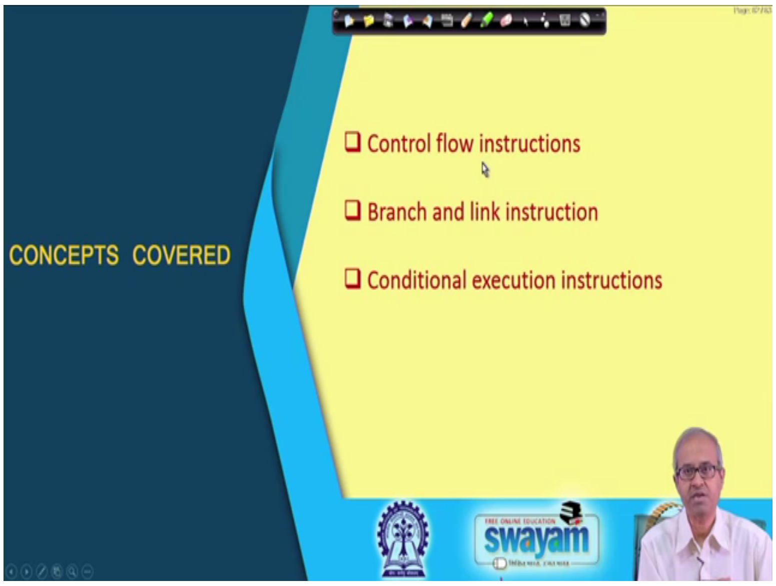
Embedded System Design with ARM
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 09
ARM Instruction Set (Part III)

So, we now discuss the control flow instructions that are available in the ARM Instruction Set.

(Refer Slide Time: 00:31)

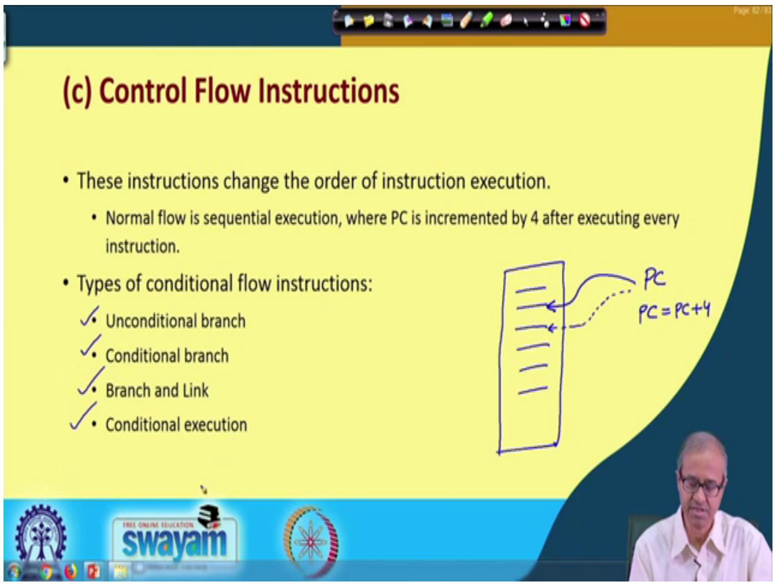


CONCEPTS COVERED

- Control flow instructions
- Branch and link instruction
- Conditional execution instructions

So, this is the third part of our lecture series on ARM instruction set. So, here we shall be talking about the control flow instructions. Well, in particular we shall be looking at the branch and link instruction which are primarily used for handling subroutine calls and we shall talk about the conditional execution instruction which you can also regard to be belonging to this category control flow.

(Refer Slide Time: 00:58)



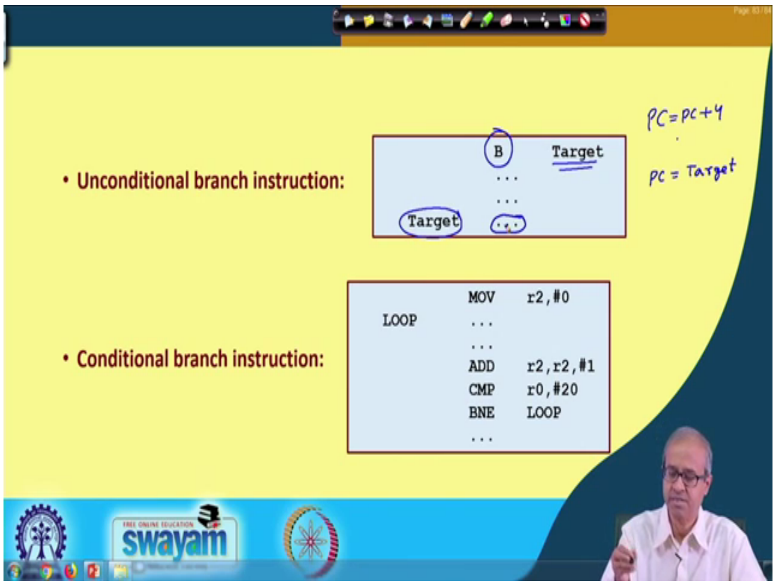
(c) Control Flow Instructions

- These instructions change the order of instruction execution.
  - Normal flow is sequential execution, where PC is incremented by 4 after executing every instruction.
- Types of conditional flow instructions:
  - ✓ Unconditional branch
  - ✓ Conditional branch
  - ✓ Branch and Link
  - ✓ Conditional execution

Let us look at the control flow instructions. Now as I had said earlier, control flow instructions are those which changes the sequence of instruction execution right. So, in the normal flow of execution; now in memory if we look into the memory map the instructions are normally stored one after the other, the program counter it points to the next instruction to be executed. So, whenever one instruction finishes execution, you increase or increment PC by 4. So, that now PC will point to the next instruction, each instruction is of size 32 bits. So, you will have to increase it by 4 bytes ok. This is how it works.

Now in control flow says that here this PC is not being incremented by 4, but rather you are updating PC with some other value. So, the next instruction will be from somewhere else. So, in terms of the types of control flow instructions, there can be unconditional branch, conditional branch, branch and link and conditional execution instructions. There can be broadly these 4 categories of instructions.

(Refer Slide Time: 02:35)


- Unconditional branch instruction:

```
        B     Target
              ...
              ...
Target        ...
```

- Conditional branch instruction:

```
        MOV   r2,#0
LOOP    ...
        ...
        ADD   r2,r2,#1
        CMP   r0,#20
        BNE   LOOP
        ...
```

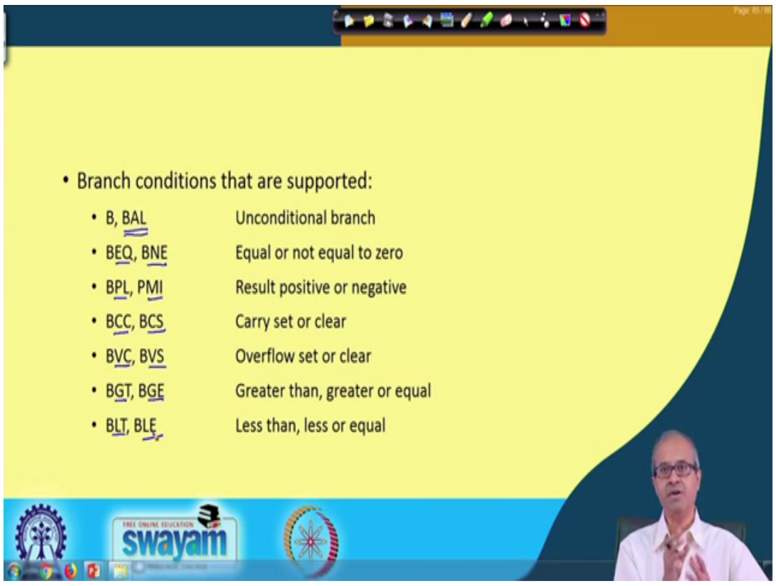
So, let us look at these instructions one by one. The first one is the unconditional branch instructions. Now in the unconditional branch instructions, we have an instruction called branch which is just B mnemonic is just B. So, in the assembly language we just say B some label Target.

So, in the assembly language level I can use a label somewhere. It means after this instruction we shall not go to the next instruction, but rather next instruction will be here. So, the way pc will be incremented is that instead of doing pc equal to pc plus 4. So, what we are saying is that we will be doing pc equal to Target; so, that the next instruction that will be fetched or executed will be this instruction which is stored here. This is called unconditional branch instruction meaning that whenever you have this B instruction, there will always be a control transfer to this new address right. This is called unconditional branch.

Now you can also have this transfer of control depending on some condition. These are called conditional branch instruction. Now here I have shown one example, I have shown one conditional branch instruction called Branch if Not Equal; BNE stands for Branch if Not Equal. Let us look at this simple code segment, you will understand what this is doing. Here we are initializing some register r2 to 0 MOV 0 to r2 r2 is becoming 0 then some instructions here some calculations are going on. Then we are adding r2 to 1 r2 plus 1 result stored back into r2.

So, r2 is in cream incremented by 1, then we are comparing whether our r 0 this will be r2 sorry this will be r2. So, here you are comparing whether r2 is becoming 20 or not, but if it is not 20 not equal then we go back to loop. We repeat this and once it becomes 20 it is equal to 20, it comes out and it continues with the next instruction which means this is a simple loop we have implemented that will be repeating a set of instructions 20 times. And, you can use this conditional branch instructions to implement this kind of conditional loop.

(Refer Slide Time: 05:42)



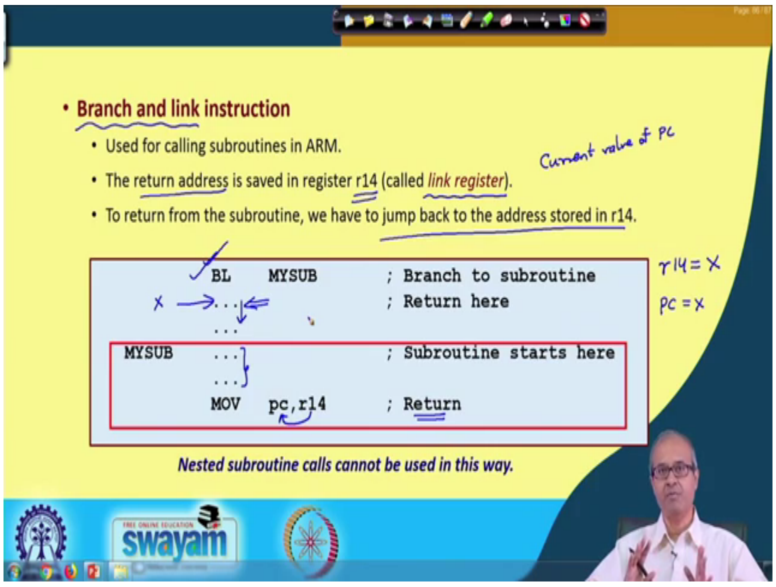
- Branch conditions that are supported:

B, BAL	Unconditional branch
BEQ, BNE	Equal or not equal to zero
BPL, PMI	Result positive or negative
BCC, BCS	Carry set or clear
BVC, BVS	Overflow set or clear
BGT, BGE	Greater than, greater or equal
BLT, BLE	Less than, less or equal

Now, not only branch if not equal a variety of other conditions are possible as this list shows. Unconditional branch you can either use B there is another mnemonic BAL, it means the same thing. Now in terms of the conditional branch, you can have equal or not equal, not equal I have already illustrated. You can have plus or minus branch if plus, branch if minus, branch if carry clear, branch if carry set depending on the carry you can take a jump. Branch if overflow flat clear overflow set; you can check whether there is an overflow or not, branch if greater, branch if greater than or equal branch if less than or branch if less than or equal.

So, you can make some calculations or comparisons based on that you can take your decision whether to jump somewhere else or not depending on all these branch conditions which are supported in r.

(Refer Slide Time: 06:54)



- Branch and link instruction
  - Used for calling subroutines in ARM.
  - The return address is saved in register r14 (called *link register*).
  - To return from the subroutine, we have to jump back to the address stored in r14.

```
        BL     MYSUB          ; Branch to subroutine
        ...                   ; Return here
        ...
MYSUB   ...                   ; Subroutine starts here
        ...
        MOV    pc,r14         ; Return
```

Nested subroutine calls cannot be used in this way.

Now, talking about subroutine calls; well in many computers or instruction set architectures many of them, you may be familiar with you may have seen there is some instruction like call. There is a call instruction which you use to call a subroutine. So, when you call a subroutine normally what happens? The program counter is pushed in the stack you jump to the subroutine, subroutine gets executed, the last instruction of the subroutine will be a return instruction. What the return will do? It will pop the last element from the stack, it will load it into program counter which means you return back to the program from where the call instruction was executed the next instruction, because PC always points to the next instruction right.

So, that is how the things are handled in a conventional processor, but in ARM there is no call instruction rather there is an instruction called branch and link, BL in short. So, what branch and link does? Here there is no stack as I told you in ARM the architectures does not support in stack. There is a special register r14 which is called the link register. So, the return address means return, what is return address? Return address is nothing, but the current value of program counter. So, whenever an instruction is executed the pc is immediately incremented by 4. So, it is pointing to the next instruction that is the return address.

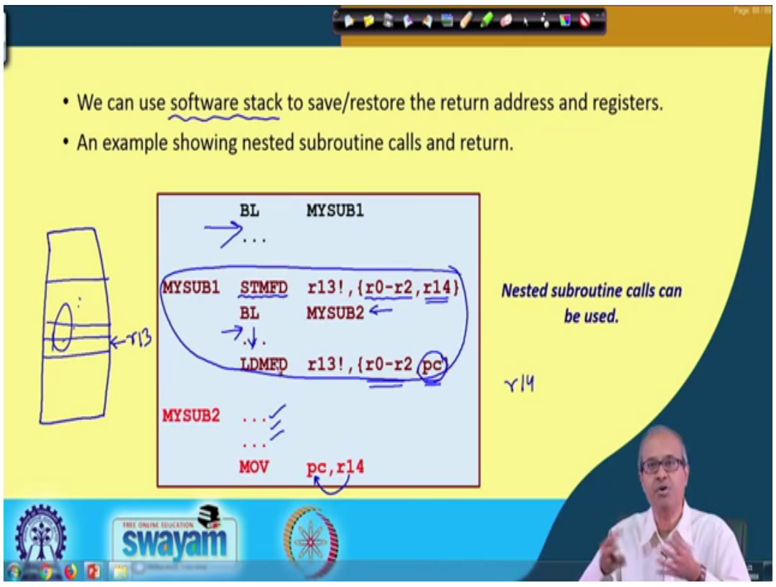
So, the return address will be saved into the link register, it will be transferred to the link register and then you jump to the subroutine branch. And when you are trying to return back so, you will have to jump back to the address which is stored in r14. This is how in ARM subroutine colon return can be handled. Let us take a simple example here; a very small code segment is shown. Suppose there is a program in the program somewhere you have given a subroutine call instruction branch and link BL is the mnemonic for that and this is the subroutine you are calling.

Let us say MYSUB is the level some subroutine you are calling and this red box indicates the body of the subroutine. Now inside this subroutine there will be some instructions, but before jump into the subroutine what will happen? In this link register this r14, this next instruction address let us say this is X, this X will get stored in r14 right. The current program counter if it is X, X will get stored in r14 and then it will be branching to MYSUB. In MYSUB, there will be several instructions, it will execute. At the end when it wants to return back, what it does? It executes some MOV instruction. What kind of MOV? MOV r14 to pc.

So, what will happen here? This r14 contains the value of X. If you load pc with X, now pc will become X. So, when the next instruction will be fetched, it will be fetched from this address X. So, essentially you are branching back or jumping back to this location and you are resuming execution from here ok. This is how you are implementing subroutine call and return in ARM ok. But here it is clearly mentioned here that in this way you cannot implement nested subroutine or recursion which is so, common like you are calling a subroutine A, A is calling A subroutine B, B is calling a subroutine C; you see here you have only one register to store the return address r14.

So, when you are calling A, your return address is stored in r14, but when a is calling B, A s return address will be stored in r14 again. So, the previous r14 value will be overwritten. So, you will never be able to come back to your program right. So, nested subroutine calls will not work in this way, you have to use some other mechanism.

(Refer Slide Time: 12:05)



- We can use software stack to save/restore the return address and registers.
- An example showing nested subroutine calls and return.

```
        BL      MYSUB1
        ...
MYSUB1  STMFD   r13!,{r0-r2,r14}
        BL      MYSUB2
        ..
        LDMFD   r13!,{r0-r2,pc}

MYSUB2  ...
        ...
        MOV     pc,r14
```

Nested subroutine calls can be used.

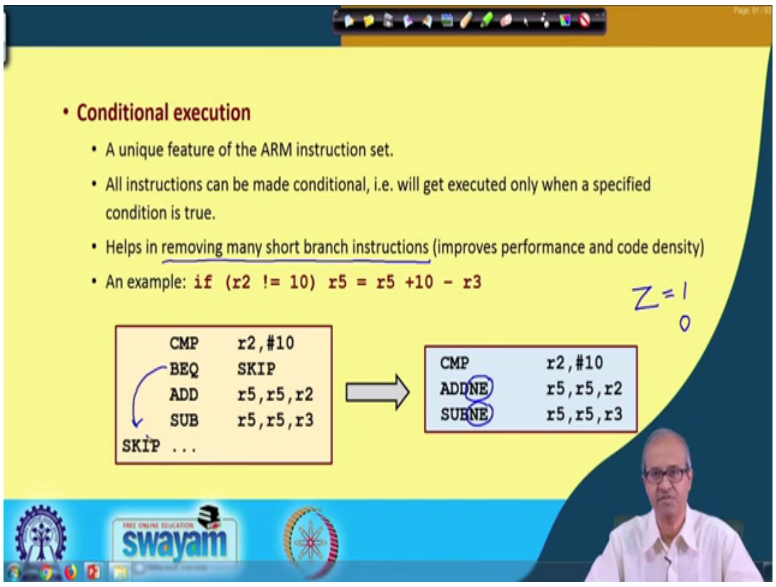
So, here I am showing a simple example again where we are illustrating the use of a software stack. So, I said ARM does not support a stack, but we can implement a stack in software when using our available instructions. Let us assume that we have a memory location we want to implement or stack here this r13. I said r13 is a register which you typically use as the stack pointer. If we want to let us say we are using r13, let us say these are some memory locations and r13 is pointing somewhere let us say r13 is pointing to here and in this program what I am illustrating from my main program, I am making a call branch and link MYSUB1, I am calling MYSUB1and from mice this is MYSUB1 ok.

This is the body of MYSUB1 and from MYSUB1, you see I am again calling another subroutine MYSUB2. This is a nested call two level nesting I am demonstrating. So, in the first level of nesting just to ensure that my r14 value does not get lost, what I am doing you see we are using a multiple register store instruction STMFD. So, what it does? It essentially saves r14 and some other registers which I may be needing I want to also say let us say r 0 r1 r2. These three and also this r14 link register all these 4 registers are stored in r13 stack here; they are stored out here right. Then you branch to MYSUB2 branch and link.

So, now your r14 gets overwritten, now it will contain the return address of this instruction. Let it be, let MYSUB2 gets executed and once it is done you return back to the address stored in r14. So, you return back here and resume execution from here. Now at the end well you have a matching load multiple register load instruction, what you do? You load again from r13; this r13 is pointing there. Here your loading means again r 0 to r2 and not r14 you are giving pc. So, the last file will be loaded to pc.

So, whatever this r14 value was saved in the stack that will now get loaded in pc which means it will automatically return back here. So, this kind of a multiple register load and store instruction you can use to implement this kind of nested subroutine call all or even recursion a subroutine calling itself. Just I am giving you an idea this multiple register load store instructions are very useful to save and restore status in stack when you are implementing subroutine calls in a general way.

(Refer Slide Time: 15:41)



- Conditional execution
  - A unique feature of the ARM instruction set.
  - All instructions can be made conditional, i.e. will get executed only when a specified condition is true.
  - Helps in removing many short branch instructions (improves performance and code density)
  - An example: `if (r2 != 10) r5 = r5 +10 - r3`

```
      CMP    r2,#10
      BEQ    SKIP
      ADD    r5,r5,r2
      SUB    r5,r5,r3
SKIP  ...
```

```
CMP      r2,#10
ADDNE    r5,r5,r2
SUBNE    r5,r5,r3
```

Fine, now a very important feature which this ARM instruction set has this already we have mentioned earlier is something called conditional execution. This is a unique feature which we normally do not see in other instruction sets. The designers for ARM have very carefully thought out these set of instructions. This says that the instructions can be made conditional; conditional means they may either execute or they may not execute. I can specify some condition well along with every instruction there will be a field where some condition is specified if this condition holds, then it is executed; otherwise it is not executed right.

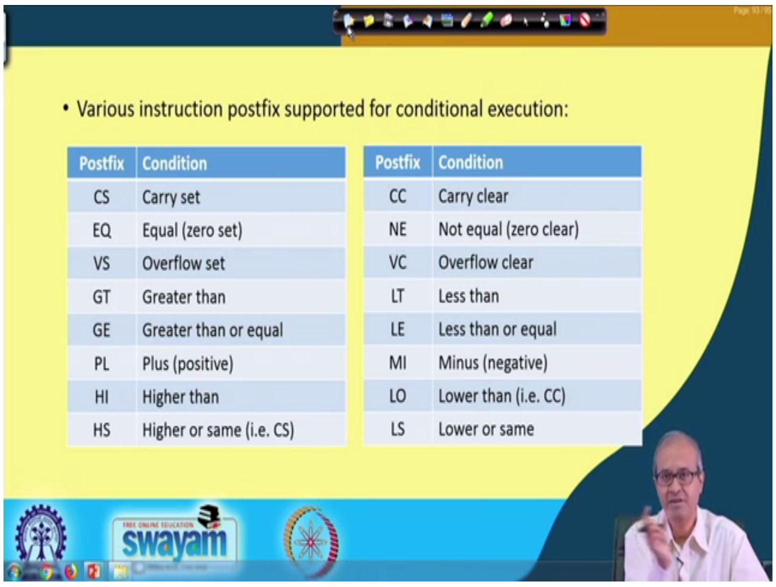
So, let us take an example. Suppose in high level language, I want to implement something like this. So, I am specifying in a high level C like syntax if the value of r2 is not equal to 10, then I want to do this calculation r 5 equal to r5 plus 10 minus r3. Suppose I want to do this. Here I have a solution just a second. Suppose this one I implement like this yeah I implement like this let us first see this implementation. So, what I do? I have to check r2 with 10, whether it is equal to 0 or not equal to 0.

So, my requirement says if it is not equal to 0, then do this calculation. So, what I am doing? I am comparing r2 with 10, here I am comparing. If it is equal then I am not supposed to do this right; if it is equal I am jumping I am branching to this skip. I am skipping these two instructions right, but if it is not equal then I am going through this add and subtract I am first adding r2 is 10, 10 to r5 then I am subtracting r3 from that right. Now see this implement is fine, there is no problem here.

But the only thing that I want to say is that here, I am requiring one branch instruction, but if I use the conditional variety of implementation like this; you see what we have done here we are again comparing r2 with 10. The next add and subtract instructions what we have done, we have added with the conditional postfix not equal to; that means, if not equal to then you do add if not equal to then you do sub. So, what is me not equal to? This 0 flag is tested the compare instruction will be will be setting or resetting the 0 flag. If 0 flag is 1, it means r2 was equal to 10. If 0 flag is 0 which means r2 was not equal to 10.

So, when I say ADDNE, the computer is actually testing the 0 flag; if it is 0 then execute this; similarly subtract NE if the 0 flag is 0, then execute this right. So, this conditional variety of instructions as you can see helps in avoiding one branch instruction. It is mentioned here helps in removing many short branch short means you are branching over a very short distance skipping a few instructions. So, in a general code you will find many such instances. These conditional instructions can help in eliminating many such branch instructions make your code shorter and more efficient right. This is the main advantage.

(Refer Slide Time: 20:18)

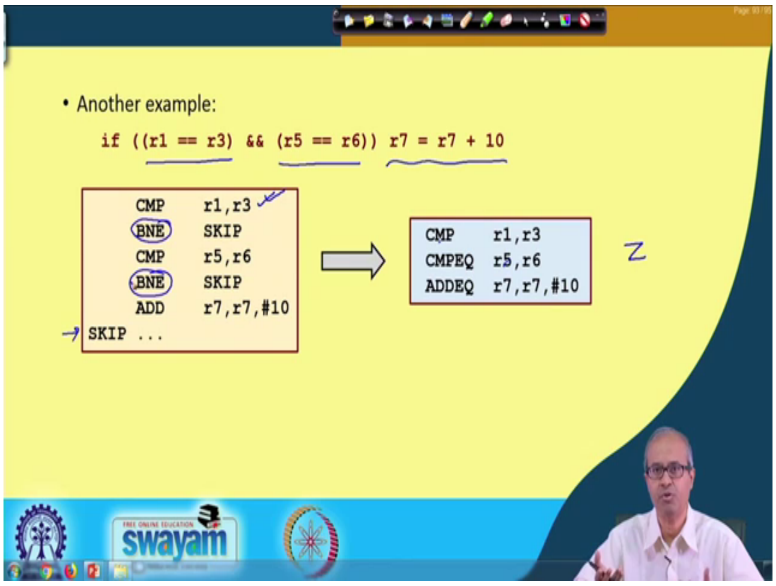


- Various instruction postfix supported for conditional execution:

Postfix	Condition	Postfix	Condition
CS	Carry set	CC	Carry clear
EQ	Equal (zero set)	NE	Not equal (zero clear)
VS	Overflow set	VC	Overflow clear
GT	Greater than	LT	Less than
GE	Greater than or equal	LE	Less than or equal
PL	Plus (positive)	MI	Minus (negative)
HI	Higher than	LO	Lower than (i.e. CC)
HS	Higher or same (i.e. CS)	LS	Lower or same

Now the various instruction postfix that are available in the early example I have showed the any version not equal to. But you can use so, many variants Carry set, Equal, Overflow set, Greater than, Greater equal, Plus, High higher on same. So, many options are there. So, along with an instruction you can specify what condition, you want to specify and the instruction will be executing depending on the validity of that condition right. You can see how many condition there 4, 5, 6, 7, 8 16 are there which means as part of the instruction there must be 4 bits reserved somewhere which will be specifying this condition which condition fine.

(Refer Slide Time: 21:13)



- Another example:

```
if ((r1 == r3) && (r5 == r6)) r7 = r7 + 10
```

```
       CMP    r1,r3
       BNE    SKIP
       CMP    r5,r6
       BNE    SKIP
       ADD    r7,r7,#10
SKIP ...
```

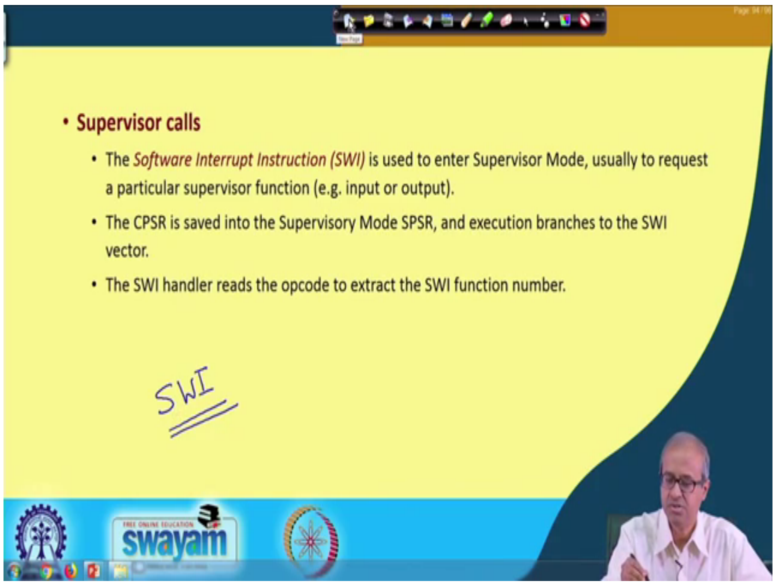
```
CMP     r1,r3
CMPEQ   r5,r6
ADDEQ   r7,r7,#10
```

Let us take another example. Here we are checking for two conditions together let us say I have a case like this. If r1 equal to r3 and r5 equal to r6. Then you do some calculation. This is the conventional approach, you first compare r1 and r3; if they are not equal you can straight away skip; that means, you have you do not have to do it. If it is not equal to straight away come to skip, then you check the next condition r5 an r6. If they are not equal then you go to skip otherwise you add.

But in the conditional variety, the second compare instruction also you can make conditional like first one is compared r1 r3 then compare if equal r5 r6; that means, if the first comparison resulted in equality then only you execute this compare. And finally, this add equal to will generate; that means, if the second one also generates equality then only add will be executed because the 0 flag will be set accordingly.

So, you see here you are eliminating two branch instructions ok. So, your code is becoming so much more efficient. So, this ARM instruction set the addition of this conditional execution of instructions, this has added a new dimension to the way people can write codes. So, the code can be very compact very efficient and in terms of execution time will also take less time because there are fewer number of instructions all right.

(Refer Slide Time: 23:14)



- Supervisor calls
  - The *Software Interrupt Instruction (SWI)* is used to enter Supervisor Mode, usually to request a particular supervisor function (e.g. input or output).
  - The CPSR is saved into the Supervisory Mode SPSR, and execution branches to the SWI vector.
  - The SWI handler reads the opcode to extract the SWI function number.

There is another kind of an instruction which also changes the value of pc of course, it is not so much important from the point of view of embedded system design. These are these are called supervisory calls. There is an instruction called software interrupt is SWI, this is like a subroutine call. This will also jump to a subroutine by saving the return address in some register r14, but the main difference is that while doing this jump it will also change the processor mode to supervisor mode because these instructions are typically used in environments where there is an operating system and you are trying to request some service from the operating system. You given is SWI call so, it jumps to the OS and also the mode changes to supervisory mode. Normally when the OS executes, the processor mode is supervisory that automatically happens right. So, we are not going into too much detail of this it is not required.

So, we have seen in a nutshell, the various kinds of instructions which are there in ARM and the unique features in particular I have talked about the conditional execution in this lecture. So, earlier you have seen in terms of the arithmetic and logic instructions when you are executing, the second operand can be specified in so many flexible ways. So, there are many ways in which you can make your code shorter by taking advantage of these flexible methods then the multiple register transfer instructions and so on.

So, as I said that in this course we shall be looking at a lot of experimental demonstrations based on some microcontroller boards. Now in the next lecture we shall be introducing you to one of the boards based on which many of the experiments shall be demonstrated. We shall be discussing the basic features of the boards the different kind of connectors and how to use them and subsequently we shall be seeing actually how they are put to use ok. So, we shall be seeing them in our next lectures.

Thank you.