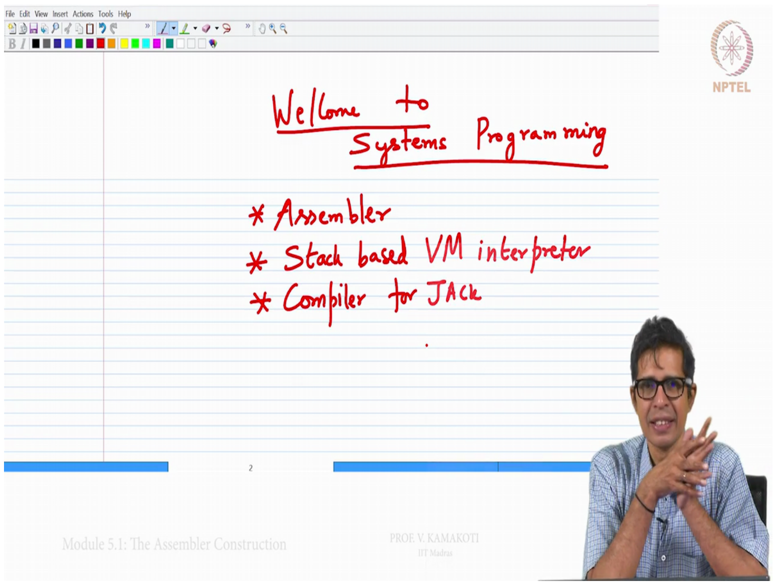
Foundations to Computer Systems Design
Professor V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology Madras
Module 5.1
The Assembler Construction

So welcome to module 5 point 1 and in this module we will be now moving on to the construction of an assembler. We have already used an assembler which was provided by the software but now we will design our own assembler and that is very important for us to understand how what we call as the system softwares work. So with this we actually welcome you to the systems programming part of this course.

(Refer Slide Time: 00:52)



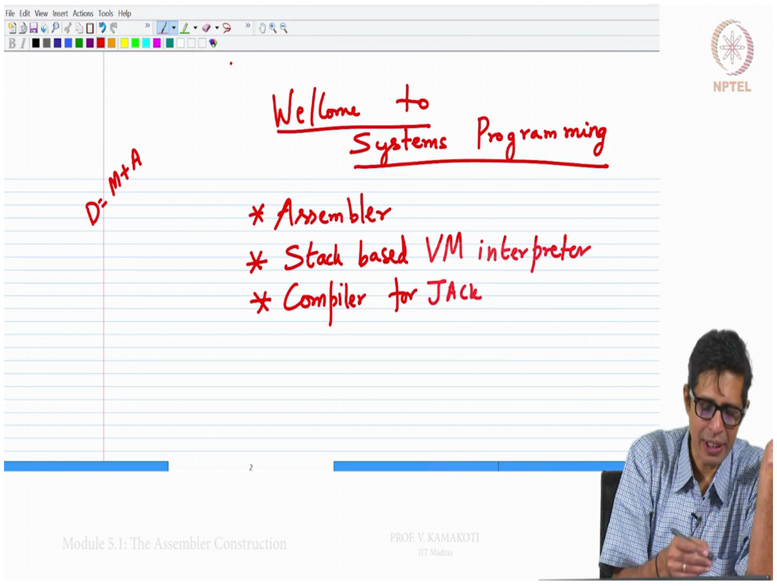
Welcome to Systems Programming

* Assembler
* Stack based VM interpreter
* Compiler for JACK

Whatever we have done so far is the hardware and now we need software that can execute on this hardware. So, building hardware is like building a dam, right? And you have to fill it with good water. The good water is the software. So software development is very important to have an efficient effective utilisation of your hardware. And what we will be doing in the remaining part of this course is to understand how software is being built for basically execution on the hardware.

And the first software that we will see is actually an assembler which will take a machine instruction in a mnemonic form and convert it into a 16 bit hack instruction, right? We have already used an assembler for example if we have D equal to M plus A, we know how to get that 16 bit instruction for a jump. All these things we have seen in the past.

(Refer Slide Time: 01:54)


Wellcome to Systems Programming

D= M+A

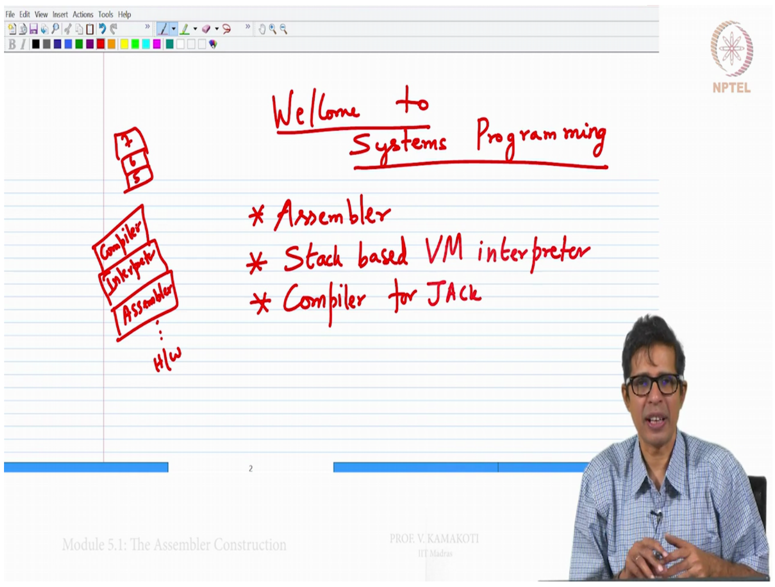
* Assembler
* Stack based VM interpreter
* Compiler for JACK

But now we need to write software which will convert the D equal to M plus A into that 16 bit string and that is essentially what the assembler will do. Now in the initial portion of this course we did mention that there is a machine instruction. Then there are some programming languages and there is a compiler which will convert your programming language constructs on to the machine language and that software is what we call as a compiler.

Compiler is software that will take a high level language and basically convert it on to a machine language. Now this conversion is not going to be that straight forward. To actually make this conversion process more effective, more simpler, all the recent software stakes. We call it as software stacks. What is a stack? Stack is nothing but a layer in which there are items one on top of another.

So a stack of numbers would have say 5, 6, 7, 7 sitting on top of 6, etc. A software stack is nothing but a collection of software one sitting on top of another. Now this is a software stack because assembler is the bottom which directly talks to the hardware, right? And on top of it we have what we call as an interpreter and then we have a compiler.

(Refer Slide Time: 03:36)

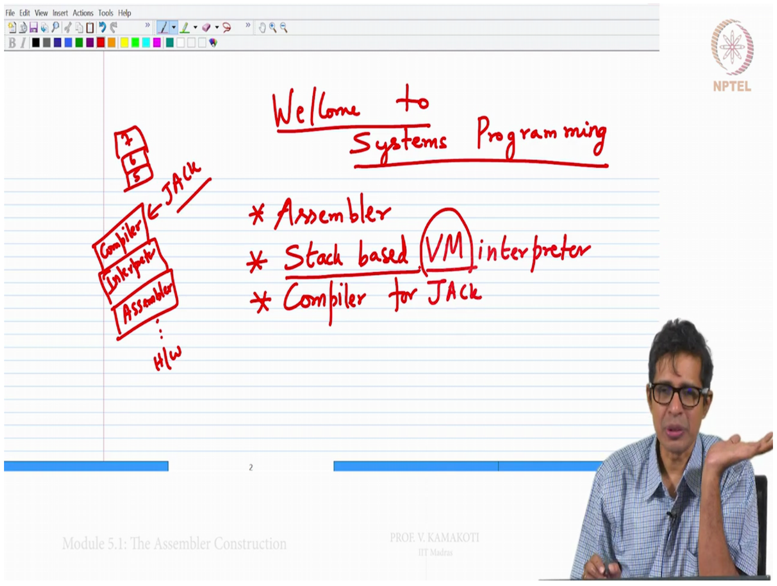

Wellcome to Systems Programming
* Assembler
* Stack based VM interpreter
* Compiler for JAck
Compiler
Interpreter
Assembler
H/W
Module 5.1: The Assembler Construction
PROF. V. KAMAKOTI
IIT Madras
NPTEL

Each of them is a software, one sitting on top of another so we call it as a software stack, okay. Now taking a high level language the language that will be introducing in this course, we cannot write a compiler for an entire C language, right? Just a part of the small course like this but never the less we need to understand how compilers work and that is also one of the most important objectives of this course. So we have taken an abridged version of Java called Jack, right?

As you can see in the prescribed reference book, I hope you have purchased the book or e copy. So it is very important for you to have that so that you can follow this course, right? So we have a Jack, right? Jack is the language and we have a compiler for Jack. So what will that compiler do? As per our initial understanding the compiler will need to take a program written in Jack and convert it into the machine language or the mnemonics, right?

So and that mnemonic win in turn be converted into binary by the assembler that we are writing now and that binary will run on the hardware but that translation from Jack directly to the mnemonics is going to be not that straight forward. So to make that process simpler we introduce a machine. It is called a machine which is a soft machine. So that is what we call it as a virtual machine, right? So we introduced a virtual machine what we call as a stack based virtual machine, right?

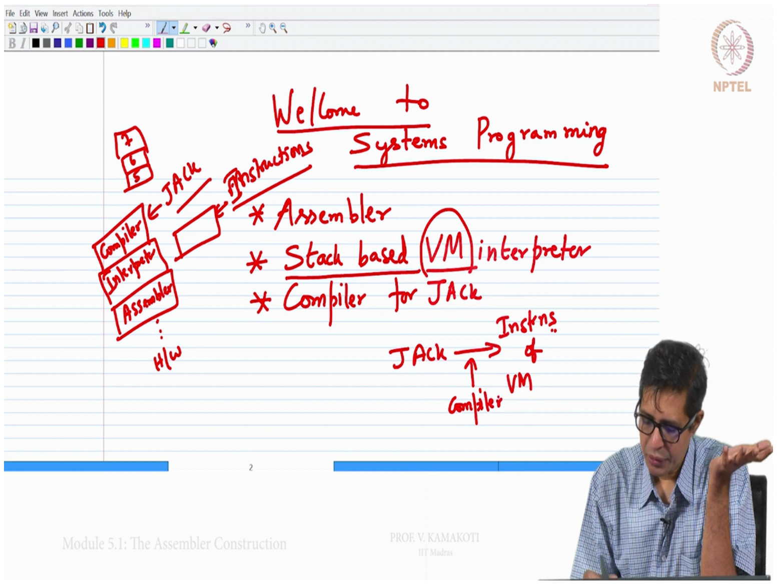
(Refer Slide Time: 05:19)


Welcome to Systems Programming

* Assembler
* Stack based VM interpreter
* Compiler for JACK

What is a virtual machine? It is actually not a physical hardware but it is a soft model of that hardware, right? So that virtual machine will have its own instructions. The virtual machine also will have its own instructions. So what happens here, your compiler will take that Jack program and translate it into the instructions of the virtual machine, right? So this is the role of your compiler.

(Refer Slide Time: 06:04)


Welcome to Systems Programming

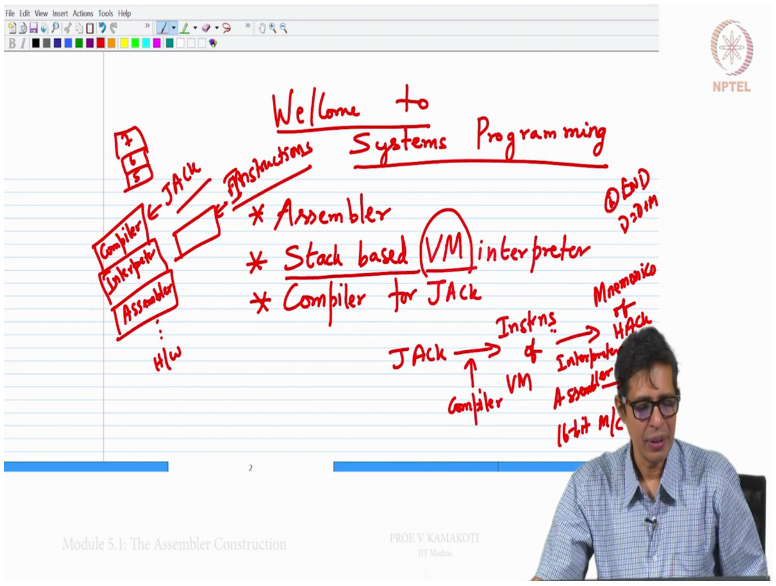
* Assembler
* Stack based VM interpreter
* Compiler for JACK

$$\begin{tikzcd} \text{JACK} \arrow[r] & \text{Instrns of VM} \\ & \text{Compiler} \arrow[u] \end{tikzcd}$$

So you write a program in Jack, your compiler will take that program and translate it into instructions of your virtual machine. Now we will write an interpreter that will translate these instructions of this virtual machine into your mnemonics of hack. Mnemonics are what we have seen no, at END, D equal to D plus M, all those programs that we have seen so far. So that will be the mnemonics of hack.

This mnemonics of hack will in turn be taken by the assembler that you are writing now and that will create you the 16 bit machine instructions which are in binary. And that 16 bit machine instruction will essentially be executed on the hack, right?

(Refer Slide Time: 07:04)



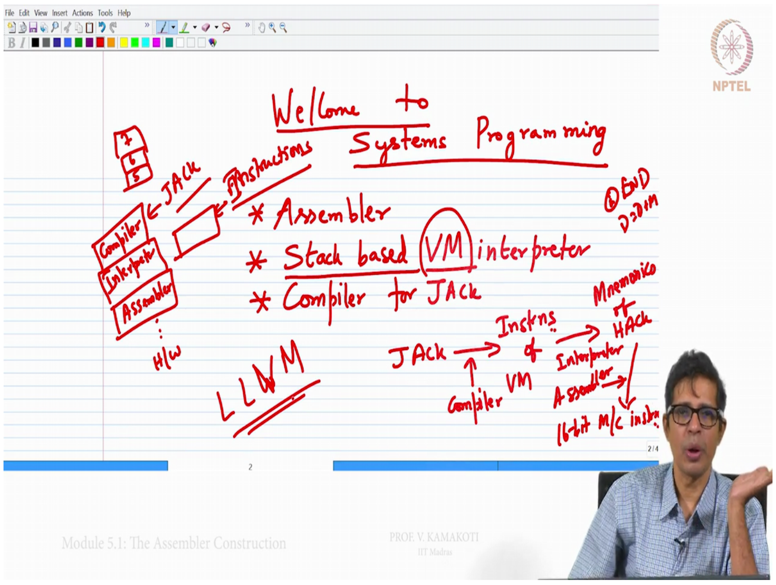
Now we have created a hardware which can accept these 16 bit binary machine instructions and run it. Now have to build this entire assembler, interpreter and compiler in the remaining part of this course. So to just sum up this, ultimately the user can write a program in Jack, use the compiler that you have made to convert it into an instruction, the entire Jack program will be converted into instructions of the virtual machine.

We will be introducing the virtual machine, its instruction set and its compiler will compile it to that and then the instructions of this virtual machine will essentially get converted into the mnemonics of hack by the interpreter that you will be writing and then that mnemonics of hack will then be converted into a 16 bit machine instruction by the assembler that you are writing. So essentially we have three software one sitting on top of another.

So from the top point the compiler is the first, then its output is interpreted by the interpreter and its output is assembled by the assembler. So essentially, this is the system programming stack. What we want to give you as a take away from this course is an experience to write this software and understand the complexities. There are a lot of simple things that you will learn which in the long run will make you understand the compiler for C and what they call as LLVM, the virtual machine.

There is a low level virtual machine LLVM. For you to actually appreciate how LLVM works and understanding the VM here will be a very useful for you to appreciate how LLVM works and then from there the interpreter to the mnemonics and of course the assembler which will take this mnemonics and convert it to 16 bits.

(Refer Slide Time: 09:00)



So this is the exercise. So the way we will be writing the software is first we will write the assembler, then we will write the interpreter, then we will write the compiler. So will go in this bottom of fashion for that, right? And that is what you will be seeing in the remaining modules of this course. Now let us look at the assembler. As I mentioned the assembler will take the mnemonics of hack and it will convert it into 16 bit hack binary codes, okay.

And you will understand this (proce) procedure of conversion from mnemonics to 16 bit hack binary code, you will understand the conversion and then we will code it. So you can start coding it in C++ or C or C sharp or Python or whatever language you know. I basically assume that everyone knows C because every curriculum across the country has C as an introductory language. So, I will give you a generic template of how to develop the software, right?

And you can use any of the languages to actually implement the template and I will be showcasing a C based implementation. But from the template you can translate it into C++ or java or python or anything of your choice whatever you are comfortable, right? But the ultimate goal is to take the mnemonics and give you a 16 bit hack. But more importantly we should also understand how the assembler works right and that is very important from understanding the systems engineering, right?

So first we will describe how the assembler works then we will go ahead and tell what is the generic template for basically developing assembler, how to write the software, right? So now to just have a quick understanding, the instructions are there in the instruction memory, the data is there in the data memory, right?

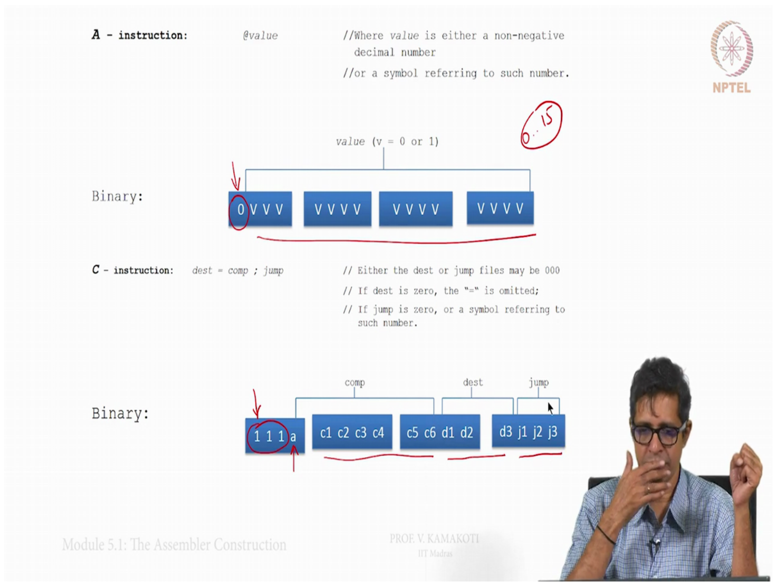
So when you assemble you assume in the contexts of hack that all your instructions will be put finally into the instruction memory and all that instructions used will be in the data memory. Both the instruction and data memory start with address 0. So if I say address 0 we need to tell whether it is address 0 of the instruction memory or address 0 of the data memory, right?

So there are two different memories and of course the data memory comprises of your ram 16 k plus your screen and also your keyboard as we have described in the previous module, right? So this is the understanding. So when we write an assembler first thing that we need to know is the memory model.

The memory model here is that instructions are separate, they start from 0 and go and then the data is separate and again it starts from 0 and we could have up to 16 k of data and of course after that we have 8 k of screen and then we have 1 word of keyboard, right? So this is how the whole thing is organised. With this as a framework let us quickly go and see, what is the role of the assembler?

Let me just revise. It is already a week now. So let me quickly revise. What is the output of the assembler? So, assembler takes an instruction and converts it into the binary. Now there are two types of instructions that we have seen already, one is the A instruction, another is the C instruction.

(Refer Slide Time: 13:43)


A - instruction: *@value* //Where *value* is either a non-negative decimal number //or a symbol referring to such number.

0..15

value (v = 0 or 1)

Binary: 0 V V V | V V V V | V V V V | V V V V

C - instruction: *dest = comp ; jump* // Either the dest or jump files may be 000 // If dest is zero, the "=" is omitted; // If jump is zero, or a symbol referring to such number.

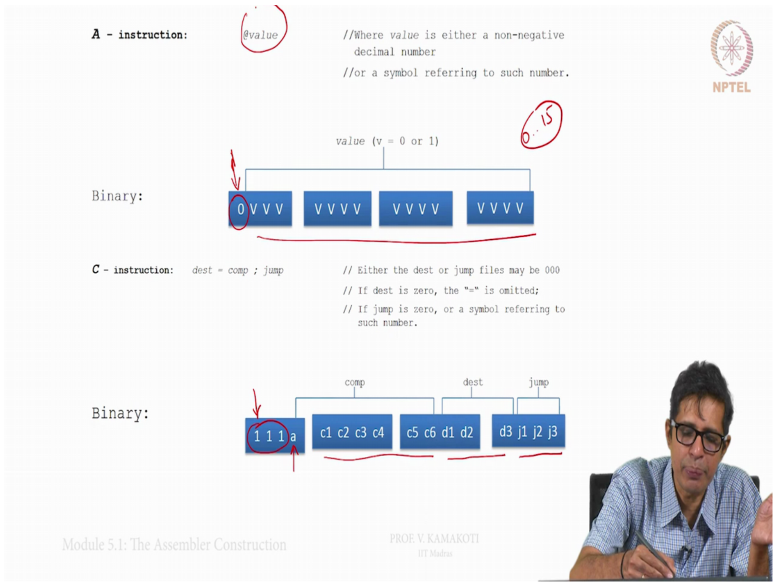
comp dest jump

Binary: 1 1 1 a | c1 c2 c3 c4 | c5 c6 d1 d2 | d3 j1 j2 j3

Module 5.1: The Assembler Construction PROF. V. KAMAKOTI IIT Madras

I will quickly revise what the A instruction is. It is again 16 bits, both the instructions are 16 bits. The most significant bit that you see here distinguishes whether it is an A instruction or a C instruction. If the most significant bit is 0 then it is an A instruction. If the three more significant bits are 1, 1, 1 then it is a C instruction. Now what will the A instruction do? What is the format of the A instruction? Then there will be a value which is a 15 bit value, right? So from the mnemonic point it is of this type at value.

(Refer Slide Time: 14:25)


A - instruction: *@value* //Where *value* is either a non-negative decimal number //or a symbol referring to such number.

0..15

value (v = 0 or 1)

Binary: 0 V V V | V V V V | V V V V | V V V V

C - instruction: *dest = comp ; jump* // Either the dest or jump files may be 000 // If dest is zero, the "=" is omitted; // If jump is zero, or a symbol referring to such number.

comp dest jump

Binary: 1 1 1 a | c1 c2 c3 c4 | c5 c6 d1 d2 | d3 j1 j2 j3

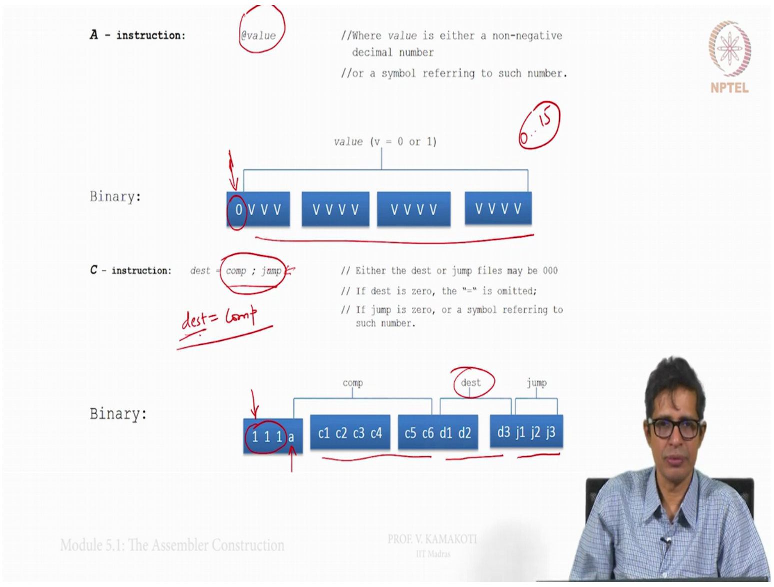
Module 5.1: The Assembler Construction PROF. V. KAMAKOTI IIT Madras

So when I say at value immediately this will be translated to having the first bit as 0 and then the values and non negative decimal number or it can also be a symbol, that 15 bit of that value should be put here following the 0 and that will be the 16 bit binary of this A instruction. The C instruction is of the form destination is equal to computations semicolon jump.

So either I will have computations semicolon jump or I could have destination is equal to some computation. So I will have two types of C instruction here, comp semicolon jump or destination equal to comp, okay. And the binary is of this form, the first three bits are 1, 1, 1. This A bit as such and the six bits basically are the computation. The A bit will tell where the source operant is and c 1 to c 6 will tell what we need to do with the source operant. We have already seen that in detail.

The destination, if at all it exist in the case of jump this destination will not exist. In the case of non jump if the destination exist then the d 1, d 2, d 3 bits as you see here will tell you what is the destination. And if it is of this jump type then this j 1, j 2, j 3 will tell you what is the type of jump.

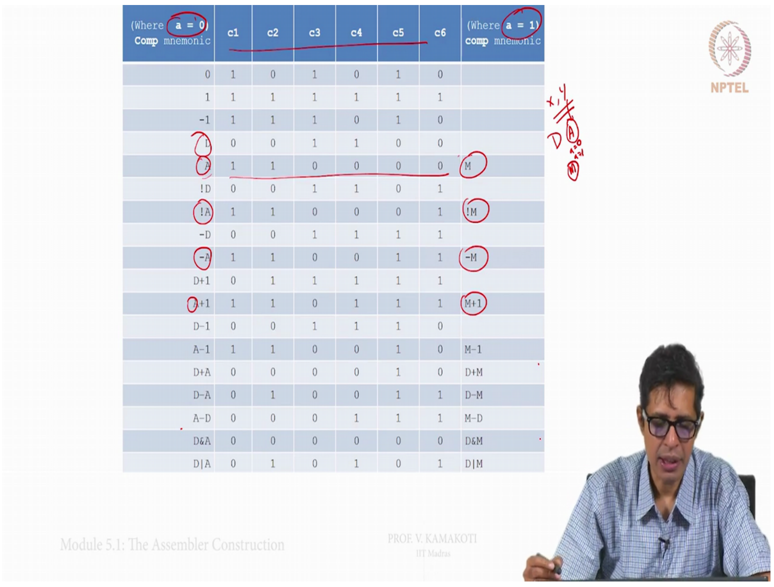
(Refer Slide Time: 16:17)



```
A - instruction:    @value     //Where value is either a non-negative
                                 decimal number
                               //or a symbol referring to such number.

                          value (v = 0 or 1)
Binary:     0 V V V   V V V V   V V V V   V V V V

C - instruction:    dest = comp ; jump    // Either the dest or jump files may be 000
                                          // If dest is zero, the "=" is omitted;
                                          // If jump is zero, or a symbol referring to
                                             such number.

                          comp              dest        jump
Binary:     1 1 1 a   c1 c2 c3 c4   c5 c6 d1 d2   d3 j1 j2 j3
```

So if I have an instruction dest is equal to comp, automatically the j 1, j 2, j 3 becomes 0. The dest will tell me what is d 1, d 2, d 3. The comp will tell me what is A, c 1, c 2 to c 6. If I have comp semicolon jump then this d 1, d 2, d 3 are 0 then of course the comp will tell me what is A and c 1 to c 6 and the jump will tell me what is j 1, j 2 and j 3, right? And we said that it will tell what they are still. So this is how it is. These tables are very clear.

(Refer Slide Time: 17:05)



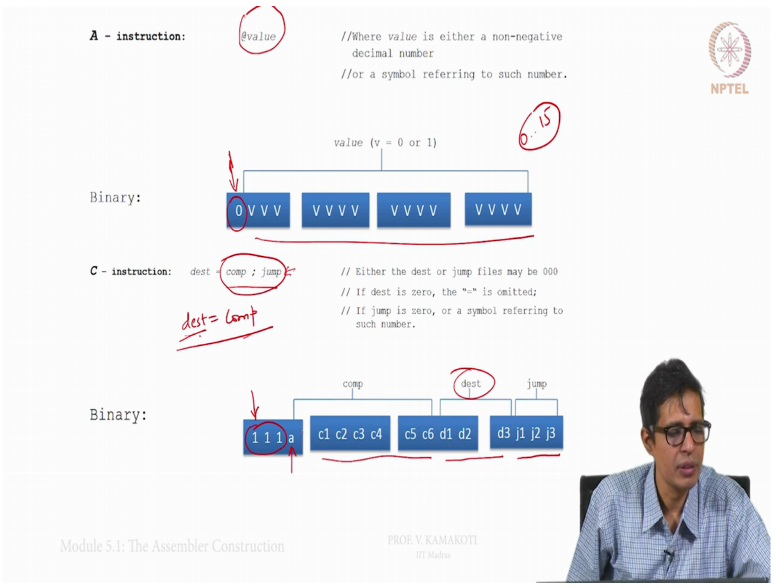
(Where a = 0) Comp mnemonic	c1	c2	c3	c4	c5	c6	(Where a = 1) comp mnemonic
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M

x, y
D
A
a=0
a=1
M
Module 5.1: The Assembler Construction
PROF. V. KAMAKOTI
IIT Madras

If your comp is 0 then I have to put A equal to 0 and 1 0, 1 0, 1 0. If your comp is 1 then I should put A equal to 0 and 1, 1, 1, 1. If your comp is minus 1 then I have to put A equal to 0 and this. Is your comp is D I have to do this. If your comp is A, I have to do this. If your comp is not D, I have to do this. Not A, minus D, minus A, D plus 1, A plus 1. If your comp is M then I have to put A equal to 1 and 1, 1, 0, 0, 0.

Not M minus M, M plus 1, M minus 1, D plus M, D minus M, M minus D, D and M, etc. So by parsing that comp, what do you mean by parsing? By actually reading that comp part alone of that instruction we can go and finalize what is A and what is c 1 to c 6 which are these bits of your C instruction, right?

(Refer Slide Time: 18:12)

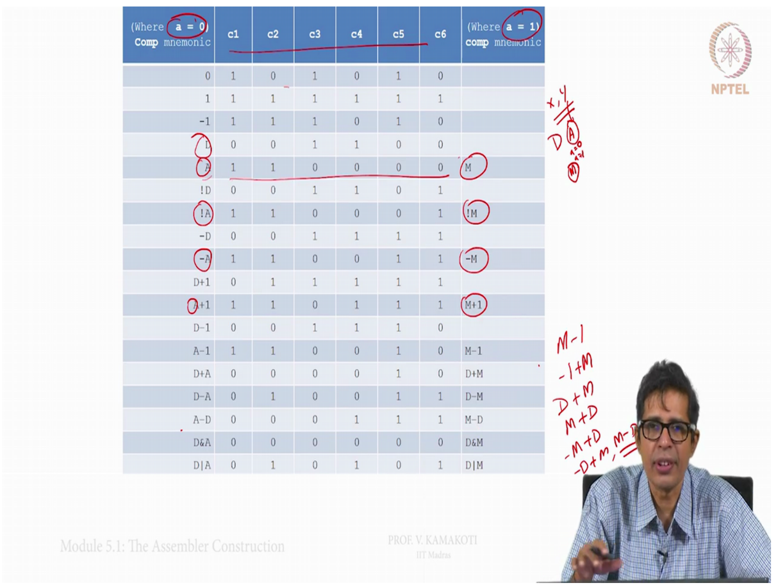


A - instruction: @value //Where *value* is either a non-negative decimal number //or a symbol referring to such number.

C - instruction: *dest = comp ; jump* // Either the dest or jump files may be 000 // If dest is zero, the "=" is omitted; // If jump is zero, or a symbol referring to such number.

Now the challenges are, if I want to write a very a robust assembler, I may write M minus 1. I can also write it as minus 1 plus M. I can write D plus M, I can write M plus D, I can write minus M plus D, minus D plus M, right? I can write M minus D. So all these things are possible, right?

(Refer Slide Time: 18:46)



(Where a = 0) Comp mnemonic	c1	c2	c3	c4	c5	c6	(Where a = 1) comp mnemonic
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M

x, y
D A
a=0
a=1
M
M-1
-1+M
D+M
M+D
-M+D
-D+M, M-D
NPTEL
Module 5.1: The Assembler Construction
PROF. V. KAMAKOTI
IIT Madras

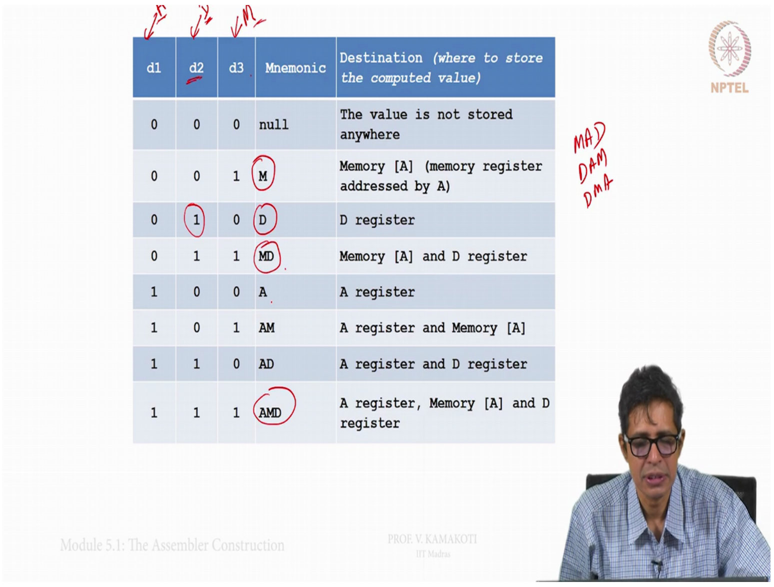
So these are all some of the possible combinations and similarly D minus A, A minus D, D and A. So, so many things are actually possible here. So that is the challenge in actually writing an assembler, right? The way the mnemonics are generated can be in different form, right? We cannot just say I want M minus 1. I could have also actually had even minus 1 plus M, right? I need not have D plus A.

I could have A plus D also and these things have to be basically catered to when you write the assembler. So these are challenges in writing the assembler and these challenges you will realise only if you write an assembler on your own. And you may not get an opportunity or you may not get time enough in your curriculum to write an assembler for your large C language, right, or for an instruction set as complex as Intel . So this is an opportunity.

So what this course essentially does is giving you an opportunity to write an assembler and understand all those small entry cases on a very small elegant machine. So that is the important thing and that is the importance of this exercise also, right? So now we understood how we fill up this A to c 6. We have already done that. I am just revising it so that I reinforce some of the concepts that we have done earlier. And this is about the d 1, d 2, d 3.

And when do we fill this d 1, d 2, d 3? When your destination exists. In the case of jump, destination does not exist. When your destination exists we fill it up, right? And this is how. So if I have just A, I have 0, 0. If I have D, I could have, if I have M then this is, but I could have A and D. I could have M A D. I could have D A M. So I could have M A D, D A M, D M A, right? Similarly M D D M, M A A M, A D D A. So all these possible combinations I could have here.

(Refer Slide Time: 21:12)

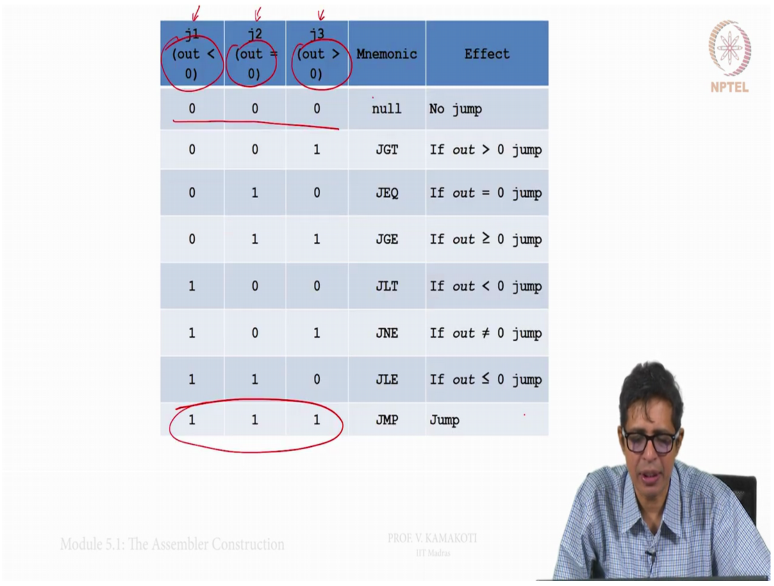


d1	d2	d3	Mnemonic	Destination *(where to store the computed value)*
0	0	0	null	The value is not stored anywhere
0	0	1	M	Memory [A] (memory register addressed by A)
0	1	0	D	D register
0	1	1	MD	Memory [A] and D register
1	0	0	A	A register
1	0	1	AM	A register and Memory [A]
1	1	0	AD	A register and D register
1	1	1	AMD	A register, Memory [A] and D register

MAD
DAM
DMA
NPTEL
Module 5.1: The Assembler Construction
PROF. V. KAMAKOTI
IIT Madras

And similarly for the jump, jump is very straight forward. So if I have null then I have to put 0, if I have J G T, J E Q, J G E, these are all the mnemonics and based on that I can basically fill up this j 1, j 2, j 3 part of this stuff, right?

(Refer Slide Time: 21:36)

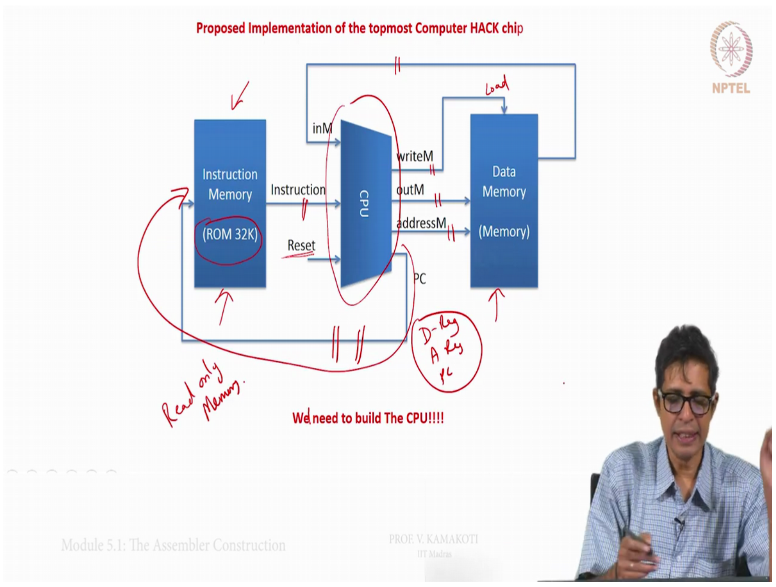


j1 (out < 0)	j2 (out = 0)	j3 (out > 0)	Mnemonic	Effect
0	0	0	null	No jump
0	0	1	JGT	If *out* > 0 jump
0	1	0	JEQ	If *out* = 0 jump
0	1	1	JGE	If *out* ≥ 0 jump
1	0	0	JLT	If *out* < 0 jump
1	0	1	JNE	If *out* ≠ 0 jump
1	1	0	JLE	If *out* ≤ 0 jump
1	1	1	JMP	Jump

NPTEL
Module 5.1: The Assembler Construction
PROF. V. KAMAKOTI
IIT Madras

So these three tables are extremely important for us to understand from the mnemonic how do we actually get to this 16 bit binary, right? So we will be using these three tables extensively as a part of this assembler development, okay. Now this is the machine model. We have already seen this machine model.

(Refer Slide Time: 22:20)



Proposed Implementation of the topmost Computer HACK chip

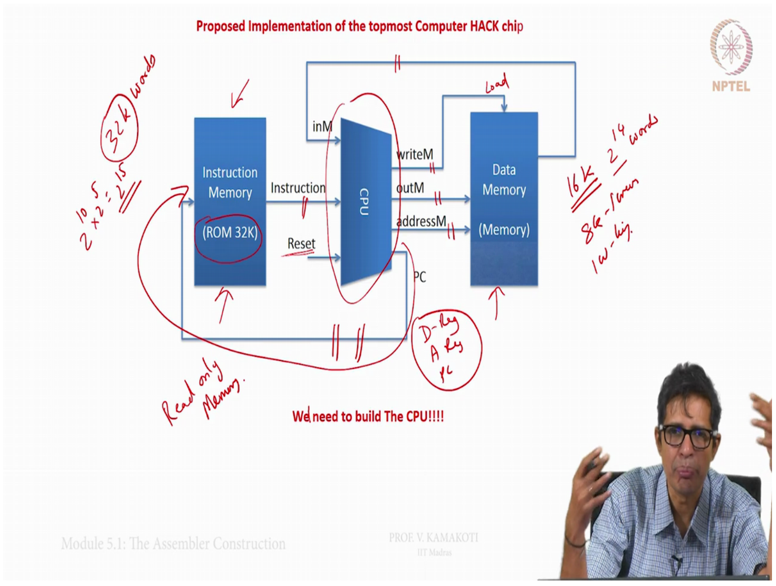
We need to build The CPU!!!!

Module 5.1: The Assembler Construction

PROF. V. KAMAKOTI
IIT Madras

So the instructions are going to be in this ROM 32 k. So we have 32 kilo words of instructions. So 32 k instructions I can put or just 32 k, 2 power 10 into 2 power 5, so 2 power 15 (ins) instructions I can put. Similarly your data memory has 16 k of this 2 power 14 words of data you could have. Every data is 16 bits in this machinery and of course 8 k of screen and 1 word of keyboard. So this is the memory model and so always my instructions will start with 0 and always my data also will start with 0 and this is how we build the assembler.

(Refer Slide Time: 23:05)



So this is the basic assumption that we have when we want to build the assembler. So now we will go to the next module where we will be describing what are the challenges in building that assembler. So some of these small integrities of building the assembler and at least before we end that module we will see what is the two pass assembler. Thank you.