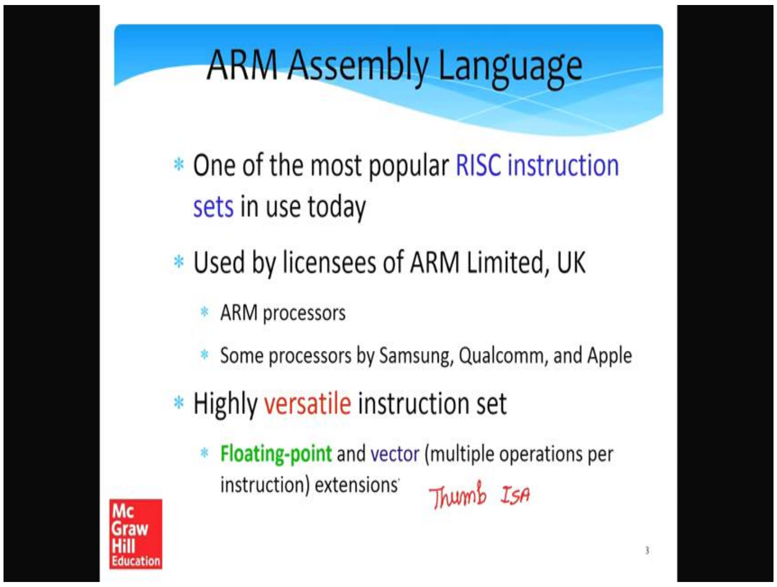
Computer Architecture

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

Lecture - 08
ARM Assembly Language Part – I

Welcome to the chapter on the ARM assembly language. ARM assembly language are some of the most popular and most exiting assembly language as to learn. So, in this chapter we will have a lot of fun. So, this chapter is the part of the book computer organization and architecture, publish by myself, Doctor Smruthi Sarangi. It was publish by Mc Graw Hill in the year 2015 and should be available in almost all book stores and online in a both normal book stores and online book stores.

(Refer Slide Time: 01:37)



ARM Assembly Language

- One of the most popular RISC instruction sets in use today
- Used by licensees of ARM Limited, UK
  - ARM processors
  - Some processors by Samsung, Qualcomm, and Apple
- Highly versatile instruction set
  - Floating-point and vector (multiple operations per instruction) extensions Thumb ISA

So, why study the ARM instruction set and so let us first look at why study and then what is the background required. So, ARM is very popular instruction set. So, almost all the phones today as of 2016 tablet us and all kinds of small computing devices, which are not desktop are laptops run on ARM processors. So, they use the ARM instruction set.

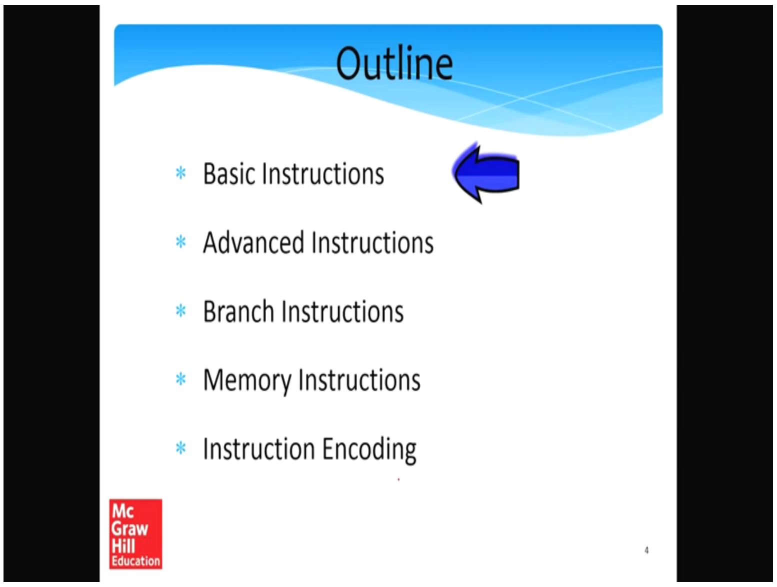
So, ARM is company located in Cambridge UK. So, they actually licensed their designs and then the designs are used by other manufacturers to you know incorporate them in silicon chips. So, before actually we start this chapter I would like to make a point that, readers should already have a good understanding assembly language, in the sense that they should have read the previous chapter which is chapter 3 and understood lot of things about how assembly programs are written, how they instructions, how assembly instructions work are the notion of function notion of the stack the notion of encoding.

So, all of this needs to be there and the reason that the book has been designed in such a way is basically that, is you know it is not possible at list in my view to give an introduction to assembly language and teach an advanced assembly language at the same time. It is better to separate the concerns. So, it is better to first teach what an assembly language is like a very simple one and then mov to teaching in advanced assembly language which is used in commercial processors. Because than a student would have the right amount of understanding, to actually understand what state of the art assembly instruction set looks like.

So, keeping this in mind I will actually start this chapter at a slightly higher level because my assumption is that readers are already coming after at chapter 3. So, they understand the basics of assembly languages, and they also understand how to write simple programs with assembly languages, and have a basic understanding of how to implement is statements loops functions using stacks using assembly language. So these are some other concepts will take for given in this particular chapter, coming back to ARM. So, ARM has a lot of licensees all over the world. So, what ARM produces is they create the design of a processor and then the sell the design. The design can be incorporated by a third party with other component. So, for example, if ARM sells a processor to a phone company, then they can incorporate the ARM processor and also have other smaller circuit surround it. For example, a circuit to process inputs on the camera, or a circuit to inter face with the accelerometer in the phone. So, all of these additional circuits can be added.

So, the ARM instruction set as such is very versatile. So, along with lot of integer instructions, is suppose floating point instructions. And it has what are called vector extensions. So, will discuss more about vector instructions in chapter 11, but vector instructions allow was to do multiple additions at the same time in the same cycle. So, will not be discussing these extensions also ARM has a very popular I was say in assembly sub language or in a extension is called the thumb instruction set. And a lot of ARM programs actually use the thumb instruction set which is the slightly simpler set off instructions, and what we present here in you know in this chapter cycle referred to it has the thumb ISA are the thumb assembly languages. So basically the thumb instruction set is similar to this it is slightly simpler will not discuss this in this chapter. We will discuss the generic ARM instruction set for integer only in this chapter.

(Refer Slide Time: 06:12)

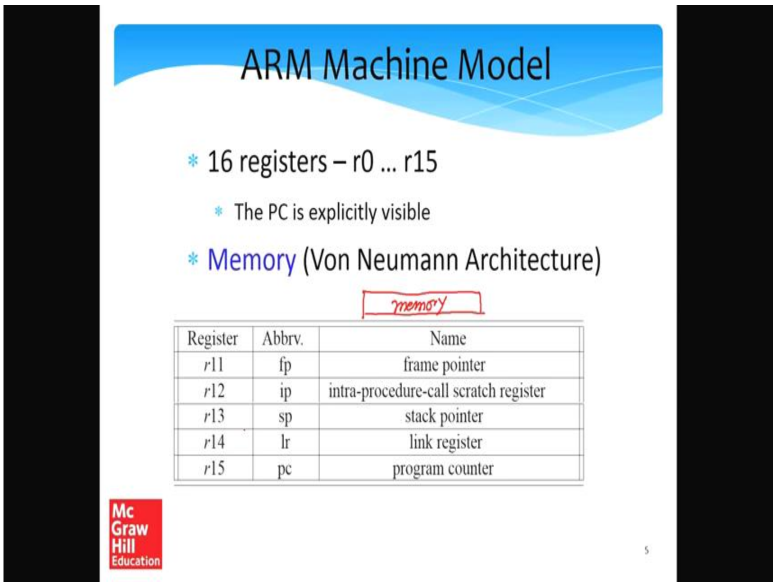


Outline

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

So, will have 5 separate sections here will first start with basic instructions. Mov to advanced instructions, then look at branch instructions, memory instructions, low store kind of instructions, and finally, considered instruction encoding.

(Refer Slide Time: 06:33)



ARM Machine Model

- 16 registers – r0 … r15
  - The PC is explicitly visible
- Memory (Von Neumann Architecture)

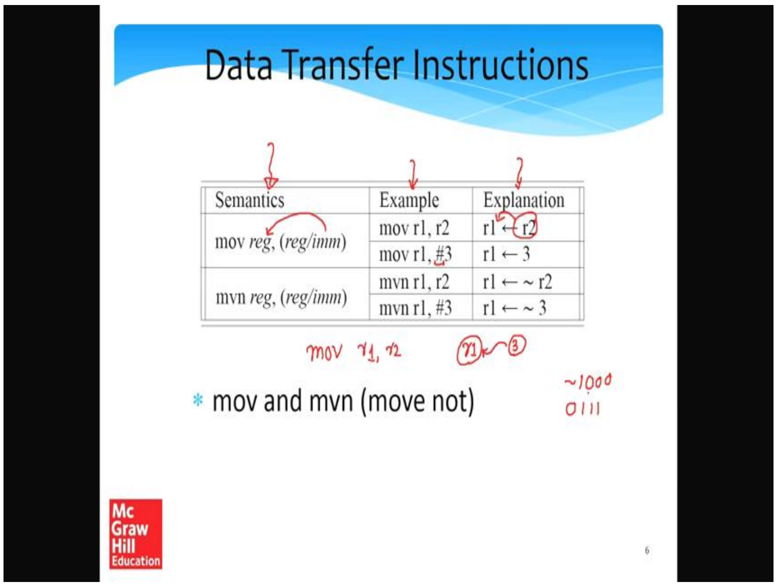
memory

Register	Abbrv.	Name
r11	fp	frame pointer
r12	ip	intra-procedure-call scratch register
r13	sp	stack pointer
r14	lr	link register
r15	pc	program counter

So, similar to simple RISC ARM has 16 registered their number from r0 to r15 and unlike simple RISC can many other assembly languages the problem counted the explicitly visible, and the program counted can be use 2 effect branches and so on.

So, this is expose to software is a expose to the assembly language. The memory is standard Von Neumann architecture what this mean is the instruction memory and the data memory is fuse to 1. So, this is you know typical facet of the Von Neumann architecture there is Von view of memory it is not separate. So, out of the 16 register at ARM has some of them a reserve for special purposes. So, let us take a look at them. So, r15 is the program counter. So, I am starting in a back words ARM r15 is the program counter it is refers to was either r15 or the pc r14 is the link registered. It is you can also think of it does the return address register. R13 is reserved for keeping the stack pointer or sp r11 and r12 have will r11 and r12 can be assigned a special connotation on times. So, r12 is an intra procedure calls scratch register which basically means it can explicitly be use to safe temporary values inside a function call, but other registered can be use as well. And r11 is called a frame pointer will discuss what is a frame pointer actually towards the end of these chapter, and, but will discuss more about a frame pointer and an next chapter on x86 assembly.

(Refer Slide Time: 08:32)



Data Transfer Instructions

Semantics	Example	Explanation
mov *reg*, (*reg*/*imm*)	mov r1, r2	r1 ← r2
	mov r1, #3	r1 ← 3
mvn *reg*, (*reg*/*imm*)	mvn r1, r2	r1 ← ~ r2
	mvn r1, #3	r1 ← ~ 3

mov r1, r2

$\boxed{r1} \leftarrow \boxed{3}$

~1000

0111

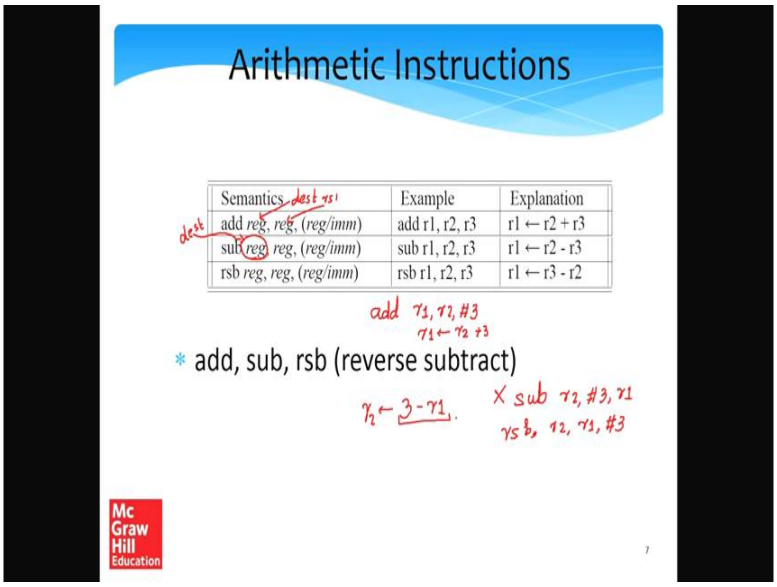
* mov and mvn (move not)

6

So, we will be using a slightly different kind of semantics in this chapter. So, we will every single instruction will be explained with the help of a table. So, in the table the first column, will look at the semantics which is something that we introduced towards the end of the last chapter. So, it will basically show what are the different modes of the instructions. Then will give a example and then will talk about the explanation and the registered transfer notation that we have introduced. So, the simplest of simple instructions in the ARM ISA is the mov instruction. So, what does the mov instruction do the mov instruction transfers the value into a registered? So, let us say may be mov r1 r1. So, it essentially transfers the valley of r2 to r1. Alternatively, the mov instructions can transfer the valley of a immediate into a registered. So, the first operand is always register. The second operand can either be a registered or an immediate, let us considered the second example mov r1. So here is the, you know idiosyncrasy of the ARM ISA. They are all immediate are preceded with the hash.

So, the moment we add this hash characters over here. So, you can see the hash character will be above 3 on your keyboard, all immediate and ARM need to be preceded or prefix to the hash. So, when we say mov r1 hash 3, what this essentially moves is that I take register r1 and I move 3 into it. So this is the simplest instruction in the ARM ISA. So, mov has variant it is called mvn, mvn is move not. So, this is similar to the not instruction in simple RISC. So, what we do is that, when we do mvn r1 comma r2 essentially the ones complement or every single bit flipped is transferred from r2 to r1. So, this tilde sign is a once complement. So, just to recapitulate the ones complement of 1 0 0 0 in Boolean is 0 1 1 1. So, every single bit is replaced with it is complement one is replaced with 0 and 0 is replaced with 1. So, the mvn instruction is essentially the same as a mov instruction, but instead of moving the register all the immediate it moves in the complement of the register or immediate.

(Refer Slide Time: 11:35)



Arithmetic Instructions

Semantics	Example	Explanation
add *reg*, *reg*, (*reg/imm*)	add r1, r2, r3	r1 ← r2 + r3
sub *reg*, *reg*, (*reg/imm*)	sub r1, r2, r3	r1 ← r2 - r3
rsb *reg*, *reg*, (*reg/imm*)	rsb r1, r2, r3	r1 ← r3 - r2

dest rs1

dest

add r1, r2, #3

r1 ← r2 +3

* add, sub, rsb (reverse subtract)

$r_2 \leftarrow \underline{3 - r1}$.

× sub r2, #3, r1

rsb, r2, r1, #3

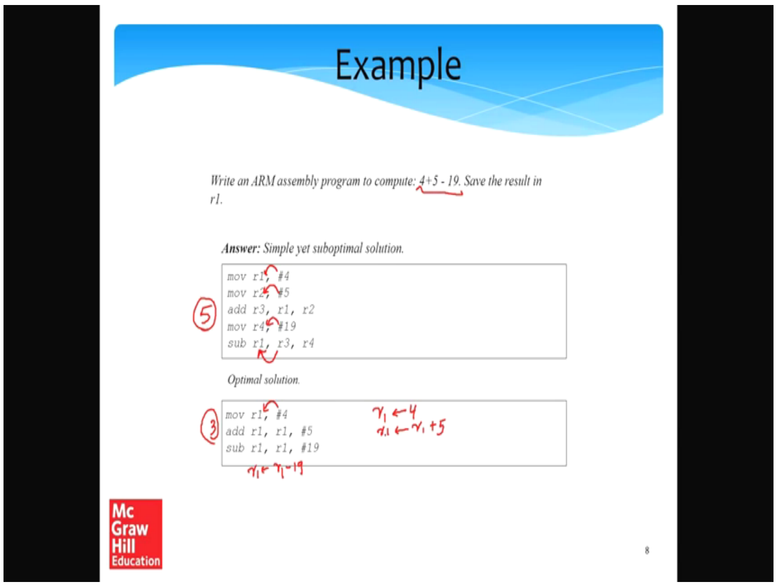
7

So, now let us take a look at arithmetic instructions, the arithmetic instructions are almost you know add and sub are exactly the same as simple RISC. So, in this case the add and sub instructions, similar to simple RISC the first operand is the destination the second operand is the first registered source which is rs 1 recall chapter 3, and the second operand can either be a register or an immediate. So, examples would be r1 r2 r3. So, in this case we add r2 plus r3 and save the result in r1. So, a again you know I would like to mention that I am deliberately going slightly fast, because my assumption is that students have already picked up a certain amount of background in the previous chapter. So, they will find covering this chapter slightly easier right, but if you let us a take a look at this chapter from scratch you will find a slightly difficulty, I would then ask you to look at the lectures or read the book for the previous chapter which is chapter 3.

So, coming back to the add instruction I can alternatively right add r1 r2 and hash 3. So, what this would do is that in r1 it would save r2 plus 3 similarly we are the subtract instruction which is exactly the same as it was in the simple RISC instruction set. So, we have the register destination first. So, the first operand over here is the register destination then we have to source operands one of them is registered source and the other is the source slash immediate source or an immediate. So, when I write sub r1 r2 r3. Essentially inside r1 we are having a placing r2 minus r3. So, similar to subtract there is another instruction is called rsb or reverse subtract.

So, in reverse subtract instead of subtracting r2 and r3 instead of doing r2 minus r3 we actually do r3 minus r1. So, here reader can ask you know if you have the sub instruction, why you need a reverse subtract instruction right, well the answer is simple in the sense, let us say that you want to compute 3 minus r1 not r1 minus 3, 3 minus r1. We cannot write an instruction of the form. So, let us say you want to set these two, r2 we cannot write an instruction are the form r2 right. So, this is not allowed because the second operand has to be a register. So, this is not allowed, but what we can write is rsb a reverse subtract oops sorry there should be a comma here r2 r1 and 3. So, what this would do is this would set r2 to 3 minus r1, which is exactly what we needed. Since simple RISC this doing this would actually require 2 instructions because we would first subtract 3 from r1. So, compute r1 minus 3 and then multiplied with minus 1. So, the designers in ARM a very smart instead of you know having this 2 instruction solution they created a new instruction call rsp reverse subtract which allows us to compute 3 minus r1 in a single instruction, all right.

(Refer Slide Time: 15:52)



Example

Write an ARM assembly program to compute: $\underbrace{4+5 - 19}$. Save the result in r1.

Answer: *Simple yet suboptimal solution.*

```
mov r1, #4
mov r2, #5
add r3, r1, r2
mov r4, #19
sub r1, r3, r4
```

$\boxed{5}$

Optimal solution.

```
mov r1, #4
add r1, r1, #5
sub r1, r1, #19
```

$\boxed{3}$

$r_1 \leftarrow 4$

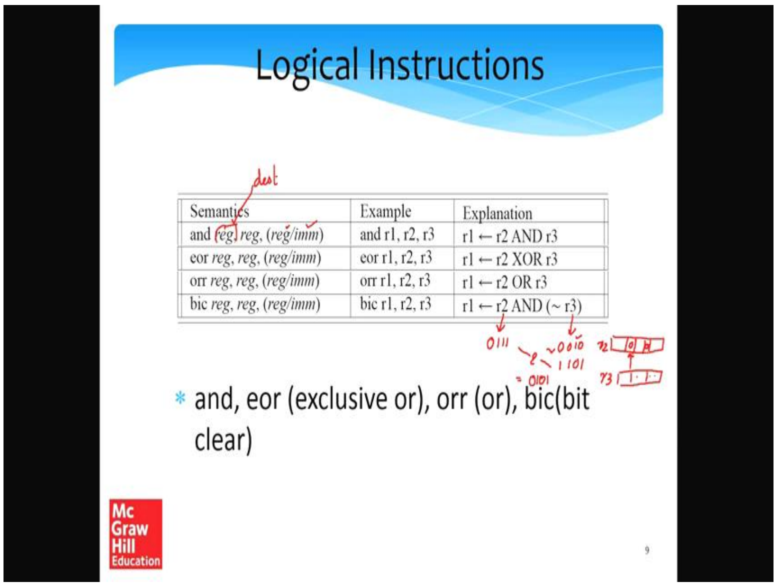
$r_1 \leftarrow r_1 + 5$

$r_1 \leftarrow r_1 - 19$

8

So, now let us take a look at some examples. So, the first example is write an ARM assembly program to compute 4 plus 5 minus 19 and save the result in r1. So, here is the simple solution which is simple, but it is not optimal, but nevertheless less to call let us take a look at it. So, we first say 4 in r1 which save 5 in r2 using the mov instruction, then we add r1 plus r2 save at an r3. We move 19 to r4 then we subtract 19 from r3 which is r1 plus r2 save the result in r1. Slightly modes are there better solution which is optimal we put 4 in r1, then we do r1 equals r1 plus 5 in the sense r1 will become 9, then we do r1 we set r1 to r1 minus 19 and a previous value of r1 is 4 plus 5 9. So, we compute 9 minus 19 has minus 10 which is the final answer. So, this was a slower solution easy to understand this is slightly difficult to understand, but not very difficult because in the first line we set r1 to 4. In a second line we do r1 is r1 plus 5. Which all of you can verify is what is exactly in the mentioned in the statement of the problem. Then we subtract a 19 from the sum and we get the right answer.

(Refer Slide Time: 17:37)



Logical Instructions

Semantics	Example	Explanation
and *reg*, *reg*, (*reg*/*imm*)	and r1, r2, r3	r1 ← r2 AND r3
eor *reg*, *reg*, (*reg*/*imm*)	eor r1, r2, r3	r1 ← r2 XOR r3
orr *reg*, *reg*, (*reg*/*imm*)	orr r1, r2, r3	r1 ← r2 OR r3
bic *reg*, *reg*, (*reg*/*imm*)	bic r1, r2, r3	r1 ← r2 AND (∼ r3)

dest

0111

~0010

1101

= 0101

r2

r3

- and, eor (exclusive or), orr (or), bic(bit clear)

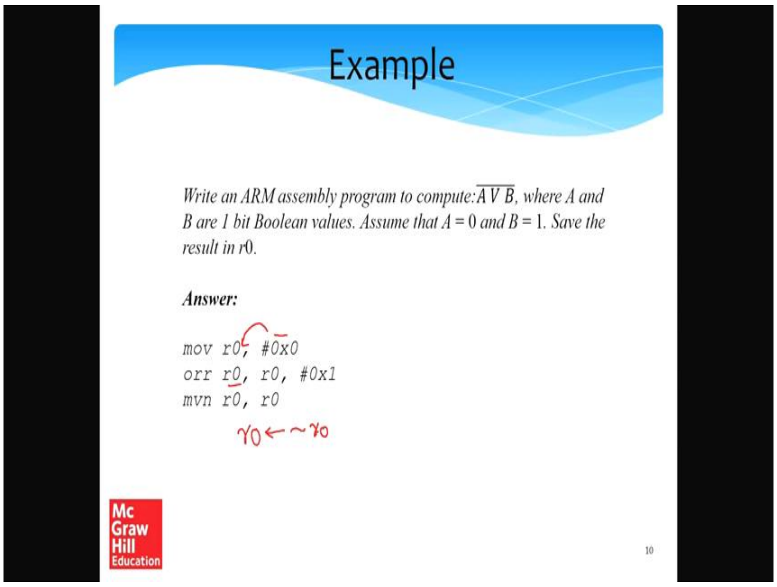
9

Now, let us take a look at logical instructions. So, logical instructions are also very similar to the arithmetic instructions. So, the first 3 instructions are somewhat easy the 4th instruction it is slightly difficult. So, the first instruction is the and instruction, which is very similar to what we had with simple RISC. So, in the and instruction is the same idea that the first. So, in all of these instructions the first operand is the destination then we are the first source and the second source. The second source operand can either be can either be a register or an immediate. So, one example would be and r1 r2 r3. So, we set r1 has r2 and r3. So, eor is actually exclusive or xor as we call it. So, in eor r 1 r2 r3 we have r1 which is being set to r2 xor r3 orr is logical or. So, in a logical or what we do, is that this is similar to the or instruction in simple RISC. So, we just compute a logical or a r2 and r3.

So, the important point to note is that in all of these instructions, the instruction per say is only 3 letters. Since simple RISC we had some to letter instructions like so on. So, that is on the case and l d and s t. So, that is not the case in ARM. So, in ARM they have tried to maintain the length of the instruction the same. So, that is the reason or they have replaced with orr. So, the bic the bit clear instruction is slightly unclear at the moment. So, what this is this is basically r1 is r2 and not of r3. So, let us may be consider simple example and see a what is these is. So let us considered r2 being, let us may be just considered 4 bits to make are like easy and rest of the bits are also easy.

And similarly let r3 b. So, not of this quantity would be 1 1 0 1. This number and this number is equal to one and 0 is 0 1 and 1 is 1. 1 and 0 is 0, 1 and 1 is 1. See if you see what is happening over here is that all the bits that are set in r3. So, all the bits that r1 in r3 in knot of r3 those bits become 0. And when that is added with some other quantity all the bits are those bit positions become 0, these that is why the name bit clear comes from, or alternatively if I want to explain if this is let says the first operand, which in this example is r2, and is the second operand which is r3. All the bits that are set at different points at similar bit positions in a final result if we have 0s, it is like those bits are those bits positions are getting cleared. So, this is called a bit clear instruction or a bic a instruction.

(Refer Slide Time: 21:19)



Example

Write an ARM assembly program to compute: $\overline{A \vee B}$, where A and B are 1 bit Boolean values. Assume that A = 0 and B = 1. Save the result in r0.

Answer:

```
mov r0, #0x0
orr r0, r0, #0x1
mvn r0, r0
```

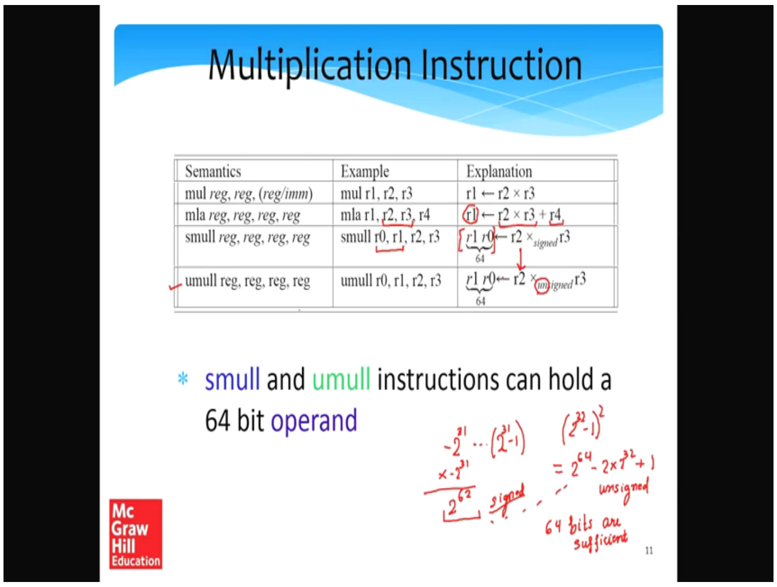
$r0 \leftarrow \sim r0$

10

So, let us again considered an example this time with Boolean variables. So, write an ARM assembly program to compute A not B sorry A or B the entire thing not where A and B are 1 bit Boolean values assume that A is 0 and B is 1. Save the result in r0. So, to compute the not of A or B what we do is that so, basically ARM accept hexadecimal numbers as well. So, the format is a same first have a hash to signify in immediate. And then you write 0 x to signify that it is an hex. So, what we do is that we load the value of 0 into r0. And we load, so I am sorry we or this with 1. So, the value of A or B at the moment is saved in r0. Subsequently what we do is we compute the knot of r0 using the mvn instruction. So, we set r0 as the knot of r0.

So, in this case, what we are doing is that we are loading to Boolean variables, a with constant were computing their or and then finally, taking the logical compliment.

(Refer Slide Time: 22:48)



Multiplication Instruction

Semantics	Example	Explanation
mul *reg*, *reg*, (*reg*/*imm*)	mul r1, r2, r3	$r1 \leftarrow r2 \times r3$
mla *reg*, *reg*, *reg*, *reg*	mla r1, r2, r3, r4	$r1 \leftarrow r2 \times r3 + r4$
smull *reg*, *reg*, *reg*, *reg*	smull r0, r1, r2, r3	$\underbrace{r1\ r0}_{64} \leftarrow r2 \times_{signed} r3$
umull reg, reg, reg, reg	umull r0, r1, r2, r3	$\underbrace{r1\ r0}_{64} \leftarrow r2 \times_{unsigned} r3$

* smull and umull instructions can hold a 64 bit operand

Let us now look at the multiplication instructions that are there in the ARM ISA. So, the simplest variant of the multiplication instruction that we have in ARM is mul m u l. So, the mul instruction is very similar to the add and subtract instructions, where the first operand is a destination register henceforth we have 2 operands. The first source operand is always a registered and the second one can be unregister immediate. So, this is straight forward r1 is r2 times r3. So, even the ARM is a RISC instruction set it has some slightly complicated instructions, and so you know there is a basic trade off.

The trade office is do we always preferred simplicity, or occasionally can we make are instruction set slightly complicated, such that you know some commonly executing patterns and programs can be accommodated. So, once has commonly executing pattern which is there in a lot of programs in a particularly coach that use linear algebra and matrix operations is the mla operation multiply an accumulate.

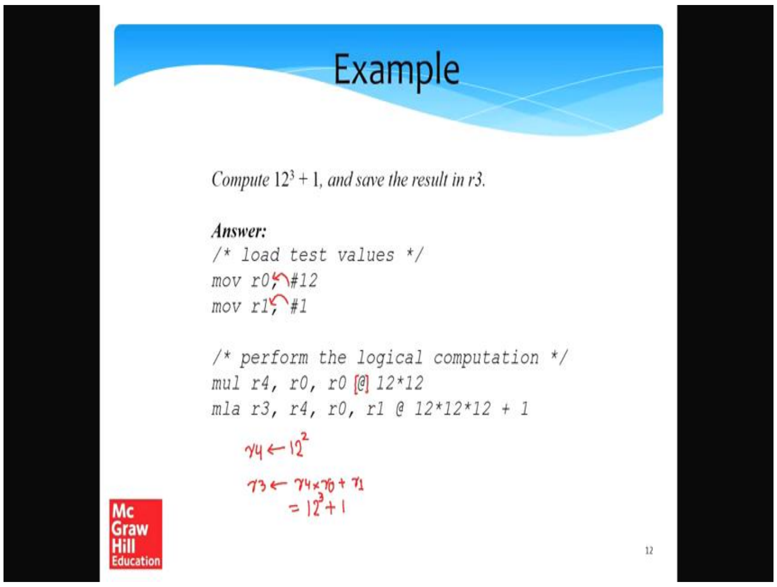
So, this actually takes 4 registers as operands. The other first register is the destination in this example r1 is the is the destination. So, out of the 3 source operand that we have what we compute is r2 times r3 plus r4. So, we have a multiplication operation where the first and second source operand have been multiplied, and we adding this this with r4. So, this is a typically required in the lot of linear algebra kind of competition that is a reason this instruction is supported. The other interesting aspect of multiplying 2 numbers is like this consider a 32-bit instruction set. So, the range of the number system is pretty much between minus 2 to the power 31 to 2 raise to the power 31 minus 1. So, let us say we multiply minus 2 to the power 31 with minus 2 to the power 31; the answer will be 2 to the power 62, which is well outside the range of a 32-bit number system. Answer as a result will have an overflow, but let us say we do not want to have an overflow and we want to have some mechanism, by which we can store the store the product without an overflow.

So, for this let us do a little bit of math here. So, as you see the largest number that we can get by multiplying 2 sign numbers 2 32-bit sign numbers 2 raise to the power 62. If I considered to unsigned number, so basically the largest number that I can get. So in a unsigned number system with 32 bits the largest number is 2 raise power 32 minus 1. If I sort of squared these then this will be 2 to the power 64 minus 2 times plus 1, we shall see that in both these cases. So, this is sign multiplication will be consider them to be sign number this is unsigned. So, in any case beat either signed or unsigned 64 bits are sufficient to keep the product, 64 bits in the sensor sufficient.

So, for this purpose ARM is 2 instructions smull and umull. So, smull is a sign multiplication, where we actually multiply the third and 4th operand you multiply r2 times r3. We do a sign multiplication we assume that r2 and r3 are sign and the final result is a sign. So, the 64-bit product is actually it cannot be saved in one register because it is a 32-bit register, but can saved in 2 register. So, the lower 32 bits can be saved in r0 and the more significant 32 bits can be saved in r1, together will have a 64-bit quantity.

So, this is the smull instruction does this with 4 operands where we multiply the third and 4th operand. And the first and second operand together store one number, where r0 stores the lower 32 bits and r1 stores the upper 32 bits. So, we have a similar instruction called umull which is an unsigned multiplication. So, this is if you would see this instruction and the upper instructional lower instruction are exactly the same. Only difference is inside of a signed multiplication it is an unsigned multiplication. So, we treat r2 and r3 also an unsigned quantities and you perform in unsigned multiplication. So, the final result will fit within 64 bits. So, lower 32 bits can be saved in r0 and the upper 32 bits can be saved it r1. So, the only difference between smull and umull is smull does a sign multiplication and umull does than unsigned multiplication.

(Refer Slide Time: 28:51)



Example

Compute $12^3 + 1$, and save the result in r3.

Answer:

```
/* load test values */
mov r0, #12
mov r1, #1

/* perform the logical computation */
mul r4, r0, r0 @ 12*12
mla r3, r4, r0, r1 @ 12*12*12 + 1
```

$r4 \leftarrow 12^2$

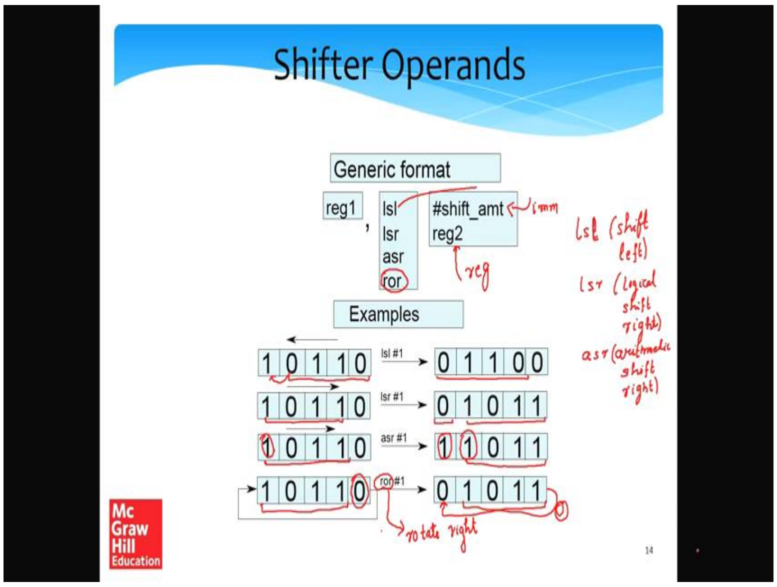
$r3 \leftarrow r4 \times r0 + r1$
$= 12^3 + 1$

Mc Graw Hill Education
12

Now, let us consider examples. So, let as compute 12 cube plus 1 and save the result in r3. So, let us do one thing, let us load the values. So, load the values basically means transfer the values. So, let us transfer 12 to r0 and let us transfer 1 to r1. So, now, let us perform the logical competition in ARM ISA, at symbol, is used to specify that what lies after it is a comment.

So, let us first multiply r0 with r0 which is let us compute 12 square and save it an r4. So, r4 will contain 12 square now here is the greatness of the mla instruction multiply and accumulate. So, here we compute r3 is r4 times r0, plus r1. R4 is already 12 squares and r0 is 12. So, this is 12 cube plus the value in r1 is 1. So, we get 12 cube plus 1. So, what we can see is that using the mla instruction computing 12 cube plus 1 actually became very simple and we could actually do it in 2 assembly instructions after we had the values in there in the registers loaded. So, this is actually you know very cheap and very fast. So, the user should keep this in mind that occasionally you should take a look at some of the advance instructions that ARM provide such as mla smull and umull to make the competition easier, and also reduce the lines of code. Because in assembly language lower is the lines of code more efficient is the program.

(Refer Slide Time: 30:57)



Shifter Operands

Generic format
reg1 , lsl lsr asr ror #shift_amt reg2
imm
reg
lsl (shift left)
lsr (logical shift right)
asr (arithmetic shift right)
Examples
1 0 1 1 0 lsl #1 0 1 1 0 0
1 0 1 1 0 lsr #1 0 1 0 1 1
1 0 1 1 0 asr #1 1 1 0 1 1
1 0 1 1 0 ror #1 0 1 0 1 1
rotate right
14

Now, let us take a look at some of ARMs advanced instructions. So, ARM has a notation of a shifter operand. So let us see what it is, so considered a register. So, we can optionally specify after a register id a shift amount. So, there are 4 ways that we can actually shift a register in ARM. So, it is lsl. So, let me first explain what is there in this table. So, lsl means logical shift left. So, let me just write or maybe I can write here lsl is shift left, lsl is logical shift right. So, this is similar to simple RISC, asr is arithmetic shift right. So, there is no arithmetic shift left. So, that is the reason we have not defined it. So, lsl is the same as the lsl in simple RISC a logical shift left. And so then there are 2 kinds of right shift. So, in the logical shift right we add 0s to the MSB and in the arithmetic shift right we replicate the sign bit. So, given a register we can add these 3 kinds of shifts to it and also we have this ror kind of shift.

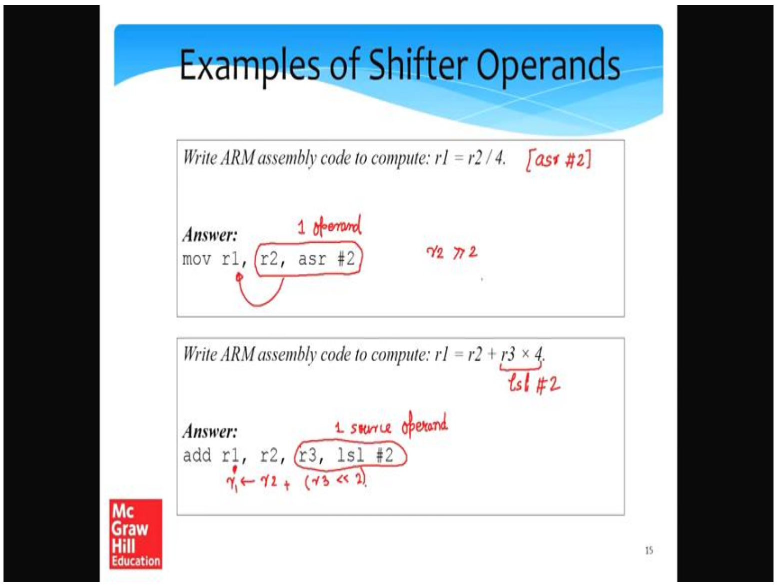
So, will discuss what an ror is when we come to this part of the figure, the lower part of the figure. And then we can shift a register either by a certain shift amount which can be between in anywhere between 0 one 31, or we can shift it by the amount written in a register return in a another register. So, basically we can put in a registered over here, or we can consider this immediate which encroach the shift amount. How much we should shifted by? So, let us consider a simple 5 bit example and let see we want to logically shifted left by one position. So, in this case 0 will come here. So, will have 0 1 1 0 which is essentially you know this part has come over here 0 1 1 0. And we shift in a 0 in a LSB position. So, this a logical shift left.

So, in ARM we will see in the next few slide that it is possible to actually fold in the shift information as a part of the operand specification. So, we do not need a separate shift instruction, but let us first appreciate the types of shifts in ARM first subsequently we have an lsr. So, in lsr if u have 1 0 1 1, so 1 0 1 1 will again get replicated over here in 1 0 1 1 and in the MSB position we will shift in a 0. Similarly, for the arithmetic shift right the bits 1 0 1 1 will get shifted to the right, and in the MSB position will shift in a 1 the reason being that we are replicating the sign bit the sign bit here is 1. So, we are just replicating it is arithmetic shift right.

Now, let us introduce the rotate right instructions. So, ror is rotate right. So, what we do in this case you want to rotated right by one position. So rotate right is basically a right shift with the values that is falling off the least significant position they come and sit in the most significant position. So, let us see. Let us consider 1 0 1 1 if you rotated right the first thing that we do is we shifted to the right by one position. So, 1 0 1 1 over here gets reflected over here. Subsequently what do we write in the MSB.

We write in the MSB whatever was shifted out of the least significant positions in this case 0 were shifted out these are the felt to the right. So, essential 0 comes and it is put in the MSB. So, essentially we are taking a set of bits and we are just is rotating them. So, another important point is that in the shifting operations in lsl lsr and asr, we are actually introducing new bits which were not originally there in the set of bits right. At either the left position or the right position, in comparison in the case of a rotate instruction we are not doing that so, whatever bits fall out of the least significant position are added to the most significant position; now, the given that these 4 shift operations are clear.

(Refer Slide Time: 36:54)



Examples of Shifter Operands

Write ARM assembly code to compute: r1 = r2 / 4. [asr #2]

Answer:

1 operand

```
mov r1, r2, asr #2
```

r2 >> 2

Write ARM assembly code to compute: r1 = r2 + $\underbrace{r3 \times 4}_{lsl\ \#2}$.

Answer:

1 source operand

```
add r1, r2, r3, lsl #2
```

$r_1 \leftarrow r2 + (r3 << 2)$

15

Let us actually take a look at an example of these. So, let us without an example it will not have been cleared.

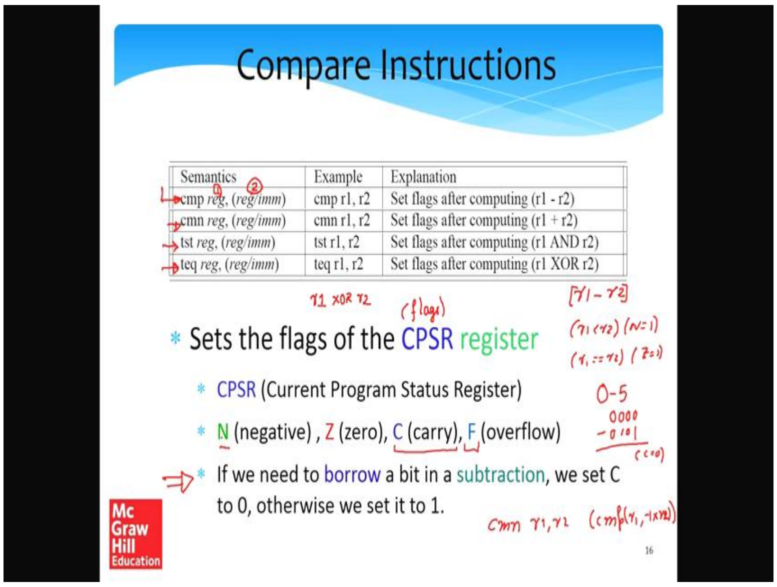
So, let us consider the following example. So, right ARM assembly code to compute r1 equals r2 by 4. So, essentially dividing a number by 4 as was discussed in chapter 2 as same as right shifting it by 2 positions, by this not a logical right shift this is an arithmetic right shift will call it an asr. And we shifted by 2 positions. So, in the case of the ARM ISA we actually do not have shift instructions. So basically in the previous slide what we have seen lsl lsr asr ror their essentially shift directives, but they are not. So, the important distinction that needs to be made is a these are not you know separate instructions in their own right.

So, what we essentially do is that we use the mov instruction and we are treating this entire new expression that you are not seen before as actually one operand. It is true that there is a comma in the middle, but this is actually being treated as one operand. And this is being moved over here. So, what is this doing, what this is doing is that this expression is taking the value of r2, and right shifting it. So, maybe I can write the right shift right shifting it by 2 positions. So, these I am using the standard right shift operator. So, we are right shifting it by 2 positions which is tantamount to equivalent to dividing the number by 4. And then this entire the result of this is being transferred to r1.

So, as I said let me just you know reemphasis that ARM in the most variant. So, of ARM assembly at least most of the simple variance, do not have dedicated shift instructions. Rather the shift is folded into the definition of the source operand itself. So, in the source operand, we can specify the source operand as a register. And you know this is not possible to do with an immediate, but as a registered we can specify the source operand and optionally specify a shift amount and the type of shift. So, there will be a comma between the source register and the type of shift in the shift amount by the entire ensemble is treated as a single operand, and what the hardware would do is it would first evaluate the value of this expression which basically mean shift r2 by 2 positions to the right and then transfer it to r1.

So, let us now take a look at a slightly more complicated example. So, let us compute r1 equals r2 plus r3 times 4. So, multiplying a number by 4 is a same as shifting it to the left by 2 position. So, what we will do is will write add r1. So, since the first operand is the destination. So, we will have r1 is being set 2. So, here the entire operand is actually just one source operand right. So, we can treat this as a single source operand so will add r1 as r2 plus r3 left shift it by 2 positions or multiplied by 4 it is a same thing. So, will have r1 is r2 plus 4 times r3 which has been folded into one single assembly statement. And this will make the execution of the assembly statement extremely efficient. So, the important point to note is that ARM does not have separate shift instructions; rather a shift is folded in to the definition of the operand itself, to the specification of the operand itself.

(Refer Slide Time: 41:38)



Compare Instructions

Semantics	Example	Explanation
cmp *reg*, (*reg*/*imm*)	cmp r1, r2	Set flags after computing (r1 - r2)
cmn *reg*, (*reg*/*imm*)	cmn r1, r2	Set flags after computing (r1 + r2)
tst *reg*, (*reg*/*imm*)	tst r1, r2	Set flags after computing (r1 AND r2)
teq *reg*, (*reg*/*imm*)	teq r1, r2	Set flags after computing (r1 XOR r2)

r1 XOR r2 (flags) [r1 − r2] (r1 < r2) (N = 1) (r1 == r2) (Z = 1)

* Sets the flags of the CPSR register
  * CPSR (Current Program Status Register)
  * N (negative) , Z (zero), C (carry), F (overflow)
  * If we need to borrow a bit in a subtraction, we set C to 0, otherwise we set it to 1.

0-5 0000 − 0101 (C = 0)

cmn r1, r2 (cmp(r1, −1×r2))

16

Now, let us take a look at compare instruction. So, they worked very you know you know very similar fashion as simple RISC. So, compare instruction at does the job of comparing and then it is sets the flags. So, let us first take a look at a simple compare instruction the cmp instruction we are a cmp simple RISC as well, and the format was the same the first operand was the register, the second operand was either a registered or an immediate. So, we are comparing r1 and r2 and after this. So, what is comparing r1 and r2 mean it essentially means that we subtract r2 from r1. So, we compute r1 minus r2 and we set the flags after computing r1 minus r2. So, in simple RISC we are a flags register in ARM it is called the cpsr register is the same as what was flag sensible RISC. So, this is cpsr register cpsr register is called the current program status register, which does not a 1 flag it has 4 flags. So, the first flag is negative. So, negative indicate that when I did r1 minus r2, r1 was less than r2.

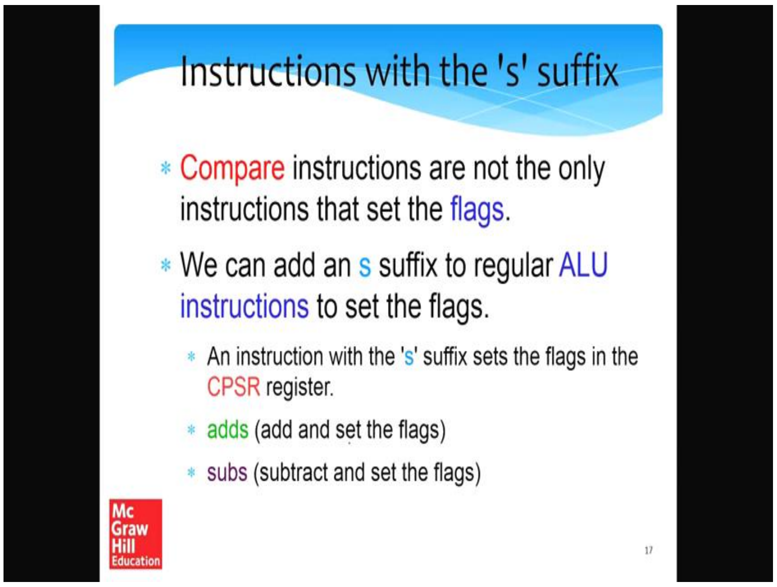
So, in this case the negative flag will be equal to 1. Then we are the 0 flag which means if r1 was equal to, equal to you know same as r2 in a 0 flag will be equal to 1 and the negative flag will be equal to 0. In the carry flag is basically done to indicate that I did not addition and after that I carry was generated. And similarly the overflow flag is set to indicate that the result acceded the range of the number system. As a result it cannot be saved and so the programmer should be told that look the addition or multiplication that you are trying to do acceded the range other number system and as a result there has been an overflow. So, here is an important point that we should need to note it is slightly known intuitive. So, that is the reason I am putting a big arrow to it is left. If we need to borrow a bit in a subtraction, carry is mainly used in the context of addition, but we need to borrow a bit as is in the case of subtraction, then we set the carry flag to 0 rights. So, the moment we need to borrow we said the carry flag to 0 and if you do not need to borrow a bit we set it to 1.

So, this is slightly tricky and non-intuitive, let me repeat it once again what I just said. What I said is that if we need to borrow a bit in a subtraction. So, a subtract 2 numbers as necessary to borrow, we will set the carry flag to 0. For example, if we or subtracting 0 minus 5. So, this in if I write in binary it is the since 0 minus right. So, in this case when I am actually subtracting there is a need to actually borrow bits. So, in this case the carry flag will be set to 0. Otherwise it will be set to 1. So, similar to the compare instruction we have a cmn which is actually called compare negative format is the same, but instead of actually ga setting the flags after computing r1 minus r2, we compute r1 plus r2 or this is the same as we can think of this cmn r1 r2 is essentially equivalent to compare r1 with minus 1 times r2 right. So, that is a reason it is called compare negative were instead of comparing r1 with r2, we are comparing r1 with minus 1 times r2. Similarly, we have the test tst instructions. So, we set the flags after computing r1 and r1. So, if r1 and r2 give the final result is 0 will set the 0 flag or will essentially take a look at the final result and based on that will set the flags.

So, let us say the final result MSB is 1 will set the negative plans and so on. So, teq basically tests if. So what we do is we compute and xor or of r1 and r1. So, we set the flags after computing r1 xor or r2. And we take a look at the result, is a result is all 0s will set the 0 flag otherwise if the if the result has a positive sign bit will set the negative flag. So tsp teq and cmn that typically not very commonly use even you can use them whether situation for demands, and if you feel let we can reduce the number of instructions by using these flags. So the ultimate reference for any kind of ARM instruction is the ARM instruction reference manual, which we can get on ARMs website. So they are will find the lot of these concepts explain in great detail. So, the reason that I am not going into more depth is basically because some of them have many cases and sub cases.

So, this is better dealt with in a manual which describes all the cases in great detail. So, v for most of our work will only strict to the compare flag. I am sorry the compare instructions and the compare instructions will tell as using the negative and 0 flags, which is very similar to what we had in simple RISC to basically the 0 flag was the equality flag, and if r1 is greater than r2 were setting the g t flag. So, in this case the flags are different with a connotation is similar. So, we will use mainly the compare instructions to do our job.

(Refer Slide Time: 48:20)



Instructions with the 's' suffix

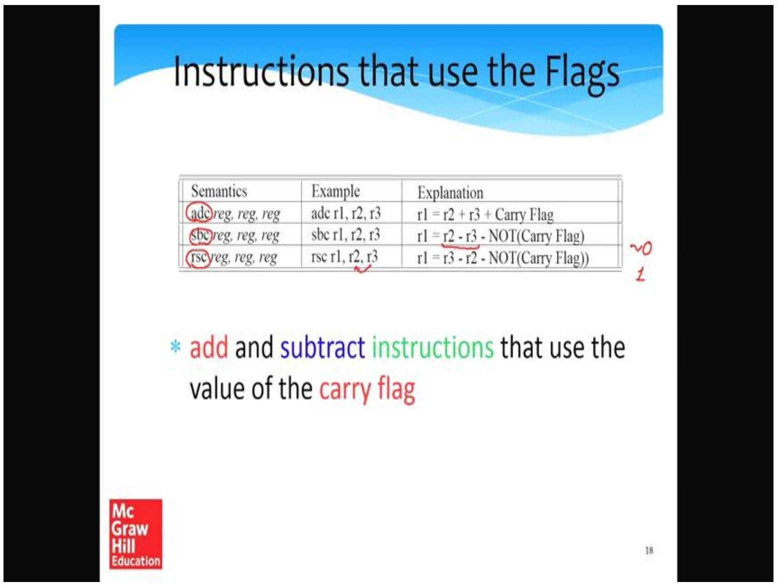
- Compare instructions are not the only instructions that set the flags.
- We can add an s suffix to regular ALU instructions to set the flags.
  - An instruction with the 's' suffix sets the flags in the CPSR register.
  - adds (add and set the flags)
  - subs (subtract and set the flags)

17

So, know ARM has some more interesting instructions. So, the compare instruction are not the only instructions that set the flags rights, in simple RISC cmp was the only instruction that was setting the flags per in ARM that is not the case you can add an s suffix to regular alu instructions such that they will set the flag.

So, in the instructions with the s suffix will set the flags in the cpsr register. For example, if I let us the add 2 numbers. So, I can replace the add instructions with the add s instructions. So, the add s or the sub s instructions will actually set the flags. So, they will perform the addition or the subtraction. Subsequently based on the result they will set the flags whether number is negative or 0 or if the addition let to an over flow. So, all of this condition can be set.

(Refer Slide Time: 49:22)



Instructions that use the Flags

Semantics	Example	Explanation
adc *reg, reg, reg*	adc r1, r2, r3	r1 = r2 + r3 + Carry Flag
sbc *reg, reg, reg*	sbc r1, r2, r3	r1 = r2 - r3 - NOT(Carry Flag)
rsc *reg, reg, reg*	rsc r1, r2, r3	r1 = r3 - r2 - NOT(Carry Flag))

~0
1

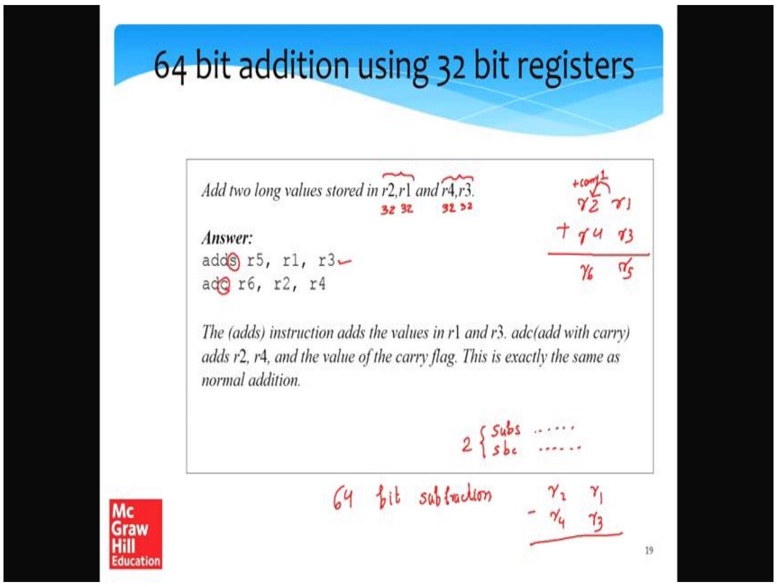
* add and subtract instructions that use the value of the carry flag

18

Now, let us take look at set of very interesting instructions and actually use the flags. So, the first instructions that will look at is adc the adc is an add with carry instruction. So, in this case we do add with carry is r1 equals r2 plus r3 plus the carry flag rights. If there are some previous carries. So, this adc instruction we shall will find an example in the book that uses it, but essentially the idea is that if previously a carry was generated then we can sort of use the carry. So, basically we will add r2 and r3 similar to the add instruction and also add the carry. Similarly, we have sbc instruction subtract with carry. Which first subtract r2 minus r3 and then it subtracts it with the knot of the carry flag. So, this is slightly tricky. So, what did we say let us go back to the slide that talked about the borrow bit. So, if a need to borrow a bit in a subtraction reset carry to 0 otherwise we set it to 1. So, in this case in the previous subtraction effect let to a borrow the carry flag is 0. So, knot of the carry flag is actually 1.

So, we what we do is that this is sort of giving effect to a borrower implementing borrow. If do the regular subtraction and then we subtract a knot of the carry flag and knot of the carry bit can be considered the borrow bit, because their connotation are exactly rewards. So, you will find an example in the book for we use the adc and sbc instructions to actually subtract large 64 bit or even larger quantities. The rsc instruction is similar to rsb were we do a reverse subtract, but in this case we do a reverse subtract which is r3 minus r2 r3 minus r1, but we also subtract the borrow bit which is the knot of the carry flag from this.

(Refer Slide Time: 51:47)



64 bit addition using 32 bit registers

Add two long values stored in $\overbrace{r2,r1}^{}$ *and* $\overbrace{r4,r3}^{}$*.*

32 32 32 32

Answer:

```
adds r5, r1, r3
adc r6, r2, r4
```

The (adds) instruction adds the values in r1 and r3. adc(add with carry) adds r2, r4, and the value of the carry flag. This is exactly the same as normal addition.

$$\begin{array}{rcc} & +\text{carry } 1 & \\ & r2 & r1 \\ + & r4 & r3 \\ \hline & r6 & r5 \end{array}$$

$2 \begin{cases} \text{subs } \ldots\ldots \\ \text{sbc } \ldots\ldots \end{cases}$

64 bit subtraction

$$\begin{array}{rcc} & r_2 & r_1 \\ - & r_4 & r_3 \\ \hline \end{array}$$

19

So, here is one example, that will show you the power of what the carry flag and the add s and adc instructions do. So, this typically I used to set as an exam question and I ask to use students to do this is simple RISC. So, this was very difficult and is used to take a lot of lines, because you know simple RISC did not have the support to do it. So, what do you want to do you want to do 64-bit addition using 32 bit integers. So, consider the smull and the umull instructions. So, they save the result in a pair of registers. So, let us assume this is the case let us assume that is 64-bit value which is called also called a long value is stored in registers r2 and r1. So, r1 contains the lower 32 bits and r2 contains the upper 32 bits. Similarly, we have another register pair r4 and r3 were r3 contains the lower 32 bits and r4 contains the upper 32 bits. And together they make a large 64-bit number. So, we want to add this pair of numbers this long numbers and also save them in 2 registers r5 and r6. So, what we do is first.

So, let us say the number are of this form r2 r1 and r4 r3 you want to add them. So, the first thing that can be done is we can add r1 and r3 and save the result in r5. This is exactly being done over here, but note the s suffix with the add instruction. The s suffix is basically telling the add instruction that look you go and set the flags. So, here which flag are interested in we are interested in only a single flag which is actually the carry flag. So, what is happening is that if the carry bit is set, we need to record this fact in this is being recorded in the flags, that if there is a one bit carry this needs to be recorded. Subsequently we add r2 and r4, but we use the adc instruction because this add r2 and r4 plus the carry. So, that is a very important thing is also takes the carry into account which might have been generated, it adds r2 r4 and they carry and saves the result in r6 right.

So, what is written is the add s instruction as the values in r1 r3. Adc add with carry as r2 r4 and the value of the carry flag, which is exactly the same as normal addition had. Add s and adc not been there it would have been fairly difficult for us to achieve this task and so considered a program that has a lot of you know smull or umull instructions. And they produce 64 bit outputs. To manage the 64 bit outputs with you know to manage them and add and subtract them would have been very difficult had we not had this particular mechanism, given the fact that we have seen this actually.

Let me go back to the previous slide given the fact that you have taken a look at 64 bit, addition I would request the readers to also look at 64-bit subtraction. So, in the case of 64-bit subtraction idea is very simple we do exactly this. So we will have exactly the same thing and instead of addition will have a subtraction. So, first instead of an add s let me just give a hint, but I will not tell you the entire solution. So, we can instead of an add s we can have a sub as you can write something, and subsequently we need to go back to a set of instructions and use the sbc instruction the sbc instruction will subtract with carry which will take the borrow into account, and we can do something and achieve also achieve subtraction in 64-bit subtraction in 2 instructions.