5-Stage Pipeline Core

Dividing the datapath into 5-stages

As we know there is a single datapath in a single cycle core which executes each instruction in one clock cycle and only after that instruction is fully completed the next instruction is fetched. To make our processor more efficient we implement the pipeline methodology which helps us increase our throughput by using our existing datapath more efficiently. What we do is that we divide our datapath into 5 stages named **Instruction Fetch**, **Instruction Decode**, **Execute**, **Memory**, **Write Back.** After the first instruction is fetched from the memory and is decoding in the datapath we start to fetch the next instruction. When the first instruction is in execution stage, the second instruction has now reached the decoding stage and a third instruction is now being fetched. So what we did is that we are not dedicating our whole datapath for just one instruction to complete and then start fetching the next instruction instead we are doing it side by side thus improving the throughput of our processor. *This in essence is all what pipelining is all about.*

Now coming to the point as to what we need to do in order to divide our datapath into the above stages. We can see that we have to pause the execution of the instruction in each stage so that we can also perform other work side by side in parallel. Meaning that when an instruction is in decode stage we need to start fetching the next instruction in the fetch stage. What we need to do is *store* the state of the instructions, so that we know that in each stage what instruction is being processed by our datapath. Things will get more clearer as we move forward, do not worry if you get confused. Remember that we need a storage element between each stage so that we can store the state of all the instructions currently in the datapath. Since we need a storage element to save the state of the instruction, we need registers which we call *pipeline registers.* They are simple registers placed between each stage to store the state of the instruction in that specific stage. As we have 5 stages we will be needing 4 pipeline registers. One in between the **Instruction Fetch** and I**nstruction Decode** stages, Second in between the **Instruction Decode** and **Execution stages,** Third in between the **Execution and Memory** stages and Fourth in between the **Memory and Write Back** stages.

Now as we know what pipelining is and what pipeline registers are why we need them. Let's discuss these stages and what they do:

1) Instruction Fetch

The name easily tells what this stage does. This stage fetches the new instruction from the instruction memory and feeds it into the IF/ID (Instruction Fetch and Instruction Decode) pipeline register.

2) Instruction Decode

In this stage the instruction that was fetched in the previous stage is decoded. The instruction bits are split into Control, Register File and Immediate Generation block and their outputs are fed forward into the ID/EX (Instruction Decode and Execute) pipeline register.

3) Execute

The instruction that was decoded is now executed by the ALU in this stage. It's output is then fed forward in the EX/MEM (Execution and Memory) pipeline register.

4) Memory

In this stage the result of the instruction is stored in the memory if the instruction is Store type. If the instruction is Load type then data from the memory is loaded in this stage and forward into the MEM/WB (Memory and Write Back) pipeline register. If the instruction is R-type (add, and, or, xor, etc) then nothing happens in this stage the result is forwarded into the MEM/WB pipeline register.

5) Write Back

In this stage the result is written back to the register file whether it is the result of the ALU if the instruction is R-Type, I-Type and for S-Type instruction nothing happens in this stage because store instruction do not write anything to the registers.

Creating the pipeline registers

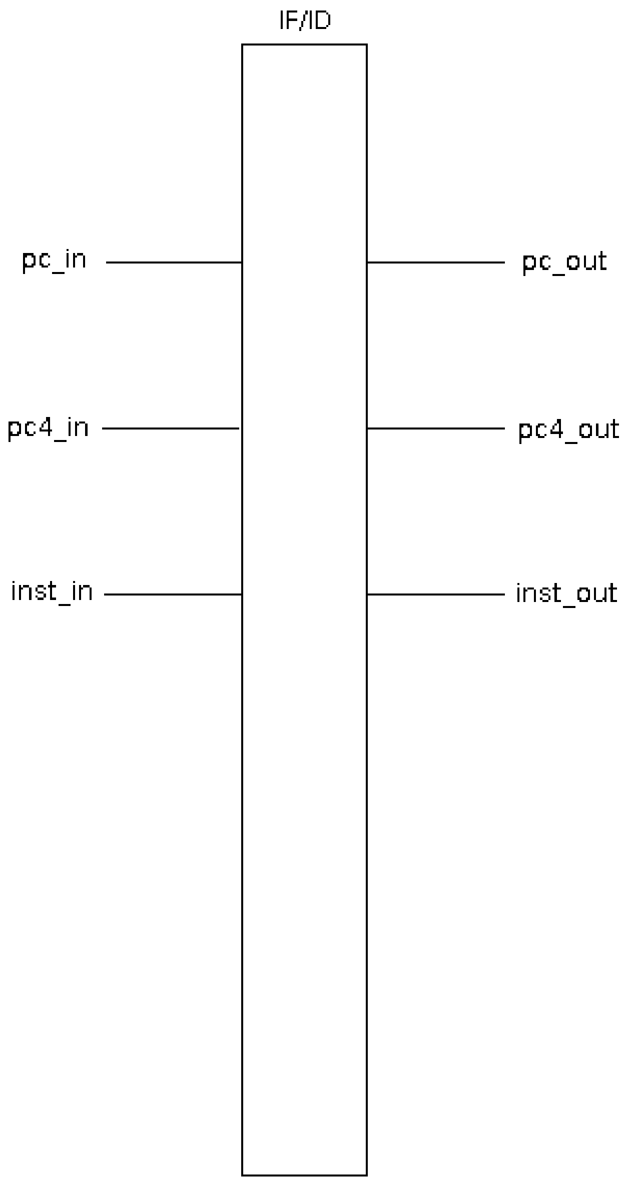
The 5-Stage pipeline core consists of 4 pipeline registers named IF_ID for the pipeline register between the Instruction Fetch and Instruction Decode stages, ID_EX for the pipeline register between the Instruction Decode and Execution stages, EX_MEM for the pipeline register between the Execution and Memory stages, MEM_WB for the pipeline register between the Memory and Write Back stages.

1) Coding IF_ID Pipeline Register

```
        class IF_ID extends Module {
          val io = IO(new Bundle {
        val pc_in = Input(SInt(32.W))
        val pc4_in = Input(SInt(32.W))
        val inst_in = Input(UInt(32.W))
        val pc_out = Output(SInt(32.W))
        val pc4_out = Output(SInt(32.W))
        val inst_out = Output(UInt(32.W))
  })

        val pc_reg = RegInit(0.S(32.W))
        val pc4_reg = RegInit(0.S(32.W))
        val inst_reg = RegInit(0.U(32.W))
        pc_reg := io.pc_in
        pc4_reg := io.pc4_in
        inst_reg := io.inst_in
        io.pc_out := pc_reg
        io.pc4_out := pc4_reg
        io.inst_out := inst_reg
}
```

Here is the Chisel code for creating an IF/ID Pipeline register. We take the current pc value, the next pc value and the instruction as inputs. Then we create registers for storing these specific values coming from the external input and then produce the output of each input using those registers.

Block Diagram:



2) Coding the ID_EX Pipeline Register

```
class ID_EX extends Module {
        val io = IO(new Bundle {
  val IF_ID_pc = Input(SInt(32.W))
  val rs1_sel_in = Input(UInt(5.W))
  val rs2_sel_in = Input(UInt(5.W))
  val rs1_in = Input(SInt(32.W))
  val rs2_in = Input(SInt(32.W))
  val imm = Input(SInt(32.W))
  val rd_sel_in = Input(UInt(5.W))
  val func3_in = Input(UInt(3.W))
  val func7_in = Input(UInt(1.W))
  val ctrl_MemWr_in = Input(UInt(1.W))
  val ctrl_MemRd_in = Input(UInt(1.W))
  val ctrl_Branch_in = Input(UInt(1.W))
  val ctrl_RegWr_in = Input(UInt(1.W))
  val ctrl_MemToReg_in = Input(UInt(1.W))
  val ctrl_AluOp_in = Input(UInt(3.W))
  val ctrl_OpA_sel_in = Input(UInt(2.W))
  val ctrl_OpB_sel_in = Input(UInt(1.W))
  val ctrl_nextPc_sel_in = Input(UInt(2.W))

  val pc_out = Output(SInt(32.W))
  val rs1_out = Output(SInt(32.W))
  val rs2_out = Output(SInt(32.W))
  val imm_out = Output(SInt(32.W))
  val func3_out = Output(UInt(3.W))
  val func7_out = Output(UInt(1.W))
  val rd_sel_out = Output(UInt(5.W))
  val rs1_sel_out = Output(UInt(5.W))
  val rs2_sel_out = Output(UInt(5.W))
  val ctrl_MemWr_out = Output(UInt(1.W))
  val ctrl_MemRd_out = Output(UInt(1.W))
  val ctrl_Branch_out = Output(UInt(1.W))
  val ctrl_RegWr_out = Output(UInt(1.W))
  val ctrl_MemToReg_out = Output(UInt(1.W))
  val ctrl_AluOp_out = Output(UInt(3.W))
  val ctrl_OpA_sel_out = Output(UInt(2.W))
  val ctrl_OpB_sel_out = Output(UInt(1.W))
  val ctrl_nextPc_sel_out = Output(UInt(2.W))
        })
val pc_reg = RegInit(0.S(32.W))
val rs1_reg = RegInit(0.S(32.W))
val rs2_reg = RegInit(0.S(32.W))
val imm_reg = RegInit(0.S(32.W))
val rd_sel_reg = RegInit(0.U(5.W))
val rs1_sel_reg = RegInit(0.U(5.W))
val rs2_sel_reg = RegInit(0.U(5.W))
val func3_reg = RegInit(0.U(3.W))
val func7_reg = RegInit(0.U(1.W))

val ctrl_MemWr_reg = RegInit(0.U(1.W))
```

```
        val ctrl_MemRd_reg = RegInit(0.U(1.W))
        val ctrl_Branch_reg = RegInit(0.U(1.W))
        val ctrl_RegWr_reg = RegInit(0.U(1.W))
        val ctrl_MemToReg_reg = RegInit(0.U(1.W))
        val ctrl_AluOp_reg = RegInit(0.U(3.W))
        val ctrl_OpA_sel_reg = RegInit(0.U(2.W))
        val ctrl_OpB_sel_reg = RegInit(0.U(1.W))
        val ctrl_nextPc_sel_reg = RegInit(0.U(1.W))



        pc_reg := io.IF_ID_pc
        rs1_reg := io.rs1_in
        rs2_reg := io.rs2_in
        imm_reg := io.imm
        rd_sel_reg := io.rd_sel_in
        rs1_sel_reg := io.rs1_sel_in
        rs2_sel_reg := io.rs2_sel_in
        func3_reg := io.func3_in
        func7_reg := io.func7_in
        // Storing Control state in the registers
        ctrl_MemWr_reg := io.ctrl_MemWr_in
        ctrl_MemRd_reg := io.ctrl_MemRd_in
        ctrl_Branch_reg := io.ctrl_Branch_in
        ctrl_RegWr_reg := io.ctrl_RegWr_in
        ctrl_MemToReg_reg := io.ctrl_MemToReg_in
        ctrl_AluOp_reg := io.ctrl_AluOp_in
        ctrl_OpA_sel_reg := io.ctrl_OpA_sel_in
        ctrl_OpB_sel_reg := io.ctrl_OpB_sel_in
        ctrl_nextPc_sel_reg := io.ctrl_nextPc_sel_in

        io.pc_out := pc_reg
        io.rs1_out := rs1_reg
        io.rs2_out := rs2_reg
        io.imm_out := imm_reg
        io.rd_sel_out := rd_sel_reg
        io.rs1_sel_out := rs1_sel_reg
        io.rs2_sel_out := rs2_sel_reg
        io.func3_out := func3_reg
        io.func7_out := func7_reg

        io.ctrl_MemWr_out := ctrl_MemWr_reg
        io.ctrl_MemRd_out := ctrl_MemRd_reg
        io.ctrl_Branch_out := ctrl_Branch_reg
        io.ctrl_RegWr_out := ctrl_RegWr_reg
        io.ctrl_MemToReg_out := ctrl_MemToReg_reg
        io.ctrl_AluOp_out := ctrl_AluOp_reg
        io.ctrl_OpA_sel_out := ctrl_OpA_sel_reg
        io.ctrl_OpB_sel_out := ctrl_OpB_sel_reg
        io.ctrl_nextPc_sel_out := ctrl_nextPc_sel_reg
}
```

What we are doing in this pipeline register is that we are taking a number of inputs that will be useful for us in the future stages.

We take **IF_ID_pc** as input which will be the current pc value of the instruction coming directly from the IF/ID pipeline register. Storing the pc value of the current instruction will help us in later stages of the pipeline.

We take the **rs1_sel_in, rs2_sel_in** and **rd_sel_in** as inputs which are 5 bit values and will be coming from the instruction that tells us the register numbers that the instruction has. We also take r**s1_in** and **rs2_in** as inputs which will be coming from the register file output that provides us with the actual values in the registers that will be needed by the ALU in the future.

We also take the **imm** as input which will be coming from the immediate generation block and filtered by the mux to correctly send the immediate value of either the I-Type instruction, S-Type instruction and U-Type instruction. This immediate value is needed by the ALU for calculation. We do not pass the SB-Type and UJ-Type immediate values because they are branch and jump instructions and we do not need ALU for computing these immediates. The immediate generation block itself adds the current pc to the SB-Type immediate or to the UJ-Type immediate to calculate the next pc value for branching or jumping to the label.

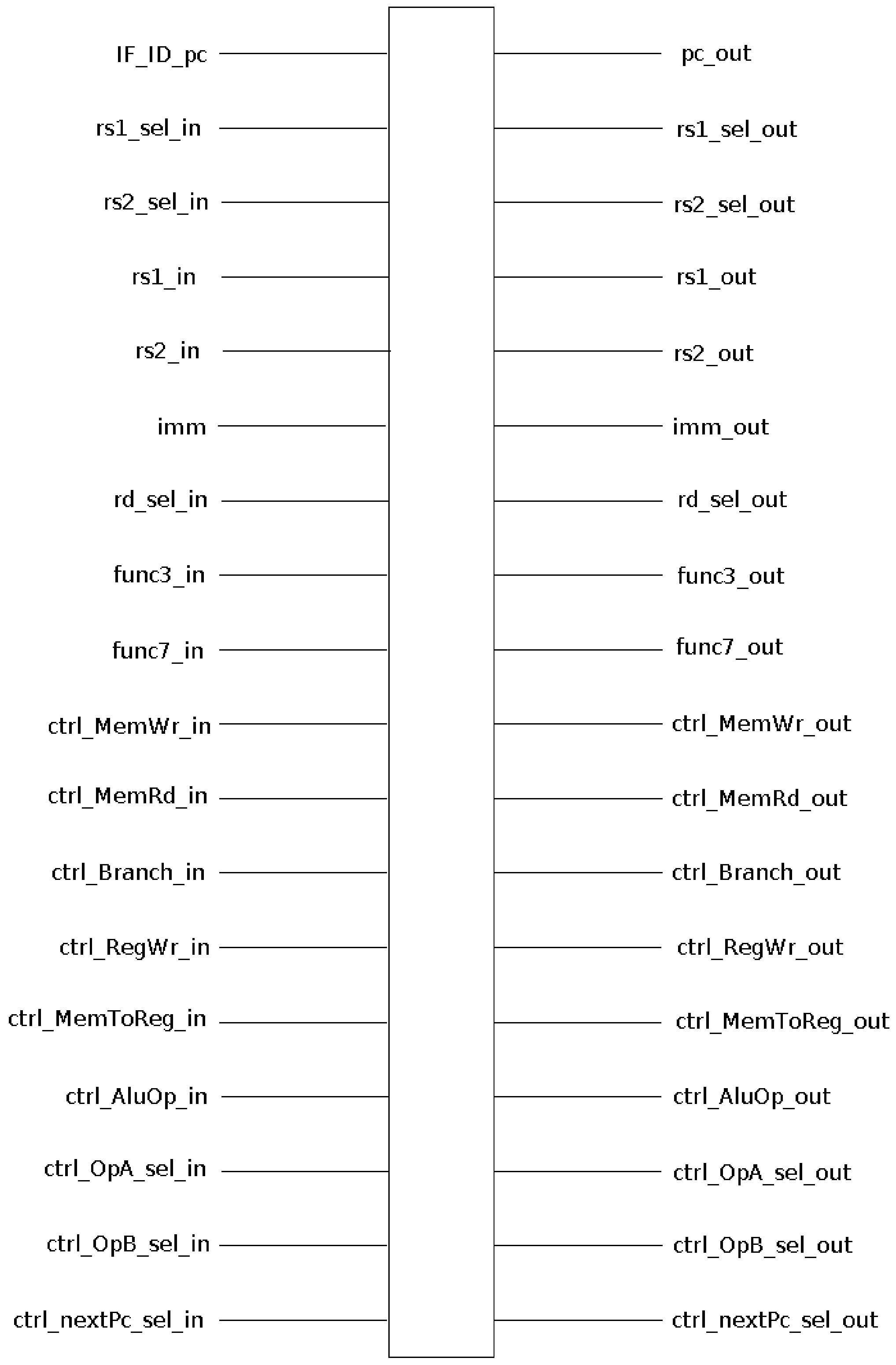
We take the **func3_in** and **func7_in** as inputs which will be coming from the instruction field that will in future go as inputs to the ALU Control which will control what operation the ALU will have to perform.

We also take all the Control pins as inputs which will come from the Control unit in the instruction decode stage that will in future tell other modules in the later stages as to what to do with the current value.

Note that we are saving the currently decoded values of the instruction in the next pipeline register and we are saying that it will be helpful in the future, this is true. Why? Because when we proceed from the decode stage to the next execution stage at the same time a new instruction will enter in the decode stage and all the values of the current instruction in the execution stage will be lost and overridden by the new instruction in the decode stage. How would the ALU know which values it has to process? This is why we were passing and storing these values in the pipeline register so that if the previous values get overridden by the new instruction, we have the values for the current instruction stored inside the pipeline register.

Block Diagram:

ID/EX



3) Coding the EX_MEM Pipeline Register

```
    class EX_MEM extends Module {
     val io = IO(new Bundle {
val ID_EX_MEMWR = Input(UInt(1.W))
val ID_EX_MEMRD = Input(UInt(1.W))
```

```
        val ID_EX_REGWR = Input(UInt(1.W))
        val ID_EX_MEMTOREG = Input(UInt(1.W))
        val ID_EX_RS2 = Input(SInt(32.W))
        val ID_EX_RDSEL = Input(UInt(5.W))
        val ID_EX_RS2SEL = Input(UInt(5.W))

        val alu_output = Input(SInt(32.W))

        val ex_mem_memWr_out = Output(UInt(1.W))
        val ex_mem_memRd_out = Output(UInt(1.W))
        val ex_mem_regWr_out = Output(UInt(1.W))
        val ex_mem_memToReg_out = Output(UInt(1.W))
        val ex_mem_rs2_output = Output(SInt(32.W))
        val ex_mem_rdSel_output = Output(UInt(5.W))
        val ex_mem_rs2Sel_output = Output(UInt(5.W))
        val ex_mem_alu_output = Output(SInt(32.W))

    })

        val reg_memWr = RegInit(0.U(1.W))
        reg_memWr := io.ID_EX_MEMWR
        io.ex_mem_memWr_out := reg_memWr

        val reg_memRd = RegInit(0.U(1.W))
        reg_memRd := io.ID_EX_MEMRD
        io.ex_mem_memRd_out := reg_memRd

        val reg_regWr = RegInit(0.U(1.W))
        reg_regWr := io.ID_EX_REGWR
        io.ex_mem_regWr_out := reg_regWr

        val reg_memToReg = RegInit(0.U(1.W))
        reg_memToReg := io.ID_EX_MEMTOREG
        io.ex_mem_memToReg_out := reg_memToReg

        val reg_rs2 = RegInit(0.S(32.W))
        reg_rs2 := io.ID_EX_RS2
        io.ex_mem_rs2_output := reg_rs2

        val reg_rd_sel = RegInit(0.U(5.W))
        reg_rd_sel := io.ID_EX_RDSEL
        io.ex_mem_rdSel_output := reg_rd_sel

        val reg_rs2_sel = RegInit(0.U(5.W))
        reg_rs2_sel := io.ID_EX_RS2SEL
        io.ex_mem_rs2Sel_output := reg_rs2_sel

        val reg_alu_output = RegInit(0.S(32.W))
        reg_alu_output := io.alu_output
        io.ex_mem_alu_output := reg_alu_output
}
```

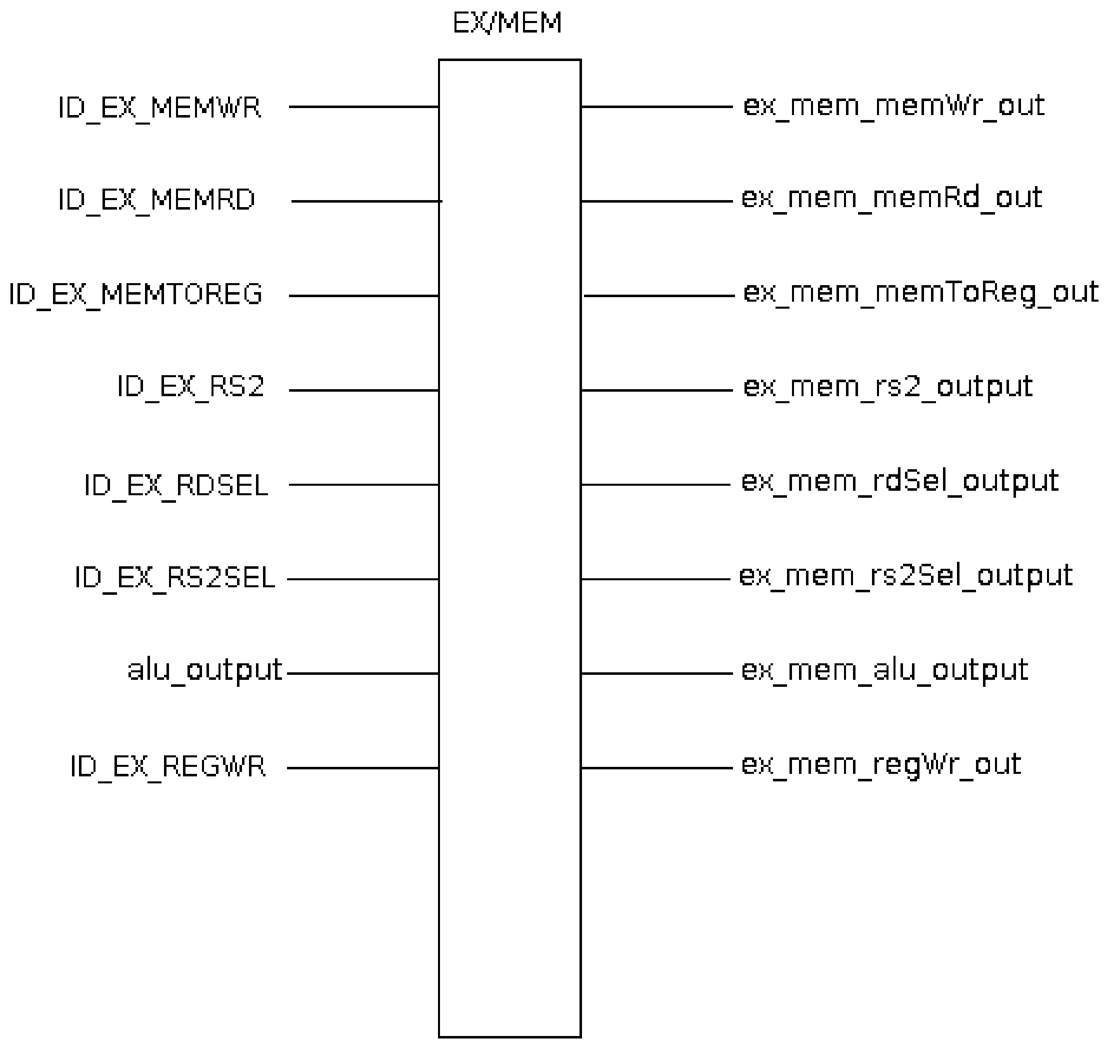
Here we take memory control signals from the ID/EX pipeline register which will be further required in the Memory stage for controlling the memory.

We take the ID_EX_RS2 as input from the execution stage that will be used in the memory stage for storing the value in the data memory in Memory stage for store type instructions.

We also take ID_EX_RDSEL as input for later stage when we get to the Write Back stage. So that when we write back the data we write it in the correct destination register identified by this input.

Other inputs like RS2 and RS2SEL will be used in future stages.

Block Diagram:

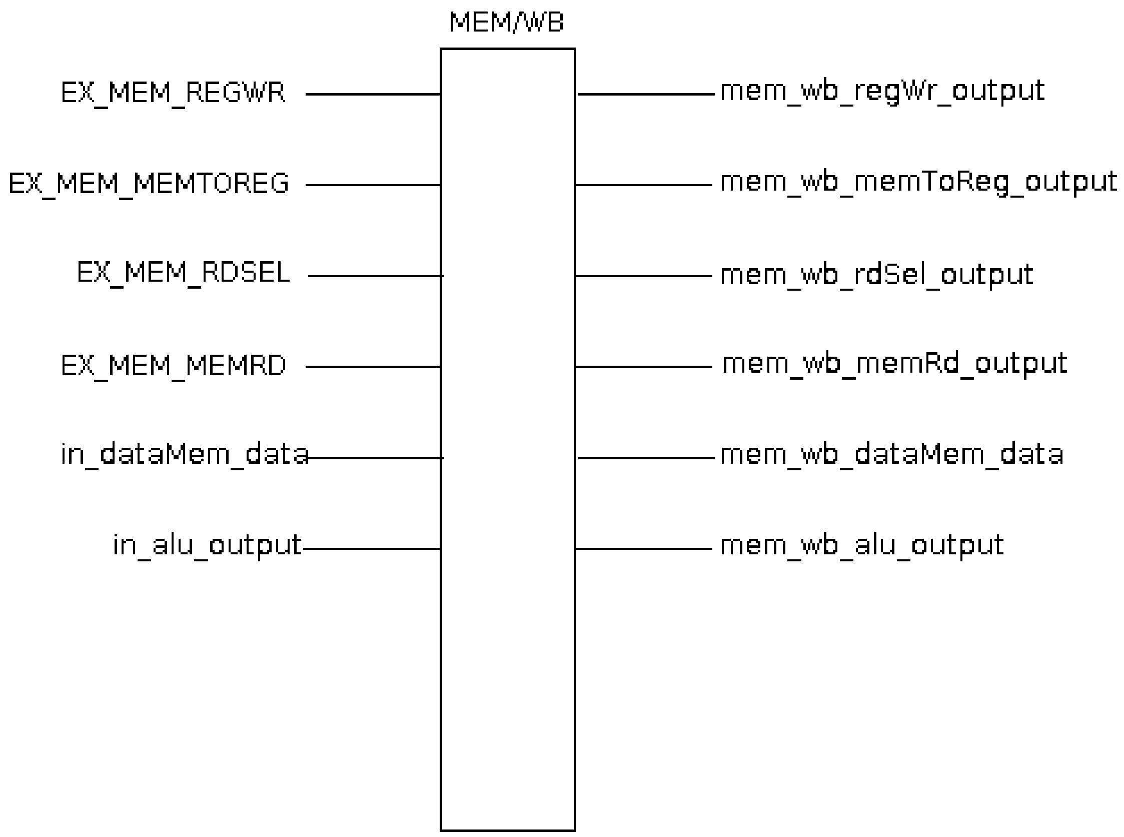


4)Coding the MEM_WB Pipeline Register

```
class MEM_WB extends Module {

val io = IO(new Bundle {
        val EX_MEM_REGWR = Input(UInt(1.W))
        val EX_MEM_MEMTOREG = Input(UInt(1.W))
        val EX_MEM_RDSEL = Input(UInt(5.W))
        val EX_MEM_MEMRD = Input(UInt(1.W))
        val in_dataMem_data = Input(SInt(32.W))
        val in_alu_output = Input(SInt(32.W))
        val mem_wb_regWr_output = Output(UInt(1.W))
        val mem_wb_memToReg_output = Output(UInt(1.W))
        val mem_wb_memRd_output = Output(UInt(1.W))
        val mem_wb_rdSel_output = Output(UInt(5.W))
        val mem_wb_dataMem_data = Output(SInt(32.W))
        val mem_wb_alu_output = Output(SInt(32.W))
    })

          val reg_regWr = RegInit(0.U(1.W))
          reg_regWr := io.EX_MEM_REGWR
          io.mem_wb_regWr_output := reg_regWr

          val reg_memToReg = RegInit(0.U(1.W))
          reg_memToReg := io.EX_MEM_MEMTOREG
          io.mem_wb_memToReg_output := reg_memToReg

          val reg_memRd = RegInit(0.U(1.W))
          reg_memRd := io.EX_MEM_MEMRD
          io.mem_wb_memRd_output := reg_memRd

          val reg_rdSel = RegInit(0.U(5.W))
          reg_rdSel := io.EX_MEM_RDSEL
          io.mem_wb_rdSel_output := reg_rdSel

          val reg_dataMem_data = RegInit(0.S(32.W))
          reg_dataMem_data := io.in_dataMem_data
          io.mem_wb_dataMem_data := reg_dataMem_data

          val reg_alu_output = RegInit(0.S(32.W))
          reg_alu_output := io.in_alu_output
          io.mem_wb_alu_output := reg_alu_output
}
```

Here we take register write, memory to register and rd_sel as inputs. Which will be used in the Write Back stage for selecting either the data from the memory to be written or the data from the ALU to be written in the register specified by the rd_sel input.

Block Diagram:



Connecting the pipeline registers in Top

Instruction Fetch Stage:

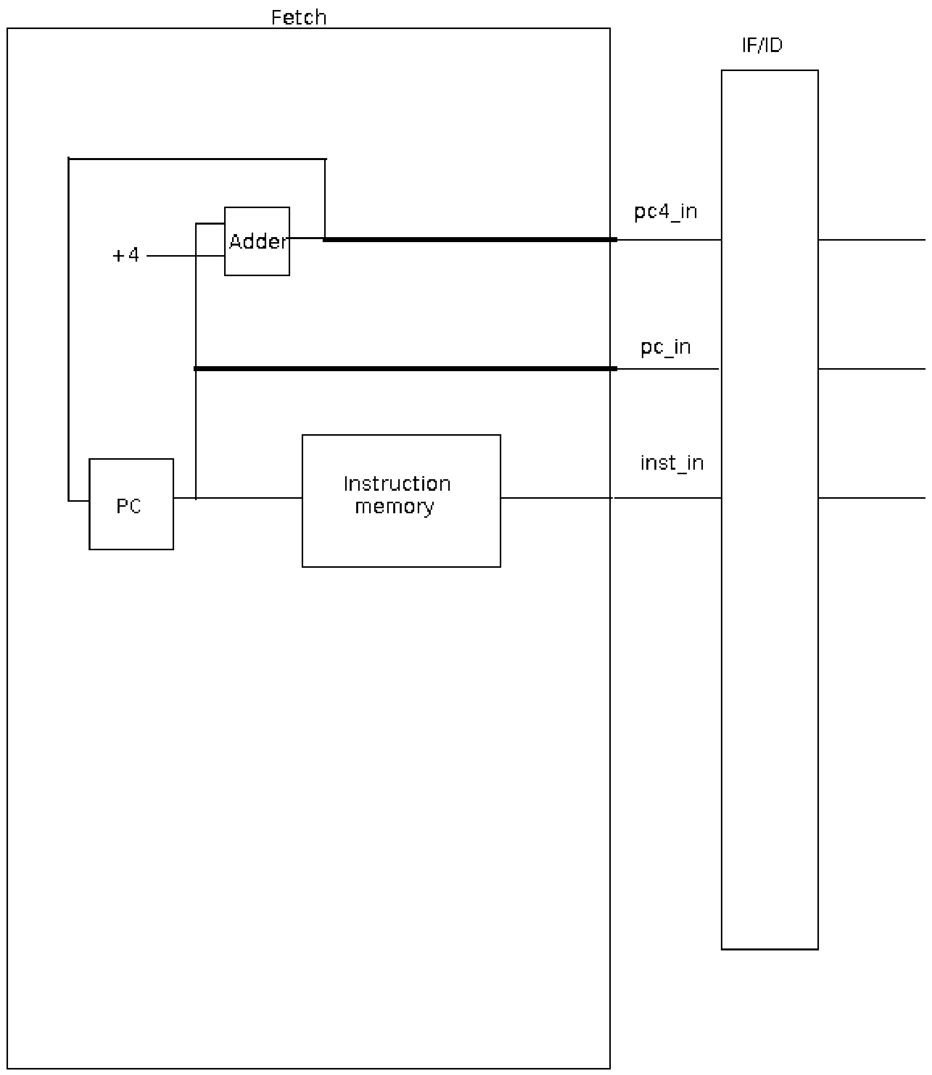
```
// *********** ----------- INSTRUCTION FETCH (IF) STAGE ----------- ********* //

(1)imem.io.wrAddr := pc.io.out(11,2).asUInt
(2)IF_ID.io.pc_in := pc.io.out
```

(3)`IF_ID.io.pc4_in := pc.io.pc4`
(4)`IF_ID.io.inst_in := imem.io.readData`
(5)`pc.io.in := pc.io.pc4`

In the instruction fetch stage we fetch the next instruction and feed it to the IF/ID Pipeline register.

(1) We wire the instruction memory address with the pc output.
(2) The current pc input of the IF/ID pipeline register is wired with the output of the pc.
(3) The next instruction pc input (pc+4) of the IF/ID pipeline register is wired with the pc +4 out of pc
(4) The instruction input of the IF/ID pipeline register is wired with the instruction memory output data
(5) PC's input value is wired with the next instruction that is PC + 4

Block Diagram:



Instruction Decode Stage:

```
// *********** ----------- INSTRUCTION DECODE (ID) STAGE ----------- ********* //
```

(1)
`control.io.in_opcode := IF_ID.io.inst_out(6, 0)`

(2)

```
reg_file.io.rs1_sel := IF_ID.io.inst_out(19, 15)
reg_file.io.rs2_sel := IF_ID.io.inst_out(24, 20)
reg_file.io.regWrite := MEM_WB.io.mem_wb_regWr_output
reg_file.io.rd_sel := MEM_WB.io.mem_wb_rdSel_output
```

(3)

```
imm_generation.io.instruction := IF_ID.io.inst_out
imm_generation.io.pc := IF_ID.io.pc_out
```

(4)

```
ID_EX.io.IF_ID_pc := IF_ID.io.pc_out
ID_EX.io.IF_ID_pc4 := IF_ID.io.pc4_out
ID_EX.io.func3_in := IF_ID.io.inst_out(14,12)
ID_EX.io.func7_in := IF_ID.io.inst_out(30)
ID_EX.io.rd_sel_in := IF_ID.io.inst_out(11,7)
ID_EX.io.rs1_sel_in := IF_ID.io.inst_out(19, 15)
ID_EX.io.rs2_sel_in := IF_ID.io.inst_out(24, 20)
ID_EX.io.rs1_in := reg_file.io.rs1
ID_EX.io.rs2_in := reg_file.io.rs2
```

(5)

```
ID_EX.io.ctrl_MemWr_in := control.io.out_memWrite
ID_EX.io.ctrl_MemRd_in := control.io.out_memRead
ID_EX.io.ctrl_Branch_in := control.io.out_branch
ID_EX.io.ctrl_RegWr_in := control.io.out_regWrite
ID_EX.io.ctrl_MemToReg_in := control.io.out_memToReg
ID_EX.io.ctrl_AluOp_in := control.io.out_aluOp
ID_EX.io.ctrl_OpA_sel_in := control.io.out_operand_a_sel
ID_EX.io.ctrl_OpB_sel_in := control.io.out_operand_b_sel
ID_EX.io.ctrl_nextPc_sel_in := control.io.out_next_pc_sel
```

(6)

```
   when(control.io.out_extend_sel === "b00".U) {
   // I-Type instruction
   ID_EX.io.imm := imm_generation.io.i_imm
} .elsewhen(control.io.out_extend_sel === "b01".U) {
   // S-Type instruction
   ID_EX.io.imm := imm_generation.io.s_imm
} .elsewhen(control.io.out_extend_sel === "b10".U) {
   // U-Type instruction
   ID_EX.io.imm := imm_generation.io.u_imm
} .otherwise {
   ID_EX.io.imm := 0.S(32.W)
}
```

In the decode stage the current instruction is decoded and is sent to the modules and their outputs are fed to the ID/EX pipeline register.

(1) Extracting the opcode from the instruction outputted from the IF/ID pipeline register and feeding it to the Control input.

(2) Extracting the rs1_sel, rs2_sel bits from the instruction and feeding it to the register file. Notice that we don't use the register write from the Control output but we use it from the MEM/WB output because this instruction will be written back after the Write Back stage so we will check there if the

register write is 1. Similar is the case with the rd_sel. We use the rd_sel from the MEM/WB pipeline register output, because the data will be written back at the Write Back stage. If we take rd_sel from the current instruction then the instruction which was in the Write Back stage will have conflict over it's rd_sel.

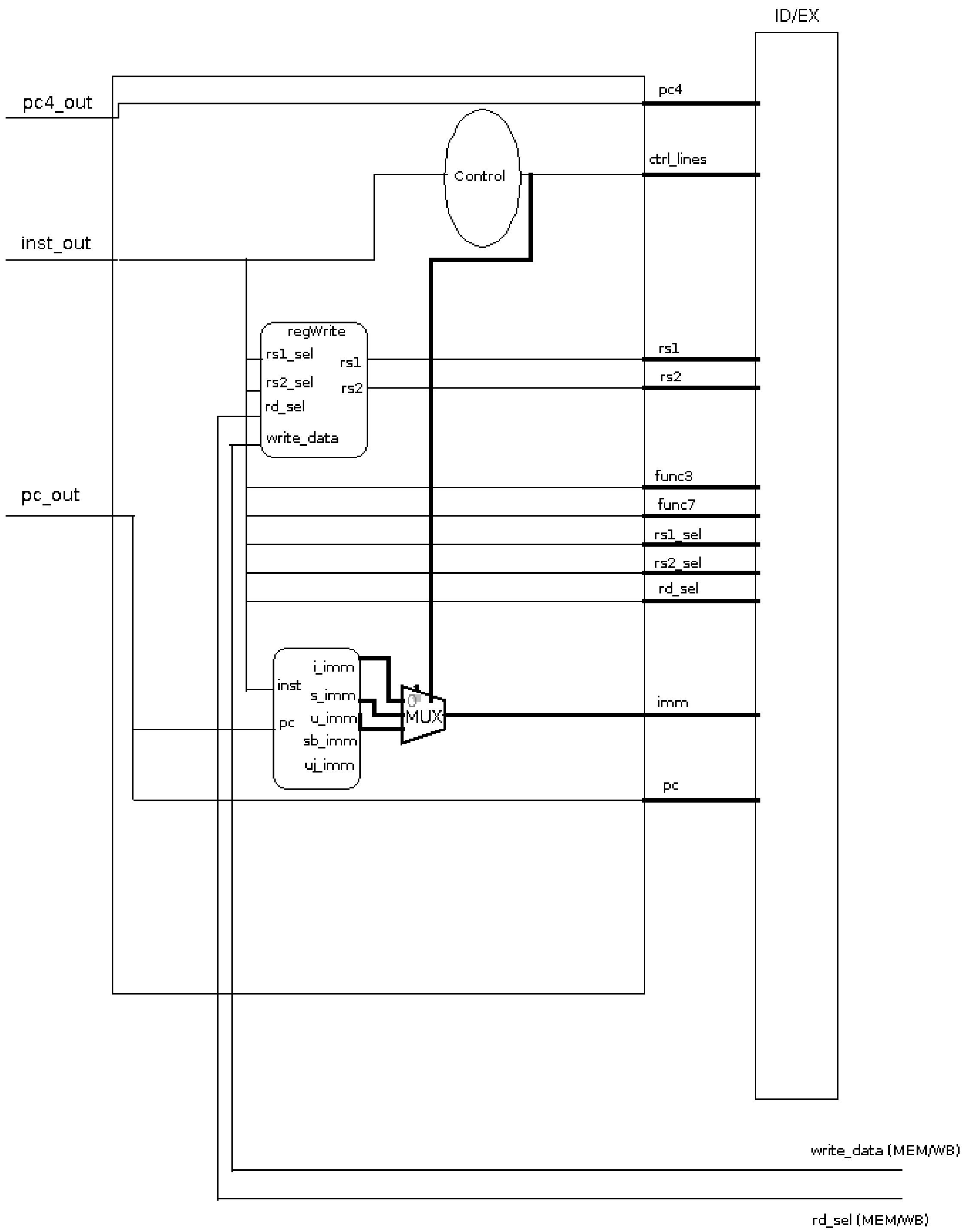
(3) We are feeding the current instruction and pc to the immediate generation.

(4) Passing the func3, func7, rs1_sel, rs2_sel, rd_sel, rs1_in, rs2_in, pc and pc4 (pc + 4) to the ID/EX pipeline register.

(5) Passing all the control signal forward to the ID/EX pipeline register.

(6) Selecting the correct immediate value from the immediate generation based on the Control's extend_sel pin to forward it to the ID/EX pipeline register.

Block Diagram:



Execute Stage:

```
// *********** ----------- EXECUTION (EX) STAGE ----------- ********* //
```

(1)

```
alu.io.oper_a := ID_EX.io.rs1_out
```

(2)

```
when(ID_EX.io.ctrl_OpB_sel_out === "b1".U) {
 alu.io.oper_b := ID_EX.io.imm_out
 } .otherwise {
 alu.io.oper_b := ID_EX.io.rs2_out
 }
```

(3)

```
alu.io.aluCtrl := alu_control.io.output
```

(4)

```
alu_control.io.aluOp := ID_EX.io.ctrl_AluOp_out
alu_control.io.func7 := ID_EX.io.func7_out
alu_control.io.func3 := ID_EX.io.func3_out
```

(5)

```
EX_MEM.io.alu_output := alu.io.output
EX_MEM.io.ID_EX_RDSEL := ID_EX.io.rd_sel_out
EX_MEM.io.ID_EX_RS2SEL := ID_EX.io.rs2_sel_out
EX_MEM.io.ID_EX_MEMWR := ID_EX.io.ctrl_MemWr_out
EX_MEM.io.ID_EX_MEMRD := ID_EX.io.ctrl_MemRd_out
EX_MEM.io.ID_EX_REGWR := ID_EX.io.ctrl_RegWr_out
EX_MEM.io.ID_EX_MEMTOREG := ID_EX.io.ctrl_MemToReg_out
```

(6)

```
EX_MEM.io.ID_EX_RS2 := ID_EX.io.rs2_out
```

(1) Connecting the operand a of ALU with the ID/EX rs1_out of pipeline register

(2) Connecting the operand b of ALU either with the immediate output of ID/EX pipeline register or with the rs2_out of the pipeline register depending on the operand_b_sel output from the ID/EX pipeline register.

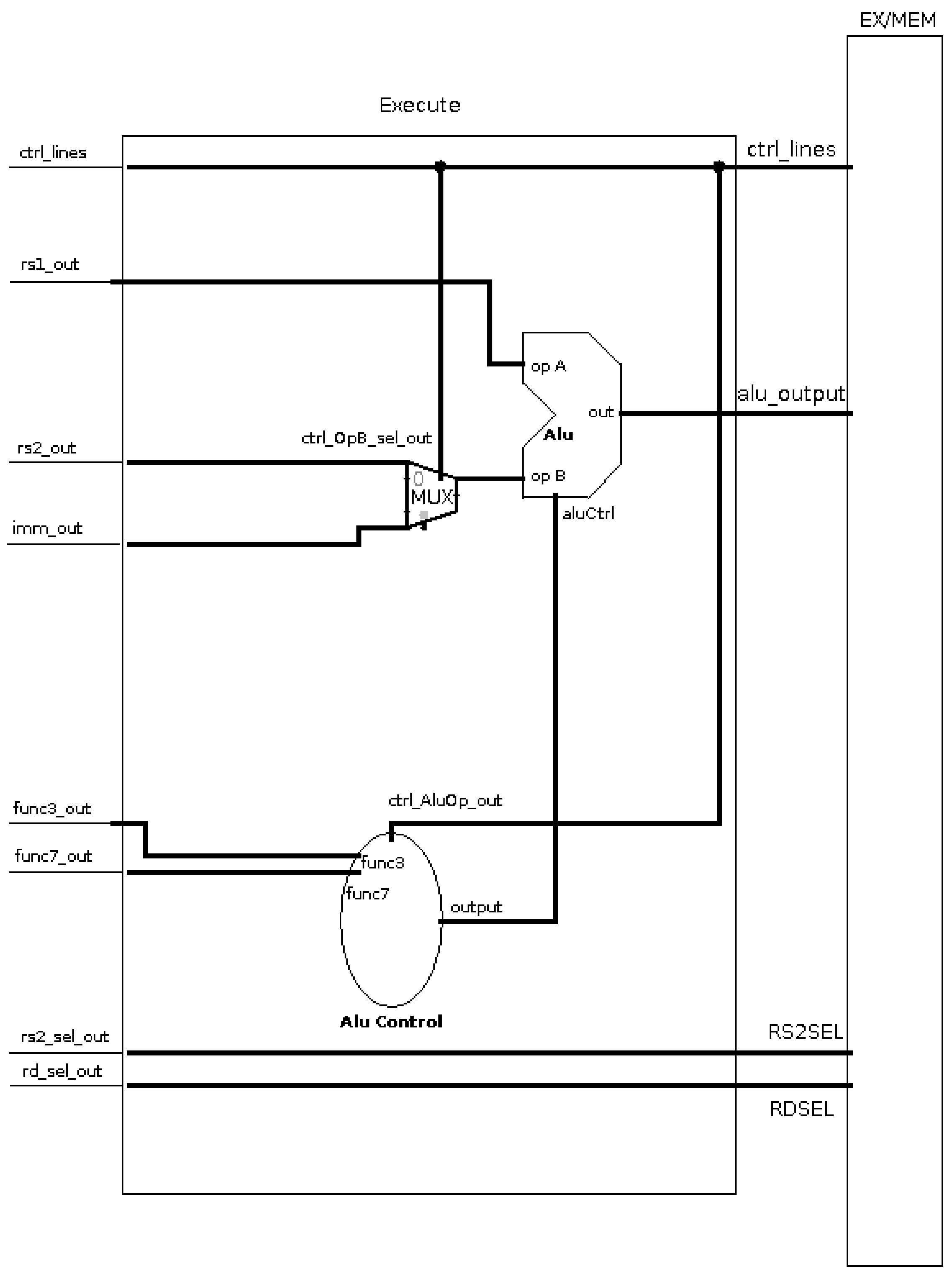
(3) Connecting the ALU control input of the ALU which controls what operation the ALU must perform with the output of the ALU Control module.

(4) Feeding the inputs to the ALU Control from the ID/EX pipeline register outputs.

(5) Sending the ALU output to the EX/MEM register along with other control signals.

(6) Send the data in source register 2 (rs2) to the EX/MEM register to be used later in the Memory stage for storing the data in the memory in case of Store instruction.

Block Diagram:



Memory Stage:

```
// *********** ----------- MEMORY (MEM) STAGE ----------- ********* //
```
(1)
```
MEM_WB.io.in_alu_output := EX_MEM.io.ex_mem_alu_output
```
(2)
```
MEM_WB.io.in_dataMem_data := dmem.io.memOut
```
(3)
```
MEM_WB.io.EX_MEM_RDSEL := EX_MEM.io.ex_mem_rdSel_output
MEM_WB.io.EX_MEM_REGWR := EX_MEM.io.ex_mem_regWr_out
MEM_WB.io.EX_MEM_MEMRD := EX_MEM.io.ex_mem_memRd_out
MEM_WB.io.EX_MEM_MEMTOREG := EX_MEM.io.ex_mem_memToReg_out
```
(4)
```
dmem.io.memAddress := EX_MEM.io.ex_mem_alu_output(11, 2).asUInt
dmem.io.memWrite := EX_MEM.io.ex_mem_memWr_out
dmem.io.memRead := EX_MEM.io.ex_mem_memRd_out
dmem.io.memData := EX_MEM.io.ex_mem_rs2_output
```

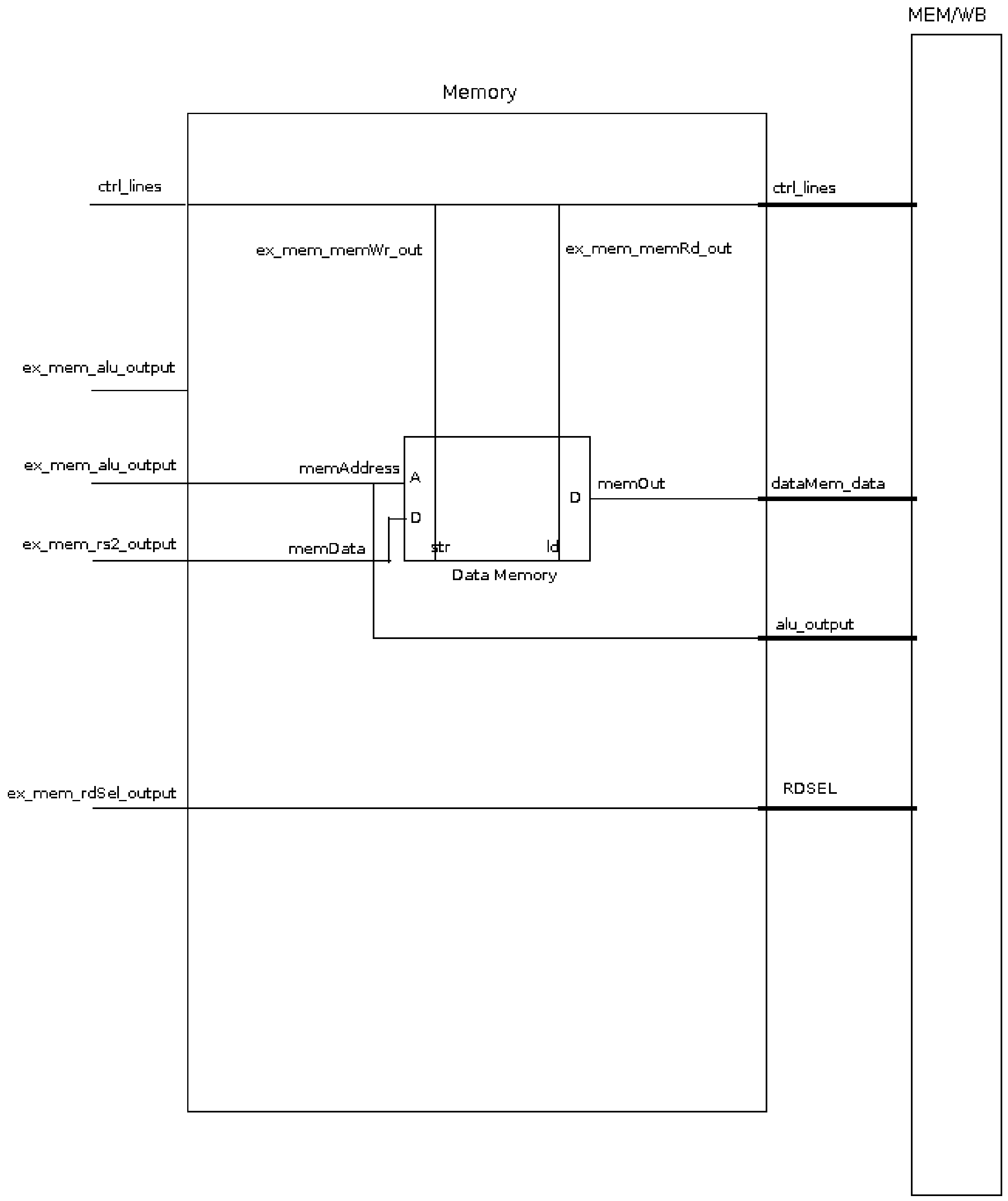
(1) Passing the ALU output from the EX/MEM pipeline register to the MEM/WB pipeline register

(2) Passing the data from the data memory to the MEM/WB pipeline register

(3) Passing the control signals from the EX/MEM pipeline register to the MEM/WB pipeline register

(4) Wiring the data memory. Connecting the address input of data memory from the alu output since for Load and Store ALU computes the address for the instruction. The memWrite and memRead control signals are wired from the EX/MEM pipeline register control outputs. Data that will be stored in the memory is wired with the rs2 output of the EX/MEM pipeline register since store uses rs2 value to store in the data memory.

Block Diagram:

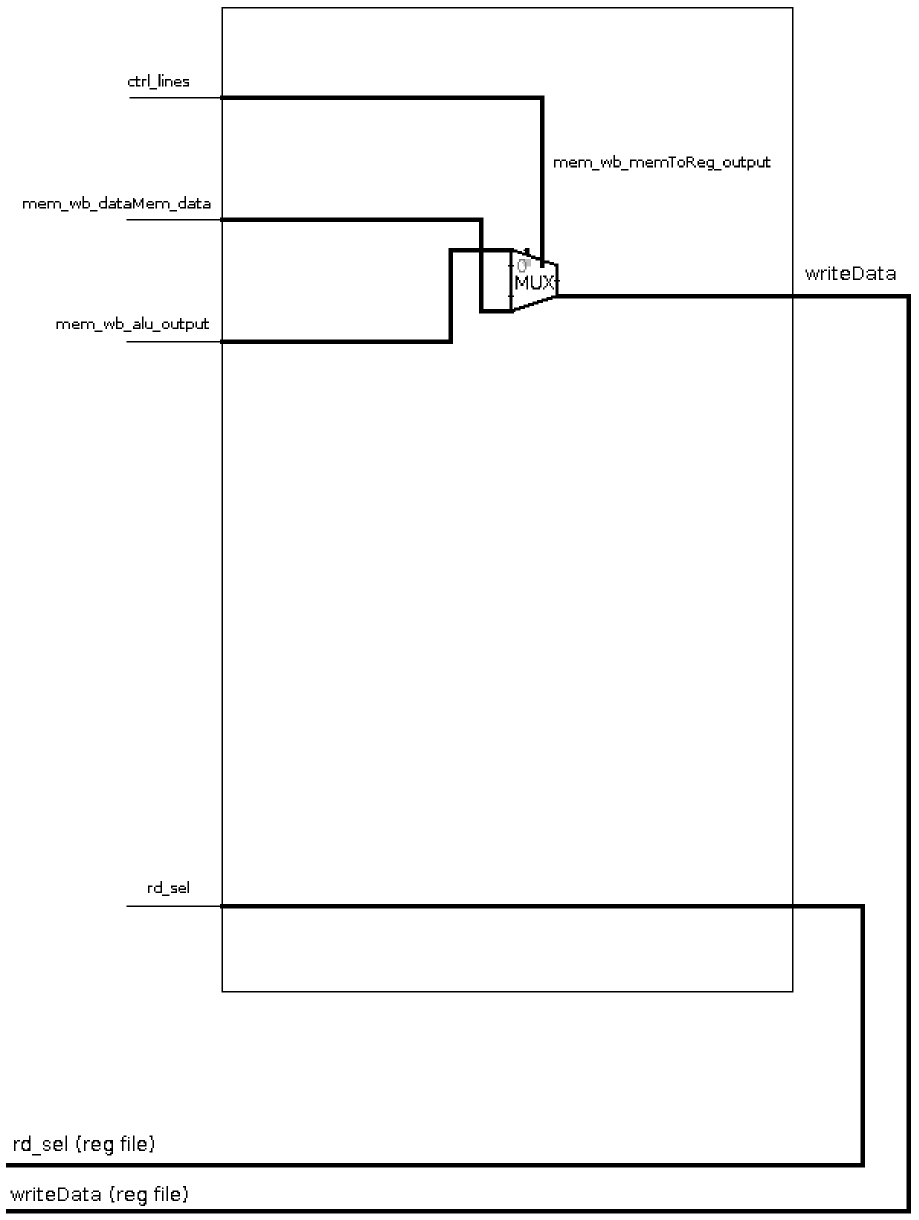


Write Back Stage:

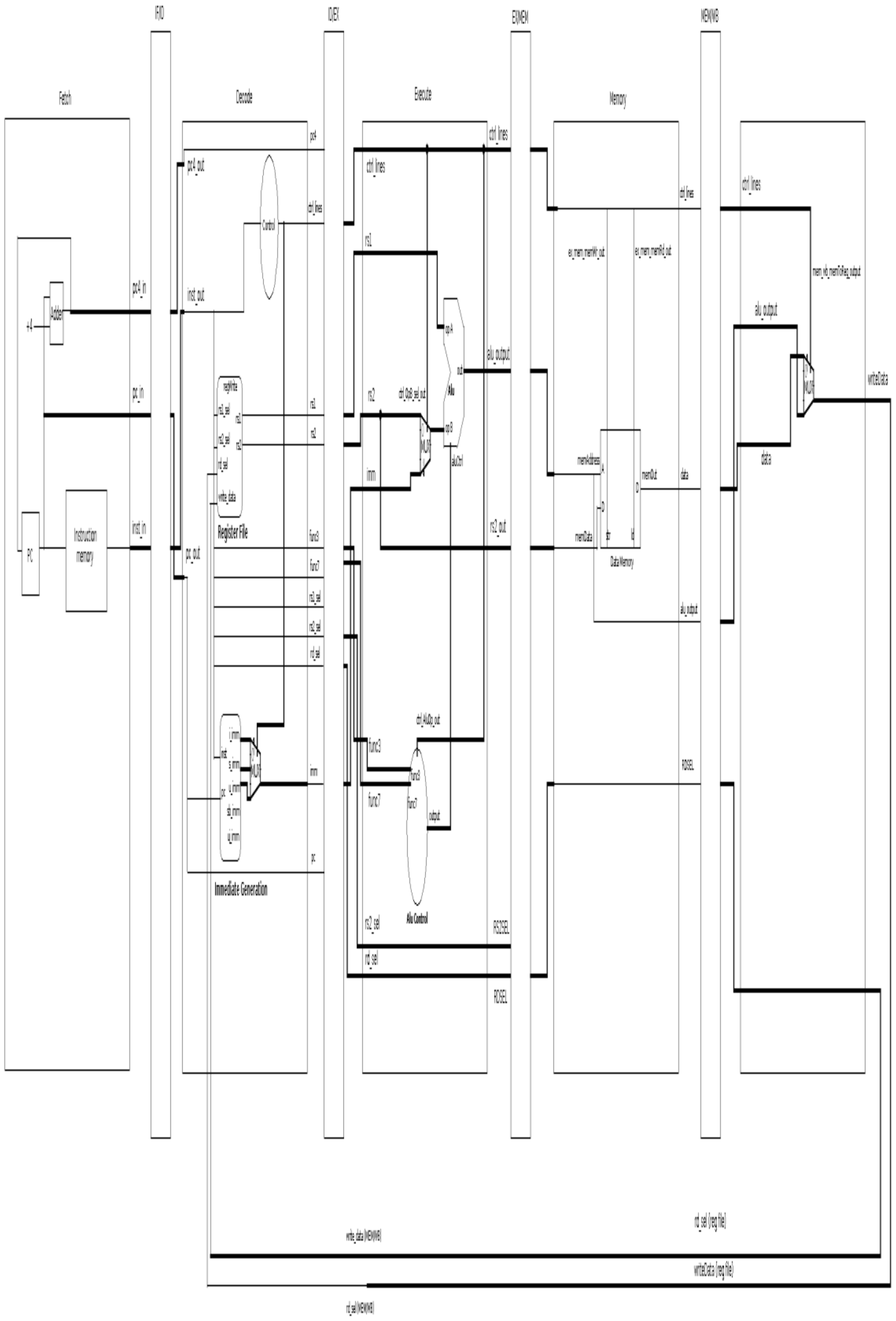
```
// *********** ----------- WRITE BACK (WB) STAGE ----------- ********* //


when(MEM_WB.io.mem_wb_memToReg_output === "b1".U) {
   reg_file.io.writeData := MEM_WB.io.mem_wb_dataMem_data
} .otherwise {
   reg_file.io.writeData := MEM_WB.io.mem_wb_alu_output
}
```

Here we check the memToReg control signal value from MEM/WB pipeline register which selects whether the data written in the register file is from the ALU output or from the data memory data.

Block Diagram:



CompleteDiagram:



Hazards in Pipelining

Pipelining though improves throughput of the processor but introduces new hazards in the datapath. These hazards arise due to the parallel execution of instructions in the datapath. We are breaking the promise of the ISA (Instruction Set Architecture). ISA tells us that each instruction executes sequentially and it is made that way, thus we had no hazards in the Single Cycle datapath where each instruction executed sequentially meaning that once the first instruction executes and is written back then only the next instruction is processed. But to improve our processor we used the pipelining methodology.

Let’s discuss what these hazards are and how can they occur.

There are a total of three types of hazards:

1) Structural Hazards

2) Data Hazards

3) Control Hazards

1) Structural Hazards:

Structural hazards arrive when a single datapath hardware module is needed by two or more instructions at the same time. Since we are executing instructions in parallel there might arise structural hazards. For example, consider one instruction is in the write back phase and is writing into the register file while another instruction is in the decode stage and is reading the register file. Since both operations are accessing the register file at the same time there is a structural hazard and if we do not resolve it our data will corrupt.

2) Data Hazards:

Data hazards arrive when an instruction needs some data and it is not ready yet. This is the most common case in pipelined datapaths. For example we have these instructions:

```
1        addi x2, x0, 5

2        add x3, x2, x0
```

The result of this instruction in our pipelined datapath will be that x3 will have 0 value written into it.

Why is that the case?

Let’s pass these instructions from the stages in the datapath and analyze what happens.

	CC1	CC2	CC3	CC4	CC5	CC6
1. addi x2, x0, 5	IM ||	REG ||	ALU ||	DM ||	REG	
2. add x3, x2, x0		IM ||	REG ||	ALU ||	DM ||	REG

The chart or graph above is a common pattern to visualize the flow of our instructions and how the instructions are being executed at each clock cycle. The top column names CC1, CC2 … stands for Clock Cycle. IM stands for Instruction Memory, REG stands for Register File, ALU is Arithmetic Logic Unit, DM is Data Memory.

As we can see when our above program initiates during the CC1 our first instruction is fetched from the instruction memory and is present on the input of the IF/ID pipeline register. At this time instruction 2 is in the instruction memory but is not *fetched* yet.

	CC1	CC2	CC3	CC4	CC5	CC6
1. addi x2, x0, 5	IM ||	REG ||	ALU ||	DM ||	REG	
2. add x3, x2, x0		IM ||	REG ||	ALU ||	DM ||	REG

In the CC2 our instruction 1 is outputted from the IF/ID pipeline register and is in the decode stage. It is reading the register file for it's operands. At the same time instruction 2 is being fetched from the instruction memory. After this clock cycle instruction 1 is decoded and its values are at the input of the ID/EX pipeline register. Instruction 2 is fetched and is at the input of IF/ID pipeline register.

	CC1	CC2	CC3	CC4	CC5	CC6
1. addi x2, x0, 5	IM ||	REG ||	ALU ||	DM ||	REG	
2. add x3, x2, x0		IM ||	REG ||	ALU ||	DM ||	REG

In CC3, instruction 1 is in the ALU and it's output is being calculated while instruction 2 is in the decode stage and is reading the register file to get it's operands.

	CC1	CC2	CC3	CC4	CC5	CC6
1. addi x2, x0, 5	IM ||	REG ||	ALU ||	DM ||	REG	
2. add x3, x2, x0		IM ||	REG ||	ALU ||	DM ||	REG

In CC4, instruction 1 is in the Memory stage, since it is an R-Type instruction it will not be doing anything at this stage. Instruction 2 is performing the calculation of it's operands. Now here is the data hazard. If you notice instruction 1 is not written back yet to the register file and instruction 2 is using instruction 1's destination register x2 as it's source register. The data isn't ready yet, addi x3, **x2**, 0 will have x2 as **0** not **5** because the first instruction has not yet written back to the register file.

We can resolve these hazards with a concept called *forwarding*. We will look into it at a later stage.

3) Control Hazards:

Control hazards are related to branching and jumping instructions. Typically your branch is calculated at the ALU stage. The ALU compares the two register operands and then update the pc with the label immediate value. By looking at the chart above we can visualize that during CC3 our first instruction will be in the ALU stage. If that instruction is an SB-Type instruction (Branch), and if the branch is taken then we have to update the pc value and jump to that address. Which means the next two instructions that will be in the Decode and Fetch stages will have to be flushed out because we do not want to execute those instructions since our branch is taken.

Removing Data Hazards

Let's look at how we can remove data hazards from our pipeline. From our discussion above we saw that data hazards occur when *an instruction's source operands either rs1 or rs2 are dependent on previous instruction's destination register.* We can resolve this hazard with *forwarding*. Forwarding is simply passing the value wherever it is needed. In the example of data hazard above, instruction 1's result is saved in the EX/MEM pipeline register while instruction 2 is in the execution stage during CC4. What if we forward the ALU output from the EX/MEM pipeline register to the input of ALU for the next instruction so that it can get the value of x2 directly from there. This is how forwarding is implemented.

So now for forwarding, the inputs of our ALU will have additional muxes. These muxes will select whether the input to the ALU is from the register file output of the ID/EX pipeline register (if there is no hazard) or from the ALU output of the EX/MEM pipeline register (in case of forwarding). We will need an additional hardware module which detects if there is a data hazard and controls the mux at the inputs of the ALU to pass the correct value to the ALU. Our forwarding unit will be in the Execution stage because this is where it needs to forward the correct value (into the ALU).

Coding the Forward Unit

```
class ForwardUnit extends Module {
   val io = IO(new Bundle {
      val EX_MEM_REGRD = Input(UInt(5.W))
      val ID_EX_REGRS1 = Input(UInt(5.W))
      val ID_EX_REGRS2 = Input(UInt(5.W))
      val EX_MEM_REGWR = Input(UInt(1.W))
      val forward_a = Output(UInt(2.W))
      val forward_b = Output(UInt(2.W))
   })

   (1)
   io.forward_a := "b00".U
   io.forward_b := "b00".U
```

```
// EX HAZARD
   (2)
  when(io.EX_MEM_REGWR === "b1".U && io.EX_MEM_REGRD =/= "b00000".U && (io.EX_MEM_REGRD
              === io.ID_EX_REGRS1) && (io.EX_MEM_REGRD === io.ID_EX_REGRS2)) {
        io.forward_a := "b01".U
              io.forward_b := "b01".U
  (3)
  } .elsewhen(io.EX_MEM_REGWR === "b1".U && io.EX_MEM_REGRD =/= "b00000".U &&
              (io.EX_MEM_REGRD === io.ID_EX_REGRS2)) {

              io.forward_b := "b01".U
  (4)
  } .elsewhen(io.EX_MEM_REGWR === "b1".U && io.EX_MEM_REGRD =/= "b00000".U &&
              (io.EX_MEM_REGRD === io.ID_EX_REGRS1)) {

              io.forward_a := "b01".U

  }
```

First let's discuss the inputs and outputs of our module.

EX_MEM_REGRD is the destination register number in 5 bits that will tell us the destination register of the instruction that is executed and is in the Memory stage.

ID_EX_REGRS1 is the source register number in 5 bits that are coming from the ID/EX pipeline register from the register file for the current instruction.

ID_EX_REGRS2 is the second source register number in 5 bits that are coming from the ID/EX pipeline register from the register file for the current instruction.

EX_MEM_REGWR is the control signal that will tell that the instruction which is in the Memory stage on which our current instruction is depended on, does it write to the register file or not. If EX_MEM_REGWR is 1 (True) then it means that the previous instruction will eventually write data into the register file after it passes through all stages.

forward_a is the output of our forwarding unit module that will control the mux at the operand A input of the ALU.

forward_b is the output of our forwarding unit module that will control the mux at the operand B input of the ALU.

Now let's discuss the working of this module

(1) By default we wire the forward a and forward b outputs to 0. So eventually if there is no hazard their values will not be updated and when forward a and forward b will be 0 we will pass the register file output from ID/EX pipeline register to the ALU inputs.

(2) Now we start with the conditional statements to detect whether we have a hazard or not. Here we check if the previous instruction intends to write to the register file in the future by checking the REGWR value, then we see if the destination register of the previous instruction was not 0. If it was 0 then it means the programmer doesn't intend to save the value in the destination register and so we should not forward it's value to our current dependent instruction and then we check if the destination register number is equivalent to both of the source register number of the current instruction , if all these conditions are true we forward the result of the EX/MEM alu output to both the inputs of the ALU.

(3) This condition is same as above the only difference is that we are checking for source 2 register of the instruction and updating the forward b value which will now select the value from EX/MEM ALU output to the Operand B of the ALU.

(4) This condition is also the same and here we are checking for source 1 register of the instruction and updating the forward a value which will now select the value from EX/MEM ALU output to the Operand A of the ALU.

Examples

1)

```
addi x2, x0, 2
add x3, x0, x2
```

Here condition (3) will be active as described above. The source register 2 is dependent on previous instruction's destination register.

2)

```
addi x2, x0, 10
add x5, x2, x0
```

Here condition (4) will be active as described above. The source register 1 is dependent on previous instruction's destination register.

3)

```
addi x2, x0, 20
add x3, x2, x2
```

Here condition (2) will be active as described above. The source register 1 and source register 2 both are depend on the previous instruction's destination register.

With these set of instructions our forwarding will be working pretty fine. But what if an instruction is dependent on an instruction that is 2 stages ahead it? Let’s visualize the instructions with our graph and see where the problem still exists.

	CC1	CC2	CC3	CC4	CC5	CC6	CC7
Addi x2,x0,10	IM	REG	ALU	DM	REG		
Addi x3,x0,5		IM	REG	ALU	DM	REG	
Add x4,x2,x0			IM	REG	ALU	DM	REG

Let’s look at the highlighted clock cycle 5 above. Right now instruction 1 is in Write Back stage but has *not yet written back to the register file*. The instruction 3 which is dependent on instruction 1 is in execution stage. At this point we cannot forward the ALU output from the EX/MEM pipeline register because the ALU output will have the output of the second instruction and we want the output of the first instruction since we are dependent on that instruction. At this stage the **MEM/WB** pipeline register will have the value of the ALU output of the first instruction since it is placed between the memory and writeback stage *(We will send the EX/MEM ALU output to the MEM/WB.ALU_output input so that we can use it here for forwarding)*. So this time we need to forward the ALU output from the MEM/WB pipeline register not the EX/MEM pipeline register. Now we need to modify our forwarding unit to forward values from the Write Back stage as well.

Let’s look at the code for removing this hazard in forwarding unit.

```
class ForwardUnit extends Module {
    val io = IO(new Bundle {
        val EX_MEM_REGRD = Input(UInt(5.W))
        val MEM_WB_REGRD = Input(UInt(5.W))
        val ID_EX_REGRS1 = Input(UInt(5.W))
        val ID_EX_REGRS2 = Input(UInt(5.W))
        val EX_MEM_REGWR = Input(UInt(1.W))
        val MEM_WB_REGWR = Input(UInt(1.W))
        val forward_a = Output(UInt(2.W))
        val forward_b = Output(UInt(2.W))
    })
    (1)
    io.forward_a := "b00".U
    io.forward_b := "b00".U
(2)
when(io.MEM_WB_REGWR === "b1".U && io.MEM_WB_REGRD =/= "b00000".U && (io.MEM_WB_REGRD
=== io.ID_EX_REGRS1) && (io.MEM_WB_REGRD === io.ID_EX_REGRS2)) {

    io.forward_a := "b10".U
    io.forward_b := "b10".U
(3)
} .elsewhen(io.MEM_WB_REGWR === "b1".U && io.MEM_WB_REGRD =/= "b00000".U &&
        (io.MEM_WB_REGRD === io.ID_EX_REGRS2)) {

        io.forward_b := "b10".U
(4)
} .elsewhen(io.MEM_WB_REGWR === "b1".U && io.MEM_WB_REGRD =/= "b00000".U &&   &&
        (io.MEM_WB_REGRD === io.ID_EX_REGRS1)) {

        io.forward_a := "b10".U

}
```

These conditions are exactly the same as above for Execution stage hazard. The only change is that now for comparisons MEM/WB destination register is being used since now our previous instruction is in the Write Back stage while our current dependent instruction is in the Execution (can be seen in the table above). Moreover, we are also changing the forward a and forward b control pins to different values which will now forward the values from the MEM/WB ALU output to the correct input operands of ALU.

Here is the full code of our forwarding unit uptill now

```
class ForwardUnit extends Module {
    val io = IO(new Bundle {
        val EX_MEM_REGRD = Input(UInt(5.W))
        val ID_EX_REGRS1 = Input(UInt(5.W))
        val ID_EX_REGRS2 = Input(UInt(5.W))
        val EX_MEM_REGWR = Input(UInt(1.W))
        val forward_a = Output(UInt(2.W))
        val forward_b = Output(UInt(2.W))
    })

    io.forward_a := "b00".U
    io.forward_b := "b00".U

// EX HAZARD

    when(io.EX_MEM_REGWR === "b1".U && io.EX_MEM_REGRD =/= "b00000".U && (io.EX_MEM_REGRD
                    === io.ID_EX_REGRS1) && (io.EX_MEM_REGRD === io.ID_EX_REGRS2)) {
            io.forward_a := "b01".U
                    io.forward_b := "b01".U

    } .elsewhen(io.EX_MEM_REGWR === "b1".U && io.EX_MEM_REGRD =/= "b00000".U &&
                    (io.EX_MEM_REGRD === io.ID_EX_REGRS2)) {

                    io.forward_b := "b01".U

    } .elsewhen(io.EX_MEM_REGWR === "b1".U && io.EX_MEM_REGRD =/= "b00000".U &&
                    (io.EX_MEM_REGRD === io.ID_EX_REGRS1)) {

                    io.forward_a := "b01".U

    }

// MEM HAZARD
      when(io.MEM_WB_REGWR === "b1".U && io.MEM_WB_REGRD =/= "b00000".U &&
(io.MEM_WB_REGRD                === io.ID_EX_REGRS1) && (io.MEM_WB_REGRD === io.ID_EX_REGRS2)) {

                    io.forward_a := "b10".U
                    io.forward_b := "b10".U

            } .elsewhen(io.MEM_WB_REGWR === "b1".U && io.MEM_WB_REGRD =/= "b00000".U &&
                    (io.MEM_WB_REGRD === io.ID_EX_REGRS2)) {

                    io.forward_b := "b10".U

            } .elsewhen(io.MEM_WB_REGWR === "b1".U && io.MEM_WB_REGRD =/= "b00000".U &&   &&
                    (io.MEM_WB_REGRD === io.ID_EX_REGRS1)) {

                    io.forward_a := "b10".U

            }
}
```

Now let's look at another example using our graph to better understand its working

	CC1	CC2	CC3	CC4	CC5	CC6	CC7
Addi x2,x0,10	IM	REG	ALU	DM	REG		
Addi x2,x2,5		IM	REG	ALU	DM	REG	
Add x4,x2,x0			IM	REG	ALU	DM	REG

Here we have three instructions. Instruction 3's source register 1 is dependent on instruction 2's destination register as well as on instruction 1's destination register.

Our current **instruction 3** in clock cycle 5 is in execution stage and our forwarding unit will be active here. Our forward unit will first run the when conditions to check for the hazard in EX stage that we covered earlier. Here we can see that our current instruction is dependent on the recent previous instruction which will be in the Memory stage. So for now our forwarding unit will set the values of forward a and forward b as "01". But then we enter into the next block of when conditions which is for MEM stage hazards. And since our current instruction is also dependent on the instruction one step previous of the current instruction then MEM/WB destination register will be equal to the source register of this instruction so our forwarding unit will update the *forward a* and *forward b* control pins to value "10" which means in the end ALU will get the output from the MEM/WB ALU output. But as we can see we need a value from the most recent previous instruction. Here our current instruction will get x2 value 10 not 15 and we want the latest data that is 15. So we need to add one more condition in our MEM hazard conditonal statements. We need to make sure that

not*(EX/MEM.REGWR == 1, EX/MEM.REGRD != 0 and EX/MEM.REGRD == source register of instruction)*

This statement means that update the forward a and forward b values **only when EX hazard is *not* present.**

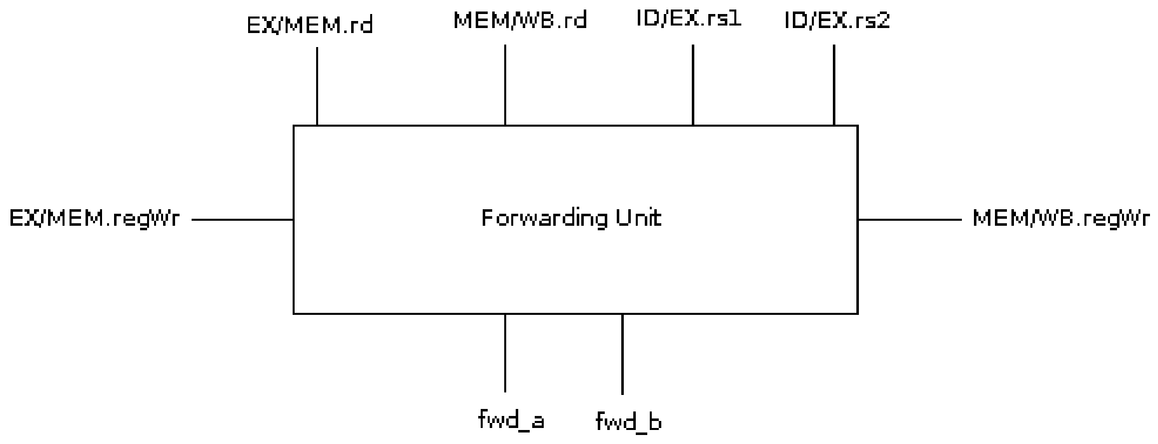
So that we get the latest value if our current instruction is dependent on both the previous instructions.

Let's review these updated conditional statements in our MEM hazard code.

```
when(io.MEM_WB_REGWR === "b1".U && io.MEM_WB_REGRD =/= "b00000".U &&
~((io.EX_MEM_REGWR === "b1".U) && (io.EX_MEM_REGRD =/= "b00000".U) && (io.EX_MEM_REGRD
=== io.ID_EX_REGRS1) && (io.EX_MEM_REGRD === io.ID_EX_REGRS2)) && (io.MEM_WB_REGRD ===
io.ID_EX_REGRS1) && (io.MEM_WB_REGRD === io.ID_EX_REGRS2)) {

        io.forward_a := "b10".U
        io.forward_b := "b10".U

} .elsewhen(io.MEM_WB_REGWR === "b1".U && io.MEM_WB_REGRD =/= "b00000".U &&
~((io.EX_MEM_REGWR === "b1".U) && (io.EX_MEM_REGRD =/= "b00000".U) && (io.EX_MEM_REGRD
=== io.ID_EX_REGRS2)) && (io.MEM_WB_REGRD === io.ID_EX_REGRS2)) {

        io.forward_b := "b10".U

} .elsewhen(io.MEM_WB_REGWR === "b1".U && io.MEM_WB_REGRD =/= "b00000".U &&
~((io.EX_MEM_REGWR === "b1".U) && (io.EX_MEM_REGRD =/= "b00000".U) && (io.EX_MEM_REGRD
=== io.ID_EX_REGRS2))  && (io.MEM_WB_REGRD === io.ID_EX_REGRS1)) {
   io.forward_a := "b10".U

}
```

Here the added condition is highlighted in color.

Block Diagram:



This completes our forwarding unit for data hazards. There may be cases when you cannot forward the data so our forwarding unit will not come to help. We will cover such conditions in the later stages. Let’s first see how our forwarding unit module will now be connected in the Top file.

Connecting the Forward Unit in Top

```
// *********** ----------- EXECUTION (EX) STAGE ----------- ********* //
```
(1)
```
forwardUnit.io.EX_MEM_REGRD := EX_MEM.io.ex_mem_rdSel_output
forwardUnit.io.MEM_WB_REGRD := MEM_WB.io.mem_wb_rdSel_output
forwardUnit.io.ID_EX_REGRS1 := ID_EX.io.rs1_sel_out
forwardUnit.io.ID_EX_REGRS2 := ID_EX.io.rs2_sel_out
forwardUnit.io.EX_MEM_REGWR := EX_MEM.io.ex_mem_regWr_out
forwardUnit.io.MEM_WB_REGWR := MEM_WB.io.mem_wb_regWr_output
```

(2)
```
// Controlling Operand A for ALU
when(forwardUnit.io.forward_a === "b00".U) {
  alu.io.oper_a := ID_EX.io.rs1_out
} .elsewhen(forwardUnit.io.forward_a === "b01".U) {
  alu.io.oper_a := EX_MEM.io.ex_mem_alu_output
} .elsewhen(forwardUnit.io.forward_a === "b10".U) {
  alu.io.oper_a := reg_file.io.writeData
} .otherwise {
  alu.io.oper_a := ID_EX.io.rs1_out
}
```

(3)
```
// Controlling Operand B for ALU
  when(ID_EX.io.ctrl_OpB_sel_out === "b1".U) {
    alu.io.oper_b := ID_EX.io.imm_out
```
(4)
```
   when(forwardUnit.io.forward_b === "b00".U) {
  EX_MEM.io.ID_EX_RS2 := ID_EX.io.rs2_out
 } .elsewhen(forwardUnit.io.forward_b === "b01".U) {
   EX_MEM.io.ID_EX_RS2 := EX_MEM.io.ex_mem_alu_output
 } .elsewhen(forwardUnit.io.forward_b === "b10".U) {
   EX_MEM.io.ID_EX_RS2 := reg_file.io.writeData
 } .otherwise {
   EX_MEM.io.ID_EX_RS2 := ID_EX.io.rs2_out
 }
}
```
(5)
```
 .otherwise {
  when(forwardUnit.io.forward_b === "b00".U) {
    alu.io.oper_b := ID_EX.io.rs2_out
    EX_MEM.io.ID_EX_RS2 := ID_EX.io.rs2_out
  } .elsewhen(forwardUnit.io.forward_b === "b01".U) {
    alu.io.oper_b := EX_MEM.io.ex_mem_alu_output
    EX_MEM.io.ID_EX_RS2 := EX_MEM.io.ex_mem_alu_output
  } .elsewhen(forwardUnit.io.forward_b === "b10".U) {
    alu.io.oper_b := reg_file.io.writeData
    EX_MEM.io.ID_EX_RS2 := reg_file.io.writeData
  } .otherwise {
    alu.io.oper_b := ID_EX.io.rs2_out
    EX_MEM.io.ID_EX_RS2 := ID_EX.io.rs2_out
  }
}
```

(1) First of all we wired all the inputs of the Forwarding unit that we needed. These inputs were all explained when coding the forwarding unit above. We wire them with the EX/MEM and MEM/WB pipeline outputs for forwarding and ID/EX pipeline source registers for comparing the current instruction source registers with the previous instruction that is in the Memory stage or in the Write Back stage.

(2) Here we are controlling the input to the operand a of ALU. We use our forwarding unit's output to control which value to forward. If forward a is 0 it means that the current instruction is not dependent on the instructions before it so we just wire it with the source registers output of the ID/EX pipeline register. If forward a is 01 it means there is a hazard in EX stage and so we forward the data from the EX/MEM ALU output to the operand a of the ALU. If forward a is 10 it means there is a hazard in the Write Back stage so we forward the data from there to the operand a of the ALU. We are forwarding the data from the register file's write data. Why is that so? Because remember our datapath writes back in the Write Back stage and it writes back either the data from the data memory in case of Load instructions or from the ALU output. There is a mux in Write Back stage that we saw that controls what is to be written. So eventually we also need to forward the correct data from the Write Back stage which can either be the data from the data memory or the output of the ALU. Since we get the filtered data from the mux in Write Back stage we just wire the register file's write data with the input of the ALU in case of forwarding. Note that we can also use a when statement here to filter out if we need the output from the MEM/WB ALU output or from the MEM/WB Data output. But we can just forward the filtered data already present at the register file's write data.
(3) Now we move towards controlling the operand b of the ALU. Remember, the operand b of ALU already has a mux which controls whether the input to the ALU is from the source register coming in from the ID/EX pipeline register or from the immediate value coming in from the ID/EX pipeline register. Here we are coding that mux. When operand_b_sel of the control is 1 it means that the immediate value needs to be forwarded.

(4) Here comes the interesting part. Uptill now we were using the forwarding conditions to send inputs to the ALU. But we need to forward the data to one more place as well and that is to the *register 2* input of the EX/MEM pipeline register. Why? Here is an example

```
addi x2, x0, 5
sw x2, 0(x0)
```

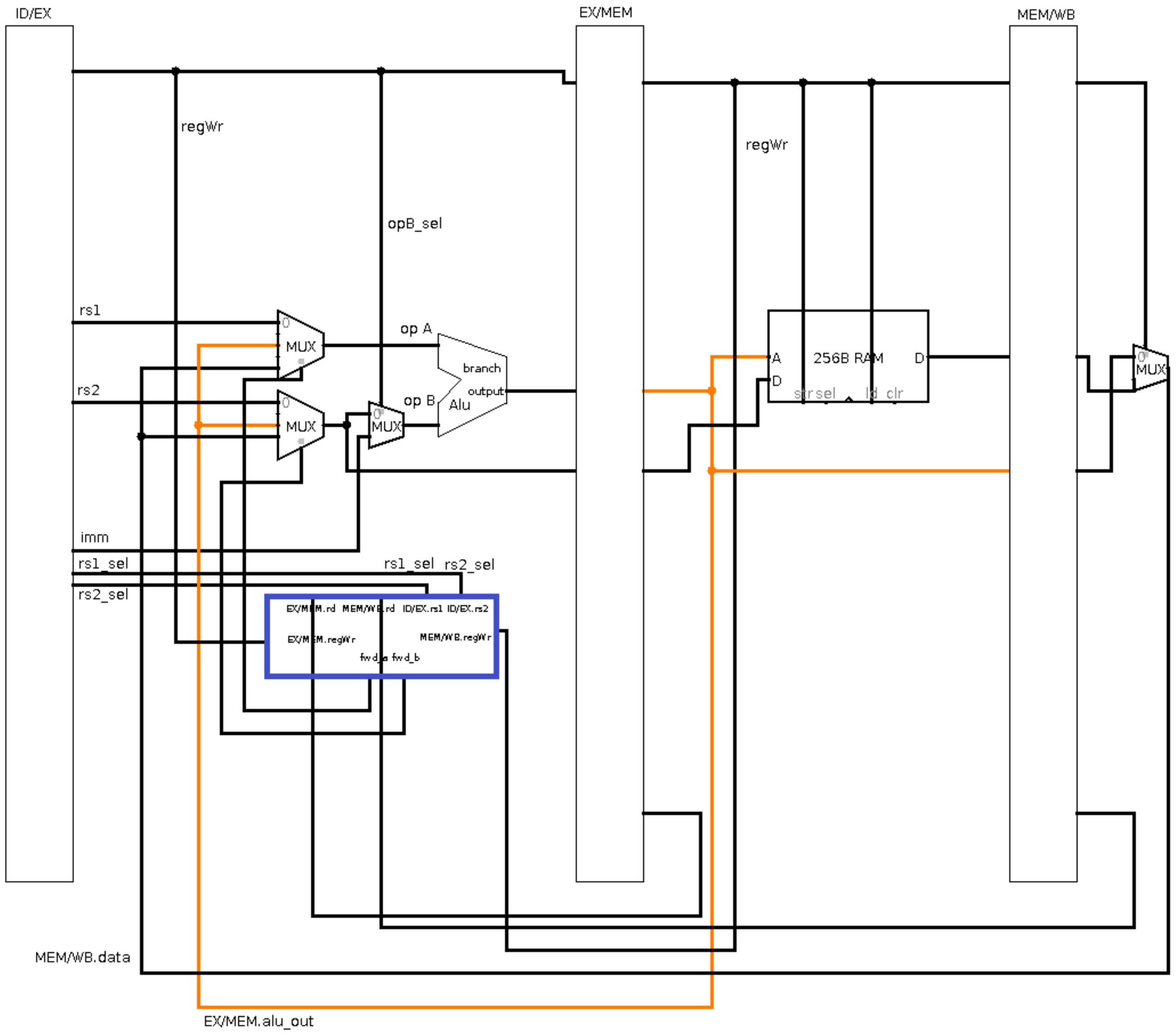
Here our Store instruction is dependent on a previous instruction. When our current instruction that is the Store instruction is executing the ALU *will be calculating the address* for storing the value in the memory which in this case is 0 + 0 = 0. Here we need the updated value of the *source register 2* since sw uses rs2 as data register that is updated into the data memory. At this point our forwarding unit will detect that there is a hazard since the rs2 of sw is dependent on the rd (destination register) of the previous instruction right before it. So our forward b will be 01 at this time. But we need to update the EX/MEM register 2 with the EX/MEM ALU output this time since we need to store the updated value of x2 which will be in the ALU output of the EX/MEM pipeline register. The operand b of the ALU will get the immediate value with rs1 to calculate the address.

So here inside the when of operand_b_sel which will be true for I-Type and S-Type instructions we use another when to test each condition of the forwarding unit and update the value in the EX/MEM register 2.

(5) In the otherwise condition of the operand_b_sel (when operand_b_sel != 1) we update the operand b of the ALU with the correct data which will be either from the ID/EX source register values or from the EX/MEM or MEM/WB pipeline register values and update the register 2 of the EX/MEM as well.

We have wired our forwarding unit in the top and can solve almost all the data hazards except some which we will cover in the next topic.

Block Diagram:



The above diagram is simplified to show the tasks of the forwarding unit only and keep the diagram simple. It doesn't have all the modules here like Alu Control is missing, the purpose is to show the working of the forwarding unit as cleanly as possible without lots of wires.

Data Hazards and Stalls

We have seen that we can easily resolve data hazards by some extra hardware and logic (forwarding unit). But there maybe cases when even forwarding cannot help and thus we have to stall. Stall means pausing the execution for one clock cycle so that our data gets ready.

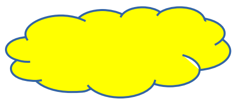
Let's get straight into these conditions:

	CC1	CC2	CC3	CC4	CC5	CC6
Lw x3, 0(x0)	IM	REG	ALU	DM	REG	
Addi x4, x3, 1		IM	REG	ALU	DM	REG

We have seen instructions that are dependent on R-Type or I-Type instructions. But what if an instruction is dependent on a Load type instruction? We can see this hazard in the highlighted CC4 stage. The Load instruction is in the Memory stage and is loading the value in x3 while the dependent instruction of Add is using the x3 value. Our forwarding unit will not help here. Why? Because forwarding unit forwards the ALU output data from the EX/MEM stage which is ready at the end of CC3 and then dependent instruction uses that value in the CC4. But here in the case of Load the data is not ready in CC3 but instead is ready at the end of CC4. So we need to stall our next instruction so that when it is in the execution stage at CC5 we can forward the value from the MEM/WB output.

Let's visualize how stalling will help:

	CC1	CC2	CC3	CC4	CC5	CC6	CC7
Lw x3, 0(x0)	IM	REG	ALU	DM	REG		
Addi x4,x3, 1		IM	REG				



	CC1	CC2	CC3	CC4	CC5	CC6	CC7
Addi x4,x3,1			IM	REG	ALU	DM	REG

What is happening here is that during CC3 we detect that there is a hazard, an instruction is dependent on a previous load instruction. We insert a stall or bubble. A stall or bubble does nothing in the next stage it just stalls the pipeline so that we get a one clock cycle delay. The dependent instruction is fetched again, decoded and then during CC5 when it is in it's execution stage our forwarding unit will now detect that there is a hazard in Write Back stage and will forward the data from the data memory to the input of the ALU.

There are two things to notice here.

1) We need a detection hardware in the Instruction Decode stage that looks for these hazards that arrive after the load instruction.

2) Upon detecting these hazards we need to add a bubble into the pipeline to get one clock cycle delay.

Step 1 is partially what we did in forwarding unit so we know a little bit about how we can detect these hazards.

But what about Step 2? How can we add a bubble to get one clock cycle delay. Well we can simply turn all the control signals to 0 and reload the current instruction in the Decode stage and preserve the fetched instruction. So that any instruction that is passed forward doesn't matter since they won't be written in the data memory or register file as their control pins will be 0. Such instructions are also called **nops** (no operation).

Let's look at the Hazard Detection Unit:

Coding Hazard Detection

```
class HazardDetection extends Module {
  val io = IO(new Bundle {
    val IF_ID_INST = Input(UInt(32.W))
    val ID_EX_MEMREAD = Input(UInt(1.W))
    val ID_EX_REGRD = Input(UInt(5.W))
    val pc_in = Input(SInt(32.W))
    val current_pc = Input(SInt(32.W))
    val inst_forward = Output(UInt(1.W))
    val pc_forward = Output(UInt(1.W))
    val ctrl_forward = Output(UInt(1.W))
    val inst_out = Output(UInt(32.W))
    val pc_out = Output(SInt(32.W))
    val current_pc_out = Output(SInt(32.W))
  })
  val rs1_sel = io.IF_ID_INST(19, 15)
  val rs2_sel = io.IF_ID_INST(24, 20)
  when(io.ID_EX_MEMREAD === "b1".U && ((io.ID_EX_REGRD === rs1_sel) || (io.ID_EX_REGRD ===
rs2_sel))) {
      io.inst_forward := 1.U
      io.pc_forward := 1.U
      io.ctrl_forward := 1.U
      io.inst_out := io.IF_ID_INST
      io.pc_out := io.pc_in
      io.current_pc_out := io.current_pc

  } .otherwise {
    io.inst_forward := 0.U
    io.pc_forward := 0.U
    io.ctrl_forward := 0.U
    io.inst_out := io.IF_ID_INST  // Doesn't matter if we pass the old instruction forward because it won't be
selected by the mux
    io.pc_out := io.pc_in         // Doesn't matter if we pass the old pc value forward because it won't be
selected by the mux
    io.current_pc_out := io.current_pc
  }
}
```

First of all we declare the inputs and outputs that we need.

Inputs

We need the current instruction **IF_ID_INST** from the IF/ID pipeline register in the Decode stage inside the hazard detection unit so that we can detect if the source registers in the current instruction are dependent on the instruction in the execution stage which will be a load instruction. We need the control signal **ID_EX_MEMREAD** from the ID/EX pipeline register which will tell us if the instruction in the execution stage is a load instruction or not, since we need to stall only if there is a load instruction in the execution other than that our forwarding unit will handle the hazards. We need the **ID_EX_REGRD** for checking the dependency. We need the **pc_in** which is the pc value of the next instruction (the address of the next fetched instruction that will be in the instruction fetch stage). We need it so that when we stall we don’t lose the instruction that was fetched earlier. We need **current_pc** which is the pc of the current instruction that is dependent on the load instruction.

Outputs

We use the **inst_forward** control signal which controls the mux to reload the current instruction again in the Decode stage by feeding the current instruction in the IF/ID pipeline register again. We need to do this because when we stall we need to preserve the current instruction so that after one clock cycle it is again decoded and execute as per normal flow. If we don’t preserve this instruction then after one clock cycle a new instruction will be fetched in the Decode stage. We use the **pc_forward** to control the mux that updates the PC value. We will update the PC value with the pc_in value which was the pc of the instruction that was fetched. We need to preserve that instruction as well. All we are doing here is that stopping the PC and IF/ID to update with new values since we need to reload them back to the previous state so that we don’t lose any instruction by stalling the pipeline. We also use the **ctrl_forward** which controls whether the output pins of the Control are set or are 0 (in case of stalling). We use the **inst_out** output which will have the current instruction to be fed into the IF/ID pipeline register if **inst_forward** control signal is high. We use the **pc_out** for feeding the PC with value of the instruction that was in the fetch stage when the hazard was detected. We use the **current_pc_out** to feed the current instruction pc value into the IF/ID pipeline register so that the pipeline register has the current instruction as well as it’s pc value inside it.

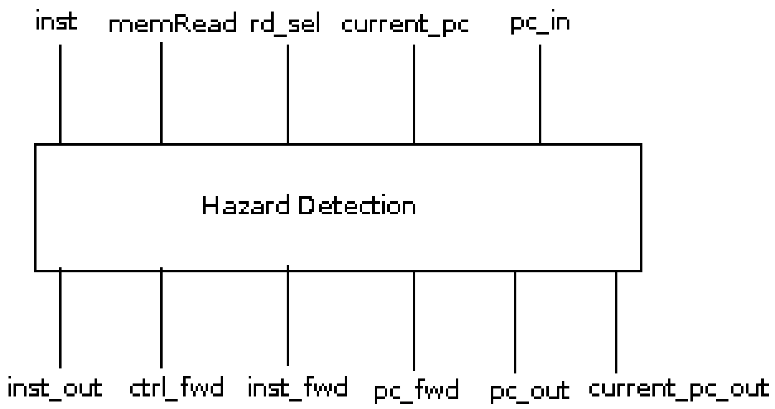
Connections

We extract the register numbers from the instruction fed into the hazard detection unit to find out the dependency.

Then our when conditions are implemented, which detects the dependency by checking if the ID/EX's MEMREAD is 1 (which will be in case of Load instruction) **and** (ID/EX's destination register (Load instruction's destination register) == source register 1 (rs1) **or** ID/EX's destination register (Load instruction's destination register) == source register 2 (rs2))

If this is true then it means that the instruction in the execution stage is a Load instruction and the instruction in the Decode stage is the instruction that is dependent on the Load instruction so we need to stall the pipeline. Thus, we set all the control signals of the output and pass the current instruction, current pc, and fetched instruction pc at the output of our unit. We will use these signals and values in the Top file to send right values in the PC and in the IF/ID pipeline register.

Block Diagram:



Connecting Hazard Detection in Top

Now let's take a look at the Top file:

```
// *********** ----------- INSTRUCTION DECODE (ID) STAGE ----------- ********* //

hazardDetection.io.IF_ID_INST := IF_ID.io.inst_out
hazardDetection.io.ID_EX_MEMREAD := ID_EX.io.ctrl_MemRd_out
hazardDetection.io.ID_EX_REGRD := ID_EX.io.rd_sel_out
hazardDetection.io.pc_in := IF_ID.io.pc4_out
hazardDetection.io.current_pc := IF_ID.io.pc_out
```

We wired all the inputs of the Hazard Detection unit in the Top file from the correct sources as described above in the Inputs explanation.

Now for hazard detection to work we need extra muxes in the Top file. We need one mux in between the instruction memory and the IF/ID pipeline register which is controlled by the **inst_forward** control pin output of the Hazard detection unit. We need another mux before the PC which is controlled by the **pc_forward** output pin of the Hazard detection unit.

```
// *********** ----------- INSTRUCTION FETCH (IF) STAGE ----------- ********* //

// The Mux before the IF_ID Pipeline register which selects the instruction input
// either from the instruction memory or from the hazard detection unit

when(hazardDetection.io.inst_forward === "b1".U) {
  IF_ID.io.inst_in := hazardDetection.io.inst_out
  IF_ID.io.pc_in := hazardDetection.io.current_pc_out
} .otherwise {
    IF_ID.io.inst_in := imem.io.readData
}
```

This is the mux that detects if inst_forward is 1 we wire the instruction input of the IF/ID pipeline register with the inst_out of the hazard detection unit. We also pass the pc value of that instruction as well. Otherwise we normally pass the instruction from the instruction memory that will be the next instruction.

```
// The Mux before the pc which selects the input to the pc
// either from the hazard detection unit or from the adder +4
when(hazardDetection.io.pc_forward === "b1".U) {
  pc.io.in := hazardDetection.io.pc_out
} .otherwise {
  pc.io.in := pc.io.pc4
}
```

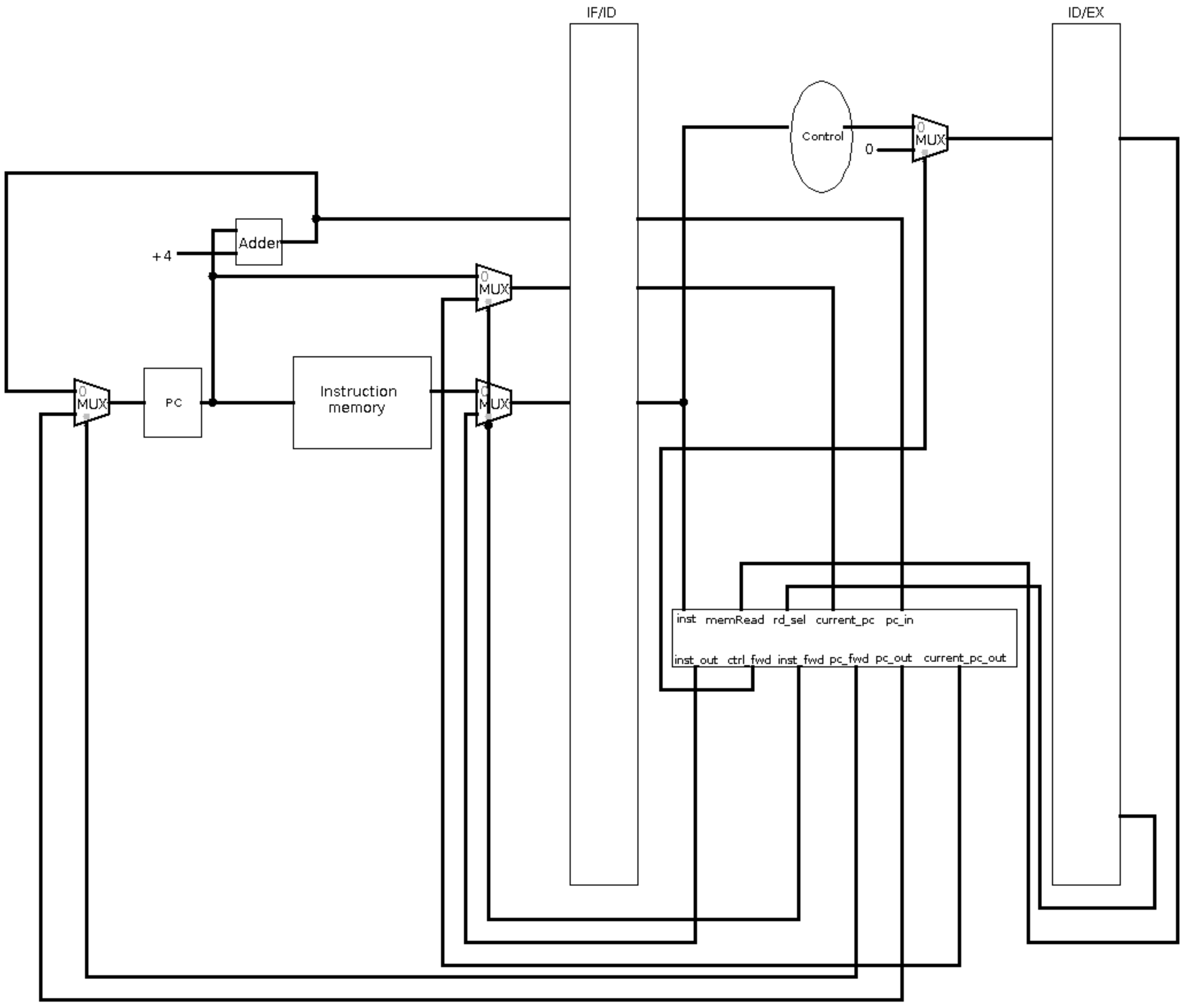
This is the mux that detects if pc_forward is 1 wire the input of the PC with hazard detection's pc_out which is the pc of the instruction that was fetched earlier. Else wire the input of the PC with the next instruction address that is pc + 4.

```
// *********** ----------- INSTRUCTION DECODE (ID) STAGE ----------- ********* //
```

```
when(hazardDetection.io.ctrl_forward === "b1".U) {
    ID_EX.io.ctrl_MemWr_in := 0.U
    ID_EX.io.ctrl_MemRd_in := 0.U
    ID_EX.io.ctrl_Branch_in := 0.U
    ID_EX.io.ctrl_RegWr_in := 0.U
    ID_EX.io.ctrl_MemToReg_in := 0.U
    ID_EX.io.ctrl_AluOp_in := 0.U
    ID_EX.io.ctrl_OpA_sel_in := 0.U
    ID_EX.io.ctrl_OpB_sel_in := 0.U
    ID_EX.io.ctrl_nextPc_sel_in := 0.U

} .otherwise {
    ID_EX.io.ctrl_MemWr_in := control.io.out_memWrite
    ID_EX.io.ctrl_MemRd_in := control.io.out_memRead
    ID_EX.io.ctrl_Branch_in := control.io.out_branch
    ID_EX.io.ctrl_RegWr_in := control.io.out_regWrite
    ID_EX.io.ctrl_MemToReg_in := control.io.out_memToReg
    ID_EX.io.ctrl_AluOp_in := control.io.out_aluOp
    ID_EX.io.ctrl_OpA_sel_in := control.io.out_operand_a_sel
    ID_EX.io.ctrl_OpB_sel_in := control.io.out_operand_b_sel
    ID_EX.io.ctrl_nextPc_sel_in := control.io.out_next_pc_sel

}
```

Lastly we change how our control is wired. If the ctrl_forward is 1 then we need to stall the pipeline so we set all the control signals to 0, else they are set to their default values.

Block Diagram:



Here we have omitted other elements so that we can just focus how our hazard detection unit is connected in the Top file and how it controls other modules.

After setting up our Hazard detection we can now forward data without any hazards and stall the pipeline. Next we will move towards Control Hazards and see how we can solve them.

Control Hazards

As we discussed previously, control hazards arrive during branch conditions. The decision whether the branch is taken or not is done at the Memory stage, while the next instructions are being fetched, decoded and executed in sequential order. If our branch was taken then it means the next three instructions that were fetched, decoded and executed needs to be killed because we don't want to perform the next instructions since our branch is taken.

Let's visualize the above concept.

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
Beq x0,x0,label	IM	REG	ALU	DM	REG				
Add x2, x3, x4		IM	REG	ALU	DM	REG			
Addi x3, x0, 1			IM	REG	ALU	DM	REG		
Addi x4, x3, 1				IM	REG	ALU	DM	REG	
Label: add x1,x1,x2					IM	REG	ALU	DM	REG

By looking at the above chart we can see that during CC3 the ALU will compare the two source registers rs1 and rs2 to see if they are equal or not (beq). The ALU branch output will be 1 in this case since x0 is always 0. This branch output is then fed into the EX/MEM pipeline register. Then in CC4 this branch output is used together with the control branch signal to make sure that the instruction really was an SB-Type instruction and a next_pc_sel control signal which tells the mux to select the SB immediate value rather than UJ immediate value to be updated in the PC. So all of this happens during the CC4 and during this clock cycle the next three instructions are also in the pipeline, one is in the execution stage, the other is in the decode stage and another is in the fetch stage as we can see above. Since our branch here is taken we will have to kill these instructions so that they don't make changes in the memories or register files because we don't want to execute these instructions. Killing these instructions means we need to make sure they don't make any changes in the datapath and then start with the instruction at the required label. This means we will have a clock cycle delay of 3. As we can see above in ideal condition our green highlighted label instruction should have been fetched at CC2 but will now be fetched at CC5 because at CC2 we did not know if the branch was taken or not. If the branch was not taken then there would be no delay in the datapath.

What if there is better way to do this? How can we reduce this delay? What if instead of relying on the ALU to perform branching comparisons why not make a branching logic unit that resides in the decode stage where as soon as the branch instruction is decoded, it's source registers are compared and if they are equal then update the PC value. Using this implementation we will have a delay of 1 clock cycle only rather than 3 clock cycles. During CC2 our Branch Logic unit will detect branching condition and then on the next clock cycle i.e in CC3 our label instruction will be fetched.

Let’s visualize what will happen when we implement the Branch Logic unit in the decode stage.

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
Beq x0,x0,label	IM	REG	ALU	DM	REG				
Add x2, x3, x4		IM (KILL)	ZERO	ZERO	ZERO	ZERO			
Addi x3, x0, 1									
Addi x4, x3, 1									
Label: add x1,x1,x2			IM	REG	ALU	DM	REG		

Now here during CC2 our branch instruction is decoded and source registers rs1 and rs2 are fed into the Branch Logic unit. The branch logic unit will do it’s task (which we will see later) and output the branch taken output (either 1 or 0). This is then AND with Control’s branch output which ensures that the current instruction was a branch instruction and then by using Control’s next_pc_sel we send the sb_imm value at the input of the PC. Meanwhile the next instruction would have been fetched in the IF/ID pipeline register which we have to kill now. We simply will reset (0) the IF/ID pipeline register inputs so that the instruction fetched would not matter and the whole instruction would simply become zero. Then in the next clock cycle i.e during CC3 the PC value will be updated with the sb_imm value and the instruction at the label would be fetched, as shown in the above chart.

Implementing Branch Logic Unit

We have seen the advantages of a separate branch logic unit though it would add extra hardware to our datapath but would improve the performance of our processor. Our branch logic unit will take rs1 and rs2 from the register file and func3 from the instruction which will guide the branch logic unit either the instruction is beq, blt, bltu etc.

In the next section we will look at the code of the Branch Logic Unit.

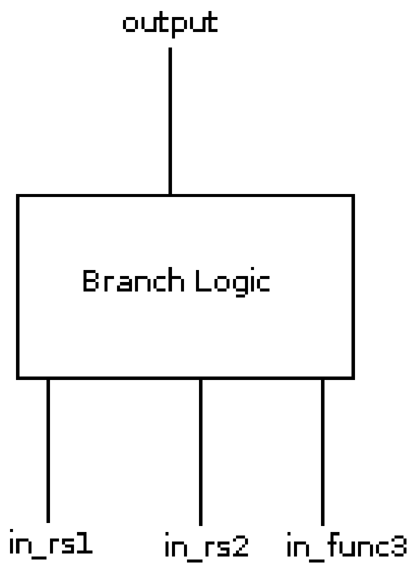
Coding the Branch Logic Unit

```
class BranchLogic extends Module {
 val io = IO(new Bundle {
   val in_rs1 = Input(SInt(32.W))
   val in_rs2 = Input(SInt(32.W))
   val in_func3 = Input(UInt(3.W))
   val output = Output(UInt(1.W))
 })

 when(io.in_func3 === "b000".U) {
   // BEQ
   when(io.in_rs1 === io.in_rs2) {
     io.output := 1.U
   } .otherwise {
     io.output := 0.U
   }
 } .elsewhen(io.in_func3 === "b001".U) {
   // BNE
   when(io.in_rs1 =/= io.in_rs2) {
     io.output := 1.U
   } .otherwise {
     io.output := 0.U
   }
 } .elsewhen(io.in_func3 === "b100".U) {
   // BLT
   when(io.in_rs1 < io.in_rs2) {
     io.output := 1.U
   } .otherwise {
     io.output := 0.U
   }
 } .elsewhen(io.in_func3 === "b101".U) {
   // BGE
   when(io.in_rs1 >= io.in_rs2) {
     io.output := 1.U
   } .otherwise {
     io.output := 0.U
   }
 } .elsewhen(io.in_func3 === "b110".U) {
   // BLTU
   when(io.in_rs1.asUInt < io.in_rs2.asUInt) {
     io.output := 1.U
   } .otherwise {
     io.output := 0.U
   }
 } .elsewhen(io.in_func3 === "b111".U) {
   // BGEU
   when(io.in_rs1.asUInt >= io.in_rs2.asUInt) {
     io.output := 1.U
   } .otherwise {
     io.output := 0.U
   }
 } .otherwise {
   io.output := 0.U
 }

}
```

Here we have the inputs as described above and an output which tells if the branch is taken or not.

The func3 tells us which branch instruction is there, we used the help of the RISC-V green card for reference to see how the func3 bits are changing for various branching instructions.

Block Diagram:



Now let’s look at the top file to see how we connect our Branch Logic unit.

Connecting Branch Logic in Top

```
branchLogic.io.in_rs1 := reg_file.io.rs1
branchLogic.io.in_rs2 := reg_file.io.rs2
branchLogic.io.in_func3 := IF_ID.io.inst_out(14,12)
```

Simply wiring the inputs of the Branch Logic unit in the Top file

```
when(hazardDetection.io.pc_forward === "b1".U) {
  pc.io.in := hazardDetection.io.pc_out
} .otherwise {
    when(control.io.out_next_pc_sel === "b01".U) {
      when(branchLogic.io.output === 1.U && control.io.out_branch === 1.U) {
        pc.io.in := imm_generation.io.sb_imm
        IF_ID.io.pc_in := 0.S
        IF_ID.io.pc4_in := 0.S
        IF_ID.io.inst_in := 0.U
      } .otherwise {
        pc.io.in := pc.io.pc4
      }
}
```

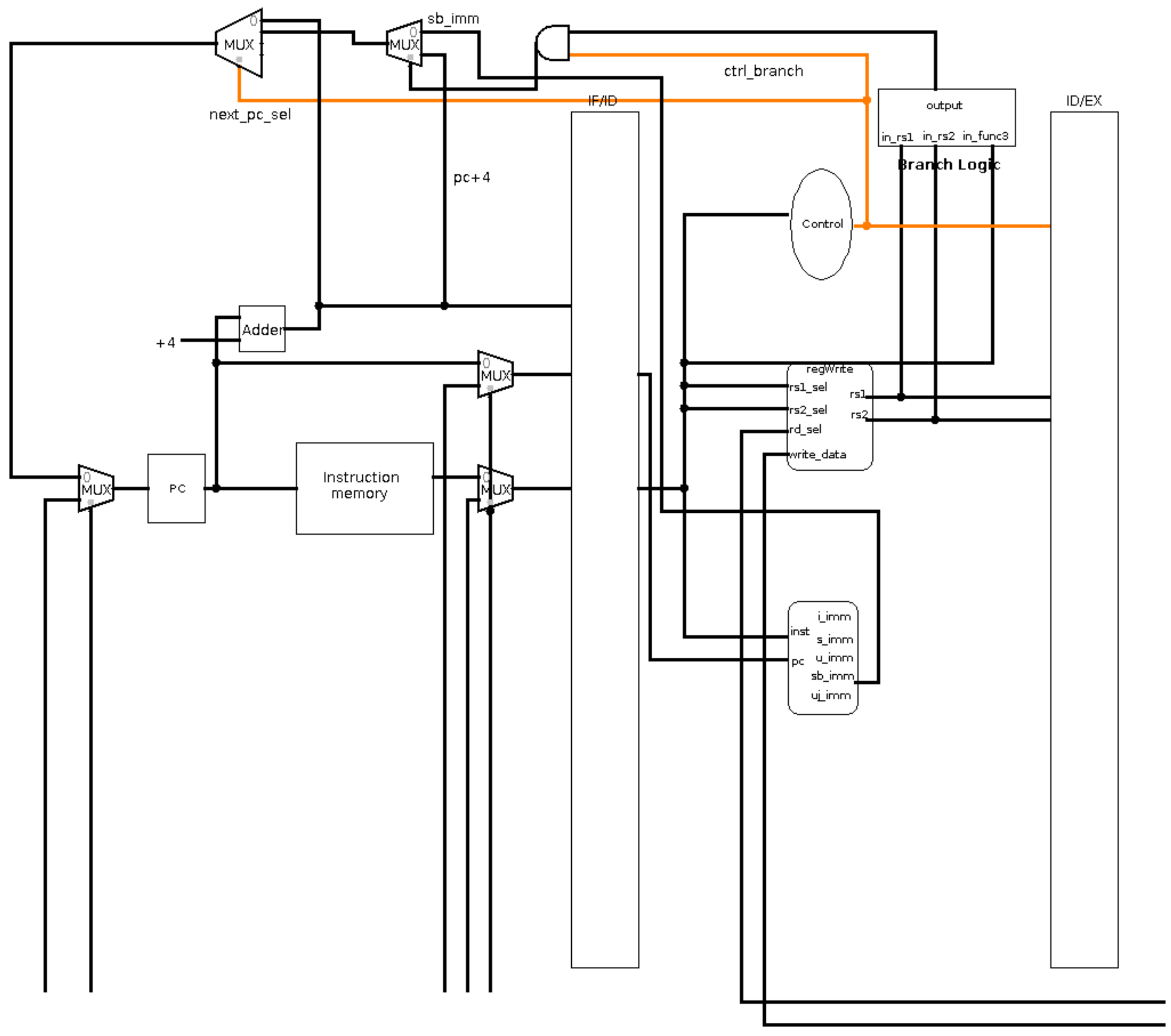
This is the mux which is controlling the PC input value as seen before in the code when we connected the Hazard Detection module in the Top file.

First of all as previously we check if there is no stall requirement, if we have to stall then we have to set the PC value from the pc output of the hazard detection, otherwise we were simply updating the pc value by pc4. Now in the otherwise we have to check another condition for the branch condition. If the branch is not taken then simply fetch the next instruction by pc + 4 but if the branch is taken then update the value of the PC with the sb_imm value.

So in the first condition we check if the next_pc_sel from the control is 01 which forwards the sb_imm value. If the next_pc_sel value was 10 it means we have to forward the uj_imm value. So first we check the next_pc_sel to verify that indeed the instruction is SB-Type and we have to forward the sb_imm value. Then inside that when condition we check whether the branch logic unit is 1 AND the control pin of the branch is 1 which essentially means that the instruction was a branch type instruction and the branch is taken. Inside this condition we wire the PC input with the sb_imm coming from the immediate generation. And we flushed the IF/ID pipeline register by setting all it's inputs to 0 as we discussed above for killing the fetched instruction which we do not need now since we have to fetch the label instruction. Otherwise if the branch is not taken we simply wire the PC input with pc4 which gets us the next instruction in the sequential order.

After these connections we can perform branching in our datapath. But everything comes with a cost (trade-offs). We improved the processing of our processor by reducing the clock cycle delay, but we added more complexity and hardware in our datapath. Due to our new Branch Logic unit in the datapath we have to resolve the data hazards for branching conditions now. If the branching had been done in the execution stage with the ALU our forwarding unit will have helped us with data hazards. But now we have to resolve these data hazards which means we have to create another forwarding unit specifically for resolving branch hazards (if branch instruction is dependent on other register values which have not been written back yet). In the next section we will analyze when problems can occur in our Branch Logic unit.

Block Diagram:



We have omitted other details to have a cleaner look at the branch logic unit and how it outputs the values which can then update the PC value.

Branching Hazards

Right now our Branch Logic unit just takes inputs of the registers from the register file. What if our branch instruction was dependent on the instruction that is just above it or that is two stages above it which means those instructions have not written back their values yet in the register file. Let's visualize the possible hazards we can get.

1. ALU hazard

	CC1	CC2	CC3	CC4	CC5	CC6	CC7
Addi x1,x0,1	IM	REG	ALU	DM	REG		
Beq x0,x1 label		IM	REG	ALU	DM	REG	

Here we can see during CC3 the add instruction is executing and it's output is at the ALU. In the same cycle our branch instruction accessing the register file (decoding) and rs1 and rs2 values are being fed into the Branch Logic unit which of-course will not have the updated value of the x1 register since it has just been computed by the ALU. So we need to detect this hazard and forward the value directly from the output of the ALU in the branch logic unit.

2. EX hazard

	CC1	CC2	CC3	CC4	CC5	CC6	CC7
Addi x1,x0,1	IM	REG	ALU	DM	REG		
Addi x2, x0, 1		IM	REG	ALU	DM	REG	
Beq x2,x1 label			IM	REG	ALU	DM	REG

Here we can see during CC4 when our branch instruction is in decoding stage it has two hazards now. The ALU hazard in the second instruction Addi x2, x0, 1 since the branch instructions's source register rs1 which is x2 is dependent on it. We have seen that we have to forward the ALU output to resolve this hazard. There is another hazard the same one as we solved in our forwarding unit. The first instruction is now in Memory stage and it has the ALU output in the EX/MEM.alu_output pipeline register. So we need to forward the EX/MEM.alu_output value into the branch logic unit as well.

3. MEM hazard

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
Addi x1,x0,1	IM	REG	ALU	DM	REG			
Addi x2, x0, 1		IM	REG	ALU	DM	REG		
Addi x3, x0, 2			IM	REG	ALU	DM	REG	
Beq x2,x1 label				IM	REG	ALU	DM	REG

Here during the CC5 when our branch instruction is in the decode stage, we have another untapped dependency and that is when the first instruction is in the Write Back stage. At this point we have the ALU output in the MEM/WB pipeline register. We can simply forward this data to the Branch Logic unit to resolve this hazard.

As we can notice the last two hazards are similar to what we resolved in the forwarding unit. But the branch forwarding unit will have an extra hazard that we have to resolve. Next, let's look at the implementation of the Branch Forwarding Unit.

The Branch forwarding unit has to take the inputs of the ID/EX's destination register for detecting the *ALU Hazard*, EX/MEM's destination register for detecting the *EX Hazard* and MEM/WB's destination register for detecting the *MEM Hazard*. It will also take the source register rs1_sel and rs2_sel from the instruction in the decode stage and a branch output of the Control which makes sure that the current instruction is a branch instruction.

Let's review our Branch Forwarding unit in the next section.

Coding Branch Forward Unit

```
class BranchForward extends Module {
 val io = IO(new Bundle {
  val ID_EX_REGRD = Input(UInt(5.W))
  val EX_MEM_REGRD = Input(UInt(5.W))
  val MEM_WB_REGRD = Input(UInt(5.W))
  val rs1_sel = Input(UInt(5.W))
  val rs2_sel = Input(UInt(5.W))
  val ctrl_branch = Input(UInt(1.W))
  val forward_rs1 = Output(UInt(3.W))
  val forward_rs2 = Output(UInt(3.W))
 })

  io.forward_rs1 := "b000".U
  io.forward_rs2 := "b000".U

  // ALU Hazard

  when(io.ctrl_branch === 1.U && io.ID_EX_REGRD =/= "b00000".U && (io.ID_EX_REGRD ===  io.rs1_sel) && (io.ID_EX_REGRD ===
io.rs2_sel)) {

    io.forward_rs1 := "b001".U
    io.forward_rs2 := "b001".U

  } .elsewhen(io.ctrl_branch === 1.U && io.ID_EX_REGRD =/= "b00000".U && (io.ID_EX_REGRD === io.rs1_sel)) {

    io.forward_rs1 := "b001".U

  } .elsewhen(io.ctrl_branch === 1.U && io.ID_EX_REGRD =/= "b00000".U && (io.ID_EX_REGRD === io.rs2_sel)) {

    io.forward_rs2 := "b001".U

  }

  // EX/MEM Hazard

  when(io.ctrl_branch === 1.U && io.EX_MEM_REGRD =/= "b00000".U && (io.EX_MEM_REGRD === io.rs1_sel) &&
(io.EX_MEM_REGRD === io.rs2_sel)) {

    io.forward_rs1 := "b010".U
    io.forward_rs2 := "b010".U

  } .elsewhen(io.ctrl_branch === 1.U && io.EX_MEM_REGRD =/= "b00000".U && (io.EX_MEM_REGRD === io.rs2_sel)) {

    io.forward_rs2 := "b010".U

  } .elsewhen(io.ctrl_branch === 1.U && io.EX_MEM_REGRD =/= "b00000".U && (io.EX_MEM_REGRD === io.rs1_sel)) {

    io.forward_rs1 := "b010".U

 }

  // MEM/WB Hazard
  when(io.ctrl_branch === 1.U && io.MEM_WB_REGRD =/= "b00000".U && (io.MEM_WB_REGRD === io.rs1_sel) &&
(io.MEM_WB_REGRD === io.rs2_sel)) {

    io.forward_rs1 := "b011".U
    io.forward_rs2 := "b011".U
  } .elsewhen(io.ctrl_branch === 1.U && io.MEM_WB_REGRD =/= "b00000".U && (io.MEM_WB_REGRD === io.rs2_sel)) {

    io.forward_rs2 := "b011".U

  } .elsewhen(io.ctrl_branch === 1.U && io.MEM_WB_REGRD =/= "b00000".U && (io.MEM_WB_REGRD === io.rs1_sel)) {

    io.forward_rs1 := "b011".U

  }
}
```

Let’s start with the inputs and outputs. As explained above we have destination registers of each pipeline register which tells us if our current instruction in the decode stage is dependent on either the execution stage instruction, memory stage instruction or on the write back stage instruction (the hazard cases we covered above in the chart). We also need the current instruction’s rs1 and rs2 which we get directly from the IF/ID’s instruction output and lastly we need the branch pin of the Control. At the outputs we have two selection pins forward_rs1 and forward_rs2. There will be a mux before each input of the Branch Logic unit which *selects* the correct input source and sends it to the inputs of the branch logic module.

In default case we set the selection pins (forward_rs1 and forward_rs2) to 0. Which means that in top at 0 case the branch logic will get inputs from the register file as there are no hazards.

In ALU hazard case we start with our *when* conditions very similar to the forwarding unit. We check if the ID/EX’s destination register is equal to the current branch instruction’s source register 1 and source register 2 both, then forward_rs1 and forward_rs2 both becomes 001. If either one of them is dependent then only their values are changed to 001. This is what is happening in the ALU hazard case.

In EX/MEM hazard case we check the same conditions but this time check the EX/MEM’s destination register with the current instruction’s source registers and set the forward_rs1 and forward_rs2 to 010.

In MEM/WB hazard case we again check the same conditions this time using the MEM/WB’s destination register with the current instruction’s source register and set the forward_rs1 and forward_rs2 to 011.

Here we again have to tweak our branch forwarding unit just like we did earlier for the forwarding unit. We add a not condition so that when the branch instruction is dependent on the instruction which is in ALU as well as in Memory stage we need the latest value and for that purpose we need to add a not condition which say forward value from EX/MEM output or MEM/WB output only if ALU hazard is ***not*** present. If ALU hazard is present, then we need the latest value.

```
// EX/MEM Hazard
when(io.ctrl_branch === 1.U && io.EX_MEM_REGRD =/= "b00000".U && ~((io.ctrl_branch === "b1".U) && (io.ID_EX_REGRD =/=
"b00000".U) && (io.ID_EX_REGRD === io.rs1_sel) && (io.ID_EX_REGRD === io.rs2_sel)) && (io.EX_MEM_REGRD === io.rs1_sel) &&
(io.EX_MEM_REGRD === io.rs2_sel)) {

  io.forward_rs1 := "b010".U
  io.forward_rs2 := "b010".U

} .elsewhen(io.ctrl_branch === 1.U && io.EX_MEM_REGRD =/= "b00000".U && ~((io.ctrl_branch === "b1".U) && (io.ID_EX_REGRD
=/= "b00000".U) && (io.ID_EX_REGRD === io.rs2_sel)) && (io.EX_MEM_REGRD === io.rs2_sel)) {

  io.forward_rs2 := "b010".U

} .elsewhen(io.ctrl_branch === 1.U && io.EX_MEM_REGRD =/= "b00000".U && ~((io.ctrl_branch === "b1".U) && (io.ID_EX_REGRD
=/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel)) && (io.EX_MEM_REGRD === io.rs1_sel)) {

  io.forward_rs1 := "b010".U

}
```

Here is the highlighted code which has been added in the EX/MEM hazard which only forwards when ALU hazard is ***not*** present.

```
// MEM/WB Hazard
when(io.ctrl_branch === 1.U && io.MEM_WB_REGRD =/= "b00000".U &&

// IF NOT ALU HAZARD
~((io.ctrl_branch === "b1".U) && (io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel) && (io.ID_EX_REGRD
=== io.rs2_sel)) &&

// IF NOT EX/MEM HAZARD
~((io.ctrl_branch === "b1".U) && (io.EX_MEM_REGRD =/= "b00000".U) && (io.EX_MEM_REGRD === io.rs1_sel) &&
(io.EX_MEM_REGRD === io.rs2_sel)) &&

(io.MEM_WB_REGRD === io.rs1_sel) && (io.MEM_WB_REGRD === io.rs2_sel)) {

  io.forward_rs1 := "b011".U
  io.forward_rs2 := "b011".U

} .elsewhen(io.ctrl_branch === 1.U && io.MEM_WB_REGRD =/= "b00000".U &&

// IF NOT ALU HAZARD
~((io.ctrl_branch === "b1".U) && (io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs2_sel)) &&

// IF NOT EX/MEM HAZARD
~((io.ctrl_branch === "b1".U) && (io.EX_MEM_REGRD =/= "b00000".U) && (io.EX_MEM_REGRD === io.rs2_sel)) &&
(io.MEM_WB_REGRD === io.rs2_sel)) {

  io.forward_rs2 := "b011".U

} .elsewhen(io.ctrl_branch === 1.U && io.MEM_WB_REGRD =/= "b00000".U &&

// IF NOT ALU HAZARD
 ~((io.ctrl_branch === "b1".U) && (io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel)) &&



// IF NOT EX/MEM HAZARD
~((io.ctrl_branch === "b1".U) && (io.EX_MEM_REGRD =/= "b00000".U) && (io.EX_MEM_REGRD === io.rs1_sel)) &&
(io.MEM_WB_REGRD === io.rs1_sel)) {

  io.forward_rs1 := "b011".U

}
```

Here is the highlighted code which has been added in the MEM/WB hazard which only forwards when ALU hazard ***and*** EX/MEM hazard is ***not*** present.

Why do we add an extra not condition in the MEM/WB stage? Here is an example:

1. addi x2, x0, 10
2. addi x2, x2, 5
3. addi x3, x0, 15
4. beq x2, x3, down
5. jal exit
6. down:
   addi x1, x0, 1
7. exit:

Here is our basic program for testing our branch instructions. Instruction 4 which is our branch instruction has one dependency of x3 and one dependency of x2.

The branch instruction gets the x3 dependency resolved by the condition in the ALU hazard since we need the ALU output directly here, because x3 is dependent on instruction 3.

Now for x2 the branch instruction has two dependencies, instruction 2 and instruction 1. Instruction 2 is the EX/MEM hazard case and instruction 1 is the MEM/WB hazard case. Now which value should be forwarded into the branch logic unit? Of-course the latest one from the instruction 2. How would our branch forward know to get the latest value? That is why in the MEM/WB stage we have two not conditions, 1) for the ALU hazard and 2) for the EX/MEM hazard. If x2 is ***not*** dependent on the instruction in the ALU hazard stage ***and*** if x2 is ***not*** dependent on the instruction in the EX/MEM hazard stage only then forward from MEM/WB.

Before we move forward to connect the Branch Forward unit in the Top file, we have to cover one more scenario in our Branch Forward unit.

Let's cover it next.

What if our branch instruction is dependent on a load instruction's destination register?

	CC1	CC2	CC3	CC4	CC5	CC6	CC7
Lw x2, 0(x0)	IM	REG	ALU	DM	REG		

Beq x1,x2 label		IM	REG	ALU	DM	REG	

Here in this program the hazard is the same as the ALU hazard and our branch forward unit will forward the ALU output to the branch logic unit which will get the wrong value of *address* that was calculated by the ALU in case of this load instruction. So we have to make sure that during ALU hazard we only forward the ALU output when the ID/EX's MEMREAD control signal is != 1 which means that the instruction in the execution stage is not a load instruction. Similar is the case if this same hazard arises in the EX/MEM and MEM/WB hazards. By this our branch forward unit will not forward the value of ALU output now if there is a load instruction in the execution stage. Now what should our branch forward unit forward then? Since our hazard detector will see that there is an instruction which dependent on the load instruction it will add a stall to the pipeline. So now in the next clock cycle our instruction will look like this:

	CC1	CC2	CC3	CC4	CC5	CC6	CC7
Lw x2, 0(x0)	IM	REG	ALU	DM	REG		
Beq x1,x2 label			IM	REG	ALU	DM	REG

Since the hazard detector added a stall by detecting it's hazard now we have this condition. The Load instruction will be in the Memory stage and our branch instruction will be in the decode stage during that clock cycle. What we can do is forward the data loaded from the data memory to the branch logic unit.

So now our finalized Branch Forward unit is shown below with changes highlighted.

```
val io = IO(new Bundle {
  val ID_EX_REGRD = Input(UInt(5.W))
  val ID_EX_MEMRD = Input(UInt(1.W))
  val EX_MEM_REGRD = Input(UInt(5.W))
  val EX_MEM_MEMRD = Input(UInt(1.W))
  val MEM_WB_REGRD = Input(UInt(5.W))
  val MEM_WB_MEMRD = Input(UInt(1.W))
  val rs1_sel = Input(UInt(5.W))
  val rs2_sel = Input(UInt(5.W))
  val ctrl_branch = Input(UInt(1.W))
  val forward_rs1 = Output(UInt(3.W))
  val forward_rs2 = Output(UInt(3.W))
})
```

Here we have added MEMRD (memory read) control signals from each pipeline register to check for load instruction in each stage.

```
// ALU Hazard

when(io.ctrl_branch === 1.U && io.ID_EX_REGRD =/= "b00000".U && io.ID_EX_MEMRD =/= 1.U && (io.ID_EX_REGRD ===
io.rs1_sel) && (io.ID_EX_REGRD === io.rs2_sel)) {

  io.forward_rs1 := "b001".U
  io.forward_rs2 := "b001".U
} .elsewhen(io.ctrl_branch === 1.U && io.ID_EX_REGRD =/= "b00000".U && io.ID_EX_MEMRD =/= 1.U && (io.ID_EX_REGRD ===
io.rs1_sel)) {

  io.forward_rs1 := "b001".U

} .elsewhen(io.ctrl_branch === 1.U && io.ID_EX_REGRD =/= "b00000".U && io.ID_EX_MEMRD =/= 1.U && (io.ID_EX_REGRD ===
io.rs2_sel)) {

  io.forward_rs2 := "b001".U

}
```

We have added an additional condition which ensures that forward the ALU output only when there is no load instruction.

```
// EX/MEM Hazard

  when(io.ctrl_branch === 1.U && io.EX_MEM_REGRD =/= "b00000".U && io.EX_MEM_MEMRD =/= 1.U && ~((io.ctrl_branch ===
"b1".U) && (io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel) && (io.ID_EX_REGRD === io.rs2_sel)) &&
(io.EX_MEM_REGRD === io.rs1_sel) && (io.EX_MEM_REGRD === io.rs2_sel)) {

    io.forward_rs1 := "b010".U
    io.forward_rs2 := "b010".U

  } .elsewhen(io.ctrl_branch === 1.U && io.EX_MEM_REGRD =/= "b00000".U && io.EX_MEM_MEMRD =/= 1.U && ~((io.ctrl_branch
=== "b1".U) && (io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs2_sel)) && (io.EX_MEM_REGRD === io.rs2_sel))
{

    io.forward_rs2 := "b010".U

  } .elsewhen(io.ctrl_branch === 1.U && io.EX_MEM_REGRD =/= "b00000".U && io.EX_MEM_MEMRD =/= 1.U && ~((io.ctrl_branch
=== "b1".U) && (io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel)) && (io.EX_MEM_REGRD === io.rs1_sel))
{

    io.forward_rs1 := "b010".U

} .elsewhen(io.ctrl_branch === 1.U && io.EX_MEM_REGRD =/= "b00000".U && io.EX_MEM_MEMRD === 1.U && ~((io.ctrl_branch
=== "b1".U) && (io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel) && (io.ID_EX_REGRD === io.rs2_sel)) &&
(io.EX_MEM_REGRD === io.rs1_sel) && (io.EX_MEM_REGRD === io.rs2_sel)) {

    // FOR Load instructions
    io.forward_rs1 := "b100".U
    io.forward_rs2 := "b100".U

}
.elsewhen(io.ctrl_branch === 1.U && io.EX_MEM_REGRD =/= "b00000".U && io.EX_MEM_MEMRD === 1.U && ~((io.ctrl_branch
=== "b1".U) && (io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs2_sel)) && (io.EX_MEM_REGRD === io.rs2_sel))
{

    io.forward_rs2 := "b100".U

}

.elsewhen(io.ctrl_branch === 1.U && io.EX_MEM_REGRD =/= "b00000".U && io.EX_MEM_MEMRD === 1.U && ~((io.ctrl_branch
=== "b1".U) && (io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel)) && (io.EX_MEM_REGRD === io.rs1_sel))
{

    io.forward_rs1 := "b100".U

}
```

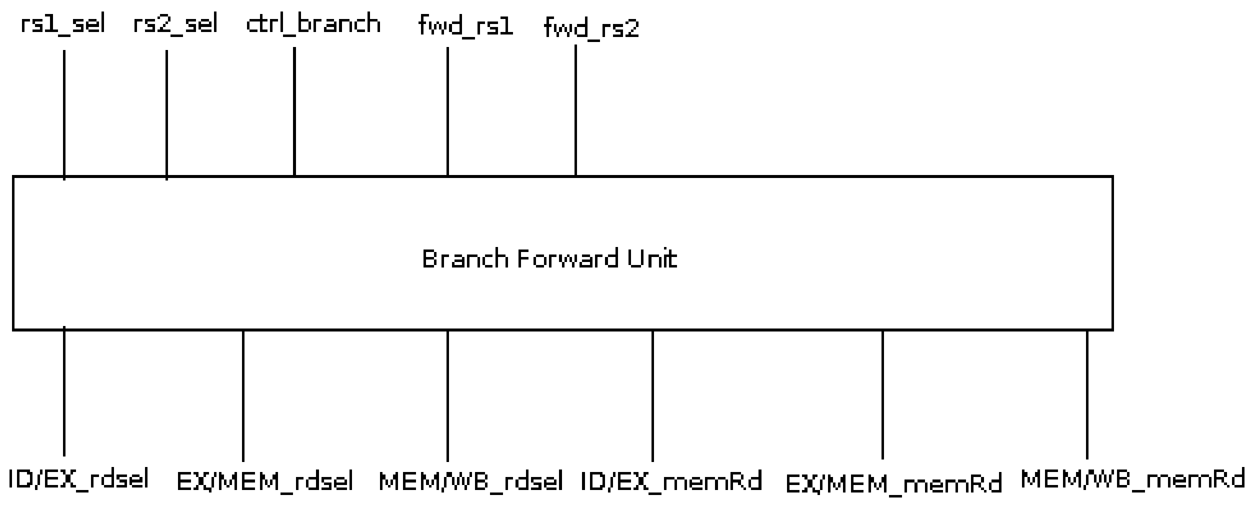
Here we have added three additional conditions in the EX/MEM hazard case. That if EX/MEM's MEMREAD is 1 which means there is a load instruction in the execution stage and a branch instruction in the decode stage and it is dependent on the load instruction then we will set the forward lines to another value 100 with which we will be forwarding the data from the data memory.

```
// MEM/WB Hazard
when(io.ctrl_branch === 1.U && io.MEM_WB_REGRD =/= "b00000".U && io.MEM_WB_MEMRD =/= 1.U &&
  // IF NOT ALU HAZARD
  ~((io.ctrl_branch === "b1".U) && (io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel) && (io.ID_EX_REGRD
=== io.rs2_sel)) &&
  // IF NOT EX/MEM HAZARD
  ~((io.ctrl_branch === "b1".U) && (io.EX_MEM_REGRD =/= "b00000".U) && (io.EX_MEM_REGRD === io.rs1_sel) &&
(io.EX_MEM_REGRD === io.rs2_sel)) &&
  (io.MEM_WB_REGRD === io.rs1_sel) && (io.MEM_WB_REGRD === io.rs2_sel)) {

  io.forward_rs1 := "b011".U
  io.forward_rs2 := "b011".U

}
  .elsewhen(io.ctrl_branch === 1.U && io.MEM_WB_REGRD =/= "b00000".U && io.MEM_WB_MEMRD =/= 1.U &&
    // IF NOT ALU HAZARD
    ~((io.ctrl_branch === "b1".U) && (io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs2_sel)) &&
    // IF NOT EX/MEM HAZARD
    ~((io.ctrl_branch === "b1".U) && (io.EX_MEM_REGRD =/= "b00000".U) && (io.EX_MEM_REGRD === io.rs2_sel)) &&
    (io.MEM_WB_REGRD === io.rs2_sel)) {

  io.forward_rs2 := "b011".U

}
  .elsewhen(io.ctrl_branch === 1.U && io.MEM_WB_REGRD =/= "b00000".U && io.MEM_WB_MEMRD =/= 1.U &&
    // IF NOT ALU HAZARD
    ~((io.ctrl_branch === "b1".U) && (io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel)) &&
    // IF NOT EX/MEM HAZARD
    ~((io.ctrl_branch === "b1".U) && (io.EX_MEM_REGRD =/= "b00000".U) && (io.EX_MEM_REGRD === io.rs1_sel)) &&
    (io.MEM_WB_REGRD === io.rs1_sel)) {

    io.forward_rs1 := "b011".U

} .elsewhen(io.ctrl_branch === 1.U && io.MEM_WB_REGRD =/= "b00000".U && io.MEM_WB_MEMRD === 1.U &&
  // IF NOT ALU HAZARD
  ~((io.ctrl_branch === "b1".U) && (io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel) && (io.ID_EX_REGRD
=== io.rs2_sel)) &&
  // IF NOT EX/MEM HAZARD
  ~((io.ctrl_branch === "b1".U) && (io.EX_MEM_REGRD =/= "b00000".U) && (io.EX_MEM_REGRD === io.rs1_sel) &&
(io.EX_MEM_REGRD === io.rs2_sel)) &&
  (io.MEM_WB_REGRD === io.rs1_sel) && (io.MEM_WB_REGRD === io.rs2_sel)) {
  // FOR Load instructions
  io.forward_rs1 := "b101".U
  io.forward_rs2 := "b101".U

}
  .elsewhen(io.ctrl_branch === 1.U && io.MEM_WB_REGRD =/= "b00000".U && io.MEM_WB_MEMRD === 1.U &&
    // IF NOT ALU HAZARD
    ~((io.ctrl_branch === "b1".U) && (io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs2_sel)) &&
    // IF NOT EX/MEM HAZARD
    ~((io.ctrl_branch === "b1".U) && (io.EX_MEM_REGRD =/= "b00000".U) && (io.EX_MEM_REGRD === io.rs2_sel)) &&
    (io.MEM_WB_REGRD === io.rs2_sel)) {

    io.forward_rs2 := "b101".U

}
  .elsewhen(io.ctrl_branch === 1.U && io.MEM_WB_REGRD =/= "b00000".U && io.MEM_WB_MEMRD === 1.U &&
    // IF NOT ALU HAZARD
    ~((io.ctrl_branch === "b1".U) && (io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel)) &&
    // IF NOT EX/MEM HAZARD
    ~((io.ctrl_branch === "b1".U) && (io.EX_MEM_REGRD =/= "b00000".U) && (io.EX_MEM_REGRD === io.rs1_sel))&&
    (io.MEM_WB_REGRD === io.rs1_sel)) {

    io.forward_rs1 := "b101".U

}
```

Similarly we added three new conditions when the MEM/WB's MEMRD is 1 for the load instruction and set the forward signal to 101 which will eventually forward the value from MEM/WB's data output.

Finally, our branch forward unit is ready and now in the next topic we will add it in the Top file.

Block Diagram:



Connecting Branch Forward in Top

```
// *********** ----------- INSTRUCTION DECODE (ID) STAGE ----------- ********* //


// Initializing Branch Forward Unit
branchForward.io.ID_EX_REGRD := ID_EX.io.rd_sel_out
branchForward.io.ID_EX_MEMRD := ID_EX.io.ctrl_MemRd_out
branchForward.io.EX_MEM_REGRD := EX_MEM.io.ex_mem_rdSel_output
branchForward.io.MEM_WB_REGRD := MEM_WB.io.mem_wb_rdSel_output
branchForward.io.EX_MEM_MEMRD := EX_MEM.io.ex_mem_memRd_out
branchForward.io.MEM_WB_MEMRD := MEM_WB.io.mem_wb_memRd_output
branchForward.io.rs1_sel := IF_ID.io.inst_out(19, 15)
branchForward.io.rs2_sel := IF_ID.io.inst_out(24, 20)
branchForward.io.ctrl_branch := control.io.out_branch
```

Here we are simply wiring the inputs of the Branch forward unit with their correct sources from the Top file.

```
// FOR REGISTER RS1 in BRANCH LOGIC UNIT
when(branchForward.io.forward_rs1 === "b000".U) {
  // No hazard just use register file data
  branchLogic.io.in_rs1 := reg_file.io.rs1
} .elsewhen(branchForward.io.forward_rs1 === "b001".U) {
  // hazard in alu stage forward data from alu output
  branchLogic.io.in_rs1 := alu.io.output
} .elsewhen(branchForward.io.forward_rs1 === "b010".U) {
  // hazard in EX/MEM stage forward data from EX/MEM.alu_output
  branchLogic.io.in_rs1 := EX_MEM.io.ex_mem_alu_output
} .elsewhen(branchForward.io.forward_rs1 === "b011".U) {
  // hazard in MEM/WB stage forward data from register file write data which will have correct data from the MEM/WB mux
  branchLogic.io.in_rs1 := reg_file.io.writeData
} .elsewhen(branchForward.io.forward_rs1 === "b100".U) {
  // hazard in EX/MEM stage and load type instruction so forwarding from data memory data output instead of EX/MEM.alu_output
  branchLogic.io.in_rs1 := dmem.io.memOut
} .elsewhen(branchForward.io.forward_rs1 === "b101".U) {
  // hazard in MEM/WB stage and load type instruction so forwarding from register file write data which will have the correct output
from the mux
  branchLogic.io.in_rs1 := reg_file.io.writeData
}
  .otherwise {
  branchLogic.io.in_rs1 := reg_file.io.rs1
}
```

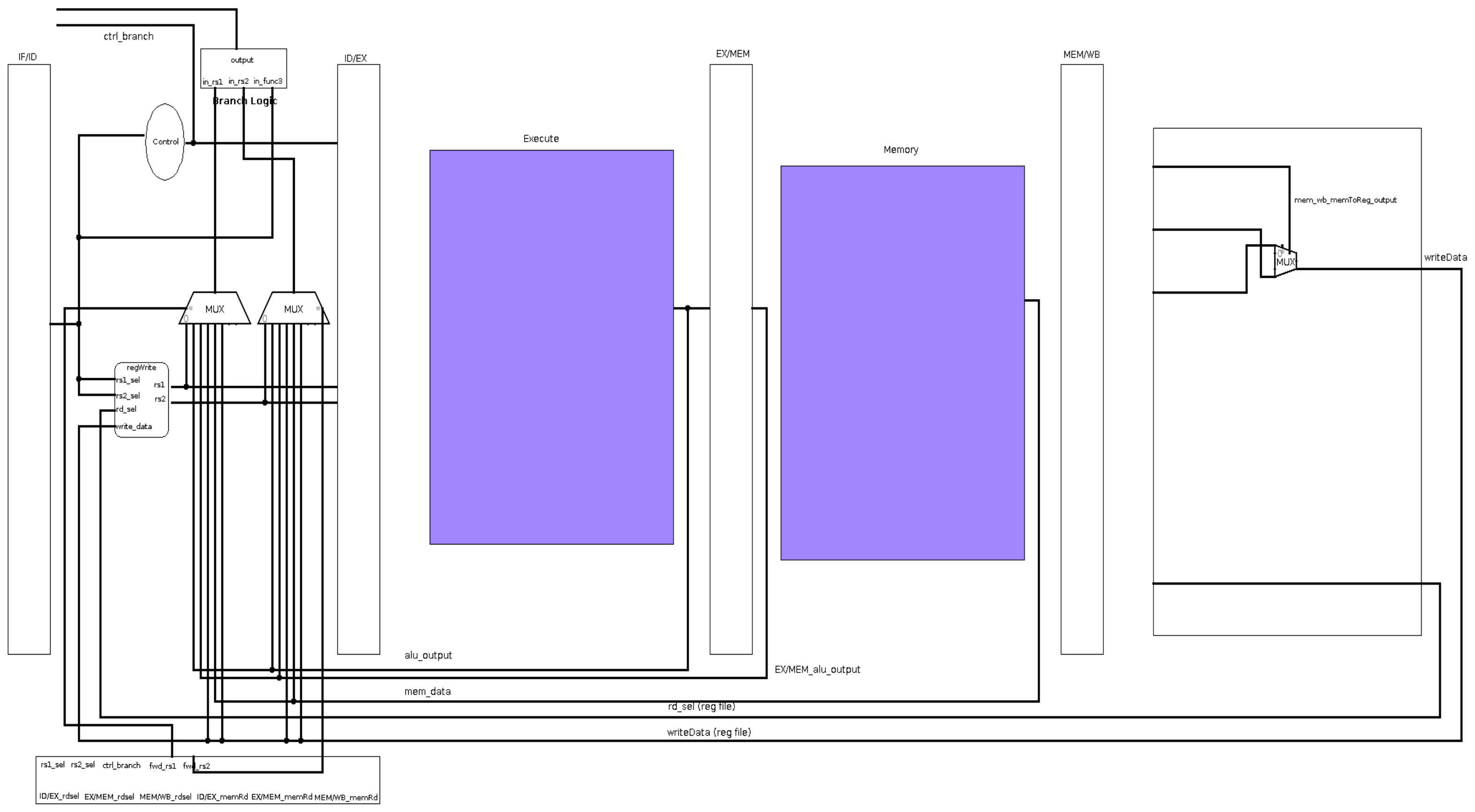
The above code is pretty clear we are forwarding the data to the rs1 input of the branch logic unit based on the forward_rs1 conditions.

```
// FOR REGISTER RS2 in BRANCH LOGIC UNIT
when(branchForward.io.forward_rs2 === "b000".U) {
  // No hazard just use register file data
  branchLogic.io.in_rs2 := reg_file.io.rs2
} .elsewhen(branchForward.io.forward_rs2 === "b001".U) {
  // hazard in alu stage forward data from alu output
  branchLogic.io.in_rs2 := alu.io.output
} .elsewhen(branchForward.io.forward_rs2 === "b010".U) {
  // hazard in EX/MEM stage forward data from EX/MEM.alu_output
  branchLogic.io.in_rs2 := EX_MEM.io.ex_mem_alu_output
} .elsewhen(branchForward.io.forward_rs2 === "b011".U) {
  // hazard in MEM/WB stage forward data from register file write data which will have correct data from the MEM/WB mux
  branchLogic.io.in_rs2 := reg_file.io.writeData
} .elsewhen(branchForward.io.forward_rs2 === "b100".U) {
  // hazard in EX/MEM stage and load type instruction so forwarding from data memory data output instead of EX/MEM.alu_output
  branchLogic.io.in_rs2 := dmem.io.memOut
} .elsewhen(branchForward.io.forward_rs2 === "b101".U) {
  // hazard in MEM/WB stage and load type instruction so forwarding from register file write data which will have the correct output
from the mux
  branchLogic.io.in_rs2 := reg_file.io.writeData
}
  .otherwise {
    branchLogic.io.in_rs2 := reg_file.io.rs2
  }
```

Similarly wiring the rs2 input of the branch logic unit based on the forward_rs2 values of the branch forwarding unit.

With these connections we have successfully implemented the Branching hazards and our datapath now supports branching instructions.

Block Diagram:



The box is shown just to hide all the details of the Execute and Memory stages. We just see the connections of the Branch logic after getting forwarded from the forward unit. Note that we also didn’t show the wiring of the inputs of the Branch forward unit since it would just complicate the diagram and make it harder to understand.

What is left now are unconditional jumps (JAL and JALR). These instructions will be implemented in the next topics.

Unconditional Jumps

In this portion we will simply add the JAL and JALR instruction facility into our datapath without thinking of the hazards for now. After implementation we will look into the hazards for these unconditional jumps.

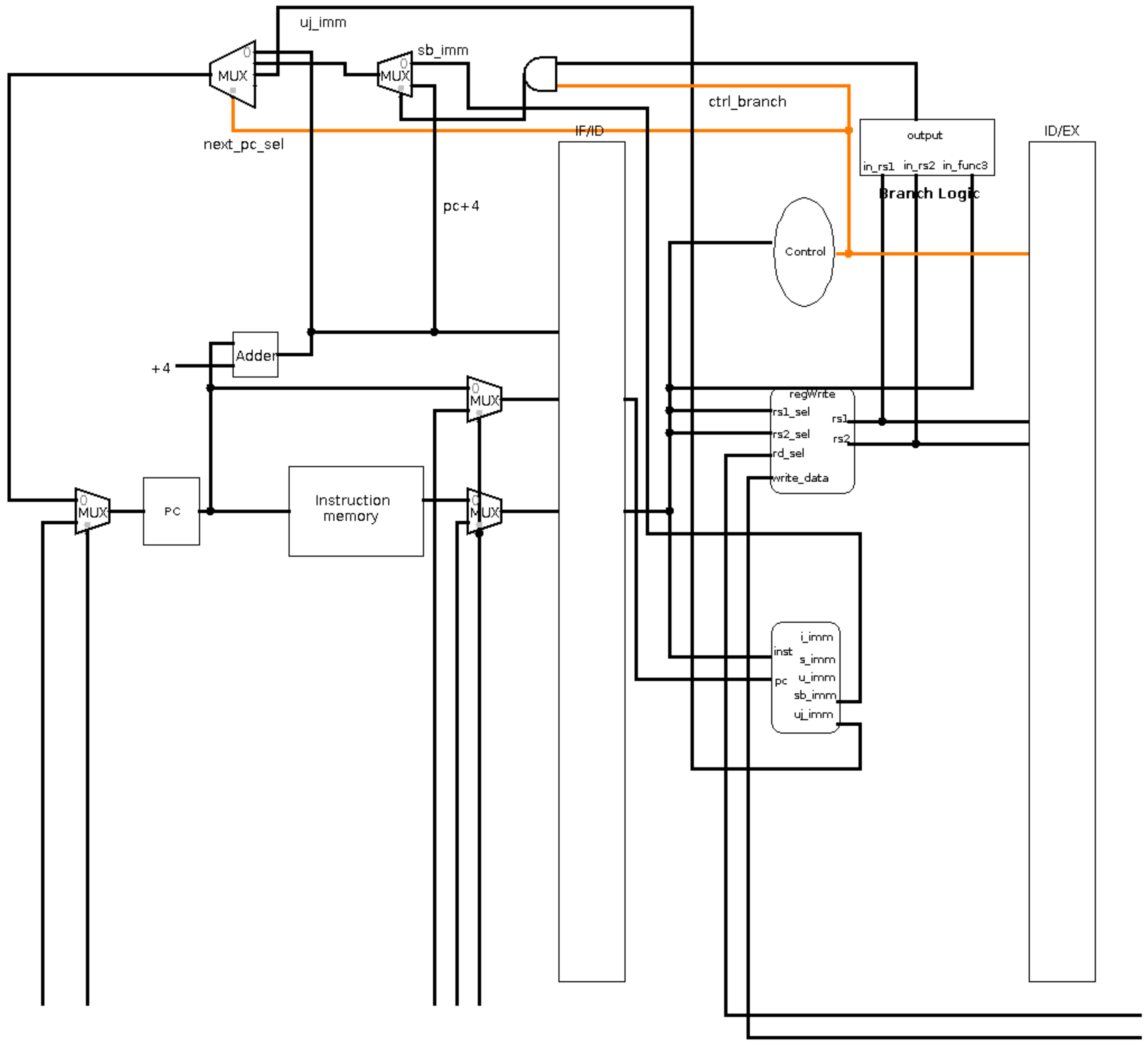
Implementing JAL in Top

For JAL instruction the destination register (rd) is used to store the address of the next instruction (pc + 4) while uj_imm value from the immediate generation is used for updating the pc to jump.

```
// The Mux before the pc which selects the input to the pc
// either from the hazard detection unit or from the adder +4 or from sb_imm
when(hazardDetection.io.pc_forward === "b1".U) {
  pc.io.in := hazardDetection.io.pc_out
} .otherwise {
    when(control.io.out_next_pc_sel === "b01".U) {
      when(branchLogic.io.output === 1.U && control.io.out_branch === 1.U) {
        pc.io.in := imm_generation.io.sb_imm
        IF_ID.io.pc_in := 0.S
        IF_ID.io.pc4_in := 0.S
        IF_ID.io.inst_in := 0.U
      } .otherwise {
        pc.io.in := pc.io.pc4
      }

    } .elsewhen(control.io.out_next_pc_sel === "b10".U) {
      pc.io.in := imm_generation.io.uj_imm
      IF_ID.io.pc_in := 0.S
      IF_ID.io.pc4_in := 0.S
      IF_ID.io.inst_in := 0.U
    }.otherwise {
      pc.io.in := pc.io.pc4
    }

}
```

For the branch condition we covered this code previously. For jal instruction we choose another condition of the nex_pc_sel as highlighted above. The next_pc_sel is 10 for UJ-Type instructions so we use the when condition to check the nex_pc_sel. Based on that we update the pc to the uj_imm value and flush the IF/ID pipeline register so that the fetched instruction becomes zero as we do not require that instruction.

Modified Block Diagram:

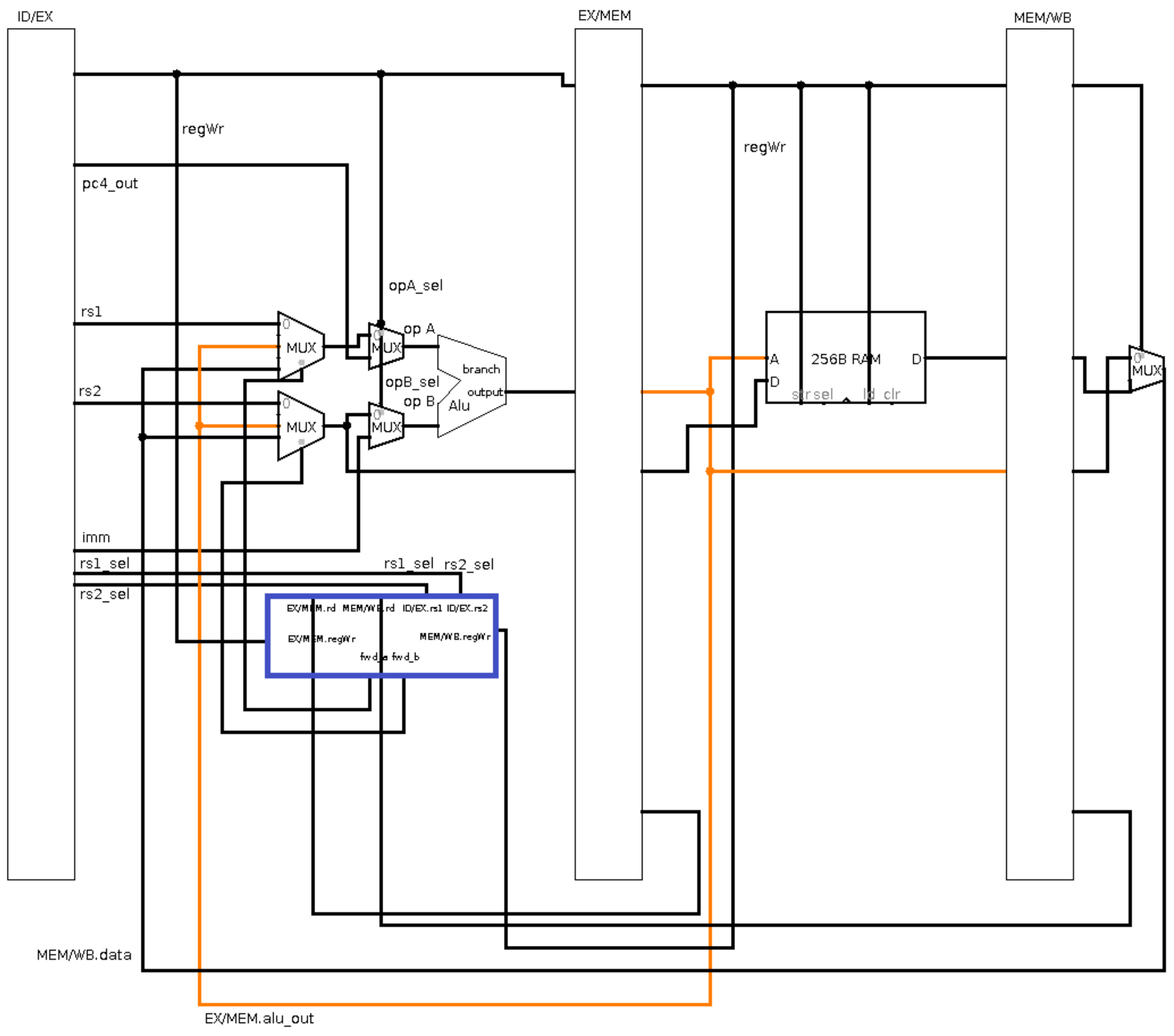


Here is the modified block diagram. We can see the uj_imm from the immediate generation in the decode stage being passed on to the mux which is controlled by the next_pc_sel from the control.

```
// Controlling Operand A for ALU
 when (ID_EX.io.ctrl_OpA_sel_out === "b10".U) {
   alu.io.oper_a := ID_EX.io.pc4_out
 } .otherwise {
     when(forwardUnit.io.forward_a === "b00".U) {
       alu.io.oper_a := ID_EX.io.rs1_out
     } .elsewhen(forwardUnit.io.forward_a === "b01".U) {
       alu.io.oper_a := EX_MEM.io.ex_mem_alu_output
     } .elsewhen(forwardUnit.io.forward_a === "b10".U) {
       alu.io.oper_a := reg_file.io.writeData
     } .otherwise {
       alu.io.oper_a := ID_EX.io.rs1_out
     }
 }
```

Here we control the input A of the ALU for jal instruction. If operand_a_sel of the control is 10 it means that the instruction is a jal instruction and we want to input the next instruction pc (pc + 4) into the input A of the ALU, otherwise we check for the forwarding conditions and forward the appropriate data into the ALU.

Block Diagram:



Here in this diagram we can see an additional mux in front of the operand A of ALU which is controlled by the opA_sel from the control and has two inputs, one from the forwarding or register file and another as pc + 4. It selects the correct data and forwards it to the operand A of the ALU.

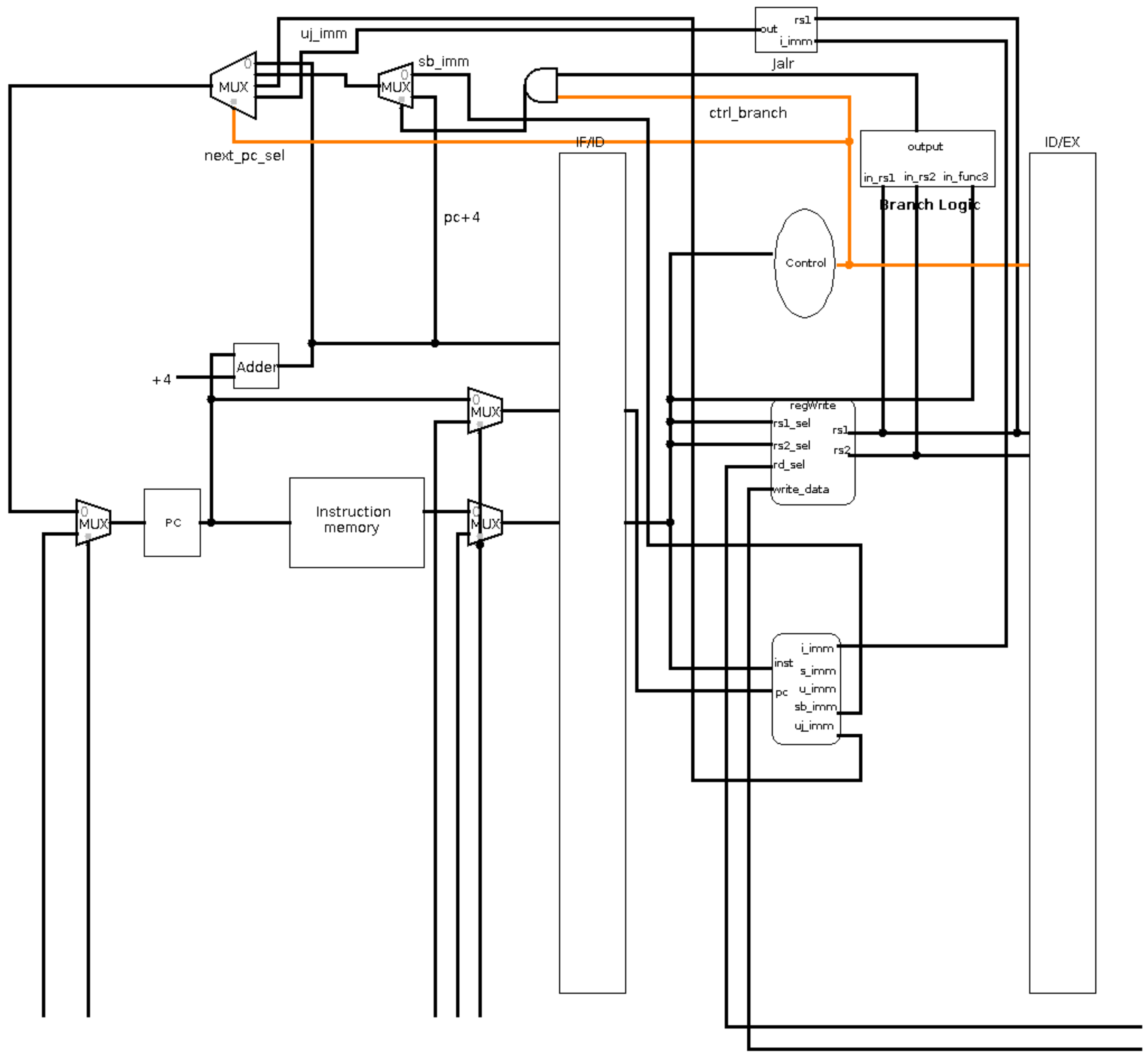
Implementing JALR in Top

For jalr instruction we have a module that takes one input from the i_imm value and one input from the source register 1 (rs1). It adds these values and outputs the immediate value to update the PC.

```
jalr.io.input_a := reg_file.io.rs1
jalr.io.input_b := imm_generation.io.i_imm


// The Mux before the pc which selects the input to the pc
// either from the hazard detection unit or from the adder +4
when(hazardDetection.io.pc_forward === "b1".U) {
  pc.io.in := hazardDetection.io.pc_out
} .otherwise {
    when(control.io.out_next_pc_sel === "b01".U) {
      when(branchLogic.io.output === 1.U && control.io.out_branch === 1.U) {
        pc.io.in := imm_generation.io.sb_imm
        IF_ID.io.pc_in := 0.S
        IF_ID.io.pc4_in := 0.S
        IF_ID.io.inst_in := 0.U
      } .otherwise {
        pc.io.in := pc.io.pc4
      }

    } .elsewhen(control.io.out_next_pc_sel === "b10".U) {
      pc.io.in := imm_generation.io.uj_imm
      IF_ID.io.pc_in := 0.S
      IF_ID.io.pc4_in := 0.S
      IF_ID.io.inst_in := 0.U
    } .elsewhen(control.io.out_next_pc_sel === "b11".U) {
      pc.io.in := jalr.io.output
      IF_ID.io.pc_in := 0.S
      IF_ID.io.pc4_in := 0.S
      IF_ID.io.inst_in := 0.U
    }
      .otherwise {
      pc.io.in := pc.io.pc4
    }

}
```

Here we have added another when condition on the next_pc_sel as highlighted above. The next_pc_sel for jalr instruction is 11. We check this condition and then update the PC input value with the jalr's output. We also flush the IF/ID pipeline register because we don't want the already fetched instruction since we have to jump to the required instruction and fetch that instruction.

Block Diagram:



Here we have implemented the Jalr module in the decode stage, which gets it’s input source register 1 (rs1) from the register file and immediate value from the i_imm of the immediate generation. It’s output then again goes to the mux which is controlled by the next_pc_sel that finally updates the pc value if the instruction is jalr.

Unconditional Jump Hazards

We don’t have any hazard for jal instructions because we simply update the pc value and jump to that instruction.

Where we do have hazards are, for jalr instruction. Because jalr instruction uses source register 1 (rs1) and an offset to compute the address where the jalr instruction jumps to. The data hazard occurs if the register rs1 value is not yet written and jalr instruction uses it. So for that purpose we need to forward the correct value.

Since we already have a forwarding unit in the decode stage named Branch Forward unit which does all the required forwarding of source register 1 (rs1) as well as of source register 2 (rs2) ***only*** when the Control’s branch signal is high. For our jalr unit purposes we need to forward only one input that is source register 1 (rs1) which jalr uses together with the i_imm from the immediate generation to calculate the next PC value.

We can reuse our Branch Forward unit by adding another condition and that is when Control’s branch pin is not high, in that case we can forward the data to the jalr inputs.

Let’s rename our Branch Forward unit to Decode Forward unit (since it is present at the Decode stage and is used for forwarding data to multiple modules) and also add some conditions for the jalr instructions.

Let’s review the changes in the code of the Decode Forward unit

Modifying Decode Forward Unit

```
class DecodeForwardUnit extends Module {
  val io = IO(new Bundle {
    val ID_EX_REGRD = Input(UInt(5.W))
    val ID_EX_MEMRD = Input(UInt(1.W))
    val EX_MEM_REGRD = Input(UInt(5.W))
    val EX_MEM_MEMRD = Input(UInt(1.W))
    val MEM_WB_REGRD = Input(UInt(5.W))
    val MEM_WB_MEMRD = Input(UInt(1.W))
    val rs1_sel = Input(UInt(5.W))
    val rs2_sel = Input(UInt(5.W))
    val ctrl_branch = Input(UInt(1.W))
    val forward_rs1 = Output(UInt(4.W))
    val forward_rs2 = Output(UInt(4.W))
  })
```

The changes here are reflected in the highlighted color. We changed the name of the class as well as increase the data bits of forward_rs1 and forward_rs2 since we need more conditions to represent now, that cannot be represented by three bits.

```
io.forward_rs1 := "b0000".U
io.forward_rs2 := "b0000".U
```

The default values are 0 of 4 bits value.

```
when(io.ctrl_branch === 1.U) {
  // ALU Hazard
  when(io.ID_EX_REGRD =/= "b00000".U && io.ID_EX_MEMRD =/= 1.U && (io.ID_EX_REGRD === 
io.rs1_sel) && (io.ID_EX_REGRD === io.rs2_sel)) {
    io.forward_rs1 := "b0001".U
    io.forward_rs2 := "b0001".U
  } .elsewhen(io.ID_EX_REGRD =/= "b00000".U && io.ID_EX_MEMRD =/= 1.U && (io.ID_EX_REGRD === 
io.rs1_sel)) {
    io.forward_rs1 := "b0001".U
  } .elsewhen(io.ID_EX_REGRD =/= "b00000".U && io.ID_EX_MEMRD =/= 1.U && (io.ID_EX_REGRD === 
io.rs2_sel)) {
    io.forward_rs2 := "b0001".U
  }

  // EX/MEM Hazard
  when(io.EX_MEM_REGRD =/= "b00000".U && io.EX_MEM_MEMRD =/= 1.U &&
    ~((io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel) && (io.ID_EX_REGRD === 
io.rs2_sel)) &&
    (io.EX_MEM_REGRD === io.rs1_sel) && (io.EX_MEM_REGRD === io.rs2_sel)) {

    io.forward_rs1 := "b0010".U
    io.forward_rs2 := "b0010".U

  } .elsewhen(io.EX_MEM_REGRD =/= "b00000".U && io.EX_MEM_MEMRD =/= 1.U &&
    ~((io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs2_sel)) &&
    (io.EX_MEM_REGRD === io.rs2_sel)) {

    io.forward_rs2 := "b0010".U

  } .elsewhen(io.EX_MEM_REGRD =/= "b00000".U && io.EX_MEM_MEMRD =/= 1.U &&
    ~((io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel)) &&
    (io.EX_MEM_REGRD === io.rs1_sel)) {

    io.forward_rs1 := "b0010".U

  } .elsewhen(io.EX_MEM_REGRD =/= "b00000".U && io.EX_MEM_MEMRD === 1.U &&
    ~((io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel) && (io.ID_EX_REGRD === 
io.rs2_sel)) &&
    (io.EX_MEM_REGRD === io.rs1_sel) && (io.EX_MEM_REGRD === io.rs2_sel)) {
    // FOR Load instructions
    io.forward_rs1 := "b0100".U
    io.forward_rs2 := "b0100".U

  } .elsewhen(io.EX_MEM_REGRD =/= "b00000".U && io.EX_MEM_MEMRD === 1.U &&
    ~((io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs2_sel)) &&
    (io.EX_MEM_REGRD === io.rs2_sel)) {

    io.forward_rs2 := "b0100".U

  } .elsewhen(io.ctrl_branch === 1.U && io.EX_MEM_REGRD =/= "b00000".U && io.EX_MEM_MEMRD ===
1.U &&
    ~((io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel)) &&
    (io.EX_MEM_REGRD === io.rs1_sel)) {

    io.forward_rs1 := "b0100".U

  }

  // MEM/WB Hazard
  when(io.MEM_WB_REGRD =/= "b00000".U && io.MEM_WB_MEMRD =/= 1.U &&
    // IF NOT ALU HAZARD
    ~((io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel) && (io.ID_EX_REGRD ===
io.rs2_sel)) &&
    // IF NOT EX/MEM HAZARD
    ~((io.EX_MEM_REGRD =/= "b00000".U) && (io.EX_MEM_REGRD === io.rs1_sel) &&
(io.EX_MEM_REGRD === io.rs2_sel)) &&
    (io.MEM_WB_REGRD === io.rs1_sel) && (io.MEM_WB_REGRD === io.rs2_sel)) {

    io.forward_rs1 := "b0011".U
    io.forward_rs2 := "b0011".U

  }
    .elsewhen(io.MEM_WB_REGRD =/= "b00000".U && io.MEM_WB_MEMRD =/= 1.U &&
      // IF NOT ALU HAZARD
      ~((io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs2_sel)) &&
      // IF NOT EX/MEM HAZARD
      ~((io.EX_MEM_REGRD =/= "b00000".U) && (io.EX_MEM_REGRD === io.rs2_sel)) &&
      (io.MEM_WB_REGRD === io.rs2_sel)) {

      io.forward_rs2 := "b0011".U

    }
    .elsewhen(io.MEM_WB_REGRD =/= "b00000".U && io.MEM_WB_MEMRD =/= 1.U &&
      // IF NOT ALU HAZARD
      ~((io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel)) &&
      // IF NOT EX/MEM HAZARD
      ~((io.EX_MEM_REGRD =/= "b00000".U) && (io.EX_MEM_REGRD === io.rs1_sel)) &&
      (io.MEM_WB_REGRD === io.rs1_sel)) {

      io.forward_rs1 := "b0011".U

    } .elsewhen(io.MEM_WB_REGRD =/= "b00000".U && io.MEM_WB_MEMRD === 1.U &&
    // IF NOT ALU HAZARD
    ~((io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel) && (io.ID_EX_REGRD ===
io.rs2_sel)) &&
    // IF NOT EX/MEM HAZARD
    ~((io.EX_MEM_REGRD =/= "b00000".U) && (io.EX_MEM_REGRD === io.rs1_sel) &&
(io.EX_MEM_REGRD === io.rs2_sel)) &&
    (io.MEM_WB_REGRD === io.rs1_sel) && (io.MEM_WB_REGRD === io.rs2_sel)) {
    // FOR Load instructions
    io.forward_rs1 := "b0101".U
    io.forward_rs2 := "b0101".U

  }
    .elsewhen(io.MEM_WB_REGRD =/= "b00000".U && io.MEM_WB_MEMRD === 1.U &&
      // IF NOT ALU HAZARD
      ~((io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs2_sel)) &&
      // IF NOT EX/MEM HAZARD
      ~((io.EX_MEM_REGRD =/= "b00000".U) && (io.EX_MEM_REGRD === io.rs2_sel)) &&
      (io.MEM_WB_REGRD === io.rs2_sel)) {

      io.forward_rs2 := "b0101".U

    }
    .elsewhen(io.MEM_WB_REGRD =/= "b00000".U && io.MEM_WB_MEMRD === 1.U &&
      // IF NOT ALU HAZARD
      ~((io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel)) &&
      // IF NOT EX/MEM HAZARD
      ~((io.EX_MEM_REGRD =/= "b00000".U) && (io.EX_MEM_REGRD === io.rs1_sel))&&
      (io.MEM_WB_REGRD === io.rs1_sel)) {

      io.forward_rs1 := "b0101".U

    }

  }
```

As highlighted above, instead of mentioning Control's branch pin in every condition we used it in the parent when condition which sets the forward_rs1 and forward_rs2 to same values when the Control's branch is 1.

```
// Forwarding for JALR unit
      .elsewhen(io.ctrl_branch === 0.U) {
        // ALU Hazard
        when(io.ID_EX_REGRD =/= "b00000".U && io.ID_EX_MEMRD =/= 1.U && (io.ID_EX_REGRD ===
           io.rs1_sel)){
          io.forward_rs1 := "b0110".U
        }

        // EX/MEM Hazard
        when(io.EX_MEM_REGRD =/= "b00000".U && io.EX_MEM_MEMRD =/= 1.U &&
          ~((io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel)) &&
          (io.EX_MEM_REGRD === io.rs1_sel)) {

          io.forward_rs1 := "b0111".U

        }
          .elsewhen(io.EX_MEM_REGRD =/= "b00000".U && io.EX_MEM_MEMRD === 1.U &&
            ~((io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel)) &&
            (io.EX_MEM_REGRD === io.rs1_sel)) {
            // FOR Load instructions
            io.forward_rs1 := "b1001".U

        }



        // MEM/WB Hazard
        when(io.MEM_WB_REGRD =/= "b00000".U && io.MEM_WB_MEMRD =/= 1.U &&
          // IF NOT ALU HAZARD
          ~((io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel)) &&
          // IF NOT EX/MEM HAZARD
          ~((io.EX_MEM_REGRD =/= "b00000".U) && (io.EX_MEM_REGRD === io.rs1_sel)) &&
          (io.MEM_WB_REGRD === io.rs1_sel)) {

          io.forward_rs1 := "b1000".U
        }
           .elsewhen(io.MEM_WB_REGRD =/= "b00000".U && io.MEM_WB_MEMRD === 1.U &&
          // IF NOT ALU HAZARD
          ~((io.ID_EX_REGRD =/= "b00000".U) && (io.ID_EX_REGRD === io.rs1_sel)) &&
          // IF NOT EX/MEM HAZARD
          ~((io.EX_MEM_REGRD =/= "b00000".U) && (io.EX_MEM_REGRD === io.rs1_sel)) &&
          (io.MEM_WB_REGRD === io.rs1_sel)) {
          // FOR Load instructions
          io.forward_rs1 := "b1010".U

        }


      }
  }
```

Then we added an elsewhen condition on the parent when condition to see if Control's branch is 0. Which means now we are forwarding the data for jalr unit. We check conditions only for rs1 and set forward_rs1 to different values.

Now let's see how we modify the Top file to connect the Branch Logic and Jalr module with this forwarding unit.

Modifying the Top file for Decode Forward Unit

```
val decodeForwardUnit = Module(new DecodeForwardUnit())
```

First of all we create the object of the class DecodeForwardUnit.

```
// Initializing Branch Forward Unit
decodeForwardUnit.io.ID_EX_REGRD := ID_EX.io.rd_sel_out
decodeForwardUnit.io.ID_EX_MEMRD := ID_EX.io.ctrl_MemRd_out
decodeForwardUnit.io.EX_MEM_REGRD := EX_MEM.io.ex_mem_rdSel_output
decodeForwardUnit.io.MEM_WB_REGRD := MEM_WB.io.mem_wb_rdSel_output
decodeForwardUnit.io.EX_MEM_MEMRD := EX_MEM.io.ex_mem_memRd_out
decodeForwardUnit.io.MEM_WB_MEMRD := MEM_WB.io.mem_wb_memRd_output
decodeForwardUnit.io.rs1_sel := IF_ID.io.inst_out(19, 15)
decodeForwardUnit.io.rs2_sel := IF_ID.io.inst_out(24, 20)
decodeForwardUnit.io.ctrl_branch := control.io.out_branch
```

Wiring the decode forward unit with the correct sources. Same as before.

```
// FOR REGISTER RS1 in BRANCH LOGIC UNIT and JALR UNIT

// These forwarding values come only when the Control's branch pin is high which means SB-Type instruction
is in the decode stage so we don't need to forward any values to the JALR unit. Hence for all these conditions
we wire JALR unit with register file's output by default.

when(decodeForwardUnit.io.forward_rs1 === "b0000".U) {
  // No hazard just use register file data
  branchLogic.io.in_rs1 := reg_file.io.rs1
  jalr.io.input_a := reg_file.io.rs1
} .elsewhen(decodeForwardUnit.io.forward_rs1 === "b0001".U) {
  // hazard in alu stage forward data from alu output
  branchLogic.io.in_rs1 := alu.io.output
  jalr.io.input_a := reg_file.io.rs1
} .elsewhen(decodeForwardUnit.io.forward_rs1 === "b0010".U) {
  // hazard in EX/MEM stage forward data from EX/MEM.alu_output
  branchLogic.io.in_rs1 := EX_MEM.io.ex_mem_alu_output
  jalr.io.input_a := reg_file.io.rs1
} .elsewhen(decodeForwardUnit.io.forward_rs1 === "b0011".U) {
  // hazard in MEM/WB stage forward data from register file write data which will have correct data from the
MEM/WB mux
  branchLogic.io.in_rs1 := reg_file.io.writeData
  jalr.io.input_a := reg_file.io.rs1
} .elsewhen(decodeForwardUnit.io.forward_rs1 === "b0100".U) {
  // hazard in EX/MEM stage and load type instruction so forwarding from data memory data output instead
of EX/MEM.alu_output
  branchLogic.io.in_rs1 := dmem.io.memOut
  jalr.io.input_a := reg_file.io.rs1
} .elsewhen(decodeForwardUnit.io.forward_rs1 === "b0101".U) {
  // hazard in MEM/WB stage and load type instruction so forwarding from register file write data which will
have the correct output from the mux
  branchLogic.io.in_rs1 := reg_file.io.writeData
  jalr.io.input_a := reg_file.io.rs1
}
```

Here we are simply connecting the sources with the Branch Logic module as we get these values only when the Control's branch pin is high. Meanwhile we set jalr's input_a which is it's source register 1 (rs1) input to the register file's rs1 output since we won't be using jalr module here in this condition

```
// These forwarding values come only when the Control's branch pin is low which means JALR
  // instruction maybe in the decode stage so we don't need to forward any values to the Branch Logic unit
  // Hence for all these conditions we wire Branch Logic unit with register file's output by default.

  .elsewhen(decodeForwardUnit.io.forward_rs1 === "b0110".U) {
    // hazard in alu stage forward data from alu output
    jalr.io.input_a := alu.io.output
    branchLogic.io.in_rs1 := reg_file.io.rs1
} .elsewhen(decodeForwardUnit.io.forward_rs1 === "b0111".U) {
    // hazard in EX/MEM stage forward data from EX/MEM.alu_output
    jalr.io.input_a := EX_MEM.io.ex_mem_alu_output
    branchLogic.io.in_rs1 := reg_file.io.rs1
} .elsewhen(decodeForwardUnit.io.forward_rs1 === "b1000".U) {
    // hazard in MEM/WB stage forward data from register file write data which will have correct data from the
MEM/WB mux
    jalr.io.input_a := reg_file.io.writeData
    branchLogic.io.in_rs1 := reg_file.io.rs1
} .elsewhen(decodeForwardUnit.io.forward_rs1 === "b1001".U) {
    // hazard in EX/MEM stage and load type instruction so forwarding from data memory data output instead
of EX/MEM.alu_output
    jalr.io.input_a := dmem.io.memOut
    branchLogic.io.in_rs1 := reg_file.io.rs1
} .elsewhen(decodeForwardUnit.io.forward_rs1 === "b1010".U) {
    // hazard in MEM/WB stage and load type instruction so forwarding from register file write data which will
have the correct output from the mux
    jalr.io.input_a := reg_file.io.writeData
    branchLogic.io.in_rs1 := reg_file.io.rs1
}
  .otherwise {
    branchLogic.io.in_rs1 := reg_file.io.rs1
    jalr.io.input_a := reg_file.io.rs1
}
```

Then here we check for forward_rs1 conditions that will only come when Control's Branch pin is low which means now we have to forward the data to the Jalr's input_a. In the meanwhile we wire Branch Logic's input to the default register file's rs1 output since we won't be using Branch Logic unit in this condition.

All of these conditions were only for the first input of both the modules. Since our Jalr module's second input is hardwired with i_imm from the immediate generation, we don't need to further check conditions for it.

```
jalr.io.input_b := imm_generation.io.i_imm
```

The next conditions are for the second input of the Branch Logic unit which is the same as before.

```
// FOR REGISTER RS2 in BRANCH LOGIC UNIT
when(decodeForwardUnit.io.forward_rs2 === "b0000".U) {
  // No hazard just use register file data
  branchLogic.io.in_rs2 := reg_file.io.rs2
} .elsewhen(decodeForwardUnit.io.forward_rs2 === "b0001".U) {
  // hazard in alu stage forward data from alu output
  branchLogic.io.in_rs2 := alu.io.output
} .elsewhen(decodeForwardUnit.io.forward_rs2 === "b0010".U) {
  // hazard in EX/MEM stage forward data from EX/MEM.alu_output
  branchLogic.io.in_rs2 := EX_MEM.io.ex_mem_alu_output
} .elsewhen(decodeForwardUnit.io.forward_rs2 === "b0011".U) {
  // hazard in MEM/WB stage forward data from register file write data which will have correct data from the
MEM/WB mux
  branchLogic.io.in_rs2 := reg_file.io.writeData
} .elsewhen(decodeForwardUnit.io.forward_rs2 === "b0100".U) {
  // hazard in EX/MEM stage and load type instruction so forwarding from data memory data output instead
of EX/MEM.alu_output
  branchLogic.io.in_rs2 := dmem.io.memOut
} .elsewhen(decodeForwardUnit.io.forward_rs2 === "b0101".U) {
  // hazard in MEM/WB stage and load type instruction so forwarding from register file write data which will
have the correct output from the mux
  branchLogic.io.in_rs2 := reg_file.io.writeData
}
  .otherwise {
    branchLogic.io.in_rs2 := reg_file.io.rs2
  }
```

Structural Hazards:

Now we have solved the data and control hazards, let’s move towards structural hazards.

Structural hazards occur when an instruction wants to use a resource in the datapath which is already occupied by some other instruction. Structural hazards can occur in the following two places:

1) Memory

2) Register file

If one instruction is writing to the memory (is in Memory stage) and if another instruction is being fetched from the same memory (is in Fetch stage) then a structural hazard may occur. But in our case since we have separate memories for both the instructions and data we bypass this structural hazard.

But what happens when an instruction is in the Write Back stage and another instruction is in the decode stage. One instruction is using the register file to write data into it and another instruction is reading the same register file. If the instruction being decoded is dependent on the instruction in the write back stage we will not get the updated value.

Let’s visualize the above scenario:

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
Addi x2,x0,10	IM	REG	ALU	DM	REG			
Addi x3,x0,5		IM	REG	ALU	DM	REG		
Add x4,x5,x0			IM	REG	ALU	DM	REG	
Addi x6, x2, 1				IM	REG	ALU	DM	REG

Here during CC5 we have the last instruction which is dependent on the first instruction. The first instruction is writing back to the register file and the last instruction is reading the register file. The last instruction will not get the updated value of x2 from the register file since we cannot read and write at the same (structural hazard).

What we can do is detect this hazard and forward the value that is being written back in the register file to the ID/EX pipeline register as well instead of taking the rs1 or rs2 value from the register file.

Uptill now our ID/EX pipeline register takes source register 1 (rs1) and source register 2 (rs2) data contents directly from the register file, but it will cause us errors in this scenario, so we need to forward the correct updated data that is being written back to the register file into this pipeline register so that we get the updated value.

Let's see how we can solve this issue. We will create a Structural Detector unit in the decode stage that will detect this hazard and forward the right value to the ID/EX pipeline register.

Coding the Structural Detector Unit:

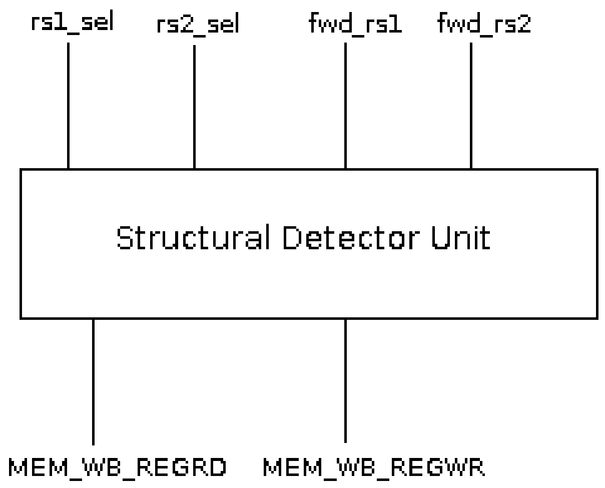
```
class StructuralDetector extends Module {
  val io = IO(new Bundle {
    val rs1_sel = Input(UInt(5.W))
    val rs2_sel = Input(UInt(5.W))
    val MEM_WB_regWr = Input(UInt(1.W))
    val MEM_WB_REGRD = Input(UInt(5.W))
    val fwd_rs1 = Output(UInt(1.W))
    val fwd_rs2 = Output(UInt(1.W))
  })
   (1)
  when(io.MEM_WB_regWr === 1.U &&  io.MEM_WB_REGRD === io.rs1_sel) {
    io.fwd_rs1 := 1.U
  } .otherwise {
    io.fwd_rs1 := 0.U
  }
   (2)
  when(io.MEM_WB_regWr === 1.U && io.MEM_WB_REGRD === io.rs2_sel) {
    io.fwd_rs2 := 1.U
  } .otherwise {
    io.fwd_rs2 := 0.U
  }

}
```

Let's discuss the inputs and outputs of this unit. It will get rs1_sel and rs2_sel from the current instruction in the decode stage for analyzing the source register numbers being used. It also takes the MEM_WB_REGRD which is the rd_sel from the MEM/WB pipeline register that provides us with the destination register number. We also take the MEM_WB_regWr control signal that tells us whether the instruction in the write back stage actually writes in the register file or not. Then we have two outputs fwd_rs1 and fwd_rs2 which are used as the selection pins of the mux in the Top file that will control which value to pass into the ID/EX Pipeline register.

(1) Here we are checking if the regWr control signal is 1 which will be true if the instruction wants to write in the register file and we check if the destination register number matches the source register 1 number then it means that the current instruction's rs1 is dependent on the write back instruction's rd. So we set the fwd_rs1 to 1, otherwise fwd_rs1 is set to 0.

(2) Similarly, we check the same condition for rs2 of the instruction and check it's dependency.

Block Diagram:



Connecting Structural Detector in Top:

(1)

```
structuralDetector.io.rs1_sel := IF_ID.io.inst_out(19, 15)
structuralDetector.io.rs2_sel := IF_ID.io.inst_out(24, 20)
structuralDetector.io.MEM_WB_REGRD := MEM_WB.io.mem_wb_rdSel_output
structuralDetector.io.MEM_WB_regWr := MEM_WB.io.mem_wb_regWr_output
```

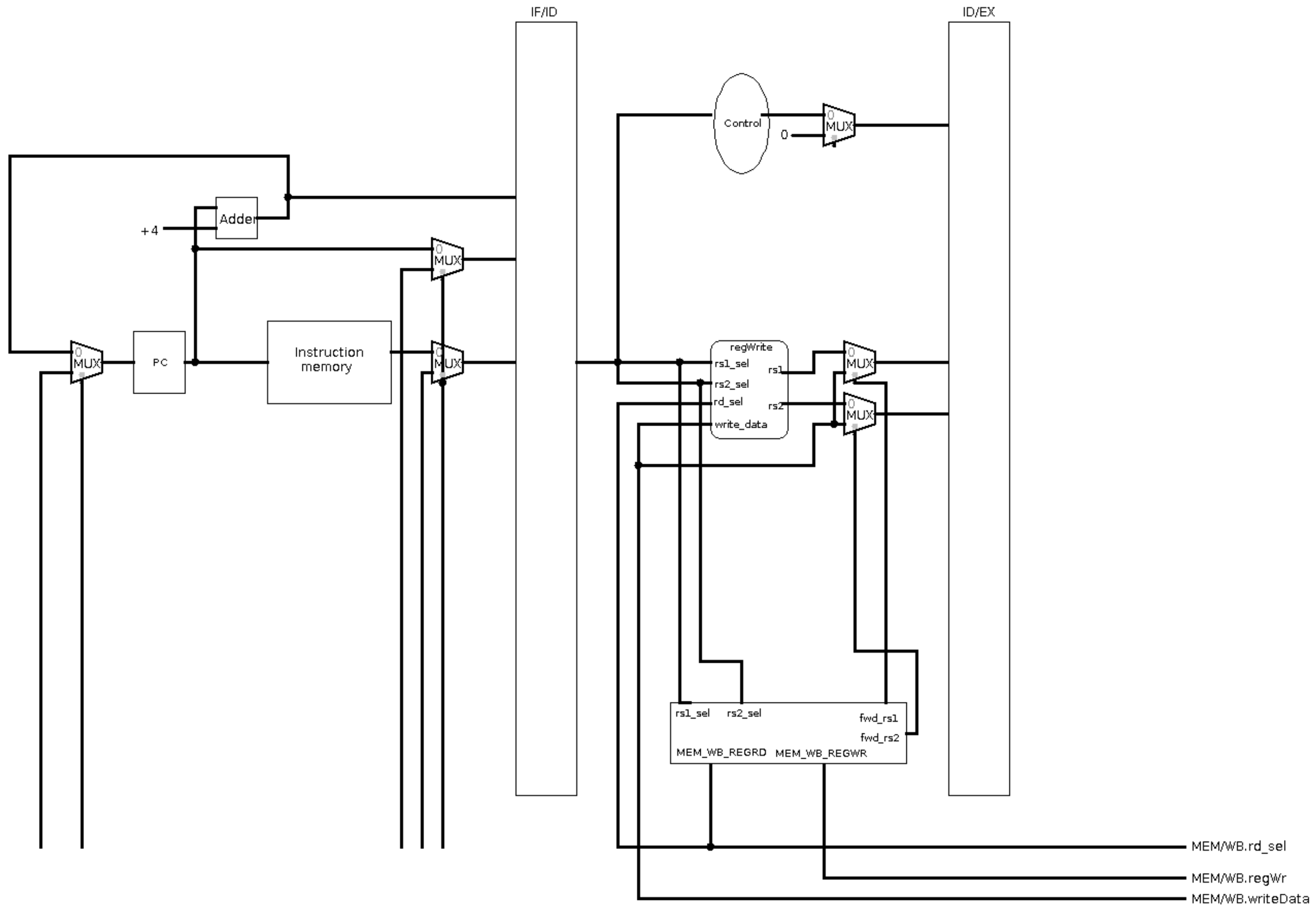
(2)

```
// FOR RS1
when(structuralDetector.io.fwd_rs1 === 1.U) {
  ID_EX.io.rs1_in := reg_file.io.writeData
} .otherwise {
  ID_EX.io.rs1_in := reg_file.io.rs1
}
// FOR RS2
when(structuralDetector.io.fwd_rs2 === 1.U) {
  ID_EX.io.rs2_in := reg_file.io.writeData
} .otherwise {
  ID_EX.io.rs2_in := reg_file.io.rs2
}
```

(1) Here we are just connecting the inputs of the structural detector unit with the correct sources.

(2) Now we check for the fwd_rs1 pin and fwd_rs2 pin for selecting the correct input to the ID/EX pipeline register. If fwd_rs1 or fwd_rs2 is 1 it means there is a hazard so we need to forward the value that is being written in the register file to the ID/EX pipeline register as well, otherwise we just wire it with the register file's data.

Block Diagram:



Here we can see that additional muxes are connected at the output of the register file which selects the proper data to the ID/EX pipeline register.