计算机组成原理实验报告

姓名： 张佳晨　　　　　　　　学号：11713020

一、 实验目的

利用`verilog`和`FPGA`实现单周期`CPU`

二、 实验内容

1. 译码单元
    1. 根据pc值读取指定内存地址的指令信息
    2. 按照汇编指令组成规则，对原始指令进行分解
    3. 实现寄存器的读写操作
2. 设计执行单元：
    1. 确定执行单元的功能及输入、输出端口
    2. 运算数的选择
    3. 运算功能的实现
        - 通过组合码实现对运算的控制
        - 运算
            - 算术运算(Arithmetic)，逻辑运算(Logic)
            - 移位运算(Shift)
            - 特殊运算(`slt`，`lui`等)
    4. 根据操作码执行指定功能，获取最终运算结果作为输出
2. 取址单元
    1. 定义指令 ROM 存储器
    2. 从指令 ROM 中获取指令
    3. 对 PC 值进行 +4 处理
    4. 完成几种跳转指令的PC修改功能
    5. 最终修改PC值
3. 控制单元
    1. 结合对CPU的理解，分析每条指令在数据通路的执行过程和涉及到的控制信号的取值
    2. 根据指令与控制信号的关系，得到每个控制信号与指令的逻辑表达式
4. 存储单元
    1. 定义数据存储单元 RAM
    2. 实现对数据存储器的读写操作
5. I/O单元
    1. 指定10根IO端口线，其中

        1. LED的片选信号为 ledCtrl
        2. 拨码开关的片选信号为 switchCtrl
    2. 考虑CPU外设的可拓展性（如多个IO接口单元），通过ioread模块实现多路输入数据的选择

6. 辅助模块
    1. 乘法器
    2. 除法器

三、 实验步骤

除法器

> 不保留余数，需要多个时钟周期才能完成操作

```
module div #(parameter WIDTH = 8)(
    input [WIDTH-1:0] a, // ������
    input [WIDTH/2-1:0] b, // ����
    input clk,
    input start,
    input resetn,
    output reg [WIDTH-1:0]  q, //     ��
    output reg [WIDTH/2-1:0] r, // ����
    output reg busy // ����������
 );
    integer cnt;
    reg [WIDTH-1:0] Quotient;
    reg [WIDTH+WIDTH/2-1:0] Divisor;
    reg [WIDTH+WIDTH/2:0] Remainder;
    reg [WIDTH+WIDTH/2:0] diff;
    reg sign_q;
    always@(posedge clk or negedge resetn) begin
        if (~resetn | (~(|b) & start)) begin // resetn �� ���� ÏO
             Quotient = 0;
             Divisor = 0;
             Remainder = 0;
             busy = 0;
             q = 0;
             r = 0;
        end else begin
            if (cnt > 0) begin      // cnt > 0 ���ڼ���
            cnt = cnt - 1;
            diff = Remainder - Divisor;
            Quotient = Quotient << 1;
            if (!diff[WIDTH+WIDTH/2]) begin
                Remainder = diff;
                Quotient[0] = 1'b1;
            end
            Divisor = Divisor >> 1;
            end else if (~busy & start) begin // busy = 0 �� start ��ч, ���г�’��
                busy = 1;
                sign_q = a[WIDTH-1]^b[WIDTH/2-1];
                if (a[WIDTH-1]) Remainder[WIDTH-1:0] = ~a + 1; // -
                else              Remainder[WIDTH-1:0] = a;
                if (b[WIDTH/2-1])    Divisor[WIDTH+WIDTH/2-1:WIDTH] = ~b+1; //-
                else                 Divisor[WIDTH+WIDTH/2-1:WIDTH] = b;
                Remainder[WIDTH+WIDTH/2: WIDTH] = 0;
                Divisor[WIDTH-1:0] = 0;
                Quotient = 0;
                diff = 0;
                cnt = WIDTH + 1;
            end else if (cnt == 0) begin // cnt = 0 �������
                busy = 0;
                if (sign_q) q = ~Quotient + 1;
                else        q = Quotient;
                if (a[WIDTH-1]) r = ~Remainder[WIDTH/2-1:0]+1;
                else            r = Remainder[WIDTH/2-1:0];
             end
         end
     end
endmodule
```

乘法器

```
module mul #(parameter WIDTH = 8) (
    input [WIDTH-1:0] a, // ������
    input [WIDTH-1:0] b, // ����
    output reg [WIDTH*2-1:0] c // �˪�
);
    integer cnt; // Ѫ������
    reg [WIDTH-1:0] A,B; // ��ű�������������Ť���
    reg sign; // ��λλ
    reg C;
    reg [WIDTH-1:0] P;
    reg [WIDTH-1:0] Y;
    always @(*)
        begin
        A = a;
        B = b;
        if(a[WIDTH-1] == 1) // �������Ǹ����Ĵ���
            A = ~A + 1;
        if(b[WIDTH-1] == 1) // �����Ǹ����Ĵ���
            B = ~B + 1;

        C = 0;
        P = {WIDTH{1'b0}};
        Y = B;
        for(cnt = 0; cnt<WIDTH; cnt=cnt+1) // Ѫ������
            begin
            if(Y[0] == 1)
                {C, P} = {C, P} + A; // □□□□+□□□□□□□
            {C, P, Y} = {C, P, Y} >> 1;
            end
        c = {P, Y};
        sign = a[WIDTH-1] ^ b[WIDTH-1];
        // ������ŵĴ���
        if (sign)
            c = ~c + 1;
        end
endmodule
```

ALU

```
module ALUx8(
    input [3:0] op,
    input [7:0] a,
    input [7:0] b,
    output reg [7:0] res,
    output cf,
    output ovf,
    output zf,
    output sf
    );
    wire [7:0] add_sub;
    addsubx8_wrapper u_1(.a(a), .b(b), .cf(cf), .ovf(ovf), .sf(sf), .sub(op[0]),
.sum(add_sub), .zf(zf));
    wire [7:0] res_or;
    orgatex8_wrapper u_2(.a(a), .b(b), .q(res_or));
    wire [7:0] res_and;
    andgatex8_wrapper u_3(.a(a), .b(b), .q(res_and));
    wire [7:0] res_xor;
    xorgatex8_wrapper u_4(.a(a), .b(b), .q(res_xor));
    wire [7:0] res_not;
    notgatex8_wrapper u_5(.a(a), .c(res_not));
    always@*
    begin
        if(~op[3]) // 0xxx
            if (~op[2])         res = add_sub;  // 00xx add_sub
            else // 01xx
                if (~op[1]) // 010x
                    if (~op[0]) res = res_or;   // 0101 or
                    else        res = res_and;  // 0100 and
                else        // 011x
                    if (~op[0]) res = res_not;  // 0110 not
                    else        res = res_xor;  // 0111 xor
        else // 1xxx
        begin
            if (~op[1]) // 1x0x
                if (~op[0]) res = a >> b[2:0];   // 1x00 srl  b[2:0]
                else        res = ($signed(a)) >>> b[2:0];  // 1x01 sr b[2:0] ��������
���Ƶ���λ
            else        // 1x1x
                if (~op[0]) res = a << b[2:0];  // 1x1x sll  b[2:0]
        end
    end
endmodule
```

Decoder

```
module decoder(
    input[31:0] Instruction,    // 取指单元获得的指令
    input[31:0] read_data,      // 来自 DATA RAM or I/O port 的 数据
    input[31:0] ALU_result,     // 从ALU获取的运算结果，需要拓展立即数到32位
    input[31:0] opcplus4,       // 来自 取指单元，用于 JAL
    input Jal,                  // 来自 控制单元， 说明是 JAL 指令
    input RegWrite,             // 来自 控制单元
    input MemtoReg,             // 来自 控制单元
    input RegDst,               // 来自 控制单元
    input clk,                  // 时钟信号
    input rst,                  // 复位信号
    output wire[31:0] read_data_1,  // 输出的第一位操作数
    output wire[31:0] read_data_2,  // 输出的第二位操作数
    output[31:0] Sign_extend,        // 译码单元输出的拓展后的32位立即数
    input reg [31:0] register[0:31] // 寄存器文件共32个32位寄存器
    );
    reg [4:0] write_register_address;       // 需要写入的寄存器地址
    reg [31:0] write_date;                  // 需要写入寄存器的数据
    wire[4:0] read_register_1_address;      // 需要读取的第一个寄存器 (rs) 的地址
    wire[4:0] read_register_2_address;      // 需要读取的第二个寄存器 (rt) 的地址
    wire[4:0] write_register_1_address;     // R-type 指令需要写入的寄存器 (rd) 的地址
    wire[4:0] write_register_2_address;     // I-type 指令需要写入的寄存器 (rt) 的地址
    wire[15:0] Instruction_immediate_value; // 指令中的立即数
    wire[5:0] opcode;                       // 指令码
    wire[5:0] funct;
    reg R_type, I_type, J_type, C_type;    // C_type All coprocessor instructions
    reg andi, ori;

    /* 指令中各分量的提取 */
    assign opcode = Instruction[31:26];                     // OP
    assign funct = Instruction[5:0];                        // funct
    always@* begin
        {andi, ori} = 2'b00;
        if (I_type)
            case(opcode)
            6'b001100:  {andi, ori} = 2'b10;
            6'b001101:  {andi, ori} = 2'b01;
            endcase
    end

    assign read_register_1_address = Instruction[25:21];    // rs
    assign read_register_2_address = Instruction[20:16];    // rt (R-type)
    assign write_register_1_address = Instruction[15:11];   // rd (R-type)
    assign write_register_2_address = Instruction[20:16];   // rt (I-type)
    assign Instruction_immediate_value = Instruction[15:0]; // data, rladr (I-type)

    /* 读取寄存器 */
    assign read_data_1 = register[read_register_1_address];
    assign read_data_2 = register[read_register_2_address];

    always @* begin
```

```
        {R_type, I_type, J_type, C_type} = 4'b0000;
            case(opcode)
            6'b0000_00: {R_type, I_type, J_type, C_type} = 4'b1000;
            6'b0000_1x: {R_type, I_type, J_type, C_type} = 4'b0010;
            6'b0100_xx: {R_type, I_type, J_type, C_type} = 4'b0001;
            default:    {R_type, I_type, J_type, C_type} = 4'b0100;
            endcase
//              $display("time:%3dns, {R_type, I_type, J_type, C_type} = %b", $time, {R_type,
I_type, J_type, C_type});
        /* 目标寄存器的指定 */
        if (R_type)     // R-type => rd
            if (write_register_1_address != 5'b0_0000) begin  // 避免写入 0 号寄存器
                write_register_address = write_register_1_address;
//                  $display("time:%3dns, R_type, write_register_address = %d,
write_register_1_address = %d, Instruction = %b", $time, write_register_address,
write_register_1_address, Instruction);
            end else
                register[0] = 0;
        if (I_type) begin// I-type => rt
            write_register_address = 5'b0_0001;
            if (write_register_2_address != 5'b0_0000) begin // 避免写入 0 号寄存器
                write_register_address = write_register_2_address;
//                  $display("time:%3dns, I_type, write_register_address = %d,
write_register_2_address = %d, Instruction = %b", $time, write_register_address,
write_register_2_address, Instruction);
            end else
                  register[0] = 0;
        end
        if (Jal)    // Jal
            write_register_address = 5'd31;
//          $display ("time = %3d, write_register_address = %d, write_register_1_address =
%d, write_register_2_address = %d", $time, write_register_address,
write_register_1_address, write_register_2_address);
        /* 准备要写的数据 */
        if (~MemtoReg) write_date = ALU_result;
        else write_date = read_data;
        if (RegDst) write_date = opcplus4;
//          $monitor ("time:%3dns, write_date = %h\n", $time, write_date);
    end

    /* 对寄存器的写入操作 */
    integer i;
    always @(posedge clk) begin
        if (rst == 1)
            for (i = 0; i < 32; i = i + 1)
                register[i] <= i;
        else begin
            if (RegWrite == 1) register[write_register_address] <= write_date;
        end
    end
    /* 立即数的扩展 */
    wire sign;
    assign sign = ~(andi || ori) && Instruction_immediate_value[15]; // andi ori:
ZeroExtImm else SignExtImm
    assign Sign_extend = sign ? {16'hffff, Instruction_immediate_value}:{16'h0000,
Instruction_immediate_value};

endmodule
```

执行单元

```
module execute_unitx32(
    input[31:0]  Read_data_1,       // 从译码单元的Read_data_1中来  R[rs]
    input[31:0]  Read_data_2,       // 从译码单元的Read_data_2中来  R[rt]
    input[31:0]  Sign_extend,       // 从译码单元来的扩展后的立即数
    input[5:0]   Function_opcode,   // 取指单元来的r-类型指令功能码,r-form instructions[5:0]
    input[5:0]   Exe_opcode,        // 取指单元来的操作码
    input[1:0]   ALUOp,             // 来自控制单元的运算指令控制编码
    input[4:0]   Shamt,             // 来自取指单元的instruction[10:6]，指定移位次数
    input wire   Sftmd,             // 来自控制单元的，表明是移位指令
    input        ALUSrc,            // 来自控制单元，表明第二个操作数是立即数（beq，bne除外）
    input        I_format,          // 来自控制单元，表明是除beq, bne, LW, SW之外的I-类型指令
    input        Jrn,               // 来自控制单元，书名是JR指令
    output       Zero,              // 为1表明计算值为0
    output reg[31:0] ALU_Result,    // 计算的数据结果
    output[31:0] Add_Result,        // 计算的地址结果
    input[31:0]  PC_plus_4          // 来自取指单元的PC+4
);
    wire[31:0]  Ainput,Binput;
    reg[31:0]   Sinput;
    reg[31:0]   ALU_output_mux;
    wire[32:0]  Branch_Add;
    wire[2:0]   ALU_ctl;
    wire[5:0]   Exe_code;
    wire[2:0]   Sftm;

    assign Sftm = Function_opcode[2:0];   // 实际有用的只有低三位(移位指令)
    assign Exe_code = (I_format==0) ? Function_opcode : {3'b000,Exe_opcode[2:0]};
    assign Ainput = Read_data_1;
    assign Binput = (ALUSrc == 0) ? Read_data_2 : Sign_extend[31:0]; //R/LW,SW  sft  else的
时候含LW和SW
    assign ALU_ctl[0] = (Exe_code[0] | Exe_code[3]) & ALUOp[1];      //24H AND
    assign ALU_ctl[1] = ((!Exe_code[2]) | (!ALUOp[1]));
    assign ALU_ctl[2] = (Exe_code[1] & ALUOp[1]) | ALUOp[0];

always @* begin  // 6种移位指令
       if(Sftmd)
        case(Sftm[2:0])
            3'b000:Sinput = Read_data_2 << Shamt;       //Sll rd,rt,shamt  00000
            3'b010:Sinput = Read_data_2 >> Shamt;       //Srl rd,rt,shamt  00010
            3'b100:Sinput = Read_data_2 << Read_data_1; //Sllu rd,rt,rs 000100
            3'b110:Sinput = Read_data_2 >> Read_data_1; //Srlu rd,rt,rs 000110
            3'b011:Sinput = ($signed(Read_data_2)) >>> Shamt;       //Sra rd,rt,shamt 00011
            3'b111:Sinput = ($signed(Read_data_2)) >>> Read_data_1; //Srav rd,rt,rs 00111
```

```verilog
            default:Sinput = Binput;
        endcase
       else Sinput = Binput;
    end

    always @* begin
        if(((ALU_ctl==3'b111) && (Exe_code[3]==1))||((ALU_ctl[2:1]==2'b11) &&
(I_format==1))) //slti(sub)  处理所有SLT类的问题
            ALU_Result = Read_data_1 < Read_data_2 ? 1'b1 : 1'b0;
        else if((ALU_ctl==3'b101) && (I_format==1)) // lui: load upper immediate
            ALU_Result[31:0] = {Binput[15:0], {16{1'b0}}};
        else if(Sftmd==1)   // 移位
            ALU_Result = Sinput;
        else  ALU_Result = ALU_output_mux[31:0];    // otherwise
    end

    assign Branch_Add = PC_plus_4[31:2] + Sign_extend[31:0];
    assign Add_Result = Branch_Add[31:0];   //算出的下一个PC值已经做了除4处理，所以不需左移16位
    assign Zero = (ALU_output_mux[31:0]== 32'h00000000) ? 1'b1 : 1'b0;

    always @(ALU_ctl or Ainput or Binput) begin
        case(ALU_ctl)
            3'b000:ALU_output_mux = Ainput & Binput;    // and, andi
            3'b001:ALU_output_mux = Ainput | Binput;    // or, ori
            3'b010:ALU_output_mux = Ainput + Binput;    // add, addi, lw, sw // 计算地址
            3'b011:ALU_output_mux = Ainput + Binput;    // addu, addiu
            3'b100:ALU_output_mux = Ainput ^ Binput;    // xor, xori
            3'b101:ALU_output_mux = ~(Ainput | Binput); // nor, lui
            3'b110:ALU_output_mux = Ainput - Binput;    // sub, slti, beq, bne
            3'b111:ALU_output_mux = Ainput - Binput;    // subu, sltiu, slt, sltu
            default:ALU_output_mux = 32'h00000000;
        endcase
    end
endmodule
```

取指单元

```verilog
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
module
Ifetc32(Instruction,PC_plus_4_out,Add_result,Read_data_1,Branch,nBranch,Jmp,Jal,Jrn,Zero,cl
ock,reset,opcplus4, next_PC, PC);
    output[31:0] Instruction;          // □□□□□□□□□□ġ��
    output[31:0] PC_plus_4_out;        // (pc+4)���e�ж
    input[31:0]  Add_result;           // □□□□□□e�ж,�������ɲ□□, beq �� bne
    input[31:0]  Read_data_1;          // �������뮝ж��jr���õĵ�
    input        Branch;               // ���ч��Ƶ�ж
    input        nBranch;              // ���ч��Ƶ�ж
    input        Jmp;                  // ���ч��Ƶ�ж
    input        Jal;                  // ���ч��Ƶ�ж
    input        Jrn;                  // ���ч��Ƶ�ж
    input        Zero;                 // □□□□□□e�ж, beq �� bne
    input         clock, reset;          // ʱ���źŻλ
    output[31:0] opcplus4;               // JAL��ר�õ�PC+4
    output[31:0] next_PC;
    output[31:0] PC;

    wire[31:0]   PC_plus_4;              // PC+4
    reg[31:0]     PC;                     // PC�Ĵ���������������������
    reg[31:0]    next_PC;                // 下一条指令的PC����һ����PC+4)
    reg[31:0]    opcplus4;

   //����64KB ROM����������ܵ����� 64KB ROM
    prgrom instmem(
        .clka(clock),                    // input wire clka
        .addra(PC[15:2]),                // input wire [13:0] addra
        .douta(Instruction)              // output wire [31:0] douta
    );

    assign PC_plus_4[31:2] = PC[31:2]+1'b1;
    assign PC_plus_4[1:0] = 2'b00;
    assign PC_plus_4_out = PC_plus_4[31:0];
    // beq $n ,$m if $n=$m branch   bne if $n /=$m branch jr
    always @* begin
        if ((Branch & Zero) | (nBranch & ~Zero)) next_PC = Add_result;
        else if (Jrn) next_PC = Read_data_1;
        else next_PC = PC_plus_4[31:2];
    end
    //����J��Jal���reset�Ĵ���
    always@(negedge clock) begin
        if (reset) begin
            PC = 32'b0;
            next_PC = 32'b1;
        end else begin
            if (Jmp) PC = opcplus4;
            else if (Jal) begin
                PC[31:28] = PC_plus_4_out[31:28];
                PC[27:2] = Instruction[25:0];
                PC[1:0] = 2'b0;
                opcplus4 = next_PC;
            end
            else
                PC[31:2] = next_PC;
        end
    end
endmodule
```

控制单元

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
module control32(Opcode, Function_opcode, Jrn, RegDST, ALUSrc, MemortoReg, RegWrite,
MemWrite, Branch, nBranch, Jmp, Jal, I_format, Sftmd, ALUOp);
```

```
    input[5:0]   Opcode;           // 来自取指单元instruction[31..26]
    input[5:0]   Function_opcode;  // 来自取指单元r-类型 instructions[5..0]
    output       Jrn;              // 为1表明当前指令是jr
    output       RegDST;           // 为1表明目的寄存器是rd，否则目的寄存器是rt
    output       ALUSrc;           // 为1表明第二个操作数是立即数（beq，bne除外）
    output       MemortoReg;       // 为1表明需要从存储器读数据到寄存器
    output       RegWrite;         // 为1表明该指令需要写寄存器
    output       MemWrite;         // 为1表明该指令需要写存储器
    output       Branch;           // 为1表明是Beq指令
    output       nBranch;          // 为1表明是Bne指令
    output       Jmp;              // 为1表明是J指令
    output       Jal;              // 为1表明是Jal指令
    output       I_format;         // 为1表明该指令是除beq，bne，LW，SW之外的其他I-类型指令
    output       Sftmd;            // 为1表明是移位指令
    output[1:0]  ALUOp;

    wire Jmp,I_format,Jal,Branch,nBranch;
    wire R_format;          // 为1表示是R-类型指令
    wire Lw;                // 为1表示是lw指令
    wire Sw;                // 为1表示是sw指令

    assign R_format = (Opcode==6'b0000_00) ? 1'b1:1'b0;     // --00h
    assign RegDST = R_format;                               // 说明目标是rd，否则是rt
    assign I_format = (Opcode[5:3] == 3'b001) ? 1'b1:1'b0;
    assign Lw = (Opcode == 6'b1000_11);
    assign Jal = (Opcode == 6'b0000_11);
    assign Jrn = (Opcode==6'b0000_00 & Function_opcode == 5'h08);
    assign RegWrite = (R_format & ~Jrn | I_format | Opcode == 6'b1000_11 | Opcode ==
6'b0000_11);

    assign Sw = (Opcode==6'b1010_11);
    assign ALUSrc = (I_format | Lw | Sw) ? 1'b1: 1'b0;
    assign Branch = (Opcode == 6'h4) ? 1'b1 : 1'b0;
    assign nBranch = (Opcode == 6'h5) ? 1'b1 : 1'b0;
    assign Jmp = (Opcode == 6'h2) ? 1'b1 : 1'b0;

    assign MemWrite = Sw ? 1'b1: 1'b0;
    assign MemortoReg = Lw ? 1'b1: 1'b0;
    assign Sftmd = (R_format & (Function_opcode == 5'h0 | Function_opcode == 5'h2)) ? 1'b1:
1'b0;

    assign ALUOp = {(R_format || I_format),(Branch || nBranch)};  // 是R-type或需要立即数作32
位扩展的指令1位为1,beq、bne指令则0位为1
endmodule
```

MemoryIO

```
module
memorio(caddress,address,memread,memwrite,ioread,iowrite,mread_data,ioread_data,wdata,rdata
,write_data,LEDCtrl,SwitchCtrl);
    input[31:0] caddress;       // from alu_result in executs32
    input memread;              // read memory, from control32
    input memwrite;             // write memory, from control32
    input ioread;               // read IO, from control32
    input iowrite;              // write IO, from control32
    input[31:0] mread_data;     // data from memory
    input[15:0] ioread_data;    // data from io,16 bits
    input[31:0] wdata;          // the data from idecode32,that want to write memory or io
    output[31:0] rdata;         // data from memory or IO that want to read into register
    output[31:0] write_data;    // data to memory or I/O
    output[31:0] address;       // address to mAddress and I/O
    output LEDCtrl;             // LED CS
    output SwitchCtrl;          // Switch CS
    reg[31:0] write_data;
    wire iorw;
    assign  address = caddress;
    assign  rdata = (iowrite == 1'b1) ? {16'b0, ioread_data} : memread;  // 可能是从memory读
出，也可能自io读出，自io读取的数据是rdata的低16bit
    assign  iorw = (iowrite||ioread);
    //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    assign  LEDCtrl = ioread;    // led 模块的片选信号，高电平有效；
    assign  SwitchCtrl = memread;    //switch 模块的片选信号，高电平有效
    always @* begin
        if((memwrite==1)||(iowrite==1)) begin
            write_data = wdata;
        end else begin
            write_data = 32'hzzzzzzzz;
        end
    end
endmodule

`timescale 1ns / 1ps
module dmemory32(read_data,address,write_data,Memwrite,clock);
    output[31:0] read_data;  // 从存储器中获得的数据
    input[31:0] address;     //来自memorio模块，源头是来自执行单元算出的alu_result
    input[31:0] write_data;  //来自译码单元的read_data2
    input  Memwrite;         //来自控制单元
    input  clock;

    wire clk;
    assign clk = !clock;     //  因为使用Block ram的固有延迟，RAM的地址线来不及在时钟上升沿准备好，
                             //  使得时钟上升沿数据读出有误，所以采用反相时钟，使得读出数据比地址准
                             //  备好要晚大约半个时钟，从而得到正确地址。

    //分配64KB RAM，编译器实际只用 64KB RAM
    RAM ram (
        .clka(clk),             // input wire clka
        .wea(Memwrite),             // input wire [0 : 0] wea
        .addra(address[15:2]),  // input wire [13 : 0] addra
        .dina(write_data),      // input wire [31 : 0] dina
        .douta(read_data)       // output wire [31 : 0] douta
    );
endmodule
```

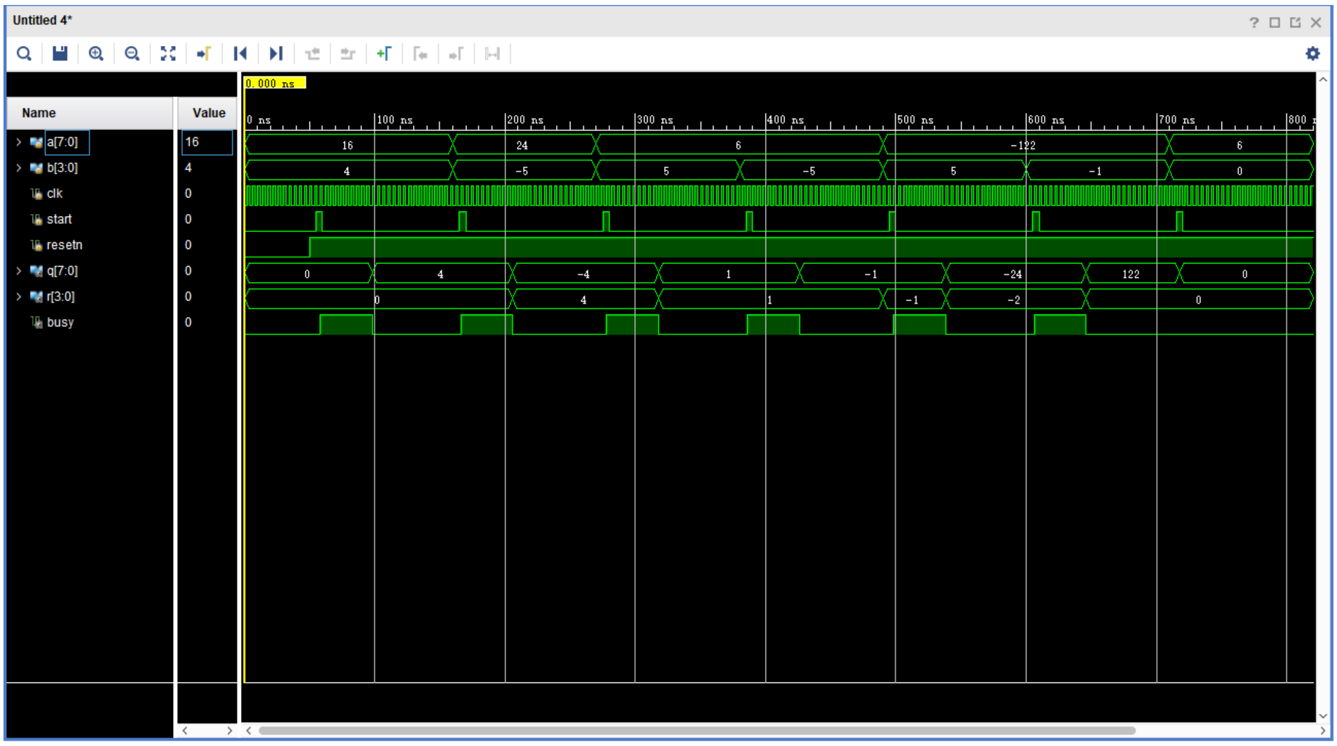
```
module ioread(reset,ior,switchctrl,ioread_data,ioread_data_switch);
    input reset;            // 复位信号
    input ior;              //  从控制器来的I/O读，
    input switchctrl;       //  从memorio经过地址高端线获得的拨码开关模块片选
    input[15:0] ioread_data_switch;  //从外设来的读数据，此处来自拨码开关
    output[15:0] ioread_data;   // 将外设来的数据送给memorio

    reg[15:0] ioread_data;

    always @* begin
        if(reset == 1)
            ioread_data = 16'b0000000000000000;
        else if(ior == 1) begin
            if(switchctrl == 1)
                ioread_data = ioread_data_switch;
            else   ioread_data = ioread_data;
        end
    end
endmodule

module leds(led_clk, ledrst, ledwrite, ledcs, ledaddr,ledwdata, ledout);
    input led_clk;              // 时钟信号
    input ledrst;               // 复位信号
    input ledwrite;            // 写信号
    input ledcs;              // 从memorio来的，由低至高位形成的LED片选信号   !!!!!!!!!!!!!!!!
    input[1:0] ledaddr;         //  到LED模块的地址低端  !!!!!!!!!!!!!!!!!!!
    input[15:0] ledwdata;     //  写到LED模块的数据，注意数据线只有16根
    output[23:0] ledout;    //  向板子上输出的24位LED信号

    reg [23:0] ledout;

    always@(posedge led_clk or posedge ledrst) begin
        if(ledrst) begin
            ledout <= 24'h000000;
        end
        //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        else if(ledcs && ledwrite) begin
            if(ledaddr == 2'b00)
                ledout[23:0] <= { ledout[23:16], ledwdata[15:0] };
            else if(ledaddr == 2'b10 )
                ledout[23:0] <= { ledwdata[7:0], ledout[15:0] };
            else
                ledout <= ledout;
        end
        else begin
            ledout <= ledout;
        end
    end
endmodule


module switchs(switclk, switrst, switchread, switchcs,switchaddr, switchrdata, switch_i);
    input switclk;                 //  时钟信号
    input switrst;                  //  复位信号
    input switchcs;                //从memorio来的，由低至高位形成的switch片选信号
!!!!!!!!!!!!!!!!!
    input[1:0] switchaddr;          //  到switch模块的地址低端  !!!!!!!!!!!!!!!
    input switchread;               //  读信号
    output [15:0] switchrdata;       //  送到CPU的拨码开关值注意数据总线只有16根
    input [23:0] switch_i;          //  从板上读的24位开关数据

    reg [15:0] switchrdata;
    always@(negedge switclk or posedge switrst) begin
        if(switrst) begin
            switchrdata <= 0;
        end
        //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        else if(switchcs && switchread) begin
            if(switchaddr==2'b00)
                switchrdata[15:0] <= switch_i[15:0];   // data output,lower 16 bits non-
extended
            else if(switchaddr==2'b10)
                switchrdata[15:0] <= { 8'h00, switch_i[23:16] }; //data output, upper 8
bits extended with zero
            else
                switchrdata <= switchrdata;
        end
        else begin
            switchrdata <= switchrdata;
        end
    end
endmodule

module top(clk, rst, ioReadCtrl, MemreadCtrl, iowriteCtrl, ledout, switch_i, MemwriteCtrl,
            register, RegAddr, RegwriteCtrl, RegreadCtrl, caddress// );
            ,ioread_data_switch, write_data, mread_data, ledwdata);
    input[31:0] caddress;           // 需要写入的地址，在memoryio中赋值给address
    input clk, rst;
    input ioReadCtrl;               // read IO, from control32
    input MemreadCtrl;              // read memory, from control32
    input MemwriteCtrl;             // write memory, from control32
    input iowriteCtrl;              // write IO, from control32
    input [23:0] switch_i;          //  从板上读的24位开关数据
    output[23:0] ledout;            //  向板子上输出的24位LED信号
    input[4:0] RegAddr;
    wire SwitchCtrl;
    input RegwriteCtrl, RegreadCtrl;
    output [31:0] register[0:31];
    wire[15:0] switchrdata;         //  送到CPU的拨码开关值注意数据总线只有16根

    // 辅助信号
    output[15:0] ioread_data_switch;
    assign ioread_data_switch = switchrdata;
    output[31:0] write_data;
    output[31:0] mread_data;
    output[15:0] ledwdata;
    /* 存储器核心单元 */
    wire[31:0] read_data;       // 从存储单元中获得的数据
    wire LEDCtrl;
    wire[31:0] rdata;           // data from memory or IO that want to read into register
    wire[31:0] address;         // address to mAddress and I/O
    wire[15:0] ioread_data;
    wire[31:0] write_data;
    wire[31:0] mread_data = MemreadCtrl ? read_data : (RegreadCtrl ? register[RegAddr] :
mread_data);

    memorio memorioU(.caddress(caddress),
                     .address(address),
                     .memread(MemreadCtrl),
                     .memwrite(MemwriteCtrl),
                     .ioread(ioReadCtrl),
                     .iowrite(iowriteCtrl),
                     .mread_data(mread_data),
                     .ioread_data(ioread_data),
                     .wdata(RegreadCtrl ? register[RegAddr] : 32'b0),
                     .rdata(rdata),
                     .write_data(write_data),              // data to memory or I/O
                     .LEDCtrl(LEDCtrl),                    // led 模块的片选信号，高电平有效;
                     .SwitchCtrl(SwitchCtrl));             //switch 模块的片选信号，高电平有效


    /* 拨码开关 */
    /* SwitchCtrl & ioReadCtrl 时可读 */
    switchs switchsU(.switclk(clk),
                     .switrst(rst),
                     .switchread(ioReadCtrl),
                     .switchcs(SwitchCtrl),
                     .switchaddr(address[1:0]),
                     .switchrdata(switchrdata),
                     .switch_i(switch_i));

    /* IO输入的多路选择器 */
    /* ioReadCtrl 时可读， SwitchCtrl 时读取拨码开关 */
    ioread ioreadU(.reset(rst),
                   .ior(ioReadCtrl),
                   .switchctrl(SwitchCtrl),
                   .ioread_data(ioread_data),
                   .ioread_data_switch(switchrdata));

    /* 存储单元 */
    dmemory32 dmemory32(.read_data(read_data),
                        .address(address),
                        .write_data(write_data),
                        .Memwrite(MemwriteCtrl),
                        .clock(clk));

    /* LED 模块 */
    wire[1:0] ledaddr = address[1:0];
    wire[15:0] ledwdata = ledaddr == 2'b00 ? mread_data[15:0] : mread_data[31:16];     //
写到LED模块的数据，注意数据线只有16根
    leds ledsU(.led_clk(clk),
               .ledrst(rst),
               .ledwrite(iowriteCtrl),
               .ledcs(LEDCtrl),
               .ledaddr(ledaddr),
               .ledwdata(ledwdata),
               .ledout(ledout));

    /* 内置寄存器 */
    reg [31:0] register[0:31];
    always@(posedge clk, negedge rst) begin
        if (rst) begin
            for (integer cnt = 0; cnt < 32; cnt = cnt + 1)
                register[cnt] <= 0;
        end
        else if (RegwriteCtrl == 1'b1) begin
            if (address[1:0] == 2'b00)
                register[RegAddr][15:0] <= ioread_data;
            else
                register[RegAddr][31:16] <= ioread_data;
        end
    end
endmodule
```

CPU

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
module minisys(rst,clk ,switch2N4 , led2N4);
    input rst;               //板上的Reset信号，低电平复位
    input clk;               //板上的100MHz时钟信号
    input[23:0] switch2N4;    //拨码开关输入
    output[23:0] led2N4;      //led结果输出到板子


    wire clock;              //clock: 分频后时钟供给系统
    wire iowrite,ioread;     //I/O读写信号
    wire[31:0] write_data;   //写RAM或IO的数据
    wire[31:0] rdata;        //读RAM或IO的数据
    wire[15:0] ioread_data;  //读IO的数据
    wire[31:0] pc_plus_4;    //PC+4
    wire[31:0] read_data_1;  //
    wire[31:0] read_data_2;  //
    wire[31:0] sign_extend;  //符号扩展
    wire[31:0] add_result;   //
    wire[31:0] alu_result;   //
    wire[31:0] read_data;    //RAM中读取的数据
    wire[31:0] address;
wire alusrc;
wire branch;
wire nbranch,jmp,jal,jrn,i_format;
wire regdst;
wire regwrite;
wire zero;
wire memwrite;
wire memread;
wire memoriotoreg;
wire memreg;
wire sftmd;
wire[1:0] aluop;
wire[31:0] instruction;
    wire[31:0] opcplus4;
    wire ledctrl,switchctrl;
    wire[15:0] ioread_data_switch;




    cpuclk cpuclk(
        .clk_in1(clk),    //100MHz
        .clk_out1(clock)    //cpuclock
    );

    Ifetc32 ifetch(
        .Instruction(instruction),
        .PC_plus_4_out(pc_plus_4),
        .Add_result(add_result),
        .Read_data_1(read_data_1),
        .Branch(branch),
        .nBranch(nbranch),
        .Jmp(jmp),
        .Jal(jal),
        .Jrn(jrn),
        .Zero(zero),
        .clock(clock),
        .opcplus4(opcplus4),
        .reset(rst)
    );

    Idecode32 idecode(
        .read_data_1(read_data_1),
        .read_data_2(read_data_2),
        .Instruction(instruction),
        .read_data(rdata),
        .ALU_result(alu_result),
        .Jal(jal),
        .RegWrite(regwrite),
        .MemorIOtoReg(memoriotoreg),
        .RegDst(regdst),
        .Sign_extend(sign_extend),
        .clock(clock),
        .reset(rst),
        .opcplus4(opcplus4)
    );


    control32 control(
        .Opcode(instruction[31:26]),
        .Function_opcode(instruction[5:0]),
        .Alu_resultHigh(alu_result[31:10]),
        .Jrn(jrn),
        .RegDST(regdst),
        .ALUSrc(alusrc),
        .MemorIOtoReg(memoriotoreg),
        .RegWrite(regwrite),
        .MemRead(memread),
        .MemWrite(memwrite),
        .IORead(ioread),
        .IOWrite(iowrite),
        .Branch(branch),
        .nBranch(nbranch),
        .Jmp(jmp),
        .Jal(jal),
        .I_format(i_format),
        .Sftmd(sftmd),
        .ALUOp(aluop)
    );

    Executs32 execute(
        .Read_data_1(read_data_1),
        .Read_data_2(read_data_2),
        .Sign_extend(sign_extend),
        .Function_opcode(instruction[5:0]),
        .Exe_opcode(instruction[31:26]),
        .ALUOp(aluop),
        .Shamt(instruction[10:6]),
        .Sftmd(sftmd),
        .ALUSrc(alusrc),
        .I_format(i_format),
        .Zero(zero),
        .Jrn(jrn),
        .ALU_Result(alu_result),
        .Add_Result(add_result),
        .PC_plus_4(pc_plus_4)
     );

    dmemory32 memory(
        .read_data(read_data),
        .address(address),
        .write_data(write_data),
        .Memwrite(memwrite),
        .clock(clock)     //16.67MHz
    );
    memorio memio(
        .caddress(alu_result),
        .address(address),
        .memread(memread),
        .memwrite(memwrite),
        .ioread(ioread),
        .iowrite(iowrite),
        .mread_data(read_data),
        .ioread_data(ioread_data),
        .wdata(read_data_2),
        .rdata(rdata),
        .write_data(write_data),
        .LEDCtrl(ledctrl),
        .SwitchCtrl(switchctrl)
    );
    ioread multiioread(
        .reset(rst),
        .ior(ioread),
        .switchctrl(switchctrl),
        .ioread_data(ioread_data),
        .ioread_data_switch(ioread_data_switch)
    );
    leds led24(
    .led_clk(clock),
    .ledrst(rst),
    .ledwrite(iowrite),
    .ledcs(ledctrl),
    .ledaddr(address[1:0]),
    .ledwdata(write_data[15:0]),
    .ledout(led2N4)
    );
    switchs switch24(
    .switclk(clock),
    .switrst(rst),
    .switchread(ioread),
    .switchcs(switchctrl),
    .switchaddr(address[1:0]),
    .switchrdata(ioread_data_switch),
    .switch_i(switch2N4)
    );
endmodule
```

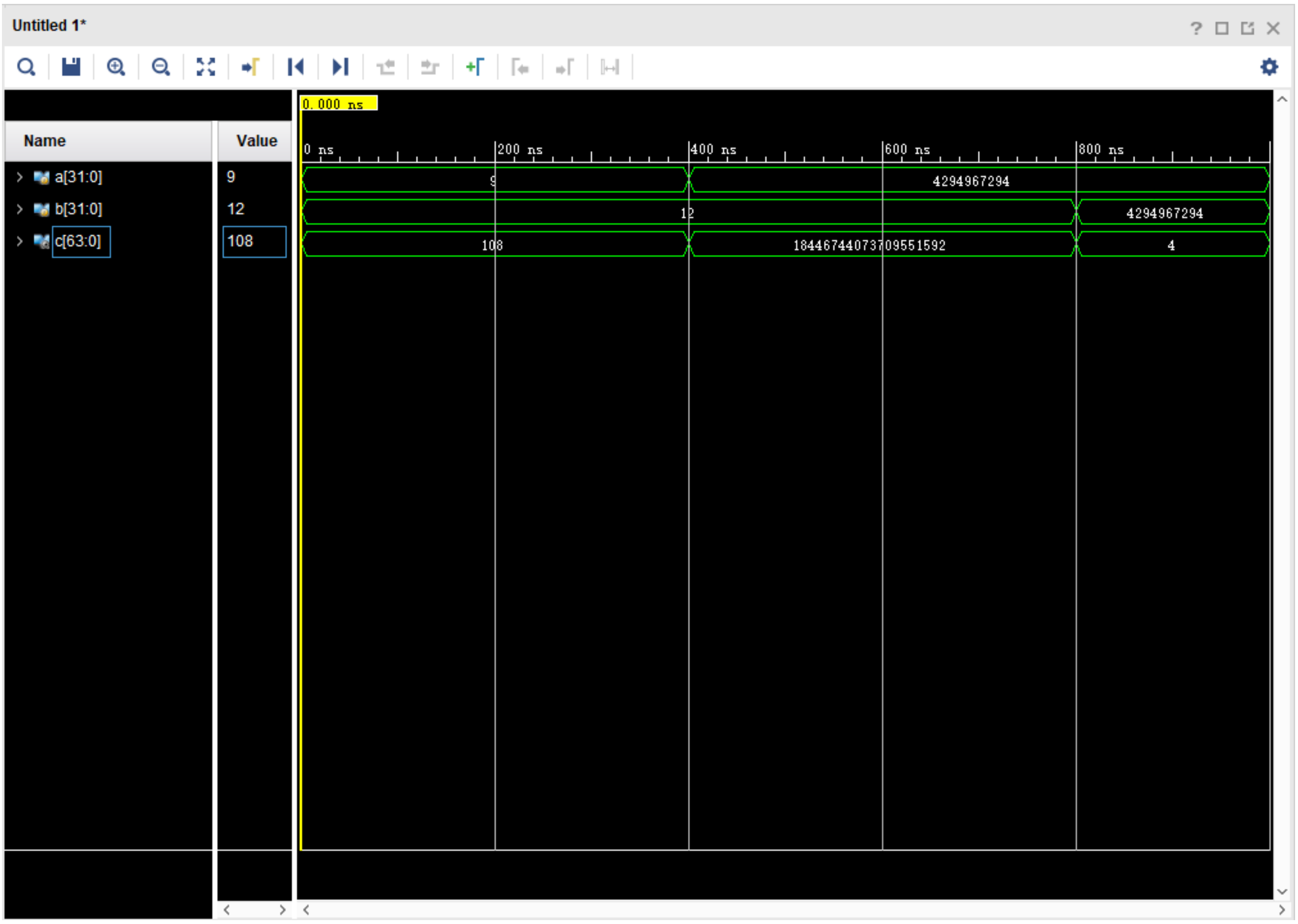
四、 实验结果（截图并配以适当的文字说明）

除法器

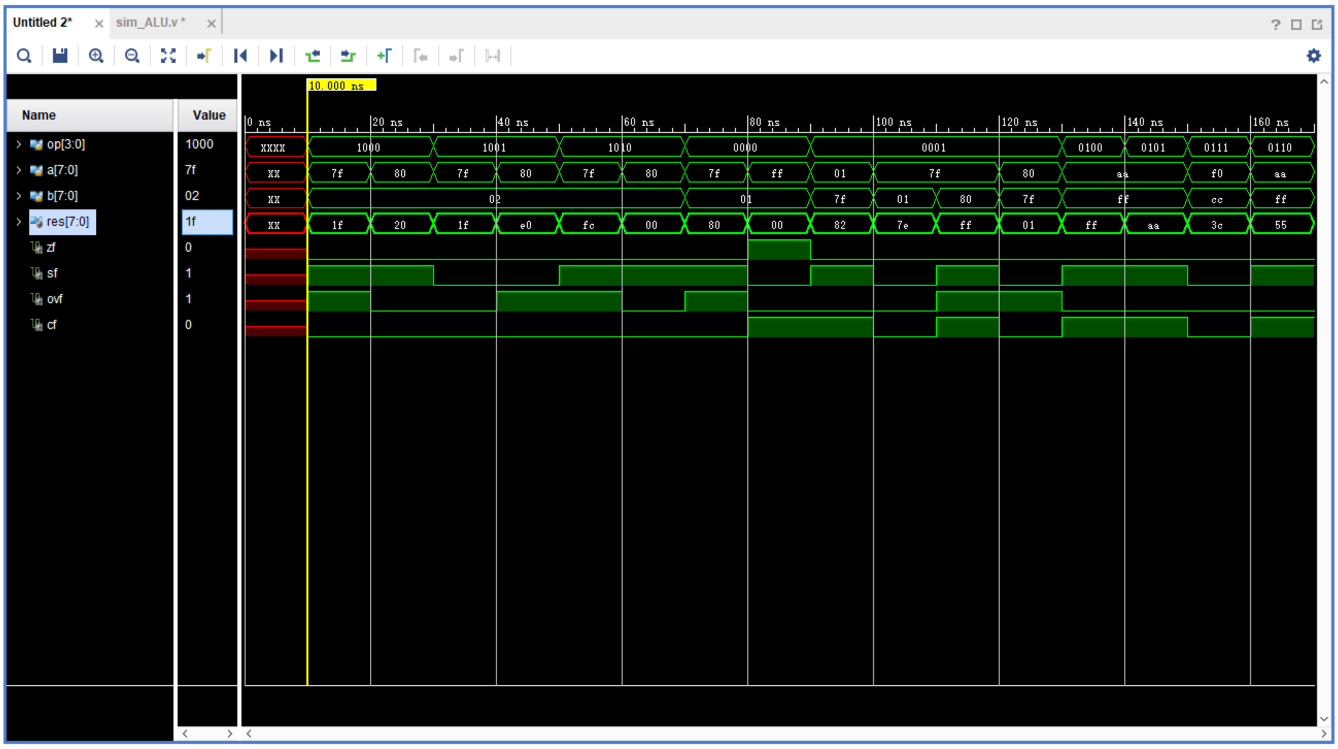


```
#50 resetn = 1;
#5start = 1;
#5 start = 0;
#100 begin a = 8'd24;b = 4'd11;end
#5start = 1;
#5 start = 0;
#100 begin a = 8'd6;b = 4'd5;end
#5start = 1;
#5 start = 0;
#100 begin a = 8'd6;b = 4'd11;end
#5start = 1;
#5 start = 0;
#100 begin a = 8'd134;b = 4'd5;end
#5start = 1;
#5 start = 0;
#100 begin a = 8'd134;b = 4'd15;end
#5start = 1;
#5 start = 0;
#100 begin a = 8'd6;b = 4'd0;end
#5start = 1;
#5 start = 0;
#100 $finish;
```

乘法器



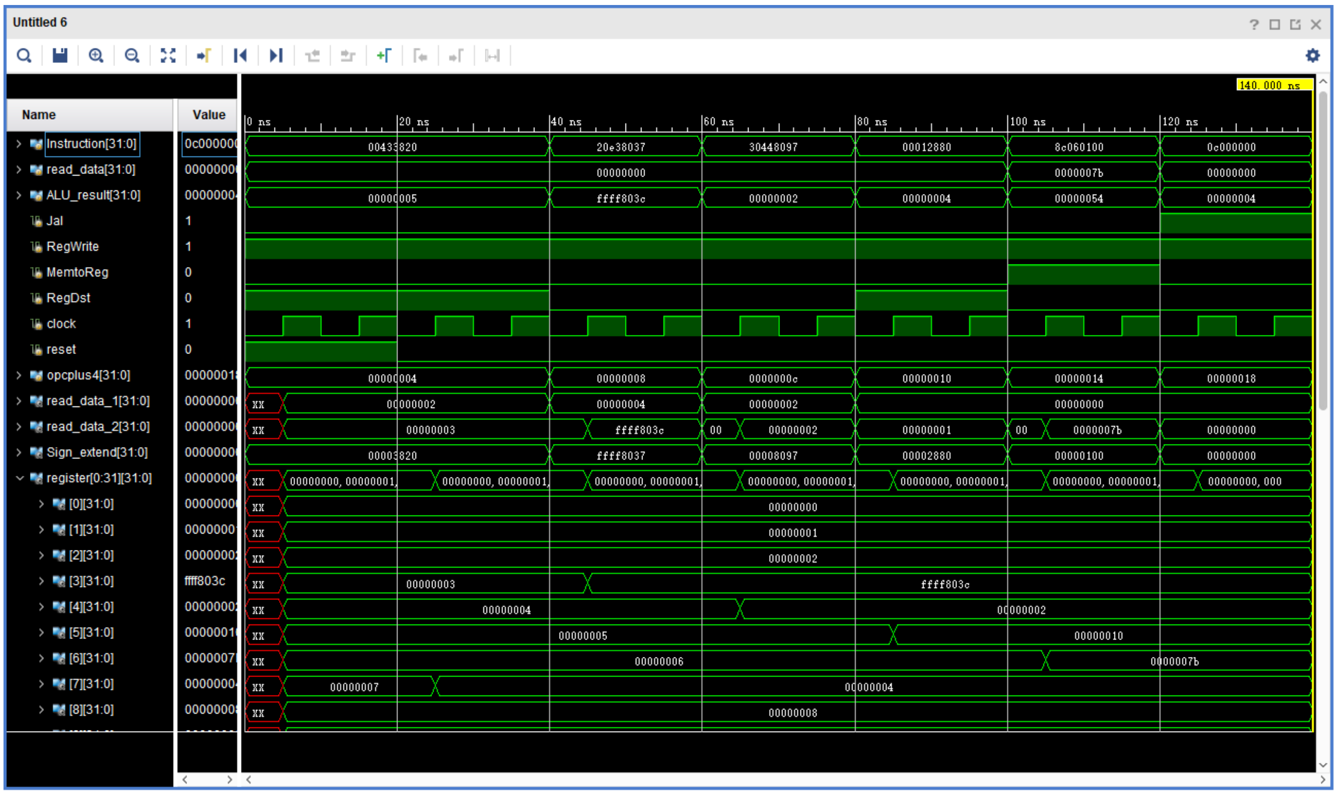
ALU



```
// shift right
#10 op = 4'b 1000; a = 8'b0111_1111; b = 8'b0000_0010;
```

```
#10 op = 4'b 1000; a = 8'b1000_0000; b = 8'b0000_0010;
// shift right logically
#10 op = 4'b 1001; a = 8'b0111_1111; b = 8'b0000_0010;
#10 op = 4'b 1001; a = 8'b1000_0000; b = 8'b0000_0010;
// shift left
#10 op = 4'b 1010; a = 8'b0111_1111; b = 8'b0000_0010;
#10 op = 4'b 1010; a = 8'b1000_0000; b = 8'b0000_0010;
// add
#10 op = 4'b 0000; a = 8'b0111_1111; b = 8'b0000_0001;
#10 op = 4'b 0000; a = 8'b1111_1111; b = 8'b0000_0001;
// sub
#10 op = 4'b 0001; a = 8'b0000_0001; b = 8'b0111_1111;
#10 op = 4'b 0001; a = 8'b0111_1111; b = 8'b0000_0001;
#10 op = 4'b 0001; a = 8'b0111_1111; b = 8'b1000_0000;
#10 op = 4'b 0001; a = 8'b1000_0000; b = 8'b0111_1111;
// and
#10 op = 4'b 0100; a = 8'b1010_1010; b = 8'b1111_1111;
// or
#10 op = 4'b 0101; a = 8'b1010_1010; b = 8'b1111_1111;
// xor
#10 op = 4'b 0111; a = 8'b1111_0000; b = 8'b1100_1100;
// not
#10 op = 4'b 0110; a = 8'b10101010; b = 8'b1111_1111;
```

Decoder

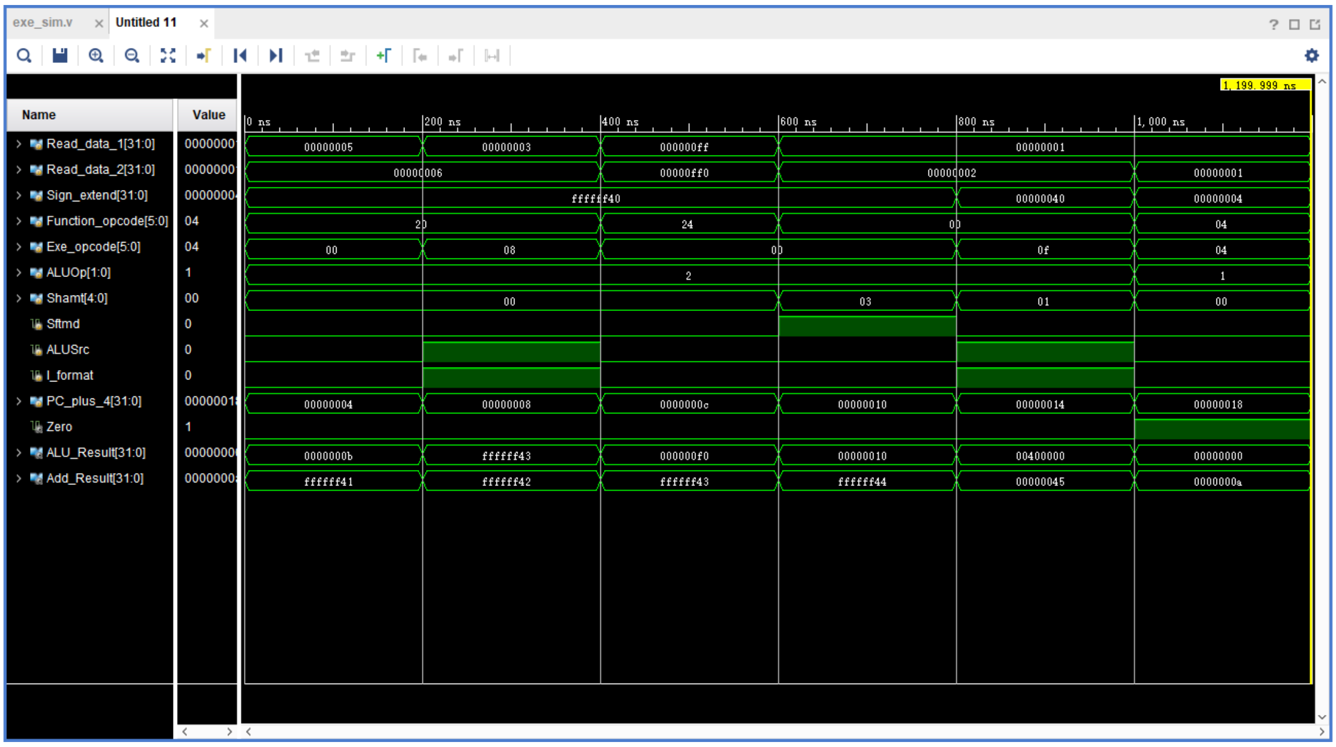

Untitled 6

```
Name                 Value
Instruction[31:0]    0c000000
read_data[31:0]      0000000
ALU_result[31:0]     0000000
Jal                  1
RegWrite             1
MemtoReg             0
RegDst               0
clock                1
reset                0
opcplus4[31:0]       0000001
read_data_1[31:0]    0000000
read_data_2[31:0]    0000000
Sign_extend[31:0]    0000000
register[0:31][31:0] 0000000
  [0][31:0]          0000000
  [1][31:0]          0000000
  [2][31:0]          0000000
  [3][31:0]          ffff803c
  [4][31:0]          0000000
  [5][31:0]          0000001
  [6][31:0]          0000007
  [7][31:0]          0000000
  [8][31:0]          0000000
```

```
#20    reset = 1'b0;
          #20    begin Instruction = 32'b001000_00111_00011_1000000000110111;
               //addi $3,$7,0X8037
                        read_data = 32'h00000000;
```

```
                ALU_result = 32'hFFFF803C;
                Jal = 1'b0;
                RegWrite = 1'b1;
                MemtoReg = 1'b0;
                RegDst = 1'b0;
                opcplus4 = 32'h00000008;
           end
      #20   begin Instruction = 32'b001100_00010_00100_1000000010010111;
         //andi $4,$2,0X8097
                       read_data = 32'h00000000;
                       ALU_result = 32'h00000002;
                       Jal = 1'b0;
                       RegWrite = 1'b1;
                       MemtoReg = 1'b0;
                       RegDst = 1'b0;
                       opcplus4 = 32'h0000000c;
            end
      #20   begin Instruction = 32'b000000_00000_00001_00101_00010_000000;
         //sll $5,$1,2
                               read_data = 32'h00000000;
                               ALU_result = 32'h00000004;
                               Jal = 1'b0;
                               RegWrite = 1'b1;
                               MemtoReg = 1'b0;
                               RegDst = 1'b1;
                               opcplus4 = 32'h00000010;
           end
      #20   begin Instruction = 32'b100011_00000_00110_0000000100000000;
         //LW $6,0(0X100)
                                      read_data = 32'h0000007B;
                                      ALU_result = 32'h00000054;
                                      Jal = 1'b0;
                                      RegWrite = 1'b1;
                                      MemtoReg = 1'b1;
                                      RegDst = 1'b0;
                                      opcplus4 = 32'h00000014;
           end
      #20   begin Instruction = 32'b000011_00000000000000000000000000;
         //JAL 0000
                                      read_data = 32'h00000000;
                                      ALU_result = 32'h00000004;
                                      Jal = 1'b1;
                                      RegWrite = 1'b1;
                                      MemtoReg = 1'b0;
                                      RegDst = 1'b0;
                                      opcplus4 = 32'h00000018;

           end
      #20 $finish;
```

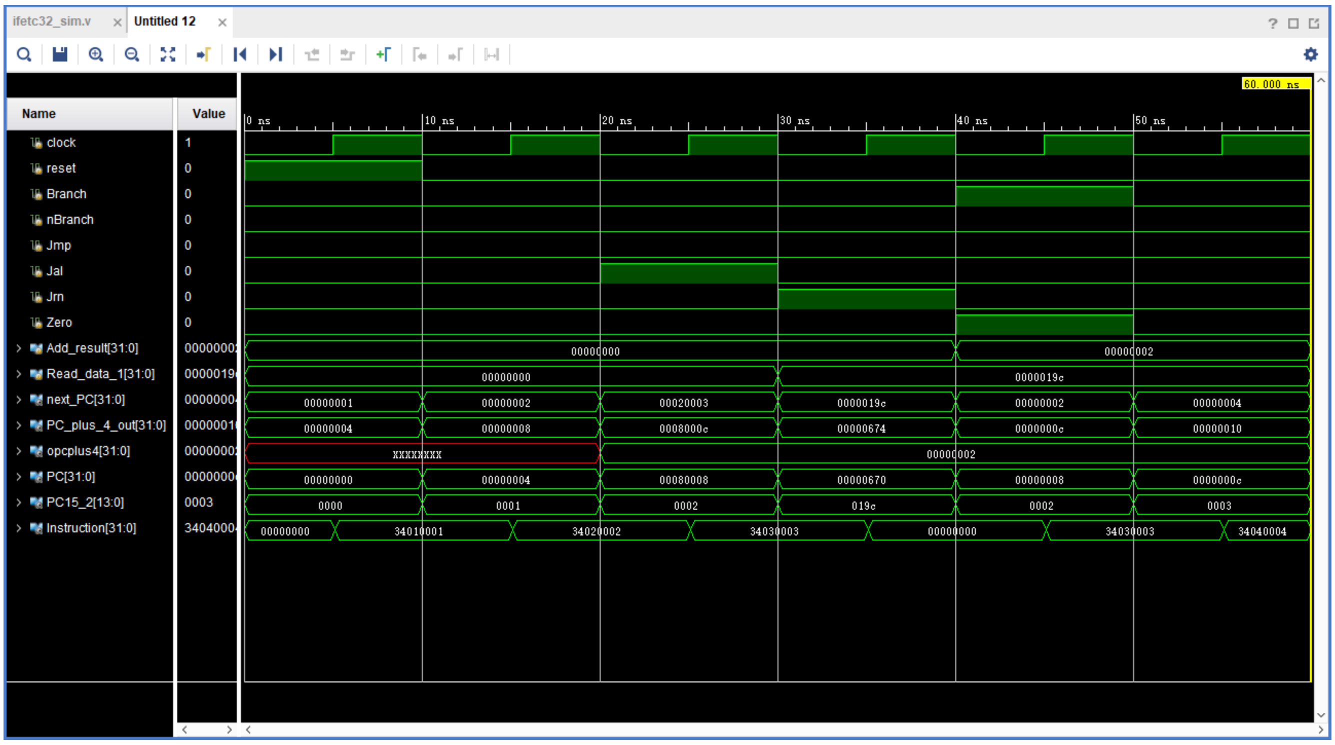
执行单元



```
#200 begin Exe_opcode = 6'b001000;  //addi
     Read_data_1 = 32'h00000003;     //r-form rs
     Read_data_2 = 32'h00000006;        //r-form rt
     Sign_extend = 32'hffffff40;
     Function_opcode = 6'b100000;       //addi
     ALUOp = 2'b10;
     Shamt = 5'b00000;
     Sftmd = 1'b0;
     ALUSrc = 1'b1;
     I_format = 1'b1;
     PC_plus_4 = 32'h00000008;
end
#200 begin Exe_opcode = 6'b000000;  //and
     Read_data_1 = 32'h000000ff;        //r-form rs
     Read_data_2 = 32'h00000ff0;        //r-form rt
     Sign_extend = 32'hffffff40;
     Function_opcode = 6'b100100;       //and
     ALUOp = 2'b10;
     Shamt = 5'b00000;
     Sftmd = 1'b0;
     ALUSrc = 1'b0;
     I_format = 1'b0;
     PC_plus_4 = 32'h0000000c;
end
#200 begin Exe_opcode = 6'b000000;  //sll
     Read_data_1 = 32'h00000001;        //r-form rs
     Read_data_2 = 32'h00000002;        //r-form rt
     Sign_extend = 32'hffffff40;
     Function_opcode = 6'b000000;       //sll
     ALUOp = 2'b10;
     Shamt = 5'b00011;
     Sftmd = 1'b1;
```

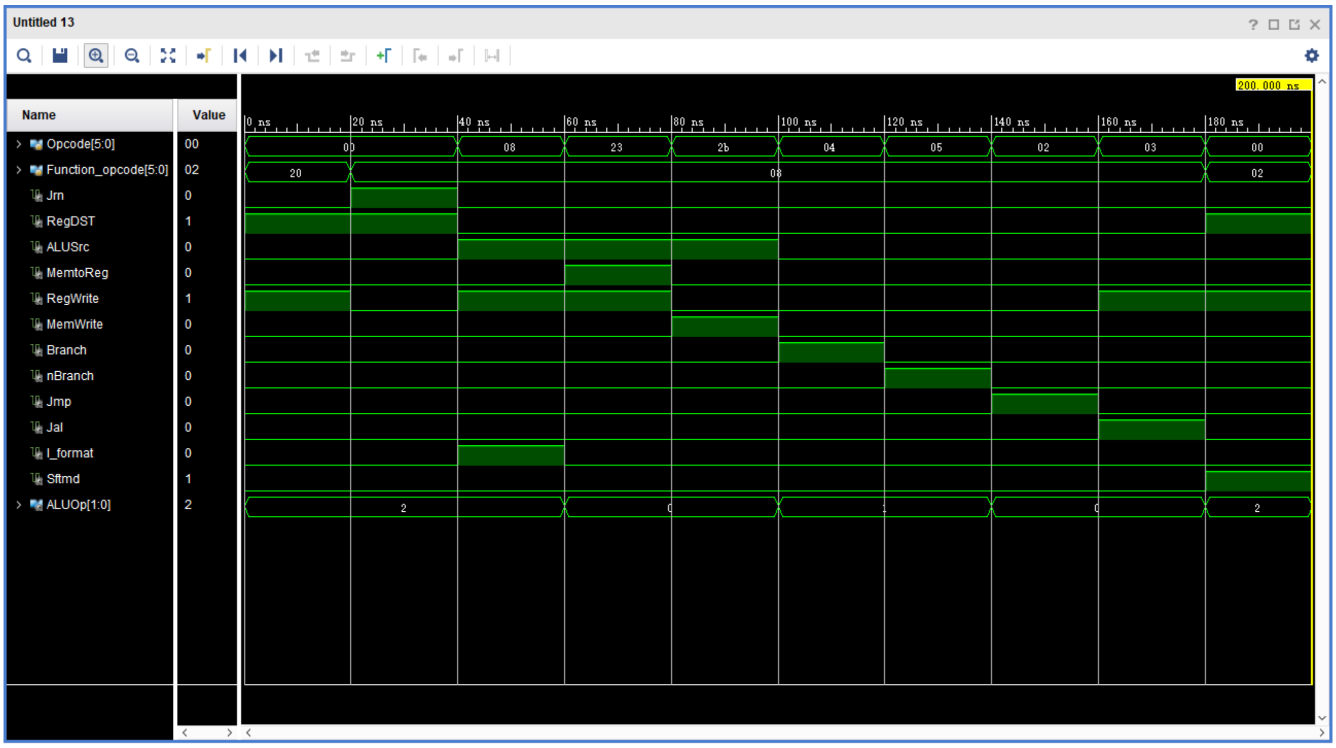
```
    ALUSrc = 1'b0;
    I_format = 1'b0;
    PC_plus_4 = 32'h00000010;
end
#200 begin Exe_opcode = 6'b001111;  // LUI
    Read_data_1 = 32'h00000001;        //r-form rs
    Read_data_2 = 32'h00000002;        //r-form rt
    Sign_extend = 32'h00000040;
    Function_opcode = 6'b000000;      //LUI
    ALUOp = 2'b10;
    Shamt = 5'b00001;
    Sftmd = 1'b0;
    ALUSrc = 1'b1;
    I_format = 1'b1;
    PC_plus_4 = 32'h00000014;
end
#200 begin Exe_opcode = 6'b000100;  // BEQ
    Read_data_1 = 32'h00000001;        //r-form rs
    Read_data_2 = 32'h00000001;        //r-form rt
    Sign_extend = 32'h00000004;
    Function_opcode = 6'b000100;      //LUI
    ALUOp = 2'b01;
    Shamt = 5'b00000;
    Sftmd = 1'b0;
    ALUSrc = 1'b0;
    I_format = 1'b0;
    PC_plus_4 = 32'h00000018;
end
#200 begin    // SLT
    Read_data_1 = 32'h00000001;        // r-form rs
    Read_data_2 = 32'h00000011;        // r-form rt
    Sign_extend = 32'h00000004;
    Function_opcode = 6'b101010;      // SLT 2a
    ALUOp = 2'b10;
    Shamt = 5'b00000;
    Sftmd = 1'b0;
    ALUSrc = 1'b0;
    I_format = 1'b0;
    PC_plus_4 = 32'h00000022;
end
#200 $finish;
```

取指单元



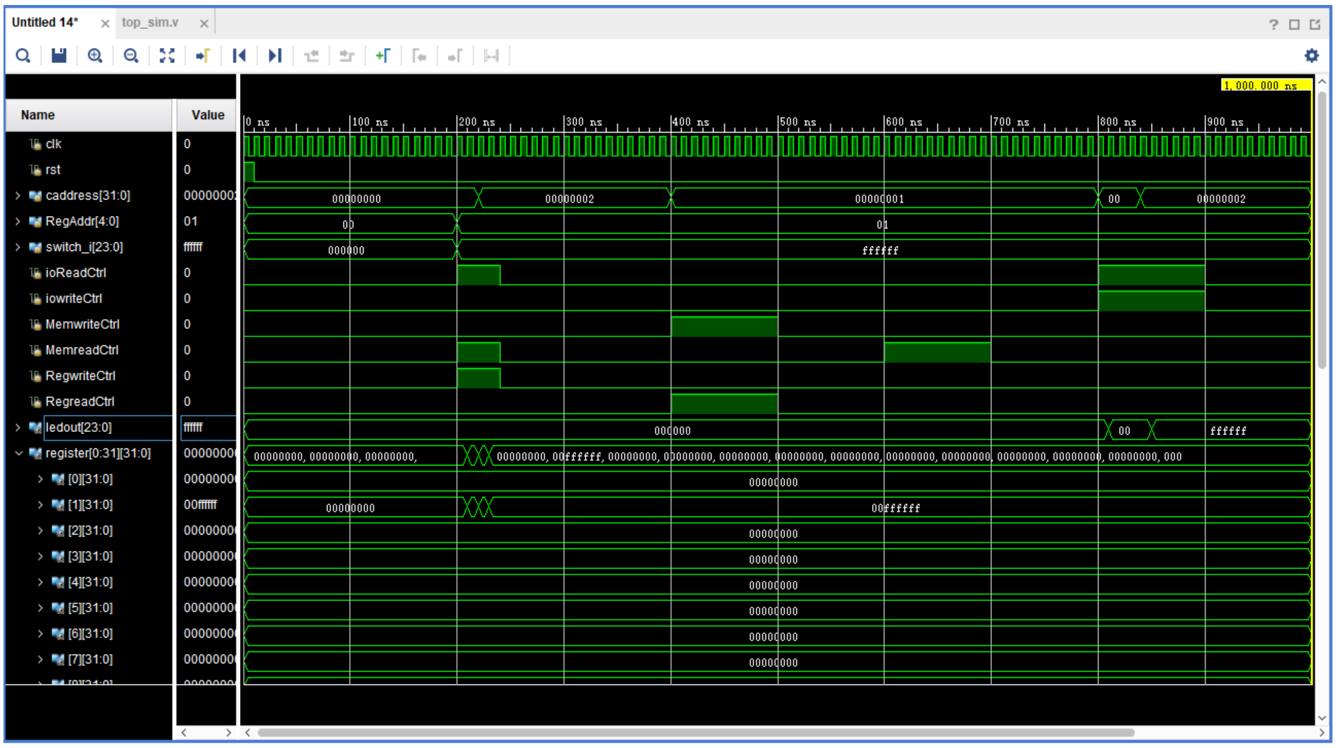
```
initial begin
     #10   reset = 1'b0;
     #10   Jal = 1;
     #10   begin Jrn = 1;Jal = 0; Read_data_1 = 32'h0000019c;end;
     #10   begin Jrn = 0;Branch = 1'b1; Zero = 1'b1; Add_result = 32'h00000002;end;
     #10   begin Branch = 1'b0; Zero = 1'b0; end;
     #10 $finish;
end
```

控制单元



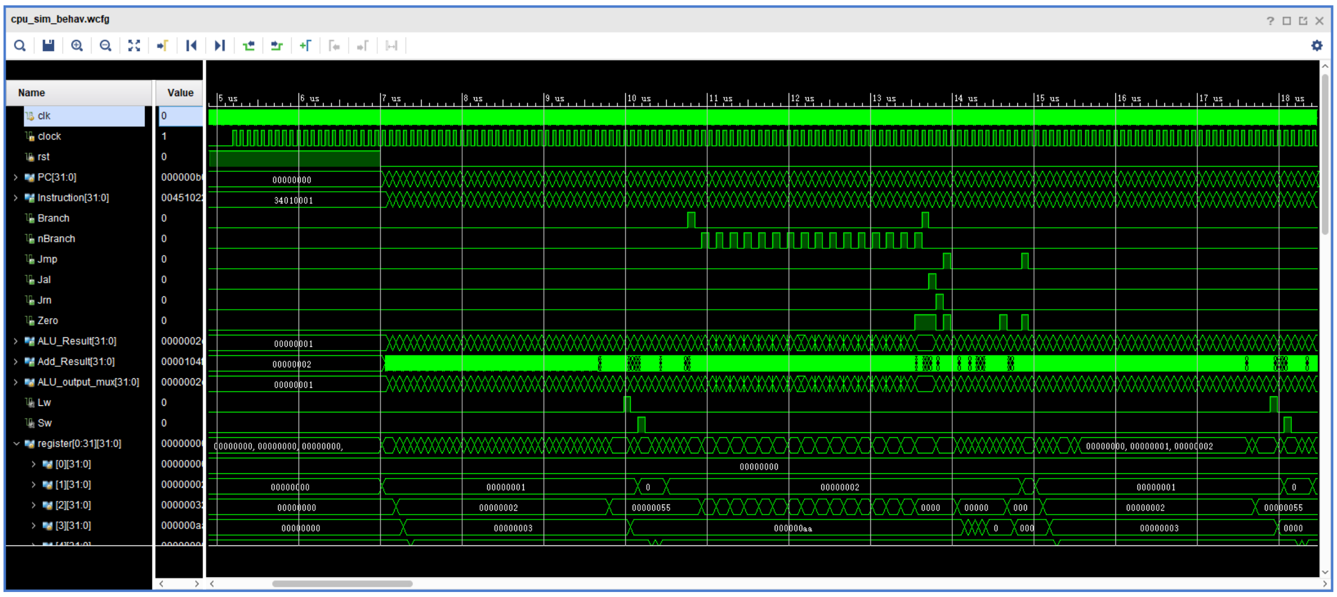
```
initial begin
    #20     Function_opcode  = 6'b001000;               // JR
    #20    Opcode = 6'b001000;                         //  ADDI
    #20    Opcode = 6'b100011;                         //  LW
    #20    Opcode = 6'b101011;                         //  SW
    #20    Opcode = 6'b000100;                         //  BEQ
    #20    Opcode = 6'b000101;                         //  BNE
    #20    Opcode = 6'b000010;                         //  JMP
    #20    Opcode = 6'b000011;                         //  JAL
    #20    begin Opcode = 6'b000000; Function_opcode  = 6'b000010; end;//  SRL
    #20 $finish;
end
```

MemoryIO



```
fork
    #10  begin // 取消复位信号，关闭所有控制信号
        rst = 1'b0;
        ioReadCtrl = 1'b0;
        iowriteCtrl = 1'b0;
        MemreadCtrl = 1'b0;
        MemwriteCtrl = 1'b0;
        RegreadCtrl = 1'b0;
        RegwriteCtrl = 1'b0;
    end
    #200 begin //  在200ns时读取 switchs 的数据，将其写入reg $1的低24bit;
        // 控制信号
        ioReadCtrl = 1'b1;  // 读取 switchs
        MemreadCtrl = 1'b1; // 读取 switchs
        RegwriteCtrl = 1'b1;// 写入寄存器
        // 输入信号
        switch_i = 24'b111111111111111111111111; // 拨码开关
        RegAddr = 5'b00001;     // 寄存器地址
        // 辅助信号
        caddress = 2'b00;     // 读取后16位
    end
    #220 caddress = 2'b10;    // 读取前8位
    #240 begin // 控制信号关闭
        ioReadCtrl = 1'b0;
        MemreadCtrl = 1'b0;
        RegwriteCtrl = 1'b0;
    end
    #400 begin  // 在400ns时读取$1上的数据，将其写入memory 地址为1的存储单元；
        // 控制信号
        RegreadCtrl = 1'b1; // 能够读取 register 的数据
        MemwriteCtrl = 1'b1;// 能够写入 memory
        // 输入信号
        RegAddr = 1'b1;     // 寄存器地址
        caddress = 32'b1;   // 写入地址
    end
    #500 begin // 控制信号关闭
        RegreadCtrl = 1'b0;
        MemwriteCtrl = 1'b0;
    end
    #600 begin // 在600ns读取memory 地址为1的存储单元的值
        MemreadCtrl = 1'b1;
    end
    #700 begin // 控制信号关闭
        MemreadCtrl = 1'b0;
    end
    #800 begin // 在800ns时将其写入到 led 作为输出;
        // 控制信号
        iowriteCtrl = 1'b1;
        ioReadCtrl = 1'b1;  // LEDCtrl 使能信号
        // 辅助信号
        caddress = 2'b00;     // 写入后16位
    end
    #840 caddress = 2'b10; // 写入前8位
    #900 begin // 控制信号关闭
        iowriteCtrl = 1'b0;
        ioReadCtrl = 1'b0;
    end
    #1000 $finish;
join
```

CPU



五、 实验分析（遇到的问题以及解决方案）

1. 创建RAM时不能勾选带使能信号 不然仿真的时候会一直写不进去
2. 给RAM增加IO接口时 需要把原来的`MemtoReg`改成`MemorIOtoReg` 而不是简单的新增
3. `IORead`/`IOWrite`/`MemRead`基本上就是照抄上面给的`MemWrite`示例 但是要稍微改一下判定条件
4. 注意到 Led和switch都是24bit 但是我们每次只能读取16bit 所以需要自己在仿真过程中手动拼接（读取/写入都是同理）
5. `ledaddr`/`switchaddr` 决定了是前8bit还是后16bit 可以直接用地址的最后两位 而不需要自己手动输入
6. `MemorIO`实际上就是一个转发器 决定到底是用IO还是用mem处理 （也可以把`ioread`看作一个输入设备的接口/父类） 最好在纸上整理好信号走向在开始写模拟
7. 模拟的时候 寄存器并不需要真的把之前的模块放进来 可以自己用一个`reg`模拟一下
8. 仿真CPU需要注意很多细节，由于没有部分指令分解的`wire`不存在，不推荐使用`block design`
9. CPU设计程序文件时，在汇编层级的延时处理需要结合波形图（跳转，算术运算等所需的时钟周期），确定最终的循环次数

六、 实验小结与反馈

最后一周太刺激了！！！

感觉最后`MemoryIO`的课件没有前面几个顺手，资料不是很全的感觉。

另外也许课件里的sim文件统一采用`fork` `join`会更便于观察？

清华（还是北大）的课件好像实现了带有分支预测的CPU，或许我们也可以试一试

检查作业的学助和老师辛苦了。