计算机系统结构实验报告 实验五

惠宇龙 518030910059

1 实验概述

1.1 实验描述

简单的类 MIPS 单周期处理器的设计与实现。

1.2 实验目的

- 完成单周期的类 MIPS 处理器整体设计
- 使单周期CPU支持16条MIPS指令

2 单周期处理器整体设计原理

导入Lab3和Lab4的代码，并加以改动。这些部件就是单周期CPU的每一个重要组成部分，除此之外还要额外设计存放指令的存储器、选择器、加法器。最终，完成每一个模块之后，按照逻辑将其端口相连接，完成单周期处理器的实现。各个模块之间的关系，如下图所示：

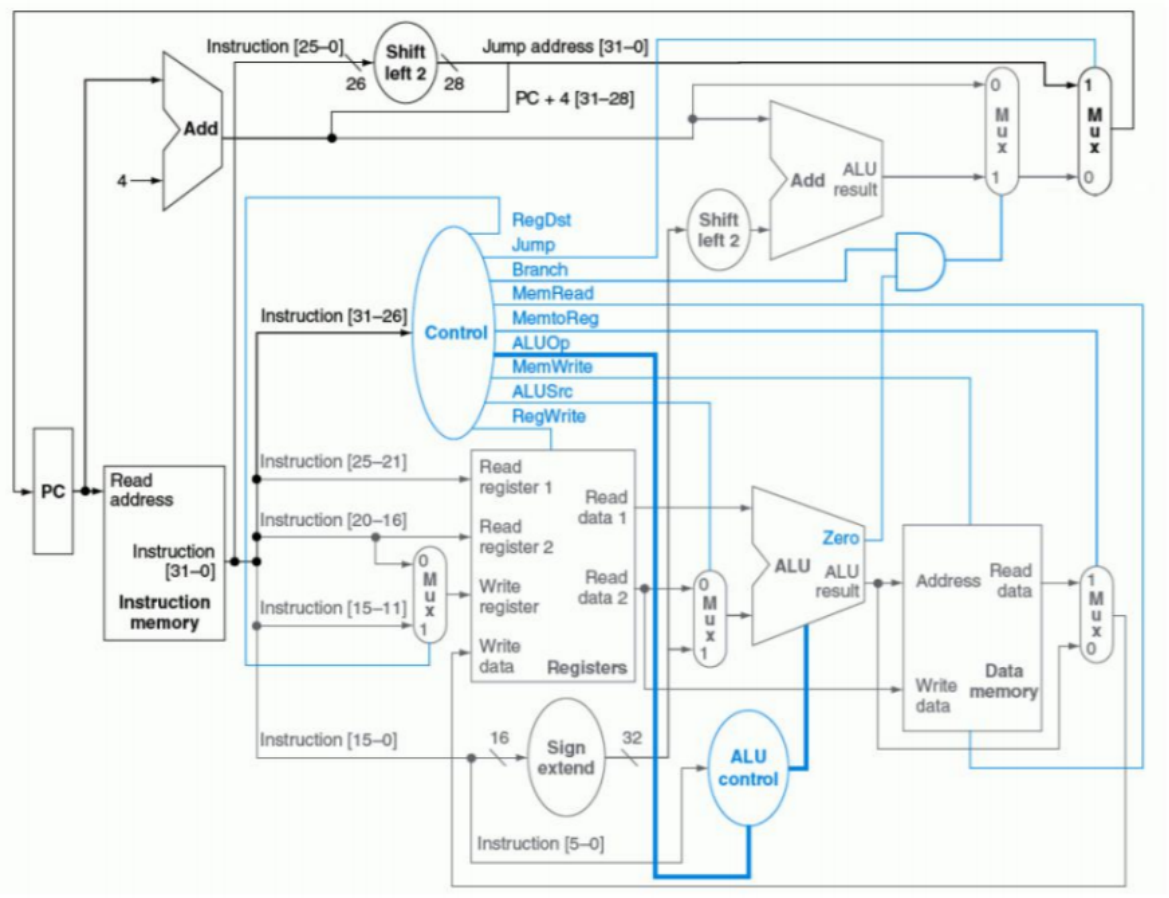


图1.类MIPS单周期处理器原理图

3 单周期处理器分模块描述

3.1 新增模块

3.1.1 指令存储器

指令存储器与之前实现的存储器类似，但是不需要写操作，因此端口更简单些。

需要注意的是，这里的指令存储器遵循字节寻址（虽然每次地址都是4的整数倍）。因此，每一个reg单元对应8个bit而不是32个。这样设置，和PC+4 （每次更新指令需要跳转四个单元）相吻合。最终输出的指令，则需要由四个reg单元拼接组成。

```
module InstMemory(
    input [31:0] readaddress,
    output reg [31:0] readinst
    );

    reg [7:0] instfile[0:255];

    always @ (readaddress)
    begin
    readinst[7:0] = instfile[readaddress+3];
    readinst[15:8]= instfile[readaddress+2];
    readinst[23:16]= instfile[readaddress+1];
    readinst[31:24]= instfile[readaddress];
    end

endmodule
```

3.1.2 数据选择器

不需要额外编写模块，可以直接使用三目运算符来实现：

```
assign A= mux? B : C ;
```

3.1.3 加法器

同样不需要额外编写模块，可以直接用加法运算解决。

3.2 修改模块

3.2.1 寄存器

由于此时寄存器需要初始化。为了避免特殊的时序性错误，我们在寄存器写数据的代码块里添加了reset的部分，如下所示(其余部分在此略去）：

```
always @ (reset or negedge Clk)
 begin
  if(reset)
    for(i=0;i<=31;i=i+1)
        regFile[i]<=0;

 else if(regwrite)
     regFile[writeReg]<= writeData;
 end
```

3.2.2 存储器

与指令存储器相似，在MIPS中，数据存储器也是按照字节寻址。但是在本实验的指令中，均是按照word来访问或者写入数据的。因此我们依然选择大小为8-bit的reg单元，在写入和读取数据时进行拆分和拼接操作：

```
module DM(
    input Clk,
    input [31:0] address,
    input [31:0] writeData,
    input memwrite,
    input memread,
    output reg [31:0] readData
    );
reg [7:0] memFile[0:255];

    always @ (address or memread or memwrite)
    if(memread)
    begin
     readData[7:0] = memFile[address+3];
     readData[15:8]= memFile[address+2];
     readData[23:16]= memFile[address+1];
     readData[31:24]= memFile[address];
    end


always @ (negedge Clk)
if (memwrite)
begin
    memFile[address+3]=writeData[7:0];
    memFile[address+2]=writeData[15:8];
```

```
    memFile[address+1]=writeData[23:16];
    memFile[address]= writeData[31:24];
end

endmodule
```

3.2.3 ALU-ctr 和 主控制器

由于要实现类MIPS 处理器，根据老师在理论课所讲内容，ALUop的代码应当为3-bit，而非Lab3中实现的简化版的2-bit。因此，我将相关的输出、输入信号都改成了3-bit。（此处略去冗杂的代码），修改后，3-bit的ALUop和不同类型指令的关系为：

Encoding ALUop as follows

	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtr	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	001	xxx

图2.类MIPS处理器ALUop与不同指令的对照表

除此之外，由于需要实现16个指令，因此主控制器等的译码过程也需要进行一定的扩展。考虑到该部分的代码变动并无太多难点但是冗杂繁复，因此在报告中略去，详细请参考项目文件。

4 顶层控制及具体实现

新建 Top.v 文件，并置于顶层。用以实现对各个模块的顶层连接和控制。下面展示了部分核心代码，详细细节请参考项目文件。

4.1 读取指令

用pc存储当前指令的地址，用inst组线接收返回的指令代码

```
wire [31:0] inst;
reg pc;

InstMemory im0(
    .readaddress(pc),
    .readinst(inst)
    );
```

4.2 通过主控制器实现整体控制

定义多种信号线，使其分别对应各种信号。这些信号线连通CU和各个部件，将CU产生的控制信息传递给各个部件。

```
    wire RegDst;
    wire Jump;
    wire Branch;
    wire MemRead;
    wire MemtoReg;
    wire [2:0]ALUOp;
    wire MemWrite;
    wire ALUsrc;
    wire RegWrite;
```

```
ctr ctr0(
     .opcode(inst[31:26]), // inst的[31:26]对应opcode部分
    .aluop(ALUOp),
    .branch(Branch),
    .jump(Jump),
    .alusrc(ALUsrc),
    .memwrite(MemWrite),
    .regwrite(RegWrite),
    .memtoreg(MemtoReg),
    .memread(MemRead),
    .regdst(RegDst)
    );
```

4.3 数据读取或写入寄存器

下列代码中，WriteReg 表示写入数据的目标寄存器位置，WriteDataR 表示需要写入寄存器的数据。二者都需要在根据控制信号选择正确的数据。data1和data2组线，用于接收寄存器中读出的数据，并将其传递给其他部件。

```
assign WriteReg = Jal? 31: RegDst? inst[15:11] : inst[20:16];
assign WriteDataR= Jal? pctmp :MemtoReg ? memdata : ALUres;
wire[31:0] data1,data2;

Reg reg0(
    .Clk(Clk),
    .readReg1(inst [25:21]),
    .readReg2(inst [20:16]),
    .writeReg(WriteReg),
    .writeData(WriteDataR),
    .regwrite(RegWrite),
    .readData1(data1),
    .readData2(data2),
    .reset(reset)
    );
```

4.4 ALU操作

根据ALUctr的信号和其他控制信号，完成ALU的操作。特别地，ALU的两个输入数据也需要由选择器进行选择。（其中shiftmux 是我新引入的选择器，用来确定ALU是否要进行移位操作，这将决定操作数据的来源）

```
    assign input2= ALUsrc ? immedata :data2;
    assign input1= shiftmux? inst[10:6] : data1;
    alu alu0(
    .input1(input1),
    .input2(input2),
    .aluCtr(ALUctr),
    .zero(zero),
    .aluRes(ALUres)
    );

    aluctr aluctr0(
    .ALUOp(ALUOp),
    .Funct(inst[5:0]),
    .Oper(ALUctr),
    .shiftmux(shiftmux),
```

```
    .Jal(Jal),
    .Jr(Jr)
    );
```

4.5 跳转指令

下一个 PC 的值，可能来自 PC + 4、`beq` 指令指定的条件跳转地址或 `j` 指令指定的无条件跳转地址。条件跳转地址为指令低 16 位的立即数符号扩展后加上 PC + 4 的值，无条件跳转地址为指令低 26 位的立即数左移两位再拼接上 PC + 4 的高 4 位的值。先根据 Branch 和 Zero 的与运算结果决定是来自 PC + 4 还是条件跳转地址，再根据 Jump 决定是来自前面计算的地址还是无条件跳转地址，将此结果作为下一个 PC 的值。特别地，当出现`jr`命令时，应当优先考虑是否选择对应寄存器内的数据作为pc的值。最终， 在时钟上升沿，将 PC 更新为下一个值，如果reset信号为高电平，则将 PC 置为零。

```
    assign branchjudge= Branch && zero;
    assign pctmp=pc+4;
    assign branchtmp = (immedata<<2)+pctmp;
    assign jumpshift[27:2]= inst[25:0];
    assign jumpshift[1:0]= 2'b00;
    assign branchmux= branchjudge ? branchtmp: pctmp;
    assign jumptmp[31:28] = pctmp[31:28];
    assign jumptmp[27:0]=jumpshift;

    assign jumpmux= Jr? data1: Jump? jumptmp: branchmux;


     always @ (reset or  posedge Clk )
     begin
     if(reset)
     pc=0;

     else if (Clk)pc = jumpmux;
     end
```

4.6 读写数据存储器的操作

将该模块实例化即可，特别地，写入存储器的数据就是寄存器读取的data2数据。

```
DM dm0(
    .Clk(Clk),
    .address(ALUres),
    .writeData(data2),
    .memwrite(MemWrite),
    .memread(MemRead),
    .readData(memdata)
    );
```

综上，我们已经介绍了各个模块的设计和实现，已经顶层控制的实现。更多的细节代码还请参考项目文件。

5 单周期处理器的检验

由于参考书所给的样例测试程序较为简单，我选择自己编写测试程序，激励文件如下：

```
module cpu_tb(
```

```
    );
    reg Clk;
    reg reset;

    Top cpu0(
    .Clk(Clk),
    .reset(reset));

    always #40 Clk=!Clk;

    initial begin
    $readmemb("C:/Archlabs/inst.mem",cpu0.im0.instfile);
    $readmemh("C:/Archlabs/mem.mem",cpu0.dm0.memFile);
    Clk=1;
    reset=1;
    #80 reset=0;
    end
endmodule
```

指令如下：

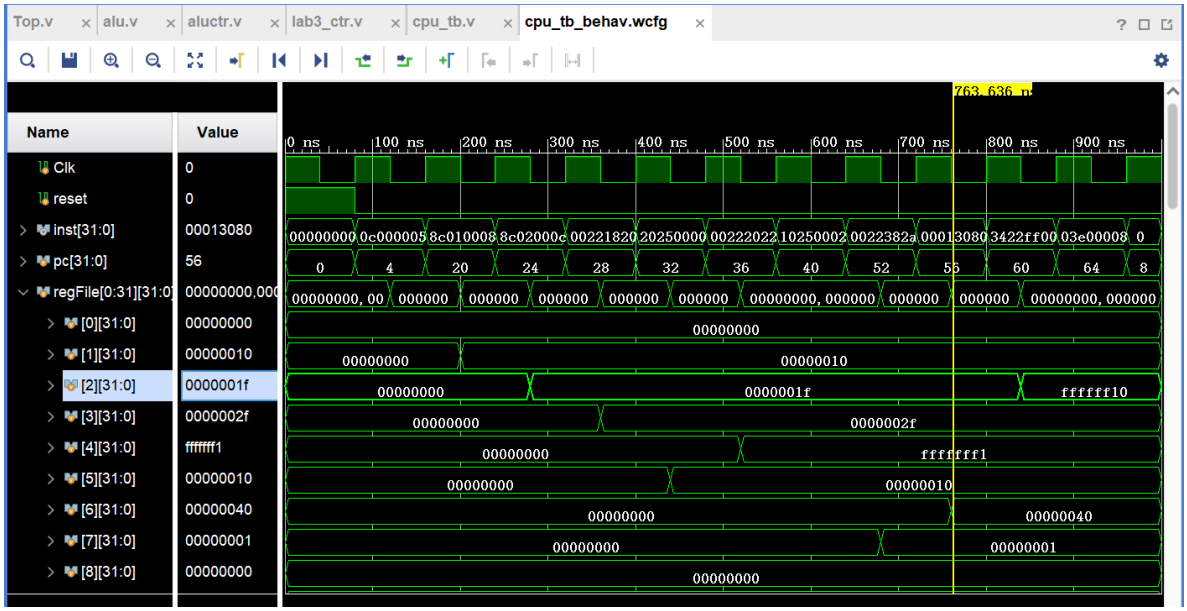
```
00000000000000000000000000000000 //nop
00001100000000000000000000000101 //jal 第5个指令
00000000000000000000000000000000 //nop
00000000000000000000000000000000 //nop
00000000000000000000000000000000 //nop
10001100000000010000000000001000  // lw $1 , 8($0) 16
10001100000000100000000000001100  //lw $2, 12($0) 20
00000000010001000011000000100000  //add $3,$1,$2
00100000010010100000000000000000 //addi $5,$1,0
00000000010001000100000000100010  //sub $4,$1,$2
00010000010010100000000000000010 //beq $1,$5,2   跳2个指令
00000000000000000000000000000000 //nop
00000000000000000000000000000000 //nop
00000000010001000111000000101010 //slt $7,$1,$2
00000000000000010011000010000000 //sll $6, $1,2
00110100001000101111111100000000 //ori $2, $1, 1111111100000000
00000011111000000000000000001000 //jr $31
```

内存的数据如下：

```
0x00
0x00
0x00
0x01  //1
0x00
0x00
0x00
0x05 //5
0x00
0x00
0x00
0x10 //16
0x00
0x00
0x00
```

```
0x1f //31
```

运行上述测试，我们可以得到如下波形图：



可以看到，pc信号在0到16，以及36到48的地方出现了跳变。这分别对应代码中的`jal`和`beq`跳转指令。在pc信号达到64之后跳转回8，这是因为`jr`指令读取了$31中的数据并写入pc。

从内存中导入数据0x10,0x1f，随后我们可以看到将其求和得到0x2f，相减得到-15(ffffffff1) 。$5=$1+0 , 故$5得到0x10。之后让$1的数据`0x10`左移两位（`sll`指令)，并载入$6,所以$6变为`0x40`。$7显示的是slt $1,$2 的结果，因为$1的数据更小，所以$7的数据为1。最后，执行`ori`，$2的结果变为-240(ffffff10)。

综上所述，波形呈现出了符合逻辑的结果，实验成功完成。

6 总结反思

本实验与此前的实验相比，难度有所增加。主要难点体现在： 整个处理器系统相对庞大，需要处理的部件很多，并且连线相对复杂。这就要求我们能够理清个部件的逻辑关系，有条理的完成实现。在此期间，CPU的原理图（图1），帮助我们建立了整体和局部的概念，发挥了重要的作用。

关于`jal`和`jr`命令的实现，是理论课堂上没有讲解过的。我按照自己的设计，在aluCtr的控制部分新设立了两个端口，分别传递这两个命令的信号，以此正确完成了二者的实现。

在理论课上我们学习了类MIPS单周期处理器的相关原理，本次实验进一步加深了我对它的理解。另一方面，单周期处理器的设计相对基础、简单，这也为Lab6 流水线的设计和实现做好了铺垫