计算机系统结构实验报告 Lab06

简单的类 MIPS 多周期流水化处理器实现

徐阳　521021910363

2023 年 5 月 3 日

目录

1 简介

在本实验中，以实验 5 实现的简单的类 MIPS 单周期处理器为基础，我理解并实现了类 MIPS 多周期流水线处理器。该处理器支持的 MIPS 指令由 16 条扩充为 31 条。对原来的简单处理器各部分进行了修改以适应流水线结构，并加入了段寄存器。

在此基础上，加入了检测竞争并停顿的 Stall 机制解决数据冒险/竞争、控制冒险和结构冒险。还实现了前向通路 (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，同时通过了预测不转移 (predict-not-taken) 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。最后我进行整体调试并通过行为仿真验证了程序的正确性。

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 9 条或 16 条扩充为 31 条，使处理器功能更加丰富（选做）
6. 掌握测试汇编代码和激励文件的编写，使用行为仿真验证程序正确性

3 各模块实验原理与功能实现

在实现流水线之前，为了将指令扩充到 31 条，需要对处理器中的各模块做一些修改。本实验实现的全部 31 条指令的格式参考于老师在微信群发布的“Mips 指令集.html”文件，其可在源文件中查看。

3.1 主控制器单元 Ctr

为了实现 31 条指令的流水线处理器，我们将主控制器单元 Ctr 的输入改为为指令的高六位 OpCode 和低六位 funct 还有空指令信号 nop。这样加入了 funct 之后在 Ctr

部分便能将 I 型指令和 J 型指令和 R 型指令中的 jr 指令的控制信号全部解析完全，生成对应的控制信号和四位 ALUOp。

提前解析控制信号的目的是为了提前知道是否有转移指令，为之后的预测不转移做准备。这样在 ALUCtr 元件处只要处理除 jr 外的 R 型指令就行了，其余指令的控制信号和 ALUOp 只需要传递就好。

这样以来在 Ctr 处的输出控制信号就变为一下几种：目标寄存器的选择信号 regDst、寄存器写使能信号 regWrite、ALU 第二个操作数的来源 ALUSrc、写寄存器的数据来源 memToReg、内存读使能信号 memRead、内存写使能信号 memWrite、无条件跳转信号 Jump、符号拓展信号 extSign、jal 指令信号 JalSign、jr 指令信号 jrSign 条件跳转 beq 指令信号 beqSign、条件跳转 bne 指令信号 bneSign、lui 指令信号 luiSign。最后还有发送给 ALUCtr 进一步解析的控制信号——四位的 ALUOp。

由于指令数量的增多，在这里特别给出 ALUOp 和指令类型的对应关系如表 1所示，剩余控制信号的解码详见 Ctr.v 文件。

表 1: ALUop 与指令的对应及作用

ALUOp	指令	ALU 运算类型
0000	add, addi	带溢出检查的加法
0001	addu, addiu, lw, sw	不带溢出检查的加法
0010	sub, subi	带溢出检查的减法
0011	subu, beq, bne	不带溢出检查的减法
0100	and, andi	逻辑与
0101	or, ori	逻辑或
0110	xor, xori	逻辑异或
0111	nor	逻辑或非
1000	slt, slti	带符号小于时置位
1001	sltu, sltiu	无符号小于时置位
1010	sll, sllv, lui	逻辑左移
1011	srl, srlv	逻辑右移
1100	sra, srav	算术右移
1101	无	不进行任何操作
1110	无	不进行任何操作
1111	j, jal, jr	不进行任何操作

具体地代码实现与实验 3、实验 5 中相同，同样是利用 case 语句进行判断，稍作修改。在这里碍于篇幅限制不再给出详细代码，具体完整代码可见于 Ctr.v。

3.2 运算单元控制器单元 ALUCtr

为了实现 31 条指令的流水线处理器，运算单元控制器单元 ALUCtr 输入改为 4 位 ALUOp 和指令低六位的 funct，输出仍为给 ALU 的运算控制信号 ALUCtrOut. 因为在 Ctr 中已经处理了众多指令，如果是已经处理过的指令 ALUCtr 只要传递 ALUOp 信号为 ALUCtrOut 就好，如果是 jr 之外的 R 型指令，则 ALUCtr 将进行进一步解码，ALUCtrOut 具体可参照之前给出的表 1。

除去 ALUCtrOut，在 ALUCtr 中还要判断如果指令为 sll，srl 或 sra 则 shamtSign 置一，其余情况为 0。

采用的 case 语句实现 ALUCtr，以下是代码的关键部分，完整代码见 ALUCtr.v。

——ALUCtr.v——

```
1    always @ (aluOp or funct)
2    begin
3        if (aluOp == 4'b1101 || aluOp == 4'b1110) begin
4            case (funct)
5                6'b100000:      // add
6                    aluCtrOut = 4'b0000;
7                6'b100001:      // addu
8                    aluCtrOut = 4'b0001;
9                // ......
10               6'b000111:      // srav
11                   aluCtrOut = 4'b1100;
12               6'b001000:      // jr
13                   aluCtrOut = 4'b1111;
14               default:
15                   aluCtrOut = 4'b1111;
16           endcase
17
18           if (funct == 6'b000000 || funct == 6'b000010 || funct == 6'b000011
     ) //sll, srl or sra
19               shamtSign = 1;
20           else
21               shamtSign = 0;
22       end else begin
23           aluCtrOut = aluOp;
24           shamtSign = 0;
25       end
26   end
```

3.3 算数逻辑运算单元 ALU

算数逻辑运算单元 ALU 的输入为两个 32 位操作数 input1 和 input2，以及运算控制信号 ALUCtr，输出为 zero 标志位和 32 位结果 aluRes。其根据 ALUCtr 的类型对两个操作数进行相应的算数逻辑运算，并将结果输出到 output，若运算结果为 0，则将 zero 标志位置一，否则其为零。

对比实验 5，我们加入了无符号小于时置位运算和逻辑右移运算，在 ALU 中，运算控制信号变成了 4 位，信号与运算对应的关系需要重新设计，ALUCtrOut 信号与 ALU 执行的运算类型的对应关系已由上述表 1展示。具体的代码详见 ALU.v。

3.4 寄存器 Registers

Registers 的原理与实验 5 中的完全相同，不再赘述。唯一需要改动的是加入了 jal 信号，在其为高电平时将相应数据写入寄存器最后一位，代码如下。

——Registers.v——

```
1    always @ (negedge clk)
2    begin
3        if(regWrite) begin
4            RegFile[writeReg] = writeData;
5        end
6        if (jalSign) begin
7            RegFile[31] = jalData;
8        end
9    end
```

3.5 数据存储器 dataMemory 与指令存储器 InstMemory

dataMemory 与 InstMemory 的原理与实现与实验 5 中的完全相同，不再赘述。

3.6 有符号扩展单元 signext

有符号扩展单元 signext 的原理与实现与实验 5 中的完全相同，不再赘述。

3.7 多路选择器 Mux

多路选择器 Mux 的原理与实现与实验 5 中的完全相同，不再赘述。

3.8 程序计数器 PC

在本次实验中由于在流水线的不同阶段有着不同的 PC, 难以用模块化实现, 故 PC 采取在 TOP.v 中实现。

3.9 顶层模块 Top

Top 的实现将在下一节“流水线”进行详细的分析。

4 流水线的原理与实现

流水线 CPU 大致可以拆为五个阶段，分别为 Instruction Fetch(IF)，Instruction Decode(ID)，Execute(EX)，Memory/Memory Access(MEM/MA)，Write Back(WB)。

流水线各阶段的任务

- IF：从 InstMemory 取指
- ID：对指令进行解码产生控制信号
- EX：通过 ALU 进行计算, 以及决定是否跳转
- MEM(MA)：访存，读写数据
- WB：把数据写回寄存器

在之前的单周期处理器中，控制信号在各个阶段有相互依赖的现象，很难将其直接拆分为五段。通过重新设计数据流和创建阶段寄存器可以很好的解决这个问题，但不是所有数据都要在段寄存器中传输，以下分析流水线的各个阶段设计的元件以确定段寄存器中存储的数据。

4.1 IF 阶段

IF 阶段在 InstMemory 中根据当前 PC 指向的地址取出一条指令，并传入下一个阶段。

4.2 ID 阶段

ID 阶段根据指令译码，进行位拓展，并产生相应的控制信号，该阶段涉及 Ctr、Registers 和有符号拓展单元。

4.3 EX 阶段

EX 阶段对指令进行更细致的解码并进行相应的算术逻辑运算，该阶段主要涉及 ALUCtr、ALU 以及一些控制信号控制的多路选择器。

4.4 MEM 阶段

MEM 阶段进行内存的访问，根据访存信号进行数据的读取和写入，主要涉及 dataMemory 和 ALU 元件，只有少部分指令会在这个阶段访存，这个阶段也是我们要进行前向通路判断的阶段。

4.5 WB 阶段

WB 阶段进行将数据写回寄存器，这可能会造成数据冒险，如果数据来不及写回寄存器，可等会导致后续指令取出错误的数据。

4.6 具体实现：Top.v

Top.v 是整个实验的核心，我们要精心设计数据的传输。各模块的实例化仍如实验 5 差不多，但各模块之间不再能直接连接。我们需要在每个阶段之间设置段寄存器以实现流水化。经分析，在各个段寄存器我们需要如下的数据如下，相应的定义也在下文给出。

段寄存器的内容

- IF-ID：当前指令和 PC
- ID-EX：符号拓展结果、各种控制信号、读寄存器的结果（ALU 的两个操作数）、当前指令和 PC
- EX-MEM：访存及写寄存器有关的控制信号、ALU 计算结果、rt 寄存器的值、目标寄存器的地址
- MEM-WB：写寄存器控制信号、写寄存器的数据、目标寄存器的地址

——Top.v——

```
1  // IF-ID
2  reg [31 : 0] IF2ID_PC;
3  reg [31 : 0] IF2ID_INST;
4  //ID-EX
5  reg [31 : 0] ID2EX_PC;
6  reg [3 : 0] ID2EX_CTR_SIGNAL_ALUOP;
```

```
reg [5 : 0] ID2EX_INST_FUNCT;
reg [4 : 0] ID2EX_INST_SHAMT;
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_DATA_1;
reg [31 : 0] ID2EX_REG_READ_DATA_2;
reg [4 : 0] ID2EX_REG_DEST;
// EX-MEM
reg [3 : 0] EX2MEM_CTR_SIGNALS;
reg [31 : 0] EX2MEM_ALU_RES;
reg [31 : 0] EX2MEM_WRITE_DATA;
reg [4 : 0] EX2MEM_REG_DEST;
// MEM-EX
reg MEM2WB_CTR_SIGNALS;
reg [31 : 0] MEM2WB_FINAL_DATA;
reg [4 : 0] MEM2WB_REG_DEST;
```

有了段寄存器，接着要实现流水线的更新。我们需要连接各器件，借助时钟信号 clk 传输数据。在每个时钟的下降沿，我们将每个阶段的数据传向段寄存器，并从前一部分的段寄存器中读取数据，具体地代码实现如下所示。我们采用非阻塞赋值，同时为所有的数据赋值。这里我们已经用到了 stall 信号和 branch 信号，它们是停顿机制和预测不转移机制产生的，具体的原理将在之后提及。

——Top.v——

```
always @(posedge clk)
begin
  // IF-ID
  if (! STALL) begin
    IF_PC <= PC_BNE;
  end
  if (BRANCH || ID_CTR_SIGNALS[12] || ID_CTR_SIGNALS[11]) begin
    IF2ID_INST <= 0;
    IF2ID_PC <= 0;
  end
  else if (!STALL)
  begin
    IF2ID_INST <= IF_INST;
    IF2ID_PC <= IF_PC;
```

```
    end
    // ID-EX
    if (STALL || BRANCH) begin
      ID2EX_PC <= IF2ID_PC;
      ID2EX_CTR_SIGNAL_ALUOP <= 4'b1111;
      ID2EX_CTR_SIGNALS <= 0;
      ID2EX_EXT_RES <= 0;
      ID2EX_INST_RS <= 0;
      ID2EX_INST_RT <= 0;
      ID2EX_REG_READ_DATA_1 <= 0;
      ID2EX_REG_READ_DATA_2 <= 0;
      ID2EX_INST_FUNCT <= 0;
      ID2EX_INST_SHAMT <= 0;
      ID2EX_REG_DEST <= 0;
    end
    else begin
      ID2EX_PC <= IF2ID_PC;
      ID2EX_CTR_SIGNAL_ALUOP <= ID_CTR_SIGNAL_ALUOP;
      ID2EX_CTR_SIGNALS <= ID_CTR_SIGNALS [7 : 0];
      ID2EX_EXT_RES <= ID_EXT_RES;
      ID2EX_INST_RS <= IF2ID_INST [25 : 21];
      ID2EX_INST_RT <= IF2ID_INST [20 : 16];
      ID2EX_REG_READ_DATA_1 <= ID_REG_READ_DATA_1;
      ID2EX_REG_READ_DATA_2 <= ID_REG_READ_DATA_2;
      ID2EX_INST_FUNCT <= IF2ID_INST [5 : 0];
      ID2EX_INST_SHAMT <= IF2ID_INST [10 : 6];
      ID2EX_REG_DEST <= ID_REG_DEST;
    end
    // EX-MEM
    EX2MEM_CTR_SIGNALS <= ID2EX_CTR_SIGNALS [3 : 0];
    EX2MEM_ALU_RES <= EX_ALU_RES;
    EX2MEM_WRITE_DATA <= RT_AFTER_FORWARDING;
    EX2MEM_REG_DEST <= ID2EX_REG_DEST;
    // ME-WB
    MEM2WB_CTR_SIGNALS <= EX2MA_CTR_SIGNALS [0];
    MEM2WB_FINAL_DATA <= MA_FINAL_DATA;
    MEM2WB_REG_DEST <= EX2MA_REG_DEST;
  end
```

另外我们需要在 Top.v 中实现 reset 功能，在 reset 被置为高电平时，我们清空所

有段寄存器，并将 IF 阶段的 PC 重新初始化。

至此，完整的流水线处理器已经被我们实现，接下来要进行的是对 Hazard 的处理与优化。

5 流水线的优化——Hazard 的处理

5.1 Stall

所谓停顿（STALL）是指在 ID 阶段解码到要取的数据（rt 或 rd）时，如果该目标数据也被上一条指令（正在 EX 阶段）使用并且要在 MEM 阶段访存，那么该数据很有可能会取到错误的、没有更新的数据，故要将正在 EX 阶段的指令之后的指令都停顿一周期，以避免 Hazard。具体的代码上，SATLL 信号只需要一个逻辑表达式即可，具体的代码实现如下。

——Top.v——

```
1  // Stall if load rd or rt from memory
2  wire STALL = ID2EX_CTR_SIGNALS[2]
3  &   ((ID2EX_INST_RT == IF2ID_INST [25 : 21]) | (ID2EX_INST_RT == IF2ID_INST
     [20 : 16]))
4  &   (IF2ID_INST[31 : 26] != 6'b100011);
```

但是需要特别注意的是，如果 ID 解码出的指令是 lw，那么是不需要停顿的，因为 lw 指令不会使用寄存器中的指令，而是从内存中取数据覆盖目标地址上的数据，在这之前该地址的数据如何变化对 lw 的正确性都不会有影响。这是一个优化的小细节，可以避免连续从内存读数据时产生不必要的停顿。

5.2 Forwarding

所谓前向通路（Forwarding），就是在当前指令准备读取 rs 或 rt 对应地址中的数据时检查该数据有没有在 EX、MEM 阶段出现。如果出现，因为 EX、MEM 阶段的数据都是已经计算完成，不会发生改变，可以将数据直接拿来给当前指令使用而覆盖掉旧数据，从而避免之后发生停顿。

要注意的有两点。一是目标指令的 rs 和 rt 都要有前向通路。二是如果 EX、MEM 阶段都有可产生前向通路的数据时，显然我们需要 EX 阶段的数据——因为它是最新的，这就需要我们在更新数据时最后判断是否采取 EX 阶段的前向通路。具体地代码实现如下，已经满足了以上两点。

——Top.v——

```
1  // Forwarding
2  wire [31 : 0] RS_FORWARDING_TEMP;
```

```
3  wire [31 : 0] RT_FORWARDING_TEMP;
4  // rs的前向通路
5  Mux32 forward_rs_selector1(
6      .sel(MA2WB_CTR_SIGNALS & (MA2WB_REG_DEST == ID2EX_INST_RS)),
7      .input1(MA2WB_FINAL_DATA),
8      .input2(ID2EX_REG_READ_DATA_1),
9      .out(RS_FORWARDING_TEMP)
10     );
11 Mux32 forward_rs_selector2(
12     .sel(EX2MA_CTR_SIGNALS[0] & (EX2MA_REG_DEST == ID2EX_INST_RS)),
13     .input1(EX2MA_ALU_RES),
14     .input2(RS_FORWARDING_TEMP),
15     .out(RS_AFTER_FORWARDING)
16     );
17 // rt的前向通路
18 Mux32 forward_rt_selector1(
19     .sel(MA2WB_CTR_SIGNALS & (MA2WB_REG_DEST == ID2EX_INST_RT)),
20     .input1(MA2WB_FINAL_DATA),
21     .input2(ID2EX_REG_READ_DATA_2),
22     .out(RT_FORWARDING_TEMP)
23     );
24 Mux32 forward_rt_selector2(
25     .sel(EX2MA_CTR_SIGNALS[0] & (EX2MA_REG_DEST == ID2EX_INST_RT)),
26     .input1(EX2MA_ALU_RES),
27     .input2(RT_FORWARDING_TEMP),
28     .out(RT_AFTER_FORWARDING)
29     );
```

5.3 predict-not-taken 与 PC 的更新

所谓预测不转移（predict-not-taken），就是在解码阶段如果是 branch 信号就猜测要执行转移指令，根据解码出的控制信号计算出可能转移的 PC 地址。在 EX 阶段如果预测出错，则产生 BRANCH 信号，清空流水线之后的指令。

预测不转移是一种静态预测，不根据历史的转移情况来进行预测，因此准确性不会很高，但是他也没有任何时间开销，可以提高流水线的效率。

在预测不转移（predict-not-taken）之外，我们还需要根据 jump、jr 指令更新我们的 PC，具体的代码实现如下所示。

——Top.v——

```
wire BEQ_BRANCH = ID2EX_CTR_SIGNALS[5] & EX_ALU_ZERO;
wire BNE_BRANCH = ID2EX_CTR_SIGNALS[4] & (~ EX_ALU_ZERO);
wire [31 : 0] BRANCH_DEST = ID2EX_PC + 4 + (ID2EX_EXT_RES << 2);
wire [31 : 0] PC_JUMP;
wire [31 : 0] PC_JR;
wire [31 : 0] PC_BEQ;
wire [31 : 0] PC_BNE;
wire BRANCH = BEQ_BRANCH | BNE_BRANCH;

Mux32 jump_selector(
    .sel(ID_CTR_SIGNALS[12]),
    .input1(((IF2ID_PC + 4) & 32'hf0000000) + (IF2ID_INST [25 : 0] << 2)),
    .input2(IF_PC + 4),
    .out(PC_JUMP)
    );

Mux32 jr_selector(
    .sel(ID_CTR_SIGNALS[11]),
    .input1(ID_REG_READ_DATA_1),
    .input2(PC_JUMP),
    .out(PC_JR)
    );

Mux32 beq_selector(
    .sel(BEQ_BRANCH),
    .input1(BRANCH_DEST),
    .input2(PC_JR),
    .out(PC_BEQ)
    );

Mux32 bne_selector(
    .sel(BNE_BRANCH),
    .input1(BRANCH_DEST),
    .input2(PC_BEQ),
    .out(PC_BNE)
    );
```

6 仿真验证

编写激励文件，进行仿真验证, 代码如下。在 innitial 模块中利用 $readmemh 和 $readmemb 分别读取十六进制和二进制文件, 初始化内存数据 memFIle 和指令 instFile。

激励文件的代码实现与实验五基本一致，因为实验五和实验六中 Top 模块给出的接口是一样的。

我进行了两次仿真验证，分别验证了老师和助教给出的测试样例和自己编写的测试 31 条指令的实现的样例，下面分别给出它们的具体样例和仿真结果。

6.1 老师和助教给出的样例

——data.txt——

```
0
0
0
0
0
0
0
0
0
0
1
5
8
```

——汇编指令内容——

```
lw $1, 40($0) ; 1
lw $2, 44($0) ; 5
lw $3, 48($0) ; 8
add $4, $1, $2 ; $4=6
sub $5, $3, $1 ; $5=7
and $6, $2, $1 ; $6=1
lw $10, 40($0) ; 1
lw $10, 40($0) ; 1
lw $10, 40($0) ; 1
or $7, $3, $1 ; $7=9
slt $8, $3, $1 ; $8=0
beq $0, $0, end ; to end
```

```
13 add $9, $7, $8 ; $9=9, not executed
14 end:
15 lw $10, 40($0) ; 1
16 lw $10, 40($0) ; 1
17 lw $10, 40($0) ; 1
18 lw $10, 40($0) ; 1
19 lw $10, 40($0) ; 1
```

——inst.txt——

```
1 10001100000000010000000000101000
2 10001100000000010000000000101100
3 10001100000000011000000000110000
4 00000000001000100010000000100000
5 00000000011000010010100000100010
6 00000000010000010011000000100100
7 10001100000010100000000000101000
8 10001100000010100000000000101000
9 10001100000010100000000000101000
10 00000000011000010011100000100101
11 00000000011000010100000000101010
12 00010000000000000000000000000001
13 00000000111010000100100000100000
14 10001100000010100000000000101000
15 10001100000010100000000000101000
16 10001100000010100000000000101000
17 10001100000010100000000000101000
18 10001100000010100000000000101000
```

仿真结果如图 1所示，可见结果与助教给出的 PDF 中结果一致，前向通路和预测不转移也作用的很好，并没有停顿产生。

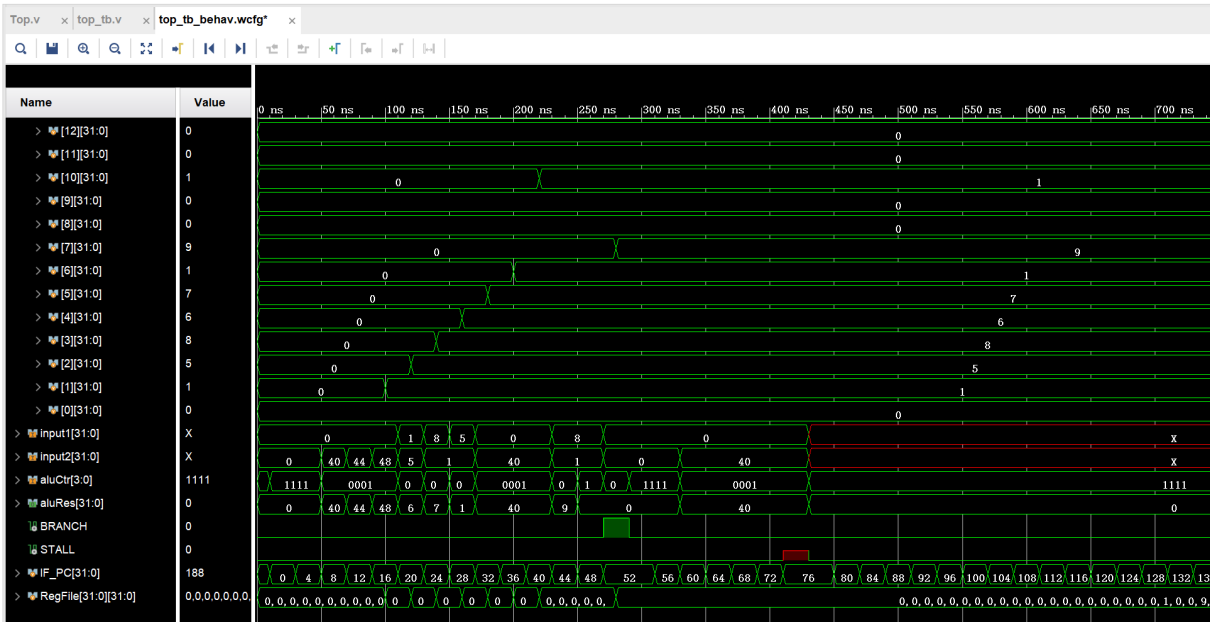

图 1: 老师和助教给出的样例

6.2 验证实现 31 条指令的样例

为了验证我实现 31 条指令，我自己编写了如下的汇编代码，并给出了代码的预期结果。

仿真结果如图 2，可见仿真结果和预测结果符合的很好，我们实现了简单的类 MIPS 多周期流水化处理器。

——data2.txt——

```
0
0
0
0
0
0
0
0
0
0
1
5
5
```

——inst2.txt——

```
1  10001100000000010000000000101000 //lw $1 = 1    (0001)
2  10001100000000010000000000101100 //lw $2 =5 (0101)
3  10001100000000110000000000110000 //lw $3 = 5    (0101)
4  00000000001000100010000000100000 //add $4, $1, $2   ($4 = 6)
5  00000000001000100010000000100001 //addu $4, $1, $2  ($4 = 6)
6  00000000001000100010000000100010 //sub $4, $1, $2   ($4 = -4)
7  00000000001000100010000000100011 //subu $4, $1, $2  ($4 = -4)
8  00000000001000100010000000100100 //and $4, $1, $2   ($4 = 1)
9  00000000001000100010000000100101 //or $4, $1, $2    ($4 = 5)
10 00000000001000100010000000100110 //xor $4, $1, $2   ($4 = 0b01000 = 4)
11 00000000001000100010000000100111 //nor $4, $1, $2   ($4 = 0b11010 = -6)
12 00000000001000100010000000101010 //slt $4, $1, $2   ($4 = 1)
13 00000000001000100010000000101011 //sltu $4, $1, $2  ($4 = 1)
14 00000000000000010001000010000000 //sll $4 = $2 << 2 ($4 = 20)
15 00000000000000010001000010000010 //srl $4 = $2 >> 2 ($4 = 1)
16 00000000000000010001000010000011 //sra $4 = $2>>>2($4 = 1)
17 00000000001000100010000000000100 //sllv $4, $2, $1  ($4 = $2 << $1 = 10)
18 00000000001000100010000000000110 //srlv $4, $2, $1  ($4 = $2 >> $1 = 2)
19 00000000001000100010000000000111 //srav $4, $2, $1  ($4 = $2 >>> $1 = 2)
20 00100000001001000000000000000111 //addi $4, $1, 7   ($4 = $1 + 7 = 8)
21 00100100001001000000000000000111 //addiu $4, $1, 7  ($4 = $1 + 7 = 8)
22 00110000001001000000000000000111 //andi $4, $1, 7   ($4 = $1 & 7 = 1)
23 00110100001001000000000000000111 //ori $4, $1, 7    ($4 = $1 or 7 = 7)
24 00111000001001000000000000000111 //xori $4, $1, 7   ($4 = $1 xor 7 = 6)
25 00010000010000110000000000000000 //beq $2, $3, 0 (if $2 = $3, PC = PC +4 +imm)
26 00111100000001000000000000000111 //lui $4, 1    ($4 = 7 * 65536 = 458752)
27 00010100010000110000000000000000 //beq $2, $3, 0 (if $2 != $3,PC = PC +4 +imm)
28 00101000001001000000000000000111 //slti $4, $1, 7   (if $1 < imm, $4 = 1)
29 00101100001001000000000000000000 //slti $4, $1, 7   (if $1 < imm, $4 = 1)
30 00001100000000000000000000000001 //jal 1 (PC = (PC +4)[31 : 28] + imm*4, $31 =
       PC+4)
```

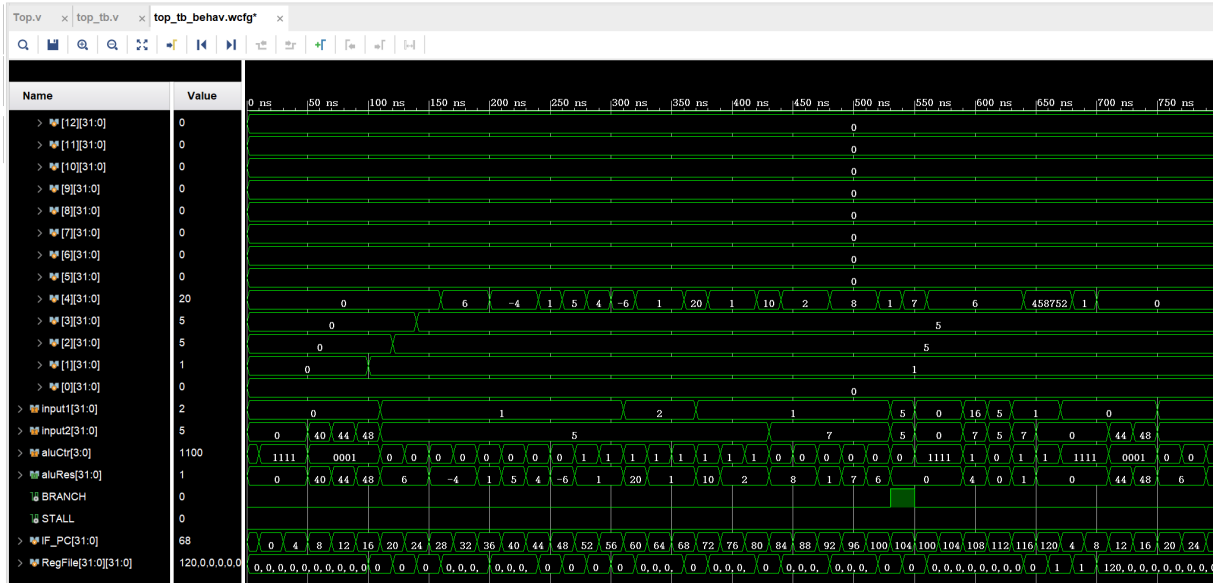

图 2: 验证实现 31 条指令的样例

7 总结与思考

在本实验中，我实现了一个支持 31 条指令的类 MIPS 多周期流水线处理器，完整的实现了流水线的功能，并实现了前向通路、停顿、预测不转移的功能，提高了处理器的性能。

在实验中，我对流水线有了更加深刻的认识，对如何消除处理器中的（Hazard）也有了更深的理解。在系统结构课上学到的比较抽象的知识也得到了具体地实现，让我感受到了它们在消除 Hazard 时的作用。通过代码实现这些功能，虽然工程量非常大，但任务完成调试成功时也给了我很多成就感。

手写汇编程序的二进制码时，我也对 MIPS 指令集的指令组成和解码逻辑运用的更加娴熟。对如何调试代码也学会了更多技巧，在这次实验中，我更加频繁地使用了实验五中使用到的“寻根溯源”的方法，在调试出错的地方一步步调出相应的信号，找到需要修改代码的地方。经过这此实验，我觉得我 Verilog 的设计与使用经验也有了不小的增长。

在实验中我也曾遇到不少问题，比如关于数据的读写在时钟周期中的位置，经过我在网络上的搜索和不断地调试，最后确认在时钟稳点高电平（!clk）时读数据，在时钟下降沿（negedge clk）写数据是最佳的、不会出现数据冒险的选择。如果是在时钟上升沿读数据，可能因为流水线的数据传输也在上升沿，导致读的数据没有及时更新。这个问题困扰了我很久，但是当解决这个问题，以及想到在 STALL 机制中连续 lw 不需要停顿时，我都感到十分开心，并感到无比的成就感。

总之，这次实验让我受益匪浅，对计算机系统结构的理解也加深了不少。

8 致谢

感谢老师课堂上对于相关内容的教学。
感谢助教课堂上对于我的疑问的解答。
感谢老师与助教提供的实验手册对于实验的指导。