

实验报告

实验名称（RISC-V 基本指令集模拟器设计与实现）

班级：智能 1602

学号：201608010707

姓名：潘曙辉

实验目标

实现单周期 CPU 的设计。

实验要求

- 采用 VHDL 编写程序
- 模拟器的输入是二进制的机器指令文件
- 模拟器的输出是 CPU 各个寄存器的状态和相关的存储器单元状态

实验内容

CPU 指令集

CPU 的基本指令集共有 47 条指令。

我的五条指令为：JAL，BLTU，SB，XORI，ADD。

程序框架

考虑到 CPU 执行指令的流程为：

1. 取指
2. 译码
3. 执行（包括运算和结果写回）

程序被分为三个部分。一是对输入输出信号、寄存器变量的定义与初始化，二是获取寄存器变量之后进行指令相应的计算与赋值，最后是写回操作。

JAL、BLTU、SB、XORI、ADD 指令的作用：

- JAL：直接跳转指令，并带有链接功能，指令的跳转地址在指令中，跳转发生时要把返回地址存放在 R[rd]寄存器中。
- BLTU：为无符号比较，当 $R[rs1] < R[rs2]$ 时,进行跳转。
- SB：SB 指令取寄存器 R[rs2]的低位存储 8 位值到存储器。有效的字节地址是通过将寄存器 R[rs1]添加到符号扩展的 12 位偏移来获得的。
- XORI:在寄存器 R[rs1]上执行位 XOR 的逻辑操作，并立即将符号扩展 12 位，将结果放在 R[rd]中。注意：XORI R[rd], R[rs1], -1 执行寄存器 R[rs1]的位逻辑反转。
- ADD:进行加法运算， $R[rs1] + R[rs2]$ ，将结果存入 R[rd]中。

入参定义如下，包括了输入输出、时钟、重置等信号。

```
entity cpu is
    port(
        clk: in std_logic;
        reset: in std_logic;
        inst_addr: out std_logic_vector(31 downto 0);
        inst: in std_logic_vector(31 downto 0);
        data_addr: out std_logic_vector(31 downto 0);
```

```

        data_in: in std_logic_vector(31 downto 0);
        data_out: out std_logic_vector(31 downto 0);
        data_read: out std_logic;
        data_write: out std_logic

    );
end entity cpu;

```

architecture 部分，声明了计算是需要使用的变量。ir 表示当前执行的指令，pc 表当前的指令的地址；7 位的 opcode，3 位的 funct3，7 位的 funct7，这三个变量读取 ir 的指令，取到对应的值。寄存器 rd,rs1,rs2 存储 ir 中读取到的对应操作值地址，src1,src2 将 rs1,rs2 中的地址对于的 reg 中的值转为 32 位保存。Imm11_0I、Imm20_1J、Imm11_0S、Imm12_1B 是三种不同类型的指令下立即数的拓展。

```

        signal ir: std_logic_vector(31 downto 0);
        signal pc: std_logic_vector(31 downto 0);

        signal next_pc: std_logic_vector(31 downto 0);

        -- Fields in instruction
        signal opcode: std_logic_vector(6 downto 0);
        signal rd: std_logic_vector(4 downto 0);
        signal funct3: std_logic_vector(2 downto 0);
        signal rs1: std_logic_vector(4 downto 0);
        signal rs2: std_logic_vector(4 downto 0);
        signal funct7: std_logic_vector(6 downto 0);

        signal Imm11_0I : std_logic_vector(31 downto 0);
        signal Imm20_1J : std_logic_vector(31 downto 0);
        signal Imm12_1B : std_logic_vector(31 downto 0);
        signal Imm11_0S : std_logic_vector(31 downto 0);

        signal src1: std_logic_vector(31 downto 0);
        signal src2: std_logic_vector(31 downto 0);
        signal subresult: std_logic_vector(31 downto 0);

        signal sb_d1: std_logic_vector(31 downto 0);
        signal sb_a1: std_logic_vector(31 downto 0);

        signal jalresult: std_logic_vector(31 downto 0);
        signal bltresult: std_logic_vector(31 downto 0);

```

```

        signal sbresult: std_logic_vector(31 downto 0);
        signal xoriresult: std_logic_vector(31 downto 0);
        signal addresult: std_logic_vector(31 downto 0);

        type regfile is array(natural range<>) of
std_logic_vector(31 downto 0);
        signal regs: regfile(31 downto 0);
        signal reg_write: std_logic;
        signal reg_write_id: std_logic_vector(4 downto 0);
        signal reg_write_data: std_logic_vector(31 downto 0);

```

取出各个值，拼凑指令要求的立即数：

```

        inst_addr <= pc;
        ir <= inst;

        -- Decode
        -- Not finished
        opcode <= ir(6 downto 0);
        rd <= ir(11 downto 7);
        funct3 <= ir(14 downto 12);
        rs1 <= ir(19 downto 15);
        rs2 <= ir(24 downto 20);
        funct7 <= ir(31 downto 25);

        Imm11_0I <= "11111111111111111111" & ir(31 downto 20) when
ir(31)='1' else

        "00000000000000000000" & ir(31 downto 20);
        Imm20_1J <= "111111111111" & ir(31) & ir(19 downto 12) & ir(20) &
ir(30 downto 21) when ir(31)='1' else

        "000000000000" & ir(31) & ir(19 downto 12) & ir(20) & ir(30 downto
21);
        Imm11_0S <= "11111111111111111111" & ir(31 downto 25) & ir(11
downto 7) when ir(31)='1' else

        "00000000000000000000" & ir(31 downto 25) & ir(11 downto 7);
        Imm12_1B <= "11111111111111111111" & ir(31) & ir(7) & ir(30 downto
25) & ir(11 downto 8) when ir(31)='1' else

        "00000000000000000000" & ir(31) & ir(7) & ir(30 downto 25) & ir(11
downto 8);

        -- Read operands from register file
        src1 <= regs(TO_INTEGER(UNSIGNED(rs1)));
        src2 <= regs(TO_INTEGER(UNSIGNED(rs2)));

```

```

        sb_d1 <= src2 and "11111111";
        sb_a1 <= STD_LOGIC_VECTOR (SIGNED(src1) +
SIGNED(Imm11_0S));

        -- Prepare index and data to write into register file
        reg_write_id <=
rd;

```

执行阶段，当满足某条指令条件时执行该指令。

```

        addresult <= STD_LOGIC_VECTOR(SIGNED(src1) +
SIGNED(src2));
        --subresult <= STD_LOGIC_VECTOR(SIGNED(src1) -
SIGNED(src2));
        jalresult <= STD_LOGIC_VECTOR(UNSIGNED(pc)+4);
        xoriresult <= STD_LOGIC_VECTOR(SIGNED(src1) or
SIGNED(Imm11_0I));
        reg_write_data <= addresult when opcode = "0110011" and
funct7 = "0000000" else
--subresult when opcode
= "0110011" and funct7 = "0100000" else
        jalresult when opcode =
"1101111" else
        --bltresult when opcode =
"1100011" and funct3 = "110" else
        --sbresult when opcode =
"0100011" and funct3 = "000" else
        xoriresult when opcode =
"0010011" and funct3 = "100" else
"00000000000000000000000000000000";
        data_addr <= sb_a1 when opcode = "0100011" and funct3 =
"000";
        data_out <= sb_d1 when opcode = "0100011" and funct3 =
"000";
        next_pc <= STD_LOGIC_VECTOR(UNSIGNED(pc) +
UNSIGNED(Imm20_1J)) when opcode = "1101111" else
        STD_LOGIC_VECTOR(UNSIGNED(pc) +
UNSIGNED(Imm12_1B)) when opcode = "1100011" and funct3 = "110" and
(SIGNED(src1) < SIGNED(src2)) else
        STD_LOGIC_VECTOR(UNSIGNED(pc)+4);

```

reg_write 为写操作的标记，当为'1'时表示需要将 reg_write_data 的值写入下标为 reg_write_id 的寄存器中。

```

signal reg_write: std_logic;
signal reg_write_id: std_logic_vector(4 downto 0);
signal reg_write_data: std_logic_vector(31 downto 0);

```

最后写回阶段，当时钟上跳时触发。

```

-- Update pc and register file at rising edge of clk
process(clk)
begin
    if(rising_edge(clk)) then
        if (reset='1') then
            pc <= "00000000000000000000000000000000";
            -- Clear register file?

        else

            pc <= next_pc;

            if (reg_write = '1') then

regs(TO_INTEGER(UNSIGNED(reg_write_id))) <= reg_write_data;
                end if; -- reg_write = '1'
            end if; -- reset = '1'
        end if; -- rising_edge(clk)
    end process; -- clk

```

测试

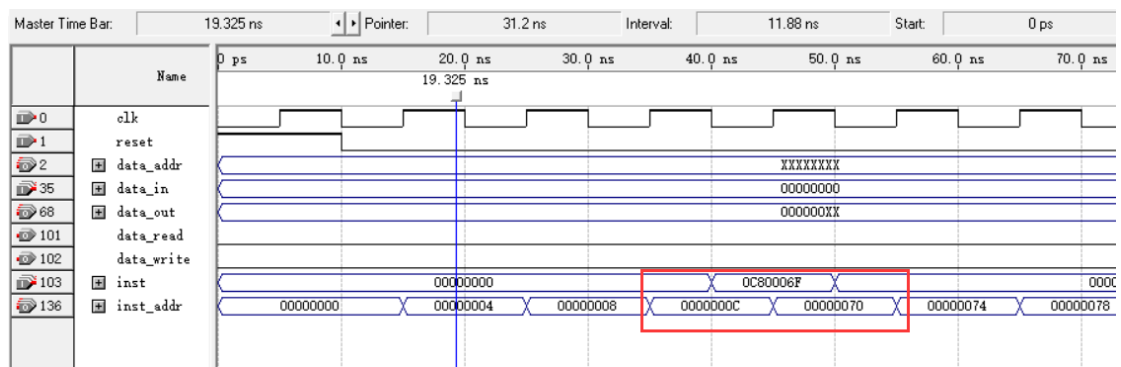
测试记录

测试指令为跳转指令，跳转到当前 PC+100 的位置

二进制代码如下:

00001100100000000000000001101111

模拟器运行过程的截图如下:



跳转到了 $0xc+100$ 即 $0x70$ 的位置

分析和结论

从测试记录来看，模拟器实现了对二进制指令文件的读入，指令功能的模拟，根据分析结果，可以认为编写的模拟器实现了所要求的功能，完成了实验目标。