COD_LAB5 多周期 MIPS-CPU

黄业琦 PB17000144

May 7, 2019

Contents

1　实验目的

设计实现多周期 MIPS-CPU，可执行如下指令：

- add, sub, and, or, xor, nor, slt
- addi, andi, ori, xori, slti
- lw, sw
- beq, bne, j

数据通路和控制单元 (状态图) 参见后页，其中寄存器堆中 R0 内容恒定为 0，存储器容量为 256x32 位。

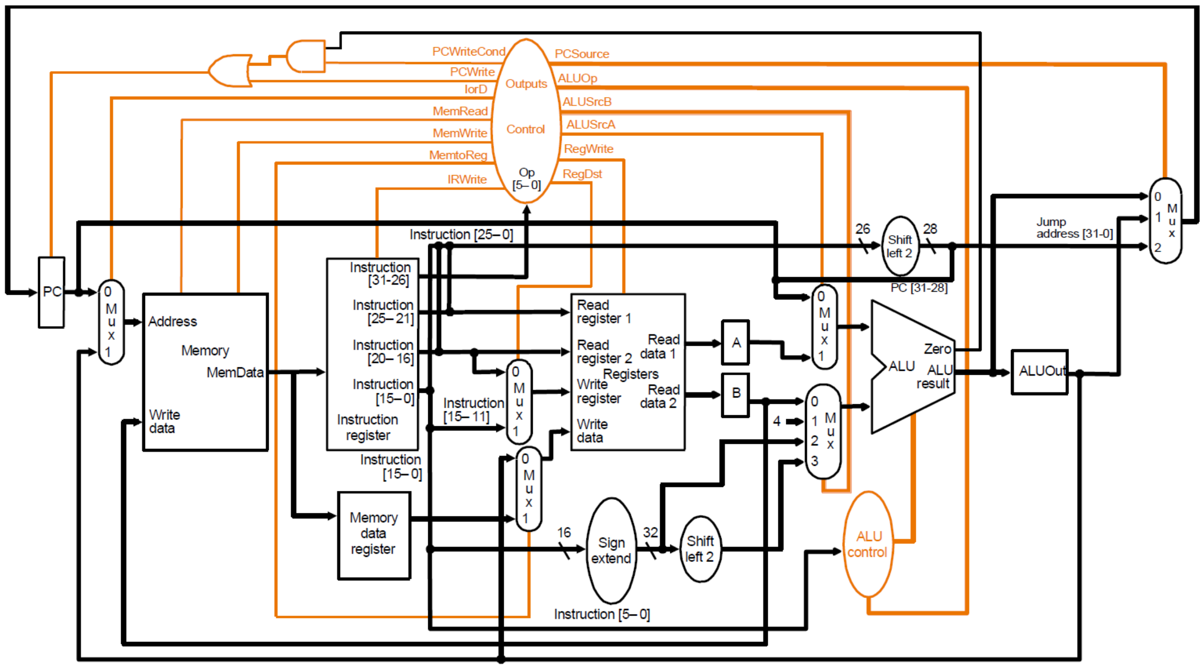


Figure 1: Route1

DDU：Debug and Display Unit，调试和显示单元
下载测试时，用于控制 CPU 运行方式和显示运行结果
数据通路中寄存器堆和存储器均需要增加 1 个读端口，供 DDU 读取并显示其中内容

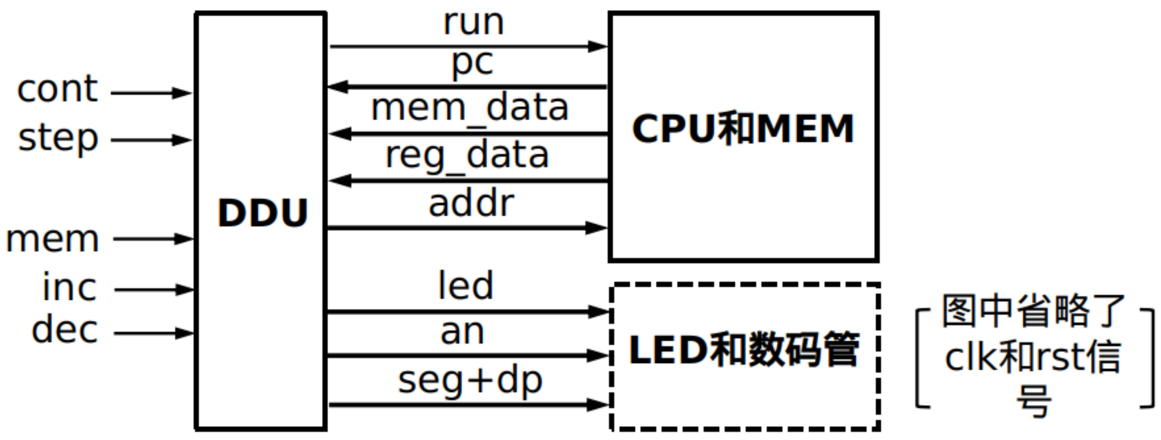


Figure 2: DDU

控制 CPU 运行方式:

- cont = 1：run = 1，控制 CPU 连续执行指令
- cont = 0：每按动 step 一次，run 输出维持一个时钟周期的脉冲，控制 CPU 执行一条指令

查看 CPU 运行状态:

- mem：1，查看 MEM；0，查看 RF
- inc/dec：增加或减小待查看 RF/MEM 的地址 addr
- reg_data/mem_data：从 RF/MEM 读取的数据
- 8 位数码管显示 RF/MEM 的一个 32 位数据
- 16 位 LED 指示 RF/MEM 的地址和 PC 的值

2 实验环境

Linux 下编程调试和仿真，使用 IVerilog，GtkWave 系列工具。
Windows 下用于生成比特流文件，使用 Vivado 2018.2，Verilog HDL
所有下载均在 Nexsy4-DDR 实验板完成。
优秀的代码风格和规范的代码格式也很重要，本次实验借助 Vscode 的 verilog-format 插件进行整理代码的工作。

3 逻辑设计

代码逻辑结构参照我们的 Figure1:Route1，但是我们需要做一点修改以方便编程实现。我们为了简化 beq 和 bne，我们需要额外增加一个加法器，专门实现 imm 和 pc 的相加。

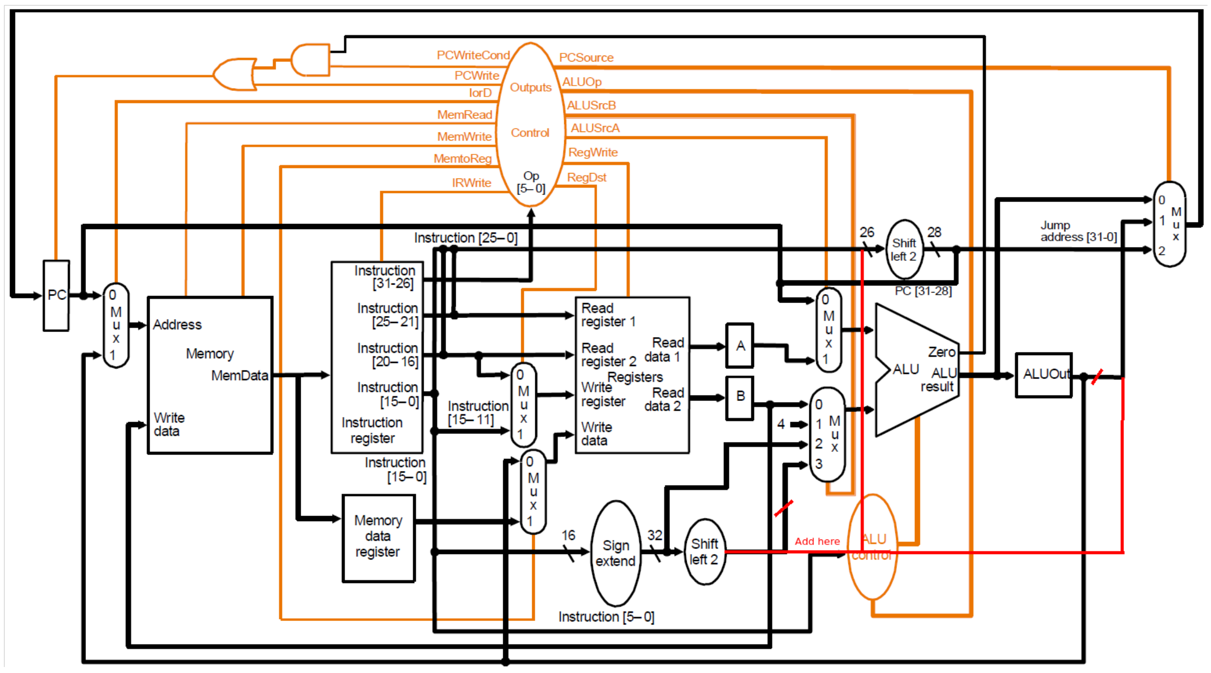


Figure 3: New Route

关于逻辑结构，我们使用有限状态机进行描述。我们讲义给定的参考状态图是 Figure: fsm，但是我们图中并没有我们关于 I-type 的算数运算的状态，为了弥补这一问题，我们对于一个状态进行了调整：fsm_new。

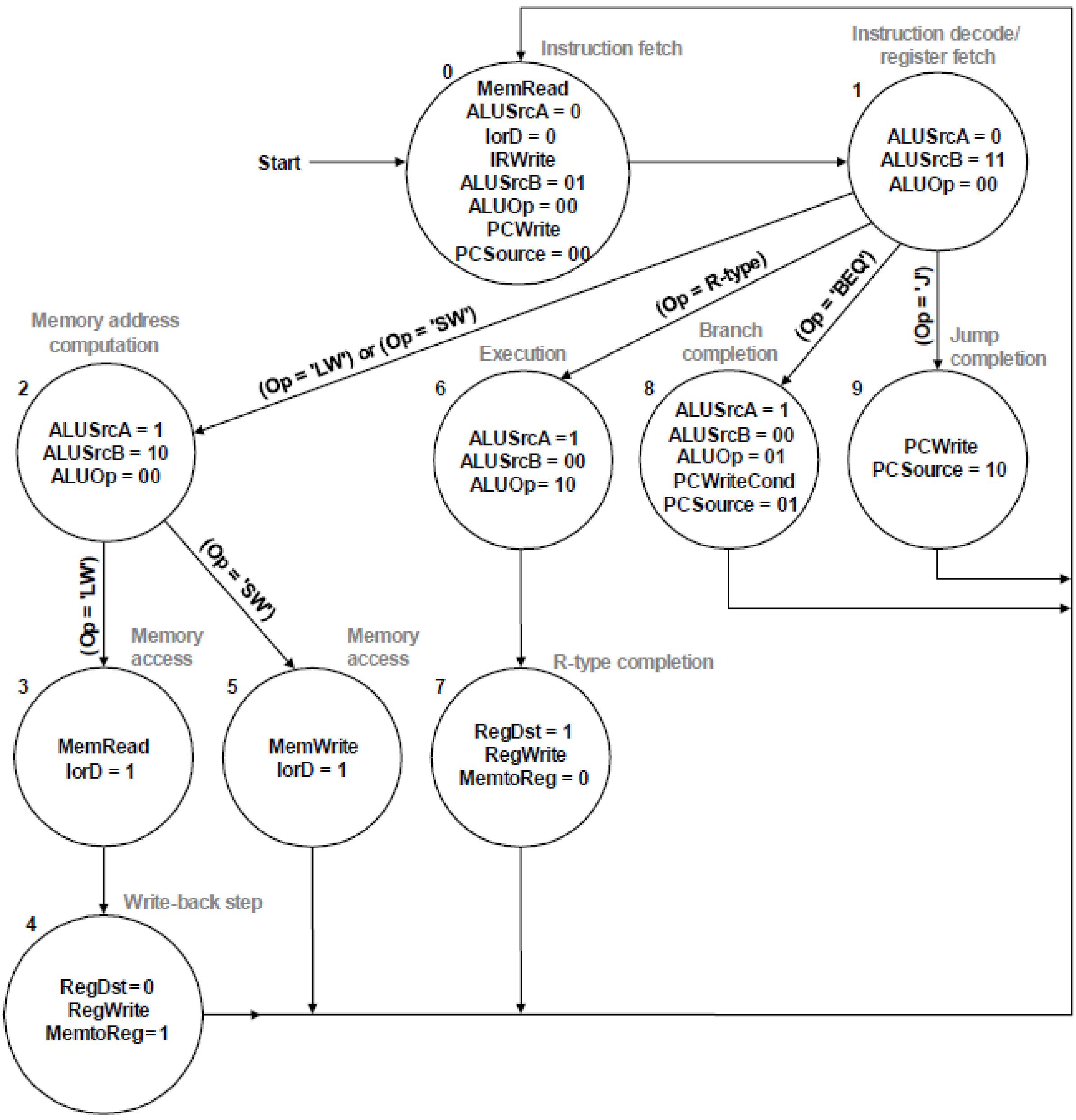

Instruction fetch
Instruction decode/ register fetch
Start
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address computation
Execution
Branch completion
Jump completion
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
6
ALUSrcA =1
ALUSrcB = 00
ALUOp= 10
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
9
PCWrite
PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory access
Memory access
R-type completion
3
MemRead
IorD = 1
5
MemWrite
IorD = 1
7
RegDst = 1
RegWrite
MemtoReg = 0
Write-back step
4
RegDst=0
RegWrite
MemtoReg=1

Figure 4: FSM

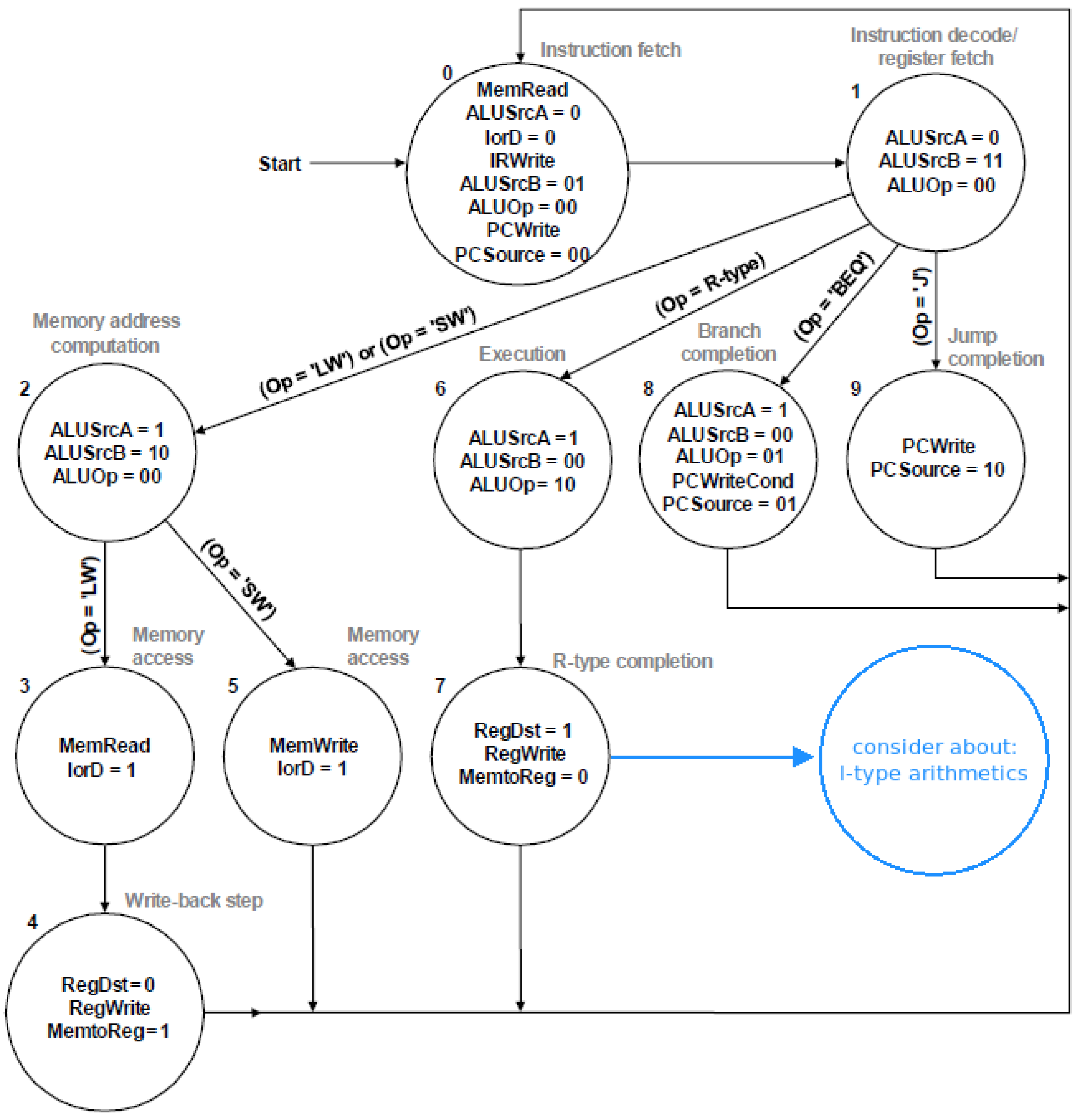

Instruction fetch
Instruction decode/ register fetch
Start
0
MemRead ALUSrcA = 0 IorD = 0 IRWrite ALUSrcB = 01 ALUOp = 00 PCWrite PCSource = 00
1
ALUSrcA = 0 ALUSrcB = 11 ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address computation
Execution
Branch completion
Jump completion
2
ALUSrcA = 1 ALUSrcB = 10 ALUOp = 00
6
ALUSrcA =1 ALUSrcB = 00 ALUOp= 10
8
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 01 PCWriteCond PCSource = 01
9
PCWrite PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory access
Memory access
R-type completion
3
MemRead IorD = 1
5
MemWrite IorD = 1
7
RegDst = 1 RegWrite MemtoReg = 0
consider about: I-type arithmetics
Write-back step
4
RegDst=0 RegWrite MemtoReg=1

Figure 5: FSM_new

这样我们就可以处理关于立即数计算的指令。
同时为了适应简化后的设计线路图，我们还做了以下改动：

1. 我们将 aluop 设作虚设，不做使用。
2. alusrcb 只有一位就足以区分，关于 pc 的处理单独设立的新的加法器。
3. alusrca 的值进行了小调整。
4. 关于 bne 和 beq 的设计。再 alu 中新添加了 Compare equal 的功能，这一设计个人认为十分巧妙在 pc_reg 中有具体介绍这么做的原因。

状态机的详细设计再 define 文件中有相应的描述。
状态设计如下，注释已经详细介绍了各个状态的具体功能。

Listing 1: State design

```
`define IF_STATE     4'b0000 // * if state
`define ID_STATE     4'b0001 // * id state
`define EXE1_STATE   4'b0010 // * exe for lw or sw
`define EXE2_STATE   4'b0011 // * exe for add/sub...(R&I type together)
`define EXE3_STATE   4'b0100 // * exe for branch (beq & bne)
`define EXE4_STATE   4'b0101 // * exe for j inst
`define MEM1_STATE   4'b0110 // * mem for lw - memread
`define MEM2_STATE   4'b0111 // * mem for sw - memwrite
`define WB1_STATE    4'b1000 // * write back for memory inst
`define WB2_STATE    4'b1001 // * write back for add/sub...
```

关于代码，采用模块化的设计，各模块及其功能如下 (顺序为字典序)：

- IP 核 clk_wiz: 避免时钟周期过短。
- IP 核 dist_mem: 实现内存。
- alu: 实现运算器，纯组合逻辑。
- alu_ctrl: 将指令的 opcode 和 funt 部分转化为可以 alu 的具体功能。
- bcd27: 数码管显示用，纯逻辑。
- control: 控制器，根据 6 位 op 实现各个控制信号的输出，同时控制有限状态机。
- cpu: cpu，组合各个 cpu 组件。
- ddu: 顶层，控制 cpu 运行以及 led、数码管的显示。
- define: 定义常量。
- extend: 16 位转换为 32 位。
- ins_split: ir 控制读写，并且拆解指令。
- mdr: 将 mem 中读取的数暂存，用于 lw 指令。
- mem: 内存控制，用于包装 IP 核。
- mux2: 二选一选择器。
- opj_extend: 针对 j 指令的扩展命令。
- pc_reg: 控制 PC。
- regfile: 寄存器组。
- segout: 数码管显示。

用于测试的汇编代码为:

```
# 本文档存储器以字节编址
j  _start

.data
.word  0,8,1,6,0xfffffff8,1,3,5,0
#编译成机器码时便器器会在前面多加个0, 所以后面lw指令地址会多加4

_start:
addi  $t0,$0,3          #t0=3
addi  $t1,$0,5          #t1=5
addi  $t2,$0,1          #t2=1

add   $s0,$t1,$t0
#s0=t1+t0=8   测试add指令  正确继续执行
lw    $s1,12($0)
bne   $s1,$s0,_fail
#不正确跳到fail

and   $s0,$t1,$t0
#s0=t1&t0=1   测试and指令  正确继续执行
lw    $s1,16($0)
bne   $s1,$s0,_fail

xor   $s0,$t1,$t0
#s0=t1^t0=6   测试xor指令  正确继续执行
lw    $s1,20($0)
bne   $s1,$s0,_fail

nor   $s0,$t1,$t0
#s0=t1 nor t0=0xfffffff8
lw    $s1,24($0)
bne   $s1,$s0,_fail

slt   $s0,$t0,$t1   #s0=1
lw    $s1,28($0)
bne   $s1,$s0,_fail

andi  $s0,$t0,7   #s0=3
lw    $s1,32($0)
bne   $s1,$s0,_fail

ori   $s0,$t1,4   #s0=5
lw    $s1,36($0)
bne   $s1,$s0,_fail

sw    $t1,40($0)
lw    $s1,40($0)
beq   $t1,$s1,_sucess

_fail:
sw    $0,8($0)
#失败通过看存储器地址0x08里值, 若为0则测试不通过, 最初地址0x08里值为0
j     _fail

_sucess:
sw    $t2,8($0)
```

```
#全部测试通过，存储器地址0x08里值为1
j     _sucess

#判断测试通过的条件是最后存储器地址0x08里值为1，说明全部通过测试
```

也就是说，加入程序正确，那么程序将在 success 处进入死循环。
我们利用 ip 核的 dist_mem 直接将汇编好的文件转为 coe 文件，直接导入。节约工作量。

4 仿真截图

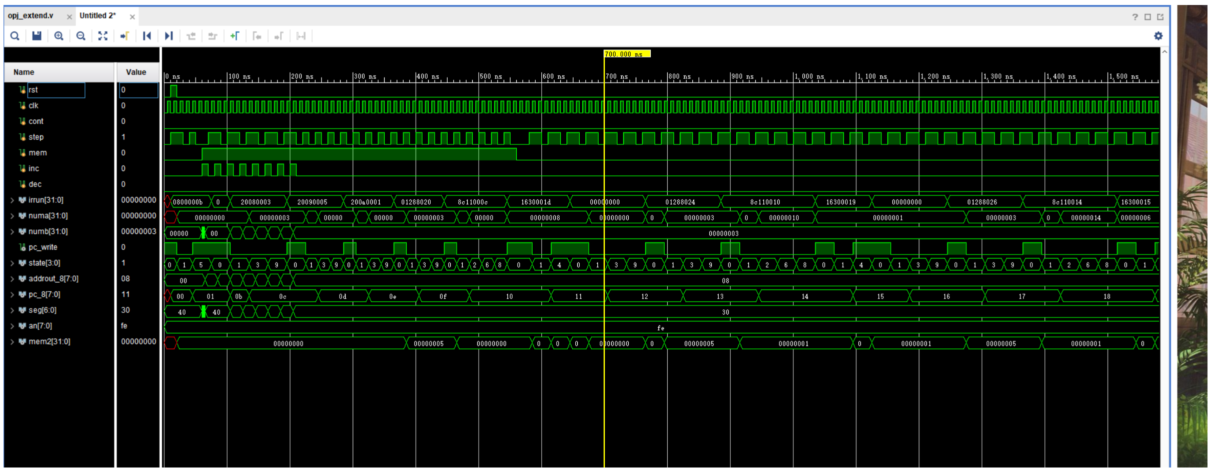


Figure 6: simulation1

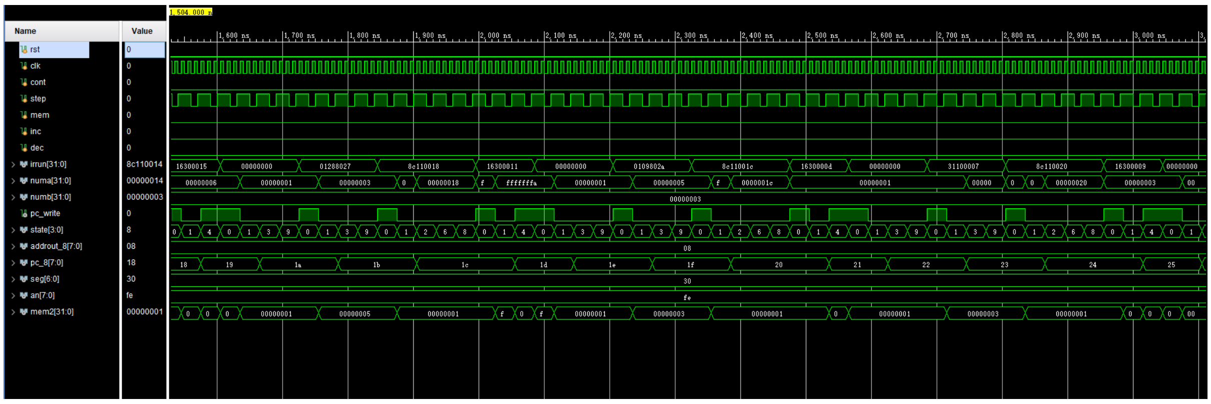


Figure 7: simulation2

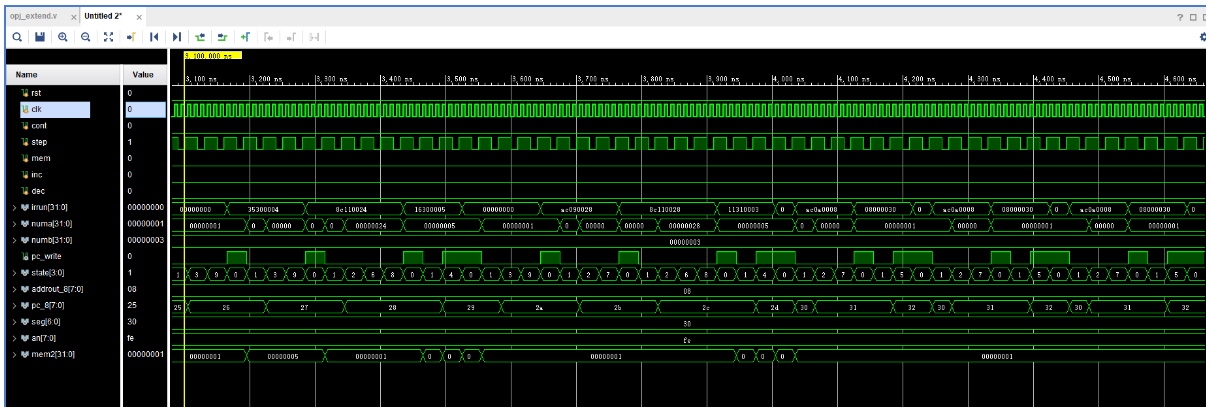
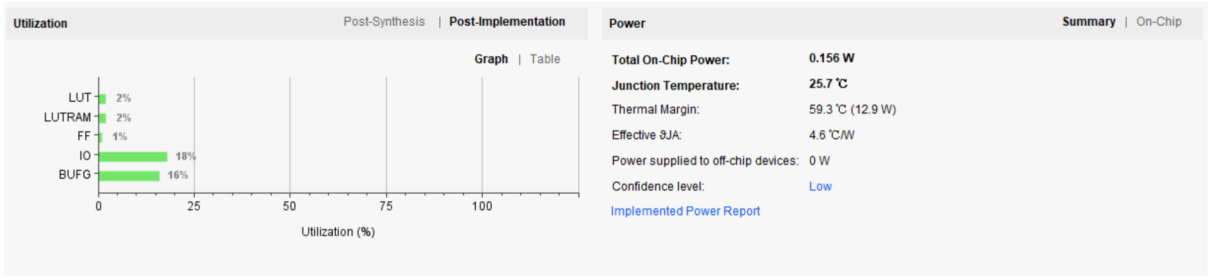


Figure 8: simulation3

拼接起来看图，最终确实进入了那个关于 success 的死循环之中。

5 性能评测截图



Utilization | Post-Synthesis | **Post-Implementation**

Graph | Table

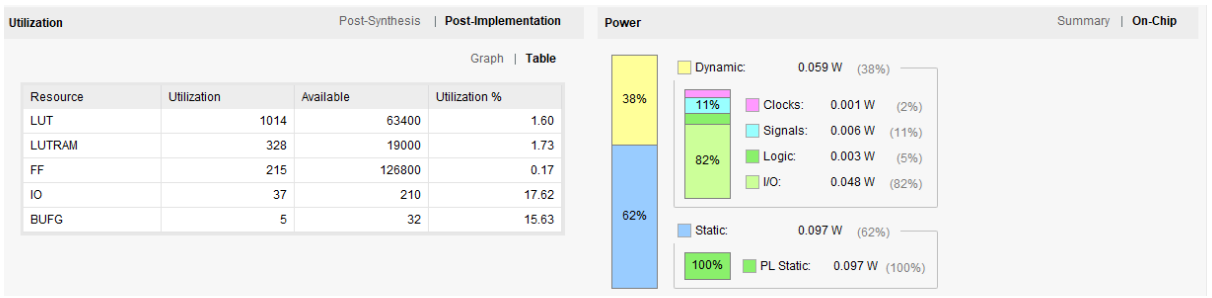
Resource	Utilization (%)
LUT	2%
LUTRAM	2%
FF	1%
IO	18%
BUFG	16%

Power | **Summary** | On-Chip

Total On-Chip Power:	**0.156 W**
Junction Temperature:	**25.7 ℃**
Thermal Margin:	59.3 ℃ (12.9 W)
Effective ϑJA:	4.6 ℃/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Implemented Power Report

Figure 9: performance1



Utilization | Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	1014	63400	1.60
LUTRAM	328	19000	1.73
FF	215	126800	0.17
IO	37	210	17.62
BUFG	5	32	15.63

Power | Summary | **On-Chip**

Dynamic:	0.059 W	(38%)
Clocks:	0.001 W	(2%)
Signals:	0.006 W	(11%)
Logic:	0.003 W	(5%)
I/O:	0.048 W	(82%)
Static:	0.097 W	(62%)
PL Static:	0.097 W	(100%)

Figure 10: performance2

6 实验代码

6.1 烧写板版本代码

```verilog
`include "define.v"
`timescale 1ns / 1ps

module alu (input wire [`EXE_BUS] aluop,
          input wire [`REG_BUS] numa,
          input wire [`REG_BUS] numb,
          output reg [`REG_BUS] num_out,
          output reg iszero);

always @* begin
    case (aluop)
        `EXE_ADD :num_out = numa + numb;
        `EXE_SUB :num_out = numa - numb;
        `EXE_CMP :num_out = numa == numb ? 1 :0;
        `EXE_SLT :num_out = numa < numb ? 1 :0;
        `EXE_NOR :num_out = ~ numa;
        `EXE_OR : num_out = numa | numb;
        `EXE_AND :num_out = numa & numb;
        `EXE_XOR :num_out = numa ^ numb;
        `EXE_NOP :num_out = numa;
    endcase
end

always @* begin
    iszero = (num_out == 0 ? 1 :0);
end
endmodule
```

Listing 3: alu.v

```verilog
`include "define.v"
`timescale 1ns / 1ps

module alu_ctrl (input wire[`OP_BUS] opcode,
                input wire [`FUNT_BUS] funt,
                output reg [`EXE_BUS] alu_op);

always @* begin
    case (opcode)
        `OP_R_TYPE :begin
            case (funt)   // * R - type translate
                `FUNC_ADD :alu_op = `EXE_ADD;
                `FUNC_SUB :alu_op = `EXE_SUB;
                `FUNC_AND :alu_op = `EXE_AND;
                `FUNC_OR :alu_op = `EXE_OR;
                `FUNC_XOR :alu_op = `EXE_XOR;
                `FUNC_NOR :alu_op = `EXE_NOR;
                `FUNC_SLT :alu_op = `EXE_SLT;
            endcase
        end
        `OP_ADDI :alu_op = `EXE_ADD;
        `OP_ANDI :alu_op = `EXE_AND;
        `OP_ORI : alu_op = `EXE_OR;
        `OP_XORI :alu_op = `EXE_XOR;
        `OP_LW  : alu_op = `EXE_ADD;
        `OP_SW  : alu_op = `EXE_ADD;
        `OP_BEQ : alu_op = `EXE_CMP;
        `OP_BNE : alu_op = `EXE_CMP;
        `OP_SLTI :alu_op = `EXE_SLT;
        `OP_J   : alu_op = `EXE_NOP;
    endcase
end

endmodule
```

Listing 4: alu_ctrl.v

```verilog
`timescale 1ns / 1ps

module BCD27(input [3:0]m,
            output [6:0] out);
reg[6:0]seg;
assign out = seg;
always@(m)
    case(m)
        //---------------1234567
        4'b0000:seg = 7'b1000000; // 0
        4'b0001:seg = 7'b1111001; // 1
        4'b0010:seg = 7'b0100100; // 2
        4'b0011:seg = 7'b0110000; // 3
        4'b0100:seg = 7'b0011001; // 4
        4'b0101:seg = 7'b0010010; // 5
        4'b0110:seg = 7'b0000010; // 6
        4'b0111:seg = 7'b1111000; // 7
        4'b1000:seg = 7'b0000000; // 8
        4'b1001:seg = 7'b0010000; // 9
        4'b1010:seg = 7'b0001000; // A
        4'b1011:seg = 7'b0000011; // b
        4'b1100:seg = 7'b1000110; // c
        4'b1101:seg = 7'b0100001; // d
        4'b1110:seg = 7'b0000110; // E
        4'b1111:seg = 7'b0001110; // F
        default:seg = 7'b1111111;
    endcase
    end
endmodule
```

Listing 5: bcd27.v

```verilog
`include "define.v"
`timescale 1ns / 1ps

//* Controler
//* input with op
//* output control information

module control (input wire clk,
                input wire rst,
                input wire [`OP_BUS] op, // * input the opcode part
                output reg pc_write_cond, // * control pc
                output reg pc_write,    // * control pc
                output reg ins_or_data, // * control mem
                output reg mem_read,    // * control mem
                output reg mem_write,   // * control mem
                output reg ir_write,    // * control ir
                output reg [1:0] pc_source, // * control pc source
                output reg [1:0] alu_op, // * control alu
                output reg alu_srca,    // * control numa
                output reg alu_srcb,    // * control numb
                output reg reg_write,   // * control regfile writable
                output reg regdst,      // * control where to write data
                output reg mem_to_reg); // * control the data source

    reg [`STATE_BUS] state,next_state;

    initial begin
        state       = `IF_STATE;
        pc_write_cond = 0;
        pc_write    = 0;
        ins_or_data = 0;
        mem_read    = 0;
        mem_write   = 0;
        ir_write    = 0;
        pc_source   = 0;
        alu_op      = 0;
        alu_srca    = 0;
        alu_srcb    = 0;
        reg_write   = 0;
        regdst      = 0;
    end

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            state <= `IF_STATE;
            end else begin
            state <= next_state;
        end
    end

    always @* begin
        case (state)
            `IF_STATE :next_state = `ID_STATE;
            `ID_STATE :begin
                case (op)
                `OP_LW : next_state = `EXE1_STATE;
                `OP_SW : next_state = `EXE1_STATE;
                `OP_BEQ :next_state = `EXE3_STATE;
                `OP_BNE :next_state = `EXE3_STATE;
                `OP_J  : next_state = `EXE4_STATE;
                default :next_state = `EXE2_STATE;
            endcase
        end
        `EXE1_STATE :begin
            if (op == `OP_LW)
                next_state = `MEM1_STATE;
            else
                next_state = `MEM2_STATE;
        end
        `EXE2_STATE :next_state = `WB2_STATE;
        `EXE3_STATE :next_state = `IF_STATE;
        `EXE4_STATE :next_state = `IF_STATE;
        `MEM1_STATE :next_state = `WB1_STATE;
        `MEM2_STATE :next_state = `IF_STATE;
        `WB1_STATE :next_state = `IF_STATE;
        `WB2_STATE :next_state = `IF_STATE;
    endcase
end

always @* begin
    // * if state == > initial state
    if (state == `IF_STATE) begin
        pc_write_cond = 0;
        pc_write    = 1;      // * pc writable
        ins_or_data = 0;
        mem_read    = 1;      // * read mem for instructions
        mem_write   = 0;
        ir_write    = 1;      // * ir writable
        pc_source   = 2'b00;
        alu_op      = 0;
        alu_srca    = 0;
        alu_srcb    = 0;
        reg_write   = 0;
        regdst      = 0;
    end

    // * id state
    if (state == `ID_STATE) begin
        pc_write = 0;
        ir_write = 0;
        mem_read = 0;
    end

    // * exe1 state
    if (state == `EXE1_STATE) begin
        alu_srca = 0;
```

```
            alu_srcb = 1;
            alu_op  = 2'b00;
        end

        // * exe2 state
        if (state == `EXE2_STATE) begin
            alu_srca = 0;
            alu_srcb = (op == `OP_R_TYPE ? 0 :1);
            alu_op  = 2'b10;
        end

        // * exe3 state
        if (state == `EXE3_STATE) begin
            alu_srca     = 1;
            alu_srcb     = 0;
            alu_op       = 2'b01;
            pc_write     = 1;
            pc_write_cond = (op == `OP_BEQ ? 1 :0);
            pc_source    = 2'b01;
        end

        // * exe4 state
        if (state == `EXE4_STATE) begin
            pc_write = 1;
            pc_source = 2'b10;
        end

        // * mem1 state
        if (state == `MEM1_STATE) begin
            mem_read   = 1;
            ins_or_data = 1;
        end

        // * mem2 state
        if (state == `MEM2_STATE) begin
            mem_write  = 1;
            ins_or_data = 1;
        end

        // * wb1 state
        if (state == `WB1_STATE) begin
            regdst    = 0;
            reg_write = 1;
            mem_to_reg = 1;
        end

        // * wb2 state
        if (state == `WB2_STATE) begin
            regdst    = (op == `OP_R_TYPE ? 1 :0);
            reg_write = 1;
            mem_to_reg = 0;
        end
    end
endmodule
```

Listing 6: control.v

```
`include "define.v"
`timescale 1ns / 1ps

module cpu (input wire clk,
           input wire rst,
           output [`REG_BUS] pcout,
           input [`REG_BUS] inputaddr,
           output [`REG_BUS] memout_show,
           output [`REG_BUS] regout_show);
    // * ir split
    wire [`NUM1_BUS] num1;
    wire [`NUM2_BUS] num2;
    wire [`NUM3_BUS] num3;
    wire [`NUM4_BUS] num4;
    wire [`NUM5_BUS] num5;
    wire [`NUM6_BUS] num6;

    wire [`REG_BUS] jext;
    wire [`REG_BUS] aluout;
    wire [`REG_BUS] numa;
    wire [`REG_BUS] numb;
    wire [`REG_BUS] alu_numa;
    wire [`REG_BUS] alu_numb;
    wire [`REG_BUS] memaddr;
    wire [`REG_BUS] memmdr;
    wire [`REG_BUS] regmem;
    wire [`REG_BUS] immext;
    wire [`REG_BUS] memdata;

    wire [`EXE_BUS] aluexe;
    wire [`OP_BUS] opcode;

    // * control wire
    wire ins_or_data;
    wire pc_write_cond;
    wire pc_write;
    wire alu_srca;
    wire alu_srcb;
    wire ir_write;
    wire mem_write;
    wire mem_read;
    wire is_zero;
    wire mem_to_reg;
    wire reg_write;
    wire regdst;
    wire [1:0] pc_source;
    wire [1:0] alu_op;

    opj_extend jexter (
    .ins (num5),
    .pc (pcout),
    .extd (jext));

    pc_reg pc (
    .clk (clk),
    .rst (rst),
    .pccond (pc_write_cond),
    .pcsource (pc_source),
    .aluinfo(is_zero),
    .pc_write(pc_write),
    .immpc (num3),
    .address(jext),
    .pcout (pcout));

    // * A little change here
    // * address must divide by 4 here --> real addr in mem

    // * get the address
    mux2 mux_ins_or_data (
    .select (ins_or_data),
    .route1 (aluout >> 2),
    .route2 (pcout >> 2),
    .outdata (memaddr));

    // * memory
    mem memory (
    .clk(clk),
    .memwrite (mem_write),
    .indata (numb),
    .addr (memaddr),
    .memdata (memdata),
    .dpra (inputaddr),
    .dpo (memout_show));

    // * ir
    ins_split ir (
    .ir_write (ir_write),
    .ins (memdata),
    .opcode (opcode),
    .num1 (num1),
    .num2 (num2),
    .num3 (num3),
    .num4 (num4),
    .num5 (num5),
    .num6(num6),
    .irrun (irrun));

```

```
// * mdr
mdr memory_data_reg (
.memread (mem_read),
.memdata (memdata),
.memout (memmdr));

// * control
control ctrl (
.clk (clk),
.rst (rst),
.op (opcode),
.pc_write_cond (pc_write_cond),
.pc_write (pc_write),
.ins_or_data (ins_or_data),
.mem_read(mem_read),
.mem_write (mem_write),
.ir_write(ir_write),
.pc_source(pc_source),
.alu_op (alu_op),
.alu_srca (alu_srca),
.alu_srcb (alu_srcb),
.reg_write (reg_write),
.regdst (regdst),
.mem_to_reg (mem_to_reg));

// * reg write
mux2 reg_write_mem (
.select (mem_to_reg),
.route1(memmdr),
.route2(aluout),
.outdata (regmem));

// * reg
regfile regs (
.clk (clk),
.rs (num1),
.rt (num2),
.rd (num4),
.numa (numa),
.numb (numb),
.reg_write (reg_write),
.regdst (regdst),
.indata (regmem),
.regaddr (inputaddr),
.reg_out(regout_show));

// * mux for numa
mux2 muxnuma (
.select (alu_srca),
.route1 (pcout),
.route2 (numa) ,
```

```verilog
    .outdata (alu_numa));

    // * extend immediate num
    extend extder (
    .ins (num3),
    .extd (immext));

    // * mux for numb
    mux2 muxnumb (
    .select (alu_srcb),
    .route1 (immext),
    .route2(numb),
    .outdata (alu_numb));

    // * translate alu info
    alu_ctrl aluexe_maker (
    .opcode (opcode),
    .funt (num6),
    .alu_op (aluexe));

    // * alu
    alu alucalc (
    .aluop (aluexe),
    .numa (alu_numa),
    .numb(alu_numb),
    .num_out (aluout),
    .iszero (is_zero));
endmodule
```

Listing 7: cpu.v

```verilog
`timescale 1ns / 1ps
`include "define.v"

module ddu(input cont,
         input rst,
         input clk_board100,
         input step,
         input mem,
         input inc,
         input dec,
         output reg [7:0] addrout_8,
         output wire [7:0] pc_8,
         output wire [6:0] seg,
         output wire [7:0] an);

    wire clk_boardtmp;

    // * translate 100Mhz
    clk_wiz_0 (.clk_in1(clk_board100),.clk_out1(clk_boardtmp));

    wire [`REG_BUS] memshow;
    reg [`REG_BUS] addrout;
    wire [`REG_BUS] pc;
    reg clk_board = 0;
    reg [14:0] cnt;

    // * slow clock
    always @(posedge clk_boardtmp) begin
        if (cnt > = 15'd100) begin
            cnt       = 0;
            clk_board = ~clk_board;
        end
        else
            cnt = cnt + 1;
    end

    reg [`REG_BUS] addr = 0;
    wire [`REG_BUS] mem1;
    wire [`REG_BUS] mem2;
    reg clkt;
    reg delay;
    reg incdelay;
    reg decdelay;
    wire clk;

    always @(posedge clk_board or negedge rst)
    begin
        if (rst) begin
            addr <= 0;
            incdelay = 1;
            decdelay = 1;
            delay    = 0;
        end
        else
        begin
            // * control addr
            if (!incdelay && inc) addr = addr + 1;
            if (!decdelay && dec) addr = addr - 1;

            incdelay = inc;
            decdelay = dec;

            addrout = addr;

            // * generate only one pulse
            if (!delay && step)
                clkt = 1;
            else
                clkt = 0;

            delay = step;
        end
    end

    assign memshow = mem ? mem1 :mem2;
    assign clk   = cont ? clk_board :clkt;

    // * cpu
    cpu CPU_sim (.clk (clk), .rst (rst), .pcout (pc), .inputaddr(addr) , .
        memout_show (mem1), .regout_show(mem2));

    always @* addrout_8 = {addr [7:0]};
    assign pc_8      = {pc[9:2]};

    // * segout
    segout segmental (.clk (clk_board), .indata (memshow), .an(an) ,.seg(seg));
endmodule
```

Listing 8: ddu.v

```
//* r_type
//* begin with 000000
`define OP_R_TYPE 6'b000000
`define FUNC_ADDU 6'b100001
`define FUNC_ADD 6'b100000 //* do
`define FUNC_SUB 6'b100010 //* do
`define FUNC_SUBU 6'b100011
`define FUNC_AND 6'b100100 //* do
`define FUNC_OR 6'b100101 //* do
`define FUNC_XOR 6'b100110 //* do
`define FUNC_NOR 6'b100111 //* do
`define FUNC_SLT 6'b101010 //* do
`define FUNC_SLTU 6'b101011
`define FUNC_SLL 6'b000000 //* do
`define FUNC_SRL 6'b000010 //* do
`define FUNC_SRA 6'b000011
`define FUNC_SLLV 6'b000100
`define FUNC_SRLV 6'b000110
`define FUNC_SRAV 6'b000111
`define FUNC_JR 6'b001000

//* i_type
`define OP_ADDI 6'b001000 //* do
`define OP_ANDI 6'b001100 //* do
`define OP_ORI  6'b001101 //* do
`define OP_XORI 6'b001110 //* do
`define OP_LUI  6'b001111
`define OP_LW   6'b100011 //* do
`define OP_SW   6'b101011 //* do
`define OP_BEQ  6'b000100 //* do
`define OP_BNE  6'b000101 //* do
`define OP_SLTI 6'b001010 //* do
`define OP_SLTIU 6'b001101

```

```
//* j_type
`define OP_J   6'b000010 //* do
`define OP_JAL  6'b000011

//* exe will be used
`define EXE_ADD 4'b0000
`define EXE_SUB 4'b0001
`define EXE_CMP 4'b0010
`define EXE_SLT 4'b0011
`define EXE_NOR 4'b0100
`define EXE_OR  4'b0101
`define EXE_AND 4'b0110
`define EXE_XOR 4'b0111
`define EXE_NOP 4'b1111

//* bus_information
`define PCCTRL_BUS 1:0
`define STATE_BUS 3:0
`define EXE_BUS  3:0
`define RS_BUS   4:0
`define RT_BUS   4:0
`define RD_BUS   4:0
`define SHAMT_BUS 4:0
`define STALL_BUS 5:0
`define OP_BUS   5:0
`define FUNT_BUS 5:0
`define ADDR_BUS 7:0
`define DATA_BUS 31:0
`define REG_BUS  31:0

`define OPCODE_BUS 31:26
`define NUM1_BUS  25:21
`define NUM2_BUS  20:16
`define NUM3_BUS  15:0
`define NUM4_BUS  15:11
`define NUM5_BUS  25:0
`define NUM6_BUS  5:0
`define FUNTCODE_BUS 5:0

`define REG_VEC  0:31
`define MEM_VEC  0:255

//* control tags
`define INS_TAG  0
`define REG_TAG  1

//* control state machine
`define IF_STATE 4'b0000 // * if state
`define ID_STATE 4'b0001 // * id state
`define EXE1_STATE 4'b0010 // * exe for lw or sw
`define EXE2_STATE 4'b0011 // * exe for add/sub...(R&I type together)
```

```
`define EXE3_STATE 4'b0100 // * exe for branch (beq & bne)
`define EXE4_STATE 4'b0101 // * exe for j inst
`define MEM1_STATE 4'b0110 // * mem for lw - memread
`define MEM2_STATE 4'b0111 // * mem for sw - memwrite
`define WB1_STATE 4'b1000 // * write back for memory inst
`define WB2_STATE 4'b1001 // * write back for add/sub...
```

Listing 9: define.v

```
`include "define.v"
`timescale 1ns / 1ps

module extend(input wire [`NUM3_BUS] ins,
            output wire [`REG_BUS] extd);
assign
extd = 32'b0 | ins[`NUM3_BUS];

endmodule
```

Listing 10: extend.v

```
`include "define.v"
`timescale 1ns / 1ps

//* Instruction spliter
//* Opcode 31:26
//* num1  25:21
//* num2  20:16
//* num3  15: 0

module ins_split (input wire ir_write,
               input wire [`REG_BUS] ins,
               output reg [`OP_BUS] opcode,
               output reg [`RS_BUS] num1,
               output reg [`RT_BUS] num2,
               output reg [`NUM3_BUS] num3,
               output reg [`RD_BUS] num4,
               output reg [`NUM5_BUS] num5,
               output reg [`NUM6_BUS] num6,
               output wire [`REG_BUS] irrun);

   reg[`REG_BUS] ir;
   always @* begin
      if (ir_write) ir = ins;
      opcode        = {ir[`OPCODE_BUS]};
      num1          = {ir[`NUM1_BUS]};
      num2          = {ir[`NUM2_BUS]};
      num3          = {ir[`NUM3_BUS]};
      num4          = {ir[`NUM4_BUS]};
      num5          = {ir[`NUM5_BUS]};
      num6          = {ir[`NUM6_BUS]};
   end
```

```

    assign irrun = ir;
endmodule
```
Listing 11: ins_split.v

```
`include "define.v"
`timescale 1ns / 1ps

module mdr (input wire memread,
            input wire [`REG_BUS] memdata,
            output reg [`REG_BUS] memout);

    reg [`REG_BUS] mem;

    initial begin
        mem = 0;
    end


    always @* begin
        if (memread)
            mem = memdata;

        memout = mem;
    end

endmodule
```
Listing 12: mdr.v

```
`include "define.v"
`timescale 1ns / 1ps

module mem (input wire clk,
            input wire memwrite,
            input wire [`REG_BUS] indata,
            input wire [`REG_BUS] addr,
            output wire [`REG_BUS] memdata,
            input wire [`REG_BUS] dpra,
            output wire [`REG_BUS] dpo);

    dist_mem_gen_0 memorys (.a(addr), .d(indata), .clk(clk), .we(memwrite), .
        spo(memdata), .dpra(dpra), .dpo(dpo));

endmodule
```
Listing 13: mem.v

```
`include "define.v"
`timescale 1ns / 1ps

```

```
module mux2(input wire select,
         input wire [`REG_BUS] route1,
         input wire [`REG_BUS] route2,
         output reg [`REG_BUS] outdata);

   always @*
   if (select)
      outdata = route1;
   else
      outdata = route2;

endmodule
```

Listing 14: mux2.v

```
`include "define.v"
`timescale 1ns / 1ps

module opj_extend (input wire [`NUM5_BUS] ins,
                input wire [`REG_BUS] pc,
                output wire [`REG_BUS] extd);

assign extd = (ins << 2) | ((pc >> 28) << 28);

endmodule
```

Listing 15: opj_extend.v

```
`include "define.v"
`timescale 1ns / 1ps

//* 00. normal
//*    PC = PC + 4
//* 01. special case 1 - bne & beq
//*    PC = immpc + 4
//* 10. special case 2 - j
//*    PC = jump address

module pc_reg (input wire clk,
            input wire rst,            // * clk & reset
            input wire pccond,
            input wire aluinfo,
            input wire pc_write,
            input wire [1:0] pcsource,
            input wire [`NUM3_BUS] immpc,
            input wire [`REG_BUS] address,
            output reg [`REG_BUS] pcout);

   reg [`REG_BUS] pc;

   initial begin
      pc = 0;
```

```verilog
    end

    always @(posedge clk or posedge rst) begin
        pcout = pc;
        if (rst) begin
            pc <= 0;
        end
        else
        begin
            if (pc_write) begin
                case (pcsource)
                    2'b00: pc <= pc + 4;
                    2'b01:
                    begin
                        // * 2 cases: pccond = 1 (beq) & iszero (from cmp = 1) = 0
                        // *     or pccond = 0 (bne) & iszero (from cmp = 0) = 1
                        if (pccond ^ aluinfo)
                            pc <= pc;
                        else
                            pc <= pc + (immpc << 2);
                    end
                    2'b10:   pc <= address;
                    default :pc <= pc;
                endcase
            end
        end



    end
endmodule
```

Listing 16: pc_reg.v

```verilog
`include "define.v"
`timescale 1ns / 1ps

module regfile(input wire clk,
               input wire [`RS_BUS] rs,
               input wire [`RT_BUS] rt,
               input wire [`RD_BUS] rd,
               input wire reg_write,
               input wire regdst,
               input wire [`REG_BUS] indata,
               output reg [`REG_BUS] numa,
               output reg [`REG_BUS] numb,
               input wire [`REG_BUS] regaddr,
               output reg [`REG_BUS] reg_out);

    reg [`REG_BUS] register [0:31];
    integer i;

    initial begin
        for (i = 0; i < 32 ; i = i + 1) begin
            register[i] = 0;
        end
    end

    always @(posedge clk) begin
        if (reg_write) begin
            if (regdst) begin
                register[rd] = indata;
                end else begin
                register[rt] = indata;
            end
        end

        numa = register [rs];
        numb = register [rt];
    end
    always@* reg_out = register [(regaddr % 31)];
endmodule
```

Listing 17: regfile.v

```
`include "define.v"
`timescale 1ns / 1ps

module segout (input wire clk,
               input wire [`REG_BUS] indata,
               output reg [7:0] an,
               output reg [6:0] seg);

    reg [3:0] Q [7:0];
    reg [3:0] num;
    wire [6:0] seg0[7:0];
    reg [31:0] count;
    reg clk1;

    BCD27 B0(Q[0],seg0[0]);
    BCD27 B1(Q[1],seg0[1]);
    BCD27 B2(Q[2],seg0[2]);
    BCD27 B3(Q[3],seg0[3]);
    BCD27 B4(Q[4],seg0[4]);
    BCD27 B5(Q[5],seg0[5]);
    BCD27 B6(Q[6],seg0[6]);
    BCD27 B7(Q[7],seg0[7]);

    initial
    begin
        an    <= 8'b11111111;
        clk1  <= 0;
        num   <= 0;
        count <= 0;
    end

    always @*
    begin
    Q[7] <= {indata[31:28]};//(indata & 32'
        b1111_0000_0000_0000_0000_0000_0000_0000) >> 28;
    Q[6] <= {indata[27:24]};// (indata & 32'
        b0000_1111_0000_0000_0000_0000_0000_0000) >> 24;
    Q[5] <= {indata[23:20]};// (indata & 32'
        b0000_0000_1111_0000_0000_0000_0000_0000) >> 20;
    Q[4] <= {indata[19:16]};// (indata & 32'
        b0000_0000_0000_1111_0000_0000_0000_0000) >> 16;
    Q[3] <= {indata[15:12]};// (indata & 32'
        b0000_0000_0000_0000_1111_0000_0000_0000) >> 12;
    Q[2] <= {indata[11: 8]};// (indata & 32'
        b0000_0000_0000_0000_0000_1111_0000_0000) >> 8;
    Q[1] <= {indata[7: 4]};// (indata & 32'
        b0000_0000_0000_0000_0000_0000_1111_0000) >> 4;
    Q[0] <= {indata[3: 0]};// (indata & 32'
        b0000_0000_0000_0000_0000_0000_0000_1111);
    end

    always@(posedge clk)
        if (count> = 31'd19999)
        begin
            clk1 <= ~clk1;
            count = 31'b0;
            num <= (num+1)%8;
        end
        else
            count = count+1;

    always@*
    case(num)
        0:begin
            an  <= 8'b11111110;
            seg <= seg0[0];
        end
        1:begin
            an  <= 8'b11111101;
            seg <= seg0[1];
        end
        2:begin
            an  <= 8'b11111011;
            seg <= seg0[2];
        end
        3:begin
            an  <= 8'b11110111;
            seg <= seg0[3];
        end
        4:begin
```

```
            an  <= 8'b11101111;
            seg <= seg0[4];
        end
        5:begin
            an  <= 8'b11011111;
            seg <= seg0[5];
        end
        6:begin
            an  <= 8'b10111111;
            seg <= seg0[6];
        end
        7:begin
            an  <= 8'b01111111;
            seg <= seg0[7];
        end
    endcase
endmodule
```

Listing 18: segout.v

```
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets inc_IBUF];
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets dec_IBUF];
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets rst_IBUF];

## Clock signal
set_property -dict { PACKAGE_PIN E3 IOSTANDARD LVCMOS33 } [get_ports {
    clk_board100 }]; #IO_L12P_T1_MRCC_35 Sch=clk
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {
    clk_board100}];


##Switches
set_property -dict { PACKAGE_PIN J15 IOSTANDARD LVCMOS33 } [get_ports { cont
    }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN H6 IOSTANDARD LVCMOS33 } [get_ports { inc }];
     #IO_L24P_T3_35 Sch=sw[12]
set_property -dict { PACKAGE_PIN U12 IOSTANDARD LVCMOS33 } [get_ports { dec
    }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
set_property -dict { PACKAGE_PIN U11 IOSTANDARD LVCMOS33 } [get_ports { step
    }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
set_property -dict { PACKAGE_PIN V10 IOSTANDARD LVCMOS33 } [get_ports { mem
    }]; #IO_L21P_T3_DQS_14 Sch=sw[15]


## LEDs
set_property -dict { PACKAGE_PIN H17 IOSTANDARD LVCMOS33 } [get_ports {
    addrout_8[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15 IOSTANDARD LVCMOS33 } [get_ports {
    addrout_8[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13 IOSTANDARD LVCMOS33 } [get_ports {
    addrout_8[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14 IOSTANDARD LVCMOS33 } [get_ports {
    addrout_8[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
set_property -dict { PACKAGE_PIN R18 IOSTANDARD LVCMOS33 } [get_ports {
    addrout_8[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
set_property -dict { PACKAGE_PIN V17 IOSTANDARD LVCMOS33 } [get_ports {
    addrout_8[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
set_property -dict { PACKAGE_PIN U17 IOSTANDARD LVCMOS33 } [get_ports {
    addrout_8[6] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
set_property -dict { PACKAGE_PIN U16 IOSTANDARD LVCMOS33 } [get_ports {
    addrout_8[7] }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
set_property -dict { PACKAGE_PIN V16 IOSTANDARD LVCMOS33 } [get_ports { pc_8
    [0] }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
set_property -dict { PACKAGE_PIN T15 IOSTANDARD LVCMOS33 } [get_ports { pc_8
    [1] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
set_property -dict { PACKAGE_PIN U14 IOSTANDARD LVCMOS33 } [get_ports { pc_8
    [2] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
set_property -dict { PACKAGE_PIN T16 IOSTANDARD LVCMOS33 } [get_ports { pc_8
    [3] }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
set_property -dict { PACKAGE_PIN V15 IOSTANDARD LVCMOS33 } [get_ports { pc_8
    [4] }]; #IO_L16P_T2_CSI_B_14 Sch=led[12]
set_property -dict { PACKAGE_PIN V14 IOSTANDARD LVCMOS33 } [get_ports { pc_8
    [5] }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
set_property -dict { PACKAGE_PIN V12 IOSTANDARD LVCMOS33 } [get_ports { pc_8
    [6] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
set_property -dict { PACKAGE_PIN V11 IOSTANDARD LVCMOS33 } [get_ports { pc_8
    [7] }]; #IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]


##7 segment display
set_property -dict { PACKAGE_PIN T10 IOSTANDARD LVCMOS33 } [get_ports { seg[0]
     }]; #IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10 IOSTANDARD LVCMOS33 } [get_ports { seg[1]
     }]; #IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16 IOSTANDARD LVCMOS33 } [get_ports { seg[2]
     }]; #IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13 IOSTANDARD LVCMOS33 } [get_ports { seg[3]
     }]; #IO_L17P_T2_A26_15 Sch=cd
set_property -dict { PACKAGE_PIN P15 IOSTANDARD LVCMOS33 } [get_ports { seg[4]
     }]; #IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11 IOSTANDARD LVCMOS33 } [get_ports { seg[5]
     }]; #IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18 IOSTANDARD LVCMOS33 } [get_ports { seg[6]
     }]; #IO_L4P_T0_D04_14 Sch=cg

set_property -dict { PACKAGE_PIN J17 IOSTANDARD LVCMOS33 } [get_ports { an[0]
    }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18 IOSTANDARD LVCMOS33 } [get_ports { an[1]
    }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9 IOSTANDARD LVCMOS33 } [get_ports { an[2]
    }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14 IOSTANDARD LVCMOS33 } [get_ports { an[3]
    }]; #IO_L19P_T3_A22_15 Sch=an[3]
```

```
set_property -dict { PACKAGE_PIN P14 IOSTANDARD LVCMOS33 } [get_ports { an[4]
    }]; #IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14 IOSTANDARD LVCMOS33 } [get_ports { an[5]
    }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2 IOSTANDARD LVCMOS33 } [get_ports { an[6]
    }]; #IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13 IOSTANDARD LVCMOS33 } [get_ports { an[7]
    }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]


##Buttons
set_property -dict { PACKAGE_PIN N17 IOSTANDARD LVCMOS33 } [get_ports { rst
    }]; #IO_L9P_T1_DQS_14 Sch=btnc
```

Listing 19: constraints.xdc

6.2 模拟版本代码

模拟版本代码中 cpu 和 ddu 以及一些模块有所不同，添加了更多的 wire output 取显示更多的调试信息。
下面仅贴出不同部分的代码，其余相同部分不再重复复制。

```
`include "define.v"
`timescale 1ns / 1ps

module cpu (input wire clk,
          input wire rst,
          output [`REG_BUS] pcout,
          input [`REG_BUS] inputaddr,
          output [`REG_BUS] memout_show,
          output [`REG_BUS] regout_show,
          output wire pc_write,
          output wire [`STATE_BUS] state,
          output wire [`REG_BUS] numatest,
          output wire [`REG_BUS] numbtest,
          output wire [`REG_BUS] irrun);

    wire [`NUM1_BUS] num1;
    wire [`NUM2_BUS] num2;
    wire [`NUM3_BUS] num3;
    wire [`NUM4_BUS] num4;
    wire [`NUM5_BUS] num5;
    wire [`NUM6_BUS] num6;

    wire [`REG_BUS] jext;
    wire [`REG_BUS] aluout;
    wire [`REG_BUS] numa;
    wire [`REG_BUS] numb;
    wire [`REG_BUS] alu_numa;
    wire [`REG_BUS] alu_numb;
    wire [`REG_BUS] memaddr;
    wire [`REG_BUS] memmdr;
    wire [`REG_BUS] regmem;
    wire [`REG_BUS] immext;
    wire [`REG_BUS] memdata;

    wire [`EXE_BUS] aluexe;
    wire [`OP_BUS] opcode;

    wire ins_or_data;
    wire pc_write_cond;
    wire pc_write;
    wire alu_srca;
    wire alu_srcb;
    wire ir_write;
    wire mem_write;
    wire mem_read;
    wire is_zero;
    wire mem_to_reg;
    wire reg_write;
    wire regdst;
    wire [1:0] pc_source;
    wire [1:0] alu_op;
    wire [`REG_BUS] irrun;
    wire [`STATE_BUS] state;

    opj_extend jexter (.ins (num5), .pc (pcout), .extd (jext));
    pc_reg pc (.clk (clk), .rst (rst), .pccond (pc_write_cond), .pcsource (
        pc_source), .aluinfo(is_zero), .pc_write(pc_write), .immpc (num3), .
        address(jext), .pcout (pcout));
    mux2 mux_ins_or_data (.select (ins_or_data), .route1 (aluout >> 2), .route2
         (pcout >> 2), .outdata (memaddr));
    mem memory (.clk(clk), .memwrite (mem_write), .indata (numb), .addr (
        memaddr), .memdata (memdata), .dpra (inputaddr), .dpo (memout_show));
    ins_split ir (.ir_write (ir_write), .ins (memdata), .opcode (opcode), .num1
         (num1), .num2 (num2) , .num3 (num3), .num4 (num4), .num5 (num5), .num6
        (num6) ,.irrun (irrun));
    mdr memory_data_reg (.memread (mem_read), .memdata (memdata), .memout (
        memmdr));
    control ctrl (.clk (clk), .rst (rst), .op (opcode), .pc_write_cond (
        pc_write_cond), .pc_write (pc_write), .ins_or_data (ins_or_data), .
        mem_read(mem_read), .mem_write (mem_write), .ir_write(ir_write), .
        pc_source(pc_source), .alu_op (alu_op), .alu_srca (alu_srca), .alu_srcb
         (alu_srcb), .reg_write (reg_write), .regdst (regdst), .mem_to_reg (
        mem_to_reg), .state_out (state));
    mux2 reg_write_mem (.select (mem_to_reg), .route1(memmdr), .route2(aluout),
         .outdata (regmem));
    regfile regs (.clk (clk), .rs (num1), .rt (num2), .rd (num4), .numa (numa),
         .numb (numb), .reg_write (reg_write), .regdst (regdst), .indata (
        regmem), .regaddr (inputaddr), .reg_out(regout_show));// , .reg_clone(
        regshow));
    mux2 muxnuma (.select (alu_srca), .route1 (pcout), .route2 (numa) , .
        outdata (alu_numa));
    extend extder (.ins (num3), .extd (immext));
    mux2 muxnumb (.select (alu_srcb), .route1 (immext), .route2(numb), .outdata
         (alu_numb));
    alu_ctrl aluexe_maker (.opcode (opcode), .funt (num6), .alu_op (aluexe));
    alu alucalc (.aluop (aluexe), .numa (alu_numa), .numb(alu_numb), .num_out (
        aluout) , .iszero (is_zero));

    assign numatest = alu_numa;
    assign numbtest = alu_numb;
endmodule
```

Listing 20: cpu.v

```
`timescale 1ns / 1ps
`include "define.v"

module ddu(input cont,
          input rst,
          input clk_board,
          input step,
          input mem,
          input inc,
          input dec,
          output reg [7:0] addrout_8,
          output wire [7:0] pc_8,
          output pc_write,
          output wire [`STATE_BUS] state,
          output wire [`REG_BUS] irrun,
          output wire [6:0] seg,
          output wire [7:0] an,
          output wire [1023:0] reg_clone,
          output [`REG_BUS] numa,
          output [`REG_BUS] numb,
          output wire [`REG_BUS] memshow);

    wire [`REG_BUS] memshow;
    reg [`REG_BUS] addrout;
    wire [`REG_BUS] pc;

    reg [`REG_BUS] addr = 0;
    wire [`REG_BUS] mem1;
    wire [`REG_BUS] mem2;
    reg clk;
    reg delay;
    reg incdelay;
    reg decdelay;

    always @(posedge clk_board or negedge rst)
    begin
        if (rst) begin
```

```verilog
            addr <= 0;
            incdelay = 1;
            decdelay = 1;
            delay   = 0;
            end else begin

            if (!incdelay && inc) addr = addr + 1;
            if (!decdelay && dec) addr = addr - 1;

            incdelay = inc;
            decdelay = dec;

            addrout             = addr;
            if (!delay && step) clk = 1;
            else clk            = 0;

            delay = step;

        end
    end
    assign memshow = mem ? mem1 :mem2;

    cpu CPU_sim (.clk (clk), .rst (rst), .pcout (pc), .inputaddr(addr) , .
        memout_show (mem1),.regout_show(mem2) , .regshow(reg_clone)
    ,.state(state) , .numatest(numa) , .numbtest(numb), .pc_write(pc_write) ,.
        irrun (irrun));

    always @* addrout_8 = {addr [7:0]};

    assign pc_8 = {pc[9:2]};

    segout segmental (.clk (clk_board), .indata (memshow), .an(an) ,.seg(seg));
endmodule
```

Listing 21: ddu.v

```verilog
`include "define.v"
`timescale 1ns / 1ps

module regfile(input wire clk,
               input wire [`RS_BUS] rs,
               input wire [`RT_BUS] rt,
               input wire [`RD_BUS] rd,
               input wire reg_write,
               input wire regdst,
               input wire [`REG_BUS] indata,
               output reg [`REG_BUS] numa,
               output reg [`REG_BUS] numb,
               input wire [`REG_BUS] regaddr,
               output reg [`REG_BUS] reg_out,
               output reg [1023:0] reg_clone);

```

```
    reg [`REG_BUS] register [0:31];
    integer i;

    initial begin
        for (i = 0; i < 32 ; i = i + 1) begin
            register[i] = 0;
        end
    end

    always @* begin
        reg_clone = 0;
        for (i = 31; i > = 0 ; i = i - 1) begin
            reg_clone = (reg_clone << 32) | register[i];
        end
    end

    always @(posedge clk) begin
        if (reg_write) begin
            if (regdst) begin
                register[rd] = indata;
                end else begin
                register[rt] = indata;
            end
        end

        numa = register [rs];
        numb = register [rt];
    end
    always@* reg_out = register [(regaddr % 31)];
endmodule
```

Listing 22: regfile.v

```
`include "define.v"
`timescale 1ns / 1ps

//* Controler
//* input with op
//* output control information

module control (input wire clk,
                input wire rst,
                input wire [`OP_BUS] op,        // * input the opcode part
                output reg pc_write_cond,       // * control pc
                output reg pc_write,            // * control pc
                output reg ins_or_data,         // * control mem
                output reg mem_read,            // * control mem
                output reg mem_write,           // * control mem
                output reg ir_write,            // * control ir
                output reg [1:0] pc_source,     // * control pc source
                output reg [1:0] alu_op,        // * control alu
                output reg alu_srca,            // * control numa
```

```
            output reg alu_srcb,          // * control numb
            output reg reg_write,         // * control regfile writable
            output reg regdst,
            output reg mem_to_reg,
            output reg [`STATE_BUS] state_out);

    reg [`STATE_BUS] state,next_state;

    initial begin
        state        = `IF_STATE;
        pc_write_cond = 0;
        pc_write     = 0;
        ins_or_data = 0;
        mem_read     = 0;
        mem_write    = 0;
        ir_write     = 0;
        pc_source    = 0;
        alu_op       = 0;
        alu_srca     = 0;
        alu_srcb     = 0;
        reg_write    = 0;
        regdst       = 0;
    end

    always @*
    state_out = state;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            state <= `IF_STATE;
            end else begin
            state <= next_state;
        end
    end

    always @* begin
        case (state)
            `IF_STATE :next_state = `ID_STATE;
            `ID_STATE :begin
                case (op)
                    `OP_LW : next_state = `EXE1_STATE;
                    `OP_SW : next_state = `EXE1_STATE;
                    `OP_BEQ :next_state = `EXE3_STATE;
                    `OP_BNE :next_state = `EXE3_STATE;
                    `OP_J  : next_state = `EXE4_STATE;
                    default :next_state = `EXE2_STATE;
                endcase
            end
            `EXE1_STATE :begin
                if (op == `OP_LW)
                    next_state = `MEM1_STATE;
                else
                    next_state = `MEM2_STATE;
            end
            `EXE2_STATE :next_state = `WB2_STATE;
            `EXE3_STATE :next_state = `IF_STATE;
            `EXE4_STATE :next_state = `IF_STATE;
            `MEM1_STATE :next_state = `WB1_STATE;
            `MEM2_STATE :next_state = `IF_STATE;
            `WB1_STATE :next_state = `IF_STATE;
            `WB2_STATE :next_state = `IF_STATE;
        endcase
    end

    always @* begin
        // * if state == > initial state
        if (state == `IF_STATE) begin
            pc_write_cond = 0;
            pc_write    = 1; // * pc writable
            ins_or_data = 0;
            mem_read    = 1; // * read mem for instructions
            mem_write   = 0;
            ir_write    = 1; // * ir writable
            pc_source   = 2'b00;
            alu_op      = 0;
            alu_srca    = 0;
            alu_srcb    = 0;
            reg_write   = 0;
            regdst      = 0;
        end

        // * id state
        if (state == `ID_STATE) begin
            pc_write = 0;
            ir_write = 0;
            mem_read = 0;
        end

        // * exe1 state
        if (state == `EXE1_STATE) begin
            alu_srca = 0;
            alu_srcb = 1;
            alu_op  = 2'b00;
        end

        // * exe2 state
        if (state == `EXE2_STATE) begin
            alu_srca = 0;
            alu_srcb = (op == `OP_R_TYPE ? 0 :1);
            alu_op  = 2'b10;
        end

```

```verilog
        // * exe3 state
        if (state == `EXE3_STATE) begin
            alu_srca     = 1;
            alu_srcb     = 0;
            alu_op       = 2'b01;
            pc_write     = 1;
            pc_write_cond = (op == `OP_BEQ ? 1 :0);
            pc_source    = 2'b01;
        end

        // * exe4 state
        if (state == `EXE4_STATE) begin
            pc_write = 1;
            pc_source = 2'b10;
        end

        // * mem1 state
        if (state == `MEM1_STATE) begin
            mem_read   = 1;
            ins_or_data = 1;
        end

        // * mem2 state
        if (state == `MEM2_STATE) begin
            mem_write  = 1;
            ins_or_data = 1;
        end

        // * wb1 state
        if (state == `WB1_STATE) begin
            regdst    = 0;
            reg_write = 1;
            mem_to_reg = 1;
        end

        // * wb2 state
        if (state == `WB2_STATE) begin
            regdst    = (op == `OP_R_TYPE ? 1 :0);
            reg_write = 1;
            mem_to_reg = 0;
        end
    end
endmodule
```

Listing 23: control.v

```verilog
`timescale 1ns / 1ps
`include "define.v"

module cpu_sim1();

    reg   rst   ;
    reg   clk  ;
    reg [`REG_BUS] dpra;
    wire [`REG_BUS] dpo;
    wire [`REG_BUS] pcout;
    wire [`REG_BUS] regdata;

    reg cont;
    reg step;
    reg mem;
    reg inc;
    reg dec;

    wire [`REG_BUS] pc;
    wire clkout;
    wire [`REG_BUS] addrout;
    wire [`REG_BUS] memdata;
    wire delay1;
    wire delay2;
    wire [`NUM5_BUS] num5;
    wire [`REG_BUS] irrun;
    wire [`REG_BUS] numa;
    wire [`REG_BUS] numb;
    initial
    begin
       rst  <= 0;
       cont <= 0;
       inc  <= 0;
       dec  <= 0;
       mem  <= 0;
       step <= 0;
       #10 rst = 1;
       #10 rst = 0;
       #20 step = 1;
       #10 step = 0;
       #10 mem = 1;
       inc     = 1;
       #10 inc = 0;
       #10 inc = 1;
       #10 inc = 0;
       #10 inc = 1;
       #10 inc = 0;
       #10 inc = 1;
       #10 inc = 0;
       #10 inc = 1;
       #10 inc = 0;
       #10 inc = 1;
       #10 inc = 0;
       #10 inc = 1;
       #10 inc = 0;
       #10 inc = 1;
       #10 inc = 0;
```

```verilog
        #10 step = 1;
        #10 step = 0;
        #10 step = 1;
        #10 step = 0;
        #10 step = 1;
        #10 step = 0;
        #10 step = 1;
        #10 step = 0;
        #10 step = 1;
        #10 step = 0;
        #10 step = 1;
        #10 step = 0;
        #10 step = 1;
        #10 step = 0;
        #10 step = 1;
        #10 step = 0;
        #10 step = 1;
        #10 step = 0;
        #10 step = 1;
        #10 step = 0;
        #10 step = 1;
        #10 step = 0;
        #10 step = 1;
        #10 step = 0;
        #10 step = 1;
        #10 step = 0;
        #10 step = 1;
        #10 step = 0;
        #10 step = 1;
        #10 step = 0;
        #10 step = 1;
        #10 step = 0;
        #10 step = 1;
        #10 step = 0;
        #10 mem = 0;
        #10000;
    end

    initial clk <= 0;
    always #5 clk <= ~clk;
    always begin #10 step = 1; #20 step = 0; end

    wire pc_write;
    wire [`STATE_BUS] state;

    wire [7:0] addrout_8;
    wire [7:0] pc_8;

    wire [6:0] seg;
    wire [7:0] an;
    wire [1023:0] reg_clone;
    wire reg_write;
    wire [31:0] memmdr;
    wire [31:0] mem2;

    ddu DUT (cont,rst,clk,step,mem,inc,dec,addrout_8,pc_8,pc_write,state,irrun,
        seg,an,mem2,numa,numb,regclone);
endmodule
```

Listing 24: cpu_sim1.v

7 总结

本次实验相当于一次比较综合的复习，复习了我们学习 COD 至多周期的各种知识。本次实验对于我来说最大的困难就是如何将程序正确的烧写到板子上。事实上，我只用了 3 天就完成了正确的模拟和仿真，但是到最终完成花费了整整 3 周的时间。具体原因在于对于时序逻辑和组合逻辑的写法规范性。

举个例子，再我 extend 的模块中，本来应该用 wire 就可以完成的任务，就不必多加一个 reg 进行中介保存，虽然理论上有着一样的效果，但是实际上，这样的代码过多会倒置无意义的锁存现象出现，从而出现各种诡异的 bug 出现。此外另一个困扰我整整一周的实验 bug 是板载时钟周期过短，无法再一个时钟周期内完成应该执行完成的任务，出现了神奇的 bug。

最终，实验也再截止日期前完成，总体来说还算顺利。

8 意见和建议

本次实验个人感觉，难度并不是很大，但是再我们烧写板子的时候，比较考研问题分析解决能力和动手能力。个人建议，之后实验助教要求检查的只有我们的仿真模拟，至于烧录可以考虑作为附加分处理。

此外，我决定我们之后可以增加一次关于流水实现的实验，也是只要求模拟，不要求仿真。考虑流水的复杂性和代码量，个人认为，设计流水 CPU 再多周期之后，并且要求同学们合作完成。这样可以达到更好的效果。

最后表示，老师还是最好不要删除单周期的实验，第一在于……通知太晚，我都写完了才说取消实验。第二在于，单周期作为复习也是很有必要的。而且，如果单周期的实验吐过顺利完成，之后的多周期会少很多问题，例如办在诗中的处理和锁存器等等。