Booth's Algorithm

Booth's multiplication algorithm multiplies two signed binary numbers in two's complement notation. It examines adjacent pairs of bits of the N-bit multiplier Y in signed two's complement representation, including an implicit bit below the least significant bit, $y_{-1}=0$. For each bit y_i, the bits y_i and y_{i-1} are considered. Where these two bits are equal, the product accumulator P is unchanged. If $y_i=0$ and $y_{i-1}=1$, the multiplicand times 2^i is added to P. If $y_i=1$ and $y_{i-1}=0$, the multiplicand times 2^i is subtracted from P. The final value of P is the signed product.

The orders of steps is not determined. Typically, it proceeds from LSB to MSB, starting at $i=0$. The multiplication by

2^i is then typically replaced by incremental shifting of the P accumulator to the right between steps.

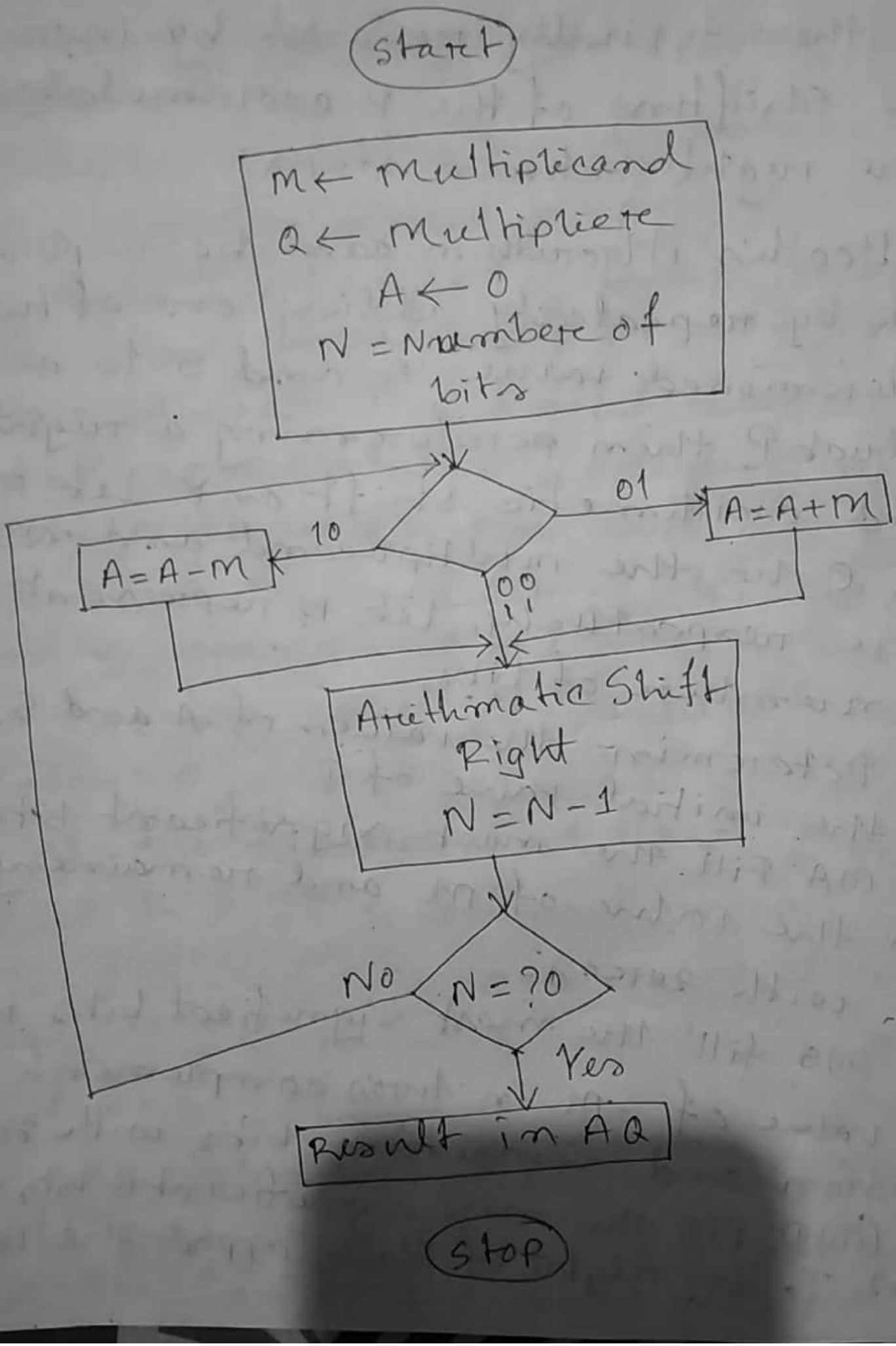
Booth's Algorithm can be implemented by repeatedly adding one of two predetermined values A and S to a product P, then performing a right-wared arithmetic shift on P. Let M and Q be the multiplicand and multiplier, respectively. Let, N represent the numbers of bits.

1. Determine the values of A and S, and the initial value of P.

(i) A: Fill the most significant bits with the value of M and remaining bits with zeros.

(ii) S: fill the most significat bits with the value of -M in two's complement notation and remaining bits with zeros.

(iii) P: Fill the most significant Q bits with zeros. To the right of this, appod the value


Start
M ← Multiplicand
Q ← Multiplicate
A ← 0
N = Number of bits
01
A=A+M
10
A=A-M
00
11
Arithmatic Shift Right
N = N - 1
NO
N = ?0
Yes
Result in AQ
Stop

of Q. Fill the least significant bit with a zero.

2. (i) If they are 01, find the value of P+A. Ignore any overflow.

(ii) If they are 10, find the value of P+S. Ignore any overflow.

(iii) If they are 00 or 11, use P directly in the next step.

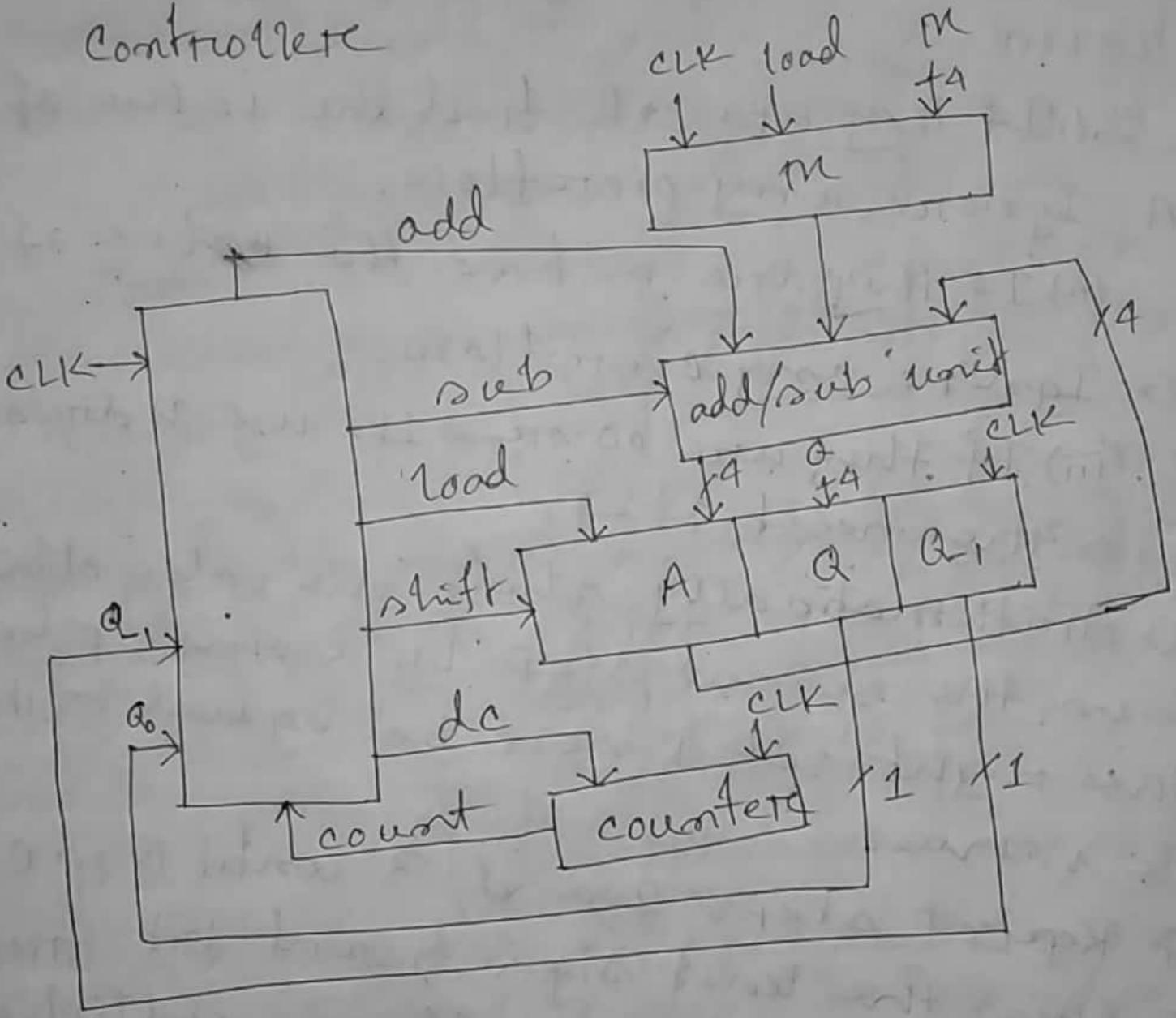
3. Arithmatically shift the value obtained in the second step by a single place to the right. Let P now be equal to its new value.

4. Repeat steps 2 and 3 until N is 0.

5. Drop the least significant bit from P. That is the result of M multiplied by Q.

Datapath

Controller



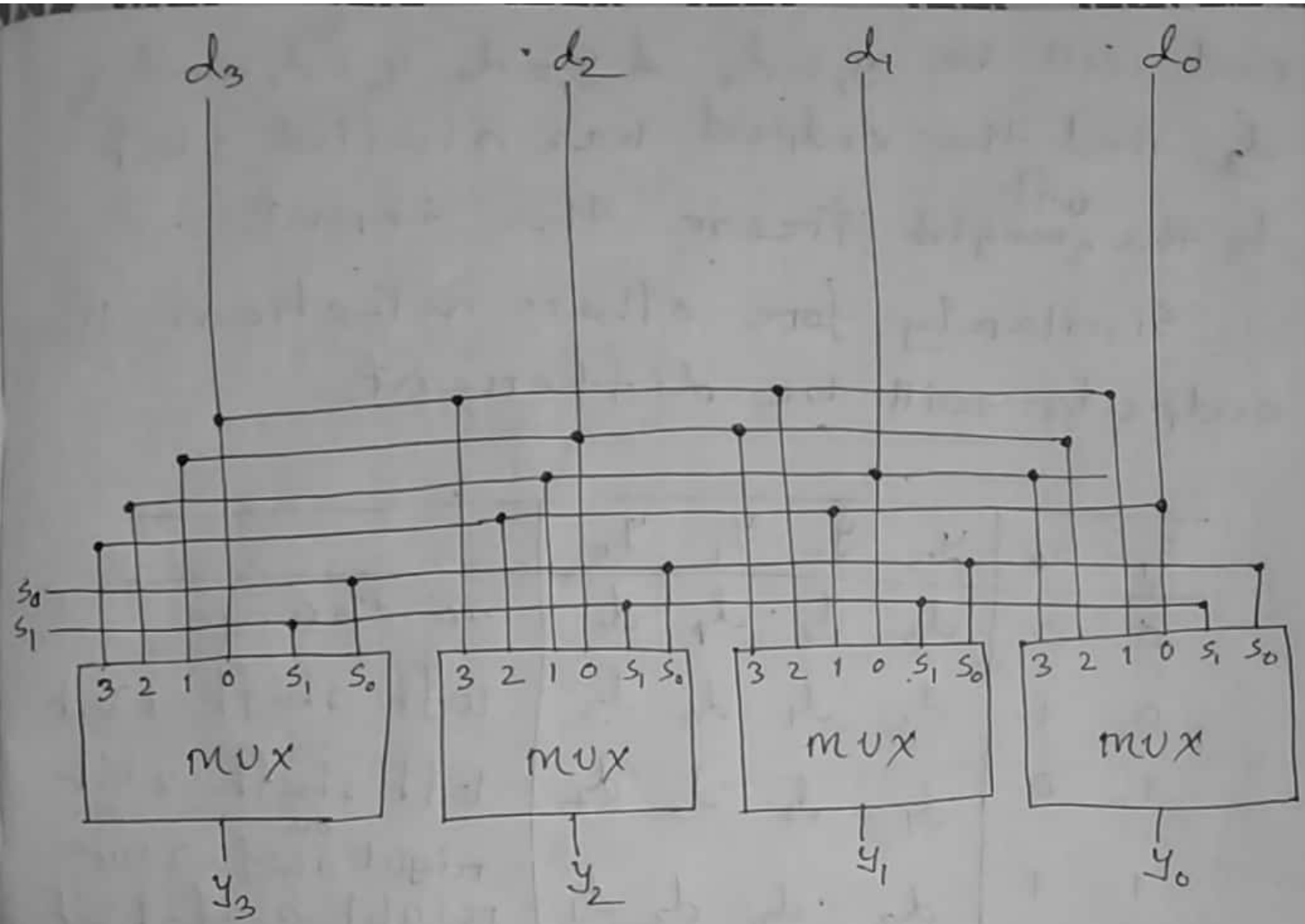
Barrel Shifters

A barrel shifter is a logic circuit for shifting a word by a varying amount. It has a control input that specifies the numbers of bit positions that it shifts by. It is implemented using a sequence of shift multiplexers. It can shift bits without the use of any sequential logic.

One way to implement it is a sequence of multiplexers where the output of one multiplexer is connected to the input of the next multiplexer in a way that depends on the shift distance.

To see how a barrel shifter works, let's see an example of 4-bit barrel shifter.

We have taken four 4-to-1 MUX for this purpose. The data bits are connected



to the input pins of all MUXs.

When S_0 and S_1 are both 0, all the MUXs will out the bit of the 0-th pin. So, $Y_0 = d_0$, $Y_1 = d_1$, $Y_2 = d_2$ and $Y_3 = d_3$. The output is in the same ordere that is in the input. so, no shifting has taken place.

When, $S_0 = 1$ and $S_1 = 0$, no. 1 pin of all the MUXs will be active. So, the out-

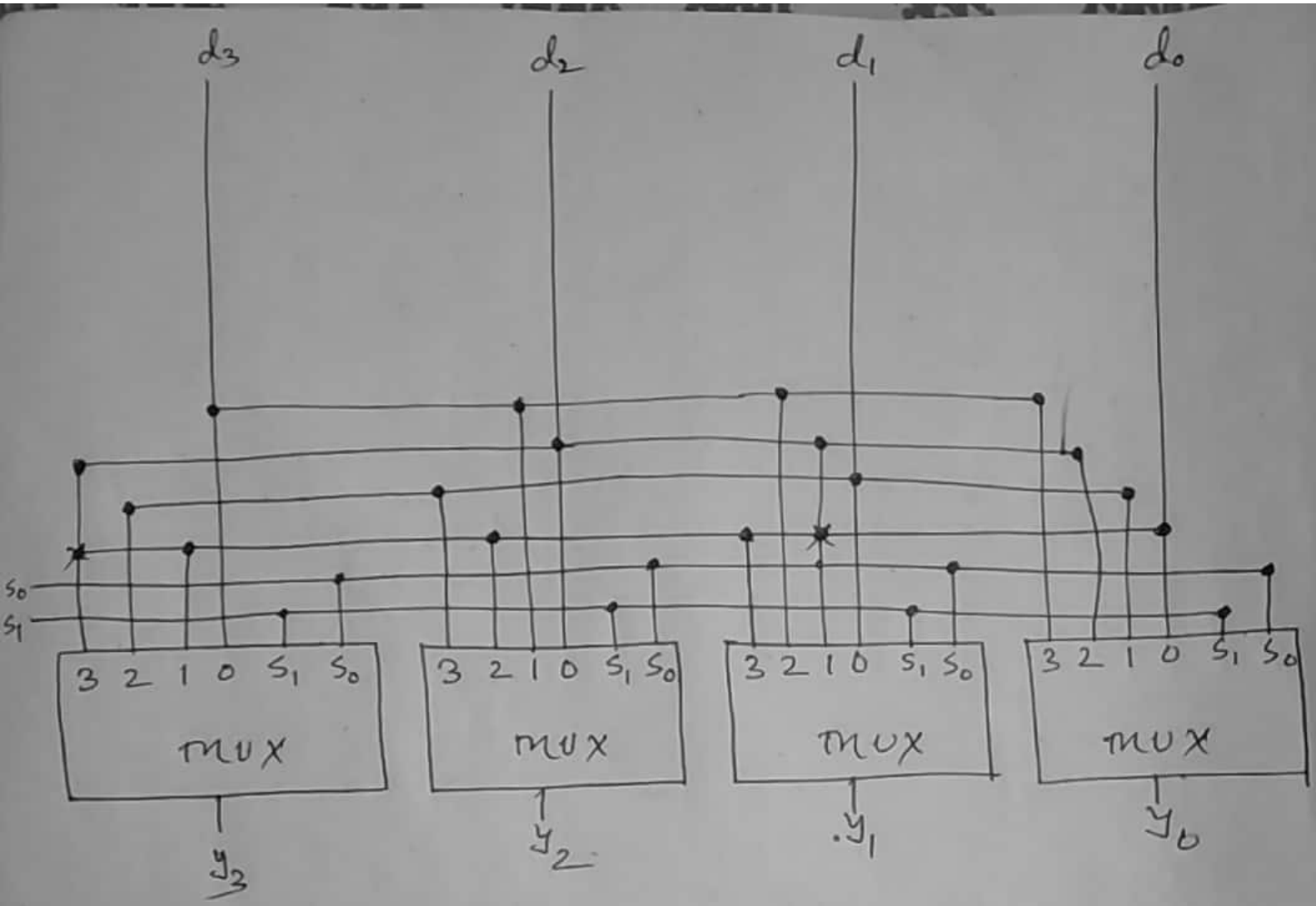
put will be $y_0 = d_3$, & $y_1 = d_0$ $y_2 = d_1$ and $y_3 = d_2$. And the output has shifted 1-bit to the ~~right~~ left from the input.

Similarly, fore othere selections, the outputs will be different.

S_1	S_0	y_3	y_2	y_1	y_0	
0	0	d_3	d_2	d_1	d_0	no change
0	1	d_2	d_1	d_0	d_3	left shift - 1 bit
1	0	d_1	d_0	d_3	d_2	left shift - 2 bit or, right shif - 2 bit
1	1	d_0	d_3	d_2	d_1	right shif - 1 bit

Truth table

With some minore changes to the connections, we can create anothere shiftere that will worek the opposite way thann this one.



S_1	S_0	y_3	y_2	y_1	y_0	
0	0	d_3	d_2	d_1	d_0	no change
0	1	d_0	d_3	d_2	d_1	right shift - 1 bit
1	0	d_1	d_0	d_3	d_2	right shift - 2 bit OR, left shift - 2 bit
1	1	d_2	d_1	d_0	d_3	left shift - 1 bit

Truth table

Memory Hierarchy

An economical solution to the desire of unlimited amounts of fast memory, is a memory hierarchy. It takes advantage of locality and trade-offs in the cost-perforemance of memory technologies.

The principle of locality says that most program do not access all code or data uniformly. Locality occurs in time and in space.

	capacity	Latency	cost/GB
Registere	1000s of bits	2ps	$$$$
SRAM	~10KB - 10MB	1 - 10ns	~$1000
DRAM	~10GB	80ns	~$10
Flash(SSD)	~100GB	100μs	~$1
Hard Disk	~1TB	10ms	~$0.10

Since fast memory is expensive, a memory hierarchy is organized into several levels - each smallere, fastere and more expensive the per byte than the

next lowere level, which is faire there from the processore. The goad is to provide a memory system with cost pere byte almost as low as the cheapest level of memory and speed almost as fast as the fastest level. A memory hierarcy can consist of multiple levels, but data is copied between only two adjacent levels at a time.

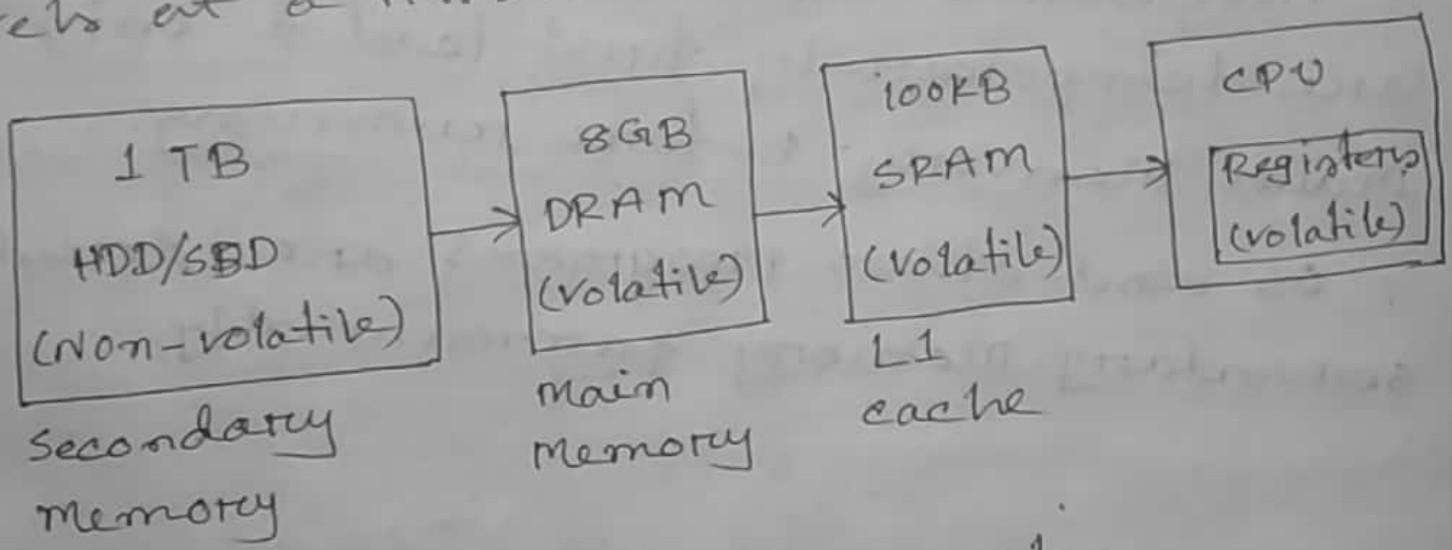


Fig: Memory Hierarchy

Volatile memory means it only contains data only when it is powered on. It loses its contents when powere is turned off. Whereas, non-volatile memory can contain data even after powere being turned off.

The machine transparently stores data in fast or slow memory, depending on usage patterns.

Data are stored in registers before they can be sent to ALU for operation.

Cache memory stores data that are frequently accessed.

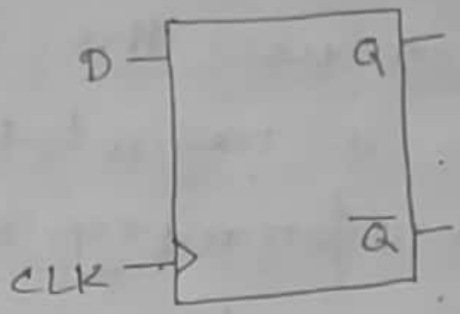
Programs instructions and data are temporarily first loaded into Main Memory before running.

OS and other programs are stored in secondary Memory paermanently.

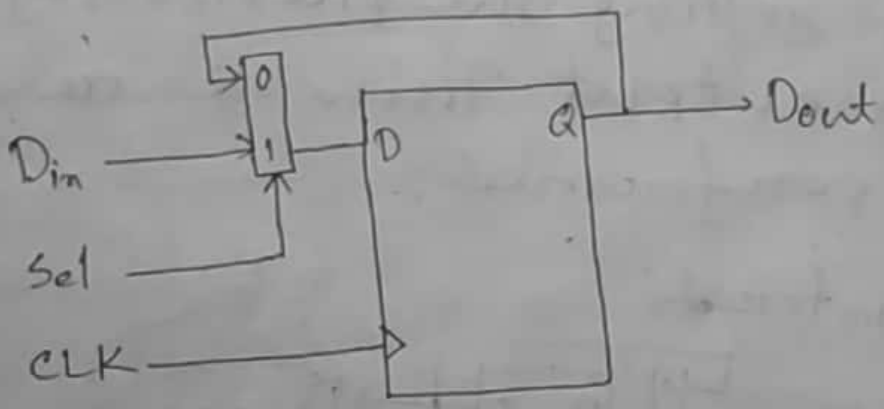
Registers

D Flip-flop: It updates its contents when there is a clock pulse. Otherwise it acts as memory.

CLK	D	Q(t+1)
0	X	Memory
1	0	0
1	1	1



1-bit Registers:



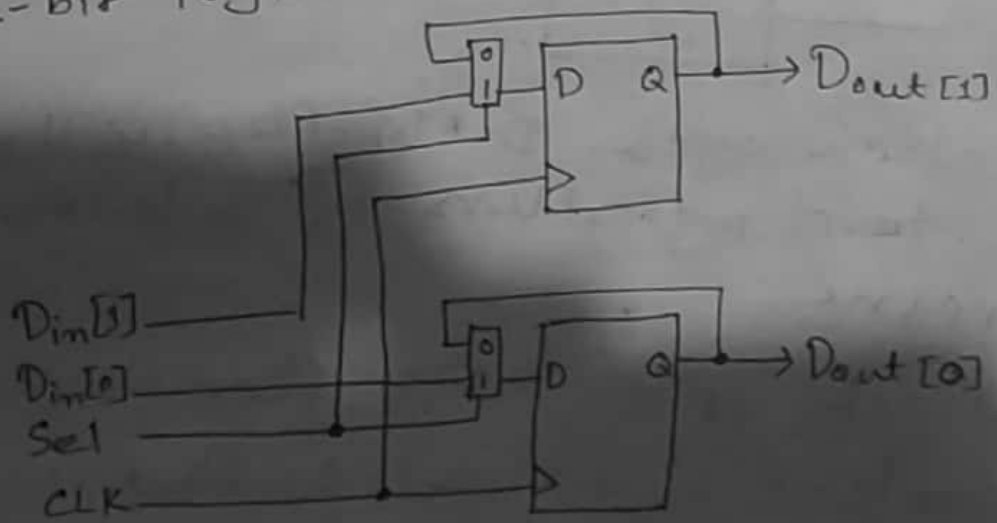
At negative clock cycle, D Flip-Flop will not update its content regardless of selection and input value.

At positive clock cycle, D Flip-Flop will update its content when Sel=0, Dout will

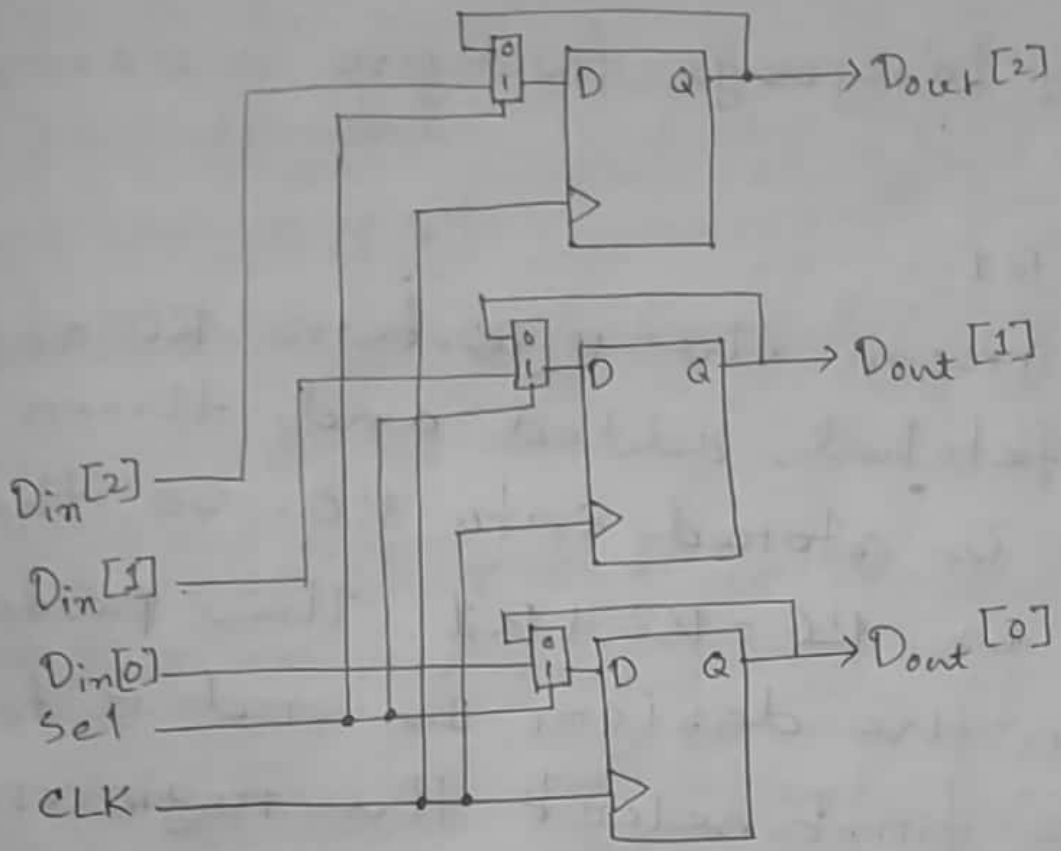
will be selected by the MUX. So, data will remain same in Flip-Flop. When Sel=1, the MUX will select D_{in} and data will get updated in Flip-Flop according to D_{in}.

We can use the selection to read or write a register. When Sel=1, we are performing a write operation. But, when Sel=0, although we are writing the same value again, we are also getting the previously stored value as output. Thus, a read operation is performed.

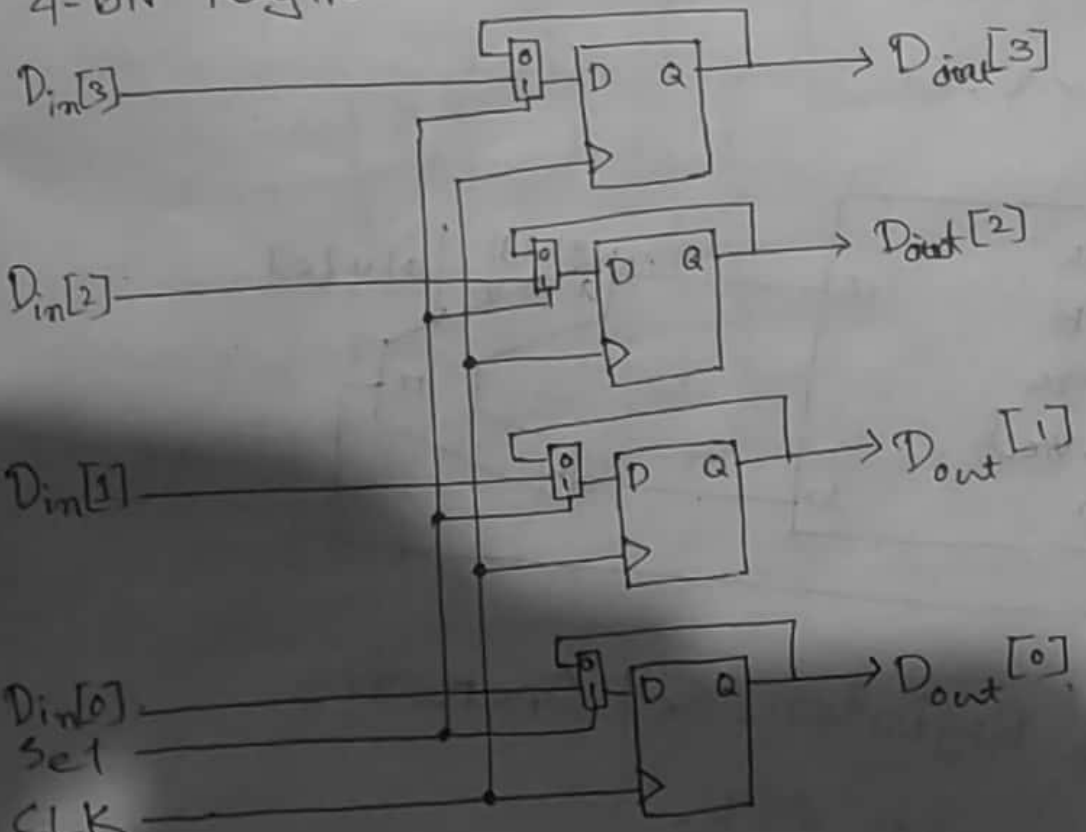
2-bit Registered:



3-bit Register:



4-bit Register:



Data Representation in Computer Systems

Decimal to Binary conversions: Fraction
al values can be approximated in all
base systems. Unlike integer values,
fractions do not necessarily have exact
representations under all radices.
Radix points seperate the integer
part of a number from its fractio-
nal part.

We can convert fractions between
different bases using methods analo-
gous to the repeated subtraction and
division remainder methods for conve-
rting integers.

Often things don't work out quite
so evenly, and we end up with repea-
ting fractions. The quantity $\frac{1}{2}$ is exact-
ly representable in the binary and
decimal systems, but is not in the

ternary (base 3) numbering system. The quantity 0.1 is exactly representable in the decimal system, but is not in the binary numbering system.

Fractional decimal values have non-zero digits to the right of the decimal point. Fractional values of other radix systems have non-zero digits to the right of the radix point. Numerals to the right of the radix point represent negative powers of the radix.

As with whole-number conversions, we can use either of two methods - a subtraction method and an easy multiplication method. The subtraction method for fractions is identical to the subtraction method for whole numbers. Instead of subtracting positive powers of the target radix we subtract negative powers of the

radix. We always start with the largest value first r^{-1}, where r is our radix, and work our way along using larger negative exponents. This method works with any base, not just binary.

$$
\begin{array}{rl}
0.8125 & \\
-0.5000 & = 2^{-1} \times 1 \\
\hline
0.3125 & \\
-0.2500 & = 2^{-2} \times 1 \\
\hline
0.0625 & \\
- \quad 0 & = 2^{-3} \times 0 \\
\hline
0.0625 & \\
-0.0625 & = 2^{-4} \times 1 \\
\hline
0 &
\end{array}
$$

$\therefore 0.8125_{10} = 0.1101_2$

In the multiplication method, we multiply the fraction part by 2, take the decimal part and keep multiplying the fraction part until the product becomes zero. This method also works with any base.

.8125
×2
1.6250
×2
1.2500
×2
0.5000
×2
1.0000

$\therefore 0.8125_{10} = 0.1101_2$

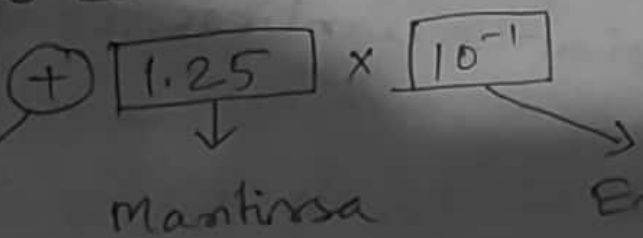
Floating Point Representation: The signed magnitude, one's complement and two's complement representation deal with integere values only. Without modifications, these foreonats are not useful in scientific ore business applications that deal with real numbere values. Floating point representation solves this problem.

We can perforem floating point calculations using any integere foreonat in floating point emulation. It is called like this because floating point values

aren't stored as such, we just create programs that make it seem as if floating point values are being used. Most of today's computers are equipped with specialized hardware that performs floating point arithmatic with no special programming required.

Floating-point numbers allow an arbitrary number of decimal places to the decimal point. They are often expressed in scientific notation.

computers use a form of scientific notation for floating point representation. In digital computers, floating point numbers consist of three parts: a sign bit, an exponent part (representing the exponent on a power of 2) and a fractional part called a significant (also called mantissa).



Computere representation of a floating point numbere consists three fixed sized fields;

1. Sign
2. Exponent
3. Significand

Sign	Exponent	Significand

This is the standared arrangement of these fields.

The one-bit sign field is the sign of the stored value.

The size of the exponent field, determines the range of values that can be represented.

The size of the significand ~~repre-sents~~ determines the precision of the representation.

In the hypothetical "simple model"

(i) A floating point number is 14 bits in length

(ii) The exponent field is 5 bits.

(iii) The significant field is 8 bits.

The significand of a floating point number is always preceded by an implied binary point. Thus, the significand always contains a fractional binary value. The exponent indicates the power of 2 to which the significand is raised.

Let's say, we wish to store the decimal # number 17 in this model. We know, $17 = 17.0 \times 10^0 = 0.17 \times 10^2$. In binary, $17_{10} = 10001 \times 2^0 = 0.10001 \times 2^5$. If we use this form, our fractional part will be 10001000 and our exponent will be 00101.

Using this form, we can store numbers of much greater magnitude than we could using a fixed point representation

of 14 bits.

Any number can be stored multiple representations using this model. These representations are synonymous, but they can cause confusion.

Another problem in this model is, there is no allowance for negative exponents. There is no way to express 0.5 (2^{-1}), since there is no sign in the exponent field.

To resolve the problem of synonymous forms, we will establish a rule that the first digit of the significand must be 1, with no ones to the left of the radix point. This process is called normalization. It results in a unique pattern for each floating number.

In this model, all significands must have the form 0.1xxxxxxxx.

For example, $4.5 = 100.1 \times 2^0 = 0.1001 \times 2^3$.
This is the normalized expression.

To provide for negative exponents, we will use a biased exponent. A bias is a number that is approximately midway in the range of values expressible by the exponent. We subtract the bias from the value in the exponent to determine its true value.

In our case, we have a 5-bit exponent.

$$2^{5-1} - 1 = 2^4 - 1 = 15$$

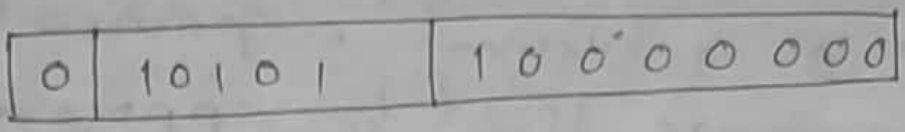
Thus, we will use 15 for our bias. Our exponent will use excess-15 representation. In this model, exponent values less than 15 are negative, representing fractional numbers.

For example, let's express 32_{10} in the revised 14-bit floating model.

$$32_{10} = 1.0 \times 2^5 = 0.1 \times 2^6$$

To use our excess-15 biased exponent we add 15 to 6, giving $21_{10} = (10101)_2$

So, we have,



The IEEE has established a standard for floating-point numbers. The IEEE-754 single precision floating point standard uses an ~~an~~ 8-bit exponent, with a bias of 127, and a 23 bit significand.

The IEEE-754 double precision standard uses an 11-bit exponent, with a bias of 1023, and a 52 bit significand.

In both IEEE single precision and double precision floating point standard, the significant has an implied 1 to the left of the radix point. The

format for significand using the IEEE format is 1.xxx...

For example, $4.5 = 0.1001 \times 2^3$ in IEEE format is. $4.5 = 1.001 \times 2^2$. The 1 is implied, which means it does not need to be listed in the significand.

Let's express -3.75 as a floating point numbers using IEEE single precision.

$-3.75 = -11.11_2 = -1.111 \times 2^1$

The bias is 127, so, $127 + 1 = 128 = 10000000$

1	10000000	11100000000000000000000

Since, we have an implied 1 in the significand, this equates to

$-1.111 \times 2^{128-127} = -1.111 \times 2^1 = -11.11 = -3.75$

Using the IEEE-754 single precision floating point standard:

i. an exponent of 255 indicates a special value.

ii. if the significand is zero, the value is ±infinity.

iii. if the significand is non-zero, the value is NaN (not a number), often used to flag an error condition.

Using the double precision standard

i. The special exponent value for a double precision number is 2047. Most FPUs use only the double precision standard.

Both the 14-bit model and the IEEE-754 floating point standard allow two representations for zero. Zero is indicated by all zeros in the exponent and the significand, but the sign bit can be either 0 or 1.

This is why it is recommended to avoid testing a floating-point value for equality to zero. Negative zero does not equal positive zero.

Floating-Point Arithmetic:

Floating point addition and subtraction are done using methods analogous to how we perform calculations using pencil and paper. The first thing that we do is express both operands in the same exponential power, then add the numbers, preserving the exponent in the sum. If the exponent requires adjustment, we do so at the end of the calculation.

For example, let's add 12_{10} and 1.25_{10} using 14-bit floating point model.

$$12_{10} = 1100 \times 2^0 = 0.1100 \times 2^4$$

$1.25_{10} = 1.01 \times 2^0 = 0.101 \times 2^1$

$= 0.000101 \times 2^4$

$\therefore$ 0.110000

0.000101

0.110101

Thus, the sum is 0.110101×2^4.

Floating point multiplication is also carried out in a manner akin to how we perform multiplication using pencil and paper. We multiply the two operands and add their exponents. If the exponent requires adjustment, we do so at the end of the calculation.

For example, let's multiply 12_{10} and 1.25_{10} using 14-bit floating point model.

$12_{10} = 0.1100 \times 2^4$ $1.25 = 0.101 \times 2^1$

Thus, the product is $0.01111 \times 2^5 = 0.1111 \times 2^4$.

The normalized product requires an exponent of $19_{10} = 10011_2$.

No matter how many bits we use in floating point representation, our model must be finite. The real number system is infinite, so our model can give nothing more than an approximation of a real value.

At some point, every model breaks down, introducing errors into our calculations. By using greater number of bits in our model, we can reduce these errors, but we can never totally illiminate them.

Our job becomes one of reducing error, or being at least aware of the possible magnitude of error in our calculations. We must be aware of the errors can compound through repetitive arithmatic operations.

For example, our 14-bit model cannot exactly represent the decimal value 128.5. In binary, it is 9 bit wide. $128.5_{10} = 10000000.1_2$

When we try to express 128.5_{10} in our 14-bit model, we lose the low-order bit, giving a relative error of $\frac{128.5 - 128}{128.5} \approx 0.39\%$

If we had a procedure that repeatedly added 0.5 to 128.5, we would have an error of nearly 2% after only 4 iterations.

Floating-point errors can be reduced when we use operands that are similar in magnitude. If we were repeatedly adding 0.5 to 128.5, it would have been better to relatively to add 0.5 to itself and then add 128.5 to

this sum. In this example, the error was caused by loss of low-order bit. Loss of the high order bit is more problematic.

Floating-point overflow and underflow can programs ~~to~~ crash. Overflow occurs when there is no room to store high order bits resulting from a calculation. Underflow occurs when the result is too small to store, possibly resulting in division by zero.

The range of a numeric integer format is the difference between the largest and the smallest values that can be expressed.

Accuracy refers to how closely a numeric representation approximates a true value. The precision of a number indicates how much informa-

tion we have about a value.

Most of the time, greatere precision leads to bettere accuteracy. but this is not always true. For example, 3.1333 is a value of pi that is accurate to two digits. but has 5 digits of precision.

There are otheie problems with floating point numbeirs. Because of truancated bits, we cannot always assume that a particulare floating point opereation is associative ore distributive. This means that we can not assume:

$$(a+b)+c = a+(b+c)$$

$$ax(b+c) = ab+bc$$

Moreoveie, to test a floating point value fore equality to some otheie numbeie, it is best to declare a "neariness to x" epsilon value. Fore

example, instead of checking to see if floating point x is equal to 2 as follows:

```
if (x == 2) : . . .
```

it is better to use

```
if (abs(x-2) < epsilon) . . .
```

assuming we have defined epsilon correctly.

Cache Memory

Cache is a small interim storage component that transparently retains or caches data from recently accessed locations. Cache memory is designed to to the memory access time of expensive, high-speed memory combined with the large memory size of less expensive, lower speed memory. If data is cached, access can be very fast. Otherwise, larger and slower cache or memory is accessed.

Computer systems often use multiple levels of caches. It is widely applied beyond hardware (e.g. web caches).

Cache Access: When a processor need to read to data, it sends the address to cache. There are possible scenar

ios that can happen:

i Cache hit: Data for this address is in cache.

ii Cache miss: Data is not in cache. Fetch data from memory and send it to the processor. Also retain the data in the cache replacing some other data.

Due to these cases, memory access time can differ. The processor needs to deal with this problem.

Cache Metrics:

$$\text{Hit ratio} = \frac{\text{hits}}{\text{hits} + \text{misses}}$$

$$\text{Miss ratio, MR} = \frac{\text{misses}}{\text{hits} + \text{misses}}$$

$$HR = 1 - MR$$

$$MR = 1 - HR$$

Average Memory Access Time (AMAT) = Hit Time +

Miss Ratio x Miss Penalty

The goal of caching is to improve average memory access time. This formula can be applied recursively in multi-level hierarchies, such as-

$$AMAT = Hit\ Time_{L_1} + Miss\ Ratio_{L_1} \times AMAT_{L2}$$

$$= Hit\ time_{L_1} + Miss\ Ratio_{L_1} \times (Hit\ time_{L2} + Miss\ Ratio_{L_2} \times AMAT_{L3}) *$$

$$= \ldots$$

The idea is to have greatere hit ratio to have lowere average memory access time.

Basic Cach Algorithm:

1 On reference to Mem[x], look for x among cache tags.

2 If there is a hit, read the data and change that data if necessary

3. If X is not found in TAG of any cache line i.e. a Miss, select some line k to hold Mem[X] for replacement selection. Read the data, set TAG for that data and store that data into the allocated memory location.

Direct Mapped Caches: Each word in memory maps into single cache line.
To access a cache with 2^n lines -

1. Index into cache with n-address bits (the index bits)
2. Read out valid bit, tag and data.
3. If valid bit is 1 and tag matches upper address bits, then we have a Hit.

Part of the address (index bits) is encoded in the location. Tag and Index bits unambiguously identify the data's address.

Block Size: To take advantage of locality. we increase block size. Another advantage is, it reduces size of tag memory. But a potential disadvantage is there would be fewer blocks in the cache.

Block Size Tradeoffs: Larger block sizes take advantage of spatial locality. It incurs larger miss penalty since it takes longer to transfere the block into the cache. It can increase the average hit time and miss rate.

Fully Associative Cache: In this cache any address can be in any location. So, there is no need for cache index. It also flexible since there is no conflict misses. But this cache is expensive as it has to compare tags of all

entries in parallel to find matching one. This can be implemented in hardware. It is called a CAM.

N-way Set-Associative Cache: It compromises between direct mapped fa and fully associative cache. It compares all tags from all ways in parallel. An N-way cache can be seen as N direct mapped caches in parallel. Direct mapped and fully-associative caches are just special cases of N-way set-associative caches.