实验报告

智能 1602
邱勒铭
201608010702

实验内容

实现单周期 CPU 的设计。

实验要求

1.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。

2.实验报告采用 markdown 语言，或者直接上传 PDF 文档

3.实验最终提交所有代码和文档

模拟环境

部件 配置 备注
CPU core i5-9300H 内存 ： 8GB
操作系统：win10 专业版

CPU 指令集

RISC-V 分为 6 类，如下图所示：

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

每种类型的具体指令表：

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

实验设计:

本实验将通用寄存器，RAM，ROM 集中设计在了一个代码中，实现 RISV-V 的前 37 条指令。

由于指令是 32 位的，因此选择 std_logic_vector 类型描述。

指令相关的寄存器，RAM 均使用信号数组实现:

```
type regfile is array(natural range<>) of std_logic_vector(31 downto 0);
signal regs: regfile(31 downto 0);

type memoryfile is array(natural range<>) of std_logic_vector(31 downto 0);
signal mems: memoryfile(3 downto 0);
```

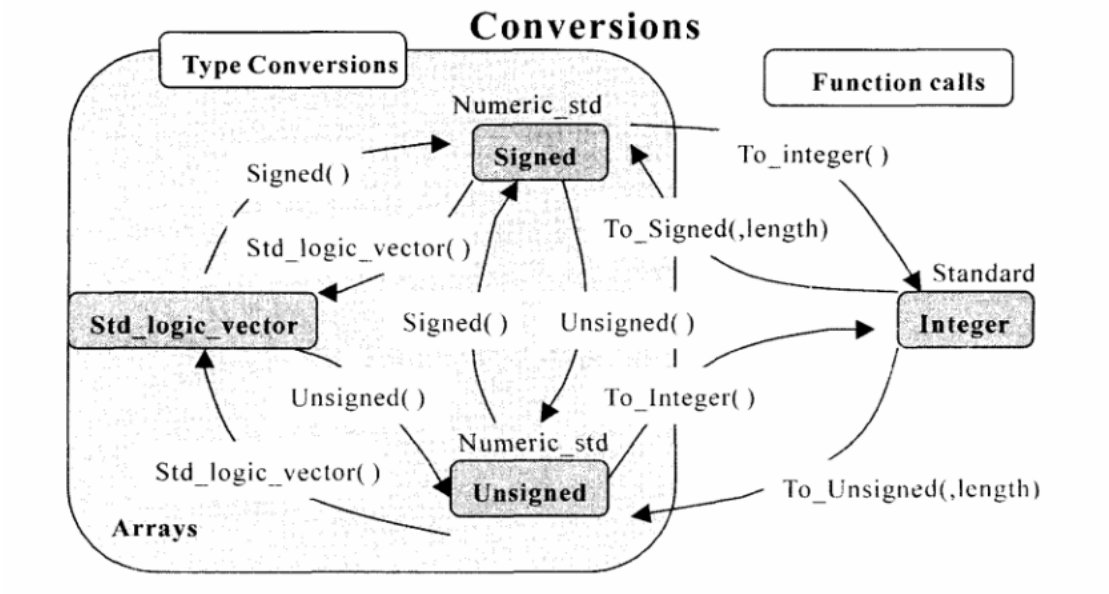
实验为了说明原理，因此 RAM 只设置了 4 个双字的大小。

简化了从 ROM 中取址的周期，每个指令周期直接在指令寄存器 inst 中赋值测试当前指令的值。

首先分析 risc 指令集的特点：
Risc-v 的特点之一是所有涉及到寄存器使用的指令中寄存器的位置都是固定的，而 opcode，func3，func7 则可以帮助我们快速的确认到具体的指令。常量的定义：

```
-- utype instructions, using opcode
constant rtype_lui: std_logic_vector(6 downto 0) := B"0110111";
constant rtype_auipc: std_logic_vector(6 downto 0) := B"0010111";

-- jtype
constant jtype_jal: std_logic_vector(6 downto 0) := B"1101111";

-- itype load instructions, using opcode, funct3
constant itype_load: std_logic_vector(6 downto 0) := B"0000011";
constant itype_jalr: std_logic_vector(6 downto 0) := B"1100111";
constant itype_lb: std_logic_vector(2 downto 0) := B"000";
constant itype_lh: std_logic_vector(2 downto 0) := B"001";
constant itype_lw: std_logic_vector(2 downto 0) := B"010";
constant itype_lbu: std_logic_vector(2 downto 0) := B"100";
constant itype_lhu: std_logic_vector(2 downto 0) := B"101";

-- rtype alu operations, using opcode, funct3, funct7
constant rtype_alu: std_logic_vector(6 downto 0) := B"0110011";
constant rtype_addsub: std_logic_vector(2 downto 0) := B"000";
constant rtype_add: std_logic_vector(6 downto 0) := B"0000000";
constant rtype_sub: std_logic_vector(6 downto 0) := B"0100000";
constant rtype_sll: std_logic_vector(2 downto 0) := B"001";
constant rtype_slt: std_logic_vector(2 downto 0) := B"010";
constant rtype_sltu: std_logic_vector(2 downto 0) := B"011";
constant rtype_xor: std_logic_vector(2 downto 0) := B"100";
constant rtype_srlsra: std_logic_vector(2 downto 0) := B"101";
constant rtype_srl: std_logic_vector(6 downto 0) := B"0000000";
constant rtype_sra: std_logic_vector(6 downto 0) := B"0100000";
constant rtype_or: std_logic_vector(2 downto 0) := B"110";
constant rtype_and: std_logic_vector(2 downto 0) := B"111";

-- btype branches, using opcode, funct3
constant btype_branch: std_logic_vector(6 downto 0) := B"1100011";
constant btype_beq: std_logic_vector(2 downto 0) := B"000";
constant btype_bne: std_logic_vector(2 downto 0) := B"001";
constant btype_blt: std_logic_vector(2 downto 0) := B"100";
constant btype_bge: std_logic_vector(2 downto 0) := B"101";
constant btype_bltu: std_logic_vector(2 downto 0) := B"110";
constant btype_bgeu: std_logic_vector(2 downto 0) := B"111";
```

由于指令众多，首先从比较熟悉的 R 类型指令开始：
ADD，SUB 指令即为算术加减，需要使用到库 ieee.numeric_std.all 其中对加减法的操作数类型进行了约束，只有整型 integer 才能参与运算，而我们的 std_logic 与整型之间的转换也需要两次才能做到，首先需要转换成 unsigned 或 signed 类型，接下来再转换成 integer 类型。其具体的关系可见下图：



故我们的 add 与 sub 指令可以设计为:

```
std_logic_vector( to_unsigned(    ( to_integer( unsigned(rs1_data)) +
to_integer( unsigned(rs2_data)) ) , 32 ) )
```

```
      std_logic_vector( to_unsigned(    ( to_integer( unsigned(rs1_data)) -
to_integer( unsigned(rs2_data)) ) , 32 ) )
```

之后是移位指令:

移位在 ieee.numeric_std.all 提供的函数中是 shift_left 与 shift_right，有无符合需要根据参数而定，移位的指令设计为:

```
std_logic_vector(    shift_left(unsigned(rs1_data) , to_integer(unsigned(rs2_data) ) ) )
std_logic_vector(shift_right(unsigned(rs1_data), to_integer(unsigned(rs2_data))))
std_logic_vector(shift_right(signed(rs1_data), to_integer(signed(rs2_data))))
```

至于用于完成比较并置一的指令 sll 和 slt 则无太多要求，直接比较即可:

```
X"00000001"
when(signed(rs1_data) < signed(rs2_data)) and funct3 = rtype_slt else
X"00000001"
when(unsigned(rs1_data) < unsigned(rs2_data)) and funct3 = rtype_sltu else
```

按位与或，异或操作:

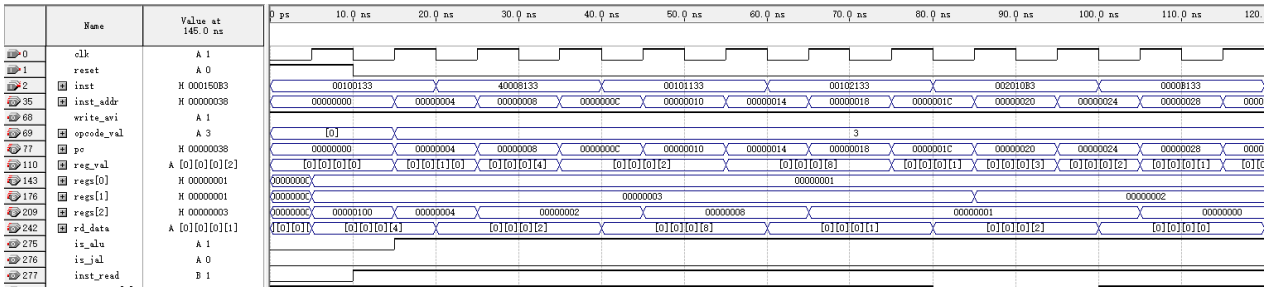
```
rs1_data xor rs2_data
when funct3 = rtype_xor else
rs1_data or rs2_data
when funct3 = rtype_or else
rs1_data and rs2_data
when funct3 = rtype_and else
```

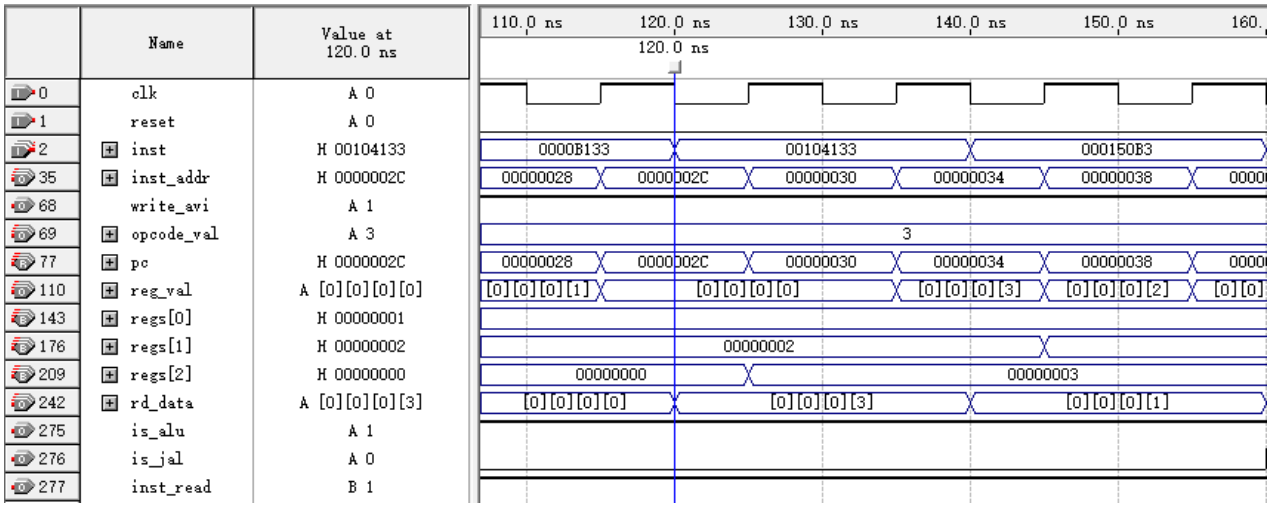
R 类型指令的测试与仿真:

首先设置测试指令:

		regs(0)	regs(1)	regs(2)	instruction	funct7(7bit	rs2	rs1	funct3(3bits)	rd	opcode(7bits)
		1	3	0x100							
r0+r1=>r2	add 0 1 2	1	3	4	0x00100133	"0000000"	"00001"	"00000"	"000"	"00010"	"0110011"
r1-r0=>r2	sub 1 0 2	1	3	2	0x40008133	"0100000"	"00000"	"00001"	"000"	"00010"	"0110011"
(r1<<r0)=>r2	sll 0 1 2	1	3	8	0x00101133	"0000000"	"00001"	"00000"	"001"	"00010"	"0110011"
	slt 2 1 0	1	3	1	0x00102133	"0000000"	"00001"	"00000"	"010"	"00010"	"0110011"
	sll 0 2 1	1	2	1	0x002010b3	"0000000"	"00010"	"00000"	"001"	"00001"	"0110011"
	sltu 1 0 2	1	2	0	0x0000b133	"0000000"	"00000"	"00001"	"011"	"00010"	"0110011"
	xor 0 1 2	1	2	3	0x00104133	"0000000"	"00001"	"00000"	"100"	"00010"	"0110011"
	srl 2 0 1	1	1	3	0x000150b3	"0000000"	"00000"	"00010"	"101"	"00001"	"0110011"

仿真结果:





无条件跳转指令 JAL:

该指令需要将下一条指令的地址 pc+4 存储到目的寄存器 rd 中，指令中提供一个立即数，该立即数作为偏移量加到当前的 pc 地址上作为下一个 pc 的值。

立即数的获得:

```
jal_imm20_1 <= ir(31) & ir(19 downto 12) & ir(20) & ir(30 downto 21);
```

这里需要说明的是指令并不参与组成该立即数的最低位，因为需要保证跳转之后的地址是 2 的整数倍。指令的 19 到 12 为为 10 downto 1。

寄存器保留下一个地址的 PC 值:

```
rd_data <= std_logic_vector( to_unsigned( (to_integer(unsigned(pc))+4) , 32 ) )
                when opcode = jtype_jal or opcode = itype_jalr else
```

该值保存了跳转命令结束后应该返回的地址。而需要跳转的 pc 值直接存储到 pc 寄存器中：

```
next_pc<=std_logic_vector( to_unsigned( (to_integer(unsigned(pc))+
                    to_integer(unsigned(jal_offset))) , 32 ))
when opcode = jtype_jal
```

与之相似的还有 LOAD 指令，JALR 指令

同样是通过偏移量找到地址，而该地址对应的却是 RAM，而我们的目的则是将 RAM 的特定地址的值加载到目的寄存器 rd 中。

具体的设计如下：

Offset 的值获取：

```
itype_imm11_0 <= ir(31 downto 20);
```

再将其加到源寄存器 rs1 上：

```
load_addr <= std_logic_vector(to_signed((to_integer(signed(rs1)) +
to_integer(signed(itype_imm11_0))),32));
```

这里考虑到接下来的 s 类型指令也有类似的操作，因此将这个需要装载的地址保存到一个统一的寄存器中：

```
data_addr <= load_addr when opcode=itype_load
else    store_addr;
```

从 RAM 中取得该值并赋值给目的寄存器 rd：

```
rd_data <= mems(to_integer(unsigned(data_addr)))
when opcode=itype_load
```

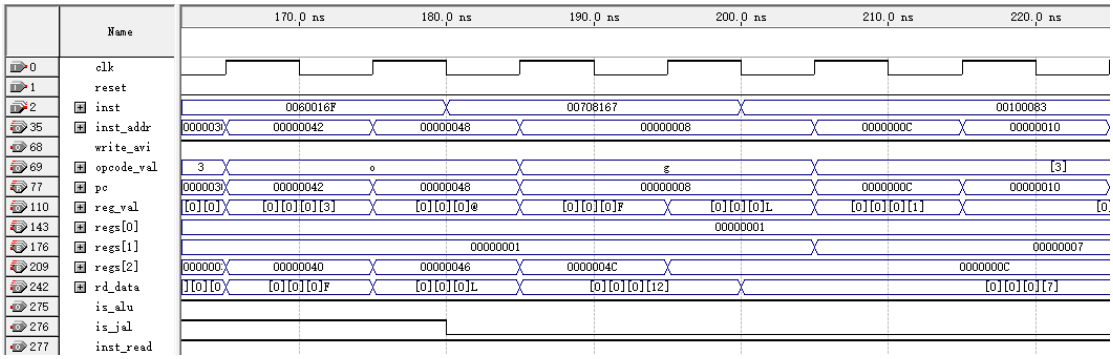
LOAD 指令又能够细分为 LW,LH,LHU,LBL,LBU 五条指令，他们不外乎是装载位数有所不同。

JALR 指令只是使用的立即数不同，其他与 JAL 是相同的。

测试：

jal	1	1	0x40	0x0060016f	"0"	"000000011"	"0"	"00000000"	"00010"	"1101111"
jalr	1	1	0x0c	0x00708167		"000000000111"	"00001"	"000"	"00010"	"1100111"
lload	1	1	8	0x00100083		"000000000001"	"00001"	"000"	"00001"	"0000011"

仿真结果为：



B 类型指令：

B 类型指令是有条件的跳转指令，当满足条件时将当前 pc 加上偏移量作为下一个 pc。需要满足的条件通过比较两个寄存器的值来决定，BEQ 和 BNE 分别是判断两个源寄存器 rs1 和 rs2 是/否相等，BLT 和 BLTU 分别使用有符号数和无符号数判断 rs1 小于 rs2；BGE 和 BGEU 则是判断 rs1 大于 rs2。

指令的设计如下:
首先得到偏移量值:

```
btype_imm12_1 <= ir(31) & ir(7) & ir(30 downto 25) & ir(11 downto 8);
```

由于偏移量时 2 的倍数，这里最低位一定是 0:

```
branch_target(13 downto 0) <= btype_imm12_1 & '0' ;
branch_target(31 downto 14) <= ( others => btype_imm12_1(12) );
```

跳转条件:

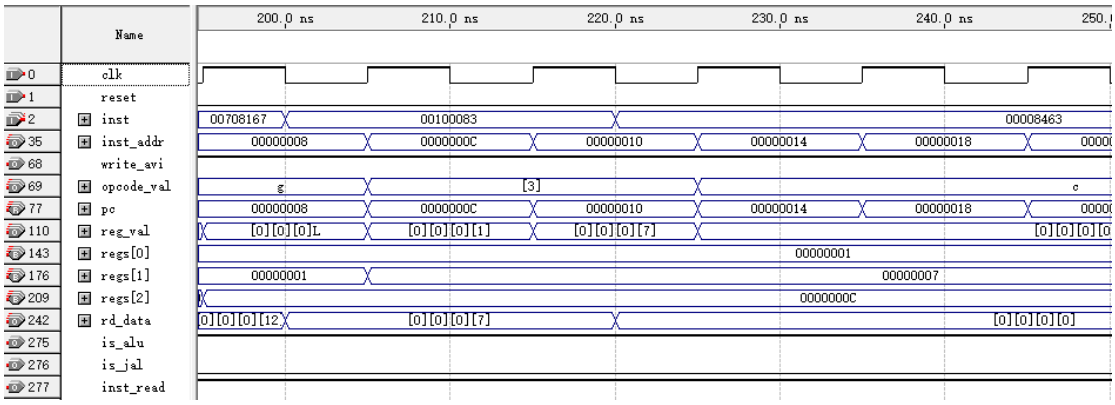
```
branch_taken <= '1' when    (rs1 = rs2 and funct3 = btype_beq )
or          (rs1 /= rs2 and funct3 = btype_bne)
or          ( signed(rs1) < signed(rs2) and funct3 = btype_blt)
or          ( unsigned(rs1) < unsigned(rs2) and funct3 = btype_bltu)
or          ( signed(rs1) > signed(rs2) and funct3 = btype_bge)
or          ( unsigned(rs1) > unsigned(rs2) and funct3 = btype_bgeu)
else '0';
```

pc 的值:

```
next_pc <= when opcode = btype_branch and branch_taken ='1';
```

测试:

beq				0x00008463	"0···0"	"00000"	"00001"	"000"	"01000"	"1100011"



实验总结:

本项目虽然不是第一次着手 cpu 的设计，但实验刚开始时仍然十分困难，不知道从何入手，当读懂了 rsic-v 指令集的每一条指令时才渐渐的明白了要做些什么，由其是那些是地址，哪些是立即数，哪些指令能够复用，当这些都想明白了再根据样例程序进行尝试，一条一条的将指令进行测试，直到得到的结果与自己要的完全一样。