Computer Architecture

Prof. Smruti Ranjan Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 04
The Language of Bits Part-III

(Refer Slide Time: 00:26)

So, now we have taken a look at Boolean algebra which is the manipulation of bits. We have taken a look at how positive integers are represented in binary, how negative integers are represented in binary. So, let me maybe just summarize. At this point, so we have for a quick recap Boolean algebra had to do with Boolean variables, so each variable could take a number between sorry not between a number which is either 0 or 1. And we defined basic operations on Boolean variables such as the AND operation which means that the AND function is true only when both the variables and all the variables involved in AND operation, all of them are 1. And the OR operation is correct only when is true only when one of the variables is 1. Then we define a NOT operation that is a compliment, and we define an XOR operation which is 1, when 1 is 0 and other is 1.

Then we define the binary system to represent positive integers with different basis. So, the important point to recapitulate you know for at this point of the lecture is that what is the relationship between binary between octal which is base 8 and between hex which is

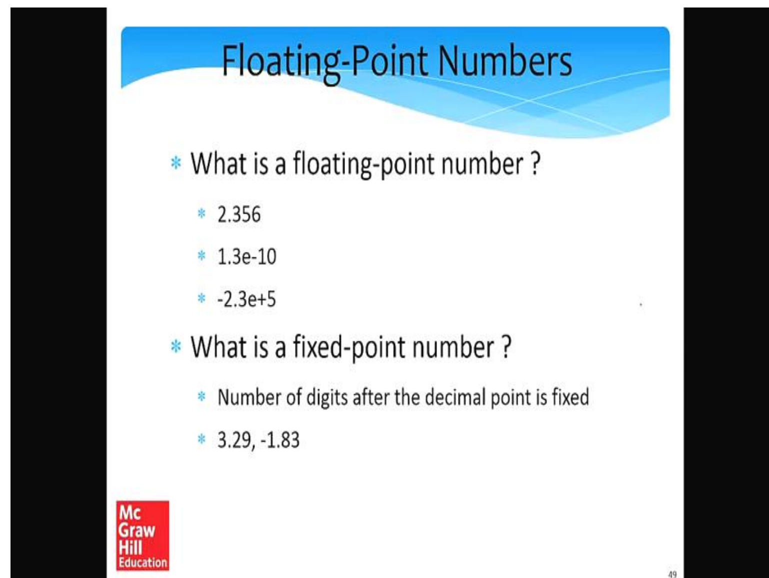
hexadecimal which is base 16. And the important result that we had proved is that the octal and hexadecimal notations can be thought as short forms you know abbreviations or reduced representations of a binary representation.

So, what do we do for example, in base 16, we these group blocks of four binary numbers; and represent them with one base 16 number. For example, a number of this form, which can be represented as this becomes 1 and this is actually 8 plus 3 - 11 which is B right. So, the instead of having 8 digits you only have 2 digits. Then we represented negative integers we looked at different methods of representation; one is a sign magnitude representation, which has a sign bit the problem there was that performing arithmetic operations was difficult. And there were two representations was zero; the other is ones complement same problems two representations.

The other was with a bias; so in with a bias we had a single representation for every number, but even addition and subtraction were relatively easier, but multiplication was hard, so that is the reason we introduced 2s complement. Where essentially the 2s complement of a positive number is the number itself; and a 2s complement of a negative number is 2 to the power n minus u. The fantastic thing about this number system is yes it preserves a notion of the sign bit, it is easy to find if a number is positive or negative just take a look at the m s b there is one representation for zero and all other numbers. And performing addition, subtraction and multiplication is simply as simple as take the unsigned representations of the numbers and add, subtract and multiply them. The last n bits of the result are the representation of; the last n bits of the result are actually the representation of what the actual result should have been right.

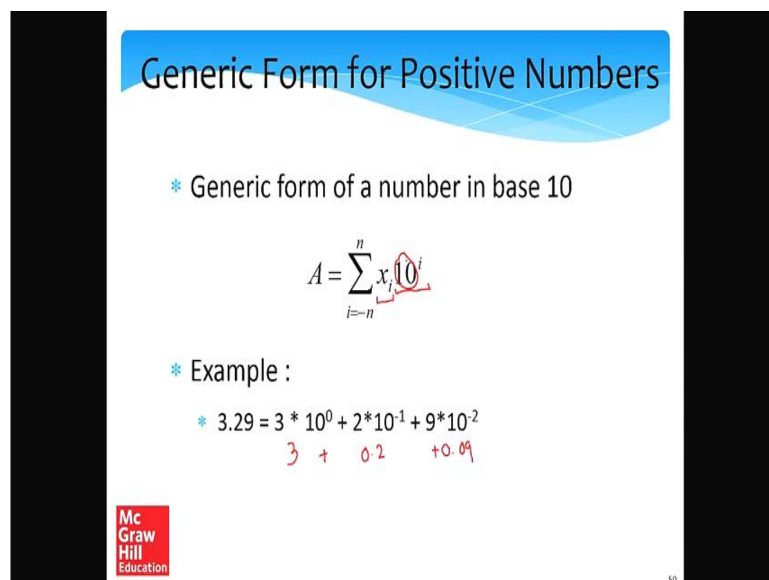
So, then we discussed some properties of this numbers system which is when what are the conditions for an overflow that is very important. And we also discussed the issues of sign extension which is that when you convert a number from one you know from an n bit system to an n bit system, where m is greater than n what do you need to do.

(Refer Slide Time: 05:09)



Now, let us discuss floating-point numbers, which are numbers the decimal point. So, what is the floating-point number is any number with a decimal point inside it like 2.356 or 1.3 times 10 to power minus 10. So, now the question is the moment we say that a number is a floating-point number, we need to answer the question what is the fixed-point number. So, in a fixed-point number, the number of digits after the decimal point is fixed. So, one example of such a number is 3.29. So, any number representing let us say currency is a fixed-point number, the number of digits after the decimal point is fixed

(Refer Slide Time: 05:51)



So, what is the generic form of a number in base 10, if we consider just positives numbers. If we just consider positives numbers, the generic form of a number in base 10 is 3 times 10 to the power 0, which is 3 plus you know 0.2 which is 2 times 10 to the power minus 1 plus 0.09 which is 9 times 10 to the power minus 2. Say, any generic form is a form of this type where we make a summation from a minus n to a plus m or plus n where we are essentially adding powers of 10 with a coefficient. So, in this case the coefficient is 2 and 9 and 3 and so on. So, what would a generic form of a number being base 2, all that we can do is that we can take 10 and replace it with 2.

(Refer Slide Time: 06:55)

Generic Form in Base 2

* Generic form of a number in base 2

$$A = \sum_{i=-n}^{n} x_i 2^i$$

Number	Expansion
0.375	$2^{-2} + 2^{-3}$
1	2^0
1.5	$2^0 + 2^{-1}$
2.75	$2^1 + 2^{-1} + 2^{-2}$
17.625	$2^4 + 2^0 + 2^{-1} + 2^{-3}$

Mc Graw Hill Education

51

So, let us take some simple numbers and expand them in this notation. So, let us say 0.375 is 2 to the power minus 2 plus 2 to the power minus 3. A number like 1.5 is 2 to the power 0 plus 2 to the power minus 1; a number of the form 17.625 is this representation as shown over here 2 to the power 4 plus 2 to the power 0 plus 2 to the power minus 1 plus 2 to the power minus 3. So, if we can use base 10 to represent all floating-point numbers right all decimal numbers like the value of pi and e and even a number like 5.36 times 10 to the power minus 17 nothing stops us mathematically to replace base 10 by base 2.

(Refer Slide Time: 07:55)

Binary Representation

- Take the base 2 representation of a floating-point (FP) number
- Each coefficient is a binary digit

Number	Expansion	BinaryRepresentation
0.375	$2^{-2}+2^{-3}$	0.011
1	2^{0}	1.0
1.5	$2^{0}+2^{-1}$	1.1
2.75	$2^{1}+2^{-1}+2^{2}$	10.11 ⟷ 2+0.75=2.75
17.625	$2^{4}+2^{0}+2^{-1}+2^{3}$	10001.101

So, what do we do we take the base 2 representation of a floating-point number. So, what we have done here is that we have replaced 4, I am sorry replaced base 10 by base 2 everywhere just one small correction here this should have actually been minus 3. So, we have replaced base 2 by base 10 everywhere. And so as I said if you go to a planet where people have only 2 fingers they would use base 2, they would not use base 10, and a pretty much any base 10 number has an equivalent base 2 expression, which is an expression of this form which is just a binary representation of the number from base 10 to base 2.

And so if you consider 10.11 in decimal this would be 2 plus 0.1 is 2 to the power minus 1 which is 0.5 plus 0.25 which is 2 plus 0.75 or 2.75. So, this gives us a very easy way to at least represent positive numbers with by just simply extending the logic and instead of a base 10 using base 2.

(Refer Slide Time: 09:16)

Normalized Form

- Let us create a standard form of all floating point numbers

$$A = (-1)^S * P * 2^X, (P = 1 + M, 0 \le M < 1, X \in Z)$$

- S → sign bit, P → significand
- M → mantissa, X → exponent, **Z** → set of integers

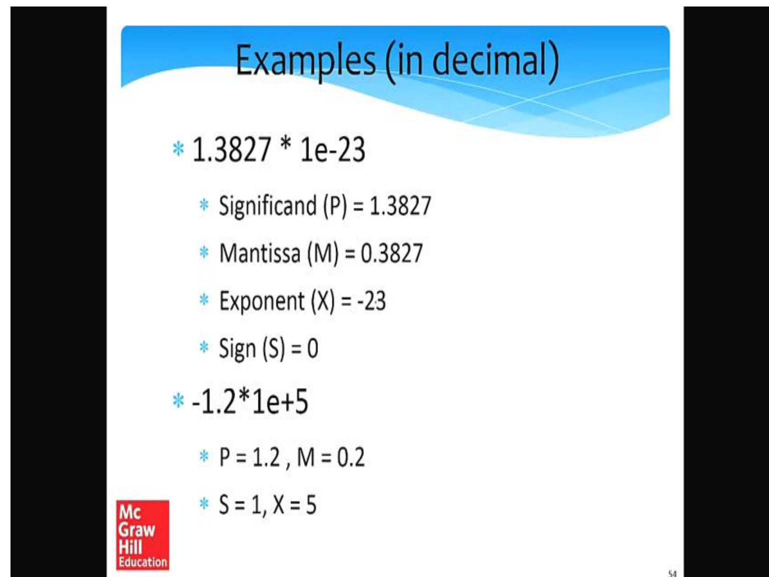
So, let us now create a standard form of all floating-point numbers. So, in this case, we will use the sign magnitude representation, because otherwise that will become too complicated. So, any number A can be represented as minus 1 to the power S, where S is a sign bit. If S is 0 then minus 1 to the power S is 1; and if S is plus 1, which means the number is negative minus 1 to the power 1 is minus 1, so number becomes negative. Then we can define P which is a significand, so the significand is typically a number between 1 and 2. In this case, not typically it is a number between 1 and 2

So, we can represent the significand as a number as 1 plus M. So, 0 is less than equal to M and M is less than 1 multiplied by 2 to the power X, where X is an integer right. So, X is element of Z, which is the set of integers and the significand. So, let us just go over this terminology once again is very important say any floating-point number, we are representing as minus 1 to the power S, where S is the sign bit multiplied by the significand, the significand is the number of the form 1 plus M, where M is strictly less than 1 and it is positive M is called the mantissa. So, the M is given a name and the name of this term is the mantissa. Multiplied by 2 to the power X, where X is the exponent and X is an integer

So, as we see any numbers can be represented in this form there is absolutely no problem. Like a number of the form 2.6 can be represented as minus 1 to the power 0 multiplied by 1.3 multiplied by 2 to the power 1, where 1.3 is a significand which is 1

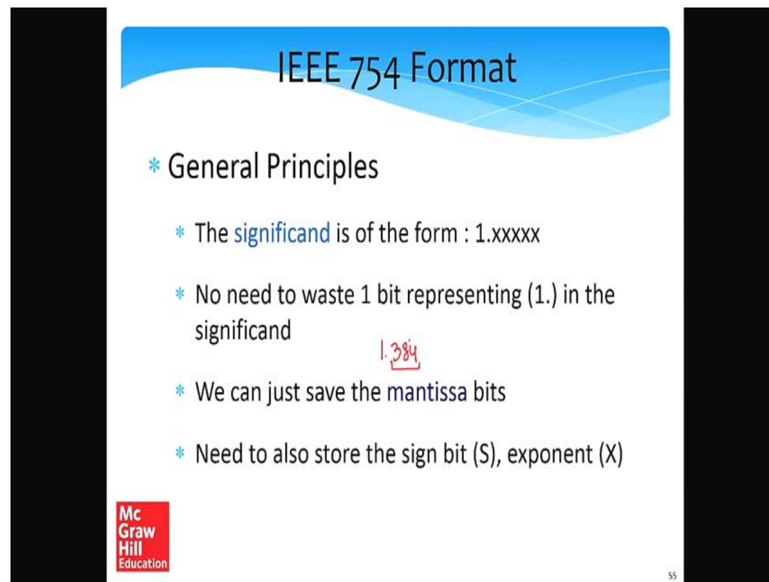
plus 0.3 where 0.3 is the mantissa. So, this is also called the normal form or the normalized form.

(Refer Slide Time: 11:38)



So, let us see some examples at least in decimal, we will go to binary later. So, 1.387 times 10 to the power minus 23. So, the significand is 1.3827, Mantissa is 0.3827. Since we consider in decimal instead of base 2, it is base 10, so the exponent is minus 23 and the sign bit is 0. Similarly, 1.2 times 10 to the power 5, the significand is 1.2, mantissa is 0.2; the sign bit is 1, which means is minus 1 to the power 1 at the leftmost point and the exponent is plus 5.

(Refer Slide Time: 12:24)



So, let us now take a look at the IEEE 754 format for representing floating-point numbers, and take a look at the general principles. So, the significand should be of the form one point something. So, there see if you think about it, if we have 32-bits, so this is a 32-bit number system. If we think about it if every number significand is one point something right, there is no need to waste 1-bit representing that 1, we can assume it as the default that is always there. We just need to save the mantissa bits right, for example, if a number is 1.384, we just need to save 384 right need a binary representation for 384, we can automatically assume that 1 is in there. We also need to store the sign bit S, and the exponent X.

(Refer Slide Time: 13:30)

IEEE 754 Format - II

Sign(S) Exponent(X) Mantissa(M)

1	8	23

- sign bit – 0 (+ve), 1 (-ve)
- exponent, 8 bits
- mantissa, 23 bits

So, the IEEE 754 format is something like this it is a 32-bit format. The MSB is for the sign bit where we use one bit for the sign bit or we use 8-bits for the exponent which means it can take 256, values and we use 23-bits for the mantissa.

(Refer Slide Time: 14:02)

Representation of the Exponent

- Biased representation
  - bias = 127
  - E = X + bias
- Range of the exponent
  - 0 – 255 ↔ -127 to +128
- Examples :
  - X = 0, E = 127
  - X = -23, E = 104
  - X = 30 , E = 157

So, let us now take a look at the representation of the exponent. So, the exponent uses the biased representation in the sense that if the exponent is equal to X then we actually saves X plus the bias. So, in this case it allows us to represent negative exponents as well. So, we can actually represent exponents in the range of minus 127 to plus 128 right.

So, since the bias is 127, what we actually save in this case is 0 till 255, which is a total of 256 numbers. So, what was the need for having a biased representation over here well the need was that the exponent can be positive or the exponent can be negative.

Hence, we need to have some kind of a representation 2s complement was found to be bit too complicated, and it was also not required because typically in floating-point numbers, we do not multiply the exponent with some other number. Most of the time you only add and subtract the exponents; in that case it was not necessary to go for something as heavy weight as 2s complement, the biased representation was found to be nice and simple.

So, as I said. So, let us consider some examples the exponent is 0, we actually save 127 if the exponent is minus 23, we save minus 23 plus 127 which is 104; last example is the exponent plus 30, we save 157. So, what are the different fields once again the one bit sign bit is there; after that we have 8-bits, and the 8-bits are for the exponent. But the exponent is actually x it is representation is E, and what is the relationship between E and X, we are saving E, where E is equal to X plus the bias. So, whatever is a real exponent we add 127 to it and we save it in this particular number system.

(Refer Slide Time: 16:21)

Normal FP Numbers

- Have an exponent between -126 and +127
- Let us leave the exponents : -127, and +128 for special purposes.

$$A = (-1)^S * P * 2^{E-bias}$$

$$(P = 1 + M, 0 \le M < 1, X \in Z, 1 \le E \le 254)$$

Mc Graw Hill Education

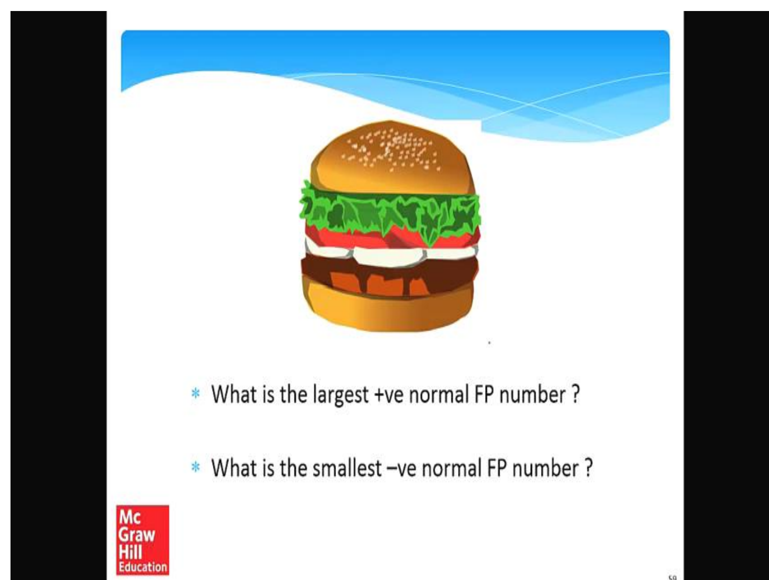
58

Now, let us con consider normal floating-point numbers in the IEEE 754 format. So, IEEE by the way is an Association of Electrical and Electronics Engineers. So, they IEEE typically makes standards and formats for representing a variety of information. In

this case, the 754 format has been made 754 standard has been made to represent floating-point numbers in binary. So, normal binary numbers have an exponent between minus 126 and plus 127. So, the other exponents which can be represented minus 127 and plus 128 are reserved for special purposes so which we will discuss later.

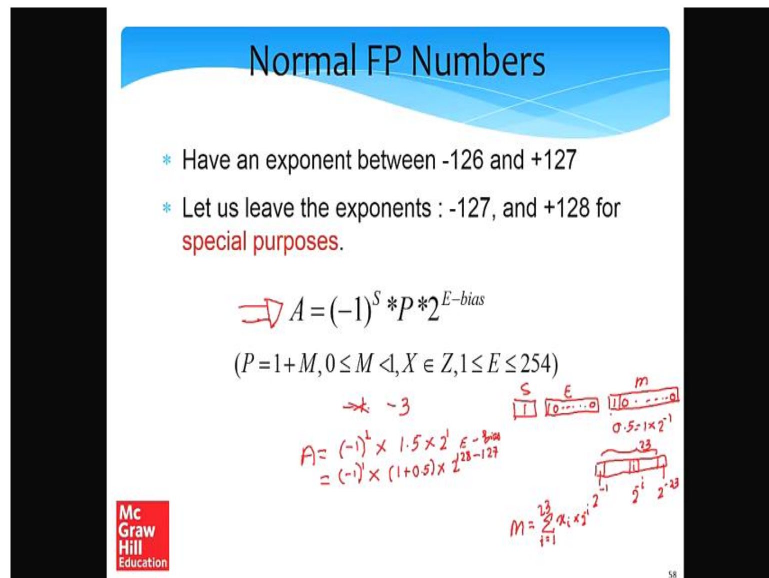
So, now the standard form or the normal form of an floating-point number in IEEE 754 format is exactly what we had seen before and with a little bit of modification. So, we have the sign bit minus 1 to the power S multiplied with the significand, where the significand is of the form 1 plus M, M being the mantissa strictly less than 1. Multiplied by 2 to the power E minus bias where E is the representation of the exponent. So, E is between 1 and 254 right. So, the values of 0 and 255 are reserved for special purposes.

(Refer Slide Time: 18:02)



So, here is the food for thought, what is the largest positive normal floating-point number, what is the smallest negative normal floating-point number. I will not answer them, but I have kept them as exercises for the reader and the answers are in the book.

(Refer Slide Time: 18:19)



So, let us now take a look at some of the special values for the floating-point numbers. So, in this case, so if you would recall the valid values of E or E actually has to be in the range of 1 and 254. So, the other the values that we are excluding as 0 and 255 which basically means that exponents with a value of minus 127 or plus 128 are not allowed. So, if they are not allowed what does it mean it means that we are using these values for denoting something special. So, what should it be what it should be is something like this that if E is equal to 255 and the mantissa is 0, let us treat this number as positive infinity plus infinity if the sign bit is 0. If E is 255, the mantissa is 0, and let us treat another sign bit is 1, let us treat this number is minus infinity. So, let us give an example 1 divided by 0 is plus infinity; and minus 1 divided by 0 is negative infinity; and infinity is represented with the fact that E is 255 and M is 0.

Now, let us consider the case the third case where E is 255 and M is not equal to 0. So, there can be many such values, but we say that all of these values represent an NAN for an NAN is an not a number. For example, what is 0 by 0, 0 by 0 is undefined. So, we treat this case as a NAN or what is log of minus 1, log of minus 1 is undefined. So, let us again treated as a NAN or sin inverse of 5, a sin inverse of 10, it is undefined. So, we treat this as an NAN. Any arithmetic expression that has an NAN will always evaluate to NAN. So, NAN plus NAN is equal to NAN, NAN minus NAN is equal to NAN, NAN plus any other number is equal to NAN. So, the moment any part of an expression evaluates to not a number. The entire expression will evaluate to not a number.

So, in this IEEE format, we sadly have two representations for zero which is not desirable, but this is still there. So, in this case, if E is equal to 0, which is one of the special cases that we had excluded if E is 0, and the mantissa is 0, then the value is 0. So, of course, there are two representations are sign bit can either be 0 or the sign bit can be 1, we thus have two representations for 0 in this particular number system which is not something that which is not much that we can do. So, the sign bit can be 0 or 1, but then the E field has to be all 0s to represent a 0, and the mantissa field also has to be all 0s. And a sign bit can either be 0 and 1, it does not matter it will still be in a both the variants will be treated as a 0.

The last subclass is very interesting is called a denormal number. So, we will discuss this in later slides. In this case, E is equal to 0, but M is not equal to 0. So, we will discuss this case in the next few slides.

(Refer Slide Time: 22:16)

So, let us but before going to denormal numbers, I just wanted to give an example of how we would represent a floating-point number. So, let us maybe consider a number of the form minus 1 point let us consider number of form minus 3. So, minus 3, the first thing would be to reduce it to standard form or a normal form which is minus 1 to the power 1 multiplied with 1.5 multiplied with 2 to the power 1. So, this is equal to minus 1 to the power 1 multiplied with 1 plus 0.5, where 0.5 is the mantissa multiplied with 2 to the power 128 minus 127, where 127 is a bias.

So now, if I consider the representation of this number in binary then the sign bit will be equal to 1, because the number is negative. The exponent right the E field will actually be equal to plus 128, because we are subtracting the bias minus 127 to get 1. So, this is E minus bias right for 127 is the bias. So, E is plus 128. So, let us have the binary representation of 128 and the binary representation of that would essentially be 1 and seven 0s

Now, let us come to the mantissa. So, the mantissa part is 0.5. So, 0.5, if you want to represent the way that we would represent, so it is essentially is 0.5 is basically equal to 1 times 2 to the power minus 1; the mantissa contains 23-bits. So, is a 23-bit mantissa. The first bit corresponds to 2 to the power minus 1, the last bit corresponds to 2 to the power minus 23, and the ith bit corresponds 2 to the power minus i. So, the mantissa can be thought as a summation from i is equal to 1 to 23 the coefficient x i multiplied by 2 to the power minus i. Since, this is 0.5 which is 2 to the power minus 1 we will have one over here which is the MSB position and the rest all will be 0s. So, this is how we would represent a number of the type minus 3 in the IEEE 754 format.

And this is actually very easy the first we represent the sign bit then we figure out the value of the E field by adding the bias to the exponent which is 128. And then we figure out the mantissa. So, the mantissa mind you is strictly between strictly less than 1 and it is greater than equal to 0; and it is essentially a summation from you know 2 to the power minus 1 to 2 to the power minus 23 each term multiplied by a coefficient in this case we just need to stop at 0.5. So, the MSB needs to be 1, because it is 2 to the power minus 1 and rest all the terms need to be equal to 0. So, this is the representation of our floating-point number in binary.

Given this, let us take a look at some of the clear aspects of floating-point math. So, the smallest normal floating-point number that we can have, the smallest normal positive floating-point number that we can have is let us work it out. So, that basically since it is positive the sign bit is 0 and the smallest value of E that we can have is actually 1. So, basically this is minus 126, this is the exponent and the smallest mantissa that we can have in a positive setting is all 0s. So, assume that f is 1 such number, which is a smallest normal floating-point number. So, we have a floating-point f is 3 to the power minus 126. We take another number g which is f divided by 2. So, this number is 2 to the power

minus 127 which is g, and g can clearly not be represented in our system of normal numbers because we do not have a representation for it.

Now, let us consider the next statement if g is equal to 0. So, now, the question is that what is the value of g, if g is equal to 0, let us print error and should this code print error and do you think this is the right behavior, well intuitively no right. So, let me maybe you know write a big no over here, intuitively no, because f is a positive number g is the same positive number divided by 2, it is not equal to 0. So, there is as such no reason of concluding the g is equal to 0 and printing error, but we also do not have a representation for g. So, we somehow need to solve this situation.

(Refer Slide Time: 28:12)

Denormal Numbers - II

$$A = (-1)^S * P * 2^{-126}$$

$$(P = 0 + M, 0 \le M < 1)$$

- Significand is of the form : 0.xxxx
- E = 0, X = -126 (why not -127?)
- Smallest +ve normal number: 2^{-126}
- Largest denormal number:
  - $0.11...11 * 2^{-126} = (1 - 2^{-23}) * 2^{-126}$
  - $= 2^{-126} - 2^{-149}$

So, what we can do is that we can define a set of denormal numbers, where the E field in the representation is 0, and the m field is not equal to 0. So, the normal form of a denormal numbers here we change the significand; instead of assuming that the significand is of the form 1 plus M, we assume it is 0 plus M, 0 becomes the default and mantissa remains the same between 0 and 1. And the exponent we assume is 2 to the power minus 126. So, in this case, E is equal to 0, and we assume that X is equal to minus 126, mind you it is not minus 127, this is a common mistake that students typically make, it is not minus 127, it is minus 126. So, and a common question that instructors typically ask is why minus 126.

So, let us try to understand what is happening. So, let us consider the number line and let us assume that these are all the floating-point numbers that we can represent. So, the smallest normal floating-point number is 2 to the power minus 126. So, basically we want to define a very small region or numbers after this such that. So, mind you the diagram is not drawn to scale. So, this part is normal right. So, we want to define a very small regional numbers around here called denormal such that you know we our programs make sense and this particular program does not print error. To actually ensure that this is the case we define a normal form of this type, but the significand is assumed to be 0.

So, in this case, let us find out what is the value of the largest possible mantissa. The value of the largest possible mantissa is pretty much equal to the mantissa or the significance. So, they are actually the same is equal to 2 to the power i where i is pretty much or I would say minus i for i is going from 1 to 23, which is equal to 2 to the power minus 1 plus 2 to the power minus 2 all the way till 2 to the power minus 23. So, this is a simple geometric series summation. So, when we look at you know any kind of geometric series summation, so we can expand the geometric series and we can do some maths. So, I will write down the result directly and the result is 1 minus 2 to the power minus 23.

So, this is an important result and this will come many times in the book and in our discussion. So, users might want to memorize this, readers might want to memorize this, but the important point over here if I want to find the largest denormal number, this is essentially equal to 1 minus 2 to the power minus 23 multiplied with minus 126 which is 2 to the power minus 126 minus 2 to the power minus 149. So, this is the largest denormal numbers. So, pretty much if we take the number line right and if this point is 0, so the largest denormal number is at this point and the smallest normal number is at this point.

So, as we see the difference between them is really small 2 to the power minus 149. So, some difference needs to be there, because it is after all the discreet number system, it is not a continuous number system, but the important point to appreciate is that this should not have been minus 127, it should be minus 126. Because that is only when we get this property over here that we have a very, very small distance between the largest denormal number in the smallest positive normal number.

So, what is the smallest, what is the range of the denormal numbers the range of the denormal numbers, the smallest positive denormal numbers would pretty much have the mantissa the last 23rd bit would be equal to 1. So, it will be minus 2 to the power minus 23 multiplied with 2 to the power minus 126, which is 2 to the power minus 149. So, just to summarize, what is happened is that we have the set of all the normal numbers, we have just created a little bit more room of denormal numbers such that a lot of our maths in a programming actually makes sense. And we do not come up with very non-intuitive answers, so that is the reason denormal numbers have been defined in this particular fashion.

So, I would request the readers to take a look at the normal form for both normal in a standard or normal form for both the normal floating-point numbers as well as the denormal floating-point numbers, find out what are the differences do some of the maths that I did just now and convince themselves for the utility of denormal numbers, and how they can help avoid non-intuitive results.

(Refer Slide Time: 34:25)

Example

Find the ranges of denormal numbers.

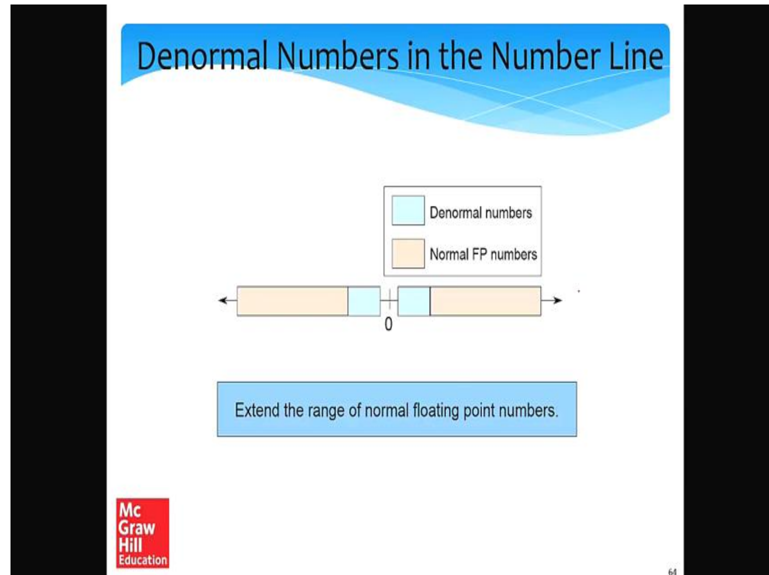
Answer

- For positive denormal numbers, the range is $[2^{-149}, 2^{-126} - 2^{-149}]$ 2^{-126}
- For negative denormal numbers, the range is $[-2^{-149}, -2^{-126} + 2^{-149}]$

So, we had pretty much the same thing in the slides but I have over written that. So, but here is the example the ranges of denormal numbers which you just found out. So, the positive denormal numbers starts from 2 to the power minus 149 to 2 to the power minus 126 minus 2 to the power minus 149, and the normal numbers then start from 2 to the

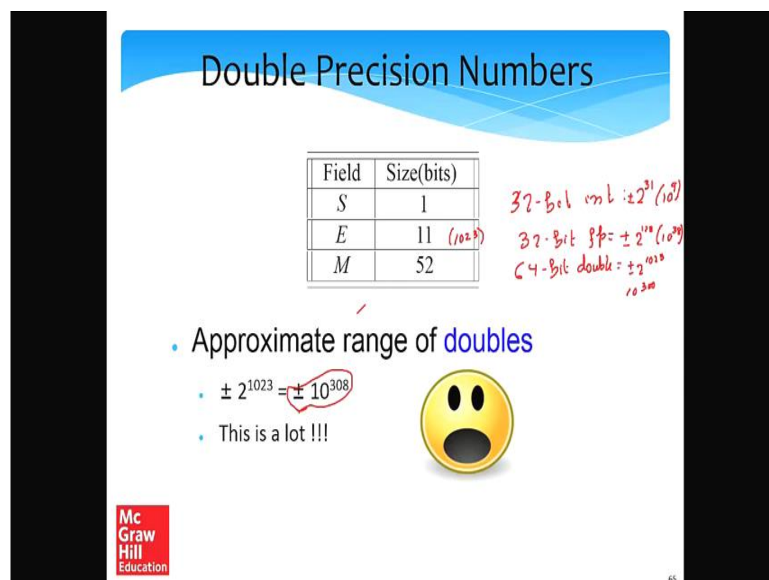
power minus 126. Similarly, for negative denormal numbers, the range is minus 2 to the power minus 149 to the same thing albeit with a sign reversals.

(Refer Slide Time: 35:03)



So, as I said all that denormal numbers do will to extend the range of floating-point numbers a little bit, and mind you such diagrams are never drawn to scale such that again our maths make sense.

(Refer Slide Time: 35:16)



So, what we saw is that the range of floating-point numbers, so let us take a look at the range of the number systems. So, for an integer with a 32-bit, so typically an integer is

32-bits right and that is what an int in C or in java would typically correspond to in a 32-bit number system it 2s complement roughly the largest number that you can represent is 2 to the power 31. So, maybe let me write it down that for a 32-bit integer, you know these are just rough figures; the maximum that we can go to is like plus minus 2 to the power 31. For a 32-bit floating-point number, the maximum that we can go to is roughly you know in the range of plus minus 2 to the power positive 127, but then of course, the mantissa can be higher. So, I can make it 128 that is maybe another largest that we can go to a very approximate figures; and 2 to the power 128 is around 10 to the power 40 typical numbers. The approximate range of doubles is much more a double precision number because this user 64-bits and does not use 32-bits.

So, in this case, we have a one bit sign bit we have 11-bits for the E field, mind you not 8-bits, but 11-bits. So, this means that. So, the bias is also different instead of a bias of 127 the bias is 1023 right; and we can cover a much larger range of numbers from minus 1023 till plus 1023. So, the range is typically plus or minus 10 to the power 308, which is a fairly large range and we would typically not need more than this for most of our calculations. So, this is a lot right and we do not typically need more than this.

So, I can add note over here that for a 64-bit double precision, what we have double in C, we are roughly at plus or minus 2 to the power 1023 and this is roughly 10 to the power 300. So, this is roughly 10 to power 300, this is roughly 10 to the power 37 or 38, and this is much, much smaller. So, 2 to the power 30 is around a billion. So, this is roughly 10 to the power is actually 3, 4 billion something like that. So, this is roughly a 10 to the power 9 kind of figure slightly more than that. So, it is several billion is limit, around 4 billion is the limit; and if I consist it is around 2 billion to be precise plus minus.

(Refer Slide Time: 38:40)

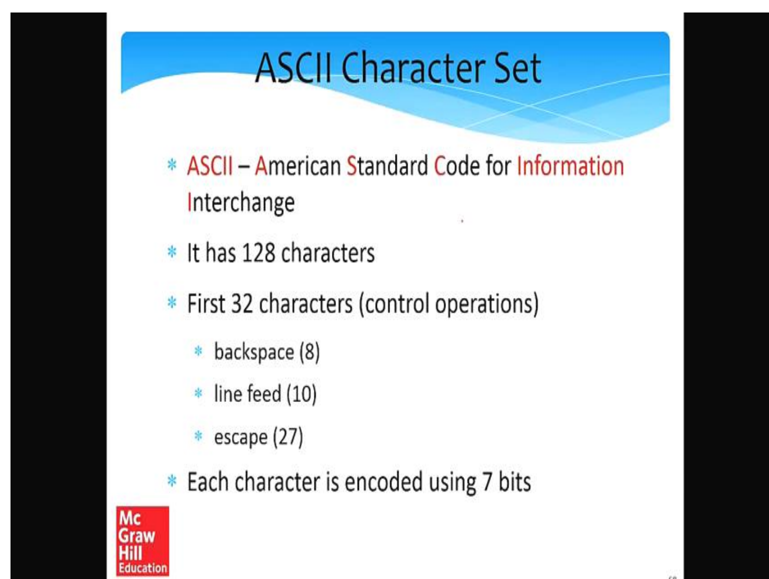
Let us now take a look at some basic floating-point math that we want to add, we have two numbers A and B, where A is 2 to the power 50, and B is 2 to the power 10. We want to add A plus B and then we want to subtract A, but A plus B we want to be done first that is the reason is there inside a bracket. So, if we add if A and B, we are essentially adding two raised to the power 50 plus 2 raised to the power 10. If we do this the result, we would look something like this that a sign bit will be 0, the exponent has to be the larger one cannot be the smaller one, the 2 to the power 50 plus the significand has to be something of the form 1 plus 2 to the power minus 40, so this is the only way that we will be able to represent such a number. So, the mantissa has to be 2 to the power minus 40, but the smallest value of the mantissa that we can possibly represent in our system is 2 to the power minus 23. So, this is the smallest value that we can represent as a result this number cannot be represented in our system.

So, this is the problems. So, what most hardware would do is that they will actually take 2 to the power 50 plus 2 to the power 10. And since the mantissa cannot be represented they will just produce 2 to the power 50, as a result; and then when we subtract 2 to the power 50 minus 2 to the power 50, C will be computed to be 0. So, this is a non-intuitive result mathematically. So, mathematically we do not expect this result, what do we expect we expect A and A to be canceled in a result to be B, which is to the power 10.

So, what a lot of smart compilers can possibly do is reorder the operations to increase precision and in this case actually break down the bracket, but this is again not what the programmer wants. So, as a result, there is a big gray area over here, but the most important thing that we need to understand is a floating-point math is approximate, it is not you know exact. The reason it is approximate is because we have a limited number of bits; and with those limited number of bits, we can own and also with a lot of constraints we can only represent a very limited set of numbers within our constraints. In this case, we cannot represent a number of the form 2 raised to the power 50 plus 2 raised to the power 10, it is simply not possible for us to represent a number of this kind. As a result here we will have a non-intuitive result are coming at the end where C, will be computed to be 0.

At least most compilers would do that a lot of compilers might want to reorder the operations or locally resolve the operations, but it is very much conceivable that this program on a lot of programming languages and hardware would actually produce 2 raise the power 10 sorry would actually produce 0. It would not produce to raise the power 10, which is a non-intuitive result. So, programmers need to keep these things in mind while writing programs with floating-point numbers and always keep in mind that is an inexact approximate representation. Now, let us take a look at the fifth part, fifth and last part of this chapter, which is representing strings. What is a string? It is a piece of text.

(Refer Slide Time: 42:45)



In any piece of text any the same way that these slides are there where I have written ASCII is American Standard Code this is a piece of text. How should we represent it. So, the most common way of representing pieces of text was with the ASCII format, and ASCII is American Standard Code for Information Interchange. It has 128 characters. The first 32 characters are actually non-printing characters therefore, control operations like. Character number 8 is for backspace to actually delete characters; character 10 is line feed which used to tell printers to jump to the next line; 27 is the escape character it corresponds to the escape key on our keyboard. And then the remaining letters small letters capital letters special characters like exclamation mark enact and numbers of course. So, since there are 128 characters, each character encoded using 7-bits.

(Refer Slide Time: 43:54)

ASCII Character Set

Character	Code	Character	Code	Character	Code
a	97	A	65	0	48
b	98	B	66	1	49
c	99	C	67	2	50
d	100	D	68	3	51
e	101	E	69	4	52
f	102	F	70	5	53
g	103	G	71	6	54
h	104	H	72	7	55
i	105	I	73	8	56
j	106	J	74	9	57
k	107	K	75	!	33
l	108	L	76	#	35
m	109	M	77	$	36
n	110	N	78	%	37
o	111	O	79	&	38
p	112	P	80	(	40
q	113	Q	81	)	41
r	114	R	82	*	42
s	115	S	83	+	43
t	116	T	84	,	44
u	117	U	85	.	46
v	118	V	86	;	59
w	119	W	87	=	61
x	120	X	88	?	63
y	121	Y	89	@	64
z	122	Z	90	^	94

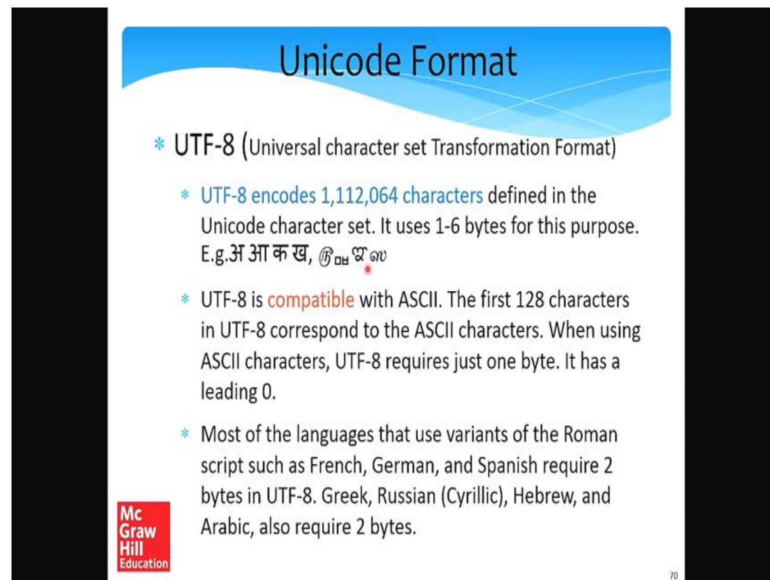
Mc Graw Hill Education

69

So, the ASCII characters set looks like this that is a here is a code of some of the common characters from small a to small z, the code goes from 92 till 122. Similarly, from capital A to capital Z, the code goes from 65 till 90. So, basically then we have a numbers from 0 to 9, where the codes are assigned from 48 to 57 and there are different kinds of code for different kinds of special characters and punctuation marks that we used like exclamation, hashed, all their brackets, comma, semicolon and so on. So, the problem with the ASCII set is first is only for English and English is a very simple language.

In English the number of characters is few, and we do not have special marks and what is there in you know other specialized characters that come up in other languages with also combinations and so on.

(Refer Slide Time: 45:08)



So, for this, we have the Unicode format. So, it is the universal character set transformation format is a UTF format. So, the UTF-8 standard enables around encodes around a million characters defined in this set and it can use 1 to 6 bytes for this purpose. So, what I have done over here is that I have written a couple of characters in different languages some of these, so this is Hindi for example, of Devnagari script. This I believe is a Tamil character; this I do not recall it might be a Korean character; and this is a I think character name from the Kannada language. So, with so many characters it is necessary to encode them.

So, for this the UTF for the Unicode format was designed which has become standard now. So, UTF-8 is compatible with ASCII in the sense that the first 128 characters in UTF-8 correspond to the ASCII characters. So, when you use ASCII characters, UTF-8 will require just one byte, and it will have a leading 0, which means that the remaining 7-bits specify ASCII characters. Most of the other languages that use variants of the roman script such as French, German, and Spanish require 2 bytes per character in UTF-8 Greek, Russian, Hebrew and Arabic also require 2 bytes.

(Refer Slide Time: 46:44)

UTF-16 and 32

- Unicode is a standard across all browsers and operating systems
- UTF-8 has been superseded by UTF-16, and UTF-32
- UTF-16 uses 2 byte or 4 byte encodings (Java and Windows)
- UTF-32 uses 4 bytes for every character (rarely used)

Mc Graw Hill Education

So, this has become a standard across all browsers and operating systems. So, nowadays it is very common for users to read articles which have been written in multiple languages maybe an ad is coming in some other language. So, all of this happens cuts a Unicode. So, UTF-8 has been superseded by UTF-16 and 32. So, UTF-16 uses 2 byte or 4 byte encodings and java and windows support UTF-16. So, as of now at least UTF-16 is a more popular character set, and UTF-32 uses 4 bytes for every characters and rarely use it is not that commonly used, but UTF-8 and UTF-16 are the encoding sub choice where basically every character is represented with a certain sequence of bits.

And in any document, you just have character by character which is essentially a sequence of bits that encode each character. So, when the document needs to be shown on the screen your word processor program extracts all the bytes out, converts them into characters for each character it draws a small image that corresponds to the way the character should look on the screen.

So, this brings us to the end of chapter two. So, what are we achieved in chapter two let us go back to the outline slide. Say in the outline slide, we have basically shown what is possible to do with a set of bits. So, we can work on them we can define an algebra, and we can have basic operations, we can represent both positive as well as negative integers using a set of bits. So, they have their constraints, but again we overcame all of those

constraints and we came up with a 2s complement notation which is the most effective notation as of now to represent negative integers.

After that, we extended the results that we had to represent floating-point numbers. So, in that case, we needed to go for a slightly different representation. The reason being that we actually needed to represent the exponent as well, and we also made a trade off for simplicity, this is one of the vital learning's that you need to have that in computer architecture sometimes you need to walk a step back and make things simple. So, in this case, instead of going for a 2s complement representation of the exponent, we actually used a bias based representation. And you also use an explicit sign bit just to make our life easy and also to leverage the pattern that you typically do not multiply the exponent you would only multiply it when a number is being raised to the power of some other number, but that is a relatively rare operation.

Most of the time, we are only adding and multiplying floating-point numbers; in that case, we do not you know for say multiply the exponent lastly we talked about strings we talked about the basic simple ASCII format for English. We also talked about the generic Unicode formats or the UTF formats two of them are very common UTF-8 and UTF-16. UTF-8 and 16 are used to represent almost all the text today in all the worlds' languages, and there are many additional characters as well and many cartoon like characters that can be used to make really expressive documents.

So, now that we have a certain hold on bits, and how they can be used. We need to design a language, a simple low level language that can communicate with the processor using these bits to accomplish fairly complex tasks which is the main goal of the next chapter, the chapter on assembly language to achieve complicated tasks using simple bits. So, see you then with chapter 3, which is going to be the next lecture.