[EDA 与 VHDL 实验报告]

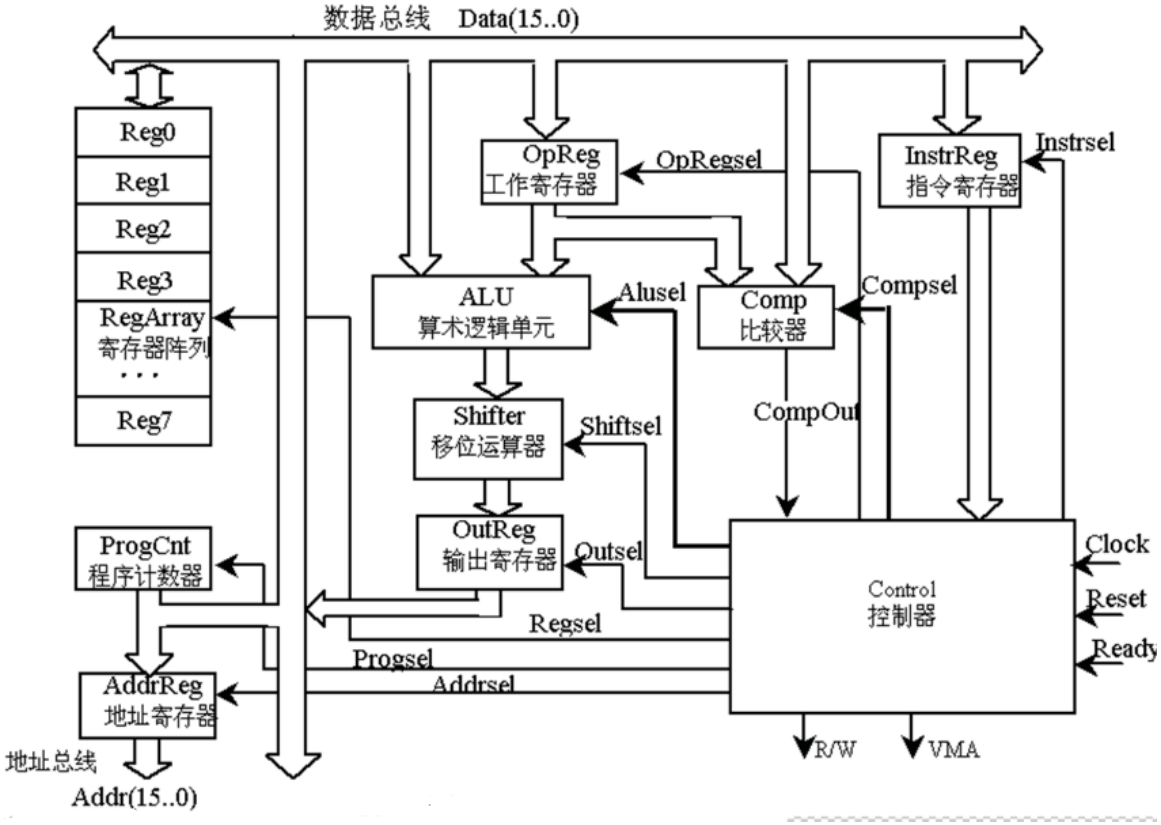
[16 位 CPU 设计]

16 位 cpu 设计

一、 顶层系统设计

1.1 组成结构

CPU 的结构如图所示。这是一个采用单总线系统架构的复杂指令系统结构的 16 位 CPU 。处理器包含了各种基本器件模块。他们是 8 个 16 位的寄存器 reg0~ reg7、一个运算器 ALU、一个移位寄存器 Shifter、一个程序计数器 PortCnt、 一个指令寄存器 InstrReg、一个比较器 Comp、一个地址寄存器 Addreg、和一个控制单元 Control。这些模块公共用一组 16 位的三台数据总线。



系统采用自顶向下的方法进行设计。顶层设计由微处理器和存储器通过一组双向数据总线连接，它们由一组地址总线和一些控制总线组成。处理器从外存储器中读取指令，并通过执行这些指令来运行程序。这些指令存储在指令寄存器中，并由控制单元译码。控制单元使得相应的信号互相作用，并使处理单元执行这些指令。

1.2. 指令系统设计

在设计处理器时首先要确定 Cpu 具有哪些功能，并针对这些功能采用哪些指令，然后确定指令的格式。为了使设计的 CPU 具有基本的运算功能，指令将设计成以下形式，可以分为如下几类.

- 装载指令： 指令从其他寄存器或存储器装载数据或是立刻赋值。
- 存储指令： 指令存储寄存器的值写到存储器
- 分支指令： 指令使处理器转到其它地址，一些分支指令为条件转移，另外一些为无条件转移
- 移位指令： 这些指令用移位寄存器单元执行移位操作，实现数据传递

1.2.1 指令格式

所有的指令都包含五位操作码。

单字节指令在低 6 位指令中包含两个 3 位寄存器，一个是源操作数寄存器，另一个是目的操作数寄存器。

双字节指令中，第一个字节中包含目标寄存器的地址，第二个字节中包含了指令地址或者操作数。

指令格式如下：

（1） 单字指令

指令的高五位是操作码，低六位是源操作数寄存器和目的操作数寄存器。指令码格式如下

Opcode 操作码						源操作数			目的操作数		
15	14	13	12	11		5	4	3	2	1	0

（2） 双字指令

第一个字中包含目标寄存器的地址，第二个字中包含了指令地址或者操作数。

1.2.2 指令操作码

操作码功能表

操作码	指令	功能
00000	NOP	空操作
00001	LOAD	装载数据到寄存器
00010	STORE	将寄存器的数据存入存储器
00011	MOVE	在寄存器之间传送操作数
00100	LOAD1	将立即数装入寄存器
00101	BRANCHI	转移到由立即数指定的地址
00110	BRANCHGTI	大于是转移到立即数指定的地址
00111	INC	加 1 指令
01000	DEC	减 1 指令
01001	AND	两个寄存器与操作
01010	OR	两个寄存器或操作
01011	XOR	两个寄存器异或操作
01100	NOT	寄存器求反
01101	ADD	两个寄存器加运算
01110	SUB	两个寄存器减运算
01111	ZERO	寄存器清零
10000	BRANCHLTI	小于时转移到由立即数指定的地址
10001	BRANCHLT	小于时转移
10010	BRANCHNEQ	不等于时转移
10011	BRANCHQI	转移到立即数指定的地址
10100	BRANCHGT	大于时转移
10101	BRANCH	无条件转移
10110	BRANCHEQ	等于时转移

10111	BRANCHQI	等于时转移到立即地址
11000	BRANCHELTEI	小于等于时转移到立即地址
11001	BRANCHLTE	小于等于时转移
11010	SHL	向左逻辑移位
11011	SHR	向右逻辑移位
11100	ROTR	循环右移
11101	ROTL	循环左移

1.3. 顶层结构的 VHDL 设计

CPU 原件的 VHDL 描述

CPU_LIB.VHDL 用来说明连接各个原件之间的信号类型。描述了多个用于规定运算器功能、移位寄存操作和用于 CPU 控制的状态的类型。

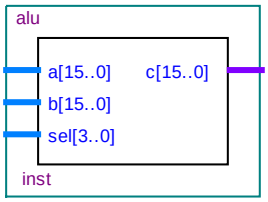
CPU_LIB.VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
package cpu_lib is
type t_shift is (shftpass, shl, shr, rotl, rotr);
subtype t_alu is unsigned(3 downto 0);
constant alupass : unsigned(3 downto 0): "0000";
constant andOp : unsigned(3 downto 0): "0001";
constant orOp : unsigned(3 downto 0): "0010";
constant notOp : unsignde(3 downto 0): "0011";
constant xorOp : unsigned(3 downto 0): "0100";
constant plus : unsignde(3 downto 0): "0101";
constant alusub : unsigned(3 downto 0): "0110";
constant inc : unsignde(3 downto 0): "0111";
constant dec : unsigned(3 downto 0): "1000";
constant zero : unsignde(3 downto 0): "1001";

type t_comp is (eq, neq, gt, gte, lt, lte);
type state is (reset1, reset2, reset3, reset4, reset5, reset6, execute, nop,
load, store, move,load2, load3,load4, store2, store3, store4, move2, move3,
move4, incPc, bgtI3, bgtI4, bgtI5, bgtI6, bgtI7, bgtI8, bgtI9, bgtI10, braI2,
braI3, braI4, braI5, braI6, loadI2, loadI3, loadI4, loadI5, loadI6, inc3,
inc4 );

subtype bit16 is std_logic_vector(15 downto 0);
end cpu_lib;
```

二、 CPU 部件设计

2.1. 算术逻辑单元 ALU

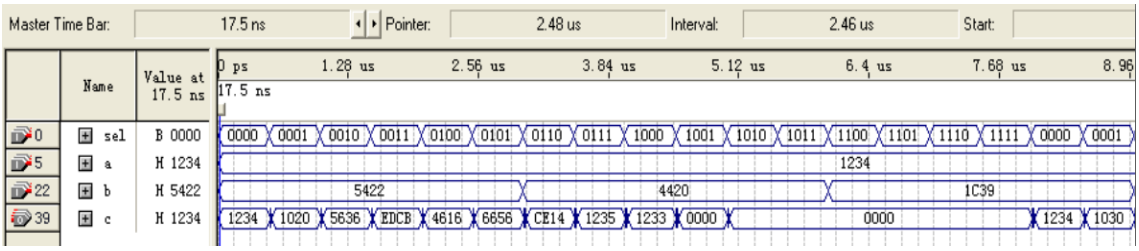
ALU 实体结构如下：



VHDL 代码：

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.cpu_lib.all;

entity alu is
port(a , b: in bit16;
 sel : in t_alu;
 c: out bit16
 );
 end alu;

architecture rtl of alu is
begin
 aluproc: process(a, b, sel)
 begin
     case sel is
         when alupass => c<=a after 1 ns ; --"0000"
         when andOp => c<=a  and b after 1 ns;--"0001"
         when orOp => c<=a or b after 1 ns ;--"0010"
         when xorOp => c<=a xor b after 1 ns;
         when notOp => c<= not a after 1 ns ;
         when plus => c<=a + b after 1 ns;
         when alusub => c<=a - b after 1 ns ;
         when inc => c<=a + "0000000000000001" after 1 ns;
         when dec => c<=a - "0000000000000001" after 1 ns;
         when zero => c<="0000000000000000" after 1 ns;
         when others => c<="0000000000000000" after 1 ns ;
     end case;
 end process;
end rtl;
```

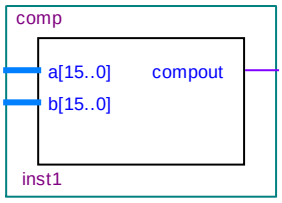
仿真结果：



2.2. 比较器 COMP

比较器的实体名为 COMP。实体 COMP 比较两个值，输出结果为 I 或 O，取决于比较对象的类型和值。比较器结构如图 6-10 所示，比较器的运算类型说明列于表 6-7 中。比较器的运算类型决定于选择输入控制信号 sel 的值。例如，欲比较输入端口 A 和 B 的值是否相等，须先将 EQ 传到端口 sek，这时如果 A 和 B 的值相等，则 COMPOUT 的值为 I；如果不相等，则为 O。

比较器实体结构：



VHDL 代码：

```
library IEEE;
        use  IEEE. std_logic_1164 . all;
        use  IEEE. std_logic_arith . all;
        use  IEEE . std_logic_unsigned. all ;
        use  work. cpu_lib. all;
entity comp is
      port(
      a, b : in bit16;
      sel : in t_comp;
      compout: out std_logic) ;
end comp;

architecture rtl of comp is
begin
compproc:  process (a,  b,  sel)
begin
 case sel is
    when eq => if a = b then compout <= '1' after 1 ns ;
        else compout<='0' after 1 ns ;
        end if ;
    when neq => if a /= b then compout <= '1' after 1 ns;
```

```
            else compout <= '0' after 1 ns ;
            end if;
        when gt=> if a > b then compout <= '1' after 1 ns;
            else compout<= '0' after 1 ns ;
            end if;
        when gte => if a >= b then compout <= '1' after 1 ns;
            else compout <= '0' after 1 ns ;
            end if;
        when lt => if a < b then compout <= '1' after 1 ns ;
            else compout <= '0' after 1 ns ;
            end if;
        when lte => if a <= b then compout <= '1' after 1 ns ;
            else compout <= '0' after 1 ns ;
            end if;
    end case ;
     end process;
    end rtl;
```

比较器 RTL 图

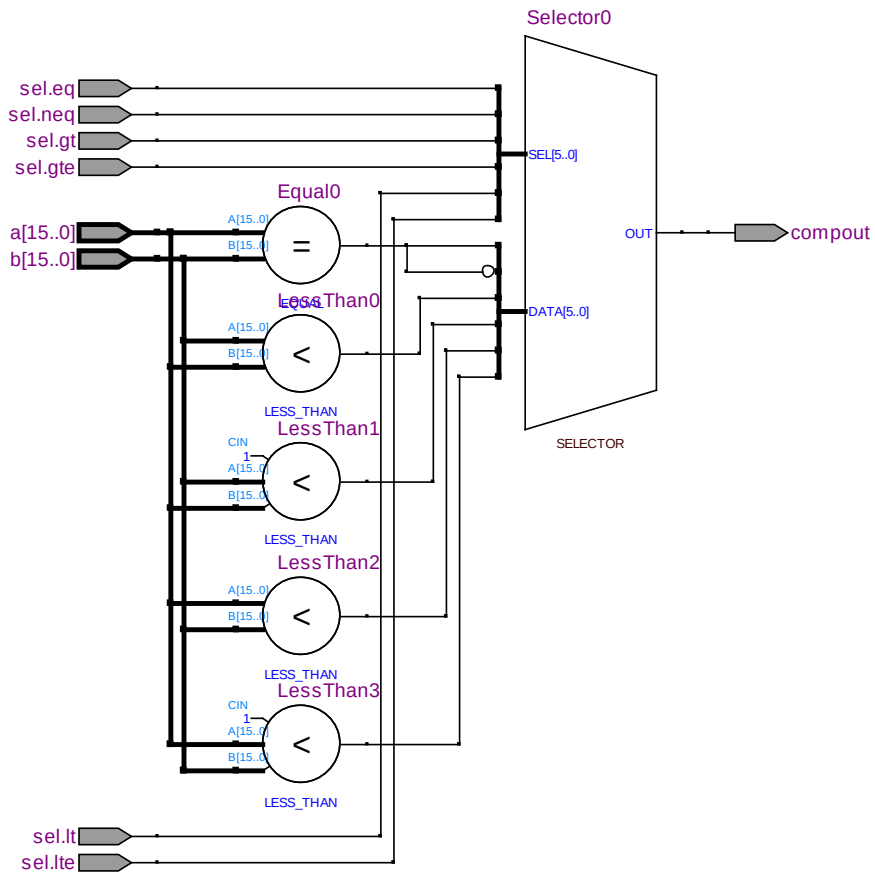


图 2.2.1 比较器 RTL 图

仿真结果：

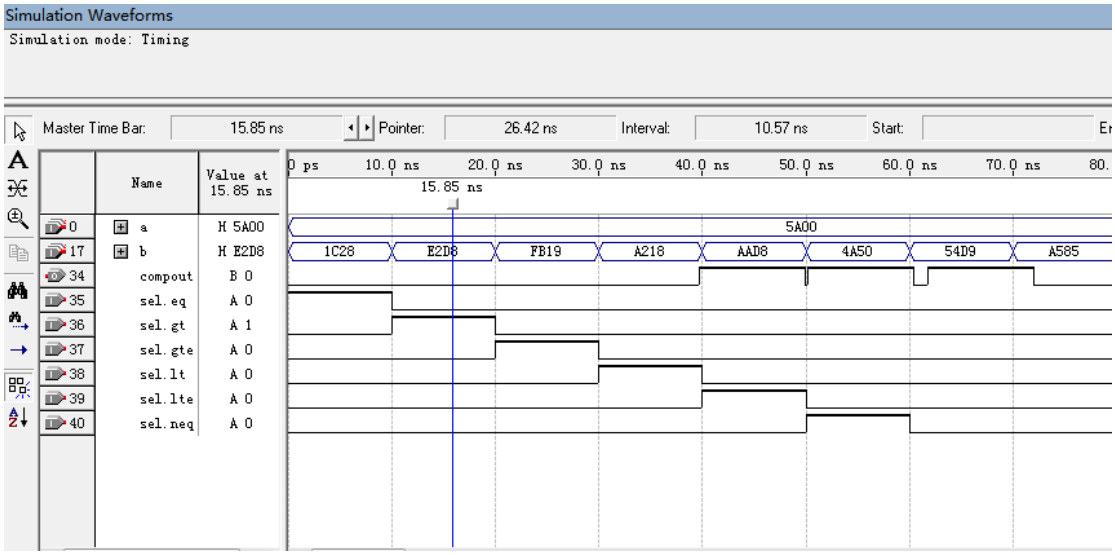


图 2.2.2 比较器时序仿真结果

2.3. 控制器 CONTROL

实体控制器提供必要的信号连线，使得数据流完全通过 CPU，达到预期的功能。结构体包含一个状态机，这个状态机根据当前的状态和输入的信号值，输出更新后的状态。之中，输入信号有： 指令信号 instrReg[15..0] 、比较器输出状态、存储器就绪信号 ready 和 CPU 复位信号 reset。

输出信号时对指令 instrReg[15..0]译码以后，对 CPU 所有组成部件按指令要求进行操作所需的控制信号。

VHDL 程序：

```
library ieee;
use IEEE.std_logic_1164.all;
use work.cpu_lib.all;

entity control is
port(    clock, reset, ready, compout: in std_logic;
         instrReg: in bit16;
         progCntrWr, progCntrRd, addrRegWr, addrRegRd, outRegWr, outRegRd: out std_logic;
         shiftSel: out t_shift;
         aluSel: out t_alu;
         compSel: out t_comp;
         opRegRd, opRegWr, instrWr, regRd, regWr, rw, vma: out std_logic;
         regSel: out t_reg
    );
end control;

architecture rtl of control is
signal current_state, next_state: state;
begin
    nxtstateproc: process( current_state, instrReg, compout, ready )
    begin
        progCntrWr <= '0';
        progCntrRd <= '0';
        addrRegWr <= '0';
        addrRegRd <= '0';
        outRegWr <= '0';
        outRegRd <= '0';
        shiftSel <= shftpass;
        aluSel    <= alupass;
        compSel   <= eq;
        opRegRd   <= '0';
        opRegWr   <= '0';
```

```
        instrWr <= '0';
        regSel <= "000";
        regRd <= '0';
        regWr <= '0' ;
        rw <= '0';
        vma <= '0';
        case current_state is
        when reset1 =>
            aluSel <= zero after 1 ns;
            shiftSel <= shftpass;
            next_state <= reset2;
        when reset2 =>
            aluSel <= zero;
            shiftSel <= shftpass;
            outRegWr <= '1';
            next_state <= reset3;
        when reset3 =>
            outregRd <= '1';
            next_state <= reset4;
        when reset4 =>
            outRegRd <= '1';
            addrRegRd <= '1';
            progCntrWr <='1';
            addrRegWr <= '1';
            next_state <= reset5;
        when reset5 =>
            vma <= '1';
            rw <= '0';
            next_state <= reset6;
        when reset6 =>
            vma <= '1' ;
            rw <= '0';
            if ready = '1' then instrWr <= '1';
                next_state <= execute;
                else next_state <= reset6;
            end if ;
        when execute=>
            case instrReg(15 downto 11) is
                when "00000" => next_state<= incPc; -- nop
                when "00001" => regSel <= instrReg(5 downto 3);
                        regRd <= '1';
                        next_state <= load2;
                when "00010" => regSel <= instrReg(2 downto 0);
                        regRd <= '1';
                        next_state <= store2; --store
                when "00011" => regSel <= instrReg(5 downto 3);
                        regRd <= '1';
                        aluSel <= alupass;
                        shiftSel <= shftpass;
                        next_state <= move2;
                when "00100" =>
                        progCntrRd<= '1';
                        alusel <= inc;
                        shiftsel <= shftpass;
                        next_state <= loadI2;
                when "00101" =>
                        progcntrRd <='1';
                        alusel <= inc;
                        shiftsel <= shftpass;
                        next_state <= braI2;
                when "00110" =>
                        regSel <= instrReg(5 downto 3);
                        regRd <= '1';
                        next_state <= bgtI2; --BranchGTImm
                when "00111" =>
                        regSel <= instrReg(2 downto 0);
                        regRd <= '1';
                        alusel <= inc;
                        shiftsel <= shftpass;
                        next_state <= inc2;
                when others =>
                        next_state <= incPc;
            end case;

        when load2 =>
            regSel <= instrReg(5 downto 3);
            regRd <= '1';
```

```
                addrRegWr <= '1';
                next_state <= load3;
            when load3 =>
                vma <= '1';
                rw <= '0';
                next_state <= load4;
            when load4 =>
                vma <= '1';
                rw <= '0';
                regSel <= instrReg( 2 downto
0);
                regWr <= '1';
                next_state <= incPc;
            when store2 =>
                regsel <= instrReg(2 downto 0);
                regRd <='1';
                addrRegWr <= '1';
                next_state <= store3;
            when store3 =>
                regSel <= instrReg(5 downto
3);
                regRd <= '1';
                next_state <= store4;
            when store4 =>
                regSel <= instrReg(5 downto
3);
                regRd <= '1';
                vma <= '1';
                rw <= '1';
                next_state <= incPc;
            when move2 =>
                regSel <= instrReg(5 downto
3);
                regRd <= '1';
                aluSel <= alupass;
                shiftSel <= shftpass;
                outRegRd <= '1';
                next_state <= move3;
            when move3 =>
                outRegRd <= '1';
                next_state <= move4;
            when move4 =>
                outRegRd <= '1';
                regSel <= instrReg(2 downto
0);
                regWr <= '1';
                next_state <= incPc;
            when loadI2 =>
                progcntrRd <= '1';
                alusel <= inc;
                shiftsel <= shftpass;
                outRegWr <= '1';
                next_state <= loadI3;
            when loadI3 =>
                outregRd <='1';
                next_state <= loadI4;
            when loadI4 =>
                outregRd <='1';
                progcntrWr <= '1';
                addrRegWr <= '1';
                next_state <= loadI5;
            when loadI5 =>
                vma <= '1';
                rw <= '0';
                next_state <= loadI6;
            when loadI6 =>
                vma <= '1';
                rw <= '0';
                if ready = '1' then regSel <=
instrReg(2 downto 0);
                    regWr <= '1';
                    next_state <= incPc;
                else next_state <= loadI6;
                end if ;
            when braI2 => progcntrRd <= '1';
                alusel <= inc;
                shiftsel <= shftpass;
                outregWr <= '1';
                next_state <= braI3;
            when braI3 => outregRd <= '1';
                next_state <= braI4;
            when braI4 => outregRd <= '1';
                progcntrWr <= '1';
                addrRegWr <= '1';
                next_state <= braI5;
            when braI5 =>
                vma <= '1' ;
                rw  <= '0';
```

```
                next_state <= braI6;
            when braI6 =>
                vma <= '1';
                rw <= '0';
                if ready = '1' then
progCntrWr <= '1';
                    next_state <= loadPc;
                else next_state <= braI6;
                end if ;
            when bgtI2 =>
                regsel <= instrReg(5 downto 3);
                regRd <= '1' ;
                opRegWr<= '1';
                next_state <= bgtI3;
            when bgtI3 =>
                opRegRd <= '1';
                regSel <= instrReg(2 downto
0);
                regRd <= '1';
                compsel <= gt;
                next_state <= bgtI4;
            when bgtI4 =>
                opRegRd <= '1' after 1 ns;
                regSel <= instrReg(2 downto
0);
                regRd <= '1';
                compsel <= gt;
                if compout = '1' then next_state
<= bgtI5;
                else next_state <= incPc;
                end if ;
            when bgtI5 =>
                progcntrRd <= '1';
                alusel <= inc ;
                shiftSel <= shftpass;
                next_state <=bgtI6;
            when bgtI6 =>
                progcntrRd <= '1';
                alusel <= inc ;
                shiftSel <= shftpass;
                outregWr <= '1';
                next_state <=bgtI7;
            when bgtI7 => outregRd <= '1';
                next_state <= bgtI8;
            when bgtI8 => outregRd <= '1';
                progcntrWr <= '1';
                addrregWr <= '1';
                next_state <= bgtI9;
            when bgtI9 => vma <= '1';
                rw <= '0';
                next_state <= bgtI10;
            when bgtI10 =>
                vma <= '1';
                rw <= '0';
                if ready = '1' then progcntrWr
<= '1'; next_state <= loadPc;
                else next_state <= bgtI10;
                end if;
            when inc2 => regSel <= instrReg(2
downto 0);
                regRd <= '1';
                alusel <= inc;
                shiftsel <= shftpass;
                outregWr <= '1';
                next_state <= inc3;
            when inc3 => outregRd <= '1';
                 next_state <= inc4;
            when inc4 => outregRd <= '1';
                regsel <= instrReg(2 downto 0);
                regWr <= '1';
                next_state <= incPc;
            when loadPc => progcntrRd <= '1';
                next_state <= loadPc2;
            when loadPc2 => progcntrRd <=
'1';
                addrRegWr <= '1';
                next_state <= loadPc3;
            when loadPc3 =>
                vma <= '1';
                rw <= '0';
                next_state <= loadPc4;
            when loadPc4 => vma<= '1';
                rw <= '0';
                if ready = '1' then instrWr <=
'1'; next_state <= execute;
                else next_state <= loadPc4 ;
                end if;
            when incPc => progcntrRd <= '1';
```

```
            alusel <= inc ;
            shiftsel <= shftpass;
            next_state <= incPc2;
        when incPc2 =>  progcntrRd <= '1' ;
            alusel <= inc ;
            shiftsel <= shftpass;
            outregWr <= '1';
            next_state <= incPc;
        when incPc3 =>  outregRd <= '1';
            next_state <= incPc4;
        when incPc4 => outregRd <= '1';
            progcntrWr <= '1';
            addrregWr <= '1';
            next_state <= incPc5;
        when incPc5 =>
            vma <= '1';
            rw <= '0';
            next_state <= incPc6;
        when incPc6 =>
            vma <= '1';
            rw <= '0';
            if ready = '1' then instrWr <= '1'; next_state <= execute ;
            else next_state <= incPc6;
            end if;
        when others => next_state <= incPc;
    end case ;
end process;

controlffProc: process( clock, reset)
begin
    if reset = '1' then current_state <= reset1 after 1 ns;
    elsif clock'event and clock = '1'
        then current_state <= next_state after 1 ns;
    end if;
end process;

end rtl;
```

控制器实体结构：

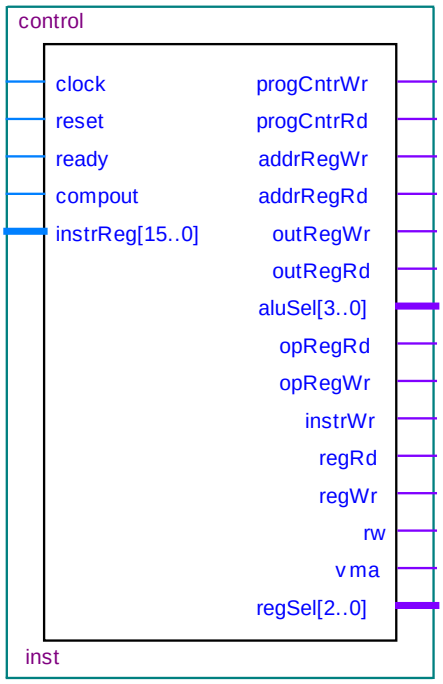


图 2.3.1 控制器实体结构

control 状态机转换表

current_state	**destination State**	转变条件
inc4	incPc	
reset1	reset2	
reset2	reset3	
reset3	reset4	
reset4	reset5	
reset5	reset6	
reset6	reset6	(!ready)
reset6	execute	(ready)
execute	load2	(instrReg[11]).(!instrReg[12]).(!instrReg[13]).(!instrReg[14]).(!instrReg[15])
execute	store2	(!instrReg[11]).(instrReg[12]).(!instrReg[13]).(!instrReg[14]).(!instrReg[15])
execute	move2	(instrReg[11]).(instrReg[12]).(!instrReg[13]).(!instrReg[14]).(!instrReg[15])
execute	incPc	(!instrReg[11]).(!instrReg[12]).(!instrReg[13]) + (!instrReg[11]).(!instrReg[12]).(instrReg[13]).(!instrReg[14]).(instrReg[15]) + (!instrReg[11]).(!instrReg[12]).(instrReg[13]).(instrReg[14]) + (!instrReg[11]).(instrReg[12]).(!instrReg[14]).(instrReg[15]) + (!instrReg[11]).(instrReg[12]).(instrReg[14]) + (instrReg[11]).(!instrReg[14]).(instrReg[15]) + (instrReg[11]).(instrReg[14])
execute	bgtI2	(!instrReg[11]).(instrReg[12]).(instrReg[13]).(!instrReg[14]).(!instrReg[15])
execute	braI2	(instrReg[11]).(!instrReg[12]).(instrReg[13]).(!instrReg[14]).(!instrReg[15])
execute	loadI2	(!instrReg[11]).(!instrReg[12]).(instrReg[13]).(!instrReg[14]).(!instrReg[15])
execute	inc2	(instrReg[11]).(instrReg[12]).(instrReg[13]).(!instrReg[14]).(!instrReg[15])
nop	incPc	
load	incPc	
store	incPc	
move	incPc	
load2	load3	
load3	load4	
load4	incPc	
store2	store3	
store3	store4	
store4	incPc	

move2	move3	
move3	move4	
move4	incPc	
incPc	incPc2	
incPc2	incPc	
incPc3	incPc4	
incPc4	incPc5	
incPc5	incPc6	
incPc6	execute	(ready)
incPc6	incPc6	(!ready)
loadPc	loadPc2	
loadPc2	loadPc3	
loadPc3	loadPc4	
loadPc4	execute	(ready)
loadPc4	loadPc4	(!ready)
bgtI2	bgtI3	
bgtI3	bgtI4	
bgtI4	incPc	(!compout)
bgtI4	bgtI5	(compout)
bgtI5	bgtI6	
bgtI6	bgtI7	
bgtI7	bgtI8	
bgtI8	bgtI9	
bgtI9	bgtI10	
bgtI10	loadPc	(ready)
bgtI10	bgtI10	(!ready)
braI2	braI3	
braI3	braI4	
braI4	braI5	
braI5	braI6	
braI6	loadPc	(ready)
braI6	braI6	(!ready)
loadI2	loadI3	
loadI3	loadI4	
loadI4	loadI5	
loadI5	loadI6	
loadI6	incPc	(ready)
loadI6	loadI6	(!ready)
inc2	inc3	
inc3	inc4	

2. 4. 寄存器与寄存器阵列

寄存器是组成时序电路的最基本单元， 在 CPU 中的寄存器常被用来暂存各种信息，如数据信息、地址信息、指令信息、控制信息等，以及与外部设备交换信息。寄存器组构成 CPU 中的工作寄存器组。

2.4.1. 寄存器 REG

实体 REG 用作地址寄存器和指令寄存器。这些寄存器在时钟上升沿到来时获得输入数据，使输出 Q 得到一个数据。 Reg 有三个端口，端口 A 是数据输入口； Q 是数据输出端口； clk 控制实体 REG 中数据的保存。当上升沿到来时，进程 REGPROC 启动，输入 A 传给输出 Q。

寄存器 Reg 的结构

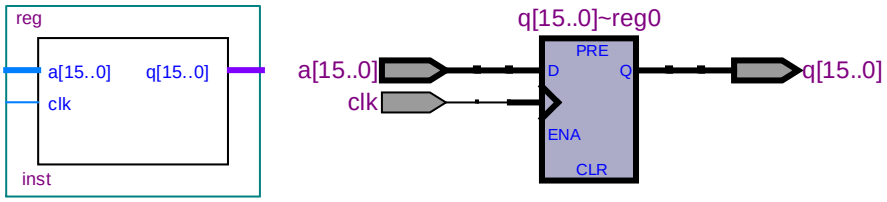


图 2.4.1 寄存器 Reg 的实体结构与 RTL 图

REG 的 VHDL 程序：

```
library IEEE;
USE IEEE.std_logic_1164.all;
use work.cpu_lib.all;

entity reg is
port(
    a : in bit16;
    clk : in std_logic;
    q : out bit16 );
end reg;

architecture rt1 of reg is
begin
regproc : process
begin
wait until clk'event and clk = '1';
q <= a after 1 ns;
end process;
End rt1;
```

2. 4. 2. 寄存器阵列 AYRREGAR

在执行指令时，寄存器中存储指令所处理的立即数，可对寄存器进行读或写操作。寄存器组相当于一个 8 x 16 位的 RAM。当向 REGARRAY 的一个单元写数据时，首先输入 sel 作为单元地址，但 clk 上升沿到来时，输入数据就被写入到该单元中。 当从 REGARRAY 的一个地址单元中读出数据时，首先输入 sel 作为读单元的地址，然后使输出允许控制信号 en

为 1 ，这时数据就会在端口 Q 输出。寄存器阵列有两个独立进程，第一个进程模拟 RAM 存储数据，此进程中包含一个局部变量 RAMDATA ， 它将存储的数据写到实体 REGARRAY，这时如果 clk 上升沿到来，则根据 SEL 选定的单元地址输入新的值。这个进程还将单元地址传给信号 TEMP-DATA ,使数据传到下一个进程。

寄存器阵列 VHDL 程序

```
LIBRARY IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

use work.cpu_lib.all;

entity regarray is
port ( data: in bit16;
       sel : in t_reg;
       en, clk : in std_logic;
       q : out bit16 );
end regarray;

architecture rtl of regarray is
type t_ram is array (0 to 7) of bit16;
signal temp_data : bit16;
begin
process (clk, sel )
    variable ramdata : t_ram;
begin
    if clk'event and clk = '1' then ramdata(conv_integer(sel)) := data;
    end if ;
    temp_data <= ramdata (conv_integer(sel)) after 1 ns ;
end process;

process(en, temp_data)
begin
    if en = '1' then q <= temp_data after 1 ns;
    else q <= "ZZZZZZZZZZZZZZZZ" after 1 ns ;
    end if ;
end process ;
end rtl;
```

寄存器阵列实体结构和 RTL 图

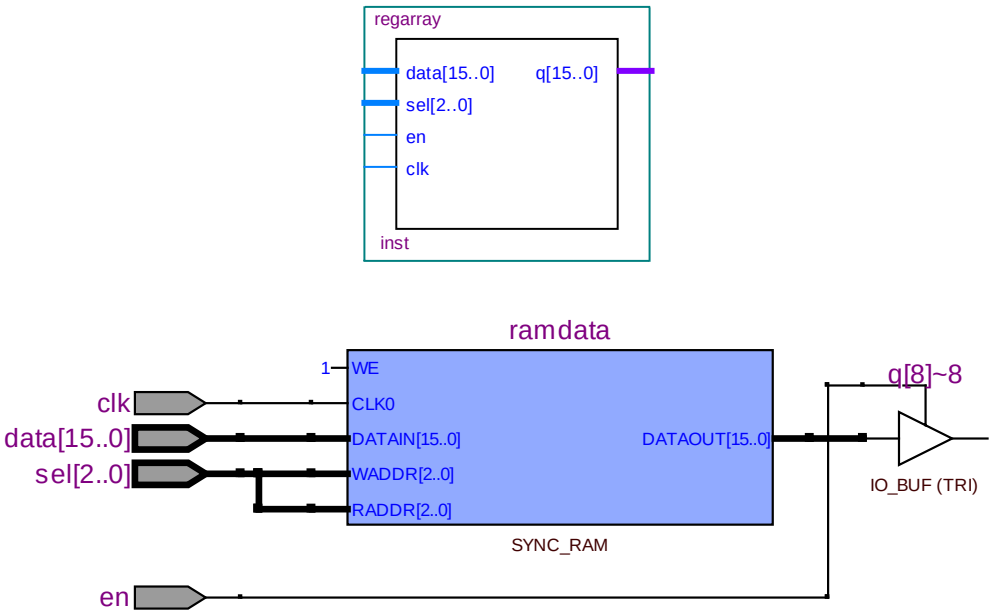
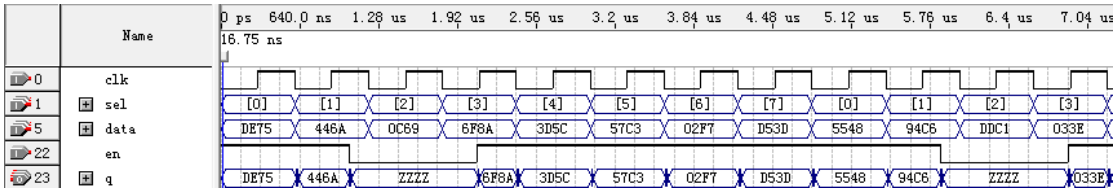


图 2.4.2 寄存器阵列实体结构（上）与 RTL 图（下）

时序仿真



2.5. 移位寄存器

实体移位寄存器 SHIFT 在 CPU 中实现移位和循环操作。 SHIFT 的输入总线为 16 位，输出总线也是 16 位，输入信号 sel 决定执行哪一种转移。 移位的类型有通过、左移和右移、左循环右移和右循环右移。方式 shiftpass 使移位器输入数据直接传给输出，不执行任何移位操作。SHL 和 SHR 决定了移位器是左移还是右移。 ROTL 和 ROTR 决定是左循环还是右循环。

移位寄存器 VHDL 程序 Shift.vhd

```
LIBRARY IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.cup_lib.all;

entity shift is
port( a : in bit16 ;
        sel : in t_shift ;
        y : out bit16;
        );
end shift;

architecture rtl of shift is
```

```
begin
shftproc : process(a, sel)
begin
    case sel is
        when shftpass =>  y <= a after 1 ns;
        when sftl => y <= a(14 downto 0)  & after 1 ns;
        when sfrr => y <= '0' & a(15 downto 1) after 1 ns;
        when rotl => y <= a(14 downto 1) & a(15) after 1 ns;
        when rotr => y <= a(0) & a(15 downto 1) after 1 ns;
        when others => y <= "000000000000000" after 1 ns;
    end case ;
    end process;
end rt1;
```

移位寄存器 RTL 图

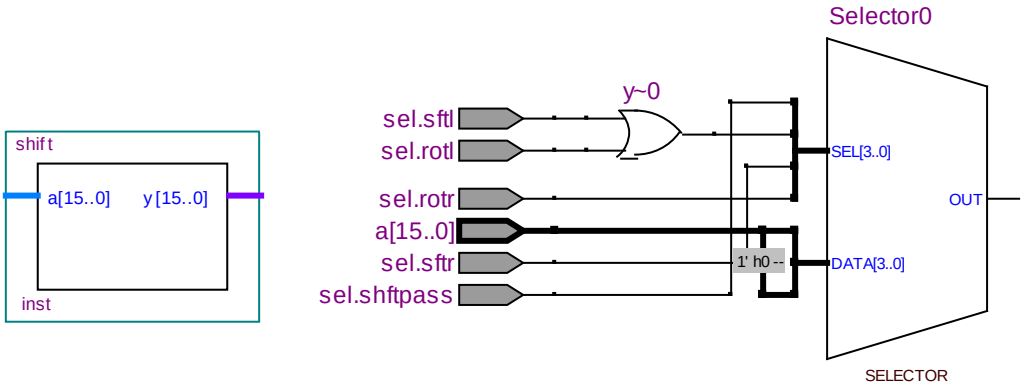


图 2.5 移位寄存器实体结构（左）与 RTL 图（右）

移位寄存器 时序仿真

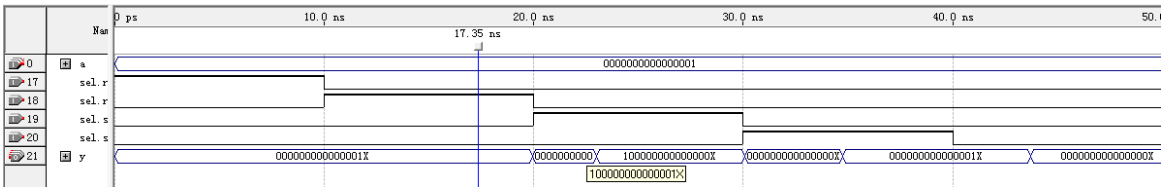
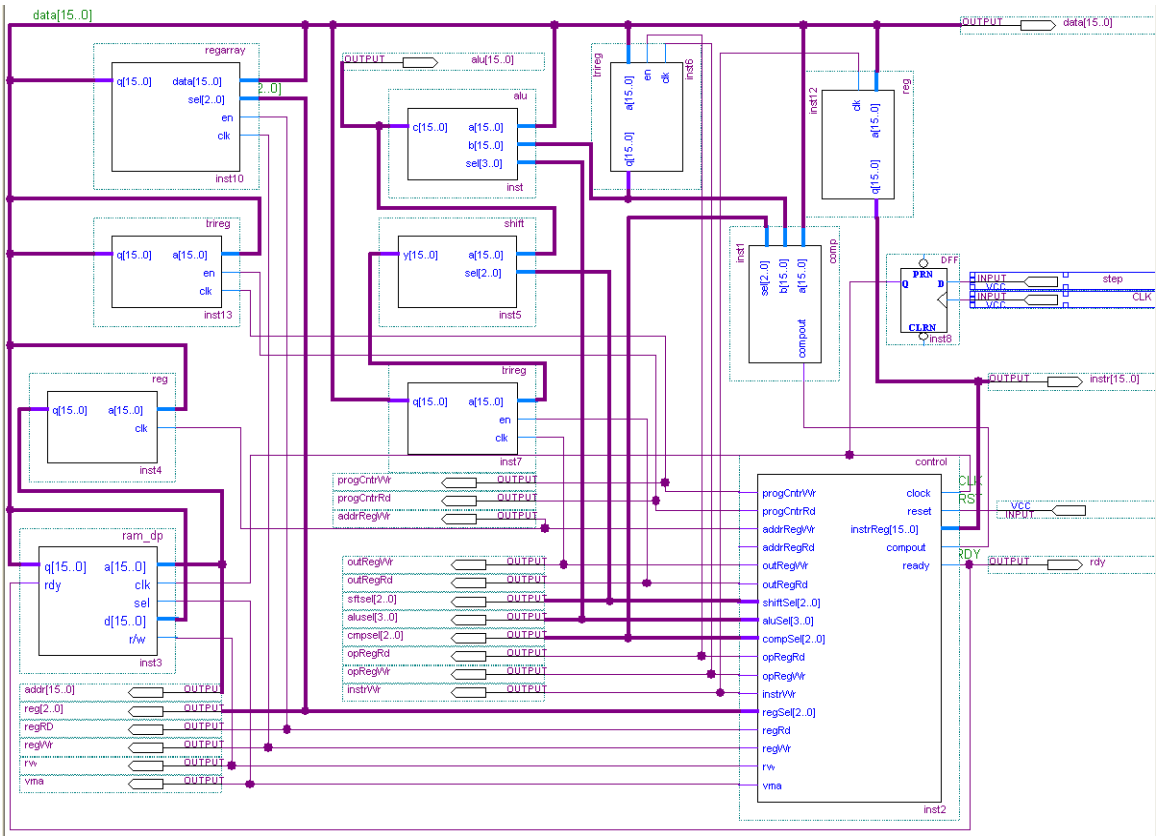


图 移位寄存器 时序仿真

三、 部件的整合

顶层结构的 bdf 文件如下：



四、仿真与下载验证