Computer Architecture and Organization

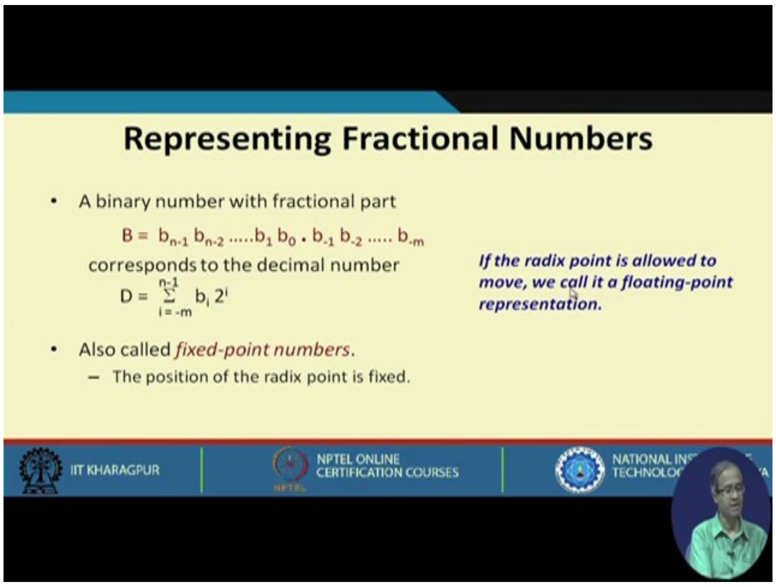
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 38
Floating - Point Numbers

In the last few lectures we have seen various arithmetic algorithms and techniques, using which we can carry out various operations on integer numbers. Specifically we looked at the design of adders, subtractors, multipliers and also dividers.

There we had seen basically how operations on integer quantities can be carried out. Today we shall be extending that concept and in this lecture we shall be specifically talking about something called floating-point numbers where we can handle fractional quantities as well. So, the topic of this lecture is floating point numbers.

(Refer Slide Time: 01:13)



Representing Fractional Numbers

- A binary number with fractional part
  $B = b_{n-1}\, b_{n-2} \ldots b_1\, b_0 \,.\, b_{-1}\, b_{-2} \ldots b_{-m}$
  corresponds to the decimal number
  $D = \sum_{i=-m}^{n-1} b_i 2^i$
- Also called *fixed-point numbers.*
  - The position of the radix point is fixed.

If the radix point is allowed to move, we call it a floating-point representation.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

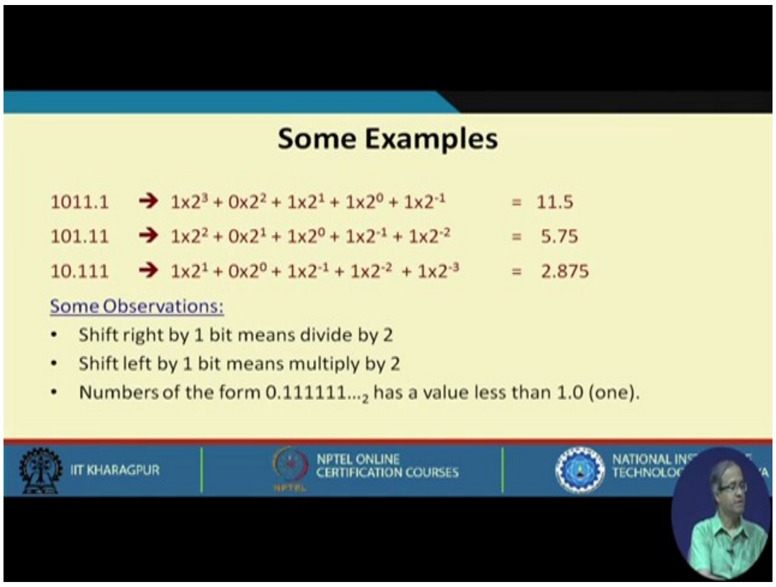
Let us see the motivation first; well here as we said we want to represent a binary number with a fractional part. In general any binary number with a fractional part can be represented like this where these b0, b1, are the binary digits; this dot is the binary radix point.

So, this side you have the integer part and this side you have the fractional part. If you want to convert this binary number into equivalent decimal, you recall that every binary position has a weight. b0 has a weight of 2 to the power 0, b1 as a weight of 2 to the power 1, and so on. Similarly on the fractional side b -1 has a weight of 2 to the power -1, b -2 has a weight of 2 to the power -2, and so on.

So, you can write the decimal equivalent of this number as this the digit multiplied by the weight 2 to the power i; where i can vary from -m up to n-1. This kind of a representation is also called fixed point numbers because the position of this radix point is fixed in the number representation, but if we allow this radix point to move (not fixed in a particular position), then we refer to this number representation as a floating point number representation.

This is the basic idea behind the so called floating point number representation. Let us look at some examples.

(Refer Slide Time: 03:02)



Some Examples

1011.1 → $1\text{x}2^3 + 0\text{x}2^2 + 1\text{x}2^1 + 1\text{x}2^0 + 1\text{x}2^{-1}$ = 11.5

101.11 → $1\text{x}2^2 + 0\text{x}2^1 + 1\text{x}2^0 + 1\text{x}2^{-1} + 1\text{x}2^{-2}$ = 5.75

10.111 → $1\text{x}2^1 + 0\text{x}2^0 + 1\text{x}2^{-1} + 1\text{x}2^{-2} + 1\text{x}2^{-3}$ = 2.875

Some Observations:

- Shift right by 1 bit means divide by 2
- Shift left by 1 bit means multiply by 2
- Numbers of the form $0.111111..._2$ has a value less than 1.0 (one).

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL IN TECHNOLO

You look at a binary numbers with fractional part, 1011.1. If you want to converted into decimal you will have to multiply each digit position by its weight 2 to the power 3, 2 to the power 2, 2 to the power 1, 2 to the power 0, and for the fractional part 2 to the power -1. So, the value becomes 11.5.

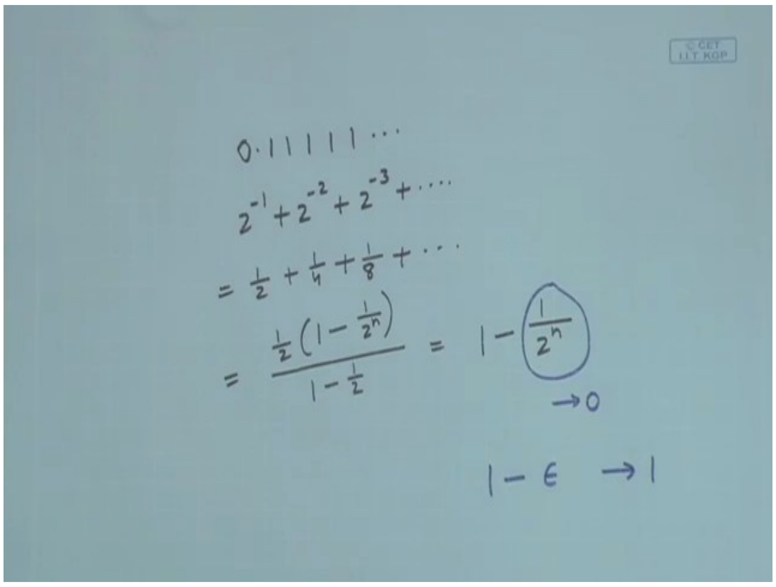
Similarly this number 101.11 we will have a value again multiplied by the weights; this will be 5.75 .

If you take a number 10.111, the weights will be this.

The value will be 2.875. Now one thing you see across the examples I have taken that the decimal point is actually shifting left by one position from this number to this, this number to this. So, when the decimal point or the radix point shifts left by one position if you look at the corresponding value, you see that this means a division by 2. 11.5, if you divide by 2, you get 5.75. 5.75, if you divide by 2, you get 2.875.

So, shifting right by 1 bit; that means, if you move this radix point to the left or right this will mean divide by 2, or multiply by 2. If you move the radix point to the left it will be divide by 2;is if you move the point to the right it will be multiply by 2; and another point to notice that if you have a number of this form 0.11111 this will have a value less than 1.

(Refer Slide Time: 05:19)



$$0.11111\cdots$$

$$2^{-1}+2^{-2}+2^{-3}+\cdots$$

$$=\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots$$

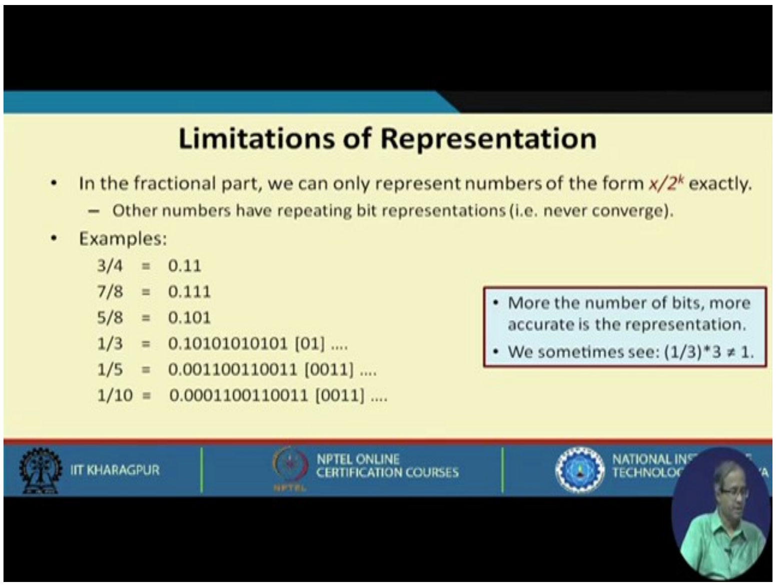
$$=\frac{\frac{1}{2}\left(1-\frac{1}{2^n}\right)}{1-\frac{1}{2}}=1-\underset{\to 0}{\frac{1}{2^n}}$$

$$1-\epsilon \to 1$$

Why this, you see if I write such a number 0.111… say there are several binary digits. If you look at the weights it is 2 to the power minus 1, then 2 to the power minus 2, then 2 to the power minus 3, and so on. You see this is a GP series; this is actually half plus one fourth, plus one 8th plus like this. So, if you calculate the sum of this series we get 1 minus 1 by 2 to the power n, where n is the number of digits.

You see as the value of n increases, as we increase this number of digits 1 1 1 1 1, the value of this second quantity tends to 0 because 2 to the power n increases rapidly. So, this is a fraction; you can write as 1 minus a very small value epsilon. This actually tends to 1 as n tends to infinity. This is exactly what we are meaning here; a number of this form we will always have a value less than 1, in the limit it will approach 1.

(Refer Slide Time: 06:57)



Limitations of Representation

- In the fractional part, we can only represent numbers of the form $x/2^k$ exactly.
  - Other numbers have repeating bit representations (i.e. never converge).
- Examples:

3/4	=	0.11
7/8	=	0.111
5/8	=	0.101
1/3	=	0.10101010101 [01] ....
1/5	=	0.001100110011 [0011] ....
1/10	=	0.0001100110011 [0011] ....

- More the number of bits, more accurate is the representation.
- We sometimes see: $(1/3)*3 \neq 1$.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INSTITUTE OF TECHNOLOGY

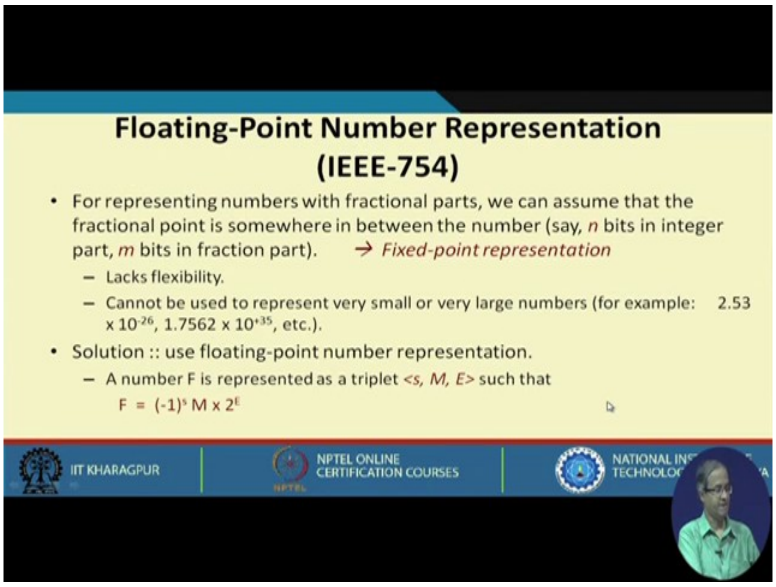
Now, in the representation there are some limitations let us look at this now. In the fractional representation in binary if you look little carefully you will see that only numbers which can be represented in the form x divided by 2 to the power k, where the denominator is some power of 2, those numbers can be precisely represented in binary; like 3/4 where 4 is a power of 2 --- you can express as 0.11.

7/8, 8 is a power of 2 as 0.111. 5/8 as 0.101, but if the denominator is not a power of 2 like here if I take examples of 3, 5 and 10. So, if you go on expanding decimal to binary you will that it will never terminate. So, this 1/3 this 1 0 1 0 1 0 alternates. 1/5 uses 0 0 1 1 0 0 1 1 alternates. 1/10 also you will see after some point 0 0 1 1 0 0 1 1 keeps alternating. So, more the number of bits you use for the representation more accurate will be our representation, but one thing we should also remember that when you are representing 1/3 in a computer like this, we have a finite number of bits to represent. So, we can never represent 1/3 exactly, it will be very close to 1/3.

The point is that you are not able to represent some of the fractions in a precise way; there is an error in the representation. By virtue of that what might happen is that suppose in a program you are diving 1/3 and the result you are again multiplying by 3. You expect that the final result should be 1, but you might see that the final result is not

1, but close to 1; and this is because of this truncation error. You can see we have finite number of bits to represent; this 1/3 you cannot represent precisely.

(Refer Slide Time: 09:22)



Floating-Point Number Representation (IEEE-754)

- For representing numbers with fractional parts, we can assume that the fractional point is somewhere in between the number (say, *n* bits in integer part, *m* bits in fraction part). → *Fixed-point representation*
  - Lacks flexibility.
  - Cannot be used to represent very small or very large numbers (for example: 2.53 x 10^{-26}, 1.7562 x 10^{+35}, etc.).
- Solution :: use floating-point number representation.
  - A number F is represented as a triplet *<s, M, E>* such that F = $(-1)^s$ M x 2^E

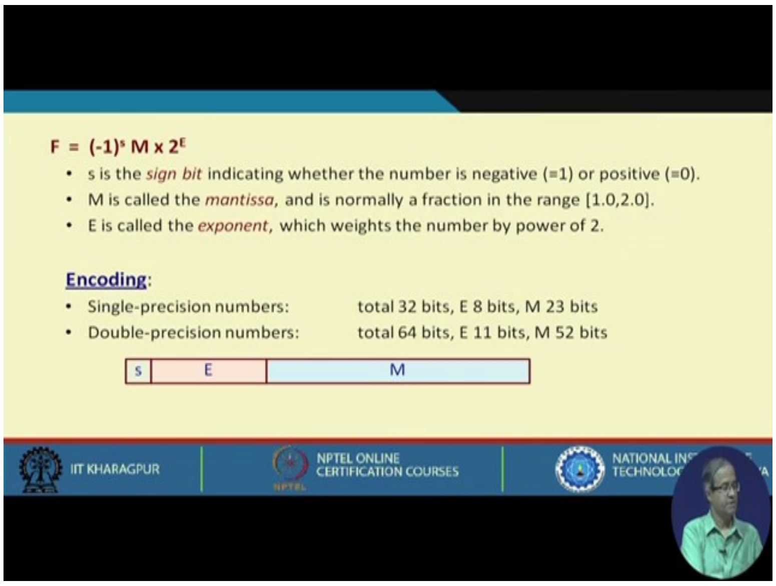
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INSTITUTE OF TECHNOLOGY

Now, let us come to the so called floating point number representation, which is around for quite some time. IEEE has come up with the standard. So, whatever we shall be discussing it will be basically based on this standard.

Let us try to see what this standard says. Firstly, we already talked about the fixed point representation earlier. So, there we had said that if we have a number with a fractional part, then we can have n bits in the integer part and m bits in the fractional part. So, the radix point is fixed, but in this representation we have limited amount of flexibility. Why? Because we cannot represent for instance very small or very large numbers, which occur very frequently in scientific computations. Like for instance I want to represent number 2.53 x 10^{-26} or 1.7562 x 10^{+35}, these kind of numbers where either the number is very large or number is very small, after the radix point there are many zeros, after that only the significant digits will start. So, these kinds of numbers you cannot represent in a finite number of bits, where the radix point is fixed in a location. For this you need something else.

So, what is the solution? The solution is to use this so called floating point number representation. Here a number F is represented as a triplet < S, M, E >, but the value of the number is minus 1 to the power S into M multiplied by 2 to the power E.

(Refer Slide Time: 11:39)


$F = (-1)^s M \times 2^E$

- s is the *sign bit* indicating whether the number is negative (=1) or positive (=0).
- M is called the *mantissa*, and is normally a fraction in the range [1.0,2.0].
- E is called the *exponent*, which weights the number by power of 2.

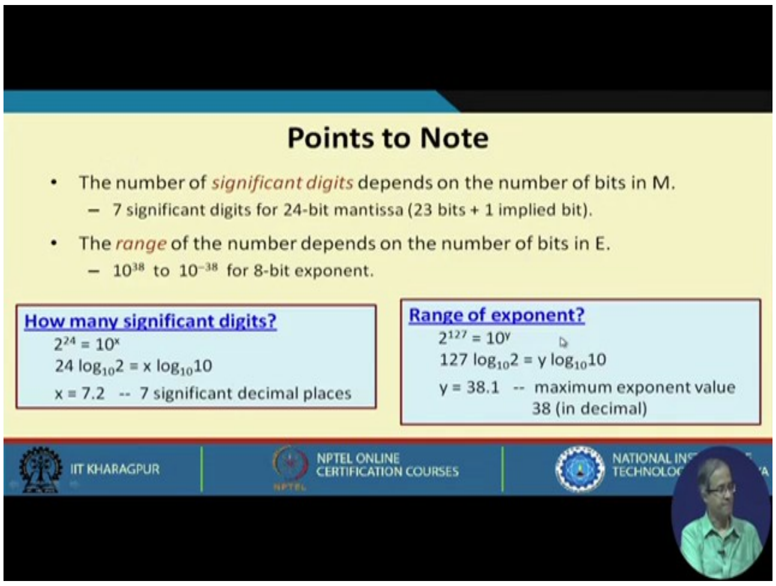
Encoding:

• Single-precision numbers:	total 32 bits, E 8 bits, M 23 bits
• Double-precision numbers:	total 64 bits, E 11 bits, M 52 bits

Coming again to this representation as I said s is the sign bit. This indicates whether the number is negative (for that s is 1) or it is a positive (s is 0). This M is conventionally known as the mantissa and I will explain a little later why this mantissa is usually a fraction, which is in the range 1.0 to 2.0.

And E is called the exponent, which is weighted as you can see by a power of 2. 2 to the power E; and in the IEEE floating point format you can have either single precision numbers or double precision numbers. The general format looks like this. The number starts with the sign bit, followed by some number of bits for the exponent, and finally some bits for the mantissa. For single precision numbers the total size is 32 bits, for E you have 8 bits and for M you have 23 bits.

(Refer Slide Time: 13:09)



Points to Note

- The number of *significant digits* depends on the number of bits in M.
  - 7 significant digits for 24-bit mantissa (23 bits + 1 implied bit).
- The *range* of the number depends on the number of bits in E.
  - 10^{38} to 10^{-38} for 8-bit exponent.

How many significant digits?

$2^{24} = 10^x$

$24 \log_{10}2 = x \log_{10}10$

$x = 7.2$ -- 7 significant decimal places

Range of exponent?

$2^{127} = 10^y$

$127 \log_{10}2 = y \log_{10}10$

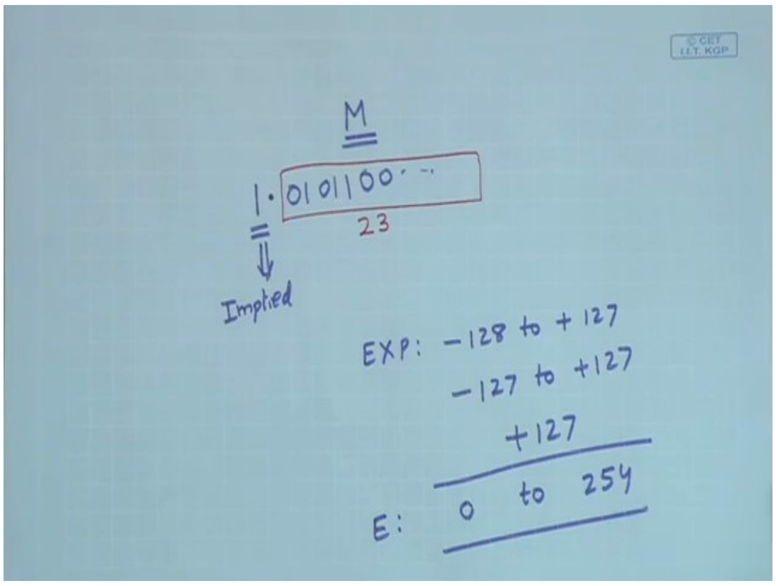
$y = 38.1$ -- maximum exponent value 38 (in decimal)

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY

But for double precision numbers you have 64 bits in total, where E is 11 bits and M is 52 bits. Now just going back once the number of significant digits in your representation will depend on how many bits you are using for the mantissa. This is the number of significant digits, it will depend on how many bits you have in M.

Now, although for single precision number we have kept 23 bits in the mantissa, but actually the idea is as follows. The mantissa is represented as a number that always starts with a 1.

(Refer Slide Time: 13:44)



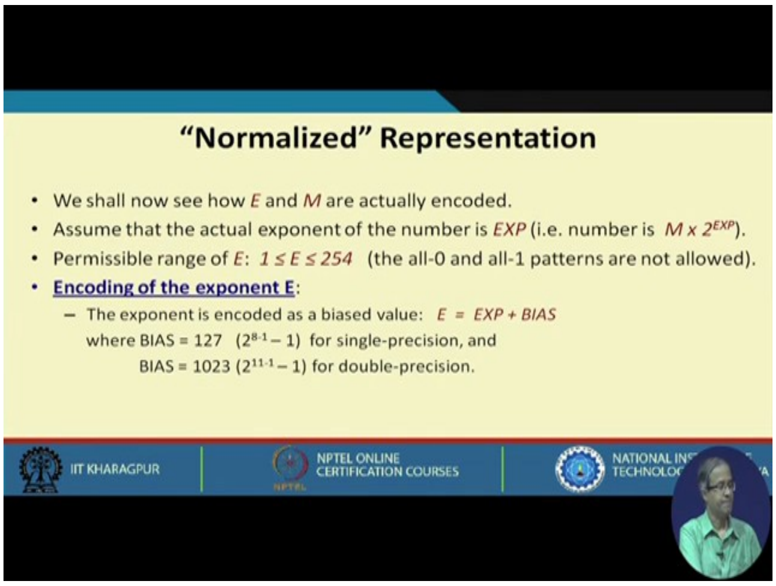
And there is a implied radix point here and after that you can have anything, 0 1 0 1 1. The first digit is always 1. We assume that this is an implied bit and because it is always 1 you do not store this; you store only the remaining number of bits. So, here for single precision there are 23 bits here, but if you also count this 1 it becomes 24. So, the number actually is a 24-bit number, but because it always starts with a 1 you are actually storing 23 bits. So, when you are calculating you will be assuming 24-bit mantissa because that implied 1 bit is also there. Now how do you calculate number of significant digits is very simple. In binary you have 24 bits, in decimal how many bits?

You use this; this equation 2 to the power 24 equal 10 to the power x. You take logarithm on both sides, and get $x = 7.2$. This means in decimal you can have 7 significant digits, but if you think of double precision number where for mantissa we have 52 bits. 52 plus 1 will be 53; 53 multiplied by log 2. So, it will be much larger --- 15 or 16 digits you will have for significant digits.

Similarly for exponent in single precision number you have 8 bits, and this exponent can be either positive or negative 2 to the power plus something or 2 to the power minus something. This 8 bit is actually you can regard it that is the 2's compliment signed integer; range will be -128 to +127.

If you want to find out the range in decimal you follow a similar calculation, 2 to the power 127 is equal to 10 to the power y. Finally y comes to about 38 point something. So, in decimal equivalently you can have maximum exponent value as 38 which means in single precision you can represent up to 7 significant decimal places and the range of the number can be 10 to the power 38 to 10 to the power -38.

(Refer Slide Time: 16:50)



"Normalized" Representation

- We shall now see how *E* and *M* are actually encoded.
- Assume that the actual exponent of the number is *EXP* (i.e. number is $M \times 2^{EXP}$).
- Permissible range of *E*: $1 \leq E \leq 254$ (the all-0 and all-1 patterns are not allowed).
- **Encoding of the exponent E**:
  - The exponent is encoded as a biased value: $E = EXP + BIAS$
    where BIAS = 127 ($2^{8-1} - 1$) for single-precision, and
    BIAS = 1023 ($2^{11-1} - 1$) for double-precision.

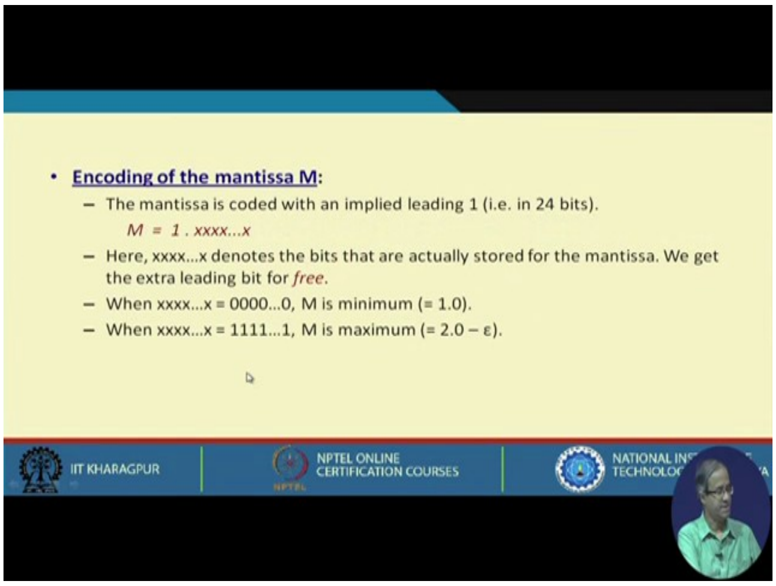
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INS TECHNOLO

Now in floating point number there is an interesting concept called normalization; let us look at the encoding part of it that how the exponent and mantissa are actually stored or encoded. Let us assume that the actual exponent of the number is EXP. The number is M multiplied with 2 to the power EXP. We are encoding this EXP in some way and we get E. Now the value of E can range from 1 up to 254; it is an 8 bit number you know for unsigned number the range is 0 up to 255, but here we are saying 1 up to 254, which means we are leaving out 0 and 255.

So, the all 0 and all 1 patterns are left out; they are not allowed. So, how we are encoding E, you have your actual exponent EXP; we are adding a bias to it. You see EXP can be either negative or positive, but after we add this bias E becomes always positive. So, how much is the bias? For 8-bit exponents for single precision you take the bias as 127. So, because you see I mean say again if you look at the value of the actual exponent it can range from -128 to +127. Now this one combination you leave out let us say it is from -127 to +127. So, if you add a bias 127 to it, -127 + 127 become 0, and 127 + 127 becomes 254. So, this is positive.

The bias is 127 for single precision, and it is 2^{11-1} - 1 which is 1023 for double precision, because in double precision we have 11 bits for the exponent.

(Refer Slide Time: 19:31)



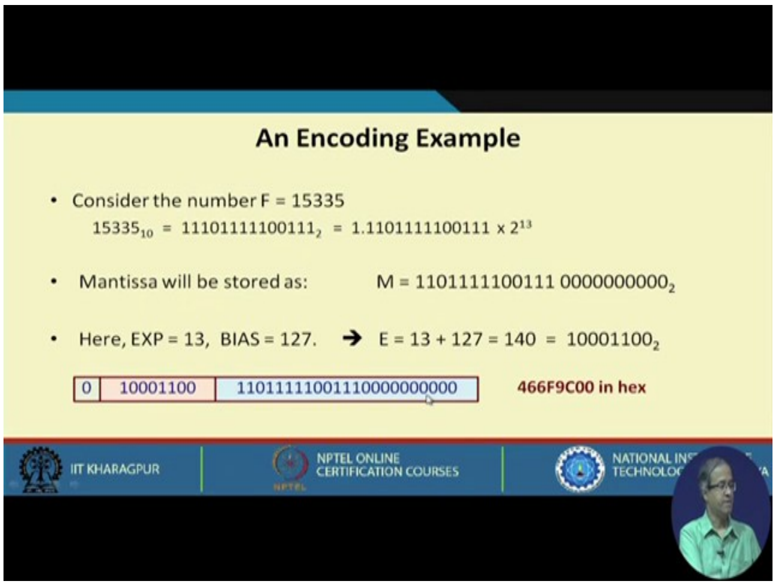
- **Encoding of the mantissa M:**
  - The mantissa is coded with an implied leading 1 (i.e. in 24 bits).

    M = 1 . xxxx...x
  - Here, xxxx...x denotes the bits that are actually stored for the mantissa. We get the extra leading bit for *free*.
  - When xxxx...x = 0000...0, M is minimum (= 1.0).
  - When xxxx...x = 1111...1, M is maximum (= 2.0 – ε).

Now, for the mantissa as had said earlier we encode the mantissa in such a way that it always starts with a leading 1 and whenever you encode mantissa in this way we say that the number is normalized. If the mantissa starts with 0 we say that it is not normalized; and this x x x denote the bits that are actually stored because the first bit is always 1; we do not store it explicitly. We are getting this extra bit for free, we are storing only the remaining 23 bits.

When the value of x is all 0's the value of M is minimum 1.0 0 0 0 0 the value is 1. When x is all 1's; that means 1.11111. As we saw earlier point 1 1 1 1 1 is the value which is very close to 1. The value of this number becomes very close to 2; 2 minus very small value epsilon. You recall we mentioned earlier that the mantissa M will be having a value between 1 and 2, and this is why it is so.

(Refer Slide Time: 20:54)



An Encoding Example

- Consider the number F = 15335
  $15335_{10} = 11101111100111_2 = 1.1101111100111 \times 2^{13}$
- Mantissa will be stored as: $M = 1101111100111\ 0000000000_2$
- Here, EXP = 13, BIAS = 127. → $E = 13 + 127 = 140 = 10001100_2$

0	10001100	11011111001110000000000

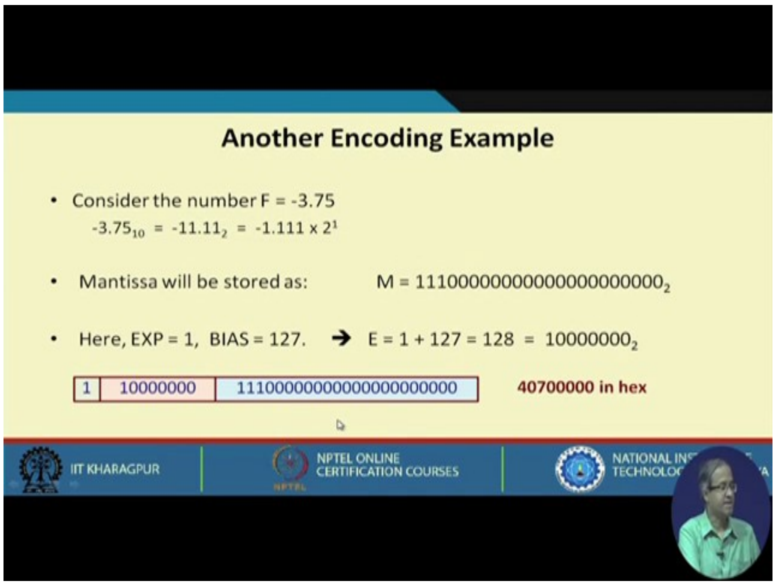
466F9C00 in hex

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INS TECHNOLO

Let us look at some encoding examples. We consider a number F let us say 15335. If you convert this into binary it is like this, and if you expressed it in fractional form, if you put a decimal point here the radix point here. So, if you count the digits into 2 to the power 13 there are 13 binary bits here.

So, if you express the number like this you see the mantissas already normalized. It starts with 1; so you are not storing this 1 you are storing the remaining bits. So, the mantissa will be stored as this. How many bits will be there for the mantissa? 23 bits and exponent is 13. So, bias for single precision is 127. So, the actual value of E will be 13 + 127 = 140. This number will be stored as sign positive, followed by exponent, followed by mantissa. If you divide this number into 4 4 bits and convert into hexadecimal you will see you will start is 0 1 0 0 which is 4, then 0 1 1 0 which is 6 ,again 0 1 1 0 which is 6, 1 1 1 1 which is F, and so on.

(Refer Slide Time: 22:24)



Another Encoding Example

- Consider the number F = -3.75
  $-3.75_{10} = -11.11_2 = -1.111 \times 2^1$
- Mantissa will be stored as: $M = 11100000000000000000000_2$
- Here, EXP = 1, BIAS = 127. $\rightarrow$ $E = 1 + 127 = 128 = 10000000_2$

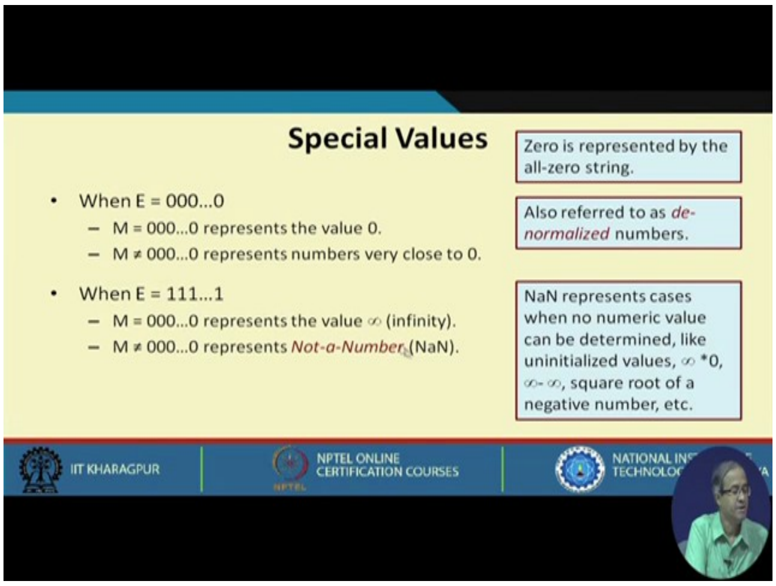
1	10000000	11100000000000000000000	40700000 in hex

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INS TECHNOLOG

Let us take another example where the number is negative -3.75, which in binary you can write like this. If you express in normalized form push the decimal point here it becomes like this. So, we will be storing the mantissa as only this 1 1 1 this one we do not stored and the remaining zeros and exponent EXP is 1, you add the bias it becomes 128 which is this.

So, the number will be represented as sign is one, negative exponent and mantissa which in hexadecimal is C070.

(Refer Slide Time: 23:19)



Special Values

- When E = 000...0
  - M = 000...0 represents the value 0.
  - M ≠ 000...0 represents numbers very close to 0.
- When E = 111...1
  - M = 000...0 represents the value ∞ (infinity).
  - M ≠ 000...0 represents *Not-a-Number* (NaN).

Zero is represented by the all-zero string.

Also referred to as *de-normalized* numbers.

NaN represents cases when no numeric value can be determined, like uninitialized values, ∞ *0, ∞- ∞, square root of a negative number, etc.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INS TECHNOLOG

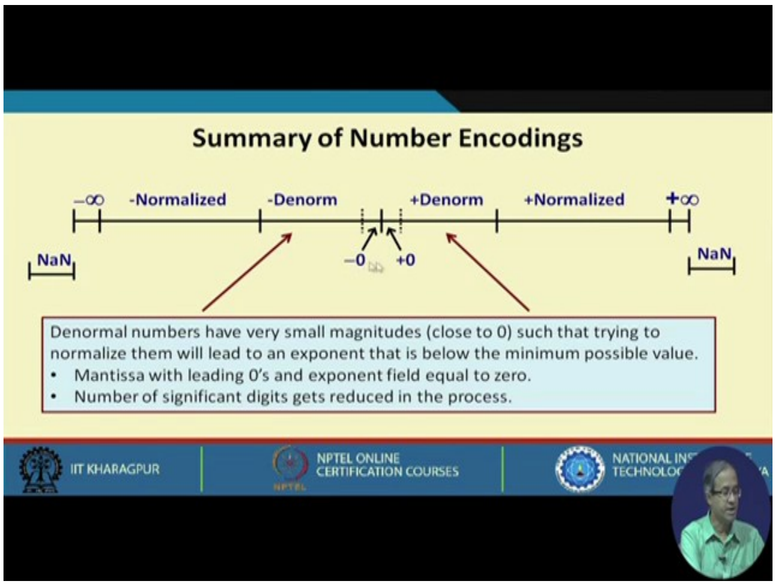
Now, some special values are supported. When we saw earlier we mentioned that the values of E all 0's and all 1's are not allowed in actual numbers, but what happens when the value of E is all 0's or all 1's. So, these are the so-called special values. When E is all 0's then if M is also all 0's this combination represents this special number 0. So, you see the number 0 is represented as an all 0 string. It can be easily checked by hardware. And for E all 0's if we have a mantissa which is not all 0's this represents some numbers which are very close to 0, because see E = 0 means it is already in biased format means 2 to the power minus 127.

After adding 127 you are getting 0. So, the magnitude of the number is really very small. Here we are representing numbers which are very close to 0 when E is all 0 these are sometimes called de-normalized numbers because if M is all 0, you do not have a 1 at all you cannot make the first bit 1.

So, for these de-normalized numbers your exponent has to be all 0's meaning that this is a special kind of a number where the mandatory requirement of the first bit of the mantissas 1 is not to be assumed in this case, and 0 is represent with all 0 string.

When E is all 1's then the all 0's combination of the mantissa represents the value infinity; this is our representation in the IEEE format, and any value which is not equal to 0 this represents an invalid number which means not a number sometimes it is called NaN. Why it is required? Because you see there is certain cases where you do not know the value of a number like you have defined some variables, but not initialized them.

(Refer Slide Time: 26:07)


Summary of Number Encodings

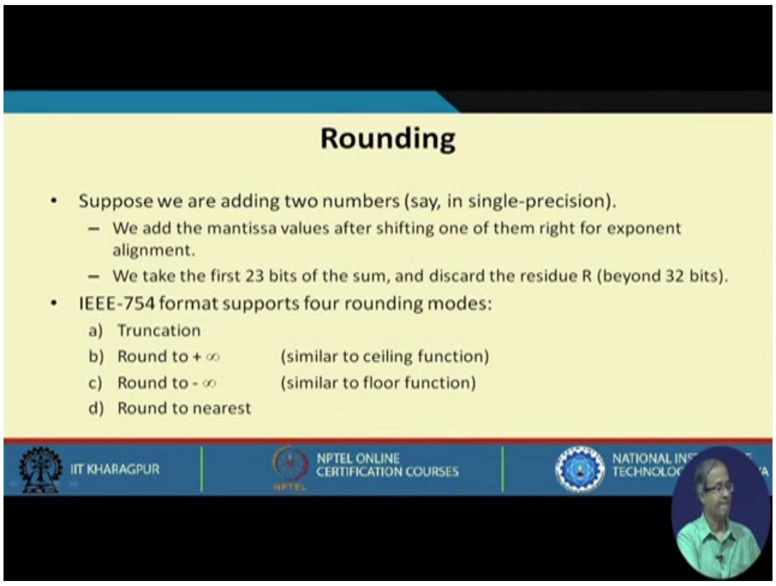
Denormal numbers have very small magnitudes (close to 0) such that trying to normalize them will lead to an exponent that is below the minimum possible value.

- Mantissa with leading 0's and exponent field equal to zero.
- Number of significant digits gets reduced in the process.

In this scale this summary of number encodings you can see that in between when you have the exponent and mantissa values all 0, you have the number 0. So, there are two representations of 0, +0 and -0. And for very small numbers you can have de-normalized numbers; and when you have normalized numbers it will be beyond that and you can go up to plus infinity, plus infinity is a special representation; minus infinity is also special representation.

Beyond that you have invalid numbers which are NaN.

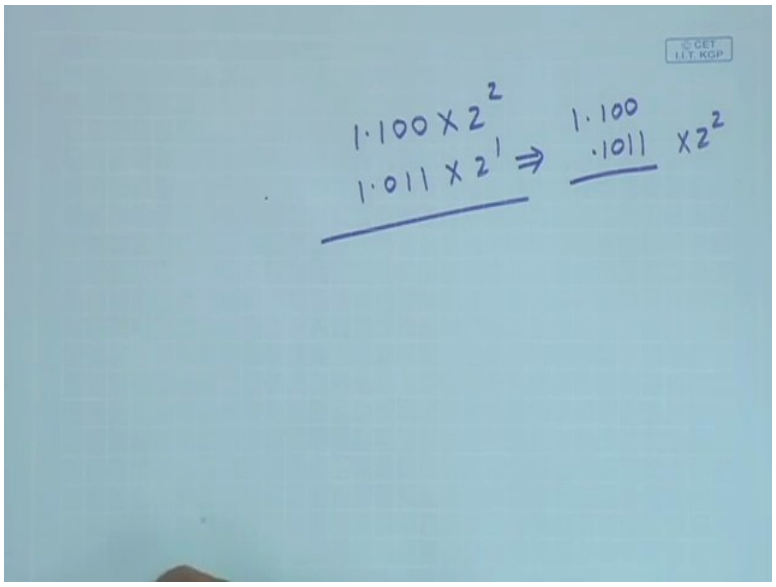
(Refer Slide Time: 26:54)


Rounding

- Suppose we are adding two numbers (say, in single-precision).
  - We add the mantissa values after shifting one of them right for exponent alignment.
  - We take the first 23 bits of the sum, and discard the residue R (beyond 32 bits).
- IEEE-754 format supports four rounding modes:
  a) Truncation
  b) Round to + ∞ (similar to ceiling function)
  c) Round to - ∞ (similar to floor function)
  d) Round to nearest

Now, there is another feature of the IEEE format, which is also very much useful this called rounding. Suppose when we add two numbers in single precision. Normally we add the mantissa value after aligning the exponents. What do you mean by exponent alignment? Suppose my one number is 1 . 1 0 0 into 2 to the power 2.

(Refer Slide Time: 27:18)



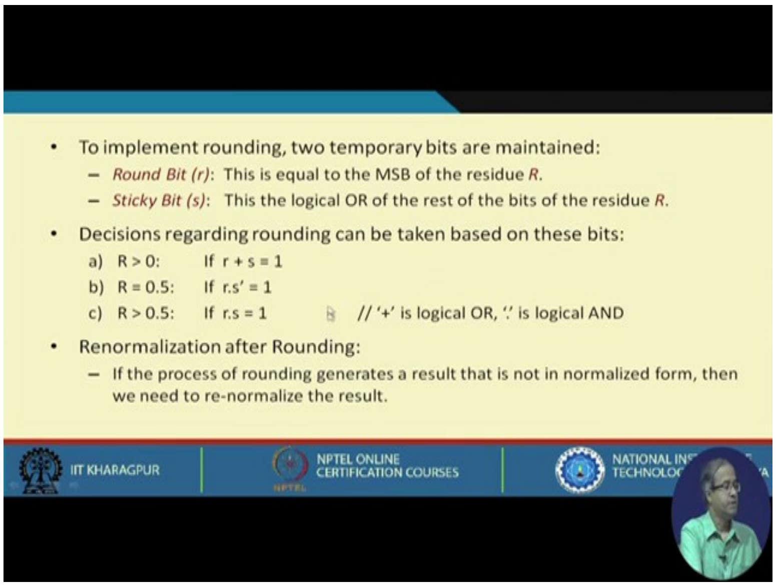
$$\begin{array}{l} 1.100 \times 2^2 \\ \underline{1.011 \times 2^1} \end{array} \Rightarrow \begin{array}{l} 1.100 \\ \underline{.1011} \end{array} \times 2^2$$

Let us say other number is 1 . 0 1 1 into 2 to the power 1. So, when we want to add we cannot straight add because the exponents and not same. What we do this second number we make it 2 to the power 2, and for that we shift the decimal point to the left by one position.

So, it becomes 0.1011 then we can add 1.100 with this, and we get the result. This is what is mentioned here. We add the mantissa value after shifting one of them. We shall be coming to this again. After adding, the first 23 bits of this sum is taken and the remaining bits are discarded; this is called the residue. Now IEEE format supports four different rounding modes, one is truncation --- beyond 23 bits you just discard the remaining bits this is truncation, rounding to plus infinity is the second mode what it says is that you look at r if you see where there r is greater than or equal to 0.5. If r is greater than or equal to 0.5 just like ceiling function, you increase mantissa by 1. Similarly if it is less than 0.5 we will rounding to minus infinity you move it to the next lower integer value, and second one is a rounding.

So, depending on the value of the r here you are either moving into the next higher or the next lower, and rounding means if it is 0.5 or higher you move it up, otherwise you move it down.

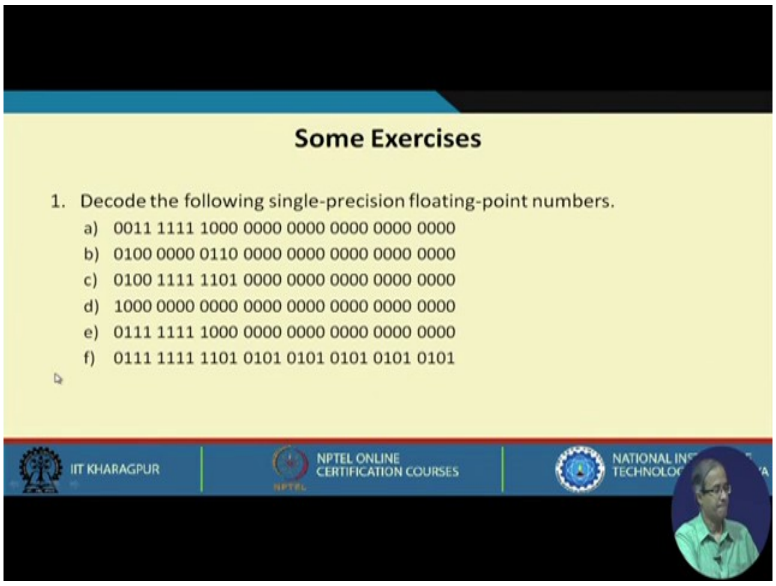
(Refer Slide Time: 29:14)



- To implement rounding, two temporary bits are maintained:
  - *Round Bit (r)*: This is equal to the MSB of the residue *R*.
  - *Sticky Bit (s)*: This the logical OR of the rest of the bits of the residue *R*.
- Decisions regarding rounding can be taken based on these bits:
  a) $R > 0$: If $r + s = 1$
  b) $R = 0.5$: If $r.s' = 1$
  c) $R > 0.5$: If $r.s = 1$ // '+' is logical OR, '.' is logical AND
- Renormalization after Rounding:
  - If the process of rounding generates a result that is not in normalized form, then we need to re-normalize the result.

This is rounding to the nearest. To implement this two temporary bits are used in the representation, one is called round bit r which is the most significant bit of the remaining residue R, and sticky bit s which is actually the logical OR of the remaining bits of R other than the MSB.

(Refer Slide Time: 30:24)



Some Exercises

1. Decode the following single-precision floating-point numbers.
   a) 0011 1111 1000 0000 0000 0000 0000 0000
   b) 0100 0000 0110 0000 0000 0000 0000 0000
   c) 0100 1111 1101 0000 0000 0000 0000 0000
   d) 1000 0000 0000 0000 0000 0000 0000 0000
   e) 0111 1111 1000 0000 0000 0000 0000 0000
   f) 0111 1111 1101 0101 0101 0101 0101 0101

Here are some exercises for you to work out.

We have seen some representations of floating point numbers and in particular the IEEE format that is almost universally used nowadays in almost all computer systems. We have seen how numbers are represented, how some special numbers are represented that are very useful during computations and also how we can do rounding of the numbers.

With this we come to the end of lecture number 38. In the next lecture we shall be starting discussion on how we can carry out arithmetic using floating point numbers. In this lecture you have looked at just the representation, later on we would be seeing how we can carry out addition, subtraction, multiplication and division on floating point numbers.

Thank you.