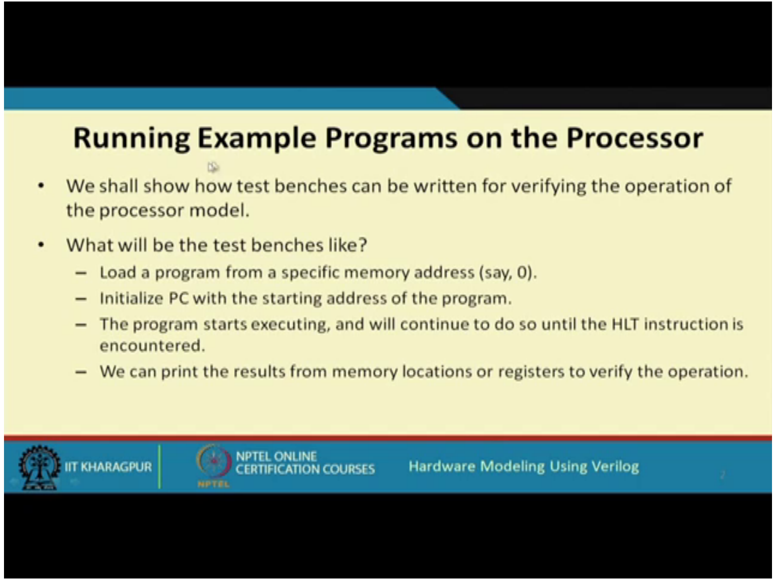
Hardware Modeling using Verilog
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 41
Verilog Modeling of the Processor (Part 2)

So, in the verilog implementation of the processor that we had discussed during our last lecture, you recall we had considered only a small subset of instructions from the MIPS 32 instructions set.

Now, these subsets of instructions have been chosen quite thoughtfully in the sense that there are many programs, which you can possibly right even using this small instruction set. Now in this lecture I shall be showing you 3 example test benches, which actually search the purpose of executing small programs in the MIPS 32 assembly language. Let us go through them and see how it works.

(Refer Slide Time: 01:20)

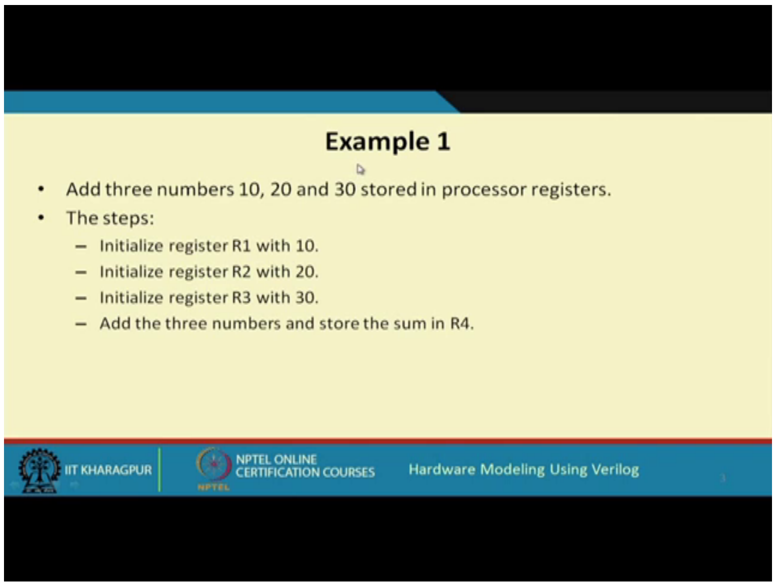


So, the title of a lecturer is verilog modeling of the processor the second part. Now here we shall actual be demonstrating by showing you how we can run some programs on the processor. Program means programs written in the MIPS 32 assembly language, we are not talking a verilog here. We have already designed a processor a CPU, now our next task is to write a program that will be running on that CPU right.

Now, the test benches that will be writing there will be having sudden kind of structure. The structure will be as follows the example programs that we shall show will be starting from a specific memory location. Well, we shall be assuming it is starting from address 0. And before we start execution we will be initializing PC with 0; that means, the first instruction that will be fetched will be from address 0, which is the first instruction.

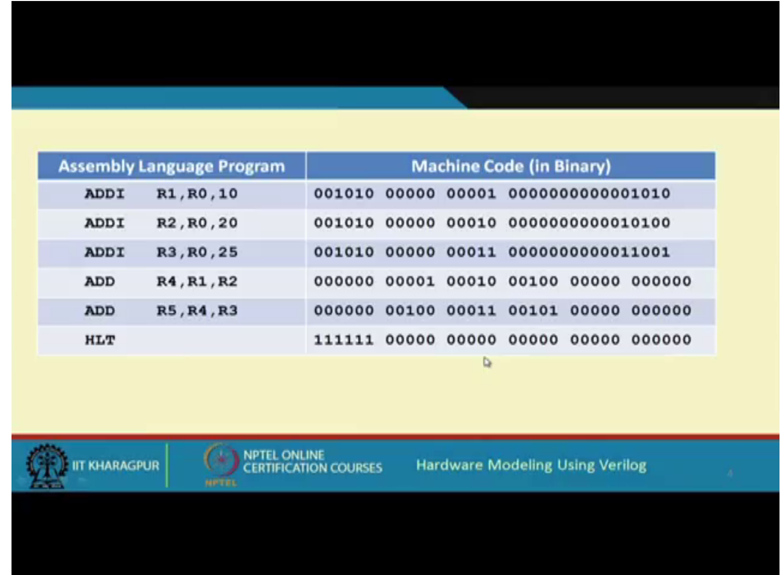
And after we have started it then clock 1 and clock 2 are automatically coming. So, instructions will be fetched from memory one after the other automatically, you do not have to do anything else in the test bench. Only the initial thing we have to we have to load the instructions in the memory set program counted to 0 and that is it right. And at the end we can print the result wherever the results are available. There may be in some memory locations or there may be in some registers we shall see.

(Refer Slide Time: 03:01)



Now, let us look at 3 examples one after the other. The first example is a very simple one, this says that we add 3 numbers 10 20 and 30 in decimal they are all stored in processor registers. So, how you write the program? That we initialize register R 1 with 10 register R 2 with 20 R 3 with 30 then add them up and the result is store in R 3 or R 5 in R 4 or R 5. In fact, we are storing in R 5 not R 4 first we are storing in R 4, and then we are storing it in R 5. Yeah, first 2 numbers we are adding storing in R 4 the third number also adding we are storing in R 5.

(Refer Slide Time: 03:58)



Assembly Language Program	Machine Code (in Binary)
ADDI R1,R0,10	001010 00000 00001 0000000000001010
ADDI R2,R0,20	001010 00000 00010 0000000000010100
ADDI R3,R0,25	001010 00000 00011 0000000000011001
ADD R4,R1,R2	000000 00001 00010 00100 00000 000000
ADD R5,R4,R3	000000 00100 00011 00101 00000 000000
HLT	111111 00000 00000 00000 00000 000000

Let us first look at the program how the program looks like. The program will look like this you forget this for the time being look at this program. What does the first instruction do? This add immediate R 0 and 10 result goes to R 1 which is as good as saying that we are initializing R 1 with the value 10. 7 instruction same thing R 0 is always 0 0 and 20 are added in R 2 R 2 equal to 20, similarly here R 3 equal to 25.

Then in the 4th instruction we are adding R 1 and R 2 which are having values 10 and 20 and storing it in R 4. So, R 4 will be getting the value 30 then in the last instruction this 30 is added to R 3, R 3 is 25. So, the result is 55, 55 should go to R 5 and then a halt.

Now, side by side we are showing the machine code because ultimately what we have to load in memory, are the machine code of the instructions right. Because the processor will be fetching the instruction those are those machine codes ok.

Let us see how you have done that. Just look at the first instruction. This add immediate you consult the opcode table the opcode for add immediate is 0 0 1 0 1 0 this 6 bit is the opcode I have shown some gapes to indicate clearly. Then the 2 source registers. Sources here there is only one source register R 0 sources R 0 it is all 0 indicating R 0, then the destination. Destinations is R 1 0 0 0 0 1 it is 1. Then 16 bit the immediate data here it is 10. So, in binary 10 is 1 0 1 0 you see this is 10 in 16 bits.

Second instruction similarly this is add immediate, this is R 0. This is to 1 0 which is R 2 and this is 20 you see 1 0 1 0 0 in binary mean 16 and 4 20. Similarly this one is R 3 and this one is 25, then there are register to register I mean instructions to of them at R 1 and R 2 are added to R 4. So, this is the opcode for add all 0s this is R 1 1, R 2 2, R 4 1 0 0 4. And as I said for these instruction the last bits will be then you (Refer Time: 06:48) some other purposes will be keeping them 0s, similarly here R 3 and R 4 this is R 4 and R 3 this is 4 this is 3 the target is 5. And halt this is the opcode all one and the remaining bits are not used to set it all to 0.

So, actually when you will be loading them in memory you can actually group them 4 bits at a time and find the hexadecimal code like the first one it would be how much 0 0 1 0 is 2 1 0 0 0 is 8 0 0 0 0 is 0 and 0 0 0 1 is 1. So, 2 8 0 1 in, and here it will be 0 0 0 a. In this way you can calculate the opcode of all the instructions in hexadecimal.

So, when you actually load the instructions in memory we shall be showing the hexadecimal codes, that you can directly get from this table or you can use the instruction format to create the instruction in coding. And find out for the opp code is coming to for the 32 bit instruction is coming to. So, you can convert it to hexadecimal after that ok.

So, after we have done this there is one point that I would like to just mention here. That I mentioned when we are discussed pipelining that instruction pipelining that there is something called hazard, because of data dependencies You can have something called data hazard.

(Refer Slide Time: 08:28)

Like suppose let us say let us take an example suppose I have an instruction add R 1, R 2 R 3. The next instruction is add let us say R 5, R 1, R 8.

So, the first instruction is producing the result in R 1, which is used by the second instruction. So, if you look at the instruction execution cycle. Instruction fetch instruction decode execution MEM and WB. You see this R 1 is getting stored only during the WB stage. And in the next instruction is trying to read the registers in the ID phase, so obviously it will be reading a wrong value

So, it has to wait at least for 2 cycles before it can read right this is this is one thing. So, if you really have a pair of instructions like this and the hardware is not automatically taking care of this, then you may have to insert some dummy instructions in between. So, what the dummy instructions will do? It will simply serve to delay this instruction this will be delayed by 2 steps or one step whatever So that you can. So, that by the time this instruction reaches ID this right back should be complete.

Now, another thing you remember that we are not using a single clock we are using clock 1 clock 2 alternately right in the stages. So, the first instructions is fetched here. Instruction fetches then only during clock 1, the second instruction is fetched in clock 1, third instruction is fetched in clock 1. So, if you delay it by one instruction that will serve your purpose. So, just one dummy instruction if you include that should be ok.

Now, in this example we do not have a problem here because this add immediate is calculating R 3, and this add is not using R 3, but here I have a problem. You are computing R 4 this R 4 is being used here right ok.

(Refer Slide Time: 11:03)

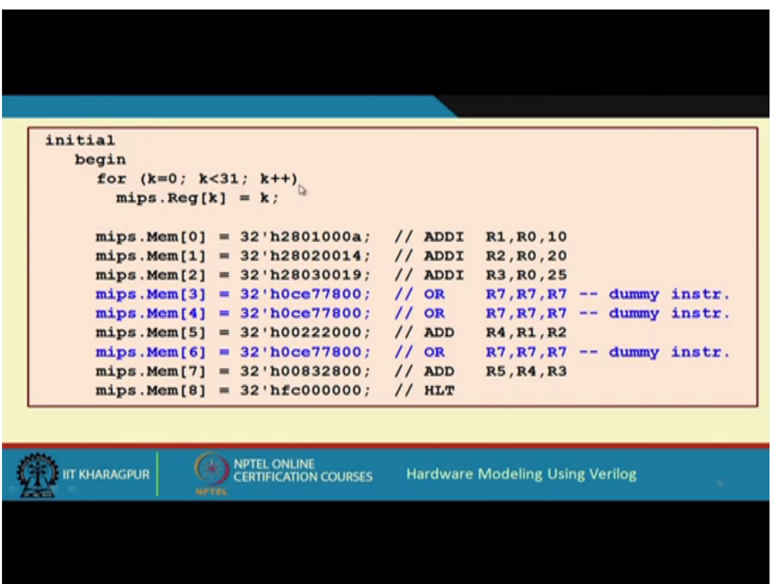
TEST BENCH

```
module test_mips32;

  reg clk1, clk2;
  integer k;

  pipe_MIPS32 mips (clk1, clk2);

  initial
    begin
      clk1 = 0; clk2 = 0;
      repeat (20)                  // Generating two-phase clock
        begin
          #5 clk1 = 1;  #5 clk1 = 0;
          #5 clk2 = 1;  #5 clk2 = 0;
        end
    end
```

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

So, this we shall see in the test bench how we have handled this. Let us now come to the test bench, this is the first part of the test bench where we have instantiated our processor we call it MIPS only 2 parameters declared as reg and available k. Here we are generating the 2 phase clock this we have also seen earlier how we can generate 2 phase clock we have the same code. Well 20 clock cycle it is sufficient for this program. So, we have repeated it for 20.

(Refer Slide Time: 11:39)

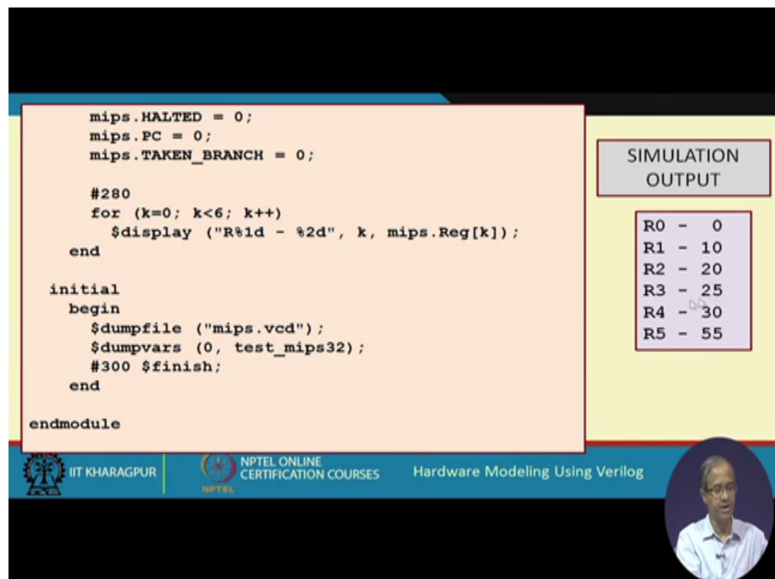


Then we load our program in memory before that we initialize the registers. K 0 to 30 1 reg k equal to k, which means reg 0 is 0 register 1 is one register 2 is 2 register 3 is 3 just some initialization.

Now, here you see we have inserted some dummy instructions. Now as I said these dummy instructions are not required you need only this, but actually here we have this inserted 2 dummy instructions. Now what is the dummy instruction we have used? We used an instruction like or R 7, R 7, R 7. What does that mean? You do a logical or with R 7 and R 7 we have initialize R 7 with 7 right. So, you do or 7 with 7 result will be 7 itself store 7 back to R 7. So, result will not change, but it will consume one clock cycle ok.

So, like this wherever you feel there is a doubt there is a hazard data dependency you can insert some dummy instruction like this. So, here I just shown. So, here we have inserted 2 dummy instructions and one dummy instruction here. The other instructions are just as we had shown earlier. And MIPS MEM 0 we are loading the hex code of the first instruction MEM one address one second instruction 2.

(Refer Slide Time: 13:18)

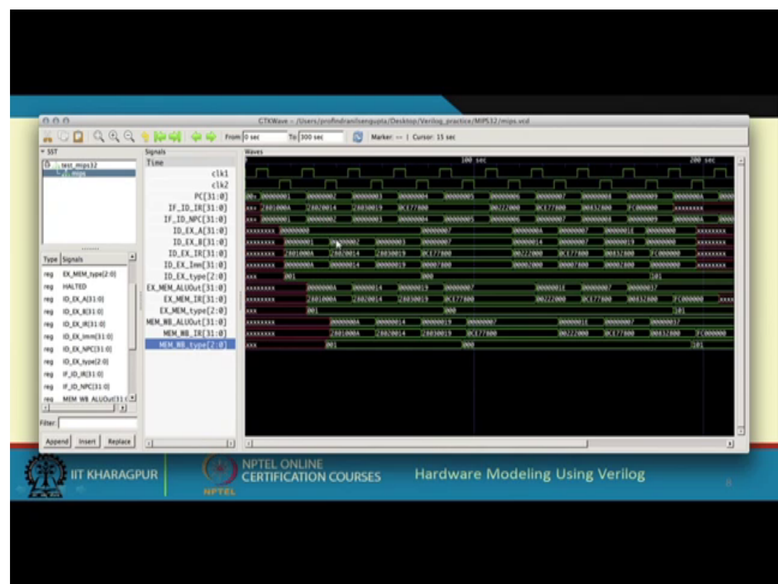


In this way there are 9 memory locations right. And before we actually start execution, we reset the halted flip flop to 0 taken branch to 0 and set PC to 0.

So, after this when the clock starts clock will start after a delay of 5 the first one. So, the instruction you start executing well. And after sufficient delay we are displaying the values of R 1 not R 1 k equal to 0 to 5 R 0, R 1, R 2, R 3, R 4 and R 5.

So, we are displaying r register number dash the value in the register, let us see. So, if you run this the simulation output will come like this, as part of this loop for loop. So, R 0 is 0 R 1 is 10, 20, 25 which we have already loaded R 4 10 and 20 was added R 4 is 30 and R 5 30 and 25 is added it is 55. So, the results are correct.

(Refer Slide Time: 14:39)



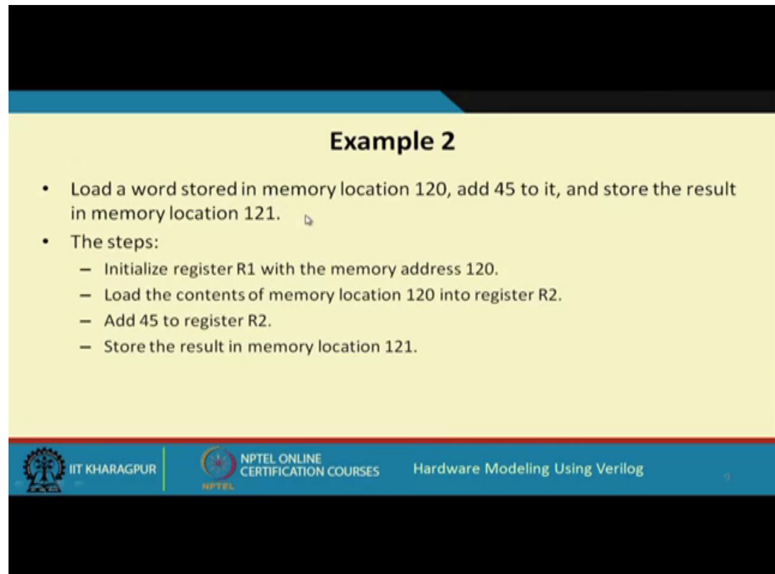
Now, in a similar way you have also captured the wave form in a file MIPS dot vcd. So, if you see the timing diagram the fonts quite small I do not know whether you are able to see it clearly or not, but you can see we have listed all the relevant signals the clocks PC then I F ID IR M PC ID EX A ID EX B IR immediate and so on.

So, you can see that how things are going on this is instruction type EX MEM it is 0 0 1 initially we had the add immediate instruction this is of RN type. You look at the ALU out, first operation was add immediate R 0 with 10 you see. ALU out contains 10 after adding 0 and 10 0 0 ALU is 10.

Next one was 20 which is one for in hexadecimal minutes 23rd one as 25 you see it was one 9 25. Then there is some dummy or instructions forget it 7 then first 2 numbers were added 10 and 20 30 you see one e is 30 then again there is a dummy or instruction 7, then lastly the last 2 are added 55 with 3 7 hexadecimal. So, this way you can track all the other signals in clocks systematically they are going on right.

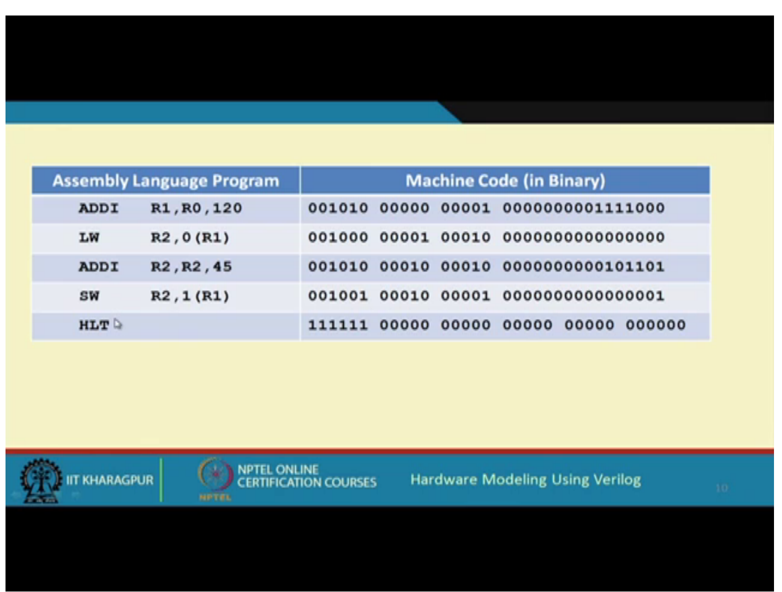
So, at the end you see when this here halted signal is not shown, but at the end you will see after everything is done the halted signal will be activated and instruction you stop you see here it had it started to become undefined that mean stopped here we. So, you see that with the small example that we are indeed able to write a program and run the program on our processor which were designed in verilog, let us taken another example.

(Refer Slide Time: 16:38)



Well in this example we are using a memory location memory. So, what we are doing? We are loading a word which is already stored in memory location 120. Then we are adding 45 to it and storing the result back in memory location 121. So, the steps are very simple which are illustrating with the program I am showing the program straight away.

(Refer Slide Time: 17:06)



So, the program is like this. Because we are storing the number in memory address 120 we are initializing a register with that address 120. This add immediate R 1 R 0 1 20

means we are loading 120 in R 1 right 0 and 120 are added store in R 1. Then you are loading from that address R 1 means 0 plus R 1 this R 1 contains 120.

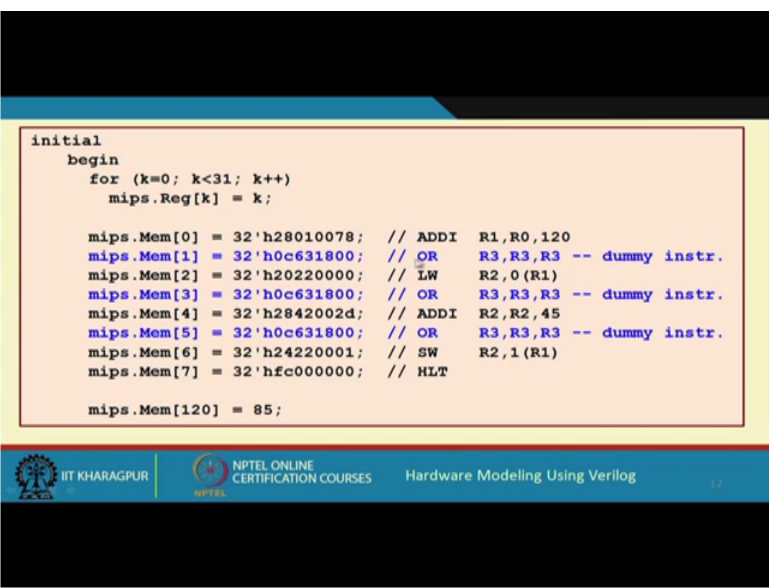
So, that memory will be fetched and the data will be stored in R 2. Then we are adding that number with 45 result in R 2 and we are storing R 2 in one R 1 which means R 1 contains 121 20 plus 1; that means, 121. So, instruction encoding is very similar this add immediate I have already showed.

let us see this l w this l w opcode is this 0 0 1 0 0 0. Here source is R 1 you see this is one R 1 destination is R 2 this is 2, and offset is 0. You see all 0, this add immediate this is add immediate R 2, R 2 this is 2 this is 2 45 this is 45, store this is store this R 1 R 2 and 1 this is R 2 this is R 1. And 1 this is 1 halt it is just halt.

So, here you see that there are lot of data dependencies. This add immediate is generating R 1 which is used in the next instruction. Here you use here you are generating R 2 which is again used in next instruction. This add immediate also is generating a new value of R 2 which is used in the next instruction. There are lot of data dependencies.

So, let us look at the test bench again. The first part of the test bench is identical, same we are instantiated it and the clock generation logic. Then this is the program.

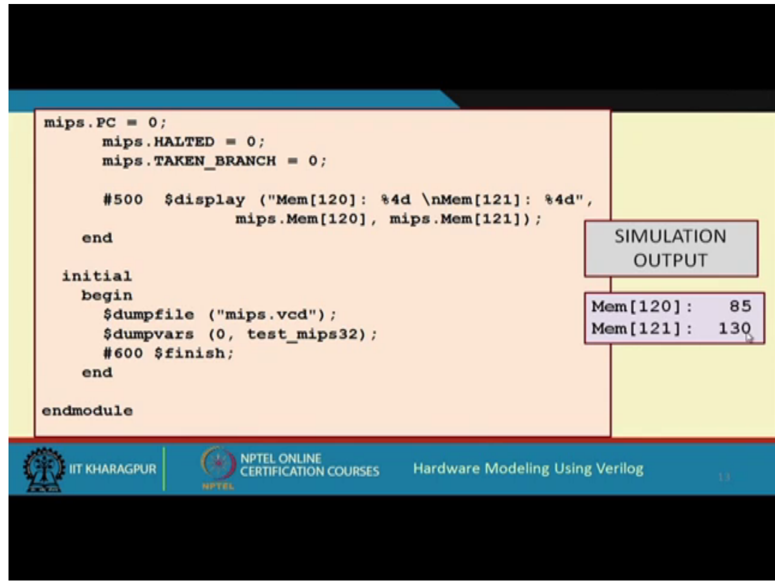
(Refer Slide Time: 19:15)



You see I told that there is a data dependency between all these pairs of instructions. So, we have inserted a dummy instruction in between every pair of them. So, here the

dummy instruction we have used the or R 3, R 3, R 3. So, this of course, you can actually verify. So, I will give it an exercise for you that or R 3, R 3, R 3 is actually this is your c 6 3 1 8 0 0 and this will be our total code. So, our useful code is 5 instructions and we are adding 3 dummy instructions it becomes 8. So, it is loaded from memory location 0 up to memory location 7. And you said that some data is already loaded in memory location 120 let us load a value 85 there 85 is already stored in 120 memory 120.

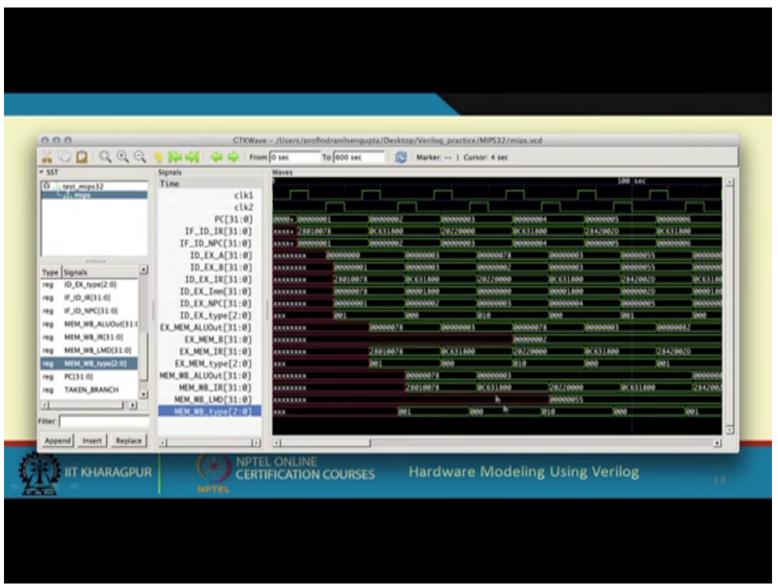
(Refer Slide Time: 20:14)



Then we again initialize the PC to 0 halted to 0 and taken branch to 0, and at the end after sufficient delay we are displaying the value of memory location 120 and 121 these 2 things were pending ok.

Let us see the output this is how the output is coming MEM 120 column 85 MEM 121 130. So, actually 45 is being added to this you see 85 plus 45 is actually 130 right.

(Refer Slide Time: 20:51)

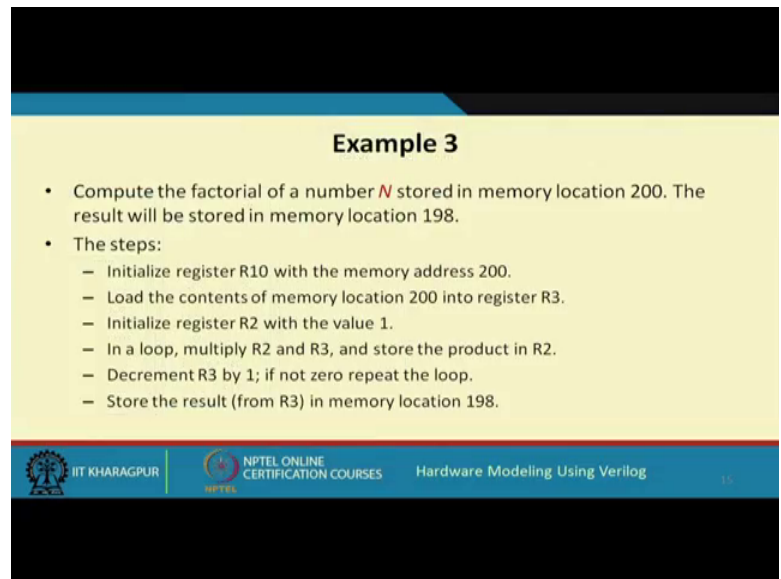


So, here also the result is correct. So, again we showed the way from here the timing diagram.

So, here you can again analyze and say how it works. Just look at this LMD for load instruction this LMD is used. We see when the load instruction executed this LMD becomes 55, 55 hexadecimal is 85. So, actually the data is getting loaded here. Then it will be added in that MIPS stores. So, so the whole thing you cannot see in the screen here I will be showing a part of it right.

Similarly, you can see the PC the instruction solving fetch address 1, 2, 3, 4, 5, 6 sequentially. The IR you see the opcodes of the instruction first opcode second opcode third opcode like this you can actually analyze what is going on in the pipeline what verilog code you have written you can actually see it step by step. And here I strongly suggest you look into this timing diagram very carefully you run your own version and see exactly what is happening step by step, then it will be very clear to you right.

(Refer Slide Time: 22:04)



Now, let us come to a slightly more complex program, which involves a loop. Loop means iteration. So, we are trying to compute the factorial of a number n. So, what you do suppose the factorial of a number n I will give a value of n let say n equal to 5. So, I take a variable initialize to one and multiply it with 5 decrement 5 4 is it 0 or no multiply with 4 again decrement 3 multiply with 3, 2 1, and when it reaches 0 I will stop. And whatever is this register contained at the end that is the value of the factorial.

So, actually we have done this. We have assumed that the number n is stored in memory location to 100. And the factorial of the number let us say we want to store it in memory location 198.

(Refer Slide Time: 23:14)

Assembly Language Program			Machine Code (in Binary)
	ADDI	R10,R0,200	001010 00000 01010 0000000011001000
	ADDI	R2,R0,1	001010 00000 00010 0000000000000001
	LW	R3,0(R10)	001000 01010 00011 0000000000000000
Loop:	MUL	R2,R2,R3	000101 00010 00011 00010 00000 000000
	SUBI	R3,R3,1	001011 00011 00011 0000000000000001
	BNEQZ	R3,Loop	001101 00011 00000 1111111111111101
	SW	R2,-2(R10)	001001 00011 01010 1111111111111110
	HLT		111111 00000 00000 00000 00000 000000

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES Hardware Modeling Using Verilog

So, again we have done it step by step fashion, but I will be explaining with the program, it is here. This is our program. See what we have done? In the first instruction we are initializing register R 10 with 200, where the value of n is stored. And this R 2 is initialized to the value one, where our factorial will be finally, stored we will be multiplying with R 2 ok.

Now, we are loading the number R 10 contains the address 200 0 R 10 loaded store it in R 3. So, R 3 contains the number n, this is a loop. So, this 3 instructions we repeat what we do multiply R 2 and R 3 store the result in R 2.

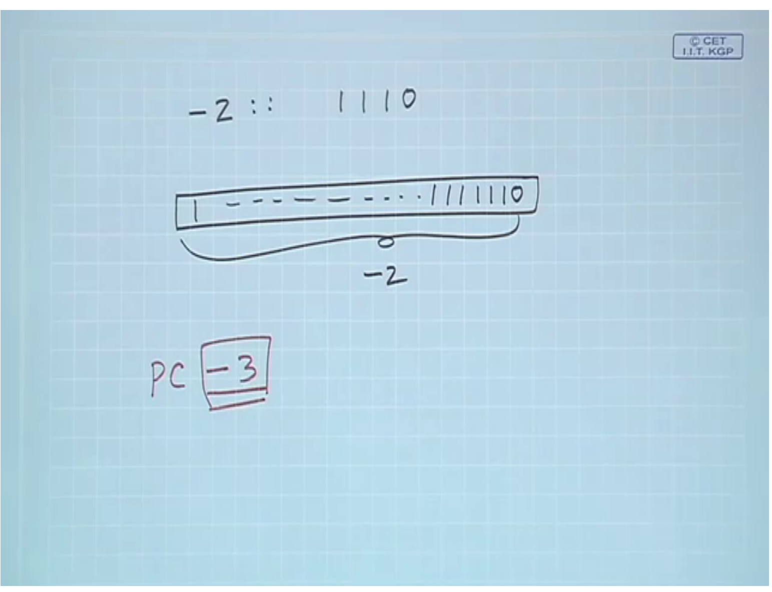
So, if let us say this R 3 was 5 we multiply 5 with one store it R 2 R 2 becomes 5, subtract 1 from R 3 and store it back in R 3. So now, R 3 becomes 4. You check not equal to 0 is R 3 is 0 or not 0 if it is not equal to 0 go back to loop. So, this 4 not equal to 0 go back to loop multiply 4 with 5 20 stored in R 2 subtract R 1 from R 3 again it becomes 3 not equal to 0 again go back to loop. So, 20 multiply by 3 60. So, like this it will go on right here multiply number one by one, and as soon as R 3 become 0 you will be coming out of the loop and you will have to store the result in 198 memory location right.

Now, R 10 contains memory location they are just 200. So, we write like this store R 2 minus 2 R 10, which means R 10 minus 2 in this address we want to store it. So, you can see the instruction encoding it is very similar, but only one thing I want to tell you say add immediate, we have already seen how to encode load also I have said let us see this

load you see once more R 10 this is 10 source 3 this is your R 3. Right now multiply similar this is multiplying R 2 R 3 R 2 R 2 R 3 R 2 subtract immediate R 3 R 3 and this is 1 ok.

Let us look at the store instruction in particular I would like to concentrate with the offset minus 2. You see here we have specified the offset as minus 2, and in the instruction encoding how offset is 16 bits.

(Refer Slide Time: 26:15)



So, how much is minus 2 in 16 bits? Well for those of you who are familiar with 2s complement number you may be knowing that in 4 bits representation minus 2 can be represented as 1 1 1 0 this is minus 2

Now, if I want to extend it to 16 bits, I just mentioned I told you the rule for sign extension. So, whatever number you have you take the sign it is one, you replicate want as many times you want. The value of this number will still be minus 2 right. So, we have done the same thing you say minus 2 is this all ones and the last as 0.

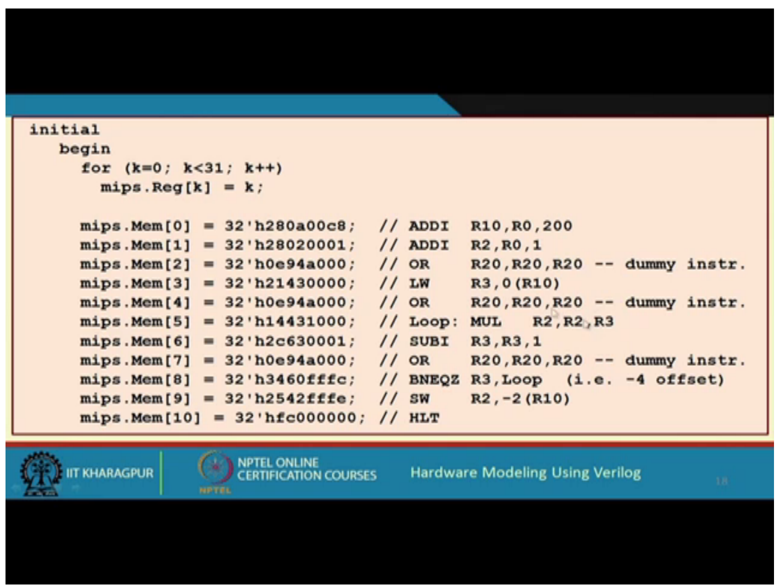
Similarly, let us look at the branch instruction. BNEQZ as an opcode of this R 3 this is 3 it does not require the second operand. So, it is second register it is 0. And loop is here. So, what should be stored here, let us see. Here the memory address is what if this is 0 0 1 2 3 loop is 3 right, 4 5 this is 5. So, when I am executing this already PC has been implemented PC contains 6. So, from 6 I have to go back to 3.

So, 6 to 3 means it should be minus 3. So, so means actually what I have to do whatever is the value of the program counter I have to subtract 3 from it. So, this minus 3 will be my offset. So, in this branch instruction the loop here whatever I wrote this is the 2s compliment of minus 3.

So, the PC is pointing here minus 1, minus 2, minus 3 it will be jumping here after this right fine.

Let us now come to the test bench. The first part is same, then these are the instructions. You say we have inserted some dummy instructions again in between one here one here one here.

(Refer Slide Time: 28:22)



So, wherever there is a dependency we have inserted a dummy you see here of course, here it was not really required R 2 here R 3 and R 3 we have inserted. Here R 3 and R 3 here also we inserted and the opcodes have been coded. So, here we need 11 instructions in total. So, as usually we have initialized register to something, but there is not only required, but we have done it.

(Refer Slide Time: 29:04)

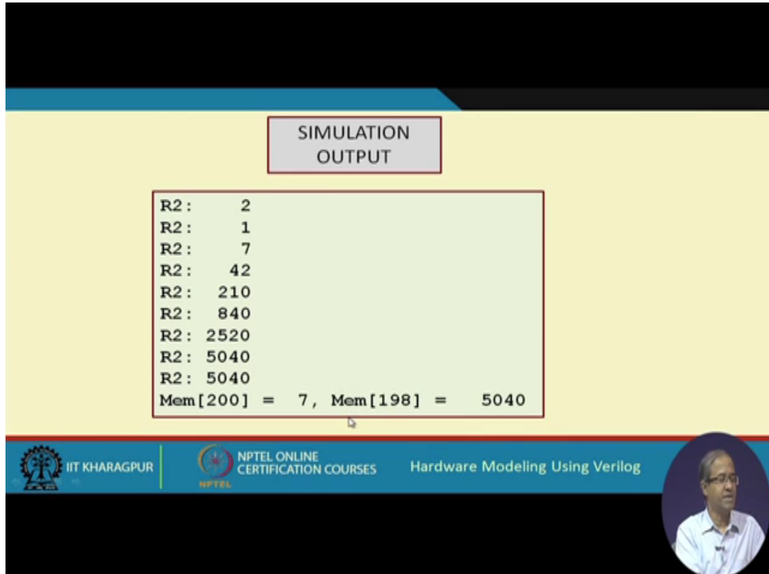
```
    mips.Mem[200] = 7;     // Find factorial of 7

    mips.PC = 0;
    mips.HALTED = 0;
    mips.TAKEN_BRANCH = 0;

    #2000  $display ("Mem[200] = %2d, Mem[198] = %6d",
                                  mips.Mem[200], mips.Mem[198]);
  end

 initial
  begin
    $dumpfile ("mips.vcd");
    $dumpvars (0, test_mips32);
    $monitor ("R2: %4d", mips.Reg[2]);
    #3000 $finish;
  end
endmodule
```

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

And suppose we want to calculate the factorial of 7. So, in memory location 200 we are storing 7.

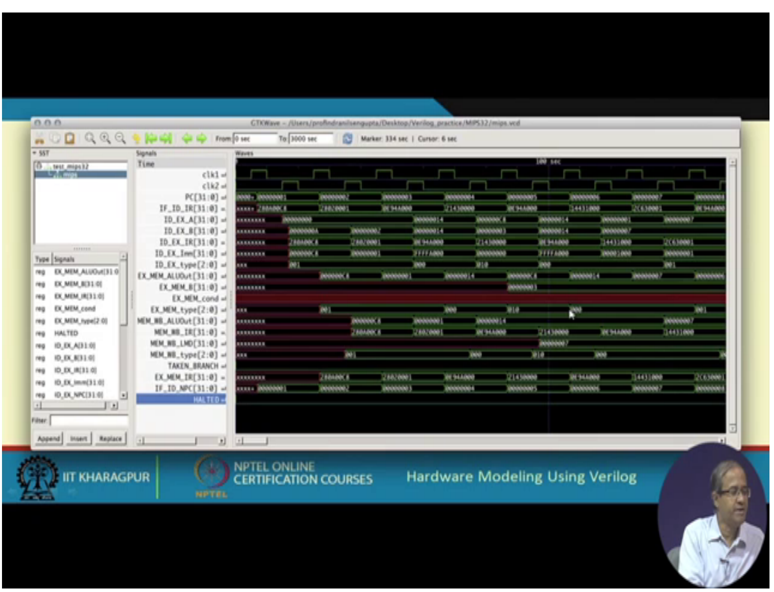
So, as usual we have initialized PC to 0 halted to 0 taken branch to 0. And at the end after sufficient delay we are displaying the contents of memory location 200 and memory location 198. And we have also monitored so that whenever it changes it will be printed the value of R 2 because you recall R 2 contains the value of the product you are multiplying it into R 2 right.

(Refer Slide Time: 29:43)



So, if you run it the simulation output comes like this you see, at the end you see MEM 200 is 7 MEM 198 5 4 2 0 you can verify factorial 7 is actually 5 0 4 0. And R 2 well this initially you are initializing all register k to k initially it was 2 after that one multiply with 7 multiply with 6 multiply with 5 4 3 2 you get final 5 4 0 like this.

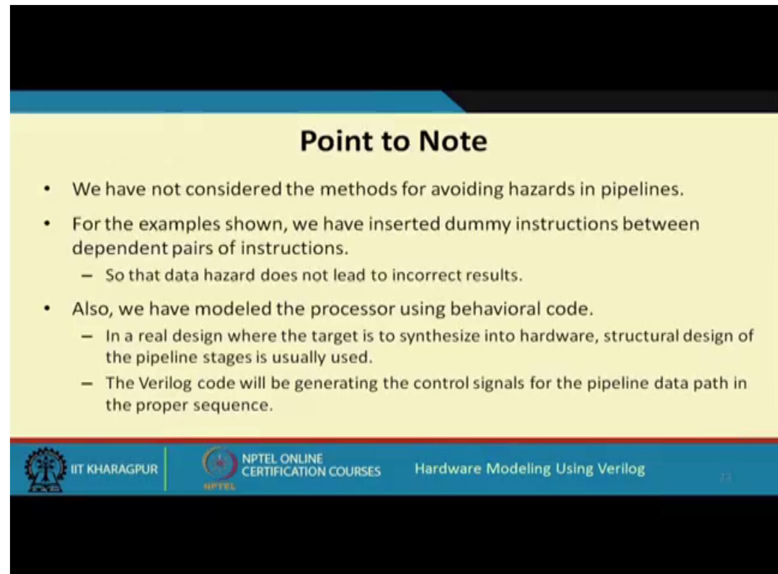
(Refer Slide Time: 30:18)



And in a similar way If you look at the timing diagrams you can see step by step what is happening.

Now, I will leave it as an exercise for you because this is quite complex and the font size it is also not quite large. So, I am not sure whether you are able to see it very clearly, but actually you can trace for the this is the first part of the timing diagram, this is the second part of the timing diagram. Still you can say execution is going on it is not finished. Well, when finished will be done and halted is still 0 halted will become 1, after that ok.

(Refer Slide Time: 30:55)



So, just one thing I would like say here see we have taken 3 very simple examples, but as I said the instruction set or the instruction that we have selected they are powerful enough. Well, if you want let us say you can try and write a program to compute the gcd of 2 numbers, that is quite possibly this instruction set. You can try and write a program to sort a list of n numbers, using any sorting algorithm like bubble sort or insertion sort. It is possible, even with the small instruction set.

So, what I mean to say is that, this instruction set even though it is very small we have chosen the some of the essential instruction which are required for condition checking arithmetic looping and so on. So, that you can write meaningful programs using this. Well of course, if you want to extend say by adding more instructions you can always make changes to a verilog code you can add more instructions ok.

Now, there are a few points we would like to mention here. That we talked about hazards, but in the design that you are presented we have not considered any method for avoiding hazards in pipeline. What I have assumed is that if there are hazards we are inserting dummy instructions. So, it is the users responsibility to do that.

Well there are many instances where this is actually done, but not the user the compiler. Suppose there is a c compiler for the MIPS machine someone has written a code in c and when the c compiler is generating machine code, it automatically checks whether there are any dependencies between consecutive instructions. If So, it will first try to move

some instructions around if possible if not it will insert automatically such dummy instructions So that the final code will be executing correctly on the pipeline right

So, this is what I mentioned for the examples where we inserted dummy instructions such that results obtained are correct. And another important thing is that the processor model that we have considered is purely behavioral model. So, we have done or we have written the code using if then else kind of statements, which is quite similar to say program to which we write in high level language. But in an actual real design where you are actually seriously wanting to synthesize your design, it is always better to go for structural code. All the pipeline elements like the registers ALU's register banks they will all be part of your data path, we have already seen earlier how to separate the data path and control path.

So, once you have the date path components you will also have to identify the control signals to activate them. Now inside your main pipeline verilog code you will be just activating the control signals one by one. That is what will be there in the code right. Just like the control path or the controller you are designing those earlier examples ok.

So, this is what is mentioned here. So, with this we come to the end of this lecture In fact, this is the last lecture of the series. So, we have seen various features and structures of the verilog language. We have looked at a number of examples as well. So, we hope with this background you will be able to create some more meaningful and serious designs, which will be helpful in the domain that you are working in.

Thank you.