COMPUTER ARCHITECTURE

UNIT I

1) Number Systems

- **A number system of base**, or radix, r is a system that uses distinct symbols for r digits. Numbers are represented by a string of digit symbols.
- **To determine the quantity that the number represents**, it is necessary to multiply each digit by an integer power of r and then form the sum of all weighted digits.
- **For example**, the decimal number system in everyday use employs the radix 10 system. The 10 symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The string of digits 724.5 is interpreted to represent the quantity 7∗102+2∗101+4∗100+5∗10−1
- **that is, 7 hundreds, plus 2 tens, plus 4 units, plus 5 tenths**. Every decimal number can be similarly interpreted to find the quantity it represents. The binary number system uses the radix 2. The two digit symbols used are 0 and 1.
- **The string of digits** 101101 is interpreted to represent the quantity 1∗25+0∗24+1+23+1∗22+0∗21+1∗20=45
- **To distinguish between different radix numbers**, the digits will be enclosed in parentheses and the radix of the number inserted as a subscript.
- **For example,** to show the equality between decimal and binary forty-five we will write $(101101)_2 = (45)_{10}$.
- **Besides the decimal and binary number systems**, the octal (radix 8) and hexadecimal (radix 16) are important in digital computer work.
- **The eight symbols of the octal system** are 0, 1, 2, 3, 4, 5, 6, and 7. The 16 symbols of the hexadecimal system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, 0, E, and F.
- **The last six symbols are, unfortunately**, identical to the letters of the alphabet and can cause confusion at times.
- **However, this is the convention that has been adopted**. When used to represent hexadecimal digits, the symbols A, B, C, D, E, F correspond to the decimal numbers 10, 11, 12, 13, 14, 15, respectively.
- **A number in radix r can be converted to the familiar decimal system** by forming the sum of the weighted digits. For example, octal 736.4 is converted to decimal as follows: (736.4)8=7∗82+3∗81+6∗80+4∗8−1=7∗64+3∗8+6∗1+4/8=(478.5)10
- **The equivalent decimal number** of hexadecimal F3 is obtained from the following calculation: (F3)16=F∗16+3=15∗16+3=(243)10
- **Conversion from decimal to its equivalent representation** in the radix r system is carried out by separating the number into its integer and fraction parts and converting each part separately. The conversion of a decimal integer into a base r representation is done by successive divisions by r and accumulation of the remainders.

- **The conversion of a decimal fraction** to radix r representation is accomplished by successive multiplications by r and accumulation of the integer digits so obtained. Figure below demonstrates these procedures.

Figure Conversion of decimal 41.6875 into binary.

Integer = 41

41	
20	1
10	0
5	0
2	1
1	0
0	1

$(41)_{10} = (101001)_2$

Fraction = 0.6875

$$\begin{array}{r} 0.6875 \\ \underline{\ 2} \\ 1.3750 \\ \underline{x\ 2} \\ 0.7500 \\ \underline{x\ 2} \\ 1.5000 \\ \underline{x\ 2} \\ 1.0000 \end{array}$$

$(0.6875)_{10} = (0.1011)_2$

$(41.6875)_{10} = (101001.1011)_2$

- **The conversion of decimal 41.6875** into binary is done by first separating the number into its integer part 41 and fraction part .6875.
- **The integer part is converted by dividing 41 by r = 2** to give an integer quotient of 20 and a remainder of 1 The quotient is again divided by 2 to give a new quotient and remainder.
- **This process** is repeated until the integer quotient becomes 0. The coefficients of the binary number are obtained from the remainders with the first remainder giving the low-order bit of the converted binary number.
- **The fraction part** is converted by multiplying it by r = 2 to give an integer and a fraction. The new fraction (without the integer) is multiplied again by 2 to give a new integer and a new fraction.
- **This process** is repeated until the fraction part becomes zero or until the number of digits obtained gives the required accuracy.
- **The coefficients of the binary fraction** are obtained from the integer digits with the first integer computed being the digit to be placed next to the binary point.
- **Finally**, the two parts are combined to give the total required conversion.

2) Octal and Hexadecimal Numbers

- **The conversion from and to binary, octal, and hexadecimal** representation plays an important part in digital computers.
- **Since 2^3 = 8 and 2^4 = 16**, each octal digit corresponds to three binary digits and each hexadecimal digit corresponds to four binary digits.
- **The conversion from binary to octal** is easily accomplished by partitioning the binary number into groups of three bits each. The corresponding octal digit is then assigned to each group of bits and the string of digits so obtained gives the octal equivalent of the binary number.
- **Starting from the low-order bit**, we partition the register into groups of three bits each (the sixteenth bit remains in a group by itself).
- **Each group of three bits** is assigned its octal equivalent and placed on top of the register. The string of octal digits so obtained represents the octal equivalent of the binary number.

- **Conversion from binary to hexadecimal** is similar except that the bits are divided into groups of four.

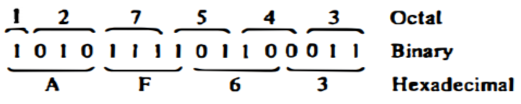


Figure Binary, octal, and hexadecimal conversion.

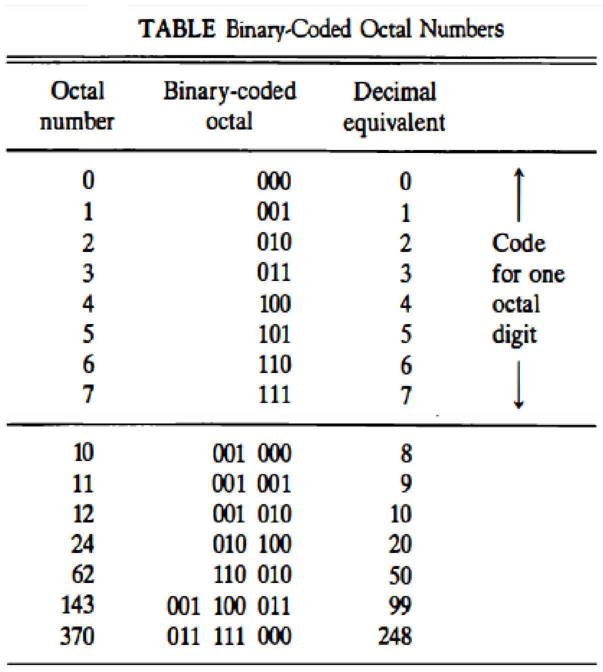
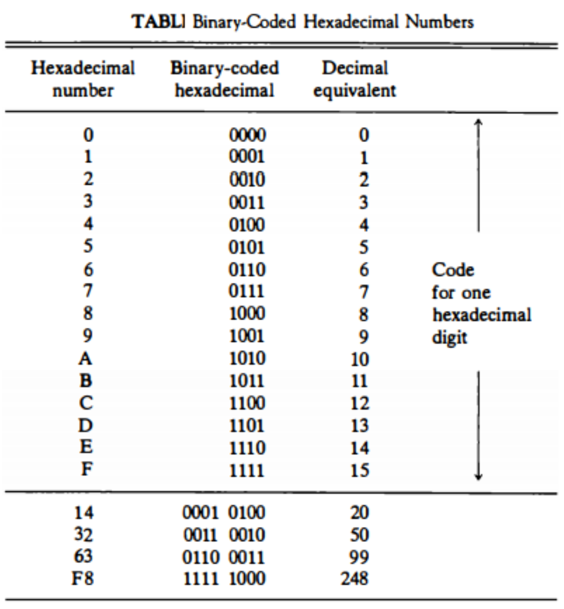


TABLE Binary-Coded Octal Numbers

Octal number	Binary-coded octal	Decimal equivalent	
0	000	0	↑
1	001	1	
2	010	2	Code
3	011	3	for one
4	100	4	octal
5	101	5	digit
6	110	6	
7	111	7	↓
10	001 000	8	
11	001 001	9	
12	001 010	10	
24	010 100	20	
62	110 010	50	
143	001 100 011	99	
370	011 111 000	248	



TABLI Binary-Coded Hexadecimal Numbers

Hexadecimal number	Binary-coded hexadecimal	Decimal equivalent	
0	0000	0	↑
1	0001	1	
2	0010	2	
3	0011	3	
4	0100	4	
5	0101	5	
6	0110	6	Code
7	0111	7	for one
8	1000	8	hexadecimal
9	1001	9	digit
A	1010	10	
B	1011	11	
C	1100	12	
D	1101	13	
E	1110	14	
F	1111	15	↓
14	0001 0100	20	
32	0011 0010	50	
63	0110 0011	99	
F8	1111 1000	248	

- **The registers in a digital computer** contain many bits. Specifying the content of registers by their binary values will require a long string of binary digits.
- **It is more convenient** to specify content of registers by their octal or hexadecimal equivalent.
- **The number of digits** is reduced by one-third in the octal designation and by one-fourth in the hexadecimal designation. For example, the binary number 1111 1111 1111 has 12 digits.
- **It can be expressed** in octals as 7777 (four digits) or in hexadecimal as FFF (three digits).

- **Computer manuals invariably** choose either the octal or the hexadecimal designation for specifying contents of registers.

3) Decimal Representation

- **The binary number system** is the most natural system for a computer, but people are accustomed to the decimal system.
- **One way to solve this conflict is to convert all input decimal numbers into binary numbers**, let the computer perform all arithmetic operations in binary and then convert the binary results back to decimal for the human user to understand.
- **However**, it is also possible for the computer to perform arithmetic operations directly with decimal numbers provided they are placed in registers in a coded form.
- **Decimal numbers enter the computer** usually as binary-coded alphanumeric characters. These codes, introduced later, may contain from six to eight bits for each decimal digit. When decimal numbers are used for internal arithmetic computations, they are converted to a binary code with four bits per digit.
- **A binary code is a group of n bits** that assume up to 2^n distinct combinations of 1's and 0's with each combination representing one element of the set that is being coded.
- **A binary code** will have some unassigned bit combinations if the number of elements in the set is not a multiple power of 2. The 10 decimal digits form such a set. A binary code that distinguishes among 10 elements must contain at least four bits, but six combinations will remain unassigned.
- **Numerous different codes** can be obtained by arranging four bits in 10 distinct combinations. The bit assignment most commonly used for the decimal digits is the straight binary assignment listed in the first 10 entries of Table below.
- **This particular code** is called binary-coded decimal and is commonly referred to by its abbreviation BCD. **It is very important** to understand the difference between the conversion of decimal numbers into binary and the binary coding of decimal numbers.
- **For example**, when converted to a binary number, the decimal number 99 is represented by the string of bits 1100011, but when represented in BCD, it becomes 1001 1001 .
- **The only difference** between a decimal number represented by the familiar digit symbols 0, 1, 2, ... , 9 and the BCD symbols 0001, 0010, ... , 1001 is in the symbols used to represent the digits-the number itself is exactly the same.

TABLE Binary-Coded Decimal (BCD) Numbers

Decimal number	Binary-coded decimal (BCD) number	
0	0000	↑
1	0001	
2	0010	
3	0011	Code
4	0100	for one
5	0101	decimal
6	0110	digit
7	0111	
8	1000	
9	1001	↓
10	0001 0000	
20	0010 0000	
50	0101 0000	
99	1001 1001	
248	0010 0100 1000	

4) Alphanumeric Representation

- **Many applications of digital computers** require the handling of data that consist not only of numbers, but also of the letters of the alphabet and certain special characters.
- **An alphanumeric character set** is a set of elements that includes the 10 decimal digits, the 26 letters of the alphabet and a number of special characters, such as $, + , and =.
- **Such a set contains between 32 and 64 elements** (if only uppercase letters are included) or between 64 and 128 (if both uppercase and lowercase letters are included).
- **In the first case**, the binary code will require six bits and in the second case, seven bits. The standard alphanumeric binary code is the ASCII (American Standard Code for Information Interchange), which uses seven bits to code 128 characters.
- **The binary code** for the uppercase letters, the decimal digits, and a few special characters is listed in Table below.
- **Note that the decimal digits in ASCII** can be converted to BCD by removing the three high-order bits, 011.
- **Binary codes play an important part in digital computer operations.** The codes must be in binary because registers can only hold binary information. One must realize that binary codes merely change the symbols, not the meaning of the discrete elements they represent.
- **The operations** specified for digital computers must take into consideration the meaning of the bits stored in registers so that operations are performed on operands of the same type.

TABLE American Standard Code for Information Interchange (ASCII)

Character	Binary code	Character	Binary code
A	100 0001	0	011 0000
B	100 0010	1	011 0001
C	100 0011	2	011 0010
D	100 0100	3	011 0011
E	100 0101	4	011 0100
F	100 0110	5	011 0101
G	100 0111	6	011 0110
H	100 1000	7	011 0111
I	100 1001	8	011 1000
J	100 1010	9	011 1001
K	100 1011		
L	100 1100		
M	100 1101	space	010 0000
N	100 1110	.	010 1110
O	100 1111	(	010 1000
P	101 0000	+	010 1011
Q	101 0001	$	010 0100
R	101 0010	*	010 1010
S	101 0011	)	010 1001
T	101 0100	−	010 1101
U	101 0101	/	010 1111
V	101 0110	,	010 1100
W	101 0111	=	011 1101
X	101 1000		
Y	101 1001		
Z	101 1010		

5) Complements: 9s complement

- **Complements are used in digital computers** for simplifying the subtraction operation and for logical manipulation. There are two types of complements for each base r system: the r's complement and the (r - 1)'s complement
- **When the value of the base r** is substituted in the name, the two types are referred to as the 2's and 1's complement for binary numbers and the 10's and 9's complement for decimal numbers.
- **(r - 1)'s Complement**

1. **Given a number N** in base r having n digits, the (r - 1)'s complement of N is defined as $(r^n - 1) - N$. For decimal numbers r = 10 and r - 1 = 9, so the 9's complement of N is $(10^n - 1) - N$.
2. **Now**, 10^n represents a number that consists of a single 1 followed by n 0's.
3. **$10^n - 1$** is a number represented by n 9's. For example, with n = 4 we have $10^4 = 10000$ and $10^4 - 1 = 9999$.
4. **It follows** that the 9's complement of a decimal number is obtained by subtracting each digit from 9.
5. **For example**, the 9's complement of 546700 is 999999 - 546700 = 453299 and the 9's complement of 12389 is 99999 - 12389 = 87610.

6) Complements: 1's complement

- **For binary numbers**, r = 2 and r - 1 = 1, so the 1's complement of N is $(2^n - 1) - N$. Again, 2^n is represented by a binary number that consists of a 1 followed by n 0's. $2^n - 1$ is a binary number represented by n 1's.
- **For example,** with n = 4, we have $2^4 = (10000)_2$ and $2^4 - 1 = (1111)_2$. Thus the 1's complement of a binary number is obtained by subtracting each digit from 1.
- **However,** the subtraction of a binary digit from 1 causes the bit to change from 0 to 1 or from 1 to 0. Therefore, the 1's complement of a binary number is formed by changing 1's into 0's and 0's into 1's.
- **For example**, the 1's complement of 1011001 is 0100110 and the 1's complement of 0001111 is 1110000.
- **The (r - 1)'s complement** of octal or hexadecimal numbers are obtained by subtracting each digit from 7 or F (decimal 15) respectively.

7) (r's) Complement: 10's complement

- **The r's complement** of an n-digit number N in base r is defined as $r^n - N$ for $N \neq 0$ and 0 for N = 0.
- **Comparing** with the (r - 1)'s complement, we note that the r's complement is obtained by adding 1 to the (r - 1)'s complement since $r^n - N = [(r^n - 1) - N] + 1$.
- **Thus the 10's complement** of the decimal 2389 is 7610 + 1 = 7611 and is obtained by adding 1 to the 9' s complement value.
- **The 2's complement** of binary 101100 is 010011 + 1 = 010100 and is obtained by adding 1 to the 1's complement value.
- **Since 10^n** is a number represented by a 1 followed by n 0's, then $10^n - N$, which is the 10's complement of N, can be formed also be leaving all least significant 0's unchanged, subtracting the first nonzero least significant digit from 10, and then subtracting all higher significant digits from 9.
- **The 10's complement** of 246700 is 753300 and is obtained by leaving the two zeros unchanged, subtracting 7 from 10, and subtracting the other three digits from 9.

8) (r's) Complement: 2's complement

- **The 2's complement** can be formed by leaving all least significant 0's and the first 1 unchanged, and then replacing 1's by 0's and 0's by 1's in all other higher significant bits.
- **The 2's complement** of 1101100 is 0010100 and is obtained by leaving the two low-order 0's and the first 1 unchanged, and then replacing 1's by 0's and 0's by 1's in the other four most significant bits.
- **In the definitions** above it was assumed that the numbers do not have a radix point.

- **If the original number N** contains a radix point, it should be removed temporarily to form the r's or (r - 1)'s complement.
- **The radix point** is then restored to the complemented number in the same relative position.
- **It is also worth** mentioning that the complement of the complement restores the number to its original value.
- **The r's** complement of N is r^n - N.
- **The complement** of the complement is $r^n - (r^n - N) = N$ giving back the original number.

9) Subtraction of Unsigned Numbers

- **The subtraction of two n-digit unsigned numbers** M - N (N ≠ 0) in base r can be done as follows:
  1. **Add the minuend M to the r's complement** of the subtrahend N. This performs $M + (r^n - N) = M - N + r^n$.
  2. **If M ≥ N**, the sum will produce an end carry r^n which is discarded, and what is left is the result M - N.
  3. **If M < N,** the sum does not produce an end carry and is equal to $r^n - (N - M)$, which is the r's complement of (N - M). To obtain the answer in a familiar form, take the r' s complement of the sum and place a negative sign in front.
- **Consider**, for example, the subtraction 72532 - 13250 = 59282. The 10's complement of 13250 is 86750. Therefore:

$$\begin{array}{rr} M = & 72532 \\ \text{10's complement of } N = & +\underline{86750} \\ \text{Sum} = & 159282 \\ \text{Discard end carry } 10^5 = & -\underline{100000} \\ \text{Answer} = & 59282 \end{array}$$

- **Now consider an example** with M < N. The subtraction 13250 - 72532 produces negative 59282. Using the procedure with complements, we have

$$\begin{array}{rr} M = & 13250 \\ \text{10's complement of } N = & +\underline{27468} \\ \text{Sum} = & 40718 \end{array}$$

- **There is no end carry**. Answer is negative 59282 = 10's complement of 40718
- **Since we are dealing with unsigned numbers**, there is really no way to get an unsigned result for the second example.
- **When subtracting with complements**, the negative answer is recognized by the absence of the end carry and the complemented result.
- **Subtraction with complements** is done with binary numbers in a similar manner using the same procedure outlined above.
- **Using the two binary numbers** X = 1010100 and Y = 1000011, we perform the subtraction X - Y and Y - X using 2's complements:

$$\begin{array}{rr} X = & 1010100 \\ \text{2's complement of } Y = & +\underline{0111101} \\ \text{Sum} = & 10010001 \\ \text{Discard end carry } 2^7 = & -\underline{10000000} \\ \text{Answer: } X - Y = & 0010001 \end{array}$$

$$\begin{array}{rr} Y = & 1000011 \\ \text{2's complement of } X = & +\underline{0101100} \\ \text{Sum} = & 1101111 \end{array}$$

- **There is no end carry**. Answer is negative 0010001 = 2's complement of 1101111

10) Fixed-Point Representation

- **Positive integers**, including zero, can be represented as unsigned numbers. However, to represent negative integers, we need a notation for negative values.
- **In ordinary arithmetic**, a negative number is indicated by a minus sign and a positive number by a plus sign. Because of hardware limitations, computers must represent everything with 1's and 0's, including the sign of a number.
- **As a consequence**, it is customary to represent the sign with a bit placed in the leftmost position of the number. The convention is to make the sign bit equal to 0 for positive and to 1 for negative.
- **In addition to the sign**, a number may have a binary (or decimal) point. The position of the binary point is needed to represent fractions, integers, or mixed integer-fraction numbers.
- **The representation of the binary point** in a register is complicated by the fact that it is characterized by a position in the register.
- **There are two ways of specifying the position of the binary point** in a register: by giving it a fixed position or by employing a floating-point representation.
- **The fixed-point method** assumes that the binary point is always fixed in one position. The two positions most widely used are (1) a binary point in the extreme left of the register to make the stored number a fraction, and (2) a binary point in the extreme right of the register to make the stored number an integer.
- **In either case**, the binary point is not actually present, but its presence is assumed from the fact that the number stored in the register is treated as a fraction or as an integer.
- **The floating-point representation** uses a second register to store a number that designates the position of the decimal point in the first register.

11) Integer Representation

- **When an integer binary number** is positive, the sign is represented by 0 and the magnitude by a positive binary number. When the number is negative, the sign is represented by 1 but the rest of the number may be represented in one of three possible ways:

- 1. Signed-magnitude representation
- 2. Signed-1's complement representation
- 3. Signed 2's complement representation

- **The signed-magnitude representation** of a negative number consists of the magnitude and a negative sign. In the other two representations, the negative number is represented in either the 1's or 2's complement of its positive value.
- **As an example**, consider the signed number 14 stored in an 8-bit register. + 14 is represented by a sign bit of 0 in the leftmost position followed by the binary equivalent of 14: 00001110.
- **Note** that each of the eight bits of the register must have a value and therefore 0' s must be inserted in the most significant positions following the sign bit.
- **Although** there is only one way to represent + 14, there are three different ways to represent - 14 with eight bits.

In signed-magnitude representation 1 0001110

In signed-1's complement representation 1 1110001

In signed-2's complement representation 1 1110010

- **The signed-magnitude** representation of - 14 is obtained from + 14 by complementing only the sign bit.
- **The signed-1's complement representation of - 14** is obtained by complementing all the bits of + 14, including the sign bit. The signed-2' s complement representation is obtained by taking the 2' s complement of the positive number, including its sign bit.
- **The signed-magnitude system** is used in ordinary arithmetic but is awkward when employed in computer arithmetic.
- **Therefore**, the signed-complement is normally used. The 1' s complement imposes difficulties because it has two representations of 0 (+0 and - 0).
- **It is seldom used for arithmetic operations** except in some older computers.
- **The 1's complement** is useful as a logical operation since the change of 1 to 0 or 0 to 1 is equivalent to a logical complement operation.

12) Arithmetic Addition

- **The addition of two numbers in the signed-magnitude system** follows the rules of ordinary arithmetic.
- **If the signs are the same**, we add the two magnitudes and give the sum the common sign.
- **If the signs are different**, we subtract the smaller magnitude from the larger and give the result the sign of the larger magnitude.
- **For example**, (+25) + (-37) = - (37 - 25) = - 12 and is done by subtracting the smaller magnitude 25 from the larger magnitude 37 and using the sign of 37 for the sign of the result.
- **This is a process that requires the comparison of the signs** and the magnitudes and then performing either addition or subtraction.
- **By contrast, the rule for adding numbers** in the signed-2's complement system does not require a comparison or subtraction, only addition and complementation.
- **The procedure can be stated as follows:** Add the two numbers, including their sign bits, and discard any carry out of the sign (leftmost) bit position.
- **Numerical examples for addition are shown below**. Note that negative numbers must initially be in 2' s complement and that if the sum obtained after the addition is negative, it is in 2's complement form.

```
 +6   00000110      −6   11111010
+13   00001101     +13   00001101
---   --------     ---   --------
+19   00010011      +7   00000111

 +6   00000110      −6   11111010
−13   11110011     −13   11110011
---   --------     ---   --------
 −7   11111001     −19   11101101
```

- **In each of the four cases**, the operation performed is always addition, including the sign bits. Any carry out of the sign bit position is discarded, and negative results are automatically in 2' s complement form.
- **The complement form** of representing negative numbers is unfamiliar to people used to the signed-magnitude system.

- **To determine the value of a negative number** when in signed-2's complement, it is necessary to convert it to a positive number to place it in a more familiar form.
- **For example**, the signed binary number 1111 1001 is negative because the leftmost bit is 1.
- **Its 2' s complement is 00000111**, which is the binary equivalent of +7. We therefore recognize the original negative number to be equal to -7.

13) Arithmetic Subtraction

- **Subtraction of two signed binary numbers** when negative numbers are in 2' s complement form can be stated as follows:
- **Take the 2's complement of the subtrahend** (including the sign bit) and add it to the minuend (including the sign bit). A carry out of the sign bit position is discarded.
- **This procedure stems from the fact** that a subtraction operation can be changed to an addition operation if the sign of the subtrahend is changed. This is demonstrated by the following relationship:

$$(\pm A) - (+B) = (\pm A) + (-B)$$

$$(\pm A) - (-B) = (\pm A) + (+B)$$

- **But changing a positive number** to a negative number is easily done by taking its 2's complement.
- **The reverse** is also true because the complement of a negative number in complement form produces the equivalent positive number.
- **Consider the subtraction** of (-6) - (- 13) = +7. In binary with eight bits this is written as 11111010 - 11110011.
- **The subtraction is changed to addition** by taking the 2's complement of the subtrahend (- 13) to give (+ 13). In binary this is 11111010 + 00001101 = 100000111. Removing the end carry, we obtain the correct answer 00000111 (+ 7).
- **It is worth noting that binary numbers** in the signed-2's complement system are added and subtracted by the same basic addition and subtraction rules as unsigned numbers.
- **Therefore**, computers need only one common hardware circuit to handle both types of arithmetic.
- **The user or programmer** must interpret the results of such addition or subtraction differently depending on whether it is assumed that the numbers are signed or unsigned.

14) Overflow

- **When two numbers of n digits** each are added and the sum occupies n + 1 digits, we say that an overflow occurred.
- **An overflow is a problem in digital computers** because the width of registers is finite. A result that contains n + 1 bits cannot be accommodated in a register with a standard length of n bits. For this reason, many computers detect the occurrence of an overflow, and when it occurs, a corresponding flip-flop is set which can then be checked by the user.
- **The detection of an overflow** after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned. When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position.
- **In the case of signed numbers**, the leftmost bit always represents the sign, and negative numbers are in 2' s complement form.
- **When two signed numbers** are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow.

- **An overflow** cannot occur after an addition if one number is positive and the other is negative, since adding a positive number to a negative number produces a result that is smaller than the larger of the two original numbers.
- **An overflow may occur** if the two numbers added are both positive or both negative. To see how this can happen, consider the following example. Two signed binary numbers, + 70 and + 80, are stored in two 8-bit registers.
- **The range of numbers** that each register can accommodate is from binary + 127 to binary - 128. Since the sum of the two numbers is + I50, it exceeds the capacity of the 8-bit register.
- **This is true if the numbers** are both positive or both negative. The two additions in binary are shown below together with the last two carries.

```
carries: 0 1          carries: 1 0
     +70   0 1000110       −70   1 0111010
     +80   0 1010000       −80   1 0110000
    +150   1 0010110      −150   0 1101010
```

- **Note that the 8-bit result** that should have been positive has a negative sign bit and the 8-bit result that should have been negative has a positive sign bit. If, however, the carry out of the sign bit position is taken as the sign bit of the result, the 9-bit answer so obtained will be correct.
- **Since the answer cannot be accommodated within 8 bits**, we say that an overflow occurred. An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position. If these two carries are not equal, an overflow condition is produced.
- **This is indicated in the examples** where the two carries are explicitly shown. If the two carries are applied to an exclusive-OR gate, an overflow will be detected when the output of the gate is equal to 1.

15) Decimal Fixed-Point Representation

- **The representation** of decimal numbers in registers is a function of the binary code used to represent a decimal digit.
- **A 4-bit decimal code requires four flip-flops for each decimal digit**. The representation of 4385 in BCD requires 16 flip-flops, four flip-flops for each digit. The number will be represented in a register with 16 flip-flops as follows: 0100 0011 1000 0101
- **By representing numbers** in decimal we are wasting a considerable amount of storage space since the number of bits needed to store a decimal number in a binary code is greater than the number of bits needed for its equivalent binary representation.
- **Also, the circuits required to perform decimal arithmetic are more complex**. However, there are some advantages in the use of decimal representation because computer input and output data are generated by people who use the decimal system.
- **Some applications, such as business data processing**, require small amounts of arithmetic computations compared to the amount required for input and output of decimal data. For this reason, some computers and all electronic calculators perform arithmetic operations directly with the decimal data (in a binary code) and thus eliminate the need for conversion to binary and back to decimal.
- **Some computer systems** have hardware for arithmetic calculations with both binary and decimal data. The representation of signed decimal numbers in BCD is similar to the representation of signed numbers in binary.
- **We can either** use the familiar signed-magnitude system or the signed-complement system. The sign of a decimal number is usually represented with four bits to conform with the 4-bit code of the decimal digits.

- **It is customary** to designate a plus with four 0' s and a minus with the BCD equivalent of 9, which is 1001 .
- **The signed-magnitude system** is difficult to use with computers. The signed-complement system can be either the 9's or the 10's complement, but the 10's complement is the one most often used.
- **To obtain the 10' s complement** of a BCD number, we first take the 9's complement and then add one to the least significant digit. The 9' s complement is calculated from the subtraction of each digit from 9.
- **The procedures** developed for the signed-2's complement system apply also to the signed-10's complement system for decimal numbers. Addition is done by adding all digits, including the sign digit, and discarding the end carry.
- **Obviously**, this assumes that all negative numbers are in 10's complement form. Consider the addition (+375) + (-240) = + 135 done in the signed 10's complement system.

$$\begin{array}{rl} 0\ 375 & (0000\ 0011\ 0111\ 0101)_{BCD} \\ +\underline{9\ 760} & \underline{(1001\ 0111\ 0110\ 0000)}_{BCD} \\ 0\ 135 & (0000\ 0001\ 0011\ 0101)_{BCD} \end{array}$$

The 9 in the leftmost position of the second number indicates that the number is negative. 9760 is the 10's complement of 0240. The two numbers are added and the end carry is discarded to obtain + 135. Of course, the decimal numbers inside the computer must be in BCD, including the sign digits. The addition is done with BCD adders

The subtraction of decimal numbers either unsigned or in the signed-10' s complement system is the same as in the binary case. Take the 10' s complement of the subtrahend and add it to the minuend.

16) Floating-Point Representation

- **The floating-point representation** of a number has two parts. The first part represents a signed, fixed-point number called the mantissa.
- **The second part designates the position** of the decimal (or binary) point and is called the exponent. The fixed-point mantissa may be a fraction or an integer.
- **For example, the decimal number + 6132.789** is represented in floating-point with a fraction and an exponent as follows: Fraction : +0.6132789 ; Exponent : +04
- **The value of the exponent** indicates that the actual position of the decimal point is four positions to the right of the indicated decimal point in the fraction.
- **This representation** is equivalent to the scientific notation $+0.6132789 \times 10^{+4}$.
- **Floating-point** is always interpreted to represent a number in the following form: $m * r^{e}$
- **Only the mantissa m and the exponent e** are physically represented in the register (including their signs). The radix r and the radix-point position of the mantissa are always assumed. The circuits that manipulate the floating-point numbers in registers conform with these two assumptions in order to provide the correct computational results.
- **A floating-point binary number** is represented in a similar manner except that it uses base 2 for the exponent. For example, the binary number + 1001 .11 is represented with an 8-bit fraction and 6-bit exponent as follows: Fraction : 01001110 ; Exponent : 000100
- **The fraction** has a 0 in the leftmost position to denote positive. The binary point ofthe fraction follows the sign bit but is not shown in the register.
- **The exponent has the equivalent binary number +4**. The floating-point number is equivalent to m x $2^{e} = +(.1001110)^{2} \times 2^{+4}$

- **A floating-point number** is said to be normalized if the most significant digit of the mantissa is nonzero. For example, the decimal number 350 is normalized but 00035 is not.
- **Regardless of where the position** of the radix point is assumed to be in the mantissa, the number is normalized only if its leftmost digit is nonzero. For example, the 8-bit binary number 00011010 is not normalized because of the three leading 0's.
- **The number** can be normalized by shifting it three positions to the left and discarding the leading 0's to obtain 11010000.
- **The three shifts** multiply the number by $2^3 = 8$. To keep the same value for the floating-point number, the exponent must be subtracted by 3.
- **Normalized numbers** provide the maximum possible precision for the floating-point number. A zero cannot be normalized because it does not have a nonzero digit.
- **It is usually represented** in floating-point by all 0's in the mantissa and exponent.
- **Arithmetic operations** with floating-point numbers are more complicated than arithmetic operations with fixed-point numbers and their execution takes longer and requires more complex hardware.

17) Register Transfer Language

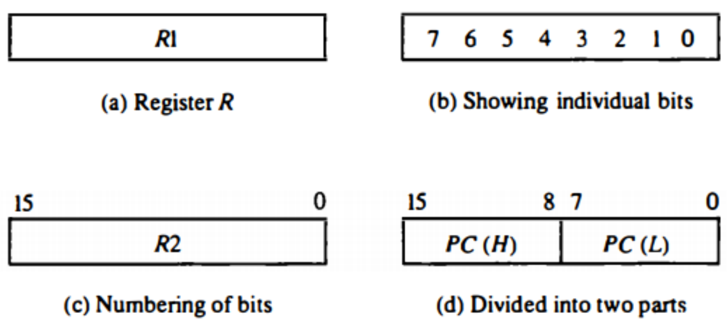
- **A digital system** is an interconnection of digital hardware module that accomplish a specific information-processing task.
- **Digital system** design invariably use a modular approach. The modules are constructed from such digital components as registers, decoders, arithmetic elements and control logic.
- **Digital modules** are best defined by the registers they contain and the operations that are performed on the data stored in them.
- **The operations** on data stored in registers are called micro-operations. 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 introduced are shift, count, clear, and load.
- **Registers implement micro-operations** a counter with parallel load is capable of performing the micro-operations increment and load. A bidirectional shift register is capable of performing the shift right and shift left microoperations.
- **The internal hardware organization** of a digital computer is best defined by specifying:
    1. **The set of registers** it contains and their function.
    2. **The sequence of microoperations** performed on the binary information stored in the registers.
    3. **The control** that initiates the sequence of microoperations.
- **It is more convenient to adopt a suitable** symbology to describe the sequence of transfers between registers and the various arithmetic and logic microoperations associated with the transfers.
- **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.

- **It is a convenient tool** for describing the internal organization of digital computers in concise and precise manner. It can also be used to facilitate the design process of digital systems.

18) 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 numbered in sequence from 0 through n - 1, starting from 0 in the rightmost position and increasing the numbers toward the left
- **The most common way** to represent a register is by a rectangular box with the name of the register inside.
- **Information transfer** from one register to another is designated in symbolic form by means of a replacement operator. The statement R2 ← R1 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.
- **A statement that specifies** a register transfer implies that circuits are available from the outputs of the source register to the inputs of the destination register and that the destination register has a parallel load capability.

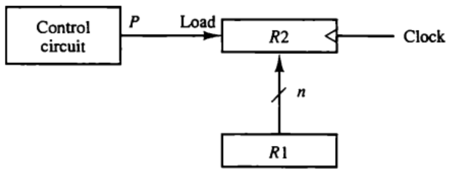
Figure Block diagram of register.



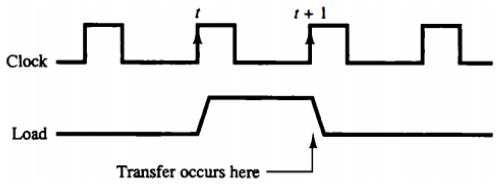
- **Normally**, we want the transfer to occur only under a predetermined control condition. This can be shown by means of an if-then statement.
- **If (P = 1) then (R2 ← R1)** where P is a control signal generated in the control section. It is sometimes convenient to separate the control variables from the register transfer operation by specifying a control function.
- **A control function** is a Boolean variable that is equal to 1 or 0. The control function is included in the statement as follows: P: R2 ← R1
- **The control condition** is terminated with a colon. It symbolizes the requirement that the transfer operation be executed by the hardware only if P = 1.
- **Every statement** written in a register transfer notation implies a hardware construction for implementing the transfer. Figure below shows the block diagram that depicts the transfer from R1 to R2. The n outputs of register R1 are connected to the n inputs of register R2.

- **The letter n will be used to indicate** any number of bits for the register. It will be replaced by an actual number when the length of the register is known. Register R2 has a load input that is activated by the control variable P.
- **It is assumed that the control variable** is synchronized with the same clock as the one applied to the register. As shown in the timing diagram, P is activated in the control section by the rising edge of a clock pulse at time t. The next positive transition of the clock at time t + 1 finds the load input active and the data inputs of R2 are then loaded into the register in parallel.

Figure Transfer from R1 to R2 when P = 1.



(a) Block diagram



(b) Timing diagram

P may go back to 0 at time t + 1; otherwise, the transfer will occur with every clock pulse transition while P remains active. Note that the clock is not included as a variable in the register transfer statements. It is assumed that all transfers occur during a clock edge transition.

- **Even though the control condition** such as P becomes active just after time t, the actual transfer does not occur until the register is triggered by the next positive transition of the clock at time t + 1.
- **The basic symbols** of the register transfer notation are listed in Table above. 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.

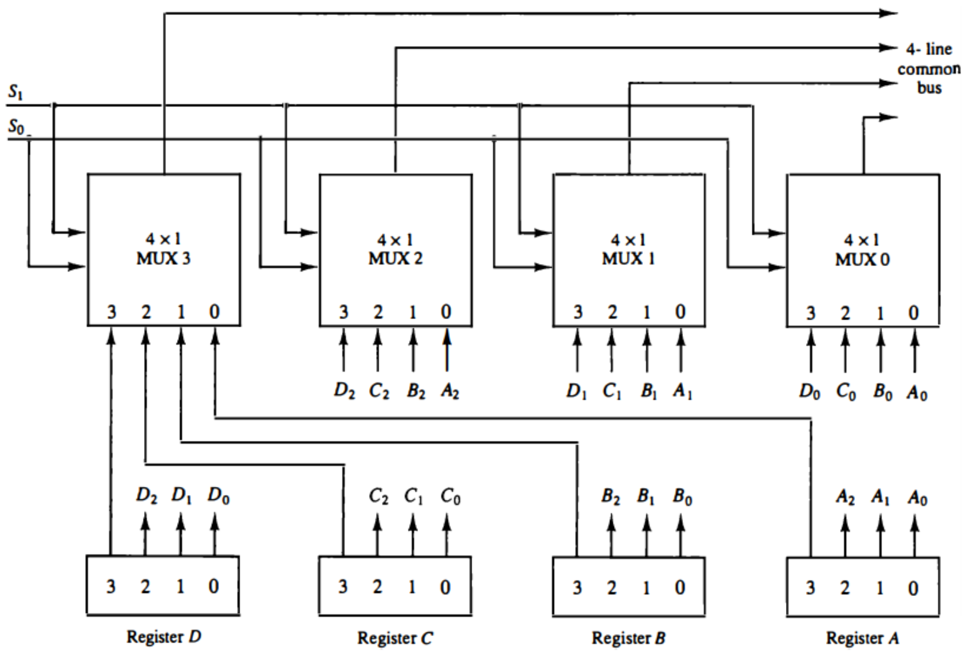
TABLE Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	*MAR*, *R*2
Parentheses ( )	Denotes a part of a register	*R*2(0–7), *R*2(*L*)
Arrow ←	Denotes transfer of information	*R*2 ← *R*1
Comma ,	Separates two microoperations	*R*2 ← *R*1, *R*1 ← *R*2

19) Bus and Memory Transfers

- **A typical digital computer** has many registers, and paths must be provided to transfer information from one register to another.
- **The number of wires** will be excessive if separate lines are used between each register and all other registers in the system.
- **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.
- **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 Fig. below. 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, S1 and S0. In order not to complicate the diagram with 16 lines crossing each other, we use labels 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 A1. The diagram shows that the bits in the same significant position in each register are connected to the data inputs of one multiplexer to form one line of the bus.
- **Thus MUX 0 multiplexes** the four 0 bits of the registers, MUX 1 multiplexes the four 1 bits of the registers, and similarly for the other two bits.

Figure Bus system for four registers.



- **The two selection lines S1 and S0** are connected to the selection inputs of all four multiplexers.
- **The selection lines** choose the four bits of one register and transfer them into the four-line common bus. When S1S0 = 00, the 0 data inputs of all four multiplexers are selected and applied to the outputs that form the bus.
- **This causes the bus lines** to receive the content of register A since the outputs of this register are connected to the 0 data inputs of the multiplexers.

- **Similarly, register B** is selected if S1S0 = 01, and so on. Table below shows the register that is selected by the bus for each of the four possible binary value of the selection lines.

TABLE Function Table for Bus of Fig.

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

- **In general**, a bus system will multiplex k registers of n bits each to produce an n-line common bus. The number of multiplexers needed to construct the bus is equal to n, the number of bits in each register. The size of each multiplexer must be k x 1 since it multiplexes k data lines.
- **For example**, a common bus for eight registers of 16 bits each requires 16 multiplexers, one for each line in the bus. Each multiplexer must have eight data input lines and three selection lines to multiplex one significant bit in the eight registers.
- **The transfer of information** from a bus into one of many destination registers can be accomplished by connecting the bus lines to the inputs of all destination registers and activating the load control of the particular destination register selected.
- **The symbolic statement** for a bus transfer may mention the bus or its presence may be implied in the statement. When the bus is includes in the statement, the register transfer is symbolized as follows: BUS ← C, R1 ← BUS
- **The content of register C** is placed on the bus, and the content of the bus is loaded into register R 1 by activating its load control input. If the bus is known to exist in the system, it may be convenient just to how the direct transfer. R1 ← C
- **From this statement the designer** knows which control signals must be activated to produce the transfer through the bus.

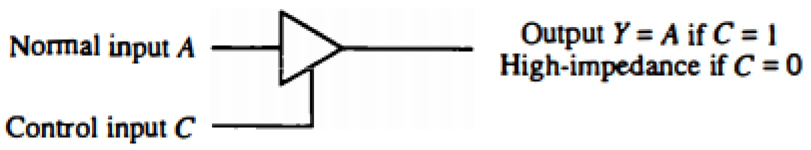
20) Three-State Bus Buffers

- **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 logic significance.
- **Three-state gates** may perform any conventional logic, such as AND or NAND. However, the one most commonly used in the design of a bus system is the buffer gate.
- **The graphic symbol** of a three-state buffer gate is shown in Fig. below. 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 high-impedance state** of a three-state gate provides a special feature not available in other gates. Because of this feature, a large number of three-state gate outputs can be connected with wires to form a common bus line without endangering loading effects.

- **The construction of a bus system** with three-state buffers is demonstrated in Fig. below. 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.
- **No more than one buffer** may be in the active state at any given time. The connected buffers must be controlled so that only one three-state buffer has access to the bus line while all other buffers are maintained in a high impedance state.
- **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. Careful investigation will reveal that Fig. below is another way of constructing a 4 x 1 multiplexer since the circuit can replace the multiplexer in Fig. Bus system for four registers.

  To construct a common bus for four registers of n bits each using three

Figure Graphic symbols for three-state buffer.



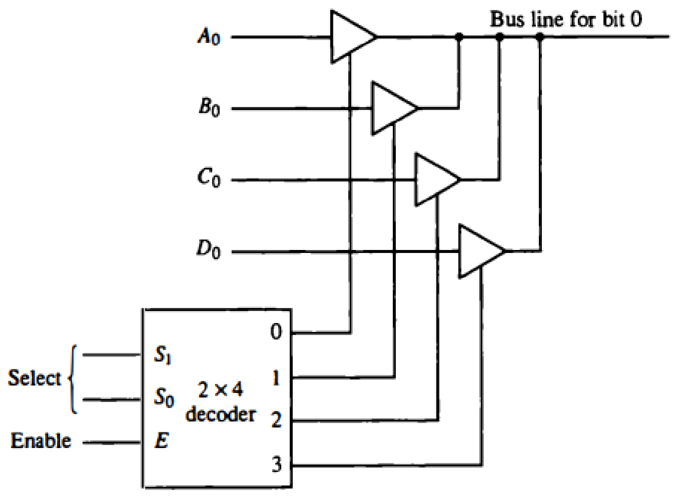


Figure Bus line with three state-buffers.

- **state buffers**, we need n circuits with four buffers in each as shown in Fig. above. Each group of four buffers receives one significant bit from the four registers.
- **Each common output** produces one of the lines for the common bus for a total of n lines. Only one decoder is necessary to select between the four registers.

21) 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.
- **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
- **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 R3 ← R1 + R2 + 1. R2 is the symbol for the 1's complement of R2.
- **Adding 1 to the 1's complement** produces the 2' s complement. Adding the contents of R1 to the 2's complement of R2 is equivalent to R1 - R2.

TABLE 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

- **The increment and decrement microoperations** are symbolized by plusone and minus-one operations, respectively. These microoperations are implemented with a combinational circuit or with a binary up-down counter.
- **The arithmetic operations** of multiply and divide are not listed in Table above. These two operations are valid arithmetic operations but are not included in the basic set of microoperations.
- **The only place** where these operations can be considered as microoperations is in a digital system, where they are implemented by means of a combinational circuit.
- **In such a case**, the signals that perform these operations propagate through gates, and the result of the operation can be transferred into a destination register by a clock pulse as soon as the output signal propagates through the combinational circuit.
- **In most computers**, the multiplication operation is implemented with a sequence of add and shift microoperations.
- **Division** is implemented with a sequence of subtract and shift microoperations.
- **To specify the hardware** in such a case requires a list of statements that use the basic microoperations of add, subtract, and shift

22) Binary Adder

- **To implement the add microoperation** with hardware, we need the registers that hold the data and the digital component that performs the arithmetic addition.
- **The digital circuit** that forms the arithmetic sum of two bits and a previous carry is called a full-adder.
- **The digital circuit** that generates the arithmetic sum of two binary numbers of any length 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. Figure below shows the 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 C0 and the output carry is C4. The S outputs of the full-adders generate the required sum bits.
- **An n-bit binary adder** requires n full-adders. The output carry from each full-adder is connected to the input carry of the next-high-order full-adder.
- **The n data bits** for the A inputs come from one register (such as R1), and the n data bits for the B inputs come from another register (such as R2).
- **The sum can be transferred** to a third register or to one of the source registers (R1 or R2), replacing its previous content.

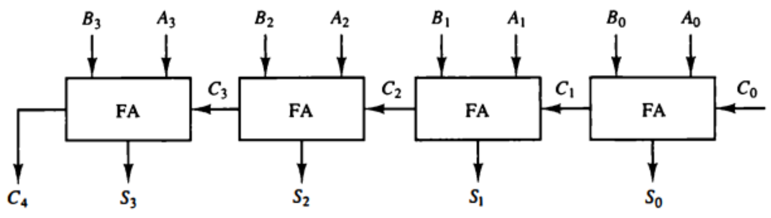


Figure 4-bit binary adder.

23) 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.
- **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. below.
- **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 O, and the circuit performs A plus B. When M = 1, we have $B \oplus 1 = B'$ and C0 = 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. 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 tha there Is no overflow.

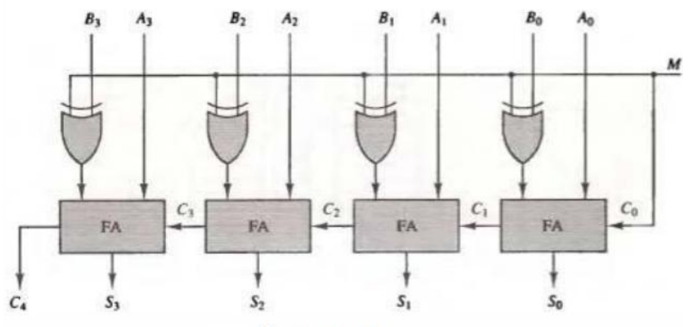


Figure 4-bit adder-subtractor.

24) 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.
- **The diagram of a 4-bit** combinational drcuit incrementer is shown in Fig. below. 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 A0 through A3. adds one to it, and generates the incremented output in S0 through S3.
- **The output carry C4** will be 1 only after incrementing binary 1111. This also causes outputs S0 through S3 to go to 0. The circuit of Fig. below can be extended to an n-bit binary incrementer by extending the diagram to include n half-adders.
- **The least significant bit** must have one input connected to logic-1. The other inputs receive the number to be incremented or the carry from the previous stage.

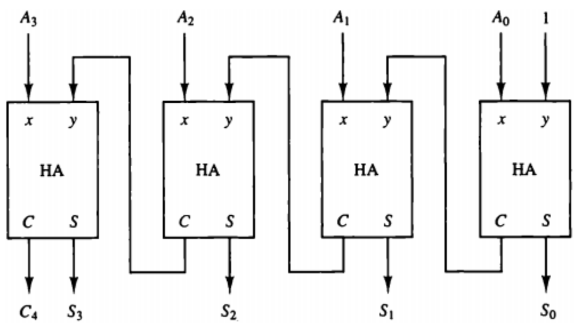


Figure 4-bit binary incrementer.

25) Arithmetic Circuit

- **The arithmetic microoperations** listed in Table below 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.

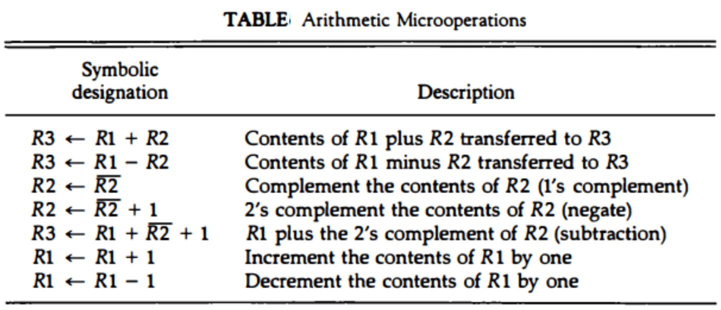


TABLE Arithmetic Microoperations

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

- **The diagram of a 4-bit arithmetic circuit** is shown in Fig. below. 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** are connected to the data inputs of the multiplexers. 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.
- **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 Table below.

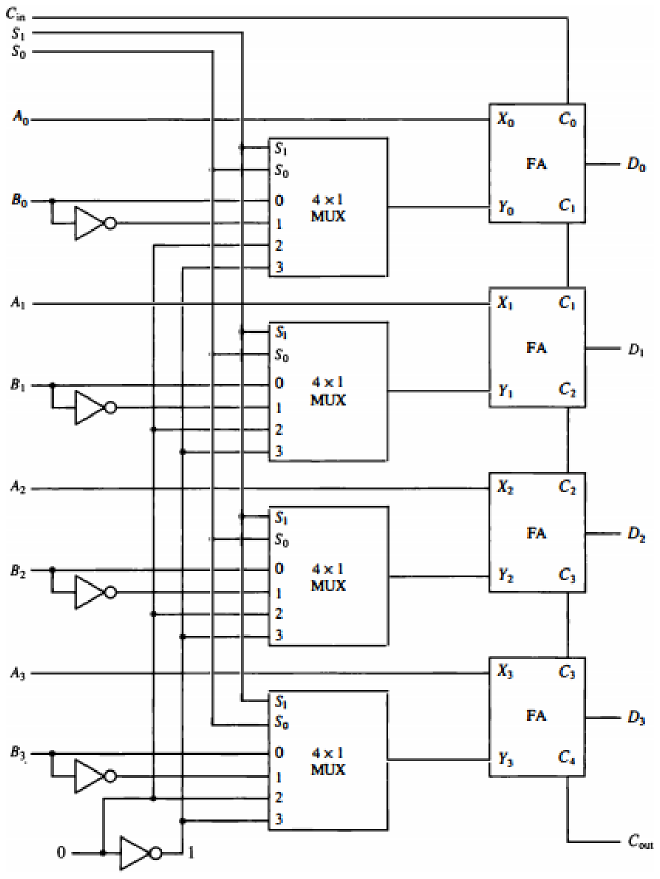


Figure 4-bit arithmetic circuit.

•

TABLE 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

- **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 + l. 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. When S_1S_0 = 10, the inputs from B are neglected, and instead, all O's are inserted into the Y inputs. The output becomes D = A + 0 + Cm路 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.

26) 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 R 1 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:

$$\begin{array}{ll} 1010 & \text{Content of } R1 \\ 1100 & \text{Content of } R2 \\ \hline 0110 & \text{Content of } R1 \text{ after } P = 1 \end{array}$$

- **The content of R1**, after the execution of the microoperation, is equal to the bit-by-bit exclusive-OR operation on pairs of bits in R2 and previous values of R1. The logic microoperations are seldom used in scientific computations, but they are very useful for bit manipulation of binary data and for making logical decisions.
- **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.
- **For example,** in the statement P + Q: R1 ← R2 + R3, R4 ← R5 ∨ R6 the + between P and Q is an OR operation between two binary variables of a control function.
- **The + between R2 and R3** specifies an add microoperation. The OR microoperation is designated by the symbol V between registers R5 and R6.

27) List of Logic Microoperations

- **There are 16 different logic operations** that can be performed with two binary variables.

- **They can be determined** from all possible truth tables obtained with two binary variables as shown in Table below.
- **In this table**, each of the 16 columns F0 through F15 represents a truth table of one possible Boolean function for the two variables x and y.

TABLE Truth Tables for 16 Functions of Two Variables

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

- **Note that the functions** are determined from the 16 binary combinations that can be assigned to F. The 16 Boolean functions of two variables x and y are expressed in algebraic form in the first column of Table below.
- **The 16 logic microoperations** are derived from these functions by replacing variable x by the binary content of register A and variable y by the binary content of register B.
- **It is important** to realize that the Boolean functions listed in the first column of Table below represent a relationship between two binary variables x and y.
- **The logic microoperations** listed in the second column represent a relationship between the binary content of two registers A and B.
- **Each bit of the register** is treated as a binary variable and the microoperation is performed on the string of bits stored in the registers.

TABLE Sixteen Logic Microoperations

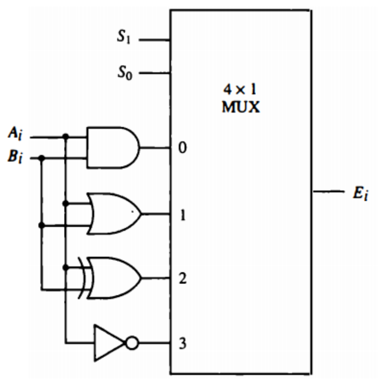
Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR
$F_8 = (x + y)'$	$F \leftarrow \overline{A \vee B}$	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

28) Hardware Implementation

- **The hardware implementation** of logic rnicrooperations 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 rnicrooperations**, most computers use only four-AND, OR, XOR (exclusive-OR), and complement from which all others can be derived.
- **Figure below** shows one stage of a circuit that generates the four basic logic rnicrooperations .
- **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. below lists the logic rnicrooperations obtained for each combination of the selection variables.

Figure One stage of logic circuit.



(a) Logic diagram

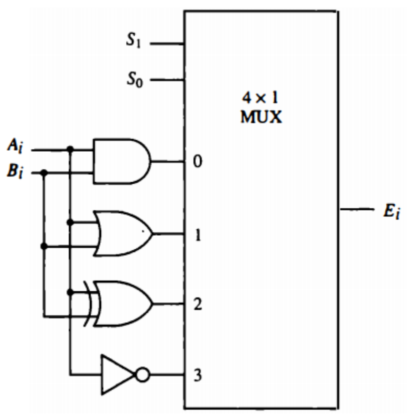
S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \bar{A}$	Complement

(b) Function table

29)Some Applications

- **Logic microoperations** are very useful for manipulating individual bits or a portion of a word stored in a register.
- **They can be used to change bit values**, delete a group of bits, or insert new bit values into a register. The following examples show how the bits of one register (designated by A) are manipulated **by logic microoperations** as a function of the bits of another register (designated by B).

Figure One stage of logic circuit.



(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	$E = \bar{A}$	Complement

(b) Function table

- **In a typical application**, register A is a processor register and the bits of register B constitute a logic operand extracted from memory and placed in register B.

- **The selective-set** operation sets to 1 the bits in register A where there are corresponding 1's in register B. It does not affect bit positions that have 0's in B.
- **The following numerical example** clarifies this operation:

```
1010    A before
1100    B (logic operand)
----
1110    A after
```

- **The two leftmost bits** of B are 1' s, so the corresponding bits of A are set to 1.
- **One of these two bits** was already set and the other has been changed from 0 to 1. The two bits of A with corresponding 0' s in B remain unchanged. The example above serves as a truth table since it has all four possible combinations of two binary variables.
- **From the truth table** we note that the bits of A after the operation are obtained from the logic-OR operation of bits in B and previous values of A. Therefore, the OR micro operation can be used to selectively set bits of a register.
- **The selective-complement** operation complements bits in A where there are selective-clear corresponding 1's in B. It does not affect bit positions that have 0's in B. For example:

```
1010    A before
1100    B (logic operand)
----
0110    A after
```

- **Again the two leftmost bits** of B are 1's, so the corresponding bits of A are complemented.
- **This example** again can serve as a truth table from which one can deduce that the selective-complement operation is just an exclusive-OR micro operation.
- **Therefore**, the exclusive-OR micro operation can be used to selectively complement bits of a register.
- **The selective-clear** operation clears to 0 the bits in A only where there are corresponding 1's in B. For example:

```
1010    A before
1100    B (logic operand)
----
0010    A after
```

- **Again the two leftmost bits** of B are 1' s, so the corresponding bits of A are cleared to 0.
- **One can deduce** that the Boolean operation performed on the individual bits is AB'. The corresponding logic microoperation is $A \leftarrow A \wedge B$
- **The mask operation** is similar to the selective-clear operation except that the bits of A are cleared only where there are corresponding 0's in B. The mask operation is an AND micro operation as seen from the following numerical example:

```
1010    A before
1100    B (logic operand)
----
1000    A after masking
```

- **The two rightmost bits of A** are cleared because the corresponding bits of B are 0' s. The two leftmost bits are left unchanged because the corresponding bits of B are 1's.

- **The mask operation** is more convenient to use than the selective clear operation because most computers provide an AND instruction, and few provide an instruction that executes the micro operation for selective-clear.
- **The insert operation** inserts a new value into a group of bits. This is done by first masking the bits and then ORing them with the required value. For example, suppose that an A register contains eight bits, 0110 1010.
- **To replace the four leftmost bits** by the value 1001 we first mask the four unwanted bits:

$$\begin{array}{ll} 0110\ 1010 & A \text{ before} \\ 0000\ 1111 & B \text{ (mask)} \\ \hline 0000\ 1010 & A \text{ after masking} \end{array}$$

- **and then** insert the new value:

$$\begin{array}{ll} 0000\ 1010 & A \text{ before} \\ 1001\ 0000 & B \text{ (insert)} \\ \hline 1001\ 1010 & A \text{ after insertion} \end{array}$$

- **The mask operation** is an AND micro operation and the insert operation is an OR micro operation.
- **The clear operation** compares the words in A and B and produces an all 0' s result if the two numbers are equal. This operation is achieved by an exclusive-OR micro operation as shown by the following example

$$\begin{array}{ll} 1010 & A \\ 1010 & B \\ \hline 0000 & A \leftarrow A \oplus B \end{array}$$

- **When A and B are equal**, the two corresponding bits are either both 0 or both 1. In either case the exclusive-OR operation produces a 0. The all-0's result is then checked to determine if the two numbers were equal.

30) Shift Micro operations

- **Shift micro operations** are used for serial transfer of data. They are also used in conjunction with arithmetic, logic, and other data-processing operations.
- **The contents of a register** can be shifted to the left or the right. At the same time that the bits are shifted, the first flip-flop receives its binary information from the serial input.
- **During a shift-left** operation the serial input transfers a bit into the rightmost position.
- **During a shift-right** operation the serial input transfers a bit into the leftmost position.
- **The information** transferred through the serial input determines the type of shift.
- **There are three types of shifts:** logical, circular, and arithmetic.
- **A logical shift** is one that transfers 0 through the serial input. We will adopt the symbols shl and shr for logical shift-left and shift-right micro operations. For example: **two micro operations** that specify a 1-bit shift to the left of the content of register R 1 and a 1-bit shift to the right of the content of register R2. The register symbol must be the same on both sides of the arrow

```
R1 ← shl R1

R2 &arr; shr R2
```

- **The bit transferred to the end position through the serial input** is assumed to be 0 during a logical shift. The circular shift (also known as a rotate operation) circulates the bits of the register around the two ends without loss of information.
- **This is accomplished by connecting** the serial output of the shift register to its serial input.
- **We will use the symbols cil and cir** for the circular shift left and right, respectively. The symbolic notation for the shift micro operations is shown in Table below.

TABLE Shift Microoperations

Symbolic designation	Description
$R \leftarrow \text{shl } R$	Shift-left register R
$R \leftarrow \text{shr } R$	Shift-right register R
$R \leftarrow \text{cil } R$	Circular shift-left register R
$R \leftarrow \text{cir } R$	Circular shift-right register R
$R \leftarrow \text{ashl } R$	Arithmetic shift-left R
$R \leftarrow \text{ashr } R$	Arithmetic shift-right R

- **An arithmetic shift** is a micro operation that shifts a signed binary number to the left or right. An arithmetic shift-left multiplies a signed binary number by 2. An arithmetic shift-right divides the number by 2.
- **Arithmetic shifts** must leave the sign bit unchanged because the sign of the number remains the same

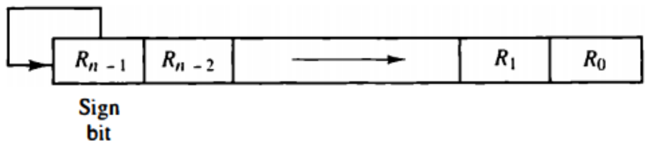


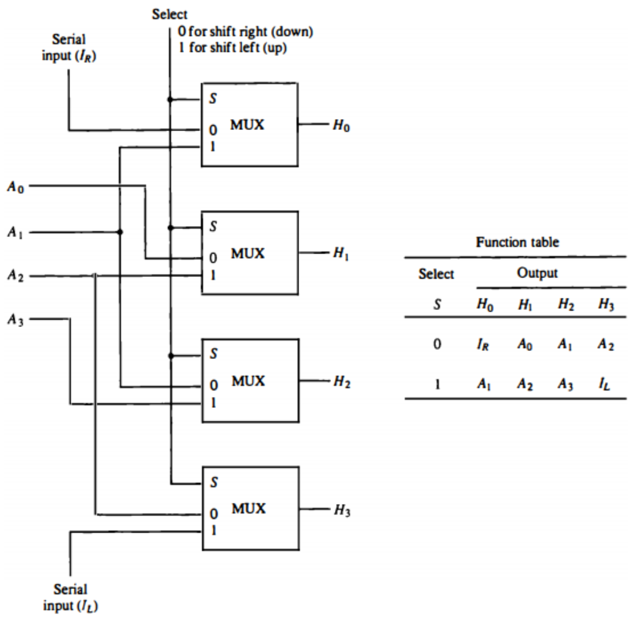
Figure Arithmetic shift right.

- **when it is multiplied or divided by 2**. The leftmost bit in a register holds the sign bit, and the remaining bits hold the number. The sign bit is 0 for positive and I for negative. Negative numbers are in 2's complement form.
- **Figure above shows a typical register of n bits**. Bit R_{n-1} in the leftmost position holds the sign bit. R_{n-2} is the most significant bit of the number and R_0 is the least significant bit.
- **The arithmetic shift-right** leaves the sign bit unchanged and shifts the number (including the sign bit) to the right.
- **Thus R_{n-1}** remains the same, R_{n-2} receives the bit from R_{n-1} and so on for the other bits in the register. The bit in R_0 is lost.
- **The arithmetic shift-left** inserts a 0 into R_0, and shifts all other bits to the left. The initial bit of R_{n-1} is lost and replaced by the bit from R_{n-2}. A sign reversal occurs if the bit in R_{n-1} changes in value after the shift.
- **This happens if the multiplication** by 2 causes an overflow. An overflow occurs after an arithmetic shift left if initially, before the shift, R_{n-1} is not equal to R_{n-2}.
- **An overflow flip-flop** V_s, can be used to detect an arithmetic shift-left overflow. $V_s = R_{n-1} \oplus R_{n-2}$
- **If $V_s = 0$**, there is no overflow, but if $V_s = 1$, there is an overflow and a sign reversal after the shift. V_s must be transferred into the overflow flip-flop with the same clock pulse that shifts the register.

31) Hardware Implementation

- **A possible choice for a shift unit** would be a bidirectional shift register with parallel load

- **Information can be transferred** to the register in parallel and then shifted to the right or left.
- **In this type of configuration**, a clock pulse is needed for loading the data into the register, and another pulse is needed to initiate the shift.
- **In a processor unit** with many registers it is more efficient to implement the shift operation with a combinational circuit.
- **In this way the content of a register** that has to be shifted is first placed onto a common bus whose output is connected to the combinational shifter, and the shifted number is then loaded back into the register.
- **This requires** only one clock pulse for loading the shifted value into the register.
- **A combinational circuit shifter** can be constructed with multiplexers as shown in Fig. below. The 4-bit shifter has four data inputs, A_0 through A_3, and four data outputs, H_0 through H_3.
- **There are two serial inputs**, one for shift left



Function table

Select	Output			
S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

Figure 4-bit combinational circuit shifter.

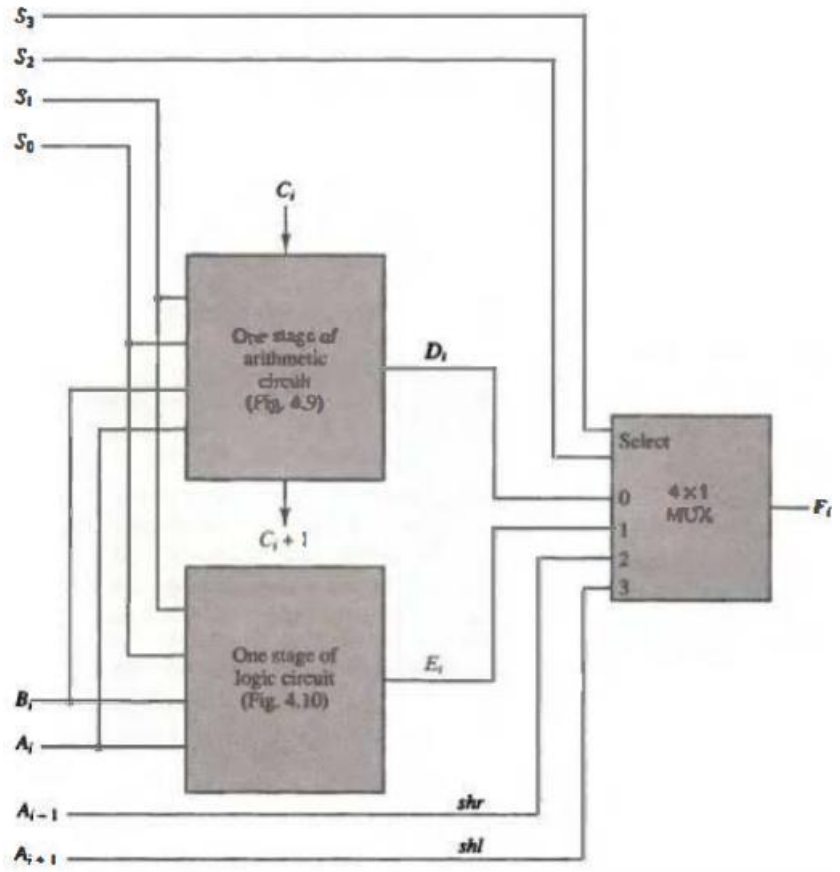
- **(I_L) and the other for shift right (I_L)**. When the selection input S = 0, the input data are shifted right (down in the diagram). When S = 1, the input data are shifted left (up in the diagram). The function table in Fig. above shows which input goes to each output after the shift.
- **A shifter with n data inputs** and outputs requires n multiplexers. The two serial inputs can be controlled by another multiplexer to provide the three possible types of shifts.

32) Arithmetic Logic Shift Unit

- **Instead of having individual registers** performing the micro operations 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 micro operation**, 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 micro operations** are often performed in a separate unit, but sometimes the shift unit is made part of the ALU.
- **The arithmetic, logic, and shift** circuits introduced in previous sections can be combined into one ALU with common selection variables.
- **One stage of an arithmetic logic shift** unit is shown in Fig. below. The subscript i designates a typical stage.
- **Inputs A_i and B_i** are applied to both the arithmetic and logic units

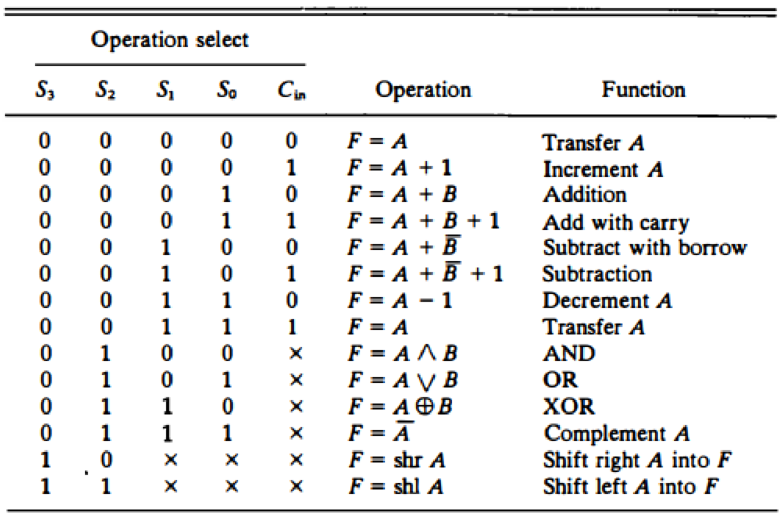
Figure One stage of arithmetic logic shift unit.



- **A particular micro operation** 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. below** 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.
- **The circuit** whose one stage is specified in Fig. above provides eight arithmetic operation, four logic operations, and two shift operations.

- **Each operation** is selected with the five variables S_3, S_2, S_1, S_0, and C_{in}.
- **The input carry** C_{in} is used for selecting an arithmetic operation only.
- **Table below** lists the 14 operations of the ALU.
- **The first eight** are arithmetic operations and are selected with S_3S_2 = 00. The next four are logic operations and are selected with S_3S_2 = 01. The input carry has no effect during the logic operations and is marked with don't-care x's.
- **The last two operations** are shift operations and are selected with S_3S_2 = 10 and 11. The other three selection inputs have no effect on the shift.

TABLE 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	×	$F = A \wedge B$	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	×	×	×	$F = \text{shr } A$	Shift right A into F
1	1	×	×	×	$F = \text{shl } A$	Shift left A into F

33) Basic Computer Organization and Design

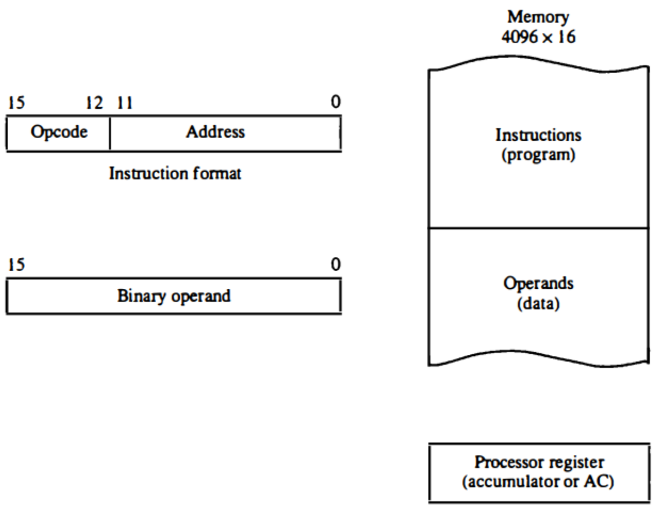
- **The organization of the computer** is defined by its internal registers, timing and control structures, and the sets of instructions it uses.
- **The internal organization** of a digital system is defined by the sequence of micro-operations it performs on data stored in its registers.
- **The general purpose computer** is capable of executing various micro-operations and can be instructed as to what specific sequence of operations it must perform. The user of a computer can control the process by means of a program.
- **A program is a set of instructions** that specify the operations operands, and the sequence by which processing has to occur. The dataprocessing task may be altered by specifying a new program with different instructions or specifying the same instructions with different data.
- **A computer instruction is a binary code** that specifies a sequence of microoperations for the computer. Instruction codes together with data are stored in memory. The computer reads each instruction from memory and places it in a control register.
- **The control then interprets the binary code** of the instruction and proceeds to execute it by issuing a sequence of microoperations. Every computer has its own unique instruction set.
- **The ability to store and execute instructions**, the stored program concept, is the most important property of a general-purpose computer.
- **An instruction code is a group of bits** that instruct the computer to perform a specific operation. It is divided into parts, each having its own particular 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.
- **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.
- **The relationship between a computer operation and a microoperation** is given next. An operation is part of an instruction stored in computer memory.
- **It is a binary code** that tells the computer to perform a specific operation. The control unit receives the instruction from memory and interprets the operation code bits.
- **It then issues a sequence of control signals** to initiate microoperations in internal computer registers. For every operation code, the control issues a sequence of microoperations needed for the hardware implementation of the specified operation.
- **For this reason**, an operation code is sometimes called a macrooperation because it specifies a set of microoperations.
- **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.
- **There are many variations** for arranging the binary code of instructions, and each computer has its own particular instruction code format. Instruction code formats are conceived by computer designers who specify the architecture of the computer.

34) 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.
- **This operand** is read from memory and used as the data to be operated on together with the data stored in the processor register.
- **Figure below** depicts this type of organization. Instructions are stored in one section of memory and data in another. For a memory unit with 4096 words we need 12 bits to specify an address since 2^{12} = 4096. If we store each instruction code in one 16-bit memory word, we have available four bits for the operation code (abbreviated opcode) to specify one out of 16 possible operations, and 12 bits to specify the address of an operand.
- **The control** reads a 16-bit instruction from the program portion of memory. It uses the 12-bit address part of the instruction to read a 16-bit operand from the data portion of memory.
- **It then executes** the operation specified by the operation code.

Figure Stored program organization.



- **Computers** that have a single-processor register usually assign to it the name accumulator and label it AC. The operation is performed with the memory operand and the content of AC.
- **If an operation** in an instruction code does not need an operand from memory, the rest of the bits in the instruction can be used for other purposes. For example, operations such as clear AC, complement AC, and increment AC operate on data stored in the AC register.
- **They do not need an operand** from memory. For these types of operations, the second part of the instruction code (bits 0 through 11) is not needed for specifying a memory address and can be used to specify other operations for the computer.

35) Indirect Address

- **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.
- **This is in contrast to a third possibility called indirect address**, where 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.
- **One bit of the instruction code** can be used to distinguish between a direct and an indirect address.
- **As an illustration** of this configuration, consider the instruction code format shown in Fig. below part (a). It consists of a 3-bit operation code, a 12-bit address, and an indirect address mode bit designated by I.
- **The mode bit is 0** for a direct address and 1 for an indirect address. A direct address instruction is hown in Fig. below part (b). It is placed in address 22 in memory.
- **The I bit is 0**, so the instruction is recognized as a direct address instruction. The opcode specifies an ADD instruction, and the address part is the binary equivalent of 457.
- **The control** finds the operand in memory at address 457 and adds it to the content of AC. The instruction in address 35 shown in Fig. below part (c) has a mode bit I = 1.
- **Therefore**, it is recognized as an indirect address instruction. The address part is the binary equivalent of 300. The control goes to address 300 to find the address of the operand.

- **The address of the operand** in this case is 1350. The operand found in address 1350 is then added to the content of AC. The indirect address instruction needs two references to memory to fetch an operand. The first reference is needed to read the address of the operand; the second is for the operand itself.
- **We define the effective address** to be the address of the operand in a computation-type instruction or the target address in a branch-type instruction.
- **Thus the effective address** in the instruction of Fig. below part (b) is 457 and in the instruction of Fig below part (c) is 1350.
- **The direct and indirect addressing modes** are used in the computer presented in this chapter.
- **The memory word** that holds the address of the operand in an indirect address instruction is used as a pointer to an array of data. The pointer could be placed in a processor register instead of memory as done in commercial computers.

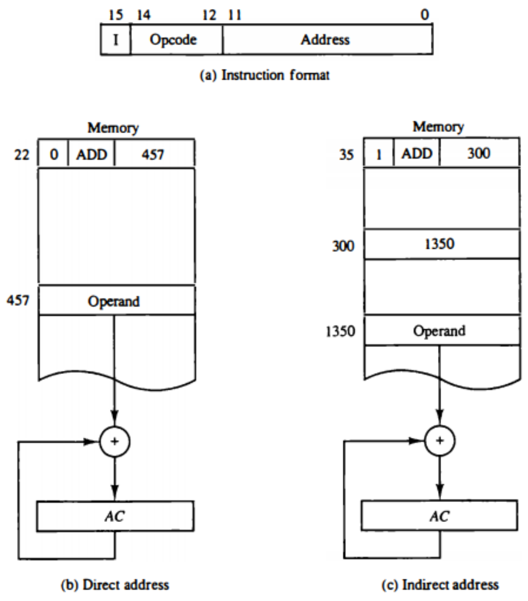


Figure Demonstration of direct and indirect address.

36) Computer Registers

- **Computer instructions** are normally stored in consecutive memory locations and are executed sequentially one at a time.
- **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.
- **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. These requirements dictate the register configuration shown in Fig. below.
- **The registers** are also listed in Table below together with a brief description of their function and the number of bits that they contain.
- **The memory unit has a capacity of 4096 words** and each word contains 16 bits. Twelve bits of an instruction word are needed to specify the address of an operand.

- **This leaves three bits** for the operation part of the instruction and a bit to specify a direct or indirect address. The data register (DR) holds the operand read from memory.
- **The accumulator (AC)** register is a general purpose processing register. The instruction read from memory is placed in the instruction register (IR). The temporary register (TR) is used for holding temporary data during the processing

TABLE 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

- **The memory address register (AR)** has 12 bits since this is the width of a memory address. The program counter (PC) also has 12 bits and it holds the address of the next instruction to be read from memory after the current instruction is executed.
- **The PC goes** through a counting sequence and causes the computer to read sequential instructions previously stored in memory. Instruction words are read and executed in sequence unless a branch instruction is encountered.
- **A branch instruction calls** for a transfer to a nonconsecutive instruction in the program.
- **The address part** of a branch instruction is transferred to PC to become the address of the next instruction. To read an instruction, the content of PC is taken as the address for memory and a memory read cycle is initiated.
- **PC is then incremented by one**, so it holds the address of the next instruction in sequence.
- **Two registers** are used for input and output. The input register (INPR) receives an 8-bit character from an input device. The output register (OUTR) holds an 8-bit character for an output device.

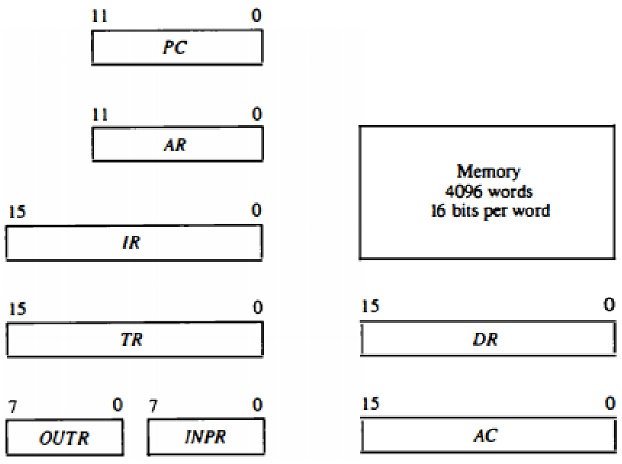


Figure Basic computer registers and memory.

37) Common Bus System

- **The basic computer has eight registers**, a memory unit, and a control unit . Paths must be provided to transfer information from one register to another and between memory and registers.
- **The number of wires will be excessive** if connections are made between the outputs of each register and the inputs of the other registers.
- **A more efficient** scheme for transferring information in a system with many registers is to use a common bus.
- **The connection of the registers** and memory of the basic computer to a common bus system is shown in Fig. below. The outputs of seven registers and memory are connected to the common bus.
- **The specific output** that is selected for the bus lines at any given time is determined from the binary value of the selection variables S_2, S_1, and S_0.
- **The number along** each output shows the decimal equivalent of the required binary selection. For example, the number along the output of DR is 3.
- **The 16-bit outputs of DR** are placed on the bus lines when $S_2S_1S_0 = 011$ since this is the binary value of decimal 3.
- **The lines from the common bus** are connected to the inputs of each register and the data inputs of the memory. The particular register whose LD (load) input is enabled receives the data from the bus during the next clock pulse transition.
- **The memory** receives the contents of the bus when its write input is activated. The memory places its 16-bit output onto the bus when the read input is activated and $S_2S_1S_0 = 111$.
- **Four registers**, DR, AC, IR, and TR, have 16 bits each. Two registers, AR **and PC, have 12 bits** each since they hold a memory address. When the contents of AR or PC are applied to the 16-bit common bus, the four most significant bits are set to 0's.

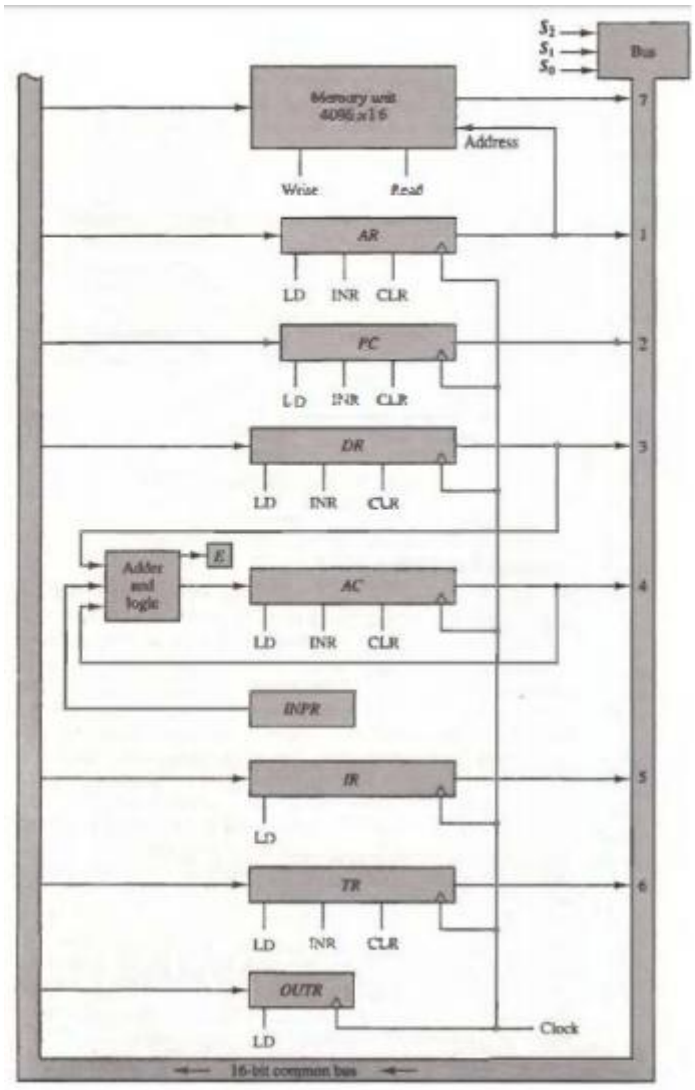


Figure 5-4 Basic computer registers connected to a common bus.

- **When AR or PC** receive information from the bus, only the 12 least significant bits are transferred into the register. The input register INPR and the output register OUTR have 8 bits each and communicate with the eight least significant bits in the bus.
- **INPR is connected to provide information** to the bus but OUTR can only receive information from the bus.
- **This is because INPR** receives a character from an input device which is then transferred to AC. OUTR receives a character from AC and delivers it to an output device. There is no transfer from OUTR to any of the other registers.
- **The 16 lines of the common bus** receive information from six registers and the memory unit. The bus lines are connected to the inputs of six registers and the memory. Five registers have three control inputs: LD (load), INR (increment), and CLR (clear).
- **This type of register** is equivalent to a binary counter with parallel load and synchronous clear. The increment operation is achieved by enabling the count input of the counter. Two registers have only a LD input.
- **The input data and output data** of the memory are connected to the common bus, but the memory address is connected to AR. Therefore, AR must always be used to specify a memory address.

- **By using a single register** for the address, we eliminate the need for an address bus that would have been needed otherwise. The content of any register can be specified for the memory data input during a write operation. Similarly, any register can receive the data from memory after a read operation except AC .
- **The 16 inputs of AC** come from an adder and logic circuit. This circuit has three sets of inputs. One set of 16-bit inputs come from the outputs of AC . They are used to implement register microoperations such as complement AC and shift AC .
- **Another set of 16-bit inputs** come from the data register DR. The inputs from DR and AC are used for arithmetic and logic rnlcrooperations, such as add DR to AC or AND DR to AC.
- **The result of an addition** is transferred to AC and the end carry-out of the addition is transferred to flip-flop E (extended AC bit). A third set of 8-bit inputs come from the input register INPR.
- **Note that the content of any register** can be applied onto the bus and an operation can be performed in the adder and logic circuit during the same clock cycle. The clock transition at the end of the cycle transfers the content of the bus into the designated destination register and the output of the adder and logic circuit into AC.
- **For example**, the two microoperations **can be executed at the same time**. This can be done by placing the content of AC on the bus (with $S_2S_1S_0 = 100$), enabling the LD (load) input of DR, transferring the content of DR through the adder and logic circuit into AC, and enabling the LD (load) input of AC, all during the same clock cycle.

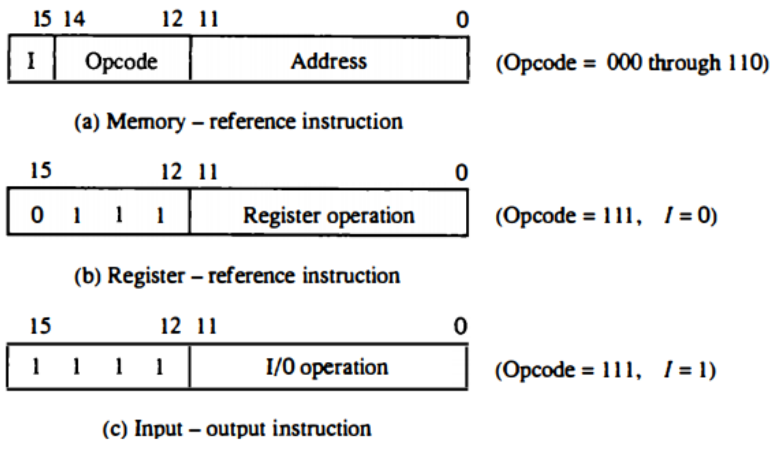
```
DR ← AC and AC ← DR
```

- **The two transfers** occur upon the arrival of the clock pulse transition at the end of the clock cycle.

38) Computer Instructions

- **The basic computer** has three instruction code formats, as shown in Fig. below. Each format has 16 bits.
- **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.
- **A register-reference** instruction specifies an operation on or a test of the AC register. An operand from memory is not needed; therefore, the other 12 bits are used to specify the operation or test to be executed.
- **Similarly**, an input-output instruction does not need a reference to memory and is recognized by the operation code Ill 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.
- **The type of instruction** is recognized by the computer control from the four bits in positions 12 through 15 of the instruction. If the three opcode bits in positions 12 though 14 are not equal to 111, the instruction is a memory-reference type and the bit in position 15 is taken as the addressing mode I.
- **If the 3-bit opcode** is equal to 111, control then inspects the bit in position 15. If this bit is 0, the

Figure Basic computer instruction formats.



- **instruction** is a register-reference type. If the bit is 1, the instruction is an input-output type. Note that the bit in position 15 of the instruction code is designated by the symbol I but is not used as a mode bit when the operation code is equal to 111.
- **Only three bits of the instruction** are used for the operation code. It may seem that the computer is restricted to a maximum of eight distinct operations.
- **However**, since register-reference and input-output instructions use the remaining 12 bits as part of the operation code, the total number of instructions can exceed eight.
- **In fact**, the total number of instructions chosen for the basic computer is equal to 25.
- **The instructions** for the computer are listed in Table below. The symbol designation is a three-letter word and represents an abbreviation intended for programmers and users. The hexadecimal code is equal to the equivalent hexadecimal number of the binary code used for the instruction.
- **By using the hexadecimal equivalent** we reduced the 16 bits of an instruction code to four digits with each hexadecimal digit being equivalent to four bits.
- **A memory-reference instruction** has an address part of 12 bits. The address part is denoted by three x's and stand for the three hexadecimal digits corresponding to the 12-bit address.
- **The last bit of the instruction** is designated by the symbol I. When I = 0, the last four bits of an instruction have a hexadecimal digit equivalent from 0 to 6 since the last bit is 0.
- **When I = I**, the hexadecimal digit equivalent of the last four bits of the instruction ranges from 8 to E since the last bit is I. Register-reference instructions use 16 bits to specify an operation. The leftmost four bits are always 0111, which is equivalent to hexadecimal 7.
- **The other three hexadecimal digits** give the binary equivalent of the remaining 12 bits. The input-output instructions also use all 16 bits to specify an operation. The last four bits are always 1111, equivalent to hexadecimal F.

TABLE Basic Computer Instructions

	Hexadecimal code		
Symbol	$I = 0$	$I = 1$	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

39) Instruction Set Completeness

- **A computer should have a set of instructions** so that the user can construct machine language programs to evaluate any function that is known to be computable.
- **The set of instructions** are said to be complete if the computer includes a sufficient number of instructions in each of the following categories:

1. Arithmetic, logical, and shift instructions
2. Instructions for moving information to and from memory and processor registers
3. Program control instructions together with instructions that check status conditions
4. Input and output instructions

- **Arithmetic, logical, and shift instructions** provide computational capabilities for processing the type of data that the user may wish to employ.
- **The bulk of the binary information** in a digital computer is stored in memory, but all computations are done in processor registers.
- **Therefore**, the user must have the capability of moving information between these two units. Decision making capabilities are an important aspect of digital computers.
- **For example**, two numbers can be compared, and if the first is greater than the second, it may be necessary to proceed differently than if the second is greater than the first.
- **Program control instructions** such as branch instructions are used to change the sequence in which the program is executed. Input and output instructions are needed for communication between the computer and the user.
- **Programs and data** must be transferred into memory and results of computations must be transferred back to the user. The instructions listed in Table above constitute a minimum set that provides all the capabilities mentioned above.
- **There is one arithmetic instruction**, ADD, and two related instructions, complement AC(CMA) and increment AC(INC).

- **With these three instructions** we can add and subtract binary numbers when negative numbers are in signed-2's complement representation. The circulate instructions, CIR and CIL, can be used for arithmetic shifts as well as any other type of shifts desired.

- **Multiplication and division** can be performed using addition, subtraction, and shifting. There are three logic operations: AND, complement AC(CMA), and clear AC(CLA).
- **The AND and complement provide a NAND operation**. It can be shown that with the NAND operation it is possible to implement all the other logic operations with two variables.
- **Moving information** from memory to AC is accomplished with the load AC(LDA) instruction. Storing information from AC into memory is done with the store AC(STA) instruction.
- **The branch instructions BUN, BSA, and ISZ**, together with the four skip instructions, provide capabilities for program control and checking of status conditions. The input (lNP) and output (OUT) instructions cause information to be transferred between the computer and external devices.
- **Although the set of instructions** for the basic computer is complete, it is not efficient because frequently used operations are not performed rapidly. An efficient set of instructions will include such instructions as subtract, multiply, OR, and exclusive-OR.

40) 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.
- **There are two major types** of control organization: hardwired control and microprogrammed 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.
- **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.
- **A hardwired control**, as the name implies, requires changes in the wiring among the various components if the design has to be modified or changed. In the microprogrammed control, any required changes or modifications can be done by updating the microprogram in control memory.
- **The block diagram of the control unit** is shown in Fig. below. It consists of two decoders, a sequence counter, and a number of control logic gates. An instruction read from memory is placed in the instruction register (IR).
- **The position of this register** in the common bus system is indicated in Fig. previously seen. The instruction register is shown again in Fig. below, where it is divided into three parts: the I bit, the operation code, and bits 0 through 11.
- **The operation code in bits 12 through 14** are decoded with a 3 x 8 decoder. The eight outputs of the decoder are designated by the symbols D0 through D7
- **The subscripted decimal number is equivalent** to the binary value of the corresponding operation code. Bit 15 of the instruction is transferred to a flip-flop designated by the symbol I.

- **Bits 0 through 11** are applied to the control logic gates. The 4-bit sequence counter can count in binary from 0 through 15. The outputs of the counter are decoded into 16 timing signals T0 through T15
- **The sequence counter SC** can be incremented or cleared synchronously
- **Most of the time**, the counter is incremented to provide the sequence of timing signals out of the 4 x 16 decoder. Once in awhile, the counter is cleared to 0, causing the next active timing signal to be T0.
- **As an example**, consider the case where SC is incremented to provide timing signals T0, T1, T2, T3, and T4 in sequence. At time T4, SC is cleared to 0 if decoder output D3 is active.
- **This is expressed symbolically** by the statement D3T4: SC ← 0. The timing diagram of Fig. below shows the time relationship of the control signals.
- **The sequence counter SC** responds to the positive transition of the clock. Initially, the CLR input of SC is active. The first positive transition of the

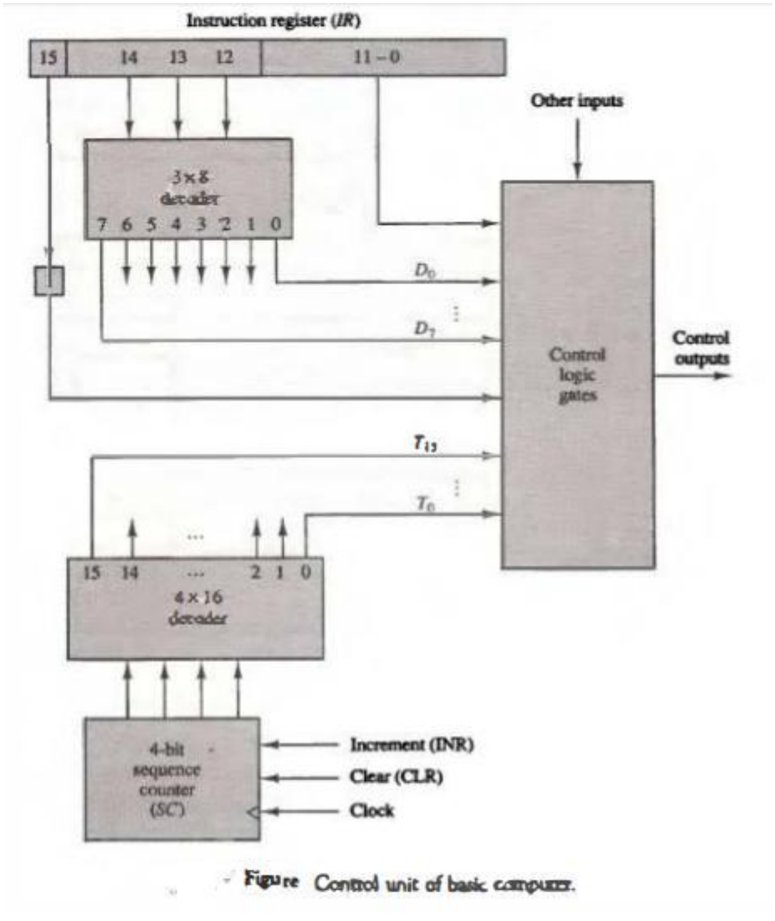


Figure Control unit of basic computer.

- **clock clears SC to 0**, which in turn activates the timing signal T0 out of the decoder. T0 is active during one clock cycle.
- **The positive clock transition** labeled T0 in the diagram will trigger only those registers whose control inputs are connected to timing signal T0.
- **SC is incremented** with every positive clock unless its CLR input is active.
- **This produces the sequence** of timing signals T0,T1,T2,T3,T4 and so on, as shown in the diagram. (Note the relationship between the timing signal and its corresponding positive clock transition.)
- **If SC is not cleared**, the timing signals will continue with T_5, T_6 up to T15 and back to T_0.

- **The last three waveforms in Fig. above** show how SC is cleared when D3T4 = 1. Output D3 from the operation decoder becomes active at the end of timing signal T2
- **When timing signal T4 becomes active**, the output of the AND gate that implements the control function D3T4 becomes active.
- **This signal is applied to the CLR input of SC**. On the next positive clock transition (the one marked T4 in the diagram) the counter is cleared to 0. This causes the timing signal T0 to become active instead of T5 that would have been active if SC were incremented instead of cleared.
- **A memory read or write cycle** will be initiated with the rising edge of a timing signal. It will be assumed that a memory cycle time is less than the clock cycle time. According to this assumption, a memory read or write cycle initiated by a timing signal will be completed by the time the next clock goes through its positive transition.
- **The clock transition** will then be used to load the memory word into a register. This timing relationship is not valid in many computers because the memory cycle time is usually longer than the processor clock cycle.
- **In such a case** it is necessary to provide wait cycles in the processor until the memory word is available. To facilitate the presentation, we will assume that a wait period is not necessary in the basic computer.
- **To fully comprehend the operation of the computer**, it is crucial that one understands the timing relationship between the clock transition and the timing signals. For example, the register transfer statement T0: AR ← PC specifies a transfer of the content of PC into AR if timing signal T0 is active.
- **T0 is active during an entire clock cycle interval**. During this time the content of PC is placed onto the bus (with $S_2S_1S_0$ = 010) and the LD (load) input of AR is enabled. The actual transfer does not occur until the end of the clock cycle when the clock goes through a positive transition.
- **This same positive clock transition** increments the sequence counter SC from 0000 to 0001 . The next clock cycle has T1 active and T0 inactive.

41) Instruction Cycle

- **A program residing** in the memory unit of the computer consists of a sequence of instructions. The program is executed in the computer by going through a cycle for each instruction. Each instruction cycle in turn is subdivided into a sequence of subcycles or phases. In the basic computer 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.
- **Fetch and Decode**
    1. **Initially, the program counter PC** is loaded with the address of the first instruction in the program.
    2. **The sequence counter SC** is cleared to 0, providing a decoded timing signal T0. After each clock pulse, SC is incremented by one, so that the timing signals go through a sequence T0, T1, T2, and so on.

3. **The rnicrooperations** for the fetch and decode phases can be specified by the following register transfer statements.

```
T0: AR ← PC

T1: IR ← M[AR], PC ← PC + 1

T2: D0, .... , D7 ← Decode IR(12-14), AR ← IR(0-11), I ← IR(15)
```

4. **Since only AR** is connected to the address inputs of memory, it is necessary to transfer the address from PC to AR during the clock transition associated with timing signal T0. The instruction read from memory is then placed in the instruction register IR with the clock transition associated with timing signal T1.

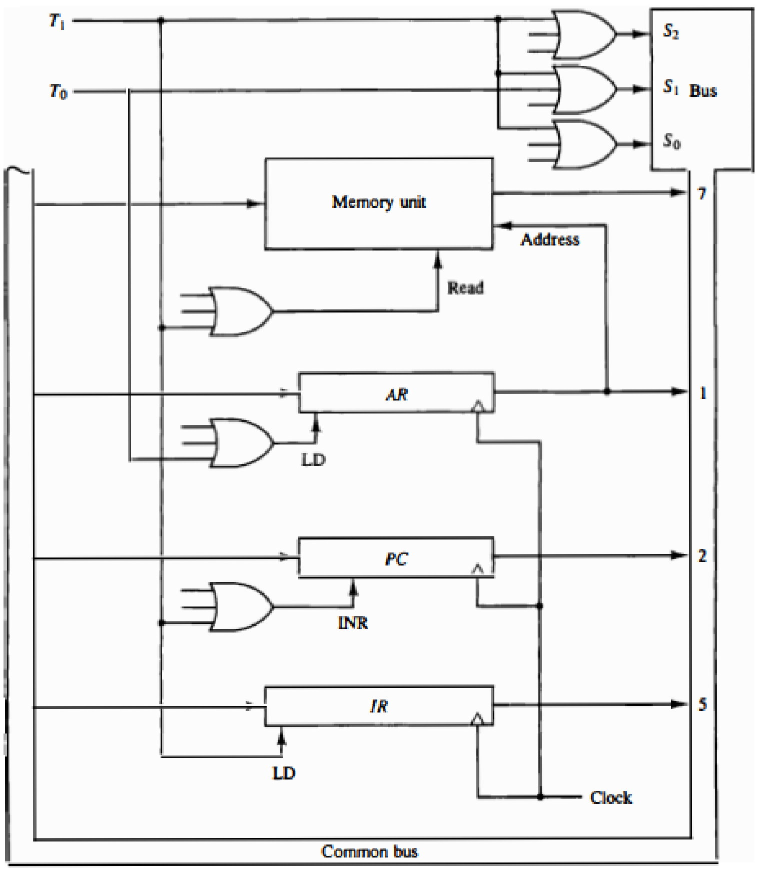


Figure Register transfers for the fetch phase.

5. **At the same time**, PC is incremented by one to prepare it for the address of the next instruction in the program. At time T2, the operation code in IR is decoded, the indirect bit is transferred to flip-flop I, and the address part of the instruction is transferred to AR .

6. **Note that SC** is incremented after each clock pulse to produce the sequence T0, T1, and T2

7. **Figure above** shows how the first two register transfer statements are implemented in the bus system.

8. **To provide** the data path for the transfer of PC to AR we must apply timing signal T0 to achieve the following connection:

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 .

9. **The next clock transition** initiates the transfer from PC to AR since T0 = 1. In order to implement the second statement : T1: IR ← M[AR], PC ← PC + 1 it is necessary to use timing signal T1 to provide the following connections in the bus system.

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.

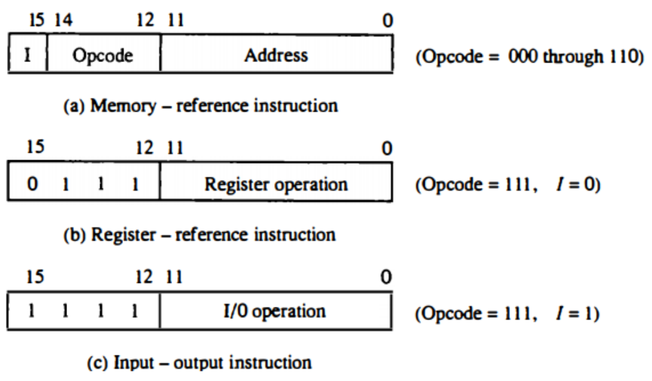
10. **The next clock transition** initiates the read and increment operations since T1 = 1.

11. **Figure above** duplicates a portion of the bus system and shows how T0 and T1 are connected to the control inputs of the registers, the memory, and the bus selection inputs. Multiple input OR gates are included in the diagram because there are other control functions that will initiate similar operations.

42) Determine the Type of Instruction

- **The timing signal** that is active after the decoding is T_3. During time T_3, the control unit determines the type of instruction that was just read from memory.
- **The flowchart of Fig. below** presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding.
- **The three possible** instruction types available in the basic computer are specified in Fig. on basic computer formats.

Figure Basic computer instruction formats.



- **Decoder output D_7 is equal to 1** if the operation code is equal to binary 111. From Fig. on basic computer formats we determine that if D_7 = 1, the instruction must be a register-reference or input-output type

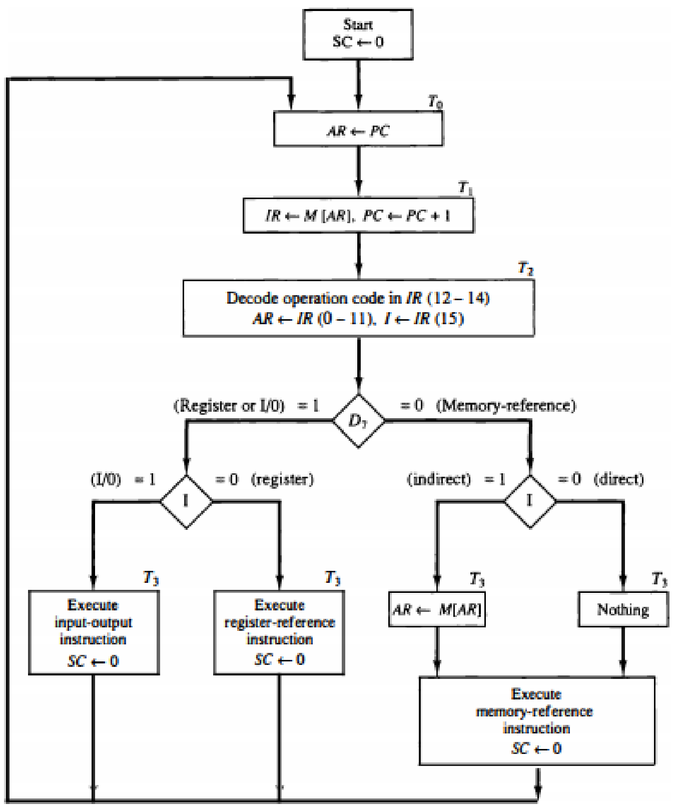

Start
SC ← 0
T_0
AR ← PC
T_1
IR ← M [AR], PC ← PC + 1
T_2
Decode operation code in IR (12 – 14)
AR ← IR (0 – 11), I ← IR (15)
(Register or I/O) = 1
D_7
= 0 (Memory-reference)
(I/0) = 1
I
= 0 (register)
(indirect) = 1
I
= 0 (direct)
T_3
Execute input-output instruction SC ← 0
T_3
Execute register-reference instruction SC ← 0
T_3
AR ← M[AR]
T_3
Nothing
Execute memory-reference instruction SC ← 0

Figure Flowchart for instruction cycle (initial configuration).

- **If D_7 = 0,** the operation code must be one of the other seven values 000 through 110, specifying a memory-reference instruction.
- **Control then inspects the value** of the first bit of the instruction, which is now available in flip-flop I. If D7 = 0 and I = 1, we have a memory reference instruction with an indirect address.
- **It is then necessary** to read the effective address from memory. The microoperation for the indirect address condition can be symbolized by the register transfer statement : AR ← M[AR]
- **Initially, AR holds the address part of the instruction**. This address is used during the memory read operation. The word at the address given by AR is read from memory and placed on the common bus. The LD input of AR is then enabled to receive the indirect address that resided in the 12 least significant bits of the memory word.
- **The three instruction types** are subdivided into four separate paths. The selected operation is activated with the clock transition associated with timing signal T_3. This can be symbolized as follows:

$D'_7 IT_3$: AR ← M[AR]

$D'_7 I'T_3$: Nothing

$D_7 I'T_3$: Execute a register-reference instruction

D_7IT_3: Execute an input-output instruction

- **When a memory-reference instruction** with I = 0 is encountered, it is not necessary to do anything since the effective address is already in AR.
- **However, the sequence counter SC** must be incremented when $D'_7T_3 = 1$, so that the execution of the memory-reference instruction can be continued with timing variable T_4
- **A register-reference or input-output instruction** can be executed with the clock associated with timing signal T_3. After the instruction is executed, SC is cleared to 0 and control returns to the fetch phase with $T_0 = 1$.

- **Note that the sequence counter SC** is either incremented or cleared to 0 with every positive clock transition. We will adopt the convention that if SC is incremented, we will not write the statement SC ← SC + 1, but it will be implied that the control goes to the next timing signal in sequence. When SC is to be cleared, we will include the statement SC ← 0.
- **The register transfers** needed for the execution of the register-reference instructions are presented in this section.

43) Register-Reference Instructions

- **Register-reference** instructions are recognized by the control when $D_7 = 1$ and $I = 0$.
- **These instructions** use bits 0 through 11 of the instruction code to specify one of 12 instructions.
- **These 12 bits** are available in IR(0-11). They were also transferred to AR during time T_2.
- **The control functions** and microoperations for the register-reference instructions are. listed in Table below. These instructions are executed with the clock transition associated with timing variable T_3.
- **Each control function** needs the Boolean relation $D_7I'T_3$, which we designate for convenience by the symbol r.
- **The control function** is distinguished by one of the bits in IR(0-11). By assigning the symbol B_i to bit i of IR, all control functions can be simply denoted by rB_i.
- **For example**, the instruction CLA has the hexadecimal code 7800, which gives the binary equivalent 0111 1000 0000 0000. The first bit is a zero and is equivalent to I'.
- **The next three bits** constitute the operation code and are recognized from decoder output D_7. Bit 11 in IR is I and is recognized from B_{11}. The control function that initiates the microoperation for this instruction is $D_7I'T_3B_{11} = rB_{11}$.
- **The execution** of a register-reference instruction is completed at time T_3.
- **The sequence counter SC** is cleared to 0 and the control goes back to fetch the next instruction with timing signal T_0.
- **The first seven register-reference** instructions perform clear, complement, circular shift, and increment microoperations on the AC or E registers.
- **The next four instructions** cause a skip of the next instruction in sequence when a stated condition is satisfied. The skipping of the instruction is achieved by incrementing PC once again (in addition, it is being incremented during the fetch phase at time T_1).
- **The condition control** statements must be recognized as part of the control conditions .
- **The AC is positive when the sign bit** in AC(15) = 0; it is negative when AC(15) = 1. The content of AC is zero (AC = 0) if all the flip-flops of the register are zero. The HLT instruction clears a start-stop flip-flop S and stops the sequence counter from counting. To restore the operation of the computer, the start-stop flip-flop must be set manually.

TABLE Execution of Register-Reference Instructions

$D_7I'T_3 = r$ (common to all register-reference instructions)
$IR(i) = B_i$ [bit in $IR(0\text{–}11)$ that specifies the operation]

	r:	$SC \leftarrow 0$	Clear SC
CLA	rB_{11}:	$AC \leftarrow 0$	Clear AC
CLE	rB_{10}:	$E \leftarrow 0$	Clear E
CMA	rB_9:	$AC \leftarrow \overline{AC}$	Complement AC
CME	rB_8:	$E \leftarrow \overline{E}$	Complement E
CIR	rB_7:	$AC \leftarrow \text{shr } AC$, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow \text{shl } AC$, $AC(0) \leftarrow E$, $E \leftarrow AC(15)$	Circulate left
INC	rB_5:	$AC \leftarrow AC + 1$	Increment AC
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$	Skip if positive
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$	Skip if negative
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1)$	Skip if AC zero
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$	Skip if E zero
HLT	rB_0:	$S \leftarrow 0$ (S is a start–stop flip-flop)	Halt computer

44) Memory-Reference Instructions

- **In order** to specify the rnicrooperations needed for the execution of each instruction, it is necessary that the function that they are intended to perform be defined precisely.
- **We will now show** that the function of the memory-reference instructions can be defined precisely by means of register transfer notation.
- **Table below** lists the seven memory-reference instructions. The decoded output D_i for i = 0, 1, 2, 3, 4, 5, and 6 from the operation decoder that belongs to each instruction is included in the table.
- **The effective** address of the instruction is in the address register AR and was placed there during timing signal T_2 when I = 0, or during timing signal T_3 when I = 1.
- **The execution** of the memory-reference instructions starts with timing signal T_4. The symbolic description of each instruction is specified in the table in terms of register transfer notation.
- **The actual execution** of the instruction in the bus system will require a sequence of microoperations.
- **This is because data** stored in memory cannot be processed directly.
- **The data** must be read from memory to a register where they can be operated on with logic circuits.
- **We now explain** the operation of each instruction and list the control functions and microoperations needed for their execution.

TABLE Memory-Reference Instructions

Symbol	Operation decoder	Symbolic description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR]$, $E \leftarrow C_{out}$
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 $PC \leftarrow PC + 1$

45) AND to AC

- **This is an instruction** that 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: DR ← M[AR]
  D_0T_5: AC ← AC ∧ DR, SC ← 0
- **The control function** for this instruction uses the operation decoder D_0 since this output of the decoder is active when the instruction has an AND operation whose binary code value is 000. Two timing signals are needed to execute the instruction.
- **The clock transition** associated with timing signal T_4 transfers the operand from memory into DR.
- **The clock transition** associated with the next timing signal T_5 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.

46) ADD to AC

- **This instruction** adds the content of the memory word specified by the effective address to the value of AC.
- **The sum is transferred into AC** and the output carry C_{out} is transferred to the E (extended accumulator) flip-flop.
- **The microoperations** needed to execute this instruction are
  D_1T_4: DR ← M[AR]
  D_1T_5: AC ← AC + DR, E ← C_{out}, SC ← 0
- **The same two timing signals**, T_4 and T_5, are used again but with operation decoder D_1 instead of D_0, which was used for the AND instruction.
- **After the instruction** is fetched from memory and decoded, only one output of the operation decoder will be active, and that output determines the sequence of rnicrooperations that the control follows during the execution of a memory-reference instruction.

47) 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: DR ← M [AR]
  D_2T_5: AC ← DR, SC ← 0
- **Note** that there is no direct path from the bus into AC (see figure under Common Bus System).
- **The adder and logic circuit** receive information from DR which can be transferred into AC.
- **Therefore**, it is necessary to read the memory word into DR first and then transfer the content of DR into AC.
- **The reason** for not connecting the bus to the inputs of AC is the delay encountered in the adder and logic circuit.
- **It is assumed** that the time it takes to read from memory and transfer the word through the bus as well as the adder and logic circuit is more than the time of one clock cycle.
- **By not connecting** the bus to the inputs of AC we can maintain one clock cycle per rnicrooperation.

48) STA: Store AC

- **This instruction stores the content of AC into the memory word specified by the effective address.** Since the output of AC is applied to the bus and the data input of memory is connected to the bus, we can execute this instruction with one microoperation:
  D_3T_4: M [AR] ← AC, SC ← 0

49) BUN: Branch Unconditionally

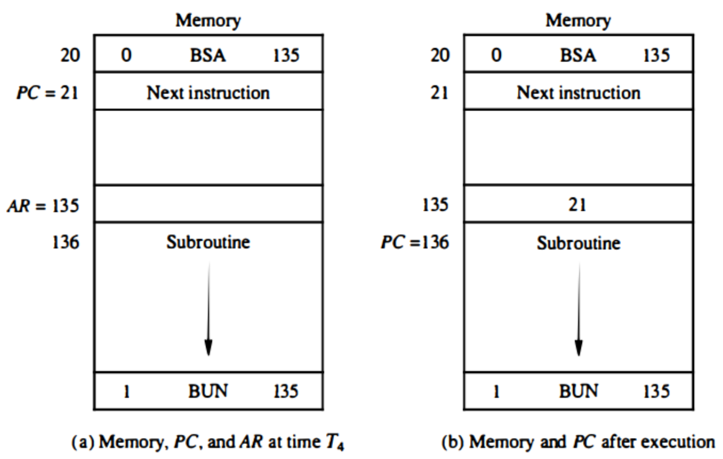
- **This instruction** transfers the program to the instruction specified by the effective address.
- **Remember** that PC holds the address of the instruction to be read from memory in the next instruction cycle.
- **PC is incremented** at time T_1 to prepare it for the address of the next instruction in the program sequence.
- **The BUN instruction** allows the programmer to specify an instruction out of sequence and we say that the program branches (or jumps) unconditionally.
- **The instruction** is executed with one microoperation:
  D_4T_4: PC ← AR, SC ← 0
- **The effective address** from AR is transferred through the common bus to PC .
- **Resetting SC to 0** transfers control to T_0. The next instruction is then fetched and executed from the memory address given by the new value in PC.

50) 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 (which is available in PC) 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.
- **This operation** was specified in Table above (see Memory-Reference Instructions) with the following register transfer:
  M[AR] ← PC, PC ← AR + 1
- **A numerical example** that demonstrates how this instruction is used with a subroutine is shown in Fig. below.
- **The BSA instruction** is assumed to be in memory at address 20. The I bit is 0 and the address part of the instruction has the binary equivalent of 135.
- **After the fetch and decode phases**, PC contains 21, which is the address of the next instruction in the program (referred to as the return address). AR holds the effective address 135.
- **This is shown in part (a) of the figure**. The BSA instruction performs the following numerical operation:
  M[135] ← 21, PC ← 135 + 1 = 136
- **The result of this operation** is shown in part (b) of the figure. The return address 21 is stored in memory location 135 and control continues with the subroutine program starting from address 136.
- **The return to the original program** (at address 21) is accomplished by means of an indirect BUN instruction placed at the end of the subroutine.
- **When this instruction is executed,** control goes to the indirect phase to read the effective address at location 135, where it finds the previously saved address 21.

- **When the BUN instruction** is executed, the effective address 21 is transferred to PC . The next instruction cycle finds PC with the value 21, so control continues to execute the instruction at the return address.
- **The BSA instruction** performs the function usually referred to as a subroutine call. The indirect BUN instruction at the end of the subroutine performs the function referred to as a subroutine return.
- **In most commercial computers**, the return address associated with a subroutine is stored in either a processor **register** or in a portion of memory called a stack. It is not possible to perform the operation of the BSA instruction in one clock cycle when we use the bus system of the basic computer.

Figure Example of BSA instruction execution.



- **To use the memory and the bus properly**, the BSA instruction must be executed With a sequence of two microoperations:
  D_5T_4: M[AR] ← PC, AR ← AR + 1
  D_5T_5: PC ← AR, SC ← 0
- **Timing signal T_4** initiates a memory write operation, places the content of PC onto the bus, and enables the INR input of AR .
- **The memory write operation** is completed and AR is incremented by the time the next clock transition occurs. The bus is used at T_5 to transfer the content of AR to PC .

51)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.
- **The programmer usually stores** a negative number (in 2's complement) in the memory word.
- **As this negative number** is repeatedly incremented by one, it eventually reaches the value of zero.
- **At that time PC** is incremented by one in order to skip the next instruction in the program.
- **Since it is not possible** to increment a word inside the memory, it is necessary to read the word into DR, increment DR, and store the word back into memory.
- **This is done** with the following sequence of microoperations:
  D_6T_4: DR ← M [AR]
  D_6T_5: DR ← DR + 1
  D_6T_6: M[AR] ← DR, if (DR = 0) then (PC ← PC + 1), SC ← 0

52) Control Flowchart

- **A flowchart** showing all microoperations for the execution of the seven memory-reference instructions is shown in Fig. below.
- **The control functions** are indicated on top of each box.
- **The microoperations** that are performed during time T_4, T_5, or T_6 depend on the operation code value. This is indicated in the flowchart by six different paths, one of which the control takes after the instruction is decoded.
- **The sequence counter SC** is cleared to 0 with the last timing signal in each case.

- **This causes a transfer of control** to timing signal T_0 to start the next instruction cycle.
- **Note** that we need only seven timing signals to execute the longest instruction (ISZ).
- **The computer** can be designed with a 3-bit sequence counter. The reason for using a 4-bit counter for SC is to provide additional timing signals for other instructions that are presented in the problems section.

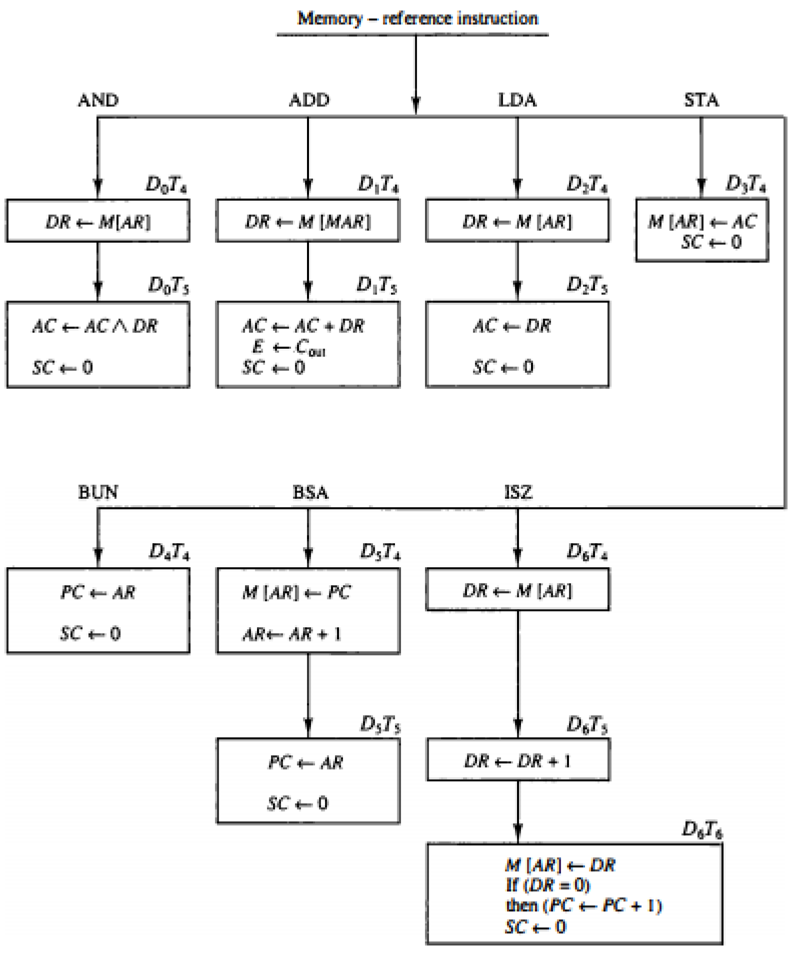


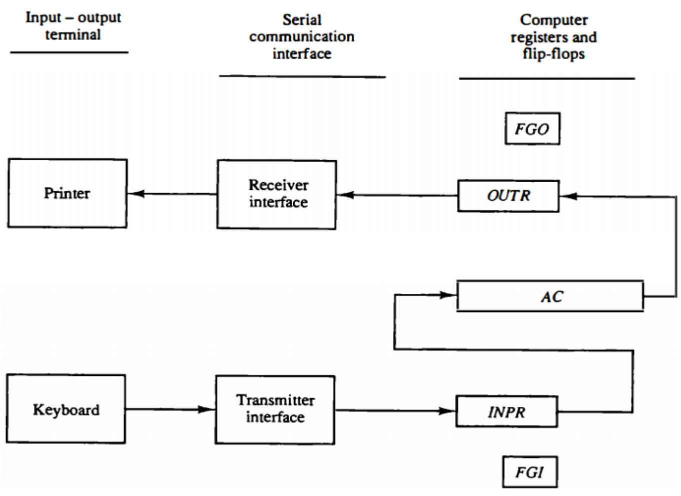
Figure Flowchart for memory-reference instructions.

53) Input-Output and Interrupt

- **A computer can serve no useful purpose** unless it communicates with the external environment. To demonstrate the most basic requirements for input and output communication, we will use as an illustration a terminal unit with a keyboard and printer.

- **Input-Output Configuration**
    1. **The terminal** sends and receives serial information. Each quantity of information has eight bits of an alphanumeric code.
    2. **The serial** information from the keyboard is shifted into the input register INPR.
    3. **The serial information** for the printer is stored in the output register OUTR.
    4. **These two registers** communicate with a communication interface serially and with the AC in parallel. The input-output configuration is shown in Fig. below. The transmitter interface receives serial information from the keyboard and transmits it to INPR.
    5. **The receiver** interface receives information from OUTR and sends it to the printer serially.
    6. **The input register INPR** consists of eight bits and holds an alphanumeric input information. The 1-bit input flag FGI is a control flip-flop. The flag bit is set to 1 when new information is available in the input device and is cleared to 0 when the information is accepted by the computer.
    7. **The flag** is needed to synchronize the timing rate difference between the input device and the computer.
    8. **The process of information** transfer is as follows. Initially, the input flag FGI is cleared to 0. When a key is struck in the keyboard, an 8-bit alphanumeric code is shifted into INPR and the input flag FGI is set to 1.
    9. **As long as the flag is set**, the information in INPR cannot be changed by striking another key. The computer checks the flag bit; if it is 1, the information from INPR is transferred in parallel into AC and FGI is cleared to 0. Once the flag is cleared, new information can be shifted into INPR by striking another key.
    10. **The output register OUTR** works similarly but the direction of information flow is reversed. Initially, the output flag FGO is set to 1.
    11. **The computer checks the flag bit**; if it is 1, the information from AC is transferred in parallel to OUTR and FGO is cleared to 0. The output device accepts the coded information, prints the corresponding character, and when the operation is completed, it sets FGO to 1.
    12. **The computer** does not load a new character into OUTR when FGO is 0 because this condition indicates that the output device is in the process of printing the character.
- **Input-Output Instructions**
    1. **Input and output instructions** are needed for transferring information to and from AC register, for checking the flag bits, and for controlling the interrupt facility.
    2. **Input-output instructions** have an operation code 1111 and are recognized by the control when $D_7 = 1$ and $I = 1$. The remaining bits of the instruction specify the particular operation.
    3. **The control functions** and microoperations for the input-output instructions are listed in Table below. These instructions are executed with the clock transition associated with timing signal T_3. Each control function needs a Boolean relation D_7IT_3, which we designate for convenience by the symbol p. The control function is distinguished by one of the bits in IR(6-11).
    4. **By assigning** the symbol B_i to bit i of IR, all control functions can be denoted by pB_i for i = 6 though 11. The sequence counter SC is cleared to 0 when $p = D_7IT_3 = 1$.

Figure Input-output configuration.



5. **The INP instruction** transfers the input information from lNPR into the eight low-order bits of AC and also clears the input flag to 0.
6. **The OUT instruction transfers** the eight least significant bits of AC into the output register OUTR and clears the output flag to 0.
7. **The next two instructions** in Table above check the status of the flags and cause a skip of the next instruction if the flag is 1.
8. **The instruction** that is skipped will normally be a branch instruction to return and check the flag again.
9. **The branch instruction** is not skipped if the flag is 0. If the flag is 1, the branch instruction is skipped and an input or output instruction is executed.
10. **The last two instructions set** and clear an interrupt enable flipflop IEN. The purpose of IEN is explained in conjunction with the interrupt operation

54)Program Interrupt

- **The process of communication** just described is referred to as programmed control transfer. The computer keeps checking the flag bit, and when it finds it set, it initiates an information transfer.
- **The difference of information flow** rate between the computer and that of the input-output device makes this type of transfer inefficient. To see why this is inefficient, consider a computer that can go through an instruction cycle in 1 μs.
- **Assume that the input-output device** can transfer information at a maximum rate of 10 characters per second. This is equivalent to one character every 100,000 μs. Two instructions are executed when the computer checks the flag bit and decides not to transfer the information.
- **This means that at the maximum rate**, the computer will check the flag 50,000 times between each transfer. The computer is wasting time while checking the flag instead of doing some other useful processing task.
- **An alternative to the programmed** controlled procedure is to let the external device inform the computer when it is ready for the transfer. In the meantime the computer can be busy with other tasks. This type of transfer uses the interrupt facility.
- **While the computer** is running a program, it does not check the flags. However, when a flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that a flag has been set.

- **The computer deviates** momentarily from what it is doing to take care of the input or output transfer. It then returns to the current program to continue what it was doing before the interrupt.
- **The interrupt enable flip-flop IEN** can be set and cleared with two instructions. When IEN is cleared to 0 (with the IOF instruction), the flags cannot interrupt the computer. When IEN is set to 1 (with the ION instruction), the computer can be interrupted.
- **These two instructions** provide the programmer with the capability of making a decision as to whether or not to use the interrupt facility.

TABLE Input-Output Instructions

$D_7IT_3 = p$ (common to all input–output instructions)
$IR(i) = B_i$ [bit in $IR(6\text{–}11)$ that specifies the instruction]

	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0\text{–}7) \leftarrow INPR, \quad FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{–}7), \quad FGO \leftarrow 0$	Output character
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$	Skip on input flag
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$	Skip on output flag
ION	pB_7:	$IEN \leftarrow 1$	Interrupt enable on
IOF	pB_6:	$IEN \leftarrow 0$	Interrupt enable off

- **The way that the interrupt** is handled by the computer can be explained by means of the flowchart of Fig. above. An interrupt flip-flop R is included in the computer. When R = 0, the computer goes through an instruction cycle.
- **During the execute phase of the instruction cycle IEN** is checked by the control. If it is 0, it indicates that the programmer does not want to use the interrupt, so control continues with the next instruction cycle. If IEN is 1, control checks the flag bits.
- **If both flags are 0**, it indicates that neither the input nor the output registers are ready for transfer of information. In this case, control continues with the next instruction cycle. If either flag is set to 1 while IEN = 1, flip-flop R is set to 1.
- **At the end of the execute phase**, control checks the value of R, and if it is equal to 1, it goes to an interrupt cycle instead of an instruction cycle. The interrupt cycle is a hardware implementation of a branch and save return address operation.
- **The return address** available in PC is stored in a specific location where it can be found later when the program returns to the instruction at which it was interrupted.
- **This location may be a processor register**, a memory stack, or a specific memory location. Here we choose the memory location at address 0 as the place for storing the return address. Control then inserts address 1 into PC and clears IEN and R so that no more interruptions can occur until the interrupt request from the flag has been serviced.
- **An example** that shows what happens during the interrupt cycle is shown in Fig. below. Suppose that an interrupt occurs and R is set to 1 while the control is executing the instruction at address 255. At this time, the return address 256 is in PC .
- **The programmer** has previously placed an input-output service program in memory starting from address 1120 and a BUN 1120 instruction at address 1. This is shown in Part (a) in Fig. below. When control reaches timing signal T_0 and finds that R = 1, it proceeds with the interrupt cycle.
- **The content of PC (256)** is stored in memory location 0, PC is set to 1, and R is cleared to 0. At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1 since this is the content of PC.
- **The branch instruction** at address 1 causes the program to transfer to the input-output service program at address 1120. This program checks the flags, determines which flag is set, and then transfers the required input or output information.

- **Once this is done**, the instruction ION is executed to set IEN to 1 (to enable further interrupts), and the program returns to the location where it was interrupted. This is shown in Part (b) in Fig. below.
- **The instruction** that returns the computer to the original place in the main program is a branch indirect instruction with an address part of 0. This instruction is placed at the end of the UO service program.
- **After** this instruction is read from memory during the fetch phase, control goes to the indirect phase (because I = 1) to read the effective address.

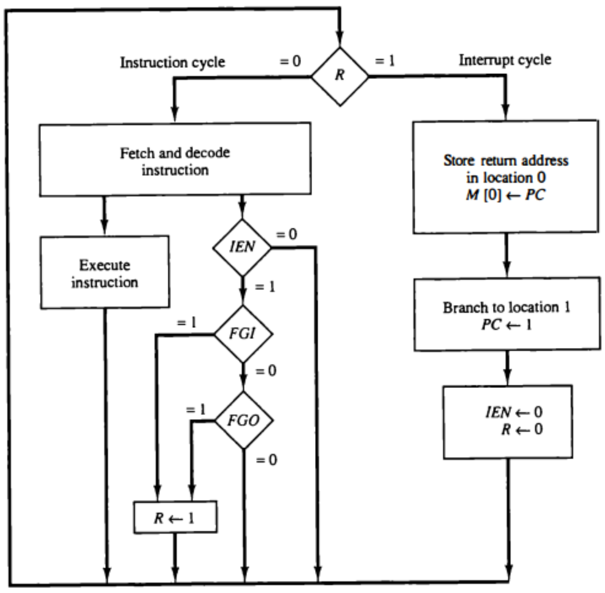


Figure Flowchart for interrupt cycle.

- **The effective address** is in location 0 and is the return address that was stored there during the previous interrupt cycle. The execution of the indirect BUN instruction results in placing into PC the return address from location 0.

55) Interrupt Cycle

- **The interrupt cycle** is initiated after the last execute phase if the interrupt flip-flop R is equal to 1. This flip-flop is set to 1 if IEN = 1 and either FGI or FGO are equal to 1.
- **This can happen** with any clock transition except when timing signals T_0, T_1 or T_2 are active. The condition for setting flip-flop R to 1 can be expressed with the following register transfer statement: T′0T′1T′2(IEN)(FGI+FGO):R←1
- **The symbol + between FGI and FGO** in the control function designates a logic OR operation. This is ANDed with IEN and T′0T′1T′2.
- **We now modify the fetch and decode phases** of the instruction cycle. Instead of using only timing signals T_0, T_1 or T_2 (as shown in Figure in Section - Determine the Type of Instruction) we will AND the three timing signals with R' so that the fetch and decode phases will be recognized from the three control functions $R'T_0$, $R'T_1$ and $R'T_2$.
- **The reason for this** is that after the instruction is executed and SC is cleared to 0, the control will go through a fetch phase only if R = 0. Otherwise, if R = 1, the control will go through an interrupt cycle.
- **The interrupt cycle** stores the return address (available in PC) into memory location 0, branches to memory location 1, and clears IEN, R, and SC to 0. This can be done with the following sequence of microoperations:

```
RT₀: AR ← 0, TR ← PC
```

RT_1: M[AR] ← TR, PC ← 0

RT_2: PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0

- **During the first timing signal AR** is cleared to 0, and the content of PC is transferred to the temporary register TR.
- **With the second timing signal**, the return address is stored in memory at location 0 and PC is cleared to 0. The third timing signal increments PC to 1, clears IEN and R, and control goes back to T_0 by clearing SC to 0.
- **The beginning** of the next instruction cycle has the condition $R'T_0$ and the content of PC is equal to 1.
- **The control** then goes through an instruction cycle that fetches and executes the BUN instruction in location 1.