计算机系统结构实验

lab5: 单周期处理器设计

周旭东 521021910829

2023 April

摘要

在 lab5: 单周期处理器设计实验中，通过学习 MIPS 单周期处理器结构图，理解简单的类 MIPS 单周期处理器的工作原理。使用并修改 lab3，lab4 所构建的 ALU，Register 等模块，使其支持 16 条 MIPS 指令；根据结构图增加所需新的模块如 InstMem 和 PC。在 Top 顶层文件中，用硬件描述语言 Verilog 进行模块的连线，并使用过 Vivado 进行指令仿真，以验证试验结果。

目录

1 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) 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 的实设计与实现
3. 使用 Vivado 进行仿真测试

2 实验原理

2.1 整体设计原理

进行处理器设计有五个关键步骤

1. 分析指令，得出对数据通路的需求
2. 选择数据通路上合适的组件
3. 连接组件构成数据通路
4. 分析每一条指令的实现，以确定控制信号
5. 集成控制信号，完成控制逻辑

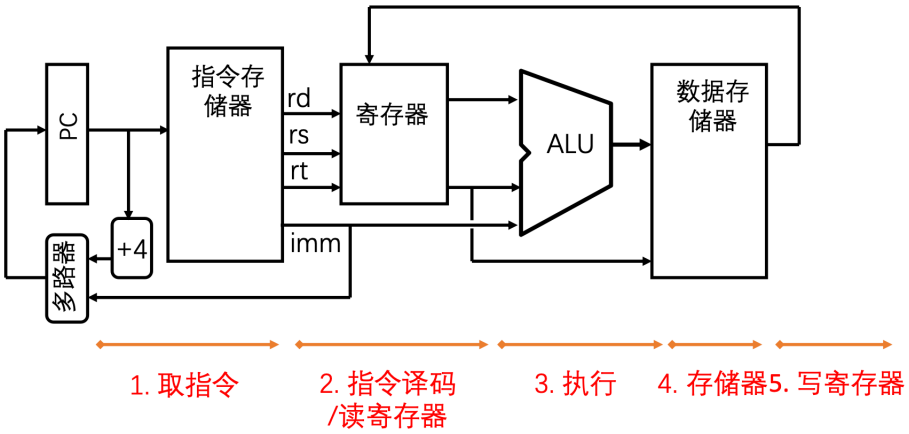

图 1: 数据通路结构

图 1展示了单周期处理器的数据通路结构。其分为五个阶段：取指，译码和读寄存器，执行，存储器处理，写寄存器。

助记符	指令格式					
Bit #	31..26	25..21	20..16	15..11	10..6	5..0
R-type	op	rs	rt	rd	shamt	func
add	0	rs	rt	rd	0	100000
sub	0	rs	rt	rd	0	100010
and	0	rs	rt	rd	0	100100
or	0	rs	rt	rd	0	100101
slt	0	rs	rt	rd	0	101010
sll	0	0	rt	rd	shamt	0
srl	0	0	rt	rd	shamt	10
jr	0	rs	0	0	0	1000
I-type	op	rs	rt	immediate		
addi	1000	rs	rt	immediate		
andi	1100	rs	rt	immediate		
ori	1101	rs	rt	immediate		
lw	100011	rs	rt	immediate		
sw	101011	rs	rt	immediate		
beq	100	rs	rt	immediate		
J-type	op	address				
j	10	address				
jal	11	address				

表 1: 需要支持的 16 条指令及其相关信息

本次实验要求完成支持 16 条指令的处理器设计。表格 1给出了本次试验需要考虑的指令及其相应格式。根据这 16 条指令，给出相应完整通路结构图 2。

在结构图的帮助下，首先实现各个模块，其次通过程序连接各个数据线。失序性的模块例如 PC 和存储器 DataMemery 需要和系统时钟 clk 相关联，当下降沿到来时做出相应改变。非时序性模块例如多路选择器 MUX 和符号扩展器 SignExt 则将信号线内容实时更新。

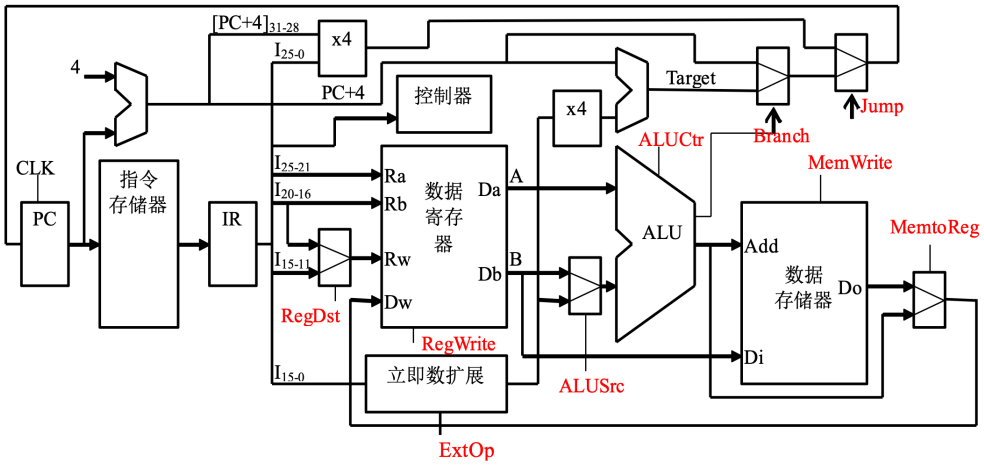

图 2: MIPS 处理器结构图

2.2 命名规则及所需模块

为避免重名和方便辨认，首先定义命名规则如下：

- 数据通路信号线定义为 `wire [a:b] INST`。即 全部大写字母。
- 模块命名为 `Register register_module()`。即 小写字母_module。
- 模块内信号线命名为 `input [31:0] data0`。即 大小写混杂（需要reg时开头小写）。
- 模块内寄存器命名为 `reg [3:0] ALUCtrOut`。即将所 `assign` 的命名改为部分大写，通常开头大写。

所需模块如表 2所示。

模块	描述
PCupdate	处理例如 branch 和 jump 指令，使下一 PC 正确
PC	在时钟下降沿到来时更新 PC 的值为 PCupdate，并送入取指
InstMem	指令寄存器，得到 PC 后给出相应位置指令
Ctr	控制单元，处理指令给出各种控制信号
ALUCtr	ALU 控制单元，通过 Ctr 给出的 ALUOP 决定 ALU 操作
ALU	通过 ALUCtr 给出的信号进行运算
signext	进行带符号或不带符号拓展
dataMemory	存储器，下降沿时进行写操作，根据 Ctr 进行读操作
MUX	32 位的多路选择器
MUX_5	5 位的多路选择器

表 2: 所需模块及其描述

3 模块设计

3.1 状态单元模块设计

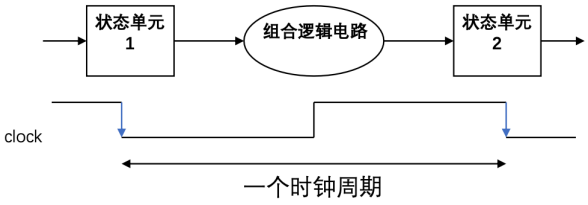

图 3: 时钟周期和状态单元关系

状态单元随着时钟下降沿进行操作，一般在一个时钟周期中只有一个状态单元工作。状态单元有：PC，寄存器文件，存储器。除执行阶段不需要等待时钟，每个阶段和时钟相关，有条不紊进行。

3.1.1 指令计数器 PC

PC 的设计较为简单，在时钟下降沿到来时将下一 PC 输出，如果 reset 为 1，则将 PCout 赋值为 0，否则将 PCout 赋值为 PCin。这个过程表明了当系统启动或进行重置操作时，将 PC 寄存器重置为 0，否则 PC 寄存器将被赋予下一条指令的地址。

实现代码为

```
always @ (posedge clk or reset)
    begin
        if (reset)
            PCout = 0;
        else
            PCout = PCin;
    end
```

3.1.2 寄存器 Registers

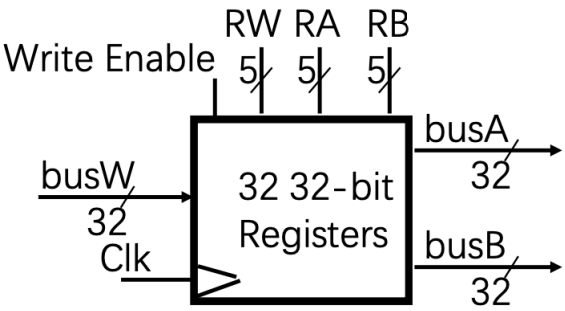

图 4: 寄存器结构

图 4给出了 MIPS 寄存器文件结构，该模块由 32 个 32 位寄存器构成；在 Clk 下降沿进行写操作，设计读操作与时钟无关；另外设计 reset 重制信号下的寄存器清空操作，将其与 clk 下降沿写在同一“always”语块中，避免同时满足时赋值混乱。清空方法为通过循环将寄存器文件中每个寄存器值设为 0。

寄存器的输入有写使能信号、32 位的写入数据、选通的寄存器地址。输出有选通的两个读寄存器的数据。在寄存器中，写使能信号用于控制是否允许写入数据，32 位的写入数据是将要存储在指定寄存器中的数据，选通的寄存器地址用于指定要进行读写的寄存器。读操作则是通过选通读取的寄存器地址来输出该寄存器中存储的数据。

表 3定义了寄存器模块各个输入输出信号量的定义并给出了相应功能的描述。

输入/输出量	信号线	描述	定义的信号量
输入	Clk	改变寄存器状态需要时钟边沿触发	clk
	Write Enable	写寄存器使能信号	regWrite
	busW（32位）	32位输入，指定写入数据寄存器	writeData
	RW（5位）	选通Rw指定的寄存器	writeReg
	RA（5位）	选通RA指定的寄存器	readReg1
	RB（5位）	选通RB指定的寄存器	readReg2
输出	busA（32位）	RA有效时输出读取的数据	readData1
	busB（32位）	RB有效时输出读取的数据	readData2

表 3: 寄存器各信号量及其定义

3.1.3 存储器 memory

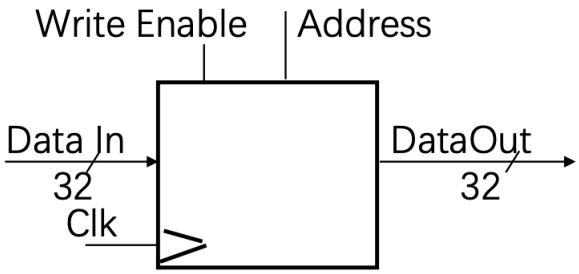

图 5: 存储器结构

图 5展示了存储器的结构，其中自定义内存大小为 256 * 32 位。

存储器的主要功能是存储和读取数据，可以根据地址信号寻址到对应的存储单元。对于 RAM 和 ROM 等可读写存储器，可以使用 Write Enable 信号控制数据的写入；对于只读存储器 ROM，没有写使能信号，只能从指定地址读取数据。它有一个 32 位的地址总线，可以寻址 2^{32} 个不同的地址，也就是 4GB 的空间。存储器的输入信号包括时钟边沿触发信号 Clk、写存储器使能信号 Write Enable、要写入存储器的数据 Data In，以及指定存储器地址的地址信号 Address。存储器的输出信号是读取的数据 DataOut，它的大小也是 32 位。

表 4定义了存储器模块各个输入输出信号量的定义并给出了相应功能的描述。时钟下降沿进行写存储器操作，读操作在读信号和地址的控制下持续进行。

输入/输出量	信号线	描述	定义的信号量
输入	Clk	改变寄存器状态需要时钟边沿触发	clk
	Write Enable	写存储器使能信号	memWrite
	Data In（32位）	写入存储器数据	writeData
	Address（32位）	指定存储器地址	address
输出	DataOut（32位）	输出读取的数据	readData

表 4: 存储器各信号量及其定义

3.2 非状态单元模块设计

非状态单元即不在时钟控制下进行状态改变，而根据输入情况随时产生输出信号。在我的设计中，包含指令存储器，主控制器，ALU 控制器，ALU，符号拓展器，PC 更新模块，多路选择器。

3.2.1 指令存储器 Instruction memory

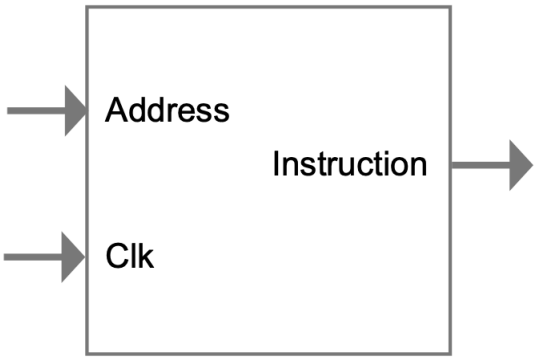

图 6: 指令存储器结构

图 6展示了指令存储器的结构，其中自定义内存大小为 256 * 32 位。

输入信号包括时钟信号和 32 位指令存储地址。时钟信号用于改变寄存器状态，并且在指令存储器中，时钟边沿触发是指令读取的触发条件。32 位指令存储地址用于指定要读取的指令在存储器中的位置。

输出信号是 32 位的指令，它是从指令存储器中读取的指令。指令存储器会根据输入的 32 位指令存储地址，读取对应位置的指令，并将其作为输出信号 DataOut 输出给 CPU。

表 5定义了指令存储器模块各个输入输出信号量的定义并给出了相应功能的描述。

输入/输出量	信号线	描述	定义的信号量
输入	Clk	改变寄存器状态需要时钟边沿触发	clk
	Address（32位）	指令存储地址	readAddr
输出	DataOut（32位）	输出读取的指令	instr

表 5: 指令存储器各信号量及其定义

指令存储器实现代码如下：

```
module InstMem(
    input [31 : 0] readAddr,
    input clk,
    input reset,
    output [31 : 0]
    );
```

```
7
8       reg [31:0] instMEM [0:255];//set size as 256
9       reg [31:0] n;
10
11      assign instr = instMEM[readAddr >> 2];
12  endmodule
```

指令存储器的代码由文件直接写入，无须设置写存储器功能。

```
1  $readmemb("Y:/Vivado/lab05/mem_inst.dat",top.instmem_module.instMEM);
```

这段代码给出了将绝对路径”Y:/Vivado/lab05/” 下文件“mem_inst.dat”中的数据读入指令存储器的方法，其中，top 为顶层文件的实例化，instMEM 为模块 instmem_module 中定义的存储单元 `reg [31:0] instMEM [0:255]`。

3.2.2 主控制器 Ctr

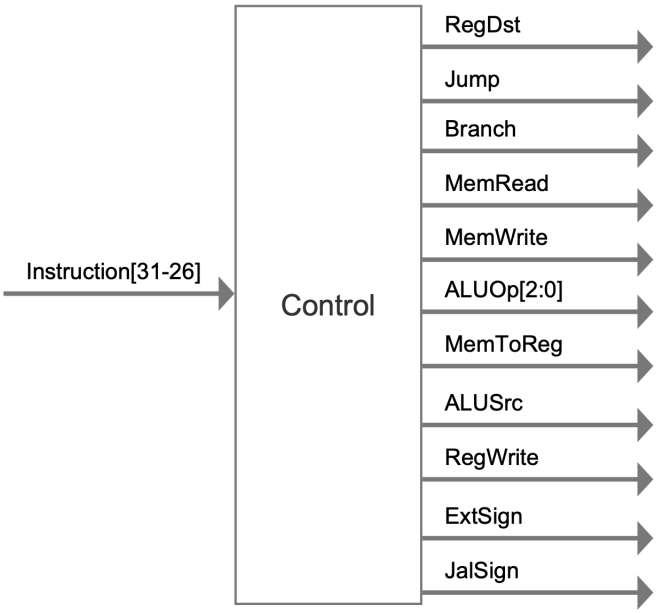

图 7: 主控制器结构

图 7展示了主控制器的结构。主控制器模块通过指令第 26-31 位获取代码指令及类型信息，根据指令给出各个位置的控制信号，并向 ALU 发出 ALUOp 信号，指示其进行相应运算操作。

RegDst 信号用于选择目标寄存器的编号。Jump 信号用于指示 CPU 是否需要进行无

条件跳转。Branch 信号用于控制分支操作，判断分支是否需要被执行。MemRead 信号和 MemWrite 信号用于控制存储器的读和写操作。ALUOp 信号用于选择 ALU 运算类型，决定 CPU 要执行哪种运算。ALUSrc 信号用于选择 ALU 第二个操作数的来源，可以是寄存器或者立即数。MemToReg 信号用于选择从存储器中读取数据还是从 ALU 中读取数据写入寄存器。RegWrite 信号用于控制寄存器的写操作。ExtSign 信号用于指示是否进行符号扩展，即将数据的符号位扩展到高位。JalSign 信号用于控制 jal 指令的执行。

表 6定义了主控制器模块各个输入输出信号量的定义并给出了相应功能的描述。

输入/输出量	信号线	描述	定义的信号量
输入	Instruction（5位）	指令的第 31-26 位，决定指令类型	opCode
输出	RegDst	目标寄存器选择信号	regDst
	Jump	无条件跳转信号	jump
	Branch	分支控制信号	branch
	MemRead	存储器读信号	memRead
	MemWrite	存储器写信号	memWrite
	ALUOp（3位）	ALU运算类型	aluOp
	ALUSrc	ALU 第二个操作数来源选择信号	aluSrc
	MemToReg	从存储器写到寄存器选择信号	memToReg
	RegWrite	寄存器写信号	regWrite
	ExtSign	是否带符号扩展信号	exrSign
	JalSign	jal指令信号	jalSign

表 6: 主控制器各信号量及其定义

在 lab3 中，ALUOp 只有两位，支持九条指令，为支持 16 条指令，扩展 ALUOp 为三位，部分拓展的指令及其对应控制信号由表 7给出。

Inst	OpCode	ALUop	RegDst	ALUsrc	MemTReg	RegWrite	MemRead	MemWrite	Branch	Jump	ExtSign	JalSign
lw	100011	000	0	1	1	1	1	0	0	0	1	0
sw	101011	000	0	1	0	0	0	1	0	0	1	0
addi	001000	001	0	1	0	1	0	0	0	0	1	0
beq	000100	110	0	0	0	0	0	0	1	0	1	0
ori	001101	010	0	1	0	1	0	0	0	0	1	0
andi	001100	011	0	1	0	1	0	0	0	0	0	0
j	000010	100	0	0	0	0	0	0	0	1	0	0
jal	000011	100	0	0	0	1	0	0	0	1	0	1
Rtype	000000	101	1	0	0	1	0	0	0	0	0	0

表 7: 部分扩展指令对应信号

3.2.3 ALU 控制器

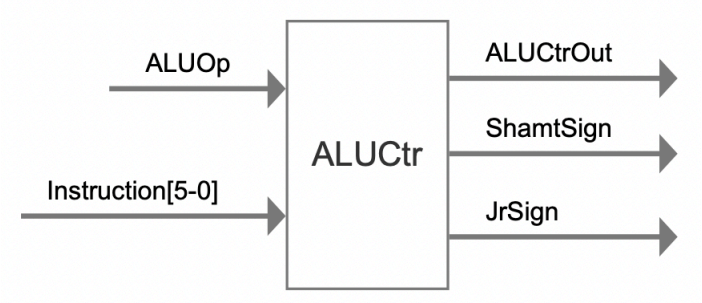

图 8: ALU 控制器结构

图 8展示了 ALU 控制器的结构。ALU 控制器通过主控制器给出的 ALUOp，结合指令第 0-5 位给出的 funct 信息，确定 16 条指令具体对应的 ALU 运算类型（R type 的解析无法在 Ctr 中完成）。并依此确定 ShamtSign 信号和 JrSign 信号。

输入信号包括 ALU 操作类型指令和指令中的 funct 字段。ALU 操作类型指令由 3 位二进制数表示，它指定了 ALU 执行的具体运算类型，例如加法、减法、与、或、异或等。指令中的 funct 字段给出了更详细的指令信息，它指定了 ALU 操作类型指令中的某些操作的具体实现方式，例如指定 ALU 执行移位操作的位数。

输出信号包括解析后的 ALU 具体运算类型，指令 shamt 段作为 ALU 输入信号和跳转至寄存器信号。解析后的 ALU 具体运算类型是一个 4 位二进制数，它指定了 ALU 具体执

行的操作类型，例如加法、减法、与、或、异或等。指令 shamt 段作为 ALU 输入信号是一个布尔信号，它指示是否将指令中的 shamt 字段作为 ALU 的输入信号。跳转至寄存器信号是一个布尔信号，它指示是否将寄存器中的值作为 ALU 的输入信号。

表 8定义了 ALU 控制器模块各个输入输出信号量的定义并给出了相应功能的描述。

输入/输出量	信号线	描述	定义的信号量
输入	ALUOp（3位）	ALU运算类型指令	aluOp
	Instruction[5-0]	指令第0-5位，给出funct	funct
输出	ALUCtrOut（4位）	解析后的ALU具体运算类型	aluCtrOut
	ShamtSign	指令 shamt 段作为 ALU 输入信号	shamtSign
	JrSign	跳转至寄存器信号	jrSign

表 8: ALU 控制器各信号量及其定义

3.2.4 逻辑运算单元 ALU

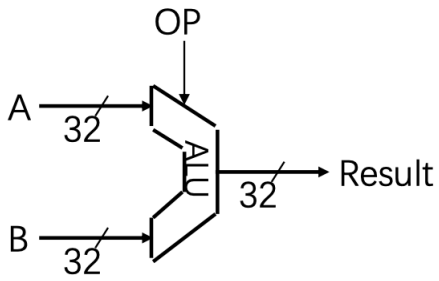

图 9: ALU 结构

图 9展示了 ALU 的结构。ALU 接收两个 32 位的输入数据（InputA 和 InputB），根据控制信号（ALUOp 和 aluCtr）选择相应的算术或逻辑运算类型，然后执行运算并输出运算结果（Output）。控制信号还会产生一个标志信号（ZeroSign），指示运算结果是否为零。

表 9定义了 ALU 模块各个输入输出信号量的定义并给出了相应功能的描述。

输入/输出量	信号线	描述	定义的信号量
输入	ALUOp（4位）	ALU运算类型指令	input1
	InputA（32位）	第一个32位输入	input2
	InputB（32位）	第二个32位输入	aluCtr
输出	Output（32位）	进行相应运算后的32位输出	aluRes
	ZeroSign	标志运算结果是否为0	zero

表 9: ALU 各信号量及其定义

ALUCtr	ALU操作	实现代码
0010	add	ALURes = input1 + input2
0110	sub	ALURes = input1 - input2
0001	or	ALURes = input1 | input2
0000	and	ALURes = input1 & input2
1100	nor	ALURes = ~(input1 | input2)
0111	slt	ALURes = ($signed(input1) < $signed(input2))
0011	sll	100ALURes = input2 << input1
0100	srl	ALURes = input2 >> input1
0101	keep	ALURes = input1

表 10: 各 ALUOp 对应的 ALU 操作及其实现

表 10定义了 ALU 模块对各个 ALUOp 信号下的运算方式及其代码实现。其中，slt 比较两个有符号整数 input1 和 input2 的大小，如果 input1 小于 input2，则结果为 1，否则结果为 0。具体地，$signed() 是一个 SystemVerilog 函数，用于将无符号数转换为有符号数，以便进行带符号的比较。因此，$signed(input1) 和 $signed(input2) 将无符号输入转换为有符号数。然后，这两个有符号数被比较，比较结果被存储在 ALURes 变量中。

3.2.5 符号拓展器 SignExt

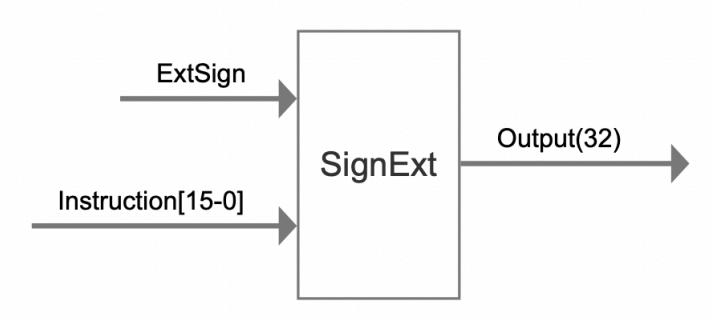

图 10: 符号拓展器结构

表 10展示了符号拓展器 SignExt 的结构。以表达式 连接字符串，?... : ... 作为条件判断语句，写出 SignExt 代码如下

```
module signext(
    input extSign,
    input [15:0] inst,
    output [31:0] data
    );
    reg [31:0] Data;
    reg [15:0] Inst;

    assign data = (extSign ? {{16{inst[15]}}, inst[15:0]} : {16'h0000, inst[15:0]});

endmodule
```

条件由表达式”extSign” 表示，假设它是一个布尔变量或表达式。如果”extSign” 为真，则赋给”data” 的值是由 16 个”inst[15]” 组成的 16 位值与”inst” 的最低 16 位连接起来的结果。如果”extSign” 为假，则赋给”data” 的值是由 16 个零组成的 16 位值与”inst” 的最低 16 位连接起来的结果。

3.2.6 多路选择器 MUX

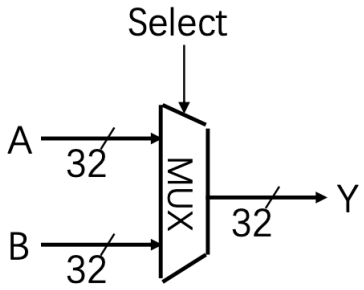

图 11: 多路选择器结构

图 9展示了多路选择器 MUX 的结构。其有多个输入端和一个输出端。Select 控制信号确定了要输出哪个输入信号。如果控制信号为 0，则输出将是第一个输入信号，否则输出将是第二个输入信号。其代码设计如下

```
module MUX(
    input Select,
    input [31 : 0] data0,
    input [31 : 0] data1,
    output [31 : 0] data
    );
    assign data = Select ? data0 : data1;
endmodule
```

另外，考虑到所设计的 CPU 结构中，在写寄存器操作中存在需要对 5 位数据进行多路选择，另设计 MUX_5 模块如下：

```
module MUX_5(
    input Select,
    input [4 : 0] data0,
    input [4 : 0] data1,
    output [4 : 0] data
    );
    assign data = Select ? data0 : data1;
endmodule
```

3.2.7 PC 更新模块

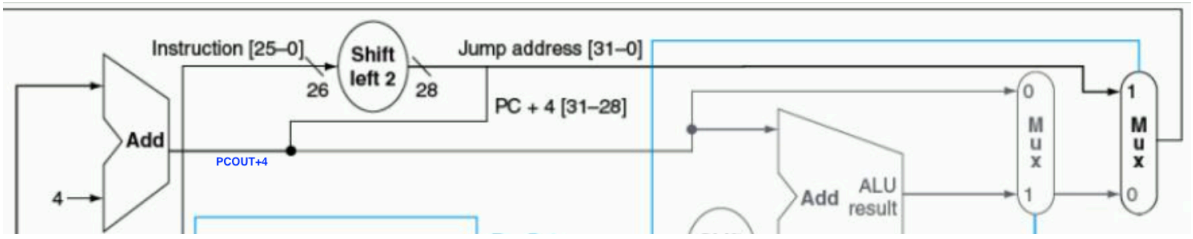

图 12: 复杂的 PC update 结构

在原设计结构中，PC 更新策略带来大量包含跳转，分支等多路选择情况。如图 12所示，若在 Top 中通过硬件描述容易发生错误，考虑将其打包成一个 PCupdate 模块。

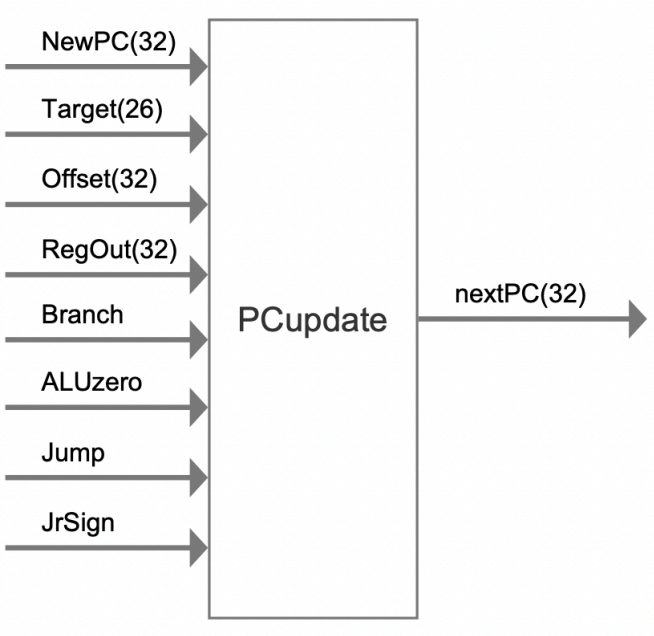

图 13: PC update 模块

图 13展示了打包设计后 PC update 的结构。该模块包含了多个输入和一个输出，用于计算下一条指令的地址（即下一次程序计数器 PC 的值）。

该模块首先将 target 左移 2 位，得到一个 28 位的地址 shtarget。然后，将 newPC 的高 4 位和 shtarget 拼接起来，得到一个 32 位的地址 jumptarget，用于实现无条件跳转。

其次，计算分支指令的目标地址 branchtarget，通过将 newPC 和 offset 相加得到。同时，根据 Branch 和 Aluzero 的值，选择要跳转到的目标地址。如果 Branch 和 Aluzero 均

为真，则跳转到 branchtarget，否则保持在当前地址 newPC。

最后，通过判断 Jump 和 JrSign 的值，选择要跳转到的地址。如果 Jump 为真，则跳转到 jumptarget，否则跳转到 nonjumptarget。如果 JrSign 为真，则跳转到寄存器中的地址 regOut，否则跳转到 nonretaddr。计算结果赋给 nextPC 输出端口。代码实现如下：

```
module PCupdate(
    input [31:0] newPC,
    input [25:0] target,
    input [31:0] offset,
    input [31:0] regOut,
    input Branch,
    input Aluzero,
    input Jump,
    input JrSign,
    output [31:0] nextPC
    );

    wire [27:0] shtarget;
    assign shtarget = target << 2;

    wire [31:0] jumptarget;
    assign jumptarget = {newPC [31:28], shtarget};

    wire [31:0] branchtarget;
    assign branchtarget = newPC + offset;

    wire [31:0] nonjumptarget;
    assign nonjumptarget = (Branch & Aluzero) ? (branchtarget) : (newPC);

    wire [31:0] nonretaddr;
    assign nonretaddr = (Jump) ? (jumptarget) : (nonjumptarget);
    assign nextPC = (JrSign) ? (regOut) : (nonretaddr);

endmodule
```

3.3 顶层 Top 模块设计

Top 模块将各个部分进行定义和连接，通过结构图，连接相应数据线两端，使得所有模块构成一个整体，从而构建完整的 MIPS 处理器。

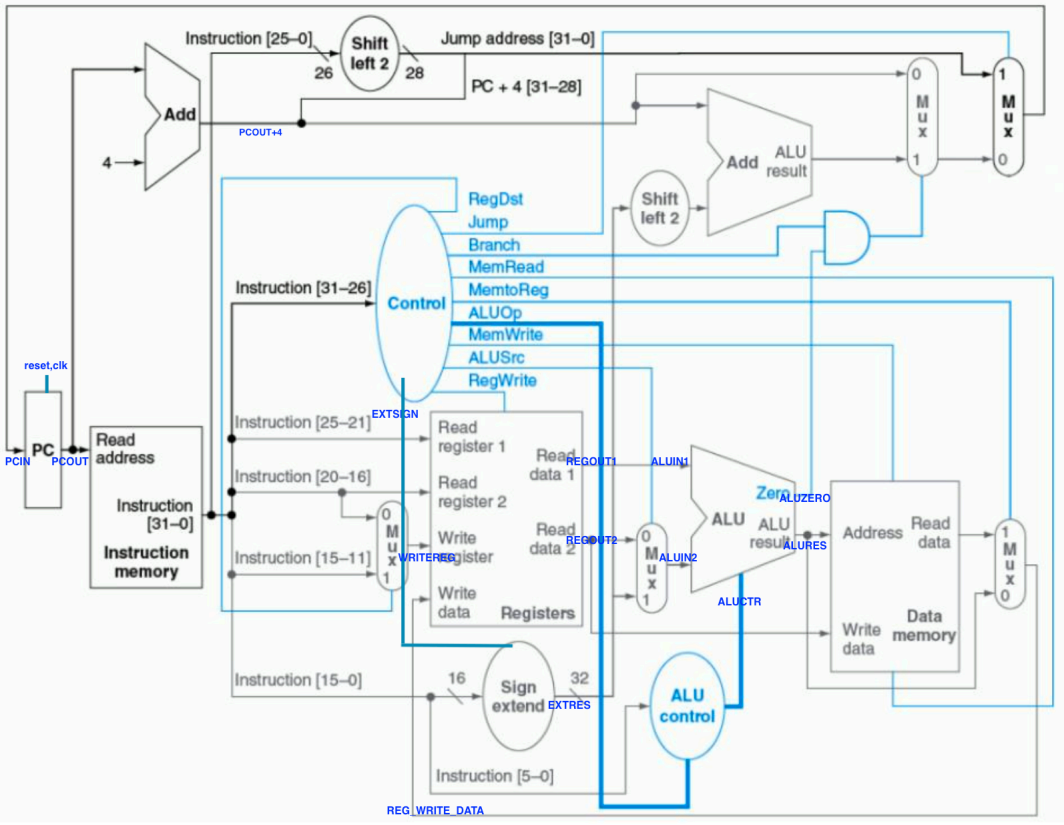

图 14: MIPS 处理器结构图及部分连线命名

对照结构图 14，进行各个模块的连线。Top 模块的输入部分为 clk 与 reset，这两个信号将在激励模块进行实例化。

```
    wire [31:0] INST;
    wire REGDST;
    wire ALUSRC;
    wire MEMTOREG;
    wire REGWRITE;
    wire MEMREAD;
    wire MEMWRITE;
    wire BRANCH;
    wire EXTSIGN;
    wire JALSIGN;
    wire [2:0] ALUOP;
    wire JUMP;
    // Control Unit
    Ctr ctr_module (
```

```
        .opCode(INST[31 : 26]),
        .regDst(REGDST),
        .aluSrc(ALUSRC),
        .memToReg(MEMTOREG),
        .regWrite(REGWRITE),
        .memRead(MEMREAD),
        .memWrite(MEMWRITE),
        .branch(BRANCH),
        .aluOp(ALUOP),
        .jump(JUMP),
        .extSign(EXTSIGN),
        .jalSign(JALSIGN)
    );
```

此段代码给出了对主控制器模块连线的过程。

首先，定义了一些 wire 类型的信号（[31:0] 为定义该信号为 32 位宽度），用于连接到控制单元（Ctr）的输入端口。这些信号用于控制 CPU 中的不同部件的行为。

接着，将这些控制信号连接到名为 ctr_module 的控制单元的输入端口中。这样，ctr_module 就可以根据当前指令的操作码（即 INST 的高 6 位）来生成相应的控制信号，从而控制 CPU 中的不同部件的行为。

相似地依次进行各个部分连线。完整代码在附录中给出。

值得一提的是，MUX 模块需要根据结构与功能要求进行多次复用，其代码实现为：

```
    wire [31 : 0] REG_WRITE_DATA_T;
    wire [4 : 0] WRITE_REG_TEMP;

    // INST[10:6] or rs
    MUX rs_shamt_selector (
        .Select(SHAMTSIGN),
        .data0({27'h0000000, INST[10 : 6]}),
        .data1(REGOUT1),
        .data(ALUIN1)
    );

    // EXTRES or rt
    MUX rt_ext_selector (
        .Select(ALUSRC),
        .data0(EXTRES),
        .data1(REGOUT2),
        .data(ALUIN2)
    );
```

```
19
20      // MEM READ DATA or ALURES
21      MUX mem_alu_selector (
22          .Select(MEMTOREG),
23          .data0(MEM_READ_DATA),
24          .data1(ALURES),
25          .data(REG_WRITE_DATA_T)
26      );
27
28      // PCOUT + 4 or REG WRITE DATA T
29      MUX jal_selector (
30          .Select(JALSIGN),
31          .data0(PCOUT + 4),
32          .data1(REG_WRITE_DATA_T),
33          .data(REG_WRITE_DATA)
34      );
35
36      // rd or rt
37      MUX_5 rt_rd_selector (
38          .Select(REGDST),
39          .data0(INST[15 : 11]),
40          .data1(INST[20 : 16]),
41          .data(WRITE_REG_TEMP)
42      );
43
44      // 11111 or rt/td
45      MUX_5 rtrd_31_selector (
46          .Select(JALSIGN),
47          .data0(5'b11111),
48          .data1(WRITE_REG_TEMP),
49          .data(WRITEREG)
50      );
```

4 结果测试

添加激励文件，给出时钟，reset 信号，并将相应初始化数据导入。代码如下：

```
1  module Single_Cycle_CPU_tb(
2      );
3      reg clk;
4      reg reset;
```

```
5
6       Top top(
7           .clk(clk),
8           .reset(reset)
9       );
10      initial begin
11          $readmemb("Y:/Vivado/lab05/mem_inst.dat", top.instmem_module.instMEM);
12          $readmemh("Y:/Vivado/lab05/mem_data.dat", top.datamemory_module.memFile);
13          reset = 0;
14          clk = 0;
15      end
16
17      always #10 clk = ~clk;
18
19      initial begin
20          reset = 0;
21          #30;
22          reset = 1;
23          #40;
24          reset = 0;
25      end
26  endmodule
```

每隔 10 个时间单位（Time Unit）将时钟信号取反。

复位信号（RESET）先设为低电平，等待 30 个时间单位后将复位信号设为高电平。之后等待 40 个时间单位后将复位信号重新设为低电平。

用汇编语言写一个利用循环实现乘法的程序，进一步测试模拟 CPU 功能。初始化时，所有 Registers 被置为 0，memFile 第一个字是 9，第二个字是 13，结果存在第 3 个字中。预期最后结果是 117.

汇编程序：

```
1  lw $1, 0($3)
2  lw $2, 4($3)
3  srl $4, $2, 1
4  sll $5, $4, 1
5  beq $5, $2, 1
6  add $8, $8, $1
7  srl $2, $2, 1
8  sll $1, $1,1
9  beq $2, $3,1
10 j2
```

```
sw $8, 8($3)
```

其机器代码为：

```
10001100011000010000000000000000
10001100011000100000000000000100
00000000000000010001000000010000010
00000000000000100001010000010000000
00010000010001010000000000000001
00000001000000001010000000100000
00000000000000010000100000010000010
00000000000000010000100000010000000
00010000011000100000000000000001
00001000000000000000000000000010
10101100011010000000000000001000
```

初始化存储器内容为

```
00000009
0000000D
```

测试结果如图 15所示。其中，最后四行内容分别为 regFile[1]，regFile[2]，regFile[8]，memFile[2] 可以看到，所展示的结果和预期相同。说明了该 single-cycle processor 在执行实际有意义的汇编程序时的正确性与适用性。

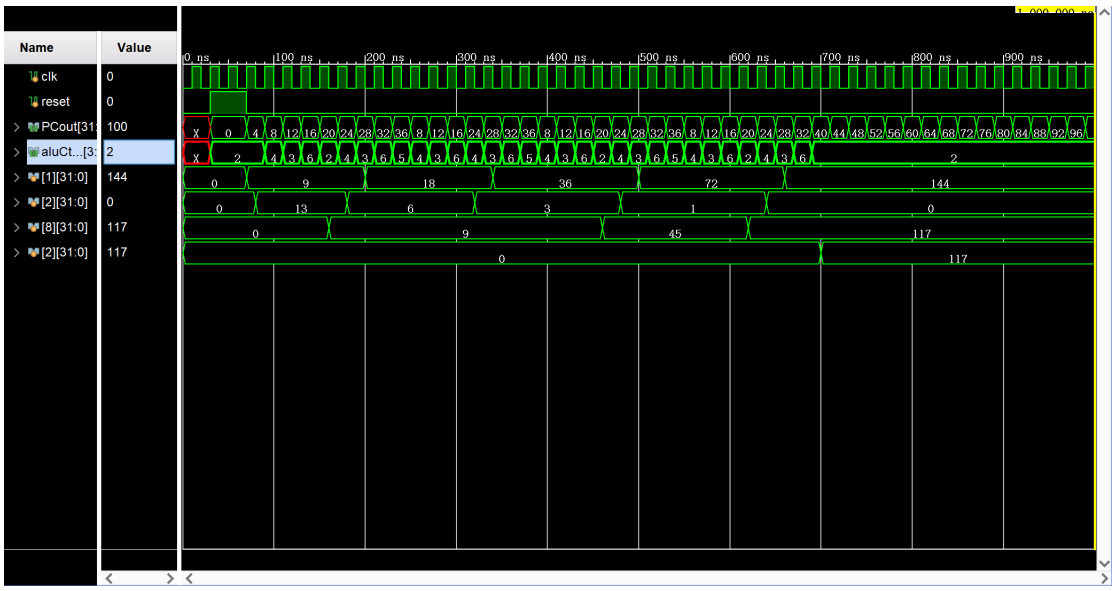

图 15: MIPS 单周期处理器测试结果

5 问题与解决

5.1 InstMem 按时序读数据错误

在一开始设计指令存储器 InstMem 时，将读指令阶段用 clk 下降沿进行约束，导致 InstMem 在输出阶段发生一个 clk 的延迟，这使得 branch 等跳转指令后 InstMem 仍然输出上一个被跳过的指令，使得结果发生错误。解决方案为将读取改为实时输出。

问题的本质在于：存储器虽然属于状态单元，但读取操作并未改变存储器状态，不应当以时钟约束。

5.2 高阻状态

在进行仿真测试时，曾发生某些信号量呈现蓝色，且显示字符“z”，这是发生高阻状态，在多次试验中，我总结了这种状态发生的可能原因：

- 信号线定义时宽度不够，例如 32 位信号线忘记加“[31:0]”
- 名称错误导致模块未识别
- 激励文件非顶层文件导致

5.3 生成仿真文件报错

在进行仿真模拟时，经常会发生报错，但报错信息有时很模糊。总结其发生的可能原因及解决为：

- 某个文件中发生语法错误
- 文件路径错误或某个变量名称错误导致无法找到
- 一个仿真已在运行（或未正常退出），可以通过任务管理器关闭其后台进程
- 仿真文件错误，可以进入项目文件夹删除.wcfg 后缀文件重新仿真

6 总结与反思

在实验 5 中，通过修改和应用之前实验搭建的模块，实现了支持 16 条指令的 MIPS 单周期处理器，并编写汇编与机器代码对构建的处理器进行仿真。该实验与计算机体系结构理论课程相辅相成，在软硬件协同的基础上加深我对计算机处理器的认识和理解，同时锻炼了自己通过更抽象的方式编写，调试代码的能力。

连接信号线的过程虽然简单，但由于信号线数量众多且命名繁杂，极易出错。我采用统一的命名范式，并且在过程中对照结构图将部分易错的信号线标注在途中，减少了错误的发生，并方便检查和调试。

基于实验 3，实验 4 构建的模块，实验 5 将之前的任务串联起来并有所发展，从根源上呈现出“分而治之”,“由易到难”的思想，使我感受到计算机工程的魅力，并对今后的专业内容起着很好的启发作用。

实验 5 的工作量和难度相较于之前的实验有很大的提升，得益于老师和助教的温馨提醒，我准备了提前量用于完成实验任务，于规定时间内较好地完成了实验。

7 致谢

刘雨桐老师为实验提供了良好的讲解，在课程中时常做出对于任务难度和时间上的提醒，并十分体谅学生，能理解我们的困难并给予帮助，深表感谢。

实验过程中，助教蔡明昕、黄正翔老师多次解决我的困惑和实验中遇到的问题，并且在困难的时候给予我鼓励和帮助，对此深表感激。

感谢黄小平老师及实验室提供的资源和硬件支持。

感谢在实验过程中给予我帮助的同学。

邓倩妮老师为使得教学效果更好，将课程体系由 RISC-V 改为 MIPS，在实验中提供很多帮助，并且本文一部分图片来自邓倩妮老师计算机体系结构课程 ppt，对此表示感谢。

8 附录：代码完整展示

该部分展示了新构建和修改后的部分模块代码。

8.1 Ctr

```
module Ctr(
    input [5:0] opCode,
    output regDst,
    output aluSrc,
    output memToReg,
    output regWrite,
    output memRead,
    output memWrite,
    output branch,
    output [2:0] aluOp,
    output jump,
    output extSign,
    output jalSign
    );

    reg RegDst;
    reg ALUSrc;
    reg MemToReg;
    reg RegWrite;
    reg MemRead;
    reg MemWrite;
    reg Branch;
    reg [2:0] ALUOp;
    reg Jump;
    reg ExtSign;
    reg JalSign;

    always @(opCode)
    begin
        case(opCode)
        6'b000000: //R type
        begin
            ALUOp = 3'b101;
            RegDst = 1;
            ALUSrc = 0;
```

```
36              MemToReg = 0;
37              RegWrite = 1;
38              MemRead = 0;
39              MemWrite = 0;
40              Branch = 0;
41              Jump = 0;
42              ExtSign = 0;
43              JalSign = 0;
44          end
45
46          //lw command
47          6'b100011:
48          begin
49              ALUOp = 3'b000;
50              RegDst = 0;
51              ALUSrc = 1;
52              MemToReg = 1;
53              RegWrite = 1;
54              MemRead = 1;
55              MemWrite = 0;
56              Branch = 0;
57              Jump = 0;
58              ExtSign = 1;
59              JalSign = 0;
60          end
61
62          //sw command
63          6'b101011:
64          begin
65              ALUOp = 3'b000;
66              RegDst = 0;
67              ALUSrc = 1;
68              MemToReg = 0;
69              RegWrite = 0;
70              MemRead = 0;
71              MemWrite = 1;
72              Branch = 0;
73              Jump = 0;
74              ExtSign = 1;
75              JalSign = 0;
76          end
77
```

```
78          //addi command
79          6'b001000:
80          begin
81              ALUOp = 3'b001;
82              RegDst = 0;
83              ALUSrc = 1;
84              MemToReg = 0;
85              RegWrite = 1;
86              MemRead = 0;
87              MemWrite = 0;
88              Branch = 0;
89              Jump = 0;
90              ExtSign = 1;
91              JalSign = 0;
92          end
93
94          //beq command
95          6'b000100:
96          begin
97              ALUOp = 3'b110;
98              RegDst = 0;
99              ALUSrc = 0;
100             MemToReg = 0;
101             RegWrite = 0;
102             MemRead = 0;
103             MemWrite = 0;
104             Branch = 1;
105             Jump = 0;
106             ExtSign = 1;
107             JalSign = 0;
108         end
109
110         //ori command
111         6'b001101:
112         begin
113             ALUOp = 3'b010;
114             RegDst = 0;
115             ALUSrc = 1;
116             MemToReg = 0;
117             RegWrite = 1;
118             MemRead = 0;
119             MemWrite = 0;
```

```
120             Branch = 0;
121             Jump = 0;
122             ExtSign = 1;
123             JalSign = 0;
124         end
125 
126         //andi command
127         6'b001100:
128         begin
129             ALUOp = 3'b011;
130             RegDst = 0;
131             ALUSrc = 1;
132             MemToReg = 0;
133             RegWrite = 1;
134             MemRead = 0;
135             MemWrite = 0;
136             Branch = 0;
137             Jump = 0;
138             ExtSign = 0;
139             JalSign = 0;
140         end
141 
142         //jump command
143         6'b000010:
144         begin
145             ALUOp = 3'b100;
146             RegDst = 0;
147             ALUSrc = 0;
148             MemToReg = 0;
149             RegWrite = 0;
150             MemRead = 0;
151             MemWrite = 0;
152             Branch = 0;
153             Jump = 1;
154             ExtSign = 0;
155             JalSign = 0;
156         end
157 
158         //jal command
159         6'b000011:
160         begin
161             ALUOp = 3'b100;
```

```
162             RegDst = 0;
163             ALUSrc = 0;
164             MemToReg = 0;
165             RegWrite = 1;
166             MemRead = 0;
167             MemWrite = 0;
168             Branch = 0;
169             Jump = 1;
170             ExtSign = 0;
171             JalSign = 1;
172         end
173 
174         default:
175         begin
176             RegDst = 0;
177             ALUSrc = 0;
178             MemToReg = 0;
179             RegWrite = 0;
180             MemRead = 0;
181             MemWrite = 0;
182             Branch = 0;
183             ALUOp = 3'b111;
184             Jump = 0;
185         end
186         endcase
187     end
188 
189     assign regDst = RegDst;
190     assign aluSrc = ALUSrc;
191     assign memToReg = MemToReg;
192     assign regWrite = RegWrite;
193     assign memRead = MemRead;
194     assign memWrite = MemWrite;
195     assign branch = Branch;
196     assign aluOp = ALUOp;
197     assign jump = Jump;
198     assign extSign = ExtSign;
199     assign jalSign = JalSign;
200 
201 endmodule
```

8.2 ALUCtr

```
module ALUCtr(
    input [2:0] aluOp,
    input [5:0] funct,al
    output [3:0] aluCtrOut,
    output shamtSign,
    output jrSign
    );

    reg [3:0] ALUCtrOut;
    reg ShamtSign;
    reg JrSign;

    always@(aluOp or funct)
    begin

        ShamtSign = 0;

        if ({aluOp, funct} == 9'b101001000) JrSign = 1;
        else JrSign = 0;

        casex({aluOp, funct})
            9'b000xxxxxx:  // lw, sw: add
            begin
                ALUCtrOut = 4'b0010;
            end

            9'b001xxxxxx:  // addi: add
            begin
                ALUCtrOut = 4'b0010;
            end

            9'b110xxxxxx:  // beq: sub
            begin
                ALUCtrOut = 4'b0110;
            end

            9'b011xxxxxx:  // andi: and
            begin
                ALUCtrOut = 4'b0000;
            end
```

```
41
42              9'b010xxxxxx:  // ori: or
43              begin
44                  ALUCtrOut = 4'b0001;
45              end
46
47              9'b101000000:  // sll
48              begin
49                  ALUCtrOut = 4'b0011;
50                  ShamtSign = 1;
51              end
52
53              9'b101000010:  // srl
54              begin
55                  ALUCtrOut = 4'b0100;
56                  ShamtSign = 1;
57              end
58
59              9'b101001000:  // jr: keep
60              begin
61                  ALUCtrOut = 4'b0101;
62              end
63
64              9'b101100000:  // add
65              begin
66                  ALUCtrOut = 4'b0010;
67              end
68
69              9'b101100010:  // sub
70              begin
71                  ALUCtrOut = 4'b0110;
72              end
73
74              9'b101100100:  // and
75              begin
76                  ALUCtrOut = 4'b0000;
77              end
78
79              9'b101100101:  // or
80              begin
81                  ALUCtrOut = 4'b0001;
82              end
```

```
83
84              9'b101101010:  // slt
85              begin
86                  ALUCtrOut = 4'b0111;
87              end
88
89              9'b100xxxxxx:  // jump: keep
90              begin
91                  ALUCtrOut = 4'b0101;
92              end
93
94              9'b100xxxxxx:  // jal: keep
95              begin
96                  ALUCtrOut = 4'b0101;
97              end
98
99          endcase
100
101     end
102
103     assign aluCtrOut = ALUCtrOut;
104     assign shamtSign = ShamtSign;
105     assign jrSign = JrSign;
106
107 endmodule
```

8.3 ALU

```
1  module ALU(
2      input [31:0] input1,
3      input [31:0] input2,
4      input [3:0] aluCtr,
5      output zero,
6      output [31:0] aluRes
7      );
8      reg Zero;
9      reg [31:0] ALURes;
10
11     always @ (input1 or input2 or aluCtr)
12     begin
```

```
13          //add
14          if(aluCtr == 4'b0010)
15              ALURes = input1 + input2;
16
17          //sub
18          else if (aluCtr == 4'b0110)
19              ALURes = input1 - input2;
20
21          //or
22          else if (aluCtr == 4'b0001)
23              ALURes = input1 | input2;
24
25          //and
26          else if (aluCtr == 4'b0000)
27              ALURes = input1 & input2;
28
29          //nor
30          else if (aluCtr == 4'b1100)
31              ALURes = ~(input1 | input2);
32
33          //slt
34          else if (aluCtr == 4'b0111)
35              ALURes = ($signed(input1) < $signed(input2));
36
37          //sll
38          else if (aluCtr ==4'b0011)
39              ALURes = input2 << input1;
40
41          //srl
42          else if (aluCtr ==4'b0100)
43              ALURes = input2 >> input1;
44
45          //keep
46          else if (aluCtr ==4'b0101)
47              ALURes = input1;
48
49          if(ALURes==0)
50              Zero = 1;
51          else
52              Zero = 0;
53      end
54
```

```
55    assign aluRes=ALURes;
56    assign zero=Zero;
57
58 endmodule
```

8.4 Top

```
1  module Top(
2          input clk,
3          input reset
4      );
5
6      wire [31:0] PCIN;
7      wire [31:0] PCOUT;
8      // PC
9      PC pc_module (
10         .PCin(PCIN),
11         .clk(clk),
12         .reset(reset),
13         .PCout(PCOUT)
14     );
15
16     wire [31:0] INST;
17     wire REGDST;
18     wire ALUSRC;
19     wire MEMTOREG;
20     wire REGWRITE;
21     wire MEMREAD;
22     wire MEMWRITE;
23     wire BRANCH;
24     wire EXTSIGN;
25     wire JALSIGN;
26     wire [2:0] ALUOP;
27     wire JUMP;
28     // Control Unit
29     Ctr ctr_module (
30         .opCode(INST[31 : 26]),
31         .regDst(REGDST),
32         .aluSrc(ALUSRC),
33         .memToReg(MEMTOREG),
```

```
34          .regWrite(REGWRITE),
35          .memRead(MEMREAD),
36          .memWrite(MEMWRITE),
37          .branch(BRANCH),
38          .aluOp(ALUOP),
39          .jump(JUMP),
40          .extSign(EXTSIGN),
41          .jalSign(JALSIGN)
42      );
43
44      wire [31:0] ALUIN1;
45      wire [31:0] ALUIN2;
46      wire [3:0] ALUCTR;
47      wire ALUZERO;
48      wire [31:0] ALURES;
49
50      // ALU
51      ALU alu_module (
52          .input1(ALUIN1),
53          .input2(ALUIN2),
54          .aluCtr(ALUCTR),
55          .zero(ALUZERO),
56          .aluRes(ALURES)
57      );
58
59      wire SHAMTSIGN;
60      wire JRSIGN;
61      // ALU Controller
62      ALUCtr aluctr_module (
63          .aluOp(ALUOP),
64          .funct(INST[5 : 0]),
65          .aluCtrOut(ALUCTR),
66          .shamtSign(SHAMTSIGN),
67          .jrSign(JRSIGN)
68      );
69
70      wire [31:0] EXTRES;
71      // EXTSIGN
72      signext signext_module (
73          .extSign(EXTSIGN),
74          .inst(INST[15 : 0]),
75          .data(EXTRES)
```

```
76      );
77
78      wire [4:0] WRITEREG;
79      wire [31:0] REGOUT1;
80      wire [31:0] REGOUT2;
81      wire [31:0] REG_WRITE_DATA;
82      // register
83      Register register_module (
84          .readReg1(INST[25 : 21]),
85          .readReg2(INST[20 : 16]),
86          .writeReg(WRITEREG),
87          .writeData(REG_WRITE_DATA),
88          .regWrite(REGWRITE & (~JRSIGN)),
89          .clk(clk),
90          .reset(reset),
91          .readData1(REGOUT1),
92          .readData2(REGOUT2)
93      );
94
95      // instruction memory
96      InstMem instmem_module (
97          .readAddr(PCOUT),
98          .clk(clk),
99          .reset(reset),
100         .instr(INST)
101     );
102
103     wire [31:0] MEM_READ_DATA;
104     // data memory
105     dataMemory datamemory_module (
106         .clk(clk),
107         .address(ALURES),
108         .writeData(REGOUT2),
109         .memWrite(MEMWRITE),
110         .memRead(MEMREAD),
111         .readData(MEM_READ_DATA)
112     );
113
114     // PC update
115     PCupdate pcupdate_module (
116         .newPC(PCOUT+4),
117         .target(INST[25 : 0]),
```

```
118         .offset(EXTRES << 2),
119         .regOut(REGOUT1),
120         .Branch(BRANCH),
121         .Aluzero(ALUZERO),
122         .Jump(JUMP),
123         .JrSign(JRSIGN),
124         .nextPC(PCIN)
125     );
126 
127 ////////////////////////////////////////////////
128     wire [31 : 0] REG_WRITE_DATA_T;
129     wire [4 : 0] WRITE_REG_TEMP;
130 
131     // INST[10:6] or rs
132     MUX rs_shamt_selector (
133         .Select(SHAMTSIGN),
134         .data0({27'b000000000000000000000000000, INST[10 : 6]}),
135         .data1(REGOUT1),
136         .data(ALUIN1)
137     );
138 
139     // EXTRES or rt
140     MUX rt_ext_selector (
141         .Select(ALUSRC),
142         .data0(EXTRES),
143         .data1(REGOUT2),
144         .data(ALUIN2)
145     );
146 
147     // MEM READ DATA or ALURES
148     MUX mem_alu_selector (
149         .Select(MEMTOREG),
150         .data0(MEM_READ_DATA),
151         .data1(ALURES),
152         .data(REG_WRITE_DATA_T)
153     );
154 
155     // PCOUT + 4 or REG WRITE DATA T
156     MUX jal_selector (
157         .Select(JALSIGN),
158         .data0(PCOUT + 4),
159         .data1(REG_WRITE_DATA_T),
```

```
160         .data(REG_WRITE_DATA)
161     );
162 
163     // rd or rt
164     MUX_5 rt_rd_selector (
165         .Select(REGDST),
166         .data0(INST[15 : 11]),
167         .data1(INST[20 : 16]),
168         .data(WRITE_REG_TEMP)
169     );
170 
171     // 11111 or rt/td
172     MUX_5 rtrd_31_selector (
173         .Select(JALSIGN),
174         .data0(5'b11111),
175         .data1(WRITE_REG_TEMP),
176         .data(WRITEREG)
177     );
178 
179 endmodule
```