实验报告

实验名称：完成一个执行 RISC-V 的基本整数指令集 RV32I 的 CPU 设计（单周期实现）

班级：智能 1602

学号：201608010609

姓名：李鹏飞

实验目标

完成一个执行 RISC-V 的基本整数指令集 RV32I 的 CPU 设计，可以实现单周期执行。

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.CPU 简介

中央处理器（CPU），是电子计算机的主要设备之一，电脑中的核心配件。其功能主要是解释计算机指令以及处理计算机软件中的数据。CPU 是计算机中负责读取指令，对指令译码并执行指令的核心部件。其主要包括两个部分，即控制器、运算器，其中还包括高速缓冲存储器及实现它们之间联系的数据、控制的总线。电子计算机三大核心部件就是 CPU、内部存储器、输入/输出设备。中央处理器的功效主要为处理指令、执行操作、控制时间、处理数据。

在计算机体系结构中，CPU 是对计算机的所有硬件资源（如存储器、输入输出单元） 进行控制调配、执行通用运算的核心硬件单元。CPU 是计算机的运算和控制核心。计算机系统中所有软件层的操作，最终都将通过指令集映射为 CPU 的操作。

2.RISC-V 指令集简介

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。

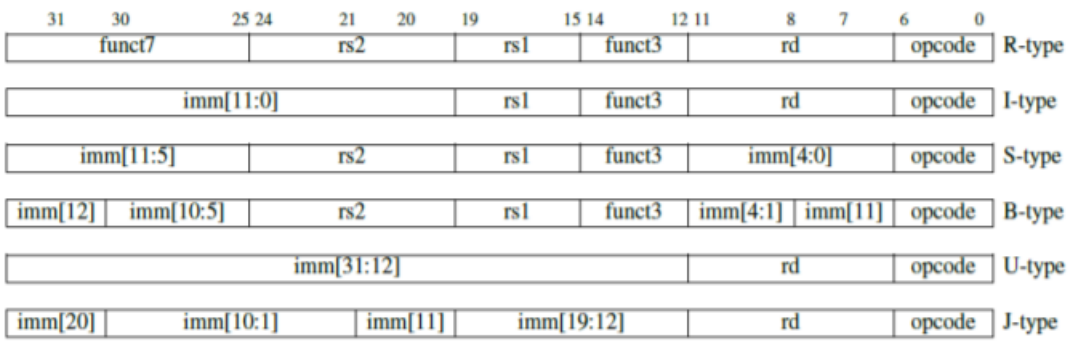
2010 年，加州大学伯克利分校的研究团队分析了 ARM、MIPS、SPARC、X86 等多种指令集，发现这些指令集不仅复杂度不断提升，且还存在知识产权风险，而处理器架构种类和处理能力并无直接关联。针对以上问题，该小组设计并推出了一套基于 BSD 协议许可的免费开放的指令集架构 RISC-V，其原型芯片也于

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。

3.RISC-V 指令集内容

我们在这里编写的是 RV32I 指令集，其包含了六种基本指令格式，分别是：用于寄存器-寄存器操作的 R 类型指令，用于短立即数和访存 load 操作的 I 型指令，用于访存 store 操作的 S 型指令，用于条件跳转操作的 B 类型指令，用于长立即数的 U 型指令和用于无条件跳转的 J 型指令。

4.RISC-V 指令集编码格式

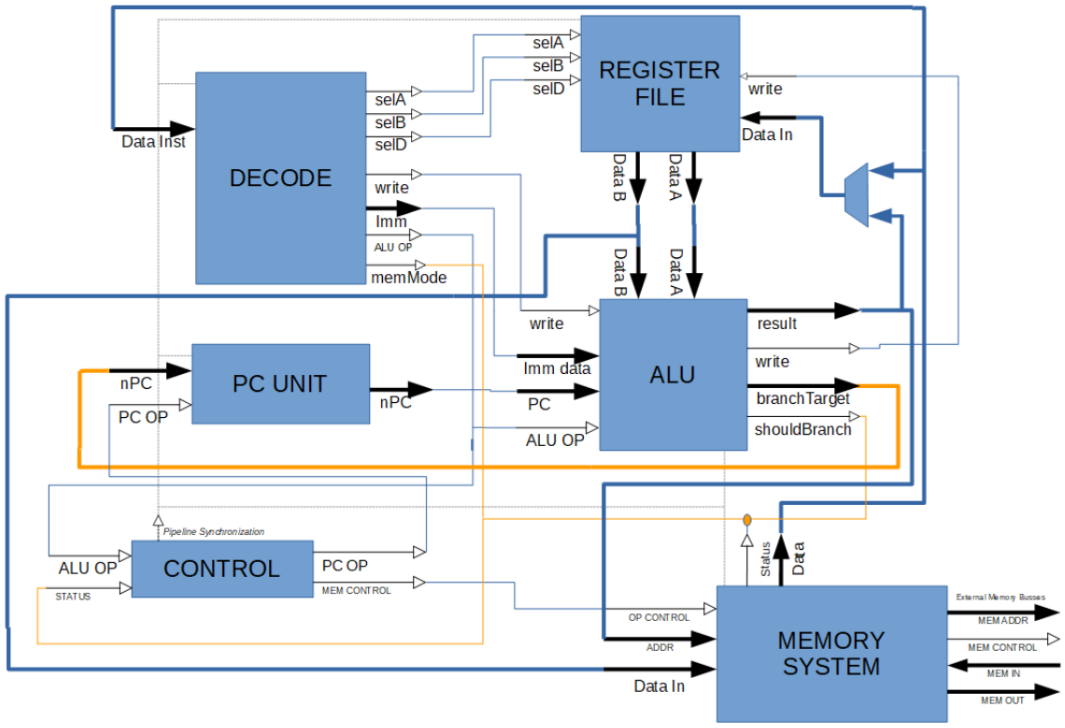


5.RISC-V 指令

Category Name	Fmt	RV32I Base	
Shifts			
Shift Left Logical	R	SLL	rd,rs1,rs2
Shift Left Log.Imm.	I	SLLI	rd,rs1,shamt
Shift Right Logical	R	SRL	rd,rs1,rs2
Shift Right Log.Imm.	I	SRLI	rd,rs1,shamt
Shift Right Arithmetic	R	SRA	rd,rs1,rs2
Shift Right Arith.Imm.	I	SRAI	rd,rs1,shamt
Arithmetic			
ADD	R	ADD	rd,rs1,rs2
ADD Immediate	I	ADDI	rd,rs1,imm
SUBtract	R	SUB	rd,rs1,rs2
Load Upper Imm	U	LUI	rd,imm
Add Upper Imm to PC	U	AUIPC	rd,imm
Logical			
XOR	R	XOR	rd,rs1,rs2
XOR Immediate	I	XORI	rd,rs1,imm
OR	R	OR	rd,rs1,rs2
OR Immediate	I	ORI	rd,rs1,imm
AND	R	AND	rd,rs1,rs2
AND Immediate	I	ANDI	rd,rs1,imm
Category Name	**Fmt**	**RV32I Base**	
Compare			
Set<	R	SLT	rd,rs1,rs2
Set<Immediate	I	SLTI	rd,rs1,rs2
Set<Unsigned	R	SLTU	rd,rs1,rs2
Set<Imm Unsigned	I	SLTIU	rd,rs1,imm
Branches			
Branch=	B	BEQ	rs1,rs2,imm
Branch≠	B	BNE	rs1,rs2,imm
Branch<	B	BLT	rs1,rs2,imm
Branch≥	B	BGE	rs1,rs2,imm
Branch<Unsigned	B	BLTU	rs1,rs2,imm
Branch≥Unsigned	B	BGEU	rs1,rs2,imm
Jump&Link			
J&L	J	JAL	rd,imm
Jump&Link Register	I	JALR	rd,rs1,imm
Synch			
Synch thread	I	FENCE	
Synch Instr&Data	I	FENCE.I	
Environment			

CALL	I	ECALL	
BREAK	I	EBREAK	
Control Status Register(CSR)			
Read/Write	I	CSRRW	rd,csr,rs1
Read&Set Bit	I	CSRRS	rd,csr,rs1
Read&Clear Bit	I	CSRRC	rd,csr,rs1
Read/Write Imm	I	CSRRWI	rd,csr,imm
Read&Set Bit Imm	I	CSRRSI	rd,csr,imm
Read&Clear Bit Imm	I	CSRRCI	rd,csr,imm
Loads			
Load Byte	I	LB	rd,rs1,imm
Load Halfword	I	LH	rd,rs1,imm
Load Byte Unsigned	I	LBU	rd,rs1,imm
Load Half Unsigned	I	LHU	rd,rs1,imm
Load Word	I	LW	rd,rs1,imm
Stores			
Store Byte	S	SB	rs1,rs2,imm
Store Halfword	S	SH	rs1,rs2,imm
Store Word	S	SW	rs1,rs2,imm

RISC-V 流程图



RISC-V 程序框架

考虑到 CPU 的处理过程分为 5 个步骤：取指令-译码-执行-访存-写回。

我们将 CPU 的框架设计如下：

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity cpu is
    port(
        clk: in std_logic;
        reset: in std_logic;
        inst_addr: out std_logic_vector(31 downto 0);  -- 指令地址
        inst: in std_logic_vector(31 downto 0);
        inst_read: out std_logic;
        data_addr: out std_logic_vector(31 downto 0);  -- 数据地址
        data: inout std_logic_vector(31 downto 0);
        data_read: out std_logic;
        data_write: out std_logic
    );
end entity;

architecture cpu_behav of cpu is
    type regfile is array(natural range<>) of std_logic_vector(31 downto 0);
    signal regs: regfile(31 downto 0);
    signal pc: std_logic_vector(31 downto 0);
    signal ir: std_logic_vector(31 downto 0);
    signal next_pc: std_logic_vector(31 downto 0);

begin
    -- 组合逻辑部分
    inst_addr <= pc;  -- 取指地址
    inst_read <= '1' when reset = '0' else '0';  -- 当 reset 无效时发出指令读取信号;
    ir <= inst;  -- 当前指令
    next_pc <= std_logic_vector(unsigned(pc) + 4);  -- when ... else ... when ... else ...; -- 需补充其它情况

    -- ..... (其它组合逻辑)


    -- 时序逻辑部分
    pc_update: process(clk)
```

```
    begin
        if(rising_edge(clk)) then
            if(reset='1') then
                pc <= X"00000000";  -- 当 reset 信号有效时，pc 被重置为 0
            else
                pc <= next_pc;
            end if;
        end if;
    end process pc_update;

    -- ..... (其它时序逻辑)

end;
```

在这里采用的思路是先把各个模块写好，再进行最后的顶层设计，在顶层设计中调用各个模块，实现 CPU 功能。

```
    COMPONENT pc_unit
    PORT(
         I_clk : IN  std_logic;
         I_nPC : IN  std_logic_vector(XLENM1 downto 0);
         I_nPCop : IN  std_logic_vector(1 downto 0);
        I_intVec: IN std_logic;
         O_PC : OUT std_logic_vector(XLENM1 downto 0)
        );
    END COMPONENT;

    COMPONENT control_unit
    PORT (
        I_clk : in  STD_LOGIC;
        I_reset : in  STD_LOGIC;
        I_aluop : in  STD_LOGIC_VECTOR (6 downto 0);
        O_state : out  STD_LOGIC_VECTOR (6 downto 0);

        I_int: in STD_LOGIC;
        O_int_ack: out STD_LOGIC;

        I_int_enabled: in STD_LOGIC;
        I_int_mem_data: in STD_LOGIC_VECTOR(XLENM1 downto 0);
        O_idata: out STD_LOGIC_VECTOR(XLENM1 downto 0);
        O_set_idata:out STD_LOGIC;
        O_set_ipc: out STD_LOGIC;
        O_set_irpc: out STD_LOGIC;
        O_instTick: out STD_LOGIC;

        I_ready: in STD_LOGIC;
        O_execute: out STD_LOGIC;
        I_dataReady: in STD_LOGIC
         );
    END COMPONENT;
```

测试

测试平台

模拟器在如下机器上进行了测试

部件	配置	备注
CPU	Core i5-6700U	
内存	DDR4 12GB	
操作系统	Windows10 家庭版	

测试记录

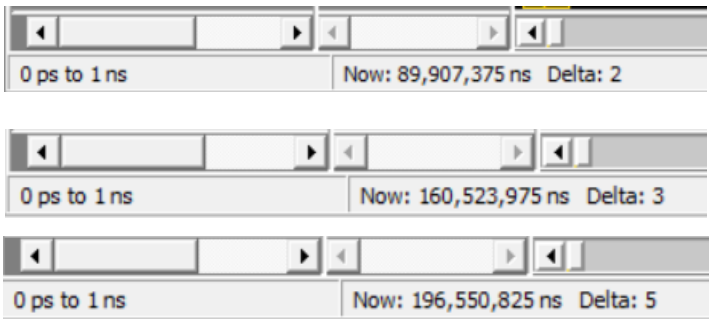
测试环境：Quartus II 16.0

利用 testbench 文件进行测试

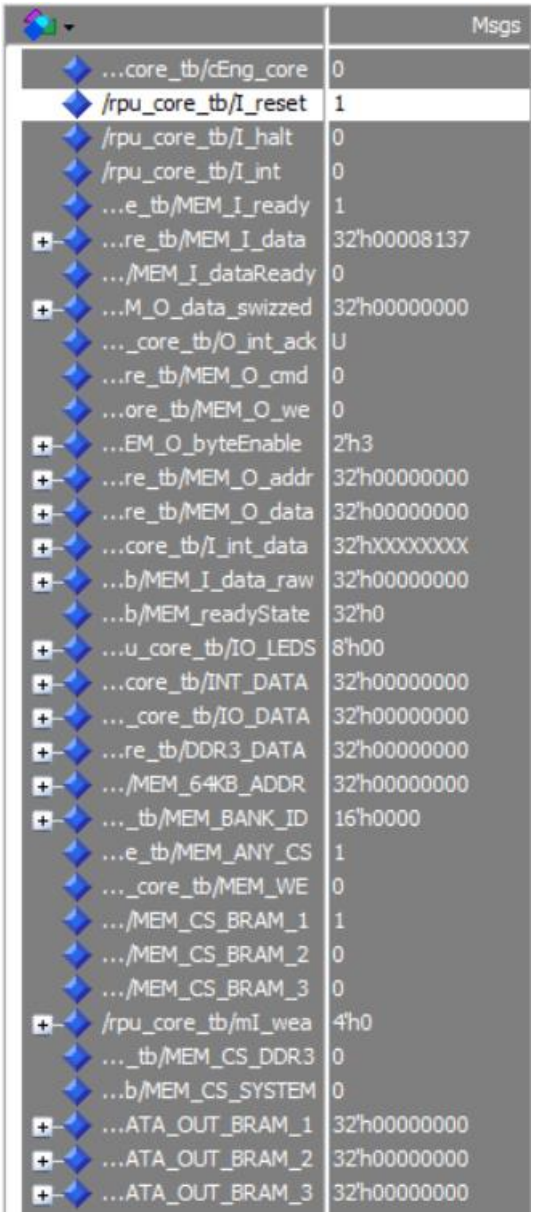
```
architecture Behavioral of rpu_core_tb is

    -- The RPU core definition
    COMPONENT core
        PORT(
            I_clk : IN  std_logic;
            I_reset : IN  std_logic;
            I_halt : IN  std_logic;
                  -- External Interrupt interface
            I_int_data: in STD_LOGIC_VECTOR(31 downto 0);
            I_int: in STD_LOGIC;
            O_int_ack: out STD_LOGIC;

            MEM_O_cmd : OUT  std_logic;
            MEM_O_we : OUT  std_logic;

            MEM_O_byteEnable : OUT  std_logic_vector(1 downto 0);
            MEM_O_addr : OUT  std_logic_vector(31 downto 0);
            MEM_O_data : OUT  std_logic_vector(31 downto 0);
            MEM_I_data : IN  std_logic_vector(31 downto 0);

            MEM_I_ready : IN  std_logic;
            MEM_I_dataReady : IN  std_logic

            ;
            O_DBG:out std_logic_vector(XLEN32M1 downto 0)
        );
    END COMPONENT;
```

具体代码见 rpu_core_tb.vht。

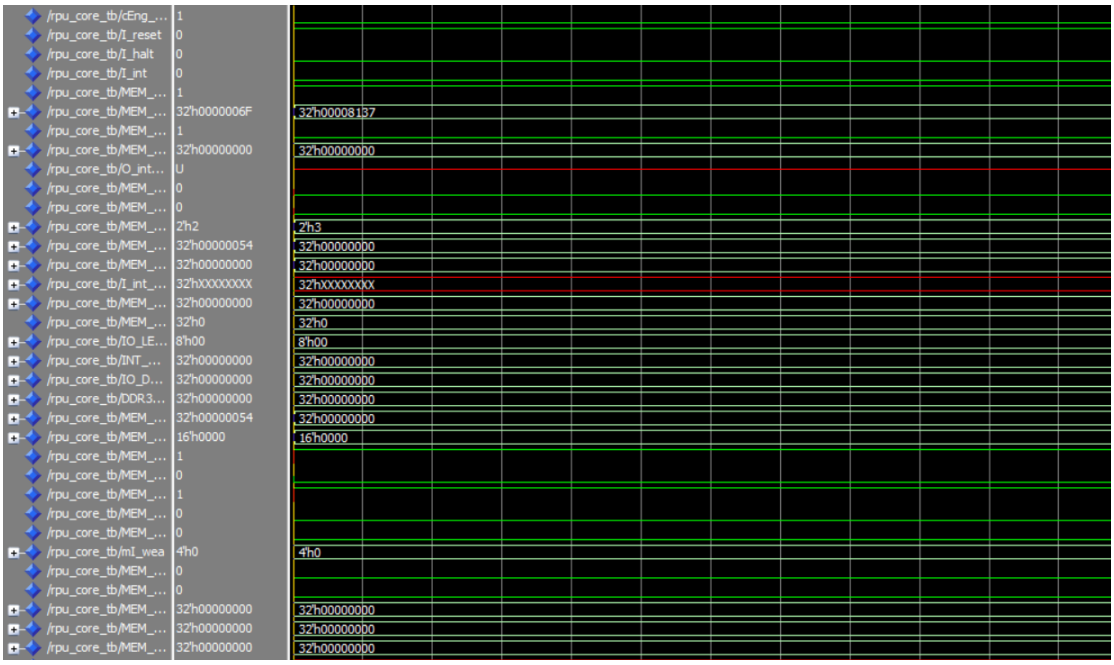
仿真过程：

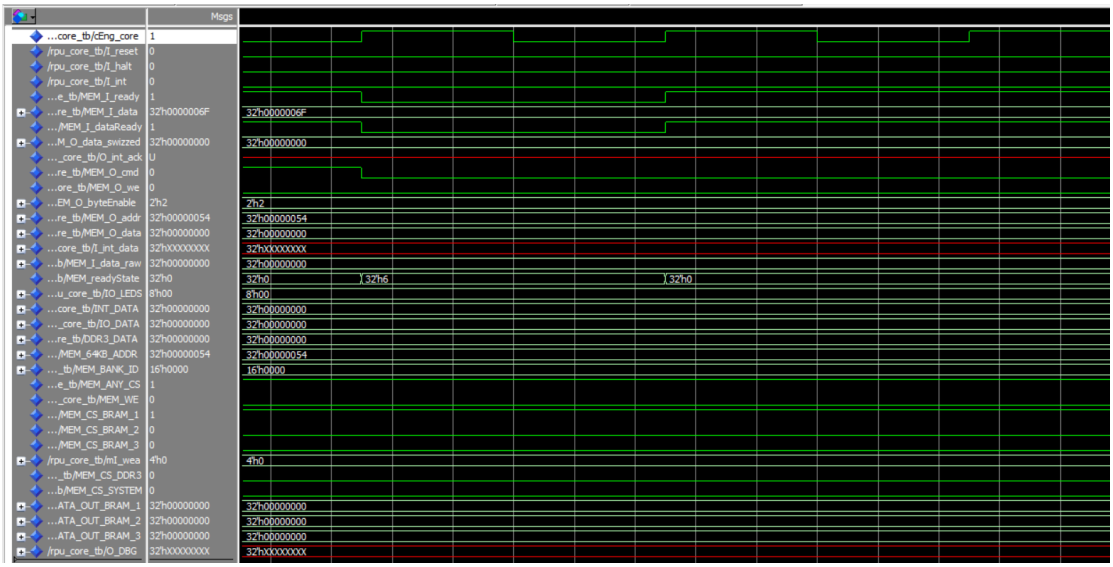

0 ps to 1 ns
Now: 89,907,375 ns Delta: 2
0 ps to 1 ns
Now: 160,523,975 ns Delta: 3
0 ps to 1 ns
Now: 196,550,825 ns Delta: 5

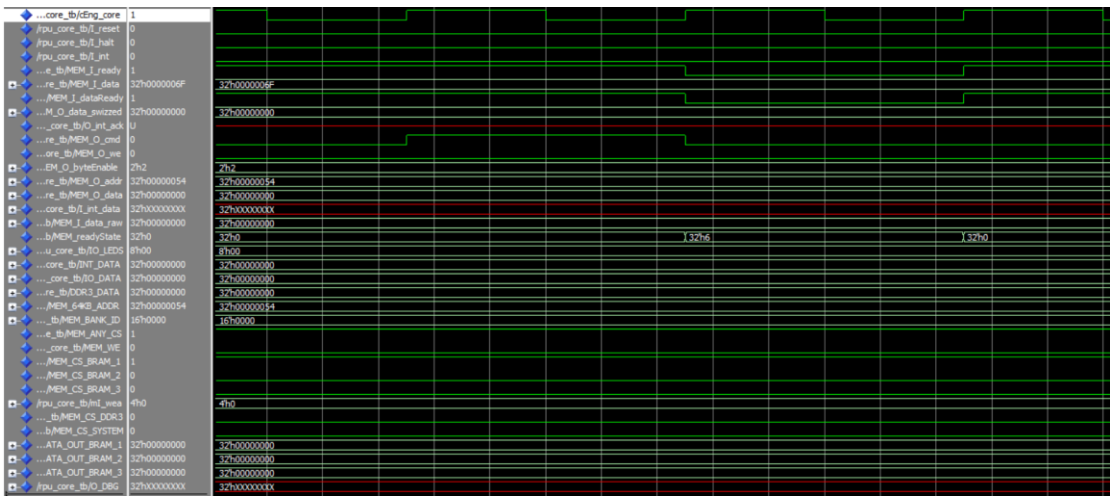
变量：


Msgs
...core_tb/cEng_core 0
/rpu_core_tb/I_reset 1
/rpu_core_tb/I_halt 0
/rpu_core_tb/I_int 0
...e_tb/MEM_I_ready 1
...re_tb/MEM_I_data 32'h00008137
.../MEM_I_dataReady 0
...M_O_data_swizzed 32'h00000000
..._core_tb/O_int_ack U
...re_tb/MEM_O_cmd 0
...ore_tb/MEM_O_we 0
...EM_O_byteEnable 2'h3
...re_tb/MEM_O_addr 32'h00000000
...re_tb/MEM_O_data 32'h00000000
...core_tb/I_int_data 32'hXXXXXXXX
...b/MEM_I_data_raw 32'h00000000
...b/MEM_readyState 32'h0
...u_core_tb/IO_LEDS 8'h00
...core_tb/INT_DATA 32'h00000000
..._core_tb/IO_DATA 32'h00000000
...re_tb/DDR3_DATA 32'h00000000
.../MEM_64KB_ADDR 32'h00000000
..._tb/MEM_BANK_ID 16'h0000
...e_tb/MEM_ANY_CS 1
..._core_tb/MEM_WE 0
.../MEM_CS_BRAM_1 1
.../MEM_CS_BRAM_2 0
.../MEM_CS_BRAM_3 0
/rpu_core_tb/mI_wea 4'h0
..._tb/MEM_CS_DDR3 0
...b/MEM_CS_SYSTEM 0
...ATA_OUT_BRAM_1 32'h00000000
...ATA_OUT_BRAM_2 32'h00000000
...ATA_OUT_BRAM_3 32'h00000000

信号跳转：


/rpu_core_tb/cEng_... 1
/rpu_core_tb/I_reset 0
/rpu_core_tb/I_halt 0
/rpu_core_tb/I_int 0
/rpu_core_tb/MEM_... 1
/rpu_core_tb/MEM_... 32'h0000006F
32'h00008137
/rpu_core_tb/MEM_... 1
/rpu_core_tb/MEM_... 32'h00000000
32'h00000000
/rpu_core_tb/O_int... U
/rpu_core_tb/MEM_... 0
/rpu_core_tb/MEM_... 0
/rpu_core_tb/MEM_... 2'h2
2'h3
/rpu_core_tb/MEM_... 32'h00000054
32'h00000000
/rpu_core_tb/MEM_... 32'h00000000
32'h00000000
/rpu_core_tb/I_int_... 32'hXXXXXXXX
32'hXXXXXXXX
/rpu_core_tb/MEM_... 32'h00000000
32'h00000000
/rpu_core_tb/MEM_... 32'h0
32'h0
/rpu_core_tb/IO_LE... 8'h00
8'h00
/rpu_core_tb/INT_... 32'h00000000
32'h00000000
/rpu_core_tb/IO_D... 32'h00000000
32'h00000000
/rpu_core_tb/DDR3... 32'h00000000
32'h00000000
/rpu_core_tb/MEM_... 32'h00000054
32'h00000000
/rpu_core_tb/MEM_... 16'h0000
16'h0000
/rpu_core_tb/MEM_... 1
/rpu_core_tb/MEM_... 0
/rpu_core_tb/MEM_... 1
/rpu_core_tb/MEM_... 0
/rpu_core_tb/MEM_... 0
/rpu_core_tb/mI_wea 4'h0
4'h0
/rpu_core_tb/MEM_... 0
/rpu_core_tb/MEM_... 0
/rpu_core_tb/MEM_... 32'h00000000
32'h00000000
/rpu_core_tb/MEM_... 32'h00000000
32'h00000000
/rpu_core_tb/MEM_... 32'h00000000
32'h00000000


Msgs
...core_tb/cEng_core 1
/rpu_core_tb/I_reset 0
/rpu_core_tb/I_halt 0
/rpu_core_tb/I_int 0
...e_tb/MEM_I_ready 1
...re_tb/MEM_I_data 32'h0000006F
32'h0000006F
.../MEM_I_dataReady 1
...M_O_data_swizzed 32'h00000000
32'h00000000
..._core_tb/O_int_ack U
...re_tb/MEM_O_cmd 0
...ore_tb/MEM_O_we 0
...EM_O_byteEnable 2'h2
2'h2
...re_tb/MEM_O_addr 32'h00000054
32'h00000054
...re_tb/MEM_O_data 32'h00000000
32'h00000000
...core_tb/I_int_data 32'hXXXXXXXX
32'hXXXXXXXX
...b/MEM_I_data_raw 32'h00000000
32'h00000000
...b/MEM_readyState 32'h0
32'h0
32'h6
32'h0
...u_core_tb/IO_LEDS 8'h00
8'h00
...core_tb/INT_DATA 32'h00000000
32'h00000000
..._core_tb/IO_DATA 32'h00000000
32'h00000000
...re_tb/DDR3_DATA 32'h00000000
32'h00000000
.../MEM_64KB_ADDR 32'h00000054
32'h00000054
..._tb/MEM_BANK_ID 16'h0000
16'h0000
...e_tb/MEM_ANY_CS 1
..._core_tb/MEM_WE 0
.../MEM_CS_BRAM_1 1
.../MEM_CS_BRAM_2 0
.../MEM_CS_BRAM_3 0
/rpu_core_tb/mI_wea 4'h0
4'h0
..._tb/MEM_CS_DDR3 0
...b/MEM_CS_SYSTEM 0
...ATA_OUT_BRAM_1 32'h00000000
32'h00000000
...ATA_OUT_BRAM_2 32'h00000000
32'h00000000
...ATA_OUT_BRAM_3 32'h00000000
32'h00000000
/rpu_core_tb/O_DBG 32'hXXXXXXXX
32'hXXXXXXXX


...core_tb/cEng_core 1
/rpu_core_tb/I_reset 0
/rpu_core_tb/I_halt 0
/rpu_core_tb/I_int 0
...e_tb/MEM_I_ready 1
...re_tb/MEM_I_data 32'h0000006F
32'h0000006F
.../MEM_I_dataReady 1
...M_O_data_swizzed 32'h00000000
32'h00000000
..._core_tb/O_int_ack U
...re_tb/MEM_O_cmd 0
...ore_tb/MEM_O_we 0
...EM_O_byteEnable 2'h2
2'h2
...re_tb/MEM_O_addr 32'h00000054
32'h00000054
...re_tb/MEM_O_data 32'h00000000
32'h00000000
...core_tb/I_int_data 32'hXXXXXXXX
32'hXXXXXXXX
...b/MEM_I_data_raw 32'h00000000
32'h00000000
...b/MEM_readyState 32'h0
32'h0
32'h6
32'h0
...u_core_tb/IO_LEDS 8'h00
8'h00
...core_tb/INT_DATA 32'h00000000
32'h00000000
..._core_tb/IO_DATA 32'h00000000
32'h00000000
...re_tb/DDR3_DATA 32'h00000000
32'h00000000
.../MEM_64KB_ADDR 32'h00000054
32'h00000054
..._tb/MEM_BANK_ID 16'h0000
16'h0000
...e_tb/MEM_ANY_CS 1
..._core_tb/MEM_WE 0
.../MEM_CS_BRAM_1 1
.../MEM_CS_BRAM_2 0
.../MEM_CS_BRAM_3 0
/rpu_core_tb/mI_wea 4'h0
4'h0
..._tb/MEM_CS_DDR3 0
...b/MEM_CS_SYSTEM 0
...ATA_OUT_BRAM_1 32'h00000000
32'h00000000
...ATA_OUT_BRAM_2 32'h00000000
32'h00000000
...ATA_OUT_BRAM_3 32'h00000000
32'h00000000
/rpu_core_tb/O_DBG 32'hXXXXXXXX
32'hXXXXXXXX

分析和结论

从测试记录来看，RISC-V 能够进行信号跳转，并且实现取指令-译码-执行-访存-写回的功能。

根据分析结果，可以认为编写的 RISC-V 实现了所要求的功能，完成了实验目标。

实验心得体会

这个实验是用 VHDL 语言写一个 RISC-V 架构的 CPU，一开始觉得很难，做起来也确定挺难的。虽然有以前做 CPU 的经验，但是 RISC-V 是一个 32 位的，要做数十条指令，确实超出了自己的想象。后来在网上找了很多的开源代码，进行了一些修改，最终完成了这次实验，在这个过程中也体验到了 testbench 文件的好处，如果要手动一个一个自己输入命令将会是一个很浩大的工程!