2.3 Complements

- **1's Complement Form**

  1's complement of any number can be obtained by reversing every bit of the magnitude. So if the bit in the number is 0 then it will be converted to 1 and if the bit in the number is 1 then it will be converted to 0.

For example

Magnitude	1's Complement
1100	0011
1010	0101
0110	1001

In above example it can be observed that the 1's complement has been achieved by complementing each bit in the magnitude.

Getting 1's complement of the positive number is simple because there is no change in representation but the negative number representation differs.

In sign bit of negative number is 1 and this does not change while taking complement of the number for example

Decimal Number	1's Complement representation
+6	00110
-6	11001

From above example we can understand that the sign bit of -6 does not get reversed or complemented.

- **2's Complement Form**

The 2's complement form is obtained by adding 1 to the 1's complement of the given number.

For example

Magnitude	1's Complement	2's Complement
0111	1000	1001
1000	0111+1	1000
1001	0110	0111

In case of positive numbers they are represented without any change. But in case of negative numbers, the sign bit is 1 and the magnitude is put as 2's complement for example

Decimal Number	2's Complement representation
+7	00111
-7	11001

It can be observed from above example that +7 have no effect in its conversion. In case of -7 its sign bit remains same.

2.4 Fixed Point Numbers

2.4.1 Integer Representation

The shifting process above is the key to understand fixed point number representation. To represent a real number in computers (or any hardware in general), we can define a fixed point number type simply by implicitly *fixing* the binary point to be at some position of a numeral. We will then simply adhere to this implicit convention when we represent numbers.

To define a fixed point type conceptually, all we need are two parameters:

- width of the number representation, and
- binary point position within the number

We will use the notation fixed<w,b> for the rest of this article, where w denotes the number of bits used as a whole (the Width of a number), and b denotes the position of binary point counting from the least significant bit (counting from 0).

For example, fixed<8,3> denotes a 8-bit fixed point number, of which 3 right most bits are fractional. Therefore, the bit pattern:

0	0	0	1	0	1	1	0

represents a real number:

00010.110_2
$= 1 * 2^1 + 1 * 2^{-1} + 1 * 2^{-1}$
$= 2 + 0.5 + 0.25$

= 2.75

Note that on a computer, a bit patter can represents anything. Therefore the same bit pattern, if we "cast" it to another type, such as a fixed<8,5> type, will represents the number:

000.10110_2
$= 1 * 2^{-1} + 1 * 2^{-3} + 1 * 2^{-4}$
$= 0.5 + 0.125 + 0.0625$
$= 0.6875$

If we treat this bit patter as integer, it represents the number:

10110_2
$= 1 * 2^4 + 1 * 2^2 + 1 * 2^1$
$= 16 + 4 + 2$
$= 22$

2.4.2 Arithmetic Addition

When adding two numbers, if the sum of the digits in a given position equals or exceeds the modulus, then a *carry* is propagated. For example, in Boolean addition, if two ones are added, the sum is obviously two (base 10), which exceeds the modulus of 2 for Boolean numbers ($\mathbf{B} = \mathbf{Z}_2 = \{0,1\}$, the integers modulo 2). Thus, we record a zero for the sum and propagate a carry valued at one into the next more significant digit

• $5_{ten} + 6_{ten}$

0000 0000 0000 0000 0000 0000 0000 0101 (5_{ten})

\+ 0000 0000 0000 0000 0000 0000 0000 0110 (6_{ten})

= 0000 0000 0000 0000 0000 0000 0000 1011 (11_{ten})

	...					
	...	(0)	(1)	(0)	(0)	(0)
	...	0	0	1	0	1
+	...	0	0	1	1	0
	...	0	(0)1	(1)0	(0)1	(0)1

Carries

2.4.3 Arithmetic Subtraction

When subtracting two numbers, two alternatives present themselves. First, one can formulate a subtraction algorithm, which is distinct from addition. Second, one can negate the subtrahend (i.e., in *a* - *b*, the subtrahend is *b*) then perform addition. Since we already know how to perform addition as well as twos complement negation, the second alternative is more practical.

• 12_{ten} - 5_{ten}

0000 0000 0000 0000 0000 0000 0000 1100 (12_{ten})

\- 0000 0000 0000 0000 0000 0000 0000 0101 (5_{ten})

= 0000 0000 0000 0000 0000 0000 0000 0111 (7_{ten})

• 12_{ten} - 5_{ten} = 12_{ten} + (- 5_{ten})

0000 0000 0000 0000 0000 0000 0000 1100 (12_{ten})

\+ 1111 1111 1111 1111 1111 1111 1111 1011 (-5_{ten})

= 0000 0000 0000 0000 0000 0000 0000 0111 (7_{ten})

2.4.5 Decimal Fixed-Point Representation

The shifting process above is the key to understand fixed point number representation. To represent a real number in computers (or any hardware in general), we can define a fixed point number type simply by implicitly *fixing* the binary point to be at some position of a numeral. We will then simply adhere to this implicit convention when we represent numbers.

To define a fixed point type conceptually, all we need are two parameters:

- width of the number representation, and

- binary point position within the number

We will use the notation fixed<w,b> for the rest of this article, where w denotes the number of bits used as a whole (the Width of a number), and b denotes the position of binary point counting from the least significant bit (counting from 0).

For example, fixed<8,3> denotes a 8-bit fixed point number, of which 3 right most bits are fractional. Therefore, the bit pattern:

0	0	0	1	0	1	1	0

represents a real number:

00010.110_2

$= 1 * 2^1 + 1 * 2^{-1} + 1 * 2^{-2}$

$= 2 + 0.5 + 0.25$

$= 2.75$

Note that on a computer, a bit patter can represents anything. Therefore the same bit pattern, if we "cast" it to another type, such as a fixed<8,5> type, will represents the number:

000.10110_2

$= 1 * 2^{-1} + 1 * 2^{-3} + 1 * 2^{-4}$

$= 0.5 + 0.125 + 0.0625$

$= 0.6875$

If we treat this bit patter as integer, it represents the number:

10110_2

$= 1 * 2^4 + 1 * 2^2 + 1 * 2^1$

$= 16 + 4 + 2$

$= 22$

Negative Numbers

So far we talked about positive numbers, but we do want to represent negative numbers, don't we? How do we represent fixed point negative numbers then?

In computer, we use 2's complement to represent negative numbers. One of the property of 2's complement numbers is that arithmetic operations of either positive of negative numbers are identical. It includes, addition, subtraction, and not surprisingly, shifting. We can divide negative 2's complement numbers by 2 via a simple 1 bit right shift with sign extension as we can do so with positive numbers.

Recall in the beginning of this article we discuss how fixed point numbers are simply a shifted version of an integer (by setting binary point to a non-zero position). Combining with the observation that shift operation applies to 2's complement negative number as well as positive numbers, we can easily see how we can represent negative number in fixed point: Use 2's complement.

As an illustration, below are all the numbers representable with 4-bits 2's complement:

Bit Pattern				Number Represented (n)	n / 2
1	1	1	1	-1	-0.5
1	1	1	0	-2	-1
1	1	0	1	-3	-1.5
1	1	0	0	-4	-2
1	0	1	1	-5	-2.5
1	0	1	0	-6	-3
1	0	0	1	-7	-3.5
1	0	0	0	-8	-4
0	1	1	1	7	3.5
0	1	1	0	6	3
0	1	0	1	5	2.5
0	1	0	0	4	2
0	0	1	1	3	1.5
0	0	1	0	2	1
0	0	0	1	1	0.5
0	0	0	0	0	0

Looking at this table, we can then easily realize we can represent the number -2.5 with bit pattern "1011", *IF* we assume the binary point is at position 1.

Pros and Cons of Fixed Point Number Representation

By now, you should find that fixed point numbers are indeed a close relative to integer representation. The two only differs in the position of binary point. In fact, you might even consider integer representation as a "special case" of fixed point numbers, where the binary point is at position 0. All the arithmetic operations a computer can operate on integer can therefore be applied to fixed point number as well.

Therefore, the benefit of fixed point arithmetic is that they are as straight-forward and efficient as integers arithmetic in computers. We can reuse all the hardware built to for integer arithmetic to perform real numbers arithmetic using fixed point representation. In other word, fixed point arithmetic comes for free on computers.

The disadvantage of fixed point number, is than of course the loss of range and precision when compare with floating point number representations. For example, in a fixed<8,1> representation, our fractional part is only precise to a quantum of 0.5. We cannot represent number like 0.75. We can represent 0.75 ith fixed<8,2>, but then we loose range on the integer part.

2.5 Floating Point Numbers

The floating point number is represented in following three parts:

1) Mantissa — fractional
2) Base — sign bit
3) Exponent

The following are the numbers in which mantissa, base and exponent are shown separately.

Number	Mantissa	Base	Exponent
$3 * 10^6$	3	10	6
$110 * 2^8$	110	2	8
2343.252	0.2343252	10	4
35.56	0.3456	10	2

The mantissa and exponent are explicitly represented in the computer. But the base is implied. Generally most of the computer follows the base as 2. In general a number f is represented as $f = m * r^e$ where m is mantissa, r is base and e is exponent. The figure 1 shows the format.

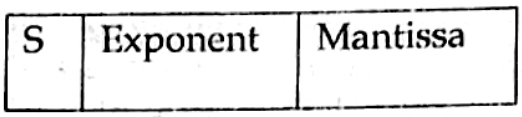


Figure 1 : Floating Point Number Format

Figure 2 and Figure 3 shows the floating point number format with single and double precision.

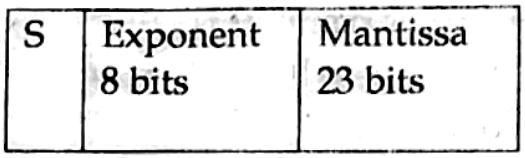


Figure 2 : Single Precision Floating-Point Number Format (32 bits)

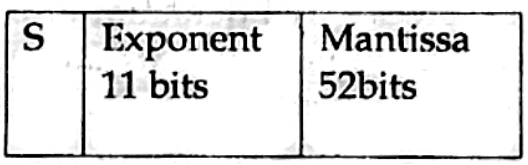


Figure : 3 : Double Precision Floating-Point Number Format (64 bits)

2.5.1 IEEE Representation

There are several ways to represent real numbers on computers. IEEE (Institute of Electrical and Electronics Engineers) has developed a standard for floating point number representation.

It is just like a one standard that we have other like SQL and ASCII. This kind of standard is necessary for the portability of the program. It helps in transferring the data written on one computer to another. So we can transfer floating point numbers from one machine to another machine.

Floating-point representation - the most common solution - basically represents reals in scientific notation. Scientific notation represents numbers as a base number and an exponent. For example, 123.456 could be represented as 1.23456×10^2. In hexadecimal, the number 123.abc might be represented as $1.23abc \times 16^2$.

Floating-point solves a number of representation problems. Fixed-point has a fixed window of representation, which limits it from representing very large or very small numbers. Also, fixed-point is prone to a loss of precision when two large numbers are divided.

Floating-point, on the other hand, employs a sort of "sliding window" of precision appropriate to the scale of the number. This allows it to represent numbers from 1,000,000,000,000 to 0.0000000000000001 with ease.

Storage Layout

IEEE floating point numbers have three basic components: the sign, the exponent, and the mantissa. The mantissa is composed of the *fraction* and an implicit leading digit (explained below). The exponent base (2) is implicit and need not be stored.

The following figure shows the layout for single (32-bit) and double (64-bit) precision floating-point values. The number of bits for each field are shown (bit ranges are in square brackets):

	Sign	Exponent	Fraction	Bias
Single Precision	1 [31]	8 [30-23]	23 [22-00]	127
Double Precision	1 [63]	11 [62-52]	52 [51-00]	1023

The Sign Bit

The sign bit is as simple as it gets. 0 denotes a positive number; 1 denotes a negative number. Flipping the value of this bit flips the sign of the number.

The Exponent

The exponent field needs to represent both positive and negative exponents. To do this, a *bias* is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200-127), or 73. For reasons discussed later, exponents of -127 (all 0s) and +128 (all 1s) are reserved for special numbers.

For double precision, the exponent field is 11 bits, and has a bias of 1023.

The Mantissa

The *mantissa*, also known as the *significant*, represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits.

To find out the value of the implicit leading bit, consider that any number can be expressed in scientific notation in many different ways. For example, the number five can be represented as any of these:

5.00×10^{0}
0.05×10^{2}
5000×10^{-3}

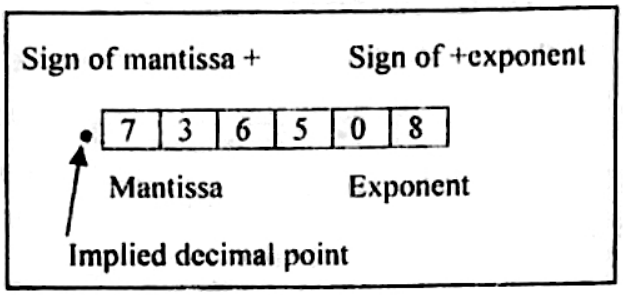
In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in *normalized* form. This basically puts the radix point after the first non-zero digit. In normalized form, five is represented as 5.0×10^{0}.

A nice little optimization is available to us in base two, since the only possible non-zero digit is 1. Thus, we can just assume a leading digit of 1, and don't need to represent it explicitly. As a result, the mantissa has effectively 24 bits of resolution, by way of 23 fraction bits.

2.5.2 Normalization

The shifting of the decimal point to the left of the most significant digit is known as normalization and the real numbers expressed in this type are called normalized floating point number. In the technique of expressing and storing real numbers, real number is represented as a combination of

a mantissa and an exponent. The mantissa lies between 1 and 1 (i.e. $1 \leq |\text{mantissa}| \leq 1.0$). The exponent is the power of 10 which multiplies the mantissa. e.g. The number 73.65×10^6 is expressed as .7365 E8 where E8 denotes 10^8. Here mantissa is .7365 & exponent is 8. Following figure shows the number stored in memory.



2.5.3 Excess Code

Excess code is a format of expressing the exponent. In this format the exponent field contains a number which is achieved by adding 128 to the actual exponent.

If 8 bits are used for exponent the 256 combinations are possible and range would be -128 to +127.

If Excess 128 code technique is used for 8 bit exponent then negative exponent will be from 0 to 127 and the positive exponent will range from 128 to 255 in the coded form.

2.6 Error Detection Codes

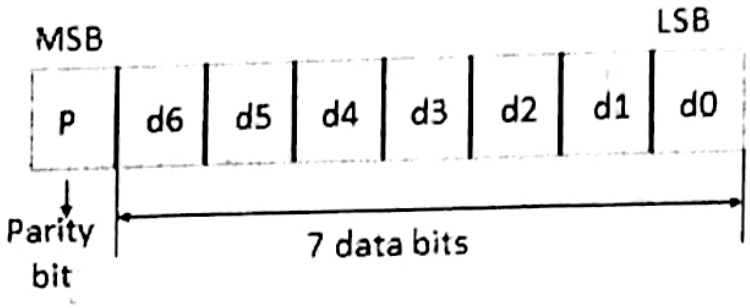
Whenever a message is transmitted, it may get scrambled by noise or data may get corrupted. To avoid this, we use error-detecting codes which are additional data added to a given digital message to help us detect if an error occurred during transmission of the message. A simple example of error-detecting code is parity check.

Ex. Parity Checking of Error Detection

It is the simplest technique for detecting and correcting errors. The MSB of an 8-bits word is used as the parity bit and the remaining 7 bits are used as data or message bits. The parity of 8-bits transmitted word can be either even parity or odd parity.

- additional bits are added to the data bits at the time of transmissi

Parity bit



Even parity -- Even parity means the number of 1's in the given word including the parity bit should be even (2,4,6,....).

Odd parity -- Odd parity means the number of 1's in the given word including the parity bit should be odd (1,3,5,....).

Use of Parity Bit

The parity bit can be set to 0 and 1 depending on the type of the parity required.

- For even parity, this bit is set to 1 or 0 such that the no. of "1 bits" in the entire word is even. Shown in fig. (a).
- For odd parity, this bit is set to 1 or 0 such that the no. of "1 bits" in the entire word is odd. Shown in fig. (b).

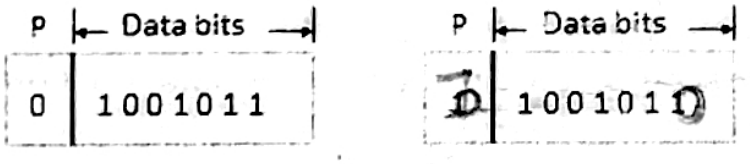


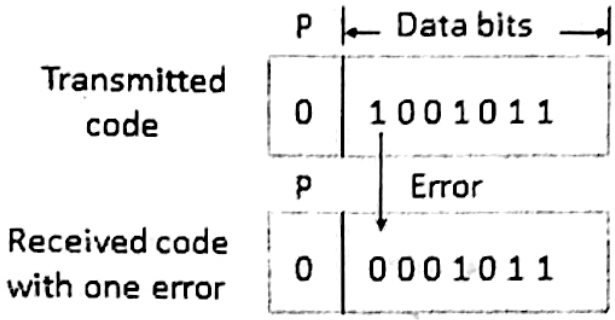
Fig. (a)

Fig. (b)

How Does Error Detection Take Place?

Parity checking at the receiver can detect the presence of an error if the parity of the receiver signal is different from the expected parity. That

means, if it is known that the parity of the transmitted signal is always going to be "even" and if the received signal has an odd parity, then the receiver can conclude that the received signal is not correct. If an error is detected, then the receiver will ignore the received byte and request for retransmission of the same byte to the transmitter.



2.7 Register Transfer Language

The symbolic notation used to describe the micro-operation transfers amongst registers is called Register transfer language.

The term register transfer means the availability of hardware logic circuits that can perform a stated micro-operation and transfer the result of the operation to the same or another register.

The word language is borrowed from programmers who apply this term to programming languages. This programming language is a procedure for writing symbols to specify a given computational process.

Following are some commonly used registers:

1. **Accumulator:** This is the most common register, used to store data taken out from the memory.

2. **General Purpose Registers:** This is used to store data intermediate results during program execution. It can be accessed via assembly programming.

3. **Special Purpose Registers:** Users do not access these registers. These registers are for Computer system,

   - **MAR:** Memory Address Register are those registers that holds the address for memory unit.
   - **MBR:** Memory Buffer Register stores instruction and data received from the memory and sent from the memory.
   - **PC:** Program Counter points to the next instruction to be executed.
   - **IR:** Instruction Register holds the instruction to be executed.

2.8 Register Transfer

Information transferred from one register to another is designated in symbolic form by means of replacement operator.

$$R2 \leftarrow R1$$

It denotes the transfer of the data from register R1 into R2.

Normally we want the transfer to occur only in predetermined control condition.

This can be shown by following if-then statement: if (P=1) then (R2 ← R1)

Here P is a control signal generated in the control section.

2.9 Bus and Memory Transfers

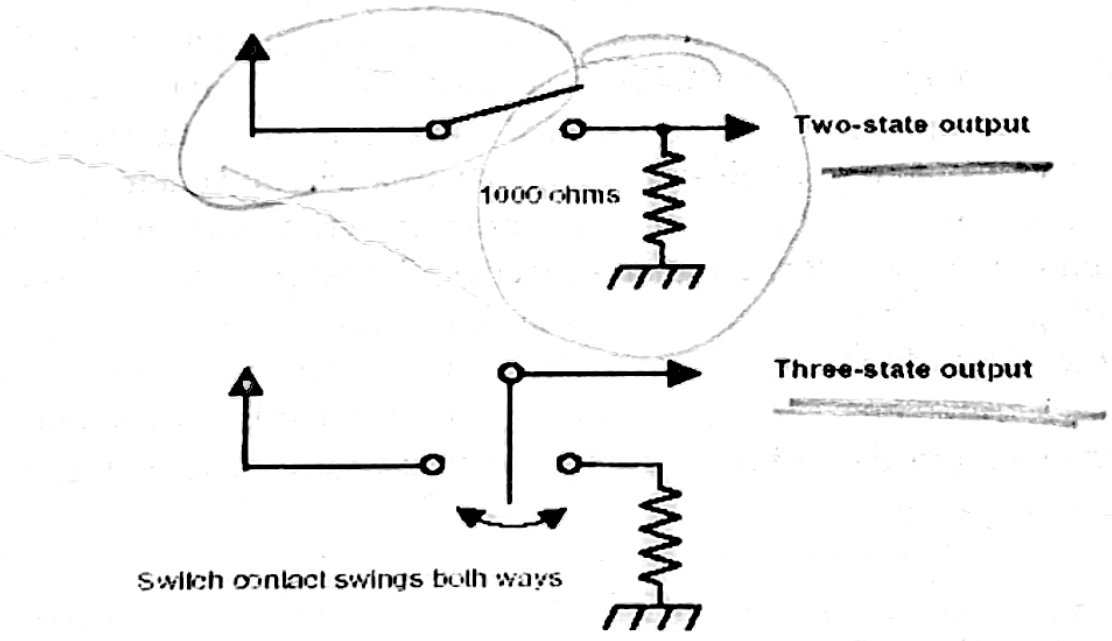
2.9.1 Three-State Bus Buffers

Computer systems often use a bi-directional data bus to allow convenient connection of memory and other devices to the registers of the processor data path. A bus is a set of parallel wires. Bi-directional means that devices connected to the bus can either take input from the bus, or put output on the bus. In order for this to happen without outputs colliding, three-state buffers are used.

Recall that a computer consists of large networks of automatic switches, each of which is either on (logical 1, or 5V) or off (logical 0, or 0V). But remember, that in order to be either 5 or 0 volts, an output has to be connected, through the logic gate circuit, to either the 5V or 0V (ground) power supply leads of the gate. What would be the state of an output that was connected to neither? That is, pretend you cut the output wire, and it is hanging in the air.

For a logic circuit output to be useful, it must be able to switch between the two voltage states and pass current. That is, if an output is 5V, it needs to be able to pass some current to drive the inputs of the other gates to which it is connected. Similarly, if an output is 0V, it needs to be able to "sink" current, in order to operate the circuits it is tied to.

The "cut wire" state is a third state, neither 1 nor 0, that is very useful in computer system design. The third state is also called "high impedance" because current will not be able to flow either into or out of an output that is in this state. The high impedance state can also be thought of as having a very high resistance. It is shortened to "Hi Z", because Z is a symbol for impedence, a general term for resistance to current flow in both DC circuits like those of a computer, and AC circuits like those of a radio. Here are simple switch diagrams of a two-state and a three-state device.

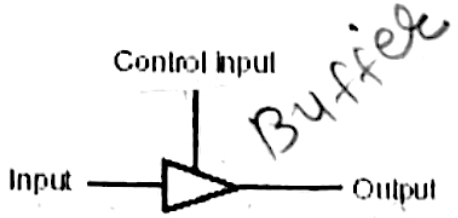


In the upper diagram, the switch can be open or closed. This type of switch is called a single-throw switch. If open, the output is connected through the resistor to ground, and will have a voltage of 0 with ability to sink some incoming current to ground through the resistor. When the switch is closed, the output will be 5V, with the ability to pass significant current to outside devices. The lower diagram shows a switch that can have three positions; closed to the 5V contact, open in the middle, or closed to the grounded contact. Such a switch is called a double-throw switch. Note that the circuit output is connected to the central pole of the switch. The middle position is the third state. It is just like a cut wire. The output, connected to the central pole of the switch, will not be able to conduct any current anywhere with the switch in the middle position and its voltage will be undefined. This is characteristic of the third, or high-impedance state.

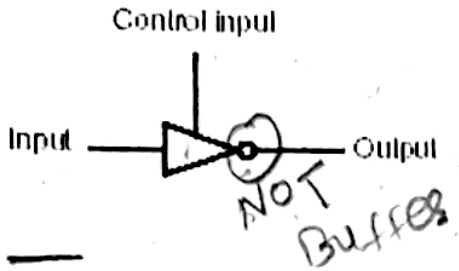
Having a three-state logic device output allows us to take a huge short cut when we assemble a true computer system. It allows us to use a single set of parallel wires to pass data in two directions.

The bi-directional bus allows us to connect many separate devices that have three-state outputs to the same set of wires, and then use logic to select which devices will communicate to each other. The outputs of the other unselected devices will remain in the third state, and will be invisible to the active circuits. They will not interfere with the data communications between the active devices. We are bringing this up here, because computer memory is such a device, with three-state data outputs. When the memory write signal is given to a selected memory chip, the outputs behave as data inputs, in order to write data in the memory. When the memory write signal is inactive, the outputs behave as outputs, sending data onto the bus. When the chip is not selected, the outputs are in the third, high-impedance state, and the chip is invisible to other devices on the bus, such as input or output ports. It is convenient to think of the memory data lines as input/output lines, because they can change direction.

It is a simple matter to make a bi-directional bus using three-state logical devices. The two most commonly used are the three-state buffer, and three-state inverting buffer.

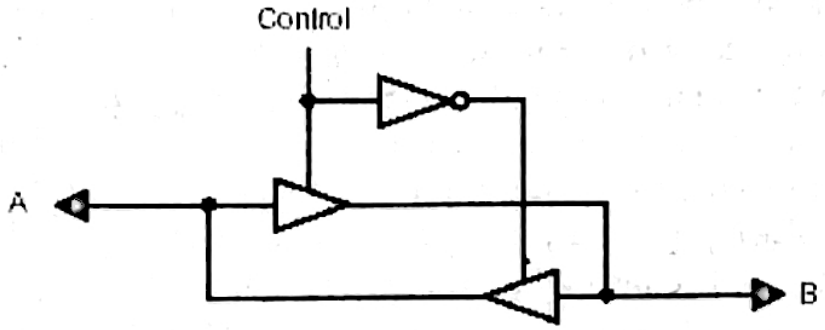


In	Control	Out
0	0	Hi Z
1	0	Hi Z
0	1	0
1	1	1



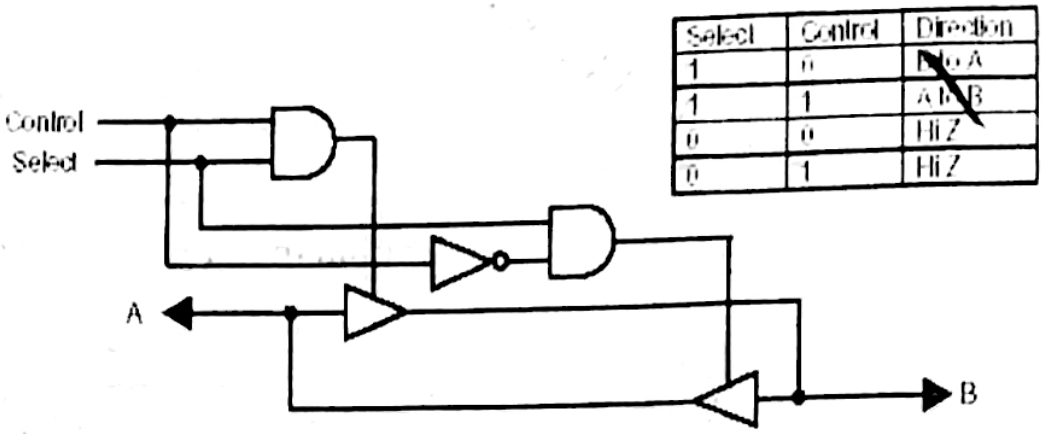
In	Control	Out
0	0	Hi Z
1	0	Hi Z
0	1	1
1	1	0

The control input is a signal that enables the gate to pass data when it is on (1). The control input is sometimes called an enable input. Here is how to make a bi-directional buffer out of two three-state buffers and an inverter.



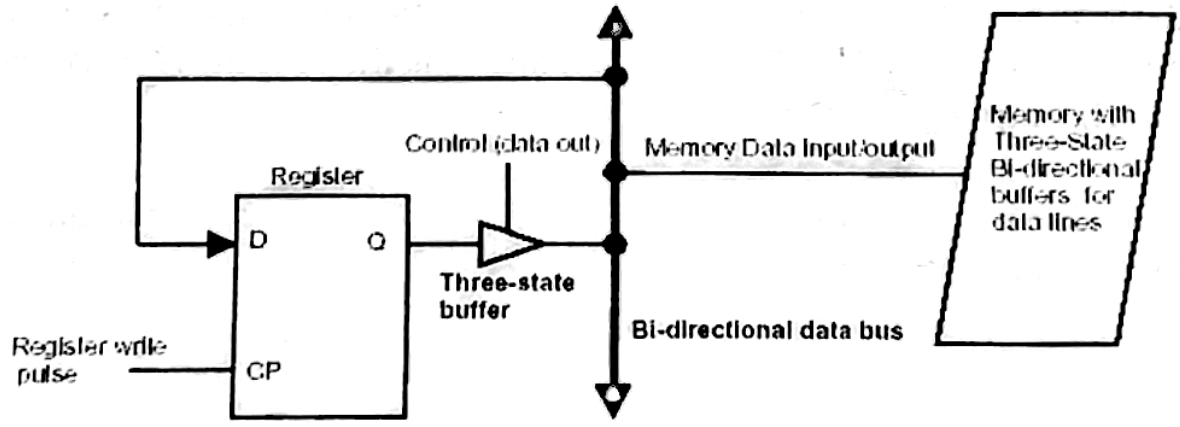
When control is 1, the data flows from A to B, and when it is 0, from B to A. In other words, when control is 1, A is the input and B is the output, and when control is 0, B is the input and A is the output. Thus A and B are input/outputs.

Inclusion of another control can allow the entire bi-directional buffer to go to the Hi Z state.



This type of circuit is used inside memory circuits to control the input/output lines.

In the data path of the homebuilt processor, we are faced with the problem of how to connect a register, with separate inputs and outputs, to a memory chip with bi-directional three-state input/outputs. First of all, the problem really only occurs during the memory write cycle, because otherwise the memory is either putting out data (if selected) or is in the high impedance state (if not selected). If the memory input/outputs are directly connected to register inputs, both of these conditions are tolerable. However, the register outputs cannot be connected directly to the memory input/outputs, because when the memory is in the output condition, you will have memory outputs connected to register outputs, which is not tolerable. We need to use a three-state buffer to prevent the register outputs from colliding with the memory outputs. The circuit is simple:



The three-state buffer protects the memory outputs from colliding with the register outputs when the memory is in the output mode. When the

memory is in the input mode, the Data out control signal comes on, and the three-state buffer becomes an output.

2.9.2 Memory Transfer

The internal bus connects only registers within the CPU, so how do we get data to and from memory?

The address register (AR) is used to select a memory address, and the data register (DR) is used to send and receive data. Both these registers are connected to the internal bus. DR is a bridge between the internal BUS and the memory data BUS.

Memory can also be connected directly to the internal BUS in theory.

Diagram showing connections to memory unit.

$M[AR] \leftarrow DR$

$DR \leftarrow M[AR]$

Hence, accessing memory outside the CPU requires at least two clock cycles. First we load AR with the desired memory address, and then transfer to or from DR. In most typical computer systems, memory transfers take many clock cycles, known as *wait states*

2.10 Arithmetic Micro-operations

Unlike register transfer microoperations,arithmetic microoperations change the information content.

- The basic arithmetic microoperations are:
  - addition
  - subtraction
  - increment
  - decrement
  - shift

The RTL statement:

$R3 \leftarrow R1 + R2$

indicates an add microoperation. We can similarly specify the other arithmetic microoperations.

- Multiplication and division are not considered microoperations.
  - Multiplication is implemented by a sequence of adds and shifts.
  - Division is implemented by a sequence of substracts and shifts.

Arithmetic Microoperations

$R1 \leftarrow R1 - 1$ Decrement content of R1 by 1
$R1 \leftarrow R1 + 1$ Increment content of R1 by 1
$R3 \leftarrow R1+R2 + 1$ R1 plus 2's comp. of R2
$R2 \leftarrow R2 + 1$ 2's complment contens of R2 (negate)
$R2 \leftarrow R2$ Complement contents of R2 (1's comp.)
$R3 \leftarrow R1 - R2$ Contents of R1 minus R2 transferred to R3
$R3 \leftarrow R1 + R2$ Contents of R1 plus R2 transferred to R3
Symbolic Description

2.10.1 Binary Adder

We implement a binary adder with registers to hold the data and a digital circuit to perform the addition (called a binary adder).

- The binary adders is constructed using full adders connected in cascade so that the carry produced by one full adder becomes an input for the next.
- Adding two n-bit numbers requires n full adders.
- The n data bits for A and B might come from R1 and R2 respectively

2.10.2 Binary Adder-Subtractor

Subtracting A – B is most easily done by adding B' to A and then adding 1.

- This makes it convenient to combine both addition and subtraction into one circuit, called an addersubtracter.

- M is the mode indicator

  - M = 0 indicates addition (B is left alone and C0is 0)
  - M = 1 indicates subtraction (B is complement and C0 is 1).

2.10.3 Binary Incrementer

The binary incrementer adds 1 to the contents of a register, e.g., a register storing 0101 would have 0110 in it after being incremented.

- There are times when we want incrementing done independent of a register. We can accomplish this with a series of cascading half-adders.

2.10.4 Arithmetic Circuit

We can implement 7 arithmetic microoperations (add, add with carry, subtract,subtract with borrow, increment, decrement and transfer) with one circuit.

- We provide a series of cascading full adders with Ai and the output of a 4x1 multiplexer.

  - The multiplexers' inputs are two selects, Bi, Bi ', logical 0 and logical 1.
  - Which of these four values we provide (together with the carry) determines which microoperation is performed.

2.11 Logic Micro-operations

In Logic operations, individual bits of registers are operated with other corresponding register bits.

Example logic operation.
Register A is 4 bits long = 0110

Register B is also 4 bits long with contents 1100.

Following is the list of logic microoperations that can be performed on the any two registers. Lets assume registers A and B for discussion.

2.11.1 List of Logic Micro-operations

The **logic micro-operation** is used for specifying binary operation for the strings of bits stored in the **register**. Here, each bit of **register** is treated as a separate binary variable. In other words, each individual bit of a **register** operates with corresponding **register** bit. Various logical operation like OR, AND, NAND, and other logical operators. Special symbols are used for representing OR and AND operators with v and ^ respectively.

Let us consider that there are two **register** R1 and R2, each with four bit. Let **register** R1 have 1010 and R2 have 1100. If OR micro-operation is performed on these **registers** and the result is stored in the **register** P, then this operation can be represented as:

P <- R1 + R2

The below table represents the AND operation on R1 and R2:

	Bit 3	Bit 2	Bit 1	Bit 0
A	1	0	1	0
B	1	1	0	0
A AND B	1	0	0	0

The below table lists all micro-operation, that can be performed on any two **registers**. Here, we will consider **register**R1 and R2 for representing these operations.

Name	Function	Description
AND	F = A*B	Bitwise AND
NAND	F = (A*B)′	Bitwise NAND
OR	F = A + B	Bitwise OR
NOR	F = (A + B)′	Bitwise NOR

Complement A	$F = A'$	Complement each bit of register A.
Complement B	$F = B'$	Complement each bit of register B.
XOR	$F = AB' + A'B$	Bitwise XOR
XNOR	$F = (AB + A'B')$	Bitwise XNOR
Transfer A	$F = A$	Unchanged content of register A.
Transfer B	$F = B$	Unchanged content of register B.
SET	$F = 1$	Set all bits.
CLEAR	$F = 0$	Clear all bits.
A AND (Comp B)	$F = A*B'$	
(Comp A) AND B	$F = A'*B$	
A OR (Comp B)	$F = A + B'$	
(Comp A) OR B	$F = A' + B$	

Symbolic representation	Description
R ← shr R	Logical Shift right R (register)
R ← shl R	Logical shift left R (register)
R ← ashl R	Arithmetic shift left R (register)
R ← ashr R	Arithmetic shift right R (register)
R ← cir R	Circular shift right R (register)
R ← cil R	Circular shift left R (register)

2.13 Arithmetic Logic Shift Unit

Arithmetic Shift Operation shifts signed (positive or negative) binary numbers either left or right by multiplying or dividing by 2. For, Arithmetic Shift left micro operation, the value in the register is multiplied by 2 and whereas for Arithmetic Shift right micro operation, the value in the register is divided by 2.

In RTL (RTL stands for Register Transfer Language), we can represent this arithmetic shift micro operations as

R ← ashl R (arithmetic shift left R (register))
R ← ashr R (arithmetic shift right R (register))
Diagram showing Arithmetic shift left operation is as follows:

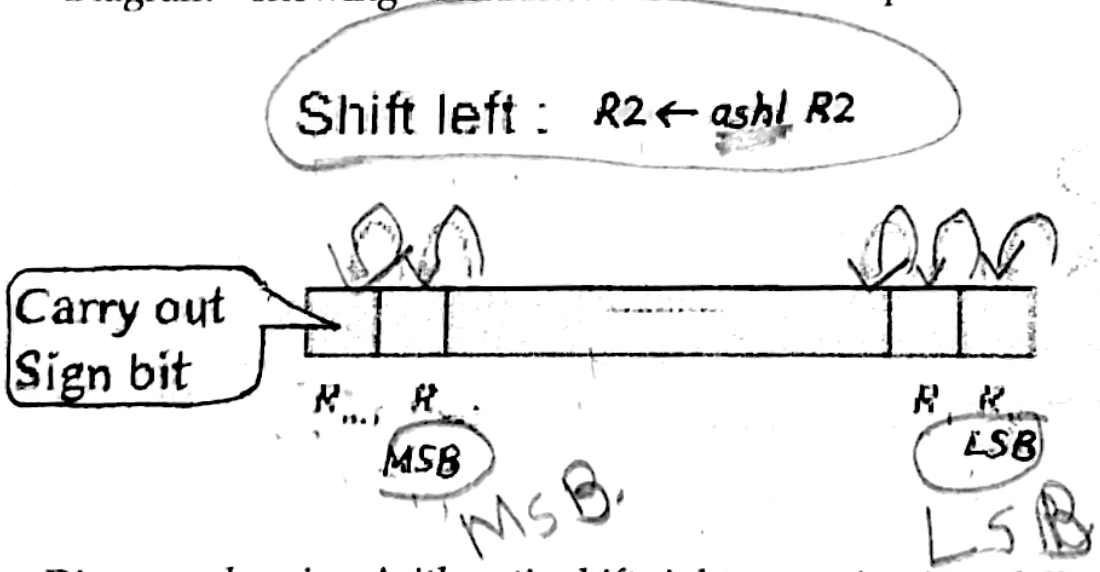
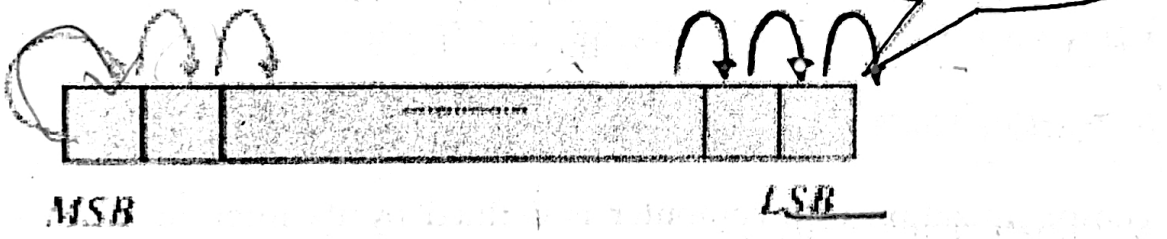


Diagram showing Arithmetic shift right operation is as follows:

Shift right : R2 ← ashr R2



***