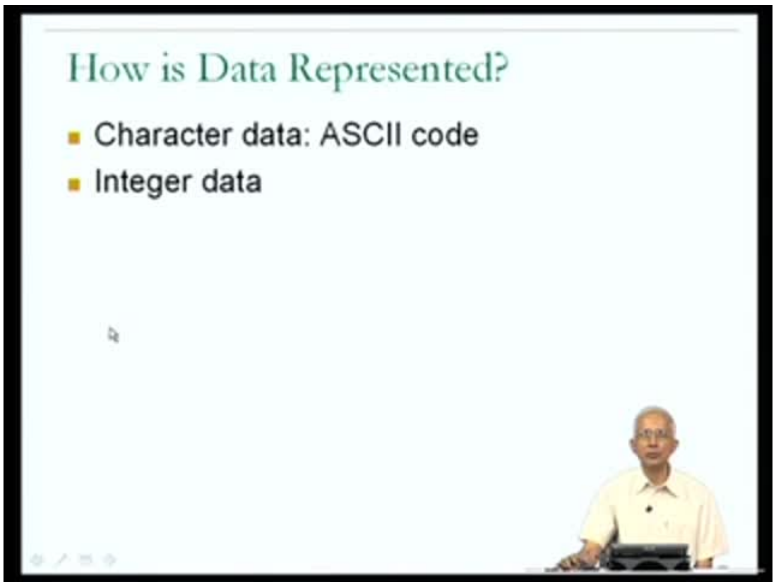
High Performance Computing

Prof. Matthew Jacob
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Module No.# 01
Lecture No. # 02

(Refer Slide Time: 00:33)

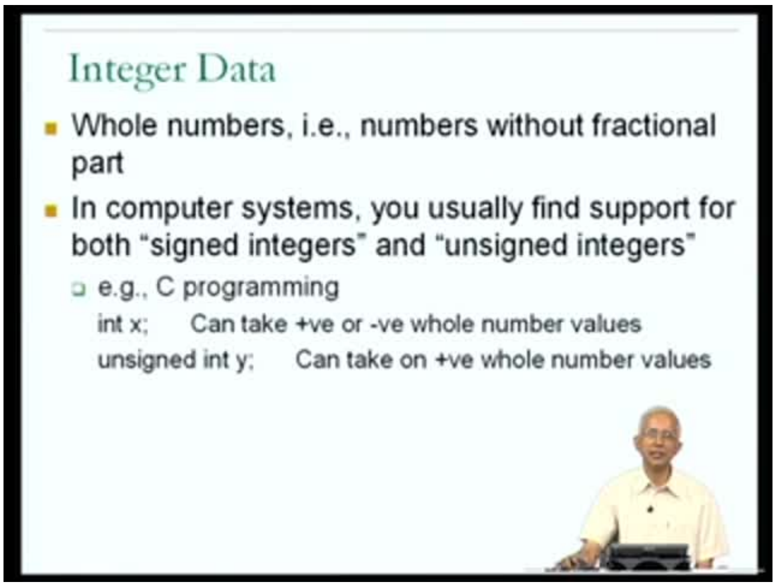


How is Data Represented?

- Character data: ASCII code
- Integer data

Welcome to the second lecture of a course on High Performance Computing. In the first lecture, we had seen something about programs, what a program is, how program may have to be translated for execution on a digital computer and we then moved on to a discussion of how data is represented. We had seen that, the different kinds of data and that character data is represented using a standard convention call the ASCII code. The ASCII code is, ASCII is the short form for the American Standard Code for Information Interchange and it is an international standard.

(Refer Slide Time: 00:58)



Integer Data

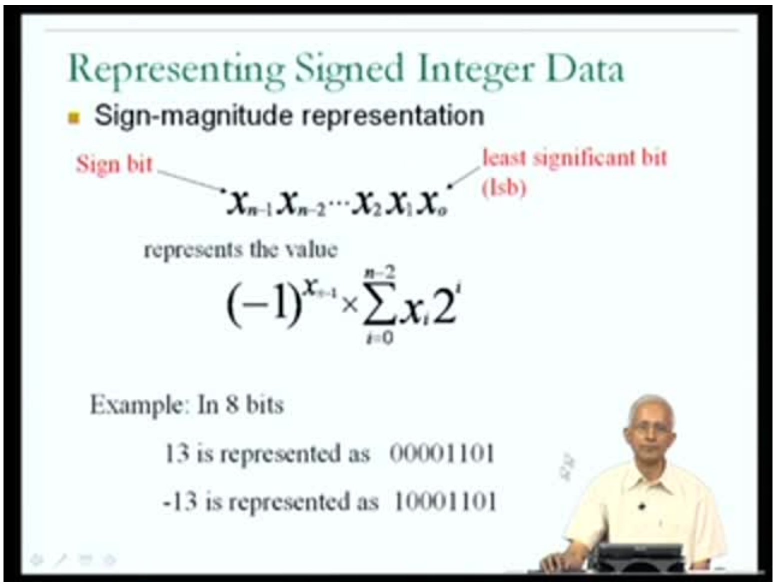
- Whole numbers, i.e., numbers without fractional part
- In computer systems, you usually find support for both "signed integers" and "unsigned integers"
  - e.g., C programming

    int x; Can take +ve or -ve whole number values

    unsigned int y; Can take on +ve whole number values

We then moved on to, how the integer data is represented. We recall that integer, integers are whole numbers, that is numbers without fractional parts. And I mentioned that, from your C programming, you are familiar with the concept of both signed and unsigned integers. It, therefore, stands to reason that in computer system, there must be different possibilities how to represent values, that are signed as opposed to those which are unsigned.

(Refer Slide Time: 01:26)



Representing Signed Integer Data

- Sign-magnitude representation

Sign bit

least significant bit (lsb)

$$x_{n-1}x_{n-2}\cdots x_2x_1x_0$$

represents the value

$$(-1)^{x_{n-1}} \times \sum_{i=0}^{n-2} x_i 2^i$$

Example: In 8 bits

13 is represented as 00001101

-13 is represented as 10001101

So, I went into one of the possibilities of how to represent signed integer data, which is the Sign-magnitude representation. Remember the, what we are trying to understand here is, a scheme that could be used in a digital computer to represent a variable, which could take on either negative or positive integer values.

Now, here the key thing to understand is that, their sign must be included in the representation. So, if I have an n bit representation for a signed integer value, then clearly the sign must be incorporated in that representation. In the Sign-magnitude representation, the most significant bit of a signed integer is used as a sign bit. In other words, the most significant bit indicates whether the signed integer is a negative or positive value. The remaining bits indicate the magnitude of the value in binary as in, as the summation on the, on the right of this expression would, would have you understand. And, therefore, the, the bit at the left or bit x n minus 1 is the sign bit in the Sign-magnitude representation and is the standard convention as I mentioned last time.

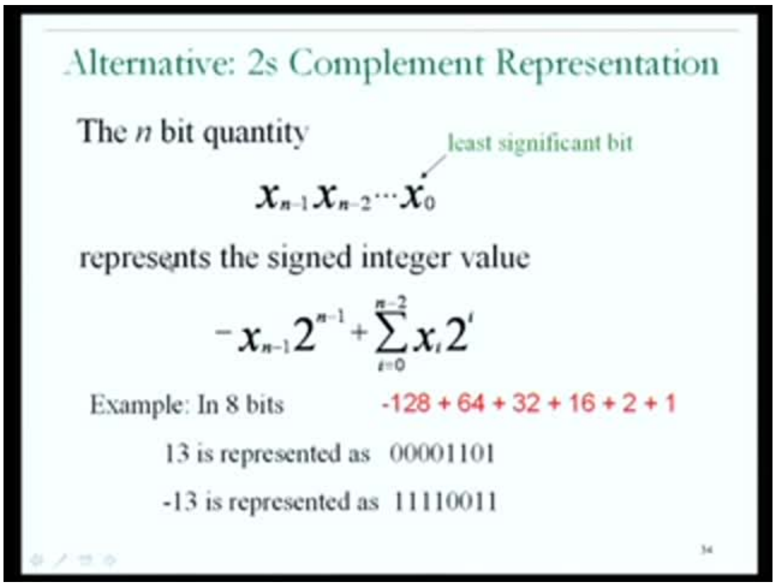
We refer to the bit which contributes the least to the magnitude of the value that is being represented, as the least significant bit. And in this case, I am showing the least significant bit on the extreme right and I am bringing it as x 0. Please note that this is just a convention. I could just as well have shown the least significant bit on the extreme left and the expression would have been exactly the same. Because, I could have the labeled the bit at the extreme left as x sub zero. So, this is one way that, signed integers can be represented. I had mentioned at the end of the previous lecture that, this is in fact, not the most popular way among computer manufacturers for representing signed integers, but let us just look at some examples to make sure we understand this. Now, let us suppose that I am representing signed integers in 8 bits. In other words, I have x 0, x 1, x 2, x 3, up to x 7, right. So, x n minus 1 is x 7. The question is how do, how do I represent 13, per plus 13? Plus 13 is a signed integer with a positive sign. The answer is very clearly the most significant bit or the sign bit is going to be a 0, because, the value has to be positive. 13 is a positive value. Therefore, minus 1 must be raised to the value 0. So, the most significant bit must be a 0, which it is and the remaining bits must be such that, this summation adds up to the value 13.

So, I notice here that, there is 1 multiplied by 2 to the power of 0, which is 1, plus 1 multiplied by 2 to the power of 2, which is 2, I am sorry which is 4, plus 1 multiplied by

2 to the power of 3, which is 8. So, the values thus being represented is, 8 plus 4 plus 1, which is 13.

So, the Sign-magnitude representation for plus 13 is 000 etcetera 1101. Similarly, the Sign-magnitude representation for minus 13 is going to be very similar except that, the sign bit is going to be a 1 ok.

(Refer Slide Time: 04:33)



Alternative: 2s Complement Representation

The n bit quantity

$$x_{n-1}x_{n-2}\cdots x_0$$

least significant bit

represents the signed integer value

$$-x_{n-1}2^{n-1}+\sum_{i=0}^{n-2}x_i2^i$$

Example: In 8 bits -128 + 64 + 32 + 16 + 2 + 1

13 is represented as 00001101

-13 is represented as 11110011

Now, if Sign-magnitude is not the most popular representation for signed integers, you ask what is and the answer is something called the 2's complement representation. Now, in the 2's complement representation, n, the n bit value x n minus 1, x n minus 2, through x 0 represents the signed integer value as shown by the expression over here.

Now, if you look at this expression carefully, you will notice that, the second term is exactly the same as the second term for the Sign-magnitude representation. I will just, just look at this term, its summation from i equal to 0 through n minus 2, x i by 2 to the power y. I am going to go back to the previous slide, so that, you can just confirm that, this is the same as this term over here. It is exactly the same.

So, the only difference between the Sign-magnitude representation and the 2's complement representation comes in the first term. In the case of the Sign-magnitude representation, the first term was minus 1 to the power n minus 1 x, I am sorry, minus 1 to the power x sub n minus 1 and that was going to be minus 1, if x n minus 1 was 1 and plus 1, if x n minus 1 was 0. In the 2's complement representation, on the other hand, we have magnitude associated with the sign bit and the magnitude is going to be quite high. In other words, if the most significant bit, that we see over here is a 0, then this is going to be x n minus 1 multiplied by 2 to the power n minus 1. So, it is going to have a positive value of great magnitude and if it is, if it is, if it is a 1, this is going to have a negative value of a great magnitude.

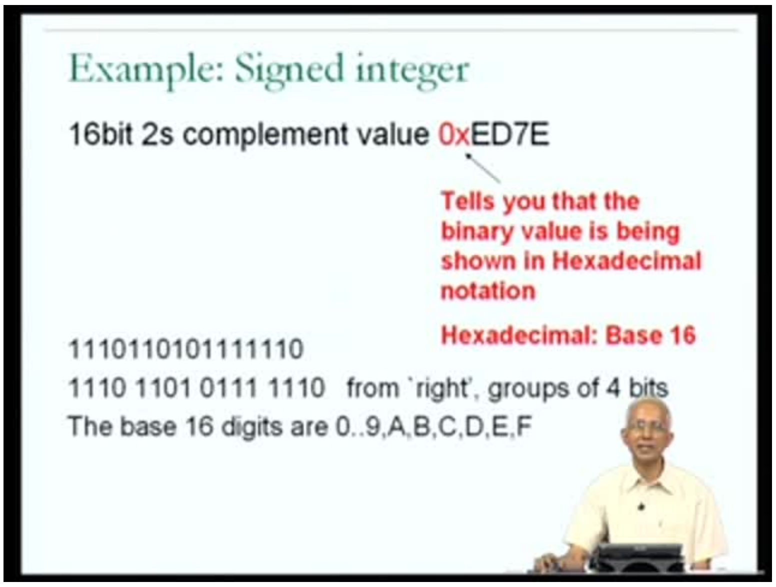
So, the net effect is that, for example if I wanted to represent in 8 bits, plus 13, the value would be similar to what we saw in the Sign-magnitude representation. x n minus 1 is 0. Therefore, this term is effectively 0 and the magnitude is the same as that, as we had in the case of the Sign-magnitude representation. Whereas, if I want to represent minus 13, I do need to have a large negative component, because, the remaining, the second term is going to add a positive value to the net result of this value. Therefore, a large negative value added to a smaller positive value, which is going to result in minus 13.

Now, let us just verify that, this is in fact, the same as minus 13. I have just put it up and we need to verify that, this is equal to minus 13, if this expression is, what is, what is used in the representation? So, how do I compute what this corresponds to? I note that, this representation includes 1. So, minus x n minus 1 is 1 multiplied by 2 to the power n minus 1, which is 2 to the power of 7. 2 to the power 7 is 128.

So, it is minus 128. That is followed by a 1, which is part of this summation, which is going to be 2 to the power of 6, in, in an 8 bit representation. So, that is 64 and so on. So, the net result is that, in evaluating this, using this expression, I find out that, it is equal to minus 128. So, 64 plus 32 plus 16. Then the, there is no 8 and there is no 4, but there is a 2 and there is a 1 and when I add this up, what I get is minus 13.

So, this is an equivalent way to represent signed integers. Now, the, the property that we see over here is that, if the value that is being represented is negative, then most definitely the most significant bit, the bit at the extreme left is going to be a 1, because, that is the only way that the sum, based on this expression, could be negative.

(Refer Slide Time: 08:48)


Example: Signed integer

16bit 2s complement value 0xED7E

Tells you that the binary value is being shown in Hexadecimal notation

Hexadecimal: Base 16

1110110101111110

1110 1101 0111 1110 from `right', groups of 4 bits

The base 16 digits are 0..9,A,B,C,D,E,F

Therefore, in some sense, the most significant bit is a sign bit, but it is not only a sign bit, because it also has a value associated with it. It is a highly significant bit in that sense, and therefore, you do not refer to it as a sign bit, but as the most significant bit. So, that is the way this 1’s, the 2's complement representation and the Sign-magnitude representation differ from each other. Now, the question that you will ask and we will answer a little bit later is, why is this better than the Sign-magnitude representation?

But let us look at an example first. Ok now, in this example, we are going to try to understand, what it means to be in the 2's complement representation. So, here what I have given you is a 2's complement representation for a value. We do not know what the value is. We, we know that is a signed integer value and tell you that is a sixteen bit 2's complement value. 0xED7E. Now, this, when you look at it, you probably do not recognize this as being a number. So, I do need to explain, what I mean by 0xED7E. Let me start, by doing that. Now, the 0 x is an indication to you that, the binary value which is being presented to you, is being shown in something called the hexadecimal notation.

So, 0 x is just a prefix telling you that, the subsequent value is actually a hexadecimal representation. Hexadecimal means base 16. So, upto now, we were talking about base 2. Binary values are represented in base 2. Here we are hearing about a number which is represented in base 16. So, we do need to understand what is happening here.

So, what is this E D 7 and E? So, let us just consider this 16 bit binary sequence. 1 1 0 etcetera, etcetera. So, there are 16 bits here. Now, I know that, if I was to view this as a binary value, and normally if this was a unsigned binary value I will say that, this is, has a value of zero. This bit, the one on the extreme right side has a value of 2, value of 4 and so on, but if I was to group these bits into groups of 4, starting from the right side, bearing in mind that 2 to the power of 4 is equal to 16, I would actually find out that the bits in anyone of these groups, for example, consider the bits in the right most group, there are 4 bits there. Now, the, those 4 bits currently have, in this particular example, those 4 bits have the value 1 1 1 0, but in general, a sequence of 4 bits could take on 16 different values from 0 0 0 0 to 0 0 0 1 to 0 0 1 0, all the way up to 1 1 1 1.

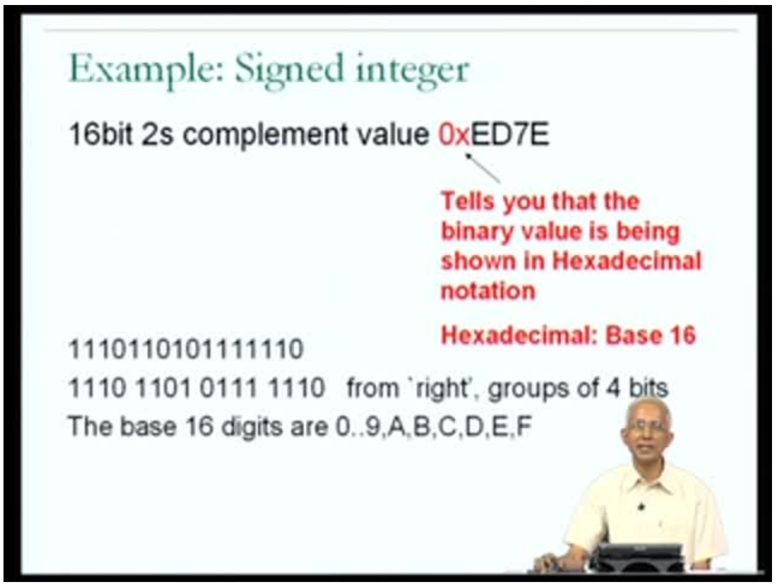
So, there are 16 different values that a group of 4 bits could take on, and these are in fact, the 16 possible values of the hexadecimal base 16 number system. So, what we have done over here is, rather than representing huge binary strings such as this 16 bit binary string by 16 bits, which will take a long time to read or to write, I am using a shorter notation, in which I compress a bit sequence into a hexadecimal sequence by just grouping the bits from the right in groups of 4 and then I will read each of those groups of 4 as a equivalent hexadecimal digit.

Now, the question you will ask is, what are the hexadecimal digits? I know that in the decimal system, the decimal digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. In the hexadecimal system, there are 16 digits and I can use 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 as the first 10 of those digits, but I need 6 more.

Therefore the convention that is used is that, we use A, B, C, D, E and F to represent the next 6 hexadecimal digits. Therefore, the 16 hexadecimal digits are 0 through 9 followed by A, B, C, D, E and F. In other words, the binary value 1 1 1 1 which corresponds to the value 15, would be associated with a hexadecimal digit F.

Similarly, the binary value 1 0 1 0 which corresponds to 10 would be associated with the hexadecimal digit A. So, when we, when we read this, we might say 0 through 9, A which is 10, B which is 11, C which is 12, D which is 13, E which is 14 and F which is 15. So, the hexadecimal notation here, it is indicated by the fact that, I prefixed the value by 0 x and the remaining are digits from the hexadecimal number system. In other words, the values between 0 and 9 or A through F, ok.

(Refer Slide Time: 08:48)



Example: Signed integer

16bit 2s complement value 0xED7E

Tells you that the binary value is being shown in Hexadecimal notation

Hexadecimal: Base 16

1110110101111110

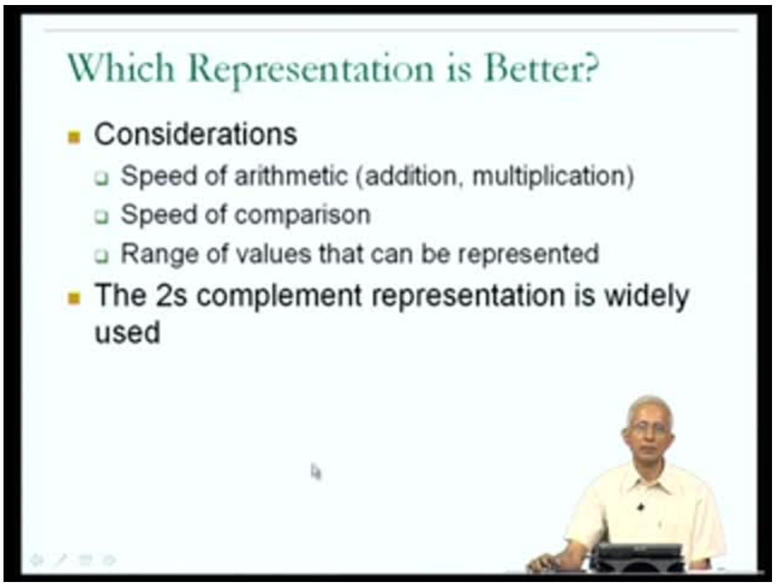
1110 1101 0111 1110 from 'right', groups of 4 bits

The base 16 digits are 0..9,A,B,C,D,E,F

So, we are going to represent values typically in the examples that are to come, using this hexadecimal notation. So, this particular values that I have, if I was to read it off, 1 1 1 0 is E, 1 1 0 1 is D, 0 1 1 1 is 7, 1 1 1 0, once again is E and therefore, this 0xED7E that I had over here, actually corresponds to this sequence of bits, ok.

Now, the question that we had started with was, what is the 16 bit 2's complement value 0xED7E? Ultimately, this is the 2's complement representation for some signed integer and I want to know in decimal, what that signed integer value is? So, the way that I can get an answer to that question is, using this expression. Just as we did in this example, I can do the same thing with this example, once I understand the meaning of the hexadecimal notation, ok.

(Refer Slide Time: 14:20)



Which Representation is Better?

- Considerations
  - Speed of arithmetic (addition, multiplication)
  - Speed of comparison
  - Range of values that can be represented
- The 2s complement representation is widely used

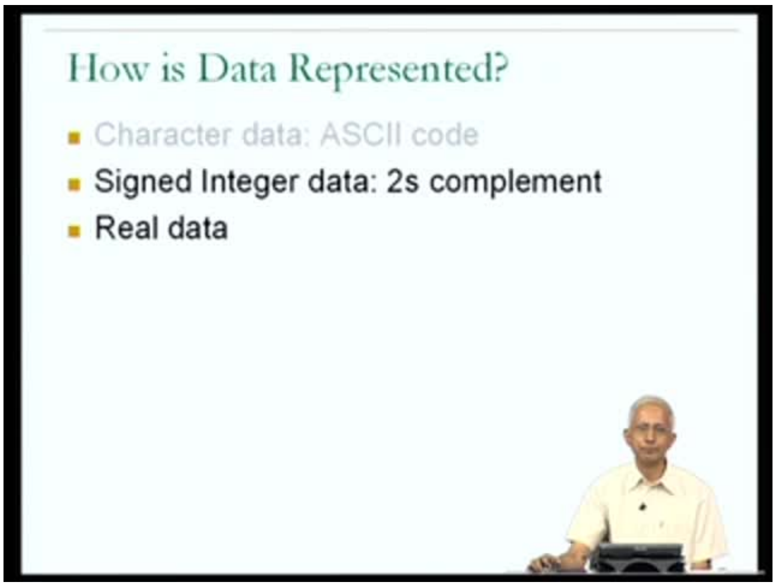
Now, I had given you some indication that the 2's complement representation is considered superior to the Sign-magnitude representation and it might be useful first to understand, on what grounds that opinion has arisen. Now, what, what, what, are the different considerations that might come into play, in deciding whether one representation for signed integers is better than another? The answer is, ultimately, as you will understand, somebody will have to build a electronic circuit to add signed integer values. This is what is going to be present in a computer, in order to execute the add instruction, that your program might make use of. So, the speed of the arithmetic might be an important consideration in deciding whether one representation is better than another. Another consideration which might come into play, is the speed of comparison, how quickly can a circuit determine whether two values, whether one value is greater than another value or less than the other value?

So, this is another operation for which a circuit might have to be built and therefore, the speed of that circuit might be critical in determining which of these representations is better for the execution of programs. Another indication, another factor that might come into play in deciding which representation is better than another, is the range of values that can be represented.

Now, just to give you a little bit of idea about this, if you think about the Sign-magnitude representation and ask yourself, what is the representation for the value 0, you will quickly realize that, there are actually 2 representations for the value 0. One representation will have a sign bit of 0, followed by all 0s. The other representation will have a value of 1, followed by all 0s. Technically, that stands for minus 0, but that you know, is the same as 0. Therefore, in the Sign-magnitude representation, there are 2 representations for the value 0.

Now, if I have, let us say, a 4 bit sign-magnitude value, I know that there are only sixteen different possible values that can be represented, from 0000 zero up through 1111. If I am using 2 of those values for 0, that means there are, only 14 other values can be represented using the Sign-magnitude representation in 4 bits. On the other hand, if I have 4 bits in the 2's complement representation, I can represent 15 other values in addition to 0. Therefore, the range of values that can be represented is larger in the case of the 2's complement representation. As it happens, the speed of arithmetic and comparison is also superior, for which reason the 2's complement representation has become the more popular of the two that we have seen. So, the 2's complement representation is very widely used in digital computers for the representation of signed integers.

(Refer Slide Time: 17:20)

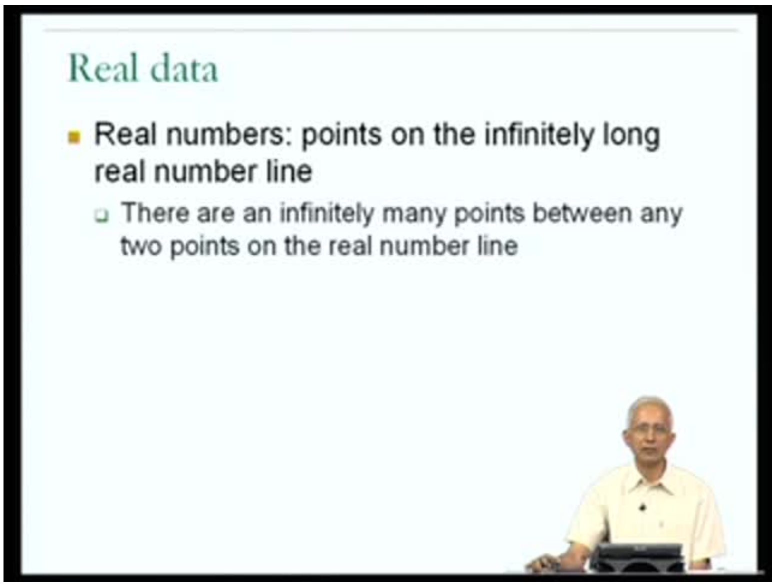


How is Data Represented?

- Character data: ASCII code
- Signed Integer data: 2s complement
- Real data

Now, with this we have seen that, character data is represented almost universally using the ASCII code and that signed integer data is represented most widely, using the 2's complement representation. What other kinds of data do we have to take into account?

(Refer Slide Time: 17:41)



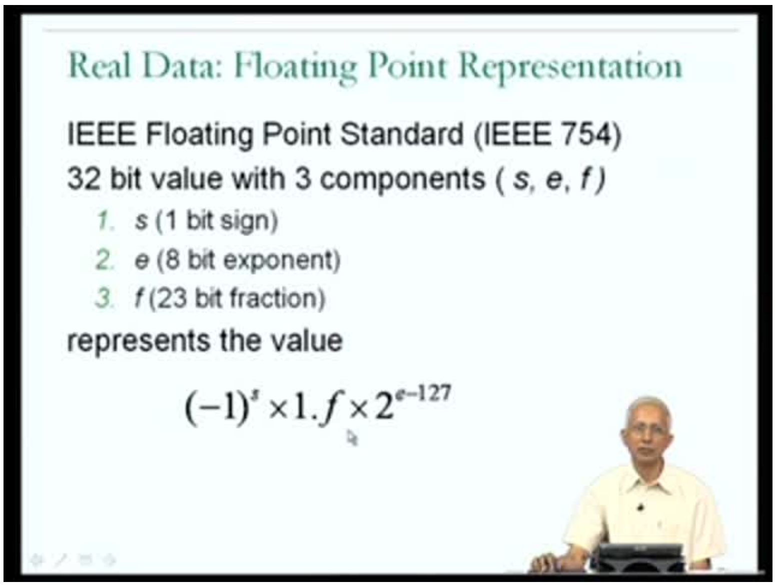
Real data

- Real numbers: points on the infinitely long real number line
  - There are an infinitely many points between any two points on the real number line

Now, I am going to talk about something called real data. All of you have encountered real data and you know that a real number is usually defined in, with reference to the real number line. You have a picture of an infinitely long real number line, which extends infinitely from your left to your right, with, in the middle somewhere, the value zero and you know that, a real value is, basically, a real number is any point on that line. And you also know that, between any two points on the real number line there are infinite numbers of other points and therefore, an infinite number of other real numbers. So, these are the properties of real data.

So, there are infinitely many points, between any 2 points on the real number line and you are all familiar with real world phenomenon which must be modeled as real numbers. And this is something, which I did not talk about in connection with signed or unsigned integers, but it is something to think about. When, in writing a computer program, you have to make a decision. You are writing a computer program to achieve some, well staged objective, in deciding whether to declare a variable as an integer variable, as signed or unsigned, you would have to know whether there is a possibility that the value - the real world phenomenon corresponding to it could take on negative values. In which case, you would declare it as an unsigned piece of data, unsigned integer.

(Refer Slide Time: 19:14)



Real Data: Floating Point Representation

IEEE Floating Point Standard (IEEE 754)

32 bit value with 3 components (*s*, *e*, *f*)

1. *s* (1 bit sign)
2. *e* (8 bit exponent)
3. *f* (23 bit fraction)

represents the value

$$(-1)^s \times 1.f \times 2^{e-127}$$

Similarly, you must be aware of situations where the real world phenomenon that you are trying to take into account in your computer program, may have to be modeled as a real value. Now, unfortunately, in computer systems, it is not possible to represent all real values exactly, because, as we saw, there are an infinite numbers of real values between any two real values, however close they are on the real number line. And therefore, different kinds of representations must be used to, to approximately represent many real values. And one such representation is, something called the floating point representation. And it is from the name of this representation that, the key word float in the C programming language comes. You declare a variable called float x, you are declaring it to be a floating point variable and essentially you are saying that, the x corresponds to some real value. Ok.

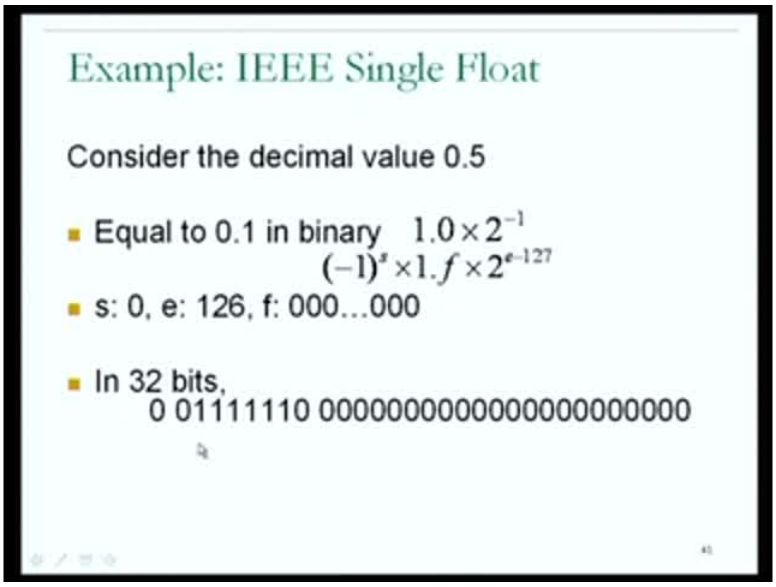
Now, among the different floating point representation possibilities, the most widely used international standard is something called the I triple E floating point standard and one of the I triple E floating point standards that I am going to talk about, is known as the I triple E 754 standard. Now, the situation as far as real values is concerned is that, like signed integers, real values could be either positive or negative. The real number line stretches infinitely to your left and to your right, half of it is negative, it corresponds to negative values, the other half of it corresponds to positive values.

In addition, in the I triple E, therefore, there is the need for a sign bit, which is the first part of the 32 bit I triple E floating point representation for a real piece of data. To represent the magnitude of the floating point, of the real value, it is not possible to use something like the magnitude representation used for signed integers, because, real values could become much larger or much smaller than what the, in whole numbers that we were concerned with as far integers are concerned. Therefore, the convention that we use in the floating point representations is that, we have something called an exponent and something called a fraction. And put together, the sign, the exponent and the fraction represent a real value. The way that the exponent is used is that, it is used as the power of some kind of a base and the way that the fraction is used is that, it is used as a fractional part which is multiplied by the base, raised to the power of the exponent.

Now, in the I triple E 754 standard, this somewhat interesting expression is used to interpret the value of a 32 bit float value. The sign bit is not too difficult to understand. If the sign bit is a 0, that means that, the overall real value is positive, whereas, if the sign bit is a 1, the overall real value is a negative real value. Next, we have the fraction part, which is, in this construct 1.f.

So, the fraction part is a 23 bit fraction, a sequence of 23 0s or 1s and 1.f defines the, in some sense, the magnitude of the, the value. Then, the exponent part minus 127 is what the power 2, is the power to which the constant 2 is raised, in order to give the multiplicative factor, through which this real value will be evaluated. So, we do need to understand a few things about this representation. First of all, why do they use this 1.f and secondly, why do not they just raise 2 to the power e, instead of having 2 raised to the power e minus 127? And, I will talk about this, but first, let us look at an example.

(Refer Slide Time: 23:01)



Example: IEEE Single Float

Consider the decimal value 0.5

- Equal to 0.1 in binary 1.0×2^{-1}
  $(-1)^s \times 1.f \times 2^{e-127}$
- s: 0, e: 126, f: 000...000
- In 32 bits,
  0 01111110 00000000000000000000000

So, this expression, we you want to find out how is the value, the decimal value 0.5 represented? What I mean by the decimal value 0.5 is, what you understand in day to day languages, half. Decimal means base 10. So, in order to evaluate 0.5 in the I triple E floating point representation, I need to get it into this kind of a form, minus 1 to the power s multiplied by 1.f into 2 to the power e minus 127.

Therefore, first of all, I need to represent this value in binary. So, how do I represent 0.5 in binary? The answer is, just as in the decimal system, I can have a binary point, and just as in the decimal system, I can have a continuation of the digits which were on the left, on to the right. Remember, there in the decimal number system, I have a situation where the first digit is multiplied by 10 to the power 0, the second digit is multiplied by 10 to the power 1, the third digit by 10 to the power 2 and so on. Similarly, on the, to the right of the decimal points, the first digit is multiplied by 10 to the power of minus 1 or divided by 10, which is why the value of 0.5 is 5 multiplied by 10 to the power of minus 1, which is equal to 5 divided by 10, which is equal to half.

I can have a similar convention in the binary system, in which the first digit to the right of the binary point, is actually multiplied by 2 to the power of minus 1, not 10 to the power of minus 1 as it was in the decimal system, but 2 to the power of minus 1, since this is the binary system. Therefore, if I wanted to evaluate half, I would have to have 1 multiplied by 2 to the power of minus 1, which is 1 divided by 2, which is half.

Therefore, in binary what I refer to as 0.5 in decimal, which I think of as half, it is actually represented in binary as 0.1, right? 1 is the binary point. 1 is the bit to the right of the binary point which is therefore, multiplied by 2 to the power minus 1, minus 1. So, half is equal to 0.1 in binary. This is still not in the form that I am looking for. I am looking for something in the form minus 1 to the power s, 1 point fraction multiplied by 2 to the power of e minus 127.

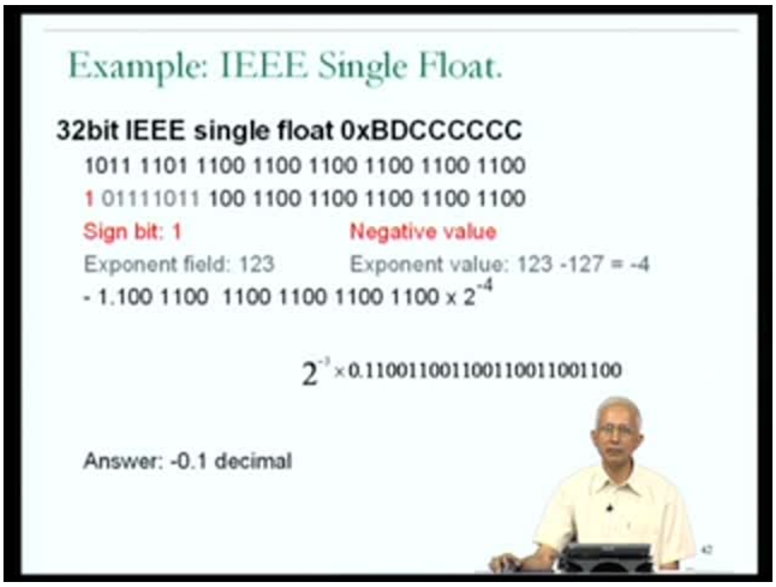
So, I need to understand what s, e and f might be, for this binary value. So, I notice that, I can write 0.1 as 1.0 multiplied by 2 to the power minus 1. So, what I have done is, I know that in the I triple E floating point standard, I need to have a 1 to the right, to the left of the binary point.

Therefore, I multiply this value 0.1 by 2. In other words, I shift that, the binary point to the right and to compensate for that, I multiply it by an additional term of 2 to the power minus 1 and you can readily see that 0.1 binary is the same as 1.0 into 2 to the power minus 1, ok.

So, this is getting closer to the form that I want. In fact, I can now read off the I triple E floating point values for s, e and f from this, directly. I will notice that s, this is the I triple E floating point representation. So, s is obviously going to be equal to 0. This is a positive value. I will also notice that f is going to be a lot of 0s, because, I have 1.0 and I want that correspond to 1.f and I can also read off that e is going to be equal to 126, so that, e minus 127 is equal to minus 1. So, 126 minus 127 is equal to minus 1. Therefore, I know that, s is 0, e is 126 and f is a lot of 0s. Why am I showing a lot of 0s here, because, I know that, in the I triple E floating point representation, the f field contains 23 0s. So, the f field must have 23 0s in it.

Now, I now know, what half looks like in the I triple E 754 representation. All I have to do now is, I take the s and make it the most significant bit. I take the value of e and write it as the next 8 bits. So, this is 126 in binary and then I take the 23 f bits and put them at the end and this 32 bit value is the I triple E floating point representation for half. It is the exact I triple E floating point representation for half. Now, there will be many real values for which there is not an exact I triple E equivalent and therefore, what is done in the I triple E standard is, to approximate a real value which cannot be represented exactly by its nearest equivalent.

(Refer Slide Time: 27:35)



Example: IEEE Single Float.

32bit IEEE single float 0xBDCCCCCC

1011 1101 1100 1100 1100 1100 1100 1100

1 01111011 100 1100 1100 1100 1100 1100

Sign bit: 1 Negative value

Exponent field: 123 Exponent value: 123 -127 = -4

$- 1.100\ 1100\ \ 1100\ 1100\ 1100\ 1100 \times 2^{-4}$

$2^{-3} \times 0.11001100110011001100 1100$

Answer: -0.1 decimal

Let us just look at one more example. This will also help us to get a clearer, better understanding of the hexadecimal digits. So, what I am going to do in this example is, I am, I am showing you a value, which has been represented in the I triple E floating point representation. I am showing it to you in hexadecimal. Remember, 0x prefix lets you know that, what follows is in hexadecimal and this hexadecimal number is BDCCCCCC. Remember, that B, D and C are all hexadecimal digits. A is the hexadecimal digit corresponding to 10, B is 11, C is 12, D is 13. Hexadecimal is the base 16 number system.

So, this is a 32 bit value. How do I know that it is 32 bits? Remember, that each of these hexadecimal digits actually corresponds to 4 bits, right? And, there are 8 of them. Therefore, this is a 32 bit value. Now, how do I evaluate this I triple E value, I triple E single, floating point value in in thei tri[ple]- in terms of a number, that I can understand? The answer is, I need to find out what is the value of s, what is the value of e and what is the value of f.

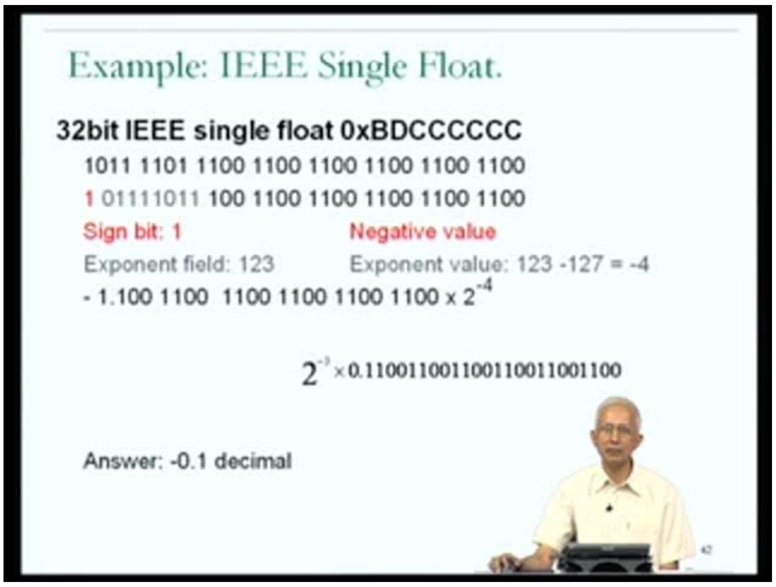
So, I need to break this up in to its components. And, for that I will need ready reference to the hexadecimal table. Remember that B, as I said corresponds to 1010 or 10, C corresponds to 1011 etcetera. So, it is useful to have this table in front of you, until you become really familiar with a hexadecimal notation. So, what I do, in order to expand this, I take the B and I replace it by 0 1 etcetera. So, I get 00,1011 for B and I get 1101 for that D and for each of the C's, I get a 1100. So, these are the 32 bits.

Now, once I have expanded the 32 bit value from hex into binary, I can now read off the sign bit, the bits corresponding to the exponent and the bits corresponding to the fraction. The most significant bit is the sign bit, which is a 1. The next 8 bits are the exponent fields and the remaining 23 bits are the fraction field. So, I read this off as the sign bit of 1, which means this is a negative value. I can now read the 8 exponent bits. I find out that, the value and I evaluate that in binary. I find out that it is the value 123. I bear in mind that, in the I triple E representation 2 is raised to the power e minus 127. Therefore, I will quickly remember that, the exponent value is actually going to be 123 minus 127 or minus 4. In other words, in the value I am going to have 2 to the power minus 4. I then have to read out the fraction bits.

So, I put 1 in front of it. Remember that its 1.f and I interpret the value as minus 1 point, the fraction bits 1001100 etcetera, multiplied by 2 to the power minus 4. and I could evaluate this and if I evaluated it, I would find out that it was equal to minus 0, approximately equal to minus 0.1 decimal. Therefore, this I triple E float, I triple E floating point value corresponds approximately to minus 0.1. It does not correspond exactly to minus 0.1, but it is the closest I triple E value to minus 0.1 and therefore, if you have a program which sets a variable to minus 0.1 and then you looked at that particular variable in its binary form, you would probably find this, as being the representation. Ok.

So, I am suggesting to you that, it is possible for you to look at the I triple E floating point representation of a value inside a program of yours and if you think about this a little bit, it may not be that difficult. Let us suppose that, you have a program in which you have a variable declared as a float, which means that it is going to be represented using the I triple E single precision floating point representation.

(Refer Slide Time: 27:35)



Example: IEEE Single Float.

32bit IEEE single float 0xBDCCCCCC

1011 1101 1100 1100 1100 1100 1100 1100

1 01111011 100 1100 1100 1100 1100 1100

Sign bit: 1 Negative value

Exponent field: 123 Exponent value: 123 -127 = -4

- 1.100 1100 1100 1100 1100 1100 x 2^{-4}

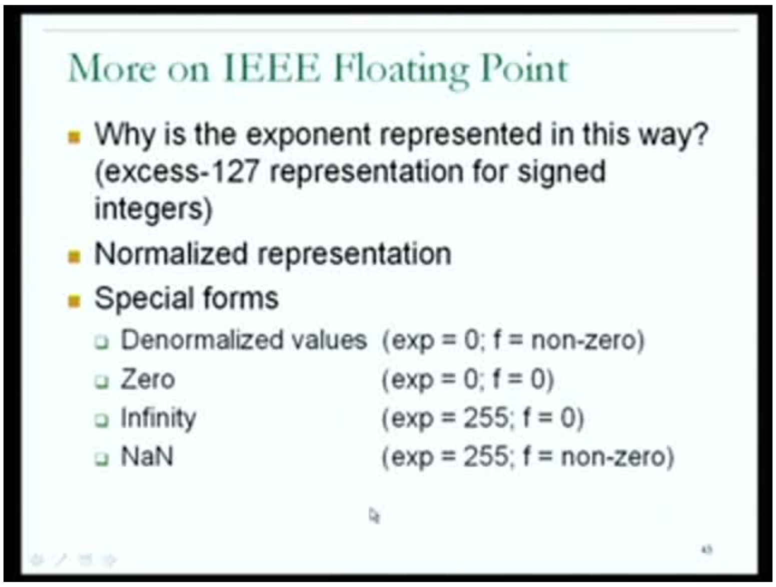
$2^{-3} \times 0.110011001100110011001100$

Answer: -0.1 decimal

So, I have a variable called float of x. So, in that, in the program I could actually print out the value of x in hexadecimal form and had I printed out the value of x in hexadecimal form, I would possibly find out that it will look something like this. So, there may be ways to write programs, in which you can examine the binary representation of a float and that you could write programs to do this. By the same token, it is conceivable that you could write programs, to look at the representation of a signed integer variable or to look at the representation of a character variable and check whether what I told you about the ASCII code and the 2's complement code in the I triple E single precision, are all correct. Ok.

So, one can, can actually think about a lot of what I am talking about, as things that can be experimentally verified on the computers that you have access to. And in general, one should think about high performance computing as being an extremely experimental discipline. Most things are going to be studied by writing programs. A lot of things can be learned, experimented with or confirmed by writing programs and learning them on the computers that you have access to. Ok.

(Refer Slide Time: 32:54)



More on IEEE Floating Point

- Why is the exponent represented in this way? (excess-127 representation for signed integers)
- Normalized representation
- Special forms
  - Denormalized values (exp = 0; f = non-zero)
  - Zero (exp = 0; f = 0)
  - Infinity (exp = 255; f = 0)
  - NaN (exp = 255; f = non-zero)

Now, I had actually asked along the way about a few of the oddities of this representation. For example, the exponent is represented in a very strange way. Remember, the exponent in the I triple E standard was represented in a form, such that you had to subtract 127 from it, in order to get the actual value of the exponent. In this particular example, the value is exponent was minus 4. In other words, the value, the exponent was a negative value and I had to, I represented it by the value e equal to 123.

So, this gives us some hint as to why the I triple E, designers of the I triple E floating point representation may have used this very strange notation for the exponent. The first thing to bear in mind is that, in general, for floating point- for to represent real values, the exponent would have to be a signed integer. For very small values, the exponent is going to be negative.

For very large values, the exponent is going to be positive. Therefore, in general, the exponent value is going to be a signed integer. Now, it would have been possible to represent the exponent value using a 2's complement representation or the Sign-magnitude representation, but designers of the I triple E standard chose to use something different. They used something called the excess-127 representation. The representation that we saw, where the value 4 is represented by 123 because, 123 minus 127 is equal to minus 4 is known as the exces-127 representation and that is what is used for the exponent in the I triple E standard.

So, the question is, why do they use the excess-127 representation instead of using something else? And the answer is that, in the excess-127 representation, it is much easier to compare values. It is easier to determine whether one value is positive and one value is negative and to compare values then with signed magnitude representation or the 2's complement representation.

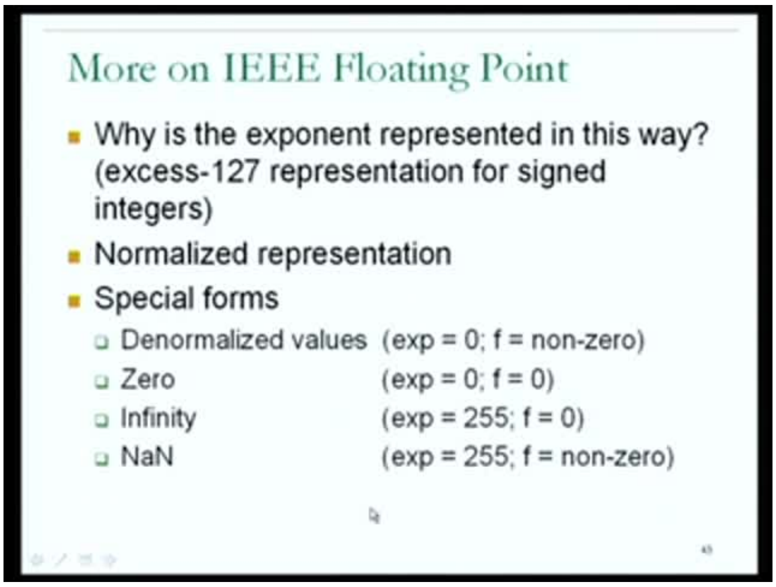
The other interesting question which arose was, why do they use that very strange representation for the fractional part, where they have 23 bits, a value inside the representation, but then they assume that there is a 1.f. And that is what is known as a normalized representation. Now, the reason for the normalized representation is, let us suppose that, they did not have the idea 1.f, minus 1 to the power s multiplied by 1.f multiplied by 2 to the power e minus 127, but instead they had used 0.f.

Now, if they had used 0.f, just note that, inside the fractional field of 23 bits, it is possible that, the first bit could have been a 0 or the first 5 bits could have been 0s. What this means is that, there would have been many different ways to represent the same real value. I could have represented the real value half, as 0.1 into 2 to the power of 0 or I could have represented the value half, as 0.1, as 0.01 into 2 to the power of 1 or I could have represented it as 0.001 into 2 to the power of 2 and so on.

So, there would have been many different possible ways to represent the same real value. What this would have meant is that, there would have been many less values that could be represented using the float, the floating point representation. So, to overcome these problems, they used this normalized representation, where they have 1.f.

So, there is only one way to represent the value half, as 1.000 multiplied by 2 to the power of minus 1 and so on. Therefore, the normalized representation is used to provide a unique representation and that is why, we have 1.f and not 0.f. Just for your additional information, I would like to mention that, there are some other features of the floating point, I triple E floating point representation, which might be of interest to you and they are known as the special forms.

(Refer Slide Time: 32:54)



More on IEEE Floating Point

- Why is the exponent represented in this way? (excess-127 representation for signed integers)
- Normalized representation
- Special forms

Denormalized values	(exp = 0; f = non-zero)
Zero	(exp = 0; f = 0)
Infinity	(exp = 255; f = 0)
NaN	(exp = 255; f = non-zero)

Now, some of the special forms may actually have become clear to you, if you think a little bit about, for example, how could you represent 0 in the I triple E floating point representation? Now, there is a problem in representing zero in the I triple E floating point representation and let us, go back a little bit, just think about representing zero in the I triple E floating point representation.

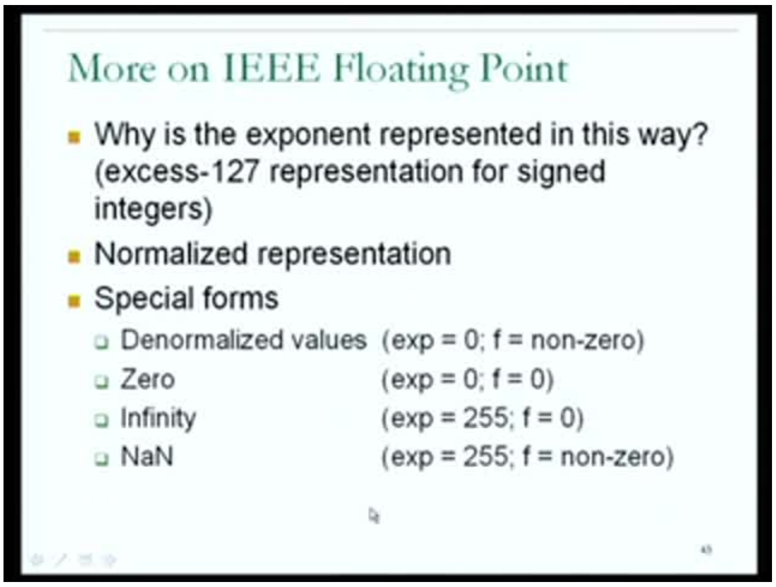
Now, I can represent plus 0 by making the sign bit equal to 0, but there is no way for me to prevent this 1 point, from creating some magnitude as far as the value is concerned. And therefore, there is in fact, no way to represent 0 precisely, using this expression. That is one of the problems with the I triple E floating point representation, which has to be overcome by a special form. In other words, the 0 must be treated, excuse me, as a special case. 0 is not representable exactly using the expression that we had. You should recall that, we might represent 0 approximately by a very small value, but typically in computations that we do, it might be important to represent at least 0 exactly, which is why 0 might have to be treated as a special case. And there are a few other special cases which the designers of the I triple E standard have included. One of these special cases is, something known as the de-normalized values. And basically, the de-normalized values are a special case, through which one can represent extremely small values. The other special case that is allowed is, a representation of 0 and 0 is represented by using an exponent value of 0 and a fractional value of 0 along with a sign bit of either 0 or 1.

So, you can represent both plus 0 as well as minus 0. An additional special case, which is taken into account is, a precise representation of infinity. So, if the exponent field has a value of 255, decimal 255 and fractional field has a value of 0, that corresponds to the value infinity. So, you can represent both plus infinity and minus infinity, by using the appropriate sign bit.

There is also representation for something called not a number, which is written as N a N and this is the value, which may result from certain types of computation. For example, if you have a computation in which, you divide 1 by 0, integer 1 by, I am sorry, floating point 1 by floating point 0 and technically one may not want to view this, as a meaningful value and describe it as something, which is not a number. Therefore, in hardware, which is built using the I triple E floating point standard, such a computation may result in a value, which is called not a number, which has an explicit representation for it.

Now, as far as the special forms are concerned, it should be taken into, you should note that, the I triple E standard defines not only ways to represent these special case values, but also for arithmetic to happen on these special case values. So, for example, if I, if I have a varying floating point variable called x and a floating point variable called y and during the execution of my program, x takes on a value which happens to be infinity and I have an expression in my program, which adds x to y. It would be useful, if my program does not, is not, first to halt because of an error, but rather is allowed to continue to execute, because, the meaning of x plus y is, is contained inside the floating point representation. For example, if it is understood that infinity plus a normalized value is actually infinity and that the hardware will do this computation, then it is possible for programs which generate not a numbers or infinities, to continue to execute and produce results that may be of value to the person, who wrote the program.

(Refer Slide Time: 32:54)



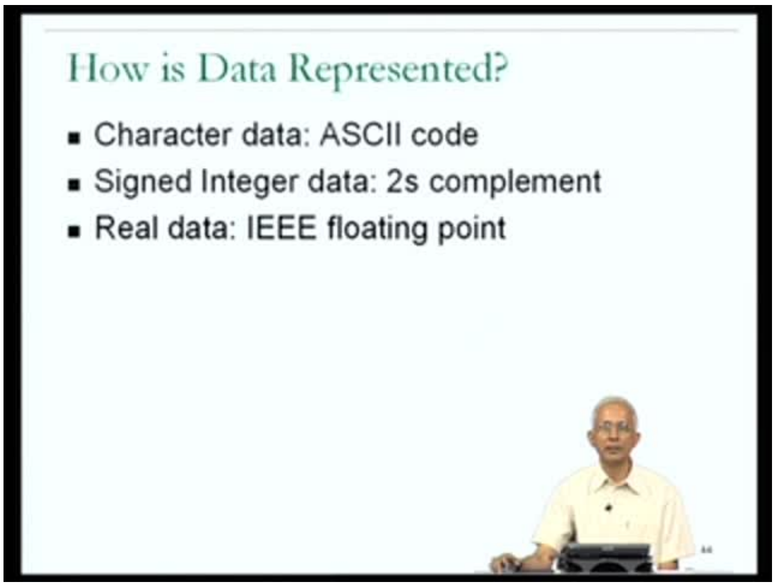
More on IEEE Floating Point

- Why is the exponent represented in this way? (excess-127 representation for signed integers)
- Normalized representation
- Special forms
  - Denormalized values (exp = 0; f = non-zero)
  - Zero (exp = 0; f = 0)
  - Infinity (exp = 255; f = 0)
  - NaN (exp = 255; f = non-zero)

Therefore, in some situations, it might be useful for you to learn more about the special forms, but in general, for many of the programs that you may write, the normalized form as well as the representation for 0 might be the only aspects of the I triple E floating point standard that you encounter. But as I had said before, also bear in mind that, it may be useful for you to be able to write a program, which could determine whether the hardware on which you are running the program, is running using I triple E floating point standard. And in order to write such a program, it may be useful for you to recap what we have said and talked about in this lecture.

You will, to write such a program, you would have to know, how the I triple E floating point representation represents the value minus 1 to the power s, multiplied by 1.f, multiplied by 2 to the power e minus 127. And then, you would have to write the program in such a way that, it looked at the bits inside the 32 bit float in order to determine whether the value corresponds to or it should have been under the semantics of the program. So, once again, thinking of writing programs which do things on the system, in order to understand more about what happens during the execution of the program.

(Refer Slide Time: 42:20)



How is Data Represented?

- Character data: ASCII code
- Signed Integer data: 2s complement
- Real data: IEEE floating point

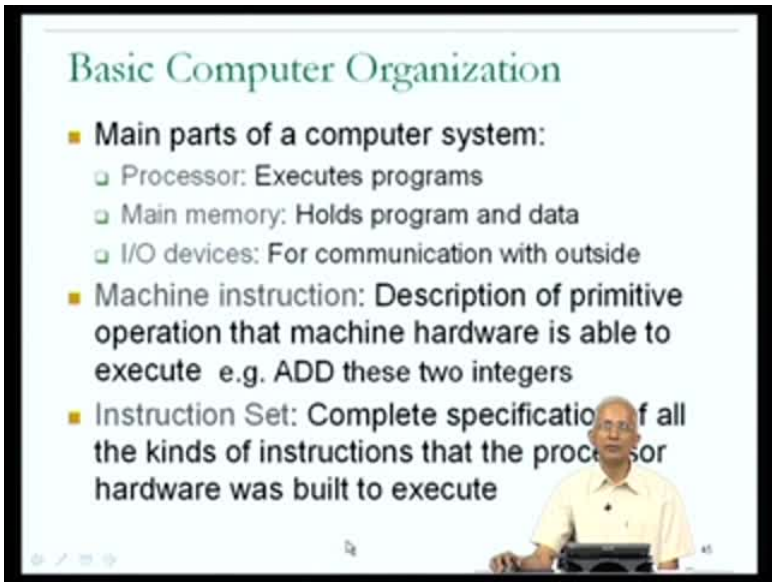
So, with this we have understood more about, how real data is represented and in short, we have understood the representation in most computer systems, of the three main kinds of data - character data, int signed integer data, as well as real data. Now, character data will be used to represent characters. The characters will be declared as characters. You might have arrays of characters which end up finding use as strings.

So, lot of what we have said may have relevance, as far as string data is concerned. Remember that, the real data I talked about, refer to the I triple E floating point. The I triple E floating point that I talked about, was the 32 bit I triple E 754 standard. You may be aware that in C programs, you can also talk about double floats. For example, you could declare variable as double x semicolon.

Now, that is the situation, where you want to have more precision inside the representation of the real value and that is not represented using the 32 bit I triple E 754 representation that we saw, but rather using a, a larger representation called the I triple E 8, which is covered by another I triple E standard.

So, there are representations for more precise real values as well and that is what would happen, if your program was written to use a double float or a double value. So, we have not covered all the aspects of characters, integers and real, but we have a very good understanding to start with.

(Refer Slide Time: 43:54)



Basic Computer Organization

- Main parts of a computer system:
  - Processor: Executes programs
  - Main memory: Holds program and data
  - I/O devices: For communication with outside
- Machine instruction: Description of primitive operation that machine hardware is able to execute e.g. ADD these two integers
- Instruction Set: Complete specificatio f all the kinds of instructions that the proc sor hardware was built to execute

So, with this, we are ready to move into the computer and for the rest of this lecture, I would like to give a quick overview of what to expect, when we actually start looking at the innards of the computer. This is something, which you may have seen, again in school, in your, earlier in your under graduate education, but it is useful to start from scratch to make sure that, we are using the same terminology.

So, I am going to start here, with few lectures on basic computer organization. Now, as you have probably heard, there are 3 main parts to a computer system. One of the more important, most important parts of the computer system is, what is called the processor or the CPU or central processing unit and it is the processor which essentially executes your programs; the processor, which contains a lot of circuitry which is capable of executing the instructions which you have used to describe the algorithm that your program is implementing. Second part, main part of a computer system is something called the main memory and basically, the main memory is the circuitry of the computer system, which holds the program and its data.

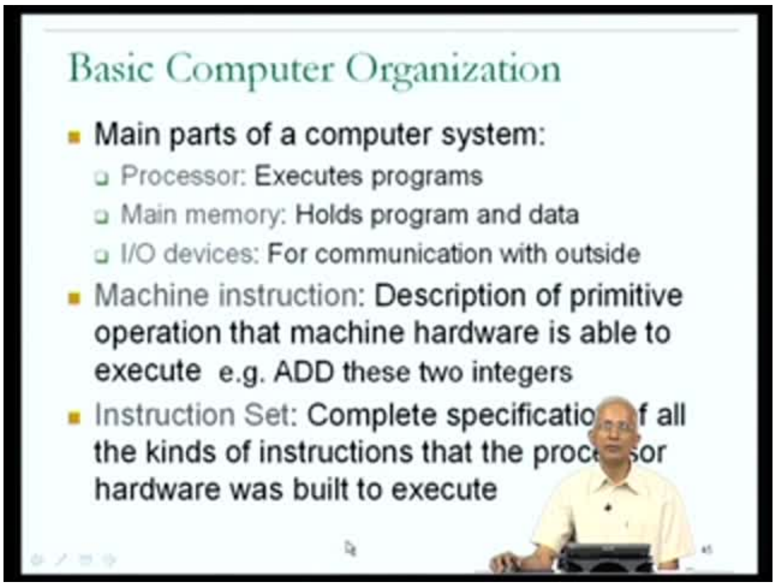
We have seen that, a program in execution is made up of many instructions, but it also manipulates data and some of the data may have been statically allocated. Some of the data may have been heap allocated, some of the data may have been stack allocated, depending on the lifetime of that data, but all of that data and all of the instructions must be present inside the computer, in order for the instructions to be executed by the processor and therefore, the main memory is very important part of a computer, because, that is where the program and its data reside, when the program is executing.

The third and another important part of a computer system, is the different I/O devices or the input and output devices, that are present in the computer system and they are important, because, they provide for communication with, between the inside of the computer and the outside world. And, very clearly, in general, for the programs that you write, if you could not communicate with your program, then it is not clear what kind of objectives could be achieved. For example, if your program was multiplying matrices, it is possible that, you would want to see the outcome of that multiplication and would therefore, need to, not just have the result of the multiplication available in memory, but also have it made available to you, possibly on a displayed device or printed on a printer, etcetera.

So, there are many different kinds of input and output devices, input devices for you to interact with the computer and output devices for information to be output from the computer. So, these are the 3 main parts of a computer system and when we talked about the program, I talked about the fact that, program contains a description of algorithms, initially programmed by you, in a language such as C, but ultimately, compiled down and present in the a dot out file in the form of machine instructions, in other words, in the form of primitive, a very primitive form, that the machine hardware is able to understand and execute. So, a machine instruction is, what I mean by the word machine instruction is, a description of a primitive operation that the machine hardware is able to execute. An example of a machine instruction might be an instruction, that is, capable of adding 2 integers.

So, that might be an example of machine instruction and we, we expect that, as part of the processor, in order to do that addition. There might be a small circuit and if the integer, if the addition is of 2 signed integers represented in 2's complement, then the circuit is going to be a circuit that is capable of adding signed complement, 2's complement signed integer values.

(Refer Slide Time: 43:54)



Basic Computer Organization

- Main parts of a computer system:
  - Processor: Executes programs
  - Main memory: Holds program and data
  - I/O devices: For communication with outside
- Machine instruction: Description of primitive operation that machine hardware is able to execute e.g. ADD these two integers
- Instruction Set: Complete specificatio f all the kinds of instructions that the proc sor hardware was built to execute

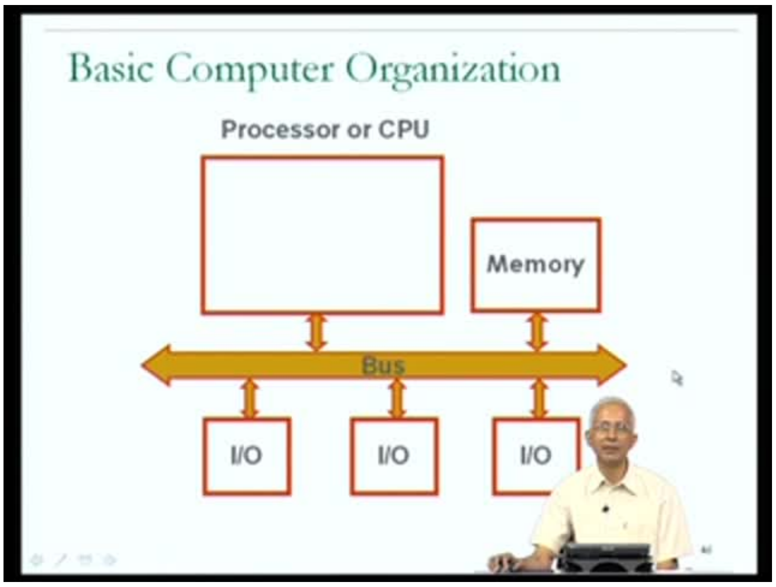
Similarly, if the, if there may be separate instructions to add 2 32 bit floating point values and corresponding to that instruction, there might be another piece of hardware, another circuit inside the processor, which can do the needful, as far as that instruction is concerned. So, there is going to be a correspondence between the different machine instructions and the different pieces of hardware inside the processor.

Now, of interest to us is, not just one instruction such as the add instruction, but possibly all of the instructions, that a particular processor is capable of executing. And, I will use a term instruction set to refer to the complete specification of all the instructions, that a particular processor, hardware was built to execute.

So, for example, if we were studying a particular processor, manufactured by a particular manufacturer, then I might want to look at the book or the manual, containing a list of all the different instructions that processor is capable of executing, because, I know that my program ultimately will get compiled into those instructions.

And, it might be that, there are several 100 instructions, different instructions that a particular piece of hardware today, is capable of executing. So, as processors have become more complicated and the needs of programs have become more sophisticated, instructions sets have become more and more interesting. Of course, we will not be studying any large instructions set in detail, in this course, but I would like to go through a simple instruction set, so that we get some idea about, what a set of instructions in a real computer might be like.

(Refer Slide Time: 49:31)



But, just starting off with a pictorial understanding of what is happening, the central and most important part of the computer system clearly, must be viewed as the processor or the CPU, because, this is where the hardware, the key hardware for executing instructions and various other activities is contained. Another very important part of a computer system, is the main memory, because, this is where the program instructions and data reside while a program is being executed. The I/O devices are also very important, because, as I said, without them you cannot interact with your program, you cannot input data, you cannot see the results of the execution of your program. Now, these are the different components of the computer system, but they must obviously be connected to each other. It is not just enough to have a processor and a lot of memory and a lot of I/O device, devices in a box together, if they are not connected and able to interact with each other, then, they cannot together cause the execution of a program.

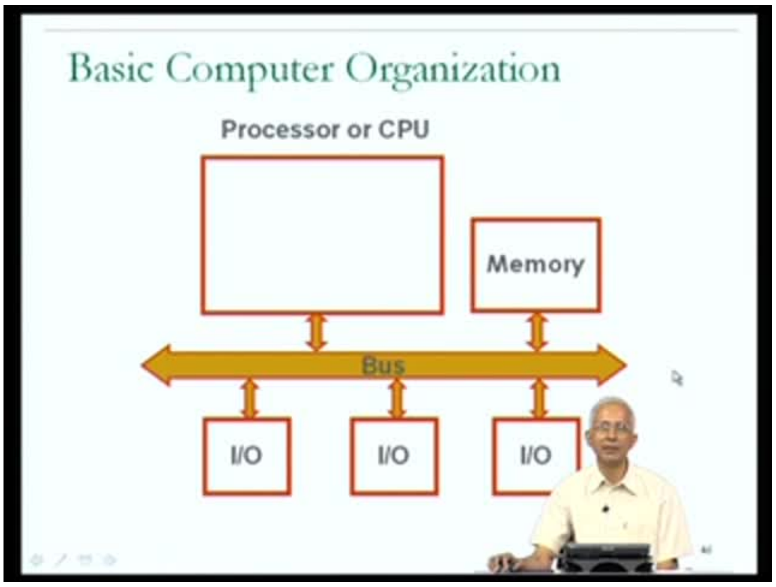
So, there must be some connectivity between them and I am going to show that here, by a central box of connectivity, which I am just labeling as bus and that just refers to some mechanism for connection between the different I/O devices, the memory and the CPU. So, we get an idea that this is an integrated whole, and that, if the instructions are initially available in the memory, they might move over the bus into the processor, in, in order to be executed and that if a piece of data is currently available at a particular memory location, if it is needed as the input of a particular add instruction, let us say, it may move over the bus into the processor, in order to reach the adder and get added. An additional complexity, which we will learn more about later, is, that before my program starts executing, in other words, before I request the computer system to execute the program, the program is not present in memory at all. Where is the program present?

If I have to type the program into a file and compiled it then, both the hello dot c, the C source code, source program file as well as the executable object file a dot out will be present on a disk and a disk is an example of an I/O device. So, if I have written a program and compiled it, then the a dot out corresponding to my program will be present on one of the I/O devices called the hard disk. When I give a command to the computer system, to execute the program, I expect, that components of that a dot out file get loaded into memory, into this main memory and that from this main memory, the instructions get transferred into the processor as needed, in order for the program to be executed.

So, this integrated whole is what we have to understand. Not just the processor, not just the memory, not just the I/O devices, but how they interact, in order to achieve the execution of a program, that I may have written. But, in order to understand the interaction, we have to start by understanding the individual components. There is no other way to do that.

So, we are actually going to start, by trying to understand each of these components, one by one, to a certain degree. And, that is the objective of basic computer organization. Initially, our interest would be on the processor. Now, I am, in your experience, you would have seen that, there are many different kinds of processors. For example, a lot of you have PCs and by default, there is a particular kind of processor inside a personal computer today. Whereas, in your, in a college or an institution, you may have come across other machines, which are meant for other purposes and have different processors in them.

(Refer Slide Time: 49:31)



And, you would, you would expect that different kinds of processors may have different instruction sets and that, therefore, they may have fundamentally different capabilities, as far as executing programs are concerned. For example, a particular processor may have a particular instruction set and therefore, will not be able to execute an a dot out which has been generated for some other processor.

So, we must bear in mind, that, there are different kinds of processors. Each processor, different kinds of processors may have different instruction sets and that they may not be compatible with each other. It is possible that, some processors may be compatible with each other, but that others may not and that, in general an a dot out which is, which was created for one particular processor may not be able to execute on another processor, but it is, in this course, we are just going to study, one particular kind of, simplified version of a commonly occurring processor, so that, we will have a good understanding about what happens in most situations, but it should be clearly understood that you may have to recompile a C source program, in order to make it execute on different programs.

Now, we will actually start looking at the more detailed composition of the processor in our next, in our next lecture. For today, let me just recap what we have learnt in today's lecture. We learnt, we, we recapped our previous lecture in which we had learnt about how character and integer, signed integer data are represented. We then went ahead to look at the I triple E floating point representation for real data. We looked at it in some detail. The real, the representation for real data is fairly important, because, a lot of the computation that you do in, in scientific or engineering applications will actually be in terms of real data. After this, we started looking a little bit at the organization of a computer and understood the basic 3 components. That is, there is processor which is a piece of hardware, which is capable of executing instructions, that there is a main memory, which is where the instructions and data of a program reside, when the program is in execution and that there are I/O devices, through which interaction with the outside world happens. In our next lecture, we will look in more detail at each of these components, starting with the processor. Thank you.