计算机系统结构实验

实验 6 报告

类MIPS多周期流水线处理器的设计与实现

Log Creative

2021 年 6 月 27 日

目录

1 实验目的

1. 理解CPU Pipeline，了解流水线冒险(hazard)及相关性，设计基础流水线CPU
2. 增加Forwarding机制解决数据竞争，减少因数据竞争带来的流水线停顿延时，提高流水线处理器性能
3. 设计支持Stall的流水线CPU。通过检测竞争并插入停顿（Stall）机制解决数据冒险、控制竞争和结构冒险
4. 通过predict-not-taken或延时转移策略解决控制冒险/竞争，减少控制竞争带来的流水线停顿延时，进一步提高处理器性能
5. ~~将CPU支持的指令数量从16条扩充为31条，使处理器功能更加丰富（选做）~~

2 原理实现

2.1 基础流水线

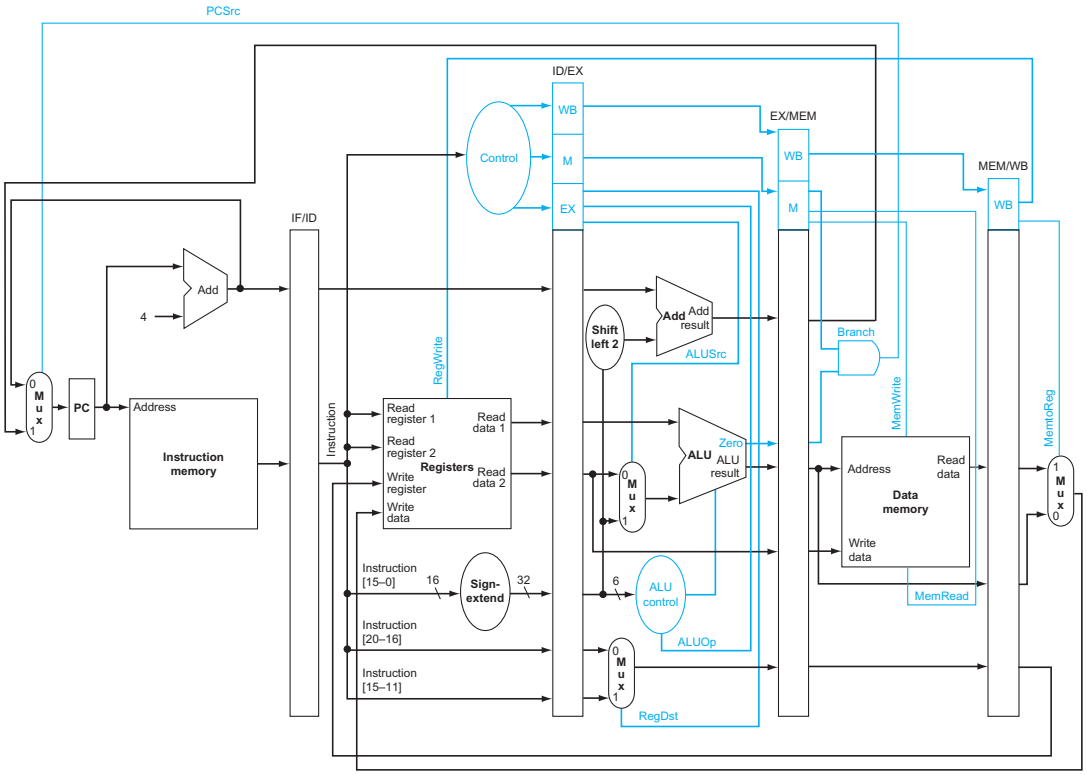


FIGURE 4.51 The pipelined datapath of Figure 4.46, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.

通过添加段寄存器，以实现每个阶段的单独运行，并实现流水化。该图是基于 9 条指令的基础流水线设计。基于 16 条指令可以依据实验五进行改造。

2.2 前向转发机制

为了消除数据相关性（比如上一条指令的结果将会作为这一条指令运行），处理器需要引入前向转发机制（Forwarding）。

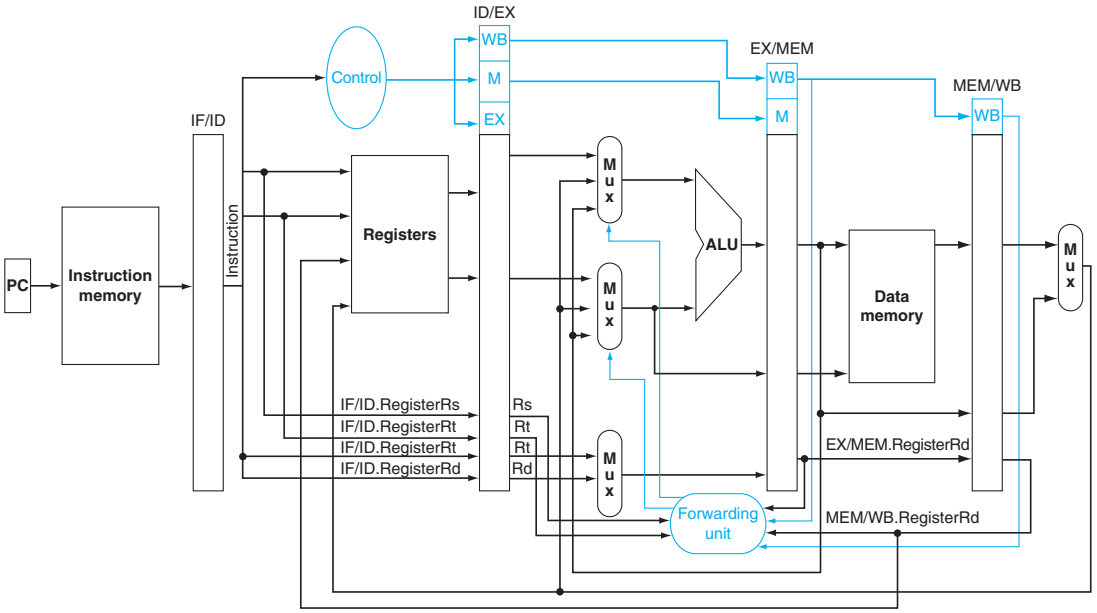


FIGURE 4.56 The datapath modified to resolve hazards via forwarding. Compared with the datapath in Figure 4.51, the additions are the multiplexors to the inputs to the ALU. This figure is a more stylized drawing, however, leaving out details from the full datapath, such as the branch hardware and the sign extension hardware.

表 1: 前向转发机制的控制信号

多选器控制	源	解释
ForwardA=00	ID	第一个 ALU 操作数来自寄存器堆
ForwardA=10	EX	第一个 ALU 操作数由上一个 ALU 运算结果转发获得
ForwardA=01	MEM	第一个 ALU 操作数从数据存储器或者前面的 ALU 运算结果中转发获得
ForwardB=00	ID	第二个 ALU 操作数来自寄存器堆
ForwardB=10	EX	第二个 ALU 操作数由上一个 ALU 运算结果转发获得
ForwardB=01	MEM	第二个 ALU 操作数由数据存储器或者前面的 ALU 结果转发获得

对于两种 EX 冒险和 MEM 冒险两种模式，根据表 1 采用以下的判定方法以及输出方式。

```
wire [1:0] ForwardA = ((EX_REG_WRITE &
    (EX_WRITE_REG != 0) &
    (EX_WRITE_REG == ID_INST[25:21])) ?
        2'b10 :
        ((MEM_REG_WRITE & (MEM_WRITE_REG != 0)
        & (MEM_WRITE_REG == ID_INST[25:21])) ?
            2'b01 :
            2'b00));

wire [1:0] ForwardB = ((EX_REG_WRITE &
    (EX_WRITE_REG != 0) &
    (EX_WRITE_REG == ID_INST[20:16])) ?
        2'b10 :
        ((MEM_REG_WRITE & (MEM_WRITE_REG != 0)
        & (MEM_WRITE_REG == ID_INST[20:16])) ?
            2'b01 :
            2'b00));

wire [31:0] ForwardA_RES =
        ForwardA == 2'b00 ? ID_READ_DATA1 :
            (ForwardA == 2'b10 ? EX_ALU_RES :
                WRITE_DATA_WB);

wire [31:0] ForwardB_RES =
        ForwardB == 2'b00 ? ID_READ_DATA2 :
            (ForwardB == 2'b10 ? EX_ALU_RES :
                WRITE_DATA_WB);
```

这样，EX 冒险（10）就可以由 ALU 转发，而 MEM 冒险（01）就可以由写回数据转发。而对于接收者也要做一定的适配，并且为了匹配立即数指令，需要在转发结果后再加选路器。

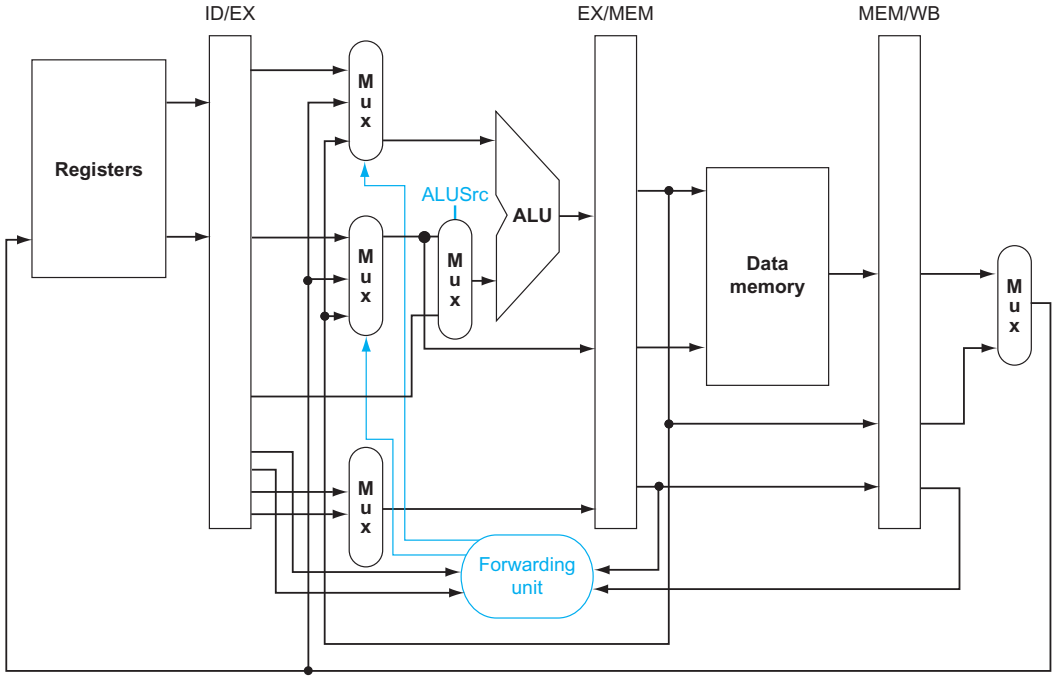


FIGURE 4.57 A close-up of the datapath in Figure 4.54 shows a 2:1 multiplexor, which has been added to select the signed immediate as an ALU input.

```
ALU alu(
    .input1(SHAMT ? ID_INST[10:6] : ForwardA_RES), // ForwardA
    .input2(ID_ALU_SRC ? ID_OPRAND : ForwardB_RES), // ForwardB
    .aluCtr(ALU_CTR),
    .zero(ZERO_EX),
    .aluRes(ALU_RES_EX)
);

dataMemory DataMemory(
    .Clk(clk),
    .address(EX_ALU_RES),
    .writeData(ForwardB_RES), // ForwardB
    .memWrite(EX_MEM_WRITE),
    .memRead(EX_MEM_READ),
    .readData(READ_DATA_MEM)
);
```

值得注意的是，存储器的写入数据使用了前向传递B的结果。

并且在 WB 级不会发生冒险的前提是

> 假设在 ID 级指令读取的存储器与 WB 级指令写入的寄存器是同一个寄存器时，就由寄存器堆提供正确的结果。

因此，寄存器堆的输出不能依赖于输入地址的变化，一旦数据出现了变化，就需要立刻输出更新值。这可以采用 `wire` 类型直接赋值实现，这样一旦时钟下沿写入，就会被立刻更新为新值，从而影响下一时钟上沿的赋值。

Listing 1: Registers.v

```
module Registers(
    input [25:21] readReg1,
    input [20:16] readReg2,
    input [4:0] writeReg,
    input [31:0] writeData,
    input regWrite,
    output [31:0] readData1,
    output [31:0] readData2,
    input clk,
    input reset
    );

    reg [31:0] RegFile [31:0];

    // reg [31:0] ReadData1;
    // reg [31:0] ReadData2;
    // always @(readReg1 or readReg2) begin
    //     ReadData1 = RegFile[readReg1];
    //     ReadData2 = RegFile[readReg2];
    // end
    // assign readData1 = ReadData1;
    // assign readData2 = ReadData2;
    assign readData1 = RegFile[readReg1];
    assign readData2 = RegFile[readReg2];

    integer i;
    always @(negedge clk or reset) begin
        if(reset) begin
            for(i = 0; i < 32; i = i + 1)
                RegFile[i] = 0;
        end
        else begin
            if(regWrite)
                RegFile[writeReg] = writeData;
        end
    end
endmodule
```

2.3 停顿机制

当寄存器尚未获得值且紧随的运算需要该寄存器的后值时，仅仅前向转发是不足的。这时候就要引入停顿机制。

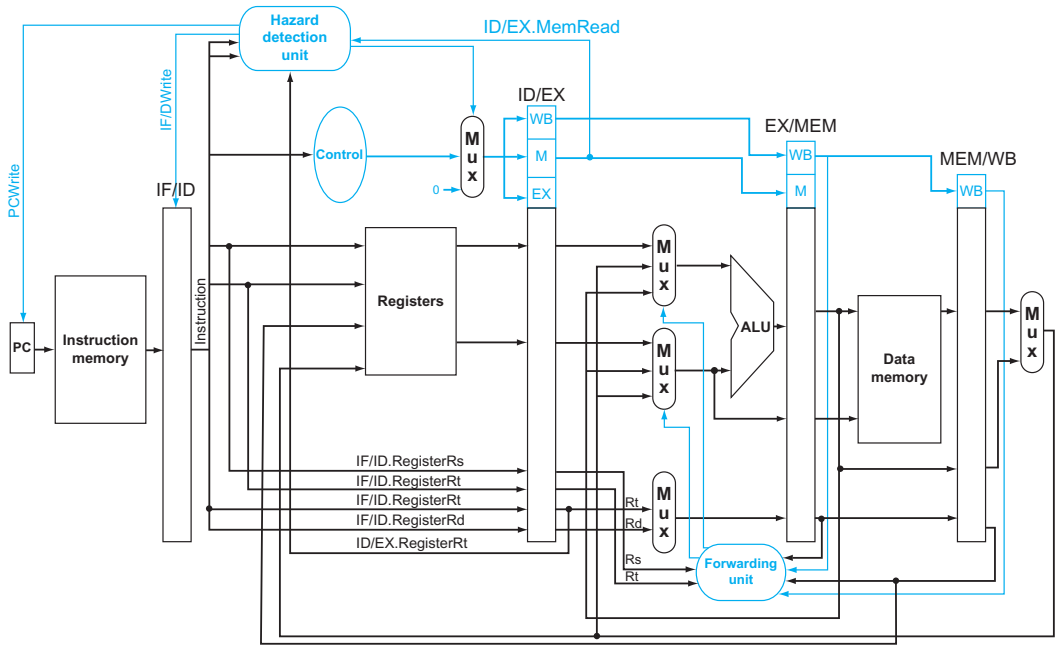


FIGURE 4.60 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit. Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware requirements.

用来解决 lw - use 冒险，就需要考察读取的存储器是不是使用的存储器。

```
wire stalling = (ID_MEM_READ &&
    ((ID_INST[20:16] == IF_INST[25:21])
    || (ID_INST[20:16] == IF_INST[20:16]))) ?
        1 : 0;
```

一旦停顿机制被激活，就会向流水线中添加一个气泡以停顿，方法就是IF 值被维持和 ID 控制信号清除（这一步预测不发生共用）。

```
1   if (stalling || (ID_BRANCH && BEQ)) begin
2       // nop
3       ID_REG_DST <= 0;
4       ID_JUMP <= 0;
5       ID_BRANCH <= 0;
6       ID_MEM_READ <= 0;
7       ID_MEM_TO_REG <= 0;
8       ID_MEM_WRITE <= 0;
9       ID_ALU_OP <= 0;
10      ID_ALU_SRC <= 0;
```

```
        ID_IMM <= 0;
        ID_JAL <= 0;
        ID_REG_WRITE <= 0;
    else //...

    // IF
    if(stalling == 0) begin
        PC <= PC + 4;
        IF_PC <= PC;          // Has already been PC + 4
        IF_INST <= INST_IF;
    end // stalling means no change
```

2.4 预测不发生机制

最后来处理分支情况。如果采用传统的方式，发生分支时，需要等待分支在 EX 段才可以判断，此前需要一直阻塞（3个周期）。这样做实在太慢，不利于流水线的运转。

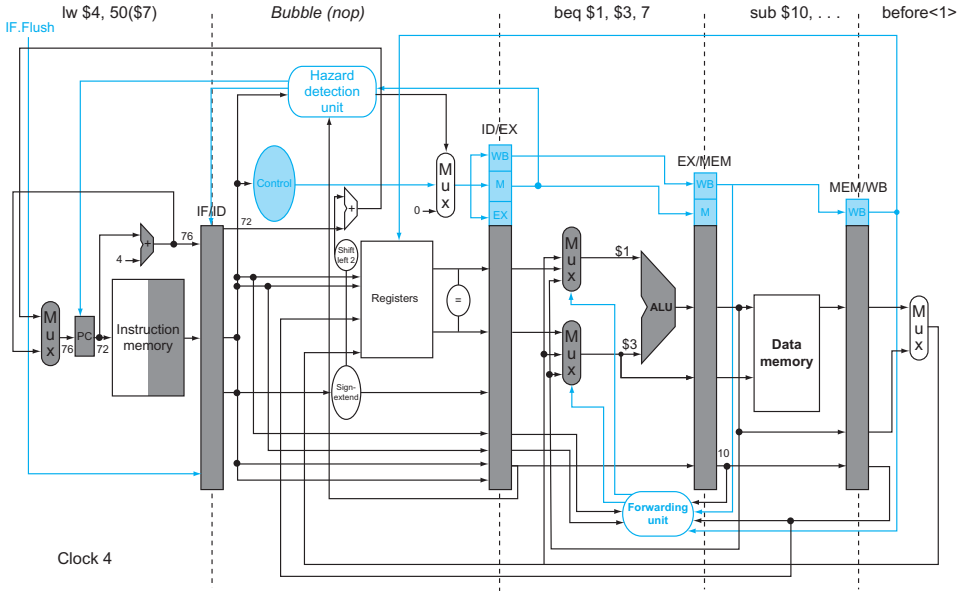


FIGURE 4.62 The ID stage of clock cycle 3 determines that a branch must be taken, so it selects 72 as the next PC address and zeros the instruction fetched for the next clock cycle. Clock cycle 4 shows the instruction at location 72 being fetched and the single bubble or `nop` instruction in the pipeline as a result of the taken branch. (Since the `nop` is really sll `$0, $0, 0`, it's arguable whether or not the ID stage in clock 4 should be highlighted.)

仔细阅读课本后，将做如下的设计：遇到分支认为不运行分支，计数器继续递增。而将 branch 判定提前到 ID 段，一旦发现判断错误，就会向流水线中加入一个气泡，并将 PC 设定为分支位置，将IF寄存器清空（IF.Flush）。气泡是在时钟上沿进行（前文已提到），而

后面的需要在时钟下沿进行，为了影响后面所有指令的进行（在后文的处理器顶端模块编写中可以看到赋值顺序是从后向前进行的，以传递数据，这样做可以在不影响整个代码风格的情况下，以影响后面的指令，否则只会不会在前面阻塞，后面赋值又会覆盖）。

```
always @(negedge clk ) begin
    // Here is the clear signal before transition.
    // IF.Flush
    if(ID_BRANCH && BEQ) begin
        PC <= BRANCH_PC_ID;
        IF_PC <= 0;
        IF_INST <= 0;
    end
    //...
end
```

判断的方法需要直接使用前向传递的数据，判定两个操作数是否相等。否则会在特定的事件产生判断错误的结果（比如分支指令前恰好是 ALU 指令，这时必然要阻塞）。

```
// predict-not-taken
wire BEQ = (ForwardA_RES == ForwardB_RES);
```

2.5 代码风格与顶端设计

有了上面的考虑之后，就可以进行流水线的顶端设计了。
本处理器的**代码风格**如下：

- 所有的段寄存器都会采用前一阶段的名称作为开始。比如 IF/ID 寄存器中的 INST，就会被定义为

```
    reg [31:0] IF_INST;
```

  以表示该值由 IF 阶段发出。

- 所有的电线采用终止寄存器结尾。比如 IF 阶段的指令存储器发出指令信息以存储到 IF/ID 寄存器，就会被定义下面的方式，表示电线将会在 IF/ID 寄存器终止。

```
    wire [31:0] INST_IF;
```

- 所有段寄存器的值会在时钟上跳沿被逆阶段更新，也就意味着寄存器的值将会晚电线值一个周期。比如

```
        always @(posedge clk) begin
        // reset
        // WB
        // MEM
        // EX
        // ID
            ID_INST <= IF_INST;
        // IF
            IF_INST <= INST_IF;
        end
```

这样做就可以防止不确定的赋值顺序。当需要实时值时需要使用以阶段结尾的电线值（一般在时钟沿之外使用），在时钟沿更新时如果没有对应的直接输出电线，就要采用寄存器值更新。

- 当需要影响全阶段的运行时，需要在时钟下沿更新值。这一点与实验五的设计是一致的。比如跳转（含 JR 指令）、分支指令判断错误等：

```
    always @(negedge clk ) begin
        // IF.Flush
        if(ID_BRANCH && BEQ) begin
            PC <= BRANCH_PC_ID;
            IF_PC <= 0;
            IF_INST <= 0;
        end
        if(EX_JR) begin
            PC <= EX_ALU_RES;
            IF_PC <= 0;
            IF_INST <= 0;
            ID_PC <= 0;
            ID_INST <= 0;
            ID_REG_DST <= 0;
            ID_JUMP <= 0;
            ID_BRANCH <= 0;
            ID_MEM_READ <= 0;
            ID_MEM_TO_REG <= 0;
            ID_MEM_WRITE <= 0;
```

```
            ID_ALU_OP <= 0;
            ID_ALU_SRC <= 0;
            ID_IMM <= 0;
            ID_JAL <= 0;
            ID_REG_WRITE <= 0;
        end else
            PC <= EX_JUMP ? EX_JUMP_PC : PC;
    end
```

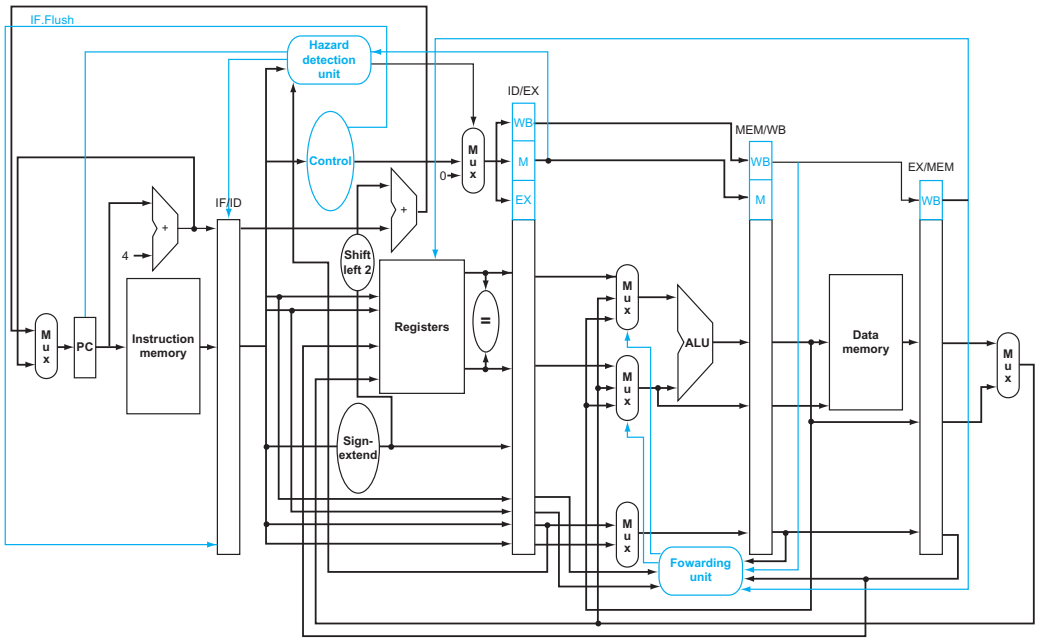


FIGURE 4.65 The final datapath and control for this chapter. Note that this is a stylized figure rather than a detailed datapath, so it's missing the ALUsrc Mux from Figure 4.57 and the multiplexor controls from Figure 4.51.

Listing 2: Top.v

```
`timescale 1ns / 1ps
//////////////////////////////////
// Company:
// Engineer: Zilong Li
//
// Create Date: 2021/06/05 12:37:39
// Design Name:
// Module Name: Top
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////


module Top(
    input clk,
    input reset
    );

```

```
    // Instruction Fetch (IF)
    reg [31:0] PC;
    reg [31:0] IF_PC;

    wire [31:0] INST_IF;

    reg [31:0] IF_INST;

    // Instruction Decoding (ID)
    reg [31:0] ID_PC;

    wire REG_DST_ID;
    wire JUMP_ID;
    wire BRANCH_ID;
    wire MEM_READ_ID;
    wire MEM_TO_REG_ID;
    wire MEM_WRITE_ID;
    wire [1:0] ALU_OP_ID;
    wire ALU_SRC_ID;
    wire IMM_ID;
    wire JAL_ID;
    wire REG_WRITE_ID;

    wire ZEXT;

    wire [31:0] READ_DATA1_ID;
    wire [31:0] READ_DATA2_ID;

    wire [31:0] OPRAND_ID;

    reg ID_REG_DST;
    reg ID_JUMP;
    reg ID_BRANCH;
    reg ID_MEM_READ;
    reg ID_MEM_TO_REG;
    reg ID_MEM_WRITE;
    reg [1:0] ID_ALU_OP;
    reg ID_ALU_SRC;
    reg ID_IMM;
    reg ID_JAL;
    reg ID_REG_WRITE;

    reg [31:0] ID_READ_DATA1;
    reg [31:0] ID_READ_DATA2;
    reg [31:0] ID_OPRAND;
    reg [31:0] ID_INST;

    wire [31:0] BRANCH_PC_ID = IF_PC + 4 + (OPRAND_ID <<
          2); // hasn't been PC + 4

    // Execution (EX)
    reg [31:0] EX_PC;
    reg [31:0] EX_JUMP_PC;

    wire ZERO_EX;
    wire [31:0] ALU_RES_EX;
    wire JR_EX;

    wire [3:0] ALU_CTR;
    wire SHAMT;

    reg EX_JR;
    reg EX_ZERO;
    reg [31:0] EX_ALU_RES;
    reg [4:0] EX_WRITE_REG;

    reg EX_JUMP;
    reg EX_BRANCH;
    reg EX_MEM_READ;
    reg EX_MEM_TO_REG;
    reg EX_MEM_WRITE;
    reg EX_JAL;
    reg EX_REG_WRITE;

    reg [31:0] EX_READ_DATA2;

    // Memory (MEM)
    reg [31:0] MEM_PC;

    wire PC_SRC = EX_BRANCH & EX_ZERO;
    wire [31:0] READ_DATA_MEM;

    reg MEM_REG_WRITE;
    reg MEM_MEM_TO_REG;
    reg MEM_JAL;

    reg [31:0] MEM_READ_DATA;
    reg [31:0] MEM_ALU_RES;
    reg [4:0] MEM_WRITE_REG;

    // Write Back (WB)
    wire REG_WRITE_WB = MEM_REG_WRITE;
    wire [31:0] WRITE_DATA_WB = MEM_JAL ? MEM_PC + 4 : (
          MEM_MEM_TO_REG ? MEM_READ_DATA : MEM_ALU_RES);
           // The next PC for jal
    wire [4:0] WRITE_REG_WB = MEM_WRITE_REG;

    // Forwarding
    wire [1:0] ForwardA = ((EX_REG_WRITE &
        (EX_WRITE_REG != 0) &
        (EX_WRITE_REG == ID_INST[25:21])) ?
            2'b10 :
            ((MEM_REG_WRITE & (MEM_WRITE_REG != 0)
            & (MEM_WRITE_REG == ID_INST[25:21])) ?
                2'b01 :
                2'b00));

    wire [1:0] ForwardB = ((EX_REG_WRITE &
        (EX_WRITE_REG != 0) &
        (EX_WRITE_REG == ID_INST[20:16])) ?
            2'b10 :
            ((MEM_REG_WRITE & (MEM_WRITE_REG != 0)
            & (MEM_WRITE_REG == ID_INST[20:16])) ?
                2'b01 :
                2'b00));
```

```
    wire [31:0] ForwardA_RES =
           ForwardA == 2'b00 ? ID_READ_DATA1 :
               (ForwardA == 2'b10 ? EX_ALU_RES :
                   WRITE_DATA_WB);

    wire [31:0] ForwardB_RES =
           ForwardB == 2'b00 ? ID_READ_DATA2 :
               (ForwardB == 2'b10 ? EX_ALU_RES :
                   WRITE_DATA_WB);


    // Stall
    wire stalling = (ID_MEM_READ & ((ID_INST[20:16] ==
          IF_INST[25:21]) | (ID_INST[20:16] == IF_INST
          [20:16]))) ? 1 : 0;

    // predict-not-taken
    wire BEQ = (ForwardA_RES == ForwardB_RES);

    // IF
    InstMemory instMemory(
        .readAddress(PC),
        .inst(INST_IF)
    );

    // ID
    Ctr mainCtr(
        .opCode(IF_INST[31:26]),
        .regDst(REG_DST_ID),
        .jump(JUMP_ID),
        .branch(BRANCH_ID),
        .memRead(MEM_READ_ID),
        .memToReg(MEM_TO_REG_ID),
        .memWrite(MEM_WRITE_ID),
        .aluOp(ALU_OP_ID),
        .aluSrc(ALU_SRC_ID),
        .imm(IMM_ID),
        .regWrite(REG_WRITE_ID),
        .jal(JAL_ID),
        .zext(ZEXT)
    );

    Registers registers(
        .clk(clk),
        .reset(reset),
        .readReg1(IF_INST[25:21]),
        .readReg2(IF_INST[20:16]),
        .writeReg(WRITE_REG_WB),        // After write
              back
        .writeData(WRITE_DATA_WB),      // ...
        .regWrite(REG_WRITE_WB),        // ...
        .readData1(READ_DATA1_ID),
        .readData2(READ_DATA2_ID)
    );

    signext signExt(
        .inst(IF_INST[15:0]),
        .zext(ZEXT),
        .data(OPRAND_ID)
    );

    // EX
    ALUCtr aluctr(
        .nop(ID_INST == 0 ? 1'b1 : 1'b0),
        .funct(ID_IMM ? ID_INST[31:26] : ID_INST[5:0]),
        .aluOp(ID_ALU_OP),
        .aluCtrOut(ALU_CTR),
        .jr(JR_EX),
        .shamt(SHAMT)
    );

    ALU alu(
        .input1(SHAMT ? ID_INST[10:6] : ForwardA_RES),
        .input2(ID_ALU_SRC ? ID_OPRAND : ForwardB_RES),
        .aluCtr(ALU_CTR),
        .zero(ZERO_EX),
        .aluRes(ALU_RES_EX)
    );

    // MEM
    dataMemory DataMemory(
        .Clk(clk),
        .address(EX_ALU_RES),
        .writeData(ForwardB_RES),
        .memWrite(EX_MEM_WRITE),
        .memRead(EX_MEM_READ),
        .readData(READ_DATA_MEM)
    );

    always @(posedge clk) begin
        if(reset) begin
            PC <= 0;
            IF_PC <= 0;
            IF_INST <= 0;

            ID_PC <= 0;
            ID_READ_DATA1 <= 0;
            ID_READ_DATA2 <= 0;
            ID_OPRAND <= 0;
            ID_INST <= 0;

            ID_REG_DST <= 0;
            ID_JUMP <= 0;
            ID_BRANCH <= 0;
            ID_MEM_READ <= 0;
            ID_MEM_TO_REG <= 0;
            ID_MEM_WRITE <= 0;
            ID_ALU_OP <= 0;
            ID_ALU_SRC <= 0;
            ID_IMM <= 0;
            ID_JAL <= 0;
            ID_REG_WRITE <= 0;

            EX_PC <= 0;
            EX_ZERO <= 0;
            EX_ALU_RES <= 0;
```

```
            EX_WRITE_REG <= 0;

            EX_JUMP <= 0;
            EX_BRANCH <= 0;
            EX_MEM_READ <= 0;
            EX_MEM_TO_REG <= 0;
            EX_MEM_WRITE <= 0;
            EX_REG_WRITE <= 0;

            EX_READ_DATA2 <= 0;

            MEM_REG_WRITE <= 0;

            MEM_READ_DATA <= 0;
            MEM_ALU_RES <= 0;
            MEM_WRITE_REG <= 0;
        end
        else begin
            // WB

            // MEM
            MEM_PC <= EX_PC;

            MEM_REG_WRITE <= EX_REG_WRITE;
            MEM_MEM_TO_REG <= EX_MEM_TO_REG;

            MEM_READ_DATA <= READ_DATA_MEM;
            MEM_ALU_RES <= EX_ALU_RES;
            MEM_WRITE_REG <= EX_WRITE_REG;
            MEM_JAL <= EX_JAL;

            // EX
            EX_PC <= ID_PC;

            EX_ZERO <= ZERO_EX;
            EX_ALU_RES <= ALU_RES_EX;
            EX_WRITE_REG <= ID_JAL ? 5'b11111 : (JR_EX ?
                  ID_INST[25:21] : (ID_REG_DST ? ID_INST
                [15:11] : ID_INST[20:16]));

            EX_JUMP <=         ID_JUMP;
            EX_BRANCH <=       ID_BRANCH;
            EX_MEM_READ <=     ID_MEM_READ;
            EX_MEM_TO_REG <=   ID_MEM_TO_REG;
            EX_MEM_WRITE <=    ID_MEM_WRITE;
            EX_REG_WRITE <=    ID_REG_WRITE;
            EX_JAL <=          ID_JAL;
            EX_JR <=           JR_EX;

            EX_JUMP_PC <= ID_PC[31:28] + (ID_INST[25:0]
                  << 2);

            EX_READ_DATA2 <=  ID_READ_DATA2;

            // ID
            ID_PC <= IF_PC;
            ID_INST <= IF_INST;
            ID_READ_DATA1 <= READ_DATA1_ID;
            ID_READ_DATA2 <= READ_DATA2_ID;
            ID_OPRAND <= OPRAND_ID;

            if (stalling || (ID_BRANCH && BEQ)) begin
                // nop
                ID_REG_DST <= 0;
                ID_JUMP <= 0;
                ID_BRANCH <= 0;
                ID_MEM_READ <= 0;
                ID_MEM_TO_REG <= 0;
                ID_MEM_WRITE <= 0;
                ID_ALU_OP <= 0;
                ID_ALU_SRC <= 0;
                ID_IMM <= 0;
                ID_JAL <= 0;
                ID_REG_WRITE <= 0;
            end else begin
                ID_REG_DST <= REG_DST_ID;
                ID_JUMP <= JUMP_ID;
                ID_BRANCH <= BRANCH_ID;
                ID_MEM_READ <= MEM_READ_ID;
                ID_MEM_TO_REG <= MEM_TO_REG_ID;
                ID_MEM_WRITE <= MEM_WRITE_ID;
                ID_ALU_OP <= ALU_OP_ID;
                ID_ALU_SRC <= ALU_SRC_ID;
                ID_IMM <= IMM_ID;
                ID_JAL <= JAL_ID;
                ID_REG_WRITE <= REG_WRITE_ID;
            end

            // IF
            if(stalling == 0) begin
                PC <= PC + 4;
                IF_PC <= PC;          // Has already been
                      PC + 4
                IF_INST <= INST_IF;
            end
        end
    end

    always @(negedge clk ) begin
        // Here is the clear signal before transition.
        // IF.Flush
        if(ID_BRANCH && BEQ) begin
            PC <= BRANCH_PC_ID;
            IF_PC <= 0;
            IF_INST <= 0;
        end
        if(EX_JR) begin
            PC <= EX_ALU_RES;
            IF_PC <= 0;
            IF_INST <= 0;
            ID_PC <= 0;
            ID_INST <= 0;
            ID_REG_DST <= 0;
            ID_JUMP <= 0;
            ID_BRANCH <= 0;
            ID_MEM_READ <= 0;
```

```
366             ID_MEM_TO_REG <= 0;
367             ID_MEM_WRITE <= 0;
368             ID_ALU_OP <= 0;
369             ID_ALU_SRC <= 0;
370             ID_IMM <= 0;
371             ID_JAL <= 0;
372             ID_REG_WRITE <= 0;
```

```
373         end else
374             PC <= EX_JUMP ? EX_JUMP_PC : PC;
375     end
376
377 endmodule
```

3 仿真结果

3.1 思考问题

在完成实现后，先回答一些思考题目。

1. 请思考和Lab5相比，Top模块中的主要变化处是什么？

   答 需要考虑赋值的顺序，不能像 Lab 5 一样将电线直接接上。更重要的是，需要考虑冒险问题。

2. 之前的模块是否要修改?

   答 需要修改，比如 Registers 的相关修改，见 2.2。

3. 另外，由于MEM级的Branch会影响PCSrc的值，从而影响下次PC，因此需要为Control加入RESET功能，将Branch置零。

   答 该问题在预测不发生机制统一考虑，见 2.4。

4. 由于各种变量名称极为复杂，推荐在着手编码之前为自己选择一套命名规范

   答 见代码风格 2.5。

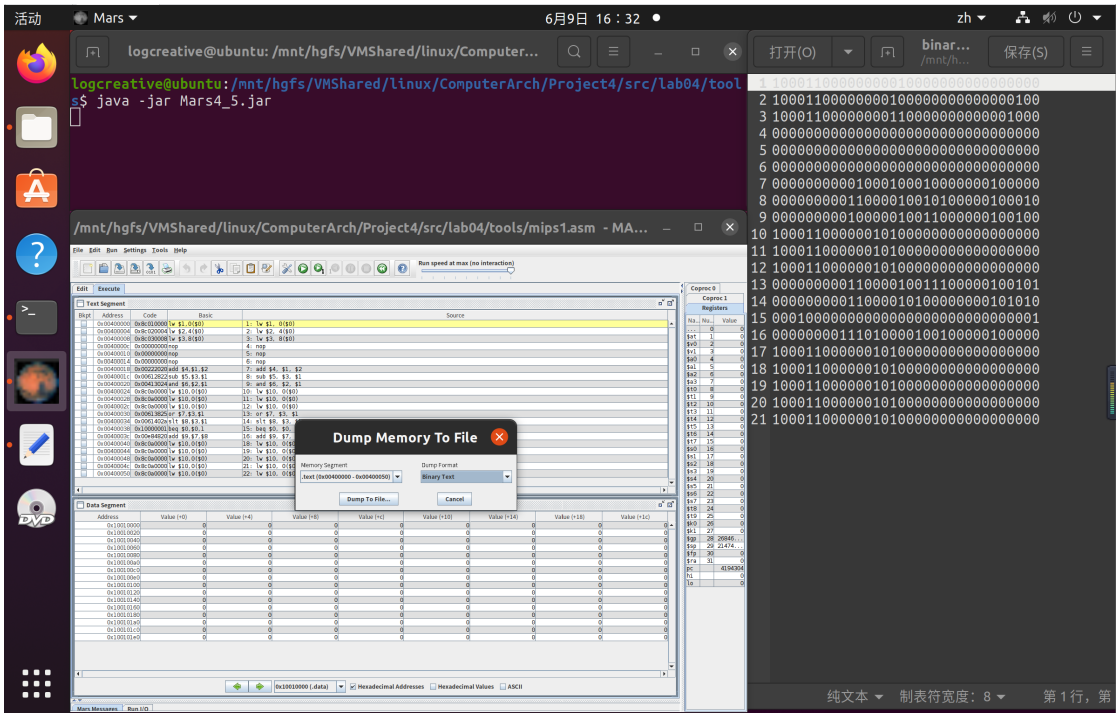
5. 在实现实验目的2.，3.，4.的内容时建议把1.的代码或工程备份一遍才开始

   答 本实验采用 git 版本控制工具统一管理。

3.2 Mars 汇编程序

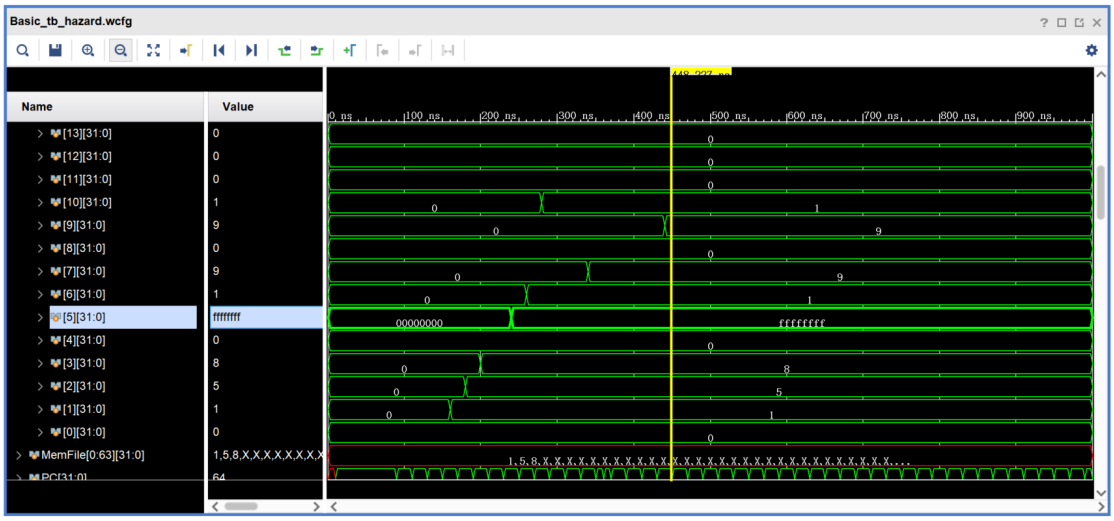
为了加快测试进度，在本实验中采用计算机结构理论课上的 Mars 工具快速生成汇编二进制代码。

打开 Mars。新建 asm 汇编程序，然后汇编后，在文件选单中 Dump Memory，另存为 Binary Text 就可以生成二进制程序码。值得注意的是，需要采用标签指定 beq 的目标位置，也就是实验指导书中的示例汇编代码风格。

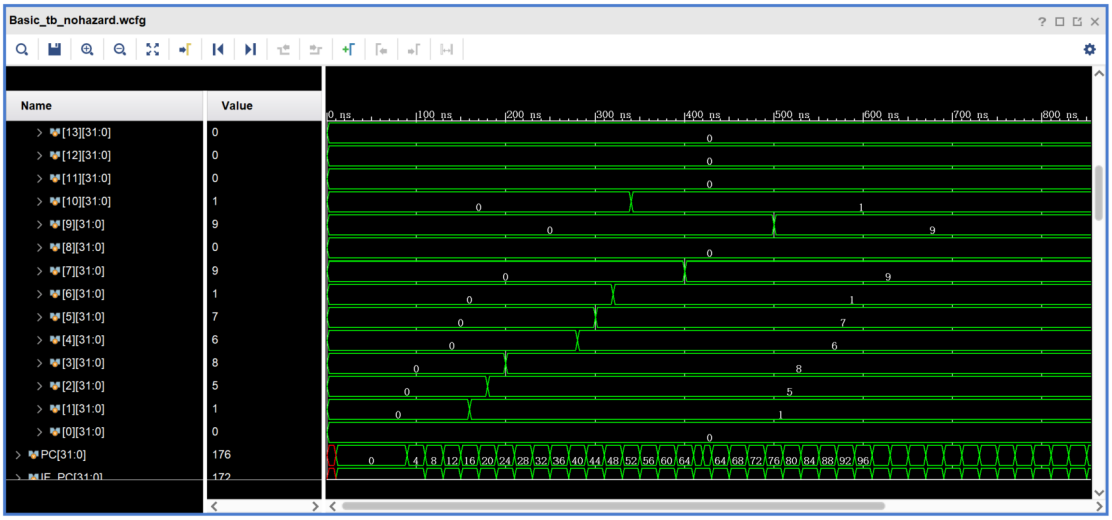


3.3 无优化仿真

首先采用教材代码，然后会有数据冒险问题。



手动插入空语句后（lw-use型需要三个空语句），就可以解决数据冒险问题。从而得到预期结果。



Listing 3: example.asm

```
lw $1, 0($0)
lw $2, 4($0)
lw $3, 8($0)
add $4, $1, $2
sub $5, $3, $1
and $6, $2, $1
lw $10, 0($0)
lw $10, 0($0)
lw $10, 0($0)
or $7, $3, $1
slt $8, $3, $1
beq $0, $0, end
add $9, $7, $8
end:lw $10, 0($0)
lw $10, 0($0)
lw $10, 0($0)
lw $10, 0($0)
```

Listing 4: example.asm(no risk)

```
lw $1, 0($0)
lw $2, 4($0)
lw $3, 8($0)
nop
nop
nop
add $4, $1, $2
sub $5, $3, $1
and $6, $2, $1
lw $10, 0($0)
lw $10, 0($0)
lw $10, 0($0)
or $7, $3, $1
slt $8, $3, $1
beq $0, $0, end
add $9, $7, $8
end:lw $10, 0($0)
lw $10, 0($0)
lw $10, 0($0)
lw $10, 0($0)
lw $10, 0($0)
```

3.4 前向传递

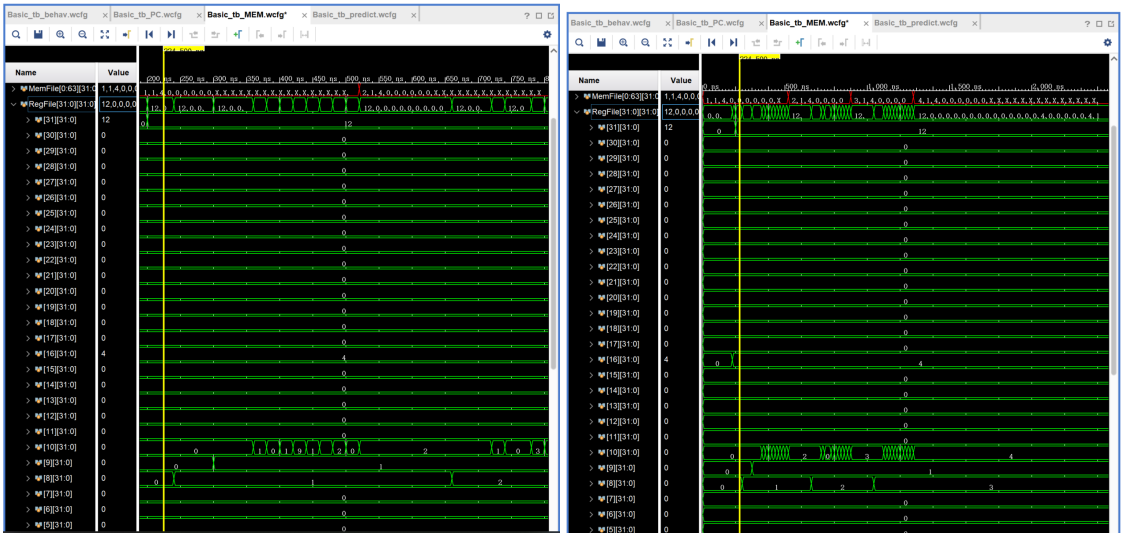
从本小节起，采用于实验 5 同样的代码来说明问题。

该指令文件主要的作用是测试所有的运算功能，并在每一个循环对 10 号寄存器 + 1，并存储到 0 号存储单元中，直到其超过刚开始的限制寄存器的存储数字（这里是 4），之后就会进入短循环，不会再对寄存器和存储器进行修改。

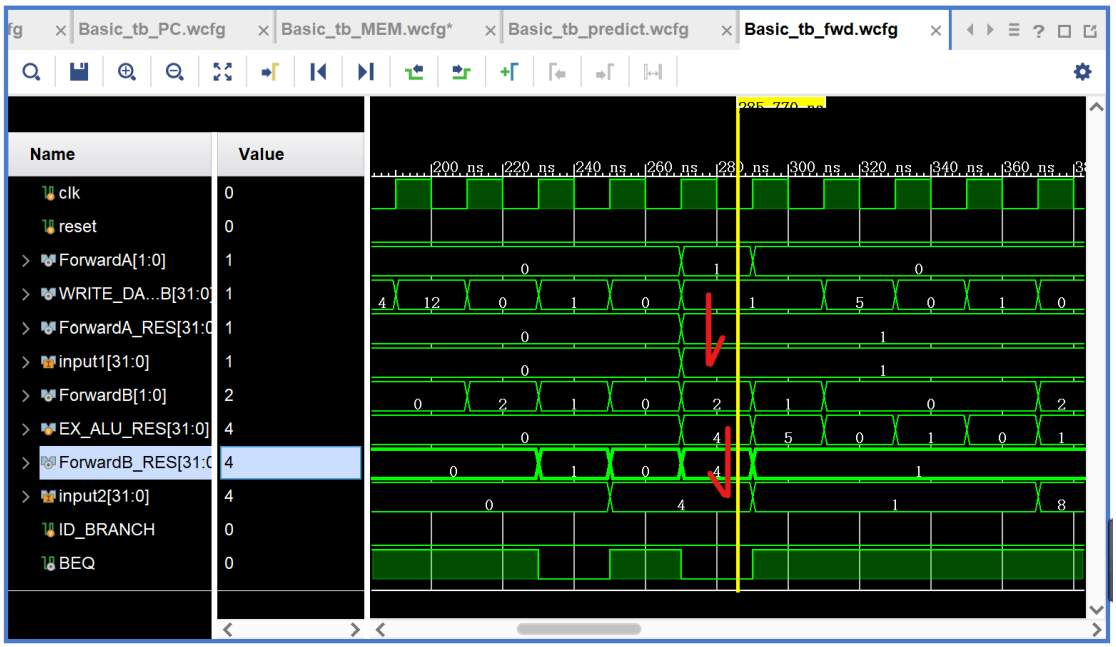
Listing 5: simple.asm

```
nop
lw  $16, 8($0)          # $0 zero register
jal 4
nop
lw  $8, 0($0)
lw  $9, 4($0)
sub $10, $8, $9
and $10, $8, $9
slt $10, $8, $9
or  $10, $8, $9
addi $10, $8, 8
andi $10, $8, -1
ori  $10, $8, -1
sll $10, $8, 1
srl $10, $8, 1
add $10, $8, $9         # final save: += 1
sw  $10, 0($0)
beq $10, $16, 1
jr $31
j 16
nop
nop
```

首先仿真得到了与上个实验相同的结果，并达到了运行预期（4 之后不更改）。

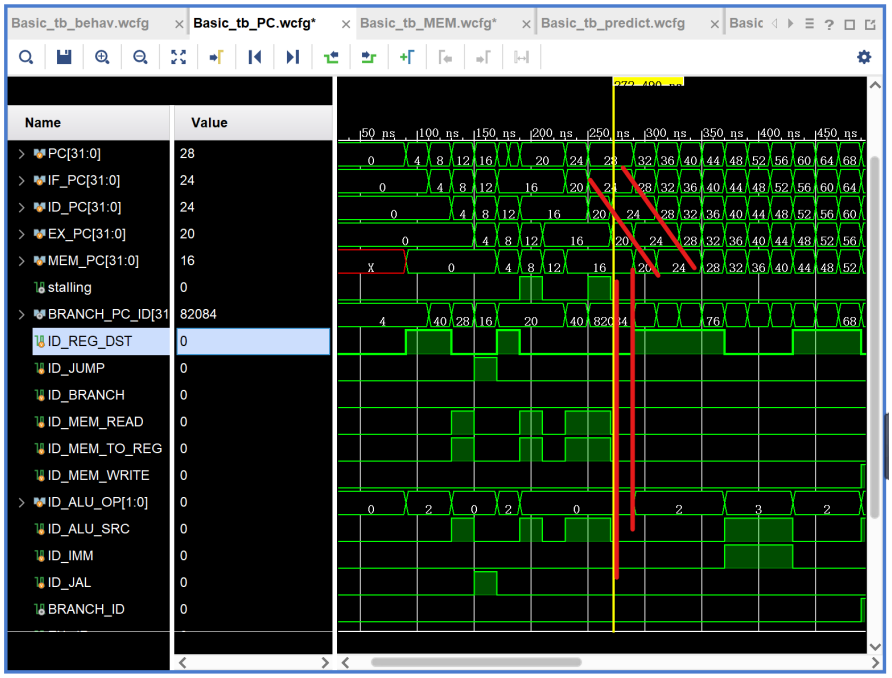


对于前向传递，可以看到由于 ForwardA 被置于 1，ALU第一个输入就会被导入写回数据；由于 ForwardB 被置于 2，ALU第二个输入就会被导入上一个 ALU 的结果。结果如下图所示。该前向传递发生在开始的运算时期，此时的 lw 尚未写回寄存器，但是需要其值。



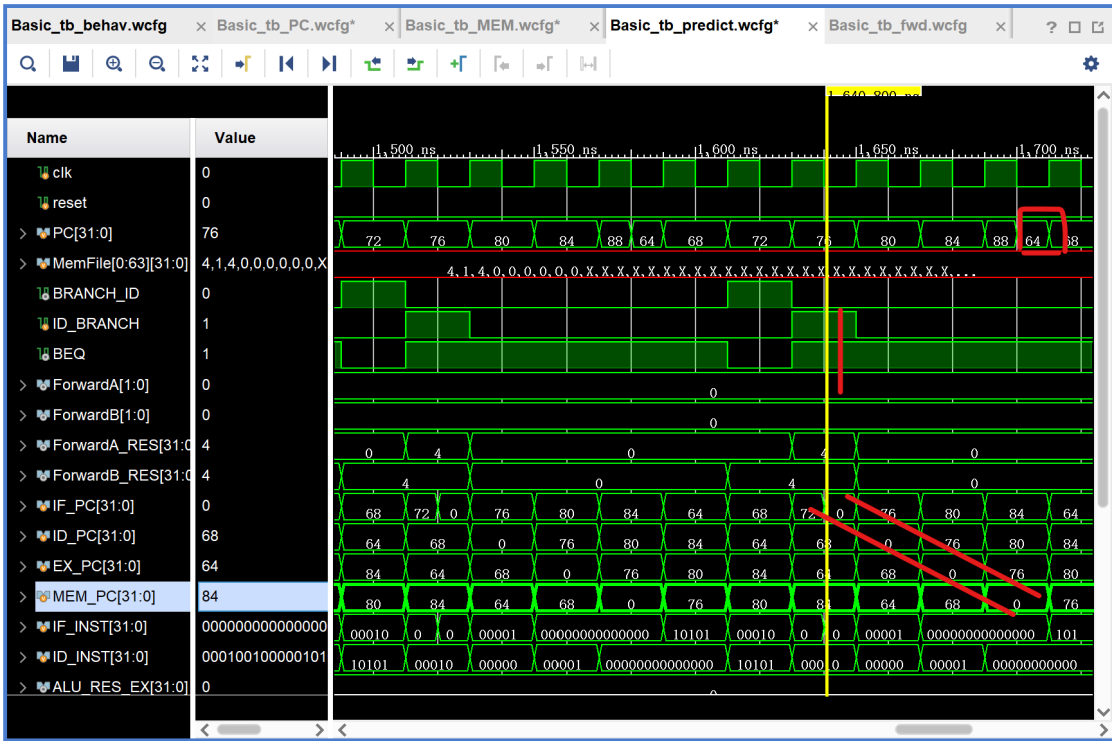
3.5 停顿机制

同一个时期，由于前向传递还不足，就需要停顿一个周期，这个时候 PC 就会被拉长。与此对应的是，所有 ID 控制信号会被清空。



3.6 预测不发生机制

当最后一个判断时，由于分支发生了，所以将会插入一个气泡，以清除，并且将PC的值进入短循环。



4 实验心得

计算机系统结构实验终于结束，奋战 4 日的流水处理器终于完成。在设计流水处理器的时候，一直在重新翻看课本，复习概念。在这个过程中加深了我对流水处理器的许多理解。

由于时间不足，所以 31 条指令的选做任务并没有完成，由于需要大改，可能会对代码风格产生影响，并且需要增加很多端口。做完一些减少停顿的策略，理解如何解决冒险足矣。

多次的失败，让我意识到是自己对细节的把握不足，认真看书，总是可以找到对应的答案。感谢实验组给我这个设计处理器的机会，该实验对于我理解处理器的核心细节有很大的帮助。

灵活使用课上的知识，可以更好的解决问题，提高效率（比如使用 Mars 汇编程序）。形成良好的代码风格有助于减少问题发生的几率。并且在无法调试出问题时，不妨建立最小工作集，新建波形，通过代码追踪相关端口，以解决问题。

本实验的处理器原理图形来自参考文献的课本。

参考文献

[1] Computer Organization and Design — The Hardware/Software Interface, David A. Patterson and John L. Hennessy, Fifth Edition, Elsevier Inc., 2014