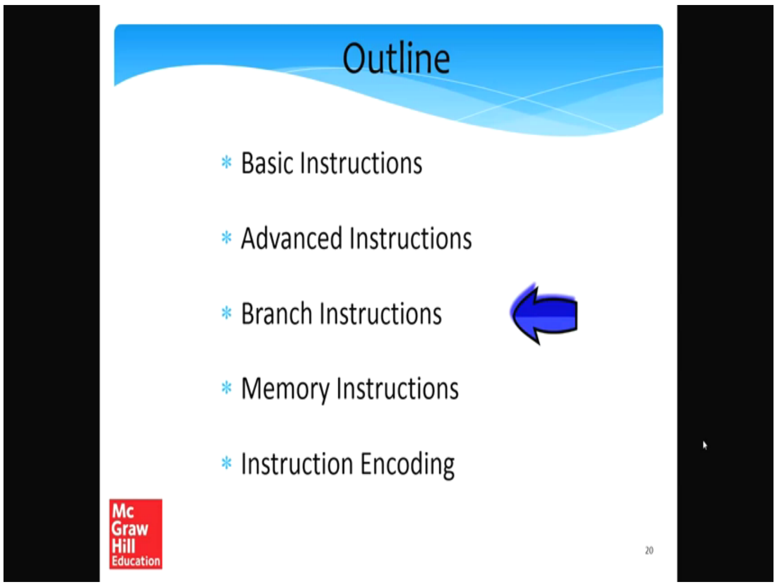
Computer Architecture

Prof. Smruti Ranjan Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 09
ARM Assembly Language Part-II

(Refer Slide Time: 00:25)



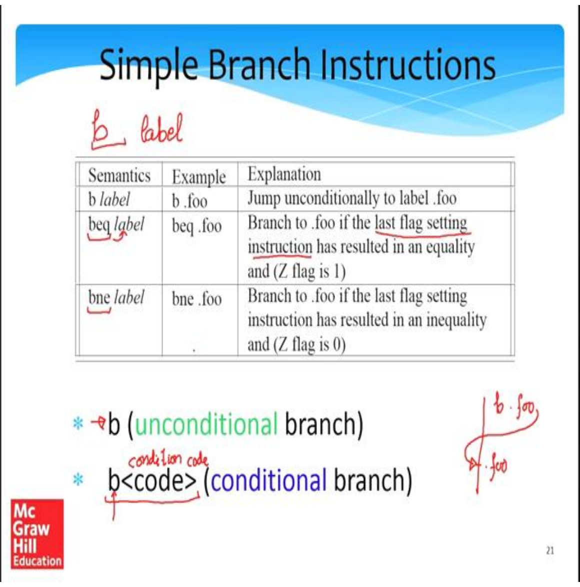
Outline

- Basic Instructions
- Advanced Instructions
- Branch Instructions
- Memory Instructions
- Instruction Encoding

Mc Graw Hill Education
20

Now, let us move to the more interesting part of the ARM instructions set that deals with branch instructions. We need branch instructions to implement statements for loops, while loops and almost any kind of program where we need to make a decision on the value of a certain variable.

(Refer Slide Time: 00:44)



Simple Branch Instructions

b label

Semantics	Example	Explanation
b *label*	b .foo	Jump unconditionally to label .foo
beq *label*	beq .foo	Branch to .foo if the last flag setting instruction has resulted in an equality and (Z flag is 1)
bne *label*	bne .foo	Branch to .foo if the last flag setting instruction has resulted in an inequality and (Z flag is 0)

- b (unconditional branch)
- b<code> (conditional branch)

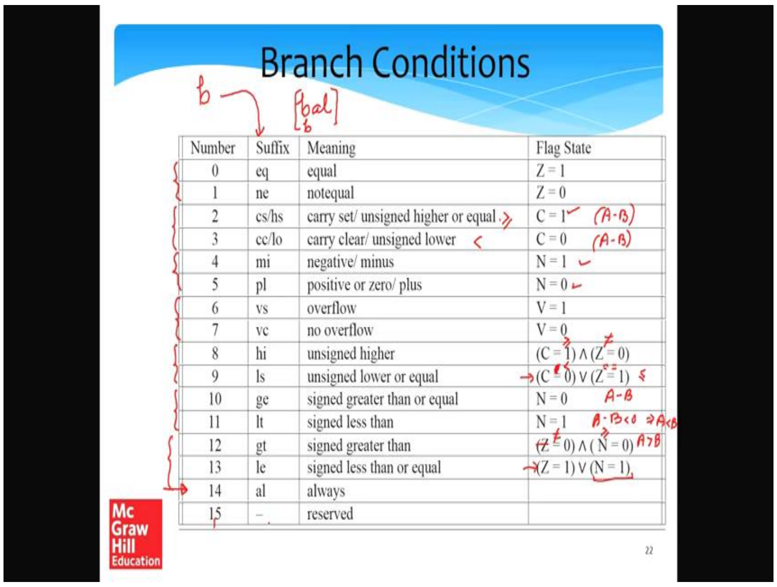
So, ARM like simple RISC suppose some very simple branch instructions, there are some complicated ones as well, but let us take a look at the simple ones. So, similar to simple RISC, the format of every branch statement is the same which is essentially a branch instruction and a label. So, we branch to the label, we jump to the label, and the label is the first instruction at the branch target.

So, let us see if this is a program and so there is a certain label dot foo over here, and we branch to dot foo. What this means is that the control gets transfer from here to here. So, the basic and the simplest instruction is an unconditional branch b. So, an example would be let us say b dot foo, it would jump unconditionally to label dot foo. Similar, to simple RISC, we have b e q branch is the flags indicate equality and branch if the flags indicate inequality. So, what I mean by flags is at the last instruction that set the flags should have had an equal to should have concluded that both the values that are been given to it are equal only on that condition with this particular branch actually branch to this label.

So, the explanation is that you branch to dot foo, if the last flag setting instructions mind you this is one phrase flag setting instruction has resulted in an equality and this will show up in the Z flag being equal to 1. Similarly, we will have branch not equal to which is just the opposite of branch if equal to where the Z flag is equal to 0. So, the fantastic thing in ARM is that we have many, many branch instructions and all of them have a standard format. So, we have b first and then a branch condition code.

So, this is the format of the branch instruction there if first have the letter b, and then we have a conditional code for the branch which so one example of this should be b e q where e q is a condition code equality and the other would be anywhere any is the condition code for lack of equality. So, we have many, many such branches in condition code. So, we have actually fifteen such condition codes that we shall see in the next slide.

(Refer Slide Time: 03:48)



Branch Conditions

b [bal] [b]

Number	Suffix	Meaning	Flag State
0	eq	equal	Z = 1
1	ne	notequal	Z = 0
2	cs/hs	carry set/ unsigned higher or equal $\geq$	C = 1 (A-B)
3	cc/lo	carry clear/ unsigned lower $<$	C = 0 (A-B)
4	mi	negative/ minus	N = 1
5	pl	positive or zero/ plus	N = 0
6	vs	overflow	V = 1
7	vc	no overflow	V = 0
8	hi	unsigned higher	$(C \overset{\geq}{=} 1) \wedge (Z \overset{\neq}{=} 0)$
9	ls	unsigned lower or equal	$(C = 0) \vee (Z = 1)$ $\leq$
10	ge	signed greater than or equal	N = 0 A-B
11	lt	signed less than	N = 1 $A-B<0 \Rightarrow A<B$
12	gt	signed greater than	$(Z \neq 0) \wedge (N = 0)$ $A>B$
13	le	signed less than or equal	$(Z = 1) \vee (N = 1)$
14	al	always	
15	–	reserved	

22

So, first let us take a look at the condition codes that we have seen. So, mind you all of these are suffixes. So, we will have b plus this suffix. So, it can b e q, b n e or b g e or b g t any of these suffixes are allowed. So, let us take a look at the first two, which we have already seen b e q and b n e. So, the first equality is when the zero flag has been set to one by the last instruction that sets the flags which we called the flag setting instruction. Similarly, the any flag not equal to flag is just the opposite of this. Now, let us take a look at one more pair of conditions which were very interesting.

So, let us consider h s, h s means unsigned higher or equal. So, what we do is that when we take two 32-bit numbers, we discard the sign bit and think of them as unsigned numbers and we compare them. So, the way that we would compare any in unsigned number a and unsigned number b is that we will subtract b from a right. So, basically given any two numbers what we would do to compare them is that we will subtract a minus b. If A minus B, A is greater than B which means it is unsigned higher or equal then there will be no borrow. So, the borrow will be false as a result the carry bit will be equal to 1. So, this is pretty much the condition that the hardware checks that is the carry bit equal to 1, in this case this particular condition is true.

So, this c s or h s the opposite of h s is lo. So, in this case, this is the first one or unsigned higher or equal. So, basically greater or equal in an unsigned sense, the opposite would be less then. So, in this case, the carry bit needs to be 0 or alternatively when we do a subtraction of the from a minus b, we actually generate a borrow. And since a carry and the borrow have a rivers sense in this case the carry bit will be equal to 0.

So, there are again another pair of conditions m i and p l which means that a minus, so basically when we subtract it, the last result that came had a minus sign. So, in this case, the n then negative bit would be set to 1. So, this essentially says that the last value or the last result that the flag setting instruction considered had a minus sign and so the m i suffix condition over them be true. So, the opposite of m i is p l which is positive or 0, so the condition is also exactly opposite.

Let us again take a look at the next pair of conditions v s and v c. So, v s is the case if an over flow occurred and over flow is a condition, where it basically means that the final result could not fit within the number of bits that are available. So, consider a 32-bit system, but let say if we multiply you know the product of a multiplication requires more bits more than 32. So, in that case, what needs to be done is that you know we either allocate more bits or we single in over flow. So, over flow in this case would mean that the result cannot fit within 32-bits. So, the over flow flag will be 1; and in the case of no over flow, the over flow flag will be 0. Now, let us consider a different variant of h s and lo, so this was mind you unsigned higher or equal this is just unsigned higher.

So, in the case of unsigned higher you will have this condition c equals 1. So, this indicates unsigned higher or equal, but along with that will also check that the 0 flag is equal to 0, which means it is also unequal. So, if let say A is greater than equal to B, and a is not equal to b, then we can automatically infer that A is greater than B. Similarly, we can have unsigned lower or equal which is just opposite of this. So, in this case, we will also take this condition C equals 0, which means it is unsigned lower or equal. And also you know ensure have an OR condition that it is either less then equal to or it is just equal to oops, I am sorry, this is a mistake let me see, let me just sorry.

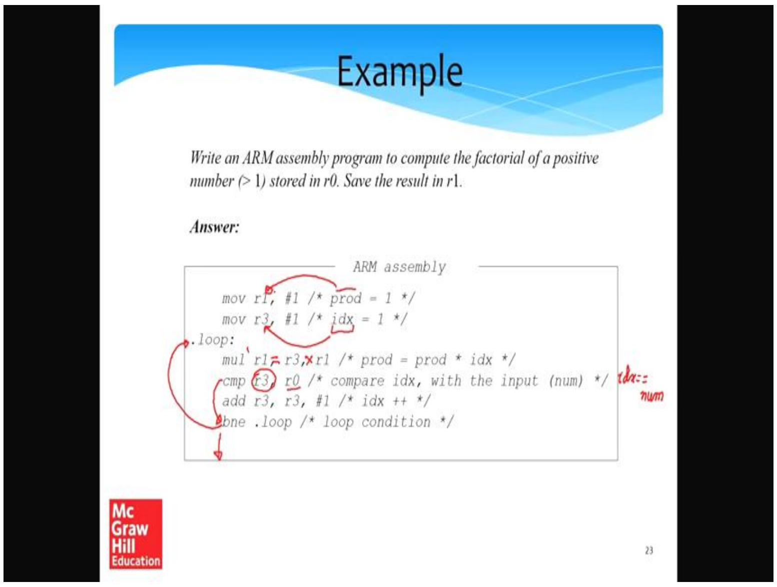
So, C equal to 0 denotes that number A is strictly less than B. So, in this case, unsigned lower, so C equal to 0 means that if both the numbers are A and B, A is less than B. So, C equal to zero or. So, this is either one conditions or it is strictly equal. So, if I combine the conditions it is less than or equal we have the condition less then equal to which is exactly what it says that the ls suffix or condition means that the numbers are that you know if I am comparing A and B, A is unsigned lower or equal.

Now, let us consider some signed operations. So, g e is sign greater than or equal. So, in this case, I do the same thing it interpret both the 32-bit numbers are signed numbers and I perform a subtraction. If the final result is not negative then we can say that the A is greater than equal to B. And in the final result is negative, we can conclude that A is less than B. So, this is same as saying that A minus B is less than 0, which implies the implication here is that if A minus B is less than 0 this means that a is less than B.

So, similarly let us consider g t and l e which is similar to the unsigned operands h i and l s that we introduced. So, g t means it is signed greater than. So, this alternatively means where the z flag is 0 is the z flag is 0 means that the numbers are not equal, and n equals 0, which means that the number are greater than an equal. So, the same logic wholes if A is greater than equal to B, and A is not equal to B then we can automatically infer that A is greater than B. And the same logic holds where signed less then or equals if you consider this expression and this expression, they are equivalent.

So, the 0 flag is needs to be 1, which means the either there is equality or there is less then and this is the condition for an unsigned less than, and this is the condition for signed less than. This condition is actually a default as a result it is typically not required which means always. So, this is same as an unconditional branch. So, in the sense b a l is also the same as b which means branch all the time. And the last suffix 15 is actually not used it is reserved such that you know it can be used later with the instruction set is expanded.

(Refer Slide Time: 11:55)



Example

Write an ARM assembly program to compute the factorial of a positive number (> 1) stored in r0. Save the result in r1.

Answer:

```
                 ARM assembly
    mov r1, #1 /* prod = 1 */
    mov r3, #1 /* idx = 1 */
.loop:
    mul r1, r3, r1 /* prod = prod * idx */
    cmp r3, r0 /* compare idx, with the input (num) */
    add r3, r3, #1 /* idx ++ */
    bne .loop /* loop condition */
```

idx::
num
Mc Graw Hill Education
23

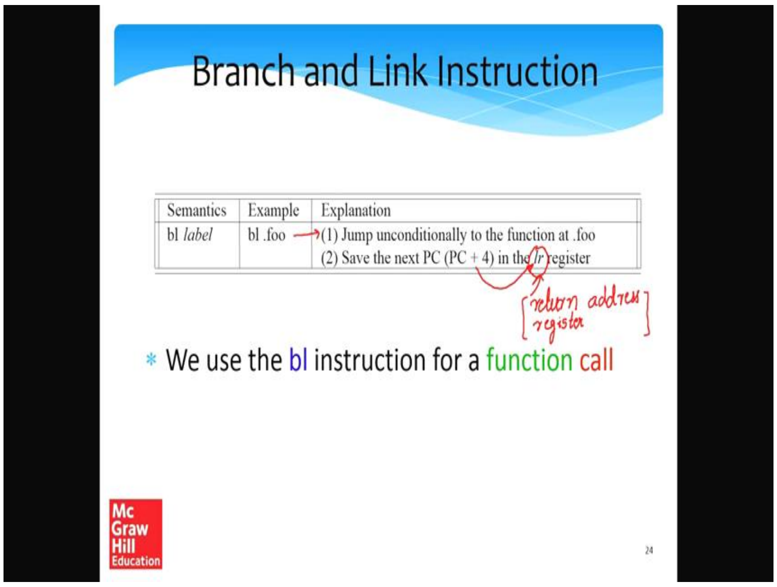
So, let us consider a simple program first that is may be the most important. So, let us write an ARM assembly program to compute the factorial of a positive number. Assume the number is large enough greater than 1. The number is stored in r 0 and save the result in r 1. So, let me give you 10 seconds to look at this particular program. So, the first thing that needs to be done in any kind of an assembly program is that we need to do register assignment.

So, let us assume that the prod variable product is saved in r 1 which is also were we need to written the result is initialized to 1, and the index the index of the loop is also initialized to 1, and it saved in r 3. So, let us do one thing let us starts since we know that the number is greater than 1, we can happily start with the multiplication and check later. So, let us first do the multiplication which is prod equal prod times i d x. So, in this case, I multiply the index which is an r 3 with the current product in r 1 and save it in r 1.

Then I compare the index with the input number. So, the index is an r 3 and the input is an r 0. So, I just compare them I compare i d x with the input with r 0. And then what I do is that a increment the index. So, a increment index I do i d x is plus plus. So, this particular point, let us assume that the index is equal to num, let us assume that we infer an equality and the index is equal to num. So, since the index is already been multiplied into the product we do not need to go for another iteration of the loop.

So, what we can do is that we can compare if this particular comparison, let to an equality or not, because mind you the add instruction is not setting the flags. So, we will only use the result of this particular comparison. And if this comparison towards indicates that the index is not equal to the number then we need to go back to dot loop once again and loop for one more time. Otherwise, if we have reach the end of the iteration, when index becomes equal to the original numbers stored in r 0, we can exit and the product is stored in r 1.

(Refer Slide Time: 14:51)



Branch and Link Instruction

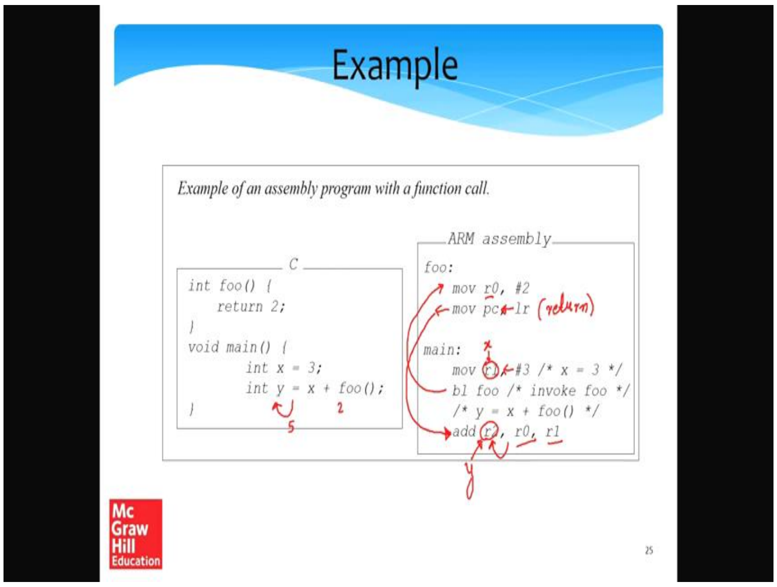
Semantics	Example	Explanation
bl *label*	bl .foo	(1) Jump unconditionally to the function at .foo (2) Save the next PC (PC + 4) in the lr register

[return address register]

* We use the bl instruction for a function call

So, now let us now that we have seen a very basic program, let us try to make our repertoire of assembly instructions slightly more complicated by introducing the branch and link instruction. So, the branch and link instruction yeah is the same as not the same, but similar to the call instruction in simple RISC, we use it to invoke functions to call functions. So, it works on exactly the same way we jumped to a label, and the label points to the first instruction in the function. So, it does the bl instruction branch and link does two things. We jump unconditionally to the function and dot foo, and we save the next PC which is the current PC plus 4 in the return address register which is also called lr, which is actually called the l r register in arm. So, this is the return address register. This particular register over here is the written address register and it is also called the link register. So, this registered will contain the value of the current PC plus 4.

(Refer Slide Time: 16:27)



Example

Example of an assembly program with a function call.

C
```
int foo() {
    return 2;
}
void main() {
        int x = 3;
        int y = x + foo();
}
```

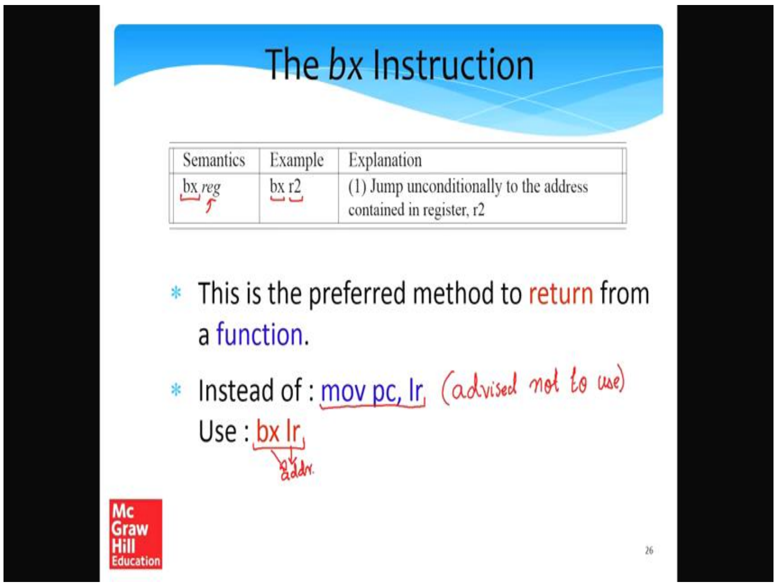
ARM assembly
```
foo:
    mov r0, #2
    mov pc, lr
main:
    mov r1, #3 /* x = 3 */
    bl foo /* invoke foo */
    /* y = x + foo() */
    add r2, r0, r1
```

So, let us write a simple assembly program with a function calls, let us considered a simple C program first. So, let us have a variable x that we set to 3, then we call the foo function that returns 2. So, we add x plus the return value of foo which is 2. So, 3 plus 2, we get 5, which will be again be saved in variable y. So, let us consider what needs to be done here. So, let us the first step is always mapping the variables to registers. So, let us map x to r 1, let us add 3 to it, I am sorry let us move 3 to it; subsequently we call the foo function. So, the foo function moves 2 to r 0 and so we need to return. So, the interesting thing is that in arm, there is actually no return function. So, this is the way that we typically return well there are few more ways also, this is one of the simpler ways.

So, what we do is that since the program counter and the link registered which is the return address register or explicitly visible to the programmer, what we do is that we set the contents of the lr to the PC which is exactly the same as returning because lr contains a pointer to the next instruction which is over here. So, if you move the contents of the return address register the lr registered to the program counter it will be the same as jumping to the last instruction in the main function. I am sorry I should not saying the last instructions is the instructions just after the function call in the main function.

So, in this case r 0 contains two. So, we are adding to r 0 with r 1, essentially we are adding the return value of the function with r 1 and saving it in r 2. With assumption is that the variable y is map to r 2. So, what is the important learning that we get from this slide, what we get to learn is that the link registered which contains the return address is mapped you know, if we transfer the contents of that to the PC which is also one of the registers that is exposed in a register file. This is the same as returning from a function, exactly the same. So, we do not need a that instruction.

(Refer Slide Time: 19:00)



The *bx* Instruction

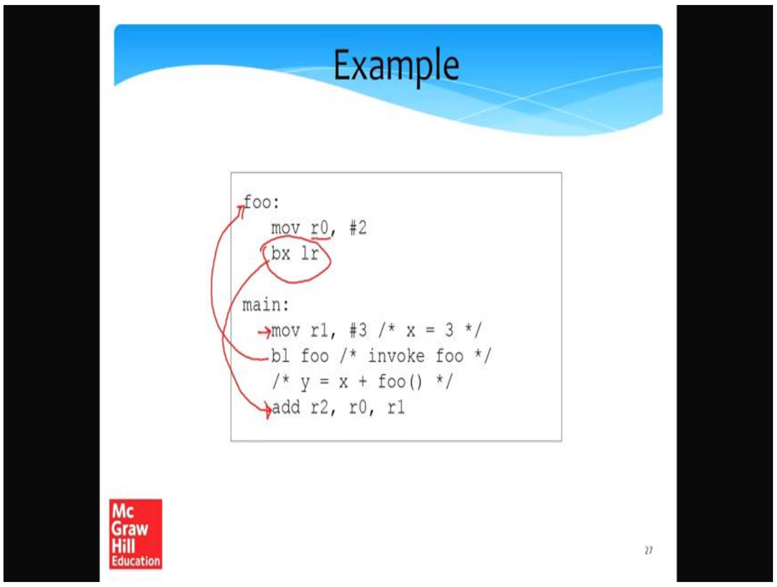
Semantics	Example	Explanation
bx *reg*	bx r2	(1) Jump unconditionally to the address contained in register, r2

- This is the preferred method to return from a function.
- Instead of : mov pc, lr, (advised not to use)
  Use : bx lr

So, let us not take a look at some another kind of instruction which is slightly easier, and now it is programmers are preferring it. So, this particular instruction is called bx. So, bx it takes the single argument a single register. So, the explanation is that lets say I have an instruction on the form bx register r 2 this means that I jump unconditionally to the address contained register r 2. So, I can think of this as a variant of the b instruction unconditional branch, but the b instruction actually took a label and the bs instruction takes a register.

So, nowadays this is the preferred method to return from a function and we actually do not use move PC lr, we are advised not to use this thing let me write it over here advised not to use, well this should be used in variant some ARM processor that do not provide the b x instruction. But if the b x instructions provided which is the case an at least most ARM processors what we are supposed to do is bx lr, which basically means that take the address that is there inside this link register and jump to it that is basically the idea say. Instead removing lr into PC, which is what we saw in the previous slide a better idea much, much better idea is to use bx lr where we jump directly to the contents of the lr register.

(Refer Slide Time: 20:45)



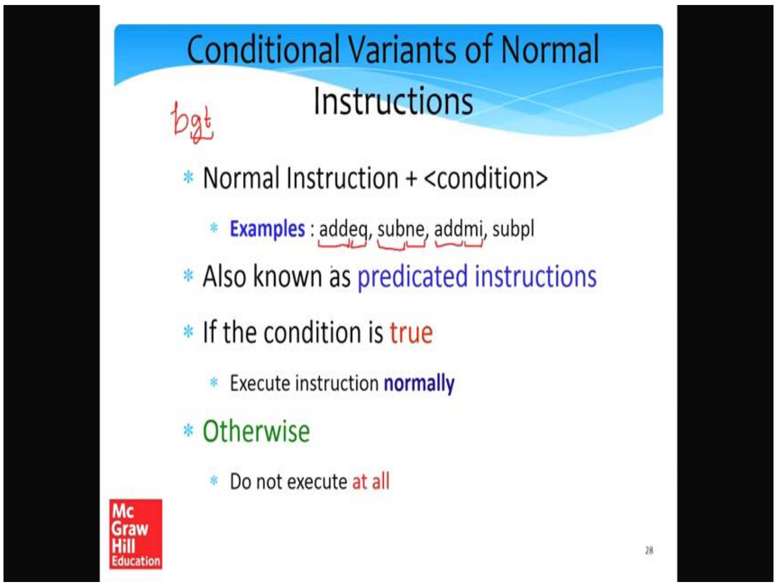
Example

```
foo:
   mov r0, #2
   bx lr

main:
   mov r1, #3 /* x = 3 */
   bl foo /* invoke foo */
   /* y = x + foo() */
   add r2, r0, r1
```

So, let us consider the same program with the bx instruction. So, we have the first instruction mov r 1 hash 3 sets x to 3. We call the foo function using the b l instruction branch and link see the lr register the return address get saved. We move 2 to r 0 and instead of moving the contents of lr to PC, this is the new instruction bx lr and in this case we move, we branch to the point to the address that is contained inside lr which is the address of the next instruction after the function call.

(Refer Slide Time: 21:27)



Conditional Variants of Normal Instructions

bgt

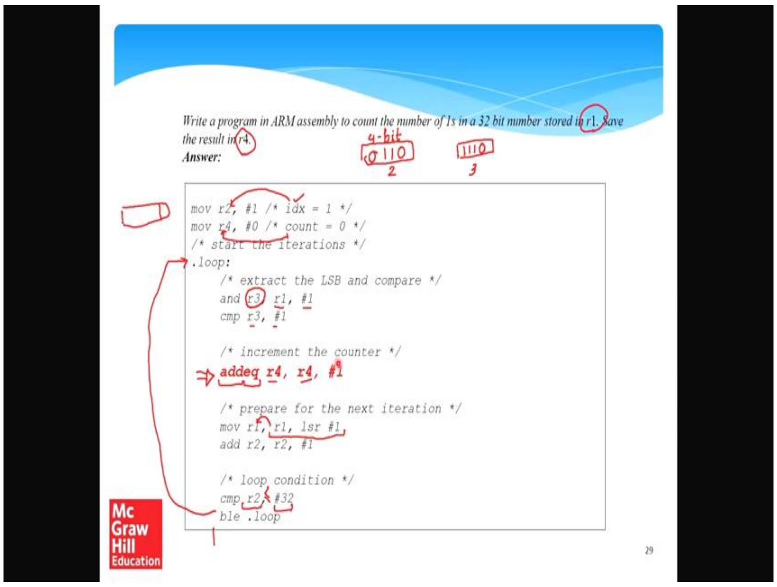
- Normal Instruction + <condition>
  - **Examples** : $\underbrace{\text{add}}\underbrace{\text{eq}}$, $\underbrace{\text{sub}}\underbrace{\text{ne}}$, $\underbrace{\text{add}}\underbrace{\text{mi}}$, subpl
- Also known as predicated instructions
- If the condition is true
  - Execute instruction **normally**
- Otherwise
  - Do not execute at all

28

So, let us now look at some conditional variants of normal instructions. So, it is just not the case that these conditions eq, ne, gt all the conditions that we studied all the fourteen condition other than the al always condition right. So, basically all the fourteen conditions that we looked that it is not necessary that they will only be put after a branch. So, previously what were we doing, we were putting them after a branch after that b instruction, it is not necessary. After a lot of normal instructions like add, subtract, multiply, we can suffix the conditions as well.

So, for example, we can have add then we can have the condition e q or we can have add and we can have the condition m i. This will basically mean that the add instruction will only execute if the m i condition holds true or the subtract instruction will only execute if the ne not equal condition holds true. So, these kind of instructions are also known as predicated instructions, which means there if the condition is true, the instruction will execute normally otherwise the instruction will not execute at all it is pretty much like a ARM.

(Refer Slide Time: 22:46)



Write a program in ARM assembly to count the number of 1s in a 32 bit number stored in r1. Save the result in r4.

Answer:

```
mov r2, #1 /* idx = 1 */
mov r4, #0 /* count = 0 */
/* start the iterations */
.loop:
    /* extract the LSB and compare */
    and r3, r1, #1
    cmp r3, #1

    /* increment the counter */
    addeq r4, r4, #1

    /* prepare for the next iteration */
    mov r1, r1, lsr #1
    add r2, r2, #1

    /* loop condition */
    cmp r2, #32
    ble .loop
```

29

So, let us take a look at a slightly more complicated example. So, let us write a program in ARM assembly to count the number of once in a 32-bit number, which is stored in r 1 and let us save the result in r 4. So, let us to let me consider a 4 bit number. So, consider a 4-bit number consider a number of 1's, the number of 1's are 1 and 2. Consider another 4-bit number consider count the number of 1's there are three 1's. So, similarly we will have to do that in a 32-bit number, we need to count the number of ones that are there in it is binary representation of course, and we need to save the result in r 4. So, let us move ahead. So, we want to have a loop. So, what is the idea, the idea is that in a 32-bit number, each of these bits needs to be considered one after the other, we need to check if it is one or not and increment a counter.

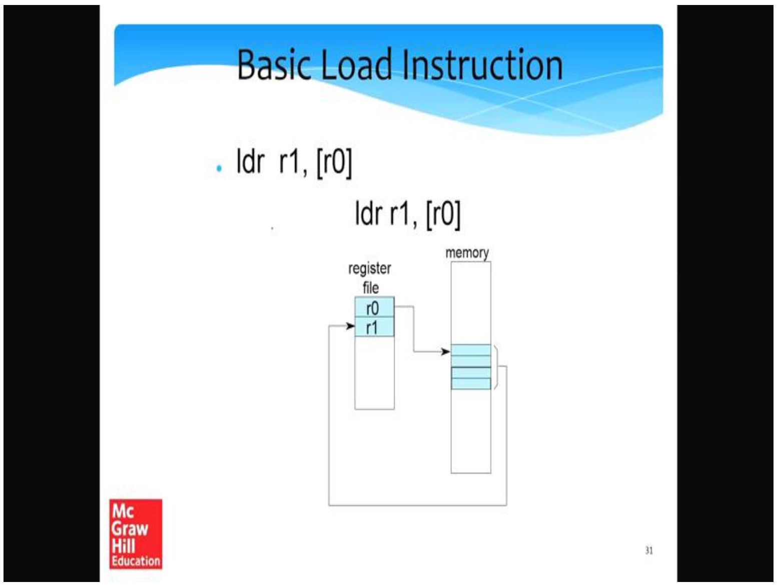
So, essentially if there is a loop we need an index. So, let us have an index variable, let us map it to r 2 initialize it to 1. Similarly, let us have a count variable, which is map to r 4, because the final result needs to be saved in r 4. So, let us take a count variable save it in r 4. And initially the count starts with 0, because we are not seen anyone yet. So, the first task would be extract the least significant bit and compare, right extract the lsb and compare. So, in this case, how do we extract the lsb? So, what we can do is that let us take the original number which is stored in r 1 and lets AND it with one. So, in this case, all the upper 31-bits will get erased and only the l s b will remain. So, in this case, the result of the AND we will stay in r 3.

Let us now compare r 3 with 1. Here is the new instruction that we insert over here, and it is very important to understand this part that only if the comparison results in an equality or alternatively only if the lsb is equal to 1, do we increment the count which is an r 4 by 1. So, only if the lsb is equal to 1, we increment the count by 1; otherwise, this instruction works as if it has just never been seen or never executed. So, in this case, we are essentially counting if this lsb is 1; if it is 1, we increment r 4.

Now let us prepare for the next iteration. So, what we do now is that we take the number r 1, and we do a logical shift right, mind you not an arithmetic shift right because in this case we do not want to replicate the sign bit. So, we shift it right by one position and we save it back in r 1; and simultaneously that we increment the index, so r 2 incremented with 1, then once the index reaches 32, so I am sorry. So, then we compare. So, we start with the index of 1, so basically one to 32 times we need to loop, but once it is thirty three we need to get out. So, we compare r 2 with 32, if r 2 is less than equal to 32 b l e then we go back to the original looping instruction, otherwise we come out and the final result is saved in r 4 which is the count.

So, the important point to understand for all of us is this add e q instruction which made our life extremely easy and the add e q instruction pretty much otherwise we would have had to have a branch we will have to you know after this point if the comparison is successful, we had to add a branch statement where we either incremented or do not decrement r 4. But that entire piece of code was folder into this one single instruction which makes our coding process very, very efficient. Now that we have seen basic instructions, advanced instructions and branch instructions, let us move to memory instructions.

(Refer Slide Time: 27:33)

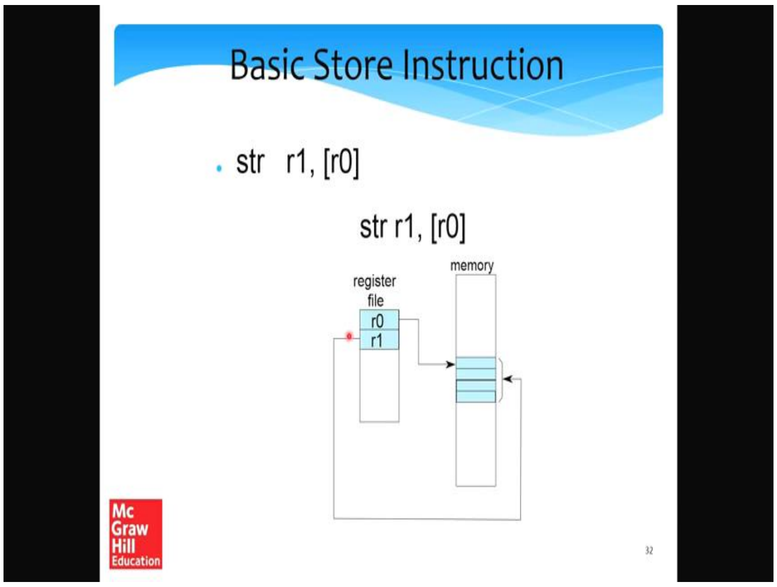


Basic Load Instruction

- ldr r1, [r0]

So, similar to simple RISC, ARM has a very basic load instruction which is load r 1 with a register indirect operant r 0. So, in this case, the address - the memory address is stored in r 0 we read it we access memory read 4 bytes and save it in r 1, exactly the same as simple RISC.

(Refer Slide Time: 27:56)

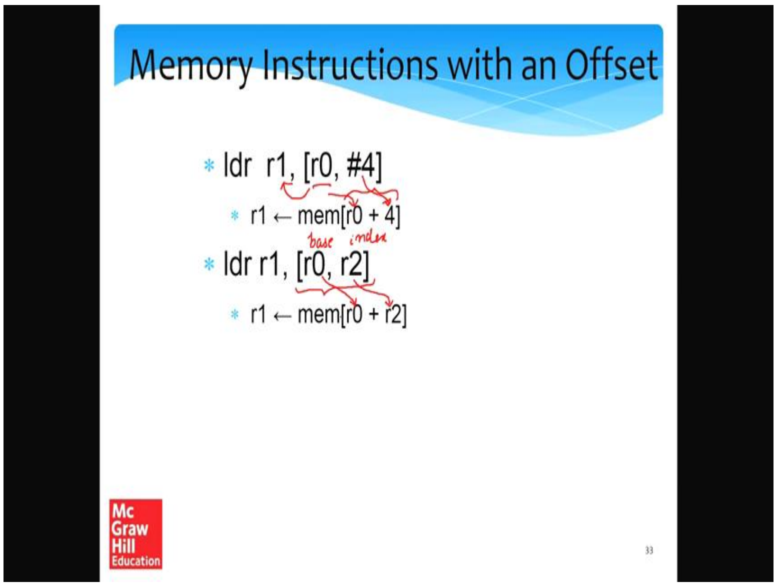


Basic Store Instruction

- str r1, [r0]

And similarly, the store instruction works in a very similar fashion here the idea is that instead of reading from memory, we write to memory the address is contained in r 0. So, we use the address to access memory then we take the contents of r 1 and store it in memory.

(Refer Slide Time: 28:18)



Memory Instructions with an Offset

- ldr r1, [r0, #4]
  - r1 ← mem[r0 + 4]
- ldr r1, [r0, r2]
  - r1 ← mem[r0 + r2]

But you know as luck would have it life is not that easy. So, ARM has many, many kinds of complicated variants of memory instructions. So, let us study let us at least attempt to study most of them in this particular chapter. So, ARM also has a base index kind of mode where we use the basic ldr instruction - the load instruction to load the value of their address to essentially load the value of the bytes that are stored at this memory location r 0 being the base register and four being the offset. So, I am sorry it is not a base index mode, it is a base offset mode and. So, in this case 4 is the offset that is being added to r 0 and then this is being saved in r 1. So, it is a base offset mode of addressing.

And similarly we can have a base register and we can have an index registered. So, in this case, what we do is that the; so essentially the address of the load instruction is supposed to be between these two square brackets. So, this particular instruction is working as follows that into r 1 we are saving the contents in memory where the address is given by r 0 being the base register, r 2 being the index register. So, we take their contents, we add the contents of r 0 and r 2, we come up with an address and use that address to access memory and read in 4 bytes.

(Refer Slide Time: 30:03)

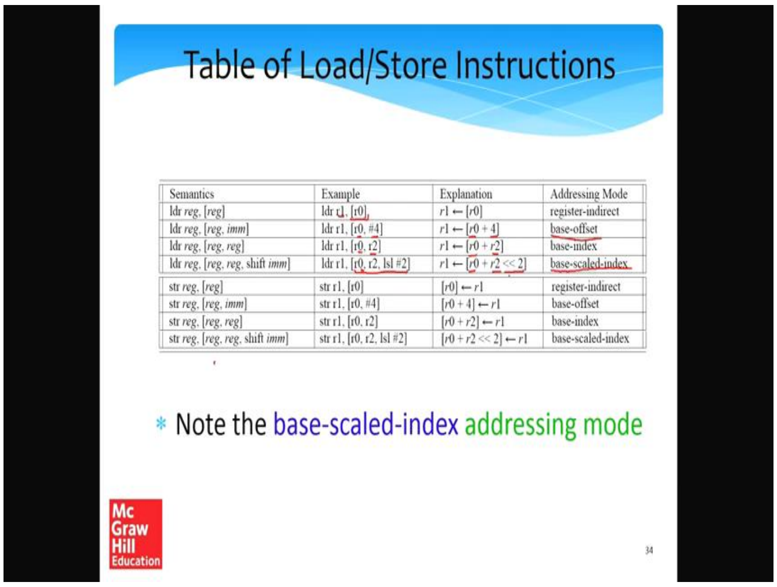


Table of Load/Store Instructions

Semantics	Example	Explanation	Addressing Mode
ldr *reg*, [*reg*]	ldr r1, [r0]	$r1 \leftarrow [r0]$	register-indirect
ldr *reg*, [*reg*, *imm*]	ldr r1, [r0, #4]	$r1 \leftarrow [r0 + 4]$	base-offset
ldr *reg*, [*reg*, *reg*]	ldr r1, [r0, r2]	$r1 \leftarrow [r0 + r2]$	base-index
ldr *reg*, [*reg*, *reg*, shift *imm*]	ldr r1, [r0, r2, lsl #2]	$r1 \leftarrow [r0 + r2 << 2]$	base-scaled-index
str *reg*, [*reg*]	str r1, [r0]	$[r0] \leftarrow r1$	register-indirect
str *reg*, [*reg*, *imm*]	str r1, [r0, #4]	$[r0 + 4] \leftarrow r1$	base-offset
str *reg*, [*reg*, *reg*]	str r1, [r0, r2]	$[r0 + r2] \leftarrow r1$	base-index
str *reg*, [*reg*, *reg*, shift *imm*]	str r1, [r0, r2, lsl #2]	$[r0 + r2 << 2] \leftarrow r1$	base-scaled-index

* Note the base-scaled-index addressing mode

Mc Graw Hill Education

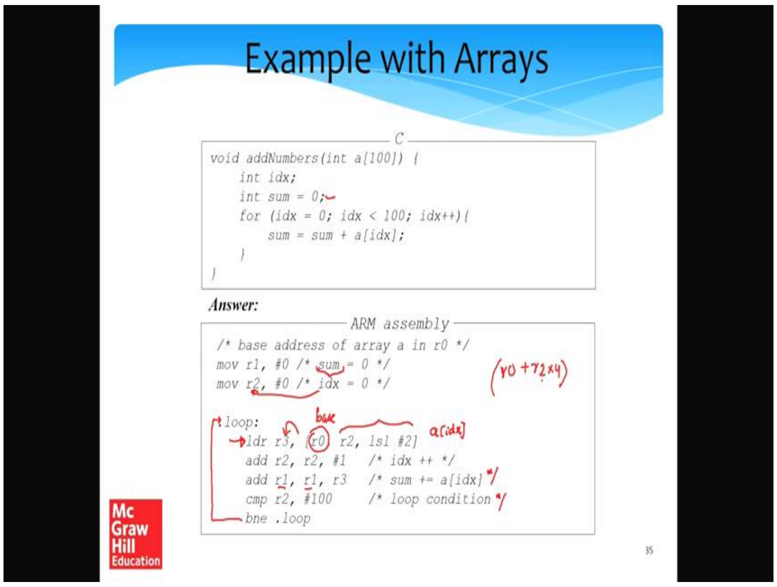
34

So, let us take a look at all kinds of addressing modes that are possible in ARM. So, load and store the addressing modes are very symmetric. So, we will only consider the case of load. So, we can have a regular register indirect mode which is of the form ldr r 1 square brackets r 0 where we take the contents of the memory address is given in r 0. We have a base offset mode. In this case, the base register is provided and the offset is provided. So, we take the contents of the base register and we add four and we save it in r 1.

Similarly, we have a base index mode where which has r 0 and r two. So, which is add the contents of r 0 and r 2 and access memory, save the bytes in r 1. Then we have this mode which is slightly complicated. So, this is a base register. So, r 0 is still the base, but we have a scaled index. So, in this case, we are shifting the index to the left by two positions that is the reason I had a left shift operator over here. So, this is the base scaled index mode which is useful for accessing arrays and so on because typically we would access arrays if the if we access them element wise like 0, 1, 2, 3, 4 in terms of bytes if each element takes 4 bytes, so they need to be accessed as 0, 4, 8 and so on. So, we can take the sequence zero one two and multiplied by 4 or left shifted by 2.

So, this is where the notion of the base and scaled index comes in, where we so the portion of text within the square brackets denotes the address, and the way that the memory address is computed is as follows, it is r 0 plus r 2 multiplied by the left shift amount or divided by the right shift amount. But let us see it is r 2, in the case of a left shift it is lsl hash two which means. So, this r 2 l s l hash two is a shifter operand and since it have l s l we can have l s r, a s r, and r o r, so all four of those shifts are allowed. So, in this particular case we have given l s l as an example. So, in this case, we are pretty much taking r 2 and shifting it to the left point two position.

(Refer Slide Time: 32:49)



Example with Arrays

C

```
void addNumbers(int a[100]) {
    int idx;
    int sum = 0;
    for (idx = 0; idx < 100; idx++){
        sum = sum + a[idx];
    }
}
```

Answer:

ARM assembly

```
/* base address of array a in r0 */
mov r1, #0 /* sum = 0 */
mov r2, #0 /* idx = 0 */

.loop:
    ldr r3, [r0, r2, lsl #2]
    add r2, r2, #1    /* idx ++ */
    add r1, r1, r3    /* sum += a[idx] */
    cmp r2, #100      /* loop condition */
    bne .loop
```

So, in this case r 0 is the base, r 2 is index. And in a scaling the index which means multiplying the index by 4, which is the same as left shifting it by two positions. I discuss the last slide, let me explain a small example with arrays. So, let us first take a look at the a c program on top. So, in this case, the input is an array of hundred elements. So, we have a sum and we have an index. So, the idea is to add all the elements in the array, in this case the index moves from 0 to 100. So, we iterate through all the elements in the array. And we do some equal sum plus a idx which essentially means where adding all the elements of the array to the sum. So, the final output will be there in the some variable and the some variable can be returned as well, but I am not showing that.

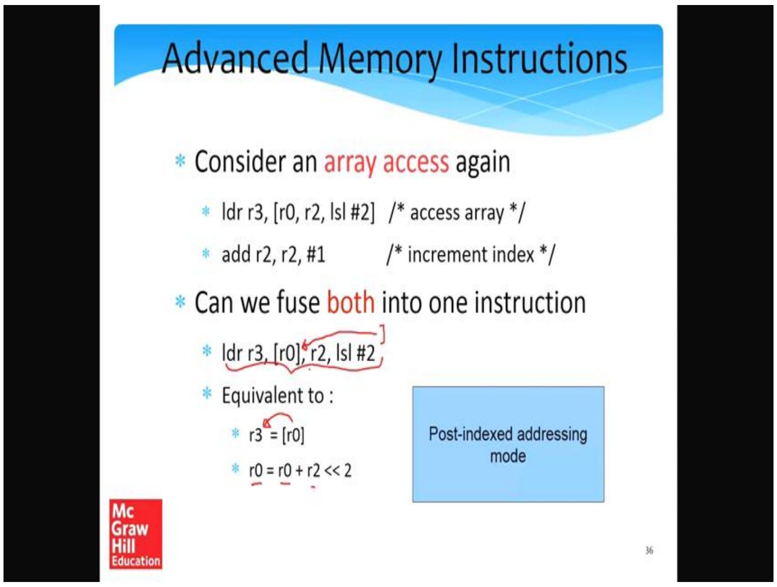
So, in ARM assembly, the first task is to first you know map for the variables to addresses. So, in this case, sum is equal to 0, which means that the sum variable is map to register r 1, and register r 1 is said to 0. So, this is also the starting point in the C program, I will set an index equal to 0 and we map index to register r 2. So, the first thing that we do is that from we need to load from memory, the value of the first array element, this can be done very easily.

So, since we assume as a base address of the array is an r 0. So, we use this as the base register then we use a shifter operant, where r 2 is the index and we essentially compute r 0 plus r 2 times 4, the reason 4 comes is because one integer is 4 bytes and we were accessing memory at the level of bytes. So, we will take one register r 2 which contains the index multiplied by 4 which is the same as shifting it to the left by two positions. So, r 2 l s l hash 2. So, this gives us the address of a idx. We read the value from memory and put it in r 3 then we increment index add r 2 r 2 hash 1. And we compute sum plus equal to a i d x. So, actually we should be having a comment over here. So, basically the comment should end, so sum plus equal to a i d x.

So, in this case we take the current sum which is an r 1, we add r 1 with the value that we read from memory which is r 3. After computing the sum, we compare the index which is an r 2 with 100. So, if you have reached 100, which is basically if you have reached 100 over here the loop terminates. So, if you are not reach 100, so if this branch not equal to which means r 2 is not equal 100 which means we have more iterations left. So, we go back to the beginning of this fields which is dot loop.

So, this let us count the number of lines of code in C, it is 1, 2, 3, 4 and we actually we should count the closing brace of the fifth line because it is essentially taking us to the beginning of the for instructions. So, let us say depending upon how we count it is four or five lines of code. The number of assembly lines is 1, 2, 3, 4, 5, 6, 7, so may 4 or 5 lines are converting to 7 lines which is actually not all that bad and the reason that we were able to have such an efficient assembly representation is only because of this ldr instruction over here, which allowed us to access the contents of memory for an array very easily. Because the address here we are being we have a very easy way of computing r 0 plus r 2 times 4 in a single expression, where we can specify the base register the index and what the index should be scaled with which in this cases for. So, this is what enabled us to write nice and compact code.

(Refer Slide Time: 37:23)



Advanced Memory Instructions

- Consider an array access again
  - ldr r3, [r0, r2, lsl #2] /* access array */
  - add r2, r2, #1 /* increment index */
- Can we fuse both into one instruction
  - ldr r3, [r0], r2, lsl #2
  - Equivalent to :
    - r3 = [r0]
    - r0 = r0 + r2 << 2

Post-indexed addressing mode

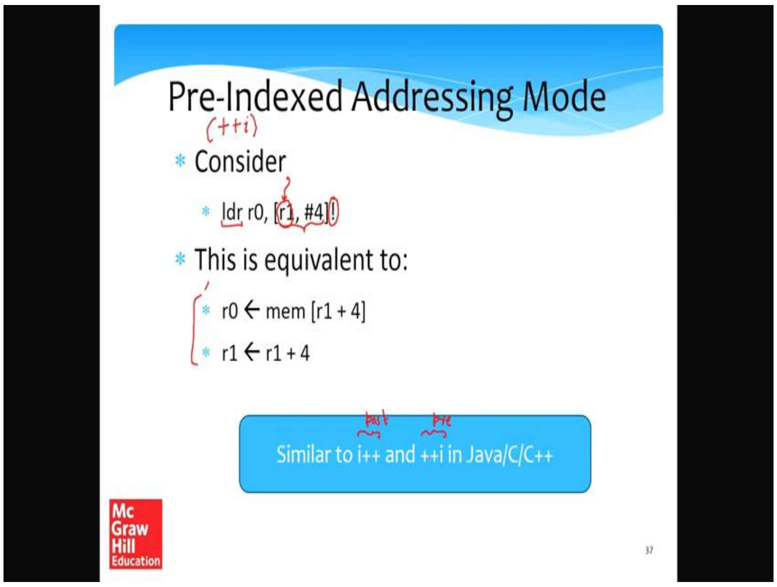
36

Now, let us take a look at even more advanced memory instruction. So, consider an array access again. So, I will just take out the relevant instruction may be if I just go to the previous slide this was the instruction that was being used to access the array. So, I can again come back over here and have load r 3, I am just in a repeating the array accessing instruction. So, we have this in subsequently we increment the index which is r 2. So, this is a very common patterns. So, whenever we access an array we will most likely access it within a loop. So, the typical pattern would be that we access in array element and then increment or decrement the array index. So, we should have a way of using both into one instruction which in this case would imply that you know we have to sort of tell the processor that after doing this increment r 2.

So, let us have a method of doing it. So, in this case let us ARM defines a new format for the format is like this that we write ldr r 3. And we sort of move the end of the square bracket to so instead of encapsulating entire thing, we move the end of the square bracket just after r 0. So, we will have ldr r 3, r 0 which is the base address registered within a square bracket and then the index and the scale a mode and the shift a mode right. So, the only change is that the square bracket has moved its position. So, it has pretty much moved from I would say this position to this position that is the only change.

So, this is a new kind of assembly instruction format that we are introducing for the ldr instructions. So, this is equivalent to set r 3 as the contents of r 0; and subsequently, you compute r 0 is equal to r 0 plus r 2 times 4. So, this is what this particular instruction is equivalent to its called a post-index addressing mode. So, basically the so what is this codes snippets over here equivalent to its equivalent to that we first read the contents of r 0 into r 3, and then we compute r 0 is equal to r 0 plus the contents of a shifter operant which is r 2 left shifted with 2.

(Refer Slide Time: 40:16)



Pre-Indexed Addressing Mode

(++i)

* Consider
  * ldr r0, [r1, #4]!
* This is equivalent to:
  * r0 ← mem [r1 + 4]
  * r1 ← r1 + 4

Similar to i++ and ++i in Java/C/C++

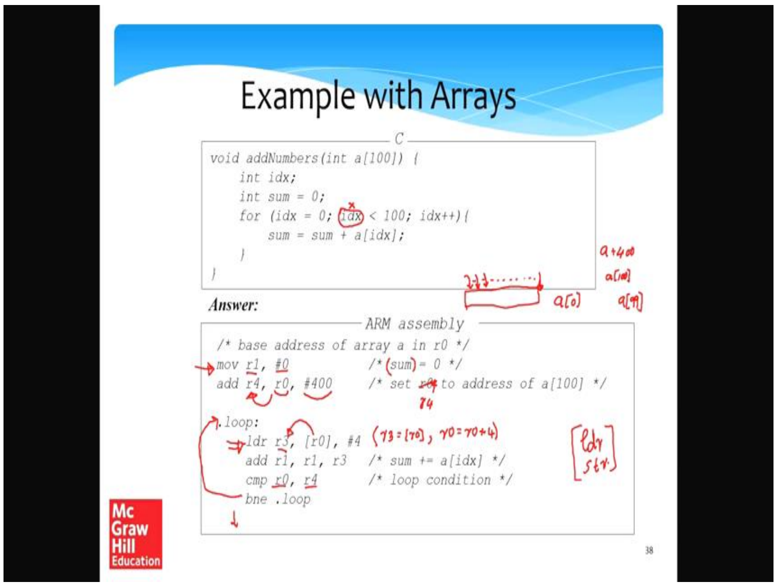
post
pre
Mc Graw Hill Education
37

Similar to the post-indexed addressing mode, we have a pre-indexing addressing mode as well. So, post-indexing is something similar to i plus plus, and C and C plus plus. So, pre-indexing is something similar to plus plus i, so where we increment the value of first and then proceed with the current instruction. So, the format over here is something like this that we first we will have ldr then. So, this part is common. Then we specify the address. So, the address can take any format, we can either have the register indirect mode, you know base offset mode base index mode base scaled index mode, ot does not matter, but the important point is we need to add an exclamation mark at the end.

So, this will essential tell us that we need to go for the pre-indexed addressing mode and this is equivalent to we are accessing so basically is equivalent to the fact that we access the memory at this point r 1 plus 4 is equivalent to that. That we first add 4 to r 1 compute the address and access the memory at that location and save the result in r 0. And we also increment r 1 to r 1 plus 4. So, both of these are done, so the difference at this in post-indexed, you would have access the memory only at location r 1. But since it is pre-index, we increment r 1 first use the incremented value to access memory.

So, if I would write equivalent C code this is what it would look like equivalent you know pseudo code, that we access the memory at the augmented location of r 1 which is r 1 plus 4, and also then we increment to r 1 to r 1 plus 4. Now, that we have seen the pre and post-indexed addressing mode which is very similar to i plus plus and plus plus i and C, it is time to see how we can use them.

(Refer Slide Time: 42:25)



Example with Arrays

C

```
void addNumbers(int a[100]) {
    int idx;
    int sum = 0;
    for (idx = 0; idx < 100; idx++){
        sum = sum + a[idx];
    }
}
```

a+400
a[100]
a[0]
a[99]

Answer:

ARM assembly

```
/* base address of array a in r0 */
mov r1, #0              /* sum = 0 */
add r4, r0, #400        /* set ~~r4~~ r4 to address of a[100] */

.loop:
    ldr r3, [r0], #4    (r3=[r0], r0=r0+4)
    add r1, r1, r3      /* sum += a[idx] */
    cmp r0, r4          /* loop condition */
    bne .loop
```

[ldr str]
38

So, let us go back to our example where we were essentially adding numbers. So, this example is as follows that let me again correct it with at the end of the comments. So, we are adding hundred numbers. So, we are adding hundred numbers in a loop and we use this complicated expression over here to pretty much compute r 0 plus r 2 times 4 times 4 is same as left shifting it be two positions. So, r 2 contain the index. So, this gave us the address of the element in the array with r 0 is a base register. So, the address would increase as follows 0, 4, 8, 12, 16 and so on.

So, now, let us go back and take a look at the same example with slightly reduced code. So, before looking at the example, I just want to stress once again that i plus plus is the pre-indexed I am sorry i plus plus is the post-index mode similar to what is there in java, C, C plus plus. And plus plus i is the pre-indexed mode, where we first increment the variable and then use it.

So, let us take a look at an example with arrays are the same example adding the first hundred elements of an array. So, basically let us see our aim is to use either a post index are pre-index addressing mode. So, in this case, we shall use the post-indexing addressing mode. So, let us start. So, first thing, that we need to do is register assignment. So, we take are some variable and so which was supposed to contain the final sum and we initialize it to zero. This is common, this is something we have to do, but here is something that we want to do. We want to eliminate the index from our computation completely from our assembly program completely. So, you know we do not want to use the index; we want to rather that use the sophisticated addressing modes.

So, let us do one thing, let us compute the end of the address of the first byte that is at the end of the array write. The first byte which is actually illegal, if you consider this is the memory region that contains the array let us consider the first byte which is not in the array, it just outside it. So, this is essentially so since a we have hundred elements in array a the elements are from a 0 to till a 99 and pretty much a 100 is an illegal element. So, the address of a 100 - the starting address would be the first address after the end of the array, and this will be a plus 100 times 4 because one integer is 4 bytes. So, this is a plus 400.

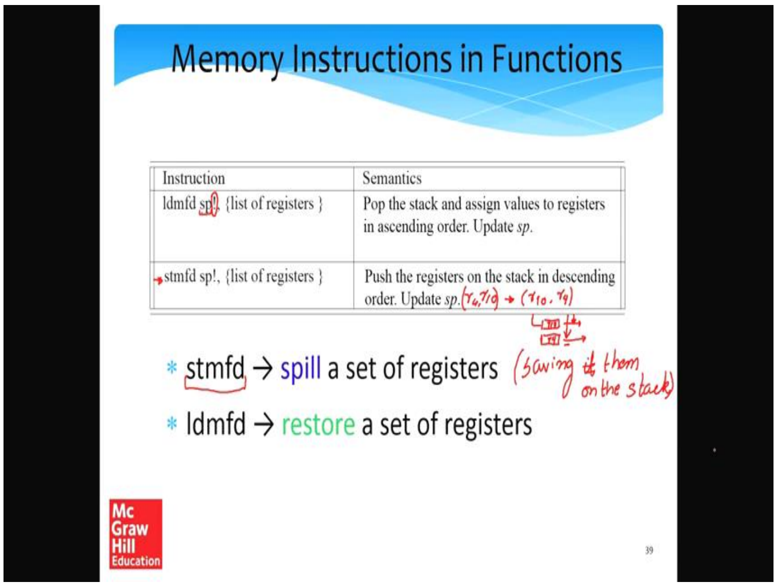
So, let us do that let us take the value of the base address r 0, let us add 400 to it. So, essentially we are setting r 0 to the address of a 100 and we save this a r 4. So, we are actually not setting r 0 there should we were setting r 4. So, let me just checks that is not correct, we are setting r 4 to the address of a 100, which is an illegal address. So, let us do one thing. So, let us see this is essentially all are all the greatness of ARM pretty much lies in this instruction which is the first instruction where. So, we starts r 0 contains a base address. So, we start an access r 0, the value is transferred to r 3; subsequently, we increment r 0 by 4. So, all of this will be done with this instruction.

So, what this instruction would do that it would set r 3 2 the contents you know the memory contents for the address is an r 0, and then it would set r 0 equals r 0 plus 4. And this is the crux of the post increment instruction that both of these things will happen within the same instruction where the instruction is ldr. So, in the ldr, the load instruction if we shall load the value you know this value r the value from memory whose address is contain in r 0 then we shall increment r 0. Subsequently the value that we read we shall add that to the sum. So, this is same as computing sum plus equals a i d x.

And here what we do is since we are incrementing r 0 in every iteration, we just compute the fact if r 0 has reached r 4 or not, which essentially means r 0 we are treating it as a moving pointer. We just first pointing to the first element then the second element then the third element till it reaches the end of the array and sort of goes out of it. So, we compare r 0 with r 4 where r 4 is an illegal address. So, if r 0 is equal to r 4 then clearly we need to exist; if it is not equal to r 4, we need to come to the loop over here.

So, if we will compare the previous let us again in a quickly go back and count the number of lines. So, the number of lines before the loop is not important mainly because the loop will actually you know heavily overshadow the number of lines before it, the loop runs for hundred times and each iteration of the loop is five instruction. So, pretty much we will have 500 instructions in the loop, but if we come here we see that each iteration of the loop actually has four instructions. So, we will have a total of 400 dynamic instructions which is a healthy savings of 20 percent right. So, we were able to reduce the number of instructions by a fifth by using this post-indexed addressing mode. So, both pre indexed and post indexed addressing mode can be used for the load instruction and the store instruction as well. All of these examples are with the load instruction, but they can be done for the store instruction as well.

(Refer Slide Time: 49:00)



Memory Instructions in Functions

Instruction	Semantics
ldmfd sp!, {list of registers }	Pop the stack and assign values to registers in ascending order. Update *sp*.
stmfd sp!, {list of registers }	Push the registers on the stack in descending order. Update *sp*. $(r_4, r_{10}) \rightarrow (r_{10}, r_4)$

- stmfd → spill a set of registers (saving ~~it~~ them on the stack)
- ldmfd → restore a set of registers

39

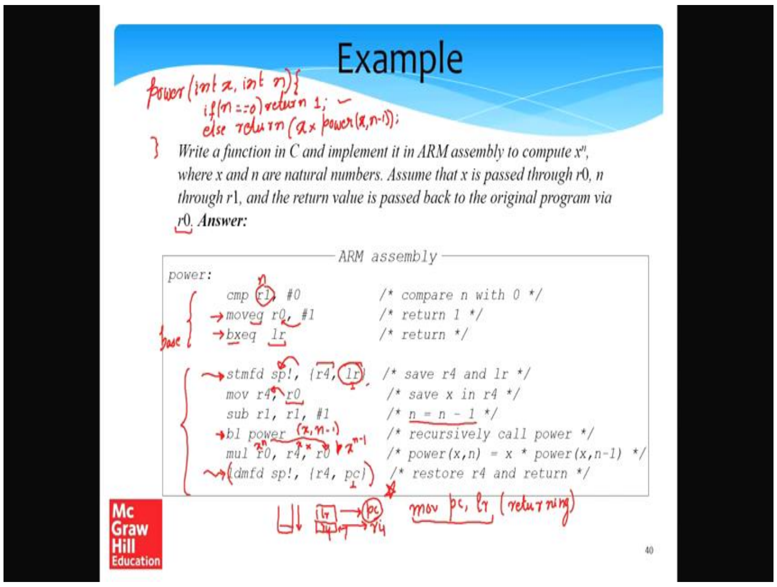
Now, let us discuss some of the memory instructions that are used in functions. So, we often see that it is necessary to store a set of registers before calling a function and it is necessary to restore them. So, this code to store registers is called spilling. So, we have two schemes caller saved and calli saved in both the schemes we spill a set of registers which means save it on the stack; and then once we return from the function, we restore the set of registers. So, we typically required several instructions to spill the set of registers and restore them. So, ARM has made this job easy. So, there is one instruction is stmfd for spelling a set of registers and essentially saving it on the stack saving not yet, but I would rather save them .

Similarly, we have the ldmfd instruction to restore a set of registers. The format of both the instructions are roughly the same. So, we first specify the stack pointer then similar to the pre-index addressing mode, we have an exclamation mark. So, the exclamation mark is basically to say that at the end of the instruction, the next instruction should see an updated stack pointer. For example, if you push in three registers then at the end of the instruction, we should see the stack pointer subtracted by 12.

So, let us may be considered the stmfd instruction first. So, here we push the registers on the stack in descending order and we update sp. So, let us take a look at this descending order. Let us assume that we want to push the values of registers r 10 let us a r 4 and r 10. So, what the hardware would do is the hardware would first sort then in descending order. So, it would be r 10 and then r 4. The values of these registers would be pushed in the same descending order. So, r 10 we will essentially be pushed first saving the stack is downward growing r 10 and then we would push r 4.

So, the ldmfd instruction does exactly the opposite. So, what it does it starts at the top of the stack and pushes and I am sorry pops the stack and assigns values to registers in ascending order, which is the reverse order. So, first you would pop r 4, then it would pop r 10 and update the value of the stack pointer to what it was. Before we shall find both of these instructions ldmfd and stmfd to be extremely useful when we discuss you know when we show one example how to used into save and restore registers.

(Refer Slide Time: 52:07)



Example

```
power(int x, int n){
    if(n==0) return 1;
    else return(x× power(x,n-1));
}
```

Write a function in C and implement it in ARM assembly to compute x^n, where x and n are natural numbers. Assume that x is passed through r0, n through r1, and the return value is passed back to the original program via r0. ***Answer:***

ARM assembly

```
power:
    cmp r1, #0            /* compare n with 0 */
    moveq r0, #1          /* return 1 */
    bxeq lr               /* return */

    stmfd sp!, {r4, lr}   /* save r4 and lr */
    mov r4, r0            /* save x in r4 */
    sub r1, r1, #1        /* n = n - 1 */
    bl power              /* recursively call power */
    mul r0, r4, r0        /* power(x,n) = x * power(x,n-1) */
    ldmfd sp!, {r4, pc}   /* restore r4 and return */
```

40

So, here is the example. So, the idea is like this that let us write a function in C and implement it in later in ARM assembly to compute x to the power n, where x and n are natural number. So, assume that x is passed through r 0 and n through r 1, and the return value is passed back to the original program via r 0. So, I will not show the C code, but the C code is easy to write. So, essentially if you have a function of the form power int x and int n, we first want to check if n is equal to equal to 0, you would return one because any number to the power 0 is 1. So, we return 1, else we return C function is very straightforward actually else we return x times power x and minus 1. So, this is just the simple function in C that we have.

So, I am not you know, it is a simple one line functions are not further spending time to explain it. So, the idea is that if the power of n, if n is equal to 0, any number to the power 0 is 1. So, we return one otherwise we return x times the power function x to the power x times x to the power n minus 1. So, now, let us take a look at how to implement it in ARM assembly. So, we can first implement the first line which is compare n with 0; since n is stored in r 1 we compare it with 0, if there is an equality. So, we use a predicated or a conditional instruction over here that if n is equal to equal to 0, and you know equality holds then we will execute this instruction moveq which will move one in r 0 which is a return value.

Similarly, if there is an equality we will execute the b x instruction and we will jump to the return address. So, mind you these two instructions are conditional instructions, if there is an equality found in the compare, these two instructions will execute; otherwise the hardware will just convert them to no ARM. So, the nice thing of using conditional instructions is that we avoid branches and so on. So, it makes a code look very neat and straight forward subsequently us in this particular approach what we are doing is that we are first saving the values of the registers that we will most likely change and then we go on and we do our changes

So, think of this is primarily callee saved scheme, because here if you assume that this function is a callee then we save all the registers that the caller might require and also the registers that we will modified and then we proceed. So, basically since it is recursive call definitely the return address register will get over written, so we save it. Along with that we save a temporary register r 4 because this also we will get over written. So, we save r 4 and lr in the stack pointer using the sdmft instructions. So, we save them on this stack. Subsequently, we move the value of x which is an r 0 to r 4, the main reason for doing this is that r 0 will also carrying a return value. So, in general in the ARM abi, abi is application binary interface other convention is that you know return values typically come via r 0 and arguments are typically passed in r 0, r 1, r 2 and so on

So, since r 0 is being used both as an argument as well as the return value before calling another version of the power function, we have to save r 0 in another register let it be r 4. Subsequently, we subtract r 1 by 1, because r 1 contains n. So, this is equivalent to n is equals n minus 1, and we call power. So, mind you in the power function, what are the arguments that have gone the arguments that have gone are x and n minus 1, and a final result of the power function is an register r 0 which contains x to the power n minus 1. We multiply this with r 4, where r 4 contains x. So, x times x to the power n minus 1, becomes x to the power n. Subsequently we need to; so since you know this particular approach is a callee saved approach what we need to do is that we need to restore the values of the registers that we have modified. So, r 4 is something that you are modified. So, we need to the restore it. And similarly, well lr we are modified, but we do not have to restore it you have to do something else. So, this instruction is actually straightly tricky, but let me explain how it works.

So, let us first consider the case of r 4. So, as I had said we save these in descending order. So, essentially if this is the stack, which is in a download growing. So, this is the stack top, we first have r 4 and then we have the value of lr. Now, when we will execute the ldmfd instruction we will pop the registers in ascending order. So, r 4 will get popped out and you get assign to r 4, which means that the value of r 4 is getting restored. Now, here is the trick. So, basically we have lr over here, but when we next pop out l r instead of saving it in l r we actually save it in PC. So, mind do you know this is the trick. So, let me put a star over here. So, this is same as moving the contents of l r into p c.

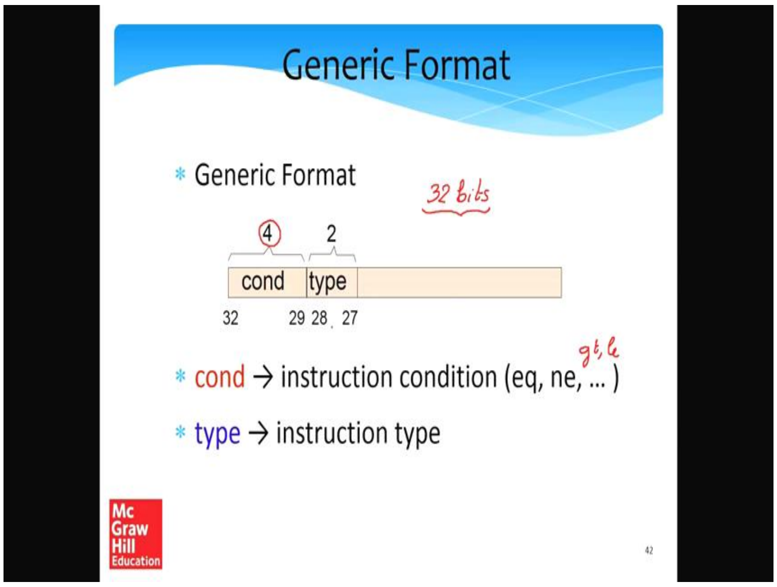
So, this is exactly the same as move lr into PC which is exactly the same as returning from the function. So, as we can see you know we are achieving multiple objectives essentially killing two birds with the same stone, we had modified r 4. So, as a result just before returning, we are restoring the value of r 4. So, the caller will not be able to know. Secondly, lr is something which you know we had modified, so that is we later on modified, so that is the reason we saved it on the stack. But now when we are returning there is no real justification of actually restoring lr because in any case it makes little sense to restore it after the function has returned. What we instead need to do is we need to effect the return.

So, we essentially pop the value of lr, but instead of saving it in lr we save it in PC. So, this has exactly the same effect as move lr into PC or essentially returned from the function. So, this is a very, very standard way of writing ARM code where we have a matching stmfd and ldmfd instructions. So, it will restore all the values of the registers, but instead of restoring the value of lr, what we do is we pop the value of l r and put it in p c. So, this is tantamount to returning from the function. So, this little piece of code, so actually if you take a look at it this c code has essentially two statements inside it and in terms of assembly code it is actually fairly complicated, so it turns 3, 4, 5, 6, 7, 8, 9 statements, but actually it could have been actually much, much more. We reduced it by using tricks in the ARM ISA. So, one trick that we used is that we use the e q suffix, the e q condition to ensure there are no branches and we put in a predicated instruction there, so that took care of the base case.

To take care of the recursive case we decided that we have to call other versions of the power function. So, to call other versions of the power function, we have to save and restore some registers because particularly register r 0 is containing an argument and is also suppose to contain the return value. So, what we do is, so we again have two options caller saved and callee saved. So, had we done caller saved kind of approach that would have also been fine? Have the caller saved approach we basically see that what are registers that the caller would actually require and we saved those registers. So, the registers are essentially r 0 and l r that the caller will required. So, we should have saved them and then again you know restore them after the power function.

In a callee saved approach the idea is straightly different, in this case what we essentially do is that we find out what are the registers if the callee is actually modify and we saved them at the beginning and restore them at the end. So, is that the caller is not able to find out what they callee did. So, in this case, r 4 and lr are registers that we save because we will modify them later; and again while returning from the function we restored r 4 and instead of restoring lr, we pop it out, but put its value in the PC which is same as returning from a branch.

(Refer Slide Time: 63:19)



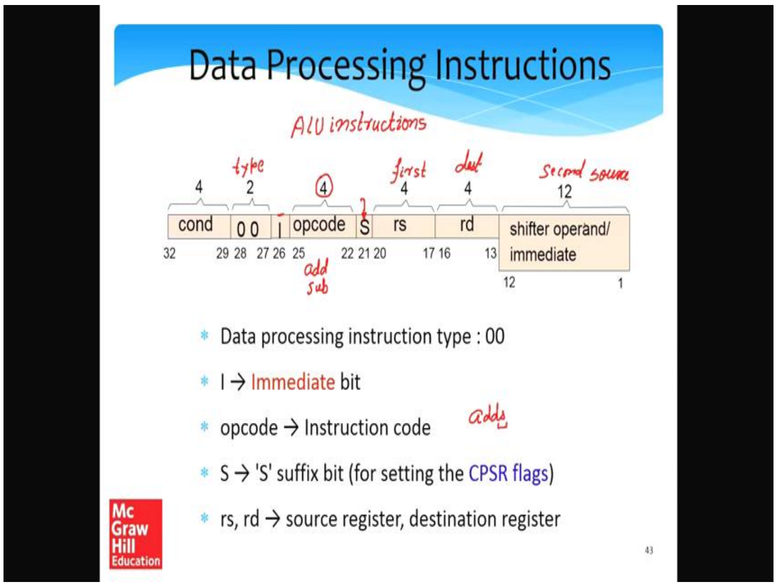
Generic Format

* Generic Format

* cond → instruction condition (eq, ne, ...)
* type → instruction type

Let us now look at ARM instruction encoding. So, here is basic problem is that we need to encode instructions in a bit field of 32-bits. So, let us first define the generic format which most of the ARM instructions follow. So, here the first 4 bits, bits 29, 30, 31 and 32 these 4-bits represent the condition. So, we had talked about fourteen condition codes in previous slides. So, this condition codes for e q, n e, g t, l e and so on. So, all of this condition codes since there are total fourteen of them plus the al condition code which means always, so fifteen of them we need 4-bits to encode them. So, for the top 4-bits of the encoding encode the condition code; the next two bits encode the type of the instruction. Since, we have two bits ARM can support mainly four kinds of four broad types of instructions. So, we will see what the types are in the next few slides.

(Refer Slide Time: 64:22)



Data Processing Instructions

- Data processing instruction type : 00
- I → Immediate bit
- opcode → Instruction code
- S → 'S' suffix bit (for setting the CPSR flags)
- rs, rd → source register, destination register

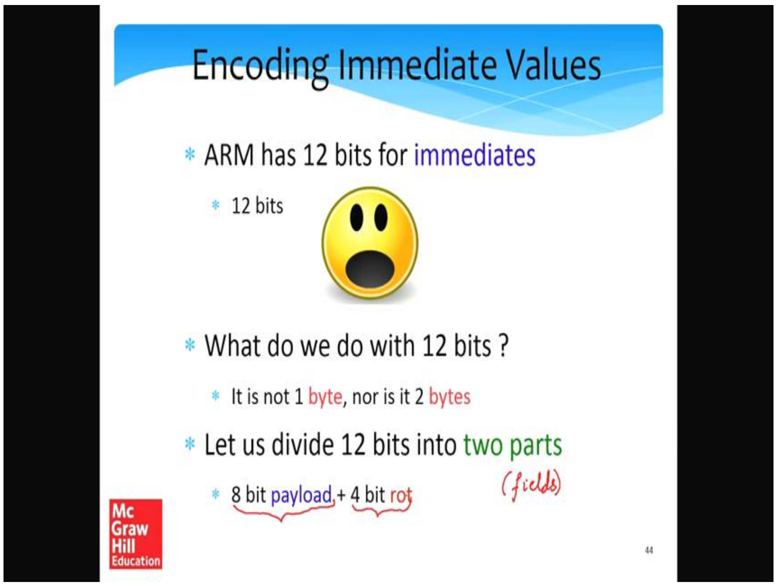
43

So, let us consider a generic format of data processing instructions which are basically ALU instructions - arithmetic and logical instructions. So, let me write it. So, let us consider. So, the for the first six bit. So, they follow the generic format. The first 4-bits of the condition and 0 0 is the type of the instruction type 0 0; subsequently the I bit or the immediate bit indicates is the second source operand. So, this is the second source operand if this is an immediate or it is a shifter operand, essentially if it is it a register shifted by a certain value or it is an immediate. So, this is specified by the I field the immediate bit. Subsequently the next 4-bit specify the off code of the operation code, what kind of operation, it is such as add, subtract, logical AND, logical OR, and so on.

So, the 21st bit is the S bit. So, the S bit is the S suffix bit that we talked about. So, for example, we talked about the instruction add S. So, the add S instruction sets the CPSR flags a program status register flags and so this is the S suffix. So, this can be added to any data processing instruction this will automatically make it set the CPSR flags. Subsequently, we have the idea the first source register since the 16 source registers we need 4-bits to encode them. So, this is the ID of the first source register. Then this is the

ID of the destination register r d, and a last 12-bits are reserved to specify either a shifter operand or an immediate.

(Refer Slide Time: 66:30)



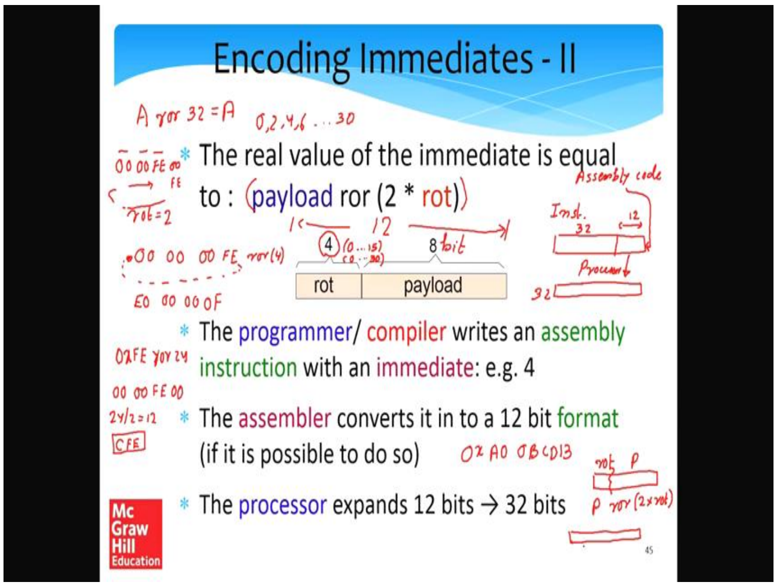
Encoding Immediate Values

- ARM has 12 bits for immediates
  - 12 bits
- What do we do with 12 bits ?
  - It is not 1 byte, nor is it 2 bytes
- Let us divide 12 bits into two parts (fields)
  - $\underbrace{\text{8 bit payload}}$ + $\underbrace{\text{4 bit rot}}$

44

So, encoding immediate values is kind of tricky in ARM. So, ARM is 12-bit for immediate. So, 12-bits, 12-bits is a bad number; it is not 8, it is not 16. So, it is essentially not 1 byte, not it is 2 bytes. So, let us divide 12-bits into two parts or two fields. So, let one be an 8-bit payload and other a 4-bit rotation value. So, we will see what these are.

(Refer Slide Time: 67:00)



Encoding Immediates - II

A ror 32 = A   0, 2, 4, 6 ... 30

00 00 FE 00   FE   rot = 2

00 00 00 FE ror(4)

E0 00 00 0F

- The real value of the immediate is equal to : (payload ror (2 * rot))

12
4 (0 ... 15) (0 ... 30)
8 bit
rot
payload
Assembly code
Inst.
32
12
Processor
32

- The programmer/ compiler writes an assembly instruction with an immediate: e.g. 4

0xFE ror 24

00 00 FE 00

24/2 = 12

$\boxed{\text{CFE}}$

- The assembler converts it in to a 12 bit format (if it is possible to do so)   0x A0 0B CD13
- The processor expands 12 bits → 32 bits

45

So, the real value of the immediate can be calculated like this. So, let us again consider the 12-bit field, so that 12-bit field over here is being divided into my 8-bit payload. So, the 8-bit payload is basically one byte of information and a 4-bit field called rot - rot. So, the way that the immediate is calculated is like this. So, considered an ARM instruction, which is encoded in a bit field of 32-bits. In the ARM instruction the least significant 12-bit are encoding an immediate. So, what the processor would do is that the processor will essentially expand the 12-bits to a 32-bit quantity inside. Actually even before that when i specify an immediate in assembly code where is consider this is as assembly code in a specified in immediate the assembly code would try to convert it encode it into this 12-bit form, if it is possible. So, we will discuss this later

Subsequently, once this has been encoded in 12-bits, once this actually executes what the processor will do is they will take these 12-bits out and expand the 12-bits into a 32-bit quantity. The way that the processor would do it is as follows; it will take the 8-bit payload and right rotate the payload with a number which is 2 times rot. So, let us consider an examples. So, let us consider a hex two hex digits, one hex digit is 4-bits. So, two hex digits are 8-bit. So, let us consider a 32-bit number, and let us consider payload of the form let say F E.

Now, let us see that the value of the rot field is equal to 2. In this case, this number needs to be right rotated by four positions. So, right rotation basically means that all the bits in the lsp position come to the msp position. If I right rotate a number by four positions, it basically means that 4-bit from the lsp come to the msp. And 4-bits is one hex digit, so what this were essentially become is E 0 0 00 0F. So, similarly I can encode a lot of immediate and figure out their actual value. So, the actual value at the processor would use, and the standard formula is this one that I take the 8-bit payload, and I right rotated with a value for which is 2 times rot.

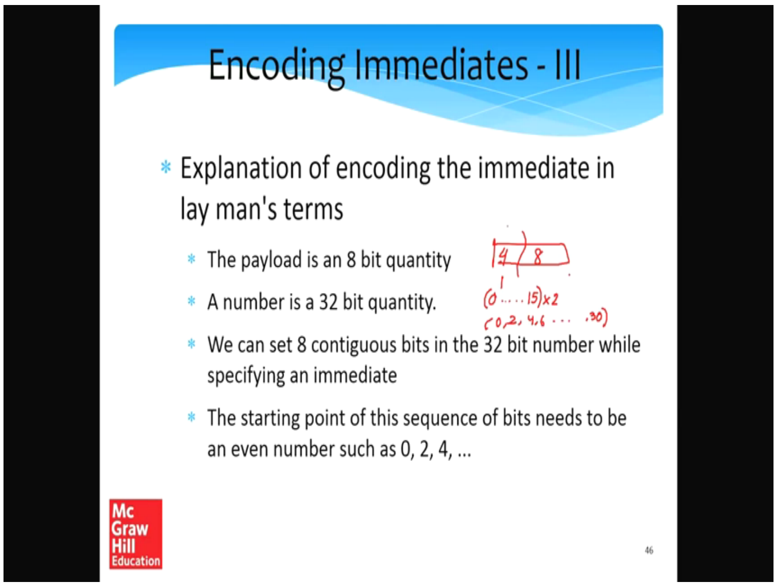
So, rot since it is a 4-bit number it can be between 0 to 15. So, two times rot can be between 0 to 30. So, one thing that is obvious is that if I take a number and I write rotate it by 32 positions then again the number the 32-bit number you will become exactly the same. So, you can see one bit is right rotated with 32 positions, it will come, come, come, again it will reach its original position the same is true for all the bits. Say any number any 32-bit number right rotated by 32 positions is equal to the number itself. So, essentially 32 is a limit, but what the ARM format is saying is that it allows you to specify 8-bits and further a right rotation such that you can rotate it by any number which is in the range of 0, 2, 4, 6 just even number till 30.

So, this sort of you know the intuition behind this is like this that is look we have a limited number of bits and instruction format. We only have 12-bits. So, 12 is a bad number; it is not 8; it is not 16. So, what is the best that we can do we can specify one byte of information, but the additional 4 bytes also give us a handle of things where that one byte will be located right. So, basically at which position, so at which even position that one byte will be located for example, it is very easy to encode a number of this fashion in the ARM ISA. So, what is this like? So, this is basically I have F E over here, and a right rotated by certain number of positions till it is sort of reaches over here. So, how many positions would this be if I right rotate F E by 8 positions, it will reach here; 16 positions will reach here; 24 positions, it will reach here

So, basically F E, I can consider 0 x F E right rotated sorry r o r 24 is essentially this number over here. So, there is encoding with basically we said 24 divided by 2 is 12. So, it is encoding will basically be that the rot field will be 12 CFE. So, this is the encoding of the immediate field. So, as we can see the notion of immediate is slightly tricky in ARM, it is actually not very simple, but in any case it is definitely possible to encode a lot of numbers with this. So, the first thing is that when the programmer or the compiler they write an assembly instruction with an immediate for example, let say 4, the job of the assembler is to convert it to a 12-bit format, if it is possible to do so.

If it is not possible to do, for example, the number of this form just has too many bits. So, it is not possible to specify it in the 12-bit format. So, multiple instructions are required, but in any case if it is we use multiple instructions, each of the instruction specifies an immediate which can be put in this format, and the assembly code is created. Subsequently, when the processor sees this instruction it takes the 12-bits out and expands them to 32-bits, how would you do it, it would basically take the field of 12-bits divided into two parts the rot and let say the payload p. It will then take payload and right rotate it with a number which is 2 times rot, this will give us a 32-bit number which the processor will internally use.

(Refer Slide Time: 74:00)



Encoding Immediates - III

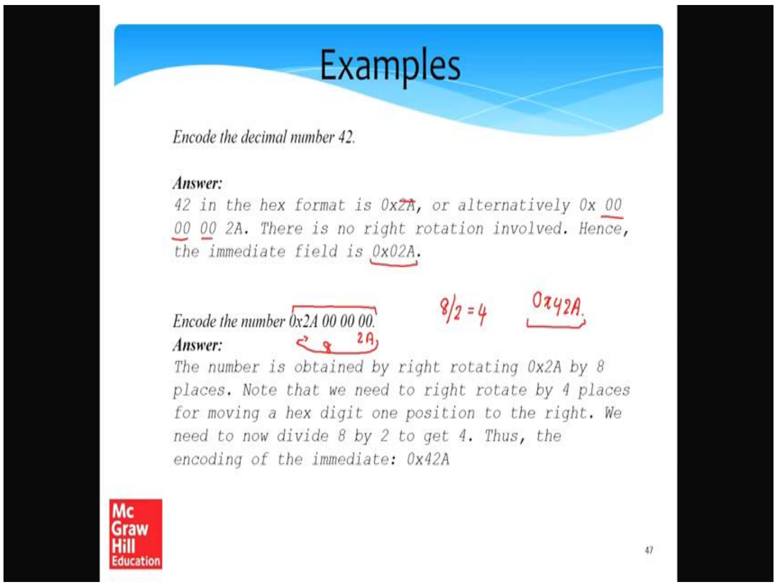
- Explanation of encoding the immediate in lay man's terms
  - The payload is an 8 bit quantity
  - A number is a 32 bit quantity.
  - We can set 8 contiguous bits in the 32 bit number while specifying an immediate
  - The starting point of this sequence of bits needs to be an even number such as 0, 2, 4, ...

4
8
(0 .... 15)x2
(0, 2, 4, 6 ... 30)
46

So, let us look at the explanation of encoding in slightly lay man’s terms. So, the payload is an 8-bit quantity, and number inside the processor or a number in a register in any 32-bit processor is a 32-bit quantity. So, since the payload is 8-bits, we can only set 8 you know preferably contiguous bits in the 32-bit number at the time of specifying an immediate, only 8-bits can be set. The additional choice that ARM assembly is giving us is that we get to choose with 8-bits these are. So, how many bits are left out of 12-bits, if this is the 8-bit payload the choice that ARM is saying is that look here four more bits do whatever you want. With 4-bits, what can you do, you can any encode any number between 0 and 15. So, since in general, we will not place numbers at odd positions starting at odd position at least ARM says that multiply this by 2. So, then the range will be. So, numbers will be at the form 0, 2, 4, 6, 8 till 30.

So, it take this 8-bit number and place them at any of this position by right rotating it. So, the starting point of the sequence is an even number and ARM gives us some amount of flexibility to actually do this. So, this is actually a fairly complicated aspect of the ARM ISA. So, I would request all the readers to spend a lot of time in understanding this particular aspect. So, the book has a lot of examples and has some amount of explanation also with regards to this, but my request to all readers is to understand this part at least very, very well.

(Refer Slide Time: 75:53)



Examples

Encode the decimal number 42.

Answer:

42 in the hex format is 0x2A, or alternatively 0x 00 00 00 2A. There is no right rotation involved. Hence, the immediate field is 0x02A.

Encode the number 0x2A 00 00 00.

$8/2 = 4$ 0x42A

8 2A

Answer:

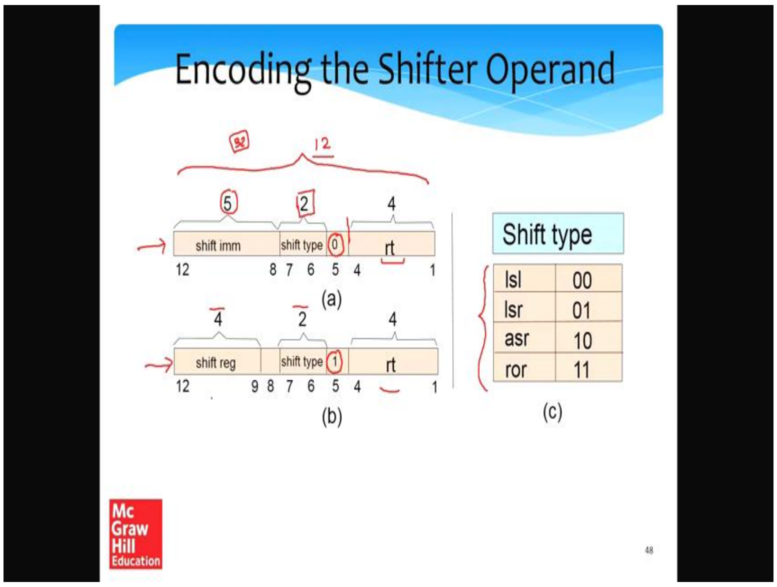
The number is obtained by right rotating 0x2A by 8 places. Note that we need to right rotate by 4 places for moving a hex digit one position to the right. We need to now divide 8 by 2 to get 4. Thus, the encoding of the immediate: 0x42A

47

So, let us look at some examples, encode the decimal number let say 42. So, 42 in the hex format is 0x2A - two times 16 plus 10. Or alternatively, if I want to make it a 32-bit quantity we can have we can put three bytes before it all zeros. So, since there is no right rotation involved the immediate field is 0x02A. Now, let us encode the number 0x2A 000. The number is obtained by including 0x2A by actually eight positions by actually right rotating 0x2A by 8 positions.

So, basically the iteration is set to move one hex digit, it takes four right rotations. So, move two hex digits, it will take so move. So, move a hex digit by actually two positions two places, it will take 8 right rotations. So, the number of right rotations is 8, we divided that by 2, we get 4. So, the encoding of the number is 0x42A. So, similarly we can have all kinds of numbers then we can try to encode them. So, these can actually be thought a trick questions with a main idea is that ARM immediate are slightly tricky, and it needs to be kept in mind that we are only setting eight contiguous bits. And so the essentially that is the trick that eight contiguous bits inside a 32-bit number are being set and their starting position can be varied using the rot - the rot field.

(Refer Slide Time: 77:34)



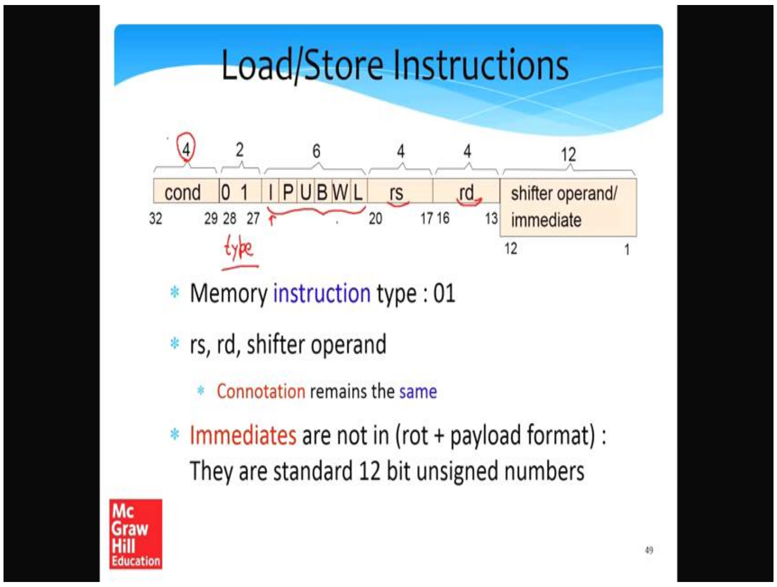
Encoding the Shifter Operand

Shift type	
lsl	00
lsr	01
asr	10
ror	11

So, next now look at encoding the shifter operand. So, basically if we are not using an immediate yet we are using a register that can possibly be shifted then let us look at how to do it. So, we have 12-bits. So, let us so there are essentially two options. So, one option is that we are actually shifting a register by an immediate value. So, this registered is specified as the lower 4-bits is called register r t and the fact that you are shifting it by an immediate value is encoded in the next bit, which if it is zero means we are shifting it when immediate value. So, the shift type, so we have four kinds of shifts in our ARM ISA, lsl, lsr, asr and ror. So, these four shift types since there are four they can be encoded with two bits with lsl being 00 and ror being 1 1. So, the shift type is encoded in a next two bits and subsequently is the shift amount. So, any number can only be left shifted and right shifted or rotated by at the max of 32 positions, so we need five bits which is exactly what we have, so that is lucky for us

Let us now consider the second variant. In this case, we again have the rt register which requires 4-bits, but instead of this number being 0 the 5th bit - the 5th bit is one which means that we are actually shifting rt by the value in a register. So, the shift type is encoded with the next two bits, and the top for bits encode the idea of the register which contains the shift amount.

(Refer Slide Time: 79:29)



Load/Store Instructions

4	2	6	4	4	12
cond	0 1	I P U B W L	rs	rd	shifter operand/ immediate
32 29	28 27		20 17	16 13	12 1

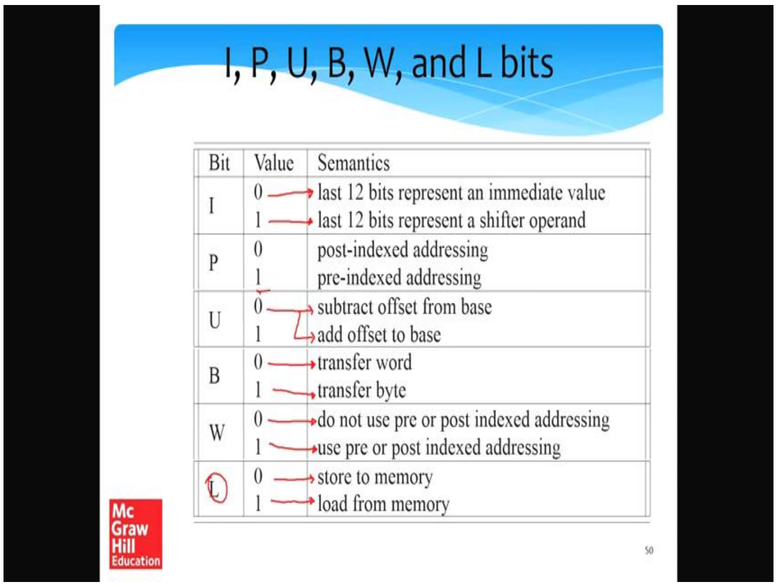
type

- Memory instruction type : 01
- rs, rd, shifter operand
  - Connotation remains the same
- Immediates are not in (rot + payload format) : They are standard 12 bit unsigned numbers

Mc Graw Hill Education
49

So, let us now discuss load store instructions. So, load store instructions are also follow the same generic format. So, the first 4-bits represent the condition the particular the all the 16 conditions that we have seen. In this case, the type of the instruction is actually 0 1. So, this is the type of the instruction 0 1; and subsequently we have six bits I, P, U, B, W, L, which I will discuss subsequently. So, then rd in the case of a load instruction is a destination of the load and rs is the base register. So, in the case of the store, they maintain the similar connotation; rs is still the base register and rd is a registered there the value comes from. And the most important thing is that immediate in the load store format or not in the load plus payload format, instead they have standard 12-bit unsigned numbers right this is an important distinction to keep in mind that these are standard 12-bit unsigned numbers, and they are not in any special format neither are they sign numbers.

(Refer Slide Time: 80:55)



I, P, U, B, W, and L bits

Bit	Value	Semantics
I	0	last 12 bits represent an immediate value
	1	last 12 bits represent a shifter operand
P	0	post-indexed addressing
	1	pre-indexed addressing
U	0	subtract offset from base
	1	add offset to base
B	0	transfer word
	1	transfer byte
W	0	do not use pre or post indexed addressing
	1	use pre or post indexed addressing
L	0	store to memory
	1	load from memory

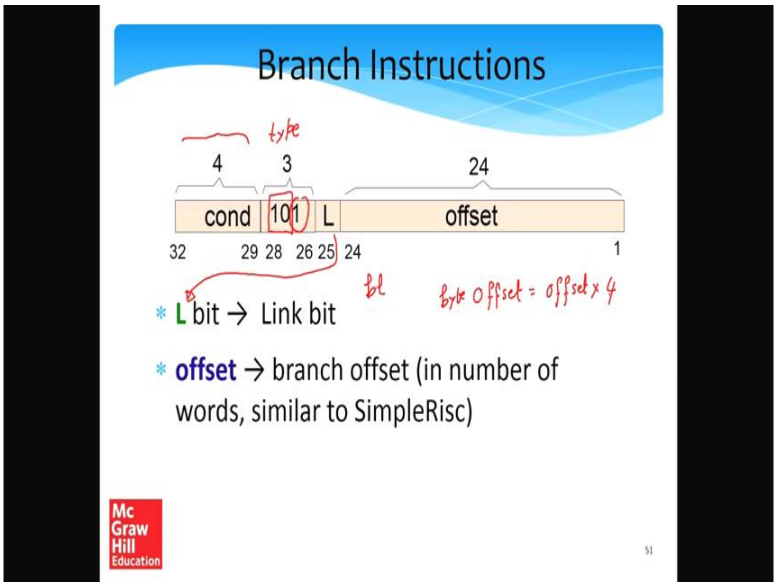
Mc Graw Hill Education

50

So, what we need to discuss now is a connotation of these fields I, P, U, B W, L. So, the I bit basically if it is 0, it says that the last 12-bits represent an immediate value, write a fixed offset from the base register; and if it is 1, it says that the last 12-bits represent a shifter operand, so that is the I bit. So, the connotation on the I bit can slightly change in one case if I is 1, it represent an immediate with that is the connotation in this case. The P bit, if it is 0, it represent post-indexed addressing mode; if it is 1, it represents the pre-indexed addressing mode. The U bit says that if the U bit is 0, we subtract the offset form the base; otherwise, we add the offset to the base register. If the B bit is 0, we transfer a memory word which in this case is 4 bytes or we transfer a single byte which is 1 byte.

Similarly, if the W bit is 0, it says that use regular addressing do not use pre or post-indexed addressing. So, W can be thought of as the right back bit were essentially the base register is changing. And if W bit is 1 means that use pre and post-indexed addressing. Lastly how do we differentiate between a load in a store well that is very simple. So, if the bit is 0 means we store to memory. So, L can be thought of as the load bit and if the bit is 1, we load from memory.

(Refer Slide Time: 82:28)



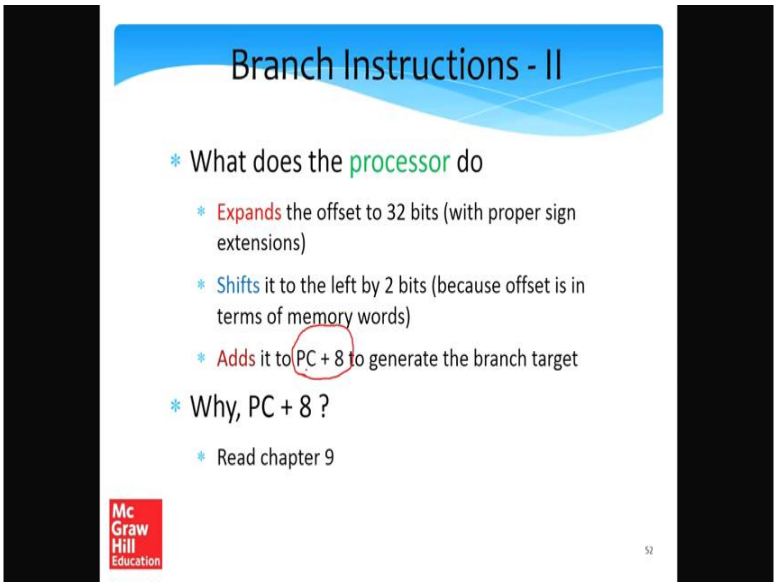
Branch Instructions

- L bit → Link bit
- offset → branch offset (in number of words, similar to SimpleRisc)

Lastly let us take a look at branch instructions. So, in the case of branch instructions, we have this exactly the same generic format. So, get a 4-bit condition and actually the type of the instruction is 1 0, so that is the type there is a further one bit for the sub type which were at least most of the common instructions is 1. Subsequently, we have the L bit or the link bit, so with a link bit recall the bl instruction. Say the link bit is one it basically means that the return address register is sets. So, we are sort of looking at a function call and subsequently we have a twenty 4-bit offset. So, this is the branch offset very similar to simple RISC, and also similar to simple RISC this is not the offset in terms of bytes is the offset in terms of memory words. Memory word is defined as four bytes in this case.

So, let us call it the byte offset is equal to the offset specified over here multiplied by 4. This is very similar to simple RISC because the idea is if you assume that every instruction starts at a boundary of 4 bytes then the last two lsb bits are 0 and 0. So, there is no need to specify an offset whose last two bits are always constant. So, instead let us specify the offset in terms of memory words. So, we will essentially save two bits in this case.

(Refer Slide Time: 84:09)



Branch Instructions - II

- What does the processor do
  - Expands the offset to 32 bits (with proper sign extensions)
  - Shifts it to the left by 2 bits (because offset is in terms of memory words)
  - Adds it to PC + 8 to generate the branch target
- Why, PC + 8 ?
  - Read chapter 9

Mc Graw Hill Education

52

So, for the branch instructions what does the processor do, it expands the offset to 32-bits with proper sign extensions. It shifts it to the left by two bits because it is in terms of memory words. And it adds it to PC plus 8 to generate the branch target. So, before you ask me let me tell you why PC plus 8, this is sort of non-intuitive right, but we are not in a position to answer this you need to wait till chapter 9 for the answer to this question. So, this brings us to the end of the beautiful chapter on ARM assembly. So, I will see you in the videos of a chapter 5 which are on x86 assembly.