#计组实验#单周期CPU设计

原创

LoHiauFung 于 2017-04-24 00:35:18 发布 6746 收藏 49

分类专栏： 计组实验

本文链接：https://blog.csdn.net/LoHiauFung/article/details/70562564

一.实验目的

1.掌握单周期CPU数据通路图的构成、原理及其设计方法

2.掌握单周期CPU的实现方法，代码实现方法

3.认识和掌握指令与CPU的关系

4.掌握测试单周期CPU的方法

二.实验内容

设计一个单周期CPU，该CPU至少能实现以下指令功能操作。需设计的指令与格式如下：

（1）**add rd , rs, rt** （**说明：以助记符表示，是汇编指令；以代码表示，是机器指令**）

000000	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs + rt。**reserved为预留部分，即未用，一般填“0”。**

（2）addi rt , rs ,**immediate**

000001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs + (sign-extend)**immediate; immediate**符号扩展再参加“加”运算。

（3）sub rd , rs , rt

000010	rs(5 位)	rt(5 位)	rd(5 位)	reserved

完成功能：rd←rs - rt

⟹ 逻辑运算指令

（4）ori rt , rs ,**immediate**

010000	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs | (zero-extend)**immediate; immediate**做“0”扩展再参加“或”运算。

（5）and rd , rs , rt

010001	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs & rt；逻辑与运算。

（6）or rd , rs , rt

010010	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs | rt；逻辑或运算。

=>移位指令

（7）sll rd, rt, sa

011000	未用	rt(5 位)	rd(5 位)	sa	reserved

功能：rd<-rt<<(zero-extend)sa，左移 sa 位 ，(zero-extend)sa

=> 存储器读/写指令

（8）sw rt ,**immediate**(rs) 写存储器

100110	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：memory[rs+ (sign-extend)**immediate**]←rt；**immediate**符号扩展再相加。

（9）lw rt , **immediate**(rs) 读存储器

100111	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt ← memory[rs + (sign-extend)**immediate**]； **immediate**符号扩展再相加。

三.实验原理

单周期CPU指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。时钟周期一般也称振荡周期。CPU在处理指令时，一般需要经过以下几个步骤：

(1) 取指令(IF)：根据程序计数器PC中的指令地址，从存储器中取出一条指令，同时，PC根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把“转移地址”送入PC，当然得到的“地址”需要做些变换才送入PC。
(2) 指令译码(ID)：对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。
(3) 指令执行(EXE)：根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。
(4) 存储器访问(MEM)：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。
(5) 结果写回(WB)：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。
单周期CPU，是在一个时钟周期内完成这五个阶段的处理。

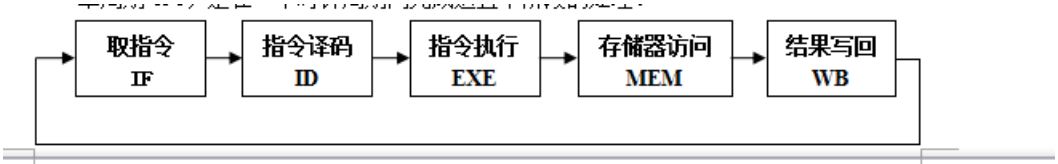


图1 单周期CPU指令处理过程

MIPS 指令的三种格式：

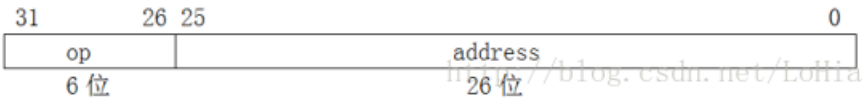
R 类型：

31 26	25 21	20 16	15 11	10 6	5 0
op	rs	rt	rd	sa	funct
6 位	5 位	5 位	5 位	5 位	6 位

I 类型：

31 26	25 21	20 16	15 0
op	rs	rt	immediate
6 位	5 位	5 位	16 位

J 类型：



其中，

op: 为操作码；

rs: 为第 1 个源操作数寄存器，寄存器地址（编号）是 00000~11111，00~1F；

rt: 为第 2 个源操作数寄存器，或目的操作数寄存器，寄存器地址（同上）；

rd: 为目的操作数寄存器，寄存器地址（同上）；

sa: 为位移量（shift amt），移位指令用于指定移多少位；

func: 为功能码，在寄存器类型指令中（R 类型）用来指定指令的功能；

immediate: 为 16 位立即数，用作无符号的逻辑操作数、有符号的算术操作数、数据加载(Laod)/数据保存(Store)指令的数据地址字节偏移量和分支指令中相对程序计数器(PC)的有符号偏移量；

address: 为地址。

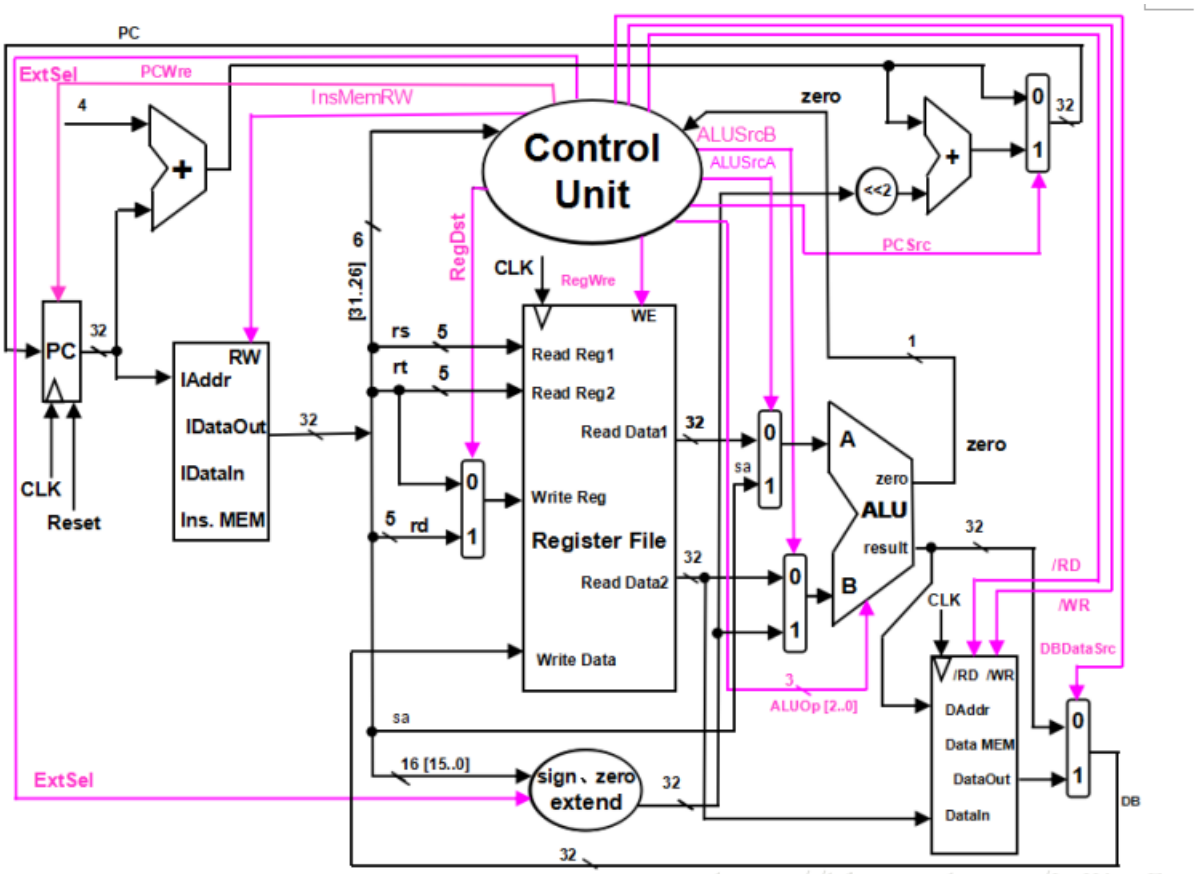


图 2　单周期 CPU 数据通路和控制线路图

图2是一个简单的基本上能够在单周期CPU上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储在不同存储器中，即有指令存储器和数据存储器。访问存储器时，先给出地址，然后由读或写信号控制操作。对于寄存器组，读操作时，先给出地址，输出端就直接输出相应数据；而在写操作时，在 WE使能信号为1时，在时钟边沿触发写入。图中控制信号作用如表1所示，表2是ALU运算功能表。

表 1 控制信号的作用

控制信号名	状态“0”	状态“1”
Reset	初始化 PC 为 0	PC 接收新地址
PCWre	PC 不更改，相关指令：halt	PC 更改，相关指令：除指令 halt 外
ALUSrcB	来自寄存器堆 data2 输出，相关指令：addu、subu、or、and、bne	来自 sign 或 zero 扩展的立即数，相关指令：addiu、ori、sw、lw

DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addi、sub、ori、or、and、sll	来自数据存储器（Data MEM）的输出，相关指令：lw
RegWre	无写寄存器组寄存器，相关指令：beq、sw、halt	寄存器组写使能，相关指令：add、addi、sub、ori、or、and、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
/RD	读数据存储器，相关指令：lw	输出高阻态
/WR	写数据存储器，相关指令：sw	无操作
ExtSel	(zero-extend)**immediate**（0 扩展）相关指令：ori，	(sign-extend)**immediate**（符号扩展）相关指令：addi、sw、lw、beq，

RegDst	写寄存器组寄存器的地址，来自 rt 字段，相关指令：addi、ori、lw	写寄存器组寄存器的地址，来自 rd 字段，相关指令：add、sub、and、or、sll
PCSrc	PC←PC+4，相关指令：add、addi、sub、ori、or、and、sw、sll、lw、beq(zero=0)	PC←PC+4+(sign-extend)**immediate**，相关指令：beq（zero=1）
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，看功能表	

相关部件及引脚说明：

Instruction Memory：指令存储器，

Iaddr，指令存储器地址输入端口

IDataIn，指令存储器数据输入端口（指令代码输入端口）

IDataOut，指令存储器数据输出端口（指令代码输出端口）

RW，指令存储器读写控制信号，为 1 写，为 0 读

Data Memory：数据存储器，

Daddr，数据存储器地址输入端口

DataIn，数据存储器数据输入端口

DataOut，数据存储器数据输出端口

RD，数据存储器读控制信号，为 1 读

WR，数据存储器写控制信号，为 1 写

Register File：寄存器组

Read Reg1，rs 寄存器地址输入端口

Read Reg2，rt 寄存器地址输入端口

Write Reg，将数据写入的寄存器端口，其地址来源 rt 或 rd 字段

Write Data，写入寄存器的数据输入端口

Read Data1，rs 寄存器数据输出端口

Read Data2，rt 寄存器数据输出端口

WE，写使能信号，为 1 时，在时钟上升沿写入

ALU： 算术**逻辑单元**

result，ALU 运算结果

zero，运算结果标志，结果为 0 输出 1，否则输出 0

表 2 ALU运算功能表

ALUOp[3..0]	功能	描述
0000	Y = A + B	加
0001	Y = A - B	减
0010	Y=（A=B）?1: 0	比较A与B是否相等 不带符号
0011	Y = B<<A	B左移A位
0100	Y = A ∧ B	与
0101	Y = /A ∧ B	A非与B
0110	Y = A ⊕ B	异或
0111	Y = A ⊙ B	同或
1000	Y = A ∨ B	或

四. 实验器材

电脑一台、Xilinx Vivado 软件一套。

五. 实验分析与设计

本CPU所包含模块如表三所示，有一些不在通路图的是自己所加。

表3 CPU模块表

名称	功能	实例名称
InstructionMemory	指令存储器	insMem
ALU32	逻辑算数单元	alu
CU	控制单元	cu
DataMemory	用于存储数据，相当于内存	dataMem
PC	指令计数器	pc
Regfile	寄存器组	regfile
extend	扩展器，16位输入32位输出，支持零扩展和符号扩展	ext
selector	2选1选择器，输入输出都是32位	ALUscA、ALUscB、DBsc、NextPCsc
selectorFor5bit	2选1选择器，输入输出都是32位	WRRegFileSC
decoder	解码器，输入指令，输出指令的immediate，rt，rs，rd，sa。	dec
singleCycleCPU	顶层模块，用于连接各底层模块	uut
CPUrunner	测试模块，用于顶层模块的	

	启动和测试	

1.InstructionMemory

该模块实现了根据RW信号读laddr地址的指令的功能，写指令功能因为本实验没有用到，注释掉了。在初始化的时候，本模块先用系统调用$readmemb装载指令。

```
module InstructionMemory(
    input [31:0] Iaddr, // 指令存储器地址输入端口
    // input [31:0] IDataIn,  // 指令存储器数据输入端口（指令代码输入端口）
    input RW,       // 指令存储器读写控制信号，为1写，为0读
    output reg[31:0] IDataOut // 指令存储器数据输出端口（指令代码输出端口）
    );

    reg[7:0] storage [127:0];

     always @(RW or Iaddr ) begin
           if(RW == 1) begin //write
            /* 本次实验不需要用到写指令功能
               storage[Iaddr] <= IDataIn[7:0];
               storage[Iaddr + 1] <= IDataIn[15:8];
               storage[Iaddr + 2] <= IDataIn[23:16];
               storage[Iaddr + 3] <= IDataIn[31:24];
            */
               end
           else begin // read
               IDataOut[7:0] <= storage[Iaddr + 3];
               IDataOut[15:8] <= storage[Iaddr + 2];
               IDataOut[23:16] <= storage[Iaddr + 1];
               IDataOut[31:24] <= storage[Iaddr];
                end
      end

      initial begin
           $readmemb("F:/ECOP_Experiment/CPU/CPU.srcs/sources_1/new/ins.txt",storage);
      end


endmodule
```

2.ALU32

ALU实现不复杂，根据表2，对每个opCode实现对应的运算，输出便可。

```
module ALU32(
    input [3:0] ALUopcode,
    input [31:0] rega,
    input [31:0] regb,
    output reg [31:0] result,
    output zero
    );

   assign zero = (result==0)?1:0;
   always @( ALUopcode or rega or regb ) begin
      case (ALUopcode)
         4'b0000 : result = rega + regb;
         4'b0001 : result = rega - regb;
         4'b0010 : result = (rega == regb)?1:0;
         4'b0011 : result = regb << rega;
         4'b0100 : result = rega & regb;
         4'b0101 : result = (!rega) & regb;
         4'b0110 : result = rega ^ regb;
         4'b0111 : result = rega ^~ regb
         4'b1000 : result = rega | regb;
         // 4'b1001 : result = (rega < regb)?1:0;
      default : begin
         result = 32'h00000000;
         $display (" no match");
        end
     endcase
   end
endmodule
```

3.CU

CU的设计较为复杂，实现的功能主要是根据输入的操作码opCode和ALU的zero信号来控制各个信号的输出。根据表1和表2，我们可以得出表3的ALU输出信号真值表。根据该表，实现每种输入对应的使出便可，如图6、图7、图8所示。

表3 CU的输出信号真值表

输入		输出												
op指令	zero	Reset	PCWre	ALUSrcB	DBDataSrc	RegWre	InsMemRW	/RD	/WR	ExtSel	RegDst	PCSrc	ALUop[3,0]	ALUsrcA
add	x	1	1	0	0	1	1	0	0	x	1	0	0000	0
addi	x	1	1	1	0	1	1	0	0	1	0	0	0000	0
sub	x	1	1	0	0	1	1	0	0	x	1	0	0001	0
ori	x	1	1	1	0	1	1	0	0	0	0	0	1000	0
and	x	1	1	0	0	1	1	0	0	x	1	0	0100	0
or	x	1	1	0	0	1	1	0	0	x	1	0	1000	0
sll	x	1	1	0	0	1	1	0	0	x	1	0	0011	1
sw	x	1	1	1	x	0	1	0	1	1	x	0	x	0
lw	x	1	1	1	1	1	1	1	0	1	0	0	x	0
beq	1	0	1	x	0	0	1	0	0	1	x	1	0010	0
	0	0	1	x	0	0	1	0	0	1	x	0	0010	0
halt	x	x	0	x	x	0	1	0	0	x	x	x	x	x


```
module CU(
    // input
    input [5:0] opCode,
    input zero,
    // output
    output RegDst,
    output InsMemRW,
    output PCWre,
    output ExtSel,
    output DBDataSrc,
    output WR,
    output RD,
    output ALUSrcB,
    output ALUSrcA,
    output PCSrc,
    output reg[3:0] ALUop,
    output RegWre
    );

     // assign Reset=(opCode == 6'b110000)? 0 : 1;
    assign RegDst=(opCode == 6'b000000 || opCode == 6'b000010 || opCode == 6'b010001 || opCode == 6'b01
    assign PCWre=(opCode == 6'b111111)? 0 : 1;
    assign ALUSrcB=(opCode == 6'b000001 || opCode==6'b010000 || opCode == 6'b100110 || opCode == 6'b100
```

```
    assign DBDataSrc=(opCode == 6'b100111)? 1 : 0;
    assign InsMemRw = 1;
    assign RD=(opCode == 6'b100111)? 1 : 0;
    assign WR=(opCode == 6'b100110)? 1 : 0;
    assign ExtSel=(opCode == 6'b010000 || opCode == 6'b010000 || opCode == 6'b100111)? 0 : 1;
    assign PCSrc=(zero == 0 && opCode == 6'b110000)? 1:0;
    assign ALUSrcA =(opCode == 6'b011000)?1: 0;
    assign RegWre=(opCode==6'b100110 || opCode==6'b110000 || opCode==6'b111111 )? 0 : 1;

    always @(opCode) begin
        case(opCode)
            // add
            6'b000000 : ALUop = 4'b0000;
            // addi
            6'b000001 : ALUop = 4'b0000;
            // sub
            6'b000010 :ALUop = 4'b0001;
            // ori
            6'b010000 :ALUop = 4'b1000;
            // and
            6'b010001 :ALUop = 4'b0100;
            // or
            6'b010010 :ALUop = 4'b1000;
            // sll
            6'b011000 :ALUop = 4'b0011;
            // sw
            6'b100110 : begin end
            // lw
            6'b100111 : begin end
            // beq
            6'b110000 : ALUop = 4'b0010;
            // halt
            6'b111111 : begin end
          default: begin
            $display (" no match");
           end
       endcase
    end
endmodule
```

4.DataMemory

DataMemory主要实验数据的读写功能，如图9所示。

```
module DataMemory(
    // input
    input [31:0] Daddr,
    input [31:0] DataIn,
    input CLK,
    input WR,
    input RD,
    // output,
    output reg[31:0] DataOut
    );
reg[7:0] storage [3:0];

// RD为真的时候读数
always @(Daddr or WR or RD) begin
    if (RD == 1) begin
        DataOut[7:0] <= storage[Daddr];
        DataOut[15:8] <= storage[Daddr+1];
        DataOut[23:16] <= storage[Daddr+2];
        DataOut[31:24] <= storage[Daddr+3];
    end else begin
        DataOut = 0;
    end
end

// 下降沿写数
always @(negedge CLK) begin
    if(WR == 1) begin
        storage[Daddr] <= DataIn[7:0];
        storage[Daddr+1] <= DataIn[15:8];
        storage[Daddr+2] <= DataIn[23:16];
        storage[Daddr+3] <= DataIn[31:24];
    end
end
endmodule
```

5.PC

PC输出当前指令执行的条数，并可接受下一条执行指令的行数作为下一时钟周期的输出。

```
module PC(
    // input
    input clk,
    input reset,
    input PCWre,
    input [31:0] nextPC,
    // output
    output reg[31:0] curPC
    );

    // 上升沿触发
    always @(posedge clk or negedge reset) begin
        if(reset == 0) begin
            curPC <= 0; // Reset,指令地址初值为0
        end else if(PCWre) begin
            curPC <= nextPC; // 下一指令地址
        end

        //在控制台输出，便于debug
        $display("curPC: ", curPC, "nextPC:", nextPC);

    end


endmodule
```

6.Regfile

Regfile是寄存器组，有读写指定寄存器的功能。

```
module Regfile(
    input CLK,
    input Wre,
    // input CLR,
    input [4:0] RdReg1addr,
    input [4:0] RdReg2addr,
    input [4:0] WrRegaddr,
    input [31:0] indata,
    output [31:0] reg1dataOut,
    output [31:0] reg2dataOut
    );
    reg     [31:0]  register [31:0];             // 寄存器宽度32位，共31个，R1 -Rr31
   integer  i;                                  // 变量

    assign reg1dataOut = (RdReg1addr== 0)? 0 : register[RdReg1addr];   // 读寄存器1数据
    assign reg2dataOut = (RdReg2addr== 0)? 0 : register[RdReg2addr];   // 读寄存器2数据

    // psedge?
    always @(negedge CLK ) begin
        if ((WrRegaddr != 0) && (Wre == 1))     // $0寄存器不能修改
          register[WrRegaddr] <= indata;        // 写寄存器，非阻塞赋值"<="
    end

        initial begin
            for (i=1; i<32; i=i+1) begin
                    register[i] <= 0;                    // 寄存器清0，非阻塞赋值"<="
            end
        end

endmodule
```

7.Extend

Extend用于将16位数扩展至32位。

```
module extend(
    // input
    input [15:0] extendee,
    input ExtSel,
    // output
    output reg[31:0] result
    );

    integer i;
    // assign result[15:0] = extendee[15:0];
    // assign result[32:16] = 0;

    always @(ExtSel or extendee) begin
            for(i = 0; i < 16; i = i + 1) begin
                result[i] <= extendee[i];
            end
            // 有符号数
            if(ExtSel) begin
                for(i = 16; i < 32; i = i + 1) begin
                    result[i] = extendee[15];
                end
            end else begin
                for(i = 16; i < 32; i = i + 1) begin
                    result[i] = 0;
                end
            end
    end

endmodule
```

8.Selector

Selector是2选1选择器。当selectBit为0，输出inputA,否则输出inputB。

```
module selector(
    // input
    input selectBit,
    input [31:0] inputA, inputB,
    // output
    output reg[31:0] selectorResult
    );

    always @(selectBit or inputA or inputB) begin
        if (0 == selectBit) begin
            selectorResult <= inputA;
        end else begin
            selectorResult <= inputB;
        end
    end
endmodule
```

9.selectorFor5bit

selectorFor5bit功能与selector类似。

```
module selectorFor5bit(
 // input
   input selectBit,
   input [4:0] inputA, inputB,
   // output
   output reg[4:0] selectorResult
    );

    always @(selectBit or inputA or inputB) begin
            if (0 == selectBit) begin
                selectorResult = inputA;
            end else begin
                selectorResult = inputB;
            end
        end
endmodule
```

10.Decoder

Decoder用于从指令中抽取rs,rt,rd,sa,immediate,ext_sa并输出，这个模块是为了方便顶层模块的各模块的连接。

```
module decoder(
    // input
    input [31:0] ins,
    // output
    output reg[5:0] opCode,
    output reg[4:0] rs,
    output reg[4:0] rt,
    output reg[4:0] rd,
    output reg[4:0] sa,
    output reg[15:0] immediate,
    output reg[31:0] ext_sa
    );

    always @(ins) begin
        opCode=ins[31:26];
        rs=ins[25:21];
        rt=ins[20:16];
        rd=ins[15:11];
        sa=ins[10:6];
        immediate=ins[15:0];
        ext_sa={ 27'b000000000000000000000000000 ,ins[10:6]};
    end
endmodule
```

11.singleCycleCPU

singleCycleCPU是顶层模块，主要是实例化各个模块，并连接对应端口。可以理解为将一个CPU的内部原件用导线连起来。

```
// 很多输出变量，为了Debug的时候，看波形图方便
module singleCycleCPU(
    input clk,
    input reset,
    output PCWre,
    output RegWre,
    output ExtSel,
    output DBDataSrc,
    output [31:0] nextPC, curPC, instcode,
    output [3:0] ALUop,
    output [31:0] alu_inputA,
    output [31:0] alu_inputB,
    output [31:0] alu_out,
    output [5:0] opCode,
    output [4:0] rs,
    output [4:0] rt,
    output [4:0] rd,
    output [4:0] sa,
    output [15:0] immediate,
    output [31:0] ext_sa,
    output [4:0] /*readReg1addr, readReg2addr,*/ wrRegaddr,
    output [31:0] readReg1out, readReg2out, wrRegdata,
    output zero,
    output [31:0] mem_out,
    output [31:0] ext_immediate,
    output RegDst, ALUSrcA,ALUSrcB,DataMemRW, InstMemRW, Extsel,WR, RD, PCSrc
    );


    // InstructionMemory(Iaddr, input RW,output reg[31:0] IDataOut);
    InstructionMemory insMem(curPC, InstMemRW,instcode);
    // module ALU32(input [3:0] ALUopcode,input [31:0] rega,input [31:0] regb,output reg [31:0] result,
    ALU32 alu (ALUop, alu_inputA, alu_inputB, alu_out, zero);
    // Control Units
    CU cu(opCode, zero, RegDst, InstMemRW, PCWre, ExtSel, DBDataSrc, WR, RD, ALUSrcB, ALUSrcA, PCSrc, A
    // module DataMemory(input [31:0] Daddr,input [31:0] DataIn,input CLK,input WR,input RD,,output reg
    DataMemory dataMem(alu_out,  readReg2out, clk, WR, RD, mem_out);
    // module PC(input clk,input reset,input PCWre,input [31:0] nextPC,output reg[31:0] curPC);
    PC pc(clk, reset, PCWre, nextPC, curPC);
    // regFile( CLK,Wre,[4:0] RdReg1addr, [4:0] RdReg2addr, [4:0] WrRegaddr, [31:0] Indata, [31:0] reg1
    Regfile regfile(clk, RegWre,rs, rt, wrRegaddr, wrRegdata, readReg1out, readReg2out);
    //  extend(input [15:0] extendee,input ExtSel,input SignOrZero, output reg[31:0] result);
    extend ext(immediate,ExtSel, ext_immediate);
    // selector(input selectBit,input [31:0] inputA, inputB,output reg[31:0] selectorResult);
    selector ALUscA(ALUSrcA,  readReg1out, ext_sa, alu_inputA);
    selector ALUscB(ALUSrcB, readReg2out, ext_immediate,alu_inputB);
    selector DBsc(DBDataSrc, alu_out, mem_out, wrRegdata);
    selector NextPCsc(PCSrc, curPC +  4, curPC + 4 + (ext_immediate << 2), nextPC);
    selectorFor5bit WRRegFileSC(RegDst, rt, rd,wrRegaddr);
    // decoder(ins, opCode, rs, rt, rd, sa, immediata,output reg[31:0] ext_sa
    decoder dec(instcode, opCode, rs, rt, rd,sa, immediate, ext_sa);

endmodule
```

12.CPUrunner

CPUrunner是本实验的测试模块。主要是声明变量，实例化一个singleCycleCPU，再设置clk和reset信号，整个CPU便开始运行。

```
// 一堆输出变量，也是为了方便Debug
module CPUrunner;
    reg  clk;
    reg  reset;
    wire PCWre;
    wire RegWre;
    wire ExtSel;
    wire DBDataSrc;
    wire [31:0] nextPC;
    wire [31:0] curPC;
    wire [3:0] ALUop;
    wire [31:0] alu_inputA;
    wire [31:0] alu_inputB;
    wire [31:0] alu_out;
    wire zero;
    wire [31:0] instcode;
    wire [5:0] opCode;
    wire [4:0] rs;
    wire [4:0] rt;
    wire [4:0] rd;
    wire [4:0] sa;
    wire [15:0] immediate;
    wire [31:0] ext_sa;
    wire [4:0] wrRegaddr;
    wire [31:0] readReg1out;
    wire [31:0] readReg2out;
    wire [31:0] wrRegdata;
    wire [31:0] mem_out;
    wire [31:0] ext_immediate;
    wire [31:0] ext_immediate_shift;
    wire RegDst;
    wire ALUSrcA;
    wire ALUSrcB;
    wire ALUM2Reg;
    wire DataMemRW;
    wire InstMemRW;
    wire Extsel;
    wire WR;
    wire RD;
    wire PCSrc;

    singleCycleCPU uut(
    .clk(clk),
    .reset(reset),
    .PCWre(PCWre),
    .RegWre(RegWre),
    .ExtSel(ExtSel),
    .DBDataSrc(DBDataSrc),
    .nextPC(nextPC),
    .curPC(curPC),
    .ALUop(ALUop),
    .alu_inputA(alu_inputA),
    .alu_inputB(alu_inputB),
    .alu_out(alu_out),
    .zero(zero),
    .instcode(instcode),
    .opCode(opCode),
    .rs(rs),
    .rt(rt),
```

```
        .rd(rd),
        .sa(sa),
        .immediate(immediate),
        .ext_sa(ext_sa),
        .wrRegaddr(wrRegaddr),
        .readReg1out(readReg1out),
        .readReg2out(readReg2out),
        .wrRegdata(wrRegdata),

        .mem_out(mem_out),
        .ext_immediate(ext_immediate),
        .RegDst(RegDst),
        .ALUSrcA(ALUSrcA),
        .ALUSrcB(ALUSrcB),
        .DataMemRW(DataMemRW),
        .InstMemRW(InstMemRW),
        .Extsel(Extsel),
        .WR(WR),
        .RD(RD),
        .PCSrc(PCSrc)
        );

        always #10 clk = ~clk;
        initial begin
            // initialize
            reset = 0;
            clk = 0;
            #100 reset=1;
        end
    endmodule
```

六.实验结果及分析

测试代码如表4所示。

表4 测试代码

地址	汇编程序	指令代码				
		op（6）	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x00000000	addi $1,$0,2	000001	00000	00001	0000000000000010	= 04010002
0x00000004	ori $2,$1,1	010000	00001	00010	0000000000000001	= 40220001
0x00000008	sub $1,$1,$2	000010	00001	00010	0000100000000000	= 08220500
0x0000000C	and $1,$1,$0	010001	00001	00000	0000100000000000	= 44200800
0x00000010	beq $1,$0,1(转 018)	110000	00001	00000	0000000000000001	= C0200001
0x00000014	addi $1,$1,1	000001	00001	00001	0000000000000001	= 04210001
0x00000018	or $1,$1,$0	010010	00001	00000	0000100000000000	= 48200800
0x0000001C	beq $1,$2,1(转 020)	110000	00001	00010	0000000000000001	= C0220001
0x00000020	add $1, $1, $2	000000	00001	00010	0000100000000000	= 00220800
0x00000024	Sll $3, $1, 2	011000	00000	00001	0001100010000000	= 60011880
0x00000028	sw $3, 0($0)	100110	00000	00011	0000000000000000	= 98030000
0x0000002C	lw $4, 0($0)	100111	00000	00100	0000000000000000	= 9C040000
0x00000030	halt	111111	00000	00000	0000000000000000	= FC000000

这是我的测试代码。可以参考。
然后测试结果什么的，分析什么的，请读者自己跑，自己debug吧:）
毕竟结果分析这些还是要自己写的。

七.实验心得

其实就是debug方法

很多指令的执行都会遇到问题。解决的办法就是看着波形图，找出错的信号，然后根据数据通路往回推，一直推到最早出现问题的信号，这就是问题的根源。

2,1,1,机器码是0100000000000010

0
0
1
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
1,
但
是
发
现
A
L
U
输
出
的
是
0
x
0
0
0
0
0
0
0
1
而
不
是
0
x
0
0
0
0
0
0
0

u
0
3
（
为
方
便
书
写
，
此
处
用
1
6
进
制
表
示
，
此
时
的
波
形
图
如
图
2
8
）
，
于
是
开
始
查
A
L
U
的
输
入
，
看
了
o
p
C
o
d

e,
A
L
U
S
rc
A,
A
L
U
S
rc
B,
A
L
Ui
n
p
ut
B
等
信
号
都
没
有
问
题
，
但
是
发
现
A
L
Ui
n
p
ut
A
是
0
x
0
0
0
0
0
0
0
0

0，而不是0x00000002。而此时的ALUSrcA=0，所以ALUinputA的信号来自RegisterFile的

的ReadData1。这说明我们读取失败，或者第一条指令addi1,0,2根本就没把结果写入$1。查看了Regi

st
er
Fi
le
代
码
后
，
感
觉
逻
辑
没
有
错
，
应
该
是
第
一
条
指
令
没
有
写
入
。
查
看
第
一
条
指
令
执
行
时
设
计
到
的
各
种
信
号
，
发
现

RegDst（用于选择写入RegFile的数据的来源）的信号为高阻态Z，而非0(此时波形图如图29)，将ALU

运算结果写入RegFile。因此问题发现了：第一条指令执行时RegDst为Z，因此ALU运算结果写入失败。继

续查找问题根源，RegDst信号理应是ControlUnit产生的，但是查看ControlUnit的代码，发现并没有写

RegDst的产生代码。查看控制信号与opCode的关系后，找到RegDst真的条件，在ControlUnit中

补充以下代码

以我的第二条测试指令为例，ori：

```
assign RegDst=(opCode == 6'b000000 || opCode == 6'b000010 || opCode == 6'b010001 || opCode == 6'b010010
```

再运行，发现问题解决（波形图如图30）。

限于篇幅，其他问题就不列举了，但是解决方法是类似的。

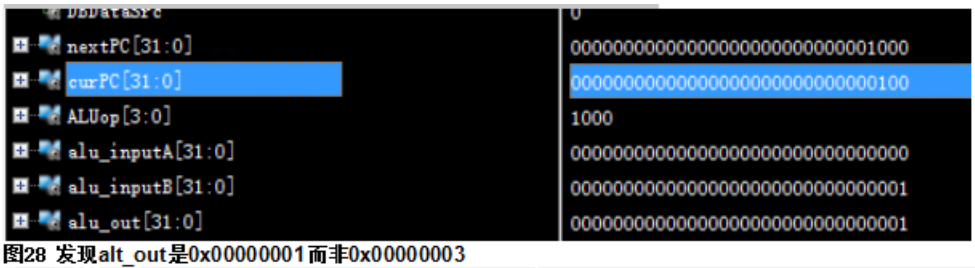


图28 发现alt_out是0x00000001而非0x00000003

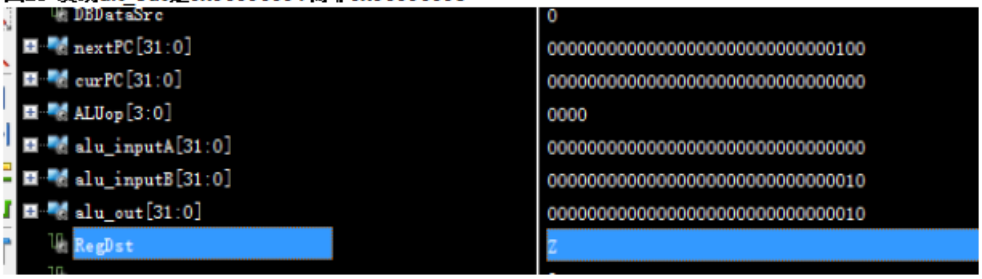


图29 发现RegDst为高阻态Z

图30 添加RegDst产生代码后，alu_out变为0x00000003,问题解决

注：

从“一、实验目的”到“四、实验器材”的内容来源于中山大学何朝东老师下发的参考文档。