# 计算机结构实验 lab06

张露伊

2022 年 4 月 20 日

**摘要**

本实验以实验 3、4 实现的功能模块以及实验 5 实现的类 MIPS 单周期处理器为基础，对部分功能模块进行了修改，增添高速缓存模块，并将支持的指令扩展到 31 条。此外，为实现流水线处理器，本实验增加了段寄存器，使用前向通路 (forwarding) 与流水线停顿 (stall) 来解决流水线冒险，通过预测不转移 (predict-not-taken) 策略提高流水线性能。最后，本实验将通过软件仿真的形式让处理器运行指令，以此进行实验结果的验证。

## 目录

# 1 实验介绍

## 1.1 实验名称

简单的类 MIPS 多周期流水化处理器实现

## 1.2 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 16 条扩充为 31 条，使处理器功能更加丰富。

6. 功能仿真

# 2 原理分析

## 2.1 各模块原理分析

### 2.1.1 主控制器模块的分析

主控制器 (Ctr) 的输入为指令的操作码 (opCode) 字段，主控制器模块对操作码进行译码，向 ALU 控制器、寄存器、数据选择器等部件输出正确的控制信号。

本实验中，主控制器模块可以识别 R 型指令和 MIPS 指令集中其他所有指令并输出对应的控制信号。相比于实验 5 中的主控制器模块，为了提高流水线执行 JR 指令的性能，需要在指令译码 (ID) 阶段完成 JR 指令的识别，而实验 5 中 jrSign 信号由 ALUCtr 产生，无法在 ID 阶段完成，所以在本实验中我们调整了主控制器模块的功能，使之产生 jrSign 信号。

相比实验五，主控制器模块额外产生的控制信号及说明如下所示：

| 信号 | 内部寄存器 | 具体说明 |
|---|---|---|
| luiSign | LuiSign | 载入立即数指令 (LUI) 信号，高电平有效 |
| jumpSign | JumpSign | 无条件跳转指令 (J) 信号，高电平有效 |
| jrSign | JrSign | 寄存器无条件跳转指令 (JR) 信号，高电平有效 |
| beqSign | BeqSign | 条件跳转指令 (BEQ) 信号，高电平有效 |
| bneSign | BneSign | 条件跳转指令 (BNE) 信号，高电平有效 |

表 1: 主控制器产生的控制信号

| OpCode | 指令 | aluOp | aluSrc | memRead | memToReg | memWrite | regDst | regWrite | extSign | beqSign | bneSign | jumpSign | jalSign | jrSign | luiSign |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 000101 | bne | 001 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 000000 | jr | 101 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 001111 | lui | 000 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

表 2: 增加指令对应的主控制器 (Ctr) 控制信号

### 2.1.2 ALU 控制器模块

该实验的设计和实验五几乎相同，唯一的区别是该模块不再产生 jrSign 信号。故而不再赘述。

### 2.1.3 ALU 模块

本实验 ALU 模块的设计与实验五完全一致，故而不再赘述。

### 2.1.4 寄存器模块

本实验寄存器模块的设计与实验五完全一致，故而不再赘述。

### 2.1.5 高速缓存模块

高速缓存 (Cache) 是用于减少处理器访问内存所需平均时间的部件，其容量远小于内存，但速度却可以接近处理器的频率。

本实验中的高速缓存采用全相联映射策略，块大小为 128 位，总共 16 个数据块，数据缓存总容量为 64 个字。对于读操作，首先检查 valid 位是否有效以及 tag 是否与内存地址一致，如果缓存命中，则直接返回缓存，如果未命中，则从内存中读取一个块的内容，覆盖缓存中对应块。对于写操作，采用直写策略，直接写入内存，同时将缓存中对应块 valid 位清空。

高速缓存的内存地址映射策略如图1所示。

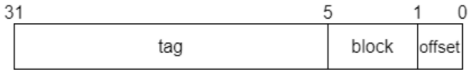


图 1: 高速缓存地址映射策略

高速缓存模块输入输出信号如表3所示。

| 输入信号 | 长度 | 说明 |
|---|---|---|
| address | 32 | 内存地址 |
| writeData | 32 | 写入数据 |
| memWrite | 1 | 内存写使能信号 |
| memRead | 1 | 内存读使能信号 |
| clk | 1 | 时钟信号 |
| 输出信号 | 长度 | 说明 |
| readData | 32 | 内存读取结果 |

表 3: 高速缓存模块输入输出信号

不难看出，本实验中高速缓存模块的输入输出与实验 5 中内存模块输入输出完全一致。因此，对于外部电路而言，使用高速缓存与直接使用内存模块在接口上没有差异。

### 2.1.6 内存单元模块

同样，本实验中的内存单元模块设计和实验五相似，只不过本实验中由于我们加入了高速缓存，为了与高速缓存模块对接，本实验中的内存单元模块将会一次性返回 4 个字的内容。

内存模块输入输出信号如表4所示。

### 2.1.7 带符号扩展模块

本实验中带符号扩展模块与实验五相同，故而不再赘述。

### 2.1.8 数据选择模块

本实验中数据选择模块与实验五相同，故而不再赘述。

| 输入信号 | 长度 | 说明 |
|---|---|---|
| address | 32 | 内存地址 |
| writeData | 32 | 写入数据 |
| memWrite | 1 | 内存写使能信号 |
| memRead | 1 | 内存读使能信号 |
| clk | 1 | 时钟信号 |
| 输出信号 | 长度 | 说明 |
| readData | 128 | 内存读取结果 |

表 4: 内存模块输入、输出信号

#### 2.1.9 指令内存模块

本实验中指令内存模块与实验五相同，故而不再赘述。

### 2.2 流水线阶段原理分析

#### 2.2.1 取指令阶段 (IF)

取指令阶段主要包括 PC 寄存器与指令内存模块，此阶段指令内存模块根据 PC 寄存器的值取出指令。

#### 2.2.2 译码阶段 (ID)

译码阶段主要包括主控制器模块 (Ctr)、寄存器模块 (Register)、符号扩展模块。主控制器产生控制信号，寄存器模块读取寄存器数据，符号扩展模块将立即数扩展为 32 位。同时，在此阶段目标寄存器选择器还会在 rt 与 rd 中选择出写入寄存器，此阶段仅选择出写入寄存器后并不会执行写操作，选择结果会一直保存到 WB 阶段进行实际写入。提前完成写入寄存器选择工作，既便于之后的数据保存、传输，也方便进行数据冒险判断与前向通路实现。

对于无条件跳转指令 j、jal、jr，其对应的所有操作都会在本阶段完成，以此来提高流水线的执行效率。

#### 2.2.3 执行阶段 (EX)

执行阶段主要包括 ALU 控制器 (ALUCtr)、ALU 以及一系列用于选择 ALU 输入数据的选择器。本阶段中 ALU 会根据 ALU 控制器的控制信号与输入数据计算出结果，对于 beq、bne 指令，本阶段也会决定是否跳转。

对于 lui 指令，lui 选择器会选取指令中立即数部分作为本阶段的执行结果的高 16 位，ALU 的计算结果在此指令下会被丢弃。

#### 2.2.4 访存阶段 (MEM)

访存阶段主要包括高速缓存模块，高速缓存模块与内存模块相连，用以加速内存操作。对于需要进行访存的指令，将在本阶段完成访存操作。本阶段还有一个数据选择器，该选择器根据 MEM_TO_REG 控制信号，在访存结果与执行阶段结果中选择一个数据作为访存阶段的结果。

### 2.2.5 写回阶段 (WB)

写回阶段主要包括寄存器模块，对于需要寄存器写入的指令，本阶段将完成寄存器写操作。其中，写入寄存器的编号在 ID 阶段已经确定，写入的数据为 MA 阶段的结果。

## 2.3 顶层模块 (top) 原理分析

### 2.3.1 段寄存器

在流水线的两个阶段之间，需要通过段寄存器临时保存上一阶段的执行结果和控制信号。

- IF-ID 段寄存器：主要包含当前指令内容及 PC
- ID-EX 段寄存器：主要包含控制信号、符号扩展的结果，rs、rt 对应寄存器的编号，目标写入寄存器的编号，funct、shamt 的值、当前指令 PC
- EX-MEM 段寄存器：主要包含控制信号、ALU 的运算结果、rt 寄存器的编号以及目标写入寄存器的编号
- MEM-WB 段寄存器：主要包含 regWrite 控制信号、最终写入寄存器的数据以及目标写入寄存器的编号

控制信号在段之间的大致传递关系如图所示:

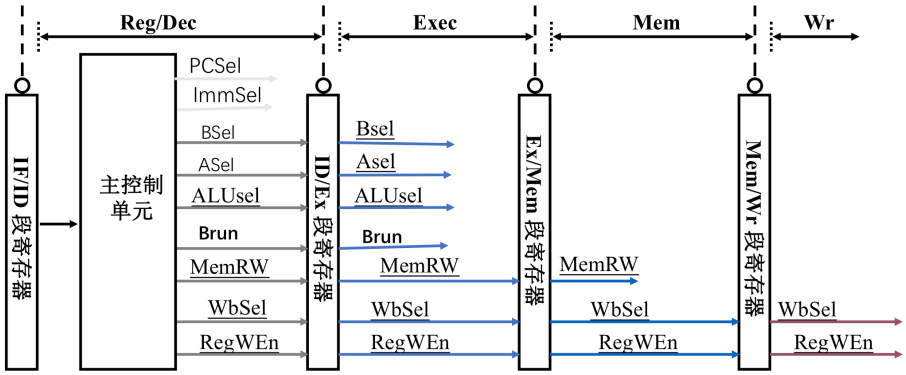


图 2: 控制信号的传递关系

### 2.3.2 数据前向传递原理分析

在流水线中，当前指令在 EX 阶段所需要的数据可能来自先前指令的写回结果，但是被依赖的指令可能才刚开始 MEM 阶段或 WB 阶段。尽管当前指令可以等待依赖指令完全执行结束再从寄存器中获得需要的数据，但是这样的策略会导致大量流水线停滞，降低处理器效率。通过添加前向数据通路，从 EX-MEM 段寄存器、MEM-WB 段寄存器获得需要的数据，可以不需要等待先前指令完成 WB 阶段，由此提高效率。

#### 2.3.3 停顿机制原理分析

对于“读内存-使用”型数据冒险，前向传递无法避免停顿，这是由于 lw 指令需要在 MEM 阶段完成后才能得到需要的数据，而下一条指令必须在 EX 阶段开始前获得需要的数据。因此，后一条指令必须在 EX 阶段开始前等待一个周期，待 lw 指令完成 MEM 阶段后，使用前向数据通路将访存结果送回 EX 阶段。

在每条指令的 ID 阶段，检测该指令是否和前一条指令形成“读内存-使用”型数据冒险，如果存在这样的冒险，则发出 STALL 信号，使得 IF、ID 阶段停顿一个周期。

#### 2.3.4 分支预测原理分析

对于跳转指令，我们通过预测不转移 (predict-not-taken) 来解决条件转移带来的控制竞争，即对于所有的指令均预测不会跳转，当预测错误时，则生成 NOP 信号请求清空 IF、ID 阶段。这种策略的正确性需要分情况说明：

- **无条件跳转指令：**这类指令会在 ID 阶段完成跳转，产生的 NOP 信号会清空 ID-EX 段寄存器，即跳转指令本身会在 ID 阶段结束后被清除，但由于跳转指令在 ID 阶段就已经完成了所有工作，因此并不会导致执行错误。
- **条件跳转指令：**这类指令会在 EX 阶段完成跳转，当跳转发生时，IF、ID 阶段的指令由于预测错误，理应被清除，因此在此情况下，我们的清除策略也是正确的。

在此，有以下两点需要特别说明：

- 当 IF-ID 段寄存器接收到 NOP 信号时，并不一定会清空段寄存器，此处我们额外加入了当前指令地址和跳转目标 PC 的比较，当两者相同时，指令会正常执行。这样的设计可以为短跳转（例如仅跳过一行指令的条件跳转）减少一个停顿周期。
- 在跳转目标 PC 的选择电路中，为了保证跳转目标 PC 的正确性，我们将条件跳转（beq、bne）的选择器安排在无条件跳转之后。

#### 2.3.5 jal 指令实现原理分析

无条件跳转语句会在 ID 阶段完成所有工作，但是 jal 指令由于需要进行寄存器写入，可能会与处于 WB 阶段的指令产生结构冒险。为解决这个问题，处理 jal 指令时，将当前处于 EX、MEM、WB 阶段的指令全部停顿一个周期，为 jal 指令让出寄存器写入端口。由于 jal 使用的 31 号寄存器不是通用寄存器，其余指令不会对此寄存器进行写入，改变寄存器写入顺序不会导致数据错误。

# 3 功能实现

## 3.1 功能模块的实现

### 3.1.1 主控制器模块的实现

主控制器模块的完整实现见 Ctr.v，相比实验 5 中实现代码，本实验中增加了部分跳转指令与 lui 指令的控制信号，同时将 jrSign 信号的产生集成到主控制器模块中。部分核心代码如下：

```
always @(opCode or funct)
begin
    case(opCode)
    6'b000000: //R type
    begin
        RegDst = 1;
        ALUSrc = 0;
        MemToReg = 0;
        MemRead = 0;
        MemWrite = 0;
        BeqSign = 0;
        BneSign = 0;
        ExtSign = 0;
        LuiSign = 0;
        JalSign = 0;
        ALUOp = 3'b101;
        JumpSign = 0;
        if (funct == 6'b001000) begin
            RegWrite = 0;
            JrSign = 1;
        end else begin
            RegWrite = 1;
            JrSign = 0;
        end
    end
    //后续代码类似，此处省略
    endcase
end
```

### 3.1.2 ALU 控制器模块的实现

ALU 控制器模块的完整实现见 ALUCtr.v，相比实验 5 中实现，本实验中 ALU 控制器模块不再产生 jrSign 信号。核心代码如下：

```
always @ (aluOp or funct)
begin
    ShamtSign = 0;
    casex({aluOp,funct})
        9'b000xxxxxx:  // lw,sw,add,addiu
            ALUCtrOut = 4'b0010;     //add
        9'b001xxxxxx:  // beq,bne
```

```
8               ALUCtrOut = 4'b0110;     //sub
9           9'b010xxxxxx:    //stli
10              ALUCtrOut = 4'b0111;
11          9'b110xxxxxx:    //stliu
12              ALUCtrOut = 4'b1000;
13          9'b011xxxxxx:  // andi
14              ALUCtrOut = 4'b0000;
15          9'b100xxxxxx:  // ori
16              ALUCtrOut = 4'b0001;
17          9'b111xxxxxx:  // xori
18              ALUCtrOut = 4'b1011;
19
20          //R type
21          9'b101001000:  // jr
22              ALUCtrOut = 4'b0101;
23          9'b101000000:  // sll
24          begin
25              ALUCtrOut = 4'b0011;
26              ShamtSign = 1;
27          end
28          9'b101000010:  // srl
29          begin
30              ALUCtrOut = 4'b0100;
31              ShamtSign = 1;
32          end
33          9'b101000011:  // sra
34          begin
35              ALUCtrOut = 4'b1110;
36              ShamtSign = 1;
37          end
38          9'b101000100:  // sllv
39              ALUCtrOut = 4'b0011;
40          9'b101000110:  // srlv
41              ALUCtrOut = 4'b0100;
42          9'b101000111:  // srav
43              ALUCtrOut = 4'b1110;
44          9'b101100000:  // add
45              ALUCtrOut = 4'b0010;
46          9'b101100001:  // addu
47              ALUCtrOut = 4'b0010;
48          9'b101100010:  // sub
```

```
49                ALUCtrOut = 4'b0110;
50            9'b101100011:   // subu
51                ALUCtrOut = 4'b0110;
52            9'b101100100:   // and
53                ALUCtrOut = 4'b0000;
54            9'b101100101:   // or
55                ALUCtrOut = 4'b0001;
56            9'b101100110:   // xor
57                ALUCtrOut = 4'b1011;
58            9'b101100111:   // nor
59                ALUCtrOut = 4'b1100;
60            9'b101101010:   // slt
61                ALUCtrOut = 4'b0111;
62            9'b101101011:   // sltu
63                ALUCtrOut = 4'b1000;
64        endcase
65    end
```

### 3.1.3 ALU 模块的实现

ALU 模块的完整实现见 ALU.v，本实验 ALU 模块实现与实验 5 完全一致，故不再赘述。

### 3.1.4 寄存器模块的实现

寄存器模块的完整实现见 Registers.v，本实验该模块实现与实验 5 完全一致，故不再赘述。

### 3.1.5 高速缓存模块的实现

对于外部系统而言，高速缓存模块的接口与普通内存模块没有差异。进行读操作时，高速缓存模块会首先检查缓存是否有效，若缓存失效则从内存读取数据并保存至缓存。进行写操作时，直接写入内存模块，同时将缓存中对应块标记为无效。

当缓存失效而从内存读取数据时，需要添加延时以等待内存模块完成读取操作，待内存模块读取完成且缓存块更新之后再输出读取结果。

高速缓存模块的完整实现见 Cache.v, 核心部分代码如下：

```
1  reg[31:0] cacheFile[0:63];
2  reg validBit[0:15];
3  reg[25:0] tag[0:15];
4  reg[31:0] ReadData;
5  wire[127:0] dataFromMemFile;
6  wire[3:0] cacheAddr = address[5:2];
7  wire[31:0] MemFileAddress = {address[31:2],2'b00};
8  integer i;
```

```
dataMemory mem(
    .clk(clk),
    .address(MemFileAddress),
    .writeData(writeData),
    .memWrite(memWrite),
    .memRead(memRead),
    .readData(dataFromMemFile)
);
initial
begin
    for(i=0;i<64;i=i+1)
        validBit[i] = 1'b0;
        tag[i] = 16'b0;
end
always @(memRead or address or memWrite)
begin
    if(memRead)
    begin
        if(validBit[cacheAddr] & tag[cacheAddr] == address[31:6])
            ReadData = cacheFile[address[5:0]];
        else begin
            tag[cacheAddr] = address[31:6];
            validBit[cacheAddr] = 1'b1;
            #5
            cacheFile[{cacheAddr,2'b11}] = dataFromMemFile[31:0];
            cacheFile[{cacheAddr,2'b10}] = dataFromMemFile[63:32];
            cacheFile[{cacheAddr,2'b01}] = dataFromMemFile[95:64];
            cacheFile[{cacheAddr,2'b00}] = dataFromMemFile[127:96];
            ReadData = cacheFile[address[5:0]];
        end
    end
end
always @(negedge clk)
begin
    if(memWrite)
        validBit[cacheAddr] = 1'b0;
end
assign readData = ReadData;
```

#### 3.1.6 内存单元模块的实现

内存模块的完整实现见 dataMemory.v, 核心部分代码如下:

```
reg [31:0] memFile [0:1023];
reg [127:0] ReadData;
always @(memRead or address or memWrite)
begin
    if(memRead)
    begin
        if(address<1023)
            ReadData = {memFile[address],memFile[address+1],
            memFile[address+2],memFile[address+3]};
        else
            ReadData = 0;
    end
end
always @(negedge clk)
begin
    if(memWrite)
        if(address<1023)
            memFile[address] = writeData;
end
assign readData = ReadData;
```

#### 3.1.7 符号扩展模块的实现

带符号扩展模块的完整实现见 Signex.v。本实验该模块的实现与实验五相同，故不再赘述。

#### 3.1.8 数据选择器模块

本实验该模块的实现与实验五相同，故不再赘述。

#### 3.1.9 指令内存模块

本实验该模块的实现与实验五相同，故不再赘述。

### 3.2 顶层模块的实现

#### 3.2.1 段寄存器实现

```
//IF to ID
reg [31:0] IF2ID_INST;
reg [31:0] IF2ID_PC;
```

```
//ID to EX
reg [2:0] ID2EX_ALUOP;
reg [7:0] ID2EX_CTR_SIGNALS;
reg [31:0] ID2EX_EXT_RES;
reg [4:0] ID2EX_INST_RS;
reg [4:0] ID2EX_INST_RT;
reg [31:0] ID2EX_REG_READ_DATA1;
reg [31:0] ID2EX_REG_READ_DATA2;
reg [5:0] ID2EX_INST_FUNCT;
reg [4:0] ID2EX_INST_SHAMT;
reg [4:0] ID2EX_REG_DEST;
reg [31:0] ID2EX_PC;
//EX to MEM
reg [3:0] EX2MA_CTR_SIGNALS;
reg [31:0] EX2MA_ALU_RES;
reg [31:0] EX2MA_REG_READ_DATA_2;
reg [4:0] EX2MA_REG_DEST;
//MEM to WB
reg MA2WB_CTR_SIGNALS;
reg [31:0] MA2WB_FINAL_DATA;
reg [4:0] MA2WB_REG_DEST;
```

### 3.2.2 功能模块连接

每个阶段中的功能模块从段寄存器中获得输入信号，产生输出信号，输出信号将在下一个时钟上升沿写入下一阶段的段寄存器中。

以主控制器模块的连接为例，其实现如下：

```
wire [12:0] ID_CTR_SIGNALS;
wire [2:0] ID_CTR_SIGNAL_ALUOP;
wire ID_JUMP_SIG;
wire ID_JR_SIG;
wire ID_EXT_SIG;
wire ID_REG_DST_SIG;
wire ID_JAL_SIG;
wire ID_ALU_SRC_SIG;
wire ID_LUI_SIG;
wire ID_BEQ_SIG;
wire ID_BNE_SIG;
wire ID_MEM_WRITE_SIG;
wire ID_MEM_READ_SIG;
wire ID_MEM_TO_REG_SIG;
```

```
wire ID_REG_WRITE_SIG;
wire ID_ALU_OP;
Ctr main_ctr(
    .opCode(IF2ID_INST[31:26]),
    .funct(IF2ID_INST[5:0]),
    .jumpSign(ID_JUMP_SIG),
    .jrSign(ID_JR_SIG),
    .extSign(ID_EXT_SIG),
    .regDst(ID_REG_DST_SIG),
    .jalSign(ID_JAL_SIG),
    .aluSrc(ID_ALU_SRC_SIG),
    .luiSign(ID_LUI_SIG),
    .beqSign(ID_BEQ_SIG),
    .bneSign(ID_BNE_SIG),
    .memWrite(ID_MEM_WRITE_SIG),
    .memRead(ID_MEM_READ_SIG),
    .memToReg(ID_MEM_TO_REG_SIG),
    .regWrite(ID_REG_WRITE_SIG),
    .aluOp(ID_CTR_SIGNAL_ALUOP)
);
//将信号合并为总线，便于段寄存器读写
assign ID_CTR_SIGNALS[12] = ID_JUMP_SIG;
assign ID_CTR_SIGNALS[11] = ID_JR_SIG;
assign ID_CTR_SIGNALS[10] = ID_EXT_SIG;
assign ID_CTR_SIGNALS[9] = ID_REG_DST_SIG;
assign ID_CTR_SIGNALS[8] = ID_JAL_SIG;
assign ID_CTR_SIGNALS[7] = ID_ALU_SRC_SIG;
assign ID_CTR_SIGNALS[6] = ID_LUI_SIG;
assign ID_CTR_SIGNALS[5] = ID_BEQ_SIG;
assign ID_CTR_SIGNALS[4] = ID_BNE_SIG;
assign ID_CTR_SIGNALS[3] = ID_MEM_WRITE_SIG;
assign ID_CTR_SIGNALS[2] = ID_MEM_READ_SIG;
assign ID_CTR_SIGNALS[1] = ID_MEM_TO_REG_SIG;
assign ID_CTR_SIGNALS[0] = ID_REG_WRITE_SIG;
```

### 3.2.3 前向通路实现

前向数据通路的实现如下:

```
wire[31:0] EX_FORWARDING_A_TEMP;
wire[31:0] EX_FORWARDING_B_TEMP;
Mux forward_A_mux1(
```

```
    .select(WB_REG_WRITE & (MA2WB_REG_DEST == ID2EX_INST_RS)),
    .input0(ID2EX_REG_READ_DATA1),
    .input1(MA2WB_FINAL_DATA),
    .out(EX_FORWARDING_A_TEMP)
);
Mux forward_A_mux2(
    .select(MA_REG_WRITE & (EX2MA_REG_DEST == ID2EX_INST_RS)),
    .input0(EX_FORWARDING_A_TEMP),
    .input1(EX2MA_ALU_RES),
    .out(FORWARDING_RES_A)
);
Mux forward_B_mux1(
    .select(WB_REG_WRITE & (MA2WB_REG_DEST == ID2EX_INST_RT)),
    .input0(ID2EX_REG_READ_DATA2),
    .input1(MA2WB_FINAL_DATA),
    .out(EX_FORWARDING_B_TEMP)
);
Mux forward_B_mux2(
    .select(MA_REG_WRITE & (EX2MA_REG_DEST == ID2EX_INST_RT)),
    .input0(EX_FORWARDING_B_TEMP),
    .input1(EX2MA_ALU_RES),
    .out(FORWARDING_RES_B)
);
```

代码中 forward_A、forward_B 为两组前项通路，分别用于将数据传输 ALU 的两个输入端口，每组通路包括两个选择器，分别从 EX-MEM 段寄存器与 MEM-WB 段寄存器获取数据。

### 3.2.4 跳转目标 PC 选择的实现

PC 地址选择的实现代码如下：

```
// ID stage
wire[31:0] PC_AFTER_JUMP_MUX;
Mux jump_mux(
    .select(ID_JUMP_SIG),
    .input1(((IF2ID_PC + 4) & 32'hf0000000) + (IF2ID_INST [25 : 0] << 2)),
    .input0(IF_PC + 4),
    .out(PC_AFTER_JUMP_MUX)
);
wire[31:0] PC_AFTER_JR_MUX;
Mux jr_mux(
    .select(ID_JR_SIG),
    .input0(PC_AFTER_JUMP_MUX),
```

```
        .input1(ID_REG_READ_DATA1),
        .out(PC_AFTER_JR_MUX)
);
// EX stage
wire EX_BEQ_BRANCH = EX_BEQ_SIG & EX_ALU_ZERO;
wire[31:0] PC_AFTER_BEQ_MUX;
Mux beq_mux(
        .select(EX_BEQ_BRANCH),
        .input1(BRANCH_DEST),
        .input0(PC_AFTER_JR_MUX),
        .out(PC_AFTER_BEQ_MUX)
);
wire EX_BNE_BRANCH = EX_BNE_SIG & (~EX_ALU_ZERO);
wire[31:0] PC_AFTER_BNE_MUX;
Mux bne_mux(
        .select(EX_BNE_BRANCH),
        .input1(BRANCH_DEST),
        .input0(PC_AFTER_BEQ_MUX),
        .out(PC_AFTER_BNE_MUX)
);
wire[31:0] NEXT_PC = PC_AFTER_BNE_MUX;
wire BRANCH = EX_BEQ_BRANCH | EX_BNE_BRANCH;
```

### 3.2.5 流水线时序实现

本部分是流水线处理器实现的核心，段寄存器写入、分支预测、停顿等功能均在本部分实现。各个段寄存器的清空、停顿条件如下：

- **IF-ID 段寄存器：**如果 NOP 信号有效且当前指令地址与目标跳转 PC 不一致，说明分支预测失败，则清空段寄存器。
- **ID-EX 段寄存器：**ID_JAL_SIG 信号有效时说明当前指令为 jal，ID-EX 段寄存器维持原样，ID 阶段结果不写入段寄存器。STALL 信号有效时表明需要停顿，清空 ID-EX 段寄存器，避免 EX 阶段执行错误指令。NOP 指令有效时说明分支预测错误，清空 ID-EX 段寄存器。
- **EX-MEM 段寄存器：**ID_JAL_SIG 信号有效时说明当前指令为 jal，ID 之后的阶段需要停顿一个周期，EX-MEM 段寄存器维持原样。
- **MEM-WB 段寄存器：**ID_JAL_SIG 信号有效时说明当前指令为 jal，ID 之后的阶段需要停顿一个周期，MEM-WB 段寄存器维持原样。

流水线时序实现代码如下：

```
always @(posedge clk)
```

```
2  begin
3      NOP = BRANCH | ID_JUMP_SIG | ID_JR_SIG;
4      STALL = ID2EX_CTR_SIGNALS[2] &
5                  ((ID2EX_INST_RT == ID_REG_RS) |
6                  (ID2EX_INST_RT == ID_REG_RT));
7      if(!STALL)
8      begin
9          if(NOP)
10         begin
11             if(IF_PC == NEXT_PC)
12             begin
13                 IF2ID_INST <= IF_INST;
14                 IF2ID_PC <= IF_PC;
15                 IF_PC <= IF_PC + 4;
16             end
17             else begin
18                 IF2ID_INST <= 0;
19                 IF2ID_PC <= 0;
20                 IF_PC <= NEXT_PC;
21             end
22         end
23         else begin
24             IF2ID_INST <= IF_INST;
25             IF2ID_PC <= IF_PC;
26             IF_PC <= NEXT_PC;
27         end
28     end
29
30     // ID - EX
31     if (!ID_JAL_SIG)
32     begin
33         if (STALL|NOP)
34         begin
35             ID2EX_PC <= IF2ID_PC;
36             ID2EX_ALUOP <= 3'b000;
37             ID2EX_CTR_SIGNALS <= 0;
38             ID2EX_EXT_RES <= 0;
39             ID2EX_INST_RS <= 0;
40             ID2EX_INST_RT <= 0;
41             ID2EX_REG_READ_DATA1 <= 0;
42             ID2EX_REG_READ_DATA2 <= 0;
```

```
            ID2EX_INST_FUNCT <= 0;
            ID2EX_INST_SHAMT <= 0;
            ID2EX_REG_DEST <= 0;
        end else
        begin
            ID2EX_PC <= IF2ID_PC;
            ID2EX_ALUOP <= ID_CTR_SIGNAL_ALUOP;
            ID2EX_CTR_SIGNALS <= ID_CTR_SIGNALS[7:0];
            ID2EX_EXT_RES <= ID_EXT_RES;
            ID2EX_INST_RS <= ID_REG_RS;
            ID2EX_INST_RT <= ID_REG_RT;
            ID2EX_REG_DEST <= ID_REG_DEST;
            ID2EX_REG_READ_DATA1 <= ID_REG_READ_DATA1;
            ID2EX_REG_READ_DATA2 <= ID_REG_READ_DATA2;
            ID2EX_INST_FUNCT <= IF2ID_INST[5:0];
            ID2EX_INST_SHAMT <= IF2ID_INST[10:6];
        end
    end
    // EX - MEM
    if (!ID_JAL_SIG)
    begin
        EX2MA_CTR_SIGNALS <= ID2EX_CTR_SIGNALS[3:0];
        EX2MA_ALU_RES <= EX_FINAL_DATA;
        EX2MA_REG_READ_DATA_2 <= FORWARDING_RES_B;
        EX2MA_REG_DEST <= ID2EX_REG_DEST;
    end
    // MEM - WB
    if (!ID_JAL_SIG)
    begin
        MA2WB_CTR_SIGNALS <= EX2MA_CTR_SIGNALS[0];
        MA2WB_FINAL_DATA <= MA_FINAL_DATA;
        MA2WB_REG_DEST <= EX2MA_REG_DEST;
    end
end
```

# 4 结果验证

编写如下表所示的汇编代码进行测试。

```
j    10
nop
```

```
3  nop
4  nop
5  jal 18
6  nop
7  lui $2,65535
8  lw  $1,2($0)
9  lw  $1,3($0)
10 lw  $2,4($0)
11 add $3,$1,$2
12 and $3,$1,$2
13 addi $3,$0,6
14 sub $4,$1,$2
15 sll $1,$1,1
16 sll $3,$3,3
17 or  $1,$3,$1
18 addiu $2,$0,6
19 slti $1,$1,1
20 ori  $1,$2,1
21 sllv $1,$3,$2
22 addu $3,$1,$2
23 sub  $3,$1,$2
24 subu $3,$1,$2
25 or   $3,$1,$2
26 xor  $3,$1,$2
27 nor  $3,$1,$2
28 slt  $3,$1,$2
29 sra  $1,$1,1
30 sw   $2,1($0)
31 sltu $1,$0,$2
32 srl  $2,$2,2
33 srav $2,$2,$2
34 xori $2,$2,2
35 sltiu $1,$2,4
36 beq   $5,$0,1
37 beq  $5,$0,1
38 bne  $5,$0,1
39 jr   $31
```

仿真结果截图如下:

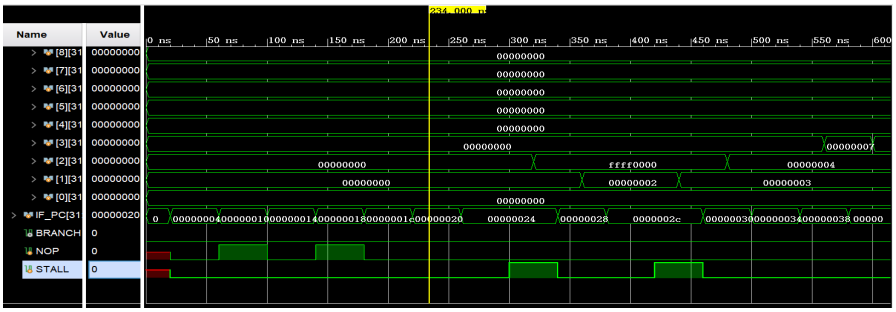


图 3: 仿真截图 1

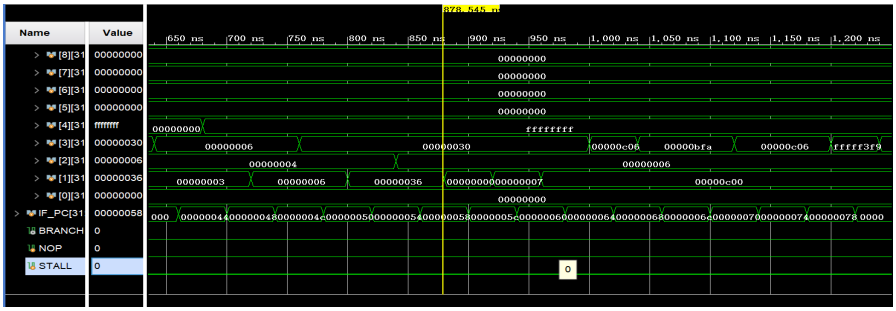


图 4: 仿真截图 2

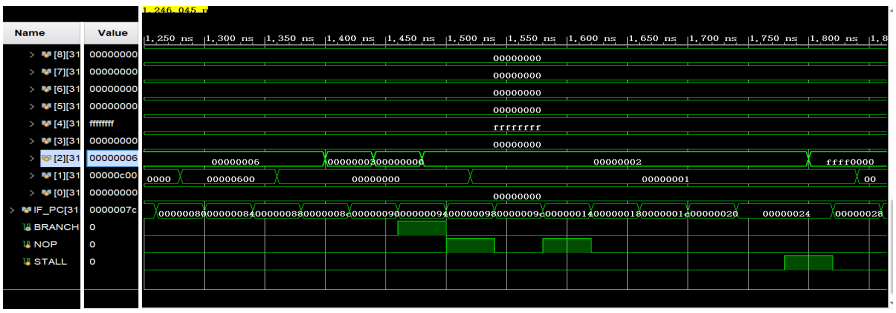


图 5: 仿真截图 3

仿真结果符合预测的期望结果，故说明多周期流水化处理器实现正确。

# 5 实验心得

本实验是前面 5 个实验的综合，个人感觉前面的实验都是一步一步实现为实现最后一个实验打基础的。从整个课程的实验可以看出，实验六应该是本门课程的最终目的，而前面的都是为完成最后一个实验做铺垫。实验开始前，我认为这个实验看起来是很复杂的。我尝试着向之前的实验一样将整体拆分成不同的组块，将流水线结构分解，逐一实现每个阶段的线路连接，其实每个阶段的实现与单周期处理器非常接近，在此之后，只需要将线路整体连接到段寄存器上即可以实现基础的流水线。在完成基础的流水线之后，我通过查找资料以及类比计算机结构中学习的 RISCV 的流水线处理器，在理解原理的情况下，逐步添加前向通路、停顿、分支预测。最终，我完成了整个流水线。

通过本实验，我加深了对流水器处理器结构的记忆与理解。同时 6 个实验的学习调试让我更加熟悉 Verilog 的语法以及 Vivado 软件的仿真调试功能。编写仿真指令的过程，我熟悉了各类 MIPS 指令的结构、功能与实现原理。