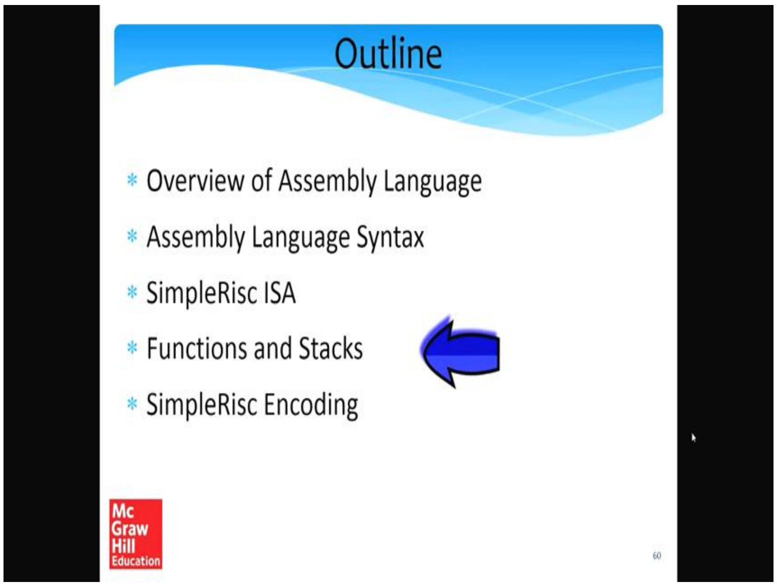
Computer Architecture
Prof. Smruti Ranjan Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 07
Assembly Language Part-III

(Refer Slide Time: 00:29)

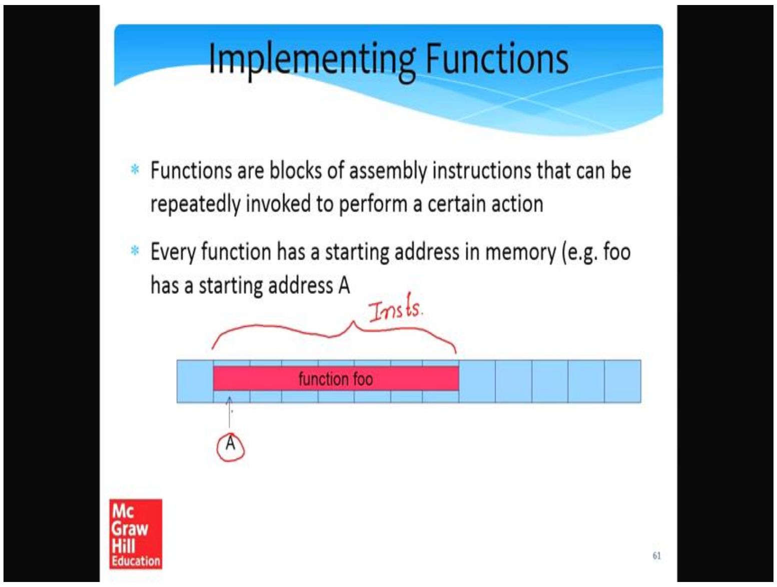


Outline

- Overview of Assembly Language
- Assembly Language Syntax
- SimpleRisc ISA
- Functions and Stacks
- SimpleRisc Encoding

Now it is time for us to look at some advanced concepts, namely functions and stacks. So, what we have discussed up till now we will take us to the certain point, but it will not allow us to write fairly difficult or complicated assembly programs. As a result it is necessary to introduce a couple of slightly more advanced and complicated concepts, as the most important of that being functions, and to support functions we need a software structure called a stack.

(Refer Slide Time: 00:59)



Implementing Functions

- Functions are blocks of assembly instructions that can be repeatedly invoked to perform a certain action
- Every function has a starting address in memory (e.g. foo has a starting address A

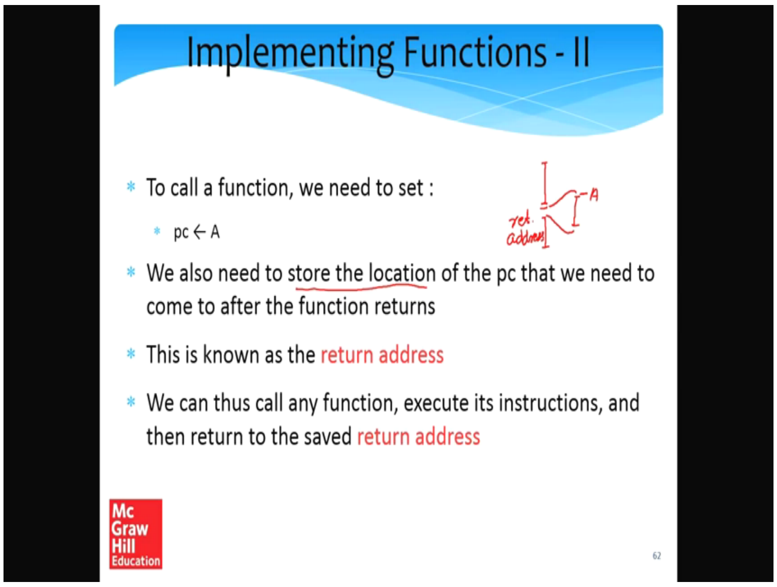
Insts.
function foo
A
61

So, what is a function? So, let us consider a c program, a function is essentially a block of code that achieves a certain specific task. So, we give a function a set of arguments and then for example, it is a function to find if a number is a prime number or not then the number is an argument to this function and the result a Boolean result of whether the number is prime or not is what the function returns.

So, similarly we should be able to write functions in assembly language as well, similar to a high level programming language. So, if you think about how functions are will actually be represented in memory, it is something like this as shown here that it is a we take a region of memory put in all the instructions of the function here. So, I will just write insts to be short. So, all the instructions we put them in a contiguous region of memory and this region of memory if it starts at this point the beginning the first byte.

So, this is the beginning of the function and says any function so foo traditionally is the name of a function when we give an example. So, in this case this continuous region of memory stores all the instructions or the function, the data is stored somewhere else we will come to that and any function will be identified by the address, the starting address or the first instruction.

(Refer Slide Time: 02:42)



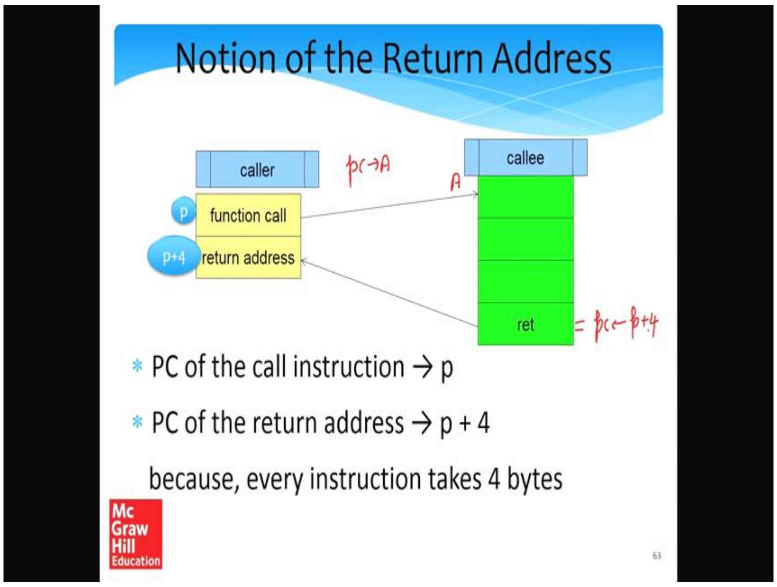
Implementing Functions - II

* To call a function, we need to set :
    * pc ← A
* We also need to store the location of the pc that we need to come to after the function returns
* This is known as the return address
* We can thus call any function, execute its instructions, and then return to the saved return address

So, to call a function what do we need to do? So, let us just go back to the previous slide if the starting address is A, we set the program counter to A. So, this is equivalent to jumping to the first address of the function. So, once we do that we can start executing the function and after the function completes, we need to store the location of the program counter at which we need to return to. So, we consider a traditional execution of a program, we keep on executing we go on execute some function, we execute the function and we come back and then we started exactly the same point.

So, we need to store the location of the point at which we return from the function, this is known as the return address. So, this also needs to be stored, so essentially what do we need to store? We need to store 2 things, we need to know what is the starting address of the function, which is A in this example and then we need to jump back to a point which is the next instruction after the function call and this is known as the return address.

(Refer Slide Time: 04:04)


Notion of the Return Address

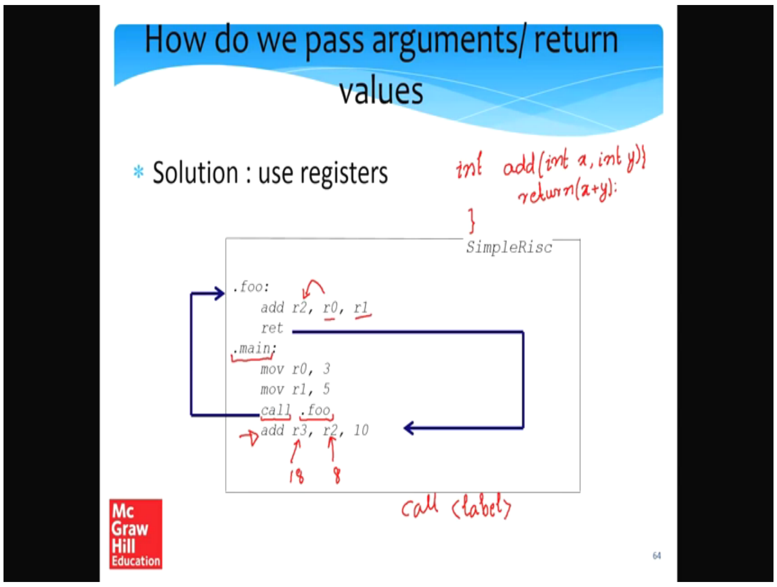
- PC of the call instruction → p
- PC of the return address → p + 4

because, every instruction takes 4 bytes

So, let us take a look at the motion of the return address once more. So, let us assume that we have a collar. So, collar is the function that invokes another function. So, the caller calls a function and. So, we have a function called instruction and let the address of this the call instruction be p. So, then we call a function which basically means that we jump to the beginning of the function. If we execute all of it is instructions then we have the ret instruction which is the return instruction, where we again come back and we come back to the instruction which is just after the function call. In simple risk since we assume that each instruction is 4 bytes, we start at address p and we return back to address p plus 4, this is the pc the program counter of the return address and so let us me just maybe expand this example with some more details.

So, let us assume that the starting address of this instruction is A. So, the call function is pretty much we will change the value of the PC equal to A, we will start executing this instruction. The return address is p plus 4 because that is the address of the next instruction. So, when I do the ret instruction, when I invoke the ret instruction it sets the program counter the PC equal to p plus 4.

(Refer Slide Time: 05:44)



How do we pass arguments/ return values

- Solution : use registers

```
int add(int x, int y)){
    return(x+y);
}
```

SimpleRisc

```
.foo:
    add r2, r0, r1
    ret
.main:
    mov r0, 3
    mov r1, 5
    call .foo
    add r3, r2, 10
```

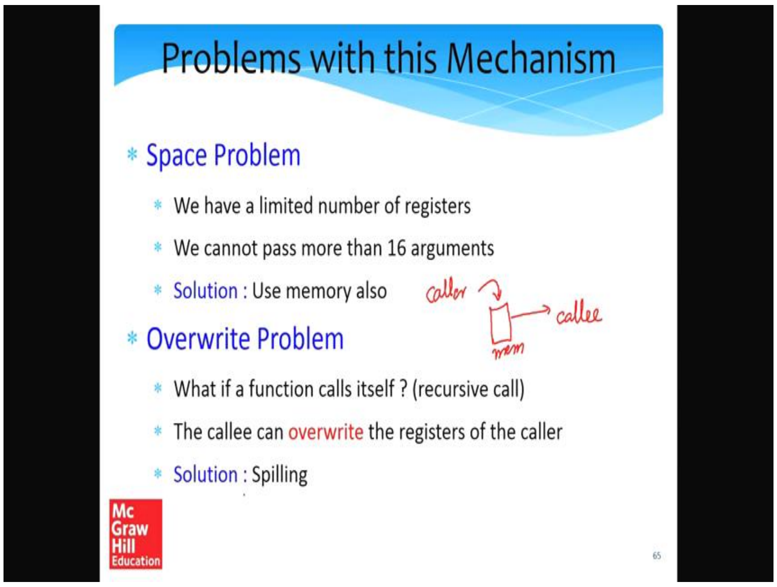
So, now that we know how a basic function call works. Let us look at a very simple example and see how exactly we pass arguments and return values 2 1 from a function. So, the example is like this that we first. So, what we want to do is, we want to write a simple function that takes 2 arguments. So, we can think of an add function, so I am writing this example in c, where we have an add function and it takes 2 arguments, let us say int x and int y and it returns it returns x plus y. So, to implement this simple function in simple risk, what we do is like this. So, similar to c, we assume that we have a dot main label where the execution of the program starts.

So, it is customary to put a colon after the label, after this let us set the value of registers we set r 0 to 3 and r 1 to 5 then now we are introducing a new instruction called the call instruction. The call instruction jumps to a label in assembly what we do is, that we designate every function with a label which is nothing, but the starting address of the first instruction. So, we call dot foo; so it jumps to this point. So, the way that we write a function is actually very simple, we start with a label this have all the instructions and then a ret instruction at the end.

So, what we are doing here is we have set the values of registers r 0 and r 1 then we call the function foo which essentially means we jump to the label dot foo and at this label we add r 0 and r 1 , we save the result in r 2 and then we return back. So, when we return back what a simple risk is supposed to do is that it is supposed to take us to the instruction after the calling instruction which is this instruction over here. So, this is exactly what we do we come back, at this point the value inside foo is r 0 plus r 1 which is 8, then what we do is we add 8 and 10 and finally, we set r 3 to 18.

So, the important takeaway points over here are that we are introducing 2 new instructions: a call instruction and a ret instruction. The call instruction is designed in such a way that we call so the, for semantics the format of the call instruction is call and label. So, in this case we actually jump to the first address of the first instruction in the function, we execute the function and execute the ret instruction, which again brings us back to one instruction after the calling instruction.

(Refer Slide Time: 09:25)



Problems with this Mechanism

- Space Problem
  - We have a limited number of registers
  - We cannot pass more than 16 arguments
  - Solution : Use memory also
- Overwrite Problem
  - What if a function calls itself ? (recursive call)
  - The callee can overwrite the registers of the caller
  - Solution : Spilling

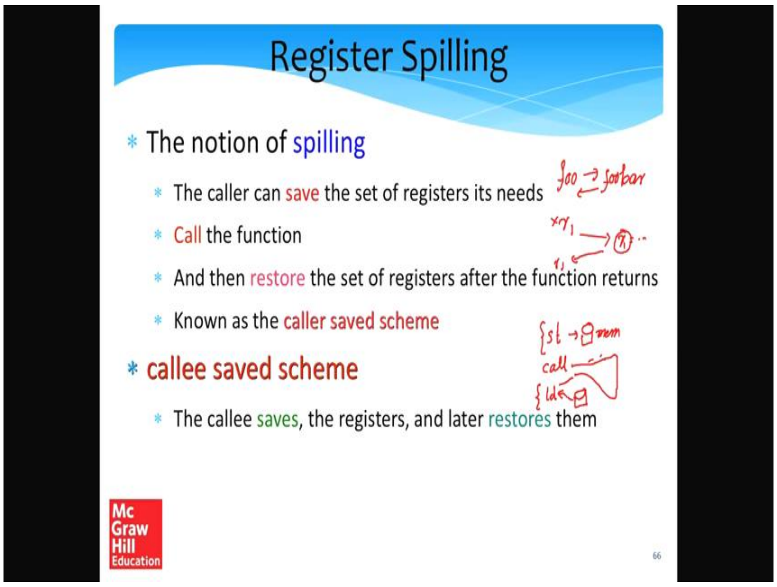
caller
callee
mem
Mc Graw Hill Education
65

So, how are we passing arguments? So, in the previous example we passed arguments by a registers, we populated some registers then the function read the registers. So, here is. So, this is one point where students typically confused. So, one thing that they need to know is that the set of registers is the same for the entire program; it is not that different functions are different sets of registers. So, you can think of the registers as being shared. So, if one function writes to a register, the other function can read from the registers. So, they see the same view of the set of registers also known as the register file.

So, there is a space problem the issue is like this that we have a limited number of registers. So, let us say in our case we have 16 registers. So, we cannot pass more than 16 arguments, it is not possible for us to actually pass more than 16 arguments to the function. So, this is called a space problem. So, one way that we can actually solve this is if we, if let us say the call the caller function puts in it is arguments into a region of memory and then the callee function reads the arguments from there. Also there is an overwrite problem which is what if a function calls itself like a recursive call.

In this case the callee will overwrite the registers that the caller may need and this is a problem. So, we will pro propose another solution, it is called register spilling. So, we will discuss both of these solutions in detail over the next few slides.

(Refer Slide Time: 11:15)



Register Spilling

- The notion of spilling
  - The caller can save the set of registers its needs
  - Call the function
  - And then restore the set of registers after the function returns
  - Known as the caller saved scheme
- callee saved scheme
  - The callee saves, the registers, and later restores them

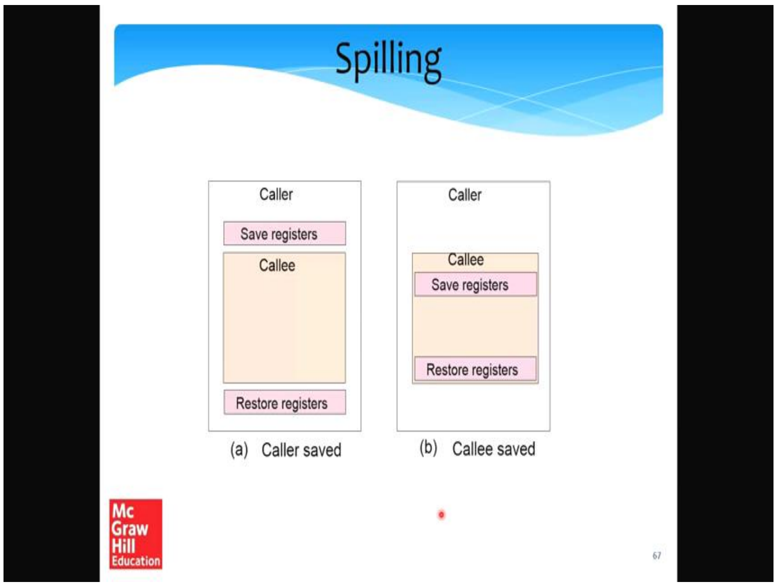
foo → foobar
Mc Graw Hill Education
66

So, when I look at a register spilling the idea is like this, that let us say I have one function foo, which is calling another function let us call it foo bar. Foo bar does something and comes back. So, it is possible that foo bar over root some of the registers that foo may require. So, maybe foo requires register r 1 and then you go to foo bar then it sets r 1 equal to something which is fine, but the thing is maybe we are not interested in that value and again when we use r 1 we might get the overwritten value and not this value. So, for that what we can do is we can use a technique called register spilling. The notion of spilling is like this, that the caller can save the set of registers that it needs to use then it can call the function and subsequently restore the set of registers after the function returns. So, this is known as the collared save scheme.

So, here essentially what you do is that you save a set of registers, that you plan to use in the future using store instructions, then you call a function the function does something and again the function returns to the next instruction. Subsequently we have a set of load instructions that load the values of these re registers that restore to memory back again. So, let us say in the store we save the value of registers to memory, and in the load we read the values of what we had stored back into our registers.

So, this particular thing is called a caller saved screen. In a callee saved scheme what happens is that the function that is called, which is called the callee that saves the registers that it might potentially overwrite and later on it restores their value when it is about to return.

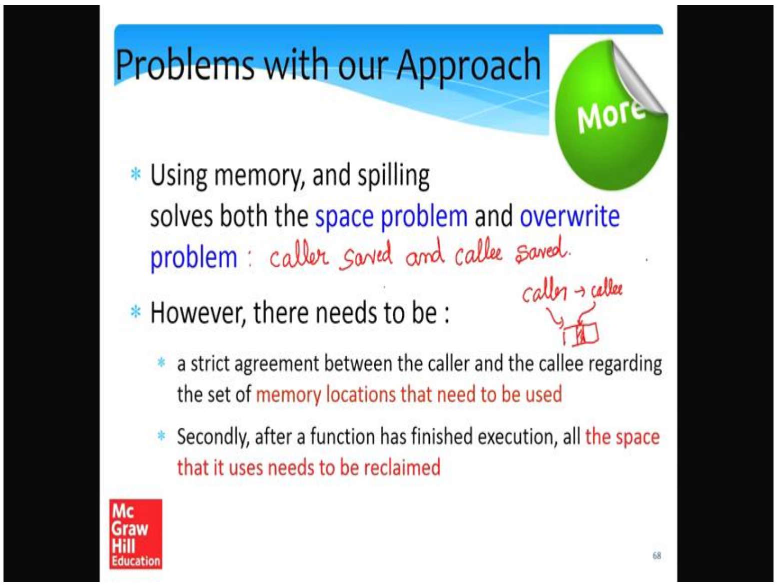
(Refer Slide Time: 13:24)


Spilling

So, if I want to think about this. So, conceptually in a caller saved scheme the way it works is something like this, that we first save our registers we store them to memory as you can see over here then we call the callee function and then we restore the registers, we restore the values of the registers. In a callee saved scheme the idea is that the caller does not do anything. So, it is absolutely. So, it assumes that all of it is registers, unless of course, you mutually decide that a certain register is going to contain a return value, all of it is registers will maintain their value.

The callee will first save the registers subsequently do it is work and just before returning restore the value of the register. So, there are 2 schemes they have their pluses and minuses, so we will you know discuss a little bit of that and most of this is actually there in the exercises at the back of the chapter, but you know the students should think when the caller save scheme makes sense and when the callee saved scheme makes sense.

(Refer Slide Time: 14:40)



Problems with our Approach

- Using memory, and spilling solves both the space problem and overwrite problem : caller saved and callee saved.
- However, there needs to be :
  - a strict agreement between the caller and the callee regarding the set of memory locations that need to be used
  - Secondly, after a function has finished execution, all the space that it uses needs to be reclaimed

So, what are we doing our solutions are like this, that we are essentially using memory and spilling which also uses memory, to solve both the space problem as well as the overwrite problem.

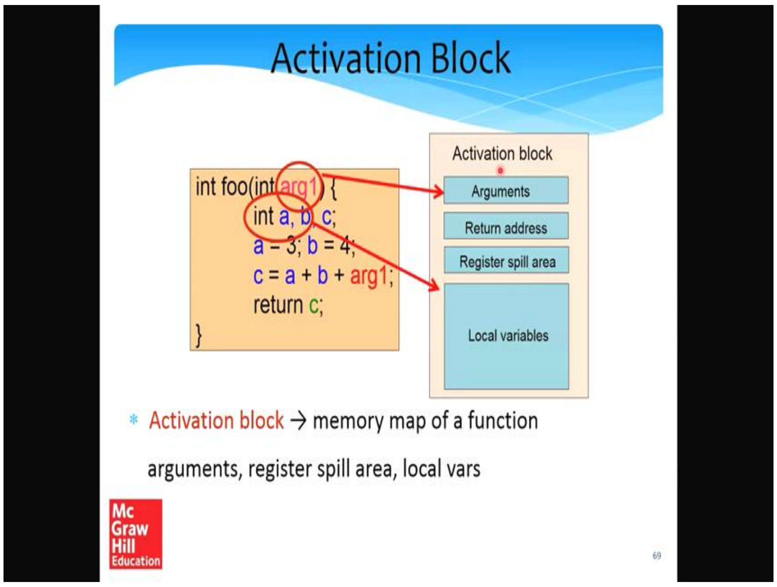
However, using this memory based approach for both solving the space problem and the overwrite problem has issues. So, let me just at this point, slightly recapitulate for these problems are the space problem basically says that we run out of registers while passing arguments. So, for example, let us say a function takes 100 arguments, you know we do not typically write such functions, but you know we can theoretically we can, but it is very common to write functions it maybe 8, 9 or 10 arguments. So, each of them is in the using the register, we will run out of registers to do other work that is the reason we will have a serious space problem, while especially passing arguments around and also you know managing them inside a function. The main reason being that we need registers for other work also like saving temporary variables and so on, temporary values and temporary variables and so on; so that is why the space problem is important. And the overwrite problem is definitely very important, the main reason being that we only have 16 registers, out of that we will see that we are typically not allowed to use more than 14 for general purpose work.

So, it is now let us say one function calls another copy of itself or other functions, then they will overwrite the registers that they require, because all of them see the same view of the re register space, that is the reason it is necessary to save to save or store registers in memory, and read them back again, this is known as the overwrite problem and we saw 2 methods of fixing it is called spilling so the 2 approaches for spilling where. So, they were callers saved and callee saved.

So; however, there needs to be a very you know to have both of these schemes running, there has to be a strict agreement between the caller and the callee regarding the set of memory locations that are being used. So, the reason for this is that let us assume the caller, saves it is registers in a certain region in memory right. So, let us say it is this region. Then the caller calls the callee, and the callee also starts saving it is registers in this region, then clearly they will start interfering in memory and this will not work out. So, the regions of memory that are to be used by different functions has to be known a priori and this is a problem and then once a function has finished, it is execution it will not need the space that it had used to store registers for spilling, this space has to be reclaimed.

So, we have to sort of somehow mark this space to be empty, such that it can be used by other parts of the program so there is a need to reclaim this space. So, this makes things pretty complicated, and the root of all the problems is the space problem and the overwrite problem.

(Refer Slide Time: 18:32)



Activation Block

```
int foo(int arg1) {
    int a, b, c;
    a = 3; b = 4;
    c = a + b + arg1;
    return c;
}
```

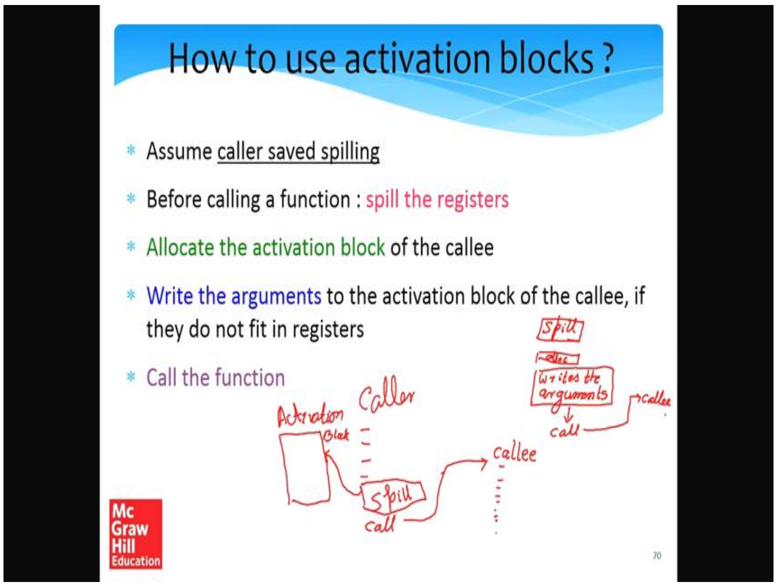
* Activation block → memory map of a function

  arguments, register spill area, local vars

So, to make a life simple, let us define an activation block. An activation block is a region of memory that stores certain information about an executing function. So, let us assume a function foo that takes a single argument r 1 it has 3 internal variables a b and c a is set to 3, b to 4, c is a plus b plus r 1 and then we return c. So, in the activation block there are a couple of things that we need to save, the first is the argument. The argument will get saved in the argument section inside the activation block, which is essentially a region of memory and we are sort of dividing the region into multiple sub regions, so in one region we save the arguments if required.

In other case we save the local variables which in this case are a b and c, the local variables are saved in this region and then we save the return address again if required and we have a register spill area, where we spill registers if required. So, the activation block is essentially the region in memory that is task to keep all the values that are required for a function to execute, and as you see after a function executes after we return c, the activation block is not required.

(Refer Slide Time: 20:12)



How to use activation blocks ?

- Assume caller saved spilling
- Before calling a function : spill the registers
- Allocate the activation block of the callee
- Write the arguments to the activation block of the callee, if they do not fit in registers
- Call the function

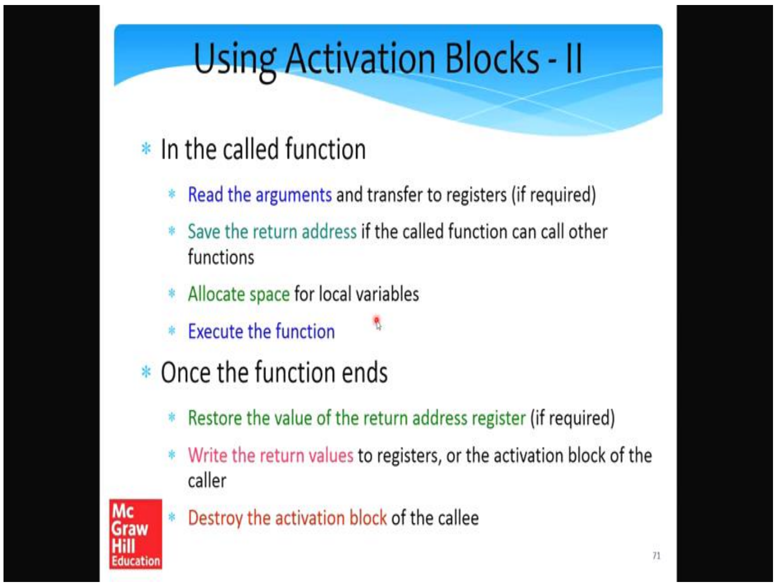
Spill
callee
writes the arguments
call
callee
Caller
Activation Block
Spill
call
callee
Mc Graw Hill Education
70

So, how do we use the activation blocks? So, let us say that we have callers saved spilling. So, before calling a function we spill the registers then we allocate the activation block of the callee. We write the arguments to the activation block of the callee, you know if they do not fit in registers, if there are one or 2 arguments they can always be passed via registers, but if there are more than one or 2 arguments then these arguments need to be passed via memory, and the way we do that is we pass the we write the arguments inside the activation block of the callee then we call the function.

So, just to illustrate how this works let me draw a small diagram; so let us assume that we have a caller function executing. So, we execute one instruction another, another, another and then we decide that we need to call a function. So, the first thing the since we assume caller saved spilling, the first thing that we do is that we spill the registers. Spill the registers basically means write the values of the registers that we are interested in into the activation block of the caller.

Subsequently we call the call instruction. The call instruction process control to the callee function, we start executing all the instructions and before actually before we before the caller calls the callee one more thing that if an additional. So, this is one example where the values are being the arguments are being passed via resistors, but we can alternatively also pass them via the activation block. So, in this case what will happen is that we spill the registers, then we allocate then what the caller does it allocates an activation block for the callee. So, I am writing alloc, then it writes the arguments function arguments to the activation block of the callee function and then we have the call function, we have the call instruction where we call the callee.

(Refer Slide Time: 23:19)



Using Activation Blocks - II

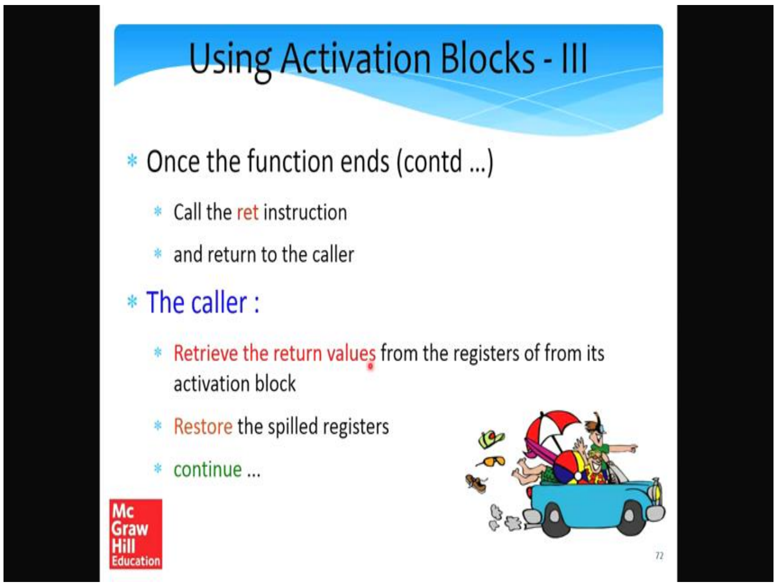
- In the called function
  - Read the arguments and transfer to registers (if required)
  - Save the return address if the called function can call other functions
  - Allocate space for local variables
  - Execute the function
- Once the function ends
  - Restore the value of the return address register (if required)
  - Write the return values to registers, or the activation block of the caller
  - Destroy the activation block of the callee

So, inside the called function which is the callee, what we do is like this; that we first read the arguments and transfer them to registers if required. We save the return address if the called function can call other functions, we allocate space for local variables and then we execute the function. Once the function ends we restore the value of the return address register if required. We write the return values to registers you know if required, but if there are a lot of values or if it is one large value we can write it to memory also, which is the activation block of the caller and then we destroy the activation block of the callee.

So, this is the broad sequence of actions that are that we do. So, I have deliberately been vague at certain points. The reason being that you know there are many more details that will sort of come in a next few slides, but the main idea is in the main take home point here is that memory is divided into many regions, one of these regions stores activation blocks.

What is an activation block? it is a continuous region of memory, that stores the execution context of a function and the execution context contains the arguments of a function, space for saving it is internal variables, space for saving the registers that it spills and also the re return address if the function is calling other functions and so pretty much that is it and once the function finishes execution, the activation block of the function is destroyed.

(Refer Slide Time: 25:06)



Using Activation Blocks - III

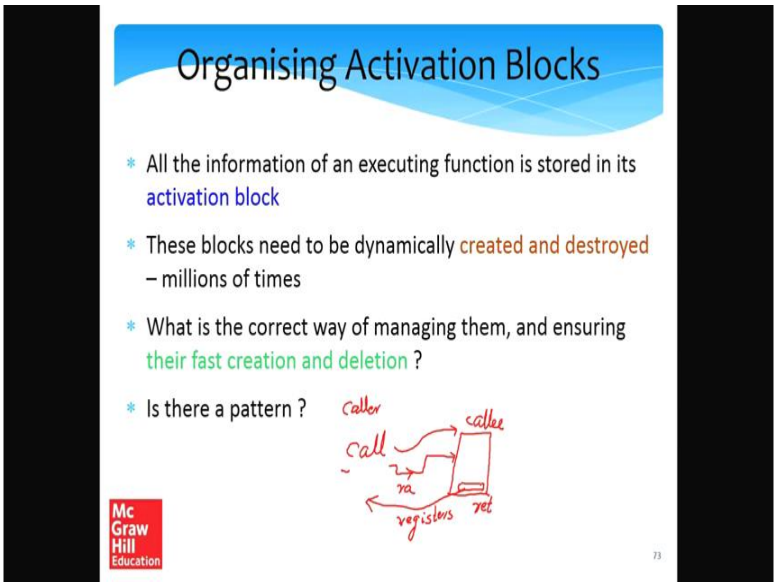
* Once the function ends (contd ...)
    * Call the ret instruction
    * and return to the caller
* The caller :
    * Retrieve the return values from the registers of from its activation block
    * Restore the spilled registers
    * continue ...

72

So, we are interview we are introduced 2 functions call and ret. So, we will discuss them in slightly more detail. So, the main idea of the ret instruction of the return instruction is that once the function ends, we call the ret instruction and return back to the caller. So, what does the collar do? It retrieves the return values from the registers or from it is activation block or sometimes you know so it depends upon the convention, that if let us say there is one large return value or many return values that are they written.

So, they might be written. So, they are typically written into registers or an activation block of the caller function, then in this case since we assume caller saved registers we restore the spilled registers and we continue it.

(Refer Slide Time: 25:57)



Organising Activation Blocks

- All the information of an executing function is stored in its activation block
- These blocks need to be dynamically created and destroyed – millions of times
- What is the correct way of managing them, and ensuring their fast creation and deletion ?
- Is there a pattern ?

Caller
callee
call
ra
ret
registers
73

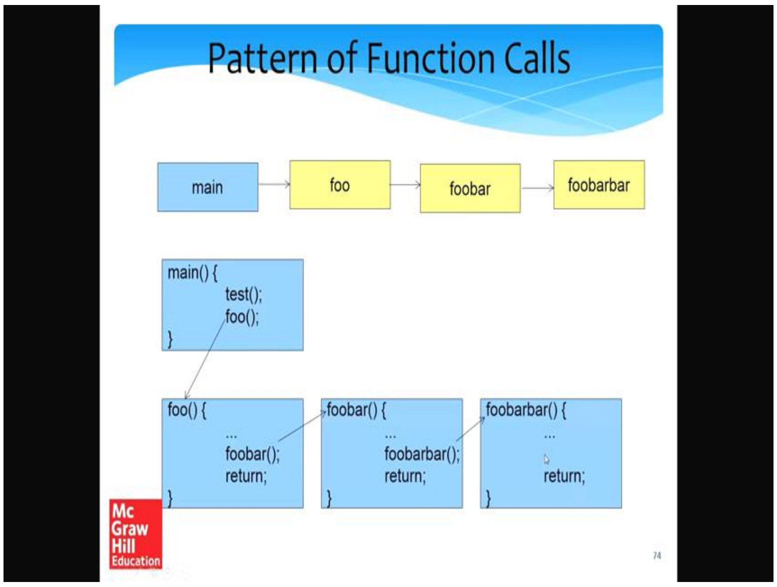
So, up till now we have discussed the main idea of an activation block, we have not really discussed more about the exact ways in which we want to implement them. But the basic idea is clear that we have 2 instructions call and ret; the call instructions what it does are a couple of things let me quickly write down. So, the call instruction what it does is like this, that it transfers the control to the beginning of the callee function.

That is one thing that the call instruction does; and additionally what it does it that it saves the address of the next instruction as a return address. So, this can be saved in many places. So, let us not talk about where exactly it is saved, but let us just refer to this piece of information as a return address, and this can optionally be saved in the activation block of the callee. Subsequently the callee does it is job, it computes this result, the return value which is what is the result of the function can come back either via registers that is the first option, the second option is that it can come back also via memory. In this case what the callee does is it writes back the result into the activation block of the caller, this is also possible.

Now, the idea is somewhat like this that. So, after the callee function finishes we invoke the ret instruction, the return instruction. The return instruction comes back to the return address, which is just one instruction after the call instruction and try it to executing the return instruction we have to destroy the activation block of the callee. So, the question now is that all the information of an execution of an executive function that is stored in activation block, has to be created millions and millions of times during the execution of a program, the reason being that a program typically you know function calls a very common instructions and even in one second millions of times functions are called.

So, we need a way of managing activation blocks, ensuring they are very fast creation and deletion. And we need to look at a certain pattern, to make this a very quick operation for all of us.

(Refer Slide Time: 28:39)



Pattern of Function Calls

```
main() {
        test();
        foo();
}
```

```
foo() {
        ...
        foobar();
        return;
}
```

```
foobar() {
        ...
        foobarbar();
        return;
}
```

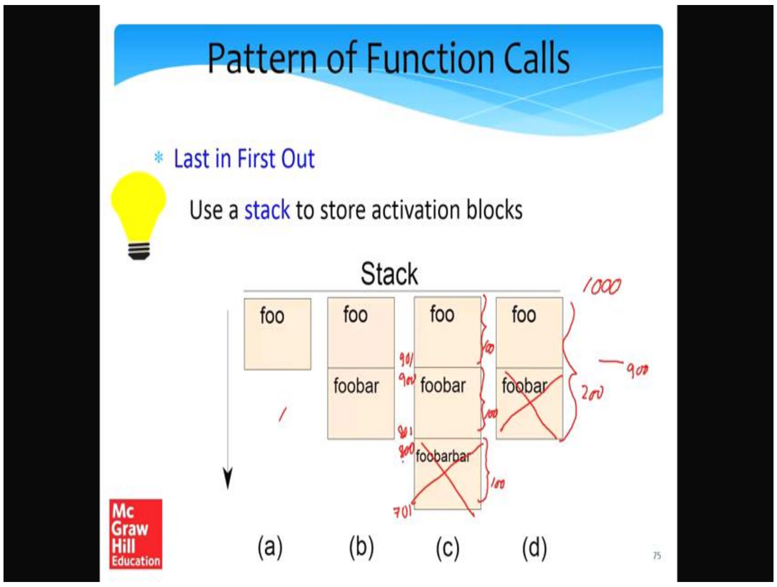
```
foobarbar() {
        ...
        return;
}
```

Let us not try to look for a pattern of executing function calls and see if there is something that can be done to actually represent the this information better and also create activation blocks very very easily. So, let us consider a main function which calls 2 other functions test and foo. So, let main call test so the activation block of test will be created and let test call test 2, so the activation block of test 2 will be created.

Once test 2 is done it will return; so the activation block of test 2 is not required anymore, so it can be destroyed. And once test is done test will return; so the activation block of test is not required anymore so it will be destroyed. Subsequently main will call function foo; so the activation block of foo will be created. So, let foo call foo bar; so the activation block of foo bar will be created. And let us foo bar call foo bar bar; so activation block of bar bar will be created.

Similarly, when we come back these activation blocks build destroyed in this sequence, but this is shown in the next slide.

(Refer Slide Time: 29:52)



Pattern of Function Calls

- Last in First Out

Use a stack to store activation blocks

So, if we consider the activation block of foo, then foo bar will be created, the activation block of foo bar will be created, then the activation block of foo bar bar will be created. And then it will be also be the first to be destroyed, then the activation block of foo bar will be destroyed and then lastly the activation block of foo. So, what do we see, the behavior here is last in first out. So, foo bar bar was the last you know was the latest function to be executed, and it is also the latest function whose activation block will be destroyed. So, whenever we here the term last in first out in computer science, it reminds us of a data structure called a stack, where a stack is something as simple as a stack of books right. So, we just stack one over the other, other, other so the last the book that we added the last is also the first that comes out.

So, similarly we can store activation blocks as a stack, where essentially we store foo then we store foo bar below a twist of foo bar bar below it and so these can be treated as memory regions. So, traditionally stacks are downward growing, what this means is? That if you want to create a new activation block, we create it in a lower memory address as compared to the previous activation blocks.

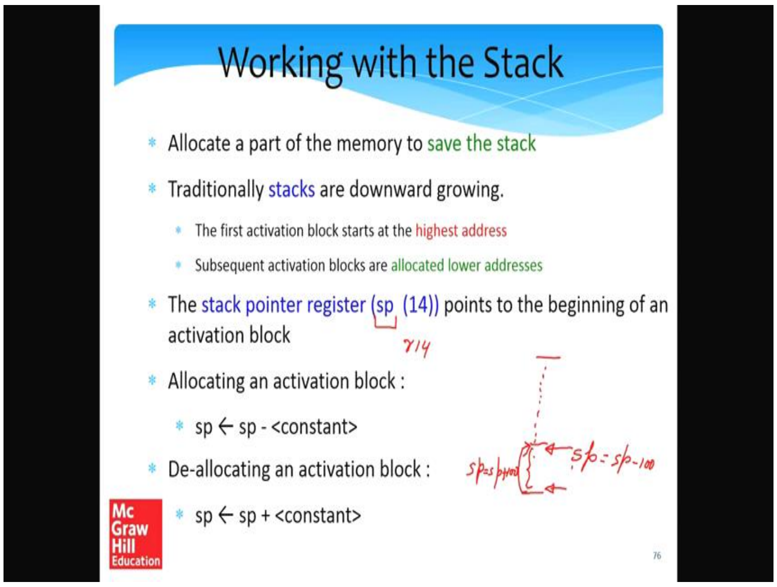
For example let us consider an example that make things easier, let us say that the stack begins at location 1000. Let us assume that the size of the ac activation block of foo is

100, foo bar is 100 and foo bar bar is 100. So, the first 100 bytes can be consumed by the activation block of foo, the next 100 by foo bar bar, and the next 100 by foo bar bar. Subsequently we delete this, so we store only 200 bytes, subsequently we again delete this and foo bar is over and we store only bytes from the region 1000 to 900.

So, this is where the notion of a stack comes from, that the stack is actually a programming language concept, where it denotes how data is stored and that the last in first out abstraction holds that a piece of information that is inserted into a stack at the end is also the first to come out. So, this is actually very similar if you see the pattern in which functions execute, they exactly execute like a stack. So, that is the reason we can store their activation blocks also as a stack. So, traditionally what you do is that stacks are a region of memory and they typically start at the highest region, which in this case is let us assume the largest address we have is a 1000 and subsequently we start storing their activation blocks in the manner that I have shown.

So, maybe the first one it is can be stored from 901 to1000 and the second from 8010 to 900 and the third from the address of 701 to 800.

(Refer Slide Time: 33:21)



Working with the Stack

- Allocate a part of the memory to save the stack
- Traditionally stacks are downward growing.
  - The first activation block starts at the highest address
  - Subsequent activation blocks are allocated lower addresses
- The stack pointer register (sp (14)) points to the beginning of an activation block
- Allocating an activation block :
  - sp ← sp - <constant>
- De-allocating an activation block :
  - sp ← sp + <constant>

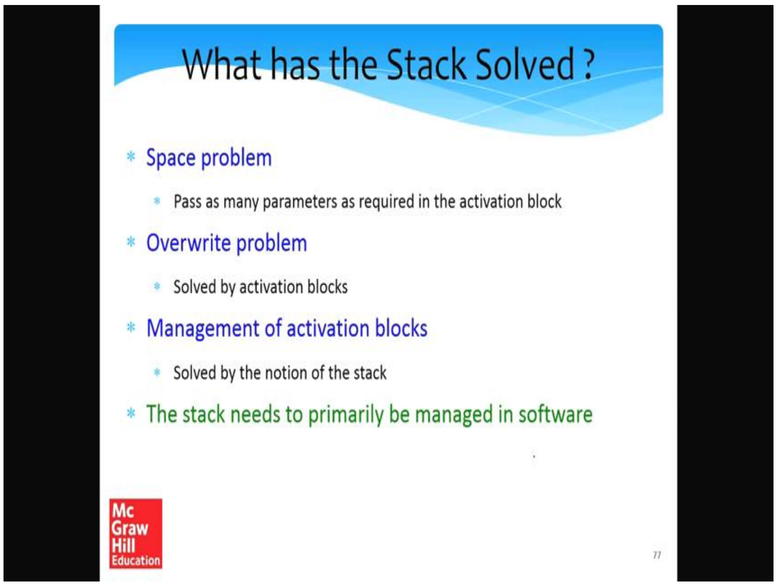
76

So, now working with the stack; so we allocate a part of the memory to save the stack as they have mentioned, stacks are downward doing. So, we dedicate one register in our set of 16 registers; particularly register 14, r 14. We call it the stack pointer register or it is also called as sp, it is also referred to as sp; so sp or as r 14. So, this points to the beginning of an activation block and so basically it also you know sort of points to the top of the stack right.

So, for example, if the stack begins at this point and then we have activation blocks this is retains. So, this is where exactly the stack pointer should point. So, you said basically the beginning of the next activation block with where the stack is ending. So, allocating an activation block is very simple, all that we do is we subtract the stack pointer by the size of the activation block if it is 100. A new stack pointer will point to this location and this region in memory becomes it is activation block. So, essentially we have booked it, we later on need to fill it with information.

De-allocating also requires a single instruction where we essentially do sp is equal to sp plus the size of the activation block. In this case if it is 100, then we sort of fold it back up again. So, essentially allocating and de-allocating an activation block is very simple, to allocate an activation block we subtract the stack pointer by the size of the block which is a constant and it should be known before hand. Similarly to de-allocate the block, we what we do is that we add the size of the block which is a constant and is known beforehand to the stack pointer. So, it is very simple allocating and de-allocating only a single instruction. So, this is also what, which is also what we wanted. So, we are using the property, the last in fist out property which has made our life very simple.

(Refer Slide Time: 35:37)



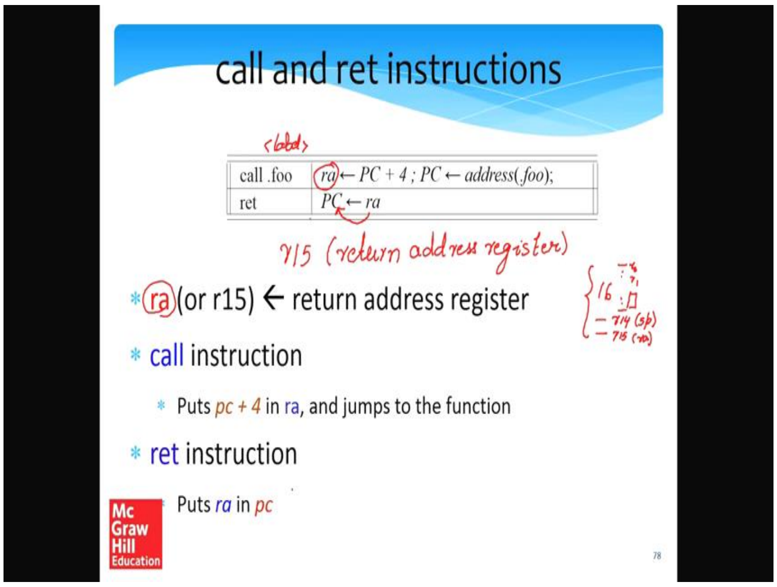
What has the Stack Solved ?

* Space problem
  * Pass as many parameters as required in the activation block
* Overwrite problem
  * Solved by activation blocks
* Management of activation blocks
  * Solved by the notion of the stack
* The stack needs to primarily be managed in software

So, what are the stack salt for us? It is definitely solve this space problem. So, we can pass as many parameters as required via this activation block mechanism. So, you know if the numbers of parameters are few, the numbers of arguments to a function are few, they can be passed via registers; otherwise they can be stored in regions within an activation block and they can be passed. The stack has also solved overwrite problem. So, now, functions do not overwrite each other's data, they have their own regions in memory where they write their own temporary variables. Managing the activation blocks is much easier because they are saved in a stack. So, creating new ones are getting rid of old ones is very easy, we just need to manipulate the stack pointer that is all using regular add and subtract instructions.

So, the stack in most instruction sets is actually a software construct. So, there is a broad agreement that a certain register needs to be treated as the stack pointer, but other than that it is mostly managed in software, which basically means that when you are writing assembly code you have to explicitly manage the stack. So, we will see an example of that, but essentially the programmer or the compiler have to add instructions to explicitly manage the stack and sort of increment and decrement the stack pointers and create the activation blocks as in when required.

(Refer Slide Time: 37:12)



call and ret instructions

<label>

call .foo	$ra \leftarrow PC + 4 ; PC \leftarrow address(.foo);$
ret	$PC \leftarrow ra$

r15 (return address register)

* ra (or r15) ← return address register
* call instruction
  * Puts *pc + 4* in ra, and jumps to the function
* ret instruction
  * Puts *ra* in *pc*

16
r14 (sp)
r15 (ra)
Mc Graw Hill Education
78

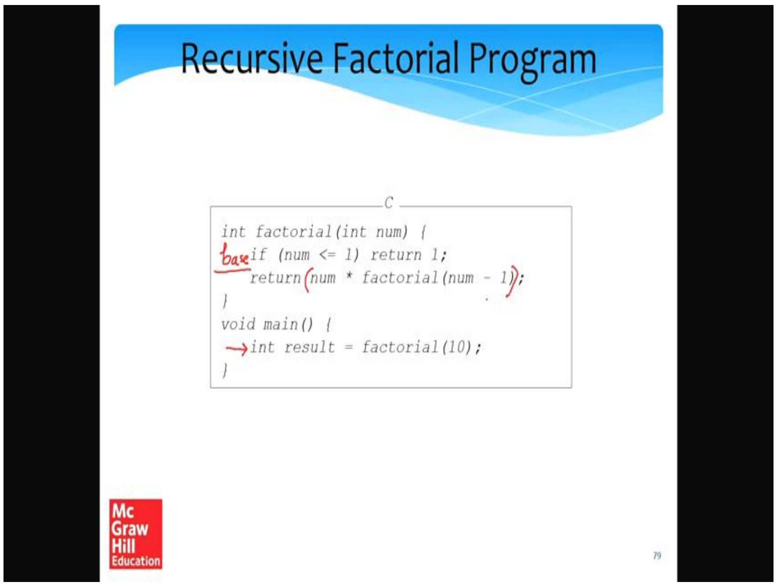
Let us now discuss the call and ret instructions. So, we have introduced the notion of calling a function and returning from a function, but that was more from a high level conceptual theoretical angle. It was not from a practical point of view of how it is implemented. So, before actually I described these instructions, it would be not worthy to first introduce the return address register, which is actually a register in our simple RISC instruction set, but there are similar concepts in arm and x 86 as we shall see in the next 2 chapters.

So, what we shall do is that let us designate the 15th register r 15 as a return address register. So, we will also refer to this register as r a or r 15 interchangeably. So, what the call instruction does is that the call instruction computes the address for the next instruction, which is the current PC plus 4 and saves it in r a the return address register, and then subsequently it sets the program counter, to the address of the function that we want to go to. So, the assembly language semantics is call label, where label is essentially pointing to the first instruction in the function and the way that this gets translated in hardware is that we set the return address and we said that PC is the program counter to the address of that corresponds to the dot foo label.

A note that we will have these dot marks before labels, it is more of a convention, but sin the simple RISC interpreter does not require dots in front of labels. The return instruction is very simple we read in the contents of the return address register to the program counter. So, this is essentially like a jump. So, in our register file that contains 16 registers, the last 2 registers namely r 14 and r 15 are reserved for special purposes; r 14 will be the stack pointer and r 15 is the return address register. So, all the other registers can be used by the program. So, here again there is some amount of semantics a different instruction sets typically define. So, it is not specific to simple RISC, but typically r 0 and r 1 would be used to pass parameters and the rest of the registers will be used to process temporary values.

So, here again as it is written for a call instruction we put PC plus 4 in r a, and jump to the function and in the return in the ret instruction you put the value of r a in the PC.

(Refer Slide Time: 40:35)

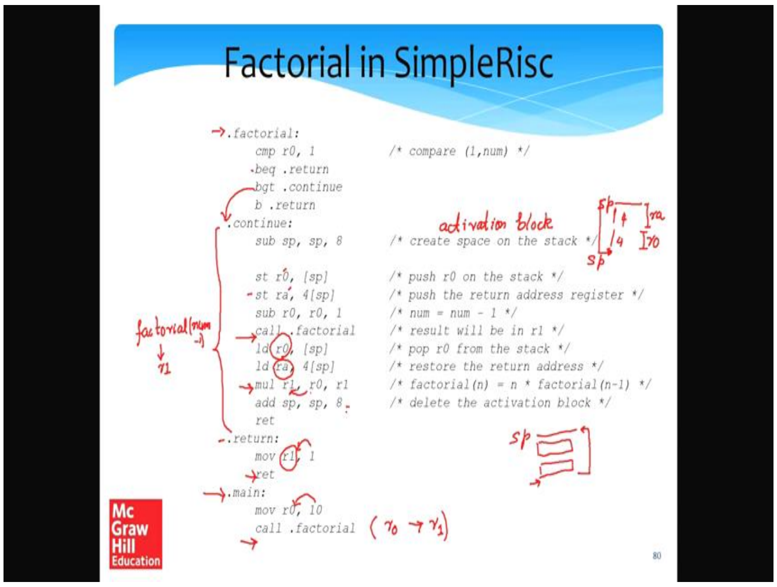


Recursive Factorial Program

```
C
int factorial(int num) {
    if (num <= 1) return 1;
    return (num * factorial(num - 1));
}
void main() {
    int result = factorial(10);
}
```

So, it is time that we write our first program in simple RISC assembly; that uses the stack as well as the call and ret instructions. So, let us write a recursive variant of the factorial function. So, in this case this is c program we will start from this line, let us compute the factorial of 10. So, inside the factorial function we will first have the base case. So, the base case says if num is less than equal to 1 return 1. So, this is where the recursion will terminate, otherwise we return num times factorial num minus 1. So, here we compute 10 times, 9 times, 8 times 7 and so on.

(Refer Slide Time: 41:27)



Factorial in SimpleRisc

```
.factorial:
    cmp r0, 1               /* compare (1,num) */
    beq .return
    bgt .continue
    b .return
.continue:
    sub sp, sp, 8           /* create space on the stack */

    st r0, [sp]             /* push r0 on the stack */
    st ra, 4[sp]            /* push the return address register */
    sub r0, r0, 1           /* num = num - 1 */
    call .factorial         /* result will be in r1 */
    ld r0, [sp]             /* pop r0 from the stack */
    ld ra, 4[sp]            /* restore the return address */
    mul r1, r0, r1          /* factorial(n) = n * factorial(n-1) */
    add sp, sp, 8           /* delete the activation block */
    ret
.return:
    mov r1, 1
    ret
.main:
    mov r0, 10
    call .factorial
```

So, to convert this into simple RISC is actually easy. So, let us start with the dot main labels where the main function will start. We move 10 into r 0 because we will be computing the factorial of 10 and then we call the factorial function. So, this takes us way to the top over here. So, let us first test for the base case that whether we should continue the recursion or not. So, as we shall see here we compare the number that is coming in the argument r 0 with 1. So, if there is equality it basically means that r 0 is equal to 1. So, we can just return.

So, I jump to the return point which is over here. If r 0 is greater than 1 then we need to do the recursion, so we jump to dot continue otherwise the only case that is remaining is r 0 is less than 1 then also we jump to the return label here. In the return label we set the result, which is the result of the factorial function to 1, because factorial of 1 is 1 and factorial of 0 is 1. So, we set the final result r 1 to be equal to 1 and we return from the function.

So, when we return we come to the next instruction after call factorial and this will find the result in r 1. So, when think of the factorial function of taking it is argument in r 0 and then returning it is result in r 1. So, the mean fun is in this part in the continued part which is actually the recursive part. So, the first thing is that we create the activation block over here. So, we create space on the stack. So, we create the activation block all right. So, if you want space to save 2 variables. So, 2 variables means 2 integers, a 1 integer is 4 bytes. So, for 2 integers it would be 8 bytes. So, what we do is that we create an activation block here; we subtract 8 from the stack pointer. So, essentially this instruction has if sp equals sp minus 8. So, we create some space. So, in the original stack pointer was over here the new stack pointers will be. So, 4 bytes, another 4 bytes a new stack pointer will point here.

Now, what we do is that. So, since we are calling another version of the factorial function itself it is necessary to save some information on the stack, otherwise the called function the callee will overwrite those values. So, what we do is that at the stack pointer which is essentially the, if this is a new location of the stack pointer, the top 4 bytes you know this location, we say r 0 which is the argument, which is essentially what came from the function that call this function so, we save it over here.

And in 4 sp which is the next location we save r a, this instruction over here we save the return address, because this also might get over it. Subsequently we subtract 1 from r 0 it is essentially num minus 1 and we call factorial once again. So, essentially we call one more copy of the factorial function. So, that will again call one more copy of the factorial function and so on. So, the same way that recursive functions execute. So, at this point the assumption is that the factorial of num minus 1 is stored in r 1. So, that is the assumption, because all factorial functions including the base case are supposed to keep the result in r 1 and that is the way that we have coded it.

So, that is we will use the value of r 1 later, let us just restore our values; because we had spilled some registers let us restore them. So, recall that our stack pointer, which is supposed to have remained, untouched. So, we will see how let us load back the value of r 0, which is something that we had stored over here. Let us load back the value of r a, which we had stored over here. So, basically values of r a and r 0 are restored and this particular scheme that we have followed here is a caller saved scheme, because the caller is saving the registers that it means later on restoring them. Subsequently we have multiply r 0 time's r 1. So, r 0 is basically a num and r 1 is factorial of num minus 1.

So, essentially we are computing factorial n is equal to n times factorial n minus 1 and the results are being saved in r 1, which we have always wanted to do. Now since the result is in r 1 let us get rid of our current activation block, so recall that we started the activation block with subtracting the stack pointer 8 from the stack pointer. Let us now delete the activation block by folding the stack pointer back, so what we do is, we set the stack pointers value back to it is old value. So, this is done by adding 8 to the stack pointer. So, think about this at the beginning of the function the stack pointer was here we needed to save 2 local variables; we need to spill 2 registers actually. So, the new stack pointer pointed here, once the function was over we restored the value of the stack pointer to again point to where it was originally pointing.

After this since the value is an r 1, we return back. So, it is easy to prove by any technique that uses mathematical induction that this program will always work, in the sense every copy or the factorial function sees the result in r 1. So, the base case at the dot return labels saves the result in r 1 and even the recursive case saves the result in r 1, the reason being that it calls many many copies of itself, each of them save the result in r

1 finally, in this line over here we multiply n times factorial n minus 1 and save it in r 1. Lastly it is very important to delete the activation block, which is being done in this line.

(Refer Slide Time: 48:59)

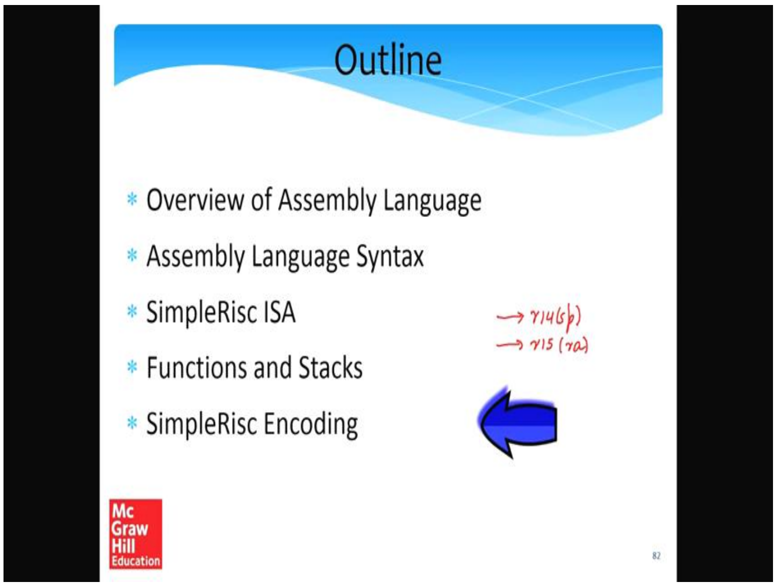
nop instruction

* nop → does nothing
* Example : nop

Mc Graw Hill Education
81

So, here is a chance that we get to introduce one instruction that does nothing at all. So, it is a nop instruction it does nothing at all, why on earth would we require it. This will be common; this will be explained to you in chapter 9 when we discuss a pipelined processor that is when I will explain why we need this particular instruction not now all right.

(Refer Slide Time: 49:30)



Outline

* Overview of Assembly Language
* Assembly Language Syntax
* SimpleRisc ISA
* Functions and Stacks
* SimpleRisc Encoding

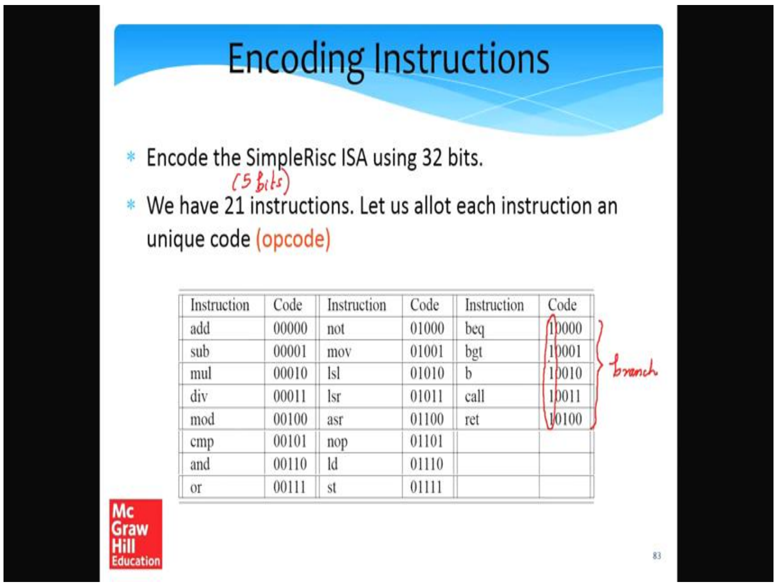
$\rightarrow r14(sp)$
$\rightarrow r15\ (ra)$

Mc Graw Hill Education
82

So, we have seen up till now a basic overview of assembly language, the syntax of assembly language. A simple instruction set, then we have looked at functions and stacks which is an advanced feature in programming languages. So, it is also possible to implement other kinds of features such as object oriented programming and so on, but these are all variants of this and I did not want to heavily complicate it, but users would realize that using the basic instruction set provided, a lot of these high level features can be implemented.

Typically in most instruction sets we only designate 2 kinds of registers as special: one is the stack pointer as we have done, and the other is the return address r a ARM and x 86 of some other special registers we will talk about it when we discuss those instruction sets. Now let us look at an encoding of the instruction set in binary bits. So, basically if you have 32 bits, we need to be able to encode all the instructions. Because after all, the computer does not understand assembly it only understand 0s and 1s.

(Refer Slide Time: 50:56)



Encoding Instructions

- Encode the SimpleRisc ISA using 32 bits.
- We have 21 instructions. Let us allot each instruction an unique code (opcode)

(5 bits)

Instruction	Code	Instruction	Code	Instruction	Code
add	00000	not	01000	beq	10000
sub	00001	mov	01001	bgt	10001
mul	00010	lsl	01010	b	10010
div	00011	lsr	01011	call	10011
mod	00100	asr	01100	ret	10100
cmp	00101	nop	01101		
and	00110	ld	01110		
or	00111	st	01111		

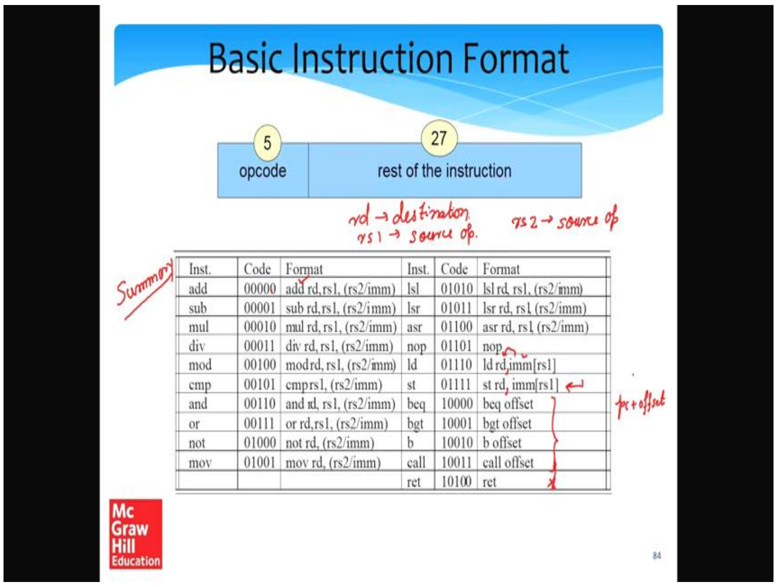
branch

83

So, every assembly instruction needs to be encoded in simple 0s and 1s. So, we have 32 bits to encode the instructions, because it is a 32 bit instruction set and we up till now have introduced 21 instructions. So, let us start out by assigning an opcode and operation code or an instruction code to every single instruction, and the instruction code has to be unique for example, add there are 21 instructions. So, we will need 5 bits right why 5 bits? Because 4 bits will not do we can only encode 16 instructions. So, we need 5 bits.

So, the add instruction gets a code of 0 then we first have all the arithmetic instructions, then we have the load and store instructions and lastly we have the branch instructions. So, why instructions have been assigned you know this kind of codes will actually be clear to you in chapter 8 when we discuss decoding. So, all that I can tell you right now is that, it should be easy to find out you know what is the type of an instruction. So, for example, the branch instructions I can easily find that an instruction is a branch by actually taking a look at the most significant bit which is 1. Similarly other kinds of tricks have been put for other kinds of instructions, so we will get a chance to discuss all of this in the chapter on decoding.

(Refer Slide Time: 52:31)



Basic Instruction Format

5
opcode
27
rest of the instruction
rd → destination
rs1 → source op.
rs2 → source op
Summary
pc + offset
Mc Graw Hill Education
84

Inst.	Code	Format	Inst.	Code	Format
add	00000	add rd, rs1, (rs2/imm)	lsl	01010	lsl rd, rs1, (rs2/imm)
sub	00001	sub rd, rs1, (rs2/imm)	lsr	01011	lsr rd, rs1, (rs2/imm)
mul	00010	mul rd, rs1, (rs2/imm)	asr	01100	asr rd, rs1, (rs2/imm)
div	00011	div rd, rs1, (rs2/imm)	nop	01101	nop
mod	00100	mod rd, rs1, (rs2/imm)	ld	01110	ld rd, imm[rs1]
cmp	00101	cmp rs1, (rs2/imm)	st	01111	st rd, imm[rs1]
and	00110	and rd, rs1, (rs2/imm)	beq	10000	beq offset
or	00111	or rd, rs1, (rs2/imm)	bgt	10001	bgt offset
not	01000	not rd, (rs2/imm)	b	10010	b offset
mov	01001	mov rd, (rs2/imm)	call	10011	call offset
			ret	10100	ret

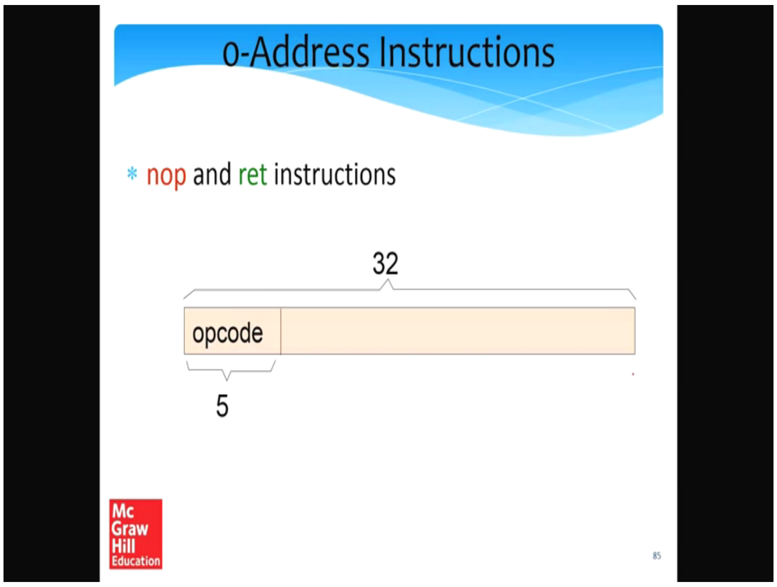
All right. So, up till now we have our basic instruction format, we have a 5 bit opcode, and the rest of the 27 bits is there to encode the rest of the instruction and here is a quick summary of the instruction set, that you can actually you know markdown for your own use, because this would be very helpful in the chapters on processor design. So, every instruction first has the instruction name; then r d is the destination register. So, let us say r d is the destination register, r s 1 is the first source operand all right r s 2 is the second register source operand and m is an immediate. So, as we can see all arithmetic instructions are of the same type, you first specify the destination then the first source register.

The second operand can either be a register or an immediate; then we have the compared instruction which does not have a register destination. Subsequently we have the logical instructions which also have the same format, and also the shift instructions having the same format, then nop instruction does is essentially 0 address instruction, it does not have take any operands at all the load instruction should actually be a comma. So, the load instruction essentially takes the first source register and an immediate and saves it in the destination. The store instruction is somewhat special in the sense that it does not actually have a register destination, but we shall see later that you know a compromise needs to be made and this is what the format of the store instruction should be, because mainly because we will not have enough space, but this is best described later we have a slide for it.

Subsequently you are the branch instruction. So, in assembly language although the 5 branch instructions branch to a label, but actually when we shall encode it in hardware, all of these 4 branch instructions I am sorry not ret, but these 4 branch instructions will actually jump to an offset. So, essentially it will be PC plus the offset is what the branch instructions will be jumping to. An offset relative to the current PC there are some more tricks we will discuss that, but essentially it is an integer offset that needs to be added to the current program counter, which will become the new program counter.

So, we will essentially come back to both of these issues; one of them is the format of the store instruction and the other is the offset in the branch instruction that we shall use; and the ret instruction is simple. It is a simple 0 address instruction, so it does not take any arguments.

(Refer Slide Time: 55:48)

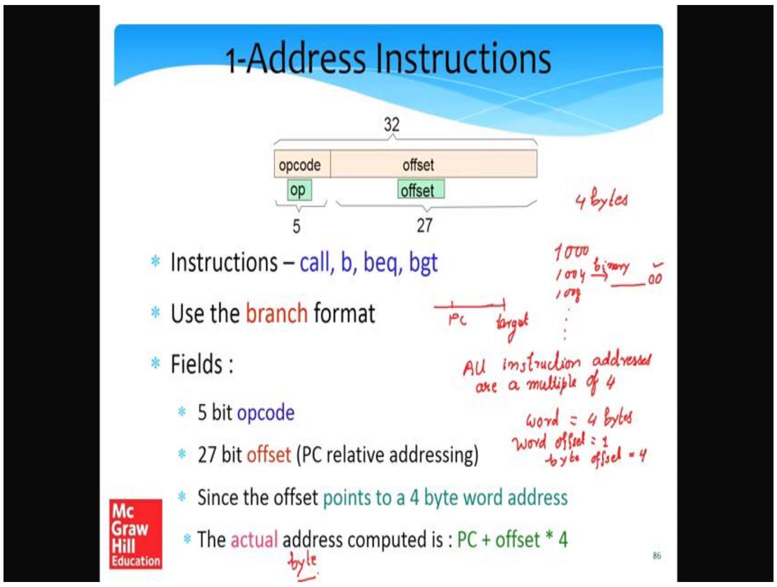


0-Address Instructions

* nop and ret instructions

So, let us first look at the low hanging fruits, which are the nop and ret instructions. So, since they do not take any arguments their encoding is very simple, they will only have the 5 bit opcode and the remaining 27 bits are useless. So, they need not be used and all that we require is the 5 bit opcode, which is in the most significant position these are all 0 address instructions.

(Refer Slide Time: 56:16)



1-Address Instructions

32
opcode
offset
op
offset
5
27
4 bytes
1000
1004 $\xrightarrow{binary}$ ___00
1008
PC
target

* Instructions – call, b, beq, bgt
* Use the branch format
* Fields :
  * 5 bit opcode
  * 27 bit offset (PC relative addressing)
  * Since the offset points to a 4 byte word address
  * The actual address computed is : PC + offset * 4

byte

All instruction addresses are a multiple of 4

word = 4 bytes
word offset = 1
byte offset = 4

Now, let us look at our one address instructions. So, we have 4 such instructions they are called unconditional branch b, branch if equal beq, branch if greater than b g t. So, they use what is called the branch format which will define right. Now the top 5 bits remain the same the 5 bit opcode. The remaining 27 bits specify an offset, this PC relative addressing. So, this is a number it is a sign number, which is added to the PC. So, it essentially we can add something or subtract something. So, as I said it is a 27 bit sign number which is added to the PC.

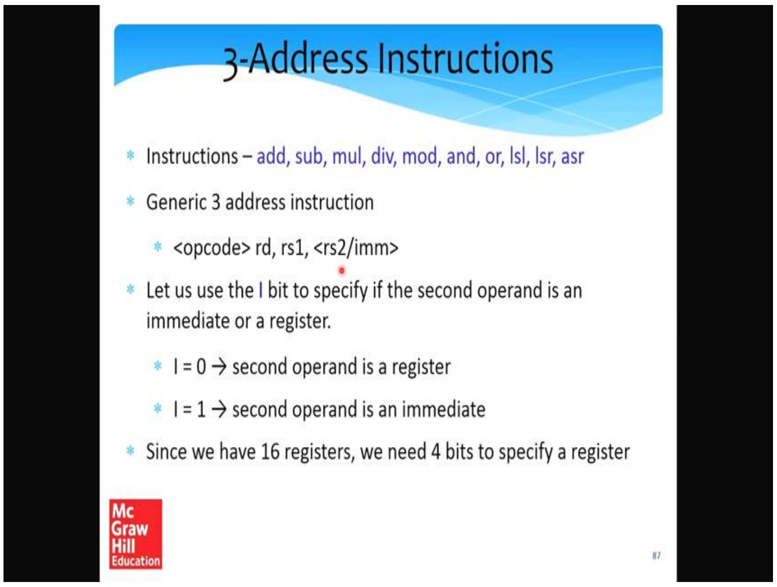
Now here is the trick that should be mentioned. So, the trick is some something like this that let us assume that the offset points to a byte. So, all our instructions are 4 bytes each right. So, if all that instructions are 4 bytes each and we assume that you know all instructions start at an address that is a multiple of 4. So, we will have addresses of this type, 1000 1004 1008 and so on right. So, all of these numbers are multiples of 4; say any number that is a multiple of 4 when it is written in binary, which is the language that the computer understands, it is last 2 bits will be 0 0.

Subsequently we will have some bits. So, here then the obvious response would be that let us not waste space, in specifying 2 bits whose value we already know, which are 0 and 0. So, it is a much better idea to instead you know the instead work with a simple assumption, the assumption is that all instruction addresses. So, let me just write it down for added emphasis are a multiple of 4. If we make this assumption then any offset if it points to a byte, the offset will also be a multiple of 4 and last 2 bits will be 0 and 0, which is not something that we want, so instead of making it a byte offset, we make it a word offset.

So, what is the word? A memory word we have defined before, but let me just define once again, a word is basically 4 bytes. So, a word offset is basically the number of additional memory words. So, let us say the word offset is 1; this means that the byte offset will actually be 1 times 4 which is 4. So, the byte offset is equal to the word offset times 4. So, the actual address that we will compute in this case will actually be PC plus offset times 4, and the reason that we are doing this is because the actual address is a byte address right. So, this precisely points to a byte.

But since we are assuming that all the addresses are a multiple of 4, we need not waste 2 bits for storing the offset because bits are very valuable. So, we actually save the word offset, which is the number of memory words that is certain instruction the. In the number of memory words between the current PC and the branch target right the target of the branch. So, this offset can either be positive or negative, but in any case it is a word offset. So, to get the byte offset we multiply it by 4 and we add it to the PC.

(Refer Slide Time: 60:53)



3-Address Instructions

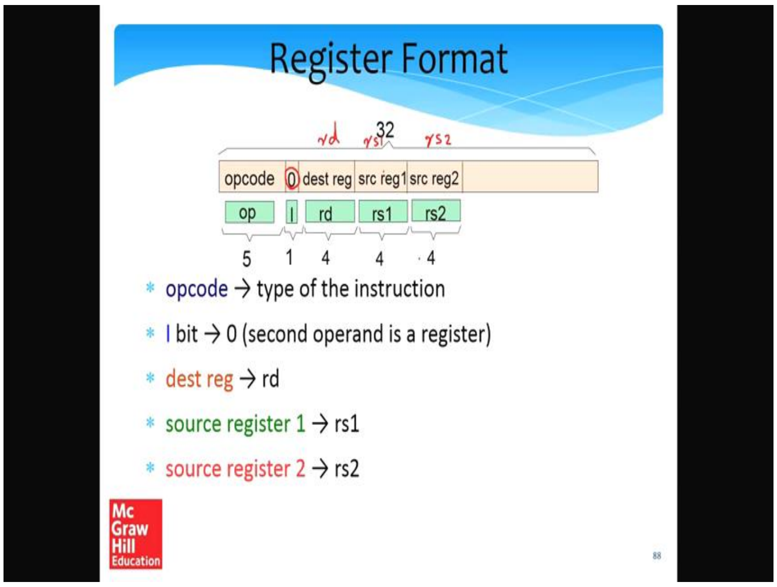
* Instructions – add, sub, mul, div, mod, and, or, lsl, lsr, asr
* Generic 3 address instruction
  * <opcode> rd, rs1, <rs2/imm>
* Let us use the I bit to specify if the second operand is an immediate or a register.
  * I = 0 → second operand is a register
  * I = 1 → second operand is an immediate
* Since we have 16 registers, we need 4 bits to specify a register

87

Let us now look at the 3 address instructions that we have. So, all our arithmetic instructions are 3 address instructions, all our logical instructions are 3 address instructions, and also all are shift instructions are also 3 address instructions. The standard format of the 3 address instruction is the opcode, the destination register, the first source register, then either a second source register or an immediate. So, the first 4 point the decision point that we have is to resolve whether we will use the second source register or an immediate.

So, let us use the I bit to specify; if the second operand is a register or an immediate. So, if I is equal to 0 let us say that the second operand is a register and if I equal to 1 let us say that the second operand is an immediate. So, basically this is one extra bit that we need to accommodate in an instruction format and it is only role is to say whether the second operand is a register or an immediate. After that we need to encode the value of the registers; since here 16 registers, we would need 4 bits to specify a register. So, we will see what the format looks like in the next slide.

(Refer Slide Time: 62:22)



Register Format

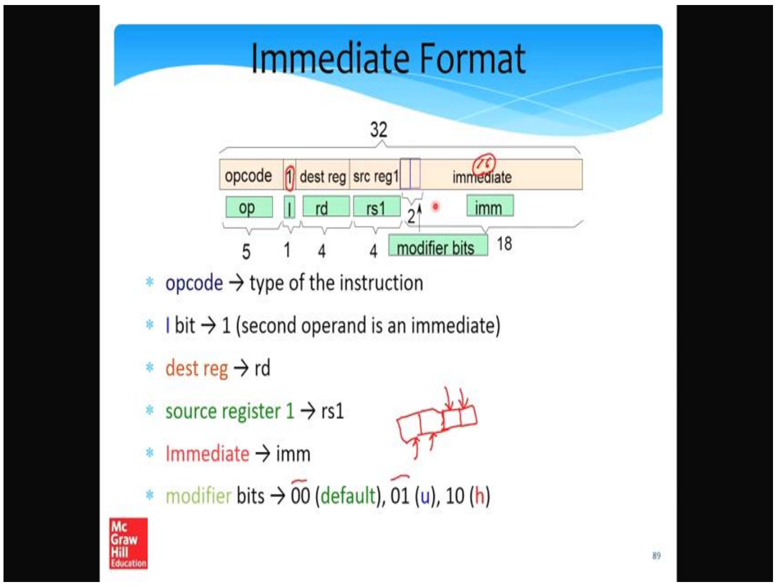
rd
rs1
rs2
32
opcode
0
dest reg
src reg1
src reg2
op
I
rd
rs1
rs2
5
1
4
4
4

- opcode → type of the instruction
- I bit → 0 (second operand is a register)
- dest reg → rd
- source register 1 → rs1
- source register 2 → rs2

So, what we have done here is that we have reserved space for our 5 bit opcode, which is the first thing that we need to do, after that we have the I bit. So, we are discussing the register format here where the second operand is a source register it is not an immediate value. So, basically in this format the first 5 bits will be the opcode and I bit will be 0 because the second operand is a register, but this will nevertheless require one bit in our format.

Then we need to specify r d which is the destination register, the first source register and the second source register. So, each of these is 4 bits, because we have 16 registers it will take 4 bits to encode them. So, we will have 4 bits each. So, the total number of bits is 5 plus 1 6, plus 4 10, 14 and 18 which is fine. So, we were able to encode such kind of instructions in 18 bits, which is less than 32. So, we will refer to this particular encoding format as the register format.

(Refer Slide Time: 63:39)



Immediate Format

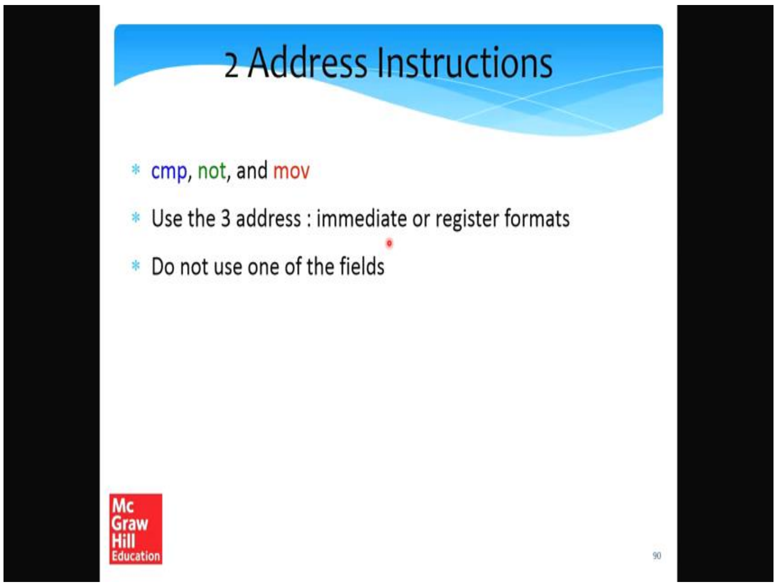
- opcode → type of the instruction
- I bit → 1 (second operand is an immediate)
- dest reg → rd
- source register 1 → rs1
- Immediate → imm
- modifier bits → 00 (default), 01 (u), 10 (h)

Now, let us consider the case where we have the 3 address instruction and the second operand is an immediate. So, we will start the same way, we will have a 5 bit opcode, we will have one bit for the immediate. So, one bit over here and it is value needs to be 1, because the second operand is an immediate then we will have a registered destination 4 bits for that, the first source register 4 bits for that. So, total number of bits you have consumed up till now is 5 plus 1 6, plus 4 10 plus 4, 14 you have 18 bits left which is exactly what we need.

So, we will have a 16 bit immediate and we can put in our 2 modifier bits over here. Recall that the modifier bits have we defined 3 kinds of modifiers 0 0 is a default modifier, 0 1 is a u modifier where we place the 2 bytes at the m s b positions, and I am sorry 0 1 is the unsigned modifier, but it place the 2 bytes at the l s b positions right.

So, basically these are the 4 bytes, then with the unsigned the 2 bytes were placed over here and 1 0 is the h modifier where they were actually placed in the m s b positions. So, we will since we have 3 modifiers, we will require 2 bits to encode them. So, these 2 bits can be stored over here just after the first source register the 2 bits can be saved over here. Subsequently we can store a 16 bit immediate, which is assigned twos complement number. So, as we can see this format is very tight in the sense all the bits are being used, the total all the 32 bits every single bit is being used and every bit is associated with storing some or other information associated with the instruction.

(Refer Slide Time: 65:49)

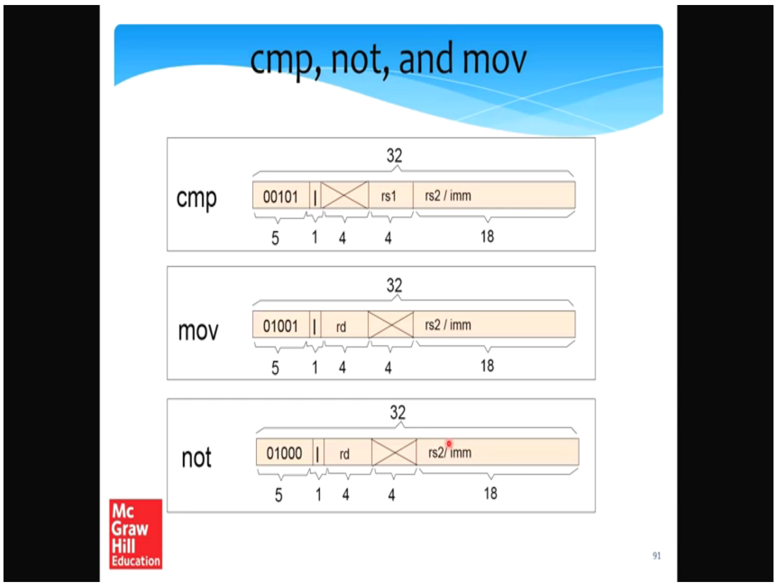


2 Address Instructions

- cmp, not, and mov
- Use the 3 address : immediate or register formats
- Do not use one of the fields

Now, let us take a look at 2 address format instructions like compare not and mov. So, we will use the simple solution which is used in many other areas of computer architecture as well; we will use the 3 address format, we can use either the immediate or register formats, but we will not use every single filed.

(Refer Slide Time: 66:17)

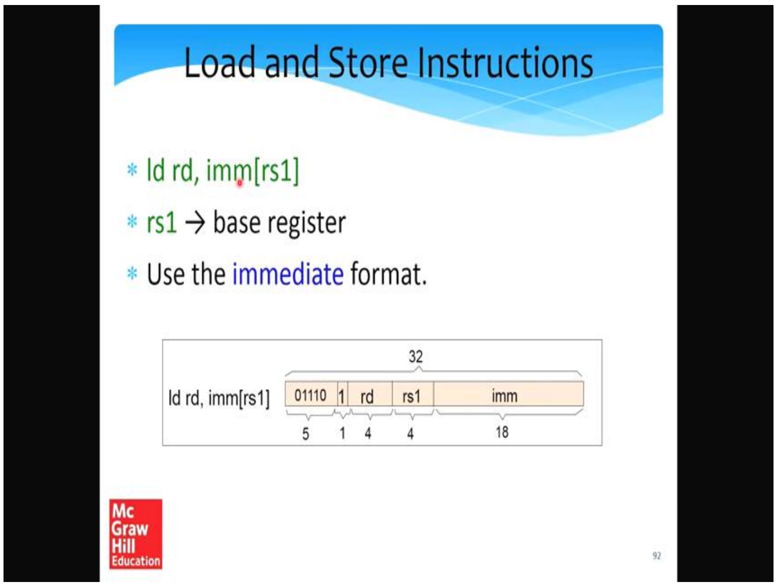


So, we will keep some fields all unoccupied for example, the compare instruction it does not have a destination. So, we will do nothing with a destination field, but we can keep the first source operand; the first source register and the second operand, which can either be a register or an immediate.

The mov instruction will not have the first source operand, but we will have a destination register and the second source operand can either be a register or an immediate. Similarly the not instruction will be similar to the mov instruction, it will not have a source register it will not have it is first source register or it will have a destination register and we will have the r s 2 slash m, where we will have either a register we have a single operand, which can either be a register or an immediate; so we will use the r s 2 field. So, in this way we have essentially we are ensuring that are decoding hardware, which is the hardware needed to understand an instruction, we are ensuring that the hardware is simple, because we will have only a very few number of predefined formats.

So, up till now we have only defined the branch format, the immediate format and the register format; all the instructions we are trying to fit in those particular formats.

(Refer Slide Time: 67:46)

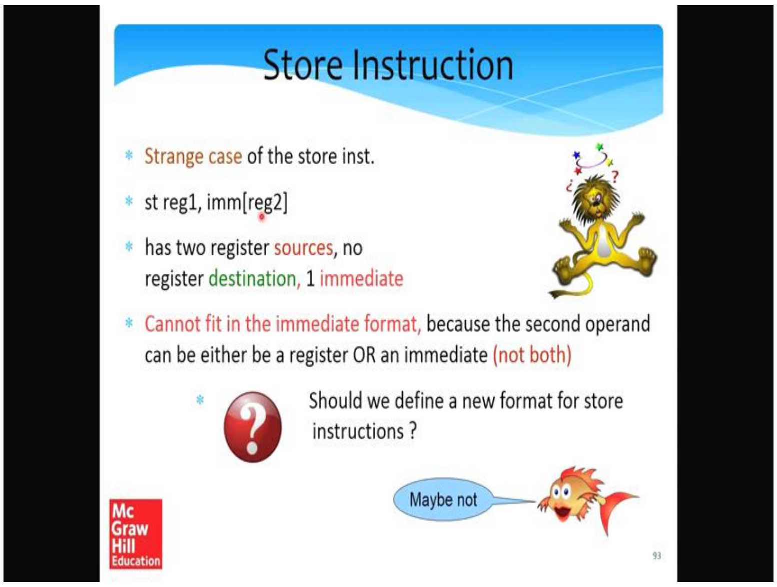


Load and Store Instructions

- ld rd, imm[rs1]
- rs1 → base register
- Use the immediate format.

Let us now discuss the load and store instructions. So, load instruction has a register destination. So, we are using the first 5 bits for the opcode, the immediate is one there is a register destination. The base register will save in r s 1 field; it has an immediate which is the offset so the immediate can be stored in the immediate field. So, this is exactly the immediate format that we are reusing for the load instruction as well. So, just to repeat the load instruction has a register destination r d, which is saved over here. It has a base register which is which we are using as the first source register r s 1; it has an immediate which we are considering to be the second operand.

(Refer Slide Time: 68:39)



Store Instruction

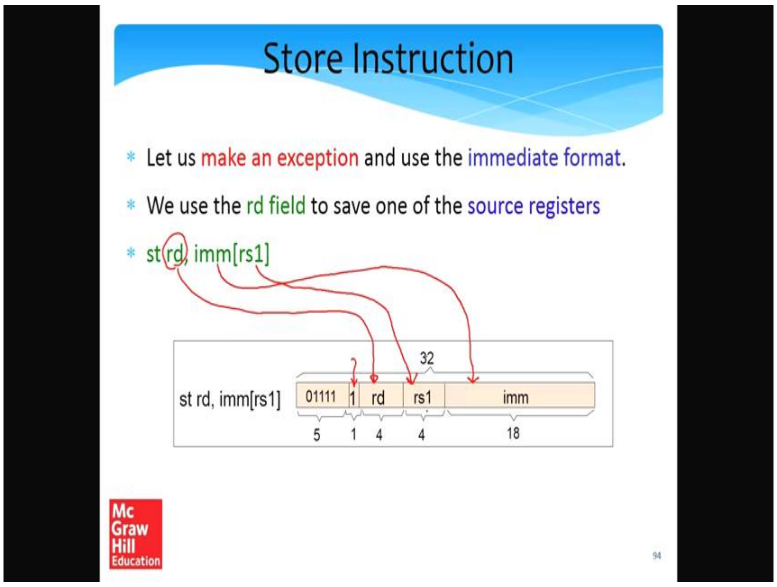
- Strange case of the store inst.
- st reg1, imm[reg2]
- has two register sources, no register destination, 1 immediate
- Cannot fit in the immediate format, because the second operand can be either be a register OR an immediate (not both)
  - Should we define a new format for store instructions ?

Maybe not
Mc Graw Hill Education
93

Let us now discuss the slightly strange case of the store instruction and. So, hopefully after the end of this slide, readers will appreciate some of the tradeoffs that need to be done by computer architects, to ensure that their hardware is simple and everything remains elegant. So, the case of the store instruction is somewhat like this; that in a story instruction, we have actually 2 sources: one of them is a register source, which provides the data that needs to be stored in a memory address, in the memory address reg 2 is the base address and m is the offset.

So, essentially it has 2 register sources: reg 1 and reg 2 from which we get data; the destination is memory. So, there is no register destination and there is one immediate. So, we cannot fit this in the immediate format, because the second operand can either be a register or an immediate are not both and we have 2 sources. So, what do we do? Should we define a new format for store instructions answer is maybe not; let us try to use this reuse some existing format. So, let us make an exception and use the immediate format.

(Refer Slide Time: 69:55)



Store Instruction

- Let us make an exception and use the immediate format.
- We use the rd field to save one of the source registers
- st rd, imm[rs1]

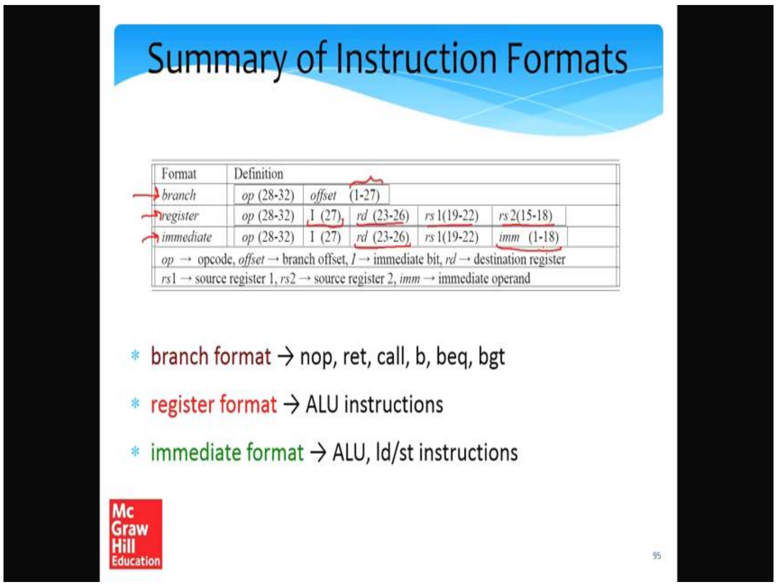
So, let us use the r d field to save one of the source registers. So, let us say we store r d; r d means it contains the register whose value is being stored in memory. We store this in memory the base address is in r s 1 and the offset is imm. So, if we do this trick, readers will appreciate that we are not defining a new format. What we are doing is that we are first using the first 5 bits to store the epcode, the immediate bit over here which is one more bit is one, which means an immediate exists. The register whose value needs to be stored is actually being stored in the r d field, which is fine it will introduce some complexities in the processor will take care of it.

The first source register r s 1; which is also the base register or memory is being stored over here and finally, the immediate which is the offset because we are using register indirect addressing mode that has a base and an index. So, in this case the immediate is being stored in the remaining 18 bits. So, this is a tight format, all 32 bits are being used. So, in that sense we are not you know wasting any bits, but the question is that if you think about it still an exception is being made, the reason being that the store instruction has a memory destination it does not have a register destination. So, we do not have any other instruction in our instruction set, which has a memory destination.

So, the store instruction in that sense is special, but we do not want to define a new format, because that will make the hardware the decoding hardware pretty complex. So, you would rather use the same format and use the fields in the format slightly differently.

So, that is the reason the register whose value needs to be stored is being saved in the destination register field with the exception that is being made and then the base register and the immediate are being stored at the respective places for the first source register and the offset.

(Refer Slide Time: 72:30)



Summary of Instruction Formats

Format	Definition				
branch	*op* (28-32)	*offset* (1-27)			
register	*op* (28-32)	I (27)	*rd* (23-26)	*rs* 1(19-22)	*rs* 2(15-18)
immediate	*op* (28-32)	I (27)	*rd* (23-26)	*rs* 1(19-22)	*imm* (1-18)
op → opcode, *offset* → branch offset, *I* → immediate bit, *rd* → destination register					
*rs*1 → source register 1, *rs*2 → source register 2, *imm* → immediate operand					

* branch format → nop, ret, call, b, beq, bgt
* register format → ALU instructions
* immediate format → ALU, ld/st instructions

Mc Graw Hill Education

95

So, what is the summary of instruction formats right now? The summary is something like this that the opcode is saved in the m s b bits are 28 to 32. So, in the branch format the summary is in bits 28 to 32, the opcode is saved; the offset is saved in the remaining 27 bits. In the register format same thing the top 5 bits always contain the opcode, then we have one bit to encode the fact that we have registered immediate or either a register operand or an immediate. So, in this case we first save the register destination, the first source operand and the second source operand which in this case are all registers.

In the immediate format we also have an immediate bit, which is set to one which means that the second operand is an immediate. We save the register destination the first source register and the second operand is an immediate it is an 18 bit immediate. So, the 18 bit immediate is divided into 2 parts. One part is the 16 bit number, which can be interpreted as sign twos complement number if required and the top 2 bits are the modifiers. So, we support 3 kinds of modifiers, one is the default where the 16 bits are interpreted as a twos complement sign number, then we have 2 modifiers u and h u is unsigned where the top

2 you know further immediate is interpreted as an unsigned number and h basically tells that the immediate will be left shifted by 16 positions.

So, this is where our des description of the simple RISC instruction set ends. So, mind you this was a very simple instruction set, this is very simple and simple RISC comes from. Real world instruction sets are much much more complicated, they have far more instruction modes and kinds of instructions, types of instructions, and they also take many different kinds of operands, there are many special conditions and flags and so on, but it is important for a reader to understand at least the nuances of designing an assembly language from scratch, what are the problems that can come; for example, in our case we had a problem of with the store instructions and we have made an exception for the store instruction. So, this is a typical you know engineering tradeoff that needs to be done that some case in sometimes for the sake of simplicity and elegance, it is important to make some tradeoffs such that the hardware remains simple.

Then we designed a full encoding format for our simple RISC instruction set, where we showed how using 32 bits all the instructions can be encoded into 0s and 1s. So, now, we will look at chapters 4 and 5 where we will discuss 2 real world instruction sets arm and x 86, and readers can use them to actually write actual assembly code in you know big software. But it is important to understand the concepts presented in this chapter, before going to those 2 chapters. So, let us end at this point and see you in the next lecture where we will be discussing the arm instruction set.