武汉大学国家网络安全学院

本科生实验报告

计算机系统基础实验

专 业 名 称 ：网络空间安全

课 程 名 称 ：计算机系统基础实验

指 导 教 师 ：

学 生 学 号 ：

学 生 姓 名 ：

二〇二三年四月

郑 重 声 明

本人呈交的实验报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本实验报告不包含他人享有著作权的内容。对本实验报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本实验报告的知识产权归属于培养单位。

本人签名：_____ 日期：_____

目 录

1 实验目的

1.1 实验目的

本实验是在理论课基础上的一次巩固实验，在对 CPU 有一定认识之后能够使用 Verilog 语言实现简易 CPU 及指令扩展。

通过本次实验，可以达到以下目的：

熟练 Verilog 语言编程；

熟练 MARS 的使用；

熟练 Vivado 及 FPGA 开发板的使用；

掌握单周期和多周期 CPU 的数据通路和模块实现；

掌握分析各个指令的执行过程及扩展方法。

2 实验环境

2.1 Verilog HDL

Verilog 是一门类 C 语言的硬件描述语言，能够清晰模拟和仿真语义，使用此语言能够轻易仿真验证，用于从算法级、门级到开关级的多种抽象设计层次的数字系统建模。用于编写 CPU。

2.2 MARS

MARS 是 MIPS 汇编模拟器，是一个轻量级的、用于教学的 MIPS 汇编语言集成开发环境（IDE），能够模拟运行 MIPS 指令和分析寄存器状态，用于调试。

2.3 ModelSim

ModelSim 是业界最优秀的 HFL 语言仿真软件，它采用直接优化的编译技术、Tcl/Tk 技术、和单一内核仿真技术，编译仿真速度快，编译的代码与平台无关，便于保护 IP 核，个性化的图形界面和用户接口，为用户加快调错提供强有力的手段，是 FPGA/ASIC 设计的首选仿真软件。用于仿真验证。

2.4 Vivado

Vivado 能够进行 Verilog 语言编程到硬件，基于 AMBA AXI4 互联规范、IP-XACT IP 封装元数据、工具命令语言(TCL)、Synopsys 系统约束(SDC) 以及其它有助于根据客户需求量身定制设计流程并符合业界标准的开放式环境，为 FPGA 实现提供平台。

2.5 Nexys 4 DDR

是一款简单易学的数字电路开发平台，能够实现从编程到硬件的实现。

3 单周期 CPU 设计

3.1 总体设计

此 CPU 实现了 add/sub/and/or/slt/sltu/addu/subu/addi/ori/lw/sw/beq/j/jal/sll/nor/lui/slti/bne/andi/srl/sllv/srlv/jr/jalr 共 26 条指令。

CPU 的主体部分的代码共包含 9 个文件，alu.v、ctrl.v、ctrl_encode_def.v、EXT.v、mux.v、NPC.v、PC.v、RF.v、sccpu.v。分别对应 alu 模块、控制信号、控制信号编码、扩展模块、多选器模块、计算 NPC、PC 寄存器、寄存器堆、总 CPU。

3.2 PC（程序计数器）

3.2.1 功能描述

程序计数器用于存放下一条指令所在单元的地址。

3.2.2 模块接口

信号名	方向	描述
NPC[31:0]	I	NPC 是经多路选择器选择后下一条指令的地址
clk	I	时钟信号
rst	I	复位信号
pc	O	输出实际情况下的下一条指令的地址

3.2.3 代码和描述说明

```
module PC( clk, rst, NPC, PC );

  input              clk;
  input              rst;
  input       [31:0] NPC;
  output reg  [31:0] PC;

  always @(posedge clk, posedge rst)
    if (rst)
      PC <= 32'h0000_0000;
//      PC <= 32'h0000_3000;
    else
      PC <= NPC;

endmodule
```

如果有复位信号，则 PC 归零，否则 PC 将变为 NPC，NPC 即为 nextpc。

3.3 RF（寄存器文件）

3.3.1 功能描述

完成寄存器的读写操作。

3.3.2 模块接口

信号名	方向	描述
clk	I	时钟信号
rst	I	复位信号
A1[4:0]	I	待读取的寄存器号 1
A2[4:0]	I	待读取的寄存器号 2
A3[4:0]	I	待写的寄存器地址
WD[31:0]	I	待写的 32 位数据
RD1[31:0]	O	A1 号寄存器内的数据
RD2[31:0]	O	A2 号寄存器内的数据
reg_sel[4:0]	I	需要读取的寄存器号
reg_data[31:0]	O	reg_sel 号寄存器内的数据

3.3.3 代码和描述说明

```
module RF(   input         clk,
             input         rst,
             input         RFWr,
             input  [4:0]  A1, A2, A3,
             input  [31:0] WD,
             output [31:0] RD1, RD2,
             input  [4:0]  reg_sel,
             output [31:0] reg_data);

reg [31:0] rf[31:0];

integer i;

always @(posedge clk, posedge rst)
  if (rst) begin    //  reset
    for (i=1; i<32; i=i+1)
      rf[i] <= 0; //  i;
   end

   else
     if (RFWr) begin
       rf[A3] <= WD;
//        $display("r[00-07]=0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X", 0, rf[1], rf[2], rf[3], rf[4], rf[5], rf[6], rf[7]);
//        $display("r[08-15]=0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X", rf[8], rf[9], rf[10], rf[11], rf[12], rf[13], rf[14], rf[15
//        $display("r[16-23]=0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X", rf[16], rf[17], rf[18], rf[19], rf[20], rf[21], rf[22], rf[
//        $display("r[24-31]=0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X", rf[24], rf[25], rf[26], rf[27], rf[28], rf[29], rf[30], rf[
       $display("r[%2d] = 0x%8X,", A3, WD);
     end


assign RD1 = (A1 != 0) ? rf[A1] : 0;
assign RD2 = (A2 != 0) ? rf[A2] : 0;
assign reg_data = (reg_sel != 0) ? rf[reg_sel] : 0;

endmodule
```

RF 完成寄存器的读写工作。基本操作是将 A1、A2 寄存器号对应的寄存器中的值取出，并以wire进行输出。当rst置为0时，所有寄存器清零。当写信号为1时，向WD(Write Address)中写入值。读寄存器时要做一次判断，如果寄存器号为 0，就置输出为 0。在 CPU 中，0 号寄存器单元应该是不可写的，值始终为 0。但是在我们信号处理时，可能有的指令会修改 0 号寄存器（例如 jr 指令），所以进行这样的处理，保证结果的正确性。

3.4 NPC

3.4.1 功能描述

计算下一条指令的地址。

3.4.2 模块接口

信号名	方向	描述
PC[31:0]	I	当前的 PC 值
A[31:0]	I	$31(ra)号寄存器的值
NPCOp[1:0]	I	NPC 选择信号
IMM[25:0]	I	26 位立即数
NPC[31:0]	O	经过计算和选择后下一条指令对应的 PC 值

3.4.3 代码和描述说明

```
`include "ctrl_encode_def.v"

module NPC(PC, NPCOp, IMM, NPC, REG);  // next pc module

   input  [31:0] PC;        // pc
   input  [31:0] REG;
   input  [1:0]  NPCOp;     // next pc operation
   input  [25:0] IMM;       // immediate
   output reg [31:0] NPC;   // next pc

   wire [31:0] PCPLUS4;

   assign PCPLUS4 = PC + 4; // pc + 4

   always @(*) begin
      case (NPCOp)
         `NPC_PLUS4:  NPC = PCPLUS4;
         `NPC_BRANCH: NPC = PCPLUS4 + {{14{IMM[15]}}, IMM[15:0], 2'b00};
         `NPC_JUMP:   NPC = {PCPLUS4[31:28], IMM[25:0], 2'b00};
         `NPC_JR:     NPC = REG;
         default:     NPC = PCPLUS4;
      endcase
   end // end always

endmodule
```

实现了计算下一条指令地址的功能。根据指令不同，NPC 共分为四种情况，即+4、branch 分支跳转、J 型指令立即数跳转、JR 指令$ra 跳转。根据 NPCOp 实现地址的选择。

3.5 ALU（算术逻辑单元）

3.5.1 功能描述

根据两个输入值和选择码进行算术或者逻辑运算。

3.5.2 模块接口

信号名	方向	描述
A[31:0]	I	输入值 A
B[31:0]	I	输入值 B
ALUOp[3:0]	I	ALU 选择信号
C[31:0]	O	计算结果
Zero	O	零标志位，标记运算结果是否为 0

3.5.3 代码和描述说明

```
`include "ctrl_encode_def.v"

module alu(A, B, ALUOp, C, Zero);

   input  signed [31:0] A, B;
   input         [3:0]  ALUOp;
   output signed [31:0] C;
   output Zero;

   reg [31:0] C;
   integer    i;

   always @( * ) begin
      case ( ALUOp )
         `ALU_NOP:  C = A;                          // NOP
         `ALU_ADD:  C = A + B;                      // ADD
         `ALU_SUB:  C = A - B;                      // SUB
         `ALU_AND:  C = A & B;                      // AND/ANDI
         `ALU_OR:   C = A | B;                      // OR/ORI
         `ALU_SLT:  C = (A < B) ? 32'd1 : 32'd0;    // SLT/SLTI
         `ALU_SLTU: C = ({1'b0, A} < {1'b0, B}) ? 32'd1 : 32'd0;
         `ALU_SLL:  C = B<<A;
         `ALU_SRL:  C = B>>A;
         `ALU_NOR:  C = ~(A|B);
         `ALU_LUI:  C = B<<16;
         default:   C = A;                          // Undefined
      endcase
   end // end always

   assign Zero = (C == 32'b0);

endmodule
```

ALU 是主要添加指令的修改模块，依据接受到的 ALUOp 信号来决定做什么操作。

3.6 EXT（扩展单元）

3.6.1 功能描述

将 16 位数扩展成 32 位数。

3.6.2 模块接口

信号名	方向	描述
Imm[15:0]	I	输入的十六位数

EXTOp[31:0]	I	EXT 选择信号（零扩展或者符号扩展）
Imm[31:0]	O	扩展后的三十二位数

3.6.3 代码和描述说明

```
module EXT( Imm16, EXTOp, Imm32 );

   input  [15:0] Imm16;
   input         EXTOp;
   output [31:0] Imm32;

   assign Imm32 = (EXTOp) ? {{16{Imm16[15]}}, Imm16} : {16'b0, Imm16}; // signed-extension or zero extension

endmodule
```

根据 EXTOp 信号来进行扩展。

3.7 mux（多选器）

3.7.1 功能描述

根据选择信号输出选择后的结果。

3.7.2 模块接口（以 mux2 为例）

信号名	方向	描述
d0[WIDTH:0]	I	输入值 d0
d1[WIDTH:0]	I	输入值 d1
s	I	多路选择信号
y[WIDTH:0]	O	经选择后的结果

3.7.3 代码和描述说明

```
// mux2
module mux2 #(parameter WIDTH = 8)
             (d0, d1,
              s, y);

    input  [WIDTH-1:0] d0, d1;
    input              s;
    output [WIDTH-1:0] y;

    assign y = ( s == 1'b1 ) ? d1:d0;

endmodule

// mux4
module mux4 #(parameter WIDTH = 8)
             (d0, d1, d2, d3,
              s, y);

    input  [WIDTH-1:0] d0, d1, d2, d3;
    input  [1:0] s;
    output [WIDTH-1:0] y;

    reg [WIDTH-1:0] y_r;

    always @( * ) begin
        case ( s )
            2'b00: y_r = d0;
            2'b01: y_r = d1;
            2'b10: y_r = d2;
            2'b11: y_r = d3;
            default: ;
        endcase
    end // end always

    assign y = y_r;

endmodule
```

```
// mux8
module mux8 #(parameter WIDTH = 8)
             (d0, d1, d2, d3,
              d4, d5, d6, d7,
              s, y);

    input  [WIDTH-1:0] d0, d1, d2, d3;
    input  [WIDTH-1:0] d4, d5, d6, d7;
    input  [2:0]       s;
    output [WIDTH-1:0] y;

    reg [WIDTH-1:0] y_r;

    always @( * ) begin
        case ( s )
            3'd0: y_r = d0;
            3'd1: y_r = d1;
            3'd2: y_r = d2;
            3'd3: y_r = d3;
            3'd4: y_r = d4;
            3'd5: y_r = d5;
            3'd6: y_r = d6;
            3'd7: y_r = d7;
            default: ;
        endcase
    end // end always

    assign y = y_r;

endmodule

// mux16
module mux16 #(parameter WIDTH = 8)
             (d0, d1, d2, d3,
              d4, d5, d6, d7,
              d8, d9, d10, d11,
              d12, d13, d14, d15,
              s, y);

    input [WIDTH-1:0] d0, d1, d2, d3;
    input [WIDTH-1:0] d4, d5, d6, d7;
    input [WIDTH-1:0] d8, d9, d10, d11;
    input [WIDTH-1:0] d12, d13, d14, d15;
    input [3:0] s;
    output [WIDTH-1:0] y;

    reg [WIDTH-1:0] y_r;

    always @( * ) begin
        case ( s )
            4'd0:  y_r = d0;
            4'd1:  y_r = d1;
            4'd2:  y_r = d2;
            4'd3:  y_r = d3;
            4'd4:  y_r = d4;
            4'd5:  y_r = d5;
            4'd6:  y_r = d6;
            4'd7:  y_r = d7;
            4'd8:  y_r = d8;
            4'd9:  y_r = d9;
            4'd10: y_r = d10;
            4'd11: y_r = d11;
            4'd12: y_r = d12;
            4'd13: y_r = d13;
            4'd14: y_r = d14;
            4'd15: y_r = d15;
            default: ;
        endcase
    end // end always

    assign y = y_r;

endmodule
```

依据选择信号来选择某一个输入的信号。mux.v 模块中设计了 2 路、4 路、8 路、16 路多选器。逻辑也非常简单，根据输入的 Op 码，输出选择后的对应结果即可，用 case 语句实现。

3.8 ctrl（控制单元）

3.8.1 功能描述

根据指令机器码输出对应的控制信号。

3.8.2 模块接口

信号名	方向	描述
Op[5:0]	I	指令 OP 段
Funct[5:0]	I	指令 Funct 段
Zero	I	Zero 信号
RegWrite	O	寄存器写信号
Memwrite	O	内存写信号
EXTOp	O	扩展信号
ALUOp[3:0]	O	ALU 信号
NPCOp[1:0]	O	NPC 信号
ALUSrc	O	立即数选取信号
sll	O	位移信号
GPRSel[1:0]	O	通用寄存器选择信号
WDSel[1:0]	O	寄存器写入数据选择

3.8.3 代码和描述说明

```
module ctrl(Op, Funct, Zero,
            RegWrite, MemWrite,
            EXTOp, ALUOp, NPCOp,
            ALUSrc, GPRSel, WDSel,sll
            );

  input  [5:0] Op;       // opcode
  input  [5:0] Funct;    // funct
  input        Zero;

  output       RegWrite; // control signal for register write
  output       MemWrite; // control signal for memory write
  output       EXTOp;    // control signal to signed extension
  output [3:0] ALUOp;    // ALU opertion
  output [1:0] NPCOp;    // next pc operation
  output       ALUSrc;   // ALU source for A
  output       sll;
  output [1:0] GPRSel;   // general purpose register selection
  output [1:0] WDSel;    // (register) write data selection

 // r format
  wire rtype  = ~|Op;
  wire i_add  = rtype& Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]&~Funct[1]&~Funct[0]; // add
  wire i_sub  = rtype& Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]& Funct[1]&~Funct[0]; // sub
  wire i_and  = rtype& Funct[5]&~Funct[4]&~Funct[3]& Funct[2]&~Funct[1]&~Funct[0]; // and
  wire i_or   = rtype& Funct[5]&~Funct[4]&~Funct[3]& Funct[2]&~Funct[1]& Funct[0]; // or
  wire i_slt  = rtype& Funct[5]&~Funct[4]& Funct[3]&~Funct[2]& Funct[1]&~Funct[0]; // slt
  wire i_sltu = rtype& Funct[5]&~Funct[4]& Funct[3]&~Funct[2]& Funct[1]& Funct[0]; // sltu
  wire i_addu = rtype& Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]&~Funct[1]& Funct[0]; // addu
  wire i_subu = rtype& Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]& Funct[1]& Funct[0]; // subu
  wire i_sll  = rtype&~Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]&~Funct[1]&~Funct[0];
  wire i_srl  = rtype&~Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]& Funct[1]&~Funct[0];
  wire i_nor  = rtype& Funct[5]&~Funct[4]&~Funct[3]& Funct[2]& Funct[1]& Funct[0];
  wire i_sllv = rtype&~Funct[5]&~Funct[4]&~Funct[3]& Funct[2]&~Funct[1]&~Funct[0];
  wire i_srlv = rtype&~Funct[5]&~Funct[4]&~Funct[3]& Funct[2]& Funct[1]&~Funct[0];
  wire i_jr   = rtype&~Funct[5]&~Funct[4]& Funct[3]&~Funct[2]&~Funct[1]&~Funct[0];
  wire i_jalr = rtype&~Funct[5]&~Funct[4]& Funct[3]&~Funct[2]&~Funct[1]& Funct[0];
```

```
  // i format
  wire i_addi = ~Op[5]&~Op[4]& Op[3]&~Op[2]&~Op[1]&~Op[0]; // addi
  wire i_ori  = ~Op[5]&~Op[4]& Op[3]& Op[2]&~Op[1]& Op[0]; // ori
  wire i_lw   =  Op[5]&~Op[4]&~Op[3]&~Op[2]& Op[1]& Op[0]; // lw
  wire i_sw   =  Op[5]&~Op[4]& Op[3]&~Op[2]& Op[1]& Op[0]; // sw
  wire i_beq  = ~Op[5]&~Op[4]&~Op[3]& Op[2]&~Op[1]&~Op[0]; // beq
  wire i_lui  = ~Op[5]&~Op[4]& Op[3]& Op[2]& Op[1]& Op[0];
  wire i_slti = ~Op[5]&~Op[4]& Op[3]&~Op[2]& Op[1]&~Op[0];
  wire i_bne  = ~Op[5]&~Op[4]&~Op[3]& Op[2]&~Op[1]& Op[0];
  wire i_andi = ~Op[5]&~Op[4]& Op[3]& Op[2]&~Op[1]&~Op[0];
 // j format
  wire i_j    = ~Op[5]&~Op[4]&~Op[3]&~Op[2]& Op[1]&~Op[0];  // j
  wire i_jal  = ~Op[5]&~Op[4]&~Op[3]&~Op[2]& Op[1]& Op[0];  // jal
 // generate control signals
 assign RegWrite   = rtype | i_lw | i_addi | i_ori | i_jal | i_lui | i_jalr | i_slti; // register write

 assign MemWrite   = i_sw;                          // memory write
 assign ALUSrc     = i_lw | i_sw | i_addi | i_ori | i_lui | i_slti;   // ALU B is from instruction immediate
 assign EXTOp      = i_addi | i_lw | i_sw | i_lui | i_slti;           // signed extension
 assign sll        = i_sll | i_srl;
 // GPRSel_RD   2'b00
 // GPRSel_RT   2'b01
 // GPRSel_31   2'b10
 assign GPRSel[0] = i_lw | i_addi | i_ori | i_lui | i_slti;
 assign GPRSel[1] = i_jal;

 // WDSel_FromALU 2'b00
 // WDSel_FromMEM 2'b01
 // WDSel_FromPC  2'b10
 assign WDSel[0] = i_lw;
 assign WDSel[1] = i_jal | i_jalr;

 // NPC_PLUS4   2'b00
 // NPC_BRANCH  2'b01
 // NPC_JUMP    2'b10
 assign NPCOp[0] = i_beq & Zero | i_bne &~Zero | i_jr | i_jalr;
 assign NPCOp[1] = i_j | i_jal | i_jr | i_jalr;
```

```
 // ALU_NOP   3'b000
 // ALU_ADD   3'b001
 // ALU_SUB   3'b010
 // ALU_AND   3'b011
 // ALU_OR    3'b100
 // ALU_SLT   3'b101
 // ALU_SLTU  3'b110
 assign ALUOp[0] = i_add | i_lw | i_sw | i_addi | i_and | i_slt | i_addu | i_srl | i_lui | i_slti | i_andi | i_srlv;
 assign ALUOp[1] = i_sub | i_beq | i_and | i_sltu | i_subu | i_nor | i_lui | i_bne | i_andi;
 assign ALUOp[2] = i_or | i_ori | i_slt | i_sltu | i_slti;
 assign ALUOp[3] = i_sll | i_srl | i_nor | i_lui | i_sllv | i_srlv;
endmodule
```

指令扩展主要需要修改的模块，主要涉及到 CPU 的各种控制操作。

3.9ctrl_encode_def

3.9.1 功能描述

将各个功能进行编码，增强代码的可读性。主要包括 NPC、ALU 对应的选择情况。

3.9.2 代码和描述说明

```
// NPC control signal
`define NPC_PLUS4   2'b00
`define NPC_BRANCH  2'b01
`define NPC_JUMP    2'b10
`define NPC_JR      2'b11

// ALU control signal
`define ALU_NOP   4'b0000
`define ALU_ADD   4'b0001
`define ALU_SUB   4'b0010
`define ALU_AND   4'b0011
`define ALU_OR    4'b0100
`define ALU_SLT   4'b0101
`define ALU_SLTU  4'b0110
`define ALU_SLL   4'b1000
`define ALU_SRL   4'b1001
`define ALU_NOR   4'b1010
`define ALU_LUI   4'b1011
```

在后续指令扩展时需要自定义指令的操作码。

3.10 sccpu

3.10.1 功能描述

建立好基础模块后就可以搭建完整的 CPU。

3.10.2 代码和描述说明

```
module sccpu( clk, rst, instr, readdata, PC, MemWrite, aluout, writedata, reg_sel, reg_data);

    input      clk;          // clock
    input      rst;          // reset
    input [31:0]  instr;     // instruction
    input [31:0]  readdata;  // data from data memory

    output [31:0] PC;        // PC address
    output        MemWrite;  // memory write
    output [31:0] aluout;    // ALU output
    output [31:0] writedata; // data to data memory

    input  [4:0] reg_sel;    // register selection (for debug use)
    output [31:0] reg_data;  // selected register data (for debug use)

    wire        RegWrite;    // control signal to register write
    wire        EXTOp;       // control signal to signed extension
    wire [3:0]  ALUOp;       // ALU opertion
    wire [1:0]  NPCOp;       // next PC operation

    wire [1:0]  WDSel;       // (register) write data selection
    wire [1:0]  GPRSel;      // general purpose register selection

    wire        ALUSrc;      // ALU source for A
    wire        Zero;        // ALU ouput zero

    wire [31:0] NPC;         // next PC

    wire [4:0]  rs;          // rs
    wire [4:0]  rt;          // rt
    wire [4:0]  rd;          // rd
    wire [5:0]  Op;          // opcode
    wire [5:0]  Funct;       // funct
    wire [15:0] Imm16;       // 16-bit immediate
    wire [31:0] Imm32;       // 32-bit immediate
    wire [25:0] IMM;         // 26-bit immediate (address)
    wire [4:0]  A3;          // register address for write
    wire [31:0] WD;          // register write data
    wire [31:0] RD1;         // register data specified by rs
    wire [31:0] B;           // operator for ALU B
    wire [31:0] sa;
    wire        sll;
```

```
wire        sll;
wire [31:0] A;
assign Op = instr[31:26];  // instruction
assign Funct = instr[5:0]; // funct
assign rs = instr[25:21];  // rs
assign rt = instr[20:16];  // rt
assign rd = instr[15:11];  // rd
assign Imm16 = instr[15:0];// 16-bit immediate
assign IMM = instr[25:0];  // 26-bit immediate
assign sa = {27'b0,instr[10:6]};
// instantiation of control unit
ctrl U_CTRL (
    .Op(Op), .Funct(Funct), .Zero(Zero),
    .RegWrite(RegWrite), .MemWrite(MemWrite),
    .EXTOp(EXTOp), .ALUOp(ALUOp), .NPCOp(NPCOp),
    .ALUSrc(ALUSrc), .GPRSel(GPRSel), .WDSel(WDSel),
    .sll(sll)
);

// instantiation of PC
PC U_PC (
    .clk(clk), .rst(rst), .NPC(NPC), .PC(PC)
);

// instantiation of NPC
NPC U_NPC (
    .PC(PC), .NPCOp(NPCOp), .IMM(IMM), .NPC(NPC), .REG(RD1)
);

// instantiation of register file
RF U_RF (
    .clk(clk), .rst(rst), .RFWr(RegWrite),
    .A1(rs), .A2(rt), .A3(A3),
    .WD(WD),
    .RD1(RD1), .RD2(writedata),
    .reg_sel(reg_sel),
    .reg_data(reg_data)
);

// mux for register data to write
mux4 #(5) U_MUX4_GPR_A3 (
    .d0(rd), .d1(rt), .d2(5'b11111), .d3(5'b0), .s(GPRSel), .y(A3)
```

```
    // mux for register data to write
    mux4 #(5) U_MUX4_GPR_A3 (
        .d0(rd), .d1(rt), .d2(5'b11111), .d3(5'b0), .s(GPRSel), .y(A3)
    );

    // mux for register address to write
    mux4 #(32) U_MUX4_GPR_WD (
        .d0(aluout), .d1(readdata), .d2(PC + 4), .d3(32'b0), .s(WDSel), .y(WD)
    );

    // mux for signed extension or zero extension
    EXT U_EXT (
        .Imm16(Imm16), .EXTOp(EXTOp), .Imm32(Imm32)
    );

    // mux for ALU B
    mux2 #(32) U_MUX_ALU_B (
        .d0(writedata), .d1(Imm32), .s(ALUSrc), .y(B)
    );
    mux2 #(32) U_MUX_ALU_A (
        .d0(RD1), .d1(sa), .s(sll), .y(A)
    );
    // instantiation of alu
    alu U_ALU (
        .A(A), .B(B), .ALUOp(ALUOp), .C(aluout), .Zero(Zero)
    );

endmodule
```

调用各个模块并通过输入控制信号来控制运行。

4 单周期 CPU 测试及结果分析

4.1 仿真代码及分析

仿真还需要将 CPU 与存储器连接。对应的 Data Memory 和 Instruction Memory 和与 cpu 连接的代码如下：

```
// instruction memory
module im(input  [8:2]  addr,
          output [31:0] dout );

  reg  [31:0] ROM[127:0];

  assign dout = ROM[addr]; // word aligned
endmodule
```

```
// data memory
module dm(clk, DMWr, addr, din, dout);
   input          clk;
   input          DMWr;
   input  [8:2]   addr;
   input  [31:0]  din;
   output [31:0]  dout;

   reg [31:0] dmem[127:0];
   wire [31:0] addrByte;

   assign addrByte = addr<<2;

   assign dout = dmem[addrByte[8:2]];

   always @(posedge clk)
      if (DMWr) begin
         dmem[addrByte[8:2]] <= din;
         $display("dmem[0x%8X] = 0x%8X,", addrByte, din);
      end

endmodule
```

```
module sccomp(clk, rstn, reg_sel, reg_data);
  input          clk;
  input          rstn;
  input [4:0]    reg_sel;
  output [31:0]  reg_data;

  wire [31:0]    instr;
  wire [31:0]    PC;
  wire           MemWrite;
  wire [31:0]    dm_addr, dm_din, dm_dout;

  wire rst = ~rstn;

 // instantiation of single-cycle CPU
  sccpu U_SCCPU(
       .clk(clk),                  // input:  cpu clock
       .rst(rst),                  // input:  reset
       .instr(instr),              // input:  instruction
       .readdata(dm_dout),         // input:  data to cpu
       .MemWrite(MemWrite),        // output: memory write signal
       .PC(PC),                    // output: PC
       .aluout(dm_addr),           // output: address from cpu to memory
       .writedata(dm_din),         // output: data from cpu to memory
       .reg_sel(reg_sel),          // input:  register selection
       .reg_data(reg_data)         // output: register data
       );

 // instantiation of data memory
  dm    U_DM(
       .clk(clk),           // input:  cpu clock
       .DMWr(MemWrite),     // input:  ram write
       .addr(dm_addr[8:2]), // input:  ram address
       .din(dm_din),        // input:  data to ram
       .dout(dm_dout)       // output: data from ram
       );

 // instantiation of intruction memory (used for simulation)
  im    U_IM (
    .addr(PC[8:2]),     // input:  rom address
    .dout(instr)        // output: instruction
  );

endmodule
```

```
addi $s1,$zero,0xc
andi $s2,$s1,0xf
andi $s3,$s1,0x4
```

```
addi $s1,$zero,1
addi $s2,$zero,2
bne  $s1,$s2,s
addi $s1,$zero,-1
s:
addi $s1,$s1,1
```

```
addi $s1,$zero,0
jal s
addi $31,$31,12
s:
addi $s1,$s1,1
jalr $s2,$31
nop
```

```
jal s
addi $ra,$ra,12
s:
addi $s2,$zero,2
jr      $ra
addi $s3,$zero,1
```

```
lui     $2, 0x5487
```

```
addi $s1,$zero,0xffffffff
addi $s2,$zero,0
nor $s3,$s1,$s1
nor $s4,$s2,$s2
nor $s5,$s1,$s2
nop
```

```
addi $s1,$zero,4
sll  $s2,$s1,1
srl  $s3,$s1,1
```

```
addi $s1,$zero,1
addi $s2,$zero,3
sllv $s3,$s1,$s2
srlv $s4,$s3,$s1
```

```
addi $s1,$zero,1
slti $s2,$s1,2
```

以上是我在测试每一条指令时所编写的代码，每次扩展一条指令时就进行测试，出错后不断 debug 来寻找出错的位置最终正确添加一条指令。

```
# r[17] = 0x00000004,
# r[18] = 0x00000008,
# r[19] = 0x00000002,
```

$s1	17	0x00000004
$s2	18	0x00000008
$s3	19	0x00000002

```
# r[17] = 0x00000000,
# r[31] = 0x004c1808,
# r[17] = 0x00000001,
# r[18] = 0x00400014,
# r[31] = 0x004c1814,
# r[17] = 0x00000002,
# r[18] = 0x004c1814,
# r[ 0] = 0x00000000,
```

$s1	17	0x00000002
$s2	18	0x00400014

以 sll，slr 和 jal 的测试指令为例，仿真结果正确。

4.2 仿真测试结果

```
# dmem[0x00000104] = 0x03345578,
# r[ 2] = 0xffff0000,
```

仿真结果正确，正确进行了排序

4.3 下载测试代码及分析

在仿真代码的 sccomp 基础上，增加了 MIO_BUS、Multi_CH32 以及七段数码管等器件，完成端口的连接。

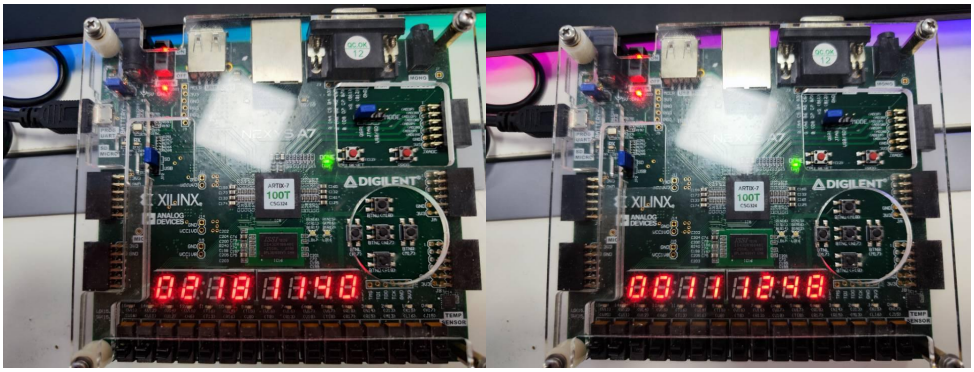
```
`timescale 1ns / 1ps

module SCCPUSOC_Top(
        input   clk,
        input   rstn,
        input  [15:0] sw_i, // output to switch
        output [7:0] disp_seg_o, disp_an_o // output to seg7
    );

    wire Clk_CPU;         // CPU clock
    wire [31:0]  instr;   // instruction
    wire [31:0]  PC;      // PC
    wire MemWrite;        // memory write
    wire [31:0]  dm_din, dm_dout; // data

    wire rst;
    assign rst = ~rstn;

    wire [31:0]  seg7_data;
    wire [6:0]  ram_addr;
    wire ram_we;
    wire seg7_we;

    wire [31:0]  cpu_data_out;       // data from CPU
    wire [31:0]  cpu_data_addr;
    wire [31:0]  ram_data_out;
    wire [31:0]  cpu_data_in;
    wire [31:0]  cpuseg7_data;
    wire [31:0]  reg_data;

   // instantiation of clock divisor
    clk_div U_CLKDIV(
          .clk(clk),       // board clock
          .rst(rst),       // reset
          .SW15(sw_i[15]), // sw15
          .Clk_CPU(Clk_CPU)// cpu clock
          );

   // instantiation of single-cycle cpu
    sccpu U_SCCPU(
          .clk(Clk_CPU),             // cpu clock
          .rst(rst),                 // reset
          .instr(instr),             // instruction
          .readdata(cpu_data_in),    // data from meory/IO to cpu
          .MemWrite(MemWrite),       // memory/IO(seg7) write signal
          .PC(PC),                   // PC
          .aluout(cpu_data_addr),    // address from cpu to memory/IO(seg7)
```

```
        .writedata(cpu_data_out),  // data from cpu to memory/IO
        .reg_sel(sw_i[4:0]),       // register selection
        .reg_data(reg_data)        // register data
        );

// instantiation of intruction memory (used for FPGA), imem is generated by IP core
 imem  U_IM(
        .a(PC[8:2]), .spo(instr)
        );

// instantiation of data memory
 dm    U_DM(
        .clk(Clk_CPU),   // cpu clock
        .DMWr(ram_we),   // ram write
        .addr(ram_addr), // ram address
        .din(dm_din),    // data to ram
        .dout(dm_dout)   // data from ram
        );

// instantiation of MIO_BUS
 MIO_BUS  U_MIO (
        .sw_i(sw_i),                  // switch
        .mem_w(MemWrite),             // memory/IO(seg7) write signal
        .cpu_data_out(cpu_data_out),  // data from cpu to memory/IO
        .cpu_data_addr(cpu_data_addr), // address from cpu to memory/IO(seg7)
        .ram_data_out(dm_dout),       // data from ram
        .cpu_data_in(cpu_data_in),    // data from memory/IO to cpu
        .ram_data_in(dm_din),         // data to ram
        .ram_addr(ram_addr),          // ram address
        .cpuseg7_data(cpuseg7_data),  // data from cpu to seg7
        .ram_we(ram_we),              // memory write signal
        .seg7_we(seg7_we)             // seg7 write signal
 );

// instantiation of Multi_CH32
 Multi_CH32 U_Multi (
         .clk(clk),                       // board clk
         .rst(rst),                       // reset
         .EN(seg7_we),                    // seg7 write enable
         .ctrl(sw_i[5:0]),                // SW[5:0]
         .Data0(cpuseg7_data),            // channel 0 (data from cpu to seg7)
          //disp_cpudata
         .data1({2'b0,PC[31:2]}),         // test channel 1--instruction no.
         .data2(PC),                      // test channel 2--PC
         .data3(instr),                   // test channel 3--instruction
         .data4(cpu_data_addr),           // test channel 4--address from cpu to memory/IO(seg7)
         .data5(cpu_data_out),            // test channel 5--data from cpu to memory/IO
         .data6(dm_dout),                 // test channel 6--data from ram
         .data7({23'b0, ram_addr, 2'b00}), // test channel 7--ram address
         .reg_data(reg_data),             // selected register data
         .seg7_data(seg7_data)            // data to seg7 display
         );

// instantiation of 16 seg7 displays
 seg7x16 U_7SEG(
        .clk(clk),           // board clock
        .rst(rst),           // reset
        .cs(1'b1),           // selection (always 1)
        .i_data(seg7_data),  // data to seg7 display
        .o_seg(disp_seg_o),  // to board disp_seg_o
        .o_sel(disp_an_o)    // to board disp_an_o
        );

endmodule
```

```
module MIO_BUS(
    input mem_w,
    input [15:0]  sw_i,               // switch input
    input [31:0]  cpu_data_out,       // data from CPU
    input [31:0]  cpu_data_addr,      // address for CPU
    input [31:0]  ram_data_out,       // data from data memory

    output reg [31:0]  cpu_data_in,   // data to CPU
    output reg [31:0]  ram_data_in,   // data to data memory
    output reg [6:0]   ram_addr,      // address for data memory
    output reg [31:0]  cpuseg7_data,  // cpu seg7 data (from sw instruction)
    output reg ram_we,                // signal to write data memory
    output reg seg7_we                // signal to write seg7 display
);


//RAM & IO decode signals:
  always @* begin

    ram_addr = 7'h0;
    ram_data_in = 32'h0;
    cpuseg7_data = 32'h0;
    cpu_data_in = 32'h0;
    seg7_we = 0;
    ram_we = 0;

    case(cpu_data_addr[31:0])
      32'hffff0004: // switch
        cpu_data_in = {16'h0, sw_i};
      32'hffff000c: begin // seg7
        cpuseg7_data = cpu_data_out;
        seg7_we = mem_w;
        end
      default: begin
        ram_addr = cpu_data_addr[8:2];
        ram_data_in = cpu_data_out;
        ram_we = mem_w;
        cpu_data_in = ram_data_out;
      end
    endcase
  end

endmodule
```

```
module    Multi_CH32(
            input clk,              // clock
            input rst,              // reset
            input EN,               // Write EN
            input[5:0]  ctrl,       // SW[5:0]
            input[31:0] Data0,      // channel 0 for CPU (sw)
            input[31:0] data1,      // data for test channel 1
            input[31:0] data2,      // data for test channel 2
            input[31:0] data3,      // data for test channel 3
            input[31:0] data4,      // data for test channel 4
            input[31:0] data5,      // data for test channel 5
            input[31:0] data6,      // data for test channel 6
            input[31:0] data7,      // data for test channel 7
            input[31:0] reg_data,   // data of selected register (ctrl[4:0] if ctrl[5] == '1')
            output reg [31:0] seg7_data // data to be displayed on seg7
            );

  reg[31:0] disp_data = 32'hAA5555AA; // default data for test channel 0

  always @( * )
      casex(ctrl) // SW[5:0]
         6'b000000: seg7_data = disp_data;
         6'b000001: seg7_data = data1;
         6'b000010: seg7_data = data2;
         6'b000011: seg7_data = data3;
         6'b000100: seg7_data = data4;
         6'b000101: seg7_data = data5;
         6'b000110: seg7_data = data6;
         6'b000111: seg7_data = data7;
         6'b001xxx: seg7_data = 32'hFFFFFFFF;
         6'b01xxxx: seg7_data = 32'hFFFFFFFF;
         6'b1xxxxx: seg7_data = reg_data;
      endcase


  always@(posedge rst, posedge clk )begin
    if (rst)
      disp_data <= 32'hAA5555AA;
    else begin
      if(EN)
        disp_data <= Data0;      //Data0
      else
        disp_data <= disp_data;
    end
  end


endmodule

////////////////////////////////////////////////////////////////////////////////////
module clk_div( input clk,
                input rst,
                input SW15,
                output Clk_CPU
               );

// Clock divider

  reg[31:0]clkdiv;

  always @ (posedge clk or posedge rst) begin
    if (rst) clkdiv <= 0; else clkdiv <= clkdiv + 1'b1; end

  assign Clk_CPU=(SW15)? clkdiv[25] : clkdiv[2];  // SW15 to select slow cpu clock or fast cpu clk

endmodule
```

4.4 下载测试结果

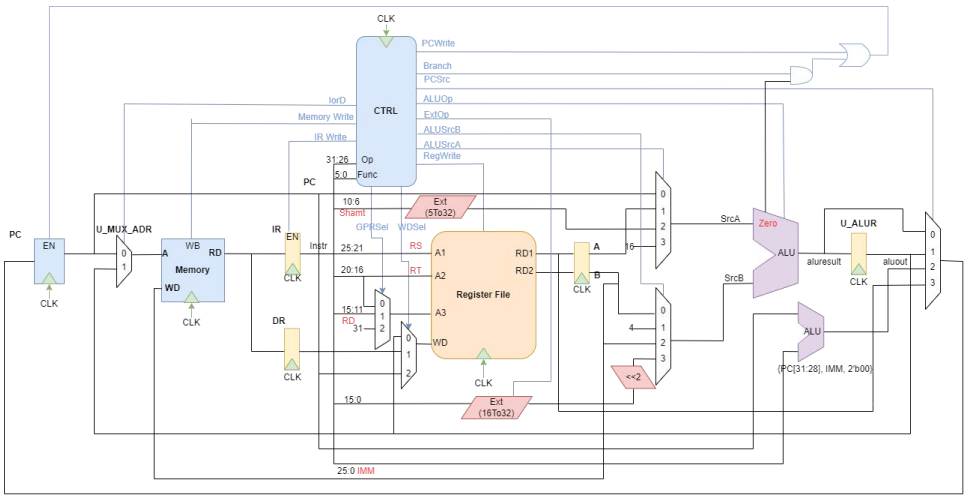


下载测试时将学号修改为自己的学号，在开发板上正确排序。

综上所述，单周期 CPU 测试完成。

5 多周期 CPU 设计

5.1 CPU 总体结构



多周期 CPU 总体数据通路及控制信号如图。

5.2 RF、EXT、mux、ALU、ctrl_encode_def

与单周期 CPU 完全相同，参照 P3 《3 单周期 CPU 设计》。

5.3 flopr（异步复位触发器）

5.3.1 功能描述

当时钟信号到来时，寄存器被赋值。当 rst 信号到来，则直接清零（实现异步）。

5.3.2 模块接口

信号名	方向	描述
clk	I	时钟信号
rst	I	复位信号
d [WIDTH-1:0]	I	寄存器赋值
q [WIDTH-1:0]	O	寄存器值

5.3.3 代码和描述说明

.v

```
module flopr #(parameter WIDTH = 8)
              (clk, rst, d, q);
   input              clk;
   input              rst;
   input  [WIDTH-1:0] d;
   output [WIDTH-1:0] q;

   reg [WIDTH-1:0] q_r;

   always @(posedge clk or posedge rst) begin
      if ( rst )
         q_r <= 0;
      else
         q_r <= d;
   end // end always

   assign q = q_r;

endmodule
```

5.4 flopenr（异步复位使能触发器）

5.4.1 功能描述

与 flopr 不同的是，只有当时钟信号到来且使能信号为正时，寄存器才会被赋值。

5.4.2 模块接口

信号名	方向	描述
clk	I	时钟信号
rst	I	复位信号
d [WIDTH-1:0]	I	寄存器赋值
q [WIDTH-1:0]	O	寄存器值
en	I	使能信号

5.4.3 代码和描述说明

```
module flopenr #(parameter WIDTH = 8)
               (clk, rst, en, d, q);

  input              clk;
  input              rst;
  input              en;
  input  [WIDTH-1:0] d;
  output reg [WIDTH-1:0] q;


  always @(posedge clk, posedge rst)
    if (rst)
      q <= 0;
    else if (en)
      q <= d;

endmodule
```

5.5 ctrl（控制单元）

5.5.1 功能描述

多周期与单周期相比最大的不同就是多个周期状态的切换。操作码 Op 和功能码 Func 确定指令信号仍然相同，接下来则根据指令信号依次确定各个周期内信号变化。State 保存当前状态，nextstate 寄存器保存下一个状态。当时钟信号到达时，将 nextstate 写入 state，从而实现周期的转换。

5.5.2 模块接口

大部分信号与单周期相同，参见 P9 3.8 ，此处仅列出不同信号。

信号名	方向	描述
PCWrite	O	PC 写信号
IRWrite	O	IR 写信号
ALUSrcA	O	ALU 操作数 A 选取信号
ALUSrcB	O	ALU 操作数 B 选取信号
PCSourse	O	PC 来源信号
IorD	O	内存访问信号

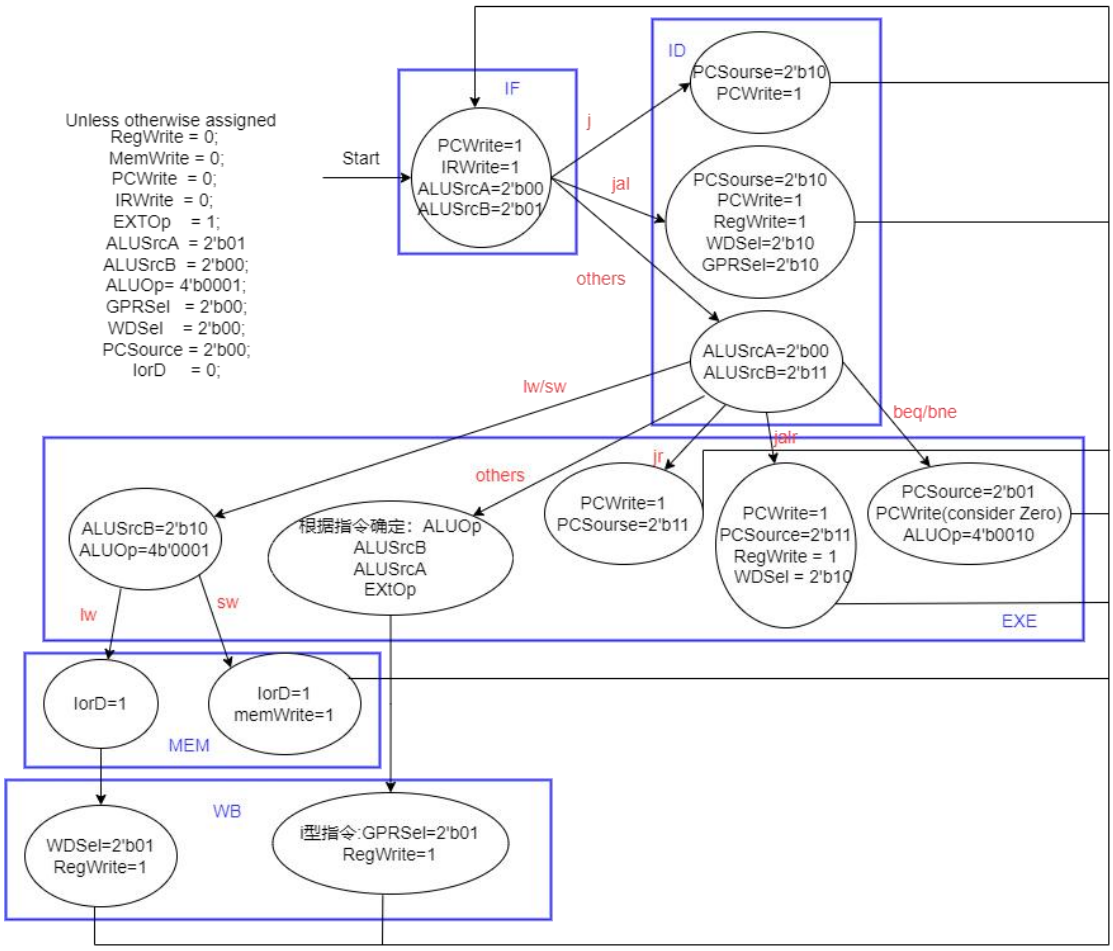
5.5.3 代码和描述说明

```
module ctrl(clk, rst, Zero, Op, Funct,
            RegWrite, MemWrite, PCWrite, IRWrite,
            EXTOp, ALUOp, PCSource, ALUSrcA, ALUSrcB,
            GPRSel, WDSel, IorD);

  input  clk, rst, Zero;
  input  [5:0] Op;       // opcode
  input  [5:0] Funct;    // funct
  output reg       RegWrite; // control signal for register write
  output reg       MemWrite; // control signal for memory write
  output reg       PCWrite;  // control signal for PC write
  output reg       IRWrite;  // control signal for IR write
  output reg       EXTOp;    // control signal to signed extension
  output reg [1:0] ALUSrcA;  // ALU source for A, 0 - PC, 1 - ReadData1
  output reg [1:0] ALUSrcB;  // ALU source for B, 0 - ReadData2, 1 - 4, 2 - extended immediate, 3 - branch offset
  output reg [3:0] ALUOp;    // ALU opertion
  output reg [1:0] PCSource; // PC source, 0- ALU, 1-ALUOut, 2-JUMP address
  output reg [1:0] GPRSel;   // general purpose register selection
  output reg [1:0] WDSel;    // (register) write data selection
  output reg       IorD;     // 0-memory access for instruction, 1 - memory access for data


  // GPRSel_RD   2'b00
  // GPRSel_RT   2'b01
  // GPRSel_31   2'b10

  // WDSel_FromALU 2'b00
  // WDSel_FromMEM 2'b01
  // WDSel_FromPC  2'b10

  // ALU_NOP   3'b000
  // ALU_ADD   3'b001
  // ALU_SUB   3'b010
  // ALU_AND   3'b011
  // ALU_OR    3'b100
  // ALU_SLT   3'b101
  // ALU_SLTU  3'b110

  parameter  [2:0] sif  = 3'b000,               // IF  state
                   sid  = 3'b001,               // ID  state
                   sexe = 3'b010,               // EXE state
                   smem = 3'b011,               // MEM state
                   swb  = 3'b100;               // WB  state
 // r format
  wire rtype  = ~|Op;
  wire i_add  = rtype& Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]&~Funct[1]&~Funct[0]; // add
  wire i_sub  = rtype& Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]& Funct[1]&~Funct[0]; // sub
  wire i_and  = rtype& Funct[5]&~Funct[4]&~Funct[3]& Funct[2]&~Funct[1]&~Funct[0]; // and
  wire i_or   = rtype& Funct[5]&~Funct[4]&~Funct[3]& Funct[2]&~Funct[1]& Funct[0]; // or
  wire i_slt  = rtype& Funct[5]&~Funct[4]& Funct[3]&~Funct[2]& Funct[1]&~Funct[0]; // slt
  wire i_sltu = rtype& Funct[5]&~Funct[4]& Funct[3]&~Funct[2]& Funct[1]& Funct[0]; // sltu
  wire i_addu = rtype& Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]&~Funct[1]& Funct[0]; // addu
  wire i_subu = rtype& Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]& Funct[1]& Funct[0]; // subu
  wire i_nor  = rtype& Funct[5]&~Funct[4]&~Funct[3]& Funct[2]& Funct[1]& Funct[0];
  wire i_sll  = rtype&~Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]&~Funct[1]&~Funct[0];
  wire i_srl  = rtype&~Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]& Funct[1]&~Funct[0];
  wire i_sllv = rtype&~Funct[5]&~Funct[4]&~Funct[3]& Funct[2]&~Funct[1]&~Funct[0];
  wire i_srlv = rtype&~Funct[5]&~Funct[4]&~Funct[3]& Funct[2]& Funct[1]&~Funct[0];
  wire i_jr   = rtype&~Funct[5]&~Funct[4]& Funct[3]&~Funct[2]&~Funct[1]&~Funct[0];
  wire i_jalr = rtype&~Funct[5]&~Funct[4]& Funct[3]&~Funct[2]&~Funct[1]& Funct[0];
 // i format
 // i format
  wire i_addi = ~Op[5]&~Op[4]& Op[3]&~Op[2]&~Op[1]&~Op[0]; // addi
  wire i_ori  = ~Op[5]&~Op[4]& Op[3]& Op[2]&~Op[1]& Op[0]; // ori
```

```
wire i_lw   =  Op[5]&~Op[4]&~Op[3]&~Op[2]& Op[1]& Op[0]; // lw
wire i_sw   =  Op[5]&~Op[4]& Op[3]&~Op[2]& Op[1]& Op[0]; // sw
wire i_beq  = ~Op[5]&~Op[4]&~Op[3]& Op[2]&~Op[1]&~Op[0]; // beq
wire i_lui  = ~Op[5]&~Op[4]& Op[3]& Op[2]& Op[1]& Op[0];
wire i_slti = ~Op[5]&~Op[4]& Op[3]&~Op[2]& Op[1]&~Op[0];
wire i_andi = ~Op[5]&~Op[4]& Op[3]& Op[2]&~Op[1]&~Op[0];
wire i_bne  = ~Op[5]&~Op[4]&~Op[3]& Op[2]&~Op[1]& Op[0];
// j format
wire i_j    = ~Op[5]&~Op[4]&~Op[3]&~Op[2]& Op[1]&~Op[0];  // j
wire i_jal  = ~Op[5]&~Op[4]&~Op[3]&~Op[2]& Op[1]& Op[0];  // jal

// valid instruction
wire i_valid = i_add | i_sub  | i_and  | i_or   | i_subu |
               i_slt | i_sltu | i_addu | i_addi |
               i_ori | i_lw   | i_sw   | i_beq  |
               i_j   | i_jal  | i_nor  | i_sll  |
               i_srl | i_lui  | i_slti | i_andi | i_sllv |
               i_srlv | i_bne | i_jr   | i_jalr;

/***********************************************/
/******              FSM                  ******/
reg [2:0] nextstate;
reg [2:0] state;

always @(posedge clk or posedge rst) begin
  if ( rst )
    state <= sif;
  else
    state <= nextstate;
end // end always

/***********************************************/
/******         Control Signal            ******/
always @(*) begin
  RegWrite = 0;
  MemWrite = 0;
  PCWrite  = 0;
  IRWrite  = 0;
  EXTOp    = 1;           // signed extension
  ALUSrcA  = 2'b01;           // 1 - ReadData1
  ALUSrcB  = 2'b00;       // 0 - ReadData2
  ALUOp    = 4'b0001;      // ALU_ADD       3'b001
  GPRSel   = 2'b00;       // GPRSel_RD     2'b00
  WDSel    = 2'b00;       // WDSel_FromALU 2'b00
  PCSource = 2'b00;       // PC + 4 (ALU)
  IorD     = 0;           // 0-memory access for instruction
  case (state)
    sif: begin
      PCWrite = 1;
      IRWrite = 1;
      ALUSrcA = 2'b00;      // PC
      ALUSrcB = 2'b01;  // 4
      nextstate = sid;
    end

    sid: begin
      if ((i_valid === 1'b0) || (i_valid === 1'bX) || (i_valid === 1'bZ))  begin
        nextstate = sif;  // invalid instruction
      end else if (i_j) begin
        PCSource = 2'b10; // JUMP address
        PCWrite = 1;
        nextstate = sif;
      end else if (i_jal) begin
```

```
      end else if (i_jal) begin
        PCSource = 2'b10; // JUMP address
        PCWrite = 1;
        RegWrite = 1;
        WDSel = 2'b10;    // WDSel_FromPC  2'b10
        GPRSel = 2'b10;   // GPRSel_31     2'b10
        nextstate = sif;
       end else begin
        ALUSrcA = 2'b00;       // PC
        ALUSrcB = 2'b11;   // branch offset
        nextstate = sexe;
      end
    end

    sexe: begin
      ALUOp[0] = i_add | i_lw | i_sw | i_addi | i_and | i_slt | i_addu | i_nor | i_srl | i_slti | i_andi | i_srlv;
      ALUOp[1] = i_sub | i_beq | i_and | i_sltu | i_subu | i_nor | i_lui | i_andi | i_bne;
      ALUOp[2] = i_or  | i_ori | i_slt | i_sltu | i_nor | i_slti;
      ALUOp[3] = i_sll | i_srl | i_lui | i_sllv | i_srlv;
      if (i_beq) begin
        PCSource = 2'b01; // ALUout, branch address
        PCWrite = i_beq & Zero;
        nextstate = sif;
      end else if (i_lw || i_sw) begin
        ALUSrcB = 2'b10; // select offset
        nextstate = smem;
      end else if (i_bne) begin
        PCSource = 2'b01; // ALUout, branch address
        PCWrite = i_bne & ~Zero;
        nextstate = sif;
      end else if (i_jr) begin
         PCSource = 2'b11;
         PCWrite = 1;
         nextstate = sif;
        end else if (i_jalr) begin
         PCSource = 2'b11;
         PCWrite = 1;
         RegWrite = 1;
         WDSel = 2'b10;    // WDSel_FromPC  2'b10
     //  GPRSel = 2'b10;   // GPRSel_31     2'b10
         nextstate = sif;
        end
        else begin
        if (i_addi || i_ori || i_lui || i_slti || i_andi)
          ALUSrcB = 2'b10; // select immediate
        if (i_ori || i_lui || i_andi || i_sllv || i_srlv)
          EXTOp = 0; // zero extension
        if (i_sll || i_srl) begin
          ALUSrcA = 2'b10;
          EXTOp = 0;
        end
        nextstate = swb;
      end
    end

    smem: begin
      IorD = 1;  // memory address  = ALUout
      if (i_lw) begin
        nextstate = swb;
      end else begin  // i_sw
        MemWrite = 1;
        nextstate = sif;
      end
    end

    swb: begin
      if (i_lw)
        WDSel = 2'b01;     // WDSel_FromMEM 2'b01
      if (i_lw | i_addi | i_ori | i_lui | i_slti | i_andi) begin
        GPRSel = 2'b01;    // GPRSel_RT     2'b01
      end
      RegWrite = 1;
      nextstate = sif;
    end

    default: begin
      nextstate = sif;
    end

  endcase
 end // end always

endmodule
```

与单周期最大的不同便是各个周期状态的转换，给每个信号添加了一个启用值，若值为 1 则执行，若为 0 则直接取下一条指令。



上图为状态信号转换的说明。

5.6 mccpu

5.6.1 功能描述

同单周期 CPU

5.6.2 代码和描述说明

```
module mccpu( clk, rst, instr, readdata, PC, MemWrite, adr, writedata, reg_sel, reg_data);

   input        clk;        // clock
   input        rst;        // reset

   output [31:0]  instr;    // instruction
   output [31:0]  PC;       // PC address
   input  [31:0]  readdata; // data from data memory

   output        MemWrite;  // memory write
   output [31:0] adr;       // memory address
   output [31:0] writedata; // data to data memory

   input  [4:0]  reg_sel;   // register selection (for debug use)
   output [31:0] reg_data;  // selected register data (for debug use)

   wire        RegWrite;    // control signal to register write
   wire        PCWrite;     // control signal for PC write
   wire        IRWrite;     // control signal for IR write
   wire        EXTOp;       // control signal to signed extension
   wire [3:0]  ALUOp;       // ALU opertion
   wire [1:0]  PCSource;    // next PC operation
   wire        IorD;         // memory access for instruction or data

   wire [1:0]  WDSel;       // (register) write data selection
   wire [1:0]  GPRSel;      // general purpose register selection

   wire [1:0]  ALUSrcA;     // ALU source for A
   wire [1:0]  ALUSrcB;     // ALU source for B
   wire        Zero;        // ALU ouput zero


   wire [31:0] aluresult;    // alu result
   wire [31:0] aluout;       // alu out

   wire [4:0]  rs;          // rs
   wire [4:0]  rt;          // rt
   wire [4:0]  rd;          // rd
   wire [5:0]  Op;          // opcode
   wire [5:0]  Funct;       // funct
   wire [15:0] Imm16;       // 16-bit immediate
   wire [31:0] Imm32;       // 32-bit immediate
   wire [25:0] IMM;         // 26-bit immediate (address)
   wire [4:0]  A3;          // register address for write
   wire [31:0] WD;          // register write data
   wire [31:0] RD1;         // register data specified by rs
   wire [31:0] RD2;         // register data specified by rt
```

```
wire [31:0] A;           // register A
wire [31:0] B;           // register B
wire [31:0] ALUA;        // ALU A
wire [31:0] ALUB;        // ALU B
wire [31:0] data;        // data
wire [31:0] NPC;         // NPC
wire [31:0] sa;
assign Op = instr[31:26];  // instruction
assign Funct = instr[5:0]; // funct
assign rs = instr[25:21];  // rs
assign rt = instr[20:16];  // rt
assign rd = instr[15:11];  // rd
assign Imm16 = instr[15:0];// 16-bit immediate
assign IMM = instr[25:0];  // 26-bit immediate
assign sa = instr[10:6];
// instantiation of control unit
ctrl U_CTRL (
    .clk(clk), .rst(rst), .Zero(Zero), .Op(Op), .Funct(Funct),
    .RegWrite(RegWrite), .MemWrite(MemWrite),
    .PCWrite(PCWrite), .IRWrite(IRWrite),
    .EXTOp(EXTOp), .ALUOp(ALUOp), .PCSource(PCSource),
    .ALUSrcA(ALUSrcA), .ALUSrcB(ALUSrcB),
    .GPRSel(GPRSel), .WDSel(WDSel), .IorD(IorD));

// instantiation of PC
flopenr #(32) U_PC (
   .clk(clk), .rst(rst), .en(PCWrite), .d(NPC), .q(PC)
);

// mux for PC source
mux4 #(32) U_MUX4_PC (
   .d0(aluresult), .d1(aluout), .d2({PC[31:28], IMM, 2'b00}), .d3(RD1),
   .s(PCSource), .y(NPC)
);

mux2 #(32) U_MUX_ADR (
   .d0(PC), .d1(aluout), .s(IorD), .y(adr)
);

// instantiation of IR
flopenr #(32) U_IR (
   .clk(clk), .rst(rst), .en(IRWrite), .d(readdata), .q(instr)
);

// instantiation of Data Register
flopr  #(32) U_DataR(
   .clk(clk), .rst(rst), .d(readdata), .q(data)
```

```
    );

    // instantiation of register file
    RF U_RF (
        .clk(clk), .rst(rst), .RFWr(RegWrite),
        .A1(rs), .A2(rt), .A3(A3),
        .WD(WD), .RD1(RD1), .RD2(RD2),
        .reg_sel(reg_sel), .reg_data(reg_data)
    );

    flopr  #(32) U_AR(clk, rst, RD1, A);//A register

    flopr  #(32) U_BR(clk, rst, RD2, B);//B register

    // mux for ALU A
    mux4 #(32) U_MUX_ALU_A (
        .d0(PC), .d1(A), .d2(sa), .d3(32'b0), .s(ALUSrcA), .y(ALUA)
    );
    // mux for signed extension or zero extension
    EXT U_EXT (
        .Imm16(Imm16), .EXTOp(EXTOp), .Imm32(Imm32)
    );

    // mux for ALU B
    mux4 #(32) U_MUX_ALU_B (
        .d0(B), .d1(4), .d2(Imm32), .d3({{14{Imm16[15]}}, Imm16, 2'b00}),
        .s(ALUSrcB), .y(ALUB)
    );

    // instantiation of ALU
    alu U_ALU (
        .A(ALUA), .B(ALUB), .ALUOp(ALUOp), .C(aluresult), .Zero(Zero)
    );

    // instantiation of ALUout Register
    flopr  #(32) U_ALUR(
        .clk(clk), .rst(rst), .d(aluresult), .q(aluout)
    );

    // mux for register data to write
    mux4 #(5) U_MUX4_GPR_A3 (
        .d0(rd), .d1(rt), .d2(5'b11111), .d3(5'b0), .s(GPRSel), .y(A3)
    );

    // mux for register address to write
    mux4 #(32) U_MUX4_GPR_WD (
        .d0(aluout), .d1(data), .d2(PC), .d3(32'b0), .s(WDSel), .y(WD)
    );

    assign writedata = B;

endmodule
```

6 多周期 CPU 测试及结果分析

6.1 仿真代码及分析

仿真还需要将 CPU 与存储器连接。对应的 Data Memory 与 cpu 连接的代码如下：

```
// data memory
module dm(clk, DMWr, addr, din, dout);
    input          clk;
    input          DMWr;
    input  [8:2]   addr;
    input  [31:0]  din;
    output [31:0]  dout;

    reg [31:0] dmem[127:0];
    wire [31:0] addrByte;

    assign addrByte = addr<<2;

    assign dout = dmem[addrByte[8:2]];

    always @(posedge clk)
        if (DMWr) begin
            dmem[addrByte[8:2]] <= din;
            $display("dmem[0x%8X] = 0x%8X,", addrByte, din);
        end

endmodule
```

```
module mccomp(clk, rstn, reg_sel, reg_data);
   input          clk;
   input          rstn;
   input [4:0]    reg_sel;
   output [31:0]  reg_data;

   wire [31:0]    instr;
   wire [31:0]    PC;
   wire           MemWrite;
   wire [31:0]    dm_addr, dm_din, dm_dout;

   wire rst = ~rstn;

  // instantiation of single-cycle CPU
   mccpu U_MCCPU(
         .clk(clk),                  // input:  cpu clock
         .rst(rst),                  // input:  reset
         .instr(instr),              // input:  instruction
         .readdata(dm_dout),         // input:  data to cpu
         .MemWrite(MemWrite),        // output: memory write signal
         .PC(PC),                    // output: PC
         .adr(dm_addr),              // output: address from cpu to memory
         .writedata(dm_din),         // output: data from cpu to memory
         .reg_sel(reg_sel),          // input:  register selection
         .reg_data(reg_data)         // output: register data
         );

  // instantiation of data memory
   dm    U_DM(
         .clk(clk),            // input:  cpu clock
         .DMWr(MemWrite),      // input:  ram write
         .addr(dm_addr[8:2]), // input:  ram address
         .din(dm_din),         // input:  data to ram
         .dout(dm_dout)        // output: data from ram
         );


endmodule
```

指令测试部分同单周期 CPU P18 4.1 节。

6.2 仿真测试结果

```
# dmem[0x00000104] = 0x03345578,
# r[ 2] = 0xffff0000,
```

仿真结果正确，正确进行了排序

6.3 下载测试代码及分析

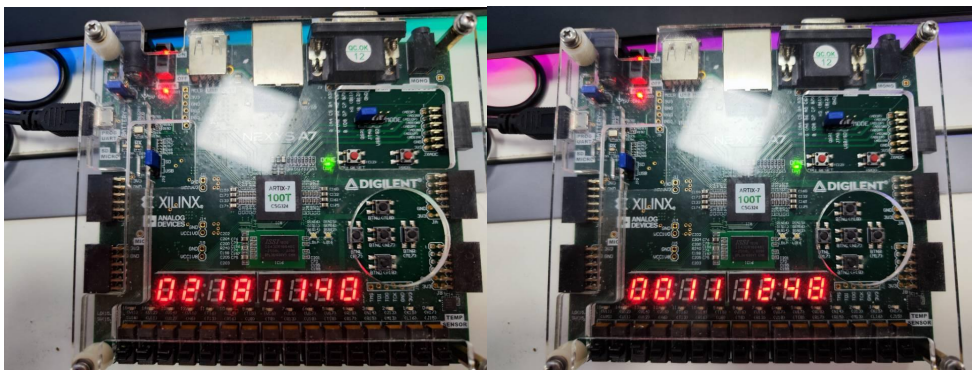
基本同单周期 CPU，除了 MCCPU 模块。包含时钟降频、MIO_BUS、Multi_CH32、七段数码管等。

```
module MCCPUSOC_Top(
        input   clk,
        input   rstn,
        input  [15:0] sw_i, // output to switch
        output [7:0] disp_seg_o, disp_an_o // output to seg7
    );

    wire Clk_CPU;         // CPU clock
    wire [31:0]  instr;   // instruction
    wire [31:0]  PC;      // PC
    wire MemWrite;        // memory write
    wire [31:0]  dm_din, dm_dout; // data

    wire rst;
    assign rst = ~rstn;

    wire [31:0]  seg7_data;
    wire [6:0]  ram_addr;
    wire ram_we;
    wire seg7_we;

    wire [31:0]  cpu_data_out;       // data from CPU
    wire [31:0]  cpu_data_addr;
    wire [31:0]  ram_data_out;
    wire [31:0]  cpu_data_in;
    wire [31:0]  cpuseg7_data;
    wire [31:0]  reg_data;

   // instantiation of clock divisor
    clk_div U_CLKDIV(
          .clk(clk),       // board clock
          .rst(rst),       // reset
          .SW15(sw_i[15]), // sw15
          .Clk_CPU(Clk_CPU)// cpu clock
          );

   // instantiation of single-cycle cpu
```

```
// instantiation of single-cycle cpu
 mccpu U_MCCPU(
        .clk(Clk_CPU),
        .rst(rst),
        .instr(instr),
        .readdata(cpu_data_in),
        .MemWrite(MemWrite),
        .PC(PC),
        .adr(cpu_data_addr),
        .writedata(cpu_data_out),
        .reg_sel(sw_i[4:0]),
        .reg_data(reg_data)
       );

// instantiation of data memory (used for FPGA), dmem is generated by IP core
 dmem    U_DM( // data memory
       .clk(Clk_CPU),
       .we(ram_we),
       .a(ram_addr),
       .d(dm_din),
       .spo(dm_dout)
       );

// instantiation of MIO_BUS
 MIO_BUS  U_MIO (
       .sw_i(sw_i),                  // switch
       .mem_w(MemWrite),             // memory/IO(seg7) write signal
       .cpu_data_out(cpu_data_out),  // data from cpu to memory/IO
       .cpu_data_addr(cpu_data_addr), // address from cpu to memory/IO(seg7)
       .ram_data_out(dm_dout),       // data from ram
       .cpu_data_in(cpu_data_in),    // data from memory/IO to cpu
       .ram_data_in(dm_din),         // data to ram
       .ram_addr(ram_addr),          // ram address
       .cpuseg7_data(cpuseg7_data),  // data from cpu to seg7
       .ram_we(ram_we),              // memory write signal
       .seg7_we(seg7_we)             // seg7 write signal
 );
```

```
        .ram_data_in(dm_din),          // data to ram
        .ram_addr(ram_addr),           // ram address
        .cpuseg7_data(cpuseg7_data),   // data from cpu to seg7
        .ram_we(ram_we),               // memory write signal
        .seg7_we(seg7_we)              // seg7 write signal
    );

    // instantiation of Multi_CH32
    Multi_CH32 U_Multi (
          .clk(clk),                       // board clk
          .rst(rst),                       // reset
          .EN(seg7_we),                    // seg7 write enable
          .ctrl(sw_i[5:0]),                // SW[5:0]
          .Data0(cpuseg7_data),            // channel 0 (data from cpu to seg7)
           //disp_cpudata
          .data1({2'b0,PC[31:2]}),         // test channel 1--instruction no.
          .data2(PC),                      // test channel 2--PC
          .data3(instr),                   // test channel 3--instruction
          .data4(cpu_data_addr),           // test channel 4--address from cpu to memory/IO(seg7)
          .data5(cpu_data_out),            // test channel 5--data from cpu to memory/IO
          .data6(dm_dout),                 // test channel 6--data from ram
          .data7({23'b0, ram_addr, 2'b00}), // test channel 7--ram address
          .reg_data(reg_data),             // selected register data
          .seg7_data(seg7_data)            // data to seg7 display
          );

    // instantiation of 16 seg7 displays
    seg7x16 U_7SEG(
          .clk(clk),           // board clock
          .rst(rst),           // reset
          .cs(1'b1),           // selection (always 1)
          .i_data(seg7_data),  // data to seg7 display
          .o_seg(disp_seg_o),  // to board disp_seg_o
          .o_sel(disp_an_o)    // to board disp_an_o
          );

endmodule
```

6.4 下载测试结果



下载测试时将学号修改为自己的学号，在开发板上正确排序。

综上所述，多周期 CPU 测试完成。

7 实验心得

本实验说难不难，说简单却也不太简单。最难的大概就是从何处下手。首先是软件的熟悉，就要花费不少时间。熟悉了软件，从众多工程中却也不知道该打开哪一个。拖了许久弄清楚了代码，改添加第一条指令时却又遇到了难关，那就是 sll 指令。sll 指令可以称得上是最难添加的指令，主要便是其 shamt 段不知从何而来。其余指令基本只用修改 ALU 和 ctrl，sll 却需要修改 sccpu 中的多路选择器，增加一个多路选择器以用来选择 shamt 段，再者，则需要添加一个位移专属的信号来控制使用这个选择器。解决完了 sll 指令，srl 也迎刃而解。其余只需要添加 OP 或是 FUNCT 码再改改 ALU 的千篇一律的指令，便显得如此简单。然后是 jr 和 jalr，在已有的 j 指令基础上很容易就能完成大部分，除了 NPC 的部分，不过有了 sll 指令和在 ALU 修改的经验，很快就能想到 NPC 也是需要修改的。在多周期部分，主要的难点就是弄清楚周期轮转的状态转换，每个周期与什么指令有关。在清楚这点后，多周期的指令添加就变得很容易了。相比于单周期的 sll，多周期采用同样的思路即可完成，而 jr 和 jalr，相比于单周期多了在 mccpu 修改的部分。在这里，sll 指令留下的 shamt 选择的宝贵经验发挥了最大的作用，在仔细搜索 mccpu 中与 pc 相关的部分后，很快就找到了在多路选择器中选择 pc 的部分，实验也就接近尾声了。

在每添加一条指令后，为了确认我确实没有出错，我会立即编写一些非常简单的汇编指令来测试，如果出错这样很容易就能发现问题所在，再配合上思考指令执行过程所需要的模块，可以很轻松地完成指令的正确添加。

此外，老师给的文件一定要仔细阅读，在 vivado 里生成多周期的 IP 核时，正是因为没有仔细阅读文档想当然地参照单周期而无法生成。最终在仔细阅读了一遍文档之后解决了问题。

8 参考文献

[1] 李亚民.计算机原理与设计——Verilog HDL 版[B].清华大学出版社，2011 年 6 月

[2] MIPS Architecture For Programmers Volume II-A：The MIPS32 Instruction Set ，Revision 5.04 ， December 11,2013

[3] 计算机组成与设计：硬件/软件接口（原书第 6 版），机械工业出版社，（美）帕特森，（美）亨尼斯著，王党辉，康继昌，安建峰等译，2022 年 6 月

教师评语评分

评语：_____

_____

_____

_____

_____

_____

_____

评分：_____

评阅人：

年　　月　　日

（备注：对该实验报告给予优点和不足的评价，并给出百分之评分。）