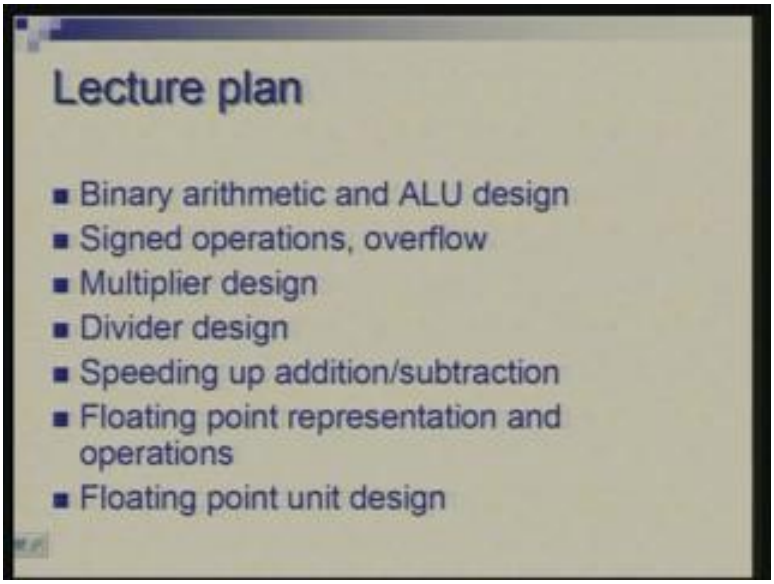
Computer Architecture
Prof.Anshul Kumar
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture - 12
ALU Design, Overflow

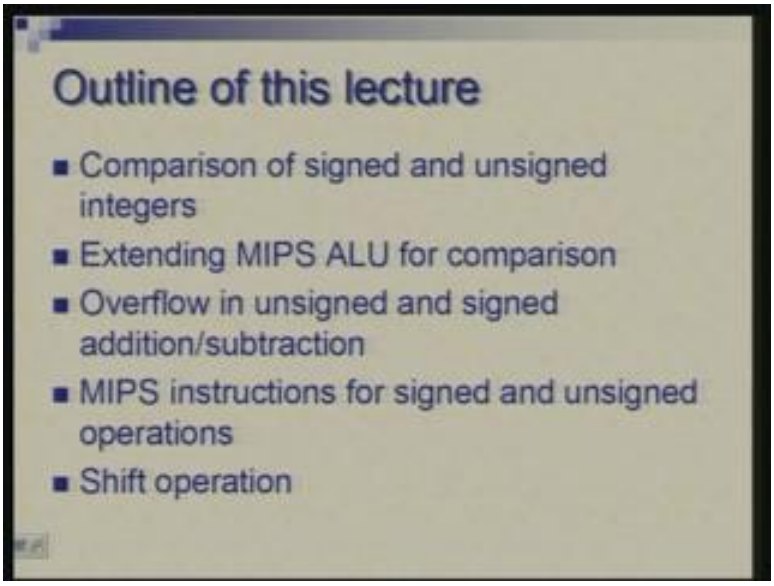
We are discussing the design of ALU which is the key component in the design of a processor. So far we have discussed an ALU which can perform addition, subtraction and logical operations like AND and OR. Today we will extend this design to include comparison equality comparison as well as less than comparison. We will also look into the issue of overflow; how do you detect that the range of the result is beyond permissible limits. So that is an overflow condition which differs from signed numbers and unsigned numbers. So this is again the same thing which we looked at last time. This is the overall plan of the lectures on design of the arithmetic and logical part of the system. So we have discussed binary arithmetic, how numbers are represented in signed and unsigned fashion and we have started with ALU design.

(Refer Slide Time: 01:42 min)



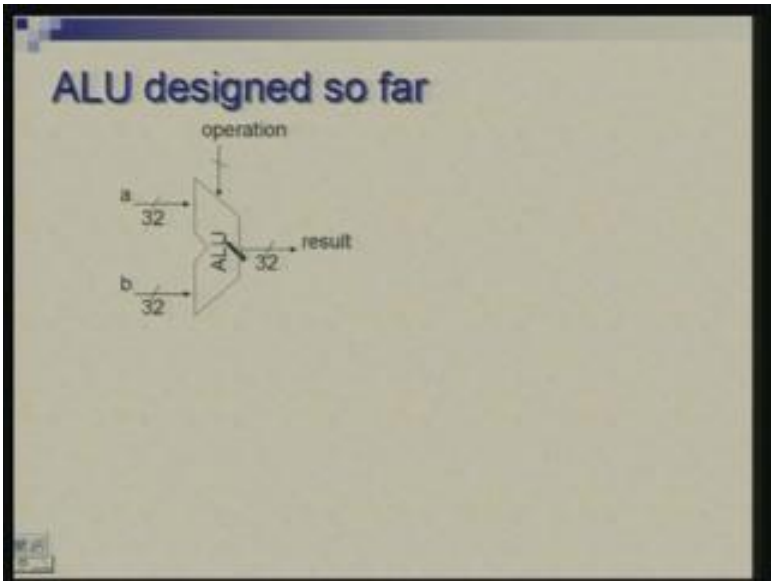
So today, in particular, we will first look at comparison operation; how do you compare signed numbers and unsigned numbers. Then we will extend the ALU design which we have discussed so far to include these operations. Then look at the issue of overflow detection for signed and unsigned numbers in addition operation and subtraction operation. I will then summarize what is the difference between instructions which have u suffix which stands for unsigned. So there are various instructions which have their unsigned counterparts so we will learn what exactly their difference is. And finally I will talk of another operation which is not an arithmetic operation; it basically manipulates the pattern of bits in a register and it is called shift operation.

(Refer Slide Time: 03:00 min)



So the ALU we have designed so far has this outline (Refer Slide Time: 3:23) that you have two inputs each 32 bits and produces a result which is 32 bits under the control of these signals which decides which operation is to be performed.

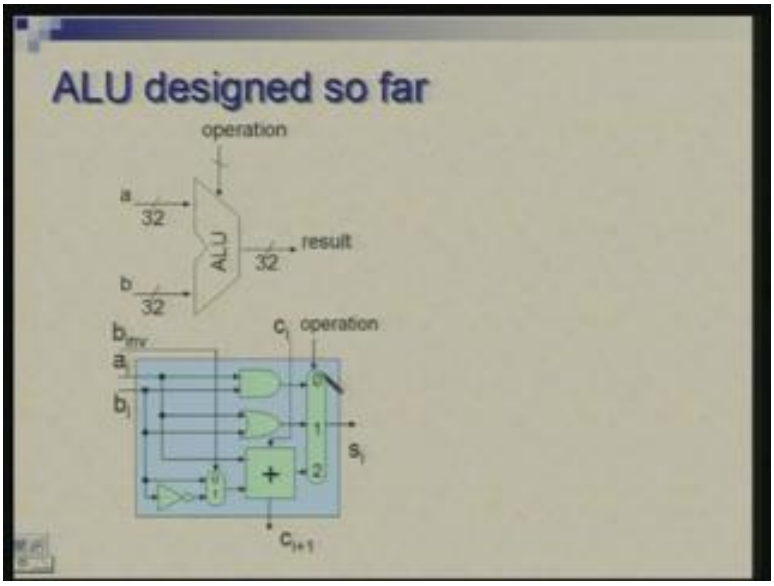
(Refer Slide Time: 3:35)



What we have discussed so far is that there are four possibilities: AND, OR, add, subtract. So which of these operations is done is decided by these controlled inputs and the number of signals here would depend upon how many different operations this is capable of performing. So the ALU design we have discussed is essentially an array or a chain of identical units each looking at 1 bit and that design is shown here. There is a 1

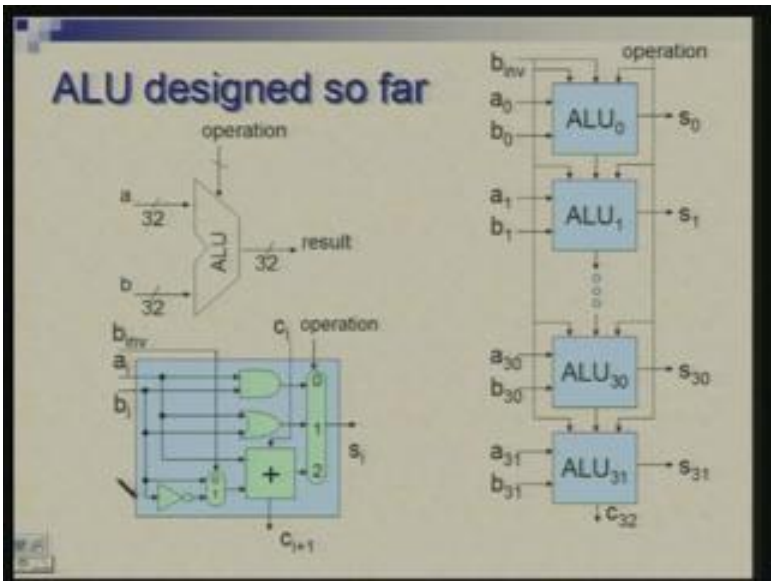
bit adder and with the help of this inverter and small multiplexer it is able to take care of addition or subtraction operation and there is an AND gate and OR gate with the corresponding logical operations and there is a multiplexer here (Refer Slide Time: 4:36) which selects output of AND gate, OR gate or the adder.

(Refer Slide Time: 4:42)



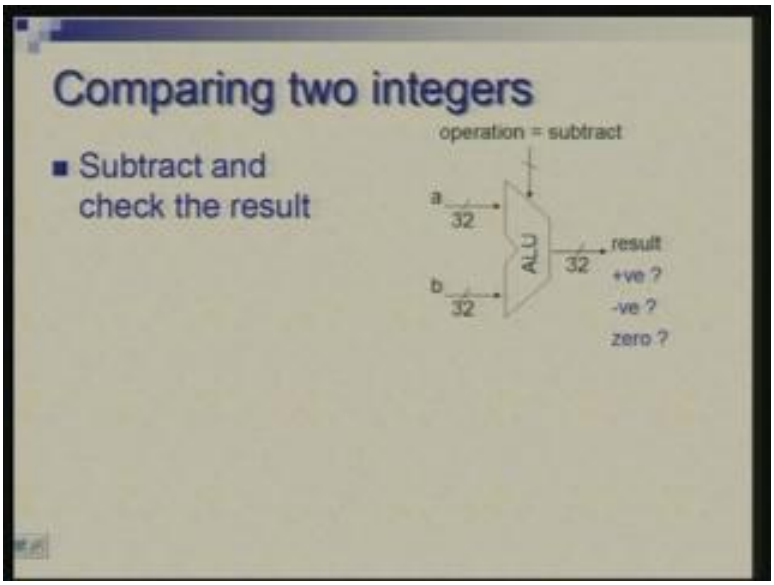
So you would require here, for example, 2 bits to decide between one of these three inputs and you require another signal 1 bit to decide whether it is addition or subtraction. So you can put these blocks thirty two of them and chain the carry through to get a 32-bit ALU.

(Refer Slide Time: 04:58 min)



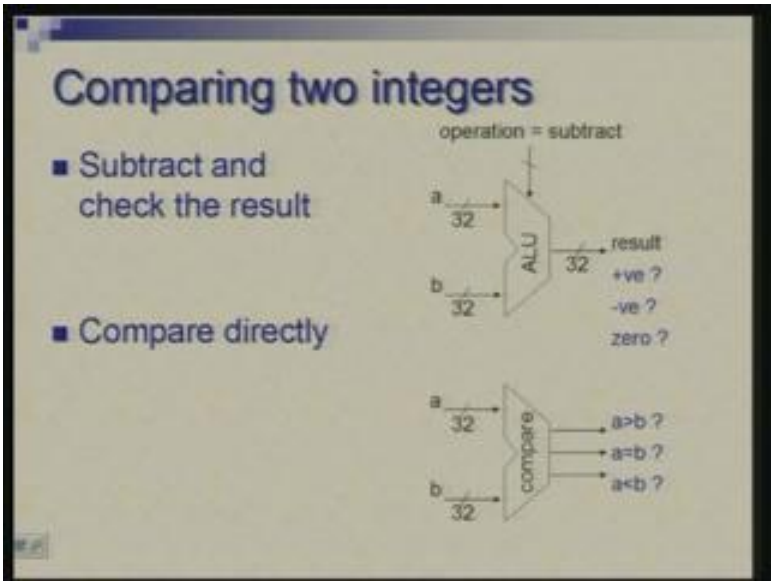
Now let us address the question of how do you compare two integers. One possibility is that you perform subtraction and then check the result. So if you have hardware built already for subtraction one could make use of that. So let us say, the same ALU which would perform….. you will control this, ask it to perform the subtraction and look at the result, check whether it is positive, negative or zero and that will decide whether a is greater than b, a is less than b or a is equal to b so that is one way.

(Refer Slide Time: 5:49)



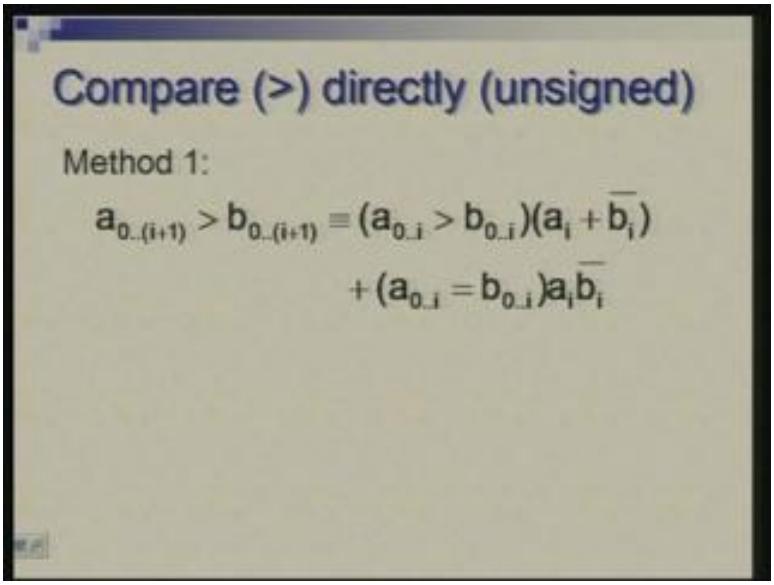
Another approach is to design a circuit which is independent. It can directly perform comparison of any of these kinds and here we can have a circuit which produces one of these three outputs directly without resorting to subtraction and then looking at the results. So we will look at both these possibilities one by one.

(Refer Slide Time: 06:12 min)



First, let us look at direct comparison because the other one is somewhat straightforward. We will look at two different ways of performing a direct comparison.

(Refer Slide Time: 6:30)



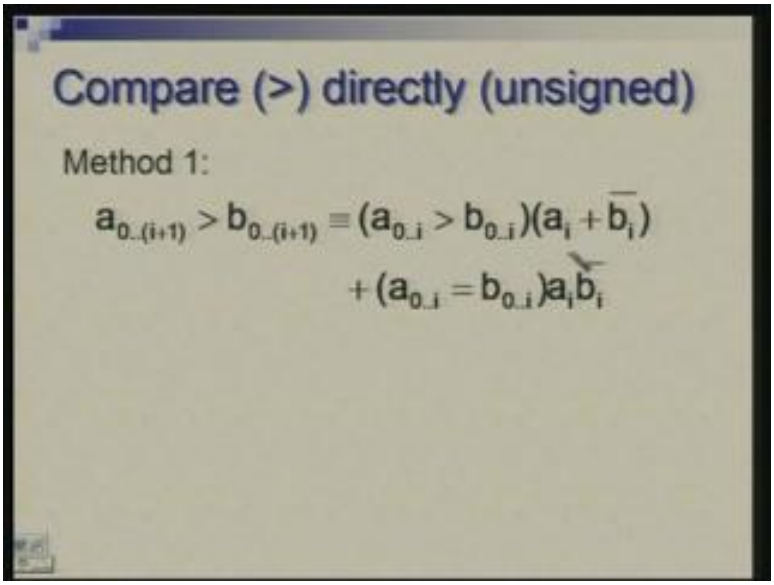
I am looking at comparison for greater than first. So you could have comparison giving a result of a equal to b, a greater than b or a less than b. So we are looking at condition when a is greater than b. here is a recursive definition here is a recursive definition of this comparison. What we are seeing here is that if you were to look at the result of bit 0 to i plus 1 of a and 0 to i plus 1 of b this part of a is greater than this part of b (Refer Slide

Time: 7:16) this condition is equivalent to comparison of bits 0 to i and some addition of….. and some additional conditions.

So, if the comparison up to bit i says that a is greater than b and this bit that is a_i and b_i says that either a is 1 of b is 0 that means as far as this bit is concerned a is either equal or greater than b and so far it is greater than b that is one possibility. The other possibility so far it is equal and now a is greater. So we are trying to define result of comparison over…… actually this (Refer Slide Time: 8:06) is i plus 2 bits interval i plus 1 bits. So in a recursive manner you can define. So that means at 1 bit position you need to look at this logic and this logic and you have condition generated for one additional bit. So here we are trying to go from LSB to MSB least significant side to most significant side.

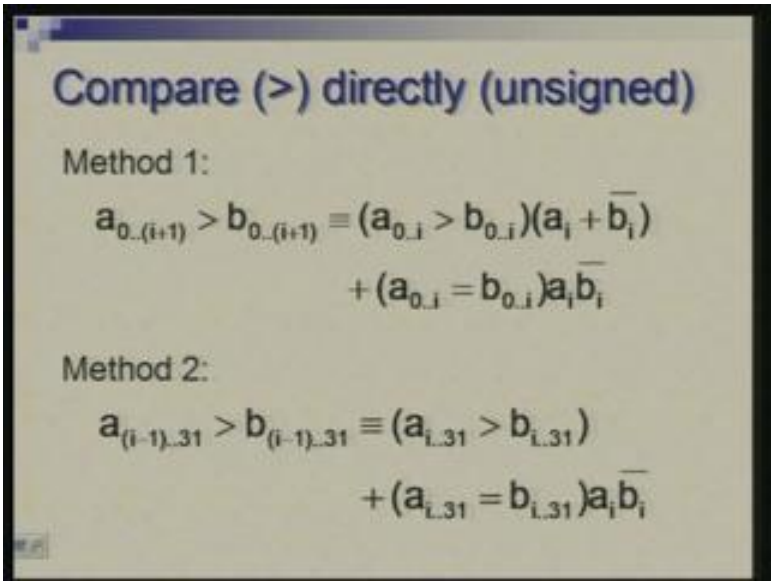
Yes, (Conversation between student and Professor…….(8:36)] a_i plus 1 b_i plus 1 is which i which i equal to the)…. Oh yeah, you are right, you are right, I think this should be okay read….. read this as i and read this as i minus 1 or alternatively you could make this i plus 1 bit and this also as i plus 1 bit. Yeah, so please note that correction.

(Refer Slide Time: 9:00 min)



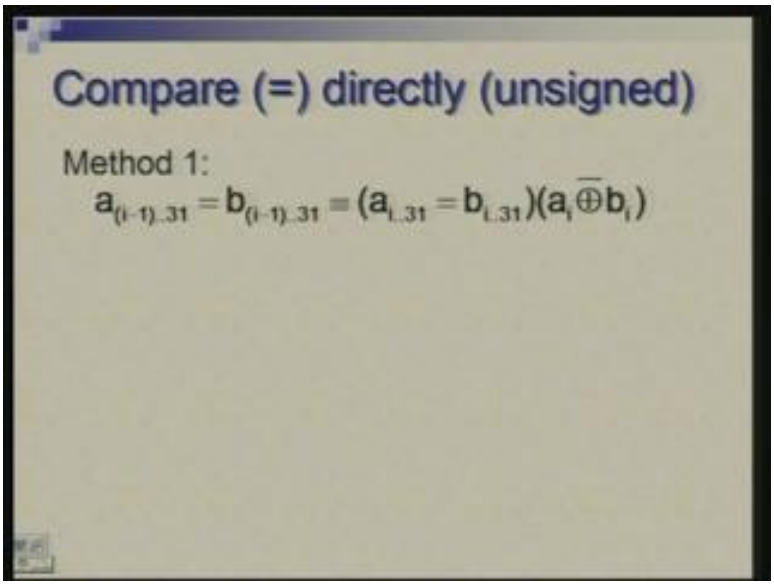
So you know the result up to i, you are comparing i plus 1 bit and updating the result for one additional bit. Yeah there is a same problem here.

(Refer Slide Time: 9:17)



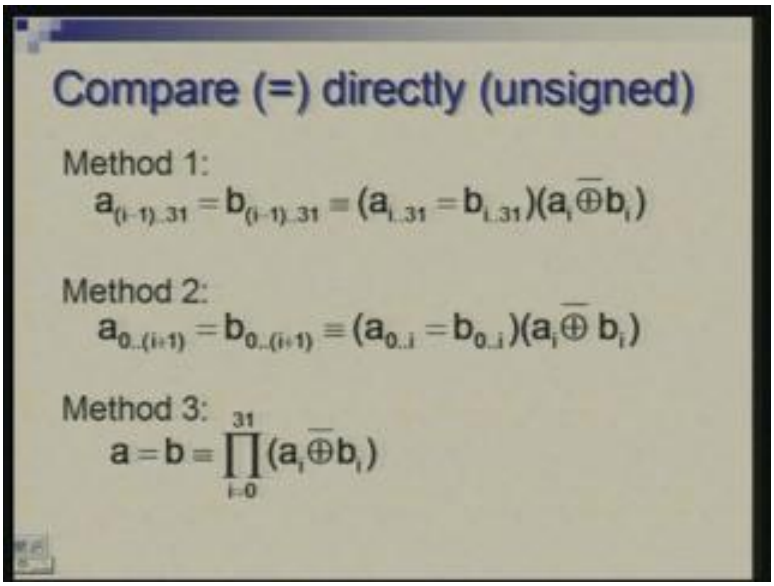
Now we are looking at results of bits i to 31 and from that we are generating the result for i minus 1 to 31. So again the comparison here is starting from one end and going to the other end. The difference between the two cases is that in the first method you are starting from LSB and propagating the result to MSB; here it is just the reverse. So clearly the second one has somewhat simpler logic and the reason is clear that it is the most significant bit which met us more so if you have the results of a few higher bits then a simple logic yields a result for one additional bit. So what we are saying here is that if you have found on the left side that a is greater than b then you do not have to do anything else or if so far you have found if a is equal to b then you should see that at this particular point a is greater than b.

(Refer Slide Time: 10:41)



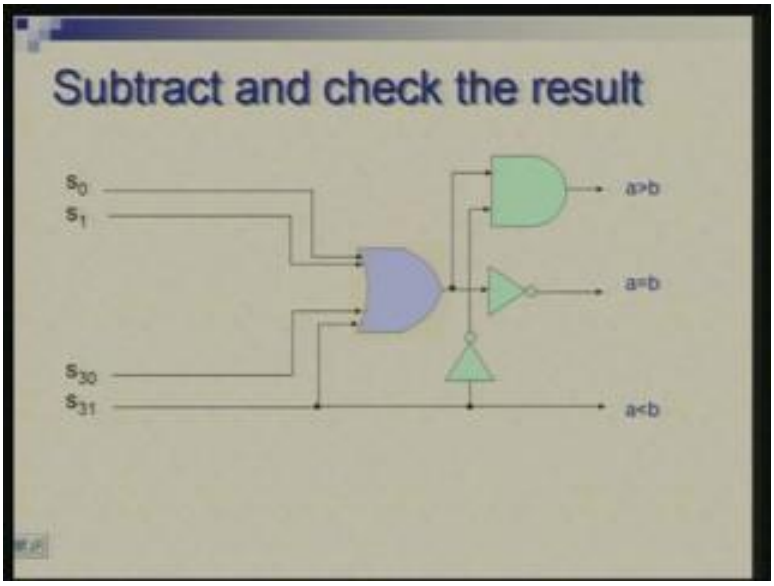
Comparison for equality can also be written in a recursive form in the same way. This is going from MSB side to LSB side. So what we have to say is that if the comparison so far is equal and the comparison now is equal then the comparison for one additional bit inclusive is equal and similarly going from the LSB side to MSB side.

(Refer Slide Time: 11:07 min)



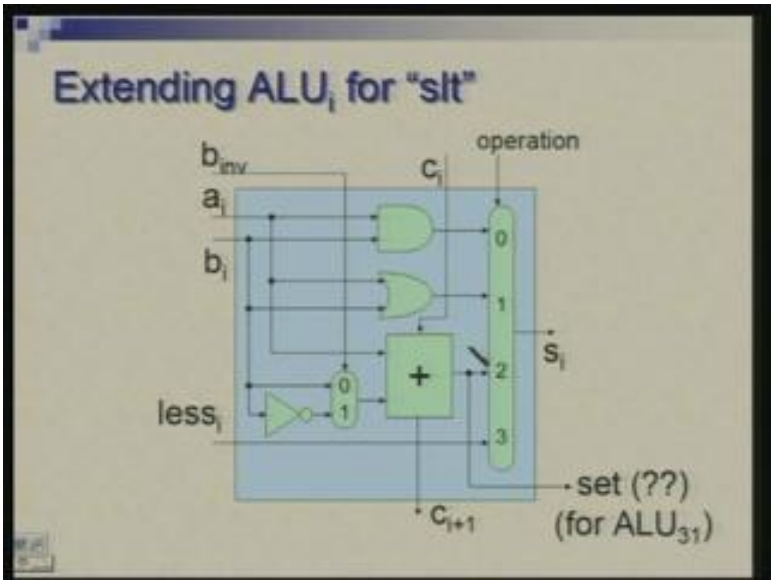
You can also write it directly, you are comparing these bits individually and you take a product or product in the sense of logical AND so this product (Refer Slide Time: 11:22) from i to 31 i equal to 0 to 31 means that you are ANDing or conjuncting all these individual bit comparisons. So this is a direct comparison, we are not subtracting and then looking at the output. On the other hand, if you want to subtract, then a simple logic can be used to check whether the result was positive, negative or zero. So first of all this sign bit, suppose S_o to S_{31} your comparator or your subtractor outputs so this is the result of subtraction (Refer Slide Time: 12:06) if you look at the sign bit if the sign bit is 1 then clearly a is less than b. To check whether the result is 0 you can put an OR gate across all these so that the output will be 0 only if all this are 0 and if you invert this we get an indication, this line will be 1 if a is equal to b, it will be 0 otherwise. So this will be 1 and only this is 0 which is the case when all these are 0.

(Refer Slide Time: 12:42 min)



Finally a greater than b if this is not zero (Refer Slide Time: 12:57) that means you have a 1 here and the sign is also 0 so this is the time to distinguish between non-negative and strictly positive. What we are saying here is that result is strictly positive excluding the 0.

(Refer Slide Time: 13:18)



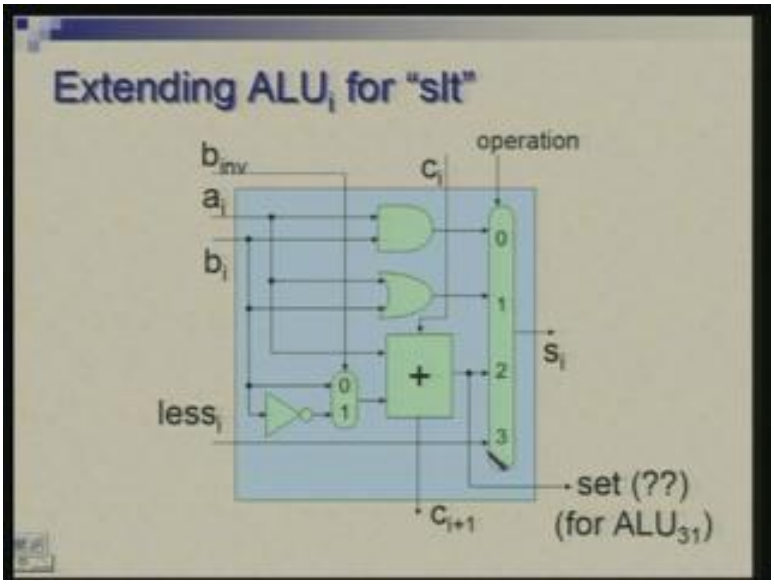
Now let us extend ALU design to include slt instructions. slt instructions says that you compare first operand with the second operand and set the result to 1 if first is less than second. So we need to have an indication of a less than b, which from the previous diagram can come out of this (Refer Slide Time: 13:49) so let us ignore these for the

moment, we are interested in a less than b. So a less than b signal can be picked from the result of subtraction of ALU_{31} .

We need to look at only one of the bits; you have subtraction output, the MSB which indicates the sign can be picked up. of course there are some question marks which I am putting here so we will come back to this and explore but according to what we have discussed if you take this signal for ALU_{31} (Refer Slide Time: 14:33) this will indicate whether we have to set the result to 0 or 1.

Now what it means is that if a is less than b then you have to set all bits to 0 and LSB to 1 and if that is not the case then you have to set everything to 0. So in any case bit 1 to bit 31 have to always be set to 0 and it is only a bit 0 of the result which needs to made a 0 or 1. We extend this multiplexor to include one more possibility which will come from an input which I am labeling as less i so in ith ALU you have an input which is labeled as less i and we will see how we make it 0 or 1 and this is tapped from the ALU_{31}. So putting these putting thirty two of such units together we get this.

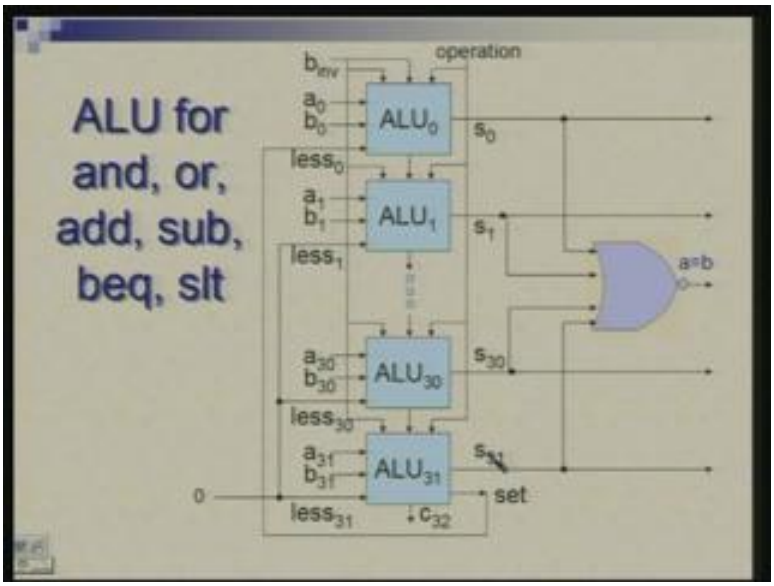
(Refer Slide Time: 13:19 min)



So the less input of all the bits except the first one that means from ALU_1 to ALU_{31} the less input is connected to 0. So, irrespective of comparison, irrespective result of comparison we have to set these 31 bits to 0 and this bit (Refer Slide Time: 16:05) will be set to 1 if you are getting a 1 here. So, if the result of comparison says that the result is negative we get a 1 here and that sets this to 1 otherwise that is set to 0 and all other bits are set as 0 in any case. So now the roll of this controlled input and this b invert these two inputs together will insure that when you want to execute slt instruction ALU should be instructed to perform subtraction and the multiplexor should be instructed to select the fourth output. So, in context of this for performing slt b_{inv} should be 1 so that this does subtraction this whole thing does subtraction and this MUX should be controlled to select

the last input so that it is not that subtraction result which is passed down, it is what your setting here on less which is passed on.

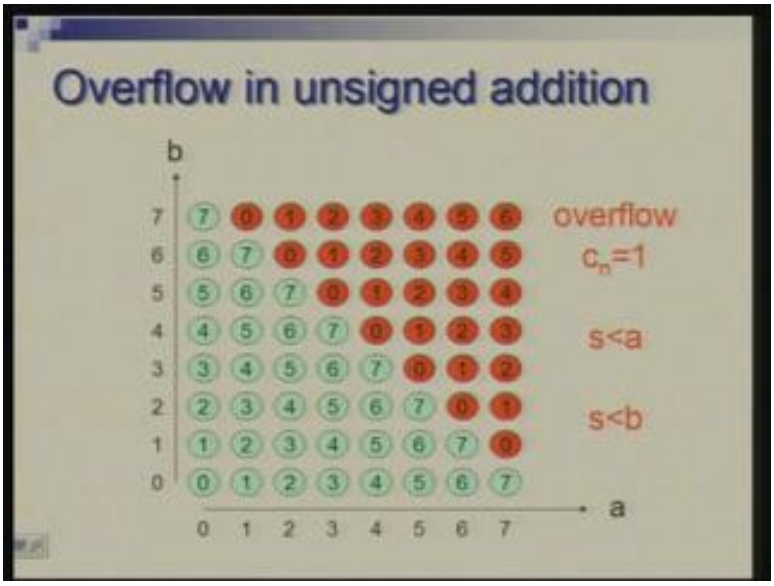
(Refer Slide Time: 15:35 min)



So basically subtraction is done but the result we are picking up is a (0, 1) setting which is decided by the logic we have just discussed. So that takes care of your slt instruction. We would also put something additional here to take care of beq. Once again it is equality test so we will do it by checking whether the output of subtraction is zero or not. So, for beq also we are asked the adder cum subtractor to perform subtraction and this gate will check whether all bits are 0 or not so I have not put inverter separately but instead of OR I have just made it a NOR gate so it will give a one output only if all the bits are 0 at its input which means the result of comparison or result of subtraction is 0 or a equal to b.

So now the ALU is capable of handling this entire set of instructions. Is there any question on this so far?
Now we go into a slightly more involved issue of overflow detection.

(Refer Slide Time: 18:40)



How do we a figure out if the result which is coming out is exceeding the range if it is beyond the limits?
Suppose we were doing unsigned addition; I am just illustrating this point in this space where we have two numbers a and b each is of three bits; just for illustration I am taking a small word size so the values can range from 0 to 7, b also ranges from 0 to 7 and of course this shows the sum. So, as the values increase the sum increases but beyond 7 which is the maximum limit the values will wrap around.

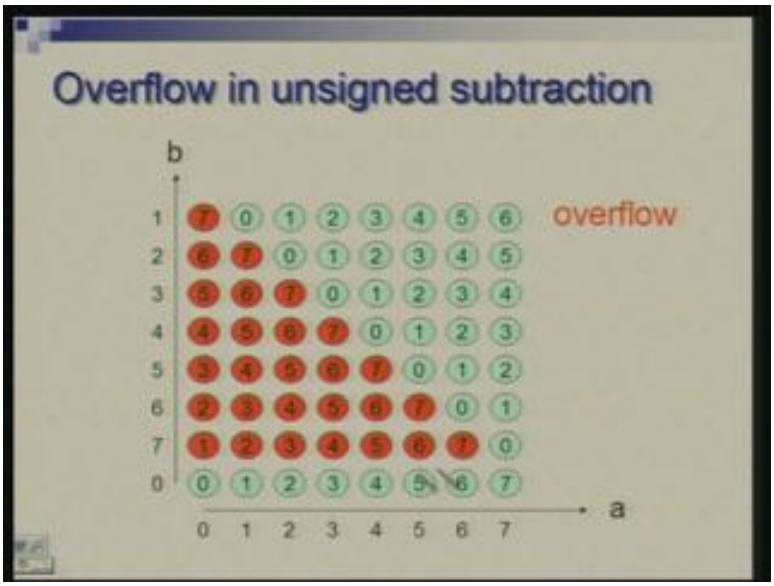
So, for example, if you take 4 and 5 the sum should be 9 but what you will get is 1 because the 8 (value of 8) will overflow go out and what will be left is 1. So anything in this domain anything in this region which is shown in red is corresponding to the overflow area and the indicators of this overflow are that there will be a carry at the extreme left end. So if you were talking of n bits c_n will be 1 the output carry from the last stage so with n equal to 32 it is c_{32} which will be 1 and that indicates overflow in unsigned addition.

Another indicator is that the sum you are getting will be less than any of the operands which we can verify here. For example, the sum here is 3 and you were trying to add 6 and 5 so this will be less than either of the two. So when you are adding two unsigned numbers the result is either more than both of them or less than both of them. So one corresponds to….. first one corresponds to correct result and the second one corresponds to overflow result.

Now let us look at subtraction, unsigned subtraction. We will assume that you have two unsigned numbers two positive values and you are trying to subtract. Also, the result we want is unsigned or positive. So what I have done basically is I am….. let us look at it again. I am assuming that subtraction is being carried out by adding 2's compliment of the number. So, for example, this row will correspond to subtraction of 6 (Refer Slide

Time: 21:49). To these numbers if I add 2, I get this equivalently from this; these numbers if I am subtracting 6 which is if I take 6 and its 2's compliment 1's compliment of 6 will be 001 and you add 1 you get 010 which is 2. So the result of subtracting 6 will look like result of adding 2. So all that I have done coming from this diagram to the next one is these numbers are same.

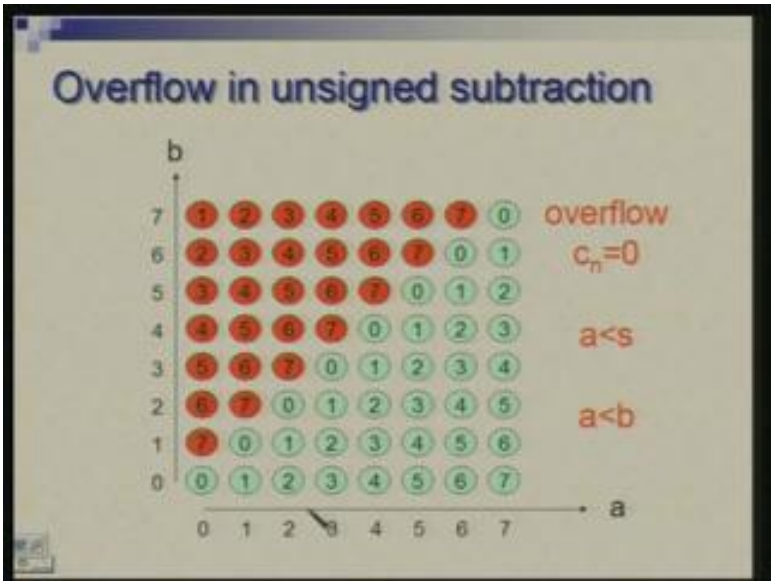
(Refer Slide Time: 22:27)



You would notice that they are progressing from zero to one diagonally, overflowing here and going to 6 but what I have changed here is I have put there 2's compliment so 0, 1, 2, 3, 4, 5, 6, 7 and in this case it is these numbers these results which are correct and these results which are invalid (Refer Slide Time: 22:56). So let us verify it. For example, from 5 if you subtract 3 you get this which is okay. If you subtract 4, 1, if you subtract 5, 0, if you subtract 6 that is an invalid case because to get a positive or unsigned result I should subtract only a smaller number from a larger number. So this red region corresponds to the case when the second number which you are subtracting is larger.

Now just to put the b value in the right order I can just flip this part (Refer Slide Time: 23:35) and you will see that this red triangle flips.

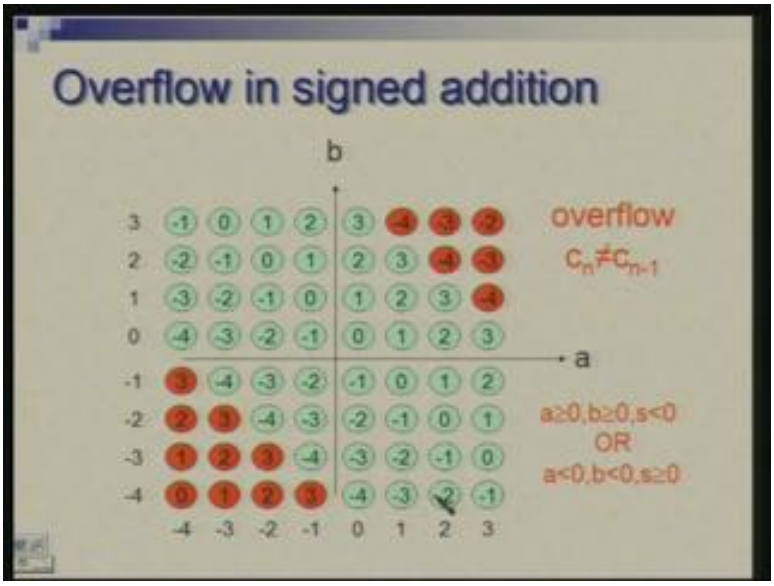
(Refer Slide Time: 23:42 min)



So basically I could have written it directly also that these are a values, these are b values so wherever b is more than a that is an invalid case and this entire region is a valid one. So obviously along the diagonal I get zero and as I move away from the diagonal I get larger and larger numbers.

So the indicators here are c_n equal to 0. In this green in this red region there is a value of c n which is 0 and in this case a will be less than both s and b; a less than b is what we know directly that indicates that things are out of range and so is the case with a less than s. The value s you are getting here is more than the value of a we have.

(Refer Slide Time: 24:52)



Now let us go to signed addition. It gets slightly more involved. Now the numbers range from minus 4 to 3. Again we are talking of same three bits so I am showing on both sides of this axis; numbers go from minus 4 to 3, a is minus 4 to 3, b is also minus 4 to 3 and it is this region where you are adding two large positive numbers or this region where you are adding two large negative numbers (Refer Slide Time: 25:25). So basically what you can see is that as you move away from this diagonal at this diagonal you have 0 and as you move you increase and when you go beyond 3 then there is an overflow. Similarly, you go on the negative side beyond minus 4 there is an overflow.

So what are the indicators of overflow condition? Let us look at this first that either you are in this region or in this region in that top red region a and b both are non-negative but the sum is negative. You are adding two positive numbers, positive or non-negative numbers but the sum is you are getting a negative number which is a anomalous condition and that is overflow or you are adding two negative numbers and you are getting a positive sum that is a region so that is also overflow. So this is one way of checking that there is an overflow.

The other way would be that the last two carries will be of opposite polarity so what it means is that, for example, there is a carry flowing into the MSB but there is no carry flowing out from the MSB that is one condition or there is no carry flowing into the MSB but there is a carry flowing out. So the first condition that there is a carry into MSB it means that the number is turning into negative when it is not supposed to. You are going towards a large positive number side but the number is exceeding on the positive side and it is turning the sign to negative.

On the other hand, you are supposed to have a negative value but a carry is turning it into positive. That means there is a carry into the MSB, I am sorry, there is no carry into MSB

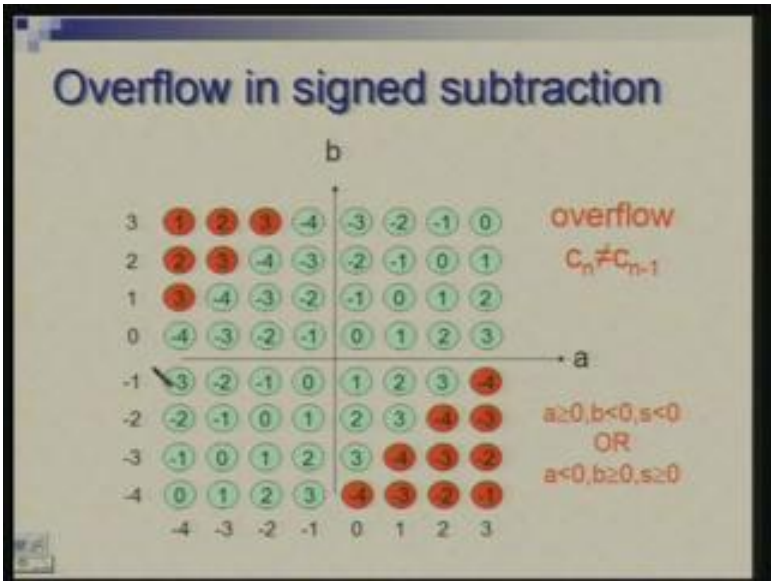
so it is turning into positive but there is a carry out which was indication of normal negative numbers. So this is another way of looking at the overflow.

Is there any question about this; is that clear?
Now let us look at signed subtraction. Signed subtraction what I will do first is take the same values and appropriately change b to its negative value. So again we will do subtraction by adding 2's compliment value. The pattern of numbers will remain same; 0 will remain 0. This will become minus 1 minus 2 minus 3 and 1, 2, 3; minus 4 will remain minus 4 because there is nothing like plus 4 in three bits. In three bits the range is from minus 4 to 3 so the numbers are same; ignore the color for the moment.

What I have changed here is that the sign of these three rows and these three rows (Refer Slide Time: 28:46) has been changed. So the color has changed here because here I am trying to subtract minus 4 value from these values. so with subtracting minus 4 from minus 4 I get 0 which is correct, subtracting minus 4 from minus 2 I get plus 2 which is also right but subtracting minus 4 from any of these non-negative numbers would mean that I am getting a value which is highly positive which is beyond the range so these are the ones which turns to red. So the sign of others remain unchanged as you can see.

So, for example, let us check it out. Suppose I subtract….. I take minus 3 here and plus 2 here so 2 minus minus 3 gives the plus 5 which is beyond the range so it is this region which continues to be beyond the range (Refer Slide Time: 30:03), this region is also giving me wrong values. I should be getting positive values here I am getting negative, I should be getting negative values here but I am getting positive. So now let us flips the rows to get the right order here and this is the picture I get.

(Refer Slide Time: 30:24 min)



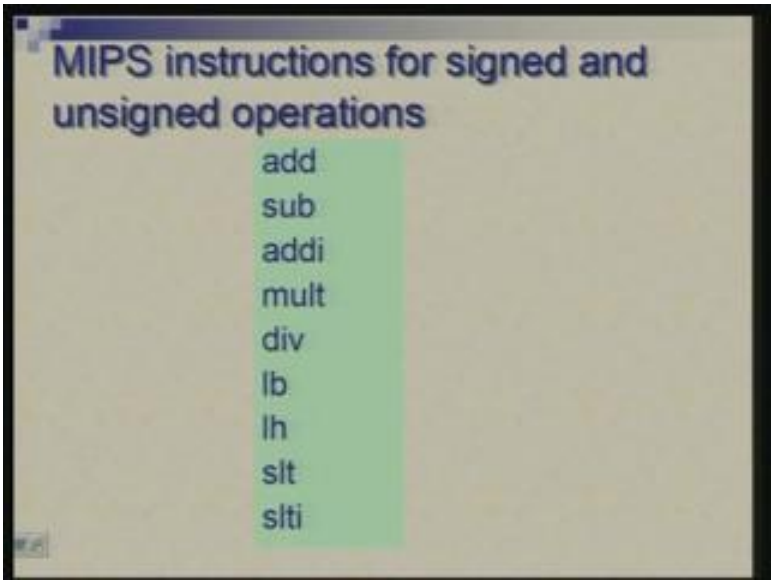
So basically in case of signed addition I had problem in these corners but now the problem areas are this corner and this corner (Refer Slide Time: 30:34). So the internal

sign the indication is that when a is non-negative and b is negative and you get a negative result then the result is wrong. Or, if a is….. well, the first one corresponds to this region. corresponding to this region a is negative, b is non-negative and s is non-negative so again something is wrong so that is the overflow area and this indication remains same. There is a change of sign by getting a wrong carry into the final bit position.

Now, with this understanding of overflow we know how to detect overflow in signed case or unsigned case. So, when you are trying to use subtractions for less than comparison than one has to be careful. we simply looked at the sign of the result and took that as an indication of whether a is less than b or not but that works only if there is no overflow; when there is overflow this result needs to be flipped back. So you need to look at the sign but if there is an overflow just reverse your decision because that decision simply goes reverse when there is an overflow. This is a correction which needs to be applied.

Now let us get back to the instruction MIPS and look at the precise difference between what they do in case of signed and in case of unsigned numbers.

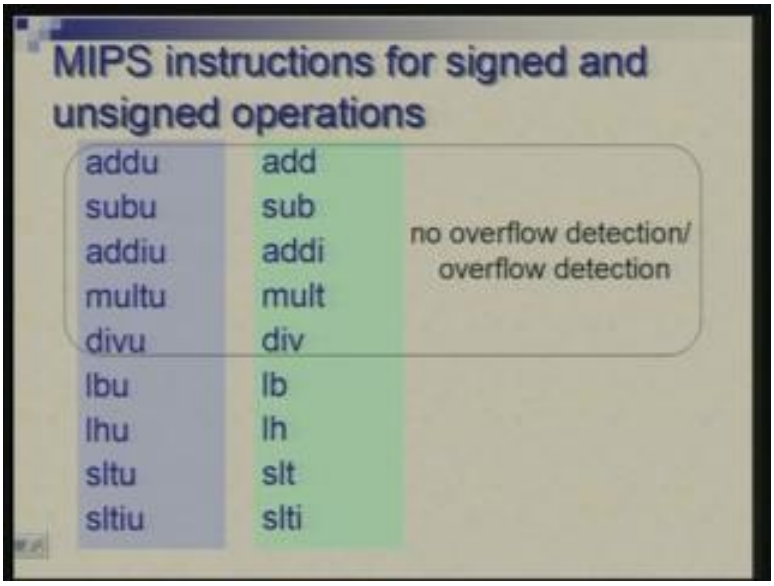
(Refer Slide Time: 32:40)



These are the instructions which have their u counterpart: add, subtract, add immediate, multiply, divide; we have not discussed multiply, divide in detail but you roughly understanding the meaning. lb is the load byte, lh is the load half word, slt and its immediate counterpart is slti so these are the instructions which have their u counterpart so u suffix means unsigned.

What exactly is the meaning?
The meaning differs from instruction to instruction. So, as far as these instructions is concerned there is first of all a question of whether overflow is detected or not detected.

(Refer Slide Time: 33:27 min)

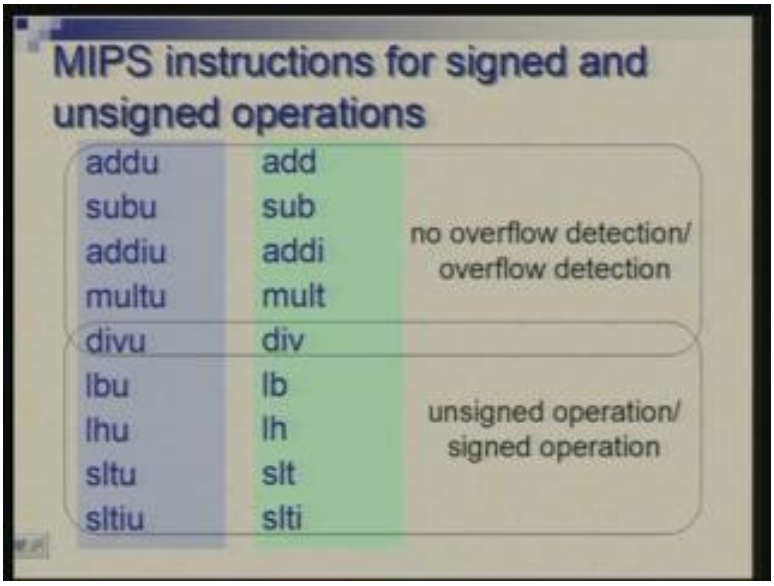


Therefore, in instruction with the u suffix overflow detection is not done whereas in instructions which are in the green column overflow detection is done.

What happens when overflow occurs in the processor?
When some instruction results in overflow the processor has to take some special action. It could be halting the program or it could be printing an error message or something so that we will see later. Right now we will remember that it is an abnormal condition which changes the course of the program so we will think of it at that…….. whereas in the instructions which are grouped now these instructions (Refer Slide Time: 34:19) there is also difference in the result. That means from between these instructions add and addu the result which will go into the destination register will not differ whether it is add or addu or subtract or subtractu and so on. So in these instructions which are outside this the result will be same in both cases whether you are using with u or without u the only difference is that overflow is being detected or not detected.

In these instructions (Refer Slide Time: 35:01) the result would differ depending upon which instruction you are using. So you would notice that divide actually falls in both these categories that means there is a difference of overflow detection and also there is a difference of result. As far as the add and subtract is concerned we have seen that with 2's compliment of representation the result actually does not differ. It is only a matter of interpretation, you take two numbers and perform addition; if you interpret these numbers as unsigned numbers you get correct unsigned sum, if you interpret these as signed numbers you get correct signed sum as long as there is no overflow. So, when overflow occurs then of course we are discarding the result and taking some special action. So, as long as there is no overflow the results are identical. That is not the case with multiplication. But for some reason whether you take mult or multu the result produced is same. multu is not multu unsigned multiplication in the true sense.

(Refer Slide Time: 34:12 min)



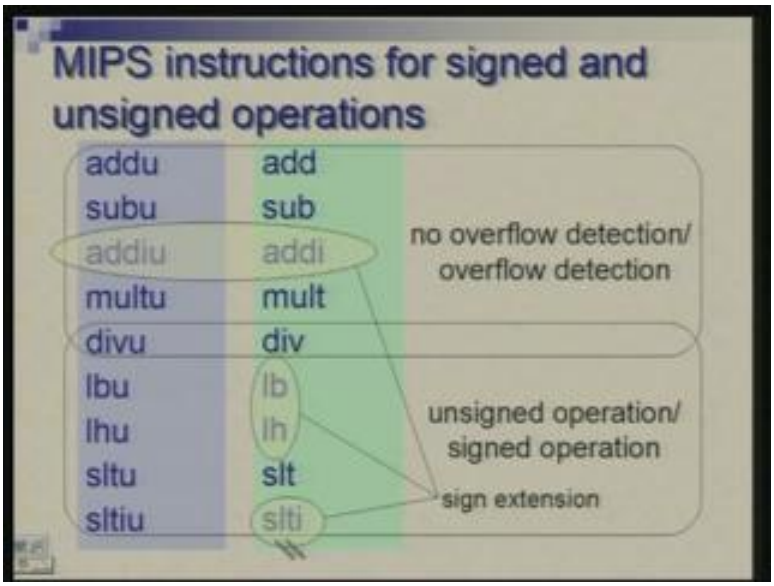
Now, comparison also we have seen that for signed comparison and unsigned comparison the result would differ. So, if you take two numbers let us say simply 2 and minus 3 if you are doing signed interpretation then 2 is not less than minus 3 but if you interpret these as unsigned numbers so minus 3 will actually look like a large positive number when you take unsigned interpretation and you will find that first one is less than second. So the result of comparison differs depending upon whether you are interpreting this number is signed or unsigned and that difference you will find among these instructions.

As far as load byte load half words are concerned since these instructions are not loading a full word in to a register you need to define what goes into the remaining part. So the remaining part gets filled with 0s if the instruction is lbu or lhu and sign extension is done if the instruction is lb or lh because lb lh will treat the byte or half word which you are getting from the memory as a signed number and the sign will fill up the remaining bits.

So, it is these instructions which are circled where signed extension is done. So in lb and lh extension is done but not in lbu, lhu. In add immediate instruction again the constant is 16 bits but operation performed is over 32 bits so there is a question of filling up the remaining 16 bits. In add immediate and add immediate unsigned in both cases sign extension is done so in that sense this u is a slightly a misnomer because the number is taken as a signed number and extension is done. Hence, it is only that interpretation of result would be an unsigned result but the constant is sign extended in both cases.

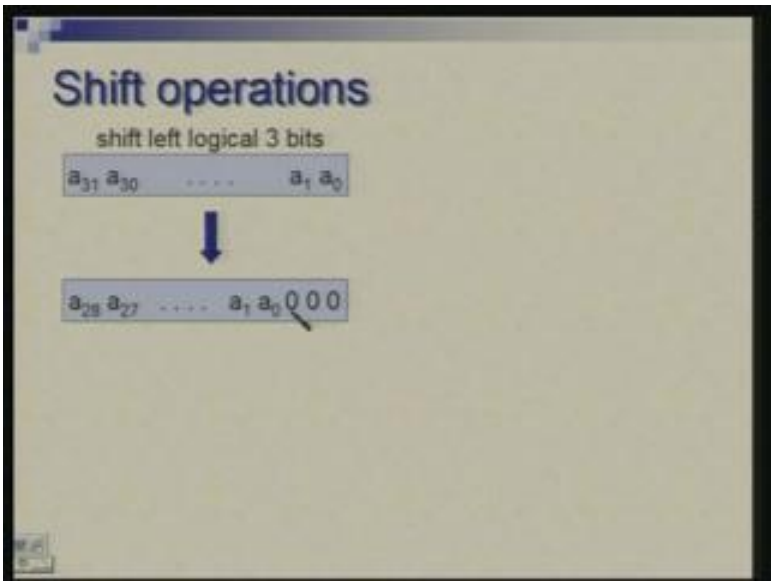
In slti sign extension is done but not in sltiu so this is a true interpretation of unsigned comparison. Is that clear? There are three issues to be seen when you are talking of these instructions: whether overflow is detected or not detected that needs to be seen, whether operation has different meaning in signed case and unsigned case and are those meanings preserved and thirdly, whether sign extension is done to convert small numbers into large numbers.

(Refer Slide Time: 37:55 min)



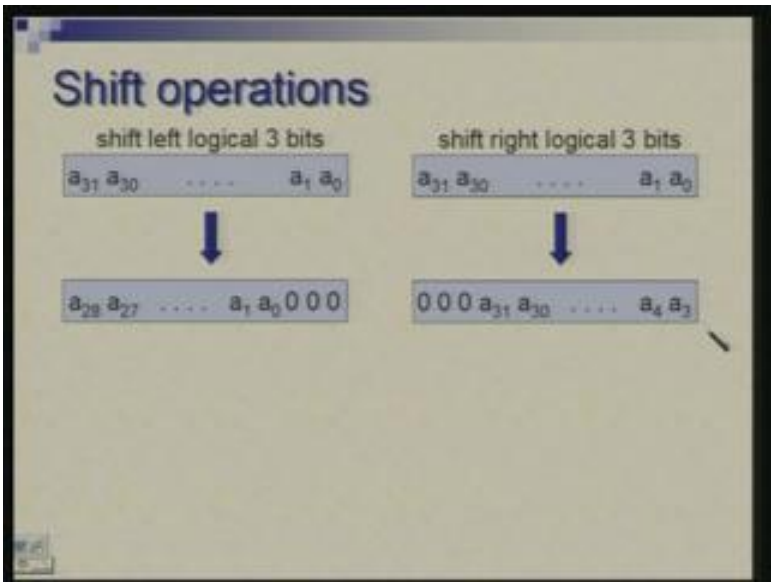
Finally we move to another kind of operation which are called shift operations and these are used to manipulate bit pattern or extracting bit fields out of a long bit pattern and so on. One of the operations is shift left logical. So this illustration shows shifting left by three bits. Suppose you have some pattern in a register a_0 to a_{31} shifting it left by three bits means all bits are pushed to left and on the right side you fill in some 0s. So you would notice that three bits have been thrown out and lost from the left side; on the right side you have three 0s which have been filled in.

(Refer Slide Time: 40:11)



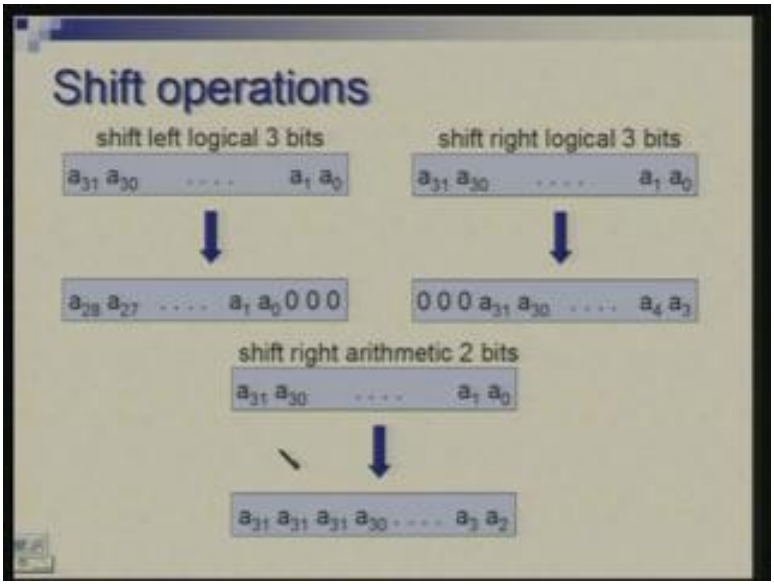
Its counterpart is shift right logical where you are shifting the pattern of 1s and 0s to right so 0s get filled in from the left side and you lose bits on the right side.

(Refer Slide Time: 40:29)



Now, with the right shift there is a complication which is a rising here. If you are interpreting these numbers as signed numbers then possibly sign could change in this operation. Why would you like to interpret these numbers as integers? If you do so you can take shift left as multiplied by two operations. 1 bit left shift is multiplied by 2, 3 bit left shift is multiplied by 8. Similarly right shift is integer division by some power of 2 so shifting by three bits you would like to see it as division by 8. So, if the numbers are unsigned numbers this is fine there is no problem but if the number is a signed number and you are shifting it right hoping to have it interpreted as division by power of 2 then problem may occur because you are possibly changing the sign. If a_{31} was 1 earlier now it is made 0 so there is yet another instruction yet another operation which is called shift right arithmetic. So what we do here is something like shift right logical but instead of filling in 0s you do sign extensions. So the spaces which I am getting are vacated or filled by signed bits.

(Refer Slide Time: 42:24)

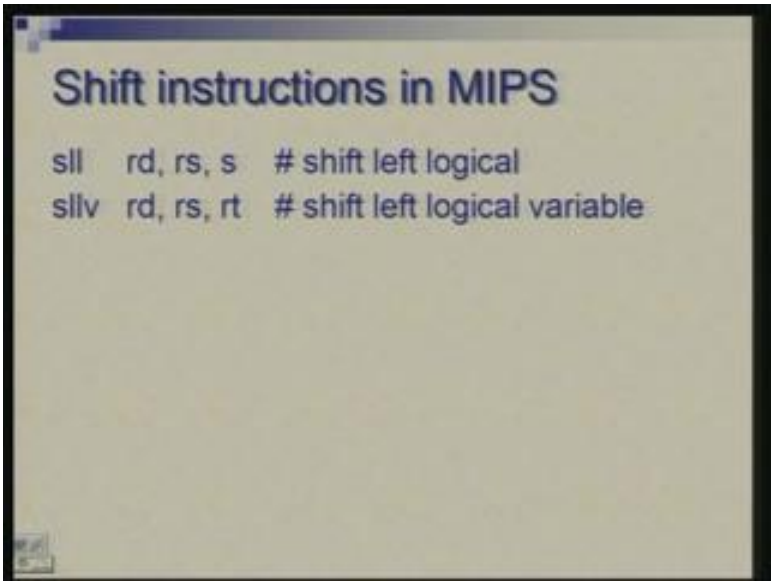


For example, here the picture shows shifting by 2 bits and a_{31} is replicated to fill up the vacating positions. So, if this is done then you are correctly dividing a signed number a signed integer by a power of 2. Of course what you are getting is the quotient the remainder is being thrown off.

So yeah [Conversation between student and Professor… (42:55)….. in the multiplication it may also happen if a_{31} was 0) yeah that is the case of overflow. If number was large let us say you had two 0s on the left and remaining there was a 1 after that; suppose a_{29} was 1 a_{29} and a_{28} they were 1s after that and these were 0s so there is a large number which you were trying to shift left multiplying it by 8 and if a 1 comes up here it might look like on negative number now so this is a case of overflow actually. Hence, if you were actually keeping a_{31} here and you are not allowing other bits to enter then you are getting a wrong value in any case so there is nothing which can be done here. But here (Refer Slide Time: 43:41) we can, since we are reducing the number it is not an overflow which is occurring but you are artificially putting 0s there so that can be prevented.

What are the instructions in MIPS which do this?
There is an instruction called sll, shift left logical; it takes two registers: destination register and source register and a constant so this is a constant which can vary from 0 to 31 and this constant is in the shift amount field. You remember, that when I talked of add instructions I said that there is a field which I labeled as shift known but I said it is not used. So this is the instruction where it is used that 5-bit field can carry a number 0 to 31 and that indicates the amount of shift you are having. But once again still we have one field which is not being used; rt field is not being used here in this instruction so there is still some wastage.

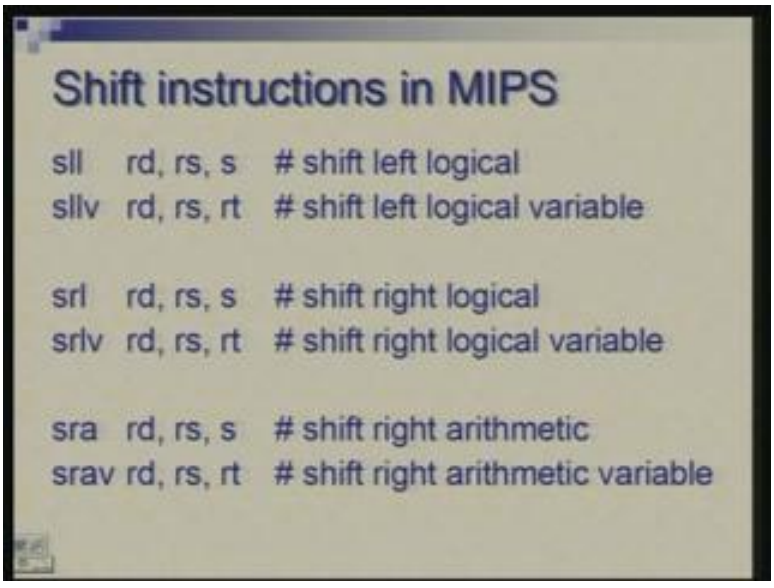
(Refer Slide Time: 44:54)



There is another version of this instruction sllv which stands for shift left logical variable. Instead of providing a constant as a parameter as the amount of shift here we use a third register to specify this number. This is like immediate specification and this is like a register mode so this value is part of this instruction here it is part of the register.

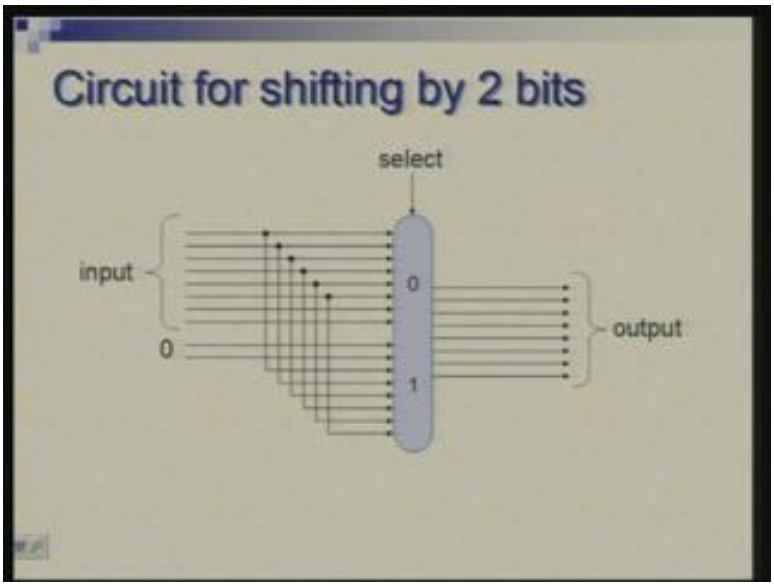
Now, since this register is a 32-bit register it can have a large value. But what we are looking at is the least 5 bits of that. Similarly, there is srl for shift right logical and srlv shift right logical variable and shift right arithmetic and its variable part.

(Refer Slide Time: 45:45 min)



What kind of hardware is required to do this kind of……..?
It is very simple.

(Refer Slide Time: 45:55)

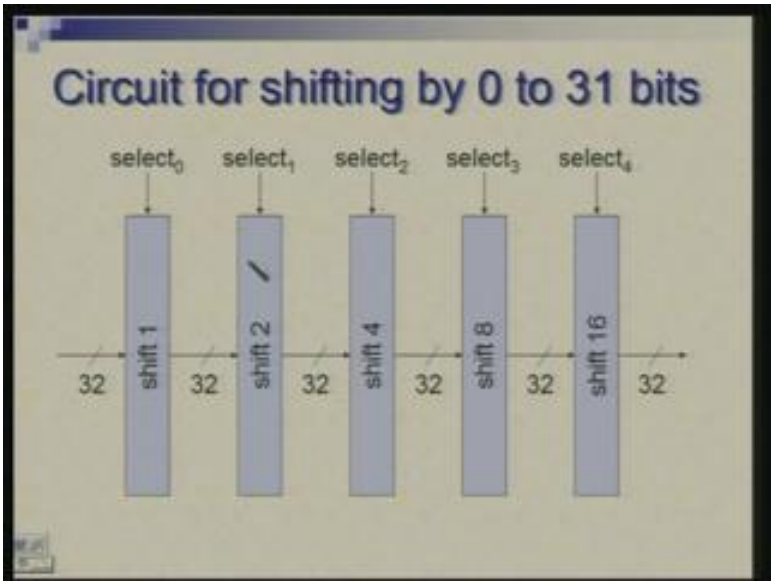


Suppose you are talking of 8-bit number and you want to shifted by two position and I am showing a circuit where you want to do it selectively; either you want to carry it straight or you want to shift it so these are let us say 8 bits of a number (Refer Slide Time: 46:20), you give it to a multiplexor and wire the other input of the multiplexer with these positions shifted. So let us say this is the LSB which I am connecting to the third position and so on; the first two positions are filled with 0s. so this multiplexor will either select this set of eight inputs and pass them on to output or this set of eight inputs and passed them on to output.

You would notice that in the second input these 2 bits are getting left out and 0s are filling into the left position. So simply a multiplexor with some wiring at the input can select between no shift and shift by 2 bits.

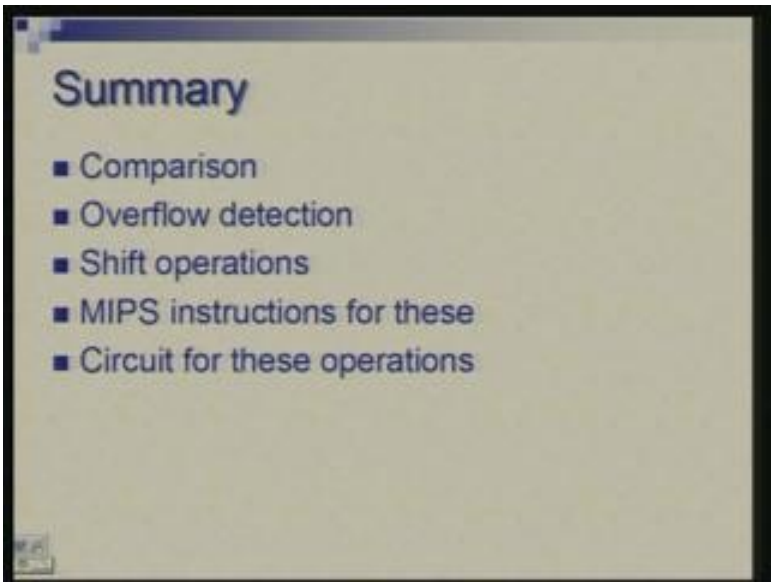
What do you do if you want arbitrary shift; anything from 0 to 31 you can put multiple units like this (Refer Slide Time: 47:19). So there are five of these precisely. The first one selects between no shift and 1 shift. This can take care of 0 shift or 2 shifts, this selects between 0 shift and 4 shift and so on. So, by suitably choosing a combination; suppose you want a total of nine bit nine position shift so you will select 8 and you will select 1 so the combination will give you shifting by nine positions. So basically that 5-bit number which specify the amount of shift is used as a control input for these five shifters.

(Refer Slide Time: 48:03)



I am not showing the details of the shifter but each of these shifters is essentially a multiplexor with appropriate wiring. The shifting in the wiring is according to this number by which this box is required to shift so this combination suitably can shift from anywhere from 0 to 31 positions.

(Refer Slide Time: 48:28 min)



To summarize what we have seen today; we have seen how comparison can be done either directly or using subtraction. We have discussed how to do overflow detection in case of sign addition, subtraction or unsigned addition, subtraction. We have discussed

shift operations; we have seen what instructions are used for comparison and what instructions are used for shifting, how the instructions differ in terms of overflow detection or excess of overflow detection and we have also looked at the circuits to support these instructions. I will stop at this, if you have any questions? Thank you.