实验报告

实验名称（相对简单 CPU 电路设计）

物联 1601　201608010424　潘婷婷

实验目标

利用 VHDL 设计相对简单 CPU 的电路并验证。

实验要求

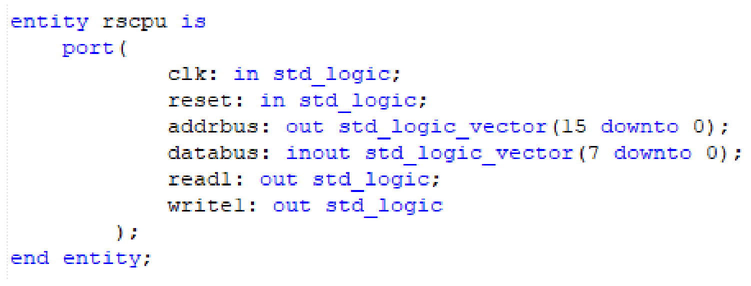
- 采用 VHDL 描述电路及其测试平台
- 采用时序逻辑设计电路
- 采用从 1 累加到 n 的程序进行测试

实验内容

相对简单 CPU 的设计需求

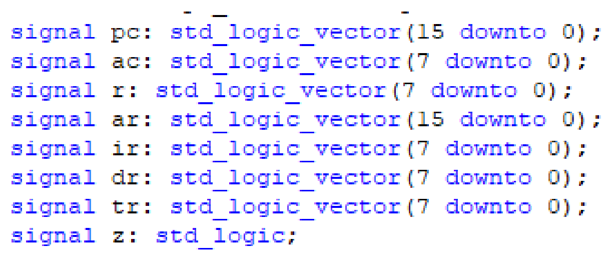
相对简单 CPU 的设计需求请详见课件，主要特征如下：

- 地址总线 16 位，数据总线 8 位



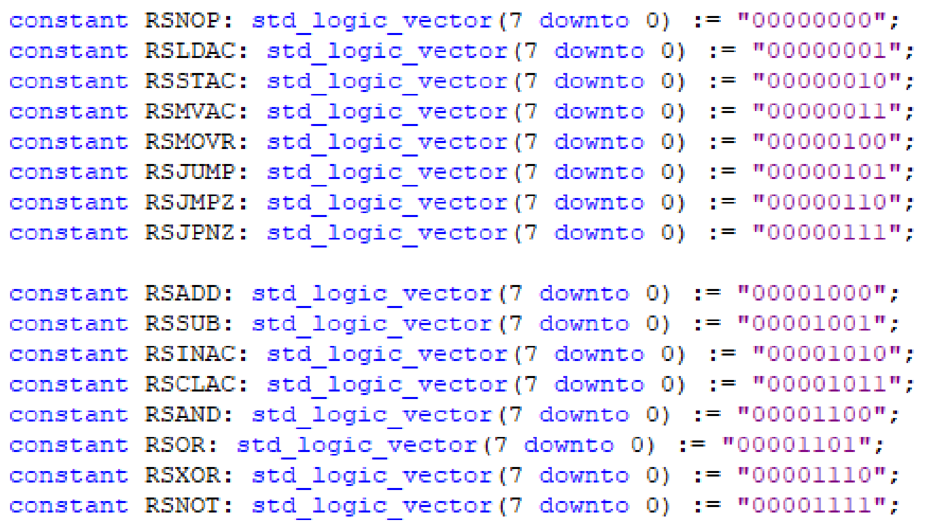
```
entity rscpu is
    port(
            clk: in std_logic;
            reset: in std_logic;
            addrbus: out std_logic_vector(15 downto 0);
            databus: inout std_logic_vector(7 downto 0);
            readl: out std_logic;
            writel: out std_logic
        );
end entity;
```

- 有一个 8 位累加寄存器 AC，一个 8 位通用寄存器 R，一个 1 位的零标志
- 有一个 16 位 AR 寄存器，一个 16 位程序计数器 PC，一个 8 位数据寄存器 DR，一个 8 位指令寄存器 IR，一个 8 位临时寄存器 TR



```
signal pc: std_logic_vector(15 downto 0);
signal ac: std_logic_vector(7 downto 0);
signal r: std_logic_vector(7 downto 0);
signal ar: std_logic_vector(15 downto 0);
signal ir: std_logic_vector(7 downto 0);
signal dr: std_logic_vector(7 downto 0);
signal tr: std_logic_vector(7 downto 0);
signal z: std_logic;
```

- 有 16 条指令，每条指令 1 个或 3 个字节，其中操作码 8 位。3 字节的指令有 16 位的地址

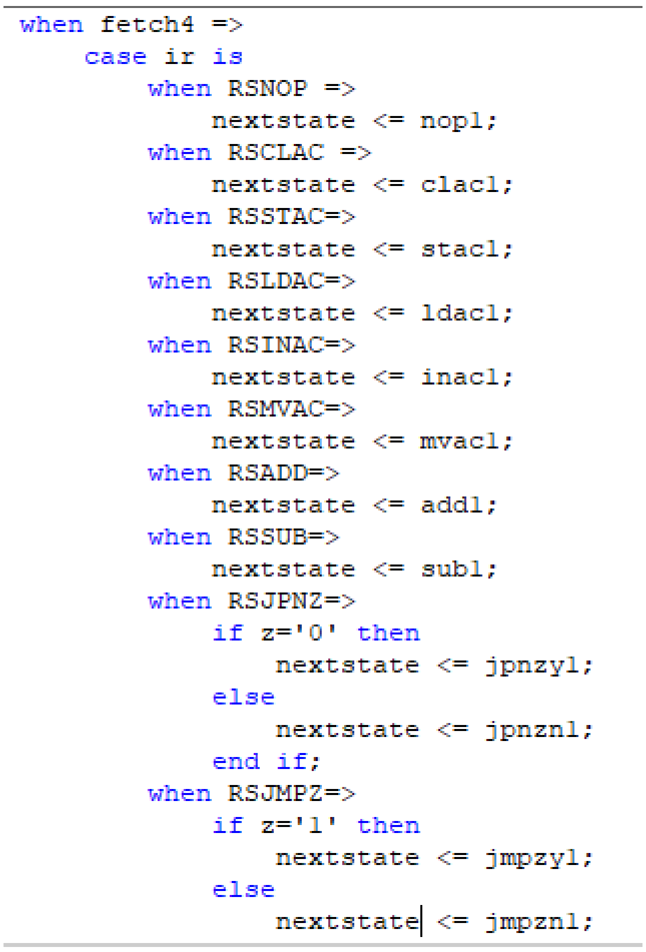


```
constant RSNOP: std_logic_vector(7 downto 0) := "00000000";
constant RSLDAC: std_logic_vector(7 downto 0) := "00000001";
constant RSSTAC: std_logic_vector(7 downto 0) := "00000010";
constant RSMVAC: std_logic_vector(7 downto 0) := "00000011";
constant RSMOVR: std_logic_vector(7 downto 0) := "00000100";
constant RSJUMP: std_logic_vector(7 downto 0) := "00000101";
constant RSJMPZ: std_logic_vector(7 downto 0) := "00000110";
constant RSJPNZ: std_logic_vector(7 downto 0) := "00000111";

constant RSADD: std_logic_vector(7 downto 0) := "00001000";
constant RSSUB: std_logic_vector(7 downto 0) := "00001001";
constant RSINAC: std_logic_vector(7 downto 0) := "00001010";
constant RSCLAC: std_logic_vector(7 downto 0) := "00001011";
constant RSAND: std_logic_vector(7 downto 0) := "00001100";
constant RSOR: std_logic_vector(7 downto 0) := "00001101";
constant RSXOR: std_logic_vector(7 downto 0) := "00001110";
constant RSNOT: std_logic_vector(7 downto 0) := "00001111";
```

相对简单 CPU 设计方案

相对简单 CPU 的设计方案请详见课件，主要思路如下：

1. 指令执行过程分为取指、译码、执行三个阶段
2. 取指包括三个状态，FETCH1，FETCH2，FETCH3，FETCH4
3. 译码体现为从 FETCH4 状态到各指令执行状态序列的第一个状态



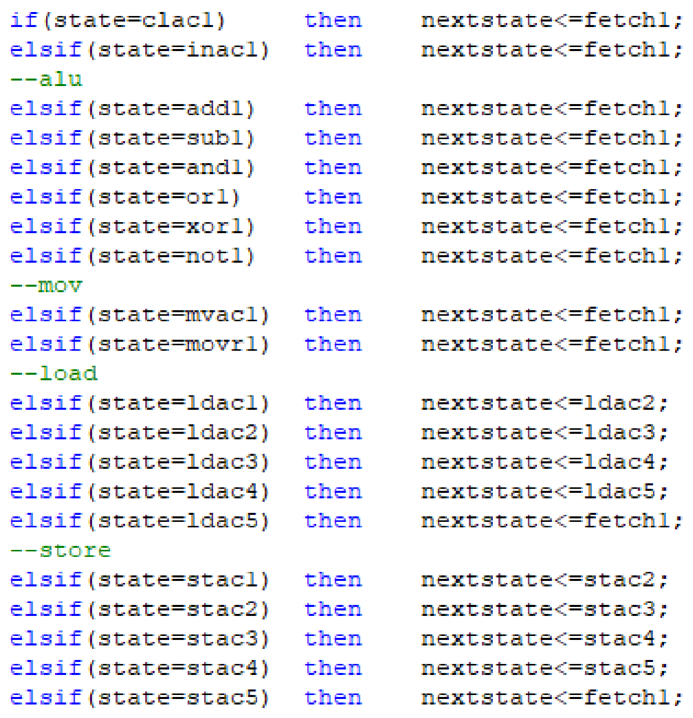
```
when fetch4 =>
    case ir is
        when RSNOP =>
            nextstate <= nop1;
        when RSCLAC =>
            nextstate <= clac1;
        when RSSTAC=>
            nextstate <= stac1;
        when RSLDAC=>
            nextstate <= ldac1;
        when RSINAC=>
            nextstate <= inac1;
        when RSMVAC=>
            nextstate <= mvac1;
        when RSADD=>
            nextstate <= add1;
        when RSSUB=>
            nextstate <= sub1;
        when RSJPNZ=>
            if z='0' then
                nextstate <= jpnzy1;
            else
                nextstate <= jpnzn1;
            end if;
        when RSJMPZ=>
            if z='1' then
                nextstate <= jmpzy1;
            else
                nextstate <= jmpzn1;
```

```
                end if;

            when others =>
                nextstate <= fetch1;
        end case;

    when others =>
        nextstate <= fetch1;
```

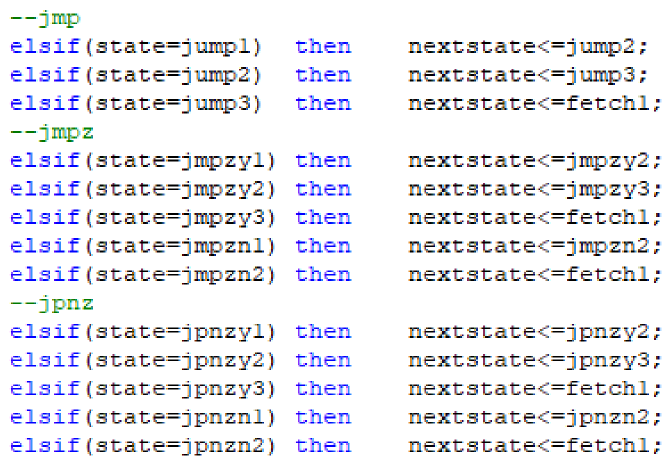
4. 执行根据指令的具体操作分为若干状态

```
constant fetch1: std_logic_vector(5 downto 0) := "000000";
constant fetch2: std_logic_vector(5 downto 0) := "000001";
constant fetch3: std_logic_vector(5 downto 0) := "000010";
constant fetch4: std_logic_vector(5 downto 0) := "000011";

constant nop1: std_logic_vector(5 downto 0)  := "000100";

constant ldac1: std_logic_vector(5 downto 0) := "000101";
constant ldac2: std_logic_vector(5 downto 0) := "000110";
constant ldac3: std_logic_vector(5 downto 0) := "000111";
constant ldac4: std_logic_vector(5 downto 0) := "001000";
constant ldac5: std_logic_vector(5 downto 0) := "001001";

constant stac1: std_logic_vector(5 downto 0) := "001010";
constant stac2: std_logic_vector(5 downto 0) := "001011";
constant stac3: std_logic_vector(5 downto 0) := "001100";
constant stac4: std_logic_vector(5 downto 0) := "001101";
constant stac5: std_logic_vector(5 downto 0) := "001110";

constant mvac1: std_logic_vector(5 downto 0) := "001111";

constant movr1: std_logic_vector(5 downto 0) := "010000";



constant jump1: std_logic_vector(5 downto 0)  := "010001";
constant jump2: std_logic_vector(5 downto 0)  := "010010";
constant jump3: std_logic_vector(5 downto 0)  := "010011";

constant jmpzn1: std_logic_vector(5 downto 0) := "010100";
constant jmpzn2: std_logic_vector(5 downto 0) := "010101";

constant jpnzn1: std_logic_vector(5 downto 0) := "010110";
constant jpnzn2: std_logic_vector(5 downto 0) := "010111";

constant jmpzy1: std_logic_vector(5 downto 0) := "011000";
constant jmpzy2: std_logic_vector(5 downto 0) := "011001";
constant jmpzy3: std_logic_vector(5 downto 0) := "011010";

constant jpnzy1: std_logic_vector(5 downto 0) := "011011";
constant jpnzy2: std_logic_vector(5 downto 0) := "011100";
constant jpnzy3: std_logic_vector(5 downto 0) := "011101";

constant add1: std_logic_vector(5 downto 0)   := "011110";
constant sub1: std_logic_vector(5 downto 0)   := "011111";
constant inac1: std_logic_vector(5 downto 0)  := "100000";
constant clac1: std_logic_vector(5 downto 0)  := "100001";
constant and1: std_logic_vector(5 downto 0)   := "100010";
constant or1: std_logic_vector(5 downto 0)    := "100011";
constant xor1: std_logic_vector(5 downto 0)   := "100100";
constant not1: std_logic_vector(5 downto 0)   := "100101";
```

5. 根据当前状态判断下一状态；执行的最后一个状态转移到 FETCH1 状态

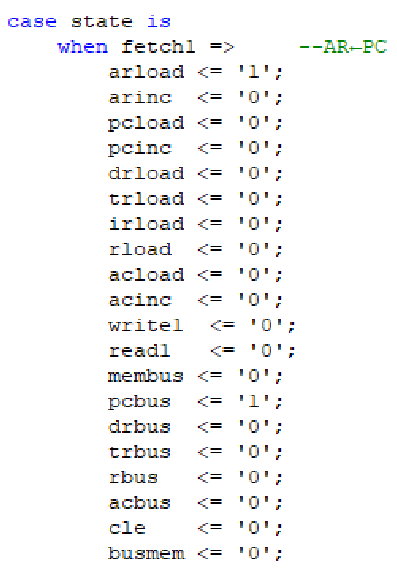


```
if(state=clac1)    then    nextstate<=fetch1;
elsif(state=inac1) then    nextstate<=fetch1;
--alu
elsif(state=add1)  then    nextstate<=fetch1;
elsif(state=sub1)  then    nextstate<=fetch1;
elsif(state=and1)  then    nextstate<=fetch1;
elsif(state=or1)   then    nextstate<=fetch1;
elsif(state=xor1)  then    nextstate<=fetch1;
elsif(state=not1)  then    nextstate<=fetch1;
--mov
elsif(state=mvac1) then    nextstate<=fetch1;
elsif(state=movr1) then    nextstate<=fetch1;
--load
elsif(state=ldac1) then    nextstate<=ldac2;
elsif(state=ldac2) then    nextstate<=ldac3;
elsif(state=ldac3) then    nextstate<=ldac4;
elsif(state=ldac4) then    nextstate<=ldac5;
elsif(state=ldac5) then    nextstate<=fetch1;
--store
elsif(state=stac1) then    nextstate<=stac2;
elsif(state=stac2) then    nextstate<=stac3;
elsif(state=stac3) then    nextstate<=stac4;
elsif(state=stac4) then    nextstate<=stac5;
elsif(state=stac5) then    nextstate<=fetch1;
```

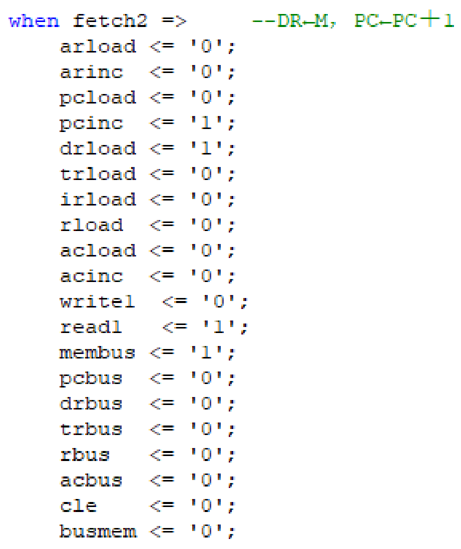


```
--jmp
elsif(state=jump1)  then    nextstate<=jump2;
elsif(state=jump2)  then    nextstate<=jump3;
elsif(state=jump3)  then    nextstate<=fetch1;
--jmpz
elsif(state=jmpzy1) then    nextstate<=jmpzy2;
elsif(state=jmpzy2) then    nextstate<=jmpzy3;
elsif(state=jmpzy3) then    nextstate<=fetch1;
elsif(state=jmpzn1) then    nextstate<=jmpzn2;
elsif(state=jmpzn2) then    nextstate<=fetch1;
--jpnz
elsif(state=jpnzy1) then    nextstate<=jpnzy2;
elsif(state=jpnzy2) then    nextstate<=jpnzy3;
elsif(state=jpnzy3) then    nextstate<=fetch1;
elsif(state=jpnzn1) then    nextstate<=jpnzn2;
elsif(state=jpnzn2) then    nextstate<=fetch1;
```

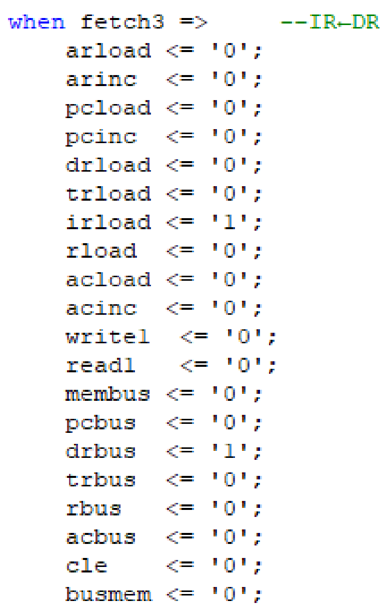
6. 控制器根据每个状态需要完成的操作产生相应的控制信号



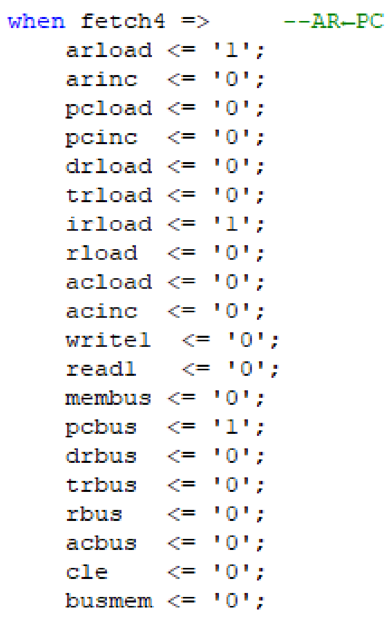
```
case state is
    when fetch1 =>      --AR←PC
        arload <= '1';
        arinc  <= '0';
        pcload <= '0';
        pcinc  <= '0';
        drload <= '0';
        trload <= '0';
        irload <= '0';
        rload  <= '0';
        acload <= '0';
        acinc  <= '0';
        write1  <= '0';
        read1   <= '0';
        membus <= '0';
        pcbus  <= '1';
        drbus  <= '0';
        trbus  <= '0';
        rbus   <= '0';
        acbus  <= '0';
        cle    <= '0';
        busmem <= '0';
```



```
when fetch2 =>      --DR←M, PC←PC＋1
    arload <= '0';
    arinc  <= '0';
    pcload <= '0';
    pcinc  <= '1';
    drload <= '1';
    trload <= '0';
    irload <= '0';
    rload  <= '0';
    acload <= '0';
    acinc  <= '0';
    write1  <= '0';
    read1   <= '1';
    membus <= '1';
    pcbus  <= '0';
    drbus  <= '0';
    trbus  <= '0';
    rbus   <= '0';
    acbus  <= '0';
    cle    <= '0';
    busmem <= '0';
```



```
when fetch3 =>      --IR←DR
    arload <= '0';
    arinc  <= '0';
    pcload <= '0';
    pcinc  <= '0';
    drload <= '0';
    trload <= '0';
    irload <= '1';
    rload  <= '0';
    acload <= '0';
    acinc  <= '0';
    write1  <= '0';
    read1   <= '0';
    membus <= '0';
    pcbus  <= '0';
    drbus  <= '1';
    trbus  <= '0';
    rbus   <= '0';
    acbus  <= '0';
    cle    <= '0';
    busmem <= '0';
```



```
when fetch4 =>      --AR←PC
    arload <= '1';
    arinc  <= '0';
    pcload <= '0';
    pcinc  <= '0';
    drload <= '0';
    trload <= '0';
    irload <= '1';
    rload  <= '0';
    acload <= '0';
    acinc  <= '0';
    write1  <= '0';
    read1   <= '0';
    membus <= '0';
    pcbus  <= '1';
    drbus  <= '0';
    trbus  <= '0';
    rbus   <= '0';
    acbus  <= '0';
    cle    <= '0';
    busmem <= '0';
```

其余状态同理，根据具体的需要执行的操作，产生相应的控制信号。

7. 根据控制信号执行相应的操作

```
-- address and data bus
addrbus <= ar;
databus <= dr when busmem='1' else "ZZZZZZZZ";


 --the bus
 thebus<=pc                          when pcbus='1'       else
         "00000000"&databus          when membus='1'      else
         "00000000"&r                when rbus='1'        else
         "00000000"&ac               when acbus='1'       else
         dr&tr                       when (trbus='1' and drbus='1')      else
         "00000000"&dr               when (drbus='1' and trbus/='1')     else
         "ZZZZZZZZZZZZZZZZ";



alu_result <= std_logic_vector(unsigned(ac)-unsigned(thebus(7 downto 0))) when state=sub1 else
              std_logic_vector(unsigned(ac)+unsigned(thebus(7 downto 0)))  when state=add1 else
              std_logic_vector(ac and thebus(7 downto 0)) when state=and1 else
              std_logic_vector(ac or thebus(7 downto 0)) when state=or1 else
              std_logic_vector(ac xor thebus(7 downto 0)) when state=xor1 else
              std_logic_vector( not ac) when state=not1 else
              thebus(7 downto 0);



update_regs: process(clk)
begin
    if(rising_edge(clk)) then

        if(arload='1') then
            ar <= thebus;
        end if;
        if(pcload='1') then
            pc <= thebus;
        end if;
        if(drload='1') then
            dr <= thebus(7 downto 0);
        end if;
        if(trload='1') then
            tr <= dr;
        end if;
        if(irload='1') then
            ir <= dr;
        end if;
        if(rload='1') then
            r <= thebus(7 downto 0);
        end if;

if(acload='1') then
    ac <= alu_result;
    if(state=ADD1 or state=SUB1 or state=NOT1 or state=AND1 or state=OR1 or state=XOR1) then
    if(alu_result="00000000") then
        z<='1';
    else
        z<='0';
    end if;
    end if;
end if;

if(arinc='1') then
    ar <= std_logic_vector(unsigned(ar)+1);
end if;
if(pcinc='1') then
    pc <= std_logic_vector(unsigned(pc)+1);
end if;
if(acinc='1') then
    ac <= std_logic_vector(unsigned(ac)+1);
    if(ac="11111111")   then     z<='1';
    else    z<='0';
    end if;
end if;
if(cle='1') then
    ac <= "00000000";
    z  <='1';
end if;
```

```
        if(reset='1') then
            pc <= X"0000";
            state <= fetch1;
        else
            --pc <= nextpc;
            state <= nextstate;
        end if;

    end if;
```

测试

测试平台

相对简单 CPU 电路在如下机器上进行了测试：

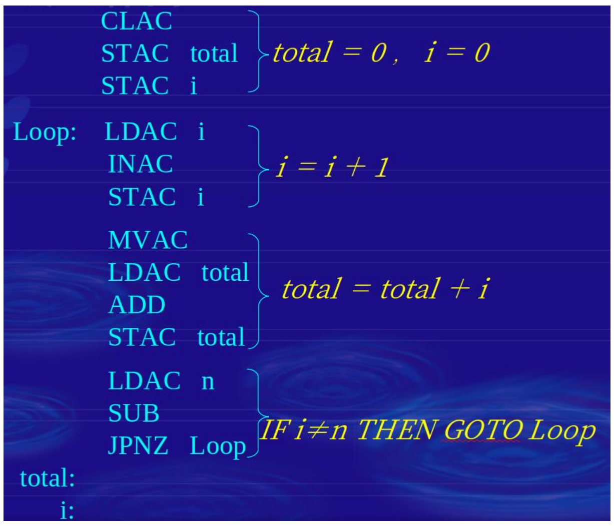
操作系统：Windows10

综合软件：Quartus II Help Version 9.0

仿真软件：Quartus II Help Version 9.0

测试输入

我们采用从 1 累加到 n 的程序作为测试输入：

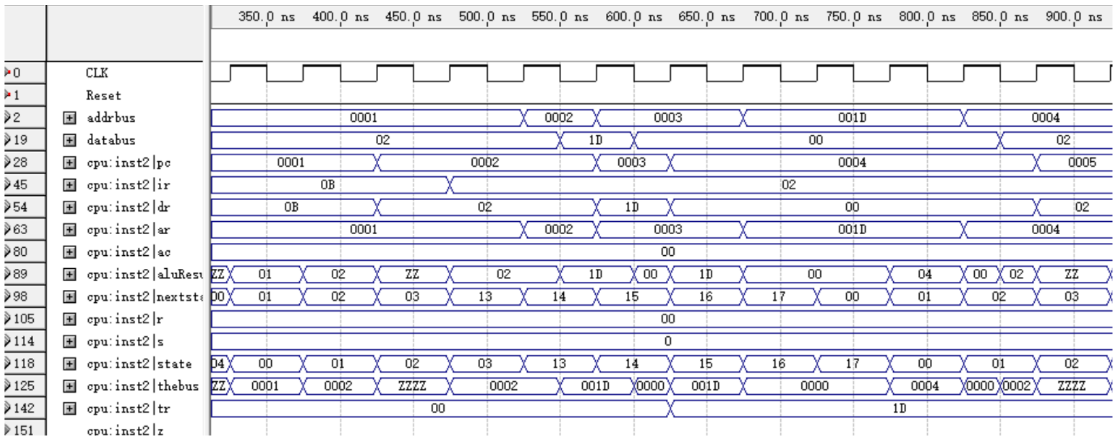


```
       CLAC
       STAC  total     } total = 0 ,  i = 0
       STAC  i
Loop:  LDAC  i
       INAC            } i = i + 1
       STAC  i
       MVAC
       LDAC  total     } total = total + i
       ADD
       STAC  total
       LDAC  n
       SUB             } IF i≠n THEN GOTO Loop
       JPNZ  Loop
total:
    i:
```

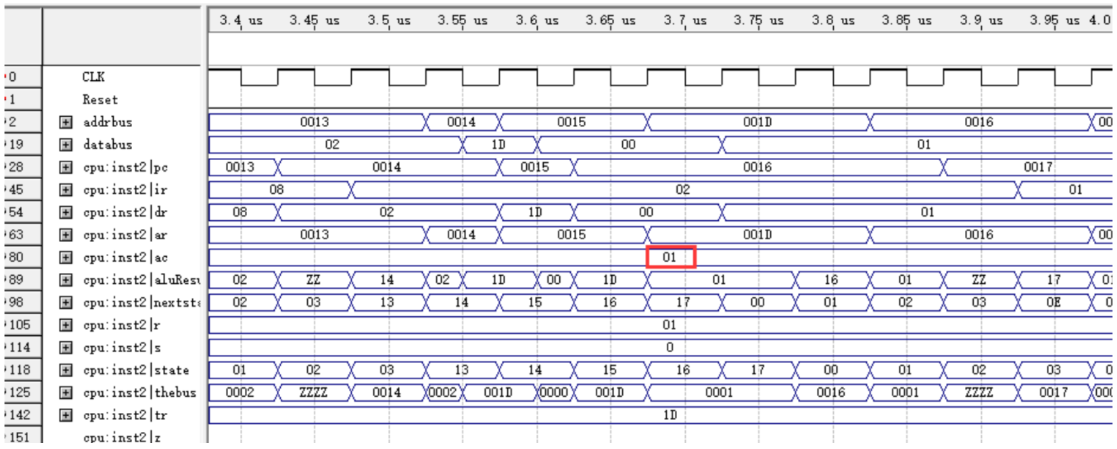
测试记录

相对简单 CPU 运行测试程序的 PC 寄存器、IR 寄存器、AC 累加器等信号波形截图如下：

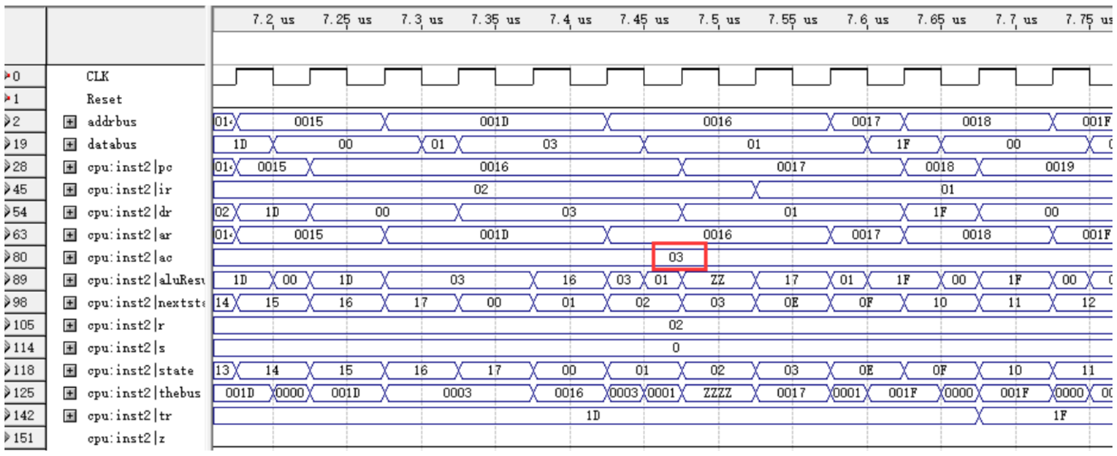
初始状态：



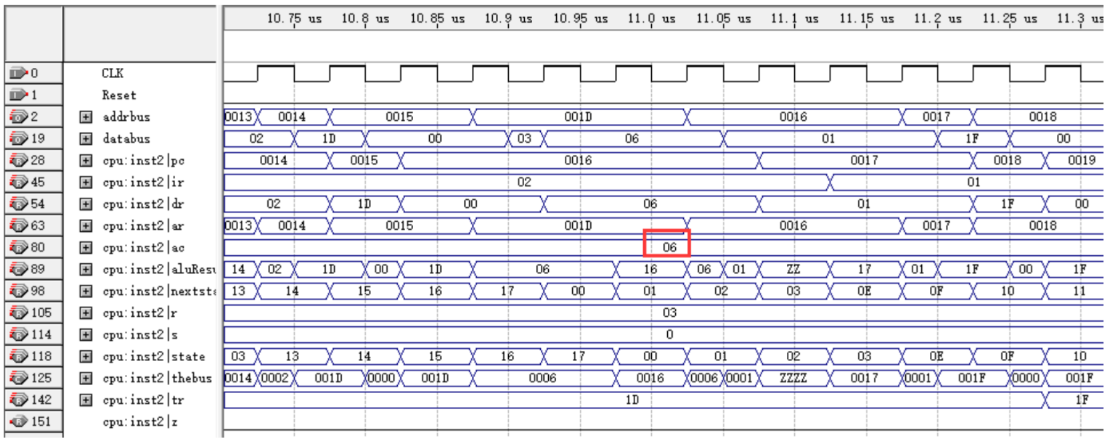
i=1,total=0   ->total=total+i=1



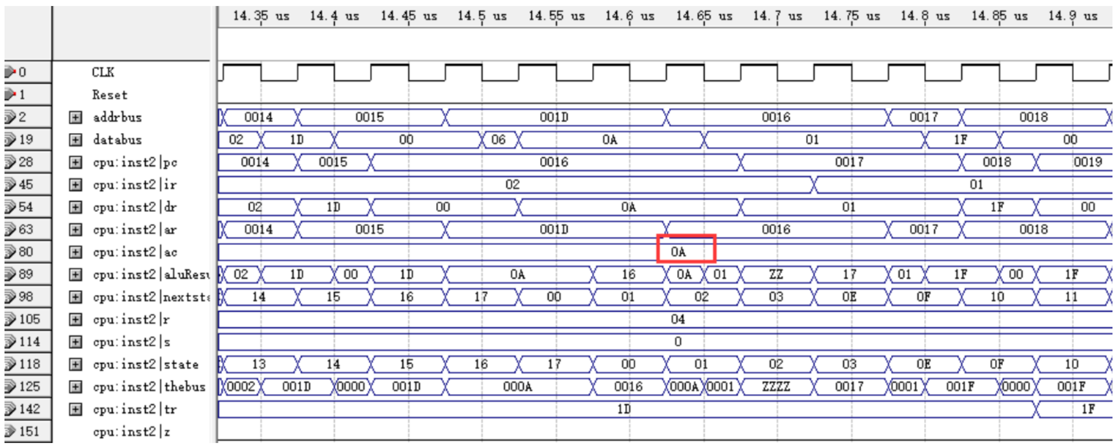
i=2,total=1   ->total=total+i=3



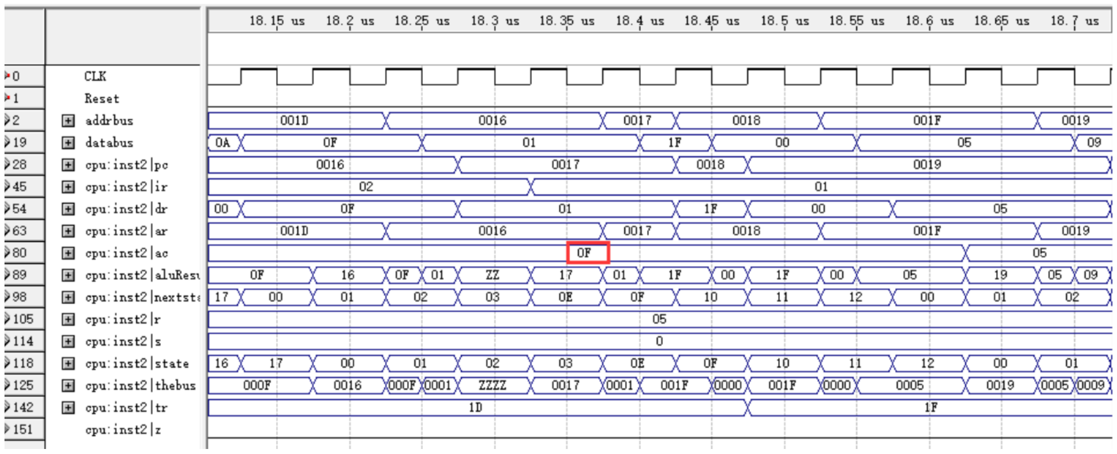
i=3,total=3 ->total=total+i=6



i=4,total=6 ->total=total+i=10(0xa)



i=5,total=10 ->total=total+i=15(0xf)



分析和结论

从测试记录来看，相对简单 CPU 实现了对测试程序指令的读取、译码和执行，得到的运算结果正确。

根据分析结果，可以认为所设计的相对简单 CPU 实现了所要求的功能，完成了实验目标。

这次实验也让我更加熟悉了计算机系统、指令执行的过程、VHDL 语言。虽然

之前也有设计过 CPU，但和这次实验也有所不同，这次实验实现的过程和之前不太一样，之前都是先设计好每个模块，再用模块图将每个模块连接起来。这次实验更注重对整体实现的流程，运行过程的把握，更用了一种纯 VHDL 语言的描述方式。在完成实验的过程中，不断学习，了解并熟悉运行过程，更有大家一起讨论交流自己遇到的问题，解决方法，实现过程。