Course Code: DIT-S202 **Course Name:** COMPUTER ORGANIZATION

Lecture Notes

Introduction

When working with any kind of digital electronics in which numbers are being represented, it is important to understand the different ways numbers are represented in these systems. Almost without fail, numbers are represented by two voltage levels which can represent a one or a zero (an interesting exception to this rule is the new memory device recently announced by Intel which uses one of four possible voltage levels, thereby increasing the amount of information that can be stored in a given space). The number system based on ones and zeroes is called the *bi*nary system (because there are only two possible digits). Before discussing the binary system, a review of the *dec*imal (ten possible digits) system is in order, because many of the concepts of the binary system will be easier to understand when introduced alongside their decimal counterpart.

You should all have some familiarity with the decimal system. For instance, to represent the positive integer one hundred and twenty-five as a decimal number, we can write (with the postivie sign implied). The subscript 10 denotes the number as a base 10 (decimal) number.

$125_{10} = 1*100 + 2*10 + 5*1 = 1*10^2 + 2*10^1 + 5*10^0$

The rightmost digit is multiplied by 10^0, the next digit to the left is multiplied by 10^1, and so on. Each digit to the left has a multiplier that is 10 times the previous digit. Hopefully this is all a review. Some observations:

- To multiply a number by 10 you can simply shift it to the left by one digit, and fill in the rightmost digit with a 0 (moving the decimal place one to the right). To divide a number by 10, simply shift the number to the right by one digit (moving the decimal place one to the left).
- To see how many digits a number needs, you can simply take the logarithm (base 10) of the absolute value of the number, and add 1 to it. The integer part of the result is the number of digits. For instance, $\log_{10}(33) + 1 = 2.5$. The integer part of that is 2, so 2 digits are needed.
- With n digits, 10^n unique numbers (from 0 to 10^n-1) can be represented. If n=3, 1000 (=10^3) numbers can be represented 0-999.
- Negative numbers are handled easily by simply putting a minus sign (-) in front of the number. This does lead, however, to the somewhat awkward situation where 0=-0. We will avoid this situation with binary representations, but with a little bit of effort.

Representing fractions is a simple extension of this idea. To wit,

$25.43_{10} = 2*10 + 5*1 + 4*0.1 + 3*0.01 = 2*10^1 + 5*10^0 + 4*10^{-1} + 3*10^{-2}$

The only pertinent observations here are:

- If there are m digits to the right of the decimal point, the smallest number that can be represented is 10^{-m}. For instance if m=4, the smallest number that can be represented is $0.0001=10^{-4}$.

After reading this document you might want to learn something about binary arithmetic.

Binary Representation of positive integers

Binary representations of positive can be understood in the same way as their decimal counterparts. For example

$86_{10} = 1*64 + 0*32 + 1*16 + 0*8 + 1*4 + 1*2 + 0*1$

or

$86_{10} = 1* 2^6 + 0* 2^5 + 1* 2^4 + 0* 2^3 + 1* 2^2 + 1* 2^1 + 0* 2^0$

or

$86_{10} = 1010110_2$

The subscript 2 denotes a binary number. Each digit in a binary number is called a bit. The number 1010110 is represented by 7 bits. Any number can be broken down this way, by finding all of the powers of 2 that add up to the number in question (in this case 2^6, 2^4, 2^2 and 2^1). You can see this is exactly analagous to the decimal deconstruction of the number 125 that was done earlier. Likewise, we can make a similar set of observations:

- To multiply a number by 2 you can simply shift it to the left by one digit, and fill in the rightmost digit with a 0. To divide a number by 2, simply shift the number to the right by one digit.
- To see how many digits a number needs, you can simply take the logarithm (base 2) of the number, and add 1 to it. The integer part of the result is the number of digits. For instance,
  $\log_2(86) + 1 = 7.426$.
  The integer part of that is 7, so 7 digits are needed.
- With n digits, 2^n unique numbers (from 0 to 2^n-1) can be represented. If n=8, 256 (=2^8) numbers can be represented 0-255.

Hexadecimal, Octal, Bits, Bytes and Words.

It is often convenient to handle groups of bits, rather than individually. The most common grouping is 8 bits, which forms a byte. A single byte can represent 256 (2^8) numbers. Memory capacity is usually referred to in bytes. Two bytes is usually called a word, or short word (though word-length depends on the application). A two-byte word is also the size that is usually used to represent integers in programming languages. A long word is usually twice as long as a word. A less common unit is the nibble which is 4 bits, or half of a byte.

It is cumbersome for humans to deal with writing, reading and remembering individual bits, because it takes many of them to represent even fairly small numbers. A number of different ways have been developed to make the handling of binary data easier for us. The most common is hexadecimal. In hexadecimal notation, 4 bits (a nibble) are represented by a single digit. There is obviously a problem with this since 4 bits gives 16 possible combinations, and there are only 10 unique decimal digits, 0 to 9. This is solved by using the first 6 letters (A.. F) of the alphabet as numbers.

There are some significant advantages to using hexadecimal when dealing with electronic representations of numbers (if people had 16 fingers, we wouldn't be saddled with the awkward decimal system). Using hexadecimal makes it very easy to convert back and forth from binary because each hexadecimal digit corresponds to exactly 4 bits ($\log_2(16) = 4$) and each byte is two hexadecimal digit. In contrast, a decimal digit corresponds to $\log_2(10) = 3.322$ bits and a byte is 2.408 decimal digits. Clearly hexadecimal is better suited to the task of representing binary numbers than is decimal.

As an example, the number $CA3_{16} = 1100\ 1010\ 0011_2$ ($1100_2 = C_{16}$, $1010_2 = A_{16}$, $0011_2 = 3_{16}$). It is convenient to write the binary number with spaces after every fourth bit to make it easier to read. Converting back and forth to decimal is more difficult, but can be done in the same way as before.

$3235_{10} = C_{16}*256 + A_{16}*16 + 3_{16}*1 = C_{16}*16^2 + A_{16}*16^1 + 3_{16}*16^0$

or

$3235_{10} = 12*256 + 10*16 + 3*1 = 12*16^2 + 10*16^1 + 3*16^0$

Octal notation is yet another compact method for writing binary numbers. There are 8 octal characters, 0...7. Obviously this can be represented by exactly 3 bits. Two octal digits can represent numbers up to 64, and three octal digits up to 512. A byte requires 2.667 octal digits. Octal used to be quiete common, it was the primary way of doing low level I/O on some old DEC computers. It is much less common today but is still used occasionally (e.g., to set read, write and execute permissions on Unix systems)

Signed Binary Integers

It was noted previously that we will not be using a minus sign (-) to represent negative numbers. We would like to represent our binary numbers with only two symbols, 0 and 1. There are a few ways to represent negative binary numbers. The simplest of these methods is called ones complement, where the sign of a binary number is changed by simply toggling each bit (0's become 1's and vice-versa). This has some difficulties, among them the fact that zero can be represented in two different ways (for an eight bit number these would be 0000 0000 and 1111 1111)., we will use a method called two's complement notation which avoids the pitfalls of one's complement, but which is a bit more complicated.

To represent an n bit signed binary number the leftmost bit, has a special significance. The difference between a signed and an unsigned number is given in the table below for an 8 bit number.

The value of bits in signed and unsigned binary numbers								
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
unsigned	$2^7 = 128$	$^6 = 64$	$2^5 = 32$	$^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$^0 = 1$
Signed	$-(2^7) = -128$	$^6 = 64$	$2^5 = 32$	$^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$

Let's look at how this changes the value of some binary numbers

Binary	Unsigned	Signed
0010 0011	35	35
1010 0011	163	-93
1111 1111	255	-1
1000 0000	128	-128

If Bit 7 is not set (as in the first example) the representation of signed and unsigned numbers is the same. However, when Bit 7 is set, the number is always negative. For this reason Bit 7 is sometimes called the sign bit. Signed numbers are added in the same way as unsigned numbers, the only difference is in the way they are interpreted. This is important for designers of arithmetic circuitry because it means that numbers can be added by the same circuitry regardless of whether or not they are signed.

To form a two's complement number that is negative you simply take the corresponding positive number, invert all the bits, and add 1. The example below illustrated this by forming the number negative 35 as a two's complement integer:

$35_{10} = 0010\ 0011_2$

invert -> $1101\ 1100_2$

add 1 -> $1101\ 1101_2$

So 1101 1101 is our two's complement representation of -35. We can check this by adding up the contributions from the individual bits

$1101\ 1101_2 = -128 + 64 + 0 + 16 + 8 + 4 + 0 + 1 = -35$.

The same procedure (invert and add 1) is used to convert the negative number to its positive equivalent. If we want to know what what number is represented by 1111 1101, we apply the procedure again

? = 1111 1101_2

invert -> 0000 0010_2

add 1 -> 0000 0011_2

Since 0000 0011 represents the number 3, we know that 1111 1101 represents the number -3.

Note that a number can be extended from 4 bits to 8 bits by simply repeating the leftmost bit 4 times. Consider the following examples

Decimal	**4 bit**	**8 bit**
3	0011	0000 0011
-3	1101	1111 1101
7	0111	0000 0111
-5	1011	1111 1011

Let's carefully consider the last case which uses the number -5. As a 4 bit number this is represented as

1011 = -8 + 2 + 1 = -5

The 8 bit number is

1111 1011 = -128 + 64 + 32 + 16 + 8 + 2 + 1 = -5.

It is clear that in the second case the sum of the contributions from the leftmost 5 bits (-128 + 64 + 32 + 16 + 8 = -8) is the same as the contribution from the leftmost bit in the 4-bit representation (-8)

This process is refered to as sign-extension, and can be applied whenever a number is to be represented by a larger number of bits. In the 320C50 Digital Signal Processor, this typically occurs when moving a number from a 16 bit register to a 32 bit register. Whether or not sign-extension is applied during such a move is determined by the sign-extension mode bit. Note that to store a 32 bit number in 16 bits you can simply truncate the upper 16 bits (as long as they are all the same as the left-most bit in the resulting 16 bit number - i.e., the sign doesn't change).

Most processors even have two separate instructions for shifting numbers to the right (which, you will recall, is equivalent to dividing the number in half). The first instruction is something like LSR (Logical Shift Right) which simply shifts the bits to the right and usually fills a zero in as the leftmost bit. The second instruction is something like ASR (Arithmetic Shift Right), which shifts all of the bits to the right, while keeping the leftmost bit unchanged. With ASR 1010 (-6) becomes 1101 (-3). Of course, there is only one instruction for a left shift (since LSL is equivalent to ASL).

Positive binary fractions

The representation of unsigned binary fractions proceeds in exactly the same way as decimal fractions. For example

$0.625_{10} = 1*0.5 + 0*0.25 + 1*0.125 = 1* 2^{-1} + 0* 2^{-2} + 1* 2^{-3} = 0.101_2$

Each place to the right of the decimal point represents a negative power of 2, just as for decimals they represent a negative power of 10. Likewise, if there are m bits to the right of a decimal, the precision of the number is 2^{-m} (versus 10^{-m} for decimal). Though it is possible to represent numbers greater than one by having digits to the left of the decimal place we will restrict ourselves to numbers less than one. These are commonly used by Digital Signal Processors.

The largest number that can be represented by such a representation is $1\text{-}2^{-m}$, the smallest number is 2^{-m}. For a fraction with 15 bits of resolution this gives a range of approximately 0.99997 to 3.05E-5.

Note that this representation is easily extended to represent all positive numbers by having the digits to the left of the decimal point represent the integer part, and the digits to the right representing the fractional part. Thus

$6.625_{10} = 110.101_2$

Signed binary fractions

Signed binary fractions are formed much like signed integers. We will work with a single digit to the left of the decimal point, and this will represent the number -1 (= -(2^0)). The rest of the representation of the fraction remains unchanged. Therefore, this leftmost bit represents a sign bit just as with two's complement integers. If this bit is set, the number is negative, otherwise the number is positive. The largest positive number that can be represented is still $1\text{-}2^{-m}$ but the largest negative number is -1. The resolution is still $1\text{-}2^{-m}$.

There is a terminology for naming the resolution of signed fractions. If there are m bits to the right of the decimal point, the number is said to be in Q*m* format. For a 16 bit number (15 bits to the right of the decimal point) this results in Q15 notation.

Signed binary fractions are easily extended to include all numbers by representing the number to the left of the decimal point as a 2's complement integer, and the number to the right of the decimal point as a positive fraction. Thus

$-6.625_{10} = (-7+0.375)_{10} = 1001.011_2$

Note, that as with two's complement integers, the leftmost digit can be repeated any number of times without affecting the value of the number.

A Quicker Method for Converting Binary Fractions.

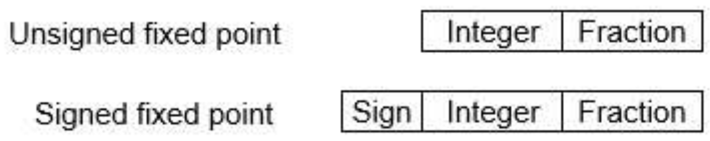
Another way to convert Q*m* numbers to decimal is to represent the binary number as a signed integer, and to divide by 2^m. To convert a decimal number to Q^m, multiply the number by 2^m and take the rightmost m digits. Note, this simply truncates the number; it is more elegant, and accurate, but slightly more complicated, to round the number.

Examples (all Q7 numbers)**:**

Convert 0.100 1001 to decimal.	Take the binary number 0100 1001 (=73_{10}), and divide by 2^7=128. The answer is 73/128=0.5703125, which agrees with the result of the previous exercise (Positive Binary Fractions).
Convert 1.100 1001 to decimal.	Take the two's complements binary number 1100 1001 (=-55_{10}), and divide 128. The answer is -0.4296875, which agrees with the result of the previous exercise (Signed Binary Fractions).
Convert 0.9 to Q7 format	Multiply 0.9 by 128 to get 115.2. This is represented in binary as 111 0011, so the Q7 representation is 0.111 0011. This agrees with the result of the previous exercise (Positive Binary Fractions).
Convert -0.9 to Q7 format	ultiply -0.9 by 128 to get -115.2. The Q7 representation is 1.000 1101. This agrees with the result of the previous exercise (Signed Binary Fractions).

Fixed-Point Representation −

This representation has fixed number of bits for integer part and for fractional part. For example, if given fixed-point representation is IIII.FFFF, then you can store minimum value is 0000.0001 and maximum value is 9999.9999. There are three parts of a fixed-point number representation: the sign field, integer field, and fractional field.



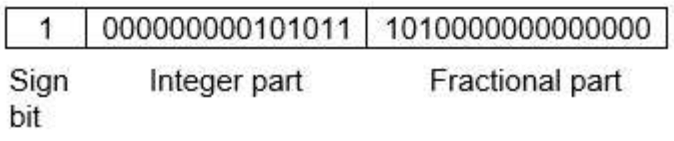
We can represent these numbers using:

- Signed representation: range from -($2^{(k-1)}$-1) to ($2^{(k-1)}$-1), for k bits.
- 1's complement representation: range from -($2^{(k-1)}$-1) to ($2^{(k-1)}$-1), for k bits.
- 2's complementation representation: range from -($2^{(k-1)}$) to ($2^{(k-1)}$-1), for k bits.

2's complementation representation is preferred in computer system because of unambiguous property and easier for arithmetic operations.

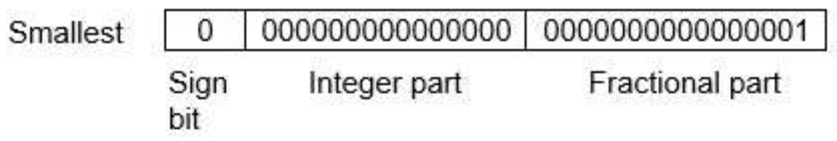
Example −Assume number is using 32-bit format which reserve 1 bit for the sign, 15 bits for the integer part and 16 bits for the fractional part.

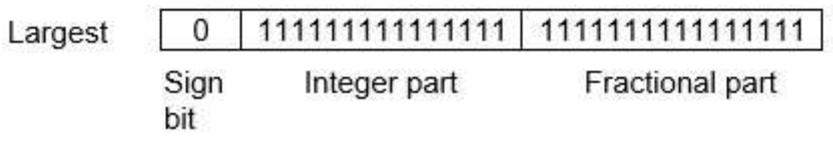
Then, -43.625 is represented as following:



Where, 0 is used to represent + and 1 is used to represent. 000000000101011 is 15 bit binary value for decimal 43 and 1010000000000000 is 16 bit binary value for fractional 0.625.

The advantage of using a fixed-point representation is performance and disadvantage is relatively limited range of values that they can represent. So, it is usually inadequate for numerical analysis as it does not allow enough numbers and accuracy. A number whose representation exceeds 32 bits would have to be stored inexactly.





These are above smallest positive number and largest positive number which can be store in 32-bit representation as given above format. Therefore, the smallest positive number is $2^{-16} \approx$ 0.000015 approximate and the largest positive number is $(2^{15}-1)+(1-2^{-16})=2^{15}(1-2^{-16})$ =32768, and gap between these numbers is 2^{-16}.

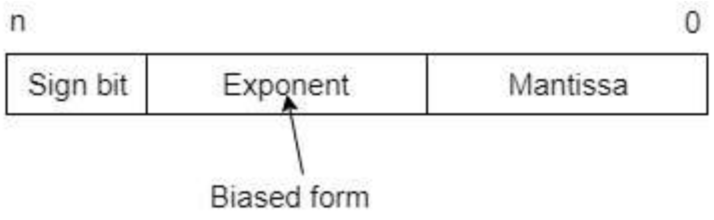
We can move the radix point either left or right with the help of only integer field is 1.

Floating-Point Representation −

This representation does not reserve a specific number of bits for the integer part or the fractional part. Instead it reserves a certain number of bits for the number (called the mantissa or significand) and a certain number of bits to say where within that number the decimal place sits (called the exponent).

The floating number representation of a number has two part: the first part represents a signed fixed point number called mantissa. The second part of designates the position of the decimal (or binary) point and is called the exponent. The fixed point mantissa may be fraction or an integer. Floating -point is always interpreted to represent a number in the following form: Mxr^e.

Only the mantissa m and the exponent e are physically represented in the register (including their sign). A floating-point binary number is represented in a similar manner except that is uses base 2 for the exponent. A floating-point number is said to be normalized if the most significant digit of the mantissa is 1.



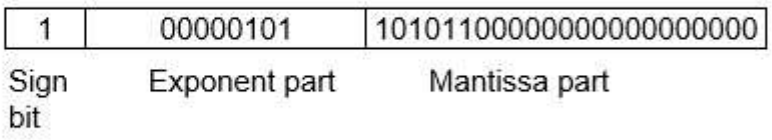
So, actual number is $(-1)^s(1+m)x2^{(e-Bias)}$, where *s* is the sign bit, *m* is the mantissa, *e* is the exponent value, and *Bias* is the bias number.

Note that signed integers and exponent are represented by either sign representation, or one's complement representation, or two's complement representation.

The floating point representation is more flexible. Any non-zero number can be represented in the normalized form of $\pm(1.b_1b_2b_3 \ldots)_2x2^n$ This is normalized form of a number x.

Example −Suppose number is using 32-bit format: the 1 bit sign bit, 8 bits for signed exponent, and 23 bits for the fractional part. The leading bit 1 is not stored (as it is always 1 for a normalized number) and is referred to as a *"hidden bit*".

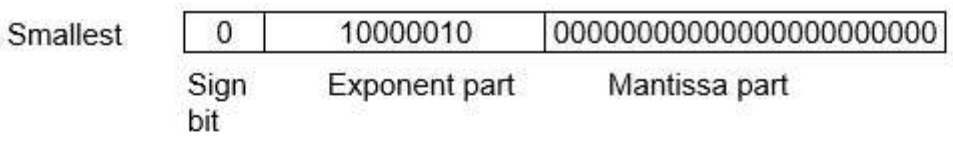
Then −53.5 is normalized as $-53.5=(-110101.1)_2=(-1.101011)x2^5$, which is represented as following below,

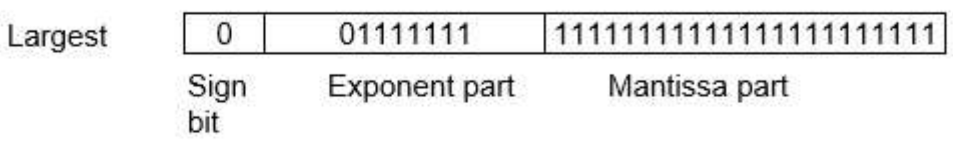


Where 00000101 is the 8-bit binary value of exponent value +5.

Note that 8-bit exponent field is used to store integer exponents $-126 \leq n \leq 127$.

The smallest normalized positive number that fits into 32 bits is $(1.00000000000000000000000)_2x2^{-126}=2^{-126}\approx1.18x10^{-38}$, and largest normalized positive number that fits into 32 bits is $(1.11111111111111111111111)_2x2^{127}=(2^{24}-1)x2^{104} \approx 3.40x10^{38}$. These numbers are represented as following below,





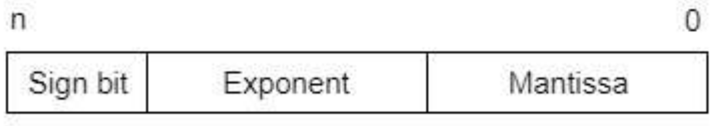
The precision of a floating-point format is the number of positions reserved for binary digits plus one (for the hidden bit). In the examples considered here the precision is 23+1=24.

The gap between 1 and the next normalized floating-point number is known as machine epsilon. the gap is $(1+2^{-23})-1=2^{-23}$for above example, but this is same as the smallest positive floating-point number because of non-uniform spacing unlike in the fixed-point scenario.

Note that non-terminating binary numbers can be represented in floating point representation, e.g., 1/3 = $(0.010101 ...)_2$ cannot be a floating-point number as its binary representation is non-terminating.

IEEE Floating point Number Representation −

IEEE (Institute of Electrical and Electronics Engineers) has standardized Floating-Point Representation as following diagram.



So, actual number is $(-1)^s(1+m)x2^{(e-Bias)}$, where *s* is the sign bit, *m* is the mantissa, *e* is the exponent value, and *Bias* is the bias number. The sign bit is 0 for positive number and 1 for negative number. Exponents are represented by or two's complement representation.

According to IEEE 754 standard, the floating-point number is represented in following ways:

- Half Precision (16 bit): 1 sign bit, 5 bit exponent, and 10 bit mantissa
- Single Precision (32 bit): 1 sign bit, 8 bit exponent, and 23 bit mantissa
- Double Precision (64 bit): 1 sign bit, 11 bit exponent, and 52 bit mantissa
- Quadruple Precision (128 bit): 1 sign bit, 15 bit exponent, and 112 bit mantissa

Special Value Representation −

There are some special values depended upon different values of the exponent and mantissa in the IEEE 754 standard.

- All the exponent bits 0 with all mantissa bits 0 represents 0. If sign bit is 0, then +0, else -0.
- All the exponent bits 1 with all mantissa bits 0 represents infinity. If sign bit is 0, then $+\infty$, else $-\infty$.
- All the exponent bits 0 and mantissa bits non-zero represents denormalized number.
- All the exponent bits 1 and mantissa bits non-zero represents error.

We know that the bits 0 and 1 corresponding to two different range of analog voltages. So, during transmission of binary data from one system to the other, the noise may also be added. Due to this, there may be errors in the received data at other system.

That means a bit 0 may change to 1 or a bit 1 may change to 0. We can't avoid the interference of noise. But, we can get back the original data first by detecting whether any error present and then correcting those errors. For this purpose, we can use the following codes.

- Error detection codes
- Error correction codes

Error detection codes − are used to detect the error present in the received data. These codes contain some bit, which are included to the original bit stream. These codes detect the error, if it is occurred during transmission of the original data. **Example** − Parity code, Hamming code.

Error correction codes − are used to correct the error present in the received data so that, we will get the original data. Error correction codes also use the similar strategy of error detection codes. **Example** − Hamming code.

Therefore, to detect and correct the errors, additional bit is appended to the data bits at the time of transmission.

Parity Code

It is easy to include one parity bit either to the left of MSB or to the right of LSB of original bit stream. There are two types of parity codes, namely even parity code and odd parity code based on the type of parity being chosen.

Even Parity Code

The value of even parity bit should be zero, if even number of ones present in the binary code. Otherwise, it should be one. So that, even number of ones present in **even parity code**. Even parity code contains the data bits and even parity bit.

The following table shows the **even parity codes** corresponding to each 3-bit binary code. Here, the even parity bit is included to the right of LSB of binary code.

Binary Code	Even Parity bit	Even Parity Code
000	0	0000
001	1	0011
010	1	0101
011	0	0110
100	1	1001
101	0	1010
110	0	1100
111	1	1111

Here, the number of bits present in the even parity codes is 4. So, the possible even number of ones in these even parity codes are 0, 2 & 4.

- If the other system receives one of these even parity codes, then there is no error in the received data. The bits other than even parity bit are same as that of binary code.
- If the other system receives other than even parity codes, then there will be an error in the received data. In this case, we can't predict the original binary code because we don't know the bit position of error.

Therefore, even parity bit is useful only for detection of error in the received parity code. But, it is not sufficient to correct the error.

Odd Parity Code

The value of odd parity bit should be zero, if odd number of ones present in the binary code. Otherwise, it should be one. So that, odd number of ones present in **odd parity code**. Odd parity code contains the data bits and odd parity bit.

The following table shows the **odd parity codes** corresponding to each 3-bit binary code. Here, the odd parity bit is included to the right of LSB of binary code.

Binary Code	Odd Parity bit	Odd Parity Code

000	1	0001
001	0	0010
010	0	0100
011	1	0111
100	0	1000
101	1	1011
110	1	1101
111	0	1110

Here, the number of bits present in the odd parity codes is 4. So, the possible odd number of ones in these odd parity codes are 1 & 3.

- If the other system receives one of these odd parity codes, then there is no error in the received data. The bits other than odd parity bit are same as that of binary code.
- If the other system receives other than odd parity codes, then there is an error in the received data. In this case, we can't predict the original binary code because we don't know the bit position of error.

Therefore, odd parity bit is useful only for detection of error in the received parity code. But, it is not sufficient to correct the error.

Hamming Code

Hamming code is a block code that is capable of detecting up to two simultaneous bit errors and correcting single-bit errors. It was developed by R.W. Hamming for error correction.

In this coding method, the source encodes the message by inserting redundant bits within the message. These redundant bits are extra bits that are generated and inserted at specific positions in the message itself to enable error detection and correction. When the destination receives this message, it performs recalculations to detect errors and find the bit position that has error.

Encoding a message by Hamming Code

The procedure used by the sender to encode the message encompasses the following steps −

- **Step 1** − Calculation of the number of redundant bits.
- **Step 2** − Positioning the redundant bits.
- **Step 3** − Calculating the values of each redundant bit.

Once the redundant bits are embedded within the message, this is sent to the user.

Step 1 − Calculation of the number of redundant bits.

If the message contains *mm*number of data bits, *rr*number of redundant bits are added to it so that *mr* is able to indicate at least ($m + r$+ 1) different states. Here, ($m + r$) indicates location of an error in each of ($m + r$) bit positions and one additional state indicates no error. Since, *rr* bits can indicate $2^r r$ states, $2^r r$ must be at least equal to ($m + r + 1$). Thus the following equation should hold $2^r \geq m+r+1$

Step 2 − Positioning the redundant bits.

The *r* redundant bits placed at bit positions of powers of 2, i.e. 1, 2, 4, 8, 16 etc. They are referred in the rest of this text as r_1 (at position 1), r_2 (at position 2), r_3 (at position 4), r_4 (at position 8) and so on.

Step 3 − Calculating the values of each redundant bit.

The redundant bits are parity bits. A parity bit is an extra bit that makes the number of 1s either even or odd. The two types of parity are −

- **Even Parity** − Here the total number of bits in the message is made even.
- **Odd Parity** − Here the total number of bits in the message is made odd.

Each redundant bit, r_i, is calculated as the parity, generally even parity, based upon its bit position. It covers all bit positions whose binary representation includes a 1 in the i^{th} position except the position of r_i. Thus −

- r_1 is the parity bit for all data bits in positions whose binary representation includes a 1 in the least significant position excluding 1 (3, 5, 7, 9, 11 and so on)
- r_2 is the parity bit for all data bits in positions whose binary representation includes a 1 in the position 2 from right except 2 (3, 6, 7, 10, 11 and so on)
- r_3 is the parity bit for all data bits in positions whose binary representation includes a 1 in the position 3 from right except 4 (5-7, 12-15, 20-23 and so on)

Decoding a message in Hamming Code

Once the receiver gets an incoming message, it performs recalculations to detect errors and correct them. The steps for recalculation are −

- **Step 1** − Calculation of the number of redundant bits.
- **Step 2** − Positioning the redundant bits.
- **Step 3** − Parity checking.
- **Step 4** − Error detection and correction

Step 1 − Calculation of the number of redundant bits

Using the same formula as in encoding, the number of redundant bits are ascertained.

$2^r \geq m + r + 1$ where *m* is the number of data bits and *r* is the number of redundant bits.

Step 2 − Positioning the redundant bits

The *r* redundant bits placed at bit positions of powers of 2, i.e. 1, 2, 4, 8, 16 etc.

Step 3 − Parity checking

Parity bits are calculated based upon the data bits and the redundant bits using the same rule as during generation of c_1,c_2 ,c_3 ,c_4 etc. Thus

c_1 = parity(1, 3, 5, 7, 9, 11 and so on)

c_2 = parity(2, 3, 6, 7, 10, 11 and so on)

c_3 = parity(4-7, 12-15, 20-23 and so on)

Step 4 − Error detection and correction

The decimal equivalent of the parity bits binary values is calculated. If it is 0, there is no error. Otherwise, the decimal value gives the bit position which has error. For example, if $c_1c_2c_3c_4$ = 1001, it implies that the data bit at position 9, decimal equivalent of 1001, has error. The bit is flipped to get the correct message.

Digital computer generations

In the electronic computer world, we measure technological advancement by generations. A specific system is said to belong to a specific "generation." Each generation indicates a significant change in computer design. The UNIVAC I represents the first generation. Currently we are moving toward the fourth generation.

FIRST GENERATION

The computers of the first generation (1951-1958) were physically very large machines characterized by the **vacuum tube**. Because they used vacuum tubes, they were very unreliable, required a lot of power to run, and produced so much heat that adequate air conditioning was critical to protect the computer parts. Compared to today's computers, they had slow input and output devices, were slow in processing, and had small storage capacities. Many of the internal processing functions were measured in thousandths of a second (millisecond). The software (computer program) used on first generation computers was unsophisticated and machine oriented. This meant that the programmers had to code all computer instructions and data in actual machine language. They also had to keep track of where instructions and data were stored in memory. Using such a **machine language** was efficient for the computer but difficult for the programmer.

- ***Advantages:***

1. It made use of vacuum tubes which are the only electronic component available during those days.
2. These computers could calculate in milliseconds.

- ***Disadvantages:***

1. These were very big in size; weight was about 30 tones.
2. These computers were based on vacuum tubes.
3. These computers were very costly.
4. It could store only a small amount of information due to the presence of magnetic drums.
5. As the invention of first generation computers involves vacuum tubes, so another disadvantage of these computers was, vacuum tubes require a large cooling system.
6. Very less work efficiency.
7. Limited programming capabilities and punch cards were used to take inputs.
8. Large amount of energy consumption.
9. Not reliable and constant maintenance is required.

SECOND GENERATION

The computers of the second generation (1959-1963), were characterized by **transistors** instead of vacuum tubes. Transistors were smaller, less expensive, generated almost no heat, and required very little power. Thus second generation computers were smaller, required less power, and produced a lot less heat. The use of small, long lasting transistors also increased processing speeds and reliability. Cost performance also improved. The storage capacity was greatly increased with the introduction of magnetic disk storage and the use of magnetic cores for main storage. High speed card readers, printers, and magnetic tape units were also introduced. Internal processing speeds increased. Functions were measured in millionths of a second (microseconds). Like the first generation, a particular computer of the second generation was designed to process either scientific or business oriented problems but not both. The software was also improved. Symbolic machine languages or assembly languages were used instead of actual machine languages. This allowed the programmer to use mnemonic operation codes for instruction operations and symbolic names for storage locations or stored variables. Compiler languages were also developed for the second generation computers.

- ***Advantages:***

1. Due to the presence of transistors instead of vacuum tubes, the size of electron component decreased. This resulted in reducing the size of a computer as compared to first generation computers.
2. Less energy and not produce as much heat as the first generation.
3. Assembly language and punch cards were used for input.
4. Low cost than first generation computers.
5. Better speed, calculate data in microseconds.
6. Better portability as compared to first generation

- ***Disadvantages:***

1. A cooling system was required.
2. Constant maintenance was required.
3. Only used for specific purposes.

THIRD GENERATION

The computers of this generation (1964-1970), many of which are still in use, are characterized by **miniaturized circuits**. This reduces the physical size of computers even more and increases their durability and internal processing speeds. One design employs solid-state logic microcircuits for which conductors, resistors, diodes, and transistors have been miniaturized and combined on half-inch ceramic squares. Another smaller design uses silicon wafers on which the circuit and its components are etched. The smaller circuits allow for faster internal processing speeds resulting in faster execution of instructions. Internal processing speeds are measured in billionths of a second (nanoseconds). The faster computers make it possible to run jobs that were considered impractical or impossible on first or second generation equipment. Because the miniature components are more reliable, maintenance is reduced. New mass storage, such as the data cell, was introduced during this generation, giving a storage capacity of over 100 million characters. Drum and disk capacities and speed have been increased, the portable disk pack has been developed, and faster, higher density magnetic tapes have come into use. Considerable improvements were made to card readers and printers, while the overall cost has been greatly reduced. Applications using online processing, real-time processing, time sharing, multiprogramming, multiprocessing, and teleprocessing have become widely accepted.

Manufacturers of third generation computers are producing a series of similar and compatible computers. This allows programs written for one computer model to run on most larger models of the same series. Most third generation systems are designed to handle both scientific and business data processing applications. Improved program and operating software has been designed to provide better control, resulting in faster processing. These enhancements are of significant importance to the computer operator. They simplify system initialization (booting) and minimize the need for inputs to the program from a keyboard (console intervention) by the operator.

- ***Advantages:***

1. These computers were cheaper as compared to second-generation computers.
2. They were fast and reliable.
3. Use of IC in the computer provides the small size of the computer.
4. IC not only reduce the size of the computer but it also improves the performance of the computer as compared to previous computers.
5. This generation of computers has big storage capacity.
6. Instead of punch cards, mouse and keyboard are used for input.
7. They used an operating system for better resource management and used the concept of time-sharing and multiple programming.

8. These computers reduce the computational time from microseconds to nanoseconds.

- ***Disadvantages:***

1. IC chips are difficult to maintain.
2. The highly sophisticated technology required for the manufacturing of IC chips.
3. Air conditioning is required.

FOURTH GENERATION

The computers of the fourth generation are not easily distinguished from earlier generations, yet there are some striking and important differences. The manufacturing of integrated circuits has advanced to the point where thousands of circuits (active components) can be placed on a silicon wafer only a fraction of an inch in size (the computer on a chip). This has led to what is called large scale integration (LSI) and very large scale integration (VLSI). As a result of this technology, computers are significantly smaller in physical size and lower in cost. Yet they have retained large memory capacities and are ultra fast. Large mainframe computers are increasingly complex. Medium sized computers can perform the same tasks as large third generation computers. An entirely new breed of computers called **microcomputers** and **minicomputers** are small and inexpensive, and yet they provide a large amount of computing power.

- ***Advantages:***

1. Fastest in computation and size get reduced as compared to the previous generation of computer.
2. Heat generated is negligible.
3. Small in size as compared to previous generation computers.
4. Less maintenance is required.
5. All types of high-level language can be used in this type of computers.

- ***Disadvantages:***

1. The Microprocessor design and fabrication are very complex.
2. Air conditioning is required in many cases due to the presence of ICs.
3. Advance technology is required to make the ICs.

FIFTH GENERATION

The period of the fifth generation in 1980-onwards. This generation is based on artificial intelligence. The aim of the fifth generation is to make a device which could respond to natural language input and are capable of learning and self-organization.This generation is based on ULSI(Ultra Large Scale Integration) technology resulting in the production of microprocessor chips having ten million electronic component. Few Examples are Desktop, Laptop, NoteBook, UltraBook. Chromebook.

- ***Advantages:***

1. It is more reliable and works faster.
2. It is available in different sizes and unique features.
3. It provides computers with more user-friendly interfaces with multimedia features.

- ***Disadvantages:***

1. They need very low-level languages.
2. They may make the human brains dull and doomed.

Computer types and Classification

The computer systems can be classified on the following basis:

1. On the basis of size.
2. On the basis of functionality.

3. On the basis of data handling.

Classification on the basis of size

1. **Super computers:** The super computers are the most high performing system. A supercomputer is a computer with a high level of performance compared to a general-purpose computer. The actual Performance of a supercomputer is measured in FLOPS instead of MIPS. All of the world's fastest 500 supercomputers run Linux-based operating systems. Additional research is being conducted in China, the US, the EU, Taiwan and Japan to build even faster, more high performing and more technologically superior supercomputers. Supercomputers actually play an important role in the field of computation, and are used for intensive computation tasks in various fields, including quantum mechanics, weather forecasting, climate research, oil and gas exploration, molecular modeling, and physical simulations. and also Throughout the history, supercomputers have been essential in the field of the cryptanalysis.

   eg: PARAM, jaguar, roadrunner.
2. **Mainframe computers:** These are commonly called as big iron, they are usually used by big organisations for bulk data processing such as statics, census data processing, transaction processing and are widely used as the servers as these systems has a higher processing capability as compared to the other classes of computers, most of these mainframe architectures were established in 1960s, the research and development worked continuously over the years and the mainframes of today are far more better than the earlier ones, in size, capacity and efficiency.

   Eg: IBM z Series, System z9 and System z10 servers.
3. **Mini computers:** These computers came into the market in mid 1960s and were sold at a much cheaper price than the main frames, they were actually designed for control, instrumentation, human interaction, and communication switching as distinct from calculation and record keeping, later they became very popular for personal uses with evolution.

   In the 60s to describe the smaller computers that became possible with the use of transistors and core memory technologies, minimal instructions sets and less expensive peripherals such as the ubiquitous Teletype Model 33 ASR.They usually took up one or a few inch rack cabinets, compared with the large mainframes that could fill a room, there was a new term "MINICOMPUTERS" coined

   Eg: Personal Laptop, PC etc.
4. **Micro computers:** A microcomputer is a small, relatively inexpensive computer with a microprocessor as its CPU. It includes a microprocessor, memory, and minimal I/O circuitry mounted on a single printed circuit board. The previous to these computers, mainframes and minicomputers, were comparatively much larger, hard to maintain and more expensive. They actually formed the foundation for present day microcomputers and smart gadgets that we use in day to day life.

   Eg: Tablets, Smartwatches.

Classification on the basis of functionality

1. **Servers :** Servers are nothing but dedicated computers which are set-up to offer some services to the clients. They are named depending on the type of service they offered. Eg: security server, database server.
2. **Workstation:** Those are the computers designed to primarily to be used by single user at a time. They run multi-user operating systems. They are the ones which we use for our day to day personal / commercial work.

3. **Information Appliances:** They are the portable devices which are designed to perform a limited set of tasks like basic calculations, playing multimedia, browsing internet etc. They are generally referred as the mobile devices. They have very limited memory and flexibility and generally run on “as-is” basis.
4. **Embedded computers:** They are the computing devices which are used in other machines to serve limited set of requirements. They follow instructions from the non-volatile memory and they are not required to execute reboot or reset. The processing units used in such device work to those basic requirements only and are different from the ones that are used in personal computers- better known as workstations.

Classification on the basis of data handling

1. **Analog:** An analog computer is a form of computer that uses the continuously-changeable aspects of physical fact such as electrical, mechanical, or hydraulic quantities to model the problem being solved. Anything that is variable with respect to time and continuous can be claimed as analog just like an analog clock measures time by means of the distance traveled for the spokes of the clock around the circular dial.
2. **Digital:** A computer that performs calculations and logical operations with quantities represented as digits, usually in the binary number system of “0” and “1”, “Computer capable of solving problems by processing information expressed in discrete form. from manipulation of the combinations of the binary digits, it can perform mathematical calculations, organize and analyze data, control industrial and other processes, and simulate dynamic systems such as global weather patterns.
3. **Hybrid:** A computer that processes both analog and digital data, Hybrid computer is a digital computer that accepts analog signals, converts them to digital and processes them in digital form.

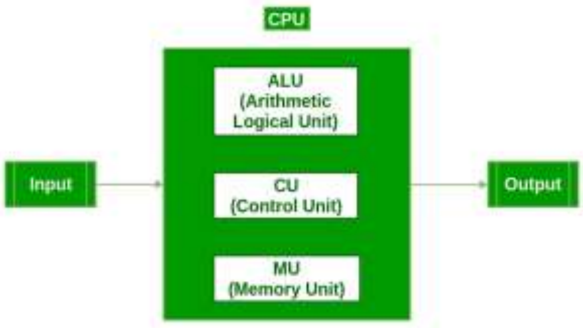
Functional units and their interconnections

Computer: A computer is a combination of **hardware and software** resources which integrate together and provides various functionalities to the user. Hardware are the physical components of a computer like the processor, memory devices, monitor, keyboard etc. while software is the set of programs or instructions that are required by the hardware resources to function properly.

There are a few basic components that aids the working-cycle of a computer i.e. the Input- Process- Output Cycle and these are called as the functional components of a computer. It needs certain input, processes that input and produces the desired output. The input unit takes the input, the central processing unit does the processing of data and the output unit produces the output. The memory unit holds the data and instructions during the processing.

Digital Computer: A digital computer can be defined as a programmable machine which reads the binary data passed as instructions, processes this binary data, and displays a calculated digital output. Therefore, Digital computers are those that work on the digital data.

Details of Functional Components of a Digital Computer



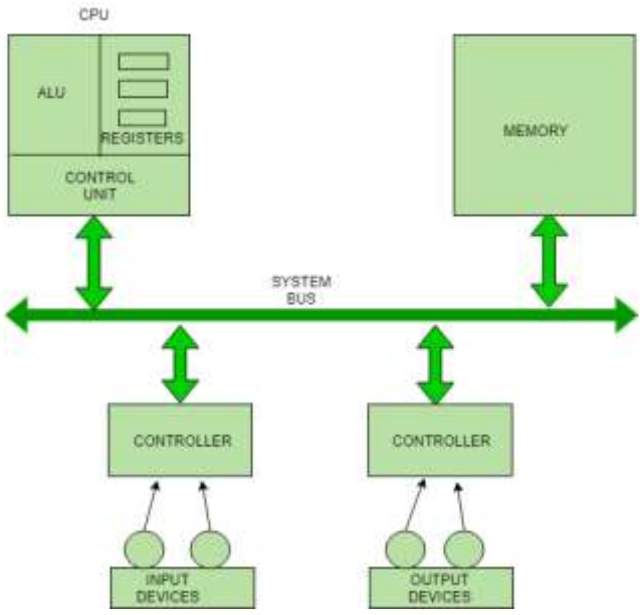
- **Input Unit:** The input unit consists of input devices that are attached to the computer. These devices take input and convert it into binary language that the computer understands. Some of the common input devices are keyboard, mouse, joystick, scanner etc.
- **Central Processing Unit (CPU):** Once the information is entered into the computer by the input device, the processor processes it. The CPU is called the brain of the computer because it is the control center of the computer. It first fetches instructions from memory and then interprets them so as to know what is to be done. If required, data is fetched from memory or input device. Thereafter CPU executes or performs the required computation and then either stores the output or displays on the output device. The CPU has three main components which are responsible for different functions – Arithmetic Logic Unit (ALU), Control Unit (CU) and Memory registers
- **Arithmetic and Logic Unit (ALU):** The ALU, as its name suggests performs mathematical calculations and takes logical decisions. Arithmetic calculations include addition, subtraction, multiplication and division. Logical decisions involve comparison of two data items to see which one is larger or smaller or equal.
- **Control Unit:** The Control unit coordinates and controls the data flow in and out of CPU and also controls all the operations of ALU, memory registers and also input/output units. It is also responsible for carrying out all the instructions stored in the program. It decodes the fetched instruction, interprets it and sends control signals to input/output devices until the required operation is done properly by ALU and memory.
- **Memory Registers:** A register is a temporary unit of memory in the CPU. These are used to store the data which is directly used by the processor. Registers can be of different sizes (16 bit, 32 bit, 64 bit and so on) and each register inside the CPU has a specific function like storing data, storing an instruction, storing address of a location in memory etc. The user registers can be used by an assembly language programmer for storing operands, intermediate results etc. Accumulator (ACC) is the main register in the ALU and contains one of the operands of an operation to be performed in the ALU.
- **Memory:** Memory attached to the CPU is used for storage of data and instructions and is called internal memory The internal memory is divided into many storage locations, each of which can store data or instructions. Each memory location is of the same size and has an address. With the help of the address, the computer can read any memory location easily without having to search the entire memory. when a program is executed, it’s data is copied to the internal memory and is stored in the memory till the end of the execution. The internal memory is also called the Primary memory or Main memory. This memory is also called as RAM, i.e. Random Access Memory. The time of access of data is independent of its location in memory, therefore this memory is also called Random Access memory (RAM). Read this for different types of RAMs

- **Output Unit:** The output unit consists of output devices that are attached with the computer. It converts the binary data coming from CPU to human understandable form. The common output devices are monitor, printer, plotter etc.

Interconnection between Functional Components

A computer consists of input unit that takes input, a CPU that processes the input and an output unit that produces output. All these devices communicate with each other through a common bus. A bus is a transmission path, made of a set of conducting wires over which data or information in the form of electric signals, is passed from one component to another in a computer. The bus can be of three types – Address bus, Data bus and Control Bus.

Following figure shows the connection of various functional components:



The address bus carries the address location of the data or instruction. The data bus carries data from one component to another and the control bus carries the control signals. The system bus is the common communication path that carries signals to/from CPU, main memory and input/output devices. The input/output devices communicate with the system bus through the controller circuit which helps in managing various input/output devices attached to the computer.

Buses

A **computer bus** is also known as a local bus, data bus, or address bus, a bus is a link between components or devices linked to a computer.

A bus, for instance, carries data through the motherboard between a CPU (Central Processing Unit) and the system memory.

The following are a few points to describe a computer bus:-

- A bus is a group of lines/wires which carry computer signals.
- A bus is the means of shared transmission.
- Lines are assigned for providing descriptive names. — carries a single electrical signal, e.g. 1-bit memory address, data bits series, or timing control that turns the device on or off.

- Data can be transferred from one computer system location to another (between different I / O modules, memory, and CPU).
- The bus is not only cable but also hardware (bus architecture), protocol, program, and bus controller.

Different components of a bus

Each bus possesses three distinct communication channels.

Following are the three components of a bus: –

- The **address** bus, a one-way pathway that allows information to pass in one direction only, carries information about where data is stored in memory.
- The **data** bus is a two-way pathway carrying the actual data (information) to and from the main memory.
- The **control** bus holds the control and timing signals needed to coordinate all of the computer's activities.

Functions of a computer bus

Below are a few of the functions in a computer bus:-

- **Data sharing** – All types of buses used in network transfer data between the connected computer peripherals. The buses either transfer or send data in serial or parallel transfer method. This allows 1, 2, 4, or even 8 bytes of data to be exchanged at a time. (A Byte is an 8-bit group). Buses are classified according to how many bits they can move simultaneously, meaning we have 8-bit, 16-bit, 32-bit, or even 64-bit buses.
- **Addressing** – A bus has address lines that suit the processors. This allows us to transfer data to or from different locations in the memory.
- **Power** – A bus supplies the power to various connected peripherals.

Structure and Topologies of Computer buses

Lines are grouped as mentioned below –

- **Power** line provides electrical power to the components connected
- **Data** lines carrying data or instructions between modules of the system
- **Address** lines indicate the recipient of the bus data
- **Control** lines control the synchronization and operation of the bus and the modules linked to the bus

Different types of computer buses?

Computers normally have two bus types:-

- **System bus** – This is the bus that connects the CPU to the motherboard's main memory. The system bus is also known as a front-side bus, a memory bus, a local bus, or a host bus.
- A number of **I / O Buses**, (I / O is an input/output acronym) connecting various peripheral devices to the CPU. These devices connect to the system bus through a 'bridge' implemented on the chipset of the processors. Other I / O bus names include "expansion bus," "external bus" or "host bus"

Below are some of the **types of Expansion buses**:-

ISA – Industry Standard Architecture

The Industry Standard Architecture (ISA) bus is still one of the oldest buses in service today.

Although it has been replaced by faster buses, ISA still has a lot of legacy devices that connect to it such as cash registers, CNC machines, and barcode scanners.

Since being expanded to 16 bits in 1984, ISA remains largely unchanged. Additional high-speed buses were added to avoid performance problems.

EISA – Extended Industry Standard Architecture

An upgrade to ISA is Extended Industry Standard Architecture or EISA. This doubled the data channels from 16 to 32 and allowed the bus to be used by more than one CPU.

Although deeper than the ISA slot, it is the same width that lets older devices connect to it.

When you compare the pins on an ISA to an EISA card (the gold portion of the card that goes into the slot), you can find that the EISA pins are longer and thinner. That is a quick way to decide if you have an ISA or an EISA card.

MCA – Micro Channel Architecture

IBM developed this bus as a substitute for ISA when they designed the PS/2 PC which was launched in 1987.

The bus provided some technological improvements over the ISA bus. The MCA, for example, ran at a speed of 10MHz faster and supported either 16-bit or 32-bit data.

One advantage of MCA was that the plug-in cards were configurable software; that means they needed minimal user input during configuration.

VESA – Video Electronics Standards Association

The Video Electronics Standards Association (VESA) Local bus was created to divide the load and allow the ISA bus to handle interrupts, and the I / O port (input/output) and the VL bus to work with Direct Memory Access (DMA) and I / O memory.

This was only a temporary solution, due to its size and other considerations. The PCI bus was easy to overtake the VL bus.

A VESA card has a range of additional pins and is longer than the ISA or EISA cards.

It was created in the early ’90s and has a 32-bit bus and was a temporary fix designed to help boost ISA’s performance.

PCI – Peripheral Component Interconnect

The PCI bus was developed to solve ISA and VL-bus-related issues. PCI has a 32-bit data path and will run at half the speed of the system memory bus.

One of its enhancements was to provide connected computers with direct access to machine memory. That increased computer efficiency while reducing the CPU’s capacity for interference.

Today’s computers mostly have PCI slots. PCI is considered a hybrid between ISA and VL-Bus that provides direct access to the connected devices’ system memory.

This uses a bridge to connect to the front side bus and CPU and is able to provide higher performance while reducing the potential for CPU interference.

PCI Express (PCI-X)

The most recent added slot is PCI Express (PCIe). It was designed to replace the AGP and PCI bus. It has a 64-bit data path and 133 MHz base speed but incorporating full-duplex architecture was the main performance enhancement.

That allowed the card to run in both directions at full speed simultaneously. PCI Express slots run at 1X, 4X, 8X, and 16X providing PCI with the highest transfer speed of any form of a slot. The multiplier specifies the maximum rate of transfer.

PCI Express is compatible backward, allowing a 1X card to fit into a 16X slot.

PCMCIA – Personal Computer Memory Card Industry Association (Also called PC bus)

The Personal Computer Memory Card Industry Association was established to give laptop computers a standard bus.

But it is used in small computers, essentially.

AGP – Accelerated Graphics Port

The Accelerated Graphics Bus (AGP) was designed to accommodate the computers' increased graphics needs. It has a data path that is 32 bits long and runs at maximum bus speed.

This doubled the PCI bandwidth and reduced the need to share the bus with other components. This means that AGP operates at 66 MHz on a regular motherboard, instead of the 33 MHz of the PCI bus.

AGP has a base speed of 66 MHz that doubles PCI speed. You can also get slots that run at speeds 2X, 4X, and 8X.

It also uses special signaling to allow twice as much data to be transmitted at the same clock speed over the port.

SCSI – Small Computer Systems Interface.

Small Computer System Interface is a standard parallel interface used for attaching peripheral devices to a computer by Apple Macintosh computers, PCs, and Unix systems.

Most common types of computer buses

Most of the listed buses are no longer used or not frequently used today.

Below is a list of the buses that are the most popular ones:-

- **ESATA and SATA**– Hard Drives and Disk Drives computer.
- **PCIe** – Video Cards and Computer Expansion Cards.
- **USB** – Peripherals to a computer.
- **Thunderbolt** – Peripherals that are connected via a USB-C cable.

Bus Arbitration

Bus Arbitration refers to the process by which the current bus master accesses and then leaves the control of the bus and passes it to another bus requesting processor unit. The controller that has access to a bus at an instance is known as a **Bus master**.

A conflict may arise if the number of DMA controllers or other controllers or processors try to access the common bus at the same time, but access can be given to only one of those. Only one processor or controller can be Bus master at the same point in time. To resolve these conflicts, the Bus Arbitration procedure is implemented to coordinate the activities of all devices requesting memory transfers. The selection of the bus master must take into account the needs of various devices by establishing a priority system for gaining access to the bus. The **Bus Arbiter** decides who would become the current bus master.

There are two approaches to bus arbitration:

1. **Centralized bus arbitration –**

   A single bus arbiter performs the required arbitration.

2. **Distributed bus arbitration –**

   All devices participating in the selection of the next bus master.

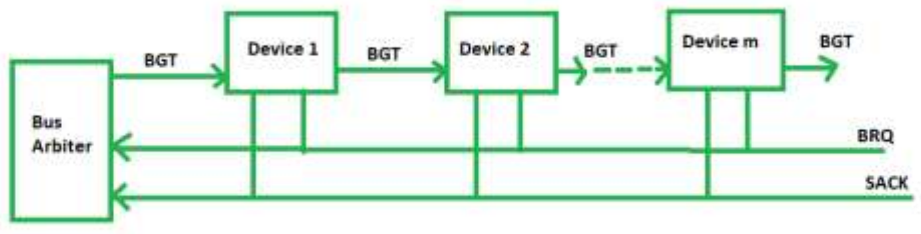
Methods of Centralized BUS Arbitration –

There are three bus arbitration methods:

(i) Daisy Chaining method –

It is a simple and cheaper method where all the bus masters use the same line for making bus requests. The bus grant signal serially propagates through each master until it encounters the first one that is requesting access to the bus. This master blocks the propagation of the bus grant signal, therefore any other requesting module will not receive the grant signal and hence cannot access the bus.

During any bus cycle, the bus master may be any device – the processor or any DMA controller unit, connected to the bus.



Daisy chained bus arbitration

Advantages –

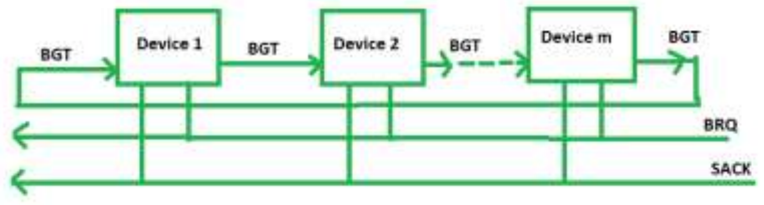
- Simplicity and Scalability.
- The user can add more devices anywhere along the chain, up to a certain maximum value.

Disadvantages –

- The value of priority assigned to a device depends on the position of the master bus.
- Propagation delay arises in this method.
- If one device fails then the entire system will stop working.

(ii) Polling or Rotating Priority method –

In this, the controller is used to generate the address for the master (unique priority), the number of address lines required depends on the number of masters connected in the system. The controller generates a sequence of master addresses. When the requesting master recognizes its address, it activates the busy line and begins to use the bus.



Rotating priority bus arbitration

Advantages –

- This method does not favor any particular device and processor.
- The method is also quite simple.
- If one device fails then the entire system will not stop working.

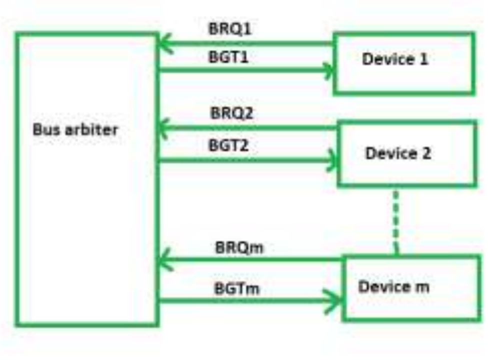
Disadvantages –

- Adding bus masters is difficult as increases the number of address lines of the circuit.

(iii) Fixed priority or Independent Request method –

In this, each master has a separate pair of bus request and bus grant lines and each pair has a priority assigned to it.

The built-in priority decoder within the controller selects the highest priority request and asserts the corresponding bus grant signal.



Fixed priority bus arbitration method

Advantages –

- This method generates a fast response.

Disadvantages –

- Hardware cost is high as a large no. of control lines is required.

Distributed BUS Arbitration:

In this, all devices participate in the selection of the next bus master. Each device on the bus is assigned a 4bit identification number. The priority of the device will be determined by the generated ID.

Register Transfer Language (RTL)

In symbolic notation, it is used to describe the micro-operations transfer among registers. It is a kind of intermediate representation (IR) that is very close to assembly language, such as that which is used in a compiler. The term "Register Transfer" can perform micro-operations and transfer the result of operation to the same or other register.

Micro-operations:

The operation executed on the data store in registers are called micro-operations. They are detailed low-level instructions used in some designs to implement complex machine instructions.

Register Transfer:

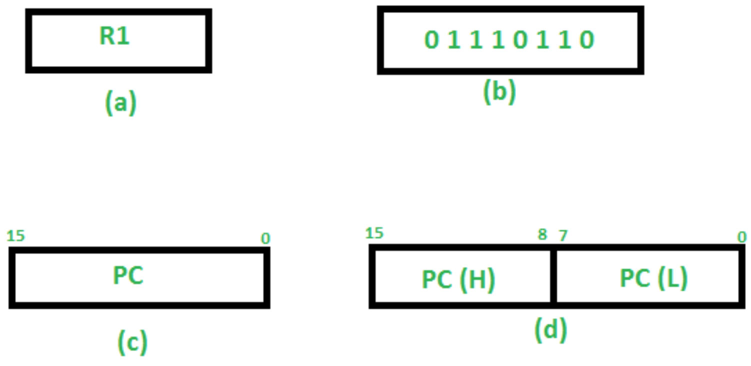
The information transformed from one register to another register is represented in symbolic form by replacement operator is called Register Transfer.

Replacement Operator:

In the statement, R2 <- R1, **<-** acts as a replacement operator. This statement defines the transfer of content of register R1 into register R2.

There are various methods of RTL –

1. General way of representing a register is by the name of the register inclosed in a rectangular box as shown in (a).
2. Register is numbered in a sequence of 0 to (n-1) as shown in (b).
3. The numbering of bits in a register can be marked on the top of the box as shown in (c).
4. A 16-bit register PC is divided into 2 parts- Bits (0 to 7) are assigned with lower byte of 16-bit address and bits (8 to 15) are assigned with higher bytes of 16-bit address as shown in (d).



Basic symbols of RTL :

Symbol	Description	Example
Letters and Numbers	Denotes a Register	MAR, R1, R2

( )	Denotes a part of register	R1(8-bit)
		R1(0-7)

<-	Denotes a transfer of information	R2 <- R1

,	Specify two micro-operations of Register Transfer	R1 <- R2
		R2 <- R1

:	Denotes conditional operations	P : R2 <- R1 if P=1
Naming Operator (:=)	Denotes another name for an already existing register/alias	Ra := R1

Register Transfer Operations:

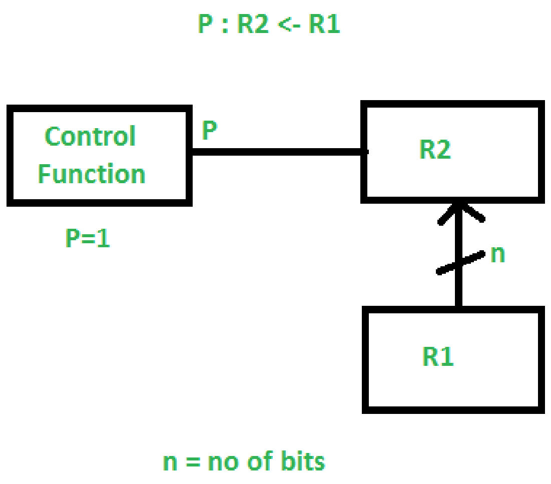
The operation performed on the data stored in the registers are referred to as register transfer operations.

There are different types of register transfer operations:

1. Simple Transfer – R2 <- R1

The content of R1 are copied into R2 without affecting the content of R1. It is an unconditional type of transfer operation.

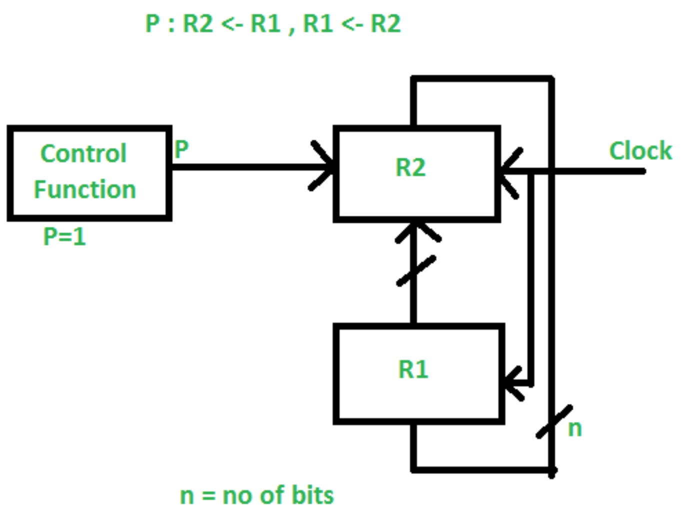
2. Conditional Transfer –



It indicates that if P=1, then the content of R1 is transferred to R2. It is a unidirectional operation.

3. Simultaneous Operations

If 2 or more operations are to occur simultaneously then they are separated with comma **(,)**.



If the control function P=1, then load the content of R1 into R2 and at the same clock load the content of R2 into R1.

Bus transfer

A digital system composed of many registers, and paths must be provided to transfer information from one register to another. The number of wires connecting all of the registers will be excessive if separate lines are used between each register and all other registers in the system.

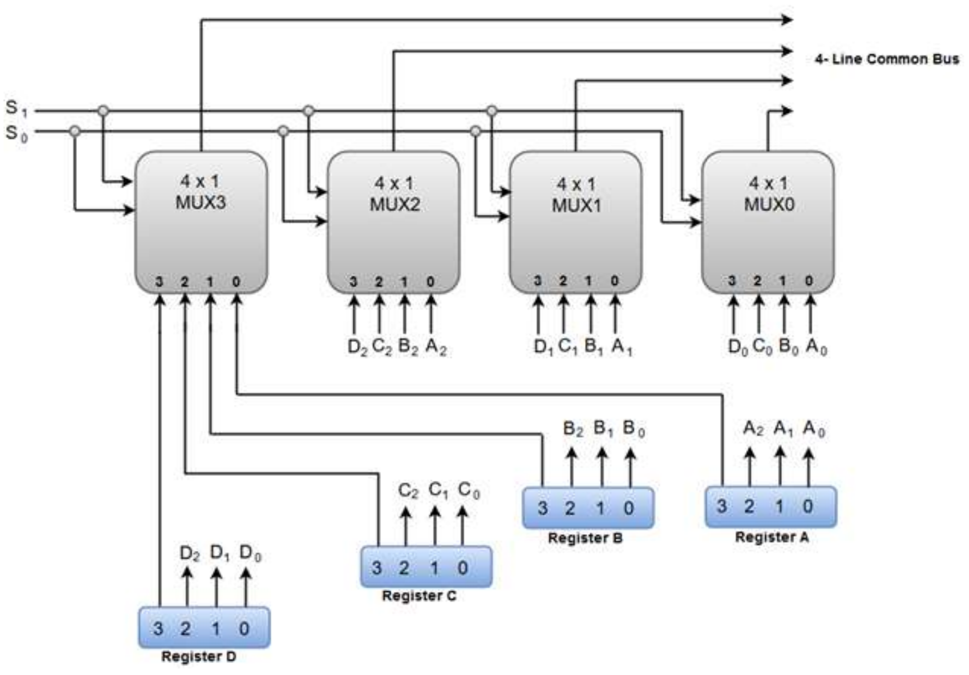
A bus structure, on the other hand, is more efficient for transferring information between registers in a multi-register configuration system.

A bus consists of a set of common lines, one for each bit of register, through which binary information is transferred one at a time. Control signals determine which register is selected by the bus during a particular register transfer.

The following block diagram shows a Bus system for four registers. It is constructed with the help of four 4 * 1 Multiplexers each having four data inputs (0 through 3) and two selection inputs (S1 and S2).

We have used labels to make it more convenient for you to understand the input-output configuration of a Bus system for four registers. For instance, output 1 of register A is connected to input 0 of MUX1.

Bus System for 4 Registers:



The two selection lines S1 and S2 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 both of the select lines are at low logic, i.e. S1S0 = 00, the 0 data inputs of all four multiplexers are selected and applied to the outputs that forms the bus. This, in turn, 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, when S1S0 = 01, register B is selected, and the bus lines will receive the content provided by register B.

The following function table shows the register that is selected by the bus for each of the four possible binary values of the Selection lines.

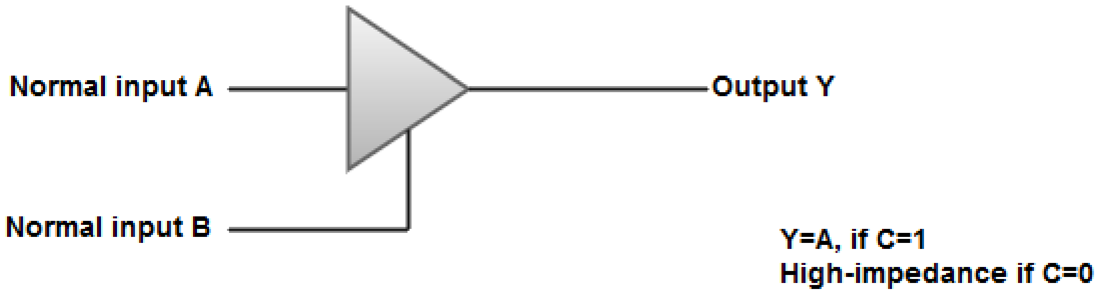
S1	S0	Register Selected
0	0	A
0	1	B
1	0	C
1	1	D

A bus system can also be constructed using **three-state gates** instead of multiplexers.

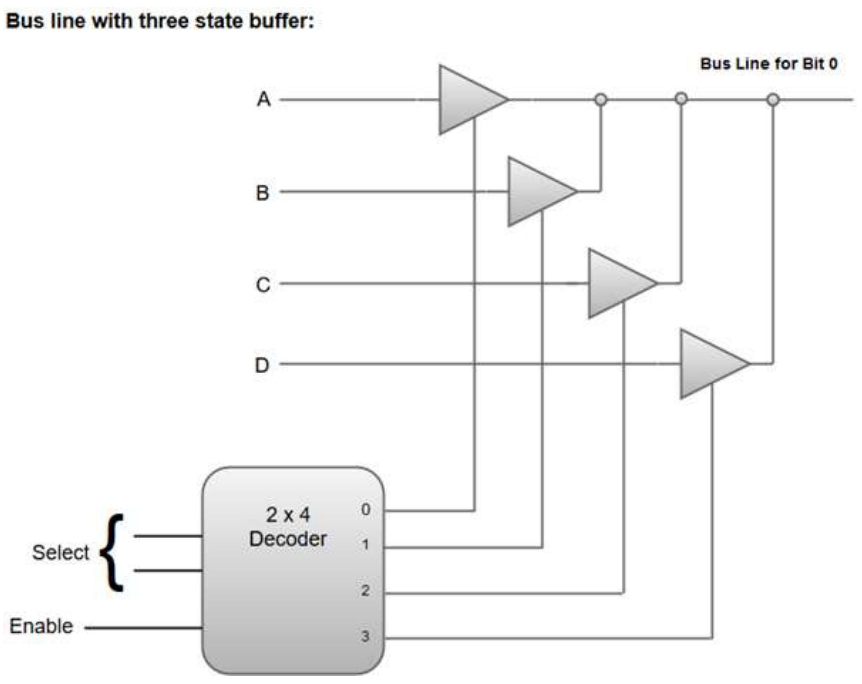
The **three state gates** can be considered as a digital circuit that has three gates, two of which are signals equivalent to logic 1 and 0 as in a conventional gate. However, the third gate exhibits a high-impedance state.

The most commonly used three state gates in case of the bus system is a **buffer gate**.

The graphical symbol of a three-state buffer gate can be represented as:



The following diagram demonstrates the construction of a bus system with three-state buffers.



- The outputs generated by the four buffers are connected to form a single bus line.
- Only one buffer can be in active state at a given point of time.
- The control inputs to the buffers determine which of the four normal inputs will communicate with the bus line.
- A 2 * 4 decoder ensures that no more than one control input is active at any given point of time.

Memory Transfer

Most of the standard notations used for specifying operations on memory transfer are stated below.

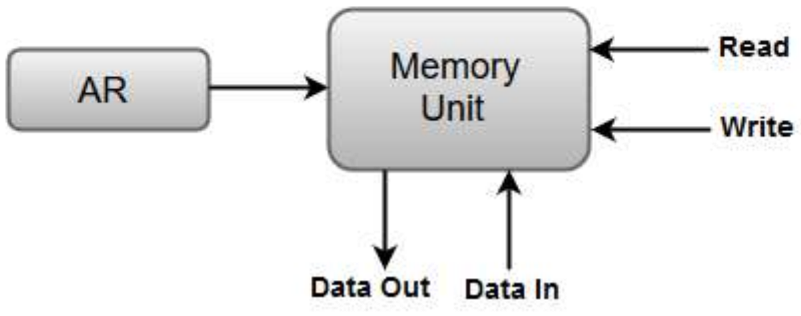
- The transfer of information from a memory unit to the user end is called a **Read** operation.
- The transfer of new information to be stored in the memory is called a **Write** operation.
- A memory word is designated by the letter **M**.
- We must specify the address of memory word while writing the memory transfer operations.
- The **address register** is designated by **AR** and the **data register** by **DR**.
- Thus, a read operation can be stated as:

1. Read: DR ← M [AR]

- The **Read** statement causes a transfer of information into the data register (DR) from the memory word (M) selected by the address register (AR).
- And the corresponding write operation can be stated as:

2. Write: M [AR] ← R1

- The Write statement causes a transfer of information from register R1 into the memory word (M) selected by address register (AR).



Addition and subtraction of signed numbers

The basic arithmetic operations are addition and subtraction.

Addition of two Signed Binary Numbers

Consider the two signed binary numbers A & B, which are represented in 2's complement form. We can perform the **addition** of these two numbers, which is similar to the addition of two unsigned binary numbers. But, if the resultant sum contains carry out from sign bit, then discard ignore it in order to get the correct value.

If resultant sum is positive, you can find the magnitude of it directly. But, if the resultant sum is negative, then take 2's complement of it in order to get the magnitude.

Example 1

Let us perform the **addition** of two decimal numbers **+7 and +4** using 2's complement method.

The **2's complement** representations of +7 and +4 with 5 bits each are shown below.

$+7+7_{10} = 0011100111_2$

$+4+4_{10} = 0010000100_2$

The addition of these two numbers is

$+7+7_{10} ++4+4_{10} = 0011100111_2+0010000100_2$

$\Rightarrow +7+7_{10} ++4+4_{10} = 0101101011_2$.

The resultant sum contains 5 bits. So, there is no carry out from sign bit. The sign bit '0' indicates that the resultant sum is **positive**. So, the magnitude of sum is 11 in decimal number system. Therefore, addition of two positive numbers will give another positive number.

Example 2

Let us perform the **addition** of two decimal numbers **-7** and **-4** using 2's complement method.

The **2's complement** representation of -7 and -4 with 5 bits each are shown below.

$-7-7_{10} = 1100111001_2$

$-4-4_{10} = 1110011100_2$

The addition of these two numbers is

$-7-7_{10} + -4-4_{10} = 1100111001_2 + 1110011100_2$

$\Rightarrow -7-7_{10} + -4-4_{10} = 110101110101_2$.

The resultant sum contains 6 bits. In this case, carry is obtained from sign bit. So, we can remove it

Resultant sum after removing carry is $-7-7_{10} + -4-4_{10} = 1010110101_2$.

The sign bit ‘1’ indicates that the resultant sum is **negative**. So, by taking 2’s complement of it we will get the magnitude of resultant sum as 11 in decimal number system. Therefore, addition of two negative numbers will give another negative number.

Subtraction of two Signed Binary Numbers

Consider the two signed binary numbers A & B, which are represented in 2’s complement form. We know that 2’s complement of positive number gives a negative number. So, whenever we have to subtract a number B from number A, then take 2’s complement of B and add it to A. So, **mathematically** we can write it as

A - B = A + 2′scomplementofB2′scomplementofB

Similarly, if we have to subtract the number A from number B, then take 2’s complement of A and add it to B. So, **mathematically** we can write it as

B - A = B + 2′scomplementofA2′scomplementofA

So, the subtraction of two signed binary numbers is similar to the addition of two signed binary numbers. But, we have to take 2’s complement of the number, which is supposed to be subtracted. This is the **advantage** of 2’s complement technique. Follow, the same rules of addition of two signed binary numbers.

Example 3

Let us perform the **subtraction** of two decimal numbers **+7 and +4** using 2’s complement method.

The subtraction of these two numbers is

$+7+7_{10} - +4+4_{10} = +7+7_{10} + -4-4_{10}$.

The **2’s complement** representation of +7 and -4 with 5 bits each are shown below.

$+7+7_{10} = 0011100111_{2}$

$+4+4_{10} = 1110011100_{2}$

$\Rightarrow +7+7_{10} + +4+4_{10} = 0011100111_{2} + 1110011100_{2} = 0001100011_{2}$

Here, the carry obtained from sign bit. So, we can remove it. The resultant sum after removing carry is

$+7+7_{10} + +4+4_{10} = 0001100011_{2}$

The sign bit ‘0’ indicates that the resultant sum is **positive**. So, the magnitude of it is 3 in decimal number system. Therefore, subtraction of two decimal numbers +7 and +4 is +3.

Example 4

Let us perform the **subtraction of** two decimal numbers **+4** and **+7** using 2’s complement method.

The subtraction of these two numbers is

$+4+4_{10} - +7+7_{10} = +4+4_{10} + -7-7_{10}$.

The **2’s complement** representation of +4 and -7 with 5 bits each are shown below.

$+4+4_{10} = 0010000100_{2}$

$-7-7_{10} = 1100111001_{2}$

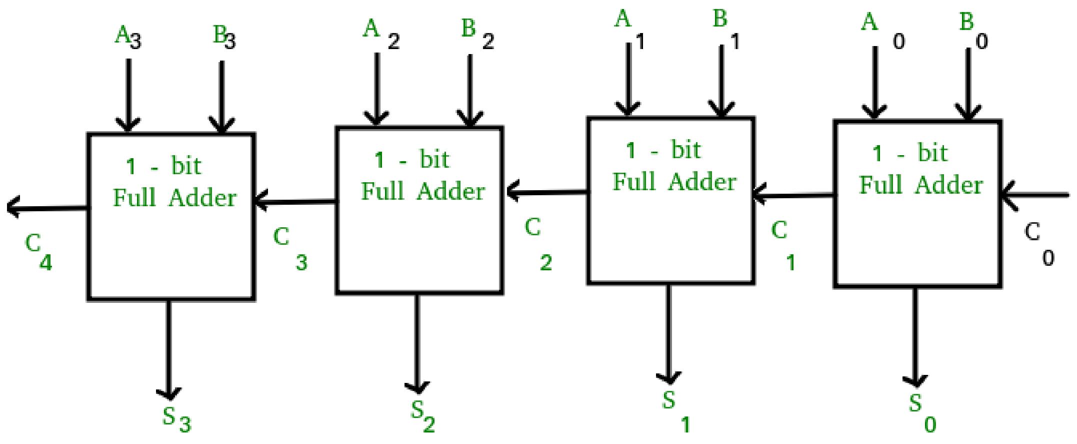
$\Rightarrow +4+4_{10} + -7-7_{10} = 0010000100_{2} + 1100111001_{2} = 1110111101_{2}$

Here, carry is not obtained from sign bit. The sign bit ‘1’ indicates that the resultant sum is **negative**. So, by taking 2’s complement of it we will get the magnitude of resultant sum as 3 in decimal number system. Therefore, subtraction of two decimal numbers +4 and +7 is -3.

Carry Look-Ahead Adder

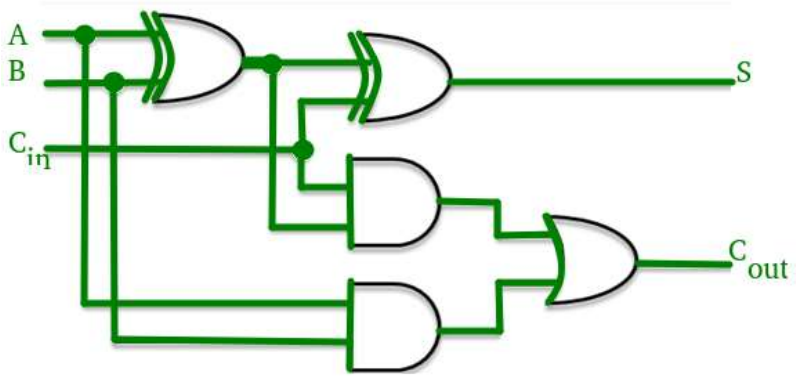
Motivation behind Carry Look-Ahead Adder:

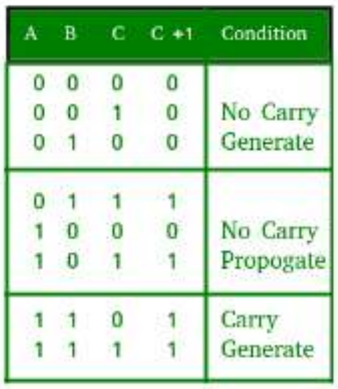
In ripple carry adders, for each adder block, the two bits that are to be added are available instantly. However, each adder block waits for the carry to arrive from its previous block. So, it is not possible to generate the sum and carry of any block until the input carry is known. The block waits for the block to produce its carry. So there will be a considerable time delay which is carry propagation delay.



Carry Look-ahead Adder :

A carry look-ahead adder reduces the propagation delay by introducing more complex hardware. In this design, the ripple carry design is suitably transformed such that the carry logic over fixed groups of bits of the adder is reduced to two-level logic. Let us discuss the design in detail.





A	B	C	C +1	Condition
0	0	0	0	No Carry Generate
0	0	1	0	
0	1	0	0	
0	1	1	1	No Carry Propogate
1	0	0	0	
1	0	1	1	
1	1	0	1	Carry Generate
1	1	1	1	

Consider the full adder circuit shown above with corresponding truth table. We define two variables as **'carry generate'** G_i and **'carry propagate'** P_i then,

$P_i = A_i + B_i$

$G_i = A_i B_i$

The sum output and carry output can be expressed in terms of carry generate G_i and carry propagate P_i as

Si = Pi+Ci

$C_{i+1} = G_i + P_i C_i$

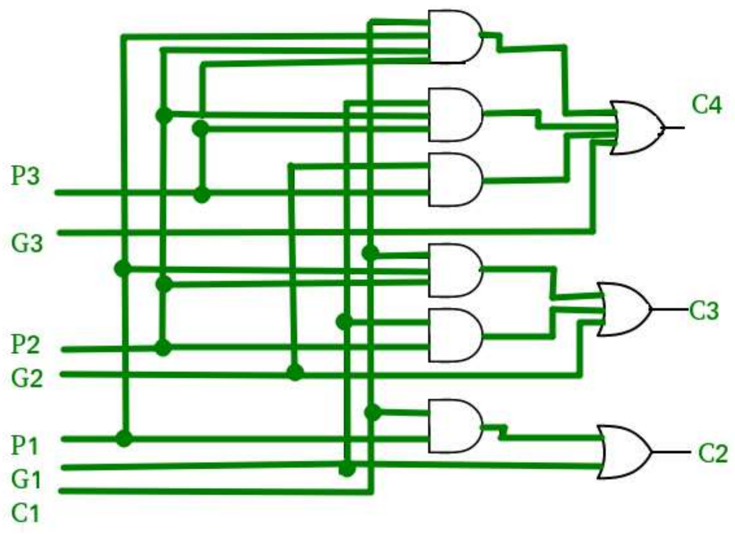
where G_i produces the carry when both A_i, B_i are 1 regardless of the input carry. P_i is associated with the propagation of carry from C_i to C_{i+1}.

The carry output Boolean function of each stage in a 4 stage carry look-ahead adder can be expressed as

$$
\begin{aligned}
C_1 &= G_0 + P_0 C_{in} \\
C_2 &= G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_{in} \\
C_3 &= G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{in} \\
C_4 &= G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_{in}
\end{aligned}
$$

From the above Boolean equations we can observe that C_4 does not have to wait for C_3 and C_2 to propagate but actually C_4 is propagated at the same time as C_3 and C_2. Since the Boolean expression for each carry output is the sum of products so these can be implemented with one level of AND gates followed by an OR gate.

The implementation of three Boolean functions for each carry output (C_2, C_3 and C_4) for a carry look-ahead carry generator shown in below figure.



Time Complexity Analysis:

We could think of a carry look-ahead adder as made up of two "parts"

1. The part that computes the carry for each bit.
2. The part that adds the input bits and the carry for each bit position.

The *log (n)* complexity arises from the part that generates the carry, not the circuit that adds the bits. Now, for the generation of the n^{th} carry bit, we need to perform a AND between (n+1) inputs. The complexity of the adder comes down to how we perform this AND operation. If we have AND gates, each with a fan-in (number of inputs accepted) of k, then we can find the AND of all the bits in $Log_k\ (n+1)$ time. This is represented in asymptotic notation as *(log n).*

Advantages and Disadvantages of Carry Look-Ahead Adder:

Advantages –

- The propagation delay is reduced.
- It provides the fastest addition logic.

Disadvantages –

- The Carry Look-ahead adder circuit gets complicated as the number of variables increase.
- The circuit is costlier as it involves more number of hardware.

Multiplication: Signed operand multiplication

Multiplication of two fixed point binary number in *signed magnitude representation* is done with process of *successive shift* and *add operation*.

```
     10111 (Multiplicand)
   x 10011 (Multiplier)
     10111
    10111
   00000
  00000
 10111
 011011010     (Product)
```

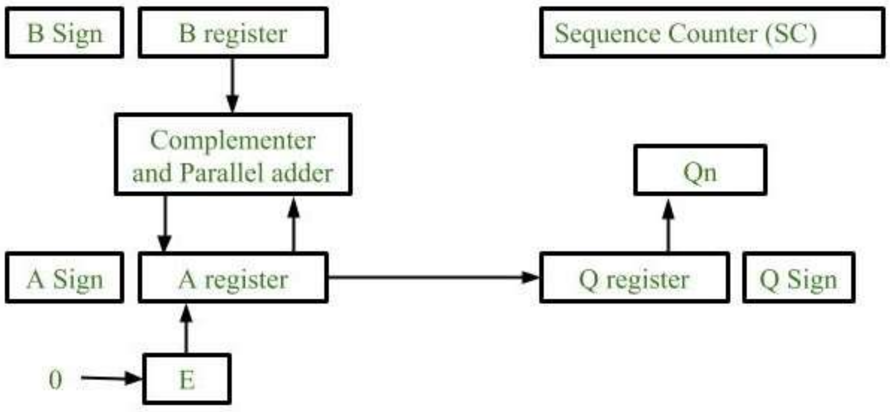
In the multiplication process we are considering successive bits of the multiplier, least significant bit first. If the multiplier bit is 1, the multiplicand is copied down else 0's are copied down.

The numbers copied down in successive lines are shifted one position to the left from the previous number. Finally numbers are added and their sum form the product.

The sign of the product is determined from the sign of the multiplicand and multiplier. If they are alike, sign of the product is positive else negative.

Hardware Implementation:

Following components are required for the *Hardware Implementation* of multiplication algorithm :



1. **Registers:**

   Two Registers B and Q are used to store multiplicand and multiplier respectively.

   Register A is used to store partial product during multiplication.

   Sequence Counter register (SC) is used to store number of bits in the multiplier.

2. **Flip Flop:**

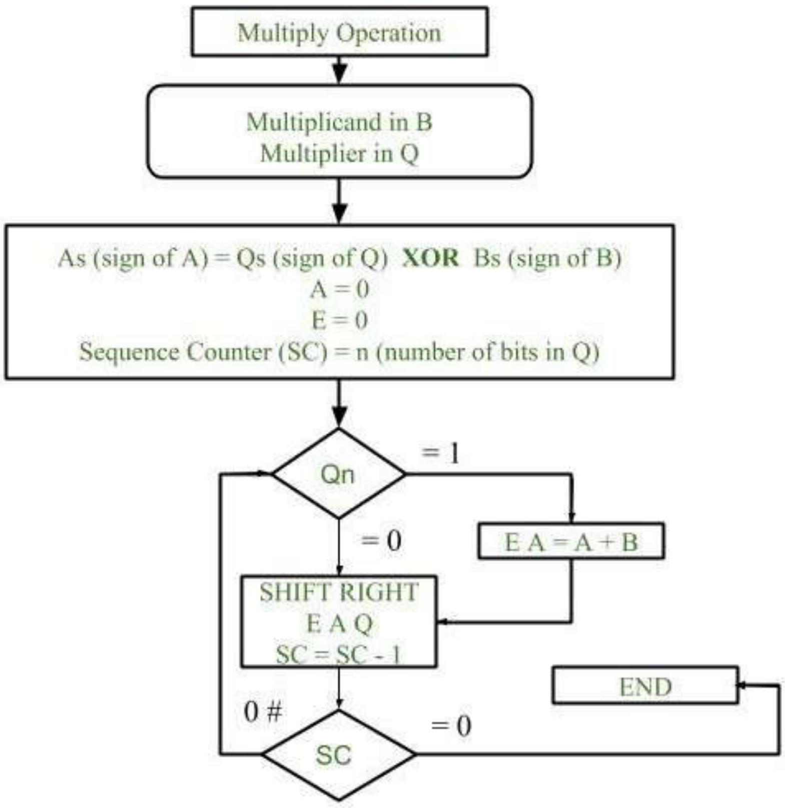
   To store sign bit of registers we require three flip flops (A sign, B sign and Q sign).

   Flip flop E is used to store carry bit generated during partial product addition.

3. **Complement and Parallel adder:**

   This hardware unit is used in calculating partial product i.e, perform addition required.

   Flowchart of Multiplication:



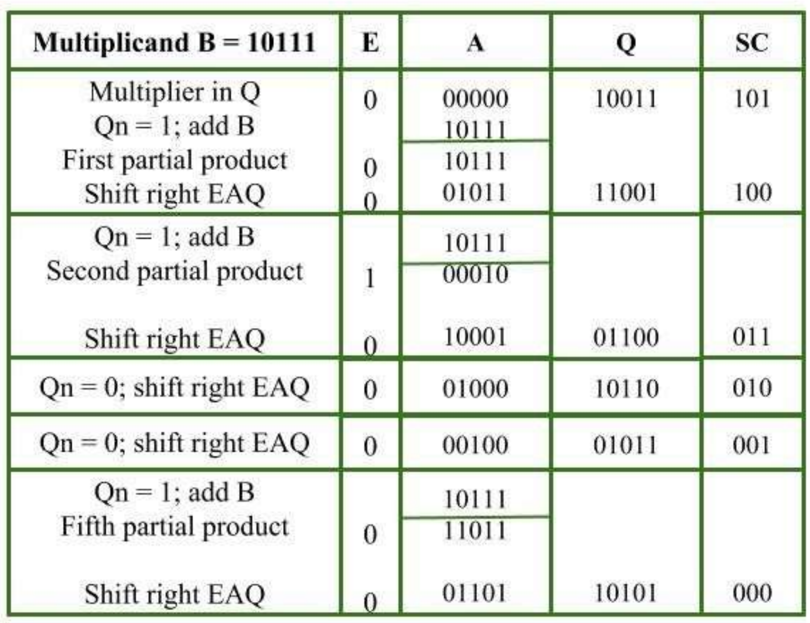
1. Initially multiplicand is stored in B register and multiplier is stored in Q register.
2. Sign of registers B (Bs) and Q (Qs) are compared using **XOR** functionality (i.e., if both the signs are alike, output of XOR operation is 0 unless 1) and output stored in As (sign of A register).

   Note: Initially 0 is assigned to register A and E flip flop. Sequence counter is initialized with value n, n is the number of bits in the Multiplier.
3. Now least significant bit of multiplier is checked. If it is 1 add the content of register A with Multiplicand (register B) and result is assigned in A register with carry bit in flip flop E. Content of E A Q is shifted to right by one position, i.e., content of E is shifted to most significant bit (MSB) of A and least significant bit of A is shifted to most significant bit of Q.
4. If Qn = 0, only shift right operation on content of E A Q is performed in a similar fashion.
5. Content of Sequence counter is decremented by 1.
6. Check the content of Sequence counter (SC), if it is 0, end the process and the final product is present in register A and Q, else repeat the process.

   Example:

   Multiplicand = 10111

   Multiplier = 10011



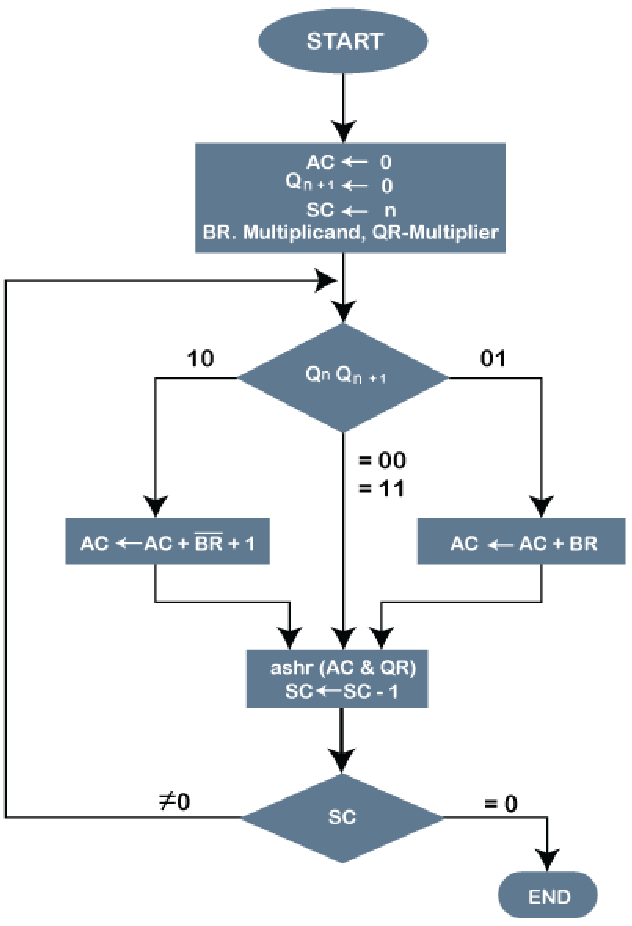
Multiplicand B = 10111	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
Qn = 1; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
Qn = 1; add B		10111		
Second partial product	1	00010		
Shift right EAQ	0	10001	01100	011
Qn = 0; shift right EAQ	0	01000	10110	010
Qn = 0; shift right EAQ	0	00100	01011	001
Qn = 1; add B		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000

Final product in AQ
0110110101

Booth's Multiplication Algorithm

The booth algorithm is a multiplication algorithm that allows us to multiply the two signed binary integers in 2's complement, respectively. It is also used to speed up the performance of the multiplication process. It is very efficient too. It works on the string bits 0's in the multiplier that requires no additional bit only shift the right-most string bits and a string of 1's in a multiplier bit weight 2^k to weight 2^m that can be considered as $\mathbf{2^{k+1} - 2^m}$.

Following is the pictorial representation of the Booth's Algorithm:



In the above flowchart, initially, **AC** and **Q_{n+1}** bits are set to 0, and the **SC** is a sequence counter that represents the total bits set **n,** which is equal to the number of bits in the multiplier. There are **BR** that represent the **multiplicand bits,** and QR represents the **multiplier bits**. After that, we encountered two bits of the multiplier as Q_n and Q_{n+1}, where Qn represents the last bit of QR, and Q_{n+1} represents the incremented bit of Qn by 1. Suppose two bits of the multiplier is equal to 10; it means that we have to subtract the multiplier from the partial product in the accumulator AC and then perform the arithmetic shift operation (ashr). If the two of the multipliers equal to 01, it means we need to perform the addition of the multiplicand to the partial product in accumulator AC and then perform the arithmetic shift operation (**ashr**), including **Q_{n+1}**. The arithmetic shift operation is used in Booth's algorithm to shift AC and QR bits to the right by one and remains the sign bit in AC unchanged. And the sequence counter is continuously decremented till the computational loop is repeated, equal to the number of bits (n).

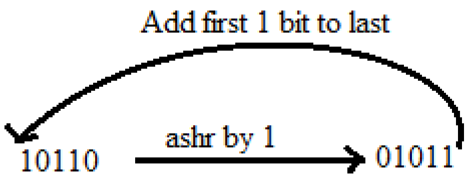
Working on the Booth Algorithm

1. Set the Multiplicand and Multiplier binary bits as M and Q, respectively.
2. Initially, we set the AC and Q_{n+1} registers value to 0.
3. SC represents the number of Multiplier bits (Q), and it is a sequence counter that is continuously decremented till equal to the number of bits (n) or reached to 0.
4. A Qn represents the last bit of the Q, and the Q_{n+1} shows the incremented bit of Qn by 1.
5. On each cycle of the booth algorithm, Q_n and Q_{n+1} bits will be checked on the following parameters as follows:
   i. When two bits Q_n and Q_{n+1} are 00 or 11, we simply perform the arithmetic shift right operation (ashr) to the partial product AC. And the bits of Qn and Q_{n+1} is incremented by 1 bit.
   ii. If the bits of Q_n and Q_{n+1} is shows to 01, the multiplicand bits (M) will be added to the AC (Accumulator register). After that, we perform the right shift operation to the AC and QR bits by 1.
   iii. If the bits of Q_n and Q_{n+1} is shows to 10, the multiplicand bits (M) will be subtracted from the AC (Accumulator register). After that, we perform the right shift operation to the AC and QR bits by 1.
6. The operation continuously works till we reached n - 1 bit in the booth algorithm.
7. Results of the Multiplication binary bits will be stored in the AC and QR registers.

There are two methods used in Booth's Algorithm:

1. RSC (Right Shift Circular)

It shifts the right-most bit of the binary number, and then it is added to the beginning of the binary bits.



2. RSA (Right Shift Arithmetic)

It adds the two binary bits and then shift the result to the right by 1-bit position.

Example: Multiply the two numbers 7 and 3 by using the Booth's multiplication algorithm.

Ans. Here we have two numbers, 7 and 3. First of all, we need to convert 7 and 3 into binary numbers like 7 = (0111) and 3 = (0011). Now set 7 (in binary 0111) as multiplicand (M) and 3 (in binary 0011) as a multiplier (Q). And SC (Sequence Count) represents the number of bits, and here we have 4 bits, so set the SC = 4. Also, it shows the number of iteration cycles of the booth's algorithms and then cycles run SC = SC - 1 time.

Q_n	**Q_{n+1}**	**M = (0111)** **M' + 1 = (1001) & Operation**	**AC**	**Q**	**Q_{n+1}**	**SC**
1	0	Initial	0000	0011	0	4
		Subtract (M' + 1)	1001			
			1001			
		Perform Arithmetic Right Shift operations (ashr)	1100	1001	1	3
1	1	Perform Arithmetic Right Shift operations (ashr)	1110	0100	1	2
0	**1**	Addition (A + M)	0111			
			0101	0100		
		Perform Arithmetic right shift operation	0010	1010	0	1
0	0	Perform Arithmetic right shift operation	**0001**	**0101**	0	0

The numerical example of the Booth's Multiplication Algorithm is 7 x 3 = 21 and the binary representation of 21 is 10101. Here, we get the resultant in binary 00010101. Now we convert it into decimal, as $(000010101)_{10}$ = 2*4 + 2*3 + 2*2 + 2*1 + 2*0 => 21.

Example: Multiply the two numbers 23 and -9 by using the Booth's multiplication algorithm.

Here, M = 23 = (010111) and Q = -9 = (110111)

Q_n	**Q_{n+1}**	**M = 0 1 0 1 1 1** **M' + 1 = 1 0 1 0 0 1**	**AC**	**Q**	**Q_{n+1}**	**SC**
		Initially	000000	110111	0	6
1	0	Subtract M	101001			
			101001			
		Perform Arithmetic right shift operation	110100	111011	1	5
1	1	Perform Arithmetic right shift operation	111010	011101	1	4
1	1	Perform Arithmetic right shift operation	111101	001110	1	3
0	1	Addition (A + M)	010111			
			010100			

		Perform Arithmetic right shift operation	001010	000111	0	2
1	0	Subtract M	101001			
			110011			
		Perform Arithmetic right shift operation	111001	100011	1	1
1	1	Perform Arithmetic right shift operation	**111100**	**110001**	**1**	0

$\mathbf{Q_{n+1}}$ = 1, it means the output is negative.

Hence, 23 * -9 = 2's complement of 111100110001 => **(00001100111)**

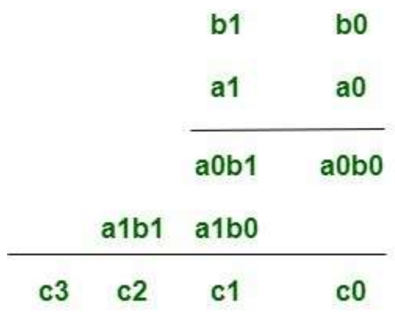
Array Multiplier

An **array multiplier** is a digital combinational circuit used for multiplying two binary numbers by employing an array of full adders and half adders. This array is used for the nearly simultaneous addition of the various product terms involved. To form the various product terms, an array of AND gates is used before the Adder array.

Checking the bits of the multiplier one at a time and forming partial products is a sequential operation that requires a sequence of add and shift micro-operations. The multiplication of two binary numbers can be done with one micro-operation by means of a combinational circuit that forms the product bits all at once. This is a fast way of multiplying two numbers since all it takes is the time for the signals to propagate through the gates that form the multiplication array. However, an array multiplier requires a large number of gates, and for this reason it was not economical until the development of integrated circuits.

For implementation of array multiplier with a combinational circuit, consider the multiplication of two 2-bit numbers as shown in figure. The multiplicand bits are b1 and b0, the multiplier bits are a1 and a0, and the product is

c3c2c1c0

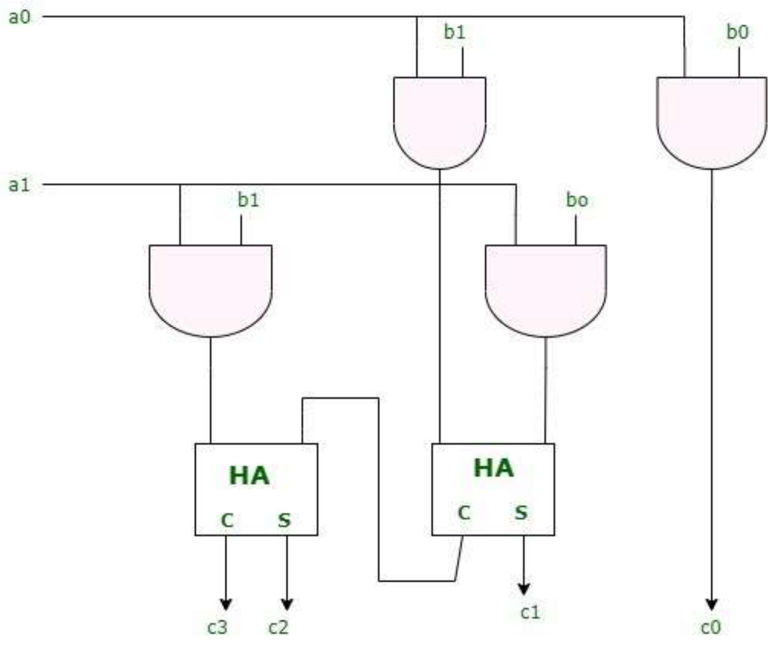


Assuming A = a1a0 and B= b1b0, the various bits of the final product term P can be written as:-

1. P(0)= a0b0
2. P(1)=a1b0 + b1a0
3. P(2) = a1b1 + c1 where c1 is the carry generated during the addition for the P(1) term.
4. P(3) = c2 where c2 is the carry generated during the addition for the P(2) term.

For the above multiplication, an array of four AND gates is required to form the various product terms like a0b0 etc. and then an adder array is required to calculate the sums involving the various product terms and carry combinations mentioned in the above equations in order to get the final Product bits.

1. The first partial product is formed by multiplying a0 by b1, b0. The multiplication of two bits such as a0 and b0 produces a 1 if both bits are 1; otherwise, it produces 0. This is identical to an AND operation and can be implemented with an AND gate.
2. The first partial product is formed by means of two AND gates.
3. The second partial product is formed by multiplying a1 by b1b0 and is shifted one position to the left.
4. The above two partial products are added with two half-adder(HA) circuits. Usually there are more bits in the partial products and it will be necessary to use full-adders to produce the sum.
5. Note that the least significant bit of the product does not have to go through an adder since it is formed by the output of the first AND gate.



A combinational circuit binary multiplier with more bits can be constructed in similar fashion. A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there are bits in the multiplier. The binary output in each level of AND gates is added in parallel with the partial product of the previous level to form a new partial product. The last level produces the product. For j multiplier bits and k multiplicand we need **j*k** AND gates and **(j-1)** k-bit adders to produce a product of **j+k** bits.

Division and logic operation

Restoring division operates on fixed point fractional numbers and depends on the assumption $0 < D < N$, where

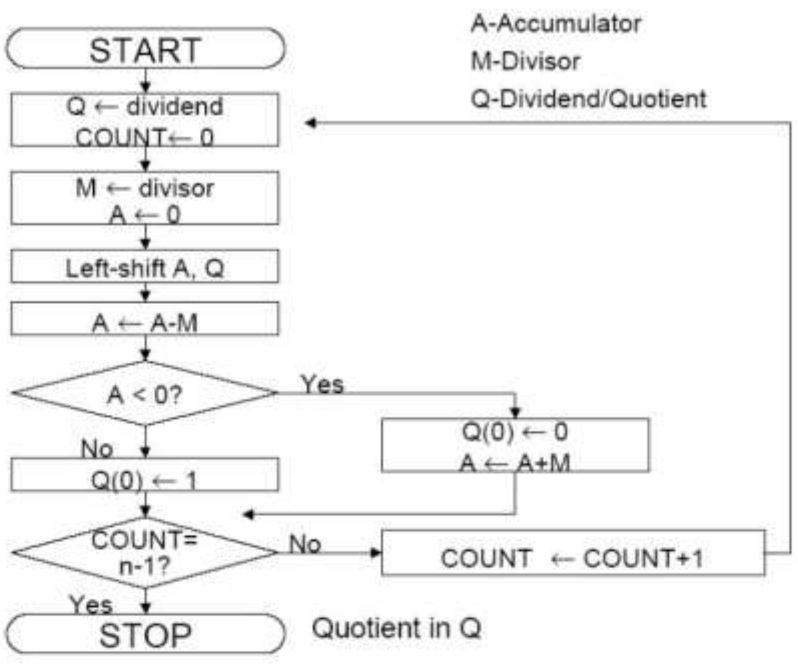
N = Numerator (dividend)

D = Denominator (divisor)

Algorithm for restoring division:

First the registers are initialized with corresponding values (Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend).Then the content of register A and Q is shifted right as if they are a single unit,the content of register M is subtracted from A and the result is stored in A. Then the most significant bit of the A is checked if it is 0 the least significant bit of Q is set to 1 otherwise if it is 1 the least significant bit of Q is set to 0 and value of register A is restored i.e the value of A before the subtraction with M.The value of counter n is decremented. If the value of n becomes zero we get of the loop otherwise we repeat from step 2. Finally, the register Q contain the quotient and A contain remainder

Flowchart for restoring division operation:

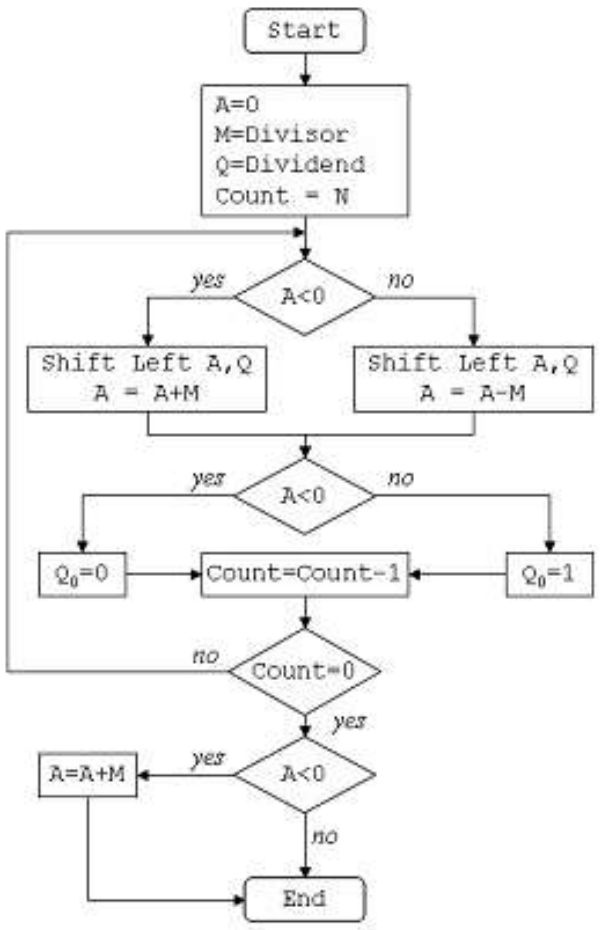


Non-restoring division uses the digit set {−1, 1} for the quotient digits instead of {0, 1}. The algorithm is more complex, but has the advantage when implemented in hardware that there is only one decision and addition/subtraction per quotient bit; there is no restoring step after the subtraction, which potentially cuts down the numbers of operations by up to half and lets it be executed faster.

Algorithm for non-restoring division:

Set the value of register A as 0 (N bits).Set the value of register M as Divisor (N bits).Set the value of register Q as Dividend (N bits).Concatenate A with Q {A,Q}.Repeat the following "N" number of times (here N is no. of bits in divisor)-->If the sign bit of A equals 0,shift A and Q combined left by 1 bit and subtract M from A,else shift A and Q combined left by 1 bit and add M to A.Now if sign bit of A equals 0, then set Q[0] as 1, else set Q[0] as 0.Finally if the sign bit of A equals 1 then add M to A.Assign A as remainder and Q as quotient.

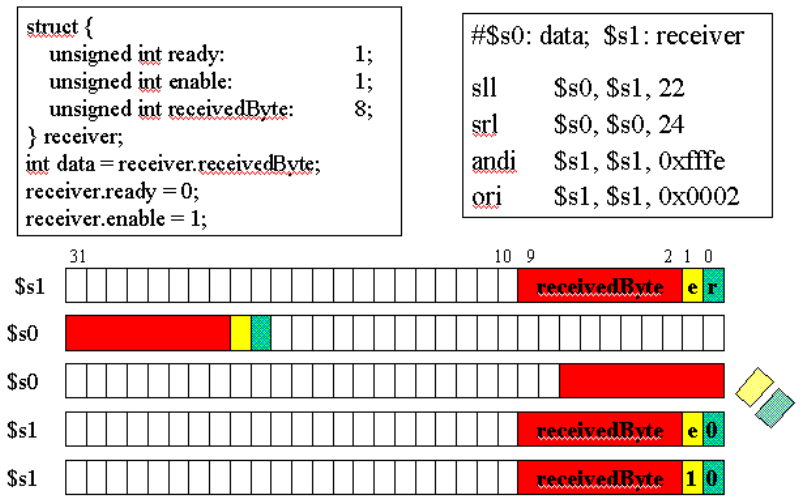
Flowchart for non-restoring division operation:



Logical Operations

Logical operations apply to fields of bits within a 32-bit word, such as bytes or bit fields (in C, as discussed in the next paragraph). These operations include shift-left and shift-right operations (sll and srl), as well as bitwise *and*, *or* (and, andi, or, ori). As we saw in Section 2, bitwise operations treat an operand as a vector of bits and operate on each bit position.

C bit fields are used, for example, in programming communications hardware, where manipulation of a bit stream is required. In Figure, presented C code for an example communications routine, where a structure called receiver is formed from an 8-bit field called *receivedByte* and two one-bit fields called *ready* and *enable*. The C routine sets receiver.ready to 0 and receiver.enable to 1.



```
struct {
   unsigned int ready:          1;
   unsigned int enable:         1;
   unsigned int receivedByte:   8;
} receiver;
int data = receiver.receivedByte;
receiver.ready = 0;
receiver.enable = 1;
```

```
#$s0: data;  $s1: receiver

sll    $s0, $s1, 22
srl    $s0, $s0, 24
andi   $s1, $s1, 0xfffe
ori    $s1, $s1, 0x0002
```

Figure Example of C bit field use in MIPS, adapted from [Maf01].

Note: how the MIPS code implements the functionality of the C code, where the state of the registers $s0 and $s1 is illustrated in the five lines of diagrammed register contents below the code. In particular, the initial register state is shown in the first two lines. The sll instruction loads the contents of $s1 (the receiver) into $s0 (the data register), and the result of this is shown on the second line of the register contents. Next, the srl instruction left-shifts $s0 24 bits, thereby discarding the *enable* and *ready* field information, leaving just the received byte. To signal the receiver that the data transfer is completed, the andi and ori instructions are used to set the enable and ready bits in $s1, which corresponds to the *receiver*. The data in $s0 has already been received and put in a register, so there is no need for its further manipulation.

Floating point arithmetic operation Processor organization

Representation

When you have to represent very small or very large numbers, a fixed point representation will not do. The accuracy will be lost. Therefore, you will have to look at floating-point representations, where the binary point is assumed to be floating. When you consider a decimal number 12.34 * 107, this can also be treated as 0.1234 * 109, where 0.1234 is the fixed-point mantissa. The other part represents the exponent value, and indicates that the actual position of the binary point is 9 positions to the right (left) of the indicated binary point in the fraction. Since the binary point can be moved to any position and the exponent value adjusted appropriately, it is called a floating-point representation. By convention, you generally go in for a normalized representation, wherein the floating-point is placed to the right of the first nonzero (significant) digit. The base need not be specified explicitly and the sign, the significant digits and the signed exponent constitute the representation.

The IEEE (Institute of Electrical and Electronics Engineers) has produced a standard for floating point arithmetic. This standard specifies how single precision (32 bit) and double precision (64 bit) floating point numbers are to be represented, as well as how arithmetic should be carried out on them. The IEEE single precision floating point standard representation requires a 32 bit word, which may be represented as numbered from 0 to 31, left to right. The first bit is the sign bit, S, the next eight bits are the exponent bits, 'E', and the final 23 bits are the fraction 'F'. Instead of the signed exponent E, the value stored is an unsigned integer E' = E + 127, called the excess-127 format. Therefore, E' is in the range 0 £ E' £ 255.

S E'E'E'E'E'E'E'E' FFFFFFFFFFFFFFFFFFFFFFF

0 1   8 9   31

The value V represented by the word may be determined as follows:

- If E' = 255 and F is nonzero, then V = NaN ("Not a number")
- If E' = 255 and F is zero and S is 1, then V = -Infinity
- If E' = 255 and F is zero and S is 0, then V = Infinity
- If 0 < E< 255 then V =(-1)**S * 2 ** (E-127) * (1.F) where "1.F" is intended to represent the binary number created by prefixing F with an implicit leading 1 and a binary point.
- If E' = 0 and F is nonzero, then V = (-1)**S * 2 ** (-126) * (0.F). These are "unnormalized" values.

- If E'= 0 and F is zero and S is 1, then V = -0
- If E' = 0 and F is zero and S is 0, then V = 0

For example,

0 00000000 00000000000000000000000 = 0

1 00000000 00000000000000000000000 = -0

0 11111111 00000000000000000000000 = Infinity

1 11111111 00000000000000000000000 = -Infinity

0 11111111 00000100000000000000000 = NaN

1 11111111 00100010001001010101010 = NaN

0 10000000 00000000000000000000000 = +1 * 2**(128-127) * 1.0 = 2

0 10000001 10100000000000000000000 = +1 * 2**(129-127) * 1.101 = 6.5

1 10000001 10100000000000000000000 = -1 * 2**(129-127) * 1.101 = -6.5

0 00000001 00000000000000000000000 = +1 * 2**(1-127) * 1.0 = 2**(-126)

0 00000000 10000000000000000000000 = +1 * 2**(-126) * 0.1 = 2**(-127)

0 00000000 00000000000000000000001 = +1 * 2**(-126) *

0.00000000000000000000001 = 2**(-149) (Smallest positive value)

(unnormalized values)

Double Precision Numbers:

The IEEE double precision floating point standard representation requires a 64-bit word, which may be represented as numbered from 0 to 63, left to right. The first bit is the sign bit, S, the next eleven bits are the excess-1023 exponent bits, E', and the final 52 bits are the fraction 'F':

S E'E'E'E'E'E'E'E'E'E'E'

FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

0 1 11 12

63

The value V represented by the word may be determined as follows:

- If E' = 2047 and F is nonzero, then V = NaN ("Not a number")
- If E'= 2047 and F is zero and S is 1, then V = -Infinity
- If E'= 2047 and F is zero and S is 0, then V = Infinity
- If 0 < E'< 2047 then V = (-1)**S * 2 ** (E-1023) * (1.F) where "1.F" is intended to represent the binary number created by prefixing F with an implicit leading 1 and a binary point.
- If E'= 0 and F is nonzero, then V = (-1)**S * 2 ** (-1022) * (0.F) These are "unnormalized" values.
- If E'= 0 and F is zero and S is 1, then V = – 0
- If E'= 0 and F is zero and S is 0, then V = 0

Arithmetic unit

Arithmetic operations on floating point numbers consist of addition, subtraction, multiplication and division. The operations are done with algorithms similar to those used on sign magnitude integers (because of the similarity of representation) — example, only add numbers of the same sign. If the numbers are of opposite sign, must do subtraction.

ADDITION

Example on decimal value given in scientific notation:

```
3.25 x 10 ** 3
+ 2.63 x 10 ** -1
———————————
```

first step: align decimal points

second step: add

```
3.25       x 10 ** 3
+  0.000263 x 10 ** 3
—————————————
3.250263 x 10 ** 3
```

(presumes use of infinite precision, without regard for accuracy)

third step: normalize the result (already normalized!)

Example on floating pt. value given in binary:

```
.25 =    0 01111101 00000000000000000000000
100 =    0 10000101 10010000000000000000000
```

To add these fl. pt. representations,

step 1: align radix points

shifting the mantissa left by 1 bit decreases the exponent by 1

shifting the mantissa right by 1 bit increases the exponent by 1

we want to shift the mantissa right, because the bits that fall off the end should come from the least significant end of the mantissa

-> choose to shift the .25, since we want to increase it's exponent.

```
-> shift by  10000101
-01111101
________
00001000    (8) places.
```

```
0 01111101 00000000000000000000000 (original value)
0 01111110 10000000000000000000000 (shifted 1 place)
(note that hidden bit is shifted into msb of mantissa)
0 01111111 01000000000000000000000 (shifted 2 places)
0 10000000 00100000000000000000000 (shifted 3 places)
0 10000001 00010000000000000000000 (shifted 4 places)
0 10000010 00001000000000000000000 (shifted 5 places)

0 10000011 00000100000000000000000 (shifted 6 places)
0 10000100 00000010000000000000000 (shifted 7 places)
0 10000101 00000001000000000000000 (shifted 8 places)
```

step 2: add (don't forget the hidden bit for the 100)

```
0 10000101 1.10010000000000000000000  (100)
+    0 10000101 0.00000001000000000000000  (.25)
_________________________________
0 10000101 1.10010001000000000000000
```

step 3: normalize the result (get the "hidden bit" to be a 1)

It already is for this example.

ılt is   )000101 10010001000000000000000

SUBTRACTION

Same as addition as far as alignment of radix points

Then the algorithm for subtraction of sign mag. numbers takes over.

before subtracting,

compare magnitudes (don’t forget the hidden bit!)

change sign bit if order of operands is changed.

don’t forget to normalize number afterward.

MULTIPLICATION

Example on decimal values given in scientific notation:

3.0 x 10 ** 1

+ 0.5 x 10 ** 2

____

Algorithm: multiply mantissas

add exponents

3.0 x 10 ** 1

+ 0.5 x 10 ** 2

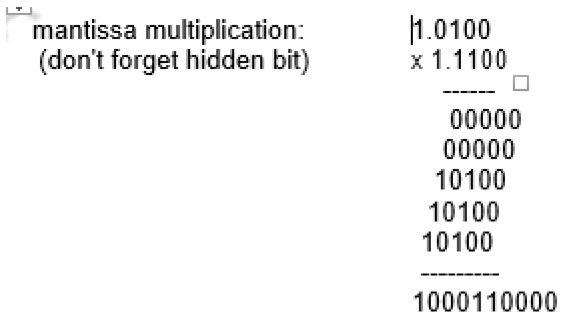
____

1.50 x 10 ** 3

Example in binary: Consider a mantissa that is only 4 bits.

0 10000100 0100

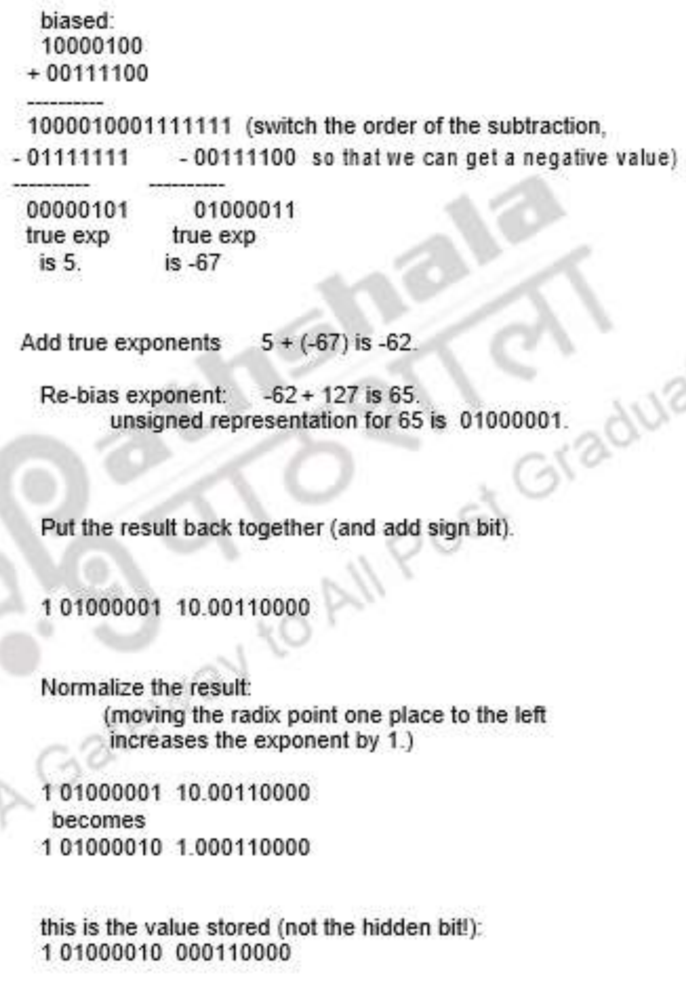
x 1 00111100 1100



```
mantissa multiplication:        1.0100
(don't forget hidden bit)       x 1.1100
                                   ------
                                    00000
                                   00000
                                  10100
                                 10100
                                10100
                                ---------
                               1000110000
```

Add exponents:

always add true exponents (otherwise the bias gets added in twice)



```
   biased:
   10000100
 + 00111100
 ----------
  1000010001111111  (switch the order of the subtraction,
- 01111111       - 00111100  so that we can get a negative value)
----------       ----------
 00000101         01000011
 true exp          true exp
   is 5.          is -67


Add true exponents      5 + (-67) is -62.

  Re-bias exponent:     -62 + 127 is 65.
          unsigned representation for 65 is  01000001.


  Put the result back together (and add sign bit).


  1 01000001  10.00110000


  Normalize the result:
          (moving the radix point one place to the left
           increases the exponent by 1.)

  1 01000001  10.00110000
    becomes
  1 01000010  1.000110000


  this is the value stored (not the hidden bit!):
  1 01000010  000110000
```

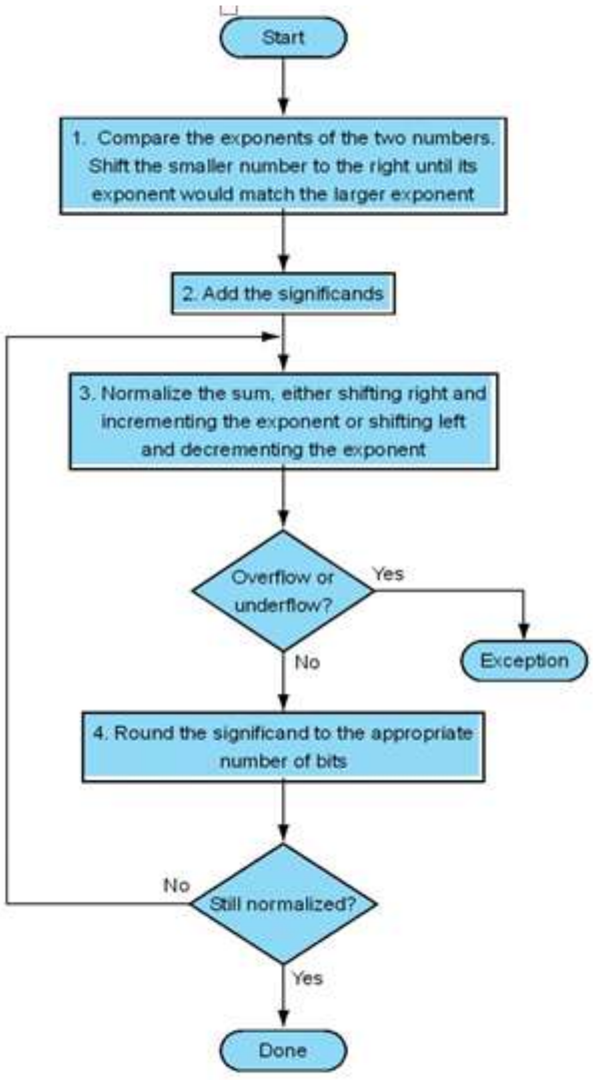
DIVISION

It is similar to multiplication.

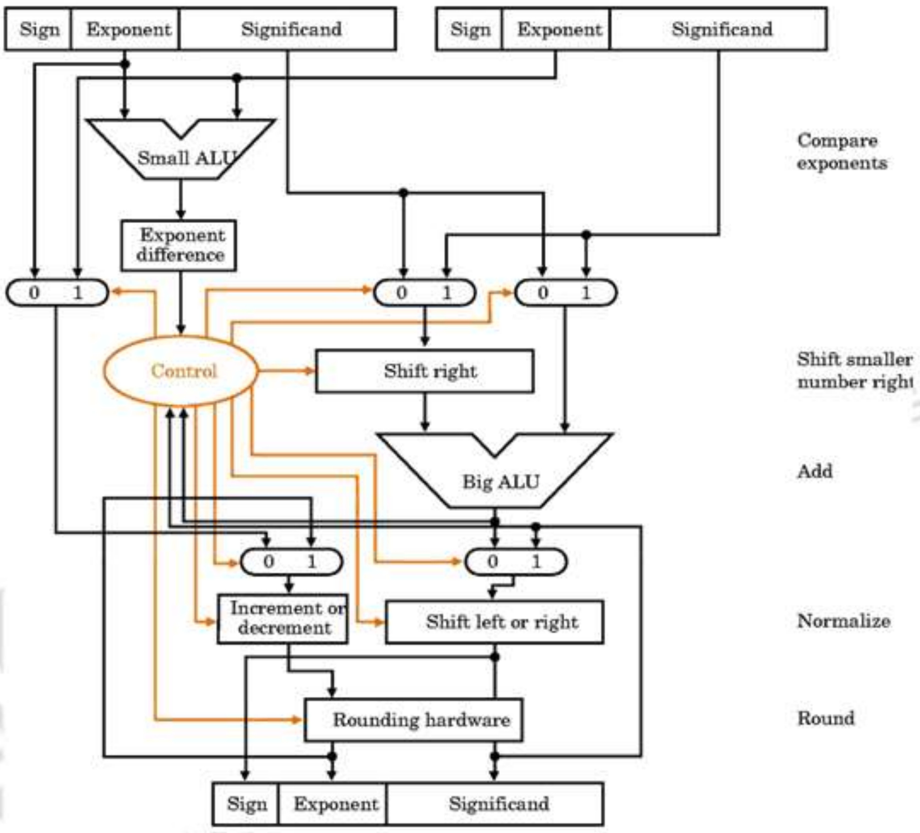
do unsigned division on the mantissas (don't forget the hidden bit)

subtract TRUE exponents

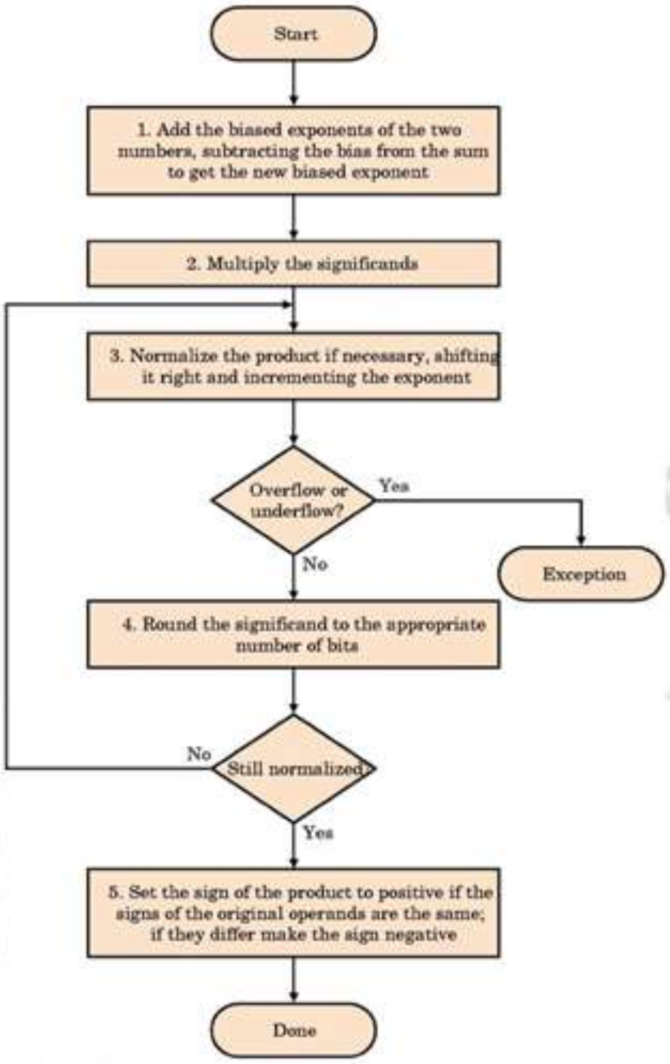
The organization of a floating point adder unit and the algorithm is given below.


Start
1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
Done

The floating point multiplication algorithm is given below. A similar algorithm based on the steps discussed before can be used for division.



Sign
Exponent
Significand
Sign
Exponent
Significand
Compare exponents
Small ALU
Exponent difference
0 1
0 1
0 1
Control
Shift right
Shift smaller number right
Big ALU
Add
0 1
0 1
Increment or decrement
Shift left or right
Normalize
Rounding hardware
Round
Sign
Exponent
Significand



Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
No
Exception
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

Rounding

The floating point arithmetic operations discussed above may produce a result with more digits than can be represented in 1.M. In such cases, the result must be *rounded* to fit into the available number of M positions. The extra bits that are used in intermediate calculations to improve the precision of the result are called *guard bits.* It is only a tradeoff of hardware cost (keeping extra bits) and speed versus accumulated rounding error, because finally these extra bits have to be rounded off to conform to the IEEE standard.

Rounding Methods:

- Truncate
  - Remove all digits beyond those supported
  - 1.00100 -> 1.00
- Round up to the next value
  - 1.00100 -> 1.01
- Round down to the previous value
  - 1.00100 -> 1.00
  - Differs from Truncate for negative numbers
- Round-to-nearest-even
  - Rounds to the even value (the one with an LSB of 0)
  - 1.00100 -> 1.00
  - 1.01100 -> 1.10
  - Produces zero average bias
  - Default mode

A product may have twice as many digits as the multiplier and multiplicand

- 1.11 x 1.01 = 10.0011

For round-to-nearest-even, we need to know the value to the right of the LSB (*round bit*) and whether any other digits to the right of the round digit are 1's (the *sticky bit* is the OR of these digits). The IEEE standard requires the use of 3 extra bits of less significance than the 24 bits (of mantissa) implied in the single precision representation – guard bit, round bit and sticky bit. When a mantissa is to be shifted in order to align radix points, the bits that fall off the least significant end of the mantissa go into these extra bits (guard, round, and sticky bits). These bits can also be set by the normalization step in multiplication, and by extra bits of quotient (remainder) in division. The guard and round bits are just 2 extra bits of precision that are used in calculations. The sticky bit is an indication of what is/could be in lesser significant bits that are not kept. If a value of 1 ever is shifted into the sticky bit position, that sticky bit remains a 1 ("sticks" at 1), despite further shifts.

General Register Organization

A set of flip-flops forms a register. A register is a unique high-speed storage area in the CPU. They include combinational circuits that implement data processing. The information is always defined in a register before processing. The registers speed up the implementation of programs.

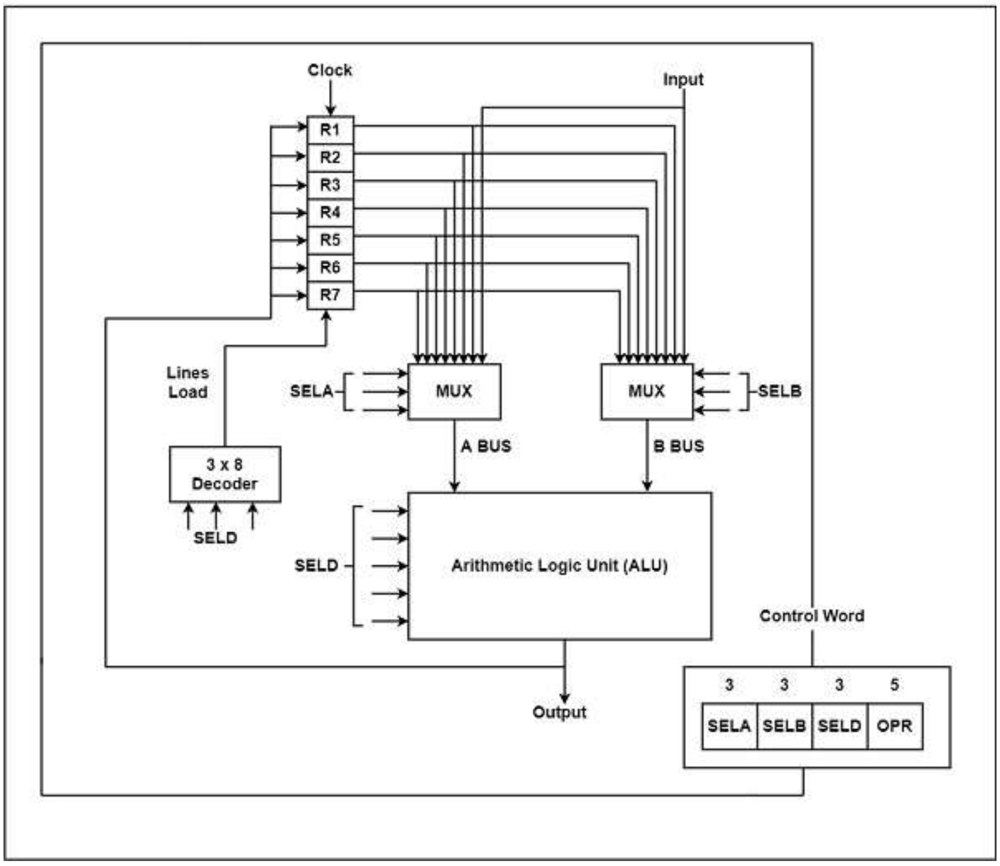
Registers implement two important functions in the CPU operation are as follows −

- It can support a temporary storage location for data. This supports the directly implementing programs to have fast access to the data if required.
- It can save the status of the CPU and data about the directly implementing program.

Example − Address of the next program instruction, signals get from the external devices and error messages, and including different data is saved in the registers.

If a CPU includes some registers, therefore a common bus can link these registers. A general organization of seven CPU registers is displayed in the figure.

General Organization of Registers



The CPU bus system is managed by the control unit. The control unit explicit the data flow through the ALU by choosing the function of the ALU and components of the system.

Consider R1 ← R2 + R3, the following are the functions implemented within the CPU −

MUX A Selector (SELA) − It can place R2 into bus A.

MUX B Selector (SELB) − It can place R3 into bus B.

ALU Operation Selector (OPR) − It can select the arithmetic addition (ADD).

Decoder Destination Selector (SELD) − It can transfers the result into R1.

The multiplexers of 3-state gates are performed with the buses. The state of 14 binary selection inputs determines the control word. The 14-bit control word defines a micro-operation.

The encoding of register selection fields is specified in the table.

Encoding of Register Selection Field

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	R1	R1	R1
010	R2	R2	R2

Binary Code	SELA	SELB	SELD
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

There are several micro-operations are implemented by the ALU. Few of the operations implemented by the ALU are displayed in the table.

Encoding of ALU Operations

OPR Select	Operation	Symbol
00000	Transfer A	TSFA
00001	Increment A	INCA
00010	Add A + B	ADD
00101	Subtract A - B	SUB
00110	Decrement A	DECA
01000	ADD A and B	AND
01010	OR A and B	OR
01100	XOR A and B	XOR
01110	Complement A	COMA
10000	Shift right A	SHRA
11000	Shift left A	SHLA

There are some ALU micro-operations are shown in the table.

ALU Micro-Operations

Micro-operation	SELA	SELB	SELD	OPR	Control Word			
R1 ← R2 – R3	R2	R3	R1	SUB	10	11	01	00101
R4 ← R4 ∨ R5	R4	R5	R4	OR	00	01	00	01010

R6 ← R6 + R1	-	R6	R1	INCA	10	00	10	00001
R7 ← R1	R1	-	R7	TSFA	01	00	11	00000
Output ← R2	R2	–	None	TSFA	10	00	00	00000
Output ← Input	Input	-	None	TSFA	00	00	00	00000
R4 ← shl R4	R4	-	R4	SHLA	00	00	00	11000
R5 ← 0	R5	R5	R5	XOR	01	01	01	01100

Stack Organization

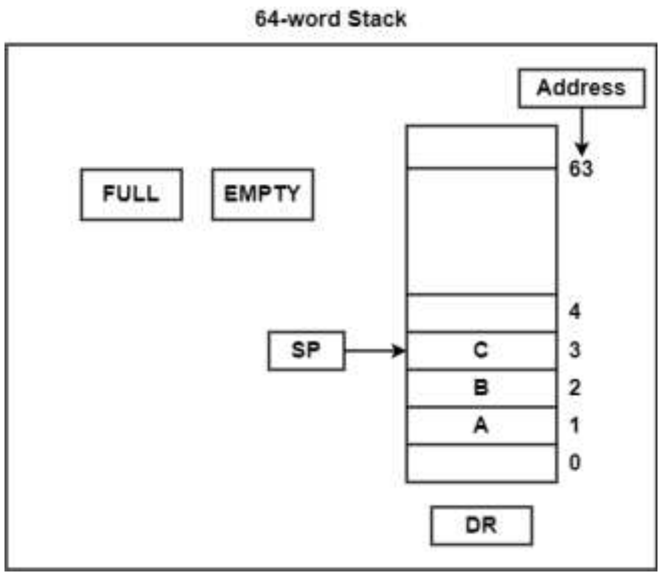
Stack is also known as the Last In First Out (LIFO) list. It is the most important feature in the CPU. It saves data such that the element stored last is retrieved first. A stack is a memory unit with an address register. This register influence the address for the stack, which is known as Stack Pointer (SP). The stack pointer continually influences the address of the element that is located at the top of the stack.

It can insert an element into or delete an element from the stack. The insertion operation is known as push operation and the deletion operation is known as pop operation. In a computer stack, these operations are simulated by incrementing or decrementing the SP register.

Register Stack

The stack can be arranged as a set of memory words or registers. Consider a 64-word register stack arranged as displayed in the figure. The stack pointer register includes a binary number, which is the address of the element present at the top of the stack. The three-element A, B, and C are located in the stack.

The element C is at the top of the stack and the stack pointer holds the address of C that is 3. The top element is popped from the stack through reading memory word at address 3 and decrementing the stack pointer by 1. Then, B is at the top of the stack and the SP holds the address of B that is 2. It can insert a new word, the stack is pushed by incrementing the stack pointer by 1 and inserting a word in that incremented location.



The stack pointer includes 6 bits, because $2^6 = 64$, and the SP cannot exceed 63 (111111 in binary). After all, if 63 is incremented by 1, therefore the result is 0(111111 + 1 = 1000000). SP holds only the six least significant bits. If 000000 is decremented by 1 thus the result is 111111.

Therefore, when the stack is full, the one-bit register 'FULL' is set to 1. If the stack is null, then the one-bit register 'EMTY' is set to 1. The data register DR holds the binary information which is composed into or readout of the stack.

First, the SP is set to 0, EMTY is set to 1, and FULL is set to 0. Now, as the stack is not full (FULL = 0), a new element is inserted using the push operation.

The push operation is executed as follows −

$SP \leftarrow SP + 1$	It can increment stack pointer
$K[SP] \leftarrow DR$	It can write element on top of the stack
If $(SP = 0)$ then $(FULL \leftarrow 1)$	Check if stack is full
$EMTY \leftarrow 0$	Mark the stack not empty

The stack pointer is incremented by 1 and the address of the next higher word is saved in the SP. The word from DR is inserted into the stack using the memory write operation. The first element is saved at address 1 and the final element is saved at address 0. If the stack pointer is at 0, then the stack is full and 'FULL' is set to 1. This is the condition when the SP was in location 63 and after incrementing SP, the final element is saved at address 0. During an element is saved at address 0, there are no more empty registers in the stack. The stack is full and the 'EMTY' is set to 0.

A new element is deleted from the stack if the stack is not empty (if EMTY = 0). The pop operation includes the following sequence of micro-operations −

$DR \leftarrow K[SP]$	It can read an element from the top of the stack
$SP \leftarrow SP - 1$	It can decrement the stack pointer
If $(SP = 0)$ then $(EMTY \leftarrow 1)$	Check if stack is empty
$FULL \leftarrow 0$	Mark the stack not full

The top element from the stack is read and transfer to DR and thus the stack pointer is decremented. If the stack pointer reaches 0, then the stack is empty and 'EMTY' is set to 1. This is the condition when the element in location 1 is read out and the SP is decremented by 1.

Addressing modes

The term addressing modes refers to the way in which the operand of an instruction is specified. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually executed.

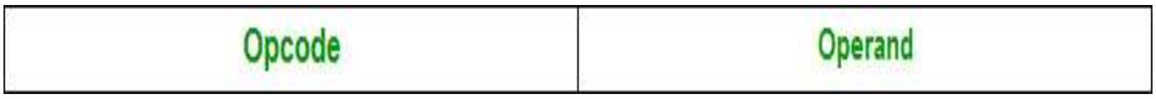
Addressing modes for 8086 instructions are divided into two categories:

1) Addressing modes for data

2) Addressing modes for branch

The 8086 memory addressing modes provide flexible access to memory, allowing you to easily access variables, arrays, records, pointers, and other complex data types. The key to good assembly language programming is the proper use of memory addressing modes.

An assembly language program instruction consists of two parts



Opcode	Operand

Types of Addressing Modes-

In computer architecture, there are following types of addressing modes-

1. Implied / Implicit Addressing Mode
2. Stack Addressing Mode
3. Immediate Addressing Mode
4. Direct Addressing Mode
5. Indirect Addressing Mode
6. Register Direct Addressing Mode
7. Register Indirect Addressing Mode
8. Relative Addressing Mode
9. Indexed Addressing Mode
10. Base Register Addressing Mode
11. Auto-Increment Addressing Mode
12. Auto-Decrement Addressing Mode

1. Implied Addressing Mode-

In this addressing mode,

- The definition of the instruction itself specify the operands implicitly.
- It is also called as **implicit addressing mode**.

Examples-

- The instruction "Complement Accumulator" is an implied mode instruction.
- In a stack organized computer, Zero Address Instructions are implied mode instructions.

(since operands are always implied to be present on the top of the stack)

2. Stack Addressing Mode-

In this addressing mode,

- The operand is contained at the top of the stack.

Example-

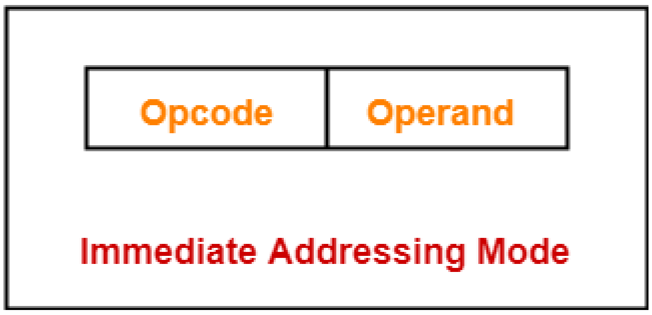
ADD

- This instruction simply pops out two symbols contained at the top of the stack.
- The addition of those two operands is performed.
- The result so obtained after addition is pushed again at the top of the stack.

3. Immediate Addressing Mode-

In this addressing mode,

- The operand is specified in the instruction explicitly.
- Instead of address field, an operand field is present that contains the operand.



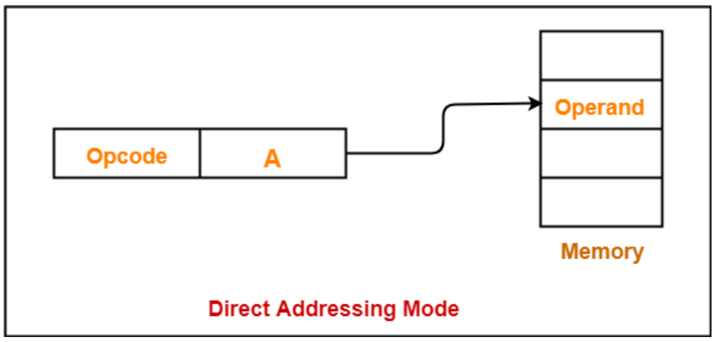
Examples-

- ADD 10 will increment the value stored in the accumulator by 10.
- MOV R #20 initializes register R to a constant value 20.

4. Direct Addressing Mode-

In this addressing mode,

- The address field of the instruction contains the effective address of the operand.
- Only one reference to memory is required to fetch the operand.
- It is also called as **absolute addressing mode**.



Example-

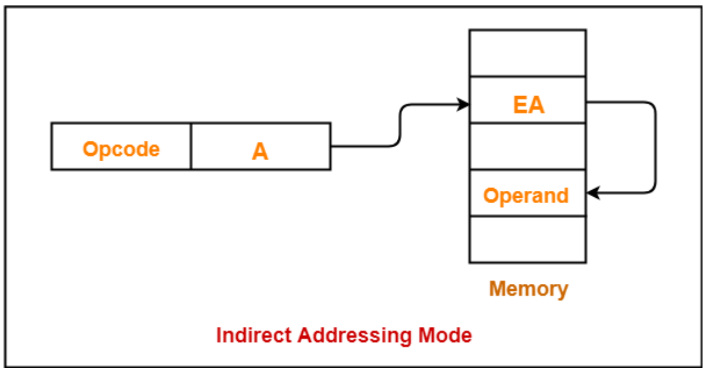
- ADD X will increment the value stored in the accumulator by the value stored at memory location X.

  AC ← AC + [X]

5. Indirect Addressing Mode-

In this addressing mode,

- The address field of the instruction specifies the address of memory location that contains the effective address of the operand.
- Two references to memory are required to fetch the operand.



Example-

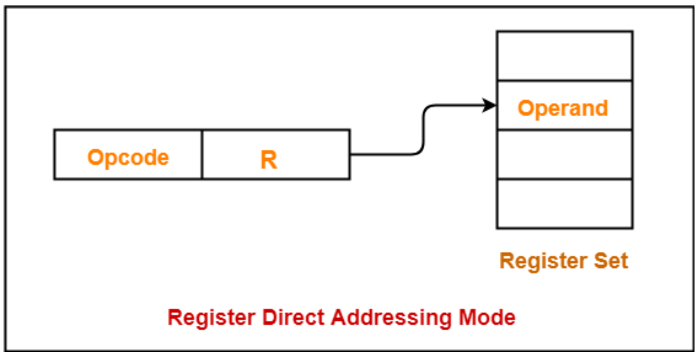
- ADD X will increment the value stored in the accumulator by the value stored at memory location specified by X.

AC ← AC + [[X]]

6. Register Direct Addressing Mode-

In this addressing mode,

- The operand is contained in a register set.
- The address field of the instruction refers to a CPU register that contains the operand.
- No reference to memory is required to fetch the operand.



Example-

- ADD R will increment the value stored in the accumulator by the content of register R.

AC ← AC + [R]

NOTE-

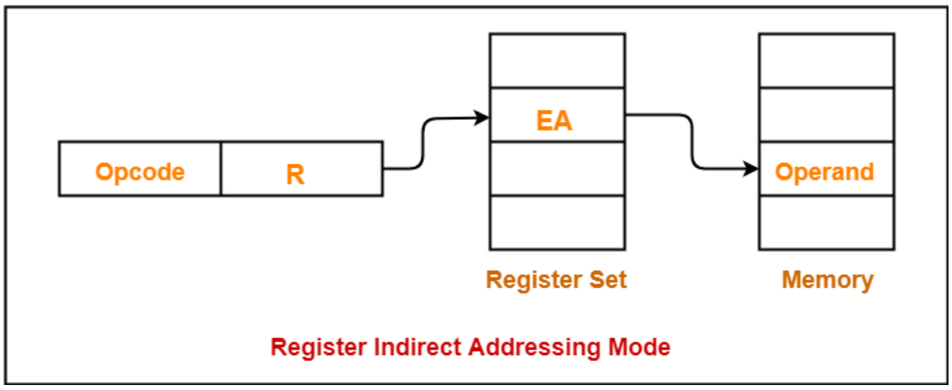
It is interesting to note-

- This addressing mode is similar to direct addressing mode.
- The only difference is address field of the instruction refers to a CPU register instead of main memory.

7. Register Indirect Addressing Mode-

In this addressing mode,

- The address field of the instruction refers to a CPU register that contains the effective address of the operand.
- Only one reference to memory is required to fetch the operand.



Example-

- ADD R will increment the value stored in the accumulator by the content of memory location specified in register R.

AC ← AC + [[R]]

NOTE-

It is interesting to note-

- This addressing mode is similar to indirect addressing mode.

- The only difference is address field of the instruction refers to a CPU register.

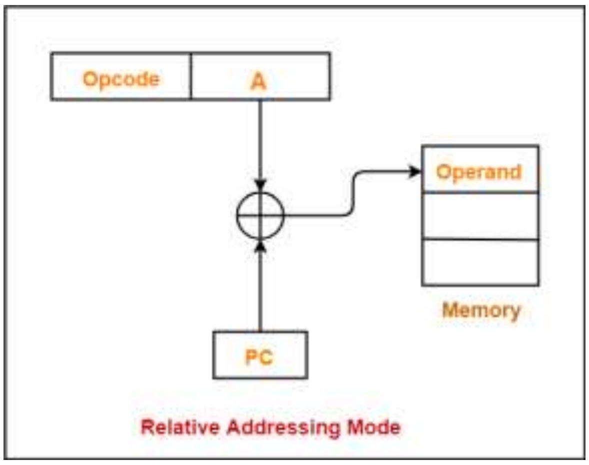
8. Relative Addressing Mode-

In this addressing mode,

- Effective address of the operand is obtained by adding the content of program counter with the address part of the instruction.

Effective Address

= Content of Program Counter + Address part of the instruction



NOTE-

- **Program counter** (PC) always contains the address of the next instruction to be executed.
- After fetching the address of the instruction, the value of program counter immediately increases.
- The value increases irrespective of whether the fetched instruction has completely executed or not.

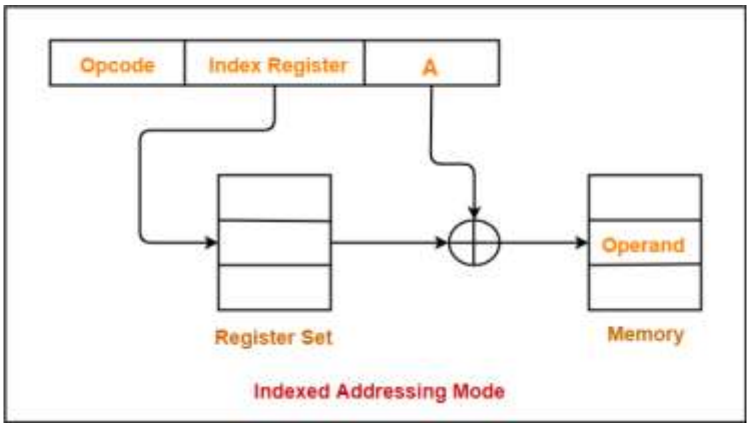
9. Indexed Addressing Mode-

In this addressing mode,

- Effective address of the operand is obtained by adding the content of index register with the address part of the instruction.

Effective Address

= Content of Index Register + Address part of the instruction



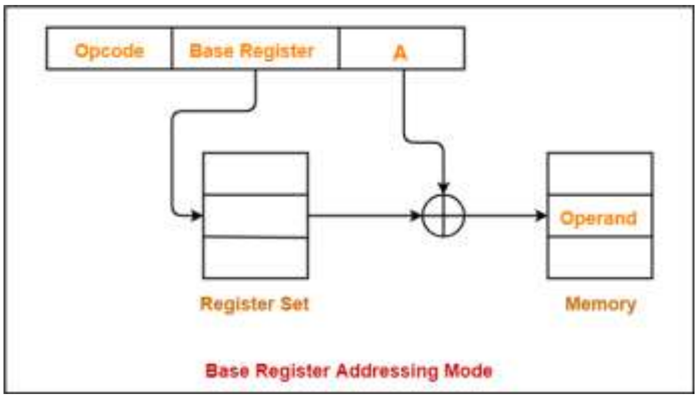
10. Base Register Addressing Mode-

In this addressing mode,

- Effective address of the operand is obtained by adding the content of base register with the address part of the instruction.

Effective Address

= Content of Base Register + Address part of the instruction



11. Auto-Increment Addressing Mode-

- This addressing mode is a special case of Register Indirect Addressing Mode where-

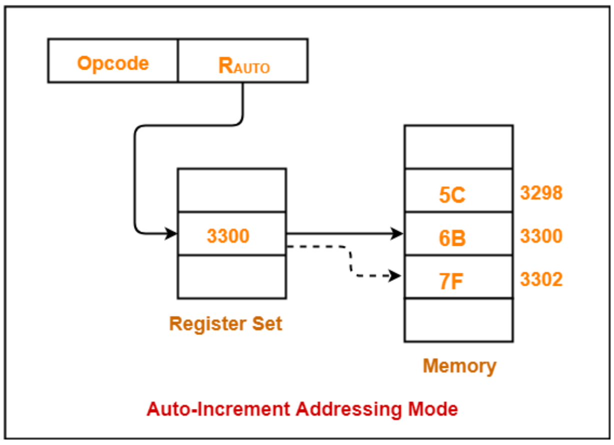
Effective Address of the Operand

= Content of Register

In this addressing mode,

- After accessing the operand, the content of the register is automatically incremented by step size ‘d’.
- Step size ‘d’ depends on the size of operand accessed.
- Only one reference to memory is required to fetch the operand.

Example-



Assume operand size = 2 bytes.

Here,

- After fetching the operand 6B, the instruction register R_{AUTO} will be automatically incremented by 2.
- Then, updated value of R_{AUTO} will be 3300 + 2 = 3302.

- At memory address 3302, the next operand will be found.

NOTE-

In auto-increment addressing mode,

- First, the operand value is fetched.
- Then, the instruction register R_{AUTO} value is incremented by step size ‘d’.

12. Auto-Decrement Addressing Mode-

- This addressing mode is again a special case of Register Indirect Addressing Mode where-

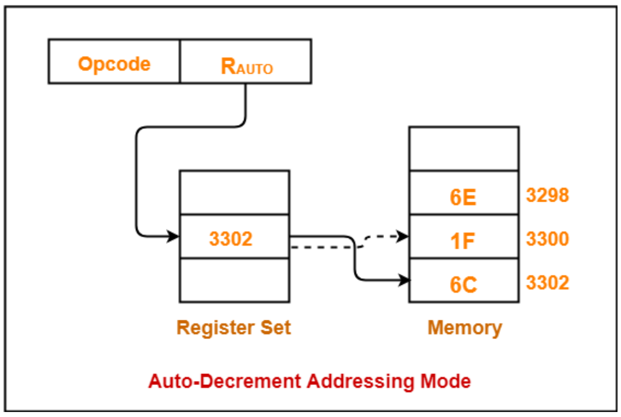
Effective Address of the Operand

= Content of Register – Step Size

In this addressing mode,

- First, the content of the register is decremented by step size ‘d’.
- Step size ‘d’ depends on the size of operand accessed.
- After decrementing, the operand is read.
- Only one reference to memory is required to fetch the operand.

Example-



Assume operand size = 2 bytes.

Here,

- First, the instruction register R_{AUTO} will be decremented by 2.
- Then, updated value of R_{AUTO} will be 3302 – 2 = 3300.
- At memory address 3300, the operand will be found.

NOTE-

In auto-decrement addressing mode,

- First, the instruction register R_{AUTO} value is decremented by step size ‘d’.
- Then, the operand value is fetched.

Applications of Addressing Modes-

Addressing Modes	Applications
Immediate Addressing Mode	• To initialize registers to a constant value

Direct Addressing Mode and Register Direct Addressing Mode	• To access static data • To implement variables
Indirect Addressing Mode and egister Indirect Addressing Mode	To implement pointers because pointers are memory locations that store the address of another variable pass array as a parameter because array name is the base address and pointer is needed to point the address
Relative Addressing Mode	• For program relocation at run time i.e. for position independent code • To change the normal sequence of execution of instructions For branch type instructions since it directly updates the program counter
Index Addressing Mode	• For array implementation or array addressing • For records implementation
Base Register Addressing Mode	r writing relocatable code i.e. for relocation of program in memory even at run time • For handling recursive procedures
uto-increment Addressing Mode and uto-decrement Addressing Mode	• For implementing loops • For stepping through arrays in a loop • For implementing a stack as push and pop

Advantages of Addressing Modes

1.To give programmers to facilities such as Pointers, counters for loop controls, indexing of data and program relocation.

2.To reduce the number bits in the addressing field of the Instruction.

Instruction types

A computer performs a task based on the instruction provided. Instruction in computers comprises groups called fields. These fields contain different information as for computers everything is in 0 and 1 so each field has different significance based on which a CPU decides what to perform. The most common fields are:

- Operation field specifies the operation to be performed like addition.
- Address field which contains the location of the operand, i.e., register or memory location.
- Mode field which specifies how operand is to be founded.

Instruction is of variable length depending upon the number of addresses it contains. Generally, CPU organization is of three types based on the number of address fields:

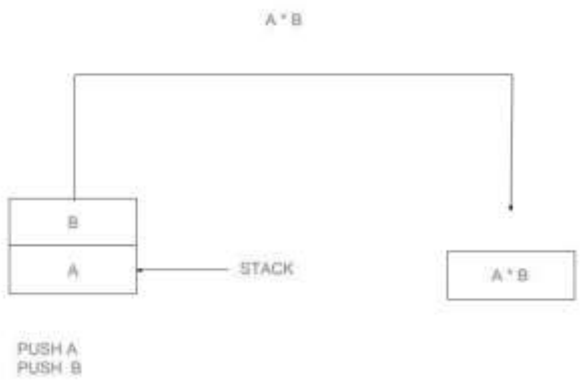
1. Single Accumulator organization
2. General register organization
3. Stack organization

In the first organization, the operation is done involving a special register called the accumulator. In second on multiple registers are used for the computation purpose. In the third organization the work on stack basis operation due to which it does not contain any address field. Only a single organization doesn't need to be applied, a blend of various organizations is mostly what we see generally.

Based on the number of address, instructions are classified as:

Note that we will use X = (A+B)*(C+D) expression to showcase the procedure.

1. **Zero Address Instructions –**



A stack-based computer does not use the address field in the instruction. To evaluate an expression first it is converted to reverse Polish Notation i.e. Postfix Notation.

Expression: X = (A+B)*(C+D)

Postfixed : X = AB+CD+*

TOP means top of stack

M[X] is any memory location

PUSH	A	TOP = A
PUSH	B	TOP = B
ADD		TOP = A+B
PUSH	C	TOP = C
PUSH	D	TOP = D
ADD		TOP = C+D
MUL		TOP = (C+D)*(A+B)
POP	X	M[X] = TOP

2. One Address Instructions –

This uses an implied ACCUMULATOR register for data manipulation. One operand is in the accumulator and the other is in the register or memory location. Implied means that the CPU already knows that one operand is in the accumulator so there is no need to specify it.

opcode	operand/address of operand	mode

Expression: X = (A+B)*(C+D)

AC is accumulator

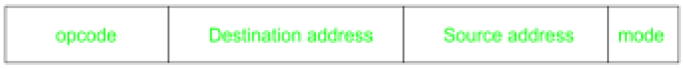
M[] is any memory location

M[T] is temporary location

LOAD	A	AC = M[A]
ADD	B	AC = AC + M[B]
STORE	T	M[T] = AC
LOAD	C	AC = M[C]
ADD	D	AC = AC + M[D]
MUL	T	AC = AC * M[T]
STORE	X	M[X] = AC

3.Two Address Instructions –

This is common in commercial computers. Here two addresses can be specified in the instruction. Unlike earlier in one address instruction, the result was stored in the accumulator, here the result can be stored at different locations rather than just accumulators, but require more number of bit to represent address.



Here destination address can also contain operand.

Expression: X = (A+B)*(C+D)

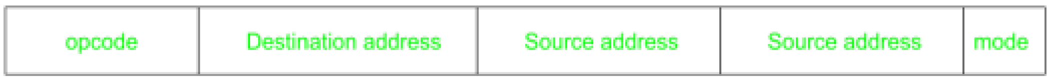
R1, R2 are registers

M[] is any memory location

MOV	R1, A	R1 = M[A]
ADD	R1, B	R1 = R1 + M[B]
MOV	R2, C	R2 = C
ADD	R2, D	R2 = R2 + D
MUL	R1, R2	R1 = R1 * R2
MOV	X, R1	M[X] = R1

4.Three Address Instructions –

This has three address field to specify a register or a memory location. Program created are much short in size but number of bits per instruction increase. These instructions make creation of program much easier but it does not mean that program will run much faster because now instruction only contain more information but each micro operation (changing content of register, loading address in address bus etc.) will be performed in one cycle only.



Expression: X = (A+B)*(C+D)

R1, R2 are registers

M[] is any memory location

ADD	R1, A, B	R1 = M[A] + M[B]
ADD	R2, C, D	R2 = M[C] + M[D]
MUL	X, R1, R2	M[X] = R1 * R2

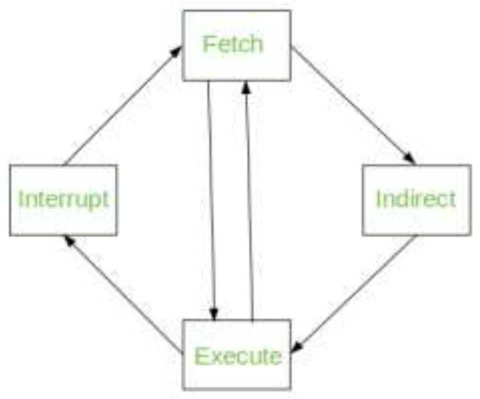
Instruction Cycle

Registers Involved in Each Instruction Cycle:

- **Memory address registers(MAR)** : It is connected to the address lines of the system bus. It specifies the address in memory for a read or write operation.
- **Memory Buffer Register(MBR)** : It is connected to the data lines of the system bus. It contains the value to be stored in memory or the last value read from the memory.
- **Program Counter(PC)** : Holds the address of the next instruction to be fetched.
- **Instruction Register(IR)** : Holds the last instruction fetched.

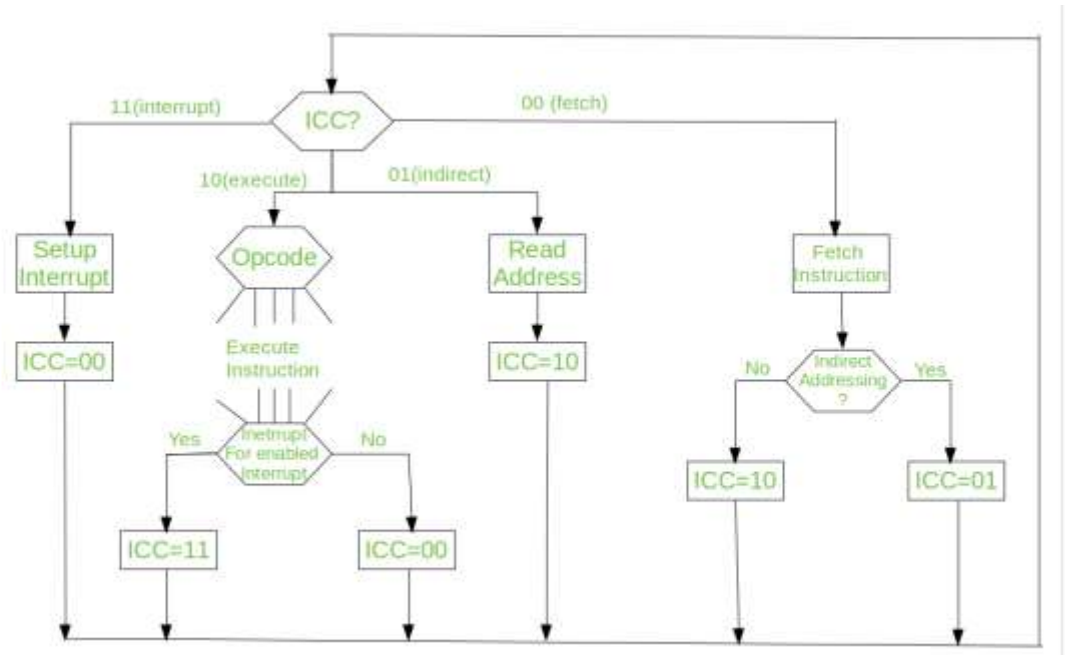
The Instruction Cycle –

Each phase of Instruction Cycle can be decomposed into a sequence of elementary micro-operations. In the above examples, there is one sequence each for the *Fetch, Indirect, Execute and Interrupt Cycles.*



The Instruction Cycle

The *Indirect Cycle* is always followed by the *Execute Cycle*. The *Interrupt Cycle* is always followed by the *Fetch Cycle*. For both fetch and execute cycles, the next cycle depends on the state of the system.



Flowchart for Instruction Cycle

We assumed a new 2-bit register called *Instruction Cycle Code* (ICC). The ICC designates the state of processor in terms of which portion of the cycle it is in:-

00 : Fetch Cycle

01 : Indirect Cycle

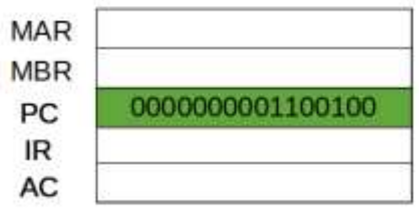
10 : Execute Cycle

11 : Interrupt Cycle

At the end of the each cycles, the ICC is set appropriately. The above flowchart of *Instruction Cycle* describes the complete sequence of micro-operations, depending only on the instruction sequence and the interrupt pattern (this is a simplified example). The operation of the processor is described as the performance of a sequence of micro-operation.

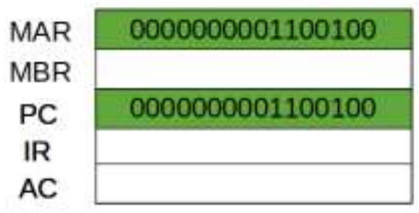
Different Instruction Cycles:

- **The Fetch Cycle –**
  At the beginning of the fetch cycle, the address of the next instruction to be executed is in the *Program Counter*(PC).



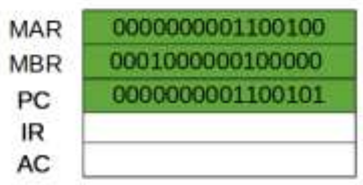
BEGINNING

- Step 1: The address in the program counter is moved to the memory address register(MAR), as this is the only register which is connected to address lines of the system bus.



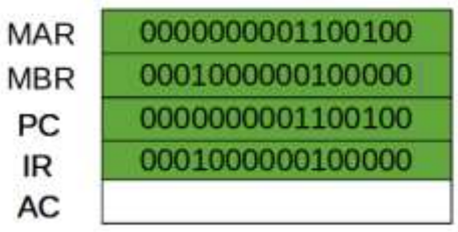
FIRST STEP

- Step 2: The address in MAR is placed on the address bus, now the control unit issues a READ command on the control bus, and the result appears on the data bus and is then copied into the memory buffer register(MBR). Program counter is incremented by one, to get ready for the next instruction. (These two action can be performed simultaneously to save time)



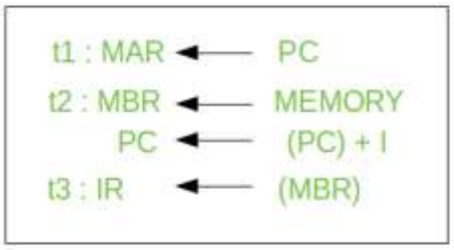
SECOND STEP

- Step 3: The content of the MBR is moved to the instruction register(IR).



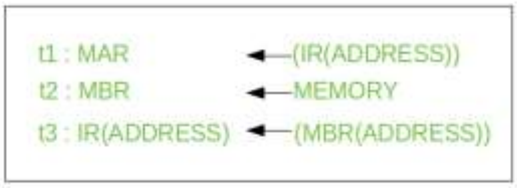
FIRST STEP

- Thus, a simple *Fetch Cycle* consist of three steps and four micro-operation. Symbolically, we can write these sequence of events as follows:-



$$
\begin{aligned}
t1 &: MAR \leftarrow PC \\
t2 &: MBR \leftarrow MEMORY \\
&\quad PC \leftarrow (PC) + I \\
t3 &: IR \leftarrow (MBR)
\end{aligned}
$$

- Here 'I' is the instruction length. The notations (t1, t2, t3) represents successive time units. We assume that a clock is available for timing purposes and it emits regularly spaced clock pulses. Each clock pulse defines a time unit. Thus, all time units are of equal duration. Each micro-operation can be performed within the time of a single time unit.
  First time unit: Move the contents of the PC to MAR.
  Second time unit: Move contents of memory location specified by MAR to MBR. Increment content of PC by I.
  Third time unit: Move contents of MBR to IR.
  Note: Second and third micro-operations both take place during the second time unit.
- **The Indirect Cycles –**
  Once an instruction is fetched, the next step is to fetch source operands. *Source Operand* is being fetched by indirect addressing(it can be fetched by any addressing mode, here its done by indirect addressing). Register-based operands need not be fetched. Once the opcode is executed, a similar process may be needed to store the result in main memory. Following *micro-operations* takes place:-



$$
\begin{aligned}
t1 &: MAR \leftarrow (IR(ADDRESS)) \\
t2 &: MBR \leftarrow MEMORY \\
t3 &: IR(ADDRESS) \leftarrow (MBR(ADDRESS))
\end{aligned}
$$

- Step 1: The address field of the instruction is transferred to the MAR. This is used to fetch the address of the operand.

Step 2: The address field of the IR is updated from the MBR.(So that it now contains a direct addressing rather than indirect addressing)

Step 3: The IR is now in the state, as if indirect addressing has not been occurred.

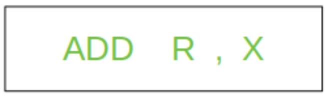
Note: Now IR is ready for the execute cycle, but it skips that cycle for a moment to consider the *Interrupt Cycle* .

- **The Execute Cycle**

  The other three cycles (*Fetch, Indirect and Interrupt*) are simple and predictable. Each of them requires simple, small and fixed sequence of micro-operation. In each case same micro-operation are repeated each time around. Execute Cycle is different from them. Like, for a machine with N different opcodes there are N different sequence of micro-operations that can occur.

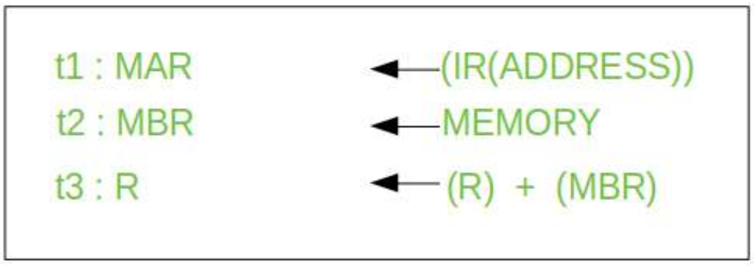
  Lets take an hypothetical example :-

  consider an add instruction:



ADD R , X

- Here, this instruction adds the content of location X to register R. Corresponding micro-operation will be:-



t1 : MAR $\leftarrow$ (IR(ADDRESS))

t2 : MBR $\leftarrow$ MEMORY

t3 : R $\leftarrow$ (R) + (MBR)

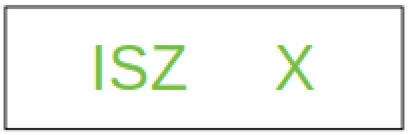
- We begin with the IR containing the ADD instruction.

  Step 1: The address portion of IR is loaded into the MAR.

  Step 2: The address field of the IR is updated from the MBR, so the reference memory location is read.

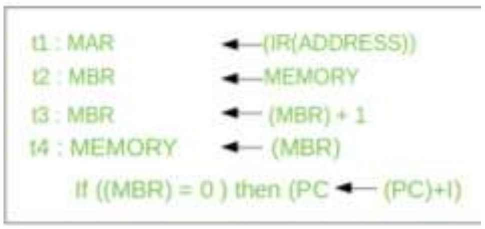
  Step 3: Now, the contents of R and MBR are added by the ALU.

  Lets take a complex example :-



ISZ X

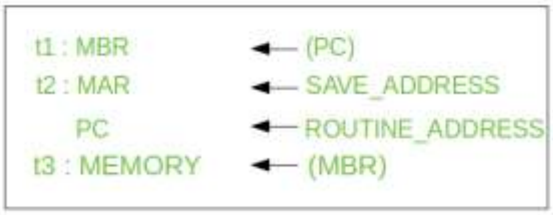
- Here, the content of location X is incremented by 1. If the result is 0, the next instruction will be skipped. Corresponding sequence of micro-operation will be :-



```
t1 : MAR        ←(IR(ADDRESS))
t2 : MBR        ←MEMORY
t3 : MBR        ← (MBR) + 1
t4 : MEMORY     ← (MBR)
     If ((MBR) = 0 ) then (PC ← (PC)+I)
```

- Here, the PC is incremented if (MBR) = 0. This test (is MBR equal to zero or not) and action (PC is incremented by 1) can be implemented as one micro-operation.
  Note : This test and action micro-operation can be performed during the same time unit during which the updated value MBR is stored back to memory.

- **The Interrupt Cycle**:
  At the completion of the Execute Cycle, a test is made to determine whether any enabled interrupt has occurred or not. If an enabled interrupt has occurred then Interrupt Cycle occurs. The nature of this cycle varies greatly from one machine to another.
  Lets take a sequence of micro-operation:-



```
t1 : MBR        ← (PC)
t2 : MAR        ← SAVE_ADDRESS
     PC         ← ROUTINE_ADDRESS
t3 : MEMORY     ← (MBR)
```

Step 1: Contents of the PC is transferred to the MBR, so that they can be saved for return.
Step 2: MAR is loaded with the address at which the contents of the PC are to be saved.
PC is loaded with the address of the start of the interrupt-processing routine.
Step 3: MBR, containing the old value of PC, is stored in memory.
Note: In step 2, two actions are implemented as one micro-operation. However, most processor provide multiple types of interrupts, it may take one or more micro-operation to obtain the save_address and the routine_address before they are transferred to the MAR and PC respectively.

Micro-Operation

The execution of a program consists of the sequential execution of instructions. Each instruction is executed during an instruction cycle made up of shorter sub-cycles (e.g., fetch, indirect, execute, interrupt). The performance of each sub-cycle involves one or more shorter operations, that is, micro-operations.
Micro-operations are the functional, or atomic, operations of a processor. In this section, we will examine micro-operations to gain an understanding of how the events of any instruction cycle can be described as a sequence of such micro-operations.

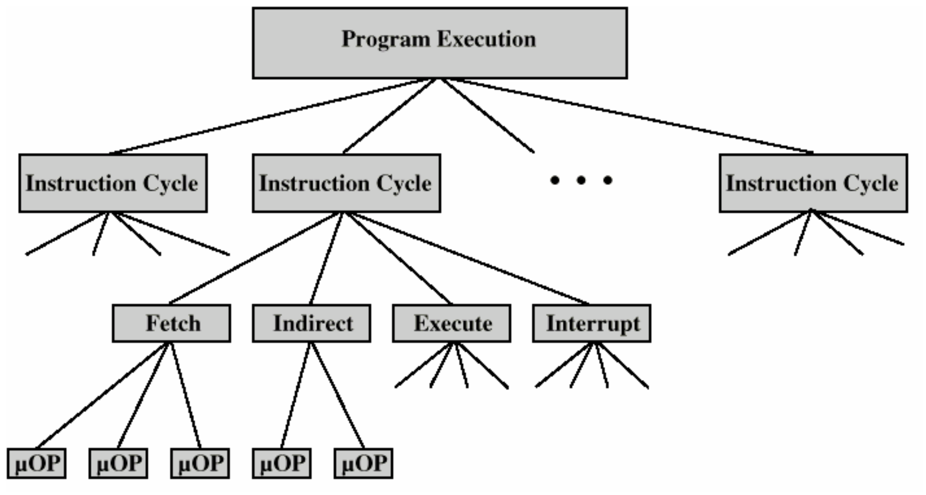


Figure- Constituent Elements of Program Execution

1.1 The Fetch Cycle

We begin by looking at the fetch cycle, which occurs at the beginning of each instruction cycle and causes an instruction to be fetched from memory. Four registers are involved:

- Memory address register (MAR): Is connected to the address lines of the system bus. It specifies the address in memory for a read or write operation.
- Memory buffer register (MBR): Is connected to the data lines of the system bus. It contains the value to be stored in memory or the last value read from memory.
- Program counter (PC): Holds the address of the next instruction to be fetched.
- Instruction register (IR): Holds the last instruction fetched.

Let us look at the sequence of events for the fetch cycle from the point of view of its effect on the processor registers. An example appears.

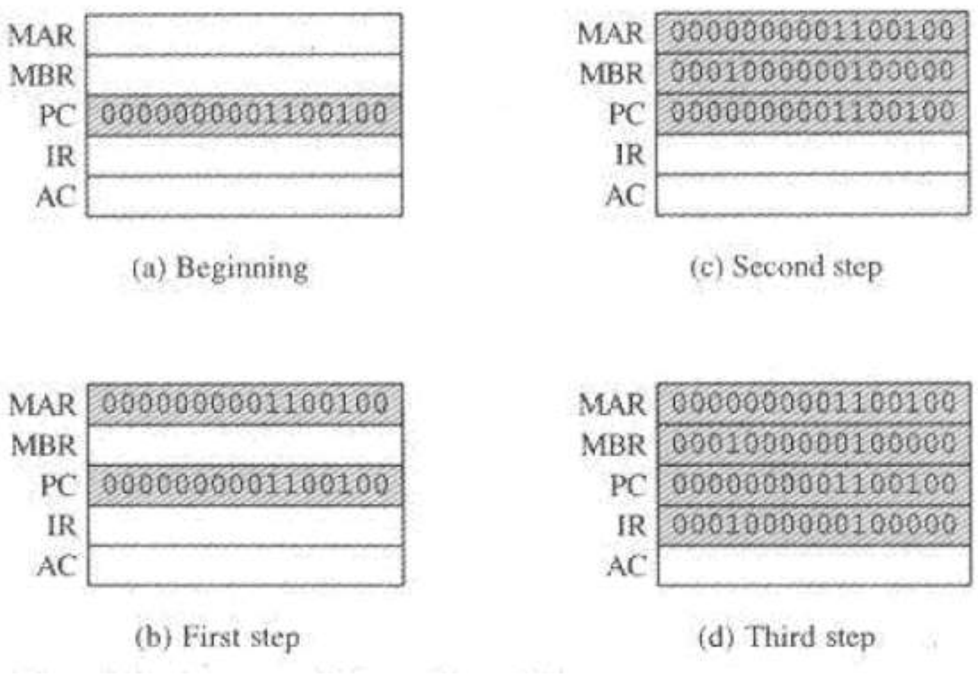


Figure-Sequence of Events, Fetch Cycle

- At the beginning of the fetch cycle, the address of the next instruction to be executed is in the program counter (PC); in this case, the address is 1100100.
- The first step is to move that address to the memory address register (MAR) because this is the only register connected lo the address lines of the system bus.

- The second step is to bring in the instruction. The desired address (in the MAR) is placed on the address bus, the control unit issues a READ command on the control bus, and the result appears on the data bus and is copied into the memory buffer register (MBR). We also need to increment the PC by 1 to get ready for the next instruction. Because these two actions (read word from memory, add 1 to PC) do not interfere with each other, we can do them simultaneously to save time.
- The third step is to move the contents of the MBR to the instruction register (IR). This frees up the MBR for use during a possible indirect cycle.

  Thus, the simple fetch cycle actually consists of three steps and four micro-operations. Each micro-operation involves the movement of data into or out of a register. So long as these movements do not interfere with one another, several of them can take place during one step, saving lime. Symbolically, we can write this sequence of events as follows:

  t1: MAR <= (PC)

  t2: MBR <= Memory

  PC <= (PC) + l

  t3: IR <= (MBR)

  where l is the instruction length. We need to make several comments about this sequence. We assume that a clock is available for timing purposes and that it emits regularly spaced clock pulses. Each clock pulse defines a time unit. Thus, all time units are of equal duration. Each micro-operation can be performed within the time of a single time unit. The notation (t1, t2, t3) represents successive time units. In words, we have
- First time unit: Move contents of PC to MAR.
- Second time unit:
  - Move contents of memory location specified by MAR to MBR.
  - Increment by l the contents of the PC.
- Third time unit: Move contents of MBR to IR.

  Note that the second and third micro-operations both take place during the second time unit. The third micro-operation could have been grouped with the fourth without affecting the fetch operation:

  t1: MAR <= (PC)

  t2: MBR <= Memory

  t3: PC <= (PC) + l

  IR <= (MBR)

  The groupings of micro-operations must follow two simple rules:

  1. The proper sequence of events must be followed. Thus (MAR <= (PC)) must precede (MBR <= Memory) because the memory read operation makes use of the address in the MAR.

  2. Conflicts must be avoided. One should not attempt to read to and write from the same register in one time unit, because the results would be unpredictable. For example, the micro-operations (MBR <= Memory) and (IR <= MBR) should not occur during the same time unit.

  A final point worth noting is that one of the micro-operations involves an addition. To avoid duplication of circuitry, this addition could be performed by the ALU. The use of the ALU may involve additional micro-operations, depending on the functionality of the ALU and the organization of the processor.

1.2 The Indirect Cycle

Once an instruction is fetched, the next step is to fetch source operands. Continuing our simple example, let us assume a one-address instruction format, with direct and indirect addressing allowed. If the instruction specifies an indirect address, then an indirect cycle must precede the execute cycle. The data flow includes the following micro-operations:

t1: MAR <= (IR (Address))

t2: MBR <= Memory

t3: IR(Address) <= (MBR(Address))

The address field of the instruction is transferred to the MAR. This is then used to fetch the address of the operand. Finally, the address field of the IR is updated from the MBR, so that it now contains a direct rather than an indirect address.

The IR is now in the same state as if indirect addressing had not been used, and it is ready for the execute cycle. We skip that cycle for a moment, to consider the interrupt cycle.

1.3 The Interrupt Cycle

At the completion of the execute cycle, a test is made to determine whether any enabled interrupts have occurred. If so, the interrupt cycle occurs. The nature of this cycle varies greatly from one machine to another. We present a very simple sequence of events, we have

t1 : MBR <= (PC)

t2 : MAR <= Save_Address

PC <= Routine_Address

t3: Memory <= (MBR)

In the first step, the contents of the PC are transferred to the MBR, so that they can be saved for return from the interrupt. Then the MAR is loaded with the address at which the contents of the PC are to be saved, and the PC is loaded with the address of the start of the interrupt-processing routine. These two actions may each be a single micro-operation. However, because most processors provide multiple types and/or levels of interrupts, it may lake one or more additional micro-operations to obtain the save_address and the routine_address before they can be transferred to the MAR and PC, respectively. In any case, once this is done, the final step is to store the MBR, which contains the old value of the PC, into memory. The processor is now ready to begin the next instruction cycle.

1.4 The Execute Cycle

The fetch, indirect, and interrupt cycles are simple and predictable. Each involves a small, fixed sequence of micro-operations and, in each ease, the same micro-operations are repealed each time around. This is not true of the execute cycle. For a machine with N different opcodes, there are N different sequences of micro-operations that can occur. Let us consider several hypothetical examples.

First, consider an add instruction:

ADD R1, X

which adds the contents of the location X to register Rl. The following sequence of micro-operations might occur:

t1: MAR <= (IR(address))

t2: MBR <= Memory

t3: Rl <= (Rl) + (MBR)

We begin with the IR containing the ADD instruction. In the first step, the address portion of the IR is loaded into the MAR. Then the referenced memory location is read. Finally, the contents of R1 and MBR are added by the ALU. Again, this is a simplified example. Additional micro-operations may be required to extract the register reference from the IR and perhaps to stage the ALU inputs or outputs in some intermediate registers.Let us look at two more complex examples. A common instruction is increment and skip if zero:

ISZ X

The content of location X is incremented by 1. If the result is 0, the next instruction is skipped. A possible sequence of micro-operations is

t1: MAR <= (CR(address))

t2: MBR <= Memory

t3: MBR <= (MBR) - 1

t4: Memory <= (MBR)

If ((MBR) = 0) then (PC <= (PC) + I)

The new feature introduced here is the conditional action. The PC is incremented if (MBR) = 0; this test and action can be implemented as one micro-operation. Note also that this micro-operation can be performed during the same time unit during which the updated value in MBR is stored back to memory.

Finally, consider a subroutine call instruction. As an example, consider a branch-and-save-address instruction:

BSA X

The address of the instruction that follows the BSA instruction is saved in location X, and execution continues al location X - l. The saved address will later be used for return. This is a straightforward technique for providing subroutine calls. the following micro-operations suffice:

t1 : MAR <= (IR(address))

MBR <= (PC)

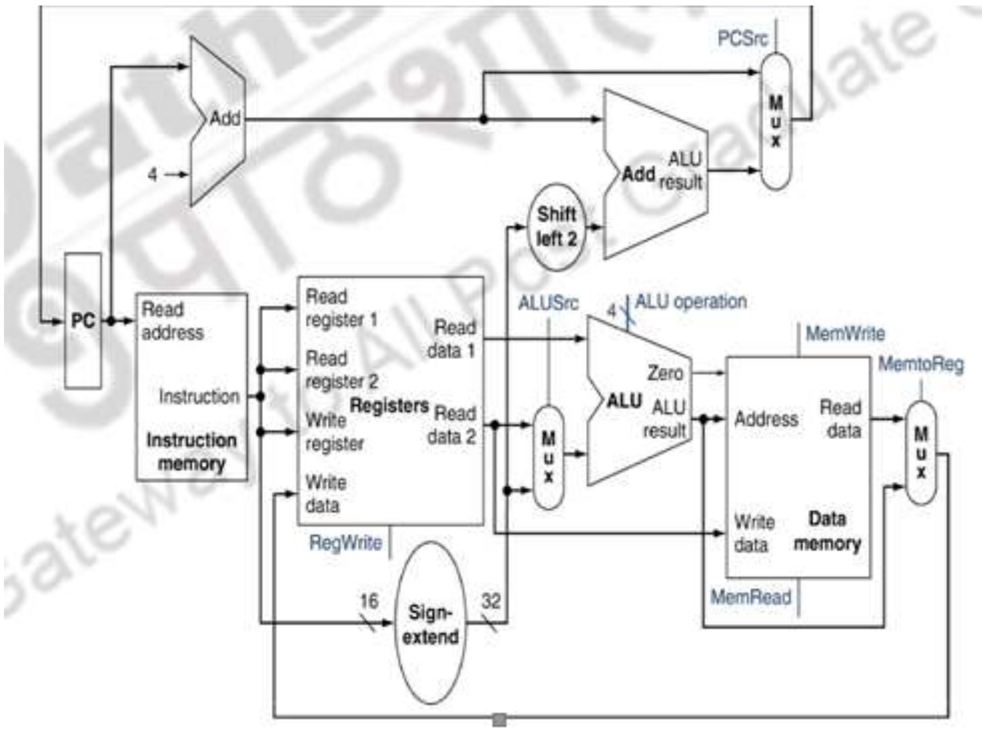
t2: PC <= (IR(address)) Memory <= (MBR)

t3: PC <= (PC) + I

The address in the PC at the start of the instruction is the address of the next instruction in sequence. This is saved at the address designated in Ihe IK. The latter address is also incremented to provide the address of the instruction for the next instruction cycle.

Execution of a complete instruction

The control unit must be capable of taking inputs about the instruction and generate all the control signals necessary for executing that instruction, for eg. the write signal for each state element, the selector control signal for each multiplexor, the ALU control signals, etc. Figure 9.1 below shows the complete data path implementation for the MIPS architecture along with an indication of the various control signals required.



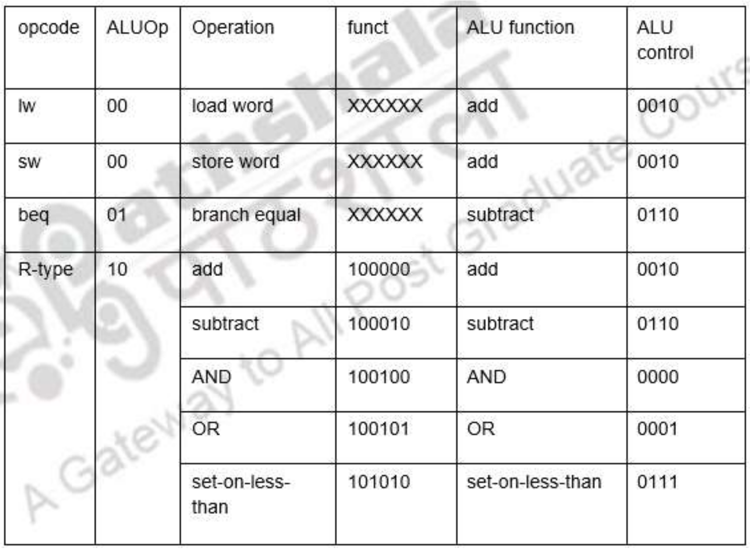
We shall first of all look at the ALU control. The implementation discussed here is specifically for the MIPS architecture and for the subset of instructions pointed out earlier. You just need to get the concepts from this discussion. The ALU uses 4 bits for control. Out of the 16 possible combinations, only 6 are used for the subset under consideration. This is indicated in Figure. Depending on the type of instruction class, the ALU will need to perform one of the first five functions. (NOR is needed for other parts of the MIPS instruction set not discussed here.) For the load word and store word instructions, we use the ALU to compute the memory address. This is done by addition. For the R-type instructions, the ALU needs to perform one of the five actions (AND, OR, subtract, add, or set on less than), depending on the value of the 6-bit funct (or function) field in the low-order bits of the instruction (refer to the instruction formats). For a branch on equal instruction, the ALU must perform a subtraction, for comparison.

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

We can generate the 4-bit ALU control input using a small control unit that takes as inputs the function field of the instruction and a 2-bit control field, which we call ALUOp. ALUOp indicates whether the operation to be performed should be add (00) for loads and stores, subtract (01) for beq, or determined by the operation encoded in the funct field (10). The output of the ALU control unit is a 4-bit signal that directly controls the ALU by generating one of the 4-bit combinations shown previously. In Figure 9.3, we show how to set the ALU control inputs based on the 2-bit ALUOp control and the 6-bit function code. The opcode, listed in the first column, determines the setting of the ALUOp bits. When the ALUOp code is 00 or 01, the desired ALU action does not depend on the function code field and this is indicated as don't cares, and the funct field is shown as XXXXXX. When the ALUOp value is 10, then the function code is used to set the ALU control input.

For completeness, the relationship between the ALUOp bits and the instruction opcode is also shown. Later on we will see how the ALUOp bits are generated from the main control unit. This style of using multiple levels of decoding—that is, the main control unit generates the ALUOp bits, which then are used as input to the ALU control that generates the actual signals to control the ALU unit—is a common implementation technique. Using multiple levels of control can reduce the size of the main control unit. Using several smaller control units may also potentially increase the speed of the control unit. Such optimizations are important, since the control unit is often performance-critical.

There are several different ways to implement the mapping from the 2-bit ALUOp field and the 6-bit function field to the three ALU operation control bits. Because only a small number of the 64 possible values of the function field are of interest and the function field is used only when the ALUOp bits equal 10, we can use a small piece of logic that recognizes the subset of possible values and causes the correct setting of the ALU control bits.



opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

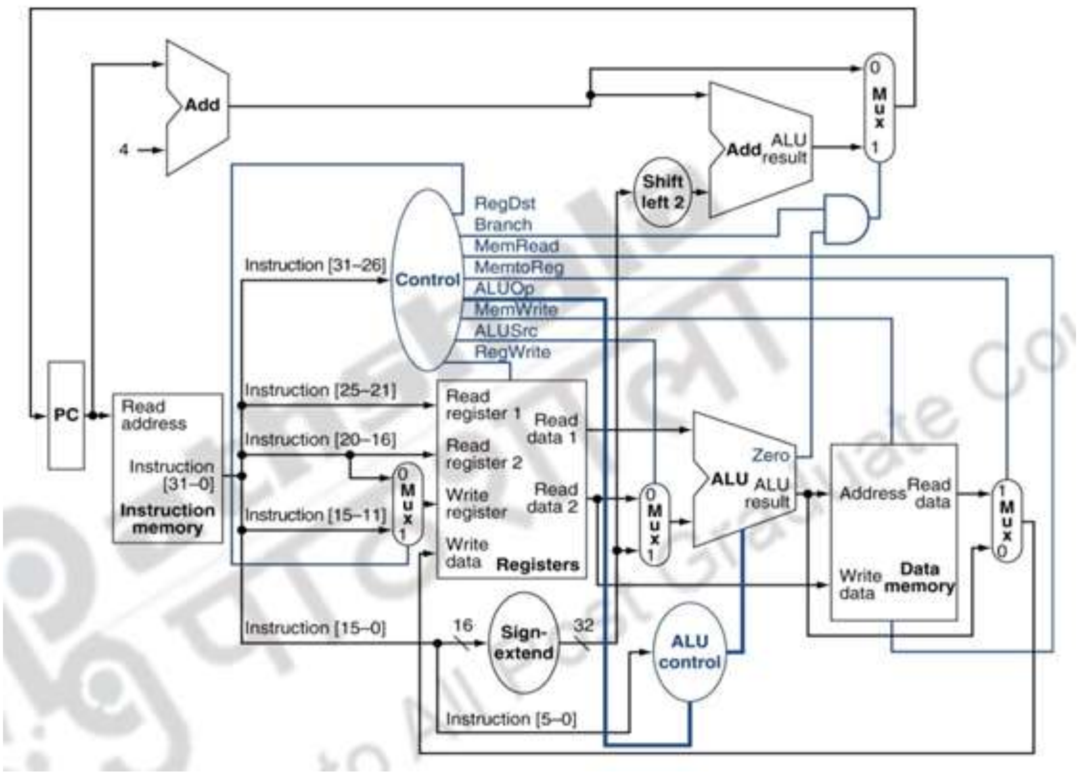
Now, we shall consider the design of the main control unit. For this, we need to remember the following details about the instruction formats of the MIPS ISA. All these details are indicated in Figure 9.4.

- For all the formats, the opcode field is always contained in bits 31:26 – Op[5:0]
- The two registers to be read are always specified by the Rs and Rt fields, at positions 25:21 and 20:16. This is true for the R-type instructions, branch on equal, and for store
- The base register for the load and store instructions is always in bit positions 25:21 (Rs)
- The destination register is in one of two places. For a load it is in bit positions 20:16 (Rt), while for an R-type instruction it is in bit positions 15:11 (Rd). To select one of these two registers, a multiplexor is needed.
- The 16-bit offset for branch equal, load, and store is always in positions 15:0.

To the simple datapath already shown, we shall add all the required control signals. Figure 9.5 shows these additions plus the ALU control block, the write signals for state elements, the read signal for the data memory, and the control signals for the multiplexors. Since all the multiplexors have two inputs, they each require a single control line. There are seven single-bit control lines plus the 2-bit ALUOp control signal. The seven control signals are listed below:

1. RegDst: The control signal to decide the destination register for the register write operation – The register in the Rt field or Rd field
2. RegWrite: The control signal for writing into the register file
3. ALUSrc: The control signal to decide the ALU source – Register operand or sign extended operand
4. PCSrc: The control signal that decides whether PC+4 or the target address is to written into the PC
5. MemWrite: The control signal which enables a write into the data memory
6. MemRead: The control signal which enables a read from the data memory
7. MemtoReg: The control signal which decides what is written into the register file, the result of the ALU operation or the data memory contents.

The datapath along with the control signals included is shown in Figure 9.5. Note that the control unit takes in the opcode information from the fetched instruction and generates all the control signals, depending on the operation to be performed.

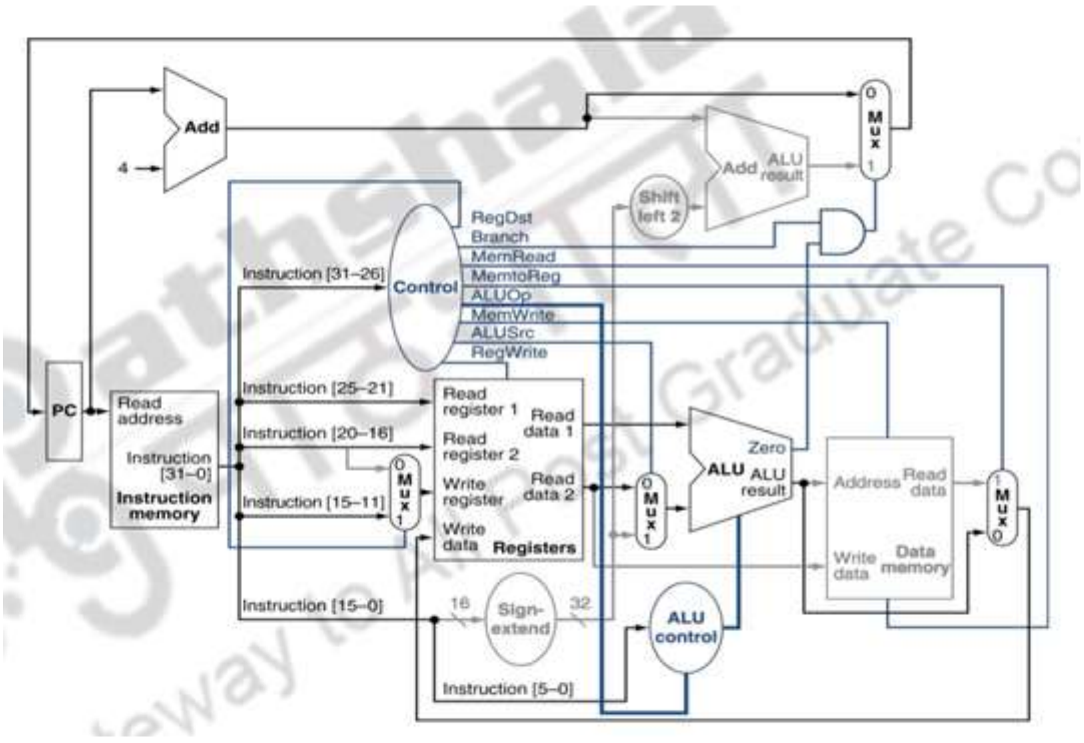


Now, we shall trace the execution flow for different types of instructions and see what control signals have to be activated. Let us consider the execution of an R type instruction first. For all these instructions, the source register fields are Rs and Rt, and the destination register field is Rd. The various operations that take place for an arithmetic / logical operation with register operands are:

- The instruction is fetched from the code memory
- Since the Branch control signal is set to 0, the PC is unconditionally replaced with PC + 4
- The two registers specified in the instruction are read from the register file
- The ALU operates on the data read from the register file, using the function code (bits 5:0, which is the funct field, of the instruction) to generate the ALU function
- The ALUSrc control signal is deasserted, indicating that the second operand comes from a register

- The ALUOp field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct field
- The result from the ALU is written into the register file using bits 15:11 of the instruction to select the destination register
- The RegWrite control signal is asserted and the RegDst control signal is made 1, indicating that Rd is the destination register
- The MemtoReg control signal is made 0, indicating that the value fed to the register write data input comes from the ALU

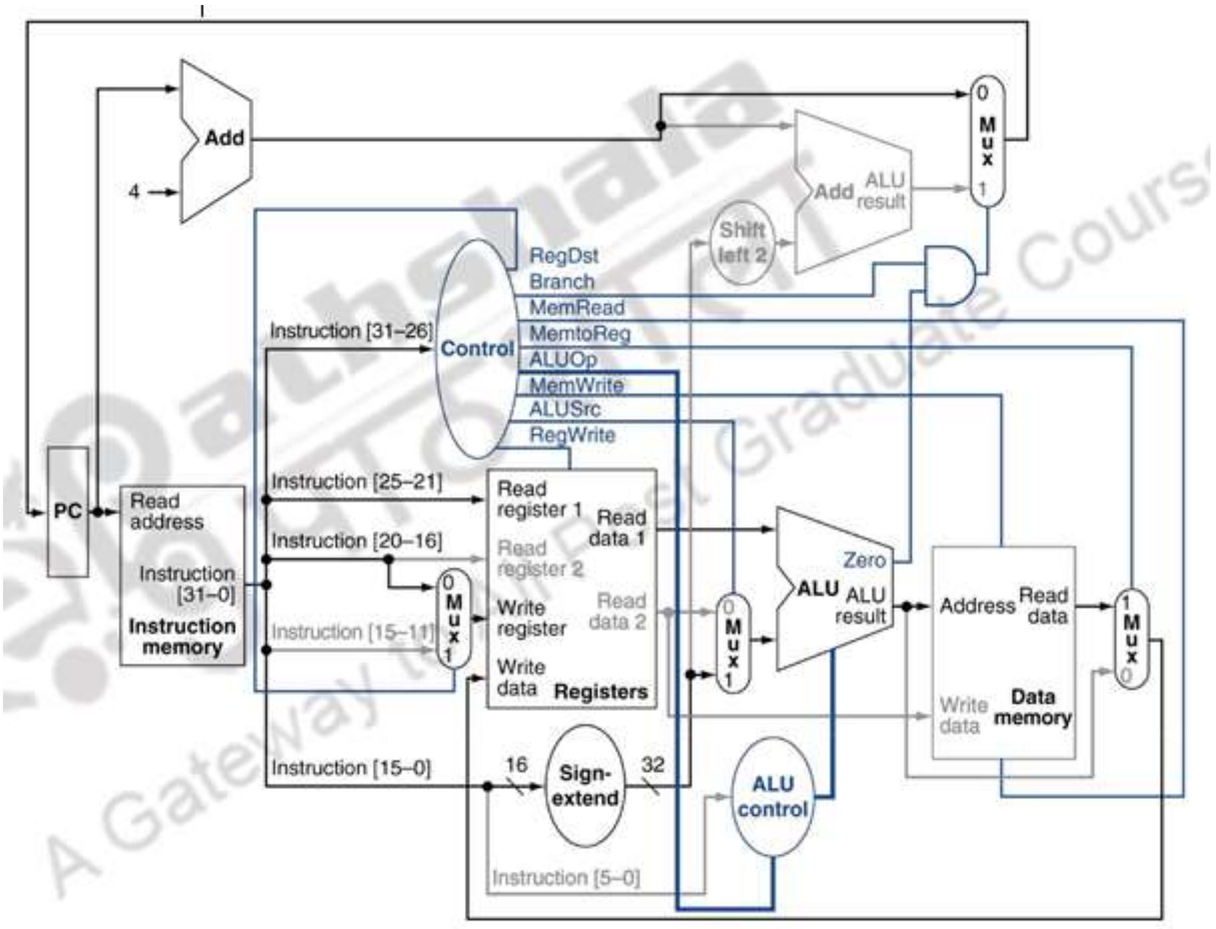
Furthermore, an R-type instruction writes a register (RegWrite = 1), but neither reads nor writes data memory. So, the MemRead and MemWrite control signals are set to 0. These operations along with the required control signals are indicated in Figure 9.6.



Similarly, we can illustrate the execution of a load word, such as lw $t1, offset($t2). Figure 9.7 shows the active functional units and asserted control lines for a load. We can think of a load instruction as operating in five steps:

- The instruction is fetched from the code memory
- Since the Branch control signal is set to 0, the PC is unconditionally replaced with PC + 4
- A register ($t2) value is read from the register file
- The ALU computes the sum of the value read from the register file and the sign-extended, lower 16 bits of the instruction (offset)
- The ALUSrc control signal is asserted, indicating that the second operand comes from the sign extended operand
- The ALUOp field for R-type instructions is set to 00 to indicate that the ALU should perform addition for the address calculation
- The sum from the ALU is used as the address for the data memory and a data memory read is performed
- The MemRead is asserted and the MemWrite control signals is set to 0

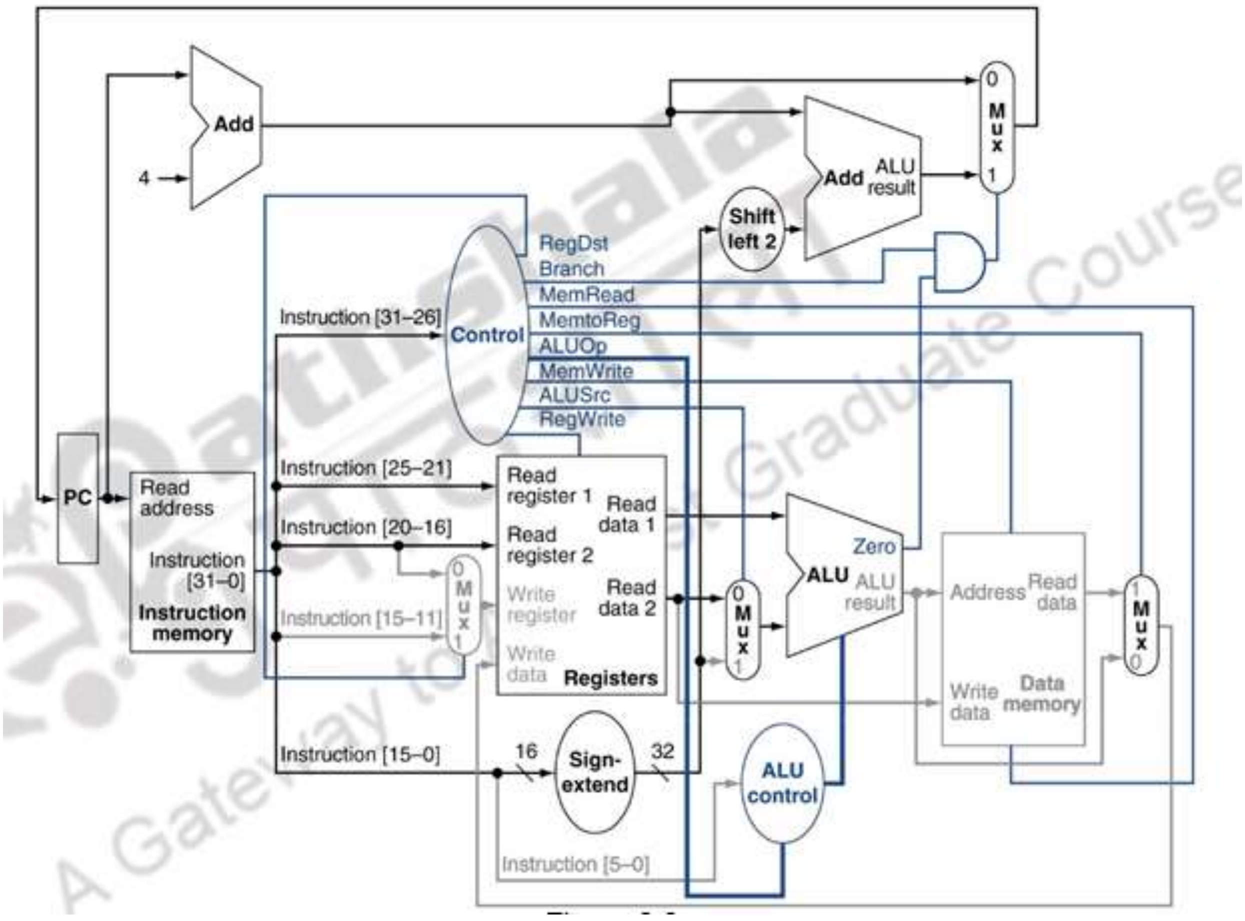
- The result from the ALU is written into the register file using bits 20:16 of the instruction to select the destination register
- The RegWrite control signal is asserted and the RegDst control signal is made 0, indicating that Rt is the destination register
- The MemtoReg control signal is made 1, indicating that the value fed to the register write data input comes from the data memory



A store instruction is similar to the load for the address calculation. It finishes in four steps The control signals that are different from load are:

- MemWrite is 1 and MemRead is 0
- RegWrite is 0
- MemtoReg and RegDst are X's (don't cares)

The branch instruction is similar to an R-format operation, since it sends the Rs and Rt registers to the ALU. The ALUOp field for branch is set for a subtract (ALU control = 01), which is used to test for equality. The MemtoReg field is irrelevant when the RegWrite signal is 0. Since the register is not being written, the value of the data on the register data write port is not used. The Branch control signal is set to 1. The ALU performs a subtract on the data values read from the register file. The value of PC + 4 is added to the sign-extended, lower 16 bits of the instruction (offset) shifted left by two; the result is the branch target address. The Zero result from the ALU is used to decide which adder result to store into the PC. The control signals and the data flow for the Branch instruction is shown in Figure 9.8.

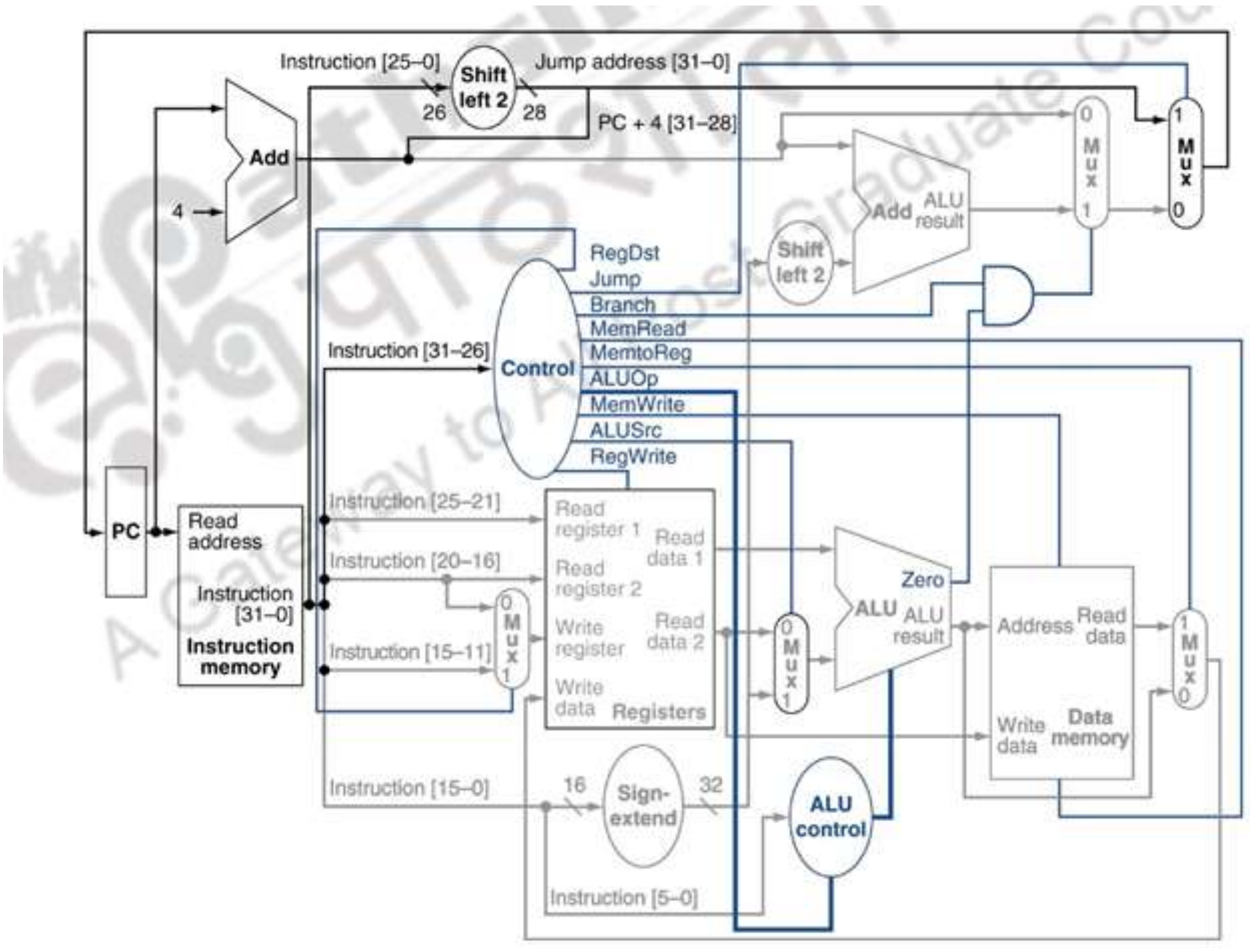


Now, to the subset of instructions already discussed, we shall add a jump instruction. The jump instruction looks somewhat similar to a branch instruction but computes the target PC differently and is not conditional. Like a branch, the low order 2 bits of a jump address are always 00. The next lower 26 bits of this 32-bit address come from the 26-bit immediate field in the instruction, as shown in Figure 9.9. The upper 4 bits of the address that should replace the PC come from the PC of the jump instruction plus 4. Thus, we can implement a jump by storing into the PC the concatenation of the upper 4 bits of the current PC + 4 (these are bits 31:28 of the sequentially following instruction address), the 26-bit immediate field of the jump instruction and the bits 00.

2	address
31:26	25:0

Figure shows the addition of the control for jump added to the previously discussed control.

An additional multiplexor is used to select the source for the new PC value, which is either the incremented PC (PC + 4), the branch target PC, or the jump target PC. One additional control signal is needed for the additional multiplexor. This control signal, called *Jump*, is asserted only when the instruction is a jump—that is, when the opcode is 2.



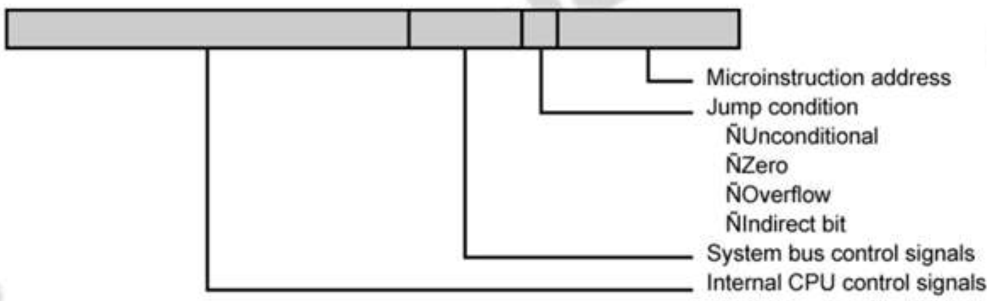
Since we have assumed that all the instructions get executed in one clock cycle, the longest instruction determines the clock period. For the subset of instructions considered, the critical path is that of the load, which takes the following path Instruction memory ® register file ® ALU ® data memory ® register file

The single cycle implementation may be acceptable for this simple instruction set, but it is not feasible to vary the period for different instructions, for eg. Floating point operations. Also, since the clock cycle is equal to the worst case delay, there is no point in improving the common case, which violates the design principle of making the common case fast. In addition, in this single-cycle implementation, each functional unit can be used only once per clock. Therefore, some functional units must be duplicated, raising the cost of the implementation. A single-cycle design is inefficient both in its performance and in its hardware cost. These shortcomings can be avoided by using implementation techniques that have a shorter clock cycle—derived from the basic functional unit delays—and that require multiple clock cycles for each instruction. In the next module, we will look at another implementation technique, called pipelining, that uses a datapath very similar to the single-cycle datapath, but is much more efficient.

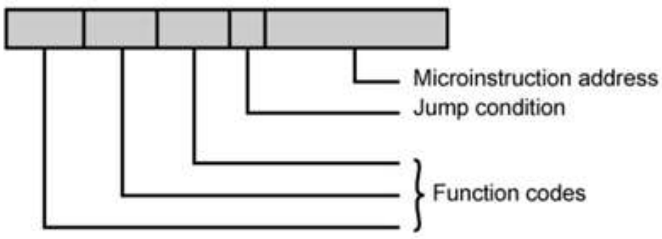
Next, we shall briefly discuss another type of control, viz. microprogrammed control. In the case of hardwired control, we saw how all the control signals required inside the CPU can be generated using hardware. There is an alternative approach by which the control signals required inside the CPU can be generated. This alternative approach is known as microprogrammed control unit. In microprogrammed control unit, the logic of the control unit is specified by a microprogram. A microprogram consists of a sequence of instructions in a microprogramming language. These are instructions that specify microoperations. A microprogrammed control unit is a relatively simple logic circuit that is capable of (1) sequencing through microinstructions and (2) generating control signals to execute each microinstruction.

The concept of microprogram is similar to computer program. In computer program the complete instructions of the program is stored in main memory and during execution it fetches the instructions from main memory one after another. The sequence of instruction fetch is controlled by the program counter (PC). Microprograms are stored in microprogram memory and the execution is controlled by the microprogram counter (PC). Microprograms consist of microinstructions which are nothing but strings of 0's and 1's. In a particular instance, we read the contents of one location of microprogram memory, which is nothing but a microinstruction. Each output line (data line) of microprogram memory corresponds to one control signal. If the contents of the memory cell is 0, it indicates that the signal is not generated and if the contents of memory cell is 1, it indicates the generation of the control signal at that instant of time.

There are basically two types of microprogrammed control – horizontal organization and vertical organization. In the case of horizontal organization, as mentioned above, you can assume that every bit in the control word corresponds to a control signal. In the case of a vertical organization, the signals are grouped and encoded in order to reduce the size of the control word. Normally some minimal level of encoding will be done even in the case of horizontal control. The fields will remain encoded in the control memory and they must be decoded to get the individual control signals. Horizontal organization has more control over the potential parallelism of operations in the datapath; however, it uses up lots of control store. Vertical organization, on the other hand, is easier to program, not very different from programming a RISC machine in assembly language, but needs extra level of decoding and may slow the machine down. Figure 9.11 shows the two formats.



(a) Horizontal microinstruction



(b) Vertical microinstruction

The different terminologies related to microprogrammed control unit are:

Control Word (CW): Control word is defined as a word whose individual bits represent the various control signal. Therefore, each of the control steps in the control sequence of an instruction defines a unique combination of 0s and 1s in the CW. A sequence of control words (CWs) corresponding to the control sequence of a machine instruction constitute the microprogram for that instruction. The individual control words in this microprogram are referred to as microinstructions. The microprograms corresponding to the instruction set of a computer are stored in a special memory that will be referred to as the microprogram memory or control store. The control words related to all instructions are stored in the microprogram memory.

The control unit can generate the control signals for any instruction by sequencially reading the CWs of the corresponding microprogram from the microprogram memory. To read the control word sequentially from the microprogram memory a microprogram counter (PC) is needed. The basic organization of a microprogrammed control unit is shown in the Figure 3.7. The starting address generator block is responsible for loading the starting address of the microprogram into the PC every time a new instruction is loaded in the IR. The PC is then automatically incremented by the clock, and it reads the successive microinstruction from memory. Each microinstruction basically provides the required control signal at that time step. The microprogram counter ensures that the control signal will be delivered to the various parts of the CPU in correct sequence.

We have some instructions whose execution depends on the status of condition codes and status flag, as for example, the branch instruction. During branch instruction execution, it is required to take the decision between alternative actions. To handle such type of instructions with microprogrammed control, the design of the control unit is based on the concept of conditional branching in the microprogram. In order to do that, it is required to include some conditional branch microinstructions. In conditional microinstructions, it is required to specify the address of the microprogram memory to which the control must be directed to. It is known as the branch address. Apart from the branch address, these microinstructions can specify which of the states flags, condition codes, or possibly, bits of the instruction register should be checked as a condition for branching to take place.

In a computer program we have seen that execution of every instruction consists of two parts – fetch phase and execution phase of the instruction. It is also observed that the fetch phase of all instruction is the same. In a microprogrammed control unit, a common microprogram is used to fetch the instruction. This microprogram is stored in a specific location and execution of each instruction starts from that memory location. At the end of the fetch microprogram, the starting address generator unit calculates the appropriate starting address of the microprogram for the instruction which is currently present in IR. After that the PC controls the execution of microprogram which generates the appropriate control signals in the proper sequence. During the execution of a microprogram, the PC is incremented everytime a new microinstruction is fetched from the microprogram memory, except in the following situations :

1. When an End instruction is encountered, the PC is loaded with the address of the first CW in the microprogram for the next instruction fetch cycle.

2. When a new instruction is loaded into the IR, the PC is loaded with the starting address of the microprogram for that instruction.

3. When a branch microinstruction is encountered, and the branch condition is satisfied, the PC is loaded with the branch address.

The organization of a microprogrammed control unit is given in Figure 9.12.

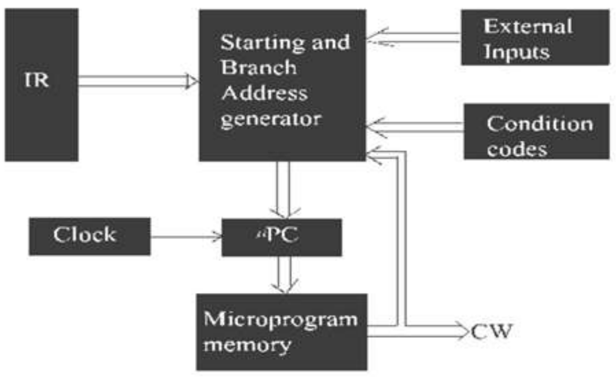


Figure 9.12 Organization of microprogrammed control

Hardwire and microprogrammed control: microprogramme sequencing

In a system or computer, most of the tasks are controlled with the help of a processor or CPU (Central processing unit), which is the main component of a computer. The CPU usually has two main systems: **control unit** (CU) and **arithmetic and logic unit** (ALU). The control unit (CU) is used to synchronize the tasks with the help of sending timings and control signals. On the other hand, mathematical and logical operations can be handled with the help of ALU. Micro programmed control units and hardwired control units can be called two types of control units. We can execute an instruction with the help of these two control units.

In the **hardwired control unit**, the execution of operations is much faster, but the implementation, modification, and decoding are difficult. In contrast, implementing, modifying, decoding **micro-programmed control units** is very easy. The micro-programmed control unit is also able to handle complex instructions. With the help of control signals generated by micro-programmed and hardwired control units, we are able to fetch and execute the instructions.

Control Signals

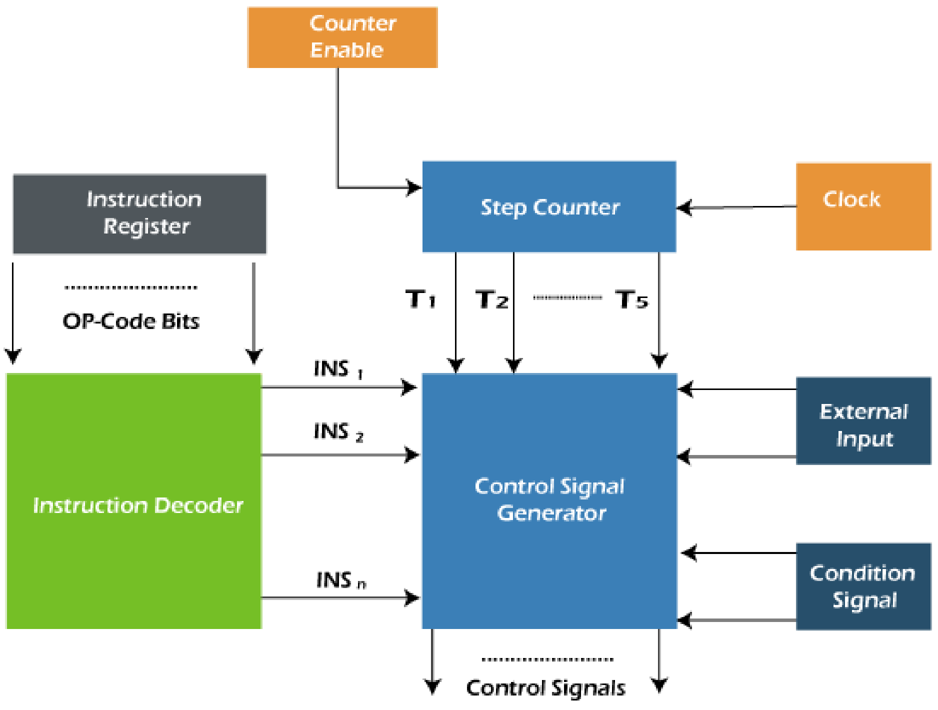
In order to generate the control signals, both the control signals were basically designed. The functionality of a processor's hardware is operated with the help of these control signals. The control signals are used to know about various types of things, which are described as follows:

- Control signals are used to know what operation is going to be performed.
- It is used to know about the sequence of operations that are performed by the processor.
- It is used to know about the timing at which an operation must be executed and many other types of things.

Hardwired Control Unit

With the help of generating control signals, the hardwired control unit is able to execute the instructions at a correct time and proper sequence. As compared to the micro-programmed, the hardwired CU is generally faster. In this CU, the control signals are generated with the help of PLA circuit and state counter. Here the Central processing unit requires all these control signals. With the help of hardware, the hardwired control signals are generated, and it basically uses the circuitry approach.

The image of a hardwired control unit is described as follows, which contains various components in the form of circuitry. We will discuss them one by one so that we can properly understand the "generation of control signals".



Hardwired Control Unit

- The **instruction register** is a type of processor register used to contain an instruction that is currently in execution. As we can see, the instruction register is used to generate the OP-code bits respective of the operation as well as the addressing mode of operands.
- The above generated Op-code bits are received in the field of an **instruction decoder**. The instruction decoder interprets the operation and instruction's addressing mode. Now on the basis of the addressing mode of instruction and operation which exists in the instruction register, the instruction decoder sets the corresponding Instruction signal INS_i to 1. Some steps are used to execute each instruction, i.e., **instruction fetch**, **decode**, **operand fetch**, **Arithmetic and logical unit**, and **memory store**. Different books might be contained different steps. But in general, we are able to execute an instruction with the help of these five steps.
- The information about the current step of instruction must be known by the control unit. Now the **Step Counter** is implemented, which is used to contain the signals from T1,…., T5. Now on the basis of the step which contains the instruction, one of the signals of a step counter will be set from T1 to T5 to 1.
- Now we have a question that how the step counter knows about the current step of instruction? So to know the current step, a **Clock** is implemented. The one-clock cycle of the clock will be completed for each step. For example, suppose that if the stop counter sets T3 to 1, then after completing one clock cycle, the step counter will set T4 to 1.
- Now we have a question, i.e., what will happen if the execution of an instruction is interrupted for some reason? Will the step counter still be triggered by the clock? The answer to this question is **No**. As long as the execution is current step is completed, the **Counter Enable** will "disable" the Step Counter so that it will stop then increment to the next step signal.
- Now we have a question, i.e., what if the execution of instruction depends on some conditions? In this case, the **Condition Signals** will be used. There are various conditions in which the signals are generated with the help of control signals that can be less than, greater than, less than equal, greater than equal, and many more.
- The **external input** is the last one. It is used to tell the Control Signal Generator about the interrupts, which will affect the execution of an instruction.

So, on the basis of the input obtained by the conditional signals, step counter, external inputs, and instruction register, the control signals will be generated with the help of Control signal Generator.

Advantages of Hardwired Control Unit:

1. Speed is high

Disadvantages of Hardwired Control Unit:

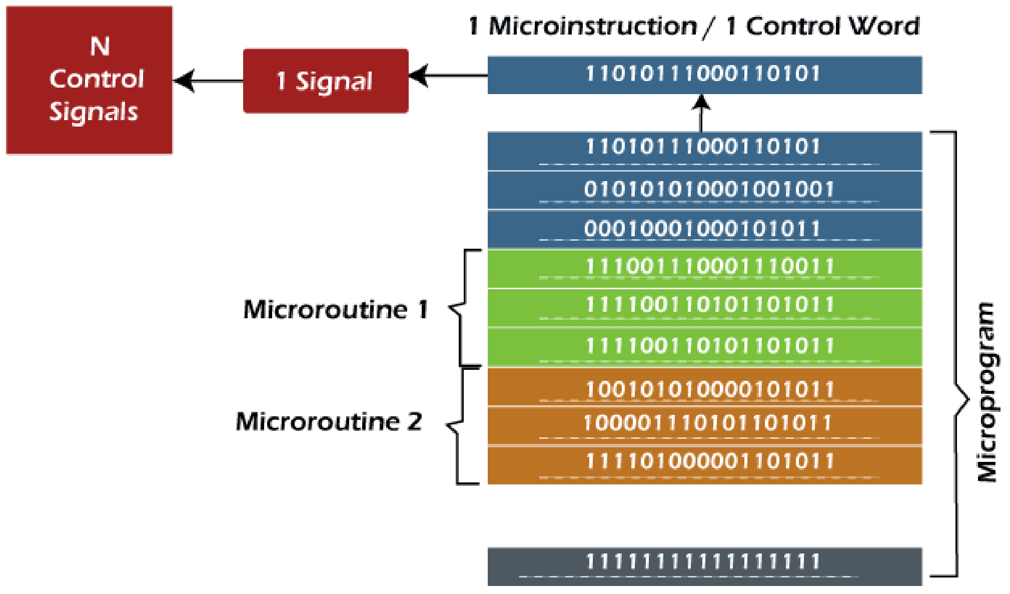
1. Instruction set, the **control** logic is directly implemented.
2. Requires change in wiring if the design has to be controlled.
3. An occurrence of an error is more.
4. Complex decoding and sequencing logic.

Micro-programmed Control Unit

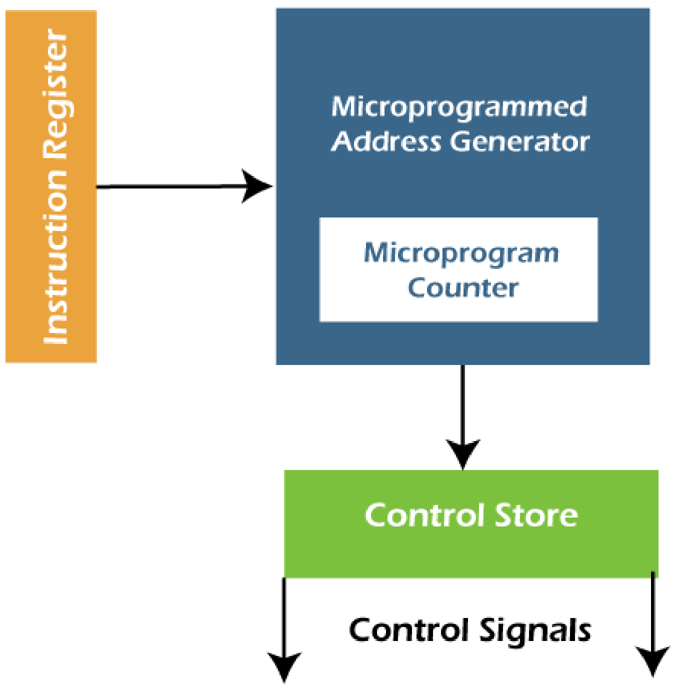
A micro-programmed control unit can be described as a simple logic circuit. We can use it in two ways, i.e., it is able to execute each instruction with the help of generating control signals, and it is also able to do sequencing through microinstructions. It will generate the control signals with the help of programs. At the time of evolution of CISC architecture in the past, this approach was very famous. The program which is used to create the control signals is known as the "Micro-program". The micro-program is placed on the processor chip, which is a type of fast memory. This memory is also known as the control store or control memory.

A micro-program is used to contain a set of microinstructions. Each microinstruction or control word contains different bit patterns. The n bit words are contained by each microinstruction. On the basis of the bit pattern of a control word, every control signals differ from each other.

Like the above, the instruction execution in a micro-programmed control unit is also performed in steps. So for each step, the micro-program contains a control word/ microinstruction. If we want to execute a particular instruction, we need a sequence of microinstructions. This process is known as the micro-routine. The image of a micro-programmed control unit is described as follows. Here, we will learn the organization of micro-program, micro-routine, and control word/ microinstruction.



Now we will learn about the organization of Micro-program CU. Then we will learn about the flow of instruction execution with the help of instruction execution steps, which are described as follows:



Microprogrammed Control Unit Organization

- **Instruction** fetch is the **first step**. In this step, the instruction is fetched from the IR (Instruction Register) with the help of a Microinstruction address register.
- **Decode** is the **second step**. In this step, the instructions obtained from the instruction register will be decoded with the help of a microinstruction address generator. Here we will also get the starting address of a micro-routine. With the help of this address, we can easily perform the operation, which is mentioned in the instruction. It will also load the starting address into the micro-program counter.
- **Increment** is the third step. In this step, the control word, which corresponds to the starting address of a micro-program, will be read. When the execution proceeds, the value of the micro-program counter will be increased so that it can read the successive control words of a micro-routine.
- **End bit** is the fourth step. In this step, the microinstruction of a micro-routine contains a bit, which is known as the end bit. The execution of the microinstruction will be successfully completed when the end bit is set to 1.
- This is the last step, and in this step, the micro-program address generator will again go back to **Step 1** so that we can fetch a new instruction, and this process or cycle goes on.

So in the micro-programmed control unit, the micro-programs are stored with the help of Control memory or Control store. The implementation of this CU is very easy and flexible, but it is slower as compared to the Hardwired control unit.

Advantages of Microprogrammed Control Unit:

1. The decoders and sequencing logic unit of a micro-programmed control unit are very simple pieces of logic, compared to the hardwired control unit, which contains complex logic for sequencing through the many micro-operations of the instruction cycle.
2. It Simplifies the design of the control unit.

Disadvantages of Microprogrammed Control Unit:

1. This is slower than the hardwired control unit because the microinstructions are to be fetched from the control memory which is time-consuming.

Differences between Hardwired Control unit and Micro-programmed Control unit

There are various differences between Micro-programmed CU and Hardwired CU, which are described as follows:

Hardwired Control Unit	Micro-programmed Control Unit
With the help of a hardware circuit, we can implement the hardwired control unit. In other words, we can say that it is a circuitry approach.	While with the help of programming, we can implement the micro-programmed control unit.
The hardwired control unit uses the logic circuit so that it can generate the control signals, which are required for the processor.	The micro-programmed CU uses microinstruction so that it can generate the control signals. Usually, control memory is used to store these microinstructions.
In this CU, the control signals are going to be generated in the form of hard wired. That's why it is very difficult to modify the hardwired control unit.	It is very easy to modify the micro-programmed control unit because the modifications are going to be performed only at the instruction level.
In the form of logic gates, everything has to be realized in the hardwired control unit. That's why this CU is more costly as compared to the micro-programmed control unit.	The micro-programmed control unit is less costly as compared to the hardwired CU because this control unit only requires the microinstruction to generate the control signals.
The complex instructions cannot be handled by a hardwired control unit because when we design a circuit for this instruction, it will become complex.	The micro-programmed control unit is able to handle the complex instructions.
Because of the hardware implementation, the hardwired control unit is able to use a limited number of instructions.	The micro-programmed control unit is able to generate control signals for many instructions.
The hardwired control unit is used in those types of computers that also use the RISC (Reduced instruction Set Computers).	The micro-programmed control unit is used in those types of computers that also use the CISC (Complex instruction Set Computers).
In the hardwired control unit, the hardware is used to generate only the required control signals. That's why this control unit is faster as compared to the micro-programmed control unit.	In this CU, the microinstructions are used to generate the control signals. That's why this CU is slower than the hardwired control unit.

Some Other differences between Micro-programmed control unit and Hardwire control unit

Now we will describe these differences on the basis of some parameters, such as speed, cost, modification, instruction decoder, control memory, etc. These differences are described as follows:

Speed

In the hardwired control unit, the speed of operations is very fast. In contrast, the micro-programmed control unit needs frequent memory access. So the speed of operation of a micro-programmed control unit is slow.

Modification

If we want to do some modifications to the Hardwired control unit, we have to redesign the entire unit. In contrast, if we want to do some modification in the micro-programmed control unit, we can do that just by changing the microinstructions in the control memory. In this case, the more flexible control unit is a micro-programmed control unit.

Cost

The implementation of a Hardwire control unit is very much compared to the Micro-programmed control unit. In this case, the micro-programmed control unit will save our money at the time of implementation.

Handling Complex Instructions

If we try to handle the complex instructions with the help of a hardwired control unit, it will be very difficult for us to handle them. But if we try to handle the complex instructions with the help of micro-programmed control unit, it will be very easy for us to handle them. In this case also, the Micro-programmed control unit is better.

Instruction decoding

In the hardwired control unit, if we want to perform instruction decoding, it will be very difficult. But if we do the same thing in a micro-programmed control unit, it will be very easy for us.

Instruction set size

A small instruction set is used by the hardwired CU. On the other hand, a large instruction set is used by the micro-programmed control unit.

Control Memory

The hardwired control unit does not use the control memory to generate the control signals, but the micro-programmed CU needs to use the control memory to generate the control signals.

Applications

The hardwired control unit is used in those types of processors that basically use a simple instruction set. This set is called a Reduced Instruction Set Computer. On the other hand, a micro-programmed control unit is used in those types of processors that basically use a complex instruction set. This set is called a Complex Instruction Set Computer.

Microinstruction with next address field

Need for designing the micro-instruction sequencing technique:

The first purpose is to minimize the size of control memory because control memory is present inside the processor.

The second purpose is to execute the micro-instructions as fast as possible. Which means the address of the next micro-instruction can be calculated as fast as possible.

The factors which are responsible for reducing the size of control memory are –

- Degree of parallelism i.e. how many microoperations which can be performed simultaneously.
- Representation/encoding of control information.
- The way of specifying the address of next microinstruction.

The number of microoperations executed in the processor depends upon the processor architecture, and encoding of instructions makes it short. But the major concern is to calculate the address of the next micro-instruction.

The address of the next micro-instruction can be –

- The address of the next micro-instruction in the sequence i.e. one after the other.
- Branch address(which can be conditional or unconditional).

- Calculated on the basis of the opcode of the instruction.

  The address of the first micro-instruction can be calculated once from the opcode of the instruction which is present in the instruction register, then that address is loaded into CMAR (Control Memory Address Register). CMAR passes the address to the decoder. The decoder identifies the corresponding micro-instructions from the Control Memory.

  A micro-instruction has two fields: a control field and an address field.

- **Control field –**

  Determines which control signals are to be generated.

- **Address field –**

  Determines the address of the next micro-instruction.

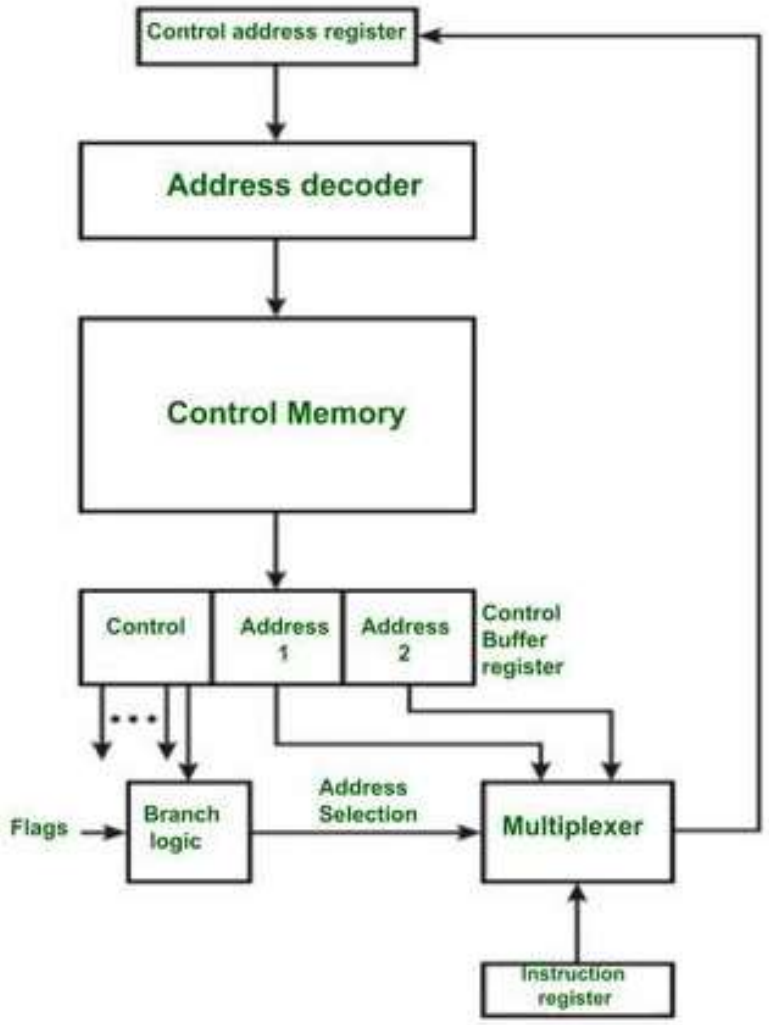
  This address is further loaded into CMAR to fetch the next micro-instruction.

  As we know, usually micro-instructions are not executed sequentially for a long time . Let's say after 4 or 5 micro-instructions the branch can usually occur. Therefore, our main motive is to make the branching algorithm better so that the address of the next micro-instruction can be calculated efficiently. Therefore, micro-instruction sequencing is the method of determining the flow of the microprogram.

  So there are techniques which are based on the number of addresses utilised for sequencing –

1. Two address fields in each microinstruction (Dual address field).
2. Single address field(Single address field).
3. Variable format microinstructions

1. Dual address field –



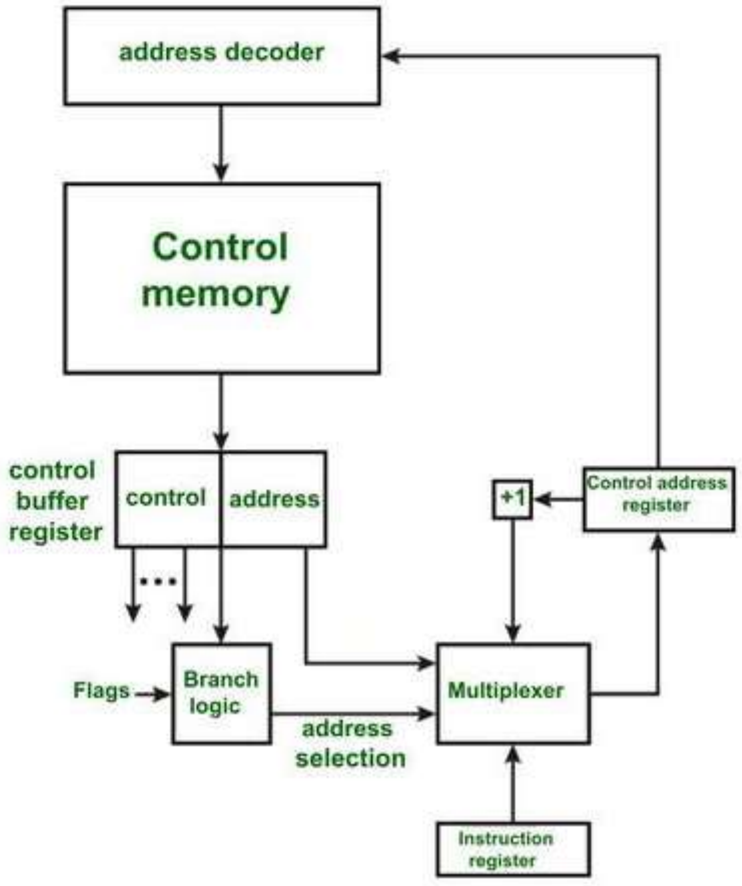
Dual address field

- In this approach, micro-instructions are not executed in a sequential manner.
- The instruction register (IR) gives the address of the first micro-instruction.
- Thereafter, each micro-instruction gives the address of the next micro-instruction.
- If it is a conditional micro-instruction, it will contain two address fields.
- One for the condition to be true and the other for false. Hence, it is called dual address field.
- The multiplexer will decide the address that will be loaded into the control memory address register (CMAR) based on the status flags.

Here, lots of control memory is wasted because at least one of the address fields is not required in many(i.e. for sequential or unconditional) micro-instructions.

2. Single address field –

With some modifications and the added logic, the number of addresses is reduced to one. Here, a new register called microprogram counter is used. In this case, the next microinstruction address can be the address of the next sequential address or it can be the address generated using op-code or it can be the address stored in the address field of the microinstruction.



Single address field.

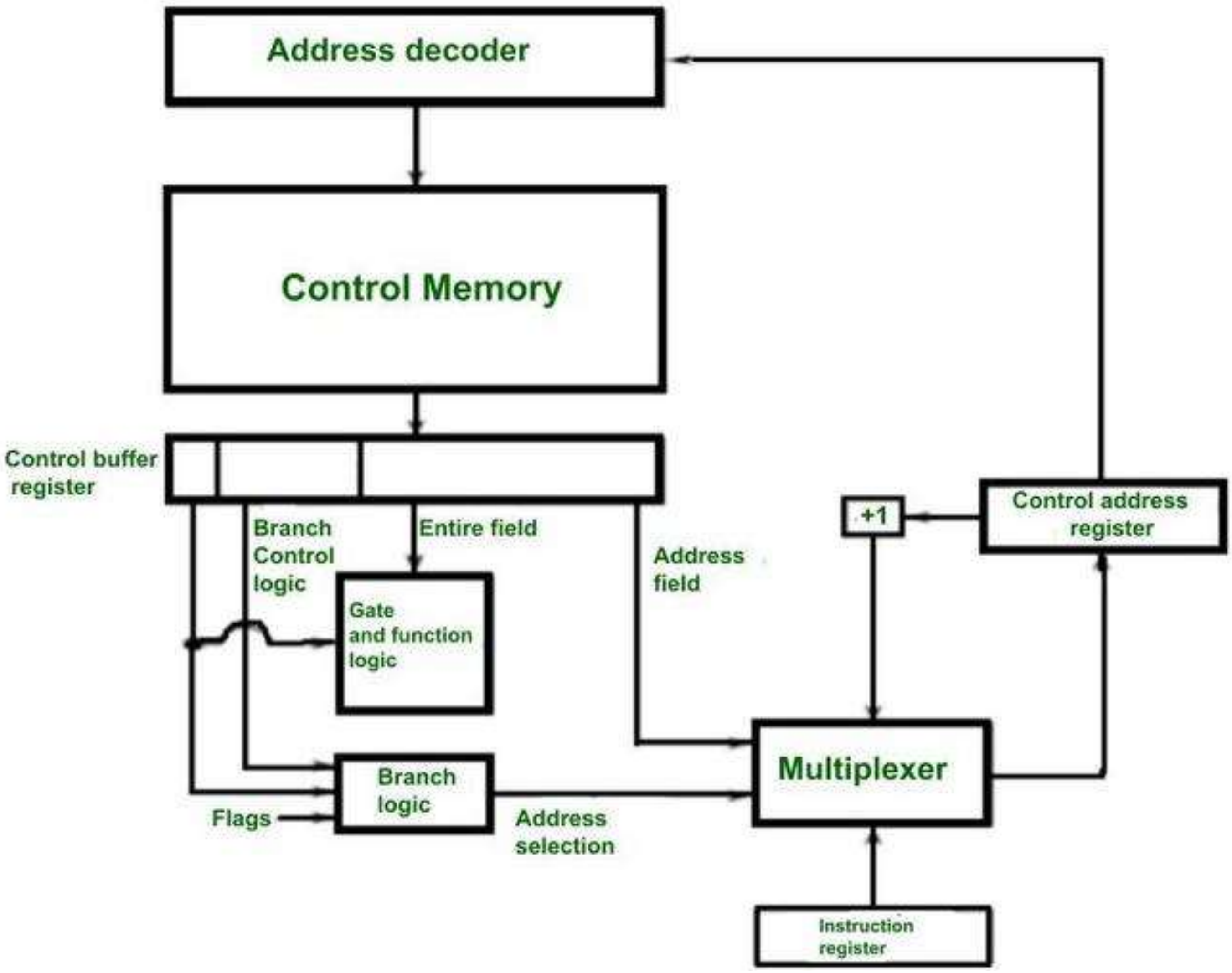
- In this approach, micro-instructions are executed in a sequential manner.
- The instruction register (IR) gives the address of the first micro-instruction into CMAR.
- Thereafter, the address is simply incremented.
- Hence, every micro-instruction need not carry the address of the next one.
- This is true so long as the micro-program is executed in a sequential manner.

- For an unconditional branch, the micro-instructions include the branch address. This address will be loaded into CMAR.
- For a conditional branch, the micro-instruction contains the branch address for true condition. If the condition is false, the current address in CMAR will be simply incremented.
- This means even in the worst case, the micro-instruction will carry only one address.
- Hence, it is called single address field.
- The multiplexer will decide the address that will be loaded into the control memory address register (CMAR) based on the status flags.

  This method is commonly used. But the space provided in each micro-instruction in a single address field is not quite useful if the instructions are executed sequentially.

  3. Variable address format –

- In this technique two formats are used. In such a technique, one bit is needed in the microinstruction to differentiate between control microinstruction or a branching microinstruction. The first format provides the control microinstruction(i.e. the bits are used to generate control signals) , while the second format provides the branch logic and address(there can be conditional or unconditional branch).
- In the first format, the microinstruction contains control signals, then the next microinstruction address is calculated either by using the op-code of the instruction register or it is the address of the next microinstruction in sequence. In this approach, an extra cycle is needed for branch microinstruction.



Variable instruction format

Pre-fetching microinstructions

Prefetching is the loading of a resource before it is required to decrease the time waiting for that resource. Examples include instruction prefetching where a CPU caches data and instruction blocks before they are executed, or a web browser requesting copies of commonly accessed web pages. Prefetching functions often make use of a cache. Prefetching allows applications and hardware to maximize performance and minimize wait times by preloading resources that users will need before they request them.

Web browsers employ prefetching by preloading commonly accessed pages. When the user navigates to the page, it loads quickly because the browser is pulling it from the cache. Some browser plugins download all of the pages that have been hyperlinked to attempt to speed up the browser.

The technique can be applied in several circumstances:

- Cache prefetching, a speedup technique used by computer processors where instructions or data are fetched before they are needed
- Prefetch input queue (PIQ), in computer architecture, pre-loading machine code from memory
- Link prefetching, a web mechanism for prefetching links
- Prefetcher technology in modern releases of Microsoft Windows
- prefetch instructions, for example provided by-
  - PREFETCH, an X86 instruction in computing
- Prefetch buffer, a feature of DDR SDRAM memory
- Swap prefetch, in computer operating systems, anticipatory paging

Concept of horizontal and vertical microprogramming

Horizontal Micro-programmed Control unit

With the help of decoded binary format, we can represent the control signals in the horizontal micro-programmed control unit, i.e., 1bit/CS. Here, n bit encoding is needed for 'n' control signals. With the help of a single control point, each bit is identified in the horizontal micro-programmed CU. This Control point is used to show that the corresponding micro-operation is going to be executed. In this control unit, every micro-program needs less number of micro-instructions. The several resources can be controlled simultaneously with the help of each and every micro-instruction. It also has a bigger advantage, i.e., it has the ability to utilize more efficient hardware.

A higher degree of parallelism is provided by the horizontal CU. This parallelism contains a separate control field and a minimum number of encoding. In the horizontal CU, the task to develop the micro-programs with the help of using resources efficiently and optimally is very complex. Each control bit in the horizontal micro-programmed control unit is independent to each other. That's why this CU provides great flexibility. The horizontal microinstruction contains more information as compared to the vertical microinstruction because horizontal microinstruction contains a greater length.

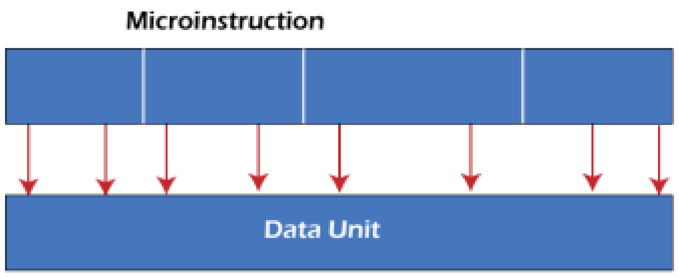


Fig: Horizontal Microcode

Vertical Micro-programmed CU

In contrast to the Horizontal micro-programmed CU, a higher degree of encoding and variable format can be applied in the vertical micro-programmed control unit. With the help of encoded binary format, we can represent the control signals in the vertical micro-programmed CU. Here, log2n bit encoding is needed for 'n' control signals. A single micro-operation is represented by every vertical micro-instruction. With the help of vertical CU, we can shorten the length of microinstruction as well as prevent the length of microinstruction from being directly affected by the increasing memory capacity.

The microinstruction is performed with the help of a code, and this code will be translated into the individual control signals with the help of a decoder. Because here we only specified the micro-operation that will be performed and the fields of microinstruction are fully utilized. There are basically 4 to 6 fields, and these fields approximately require 16 to 32 bits per instruction. As compared to the horizontal micro-programmed, we can easily write the vertical micro-programmed. Same as the conventional machine language format, the vertical microinstruction also contains a few operands and one operation. So we can easily use the vertical microinstruction for micro-programming.

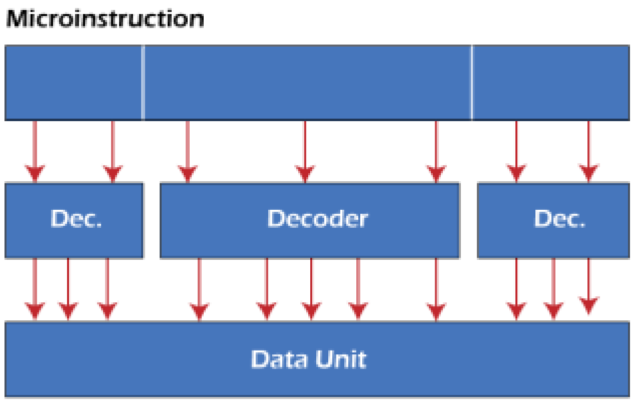


Fig: Vertical Microcode

Differences between Horizontal and Vertical Micro-programmed CU

There are various differences between the vertical programmed CU and horizontal programmed CU, which are described as follows:

Horizontal Micro-programmed CU	Vertical Micro-programmed CU
This control unit is able to support **longer control words.**	This control unit is able to support **shorter control words.**
In this CU, we don't need any type of additional hardware.	In this CU, we need additional hardware to generate the control signals. These types of hardware must be in the form of decoders.
Compared to the vertical micro-programmed control unit, this control unit **is less flexible.**	Compared to the horizontal micro-programmed control unit, this control unit is **more flexible.**
Compared to the vertical micro-programmed control unit, this control unit is **faster.**	Compared to the horizontal micro-programmed control unit, this control unit is **slower.**

Compared to the vertical micro-programmed control unit, this control unit makes **less use** of **ROM encoding.**	Compared to the horizontal micro-programmed control unit, this control unit makes **more use** of **ROM encoding** so that it can reduce the control world's length.
A higher degree of parallelism is allowed by the horizontal micro-programmed CU. If there are is 'n' number of degrees, n control signals will be enabled at a time.	The low degree of parallelism is allowed by the vertical micro-programmed CU. That means there can either be 0 or 1 degree of parallelism.
The horizontal microinstruction is used by the horizontal micro-programmed CU. Here control line is attacked with every bit of the control field.	The vertical microinstruction is used by the vertical micro-programmed CU. Here each action will be performed with the help of a code, and this code will be translated into the individual control signals with the help of a decoder.

Memory Hierarchy

Concept of memory:

Memory is internal storage areas in the computer system. The term memory identifies data storage that comes in the form of chips, and the word storage is used for memory that exists on tapes or disks. It is used to store data and instructions. Computer memory is the storage space in the computer, where data is to be processed and instructions required for processing are stored. The memory is divided into large number of small parts called cells.

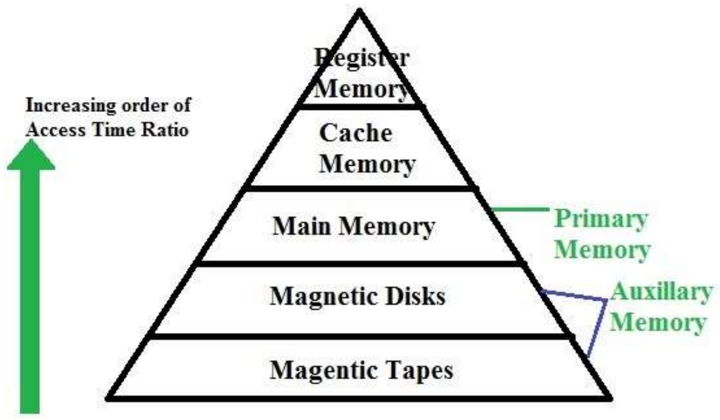
Memory Hierarchy

The memory in a computer can be divided into five hierarchies based on the speed as well as use. The processor can move from one level to another based on its requirements. The five hierarchies in the memory are registers, cache, main memory, magnetic discs, and magnetic tapes. The first three hierarchies are volatile memories which mean when there is no power, and then automatically they lose their stored data. Whereas the last two hierarchies are not volatile which means they store the data permanently.

A memory element is the set of storage devices which stores the binary data in the type of bits. In general, the storage of memory can be classified into two categories such as volatile as well as non- volatile.

Memory Hierarchy in Computer Architecture

The **memory hierarchy design** in a computer system mainly includes different storage devices. Most of the computers were inbuilt with extra storage to run more powerfully beyond the main memory capacity. The following **memory hierarchy diagram** is a hierarchical pyramid for computer memory. The designing of the memory hierarchy is divided into two types such as primary (Internal) memory and secondary (External) memory.



Primary Memory

The primary memory is also known as internal memory, and this is accessible by the processor straightly. This memory includes main, cache, as well as CPU registers.

Secondary Memory

The secondary memory is also known as external memory, and this is accessible by the processor through an input/output module. This memory includes an optical disk, magnetic disk, and magnetic tape.

Characteristics of Memory Hierarchy

The memory hierarchy characteristics mainly include the following.

Performance

Previously, the designing of a computer system was done without memory hierarchy, and the speed gap among the main memory as well as the CPU registers enhances because of the huge disparity in access time, which will cause the lower performance of the system. So, the enhancement was mandatory. The enhancement of this was designed in the memory hierarchy model due to the system's performance increase.

Ability

The ability of the memory hierarchy is the total amount of data the memory can store. Because whenever we shift from top to bottom inside the memory hierarchy, then the capacity will increase.

Access Time

The access time in the memory hierarchy is the interval of the time among the data availability as well as request to read or write. Because whenever we shift from top to bottom inside the memory hierarchy, then the access time will increase

Cost per bit

When we shift from bottom to top inside the memory hierarchy, then the cost for each bit will increase which means an internal Memory is expensive compared with external memory.

Memory Hierarchy Design

The memory hierarchy in computers mainly includes the following.

Registers

Usually, the register is a static RAM or SRAM in the processor of the computer which is used for holding the data word which is typically 64 or 128 bits. The program counter register is the most important as well as found in all the processors. Most of the processors use a status word register as well as an accumulator. A status word register is used for decision making, and the accumulator is used to store the data like mathematical operation. Usually, computers like **complex instruction set computers** have so many registers for accepting main memory, and **RISC- reduced instruction set** computers have more registers.

Cache Memory

Cache memory can also be found in the processor, however rarely it may be another **IC (integrated circuit)** which is separated into levels. The cache holds the chunk of data which are frequently used from main memory. When the processor has a single core then it will have two (or) more cache levels rarely. Present multi-core processors will be having three, 2-levels for each one core, and one level is shared.

Main Memory

The main memory in the computer is nothing but, the memory unit in the CPU that communicates directly. It is the main storage unit of the computer. This memory is fast as well as large memory used for storing the data throughout the operations of the computer. This memory is made up of RAM as well as ROM.

Magnetic Disks

The magnetic disks in the computer are circular plates fabricated of plastic otherwise metal by magnetized material. Frequently, two faces of the disk are utilized as well as many disks may be stacked on one spindle by read or write heads obtainable on every plane. All the disks in computer turn jointly at high speed. The tracks in the computer are nothing but bits which are stored within the magnetized plane in spots next to concentric circles. These are usually separated into sections which are named as sectors.

Magnetic Tape

This tape is a normal magnetic recording which is designed with a slender magnetizable covering on an extended, plastic film of the thin strip. This is mainly used to back up huge data. Whenever the computer requires to access a strip, first it will mount to access the data. Once the data is allowed, then it will be unmounted. The access time of memory will be slower within magnetic strip as well as it will take a few minutes for accessing a strip.

Advantages of Memory Hierarchy

The need for a memory hierarchy includes the following.

- Memory distributing is simple and economical
- Removes external destruction
- Data can be spread all over
- Permits demand paging & pre-paging
- Swapping will be more proficient

Semiconductor RAM memories

Semiconductor memory is used in any electronics assembly that uses computer processing technology. Semiconductor memory is the essential electronics component needed for any computer based PCB assembly.

In addition to this, memory cards have become commonplace items for temporarily storing data - everything from the portable flash memory cards used for transferring files, to semiconductor memory cards used in cameras, mobile phones and the like.

The use of semiconductor memory has grown, and the size of these memory cards has increased as the need for larger and larger amounts of storage is needed.

To meet the growing needs for semiconductor memory, there are many types and technologies that are used. As the demand grows new memory technologies are being introduced and the existing types and technologies are being further developed.

A variety of different memory technologies are available - each one suited to different applications.. Names such as ROM, RAM, EPROM, EEPROM, Flash memory, DRAM, SRAM, SDRAM, as well as F-RAM and MRAM are available, and new types are being developed to enable improved performance.

Terms like DDR3, DDR4, DDR5 and many more are seen and these refer to different types of SDRAM semiconductor memory.

In addition to this the semiconductor devices are available in many forms - ICs for printed board assembly, USB memory cards, Compact Flash cards, SD memory cards and even solid state hard drives. Semiconductor memory is even incorporated into many microprocessor chips as on-board memory.

There are two main types or categories that can be used for semiconductor technology. These memory types or categories differentiate the memory to the way in which it operates:

- ***RAM - Random Access Memory:*** As the names suggest, the RAM or random access memory is a form of semiconductor memory technology that is used for reading and writing data in any order - in other words as it is required by the processor. It is used for such applications as the computer or processor memory where variables and other stored and are required on a random basis. Data is stored and read many times to and from this type of memory.

  Random access memory is used in huge quantities in computer applications as current day computing and processing technology requires large amounts of memory to enable them to handle the memory hungry applications used today. Many types of RAM including SDRAM with its DDR3, DDR4, and soon DDR5 variants are used in huge quantities.
- ***ROM - Read Only Memory:*** A ROM is a form of semiconductor memory technology used where the data is written once and then not changed. In view of this it is used where data needs to be stored permanently, even when the power is removed - many memory technologies lose the data once the power is removed. As a result, this type of semiconductor memory technology is widely used for storing programs and data that must survive when a computer or processor is powered down. For example, the BIOS of a computer will be stored in ROM. As the name implies, data cannot be easily written to ROM. Depending on the technology used in the ROM, writing the data into the ROM initially may require special hardware. Although it is often possible to change the data, this gain requires special hardware to erase the data ready for new data to be written in.

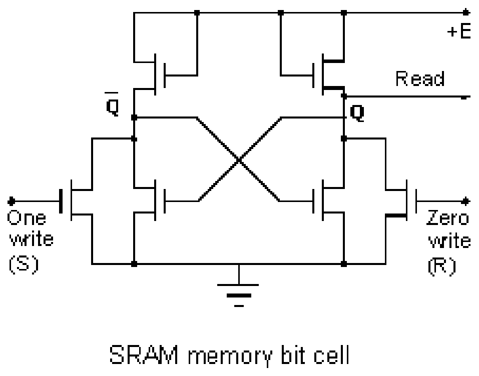
Semiconductor RAM (read-write) memory implementation

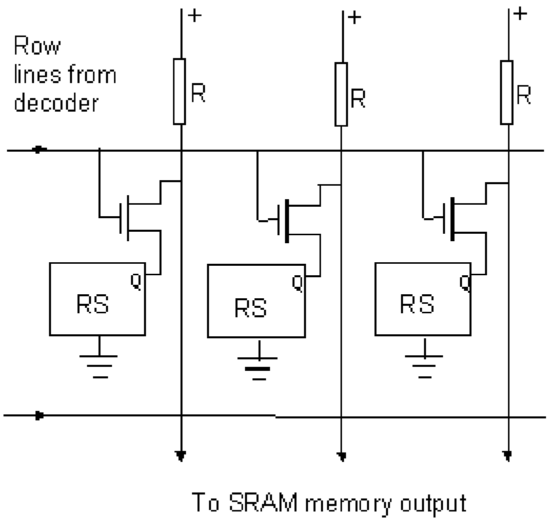
Static RAM memories

Static RAM memory cells are built as static RS flip-flops based on bipolar or MOS transistors. The structure of a basic memory bit cell built with MOS transistors (without control) is shown below. The cell is built of 6 MOS transistors that are coupled to create a static RS flip-flop. To each bit flip-flop, a row select line has to be attached. In a stable state, the Q output is in the state 0 (0V) and the "not-Q" output is in the state1 (+EV). Inserting 1 at S input makes the flip-flop enter an opposite state, for which the Q output shows 1. Inserting 1 at R input makes the flip-flop take the 0 state with 0 at Q output.

Such flip-flops are inserted at intersections of row select lines and output signal column lines in a matrix of memory cells, similar to that which appeared in the ROM memory. Each flip-flop stores one bit of a word written in a row. Outputs Q of all flip-flops in a row are connected to output column lines by means of control transistors, which are opened by signal from the row line. If a given row is selected, all flip-flops that are in the state 0, output this state to the column lines through control transistors. For flip-flops in the state 1, the control transistors will not be opened and the column lines will remain in the state 1. In a similar way, the write lines for "zeros" and "ones", which go perpendicularly in columns, are connected to all bit cells in the memory, through transistors which are opened by signals from row lines.

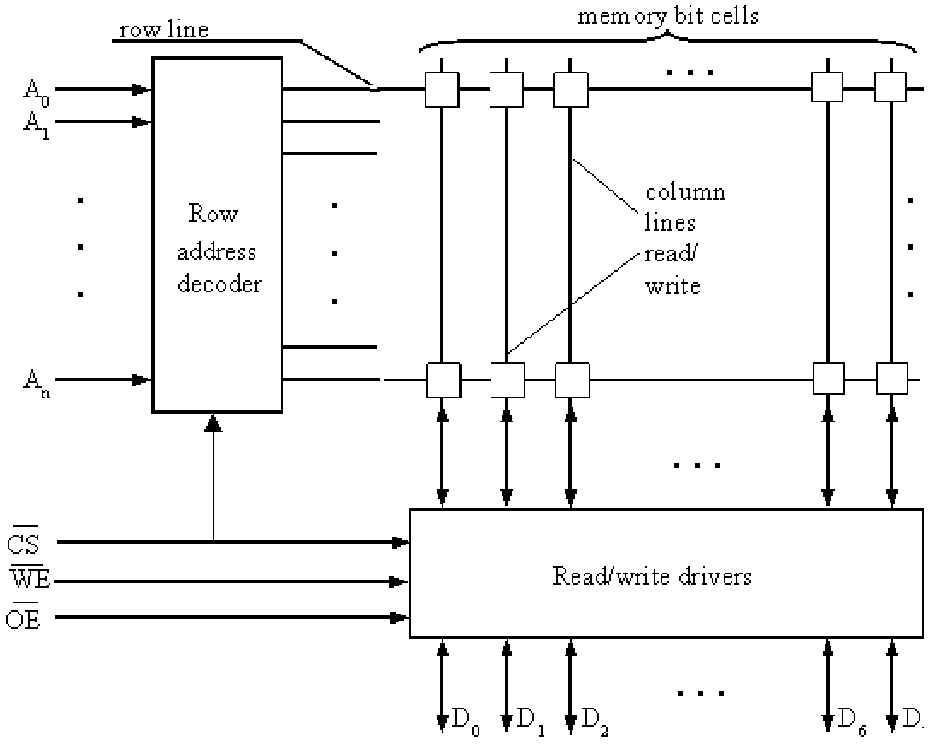
A schematic of these connections can be easily drawn as an exercise to this lecture.





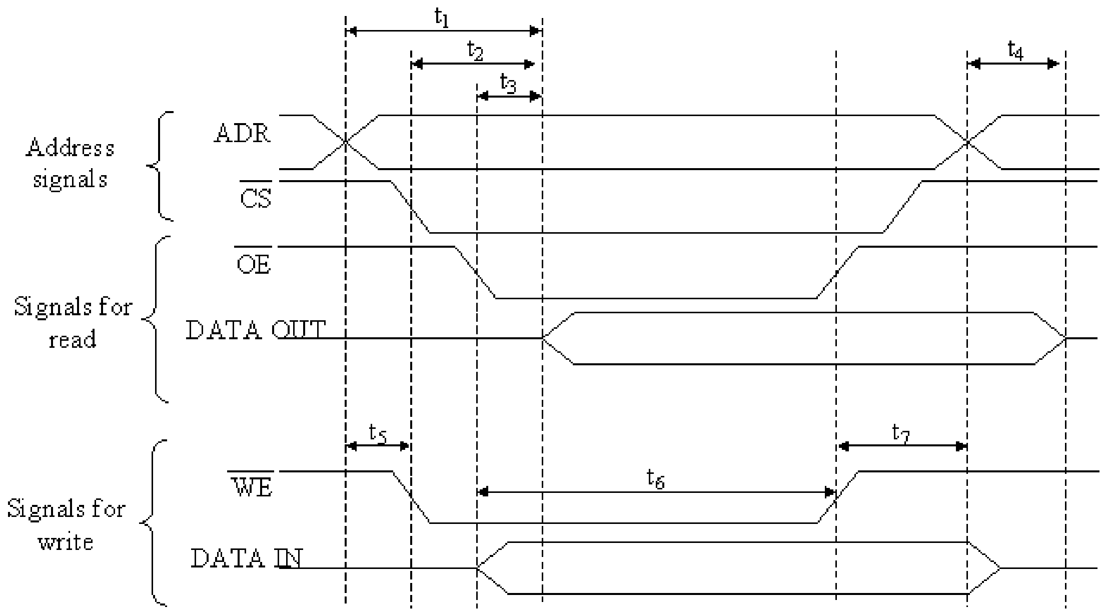
SRAM memory internal structure based on MOS transistors: bit cell and bit read control

A block diagram of a SRAM memory module with 8-bit word is shown below. We can see the pins to which address bits are supplied - A_1 - A_n, data on write and read - D_0 - D_7 (the same pins are used for this) and control signals: the memory module select signal - CS (from Chip Select), the signal that opens data input on write - WE (from Write Enable), the signal which opens data output on read - OE (from Output Enable). The CS signal is generated on the basis of decoding of the most significant address bits done outside the memory module.



Block diagram of a SRAM memory module

Timing diagram of control signals that appear in the SRAM memory cycle is shown below. The time intervals t_1, ...,t_8 specify timing requirements that assure correct memory functioning. On read, the address adr has to be supplied to the memory input with by t_1 before data issue on output. CS signal has to be supplied with the advance t_2. The OE signal that opens memory output has to be given with the advance t_3. After address is removed, data are present on output during time t_4. On write, WE and CS signals have to inserted after time t_5 in respect to address insertion time. WE signal has to supplied during an interval not shorter than t_6 and it has to complete by t_7 before a change of address. Data have to be on input during an interval not shorter than t_5 after insertion of WE. The sum of the address supply time and t_4 determines the read cycle time. The sum of t_5, t_6 and t_7 determines the minimal write cycle time.

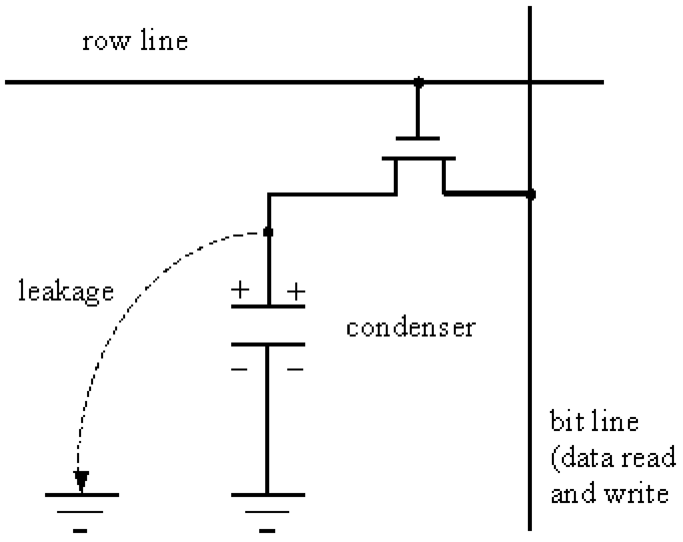


Timing signals for SRAM memory

The presented description of SRAM memory implementation concerns an **asynchronous SRAM** memory, in which data read and write operations are not synchronized by processor clock. Currently **synchronous SRAM** memories are produced, in which clock signal CLK is supplied to the memory control unit, to synchronize successive operations. These memories work in the **burst mode**, which means that after supplying memory row address, the memory control unit generates addresses for consecutive four read or write cycles, executed synchronously with the processor clock. Access times of such memories are in the range of several ns.

Dynamic RAM memories

Bit cells of the dynamic RAM memory are built based on the electric charge storing in condensers. A bit cell constitutes a transistor with a condenser interconnected to the line that selects a memory row and to the bit line in the word (bit read and write lines). The figure below shows such a bit cell based on MOS transistor. A write takes place as a result of row line selection (positive voltage) and insertion through a bit line of the voltage that corresponds to the stored bit: 0V for logical zero and the positive voltage for one. For logical one, the condenser will charge through the conducting transistor to the positive voltage. For zero, the transistor will be turned off and the condenser will discharge if loaded or it will remain not charged (in both cases the condenser plate at the transistor side will reach 0V potential). On read, a row line will be set to the positive potential and the transistor will be turned on. If the condenser was charged (the bit cell was storing one), the positive voltage from the condenser plate will be transferred into the bit line (readout of one) after which the condenser discharges through the bit line. If the condenser was not charged, 0V will be transferred to the bit line i.e. a logical zero, stored in this cell. After readout of a bit cell, the condenser has to be charged again to restore the previous contents of the memory cell. It is done by execution of the read cycle for the same information. As we can see, a data readout from dynamic RAM memory is destructive and in this memory a read cycle is always followed by a write cycle.



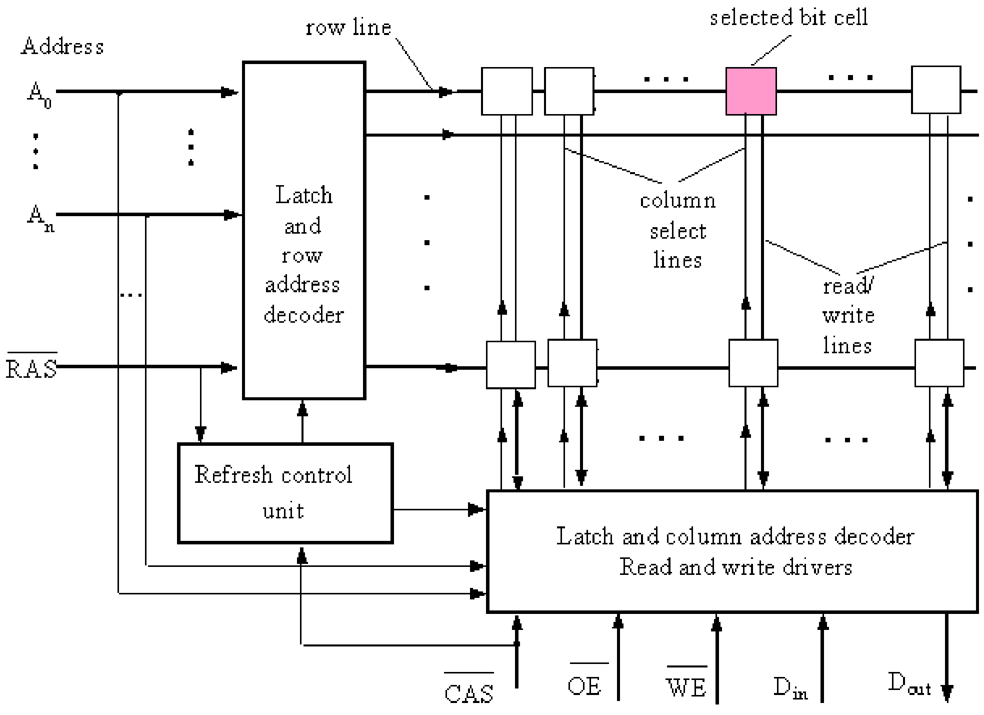
Structure of a dynamic RAM bit cell

A semiconductor dynamic RAM memory is a volatile memory, since charged condensers are subject to spontaneous discharging. The reason for this is the leakage that results from impurities in the crystalline structure of silicon. Therefore, dynamic RAM memory requires periodic refreshing of stored data. This is done by special refresh circuits, which are always added as an extension of the proper data storing circuitry.

Asynchronous dynamic RAM memories - PM DRAM i EDO RAM

Semiconductor dynamic RAM memories (**DRAMs)** are built using several techniques. The oldest one, which is not currently used is the asynchronous technique. With this technique, the memory works in asynchronous manner in respect to the processor, i.e. memory access is not synchronized by processor clock.

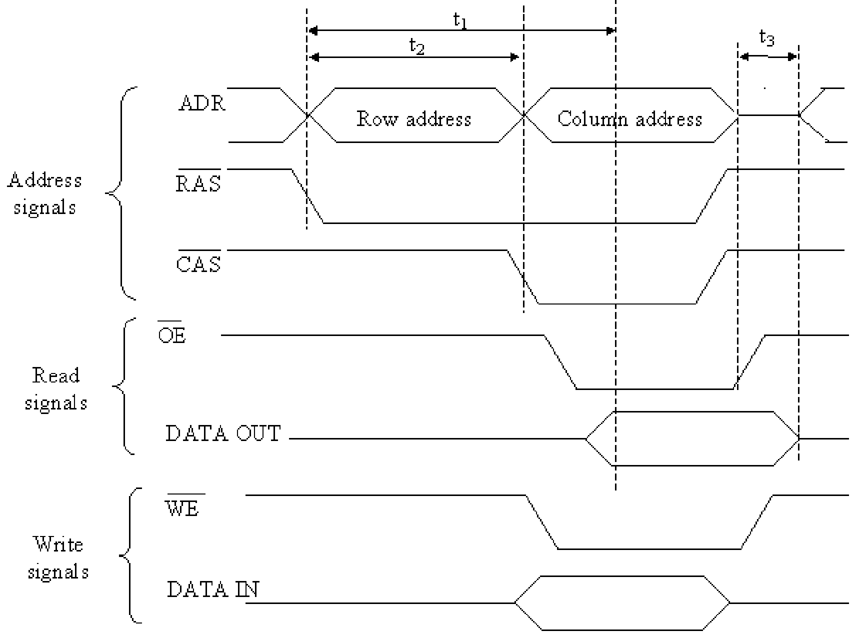
This type of memory is called **Page Mode DRAM,** from calling a row of the memory a page in memory designers jargon. A block diagram of a module of the asynchronous DRAM memory is shown below. This memory has two dimensional cell selection by the use of row and column lines. Bit cells are organized in plates, which correspond to successive bit positions in the memory word. Word address is divided into row and column addresses, which are sent to the memory module in the multiplexed way i.e. sequentially with the use of the same address bus. The row and column addresses are first latched in buffer registers that co-operate with row and column address decoders. The module includes also a control unit. Refreshing is done by rows of bit cells. During refreshing, access from the processor side and memory output to processor are blocked. The refreshing is done each several to several tens of milliseconds.



Block diagram of a DRAM memory module

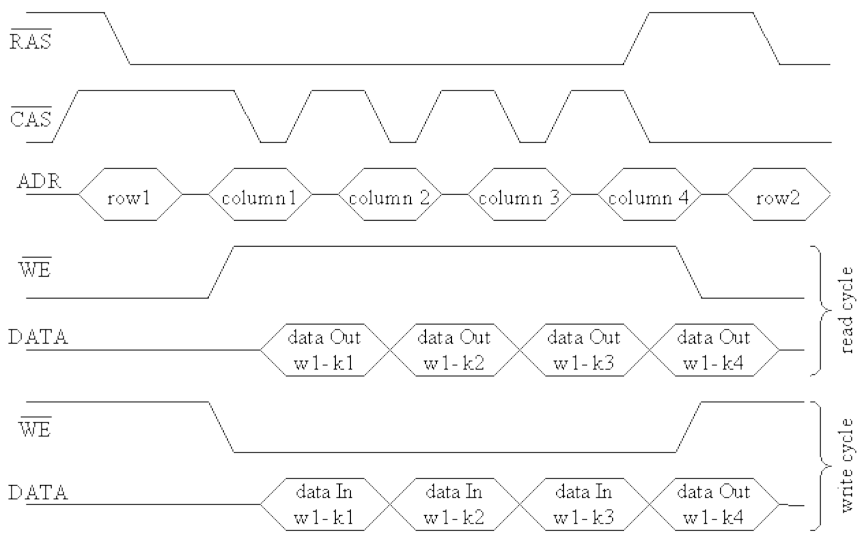
In an asynchronous DRAM memory, the pins have the following use: **A_0-A_n** - row and column addresses, **RAS** (from Row Address Strobe) - row address identification, **CAS** (from Column Address Strobe) - column address identification, **WE** (from Write Enable) - write control, **OE** (from Output Enable) - read control, **D_{in}** - input data line, **D_{out}** - output data line. The diagram below presents control signals for the asynchronous dynamic DRAM memory.

Access times for EDO RAM were several tens ns and the transmission speed was up to 300 MB/s.



Timing signals for asynchronous DRAM

Improved versions of DRAM memory were **EDO RAM i BEDO RAM (from Extended Data Out RAM** i **Burst EDO RAM**). In these memories, the **burst mode** was implemented, in which row address was inserted once each four access cycles. Next read cycle could begin before completion of the previous one. With pipelined internal functioning and internal generating of column addresses, the BEDO RAM transfer rate was increased to 500 MB/s. Timing signals for EDO RAM memory are shown below. High WE signal means read, the low one means write.

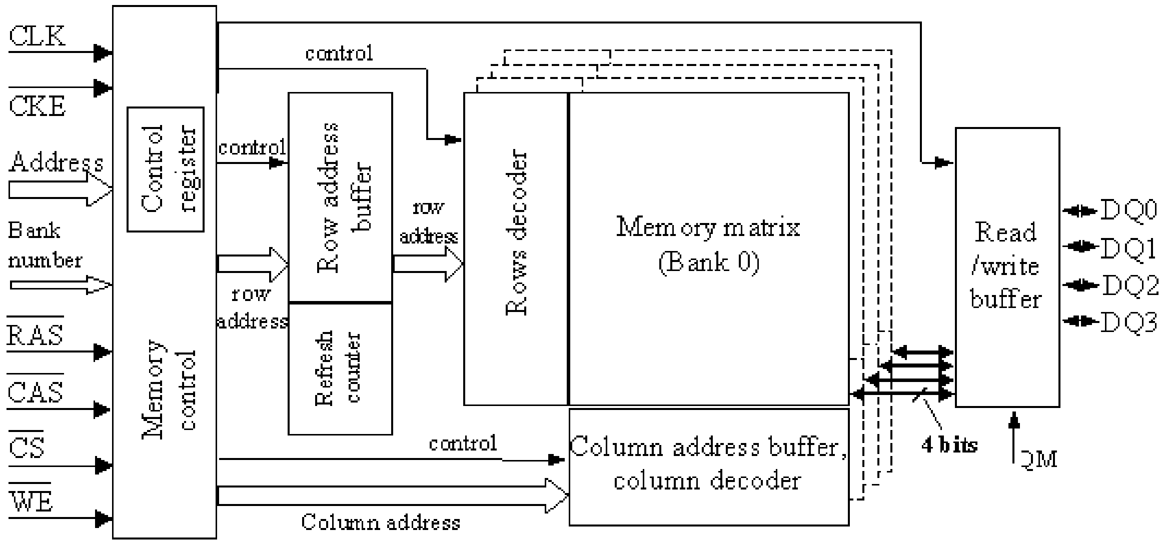


Timing signals for EDO RAM asynchronous memory.

- **Synchronous dynamic RAM memories - SDRAM**

Synchronous Dynamic RAM semiconductor memory - SDRAM is built of bit cells with two dimensional selection, similar to the asynchronous memory. The SDRAM memory is synchronized by processor clock signals. SDRAM works in the burst mode. After supplying a row and column address to the memory, the access is done not to a single cell but a package of cells (2, 4, 8) that have consecutive column addresses generated inside the memory module. Such access organization is coherent with fetching data and instructions by blocks when cache memories and instruction pre-fetching are used in modern microprocessors. All reads are synchronized by the clock ex. by rising edges in the clock pattern.

To enable burst mode, the SDRAM memory is built of several **memory banks** - i.e. of several cell matrices with independent selection (decoding) of rows and columns. With addressing of consecutive words interleaved between the memory banks, a very fast readout of series of words from consecutive memory banks is possible, without waiting for signal stabilization, which appear when reads are done in the same memory bank. The SDRAM memory features long waiting time (several clock cycles) for data after address insertion. Its access time is several ns. However, several memory banks working in parallel enables shortening read time for data at consecutive addresses. The SDRAM memories that have currently been replaced by newer memory solutions, provided transfer rates of 1 GB/s with the clock frequency of 133 MHz.



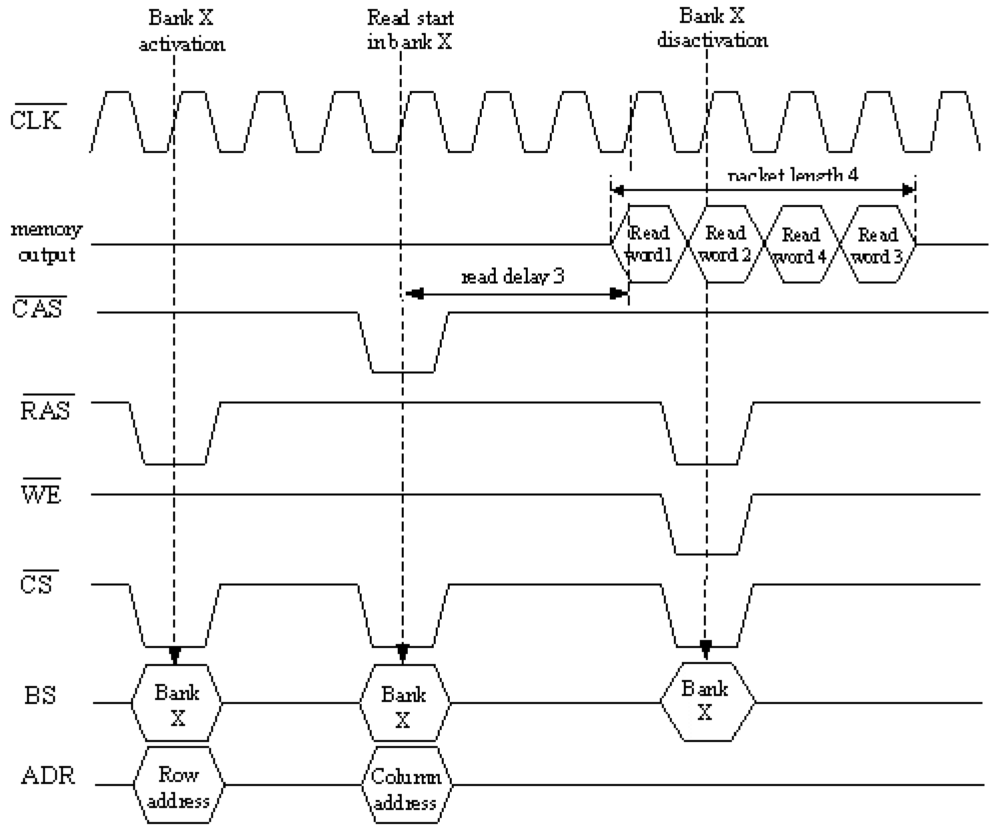
Typical SDRAM memory module organization

A typical block diagram of the SDRAM memory module is shown above. In this diagram, the memory is built of four banks, each containing 4-bit words. The module has a built in refreshing unit.

The following information and signals come to this memory module: ADR - row/column address bits, BS (from bank select) - memory bank selection bits, RAS - row address signals, CAS - column address signal, CS - module select signal, WE - write enable signal, CLK - clock signal, CKE - clock enable signal), DQM - write mask/read signal.

CKE enables or not clock signals to the memory module. In this way, the module functioning can be suspended. In a module that has no clock signals supplied no action takes place and power consumption is lowered to about 1%. The DQM signal sets the normal (unmasked) mode or the masked packet mode for read and write.

Some working parameters can be programmed for this module, such as packet length (1, 2, 4, 8), the delay after CAS, the order of bank use, a mask for packet mode, activation and disactivation of banks, interleaved or sequential address mode. The parameter setting is done with the use of ADR lines and with simultaneous insertion of the respective configuration of RAS, CAS, CS and WE signals. Word addressing is done by giving a bank number together with a row address, which is followed by a column address. A memory bank can remain active until a disactivation controlled by setting the lines as above together with the bank number in the address ADR part. All banks can be disactivated at the same time. After that banks have to be activated again. After a specified number of cycles, a readout takes place. In the packet mode, a packet containing a given number of words is read, which differ in column addresses.

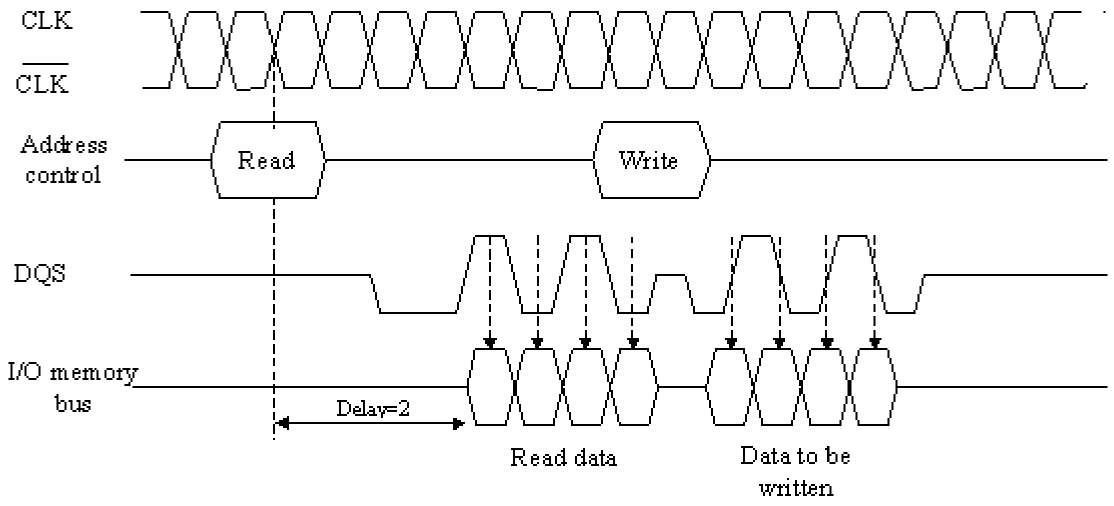


Timing signals for read from the SDRAM memory

The timing diagram shown above corresponds to the readout from a SDRAM memory in the packet mode with the packet length equal to 4 and the initial read delay equal to 3. Due to configuration signals CAS, RAS, WE, CS (complemented) set equal to 1010, a row selection and activation of the bank with the X identifier given in BS takes place. Due to configuration signals - 0110, the selection of the column and read activation take place. After 3 clock pulses, data appear on output - a packet of words from the same bank, with the same row number and consecutive column numbers, starting from the specified column number. Due to 1000 signal configuration, the disactivation of the bank with the number (X) given in BS takes place.

- **DDR SDRAM semiconductor memories**

One of the newest commonly used synchronous SDRAM memory types is the **Double Data Rate SDRAM.** Its basic construction is based on a structure with two-dimensional words selection, known from the SDRAM memory. Addresses are supplied by a multiplexed address bus, controlled by RAS and CAS signals. Packetized reads from the memory, synchronized by processor clock signals take place in this memory, built as a multi-bank structure. The essential difference in the functioning of the DDR SDRAM memory comparing the SDRAM is a different operation control. The DDR SDRAM control method is illustrated in the figure below.



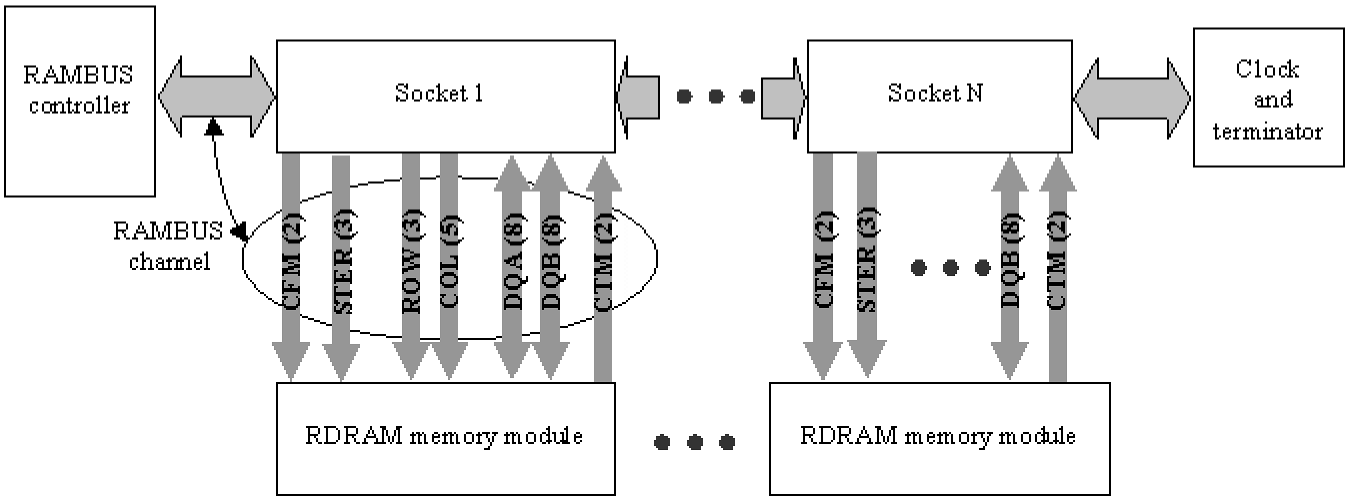
Simplified timing signals for the read and write in the DDR SDRAM memory

In the DDR SDRAM memory, two clock signals appear - CLK and complemented CLK, which are mutually shifted by half cycle. It is accompanied by the special DQS signal (from Data Query Strobe), which controls the type of operation: read or write. The two clock signals enable two times more frequent accesses to the memory, since access operations can be controlled by a coincidence of edges of both signals. In the packet mode (burst), this enables two times faster operation of the memory. The DQS signal is synchronous with the clock signal but it contains three levels: zero, low and high. The zero level is set in the passive state of the memory. To perform a write, the computer memory controller (a chipset) generates the DQS signal - (4 high-low pulses), which denote data present on the input memory bus. The pulse centers denote data present on the bus. After readout, the DDR SDRAM memory control unit generates the DQS signal (4 pulses), which informs the processor that data are present on the memory output and on the memory bus. In this case, the edges of the signal (rising and falling down) determine the presence of data on the data bus. With the clock frequency 133 MHz, the transmission data rate for the DDR SDRAM memory is equal to 2,1 GB/s.

- **RDRAM semiconductor memories**

A competitive technology for DDR SDRAM is another new technology called **RDRAM** (from **Rambus DRAM**). This technology has been developed and patented by Rambus company from the USA. The RDRAM memory is based on a fast data bus (**Rambus channel**), implemented on computer main boards as a series of sockets for integrated memory modules of the RDRAM type. The Rambus channel is based on a 8-bit address bus: 3 bits for row address (ROW) and 5 bits for column address (COL), 16-bit data bus (DQA, DQB) and 7-bit control bus (STER, CTM, CFM). The channel starts from the Rambus memory controller and ends on clock signals generator. 32 RDRAM modules can be connected to the sockets of the bus. The Rambus is controlled by a very fast clock with the frequency above 400 MHz. With the clock frequency of 400MHz, the data rate for the RDRAM is 1.6 GB/s.

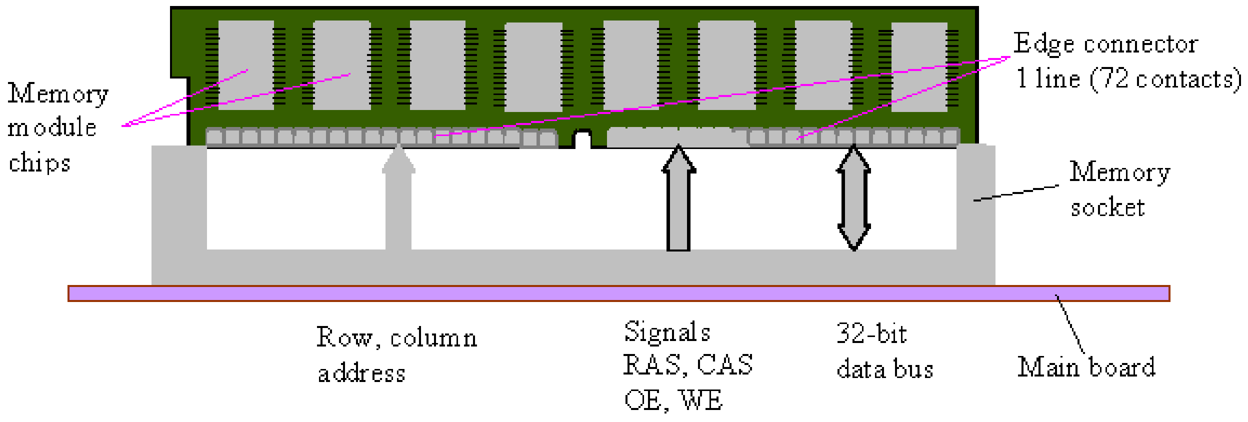
Internally, the integrated RDRAM module is based on two-dimensional selection of multi-bit dynamic RAM cells organized in 32 banks. The banks have double access with the width of 64-bits, which creates a 128 bit wide internal data bus. The access is synchronous. For synchronization of accesses to memory banks, two half-cycle shifted clock patterns are used, as in the DDR SDRAM, with synchronization done by edge coincidence of the two clock signals. In the RDRAM channel the two clock signals are distributed in a special way. One signal enters the channel at the end of the bus, traverses all sockets inside the CTM clock line (from Clock to Master) and returns to the generator as the second signal (complemented) by another line CFM (from Clock from Master). In this way, the clock signals are distributed over the RDRAM channel in a given direction synchronously with the transmitted data. A simplified block diagram of the RDRAM bus with the respective control signals is shown in the figure below.



Simplified block diagram of the RDRAM memory

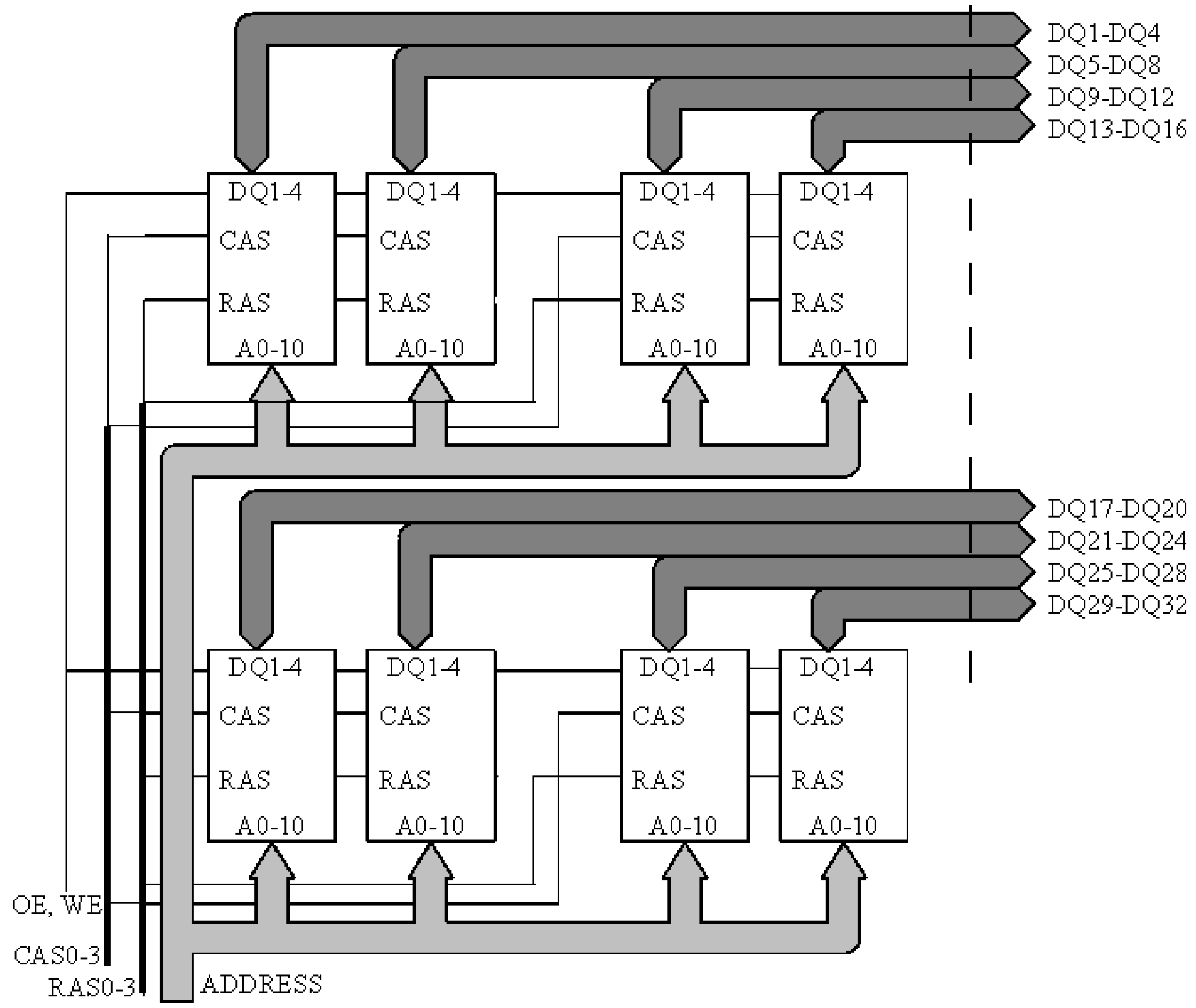
- **Design methods for semiconductor RAM memory modules**

Main memory modules are inserted into computer main board sockets as standardized modules. They constitute small printed boards with edge connectors, on which integrated chips of RAM memory modules are placed by soldering. Depending on the structure of the edge connector, one-sided or double-sided, we distinguish **SIMM** modules (from **Single In-line Memory Module**) and **DIMM** modules (from **Dual In-line Memory Module**). SIMM modules were used in old models of RAM memory. The most popular SIMM modules had 32-bit data bus and 12-bit multiplexed address bus. SIMM modules have one notch for module positioning inside a memory socket on a main board.



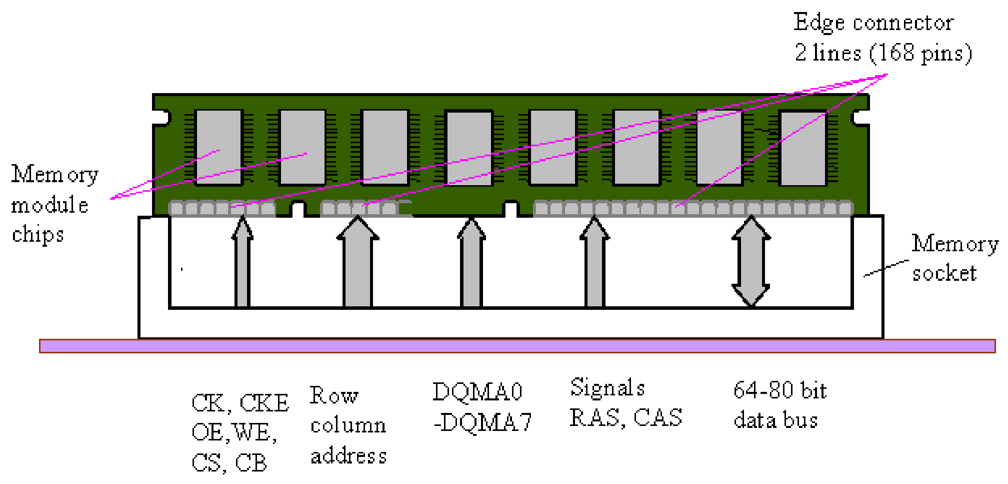
Structure of a SIMM module

Block diagram of an exemplary SIMM module of the EDO RAM memory that stores 16 MB (4 million 32-bit words) is shown below. The module is built of 8 memory chips, each storing 4 million words of 4-bit size. To all chips, the same CAS, RAS signals are supplied in parallel (CAS0-3 and RAS0-3), write and read control WE, OE and row and column address bits A0-10. The address contains 22 bits in total. Input/output lines from consecutive chips are fed in series to the edge connector (pins DQ0 - DQ31).



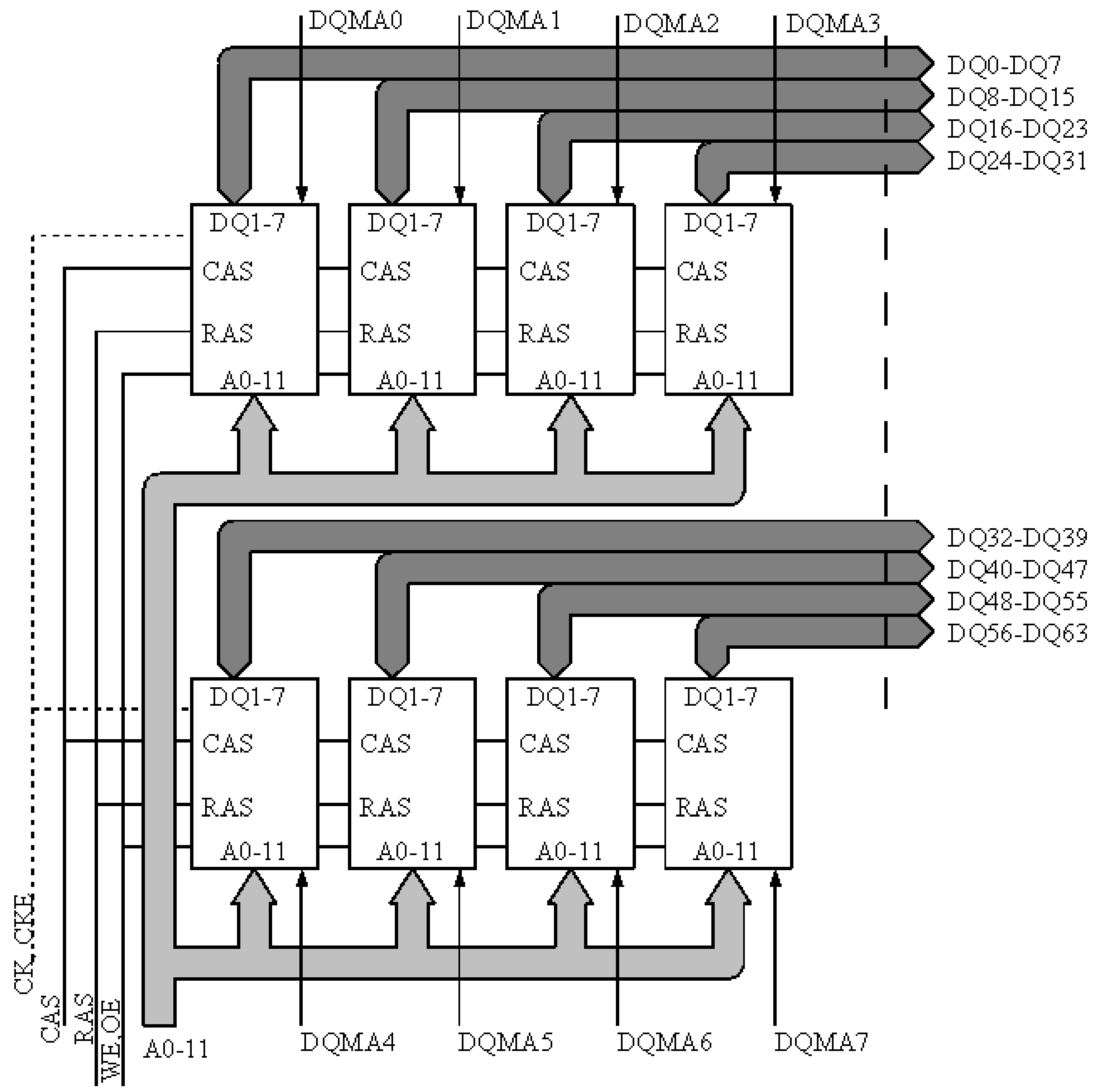
Block diagram of a SIMM module of DRAM memory

DIMM modules for DRAM and SDRAM memories have 168 pins in two lines on edge connectors. Two notches are used for positioning a module in a socket. Configuration and destination of pins on edge connectors differ with the memory type. The data bus has 64, 72 or 80-bit width and the multiplexed address bus is 14 or 16 bit wide. In DIMM SDRAM modules there are clock pins CK0-3, clock enable pins CKE0-1 and data bit masking control signals DQMA0-7. There are also CB bits for data transfer correction control (ECC -Error Correction Code or parity bits) (8 or 16 bit wide depending on the data bus width) and bank select bits CS0-CS3 (sometimes S0-S3). General structure of a DIMM module of SDRAM memory is shown below.



DIMM module structure for the SDRAM memory

A simplified block diagram of an exemplary DIMM module of the SD RAM memory which store 64 MB of data (16 million 64-bit words) is shown below. The memory is built of 8 memory chips, each storing 16 million of 8-bit words. To all chips the same CAS, RAS signals are supplied in parallel (CAS0-3 and RAS0-3), write and read control WE, OE and row/column address bits A0-11. An address contains 24 bits in total. To each memory chip, DQMA0-7 signals are separately supplied to control bit masking. Input/output lines from consecutive chips are fed in series to the edge connector (pins DQ0 - DQ63).



Block diagram of a DIMM module of SDRAM memory

There are usually several memory sockets placed on a computer main board. In SDRAM DIMM modules, a part of address bits are used as bank address bits in the same memory chip. In very large memories, DIMM sockets can be sub-divided into groups, which are selected by control signals RAS, CAS, WE, CS, generated in additional decoder units for decoding more significant bits of addresses from instruction words.

DDR SDRAM memory DIMM modules contain 184 pins and are not compatible with SDRAM DIMM modules. They require special control unit chips (chipsets) built for the DDR SDRAM memory type. Modules for RDRAM memory are built as **RIMM** modules. They have 184 pins but are not compatible with other types of memory. Because of strong heat dissipation, these modules are covered with radiators.

2D and 2.5D Memory organization

The **internal structure** of Memory either RAM or ROM is made up of memory cells that contain a memory bit. A group of 8 bits makes a byte. The memory is in the form of a multidimensional array of rows and columns. In which, each cell stores a bit and a complete row contains a word. A memory simply can be divided into this below form.

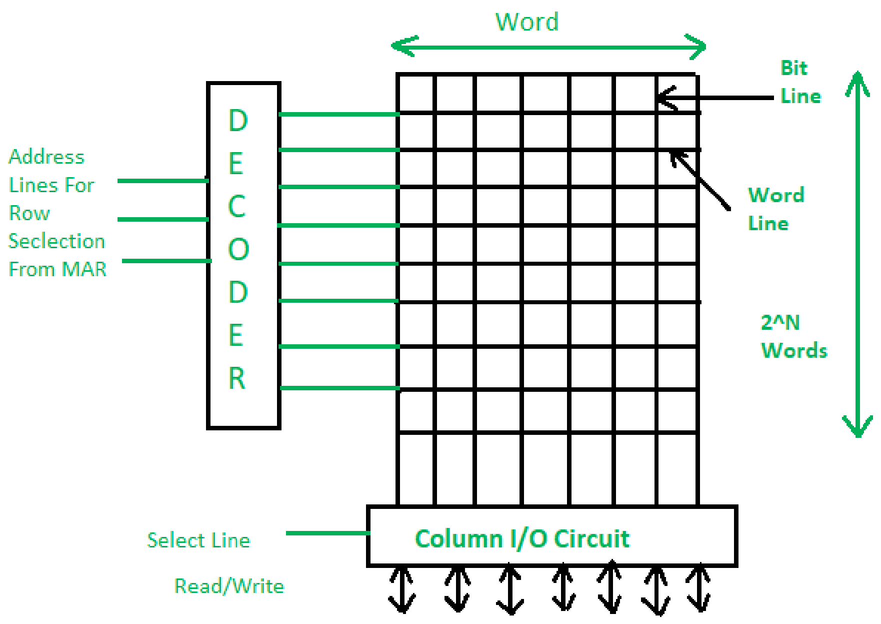
$2^n = N$

where n is the no. of address lines and N is the total memory in bytes.

There will be 2^n words.

2D Memory organization –

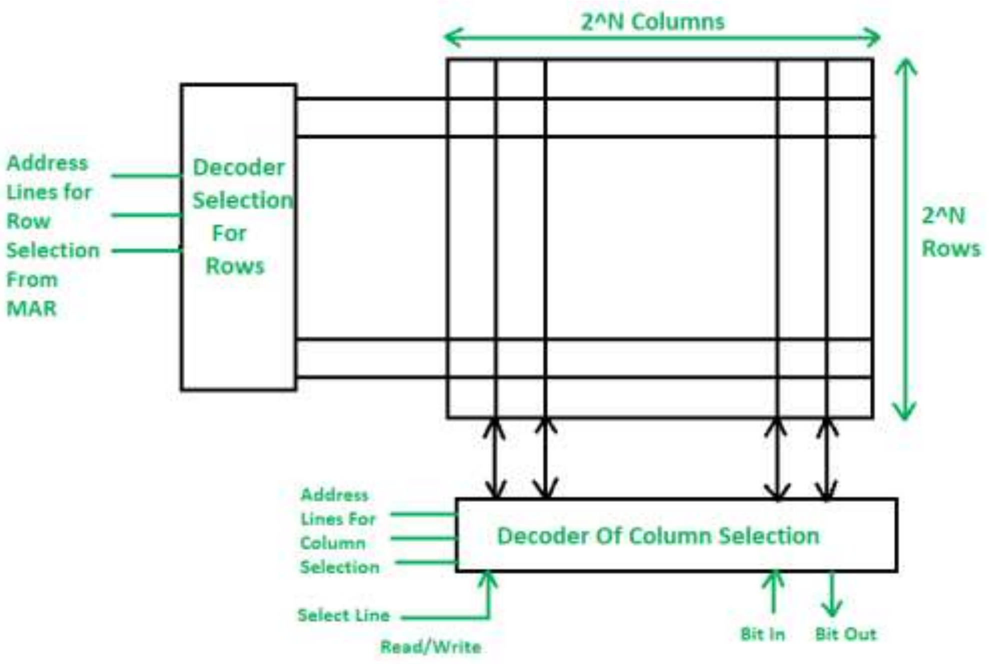
In 2D organization, memory is divided in the form of rows and columns(Matrix). Each row contains a word, now in this memory organization, there is a decoder. A decoder is a combinational circuit that contains n input lines and 2^n output lines. One of the output lines selects the row by the address contained in the MAR and the word which is represented by that row gets selected and is either read or written through the data lines.



2D Memory Organization

2.5D Memory organization –

In 2.5D Organization the scenario is the same but we have two different decoders one is a column decoder and another is a row decoder. Column decoder is used to select the column and a row decoder is used to select the row. The address from the MAR goes as the decoders' input. Decoders will select the respective cell through the bit outline, then the data from that location will be read or through the bit, inline data will be written at that memory location.


2^N Columns
Address Lines for Row Selection From MAR
Decoder Selection For Rows
2^N Rows
Address Lines For Column Selection
Decoder Of Column Selection
Select Line
Read/Write
Bit In
Bit Out

2.5D Memory Organization

Read and Write Operations –

1. If the select line is in Reading mode then the Word/bit which is represented by the MAR will be available to the data lines and will get read.
2. If the select line is in write mode then the data from the memory data register (MDR) will be sent to the respective cell which is addressed by the memory address register (MAR).
3. With the help of the select line, we can select the desired data and we can perform read and write operations on it.

Comparison between 2D & 2.5D Organizations –

1. In 2D organization hardware is fixed but in 2.5D hardware changes.
2. 2D Organization requires more gates while 2.5D requires less.
3. 2D is more complex in comparison to the 2.5D organization.
4. Error correction is not possible in the 2D organization but in 2.5D it could be done easily.
5. 2D is more difficult to fabricate in comparison to the 2.5D organization.

Read Only Memory(ROM)

ROM stands for **Read Only Memory**. The memory from which we can only read but cannot write on it. This type of memory is non-volatile. The information is stored permanently in such memories during manufacture. A ROM stores such instructions that are required to start a computer. This operation is referred to as **bootstrap**. ROM chips are not only used in the computer but also in other electronic items like washing machine and microwave oven.

Let us now discuss the various types of ROMs and their characteristics.

MROM (Masked ROM)

The very first ROMs were hard-wired devices that contained a pre-programmed set of data or instructions. These kind of ROMs are known as masked ROMs, which are inexpensive.

PROM (Programmable Read Only Memory)

PROM is read-only memory that can be modified only once by a user. The user buys a blank PROM and enters the desired contents using a PROM program. Inside the PROM chip, there are small fuses which are burnt open during programming. It can be programmed only once and is not erasable.

EPROM (Erasable and Programmable Read Only Memory)

EPROM can be erased by exposing it to ultra-violet light for a duration of up to 40 minutes. Usually, an EPROM eraser achieves this function. During programming, an electrical charge is trapped in an insulated gate region. The charge is retained for more than 10 years because the charge has no leakage path. For erasing this charge, ultra-violet light is passed through a quartz crystal window (lid). This exposure to ultra-violet light dissipates the charge. During normal use, the quartz lid is sealed with a sticker.

EEPROM (Electrically Erasable and Programmable Read Only Memory)

EEPROM is programmed and erased electrically. It can be erased and reprogrammed about ten thousand times. Both erasing and programming take about 4 to 10 ms (millisecond). In EEPROM, any location can be selectively erased and programmed. EEPROMs can be erased one byte at a time, rather than erasing the entire chip. Hence, the process of reprogramming is flexible but slow.

Advantages of ROM

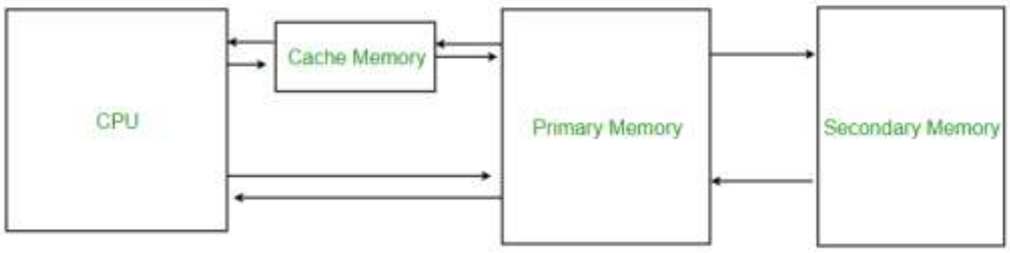
The advantages of ROM are as follows −

- Non-volatile in nature
- Cannot be accidentally changed
- Cheaper than RAMs
- Easy to test
- More reliable than RAMs
- Static and do not require refreshing
- Contents are always known and can be verified

Cache memories: concept and design issues 9 performance

Cache Memory is a special very high-speed memory. It is used to speed up and synchronizing with high-speed CPU. Cache memory is costlier than main memory or disk memory but economical than CPU registers. Cache memory is an extremely fast memory type that acts as a buffer between RAM and the CPU. It holds frequently requested data and instructions so that they are immediately available to the CPU when needed.

Cache memory is used to reduce the average time to access data from the Main memory. The cache is a smaller and faster memory which stores copies of the data from frequently used main memory locations. There are various different independent caches in a CPU, which store instructions and data. It plays a significant role in reducing the processing time of a program by provide swift access to data/instructions. Cache memory is small and fast while the main memory is big and slow.

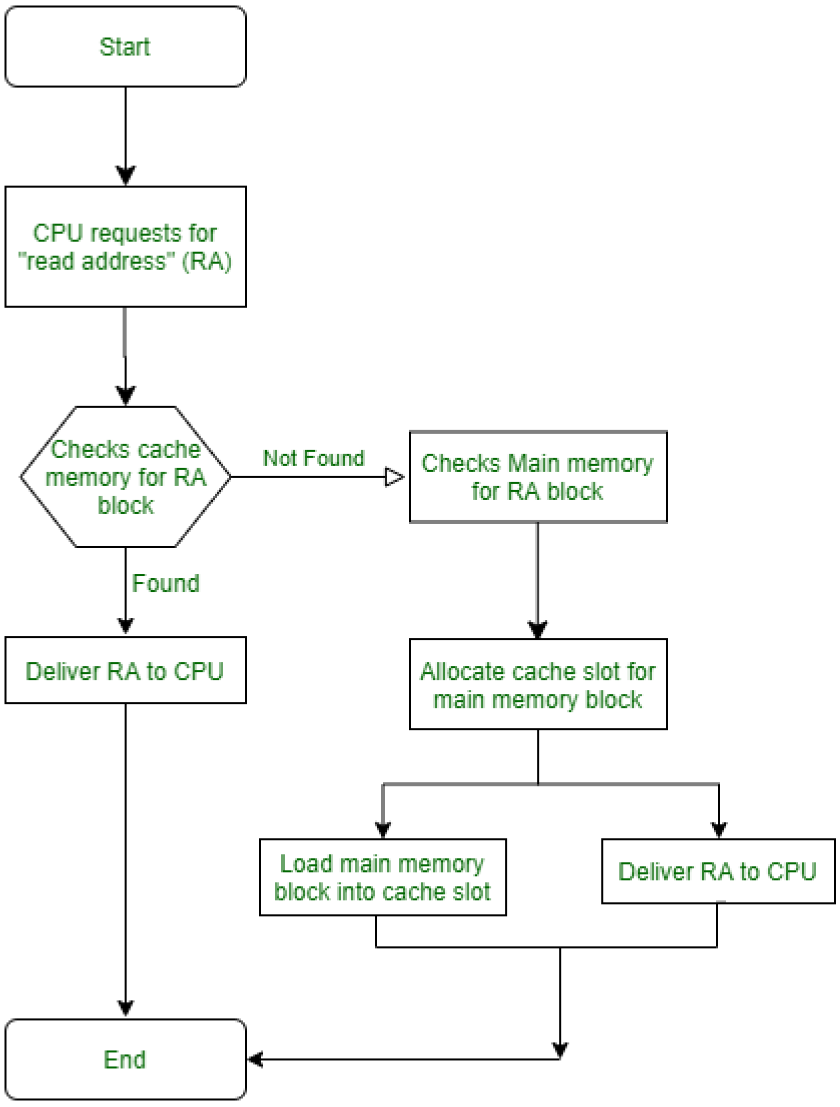


The concept of caching is explained below.

Caching Principle:

The intent of cache memory is to provide the fastest access to resources without compromising on size and price of the memory. The processor attempting to read a byte of data, first looks at the cache memory. If the byte does not exist in cache memory, it searches for the byte in the main memory. Once the byte is found in the main memory, the block containing a fixed number of byte is read into the cache memory and then on to the processor. The probability of finding subsequent byte in the cache memory increases as the block read into the cache memory earlier contains relevant bytes to the process because of the phenomenon called Locality of Reference or Principle of Locality.

Cache Memory Design :


Start
CPU requests for "read address" (RA)
Checks cache memory for RA block
Not Found
Checks Main memory for RA block
Found
Deliver RA to CPU
Allocate cache slot for main memory block
Load main memory block into cache slot
Deliver RA to CPU
End

1. **Cache Size and Block Size –**

To align with the processor speed, cache memories are very small so that it takes less time to find and fetch data. They are usually divided into multiple layers based on the architecture. The size of cache should accommodate the size of blocks, which are again determined by the processor's architecture. When block size increase, hit ratio increases initially because of the principle of locality.

Further increase in the block size resulting in bringing more data into the cache will decrease the hit ratio because, after certain point, the probability of using the new data brought in by the new block is less than the probability of reusing the data that is being flushed out to make room for newer blocks.

2. **Mapping function –**

   When a block of data is read from the main memory, the mapping function decides which location in the cache gets occupied by the read-in main memory block. There will be a need to replace the cache memory block with the main memory block if the cache is full and this rises to complexities. Which cache block should be replaced? Care should be taken to not replace the cache block that is more probable to get referred by the processor. Replacement algorithm directly depends on the mapping function such that if the mapping function is more flexible, the replacement algorithm will provide the maximum hit ratio. But, in order to provide more flexibility, the complexity of the circuitry to search cache memory to determine if the block is in the cache increases.

3. **Replacement Algorithm –**

   It decides which block in cache gets replaced by the read-in block from main memory when the cache is full, with a certain constraint from the mapping function. The block of cache that would not be referred in the near future should be replaced but it is highly improbable to determine which block would not be referred. Hence, the block in cache that has not been referred for a long time should be replaced by the new read-in block from the main memory. This is called Least-Recently-Used algorithm.

4. **Write Policy –**

   One of the most important aspect of memory caching. The block of data from cache that is chosen to be replaced by the new read-in main memory block should first be placed back in the main memory. This is to prevent loss of data. A decision should be made when the cache memory block would be put back in the main memory.

   These two available option are as follows –

   1. Place the cache memory block in the main memory when it is chosen to be replaced by a new read-in block from main memory.
   2. Place the cache memory block in the main memory after every update to the block.

      The write policy decides on when to write back the cache block on to the main memory. If option 1 is chosen, then there is an excessive write operation performed on the main memory. If option 2 is chosen, in case of a multi processor system, the block in main memory is obsolete as it has not yet been replaced from cache memory but it has undergone changes.

Levels of memory:

- **Level 1 or Register –**

  It is a type of memory in which data is stored and accepted that are immediately stored in CPU. Most commonly used register is accumulator, Program counter, address register etc.

- **Level 2 or Cache memory –**

  It is the fastest memory which has faster access time where data is temporarily stored for faster access.

- **Level 3 or Main Memory –**

  It is memory on which computer works currently. It is small in size and once power is off data no longer stays in this memory.

- **Level 4 or Secondary Memory –**

  It is external memory which is not as fast as main memory but data stays permanently in this memory.

  Cache Performance:

  When the processor needs to read or write a location in main memory, it first checks for a corresponding entry in the cache.
- If the processor finds that the memory location is in the cache, a **cache hit** has occurred and data is read from cache
- If the processor **does not** find the memory location in the cache, a **cache miss** has occurred. For a cache miss, the cache allocates a new entry and copies in data from main memory, then the request is fulfilled from the contents of the cache.

  The performance of cache memory is frequently measured in terms of a quantity called **Hit ratio.**

  Hit ratio = hit / (hit + miss) = no. of hits/total accesses

  We can improve Cache performance using higher cache block size, higher associativity, reduce miss rate, reduce miss penalty, and reduce the time to hit in the cache.

  Cache Mapping:

  There are three different types of mapping used for the purpose of cache memory which are as follows: Direct mapping, Associative mapping, and Set-Associative mapping. These are explained below.

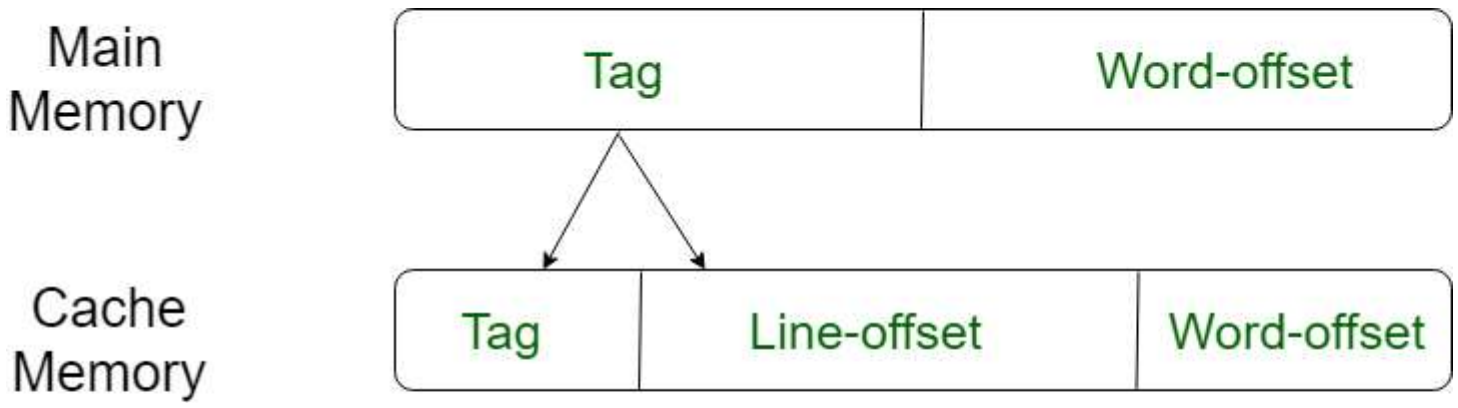
1. **Direct Mapping –**

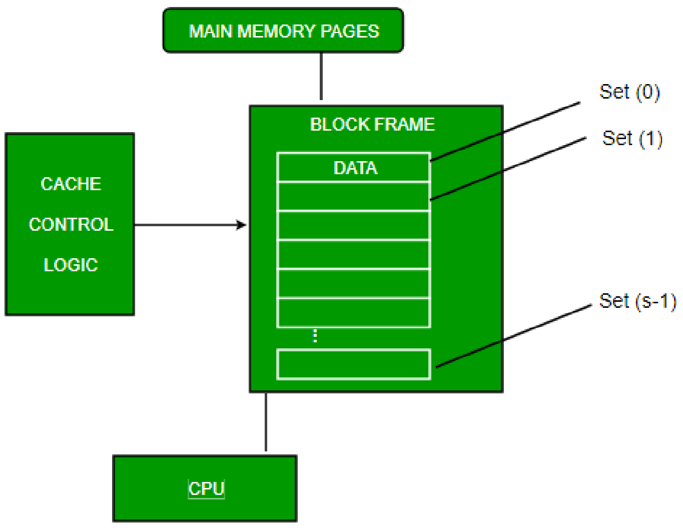
   The simplest technique, known as direct mapping, maps each block of main memory into only one possible cache line. or

   In Direct mapping, assign each memory block to a specific line in the cache. If a line is previously taken up by a memory block when a new block needs to be loaded, the old block is trashed. An address space is split into two parts index field and a tag field. The cache is used to store the tag field whereas the rest is stored in the main memory. Direct mapping`s performance is directly proportional to the Hit ratio.
2. i = j modulo m
3. where
4. i=cache line number
5. j= main memory block number

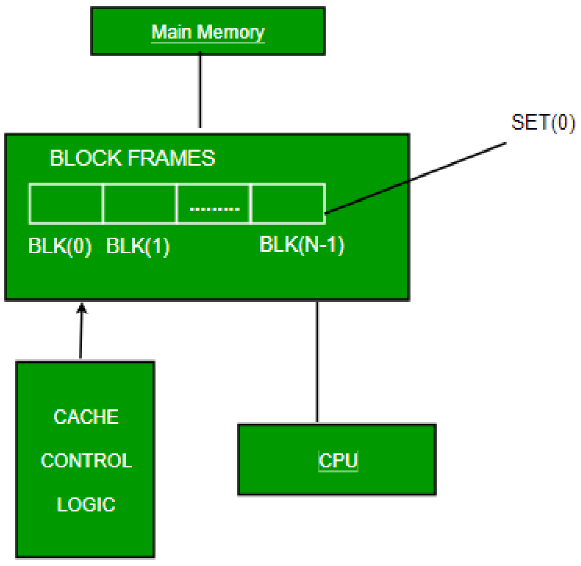
   m=number of lines in the cache

   For purposes of cache access, each main memory address can be viewed as consisting of three fields. The least significant w bits identify a unique word or byte within a block of main memory. In most contemporary machines, the address is at the byte level. The remaining s bits specify one of the 2^s blocks of main memory. The cache logic interprets these s bits as a tag of s-r bits (most significant portion) and a line field of r bits. This latter field identifies one of the $m=2^r$ lines of the cache.





6. **Associative Mapping –**
   In this type of mapping, the associative memory is used to store content and addresses of the memory word. Any block can go into any line of the cache. This means that the word id bits are used to identify which word in the block is needed, but the tag becomes all of the remaining bits. This enables the placement of any word at any place in the cache memory. It is considered to be the fastest and the most flexible mapping form.



7. **Set-associative Mapping –**
   This form of mapping is an enhanced form of direct mapping where the drawbacks of direct mapping are removed. Set associative addresses the problem of possible thrashing in the direct mapping method. It does this

by saying that instead of having exactly one line that a block can map to in the cache, we will group a few lines together creating a ***set***. Then a block in memory can map to any one of the lines of a specific set..Set-associative mapping allows that each word that is present in the cache can have two or more words in the main memory for the same index address. Set associative cache mapping combines the best of direct and associative cache mapping techniques.

In this case, the cache consists of a number of sets, each of which consists of a number of lines. The relationships are

m = v * k

i= j mod v

where

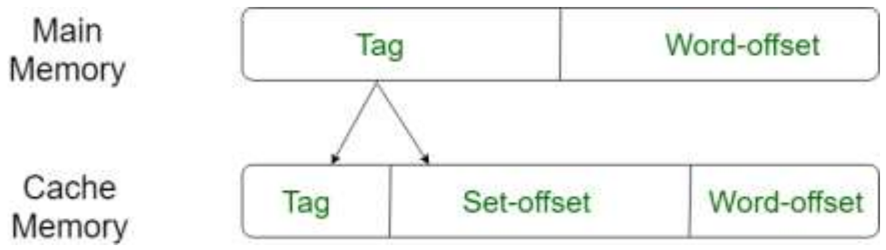
i=cache set number

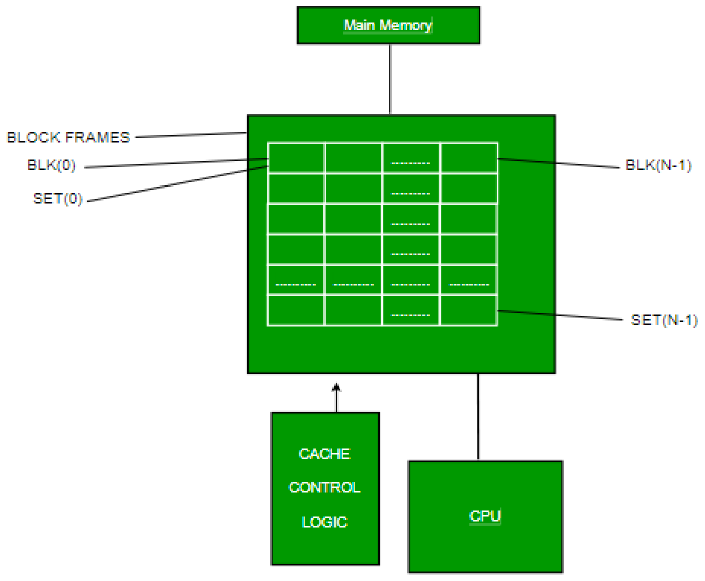
j=main memory block number

v=number of sets

m=number of lines in the cache number of sets

k=number of lines in each set





Application of Cache Memory –

1. Usually, the cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the main memory.
2. The correspondence between the main memory blocks and those in the cache is specified by a mapping function.

Types of Cache –

- **Primary Cache –**

  A primary cache is always located on the processor chip. This cache is small and its access time is comparable to that of processor registers.

- **Secondary Cache –**

  Secondary cache is placed between the primary cache and the rest of the memory. It is referred to as the level 2 (L2) cache. Often, the Level 2 cache is also housed on the processor chip.

  Locality of reference –

  Since size of cache memory is less as compared to main memory. So to check which part of main memory should be given priority and loaded in cache is decided based on locality of reference.

  Types of Locality of reference

5. **Spatial Locality of reference**

   This says that there is a chance that element will be present in the close proximity to the reference point and next time if again searched then more close proximity to the point of reference.

6. **Temporal Locality of reference**

   In this Least recently used algorithm will be used. Whenever there is page fault occurs within a word will not only load word in main memory but complete page fault will be loaded because spatial locality of reference rule says that if you are referring any word next word will be referred in its register that's why we load complete page table so the complete block will be loaded.

   Cache Mapping:

   Cache mapping is a technique by which the contents of main memory are brought into the cache memory. Cache mapping defines how a block from the main memory is mapped to the cache memory in case of a cache miss.

   Different cache mapping techniques are:

1. Direct Mapping:

a. It maps each block of main memory into only one possible cache line.

b. A particular block of main memory can map to only one particular line of the cache.

c. The line number of cache to which a particular block can map.

d. Direct mapping is a cache mapping technique that allows to map a block of main memory to only one particular cache line.

e. It means that multiple memory locations map to a single place in the cache.

2. Set Associative Mapping:

After CPU generates a memory request,

a. The set number field of the address is used to access the particular set of the cache.

b. The tag field of the CPU address is then compared with the tags of all k lines within that set.

c. If the CPU tag matches the tag of any cache line, a cache hit occurs.

d. If the CPU tag does not match to the tag of any cache line, a cache miss occurs.

e. In case of a cache miss, the required word has to be brought from the main memory.

f. If the cache is full, a replacement is made in accordance with the employed replacement policy.

3. Associative Mapping:

a. A block of main memory can be mapped to any freely available cache line.

b. This makes fully associative mapping more flexible than direct mapping.

c. A replacement algorithm is needed to replace a block if the cache is full.

d. A main memory block can load into any line of cache.

e. Memory address is interpreted as tag and word(data).

f. Tag uniquely identifies blocks of memory.

Auxiliary memories: magnetic disk, magnetic tape and optical disks

Auxiliary memory (also referred to as secondary storage) is the non-volatile memory lowest-cost, highest-capacity, and slowest-access storage in a computer system.

It is where programs and data are kept for long-term storage or when not in immediate use.

It is not directly accessible by the CPU instead, it stores noncritical system data like large data files, documents, programs and other back up information that is supplied to primary memory from auxiliary memory over a high-bandwidth channel, which will be used whenever necessary.

Auxiliary memory holds data for future use, and that retains information even the power fails.

Commonly used Auxiliary Memory:

The most common forms of auxiliary storage have been magnetic disks, magnetic tapes, and optical discs.

Types of Auxiliary Memory:

1. **Magnetic Tapes:** It is a common form of storage for mainframe computers. Information is accessed sequentially. Massive storage for low cost but retrieval is slow.

Characteristics of Magnetic Tapes

♣ No direct access, but very fast sequential access.

♣ Resistant to deferent environmental conditions.

♣ Easy to transport, store, cheaper than disk.

♣ Before, it was widely used to store application data; nowadays,

♣ it's mostly used for backups or archives (tertiary storage).

Advantages of Magnetic Tape

• **Compact:** A 10-inch diameter reel of tape is 2400 feet long and is able to hold 800, 1600 or 6250 characters in each inch of its length. The maximum capacity of such type is 180 million characters. Thus data are stored much more compact on tape

• **Economical:** The cost of storing characters on tape is very less as compared to other storage devices.

• **Fast:** Copying of data is easier and fast.

• **Long term Storage and Re-usability:** Magnetic tapes can be used for long term storage and a tape can be used repeatedly without loss of data.

2. **Magnetic Disks:** They are the foundation of external memory on virtually all computer systems. Remains the most important component of external memory. Both removable and fixed, or hard, disks are used in systems ranging from personal computers to mainframe computers and supercomputers.

3. **Flash Memory:** Flash memory is a non-volatile memory chip used for storage and for transferring data between a personal computer (PC) and digital devices. It has the ability to be electronically reprogrammed and erased. It is often found in USB flash drives, MP3 players, digital cameras and solid-state drives.

Flash memory is a type of electronically erasable programmable read only memory (EEPROM), but may also be a standalone memory storage device such as a USB drive.

4. **Optical Disk:** An optical disk is any computer disk that uses optical storage techniques and technology to read and write data. It is a storage device in which optical (light) energy is used. It is a computer storage disk that stores data digitally and uses laser beams to read and write data. It uses the optical technology in which laser light is centred to the spinning disks.

Virtual memory: concept implementation

Virtual Memory is a storage allocation scheme in which secondary memory can be addressed as though it were part of the main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites, and program-generated addresses are translated automatically to the corresponding machine addresses.

The size of virtual storage is limited by the addressing scheme of the computer system and the amount of secondary memory is available not by the actual number of the main storage locations.

It is a technique that is implemented using both hardware and software. It maps memory addresses used by a program, called virtual addresses, into physical addresses in computer memory.

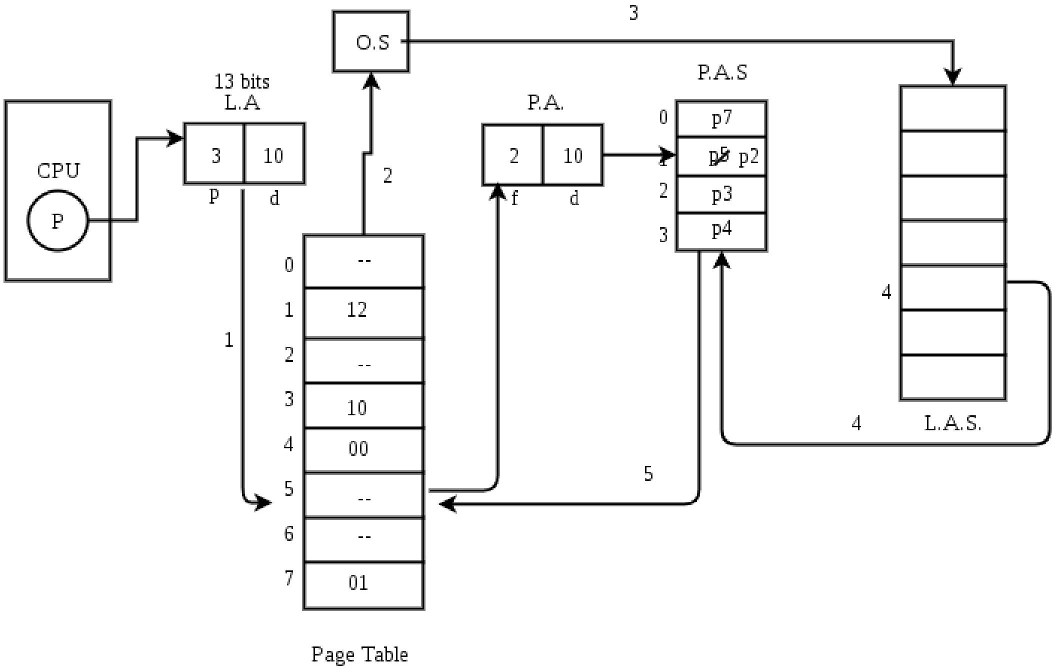
1. All memory references within a process are logical addresses that are dynamically translated into physical addresses at run time. This means that a process can be swapped in and out of the main memory such that it occupies different places in the main memory at different times during the course of execution.
2. A process may be broken into a number of pieces and these pieces need not be continuously located in the main memory during execution. The combination of dynamic run-time address translation and use of page or segment table permits this.

   If these characteristics are present then, it is not necessary that all the pages or segments are present in the main memory during execution. This means that the required pages need to be loaded into memory whenever required. Virtual memory is implemented using Demand Paging or Demand Segmentation.

Demand Paging:

The process of loading the page into memory on demand (whenever page fault occurs) is known as demand paging.

The process includes the following steps:



1. If the CPU tries to refer to a page that is currently not available in the main memory, it generates an interrupt indicating a memory access fault.
2. The OS puts the interrupted process in a blocking state. For the execution to proceed the OS must bring the required page into the memory.

3. The OS will search for the required page in the logical address space.
4. The required page will be brought from logical address space to physical address space. The page replacement algorithms are used for the decision-making of replacing the page in physical address space.
5. The page table will be updated accordingly.
6. The signal will be sent to the CPU to continue the program execution and it will place the process back into the ready state.

Hence whenever a page fault occurs these steps are followed by the operating system and the required page is brought into memory.

Advantages:

- More processes may be maintained in the main memory: Because we are going to load only some of the pages of any particular process, there is room for more processes. This leads to more efficient utilization of the processor because it is more likely that at least one of the more numerous processes will be in the ready state at any particular time.
- A process may be larger than all of the main memory: One of the most fundamental restrictions in programming is lifted. A process larger than the main memory can be executed because of demand paging. The OS itself loads pages of a process in the main memory as required.
- It allows greater multiprogramming levels by using less of the available (primary) memory for each process.

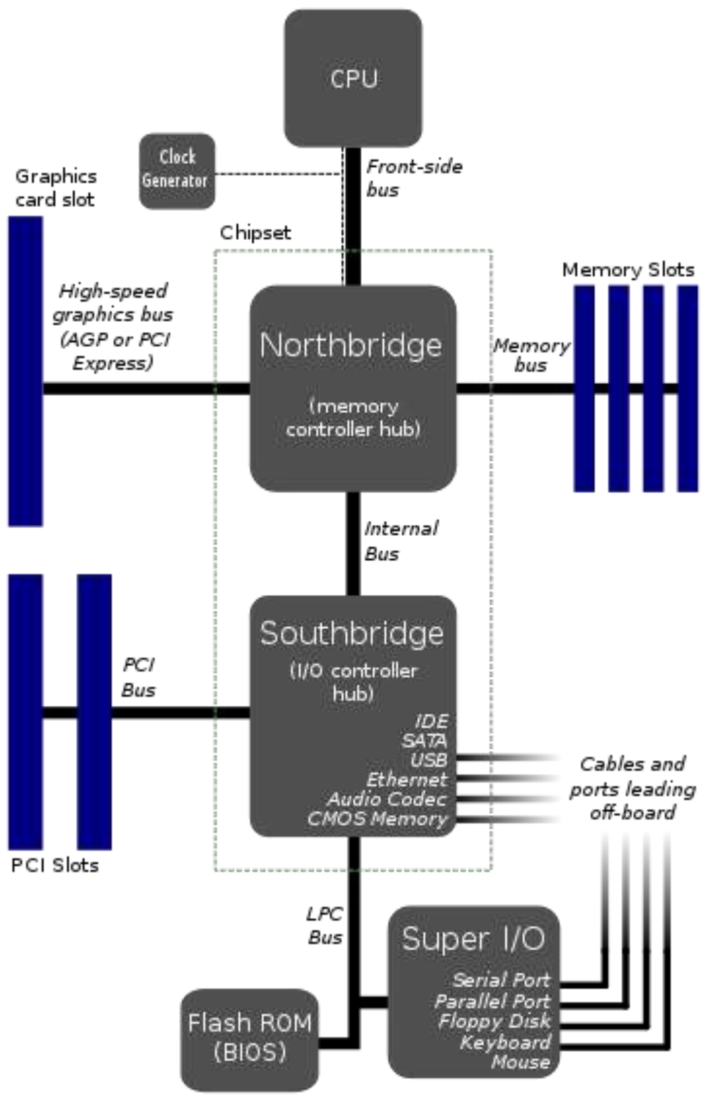
Peripheral Interfaces

With the steadily declining costs and size of integrated circuits, it is now possible to include support for many peripherals on the motherboard. By combining many functions on one PCB, the physical size and total cost of the system may be reduced; highly integrated motherboards are thus especially popular in small form factor and budget computers.

For example, the ECS RS485M-M, a typical modern budget motherboard for computers based on AMD processors, has on-board support for a very large range of peripherals:

- Disk controllers for a floppy disk drive, up to 2 PATA drives, and up to 6 SATA drives (including RAID 0/1 support)
- integrated graphics controller supporting 2D and 3D graphics, with VGA and TV output
- integrated sound card supporting 8-channel (7.1) audio and S/PDIF output
- Fast Ethernet network controller for 10/100 Mbit networking
- USB 2.0 controller supporting up to 12 USB ports
- IrDA controller for infrared data communication (e.g. with an IrDA-enabled cellular phone or printer)
- Temperature, voltage, and fan-speed sensors that allow software to monitor the health of computer components

As shown in the schematic –



Programmed I/O, Interrupt- initiated I/O and Direct memory access (DMA)

The method that is used to transfer information between internal storage and external I/O devices is known as I/O interface. The CPU is interfaced using special communication links by the peripherals connected to any computer system. These communication links are used to resolve the differences between CPU and peripheral. There exists special hardware components between CPU and peripherals to supervise and synchronize all the input and output transfers that are called interface units.

Mode of Transfer:

The binary information that is received from an external device is usually stored in the memory unit. The information that is transferred from the CPU to the external device is originated from the memory unit. CPU merely processes the information but the source and target is always the memory unit. Data transfer between CPU and the I/O devices may be done in different modes.

Data transfer to and from the peripherals may be done in any of the three possible ways

1. Programmed I/O.
2. Interrupt- initiated I/O.
3. Direct memory access (DMA).

Now let’s discuss each mode one by one.

1. **Programmed I/O:** It is due to the result of the I/O instructions that are written in the computer program. Each data item transfer is initiated by an instruction in the program. Usually the transfer is from a CPU register and memory. In this case it requires constant monitoring by the CPU of the peripheral devices.

   Example of Programmed I/O: In this case, the I/O device does not have direct access to the memory unit. A transfer from I/O device to memory requires the execution of several instructions by the CPU, including an input instruction to transfer the data from device to the CPU and store instruction to transfer the data from CPU to memory. In programmed I/O, the CPU stays in the program loop until the I/O unit indicates that it is ready for data transfer. This is a time consuming process since it needlessly keeps the CPU busy. This situation can be avoided by using an interrupt facility. This is discussed below.
2. **Interrupt- initiated I/O:** Since in the above case we saw the CPU is kept busy unnecessarily. This situation can very well be avoided by using an interrupt driven method for data transfer. By using interrupt facility and special commands to inform the interface to issue an interrupt request signal whenever data is available from any device. In the meantime the CPU can proceed for any other program execution. The interface meanwhile keeps monitoring the device. Whenever it is determined that the device is ready for data transfer it initiates an interrupt request signal to the computer. Upon detection of an external interrupt signal the CPU stops momentarily the task that it was already performing, branches to the service program to process the I/O transfer, and then return to the task it was originally performing.

   Note: Both the methods programmed I/O and Interrupt-driven I/O require the active intervention of the processor to transfer data between memory and the I/O module, and any data transfer must transverse a path through the processor. Thus both these forms of I/O suffer from two inherent drawbacks.

- The I/O transfer rate is limited by the speed with which the processor can test and service a device.
- The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfer.

3. **Direct Memory Access**: The data transfer between a fast storage media such as magnetic disk and memory unit is limited by the speed of the CPU. Thus we can allow the peripherals directly communicate with each other using the memory buses, removing the intervention of the CPU. This type of data transfer technique is known as DMA or direct memory access. During DMA the CPU is idle and it has no control over the memory buses. The DMA controller takes over the buses to manage the transfer directly between the I/O devices and the memory unit.

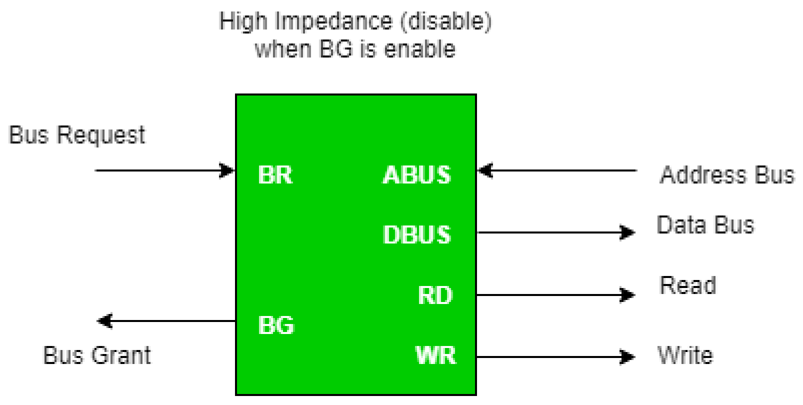


Figure - CPU Bus Signals for DMA Transfer

I/O ports

I/O stands for Input and Output. The most common device for input is the keyboard. When you type, you are putting information into the computer, which is known as input. The most common device for output is the monitor. After the information has made its way through the computer, it is sent out to the monitor for us to see. This is known as output. On the back of computers are several I/O, (or Input/Output), ports. Above, on the very top are two PS/2 ports, normally used for mouse and keyboard connections. Below that are the USB, (or Universal Serial Bus), ports. In computing, input/output or I/O is the communication between an information processing system (such as a computer) and the outside world, possibly a human or another information processing system. Inputs are the signals or data received by the system, and outputs are the signals or data sent from it. The term can also be used as part of an action; to “perform I/O” is to perform an input or output operation. I/O devices are used by a person (or other system) to communicate with a computer. For instance, a keyboard or a mouse may be an input device for a computer, while monitors and printers are considered output devices for a computer. Devices for communication between computers, such as modems and network cards, typically serve for both input and output.

An I/O interface is required whenever the I/O device is driven by the processor. The interface must have necessary logic to interpret the device address generated by the processor. Handshaking should be implemented by the interface using appropriate commands (like BUSY, READY, and WAIT), and the processor can communicate with an I/O device through the interface. If different data formats are being exchanged, the interface must be able to convert serial data to parallel form and vice-versa. There must be provision for generating interrupts and the corresponding type numbers for further processing by the processor if required.

Interrupts: Interrupt Hardware

The interrupt is a signal emitted by hardware or software when a process or an event needs immediate attention. It alerts the processor to a high-priority process requiring interruption of the current working process. In I/O devices one of the bus control lines is dedicated for this purpose and is called the *Interrupt Service Routine (ISR)*.

When a device raises an interrupt at let’s say process i, the processor first completes the execution of instruction i. Then it loads the Program Counter (PC) with the address of the first instruction of the ISR. Before loading the Program Counter with the address, the address of the interrupted instruction is moved to a temporary location. Therefore, after handling the interrupt the processor can continue with process i+1.

While the processor is handling the interrupts, it must inform the device that its request has been recognized so that it stops sending the interrupt request signal. Also, saving the registers so that the interrupted process can be restored in the future, increases the delay between the time an interrupt is received and the start of the execution of the ISR. This is called Interrupt Latency.

Hardware Interrupts:

In a hardware interrupt, all the devices are connected to the Interrupt Request Line. A single request line is used for all the n devices. To request an interrupt, a device closes its associated switch. When a device requests an interrupt, the value of INTR is the logical OR of the requests from individual devices.

The sequence of events involved in handling an IRQ:

1. Devices raise an IRQ.
2. The processor interrupts the program currently being executed.
3. The device is informed that its request has been recognized and the device deactivates the request signal.
4. The requested action is performed.

5. An interrupt is enabled and the interrupted program is resumed.

Handling Multiple Devices:

When more than one device raises an interrupt request signal, then additional information is needed to decide which device to be considered first. The following methods are used to decide which device to select: Polling, Vectored Interrupts, and Interrupt Nesting. These are explained as following below.

1. **Polling:** In polling, the first device encountered with the IRQ bit set is the device that is to be serviced first. Appropriate ISR is called to service the same. It is easy to implement but a lot of time is wasted by interrogating the IRQ bit of all devices.
2. **Vectored Interrupts:** In vectored interrupts, a device requesting an interrupt identifies itself directly by sending a special code to the processor over the bus. This enables the processor to identify the device that generated the interrupt. The special code can be the starting address of the ISR or where the ISR is located in memory and is called the interrupt vector.
3. **Interrupt Nesting:** In this method, the I/O device is organized in a priority structure. Therefore, an interrupt request from a higher priority device is recognized whereas a request from a lower priority device is not. The processor accepts interrupts only from devices/processes having priority.

   Processors' priority is encoded in a few bits of PS (Process Status register). It can be changed by program instructions that write into the PS. The processor is in supervised mode only while executing OS routines. It switches to user mode before executing application programs.

Types of interrupts and exceptions

Although interrupts have highest priority than other signals, there are many type of interrupts but basic type of interrupts are

1. **Hardware Interrupts:** If the signal for the processor is from external device or hardware is called hardware interrupts. Example: from keyboard we will press the key to do some action this pressing of key in keyboard will generate a signal which is given to the processor to do action, such interrupts are called hardware interrupts. Hardware interrupts can be classified into two types they are
   - **Maskable Interrupt:** The hardware interrupts which can be delayed when a much highest priority interrupt has occurred to the processor.
   - **Non Maskable Interrupt:** The hardware which cannot be delayed and should process by the processor immediately.
2. **Software Interrupts:** Software interrupt can also divided in to two types. They are
   - **Normal Interrupts:** the interrupts which are caused by the software instructions are called software instructions.
   - **Exception:** unplanned interrupts while executing a program is called Exception. For example: while executing a program if we got a value which should be divided by zero is called a exception.

Classification of Interrupts According to Periodicity of Occurrence:

1. **Periodic Interrupt:** If the interrupts occurred at fixed interval in timeline then that interrupts are called periodic interrupts
2. **Aperiodic Interrupt:** If the occurrence of interrupt cannot be predicted then that interrupt is called aperiodic interrupt.

Classification of Interrupts According to the Temporal Relationship with System Clock:

1. **Synchronous Interrupt:** The source of interrupt is in phase to the system clock is called synchronous interrupt. In other words interrupts which are dependent on the system clock. Example: timer service that uses the system clock.

2. **Asynchronous Interrupts:** If the interrupts are independent or not in phase to the system clock is called asynchronous interrupt.

Exception and Interrupt Handling:

Whenever an exception or interrupt occurs, execution transition from user mode to kernel mode where the exception or interrupt is handled. In detail, the following steps must be taken to handle an exception or interrupts. While entering the kernel, the context (values of all CPU registers) of the currently executing process must first be saved to memory. The kernel is now ready to handle the exception/interrupt.

1. Determine the cause of the exception/interrupt.
2. Handle the exception/interrupt.

When the exception/interrupt have been handled the kernel performs the following steps:

1. Select a process to restore and resume.
2. Restore the context of the selected process.
3. Resume execution of the selected process.

At any point in time, the values of all the registers in the CPU defines the context of the CPU. Another name used for CPU context is CPU state.

The exception/interrupt handler uses the same CPU as the currently executing process. When entering the exception/interrupt handler, the values in all CPU registers to be used by the exception/interrupt handler must be saved to memory. The saved register values can later restored before resuming execution of the process.

The handler may have been invoked for a number of reasons. The handler thus needs to determine the cause of the exception or interrupt. Information about what caused the exception or interrupt can be stored in dedicated registers or at predefined addresses in memory.

Next, the exception or interrupt needs to be serviced. For instance, if it was a keyboard interrupt, then the key code of the key press is obtained and stored somewhere or some other appropriate action is taken. If it was an arithmetic overflow exception, an error message may be printed or the program may be terminated.

The exception/interrupt have now been handled and the kernel. The kernel may choose to resume the same process that was executing prior to handling the exception/interrupt or resume execution of any other process currently in memory.

The context of the CPU can now be restored for the chosen process by reading and restoring all register values from memory.

The process selected to be resumed must be resumed at the same point it was stopped. The address of this instruction was saved by the machine when the interrupt occurred, so it is simply a matter of getting this address and make the CPU continue to execute at this address.

Modes of Data Transfer: Programmed I/O, interrupt initiated I/O and Direct Memory Access

---

Programmed I/O

- **In the programmed I/O method**, the I/O device does not have direct access to memory.
- **A transfer from an I/O** device to memory requires the execution of several instructions by the CPU, including an input instruction to transfer the data from the device to the CPU and a store instruction to transfer the data from the CPU to memory.
- **Other instructions** may be needed to verify that the data are available from the device and to count the numbers of words transferred.

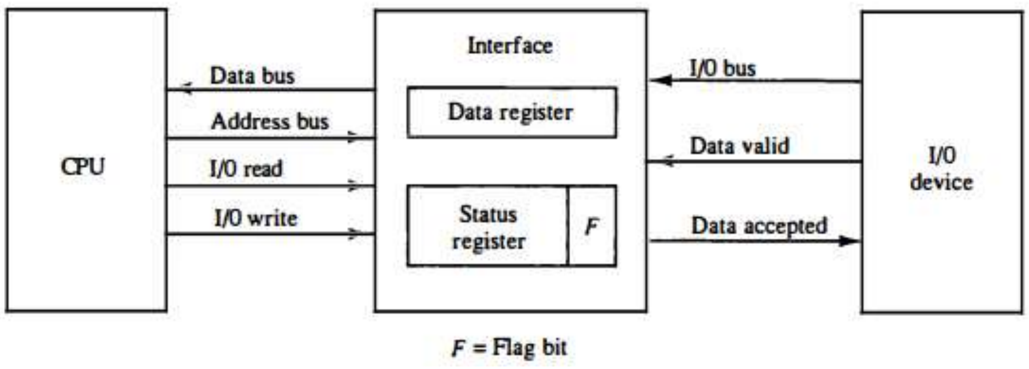
- **An example of data transfer** from an I/O device through an interface into the CPU is shown in Fig. 10. The device transfers bytes of data one at a time as they are available.
- **When a byte of data** is available, the device places it in the I/O bus and enables its data valid line.
- **The interface accepts** the byte into its data register and enables the data accepted line.
- **The interface sets** a bit in the status register that we will refer to as an F or "flag" bit. The device can now disable the data valid line, but it will not transfer another byte until the data accepted line is disabled by the interface.
- **This is according** to the handshaking procedure established in Fig. 5.
- **A program is written** for the computer to check the flag in the status register to determine if a byte has been placed in the data register by the I/O device.
- **This is done by reading** the status register into a CPU register and checking the value of the flag bit.
- **If the flag is equal** to 1, the CPU reads the data from the data register.
- **The flag bit** is then cleared to 0 by either the CPU or the interface, depending on how the interface circuits are designed.
- **Once the flag is cleared**, the interface disables the data accepted line and the device can then transfer the next data byte.
- **A flowchart of the program** that must be written for the CPU is shown in Fig. 11. It is assumed that the device is sending a sequence of bytes that must be stored in memory.

  The transfer of each byte requires three instructions:

- 1. Read the status register.
- 2. Check the status of the flag bit and branch to step 1 if not set or to step
- 3 if set.
- 3. Read the data register.

- **Each byte** is read into a CPU register and then transferred to memory with a store instruction. A common I/O programming task is to transfer a block of words from an I/O device and store them in a memory buffer.

Figure 10 Data transfer from I/O device to CPU.



-

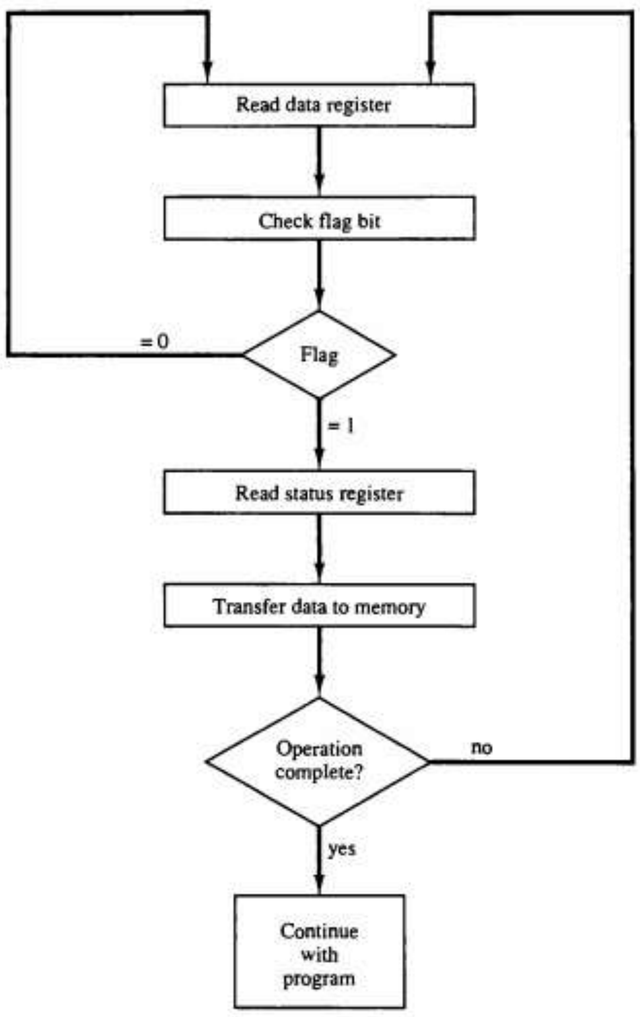


Figure ·11 Flowchart for CPU program to input data.

- **The programmed** I/O method is particularly useful in small low-speed computers or in systems that are dedicated to monitor a device continuously.
- **The difference** in information transfer rate between the CPU and the I/O device makes this type of transfer inefficient.
- **To see why this** is inefficient, consider a typical computer that can execute the two instructions that read the status register and check the flag in 1 μS.
- **Assume that** the input device transfers its data at an average rate of 100 bytes per second.
- **This is equivalent to one byte** every 10,000 μS.
- **This means** that the CPU will check the flag 10,000 times between each transfer.
- **The CPU is wasting** time while checking the flag instead of doing some other useful processing task.

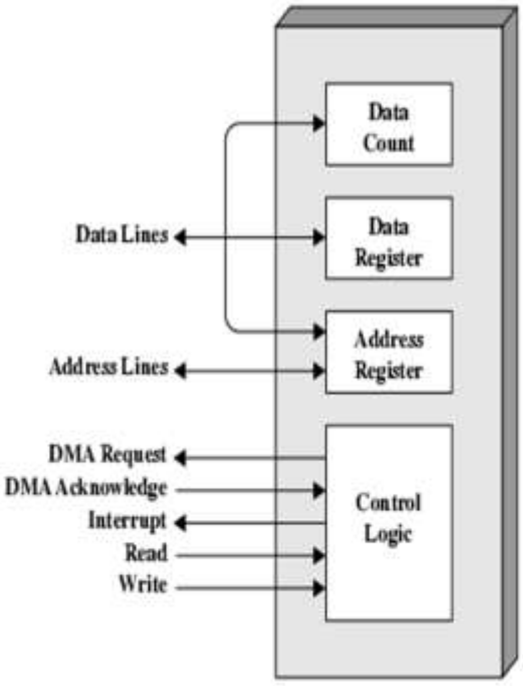
Interrupt-Initiated I/O

- **An alternative** to the CPU constantly monitoring the flag is to let the interface inform the computer when it is ready to transfer data.
- **This mode of transfer** uses the interrupt facility. While the CPU is running a program, it does not check the flag.
- **However**, when the flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that the flag has been set.
- **The CPU deviates** from what it is doing to take care of the input or output transfer.

- **After the transfer** is completed, the computer returns to the previous program to continue what it was doing before the interrupt.
- **The CPU responds** to the interrupt signal by storing the return address from the program counter into a memory stack and then control branches to a service routine that processes the required I/O transfer.
- **The way that the processor chooses** the branch address of the service routine varies from one unit to another.
- **In principle**, there are two methods for accomplishing this.
- **One is called vectored**interrupt and the other, nonvectored interrupt. In a non vectored interrupt, the branch address is assigned to a fixed location in memory.
- **In a vectored interrupt**, the source that interrupts supplies the branch information to the computer. This information is called the interrupt vector.
- **In some computers the interrupt vector** is the first address of the I/O service routine.
- **In other computers the interrupt vector** is an address that points to a location in memory where the beginning address of the I/O service routine is stored.

Direct Memory Access (DMA)

- Interrupt driven and programmed I/O require active CPU intervention
- Transfer rate is limited (processor to test and service the device)
- CPU is tied up for managing I/O transfer.
- Additional Module (hardware) on bus
- DMA controller takes over from CPU for I/ODMA is the answer.
- DMA module must use the bus only when the processor does not need it,
- It must force the processor to suspend operation temporarily. This technique is called cycle stealing



DMA Operation:

- CPU tells DMA controller:-

  –Read/Write

  –Device address

  –Starting address of memory block for data

  –Amount of data to be transferred

- CPU carries on with other work
- DMA controller deals with transfer
- DMA controller sends interrupt when finished

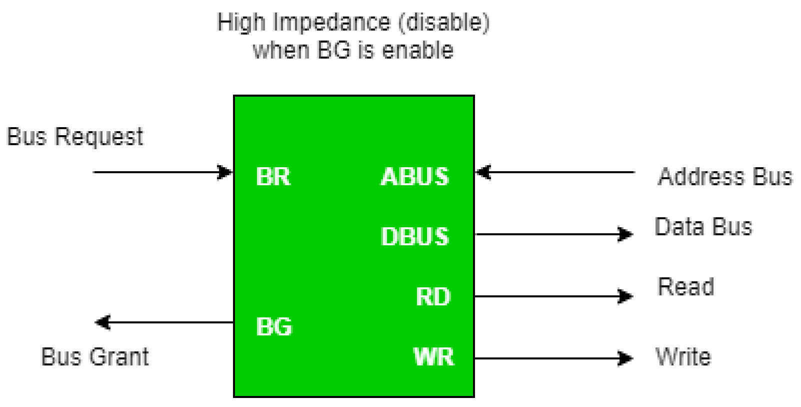


Figure - CPU Bus Signals for DMA Transfer

Cyclic Stealing:

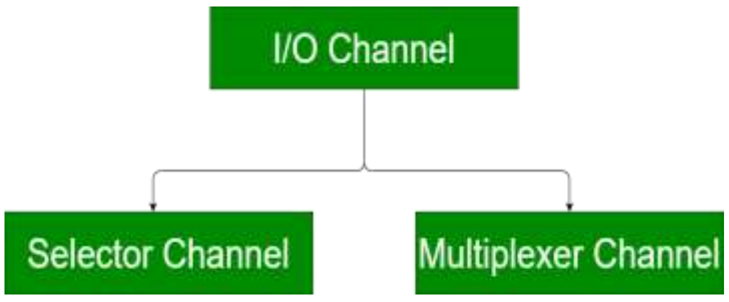
In this DMA controller transfers one word at a time after which it must return the control of the buses to the CPU. The CPU merely delays its operation for one memory cycle to allow the direct memory I/O transfer to "steal" one memory cycle.

I/O channels and processors

I/O Channel is an extension of the DMA concept. It has ability to execute I/O instructions using special-purpose processor on I/O channel and complete control over I/O operations. Processor does not execute I/O instructions itself. Processor initiates I/O transfer by instructing the I/O channel to execute a program in memory.

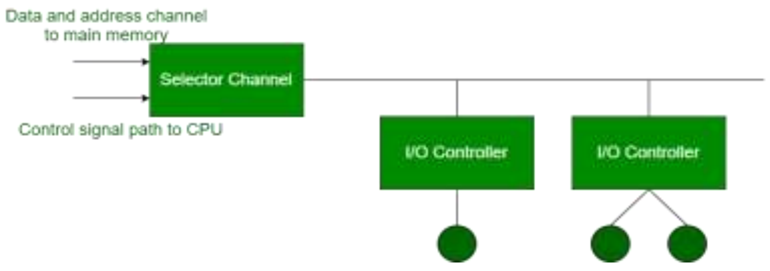
Program specifies – Device or devices, Area or areas of memory, Priority, and Error condition actions

Types of I/O Channels:



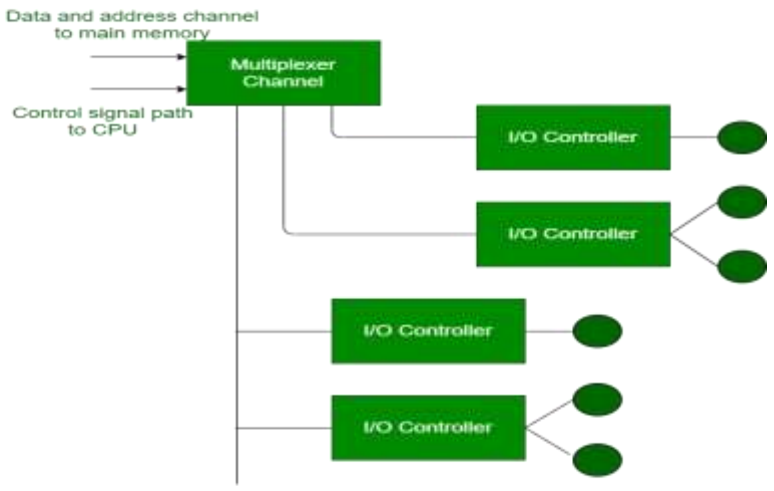
1. Selector Channel:

Selector channel controls multiple high-speed devices. It is dedicated to the transfer of data with one of the devices. In selector channel, each device is handled by a controller or I/O module. It controls the I/O controllers shown in the figure.


Data and address channel to main memory
Selector Channel
Control signal path to CPU
I/O Controller
I/O Controller

2. Multiplexer Channel:

Multiplexer channel is a DMA controller that can handle multiple devices at the same time. It can do block transfers for several devices at once.



Two types of multiplexers are used in this channel:

1. **Byte Multiplexer –**

   It is used for low-speed devices. It transmits or accepts characters. Interleaves bytes from several devices.

2. **Block Multiplexer –**

   It accepts or transmits block of characters. Interleaves blocks of bytes from several devices. Used for high-speed devices.

Synchronous Communications and Non- Synchronous Communications

Synchronous Communications

In synchronous communications, data is not sent in individual bytes, but as frames of large data blocks. Frame sizes vary from a few bytes through 1500 bytes for Ethernet or 4096 bytes for most Frame Relay systems. The clock is embedded in the data stream encoding, or provided on separate clock lines such that the sender and receiver are always in synchronization during a frame transmission. Most modern WAN framing is built on the High-Level Data Link Control (HDLC) frame structure. An HDLC frame has the following general structure:

LAG	ADDRESS	CONTROL	DATA PAYLOAD	CRC BYTES	LAG

The **flag** is a sequence 01111110 which delimits the start of the frame. A technique known as bit stuffing is used to insert additional zeros into the data so that a flag sequence never appears anywhere but at the start and end of a frame. These extra bits are “unstuffed” again by the receiver.

The **address** field is usually one byte, but may be more. It is used to indicate the sender or intended receiver of the frame. It is possible to have multiple stations connected to a single wire, and to design the system so that each receiver only “sees” frames with its own address. By this means multiple stations can communicate with each other using just one line (for instance on a Local Area Network).

The **control** field is one or more bytes. It contains information on the type of frame (for instance, whether this is a frame containing user data or a supervisory frame which performs some sort of link control function). It also often contains a rotating **sequence number** that allows the receiver to check that no frame has been lost.

The "payload" of the frame is the **data field**. The data in this field is completely transparent. In fact, it does not even have to be organized in 8 bit bytes, it is a purely arbitrary collection of bits.

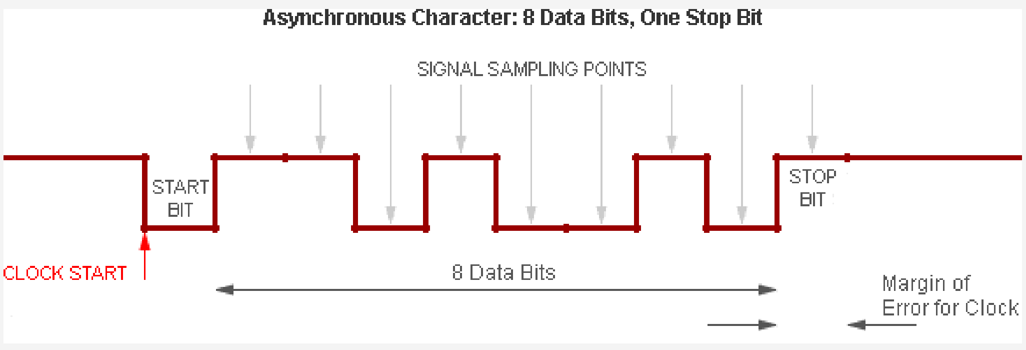
Following the data field are two bytes comprising the **Cyclic Redundancy Check**(**CRC**). The value of these bytes is the result of an arithmetic calculation based on every bit of data between the flags. When the frame is received, the calculation is repeated and compared with the received CRC bytes. If the answers match then we are sure to a very high degree of certainty that the frame has been received exactly as transmitted. If there is a CRC error the received frame is usually discarded.

Finally, the frame is terminated by another **flag** character.

Synchronous communication is usually much **more efficient** in use of bandwidth than Asynch. The data field is usually large in comparison to the flag, control, address, and CRC fields, so there is very little overhead. A 56kbps synchronous line can be expected to carry close to 7000 bytes per second (i.e. 56000/8, whereas the asynch. data rate would be 56000/10). Another advantage of synchronous communications is that the frame structure allows for easy handling of control information. There is a natural position (usually at the start of the frame) for any special codes that are needed by the communication protocol.

Asynchronous Communications

This is the method most widely used for PC or simple terminal serial communications. In asynchronous serial communication, the electrical interface is held in the mark position between characters. The start of transmission of a character is signaled by a drop in signal level to the space level. At this point, the receiver starts its clock. After one bit time (the start bit) come 8 bits of true data followed by one or more stop bits at the mark level. The receiver tries to sample the signal in the middle of each bit time. The byte will be read correctly if the line is still in the intended state when the last stop bit is read.


Asynchronous Character: 8 Data Bits, One Stop Bit
SIGNAL SAMPLING POINTS
START BIT
STOP BIT
CLOCK START
8 Data Bits
Margin of Error for Clock

Thus, the transmitter and receiver only have to have **approximately the same clock rate**. A little arithmetic will show that for a 10 bit sequence, the last bit will be interpreted correctly even if the sender and receiver clocks differ by as much as 5%.

Asynchronous is **relatively simple**, and therefore inexpensive. However, it has a **high overhead**, in that each byte carries at least two extra bits: a 25% loss of line bandwidth. A 56kbps line can only carry 5600 bytes/second asynchronously, in ideal conditions.

Communications Interfaces

Communications Interfaces means the interfaces and protocols that enable software, directories, networks, Operating Systems, network Operating Systems or Web-Based Software installed on one computer (including Personal Computers, servers and Handheld Computing Devices) to Interoperate with the Microsoft Platform

Software on another computer including without limitation communications designed to ensure security, authentication or privacy.

Examples of Communications Interfaces in a sentence

- To achieve this security, **Communications Interfaces** to all meters must be located in a physically secure location and include strong password protection with either a network firewall or encrypted connection to limit the meter's network access to the PDP and/or a defined list of authorized users.
- **Selecting and** Setting a Security Policy A security policy is a set of security settings that control how SafeNet Protect Tool kit-C is allowed to function.
- Communication system requirements are contained in a separate specification section as identified in paragraph entitled "**Communications Interfaces**".
- If the Court determines that Microsoft has undertaken such action, it shall issue an order enjoining Microsoft from asserting or enforcing any intellectual property rights in related APIs, **Communications Interfaces**, or Technical Information.
- The aforementioned license shall grant a royalty-free, non-exclusive perpetual right on a non-discriminatory basis to make, use, modify and distribute without limitation products implementing or derived from Microsoft's source code, and a royalty-free, non- exclusive perpetual right on a non-discriminatory basis to use any Microsoft APIs, **Communications Interfaces** and Technical Information used or called by Microsoft's Browser products or Browser functionality not otherwise covered by this paragraph.
- Microsoft shall disclose to each OEM and Third-Party Licensee all APIs, **Communications Interfaces** and Technical Information necessary to permit them to fully exercise their rights under Section 2.c.
- Disclosure of APIs, **Communications Interfaces** and Technical Information.
- NI's hardware products include Data Acquisition Hardware, PXI Modular Instruments, Modular Instruments, Machine Vision/Image Acquisition, Motion Control, and NI LabVIEW Reconfigurable I/O (RIO) Architecture, Industrial **Communications Interfaces**, and GPIB Interface products.
- **Communications Interfaces** Specify the various interfaces to communications such as local network protocols, etc.
- The purpose of this policy is to provide a standard for Health Net regarding communication interfaces ISS eNS1010POL - Network Security Use of Additional **Communications Interfaces**.

Related to Communications Interfaces

Hardware means any physical device that accepts input, processes input, stores the results of processing, and/or provides output.

Communications means, collectively, any notice, demand, communication, information, document or other material provided by or on behalf of any Loan Party pursuant to any Loan Document or the transactions contemplated therein which is distributed by the Administrative Agent, any Lender or the Issuing Bank by means of electronic communications pursuant to this Section, including through an Electronic System.

Electronic and Information Resources means information resources, including information resources technologies, and any equipment or interconnected system of equipment that is used in the creation, conversion, duplication, or delivery of data or information. The term includes telephones and other telecommunications products, information kiosks, transaction machines, Internet websites, multimedia resources, and office equipment, including copy machines and fax machines.