So, if the two in a positive numbers; with sign 0 are added and yields a negative number we will see why what is the reason. So, if there are it's a signed arithmetic for example, assume and there are two numbers and you add them and then there is an overflow.

Because you all know in digital design what is the concept of an overflow, but we will also look in details with some examples in this. For example, so ah like as I have told you let us take an unsigned number already we have taken the example; so, let us take 1111. So, definitely if I if I there will be a carry over there and in fact, in fact, if I say it cannot be accommodated in the 4 bits. So, if I assume that the result has to be given in 4 bits and I have two numbers like 1000 and all triple ones.

So, of course, there will be overflow will be generated and also we will see the idea. So, if the negative number positive numbers whatever happens. So, in other words in a digital arithmetic if a overflow is generated based on the number of bits you store for the answer and number of bits you store for the operands if it's a overflow is there it bit will be it will be set other case it will be reset.

Like for example, if you add 0000 with triple 0 with. So, 0004; so, the answer is 1000 unsigned arithmetic of course, no overflow is generated the overflow flag is reset in this case very simple. Equality as I told you if it's a; it will this is restricted to a compare instruction.

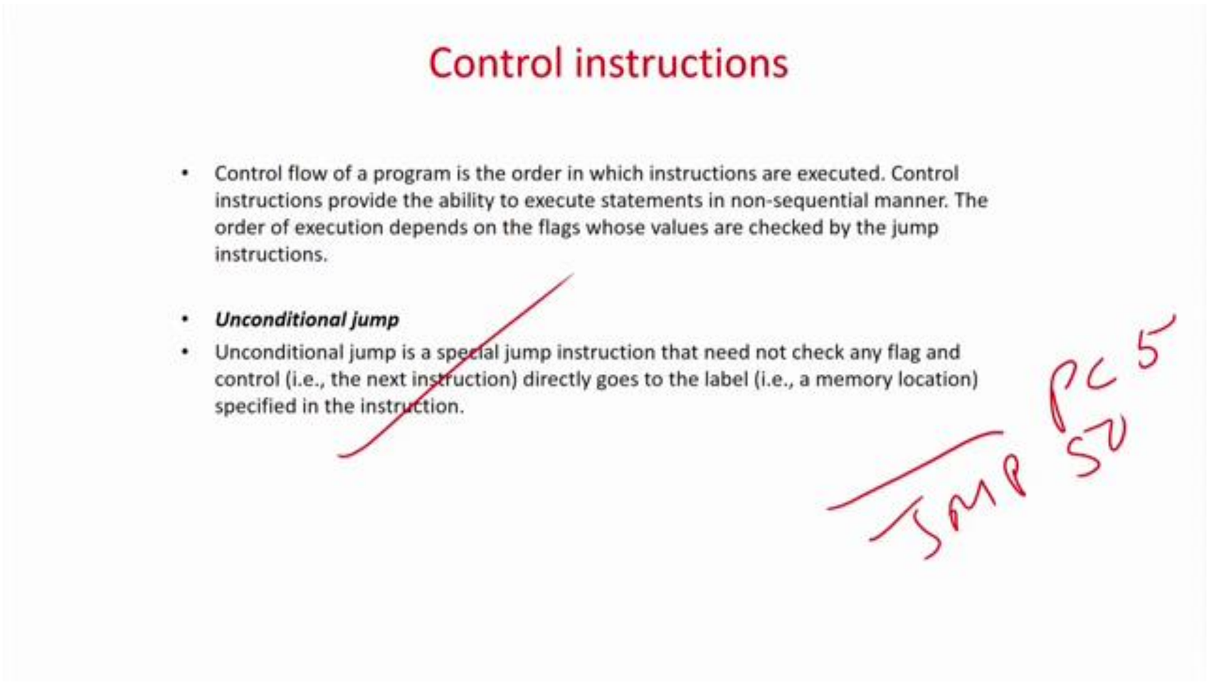
So, if there is an instruction called compare and then if the two numbers are equal then this flag is set. So, it is a comp instruction and you give two operands and if they are equal the answer with that bit is set in the flag register otherwise its reset.

Interrupt enable so this is also a flag in which case you allow an interrupt to be occurred or not; that means, a main code is running whether we will allow some other code to interrupt if you allow it. So, it will be its flag will be 1 and if you are not allowing such an interrupt to interrupt your code then the that flag will be set to 0.

So, at this point of time I am not going to elaborate more on interrupt enable flag because there a full unit and module which is dedicated to I/O and interrupts. And simply like supervisor mode also some of the some of the codes may like execute in the supervisor mode. So, in all those cases you have to set that flag and if you are not allowing any code to run in the supervisor mode or user privileged or super user privileged mode you can reset this bits.

So, these two will be discussed later whenever you are going to some advanced modules mainly we will be talking in a further down the line on the I/O module about interrupts. Supervisor mode is something also related to operating system and executing in a coordinate super user mode etcetera, but in details we will be looking at interrupt flags whenever we will be discussing the chapter on them. Now based on this some of the very important flags for us is the sign flag, zero flag, carry flag, parity flag, overflow flag and equality flag. So, these are some of the most typically important flags which will be used in everyday life of designing control instructions.

(Refer Slide Time: 21:20)



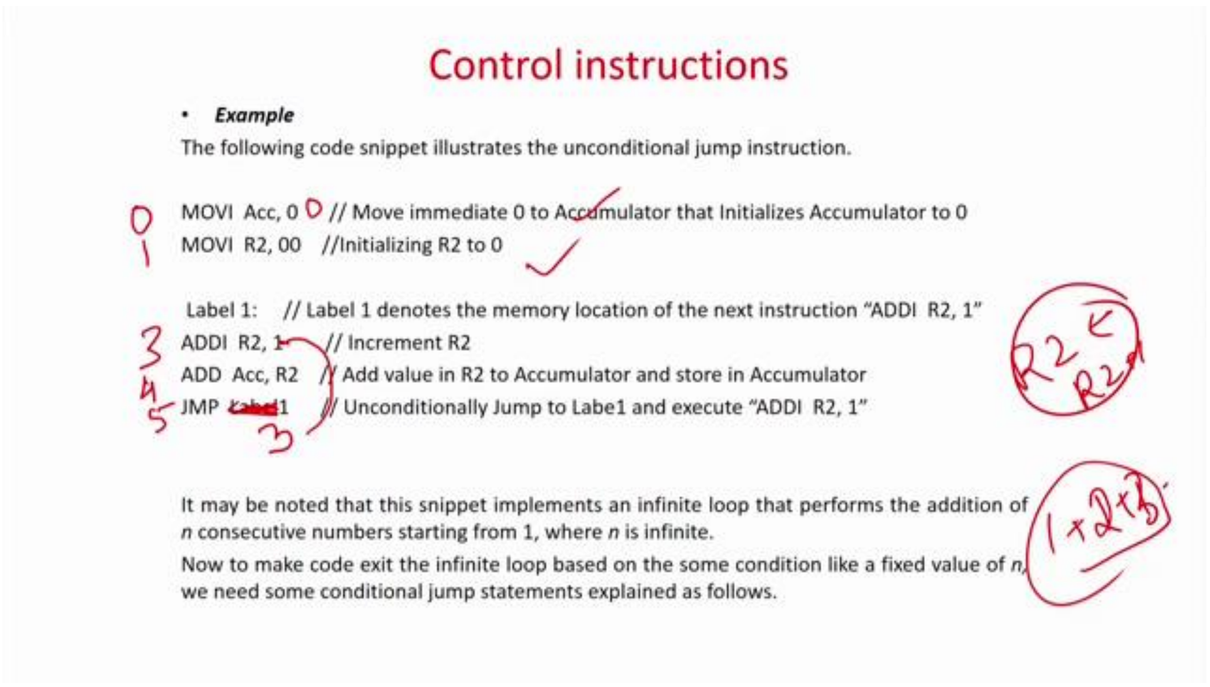
Control instructions

- Control flow of a program is the order in which instructions are executed. Control instructions provide the ability to execute statements in non-sequential manner. The order of execution depends on the flags whose values are checked by the jump instructions.
- ***Unconditional jump***
- Unconditional jump is a special jump instruction that need not check any flag and control (i.e., the next instruction) directly goes to the label (i.e., a memory location) specified in the instruction.

So, now we will be looking at some of the typical control instructions based on the flags. The first simpler one is the unconditional instruction; unconditional jump that is you are at this memory location the PC is say 5 this is the 5th memory location where the code is using then you can say jump 50. So, without looking at anything the program counter is going to become 50.

Whatever instruction is present in the memory location 50 will be executed, that is a very simple unconditional jump instruction no flags are required for that.

(Refer Slide Time: 21:50)



Control instructions

- ***Example***

The following code snippet illustrates the unconditional jump instruction.

```
0  MOVI  Acc, 0 0  // Move immediate 0 to Accumulator that Initializes Accumulator to 0
1  MOVI  R2, 00    //Initializing R2 to 0

    Label 1:    // Label 1 denotes the memory location of the next instruction "ADDI  R2, 1"
3  ADDI  R2, 1     // Increment R2
4  ADD  Acc, R2    // Add value in R2 to Accumulator and store in Accumulator
5  JMP  Label1     // Unconditionally Jump to Labe1 and execute "ADDI  R2, 1"
         3
```

$R2 \leftarrow R2+1$

It may be noted that this snippet implements an infinite loop that performs the addition of *n* consecutive numbers starting from 1, where *n* is infinite.

Now to make code exit the infinite loop based on the some condition like a fixed value of *n*, we need some conditional jump statements explained as follows.

$1+2+3$

An example; so, move accumulator 0; so, in this case move immediate so, they have already mentioned about the accumulator ah this depends on the mnemonics or the instruction type of this machine. So, they say that move immediate accumulator 0; so, move the value of 0 to accumulator sometimes as I told you many times you can also drop this.

We say that move immediate 0; it means the default destination operand is the accumulator. So, MOVI 0 means the value of 0 will be loaded to the accumulator then MOVI $R2$, 00. So, initialize this is also move immediate $R2$ 00 so in fact, it is better to write this one 00 because from the size.

So, it is saying that move immediate ah $R2$ 00 that is you are resetting register $R2$ as well as you are resetting accumulator to 00. So, in this case I am assuming the accumulator is an 8 bit accumulator. So, these two are just initializing accumulator to 0 initializing user register to zero. So, in this in this instruction in this machine they are explicitly solved.

So, in this case they are explicitly keeping the value of accumulator in the instruction itself mentioning let us keep it in that way. And then another very important thing in when you are doing conditional instructions is the label. So that means, we can attach some labels to the instruction. So, these are not actually will be written in the memory when the code will be executed, but actually it is a label.

So, label means it is same as a name to the instruction for example, the label 1 it is saying that ADDI $R2$, 1; that means, whatever is the content of $R2$ it will be added with 1 and the value will be given to the $R2$; that means, $R2$= $R2$ +1 that is nothing but increment $R2$.

So, the label 1 colon this 1 it means that ADDI $R2$, 1 this has the label 1 then what I am doing? You are adding accumulator to $R2$; so, whatever is in the value of $R2$ is stored back to the accumulator. So, now, $R2$ is dumped into sorry add the value of accumulator to $R2$.

So, whatever is in the accumulator will be added to $R2$ and stored back to the accumulator see the accumulator is equal to accumulator plus $R2$. Then jump to label 1 unconditionally again you jump back. So, what do I mean by jump at label 1 means I want to jump and execute this instruction.

That is from jump label 1 means jump to add immediate this instruction; you want to jump to this instruction. So, then how can I tell that jump unconditional to this instruction. So, some name has to be given. So, label is nothing but the name which is given to this instruction; so, this label 1 is a name which is given to this instruction.

That is just after adding $R2$ to accumulator and storing back the value in accumulator; again I wanted to jump back to this one. That means, you are going to execute these two instructions in a indefinitely; there is no condition you execute this then you execute this; that means, $R2$ = $R2$ +1 that is you are incrementing $R2$.

And then again you are going to add the value of $R2$ to accumulator and save it to the accumulator. That means, you are doing 1 +2 + and you are doing 1 +1 +1 and you are actually keeping on doing it. And it is the infinite loop; if you like see what is happening it is the infinite loop. So, add immediate $R2$ to 1 that means $R2$ is initially reset to 0.

So, every time here incrementing 1, 2, 3, 4 and you are adding the same value to the accumulator. So, you are adding 1 +2 +3 …, but as this your unconditional jump label is the name of this instruction. So, you are jumping over there; so, you are actually having a infinite loop there is no exit from this loop. So, this just shows two very important things forget about this ah infinite loop business; the idea is that jump unconditional means without checking anything you jump over here and we are doing 1 +2 +3 so, on. And basically as I want to say jump from this to some instruction.

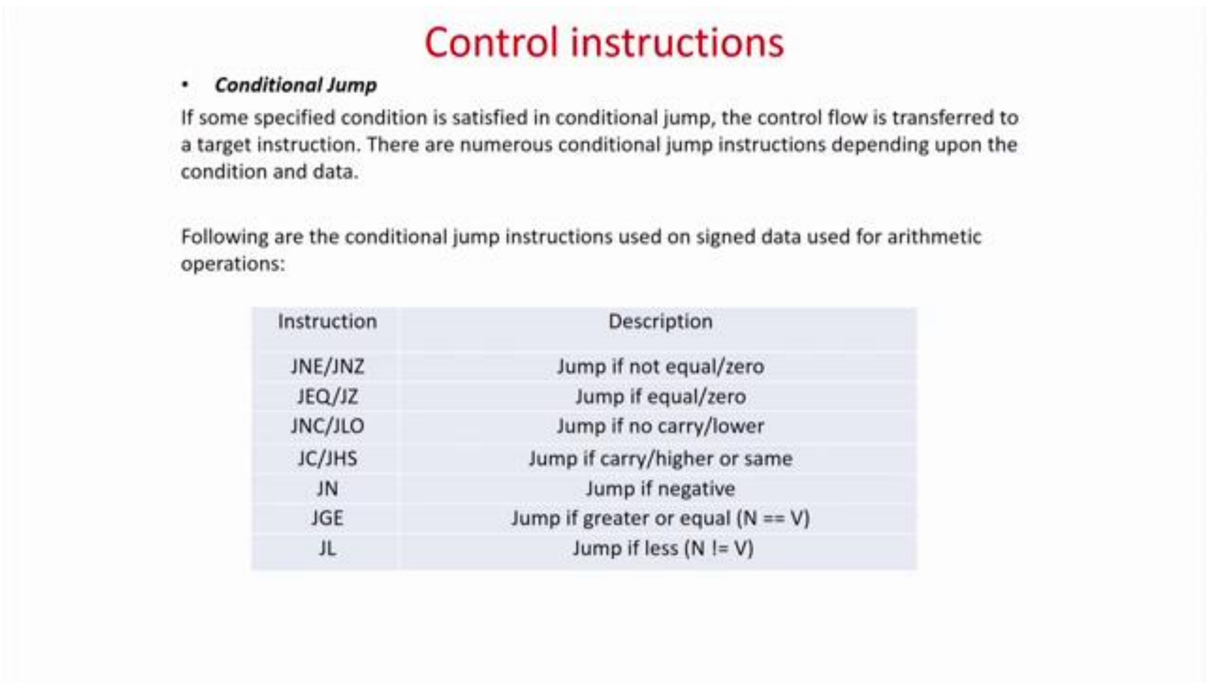
So, some label is there; so, label is the name of the instruction where you want to jump. So, in this case I have given the name of this instruction. So, whenever I will load the code in that case I will replace the value of name of the label with a memory location.

Say for example, I load this at memory location 1, I load this as memory location 2; and then basically ah this instruction is say memory location 3 because label and this is in the same row. So, label equation number label number 4; so, this is instruction number 5; sorry memory location number 5. So, it will say jump to label 1; so that means, jump to this memory location label 1.

So, label 1 is nothing, but memory location 3 where the instruction add $R2 + R1$ is there. So, when the code will be assembled and linked and loaded you will replace it with the value of memory location 3 that means after at 5 that memory location number fifth, you unconditionally jump to memory location number 3 and you keep on doing it 3, 4, 5, 3, 4, 5, 3, 4, 5 it will be continuously executing.

But that is all these labels etcetera are replaced with the memory location values when they code is parsed that is when the code be assembled. So, these are all thought in details in a course called assembler linker loader that is the system programming, but for us it is enough to understand right now that from jump to label 1 means you are jumping from this one to the name to this instruction whose name is label 1 and in this. In fact, this is a it is a loop instruction basically because there is no condition it has to be say infinite loop.

(Refer Slide Time: 27:16)



Control instructions

- ***Conditional Jump***

If some specified condition is satisfied in conditional jump, the control flow is transferred to a target instruction. There are numerous conditional jump instructions depending upon the condition and data.

Following are the conditional jump instructions used on signed data used for arithmetic operations:

Instruction	Description
JNE/JNZ	Jump if not equal/zero
JEQ/JZ	Jump if equal/zero
JNC/JLO	Jump if no carry/lower
JC/JHS	Jump if carry/higher or same
JN	Jump if negative
JGE	Jump if greater or equal (N == V)
JL	Jump if less (N != V)

Right now there are as we have told that actually jump unconditionally used many times, but the main heart of instructions on control instructions are basically on conditional instructions.

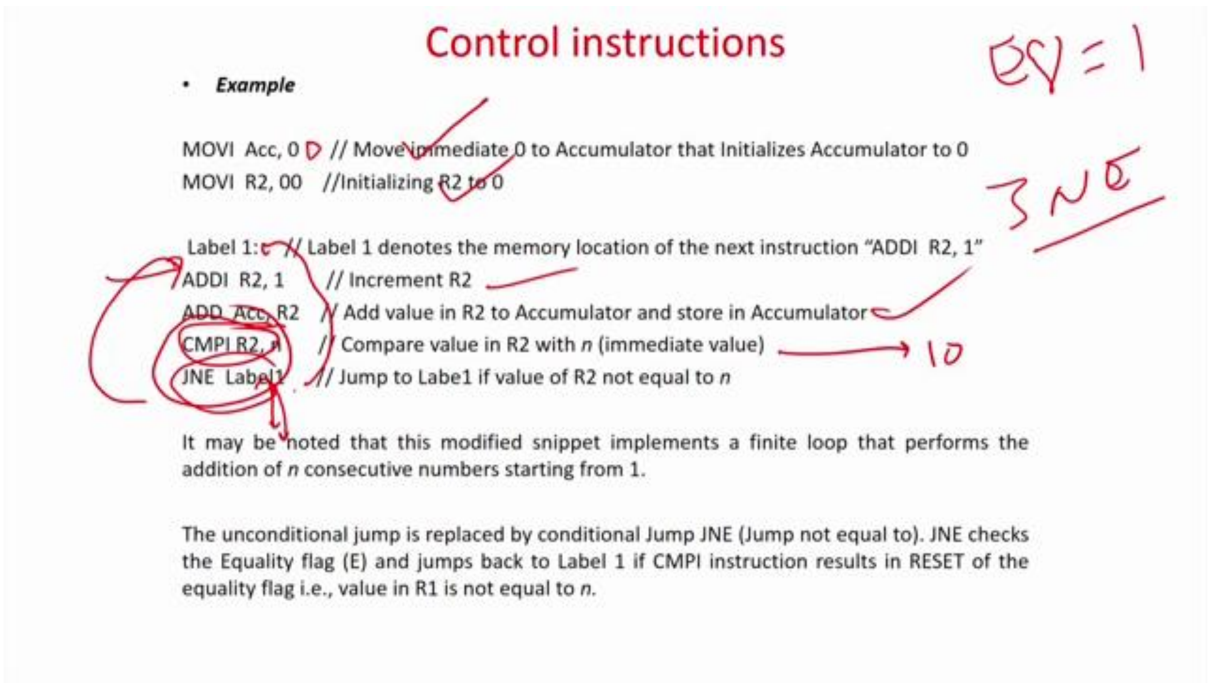
Like some of the examples it is given jump NE or jump on zero, there jump if not equal to zero jump NE jump not equal to zero or jump not zero right. JEQ jump is equal to jump on Z, JNC JLO jump if no carry or jump if carry.

So, you can go through it several type of instructions they are there jump L jump if less, JGE jump if greater than or equal, JN jump if negative.

So, based on the flag like it is saying that jump NE or jump not Z in. In fact, this time you are checking the jump not zero means you are checking the zero flag. Jump NE jump not equal to means you are checking the equality flag.

Jump negative you are checking the negativity flag that is the sign flag. So, if the sign flag is set; that means, it’s a negative number jump ah if less; so, you can check the equality flag. So, all those different flags will be present based on the flag values or the flag registers available you can correspondingly decide or design your instruction set on the jumps.

(Refer Slide Time: 28:33)



Control instructions

- ***Example***

```
MOVI  Acc, 0  // Move immediate 0 to Accumulator that Initializes Accumulator to 0
MOVI  R2, 00   //Initializing R2 to 0

 Label 1:  // Label 1 denotes the memory location of the next instruction "ADDI  R2, 1"
ADDI  R2, 1     // Increment R2
ADD  Acc, R2   // Add value in R2 to Accumulator and store in Accumulator
CMPI R2, n     // Compare value in R2 with n (immediate value)
JNE  Label1    // Jump to Labe1 if value of R2 not equal to n
```

It may be noted that this modified snippet implements a finite loop that performs the addition of *n* consecutive numbers starting from 1.

The unconditional jump is replaced by conditional Jump JNE (Jump not equal to). JNE checks the Equality flag (E) and jumps back to Label 1 if CMPI instruction results in RESET of the equality flag i.e., value in R1 is not equal to *n*.

Like we are now going back to the same example of ah the same thing that there is a loop, we are resetting the value of accumulator. we are resetting the value of $R2$ that is we are adding 1 +2 +3 +4 like that, but in the previous step we were actually jumping it unconditionally back to the initial one; reset accumulator and register 2 then every time it was making $R2$+1 incrementing $R2$.

Then every time you are adding the value of $R2$ to accumulator repeatedly; that means, we are not we are not exiting out of the loop based on some condition. Here we are actually using a loop, the same example we are going to take, but here we are going to come out based on certain conditions. Like if you look label 1 is the name of this instruction then add $R2$ to accumulator ADD $R2$, $R1$.

That is increment the value of $R2$ same thing as above, add the value of $R2$ to the accumulator that is add accumulator to $R2$; here the conditional step comes in. That is CMPI that is compare the value of $R2$ to n; that means, you are incrementing the value of $R2$ say I want to add 1 +2 +3 +4 up to 10.

So, this n is in this case going to be 10. So, after doing the add $R2$ to $R2$ to accumulator and saving back the value of accumulator we are going to check whether $R2$ has reached the value of 10 or not; here it is 10. So, there is a comp CMPI instruction and whenever $R2$ will be equal to 10; that corresponding equality flag will be checked it will be made set.

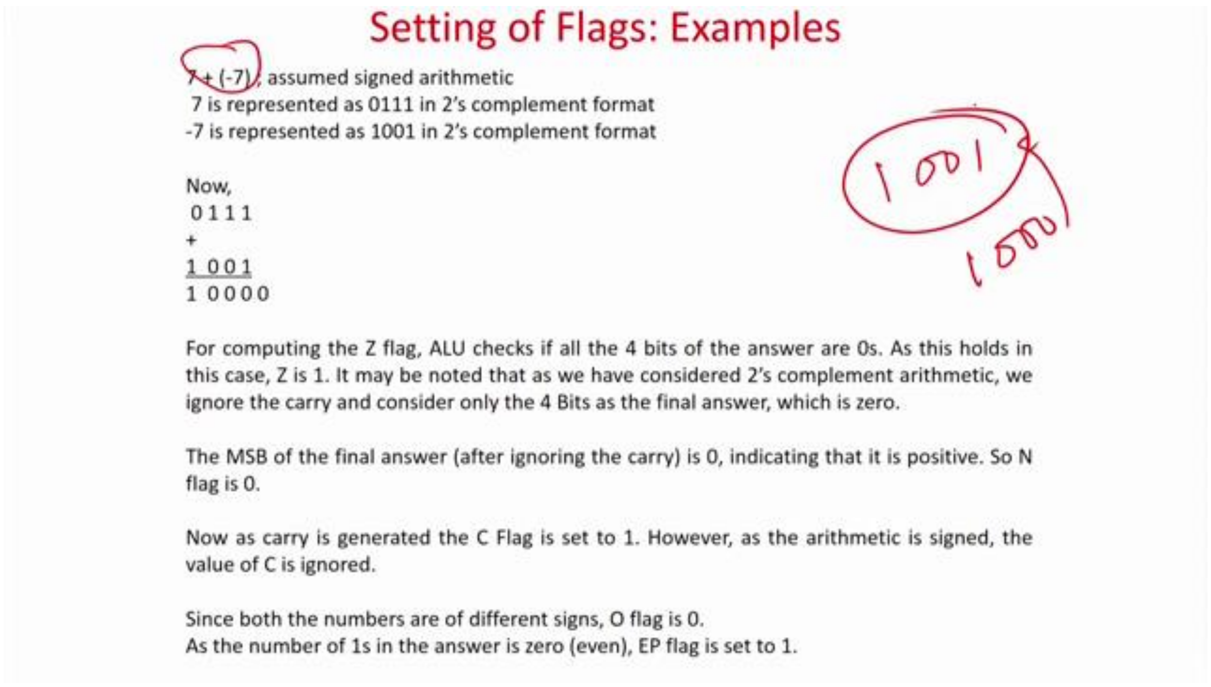
And then you can say jump not equal; that means, if the equality flag is not set jump not equal then again you jump back to label. And whenever this will be equal that $R2$ and n will be equal because n is = 10. So, n $and\ R2$ will have the value of 10 then jump not equal to label 1 will become false; because now they will be equal.

Because jump not equal J not equal; it is true if and only if the equality flag is reset not equal. Whenever the equality flag is true jump not equal to will become false because the equality flag will be set. So, there is something called equality flag; so, if the equality flag is equal to 1 sorry equality flag is 0; that means, the two stuffs are not equal, two operands are not equal then jump not equal will be true, but whenever two numbers will be equal then what is going to happen the equality flag will be set.

And then jump not equal to will become false whenever jump not equal to will be false; it will not jump to label, but we will go and execute the next instruction; so, it will come out of the loop. So, it gives a very nice example that how the other infinite loop of adding 1 +2 +3 has been modified to add 1 +2 +3 up to 10.

So, it gives a very nice idea that ah just before a comparison instruction we do a corresponding comparison ah sorry we do a comparison instruction set the corresponding flags and just by looking at the flag; we decide either to go to the top of the loop and re execute the loop or we come out of this one. So, that is actually the very concrete example of using a control instruction.

(Refer Slide Time: 31:27)



Setting of Flags: Examples

7 + (-7), assumed signed arithmetic
7 is represented as 0111 in 2's complement format
-7 is represented as 1001 in 2's complement format

Now,
```
 0 1 1 1
+
 1 0 0 1
1 0 0 0 0
```

For computing the Z flag, ALU checks if all the 4 bits of the answer are 0s. As this holds in this case, Z is 1. It may be noted that as we have considered 2's complement arithmetic, we ignore the carry and consider only the 4 Bits as the final answer, which is zero.

The MSB of the final answer (after ignoring the carry) is 0, indicating that it is positive. So N flag is 0.

Now as carry is generated the C Flag is set to 1. However, as the arithmetic is signed, the value of C is ignored.

Since both the numbers are of different signs, O flag is 0.
As the number of 1s in the answer is zero (even), EP flag is set to 1.

Now because ah whenever we will be looking at more ah different complicated codes in the next module; we will be always using so many times these ah control instructions, but in this unit we let us look at the more interesting part of it that is how the flags are set or which flags are set. If we can find out which flags are set or reset then after that it is very simple to think about the control instructions because there are just option true or false.

If it is true generally it will go to the next to the desired position of the label which is which is the conditional instruction is pointing to the label; it will go to that label. If the condition is true else we will just execute the next instruction after the jump instruction; extremely simple about it.

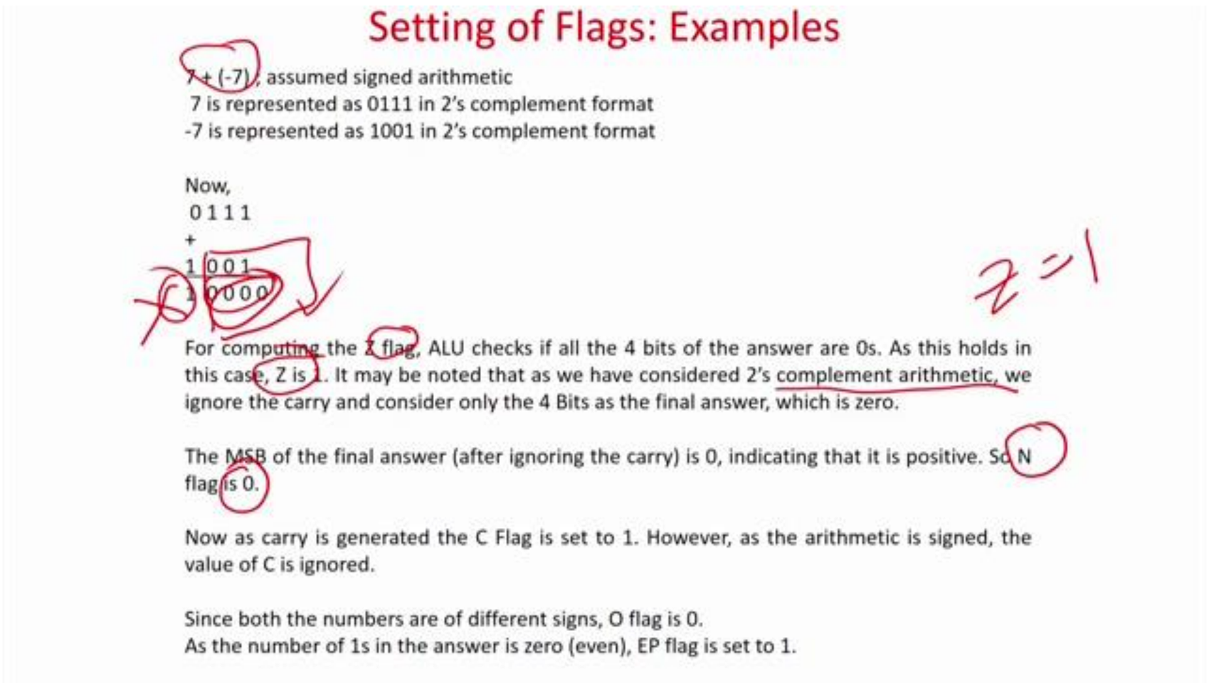
So, now we are seeing with different examples how different flags are set which flags are set, and which flags are reset that is more interesting. So, for example, they are doing +7 and -7; so, as both + and - are involved. So, it's a signed arithmetic. 7 is represented as 011 in 2's complement -7 is nothing, but in 2's complement it is 011 and again you have to add a 1.

So, this is actually 2's complement of -7 because a 1' complement of -7 is nothing, but 1000, you add a 1; you get this. So, this is actually the 2's complement of 7 that is -7. So, -7 and +7 are represented over here just I recall from the digital design.

So, the LSB is 0; it's a positive number; so, this is a positive number and this is a negative number; now let us add it. So, as it's a signed arithmetic. So, if you add it you are going to get

1, 1 + 1 is 0 carry will be 1. So, 1 + 1 again 0; the carry will be 1; so, 1 + 1 is 0 again 1 and this is 0. So, this is basically your answer.

(Refer Slide Time: 33:08)



Setting of Flags: Examples

7 + (-7), assumed signed arithmetic
7 is represented as 0111 in 2's complement format
-7 is represented as 1001 in 2's complement format

Now,
0 1 1 1
+
1 0 0 1
1 0 0 0 0

For computing the Z flag, ALU checks if all the 4 bits of the answer are 0s. As this holds in this case, Z is 1. It may be noted that as we have considered 2's complement arithmetic, we ignore the carry and consider only the 4 Bits as the final answer, which is zero.

The MSB of the final answer (after ignoring the carry) is 0, indicating that it is positive. So N flag is 0.

Now as carry is generated the C Flag is set to 1. However, as the arithmetic is signed, the value of C is ignored.

Since both the numbers are of different signs, O flag is 0.
As the number of 1s in the answer is zero (even), EP flag is set to 1.

And this is your some extra bit has been generated. So, 7 - 7 = 0; so, answer is 0 which is correct now you see what are the flags that are set and reset. In flag all flags are either set or reset, but some will be used and some will be discarded based on the context. So, for example, zero flag is set the 4 bit answer is 0000. So, it holds; so, this in this case Z = 0. So, the zeroth flag is set.

It may be noted that we have considered 2's arithmetic. So, we ignore be carry this is very important in 2's complement the arithmetic we generally ignore the carry ah, but anyway for calculating the zeroth flag which is not at all going into look at the carry business, for the zeroth flag checking this is only of matter of importance that is the answer 4 bits.

Whether a carry is generated if it's not generated whether you want to reject be carry because of 2's complement arithmetic it has nothing to do. It has got the four 0's as the answer; so, the zeroth flag is set. When you check all the 4 bits ah if the answer is 0 as this holds in this case the zeroth flag is 1. Then the MSB of the final answer after negating the carry because. So, in this case zero flag is equal to 1 that is the first thing because it has nothing to do with any other stuff.

You just take the 4 bits as the answer all 0 zeroth flag is set. Now the MSB is 0 because as I told you in 2’s complement arithmetic we can neglect the carry 0; so, indicating that is a positive answer. So, the ah; so, the answer is positive, so if there is a positive flag the answer will be 1; in this case N say it is a negative flag N.

So, the negative flag is 0 because the answer is a positive flag in the case it is a positive. So, negative flag is 0 so there is a negative flag call N. So, it will be reset because the answer is a positive answer. So, had it been; so, we will see if the answer is a negative answer then what will be value of the negative flag N flag; in this case as carry is generated. So, if you see; so, zeroth flag is 1 negative flag is 0.

Negative flag is reset because this MSB is equal to 0, 0 in 2’s complement arithmetic at the MSB will denotes a positive number. Now look at the carry; so, a carry is generated. So, as a carry is generated the C flag of the carry flag is set to be 1, but again as the arithmetic is signed the value of carry is ignored.

So, that is again very important; so, in a 2’s complement arithmetic what happens? We have done this and if you look at the de facto standard in digital design in 2’s complement arithmetic, we are not actually bothering about the carry which is generated that is a don’t care condition. But in a hardware when the flags are set or reset it does not look at all those contexts.

This is zero the zeroth flag will just check what are the answer of the 4 bits as it is 0 the 0th flag is set this bit is 0; that means, the positive number. So, the positive flag will be set or the negative flag is reset, a carry has been generated so in this case the carry flag is set. But as the arithmetic or the instructions we have executed we know these are 2’s complement numbers I have given as input; so, I will not use the carry flag.

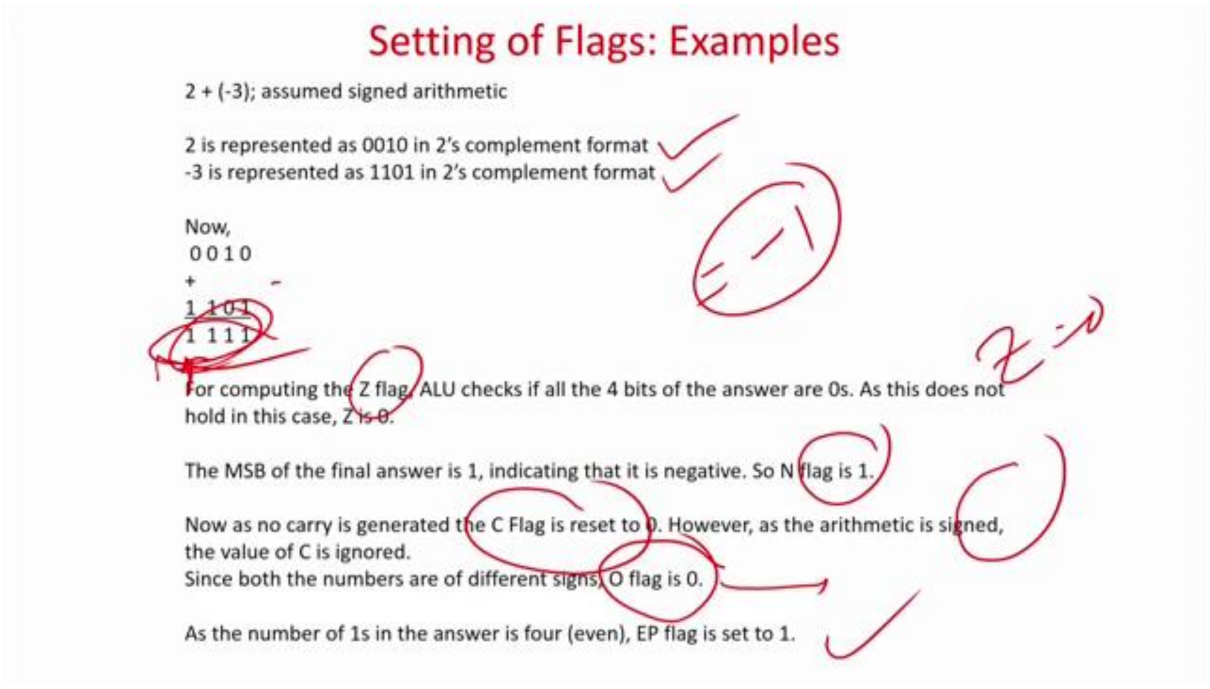
So, in other words the flag setting logic is totally blind; it is taking two numbers 0111 and 1001 and generating the answer as 1 as a carry and four 0s as the answer. And accordingly the flag bits of 0 is set, the flag bit of negative number is reset and a carry flag is set. But as I know as a programmer that I have given the two numbers which are input as 2’s complement arithmetic and in this case the carry is not calculated or ignored. So, I have to myself ignore the carry flag even if we set I should not use it as a. If I use the value of carry bit immediately after this instruction to do some conditional check and jump, there may be a logical error in my code.

So, as a programmer I have to know that I have to ignore the carry flag for this instruction. Since the both the numbers of; so, anyway ah say what other flags we can think since both the numbers are of different sign, the output flag is zero anyway we will see that as the number of 1s in the answer is 0; so, even parity flag is set to one and so forth. So, anyway all this ah flag bits can be easily understood ok.

So, again I will come to that; so, important are basically these 3 that the zero flag is set, the negative flag is reset, a carry flag is set, but it has to be ignored similarly the 4 bits the answer is 0. So, the zeroth flag will be set and there are 4 0’s; so, the number of parity is even.

So, the even parity will be set. The more the as the numbers are of different signs, some sign flag will be set to 0 and so forth. So, lot of flags will be there and based on the values the flags will be set or reset, but which flag has to be ignored has to be decided by us. Again I will take another example; so, to make the things easier like for example, I have taken 2 and I have taken -3.

(Refer Slide Time: 38:10)


Setting of Flags: Examples

2 + (-3); assumed signed arithmetic

2 is represented as 0010 in 2’s complement format

-3 is represented as 1101 in 2’s complement format

Now,

```
 0 0 1 0
+
 1 1 0 1
 1 1 1 1
```

For computing the Z flag, ALU checks if all the 4 bits of the answer are 0s. As this does not hold in this case, Z is 0.

The MSB of the final answer is 1, indicating that it is negative. So N flag is 1.

Now as no carry is generated the C Flag is reset to 0. However, as the arithmetic is signed, the value of C is ignored.

Since both the numbers are of different signs, O flag is 0.

As the number of 1s in the answer is four (even), EP flag is set to 1.

That is 2 -3 I am going to do. So, this is a 2’s complement implementation of -3 and 2 if I do the answer I am going to get this as the answer. So it is basically the answer should be equal to nothing, but -1; so, in this case what happens? The four 1s are there. So, it is checking that the last bit is 1.

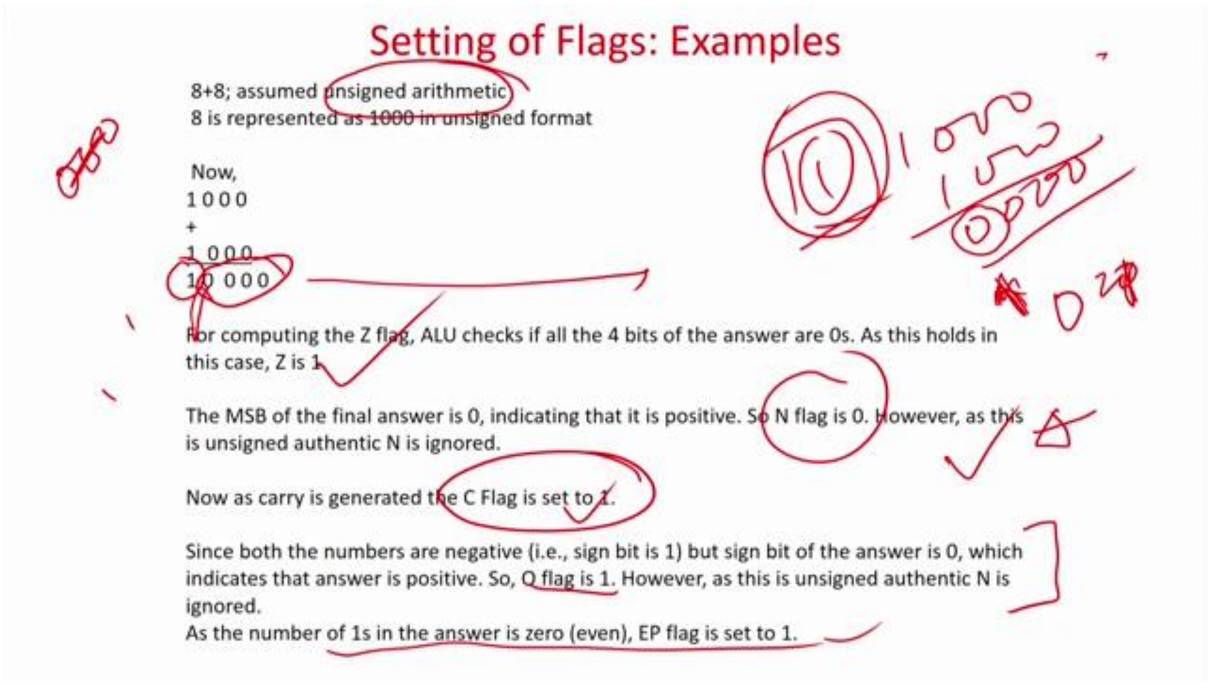
So, the zeroth flag so anyway let us first illustrate with zeroth flag; so, the all the answers are 1; so in fact,; obviously, the answer is not 0. So, the zeroth flag is set to 0 that is obvious. The MSB is 1; so, it's a negative number; so, the negative flag is set to 1 obviously it is a negative number because there so, 2 -3 is -1; so, negative flag is set there is no carry will be generated.

So, the carry flag is reset to 0 but again as a programmer; you have to always do not consider the carry flag as of now because even if the carry flag is reset because there is no carry generated, but in 2's complement the arithmetic carry flags are not used. So, that you have to overflow since both the numbers of different size the overall flag is 0.

So, why what I was telling about this overflow flag now and this means the idea is that if the two numbers; one is positive and one is negative, a overflow can never happen. So, basically in such cases always the overflow flag is reset. So, whenever I means whenever I take some new examples now when both the numbers will be positive or both the numbers will be negative; the overflow flag will be talked about.

So, in both the cases the overflow flag is set to 0; the answer is 4 1's. So, if the 4 answers are 1 then what is the case? It's an even parity; so, the even parity flag is set to 1.

(Refer Slide Time: 39:44)



Setting of Flags: Examples

8+8; assumed unsigned arithmetic
8 is represented ~~as 1000 in unsigned~~ format

Now,
1 0 0 0
+
1 0 0 0
1 0 0 0 0

For computing the Z flag, ALU checks if all the 4 bits of the answer are 0s. As this holds in this case, Z is 1

The MSB of the final answer is 0, indicating that it is positive. So N flag is 0. However, as this is unsigned authentic N is ignored.

Now as carry is generated the C Flag is set to 1.

Since both the numbers are negative (i.e., sign bit is 1) but sign bit of the answer is 0, which indicates that answer is positive. So, O flag is 1. However, as this is unsigned authentic N is ignored.
As the number of 1s in the answer is zero (even), EP flag is set to 1.

Now, as I was telling you that all the flags we have considered, but every time the output or the sorry the overflow flag O is the overflow flag. The overflow flag we are actually not considering right now because one number is positive and one number is negative.

If both the numbers are differences sign the overflow flag is neglected. Now we are taking two numbers as 8 and we are using an unsigned arithmetic. So, now all the other flags importance will start coming up because two numbers are splitting and if you are adding you may get the overflow, you may get the carry because you are going an unsigned arithmetic in unsigned arithmetic carry etcetera are of importance.

So, I add two numbers 8 +8; so, you are going to get the answer as 10000 that is 16, but now you see these are the 4 bits which is of importance this one bit carry or overflow has been generated. So, now, this sorry; so, as the hardware it will just check the answers are 4 0. So, that is true then the zero flag is set.

The MSB is zero; so, it is not going to check the carry one. So, the answer is 1000; 1000 so the answer is 0000 the carry is generated as 1. So, if you check the MSB is 0; it is not going to look at the overflow; so, the negative flag is reset again ah in this case also you can ah ignore the ah negative or positive flag here; using this case is an unsigned arithmetic.

Previous version we are using a signed arithmetic in 2’s complement; this was very very important; then again the flag was very important. In that case the zeroth MSB means a positive number and 1 in the MSB as a negative number. So, in last context you had to take into mind that I have to consider this flag. So, if I take 100 and 100 sorry 1000 and 1000 in 2’s complement arithmetic, these are two negative numbers basically. So, if you are as a programmer you know that I have given the two numbers as inputs which are in 2’s complement in that case you have to very deliberately keep in mind about the sign flag.

But as the present example we are taking a unsigned arithmetic. So, we have to neglect the negative flag carry has been generated definitely. So, the carry flag is set to 1 again previous case we have to neglect the carry flag; that means, because we are using ah 2’s complement arithmetic and one number was negative and one number was positive. So, we are neglecting the carry flag, but here as we are using an unsigned arithmetic a carry has been generated and in such unsigned arithmetic the carry flag is set to 1, a carry has been generated and you have to consider this.