- **REGISTER TRANSFER AND MICRO OPERATIONS**

# Register Transfer and Microoperations

- **The operations executed on data stored in registers are called *microoperations*. A microoperation is an elementary operation performed on the information stored in one or more registers. The result of the operation may replace the previous binary information of a register or may be transferred to another register. Examples of microoperations are shift, count, clear, and load. The internal hardware organization of a digital computer is best defined by specifying:**
- ***The set of registers it contains and their function.***
- ***The sequence of microoperations performed on the binary information stored in the registers.***
- ***The control that initiates the sequence of microoperations.***
- The symbolic notation used to describe the microoperation transfers among registers is called a *register transfer language*. The term "register transfer" implies the availability of hardware logic circuits that can perform a stated microoperation and transfer the result of the operation to the same or another register. A register transfer language is a system for expressing in symbolic form the microoperation sequences among the registers of a digital module.

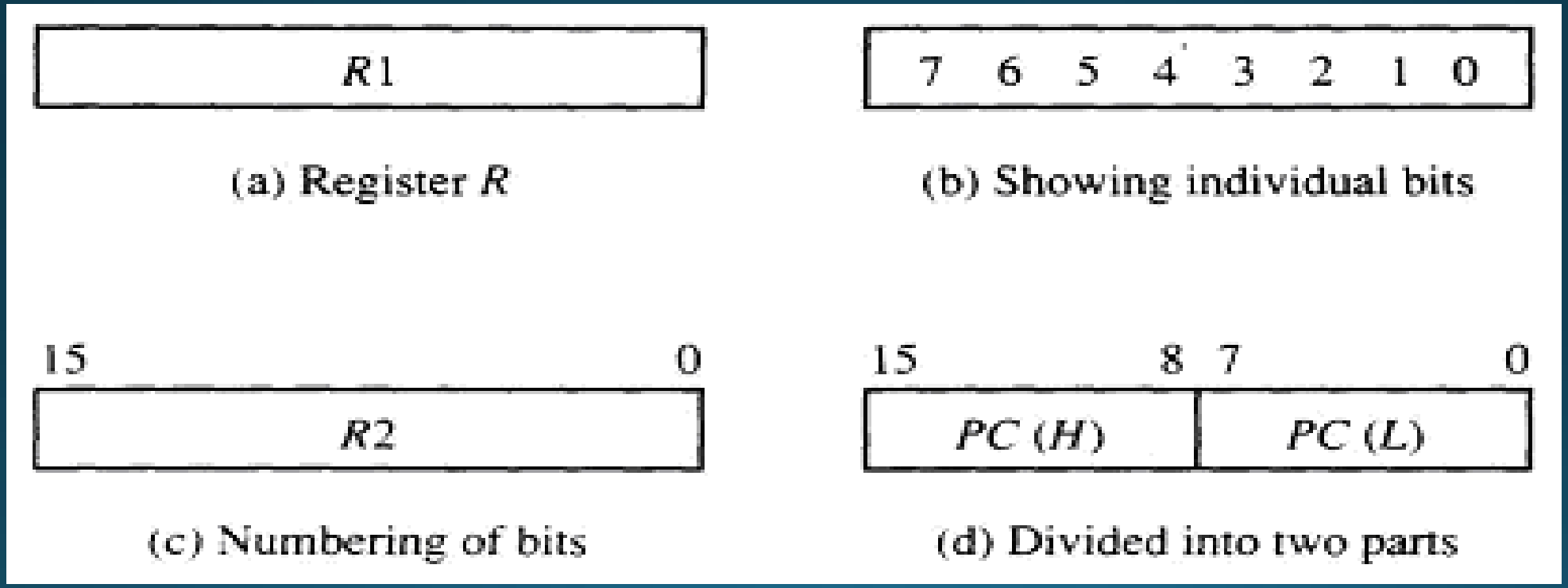
# REGISTER TRANSFER

- Computer registers are designated by capital letters (sometimes followed by numerals) to denote the function of the register. For example, the register that holds an address for the memory unit is usually called a *memory address register* and is designated by the name *MAR*. Other designations for registers are *PC* (for *program counter*), *IR* (for *instruction register*), and *R1* (for *processor register*). The individual flip-flops in an *n-bit* register are numbers in sequence from *0* through *n-1*, starting from *0* in the rightmost position and increasing the numbers toward the left. Fig.1 shows the representation of registers in block diagram form.

# Register Transfer

- The most common way to represent a register is by a rectangular box with the name of the register inside, as in diagram. The individual bits can be distinguished as in (b). The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c). A 16-bit register is partitioned into two parts in (d). Bits 0 through 7 are assigned the symbol *L* (for low byte) and bits 8 through 15 are assigned the symbol *H* (for high byte). The name of the 16-bit register is PC. The symbol *PC (0-7)* or *PC (L)* refers to the low-order byte and *PC (8-15)* or *PC (H)* to the high-order byte.

# DIAGRAMMATIC VIEW



# Register Transfer

Information transfer from one register to another is designated in symbolic form by means of a replacement operator. The statement R1 ← R2 denotes a transfer of the content of register R1 into register R2. It designates a replacement of the content of R2 by the content of R1. By definition, the content of the source register R1 does not change after the transfer.

The basic symbols of the register transfer notation are listed in Table .

| 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* |

# Register Transfer

- Registers are denoted by capital letters, and numerals may follow the letters. Parentheses are used to denote a part of a register by specifying the range of bits or by giving a symbol name to a portion of a register. The arrow denotes a transfer of information and the direction of transfer. A comma is used to separate two or more operations that are executed at the same time. The statement
  - T: R2 ← R1, R1 ← R2
- denotes an operation that exchanges the contents of two registers during one common clock pulse provided that T = 1. This simultaneous operation is possible with registers that have edge-triggered flip-flops.

# Bus and Memory Transfer

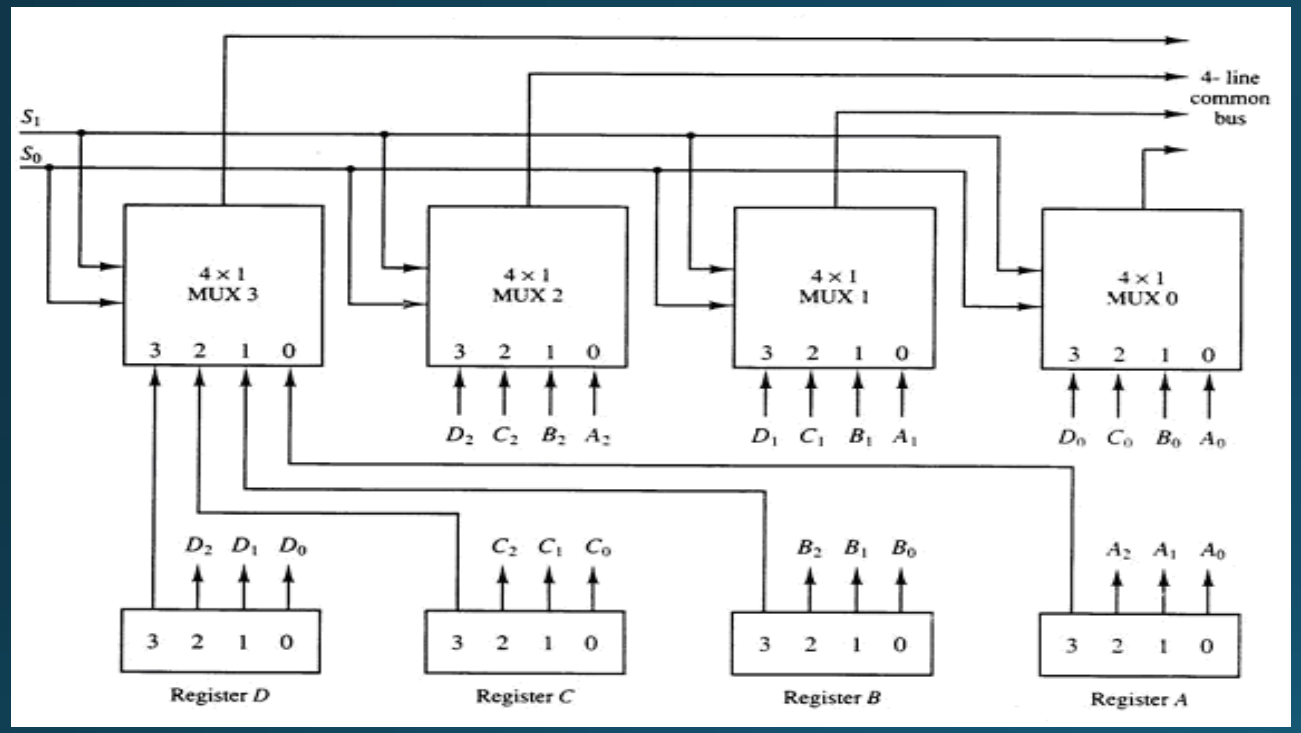
- A more efficient scheme for transferring information between registers in a multiple-register configuration is a *common bus system*. A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals determine which register is selected by the bus during each particular register transfer.

# Bus and Memory Transfer

- One way of constructing a common bus system is with *multiplexers*. The multiplexers select the source register whose binary information is then placed on the bus. The construction of a bus system for four registers is shown in following diagram. Each register has four bits, numbered 0 through 3. The bus consists of four 4 x 1 multiplexers each having four data inputs, 0 through 3, and two selection inputs, $S_1$ and $S_0$.

# Bus and Memory Transfer

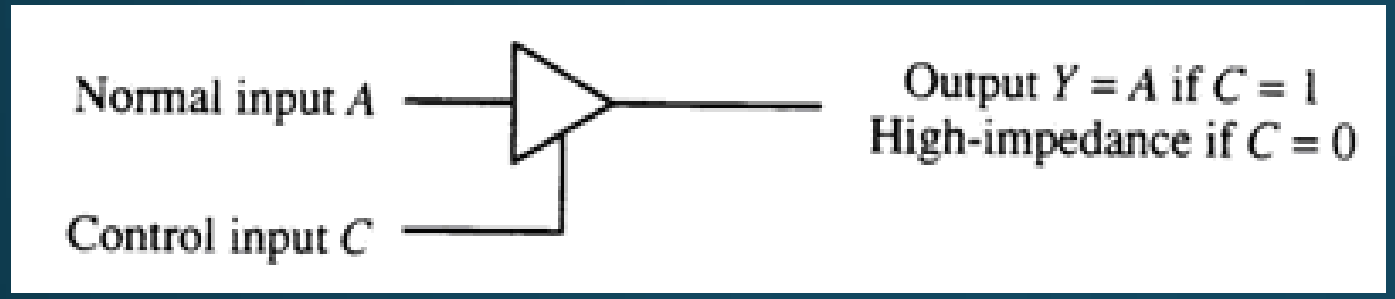
- In order not to complicate the diagram with 16 lines crossing each other, labels are used to show the connections from the outputs of the registers to the inputs of the multiplexers. For example, output 1 of register A is connected to input 0 of MUX 1 because this input is labeled




| $S_1$ | $S_0$ | Register selected |
|---|---|---|
| 0 | 0 | A |
| 0 | 1 | B |
| 1 | 0 | C |
| 1 | 1 | D |

# *Three-State Bus Buffers*

- A bus system can be constructed with *three-state gates* instead of multiplexers. A three-state gate is a digital circuit that exhibits three states. Two of the states are signals equivalent to logic 1 and 0 as in a conventional gate. The third state is a *high-impedance state*. The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have a logical significance.
- The graphic symbol of a *three-state buffer* gate is shown in diagram. It is distinguished from a normal buffer by having both a normal input and a control input. The control input determines the output state. When the control input is equal to 1, the output is enabled and the gate behaves like any conventional buffer, with the output equal to the normal input. When the control input is 0, the output is disabled and the gate goes to a high-impedance state, regardless of the value in the normal input




The symbol of three state bus buffer

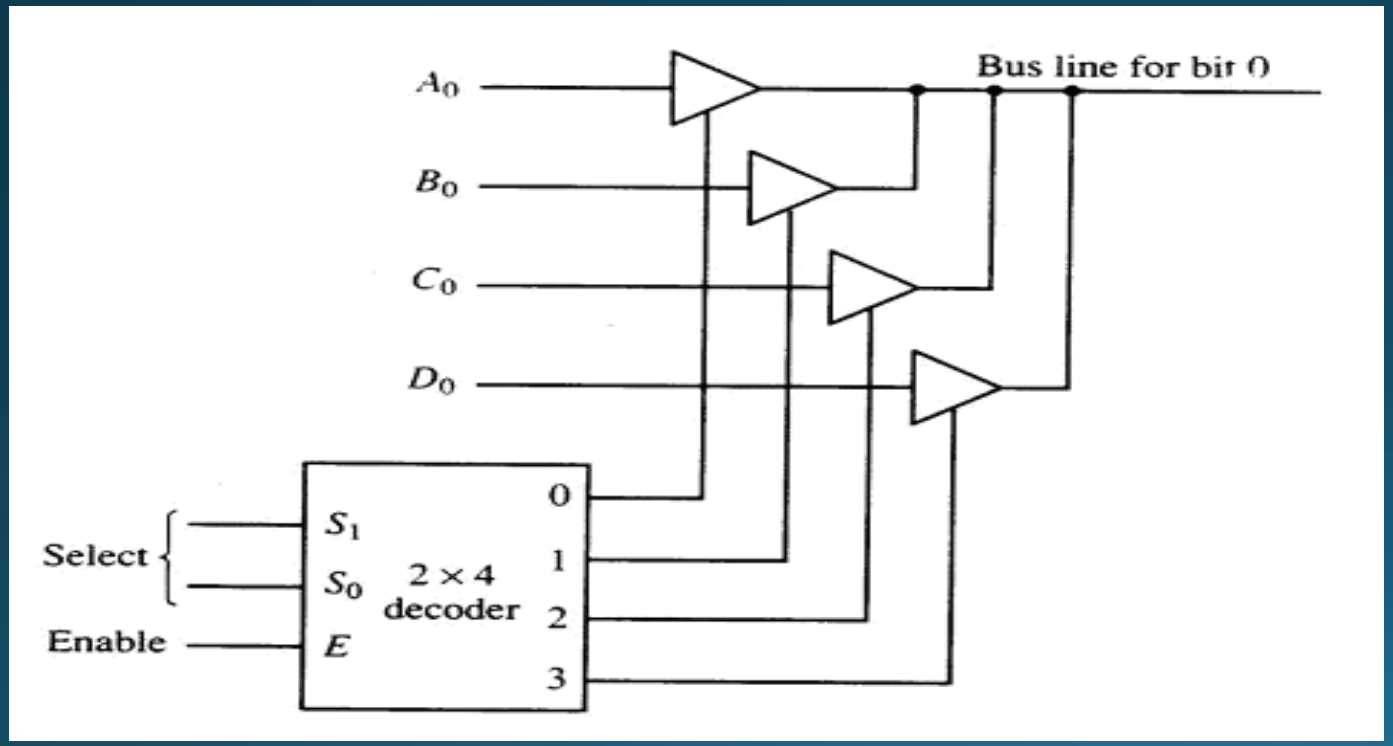



Diagram of bus line with three state-buffers

- The construction of a *bus system* with three-state buffers is demonstrated in above diagram . The outputs of four buffers are connected together to form a single bus line. (It must be realized that this type of connection cannot be done with gates that do not have three-state outputs). The control inputs to the buffers determine which of the four normal inputs will communicate with the bus line.
- One way to ensure that no more than one control input is active at any given time is to use a decoder, as shown in the diagram. When the enable input of the decoder is 0, all of its four outputs are 0, and the bus line is in a high-impedance state because all four buffers are disabled. When the enable input is active, one of the three-state buffers will be active, depending on the binary value in the select inputs of the decoder.

# Memory Transfer

- The transfer of information from a memory word to the outside environment is called a *read* operation. The transfer of new information to be stored into the memory is called a *write* operation. A memory word will be symbolized by the letter M. The particular memory word among the many available is selected by the memory address during the transfer. It is necessary to specify the address of M when writing memory transfer operations. This will be done by enclosing the address in square brackets following the letter M.

# Memory Transfer

- Consider a memory unit that receives the address from a register, called the address register, symbolized by AR. The data are transferred to another register, called the data register, symbolized by DR. The read operation can be stated as follows:

Read: DR ← M[AR]

- This causes a transfer of information into DR from the memory word M selected by the address in AR.

# Memory Transfer

- The write operation transfers the content of a data register to a memory word M selected by the address. Assume that the input data are in register R1 and the address is in AR. The write operation can be stated symbolically as follows:
- Write: M[AR] ← R1
- This causes a transfer of information from R1 into the memory word M selected by the address in AR.

# *Arithmetic Microoperations:*

- A microoperation is an elementary operation performed with the data stored in registers. The microoperations most often encountered in digital computers are classified into four categories:
- 1. Register transfer microoperations transfer binary information from one register to another.
- 2. Arithmetic microoperations perform arithmetic operations on numeric data stored in registers.
- 3. Logic microoperations perform bit manipulation operations on non-numeric data stored in registers.
- 4. Shift microoperations perform shift operations on data stored in registers

# Arithmetic Microoperations

- The basic arithmetic microoperations are addition, subtraction, increment, decrement, and shift. Arithmetic shifts are explained later in conjunction with the shift microoperations. The arithmetic microoperation defined by the statement
  - $R3 \leftarrow R1 + R2$
- The above microoperation specifies an *add* microoperation. It states that the contents of register R1 are added to the contents of register R2 and the sum transferred to register R3. To implement this statement with hardware we need three registers and the digital component that performs the addition operation. The other basic arithmetic microoperations are listed in Table 3. Subtraction is most often implemented through complementation and addition. Instead of using the minus operator, we can specify the subtraction by the following statement:
  - $R3 \leftarrow R1 + R2 + 1$

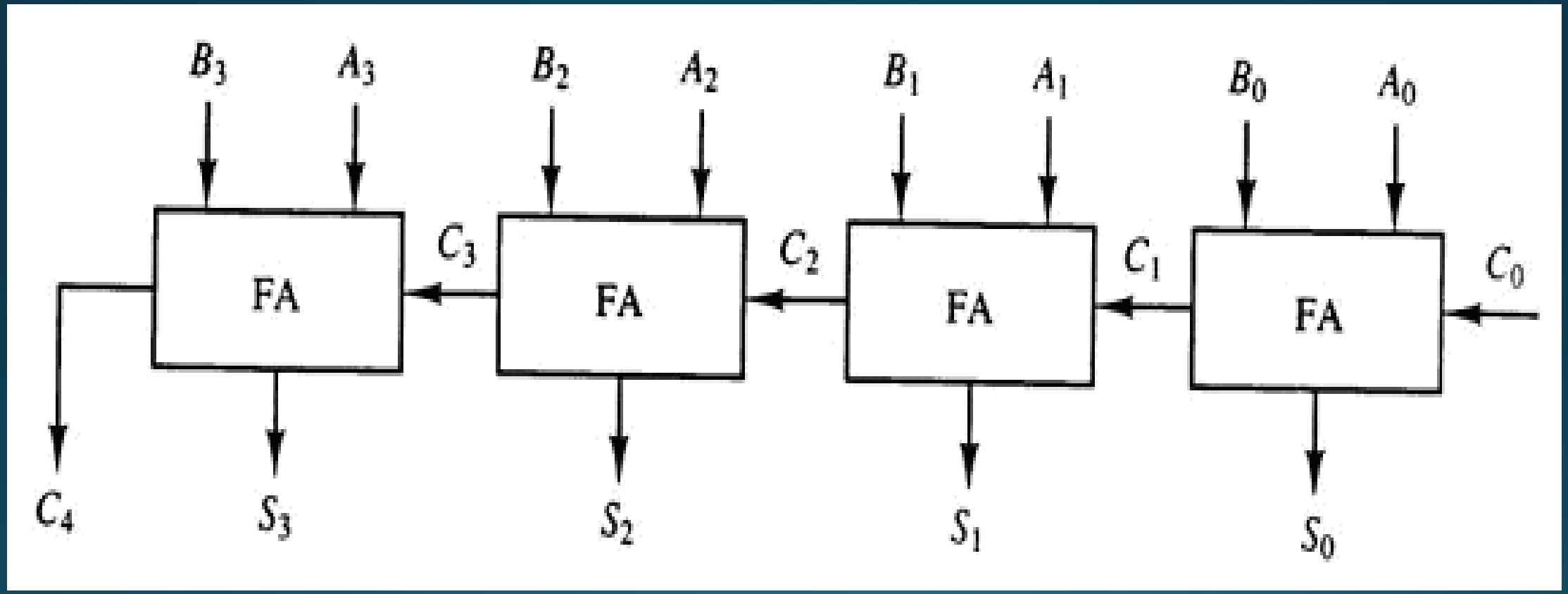
# List of Microoperations

| 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*

- The digital circuit that generates the arithmetic sum of two binary numbers of any lengths is called a binary adder. The binary adder is constructed with full-adder circuits connected in cascade, with the output carry from one full-adder connected to the input carry of the next full-adder. The following diagram shows interconnections of four full-adders (FA) to provide a 4-bit binary adder. The augend bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the low-order bit. The carries are connected in a chain through the full-adders. The input carry to the binary adder is $C_0$ and the output carry is $C_4$. The S outputs of the full-adders generate the required sum bits

# Diagram of Binary Adder



# Binary Incrementer

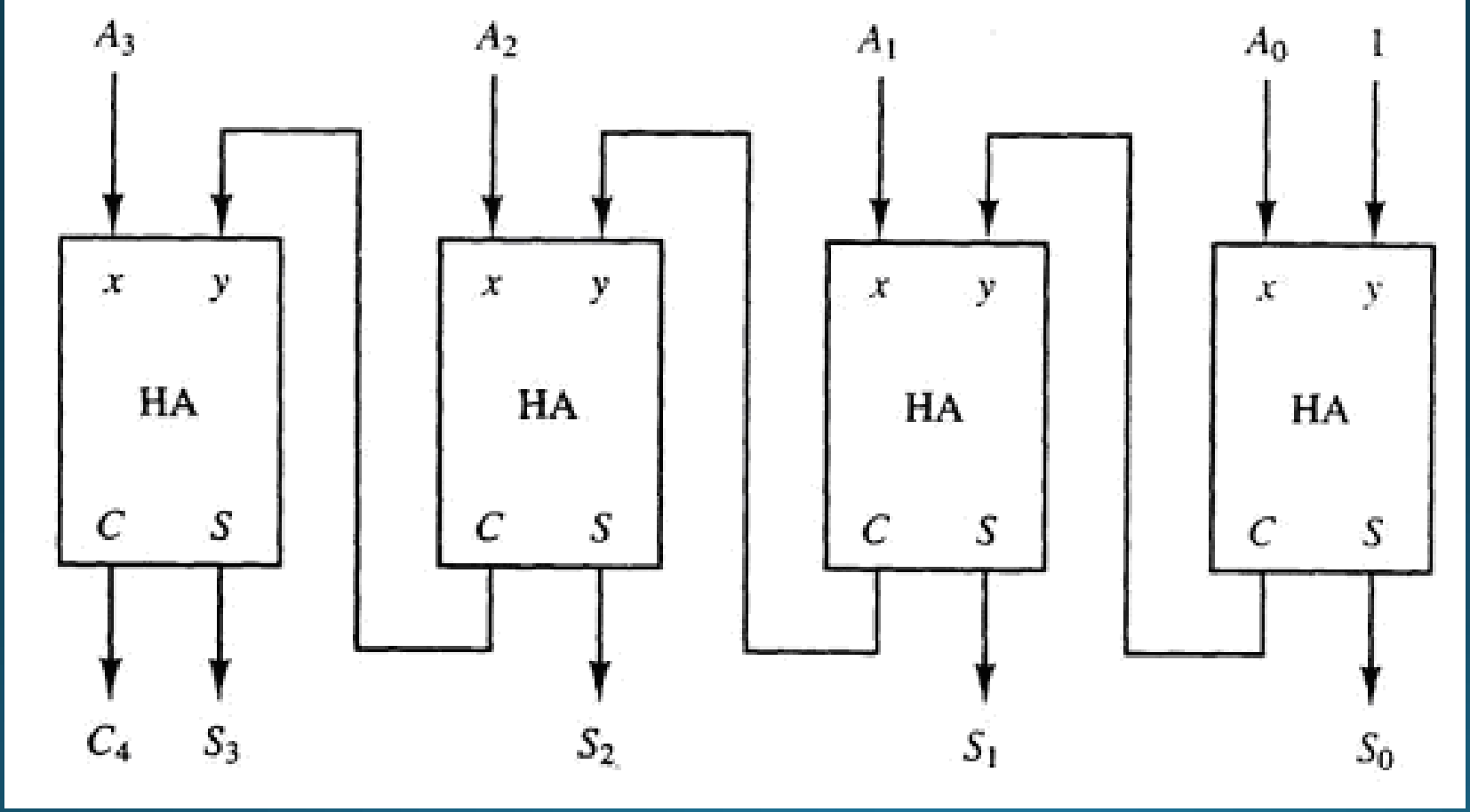
:

- The increment microoperation adds one to a number in a register. For example, if a 4-bit register has a binary value 0110, it will go to 0111 after it is incremented. This microoperation is easily implemented with a binary counter. Every time the count enable is active, the clock pulse transition increments the content of the register by one. There may be occasions when the increment microoperation must be done with a combinational circuit independent of a particular register. This can be accomplished by means of half-adders connected in cascade.

# Binary Incrementer

- The diagram of a 4-bit combinational circuit, incrementer is shown in the following diagram. One of the inputs to the least significant half-adder (HA) is connected to logic-1 and the other input is connected to the least significant bit of the number to be incremented. The output carry from one half-adder is connected to one of the inputs of the next-higher-order half-adder. The circuit receives the four bits from $A_0$ through $A_3$, adds one to it, and generates the incremented output in $S_0$ through $S_3$. The output carry $C_4$ will be 1 only after incrementing binary 1111. This also causes outputs $S_0$ through $S_3$ to go to 0.

# Diagram of Binary Incrementer



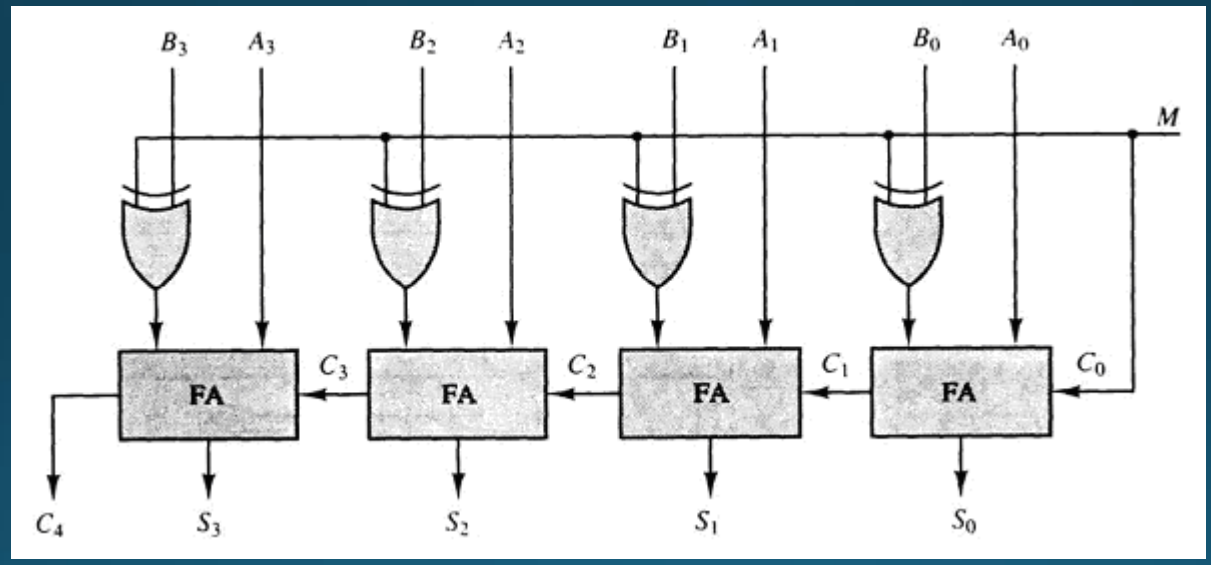
# *Binary Adder/Subtractor*

- The subtraction of binary numbers can be done most conveniently by means of complements. Remember that the subtraction, A – B can be done by taking the 2's complement of B and adding it to A. The 2's complement can be obtained by taking the 1's complement and adding one to the least significant pair of bits. The 1's complement can be implemented with inverters and a one can be added to the sum through the input carry.

# Binary Adder/Subtractor

- The addition and subtraction operations can be combined into one common circuit by including an exclusive-OR gate with each full-adder. A 4-bit adder-subtractor circuit is shown in Fig.6. The mode input M controls the operation. When M = 0 the circuit is an adder and when M = 1 the circuit becomes a subtractor. Each exclusive-OR gate receives input M and one of the inputs of B. When M = 0, we have $B \oplus 0 = B$. The full-adders receive the value of B, the input carry is 0, and the circuit performs A plus B. When M = 1, we have $B \oplus 1 = B'$ and $C_0 = 1$. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2's complement of B

# Binary Adder/Subtractor



# Arithmetic Circuit

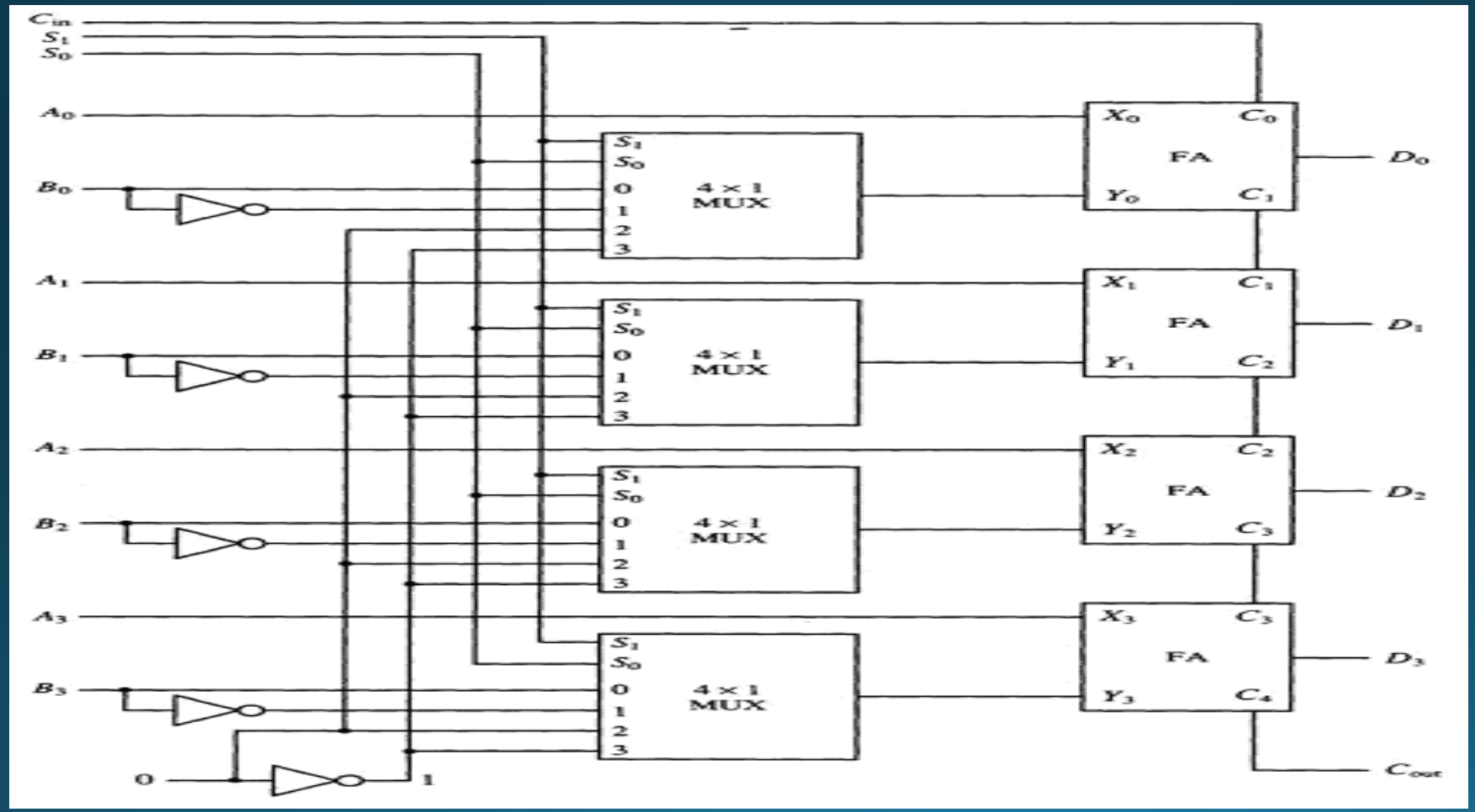
- The arithmetic microoperations listed in the above Table can be implemented in one composite arithmetic circuit. The basic component of an arithmetic circuit is the parallel adder. By controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations.

- The diagram of a 4-bit arithmetic circuit as shown in the following diagram. It has four full-adder circuits that constitute the 4-bit adder and four multiplexers for choosing different operations. There are two 4-bit inputs A and B and a 4-bit output D. The four inputs from A go directly to the X inputs of the binary adder. Each of the four inputs from B is connected to the data inputs of the multiplexers.

# *Arithmetic Circuit*

- The multiplexers data inputs also receive the complement of B. The other two data inputs are connected to logic-0 and logic-1. Logic-0 is a fixed voltage value (0 volts for TTL integrated circuits) and the logic-1 signal can be generated through an inverter whose input is 0. The four multiplexers are controlled by two selection inputs, $S_1$ and $S_0$. The input carry $C_{in}$ goes to the carry input of the FA in the least significant position. The other carries are connected from one stage to the next.

# Diagram of Arithmetic Circuit



# Arithmetic Circuit

- The output of the binary adder is calculated from the following arithmetic sum: $D = A + Y + C_{in}$, where A is the 4-bit binary number at the X inputs and Y is the 4-bit binary number at the Y inputs of the binary adder. $C_{in}$ is the input carry, which can be equal to 0 or 1. Note that the symbol + in the equation above denotes an arithmetic plus. By controlling the value of Y with the two selection inputs $S_1$ and $S_0$ and making $C_{in}$ equal to 0 or 1, it is possible to generate the eight arithmetic microoperations listed in the following table.

# Arithmetic Circuit

| 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 $A$ |

# Arithmetic Circuit

- When $S_1S_0$ = 00, the value of B is applied to the Y inputs of the adder.If $C_{in}$ = 0, the output D = A + B. If $C_{in}$ = 1, output D = A + B + 1. Both cases perform the add microoperation with or without adding the input carry.

- When $S_1S_0$ = 01, the complement of B is applied to the Y inputs of the adder. If $C_{in}$ = 1, then D = A + B + 1. This produces A plus the 2's complement of B, which is equivalent to a subtraction of A – B. When $C_{in}$ = 0, then D = A + B. This is equivalent to a subtract with borrow, that is, A – B – 1.

# Arithmetic Circuit

- When $S_1S_0$ = 10, the inputs from B are neglected, and instead, all 0's are inserted into the Y inputs. The output becomes D = A + 0 + $C_{in}$. This gives D = A when $C_{in}$ = 0 and D = A + 1 when $C_{in}$ = 1. In the first case we have a direct transfer from input A to output D. In the second case, the value of A is incremented by 1.

- When $S_1S_0$ = 11, all 1's are inserted into the Y inputs of the adder to produce the decrement operation D = A – 1 when $C_{in}$ = 0. This is because a number with all 1's is equal to the 2's complement of 1 (the 2's complement of binary 0001 is 1111). Adding a number A to the 2's complement of 1 produces F = A + 2's complement of 1 = A – 1. When $C_{in}$ = 1, then D = A – 1 + 1 = A, which causes a direct transfer from input A to output D. Note that the microoperation D = A is generated twice, so there are only seven distinct microoperations in the arithmetic circuit.

# Logic Microoperations

- ***Logic Microoperations*:**
- Logic microoperations specify binary operations for strings of bits stored in registers. These operations consider each bit of the register separately and treat them as binary variables. For example, the exclusive-OR microoperation with the contents of two registers R1 and R2 is symbolized by the statement
  - P: R1 ← R1 ⊕ R2
- It specifies a logic microoperation to be executed on the individual bits of the registers provided that the control variable P = 1. As a numerical example, assume that each register has four bits. Let the content of R1 be 1010 and the content of R2 be 1100. The exclusive-OR microoperation stated above symbolizes the following logic computation:

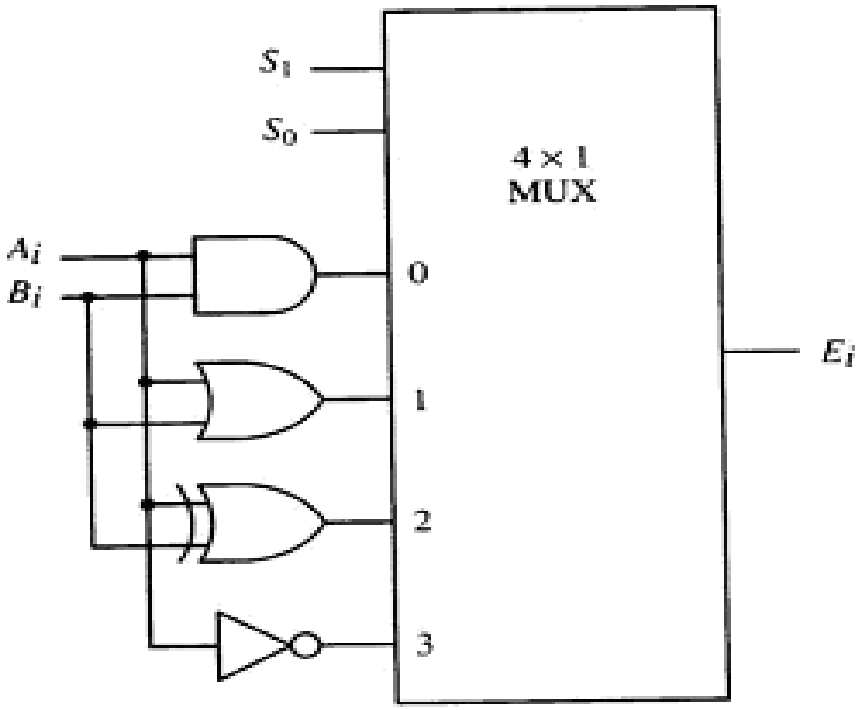
# Logic Microoperations

- Special symbols will be adopted for the logic microoperations OR, AND, and complement, to distinguish them from the corresponding symbols used to express Boolean functions. The symbol V will be used to denote an OR microoperation and the symbol ∧ to denote an AND microoperation. The complement microoperation is the same as the 1's complement and uses a bar on top of the symbol that denotes the register name. By using different symbols, it will be possible to differentiate between a logic microoperation and a control (or Boolean) function. Another reason for adopting two sets of symbols is to be able to distinguish the symbol +, when used to symbolize an arithmetic plus, from a logic OR operation. Although the + symbol has two meanings, it will be possible to distinguish between them by noting where the symbol occurs. When the symbol + occurs in a microoperation, it will denote an arithmetic plus. When it occurs in a control (or Boolean) function, it will denote an OR operation. We will never use it to symbolize an OR microoperation.

# Hardware Implementation

- **Hardware Implementation**
- 
- The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers to perform the required logic function. Although there are 16 logic microoperations, most computers use only four – AND, OR, XOR (exclusive-OR), and complement –from which all others can be derived.
- The following diagram shows one stage of a circuit that generates the four basic logic microoperations. It consists of four gates and a multiplexer. Each of the four logic operations is generated through a gate that performs the required logic. The outputs of the gates are applied to the data inputs of the multiplexer. The two selection inputs $S_1$ and $S_0$ choose one of the data inputs of the multiplexer and direct its value to the output. The diagram shows one typical stage with subscript *i*. For a logic circuit with *n* bits, the diagram must be repeated *n* times for *i = 0, 1, 2, ..., n-1*. The selection variables are applied to all stages. The function table in Fig.9 (b) lists the logic microoperations obtained for each combination of the selection variables.

# LOGIC MICROOPERATIONS




(a) Logic diagram

| $S_1$ | $S_0$ | Output | Operation |
|---|---|---|---|
| 0 | 0 | $E = A \wedge B$ | AND |
| 0 | 1 | $E = A \vee B$ | OR |
| 1 | 0 | $E = A \oplus B$ | XOR |
| 1 | 1 | $E = \overline{A}$ | Complement |

(b) Function table

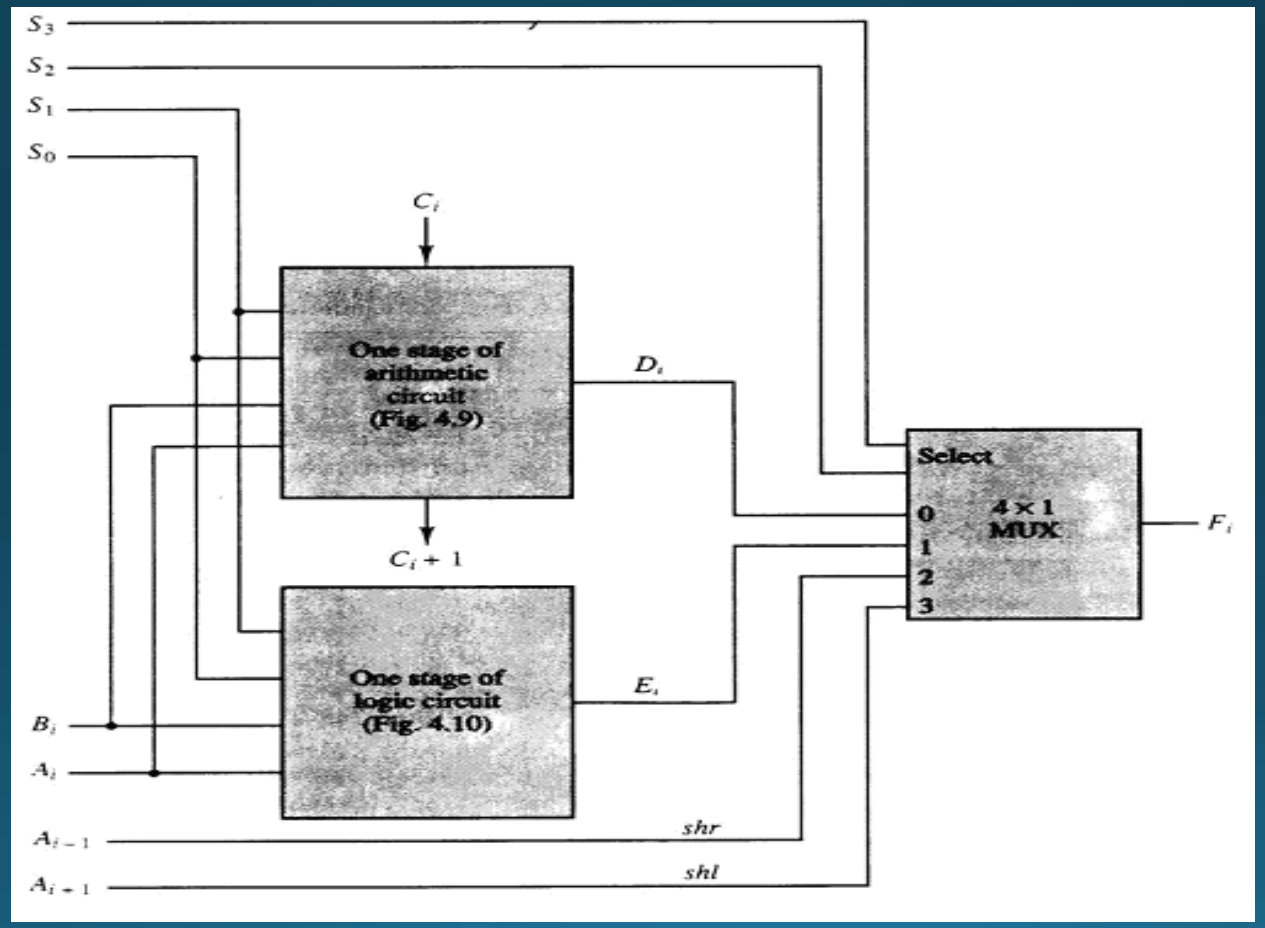
# Arithmetic Logic Shift Unit

- Instead of having individual registers performing the microoperations directly, computer systems employ a number of storage registers connected to a common operational unit called an arithmetic logic unit, abbreviated ALU. To perform a microoperation, the contents of specified registers are placed in the inputs of the common ALU. The ALU performs an operation and the result of the operation is then transferred to a destination register. The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period. The shift microoperations are often performed in a separate unit, but sometimes the shift unit is made part of the overall ALU.

# Arithmetic Logic Shift Unit

- The arithmetic, logic, and shift circuits can be combined into one ALU with common selection variables. One stage of an arithmetic logic shift unit is as shown in the following diagram. The subscript *i* designates a typical stage. Inputs A*i* and B*i* are applied to both the arithmetic and logic units. A particular microoperation is selected with inputs $S_1$ and $S_0$. A 4 x 1 multiplexer at the output chooses between an arithmetic output in $E_i$ and a logic output in $H_i$. The data in the multiplexer are selected with inputs $S_3$ and $S_2$. The other two data inputs to the multiplexer receive inputs $A_{i-1}$ for the shift-right operation and $A_{i+1}$ for the shift-left operation. Note that the diagram shows just one typical stage. The circuit of Fig.12 must be repeated *n* times for an *n*-bit ALU. The output-carry $C_{i+1}$ of a given arithmetic stage must be connected to the input carry $C_i$ of the next stage in sequence. The input carry to the first stage is the input carry $C_{in}$, which provides a selection variable for the arithmetic operations.

# Arithmetic Logic Shift Unit



# List of Operations of ALSU

| 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 | × | $F = A \wedge B$ | AND |
| 0 | 1 | 0 | 1 | × | $F = A \vee B$ | OR |
| 0 | 1 | 1 | 0 | × | $F = A \oplus B$ | XOR |
| 0 | 1 | 1 | 1 | × | $F = \overline{A}$ | Complement $A$ |
| 1 | 0 | × | × | × | $F =$ shr $A$ | Shift right $A$ into $F$ |
| 1 | 1 | × | × | × | $F =$ shl $A$ | Shift left $A$ into $F$ |