计算机组成原理实验报告

LAB05

题目：单周期CPU

姓名：林祥

学号：PB16020923

实验目的

1、 综合前几次实验的结果，实现一个单周期 MIPS 指令集的 CPU， 然后运行一个计算斐波那契数列的程序。

实验平台

ISE 14.7

实验过程（分析）

1、 模块化设计，主要有以下几个模块

a) alu 模块——算术逻辑单元

b) regfile 模块——寄存器文件

c) mux 模块——选择器

d) IP 核生成的 IMem 模块和 DMem 模块——存放数据段和代码段

e) nextpclogic 模块——计算下一个 PC

f) control 模块——控制 regfile、IMem、DMem、nextpclogic 和 alu

g) top 模块——实例化前几个模块，连接各个信号

2、alu 模块使用 case 语句判断 8 种操作类型。

3、regfile 模块用组合逻辑读，时序逻辑写。

4、IMem 和 DMem 是异步读，同步写，且由指定 coe 文件初始化，coe 文件的内容是 16 进制文本，由 Mars 编译一个汇编代码生成。

5、nextpclogic 模块通过组合逻辑计算出 nextPC 的值。

6、 control 模块根据输入的操作符 Op 对各控制变量进行赋值，根据 ALUOp 对 ALUControl 进行赋值。

7、 top 模块实例化前几个模块，连接各个信号。

8、bgtz 的实现：bgtz 是伪指令（类似地有 la 和 li），不是 MIPS 指令集的指令，编译时会进行处理，转换为几条指令实现。

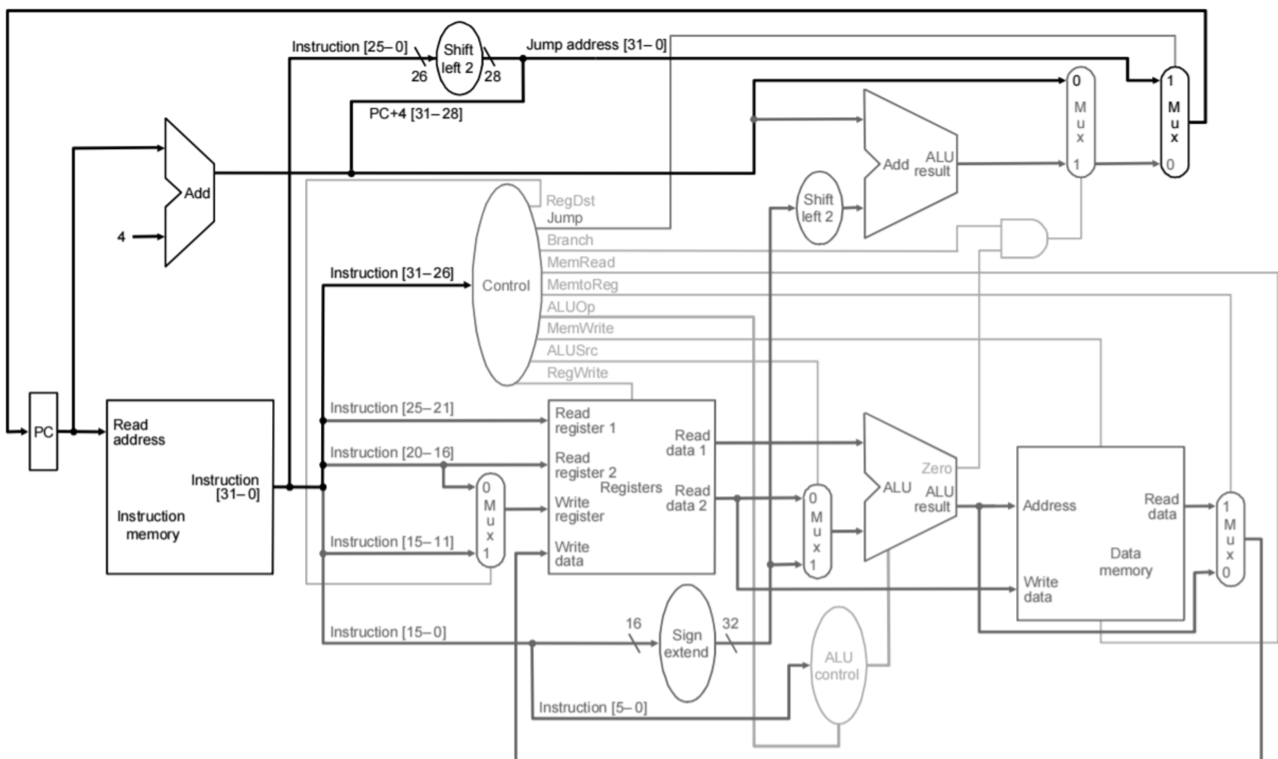
这里我用以下三条指令

```
slt $t6,$zero,$t1     # if $t1 > 0, that $t6 = 1, else $t6 = 0
addi $t7,$zero,1      # set $t7 = 1
beq $t6,$t7,loop      # jump if $t6 = 1, that is to say, $t1 > 0
```

实现

```
bgtz $t1, loop        # repeat if not finished yet.
```

9、完整的数据通路



10、分析结果

```
.data
fibs: .word   0 : 20        # "array" of  20 words to contain fib values
size: .word  20             # size of "array"
temp: .word 3 3
.text
     la   $t0, fibs         # load address of array
     la   $t5, size         # load address of size variable
     lw   $t5, 0($t5)       # load array size
     la   $t3, temp         # load
     lw   $t3, 0($t3)
     la   $t4, temp
     lw   $t4, 4($t4)

     sw   $t3, 0($t0)       # F[0] = $t3
     sw   $t4, 4($t0)       # F[1] = $t4
     addi $t1, $t5, -2       # Counter for loop, will execute (size-2) times
loop: lw   $t3, 0($t0)       # Get value from array F[n]
     lw   $t4, 4($t0)       # Get value from array F[n+1]
     add  $t2, $t3, $t4     # $t2 = F[n] + F[n+1]
     sw   $t2, 8($t0)       # Store F[n+2] = F[n] + F[n+1] in array
     addi $t0, $t0, 4       # increment address of Fib. number source
     addi $t1, $t1, -1       # decrement loop counter
     slt $t6,$zero,$t1       # if $t1 > 0, that $t6 = 1, else $t6 = 0
     addi $t7,$zero,1       # set $t7 = 1
     beq $t6,$t7,loop       # jump if $t6 = 1, that is to say, $t1 > 0
out:
   j out
```

完整汇编代码如上，可知程序先向内存数据段读取数组首地址和大小（循环次数），读取并写入 f[0]=3, f[1]=3，然后循环 18 次，执行 f[i]=f[i-1]+f[i-2]计算斐波那契数列。

如果 CPU 编写正确，DMem[0-19]内容将是 3、3、6、9、15、24、39、63、102、165、267、432、699、1131、1830、2961、4791、7752、12543、20295

实验结果

仿真结果

IP 核 ram 设置界面中 Load In 设置初始化 coe 文件的路径，

其中 IMem 内容为

```
MEMORY_INITIALIZATION_RADIX=16;
MEMORY_INITIALIZATION_VECTOR=
20080000,
200d0050,
8dad0000,
200b0054,
8d6b0000,
200c0054,
8d8c0004,
ad0b0000,
ad0c0004,
21a9fffe,
8d0b0000,
8d0c0004,
016c5020,
ad0a0008,
21080004,
2129ffff,
1d20fff9,
08000011,
```

DMem 内容为

```
MEMORY_INITIALIZATION_RADIX=16;
MEMORY_INITIALIZATION_VECTOR=
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
```

```
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000014
00000003
00000003
```

仿真得 DMem 和 reg 数据为

Address	Value
0x1B	0
0x1A	0
0x19	0
0x18	0
0x17	0
0x16	3
0x15	3
0x14	20
0x13	20295
0x12	12543
0x11	7752
0x10	4791
0xF	2961
0xE	1830
0xD	1131
0xC	699
0xB	432
0xA	267
0x9	165
0x8	102
0x7	63
0x6	39
0x5	24
0x4	15
0x3	9
0x2	6
0x1	3
0x0	3

DMem

	0
0x0	0
0x1	0
0x2	0
0x3	0
0x4	0
0x5	0
0x6	0
0x7	0
0x8	72
0x9	0
0xA	20295
0xB	7752
0xC	12543
0xD	20
0xE	0
0xF	0
0x10	0
0x11	0
0x12	0
0x13	0
0x14	0
0x15	0
0x16	0
0x17	0
0x18	0

Regfile

可见运行结果符合分析。

附录:

一、 编写过程中遇到问题和一些发现

1、位拼接作为左值，单个变量也要花括号，如

{{RegDst},{ALUSrc},{MemtoReg},{RegWrite},{MemWrite},{Branch},{ALUOp[1:0]},{Jump}}=9'b100100100;

2、"=="会严格匹配每一位，比如 module 内 5 位数字比较（一个常量，一个 input），但外部传入的是 3 位的，高位自动拓展好像是 x，就导致“==”不成立，这是一个不好发现的问题。

3、仿真遇到高阻态，多半是这个信号不存在，经常是拼写错了，有时候 ise 检查不出来。

4、发现如果 Mars 的 memory configuration 设置为默认，则编译后导出代码里面的所有 la（给某个寄存器赋值一个 32 位地址）被翻译为连续两句 lui 和 ori；若 memory configuration 设置为 text at address 0 则编译后导出代码里面的所有 la 被翻译为一句 addi。我觉得原因是 text 地址从 0 开始的话，如果 text 部分较短，地址非 0 部分未超过 16 位，那么 la 只需要给低 16 位赋值就行了，就可以直接用 addi 实现（addi 指令末 16 位为立即数）。如果地址是实打实的 32 位那 addi 就无能为力了，这时候需要 lui 给高 16 位赋值，低 16 位用 ori（ori 的 rs 取$0，作用就和 addi 一样）。

二、 模块源代码

top.v

```
module top(
    input clk,
    input rst_n,
    );

    wire ALUSrc;
    wire [31:0] ALUSrcA;
    wire [31:0] ALUSrcB;
    wire [2:0] ALUControl;
    wire [31:0] ALUResult;
    wire Zero;

    wire [31:0] SignExtented;

    wire [5:0] RegRdaddr1;
    wire [31:0] RegRdout1;
    wire [5:0] RegRdaddr2;
    wire [31:0] RegRdout2;
    wire [5:0] RegWdaddr;
    wire [31:0] RegWdin;
    wire RegWrite;
    wire RegDst;

    wire [31:0] DMemaddr;
    wire [31:0] DMemout;
    wire [31:0] DMemin;
    wire DMemWrite;
    wire [31:0] IMemRdaddr;
    wire [31:0] IMemRdout;
    wire MemtoReg;

    wire [31:0] Instr;

    reg [31:0] PC=0;
    wire [31:0] nextPC;

    wire [5:0] Funct;
    wire [15:0] IMM16;
    wire [4:0] Rd;
    wire [4:0] Rt;
    wire [4:0] Rs;
    wire [5:0] Op;
```

```
wire [25:0] JumpIMM;
wire Jump;

wire Branch;
integer first;

//======================PC=======================
always @(posedge clk or negedge rst_n)
   if(~rst_n) begin first <= 1; PC <= 0; end
   else if(first == 1) begin first <= 0; PC <= PC; end
   else
      PC <= nextPC;

nextpclogic nextpclogic(PC,Jump,JumpIMM,Branch,Zero,SignExtented,nextPC);

//=====================Instruction Memory======================
assign IMemRdaddr = PC >> 2;
//>>2 是因为这里 IMem 是每个地址存储 4 字节，和实际上的（一地址一字节）不一样
IMem IMem(0,0,IMemRdaddr,0,0,IMemRdout);
assign Instr = IMemRdout;

assign JumpIMM = Instr[25:0];
assign Funct = Instr[5:0];
assign IMM16 = Instr[15:0];
assign Rd = Instr[15:11];
assign Rt = Instr[20:16];
assign Rs = Instr[25:21];
assign Op = Instr[31:26];

//======================Regfile=======================
assign RegRdaddr1 = Rs;
assign RegRdaddr2 = Rt;
mux #(6) MUXRegWdaddr(RegDst,{1'b0,Rt},{1'b0,Rd},RegWdaddr);
regfile regfile(~clk,rst_n,RegRdaddr1,RegRdout1,RegRdaddr2,RegRdout2,
RegWdaddr,RegWdin,RegWrite);

//=========================ALU==========================
assign SignExtented = {{16{IMM16[15]}},IMM16};
assign ALUSrcA = RegRdout1;
mux MUXALUSrc(ALUSrc,RegRdout2,SignExtented,ALUSrcB);
alu alu(ALUSrcA,ALUSrcB,{2'b00,{ALUControl}},ALUResult,Zero);

//======================Control=======================
control control(clk,Op,Funct,RegDst,ALUSrc,MemtoReg,RegWrite,DMemWrite,
Branch,ALUControl,Jump);
```

```verilog
    //======================Data Memory=======================
    assign DMemaddr = ALUResult >> 2;//>>2 理由同上
    assign DMemin = RegRdout2;
    DMem DMem(DMemaddr,DMemin,~clk,DMemWrite,DMemout);
    mux MUXtoReg(MemtoReg,ALUResult,DMemout,RegWdin);

endmodule
```

alu.v

```verilog
parameter A_NOP =5'h00; //nop
parameter A_ADD =5'h01; //sign_add
parameter A_SUB =5'h02; //sign_sub
parameter A_AND =5'h03; //and
parameter A_OR  =5'h04; //or
parameter A_XOR =5'h05; //xor
parameter A_NOR =5'h06; //nor
parameter A_SLT =5'h07; //slt

module alu(
    input [31:0] alu_a,
    input [31:0] alu_b,
    input [4:0] alu_op,
    output reg [31:0] alu_out,
     output zero
    );
    assign zero = (alu_out == 32'b0)?1:0;
    always@(*)
       case (alu_op)
           A_NOP: alu_out = 0;
           A_ADD: alu_out = alu_a + alu_b;
           A_SUB: alu_out = alu_a - alu_b;
           A_AND: alu_out = alu_a & alu_b;
           A_OR : alu_out = alu_a | alu_b;
           A_XOR: alu_out = alu_a ^ alu_b;
           A_NOR: alu_out = ~(alu_a | alu_b);
           A_SLT: //a<b(signed) return 1 else return 0;
               begin
                   if(alu_a[31] == alu_b[31]) alu_out = (alu_a < alu_b) ? 32'b1 : 32'b0;
                   //同号情况，后面的小于是视为无符号的比较
                   else alu_out = (alu_a[31] < alu_b[31]) ? 32'b0 : 32'b1;
                   //有符号比较符号
               end
           default: ;
```

```
        endcase
endmodule
```

regfile.v

```
module regfile(
    input   clk,
    input rst_n,
    input [5:0] rAddr1,//读地址 1
    output [31:0] rDout1,//读数据 1
    input [5:0] rAddr2,//读地址 2
    output [31:0] rDout2,//读数据 2
    input [5:0] wAddr,//写地址
    input [31:0] wDin,//写数据
    input wEna//写使能

);
    reg [31:0] data [0:63];
    integer i;
    assign rDout1=data[rAddr1];//读 1
    assign rDout2=data[rAddr2];//读 2
    always@(posedge clk or negedge rst_n)//写和复位
        if(~rst_n)
        begin
            for(i=0; i<64; i=i+1) data[i]<=0;
        end
        else
        begin
            if(wEna)
                data[wAddr]<=wDin;
        end
endmodule
```

nextpclogic.v

```
module nextpclogic(
   input [31:0] PC,
    input Jump,
    input [25:0] JumpIMM,
   input Branch,
   input Zero,
    input [31:0] SignExtented,
    output [31:0] nextPC
   );
```

```
    wire [31:0] ShiftLeft2;
    wire PCSrc;
    wire [31:0] PCPlus4;
    wire [31:0] PCBeq;
    wire [31:0] tmpPC;
    wire tmpzero1,tmpzero2;//no used

    assign ShiftLeft2 = SignExtented << 2;
    alu ALUPCPlus4(PC,4,A_ADD,PCPlus4,tmpzero1);
    alu ALUPCOffset(ShiftLeft2,PCPlus4,A_ADD,PCBeq,tmpzero2);
    and(PCSrc,Branch,Zero);
    mux MUXPC(PCSrc,PCPlus4,PCBeq,tmpPC);
    mux MUXPC2(Jump,tmpPC,{{PCPlus4[31:28]},{{2'b00,JumpIMM}<<2}},nextPC);
endmodule
```

control.v

```
module control(
    input clk,
    input [5:0] Op, //instr[31:26]
    input [5:0] Funct,//instr[5:0]
    output reg RegDst,
    output reg ALUSrc,
    output reg MemtoReg,
    output reg RegWrite,
    output reg MemWrite,
    output reg Branch,
    output reg [2:0] ALUContol,
    output reg Jump
    );
    reg [1:0] ALUOp;

    always @(*)
    begin
        case(Op)
            6'b000000://R type
                {{RegDst},{ALUSrc},{MemtoReg},{RegWrite},{MemWrite},{Branch},
                 {ALUOp[1:0]},{Jump}}=9'b100100100;
            6'b100011://lw
                {{RegDst},{ALUSrc},{MemtoReg},{RegWrite},{MemWrite},{Branch},
                 {ALUOp[1:0]},{Jump}}=9'b011100000;
            6'b101011://sw
                {{RegDst},{ALUSrc},{MemtoReg},{RegWrite},{MemWrite},{Branch},
                 {ALUOp[1:0]},{Jump}}=9'bx1x010000;
```

```
            6'b000100://beq
                {{RegDst},{ALUSrc},{MemtoReg},{RegWrite},{MemWrite},{Branch},
                 {ALUOp[1:0]},{Jump}}=9'bx0x001010;
            6'b000010://jump
                {{RegDst},{ALUSrc},{MemtoReg},{RegWrite},{MemWrite},{Branch},
                 {ALUOp[1:0]},{Jump}}=9'bxxxxxxxx1;
            6'b001000://addi
                {{RegDst},{ALUSrc},{MemtoReg},{RegWrite},{MemWrite},{Branch},
                 {ALUOp[1:0]},{Jump}}=9'b010100000;
            default: ;
        endcase
    end

    always @(*)
    begin
        case(ALUOp)
            2'b00://lw,sw,addi
                ALUContol = 5'h01;//add
            2'b01://beq
                ALUContol = 5'h02;//sub
            2'b10://R type
                case(Funct)
                    6'b100000: ALUContol = 5'h01;//add
                    6'b100010: ALUContol = 5'h02;//sub
                    6'b100100: ALUContol = 5'h03;//and
                    6'b100101: ALUContol = 5'h04;//or
                    6'b100110: ALUContol = 5'h05;//xor
                    6'b100111: ALUContol = 5'h06;//nor
                    6'b101010: ALUContol = 5'h07;//slt
                    default: ;
                endcase
            2'b11:
                ALUContol = 5'h00;//nop
            default: ;
        endcase
    end

endmodule
```

mux.v

```
module mux #(parameter WIDTH = 32)(
    input sel,
    input [WIDTH-1:0] d0,
    input [WIDTH-1:0] d1,
```

```
    output [WIDTH-1:0] out
    );
    assign out = (sel == 1'b1 ? d1 : d0);
endmodule
```

test.v

```
module test;

    // Inputs
    reg clk;
    reg rst_n;

    // Instantiate the Unit Under Test (UUT)
    top uut (
        .clk(clk),
        .rst_n(rst_n),
    );
    always #10 clk=~clk;
    initial begin
        // Initialize Inputs
        clk = 0;
        rst_n = 0;

        // Wait 100 ns for global reset to finish
        #100;
        rst_n = 1;

        // Add stimulus here

    end

endmodule
```