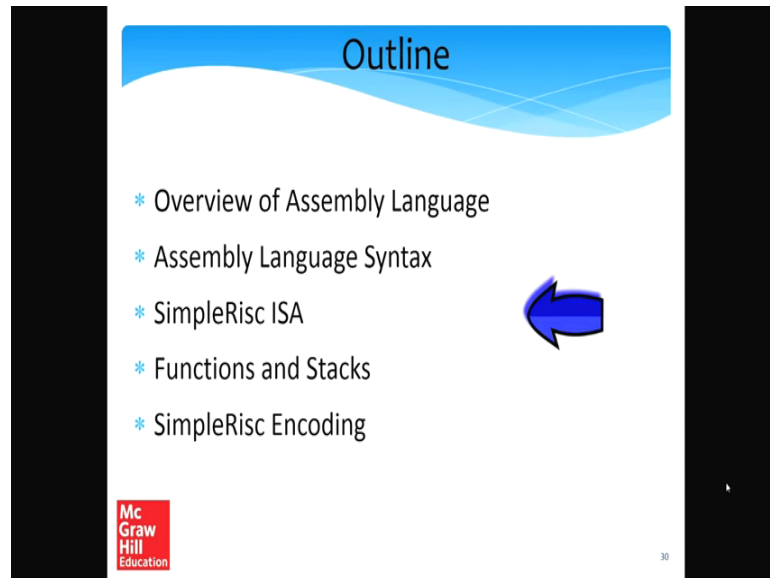


Computer Architecture
Prof. Smruti Ranjan Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

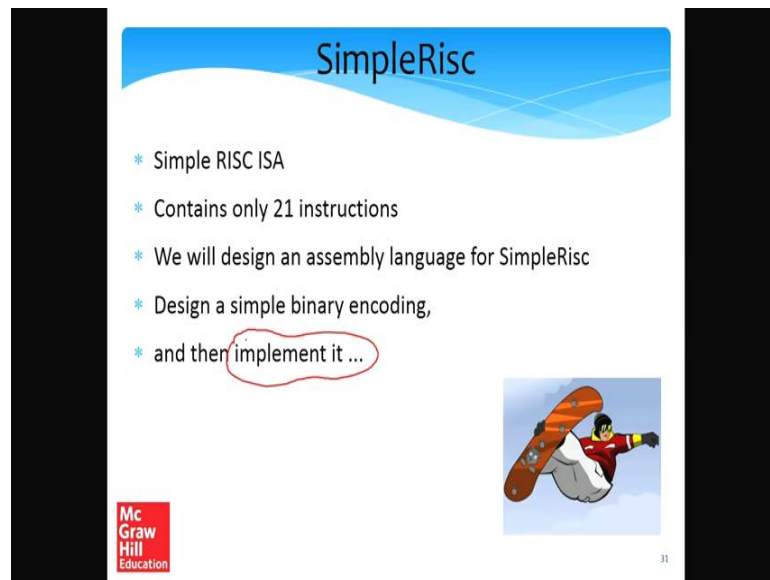
Lecture - 06
Assembly Language Part-II

(Refer Slide Time: 00:25)



Let us now discuss the Simple RISC ISA. So, Simple RISC is an assembly language that we shall create, you and me shall create from scratch. So, this is supposed to be a very very Simple RISC like language, that has a very few instructions, and the instructions have a very regular structure. So, as we go through the next few slides we will see what it actually takes to build and design an assembly language and we shall look at the netegrities of designing one such language.

(Refer Slide Time: 01:04)



The slide features a blue header with the text 'SimpleRisc'. Below the header is a list of five bullet points, each preceded by an asterisk. The last bullet point, 'and then implement it ...', is circled in red. To the right of the text is a small illustration of a snowboarder in mid-air. In the bottom left corner, there is a red square logo with the text 'Mc Graw Hill Education'. In the bottom right corner, the number '31' is visible.

- * Simple RISC ISA
- * Contains only 21 instructions
- * We will design an assembly language for SimpleRisc
- * Design a simple binary encoding,
- * and then implement it ...

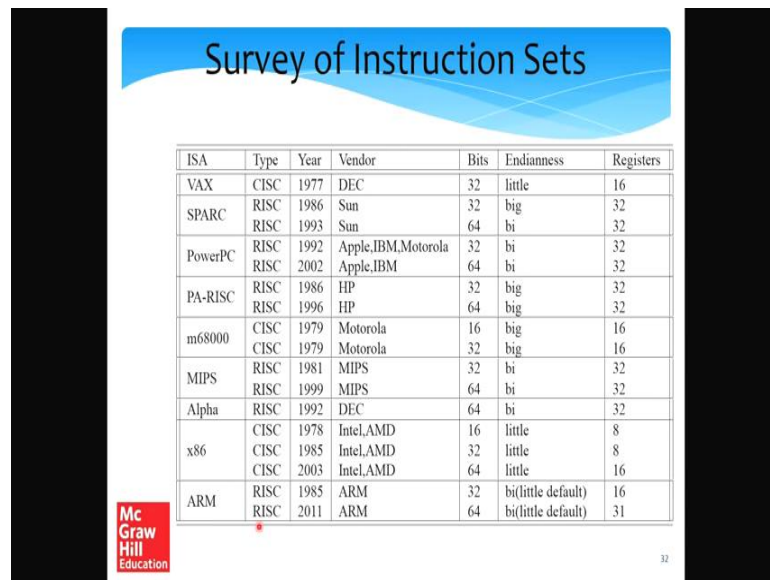
Mc Graw Hill Education

31

So, let us take a look at Simple RISC. So, Simple RISC is a very simple you know as a name suggests, it is a very very simple assembly language one of the easiest to learn. So, my thinking is that it is better to learn a very simple assembly language and get an idea first. Before moving on to real world assembly languages such as ARM and x 86, because they have a lot of other details which might overwhelm the retain at this point of time.

So, we will design an assembly language called Simple RISC and then we will design a simple binary encoding for it which means to convert instructions into sequences 0s and 1s and then in the processor chapter we will create a processor to actually implement the Simple RISC instruction set.

(Refer Slide Time: 01:59)



The slide features a blue header with the title "Survey of Instruction Sets". Below the header is a table with 7 columns: ISA, Type, Year, Vendor, Bits, Endianness, and Registers. The table lists various instruction sets from 1977 to 2011. Logos for "Mc Graw Hill Education" and a small "32" are visible at the bottom of the slide.

ISA	Type	Year	Vendor	Bits	Endianness	Registers
VAX	CISC	1977	DEC	32	little	16
SPARC	RISC	1986	Sun	32	big	32
	RISC	1993	Sun	64	bi	32
PowerPC	RISC	1992	Apple,IBM,Motorola	32	bi	32
	RISC	2002	Apple,IBM	64	bi	32
PA-RISC	RISC	1986	HP	32	big	32
	RISC	1996	HP	64	big	32
m68000	CISC	1979	Motorola	16	big	16
	CISC	1979	Motorola	32	big	16
MIPS	RISC	1981	MIPS	32	bi	32
	RISC	1999	MIPS	64	bi	32
Alpha	RISC	1992	DEC	64	bi	32
x86	CISC	1978	Intel,AMD	16	little	8
	CISC	1985	Intel,AMD	32	little	8
	CISC	2003	Intel,AMD	64	little	16
ARM	RISC	1985	ARM	32	bi(little default)	16
	RISC	2011	ARM	64	bi(little default)	31

So, let us have a brief survey of instruction sets say almost every processor vendor has created its own instruction set. So, let us go back to way back to 1977 when the VAX instruction set was created. It was a very complex CISC instruction set with a lot of instructions, it had 16 registers and it used a little Endian format, then one of the more popular instruction sets was a Sun's SPARC instruction set. So, there were actually 2 variants of SPARC; the 32 bit variant was introduced in 1986, and the 64 bit variant was introduced in 1993.

So, what 32 bits means? That 1 instruction was 32 bits in SPARC the size of a register was 32 bits. So, that is what it means to be a 32 bit instruction set and what it means to be a 64 bit instruction set is also the same size of a register is 64 bits, or the size of an instruction in the case of a RISC architecture is 64 bits. So, the 64 bit instruction set was actually bi Endian means, it could be considered to be either little Endian or big Endian and where 32 registers; then we come to the PowerPC instruction set, which was mostly used by IBM and later Apple and Motorola, used it.

So, PowerPC showed a similar revolution, they had a 32 bit instruction set in 1992 and then migrated to a 64 bit instruction set later on and both of them were bi Endian with 32 bit with 32 registers. Similarly HP had its own instruction set pa RISC, that is still used in high end HP servers and Motorola had its m68k or m68000 instruction set, that is still well that was being used in Motorola processors. So, now, of course, they are not in

the processor business anymore. Another very popular instruction set is the MIPS instruction set, which is also a risk instruction set MIPS has 32 registers. So, now, MIPS is used in some small embedded processors, the most of the embedded processors have actually migrated to Intel and ARM. Here is one more instruction set which is my processors favorite, the reason being that this is the first instruction set that I had learnt; is the DEC alpha instruction set, introduced circa 1992. So, it started out at the 64 bit instruction set with 32 registers; then we come to the x86 instruction set. So, it is actually a family of instruction sets, it is not just one instruction set, but many. So Intel started his journey in 1978 with a 16 bit instruction set that had 8 registers.

So, this x86 instruction set is now an open standard. So, it is used by AMD it is actually used a many other vendors as well, because. So, even since the standard is open, readers can implement their own processors with the x86 instruction set. So around 1985 Intel transition to the 32 bit instruction set, again with 8 registers and then as recently as 2003 both Intel and AMD together migrated to an x86_64 instruction set. Actually AMD was the first to migrate and Intel then migrated say AMD came up with a octoroon processor, that had a 64 bit instruction set called x86_64 .

So, all Intel instruction sets have been brittle endian, and the 64 bit instruction set of Intel which we shall also take a look at in the fifth chapter of this book has 16 registers. Lastly the latest entrant in the world of processors and instruction sets is ARM. So, ARM is a very interesting company in the sense that ARM does not makes it is own chips, rather ARM makes the design of a processor and sends the design to other companies like Texas instruments, or Volcom or Samsung and then they make a chip and embed some additional elements if required and then they make an fabricated chip out of it.

So, ARM is not that new. In fact, ARM used to exist way back in 1985, when they came up with the 32 bit instruction set. But off late ARM processors are gaining in prominence and a big reason for that, is that ARM processes are used in mobile phones and mobile phones have spread exponentially. Say in 2011 ARM also came up with a 64 bit instruction set for larger computing devices, such as tablets and you know possibly servers and. So, this 64 bit instruction sets are always targeted for larger devices.

So, we will see why. So, we will we will have a lot of opportunities to discuss what is the advantage of a 64 bit instruction set, but this is not the right point for it. So, ARM is also

bi endian, little Endian is the default, but it can be considered to be big Endian. So, big Endian or little Endian is just a matter of convention.

(Refer Slide Time: 08:13)

Registers

- * SimpleRisc has 16 registers
 - * Numbered : r0 ... r15
 - * r14 is also referred to as the stack pointer (sp)
 - * r15 is also referred to as the return address register (ra)
- * View of Memory
 - * Von Neumann model
 - * One large array of bytes
- * Special flags register → contains the result of the last comparison
 - * flags.E = 1 (equality), flags.GT = 1 (greater than)

Handwritten notes in red ink:
cmp r1, r2 r1 == r2, { flags.E=1
 r1 > r2, { flags.GT=1

Mc Graw Hill Education 33

Let us now discuss the basics of Simple RISC instruction set. So, let us Simple RISC has 16 registers, and as is that convention let us number the registers from r0 till r15. So, the 14th and the 15th register, r14 and r15 let me refer to them with special names.

So, let us refer to r14 as the stack pointer or sp. Similarly let us refer to r15 as the return address register or ra. So, the meaning of the term stack pointer and the return address register will be is they actually correspond to pretty complex concepts. So, we will discuss what they are in the future slides, but let us just remember for the time being that the stack pointer sp is r14, and r15 is we will also be referred towards the return address register. So, they have some special meanings.

So, coming to the view of memory, let us assume a von Neumann model. So, we will assume that the memory is 1 larger array of bytes, it contains data as well as the program; and a program assumes the entire memory is for it is own personal consumption and run over their programs. So, let us play with some very very simplistic assumptions for the time being. So, let us also have a special flags register, which contains the result of the last comparison. So, let me give an example. So, this register is not visible to the assembly programmer.

However it will be visible to the hardware designer and that is very important. So, let us do let me show an example here. So, the example is let us have a compare in Simple RISC assembly, where we compare the contents of register r 1 and r 2. If the contents are equal, what the hardware would do is that it would set the e bit inside the flags register, say if let us say you know r 1 is equal to let me put a c style equal to. If this is the case then you set the hardware sets, flags dot E equal to 1. If r 1 is greater than r 2, well then flags dot e. So, the default e and g t 0, so I am not mentioning that is the e v a field in this case is clearly 0.

So, we set flags dot G T greater than as 1. If r 1 is less than r 2, so clearly the quality flag flags dot E is 0, and flags dot GT is 0. So, we can thus make out what is the result of the last comparison we are taking a look at these 2 flags: flags dot E and flags dot GT. So, this is the broad idea, but we will get a chance to discuss the meaning of these 2 flags in great detail. So, when we look at the compare instruction.

(Refer Slide Time: 1 1:44)

mov instruction

mov r1,r2	r1 ← r2
mov r1,3	r1 ← 3

- * **Transfer** the contents of one **register** to another
- * Or, transfer the contents of an **immediate** to a register
- * The value of the **immediate** is embedded in the instruction
- * SimpleRisc has 16 bit immediates
- * Range -2^{15} to $2^{15} - 1$

opcode dest, src
mov r1, 3
/* r1=3 */

34

Now, it is time to take look at our first assembly instruction. So, let us introduce our first basic assembly instruction, it is called the mov instruction. The job of the mov instruction is very simple. So, it comes in 2 flavors. So, in one flavor we transfer the contents of a register into one more register, and in the second flavor we transfer an immediate a constant to a register right. So, these are the only 2 variants of the mov instructions that are possible, and the way that we actually write an assembly statement is

like this that we first. So, in Simple RISC, what we do is that we first write the name of the assembly opcode which in this case is mov.

Then we write the destination, and then we give a comma, and then we write the name of the source. So, for example, if we write mov r 1 comma 3, which essentially means set r 1 equal to 3, that is what it means right. Let me also put comment sign to indicate that you know this is not a valid assembly statement. So, what we are doing here is that we are taking the value 3 and setting it to register r 1. So, you can think of this as similar to any statement in c or java, where we write something e on the form x equals 3. So, in this case x is the variable and we are setting the value of the variable 2 3 or 4 or 5 or any immediate value, also we can have statement x equals y. So, in this case this is same as mov r 1 r 2, but the contents of r 2 are mov to r 1.

So, the value of the immediate in this case is actually embedded inside the instruction. So, Simple RISC supports 16 bit immediate and the range of an immediate is. So, this is the twos complement number and the range of an immediate is from minus 2 raise to the power 15, to 2 to the power 15 minus 1. So, this is the standard twos complement range for a 16 bit number.

(Refer Slide Time: 1 4:27)

Arithmetic/Logical Instructions

* SimpleRisc has 6 arithmetic instructions

* add, sub, mul, div, mod, cmp

add reg, reg, (reg/imm)
r₁ = r₂ + 10

Example	Explanation
add r1, r2, r3	r1 ← r2 + r3
add r1, r2, 10	r1 ← r2 + 10
sub r1, r2, (r3/imm)	r1 ← r2 - r3
mul r1, r2, r3	r1 ← r2 × r3
div r1, r2, r3	r1 ← r2 / r3 (quotient)
mod r1, r2, r3	r1 ← r2 mod r3 (remainder)
cmp r1, r2	set flags

r₁ - r₂ = 0

r₁ - r₂ > 0

flags Z=1

flags GT=1

mul r₁, r₂, 10

r₁ ← r₂ × 10

[X = Y + Z]

Sub r₁, r₂, 10

r₁ ← r₂ - 10

flags ← Comp...

⋮

cmp

35

So, now let us look at another family of instruction, since the entire family is very similar I have listed all 6 together in the same place.

So, Simple RISC has 6 arithmetic instructions and as the name suggests their addition, let me just change the pointer; addition, subtraction, add sub, mul for multiplication, div for division, mod for getting the remainder right. So, this is basically getting the modulo or the remainder, and cmp is for compare. So, the compare instruction is slightly special, but the rest are very straight forward.

So, let us take a look at the add instruction first. So, the add instruction is actually a 3 address format instruction, what this essentially means is that it takes 3 arguments. So, the destination always has to be a register, the second argument has to be a register and so basically if you want me to write this slightly formally so the destination which comes first has to be a register. The second operand which is actually the first source operand has to be a register. The third operand can either be a register or an immediate. So, that is a constraint also on the third operand which can either be a register or an immediate. So, here as if we write add r 1, r 2, r 3 essentially what we are instructing machine to do is take the contents of r 2, take the contents of r 3 add them and save the result in r 1.

Similarly, if I say r 2 and 10, I can say r 2 plus 10. So, this is the simple add instruction which has a nice analogue with traditional programming languages, where if you write you know say c function of a form x equals, y plus z, we can directly translate it to an assembly instruction of this form fine. So, after the addition instruction the rest 3 the rest 4 instructions sub, mul, div and mod have exactly the same format. So, I have not shown the immediate values over here, but I will explain with examples. So, these 4 instructions over here as the name suggests the subtract instruction subtracts the contents of r 3 from r 2; so it sets r 1 equal to r 2 minus r 3.

So, in this case it is possible to write the subtract instruction with the third operand being an immediate as well, but mind you only the third operand can be an immediate not the other 1s. So, I can write very well r 1, r 2 and 10. So, what this will essentially do is that it will set r 1 to r 2, the contents of r 2 minus 10. So, I have not shown this case for lack of space, but the third operand can either be a register or it can be an immediate any register or it can be a immediate. Same is the case for multiplication exactly the same format.

So, let me write a multiplication instruction with the third operand as an immediate. So, we shall see that the way of specifying the format is the same. So, I can very well write

mul r 1, r 2 and 10. So, this would set r 1 equal to r 2 multiplied with 10. So, divisional mod have also the same format where we divide r 2 by r 3 to get the quotient and we take r 2 mod r 3 to get the remainder. So, these are the 5 adds or mul div and mod other 5 arithmetic operator operating instructions, and as we can see there are very similar to their high level programming equivalence. So, it is just that, they written slightly differently.

So, one thing that is common is that even in a high level language such as you know the statement over here, the destination comes first subsequently the arguments come later. This is a same case in assembly as well where the destination comes first and the operands or arguments come later. So, this is one similarity. The other similarities are that we also you know we clearly mention what is the operation that we want to perform. So, either it is an add or subtract or multiplication, division whatever is the operation that is you know that is placed at a different point. So, plus is over here, but in assembly we first mention what is the operation. but you know other than this small difference this statements like add r 1, r 2 r 10 is pretty similar to what we would actually write in a high level programming language, assuming r 1 and r 2 are variables we would write r 1 equals r 2 plus 10.

So, now let us take a look at the compare instruction which is actually very interesting because it is different. So, we typically do not write compare instructions of this type in a high level language. So, in this case what we do is that we pretty much subtract r 1 minus r 2. If the result is equal to 0 then we can infer an equality. So, we set flags dot E, which is the equality flag to 1. If r 1 minus r 2 is greater than 0, then we can set flags dot GT equal to 1 which basically means that r 1 is greater than r 2, it indicates the fact that the greater than flag is 1.

Similarly, if r 1 minus r 2 is less than 0, then both the equality flag and the greater than flag both will be set to 0. So, we can infer a less than condition. So, it is important to note that in Simple RISC, the only instruction that sets the flags is the compare instruction and the flags remain in their state till the next compare instructions. So, let me explain with an example assume we have a compare instruction. So, the compare instruction is essentially go and set the flags. After that we can have many many other instructions, but till the next compare instructions the flags maintain their value and they can be used by other instructions as we shall see later.

(Refer Slide Time: 22:09)

Examples of Arithmetic Instructions


- * Convert the following code to assembly

```
a = 3  
b = 5  
c = a + b  
d = c - 5
```

- * Assign the variables to registers

- * $a \leftarrow r0, b \leftarrow r1, c \leftarrow r2, d \leftarrow r3$

```
mov r0, 3  
mov r1, 5  
add r2, r0, r1  
sub r3, r2, 5
```



36

Now, let us take a look at very simple program, and let us try to convert it into assembly language. So, let us outline a generic method. So, assume that we have this codes that are here in a high level language, the question is how do we convert it to assembly language. So, the first thing that we do is that we assign the variables here to registers that is the first thing that is always what we do that is the first thing that we assign the variables to registers. So, let us assume that variable a is assigned to r 0, b to r 1, c to r 2 and d to r 3 right. So, this is the assignment of variables to registers. So, now, what we do is that a equals 3 just becomes mov 3 to r 0, b equals 5 becomes mov 5 to r 1, c equals a plus b this essentially means add r 0 and r 1, and less assign it to r 2 because r 2, and c are when r 2 and c are r 2 is supposed to contain the value of c.

Then to subtract c minus 5 again take r 2, subtract 5 from it and save the result in r 3 which is essentially map to d. So, here we have 4 high level statements and 4 assembly statements, because it is a simple program. But in general the number of assembly statements will typically be more sometimes much more, but the important thing to note that it is fairly simple to map a high level program to assembly, what essentially needs to be done is that the first. So, there are 2 steps: first is assign the variables to registers that is step 1, step 2 is that after the assignment each and every high level statements gets converted to assembly.

So, the important point that the all of you need to know that writing assembly programs is pretty much similar to writing high level programs, and irrespective of how many books you read and how many online courses you take such as this one, the only way that a student will actually is the ability to write high level assembly code, is to you know physically go and write as many assembly programs as possible, practice makes perfect.

So, only by writing more and more and more programs, would a student actually get some expertise of how to write assembly programs, and would get the confidence of writing assembly programs. So, the codes and books and other material is fine to at least introduce the concepts, but the real knowledge will be gained only by writing and by practice. So, if we go to the website of this book you will find simple disk emulators, which allow you to write assembly programs and subsequently the emulator will run the assembly program instruction by instruction and finally, print the result.

(Refer Slide Time: 25:21)

The slide is titled "Examples - II" and contains the following content:

- * Convert the following code to assembly

```
a = 3
b = 5
c = a * b
d = c mod 5
```

- * Assign the variables to registers
- * $a \leftarrow r0, b \leftarrow r1, c \leftarrow r2, d \leftarrow r3$

```
mov r0, 3
mov r1, 5
mul r2, r0, r1
mod r3, r2, 5
```

The slide also features the McGraw Hill Education logo in the bottom left corner and the number 37 in the bottom right corner.

So, let us now take a look at one more example, a equals 3 b equals 5 multiply a and b and compute c mod 5. So, again the first stage is assign the variables to registers, say a you know r 0 is mapped to a, r 1 is mapped to b, r 2 to c, and r 3 to d. So, the first 2 statements are the same as what they were in the previous example, then we multiply r 0 and r 1 and we save the result in r 2. So, r 2 is c and then again we do r 2 mod 5. So, I am

putting the percentage because that is what it is in c, and we do $r \ 2 \ \text{mod} \ 5$ and save the result in r 3.

(Refer Slide Time: 26:13)

The slide is titled "Compare Instruction" and contains the following content:

- * Compare 3 and 5, and print the value of the flags
- Code block 1:

```
a = 3
b = 5
compare a and b
```
- Code block 2:

```
mov r0, 3
mov r1, 5
cmp r0, r1
```
- * flags.E = 0, flags.GT = 0
- Handwritten note: $3 < 5$
- Mc Graw Hill Education logo
- Page number: 38

Now, let us take a look at the compare instruction; the compare instruction is different it is a different concept, it is not there in high level languages. So, let us see we want to compare 3 and 5 and then you know the idea compare 3 and 5 and then we want to print the value of the flags not in assembly, but at least as a part of the answer. So, you should actually we write the value of the flags. So, assume a is 3, and b is 5; so we first to a register assignment, we mov 3 to r 0, 5 to r 1 and compare r 0 and r 1.

Since 3 is less than 5 we will have the less than condition right. So, what we have is that 3 is less than 5, so there is no equality and there is no greater than; so the equal flag will be 0 and the greater than flag will be 0.

(Refer Slide Time: 27:10)

Compare Instruction

- * Compare 5 and 3, and print the value of the flags

```
a = 5
b = 3
compare a and b
```

```
mov r0, 5
mov r1, 3
cmp r0, r1
```

5 > 3

- * flags.E = 0, flags.GT = 1

Mc Graw Hill Education

39

Now let us compare 5 and 3. So, in this case they get reversed. So, we can clearly see that 5 is greater than 3. So, in this case the flag for equality is 0 with the flag for greater than flags dot GT is equal to 1. So, this essentially means that the comparison yielded a result and the result was that the first operand is greater than the second operand. So, that is the reason the GT flag flags, dot GT is equal to 1.

(Refer Slide Time: 27:51)

Compare Instruction

- * Compare 5 and 5, and print the value of the flags

```
a = 5
b = 5
compare a and b
```

```
mov r0, 5
mov r1, 5
cmp r0, r1
```

(5 = 5)

- * flags.E = 1, flags.GT = 0

Mc Graw Hill Education

40

Lastly let us compare 2 numbers with the same value 5 and 5, a is 5 and b is 5, let us compare a and b. So, mov 5 to r 0 we mov 5 to r 1, we compare r 0 and r 1. So, flags dot

E in this case is equal to 1, and flags dot GT is equal to 0. The flags dot E is equal to 1 because there is an equality, we clearly see that 5. So, I am writing the equal to the same way that is written in c equal to equal to. So, we clearly see that 5 is equal to equal to 5, there is equality there is no greater than. So, the greater than flag is 0 the equality flag is 1.

(Refer Slide Time: 28:39)

Example with Division

Write assembly code in SimpleRisc to compute: $31 / 29 - 50$, and save the result in r4.
Answer:

```
SimpleRisc
mov r1, 31
mov r2, 29
div r3, r1, r2
sub r4, r3, 50
```

Mc
Graw
Hill
Education

41

Now, let us show some more examples. So, let us say right assembly code is Simple RISC to compute 31 divided by 29 minus 50 and save the result in r 4. No problem, so mind you this is the integer division, so 31 divided by 29 the answer will be 1. So, let us map them let us map 31 to r 1, let us put 29 in r 2, as can be seen in these 2 statements, then let us divide r 1 divided by r 2 and the result will get saved in r 3 and from r 3 we subtract 50, so we compute r 3 minus 50 is equal to r 4. And r 4 contains the final result as mentioned in the question. So, this is how we write assembly programs, but these are very simple examples of assembly programs.

So, my advice to the student of the reader would be, to actually go and write as many assembly programs as possible and I would say that by the end of this course the student should have written at least 3000 lines of assembly programs, 1000 each in Simple RISC ARM x 86 that would at least give the student a basic understanding of how assembly programs are to be written.

(Refer Slide Time: 30:08)

The slide is titled "Logical Instructions". It contains a table of instructions, a handwritten binary example, and a problem statement.

and r1, r2, r3	r1 ← r2 & r3
or r1, r2, r3	r1 ← r2 r3
not r1, r2	r1 ← ~r2
& bitwise AND, bitwise OR, ~ logical complement	

Handwritten binary example:

$$\begin{array}{r} 1100 \\ 20101 \\ \hline 0100 \end{array}$$

* The second argument can either be a register or an immediate

Handwritten note: not r1, r2

Handwritten binary example:

$$\begin{array}{r} 0010 \\ \hline 1101 \end{array}$$

Compute (a | b). Assume that a is stored in r0, and b is stored in r1. Store the result in r2.
Answer: SimpleRisc

McGraw Hill Education logo and page number 42 are also visible.

Now, let us move ahead in our journey of introducing instruction, so the introduced 3 more. So, these are logical instructions which compute a bitwise and an or, or a not. So, let us look at some of these. So, the and or operations are exactly in the same way in the 3 address format, as the add, subtract, multiply operations. So, in this so the first 2 operands have to be registers and the third operand has to be either a register or an immediate. So, the and operand computes r1, is equal to r2 and r3, where this is a bitwise and. So, let me give an example of a bitwise and, in just case people have forgotten. So, let us say 1100 in a bitwise and it with 0101.

So, 1 and 0 is 0, 0 and 0 is 0 1 and 1 is 1, 0 and 1 is 0. Similarly we have bitwise or which can do r2 or r3 and. So, we have a bitwise or over here, which is exactly the same, but we replace the and sign by an or sign; then we have the not operand which is actually a 2 address format operands, we have only one source, we have only one source operand which can be either a register or an immediate. So, in this case what we do is that sorry. So in this case, if there is a single source operand, so what we can see as a general rule that only one source operand is allowed to be an immediate, but both the source operands are not allowed to be immediate.

So, when we say not r1, r2 what this essentially means is that let us say the value of r2 is 0010. We take a not of it, once we take a not of it this becomes 1101. So, the second argument in this case can either be a register or an immediate, the second source

argument. So, now, let us compute a or b so. So, now, let us consider an example, say in this example we compute a or b. So, assume that a is stored in r 0, and b is stored in r 1. So, let us store the result in r 2. So, it is very simple we do it the same way as our add, subtract, multiply instructions we compute or r 2, which is essentially equal to r 0 or r 1. So, this is similar to r 0 or r 1.

(Refer Slide Time: 33:17)

Shift Instructions

- * Logical shift left (lsl) (<< operator)
 - * $0010 \ll 2$ is equal to 1000
 - * $\ll n$ is the same as multiplying by 2^n
- * Arithmetic shift right (asr) (>> operator)
 - * $0010 \gg 1 = 0001$
 - * $1000 \gg 2 = 1110$
 - * same as dividing a signed number by 2^n

Handwritten notes on the slide:

For Logical Shift Left: $0010 \ll 2$ is shown as 0010 shifted left to 1000 . The original bits are labeled $2^2, 2^1, 2^0$. The result is 1000 with bits labeled $2^3, 2^2, 2^1, 2^0$.

For Arithmetic Shift Right: $0010 \gg 1 = 0001$ and $1000 \gg 2 = 1110$ are shown. The general formula is given as $N = \sum_{i=1}^n x_i 2^{i-1}$. For right shift by k , $N \gg k = \sum_{i=k+1}^{n+k} x_{(i-k)} 2^{i-1} = \left(\sum_{i=1}^n x_i 2^{i-1} \right) \times 2^{-k} = N \times 2^{-k}$.

McGraw Hill Education logo is visible in the bottom left corner of the slide.

So, let us move ahead and introduce some more instructions that are required. So, let us discuss the shift operations. So, shift operations are something like this that let us first discuss the left shift operation. So, consider the number 0 0 1 0, see we left shift it by 2 which means that every bit moves 2 positions to the left. So, in this case if 0 0 1 0 is being left shifted by 2 positions, so what will we have? What we will essentially have is that the 1 will come over here, 0 will come over here and the 2 new positions will be created in this case we just add 2 more 0s. So, 0 0 1 0 left shifted by 2 is equal to 1 0 0. So, we can clearly see that this number is equal to 2, and this number is equal to 8 in decimal; so left shifting by n positions basically means, that it is the same as multiplying a number by 2 to the power n.

So, why is this the case? So, we shall see in some time. So, let me just give one more example of left shifting, let us assume that the number is 1 0 1 0 and we left shift it by let say 1 position right. So, we left shift the number by shifted to the left by 1 position. Say in this case 0 will come over here, 1 will come over here and a new position will be

created we put a 0 over here, assuming the number is remaining within this 4 bits. Say the reason that this is equivalent to multiplying it by 2 to the power n, is can be found out there are many arguments. So, may be let us starts with the couple of examples. So, let us take a look at this example, this number is 2 in decimal and this number is 8 in decimal.

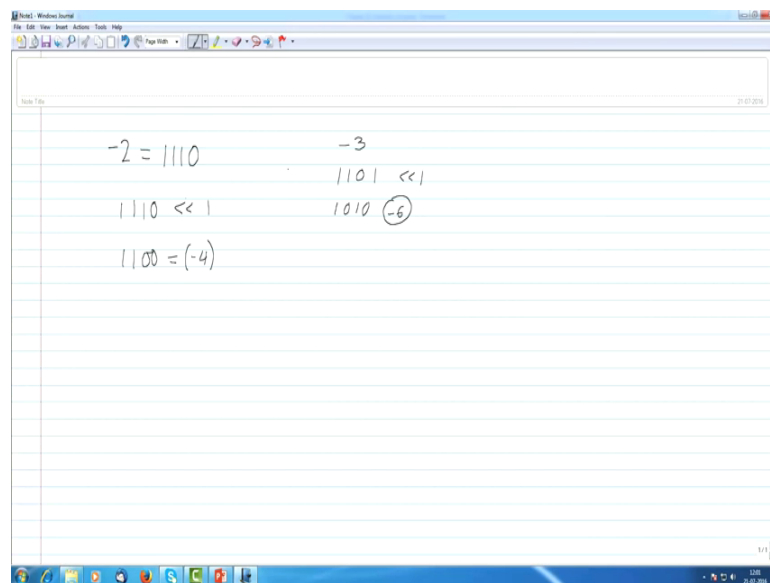
So, what we are essentially doing is that we have left shifted it by 2 positions, which is equivalent to multiplying this number by 2 square. So, the multiplication argument is actually interesting and the reason that you know this holds is that any number in a binary expansion can be thought of as. We will so let us start counting from 1, because we in general we have done that in other places as well. So, assume I goes from 1 to n any binary numbers can be expressed in this form.

Now, if we let say left shift the number by you know k positions. So, what this number essentially becomes? You know that is the most important thing, either this number essentially becomes right. So, basically the number remains the same and it is left shifted by k positions. So, what it essentially becomes is that it is pretty much the same number, but each of these coefficients, if the coefficients remain exactly the same that the set of coefficient set, because it is getting shifted to the left, but these exponents over here pretty much get multiplied get offset by a factors.

So, actually what I can do over here is that I can you know replace this again as it was. So, for those who followed the map over here, what we have done is that we have taken number and we have left shifted it by k positions, but we have not gotten rid off any digits say, then what we have done is this is exactly the same as instead of. So, the first k positions are 0 so we do not count them, then from k plus 1 till n plus k, from k plus 1 till k plus n pretty much we have the same set of coefficients which we had earlier and, but the thing is that the powers of 2, pretty much get shifted or get multiplied by 2 the power k. So, what we do is that we sort of do a little bit of algebra and so we separate the 2 to the power k from the equations and so what we are left with this that this becomes the original number n, multiplied by 2 to the power k. So, what we see is that this is a nice way in unless there are overflows, right we are exceeding the range of the number system, then it is a separate issue; but it that is not happening this is a nice way of actually multiplying a number by any power of 2, multiplying a number by 2 to the power k.

So, let me may be consider let say one more for example, consider 0 01 1; we then left shift it by 1 position. So, this becomes 0 1 1 0; in decimal this number is 3 and this number is 6. So, this is equivalent to multiplying 3 with I am sorry multiplying 3 with 2 to the power 1 so we get 6 all right. So, given that left shifting actually works, let us see if left shifting holds for you know negative twos compliment numbers, here again we assume that the there are no overflows.

(Refer Slide Time: 40:27)



So, let us so instead of cluttering of this power point slide, let me actually move to another software. So now, let us see that whether left shifting actually holds a negative numbers. So, in this case let us consider a negative number minus 2 in a 4 bit system. So, minus 2 is 1 1 0 fine. So, let us take this number and let us left shift it by 1 position. So, then this becomes 1 1 0 0. So, 1 1 0 0 what is it is minus 4 so we see that it holds.

Let us consider a one more example minus 3. So, minus 3 is representation in twos complement notation is actually plus 13. So, which is 1 1 0 1; no problem now let us left shift it by 1 position. So, then what do we get 1 0 1 0. So, what is this? So, this is 8 plus 2 is 10. So, in decimal it is 10 minus 16 the minus 6. So, what we see is that left shifting holds for even positive number as well as negative twos complement numbers.

(Refer Slide Time: 41:59)

Shift Instructions

- * Logical shift left (lsl) (<< operator)
 - * 0010 << 2 is equal to 1000
 - * (<< n) is the same as multiplying by 2^n
- * Arithmetic shift right (asr) (>> operator)
 - * 0010 >> 1 = 0001
 - * 1000 >> 2 = 1110
 - * same as dividing a signed number by 2^n

Handwritten examples on the slide:
0100 >> 2 = 0001
1010 >> 1 = 1101
-> 6

McGraw Hill Education logo and page number 43 are visible at the bottom.

So, now, that we have discussed left shifting, let us discuss what is right shifting; so let us go back to the slides and discuss what is right shifting. So, in right shifting what we do is there are 2 kinds of right shifting; there is arithmetic right shifting and logical right shifting. So, in arithmetic right shifting which is represented by another operator, if we right shift this number by 1 position, what we do is that all the bits move 1 position to the right, but the important point is the sign bit. So, the sign bit in this case is replicated. So, the sign bit was originally 0 we just replicate the sign bit and have one more 0 and in the sign bit was originally 1 we just replicate the sign bit and have a 1.

So, what my claim is that right shifting is similar to dividing a sign number by 2 to the power n, and the proof will exactly be the same as the way we proved for the left shift and is just that instead of 2 to the power plus k, it will become 2 to the power minus k. But let us consider a examples that is will be lot of fun. So, let us again consider a 4 bit system now let us consider a 0 1 0 0; so this is equal to 4. Now let us right shift it by 2 positions we will get 0 0 0 1. So, this number was 4 and this number is 1 so we see that division holds.

Now, let us consider negative numbers. So, let us say consider minus 6, which is 1 0 1 0; so let us right shift it by 1 position. In this case if we right shift it by 1 position, this is how all the bits will actually move. So, no problem let us write 1 0 1 and we replicate the sign bit. So, this number is 8 plus 4 12, plus 1 13. So, in twos complement this is 13

minus 16 or this number is minus 3. So, we clearly see that right shifting a number by n positions, can be a positive number or can be a twos complement negative number is same as dividing a sign number by 2 to the power n. So, a logical left shift and in arithmetic right shift, essentially are very useful operations and they help us in multiplying or dividing numbers by 2 raise to the power n, by any power of 2 all right.

(Refer Slide Time: 44:40)

Shift Instructions - II

- * logical shift right (lsr) (>>> operator)
 - * $1000 \ggg 2 = 0010$
 - * same as dividing the unsigned representation by 2^n

lsl, lsr, asr

Example	Explanation
<code>lsl r3, r1, r2</code>	$r3 \leftarrow r1 \ll r2$ (shift left)
<code>lsl r3, r1, 4</code>	$r3 \leftarrow r1 \ll 4$ (shift left)
<code>lsr r3, r1, r2</code>	$r3 \leftarrow r1 \ggg r2$ (shift right logical)
<code>lsr r3, r1, 4</code>	$r3 \leftarrow r1 \ggg 4$ (shift right logical)
<code>asr r3, r1, r2</code>	$r3 \leftarrow r1 \gg r2$ (arithmetic shift right)
<code>asr r3, r1, 4</code>	$r3 \leftarrow r1 \gg 4$ (arithmetic shift right)

Mc Graw Hill Education

44

So, now let us consider a logical shift right, which is actually represented by the 3 greater than operators right, is the terminology has come from java. So, in this case the sign bit is always set to 0. So, this is as same as dividing the unsigned representation by 2 to the power n right that is it is connotation. So, in this case we just move all bits 2 steps I mean n steps to the right and msbs are filled with 0s. So, the food for thought question our here is that why do not we have an arithmetic shift left? Well the reason is that the sign bit always gets over written a numbers from the left. So, there is no meaning of an arithmetic shift left, you only have 1 shift left, but 2 shift rights.

So, we have a couple of Simple RISC instructions, which we will you know 3 to be pre size and their format is exactly the same as the add sub, multiply instructions. So, the logical shift left instruction the destination comes first which is the register, the first source operand is a register and a second source operand can either be a register or an immediate, reg slash m write either a register or an immediate, and the name of the instruction is the lsl logical shift left.

Similarly, we have a logical shift right operator which is called lsr, exactly same notation and we have an asr an arithmetic shift right operator, which is called asr. So, what are the 3 new instructions that we are introducing? The 3 new instructions are lsl, lsr and asr logical shift left, logical shift right and arithmetic shift right.

(Refer Slide Time: 46:45)

Example with Shift Instructions

* Compute $101 * 6$ with shift operators

```

mov r0, 101
lsl r1, r0, 1
lsl r2, r0, 2
add r3, r1, r2

```

mul r0, r0, 6

(mul, div) expensive

Shift (ultra fast)

McGraw Hill Education

45

So, let us give some neat examples with shift based instructions, let us assume we want to compute 101_2 multiplied with 6_{10} with just shift based operators. So, the first task is register assignments, so we mov 101_2 to register r 0. So, we shift r 0 by one position which is essentially equal to r 1 is equal to 101_2 multiplied with 2, because we are shifting it by one position and in this case now then again we compute r 2, which is shifting 101_2 by 2 positions and multiplying it by 4.

Then we add r 1 plus r 2 and we set the result equal to r 3. So, essentially r 3 becomes 101_2 multiplied by 6. So, you know skeptic would ask that why did we do this, we should have simply loaded 101_2 into r 0, and you know written an instruction of this form, as simple as this we should taken r 0, multiplied it with 6 and save the result in r 0 or r 3 or any other register right, why did we have to shifts? Here is the answer. So, typically in hardware multiplication and division right multiplication and division are very very expensive operations; expensive in the sense that they take a lot of time right. So, I can say you know plus plus in terms of time right, they take a lot of time.

So, lot of compilers actually you know lot of compilers and good programmers encourage the user, not to use very expensive multiplication, division, instructions and use shifts instead which are very fast right. So, shift instructions are ultra fast. So, the shift instructions are ultra ultra fast and they are extremely fast. So, that is the reason it is a good idea to use shift instructions in a place of expansion, multiplication and division instructions wherever it is possible.

(Refer Slide Time: 49:28)

Example - II

* Compute 102 * 7.5 with shift operators

```

mov r0, 102
lsl r1, r0, 3
lsr r2, r0, 1
sub r3; r1; r2

```

$r_1 = 102 \times 8$
 $r_2 = 102 \times 0.5$

$r_3 = 102 \times (8 - 0.5)$
 $= 102 \times 7.5$

McGraw Hill Education

46

So, let us consider one more example which is not that difficult once you know the answer. So, let us compute 102 times 7.5 with the help of shift operators. So, what we can do? The first thing is assign 102 to a register r 0. So, then we shift it to the left by 3 positions, which means r 1 is equal to 102 multiplied by 2 to the power 3, which is 8 and then we shift it to the right.

So, since 102 is positive logical and arithmetic shift are the same. So, we shift it to the right by one position, which means multiply 102 with 0.5, well then it is very straight forward we subtract r 1 minus r 2. So, r 3 becomes equal to well this is exactly what we wanted to do. So, you see this is very easy and instead of we have avoided the costly expensive time consuming division instruction and in the place of that, what we have done is that we have a very very nimble and efficient solution with shift instructions ok.

(Refer Slide Time: 50:56)

Load-store instructions

$\text{ld } r1, 10(r2)$	$r1 \leftarrow [r2+10]$
$\text{st } r1, 10(r2)$	$[r2+10] \leftarrow r1$

* 2 address format, base-index addressing

* Fetch the contents of r2, add the offset (10), and then perform the memory access

Mc
Graw
Hill
Education

47

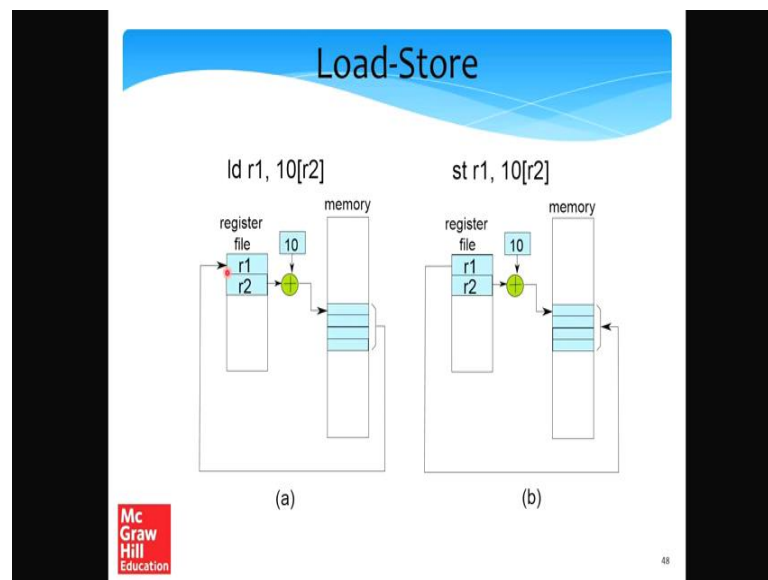
Now, let us introduce some of the real the instructions that help us write big and scalable programs. So, let us intros introduce load store instruction. So, load store instructions job it is essentially a load instructions job is to take a value from memory and bring it into a register. So, this is a load instruction and a store instruction job is to take a value from a register, and put it in memory right. So, what does a load instruction do? It takes a piece of data from memory and puts it in a register; and similarly a store instruction takes data from a register and puts it in memory.

So, let us take a look at the format of a load instructions; so in a load instruction the register which needs to be loaded or is a destination right. So, this is the destination, in this 2 address format the first is the destination and the other uses base index I am sorry not base index it is base offset addressing; it should be base offset addressing. So, in this case what we do is that the base address is specified in register r 2. So, we take register r 2 we add 10 to it, we get a new memory address. We access the memory using this address read the contents and put it in r 1. So, in this 2 address format instruction the second operand is always points to a certain memory location a value and a memory location, the hardware's job is to read this value that is there in the memory location to read this value and put it in a register.

So, store instruction has exactly the same format right, but the store instruction if you think about it does not actually have a destination, the destination is memory. It does not

have a register destination. So, what the store instruction does is that the register that needs to be read right the register that is actually read is specified first, and then the memory location is specified after this using exactly same addressing mode or base offset addressing mode.

(Refer Slide Time: 53:34)



So, let us show this thing graphically. So, graphically what we do is that in the case of load, we take the value of `r2`. One second; we take the value in of `r2` from the register file we add 10, we get the address. This address is used to access memory; in a 32 bit system we read 4 bytes from memory, which is the data, the data comes back and it is fed into the register `r1`. So, instead of 4 bytes it can be in a 64 bit system it can be 8 bytes that does not matter.

So, in this case in a 32 bit system we read 4 bytes from memory and put it in the register. So, store actually does the reverse. So, in this case also we compute the address, but instead of the flow of the data being from memory to a register, the flow of data is actually reverse it is from a register to memory.

(Refer Slide Time: 54:42)

Example – Load/Store

* Translate :

```
int arr[10];
arr[3] = 5;
arr[4] = 8;
arr[5] = arr[4] + arr[3];
```

```
/* assume base of array saved in r0 */
mov r1, 5
st r1, 12[r0]
mov r2, 8
st r2, 16[r0]
add r3, r1, r2
st r3, 20[r0]
```

Mc
Graw
Hill
Education

49

So, let us give an example of a load store list. So, let us consider an example with arrays. So, let us assume that an arrays base address is stored in r 0. So, the way that an array would actually look like in memory is something like this, that an array with 10 integers where 1 integer is 4 bytes is a contiguous region of 40 bytes. So, in this let us assume that the base address of the array is r 0, which means the first integer is stored in the locations that r 0. So, let us assume that the contents of r 0 are v. So, the first integer is stored in the locations v 2, v plus 3. The second integer is stored in the location v plus 4 v plus 7 and so on.

So, we can say that may be the starting address of the array in this case array 0s starting address is actually v, the starting address of the first entry is v plus 4 and so on. So, similarly the starting address of the 9th entry, the 9th and the last entry in the array, where we start counting from 0, is essentially v plus 36. So, given the simple argument which I am sure most of you would have learnt in your basic C or C++ or java programming class, let us try to implement this small program in assembly language.

So, we have an array of 10 integers, where we have assume the base of the array is saved in register r 0, we set the third element of 5 array 3. Now the element at index 3 to 5, the element at index 4 to 8 and we set r 5 is equal to r 4 plus r 3. So, now, let us do some amount of register assignments. So, let us move the constraint 5 to r 1, and let us save the value r 1 in actually the location array 3.

So, what is this starting address of array 3? Well the starting address of array 3 is pretty much $r_0 + 12$, the reason being let us go back to this discussion over here. So, the starting address of array 0 is the contents of r_0 , starting address of r_1 is contents of $r_0 + 4$, similarly starting address of array n is $r_0 + 4 \times n$ minus 1 sorry $r_0 + 4 \times n$. So, in this case it is $r_0 + 12$. So, we write it as $12 r_0$. So, this is the very very important point to understand and I want to ensure that all of you have more or less understood it.

So, the idea is that what is the content of r_0 ? R_0 pretty much points to a memory address in memory that contains 40 contiguous bytes. So, the first 4 bytes are r_0 , the next 4 bytes r_1 and so on. So, what is the starting address of r_0 ? It is of array is 0. So, starting address of array 0 is r_0 right let us say the starting address. Let us starting address of array 1, r_1 is $r_0 + 4$. 4 bytes if the size of one integer is 4 bytes, similarly the starting address of r_n is $r_0 + 4 \times n$. So, that is the reason the starting address of r_3 is $12 r_0$.

So, we saved the values, we do the same. So, we do what we have done here we do the same for r_4 , where we save 8 and r_2 and then we store the value of r_2 in $16 r_0$, which is essentially the same as setting r_4 array 4 is equal to 8 finally, we add r_1 plus r_2 because they contain the values that were stored and we save it in r_3 and the value of r_3 is again saved in 20; 20 is the offset and r_0 is the base. So, this is essentially equal to the fifth memory address of the fifth element. So, here we save r_3 which is r_1 plus r_2 or in a sense the same as. So, by this time the advantage of using of base offset addressing mode should be clear, that it allows us to implement arrays.

So, what we can do is that we can save the base address in an array and then we can use the offset to actually access different elements of the array right. So, we have the always need to be mind full of the size of an element. So, in this case the element is an integer and the size of each element is 4 bytes that is the reason to go to the n th element, where we start counting from 0 is the starting address plus $4 \times n$. So, once that is clear, we can write programs of this type to load and store values from memory from you know different data structures in memory the array being 1 to read them into registers work on them and write them back.

(Refer Slide Time: 61:15)

Branch Instructions

* Unconditional branch instruction

b .foo	branch to .foo
--------	----------------

b <name of label>

```
add r1, r2, r3
b .foo
...
...
.foo:
    add r3, r1, r4
```

50

Mc
Graw
Hill
Education

Now, it is time to actually get to the real fun part of assembly instructions; and so what have we covered up till now? We have covered arithmetic logical instructions and we have covered load store instructions right. So, it is now you know the apt time to actually get into the real fun part of assembly programming, which is to add branching. When we add branching we will be able to implement high level concepts, such as if statements for loops while loops and so on. So, as I have mentioned in some of the early slides, that an assembly statement can be associated with a label and this label can be used to uniquely indicate a assembly statement.

What we can later on do is that we can have this statement b dot foo, where the branch statement would essentially transfer the control of the program, from this particular line over here to this line here. So, the program counter would initially be pointing to this line and subsequently it will start pointing, to this line and this statement will get executed. So, the branch statement in assembly is very simple we just do b, and then the name of the label. So, the format in this case is b and the name of the label.


(Refer Slide Time: 63:05)

Conditional Branch Instructions

cmp .foo .foo

→ beq .foo .foo	branch to .foo if $flags.E = 1$
bgt .foo .foo	branch to .foo if $flags.GT = 1$

- * The flags are only set by cmp instructions
- * **beq (branch if equal)**
 - * If $flags.E = 1$, jump to .foo
- * **bgt (branch if greater than)**
 - * If $flags.GT = 1$, jump to .foo

 51

So, there are some conditional branch instructions as well, where we will actually see the power of having the flags register right.

So, let us have a beq instruction branch if equal. So, what this means is that look? We have first introduced we first have a compare instruction, we have many more instructions that are not compares and then we have a beq instruction. So, this instruction first takes look at a flags dot e, if flags dot e is equal to 1 means that this compare over here had resulted in equality, it will go jump to foo very much branch to foo. The same is true with the b g t instructions which takes a look at flags dot GT and if flags dot GT had been set to 1 by the last compare instruction; the b g t instruction would jump or branch to dot foo.

Essentially the statement which is there at the dot foo label and these flags mind your only set by compare instructions and who are they meant for? They are meant for later branch instructions like b e q and b g t to take a look at them and jump.

(Refer Slide Time: 64:25)

Examples

* If $r1 > r2$, then save 4 in $r3$, else save 5 in $r3$

```
→ cmp r1, r2
→ bgt .gtlabel
→ mov r3, 5
...
→ .gtlabel:
    mov r3, 4
```

McGraw Hill Education

52

So, now it is time to write slightly difficult programs and realize the power of assembly language. So, let us try to write this program if $r1$ is greater than $r2$, then let us say 4 in $r3$ else we say 5 in $r3$. So, well that is easy the first thing that we do is that we compare $r1$ and $r2$. So, if $r1$ is greater than $r2$ then. So, we always come one statement down, if it is greater than $r2$ the flags dot GT bit would be set to 1. So, if it is greater than we jump to $gtlabel$, which is over here and we save 4 in $r3$.

Otherwise we do not jump and by default we come to the next statement, where we save 5 in $r3$. So, this is a simple program which shows us the power of the bgt instruction that if that element condition is true in the flags register, then we jump to the target; in this case the target is dot $gtlabel$, otherwise we just go to the next or the subsequent instruction.

(Refer Slide Time: 65:49)

Example - II

Answer: Compute the factorial of the variable num.

$$n! = 1 \times 2 \times 3 \times \dots \times (n-1) \times n.$$

```
int prod = 1;
int idx;
for(idx = num; idx > 1; idx--) {
    prod = prod * idx;
}
```

Let us now try to convert this program to SimpleRisc. (num ← r0)

```
SimpleRisc
mov r1, 1          /* prod = 1 */
mov r2, r0        /* idx = num */
loop:
mul r1, r1, r2    /* prod = prod * idx */
sub r2, r2, 1     /* idx = idx - 1 */
cmp r2, 1         /* compare (idx, 1) */
bgt .loop        /* if (idx > 1) goto .loop */
```

Mc Graw Hill Education

53

Now, let us move on to a more complicated example and this is a genuine assembly program in the sense it has it is most of it is features. So, let us compute the factorial of the variable called num right. So, which is as specified in the c program over here. So, to write a program in assembly language, let us may be add a step 0. If the program is complicated, it might actually make a lot of sense, to first write it in a high level language such a such as C or java. The reason for that is that it will help us in a visualized all the parts, and it will become fairly easy for us to actually code it. So, we will follow this approach over here.

So, to compute a factorial first we define the product variable, which is initialized to 1 then we define an index that starts at num and we assume you know for all practical purposes num is a large enough number, so we do not have to check for special cases. So, here if i d x is equal to we start the index for loop equal to num, and we keep going till num reaches 1 and the moment reaches 1 we jump out of the loop and we subtract the index; I am sorry we keep going till i d x reaches 1 and we subtract the index by 1 in each step, and this is the multiplicative step prod equals prod multiplied by i d x. So, this code is enough to compute the factorial of a number. So, recall that the factorial of a number n factorial is 1 multiplied by 2 all the way till n right.

So, let us now try to convert this program to Simple RISC. So, in this case the first thing that we do step 1 is register assignment. So, r 1 is assigned to prod and. So, we set prod

equal to 1 and we set. So, this is $i dx$. So, r_2 is assigned to $i dx$, and we initialize it with num. So, as the assumption here is that num is assigned to r_0 . So, r_0 contains num. So, that is the assumption that is the assumption that we begin with. So, sometimes if assumption is not specified in the question, users can make their assumption as we have done over here that the num variables whose factorial needs to be computed is there in r_0 and so what we do is that the variable $i dx$ in the c program, we assign it to r_2 and we initialize we set r_2 equal to r_0 where r_0 contains num.

So, essentially the starting value of $i dx$ is equal to num. Subsequently we add the dot loop label over here, to signify the fact that later on we might want to jump to this point, then what we do is that we multiply r_1 with r_2 ; recall that r_1 is mapped to prod r_2 is mapped to $i dx$. So, we set r_1 equal to r_1 times r_2 , which is equivalent to prod equals the product is equal to product times $i dx$. So, this is the multiplicative step; then what we do is that we subtract the index we subtract 1 from the index, which is essentially this step over here, we subtract 1 from the index $i dx$ equals $i dx$ minus 1.

So, when do we actually stop the loop we stop the loop, when the index becomes equal to 1, because there is no point multiplying a number by 1. So, we compare r_2 with 1 all right. So, we compare r_2 with 1 in this step, and we see if r_2 contains $i dx$, if $i dx$ is greater than 1 or not. If $i dx$ is greater than 1, then the flags dot g t bit would have been saved. So, in this case the b g t condition would evaluate to true and we would jump to loop, which is essentially we will jump back like this.

So, what we have done, we have actually these 6 lines. I would advise the reader or the student the listener of this video, take a look at this example 5 times, 10 times, 100 times if required and understand each and every line of this program, because the program might be 6 lines, but it is 10 to the power 6 times difficult. It is difficult basically because you are looking at an assembly program for the first time and an assembly program which is fairly complicated for the first time, so let me look at the 3 steps that we had in this exercise.

The first step is we realize that the problem is difficult. So, we wrote our small program in c that is the first thing we did. In the program in c we had a product variable and an index that goes down from the number till 1 and at each point we multiplied. So, we are doing exactly the same in assembly nothing different. So, the first is that we do a register

assignment, so num is assigned to r 0 prod to r 1, i d x to r 2. We start the iteration the same way as it is in the for loop with i d x equal to num in pretty much this statement over here. So, then we do the multiplicative step we multiply and as I said am making some simplistic assumptions over that num is actually large enough so I am not doing some checks.

So, then the next step is once you do the multiply let us reduce the index by 1. So, let us have i d x equals i d x minus 1, with a subtract instruction and then we compare. So, if you have reached the exit condition, which is if r 2 is equal to 1 then we can exit the loop. So, we do not have to anything that b g t instruction over here will evaluate to falls. So, we will just simply fall down and evaluate the next instruction. Otherwise if the index is still not equal to 1 we need to do another iteration of the loop. So, we will jump to dot loop.

So, similarly the loop will continue for num minus 1 times till the factorial computation is over. So, I would request once again the reader to take a look at this several times may be work it out with the paper and pencil, and only when the reader is convinced we actually move to slide number 54.

(Refer Slide Time: 73:04)

* Write a SimpleRisc assembly program to find the smallest number that is a sum of two cubes in two different ways $\rightarrow 1729 = 10^3 + 9^3 = 12^3 + 1^3$

McGraw Hill Education

*z -> r+i
C
LRAJ
Assembly*

So, here is a food thought for burger question, write a Simple RISC assembly program to actually find the Ramanujan numbers. This is difficult, but can be done it is a fair amount of work, but with the emulators that are there on the websites can be done. So, the

Ramanujan number is a very interesting number and so basically the story goes like this that once Srinivas Ramanujan was very sick and his guide personally had come to meet him.

So, number 1729 was written on his taxi. So, he asked Ramanujan do you know, what is the significance of this number, can you find something special with this number? Ramanujan brilliant as he was said yes, so 1729 is actually the smallest number that is a sum of 2 cubes in 2 different ways, it is 10 cube plus 9 cube, which is also 12 cube plus 1 cube. So, it is a sum of 2 cubes in 2 different ways. So, the question was can you write an assembly program to actually compute the Ramanujan number? Well the answer is very simple first write the program in c, after writing the program in c gradually convert it to assembly.

So, the approach would be that we take all the numbers starting from let us say 2 till you know infinity right and we stop at the Ramanujan number. For 2 you essentially find out is it possible to write it as sum of cubes, well yes 1 cube plus 1 cube, but any other combination of cubes? No. So, you go to the next, go to the next keep going, going, going you will have multiple for loops, but the trick is first write and see then do a register assignment and then write it in assembly language that is a trick. If you are able to do this our objectives are satisfied.

(Refer Slide Time: 75:29)

Modifiers

- * We can add the following modifiers to an instruction that has an immediate operand
- * Modifier :
 - * **default** : mov → treat the 16 bit immediate as a **signed number** (automatic sign extension)
 - * **(u)** : movu → treat the 16 bit immediate as an unsigned number
 - * **(h)** : movh → left shift the 16 bit immediate by 16 positions

mov r0, -3
sign ext: 11...1101
movl
0-0 16
movh
0-0-0-0 16

55

**Mc
Graw
Hill
Education**

Now, let us discuss our last important concept in this sub section of the lecture called modifiers. So, if you would recall we had discussed that Simple RISC, support 16 bit immediates. So, it does not support larger immediates. So, the question is that let us say I have an instruction of the form `mov r0, and minus 3`. So, what is it that would happen? So, if we consider what is minus 3 in a 4 bit system? Minus 3 in a 4 bit system is plus 13 or 1 1 0 1; what is minus 3 in a 16 bit system? Nothing just performs sign extension it remains the same.

So, now the question is that when this is being moved to r0, which is actually a 32 bit register what should be the default behavior if I do this? What should be the default behavior is that we have the contents of the immediate mov to the lower 16 bits and the upper 16 bits is just a simple sign extension, so it preserves the sign of the immediate. So, this is the default behavior which all of us expect should be happening, that is the sign extension will happen. So, essentially we are treating the 16 bit immediate as a sign number, and there is automatic sign extension happening. So, this ensures that if we have represented minus 3 in assembly, minus 3 is also what gets stored in the register r0, but of course, since r0 is 32 bits relevant sign extension is being done.

Now, let us define 2 variants of this same instruction and these variants hold for other instructions as well, but let us only discuss this instruction. So, let us define the `mov u` instruction. So, in the `mov u` instruction we treat the 16 bit immediate as an unsigned number, so essentially what happens is that in the 32 bit field, the first 16 bits is when the is where the immediate comes in and the remaining 16 bits are all 0s. So, we treated as an unsigned number; and in `mov h` what is done it is actually have 1 graph over here 1 diagram over here but let me never the less explain it over here.

So, `mov h` what is typically done is that again if we consider a 32 bit register with 2 16 bit parts. So, what happens is that the immediate actually gets loaded to the upper 16 bits, the most significant 16 bits and the lower 16 bits are set to all 0s. So, you can think of left shifting the immediate and then loading it to the upper 16, to the lower 16 are all 0s. So, the u and h are generic modifiers, so the default is always to treat the immediate as a fine number, but or we can use the u and h modifiers. So, the u treats it as an unsigned number, and the h actually left shifts the immediate by 16 positions.

(Refer Slide Time: 79:08)

Mechanism

- * The processor **internally converts** a 16 bit immediate to a 32 bit number
- * It uses **this 32 bit number** for all the computations
- * Valid only for arithmetic/logical insts
- * We can control the generation of this 32 bit number
 - * sign extension (**default**)
 - * treat the 16 bit number as unsigned (**u suffix**)
 - * load the 16 bit number in the upper bytes (**h suffix**)

McGraw Hill Education


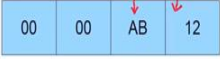
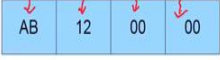
56


So, what is the mechanism the mechanism is that when an immediate is specified in an assembly instruction, the processor will take the immediate and put it in an internal register which is not visible and convert it into a 32 bit number right? So, essentially the immediate is part of the assembly instructions, it is somewhere inside, this immediate is taken by the CPU, and it is converted to a 32 bit immediate right and this 32 bit immediate is used in the computations. So, for all arithmetic logical computations this 32 bit immediate would be used.

So, we can control the generation of this 32 bit number internally. So, the default is sign extension that we can treat the 16 bit number as unsigned with the u suffix or load it in the upper 2 bytes, right the most significant 2 bytes with the x suffix.

(Refer Slide Time: 80:07)

More about Modifiers

- * default : `mov r1, 0xAB 12`

- * unsigned : `movu r1, 0xAB 12`

- * high: `movh r1, 0xAB 12`




57

So, let us take a look at some examples. So, let us first take a look at the default behavior of `AB 12` whether the msb is equal to 1. So, in this case we load `AB 12` in the lower 2 bytes, and the upper bytes upper 2 bytes consist of the sign bit right. So, this is sign bit here is 1 so all these 2 16 bits are 1.

If we consider the unsigned so then again the byte `AB` is loaded into this byte and so `AB` and `12` are loaded into lower 2 bytes, but the upper 2 bytes are set to 0s right, 8 0s and 8 eight 0s, 16 0s. If we consider the same thing `movh r1, 0xAB 12`; so `AB 12` are actually move to the 2 msb byte positions, upper byte positions and the lower positions are set to 0.

(Refer Slide Time: 81:10)

Examples

- * Move : 0x FF FF A3 2B in r0
mov r0, 0xA32B
- * Move : 0x 00 00 A3 2B in r0
movur0, 0xA32B
- * Move : 0x A3 2B 00 00 in r0
movhr0, 0xA32B

McGraw Hill Education

58

So, let us take a look at some examples; so the main advantage of having modifiers right why have them? The main advantage of having modifiers is that it is possible to load all kinds of constants into registers; we will load meaning not from memory, but you know sort of store immediates and registers, all kinds of immediates and registers with a minimal number of instructions for example, if we want to save FF FF A 3 2 B in r 0; all that we do is that we actually move in A 3 2 B into r 0. So, it will load A 3 2 B in a lower positions and upper positions will replicate the sign bit; since the sign bit is 1 the upper 2 bytes will become FF and FF. If we want to have set the upper 2 bytes as all 0s right in this case all that we need to do is we need to have the u modifier, which will ensure that the most significant 2 bytes are 0s.

Likewise if we want A 3 and 2 B to be in the 2 upper positions most significant positions, then what we do is that we will use the h modifier, to move them to the most significant positions and set the rest of the bits or bytes as 0.

(Refer Slide Time: 82:36)

Example

* Set $r0 \leftarrow 0x12AB A9 2D$

1) `movh r0, 0x12AB`
2) `addu r0, 0xA92D`

12 AB 00 00
+
00 00 A9 2D

12 AB A9 2D

Mc
Graw
Hill
Education

59

So, now let us look at our simple example; let us assume that this complicated constant over here needs to be loaded sorry I mean saved into $r0$. So, this if we don't have the modifier it will actually take us a lot of instructions it is complicated.

So, let us now break down this problem into 2 simple problems and solve the problems in parts. So, the first problem is that we need to take 12 AB and load them in the 2 upper bytes that is very easy. So, we consider $0x12AB$ right in the hex form, we use the `h` modifier to actually move this, the 2 upper bytes. Then we need to move A9 2D to the lower bytes. So, this is also easy. So, basically what do we have what is the status of the register at the moment, just before the second instruction we have 12 and AB and then we have 00 and 00 in the 2 lower bytes.

So, what we can do is that we can add this with 00 A9 and 2D. So, this will give us the final result which is this result. So, that is the reason we have an add instruction, we have $r0$ and we have A9 2D, but here is the catch if A9 2D goes into the machine, it will be expanded into a 32 bit value and in the 32 bit value the lower 2 will be A9 2D, the lower 2 bytes, but the upper bytes will just replicate the sign of A which is F F F F, which is not something we want to do. So, that is the reason we specify the `u` modifier with the add instruction.

So, essentially the modifiers can be used with any arithmetic logical instructions and the move instruction right. So, let me say this again, the `u` and `h` modifiers can be used with

any instruction that uses immediates namely the arithmetic logical instructions and the move instruction. So, in this case in the second instructions this is instruction 1 and it is instruction 2. So, in the second instruction we use the u modifier, to actually expand A 9 2 D internally into A 9 2 D and the upper 2 bytes are 0 0 and 0 0, which is exactly what we want. Have we not given the modifier, we would have gotten the wrong answer; to get the right answer we add the u modifier to ensure that these 2 upper bytes are 0 and 0 and then we do the addition, so we get 1 2 AB, A9 and 2 D, which is exactly the answer that we wanted.

So, one thing that this example shows what is the take home point of this example? The take home point of this example is that to load a constant, I am sorry I use the word load, but the word should be taken in the context, in the connotation if you want to save an immediate into a register. Say it would have otherwise been difficult had the modifiers not been there to actually put a constant in to a register given r i s a, but with the modifiers it is actually become easy. So, we use the h and u modifiers and with just 2 instructions we are able to put in 32 bits, into a register which is great.