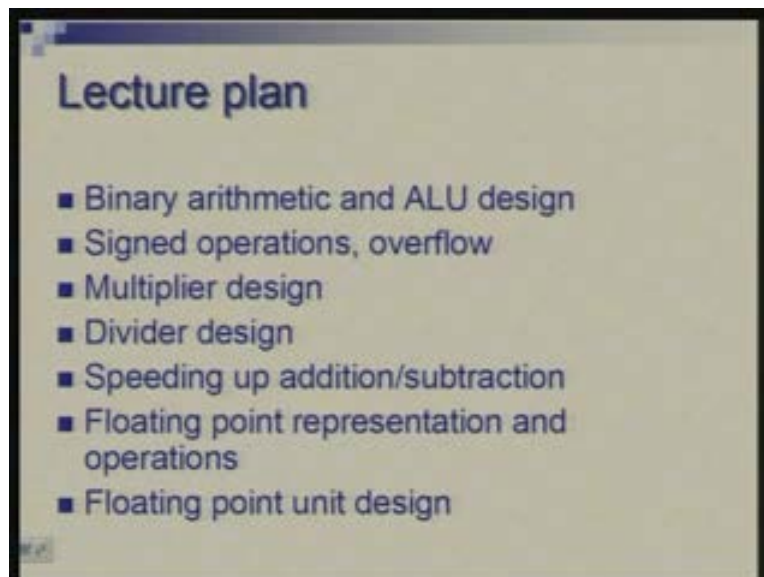


Computer Architecture
Prof. Anshul Kumar
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture - 16
Floating Point Arithmetic

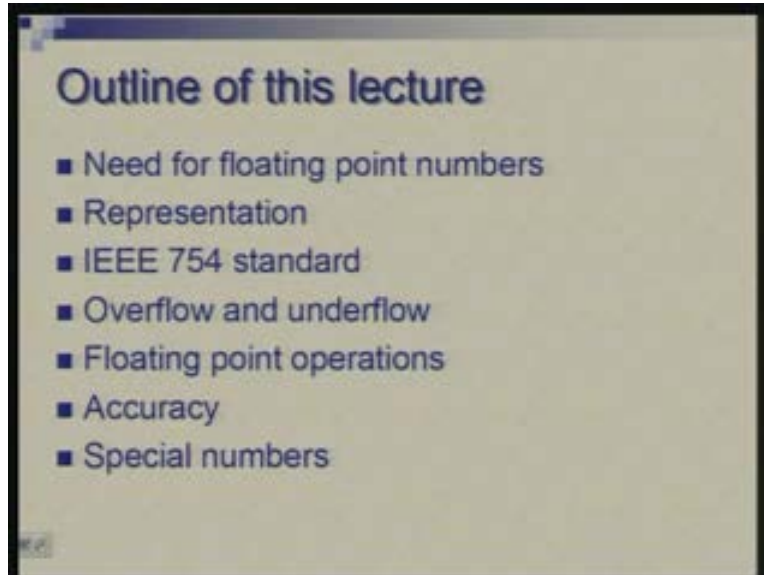
So today it is the last lecture in the series of lectures on arithmetic operation and arithmetic unit design. Today we will be talking of floating point operations. So, when numbers cannot be represented as integers you represent them as floating point numbers. And we will see how one could do the usual arithmetic and other comparison operations on floating point numbers. So we have been talking of primarily integer arithmetic; we also talked about logical operation and other operations.

(Refer Slide Time: 01:26 min)



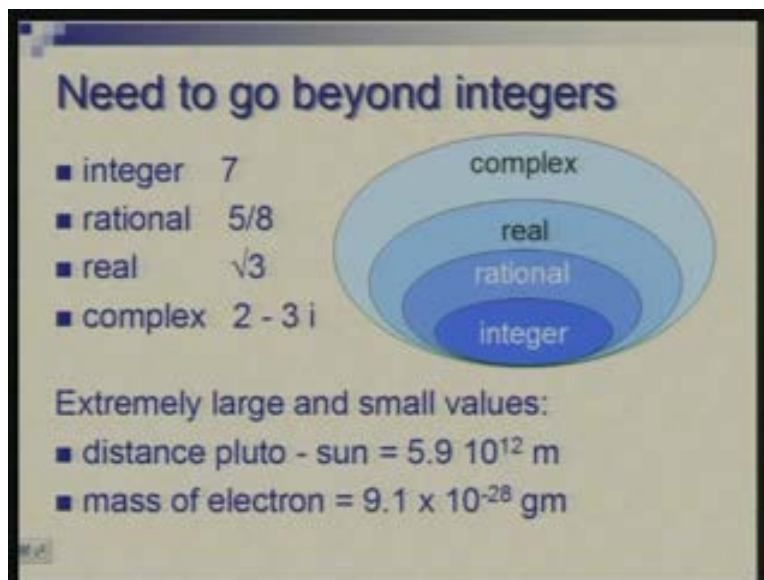
I will first of all talk about why we need to floating point numbers; what kind of representation we have and then we will work about the hardware unit for carrying out the operation. So first what is need for floating point numbers; how we represent them? I will talk about a standard representation which is IEEE 754 then define the concept of overflow and underflow in case of floating point numbers which is different from that we have in integers; how operations are carried out; we will talk over primarily add, subtract, multiply and divide and talk little about comparison as well. The issue of accuracy is very important in floating point unit; how we can get best accuracy by using reasonable cost that will be discussed. There are some special numbers which need special care we will talk of that and finally I will conclude by talking of what kind of provisions are there in processor like MIPS.

(Refer Slide Time: 02:34 min)



So first let us look at why we need to go beyond the integers.

(Refer Slide Time: 2:51)



We are familiar different data types as we encounter in programming languages and also in the world of mathematics. We talk of integers which are actually a subset of rational numbers, which is subset of real number and then subset of complex number so it is a large space of numeric world and integers are only a small subset so how do you go beyond integers that will be the topic of today.

We need to also take care of situations when the numbers are very large in terms of their values. One is that you may like to represent something which is not quite an integer. Let us say 7.4 which is neither 7 nor 4 but on the other hand, there is also a need for representing values which are extremely large or extremely small. Particularly in scientific domain we have to talk of huge distances. for example, in astronomy if you are looking at distance between let us say Pluto and Sun this is 5.9×10^{12} meters so you would not like to represent that as an integer; it is not that you are interested in values; even if your accuracy requirement is limited you are not looking at fractional values but the number is so large that integer representation is not the suitable one.

On the other extreme if you look at quantity like mass of electron which is 9.1×10^{-28} grams again it is a so small value that you will either have to scale it to that extent or think of a different representation.

So how do we represent fractions; that is one question and how do we accommodate large range that is the second part. Representing fraction means you want something which has an integer part or there is a part which is a non-integer. Also, there could be you may be looking at non-integer values as rational number.

For example, you have rational numbers they can be thought of integer pairs so 5 or 8 can be represented by two integers. Or if you have a number like let us say minus 247.09 you can represent this as a string. Minus is one character, 2 is one character, 4 is another, 7 another and so on. So each of these boxes which you see is a separate character so as a string; you can represent a number which has something to the left of the decimal and something to the right of the decimal and an appropriate sign.

(Refer Slide Time: 5:53)

Representing fractions

- Integer pairs (for rational numbers)
5 8 = 5/8
- Strings with explicit decimal point
- 2 4 7 . 0 9
- Implicit point at a fixed position
0 1 0 0 1 1 0 1 0 1 1 0 0 0 1 0 1 1
implicit point
- Floating point
fraction x base power

Or in binary form you can represent a number with a fixed position of the binary point. For example, look at this number string of 1s and 0s and you might say that here is my binary point which is equivalent to.... which is the counterpart of decimal point. So you may not, unlike this representation where you are using a character to represent that position this position may be implicit so you would say that out of so many bits the point actually lies after the fifth bit from the right then you need to only worry about the string of so many bits and the position of the binary point is fixed so this could be considered as fixed point notation (Refer Slide Time: 6:45). But this representation although is okay it does not take care of large range.

So, to get a large range both in terms of extremely large value and extremely small value you need what is called floating point where you have a fraction part which could be represented in any of the manners described above and there is a base to which you can specify you can raise to a specified power. So it is this ability that you have a base and some power associated with it that you can work with large ranges. You can have a positive power to represent extremely high values and negative power to represent extremely low values. The precision or the accuracy of what you are representing would actually come from this fraction part (Refer Slide Time: 7:42); how many significant digits you have would come from this whereas this power will take care of the range.

Now we must remember that in all such representation where you are using a finite number of bits or digits you are representing basically values which are rational. To represent irrational quantities you need infinite length of storage. So, as far as irrational numbers are concerned we would typically represent only a reasonable approximation of those numbers and all representation which are infinite number bits or bytes would be actually representing rational number in some sense.

(Refer Slide Time: 08:38 min)

Numbers with binary point

$$101.11 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}$$
$$= 4 + 1 + 0.5 + 0.25 = 5.75_{10}$$
$$0.6 = 0.1001100110011001 \dots$$
$$\begin{aligned} .6 \times 2 &= 1 + .2 \\ .2 \times 2 &= 0 + .4 \\ .4 \times 2 &= 0 + .8 \\ .8 \times 2 &= 1 + .6 \end{aligned}$$

So now let us go deeper and understand the meaning of binary numbers where there is a point somewhere. So, for example, take a simple number 101.11 so there is a this is the integer part and there is a fractional part. What is the value of such a number? We understand very well the part two the left of the point 1 into 2 raised to the power 2; 0 into 2 raised to the power 1 and 1 into 2 raised to the power 0 corresponding to this 1 0 and 1. The bits on the right of the point have weightage which is negative powers of 2. So this 1 here (Refer Slide Time: 9:24) has a weightage of 2 raised to the power minus 1 and the next one has 2 raised to the power minus 2.

So, if you add now all these bits with appropriate weightage which be which is a power of 2 positive or negative you get this 4 that this is the 4 0 that gives you 1 that gives you 0.5 that gives you 0.25 and all put together we get 5.75 in base 10 so this shows how you can actually look at binary fraction like this and get its equivalent decimal value.

On the other hand, suppose you have a decimal fraction see 0.6 how do you get its binary representation?

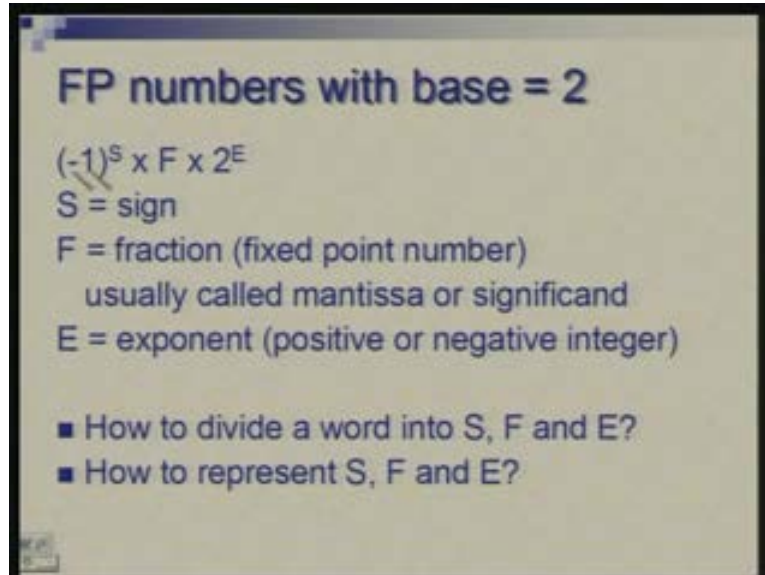
Here is the representation but let us go through the steps which will give you this. The process is that you repeatedly multiply this fraction and keep on collecting keep on multiplying by 2 and keep on collecting the integer parts. You multiply 0.6 by 2 you get 1.2 so 1 is the integer part and 0.2 is the remaining fractional part. then 0.2 is again multiplied by 2 you get 0.4 so look ahead is at 0.4 so 0 is the integer part 0.4 is the fractional part; repeat the process you get 0.8, 1.6 and now 0.6 since you got 0.6 it will again follow same steps.

So, in terms of it is binary representation you will get 0.1001 and this will repeat; just to show the repetition I have shown different colors. So now this means that you repeat this still infinity. But if you are talking of finite number of bits again this cannot be accurately represented because you will have to truncate it somewhere. Therefore, we cannot represent 0.6 exactly in binary by this method.

Of course one of things which I mentioned in the previous slide is that, rational number you will represent as a pair of integers that could be done. You could represent this as 6 and 10 that would be an exact representation but that is not very convenient to work with. What is convenient to work with is something like this (Refer Slide Time: 12:02) but there is some loss of accuracy which we must keep in mind.

Now let us introduce the exponent part. You have some base and you raise it to some power and also let us bring in the sign. So typically, a floating point representation would have a form like this so minus 1 raised to the power s, s takes care of the sign s is either 0 or 1 multiplied by F, F is the fractional part so fraction is in some fixed point sum that means you have a string of bits and there is some position where you assume binary point to be there. The base here is 2; we are talking a binary now so 2 raised to the power E, E is the exponent part, this F is the fractional part which is usually called mantissa or significand and E is the exponent part; this itself can be positive or negative we have seen in need for both why we need positive exponent; why we need negative exponent.

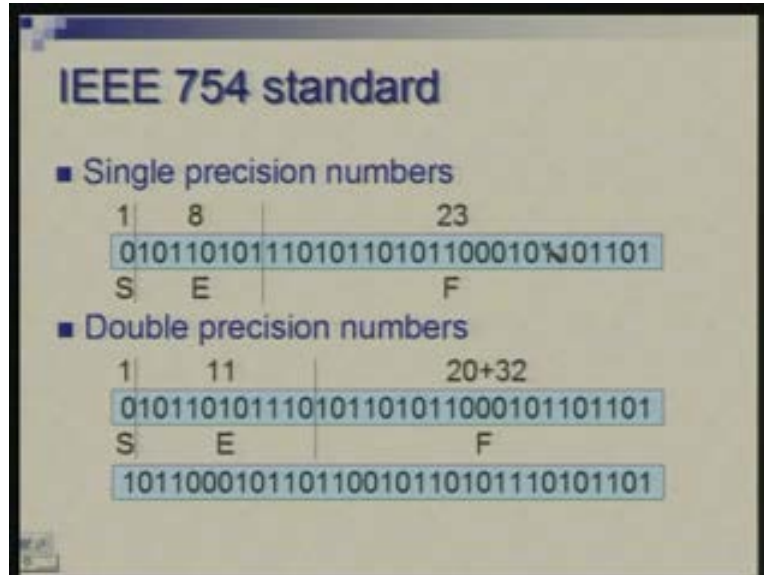
(Refer Slide Time: 13:08)



So now with this concept the question is that a given word of memory or a given register how do you divide this into three parts. That means how you allocate bits to S, F and E, S of course is straightforward it requires only one bit but how many bits for S, how many bits for E so there could be lots of options; lots of options are there and also there could be lots of options as how you represent F where do you assume the point to be and how do you represent E what about the sign of E so there are all these issues and there are multiple answers.

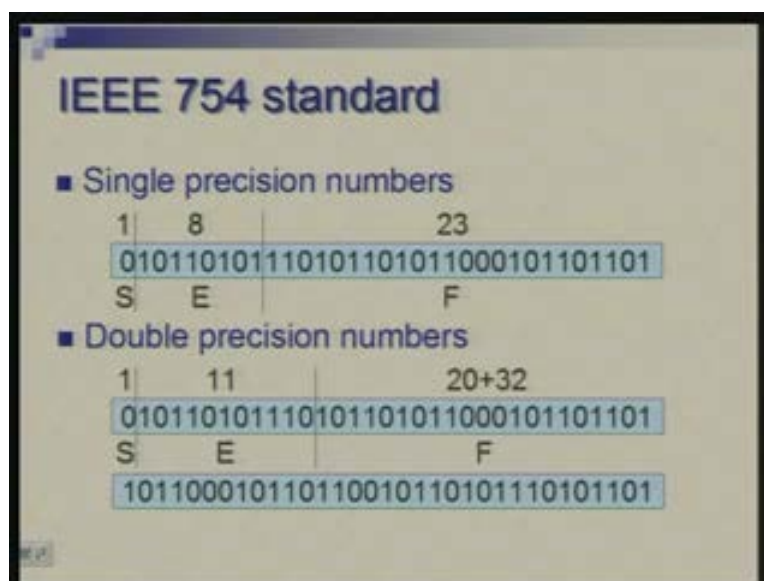
So what has happened is that over a period of time a standard has emerged which is largely acceptable and today most of the processors actually use that standard. This is IEEE 754 standard. IEEE stands for Institution of Electrical and Electronics Engineers. This is an organization, which, apart from publishing the scientific literature, they also define various standards. So this standard defines two floating representation which is called single precision and double precision. Double precision naturally has larger number of significant digits. So the division of the word which is 32-bit is as follows. The left most bit is the sign bit, next 8 bits are kept for exponent and the remaining 23 bits are for F or the mantissa.

(Refer Slide Time: 14:47)



In double precision we not only extend the precision but also increase the range of E. So there are three additional bits given here. This small difference you will see makes a big impact so you do not need to really increase too much here. so 11 bits for E and the remaining bits of this word and another word; it is a total 64-bit representation for double precision so 20 remaining bits here and 32 of the next word so together 52 bits are used for the F part or the mantissa part.

(Refer Slide Time: 14:02 min)

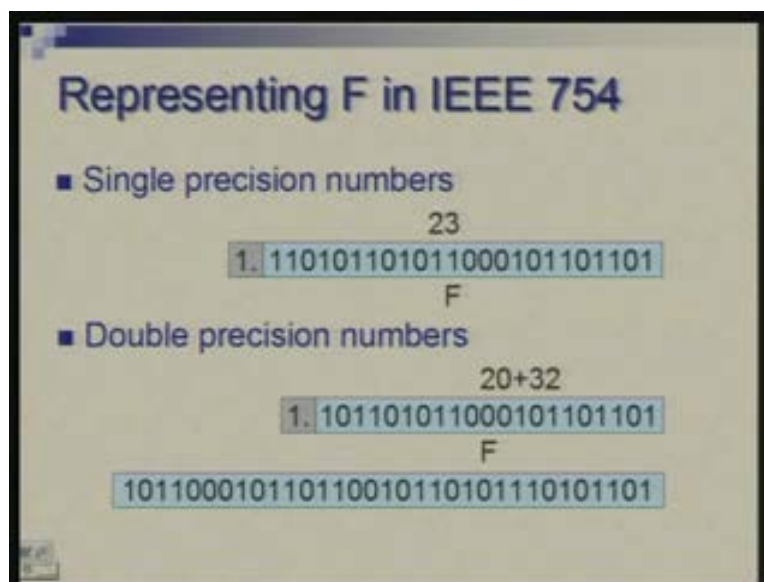


Now let us go into further details of how E and F are represented. So first let us look at F. F is the significant part or the mantissa part where in single precision case we have

actually 23 bits; we assume the point to be on the extreme left and we also assume that there is a one sitting there invisible. There is an implicit 1 here (Refer Slide Time: 16:00) which means that the numbers here we are going to talk of will be 1.something so this would decide the range as you would see that smallest number would be when it is 1. All 0s and the largest value of F would be when it is 1.followed by all 1s; we will see what it means.

Similarly, in double precision again the point is to the left extreme and there is an invisible 1 here; invisible or implicit that means you assume that 1 to be there. So effectively we are having 24 bits here; 1 bit we do not have to represent explicitly.

(Refer Slide Time: 16:28 min)



So now what are the value ranges?

Naturally with all 23-bit 0 the value of F will be 1 1.00 or effectively 1. And if you have, get in back to this if you have all the 1s here then what is the value; it is nearly 2 but not quite equal to 2; how much short of 2 it is? Corresponding to a 1 in this position. So, if you take 1.all 1s and then add a 1 in this position you will have carries going all through and you will get a 2 here (Refer Slide Time: 17:19).

We are only slightly short of 2 and to be more precise it is 2 minus 2 raised to the power minus 23 because that is the weightage of the last bit twenty third bit it is equivalent to 2 raised to the power minus 23. So you could say that F lies between 1 and 2 minus 2 raised to the power minus 23 or more approximately you could say that 1 is less than equal to F less than 2 so we are not saying how less but we know that it is nearly almost 2.

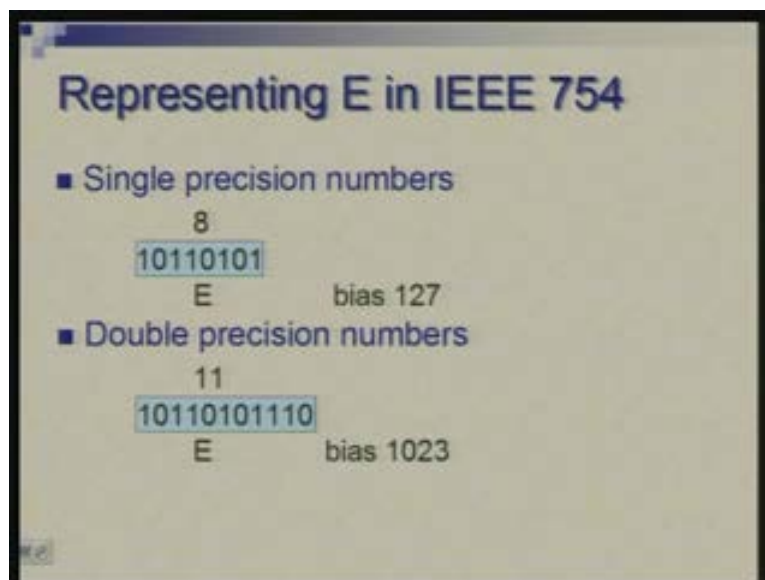
Similarly, for double precision numbers except for 23 in place of that we put 52. So now this way of representing the floating point number or the mantissa of this is called normalized representation. So, if the value actually falls beyond this; suppose you are talking of you want to say 5 into 2 raised to the power something so instead of saying 5

you divide by suitable power of 2 so that the value gets in the range of 1 and 2 so it will be actually 1.25 and you make the appropriate adjustment in the exponent.

So, suppose you wanted to say 5 into 2 raised to the power 4 equivalently you could say 1.25 into 2 raised to the power 6 so you divide here by 2 raised to the power 2 and adjust it by increasing the exponent. Similarly if F happens to be very small you multiply it by powers of 2 suitably and decrement the exponent appropriately so that the value always lies in the range 1 and 2; the value of F is always in between 1 and 2. If it is exactly 2 you can divide by 2 and make it 1 and adjust the exponent.

Let us now look at E part or the exponent part.

(Refer Slide Time: 19:18)



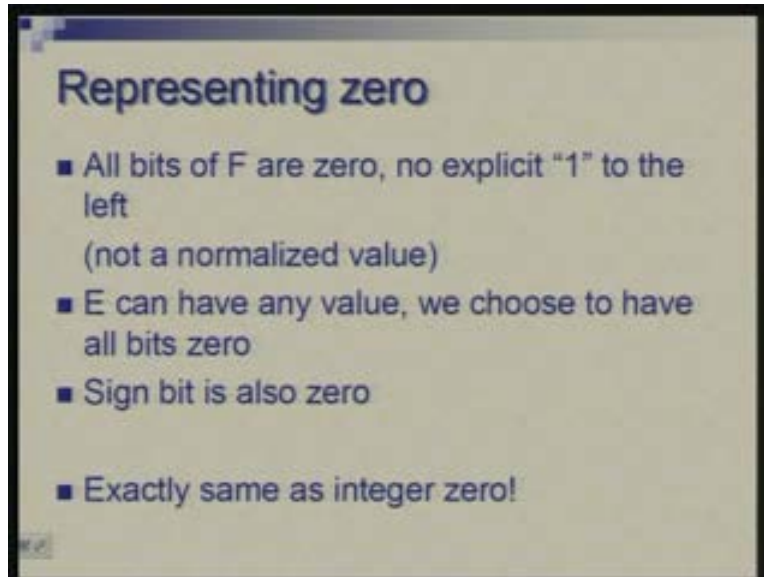
For single precision numbers we have 8 bits here. We represent exponents with a biased notation. The signed exponents are not represented as 2's complement there is a reason **I will come to I will explain little later.** So the bias used here is 127. That means if your exponent is minus 126 minus 126 plus 127 it will appear as 1. So the values these 8 bits actually can carry a value from 0 to 255 so leaving the two extremes 0 and 255 which are for special purpose **I will explain that later**; the real range is 1 to 254 and 1 corresponds to the most negative exponent, 254 corresponds to most positive exponents and with the bias of 127 1 actually corresponds to minus 156 and 254 corresponds to plus 127

Similarly, in double precision numbers the bias is 1023 and the range will go from minus 102 to plus 1023 that is shown here. The **range in** range of E in single precision is minus 126 to 127 we are excluding the cases when all E bits are 0 or all E bits are 1 that is the two extremes are used for something special and **we will discuss that later.**

Now we have talked of positive and negative numbers what about 0?

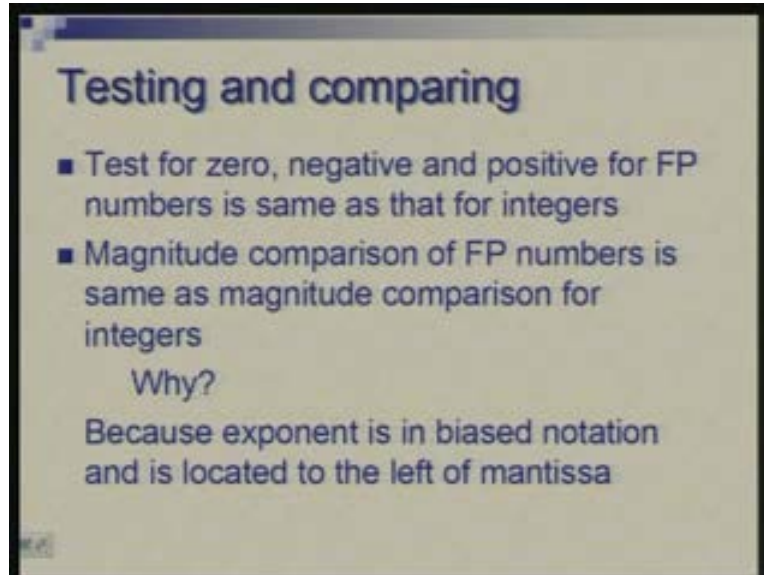
So 0 would require actually all F bits to be 0 and we cannot assume an explicit 1 to the left of the decimal to left of the point because that would mean it is 1.something; that range excludes 0 so 0 again has to be treated specially. So what we say is when F is all bits 0 and E is also all bits 0, signed bit is also 0 then it actually represents a 0 in floating point. So floating point 0 is also same as integer 0 in representation. All 32 bits are 0 it signifies floating point 0.

(Refer Slide Time: 21:45 min)



In principle any number which has f0 and any value of exponent would represent 0 but that would correspond to multiple representations. So we have a unique representation in IEEE 754 which is all bits are 0. So now with this we described how do we test the numbers for being zero, non-zero, positive, negative, how do we compare and so on. So test for 0 as you can now guess is same as what you do for integer. So, if you have a circuit which tests a number for integer 0 it will naturally test for floating point 0 also.

(Refer Slide Time: 22:39)



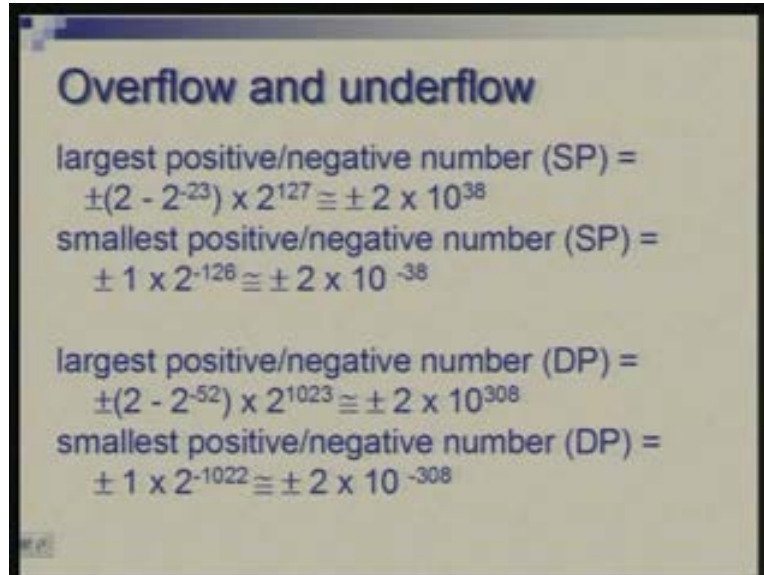
Similarly, tests for sign test for being negative or positive requires only the sign bit to be look at which is same in the two cases: integer as well as floating point.

What about a magnitude comparison?

Now, magnitude comparison also turns out to be similar to integer because of certain choices we have made. Firstly we have kept exponent part to the left of the mantissa part. So now, if two numbers have same mantissa part but one has a larger exponent obviously that will be larger because exponent has a larger weightage. Even if two numbers have different exponents **sorry** different mantissas let us say A has larger mantissa than B but if B has a larger exponent B will be larger. So it is the exponent which is more predominant and in terms of significance it naturally places towards the left. So therefore, when you compare to floating point numbers treating them as integers and if you find that exponent bits in one are larger than the exponent bits in the other then you do not need to really look at the mantissa so that is how it will happen in case of integer. Also, if some MSBs give you a decision about A is larger than B or B is larger than A then you do not need to look further. And another thing which helps in this is that the bias representation. So, irrespective of sign of the exponent this comparison will work out.

If you have, for example, 2's complement representation then the signed comparison and unsigned comparison differ. So now we can do magnitude comparison of two floating point numbers which means **we are not** we are excluding the sign bit rest we compare as if we were comparing magnitudes of integers.

(Refer Slide Time: 24:52 min)



Overflow and underflow

largest positive/negative number (SP) =
 $\pm(2 - 2^{-23}) \times 2^{127} \cong \pm 2 \times 10^{38}$

smallest positive/negative number (SP) =
 $\pm 1 \times 2^{-126} \cong \pm 2 \times 10^{-38}$

largest positive/negative number (DP) =
 $\pm(2 - 2^{-52}) \times 2^{1023} \cong \pm 2 \times 10^{308}$

smallest positive/negative number (DP) =
 $\pm 1 \times 2^{-1022} \cong \pm 2 \times 10^{-308}$

In integers we talked about overflow; we talked of overflow when you are trying to get to a very large positive integer or very large negative integer so if we exceed the positive limit or negative limit you have overflow. Here we have additional concept of underflow. So, if we look at the largest and the smallest positive or negative numbers we can represent you would notice that there is a limit of values in both directions. So you may have a number which is **which is** larger than the largest positive number, for example, so that is an overflow condition. But if you have a number which is not 0 but smaller than the smallest floating point number you can have; it is too small to be represented **in the** in the given floating point notation then it is underflow. So, number is not 0 but it cannot be.

In integer the smallest integer is 1 and you can represent that so there is no underflow but here there is a concept of underflow which you must remember.

So now let us look at the extreme values. We take the largest value of the mantissa which was 2 **into** 2 minus 2 raised to the power minus 23 or approximately 2 and largest positive power is 127. So, basically, it is 2 into 2 raised to the power 127 that is the largest number we have and 2 raised to the power 127 turns out to be approximately 10 raised to the power 38. So in terms of familiar decimal system that is the range we are getting; we are we can go up to power 38 of 10.

The smallest positive or negative integer..... so here as far as the overall sign is concerned that is the separate bit so I am not writing separate range of positive and negative; I am just prefixing a plus minus sign with this. So the smallest positive and negative value is we take the smallest F value which is 1 and the smallest E value which is minus 126 so this can also be written as 2 into 2 raised to the power minus 127 which means it is 2 raised to the power 2 into 10 raised to the power minus 38. So basically that is the range powers of 10 going from minus 38 to plus 38.

In the double precision here it is a power 2 raised to the power 1023 which turns out to be 10 raised to the power 308. So it is extremely a large number which is good enough for almost all practical purposes. Another thing you need to be concerned here is how many significant digits are there, **let me see if I have**..... another question is how many significant digits you have. See, we have significant bits as 23 here and 52 here so what does it translate to equivalent decimal digits. So to get an idea of the precision of the number you are trying to represent you can actually divide this 23 by log of 10 to the base 2 and you will get the number of digits. Similarly divide 52 by log of 10 to the base 2 and you will get the number of significant digits. So I think this will be something like 8 digits, 7 or 8 digits or so on.

Now finally let us come to how we perform operations on floating point numbers. So first let us look at addition and subtraction. Let one number be minus 1 raised to the power S1 multiplied by F1 multiplied by 2 raised to the power E1. So S1 F1 E1 are the three components of this floating point number. Similarly, there is the other one S2 F2 and E2 and we are required to either add or subtract. Before we can do this addition subtraction we must bring the exponents to the same level. Suppose E1 is greater than E2 find out which is larger let E1 be greater than E2 then we can rewrite the second number in such a manner that its exponent is made even; we do over with normalization here, change the fraction part to F2 prime where F2 prime is obtained by dividing F2 with some power of 2. So, specifically 2 raised to the power..... the difference is of the two exponents.

(Refer Slide Time: 30:16)

Floating point operations

- Add/subtract

$$[(-1)^{S1} \times F1 \times 2^{E1}] \pm [(-1)^{S2} \times F2 \times 2^{E2}]$$

suppose $E1 > E2$, then we can write it as

$$[(-1)^{S1} \times F1 \times 2^{E1}] \pm [(-1)^{S2} \times F2' \times 2^{E1}]$$

where $F2' = F2 / 2^{E1-E2}$,

The result is

$$(-1)^{S1} \times (F1 \pm F2') \times 2^{E1}$$

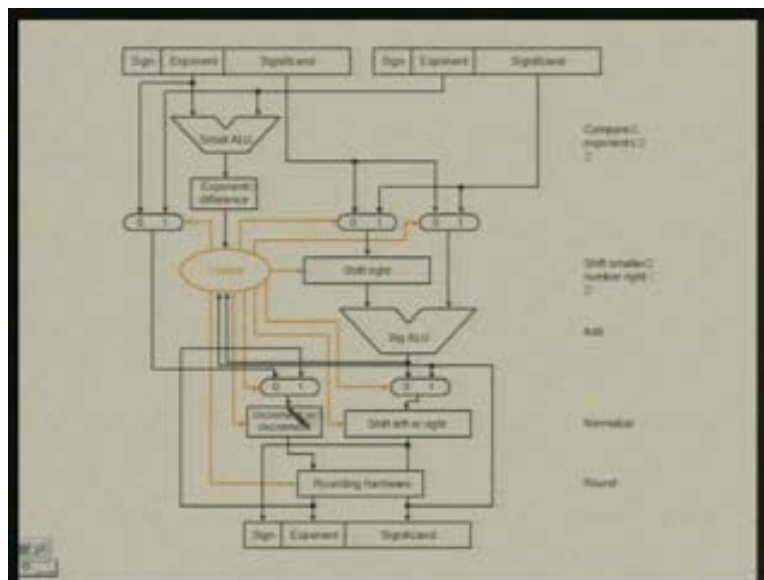
It may need to be normalized

So now dividing by power of 2 actually you can imagine that in hardware circuit it would mean that you are shifting right. So you shift F2 by so many positions E1 minus E2 and corresponding adjustment has been done in the exponent; exponent has gone from E2 to E1. So now you are in a position to add or subtract F1 and F2 prime. So, depending upon what the signs are and what the operation is you will be performing one of these operations: sum or a difference so the result is minus 1 raised to the power S1 the sum or

difference of these (Refer Slide Time: 31:02) into 2 raised to the power E1. Well, by the way this itself could be positive or negative and accordingly that might also change.

So the logic for determining sign is not difficult to work out. What I like to point out here is that when you do this you may find that this is not normalized; this sum or difference might go beyond the range of 1 to 2 which we would like to have. So, therefore, this operation has to be followed by normalization process. If the number is become too large shift it right and adjust the exponent; if it has become too small shift it left and adjust the exponent in the opposite manner.

(Refer Slide Time: 31:47)



So here this is not a detailed a circuit but a kind of block diagram indicating the flow of information as it would be in a floating point adder cum subtractor. So let us say we have these two registers holding the two numbers sign exponent and significant part. The first thing which will be done is compare the exponent; the circuit comparing these (Refer Slide Time: 32:16) is called small ALU because it is looking at only 8-bit operands. It computes the difference of the two exponents and that is the amount by which you have to shift one of the significands so that the exponent gets aligned. This is called the alignment stage.

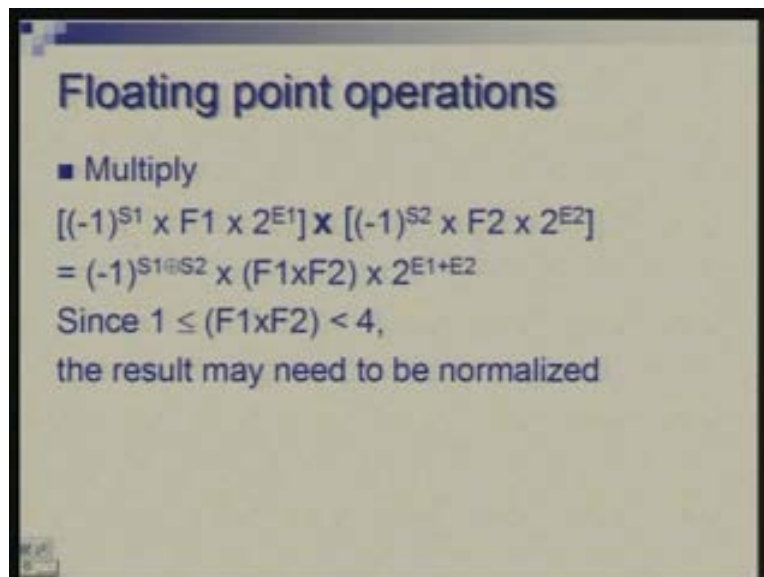
Here there is a shifter which will perform right shift on one of the significands and that is selected by this multiplexer and which one selected is determined by this control unit (Refer Slide Time: 32:54) and the control unit is guided by the difference which you have found out. So knowing whether E1 is greater than E2 or E2 is greater than E1 one of the two will be selected here, other one will be selected to go directly and this big ALU adds or subtracts the two significands or mantissas and you get the difference here. **so this is** This you could say is the first stage where the exponents are getting compared (Refer Slide Time: 33:27); this is the second stage where the number with smaller exponent is brought in par with the one with a large exponent. This is the stage where **this is the stage**

where addition or subtraction is done, this is the stage where normalization takes place. So normalization would involve output of ALU..... ignore these multiplexers for the moment I will come back to this in a moment; this output to the ALU may need to be shifted left or right depending upon how far you are from the normalized value so you need to bring it between the range of 1 and 2. And accordingly there will be an increment or decrement of the exponent which will be done which is actually coming from here; the one of the two exponents selected and passed out for increment or decrement.

Now, after normalization you need to what is called rounding. Rounding means you like to get to the LSB to the best of the accuracy. So you may be throwing something towards the right and whether you need to make an adjustment here or you can just afford to through it. I will come to this shortly.

You have some rounding logic here which may further adjust the mantissa; it may need adjustment in the exponents and after rounding off you may find that we have lost normalization once again so there may be another round of normalization so that is why these are being fed back and the multiplexer will select either the original value coming from top or after rounding and finally the result can be placed in another register. This is the hardware for add or subtract.

(Refer Slide Time: 35:32)

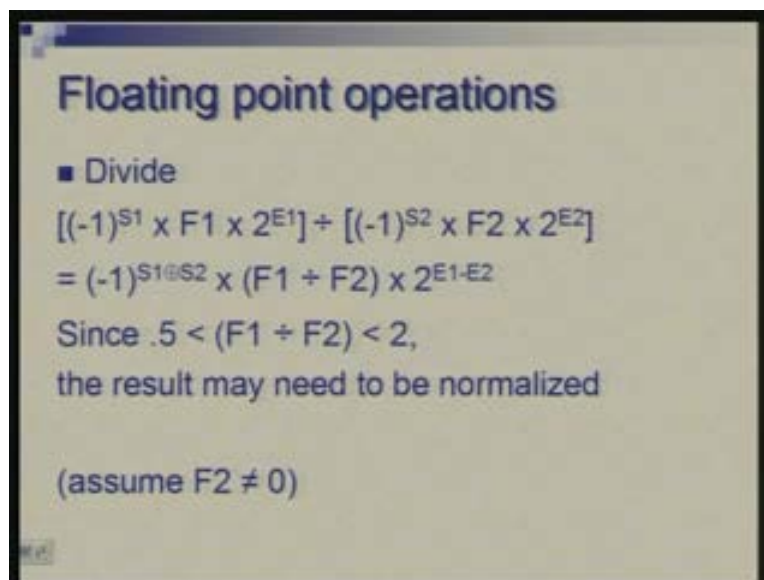


Conceptually multiply operation is somewhat simpler. If you look at two numbers: this is one, this is second, their product can be obtained straightaway, the sign of the result is Exclusive OR of the signs, if both are 1s or both are 0s the result is 0 and 1 otherwise. The fraction part is the product of the two fractions and the exponents gets summed so it rather straightforward there is no initial alignment which is required. But of course you may need to do normalization and rounding the reason for that is this product of two values which are individually in the range 1 to 2 the product could be in the range 1 to 4

so that means on the larger side it can exceed 2 and it may require one smaller adjustment you may need to bring it down. If it has exceeded 2 you divide it by 2 and increment the exponent terms.

Divide is similar; two numbers (Refer Slide Time: 36:43) same thing, the sign is determined exactly in the same way, the fractions get divided, the exponents get subtracted. Now this ratio of the fractions could lie in the range 0.5 to 2 so it will not exceed 2 because each one is in the range 1 to 2 but it can become small so **you would** you may need to normalize it here and again **there is a** there is always a need for rounding off. So of course before you do all that you need to check whether F2 is 0 or not. So you can proceed with this if F2 is not 0.

(Refer Slide Time: 37:22 min)

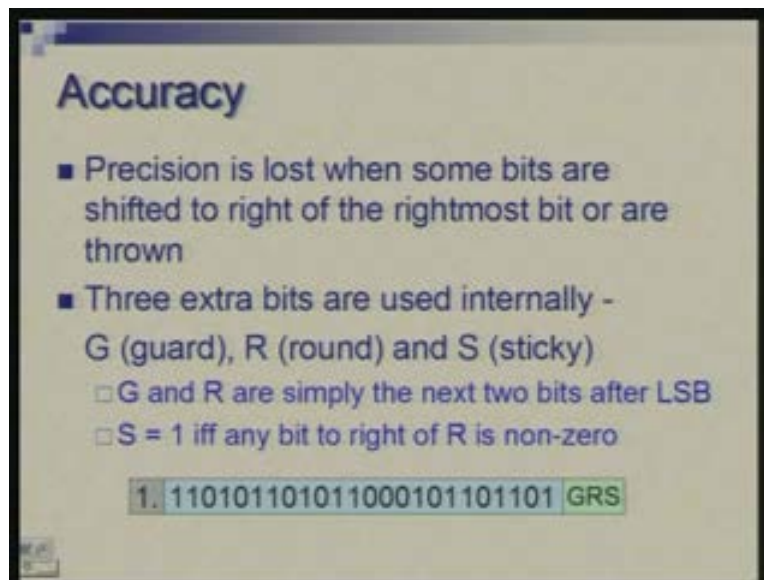


Now the question of accuracy and issue of rounding off.....so when when while doing the operations, for example, when you are aligning you are shifting the number to the right, you are throwing off some bits. When you are multiplying two fractions you have two 32-bit or let us say two 24-bits including the invisible 1 bit. You have two 24-bit exponents **sorry 22** 24-bit fractions which you are multiplying the result would be, as we have seen for integers, the result would be 48-bits in general. So you have to now throw the remaining bits and retain the 24-bits finally. So there is a loss of precision in this process because you are working with finite word length and of course you have to have finite word length so some loss would be there but you want to minimize the loss.

In the given number of bits what is the best you can do. So it turns out that if you use a few extra bits for intermediate calculation and finally do a round off based on those then you can get to the accuracy which is theoretically possible. So these bits are called G, R or S. G stands for guard bit, R for round bit and S for sticky bit. So G and R are nothing but the next two bits after those 24 bits. We normally have 24 bits as seen externally but suppose we had if we had retained two more bits these are G and R. So, in the

intermediate calculation we retain those. Apart from this we keep one more bit which is called S bit and S bit is 1 if and only if any bit towards right of R is non-zero. So **S distinguishes**; S is like a summary of the remaining bits which we are throwing off and tells it distinguishes the case which is all zeros and case where there is some at least a single 1. You can look upon the number as 1.all these followed by three more bits G, R and S.

(Refer Slide Time: 37:25 min)

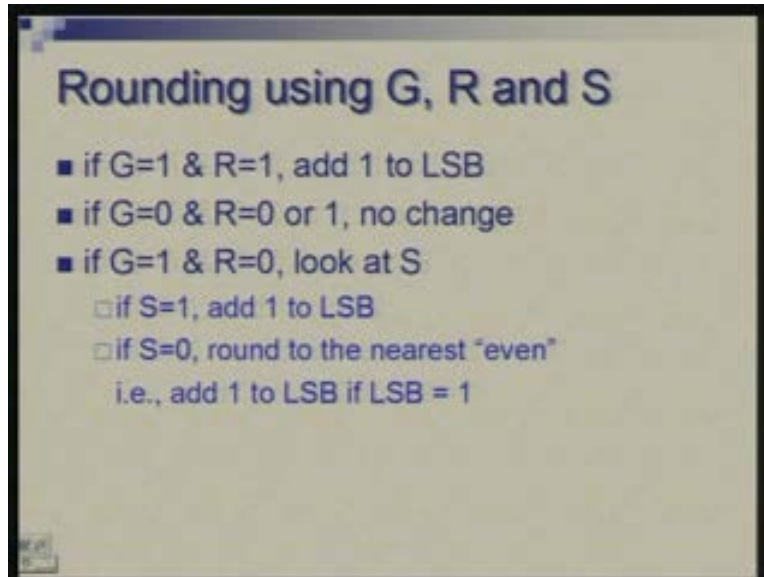


Now, given these bits; given these additional bits **which are** which are kept in the intermediate stages how do you perform rounding off. So now first let us look at G and R and just for relative sense let us imagine that the point is to the left of G. So, on a relative level let us assume that the binary point is here actually; it is just for concept otherwise actually the point is there. **So G has a weightage of** in relation to this G has a weightage of 0.5, R has a weightage of 0.25. So when G and R both are one it is like 0.75 and you will round up you will add 1 to the LSB. Now LSB means the bit to the left of G. if **both are if** G is 0 and R is 0 it is the case when both are 0, I mean it represents 0 to the right of the point or when R is 1 but G is 0 it corresponds to 0.25 so when it is 0.25 or 0.00 now you want to throw away the value towards the right of the point you want to round it down.

When G is 1 and R is 0 it corresponds to 0.5 you are exactly in the middle so whether you go up or you go down; in this case we will look at S-bit so we want to get more information to be fair. If S is 1 that means there is something which is non-zero towards the right of R so we can round up we can add 1 to LSB; if S is also 0 then we are dead at 0.5 and which way do we go? So now assuming that half the time you want to round up half the time you want to round down and all numbers are equal likely the logic which is followed in IEEE 754 is that you round to the nearest even; even in the sense of **this bit** this bit being considered as LSB so 1 here means it is odd and 0 here means it is even. So if it is 0 here **you will** you leave the number as it is, if it is 1 here it is like odd so you add

1 to make it even. So this is the meaning of round to the nearest even. So you round in a manner that the last bit twenty third bit becomes 0. So effectively what you are doing is you are trying to be fair.

(Refer Slide Time: 42:23)



Now such things are actually important, for example, when you are dealing with the financial applications **so if you if you** if there is a bias in your rounding method then over a period of time the gains may accumulate for one party and losses may accumulate for another party which may not be fair so you would like that rounding up takes place half the time rounding down takes place the other half of the time.

I mentioned that some codes for exponents are reserved for special numbers so here you will see the entire picture.

(Refer Slide Time: 43:11)

Special numbers				
Single precision		Double precision		object
exponent	mantissa	exponent	mantissa	
0	0	0	0	zero
0	nonzero	0	nonzero	denorm
1-254	any	1-2046	any	norm
255	0	2047	0	infinity
255	nonzero	2047	nonzero	NaN

The first two columns are for single precision numbers and the next two columns are for double precision number and the last one represents last one indicates what object or what kind of number we are trying to represent. So if you have everything as 0 when it is a special case it is representing a 0, when..... **let me look at this**..... when exponent is at it is peak high the highest value 255 and mantissa is 0 then it represents infinity. The sign **the sign** bit is also there which could be plus or minus and it will represent accordingly plus infinity or minus infinity so now what do programs do with infinity.

When you are writing a program you can always check this condition. For example, at some point you let us say reach a situation that the result is coming as infinity how do you represent; you do not want to stop computation there, you want to represent infinity and proceed further. So **you can write** you can express this in this way and subsequent part in the program would understand that the number is infinity so you will not try adding infinity to something which is not infinity and so on so this gives a method for doing so.

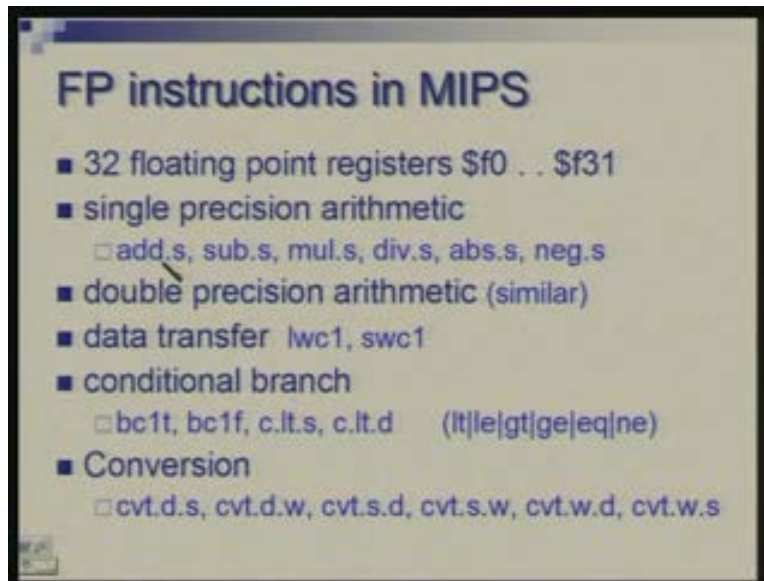
The other combination, for example, suppose exponent is 0 but mantissa is non-zero this represents a denormalized number. So when underflow occurs you can still proceed with your computations; you can represent numbers by doing away with the requirement of normalization because if you insist in normalization then there is a smallest value you cannot go beyond that so you do go beyond you force your way beyond that represent them as denormalized numbers but you must know that now **there is** that invisible 1 is not there. This is the normal situation that exponent is the range 1 to 254 and mantissa could be anything and the last one is that exponent is 255 same as infinity but you have a non-zero value. So **this is the** this is used to represent something which is not a number.

For example, 0 by 0 you cannot say it is infinity it is something which is undefined. So there could be different situations which lead to numbers which are undefined and by

choosing some code here you can have programs and specific meaning of all those situations.

Finally I will give a summary of the instructions which are there in the MIPS processor to handle floating point number. First of all there are thirty two floating point registers labeled as dollar f0 to dollar f31. These are in addition to the thirty two register we have for integer. Actually this entire hardware which does floating point activity is considered as a co-processor so it is a companion of the processor whose special task is to work with floating point numbers. So there are usual operations: add, subtract, multiply, divide, absolute value, negate which can work on single precision data or multi-precision double precision data. So you have two versions of each add dot s and add dot d stands for single precision addition or double precision addition.

(Refer Slide Time: 47:15)



So, like integer MIPS addition this requires three floating point registers to be specified. You would say add dot s and then give three registers. When you are working with double precision arithmetic you need to specify registered pairs; f0 f1 form one pair; f2 f3 form another pair and so on there are sixteen pairs; then there are instructions lwc1 and swc1 so this is load word but we are also saying c1 which stands for co-processor 1.

There can be many co-processors attached with the main processor and floating point processor is considered as co-processor 1. So these instructions would load a word from memory into the co-processor 1 which is the floating point unit that means one of these registers so you specify which registers you want to load into and you specify one integer register in the usual manner to specify the address so there is a constant part 16-bit constant part and a base register so together they specify memory address; the store instruction is similar.

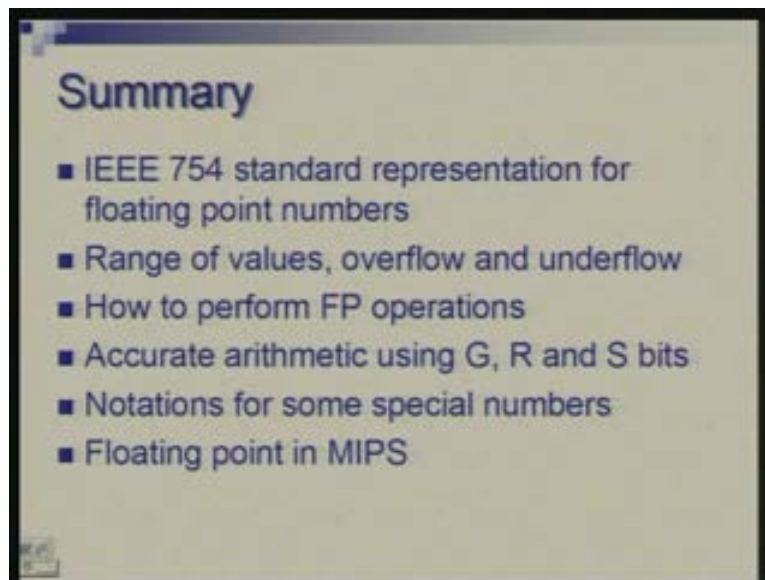
Then there are conditional branches bc1t and bc1f. b stands for branch, c1 stands for co-processor 1 and t and f stands for true and false condition.

Now how do you determine the condition?

You need to use one of these compare instructions to set the condition. Imagine that condition is a 1-bit register or a flag which is set by one of the compare instructions. So, in SLT for example, in SLT instructions the result of comparison was brought into one of those thirty two registers but here the result of comparison is brought into a separate flag which is tested by this so test for truth and test for falsehood. These two instructions do the comparison so c means compare, then lt means less than, s or d would mean single precision or double precision. So lt could be replaced by le gt ge so there are six different instructions for single precision, six for double precision. Then there are instructions for conversions. Because you have three representations: integer, single precision and double precision and you need conversions.

Suppose you have a number 3.5 it will have some representation in single precision, some representation in double precision so a 32-bit pattern or a 64-bit pattern and there may be sometimes a need for conversions or conversion to integer which means to take out the integer part of if you have integer 3 you can convert to floating 3.0 so these conversions could be somewhat lossy; particularly when you are going from higher precision to lower precision you may lose some information.

(Refer Slide Time: 50:30 min)



Finally what we have learnt today is we talked about range of values which we can handle, definition of overflow and underflow, we learnt how to perform various operations, we talked about accuracy use of G, R and S bit is for rounding off, we also looked at notations for some special numbers and finally looked at floating point operations given by different instructions in MIPS processor. Thank you.