实验报告

智能 1602 班 201608010623 李路

实验目标:

实现单周期 CPU 的设计

实验要求 :

硬件设计采用 VHDL 或 Verilog 语言，软件设计采用 C/C++或 SystemC 语言，其它语言例如 Chisel、MyHDL 等也可选

实验报告采用 markdown 语言，或者直接上传 PDF 文档

实验最终提交所有代码和文档

实验内容 :

1. RISC-V 指令集

RISC-V（英文发音为"risk-five"）是一个全新的指令集架构，该架构最初由美国加州大学伯克利分校的 EECS 部门的计算机科学部门的 Krste Asanovic 教授、Andrew Waterman 和 Yunsup Lee 等开发人员于 2010 年发明。 其中"RISC"表示精简指令集，而其中"V"表示伯克利分校从 RISC I 开始设计的第五代指令集。 2010 年，加州大学伯克利分校的研究团队分析了 ARM、MIPS、SPARC、X86 等多种指 令集，发现这些指令集不仅复杂度不断提升，且还存在知识产权风险，而处理器架构种类和 处理能力并无直接关联。针对以上问题，该小组设计并推出了一套基于 BSD 协议许可的免 费开放的指令集架构 RISC-V，其原型芯片也于 2013 年 1 月成功流片。RISC-V 指令集具有 性能优越，彻底免费开放两大特征。RSIC-V 的设计目标是能够满足从微控制器到超级计算 机等各种复杂程度的处理器需求，支持从 FPGA、ASIC 乃至未来器件等多种实现方式，同时 能够高效地实现各种微结构，支持大量定制与加速功能，并与现有软件及编程语言可良好适 配。RISC-V 产业生态正进入快速发展期。加州大学伯克利分校在 2015 年成立非盈利组织 RISC-V 基金会，该基金会旨在聚合全球创新力量共同构建开放、合作的软硬件社区，打造 RISC-V 生态系统。三年多来，谷歌、高通、IBM、英伟达、NXP、、西部数据、Microsemi、 中科院计算所、麻省理工学院、华盛顿大学、英国宇航系统公司等 100 多个企业和研究机构 先后加入了 RISC-V 基金会。

2. RISC-V 指令集编码格式

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

3. RISC-V 指令集

Instruction	Constraints	Code Points	Purpose
LUI	*rd*=x0	2^{20}	*Reserved for future standard use*
AUIPC	*rd*=x0	2^{20}	
ADDI	*rd*=x0, and either *rs1*≠x0 or *imm*≠0	$2^{17}-1$	
ANDI	*rd*=x0	2^{17}	
ORI	*rd*=x0	2^{17}	
XORI	*rd*=x0	2^{17}	
ADDIW	*rd*=x0	2^{17}	
ADD	*rd*=x0	2^{10}	
SUB	*rd*=x0	2^{10}	
AND	*rd*=x0	2^{10}	
OR	*rd*=x0	2^{10}	
XOR	*rd*=x0	2^{10}	
SLL	*rd*=x0	2^{10}	
SRL	*rd*=x0	2^{10}	
SRA	*rd*=x0	2^{10}	
ADDW	*rd*=x0	2^{10}	
SUBW	*rd*=x0	2^{10}	
SLLW	*rd*=x0	2^{10}	
SRLW	*rd*=x0	2^{10}	
SRAW	*rd*=x0	2^{10}	
FENCE	*pred*=0 or *succ*=0	$2^{5}-1$	
SLTI	*rd*=x0	2^{17}	*Reserved for custom use*
SLTIU	*rd*=x0	2^{17}	
SLLI	*rd*=x0	2^{11}	
SRLI	*rd*=x0	2^{11}	
SRAI	*rd*=x0	2^{11}	
SLLIW	*rd*=x0	2^{10}	
SRLIW	*rd*=x0	2^{10}	
SRAIW	*rd*=x0	2^{10}	
SLT	*rd*=x0	2^{10}	
SLTU	*rd*=x0	2^{10}	

模拟器程序框架:

cpu 执行指令的流程为 1. 取指 2. 译码 3. 执行
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 指令的作用分别如下:
1、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。
2、BLTU：为无符号比较，当 R[rs1]<R[rs2]时,进行跳转。
3、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。
4、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：XORIR[rd]，R[rs1]，-1 执行寄存器 R[rs1]的位逻辑反转。
5、ADD:进行加法运算，R[rs1]+R[rs2]，将结果存入 R[rd]中。 输入参定义如下，包括了输入输出、时钟、重置等信号。

五条指令中，LUI 存放立即数到 rd 的高 20 位，低 12 位置 0。BGE 在满足 src1>=src2 条件时跳转，跳转范围为 pc(+/-)4KB。LBH 从存储器加载一个 8 位值，然后在存储到 reg[rd]之前将零扩展到 32 位。SLTIU 在 src1 小于立即数(都是无符号整数)的情况下将 reg[rd]置 1,否则置 0。SRAI 将 src1 里面的数据算数右移，并存入 reg[rd]内 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;
```

结构部分，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。

```
    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 shamt: std_logic_vector(4 downto 0);
    signal Imm31_12UtypeZeroFilled: std_logic_vector(31 downto 0);
    signal Imm12_1BtypeSignExtended: std_logic_vector(31 downto 0);
    signal Imm11_0ItypeSignExtended: std_logic_vector(31 downto 0);

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

    signal LUIresult: std_logic_vector(31 downto 0);
    signal AUIPCresult: std_logic_vector(31 downto 0);
    signal BGEresult: std_logic_vector(31 downto 0);
    signal LBUresult: std_logic_vector(31 downto 0);
    signal SLTIUresult: std_logic_vector(31 downto 0);
    signal SRAIresult: std_logic_vector(31 downto 0);
```

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);
```

获取对于指令要求的立即数的值：

```
    inst_addr <= pc;
    ir <= inst;

    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);
    shamt <= rs2;
    Imm31_12UtypeZeroFilled <= ir(31 downto 12) & "000000000000";
    Imm12_1BtypeSignExtended <= "1111111111111111111" & ir(31) & ir(7) & ir(30 downto 25) & ir(11 downto 8) when ir(31)='1' else
                                "0000000000000000000" & ir(31) & ir(7) & ir(30 downto 25) & ir(11 downto 8);
    Imm11_0ItypeSignExtended <= "11111111111111111111" & ir(31 downto 20) when ir(31)='1' else
                                "00000000000000000000" & ir(31 downto 20);

    src1 <= regs(TO_INTEGER(UNSIGNED(rs1)));
    src2 <= regs(TO_INTEGER(UNSIGNED(rs2)));

    reg_write_id <= rd;
```

当程序执行到一定步骤，将结果保存下来

```
    addresult <= STD_LOGIC_VECTOR(SIGNED(src1) + SIGNED(src2));
    subresult <= STD_LOGIC_VECTOR(SIGNED(src1) - SIGNED(src2));
    LUIresult <= Imm31_12UtypeZeroFilled;
```

```
AUIPCresult <= STD_LOGIC_VECTOR(SIGNED(pc) + SIGNED(Imm31_12UtypeZeroFilled));
SRAIresult <= to_stdlogicvector( to_bitvector(src1) SRA to_integer(unsigned(shamt)) ) ;
SLTIUresult <= "00000000000000000000000000000001" when
TO_INTEGER(UNSIGNED(src1)) < TO_INTEGER(UNSIGNED(Imm11_0ItypeSignExtended)) else
"00000000000000000000000000000000";
LBUresult <= "000000000000000000000000" & data_in(7 downto 0);
-- more
-- ……

reg_write_data <= addresult when opcode = "0110011" and funct7 = "0000000" else
subresult when opcode = "0110011" and funct7 = "0100000" else
LUIresult when opcode = "0110111" else
AUIPCresult when opcode = "0010111" else
LBUresult when opcode = "0000011" and funct3 = "100" else
SRAIresult when opcode = "0010011" and funct3 = "101" and ir(31
downto 25) = "0100000" else
SLTIUresult when opcode = "0010011" and funct3 = "011" else
-- more
-- ……
-- At last, set a default value
"00000000000000000000000000000000";

-- Execute
-- Not finished

next_pc <= STD_LOGIC_VECTOR(SIGNED(pc) + SIGNED(Imm12_1BtypeSignExtended))
when opcode = "1100011" and funct3 = "101" and SIGNED(src1) >= SIGNED(src2) else
STD_LOGIC_VECTOR(SIGNED(pc) + 4);
```

执行阶段，根据获取到的值，计算出指令的结果。其中 nextpc 正常情况下+4，在满足 BGE 条件时跳转到对应地址。

```
LUIresult <= Imm31_12UtypeZeroFilled;
AUIPCresult <= STD_LOGIC_VECTOR(SIGNED(pc) + SIGNED(Imm31_12UtypeZeroFilled));
SRAIresult <= to_stdlogicvector( to_bitvector(src1) SRA to_integer(unsigned(shamt)) ) ;
SLTIUresult <= "00000000000000000000000000000001" when
TO_INTEGER(UNSIGNED(src1)) < TO_INTEGER(UNSIGNED(Imm11_0ItypeSignExtended)) else
"00000000000000000000000000000000";
LBUresult <= "000000000000000000000000" & data_in(7 downto 0);

reg_write_data <= addresult when opcode = "0110011" and funct7 = "0000000" else
subresult when opcode = "0110011" and funct7 = "0100000" else
LUIresult when opcode = "0110111" else
AUIPCresult when opcode = "0010111" else
```

```
                    LBUresult when opcode = "0000011" and funct3 = "100" else
                    SRAIresult when opcode = "0010011" and funct3 = "101" and ir(31
downto 25) = "0100000" else
                    SLTIUresult when opcode = "0010011" and funct3 = "011" else
                    -- more
                    -- ......
                    -- At last, set a default value
                    "00000000000000000000000000000000";

    next_pc <= STD_LOGIC_VECTOR(SIGNED(pc) + SIGNED(Imm12_1BtypeSignExtended))
when opcode = "1100011" and funct3 = "101" and SIGNED(src1) >= SIGNED(src2) else
                STD_LOGIC_VECTOR(SIGNED(pc) + 4);
```

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:

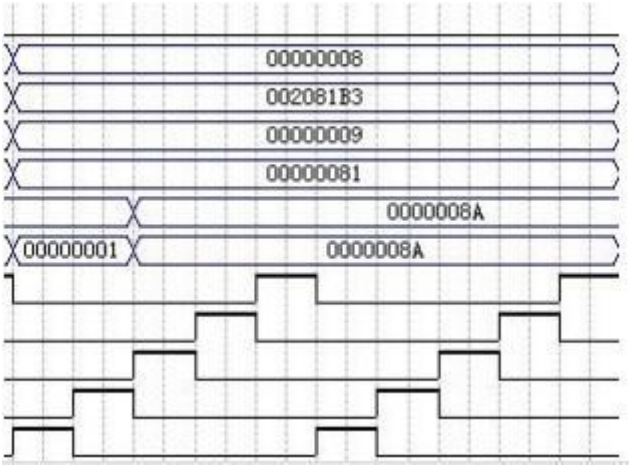
部件	配置
CPU	core i7-6300U
内存	8GB
操作系统	windows 10

测试记录:

ADD 指令

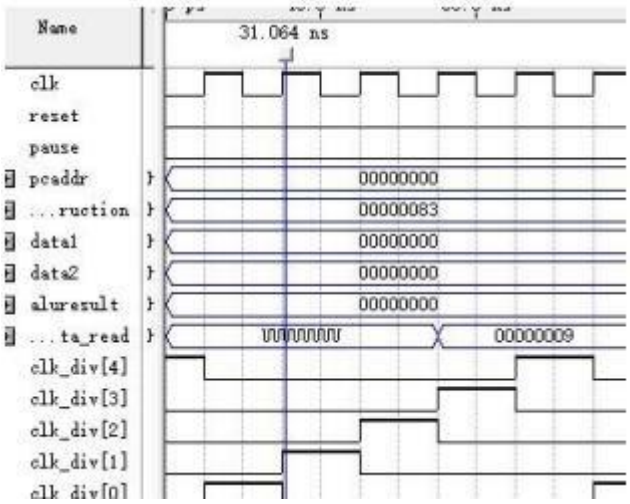
00000000001000001000000110110011

首先在寄存器 1 与 2 中都写入 01111111111111111111111111111110，故当寄存器 1 的值与寄存器 2 的值相加时，加法溢出。结果显示此时加法已溢出，并摒弃了最高位，该指令正确执行



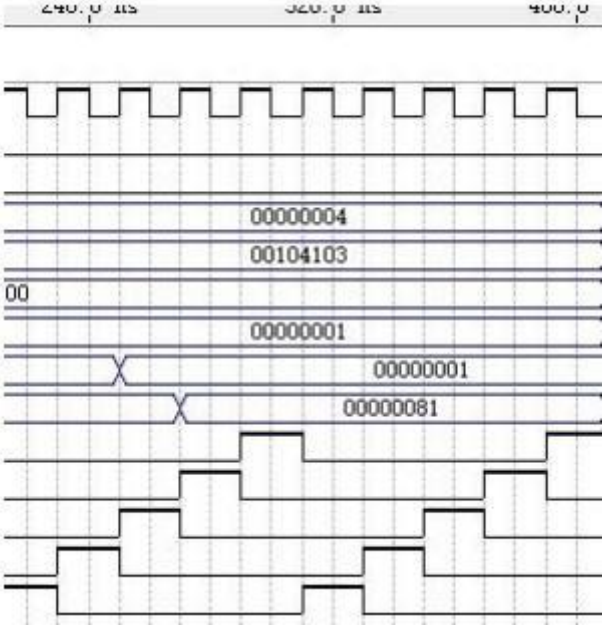
LB 指令

00000000000000000000000010000011



LBU 指令

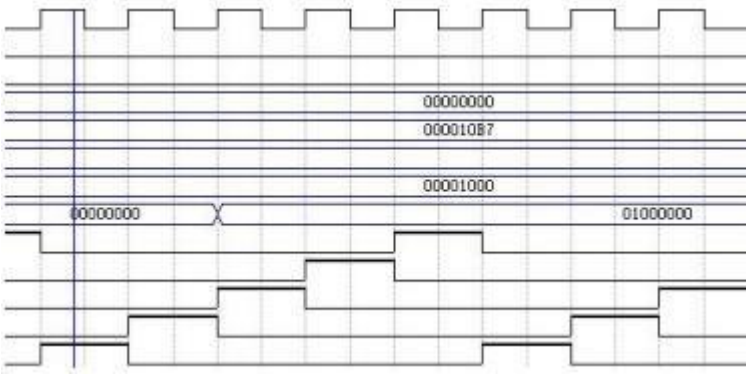
00000000000010000010000010000011



LUI x1，0x1

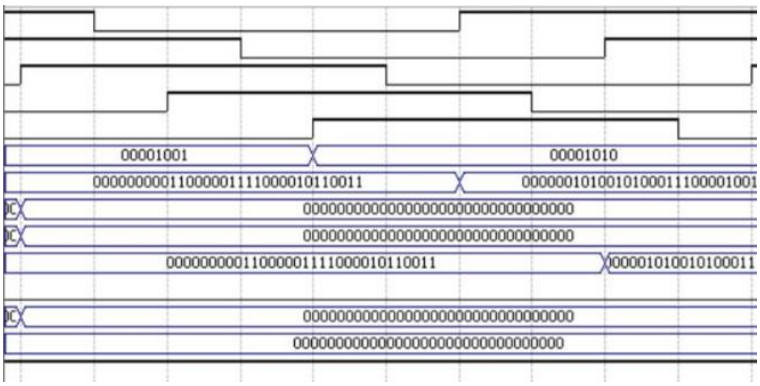
00000000000000000001000010110111

r1=00000000000000000001000000000000



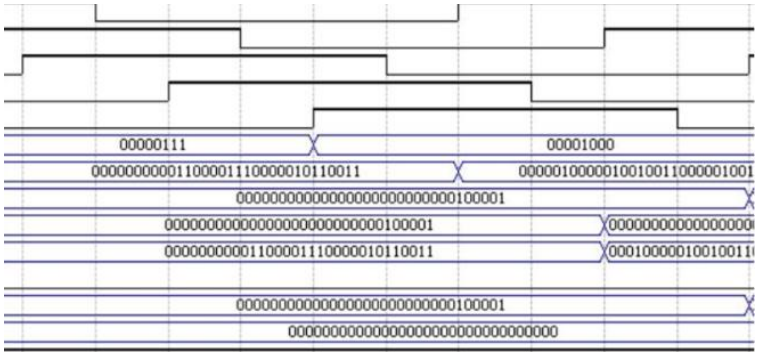
AND 指令

00000000011000001111000010110011



OR 指令

00000000001100001110000010110011



分析和结论：

从测试结果可以看出编写的 cpu 能够完成指令的工作，达到了实验的目的。