实验项目名称：Single-cycle CPU Design

一、 实验目的和要求

1. Understand the principles of CPU Controller and master methods of CPU Controller design
2. Understand the principles of Datapath and master methods of Datapath design
3. Understand the principles of Single-cycle CPU and master methods of Single-cycle CPU design
4. Master methods of program verification of CPU

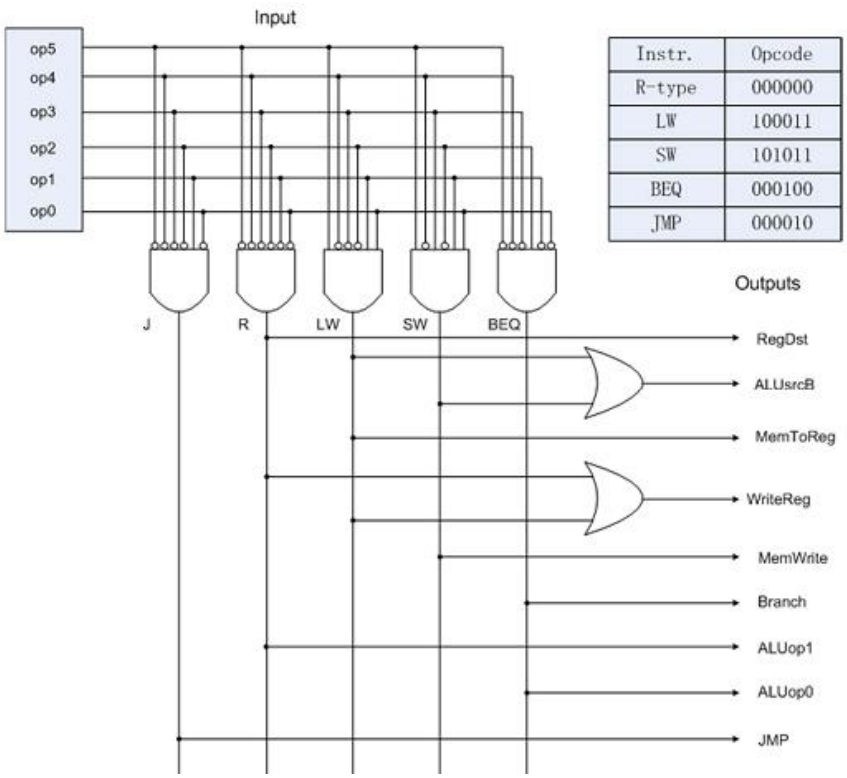
二、 实验内容和原理

1. CPU Controller

The following table shows the output of CPU Controller:

Signal	Meaning when signal is 1	Meaning when signal is 0
RegDst	rd	rt
ALUsrcB	Extended-Imm.	register
MemtoReg	Mem. output	ALU output
RegWrite	Write to Register file, the addr is up to RegDst, the data is up to MemtoReg	Not write to Register file
MemWrite	Write to Memory	Not write to memory
Branch	Branch addr.	Not branch
Jump	Jump addr.	Branch addr
ALUC[1:0]	ALU operation code	

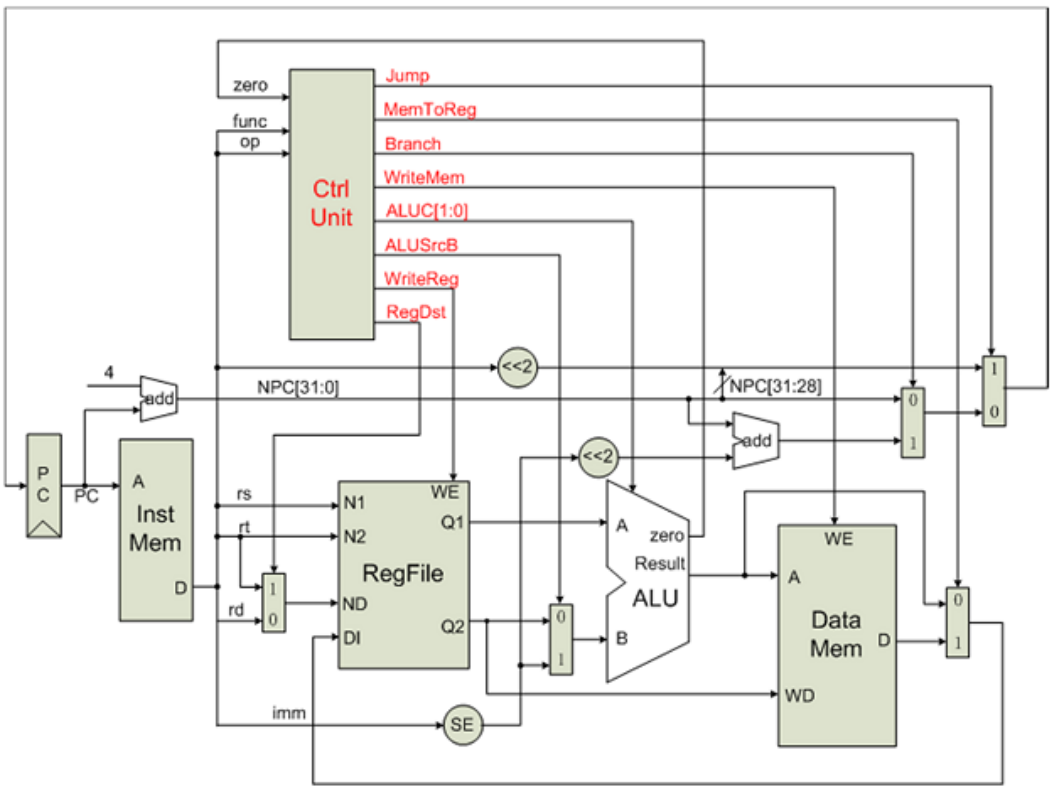
The following picture shows the principle of CPU Controller:



Instr.	Opcode
R-type	000000
LW	100011
SW	101011
BEQ	000100
JMP	000010

2. Single-cycle CPU

The following picture shows the datapath of Single-cycle CPU:



The basic units of Single-cycle CPU contains:

1) CPU Controller

2) ALU and ALU Controller

3) Register file

4) Instruction Mem. and Data Mem.

5) others：Register, adder, sign-extend Unit, shifter, multiplexor

3. Program for verification

calculate 1+2+3+ … + 10

MIPS code	Comment
lw $1, 0($0)	$1 = 1: step
lw $2, 4($0)	$2 = 0: sum
lw $4, 1($0)	$4 = 11: upper
add $3, $1, $2	$3 = 1: index
repeat: add $2, $3, $2	do{$2=$2+$3
add $3, $3, $1	$3=$3+1
beq $3, $4, out	}while($3==$4) ;
j repeat;	
out: sw $2, 2($0)	save $2 to 2($0)

The corresponding content of Instruction Mem coe file:

```
MEMORY_INITIALIZATION_RADIX=2;
MEMORY_INITIALIZATION_VECTOR=
10001100000000010000000000000000,
10001100000000010000000000000100,
10001100000001000000000000000001,
00000000001000100001100000100000,
00000000011000100001000000100000,
00000000011000100001100000100000,
00010000011001000000000000000001,
00001000000000000000000000000100,
10101100000000010000000000000010,
00000000000000000000000000000000,
…
00000000000000000000000000000000
```

The corresponding content of Data Mem coe file:

```
MEMORY_INITIALIZATION_RADIX=2;
MEMORY_INITIALIZATION_VECTOR=
00000000000000000000000000000001,
00000000000000000000000000001011,
00000000000000000000000000000000,
00000000000000000000000000000000,
00000000000000000000000000000000,
…
00000000000000000000000000000000
```

三、 实验过程和数据记录

1. CPU Controller

The Verilog code is as follows:

```
module ctltop(clk, OP, RegDst, ALUsrc, ALUop, MemtoReg,RegWrite, MemRead,
MemWrite, Branch, Jump);
    input clk;        //clock signal
    input [5:0] OP;   //the first six bits of the instruction
    //output control signals
    output RegDst;
    output ALUsrc;
    output [1:0] ALUop;
    output MemtoReg;
    output RegWrite;
    output MemRead;
    output MemWrite;
    output Branch;
    output Jump;
    assign RegDst = ~OP[5] & ~OP[4] & ~OP[3] & ~OP[2] & ~OP[1] & ~OP[0];
    assign MemtoReg =OP[5] & ~OP[4] & ~OP[3] & ~OP[2] & OP[1] & OP[0];
    assign RegWrite =~OP[4] & ~OP[3] & ~OP[2];
    assign ALUsrc =OP[5] & ~OP[4] & ~OP[2] & OP[1] & OP[0];
    assign ALUop[1] =~OP[5] & ~OP[4] & ~OP[3] & ~OP[2] & ~OP[1] & ~OP[0];
    assign ALUop[0] =~OP[5] & ~OP[4] & ~OP[3] & OP[2] & ~OP[1] & ~OP[0];
    assign MemRead =OP[5] & ~OP[4] & ~OP[3] & ~OP[2] & OP[1] & OP[0];
    assign MemWrite =OP[5] & ~OP[4] & OP[3] & ~OP[2] & OP[1] & OP[0];
    assign Branch =~OP[5] & ~OP[4] & ~OP[3] & OP[2] & ~OP[1] & ~OP[0];
    assign Jump =~OP[5] & ~OP[4] & ~OP[3] & ~OP[2] & OP[1] & ~OP[0];
endmodule
```

2. Single-cycle CPU

The content of the file “macro.vh” :

```
//macro.vh for macro definitions
//ALU CONTROL CODES
`define ALU_AND      3'b000
`define ALU_OR       3'b001
`define ALU_ADD      3'b010
`define ALU_SUB      3'b110
`define ALU_SLT      3'b111
//INSTRUCTION TYPE
`define INSTR_LW     6'b100011
`define INSTR_SW     6'b101011
`define INSTR_BEQ    6'b000100
`define INSTR_RTYPE  6'b000000
`define INSTR_JUMP   6'b000010
//R-TYPE FUNCTION CODE
`define RTYPE_ADD    6'b100000
`define RTYPE_SUB    6'b100010
`define RTYPE_SLT    6'b101010
`define RTYPE_AND    6'b100100
`define RTYPE_OR     6'b100101
`define RTYPE_XOR    6'b100110
`define RTYPE_NOR    6'b100111
`define RTYPE_ADDU   6'b100001
`define RTYPE_SLTU   6'b101011
```

The Verilog code:

```
module top(input wire CCLK, BTN3, BTN2, input wire [3:0]SW, output wire
```

```
LED, LCDE, LCDRS, LCDRW, output wire [3:0]LCDDAT);
    wire [8:0] pc_out;
    wire [8:0] pc_in;
    wire [8:0] pc_plus_4;
    wire [4:0] reg3_out;
    wire [31:0] wdata_out;
    wire [31:0] instr_out;
    //CPU control signals
    wire regwrite;
    wire alusrc;
    wire [1:0] aluop;
    wire memwrite;
    wire memtoreg;
    wire memread;
    wire branch;
    wire jump;
    wire regdst;

    wire [31:0] reg1_dat;
    wire [31:0] reg2_dat;
    wire [2:0] alu_ctrl;
    wire [31:0] signext_out;
    wire [31:0] mux_to_alu;
    wire [31:0] alu_out;
    wire alu_zero;
    wire [31:0] mem_dat_out;
    wire and_out;
    wire [31:0] branch_addr_out;
    wire [31:0] branch_mux_out;
    wire [31:0] gpr_disp_out;   //the content of the selected register
    reg [15:0] clk_cnt;         //clock count singal
    //temp counter to assist time-dividing the on-board 50MHz clock
    reg [15:0] tmp_cnt;
    //the signal of time-dividing the on-board 50MHz clock
    reg [1:0] disp_clk_cnt;
    reg [255:0] strdata;        //data displayed in the LCD
    reg [3:0] SW_old;           //the previous state of slide buttons
    //flag used to indicate whether the state of slide button is changed
    reg cls;
    //anti-jitter push button signal of BTN2 and BTN3,correspondingly
    wire button3,button2;
    wire [3:0] lcdd;
    wire rslcd, rwlcd, elcd;
    assign LCDDAT[3]=lcdd[3];
    assign LCDDAT[2]=lcdd[2];
    assign LCDDAT[1]=lcdd[1];
    assign LCDDAT[0]=lcdd[0];
    assign LCDRS=rslcd;
    assign LCDRW=rwlcd;
    assign LCDE=elcd;
    //initialize display data
    initial begin
        strdata = "01234567 00 00  0123          ";
    end
    //display module, show the content of 'strdata' in the LCD
    display M0 (CCLK, cls, strdata, rslcd, rwlcd, elcd, lcdd);
    //pbdebounce module, output the anti-jitter push button signal
    pbdebounce M1(CCLK, BTN2, button2);
    pbdebounce M2(CCLK, BTN3, button3);
    //assign the anti-jitter push button signal of button3 to LED
    assign LED=button3;
```

```
    //display the result in the LCD using sexadecimal notion
    always @(posedge CCLK) begin
        if ((button3 == 1'b1) || (button2 == 1'b1)) begin
            //first line 8 4-bit Instrution
            if(instr_out[31:28] < 10)
                strdata[255:248] = 8'h30 + instr_out[31:28];
            else
                strdata[255:248] = 8'h61 + instr_out[31:28] - 10;
            if(instr_out[27:24] < 10)
                strdata[247:240] = 8'h30 + instr_out[27:24];
            else
                strdata[247:240] = 8'h61 + instr_out[27:24] - 10;
            if(instr_out[23:20] < 10)
                strdata[239:232] = 8'h30 + instr_out[23:20];
            else
                strdata[239:232] = 8'h61 + instr_out[23:20] - 10;
            if(instr_out[19:16] < 10)
                strdata[231:224] = 8'h30 + instr_out[19:16];
            else
                strdata[231:224] = 8'h61 + instr_out[19:16] - 10;
            if(instr_out[15:12] < 10)
                strdata[223:216] = 8'h30 + instr_out[15:12];
            else
                strdata[223:216] = 8'h61 + instr_out[15:12] - 10;
            if(instr_out[11:8] < 10)
                strdata[215:208] = 8'h30 + instr_out[11:8];
            else
                strdata[215:208] = 8'h61 + instr_out[11:8] - 10;
            if(instr_out[7:4] < 10)
                strdata[207:200] = 8'h30 + instr_out[7:4];
            else
                strdata[207:200] = 8'h61 + instr_out[7:4] - 10;
            if(instr_out[3:0] < 10)
                strdata[199:192] = 8'h30 + instr_out[3:0];
            else
                strdata[199:192] = 8'h61 + instr_out[3:0] - 10;
            //space
            //strdata[191:184] = " ";
            //2 4-bit CLK
            if(clk_cnt[7:4] < 10)
                strdata[183:176] = 8'h30 + clk_cnt[7:4];
            else
                strdata[183:176] = 8'h61 + clk_cnt[7:4] - 10;
            if(clk_cnt[3:0] < 10)
                strdata[175:168] = 8'h30 + clk_cnt[3:0];
            else
                strdata[175:168] = 8'h61 + clk_cnt[3:0] - 10;
            //space
            //strdata[167:160] = " ";
            //2 4-bit PC
            if(pc_out[7:4] < 10)
                strdata[159:152] = 8'h30 + pc_out[7:4];
            else
                strdata[159:152] = 8'h61 + pc_out[7:4] - 10;
            if(pc_out[3:0] < 10)
                strdata[151:144] = 8'h30 + pc_out[3:0];
            else
                strdata[151:144] = 8'h61 + pc_out[3:0] - 10;
        end
        //update the selected register content if the state of slide buttons
is changed
```

```
       if((button3 == 1'b1) || (button2 == 1'b1) || (SW_old != SW)) begin
          if(gpr_disp_out[15:12] < 10)
             strdata[127:120] = 8'h30 + gpr_disp_out[15:12];
          else
             strdata[127:120] = 8'h61 + gpr_disp_out[15:12] - 10;
          if(gpr_disp_out[11:8] < 10)
             strdata[119:112] = 8'h30 + gpr_disp_out[11:8];
          else
             strdata[119:112] = 8'h61 + gpr_disp_out[11:8] - 10;
          if(gpr_disp_out[7:4] < 10)
             strdata[111:104] = 8'h30 + gpr_disp_out[7:4];
          else
             strdata[111:104] = 8'h61 + gpr_disp_out[7:4] - 10;
          if(gpr_disp_out[3:0] < 10)
             strdata[103:96] = 8'h30 + gpr_disp_out[3:0];
          else
             strdata[103:96] = 8'h61 + gpr_disp_out[3:0] - 10;
          SW_old = SW;  //update the previous state of slide buttons
          cls = 1;      //set the flag
       end
       else
          cls = 0;      //reset the flag
    end
    //update the clock count singal
    always @(posedge button3 or posedge button2) begin
       if (button2 == 1'b1) begin
          clk_cnt = 16'h0000;     //reset the clock count
       end
       else begin
          clk_cnt = clk_cnt + 1;
       end
    end
    //time-divide the on-board 50MHz clock
    always @(posedge CCLK or posedge button2) begin
    if (button2==1) begin       //reset
        disp_clk_cnt=2'b00;
        tmp_cnt=0;
      end
     else begin
        tmp_cnt=tmp_cnt+1;
     if (tmp_cnt==16'h0000)
        disp_clk_cnt=disp_clk_cnt+1;
      end
    end

    assign o_instr = instr_out[31:26];
    assign and_out = alu_zero & branch;
    assign pc_in = jump ? instr_out[8:0] : branch_mux_out[8:0];

    single_pc  x_single_pc(button3,button2,pc_in,pc_out);
    c_instr_mem   x_c_instr_mem(pc_out,CCLK,instr_out);
    single_pc_plus4   x_single_pc_plus4(pc_out,pc_plus_4);
    single_mux5
    x_single_mux5(instr_out[20:16],instr_out[15:11],regdst,reg3_out);
    single_gpr
    x_single_gpr(button2,button3,instr_out[25:21],instr_out[20:16],{1
'b0,SW},reg3_out,wdata_out,regwrite,reg1_dat,reg2_dat,gpr_disp_out);
    single_aluc   x_single_aluc(aluop,instr_out[5:0],alu_ctrl);
    single_signext    x_single_signext(instr_out[15:0],signext_out);
    single_mux32
    x_single_mux32(reg2_dat,signext_out,alusrc,mux_to_alu);
```

```
    single_alu
x_single_alu(reg1_dat,mux_to_alu,alu_ctrl,alu_zero,alu_out);
    c_dat_mem
    x_c_dat_mem(alu_out[8:0],CCLK,reg2_dat,mem_dat_out,memwrite);
    single_mux32
    x_single_mux32_2(alu_out,mem_dat_out,memtoreg,wdata_out);
    single_add
    x_single_add(signext_out,{{23'b0},pc_plus_4},branch_addr_out);
    single_mux32
    x_single_mux32_3({{23'b0},pc_plus_4},branch_addr_out,and_out,bran
ch_mux_out);
    single_ctl
    x_single_ctl(button2,instr_out[31:26],regdst,jump,branch,memread,
memtoreg,aluop,memwrite,alusrc,regwrite);
endmodule
//PC register module which is used to store the instruction address
module single_pc(clk,rst,ipc,opc);
       input clk;
       input rst;
       input [8:0] ipc;
       output [8:0] opc;
       wire [8:0] tpc;
       dff x_dff(clk,ipc,tpc);
       sel x_sel(rst,tpc,opc);
endmodule
//D flip-flop module
module dff(clk,d,q);
       input clk;
       input [8:0] d;
       output [8:0] q;
       reg [8:0] q;
       initial begin
           q<=0;
       end
       always @ (posedge clk) begin
           q<=d;
       end
endmodule
//Module ‘sel’ is used to reset PC when signal ‘rst’ is high
module sel(rst,ipc,opc);
       input rst;
       input [8:0] ipc;
       output [8:0] opc;
       assign opc = rst ? 9'b111111111 : ipc;
endmodule
//32-bit adder module
module single_add(op1,op2,out);
          input [31:0] op1;
          input [31:0] op2;
          output [31:0] out;
          assign out = op1 + op2;
endmodule
//pc+4 module
module single_pc_plus4(ipc,opc);
    input [8:0] ipc;
    output [8:0] opc;
    assign opc[8:0] = ipc[8:0] + 1;
endmodule
//16-32 sign extension module
module single_signext(in, out);
    input [15:0] in;
```

```
    output [31:0] out;
    reg [31:0] out;
    always @(in) begin
           if(in[15] == 1'b1)
       out <= {{16'b1111111111111111},in};
           else
       out <= {{16'b0000000000000000},in};
    end
endmodule
//32-bit 2-1 multiplexer
module single_mux32(Ai,Bi,sel,out);
       input [31:0] Ai;
       input [31:0] Bi;
       input sel;
       output [31:0] out;
       assign out[31:0] = (sel == 1) ? (Bi[31:0]) : (Ai[31:0]);
endmodule
//5-bit 2-1 multiplexer
module single_mux5(Ai,Bi,sel,out);
        input [4:0] Ai;
       input [4:0] Bi;
       input sel;
       output [4:0] out;
       assign out[4:0] = (sel == 1) ? (Bi[4:0]) : (Ai[4:0]);
endmodule
//ALU controller module
`include "macro.vh"
module single_aluc(aluop, func, aluc);
    input [1:0] aluop;
    input [5:0] func;
    output [2:0] aluc;
    reg [2:0] aluc;
     always @(aluop or func) begin
        case (aluop)
            2'b00: begin
                aluc = `ALU_ADD;         //lw | sw
            end
            2'b01: begin
                aluc = `ALU_SUB;         //beq
            end
            2'b10: begin
                case (func)
                  `RTYPE_ADD:  aluc = `ALU_ADD;
                  `RTYPE_SUB:  aluc = `ALU_SUB;
                  `RTYPE_SLT:  aluc = `ALU_SLT;
                  `RTYPE_AND:  aluc = `ALU_AND;
                  `RTYPE_OR:  aluc = `ALU_OR;
                    default: aluc = 3'b000;
                endcase
            end
            default: begin
                aluc=3'b000;
            end
        endcase
    end
endmodule
//ALU module
`include "macro.vh"
module single_alu(i_r,i_s,i_aluc,o_zf,o_alu);
    input [31:0] i_r;        //i_r: r input
    input [31:0] i_s;        //i_s: s input
```

```
    input [2:0] i_aluc;      //i_aluc: ctrl input
    output o_zf;             //o_zf: zero flag output
    output [31:0] o_alu;     //o_alu: alu result output
    reg o_zf;
    reg [31:0] o_alu;
    always @(i_aluc or i_r or i_s) begin
        case (i_aluc)
            `ALU_AND: begin
               o_zf = 0;
               o_alu = i_r & i_s;
            end
            `ALU_OR: begin
               o_zf = 0;
               o_alu = i_r | i_s;
            end
            `ALU_ADD: begin
               o_zf = 0;
               o_alu = i_r + i_s;
            end
            `ALU_SUB: begin
               o_alu = i_r - i_s;
               o_zf  = (o_alu == 0);
            end
            `ALU_SLT: begin
               o_zf = 0;
              if (i_s < i_r)
                  o_alu = 1;
               else
                 o_alu = 0;
            end
            default: begin
               o_alu = 0;
               o_zf = 0;
            end
        endcase
    end
endmodule
//GPR(General Purpose Registers) module
module single_gpr(
    rst,       //reset
    clk,       //clock
    i_adr1,    //register index 1
    i_adr2,    //register index 2
    i_adr3,    //register index 3
    i_wreg,    //register to write
    i_wdata,   //data to write
    i_wen,     //write enable
    o_op1,     //read data1, out
    o_op2,     //read data2, out
    o_op3      //read data3, out
);

     input clk;
     input rst;
     input [4:0] i_adr1;
     input [4:0] i_adr2;
     input [4:0] i_adr3;
     input [31:0] i_wdata;
     input [4:0] i_wreg;
     input i_wen;
     output [31:0] o_op1;
```

```
    output [31:0] o_op2;
    output [31:0] o_op3;
    reg [31:0] mem[31:0];
       assign o_op1 = mem[i_adr1];
       assign o_op2 = mem[i_adr2];
       assign o_op3 = mem[i_adr3];
    always @(posedge clk or posedge rst) begin
       if (rst == 1) begin     //reset, clear the content of registers
            mem[0] <= 32'h00000000;
            mem[1] <= 32'h00000000;
            mem[2] <= 32'h00000000;
            mem[3] <= 32'h00000000;
            mem[4] <= 32'h00000000;
       end
       else if (i_wen) begin
       mem[i_wreg] <= (i_wreg == 5'b00000) ? 32'h00000000 : i_wdata;
       end
    end
endmodule
//single-cycle CPU Controller
module single_ctl(clk, OP, RegDst, Jump,  Branch, MemRead, MemtoReg,
ALUop, MemWrite,ALUsrc, RegWrite);
   input clk;        //clock signal
   input [5:0] OP;   //the first six bits of the instruction
   //output control signals
   output RegDst;
   output ALUsrc;
   output [1:0] ALUop;
   output MemtoReg;
   output RegWrite;
   output MemRead;
   output MemWrite;
   output Branch;
   output Jump;
   assign RegDst = ~OP[5] & ~OP[4] & ~OP[3] & ~OP[2] & ~OP[1] & ~OP[0];
   assign MemtoReg =OP[5] & ~OP[4] & ~OP[3] & ~OP[2] & OP[1] & OP[0];
   assign RegWrite =~OP[4] & ~OP[3] & ~OP[2];
   assign ALUsrc =OP[5] & ~OP[4] & ~OP[2] & OP[1] & OP[0];
   assign ALUop[1] =~OP[5] & ~OP[4] & ~OP[3] & ~OP[2] & ~OP[1] & ~OP[0];
   assign ALUop[0] =~OP[5] & ~OP[4] & ~OP[3] & OP[2] & ~OP[1] & ~OP[0];
   assign MemRead =OP[5] & ~OP[4] & ~OP[3] & ~OP[2] & OP[1] & OP[0];
   assign MemWrite =OP[5] & ~OP[4] & OP[3] & ~OP[2] & OP[1] & OP[0];
   assign Branch =~OP[5] & ~OP[4] & ~OP[3] & OP[2] & ~OP[1] & ~OP[0];
   assign Jump =~OP[5] & ~OP[4] & ~OP[3] & ~OP[2] & OP[1] & ~OP[0];
endmodule
```

四、 实验结果分析

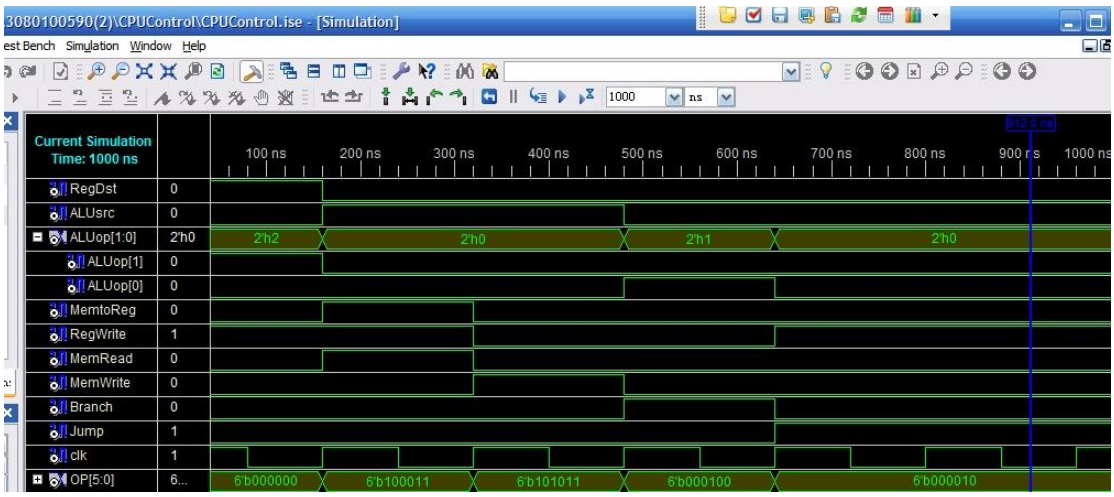
1. CPU Controller

The code of the test fixture file:

```
module ctltest_v;
   // Inputs
   reg clk;
   reg [5:0] OP;
   // Outputs
   wire RegDst;
   wire ALUsrc;
   wire [1:0] ALUop;
   wire MemtoReg;
   wire RegWrite;
```

```
    wire MemRead;
    wire MemWrite;
    wire Branch;
    wire Jump;
    // Instantiate the Unit Under Test (UUT)
    ctltop uut (
        .clk(clk),
        .OP(OP),
        .RegDst(RegDst),
        .ALUsrc(ALUsrc),
        .ALUop(ALUop),
        .MemtoReg(MemtoReg),
        .RegWrite(RegWrite),
        .MemRead(MemRead),
        .MemWrite(MemWrite),
        .Branch(Branch),
        .Jump(Jump)
    );
    always begin
        clk =1;
        #80;
        clk =0;
        #80;
    end
    initial begin
        OP=6'b000000;            //R-type
        #160   OP=6'b100011;     //lw
        #160   OP=6'b101011;     //sw
        #160   OP=6'b000100;     //beq
        #160   OP=6'b000010;     //jmp
        #1000  $dumpflush;
        $stop;
    end
endmodule
```

The following picture is the result of simulation:



The result indicates that the code implements the required functions correctly.

2. Single-cycle CPU

The following table is the detail result of the execution (Press the west button to step, press the south button to reset and set the state of slide buttons to decide the address

of register):

Clock count	PC	$1	$2	$3	$4	Instruction
0	0	0	0	0	0	lw $1, 0($0)
1	1	1	0	0	0	lw $2, 4($0)
2	2	1	0	0	0	lw $4, 1($0)
3	3	1	0	0	b	add $3, $1, $2
4	4	1	0	1	b	repeat: add $2, $3, $2
5	5	1	1	1	b	add $3, $3, $1
6	6	1	1	2	b	beq $3, $4, out
7	7	1	1	2	b	j repeat
8	4	1	1	2	b	repeat: add $2, $3, $2
9	5	1	3	2	b	add $3, $3, $1
a	6	1	3	3	b	beq $3, $4, out
b	7	1	3	3	b	j repeat
c	4	1	3	3	b	repeat: add $2, $3, $2
d	5	1	6	3	b	add $3, $3,

						$1
e	6	1	6	4	b	beq $3, $4, out
f	7	1	6	4	b	j repeat
10	4	1	6	4	b	repeat: add $2, $3, $2
11	5	1	a	4	b	add $3, $3, $1
12	6	1	a	5	b	beq $3, $4, out
13	7	1	a	5	b	j repeat
14	4	1	a	5	b	repeat: add $2, $3, $2
15	5	1	f	5	b	add $3, $3, $1
16	6	1	f	6	b	beq $3, $4, out
17	7	1	f	6	b	j repeat
18	4	1	f	6	b	repeat: add $2, $3, $2
19	5	1	15	6	b	add $3, $3, $1
1a	6	1	15	7	b	beq $3, $4, out

1b	7	1	15	7	b	j repeat
1c	4	1	15	7	b	repeat: add $2, $3, $2
1d	5	1	1c	7	b	add $3, $3, $1
1e	6	1	1c	8	b	beq $3, $4, out
1f	7	1	1c	8	b	j repeat
20	4	1	1c	8	b	repeat: add $2, $3, $2
21	5	1	24	8	b	add $3, $3, $1
22	6	1	24	9	b	beq $3, $4, out
23	7	1	24	9	b	j repeat
24	4	1	24	9	b	repeat: add $2, $3, $2
25	5	1	2d	9	b	add $3, $3, $1
26	6	1	2d	a	b	beq $3, $4, out
27	7	1	2d	a	b	j repeat
28	4	1	2d	a	b	repeat: add $2, $3, $2

29	5	1	37	a	b	add $3, $3, $1
2a	6	1	37	b	b	j repeat
2b	8	1	37	b	b	out: sw $2, 2($0)

The final result of calculation is 37 in hexadecimal notion, which equals to 55 in decimal notion.

The result indicates that the code implements the required functions correctly.

五、 讨论与心得

The traditional method to implement displaying in hexadecimal notion results code redundancy, like the following example:

```
always @(posedge CCLK) begin
    if ((button3 == 1'b1) || (button2 == 1'b1)) begin
        //first line 8 4-bit Instrution
        if(instr_out[31:28] < 10)
            strdata[255:248] = 8'h30 + instr_out[31:28];
        else
            strdata[255:248] = 8'h61 + instr_out[31:28] - 10;
        if(instr_out[27:24] < 10)
            strdata[247:240] = 8'h30 + instr_out[27:24];
        else
            strdata[247:240] = 8'h61 + instr_out[27:24] - 10;
        if(instr_out[23:20] < 10)
            strdata[239:232] = 8'h30 + instr_out[23:20];
        else
            strdata[239:232] = 8'h61 + instr_out[23:20] - 10;
        if(instr_out[19:16] < 10)
            strdata[231:224] = 8'h30 + instr_out[19:16];
        else
            strdata[231:224] = 8'h61 + instr_out[19:16] - 10;
        if(instr_out[15:12] < 10)
            strdata[223:216] = 8'h30 + instr_out[15:12];
        else
            strdata[223:216] = 8'h61 + instr_out[15:12] - 10;
        if(instr_out[11:8] < 10)
            strdata[215:208] = 8'h30 + instr_out[11:8];
        else
            strdata[215:208] = 8'h61 + instr_out[11:8] - 10;
        if(instr_out[7:4] < 10)
            strdata[207:200] = 8'h30 + instr_out[7:4];
        else
            strdata[207:200] = 8'h61 + instr_out[7:4] - 10;
        if(instr_out[3:0] < 10)
            strdata[199:192] = 8'h30 + instr_out[3:0];
        else
            strdata[199:192] = 8'h61 + instr_out[3:0] - 10;
    /*…*/
    end
/*…*/
```

```
    end
```

However, with the help of the generate statement which is provided by Verilog-2001 Standard, we can reduce such code redundancy. Here is the improved version of the code above:

```
parameter base_width = 192;
genvar j;
generate
for(j=0; j<8; j++)
begin: unit
    always @(posedge CCLK) begin
        if ((button3 == 1'b1) || (button2 == 1'b1)) begin
            if(instr_out[j*4+:4] < 10)
                strdata[base_width+j*8+:8] = 8'h30 + instr_out[j*4+:4];
            else
            strdata[base_width+j*8+:8] = 8'h61 + instr_out[j*4+:4] - 10;
        end
    end
end
endgenerate
```

Note that the representation like `instr_out[j*4+:4]` is also a new feather that Verilog-2001 Standard supports:

Indexed vector part selects

[*base_expr* **+:** *width_expr*] //positive offset

[*base_expr* -: *width_expr*] //negative offset

Reference：《The IEEE Verilog 1364-2001 Standard; What's New and Why You Need It》Stuart Sutherland, Sutherland HDL, Inc.