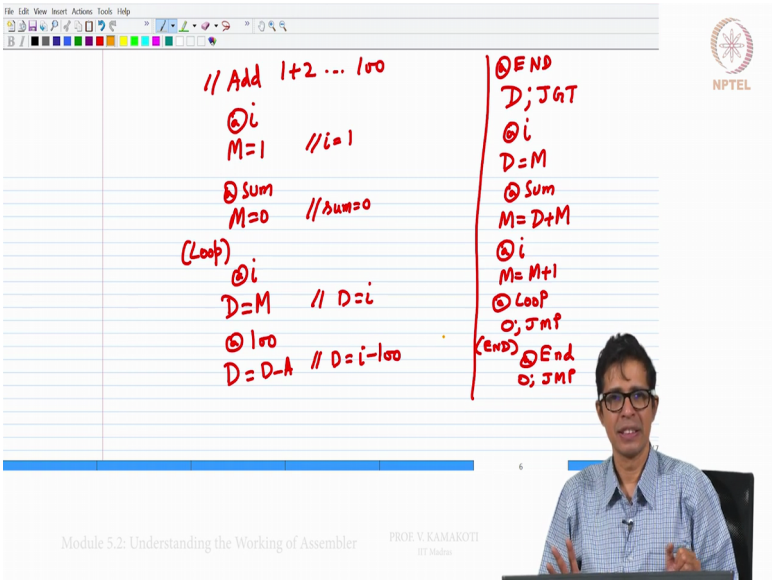
Foundations to Computer Systems Design
Professor V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology Madras
Module 5.2
Understanding the Working of Assembler

So welcome to module 5 point 2. In this we will understand the working of the assembler, right. So this is the input to your assembler. This is the mnemonics.

(Refer Slide Time: 00:32)

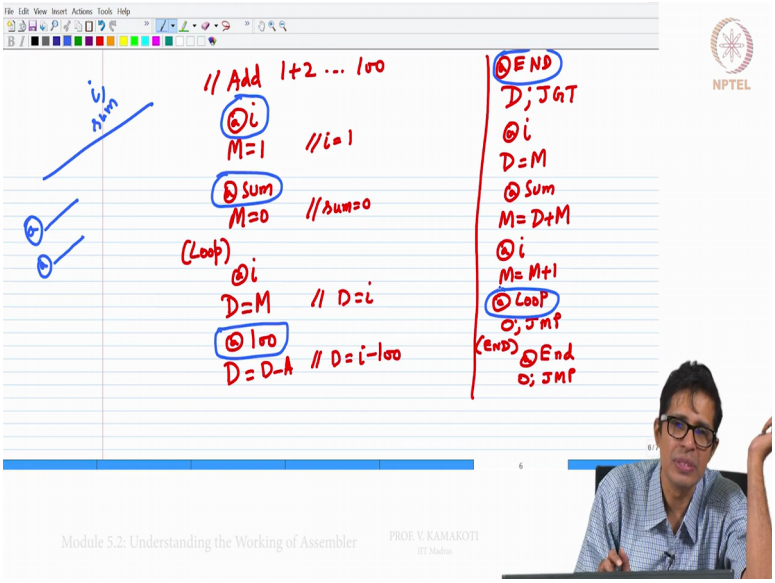


Now we have to convert this mnemonics into the series of 16 bit instruction. Now as you see there are 20 instructions here and we have to convert each one into its equivalent 16 bit part. Now these instructions are going to be in the instruction memory, the ROM 32 k and the data that it is going to use, what are the data here? For example i is a data, sum is another data, at least two data points that we have seen, i and sum are basically going to be in the data memory, right.

So when the assembler is assembling it should also assign some memory location for i and sum in the data memory, right? So the role of the assembler is also allocating memory for i and sum in the data memory, right? Now let us just look at this. Understanding of the C type instruction is very straight forward, right? But understanding of the A type instruction there are some things that we need to note here. Let us first look at the A type instruction at.

At the at you can have a constant for example at 100 as you see here right, or after at I could have a variable as you see at i here, at sum here, at end here, at loop here, right? So there is a difference between at i, at sum, at end and at loop.

(Refer Slide Time: 02:45)

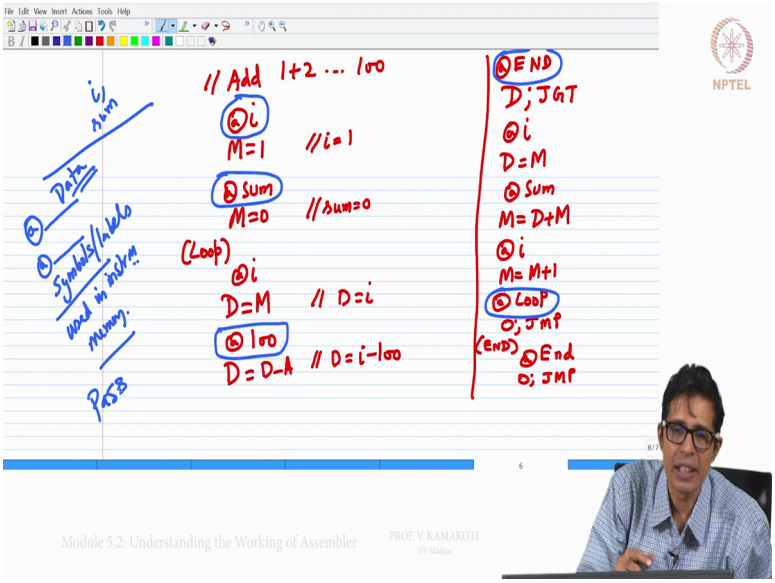


The difference is very simple, i and sum are basically data while loop and end are basically symbols or labels used in instruction. What do you mean by used in instruction memory? They are used to designate locations in the instruction memory. If I say jump to loop, where is that address, jump to which address? Jump to an address within the instruction memory. So at instruction itself we need to distinguish three things, whether what is going to follow that is a constant, right? It should be a decimal constant.

It is a non negative decimal constant or it will be a symbol. This can point to data, this can also point to a label like loop and end which will be pointing to some location in the instruction memory. In addition you will have comments and these comments have to be ignored, right? So, one of the basic steps that is involved in assembly is to first resolve these labels. Finally when I translate to binary every one of this at instruction will have a value, right?

We do not have i in the binary representation. I need to replace i, sum, loop, you know, end, all these things which have 15 bit binary value and that is very important here, right? So the assembler will first go through this program once. That is called a pass.

(Refer Slide Time: 04:55)



It will pass through this program once and then it will pass through second time. So even for the major system the assemblers are two pass assemblers. What do you mean by two pass assemblers? The assembler will look at the program twice. It will read the program twice. In the first pass it will resolve all the symbols. So the first pass it will resolve all the symbols. What do you mean by resolving symbol? For every symbol that is following an at instruction here, I need to replace it with a 15 bit binary.

So that means I need to have a constant value assigned to every such labels that is following an at instruction for example i, sum, end, loop, all these things you are seeing in this particular code, I need to replace it with a constant. Then only I can convert it into the corresponding A instruction, right? So the first pass of the assembler is basically replace these symbols with constant values so that in the second pass it basically generates the binary. There is no problem for us.

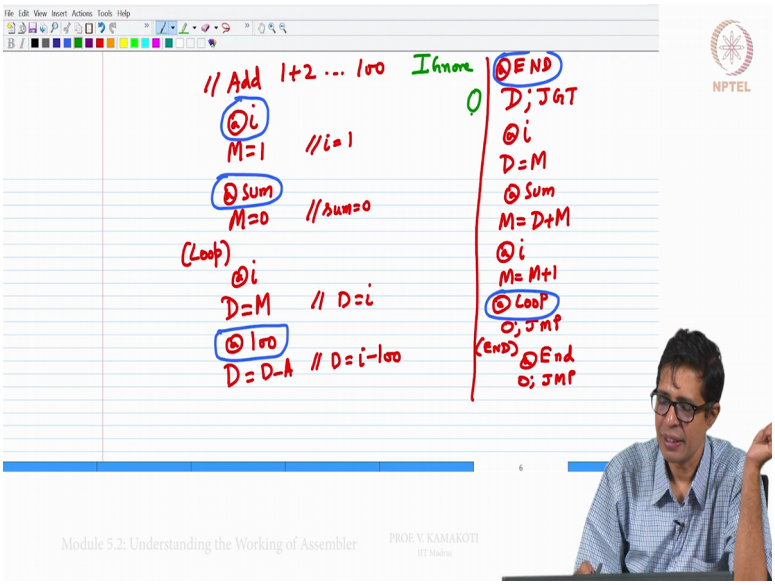
We have understood how to convert M equal to 1 as you see here, the second instruction or any of these C instructions. How it is converting into the binary, there is no issue. The A instruction, this is the issue we need to resolve the symbols. We call it the symbol resolution. To do the symbol resolution we basically have something called a symbol table. So when the assembler is going through the first pass, it crates something called a symbol table and what is that symbol table? It is very simple.

For every symbol that I am using, its equivalent constant that I need to replace will be there in the symbol table. So the symbol table well map every symbol in your mnemonic program

into its corresponding value, right? So the pass one of the assembler is basically to construct this symbol table, right? Now let us see how it is going to do that. Okay, now the first one is a comment. So your first line is a comment. You have to ignore. Nothing there, so a comment does not become binary.

The second one is at A instruction. So you should maintain a counter here. The counter currently is 0. This means the equivalent of this A instruction will be in the ROM, that is instruction memory, location 0, right?

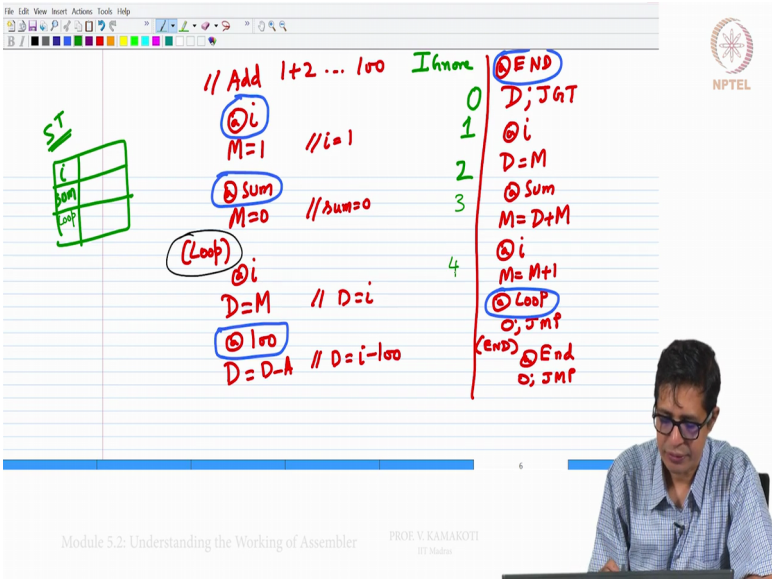
(Refer Slide Time: 08:39)



At this point you have found a symbol i so you register that symbol i in the symbol table. You do not know what that i is but you just register. This i can be a label, this i can be a label like loop, right? We do not know what is i. So register i. Now go to the next instruction. Now this is not an at instruction, this is a C instruction so just make it 1. I am now describing the pass one of the assembler, the first pass okay. Now I come to the third instruction that is 2. I see another symbol here. I register that symbol sum in the table, okay.

Now I see that fourth instruction. Now I see a symbol look, okay. This is a symbol, right? So the loop is registered into the symbol table. This is three, this is pointing to the fourth instruction.

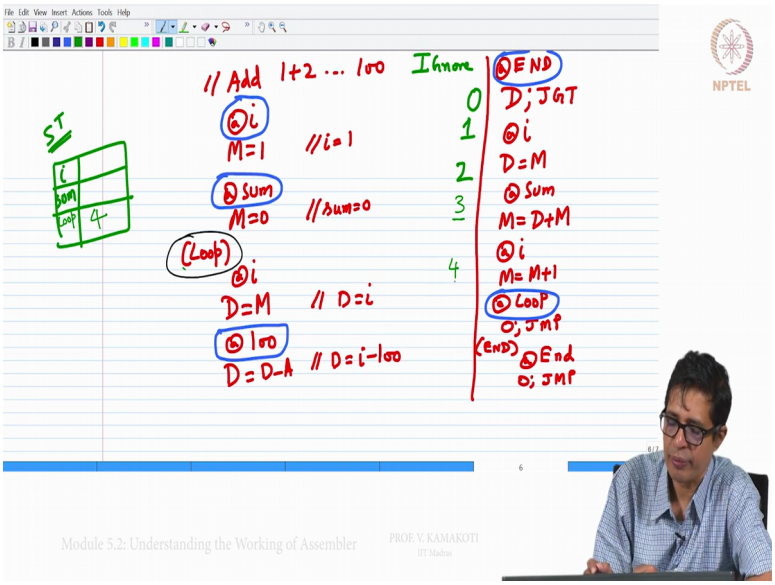
(Refer Slide Time: 10:08)



So if I say at loop means I have to jump to the fourth location in the ROM. So when the moment I see a symbol that is with the parenthesis here, this is the label here. Moment I see a label, the counter here is 3. I put the next 3 plus 1, 4 here. So loop becomes 4. Then next is an at i instruction. So I see this i, I go to the table and see if i is already registered. I need not register again. So that is one very important thing. When we are populating the symbol table we are just building the symbol table.

Every time a new entry has to be entered we should check whether it is already registered. I is already registered so nothing to do. We make it 4. Counter becomes 4. So for the symbol there is no equivalent binary, right? So this is not counted. So I come to i and make it 4.

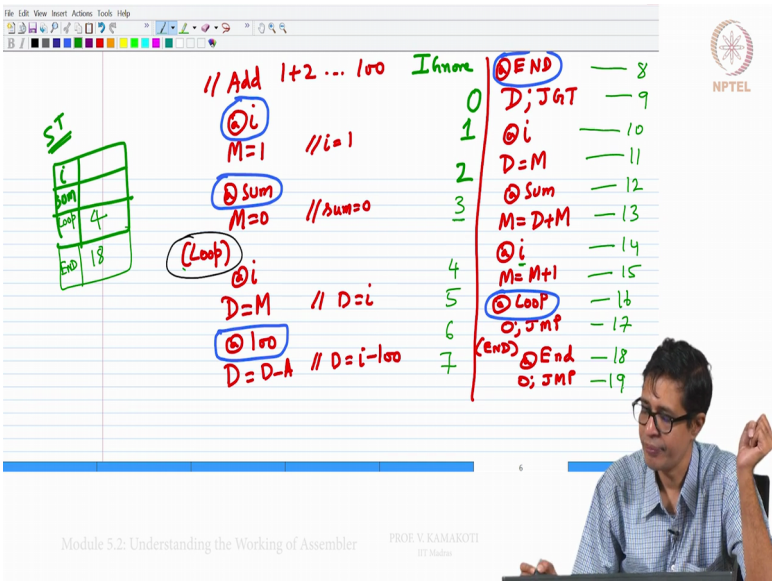
(Refer Slide Time: 11:14)



Now D equal to M, this is 5. Now at 100, 100 is a constant so nothing to do. This becomes 6. D equal to D mina A, 7, at end. So we enter end here and this is going to be your 8th instruction. Then this is your 9th. We are just counting this. At i this is your 10th. I is already registered so nothing. Then this is 11. At sum is 12. Sum is already registered, nothing to do. This is 13, this is 14, this is 15. Again i is already registered here so nothing to do. At loop, again loop is already registered here, nothing to do.

Now 0 colon jump, this is nothing to do. Now end is this symbol. So, end points to the 18th instruction in your RAM, 17 plus 1. Now I go and see if end is registered. Yes end is already registered so now I update that value to 18 and this is 19.

(Refer Slide Time: 12:35)

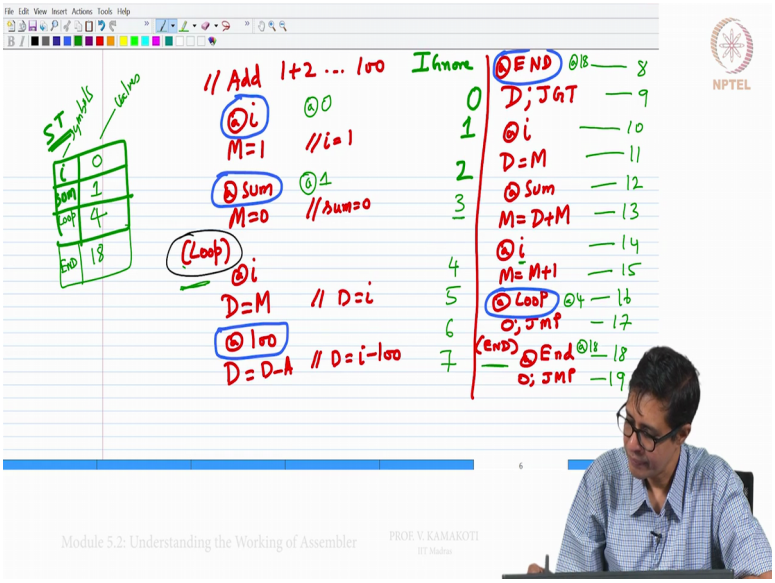


Now after you complete the program, these are the symbols and these are the values. The second thing is the value. For whatever symbols the values are already populated they are all labels in the program. For example loop and end as you see here are labels in the program and those values the remaining thing whatever is not populated, they correspond to data. So i and sum are data, okay. So i and sum they are in the data memory so I can make i as 0 and sum as 1.

So i will be in 0th location in the data memory. Sum will be mapped onto the 1st location in data memory. Let us understand this. Now what has happened as a part of your first pass? All the symbols have been resolved and we had clearly made a distinction between whether it is a label of the program inside the instruction memory or it is a data pointing to some variables in the data memory.

In this case loop and end are basically symbols in the program so they get 4 and 18 as you see here while i and sum are basically data in the data memory and then for that we assign 0 and 1. So now we can just see that this is going to be at 0, this is going to be at 1 and this is at 100. This is going to be at 18 and this is going to be at 4 and this is going to be at 18, right?

(Refer Slide Time: 14:41)

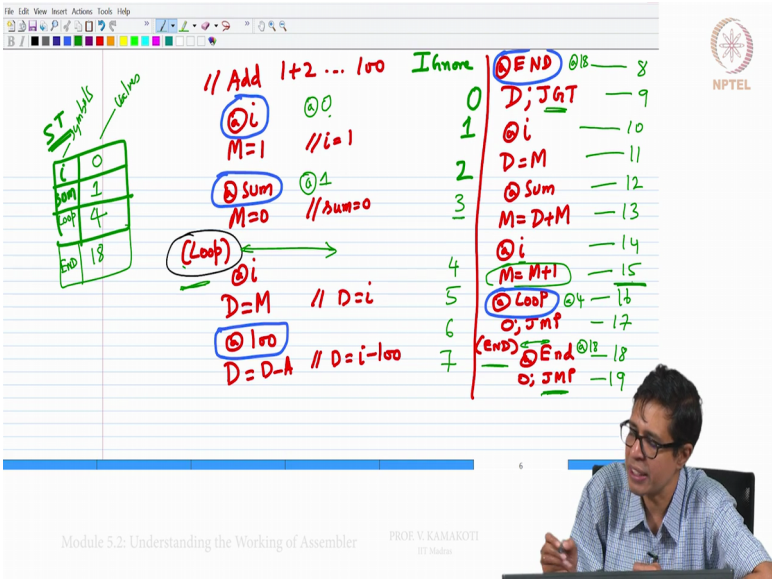


Now the program works so at loop is nothing but at 4. I jump to the 4th instruction. At i is I access i which is equivalent to at 0. At 0 is in data memory at 1. Similarly at end when I jump 1 greater than I jump to at 18. At 18 is this and I do (con) unconditional jump again back to 18, it is at 18.

So essentially we have actually resolved all the symbols and this is precisely what your pass one of the assembler has to do. So let me just sum up some of the important points here. First thing is we have a running counter. That counter will ignore all the comments like it has ignored at and ignore all the lines which basically has only the symbol like loop had only a symbol, end had only a symbol, etc.

And then for every instruction that it sees, a C instruction or A instruction, it increase. It starts with 0. So ultimately this counter tells for every instruction which memory location in the instruction memory it is going to be loaded. For example at i will be loaded at 0. This M equal to M plus 1 that you see here is going to be loaded at 15, right?

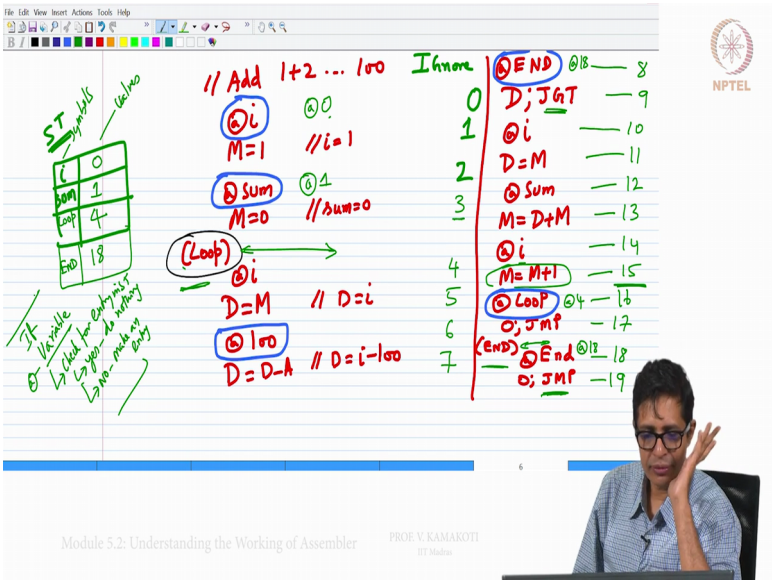
(Refer Slide Time: 16:18)



So like that, okay. And when we are doing this part whenever we see an at instruction and if there is a constant we do nothing. If there is a variable then you go to the symbol table and see if that variable is already registered. If the variable is registered, do nothing. If the variable is not registered then you make an entry.

This is what we do at the at instruction. So what we do at the at instruction? If variable is there, check for entry in symbol table. Yes means do nothing. No means make an entry, okay. Yes means do nothing, no means make an entry, right?

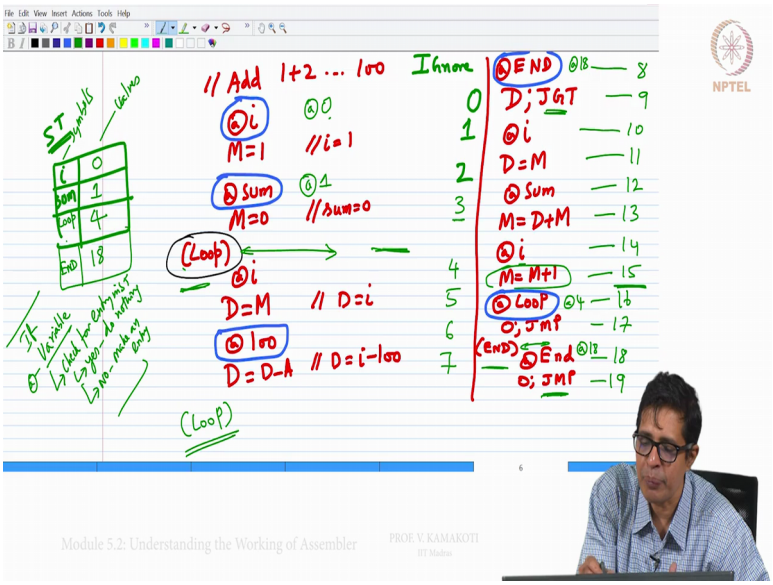
(Refer Slide Time: 17:26)



And this is what you do for every at instruction. And whenever you see a symbol, symbol is with parenthesis. There is a counter, whatever that current counter plus 1, right? Whenever you see a symbol, first go to the symbol table and find out if the symbol is registered. If it is registered, fine. If it is not registered, register it.

In addition also whatever the current counter that plus 1 you put in the value of that symbol, right? So first time we saw loop. Loop was not registered in the symbol table, the loop that you see here. I am marking it here.

(Refer Slide Time: 18:13)



That loop was not registered in symbol table. So I go and register that in the symbol table as you see here. Loop is registered and also we enter that counter. The counter was 3 when we encountered loop. So we basically 3 plus 1, 4 is put here. So the moment when I see loop I put 4. So whenever I see a symbol which is within two parenthesis, a label, we will call it label henceforth, I see whether that label is already registered. If it is not registered you register it in the symbol table.

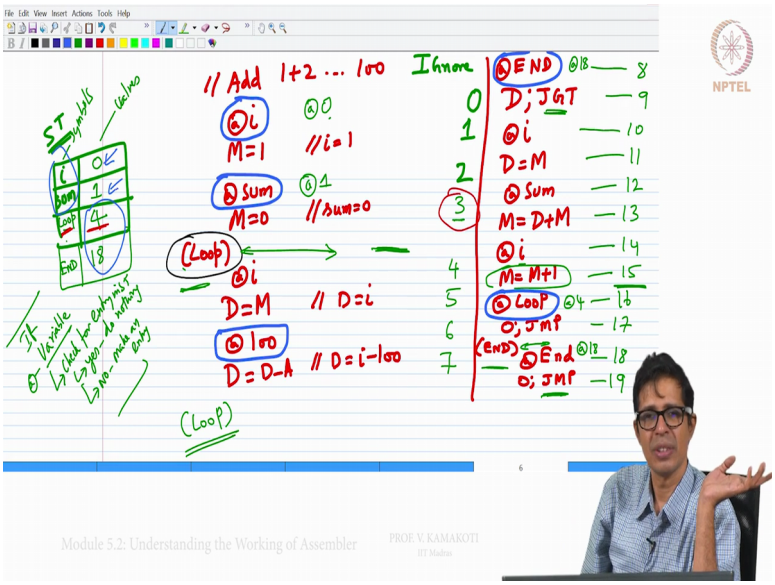
And also along with it associate the value which is the counter plus 1, right? And that is what we do. So this is the thing. So in the pass one we have to have the counter running and every time I encounter an at instruction or a label I know it start going and accessing the symbol table and do the actions as I had mentioned.

At the end of this complete one pass after we finish say in this case all the 19-20 instructions, then you go and see what are all the symbols in the symbol table where the values are already finalized. In this case you will find that I am again going back to symbol table. I will find that

this 4 and 18 are already finalized. For which the symbols are already finalized they are actually labels in the program obviously, right?

And for the remaining thing for which the values are not finalised in this case i and sum, you just sequentially assign them starting from 0. In this case so I will assign i as 0 and sum as 1.

(Refer Slide Time: 20:22)

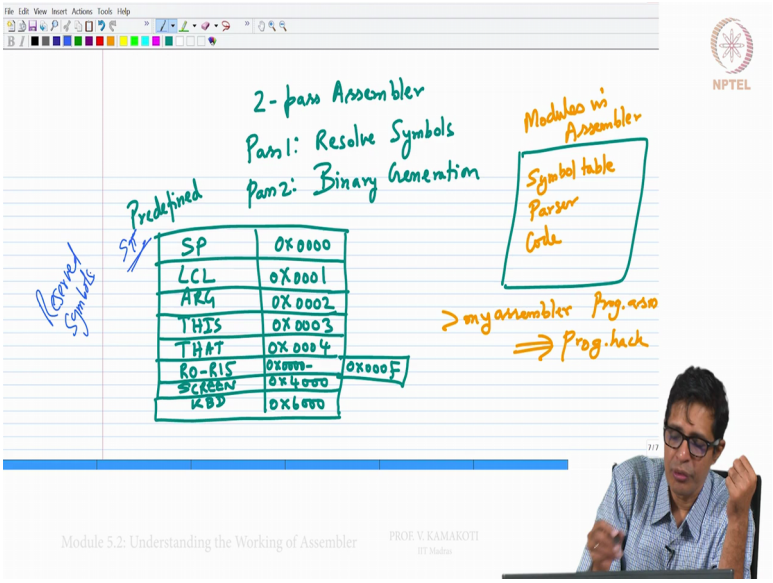


And if you do that, then essentially the resolution of the symbol is over. Once I have that resolution of the symbol done now it is a more straight forward conversion for me during the phase two or the pass two of your assembler. So this is pass one of the assembler. Now the only practical change between what we have seen here and the real construction that we will be doing as a part of your project 6 is that there are some predefined symbols like you have been using at screen, at keyboard, etc. right?

We had seen in the previous case, so like that there are lot of predefined symbols that will be used actually by your virtual machine later. So as of now there are certain predefined symbols. An example of your predefined symbol is your screen and keyboard. So when you initialise your symbol table please note that this is also there in the book. When you initialise the symbol table, right, the symbol table will have these entries already.

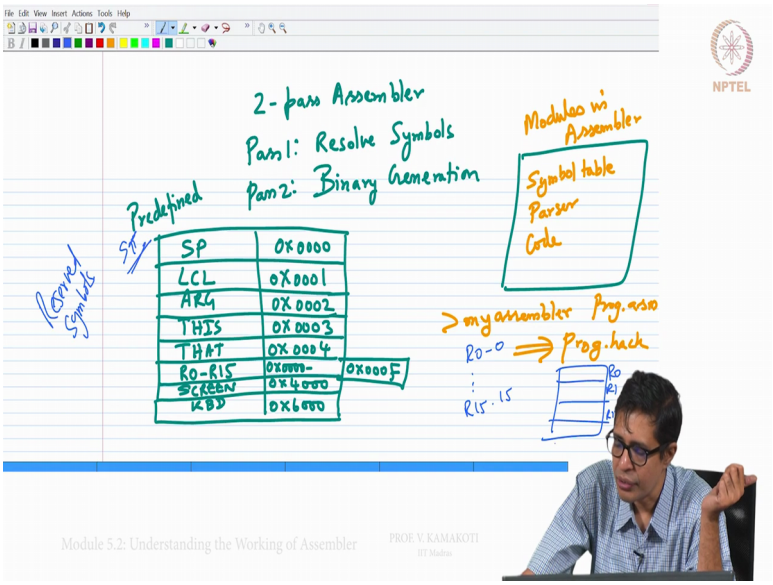
For example it will have SP which is basically stack pointer. We will see what is stack pointer later, right? LCL local right, ARG argument. So SP, LCL, ARG, THIS, THAT, we will see what is THIS, what is THAT later. Then you will have 16 symbols, R 0, R 1, R 2, R 3 till R 15, then screen and keyboard. These are all what we call as reserved symbols. We cannot use this symbol in our program.

(Refer Slide Time: 22:13)



For example in our previous thing I could not say that this is at R 0 or at screen. I cannot say at screen and have my own variable, right? So these are all reserved things so I cannot use in of i any of these symbols, right? So these are reserved symbols. So SP, local, this thing. So in your data memory R 0 to R 15 are reserved. That is 0 to 15 are reserved. So in your data memory what will stay is R 0, R 1 to R 15 will stay in the data memory first. So the first sixteen locations of your data memory is reserved for R 0 to R 15, okay.

(Refer Slide Time: 23:05)



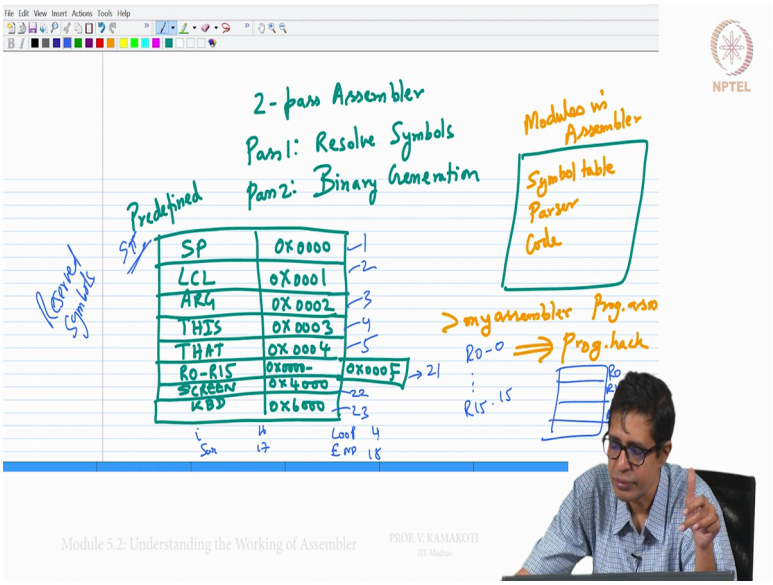
So that means that your i and sum cannot be in your 0 and 1 location. Your i and sum will start from the 16th and 17th location. So the only change between the explanation I give earlier and this is that after you finish your one pass, you will go and find out what are the

symbols that are already resolved like loop and end that are 4 and 18. There is no change about it. Then there will be some symbols which are not resolved. Those are variables in the program in this case.

What I told earlier was you start numbering them from 0 onwards. Instead of doing from 0 onwards do from 16 onwards, so 16 and 17, right? So essentially your symbol table will have i, sum, loop, end which is 16, 17, 4 and 18 and that in addition to this SP to this. So there is one entry, second entry, third entry, fourth entry, fifth entry, there are sixteen entries here so which will be to 21, 22, 23.

So, 23 entries are always there by default when you create the symbol table. In addition for this particular program you would have created four entries which will be beyond this. So i is 16, sum is 17, then loop and end are 4 and 18, okay.

(Refer Slide Time: 24:40)

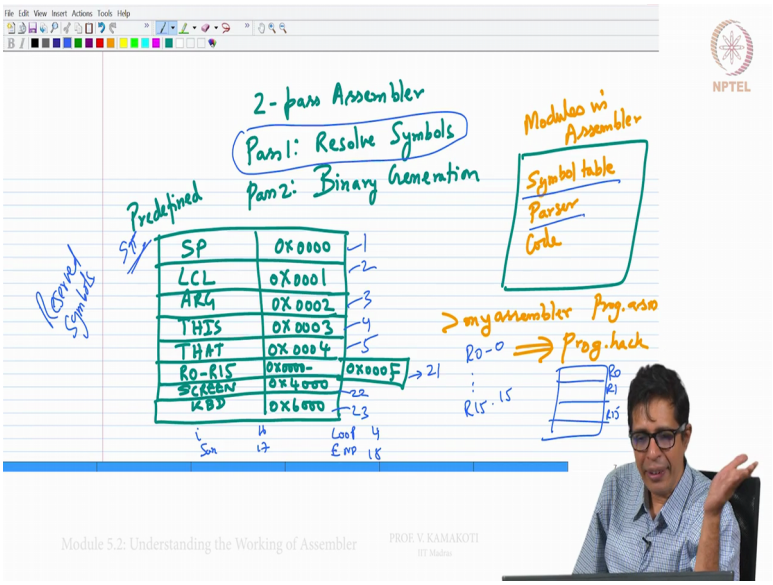


So totally this symbol table now will become twenty seven entries, right? So this is what you do in pass one where we resolve symbols. Now in the pass two what we do, every symbol is resolved so we just have to now convert this each instruction into its corresponding binary. So twenty binary instructions will be created by the assembler when we take this.

So predominantly the assembler has three modules, one is the symbol table module which will be used in pass one, (cons) the symbol table will be constructed in pass one and it will be used in pass two. In the pass two where do we use the symbol table? Whenever I see at i, I will go now to the symbol table to find out what is i at sum. I will go to find out what is sum, similarly at i, at sum, at loop, at end.

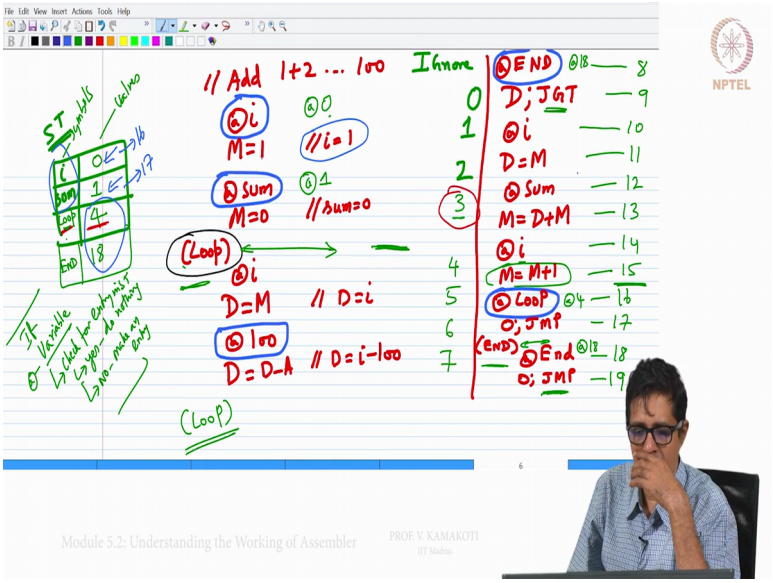
Whenever I see an at instruction and the symbol I will go and find out what is that, right? The other things I just have to ignore. So the symbol table is used in the pass two. So there are three modules as I have noted here. That module will be used to construct during pass one and that module will be queried in pass two when we are generating the binary. Then there is a parser.

(Refer Slide Time: 26:30)



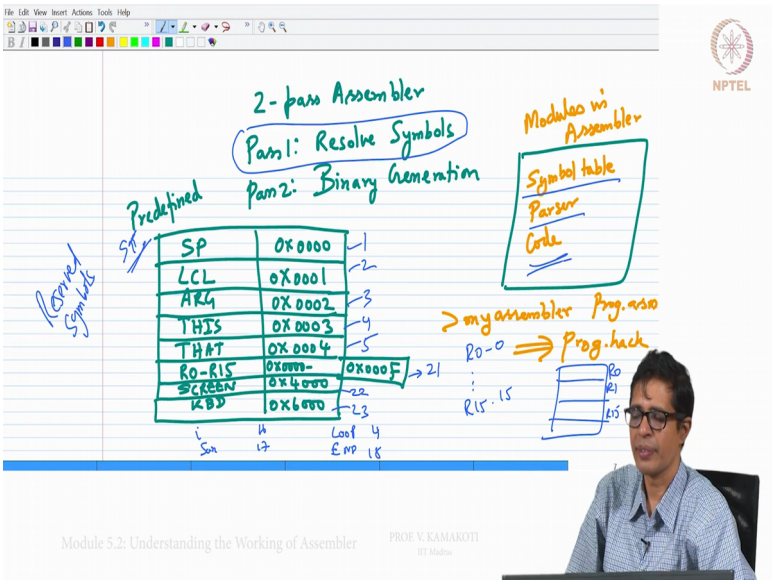
What will the parser do? It will basically read line by line and make a decision of what we need to do. For example the parser in this case will read line by line. The first line it will read, oh it is a comment. The second line it will see, oh it is an at instruction and then what is it, right? And then, oh the third instruction is M equal to 1 followed by a comment here. So this comment I have to ignore. I know what is M equal to 1. Then similarly at sum, at M equal to 0, loop, at i, all these things it basically loop.

(Refer Slide Time: 27:04)



Here in the pass two the parser need not do anything. In the pass one it should say it is a symbol, it is a label. Then only I have to go to the symbol table and do. So like that the parser basically is very much responsible for identifying the different components. So to sum up the parser actually makes a distinction between whether it is an A type or a C type or a label and then based on that we can take necessary action. The last module that we see is the code here. The code will basically take the past instruction and generate its equivalent 16 bit code. The code will be used for this.

(Refer Slide Time: 27:52)



So these are the three important modules that we need to generate as part of our construction of this assembler, right? Ultimately we need to generate the program called my assembler. To

this my assembler I will give you the program dot asm and my assembler should generate the program dot hack which is nothing but take all the instructions in your program dot asm, ditch all the comments and give a 16 bit equivalent of that which I can take it forward and run it on my hack hardware that we have.

So this is basically the two pass assembler and understanding of the two pass assembler. So next module we will now go and talk about some of the simple but vital things of how to make the assembler software. So this will give you an introduction of how to develop software, how to develop what you call as APIs, application programming interfaces. So this will be a good introduction to systems programming which is a very important part.

Today one of the major talent that our country needs here is system architects, people who can actually understand operating system, people who can actually understand compilers. There is a very big need for that type of a talent and I am sure when we start writing this assembler, when you go deep into how to write the software for that assembler, we will understand many things which in a long run will be extremely beneficial to us. So we will move to the next module. Thank you.