上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

计算机系统结构实验报告lab5

姓　　名：	张卫鸣
学　　号：	520021911141
专　　业：	计算机科学与技术

2022 年 4 月 10 日

计算机系统结构实验lab5

目录

一、实验概述

（一）实验名称

简单的类MIPS单周期处理器的实现

（二）实验目的

1. 理解简单的类MIPS单周期处理器的工作原理（即几类基本指令执行时所需的数据通路和与之对应的控制线路及其各功能部件间的互联定义、逻辑选择关系）
2. 完成简单的类MIPS单周期处理器
    1. 9条MIPS指令(lw,sw,beq,add,sub,and,or,slt,j)CPU的实现与调试
    2. 拓展至16条指令(增加addi,andi,ori,sll,srl,jal,jr)CPU的设计与实现
3. 仿真测试

二、实验描述

（一）主控制器模块Ctr

1. 主控制器模块描述

主控制器模块需要解析指令的最高六位opCode，初步判断指令类型，并且利用分支语句产生相应的控制型号。在第一步我们由opCode所区分的主要为：R型指令，I型指令的lw,sw,beq,andi,addi,ori，J型指令的j,jal。模块的信号如下表1所示：

信号	输入/输出	位数	说明
opCode	input	6	指令操作码
func	input	6	运算功能码
regDst	output	1	目标寄存器选择信号
aluSrc	output	1	ALU第二个操作数来源
memToReg	output	1	写寄存器数据来源
regWrite	output	1	写寄存器信号
memRead	output	1	读取内存信号
memWrite	output	1	写入内存信号
branch	output	1	条件跳转信号
aluOp	output	3	运算类型解析信号
jump	output	1	j指令跳转信号
extSign	output	1	符号扩展信号
jalSign	output	1	jal指令跳转信号
jrSign	output	1	jr指令跳转信号

表1

在这个模块的设计中，我们需要用分支语句来实现不同控制信号的输出，主控制器模块所产生的对应控制信号如下表所示(部分信号用首字母缩写表示)：

Opcode	指令	aluOp	RD	AS	MTR	RW	MR	MW	B	Ext	Jal	Jr	J
000000	R	101	1	0	0	1	0	0	0	0	0	0	0
000000	jr	101	1	0	0	1	0	0	0	0	0	1	0
100011	lw	000	0	1	1	1	1	0	0	1	0	0	0
101011	sw	000	0	1	0	0	0	1	0	1	0	0	0
000100	beq	001	0	0	0	0	0	0	1	1	0	0	0
000010	j	110	0	0	0	0	0	0	0	0	0	0	1
000011	jal	110	0	0	0	0	0	0	0	0	1	0	1
001000	addi	010	0	1	0	1	0	0	0	1	0	0	0
001100	andi	011	0	1	0	1	0	0	0	0	0	0	0
001101	ori	100	0	1	0	1	0	0	0	0	0	0	0

表2

从表2可以看出，opCode主要是对非R型指令进行了分流，并在输出ALUOp进一步确定下面各个指令对应什么样的运算。于是ALUOp信号将进一步作为ALUCtr信号输入来实现不同指令的运算形式。此处具体的信号对应见ALU控制器模块。

2. 主控制器模块代码实现

主控制器模块的部分代码如下：

```
always @(opCode)
   begin
       case(opCode)
           6'b000000:     //R type: add sub and or slt sll srl jr
           begin
               RegDst=1;
               ALUSrc=0;
               MemToReg=0;
               RegWrite=1;
               MemRead=0;
               MemWrite=0;
               Branch=0;
               ExtSign=0;
               JalSign=0;
               ALUOp=3'b101;
               Jump=0;
               if(func==6'b001000)begin
                   JrSign=1;
               end else begin
                   JrSign=0;
               end
           end
           6'b100011:              //lw
           begin
               RegDst=0;
               ALUSrc=1;
               MemToReg=1;
               RegWrite=1;
               MemRead=1;
               MemWrite=0;
               Branch=0;
               ExtSign=1;
               JalSign=0;
               ALUOp=3'b000;
               Jump=0;
               JrSign=0;
           end
           //其他指令类似
           //
       endcase
   end
```

此处代码是根据lab3的主控制器模块增添了一些信号修改得到，具体的实现可以参考lab3的主控制器模块。

（二）ALU控制器模块ALUCtr

1. ALU控制器模块描述

ALU控制器模块的输出是根据主控制器模块输出的aluOp和指令中的func共同决定。ALU控制器模块的输出ALUCtrOut传递给ALU运算单元进行不同的计算。ALU控制器模块的解析如表3所示。

由于对于一些R型指令我们没有对应的func指令，所以我们需要用casex进行分支。

指令	ALUOp	func	ALUCtrOut	说明
lw	000	xxxxxx	0010	加法运算
sw	000	xxxxxx	0010	加法运算
add	101	100000	0010	加法运算
addi	010	xxxxxx	0010	加法运算
beq	001	xxxxxx	0110	减法运算
sub	101	100010	0110	减法运算
andi	011	xxxxxx	0000	逻辑与运算
and	101	100100	0000	逻辑与运算
ori	100	xxxxxx	0001	逻辑或运算
or	101	100101	0001	逻辑或运算
sll	101	000000	0011	逻辑左移运算
srl	101	000010	0100	逻辑右移运算
slt	101	101010	0111	小于时置位运算
jr	101	001000	0101	无运算
jal	110	xxxxxx	0101	无运算
j	110	xxxxxx	0101	无运算

表3

在解析完指令后，我们需要将一些控制信号传输到ALU运算单元。模仿我们在lab3中的设计，如表4构建ALUCtr的输入输出信号端口。在这里多了一个信号输出shamtSign表示判断是否需要移位，在指令为sll和srl时置为高电平。

信号	输入/输出	长度	说明
aluOp	input	3	运算类型解析信号
funct	input	6	指令功能码
aluCtrOut	output	4	ALU运算信号
shamtSign	output	1	移位信号

表4

2. ALU控制器模块代码实现

与lab3类似，我们用casex语句进行分支，代码如下：

```
module ALUCtr(
    input [2:0] aluOp,
    input [5:0] funct,
    output reg [3:0] aluCtrOut,
    output reg shamtSign
    );
    always @ (aluOp or funct)
    begin
        casex({aluOp,funct})
            9'b000xxxxxx://lw sw
                aluCtrOut = 4'b0010;
            9'b001xxxxxx://beq
                aluCtrOut = 4'b0110;
            9'b010xxxxxx://addi
                aluCtrOut = 4'b0010;
            9'b011xxxxxx://andi
                aluCtrOut = 4'b0000;
            9'b100xxxxxx://ori
                aluCtrOut = 4'b0001;
            9'b101000000://sll
                aluCtrOut = 4'b0011;
            9'b101000010://srl
                aluCtrOut = 4'b0100;
            9'b101001000://jr
                aluCtrOut = 4'b0101;
            9'b101100000://add
                aluCtrOut = 4'b0010;
            9'b101100010://sub
                aluCtrOut = 4'b0110;
            9'b101100100://and
                aluCtrOut = 4'b0000;
            9'b101100101://or
                aluCtrOut = 4'b0001;
            9'b101101010://slt
                aluCtrOut = 4'b0111;
```

```
            9'b110xxxxxx://j jal
                aluCtrOut = 4'b0101;
        endcase
        if({aluOp, funct}==9'b101000000||
{aluOp,funct}==9'b101000010)
            shamtSign=1;
        else
            shamtSign=0;
    end

endmodule
```

（三）算术逻辑运算单元ALU

1. 算术逻辑运算单元描述

算术逻辑运算单元ALU根据ALU控制单元输出的ALUCtrOut进行分支，从而让ALU进行对应的算数逻辑运算。具体运算的类型参照表3。

同lab3里的ALU运算单元类似，ALU还有两个32位的操作数输入和一个32位的结果输出以及0判断输出信号。输入输出信号如表5所示：

信号	输入/输出	长度	说明
input1	input	32	操作数1
input2	input	32	操作数2
aluCtr	input	4	ALU运算信号
zero	output	1	0标志信号
aluRes	output	32	ALU输出结果

表5

2. 算术逻辑运算单元代码实现

在本次实验中我们只是对lab3中的ALU模块进行了一些扩充，分支部分代码如下：

```
always @ (input1 or input2 or aluCtr)
    begin
        case(aluCtr)
            4'b0000:    //and
                ALURes=input1&input2;
            4'b0001:    //or
                ALURes=input1|input2;
            4'b0010:    //add
                ALURes=input1+input2;
            4'b0011:    //left-shift
```

```
                ALURes=input2<<input1;
            4'b0100:    //right-shift
                ALURes=input2>>input1;
            4'b0101:    //j
                ALURes=input1;
            4'b0110:    //sub
                ALURes=input1-input2;
            4'b0111:    //set less than
                ALURes=($signed(input1)<$signed(input2));
            default:
                ALURes=0;
        endcase
```

（四）寄存器模块Registers

本模块基本和lab4中的实现类似，但由于在本次实验我们要求实现完整的MIPS单周期处理器，我们还要加入对reset信号的处理。也就是当reset为高电平时，寄存器将会清零。

在这里我们对lab4也做了一些改进，因为在此实验中我们要实现jal指令，而jal指令将会写进寄存器。所以我们在寄存器模块加入信号接口，并修改模块代码，来实现jal指令写进寄存器的选择。

寄存器模块的输入输出信号如表6所示。

信号	输入/输出	长度	说明
readReg1	input	5	读取寄存器1的编号
readReg2	input	5	读取寄存器2的编号
writeReg	input	5	写入寄存器编号
writeData	input	32	写入数据
regWrite	input	1	写信号
jalSign	input	1	jal跳转信号
jalData	input	32	jal写入数据
clk	input	1	时钟信号
reset	input	1	reset信号
readData1	output	32	读取寄存器1的结果
readData2	output	32	读取寄存器2的结果

表6

```
module Registers(
    input [4:0] readReg1,
    input [4:0] readReg2,
```

```
    input [4:0] writeReg,
    input [31:0] writeData,
    input regWrite,
    input jalSign,
    input[31:0] jalData,
    input clk,
    input reset,
    output [31:0] readData1,
    output [31:0] readData2
    );
    reg [31:0] regFile [31:0];
    integer i;


    assign readData1=regFile[readReg1];
    assign readData2=regFile[readReg2];


    always @ (negedge clk)
    begin
        if(regWrite)begin
            regFile[writeReg]=writeData;
        end
        if(jalSign)begin
            regFile[31]=jalData;
        end
    end
    always@(reset)
    begin
        for(i=0;i<=31;i=i+1)
            regFile[i]=0;
    end
endmodule
```

（五）数据内存单元dataMemory

数据内存单元和我们在lab4中的实现完全一致，在此处不再展示具体的信号和代码。

（六）带符号扩展单元SignExt

带符号扩展单元亦和我们在lab4中的实现完全一致，此处不再展示具体的信号和代码。

（七）指令存储单元InstMemory

1. 指令存储单元描述

在本实验所实现的MIPS单周期处理器中，我们将数据内存和指令内存分离处理。我们的指令内存模块以一个32位地址(来源于PC)作为输入，以一个32位的指令作为输出，如表7所示。

信号	输入/输出	长度	说明
address	input	32	指令所在地址
inst	output	32	取出的指令

表7

2. 指令存储单元代码实现

```
module InstMemory(
    input [31:0] address,
    output [31:0] inst
    );
    reg [31:0] instFile [0:63];
    assign inst = (instFile[address>>2]);
endmodule
```

（八）程序计数器模块PC

1. 程序计数器描述

程序计数器的功能是在时钟的上升沿将PC修改后输出，所以有一个输入PC和一个输出PC。在这里我们作为一个简单的模块实现，并考虑reset信号清零的效果，设计如表8。

信号	输入/输出	长度	说明
clk	input	1	时钟信号
reset	input	1	重置信号
pcIn	input	32	PC输入
pcOut	output	32	PC输出

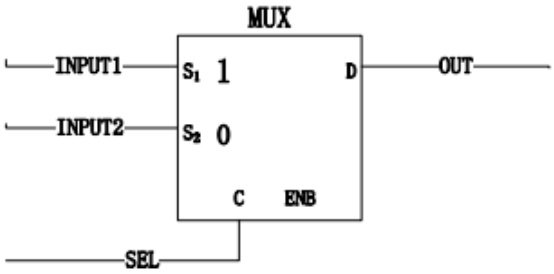
表8

2. 程序计数器代码实现

```
module PC(
    input [31:0] pcIn,
    input clk,
    input reset,
    output reg[31:0] pcOut
    );
    initial pcOut=0;
    always @ (posedge clk or reset)
    begin
        if(reset)
            pcOut=0;
        else
            pcOut=pcIn;
        $display("PC:%d\n",pcOut);
    end
endmodule
```

（九）多路选择器MUX实现

多路选择器模块接受两个输入信号和一个选择信号，而产生一个输出信号，实现“二选一”的效果。多路选择器的模型如图1所示。



在本次实验中，我们不仅需要对32位的数据进行选择，有时也需要对5位的寄存器选择信号进行选择，所以在本实验中我们设计了两种选择器模块，分别是Mux和Mux_s。

以Mux(32位)为示例，代码如下：

```
module Mux(
    input selectSignal,
    input [31:0] input1,
    input [31:0] input2,
    output [31:0] out
    );
    assign out=selectSignal ?input1:input2;
endmodule
```

在本实验中，我们有多处需要用到多路选择器，选择器的具体分支效果如下表8所示。(其中的选择信号在Top顶层模块文件中以输出信号实现)

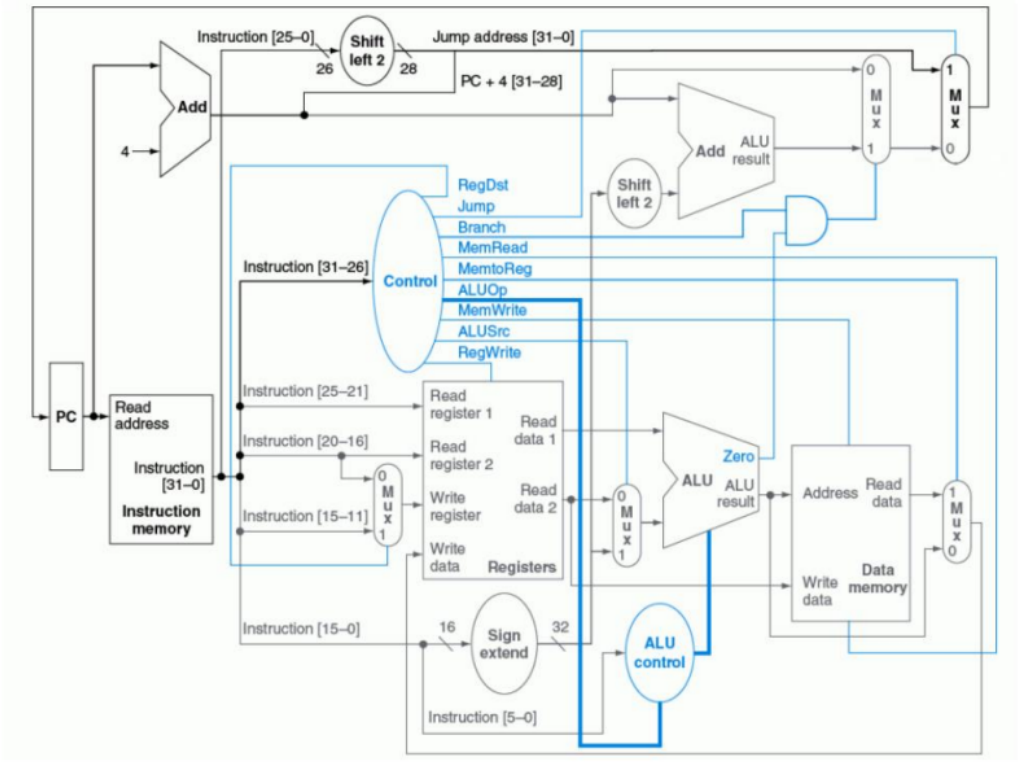
多路选择器	选择信号	说明
shamt_rs_mux	SHAMT_SIGN	选择是否移位操作
inst_rt_mux	ALU_SRC	选择ALU输出结果
rt_rd_mux	REG_DST	选择rt/rd作为寄存器
mem_alu_mux	MEM_TO_REG	选择返回寄存器的数据
branch_mux	BRANCH&ALU_ZERO	选择是否跳转
jump_mux	JUMP	选择是否jump跳转
jr_mux	JR_SIGN	选择是否jr指令跳转

表9

（十）顶层模块Top

1. 顶层模块描述

顶层模块是用来将我们之前所设计实现的小模块实例化并且组装成一个完整的MIPS单周期处理器。参考实验指导书，一个简单的单周期处理器设计图如图2所示。



但在扩展成16指令时，我们添加了几个Mux多路选择器来处理增加的指令，具体见（九）。剩下的我们按照改进过的设计图增加wire进行连线得到一个初步的单周期处理器。

2. 顶层模块代码实现

```
module Top(
    input clk,
    input reset
    );
    wire REG_DST;               //reg dst, choose rt or rd
    wire JUMP;                  //jump
    wire BRANCH;                //branch
    wire MEM_READ;              //mem read
    wire MEM_TO_REG;            //mem to reg
    wire MEM_WRITE;             //mem right
    wire [2:0] ALU_OP;          //alu op
    wire ALU_SRC;               //alu src
    wire REG_WRITE;             //reg write
    wire [31:0] INST_ADDR;      //instruction address
    wire [31:0] INST;           //instruction
    wire SHAMT_SIGN;            //shamt sign
    wire JR_SIGN;               //jr sign
    wire EXT_SIGN;              //ext sign
    wire JAL_SIGN;              //jal sign
    wire ALU_ZERO;              //alu zero
```

```
21      wire [3:0] ALU_CTR;         //alu ctr
22
23      wire [31:0] ALU_INPUT1;     //alu input1
24      wire [31:0] ALU_INPUT2;     //alu input2
25      wire [31:0] ALU_RES;        //alu result
26      wire [31:0] PC_IN;          //pc input
27      wire [31:0] PC_OUT;         //pc output
28      wire [4:0] REG_READ_SELECT1;       //read reg1
29      wire [4:0] REG_READ_SELECT2;       //read reg2
30      wire [4:0] REG_WRITE_SELECT;       //write reg2
31      wire [31:0] REG_WRITE_DATA; //reg write data
32      wire [31:0] MEM_READ_DATA;  //mem read data
33      wire [31:0] REG_READ_DATA1; //read reg1
34      wire [31:0] REG_READ_DATA2; //read reg2
35      wire [31:0] EXT;            //extend result
36
37      wire [31:0] PC_BRANCH;
38      wire [31:0] PC_JUMP;
39      wire [4:0] REG_WRITE_JAL;
40      wire [31:0] REG_WRITE_FROM_MEM;
41
42      Ctr main_ctr(
43          .opCode(INST[31:26]),
44          .func(INST[5:0]),
45          .regDst(REG_DST),
46          .aluSrc(ALU_SRC),
47          .memToReg(MEM_TO_REG),
48          .regWrite(REG_WRITE),
49          .memRead(MEM_READ),
50          .memWrite(MEM_WRITE),
51          .branch(BRANCH),
52          .extSign(EXT_SIGN),
53          .jalSign(JAL_SIGN),
54          .aluOp(ALU_OP),
55          .jump(JUMP),
56          .jrSign(JR_SIGN)
57      );
58
59      ALUCtr alu_ctr(
60          .aluOp(ALU_OP),
61          .funct(INST[5:0]),
62          .aluCtrOut(ALU_CTR),
63          .shamtSign(SHAMT_SIGN)
64      );
65
66      ALU alu(
67          .input1(ALU_INPUT1),
68          .input2(ALU_INPUT2),
```

```
        .aluCtr(ALU_CTR),
        .zero(ALU_ZERO),
        .aluRes(ALU_RES)
    );


    Registers register_file(
        .readReg1(INST[25:21]),
        .readReg2(INST[20:16]),
        .writeReg(REG_WRITE_SELECT),
        .writeData(REG_WRITE_FROM_MEM),
        .regWrite(REG_WRITE&(~JR_SIGN)),
        .jalSign(JAL_SIGN),
        .jalData(PC_OUT+4),
        .clk(clk),
        .reset(reset),
        .readData1(REG_READ_DATA1),
        .readData2(REG_READ_DATA2)
    );
    PC pc(
        .pcIn(PC_IN),
        .clk(clk),
        .reset(reset),
        .pcOut(PC_OUT)
    );

    InstMemory inst_mem(
        .address(PC_OUT),
        .inst(INST)
    );

    dataMemory data_mem(
        .clk(clk),
        .address(ALU_RES),
        .writeData(REG_READ_DATA2),
        .memWrite(MEM_WRITE),
        .memRead(MEM_READ),
        .readData(MEM_READ_DATA)
    );
    signext sign_ext(
        .extSign(EXT_SIGN),
        .inst(INST[15:0]),
        .data(EXT)
    );


    Mux shamt_rs_mux(
        .selectSignal(SHAMT_SIGN),
```

```
        .input1({27'b0,INST[10:6]}),
        .input2(REG_READ_DATA1),
        .out(ALU_INPUT1)
    );
    Mux inst_rt_mux(
        .selectSignal(ALU_SRC),
        .input1(EXT),
        .input2(REG_READ_DATA2),
        .out(ALU_INPUT2)
    );
    Mux_s rt_rd_mux(
        .selectSignal(REG_DST),
        .input1(INST[15:11]),
        .input2(INST[20:16]),
        .out(REG_WRITE_SELECT)
    );
//    Mux_s rt_rd_next_mux(
//        .selectSignal(JAL_SIGN),
//        .input1(5'b11111),
//        .input2(REG_WRITE_SELECT),
//        .out(REG_WRITE_JAL)
//    );
    Mux mem_alu_mux(
        .selectSignal(MEM_TO_REG),
        .input1(MEM_READ_DATA),
        .input2(ALU_RES),
        .out(REG_WRITE_FROM_MEM)
    );
//    Mux jal_mux(
//        .selectSignal(JAL_SIGN),
//        .input1(PC_OUT+4),
//        .input2(REG_WRITE_FROM_MEM),
//        .out(REG_WRITE_DATA)
//    );
    Mux branch_mux(
        .selectSignal(BRANCH & ALU_ZERO),
        .input1(PC_OUT+4+(EXT<<2)),
        .input2(PC_OUT+4),
        .out(PC_BRANCH)
    );
    Mux jump_mux(
        .selectSignal(JUMP),
        .input1(((PC_OUT+4)&32'hf0000000)+(INST[25:0]<<2)),
        .input2(PC_BRANCH),
        .out(PC_JUMP)
    );
    Mux jr_mux(
        .selectSignal(JR_SIGN),
```

```
        .input1(REG_READ_DATA1),
        .input2(PC_JUMP),
        .out(PC_IN)
    );
endmodule
```

三、实验结果验证

（一）汇编代码设计与编写

为了测试单周期处理器是否能够正确执行指令，我设计了一串MIPS指令并用代码进行输入验证。

测试用的代码如下：

```
00001000000000000000000000000100    //j
00000000000000000000000000000000    //nop
00000000000000000000000000000000    //nop
00000000000000000000000000000000    //nop
00001100000000000000000000000110    //jal
00000000000000000000000000000000    //nop
10001100000000010000000000000010    //lw 1,2(0)     $1:2
10001100000000010000000000000110    //lw 2,6(0)     $2: 6
00000000001000100001100000100000    //add 3,1,2     $3:8
00000000010000010010000000100010    //sub 4,2,1     $4:4
00000000001001000001000000100010    //sub 2,1,4     $2:-2
00000000001001000001100000100100    //and 3,1,4     $3:0
00000000000000010000100001000000    //sll 1,1,1     $1:4
00100000000000010000000000000010    //addi 1,0,2    $1:2
00110100100000100000000000000011    //ori 2,4,3     $2:7
00000000001000100001100000101010    //slt 3,1,2     $3:1
10101100000000100000000000000001    //sw 2,1(0)
00010000101000000000000000000001    //beq 5,0,1
00000000010000000000000000001000    //jr 2
00000011111100000000000000001000    //jr 31
```

内存的初始值如下：

```
00000000
00000001
00000002
00000003
00000004
00000005
00000006
00000007
00000008
```

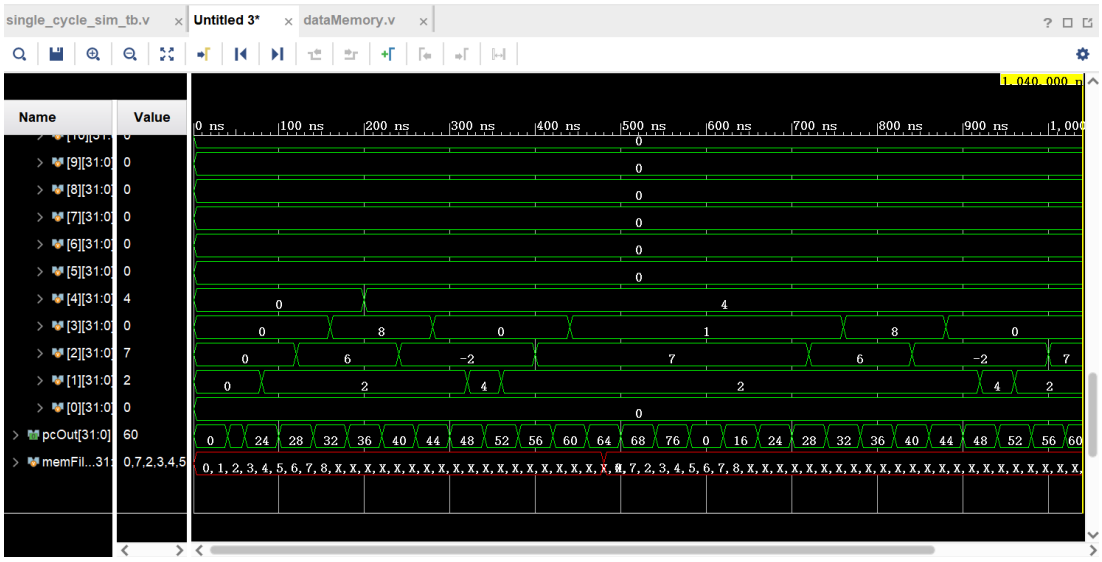
将内存的初始值写在data.dat中，将数据写在inst_data.dat中。

（二）仿真测试代码

激励文件代码如下：

```
module single_cycle_sim_tb(

    );
    reg clk;
    reg reset;
    Top processor(.clk(clk),.reset(reset));

    initial begin

 $readmemb("D:/C/Arch/lab5/inst_data.dat",processor.inst_mem.instFile);

 $readmemh("D:/C/Arch/lab5/data.dat",processor.data_mem.memFile);
        reset=1;
        clk=0;
    end

    always #20 clk=~clk;
    initial begin
        #40 reset=0;
        #1000;
        $finish;
    end
endmodule
```

（三）仿真波形



如图3所示，我们设计的MIPS单周期处理器正确实现了我设计的测试代码。

四、实验心得

本次实验的难度比前几个实验有了陡然的提升。刚开始做实验时有点一头雾水不知道该写什么。后来在询问同学的过程中我明白是将lab3和lab4中实现的模块进行连线从而构成一个CPU整体。

我本人在做实验时是跳过了先前实验为基础的9条指令而直接实行16条指令的。我先上网搜索了16条指令的相关编码，然后自己重新从头开始进行ALUOp和ALUCtrOut的编码。之后便先忙着去组装其他模块了。在最终调试的过程中我发现PC的变化是正确的，但是寄存器的值却始终不正确。在咨询学长和同学之后我意识到自己的编码出了问题，没有考虑到比如lw也是利用的add功能，从而导致一些运算的不正确。于是我参考了学长的编码方式再结合自己的编码重新进行了一次，结果发现还是不对！又仔细看了好久终于找到问题，是sll指令的操作数写反了。这个调试花了我很长的时间，在此认识到打牢基础，细心编码的重要性。这个过程我深深地重新复习了MIPS的指令设计。

以及本次设计比较难的一点在于对于jal,beq,j,jr指令的多路选择器的安置。这边也是我上网搜寻相关资料和设计所得，就是多个Mux叠加来得到最终返回pc的值，而使得各个指令都可以得以实现，感觉本质上有点像串联的意思，实现连续传输，相信这个思想也会在更为复杂的lab6里继续得到应用。

以及在这个实验中我对于在vivado中连线的技能得到了很大的磨练，也解开了之前的一些困惑比如wire和register的区别（前者只是用于连接，后者能够暂存数据）。

本次相较于前几次的实验我对设计也有了自己的一些思考，在一开始考虑jal指令的时候我是用多路选择器来实现的，后来在完成后续的实验的时候我突然想到了可以简化电路的方法：将jal的选择直接转移到Registers模块中，增加信号数据接口，这样便一定程度简化了Top顶层模块的连线。

还有一个比较有趣的点在于自己编写MIPS指令来验证自己的单周期处理器，在此过程中对MIPS指令编码又得到了巩固。

总之本次实验我受益匪浅，对于单周期处理器的各方面知识有了较为全面的了解。

五、参考资料

2022计算机系统结构实验指导书lab5