

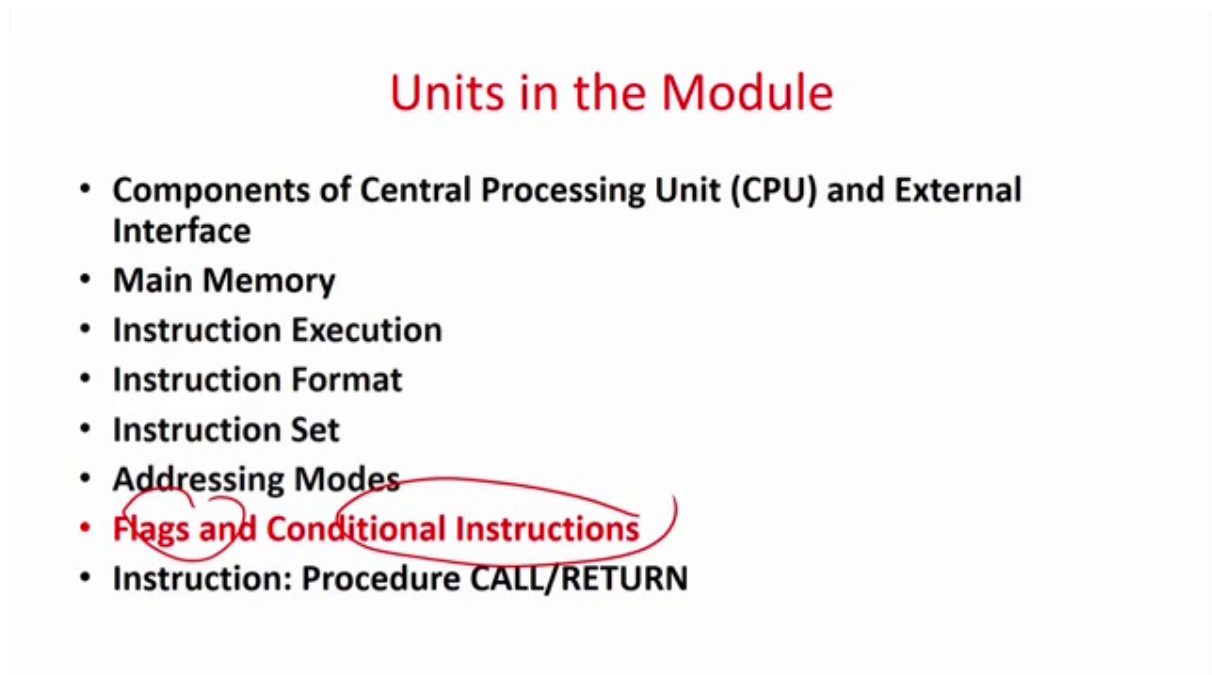
**Computer Organization and Architecture: A Pedagogical Aspect**  
**Prof. Jatindra Kr. Deka**  
**Dr. Santosh Biswas**  
**Dr. Arnab Sarkar**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Guwahati**

**Lecture – 13**  
**Flags and Conditional Instructions**

[noise]

Ok. So, ah welcome to the next unit on the module on addressing [noise] mode instruction set and [vocalized-noise] instruction execution flow [vocalized-noise].

(Refer Slide Time: 00:30)



**Units in the Module**

- **Components of Central Processing Unit (CPU) and External Interface**
- **Main Memory**
- **Instruction Execution**
- **Instruction Format**
- **Instruction Set**
- **Addressing Modes**
- **Flags and Conditional Instructions**
- **Instruction: Procedure CALL/RETURN**

So, ah till now whatever we were discussing [vocalized-noise] basically what [noise] are the different in type [noise] of instructions? How it looks? What are the formats [vocalized-noise]? What are the different components of an instruction? [vocalized-noise] And in the all the cases if you observe we were just assuming or thinking [noise] that [vocalized-noise] the instructions would execute a sequential manner [vocalized-noise].

That is first instruction may load something from the memory [noise] then it may do some memory operation [noise] addition operation subtraction operation [vocalized-noise] and then will again write to the memory [vocalized-noise]. So in fact, [noise] we are assuming that everything would go in a very sequential flow [vocalized-noise], but [vocalized-noise] as all of us have written some c code [vocalized-noise] or any any high level language code in our line [noise]. So, we are [vocalized-noise] it is really obvious to us [vocalized-noise] that [vocalized-noise] there is nothing which is very sequential in a code [noise].

That means, [vocalized-noise] every time you will have some logic [noise] and it will depend on some input conditions [noise] and based on that [vocalized-noise] you will either jump to another instruction or even continue; like a loop [vocalized-noise] based on some instruction you will go back to the loop and based on the [noise] exit condition you will exit out of the loop [vocalized-noise].

In other words a very very important concept which we need to understand; when you are looking at the instruction format and execution of instructions [vocalized-noise] is that [vocalized-noise]; we are always have some conditions [vocalized-noise] and based on the condition some of the instructions [vocalized-noise] will execute the next instructions or jump to some other instruction, [vocalized-noise] which will be [noise] executed [vocalized-noise] once the conditions are satisfied [vocalized-noise]. In fact, they are called as this [noise] conditional [noise] instruction [vocalized-noise] and without a conditional instruction no coding paradigm is complete [vocalized-noise].

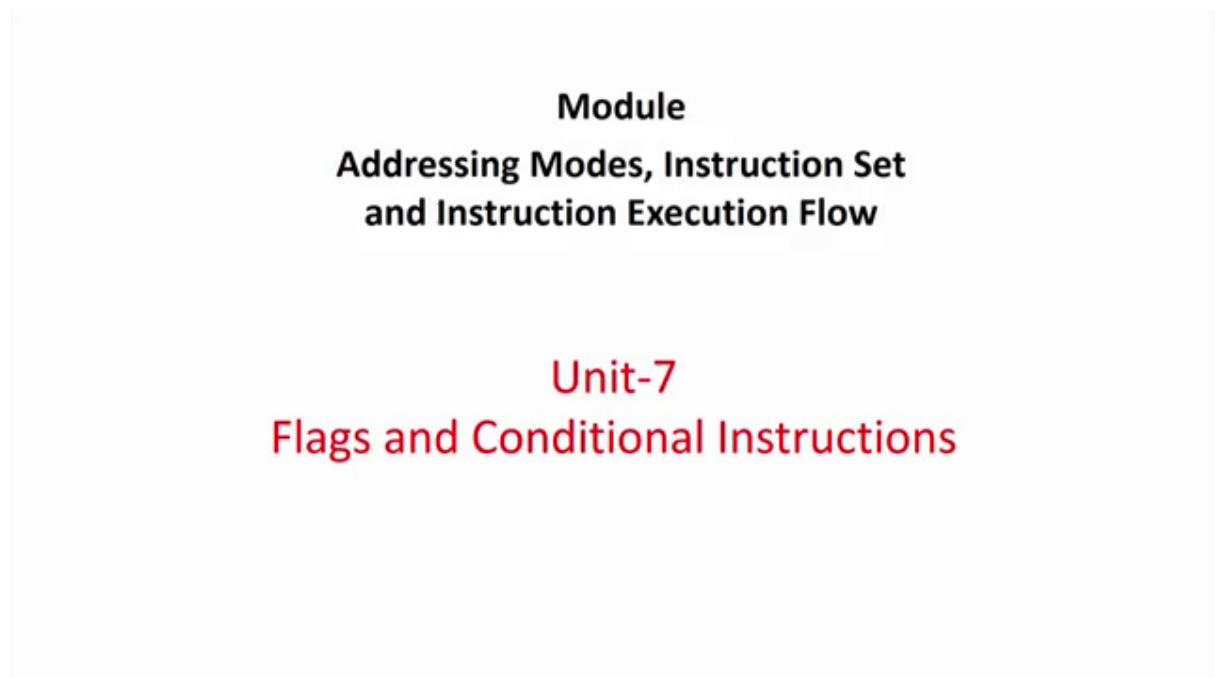
So, along with conditional instruction [noise] in this unit we are going to look at flags and conditional instructions [vocalized-noise]. So, conditional instructions are very [vocalized-noise] similar to our [noise] like or if then else statement while conditions, jump, loops, etcetera [vocalized-noise], but [vocalized-noise] whenever we say something like [vocalized-noise] ah [noise] if x is greater than y then do something [vocalized-noise].

So, a high level language we have a condition like x is greater than y [vocalized-noise] and [noise] how a condition is internally checked [noise] in a [vocalized-noise] hardware or in your [vocalized-noise] CPU [vocalized-noise] is basically depending on certain kind of flag registers [vocalized-noise] or the registers are there and there are some bits

[vocalized-noise] which are actually set [noise] or reset depending on some conditions [vocalized-noise]. And your conditional instruction; actually checks those flag [vocalized-noise] and then decide what to do [vocalized-noise]. So, this unit will be dedicated to [vocalized-noise] such conditional instructions [vocalized-noise] and what are the flags and how they coordinate.

And how they are executed [noise]. So, this is about the [vocalized-noise] in the number 7 of this module [vocalized-noise] on flags and conditional instructions [vocalized-noise]. So, as we are [vocalized-noise] doing at hardware [vocalized-noise] the term flag is actually coming with the conditional instructions.

(Refer Slide Time: 02:56)



But or a high level language version [vocalized-noise] we generally talk about conditional instructions [noise] if there else [vocalized-noise] for loops etcetera [noise].

(Refer Slide Time: 03:04)

## Unit Summary

The most common way of generating the condition to be tested in a conditional branch instruction is the use of flags. The flag code register consists of individual bits that are set or cleared depending on the result of an operation that is carried out by execution of an instruction. These bits are used to remember the effect of computation after executing the current instruction. These bits are then used as conditions for next conditional instructions.

Sign, Zero, Carry, Auxiliary carry, even parity, overflow, equal etc. are some of the most common flags.

So, [vocalized-noise] as the whole course is ah based on pedagogical acts aspect [vocalized-noise]; so, what is the unit summary? [noise] [vocalized-noise]. So, or what you are going to look in this unit [vocalized-noise]. So, basically this unit will be mainly focusing on [noise] this that [noise] there are certain instructions [vocalized-noise] whose executions are not sequential [vocalized-noise], they actually depend on conditions [vocalized-noise]. So, actually they as you all know there is something called program counter [vocalized-noise] which will help us to know what is the next instruction [vocalized-noise].

So, if there is no conditional instruction as such [vocalized-noise] then the program counter increments by 1 [vocalized-noise], but whenever there is a conditional instruction [vocalized-noise] based on the truth of the condition [vocalized-noise]; the program counter value changes and it jumps to the required instruction [noise] [vocalized-noise].

So, there are two type of conditional instruction that is conditional branch and unconditional branch [vocalized-noise]. Conditional branching means from statement x you go to statement y [noise] or you just execute statement x plus 1 [vocalized-noise] next instruction [vocalized-noise] depend on [vocalized-noise] some condition like [vocalized-noise] x is greater than y, then you execute the next instruction and if it is

false [noise] you jump to some other memory location and execute the instruction there [vocalized-noise].

But there are some unconditional jumps also [vocalized-noise] ah unconditional statement that you come here [vocalized-noise] and then jump [noise] to such and such memory location execute the instruction [noise] there without waiting for any condition [vocalized-noise] or without validating any condition; that is jump unconditional [vocalized-noise] that is just like a function call [vocalized-noise]. So, you come over here [vocalized-noise] without any condition you just execute the instruction [noise] which is corresponding to the function [noise] which is being called.

So, they are actually unconditional [vocalized-noise] jump instructions [vocalized-noise]. So, in this unit basically we are going to look at [vocalized-noise] ah what are the conditional [vocalized-noise] instructions? [noise] How they actually change the program counter? [noise] [vocalized-noise] When they are stored? What are the internal dynamics of it [vocalized-noise]? And basically there are two types of conditional treatment branch an [vocalized-noise] unconditional branch and conditional branch [noise] [vocalized-noise]. And then we will look at the basic heart or basically what actually ah determines this condition [vocalized-noise].

In high level language we say  $x$  is greater than  $y$  [vocalized-noise] or the fall [vocalized-noise] I mean say [noise] loop continues from 0 to 10 [vocalized-noise] and every point we increment [noise] the counter by 1 [vocalized-noise], but when the counter which is 10 [vocalized-noise] then we exit [vocalized-noise]. If hardware how actually it is reflected [vocalized-noise]? So, it is reflected in terms of certain [noise] flag bits [vocalized-noise] which are [vocalized-noise] some registers call the flag registers and inside the flag register [vocalized-noise] there are certain bits allocated for [noise] some important parameters like sign, zero, carry, parity, [noise] overflow, equality etcetera [vocalized-noise].

So, what they are basically [vocalized-noise]? When some conditional [vocalized-noise] instructions are executed like say add [noise] [vocalized-noise] and then certain flags will be set [noise] in the flag register depending on the value of the operation [vocalized-noise] say for example, if I subtract two numbers [vocalized-noise] and the answer is 0 [vocalized-noise].

So, in that case the flag bit [vocalized-noise] corresponding to 0 will be set [vocalized-noise] if I add two numbers and the sum is say 5 [vocalized-noise]. So, in this case the zeroth flag [vocalized-noise] which will be say reset because [noise] the answer is more than 1 [vocalized-noise]. Then there is something called [vocalized-noise] even parity; so, if the answer is 5 1 0 1 is an odd parity number [vocalized-noise]. So, this even parity bit will be reset and so, forth.

That means, in other words [vocalized-noise] in this unit we are also going to look at certain flag [noise] bits [vocalized-noise]. And how they are set and how they are reset depending on the arithmetic operation [vocalized-noise] just before a [vocalized-noise] just [vocalized-noise] ah after [noise] going for this instructions corresponding to addition subtraction equality checking [noise] etcetera [vocalized-noise]. And then we will see [vocalized-noise] how the charm conditional jump instructions basically execute [vocalized-noise] by looking at the [noise] value of this bits in the flag register [noise]. So, what are the objective of the [vocalized-noise] unit?

(Refer Slide Time: 06:25)

## Unit Objectives

- **Comprehension: Discuss:--**Discuss about flag bits and how these flag bits get set or reset.
- **Synthesis: Design:--**Uses of flag bits to design conditional statements.

This is a small unit [vocalized-noise] and the objectives are basically to discuss [noise] that [vocalized-noise] after this object code after this unit [vocalized-noise] as a comprehension you will be able to discuss [vocalized-noise] flag bits [noise] and how this flag bits are set and reset [noise] [vocalized-noise]. This flag bits are heart of [noise]

any kind of a conditional instruction [vocalized-noise] and you will be able to discuss [noise] [vocalized-noise] in comprehension [vocalized-noise] that what are the bits how are this bit set [noise] and how are the bits reset [noise]? And then you will be able to design the synthesis objective using this flag bits you will be able to design conditional statements.

That is [vocalized-noise] ah [noise] are based on the because [noise] no conditional instructions can be executed without the flag bits [vocalized-noise]. So, using these flag bits; so, what are the conditional instructions can be designed [vocalized-noise]? Like for example, if you have a zeroth flag [vocalized-noise] then you [vocalized-noise] you can have instructions corresponding the equality checking [vocalized-noise] that [noise] if two numbers are compared [vocalized-noise] or if I subtract two numbers [vocalized-noise] and then if the [noise] answer is 0; then this zeroth flag will be set [noise] then [vocalized-noise] I can have a instruction [vocalized-noise] called I can have a subtraction instruction just before it [vocalized-noise] and then I can say jump on 0 [vocalized-noise].

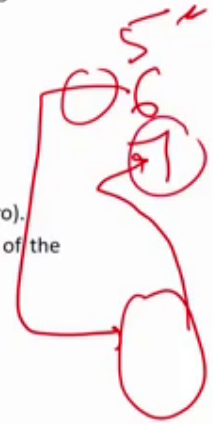
That means, say for example, I have a counter [vocalized-noise] and I am [vocalized-noise] I actually want to increment the counter till 10 [vocalized-noise] and [vocalized-noise] just after [vocalized-noise] executing of the each loop, [vocalized-noise] I will decrement [noise] or I will increment the value of the index by 1 [vocalized-noise]. And as [vocalized-noise] as I check as I check [vocalized-noise] I will subtract the [noise] index ah with 10 [vocalized-noise]. So, whenever the answer is 0; that means, the loop [vocalized-noise] has reached [noise] up to 10 [noise] and it should exit [vocalized-noise].

So, in that case I will use the jump [noise] conditional jump on wet [noise] [vocalized-noise], but just before that I will subtract [noise] the index [vocalized-noise] variable with 10 [vocalized-noise] and if it is zero the zeroth flag will be set and I can have a special instruction [noise] called jump on 0 [noise] [vocalized-noise]. So, this is actually a synthesis objective just by looking at the flag bits, you can decide what are the instructions can be designed for this conditional course [noise].

(Refer Slide Time: 08:09)

## Program status word

- The program status word (PSW) is a part of memory or registers which contain information about the present state of a program. By storing the current PSW during an interruption, the status of the CPU can be preserved for subsequent executing after returning from the Interrupt Service Routine.
- Error Status of the programs
- Pointer to the next instruction to be executed
- Sign bit of the result of the last arithmetic operation.
- Zero bit, which is set (reset) when the result of an arithmetic operation is 0 (no zero).
- Carry bit, which is set (reset) if an operation resulted in a carry (no carry) out of the MSBs of the operands.



So, before we start off this one [noise]. So, we know that whenever you talk of a [vocalized-noise] jump instruction then what basically happens [vocalized-noise] you are executing certain set of code [noise] or you are in a certain [vocalized-noise] temporal part of a code [vocalized-noise]. And as a jump instruction you generally you can go and serve a procedure or a function [vocalized-noise]. So, before we jump [vocalized-noise] from the [vocalized-noise] main program to some other [noise] function or.

From one location to a some other location [vocalized-noise] the current context of the code has to be saved [vocalized-noise]. So, [vocalized-noise] therefore, that is certain what are the temporary [noise] memory location, what are different register values at this time [noise], what was the accumulator value at this time [vocalized-noise]. Say for example, you have loaded some variable you have added with something [noise] as store the value in the accumulator and [vocalized-noise] just after that [vocalized-noise] before saving it to the main memory [vocalized-noise], you got a procedural code and you job [vocalized-noise].

So, whenever I come back and [vocalized-noise] execute from this one [noise]. So, what basically happens [vocalized-noise]? The program [noise] counter say [noise] is that memory location 5 [vocalized-noise] in each case what I have done [noise] I have added something with accumulator [noise] and stored the value accumulator [vocalized-noise]



now the sixth location was storing the accumulator to the main memory [vocalized-noise].

But the sixth memory locations what happened [vocalized-noise]? Just before [noise] that [vocalized-noise] ah [vocalized-noise] sorry the [noise] location may be a [vocalized-noise] this accumulated operation, then there is a function [noise] which may be a condition and operation that depending on something [vocalized-noise] you execute the next instruction [vocalized-noise] or jump to a [noise] function [vocalized-noise].

As a next instruction may be to store the value of the accumulator to the memory [vocalized-noise] just, but just before that there was a conditional instruction [vocalized-noise] based on something something you jump [vocalized-noise]. So, just after the accumulator operation at memory location fifth [noise]; you add [vocalized-noise] add something else to be accumulator, but next was a conditional instruction [vocalized-noise]. So, without saving it to the memory you have a executed a procedure [noise] call.

But then what happens the accumulator will also be used in the procedure code; so, the intermediate value [noise] of this sum which in the accumulator will be lost [vocalized-noise]. So, what do you have to do? [vocalized-noise] Before you go to the executing the function [noise] you have to store the value of the accumulator [vocalized-noise], you have to remember the value of program counter [vocalized-noise] that now I am in 5 [noise] sixth was the [vocalized-noise] I mean [vocalized-noise] conditional instruction [vocalized-noise].

And after I come back [noise] I have to start executing some ah [vocalized-noise] means memory location number 7 [vocalized-noise] the instruction that will [noise] save the [vocalized-noise] accumulator value to the same memory [noise] [vocalized-noise]. So, you have to also remember [noise] that what position that is if I jump from this location [noise] to this location [noise]. So, I have fifth the accumulator operation [noise] sixth is your conditional jump seventh is again.

You are storing back the value of accumulator [noise] whatever you have done over here in the main memory [vocalized-noise]. So, after doing this function you have to begin come back [noise] an execute [noise] for memory location 7 [vocalized-noise]. So, when I am jumping you have to remember that I have to come back to memory location number 7 [vocalized-noise] whatever you need inter need value has to be stored in the

memory register memory. And so, many temporary variable or context of the program has to be saved [noise] [vocalized-noise]. So, that when I come [vocalized-noise] after executing the function [vocalized-noise] I can recollect everything back [noise] and reload the registers [noise] accordingly [vocalized-noise]. So, there is something called a program status word [noise] which the part of the memory [vocalized-noise] or registers which contained information [vocalized-noise] about the present state of the program [vocalized-noise] by storing [vocalized-noise] current PSW [vocalized-noise] during interaction [noise].

Or procedure call [vocalized-noise] then everything will safe and then you come back after the executing your function [vocalized-noise] you can ah get back the whole [noise] code and all the intermediate values we feel and start executing from we have left [vocalized-noise]. So, what are the [vocalized-noise] why you are. So, [noise] PSW [vocalized-noise] PSW is.

So, important [noise] [vocalized-noise] when you are studying about jump instructions [noise] [vocalized-noise] because jump means you may leave the context of the current code [vocalized-noise] and execute some other context [noise] and again come back and re execute from where I have there [vocalized-noise]. So, the whole [noise] intermediate state of your program [vocalized-noise] is saved as a PSW [vocalized-noise] and it will collect that [vocalized-noise].

So, what the PSW has? It has lot of components [noise] some of them I have listed error status of code point up the next instruction to be executed like in this case it is 7 [noise] where I have left [noise] sign bits [noise] 0 bits [noise] carry bits [noise] reset beeps overflow bits [vocalized-noise] and so, many other things which is listed over here [vocalized-noise].

(Refer Slide Time: 11:58)

## Program status word

- Equal bit, which set (reset) if logic comparison result of the corresponding instruction is true.
- Overflow bit, which is used to indicate arithmetic overflow.
- Interrupt enable/disable, used to enable or disable interrupts.
- Supervisor bit to indicate whether the processor is executing in the supervisor or user mode.



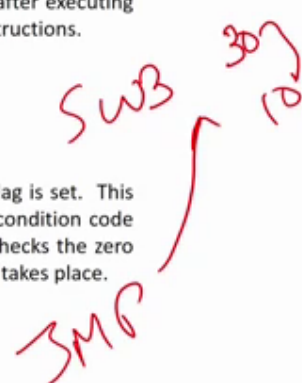
So in fact, [noise]. So, when we destroy [noise] books free [noise] advancing in different [noise] [vocalized-noise] other [vocalized-noise] newer modules [noise] on interrupts [noise], but then ah actually on memory operations then we will be looking at more in details about what exactly [noise] the PSW stores [vocalized-noise], but for at the time being [noise] you can see what they are [noise] there are status [noise] pointer pc is next value all the flag bits [noise] current values of accumulator if there are some [noise] [vocalized-noise] if [vocalized-noise] they are some [noise] registers r 1, r 2, r 3 [vocalized-noise] which are the user defined registers all those variables you will store [noise] it as a program status words [vocalized-noise].

So, whenever I come back [vocalized-noise] I can recollect [noise] everything and I can [noise] reuse [vocalized-noise]. So, that is one very important thing that just before executing a jump to a [noise] function [vocalized-noise] or an interrupt service routine which store the program status word [vocalized-noise].

(Refer Slide Time: 12:34)

### Condition codes or flag bits

- The condition /flag code register consists of individual bits that are set or cleared depending on the result of an operation that is carried out by execution of an instruction. These bits are used to remember the effect of computation after executing the current instruction. Also, these bits are used as conditions for next instructions.
- For example if a subtraction instruction results in a zero, then the Zero flag is set. This bit remains set until another instruction that affects the zero bit in the condition code register executes. Now, if the next instruction is "Jump if Zero" then it checks the zero flag. As the zero flag is set the condition of the instruction holds and jump takes place.



Now that is about the background that before I ah I am doing some work now [vocalized-noise] I have just go and do something else [vocalized-noise]. So, before that actually I save my work in some intermediate memory [vocalized-noise] and when I come back I can do that [vocalized-noise].

Now we are going to the real clocks of these ah [vocalized-noise] I means conditional instructions that is flags [noise] [vocalized-noise]. So, this was about bookkeeping [vocalized-noise]; so, once you have the more details about bookkeeping etcetera will be doing when you are doing the [noise] module or interrupts [noise] [vocalized-noise] that is all I go [vocalized-noise] then ah [noise] because the [noise] whenever there is an interrupt you have to leave the main good well serviced interrupt [noise]. So, you have to store the context of the main [vocalized-noise] main [vocalized-noise] main code [vocalized-noise] ok.

So, leaving aside that [noise] that is just a quantity of the keep in mind for the time being [vocalized-noise], then we are going to something called the heart of [noise] [vocalized-noise] ah this job instruction that this is the code or flags [vocalized-noise]. The code or flag is a register basically [noise] it is something called a flag register [vocalized-noise] you have individual bits [noise] which are set and reset [noise] by the some execution of a enigmatic [noise] operational logic operation just [noise] before the setting of it [noise].

In other words there is a register is a flag register [noise] [vocalized-noise]. So, whenever [noise] some ah the arithmetic operation or logic operations happens [vocalized-noise]; the corresponding bits are set or reset [vocalized-noise]. For example, [vocalized-noise] if a subtraction instruction reads to zero [vocalized-noise] then the zeroth flag is set [vocalized-noise]; that means, there are a lot of flag like zeroth flag, parity flag, [noise] sign flag, overflow flag etcetera [vocalized-noise].

So, [noise] this also some arithmetic operation [noise] of a subtraction addition [vocalized-noise]; the corresponding bits are certainly set [vocalized-noise] and for many cases [vocalized-noise] basically some of the flag bits are set or reset, but they are [noise] not taken into picture [vocalized-noise] for example, if I am doing a subtraction operation [noise] a minus b [vocalized-noise] so; obviously, there will be no carry generated [noise] because unless I add two negative numbers or ah add two positive numbers [noise] [vocalized-noise] the carry [noise] will not be generated.

But there is no overflow [vocalized-noise] a carry can be generated a as we will see in 2's complement [noise] arithmetic [vocalized-noise], but a overflow is not generated [vocalized-noise]. So, even if the [noise] overflow flag is set or reset if I am just adding two [noise] ah one if I am doing a subtraction operation [noise] that is I subtract a negative number for positive [noise] number [vocalized-noise] or one number is positive and one number is add negative we are adding it [noise] that is [vocalized-noise] two sign numbers are there.

But of the opposite sign [noise] and we are adding it [vocalized-noise] then the overflow flag is immaterial the flag will be set or reset [noise] there is some different conditions [noise] as we will see [vocalized-noise], but it will be a do not care condition [noise] [vocalized-noise], but a very concrete example [noise] is given over there if I subtract two numbers [vocalized-noise] and the answer is zero [noise] then the zero flag is set [vocalized-noise] and then you can use it [noise] for an instruction like jump if zero there is jump on zero.

So, [noise] if there is an instruction [noise]; so, if I have a operation like say SUB [noise] SUB sub some say [noise] ah 30 [noise]. So, ah [noise] [vocalized-noise] what is we it is taking the value of memory location 30 [vocalized-noise] whatever is [vocalized-noise] the assuming [noise] it to be a direct instruction [vocalized-noise]. So, ah [noise] it will

go to the location 30; find out what is the value in the memory location 30, subtract with the accumulator and store back [vocalized-noise]; you see if we [noise] content up memory location 30 is equal to the accumulator you will get a zero over here [vocalized-noise].


Immediately zero flag will be set [vocalized-noise] now with the next instruction you can have a conditional instruction [noise] you can see that [noise] jump if zero [vocalized-noise]. So; that means, if ah [vocalized-noise] the means that the memory location have the value of 10 30 [noise] memory location 30 has the value 10 [vocalized-noise] and accumulator is an index of a counter [vocalized-noise] which is also got 10 [noise] you subtracted [noise] then the loop has been completed [noise] and I have to jump [vocalized-noise] out of the boot.

So, you can say have a instruction called jump on zero [vocalized-noise] that is up to this abstraction instruction [vocalized-noise]. So, it will jump [noise] and go out to some other memory location because the instruction has been [noise] satisfied conditional instruction will be satisfied because the zeroth flag will set [noise] [vocalized-noise].

(Refer Slide Time: 15:58)

**Condition codes or flag bits**

FLAGS	RULES TO SET/RESET	
	SET	RESET
<b>S (Sign)</b> This flag is of importance if the arithmetic is signed	If the result of an operation is positive or 0.	If the result of an operation is negative
<b>Z (Zero)</b>	If the result of an operation is 0.	If the result of an operation is not equal to 0. In all other cases.
<b>C (Carry)</b> This flag is of importance if the arithmetic is unsigned	1) If the addition of two numbers results in carry out of the most significant bits. 1) If a subtraction of two numbers requires borrow into the most significant bits that are subtracted.	In all other cases.
<b>EP (Even Parity)</b>	If the result of operation has even number of 1's.	In all other cases.



So, now, we are going to see [vocalized-noise] different type of [noise] ah what are the different type of ah [noise] typical [noise] in a typical CPU, what are the different types

of flags? [vocalized-noise] First is the sign flag [vocalized-noise]; so, this is flag is of importance the arithmetic [noise] is sign [vocalized-noise]; that means, if I am using unsigned arithmetic for time being [noise]. So, these flag is of no importance [vocalized-noise] if the operation that is if I am [noise] is in the 2's complement [noise] arithmetic. So, if I know the MSB is 1; it is a negative number [vocalized-noise] and if the MSB the 0 it is a positive number [vocalized-noise].

So, ah the sign basis of importance if the arithmetic is sign [vocalized-noise]; so, if the [noise] answer is 0 that is positive [noise] if the MSB is 0 [noise]. So, it is set [vocalized-noise] that is a positive number [vocalized-noise] and if the [noise] MSB is a 1 then it is a negative number and it is reset [noise] 0 [noise]. So, if you do some operation and the answer is 0; so, it is set [noise] and it is if it is the answer is not [noise] equal to 0 user is set [noise].

So, of course, for any operation everything [noise] zeroth flag is very very important [vocalized-noise]. Unlike if it is a unsigned arithmetic [noise] then the sign has no meaning [vocalized-noise]; carry flag [vocalized-noise] if two numbers if the addition of two numbers result in a carry out [noise] of the most significantly [noise] is obvious [vocalized-noise] if I am having two numbers [noise] say example [vocalized-noise] 0 1 1 1 and if I have a ah number 1 0 0; unsigned arithmetic I am considering [noise] [vocalized-noise] sorry if I take 1 1 sorry [noise].

If I 7 and this is 12 if I add [noise] 7 with 12 [noise] we are going to get 1 1 0 [noise] 0 [noise] and then the carry generated [vocalized-noise]. So, you you in case of a unsigned arithmetic [noise] such a carry is [vocalized-noise] generated the more [noise] significantly; so, the carry flag is set [vocalized-noise]. So, if I am taking two [vocalized-noise] if I subtract [vocalized-noise] two numbers [noise] and there is some [vocalized-noise] carry is borrowed [noise].

Then also ah sign [noise] carry flag will be set [vocalized-noise] and in all other cases it is reset [vocalized-noise]. So, is very simple words [noise] ah as I am giving you an example if some [noise] carry is generated by adding two numbers at the MSB [vocalized-noise] or if the subtraction of two numbers [vocalized-noise] required a borrow [noise] and this then ah carry flag is set [noise]. Similarly [noise] even parity [noise] the results are generated; so, if the number of once [noise] is even the parity is set

in the another cases [noise] and you say like in this case [noise] the carry is 1, but the 4 bit answer is [noise] 0 0 1 1. So, it a number of 1s or 2; so, we say even, so even parity is set [noise].

(Refer Slide Time: 18:05)

**Condition codes or flag bits**

<b>O (Overflow)</b> This flag is of importance if the arithmetic is signed	1) If the sum of two <u>numbers positive</u> (with sign bit 0) yields a <u>negative</u> number (with sign bit 1.) 2) If the sum of two <u>negative numbers</u> (with sign bit 1) yields a positive number (with sign bit 0.)	In all other cases.
<b>(E) Equal</b>	If the result of a comparison instruction is true	If the result of a comparison instruction is false
<b>Interrupt enable</b>	If the flag is set to 1, mask-able hardware interrupts will be served.	If cleared (interrupt enable set to 0), such interrupts will be ignored. However this flag does not affect the handling of non-mask-able interrupts.
<b>Supervisor mode</b>	If this flag is set, it indicates that the processor is executing in the supervisor mode. Certain privileged instruction can be executed only in supervisor mode and certain area of memory can be accessed only in supervisor mode.	If this flag is reset, it indicates that the processor is executing in the user mode.

*Handwritten notes:*  
 Cmp  
 ①  
 1000  
 1001  
 ---  
 1000  
 1000  
 ---  
 1000

Several others [noise] overflow flag [vocalized-noise]; so, they tell you what is an overflow [vocalized-noise]? Overflow will happen if the two numbers are positive [vocalized-noise] or the two numbers are negative [vocalized-noise] because a negative and a positive number will never generate a carry [vocalized-noise].

So, if the two in a positive numbers; with sign 0 are added [vocalized-noise] and [vocalized-noise] is a negative number [noise] we will see why why [noise] [vocalized-noise] what is the reason [vocalized-noise]. So, if there are its a signed arithmetic [noise] for example, assume [noise] and there are two numbers [vocalized-noise] and you add them and then these are over.

Because you all know in digital design what is the concept of an over, but we are also looking in details with some examples in this [vocalized-noise]. For example, [noise] so ah oh [noise] like as I have told you [noise] let us take an unsigned number [noise] already we have taken the example; so, let us take 1 1 1 1 [noise]. So, definitely if I if I will be a carry over there [vocalized-noise] and in fact, [noise] in fact, if I say [noise]



[vocalized-noise] it cannot be accommodate be in the 4 bit [noise]. So, if I assume that the result has to be given in 4 bits [noise] and I have two numbers like 1 0 0 0 [vocalized-noise] and all triple ones [vocalized-noise].

So, of course, there will be overflow will be generated [vocalized-noise] and [vocalized-noise] and [vocalized-noise] also we will see the idea [noise]. So, if the negative [noise] number positive numbers [noise] whatever happens [vocalized-noise]. So, in other words [vocalized-noise] in a digital arithmetic if a overflow is generated [noise] based on the number of bits to store for the answered and number of bits to store for the operands [noise] [vocalized-noise] if it is a [noise] overflow [noise] is there [noise] it [vocalized-noise] between this [vocalized-noise] ah [vocalized-noise] it will be set [noise] other than this will be reason [vocalized-noise].

Like [noise] for example, if you add 0 0 0 0 with [noise] triple 0 with [noise]. So, 0 0 0 4 [vocalized-noise]; so, the answer is 1 0 0 [noise] unsigned [noise] arithmetic [vocalized-noise] of course, no over flow is generated the overflow flag is reset [noise] in this case very simple [vocalized-noise]. Equality as I told you [vocalized-noise] if [vocalized-noise] if [vocalized-noise] it is a [vocalized-noise]; it will [noise] this is restricted to a [vocalized-noise] compare instruction [vocalized-noise].

So, if there is the instruction called compare [vocalized-noise] and [noise] [vocalized-noise] then if the two numbers are equal [noise] then this flag is set [noise]. So, it is a come to structure [noise] and you give two operands [vocalized-noise] and if they are equal [noise] the answer with thus bit is set in the flag register otherwise it is a reset [noise] [vocalized-noise] interrupt enabled [noise].

So, we will this also a flag in which case you allow a interrupt to be [noise] occurred or not; that means, [vocalized-noise] ah main code is done in [vocalized-noise] whether will allow some other course to interrupt them [vocalized-noise] if you allow it [noise]. So, it will be [noise] its flag will be 1 [noise] [vocalized-noise] and if you are not allowing such a interrupt to interrupt [vocalized-noise] your code [vocalized-noise] then the [vocalized-noise] that is flag will be set to 0 [vocalized-noise].

So, at this point of time [noise] I am not going to elaborate more on interoperable enabled flag because it is a full unit [vocalized-noise] and module which is dedicated io and interrupts [noise] [vocalized-noise]. And simply like supervisor mode also

[vocalized-noise] some of the [noise] some some of the codes may like [vocalized-noise] assume in the supervisor mode [vocalized-noise]. So, in all those cases you have to set that flag [vocalized-noise] and if you are not allowing any code to run in the supervisor mode [noise] or user privileged or [vocalized-noise] super user privileged mode [noise] you can reset this bits [vocalized-noise].

So, these two will be discussed [noise] later whenever you are going to some advanced modules mainly we will be talking in a further down the line [noise] on the IO module [vocalized-noise] about interrupts [noise] [vocalized-noise]. Supervisor mode is something [noise] also related to [noise] [vocalized-noise] operating system and executing in a [noise] coordinate super user mode etcetera [vocalized-noise], but in details we will be looking at interrupt [noise] flags.

Whenever we will be discussing [noise] the chapter on with [noise] them [vocalized-noise]. Now [vocalized-noise] based on this [noise] some of the very important flags for us is the sign flag, zero flag, carry flag, parity flag, overflow flag and equality flag [vocalized-noise]. So, these are some of the most typically important flags [vocalized-noise] which [noise] will be used in everyday life of designing [noise] [vocalized-noise] sign a control instructions [noise] [vocalized-noise].

(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.

*JMP PC 5*  
*SU*

So, now we will look at some of the typical [vocalized-noise] control instructions based on the flags [vocalized-noise]. We first move simpler what is in the unconditional instruction [noise]; unconditional jump [vocalized-noise] that is you are at this memory location the Pc is say 5 [noise] this is the safe memory location [noise] where the code is using [vocalized-noise] where you can jump [noise] 50 [vocalized-noise]. So, without looking in [noise] anything [noise] the program counter [noise] is [vocalized-noise] going to become 50 [noise] [vocalized-noise].

Whatever instruction is present in the memory location 50 will be execute [vocalized-noise] there is a very simple unconditional jump instruction [noise] [vocalized-noise] no flags are required for that [noise] [vocalized-noise].

(Refer Slide Time: 21:50)

### Control instructions

- Example**  
The following code snippet illustrates the unconditional jump instruction.

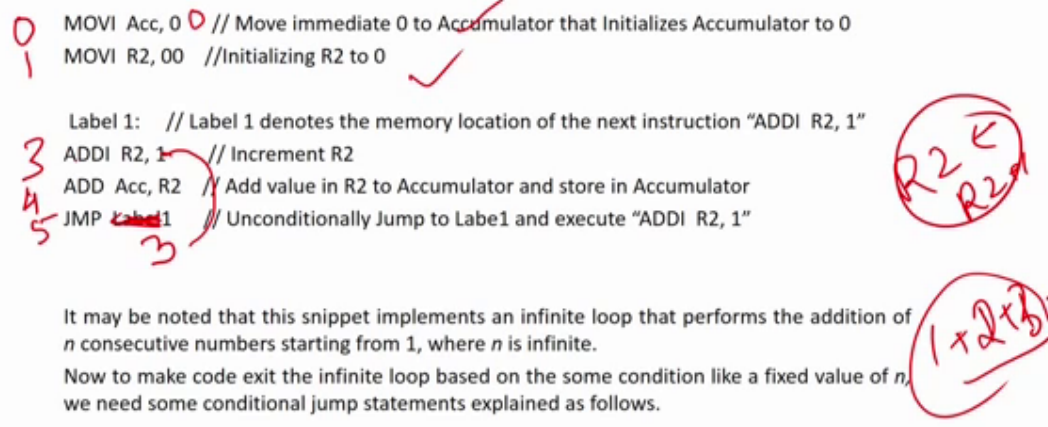
```

0  MOVI Acc, 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 Label1 and execute "ADDI 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 some condition like a fixed value of  $n$ , we need some conditional jump statements explained as follows.



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

We say that `move immediate 0`; it means the default destination `destination operand in the accumulator`. So, `move I 0` means the value of 0 will be directly the accumulator then `move R 2 0 0`. So, initialize this is also `move immediate R 2 0 0` so in fact, it is better to write `this 1 0 0` because from the size.

So, it is saying that `move immediate ah R 2 0 0` that is your resetting register R 2 as well as you are resetting accumulator to 0. So, in this case I am assuming the accumulate there is a 8 bit accumulator. So, these two are just initializing accumulator to 0 initializing user register to zero. So, in this ah 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 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 ah the it 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 `ADD 1 R 2 R 1`; that means, whatever this the content of R 2 it will be added with 1 and the value will be given to the R 2; that means, R 2 is equal to R 2 plus 1 that is nothing, but increment R 2.

So, the label 1 colon this 1 it means that `add I R 2 1` this as with the label `add` then what I am doing? You are adding accumulator to R 2; so, whatever is in the value of R 2 is stored back to the accumulator. So, now, R 2 is dumped into a `add` sorry `add` the value of accumulator to R 2.

So, whatever is in the accumulator [vocalized-noise] we will be added to R 2 and stored back to the accumulator [noise] see the [noise] accumulator is equal to [noise] accumulator plus R 2 [noise] [vocalized-noise]. Then jump to [noise] level 1 [noise] unconditionally again you jump back [noise] [vocalized-noise]. So, what do I mean by jump [noise] at level 1 means I want to jump and execute [noise] this instruction [noise] [vocalized-noise].

That is from jump level 1 is jump to [noise] add [vocalized-noise] ah jump to add immediate [noise] this instruction [noise] [vocalized-noise]; you want to jump [noise] to this instruction [vocalized-noise]. So, then how can I tell that [noise] jump [noise] unconditional to this instruction [noise]. So, it; so, some [noise] name has to be given [vocalized-noise]. So, label is nothing [vocalized-noise], but in the name which is given to this instruction [vocalized-noise]; so, this label 1 is a name [noise] which is given to this [noise] instruction [vocalized-noise].

That is just after [vocalized-noise] adding R 2 [noise] to accumulator [noise] and storing back the value in accumulator; again I wanted to jump back to this what. That means, [noise] [vocalized-noise] you are going to execute these two instructions [noise] in a [noise] indefinitely; there is no condition [noise] you execute this [noise] then you execute this [noise]; that means, R 2 equal to R 2 plus 1 [vocalized-noise] that is one incrementing R 2 [vocalized-noise].

And then again you are going to add the value of R 2 accumulator [vocalized-noise] and save it to the accumulator. That means, you are doing 1 plus, 2 plus [noise] and you are doing 1 plus [noise] 1 plus 1 [noise] and you are actually keeping on doing it [noise]. And it is the infinite loop [vocalized-noise]; if you like see [noise] what is happening it is the finite loop [vocalized-noise]. So, add immediate R 2 2 1 [vocalized-noise] that means R 2 is initially reset [noise] to 0 [vocalized-noise].

So, every time here [vocalized-noise] incrementing 1, [noise] 2, 3, 4 [vocalized-noise] and you are adding the same value to the accumulator [noise]. So, you are adding 1 plus 2 plus 3 [noise] plus dot dot dot [vocalized-noise] [noise], but as this your unconditional [vocalized-noise] jump [noise] label is the name of this instruction. So, you are jumping over there [vocalized-noise]; so, you are actually having a infinitum [noise] there is no exit from these two [vocalized-noise]. So, this just shows two very important things

forget about this ah [vocalized-noise] infinite loop [noise] business [vocalized-noise]; the idea is that jump [noise] unconditional means [noise] without taking anything you jump over here [vocalized-noise] and we are doing 1 plus 2 plus 3 [noise] so, on [vocalized-noise]. And [noise] basically as I want to say jump from [noise] this to some instruction [vocalized-noise].

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

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

So, label 1 is nothing, but memory location 3 where the instruction and R 2 plus R 1 is there [noise] [vocalized-noise]. So, when the code will be assembled and linked and loaded [vocalized-noise] you will replace it with the value of memory location 3 [vocalized-noise] that means [noise] after [vocalized-noise] at 5 that memory location number fifth [noise], you unconditionally jump [noise] to memory location number 3 and you keep on doing it [noise] 3, 4, 5, [noise] 3, 4, 5, 3, 4, 5 [noise] it will be continuously [noise] executing [vocalized-noise].

But that is [vocalized-noise] all these labels are etcetera are replaced to the memory location values [vocalized-noise] when they quote is par [vocalized-noise] because when you code be assembled [vocalized-noise]. So, the [vocalized-noise] these are all [vocalized-noise] thought in details [noise] that are code [noise] called assembled link control [noise] that is the system programming [noise] [vocalized-noise], but for us is enough to understand right now [noise] that from jump [noise] to label 1 means [noise] you are jumping from this one [vocalized-noise] to the name to this instruction whose

name is label 1 [vocalized-noise] and in this [vocalized-noise]. In fact, this is a [noise] [vocalized-noise] it is a loop instruction basically because there is no condition it has to if say infinitely [vocalized-noise].

(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 you are [vocalized-noise] as we have told [noise] that actually jump unconditionally use many things [noise], but we need [vocalized-noise] heart of [vocalized-noise] instructions on [noise] control instructions are basically on conditional instructions [noise] [vocalized-noise].

Like some of the examples it is given [vocalized-noise] jump any or jump on zero [noise] there jump [noise] if not equal to [noise] zero [noise] [vocalized-noise] jump any [noise] jump not equal to zero [noise] or jump not zero [noise] [vocalized-noise] right JEQ jump is equal to [noise] jump on Z [noise] JNC JL [vocalized-noise] JLO [noise] jump; if no carry [noise] or jump if carry [noise].

So, you can go through it [vocalized-noise] [noise] several [noise] type of instructions that there [vocalized-noise] jump L [vocalized-noise] jump [noise] if less [noise] [vocalized-noise] ah JGE [noise] jump is greater than or equal [noise] JN [noise] jump if negative [vocalized-noise].

So, based on the flag [vocalized-noise] like it is saying that [noise] jump any or jump not Z [noise] [vocalized-noise] in. In fact, [vocalized-noise] this time you are checking the jump not zero means you are checking the zero flag. [noise] [vocalized-noise] Jump any [noise] jump not equal to means if you are checking the [noise] equality flag [noise].

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

(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 Label1 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  $n$ .

*Handwritten notes:* EQ = 1, 3 NE, 10

Like we are now going back to the same example [noise] [vocalized-noise] of ah the same thing that there is a loop, [noise] we are resetting the value of accumulator. [noise] we are resetting the value of [noise] R 2 [noise] that is we are adding 1 plus 2 plus 3 plus 4 like that [vocalized-noise], but [vocalized-noise] in the previous step we were actually jumping it [noise] unconditionally [noise] back to the initial one [vocalized-noise] [noise] [vocalized-noise]; set reset this accumulator and registered 2 [vocalized-noise] then every time he was making R 2 1 1 [noise] implementing R 2 [noise].



Then every time you are adding the value of R 2 2 [noise] accumulator [noise] and the repeating this [vocalized-noise]; that means, we are not [noise] we are not existing out of the loop based on some [noise] condition [vocalized-noise]. Here we are actually using a loop [vocalized-noise] [noise] [vocalized-noise], the same example we are going to take, but here we are going to [noise] come out distance have several conditions [noise] [vocalized-noise]. Like if you look [noise] label 1 is the [noise] name of this instruction [vocalized-noise] then add R 2 the accumulator [noise] add R 2, R 1.

That is increment the value of R 2 same thing as above [noise] at the value of R 2 the [noise] accumulator [noise] that is add [noise] add accumulator to R 2 [noise]; here the conditional step comes in [vocalized-noise]. That is CMPI [vocalized-noise] that is compare the value [vocalized-noise] of R 2 to n [vocalized-noise]; that means, [noise] you are incrementing the value of R 2 [vocalized-noise] say I want to ADD 1 plus 2 plus 3 plus 4 up to 10 [vocalized-noise].

So, this n is in this case going to be 10 [vocalized-noise]. So, after doing the add R 2 [vocalized-noise] ah [vocalized-noise] R 2 2 accumulator and saving that the value accumulator [vocalized-noise] we are going to check [vocalized-noise] whether R 2 has reached the value of 10 or not [noise]; here it is 10 [vocalized-noise]. So, there is a [vocalized-noise] CMPI instruction [noise] [vocalized-noise] and whenever R 2 will be equal to 10 [vocalized-noise]; that corresponding [noise] equality flag will be checked [noise] when they made set [vocalized-noise].

And then you can say [noise] jump not equal [vocalized-noise]; that means, if the equality flag is not set [noise] jump not equal then again you jump back to label [vocalized-noise]. And whenever this will be equal that is [vocalized-noise] R 2 and n will be equal because n is equal to 10. So, n R 2 will have the value of 10 [vocalized-noise] then jump not equal to label 1 will become false; because now they will be equal.

Because jump not equal [noise] J not equal; it is true [noise] if and only if [noise] the equality flag is reset (Refer Time: 30:21). Whenever the equality flag is true jump not equal who will become false you know the equality flag will be set [noise]. So, there is something called equality flag [vocalized-noise]; so, if the equality flag is equal to [noise] 1 sorry equality flag is 0; that means, when two stuffs are not equal, two operands are not equal [noise] [vocalized-noise] then jump not equal will be true [vocalized-

noise], but whenever two numbers will be equal then what is going to happen the equality flag will be set [vocalized-noise].

And then jump not equal who will become false [vocalized-noise] whenever jump not equal 2 will be false; [noise] it will not jump to label [noise], but we will go and execute the next instruction [vocalized-noise]; so, it will come out of them [vocalized-noise]. So, it gives a very nice example [vocalized-noise] that how the other [vocalized-noise] infinite loop of adding 1 plus 2 plus 3 has been modified to [vocalized-noise] ADD 1 plus 2 plus 3 [noise] up to that [vocalized-noise].

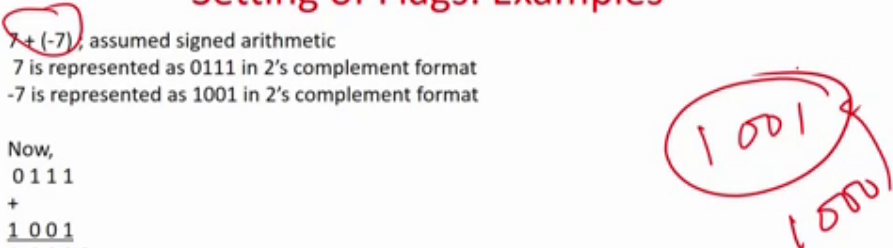
So, it gives a very nice idea that ah just before a comparison instruction we do a corresponding comparison [vocalized-noise] ah sorry we do a comparison instruction set the corresponding flags [vocalized-noise] and just by looking at this flag [noise]; we decide either to go to the [noise] top of the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

(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 [vocalized-noise] because [noise] ah whenever we will be looking at more ah different complicated codes [noise] in the next module; we will be always using so, many times [vocalized-noise] these ah control instructions [noise], but [noise]

[vocalized-noise] in this unit we let us look at the more interesting [noise] part of it that is how the flags are set [vocalized-noise] or which flags are set [vocalized-noise]. If we can find out which flags are set or reset [vocalized-noise] then after that it is very simple to think about the control instructions [vocalized-noise] because they have [noise] just option [noise] true or false.

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

So, now we are seeing [vocalized-noise] with different examples [vocalized-noise] how different flags are set which flags are set, [noise] and which flags [noise] are reset [vocalized-noise] that is more interesting [vocalized-noise]. So, for example, they are doing 7 and minus 7; so, as both plus and minus are involved [noise]. So, it is a sign that is value [vocalized-noise] 7 [vocalized-noise] is represented as 0 1 1 is 2's complement [vocalized-noise] minus 7 is nothing, but in 2's complement it is 0 1 1 and [noise] again you have to add a 1.

So, it is [noise] [vocalized-noise] sorry [vocalized-noise]; so, this is actually [noise] 2's complement [noise] of minus 7 [vocalized-noise] because [vocalized-noise] a once complement of minus 7 is nothing, but [vocalized-noise] 1 0 0 0, you add a 1; you get this [noise]. So, this is actually the [noise] [vocalized-noise] 2's complement of [noise] 7 that is [noise] minus 7 [vocalized-noise]. So, minus 7 and plus 7 are represented over here [vocalized-noise] just I recall from the digital design [noise].

So, the LSB is 0; [noise] it is a positive number [noise]; so, this is a positive number and this any number [vocalized-noise]; now let us add it [noise]. So, as [noise] it is a it is a signed arithmetic [vocalized-noise]. So, if you add it you are going to get 1 [noise] 1 plus 1 is 0 [vocalized-noise] carry will be 1. So, 1 plus 1 again 0; the carry will be 1; so, 1 plus 1 is 0 [noise] again 0 and this is generating [noise]. 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

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.

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

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

Whether a carry is generated it is [noise] not generated whether you want to reject be carry because [noise] of 2's complement [noise] arithmetic it has nothing to do [vocalized-noise] it has got the 4; 0's as the answer [vocalized-noise]; so, the zeroth flag is set [noise] [vocalized-noise]. A new checks or the 4 bits ah if the answer is 0 as this words [vocalized-noise] in this case the zeroth flag is 1 [vocalized-noise]. Then the MSB

of the final answer [noise] after negating we carry because [vocalized-noise]. So, in this case zero flag is equal to 1 that is the first thing [noise] because [noise] it has nothing to do with [noise] any other stuff [noise].

You just take the 4 bits as the answer [noise] or 0 [noise] serious flag is set [noise] [vocalized-noise]. Now the MSB is 0 [noise] because as I told you [noise] in 2's complement arithmetic [vocalized-noise] we cannot neglect the carry [noise] 0; so, indicating that is a positive answer [noise]. So, [noise] the ah; so, the answer is positive, so if there is a positive flag the answer will be 1; [noise] in these case N [noise] in case say is a negative flag N.

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

Negative flag is reset because these MSB is equal to [noise] 0 [noise] 0 is I thing 2's complement arithmetic at the MSB will [noise] denotes a positive number [vocalized-noise]. Now look at the carry [noise]; so, a carry is generated [vocalized-noise]. So, as a carry is generated the [vocalized-noise] C flag of the carry flag is set to be 1 [noise] [vocalized-noise], but again as we arithmetically signed [noise] the value of carry is ignored [vocalized-noise].

So, that is very important [vocalized-noise]; so, in a 2's complement arithmetic what happens? We have [noise] done this and [noise] if you look at the de facto [noise] standard in digital design [noise] in 2's complement [noise] arithmetic, we are not actually bothering [noise] about the carry which is generated [noise] that is a do not care condition [vocalized-noise]. But [noise] in a hardware when the flags are set or reset in to have a look at all those contexts [vocalized-noise].

This is zero the zeroth flag will just check [noise] what are the [vocalized-noise] answer of the [vocalized-noise] 4 bits [noise] as it is 0 [noise] these 0 the flag is set [noise] this bit is [noise] 0; that means, the positive number. So, the [noise] positive flag will be set [noise] or the negative flag is reset [noise] a carry has been generated [noise]. So, in this

case the carry flag is set [noise], but as [noise] the arithmetic or the instructions [noise] you have executed [noise] [vocalized-noise] we know the in the 2's [noise] complement numbers I have given as input [vocalized-noise]; so, I will not use the carry flag [vocalized-noise].

So, in other words the [noise] flag setting logic [noise] is totally blind [noise]; it is taking two numbers 0 1 1 1 and 1 0 1 and generating the answer as 1 as a carry [vocalized-noise] and 4; 0s as the answer [vocalized-noise]. And accordingly the flag bits of 0 is set [noise] [vocalized-noise] the flag bit of negative number is reset [vocalized-noise] and [noise] a carries [noise] the flag set [vocalized-noise], but as I know [noise] as a programmer.

That I have given the two numbers which are the input has 2's complement arithmetic [noise] and in this case the carry is not [vocalized-noise] calculated or ignored [vocalized-noise]. So, I have to myself ignore the carry flag [vocalized-noise] even if they said I should not use it as a [noise] [vocalized-noise]. If I use the value of carry bit [vocalized-noise] immediately after this instruction [noise] to do some conditional check and jump [vocalized-noise], there may be a logical error in my code [vocalized-noise].

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

So, again I will come to that; so, [vocalized-noise] important add [vocalized-noise] basically this 3 [noise] [vocalized-noise] that the zero flag is set, [noise] the negative flag is reset, a carry flag is set [noise], but it has to be ignored [noise] [vocalized-noise] similarly the [vocalized-noise] 4 bits the answer [noise] is 0. So, the th flag will be set [vocalized-noise] and [noise] if the sorry I mean the ah [vocalized-noise] the 4; there are 4 0's [vocalized-noise]; so, the number of parity is even [vocalized-noise].

So, given parity [vocalized-noise] is decide [noise] [vocalized-noise] to be more the [noise] and the [noise] numbers are of different signs, [noise] some sign flag will be set

to 0 and so, [vocalized-noise] Sso, forth [noise]. So, lot of flags will be there and based on the values the flags will be set or rest [vocalized-noise], but which flag has to be ignored has to be decided by us [noise]. Again I will take another example; so, [noise] to make the things [noise] more [noise] easier [vocalized-noise] like for example, I have taken 2 [noise] and I have taken minus 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. Z →

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 minus 3 I am going to do [vocalized-noise]. So, this is our 2's complement [noise] implementation of the minus 3 and 3 [noise] if I do the answer [vocalized-noise] and ready to get this as the answer [vocalized-noise]. So in the basically if the answer should be equal to nothing [noise], but minus 1 [vocalized-noise]; so, in this case what happens? [noise] the 4; 1s are there [vocalized-noise]. So, the [noise] it is checking that the last bit is 1 [noise].

So, the zero is flag [noise] let us first say less at the zeroth flag [vocalized-noise]; so, the all the answers are 1 [noise]; so in fact, [noise]; obviously, the answer is not the [vocalized-noise] 0. So, the zeroth flag is set to 0 [noise] that is obvious [vocalized-noise]. The MSB is 1; so, it is a negative number [noise]; so, the negative flag is set to 1 [noise] obviously there is a [vocalized-noise] negative number because there ah [vocalized-noise]. So, the 2 minus 3 is minus 1 [noise]; so, negative flag you set [vocalized-noise] there is no carry will be generated [vocalized-noise].

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

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

So, in both the cases the overflow flag is set to 0; the 4 answers are 4 1's [vocalized-noise]. So, if the 4 answers are 1 [noise] then what is the case? Is an even parity; so, the even parity flag is set to 1 [vocalized-noise].

(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, Q 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 [noise] considered [vocalized-noise], but every time the output or the [vocalized-noise] sorry in the overflow flag O is the overflow flag [vocalized-noise]. The overflow flag we are actually not considering right now because [noise] one number is positive and one number is negative [vocalized-noise].

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

So, I add two numbers 8 plus 8; so, you are going to get the answer as [noise] 1 [noise] 0 0 0 that is 16 [vocalized-noise], but now you see these are the 4 bit which are of importance [vocalized-noise] this one bit carry or overflow as been generated [vocalized-noise]. So, now, this [noise] sorry [noise]; so, as the hardware it will just check [vocalized-noise] the answers are 4 0. So, that is true that the zero flag is set [vocalized-noise].

The MSB is [noise] zero [vocalized-noise]; so, [vocalized-noise] it is not going to check the carry 1 [noise]. So, the answer is 1 0 0 0; 1 0 0 0 [vocalized-noise] both the answer is a 0 0 0 0 [vocalized-noise] the carry is generated as 1 [vocalized-noise]. So, [vocalized-noise] this is checked the MSB is 0; [noise] it is not going to look at the overflow [vocalized-noise] [noise]; so, the negative flag is reset [noise] [vocalized-noise] again ah [noise] in this case also you can ah ignore the [noise] ah negative or [vocalized-noise] positive flag here [vocalized-noise]; using this case is an unsigned arithmetic [vocalized-noise].

Previous version [vocalized-noise] using a signed arithmetic in 2's complement [noise] [vocalized-noise]; this was very very important [noise]; then you get the flag was [noise] very important. In that case the [noise] zeroth MSB means a [noise] [vocalized-noise] positive number and 1 in the MSB as a [noise] negative number [vocalized-noise]. So, in large context [noise] you had to take [noise] into mind that I have a consider [noise] this flag [vocalized-noise]. So, if I take 1 0 0 [noise] and 1 0 0 [vocalized-noise] sorry 1 0 0 0

and 1 0 0 0 in 2's complement arithmetic, these are two negative numbers basically. So, if you are a programmer you know that I have given the two numbers as input 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 unsigned arithmetic. So, you have two negative flag 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 2's complement arithmetic and one number was negative and one numbers positive. So, you are neglecting the carry flag, but here as we are using an unsigned arithmetic a carry as been generated and in such unsigned arithmetic the carry flag is set to 1, ah carry has been generated and you have to consider this.

Similarly ah the number of plus if you see; the number of ones in the answer is 0, the parity flag is set to 1, but again since both numbers are negative, but the sign is 0 which indicates the answer is positive.

So, overflow flag is 1; so, in this case you see what why is an overflow ah for example, if both the numbers are negative. So, if you consider ah; so, again this one is very important is you are considering ah ah sign arithmetic. So, if you are taking a sign arithmetic then 1 and 1; they are two negative numbers.

Since both the numbers are negative sine bit is 1, if you are considering a 2's complement arithmetic, but the sign bit of the answer is 0 that is this is a 0. So, had these been the two negative numbers; so, what are the 1 0 0 and 1 0 0. So, if you are taking a 2's complement arithmetic it will be 0 0 ah it will be 0 sorry ah this is actually nothing, but 0 1 1 1, you add a 1 nothing, but minus 8.

But this is nothing but [noise] equal to minus 8 [noise] and minus 8 [noise] [vocalized-noise]. So, if you are taking it is a signed arithmetic format [noise] this is minus 8 [noise] basically [noise] [vocalized-noise]. So, in that case [noise] you are adding minus 8 [vocalized-noise] and minus a. So, in this case your answer should be minus 16 [vocalized-noise], but somehow you are giving the MSB as 0 [vocalized-noise]. So, in this case there is a [noise] which is the both the numbers are negative [vocalized-noise], but the answer is 0 which indicates that the answer is positive [vocalized-noise].

So, the overflow flag is 0 [vocalized-noise]; so, the over flag overflow [vocalized-noise] or [vocalized-noise] overflow [noise] flag is set to 0; that means, [vocalized-noise] ah [vocalized-noise] the overflow flag is set to 1 [vocalized-noise] because [vocalized-noise] had this is a signed arithmetic. So, this minus 8 minus 8; [noise] the answer should have been minus 16 [vocalized-noise].

But some of the answer is showing [noise] MSB as 0 [noise] which is basically wrong due to an overflow [vocalized-noise]; so, the [noise] overflow flag is set to 1 [noise]. But [noise] in this context [noise] as we are using an unsigned arithmetic [noise], you have to totally neglect the overflow flag [vocalized-noise]. So, in this case we are going to neglect the overflow flag, but we are going to take the [noise] carry flag because they are both unsigned numbers [vocalized-noise] one has been generated which is nothing, but your [noise] carry flag [vocalized-noise] [noise].

But [noise] this is not a mistake with there is a no overflow [noise] because we are considering only [noise] unsigned positive number [noise]. So, in this case this is plus 8 [noise] and not minus 8 [noise] in the 2's complement format [vocalized-noise]. So, in other words [noise] it is very very important [vocalized-noise] to decide what is the context and what flex I have to take and what flex [noise] I do not have to take [vocalized-noise].

Like when I am adding two numbers [vocalized-noise] which are positive [noise] [vocalized-noise] immediately have to think that [vocalized-noise] I will take the [noise] zeroth [noise] ah I will take the zeroth flag [vocalized-noise], I can take the negative flag [vocalized-noise], I can take the parity flag, I can take the carry flag, but as they are two sign arithmetics I am [noise] [vocalized-noise] adding it [vocalized-noise]. So, the

overflow flags can be neglected for the time being like just like this; here [noise] it is a [noise] unsigned [noise] arithmetic two numbers you are generating [vocalized-noise].

So, [noise] the carry flag will be neglected over here [noise] because [vocalized-noise] in a unsigned [noise] binary one is negative one is positive [vocalized-noise] or in 2's complement subtraction when you are doing; [noise] we always neglect the [noise] carry [noise] [vocalized-noise] 2's complement positive [noise] or negativity whatever when you are doing in 2's complement we may neglect the [noise] [vocalized-noise] carry [vocalized-noise].

So, since both the numbers are of different size ah [vocalized-noise] ah the the different symbols like one is positive and one is negative [noise]; the overflow can never [noise] will generate [noise]. So, the overflow flag will always be [vocalized-noise] 0 as discussed [noise], but we are not neglecting [noise] that is very very important over here [vocalized-noise] ok ah some more [vocalized-noise] very simple examples [noise].

(Refer Slide Time: 45:30)

**Setting of Flags: Examples**

4+5; assumed unsigned arithmetic  
4 is represented as 0100 in unsigned format  
5 is represented as 0101 in unsigned format

Now,  
0100  
+  
0101  
1001

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. However, as this is unsigned authentic N is ignored.

Now as no carry is generated the C Flag is reset to 0.

Since both the numbers are positive (i.e., sign bit is 0) but sign bit of the answer is 1, which indicates that answer is negative. So, O flag is 1. However, as this is unsigned authentic N is ignored.

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

Like 5 plus 4 [noise] if you are going to do [noise] all this is a unsigned arithmetic [vocalized-noise]. So, you add this you are going to get the answer as this [noise] [vocalized-noise]; obviously, the zero flag is reset [noise] because the answer is not 0; [vocalized-noise] the MSB is 1 indicating is a negative number [noise], but again is an

unsigned arithmetic [noise]. So, you have to neglect the negative flag [vocalized-noise] there is no carry generated over here. So, the carry flag is reset we have to consider this [vocalized-noise] because 4 plus 5 is 9; [vocalized-noise] you are doing the operation in a 4 bit arithmetic.

So; obviously, [noise] no carry will be generated [noise] two numbers you are taking plus 4 and minus 4 the MSBs are 1 [noise] which is a negative number [noise]. So in fact, a overflow flag will be set [noise] because two negative numbers [noise] numbers who are takes [vocalized-noise] two positive numbers who are taking the answer should [noise] always be positive [vocalized-noise], but the answer is showing an MSB. But again as I told you is an answer in the arithmetic [vocalized-noise] overflow flag will be set to 1, but is an answer in arithmetic. So, it is ignored [noise] the number of 1's are [noise] 2 over here even parity [noise] even if fact flag is set and you have to consider this flag [noise].

(Refer Slide Time: 46:28)

### Setting of Flags: Examples

7+1; assumed signed arithmetic  
 7 is represented as 0111 in 2's complement format  
 1 is represented as 0001 in 2's complement format

```

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

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.

Since both the numbers are positive (i.e., sign bit is 0) but sign bit of the answer is 1, which indicates that answer is negative. So, O flag is 1. As O is 1, the answer is not valid and so is the flag N.

As the number of 1s in the answer is 1 (odd), EP flag is set to 0.

Similarly, [noise] again if I ah [noise] take some other example 7 plus 1 is equal to 8 [noise]; same thing [noise] is going to be the answer answer is not all 0 [noise] zeroth flag is reset we have to take it [noise] look at the flag MSB we final answer is negative [noise]. So, negative flag is set to 1; again [noise] ok [noise] just a minute [noise].

So, let us assume that this slightly ah [noise] ah different in this case. So, in this case I take [noise] 7 plus 1 [vocalized-noise], but in this case I assume a 2's complement arithmetic [noise] like in this case this was just to just to keep your variation [vocalized-noise]. So, in this case ah is a unsigned arithmetic [vocalized-noise] I could have also taken this in unsigned arithmetic version [vocalized-noise], but in this case [noise] I am using [noise] two positive numbers, but still I am using a signed arithmetic version [noise].

So, just I am trying to see what is the difference [vocalized-noise]? So, in this case; so, I am adding 7 plus 1. So, it is 8 [noise] to this is the case [vocalized-noise]; so, in this case I am using a 2's [noise] arithmetic [vocalized-noise] 2's complement arithmetic [noise] 2's numbers are positive [vocalized-noise], but still I am using [noise] a sign that here [noise] just to give the example [vocalized-noise]. So, MSB is 1; so, indicating the number is negative; so, the negative flag is set to 1 [noise] [vocalized-noise].

So, there is no carry generated [noise]; so, the carry flag is set to 0 [vocalized-noise]. Since both the numbers are positive [noise] that is sign bits are 0 [vocalized-noise], but the sign of the answer is one which is a reverse [noise]. So, the overflow flag is set to 1 [noise] [vocalized-noise] as 1 is also this is [vocalized-noise] as [noise] 0 is 1 [noise] the answer is not [noise] valid [noise] and so, is flag a [noise].

Now we are going to see in details [noise] only 1 1 in this answer. So, in this case [noise] is a odd parity [noise]; so, the odd parity flag is set to 0 [vocalized-noise]. Again in this case [noise] all flags are valid [noise] again let us we look what we are doing and what is the [noise] difference [vocalized-noise]. So, in this case plus 5 and plus and 4 they are positive numbers we are using a [noise] unsigned arithmetic [noise] format [vocalized-noise]. So, I have told you [noise] what it means [noise] in this case [noise] I am taking two [noise] positive numbers [noise] like 7 and 7 [vocalized-noise].

But in fact, [noise] I am using a sign arithmetic version [vocalized-noise] that is ah is ah [vocalized-noise] 2's complement arithmetic [noise] where the ranges from minus 8 in this 4 bit number [noise] minus a to plus 8 [noise] this is the rate [noise] you all know from digital design fundamentals [vocalized-noise]. But in this case there are 4 bit numbers [vocalized-noise] and this is a unsigned arithmetic [noise]. So, you can go from 0 to [noise] 50 [noise] that is the range difference [noise] [vocalized-noise].

(Refer Slide Time: 48:39)

### Setting of Flags: Examples

7+1; assumed signed arithmetic  
7 is represented as 0111 in 2's complement format  
1 is represented as 0001 in 2's complement format

Now,  
0 1 1 1  
+  
0 0 0 1  

---

  
1 0 0 0

-8

+7

+8

01000

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.

Since both the numbers are positive (i.e., sign bit is 0) but sign bit of the answer is 1, which indicates that answer is negative. So, O flag is 1. As O is 1, the answer is not valid and so is the flag N.

As the number of 1s in the answer is 1 (odd), EP flag is set to 0.

So, in this case I have [vocalized-noise] talking on a 2's complement rate [noise] that is minus 7 to [noise] plus 7 [vocalized-noise]. So, if I do the addition [noise] we are going to give this as the answer [vocalized-noise]. So, as [vocalized-noise] if you look at it [noise] it is a invalid answer [vocalized-noise] because [noise] 7 plus 8 is going to be positive 8 [noise] [vocalized-noise], but positive 8 cannot be represented in 4 bits in 2's complement arithmetic [noise]; these are very well known thing I think if if you are [noise] forgot.

You can just go and [vocalized-noise] if I do a digital [noise] design fundamentals [noise] [vocalized-noise]. Because positive 8 will be 0 [noise] 1 0 0 [noise] this is actually positive 8 [vocalized-noise] because 1 0 0 0 is [noise] negative 8 in [noise] 2's complement arithmetic [vocalized-noise]. So, this in fact, [noise] that is why I told you them going out of [noise] range; so, this will be the incorrect answer [vocalized-noise].

So, if you do this [noise] we are going to get the answer [noise] this one; 1 0 0 0 [vocalized-noise] as a hardware you does not understand much, [noise] it [noise] will just [vocalized-noise] develop the value of flags [vocalized-noise] based on certain hardware computation [vocalized-noise]. So, 0 0 flag is reset [noise] it is taken [vocalized-noise] MSB is 1 [noise] directly indicating that is a negative number [noise]. So, negative flag is set [noise] it is a negative number. In fact, as I told you the answer should be [noise]

positive [noise]; positive 8 [vocalized-noise], but we are not going to get the answer positive 8 in a 4 bit answer [noise].

So, therefore, [vocalized-noise] you require a larger space or a 5 bit [noise] space to implement [noise] [vocalized-noise]; if you could have done in this way then your answer would have been correct [vocalized-noise] would have [noise] bought this as the answer [vocalized-noise] which is going to give you [noise] the correct answer in signed arithmetic [vocalized-noise].

But as you are using a [noise] 4 bit number to do this [noise] so in fact, that is why [noise] you are going to get an overflow [noise] and all the problems have started [vocalized-noise]. So, MSB is 1 1 [noise] the negative flag is 1 [vocalized-noise] there is no carry generated [noise], those carry flag is listed [vocalized-noise], but here the overflow flag is actually our main role player here [vocalized-noise]. So, it is finding [vocalized-noise] both did answers [noise]; both the inputs are 0 0 [noise] that is that positive numbers, but the answer is a [noise] negative number [vocalized-noise].

So, immediately [noise] it is going to say that the [noise] overflow flag is 1 [vocalized-noise]. So, now, we have to check [noise] [vocalized-noise] that when the overflow flag is 1 [vocalized-noise] and the answer is negative [noise]. So, both of them are saying that the answer is not valid [noise] and so, is the [noise] negative flag [noise]. So, immediately whenever a overflow is generated [noise]; that means, you have to understand that overflow is different from carry [vocalized-noise].

Carry means some carry has been generated [vocalized-noise] and [noise], but the answer is valid [noise]. But in [vocalized-noise] in this case what happens [vocalized-noise]; so, with the carry the whole answer is done [noise], but in this case mainly the overflow generated [vocalized-noise] with this not carry generated [vocalized-noise], but the answer is actually a wrong answer [vocalized-noise] because of the over flow [noise].

So, I could have easily connected [noise] by putting its [noise] as a 1 bit [noise] additional I mean I could have [noise] if I would have done with the 5 bit; the answer would have been carry [vocalized-noise]. So, whenever the overflow flag is said [noise] so, immediately it will say the answer is invalid [vocalized-noise] as well as the negative flag is also invalid [noise] this sign is also invalid [vocalized-noise] and the answer is also invalid [noise] [vocalized-noise].



But that is why if I just compare [noise] [vocalized-noise] again with the [noise] this one [noise] you could have checked [noise] the ah [vocalized-noise] [noise] see both the numbers of different sizes, this is of different size, this is of different size; then immediately the [noise] ah over the flag is reset [noise]; that means, if the two numbers of different [noise] like one number is positive one number is negative [noise] in a signed arithmetic [vocalized-noise]; you can never generate an overflow [vocalized-noise].

Because then if two numbers are subtracted; [vocalized-noise] the answer is always less [noise]. So, if you can represent [noise] one negative number sorry positive [noise] and one negative number in 4 bits [noise]; then the answer will always can be represented in 4 bits [noise] because you are making the number [noise] less due to subtraction [vocalized-noise].

But if there are two numbers of same sign [noise] positive number or negative number [vocalized-noise]; then the number can become larger than the two operand itself and it requests to take more number of bits [noise] that is what actually has happened [vocalized-noise]. 7 plus 1 is 8 [noise] [vocalized-noise] which is plus 8 in the signed arithmetic is nothing, but is 0 1 0 0 [noise] 0; 5 bits [noise] it cannot be accommodated [noise] in 4 bit [noise].

So, ah [noise] overflow has been generated which actually [noise] neglects the [noise] [vocalized-noise] you have it will not do anything [vocalized-noise], but just stuff [noise] with the operation some flags are set and reset [vocalized-noise]. So, whenever you find about this overflow flag is set [vocalized-noise]. So, we have to know that the answer is wrong and therefore, in fact, you have to [noise] give more precision to this answer. So [vocalized-noise] so, that is what is the idea of setting the flag [vocalized-noise].

So, we have very very complicated situation [noise] and maybe later we will see how to use this overflow flag to set a [vocalized-noise] ah [vocalized-noise] how can you [noise] generate a [vocalized-noise] instruction [vocalized-noise] based on the overflow flag [vocalized-noise] to see whether the [noise] answer is valid or not [vocalized-noise]. So, for example, if you have a 32 bit machine [vocalized-noise] and you are taking all [noise] largest possible number you have fitted up into the [noise] memory [vocalized-noise].

And you are doing the [vocalized-noise] computation [vocalized-noise]; so, every time [noise] you have to check whether there is an overflow or not [noise] [vocalized-noise]. Since you are getting a overflow bits set [vocalized-noise] by default you have to be clear [vocalized-noise] that this is a valid answer ah or the position is an error; so, all these things you have to [noise] report [vocalized-noise]. So, this complicates things maybe we will try to see whenever we will [vocalized-noise] knowing [noise] going more in to assembly language coding [vocalized-noise] and micro [vocalized-noise] micro programming etcetera [noise] [vocalized-noise].

(Refer Slide Time: 53:17)

### Questions and Objectives

- Q1: What is a program status word and what does it contain.
- Q2: What do you mean by condition codes or flag bits? Indicate the purpose and use of the following flags. Also indicate when these bits get SET or RESET. Sign, Zero, Carry, Auxiliary carry, even parity, overflow, equal, interrupt enable, supervisor mode.
- Q 3: Indicate the use of these flag bits to design some instructions for the processor. Explain with examples.

- **Comprehension: Discuss:--**Discuss about flag bits and how these flag bits get set or reset.
- **Synthesis: Design:--** Uses of flag bits to design conditional statements.

So in fact, what we have done in this class [noise]? In this class basically we have seen in the crux of what is the ah [noise] conditional instructions and more importantly [noise] [vocalized-noise]; how basically flags are set and said that is [noise] the most difficult part of it [vocalized-noise].

Once you understand how a flag is set and reset [vocalized-noise] and then accordingly you can easily generate [vocalized-noise] the instruction based on that. And visually [noise] truth and false you go over to the label [vocalized-noise] or you just execute the instructions [noise] let us do it [noise] [vocalized-noise]. So, ah some typical questions [noise] basically which can be asked [noise] and let us see how we fix your objective [vocalized-noise] like for example, the first question is [vocalized-noise] what is the

program status word [noise] and what it contains? [vocalized-noise] Then the second question said [noise] what are flag bits, [vocalized-noise] what are different types of flag bits [noise] like set [noise], reset, auxiliary set, [noise] overflow [noise] etcetera [vocalized-noise].

Indicate the use of these flag registers [noise] and with examples [noise] how can you do it [noise] and basically [vocalized-noise] what instructions can be done [noise] ah what purpose it can solve [vocalized-noise]. So, if you look at the [noise] discuss about flag bits and how these flag bits are [noise] set and reset [vocalized-noise]. So, if you are able to answer this and if you are able to answer this of course, [noise] this [vocalized-noise] instruction is made [noise] [vocalized-noise].

Use of flag bits to design [noise] conditional statements [noise]; that is again ah this ah [vocalized-noise] if indicating use of [noise] flag bit should [noise] design some instructions [vocalized-noise]; so, is the synthesis objective [noise]. So, after the [noise] after doing this unit [vocalized-noise] when you will be able to [noise] as if you are answering the question [noise] number 3 that you are designing [vocalized-noise] newer instruction [noise] some instruction sets for a [vocalized-noise] processor based on these flags, [noise] then basically you are able to satisfy the synthesis [noise] of the tips of designing [vocalized-noise] conditional statements [noise].

And when about like talking about [vocalized-noise] program status word; so, whenever this flag bits etcetera [noise]; that means, flag bit self [noise] [vocalized-noise] you can [vocalized-noise] whenever you are [noise] synthesizing on the use of flag bits for [noise] design conditional statements this implies that [vocalized-noise] you have to know that when I am ah [noise] jumping from [noise] one context to another [vocalized-noise] then everything has to be set [vocalized-noise].

And basically what are the values like intermediate registers acceptor which has to be [noise] which are in a program status word what it contains [vocalized-noise] basically where it is to be saved [noise] [vocalized-noise]? This will actually satisfy the objective on comprehension [noise] [vocalized-noise] so in fact, ah just strapless unit [noise], you will be able to meet these two objectives which we targeted [noise] in the beginning [vocalized-noise].

So, ah this actually completes the ah [vocalized-noise] second one in the seventh [noise] unit of this module [vocalized-noise] and next time we will see how to use these instructions please jump instructions or conditional instructions [vocalized-noise] to execute one very important part of your [noise] programming paradigm; that is actually your functions and procedures.

Thank you [noise].