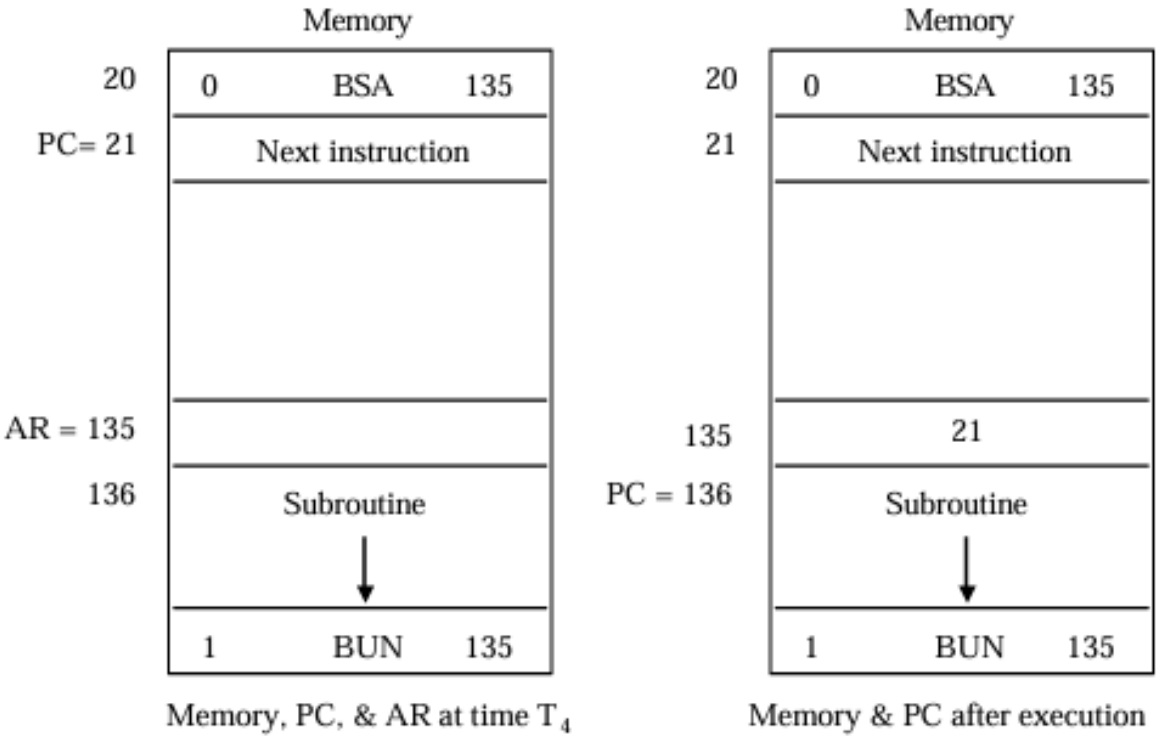
- STA stores the contents of the AC, which can be applied directly to the bus:
  - D_3T_4 : M[AR] ←AC, SC ←0
- BUN transfers control unconditionally to the effective address indicated by the effective address:
  - D_4T_4 : PC ←AR, SC ←0
- BSA is used to branch to a subprogram. This requires saving the return address, which is saved at the operand's effective address with the subprogram beginning one word later in memory:
  - D_5T_4 : M[AR] ← PC, AR ← AR + 1
  - D_5T_5 : PC ← AR, SC ← 0
- ISZ skips the next instruction if the operand stored at the effective address is 0. This requires that the PC be incremented, which cannot be done directly:
  - D_6T_4 : DR ←M[AR]
  - D_6T_5 : DR ←DR + 1

Memory-Reference Instruction

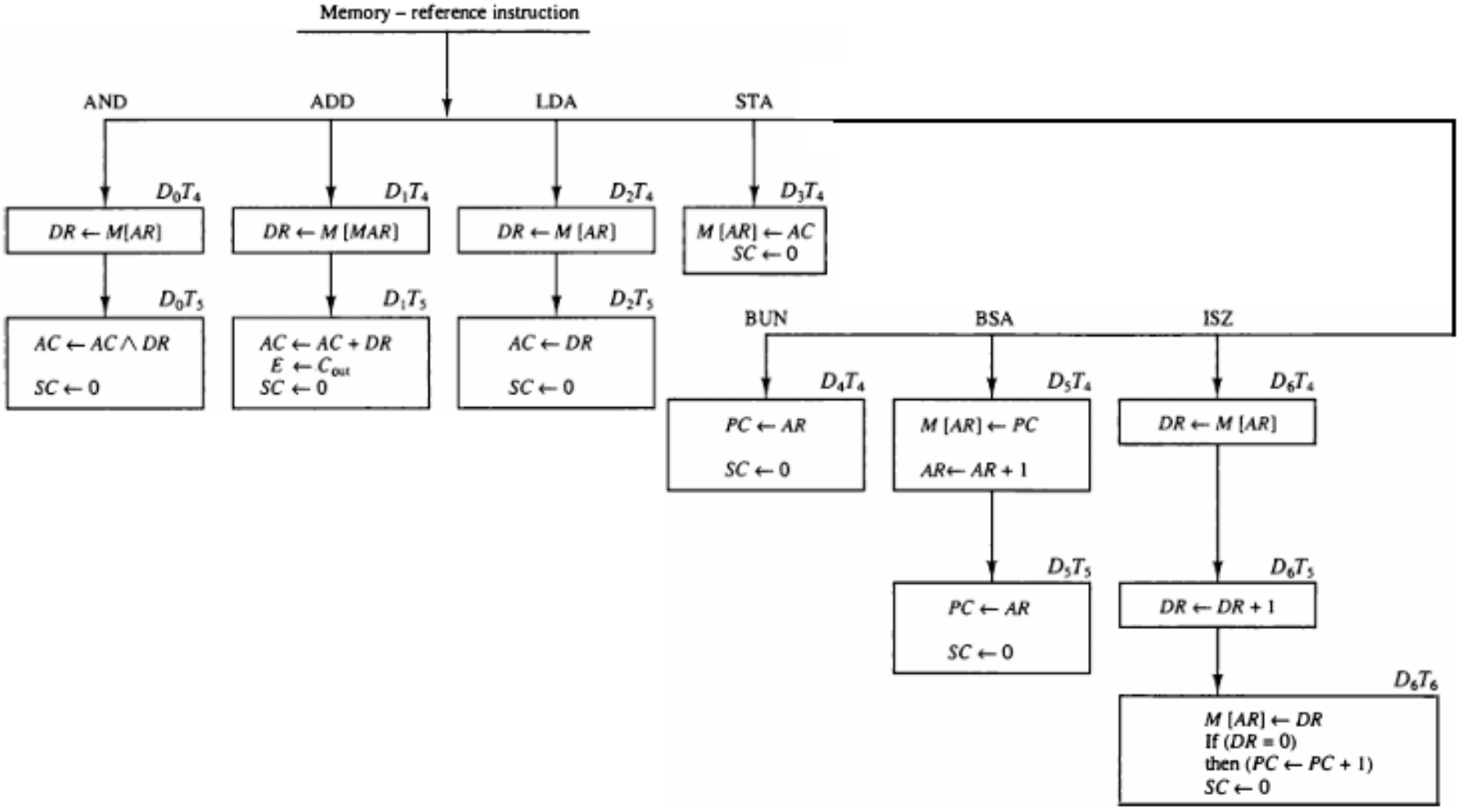
– D_6T_6 : M[AR] ←DR,

if (DR = 0) then (PC ←PC + 1), SC ←0

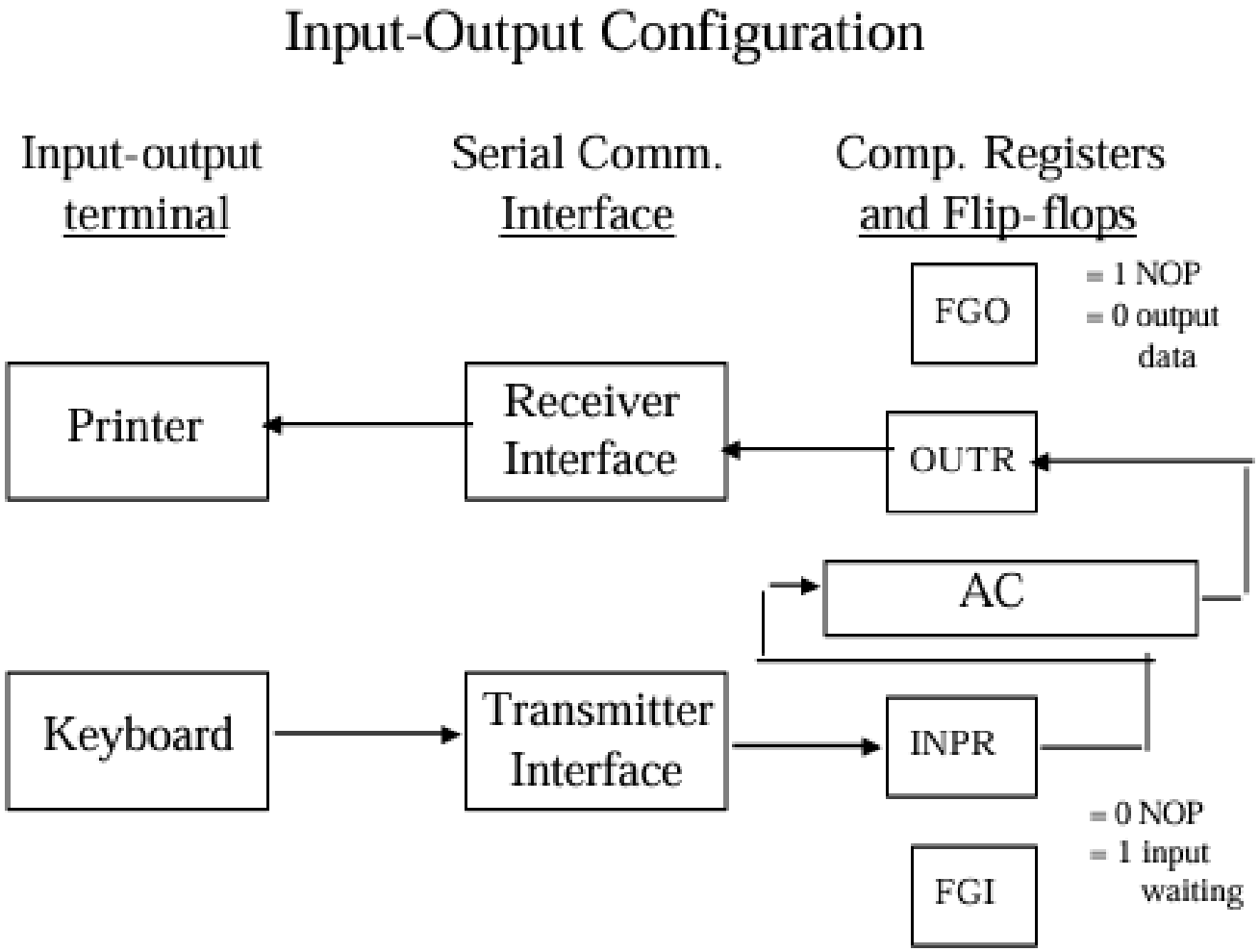
Example of BSA Instruction Execution



Memory-Reference Instruction



Input – Output and Interrupt

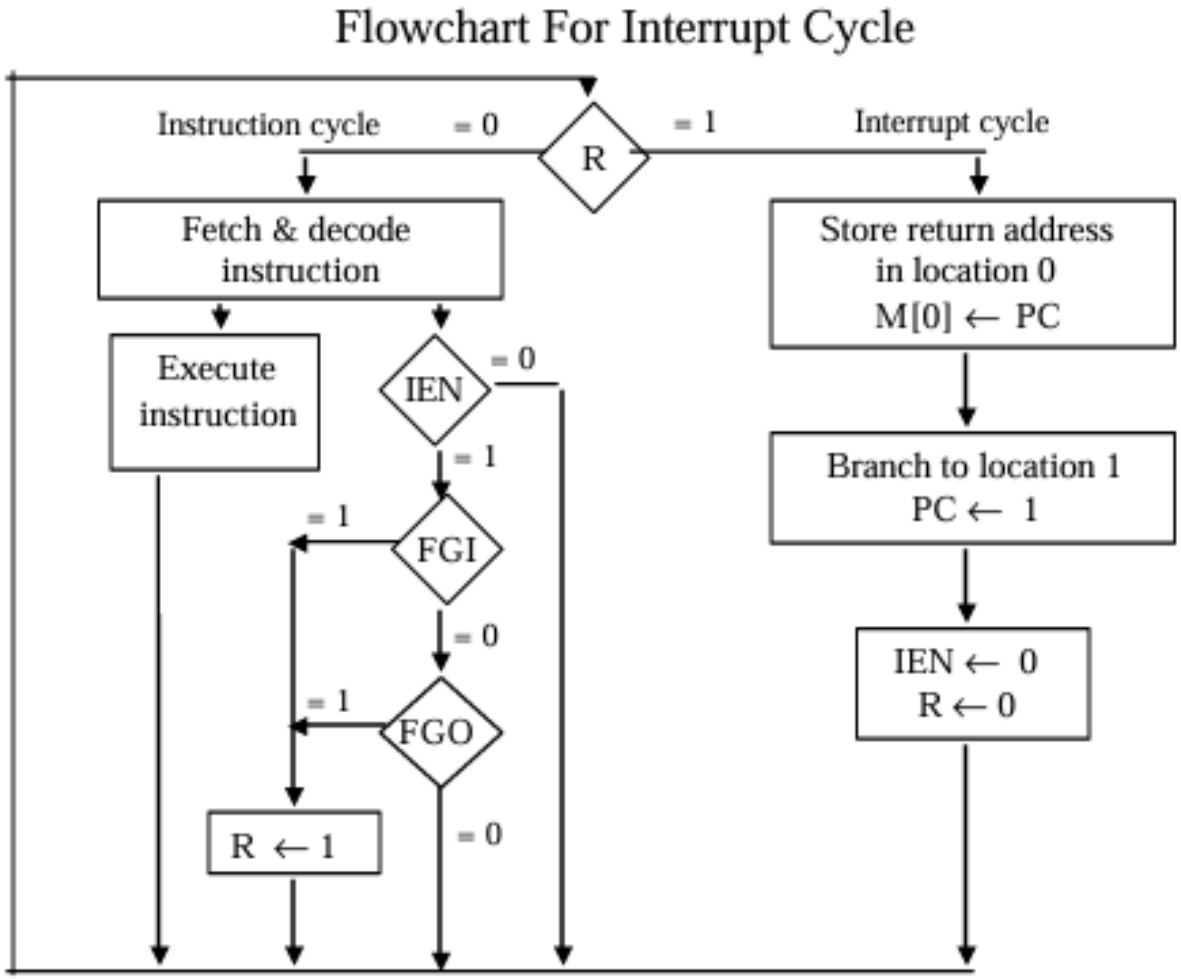


Input – Output and Interrupt

- **Input – Output Instructions:**

	p	SC ← 0	Clear SC
INP	pB11	AC(0-7) ← INPR, FGI ← 0	Input character
OUT	pB10	OUTR ← AC(0-7), FGO ← 0	Output character
SKI	pB9	If (FGI = 1) Then PC ← PC + 1	Skip on input flag
SKO	pB8	If (FGO = 1) Then PC ← PC + 1	Skip on output flag
ION	pB7	IEN ← 1	Interrupt enable on
IOF	pB6	IEN ← 0	Interrupt enable off

Input – Output and Interrupt



Input – Output and Interrupt