Lab4
Ke Ding
Zhanglu Wang
Changsheng Su

Introduction:
In this lab we are implementing dual issues pipeline. We duplicate some of the functional part such ALU, controller, and input and output port of the register file, which allows two instructions at the same cycle. Our goal is to implement such structure and set up stall and forwarding to deal with hazard.

Illustration of instructions:
The first step is to make it a dual port ROM. For branch prediction, we are using the predictor in step 3 for our last lab which is two level correlating global branch predictor. Only one instruction can have access to this.

In this lab, we will implement a superscalar dual-issue processor with in-order scheduling.
The list of hazards we need to consider is:
1) Data hazard in same cycle
2) Data hazard in different cycle.
3) Hazard in branch or jump
4) Hazard in memory

For instruction hazard in the same cycle. We let the second instruction to perform no operation operation and execute the second one in next cycle
For data hazard in different cycle, we forward the data to where it its needed.
For the hazard in branch or jump, we let another instruction to perform no operation operation and predicts the result for branch. If another instruction is not happening our branch adder we let us to perform the instruction in the calculated PC.
For hazard in memory, we allow only one instruction to access memory and another instruction will have to wait for the next cycle.

Design methodology:

Our strategy is to create a sub-datapath, which doesn't include reg-file and memory and duplicate this sub-datapath and implement them all together in the pipeplie, together with a hazard unit to complete the program.

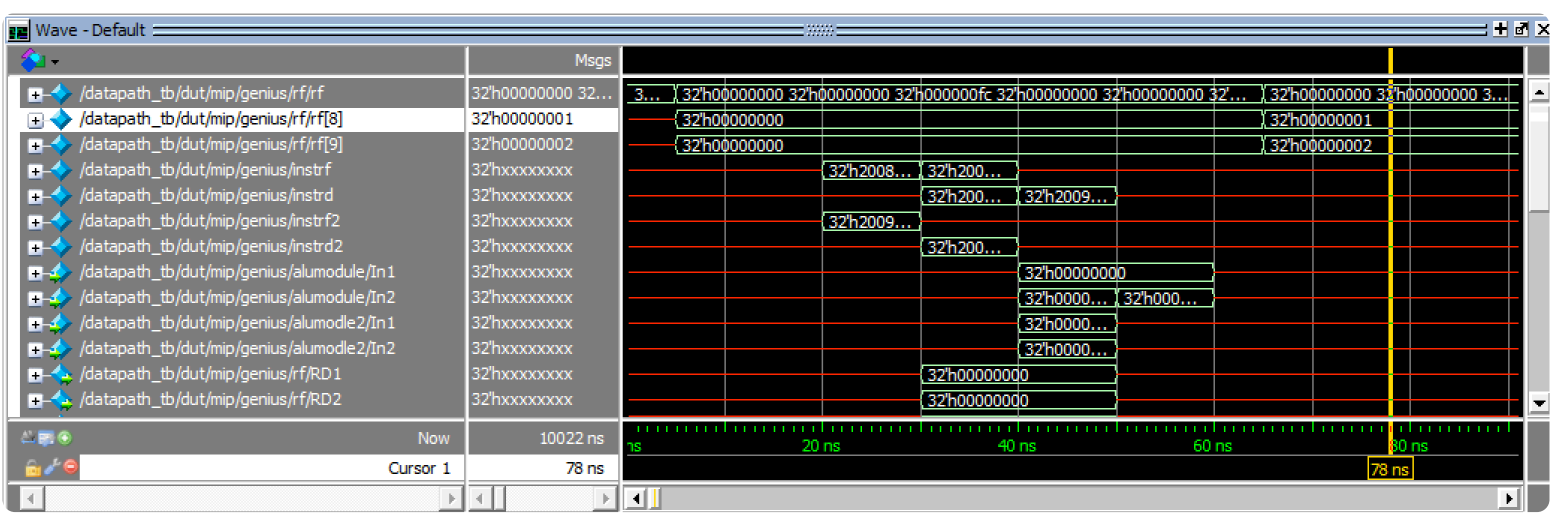
Step 1 & 2:
For step 1 we simply duplicate the datapath and remove reg file and memory out out the datapath. Then make connection between those module. This case we do not need to consider the hazard, thus we can ignore the forwarding MUX in the depth.
We now test with two consecutive add instruction and we don't worry about the hazard for this step.

The instruction we are running is

```
20080001
20090002
```

That is: ADDI $t0 $zero 0x0001
ADDI $t1 $zero 0x0002

The waveform will be displaced below:

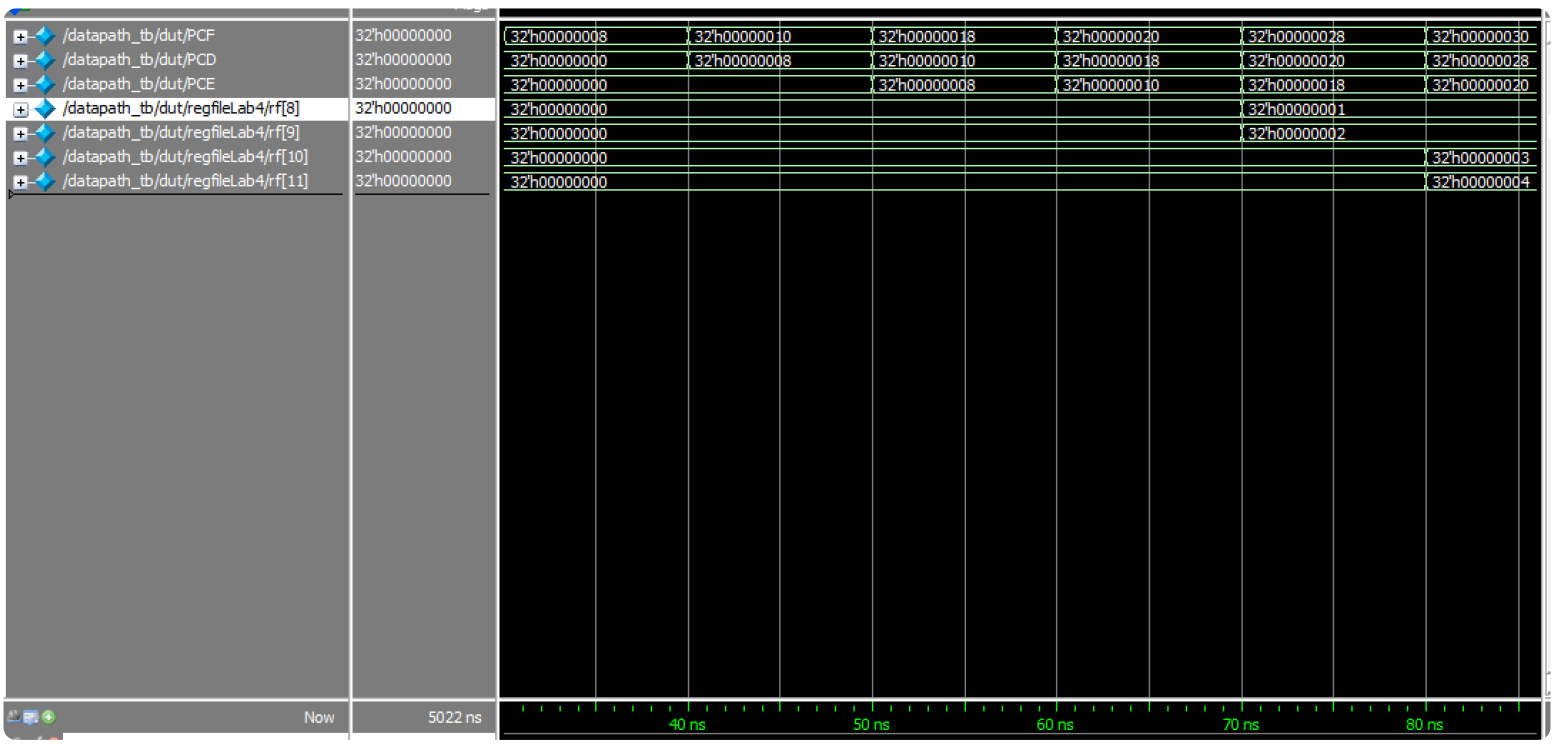


As represent above, we can see rf[8] and rf[9] which is t0 and t1 location in reg file, their datas are stored in the same number of cycle. Every instruction is execute in parallel.

Step 3:
We need to consider more than two instructions, that means there will be multiple cycle of loading the instruction to instruction memory. We need to modifies our PC+4 to Pc+8. However we will also use PC+4 later on, so we will give them a MUX, and decide which one to use later on.

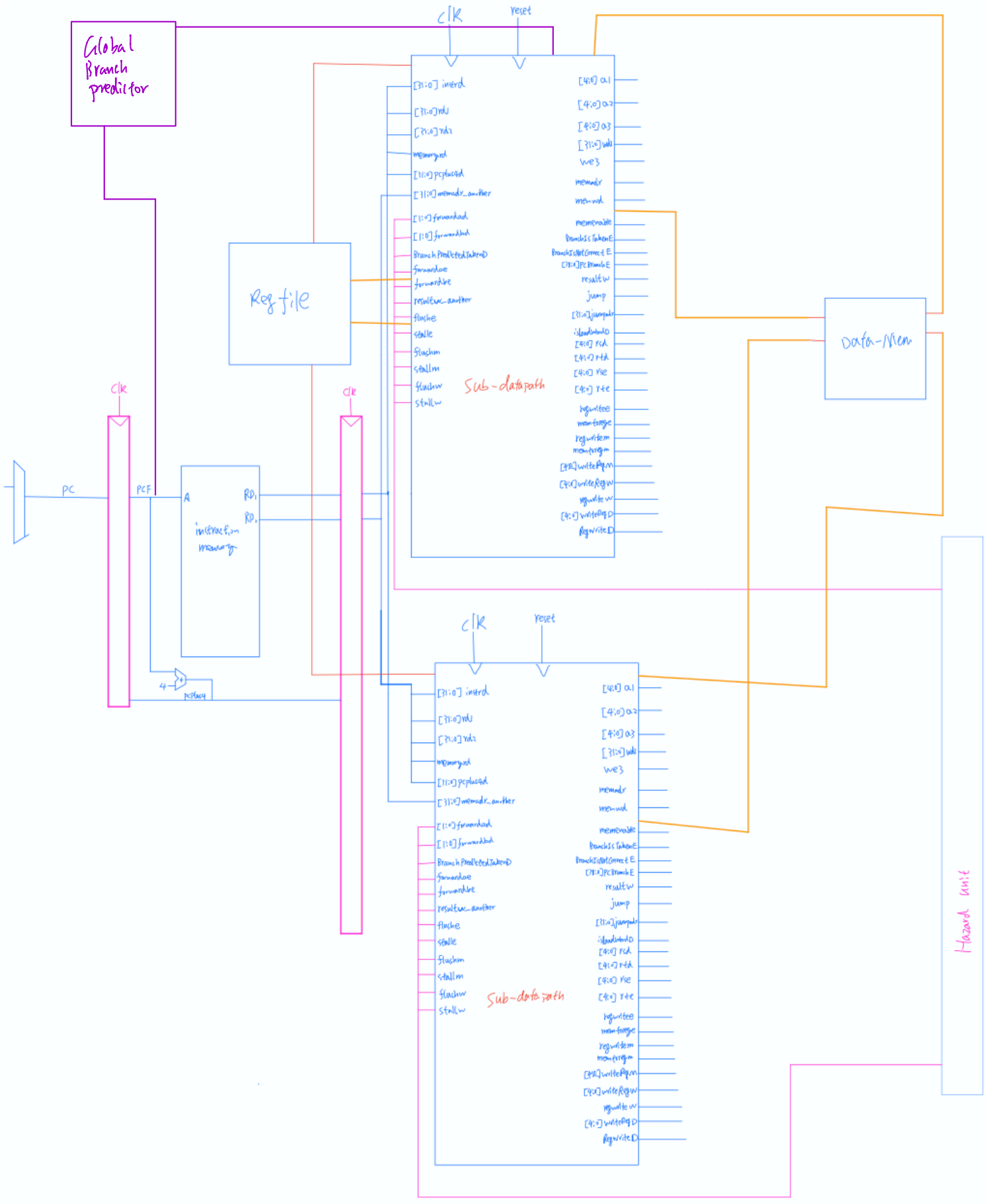
The instruction we are use to test is:
20080001 addi $t0 $zero 0x0001
20090002 addi $t1 $zero 0x0002
200a0003 addi $t2 $zero 0x0003
200b0004 addi $t3 $zero 0x0004

The waveform will be displaced below:



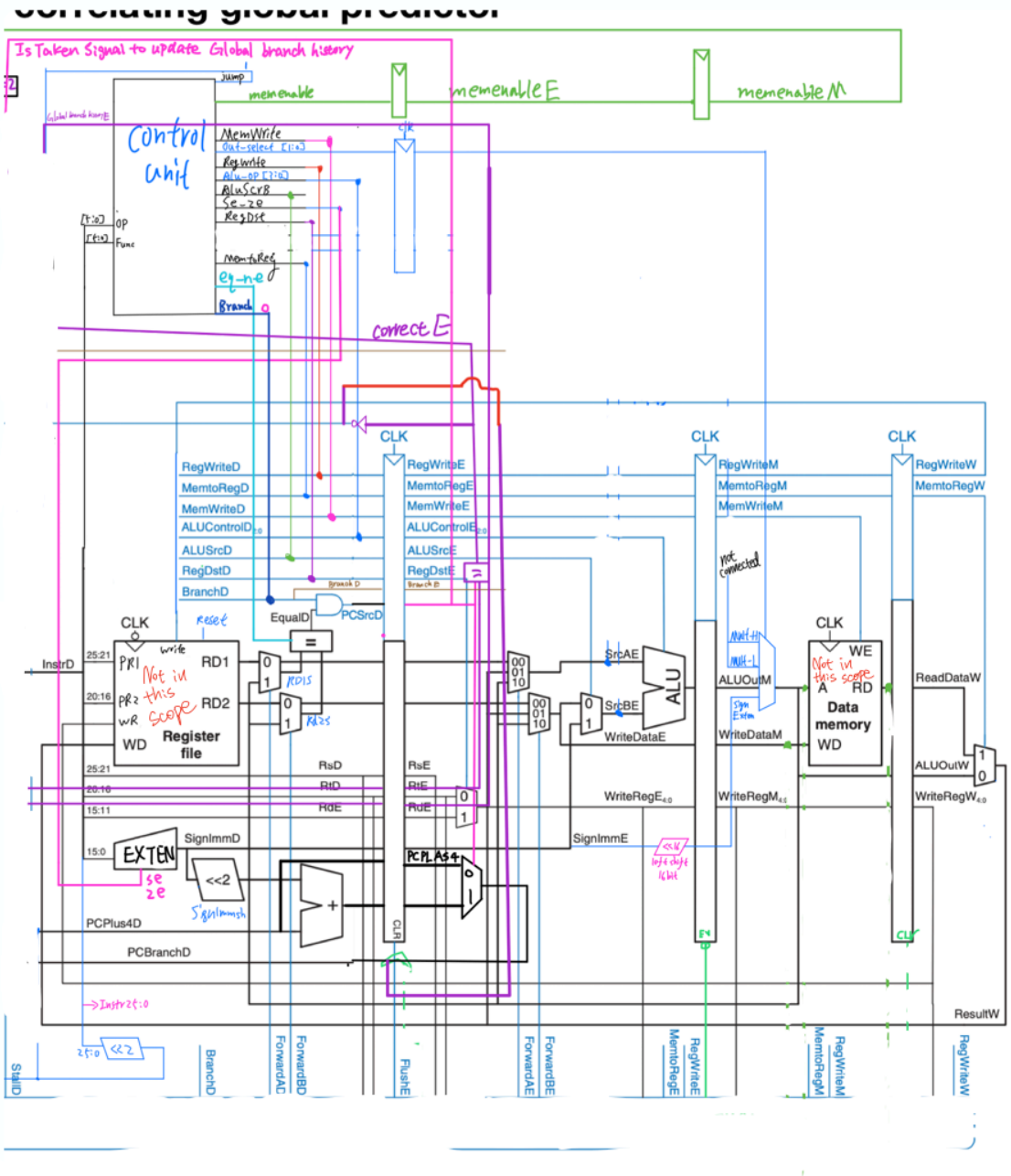
As we can see, this time PC increments by 8 each cycle, and we are not considering hazard at this point, and we can see every two instructions they store the data into the regale in parallel. rf[8] and rf[9] stores 1 and 2 in parallel and rf[10] and rf[11] stores 3 and 4 in parallel.

Step 4:



Above is general idea of our pipeline.

Below is the detail of the sub-datapath, notice that reg file and data memory will not be in the scope:



To let the sub-datapath interact with reg file, the sub-datapath has output (PR1, PR2, WR, WD) which will become input of the reg file, and the sub-datapath will have input RD1 and RD2 which are from the output of the reg file.
To let the sub-datapath interact with reg file, the sub-datapath has output of (memaddr, WD, WE, memwrite) which becomes the input of the memory and the sub-datapath has the input of readData which is the output of the memory.

It includes PC reg, reg file, instruction memory sub-datapath, data memory, hazard unit and all 5 stages register. Sub-datapath starts from decode stage end with write back stage. The general flow is like this. Instruction memory retrieve instructions from our testfile and output two instruction for our input PC value. Then it input the instruction to fetch to decode stage register. From fetch to decode stage register, it has output of two instruction and connect to the register file and sub-datapath. There are two sub-datapath deal with two instructions. Each sub-datapath includes decode stage, execute stage, memory stage, and write back stage. In memory stage the memory is placed outside the sub-datapath and will be used when it is needed in memory stage. In decode stage reg file is placed out side of the sub-datapath and will be used when it is needed in decode stage.

The hazard units have input and output of:

If instr[PC] is branch, sub-datapath 2 will perform no op, pc_next will be predictable address.

If instr[PC+4] is branch, sub-datapath 2 will perform no op, and pc_next will be PC+4
If instr[PC] and instr[PC+4] is memory operation, sub-datapath 2 will perform no op and pc_next = PC+4

If instr[PC+4]'s Rs/Rt = instr[PC]'s WriteRegF, it will send out stall signal
If there is a load word hazard, the pipeline needs to stall.
If the previous cycle contains lw, and need to use rs or rt in this instruction. This instruction will become no op for both sub-datapath.
If our branch prediction fails, it will send out a signal to flush_all, and send a signal load new instruction.

Forward:

for same subdatapath
If write_reg in memory stage = rs or rt in execute stage
data in sub-datapath in memory stage will forward to execute stage of sub-datapath.

For the same subdatapath
if write_reg in write back stage = rs or rt in execute stage
write_back data in sub-datapath in write back stage will forward to execute stage of sub-datapath

For the different subdatapath
If write_reg in memory stage in sub-datapath 2 = rs or rt in execute stage in sub-datapath 1,
data in memory stage in sub-datapath 2 will forward to rs or rt in execute stage in sub-datapath 1.

For the different subdatapath
if writte_reg in write back stage in sub-datapath 2 = rs or rt in execute stage of sub-datapath 1
data in write back stage in sub-datapath 2 will forward to execute stage in sub-datapath 1.

sub-datapath 1 can forward to sub-datapath 2 using the same idea.

There will be a five input MUX in execute stage before ALU and hazard unit will decide which data will be forwarded in to the ALU

A-Rs is just the input without any hazard, M-slotA is memory data for sub-datapath 1 forwarding outcome. W-slotA is data in write backstage in sub-datapath 1 forwarding outcome. M-slotB is memory data for sub-datapath 2 forwarding outcome. W-slotB is data in write backstage in sub-datapath 2 forwarding outcome.

Write back stage also needs to forward to decode stage sometimes. If register's read address A1 or A2 equals to to any sub-datapath write_reg_w the forwarding will take place.

Memory stage also needs to forward to decode stage when there is a branch operation. If the input of the memory which is the result of the ALU in memory stage, it can be forwarded to the quality comparator through two new 3-input multiplexers.

Step 5 & step 6

This times, we implement full detail of the hazard unit and test for the program that includes hazard for different instruction.

The instruction are shown below:
20080003 addi t0 zero 0x3 t0 = 3
20090004 addi t1 zero 0x4 t1 = 4
01095020 add t2 t0 t1 // might hang here t2 = 7

01495820 add t3 t2 t1 //   t3 = 11
200C0005 addi t4 zero 0x5 t4 = 5
200D0006 addi t5 zero 0x6 t5 = 6
0188702A slt t6 t4 t0 t6 = 0, t4 > t0

0109682A slt t5 t0 t1 t5 = 1, t0 < t1
11AE0004 beq t5 t6 0x4 should not taken, t5!=t6
20090005 addi t1 zero 0x5 t1 = 5
15AE0004 bne t5 t6 0x4 taken t5!=t6

20080006 addi t0 zero 0x6 this should not happened

01285822 sub t3 t1 t0 t3 = 2
01285820 add t3 t1 t0 t3 = 8

AC090000 sw t1 0x0 zero Mem[0x0] = 5
AC0A0004 sw t2 0x4 zero Mem[0x4] = 7
AC0B0008 sw t3 0x8 zero Mem[0x8] = 8
AC0C000C sw t4 0xc zero Mem[0xc] = 5

8C0D0000 lw t5 0x0 zero t1 = 5
8C0E0008 lw t6 0x8 zero t6 = 8
8C08000C lw t0 0xc zero t0 = 5
8C0F0004 lw t7 0x4 zero t7 = 7
010FC020 add t8 t0 t7 t8 = 12 = C

00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000

The waveform is shown below.

Let's look at our first four instruction,

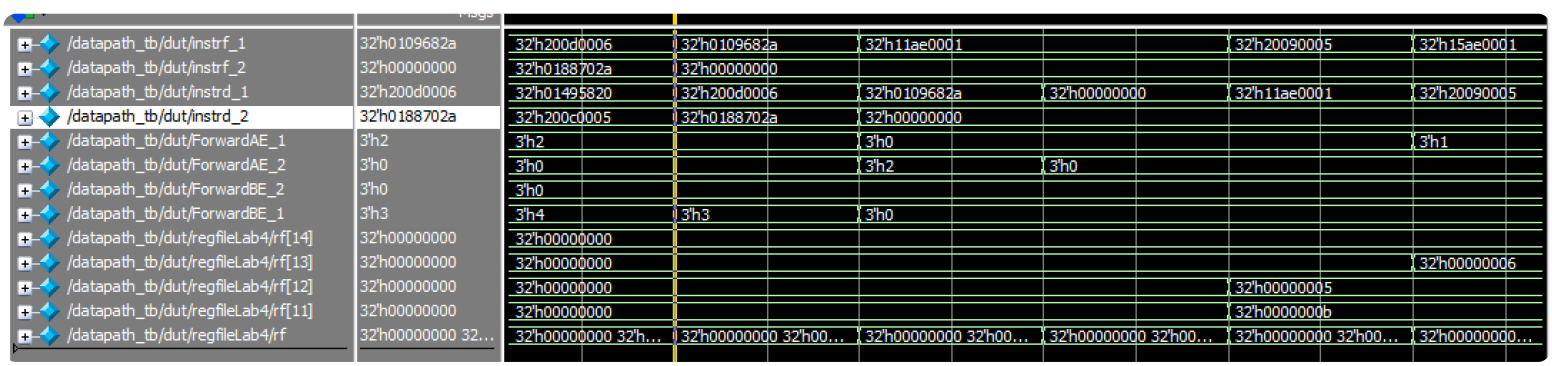
20080003 addi t0 zero 0x3 t0 = 3
20090004 addi t1 zero 0x4 t1 = 4
01095020 add t2 t0 t1 // might hang here t2 = 7
01495820 add t3 t2 t1 // t3 = 11

There will be a data hazard at line 3, add t2 t0 t1, (t2 is rf[10], t1 is rf[9], t0 is rf[8]) Line 3 needs data from line 1 and line 2, thus we forward their ALU result to line three, as you can see data '7' is stored in rf[10] right after rf[8] and rf[9] stores 3 and 4.
At the same time, line 4 has a data hazard, it needs data from line 3, (t2). What we did is, when loading two instruction at the same cycle, since line 3 and line 4 were supposed to be executed at the same cycle, however there is data hazard which t3 needs data from t2. Thus, we need to first let the fourth line become no op operation and wait for next cycle to execute the line 4. As you can see, nothing is stored in rf[11] (no op operation) when rf[10] is stored, but rf[11] will store 0xb in the next cycle. Originally this instruction was in sub-datapath 2, since no op is performed, we move this instruction to next cycle, and it will be loaded in the sub-datapath 1.

Now, lets look at these instructions:

01495820 add t3 t2 t1 // t3 = 11
200C0005 addi t4 zero 0x5 t4 = 5

200D0006 addi t5 zero 0x6 t5 = 6
0188702A slt t6 t4 t0 t6 = 0, t4 > t0

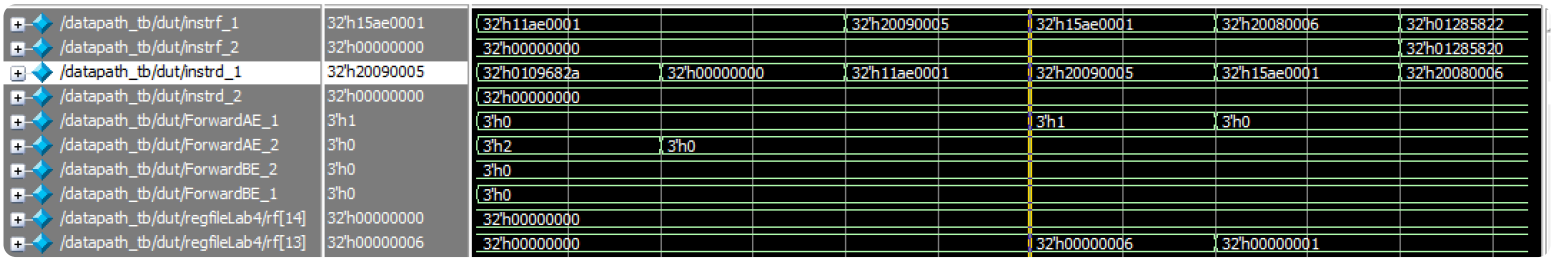


The waveform for the corresponding instruction is shown below:

We can see hazard in fourth line where t6 needs data from rf[14], (t4 in line 2)
What it does is simply forward that data from sub-datapath 2 to sub-datapath 2 in the next cycle.

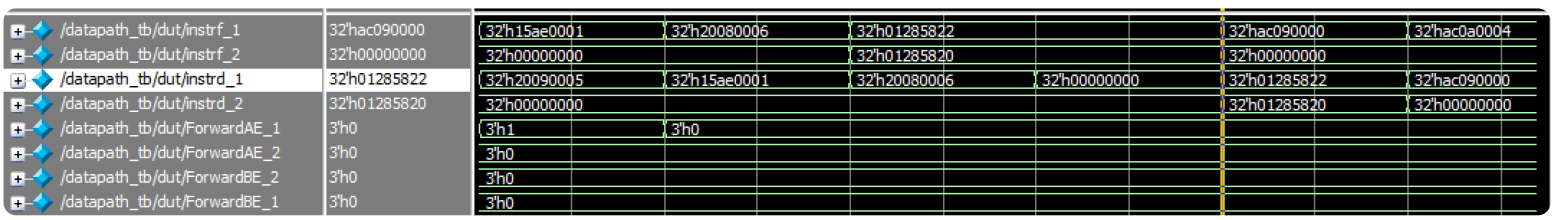
Let's look at the instrd_2 after 0188702a, it that is the execute stage for this instruction and we see a forwarding signal is triggered and set to 2, so we know that the data is forwarding from previous cycle successfully.

Let's look at instruction:
0109682A slt t5 t0 t1 t5 = 1, t0 < t1
11AE0004 beq t5 t6 0x4 should not taken, t5!=t6
20090005 addi t1 zero 0x5 t1 = 5
15AE0004 bne t5 t6 0x4 taken t5!=t6



The waveform is described below:

Here, bet need t5 from line 1, and here line 1 and line 2 are executed in the same cycle. Thus, we need to let sub-datapath 2 become no-op, and let it execute in the next cycle. Also, t5 is not done yet with executing, we need to stall one cycle for beq. This beq is not taken, and we predict it as not take, so we just jump to next instruction. Which is
20090005 addi t1 zero 0x5 t1 = 5
15AE0004 bne t5 t6 0x4 taken t5!=t6



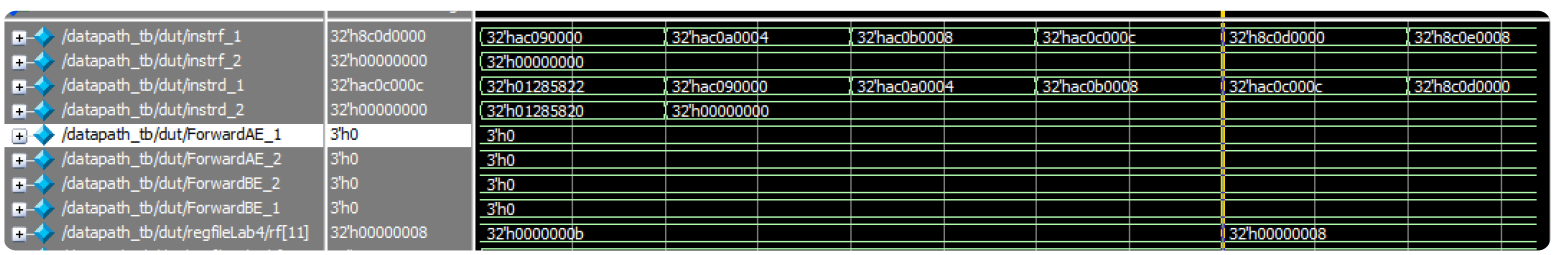
Also, we see forwardAE_1 become 1, which means the data is also forwarded.

Now we have bne in sub-datapath 2, so we need to move it to the next cycle as we can see

Bne is moved to the next cycle, and since it is taken and we predicts it wrong and the next instruction will be flush, and we will have the correct instruction after that cycle which is (01285822, 01285820)

Now, let's look at instruction:
01285822 sub t3 t1 t0 t3 = 2
01285820 add t3 t1 t0 t3 = 8

We are loading data to t3 at the same cycle for two different instruction.



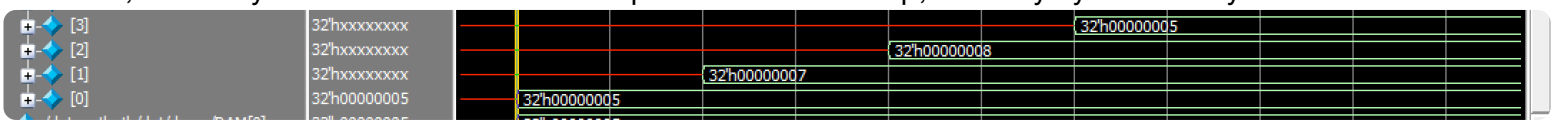
The waveform is shown below:

What our implementation is second instruction will finalize the value of t3. As we can see only 8 will be stored in the rf[11] which is t3.

Now, let's look at instruction:

AC090000 sw t1 0x0 zero Mem[0x0] = 5
AC0A0004 sw t2 0x4 zero Mem[0x4] = 7
AC0B0008 sw t3 0x8 zero Mem[0x8] = 8
AC0C000C sw t4 0xc zero Mem[0xc] = 5

This time, we always let instruction in sub-datapath 2 become no op, so every cycle will only execute one sw



instruction.

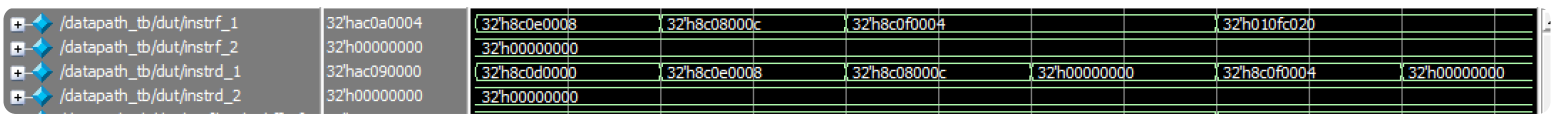
This is our data memory storing the data.

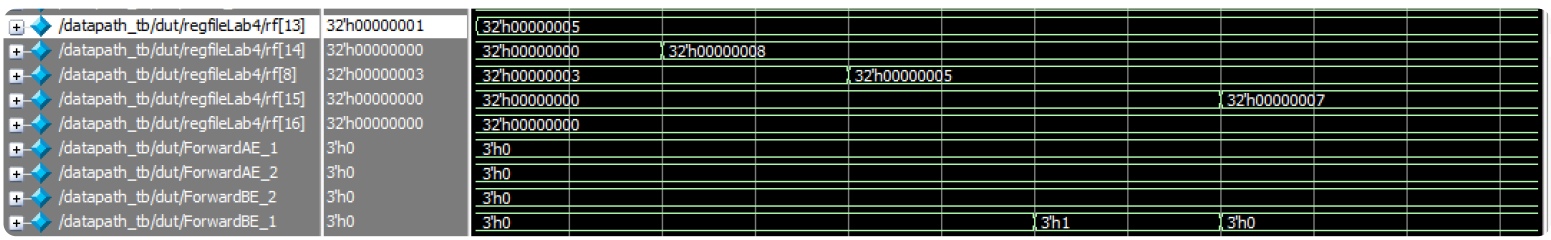
Now, let's look at instruction:

8C0D0000 lw t5 0x0 zero t1 = 5
8C0E0008 lw t6 0x8 zero t6 = 8
8C08000C lw t0 0xc zero t0 = 5
8C0F0004 lw t7 0x4 zero t7 = 7
010FC020 add t8 t0 t7 t8 = 12 = C

Last line will trigger the hazard of last two instructions. (t0 and t7 are needed)

The waveform is shown below:





8c08000c will check the following two instruction and find the instruction 010FC020 uses t0, we will see no op after this instruction. For 0C0F0004, it will check 010FC020 need t7, we will see no op after this instruction.
Instruction 010FC020 needs data from t7, we see the forwardBE_1 is triggered, we know it forwards successfully.

Hour spent: about 100hrs
The common mistake in verilog is implementing the result using the incorrect naming, which results us to miss the data.

Conclusion:
In this lab we learned how hazard can be handled in the dual issue superscalar pipeline.

The CPI for our program is 26/23 = 1.13 Which is still greater than 1. This is because, there are many hazards in one cycle that we have to stall. The program we wrote is mainly used to test our hazard, we will see a lot of hazard in this case. Notice, the final codes will be shown below.

Verilog Code section:

regfile.v

```
module regfile_especially_for_lab4(input clk,
            input Write_1, Write_2,
            input Reset,
            input [4:0] PR1_1, PR2_1, WR_1,
            input [31:0] WD_1,
            output [31:0] RD1_1, RD2_1,
            input [4:0] PR1_2, PR2_2, WR_2,
            input [31:0] WD_2,
            output [31:0] RD1_2, RD2_2);

  reg [31:0] rf[31:0];

  always @(posedge clk) begin

    if ( Write_1 && ~Reset) rf[WR_1] <= WD_1;
    if ( Write_2 && ~Reset) rf[WR_2] <= WD_2;

        if(Reset) begin

              rf[0] = 32'b0;
              rf[1] = 32'b0;
              rf[2] = 32'b0;
              rf[3] = 32'b0;
              rf[4] = 32'b0;
              rf[5] = 32'b0;
              rf[6] = 32'b0;
              rf[7] = 32'b0;
              rf[8] = 32'b0;
              rf[9] = 32'b0;
              rf[10] = 32'b0;
              rf[11] = 32'b0;
              rf[12] = 32'b0;
              rf[13] = 32'b0;
              rf[14] = 32'b0;
              rf[15] = 32'b0;
              rf[16] = 32'b0;
              rf[17] = 32'b0;
              rf[18] = 32'b0;
              rf[19] = 32'b0;
              rf[20] = 32'b0;
              rf[21] = 32'b0;
              rf[22] = 32'b0;
              rf[23] = 32'b0;
              rf[24] = 32'b0;
              rf[25] = 32'b0;
              rf[26] = 32'b0;
              rf[27] = 32'b0;
              rf[28] = 32'b0;
              rf[29] = 32'b11111100;
              rf[30] = 32'b0;
              rf[31] = 32'b0;


        end
```

```
    end

// The output port will be double
  assign RD1_1 = (PR1_1 == 0) ? 0 :
              (( Write_2 && ~Reset) && (WR_2 == PR1_1)) ? WD_2 :
              (( Write_1 && ~Reset) && (WR_1 == PR1_1)) ? WD_1 :
              rf[PR1_1];
  assign RD2_1 = (PR2_1 == 0) ? 0 :
              (( Write_2 && ~Reset) && (WR_2 == PR2_1)) ? WD_2 :
              (( Write_1 && ~Reset) && (WR_1 == PR2_1)) ? WD_1 :
              rf[PR2_1];

  assign RD1_2 = (PR1_2 == 0) ? 0 :
              (( Write_2 && ~Reset) && (WR_2 == PR1_2)) ? WD_2 :
              (( Write_1 && ~Reset) && (WR_1 == PR1_2)) ? WD_1 :
              rf[PR1_2];
  assign RD2_2 = (PR2_2 == 0) ? 0 :
              (( Write_2 && ~Reset) && (WR_2 == PR2_2)) ? WD_2 :
              (( Write_1 && ~Reset) && (WR_1 == PR2_2)) ? WD_1 :
              rf[PR2_2];
endmodule
```

Data_memory.v (this file includes both datameory and instr_memory)

```
module data_memory(input          clk, write,
          input   [31:0] address, write_data,
          output  [31:0] Read_data);

// **PUT YOUR CODE HERE**
  reg [31:0] RAM[63:0];

  assign Read_data = RAM[address[31:2]]; //word aligned

  always @(posedge clk) begin
    if (write) RAM[address[31:2]] <= write_data;
  end
endmodule




module inst_memory_2_output(input   [31:0]  address,
          output  [31:0] Read_data, Read_Next_Data);

  reg [31:0] RAM[63:0];

  initial
    begin
      $readmemh("memfile.dat",RAM); // initialize memory with test program. Change this with memfile2.dat for the
modified code
    end

  assign Read_data = RAM[address[31:2]]; // word aligned
  assign Read_Next_Data = RAM[address[31:2]+1]; // read next data
endmodule
```

hazard.v

```
module hazard(// branch related
        input [4:0] WriteRegD_1, WriteRegD_2,
        input RegWriteD_1, RegWriteD_2,

        input reset,
        //dp1
        output [1:0] ForwardAD_1, ForwardBD_1,
        output reg [2:0] ForwardAE_1, ForwardBE_1,
        input [4:0] RsD_1, RtD_1,
        input [4:0] RsE_1, RtE_1,
        input RegWriteE_1, MemtoRegE_1,
        input RegWriteM_1, MemtoRegM_1,
        input [4:0] WriteRegM_1, WriteRegW_1,
        input RegWriteW_1,

        //dp2
        output [1:0] ForwardAD_2, ForwardBD_2,
        output reg [2:0] ForwardAE_2, ForwardBE_2,
        input [4:0] RsD_2, RtD_2,
        input [4:0] RsE_2, RtE_2,
        input RegWriteE_2, MemtoRegE_2,
        input RegWriteM_2, MemtoRegM_2,
        input [4:0] WriteRegM_2, WriteRegW_2,
        input RegWriteW_2,

        //fetch stage early noop predict
        input RegWriteF_1, RegWriteF_2,
        input RegDstF_1, RegDstF_2,
        input BranchF_1, BranchF_2, // BranchF2 may not be used/Branch could not be in slot2
        input MemEnableF_1, MemEnableF_2,
        input [4:0] RsF_1, RsF_2,
        input [4:0] RtF_1, RtF_2,
        input [4:0] RdF_1, RdF_2,
        input JumpF_1, JumpF_2,
        input IsLoadWordD_1, IsLoadWordD_2,
        output NoopSlotF2,


        // stall and flush
        output stallF,
        output flushD,
        output flushE,
        // branch related
        input BranchIsNotCorrectE_1
        //input BranchIsNotCorrectE_2 // NOT USED


        );

        wire lwstall;
        reg lwstall_helper;
        assign lwstall = lwstall_helper;
        always @(*) begin
                lwstall_helper <= 0;
                if(reset) lwstall_helper <= 0;
```

```
        else if(IsLoadWordD_1 && ((RsF_1 == RtD_1) || (RsF_2 == RtD_1) || (RtF_1 == RtD_1) || (RtF_2 ==
RtD_1))) lwstall_helper <= 1;
        else if(IsLoadWordD_2 && ((RsF_1 == RtD_2) || (RsF_2 == RtD_2) || (RtF_1 == RtD_2) || (RtF_2 ==
RtD_2))) lwstall_helper <= 1;

    end

    wire branchStall;
    assign branchStall = BranchIsNotCorrectE_1;


    reg branchStall_original;// handle branch that use last instructions
    always @(*) begin
        branchStall_original <= 0;

        if(BranchF_1) begin
            if((RsF_1==WriteRegD_1) & RegWriteD_1) branchStall_original <= 1;
            else if((RtF_1==WriteRegD_1) & RegWriteD_1) branchStall_original <= 1;
            else if((RsF_1==WriteRegD_2) & RegWriteD_2) branchStall_original <= 1;
            else if((RtF_1==WriteRegD_2) & RegWriteD_2) branchStall_original <= 1;
        end
    end

    assign stallF = lwstall | branchStall | branchStall_original;
    assign flushD = lwstall | branchStall | branchStall_original;
    assign flushE = branchStall;



    wire [4:0] WriteRegF_1, WriteRegF_2;
    assign WriteRegF_1 = RegDstF_1 ? RdF_1 : RtF_1;
    assign WriteRegF_2 = RegDstF_2 ? RdF_2 : RtF_2;

    reg noop_helper;
    assign NoopSlotF2 = noop_helper;
    always @(*) begin
        noop_helper <= 0;
        if(reset==1) noop_helper <= 0;
        else if(JumpF_1==1) begin noop_helper <= 1; $display("noop case1 %0t", $time);end
        else if(JumpF_2==1)  begin noop_helper <= 1; $display("noop case2 %0t", $time); end
        else if(BranchF_1==1) begin noop_helper <= 1; $display("noop case3 %0t", $time);end
        else if(BranchF_2==1) begin noop_helper <= 1; $display("noop case4 %0t", $time);end
        else if(MemEnableF_1 & MemEnableF_2) begin noop_helper <= 1; $display("noop case5 %0t",
$time);end
        else if(RegWriteF_1) begin
            if(RegDstF_2) begin

                if((RtF_2 == WriteRegF_1) | (RsF_2 == WriteRegF_1))begin noop_helper <= 1;
$display("noop case6 %0t", $time);end
            end
            else begin
                if(RsF_2 == WriteRegF_1) begin noop_helper <= 1; $display("noop case7 %0t",
$time);end
            end
        end
        //else begin noop_helper <= 0; $display("noop case8 %0t", $time);end
    end
```

```
reg [1:0] forwardAD_1, forwardBD_1;
reg [1:0] forwardAD_2, forwardBD_2;

assign ForwardAD_1 = forwardAD_1;
assign ForwardAD_2 = forwardAD_2;
assign ForwardBD_1 = forwardBD_1;
assign ForwardBD_2 = forwardBD_2;
//Decode stage forwarding
always @(*) begin
	forwardAD_1 = 2'b00;
	if(RsD_1 != 0 & (RsD_1 == WriteRegM_1) & RegWriteM_1)  forwardAD_1 = 2'b01;
	else if(RsD_1 != 0 & (RsD_1 == WriteRegM_2) & RegWriteM_2)  forwardAD_1 = 2'b10;

	forwardAD_2 = 2'b00;
	if(RsD_2 != 0 & (RsD_2 == WriteRegM_2) & RegWriteM_2)  forwardAD_2 = 2'b01;
	else if(RsD_2 != 0 & (RsD_2 == WriteRegM_1) & RegWriteM_1)  forwardAD_2 = 2'b10;

	forwardBD_1 = 2'b00;
	if(RtD_1 != 0 & (RtD_1 == WriteRegM_1) & RegWriteM_1) forwardBD_1 = 2'b01;
	else if (RtD_1 != 0 & (RtD_1 == WriteRegM_2) & RegWriteM_2) forwardBD_1 = 2'b10;

	forwardBD_2 = 2'b00;
	if((RtD_2 != 0) & (RtD_2 == WriteRegM_2) & RegWriteM_2) forwardBD_2 = 2'b01;
	else if (RtD_2 != 0 & (RtD_2 == WriteRegM_1) & RegWriteM_1) forwardBD_2 = 2'b10;

end


// execute stage forwarding
always @* begin
	// for dp1
	ForwardAE_1 <= 3'b000;
	ForwardBE_1 <= 3'b000;
	if (RsE_1 != 0) begin
		if ((RsE_1 == WriteRegM_1) & RegWriteM_1) begin
			ForwardAE_1 <= 3'b010;
		end
		else if ((RsE_1 == WriteRegW_1) & RegWriteW_1) begin
			ForwardAE_1 <= 3'b001;
		end
	end

	if (RsE_1 != 0) begin
		if ((RsE_1 == WriteRegM_2) & RegWriteM_2) begin
			ForwardAE_1 <= 3'b100;
		end
		else if ((RsE_1 == WriteRegW_2) & RegWriteW_2) begin
			ForwardAE_1 <= 3'b011;
		end
	end

	if (RtE_1 != 0) begin
		if ((RtE_1 == WriteRegM_1) & RegWriteM_1) begin
			ForwardBE_1 <= 3'b010;
		end
		else if ((RtE_1 == WriteRegW_1) & RegWriteW_1) begin
			ForwardBE_1 <= 3'b001;
		end
	end
```

```
        if (RtE_1 != 0) begin
            if ((RtE_1 == WriteRegM_2) & RegWriteM_2) begin
                ForwardBE_1 <= 3'b100;
            end
            else if ((RtE_1 == WriteRegW_2) & RegWriteW_2) begin
                ForwardBE_1 <= 3'b011;
            end
        end
        // for dp2
        ForwardAE_2 <= 3'b000;
        ForwardBE_2 <= 3'b000;
        if (RsE_2 != 0) begin
            if ((RsE_2 == WriteRegM_2) & RegWriteM_2) begin
                ForwardAE_2 <= 3'b010;
            end
            else if ((RsE_2 == WriteRegW_2) & RegWriteW_2) begin
                ForwardAE_2 <= 3'b001;
            end
        end

        if (RsE_2 != 0) begin
            if ((RsE_2 == WriteRegM_1) & RegWriteM_1) begin
                ForwardAE_2 <= 3'b100;
            end
            else if ((RsE_2 == WriteRegW_1) & RegWriteW_1) begin
                ForwardAE_2 <= 3'b011;
            end
        end

        if (RtE_2 != 0) begin
            if ((RtE_2 == WriteRegM_2) & RegWriteM_2) begin
                ForwardBE_2 <= 3'b010;
            end
            else if ((RtE_2 == WriteRegW_2) & RegWriteW_2) begin
                ForwardBE_2 <= 3'b001;
            end
        end

        if (RtE_2 != 0) begin
            if ((RtE_2 == WriteRegM_1) & RegWriteM_1) begin
                ForwardBE_2 <= 3'b100;
            end
            else if ((RtE_2 == WriteRegW_1) & RegWriteW_1) begin
                ForwardBE_2 <= 3'b011;
            end
        end
    end


endmodule
```

subdatapath.v

```
module subdatapath(
	// global clk, reset
	input clk,
	input reset,
	// decode stage
	input [31:0] instrd,
	// regfile
	output [4:0] a1, // give address of regfile, get rd1
	output [4:0] a2, // give address of regfile, get rd2
	input [31:0] rd1, rd2, // get data from regfile
	output [4:0] a3, // write address
	output [31:0] wd3, // write data 3
	output we3, // regfile write enable

	// datamem
	output [31:0] memadr, // memory address && forward use
	output [31:0] memwd, // memory write data
	input [31:0] memoryrd, // memory read data
	output memenable, // memory enable
	output memwritem,

	// branch
	// stub, implement later
	input [31:0] pcplus4d,
	input [31:0] memadr_another, // memadr from another slot
	input [1:0] forwardad,
	input [1:0] forwardbd,
	input branchPredictedTakenD,
	output BranchIsTakenE,

	output branchIsNotCorrectE,
	output [31:0] PCBranchE,

	input [2:0] forwardae, forwardbe,
	output [31:0] resultw,
	input [31:0] resultw_another,

	// jump related
	output jump,
	output [31:0] jumpadr,
	// hazard
	input flushe,
	input stalle,

	input flushm,
	input stallm,

	input flushw,
	input stallw,
	output isLoadWordD,
	//
	output [4:0] rsd, rtd, rse, rte,
	//hazard needs them
	output regwritee,memtorege,
	output regwritem, memtoregm,
```

```
        output [4:0] WriteRegM, WriteRegW,
        output regwritew,
        output [4:0] WriteRegD,
        output RegWriteD
);
        // assign regs
        assign a1 = instrd[25:21];
        assign a2 = instrd[20:16];
        assign a3 = WriteRegW;
        assign isLoadWordD = (instrd[31:26] == 6'b100011);

        // branch related
        wire branchd; // NOT USED
        wire [1:0] PCSrcD; // Only Used Internally
        wire branchistakend; // Only use in decode
        assign {jump, branchistakend} = PCSrcD;


        // controller signal decode
        wire se_ze; // only in decode
        wire memwrited, memwritee;
        wire [1:0] outseld, outsele, outselm; // this is used to select from Hi/Lo/aluoutm/Lui
        wire regwrited;
        wire alusrcbd, alusrcbe;
        wire regdstd, regdste;
        wire memtoregd,memtoregw;
        wire [3:0] aluopd, aluope;
        wire memenabled,memenablee,memenablem; // this is used for cache, MAY NOT USED
        wire [4:0] rdd; // Rs & Rt May need to be output
        assign rsd = instrd[25:21];
        assign rtd = instrd[20:16];
        assign rdd = instrd[15:11];

        // multiplier related: MAY NOT BE IMPLEMENTED
        wire Start_mult;
        wire Mult_sign;


        // decode stage
        wire [31:0] branch_check1, branch_check2;
        mux3 #(32) branch_check_mux_1(rd1, memadr, memadr_another, forwardad, branch_check1);
        mux3 #(32) branch_check_mux_2(rd2, memadr, memadr_another, forwardbd, branch_check2);
        wire Eq_ne; // eq == 1, ne ==0
        wire EqualD;
        assign EqualD = Eq_ne ? (branch_check1 == branch_check2) : (branch_check1 != branch_check2);
        wire [31:0] immextd, immexte;
        extend ext(instrd[15:0], se_ze, immextd); // extent mudoule output as SignImmD labled in the diagram
        wire branchIsNotCorrectD;
        assign branchIsNotCorrectD = (branchPredictedTakenD != branchistakend); // check if branch is taken in
decode stage
        wire [31:0] PCBranchD;
        wire [31:0] immextdShift2;
        assign immextdShift2 = immextd << 2;
        assign PCBranchD = branchistakend ? (immextdShift2 + pcplus4d) : pcplus4d;


        // decode to execute
        wire [31:0] rd1e, rd2e;
        wire [4:0] rde; // Rs & Rt May need to be output
```

```
    resetclearenablereg #(157)
            d2e(clk, reset, flushe,(~stalle),
             {rd1, rd2, immextd, branchIsNotCorrectD, branchistakend, rsd, rtd, rdd, PCBranchD, memwrited,
outseld, regwrited, alusrcbd, regdstd, memtoregd, aluopd, memenabled},
             {rd1e,rd2e,immexte, branchIsNotCorrectE, BranchIsTakenE, rse, rte, rde, PCBranchE, memwritee,
outsele, regwritee, alusrcbe, regdste, memtorege, aluope, memenablee});

    // execute stage
    // PCBranchE & branchIsNotCorrectE does not need any operation
    wire [31:0] SrcAE, SrcBE_BeforeMux, SrcBE, WriteDataM; // SrcBE_BeforeMux is same to WriteDataE
    ForwardSpecialMux5 forwardMuxAE(forwardae, rd1e, resultw, memadr, resultw_another, memadr_another,
SrcAE);
    ForwardSpecialMux5 forwardMuxBE(forwardbe, rd2e, resultw, memadr, resultw_another, memadr_another,
SrcBE_BeforeMux);

    assign SrcBE = alusrcbe ? immexte : SrcBE_BeforeMux;

    wire [31:0] ALUOutE, ALUOutM;
    ALU ourOwnALU(SrcAE, SrcBE, aluope, ALUOutE);
    wire [4:0] WriteRegE; // May need to be output
    assign WriteRegE = regdste ? rde : rte;

    wire [31:0] luiImmE, luiImmM;
    assign luiImmE = {immexte[15:0], 16'b0};

    resetclearenablereg #(107)
            e2m(clk, reset, flushm,(~stallm),
             {ALUOutE, SrcBE_BeforeMux, WriteRegE, luiImmE, memwritee, outsele, regwritee, memtorege,
memenablee},
             {ALUOutM, WriteDataM     , WriteRegM, luiImmM, memwritem, outselm, regwritem, memtoregm,
memenablem});
    assign memenable = memenablem;
    // memory stage
    mux4 #(32) memadrSelect(32'b0, 32'b0, ALUOutM, luiImmM, outselm, memadr); // assign memadr
    assign memwd = WriteDataM;

    wire [31:0] ReadDataW, ALUOutW;
    resetclearenablereg #(71)
            m2w(clk, reset, flushw,(~stallw),
             {memoryrd , memadr , WriteRegM, regwritem, memtoregm},
             {ReadDataW, ALUOutW, WriteRegW, regwritew, memtoregw});
    // write back
    assign resultw = memtoregw ? ReadDataW : ALUOutW;
    assign wd3 = resultw;
    assign we3 = regwritew;


    // controller
    Controller supercontroller(EqualD, instrd[31:26], instrd[5:0], reset, memwrited,
    outseld, regwrited, alusrcbd, se_ze, regdstd,
    Start_mult, Mult_sign, memtoregd, Eq_ne, branchd, PCSrcD, aluopd,
    memenabled);


    assign WriteRegD = regdstd ? rdd : rtd;
    assign RegWriteD = regwrited;

endmodule
```

FriendlyMIPS.v

```
module MipsBob (input clk, reset);
	wire [15:0] immF;


	// hazard related
	wire noop;
	wire RegWriteD_1, RegWriteD_2;
	wire [4:0] WriteRegD_1, WriteRegD_2;


	// fetch stage
	wire [31:0] PCNext, PCF, PCPlus4F, PCPlus8F, PCBranchE, jumpAdrD;
	assign PCPlus4F = PCF+4;
	assign PCPlus8F = PCF+8;
	//if jump, do jump; if branchis not Correct, do PCBranchE, if PredictF: do predicted pc, if noop, do pc+4,
default + 8
	wire jumpD;
	wire branchIsNotCorrectE;
	wire predictF;
	wire [31:0] PCIfTaken;
	assign PCNext = jumpD ? jumpAdrD : ((branchIsNotCorrectE) ? PCBranchE : (predictF ? PCIfTaken : (noop ?
PCPlus4F : PCPlus8F)));

	// pcreg
	wire stallF;
	flopr #(32) pcreg(clk, reset, (~stallF), PCNext, PCF);
	wire [31:0] PCD, PCE;
	//branch predictor
	wire [1:0] branchHistoryF,branchHistoryE;
	PCBranchBufForStep3 branchPredictor(clk, reset, branchF_1, branchE_1, (~branchIsNotCorrectE),PCF,
PCE,immF, branchHistoryE, BranchIsTakenE_1, predictF, PCIfTaken, branchHistoryF);

	// instrmen

	wire [31:0] instrf_1, instrf_2_before_mux, instrf_2;
	wire [31:0] instrd_1, instrd_2;
	inst_memory_2_output instr_mem_out (PCF, instrf_1, instrf_2_before_mux);
	assign instrf_2 = noop ? 32'b0 : instrf_2_before_mux;
	assign immF = instrd_1[15:0];


	// otherthing to do in fetch stage: e.g. Hazard
	wire RegWriteF_1, RegWriteF_2;
	wire RegDstF_1, RegDstF_2;
	wire BranchF_1, BranchF_2; // BranchF2 may not be used/Branch could not be in slot2
	wire MemEnableF_1, MemEnableF_2;
	wire JumpF_1,JumpF_2;
	MainDec fetch_decoder_1(instrf_1[31:26], reset, RegWriteF_1, RegDstF_1, BranchF_1, MemEnableF_1,
JumpF_1);
	MainDec fetch_decoder_2(instrf_2_before_mux[31:26], reset, RegWriteF_2, RegDstF_2, BranchF_2,
MemEnableF_2, JumpF_2);

	// f2d register
```

```
    wire predictD;

    wire [1:0] branchHistoryD;
    wire flushD;
    wire flushe,stalle, flushm,stallm,flushw,stallw;


    assign stalle = 0;
    assign flushm = 0;
    assign stallm = 0;
    assign flushw = 0;
    assign stallw = 0;
    wire stallD;
    assign stallD = 0;
    wire [31:0] PCPlus4D;
    resetclearenablereg #(131) f2d(clk, reset, flushD, (~stallD),
            {instrf_1, instrf_2, PCPlus4F, PCF, predictF, branchHistoryF},
            {instrd_1, instrd_2, PCPlus4D, PCD, predictD, branchHistoryD});
    resetclearenablereg #(34) d2e_for_PC_only(clk, reset, flushe,(~stalle),
                                        {PCD, branchHistoryD},
                                        {PCE, branchHistoryE});

    // datapath 1 prototal
    wire [4:0] a1_1, a2_1, a3_1;
    wire [4:0] a1_2, a2_2, a3_2;
    wire [31:0] rd1_1, rd2_1, wd3_1;
    wire [31:0] rd1_2, rd2_2, wd3_2;
    wire we3_1;
    wire we3_2;
    wire [31:0] memadr_1;
    wire [31:0] memadr_2;
    wire [31:0] memwd_1;
    wire [31:0] memwd_2;
    wire [31:0] memoryrd_1;
    wire [31:0] memoryrd_2;
    wire memenable_1;
    wire memenable_2;
    wire [31:0] resultw_1;
    wire [31:0] resultw_2;
    wire IsLoadWordD_1, IsLoadWordD_2;
    // hazard
    wire [1:0] ForwardAD_1, ForwardBD_1;
    wire [2:0] ForwardAE_1, ForwardBE_1;
    wire [1:0] ForwardAD_2, ForwardBD_2;
    wire [2:0] ForwardAE_2, ForwardBE_2;

    // shared hazard command

    wire [4:0] RsD_1, RtD_1, RsD_2, RtD_2, RsE_1, RtE_1, RsE_2, RtE_2;
    wire regwritee_1,regwritee_2, memtorege_1, memtorege_2, RegWriteM_1, MemtoRegM_1, RegWriteM_2,
MemtoRegM_2;
    wire [4:0] WriteRegM_1, WriteRegW_1, WriteRegM_2, WriteRegW_2;
    wire RegWriteW_1, RegWriteW_2, memwritem_1, memwritem_2;

    subdatapath sub_data_path_1(clk,reset,instrd_1,a1_1, a2_1, rd1_1, rd2_1, a3_1, wd3_1, we3_1, memadr_1,
memwd_1, memoryrd_1, memenable_1, memwritem_1, // memory enable
                        PCPlus4D,memadr_2, ForwardAD_1, ForwardBD_1, predictD, BranchIsTakenE_1,
```

```
branchIsNotCorrectE, PCBranchE,
                        ForwardAE_1, ForwardBE_1,resultw_1,resultw_2, jumpD, jumpAdrD, flushe,stalle,
flushm,stallm,flushw,stallw,IsLoadWordD_1,
                        RsD_1, RtD_1, RsE_1, RtE_1,
                        regwritee_1,memtorege_1,
                        RegWriteM_1, MemtoRegM_1,
                        WriteRegM_1, WriteRegW_1,
                        RegWriteW_1, WriteRegD_1, RegWriteD_1);

    //useless trash
    wire [31:0] PCPlus4D_2, PCBranchE_2, jumpAdrD_2;
    wire predictD_2, branchIsNotCorrectE_2, BranchIsTakenE_2;
    wire jumpD_2;
    subdatapath sub_data_path_2(clk,reset,instrd_2, a1_2, a2_2, rd1_2, rd2_2, a3_2, wd3_2, we3_2, memadr_2,
memwd_2, memoryrd_2, memenable_2, memwritem_2,// memory enable
                        PCPlus4D_2,memadr_1, ForwardAD_2, ForwardBD_2, predictD_2, BranchIsTakenE_2,
branchIsNotCorrectE_2, PCBranchE_2,
                        ForwardAE_2, ForwardBE_2,resultw_2,resultw_1, jumpD_2, jumpAdrD_2, flushe,stalle,
flushm,stallm,flushw,stallw,IsLoadWordD_2,
                        RsD_2, RtD_2, RsE_2, RtE_2,
                        regwritee_2,memtorege_2,
                        RegWriteM_2, MemtoRegM_2,
                        WriteRegM_2, WriteRegW_2,
                        RegWriteW_2, WriteRegD_2, RegWriteD_2);


    // regfile
    regfile_especially_for_lab4 regfileLab4(clk, we3_1, we3_2, reset, a1_1, a2_1, a3_1, wd3_1, rd1_1, rd2_1,
                                            a1_2, a2_2, a3_2, wd3_2, rd1_2, rd2_2);

    // memory
    wire dmem_write;
    wire [31:0] dmem_address, dmem_write_data, dememd_read_data;
    data_memory dmem(clk, dmem_write, dmem_address, dmem_write_data,dememd_read_data);

    assign dmem_write = memwritem_1 | memwritem_2;
    assign dmem_address =  memenable_1 ? memadr_1 : memadr_2;
    assign dmem_write_data = memenable_1 ? memwd_1 : memwd_2;
    assign memoryrd_1 = dememd_read_data;
    assign memoryrd_2 = dememd_read_data;



    hazard     hazardunit
        (
        WriteRegD_1, WriteRegD_2,
        RegWriteD_1, RegWriteD_2,
        reset,
        //dp1
        ForwardAD_1, ForwardBD_1,
        ForwardAE_1, ForwardBE_1,
        RsD_1, RtD_1,
        RsE_1, RtE_1,
        RegWriteE_1, MemtoRegE_1,
        RegWriteM_1, MemtoRegM_1,
        WriteRegM_1, WriteRegW_1,
        RegWriteW_1,
```

```
        //dp2
        ForwardAD_2, ForwardBD_2,
        ForwardAE_2, ForwardBE_2,
        RsD_2, RtD_2,
        RsE_2, RtE_2,
        RegWriteE_2, MemtoRegE_2,
        RegWriteM_2, MemtoRegM_2,
        WriteRegM_2, WriteRegW_2,
        RegWriteW_2,

        //fetch stage early noop predict
        RegWriteF_1, RegWriteF_2,
        RegDstF_1, RegDstF_2,
        BranchF_1, BranchF_2, // BranchF2 may not be used/Branch could not be in slot2
        MemEnableF_1, MemEnableF_2,
        instrf_1[25:21], instrf_2_before_mux[25:21],
        instrf_1[20:16], instrf_2_before_mux[20:16],
        instrf_1[15:11], instrf_2_before_mux[15:11],
        JumpF_1, JumpF_2,
        IsLoadWordD_1, IsLoadWordD_2,
        noop,


        // stall and flush
        stallF,
        flushD,
        flushe,
        // branch related
        branchIsNotCorrectE
        //input BranchIsNotCorrectE_2 // NOT USED


        );



endmodule
```