**Date: 15-04-2021**

**Branch: CSE (III Year), EE (III Year) and MCA (I Year)**

**Topic: Micro Operation and Floating-Point Representation**

**Time: 08:00 AM -9:00 AM**

Register is a very fast computer memory, used to store data/instruction in-execution.

A **Register** is a group of flip-flops with each flip-flop capable of storing **one bit** of information. An *n-bit* register has a group of *n flip-flops* and is capable of storing binary information of *n-bits*.

A register consists of a group of flip-flops and gates. The flip-flops hold the binary information and gates control when and how new information is transferred into a register. Various types of registers are available commercially. The simplest register is one that consists of only flip-flops with no external gates.

These days registers are also implemented as a register file.

## Loading the Registers

The transfer of new information into a register is referred to as loading the register. If all the bits of register are loaded simultaneously with a common clock pulse than the loading is said to be done in parallel.

## 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.

## Register Transfer

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

**R2 ← 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.

## Control Function

A control function is a Boolean variable that is equal to 1 or 0. The control function is shown as:

**P: R2 ← R1**

The control condition is terminated with a colon. It shows that transfer operation can be executed only if P=1

## Micro-Operations

The operations executed on data stored in registers are called micro-operations. A micro-operation is an elementary operation performed on the information stored in one or more registers.

**Example:** Shift, count, clear and load.

**Types of Micro-Operations**

The micro-operations in digital computers are of 4 types:

1. Register transfer micro-operations transfer binary information from one register to another.
2. Arithmetic micro-operations perform arithmetic operations on numeric data stored in registers.
3. Logic micro-operations perform bit manipulation operation on non-numeric data stored in registers.
4. Shift micro-operations perform shift micro-operations performed on data.

### Arithmetic Micro-Operations

Some of the basic micro-operations are addition, subtraction, increment and decrement.

#### Add Micro-Operation

It is defined by the following statement:

*R3 → R1 + R2*

The above statement instructs the data or contents of register R1 to be added to data or content of register R2 and the sum should be transferred to register R3.

#### Subtract Micro-Operation

Let us again take an example:

*R3 → R1 + R2' + 1*

In subtract micro-operation, instead of using minus operator we take **1's compliment** and add 1 to the register which gets subtracted, i.e **R1 - R2** is equivalent to **R3 → R1 + R2' + 1**

**Increment/Decrement Micro-Operation**

Increment and decrement micro-operations are generally performed by adding and subtracting 1 to and from the register respectively.

*R1 → R1 + 1*

*R1 → R1 – 1*

|  |  |
| --- | --- |
| **Symbolic Designation** | **Description** |
| R3 ← R1 + R2 | Contents of R1+R2 transferred to R3. |
| R3 ← R1 - R2 | Contents of R1-R2 transferred to R3. |
| R2 ← (R2)' | Compliment the contents of R2. |
| R2 ← (R2)' + 1 | 2's compliment the contents of R2. |
| R3 ← R1 + (R2)' + 1 | R1 + the 2's compliment of R2 (subtraction). |
| R1 ← R1 + 1 | Increment the contents of R1 by 1. |
| R1 ← R1 - 1 | Decrement the contents of R1 by 1. |

#### Logic Micro-Operations

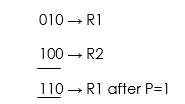
These are binary micro-operations performed on the bits stored in the registers. These operations consider each bit separately and treat them as binary variables.

Let us consider the X-OR micro-operation with the contents of two registers R1 and R2.

*P: R1 ← R1 X-OR R2*

In the above statement we have also included a Control Function.

Assume that each register has 3 bits. Let the content of R1 be **010** and R2 be **100**. The X-OR micro-operation will be:



#### Shift Micro-Operations

These are used for serial transfer of data. That means we can shift the contents of the register to the left or right. In the **shift left** operation the serial input transfers a bit to the right most position and in **shift right** operation the serial input transfers a bit to the left most position.

4=100 --🡪010 (2) 🡨--1000(8)

There are three types of shifts as follows:

**a) Logical Shift**

It transfers 0 through the serial input. The symbol **"shl"** is used for logical shift left and **"shr"** is used for logical shift right.

*R1 ← she R1*

*R1 ← she R1*

The register symbol must be same on both sides of arrows.

**b) Circular Shift**

This circulates or rotates the bits of register around the two ends without any loss of data or contents. In this, the serial output of the shift register is connected to its serial input. **"cil"** and **"cir"** is used for circular shift left and right respectively.

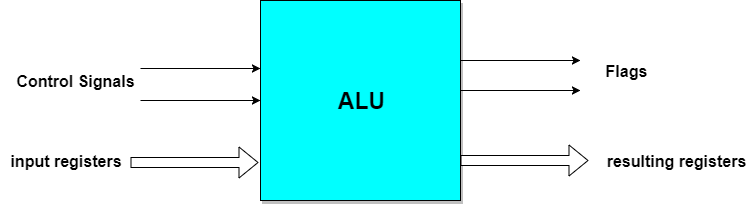
101-🡪R >>110 <<011

**c) Arithmetic Shift**

This shifts a signed binary number to left or right. An **arithmetic shift left** multiplies a signed binary number by 2 and **shift left** divides the number by 2. Arithmetic shift micro-operation leaves the sign bit unchanged because the signed number remains same when it is multiplied or divided by 2.

## Arithmetic Logical Unit

Instead of having individual registers performing the micro-operations, computer system provides a number of registers connected to a common unit called as Arithmetic Logical Unit (ALU). ALU is the main and one of the most important unit inisde CPU of computer. All the logical and mathematical operations of computer are performed here. The contents of specific register is placed in the in the input of ALU. ALU performs the given operation and then transfer it to the destination register.

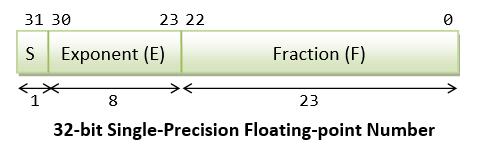


1.5 =0001.1

#### IEEE-754 32-bit Single-Precision Floating-Point Numbers

In 32-bit single-precision floating-point representation:

* The most significant bit is the sign bit (S), with 0 for positive numbers and 1 for negative numbers.
* The following 8 bits represent exponent (E).
* The remaining 23 bits represents fraction (F).



##### Normalized Form

Let's illustrate with an example, suppose that the 32-bit pattern is 1 1000 0001 011 0000 0000 0000 0000 0000, with:

* S = 1
* E = 1000 0001
* F = 011 0000 0000 0000 0000 0000

In the normalized form, the actual fraction is normalized with an implicit leading 1 in the form of 1.F. In this example, the actual fraction is 1.011 0000 0000 0000 0000 0000 = 1 + 1×2^-2 + 1×2^-3 = 1.375 D.

The sign bit represents the sign of the number, with S=0 for positive and S=1 for negative number. In this example with S=1, this is a negative number, i.e., -1.375D.

In normalized form, the actual exponent is E-127 (so-called excess-127 or bias-127). This is because we need to represent both positive and negative exponent. With an 8-bit E, ranging from 0 to 255, the excess-127 scheme could provide actual exponent of -127 to 128. In this example, E-127=129-127=2D.

Hence, the number represented is -1.375×2^2=-5.5D.

##### De-Normalized Form

Normalized form has a serious problem, with an implicit leading 1 for the fraction, it cannot represent the number zero! Convince yourself on this!

De-normalized form was devised to represent zero and other numbers.

For E=0, the numbers are in the de-normalized form. An implicit leading 0 (instead of 1) is used for the fraction; and the actual exponent is always -126. Hence, the number zero can be represented with E=0 and F=0 (because 0.0×2^-126=0).

We can also represent very small positive and negative numbers in de-normalized form with E=0. For example, if S=1, E=0, and F=011 0000 0000 0000 0000 0000. The actual fraction is 0.011=1×2^-2+1×2^-3=0.375D. Since S=1, it is a negative number. With E=0, the actual exponent is -126. Hence the number is -0.375×2^-126 = -4.4×10^-39, which is an extremely small negative number (close to zero).

##### Summary

In summary, the value (N) is calculated as follows:

* For 1 ≤ E ≤ 254, N = (-1)^S × 1.F × 2^(E-127). These numbers are in the so-called normalized form. The sign-bit represents the sign of the number. Fractional part (1.F) are normalized with an implicit leading 1. The exponent is bias (or in excess) of 127, so as to represent both positive and negative exponent. The range of exponent is -126 to +127.
* For E = 0, N = (-1)^S × 0.F × 2^(-126). These numbers are in the so-called denormalized form. The exponent of 2^-126 evaluates to a very small number. Denormalized form is needed to represent zero (with F=0 and E=0). It can also represents very small positive and negative number close to zero.
* For E = 255, it represents special values, such as ±INF (positive and negative infinity) and NaN (not a number). This is beyond the scope of this article.

**Example 1:** Suppose that IEEE-754 32-bit floating-point representation pattern is 0 10000000 110 0000 0000 0000 0000 0000.

Sign bit S = 0 ⇒ positive number

E = 1000 0000B = 128 D (in normalized form)

110 0000 0000 0000 0000 0000.

1+2^-1+2^-2

Fraction is 1.11B (with an implicit leading 1) = 1 + 1×2^-1 + 1×2^-2 = 1.75D

+F\*2^(E-127)=

The number is +1.75 × 2^(128-127) = +3.5D

**Example 2:** Suppose that IEEE-754 32-bit floating-point representation pattern is 1 01111110 100 0000 0000 0000 0000 0000.

Sign bit S = 1 ⇒ negative number

E = 0111 1110B = 126D (in normalized form)

Fraction is 1.1B (with an implicit leading 1) = 1 + 2^-1 = 1.5D

The number is -1.5 × 2^(126-127) = -0.75D

**Example 3:** Suppose that IEEE-754 32-bit floating-point representation pattern is 1 01111110 000 0000 0000 0000 0000 0001.

Sign bit S = 1 ⇒ negative number

E = 0111 1110B = 126D (in normalized form)

Fraction is 1.000 0000 0000 0000 0000 0001B (with an implicit leading 1) = 1 + 2^-23

The number is -(1 + 2^-23) × 2^(126-127) = -0.500000059604644775390625 (may not be exact in decimal!)

**Example 4 (De-Normalized Form):** Suppose that IEEE-754 32-bit floating-point representation pattern is 1 00000000 000 0000 0000 0000 0000 0001.

Sign bit S = 1 ⇒ negative number

E = 0 (in de-normalized form)

Fraction is 0.000 0000 0000 0000 0000 0001B (with an implicit leading 0) = 1×2^-23

The number is -2^-23 × 2^(-126) = -2×(-149) ≈ -1.4×10^-45

# Chapter 7 -- floating point arithmetic

about FLOATING POINT ARITHMETIC

-------------------------------

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 fl 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.

result is

0 10000101 10010001000000000000000

SUBTRACTION

like 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: use a mantissa that is only 4 bits so that

I don't spend all day just doing the multiplication

part.

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

becomes 10.00110000

add exponents: always add true exponents

(otherwise the bias gets added in twice)

biased:

10000100

+ 00111100

----------

10000100 01111111 (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

similar to multiplication.

true division:

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

subtract TRUE exponents

The IEEE standard is very specific about how all this is done.

Unfortunately, the hardware to do all this is pretty slow.

Some comparisons of approximate times:

2's complement integer add 1 time unit

fl. pt add 4 time units

fl. pt multiply 6 time units

fl. pt. divide 13 time units

There is a faster way to do division. Its called

division by reciprocal approximation. It takes about the same

time as a fl. pt. multiply. Unfortunately, the results are

not always the same as with true division.

Division by reciprocal approximation:

instead of doing a / b

they do a x 1/b.

figure out a reciprocal for b, and then use the fl. pt.

multiplication hardware.

example of a result that isn't the same as with true division.

true division: 3/3 = 1 (exactly)

reciprocal approx: 1/3 = .33333333

3 x .33333333 = .99999999, not 1

It is not always possible to get a perfectly accurate reciprocal.

ISSUES in floating point

note: this discussion only touches the surface of some issues that

people deal with. Entire courses could probably be taught on each

of the issues.

rounding

--------

arithmetic operations on fl. pt. values compute results that cannot

be represented in the given amount of precision. So, we must round

results.

There are MANY ways of rounding. They each have "correct" uses, and

exist for different reasons. The goal in a computation is to have the

computer round such that the end result is as "correct" as possible.

There are even arguments as to what is really correct.

3 methods of rounding:

round toward 0 -- also called truncation.

figure out how many bits (digits) are available. Take that many

bits (digits) for the result and throw away the rest.

This has the effect of making the value represented closer

to 0.

example:

.7783 if 3 decimal places available, .778

if 2 decimal places available, .77

round toward + infinity --

regardless of the value, round towards +infinity.

example:

1.23 if 2 decimal places, 1.3

-2.86 if 2 decimal places, -2.8

round toward - infinity --

regardless of the value, round towards -infinity.

example:

1.23 if 2 decimal places, 1.2

-2.86 if 2 decimal places, -2.9

in binary -- rounding to 2 digits after radix point

----------------------------------------------------

round toward + infinity --

1.1101

|

1.11 | 10.00

------

1.001

|

1.00 | 1.01

-----

round toward - infinity --

1.1101

|

1.11 | 10.00

------

1.001

|

1.00 | 1.01

-----

round toward zero (TRUNCATE) --

1.1101

|

1.11 | 10.00

------

1.001

|

1.00 | 1.01

-----

-1.1101

|

-10.00 | -1.11

------

-1.001

|

-1.01 | -1.00

-----

round toward nearest --

ODD CASE:

if there is anything other than 1000... to the right

of the number of digits to be kept, then

rounded in IEEE standard such that the least significant

bit (to be kept) is a zero.

1.1111

|

1.11 | 10.00

------

1.1101

|

1.11 | 10.00

------

1.001 (ODD CASE)

|

1.00 | 1.01

-----

-1.1101 (1/4 of the way between)

|

-10.00 | -1.11

------

-1.001 (ODD CASE)

|

-1.01 | -1.00

-----

NOTE: this is a bit different than the "round to nearest" algorithm

(for the "tie" case, .5) learned in elementary school for decimal numbers.

use of standards

----------------

--> allows all machines following the standard to exchange data

and to calculate the exact same results.

--> IEEE fl. pt. standard sets

parameters of data representation (# bits for mantissa vs. exponent)

--> Pentium architecture follows the standard

overflow and underflow

----------------------

Just as with integer arithmetic, floating point arithmetic operations

can cause overflow. Detection of overflow in fl. pt. comes by checking

exponents before/during normalization.

Once overflow has occurred, an infinity value can be represented and

propagated through a calculation.

Underflow occurs in fl. pt. representations when a number is

to small (close to 0) to be represented. (show number line!)

if a fl. pt. value cannot be normalized

(getting a 1 just to the left of the radix point would cause

the exponent field to be all 0's)

then underflow occurs.

HW vs. SW computing

-------------------

floating point operations can be done by hardware (circuitry)

or by software (program code).

-> a programmer won't know which is occuring, without prior knowledge

of the HW.

-> SW is much slower than HW. by approx. 1000 times.

A difficult (but good) exercize for students would be to design

a SW algorithm for doing fl. pt. addition using only integer

operations.

SW to do fl. pt. operations is tedious. It takes lots of shifting

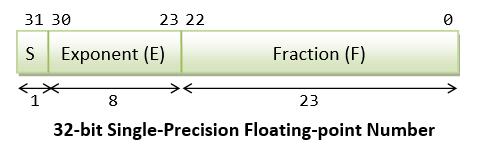
and masking to get the data in the right form to use integer arithmetic

operations to get a result -- and then more shifting and masking to put

the number back into fl. pt. format.

A common thing that manufacturers used to do is to offer 2 versions of the

same architecture, one with HW, and the other with SW fl. pt. ops.



(1259.125)10 =(10011101011.001)2

1259=10011101011

.125=001

122.25 =1.2225\*10^2

100=1\*102

**10011101011.001**

Normazize

1.0011101011001\*210

(1.N)2E-127

E-127=10🡪E=137

E= 10001001

0 10001001 0011101011001000000000