计算机系统结构实验报告 Lab05

简单的类 MIPS 单周期处理器的实现-整体调试

徐阳　521021910363

2023 年 5 月 3 日

目录

1 简介

在本实验中，以实验 3、4 为基础，我理解并实现了简单的类 MIPS 单周期处理器。该处理器支持 16 条 MIPS 指令 (lw, sw, beq, add, sub, and, or, slt, j, addi, andi, ori, sll, srl, jal, jr)。最后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. 理解简单的类 MIPS 单周期处理器的工作原理理（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，从 9 条指令拓展为 16 条指令
3. 掌握激励文件的编写，使用行为仿真验证程序正确性

3 实验原理与功能实现

3.1 主控制器单元 Ctr

3.1.1 原理分析

主控制器单元 Ctr 的输入为指令的高六位 OpCode，输出与实验三中的 Ctr 模块相近。主控制器单元 Ctr 根据指令高六位的 OpCode 初步判断指令是 R 型，I 型还是 J 型指令，并产生相应的处理器各部分控制信号。

除了原有的目标寄存器的选择信号 regDst、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、条件跳转信号 branch、无条件跳转信号 Jump 之外，为了实现 16 条 MIPS 指令，我们还要新加入符号拓展信号 extSign，jal 指令信号 JalSign，

同时因为指令的增多，ALUOp 需要由两位拓展为三位才能满足中间的解码需求。

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。

3.1.2 代码实现

用 case 语句判断指令类型，在这里给出了 R 类型指令的解码代码。

同样地，我们要在 case 语句最后加上 default 指令，在列出的情况外的指令将被视为 nop 指令（空指令），我们将 ALUOp 设置为 111（未定义的运算操作），这样 ALU 不会执行任何操作。

——ALU.v——

表 1: ALUop 与指令的对应及作用

ALUOp	指令	ALU 运算类型
000/010	lw, sw, addi	add
001	beq	sub
011	andi	and
100	ori	or
101	R-Type	在 ALUCtr 中具体分析
110	j,jal	ALU 不进行运算

```
always @ (opCode)
begin
    case(opCode)
        6'b000000:      // R Type
        begin
            regDst = 1;
            aluSrc = 0;
            memToReg = 0;
            regWrite = 1;
            memRead = 0;
            memWrite = 0;
            branch = 0;
            aluOp = 3'b101;
            jump = 0;
            extSign = 0;
            jalSign = 0;
        end
        ...
        default:        // default
        begin
            regDst = 0;
            aluSrc = 0;
            memToReg = 0;
            regWrite = 0;
            memRead = 0;
            memWrite = 0;
            branch = 0;
            aluOp = 3'b111;
```

```
29                  jump = 0;
30                  extSign = 0;
31                  jalSign = 0;
32              end
33          endcase
34      end
```

3.2 运算单元控制器单元 ALUCtr

3.2.1 原理分析

运算单元控制器单元 ALUCtr 输入为三位 ALUOp，指令低六位的 funct，输出为给 ALU 的运算控制信号 ALUCtrOut，该控制信号作用于 ALU，指定其运算类型，因为指令类型的增加，我们需要加入 shamtSign 和 jrSign 这两个输出的控制信号。

具体的解析及型号对应如表 2所示，除去表中所示的 ALUCtrOut，如果指令为 sll 或 srl 则 shamtSign 置一，如果指令为 jr 则 jrSign 置零，其余情况两者皆为 0。

表 2: ALUCtr 中的信号解析

指令	ALUOp	funct	ALUCtrOut	运算类型
lw	000	xxxxxx	0010	add
sw	000	xxxxxx	0010	add
beq	001	xxxxxx	0110	sub
addi	010	xxxxxx	0010	add
andi	011	xxxxxx	0000	and
ori	100	xxxxxx	0001	or
sll	101	000000	0011	sll
srl	101	000010	0100	srl
jr	101	001000	0101	nop
add	101	100000	0010	add
sub	101	100010	0110	sub
and	101	100100	0000	and
or	101	100101	0001	or
slt	101	101010	0111	slt
j	110	xxxxxx	0101	nop
jal	110	xxxxxx	0101	nop

3.2.2 代码实现

和实验三中一样采用支持通配符 x 的 casex 语句实现 ALUCtr，以下是代码的关键部分。

注意最后要判断是否产生 shamtSign 和 jrSign 信号，如果指令为 sll 或 srl 则 shamtSign 置一，如果指令为 jr 则 jrSign 置零。

——ALUCtr.v——

```
always @ (aluOp or funct)
begin
    casex ({aluOp, funct})
        9'b000xxxxxx:   // lw or sw
            aluCtrOut = 4'b0010;    // add
        9'b001xxxxxx:   // beq
            aluCtrOut = 4'b0110;    // sub
        ...
        9'b110xxxxxx:  // j or jal
            aluCtrOut = 4'b0101;    // not change
    endcase

    if ({aluOp, funct} == 9'b101000000 || {aluOp, funct} == 9'b101000010)
     // sll or srl
        shamtSign = 1;
    else
        shamtSign = 0;

    if ({aluOp, funct} == 9'b101001000)     // jr
        jrSign = 1;
    else
        jrSign = 0;
end
```

3.3 算数逻辑运算单元 ALU

3.3.1 原理分析

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，该标志位将在之后的实验中用于条件转移指令的判断。

对比实验 3，我们加入了左移、右移运算和不进行运算，在 ALU 中，ALUCtr 信号与 ALU 执行的运算类型的对应关系如表 3所示。

表 3: ALUCtr 与 ALU 中的运算类型

ALUCtr	运算类型
0000	and
0001	or
0010	add
0011	sll
0100	srl
0101	nop
0110	sub
0111	slt
1100	nor

3.3.2 代码实现

在 Verilog 中已有左移和右移操作，只要对原代码稍作修改即可。

——ALU.v——

```
always @ (input1 or input2 or aluCtr) begin
        case(aluCtr)
        ...
        4'b0011
            ALURes = input2 << input1;
        4'b0100:    //srl
            ALURes = input2 >> input1;
        4'b0101:    //not change
            ALURes = input1;
        ...
        default: ALURes = 0;
        endcase
        if(ALURes == 0)
            Zero = 1;
        else
            Zero = 0;
    end
```

3.4 寄存器 Registers

3.4.1 原理分析

Registers 的原理与实验 4 中的基本相同，不再赘述。唯一需要改动的是需要加入一个 reset 信号，在 reset 信号为高电平时将寄存器初始化为全 0。

3.4.2 代码实现

只需要加入 reset 的判断即可，在这里给出，注意 reset 的判断要写在已有的判断是否写数据的 always 模块中，不然新加入一个 always 模块，当个两个“写” always 同时满足时，就会混乱不知赋什么值了。后续 dataMery 模块的 reset 与此相同。

——Registers.v——

```
...
    always @ (negedge clk)
    begin
        if (reset) begin
            for(i = 0;i <= 31;i = i+1) begin
                RegFile[i] = 0;
            end
        end
        else
            ...
    end
...
```

3.5 数据存储器 dataMemory

3.5.1 原理分析

dataMemory 的原理与实验 4 中的基本相同，不再赘述。需要注意的是因为仿真测试的代码是按字节寻址的，所以这里在寻址时输入的地址需要除以四。

3.5.2 代码实现

dataMemory 的实现与实验 4 中的基本相同，不再赘述。

3.6 指令存储器 InstMemory

3.6.1 原理分析

指令存储器 InstMemory 与数据存储器 dataMemory 相近，不过更为简单，其输入为 32 位地址，输出为地址所指向的空间存储的 32 位指令，如果地址超出空间则返回空指令。其接口如图 1所示。

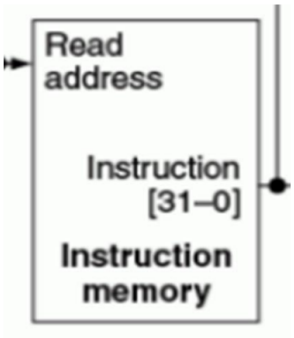

图 1: 指令存储器 InstMemory

3.6.2 代码实现

——InstMemory.v——

```
module InstMemory(
    input [31:0] address,
    output [31:0] inst
    );
    reg [31 : 0] instFile [0 : 63];
    integer i;
    assign inst = (address / 4 <= 63) ? instFile[address / 4] : 0;
endmodule
```

3.7 有符号扩展单元 signext

3.7.1 原理分析

有符号扩展单元 signext 原理与实验 4 中的基本相同，只需要加入一个是否为有符号拓展的输入信号 extSign 即可，这个信号也就是 Ctr 中新加入的产生的信号。

3.7.2 代码实现

具体实现代码如下所示，依然使用了花括号的拼接功能，用三目运算符实现了是否为有符号拓展的判断。

——signext.v——

```
module signExt(
    input extSign,
    input [15:0] inst,
    output [31:0] data
    );
    //extSign为1则有符号数位拓展，为0则是无符号数位拓展
    assign data = extSign ? { {16{inst[15]}}, inst[15 : 0] } : { 16'h0000,
    inst[15 : 0] };
endmodule
```

3.8 多路选择器 Mux

3.8.1 原理分析

多路选择器 Mux 的作用是根据输入的控制信号 select 和待选择数据 input1 和 input2，若 select 为 1，则输出 input1，否则输出 input2，示例图如图 2所示。

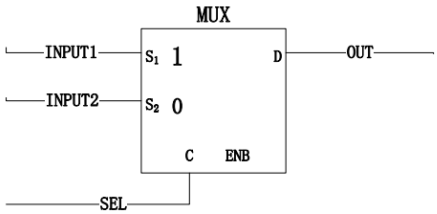

图 2: 多路选择器 Mux

3.8.2 代码实现

Mux 的实现比较简单，一个三目运算符即可。根据输入数据的大小，我们实现了 32 位 Mux 和 5 位 Mux，以下给出 32 位 Mux 的代码，5 位的数据同理，只要把数据大小改为五位即可。

——Mux32.v——

```
module Mux32(
    input sel,
    input [31 : 0] input1,
    input [31 : 0] input2,
    output [31 : 0] out
    );
```

```
    assign out = sel ? input1 : input2;
endmodule
```

3.9 程序计数器 PC

3.9.1 原理分析

程序计数器 PC 在每个时钟上升沿加 4，转移指令同样也会改变 PC，是这个简单 CPU 能够跑起来的关键，PC 可以用模块来实现，也可以在 Top.v 中实现，在本次实验中我们用模块来实现它。

3.9.2 代码实现

利用时钟进行 PC 的更新，时钟上升沿时，将下一时刻的 PC（加 4 或者跳转到别的地址）写入 PC 的输出。PC 模块中也要加入 reset 信号，触发时将 PC 重置。

注意在 always 模块中，敏感列表最好同时为触发沿或者同时为稳定电平而不是两者混杂。

——PC.v——

```
module PC(
    input clk,
    input reset,
    input [31 : 0] pcIn,
    output reg [31 : 0] pcOut
    );
    initial begin
        pcOut = -4;
    end

    always @ (posedge clk or posedge reset)
    begin
        if (reset)
            pcOut = -4;
        else
            pcOut = pcIn;
    end
endmodule
```

3.10 顶层模块 Top

3.10.1 原理分析

顶层模块 Top 是最关键的模块，其作用为将之前提到的模块实例化并进行合理的连接组装，这里给出单周期 MIPS 处理器的原理图如图 3所示，我们需要做的就是定义合适的变量将图中的元件连接起来。

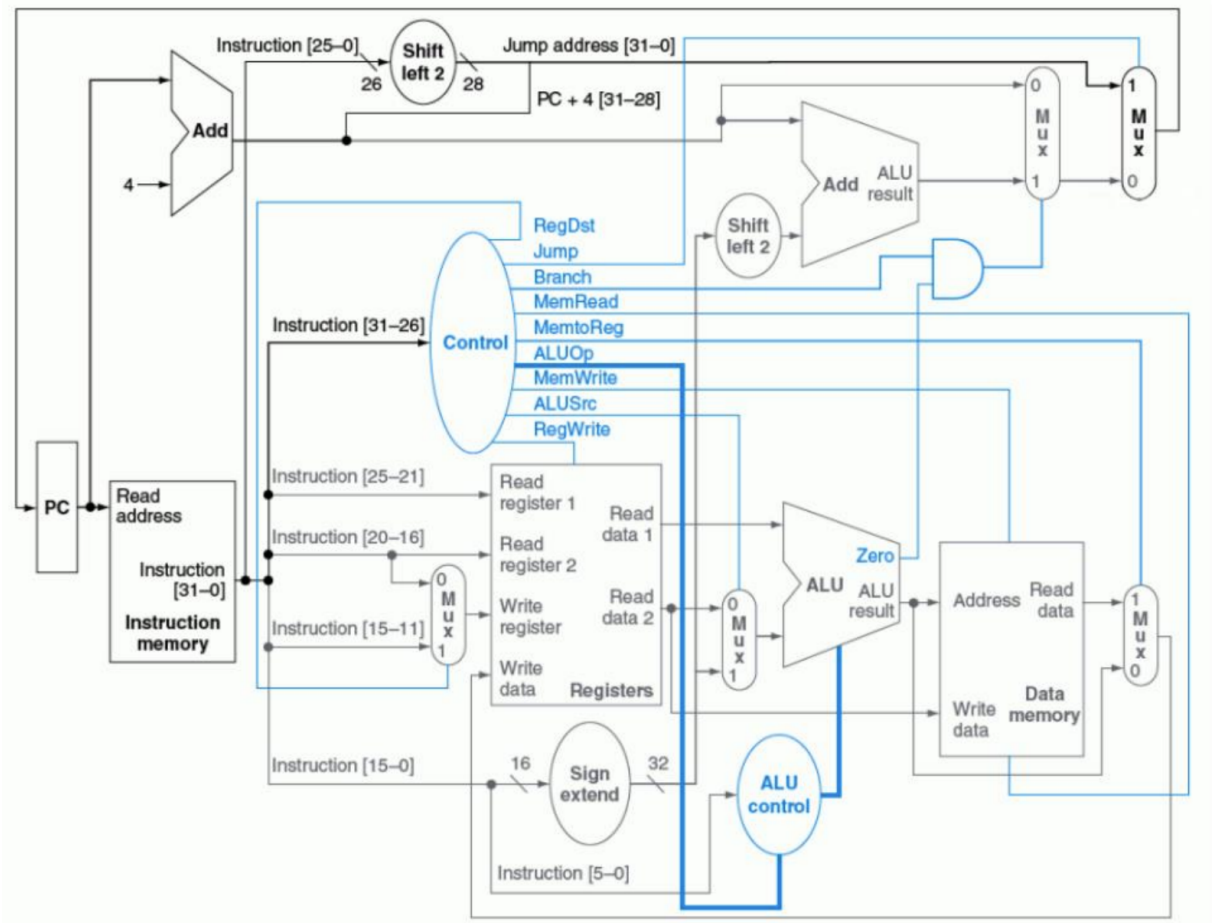

图 3: 简单的 MIPS 单周期处理器电路设计图

3.10.2 代码实现

根据已定义的模块定义其需要用到的变量并进行相应的实例化，根据设计图，除去基本的模块外，我们需要 7 个 32 位多路选择器和 2 个五位多路选择器来连接各部分。

——Top.v——

```
1    //PC
2    wire [31 : 0] PCIn;
3    wire [31 : 0] PCOut;
```

```
    wire [31 : 0] PCTemp1;
    wire [31 : 0] PCTemp2;
    //InstMemory
    wire [31 : 0] Inst;
    wire [31 : 0] InstAddr;
    //dataMemory
    wire [31 : 0] MemReadData;
    //Ctr
    wire RegDst, AluSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch, Jump,
    ExtSign, JalSign;
    wire [2 : 0] AluOp;
    //ALUCtr
    wire [2 : 0] ALUOp;
    wire [3 : 0] ALUCtrOut;
    wire ShamtSign, JrSign;
    //ALU
    wire [31 : 0] ALUInput1;
    wire [31 : 0] ALUInput2;
    wire [31 : 0] Zero;
    wire [31 : 0] ALURes;
    //signExt
    wire [31 : 0] ExtRes;
    //Registers
    wire [31 : 0] RegWriteData;
    wire [31 : 0] RegWriteDataTemp;
    wire [4 : 0] WriteReg;
    wire [4 : 0] WriteRegTemp;
    wire [31 : 0] RegOut1;      // rs
    wire [31 : 0] RegOut2;      // rt
```

在这里给出 Ctr 模块和 ALUCtr 模块和的实例化，可见 Ctr 输入为指令的高 6 位，ALUCtr 输入为 Ctr 输出的 ALUOp 和指令低六位 funct，而他们的输出又将作为其余各模块的输入控制信号。其余模块的实例化将在附录中给出。

——Top.v——

```
    // Ctr
    Ctr mainController(
        .opCode(Inst[31 : 26]),
        .regDst(RegDst),
        .aluSrc(AluSrc),
```

```
6           .memToReg(MemToReg),
7           .regWrite(RegWrite),
8           .memRead(MemRead),
9           .memWrite(MemWrite),
10          .branch(Branch),
11          .extSign(ExtSign),
12          .jalSign(JalSign),
13          .aluOp(ALUOp),
14          .jump(Jump)
15      );
16
17      // ALUCtr
18      ALUCtr aluController(
19          .aluOp(ALUOp),
20          .funct(Inst[5 : 0]),
21          .aluCtrOut(ALUCtrOut),
22          .shamtSign(ShamtSign),
23          .jrSign(JrSign)
24      );
```

4 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。

——Top_tb.v——

```
    Top processor(
        .clk(clk),
        .reset(reset)
    );
    always #10 clk = ~clk;       //T = 20ns
    initial begin
        $readmemh("E:/archlab/lab05/data.txt", memFile);
        $readmemb("E:/archlab/lab05/instruction.txt", instFile);
        reset = 1;
        clk = 0;
        #10 reset = 0;
        #200;
    end
```

data.txt 和 instruction.txt 中数据均来自于老师和助教给出的材料，其具体内容如下。

——data.txt——

```
00000009
0000000D
```

——汇编指令内容——

```
lw $1, 0($3)
lw $2, 4($3)
srl $4, $2, 1
sll $5, $4, 1
beq $5, $2, 1
add $8, $8, $1
srl $2, $2, 1
sll $1, $1,1
beq $2, $3,1
j 2
sw $8, 8($3)
```

——二进制乘法程序 instructions.txt——

```
1 10001100011000010000000000000000
2 10001100011000010000000000000100
3 00000000000000010001000000100010
4 00000000000001000010100001000000
5 00010000010001010000000000000001
6 00000001000000010100000000100000
7 00000000000000010000100000100010
8 00000000000000010000100001000000
9 00010000011000100000000000000001
10 00001000000000000000000000000010
11 10101100011010000000000000001000
```

仿真初始化时，所有 Registers 被置为 0。这是一个乘法程序，将 memFile 中的前两个程序相乘，乘法结果存在第 3 个数据中。memFile 第一个数据是 9，第二个数据是 13，预期最后结果是 117。

文件在仿真中对多种情况进行了检验，仿真结果如图 4所示。仿真结果正确，我们实现了简单的类 MIPS 单周期处理器。

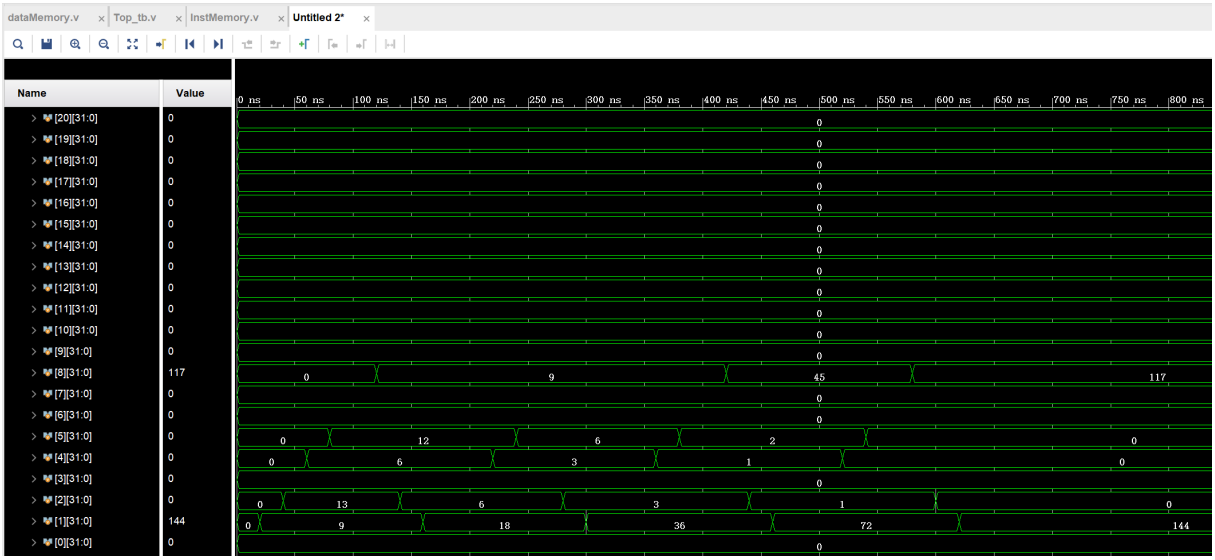

图 4: 仿真结果

5 总结与思考

本实验的目标是理解并实现支持 16 条指令的简单的 MIPS 单周期处理器，并通过编写激励文件，进行行为仿真验证了他们的正确性。

以实验 3、4 为基础，对已有模块进行修改，并适当引入新模块，最后进行连接组装的整个过程深入浅出，让我学到了很多，也惊叹于 MIPS 处理器的精妙。

在连接的过程中，老师提供的形象化的电路设计图给了我许多帮助，让我少走了许多弯路，这也让我认识到，直接进行代码上的抽象的设计或许是困难的，但是我们可以先做出相应的示意图，可以大大的减少难度，也更容易掌握目前工作的进度。

另一个我学到的技巧就是要善用行为仿真中的信号数据进行调试。在仿真中我曾经遇到这样的一个情况，运行仿真数据不符合预期。我发现有的数据中出现了 x，而不是具体地 0 或 1。通过不符合预期的那部分数据的控制信号，我定位到了出现问题的变量和相应的代码，才发现我少写了一个多路选择器。这样的调试过程给我的经验就是：遇到问题可以一点点的寻根溯源，找到出现问题的数据乃至代码进行更正，这或许就是执行仿真测试的作用。

总之，这次工程量颇大的实验让我受益匪浅，相信它也会给我的下一次实验提供不少经验。

6 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。

7 附录：完整的 Top.v 代码

——Top.v——

```
module Top(
    input clk,
    input reset
    );

    //PC
    wire [31 : 0] PCIn;
    wire [31 : 0] PCOut;
    wire [31 : 0] PCTemp1;
    wire [31 : 0] PCTemp2;

    //InstMemory
    wire [31 : 0] Inst;
    wire [31 : 0] InstAddr;

    //dataMemory
    wire [31 : 0] MemReadData;

    //Ctr
    wire RegDst, AluSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch, Jump,
    ExtSign, JalSign;
    wire [2 : 0] AluOp;

    //ALUCtr
    wire [2 : 0] ALUOp;
    wire [3 : 0] ALUCtrOut;
    wire ShamtSign, JrSign;

    //ALU
    wire [31 : 0] ALUInput1;
    wire [31 : 0] ALUInput2;
    wire [31 : 0] Zero;
    wire [31 : 0] ALURes;

    //signExt
    wire [31 : 0] ExtRes;
```

```
36
37      //Registers
38      wire [31 : 0] RegWriteData;
39      wire [31 : 0] RegWriteDataTemp;
40      wire [4 : 0] WriteReg;
41      wire [4 : 0] WriteRegTemp;
42      wire [31 : 0] RegOut1;      // rs
43      wire [31 : 0] RegOut2;      // rt
44
45      // PC
46      PC pc(
47          .pcIn(PCIn),
48          .clk(clk),
49          .reset(reset),
50          .pcOut(PCOut)
51      );
52
53      // InstMemory
54      InstMemory instMemory (
55          .address(PCOut),
56          .inst(Inst)
57      );
58
59      // dataMemory
60      dataMemory dataMemory (
61          .clk(clk),
62          .address(ALURes),
63          .writeData(RegOut2),
64          .memWrite(MemWrite),
65          .memRead(MemRead),
66          .readData(MemReadData)
67      );
68
69      // Register
70      Registers registers (
71          .readReg1(Inst[25 : 21]),
72          .readReg2(Inst[20 : 16]),
73          .writeReg(WriteReg),
74          .writeData(RegWriteData),
```

```
75          .regWrite(RegWrite & (~JrSign)),
76          .clk(clk),
77          .reset(reset),
78          .readData1(RegOut1),
79          .readData2(RegOut2)
80      );
81
82      // signExt
83      signExt signExt (
84          .extSign(ExtSign),
85          .inst(Inst[15 : 0]),
86          .data(ExtRes)
87      );
88
89      // Ctr
90      Ctr mainController(
91          .opCode(Inst[31 : 26]),
92          .regDst(RegDst),
93          .aluSrc(AluSrc),
94          .memToReg(MemToReg),
95          .regWrite(RegWrite),
96          .memRead(MemRead),
97          .memWrite(MemWrite),
98          .branch(Branch),
99          .extSign(ExtSign),
100         .jalSign(JalSign),
101         .aluOp(ALUOp),
102         .jump(Jump)
103     );
104
105     // ALUCtr
106     ALUCtr aluController(
107         .aluOp(ALUOp),
108         .funct(Inst[5 : 0]),
109         .aluCtrOut(ALUCtrOut),
110         .shamtSign(ShamtSign),
111         .jrSign(JrSign)
112     );
113
```

```
114     // ALU
115     ALU alu(
116         .input1(ALUInput1),
117         .input2(ALUInput2),
118         .aluCtr(ALUCtrOut),
119         .zero(Zero),
120         .aluRes(ALURes)
121     );
122 
123     //Mux32 and Mux5
124 
125     Mux32 rs_shamt_selector (
126         .sel(ShamtSign),
127         .input1({27'b000000000000000000000000000, Inst[10 : 6]}),
128         .input2(RegOut1),
129         .out(ALUInput1)
130     );      // ShamtSign ? Inst[10 : 6] : RegOut1(rs)
131 
132     Mux32 rt_ext_selector (
133         .sel(AluSrc),
134         .input1(ExtRes),
135         .input2(RegOut2),
136         .out(ALUInput2)
137     );      // AluSrc ? ExtRes : RegOut2(rt)
138 
139     Mux5 rt_rd_selector (
140         .sel(RegDst),
141         .input1(Inst[15 : 11]),
142         .input2(Inst[20 : 16]),
143         .out(WriteRegTemp)
144     );      // RegDst ? rd : rt
145 
146     Mux5 rtrd_31_selector (
147         .sel(JalSign),
148         .input1(5'b11111),
149         .input2(WriteRegTemp),
150         .out(WriteReg)
151     );      // JalSign ? 31 : (rt or rd)
152 
```

```
153     Mux32 mem_alu_selector (
154         .sel(MemToReg),
155         .input1(MemReadData),
156         .input2(ALURes),
157         .out(RegWriteDataTemp)
158     );      // MemToReg ? MemReadData : ALURes
159 
160     Mux32 jal_selector (
161         .sel(JalSign),
162         .input1(PCOut + 4),
163         .input2(RegWriteDataTemp),
164         .out(RegWriteData)
165     );      // JalSign ? PCOut + 4 : RegWriteDataTemp
166 
167     Mux32 branch_selector (
168         .sel(Branch & Zero),
169         .input1(PCOut + 4 + (ExtRes << 2)),
170         .input2(PCOut + 4),
171         .out(PCTemp1)
172     );      // Branch and Zero ? Branch Jemp : Branch not Jump
173 
174     Mux32 jump_selector (
175         .sel(Jump),
176         .input1(((PCOut + 4) & 32'hf0000000) + (Inst[25 : 0] << 2)),
177         .input2(PCTemp1),
178         .out(PCTemp2)
179     );      // Jump ? JumpPC : PC
180 
181     Mux32 jr_selector (
182         .sel(JrSign),
183         .input1(RegOut1),
184         .input2(PCTemp2),
185         .out(PCIn)
186     );      // Jr ? JrPC : PC
187 endmodule
```