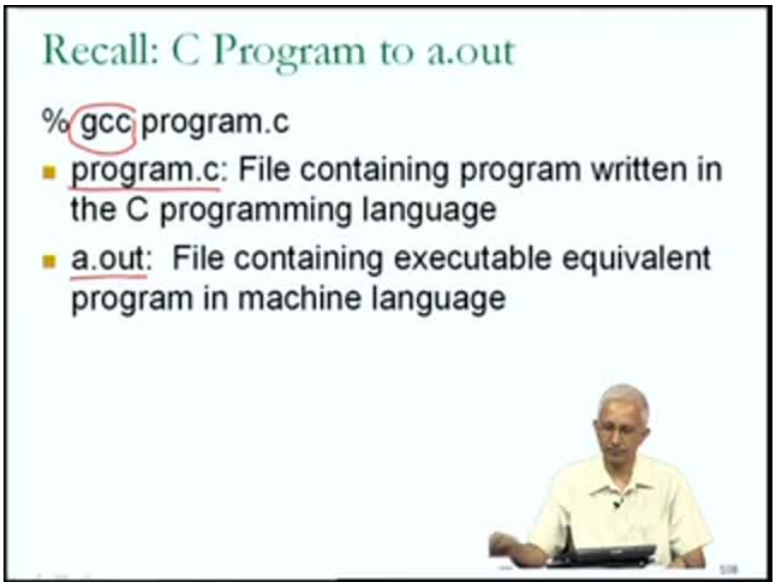
High Performance Computing

Prof. Matthew Jacob
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Module No. # 02
Lecture No. # 07

(Refer Slide Time: 00:44)



Recall: C Program to a.out

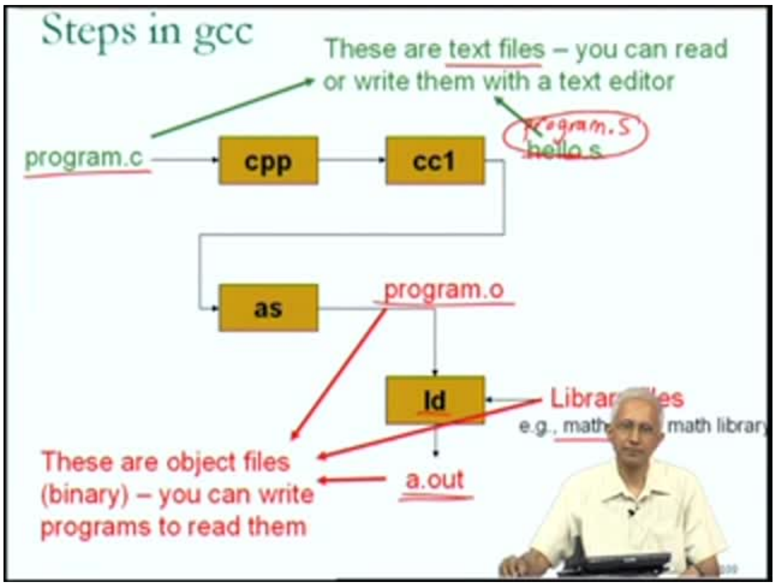
```
% gcc program.c
```

- program.c: File containing program written in the C programming language
- a.out: File containing executable equivalent program in machine language

Welcome to the 7th lecture in our course on high performance computing. In the previous few lectures we had learnt in some detail risk instruction set, which will use in examples in the rest of the course. And to start off one of the examples, I thought I would just remained you a little bit about something that we saw in the first or second lecture of the course, relating to how a c program gets converted into a form they can be executed on the hardware.

You will recall that this was done by a program translator, the example that we used was the program translator GNU C compiler. And the idea was that, you typed your c program into a file in this case, the file is called program dot c and what GCC does is, it takes your program as input and generates as output an equivalent program in the machine language in the default name of the equivalent program is a dot out.

(Refer Slide Time: 01:17)


Steps in gcc
These are text files – you can read or write them with a text editor
program.S
hello.s
program.c
cpp
cc1
as
program.o
ld
Library files
e.g., math
math library
a.out
These are object files (binary) – you can write programs to read them

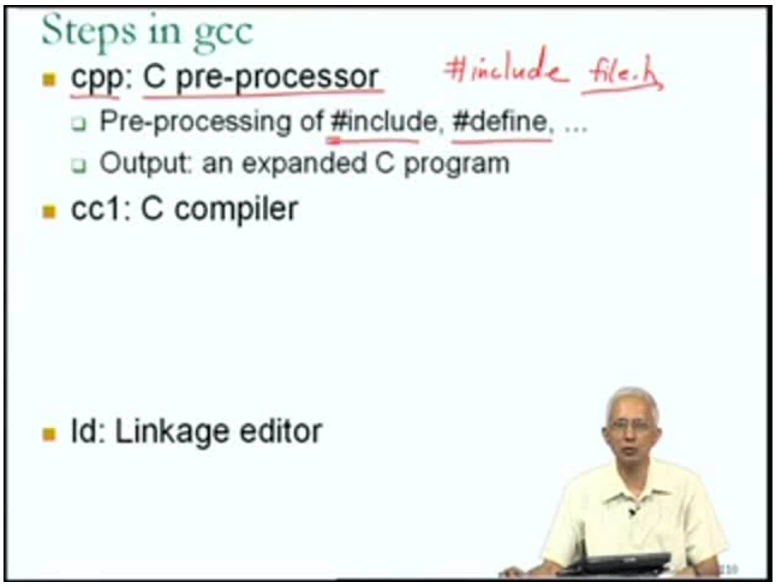
Let us just back up a little bit, you will recall that GCC does this; I had mentioned that there are number of steps in this translation process the c program, program dot c goes through cpp, cc1, as, and ld before a dot out is generated.

And we had seen what ld does. ld was the linkage editor, which merges or joins multiple files of the kind of program dot o or math dot o into a single file. So, it generates an executable file out of multiple object files, machine language files. Now, I had mentioned that between these steps of GCC, temporary files are used for the communication between these different stages of dcc. And this is going to be of interest to us, because we may be able to look at and learn something from some of those temporary files. And particular, I am going to talk about the temporary file, which is generated out of cc1. And this particular example, this should not read hello dot S, but program dot S since name of an input file is program dot c, here we would expect to see a file called program dot S.

Now, the think to note here is, I am talking about several different files, program dot c is initial file that you created, program dot S, program dot o are temporary files created during GCC compilation task, a dot out is the ultimate output file, the math library file exists on the computer system, it was generated and kept there. So, of these different kinds of files, it would help us to understand a little bit about what these files are like. So, I had pointed out last time that both, program dot c and program dot S, what are called text files. Basically, they contain characters that you can read, you could actually open either of these files using a text editor and you could read them or write them.

So, they are user friendly files and that you can read them, whereas program dot o, a dot out and the math library file, math dot o are what are called object files and they are in binary; they cannot be opened readily with a text editor and read or written. On the other hand, you could the write c program for the purpose of opening any one of these files and studying their structure. In fact, I had suggested that you could do this for the, a dot out file, that you could write a c program, which opens a dot out as input and studies its structure. I had mention that each of these object files has a well-defined format.

(Refer Slide Time: 04:11)



Steps in gcc

- cpp: C pre-processor
  - Pre-processing of #include, #define, ...
  - Output: an expanded C program
- cc1: C compiler
- ld: Linkage editor

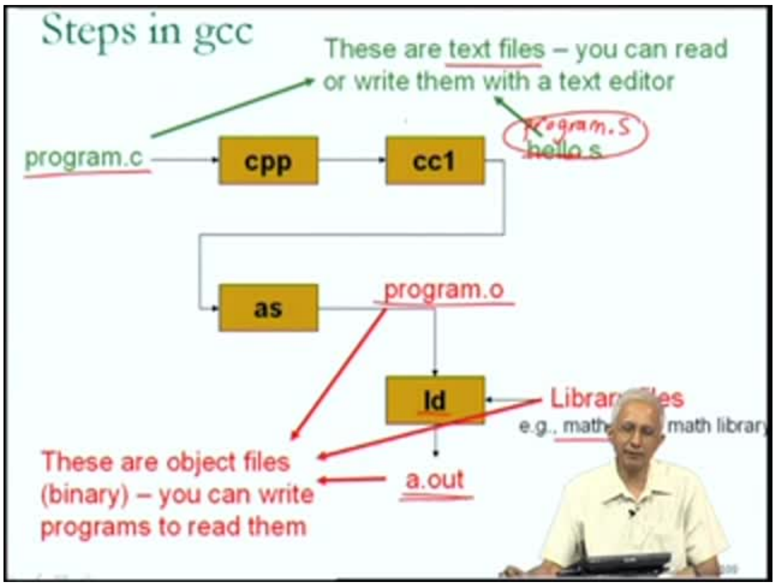
Now, the thing to note here is that, this program dot S file is a file, which you can actually read and it is a file, which is generated during the compilation process by GCC. Now, let me just say tell you a little bit more about the four steps in GCC. I have told you their names cpp, cc1, as and ld, and I have told you a little bit about ld, the linkage editor. So, I would not say anything more about that, but may be a little bit about each of the others. First of all the first stage in GCC, GCC does is something called cpp or the c pre-processor and without knowing it, you are all probably quite familiar with what cpp does.

Because all that cpp does is to pre-process to file by handling all of the lines in your c program file, which start with the hash sign basically, the count sign as you make call. So, the count sign include count sign defined these are really commands, which the c pre-processor is going to handle and as a result, the file which is outputted by cpp would not contain any of these files.

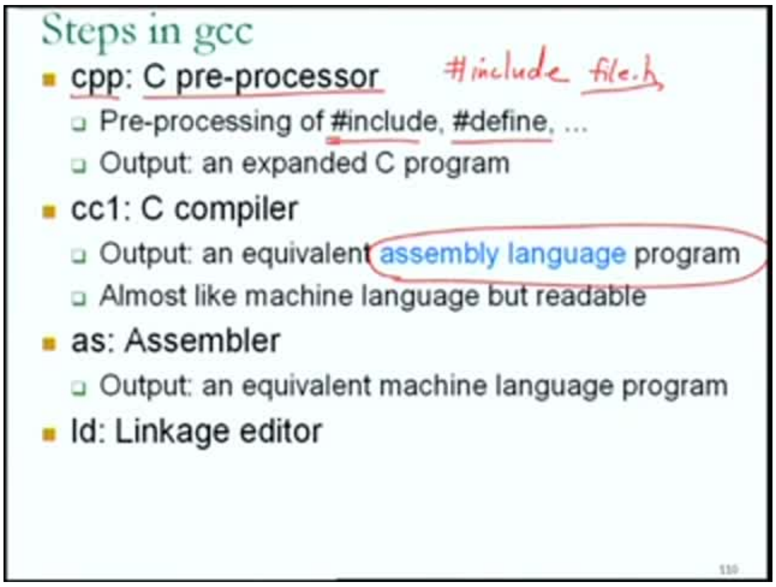
For example, that the count sign of the hash include directive, you could mention the name of a file as an argument of the coincidental and you know that the result of this is that the contents of the file, which you have mentioned will get included into your program.

And that is one of the kinds of things the cpp does. Count sign define, actually define something called a macro and the job of the cpp in connection with macros is a little bit different. But in general, what cpp does is to handle all of the directives of the kind, count sign include inside the file. And in assents what cpp generates its output is just an expanded inversion of your c program; it is still a c program, if you could look at it by opening it a text editor, you will find out it is in fact, an ASCII file a text file. You would find out that it all the lines of your c program are there, pretty much in the same form as what you had typed in.

(Refer Slide Time: 06:20)


Steps in gcc
These are text files – you can read or write them with a text editor
program.c
cpp
cc1
program.s
hello.s
as
program.o
ld
Library files
e.g., math library
a.out
These are object files (binary) – you can write programs to read them

(Refer Slide Time: 06:49)


Steps in gcc

- cpp: C pre-processor #include file.h
  - Pre-processing of #include, #define, ...
  - Output: an expanded C program
- cc1: C compiler
  - Output: an equivalent assembly language program
  - Almost like machine language but readable
- as: Assembler
  - Output: an equivalent machine language program
- ld: Linkage editor

Now this file, the file which is generated as output by cpp is input to cc1. And cc1 actually does some compilation, so it translates from c into something like the machine language. But as you would have noticed from the diagram, which we had over here what cc1 does is, it takes the output from cpp and generates a file. But the file that is generates its output is a readable file program dot S, which means that is not actually a machine language file. We would expect that a machine language file is going to contain machine instructions in binary and with therefore, it is unlikely that it would be readable. Therefore, the program dot S file in other words, the output generated by cc1 is not exactly a machine language file, but something else. What is it?

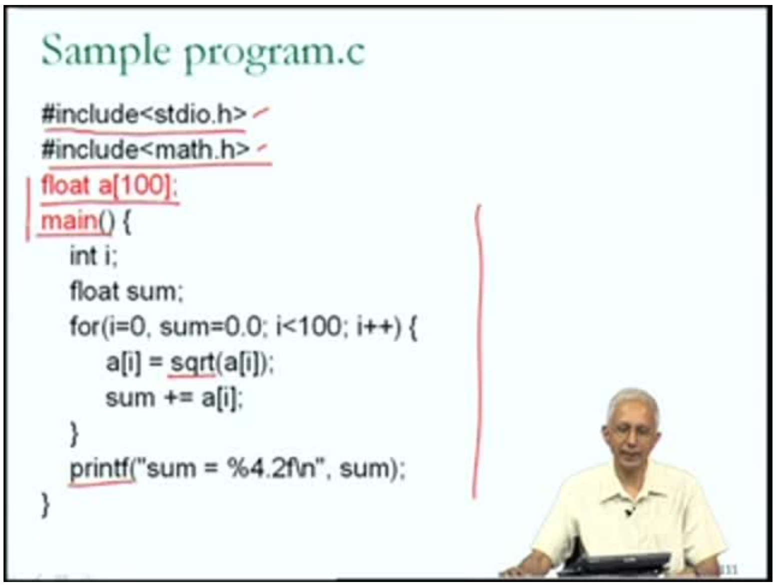
It is what you would call an assembly language file. It is an equivalent was So, cc1 from this diagram we would imagine it is generating an equivalent of your c program in something called the assembly language. And it is available in that file called program dot S, which you can actually open and study; therefore, this is of interest to us. So, what is this assembly language? Well, the assembly language is very similar to the machine language, only difference that it is readable. And in fact, we will if we when we look at assembly program, we will find the mnemonics or something is very similar to them that we had talked about, when we refer to the MIPS one in machine language or the MIPS one instruction set.

So, there may be certain things, which are not MIPS instructions inside an assembly language program and require a little bit of translation, but those would be almost trivial translations. And in addition, the assembly language program makes use of labels, rather than, making use of the actual addresses of operands. So, in some sense, the notation that we were using when we discussed the MIPS one instruction said, architecture was for the most part what you might call assembly language notation. Because we too are using the mnemonics; in other words, for the various operation of the instructions and we were using labels, rather than, actual addresses. So, what happens to the assembly language program, which is generated by cc1? Is still not in a form, they can be executed on the hardware, because it is not a machine language program.

So, it is the task of the next stage of the compiler, in other words, as or the assembler. To convert the assembly language program, program dot S in our example, into an equivalent object file, a machine language equivalent called program dot o and that is all that the assembler does.

So, we now, have a much better picture of what is in GCC and we understand that one of the critical points that we can access along the way is the assembly language program. If we could find the way to ask GCC to produce and let us see the assembly language program, then we could actually see very low language equivalent; our input c program, low level in the sense, that it is very similar to the machine language, but in the form that we should be able to understand.

(Refer Slide Time: 09:07)



Sample program.c

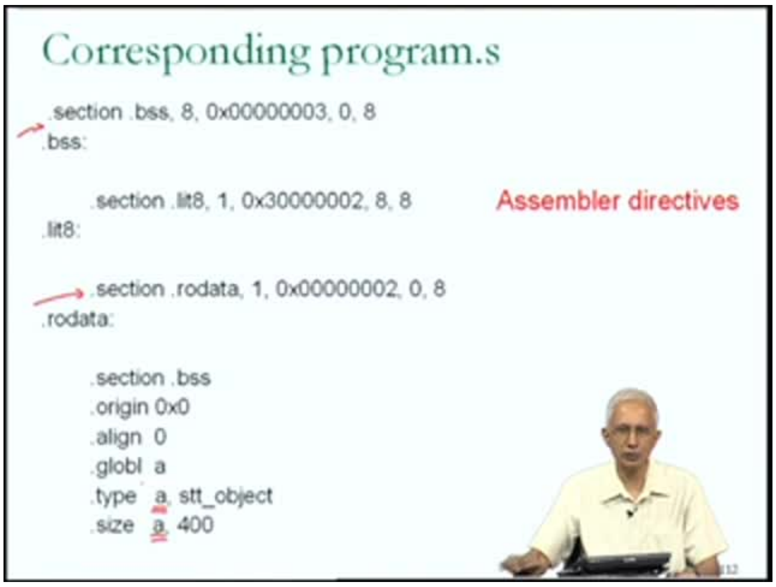
```
#include<stdio.h>
#include<math.h>
float a[100];
main() {
   int i;
   float sum;
   for(i=0, sum=0.0; i<100; i++) {
      a[i] = sqrt(a[i]);
      sum += a[i];
   }
   printf("sum = %4.2f\n", sum);
}
```

So, let us just look at an example of this. So, let’s suppose, support that my c program file was a simple program of this kind, you notice that they are some I am including the standard IO library and including the math library, the file for the standard IO and math functions. And this particular case, this program is using one of the math functions and it is also using something, which is going to come from the standard IO library, which is why these two include lines have to be there. So, it is very simple, little program; I will just draw your attention to the fact that this program declares and uses a floating point array a, of size 100 and then, other than the function may name, it does not have any of the functions. So, this is just the main program, the main function which executes.

So, if this was the input program to which I pass to GCC and I found a way to ask GCC to generate program dot S, rather than, producing program dot o, the object file equivalent or a dot out, the executable equivalent, what would I see, that is the question we will ask next.

So, as I indicated, you would expect that we would see an assembly language equivalent of our c program. In other words, pseudo mnemonics and labels and various other things which is what the compiler had basically translated a c program into. So, let me just show you parts of the program dot S. I am not going to show you the whole thing and we would not try to understand the compilation process or all the lines inside the program dot S file the assembly language file, but I will just highlight some of the interesting things to observe. So, once again, let me just remained you to note this, our input c program declares and uses the floating point array a, of size a 100 and it has a function call main.

(Refer Slide Time: 10:50)



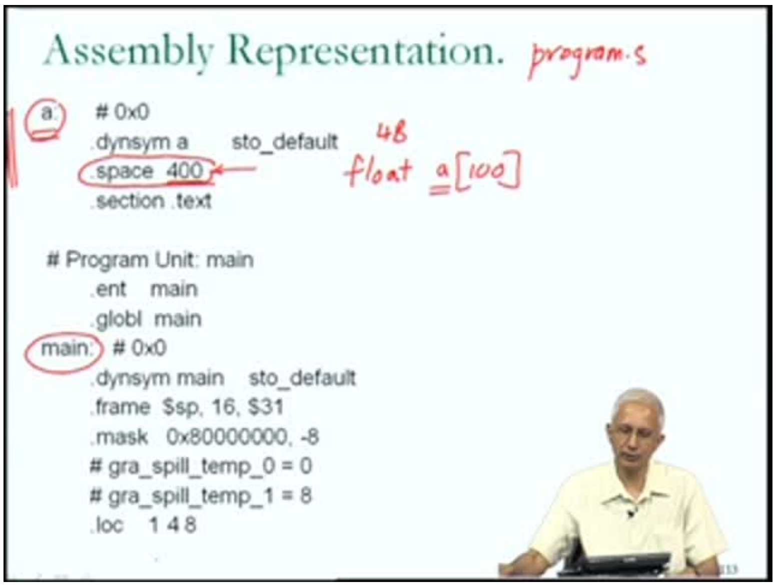
Corresponding program.s

```
.section .bss, 8, 0x00000003, 0, 8
.bss:

      .section .lit8, 1, 0x30000002, 8, 8
.lit8:

      .section .rodata, 1, 0x00000002, 0, 8
.rodata:

      .section .bss
      .origin 0x0
      .align  0
      .globl  a
      .type   a, stt_object
      .size   a, 400
```

Assembler directives

So, when we look into the program dot S, it is actually fairly large file, I was unable to fit it on to one slide in fact, it does not even fit on three or four slides; so, I am just going to show the early parts in the program dot S, actually not going to get into the interesting part very much.

(Refer Slide Time: 11:46)



Assembly Representation. program.s

```
a:    # 0x0
      .dynsym a      sto_default
      .space  400
      .section .text

# Program Unit: main
      .ent   main
      .globl  main
main:  # 0x0
      .dynsym main   sto_default
      .frame  $sp, 16, $31
      .mask   0x80000000, -8
      # gra_spill_temp_0 = 0
      # gra_spill_temp_1 = 8
      .loc    1 4 8
```

48
float a[100]

So, you will notice that the first part of program dot S, there are number of lines all of its start with dot and none of which looks in anyway related to our program. I noticed, I identified a over here, which must be something to do with my array a, but other than that, there is not much of use over here. Now, all of the lines that we see on the screen here are actually, what are called assembler directives; they are lines, which has been created by cc1 for the purpose of asking the assembler to do something. In sense, we are not particularly interested in learning about how an assembler works, I am just going to gloss over this and move on to the next foil, the next slide.

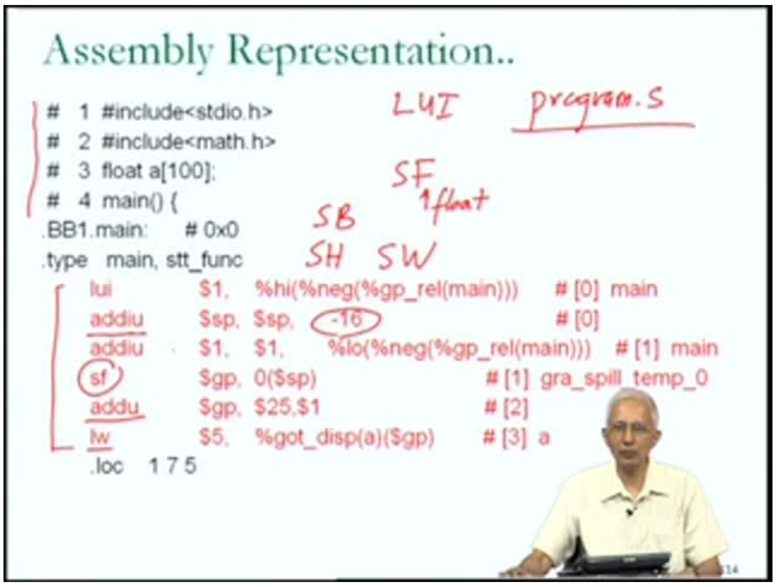
If we look at the second slide, this is the continuation of the program dot S, contains contents of program dot S. And here, I notice that there is declaration a, there is a with colon after, we suggest that this has something to do with assigning an address to a and the interesting thing that I note here is that there was a directive called dot space 400. Once again, it is starts with the dot; so it is in assembly directive. But it is from the name, you would guess that this is a directive, which is requesting some space to be allocated.

And this case, in the number of the number of bytes of space being requested is 400, which seems to be easy for us to understand. Given that the, a, that we are referring to here must be the array a, that we had declared in our program; it was an array a of size 100 floats. Now, we know the size of each float is 4 bytes and therefore, we know that the size was this entire array a, is this 400 bytes of space, which is being allocated.

Now, it looks from this particular set of lines inside the assembly language, that a, is being a sign space not on a stack, because if it was being a sign space on a stack, we would have of seen some indication of that and not dynamically. Because this is a static allocation; this is in fact, an example of static allocation, the array a, is being statically allocated 400 bytes.

So, I go further down and I noticed that, there is another label and this is the beginning of a label for the beginning of my main function followed by a number of lines all of which are assembler directive ; so, I want once again, I will gloss over those and move on to the next slide.

(Refer Slide Time: 13:33)



Assembly Representation..

```
#   1 #include<stdio.h>
#   2 #include<math.h>
#   3 float a[100];
#   4 main() {
.BB1.main:      # 0x0
.type   main, stt_func
        lui       $1,   %hi(%neg(%gp_rel(main)))     # [0]  main
        addiu     $sp,  $sp,  -16                    # [0]
        addiu     $1,   $1,   %lo(%neg(%gp_rel(main)))  # [1] main
        sf        $gp,  0($sp)                  # [1]  gra_spill_temp_0
        addu      $gp,  $25,$1                  # [2]
        lw        $5,   %got_disp(a)($gp)       # [3]  a
        .loc    1 7 5
```

Now, in the next slide I notice that number of lines, which seem to be copied from my program, but what follows is of interest. Because the lower half of this slide, the part of the lines, which are now gradually turning into red; for the most part, something's that we can understand, we have seen some of these before. For example, we see that this is the MIPS one LuI instruction, which we all know is the lower upper immediate. I had mentioned that this is an important instruction, because it can be used to load an address into a register. And that seems to be, what is being done by the instructions that follow, but again, for the moment we will just forget about that.

After this there is an add immediate unsigned and notice that there is a 16 bit immediate operand, which happens to be the value minus 16. There is an add unsigned, there is a load word these are all things that we understand. There is one instruction, one mnemonic among the among the instructions over here, that we have not seen before and that is this SF instruction.

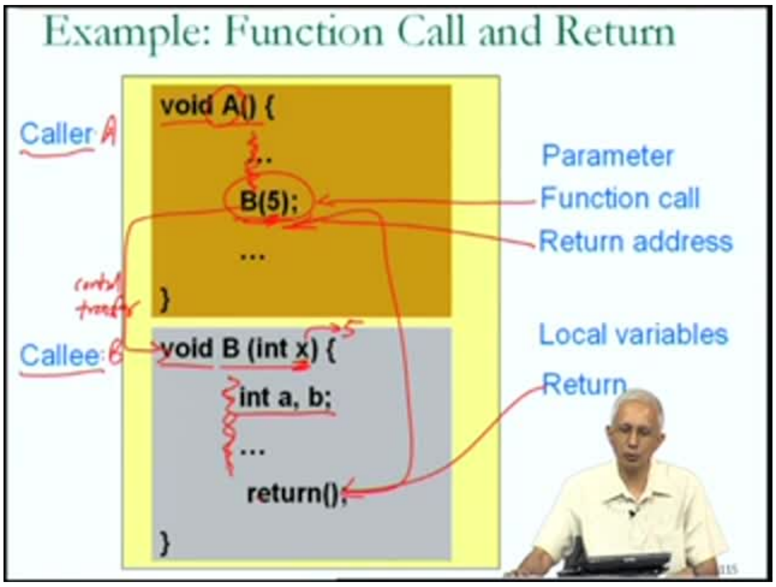
Now, the instructions that we saw which start with an S, where SB, SH and SW, and these were the store instructions. SB was the store byte instruction, which could be used to store 8 bits from a register into memory. SA it was the store half instruction, which could be used to store 16 bits, the least significant 16 bits of the operand register into memory. And SW was the store word instruction; it could be used to store 32 bits from register into memory.

We have something call SF, this is in fact a store instruction to be used in connection with floating point values as I had mentioned, but our discussion of the MIPS instruction set covered only the integer instructions. But the MIPS instruction set thus have capabilities for processing floating point values and there are more instructions, which I had not mentioned and this is an example of one of them. This is an instruction, which can be used to store a floating point value from register into memory. Similarly, there would be instructions, which can be used to add floats, multiply floats, etcetera family of floating point instructions.

But this is what we now understand, we would see if you looked at a file like program dot S. In other words, assembly language files, one of the temporary files that is generated by GCC in the process of compiling a c program into an executable file. And in the assembly language program, we see that there are instructions which we have heard about and conceivably we could think of modifying the assembly language program. For example, if I want to add or change some of these instructions, I could edit the assembly language program and experiment with that and then, pass it back to GCC for continued processing by the remaining steps of GCC. So that is something, which we may look into later.

So, we now understand that while the machine language that we saw is meant for the machine to execute; there is slightly more human a readable version of machine language called the assembly language and that is the level at which we are going to use examples. And that could also be the level at which one could modify once program if one was intend on doing so at the very low level. So, much for GCC we will now move back to trying to understand the MIPS one instruction set by trying to write some simple examples.

(Refer Slide Time: 16:59)


Example: Function Call and Return

```
void A() {
    ...
    B(5);
    ...
}
void B (int x) {
    int a, b;
    ...
    return();
}
```

Now, the first well predominant example, which I am going to use; this is a dual purpose example, we are actually going to try to understand the various sequence of operation that has to happen during a function call and return. This is going to be useful to us; we have a general idea about function, how function calls and returns operate. But we are look in detail at various things, which must happen as a part of function call and as a part of function return and we will do so in the MIPS one assembly language. So, therefore, it is a dual purpose example.

Now, if you look at the example that I have here, the outer yellow box is some program and I am concentrating on only two functions of the program, the function A and the function B. Just notice that is not at the both of these functions do not return a value, which is why a void function, function A does not take parameters, function B takes 1 integer parameter. Now, in this particular situation, if you look at function A, you will notice that function A calls function B with 1 integer parameter and inside function B there is a return.

So, we know that the way that the function call operates is when there is execution in function A, when one reaches the function call, rather than, just going after in executing the function call, rather than, just going on to the next instruction in A control is transfer to function B.

So, we were going down in function A executing the instructions of A one after the other until, we came to the function call at which point control gets transferred. So, this was the control transfer associated with a function call.

After this, the instructions of B are executed, until they return at which point, control gets transferred back to just after the function call inside A. So, this is what we know about function calls and let me just first of all introduce some terminology. Now, we have two functions over here; there is the function in which the function call is present and there is the function, which is being called. Since there are two functions, we need some terminology to distinguish between them. And the point in function A, where function B is called is what we call, the function call that is fairly clear. The function in which the function call is made is call the caller function.

So, in this particular example, the caller is A and the function, which is being called I will refer to as the callee. So, in this particular example, the callee is B, it is a fairly standard terms caller and callee. So, the function call is present inside the body of the caller and as a result of function call, control is transferred from the caller to the callee.

Now, often in a function call, as in this example, they may be they need to pass parameters. So, in this particular example, the value 5 is a value that has to be passed by A to B; inside the function B, the parameters is known by the name x.

So, this particular situation, the value, the variable x inside the function B, would be associated with a value 5 as a result of this function call, that is, how a parameter is passed from function A or from the call caller to function B or the callee.

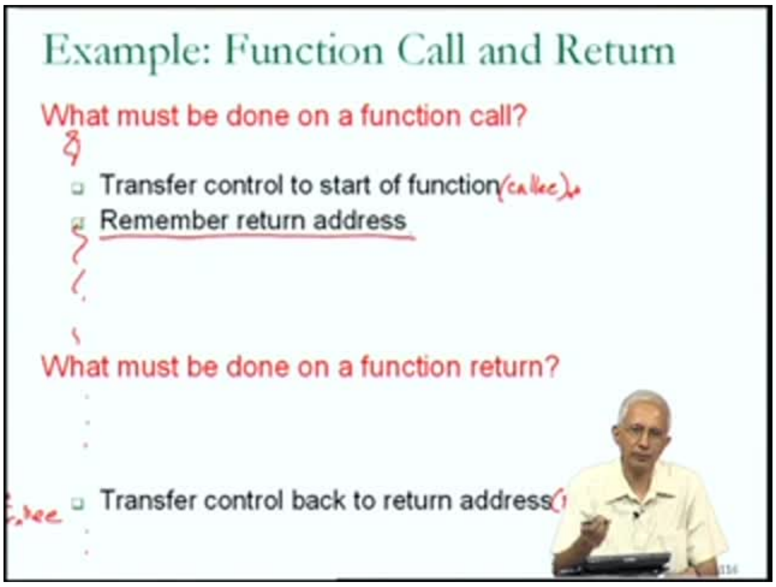
So, this is motion of passing parameters, then we also know that in functions for example, here function B has two local variables. So, the local variables of function B in this particular example are called A and B not very good names, but in this example that is what we have.

So, in B the body of the function gets executed and then there is the return, which is the point doing execution, when the control is supposed to be transfered back to the caller. And the question of where should control be transfer back in the caller is that, control is transfers back to the point immediately after the function call, which is what we will refer to as the return address.

Now, in your experience with programming you know that there are many different mechanisms for passing parameters. This particular kind of parameter passing that we have here is called parameter passing by value. Since the value 5 gets passed and is locally within the function known by the name x, value of x could change that does not cause any change to the value 5 obviously, because 5 is a constant. And there are other mechanisms for parameters passing, which would be a little different from what we will be talking about in this example, but you will be able to understand how to work out those details on your own, I think. So, we will just try to understand how this particular function call would work.

So, we need to understand, what are the various low level operation, which must happen in order to allow for what we just saw, this functionality to happen, when the program containing these two functions is executive.

(Refer Slide Time: 22:11)



Example: Function Call and Return

What must be done on a function call?

- Transfer control to start of function(callee).
- Remember return address

What must be done on a function return?

- Transfer control back to return address

So, we will break this up into the various things that must be done as a part of the call and the various things that must be done as a part of the return. And we are going try to work this out that at a very low level, because we ultimately want to write MIPS instructions corresponding to each of the steps that we are going to put down here.

Now, what is really obvious is that, when the function call occurs, there is a need to transfer control to the start of the callee function. In other words, the function call happened in A, so there is need to transfer control to the start of function B or the callee. So that is one of things that must happen as a part two function call. Similarly, as part of function return it’s fairly clear, one of the things that must happen is the control must be transfer control back to the return address inside the caller.

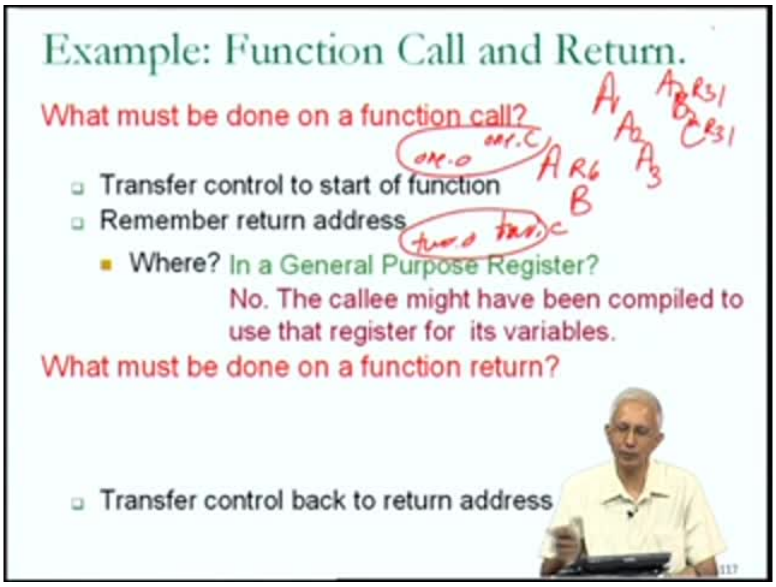
So, just you make sure, you understand what I am doing here; there are many steps which must be executed. So, I am going to list all of them one after the other, there are many different other steps are going to pop here, one after the other as we understand what this steps are. And one of the early steps in doing the function call is to the need to transfer control to the start of the function, there may be one or two steps before that which is why we left a gap over here. Obviously, several steps after that which is why, I have left gaps over here.

So, now, we understand the part of the function call, there is a need to transfer control to the start of the callee, and as a part of function return, there is a need to transfer control back to the return address inside the caller.

Now, one issue, which this raises, is the following as part of function return there is a need to transfer control back to the return address inside the caller. Now, this activity in other words, the transfer of control is going to be an instruction inside the callee, inside the function which was called. Remember that the function return is inside the callee and therefore, that the instruction which does transfer of control back to the return address is going to be inside the callee.

Now, inside the callee unless, I explicitly had remembered what the return address was they would be no way for this transfer of control back to the return address to happen. So, unless at the time of the call we had remembered, what the return address was, then at the time of the return it would not be possible to identify the return address. Therefore, going back to the point of call, we now realize that one of the important things that must happen of the point of call is that, before we execute too much inside the callee, we have to remember the return address within the caller that is important.

(Refer Slide Time: 25:37)


Example: Function Call and Return.

What must be done on a function call?

- Transfer control to start of function
- Remember return address
  - Where? In a General Purpose Register?
    No. The callee might have been compiled to use that register for its variables.

What must be done on a function return?

- Transfer control back to return address

So, to remember the return address we have to use memory. So, the question which we need to first figure out is, how and where do we remember the return address? We are going to concentrate on this step for few minutes. How and where and do I remember the return address? Now, they are many possibilities; one possibility is that I could remember the return address in a general purpose register. And in fact, those of you who remember the jump and link instruction, we will realize that the jump and link instruction remembers the return address for you inside register R 31. We will come to that shortly. Now, the problem with thinking of remembering the return address in a general purpose register is that, this may not actually be a safe solution. Because there is no guaranty that the general purpose register at which I saves the return address we will survive with the value that I have saved there for long.

Let me just give you a single example, let suppose that I have a situation, where there is a function called A, which is calling itself in other words, the recursive function call. So, A calls itself and then, A might call itself again, recursive function is a function which calls itself.

So, the situation here is that function A calls itself and then inside the So, there is the first call to function A, there is the second call to function A, there is the third call to function A one after the other. Now, inside after the first call to function A, there is a need to remember the return address of that call and I might save it in a general purpose register. But after the second call to function A, there is a need to remember the second return address and if I have a single piece of code for the function, then it is going to use the same return address that I had used for the first saving, in other words, the same general purpose register will be used for that purpose.

You could also see how this problem may arise if there are other sequences of, for example, if there is a nested function, call function A, call function B, call function C, then I if I am using there is R 31 to remember all return addresses. Then in transferring control from A to B ,the return address inside A will be present in R 31, but then inside function A when I call function C, register R 31 will be to used to save the return address inside B and I will therefore, loose return address inside A. Therefore, in general, is just thinking that I can save written addresses inside a general purpose register is not going to work, we need something better.

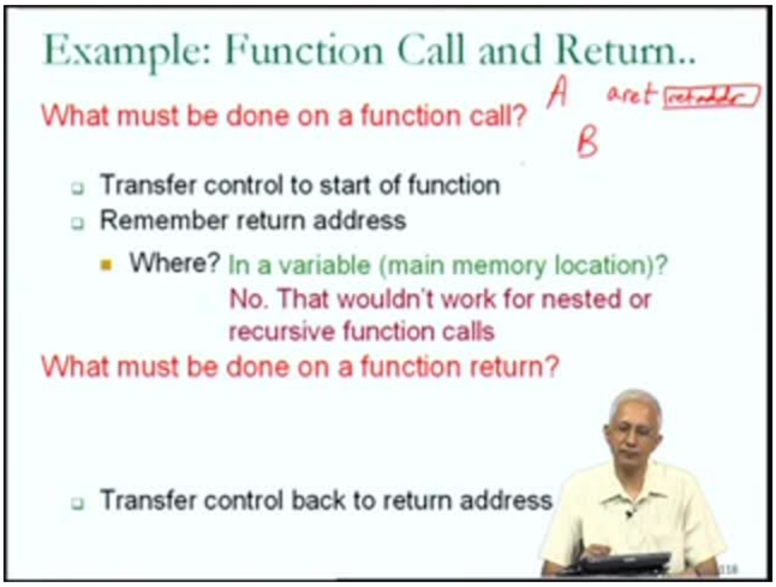
So, this may not work, because well this is another reason that this may not be a good idea; what I have return over here is this may not be a good idea, because the callee might have been compiled use that register for its variables. Let me just explain about, this means now, that the idea here is that I am suggesting that when function A calls function B as a part of the function call. If in remembering the return address I might remember the return address in register R 6, in other words R 6 will contain the return the address inside function A.

Now, you will remember that function B is itself piece of code, which does execution of various instructions. And it is conceivable that functions A and B were compiled separately, in other words I may have written function A in one file and compiled it. Function A may have been present in a file called 1 dot c, function B may have been present in a file called 2 dot c and I may have compiled 1 dot c and got a file called 1 dot o. And I may have later, complied 2 dot c and got a file got 2 dot o and then, ultimately I may have linked the files together.

That is what I mean by saying that A and B may have been compiled separately. Now, this is the case, then when I complied 1 dot c into 1 dot o, the complier may have used many registers for many things, for example the compiler in this example may be using R 6 as a return address. Later on, when I compiled 2 dot c to 2 dot o, the compiler once again is using registers for various things and there is no telling, what it might have used

R 6 for. So, this is another reason that just using a general purpose register to remember a return address might not be a good idea.

(Refer Slide Time: 29:45)



Example: Function Call and Return..

What must be done on a function call?

A aret $\boxed{\text{retaddr}}$

B

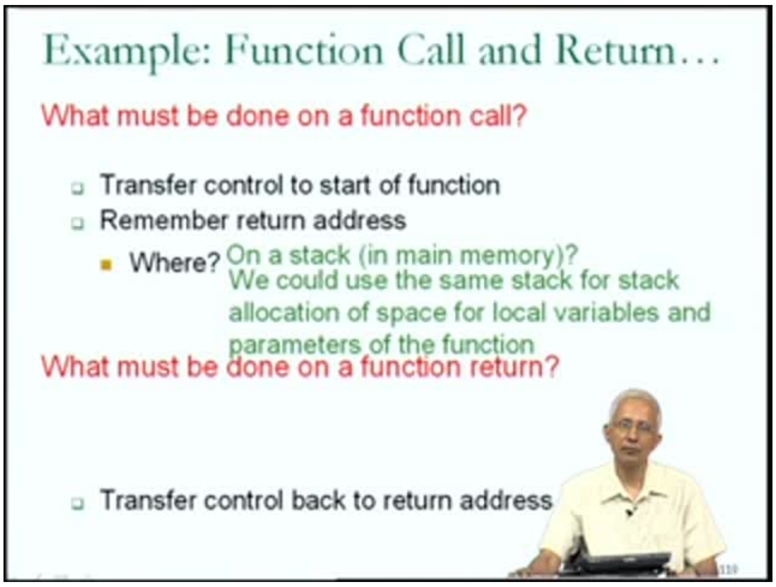
- Transfer control to start of function
- Remember return address
  - Where? In a variable (main memory location)? No. That wouldn't work for nested or recursive function calls

What must be done on a function return?

- Transfer control back to return address

So, we have to look for alternatives; just using general purpose register is not a sound solution. Now, in another possibility then I could remember the return address in a variable, in other words in a main memory location. So, the simple idea here is that when function A calls function B, then associated with the return address of function A, I could have a memory location, which I can refer to as A return.

(Refer Slide Time: 30:38)



Example: Function Call and Return…

What must be done on a function call?

- Transfer control to start of function
- Remember return address
  - Where? On a stack (in main memory)? We could use the same stack for stack allocation of space for local variables and parameters of the function

What must be done on a function return?

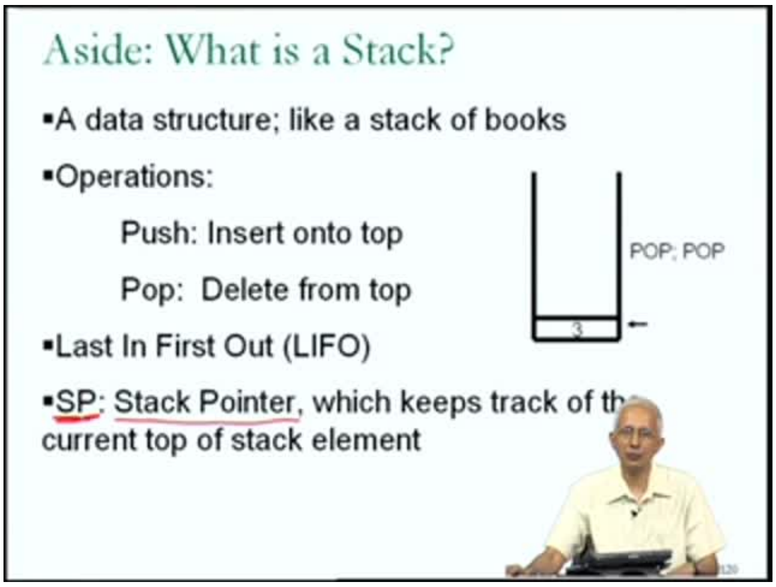
- Transfer control back to return address

And when the function call happens I will remember the return address of function A inside that variable or that memory location and this would not have the problem, which separate compilation that I have talked about. But it will have a problem with the case of nested or recursive function calls that we had talked about prior to that. Therefore, just using a memory location to remember the return address may not be again an adequate solution for all situations; therefore, we need something different. The next possibility that we will talk about is since I am concerned about losing a return address, if I use one register or one memory location, why do not I just try to remember return addresses on a stack? In other words, in main memory on a stack, the advantage of using a stack is I am not using one memory location; I am using any number of memory locations. If I have to remember 6 return addresses, I will push them on to the stack one after the other. Therefore, this will not have a problem with recursive function calls or nested function calls and is in fact, a resemble solution for current situation.

So, we are going to assume that this is what we will do. We will assume that when there is a function call, as part of the transfer of control to the caller, I am sorry to the callee, the return address must be remembered; the return address inside the caller and that will be done on a stack in main memory. Now, we have already seen that stack allocated variables such as the local variables and parameters of a function are going to be allocated space on a stack.

So, we could just use the very same stack for remembering the return address. So, the stack that we talked about earlier, where stack allocation of space for variables is done is going to be used not only allocate space for local variables of a function and for parameters of a function, but also for remembering the return addresses of calls to functions.

(Refer Slide Time: 32:19)



Aside: What is a Stack?

- A data structure; like a stack of books
- Operations:
  - Push: Insert onto top
  - Pop: Delete from top
- Last In First Out (LIFO)
- SP: Stack Pointer, which keeps track of the current top of stack element

So, this is going to be a single stack, which will serve all these purposes. With this we have to tackle the problem of transferring control to the function and back from the function to the caller, because we now know where to save the return address. Now, just to make sure, we understand what is stack is. Let me quickly recap what is a stack is.

So, when we talked about stack earlier, I had mentioned stack is a data structure and when thinking about a stack, you think of a stack of books in fact that is where name comes from. On this table, if I were to stack books I would put one book and then, if I want to add another book to this stack I would add it on top of the first book and if I want to add a third book on this stack I added on top of the second book. Then if I want through remove a book from this stack I would remove from the top, because if I try to remove lower books I could either damage the books or hurt myself.

So, the general property of the stack is that it is the data structure, which has operations of putting something new on top of the stack or removing the top most element form the stack and these operations are usually called push for the insertion operation and pop for the deletion operation.

So, both push and pop deal with the top of the stack; therefore, an implementation as I have mentioned last time one things of the stack is being a last in first out data structure. Because it is a last thing that you put into the stack, in other words book which is on the top of the stack, which is the next thing that you will get when you pop from the stack. So, this called a last in first out data structure.

Now, very clearly the since the focus of interest of the stack is the top of the stack, in implementing a stack we will need to remember the memory location, the stack is going to be implemented in memory. As we said is going to grow depending on as more and more elements get pushed on to this stage it will grow in memory, but in any given point in time we will need keep track of which element is the current top of stack. And we will think of doing that using of variable or pointer called the stack pointer. So, the stack pointer is a variable, which always contains the address of possibly the current top stack of the top element of the stack. And stack pointer is typically abbreviated as SP.

Now, moving on, we will try to understand how we could implement the stack using the MIPS instructions. Since we clearly going to need a stack to do the function call, we will objective with this point is still to understand the function call and return mechanism completely. But there is the sub problem of being able to implement a stack, because the stack is important for the implementation of the function call. Because that is where the stack is where the local variables and the parameters are going to be stack allocated and the stack is also where the return address is going to be remembered.

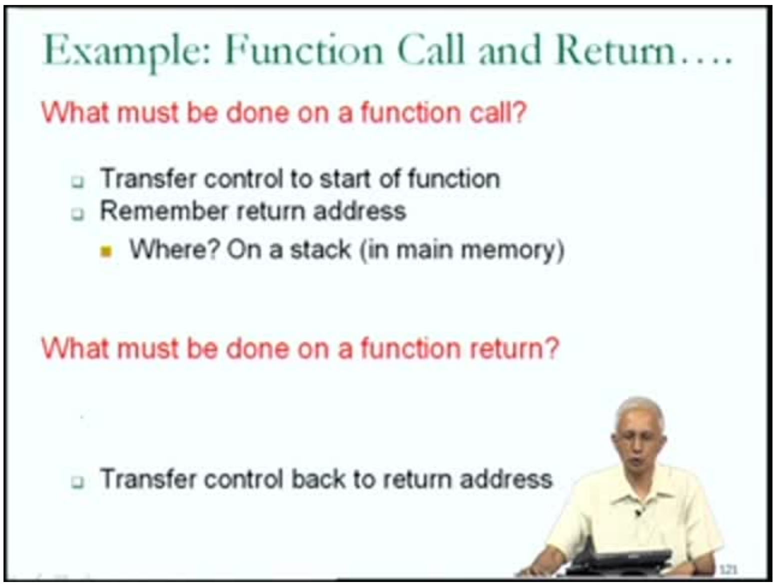
Since we are going to need a stack certainly for the implementation of function call and return, this aside will help us to understand how a stack could be implemented in the MIPS one language. Now, to make sure we understand the push and pop operations. Let suppose that I have a stack and currently the situation on the stack is that there is a value 3 on the bottom and value minus 4 on the top, they happen to be there from previous operations on the stack. Therefore, the current stack, top of stack pointer will point at the minus 4.

If there is stack pointer currently points at minus 4 element. At this point, if I were to push a new value on to the stack how would this change this stack? The answer is the 47 would be an additional value on top of the stack and the stack pointer, in status instead of pointing at minus 4 would now point at the 47.

So, this is what the stack would look like after a push 47. What will happen if I then did two pop operations? Remember each of the pop operations is going to remove the top most element from the stack. Therefore, If I do two pop operations: the first one would remove the 47 from this stack and the second one would remove the minus 4 from the stack leaving only the 3 on the stack and the stack pointer would be pointing at the 3.

So, this is what the stack would look like after two pops. So, with this we understand that to implement a stack I must allocate space in memory; I must have space in memory for the stack to grow. I must have something which will point at the top most element in the stack and whenever there is a need to push on to the stack, I have to store new values into memory and updates the stack pointer; whenever there is a need to pop something from the stack, I need to change the stack pointer. So, very clearly the operations are quite simple, we just have to see how to implement them in the MIPS language.

(Refer Slide Time: 36:42)



Example: Function Call and Return….

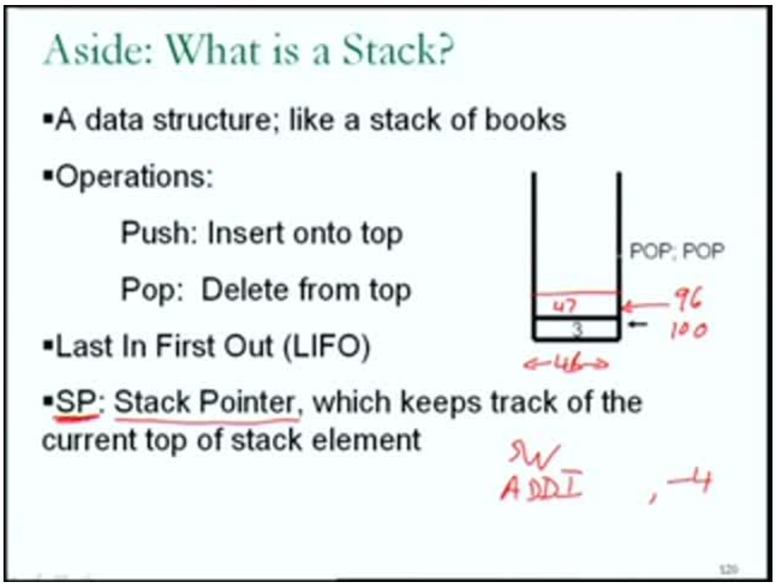
What must be done on a function call?

- Transfer control to start of function
- Remember return address
  - Where? On a stack (in main memory)

What must be done on a function return?

- Transfer control back to return address

(Refer Slide Time: 36:56)



Aside: What is a Stack?

- A data structure; like a stack of books
- Operations:
  - Push: Insert onto top
  - Pop: Delete from top
- Last In First Out (LIFO)
- SP: Stack Pointer, which keeps track of the current top of stack element

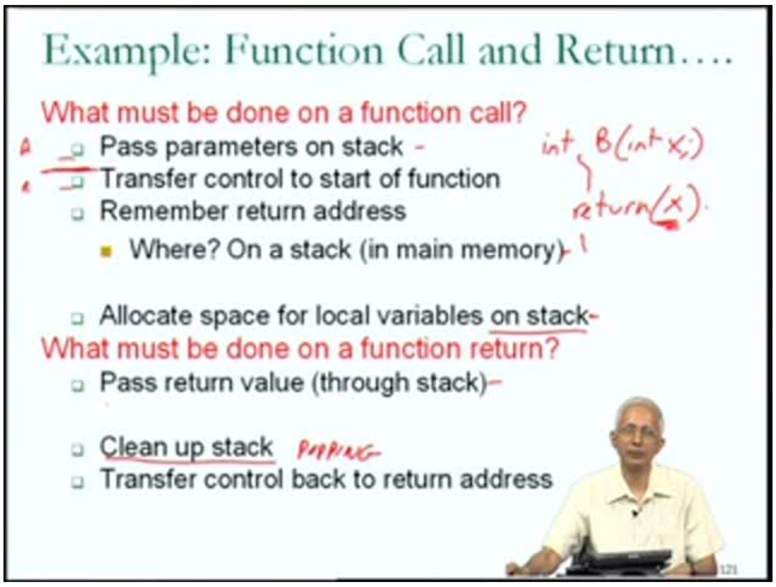
POP; POP
47
3
96
100
4b
SW
ADDI , -4

We will possibly come back to that shortly, but for the moment it is clear that the instructions which will be used to do these operations are going to be simple. In order to push something on to the stack, if the current stack pointer is pointing here, if I wanted to push 47 on to the stack I would have to use a store instruction, because to write anything into memory I need to use a store instruction. And in order to modify the stack pointer to point over here, I would have to use some kind of an add or subtract instruction depending on what the nature of the stack pointer is.

In this particular example, it looks like the stack is going into a lower memory address, therefore, I would have to subtract from the value of the stack pointer. If the address of this memory location was 100, the address of this memory location would have been 96 if the size of each stack element is 4 bytes, which is why I said that in order to put the 47 into this location I would have to use a store word, because the stack is in memory. And there are if I were to update the stack pointer, I may have to use the subtract instruction or an add immediate instruction with an operand of minus 4 by which I could subtract 4 and get the 100 to pointer 96.

So, the instructions dealing with implementation of the stack are going to be may be a pair of instructions for a push and a pair off instructions for a pop. And for the push, it is going to be a store and an add immediate, and for the pop is going to be a load, in other words copy a value from memory into a register, since I want the pop's value to be available for some purpose.

(Refer Slide Time: 38:31)


Example: Function Call and Return….

What must be done on a function call?

- Pass parameters on stack
- Transfer control to start of function
- Remember return address
  - Where? On a stack (in main memory)
- Allocate space for local variables on stack

What must be done on a function return?

- Pass return value (through stack)
- Clean up stack
- Transfer control back to return address

int B(int x;)
return(x).
popping

So, the stack seems to be fairly easy to understand. Just go back to the function call and return, this where we are. We understood that as part of function call, I have to transfer control and remember return address on the stack. As part of function return, I have to transfer control to that return address, what else has to be done? Now, we had seen that the function A in calling the function B passed a parameter and the parameter in that particular example was the value 5. But in general, if parameters have to be passed, then there must be a mechanism by which the parameter, which was known to the caller becomes known to the callee. And once again, one way to do this actually to pass the parameters on the stack there are other ways to do it, but let us just assume that this is how we are going to implement the function call.

So, how will we pass the value 5 from the caller to the callee? We will do so by allocating space for the parameter on the stack and putting a 5 into that particular location on the stack as part of the function call, which is why I put pass parameters on the stack as something that must happen before I transfer control to the start of the function. Thus once I transfer control to the start of the function I mean function B.

So, this is the point at which I move from function A to function B and this point I mean the caller, at this point I mean the callee. And I know the value of the parameter only when I mean, the caller. Therefore, the passing of the parameter must be done inside function A and that is why it is the first thing that must be done in the sequence of steps. What are the other things that must be done?

Now, as we know, before I actually start executing the body of the function, which is being called its local variables must have space allocated for them. Therefore, the last thing that I talk about doing as part of the function call is going to be to allocate space for the local variables of the function, which is being called for the function B. And as we had already agreed, that is going to be done on the same stack.

Now, unfortunately I still many steps that seem to have to be filled up. So, where we seem to be about two-thirds of our way done with the function call and return, but it is not clear what remains to be done. Until we remember that unlike, I have particular example where both function A and function B are void functions and do not have they need to pass return values. In general, functions may have to pass return value, in other words the function which is being called the callee may have to pass a value back to the function which called it. And that would have been the example in our case too, if the function B had been declared as the integer function. Because if function B had been an integer function, then the situation may have been something like this, the return instruction which would have mentioned the value that has to be pass back to the function A.

So, in general, we have to have we have to be aware of, how one can pass a return value from the function, which was called back to the function which called it. And at this point it should be fairly clear; the one way to pass the return value would be to for the function, which is being called to callee to push the return value on to the stack.

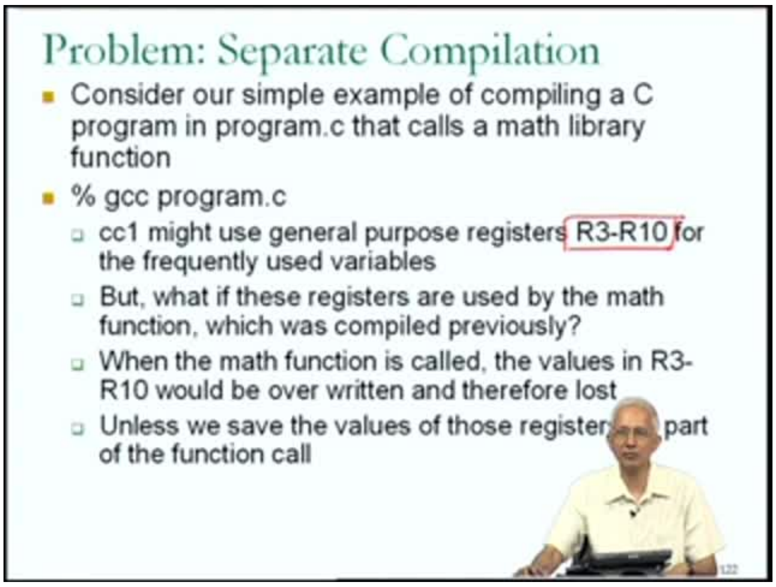
So that after controllers transfer back to the caller, the caller can just read the return value of the stack. So, return values too can be communicated through the stack. Now, we have nearing the end of or sequence of steps, because there is not whole or a white space left, but they seems to be some critical functionality that has been left out. And does it happens this relates to the separate compilation issue, which we had talked about, but I will come back to that in a second.

Now, in this sequence of steps that we have, you noticed that lots of things were getting pushed on to this stack. At this point, we have not talked about removing anything from this stack and therefore, passing parameters on this stack is pushing, remember return address on this stack is pushing, passing return value is pushing; so all of local variables on this stack is pushing space on to the stack.

Therefore, as a result of any point function call, a lot of space is going to get allocated on the stack and unless, we explicitly include the reclamation of the space inside our series or steps, for what has to happen during a function call and return, this stack will keep on growing in definitely. Therefore, explicitly over here, I have a step, which says clean up the stack, in another words pop all the difference things, which had been pushed on to this stack in connection with this function call.

So, this stack is back to the situation that it was at the point of the call. So, the cleaning above this stack is basically popping of the various things you had been pushed on to this stack as part of the implementation of the function call.

(Refer Slide Time: 43:25)



Problem: Separate Compilation

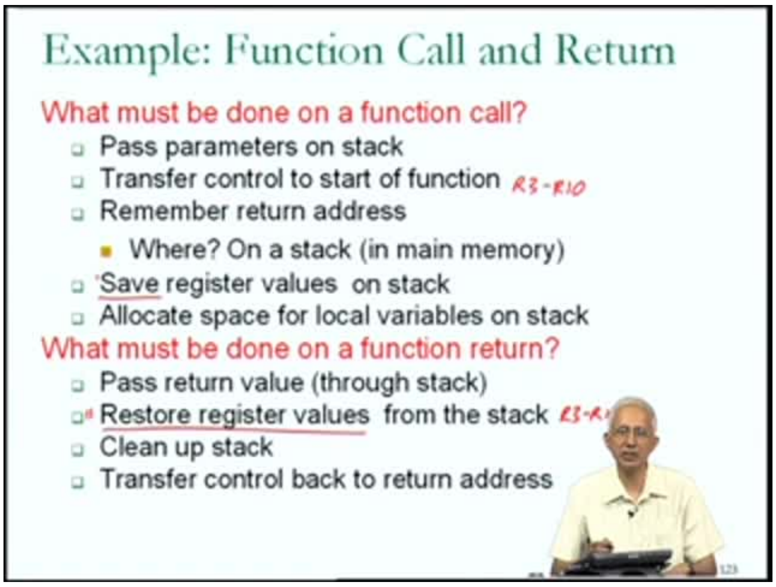
- Consider our simple example of compiling a C program in program.c that calls a math library function
- % gcc program.c
  - cc1 might use general purpose registers R3-R10 for the frequently used variables
  - But, what if these registers are used by the math function, which was compiled previously?
  - When the math function is called, the values in R3-R10 would be over written and therefore lost
  - Unless we save the values of those register part of the function call

So, we are almost there. Now, there is only one or two more things that have to be done and these as I said, relate to the problem separate compilation. Now, I had briefly a little to this little while back, but let me spell it out more detail. Now, separate compilation is the idea that the program, which you are compiling, may generate a dot o file as a result of the compilation which has to be linked with other files, such as math library. And in the compilation of program dot c into program dot o, is possible that cc1 used some registers in order to frequently used variables in order to speed up the execution of the functions inside that particular part of the program.

So, let us suppose, for example that cc1 use general purpose register R 3 through R 10 inside program dot o for the frequently used variables of that function, of the functions inside program dot o. Now, it is quite possible that the same register I used by functions inside math dot o, we have no idea what registers are used by the functions inside math dot o; therefore, it is certainly possible that R 3 through R 10 were used inside the math library function as well.

So, if that is the case, then whenever function of my inside program dot c calls one of the math functions, the values which had assumed it had placed in to the registers R 3 through R 10 would get over return by the use of those same registers inside the math function. Therefore, there is a problem here in that, every time for within my program dot c a math function is called during the execution of the program values, which they assume would be way available in R 3 through R 10 will disappear that be over return by other values. And the program will not do what thought it was going to do.

(Refer Slide Time: 45:57)



Example: Function Call and Return

What must be done on a function call?

- Pass parameters on stack
- Transfer control to start of function R3-R10
- Remember return address
  - Where? On a stack (in main memory)
- Save register values on stack
- Allocate space for local variables on stack

What must be done on a function return?

- Pass return value (through stack)
- Restore register values from the stack R3-R
- Clean up stack
- Transfer control back to return address

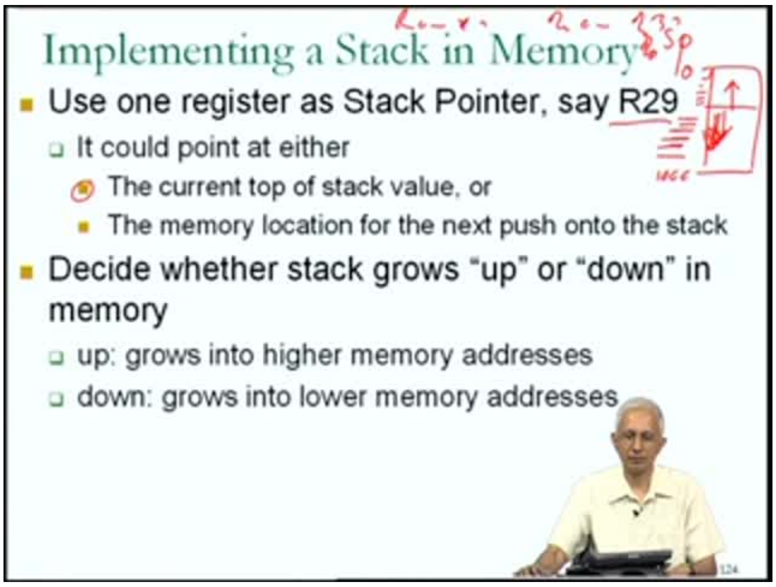
Therefore, it is necessary as part of the function call that we save the values of those registers. In this particular example, R 3 through R 10 as part of the function call. So that before control is transferred in this case to the math library, the values of R 3 through R 10 have been saved and then subsequently on return from the math library function back to the function which did the call, the values of R 3 through R 10 could be restored from wherever I had save them. Therefore, the saving and restoring of registers whose values might be over return by the function, which is being called is in essential part of the function call and return from our perspective. And that is in fact what remains we put into this sequence of steps.

As part of the function call, which is up here I need to save the values are registers which might be used, but which have been used in the function that is being called in the caller; so that they could be restore on return. So, this registers are saved as part of function call and they are restored as part of function return and only this registers, which I need. In other words registers, which contain values of frequently use variables, only those registers as in our example may be on the R 3 through R 10, it is not always necessary to save all the registers. The compiler knows exactly, which registers it has been using for the frequently use variables and can therefore save only those. This of course, this is the one for last question, where will the saving of these register values be done and once again the saving of the register values can be done on the stack.

So, the values can be saved on to the stack as part of function call and they can be restored from the stack into the registers R 3 to R 10 by load instructions. So, this anything is going to happen through store instructions and the restoring is going to happen through load instructions back into the registers R 3 to R 10.

So, this is what happens in a simple function call, not as trivial as we may have thought. And all of these have to be implemented by the compiler or by the person who is writing the program, if you are not writing the program in c. So, if I was writing this program in assembly language or in machine language I would have to do all these steps to achieve a function call. Or if my c program is being compiled by GCC, then it will have to include instructions in the machine language equivalent program, which do all these steps.

(Refer Slide Time: 47:50)



Implementing a Stack in Memory

- Use one register as Stack Pointer, say R29
  - It could point at either
    - The current top of stack value, or
    - The memory location for the next push onto the stack
- Decide whether stack grows "up" or "down" in memory
  - up: grows into higher memory addresses
  - down: grows into lower memory addresses

So, let us just make sure we understand what I was referring to earlier about implementing a stack in memory. We understand what a stack is and we understand that associated with the stack we must have a stack pointer.

So, since the stack pointer is going to be used very frequently as part of doing all the operations on the stack I am just going to assume that it is in the interest of the program to use a general purpose register as the stack pointer. And I will assume that I am using R 29 as the stack pointer and that R 29 will not be used for anything else. And the just for completeness, it is often the case that compilers will use, such as GCC may use a particular register as the stack pointer. And for example, if you look at the assembly language program that we looked at, you will notice that there is a reference to a register Sp. The notation that the use in that particular assembly languages, they do not refer to the registers as R 0 through R 31, but they refer to them as I think percent 0 through percent 31.

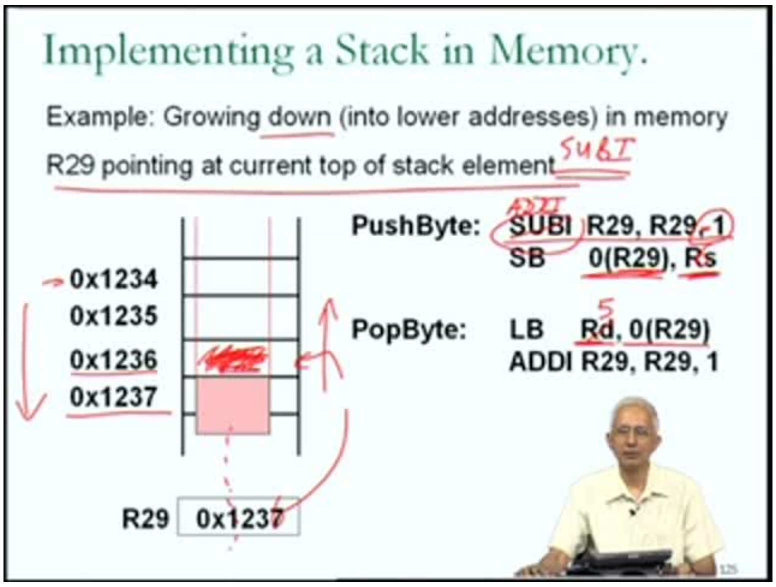
And one of the registers therefore is referred to as percent Sp. And that is in fact, the assembly language way of referring to the register, which is used as the stack pointer. And typically, I believe is the R 29 is the register which is use as the stack pointer. Also we need to use a register as the stack pointer to make the operation of the stack as fast as possible.

Now, what could the stack pointer point at? Now, I had mentioned that the stack pointer should always contain the address in memory of the top most element in the stack. But you could also choose to call the stack pointer to point at the memory location of the next free location, which will be used for the next push onto the stack. Either of these would be perfect, there is no just point out, that this is not the only of the implementing the stack in terms of the stack pointer. Now, the other decision that one may have to make about in terms or implementing a stack in memory in the assembly language or machine language this I may have to decide whether this stack grows up or down in memory.

We talk about this stack having to grow, so if this is memory and the stack is currently up to this point the question of whether it goes in this direction or grows in that direction is what we are talking about over here. In other words, does it grow into lower and lower memory addresses or does it grow into higher and higher memory addresses.

And typically, when I use the growing up in memory I will be referring to growing into higher memory addresses. Therefore, the arrow moving in this direction is what we were referring to as growing up in memory. This note that typically one, which shows the lowest memory address the top and the highest memory address 8000 and whatever at the bottom. And therefore, even though this arrow is pointing down it is growing into higher and higher memory addresses, which is where I refer to it as growing up. The alternative is also possible depending on how the allocation of this space of memory is being done by the system.

(Refer Slide Time: 51:05)



Implementing a Stack in Memory.

Example: Growing down (into lower addresses) in memory

R29 pointing at current top of stack element SUBT

PushByte: SUBI R29, R29, 1
SB 0(R29), Rs

PopByte: LB Rd, 0(R29)
ADDI R29, R29, 1

So, just you run through a quick MIPS one implementation. Let us assume that I have a situation, where I want to implement. I want to show I want to write the instructions for pushing and popping elements on to a stack and here I am showing a reason of memory, which contains the stack. These are byte addresses the addresses are shown in x, 1234 1234 1236 1237. So this is increasing memory addresses.

So, I am going to assume by the stack grows down in memory, in other words a grows not in the direction of the arrow, but in this direction it is growing into lower and lower memory addresses what is what you mean by down. I will also assume that R 29 is being use at the stack pointer and that it points at the current top of stack element.

So, let us assume that right now the stack looks like this, what is this mean? This means that I stack they could they could be stack elements in higher and higher memory addresses, but the top most element in this stack is at memory location 1237. Therefore, the stack pointer, which is at R 29 currently, contains the address x 1237 because R 29 points are the current top of stack element.

How now do I implement push? Let suppose that the particular stack which I am dealing with the stack operations, which I am dealing with or to push a byte and to pop a byte, out of the stack. How I push a byte? In order to push a byte, I know that the new byte is going to occupy this memory location right and that in order to write something into that memory location I could first of all decrement the stack pointer.

How much to I decrement the stack pointer by I will need to decrement the stack pointer from 1237 to 1236, which is why I have this subtract instruction. Well you could read this, as we saw they may not be a subtract instruction so this might be add minus 1. And so minus changes to this, add immediate R 29 R 29 one's minus 1.

So, what is this instruction doing? It is causing this stack pointer to get decremented by 1 and that is going to call this stack pointer to now pointer 1236. Now, in order to push a byte on to the stack I can use a store byte. Store byte into which memory location? Into the memory location with displacement of 0 of the base register R 29. In other words, into 1236 plus 0, remember that this is the base displacement addressing mode which is use in all load and store instructions. And in other fact is that whatever value I want to push lets I am assuming that the value is currently available in some register Rs some source register it could be it could be R 6 R 7 whatever it happens to be.

So, the byte the least significant byte from that register will get return into the top of stack memory location. So, these two instructions implement the push byte. How do I implement the pop byte? I could leave this as a simple example for you to work out. But it is clearly going to be the opposite in the sense that it is going to start with a load byte, in other words read from memory that byte which is at memory location base address R 20 nine plus off set displacement of 0. In other words, whatever the top of stack is currently pointing at and read from that memory location into some register. The destination register again this could be R 5, whatever register I want value to come into for subsequent computation. Subsequently, I need to update the stack pointer, which I do by incrementing it by 1 and there effective that would be that the top of stack gets a raised and I can do the incrementing using add immediate R 29 R 29 1.

Now, before closing with this example let me just talk about what I had previously over here. Previously, instead of using the add immediate instruction, I had used something called subtract immediate, but in talking about the MIPS instructions I had explicitly said that there is no need for us subtract immediate instruction. And therefore, they would not be subtract immediate instruction implemented in the MIPS one hardware because whatever the subtract immediate instruction would do could have been done by an add immediate instruction.

However it is conceivable that in the assembly language subtract immediate could be supported as an aid to the programmer. And all that the assembler has to do is that whenever it sees a subtract immediate, we can replace by add immediate and then just take the negative of whatever the immediate value is.

So, this is another kind of an example what an assemble assembly, how an assembly language might defer from a machine language. Some additional, simple instructions might be added which might not actually be implemented inside the hardware something like this subtract immediate, but which might be a value to the programmer in order to more easily understand and write programs in style that is let me say more readable.

So, with this we have understood the sequence of steps that must happen in a function call and return, and we understood that is stack is very important part of the sequence of steps. We have also seen how to implement a stack in memory using the MIPS one instruction.

Thank you.