计算机系统结构实验报告　实验 6

颜培深 518030910094

2020 年 5 月 22 日

摘要

本实验实现了简单的类 MIPS 多周期流水线处理器，支持 16 条 MIPS 指令（包括 R 型指令：add、sub、and、or、slt、sll、srl、jr；I 型指令：lw、sw、addi、andi、ori、beq；J 型指令：j、jal），由于实验 5 中部分模块代码对于流水线设计可继承性较差，又进行了重新设计，同时为了实现流水线效果，增加了流水段寄存器等部件，支持流水线冒险 (hazard) 检测，插入停顿 (stall) 机制解决冒险，增加了前项通路 (forwarding) 减少流水线停顿延时，并通过 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，提高流水线处理器性能。实验通过软件仿真的方式进行结果验证。

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. 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. 在此基础上增加 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. 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

2 原理分析

2.1 取指阶段 (IF) 设计

本阶段主要包括 PC 寄存器以及指令存储器两大模块，功能为读取当前指令地址并根据该指令地址取出指令字，为译码阶段做准备。

2.1.1 PC 寄存器的设计

PC 寄存器的主要功能是实现指令地址的赋值与读取操作，这里我们将 PC 寄存器不单独实现，而是将其包含在顶层模块中，在每个时钟周期上升沿进行更新，读取来自下一指令地址的数据（由当前执行指令对应的控制信号以及跳转地址等因素决定），同时，PC 寄存器的输出端始终与 PC 寄存器当前值保持一致。

2.1.2 指令存储器 (Instruction Memory) 的设计

指令存储器的主要功能是读取来自 PC 寄存器的指令地址，在对应位置读取指令并将指令二进制编码送入输出端，供每个时钟周期上升沿 IF/ID 段寄存器读取。

2.2 译码阶段 (ID) 设计

本阶段包含主控制器、寄存器以及有符号数拓展三大模块，功能是根据上一周期取指得到的指令字进行控制信号的解析以及寄存器文件的读取，或在某些情况下进行指令字中数值的有符号拓展，并且无条件转移指令的跳转地址也是在该阶段完成，无条件转移指令跳转地址的计算公式如下：

$$JUMP\_ADDRESS = (IF\_ID\_inst[25:0] << 2) + (IF\_ID\_pcp \& 32'hf0000000)$$

其中，$IF\_ID\_inst[25:0]$ 为执行指令字的 0 25 位，IF_ID_pcp 为执行指令对应的地址加四。
同样，条件转移指令的跳转地址也是在本阶段获得，计算公式如下：

$$BRANCH\_ADDRESS = ID\_EX\_pcp + ID\_EX\_branchshift$$

其中，$ID\_EX\_branchshift$ 为执行指令对应中的位移量。

2.2.1 主控制器 (Ctr) 的设计

在流水线处理器的实现中由于主控制器参与指令译码 (Instruction Decode) 阶段，而算术逻辑运算控制单元参与执行阶段 (Execute)，所以不能像实验 5 中那样简单的实现在一个模块中，而是需要添加一些功能使之能够分开独立工作。这里我们在主控制器中功能发生一定的变化，具体功能（控制信号及对应含义）如表1所示。

信号名称	具体含义
ALUSrc	ALU 的第二个输入操作数来源（0：使用 rt 读出值；1：使用立即数）
RegWrite	寄存器写使能信号，高电平有效
RegDst	目标寄存器选择信号（0：写入 rt；1：写入 rd）
MemRead	内存读使能信号，高电平有效
MemWrite	内存写使能信号，高电平有效
MemToReg	选择是否将内存读取内容写入寄存器（0：不写回；1：写回）
Branch	条件跳转使能信号，高电平有效
Jump	无条件跳转使能信号，高电平有效
Jr	读寄存器跳转使能信号，高电平有效
aluOp	ALU 需要执行的操作类型

表 1: 主控制器发出的控制信号含义

与之前不同的是，这里 Shift(Shamt) 信号并没有出现，我们选择将其保留在算术逻辑运算控制单元中进行控制。同时这里 aluOp 为一个两位的控制信号，在主控制器中的输出值仅包含 00、10、01 三种，取值与含义的对应关系如表2所示。

aluOp 取值	含义
10	当前指令为 R 型，具体算术逻辑运算方式取决于指令第六位（即 Funct 域）
00	ALU 执行加法运算
01	ALU 执行减法运算

表 2: aluOp 信号含义

并且，与单周期处理器不同的是，主控制器的所有输入不再是由指令存储器直接提供，而是每个时钟周期上升沿从 IF/ID 段寄存器中获取，而且输出的数据也是大部分存储在 ID/EX 段寄存器中，只有 Jump 信号直接用于判定当前指令是否为无条件转移指令并用于无条件转移。

2.2.2 有符号拓展 (Signal Extend) 的设计

本模块与单周期处理器中实现基本一致，接受一个 16 位二进制数值带符号拓展位为 32 位数值送入输出端，不再赘述。

2.2.3 寄存器 (Register) 的设计

寄存器功能以及实现与单周期处理器一致，但其输入与输出有所变化，用于寄存器读取的 rs、rt 寄存器地址来源于 IF/ID 段寄存器，而 regWrite 控制信号以及写寄存器的数据和目标寄存器的地址来源于 MEM/WB 段寄存器。

2.3 执行阶段 (EX) 的设计

本阶段包含前项通路模块、算术逻辑运算控制单元、ALU 以及一个数据选择器（用于选择记录当前指令对应的写寄存器地址）。同时，如果该指令为 Jr 指令，则其跳转地址在本阶段获取，就等于 ALU 计算结果。

2.3.1 前向通路 (Forwarding) 的设计

前向通路中包含四个控制信号量，分别对应四种类型，对应关系如表3所示。

前向通路控制信号	含义/满足条件
FORWARD$_EX_A$	寄存器文件读取 rs 地址与上一条指令 alu 运算结果写寄存器地址相同
FORWARD$_EX_B$	寄存器文件读取 rt 地址与上一条指令 alu 运算结果写寄存器地址相同
FORWARD_MEM$_A$	寄存器文件读取 rs 地址与上上条指令写寄存器地址相同
FORWARD_MEM$_B$	寄存器文件读取 rt 地址与上上条指令写寄存器地址相同

表 3: 前项通路信号含义及条件

2.3.2 算术逻辑运算控制单元 (ALUCtr) 的设计

本实验中我们将用于逻辑位移指令的 Shift(Shamt) 控制信号的设置放在了算术逻辑运算控制单元，这是因为解析逻辑位移指令所用到的 funct 域在该阶段被读取解析，而该模块的输入（opCode 域以及 aluOp 信号）都来自于 ID/EX 段寄存器保存的上一周期的值。

并且这里输出信号为四位功能信号 ALUCtrOut，该信号取值与 ALU 功能对应关系如表4所示。

ALUCtrOut 取值	ALU 功能
0000	与运算
0001	或运算
0010	加法运算
0011	逻辑左移运算
0100	逻辑右移运算
0101	jr 操作运算
0110	减法运算
0111	小于时置位

表 4: ALUCtrOut 信号含义

R 型指令	funct 域	ALUCtrOut
and	100100	0000
or	100101	0001
add	100000	0010
sll	000000	0011
srl	000010	0100
jr	001000	0101
sub	100010	0110
slt	101010	0111

表 5: R 型指令对应 ALUCtrOut 输出

在 R 型指令的判断部分，利用指令 funct 域进行分类，对 ALUCtrOut 输出值进行更新，同时，当控

制器判定该指令为 jr 指令时，对应的 Jr 控制信号置为 1，当控制器判定该指令为 sll 或 srl 指令（即为逻辑位移指令）时，对应的 Shift(Shamt) 控制信号置为 1。R 型指令判断对应关系如表5所示。

2.3.3 ALU 的设计

ALU 部分与单周期处理器操作一致，但是参与计算的两个输入来源有多种考量，与控制信号之间关系如表7所示。

控制信号状态	input1 来源
Shift(Shamt)= 1	ID/EX 段寄存器中指令 Shamt 域
FORWARD_EX_A= 1	EX/MEM 段寄存器中上条指令的 ALU 计算值
FORWARD_MEM_A= 1	MEM/WB 段寄存器中上上条指令的写寄存器值
无状态	ID/EX 段寄存器中读寄存器文件输出 1
控制信号状态	input2 来源
ID_EX_ALUSRC= 1	ID/EX 段寄存器中立即数
FORWARD_EX_B= 1	EX/MEM 段寄存器中上条指令的 ALU 计算值
FORWARD_MEM_B= 1	MEM/WB 段寄存器中上上条指令的写寄存器值
无状态	ID/EX 段寄存器中读寄存器文件输出 2

表 6: ALU 输入来源

2.3.4 写寄存器地址选择器的设计

该模块功能就是从指令对应的 rt 寄存器以及 rd 寄存器二者中选取一个作为写寄存器的目标地址，选择控制信号为 RegDst 信号，其余细节与之前实验数据选择器相同，不再赘述。

2.4 访存阶段 (MEM) 的设计

本阶段仅包含数据存储器一个模块，功能即为对数据存储器的读写，控制信号以及访存地址均来源于 EX/MEM 段寄存器。

2.4.1 数据存储器 (Data Memory) 的设计

该部分设计与之前实验中内容完全一致，数据来源也已声明，叙述不再展开。

2.5 写回阶段 (WB) 的设计

2.5.1 写寄存器数据选择器的设计

该模块功能是从访存阶段读取的数据以及 ALU 计算结果二者中选取一个作为写寄存器的操作数，选择控制信号为 MemToReg，其余细节与之前实验数据选择器相同，不再赘述。

2.6 顶层模块 (Top) 的设计

顶层模块的功能是在功能仿真前对所有部件进行实例化组装，将上述的所有部件模块输入输出端利用本模块中定义的 wire 接线以及 reg 变量连接起来，另外在本模块中在每个时钟周期上升沿对各模块进行一次更新，主要是对 IF/ID、ID/EX、EX/MEM、MEM/WB 四个段寄存器以及 PC 寄存器输入端读取的数据进行一次更新，同时在 IF/ID、ID/EX 段寄存器更新后判定是否存在前面两条指令为转移指令的情况，即判定 Jump、Jr、Branch 三个控制变量是否为高电平状态，如果是肯定答案，则将 IF/ID、ID/EX 段寄存器清零，所有其中的控制信号也都清零，这样就保证了跳转指令之后没有执行的指令不会对存储的数据造成影响；在 EX/MEM 段寄存器更新后判定是否存在前两条指令为 Beq 指令或 Jr 指令情况，如果结果为肯定，则应当将 EX/MEM 段寄存器清零，目的与上述段寄存器清零相同。除此之外，还包含重置操作部分，当收到 reset 信号为高电平状态，该模块将所有寄存器（包括 PC 寄存器，各段寄存器等），而寄存器文件等的重置操作则在各模块中具体实现。

3 功能实现

3.1 取指阶段 (IF) 的实现

3.1.1 PC 寄存器的实现

该模块实现在顶层模块中，仅由一个寄存器变量进行存储，在时钟周期上升沿进行更新即可，代码不再展示。

3.1.2 指令寄存器 (Instruction Memory) 的实现

由于指令计算以字 (Byte) 为单位，而指令存储在实现中我设置了 4Byte 对齐（即每次取值都为 32 位二进制编码），所以我们在进行指令地址换算时直接除 4，并且将整个 32 位指令编码送入输出端，实现代码如下：

```
reg[31:0] instFile[0:1023];
always @(instAddr) begin
    inst = instFile[instAddr/4];
end
```

3.2 译码阶段 (ID) 的实现

3.2.1 主控制器 (Ctr) 的实现

主控制器的功能是利用指令 OpCode 域进行指令操作解析，可以利用 Verilog 中 case 语句进行实现，R 型指令部分代码如下，为了方便这里我们选择将 Jr 指令的判定也一并完成，所以在这里用到了 funct 域，当然也可以将 Jr 指令的判定以及信号控制输出置于下一阶段的算术逻辑运算控制单元的判定运算类型中一并操作，具体详见工程文件：

```
 always @(opCode or funct) begin
    case(opCode)
    6'b000000: //R type
     begin
```

```
        RegDst = 1;
        ALUSrc = 0;
        MemToReg = 0;
        RegWrite = 1;
        MemRead = 0;
        MemWrite = 0;
        Branch = 0;
        ALUOp = 2'b10;
       Jump = 0;
       if(funct==6'b001000)
            Jr = 1;
        else
            Jr = 0;
    end
end
```

3.2.2 有符号拓展模块 (Signal Extend) 的实现

本模块功能就是将符号位填满前 16 位从而形成一个 32 位数，实现代码如下：

```
always @(data_in) begin
if(data_in[15])
   data_out = data_in | 32'hffff0000;
else
    data_out = data_in | 32'h00000000;
end
```

3.2.3 寄存器文件 (Register) 的实现

寄存器模块读寄存器文件的输出始终保持循环，而写寄存器文件的操作仅在时钟周期下降沿进行，还需要具有重置功能，实现代码如下：

```
always @(*) begin
   readData1 = regFile[readReg1];
   readData2 = regFile[readReg2];
end
always @(negedge clk) begin
   if(regWrite) begin
       regFile[writeReg] <= writeData;
   end
end
always @(reset) begin
   for(i=0;i<32;i=i+1)
       regFile[i] = 0;
```

```
end
```

3.3 执行阶段 (EX) 的实现

3.3.1 前项通路 (Forwarding) 的实现

本模块的实现仅仅是相关控制变量的计算，实现代码如下：

```
wire FORWARD_EX_A = (EX_MEM_REGWRITE & EX_MEM_regd==ID_EX_inst_rs & (EX_MEM_regd!=0)
    );
wire FORWARD_EX_B = (EX_MEM_REGWRITE & EX_MEM_regd==ID_EX_inst_rt & (EX_MEM_regd!=0)
    );
wire FORWARD_MEM_A = (MEM_WB_REGWRITE & MEM_WB_regd==ID_EX_inst_rs & MEM_WB_regd!=0)
     & !(EX_MEM_REGWRITE & EX_MEM_regd==ID_EX_inst_rs & EX_MEM_regd!=0);
wire FORWARD_MEM_B = (MEM_WB_REGWRITE & MEM_WB_regd==ID_EX_inst_rt & (MEM_WB_regd
    !=0)) & !(EX_MEM_REGWRITE & EX_MEM_regd==ID_EX_inst_rt & (EX_MEM_regd!=0));
```

3.3.2 算术逻辑运算控制单元 (ALUCtr) 的实现

本模块根据当前指令字的 funct 域对 ALUCtrOut 输出进行赋值，同时判断是否为逻辑位移操作对 Shift(Shamt) 信号电平进行修改，这里只展示部分实现代码：

```
always @({aluOp,funct} or opCode) begin
    shamt = 0;
    casex({aluOp,funct})
        8'b10100000: aluCtrOut = 4'b0010;
        8'b10100010: aluCtrOut = 4'b0110;
        8'b10100100: aluCtrOut = 4'b0000;
        8'b10100101: aluCtrOut = 4'b0001;
        8'b10101010: aluCtrOut = 4'b0111;
        8'b10000000: begin
             aluCtrOut = 4'b0011;
           shamt = 1;
        end
        8'b10000010: begin
             aluCtrOut = 4'b0100;
           shamt = 1;
        end
        8'b10001000: aluCtrOut = 4'b0101;
    endcase
    ...
    ...
end
```

3.3.3 ALU 的实现

本部分实现与单周期处理器中完全一致，代码具体见工程文件。

3.3.4 写寄存器地址选择器的实现

本模块两路输出根据选择控制信号选择一路输出，可以利用 if-else 语句实现，代码如下：

```
always @(signal or input1 or input2) begin
if(signal)
    outdata = input1;
else
    outdata = input2;
end
```

3.4 访存阶段 (MEM) 的实现

3.4.1 数据存储器 (Data Memory) 的实现

本模块实现与单周期处理器实现完全一致，具体代码见工程文件。

3.5 写回阶段 (WB) 的实现

3.5.1 写寄存器数据选择器的实现

数据选择器在选择控制信号的控制下对两个输入值之一进行选择并送入输出端，直接利用 Verilog 中 if-else 语句实现，实现代码如下：

```
always @(input1 or input2 or signal) begin
    if(signal == 0)
        output_data = input1;
    else
        output_data = input2;
end
```

3.6 顶层模块 (Top) 的实现

顶层模块中对其他各模块的实例化就不一一在这里展示，仅展示每个时钟周期上升沿对流水段寄存器以及 PC 寄存器的操作，代码如下：

```
 always @(posedge clk) begin
        MEM_WB_aluRes <= EX_MEM_aluRes;
        MEM_WB_memData <= MEM_READ_DATA;
        MEM_WB_regd <= EX_MEM_regd;
        MEM_WB_ctr <= EX_MEM_ctr[1:0];

        EX_MEM_aluRes <= ALU_RES;
```

```
8         EX_MEM_writeData <= MEM_WRITE_DATA;
9         EX_MEM_regd <= REGD;
10        EX_MEM_ctr <= ID_EX_ctr[4:0];
11
12        if(!STALL & !BRANCH & !JR) begin
13        ID_EX_branchshift <= IMM_SIGNEXT_SHIFT;
14        ID_EX_inst <= IF_ID_inst;
15        ID_EX_imm <= IMM_SIGNEXT;
16        ID_EX_readData1 <= REG_READ_DATA1;
17        ID_EX_readData2 <= REG_READ_DATA2;
18        ID_EX_inst_rs <= IF_ID_inst_rs;
19        ID_EX_inst_rt <= IF_ID_inst_rt;
20        ID_EX_inst_rd <= IF_ID_inst_rd;
21        ID_EX_ctr <= CTR_OUT[8:0];
22        ID_EX_shamt <= IF_ID_inst[10:6];
23        ID_EX_pcp <= IF_ID_pcp;
24        ID_EX_JR <= CTR_OUT[10];
25        end
26        else begin
27        ID_EX_branchshift <= 0;
28        ID_EX_ctr <= 0;
29        ID_EX_inst <= 0;
30        ID_EX_imm <= 0;
31        ID_EX_readData1 <= 0;
32        ID_EX_readData2 <= 0;
33        ID_EX_inst_rs <= 0;
34        ID_EX_inst_rt <= 0;
35        ID_EX_inst_rd <= 0;
36        ID_EX_shamt <= 0;
37        ID_EX_pcp <= 0;
38        ID_EX_JR <= 0;
39        end
40
41        if(!STALL) begin
42            PC <= NEXT_PC;
43            if(!JUMP & !BRANCH & !JR) begin
44            IF_ID_pcp <= PCP;
45            IF_ID_inst <= IF_INST;
46            end
47        end
48        if(JUMP) begin
```

```
            IF_ID_inst <= 0;
            IF_ID_pcp <= 0;
        end
        if(BRANCH) begin
            IF_ID_inst <= 0;
            IF_ID_pcp <= 0;
        end
        if(JR) begin
            IF_ID_inst <= 0;
            IF_ID_pcp <= 0;
        end
    end
```

4 结果验证（单周期类 MIPS 处理器的测试）

对于本实验中实现的简单的类 MIPS 多周期流水线处理器，我们采用软件仿真的方法进行测试，用 Verilog 语言编写测试激励文件，进行仿真，主要观察寄存器以及存储器信号波形变化，并通过我在各模块中内置的一些输出语句（$display(...);），观察 Tcl Console 输出框内的输出，比对是否符合预期。

在仿真激励文件中，我们通过 Verilog 指令直接将指令存储器文件以及数据存储器文件读入到对应存储位置，这里我们给出测试用指令文件以及对应的测试内容，如表7所示。

指令字	具体含义
10001100000000010000000000001010	lw $1, 10($0)
10001100000000010000000000001011	lw $2, 11($0)
00000000001000100001100000100000	add $3, $1, $2
00000000001000100010000000100010	sub $4, $1, $2
00000000001000100010100000100100	and $5, $1, $2
00000000001000100011000000100101	or $6, $1, $2
00000000001000100101000000101010	slt $10, $1, $2
00000000000000001010110001000000	sll $11, $1, 2
00000000000000001011000010000010	srl $12, $1, 2
00001000000000000000000000001011	j 11
00000011111100000000000000001000	jr $31
00001100000000000000000000001100	jal 12
10101100000000010000000000001010	sw $1,10($0)
00010011111111110000000000000000	beq $31, $30, 0
00100000001001110000000000000010	addi $7, $1, 2
00110000001010000000000000000010	andi $8, $1, 2
00110100001010010000000000000010	ori $9, $1, 2

表 7: 指令字及对应含义

根据如上述指令字序列执行仿真程序得到的程序输出如图1所示；仿真波形结果如图2、图3所示。

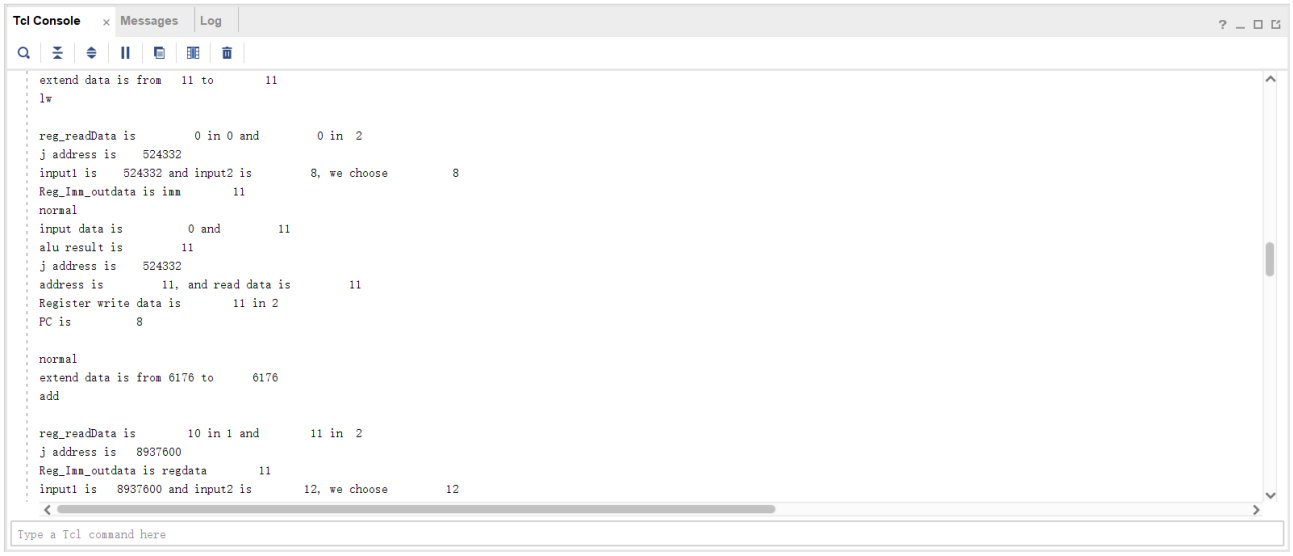

图 1: 仿真程序部分输出

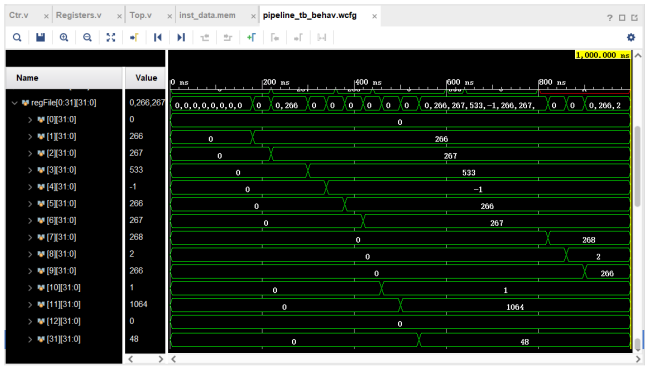

图 2: 多周期流水线处理器仿真波形-1

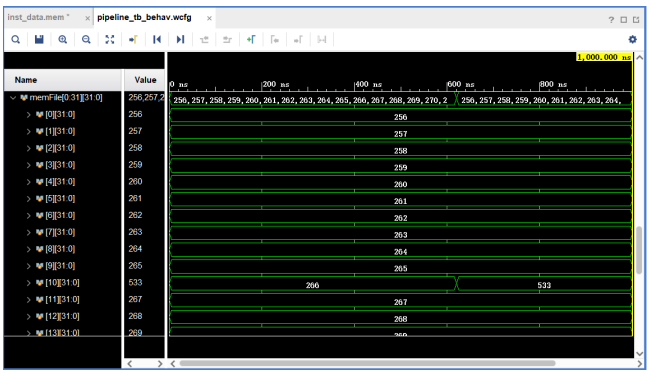

图 3: 多周期流水线处理器仿真波形-2

另外想要测试 jr 指令，仅需将 jump 指令跳转地址由 44 改为 40（即将指令字中对应的二进制编码 1011 改为 1010），这样修改以后由于 jr 指令对应寄存器值为零，所以 jr 指令之后的所有指令无法执行，修改后的仿真程序输出如图4所示；仿真波形结果如图5、图6所示。

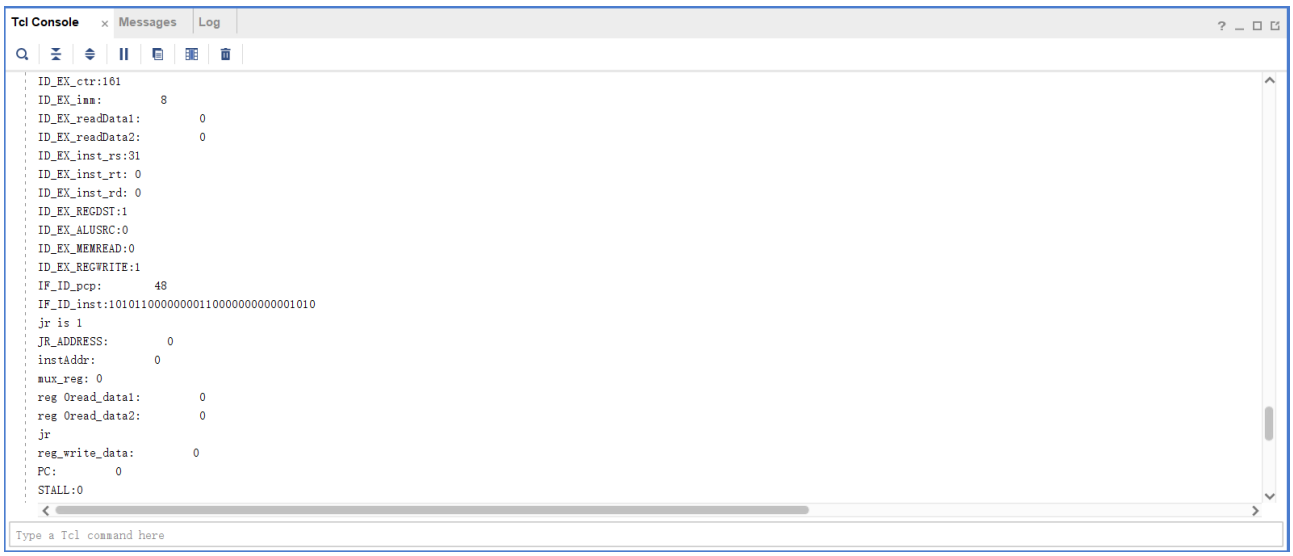

图 4: 修改后仿真程序部分输出

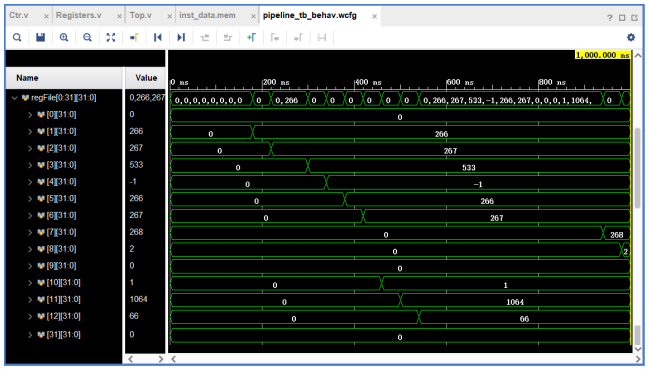

图 5: 多周期流水线处理器仿真波形-3

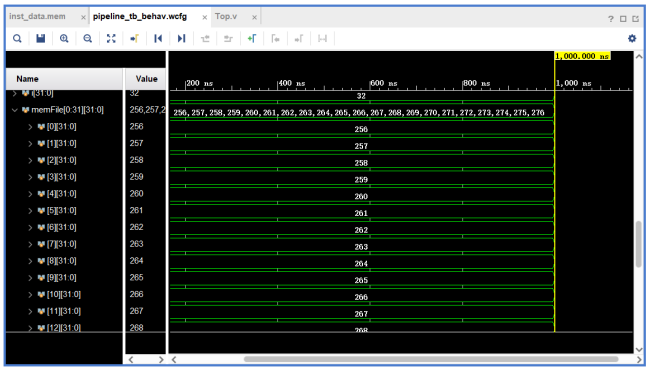

图 6: 多周期流水线处理器仿真波形-4

比照以上六张图以及程序的完整输出逻辑，验证仿真实验功能完全成功。

5 总结与反思

本实验完整实现了支持 16 指令的类 MIPS 多周期处理器的仿真，通过由顶层模块对其余九个各阶段执行模块的组装接线，以及每个时钟周期各流水段寄存器的赋值输入来源的了解，让我对流水线处理器的运作过程有了更加深刻的理解，另外，在本次实验中，对于要求 2、3、4 作为一个整体进行设计，使得我对于流水线执行过程中的各种竞争冒险（数据冒险、控制冒险和结构冒险）更加熟悉，并且通过设计前项通路 (forwarding) 以及 predicate-not-take 机制对于解决数据竞争、减少因数据竞争带来的流水线停顿延时、解决控制冒险、减少控制竞争带来的流水线停顿延时，提高了流水线处理器的性能。但是在实验中也还有许多不足之处，例如在主控制器的设计与实现中可以不用 funct 域，而是将 Jr 指令的解析延后至算术逻辑运算控制单元中进行解析；并且在算术逻辑运算控制单元中也无需 Opcode 域的参与，对于非 R 型指令的 ALU 操作仅通过 aluOp 控制信号即可识别，只是在本次实验中我们为了方便并没有这样去做。由于单周期处理器在设计过程中两模块合并的原因，在本次实验开始阶段尝试对实验 5 的模块进行修改但是没有成功，迫不得已将所有模块重新实现，但是就是这个过程中，我对于各模块的功能实现更加清晰，变量命名也有了更多的意义，整体编程 Verilog 语言的能力有了一定的提升。同时在这次实验的调试过程中对于波形图的操作更加熟练（终于学会了不同变量

的波形显示不同进制数值的功能），时间不太足够就没来得及实现 16 指令拓展到 32 指令的功能，有机会自己会尝试。

6 鸣谢

感谢本次实验中实验指导老师对一些问题的建议与帮助，中间远程桌面出问题也是在老师积极帮助下解决，同时感谢系统结构课程老师帮助我对于流水线中数据通路等问题的进一步加深理解和提出的建设性指导，感谢上海交通大学网络信息中心提供的远程桌面资源。