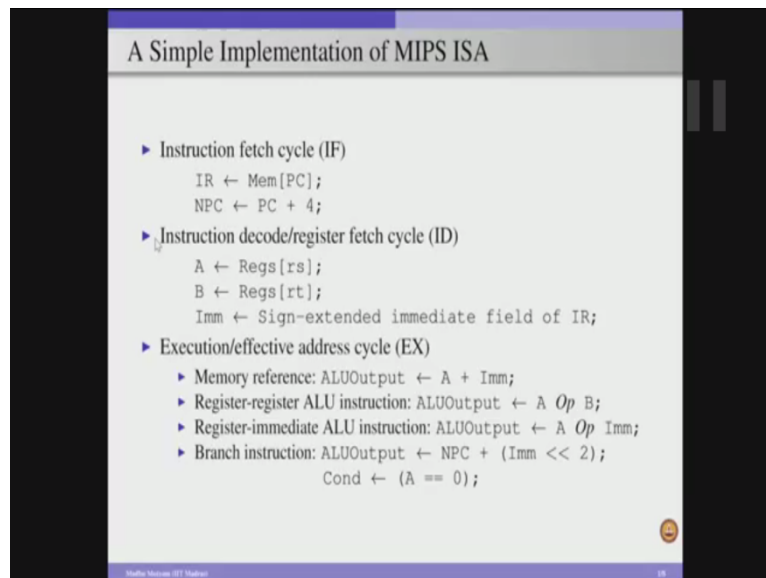


**Computer Architecture**  
**Prof. Madhu Mutyam**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Madras**

**Module – 05**  
**Lecture – 17**  
**Implementation of MIPS pipeline**

So, in the last 4 modules we discussed fundamentals of pipelining. We considered instruction pipelining. We also discussed pipeline hazards and techniques to minimize the overheads associated with this pipeline hazards. In this module we are going to consider pipeline implementation of subset of MIPS ISA. So, again this MIPS ISA is a class of RISC architectures and we are going to consider a five stage pipeline implementation. So, our five stage pipeline consists of the Instruction Fetch, Instruction Decode, Execute stage, Memory access and Write Back.

(Refer Slide Time: 01:04)



A Simple Implementation of MIPS ISA

- ▶ Instruction fetch cycle (IF)  
 $IR \leftarrow \text{Mem}[PC];$   
 $NPC \leftarrow PC + 4;$
- ▶ Instruction decode/register fetch cycle (ID)  
 $A \leftarrow \text{Regs}[rs];$   
 $B \leftarrow \text{Regs}[rt];$   
 $\text{Imm} \leftarrow \text{Sign-extended immediate field of IR};$
- ▶ Execution/effective address cycle (EX)
  - ▶ Memory reference:  $\text{ALUOutput} \leftarrow A + \text{Imm};$
  - ▶ Register-register ALU instruction:  $\text{ALUOutput} \leftarrow A \text{ Op } B;$
  - ▶ Register-immediate ALU instruction:  $\text{ALUOutput} \leftarrow A \text{ Op } \text{Imm};$
  - ▶ Branch instruction:  $\text{ALUOutput} \leftarrow NPC + (\text{Imm} \ll 2);$   
 $\text{Cond} \leftarrow (A == 0);$

In the first stage, that is the instruction fetch cycle, we will read the contents from the program counter and we go to the memory and get the data stored in that particular address and store that result in another register called as IR- instruction register. And parallelly we also increment by PC by 4 because here we are assuming it is a 32 bit address. So, that next PC will be the previous PC plus 4.

And once the instruction fetch cycle is completed the next one is Instruction Decode. As we discussed earlier in the instruction decode we also perform register read operation. So, once the instruction is decoded, we know what is the type of instruction it is? and we also know the addressing mode. And parallelly in the register fetch phase, we are going to read the contents from the register file from the location specified by this 'rs' and 'rt'. Here rs, rt are the register fields in the instruction.

And if it is of immediate addressing mode, then automatically we will have to read the immediate field from the instruction register. And we will extend the sign that will be stored in our temporary register. Here A, B and Imm are temporary registers where we are going to store these corresponding values. And once the instruction decode and register fetch cycle is completed, the next one is execute stage. In the execute stage we also perform the effective address calculation, if the instruction is memory instruction. So, we already know from the id stage what is the instruction this particular bit stream that is stored in IR or what is the operation of the instruction that we loaded in the instruction register.

So, based on that type we are going to different operations. If the instruction is memory reference instructions, that is either a load or a store instruction then we are going to perform the effective address calculation. That will be by adding the content of A with the immediate value. So, it is effectively, we have a base address and then we are going to add offset with that and effectively we are going to get the effective address for our memory reference instruction. So, that is stored in the temporary register which is ALU output, but if the instruction is a register-register ALU type instruction, then we are going to perform operation on the contents of A and B and store that in our temporary register associated with the functional unit that is ALU output.

And in the case of register immediate ALU instruction type we are going to perform operation on A and immediate registers and store the result in the temporary register ALU output. And note that here r indicates the operation that we are going to perform and that is known from the opcode field from the id stage because after the id stage we know what is the type of instruction from the opcode field and based on the type of instruction we are going to perform the corresponding operation on the contents of this temporary registers A, B or Imm. And if the instruction is a branch instruction then we have to test the branch condition.

That will be tested by a using the second instruction here. That is condition is evaluated to whether A equal to 0 it is a branch equal to 0 or not, that is what we are going to consider in this particular simple implementation of MIPS ISA. And parallelly if the branch is going to take place then we have to actually consider the target address. And for that we will perform the effective address calculation. That is by adding this next PC value and this Imm, but Imm is shifted by 2 positions to get the word. So because we are working at the word level granularity, we are going to get the word address by shifting 2 positions and this value is added to NPC.

So, that that will be the effective target address, if the branch is going to take place. So, effectively in these 3 stages, we fetch the instruction, we incremented the PC by 4, we decoded the instruction and came to know the type of instruction we are going to perform. And similarly, we perform the register read operations from the register file and if it is an immediate addressing mode is used in instruction, then we will read the immediate value from the immediate field in the instruction to the temporary register. And based on the type instruction, we calculate the effective address or we are performing the actual operation. And once this execute stage is completed the next one is the memory access.

(Refer Slide Time: 06:39)

A Simple Implementation of MIPS (Contd)

- ▶ Memory access/branch completion cycle (MEM)
  - ▶  $PC \leftarrow NPC;$
  - ▶ Memory reference:  $LMD \leftarrow Mem[ALUOutput]$  or  $Mem[ALUOutput] \leftarrow B;$
  - ▶ Branch instruction:  $if(Cond) PC \leftarrow ALUOutput;$
- ▶ Write-back cycle (WB)
  - ▶ Register-register ALU instruction:  $Regs[rd] \leftarrow ALUOutput;$
  - ▶ Register-immediate ALU instruction:  $Regs[rt] \leftarrow ALUOutput;$
  - ▶ Load instruction:  $Regs[rt] \leftarrow LMD;$

18

And in this particular implementation we consider the branch completion only in the memory access stage. So, here in the memory access if it is a load or a store instruction we have to go to the memory and then perform the actual operation. So, before that we are storing this PC

value, new PC value into the register PC. And we perform the actual memory reference operation. If it is a load instruction then based on the address stored in this ALU output which is calculated in the previous stage as part of effective address calculation and from the memory in that particular location, then we get the content and that data is stored in the load memory data register which is also another temporary register.

And if it is a store instruction, then the value that is stored in B is going to be stored in the memory location specified by the ALU output which is also again an effective address calculation performed in the previous stage. So, ALU output is going to give you the effective address, which is calculated in the previous stage. And we go to the memory to that particular location and store the contents of B in that and if it is a branch instruction. So, based on the condition we see whether PC needs to be updated or not. If the condition is true then only PC will be updated with the new ALU output.

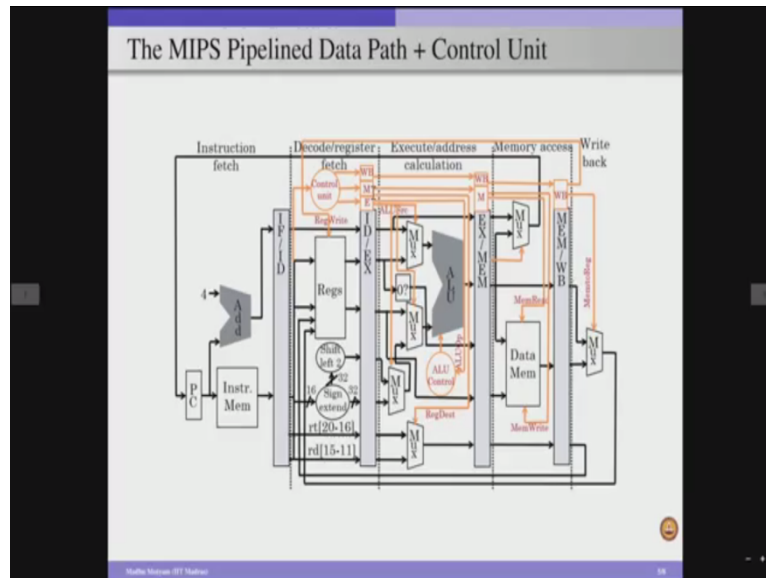
So, effectively at the end of this fourth stage, our branch target address is known and branch outcome is known. And also we perform the memory access whether it is a store operation or a load operation. In the case of store operation, the store is said to be completed at the end of this fourth stage. And in the case of this load operation we read the value from the memory and stored in a temporary register called as LMD.

And finally, in the last stage that is the write back stage. So, now we are going to write the computed values into the actual registers. If it is a register-register ALU instruction then we are going to write the value from the ALU output to the register file in a specific location specified by this rd. And if it is a register immediate ALU instruction, then the ALU output will be written to the register file at a location specified by rt. And if it is a load instruction then we are going to store the LMD value to register file at location rt.

Now, we are going to see how this is implemented by considering the data path as well as the control path? So, before going for the pipeline implementation we will just consider the non-pipeline implementation of this mixed data path. By the way, any processor design consists of data path as well as the control path. The data path is going to give you the interconnection among different hardware components and the control path is going to give you the timing signals to different components. So, that these components will perform operations based on some event is happened or at a specific time instance. So, now we are going to look at the

data path associated with this non pipeline implementation of the MIPS instructions. So, the first stage we discussed the instruction fetch

(Refer Slide Time: 10:10)



So, we are going to consider program counter register, the instruction memory, there is another register called as an IR instruction register and there is another temporary register called as a next PC. And we have an order which is going to perform the addition operation for the contents of PC and 4 and that will be stored in NPC, the next PC. And the next, based on this IR we take the contents in the IR and give it to the decoder, instruction decoder and instruction decoder is going to decode what is the type of instruction, what is the destination addresses, what is the source address for the operands all these things will be performed here by using a decoder. And we index into the register file to read the register contents.

And these contents will be stored in temporary registers A, B. And similarly, if immediate addressing mode is used in the instruction, then we have to extend that immediate field, which is typically 16 bit field in our instruction, and we extend that to 32 bit by extending the sign and store that either Imm field or signed sign extended immediate the temporary register. And then next, once we have the contents in A, B registers then we consider collection of muxes to collect between the NPC or A here as input to the ALU. And similarly, the A value will be compared to 0 to see whether the condition is going to be true or false. And that will be stored in the condition register here. And there is another mux which is going to select between the contents of B as well as the contents coming out of this mux.

And this mux has inputs from the immediate as well as sign extended immediate. And also we have these registers  $rt$  and  $rd$  which are given as an input to mux. And this ALU functional unit is the one which is actually performing the effective address calculation or the actual operation specified by our opcode. And the output will be stored in the temporary register associated with this functional unit that is ALU output and the next stage. So, we have another mux which takes the input from this NPC or output from the ALU output and it will select one of these two.

And then that will be given as an input to the PC depending on whether the branch is taken place or not taken place, the mux will select one of these 2 inputs. And then that will be sent as an input to the PC so, that the next PC will be updated accordingly. And if it is a memory access operation then we consider this ALU output as one input and similarly, the B as another input and so based on that we perform the actual operation, whether it is a write operation or a read operation.

And if it is a read operation we are going to read from the specified location and the data will be stored in another temporary register LMD. And finally, we consider another mux which has 2 inputs, one is the output from this ALU output and the other one is from the LMD. If it is an ALU instruction we are going to select this output and if it is a load instruction we are going to select this input from the mux. And that will be sent to the register file, because last stage is the write back stage, where we are going to write to the register file.

And this consists of the complete data path for implementing any instruction in the MIPS, subset of the MIPS ISA. This entire setup is going to give you a non-pipeline implementation of MIPS data path. And now we know that there are so many temporary registers and if I am not going to perform the pipeline operation here, then effectively this is going to take significant amount of time to perform each of these operations. And as a result in order to improve the throughput, that is what we discussed as part of the last 4 modules, we have to go for a pipeline design.

So, similarly, here this data path also can be implemented in a pipeline fashion by considering the pipeline registers between each of these stages. So, if we see here, compared to the previous one so, here the previous one we have all these temporary registers and these temporary registers are replaced with pipeline registers. And we consider 4 pipeline registers here.

The first pipeline register is between the instruction fetch and the decode stage. And that is represented by IF by ID and second pipeline register is in between the decode and the execute stage. So, it is represented by ID/EX. And the third pipeline register is between execute and memory access EX/MEM. And finally, we have another pipeline register between memory access and write back, MEM/Write Back. And all these pipeline registers are latches. So, there is a clock associated with that and at the positive edge or negative edge of the clock these latches are going to latch the inputs coming to them, from the previous stage.

And that will be latched in this registers and these contents will be considered as an input to the next stage. And it will take some amount of time, that is the pipeline cycle time. So, that the next stage. So, perform the operations and again at the end of, again when we apply the next passage of the negate then the outcomes from this stage will be stored in the next pipeline register and it will proceed further. Effectively so, when we have this pipeline register to separate different stages then we can improve the overall throughput.

So, in other words, while one instruction is in the execute stage, we can perform the decode operation on the next instruction. And similarly, we can fetch the next instruction to that. And as a result at any point in time 5 instructions can be there in these 5 stages of the pipeline. And as a result we can improve the overall throughput of the system. By the way so, this shows the pipeline data path for the MIPS ISA. Now, you want to add the control path to this so that we will make the full fledged pipeline processor design.

So, for that we consider a control unit and the control unit generates signals to different hardware components in our data path so that these components will work properly and then the compute the given required function efficiently. This control unit generates signal to the execute stage, memory stage, write back stage. And in the execute stage actually it generates signals as a select input to the muxes. For, example this first mux which is going to take 2 inputs, one is coming from this register file and other one is coming from this PC. So, now this ALU source signal is going to select one of these two.

And based on the control signal value whatever is generated for this ALU source, one input will be selected from this mux and given as input to the ALU. And similarly, this ALU source also can be given as the source signal to the select signal to the another mux, which is going to take 2 inputs, one from the register file the other one is from this immediate field. And similarly, for this mux also we have the select signal coming from this ALU source. So, in

other words this control unit is generating set of ALU source signals and these signals will be given as select inputs to all these muxes.

And based on the instruction type this ALU signal, ALU source signal, will be selected and accordingly the mux is going to transfer the correct data to the ALU in the execute stage. And also this control unit generates a signal that is ALU op signal which is based on the opcode that is there in the instruction. And based on that this ALU control unit will act and accordingly it will select a required functional unit in this ALU. Here ALU consists of collection of functional units and based on the ALU op signal, ALU controller will generate appropriate signal to select a required functional unit and that functional unit is going to perform operation on the operands.

Also, this control unit generates a signal called a register destination and which is going to select one of these 2 registers rt and rd, which are given as inputs to this mux. Because we require this one of these 2 based on the type of instruction we are executing to finally, store the contents in the register file. For that we are going to select the register destination and that also again in the instruction when we decoded we know the register destination address and based on that we are going to give appropriate signal here. And these are about the control signals that are required in the execute stage.

And now we will move onto the memory the access the next stage. And in this the control unit is going to generate a signal for memory read or memory write. If we are performing a load instruction and if our instruction is load instruction then memory read signal will be selected and memory write signal will be deselected. And if it is a store operation, memory read is deselected and memory write is selected and because we already sent the address and the data here.

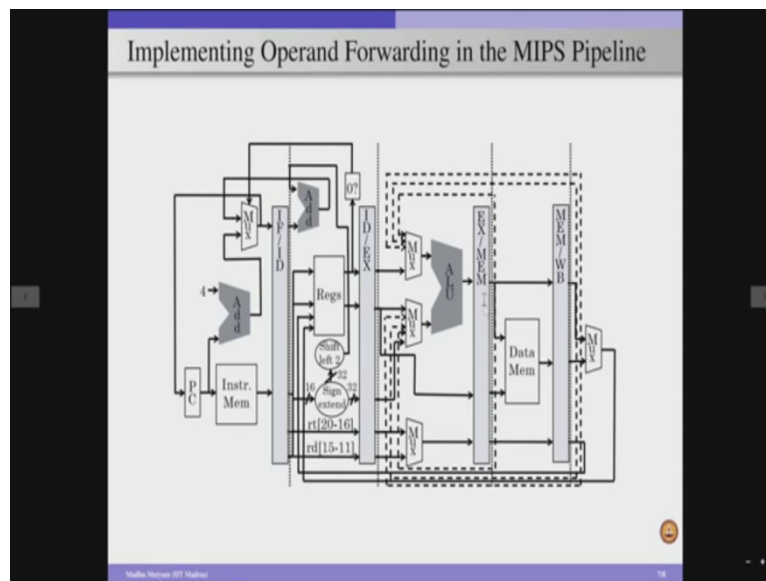
If it is a store then it will be written to the appropriate location. Whereas, if it is a load instruction then the address is given here and we go to the memory and read the data and the data will be stored in the temporary register or pipeline register between the memory access stage and the write back. We give a signal to the mux based on the condition of our branch. And based on that our PC will be updated because we already discussed the branch outcome and the branch test condition is completed only in the memory access stage in this particular 5 stage pipeline design.



So, as a result we require this control signal for this mux. So, in the write back stage we are going to write the contents to the register file. So, here again if we are going to perform the load operation, we are going to write the contents that are read from the memory to the registers. So, as a result we have to this signal, memory to register, and as a result it is going to select the data that is read from this data memory. And it will be passed through this bus and then finally, it will be written to the register file.

And also we have to give this signal to the register file, indicating that we are performing register write. And note that we are writing to the register file only in the write back stage. So, as a result the write back signal or register write signal will be selected only in the last stage. And this is about the pipeline implementation of a subset of MIPS ISA. And here, if we see, we are actually performing the branch instruction in the fourth stage. So, in order to minimize the penalty associated with the branches, we can move this branch computation from the fourth stage to the second stage.

(Refer Slide Time: 22:35)



So, for that what we can do is, we can add extra hardware component that is in order in the second stage itself. Previously we were using this ALU to compute the branch target address and also the test condition for the branch, but now we are adding a separate hardware component in the second stage. And so that our branch outcome and the branch target will be completed in the second stage, but the trade-off is we are going to incur extra hardware overhead associated with the adder, but the advantage is we are going to resolve the branches

quickly. And finally, in the case of operand forwarding we can forward from this particular pipeline register back to the muxes.

So that if there is a match for the subsequent instruction which is requiring this particular value then that value will be sent to this mux and as a result ALU will compute the subsequent instruction without creating any stalls. And similarly, if we are going to forward the load value to a subsequent store or ALU operation, then we can forward from this pipeline register. And also this pipeline register can even forward the value that is computed from an ALU operation also, because ALU operation whatever it computes that will be written to the register only in the last stage.

In the meanwhile if there is any dependent instructions which requires this computed value it can be forwarded either from this pipeline register or this pipeline register. But for the load forwarding, we have to use only this forwarding pipeline register to forward the value. And so with that we complete the pipelining unit. And so I am concluding this pipelining unit with this.

Thank you