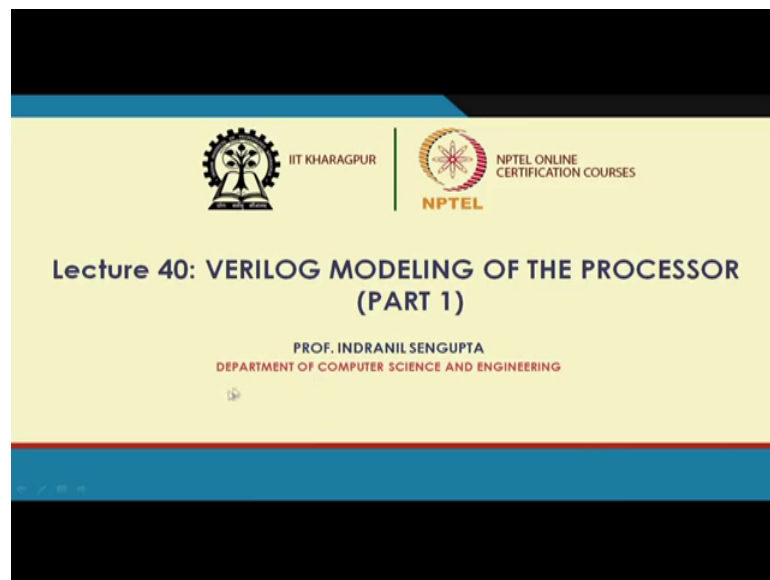


**Hardware Modeling using Verilog**  
**Prof. Indranil Sengupta**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture - 40**  
**Verilog Modeling of the Processor (Part 1)**

In the last few lectures, if you recall we were discussing how to implement a particular processor. Now, as an example, we took them MIPS 32 reduced instruction set computer architecture and we looked at how we can have a pipelined implementation of a subset of the MIPS 32 instruction set. Now, in this lecture, what we shall be looking at that whatever we talked about with respect to pipelining of the MIPS 32, there you recall we had identified the five stages instruction fetch, decode, execute, memory access and write back. And we also mentioned what are the required micro operations for the different stages. So, we shall now show how we can translate the description as we have presented in the last lecture into verilog code directly.

(Refer Slide Time: 01:24)



So, the topic of our lecture today is verilog modeling of the processor, the first part of it.



fine. Now, let us suppose that we have a sequence of instructions that we have written and the last instruction is a halt instruction. So, when we execute halt, the machine is supposed to stop. Let us just assume that instruction  $i + 1$ , let say was the halt instruction. Let us assume, this was the halt. So, there was some instructions before it,  $i - 1$ ,  $i - 2$  there are some instructions before it.

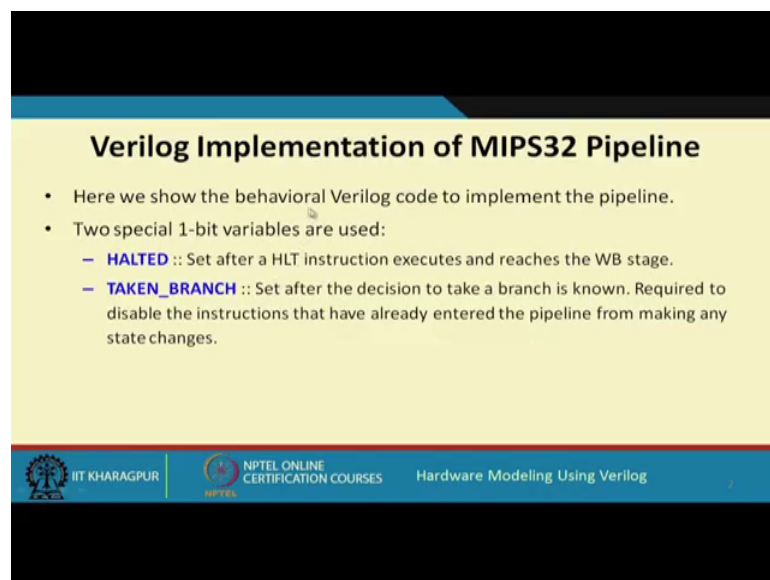
You see this halt instruction will be decoded during the ID phase. So, it is here after instruction fetch is done we will be knowing that this is actually a halt instruction, but we cannot stop the processor right here. Why, because the previous instruction which was here before halt that has still not completed it was still here in the EX stage only and the instruction before that  $i - 1$  that was in the MEM stage; and the one before that was in the WB stage. So, you cannot stop the processor as soon as you detect it is halt, but rather you wait till the right back stage is reached and only then you halt the processor.

But you remember that the halt instruction was found. So, you maintain a flip flop kind of a thing this we referred to as halted say a single bit variable, you set this flip flop to 1, whenever you find there is a halt instruction. So, what this will do you see just after the halt instruction again some more instructions will automatically be entering the pipe, but the machine is supposed to stop here, these should not be executed. So, while you execute a instruction, you check whether this flag is set or not. So, instructions  $i + 2$  and  $i + 3$  will be finding and will be detecting that this flag is indeed set to 1. So, it will not be doing any right like it; if it was a store instruction, it will not be writing into memory; and if it is a register instruction, it will not be writing into register. So, writing will be disabled if this halted state is there.

Now, similar will be the case if let us assume a scenario, there is a branch instruction, let say branch equal to zero something. And there are some instructions after that also. Let us assume again that this  $i + 1$  this was my branch instruction BEQZ. So, it is during ID you detect that it is a branch and if you look at the micro operations you will be identifying only at the end of EX whether the branch is actually to be taken or not because the target address is also computed and also the condition. Let say I have specified here R5, it is jumping to some label loop. So, it will check whether R5 is zero or not that is also done in the EX stage. So, whatever instruction has been fetched in the next cycle, if the branch is actually taken then this will have to be discarded right.

So, because of the same reason we maintain another flip flop this we call as taken branch. So, it means whenever you detect in the EX stage that the branch is actually to be taken that the condition is true you will actually have to go to loop. So, you set this variable to 1 again. And all the following instructions which have already entered, it will be checking this variable. If this variable is equal to 1, it will similarly not write anything in the registers or memory. So, this is as good as saying that all the instructions which are following the branch, they will not be executed. So, this is one change that you are making with respect to you can see the micro operation that we have seen means earlier during our last lecture.

(Refer Slide Time: 07:39)



**Verilog Implementation of MIPS32 Pipeline**

- Here we show the behavioral Verilog code to implement the pipeline.
- Two special 1-bit variables are used:
  - **HALTED** :: Set after a HLT instruction executes and reaches the WB stage.
  - **TAKEN\_BRANCH** :: Set after the decision to take a branch is known. Required to disable the instructions that have already entered the pipeline from making any state changes.

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

So, these are the two points I now just mentioned. So, we are using two 1-bit variables halted and taken branch. The reason I have already explained right fine.



(Refer Slide Time: 07:54)

```
module pipe_MIPS32 (clk1, clk2);
    input clk1, clk2;           // Two-phase clock

    reg [31:0] PC, IF_ID_IR, IF_ID_NPC;
    reg [31:0] ID_EX_IR, ID_EX_NPC, ID_EX_A, ID_EX_B, ID_EX_Imm;
    reg [2:0] ID_EX_type, EX_MEM_type, MEM_WB_type;
    reg [31:0] EX_MEM_IR, EX_MEM_ALUout, EX_MEM_B;
    reg EX_MEM_cond;
    reg [31:0] MEM_WB_IR, MEM_WB_ALUout, MEM_WB_LMD;


    reg [31:0] Reg [0:31];      // Register bank (32 x 32)
    reg [31:0] Mem [0:1023];    // 1024 x 32 memory

    parameter ADD=6'b000000, SUB=6'b000001, AND=6'b000010, OR=6'b000011,
              SLT=6'b000100, MUL=6'b000101, HLT=6'b111111, LW=6'b001000,
              SW=6'b001001, ADDI=6'b001010, SUBI=6'b001011, SLTI=6'b001100,
              BNEQZ=6'b001101, BEQZ=6'b001110;
endmodule
```



NPTEL ONLINE  
CERTIFICATION COURSES

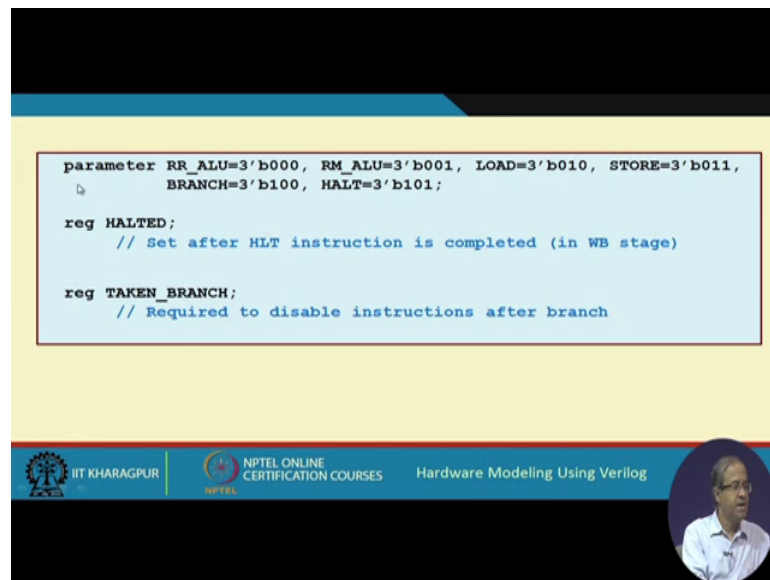
Hardware Modeling Using Verilog



Now, let us straight away come to the verilog code. This is the header of the description of the pipeline, where you declare all the registers. So, you see here in this module the inputs are only the clock nothing else, because instructions are coming from memory, there have been loaded in memory that is all; there is no separate input that is coming to the processor only clock one and clock two let say these are inputs. So, just like with the last example of pipelining we discussed a few lectures back we shall be using a two phase clock.

And stage wise we are identifying all the variables that are required there will be part of the inter stage latch and we are using the same variable naming convention that we have mentioned last lecture. PC - program counted will be a 32-bit register; and in addition, for the IF ID latch, we need IR and next PC NPC. For ID EX you need IR NPC A, B and Imm these are all 32-bit quantities. Now, in addition we also maintain some variable called type. Well, type let me just go ahead in next slide and show you what is type.

(Refer Slide Time: 09:27)



```
parameter RR_ALU=3'b000, RM_ALU=3'b001, LOAD=3'b010, STORE=3'b011,
          BRANCH=3'b100, HALT=3'b101;

reg HALTED;
  // Set after HLT instruction is completed (in WB stage)

reg TAKEN_BRANCH;
  // Required to disable instructions after branch
```

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

Type is actually the type of the instruction that is encoded in three bits we are identifying a few classes of instructions like register-register ALU we call it type 000 be type zero. Register to memory ALU 001; load – 010; store – 011; branch - 100 and halt - 101. So, these are the codes that you are defining and using parameters we can use this names instead of the numbers right. So, after the instructions decoded in the ID stage you recall during the ID stage instruction is already being fetched it is stored in IF, ID, IR the instruction register; from there the first six bits, you recall the instruction encoding if first six bits are the opcodes. So, the opcode will be decoded and there you can find out whether it is an add instruction add immediate load store jump whatever.

So, depending on that you set this variable type. And as it said type is a three bit variable, so it is a three bit register, and this type we forwarded across stages. So, there will be one ID EX type that will be forwarded to EX MEM type, MEM WB type and so on. Because in each stage, we will be taking some decision depending on the type of the instruction that is why we are keeping track of this information

Then in the EX MEM stage we have registers like IR ALU out, B and a single bit variable condition cond. This is required for jump or branch condition checking. And MEM WB we have IR, ALU out and for load instruction load memory data LMD. Here we declare the registers they are there are 32 registers 0 to 31 each of 32 bits. So, this is a


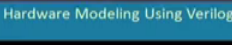



register bank 32 by 32. And this is how we declare the memory we are assuming there is 1024 words, each word of 32. So, the memory is 1024 by 32

Well, here in this parameter declaration, we are defining the opcodes, which we have already discussed in an earlier lecture we had assigned some six bit opcodes to all the instructions, and this parameter we actually summarising that. So, instead of referring the opcodes by their bit patterns, we will be using just add, sub then add immediate and so on because it will make our verilog code much easier to understand. We just using parameter this example illustrates, we can make our code more readable, because you can immediately know what that number actually stands for. So, here all these opcodes they are assigned a mnemonic add, sub and so on. So, we have assigned opcodes to all the instructions we are considering right.

Now, this we have already showed. These are the types and these are the two single bit flip flops I mentioned, halted, this will be set after the halt instruction is completed in the write backstage after the halt instruction reaches WB. And taken branch is another single bit register which will be set. After you decide that you are taking a branch and that is known only at the end of the EX stage as I have mentioned; only at the end of EX stage, you will be able to set this variable, so that all the following instructions which have entered the pipe they can check that value of the variable. And they will know that well the branch instruction is actually been taken, so we must not make any changes anywhere, so all rights will be disabled.

(Refer Slide Time: 13:53)

```
always @(posedge clk1) // IF Stage
  if (HALTED == 0)
  begin
    if (((EX_MEM_IR[31:26] == BEQZ) && (EX_MEM_cond == 1)) ||
        ((EX_MEM_IR[31:26] == BNEQZ) && (EX_MEM_cond == 0)))
    begin
      IF_ID_IR    <= #2 Mem[EX_MEM_ALUOut];
      TAKEN_BRANCH <= #2 1'b1;
      IF_ID_NPC   <= #2 EX_MEM_ALUOut + 1;
      PC         <= #2 EX_MEM_ALUOut + 1;
    end
  else
  begin
    IF_ID_IR    <= #2 Mem[PC];
    IF_ID_NPC   <= #2 PC + 1;
    PC         <= #2 PC + 1;
  end
end
```

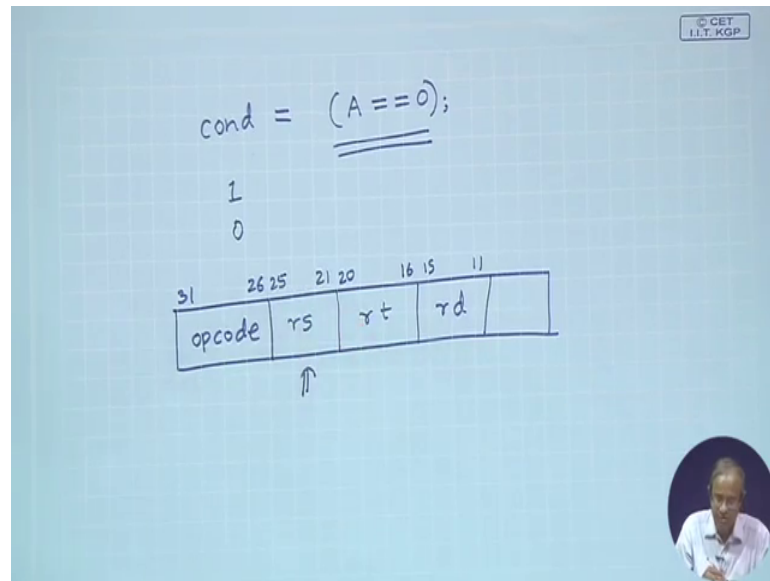


So, now let us look at the different stages of the pipeline. This is the instruction fetch stage. Well, in instruction fetch stage, it is triggered by positive edge of clock one. Well, we do this only if halted is not set, if a halt instruction is already set the halted flag that means, there is no need of fetching any further instruction. So, we execute this block only when halted this flag is equal to 0, this flip-flop.

Now, what do you check here? First in this if statement we are checking whether there is a branch, which is being taken. How we are checking it, you see we are first checking whether the opcode, you recall in the instruction register the opcodes were bit number 26 to 31. So, we are checking whether the opcode is equal to BEQZ. Well, in the parameter, we have already defined a bit pattern for BQZ. So, it is B branch equal to 0 and the condition flag equal to 1.



(Refer Slide Time: 15:08)



So, what these this condition flag will represent, this condition flag will actually represent this condition this register A you recall the output of the first register that you are fetching this equal to equal to 0, this is the expression you are assigning to cond, cond is a single bit variable. So, if this is true, cond will get 1; if this is false, cond will get 0. So, if the register value that is fetched in A is 0, then cond will be 1. So, this condition says that if it is a branch equal to instruction and it is actually equal to 0 that means, cond is 1. So, you have to branch or it was a branch not equal to 0 instruction and cond was equal to 0 that means, it was the register was not equal to 0 that means, cond is 0. If either of these is true which means we have to jump.

In that case, what we do, so this instruction register we do not fetch from p c, but rather we fetch it from the address we just calculated in EX MEM. You see for the branch instruction as it said at the end of the EX stage the address of the branch is already calculated and stored in EX MEM ALU out. So, you take the address from there that memory location you load in IR. And you set the taken branch flip-flop to one these variable to one indicating that we are actually taking the branch; and this is the address from where you are fetching the instruction. So, now my PC or the NPC which is pointing to the next instruction must be this plus one just the next address and else if it is not a branch, branch is not taken then it is the normal sequence of execution you fetch the instruction from PC, then increase PC and NPC both by 1. This is the instruction fetch stage.


Then let's come to instruction decode. So, in the instruction decode you recall, we basically do three things. First one we actually decode the instruction well which we are not showing in the verilog code because there is implied in the case statements. Secondly, we are pre-fetching to source registers and thirdly we are sign extending the 16-bit offset let us see how we are doing that.

(Refer Slide Time: 18:00)


```
always @(posedge clk2) // ID Stage
  if (HALTED == 0)
  begin
    if (IF_ID_IR[25:21] == 5'b00000) ID_EX_A <= 0;
    else ID_EX_A <= #2 Reg[IF_ID_IR[25:21]]; // "rs"

    if (IF_ID_IR[20:16] == 5'b00000) ID_EX_B <= 0;
    else ID_EX_B <= #2 Reg[IF_ID_IR[20:16]]; // "rt"

    ID_EX_NPC <= #2 IF_ID_NPC;
    ID_EX_IR <= #2 IF_ID_IR;
    ID_EX_Imm <= #2 {{16{IF_ID_IR[15]}}, {IF_ID_IR[15:0]}};
  end
```



IIT KHARAGPUR



NPTEL ONLINE  
CERTIFICATION COURSES

Hardware Modeling Using Verilog

So, here again we start by checking the halted flag. If halted is zero only then you do it; if halted is set you skip. Well, here when you are fetching the registers, you make a special check you recall I mentioned that R0 is a register special register, which is assumed to always contain the value zero, nothing else can be written to R0. So, here we are checking you see bit number is 21 to 25 in the instruction register this represents the first register operand. Just you recall again in the instruction register, the first 6 bits is the opcode. So, it is bit number 31 up to 26. Then you have register source 25, 21 then you have r t, r t is 16 to 20; then you have r d, r d is 11. So, here we are accessing r s bit number is 21 to 25.

So, we check whether this is 0 or not zero means we are trying to access r 0. So, if it is r 0; then we do not access register bank you straightaway assign zero to this A variable; otherwise A is assign to access the register with this register number. So, whatever five bit register number is there that is r s that will be loaded into A I D EX A.

Then you load the second operand, which is in bit number 16 to 20 same thing. If it is 0 you load the value 0 to B; otherwise, you read from the register this is your r t field load it into B. Then the other routine things this NPC, you simply forwarded because you need it later I f ID NPC will go to ID EX NPR IR also you forward to the next latch stage and immediate here you are doing the extension sign extension. So, in the IR bit number 0 to 15 indicates the offset. So, this will be the last 16 bits and the highest 16 bits you take bit number 15 replicated 16 times I said the sign bit will be replicated 16 times and then concatenate, this will become a 32 bit quantity. So, immediate is created by sign extension.

(Refer Slide Time: 20:59)

```

case (IF_ID_IR[31:26])
  ADD, SUB, AND, OR, SLT, MUL: ID_EX_type <= #2 RR_ALU;
  ADDI, SUBI, SLTI:           ID_EX_type <= #2 RM_ALU;
  LW:                         ID_EX_type <= #2 LOAD;
  SW:                         ID_EX_type <= #2 STORE;
  BNEQZ, BEQZ:                ID_EX_type <= #2 BRANCH;
  HLT:                         ID_EX_type <= #2 HALT;
  default:                    ID_EX_type <= #2 HALT;
                                // Invalid opcode
endcase
end

```

Then you also do another thing. You check the opcode again this is the opcode bit number 26, 31. Depending on the opcode type, you set the value of this type. This type you recall we have already defined three bit codes to the different types; and using parameter instead of writing 000001, we can write it in a much more readable way like RR ALU, RM ALU, load, store etcetera. And these are the instruction this opcodes also have been declared using parameter. So, you understand now what is the advantage of using parameters our code becomes much more readable. So, now you can actually see by looking at the code that well yes we are looking at the add instruction; otherwise seeing the opcode look at the table that well this opcode means add we have to refer every time, but this makes your code more readable right fine.

So, if it is either add, subtract, and, or set, less than or multiply, you call it as a register-to-register ALU type. If it is add immediate, subtract immediate, result immediate call it as RM type. If it is load word load; store word store; branch NEQZ or branch EQZ, it is branch. HLT is halt. And by default if it is something else by chance then also you make it halt there is an invalid opcode case. Just one thing this ID stage is triggered by clock 2.


(Refer Slide Time: 22:42)

```

always @(posedge clk1) // EX Stage
  if (HALTED == 0)
  begin
    EX_MEM_type <= #2 ID_EX_type;
    EX_MEM_IR <= #2 ID_EX_IR;
    TAKEN_BRANCH <= #2 0;

    case (ID_EX_type)
      RR_ALU: begin
        case (ID_EX_IR[31:26]) // "opcode"
          ADD: EX_MEM_ALUout <= #2 ID_EX_A + ID_EX_B;
          SUB: EX_MEM_ALUout <= #2 ID_EX_A - ID_EX_B;
          AND: EX_MEM_ALUout <= #2 ID_EX_A & ID_EX_B;
          OR: EX_MEM_ALUout <= #2 ID_EX_A | ID_EX_B;
          SLT: EX_MEM_ALUout <= #2 ID_EX_A < ID_EX_B;
          MUL: EX_MEM_ALUout <= #2 ID_EX_A * ID_EX_B;
          default: EX_MEM_ALUout <= #2 32'hxxxxxxxx;
        endcase
      end
    endcase
  end

```






Then next comes the EX stage which is again triggered by clock one and the way we started is the same we check the halted flag. So, if it is equal to zero only then we do it; otherwise, you skip it. First we forward a few things well this EX type, whatever type you have calculated earlier in the last stage will be forwarded to the next stage if IR is also forwarded. And taken branch you see taken branch this variable was set in the IF stage right if you see earlier. So, in the IF we had set this variable here. Because of this the next instruction that we will see later that we will not be allowed to write into the WB stage, but all instruction after that again should be allowed to continue because there will be starting to fetch from the correct address because already program counter has been updated.

So, now we are resetting the taken branch back to 0 here. Now, here depending on the instruction type were carrying out the operations there is a case on type ID EX type. So, if it is register-register ALU then again we use a case statement on the opcode this is opcode. So, if it is add, subtract, and, or, slt, or mul, then specify what you are doing. If it

is add, ID EX A and ID EX B are added result is stored in ALU out. If it subtract subtract, and, or; if it is set if less than you just do a comparison less than if it is true then one will be stored in ALU out; if it is false zero will be stored in ALU out. And mul is multiplication. And if it is an invalid opcode, then you just load this undefined xxx in ALU out.

(Refer Slide Time: 24:50)

```
RM_ALU: begin
  case (ID_EX_IR[31:26]) // "opcode"
    ADDI: EX_MEM_ALUOut <= #2 ID_EX_A + ID_EX_Imm;
    SUBI: EX_MEM_ALUOut <= #2 ID_EX_A - ID_EX_Imm;
    SLTI: EX_MEM_ALUOut <= #2 ID_EX_A < ID_EX_Imm;
    default: EX_MEM_ALUOut <= #2 32'hxxxxxxxx;
  endcase
end
```



NPTEL ONLINE  
CERTIFICATION COURSES

Hardware Modeling Using Verilog


Next if it is registered to memory RM ALU this type then again you look at the opcode, these are the three kinds of RM ALU instructions add immediate. Here will be adding a with the immediate data. Subtract will be subtracting immediate data from a SLTI will be compared in less than A and I mm, rest is same.

(Refer Slide Time: 25:17)


```
LOAD, STORE:
begin
  EX_MEM_ALUOut <= #2 ID_EX_A + ID_EX_Imm;
  EX_MEM_B      <= #2 ID_EX_B;
end

BRANCH: begin
  EX_MEM_ALUOut <= #2 ID_EX_NPC + ID_EX_Imm;
  EX_MEM_cond   <= #2 (ID_EX_A == 0);
end

endcase
end
```




IIT KHARAGPUR



NPTEL ONLINE  
CERTIFICATION COURSES

Hardware Modeling Using Verilog



Then if it is load or store then you are calculating the address of the memory address of the operand. What you do you simply add the value of A and I mm. So, this will give you the address of the memory this is stored in ALU out. And the values of B is forwarded because this will be requiring for store instructions later. And if it is branch then you see this is where you take the decision branch, you calculate the target address of the branch. How you do it the value of the program counter NPC is added to the offset I mm. So, here you are calculating the target address if you have to take a branch and also you are evaluating this variable cond right.

Now, you see this ALU out and cond, which you are calculating in the EX stage let us again go back to the IF stage you are actually using them in IF stage you see you are looking at this cond and this branch address these are being used here. So, now you can understand from EX MEM, how they are coming they are actually getting calculated here fine. Let us move on to the MEM stage, this is again triggered by clock two. So, again if this halted is zero only then you execute this. So, what you do type is forwarded, IR is also forwarded. And here again there is a case statement well let us make it uniform here I did not put this hash to I given a delay just for notational notational purpose. Let us use the same delay to a hash two fine.


(Refer Slide Time: 27:14)

```
always @(posedge clk2) // MEM Stage
  if (HALTED == 0)
  begin
    MEM_WB_type <= #2 EX_MEM_type;
    MEM_WB_IR <= #2 EX_MEM_IR;


    case (EX_MEM_type)
      RR_ALU, RM_ALU:
        MEM_WB_ALUOut <= #2 EX_MEM_ALUOut;

      LOAD:
        MEM_WB_LMD <= #2 Mem[EX_MEM_ALUOut];

      STORE:
        if (TAKEN_BRANCH == 0) // Disable write
          Mem[EX_MEM_ALUOut] <= #2 EX_MEM_B;
    endcase
  end
```




IIT KHARAGPUR



NPTEL ONLINE  
CERTIFICATION COURSES


Hardware Modeling Using Verilog




So, if it is a register to register RR to RM, you do nothing just to forward ALU out to the next latch. If it is load, the address of memory is already in ALU out, we have calculated in the last type EX we access memory and store it in LMD. And store well in store you check whether taken branch is 0 or 1; if it is 1, you do not do a write disable write; if taken branch has been 0 only then you do the right whatever is there will be that you store in the memory location.

(Refer Slide Time: 27:54)

```
always @(posedge clk1) // WB Stage
  begin
    if (TAKEN_BRANCH == 0) // Disable write if branch taken
    case (MEM_WB_type)
      RR_ALU: Reg[MEM_WB_IR[15:11]] <= #2 MEM_WB_ALUOut; // "rd"
      RM_ALU: Reg[MEM_WB_IR[20:16]] <= #2 MEM_WB_ALUOut; // "rt"
      LOAD: Reg[MEM_WB_IR[20:16]] <= #2 MEM_WB_LMD; // "rt"
      HALT: HALTED <= #2 1'b1;
    endcase
  end
endmodule
```




IIT KHARAGPUR



NPTEL ONLINE  
CERTIFICATION COURSES

Hardware Modeling Using Verilog



And lastly in the write back or WB stage here triggered by positive edge of clock one again. Here we check if taken branch is zero only then you do it. If taken branch is one means someone is taking the branch you disable the writes here, you should not write into the register, these registers, these instructions are actually dummy instructions they should be ignored. So, what we do if it is RR ALU, you recall there the destination register is stored in bit numbers 11 to 15 in the instruction register. So, ALU out will be stored here that is rd. If it is rn it is stored in rt bit numbers 16 to 20. And if it is load again the data is in LMD, it will be stored in rt again 16 to 20. And if it is the halt instruction it is here in the WB stage, you are setting this flag halted to 1, so that after you have halted you have set the flag to 1, all further instructions will not do anything else, the machine will basically stop.

So, with this we come to the end of this lecture, where basically we have translated the micro operation that we discussed in our last lecture to Verilog. Of course, we have done a few things that are very specific to pipeline implementation like those two additional flags we have maintained and we have checked those flags at the beginning of every stage whether that flag is active or inactive, if necessary you skip some stage. So, in the next and final lecture, what we shall be discussing, we shall be looking at some example programs that you want to run on this processor we have designed, and let us see how it works, this we shall be seeing in our next lecture.

Thank you.