计算机系统结构试验

Lab06: 简单的类 MIPS 多周期流水化处理器的实现

姓名：N/A

摘要

在 Lab06 中，我实现了 MIPS 多周期流水化处理器的主要功能，使用 Forwarding 和 Stall 机制解决了 Data Hazard 和 Control Hazard，使用分支预测减少 Control Hazard，并在此基础之上将指令集扩展到了 31 条。实验包括了 Instruction Fetch、Instruction Decode、Execution、Memory 和 Buffers 等模块的设计和实现，以及修改 Lab5 中的其余模块。通过本次实验，我进一步学习了对 Verilog 语言的运用，掌握 MIPS 流水线处理器的组成、设计和实现方法，并且深入理解了流水线 Stall 和 Forwarding 机制，给我带来宝贵的经验和收获。

目录

摘要 1
1. 实验目的 3
2. 原理分析 3
2.1 Vivado 工程的基本组成 3
2.2 流水化处理器的原理 3
2.2.1 指令流水线 3
2.2.2 流水线寄存器 4
2.2.3 时序控制 4
2.3 控制冒险的处理 4
2.3.1 Jump 指令控制冒险 4
2.3.2 Branch 指令控制冒险 5
2.4 数据冒险的处理 5
2.4.1 寄存器 Load-Use 冒险 5
2.4.2 内存 Load-Use 冒险 6
2.5 分支预测的原理 6
3. 功能实现 6
3.1 流水线处理器的实现 6
3.1.1 IF 阶段 6
3.1.2 ID/WB 阶段 7
3.1.3 EX 阶段 8
3.1.4 MEM 阶段 9
3.1.5 流水线寄存器 9
3.2 消除控制冒险的实现 10
3.3 消除数据冒险的实现 10
3.3.1 消除寄存器 Load-Use 冒险 10

3.3.2 消除内存 Load-Use 冒险 11
3.4 分支预测的实现 11
4. 结果验证 12
4.1 提供的测试代码 12
4.2 冒险与分支预测的测试 13
4.3 其他指令的测试 14
4.3.1 jr，j 和 jal 的测试 14
4.3.2 sll，srl 和 sra 的测试 14
4.3.3 addi，addiu 和 lui 的测试 15
4.3.4 slti 和 sltiu 的测试 15
4.5 综合测试 15
5. 总结与反思 16

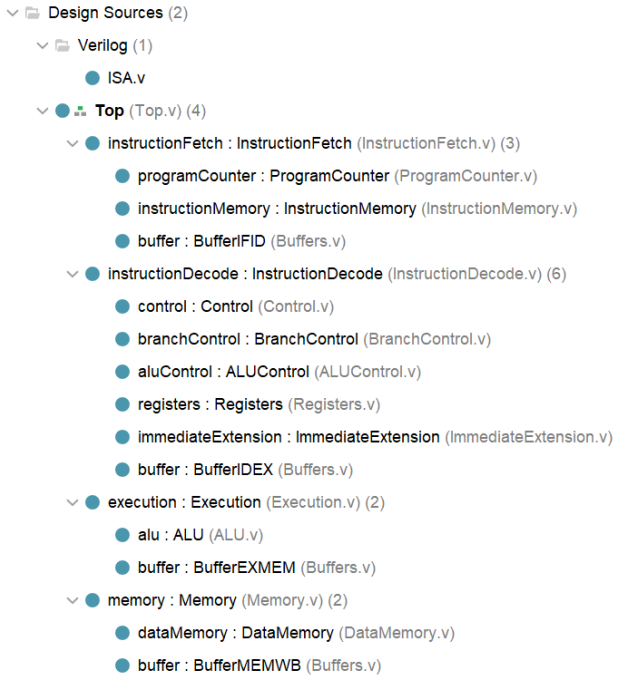
1. 实验目的

（1）理解 CPU Pipeline、流水线冒险(hazard)及相关性，在 Lab5 基础上设计简单流水线 CPU；
（2）在 1 的基础上设计支持 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/竞争、控制冒险和结构冒险；
（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；
（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）在 4 的基础上，将 CPU 支持的指令数量从 9 条或 16 条扩充为 31 条，使处理器功能更加丰富。

2. 原理分析

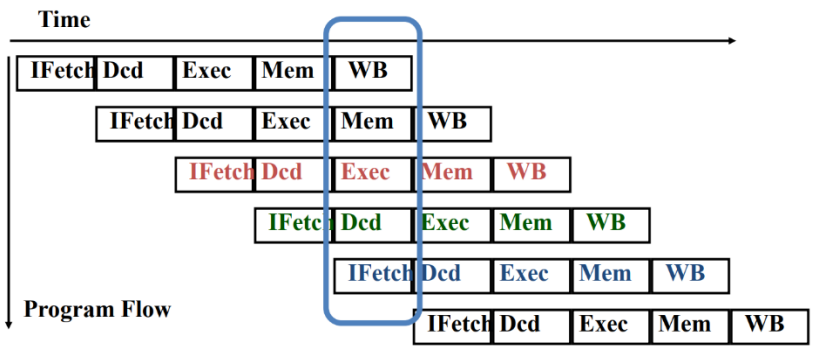
2.1 Vivado 工程的基本组成

Vivado 工程的结构如下：



2.2 流水化处理器的原理

2.2.1 指令流水线



MIPS 多周期流水线处理器将指令的执行分成 5 个阶段：

（1）取指阶段(IF)：从指令存储器中取出指令。

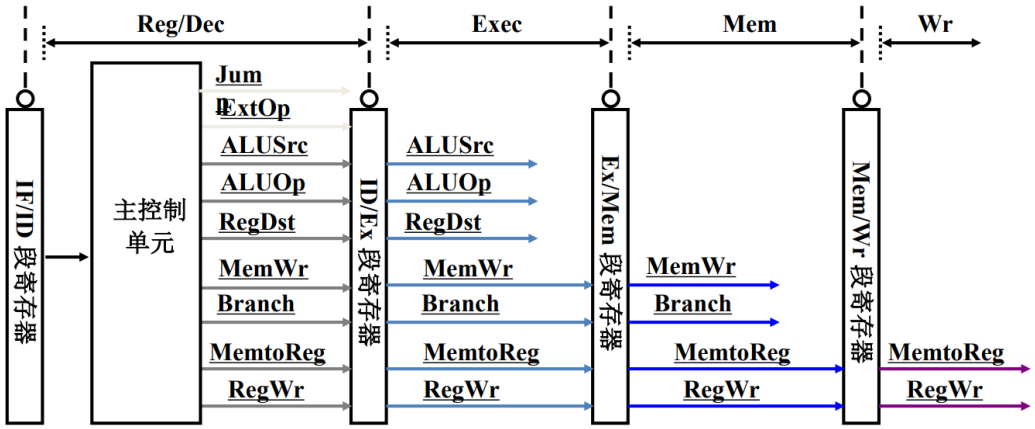
（2）译码阶段(ID)：指令译码、寄存器读取，生成控制信号。

（3）执行阶段(EX)：读取操作数，进行算术或逻辑运算，或计算内存地址。

（4）访存阶段(MEM)：读取内存地址，执行内存读/写操作。

（5）写回阶段(WB)：读取计算结果或内存数据，将结果写回寄存器文件。

2.2.2 流水线寄存器



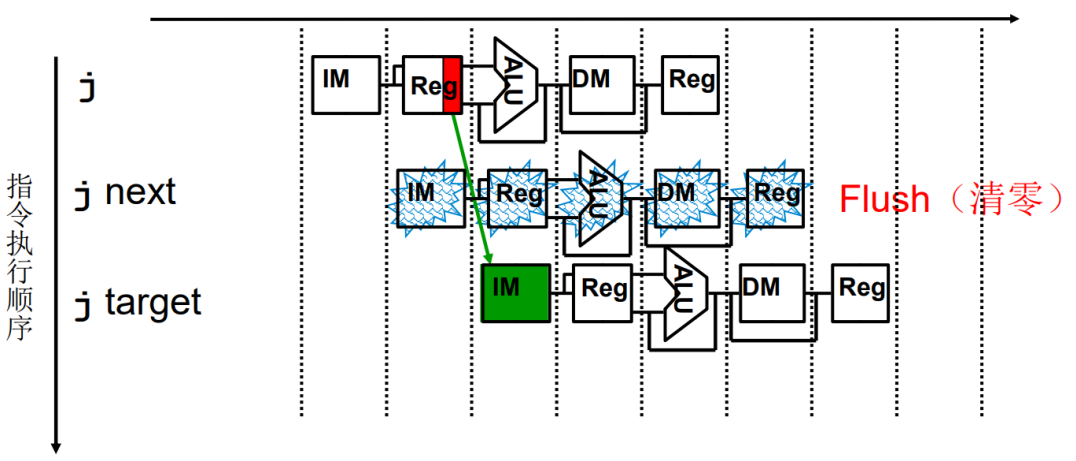
为了在各个阶段之间传递指令信息，在每个阶段之间引入流水线寄存器，用来存储指令的某些部分、控制信号和中间结果，确保在不同的周期里各阶段之间的数据信息可以正确传递。图中展示了四个关键的流水线寄存器：IF/ID, ID/EX, EX/MEM 和 MEM/WB 中保存的控制信号。

2.2.3 时序控制

在 5 个流水线阶段中，IF、ID 和 MEM 阶段需要注意时序问题。Program Counter 和各 buffer 应该在同一时钟沿更新，保证每个流水阶段执行的指令在同一时刻更新到下一阶段，本次实验中设置为时钟下降沿更新。为了防止读写的 Hazard 问题，Registers 和 Data Memory 的写操作为时钟上升沿更新。

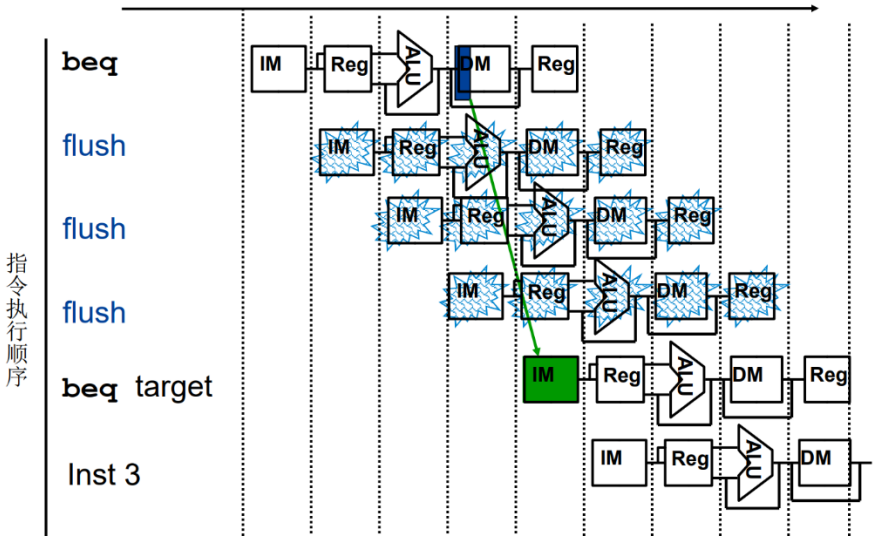
2.3 控制冒险的处理

2.3.1 Jump 指令控制冒险



在 Jump 类指令（jr，j，jal）中，指令在 ID 阶段译码完成并得到跳转地址，然而此时 IF 阶段已经得到了顺序执行下一条指令的内容并保存在 IF/ID 寄存器内。因此，在将 jump 信号和目标发给 IF 阶段的同时，还要清空 IF/ID 寄存器中错误的指令。此时 ID/EXE 寄存器中还需保留有 jal 指令的写回操作，无需清空 ID/EXE 寄存器。

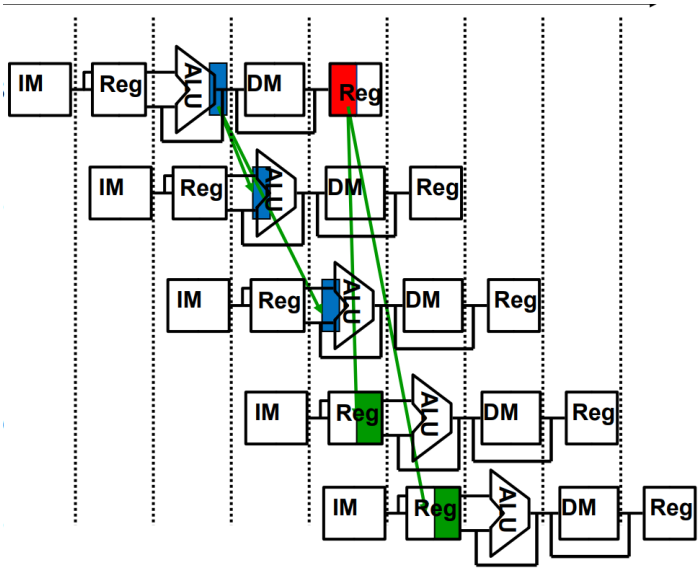
2.3.2 Branch 指令控制冒险



在 Branch 类指令（beq，bne）中，指令在 EX 阶段计算出 zero 值并判定是否跳转（实际判定了分支预测是否错误），此时 IF、ID 阶段可能已经获得了错误的指令，需要清空 ID/IF 和 IF/EX 两个寄存器。此时 EX/MEM 寄存器保存了 Branch 指令，在 MEM 和 WB 阶段没有操作，因此清空与否都可以。

2.4 数据冒险的处理

2.4.1 寄存器 Load-Use 冒险



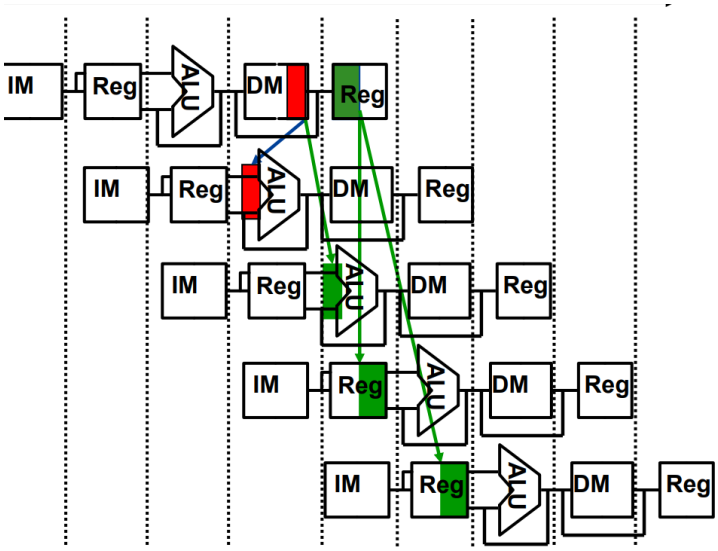
在寄存器 Load-Use 冒险中共有三种情况：

（1） EX 阶段用到前一条指令的结果：在 EX 模块内部，设置一个 Forwarding 单元将 EX/MEM 寄存器中的数据传给 ALU 的输入。

（2） EX 阶段用到前两条指令的结果：设置一个 Forwarding 单元，将此时的 MEM/WB 寄存器中的数据传给 ALU 的输入。

（3） EX 阶段用到前三条指令的结果：在上一个时钟周期，前三条指令正在执行 WB 阶段写寄存器，该条指令正在执行 ID 阶段读寄存器。在 Registers 模块内部设置一个 Forwarding 单元将写入数据传给读出数据。

2.4.2 内存 Load-Use 冒险



在内存 Load-Use 冒险中，前一条指令的 MEM 阶段和此条指令的 ALU 阶段同时发生。如果做 Forwarding，那么流水线延迟将变为 MEM 和 EX 的延迟之和，会严重降低 CPU 的频率。因此这里采用 Stall 机制实现，当出现内存 Load-Use 冒险时，IF、ID 和 EX 阶段以及相应的寄存器将会暂停一个周期，等待前一条指令进入 WB 阶段，并通过 Registers 的 Forwarding 机制传给 EX 阶段。

2.5 分支预测的原理

在 IF 阶段可以保存一个哈希表，记录指令上次跳转情况，并预测 Branch 指令的行为与上次一致。为了实现提前跳转，IF 阶段需要提前译码指令，跳转到相应位置。在 EX 阶段通过实际计算判断预测是否正确，若错误则 flush 取到的错误指令，并通知 IF 阶段修改哈希表中的预测值。

3. 功能实现

3.1 流水线处理器的实现

3.1.1 IF 阶段

Instruction Fetch 模块的部分代码如下：

```
ProgramCounter programCounter (
    .clk  (clk),
    .reset(reset),
    .stall(i_stall),

    .jump    (i_jump),
    .jumpAddr(i_jumpAddr),

    .branch    (i_branch),
    .branchAddr(i_branchAddr),

    .predict    (branchPredictionTable[PC[6:3]]),
    .instruction(instruction),

    .programCounter(PC)
);

InstructionMemory instructionMemory (
    .clk(clk),

    .address(PC),

    .instruction(instruction)
);

BufferIFID buffer (
    .clk  (clk),
    .reset(reset || flush),
    .stall(i_stall),

    .instruction(instruction),
    .PC         (PC),

    .branchPredict(branchPredictionTable[PC[6:3]]),

    .instructionOut(o_instruction),
    .PCOut         (o_PC),

    .branchPredictOut(o_branchPredict)
);
```

该阶段包括了 Program Counter 和 Instruction Fetch 模块，并保存了分支预测的哈希表。输出信号包括 PC 值、指令和分支预测信号。

3.1.2 ID/WB 阶段

Instruction Decode 模块部分代码如下：

```
wire [4:0] WB_writeReg = i_WB_jal ? 31 : i_WB_writeReg;
wire [31:0] WB_writeData = i_WB_jal ? i_WB_PC + 4 : (i_WB_memToReg ? i_WB_readMemData : i_WB_ALUResult);

wire jump = jr || j || jal;
wire [31:0] jumpAddr = jr ? readRegData1 : (i_instruction[25:0] << 2) + ((i_PC + 4) & 32'hf0000000);
```

```
Registers registers (
    .clk(clk),

    .readReg1(i_instruction[25:21]),
    .readReg2(i_instruction[20:16]),

    .writeReg (WB_writeReg),
    .writeData(WB_writeData),
    .regWrite (i_WB_regWrite),

    .readData1(readRegData1),
    .readData2(readRegData2)
);

ImmediateExtension immediateExtension (
    .source(i_instruction[15:0]),
    .opCode(i_instruction[31:26]),

    .extension(immediate)
);
```

```
Control control (
    .opCode(i_instruction[31:26]),

    .regDst  (regDst),
    .ALUSrc  (ALUSrc),
    .memToReg(memToReg),
    .regWrite(regWrite),
    .memRead (memRead),
    .memWrite(memWrite),
    .ALUOp   (ALUOp)
);

BranchControl branchControl (
    .opCode(i_instruction[31:26]),
    .funct (i_instruction[5:0]),

    .beq(beq),
    .bne(bne),
    .jr (jr),
    .j  (j),
    .jal(jal)
);

ALUControl aluControl (
    .ALUOp(ALUOp),
    .funct(i_instruction[5:0]),

    .ALUCtr(ALUCtr)
);
```

该阶段包括了 Control、ALU Control、Branch Control、Registers 和 Immediate Extension 模块。此外，Jump 类指令的跳转信息由此处发给 IF 阶段。WB 阶段也在该模块实现。

3.1.3 EX 阶段

Execution 的部分代码如下：

```
wire innerForward1 = i_inputReg1 == o_writeReg && o_writeReg != 0;
wire innerForward2 = i_inputReg2 == o_writeReg && o_writeReg != 0;

wire memForward1 = i_inputReg1 == i_forwardWriteReg && i_forwardWriteReg != 0;
wire memForward2 = i_inputReg2 == i_forwardWriteReg && i_forwardWriteReg != 0;

wire isBranch = i_beq || i_bne;
wire branchTaken = (i_beq && zero) || (i_bne && !zero);
wire [31:0] branchAddr = i_PC + 4 + (i_immediate << 2);
wire predictCorrect = !isBranch || (i_branchPredict && branchTaken) || (!i_branchPredict && !branchTaken);
```

```
ALU alu (
    .input1(innerForward1 ? o_result : (memForward1 ? i_forwardData : i_input1)),
    .input2(innerForward2 ? o_result : (memForward2 ? i_forwardData : i_input2)),
    .shamt (i_shamt),
    .ALUCtr(i_ALUCtr),

    .zero  (zero),
    .result(result)
);
```

Execution 阶段只有 ALU 模块。在该阶段中，处理了 Forwarding 的相关机制和 Branch 的判定。

3.1.4 MEM 阶段

Memory 模块的部分代码如下：

```
assign o_forwardData     = o_memToReg ? o_readMemData : o_ALUResult;
assign o_forwardWriteReg = o_regWrite ? o_writeReg : 0;

DataMemory dataMemory (
    .clk(clk),

    .address  (i_address),
    .writeData(i_writeData),
    .memRead  (i_memRead),
    .memWrite (i_memWrite),

    .readData(readData)
);
```

Memory 阶段也只有 Data Memory 模块。这里处理了由 MEM 向 ALU 发送的 Forwarding 数据，并将 WB 相关信号发给 ID 阶段。

3.1.5 流水线寄存器

流水线寄存器将输入的值在时钟下降沿更新到输出寄存器，行为类似 D 触发器。以 IF/ID 阶段为例：

```
module BufferIFID (
    input wire clk,
    input wire reset,
    input wire stall,

    input wire [31:0] instruction,
    input wire [31:0] PC,

    input wire branchPredict,

    output reg [31:0] instructionOut,
    output reg [31:0] PCOut,

    output reg branchPredictOut
);

  always @(negedge clk, posedge reset) begin
    if (reset) begin
      instructionOut   = {32{1'b1}};
      PCOut            = {32{1'b1}};

      branchPredictOut = 1'b0;
    end else if (!stall) begin
      instructionOut   = instruction;
      PCOut            = PC;

      branchPredictOut = branchPredict;
    end
  end

endmodule
```

这里接收时钟信号和 reset、stall 两个控制信号。在 reset 时清空当前 buffer 内容，在 stall 时不做操作，否则在时钟下降沿将输出值更新。

3.2 消除控制冒险的实现

Jump 指令在 ID 阶段的译码如下：

```
wire jump = jr || j || jal;
wire [31:0] jumpAddr = jr ? readRegData1 : (i_instruction[25:0] << 2) + ((i_PC + 4) & 32'hf0000000);
```

Branch 指令在 EX 阶段的判定如下：

```
wire isBranch = i_beq || i_bne;
wire branchTaken = (i_beq && zero) || (i_bne && !zero);
wire [31:0] branchAddr = i_PC + 4 + (i_immediate << 2);
wire predictCorrect = !isBranch || (i_branchPredict && branchTaken) || (!i_branchPredict && !branchTaken);
```

在发生 Jump/Branch 时，清空相应流水线寄存器：

```
InstructionFetch instructionFetch (
    .clk  (clk),
    .reset(reset),
    .flush(ID_jump || !EX_predictCorrect),
```

```
InstructionDecode instructionDecode (
    .clk  (clk),
    .reset(reset),
    .flush(!EX_predictCorrect),
```

在 Program Counter 内根据输入信号跳转：

```
always @(negedge clk, posedge reset, posedge jump, posedge branch) begin
  if (reset) begin
    programCounter = 32'b0;
  end else if (branch) begin
    programCounter = branchAddr;
  end else if (jump) begin
    programCounter = jumpAddr;
  end else if (!stall) begin
    programCounter = isBranch && predict ? branchTarget : programCounter + 4;
  end
end
```

3.3 消除数据冒险的实现

3.3.1 消除寄存器 Load-Use 冒险

根据三种寄存器 Load-Use 数据冒险情况，需要实现机制如下：

（1）在 EX 模块内部，设置一个 Forwarding 单元将 EX/MEM 寄存器中的数据传给 ALU 的输入：

```
wire innerForward1 = i_inputReg1 == o_writeReg && o_writeReg != 0;
wire innerForward2 = i_inputReg2 == o_writeReg && o_writeReg != 0;
```

（2）设置一个 Forwarding 单元将 MEM/WB 寄存器中的数据传给 ALU 的输入：

在 ALU 内：

```
wire memForward1 = i_inputReg1 == i_forwardWriteReg && i_forwardWriteReg != 0;
wire memForward2 = i_inputReg2 == i_forwardWriteReg && i_forwardWriteReg != 0;
```

在 MEM 阶段：

```
assign o_forwardData     = o_memToReg ? o_readMemData : o_ALUResult;
assign o_forwardWriteReg = o_regWrite ? o_writeReg : 0;
```

（3）在 Registers 模块内部设置一个 Forwarding 单元将写入数据传给读出数据：

```
always @(*) begin
  readData1 = regWrite && writeReg == readReg1 ? writeData : regFile[readReg1];
  readData2 = regWrite && writeReg == readReg2 ? writeData : regFile[readReg2];
end
```

最后在 ALU 内选择输入：

```
ALU alu (
    .input1(innerForward1 ? o_result : (memForward1 ? i_forwardData : i_input1)),
    .input2(innerForward2 ? o_result : (memForward2 ? i_forwardData : i_input2)),
    .shamt (i_shamt),
    .ALUCtr(i_ALUCtr),

    .zero  (zero),
    .result(result)
);
```

此外在传递用作写入内存的寄存器的读取值时，也做了相同原理的 Forwarding 机制，此处省略。

3.3.2 消除内存 Load-Use 冒险

在内存的 Load-Use 冒险中，实际上没有数据被 Forward，只是 IF 和 ID 阶段发生了 stall，stall 的检测如下：

```
// Load-Use Hazard
wire MEM_stall = EX_memRead && (EX_writeReg == ID_instruction[20:16] || EX_writeReg == ID_instruction[15:11] && ID_ALUSrc == 0);
```

3.4 分支预测的实现

在 Instruction Fetch 阶段，需要提前译码当前指令，根据 Branch Table 预测本次跳转是否发生，如果满足条件直接跳转，并且发出 predict 信号为真给 EX 阶段。这里哈希值取了指令的 6-3 位，哈希表大小为 16。

```
.predict     (branchPredictionTable[PC[6:3]]),
.instruction(instruction),
```

```
// Decode current instruction for branch prediction
wire isBranch = instruction[31:26] == `OP_BEQ || instruction[31:26] == `OP_BNE;
wire [16:0] branchImm = programCounter + 4 + (instruction[15:0] << 2);
wire [31:0] branchTarget = {{16{branchImm[15]}}, branchImm};

always @(negedge clk, posedge reset, posedge jump, posedge branch) begin
  if (reset) begin
    programCounter = 32'b0;
  end else if (branch) begin
    programCounter = branchAddr;
  end else if (jump) begin
    programCounter = jumpAddr;
  end else if (!stall) begin
    programCounter = isBranch && predict ? branchTarget : programCounter + 4;
  end
end
```

在 EX 阶段，计算出实际跳转是否发生，并输出 predict correct 信号。如果信号为 false，那么 IF/ID 和 ID/EX 寄存器将被清空。同时 IF 阶段会相应修改 Branch Table 的对应位。

```
wire isBranch = i_beq || i_bne;
wire branchTaken = (i_beq && zero) || (i_bne && !zero);
wire [31:0] branchAddr = i_PC + 4 + (i_immediate << 2);
wire predictCorrect = !isBranch || (i_branchPredict && branchTaken) || (!i_branchPredict && !branchTaken);
```

```
reg  [15:0] branchPredictionTable;

always @(posedge clk) begin
  if (reset) begin
    branchPredictionTable = 16'b0;
  end else if (!i_predictCorrect) begin
    branchPredictionTable[i_predictPC[6:3]] <= !branchPredictionTable[i_predictPC[6:3]];
  end
end
```

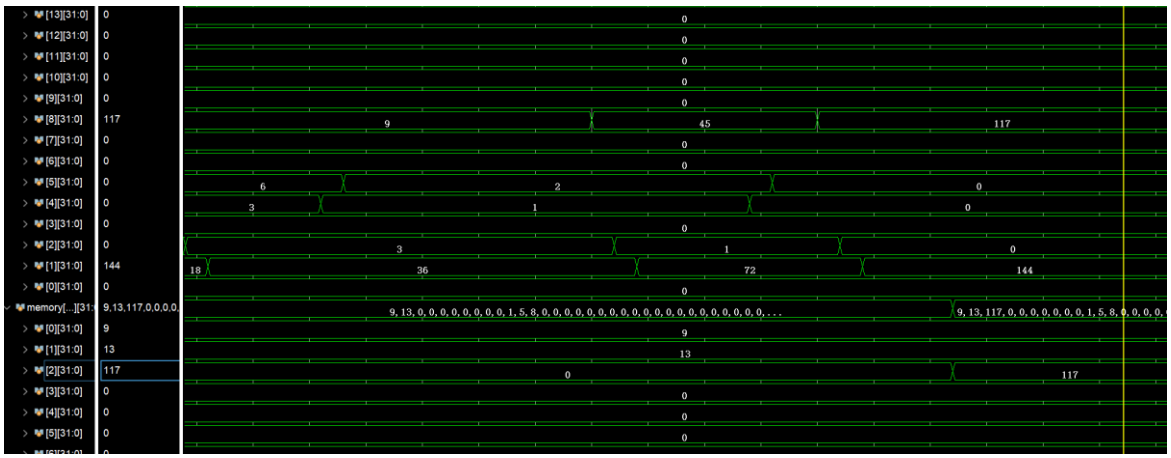
4. 结果验证

4.1 提供的测试代码

在乘法器测试中，编写 Instruction.txt 文件如下：

```
// Test program for multiplication
100011_00011_00001_0000000000000000      // lw  $1, 0($3)
100011_00011_00010_0000000000000100      // lw  $2, 4($3)
000000_00000_00010_00100_00001_000010    // srl $4, $2, 1
000000_00000_00100_00101_00001_000000    // sll $5, $4, 1
000100_00010_00101_0000000000000001      // beq $5, $2, 1
000000_01000_00001_01000_00000_100000    // add $8, $8, $1
000000_00000_00010_00010_00001_000010    // srl $2, $2, 1
000000_00000_00001_00001_00001_000000    // sll $1, $1, 1
000100_00011_00010_0000000000000001      // beq $2, $3, 1
000010_00000000000000000000000010        // j   2
101011_00011_01000_0000000000001000      // sw  $8, 8($0)
```

测试结果如图所示：

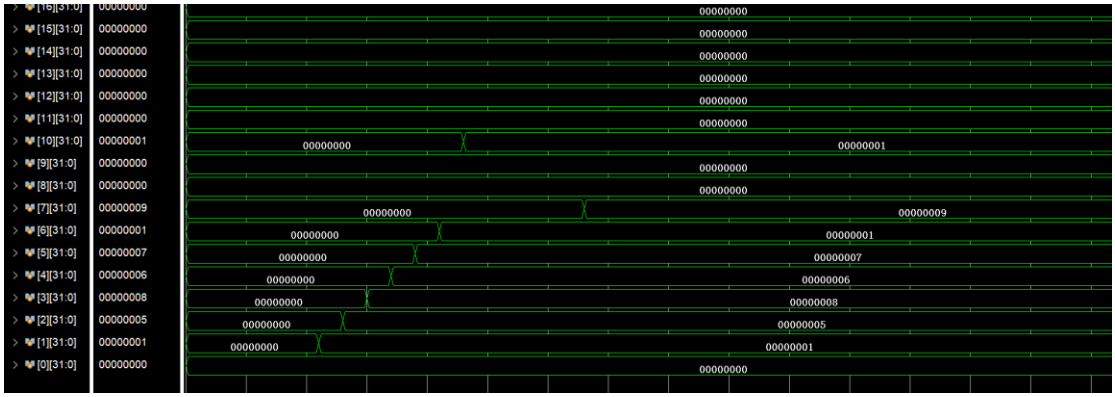


输出和计算过程符合预期。

在提供的测试代码中，编写 Instruction.txt 文件如下：

```
// Test program in the given handout
100011_00000_00001_0000000000101000      // lw  $1, 40($0)  ; $1  = 1
100011_00000_00010_0000000000101100      // lw  $2, 44($0)  ; $2  = 5
100011_00000_00011_0000000000110000      // lw  $3, 48($0)  ; $3  = 8
000000_00001_00010_00100_00000_100000    // add $4, $1, $2  ; $4  = 6
000000_00011_00001_00101_00000_100010    // sub $5, $3, $1  ; $5  = 7
000000_00010_00001_00110_00000_100100    // and $6, $2, $1  ; $6  = 1
100011_00000_01010_0000000000101000      // lw  $10, 40($0) ; $10 = 0
100011_00000_01010_0000000000101000      // lw  $10, 40($0) ; $10 = 0
100011_00000_01010_0000000000101000      // lw  $10, 40($0) ; $10 = 0
000000_00011_00001_00111_00000_100101    // or  $7, $3, $1  ; $7  = 9
000000_00011_00001_01000_00000_101010    // slt $8, $3, $1  ; $8  = 0
000100_00000_00000_0000000000000001      // beq $0, $0, 1   ; goto end
000000_00111_01000_01001_00000_100000    // add $9, $7, $8  ; $9  = 9 (not executed)
100011_00000_01010_0000000000101000      // lw  $10, 40($0) ; $10 = 0 (label end)
100011_00000_01010_0000000000101000      // lw  $10, 40($0) ; $10 = 0
100011_00000_01010_0000000000101000      // lw  $10, 40($0) ; $10 = 0
100011_00000_01010_0000000000101000      // lw  $10, 40($0) ; $10 = 0
100011_00000_01010_0000000000101000      // lw  $10, 40($0) ; $10 = 0
```

测试结果如图所示：



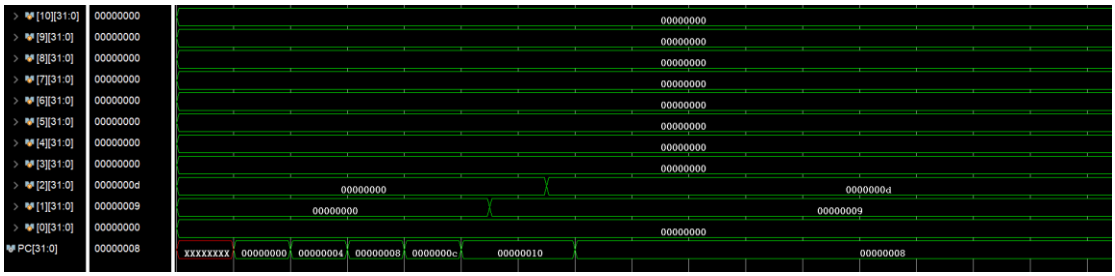
没有 Data Hazard 产生。

4.2 冒险与分支预测的测试

编写 Instruction 如下：

```
// Test program of hazards and branch prediction
100011_00011_00001_0000000000000000      // lw   $1, 0($3)
100011_00011_00010_0000000000000100      // lw   $2, 4($3)
000101_00001_00010_1111111111111111      // bne  $2, $1, -1
001000_00000_00011_0000000000000001      // addi $3, $0, 1
```

仿真结果如图所示



在测试中，出现了内存的 Load-Use 冒险，最终 bne 成功跳转，说明数据冒险已经消除。在 Branch 中，前面阶段的 PC 已经取到 0x0c 和 0x10，但因为相关流水线寄存器被清空，addi 指令没有执行成功，说明控制冒险已经消除。最后，由于 bne 指令一直跳转到自身，在第二次分支时分支预测判断会发生分支，因此 PC 一直保持 0x08，分支预测行为符合预期。

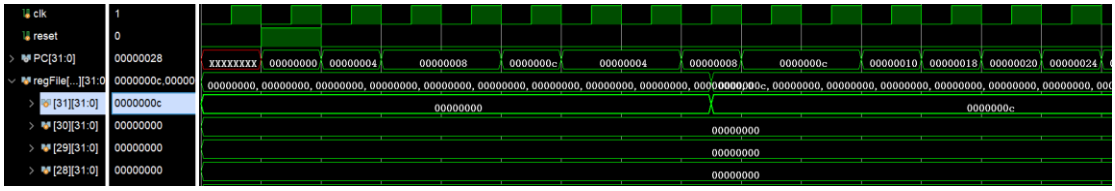
4.3 其他指令的测试

4.3.1 jr，j 和 jal 的测试

编写 Instruction 如下：

```
// Test of jr, j, jal
000010_00000000000000000000000010       // j   2
000000_11111_00000_00000_00000_001000   // jr  $31
000011_00000000000000000000000001       // jal 1
```

结果如图所示：



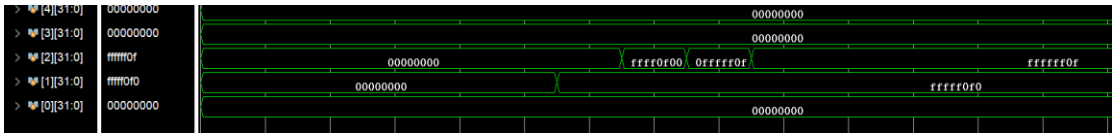
测试代码先使用 j 2 无条件跳转到 PC = 2 << 2 = 8 处，此时执行第三条指令 jal 1，将 PC + 4 = 0x0c 存入 ra 寄存器，并跳转到 PC = 1 << 2 = 4 处。接下来执行第二条指令 jr $31 将 PC 设为 ra 的值 0x0c。可以看到相应的跳转位置因为控制冒险导致了流水线停顿。

4.3.2 sll，srl 和 sra 的测试

编写 Instruction 如下：

```
// Test of sll, srl, sra
001000_00000_00001_1111000011110000     // addi $1, $0, 0xf0f0
000000_00000_00001_00010_00100_000000   // sll  $2, $1, 2
000000_00000_00001_00010_00100_000010   // srl  $2, $1, 2
000000_00000_00001_00010_00100_000011   // sra  $2, $1, 2
```

结果如图所示：



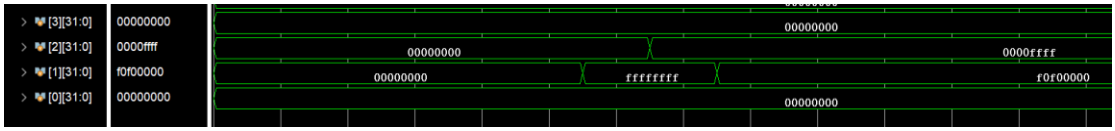
测试代码将 1 号寄存器设为 0xfffff0f0（有符号数扩展）。在 sll 指令中，1 号寄存器左移 4 位得到 0xffff0f00。在 srl 指令中，1 号寄存器右移四位得到 0x0fffff0f。在 sra 指令中，1 号寄存器右移 4 位，并在高位填充符号位 1，得到 0xffffff0f。仿真符合预期，也没有数据冒险错误。

4.3.3 addi，addiu 和 lui 的测试

编写 Instruction 如下：

```
// Test of addi, addiu, lui
001000_00000_00001_1111111111111111     // addi  $1, $0, 0xffff
001001_00000_00010_1111111111111111     // addiu $2, $0, 0xffff
001111_00000_00001_1111000011110000     // lui   $1, 0xf0f0
```

结果如图所示：



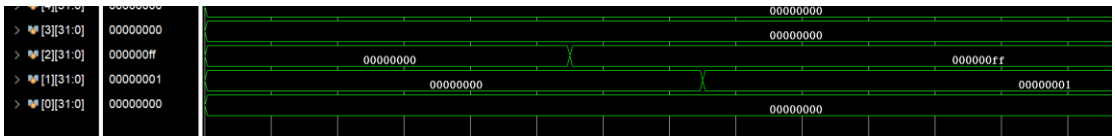
仿真代码中 addi 指令将 0xffff 做有符号数扩展得到 0xffffffff，addiu 指令将 0xffff 做无符号数扩展得到 0x0000ffff。lui 指令将立即数 0xf0f0 填入高 16 位，并把低 16 位清零得到 0xf0f00000。仿真符合预期，也没有数据冒险错误。

4.3.4 slti 和 sltiu 的测试

编写 Instruction 如下：

```
// Test of slti, sltiu
001000_00000_00010_0000000011111111     // addi  $2, $0, 0xff
001010_00010_00001_1000000000000000     // slti  $2, $1, 0x8000
001011_00010_00001_1000000000000000     // sltiu $2, $1, 0x8000
```

结果如图所示：



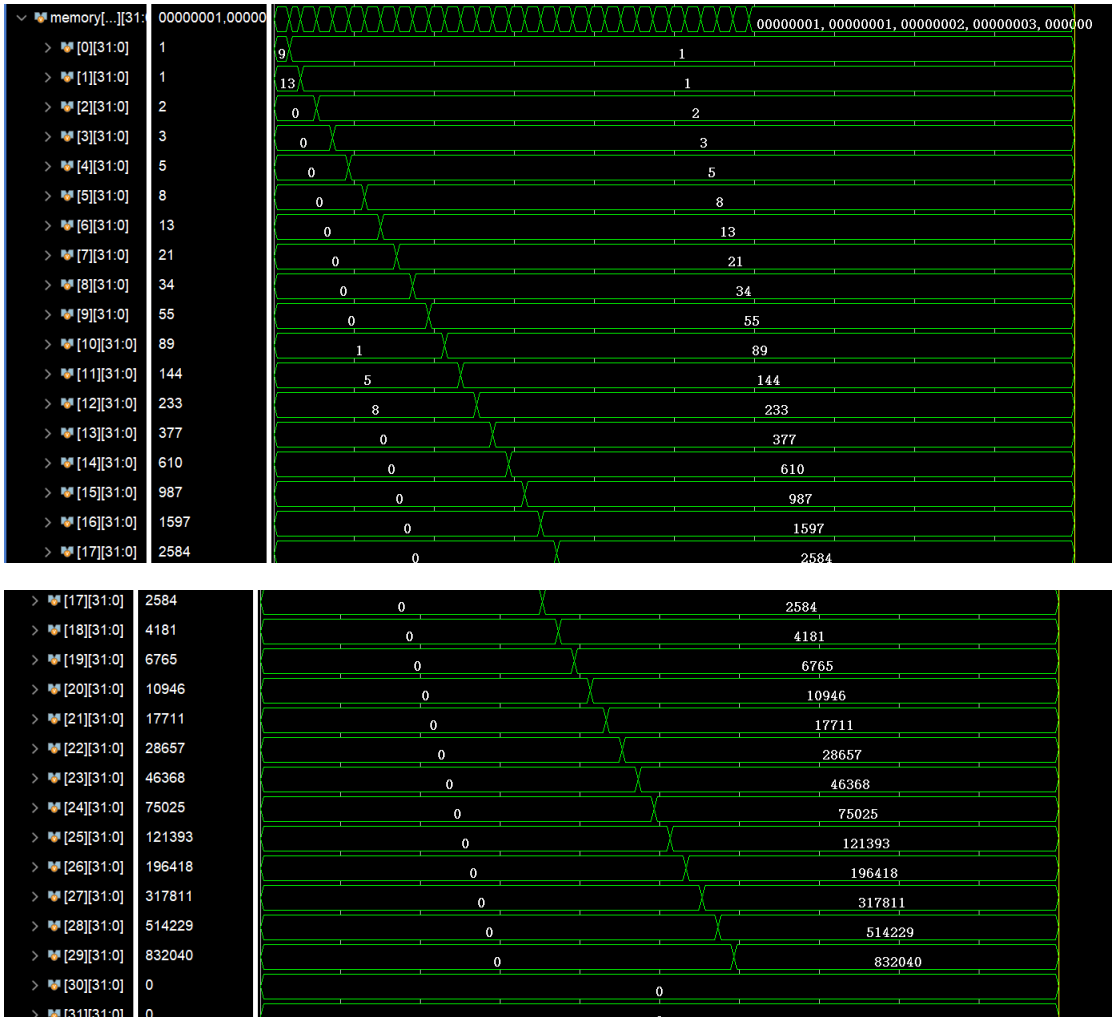
测试代码把 2 号寄存器设为 0x00ff。在 slti 指令中，0x8000 做有符号数扩展，为负数，0x00ff < 0xffff8000 不成立，故 1 号寄存器不设为 1。在 sltiu 指令中，0x8000 做无符号数扩展 0x00ff < 0xffff 成立，故 1 号寄存器设为 1。仿真符合预期，也没有数据冒险错误。

4.5 综合测试

编写斐波那契数列计算代码如下：

```
// Test program of Fibonacci sequence
001111_00000_00001_0000000000001111     // lui   $1, 0x000f     ; Calculation stops if result is greater than 1000000
001001_00001_00001_0100001001000000     // addiu $1, $1, 0x4240 ; $1 = 0x000f4240 = 1000000
001000_00000_00010_0000000000000100     // addi  $2, $0, 4      ; $2 = 4 (memory address to store the result)
001000_00000_00011_0000000000000001     // addi  $3, $0, 1      ; $3 = 1 (current Fibonacci number)
001000_00000_00100_0000000000000000     // addi  $4, $0, 0      ; $4 = 0 (previous Fibonacci number)
101011_00000_00011_0000000000000000     // sw    $3, 0($0)      ; Store the first Fibonacci number at memory address 0
001000_00000_00111_0000000000000001     // addi  $7, $0, 1      ; $7 = 1
000000_00011_00100_00101_00000_100000   // add   $5, $3, $4     ; loop: $5 = $3 + $4 (next Fibonacci number)
000000_00011_00000_00100_00000_100000   // add   $4, $3, $0     ; $4 = $3 (update previous Fibonacci number)
000000_00101_00000_00011_00000_100000   // add   $3, $5, $0     ; $3 = $5 (update current Fibonacci number)
000000_00101_00001_00110_00000_101010   // slt   $6, $5, $1     ; $6 = 1 if $5 < $1, 0 otherwise
000101_00110_00111_0000000000000011     // bne   $6, $7, end    ; if $6 != 1 ($5 >= 1000000), go to end
101011_00010_00101_0000000000000000     // sw    $5, 0($2)      ; Store $5 at memory address $2
001000_00010_00010_0000000000000100     // addi  $2, $2, 4      ; $2 = $2 + 4 (update memory address)
000010_00000000000000000000000111       // j     loop           ; Go to loop
000010_00000000000000000000001110       // j     14             ; end: jump to itself (infinite loop)
```

测试结果如图所示：



测试结果符合预期。

5. 总结与反思

在 Lab06 中，我深入了解了 MIPS 多周期流水线处理器的各部件功能和关系。通过完成消除控制冒险、消除数据冒险、进行分支预测等功能，我熟悉了处理器 IF、ID、EX、MEM 和 WB 阶段的具体流程，掌握了冒险的成因和解决方法，也了解了如何写出减少流水线停顿的代码。通过使用 Vivado 开发环境，我能够更好地编写仿真文件调试 Verilog HDL 的代码和模块。

我要感谢课程组为我们提供的详细指导书，它为我提供了清晰的实验步骤，使我能够更好地理解和实践所学的知识。通过这次实验，我不仅巩固了 Verilog 和 MIPS 处理器的基础知识，还为的学习和设计打下了坚实的基础。