

同济大学计算机系

计算机组成原理课程实验报告



一、实验内容

本次实验，将使用 Verilog HDL 实现 53 条 MIPS 指令的 CPU 的设计、前仿真、后仿真和下板调试运行。

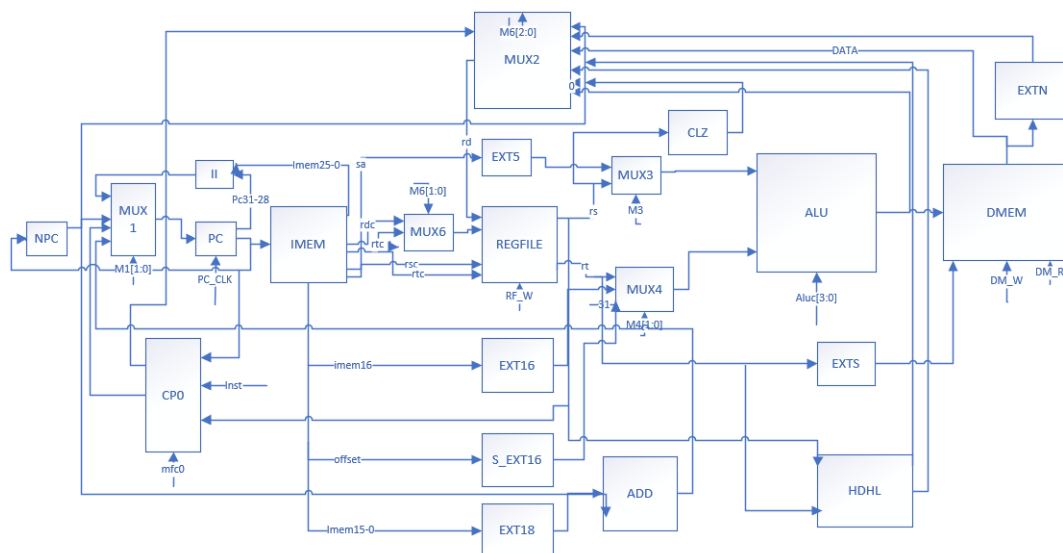
二、设计流程

1. 确定各条指令所需要的部件
2. 确定各条指令中各个部件的输入输出关系，设计各部件的数据通路
3. 确定数据通路总图
4. 根据各条指令所需要的部件以及总通路图，确定指令操作时间表（真值表）
5. 根据指令操作时间表，设计控制器
6. 根据总通路图以及指令操作时间表，编写 Verilog 代码
7. 根据指令测试程序进行测试

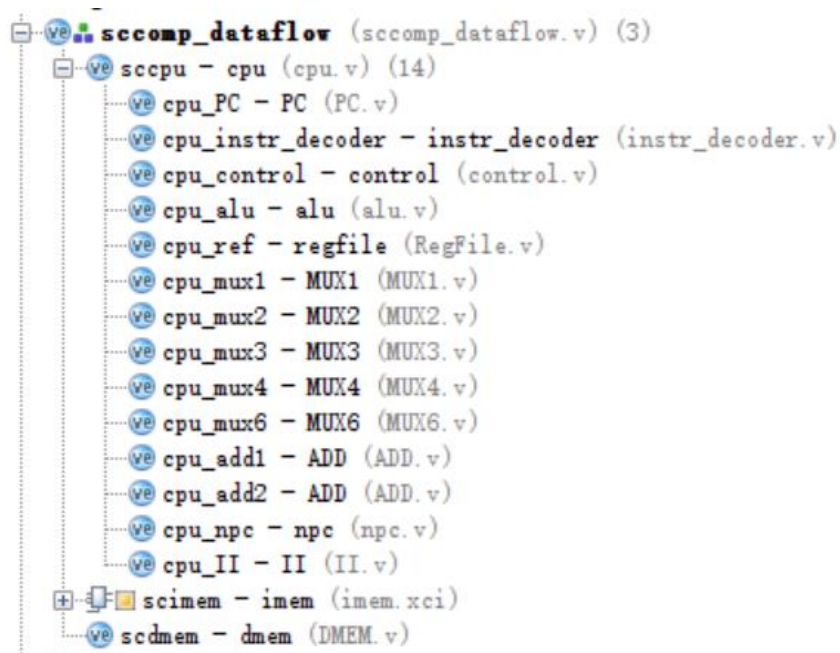
三、模块建模

（该部分要求对实验中建模的所有模块进行功能描述，并列出各模块建模的 verilog 代码）

数据通路如图：



程序总体结构如图：



(1) 顶层模块:

接口:

```
module sccomp_dataflow(
    input clk_in,
    input reset,
    output [31 : 0] inst, //指令寄存器中获取的指令码
    output [31 : 0] pc, //当前指令地址
    output [31 : 0] addr //dmem存储数据地址
);
```

功能: 本模块为顶层模块, 实例化指令存储器 imem, 数据存储器 dmem, CPU 主要功能模块 CPU。

(2) CPU 模块:

接口:

```
module cpu(
    input clk,
    input reset,
    input [31 : 0] inst, //指令寄存器中获取的指令码
    input [31 : 0] rdata, //dmem的读数据
    output [31 : 0] pc, //当前指令的地址-
    output [31 : 0] addr, //DMEM存入数据的地址
    output [31 : 0] wdata, //dmem的写数据
    //output IM_R,
    output DM_CS,
    output DM_R,
    output DM_W
);
```

功能: 调用各功能模块, 完成指令功能。

（3）数据存储单元 dmem 模块

接口：

```
module dmem(clk, DM_CS, DM_W, DM_R, addr, data_in, data_out);  
    input clk; //存储器时钟信号，上升沿时向 ram 内部写入数据  
    input DM_CS; //存储器有效信号，高电平时存储器才运行，否则输出 z  
    input DM_W; //存储器写有效信号，高电平为写有效，与 DM_CS同时有效时才可对存储器进行写  
    input DM_R; //存储器读有效信号，高电平为读有效，与 DM_CS同时有效时才可对存储器进行读  
    input [31 : 0] addr; //输入地址，指定数据读写的地址  
    input [31 : 0] data_in; //存储器写入的数据，在 clk 上升沿时被写入  
    output wire [31 : 0] data_out; //存储器读出的数据
```

功能：存储或输出数据。

（4）指令存储单元 imem

接口（以自己编写的为例）：

```
module IMEM(  
    input [10 : 0] addr,  
    output [31 : 0] inst  
);
```

功能：读取外存中程序，保存到存储器中。

（5）寄存器堆模块 regfile

接口：

```
module regfile(clk,rst,we,raddr1,raddr2,waddr,wdata,rdata1,rdata2);  
    input clk; //寄存器组时钟信号，下降沿写入数据  
    input rst; //reset 信号，异步复位，高电平时全部寄存器置零  
    input we; //寄存器读写有效信号，高电平时允许寄存器写入数据  
    input [4 : 0] raddr1; //所需读取的寄存器的地址  
    input [4 : 0] raddr2; //所需读取的寄存器的地址  
    input [4 : 0] waddr; //写寄存器的地址  
    input [31 : 0] wdata; //写寄存器数据，数据在 clk 下降沿时被写入  
    output wire [31 : 0] rdata1; //raddr1 所对应寄存器的输出数据  
    output wire [31 : 0] rdata2; //raddr2 所对应寄存器的输出数据
```

功能：根据需要存储、输出数据。

（6）运算器模块 alu

接口：

```

module alu(a, b, aluc, r, zero/*, carry*/, negative, overflow);
    input [31 : 0]a;//32位输入, 操作数1
    input [31 : 0]b;//32位输入, 操作数2
    input [3 : 0]aluc;//4位输入, 控制alu的操作
    output [31 : 0]r;//32位输出, 由a、b经过aluc指定的操作生成
    output zero;//0标志位
    //output carry;//进位标志符
    output negative;//负数标志符
    output overflow;//溢出标志符

```

功能：进行 add, addu, sub, subu, slt, lui, nor, xor, or, and, srl 等算数逻辑运算。

(7) 指令译码器模块 instr_decoder

接口：

```

module instr_decoder(
    input [31 : 0] imem,
    output reg [31 : 0] to_
);

```

功能：将指令存储器中指令译码为相应功能。

(8) PC 寄存器模块

接口：

```

module PC(
    input clk,
    input reset,
    input pcreg_ena,
    input [31 : 0] data_in, // 从指令中取出进行符号扩展后得来的
    output reg [31 : 0] data_out
);

```

功能：存放取下一条指令的地址。

(9) 数据选择器模块 MUX

接口：

```

module MUX6(//指令读取时判断是rt还是rd进入寄存器
    input [4 : 0] rdc, //0
    input [4 : 0] rtc, //1
    input [4 : 0] jal, //2
    input [4 : 0] none6, //3
    input [1 : 0] M6,
    output reg [4 : 0] to_regfile
);

```

功能：将输入的四个或两个数据根据控制信号选择其中一个。

（10）加法器 ADD 模块

接口：

```

module ADD(
    input [31 : 0] a,
    input [31 : 0] b,
    output [31 : 0] z
);
    assign z = b + a;

```

功能：实现两个数加法，输出结果。

（11）数据拼接模块

接口：

```

module II(
    input [3 : 0] pc31_28,
    input [25 : 0] imem25_0,
    output [31 : 0] to_mux1
);

```

功能：根据要求把数据拼接并输出。

（12）NPC 模块

接口：

```

module npc(
    input [31 : 0] pc,
    output [31 : 0] to_
);

```

功能：每次把取指令地址+4

另：16 位、18 位数据扩展模块之间在 CPU 模块中用 `assign` 语句实现。

四、测试模块建模

（要求列写各建模模块的 `test bench` 模块代码）

在测试模块中，将程序运行的结果如 `pc` 中的地址、指令和寄存器内容打印到自己设定的路径下文件中。

```
module sccomp_dataflow_tb();
    reg clk_in;
    reg reset;

    // Outputs
    wire [31 : 0] inst;
    wire [31 : 0] pc;
    wire [31 : 0] addr;
    integer file_output;
    integer counter = 0;
    initial
    begin
        file_output = $fopen("d:/cpu_test_result/coe_result_imem.txt");
        clk_in = 0;
        reset = 1;
        #0.25 reset = 0;
    end

    always
    begin
        #1.25 ;
        clk_in = ~clk_in;
        if(clk_in == 1'b1 )
        begin
            if(test.sccpu.inst === 32'hxxxxxxx)
                $fclose(file_output);
            else
            begin
                counter = counter + 1;

                $fdisplay(file_output,"pc: %h",pc);
                $fdisplay(file_output,"instr: %h",inst);

                $fdisplay(file_output,"regfile0: %h",test.sccpu.cpu_ref.array_reg[0]);
                $fdisplay(file_output,"regfile1: %h",test.sccpu.cpu_ref.array_reg[1]);
```

```

$fdisplay(file_output,"regfile2: %h",test.sccpu.cpu_ref.array_reg[2]);
$fdisplay(file_output,"regfile3: %h",test.sccpu.cpu_ref.array_reg[3]);
$fdisplay(file_output,"regfile4: %h",test.sccpu.cpu_ref.array_reg[4]);
$fdisplay(file_output,"regfile5: %h",test.sccpu.cpu_ref.array_reg[5]);
$fdisplay(file_output,"regfile6: %h",test.sccpu.cpu_ref.array_reg[6]);
$fdisplay(file_output,"regfile7: %h",test.sccpu.cpu_ref.array_reg[7]);
$fdisplay(file_output,"regfile8: %h",test.sccpu.cpu_ref.array_reg[8]);
$fdisplay(file_output,"regfile9: %h",test.sccpu.cpu_ref.array_reg[9]);
$fdisplay(file_output,"regfile10: %h",test.sccpu.cpu_ref.array_reg[10]);
$fdisplay(file_output,"regfile11: %h",test.sccpu.cpu_ref.array_reg[11]);
$fdisplay(file_output,"regfile12: %h",test.sccpu.cpu_ref.array_reg[12]);
$fdisplay(file_output,"regfile13: %h",test.sccpu.cpu_ref.array_reg[13]);
$fdisplay(file_output,"regfile14: %h",test.sccpu.cpu_ref.array_reg[14]);
$fdisplay(file_output,"regfile15: %h",test.sccpu.cpu_ref.array_reg[15]);
$fdisplay(file_output,"regfile16: %h",test.sccpu.cpu_ref.array_reg[16]);
$fdisplay(file_output,"regfile17: %h",test.sccpu.cpu_ref.array_reg[17]);
$fdisplay(file_output,"regfile18: %h",test.sccpu.cpu_ref.array_reg[18]);
$fdisplay(file_output,"regfile19: %h",test.sccpu.cpu_ref.array_reg[19]);
$fdisplay(file_output,"regfile20: %h",test.sccpu.cpu_ref.array_reg[20]);
$fdisplay(file_output,"regfile21: %h",test.sccpu.cpu_ref.array_reg[21]);
$fdisplay(file_output,"regfile22: %h",test.sccpu.cpu_ref.array_reg[22]);
$fdisplay(file_output,"regfile23: %h",test.sccpu.cpu_ref.array_reg[23]);
$fdisplay(file_output,"regfile24: %h",test.sccpu.cpu_ref.array_reg[24]);
$fdisplay(file_output,"regfile25: %h",test.sccpu.cpu_ref.array_reg[25]);
$fdisplay(file_output,"regfile26: %h",test.sccpu.cpu_ref.array_reg[26]);
$fdisplay(file_output,"regfile27: %h",test.sccpu.cpu_ref.array_reg[27]);
$fdisplay(file_output,"regfile28: %h",test.sccpu.cpu_ref.array_reg[28]);
$fdisplay(file_output,"regfile29: %h",test.sccpu.cpu_ref.array_reg[29]);
$fdisplay(file_output,"regfile30: %h",test.sccpu.cpu_ref.array_reg[30]);
$fdisplay(file_output,"regfile31: %h",test.sccpu.cpu_ref.array_reg[31]);
end
end
end
sccomp_dataflow test(
.clk_in(clk_in),
.reset(reset),
.inst(inst),
.pc(pc),
.addr(addr)
);
endmodule

```

五、调试方法

前仿真：本实验在 31 条单周期 CPU 上实现了单周期 54 条指令 CPU。主要增加了 CP0, CLZ, EXTN, EXTS, MDHL 部件，在原有 CPU 上进行改进时，首先对每一条新增指令确定所需部件，画出数据通路，而后增加到总的数据通路上，在指令时间分析表上增加有关指令。编写代码时，首先增添有关模块，而后按照新的指令时间表对控制信号进行改动。调试时，一种简便的方法是在仿真前先综合一下，根据警告提示改正不规范或不正确的部分，这样会大幅减少 debug 的时间。

后仿真：在 tb 中删除 \$fdisplay() 等不可综合语句，选择 Post-Synthesis Timing Simulation 进行仿真，要调试控制时钟翻转的周期，过短则会产生竞争冒险导致仿真失败。

下板：设置分频器，将开发板自带时钟降频，可以将 pc 或指令 inst 输出到七段数码管上，查看结果是否正确。

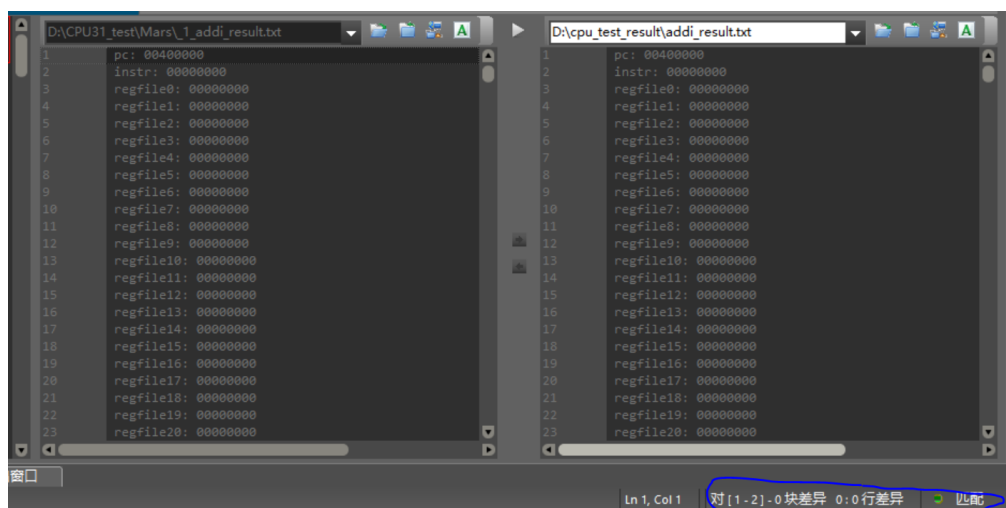
六、实验结果

（该部分可截图说明，要求 modelsim 仿真波形图、以及下板后的实验结果贴图（实验步骤中没有下板要求的实验，不需要下板贴图））

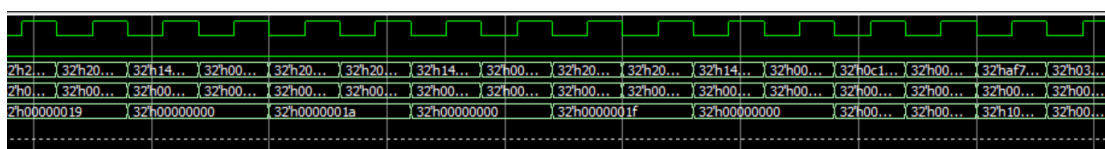
（1）测试 54 条指令：

用给出的 54 条指令程序分别测试，实验结果与 Mars 编译器跑出的结果相同。

前仿真：在 tb 中将 pc, inst 和 31 个寄存器中内容打印到文件中，用 Ultracompare 软件与给出的标准结果进行比对，通过比对，54 条指令运行结果与给出的全部相同。以 addi 比对结果为例：



modelsim 波形图：



（2）测试 coe 文件：

Modelsim 前仿真波形图：

Modelsim 后仿真波形图:

当`timescale 1ns / 1ps , #50 clk=~clk 时后仿真波形:

当`timescale 1ns / 1ps , #15 clk=~clk 时后仿真波形:

当时钟周期小于十个单位时间（1ns）时会产生竞争冒险，后仿真无法成功。

用 IP 核进行 coe 文件测试，网站提交成功。

使用网站提供的 coe 下板:

pc 最后结果: 00400fe0

inst 最后结果:081003F8