LC2K 硬件仿真实验报告

13349051 计科一班 劳嘉辉

实验目的

使用硬件实现 LC2K 的多周期处理器。在本次试验中，先使用 VHDL 语言在 modelsim 软件模拟 LC2K 的多周期处理器。

实验问题

本次实验的难点主要如下：

1、 各组件的实现。

2、 FSM controller 的设计。

3、 VHDL 语言的技巧。

实验调试

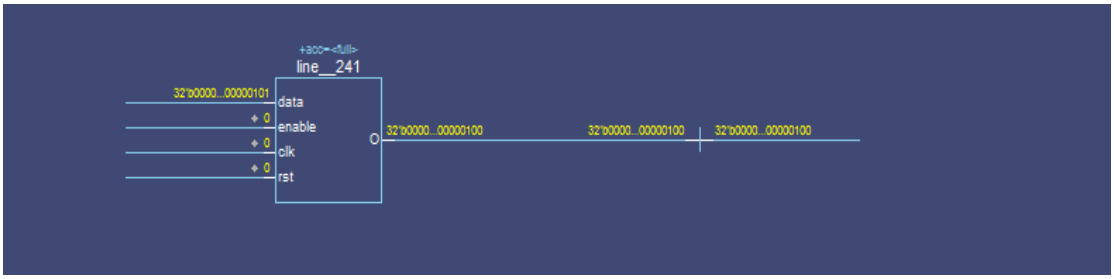
在 RAM_TEST 中我提供了几个测试文件。每个 VHDL 文件都是一个 RAM MODULE 部分的代码。区别在于初始化时的值不一样。测试的时候只要将这些文件的 ram module 的代码与主文件的 ram module 部分更换即可。如果想看测试的是什么指令，请点开“测试例子的指令.txt”查看

实验过程

一、 先使用 VHDL 实现各个 component

在本次实验中，需要用到以下的部件：

PC reg :



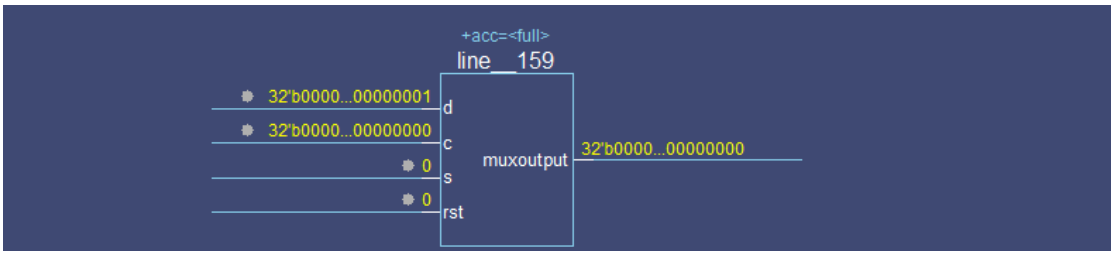
寄存器 VHDL 代码部分是由老师给出的，在这里不再重复。

PC reg 在程序中的 map 映射：

U_PC : reg port map (clk, PCBRANCHen ,ALUresult, PCout);

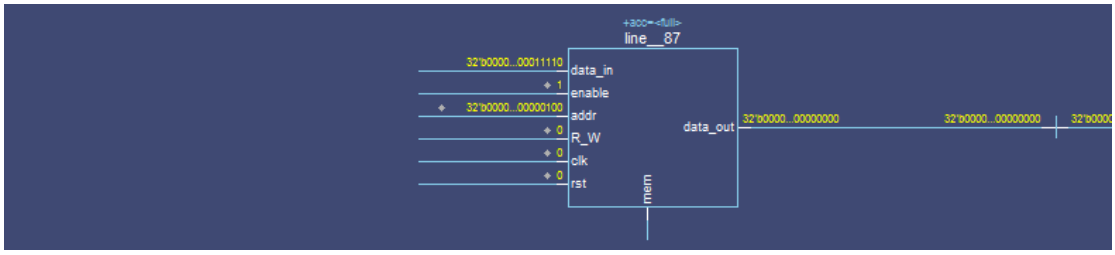
该寄存器由 clk 的上升沿触发，由 signal PCBRANCHen 控制使能，输入为 ALU RESULT 寄存器的结果，输出为 PCOUT。

ADDR MUX 地址二选一多选器



控制信号为 0 时，输出 PCOUT，控制信号为 1 时，输出 ALU RESULT 多选器不需要时钟的上升沿触发。只有一个 std_logic 的 s 控制信号以及一个 rst 的清零信号。

RAM　　内存



RAM 在 lc2k 中的映射

U_ram : ram_module port map (rst,clk, MEMen, RW, ADDRout, REGout2, RAMout);

该部件由信号 rst 清零并将数据存进 Memory 中。由信号 clk 触发。由 memen 信号控制使能。信号 R_W 控制读写的过程。信号 ADDRout 选择地址。

在这里声明了一个 65536 的大小的数组用来模拟内存。

FSM controller 有限状态机

Rst='1' 为清零状态初始化状态。

在这里，由于 PC reg 和 RAM 的输出都需要 clk 的触发，所以说从 fetch state 到 decode state 就不止一个 cycle 了。

FET_0：

PC_en <= '0';

Mem_mux <= '0';//选择了来自 pc 寄存器的地址

Mem_en <= '1';//使能内存

Mem_rw <= '0';//内存处于 read 状态

IR_en <= '1';

Dest_mux <= '0';

RData_mux <= '0';

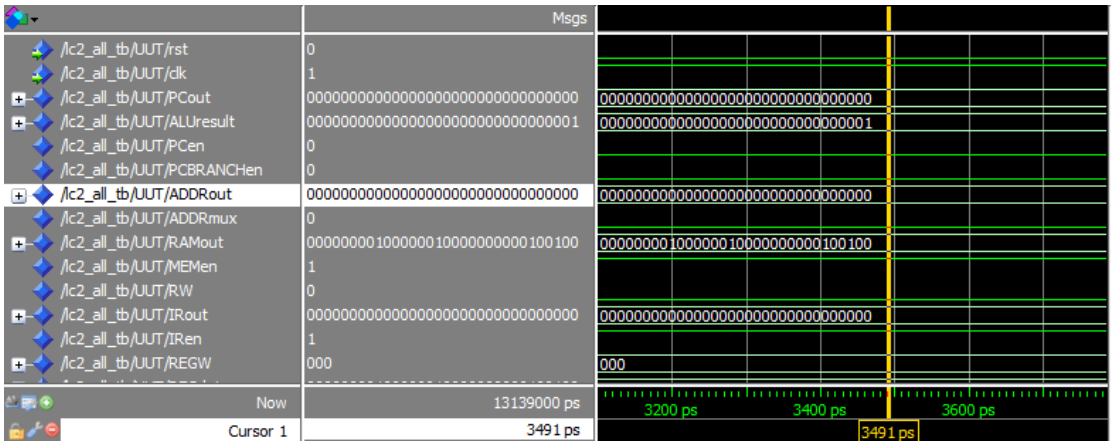
Reg_en <= '0';

ALU_mux1 <= '0';//选择了 PC reg 的值

ALU_mux2 <= "01";//选择了“1”

ALU_op <= '0';//算出了 pc+1 的值

DEC_1 和 DEC_2 是为 DEC_3 而准备的。由于 RAM 和 IR REG 都是需要在使能后在一下个时延才能装入。所以这里的话需要拖延两个 stage 才能够在 DEC_3 中正确 decode。



由图可以看出，在 DEC_1 这个阶段中，虽然 RAMOUT 已经有正确的值且 IREN 已经是 1，可是因为不是 clk 上升沿，因此并没有读入 IReg 中。

R 型指令(ADD 为例)

```
--Add State 4------------------------------------------------------

when ADD_4=>

cpu_state <= ADD_5;

PC_en <= '0';

Mem_mux <= '0';

Mem_en <= '0';

Mem_rw <= '0';

IR_en <= '0';

Dest_mux <= 0';

RData_mux <= '0';

Reg_en <= '0';

ALU_mux1 <= '1';//读入的是 REGA 的值

ALU_mux2 <= "00";//读入的是 REGB 的值

ALU_op <= '0';//进行加法


--Add State 5------------------------------------------------------

when ADD_5=>

cpu_state <= FET_0;

PC_en <= '0';

Mem_mux <= '0';

Mem_en <= '0';
```

```
Mem_rw <= '0';

IR_en <= '0';

Dest_mux <= '1';

RData_mux <= '1';

Reg_en <= '1';

ALU_mux1 <= '0';

ALU_mux2 <= "01";

ALU_op <= '0';
```

L 型指令（以 LOAD 为例）

```
--Load State 8-----------------------------------------------------

when LW_8=>

cpu_state <= LW_9;

PC_en <= '0';

Mem_mux <= '0';

Mem_en <= '0';

Mem_rw <= '0';

IR_en <= '0';

Dest_mux <= '0';

RData_mux <= '0';

Reg_en <= '0';

ALU_mux1 <= '1';

ALU_mux2 <= "11";

ALU_op <= '0';


--Load State 9-----------------------------------------------------

when LW_9=>

cpu_state <= LW_10;

PC_en <= '0';

Mem_mux <= '1';

Mem_en <= '1';
```

```
Mem_rw <= '0';
IR_en <= '0';
Dest_mux <= '0';
RData_mux <= '0';
Reg_en <= '0';
ALU_mux1 <= '0';
ALU_mux2 <= "00";
ALU_op <= '0';
--Load State 10-----------------------------------------------------
when LW_10=>
cpu_state <= FET_0;
PC_en <= '0';
Mem_mux <= '0';
Mem_en <= '0';
Mem_rw <= '0';
IR_en <= '0';
Dest_mux <= '0';
RData_mux <= '0';
Reg_en <= '1';
ALU_mux1 <= '0';
ALU_mux2 <= "00";
ALU_op <= '0';
```

BEQ 指令

```
--branch equal State 13-----------------------------------------------
when BEQ_13=>
cpu_state <= BEQ_14;
PC_en <= '0';
Mem_mux <= '0';
Mem_en <= '0';
Mem_rw <= '0';
IR_en <= '0';
Dest_mux <= '0';
RData_mux <= '0';
Reg_en <= '0';
ALU_mux1 <= '0';
ALU_mux2 <= "11";
ALU_op <= '0';
//这个阶段是计算跳转的地址
--branch equal State 14-----------------------------------------------
when BEQ_14=>
cpu_state <= FET_0;
PC_en <= '0';
Mem_mux <= '0';
Mem_en <= '0';
```

```
Mem_rw <= '0';
IR_en <= '0';
Dest_mux <= '0';
RData_mux <= '0';
Reg_en <= '0';
ALU_mux1 <= '1';
ALU_mux2 <= "00";
ALU_op <= '0';
```

//这个阶段比较的是 REGA 和 REGB 的值。这里用到了 ALU 的 equ 信号。

我们来看看 ALU 的声明：

```
--LC-2 ALU-------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity LC2_ALU is
port( rst : in std_logic;
A: in std_logic_vector (31 downto 0);
B: in std_logic_vector (31 downto 0);
S: in std_logic; --_vector (1 downto 0);
```

```
O: out std_logic_vector (31 downto 0);
EQ: out std_logic);
end LC2_ALU;

architecture bhv of LC2_ALU is
begin
process(A, B, S,rst)
begin
     if(rst ='1') then
        O<="00000000000000000000000000000000";
        EQ<='0';
     else
case S is
when '0' => O <= A+B;
when '1' => O <= not (A and B);
when others => null;
end case;
if (A=B) then EQ <= '1';
else EQ <= '0';
end if;
end if;
end process;
```

```
end bhv;
```

当 A==B 时，EQ=1；这个信号被传送到 BRANCH 原件中。

该原件在 LC2K 中的映射：

```
U_PC_BRANCH_EN: branch port map(rst,ALUeql,IRout(24 downto 22),PCen,PCBRANCHen);
--branch equal control--------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity branch is
   port(rst :in std_logic;
      beq :in std_logic;
      opcode :in std_logic_vector(2 downto 0);
      pccontrol :in std_logic;
      pcen:out std_logic);
   end branch;

architecture bhv of branch is
begin
   process(rst,beq,opcode,pccontrol)
   variable isbeq : std_logic;
   variable gate: std_logic :='0';
   begin
```

```
        if(rst = '1') then
            pcen <= '0';
         else
    case opcode is
    when "100" => isbeq:='1';
    when others => isbeq:='0';
    end case;
    if isbeq='1' and beq = '1' then
          gate:='1';
       else
          gate:='0';
    end if;
    pcen<= (gate or pccontrol);
end if;
end process;
end architecture bhv;
```

这里的功能其实很简单。如果 opcode 为“100”的时候，isbeq 信号就会变成‘1’，其他情况就会变成‘0’。如果 isbeq 和 ALU 的 beq 信号同时为高时，那么 gate 信号就会变成‘1’。而 pc 的 control 就由 gate 和 FSM 控制的 pccontrol 相与来决定。

J 型指令

```
--jump register State 15-----------------------------------------------
when JALR_15=>
cpu_state <= JALR_16;
PC_en <= '0';
Mem_mux <= '0';
Mem_en <= '0';
Mem_rw <= '0';
IR_en <= '0';
Dest_mux <= '0';
RData_mux <= '1';
Reg_en <= '1';
ALU_mux1 <= '0';
ALU_mux2 <= "10";
ALU_op <= '0';
```

//这个阶段是为了让 regb 读入 pc+1 的值。

```
--jump register State 16-----------------------------------------------
when JALR_16=>
cpu_state <= JALR_17;
PC_en <= '0';
Mem_mux <= '0';
Mem_en <= '0';
```

```
Mem_rw <= '0';

IR_en <= '0';

Dest_mux <= '0';

RData_mux <= '0';

Reg_en <= '0';

ALU_mux1 <= '1';

ALU_mux2 <= "10";

ALU_op <= '0';

--jump register State 17----------------------------------------------

when JALR_17=>

cpu_state <= FET_0;

PC_en <= '1';

Mem_mux <= '0';

Mem_en <= '0';

Mem_rw <= '0';

IR_en <= '0';

Dest_mux <= '0';

RData_mux <= '0';

Reg_en <= '0';

ALU_mux1 <= '1';

ALU_mux2 <= "10";

ALU_op <= '0';
```

//这两个阶段是为了让 rega 的值读入 pc，类似于 ADD 指令。

TEST BENCH（测试平台）

```
--Test bench-----------------------------------------------
library ieee;
use ieee.std_logic_arith.all;
use ieee.STD_LOGIC_UNSIGNED.all;
use ieee.std_logic_1164.all;


entity lc2_all_tb is
end lc2_all_tb;
```

//空的 entity

```
architecture TB_ARCHITECTURE of lc2_all_tb is
-- Component declaration of the tested unit
component LC2_all
port(rst : in std_logic;clk : in std_logic );
end component;
```

//调用 LC_2K 原件

```
signal rst : std_logic:='1';
signal clk : std_logic;
begin
process
begin
```

```
clk<='0';
wait for 1 ns;
clk<='1';
wait for 1 ns;
end process;
//模拟时钟
process
begin
rst <= '0' after 1 ns;
wait;
end process;
//模拟初始化信号
UUT : LC2_all port map(rst,clk);
end TB_ARCHITECTURE;

configuration TESTBENCH_FOR_LC2_all of lc2_all_tb is
for TB_ARCHITECTURE
for UUT : LC2_all
use entity work.LC2_all(str);
end for;
end for;
end TESTBENCH_FOR_LC2_all;
```

实验感想

虽然上个学期已经使用过 VERILOG 语言实现一个 ALU，但觉得还没有完全真正掌握汇编语言。这次的使用 VHDL 语言实现一个 LC2K multiply cycle processor。其实难度在于不熟悉信号的运作过程。刚开始的时候没有了解到信号是什么时候才被成功赋值，所以在给 memory 初始化的时候卡住了一段时间。另外在于设计 FSM 的时候 DECODE stage 的特殊性。由于在使用 if 语句进行 decode 的时候 instruction reg 的输出还不是一个指令，所以程序在执行到后来会出错。不过后来拖延了两个 stage 之后就成功解决了这个问题。而 beq 和 jalr 指令并没有想象中那么难，只要知道信号什么时候被赋值以及原件什么时候输出的话就很好解决了。

另外也熟悉地使用了 Modlesim 这个软件。也熟悉使用了不同的功能。这次的实验感觉收获挺多的。