

6.111演讲11

今天的主题:

握手

“并行”和“顺序”语句

(另一个例子: 计数器)

另外一个例子: 小型ALU

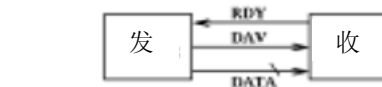
资源用法的简要讨论

第1页

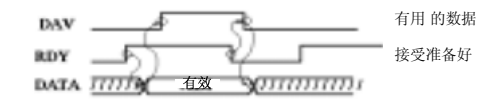
握手

当输入是多线程的时候就需要了。

这是“全握手”。注意上升沿和下降沿的变换在两个方向都是重要的。



这是“全握手”



接收器通过置位RDY显示对接受数据准备好。

发送器设置数据有效, 然后置位DAV

接收器读取数据, 然后清零RDY

发送器通过清零DAV确认

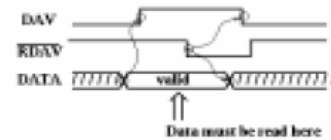
第2页

一个不是很详细论述的握手

这经常用于UART, 它们处理它们不能控制的异步数据流



这是一个“局部的”握手, 用于异步通信



发送器稳定数据, 而且置位DAV。

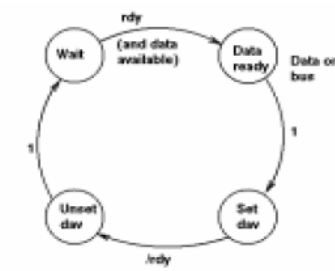
接收器读取数据, 而且清零RDY。

发送器重新确认数据, 清零DAV。

典型的, 发送器在发送新数据前不等待/RDY。这能用来检测“超限”错误。

第3页

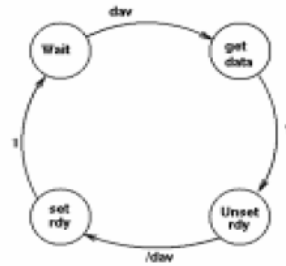
我们可以用简单的有限状态机描述发送和接受动作。这是一个发送端的有限状态机: (全握手)



```
library ieee;
use ieee.std_logic_1164.all;
entity fullsend is
generic (size: integer := 4);
port (rdv, clk : in std_logic;
      datin : in std_logic_vector(size-1 downto 0);
      dav : out std_logic;
      datout : out std_logic_vector(size - 1 downto 0));
end fullsend;
```

第4页

这是接收端的有限状态机:



```

library ieee;
use ieee.std_logic_1164.all;

entity fullrecv is
    generic (size: integer := 4);
    port (dav, rclk : in std_logic;
          datin : in std_logic_vector(size-1 downto 0);
          rdy : out std_logic;
          datout : out std_logic_vector(size - 1 downto 0));
end fullrecv;
  
```

第5页

```

architecture behavioral of fullsend is
    type StateType is (wt, dat, d_av, r_dy);
    attribute enum_encoding of StateType: type is "00 01 11 10";
    signal state : StateType;
begin
    dav <= '1' when (state = d_av) or (state = r_dy) else '0';
    handshake : process(clk)
    begin
        if rising_edge(clk) then
            case state is
                when wt =>
                    if rdy = '1' then
                        state <= dat;
                    else
                        state <= wt;
                    end if;
                when dat =>
                    datout <= datin;
                    state <= d_av;
                when d_av =>
                    state <= r_dy;
                when r_dy =>
                    if rdy = '0' then
                        state <= wt;
                    else
                        state <= r_dy;
                    end if;
            end case;
        end if;
    end process handshake;
end;
  
```

第6页

```

architecture behavioral of fullrecv is
    type StateType is (w_dav, datav, r_rdy, wt_ndav);
    attribute enum_encoding of StateType: type is "00 01 11 10";
    signal state : StateType;
begin
    rdy <= '1' when (state = w_dav) or (state = datav) else '0';
    handshake : process(rclk)
    begin
        if rising_edge(rclk) then
            case state is
                when w_dav =>
                    if dav = '1' then
                        state <= datav;
                    else
                        state <= w_dav;
                    end if;
                when datav =>
                    datout <= datin;
                    state <= r_rdy;
                when r_rdy =>
                    state <= wt_ndav;
                when wt_ndav =>
                    if dav = '0' then
                        state <= w_dav;
                    else
                        state <= wt_ndav;
                    end if;
            end case;
        end if;
    end process handshake;
end;
  
```

第7页

这是'163计数器的仿真器的一种实现方法

- 这是一个寄存器，能保持4位
- 当P=T=1时计数，当P\*T=0时保持
- 当/LD=0时装载数据
- 当/CL=0时清零数据
- 所有的这些都是同步的：只在时钟沿（上升沿）发生菊花链是可能的：RCO链接到下一个有效位的T
- RCO是T\*Q3\*Q2\*Q1\*Q0

这是这一部分的实体语句

```

-- '163 emulator
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

entity ctr is
    generic (size: integer := 4);
    port (n_clr, n_ld, p, t, clk : in std_logic;
          data: in std_logic_vector(size-1 downto 0);
          count: out std_logic_vector(size-1 downto 0);
          rco : out std_logic);
end ctr;
  
```

第8页

```

architecture behavioral of ctr is
  signal cnt_int : std_logic_vector(size - 1 downto 0);
  signal int_cnt : std_logic_vector(size - 1 downto 0); -- 内部计数
  signal all_ones : std_logic_vector(size downto 0);
begin -- behavioral
  all_ones <= (others => '1');
  rco <= '1' when (t & cnt_int) = all_ones else '0';
  count <= cnt_int;

  logical:process(p, t, n_clr, n_ld, cnt_int, data)
  begin
    if n_clr = '0' then
      int_cnt <= (others => '0');
    elsif n_ld = '0' then
      int_cnt <= data;
    elsif p = '0' or t = '0' then
      int_cnt <= cnt_int;
    else
      int_cnt <= cnt_int + 1;
    end if;
  end process logical;
  state_transition:process(clk)
  begin
    if rising_edge(clk) then
      cnt_int <= int_cnt;
    end if;
  end process state_transition;
end behavioral;

```

注意这里的两个进程：  
 一个是组合电路，与该部件的逻辑相关  
 另一个有状态转换动态，与时钟沿相关

DESIGN EQUATIONS (11:32:34)

```

rco =
  t * count_2.Q * count_1.Q * count_0.Q * count_3.Q

count_3.D =
  t * count_2.Q * count_1.Q * count_0.Q * n_clr * n_ld * p *
  /count_3.Q
  + n_clr * n_ld * /p * count_3.Q
  + /count_0.Q * n_clr * n_ld * count_3.Q
  + /count_1.Q * n_clr * n_ld * count_3.Q
  + /count_2.Q * n_clr * n_ld * count_3.Q
  + /t * n_clr * n_ld * count_3.Q
  + n_clr * /n_ld * data_3

count_3.C =
  clk

count_2.D =
  t * /count_2.Q * count_1.Q * count_0.Q * n_clr * n_ld * p
  + count_2.Q * n_clr * n_ld * /p
  + count_2.Q * /count_0.Q * n_clr * n_ld
  + count_2.Q * /count_1.Q * n_clr * n_ld
  + /t * count_2.Q * n_clr * n_ld
  + n_clr * /n_ld * data_2

```

```

count_2.C =
  clk

count_1.D =
  t * /count_1.Q * count_0.Q * n_clr * n_ld * p
  + count_1.Q * n_clr * n_ld * /p
  + count_1.Q * /count_0.Q * n_clr * n_ld
  + /t * count_1.Q * n_clr * n_ld
  + n_clr * /n_ld * data_1

count_1.C =
  clk

count_0.D =
  t * /count_0.Q * n_clr * n_ld * p
  + count_0.Q * n_clr * n_ld * /p
  + /t * count_0.Q * n_clr * n_ld
  + n_clr * /n_ld * data_0

count_0.C =
  clk

```

这些正是你所期望的。  
 注意许多不重要的已经被优化了。

RESOURCE ALLOCATION (11:32:34)

Information: Macrocell Utilization.

Description	Used	Max
Dedicated Inputs	8	8
Clock/Inputs	1	1
Enable/Inputs	0	1
Output Macrocells	5	8
14 / 18 = 77 %		

这是在16v8上的实现。  
 这是我们使用的部件相关资源的数目。

Information: Output Logic Product Term Utilization.

Node#	Output Signal Name	Used	Max
12	count_3	7	8
13	count_2	6	8
14	count_1	5	8
15	count_0	4	8
16	rco	1	8
17	Unused	0	8
18	Unused	0	8
19	Unused	0	8
23 / 64 = 35 %			

```

library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all; -- needed for integer + signal
entity test_tri is
  port (clk, oe, cnt_enb : in std_logic;
        counter : buffer std_logic_vector(3 downto 0);
        data : inout std_logic_vector(3 downto 0));
end test_tri;
architecture foo of test_tri is
  -- signal counter : std_logic_vector(3 downto 0);
begin
  process (oe, counter)
  begin
    if (oe = '1') then data <= counter;
    else
      data <= "ZZZZ"; -- N.B. Z must be UPPERCASE!
    end if;
  end process;
  process (clk)
  begin
    if rising_edge(clk) then
      if (oe = '0') and (cnt_enb = '1') then
        counter <= counter + 1;
      end if;
    end if;
  end process;
end architecture foo;

```

第13页

用三态作输出的仿真

注意当oe为低时，data (3 downto 0) 对应白线（意味着输入）。



第14页

现在我们将考虑一个严格的组合电路：算术逻辑单元（ALU）

采用2个数（本文情况下：每个数2位）

（带进位）

而且可以加，减和向左移位

可以有多种方法实现。考虑加法：

1. a\_int <= '0' & a  
b\_int <= '0' & b  
if c\_in = 0, c <= a\_int + b\_int  
if c\_in = 1, c <= a\_int + b\_int + 1
2. a\_int <= '0' & a & c\_in  
b\_int <= '0' & b & c\_in  
c\_int <= a\_int + b\_int  
c <= c\_int(width downto 1)

这些实现方式有很多不同，而且当我们去做全alu的时候会发现另外一种方式。

```

library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all; -- needed for integer + signal
entity alu is
  port (cin : in std_logic;
        a, b : in std_logic_vector(1 downto 0);
        alu_ctl : in std_logic_vector(1 downto 0);
        c : out std_logic_vector(2 downto 0));
end alu;

```

第15页

这是加法器的一个结构体

```

architecture justright of alu is
  signal a_int, b_int : std_logic_vector(2 downto 0);
  constant add : std_logic_vector(1 downto 0) := "00";
  constant sub : std_logic_vector(1 downto 0) := "01";
  constant shift : std_logic_vector(1 downto 0) := "10";
begin
  a_int <= '0' & a;
  b_int <= '0' & b;
  small_alu : process (a_int, b_int, cin, alu_ctl)
  begin
    case alu_ctl is
      when add => if cin = '0'
                    then c <= a_int + b_int;
                    else c <= a_int + b_int + 1;
                  end if;
      when sub => c <= a_int - b_int;
      when shift => if cin = '0'
                      then c <= a_int + a_int;
                      else c <= a_int + a_int + 1;
                    end if;
      when others => c <= (others => '-');
    end case;
  end process small_alu;
end architecture justright;

```

注意进位位用来确定估值公式：在逻辑上，它是一种多路选择器。'opcode'是另外一种多路选择器：在这种情况下，是3:1的。这就有点开销过大。

第16页

这是加法器的另外一个结构体：做同样的事情

```
architecture justright of alu is
    signal a_int, b_int, c_int : std_logic_vector(3 downto 0);
    constant add : std_logic_vector(1 downto 0) := "00";
    constant sub : std_logic_vector(1 downto 0) := "01";
    constant shift: std_logic_vector(1 downto 0) := "10";
begin
    a_int <= '0' & a & cin;
    b_int <= '0' & b & cin;
    small_alu: process(a_int, b_int, cin, alu_ctl)
    begin
        case alu_ctl is
            when add => c_int <= a_int + b_int;
            when sub => c_int <= a_int - b_int;
            when shift => c_int <= a_int + a_int;
            when others => c_int <= (others => '-');
        end case;
    end process small_alu;
    c <= c_int(3 downto 1);
end architecture justright;
```

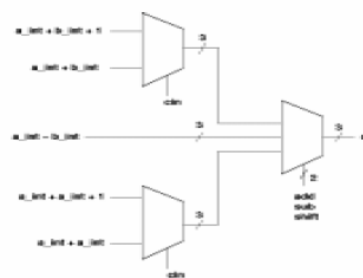
这也许会带来更多的开销。注意通过在两个输入（在二进制小数点之后的第一个位置是有价值的）加Cin。我们至少保留了一些符号开销。在减法的时候取消。但是我们必须舍弃最右边的位。

第17页

```
architecture justright of alu is
    signal a_int, b_int, c_int : std_logic_vector(3 downto 0);
    signal a_1, n_b, upper, lower : std_logic_vector(3 downto 0);
    constant add : std_logic_vector(1 downto 0) := "00";
    constant sub : std_logic_vector(1 downto 0) := "01";
    constant shift: std_logic_vector(1 downto 0) := "10";
begin
    a_int <= '0' & a & cin;
    b_int <= '0' & b & cin;
    a_1 <= '0' & a & '1';
    n_b <= '1' & (not b) & '1';
    upper: process(a_int, a_1, alu_ctl)
    begin
        case alu_ctl is
            when add => upper <= a_int;
            when sub => upper <= a_1;
            when shift => upper <= a_int;
            when others => upper <= (others => '-');
        end case;
    end process upper;
    lower: process(a_int, b_int, n_b, alu_ctl)
    begin
        case alu_ctl is
            when add => lower <= b_int;
            when sub => lower <= n_b;
            when shift => lower <= a_int;
            when others => lower <= (others => '-');
        end case;
    end process lower;
    c_int <= upper + lower;
    c <= c_int(3 downto 1);
end architecture justright;
```

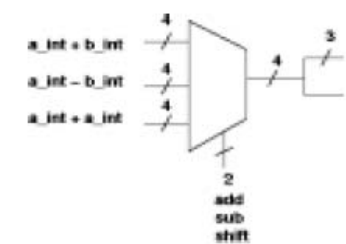
最终的运算是加法，需要我们在早期阶段做更多的工作，比如做b非的二进制补码。我们使用相同的技巧完成加法的两边的进位链接。

第18页



这是第一个实现方案的框图。最后的选择是9: 3多路选择器，在它前面有两个6: 3多路选择器。而且在这之前是同样简单的组合电路，用以产生和。

第19页



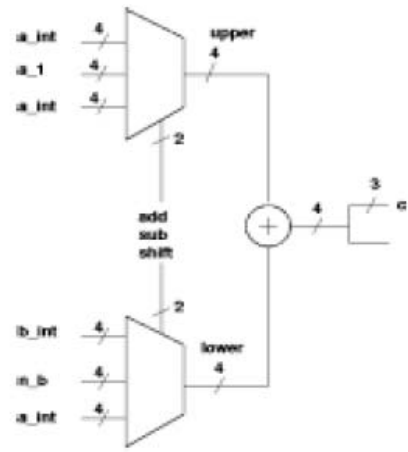
这是两个方案中的第二个方案。

还是有一个多路选择器：这次是12: 4的，但是它只有两个'操纵'位。

在最后的結果中舍弃了一位。

我们也需要构造从左边来到这个数据选择器的信号。

第20页



这是第三种操作方式：有两个12:4的多路选择器，采用一个简单的加法器。

需要采用一些逻辑整理左边的信号。

在输出中舍弃了一位。

第21页

第一种方案：

Information: Macrocell Utilization.

Description	Used	Max
Dedicated Inputs	1	1
Clock/Inputs	4	4
I/O Macrocells	7	64
Buried Macrocells	3	64
PIM Input Connects	12	312

27 / 445 = 6 %

	Required	Max (Available)
CLOCK/LATCH ENABLE signals	0	4
Input REG/LATCH signals	0	69
Input PIN signals	5	5
Input PINs using I/O cells	2	2
Output PIN signals	5	62
<b>Total PIN signals</b>	<b>12</b>	<b>69</b>
<b>Macrocells Used</b>	<b>8</b>	<b>128</b>
<b>Unique Product Terms</b>	<b>43</b>	<b>640</b>

第22页

第二种方案：

Information: Macrocell Utilization.

Description	Used	Max
Dedicated Inputs	1	1
Clock/Inputs	4	4
I/O Macrocells	6	64
Buried Macrocells	2	64
PIM Input Connects	12	312

25 / 445 = 5 %

	Required	Max (Available)
CLOCK/LATCH ENABLE signals	0	4
Input REG/LATCH signals	0	69
Input PIN signals	5	5
Input PINs using I/O cells	2	2
Output PIN signals	4	62
<b>Total PIN signals</b>	<b>11</b>	<b>69</b>
<b>Macrocells Used</b>	<b>6</b>	<b>128</b>
<b>Unique Product Terms</b>	<b>28</b>	<b>640</b>

第23页

第三种方案：

Information: Macrocell Utilization.

Description	Used	Max
Dedicated Inputs	1	1
Clock/Inputs	4	4
I/O Macrocells	5	64
Buried Macrocells	1	64
PIM Input Connects	8	312

19 / 445 = 4 %

	Required	Max (Available)
CLOCK/LATCH ENABLE signals	0	4
Input REG/LATCH signals	0	69
Input PIN signals	5	5
Input PINs using I/O cells	2	2
Output PIN signals	3	62
<b>Total PIN signals</b>	<b>10</b>	<b>69</b>
<b>Macrocells Used</b>	<b>4</b>	<b>128</b>
<b>Unique Product Terms</b>	<b>28</b>	<b>640</b>

第24页