组成原理课程第 七 次实报告

实验名称：riskv cpu 设计

学号：2310420 姓名：王晶 班次：周二三四节

实验目的

1.加深对 RISC-V 指令集体系结构的理解

2.掌握指令的取指、译码、执行、访存和写回等五个基本阶段的实现方式

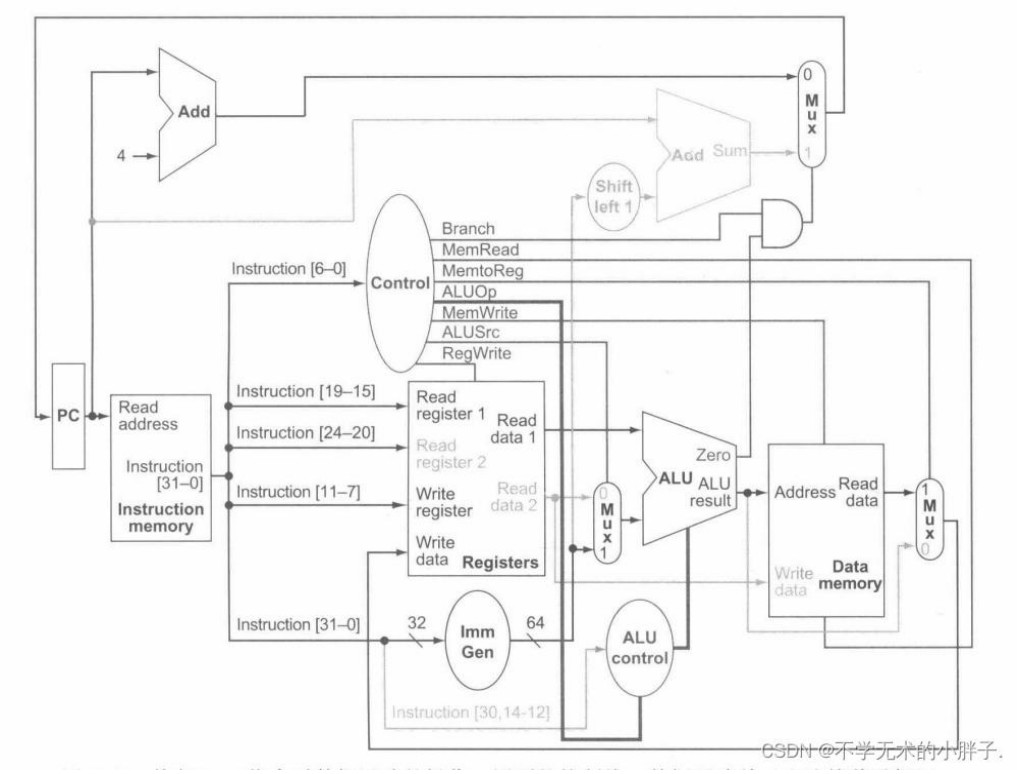
3.培养使用硬件描述语言进行模块化设计与调试的能力

4.同时为后续多周期或流水线 CPU 的学习与扩展奠定基础。

实验内容

尝试制作一个简单的 RISC-V 指令集的 CPU，简单单周期即可，在实验报告中整理总结实现思路和过程

实验原理图



实验步骤

（1）单周期 CPU

按照实验原理图中的架构，在单周期 CPU 中，我们设计的模块有 control 模块，pc 寄存器模块，加法器模块，寄存器堆模块，符号拓展模块，alu 模块，指令存储器，数据存储器，以及不同位置的多路选择器模块，除了具体功能模块，我们还需要顶层模块等

1.1 实验代码（见附录）

1.1.1 整体框架

top 模块是整个单周期 CPU 的顶层模块，主要完成以下功能：

实例化 rv32i（核心 CPU 控制与数据通路模块）

实例化了 instr_rom（指令存储器，用于取指;使用 pc[9:0] 地址访问指令存储器，输出 32 位指令 inst;只读）

实例化了 data_ram（数据存储器，用于访存;地址为 alu_result[9:0];写数据为 reg_data_2;写控制为 memwrite;读数据输出为 mem_rdata）

并通过 wire 信号将控制信号和数据路径信号从核心模块 rv32i 输出到各个子模块之间，构成完整的数据通路。

1.1.2 具体实现

——取指

使用 pc 寄存器决定当前执行的指令地址。instr_rom 用 pc[9:0] 作为地址

（意味着地址空间为 1024 条指令）。输出 inst 给 rv32i 处理。

——译码

将 inst 解码为以下控制信号：控制类：jump, branch, alu_a_src, memtoreg, memwrite, regwrite 扩展类型：extop（表示立即数提取的方式，如 I、S、B、U、J）ALU 控制：alu_ctr（指明是哪种操作：加法、减法、逻辑等）提取源寄存器索引，访问寄存器堆，输出：reg_data_1：源寄存器 1 的值。reg_data_2：源寄存器 2 的值

——立即数生成

根据 extop 和 inst 提取不同类型的立即数，输出 imm。

——执行

ALU 输入：alu_srcA 来自 reg_data_1 或 pc；alu_srcB 由 alu_b_src 控制（选择 reg_data_2 或 imm）；ALU 操作由 alu_ctr 决定，输出 alu_result

——访存

当执行 lw/sw 等访存指令时：alu_result 作为地址 reg_data_2 作为写入数据（sw）控制信号 memwrite 控制写使能；mem_rdata 是 lw 的读结果

——写回

根据 memtoreg 控制信号决定是否将 mem_rdata 写入寄存器，否则写入 alu_result。写入的最终数据为 reg_data_write，目标寄存器由译码阶段决定 regwrite 控制写使能

1.2 实验结果分析

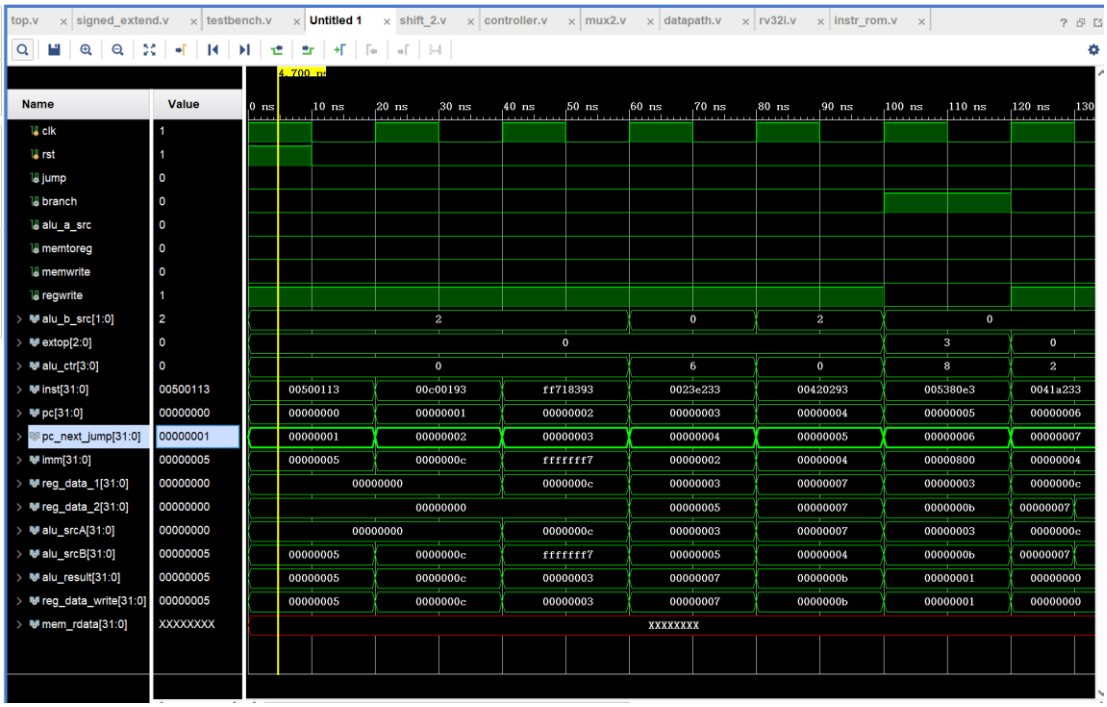
我们实验测试时采用的指令如下

```
// start:
memory[0] = 32'h00500113; // addi $2, $0, 5
memory[1] = 32'h00c00193; // addi $3, $0, 12
memory[2] = 32'hff718393; // addi $7, $3, -9
memory[3] = 32'h0023e233; // or $4, $7, $2
memory[4] = 32'h00420293; // addi $5, $4, 4
memory[5] = 32'h005380e3; // beq $5, $7, end
memory[6] = 32'h0041a233; // slt, $4,$3,$4
memory[7] = 32'h00400163; // beq $4, $0, around
memory[8] = 32'h00500393; // add $7, $3, $2

// around:
memory[9] = 32'h0023a233;  // slt, $4,$7,$2
memory[10] = 32'h0471a223; // sw,$7, 68($3)
memory[11] = 32'h05002103; // lw, $2, 80($0)
memory[12] = 32'h0020046f; // jal $8, end
memory[13] = 32'h0024a233; // sltu, $2,$3,$4

// end:
memory[14] = 32'h0000f137; // lui, $2, 15 =>0000f000
memory[15] = 32'h04202a23; // sw, $2, 54($0)
```

分析指令的执行过程

首先是$2=5,$3=12,$7=$3-9=3,$4=$7or$2=7,$5=11,第六条指令时，$5!=$7,则不跳转，($3 > $4)则$4=0，从而跳转到 around 处，$7 < $2，则$4=1，MEM[80]=3，$2 = 3，$8 = 52 且跳转到地址 end = 14，$2 = 0x0000f000，MEM[54] = $2 = 0x0000f000

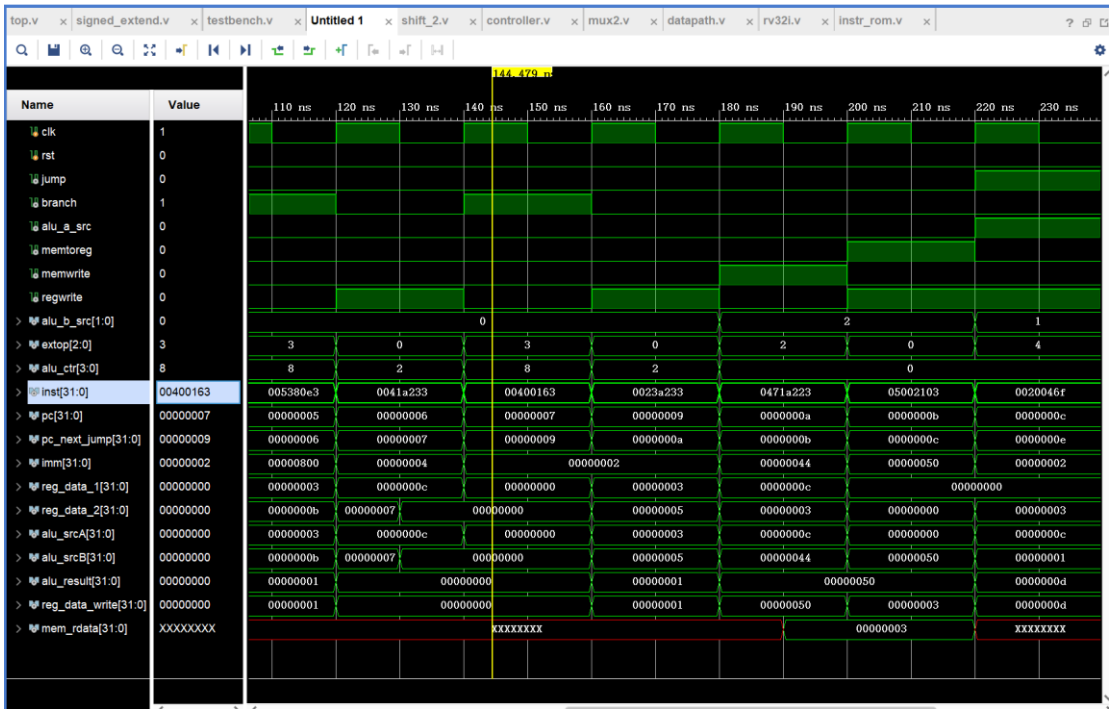
分析仿真结果



addi $2, $0, 5

可以看到，当 inst=00500113 时，pc=00000000，为当前指令 next_pc=00000001，为下一条指令，imm 为符号拓展的立即数为 1，从寄存器堆读出的两个寄存器为$2,$0，此时的值均为 0，alu_src 为输入 alu 的两个操作数，一个为 0，一个为 5，alu_result 为 alu 运算的结果，即为 5，reg_data_write 为写入寄存器的值，为 5，mem_rdata 为从数据存储器读出的数据，该条指令中不读出数据，置为 X

后面几条指令类似，我们不再重复分析



beq $4, $0, around

分析这条指令，由于$4=0,则会发生跳转，跳转到 around 处，可以看到 inst=00400163,即该条指令，pc 为该条指令的地址，但是 nextpc 为 around 处的地址，为 00000009，从寄存器取出的$4 $0 均为 0，输入 alu 的两个操作数也为 0，做一个减法后，alu_result 为 0，写入寄存器堆的数据为 0，不从数据存储器读数据

（2）多周期 CPU

2.1 实验代码（见附录）

1.1.1 整体框架

顶层模块 MC_RV32Core 是整个五级流水线 CPU 的控制中心，主要功能如下：

实例化控制模块：ControlUnit（ControlUnit.sv）控制信号生成器：根据指令类型产生 jump、branch、memwrite、regwrite 等控制信号。

实例化运算与判断模块：ALU（ALU.sv）：算术逻辑单元：执行加减、与或异或、比较等操作。BranchUnit（BranchUnit.sv）分支判断逻辑：判断分支条件是否成立。

实例化存储与扩展模块：DataMem（DataMem.sv）数据存储器：实现对数据内存的读写操作。Data_Ext（Data_Ext.sv）数据扩展器：对访存读取的数据进行符号扩展（如 lb -> sign-extend 32 位）。RegisterFile（RegisterFile.sv 通用寄存器堆：提供 32 个寄存器的读写接口。

实例化指令与地址处理模块：ImmUnit（ImmUnit.sv）立即数生成器：根据 inst 提取 I/S/B/U/J 类型的立即数。InstrMem（InstrMem.sv）指令存储器：根据 PC 地址读取指令。NPC_Generator（NPC_Generator.sv）下一条指令地址生成器：根据跳转、分支等情况生成新 PC。

2.1.2 具体实现

——取指阶段

InstrMem：指令存储器。NPC_Generator：下一 PC 地址生成器

当前 PC 输入给 InstrMem，取出当前指令 inst。NPC_Generator 根据控制信号（如 jump、branch）计算下一条指令地址（PC + 4、跳转目标等）

——译码阶段

ControlUnit：生成控制信号/RegisterFile：读出源寄存器 rs1、rs2/ImmUnit：立即数生成/BranchUnit：判断分支条件

指令送入 ControlUnit，生成诸如 regwrite, branch, jump, aluop,memwrite, memtoreg 等信号；寄存器堆 RegisterFile 读出 rs1、rs2 的数据；ImmUnit 根据 inst 中编码规则提取对应立即数；若是分支指令，BranchUnit 判断是否满足跳转条件

——执行阶段

ALU：执行加减、逻辑运算等/BranchUnit（部分）：分支目标地址计算/NPC_Generator：生成跳转地址（如果跳转成立）

ALU 输入来自 rs1 和 rs2 或立即数 imm，控制信号指明是加减、逻辑、比较等操作；若为跳转/分支，计算跳转目标地址；结果传入下一流水线级进行访存或写回

——访存阶段

DataMem：数据存储器/Data_Ext：对读取数据进行扩展

对于 load/store 指令，使用 DataMem 进行数据读写；若为 load 指令，使用 Data_Ext 对读取结果进行符号/零扩展；存储地址来自 EX 阶段 ALU 计算结果

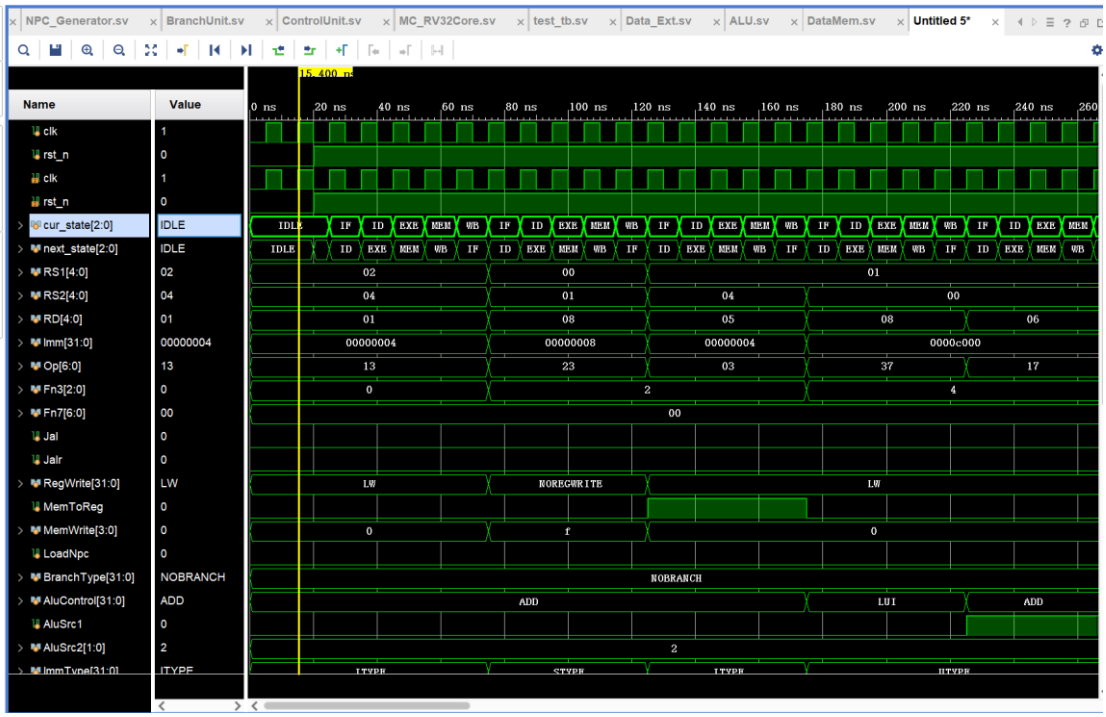
写入数据来自寄存器 rs2

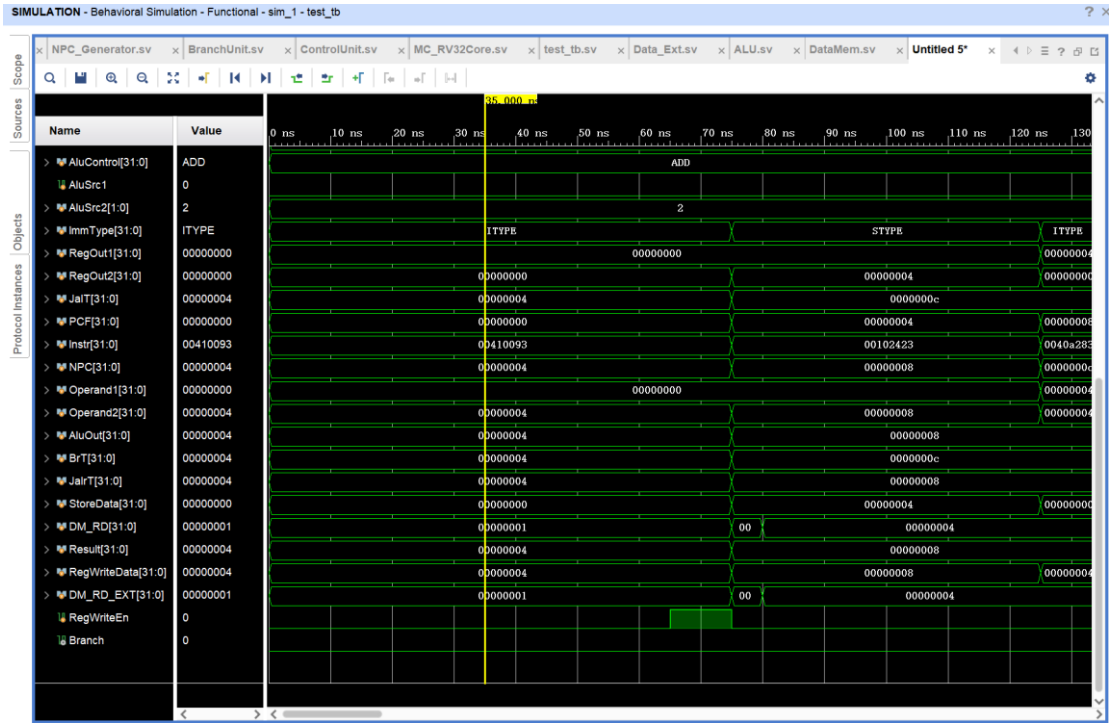
——写回阶段

RegisterFile：目标寄存器写入

根据控制信号 memtoreg 决定写回值是：从 Data_Ext（load 指令）或是 ALU 结果（算术逻辑指令）；写回 rd 寄存器，完成该指令周期处理

2.2 实验结果分析





我们测试的指令为

```
addi x1,x2,4
SW x1,8(x0)
lw x5,4(x1)
lui x8,12
auipc x6,12
add x2,x1,x5
beq x1,x5,L1
add x1,x1,x5
L1:jal x1,L2
add x1,x1,x5
add x1,x1,x5
L2:sub x8,x1, x5
slli x10,x8,2
sra x12,x10,x1
sub x7,x0,x1
srai x7,x7,1
jalr x16,68(x5)
add x1,x1,x1
andi x16,x16,1
```

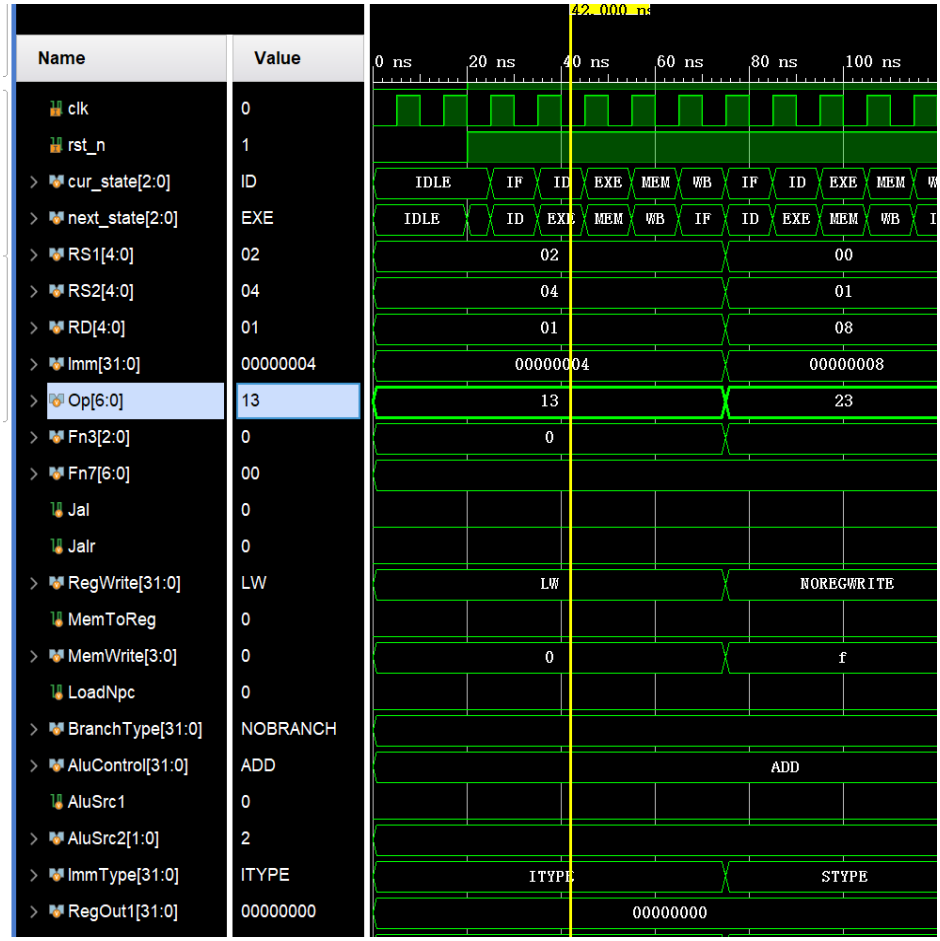
以第一条指令 addi x1,x2,4 的分析为例

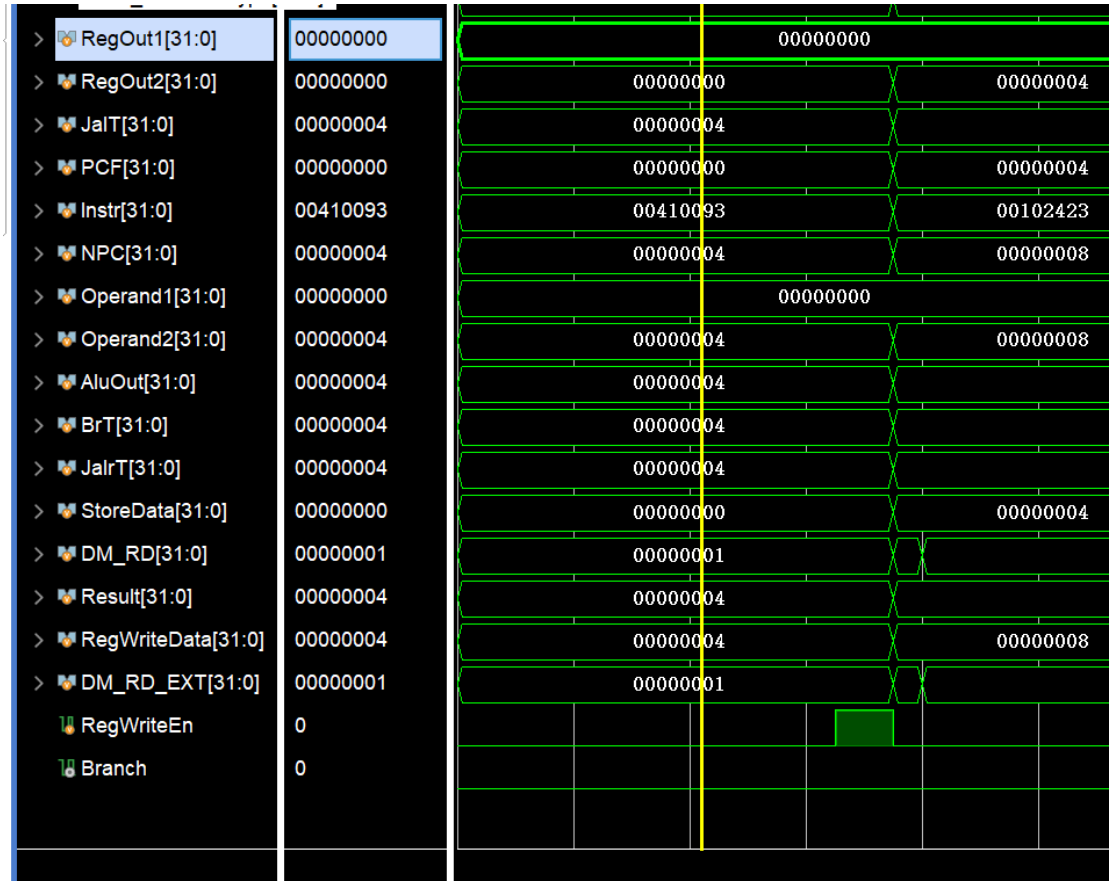
需要用到 1 寄存器为目的寄存器，2 寄存器为操作数寄存器，立即数为 4，需要

符号拓展，解析出的 opcode 为 13，funct 为 0，jal jalr 等信号都为 0，regwrite 为 LW，表示来自寄存器的控制器，正在执行 LW，memtoreg 为写回数据选择，此时为 0，表明来自 ALU 的输出，memwrite 为 0，表示不写入内存，loadnpc 为 0，表明将下一条指令写入 PC，Branchtype 为不分支跳转，alucontrol 为 add，alusrc 为 ALU 输入来源信号，RegOut1, RegOut2 为从寄存器堆读出的两个寄存器的值，此处均为 0；Operand1, Operand2 为 ALU 的两个操作数输入值，当前均为 0；AluOut 是 ALU 输出结果，当前为 00000004

DM_RD 是从数据存储器（Data Memory）读出的数据，当前为 00000001；StoreData 为要写入数据存储器的数据（来自 rs2），此处为 00000000；PCF 为当前的程序计数器 PC 值（旧 PC），此处为 00000000；NPC 为下一条指令的 PC 值（Next PC），此处为 00000004；BrT 为分支目标地址（Branch Target），此处为 00000004；JalT 为无条件跳转目标地址（JAL Target）00000004

instr 为当前正在执行的指令机器码，为 00410093，result 为 ALU 的运算结果为 4，regwritedata 为写回寄存器的值，为 4





附录部分

（1）单周期 CPU

top.v

```
module top(
        input wire clk, rst,
        output wire jump, branch, alu_a_src, memtoreg, memwrite,
regwrite,
        output wire [1:0] alu_b_src,
        output wire [2:0] extop,
        output wire [3:0] alu_ctr,
        output wire [31:0] inst,
        output wire [31:0] pc, pc_next_jump,
        output wire [31:0] imm, reg_data_1, reg_data_2,
        output wire [31:0] alu_srcA, alu_srcB, alu_result,
        output wire [31:0] reg_data_write,
        output wire [31:0] mem_rdata
```

```
    );

    rv32i CPU(
              .clk(clk),
              .rst(rst),
              .mem_rdata(mem_rdata),
              .inst(inst),

              .jump(jump),
              .branch(branch),
              .alu_a_src(alu_a_src),
              .memtoreg(memtoreg),
              .memwrite(memwrite),
              .regwrite(regwrite),
              .alu_b_src(alu_b_src),
              .extop(extop),
              .alu_ctr(alu_ctr),

              .pc(pc),
              .pc_next_jump(pc_next_jump),
              .imm(imm),
              .reg_data_1(reg_data_1),
              .reg_data_2(reg_data_2),
              .alu_srcA(alu_srcA),
              .alu_srcB(alu_srcB),
              .alu_result(alu_result),
              .reg_data_write(reg_data_write)
          );


    // inst-ram
    instr_rom instr_rom(
                  .addr     (pc[9:0]),
                  .inst     (inst)
              );

    //ram
    data_ram data_ram(
                 //ports
                 .clk       ( ~clk                      ),
                 .addr            ( alu_result[9:0]),
                 .data_w          ( reg_data_2  ),
                 .mem_r_w         ( memwrite          ),
                 .data_r          ( mem_rdata )
```

```
            );
endmodule
```

rv32i.v

```
module rv32i(
        input wire clk, rst,
        input wire [31:0] mem_rdata, inst,
        output wire jump, branch, alu_a_src, memtoreg, memwrite,
regwrite,
        output wire [1:0] alu_b_src,
        output wire [2:0] extop,
        output wire [3:0] alu_ctr,
        output wire [31:0] pc, pc_next_jump,
        output wire [31:0] imm, reg_data_1, reg_data_2,
        output wire [31:0] alu_srcA, alu_srcB, alu_result,
        output wire [31:0] reg_data_write
    );

    controller controller(
                    .inst(inst),
                    .jump(jump),
                    .branch(branch),
                    .alu_a_src(alu_a_src),
                    .memtoreg(memtoreg),
                    .memwrite(memwrite),
                    .regwrite(regwrite),
                    .alu_b_src(alu_b_src),
                    .extop(extop),
                    .alu_ctr(alu_ctr)
                );

    datapath datapath(
                  .clk(clk),
                  .rst(rst),
                  .mem_rdata(mem_rdata),
                  .inst(inst),

                  .jump(jump),
              .branch(branch),
         .alu_a_src(alu_a_src),
        .memtoreg(memtoreg),
     .regwrite(regwrite),                              胯祫
```

```
                .alu_b_src(alu_b_src),
                .extop(extop),                                  .alu_c
tr(alu_ctr),
                .pc(pc),
                .pc_next_jump(pc_next_jump),
                .imm(imm),
                .reg_data_1(reg_data_1),
                .reg_data_2(reg_data_2),
                .alu_srcA(alu_srcA),
                .alu_srcB(alu_srcB),
                .alu_result(alu_result),
                .reg_data_write(reg_data_write)

            );


endmodule
```

controller.v

```
module controller(
        input wire [31:0] inst,
        output wire jump,
        branch,
        alu_a_src,
        memtoreg,
        memwrite,
        regwrite,
        output wire [1:0] alu_b_src,
        output wire [2:0] extop,
        output wire [3:0] alu_ctr
    );

    wire [6:0] opcode = inst[6:0];
    wire [2:0] funct3 = inst[14:12];

    parameter [6:0] op_R_type = 7'h33;
    parameter [6:0] op_I_type_load        = 7'h03;
    parameter [6:0] op_I_type_alu        = 7'h13;
    parameter [6:0] op_I_type_jump        = 7'h6F;
    parameter [6:0] op_S_type = 7'h23;
    parameter [6:0] op_B_type = 7'h63;
```

```
    parameter [6:0] op_U_type_load        = 7'h37;

    assign branch = opcode == op_B_type;                // B-type

    assign jump = opcode == op_I_type_jump ;            // Jump

    assign alu_a_src = opcode == op_I_type_jump;        // Jump

    assign memtoreg = opcode == op_I_type_load;         // load

    assign regwrite = ( opcode == op_R_type )
           | ( opcode == op_I_type_alu)
           | ( opcode == op_U_type_load )
           | ( opcode == op_I_type_load )
           | ( opcode == op_I_type_jump );               // R-type
/ I-alu / lui / load / Jump

    assign memwrite = (opcode == op_S_type);               // S-
type

    assign alu_ctr[3] = ( opcode == op_U_type_load )
// lui / B-type
           | ( opcode == op_B_type );

    assign alu_ctr[2] = (( opcode == op_R_type )
// R-type / I-alu / lui
                         | ( opcode == op_I_type_alu )) &
funct3[2]
           | ( opcode == op_U_type_load) ;

    assign alu_ctr[1] = (( opcode == op_R_type )
                         | ( opcode == op_I_type_alu )) &
funct3[1]
           | ( opcode == op_U_type_load) ;

    assign alu_ctr[0] = (( opcode == op_R_type )
                         | ( opcode == op_I_type_alu )) &
funct3[0]
           | ( opcode == op_U_type_load) ;

    assign alu_b_src[1] = ( opcode == op_I_type_alu)
           | ( opcode == op_I_type_load)
           | ( opcode == op_U_type_load)
           | ( opcode == op_S_type);
```

```
    assign alu_b_src[0] = ( opcode == op_I_type_jump);

    assign extop[2] = ( opcode == op_I_type_jump);
    assign extop[1] = ( opcode == op_S_type) | ( opcode ==
op_B_type);
    assign extop[0] = ( opcode == op_U_type_load) | ( opcode ==
op_B_type);


endmodule
```

datapath.v

```
module datapath(
        input wire clk,
        rst,
        jump,
        branch,
        alu_a_src,
        memtoreg,
        regwrite,
        input wire [1:0] alu_b_src,
        input wire [2:0] extop,
        input wire [3:0] alu_ctr,
        input wire [31:0] mem_rdata, inst,
        output wire [31:0] pc, pc_next_jump,
        output wire [31:0] imm, reg_data_1, reg_data_2,
        output wire [31:0] alu_srcA, alu_srcB, alu_result,
        output wire [31:0] reg_data_write
    );

    wire [31:0] pc_srcB;
    wire zero, pcsrc, overflow;
    wire [4:0] reg_des = inst[11:7];
    wire [4:0] reg_source_1         = inst[19:15];
    wire [4:0] reg_source_2         = inst[24:20];

    assign pcsrc = ( zero & branch ) | jump ;

    // pc
    pc pc_(
           .clk     (clk),
```

```
        .rst     (rst),
        .din     (pc_next_jump),
        .q       (pc)
    );

// pu_plus
mux2 #(32) mux_pc_srcB(
         .a         (imm),
         .b         (32'd1),
         .s         (pcsrc),
         .y         (pc_srcB)
     );

adder pc_plus(
          .a  (pc),
          .b  (pc_srcB),
          .y  (pc_next_jump)
      );

// regfile
regfile regfile(
            .clk        (clk),
            .addr_a     (inst[19:15]),
            .addr_b     (inst[24:20]),
            .addr_d     (inst[11:7]),
            .data_a     (reg_data_1),
            .data_b     (reg_data_2),
            .data_d     (reg_data_write),
            .reg_w_en   (regwrite)
        );

// imm
extender imm_extender(
             .inst      (inst),
             .op_ext    (extop),
             .imm       (imm)
         );

// mux for alu_srcA
mux2 #(32) mux_alu_srcA(
         .a             (pc),
         .b             (reg_data_1),
         .s             (alu_a_src),
         .y             (alu_srcA)
```

```
        );

    mux3 mux_alu_srcB(
             .a             (reg_data_2),
             .b             (32'd1),
             .c             (imm),
             .s             (alu_b_src),
             .y             (alu_srcB)
        );

    // alu
    alu alu(
            //ports
            .data_a         ( alu_srcA ),
            .data_b         ( alu_srcB ),
            .opcode         ( alu_ctr),
            .alu_result     ( alu_result ),
            .overflow       ( overflow ),
            .zero           ( zero )
        );

    // mux2
    mux2 #(32) mux_result(
             .a         (mem_rdata),
             .b         (alu_result),
             .s         (memtoreg),
             .y         (reg_data_write)
         );

endmodule
```

pc.v

```
module pc(
    input wire clk,
    input wire rst,
    input [31:0] din,
    output reg [31:0] q
);

always @(posedge clk) begin
    if (rst) begin
        q <= 32'b0;
```

```
    end
    else begin
        q <= din;
    end
end
endmodule
```

mux2.v

```
module mux2 #(parameter WIDTH = 32 )(
   input wire [WIDTH - 1:0] a,
   input wire [WIDTH - 1:0] b,
   input wire s,
   output wire [WIDTH - 1:0] y
);

assign y = s ? a : b;

endmodule
```

adder.v

```
module adder(
    input  [31:0] a,
    input  [31:0] b,
    output [31:0] y
);

// assign {cout, s} = a + b + cin;
    assign y = a + b;
endmodule
```

regfile.v

```
module regfile (
        input wire clk,
        input [4:0] addr_a,
        input [4:0] addr_b,
        input [4:0] addr_d,
        input [31:0] data_d,
        input reg_w_en,

        output [31:0] data_a,
        output [31:0] data_b
```

```
);

    reg [31:0] registers[31:0];
        integer i;

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

    // read
        assign data_a = (addr_a==5'b0) ? 32'b0 :
registers[addr_a];
        assign data_b = (addr_b==5'b0) ? 32'b0 :
registers[addr_b];

    // write
        always @ (negedge clk) begin
                if (reg_w_en) begin
                        registers[addr_d] <= data_d;
                end
        end


endmodule
```

extender.v

```
module extender (
        input [31:0] inst,
        input [2:0] op_ext,
        output reg signed [31:0] imm
);

    wire signed [31:0] imm_I, imm_U,imm_B,imm_S, imm_J;
    assign imm_I = { {20{inst[31]}}, inst[31:20]};
        assign imm_U = { inst[31:12], 12'b0};
        assign imm_S = { {20{inst[31]}}, inst[31:25], inst[11:7]};
        assign imm_B = { {20{inst[31]}}, inst[7], inst[30:25],
inst[11:8], 1'b0}; // default pc+1
    assign imm_J = { {12{inst[31]}}, inst[19:12], inst[20],
inst[30:21], 1'b0};
```

```
        always @ (*) begin
                case(op_ext)
                        0: imm = imm_I;
                        1: imm = imm_U;
                        2: imm = imm_S;
                        3: imm = imm_B;
                        4: imm = imm_J;
                        default: imm = 0;
                endcase
        end
endmodule
```

mux2.v

```
module mux2 #(parameter WIDTH = 32 )(
   input wire [WIDTH - 1:0] a,
   input wire [WIDTH - 1:0] b,
   input wire s,
   output wire [WIDTH - 1:0] y
);

assign y = s ? a : b;

endmodule
```

mux3.v

```
module mux3 (
        input wire [31:0] a,
        input wire [31:0] b,
        input wire [31:0] c,
        input wire [1:0] s,
        output reg [31:0] y
    );

    always @ (*) begin
        case(s)
            2'b00:
                y = a;
            2'b01:
                y = b;
            2'b10:
                y = c;
            default:
```

```
                y = c;
        endcase
    end

endmodule
```

alu.v

```
module alu (
        input [31:0] data_a,
        input [31:0] data_b,

        input [3:0] opcode,

        output reg [31:0] alu_result,
        output overflow, zero

    );
    wire signed [31:0] data_a_signed = data_a;
    wire signed [31:0] data_b_signed = data_b;

    assign overflow = 1'b0;
    assign zero = data_a_signed == data_b_signed;

    always @ (*) begin
        case (opcode)
            4'b0000:
                alu_result = data_a + data_b;
            4'b0001:
                alu_result = data_a - data_b;
            4'b0010:
                alu_result = (data_a_signed < data_b_signed )? 1:
0;
            4'b0011:
                alu_result = (data_a < data_b )? 1: 0;
            4'b0100:
                alu_result = data_a ^ data_b;
            4'b0101:
                alu_result = data_a << data_b;
            4'b0110:
                alu_result = data_a | data_b;
            4'b0111:
                alu_result = data_a & data_b;
```

```
            4'b1000:
                alu_result = (data_a_signed < data_b_signed )? 1:
0;
            4'b1001:
                alu_result = (data_a < data_b )? 1: 0;
             4'b1111:
                alu_result = data_b;
            default:
                alu_result = 'b0;
        endcase
    end

endmodule
```

instr_rom.v

```
module instr_rom (
        input [9:0] addr,
        output [31:0] inst
);
        reg [31:0] memory[1023:0];

        assign inst = memory[addr];

        initial begin
        // Example program:
        // start:
        memory[0] = 32'h00500113; // addi $2, $0, 5
        memory[1] = 32'h00c00193; // addi $3, $0, 12
        memory[2] = 32'hff718393; // addi $7, $3, -9
        memory[3] = 32'h0023e233; // or $4, $7, $2
        memory[4] = 32'h00420293; // addi $5, $4, 4
        memory[5] = 32'h005380e3; // beq $5, $7, end
        memory[6] = 32'h0041a233; // slt, $4,$3,$4
        memory[7] = 32'h00400163; // beq $4, $0, around
        memory[8] = 32'h00500393; // add $7, $3, $2

        // around:
        memory[9] = 32'h0023a233;  // slt, $4,$7,$2
        memory[10] = 32'h0471a223; // sw,$7, 68($3)
        memory[11] = 32'h05002103; // lw, $2, 80($0)
        memory[12] = 32'h0020046f; // jal $8, end
        memory[13] = 32'h0024a233; // sltu, $2,$3,$4
```

```
        // end:
        memory[14] = 32'h0000f137; // lui, $2, 15 =>0000f000
        memory[15] = 32'h04202a23; // sw, $2, 54($0)

    end

endmodule
```

data_ram.v

```
module data_ram (
        input [9:0] addr,
        input [31:0] data_w,
        input mem_r_w,
        input clk,

        output [31:0] data_r
);
        reg [31:0] memory[1023:0];

        assign data_r = memory[addr];

        always @ (posedge clk) begin
        if (mem_r_w) begin
            memory[addr] <= data_w;
        end
    end
endmodule
```

signed_extend.v

```
module sign_extend (
   input wire [15:0] a,
   output wire [31:0] y
);

assign y = {{16{a[15]}}, a};
endmodule
```

shift_2.v

```
module sl2(
    input wire [31:0] a,
    output wire [31:0] y
```

```
);

assign y = {a[29:0], 2'b00};

endmodule
```

（2）多周期 CPU

MC_RV32Core.v

```
`timescale 1ns / 1ps
`include "Parameters.v"
import Params::*;

module MC_RV32Core(
    input logic clk,
    input logic rst_n
);

typedef enum bit [2:0] {
    IDLE,
    IF, ID, EXE, MEM, WB
} State;
State cur_state, next_state;

logic [4:0] RS1;
logic [4:0] RS2;
logic [4:0] RD;
logic [31:0] Imm;
logic [6:0] Op;
logic [2:0] Fn3;
logic [6:0] Fn7;
logic Jal;
logic Jalr;
LoadType RegWrite;
logic MemToReg;
logic [3:0] MemWrite;
logic LoadNpc;
BType BranchType;
AluOp AluControl;
logic AluSrc1;
logic [1:0] AluSrc2;
Type ImmType;
logic [31:0] RegOut1;
logic [31:0] RegOut2;
```

```
logic [31:0] JalT;

logic [31:0] PCF;
logic [31:0] Instr;
logic [31:0] NPC;

logic [31:0] Operand1;
logic [31:0] Operand2;
logic [31:0] AluOut;
logic [31:0] BrT;
logic [31:0] JalrT;

logic [31:0] StoreData;
logic [31:0] DM_RD;
logic [31:0] Result;

logic [31:0] RegWriteData;
logic [31:0] DM_RD_EXT;
logic RegWriteEn;

assign RS1 = Instr[19:15];
assign RS2 = Instr[24:20];
assign RD = Instr[11:7];
assign Op = Instr[6:0];
assign Fn3 = Instr[14:12];
assign Fn7 = Instr[31:25];

always_comb
begin
    if (cur_state == WB)
    begin
        if (RegWrite == NOREGWRITE)
            RegWriteEn = 0;
        else
            RegWriteEn = 1;
    end
    else
        RegWriteEn = 0;
end

always_ff @(posedge clk, negedge rst_n)
    if (~rst_n)
        PCF <= 0;
    else if (cur_state == WB)
```

```
        PCF <= NPC;

always_comb
begin
    if (AluSrc1 == 1'b1)
        Operand1 = PCF;
    else
        Operand1 = RegOut1;
end

always_comb
begin
    if (AluSrc2 == 2'b00)
        Operand2 = RS2;
    else if (AluSrc2 == 2'b01)
        Operand2 = RegOut2;
    else
        Operand2 = Imm;
end

assign StoreData = RegOut2;

always_comb
begin
    if (LoadNpc)
        Result = PCF + 4;
    else
        Result = AluOut;
end

always_comb
begin
    if (MemToReg)
        RegWriteData = DM_RD_EXT;
    else
        RegWriteData = Result;
end

assign JalT = Imm + PCF;
assign JalrT = AluOut;
assign BrT = Imm + PCF;

always_ff @(posedge clk, negedge rst_n)
    if (~rst_n)
```

```
        cur_state <= IDLE;
    else
        cur_state <= next_state;

always_comb
begin
    case (cur_state)
        IDLE:
            if (~rst_n)
                next_state = IDLE;
            else
                next_state = IF;
        IF: next_state = ID;
        ID: next_state = EXE;
        EXE: next_state = MEM;
        MEM: next_state = WB;
        WB: next_state = IF;
        default: next_state = IDLE;
    endcase
end

ControlUnit U1(
    .Op(Op),
    .Fn3(Fn3),
    .Fn7(Fn7),
    .JalD(Jal),
    .JalrD(Jalr),
    .RegWriteD(RegWrite),
    .MemToRegD(MemToReg),
    .MemWriteD(MemWrite),
    .LoadNpcD(LoadNpc),
    .BranchTypeD(BranchType),
    .AluControlD(AluControl),
    .AluSrc2D(AluSrc2),
    .AluSrc1D(AluSrc1),
    .ImmType(ImmType)
);

ALU U2(
    .Operand1(Operand1),
    .Operand2(Operand2),
    .AluControl(AluControl),
    .AluOut(AluOut)
);
```

```
BranchUnit U3(
    .Reg1(Operand1),
    .Reg2(Operand2),
    .BrType(BranchType),
    .BranchE(Branch)
);

DataMem U4(
    .clk(clk),
    .A(AluOut),
    .WD(StoreData),
    .WE(MemWrite),
    .RD(DM_RD)
);

Data_Ext U5(
    .LoadByteSelect(AluOut[1:0]),
    .RegWrite(RegWrite),
    .In(DM_RD),
    .Out(DM_RD_EXT)
);

ImmUnit U6(
    .ImmType(ImmType),
    .In(Instr),
    .Out(Imm)
);

InstrMem U7(
    .InstrAddr(PCF),
    .Instr(Instr)
);

NPC_Generator U8(
    .Jal(Jal),
    .Jalr(Jalr),
    .Branch(Branch),
    .JalT(JalT),
    .JalrT(JalrT),
    .BrT(BrT),
    .PC(PCF),
    .NPC(NPC)
);
```

```
RegisterFile U9(
    .clk(clk),
    .RS1(RS1),
    .RS2(RS2),
    .RegOut1(RegOut1),
    .RegOut2(RegOut2),
    .WD(RegWriteData),
    .RD(RD),
    .WE(RegWriteEn)
);

endmodule
```

cnotrolunit.v

```
`timescale 1ns / 1ps
`include "Parameters.v"
import Params::*;

module ControlUnit(
    input logic [6:0] Op,
    input logic [2:0] Fn3,
    input logic [6:0] Fn7,
    output logic JalD,
    output logic JalrD,
    output LoadType RegWriteD,
    output logic MemToRegD,
    output logic [3:0] MemWriteD,
    output logic LoadNpcD,
    output BType BranchTypeD,
    output AluOp AluControlD,
    output logic [1:0] AluSrc2D,
    output logic AluSrc1D,
    output Type ImmType
);

assign JalD = (Op == 7'b1101111) ? 1'b1 : 1'b0;
assign JalrD = (Op == 7'b1100111) ? 1'b1 : 1'b0;

always_comb
begin
    if (Op == 7'b0000011)
    begin
        case (Fn3)
```

```
            3'b000: RegWriteD = LB;
            3'b001: RegWriteD = LH;
            3'b010: RegWriteD = LW;
            3'b100: RegWriteD = LBU;
            3'b101: RegWriteD = LHU;
            default: RegWriteD = NOREGWRITE;
        endcase
    end
    else if (Op == 7'b0010011 || Op == 7'b0110011 || Op ==
7'b0110111 || Op == 7'b0010111 || Op == 7'b1101111 || Op ==
7'b1100111)
    begin
        RegWriteD = LW;
    end
    else
    begin
        RegWriteD = NOREGWRITE;
    end
end

always_comb
begin
    if (Op == 7'b0000011)
        MemToRegD = 1;
    else
        MemToRegD = 0;
end

always_comb
begin
    if (Op == 7'b0100011)
    begin
        case (Fn3)
            3'b000: MemWriteD = 4'b0001;
            3'b001: MemWriteD = 4'b0011;
            3'b010: MemWriteD = 4'b1111;
            default: MemWriteD = 4'b0000;
        endcase
    end
    else
        MemWriteD = 4'b0000;
end

assign LoadNpcD = (JalD | JalrD) ? 1'b1 : 1'b0;
```

```
always_comb
begin
    if (Op == 7'b1100011)
    begin
        case (Fn3)
            3'b000: BranchTypeD = BEQ;
            3'b001: BranchTypeD = BNE;
            3'b100: BranchTypeD = BLT;
            3'b101: BranchTypeD = BGE;
            3'b110: BranchTypeD = BLTU;
            3'b111: BranchTypeD = BGEU;
            default: BranchTypeD = NOBRANCH;
        endcase
    end
    else
        BranchTypeD = NOBRANCH;
end

always_comb
begin
    case (Op)
        7'b0000011: AluControlD = ADD;
        7'b0100011: AluControlD = ADD;
        7'b0110111: AluControlD = LUI;
        7'b0010111: AluControlD = ADD;
        7'b1100011: AluControlD = ADD;
        7'b1101111: AluControlD = ADD;
        7'b1100111: AluControlD = ADD;
        7'b0110011:
        begin
            case (Fn3)
                3'b000: AluControlD = (Fn7[5] == 0) ? ADD : SUB;
                3'b001: AluControlD = SLL;
                3'b010: AluControlD = SLT;
                3'b011: AluControlD = SLTU;
                3'b100: AluControlD = XOR;
                3'b101: AluControlD = (Fn7[5] == 0) ? SRL : SRA;
                3'b110: AluControlD = OR;
                3'b111: AluControlD = AND;
                default: AluControlD = ADD;
            endcase
        end
        7'b0010011:
```

```
        begin
            case (Fn3)
                3'b000: AluControlD = ADD;
                3'b010: AluControlD = SLT;
                3'b011: AluControlD = SLTU;
                3'b100: AluControlD = XOR;
                3'b110: AluControlD = OR;
                3'b111: AluControlD = AND;
                3'b001: AluControlD = SLL;
                3'b101: AluControlD = (Fn7[5] == 0) ? SRL : SRA;
                default: AluControlD = ADD;
            endcase
        end
        default: AluControlD = ADD;
    endcase
end

always_comb
begin
    if (Op == 7'b0010111)
        AluSrc1D = 1;
    else
        AluSrc1D = 0;
end

always_comb
begin
    if (Op == 7'b0010011 && (Fn3 == 3'b001 || Fn3 == 3'b101))
        AluSrc2D = 2'b00;
    else if (Op == 7'b1100011 || Op == 7'b0110011)
        AluSrc2D = 2'b01;
    else
        AluSrc2D = 2'b10;
end

always_comb
begin
    case (Op)
        7'b0110011: ImmType = RTYPE;
        7'b0010011: ImmType = ITYPE;
        7'b0000011: ImmType = ITYPE;
        7'b1100111: ImmType = ITYPE;
        7'b1100011: ImmType = BTYPE;
        7'b0100011: ImmType = STYPE;
```

```
        7'b1101111: ImmType = JTYPE;
        7'b0110111: ImmType = UTYPE;
        7'b0010111: ImmType = UTYPE;
        default: ImmType = RTYPE;
    endcase
end

endmodule
```

alu.v

```
`timescale 1ns / 1ps
import Params::*;
`include "Parameters.v"
module ALU(
input logic [31:0] Operand1,
input logic [31:0] Operand2,
input AluOp AluControl,
output logic [31:0] AluOut
    );

always_comb
begin
    case(AluControl)
       SLL:AluOut=Operand1<<(Operand2[4:0]);
       SRL:AluOut=Operand1>>(Operand2[4:0]);
       SRA:AluOut=$signed(Operand1)>>>(Operand2[4:0]);
       ADD:AluOut=Operand1+Operand2;
       SUB:AluOut=Operand1-Operand2;
       XOR:AluOut=Operand1^Operand2;
       OR:AluOut=Operand1|Operand2;
       AND:AluOut=Operand1&Operand2;

SLT:AluOut=($signed(Operand1)<$signed(Operand2))?32'd1:32'd0;
       SLTU:AluOut=(Operand1<Operand2)?32'd1:32'd0;
       LUI:AluOut=Operand2;
       default:AluOut=0;
    endcase
end
endmodule
```

branchunit.v

```
PHP
`timescale 1ns / 1ps
```

```
`include "Parameters.v"
import Params::*;
module BranchUnit(
input logic [31:0] Reg1,
input logic [31:0] Reg2,
input BType BrType,
output logic BranchE
    );
always_comb
begin
    case(BrType)
        BEQ:if(Reg1==Reg2)
                BranchE=1;
            else
                BranchE=0;
        BNE:if(Reg1!=Reg2)
                BranchE=1;
            else
                BranchE=0;
        BLT:if($signed(Reg1)<$signed(Reg1))
                BranchE=1;
            else
                BranchE=0;
        BLTU:if(Reg1<Reg2)
                BranchE=1;
             else
                BranchE=0;
        BGE:if($signed(Reg1)>=$signed(Reg2))
                BranchE=1;
            else
                BranchE=0;
        BGEU:if(Reg1>=Reg2)
                BranchE=1;
             else
                BranchE=0;
        default:BranchE=0;
    endcase
end
endmodule
```

datamem.v

```
module DataMem(
input logic clk,
input logic [31:0] A,
```

```
input logic [31:0] WD,
input logic [3:0] WE,
output logic [31:0] RD
    );
parameter  N = 4096;
logic [31:0] DMEM [0:N-1];
initial begin
    for(int i=0;i<N;i++)
        DMEM[i]=i;
end
assign RD=DMEM[A[31:2]];
always_ff@(negedge clk)
begin
    case(WE)
        4'b0001:DMEM[A[31:2]][7:0]<=WD[7:0];
        4'b0011:DMEM[A[31:2]][15:0]<=WD[15:0];
        4'b1111:DMEM[A[31:2]]<=WD;
        default:;
    endcase
end
endmodule
```

data_ext.v

```
`timescale 1ns / 1ps
`include "Parameters.v"
import Params::*;

module Data_Ext(
input logic [1:0] LoadByteSelect,
input LoadType RegWrite,
input logic [31:0] In,
output logic [31:0] Out
    );
always_comb
begin
    case(RegWrite)
        LW:Out=In;
        LH:begin
            case(LoadByteSelect)
               2'b00:Out={{16{In[15]}},In[15:0]};
               2'b01:Out={{16{In[23]}},In[23:8]};
               2'b10:Out={{16{In[31]}},In[31:16]};
               2'b11:Out=0;
            endcase
```

```
        end
        LHU:begin
            case(LoadByteSelect)
               2'b00:Out={16'd0,In[15:0]};
               2'b01:Out={16'd0,In[23:8]};
               2'b10:Out={16'd0,In[31:16]};
               2'b11:Out=0;
            endcase
        end
        LB:begin
            case(LoadByteSelect)
               2'b00:Out={{24{In[7]}},In[7:0]};
               2'b01:Out={{24{In[15]}},In[15:8]};
               2'b10:Out={{24{In[23]}},In[23:16]};
               2'b11:Out={{24{In[31]}},In[31:24]};
            endcase
        end
        LBU:begin
            case(LoadByteSelect)
               2'b00:Out={24'd0,In[7:0]};
               2'b01:Out={24'd0,In[15:8]};
               2'b10:Out={24'd0,In[23:16]};
               2'b11:Out={24'd0,In[31:24]};
            endcase
        end
        default:Out=In;
    endcase
end
endmodule
```

immunit.v

```
`timescale 1ns / 1ps
import Params::*;
`include "Parameters.v"

module ImmUnit(
input Type ImmType,
input logic [31:0] In,
output logic [31:0] Out
    );
always_comb
begin
```

```
    case(ImmType)
        RTYPE:Out=32'd0;
        ITYPE:Out={{20{In[31]}},In[31:20]};
        UTYPE:Out={In[31:12],12'd0};

BTYPE:Out={{19{In[31]}},In[31],In[7],In[30:25],In[11:8],1'b0};

JTYPE:Out={{11{In[31]}},In[31],In[19:12],In[20],In[30:21],1'b0};
        STYPE:Out={{20{In[31]}},In[31:25],In[11:7]};
        default:Out=32'd0;
    endcase
end
endmodule
```

instrmem.v

```
module InstrMem(
input logic [31:0] InstrAddr,
output logic [31:0] Instr
    );
parameter  N = 64;
logic [31:0] IMEM [0:N-1];
initial begin
    IMEM[0]=32'h00410093;
    IMEM[1]=32'h00102423;
    IMEM[2]=32'h0040a283;
    IMEM[3]=32'h0000c437;
    IMEM[4]=32'h0000c317;
    IMEM[5]=32'h00508133;
    IMEM[6]=32'h00508463;
    IMEM[7]=32'h005080b3;
    IMEM[8]=32'h00c000ef;
    IMEM[9]=32'h005080b3;
    IMEM[10]=32'h005080b3;
    IMEM[11]=32'h40508433;
    IMEM[12]=32'h00241513;
    IMEM[13]=32'h40155633;
    IMEM[14]=32'h401003b3;
    IMEM[15]=32'h4013d393;
    IMEM[16]=32'h04428867;
    IMEM[17]=32'h001080b3;
    IMEM[18]=32'h00187813;
end
assign Instr=IMEM[InstrAddr[31:2]];
```

```
endmodule
```

NPC_generator.v

```
Python
`timescale 1ns / 1ps
module NPC_Generator(
input logic Jal,
input logic Jalr,
input logic Branch,
input logic [31:0] JalT,
input logic [31:0] JalrT,
input logic [31:0] BrT,
input logic [31:0] PC,
output logic [31:0] NPC
    );
always_comb
begin
    if(Jalr)
        NPC=JalrT;
    else if(Branch)
        NPC=BrT;
    else if(Jal)
            NPC=JalT;
    else
        NPC=PC+4;
end
endmodule
```

registerfile.v

```
module RegisterFile(
input logic clk,
input logic [4:0] RS1,
input logic [4:0] RS2,
output logic [31:0] RegOut1,
output logic [31:0] RegOut2,
input logic [31:0] WD,
input logic [4:0] RD,
input logic WE
    );

logic [31:0] RegFile [0:31];
initial
begin
```

```
    for(int i=0;i<32;i++)
        RegFile[i]=0;
end
always_ff@(negedge clk)
begin
    if(WE)
    begin
        if(RD!=0)
            RegFile[RD]<=WD;
    end
end
always_comb
begin
    RegOut1=RegFile[RS1];
    RegOut2=RegFile[RS2];
end
endmodule
```