CS334 计算机组成实验

LAB6

简单的类 MIPS 多周期流水化处理器

周鼎
5140219268

指导老师：王老师

April 7, 2016

Contents

Part I
基本的流水线实现

1 五级流水线的顶层结构

将指令执行分为五个阶段，每个阶段每条指令只使用一个主要模块，如下图所示，相比单周期，每阶段间添加寄存器存储必要的信息。

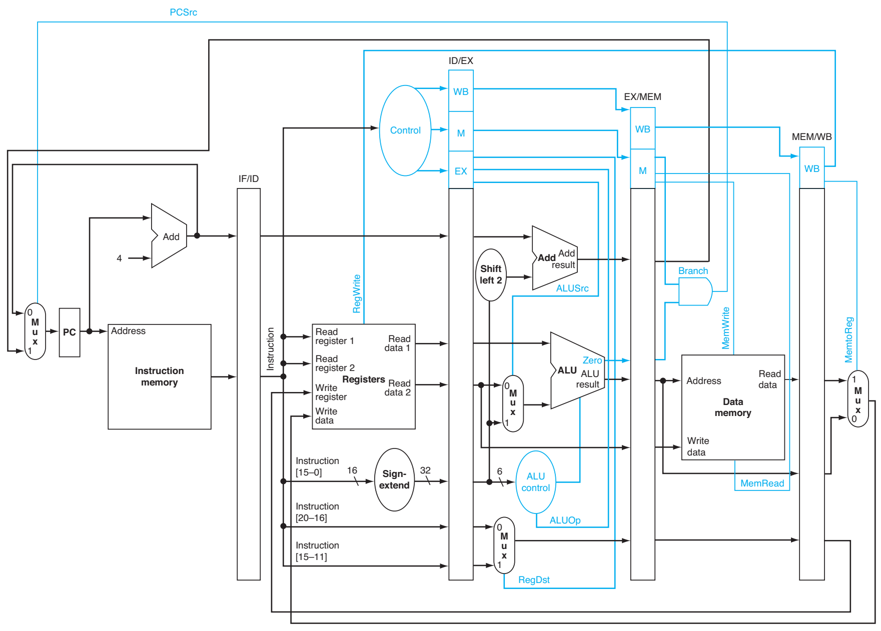


Figure 1: 顶层设计

2 系统定义顶层所需的寄存器

流水线的顶层中需要较多的寄存器，为了做到不重不漏，按阶段和用途分类定义。分为 IF/ID, ID/EX, EX/MEM, MEM/WB 四个大块，内分为控制信号和指令信息，指令信息里有 PC+4，也有后续指令执行过程得到的结果；控制信号按信号使用的阶段分类，具体见如下代码。

```
    //1.0 For stage IF to ID;
    reg [31:0] IF_ID_PCAdd4;
    reg [31:0] IF_ID_Instr;

    //2.0 For stage ID to EX;
    reg [31:0]  ID_EX_PCAdd4;
    reg [31:0]  ID_EX_RegReadData1;
    reg [31:0]  ID_EX_RegReadData2;
    reg [31:0]  ID_EX_SignExt;
    reg [20:16] ID_EX_InstHigh;
    reg [15:11] ID_EX_InstLow;
    //2.1 To EX
    reg ID_EX_RegDst;
    reg [1:0] ID_EX_ALUOp;
    reg       ID_EX_ALUSrc;
    //2.2 To MEM
    reg ID_EX_Branch;
    reg ID_EX_MemWrite;
    reg ID_EX_MemRead;
    //2.3 To WB
    reg ID_EX_MemToReg;
    reg ID_EX_RegWrite;

    //3.0 For stage EX to MEM;
    reg        EX_MEM_Zero;
    reg [31:0] EX_MEM_ALUOut;
    reg [31:0] EX_MEM_BranchAddress;
    reg [31:0] EX_MEM_RegReadData2;
    //3.1 To MEM
    reg EX_MEM_Branch;
    reg EX_MEM_MemWrite;
    reg EX_MEM_MemRead;
    reg [4:0]  EX_MEM_RegWriteAddress;  //rt or rd
    //3.2 To WB
    reg EX_MEM_MemToReg;
    reg EX_MEM_RegWrite;

    //4.0 For stage MEM to WB;
    reg [31:0] MEM_WB_ALUOut;
    reg [31:0] MEM_WB_MemReadData;
    reg [4:0]  MEM_WB_RegWriteAddress;
    reg        MEM_WB_RegWrite;
    reg        MEM_WB_MemToReg;
```

reg_def_top.v

3 分阶段定义网线类型并完成实例化和连接

3.1 IF Stage

IF 阶段的实现要特别注意线型变量不要和 WB 阶段混淆，以及 NEXT_PC 的实现。

```
//1.0 IF
  wire IF_PCSrc,  //for MUX sel signal
       IF_Branch,
      IF_Zero;
  wire [31:0] IF_BranchAddress;
  wire [31:0] IF_CurrPC;
  wire [31:0] IF_NextPC;
  wire [31:0] IF_PCAdd4;
  wire [31:0] IF_Instr;
  //associate with reg
  assign IF_BranchAddress = EX_MEM_BranchAddress;
  assign IF_Zero = EX_MEM_Zero;
  assign IF_Branch = EX_MEM_Branch;
  //Combinational Logic
  assign IF_PCAdd4 = IF_CurrPC +4;
  assign IF_PCSrc  = IF_Branch & IF_Zero & ~RESET;
  assign IF_NextPC = IF_PCSrc? IF_BranchAddress: IF_PCAdd4;  //MUX
  //assign IF_NextPC = IF_PCAdd4;
  //update PC at posedge
```

```
Pc mainPC (
   .clock_in(CLK),
   .nextPC(IF_NextPC),
   .currPC(IF_CurrPC),
   .rst(RESET)
   );
wire [31:0] IF_Index;
assign IF_Index = IF_CurrPC>>2;
instructionMemory InstrMemory (
   .address(IF_Index),
   .clock_in(CLK),
   .reset(RESET),
   .readData(IF_Instr)
   );
//InstMem;

```

IF.v

3.2 ID Stage

ID 阶段所需要的 mainCtr 与单周期的功能完全一样，课直接使用，唯一不同是产生的控制信号需要用寄存器存储至相应阶段使用。

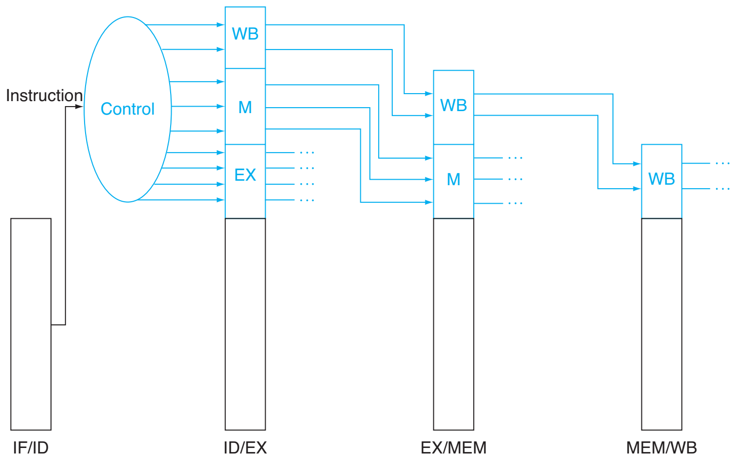


Figure 2: 后三阶段的控制信号

```
wire [5:0] ID_OpCode;
wire [31:0] ID_PCAdd4;
wire [4:0] ID_RegReadAddress1;
wire [4:0] ID_RegReadAddress2;
wire [15:0] ID_Imm;
wire [31:0] ID_SignExt;
wire [20:16] ID_InstHigh;
wire [15:11] ID_InstLow;
wire [31:0] ID_RegReadData1;
wire [31:0] ID_RegReadData2;
wire WB_RegWrite;
wire WB_MemToReg;
wire [4:0]  WB_RegWriteAddress;
wire [31:0] WB_MemReadData;
```

```
  wire  [31:0]  WB_ALUOut;
  wire  [31:0]  WB_RegWriteData;
  //associate with reg
  assign ID_Instr = IF_ID_Instr;
  assign ID_PCAdd4 = IF_ID_PCAdd4;
  assign ID_OpCode = ID_Instr[31:26];
  assign ID_RegReadAddress1 = ID_Instr[25:21];
  assign ID_RegReadAddress2 = ID_Instr[20:16];
  assign ID_Imm = ID_Instr[15:0];
  assign ID_InstHigh = ID_Instr[20:16];
  assign ID_InstLow  = ID_Instr[15:11];
  assign WB_RegWrite = MEM_WB_RegWrite;
  assign WB_MemToReg = MEM_WB_MemToReg;
  assign WB_RegWriteAddress = MEM_WB_RegWriteAddress;
  assign WB_ALUOut = MEM_WB_ALUOut;
  assign WB_MemReadData = MEM_WB_MemReadData;
  assign WB_RegWriteData = WB_MemToReg? WB_MemReadData: WB_ALUOut;

  Register mainReg (
    .clock_in(CLK),
    .regWrite(WB_RegWrite),
    .readReg1(ID_RegReadAddress1),
    .readReg2(ID_RegReadAddress2),
    .writeReg(WB_RegWriteAddress),
    .writeData(WB_RegWriteData),
    .reset(RESET),
    .readData1(ID_RegReadData1),
    .readData2(ID_RegReadData2),
   .ioinput(SW),
   .iooutput(LED)
    );

   //2.1 To EX
   wire [1:0] ID_ALUOp;
   wire ID_RegDst;
   wire ID_ALUSrc;
   //2.2 To MEM
   wire ID_Branch;
   wire ID_MemWrite;
   wire ID_MemRead;
   //2.3 To WB
   wire ID_MemToReg;
   wire ID_RegWrite;
   wire JUMP;  //of no use
   Ctr mainCtr (
    .opCode(ID_OpCode),
    .regDst(ID_RegDst),
    .aluSrc(ID_ALUSrc),
    .memToReg(ID_MemToReg),
    .regWrite(ID_RegWrite),
    .memRead(ID_MemRead),
    .memWrite(ID_MemWrite),
    .branch(ID_Branch),
    .aluOp(ID_ALUOp),
    .jump(JUMP)
    );
   signExt mainSignExt (
    .inst(ID_Imm),
    .data(ID_SignExt)
    );
```

ID.v

3.3 EX & MEM

后面的过程与单周期十分类似，不在赘述，仅列出代码。

```verilog
//2.0 For stage ID to EX;
wire [31:0] EX_PCAdd4;
wire [31:0] EX_ALUSrc1;
wire [31:0] EX_ALUSrc2;
wire [31:0] EX_RegReadData2;
wire [31:0] EX_SignExt;
wire [20:16] EX_InstHigh;
wire [15:11] EX_InstLow;
wire EX_RegDst,
     EX_ALUSrc;
wire [1:0] EX_ALUOp;
wire [4:0] EX_RegWriteAddress;
wire [5:0] EX_Funct;
//associate with Reg
assign EX_RegDst   = ID_EX_RegDst;
assign EX_ALUOp    = ID_EX_ALUOp;
assign EX_ALUSrc1 = ID_EX_RegReadData1;
assign EX_ALUSrc   = ID_EX_ALUSrc;       //contral signal
assign EX_ALUSrc2 = EX_ALUSrc? EX_SignExt: EX_RegReadData2;      //MUX
assign EX_RegWriteAddress = EX_RegDst? EX_InstLow: EX_InstHigh;
assign EX_PCAdd4 = ID_EX_PCAdd4;
assign EX_RegReadData2 = ID_EX_RegReadData2;
assign EX_SignExt   = ID_EX_SignExt;
assign EX_Funct      = ID_EX_SignExt[5:0];
assign EX_InstHigh = ID_EX_InstHigh;
assign EX_InstLow   = ID_EX_InstLow;
//Instances
wire [3:0]    EX_ALUCtr;
AluCtr mainALUCtr (
  .aluOp(EX_ALUOp),
  .funct(EX_Funct),
  .aluCtr(EX_ALUCtr)
  );
wire EX_Zero;
wire [31:0] EX_ALUOut;
Alu mainALU (
  .input1(EX_ALUSrc1),
  .input2(EX_ALUSrc2),
  .aluCtr(EX_ALUCtr),
  .zero(EX_Zero),
  .aluRes(EX_ALUOut)
  );
 //2.2 To MEM
 wire EX_Branch,
    EX_MemWrite,
    EX_MemRead;
 assign EX_Branch    = ID_EX_Branch;
 assign EX_MemWrite = ID_EX_MemWrite;
 assign EX_MemRead   = ID_EX_MemRead;
 //2.3 To WB
 wire EX_MemToReg,
    EX_RegWrite;
 assign EX_MemToReg = ID_EX_MemToReg;
 assign EX_RegWrite = ID_EX_RegWrite;
 //BranchAddress
 wire [31:0] EX_BranchAddress;
 assign EX_BranchAddress = (EX_SignExt<<2) + EX_PCAdd4;
```

EX.v

```
//4.0 MEM
  wire MEM_MemWrite,
       MEM_MemRead;
  wire [4:0] MEM_RegWriteAddress;
  wire [31:0] MEM_ALUOut;
  wire [31:0] MEM_MemWriteData;
  wire [31:0] MEM_MemReadData;
  //associate with reg
  assign MEM_MemWrite = EX_MEM_MemWrite;
  assign MEM_MemRead = EX_MEM_MemRead;
  assign MEM_RegWriteAddress = EX_MEM_RegWriteAddress;
  assign MEM_ALUOut = EX_MEM_ALUOut;
  assign MEM_MemWriteData = EX_MEM_RegReadData2;
  //instances
   dataMemory DataMemory (
     .clock_in(CLK),
     .address(MEM_ALUOut),
     .writeData(MEM_MemWriteData),
     .readData(MEM_MemReadData),
     .memWrite(MEM_MemWrite),
     .memRead(MEM_MemRead)
     );
   //to WB
   wire MEM_MemToReg,
        MEM_RegWrite;
   assign MEM_MemToReg = EX_MEM_MemToReg;
   assign MEM_RegWrite = EX_MEM_RegWrite;
```

MEM.v

3.4 阶段间的寄存器更新

每个阶段之间的寄存器在上升沿更新,

```
//1.5 IF/ID REG UPDATE
  always @(posedge CLK) begin
        IF_ID_PCAdd4 <= IF_PCAdd4;
        IF_ID_Instr <= IF_Instr;
    end
//2.5 ID/EX REG UPDATE
  always @(posedge CLK) begin
        //2.0 For Stage ID To EX
         ID_EX_PCAdd4        <= ID_PCAdd4;
         ID_EX_RegReadData1  <= ID_RegReadData1;
         ID_EX_RegReadData2  <= ID_RegReadData2;
         ID_EX_SignExt       <= ID_SignExt;
         ID_EX_InstHigh      <= ID_InstHigh;
         ID_EX_InstLow       <= ID_InstLow;
         //2.1 To EX
         ID_EX_RegDst        <= ID_RegDst;
         ID_EX_ALUOp         <= ID_ALUOp;
         ID_EX_ALUSrc        <= ID_ALUSrc;
         //2.2 To MEM
         ID_EX_Branch        <= ID_Branch;
         ID_EX_MemWrite      <= ID_MemWrite;
         ID_EX_MemRead       <= ID_MemRead;
         //2.3 To WB
         ID_EX_MemToReg      <= ID_MemToReg;
         ID_EX_RegWrite      <= ID_RegWrite;
    end
//3.5 EX/MEM REG UPDATE
```

```
    always @(posedge CLK)   begin
      EX_MEM_Branch <= EX_Branch;
      EX_MEM_MemWrite <= EX_MemWrite;
      EX_MEM_MemRead <= EX_MemRead;
      EX_MEM_MemToReg <= EX_MemToReg;
      EX_MEM_RegWrite <= EX_RegWrite;
      EX_MEM_BranchAddress <= EX_BranchAddress;
      EX_MEM_Zero <= EX_Zero;
      EX_MEM_ALUOut <= EX_ALUOut;
      EX_MEM_RegWriteAddress <= EX_RegWriteAddress;
      EX_MEM_RegReadData2 <= EX_RegReadData2;
    end
//4.5 MEM/WB REG UPDATE
  always @(posedge CLK) begin
        MEM_WB_ALUOut = MEM_ALUOut;
        MEM_WB_MemReadData = MEM_MemReadData;
        MEM_WB_RegWriteAddress = MEM_RegWriteAddress;
        MEM_WB_RegWrite = MEM_RegWrite;
        MEM_WB_MemToReg = MEM_MemToReg;
      end
```

internalRegUpdate.v

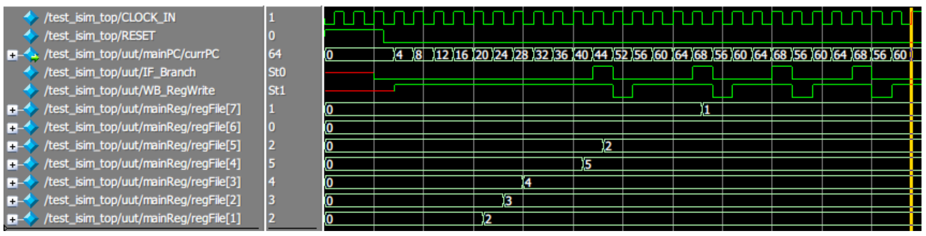
4 ModelSim 仿真

```
10001100000000001000000000101100
10001100000000010000000000110000
10001100000000011000000000110100
00000000000000000000000000000000
00000000000000000000000000000000
00000000001000100010000000100000
00000000011000010010100000100010
00010000000000000000000000000100
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000010000110011000000100000
10001100000001110000000000101000
00010000000000001111111111111111
00000000000000000000000000000000
00000000000000000000000000000000
```

```
lw $1, 44($0)    ; 2
lw $2, 48($0)    ; 3
lw $3, 52($0)    ; 4
NOP
NOP
add $4, $1, $2   ; $4=5
sub $5, $3, $1   ; $5=2
beq $0, $0, end  ;
NOP
NOP
NOP
add $6, $2, $3   ; not executed
end:    lw $7, 40($0)   ; 1
beq $0, $0, -1 ;
NOP
NOP
```

Figure 3: 二进制指令

Figure 4: MIPS 指令



/test_isim_top/CLOCK_IN 1
/test_isim_top/RESET 0
/test_isim_top/uut/mainPC/currPC 64
0 4 8 12 16 20 24 28 32 36 40 44 52 56 60 64 68 56 60 64 68 56 60 64 68 56 60
/test_isim_top/uut/IF_Branch St0
/test_isim_top/uut/WB_RegWrite St1
/test_isim_top/uut/mainReg/regFile[7] 1
/test_isim_top/uut/mainReg/regFile[6] 0
/test_isim_top/uut/mainReg/regFile[5] 2
/test_isim_top/uut/mainReg/regFile[4] 5
/test_isim_top/uut/mainReg/regFile[3] 4
/test_isim_top/uut/mainReg/regFile[2] 3
/test_isim_top/uut/mainReg/regFile[1] 2

Figure 5: ModelSim 仿真

Part II
改进

实现的最基本的流水线已经能够运行，但在测试程序时很容易发现问题：

1. BEQ 指令后必须插入 NOP，这在 beq 不跳转时会浪费时钟周期；
2. 没有实现 jump 指令；
3. 没有解决 data hazard, 必须插入 NOP 避免 read-after-write;

下面将着手解决这些问题。

5 Flush

5.1 改进方案

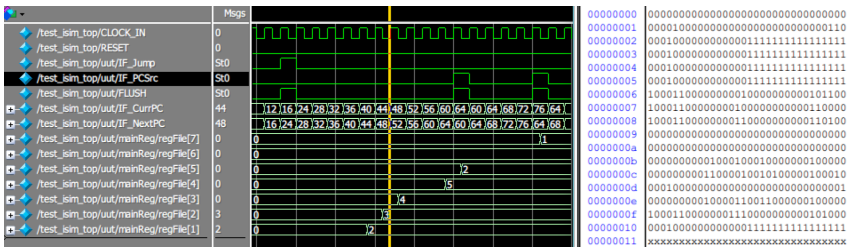
执行 beq 指令时，当 IF_Branch 有效之后，下一上升沿将 load 跳转后的指令，此时已经执行到 ID,EX 阶段的指令需要 Flush 掉，因此 ID/EX, EX/MEM, MEM/WB 的所有寄存器清零。下面是修改后的 regUpdate 代码。

```
//1.5 IF/ID REG UPDATE
always @(posedge CLK) begin
      IF_ID_PCAdd4 <= IF_Branch ? 0: IF_PCAdd4;
      IF_ID_Instr  <= IF_Branch ? 0: IF_Instr;
    end
//2.5 ID/EX REG UPDATE
always @(posedge CLK) begin
       //2.0 For Stage ID To EX
       ID_EX_PCAdd4       <= IF_Branch ? 0: ID_PCAdd4;
       ID_EX_RegReadData1 <= IF_Branch ? 0: ID_RegReadData1;
       ID_EX_RegReadData2 <= IF_Branch ? 0: ID_RegReadData2;
       ID_EX_SignExt      <= IF_Branch ? 0: ID_SignExt;
       ID_EX_InstHigh     <= IF_Branch ? 0: ID_InstHigh;
       ID_EX_InstLow      <= IF_Branch ? 0: ID_InstLow;
       //2.1 To EX
       ID_EX_RegDst       <= IF_Branch ? 0: ID_RegDst;
       ID_EX_ALUOp        <= IF_Branch ? 0: ID_ALUOp;
       ID_EX_ALUSrc       <= IF_Branch ? 0: ID_ALUSrc;
       //2.2 To MEM
       ID_EX_Branch       <= IF_Branch ? 0: ID_Branch;
       ID_EX_MemWrite     <= IF_Branch ? 0: ID_MemWrite;
       ID_EX_MemRead      <= IF_Branch ? 0: ID_MemRead;
       //2.3 To WB
       ID_EX_MemToReg     <= IF_Branch ? 0: ID_MemToReg;
       ID_EX_RegWrite     <= IF_Branch ? 0: ID_RegWrite;
    end
//3.5 EX/MEM REG UPDATE
always @(posedge CLK)   begin
        EX_MEM_Branch <= IF_Branch ? 0: EX_Branch;
        EX_MEM_MemWrite <= IF_Branch ? 0: EX_MemWrite;
        EX_MEM_MemRead <= IF_Branch ? 0: EX_MemRead;
        EX_MEM_MemToReg <= IF_Branch ? 0: EX_MemToReg;
        EX_MEM_RegWrite <= IF_Branch ? 0: EX_RegWrite;
        EX_MEM_BranchAddress <= IF_Branch ? 0: EX_BranchAddress;
        EX_MEM_Zero <= IF_Branch ? 0: EX_Zero;
        EX_MEM_ALUOut <= IF_Branch ? 0: EX_ALUOut;
```

```
            EX_MEM_RegWriteAddress <= IF_Branch ? 0: EX_RegWriteAddress;
            EX_MEM_RegReadData2 <= IF_Branch ? 0: EX_RegReadData2;
        end
```

regUpdate.v

5.2 ModelSim 仿真



```
/test_isim_top/CLOCK_IN                   0
/test_isim_top/RESET                      0
/test_isim_top/uut/IF_Jump                St0
/test_isim_top/uut/IF_PCSrc               St0
/test_isim_top/uut/FLUSH                  St0
/test_isim_top/uut/IF_CurrPC              44
/test_isim_top/uut/IF_NextPC              48
/test_isim_top/uut/mainReg/regFile[7]     0
/test_isim_top/uut/mainReg/regFile[6]     0
/test_isim_top/uut/mainReg/regFile[5]     0
/test_isim_top/uut/mainReg/regFile[4]     0
/test_isim_top/uut/mainReg/regFile[3]     0
/test_isim_top/uut/mainReg/regFile[2]     3
/test_isim_top/uut/mainReg/regFile[1]     2
```

```
00000000  00000000000000000000000000000000
00000001  00001000000000000000000000000110
00000002  00010000000000001111111111111111
00000003  00010000000000001111111111111111
00000004  00010000000000001111111111111111
00000005  00010000000000001111111111111111
00000006  10001100000000010000000000101100
00000007  10001100000000010000000000110000
00000008  10001100000000011000000000110100
00000009  00000000000000000000000000000000
0000000a  00000000000000000000000000000000
0000000b  00000000001000100010000000100000
0000000c  00000000011000010010100000100010
0000000d  00010000000000000000000000000001
0000000e  00000000010000110011000000100000
0000000f  10001100000001110000000000101000
00000010  00010000000000001111111111111111
00000011  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 6: ModelSim Flush 仿真

```
00000000000000000000000000000000
10001100000000010000000000101100
10001100000000010000000000110000
10001100000000011000000000110100
00000000000000000000000000000000
00000000000000000000000000000000
00000000001000100010000000100000
00000000011000010010100000100010
00010000000000000000000000000001
00000000010000110011000000100000
10001100000001110000000000101000
00010000000000001111111111111111
```

```
NOP
lw $1, 44($0)   ; 2
lw $2, 48($0)   ; 3
lw $3, 52($0)   ; 4
NOP
NOP
add $4, $1, $2   ; $4=5
sub $5, $3, $1   ; $5=2
beq $0, $0, end ;
add $6, $2, $3   ; not executed
end:    lw $7, 40($0)   ; 1
beq $0, $0, -1 ;
```

Figure 7: 二进制指令

Figure 8: MIPS 指令

6 JUMP

6.1 改进方案

实现 jump 需要生成跳转地址，给 nextPC 添加一个 source，为了避免 jump 和 beq 指令的冲突，将 jump 指令延长至 MEM 阶段完成，同时 jump 也需要 Flush 功能。，

```
\\Def Regs and Wires
    reg ID_EX_Jump;
    reg [31:0] ID_EX_Instr;
    reg EX_MEM_Jump;
    reg [31:0] EX_MEM_JumpAddress;
    wire ID_Jump;
    wire EX_Jump;
    wire [31:0] EX_JumpAddress;
    wire IF_Jump;
    wire  [31:0]  IF_JumpAdress;
    assign  EX_Jump = ID_EX_Jump;
    assign  IF_Jump = EX_MEM_Jump;
    assign  EX_Instr = ID_EX_Instr;
    assign  EX_JumpAddress[27:2]  = EX_Instr[25:0];
    assign  EX_JumpAddress[1:0]  = 2'b00;
    assign  IF_JumpAddress = EX_MEM_JumpAddress;
\\Update Regs
    always @(posedge CLK)   begin
     ID_EX_Jump <= ID_Jump;
     ID_EX_Instr <= ID_Instr;
     EX_MEM_Jump <= EX_Jump;
     EX_MEM_JumpAddress <= EX_JumpAddress;
    end
\\Update PC
   assign  IF_PCAdd4 = IF_CurrPC +4;
   assign  IF_PCSrc   = IF_Branch & IF_Zero & ~RESET;
   wire  [31:0]  IF_NextPCJ;
   assign  IF_NextPCJ = IF_Jump? IF_JumpAddress : IF_PCAdd4;
   assign  IF_NextPC = IF_PCSrc? IF_BranchAddress: IF_NextPCJ;   //MUX
```

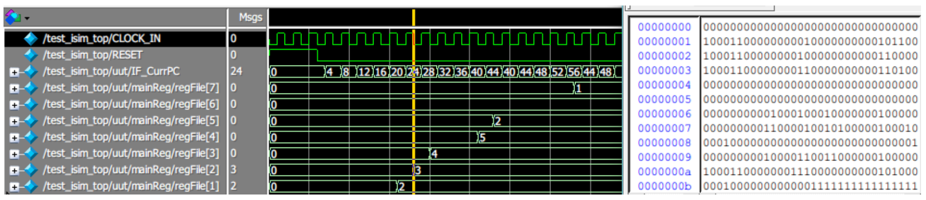
6.2 ModelSim 仿真

```
00001000000000000000000000000110
00010000000000001111111111111111
00010000000000001111111111111111
00010000000000001111111111111111
00010000000000001111111111111111
10001100000000010000000000101100
10001100000000100000000000110000
10001100000000110000000000110100
00000000000000000000000000000000
00000000000000000000000000000000
00000000001000100010000000100000
00000000011000010010100000100010
00010000000000000000000000000001
00000000010000110011000000100000
10001100000001110000000000101000
00010000000000001111111111111111
```

Figure 9: 二进制指令

```
J 6;
circle1:     beq $0, $0, circle1 ;
circle2:     beq $0, $0, circle2 ;
circle3:     beq $0, $0, circle3 ;
circle4:     beq $0, $0, circle4 ;
lw $1, 44($0)   ; 2
lw $2, 48($0)   ; 3
lw $3, 52($0)   ; 4
NOP
NOP
add $4, $1, $2   ; $4=5
sub $5, $3, $1   ; $5=2
beq $0, $0, end  ;
add $6, $2, $3   ; not executed
end:     lw $7, 40($0)   ; 1
beq $0, $0, -1 ;
```

Figure 10: MIPS 指令



Msgs
/test_isim_top/CLOCK_IN 0
/test_isim_top/RESET 0
/test_isim_top/uut/IF_CurrPC 24
/test_isim_top/uut/mainReg/regFile[7] 0
/test_isim_top/uut/mainReg/regFile[6] 0
/test_isim_top/uut/mainReg/regFile[5] 0
/test_isim_top/uut/mainReg/regFile[4] 0
/test_isim_top/uut/mainReg/regFile[3] 0
/test_isim_top/uut/mainReg/regFile[2] 3
/test_isim_top/uut/mainReg/regFile[1] 2
0 4 8 12 16 20 24 28 32 36 40 44 40 44 48 52 56 44 48

```
00000000 00000000000000000000000000000000
00000001 10001100000000010000000000101100
00000002 10001100000000100000000000110000
00000003 10001100000000110000000000110100
00000004 00000000000000000000000000000000
00000005 00000000000000000000000000000000
00000006 00000000001000100010000000100000
00000007 00000000011000010010100000100010
00000008 00010000000000000000000000000001
00000009 00000000010000110011000000100000
0000000a 10001100000001110000000000101000
0000000b 00010000000000001111111111111111
```

Figure 11: ModelSim JUMP 仿真

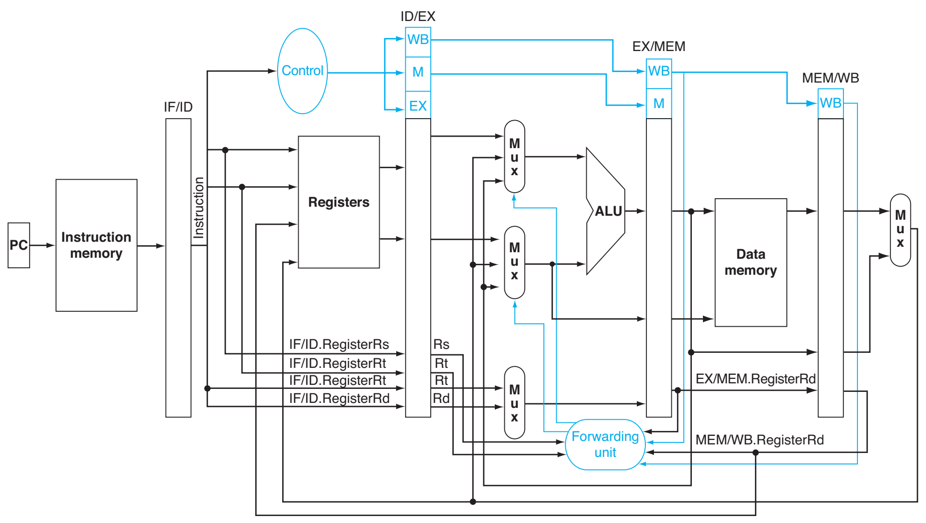


Figure 12: top with Forward Unit

7 数据冒险

7.1 MUX

对 ALU 的两个输入增加两个 MUX，其输入分别由信号 ForwardA，ForwardB 控制，控制信号对应的输入如下表所示；将其模块化，代码如下

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Figure 13: MUX 控制信号

```
module forwardMux(
    input  [31:0] ID_EX,
    input  [31:0] EX_MEM,
    input  [31:0] MEM_WB,
    input  [1:0]  Forward,
    output [31:0] Sel
    );
   wire  [31:0] TEMP;
   assign Sel =  Forward[1]  ? EX_MEM : TEMP;
   assign TEMP= Forward[0]  ? MEM_WB : ID_EX;
endmodule
```

forwardMUX.v

7.2 Forward Unit

对于 R-type 指令引起的 read-after-write, 希望通过如下图所示的方案解决数据冒险。创建模块 ForwardUnit 产生 forwardMux 的控制信号，实现图示的旁路。

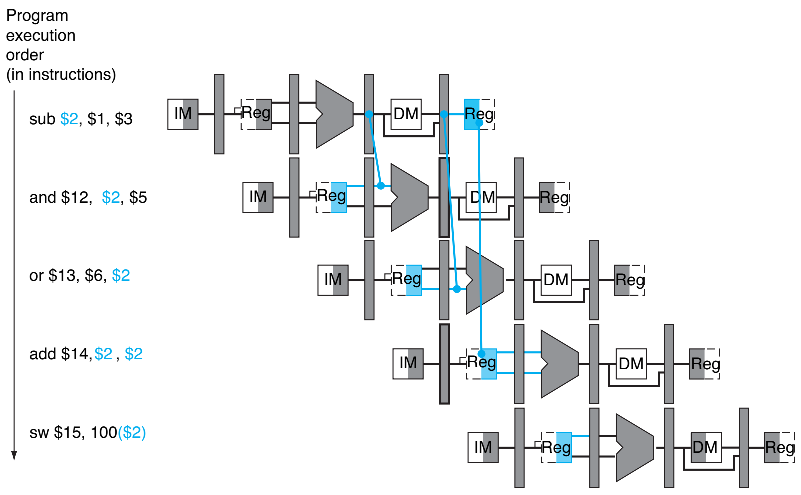


Figure 14: Forwarding

旁路 Forward 逻辑很直观，见代码内部，不再单独列出。

```
module forwardUnit(
    input  [4:0]  rs,
    input  [4:0]  rt,
    input  MEM_WB_regWrite,
    input  [4:0]  MEM_WB_rd,
    input  EX_MEM_regWrite,
    input  [4:0]  EX_MEM_rd,
    output reg  [1:0]  forwardA,
    output reg  [1:0]  forwardB,
  input  rst
    );
   always @(*)   begin
   if(rst  | MEM_WB_rd = 0  | EX_MEM_rd = 0)   begin
    forwardA  =  'b00;
    forwardB  =  'b00;
    end
      else begin
    if(MEM_WB_regWrite & MEM_WB_rd == rs)     forwardA  =  'b01;
    else  if(EX_MEM_regWrite & EX_MEM_rd == rs)
                                              forwardA  =  'b10;
      else forwardA  =  'b00;

    if(MEM_WB_regWrite  & MEM_WB_rd == rt)    forwardB  =  'b01;
    else  if(EX_MEM_regWrite  & EX_MEM_rd == rt)
                                              forwardB  =  'b10;
      else forwardB  =  'b00;
    end
   end
endmodule
```

forwardUnit.v

7.3 连接信号线及实例化

如顶层的信号连接所示，连接相应的信号。

```
//FORWARDING PART
//MUX
  //wire [31:0] EX_ALUSrc1_f;   //as mux output;
  //wire [31:0] EX_ALUSrc2_f;   //defined before ALU
  wire [31:0] EX_MEM_ALU;   //input from other stage
  wire [31:0] MEM_WB_ALU;
  assign EX_MEM_ALU = EX_MEM_ALUOut;
  assign MEM_WB_ALU = MEM_WB_ALUOut;
  wire [1:0] forwardA;
  wire [1:0] forwardB;

//FORWARD UNIT
    wire [4:0] EX_MEM_regWriteAddress_f;
    wire [4:0] MEM_WB_regWriteAddress_f;
    wire MEM_WB_regWrite_f;
    wire EX_MEM_regWrite_f;
    wire [4:0] rs_f;
    wire [4:0] rt_f;
    assign EX_MEM_regWriteAddress_f = EX_MEM_RegWriteAddress;
    assign MEM_WB_regWriteAddress_f = MEM_WB_RegWriteAddress;
    assign EX_MEM_regWrite_f = EX_MEM_RegWrite;
    assign MEM_WB_regWrite_f = MEM_WB_RegWrite;
    assign rs_f = ID_EX_Instr[25:21];//ID_EX
    assign rt_f = ID_EX_Instr[20:16];
//instances
    forwardUnit mainFUnit (
      .rs(rs_f),
      .rt(rt_f),
      .MEM_WB_regWrite(MEM_WB_regWrite_f),
      .MEM_WB_rd(MEM_WB_regWriteAddress_f),
      .EX_MEM_regWrite(EX_MEM_regWrite_f),
      .EX_MEM_rd(EX_MEM_regWriteAddress_f),
      .forwardA(forwardA),
      .forwardB(forwardB),
    .rst(RESET)
      );
  forwardMux MUXA (
      .ID_EX(EX_ALUSrc1),
      .EX_MEM(EX_MEM_ALU),
      .MEM_WB(MEM_WB_ALU),
      .Forward(forwardA),
      .Sel(EX_ALUSrc1_f)
      );
    forwardMux MUXB (
      .ID_EX(EX_ALUSrc2),
      .EX_MEM(EX_MEM_ALU),
      .MEM_WB(MEM_WB_ALU),
      .Forward(forwardB),
      .Sel(EX_ALUSrc2_f)
      );
endmodule
```

forwardMUX.v

7.4 ModelSim 仿真

使用 ModelSim 进行仿真，可以看到 data hazard 已经解决。

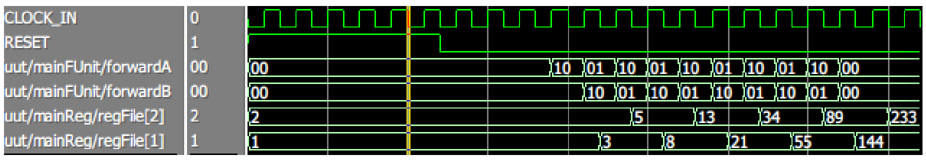


Figure 15: ModelSim Forwarding Unit

```
00000000001000100000100000100000
00000000001000100001000000100000
00000000001000100000100000100000
00000000001000100001000000100000
00000000001000100000100000100000
00000000001000100001000000100000
......
```

```
add $1, $1, $2  ;
add $2, $1, $2  ;
add $1, $1, $2  ;
add $2, $1, $2  ;
add $1, $1, $2  ;
add $2, $1, $2  ;
......
```

Figure 16: 二进制指令

Figure 17: MIPS 指令

Part III
上板试验

8 IO Scheme

IO Port	Function
LED[7]	查看 clock 信号
LED[6]	查看 reset 信号
LED[5:0]	查看寄存器数值
SW[3]	调整时钟快/慢模式
SW[2:0]	设置寄存器数值输出模式
Button	输入 reste 信号

Table 1: IO Scheme

SW[2:0]	MODE
000	led[5:3] 为 $2,led[2:0] 为 $1
001	led 显示 $1
011	led 显示 $2
111	led 显示 PC

设计使用 8 个 LED 作为输出，4 个开关和 1 个复位式按钮作为输入，具体功能见下图。因板上接口限制，将只对输出寄存器 $1,$2 和 PC，通过 SW[2:0] 选择输出模式，见左图。

9 IO 控制逻辑

```
////                GENRERATING SLOW CLOCK                ////
  wire CLK;
  reg [26:0] Buffer = 0;
  always@ (posedge CLOCK) Buffer = Buffer + 1;
  assign CLK = FAST ? CLOCK : Buffer[26];
////                IO MODE SEL                           ////
  wire [31:0] IF_CurrPC;
  assign LED[7] = RESET;
  assign LED[6] = CLK;
  wire [5:0] OUTPUT;
  assign LED[5:0] = OUTPUT;
  wire [31:0] reg1;
  wire [31:0] reg2;
  wire [5:0]  reg12;
  assign reg12[5:3] = reg2;
  assign reg12[2:0] = reg1;
  wire [31:0] CURR_PC_IO;
  assign CURR_PC_IO = IF_CurrPC>>2;
  wire [5:0] TEMP1;
  wire [5:0] TEMP2;
  assign TEMP1 = MODE[0] ? reg1:reg12;
  assign TEMP2 = MODE[1] ? reg2:TEMP1;
  assign OUTPUT = MODE[2]? CURR_PC_IO:TEMP2;
```

IO.v

10 User Constraint File

```
NET "CLOCK"    LOC = C9;
NET "LED[0]"   LOC = F12;
NET "LED[1]"   LOC = E12;
NET "LED[2]"   LOC = E11;
NET "LED[3]"   LOC = F11;
NET "LED[4]"   LOC = C11;
NET "LED[5]"   LOC = D11;
NET "LED[6]"   LOC = E9;
NET "LED[7]"   LOC = F9;
NET "MODE[0]"  LOC = L13;
NET "MODE[1]"  LOC = L14;
NET "MODE[2]"  LOC = H18;
NET "FAST"     LOC = N17;
NET "RESET"    LOC = K17 | IOSTANDARD = LVTTL | PULLDOWN;
```

top.ucf