计算机系统结构实验报告 实验 6

姓名：卞思沅

学号：518021910656

日期：2020/06/27

目录：

1. 实验概述

1.1 实验名称

简单的类 MIPS 多周期流水线处理器设计与实现

1.2 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．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

3．在 2.的基础上，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

PS：也允许考虑将 Stall 与 Forwarding 结合起来实现

4．在 3.的基础上，通过 predict-not-taken 或延时转移策略解决控制冒险/竞争，减少 制竞争带来的流水线停顿延时，进一步提高处理器性能

PS：也允许考虑将 2.、3.和 4.结合起来设计

5．在 4.的基础上，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）

6．中断、异常处理（选做）

7．Cache 的设计（选做）

1.3 实验内容

1．CPU 的流水化设计与软、硬件实现

2．功能仿真

3．上板验证

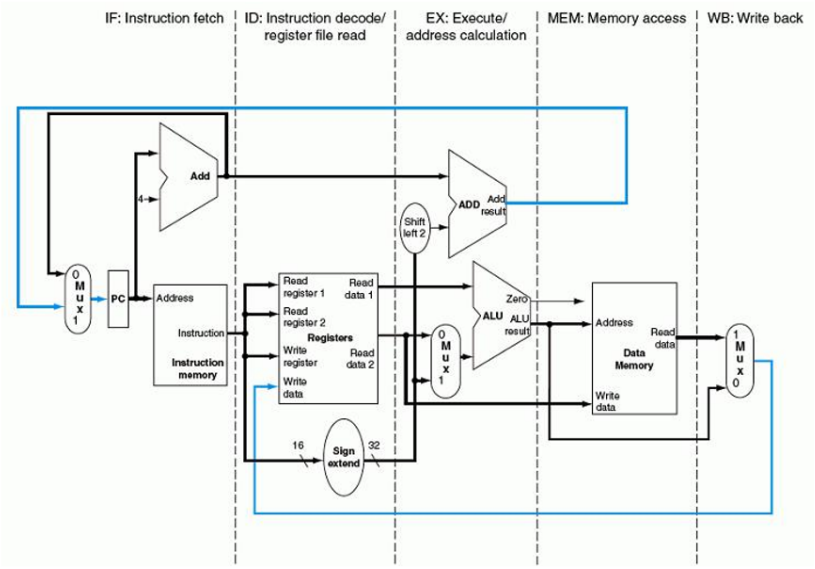
2. 实验描述

2.1 TOP 模块

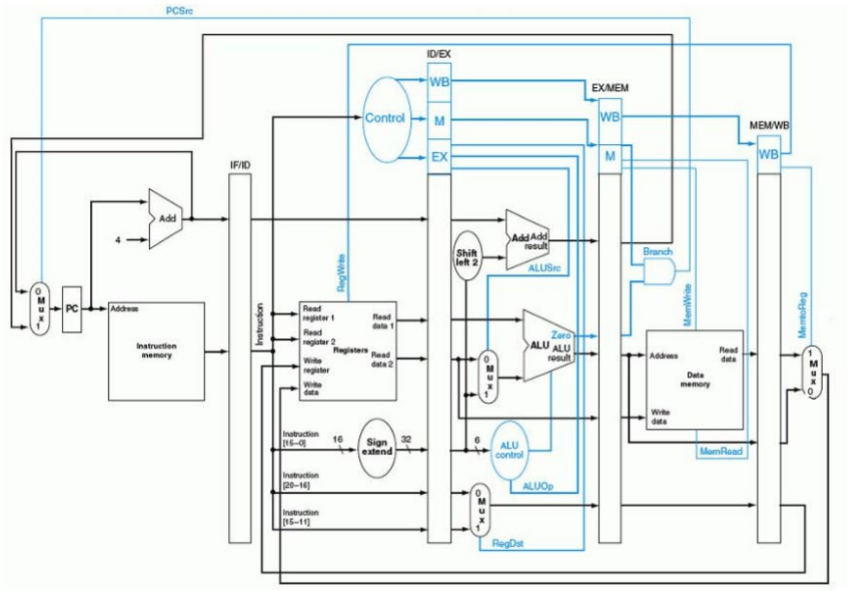
2.1.1 TOP 模块简述

1. 实验描述：前几次实验已经完成了 CPU 各部分的主要功能模块，因此需要设计流水线的 Top 模块（包括修改 Control 模块等，以及修改模块间互联的定义，根据设计需要可添加所需的功能模块）。

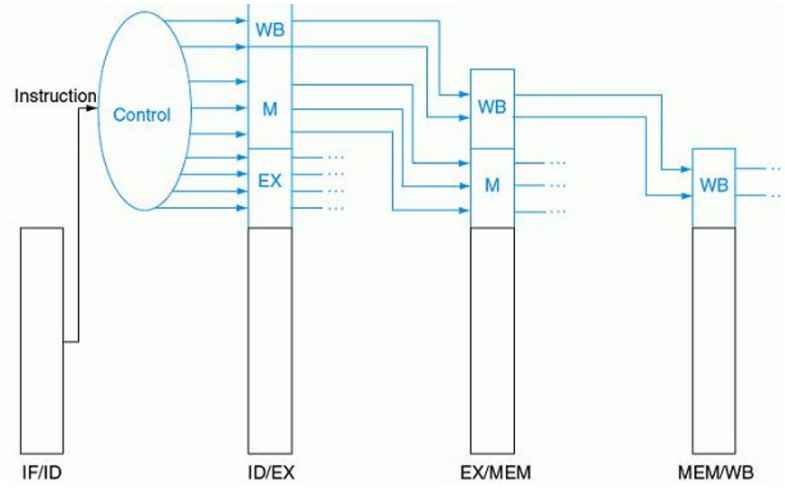
2. 模块描述：下面是流水的主要结构。



将单周期 CPU 进行分割，插入 4 级寄存器，将其分割为 IF，ID，EX，M，WB 五大部分：



其中 Control 的输出需要被加入流水线寄存器保存下来，以供后续每级流水使用。如下图所示：



3. 添加先前已经完成的各模块

添加之前完成的功能模块（包括指令源文件），还需自己查阅流水线 CPU 的理论及技术资 料来设计自己考虑所需的模块。

4. 新建 Top 模块源文件

5. 仿真测试

2.1.2 TOP 模块实验代码及结果

下面分别展示了 Top.v(主函数)， IsShift 模块（检查该指令是否是 shift 指令且需要用到 shamt）,Stall_detect.v (检测是否需要添加 stall), NeedFlush.v (检测是否需要将指令 flush)。具体解释见下面代码中的注释。

TOP.v:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2020/06/29 17:25:31
// 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
    );

    reg [31:0] PC;              // PC
    reg [31:0] INST;            // The inst at IF stage
    reg [31:0] mem_inst[31:0];
    reg [31:0] Inst_ID;         // The inst at ID stage

    // pipeline registers
    //For example, the variable ended by xxx_IDEX means it is the pipeline register between ID and
 EX stage
    reg MemRead_IDEX, MemWrite_IDEX, MemtoReg_IDEX, RegWrite_IDEX, ALUSrc_IDEX, isShift_IDEX;
    reg [3:0] ALUCtrOut_IDEX;
    reg [31:0] ReadData1_EX;
    reg [31:0] ReadData2_EX;
    reg [31:0] Ext_data_IDEX;
    reg [4:0] DstReg_IDEX;
    reg [4:0] shamt_IDEX;
    reg [4:0] Reg_rs_EX;
    reg [4:0] Reg_rt_EX;

    reg MemRead_EXMEM, MemWrite_EXMEM, MemtoReg_EXMEM, RegWrite_EXMEM;
    reg [31:0] ReadData2_EXMEM;
    reg [31:0] ALURes_EXMEM;
    reg Zero_EXMEM;
    reg [4:0] DstReg_EXMEM;

    reg MemtoReg_MEMWB;
    reg RegWrite_MEMWB;
```

```
reg [4:0] DstReg_MEMWB;
reg [31:0] ALURes_MEMWB;
reg [31:0] ReadData_MEMWB;

// Those pipeline registers are only used by jal instruction
reg [31:0] PC_IFID;
reg [31:0] PC_IDEX;
reg [31:0] PC_EXMEM;
reg [31:0] PC_MEMWB;
reg Jal_IDEX, Jal_EXMEM, Jal_MEMWB;


//Initialize
initial begin
    $readmemb("mem_inst.txt", mem_inst ,0);
    PC <= 0;
    INST = 0;
end



//IF STAGE
wire [31:0] JUMP_ADDRESS;
wire [31:0] Branch_res;
wire [31:0] Branch_res0;
wire [31:0] PC_4;
wire [31:0] PC_NEW0;
wire [31:0] PC_NEW;
wire STALL, FLUSH;

always@ (posedge CLK)
begin
    if(Reset)
        PC<=0;
    else
        if(!STALL && !FLUSH)
            PC <= PC_NEW;
        else PC <= PC_4;
    INST <= mem_inst[PC>>2];
end


always@ (posedge CLK)
begin
    if(FLUSH)
```

```
        Inst_ID <= 0; // If need flush, do not update ID!
    else if(!STALL) begin
        Inst_ID <= INST;
        PC_IFID <= PC;
    end
    // If stall, do not update PC
end



//ID STAGE
wire REG_DST, JUMP, MEM_READ, MEM_TO_REG, MEM_WRITE, ALU_SRC, REG_WRITE_ID,
        REG_WRITE, IS_SHIFT;
wire [1:0] BRANCH; //10 -> beq, 11 -> bne

wire [3:0] ALU_OP;
wire [3:0] ALU_CTR_OUT;
wire [31:0] READ_DATA1;
wire [31:0] READ_DATA2;
wire [31:0] EXT_DATA;
wire [4:0] WRITE_REG;
wire [31:0] WRITE_DATA_REG;
wire [31:0] WRITE_DATA_REG0;

Ctr mainCtr(
    .OpCode(Inst_ID[31:26]),
    .regDst(REG_DST),
    .aluSrc(ALU_SRC),
    .memToReg(MEM_TO_REG),
    .regWrite(REG_WRITE_ID),
    .memRead(MEM_READ),
    .memWrite(MEM_WRITE),
    .branch(BRANCH),
    .aluOp(ALU_OP),
    .jump(JUMP)
);

Registers registers(
    .Clk(CLK),
    .readReg1(Inst_ID[25:21]),
    .readReg2(Inst_ID[20:16]),
    .writeReg(WRITE_REG),
    .writeData(WRITE_DATA_REG),
    .regWrite(REG_WRITE),
    .readData1(READ_DATA1),
```

```
        .readData2(READ_DATA2)
    );

    signext se(
        .inst(Inst_ID[15:0]),
        .data(EXT_DATA)
    );

    ALUCtr aluCtr(
        .ALUop(ALU_OP),
        .Funct(Inst_ID[5:0]),
        .ALUCtrOut(ALU_CTR_OUT)
    );

    wire IS_LOAD = MemtoReg_IDEX & RegWrite_IDEX; //whether it is a load instruction

    Stall_detect stalldetect(
        .CLK(CLK),
        .OpCode(Inst_ID[31:26]),
        .funct(Inst_ID[5:0]),
        .readReg1(Inst_ID[25:21]),
        .readReg2(Inst_ID[20:16]),
        .needWrite(RegWrite_IDEX),
        .isLoad(IS_LOAD),
        .LoadReg(DstReg_IDEX),
        .Stall(STALL)
    );

    wire [31:0] INPUT1;
    wire [31:0] INPUT2;
    wire ZERO, ZERO1, JAL;

    IsShift isShift(.opcode(Inst_ID[31:26]), .funct(Inst_ID[5:0]), .shift(IS_SHIFT)); // whether i
t is a shift instruction and needs shamt
    assign ZERO1 = READ_DATA1==READ_DATA2 ? 1:0;         // whether rs = rt
    assign JAL = (Inst_ID[31:26] == 6'b000011) ? 1:0;   // whether it is a JAL instruction

    assign PC_4 = PC+4;
    assign JUMP_ADDRESS = {PC_4[31:28],Inst_ID[25:0]<<2};
    assign Branch_res0 =(BRANCH == 2'b10 & ZERO1) ? (EXT_DATA<<2)+PC_4 : PC_4;       // whether be
q is taken
    assign Branch_res =(BRANCH == 2'b11 & !ZERO1) ? (EXT_DATA<<2)+PC_4 : Branch_res0;// whether bn
e is taken
```

```
    assign PC_NEW0 = JUMP ? JUMP_ADDRESS : Branch_res;                              // whether it
is a jump inst
    assign PC_NEW = ({Inst_ID[31:26], Inst_ID[5:0]} == 12'b000000_001000) ? READ_DATA1 : PC_NEW0;
// for jr inst

    NeedFlush needflush(
        .opcode(Inst_ID[31:26]),
        .funct(Inst_ID[5:0]),
        .zero(ZERO1),
        .flush(FLUSH)
    );

    always@ (posedge CLK)
    begin
        // Update of IDEX registers
        if(STALL == 0)
        begin
            MemRead_IDEX <= MEM_READ;
            MemWrite_IDEX <= MEM_WRITE;
            MemtoReg_IDEX <= MEM_TO_REG;
            RegWrite_IDEX <= REG_WRITE_ID;
            DstReg_IDEX <= REG_DST ? Inst_ID[15:11] : Inst_ID[20:16]; // Rt : Rd
            ReadData1_EX <= READ_DATA1;
            ReadData2_EX <= READ_DATA2;
            ALUSrc_IDEX <= ALU_SRC;
            Reg_rs_EX <= Inst_ID[25:21];
            Reg_rt_EX <= Inst_ID[20:16];
            isShift_IDEX <= IS_SHIFT;
            shamt_IDEX <= Inst_ID[10:6];

            Ext_data_IDEX <= EXT_DATA;
            ALUCtrOut_IDEX <= ALU_CTR_OUT;
            Jal_IDEX <= JAL;
            PC_IDEX <= PC_IFID;
        end

        else begin
            RegWrite_IDEX <= 0;
            MemtoReg_IDEX <= 0;
            MemWrite_IDEX <= 0;
            isShift_IDEX <= 0;
            Jal_IDEX <= 0;
        end
```

```
    end

    // FORWARD!
    always@*
    begin
        if(Reg_rs_EX == DstReg_EXMEM && RegWrite_EXMEM == 1 && MemtoReg_EXMEM == 0) // R type or i
mmediate, EX-MEM
            ReadData1_EX = ALURes_EXMEM;
        else if(Reg_rt_EX == DstReg_EXMEM && RegWrite_EXMEM == 1 && MemtoReg_EXMEM == 0)
            ReadData2_EX = ALURes_EXMEM;
        else if(Reg_rs_EX == DstReg_MEMWB && RegWrite_MEMWB == 1 && MemtoReg_MEMWB == 0) // R type
 or immediate, MEM-WB
            ReadData1_EX = ALURes_MEMWB;
        else if(Reg_rt_EX == DstReg_MEMWB && RegWrite_MEMWB == 1 && MemtoReg_MEMWB == 0)
            ReadData2_EX <= ALURes_MEMWB;
        else if(Reg_rs_EX == DstReg_MEMWB && RegWrite_MEMWB == 1 && MemtoReg_MEMWB == 1) //load, M
EM-WB
            ReadData1_EX = ReadData_MEMWB;
        else if(Reg_rt_EX == DstReg_MEMWB && RegWrite_MEMWB == 1 && MemtoReg_MEMWB == 1)
            ReadData2_EX = ReadData_MEMWB;
    end



    //EX STAGE
    wire [31:0] ALU_RES;

    assign INPUT1 = (isShift_IDEX == 0) ? ReadData1_EX : shamt_IDEX;
    assign INPUT2 = (ALUSrc_IDEX == 1)? Ext_data_IDEX : ReadData2_EX;

    ALU alu(
        .input1(INPUT1),
        .input2(INPUT2),
        .aluCtr(ALUCtrOut_IDEX),
        .zero(ZERO),
        .aluRes(ALU_RES)
    );

    always@ (posedge CLK)
    begin
        // Update of EXMEM registers
        MemRead_EXMEM <= MemRead_IDEX;
        MemWrite_EXMEM <= MemWrite_IDEX;
        MemtoReg_EXMEM <= MemtoReg_IDEX;
        RegWrite_EXMEM <= RegWrite_IDEX;
```

```
    ReadData2_EXMEM <= ReadData2_EX;
    ALURes_EXMEM <= ALU_RES;
    Zero_EXMEM <= ZERO;
    Jal_EXMEM <= Jal_IDEX;
    DstReg_EXMEM <= DstReg_IDEX;
    PC_EXMEM <= PC_IDEX;
end



//MEM STAGE
wire [31:0] READ_DATA;
dataMemory dataMemory(
    .Clk(CLK),
    .address(ALURes_EXMEM),
    .writeData(ReadData2_EXMEM),
    .memWrite(MemWrite_EXMEM),
    .memRead(MemRead_EXMEM),
    .readData(READ_DATA)
);

always@ (posedge CLK)
begin
    // Update of MEMWB registers
    MemtoReg_MEMWB <= MemtoReg_EXMEM;
    RegWrite_MEMWB <= RegWrite_EXMEM;
    ALURes_MEMWB <= ALURes_EXMEM;
    ReadData_MEMWB <= READ_DATA;
    DstReg_MEMWB <= DstReg_EXMEM;
    Jal_MEMWB <= Jal_EXMEM;
    PC_MEMWB <= PC_EXMEM;
end


//WB
assign WRITE_DATA_REG0 = MemtoReg_MEMWB? ReadData_MEMWB : ALURes_MEMWB;
assign WRITE_DATA_REG = Jal_MEMWB? PC_MEMWB : WRITE_DATA_REG0;  //jal
assign REG_WRITE = RegWrite_MEMWB;
assign WRITE_REG = Jal_MEMWB? 31 : DstReg_MEMWB;                //jal


always @ (CLK)
begin
    if(Reset)
    begin
```

```
            //If Reset
            PC <= 0;
            Inst_ID <= 0;
            INST <= 0;
            MemRead_IDEX <= 0; MemWrite_IDEX <= 0;
            MemtoReg_IDEX <= 0; RegWrite_IDEX <= 0; ALUSrc_IDEX <= 0;
            Ext_data_IDEX <= 0;
            DstReg_IDEX <= 0;
            shamt_IDEX <= 0;
            Reg_rs_EX <= 0;
            Reg_rt_EX <= 0;
            Jal_IDEX <= 0;

            MemRead_EXMEM <= 0; MemWrite_EXMEM <= 0; MemtoReg_EXMEM <= 0;
            RegWrite_EXMEM <= 0; Jal_EXMEM <= 0;

            MemtoReg_MEMWB <= 0; RegWrite_MEMWB <= 0; Jal_MEMWB <= 0;

            ReadData_MEMWB <= 0; ALURes_MEMWB <= 0; DstReg_MEMWB <= 0;
        end
    end

endmodule
```

IsShift.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2020/07/01 11:04:22
// Design Name:
// Module Name: isShift
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
```

```
//////////////////////////////////////////////////////////////////////////////////


module IsShift(
    input [5:0] opcode,
    input [5:0] funct,
    output reg shift
    );

    // whether it is a shift instruction and needs shamt
    always @ (funct or opcode) begin
        if(opcode == 5'b00000)
        begin
        case (funct)
            6'h02: shift = 1;
            6'h03: shift = 1;
            6'h00: shift = 1;
            default: shift = 0;
        endcase
        end
        else shift = 0;
    end

endmodule
```

NeedFlush.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2020/07/01 20:57:26
// Design Name:
// Module Name: NeedFlush
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
```

```
//
//////////////////////////////////////////////////////////////////////////////////


module NeedFlush(
    input [5:0] opcode,
    input [5:0] funct,
    input zero,
    output flush
    );

    reg Flush;

    always@ (opcode or zero or funct)
    begin
        if(opcode == 6'b000010 || opcode == 6'b000011) //jump, jal
            Flush = 1;
        else if(opcode == 6'b000000 && funct == 6'b001000)  //jr
            Flush = 1;
        else if(opcode == 6'b000100 && zero == 1)   //beq
            Flush = 1;
        else if(opcode == 6'b000101 && zero == 0)   //bne
            Flush = 1;
        else Flush = 0;
    end

    assign flush = Flush;
endmodule
```

Stall_detect.v:

```
module Stall_detect(
        input CLK,
        input [5:0] OpCode,
        input [5:0] funct,
        input [4:0] readReg1,
        input [4:0] readReg2,
        input needWrite,
        input isLoad,
        input [4:0] LoadReg,
        output Stall
    );

    reg stall;
```

```
    always@ (OpCode or readReg1 or readReg2 or isLoad or LoadReg or needWrite)
    begin
        if(isLoad == 1)
        begin
            if(OpCode == 6'b000000)
            // current is R type, and previous is LW
            begin
                if(readReg1 == LoadReg || readReg2 == LoadReg)
                    stall = 1;
            end
            else if(OpCode != 6'b000010)
            // all other instruction except jump type
            begin
                if(readReg1 == LoadReg)
                    stall = 1;
            end
        end
        else if(OpCode == 6'b000000 && funct == 6'b001000 && needWrite)
        //jr, need to get register result at ID stage
        begin
            if(readReg1 == LoadReg)
                    stall = 1;
        end
        else if((OpCode == 6'b000100 || OpCode == 6'b000101) && needWrite)
        //branch, need to get register result at ID stage
        begin
            if(readReg1 == LoadReg)
                    stall = 1;
            else if(readReg2 == LoadReg)
                    stall = 1;
        end
        else stall = 0;

        if(needWrite != 1) stall = 0;
    end

    assign Stall = stall;

endmodule
```

NeedFlush.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
```

```
// Company:
// Engineer:
//
// Create Date: 2020/07/01 20:57:26
// Design Name:
// Module Name: NeedFlush
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module NeedFlush(
    input [5:0] opcode,
    input [5:0] funct,
    input zero,
    output flush
    );

    reg Flush;

    always@ (opcode or zero or funct)
    begin
        if(opcode == 6'b000010 || opcode == 6'b000011) //jump, jal
            Flush = 1;
        else if(opcode == 6'b000000 && funct == 6'b001000)  //jr
            Flush = 1;
        else if(opcode == 6'b000100 && zero == 1)   //beq
            Flush = 1;
        else if(opcode == 6'b000101 && zero == 0)   //bne
            Flush = 1;
        else Flush = 0;
    end

    assign flush = Flush;
endmodule
```

测试使用的代码如下：

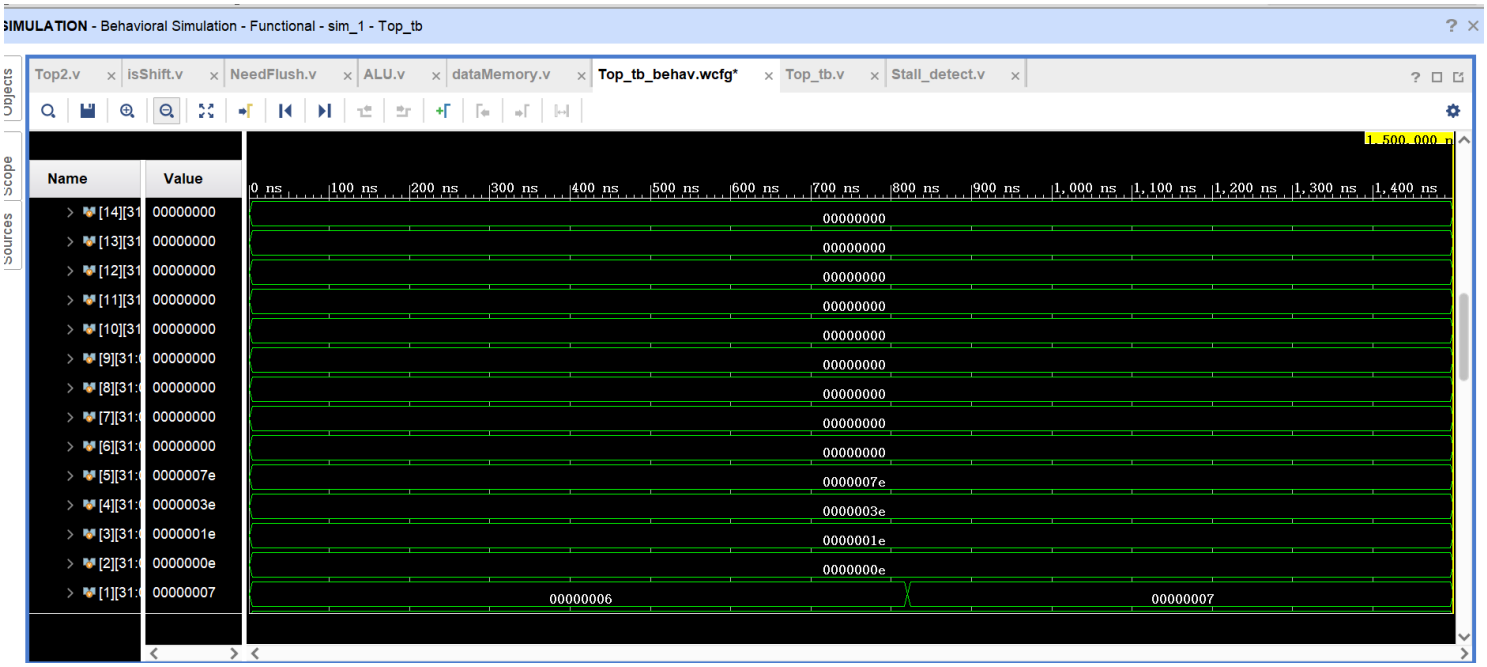
C: > Users > biany > Desktop > Lab05工程文件 > tmp.v

```
1   000010_00000_00000_00000_00000_000010 //j 2
2   100011_00001_00011_00000_00000_000011 //lw $3,3($1)         (flush)
3   000011_00000_00000_00000_00000_000100 //jal 4
4   100011_00001_00011_00000_00000_000011 //lw $3,3($1)         (flush)
5   001000_00000_00001_00000_00000_000111 //addi $1, $0, 7      $1  = 7
6   001000_00000_10000_00000_00000_100000 //addi $16, $0, 32    $1  = 7
7   000000_10000_00000_00000_00000_001000 //jr $16
8   100011_00001_00011_00000_00000_000011 //lw $3,3($1)         (flush)
9   000000_00001_00010_00100_00000_100000 //add $4, $1, $2      $4 = 7 + 7 = e
10  000000_00001_00100_00011_00000_100000 //add $3, $1, $4      (forward) $3 = 7 + e  = 15
11  000000_00011_00001_00101_00000_100010 //sub $5, $3, $1      $5 = e
12  000000_00010_00001_00110_00000_100100 //and $6, $2, $1      $6 = 7
13  000000_00010_00010_00111_00000_100101 //or $7, $2, $2       $7 = 7
14  000000_00000_00001_00001_00000_101010 //slt $1, $0, $1      $1 = 1
15  000000_00000_00011_01000_00010_000000 //sll $8, $3, 2       $8 = 54
16  000000_00000_00011_01001_00010_000010 //srl $9, $3, 2       $9 = 5
17  101011_00000_00001_00000_00000_000001 //sw $1, $0, 1        mem 1 = 1
18  100011_00000_00001_00000_00000_000010 //lw $1, $0, 2        $1 = e
19  100011_00000_00010_00000_00000_000011 //lw $2, $0, 3        $2 = 1e
20  000000_00010_00100_00011_00000_100000 //add $3, $2, $4      (stall) $3 = 2c
21  000100_00000_00001_00000_00000_000001 //beq $0, $1, 1       next instruction
22  001000_00000_00001_00000_00000_000000 //addi $1, $0, 0      $1 = 0
23  000100_00000_00001_00000_00000_000001 //beq $0, $1, 1       $jump forward
24  001000_00000_00001_00000_00000_000010 //addi $1, $0, 2      $1 = 2
25  001000_00000_00001_00000_00000_000001 //addi $1, $0, 1      $1 = 1
```

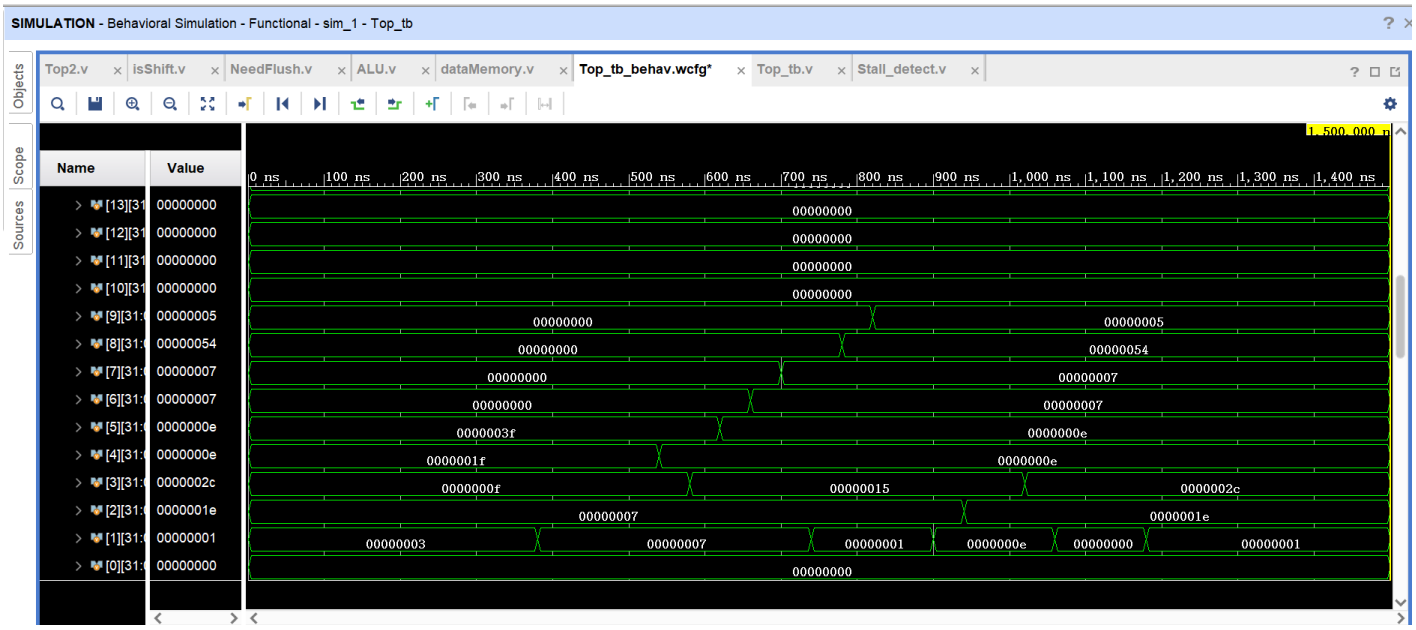
三列分别为：二进制指令、其对应的 MIPS 指令、CPU 正确工作时应该得到的结果

下面是仿真结果。可以看出，和计算结果完全一样：

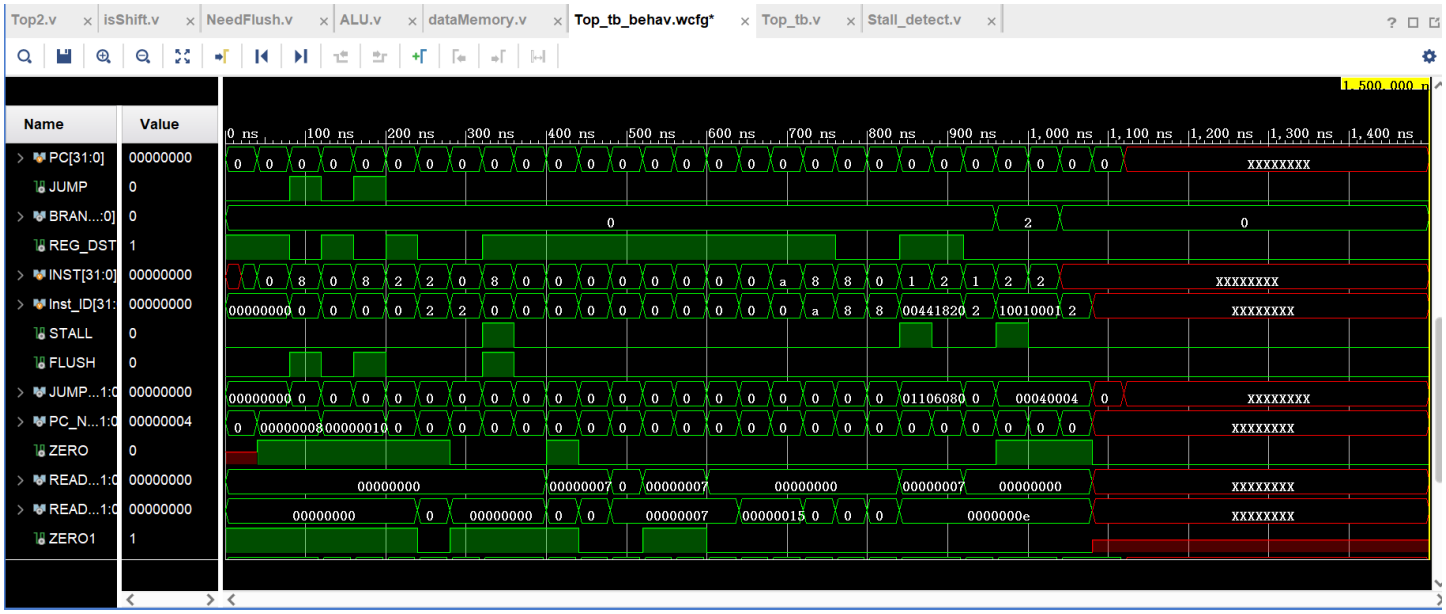
Memory:



Registers:



其它变量:



3. 实验心得与总结

在本实验中，我在普通 pipeline 的基础上加入了 stall 机制，增加 Forwarding 机制解决数据竞争，通过 predict-not-taken 或延时转移策略解决控制冒险/竞争，并将支持的指令进行了扩充（支持 30 个指令左右,由于时间限制 lui, sra, srav 的实现没有来得及写完）。

此实验比上一个实验难度增加了很多。

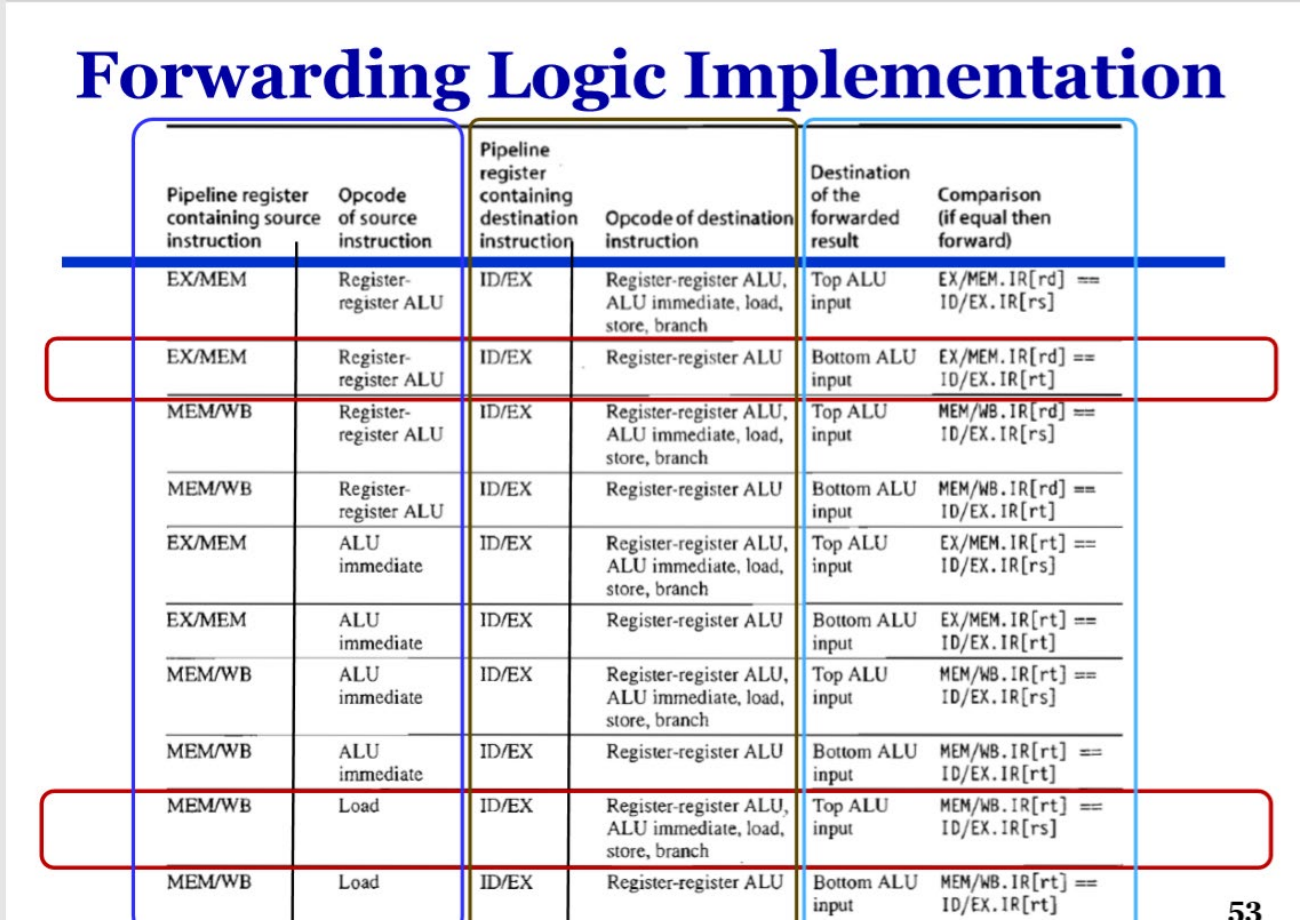
我起初将 IF, ID, EX, MEM, WB 五个部分分成了不同的 module，认为这样可以简化逻辑，并且减少命名冲突。然而，在实际中，我发现了一些问题：首先，pipeline 的阶段的划分与硬件的划分并不完全一致。举例来说，在 ID 和 WB 两个阶段都会用到 Register 模块；然而将 ID 和 WB 分成不同模块之后两者很难同时使用某一硬件，在编程时需要一定技巧，且复杂度会升高。其次，不同的阶段间有很多联系（需要更新很多的 pipeline register），这导致每个 pipeline 模块的 input 和 output 都及其冗长，甚至需要有 20 个变量以上。这不仅使得逻辑更加复杂，也造成了许多本无太大必要的 assign 语句。经过权衡，我最终决定还是将所有的 stage 放在同一模块中。

Stall 和 predict 错误后的 flush 机制（即在 Branch 预测错误的情况下将上一条指令“删去”）是本实验中最为困难的两个部分。首先，我们需要找出所有需要 stall 和 flush 的情况，做到不遗漏。此外，我们需要在 stall 和 flush 指令结束后回到正常的执行状态，而不引发其它冲突；即使处在特殊情况，出现多个连续的 stall 或是 flush，我们的代码也不能引发错误。

最后，由于我新增了部分指令，新增的指令中有的需要额外添加部分逻辑，因此需要确保原本的 stall 和 flush 模块与新添的指令相兼容。

在编程中，有一个很难理解的点是在 flush 和 stall 之后 PC 的变化情况。由于 flush 与 stall 都会使得部分指令“失效”，因此我们需要保证“失效”的指令无法修改 PC 值；但是其它不失效的指令都必须能按正确的逻辑修改 PC。因此，我们需要在一定程度上“记住”那条指令已经“失效”。此时，画出指令的流程图对我起到了很大的帮助。我认为，我最终的方案应该做到了这一点（当然也可能仍有一些我没有思考到的情况）。

而相对而言，forward 反而比较容易实现。下图展示了 forward 所需要的逻辑结构：



Forwarding Logic Implementation

Pipeline register containing source instruction	Opcode of source instruction	Pipeline register containing destination instruction	Opcode of destination instruction	Destination of the forwarded result	Comparison (if equal then forward)
EX/MEM	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR[rd] == ID/EX.IR[rs]
EX/MEM	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR[rd] == ID/EX.IR[rt]
MEM/WB	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR[rd] == ID/EX.IR[rs]
MEM/WB	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR[rd] == ID/EX.IR[rt]
EX/MEM	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR[rt] == ID/EX.IR[rs]
EX/MEM	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR[rt] == ID/EX.IR[rt]
MEM/WB	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR[rt] == ID/EX.IR[rs]
MEM/WB	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR[rt] == ID/EX.IR[rt]
MEM/WB	Load	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR[rt] == ID/EX.IR[rs]
MEM/WB	Load	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR[rt] == ID/EX.IR[rt]

53

此外，在设计时，对于 jump 相关指令（j, jal, jr）是在 IF 阶段便得到解析并更新 PC,还是需要在 ID 阶段才可以更新 PC，不同资料似乎说法并不一致。为了保持代码间的一致性，最终我选择在 ID 阶段更新 PC。这样，在每次跳转前都会出现一次 flush 操作。这样似乎增加了指令执行需要的总时间。若是将跳转类指令放在 IF 阶段，节约了时间，然而在实际中，我很怀疑是否可以在一个 clock cycle 内完成读取指令与解析指令这两项操作，因为内存的读取本身就十分耗费时间。

总而言之，这几次的实验让我入门了 verilog 的硬件相关编程，让我从一个原先的小白，逐渐掌握了编程所需要的技巧；也让我对 CPU 的逻辑有了远比以前深入的多的了解，很有收

获。

4、参考资料

2020 计算机系统结构实验指导书-LAB06_M