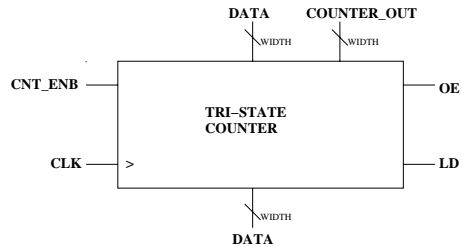


11.1 Tri-State to Multiplex IO Pins

Monday, March 5, 2001

```
-- Use tri-state logic to multiplex IO pins.
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all; -- needed for integer + signal
entity ldcnt is
  generic (width : integer := 3);
  port(clk, ld, oe, cnt_enb : in std_logic;
       -- Use counter_out_out only for simulation.
       counter_out : out std_logic_vector(width - 1 downto 0);
       data : inout std_logic_vector(width - 1 downto 0));
end ldcnt;
```



11.2 Tri-State Architecture

```
-- purpose: count with an output enable
architecture archldcnt of ldcnt is
  signal counter : std_logic_vector(width - 1 downto 0);
begin
  counter_out <= counter; -- only use counter_out for simulation
  cnt: process(clk)
  begin
    if rising_edge(clk) then
      if ld = '1' and oe = '0' then
        counter <= data;
      elsif cnt_enb = '1' then
        counter <= counter + 1;
      end if;
    end if; -- rising_edge(clk)
  end process cnt;
  outen: process(oe, counter)
  begin
    if oe = '1' then data <= counter;
    else
      data <= (others => 'Z'); -- N.B. Z must be UPPERCASE!
    end if;
  end process outen;
end architecture archldcnt;
```

11.3 Simulation of Tri-State Counter

□ Note that data(2 downto 0) are white (meaning an input) when oe is low.

□ Also that counter doesn't ld unless oe is low.



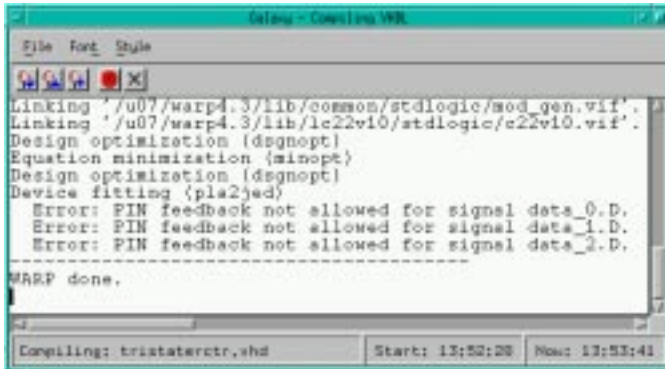
11.4 Without Counter Out

□ One has to poke around for the jed nodes.



11.5 Without Counter Out for a 22V10

- The tri-state counter works without counter_out, but you get this error if you try to use a 22v10. Why is this?



11.6 A Clue From the Working 374i

- Only 3 I/O macrocells are used.

Extract of pinout for 374i

```

20 : clk
23 : cnt_enb
24 : data_1
27 : data_2
28 : data_0
41 : oe
62 : ld
    
```

Information: Macrocell Utilization.

| Description | Used | Max |
|--------------------|------|------|
| Dedicated Inputs | 1 | 1 |
| Clock/Inputs | 3 | 4 |
| I/O Macrocells | 3 | 64 |
| Buried Macrocells | 0 | 64 |
| PIM Input Connects | 9 | 312 |
| 16 / 445 | | = 3% |

11.7 A Clue From the working 22v10

- Separate pins are used for data and counter_out.
- Because PIN feedback from flip-flop outputs is not allowed.

c22v10

```

clk =| 1|          |24|* not used
cnt_enb =| 2|        |23|= counter_out_0
oe =| 3|           |22|= data_0
ld =| 4|           |21|= data_2
not used *| 5|        |20|* not used
not used *| 6|        |19|* not used
not used *| 7|        |18|* not used
not used *| 8|        |17|* not used
not used *| 9|        |16|= data_1
not used *|10|        |15|= counter_out_2
not used *|11|        |14|= counter_out_1
not used *|12|        |13|* not used
    
```

11.8 A Simple Adder

- An adder produces a carry out. Normally we think of this as a longer signal (vector).
 - We use the concatenation operator, &, to create two internal signals which are one bit longer and then use them to create the actual output signal.

```

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

architecture toomany of twoadd is
    signal a_int, b_int : std_logic_vector(2 downto 0);
begin
    a_int <= '0' & a;
    b_int <= '0' & b;
    c <= a_int + b_int when cin = '0' else a_int + b_int + 1;
end architecture toomany;
    
```

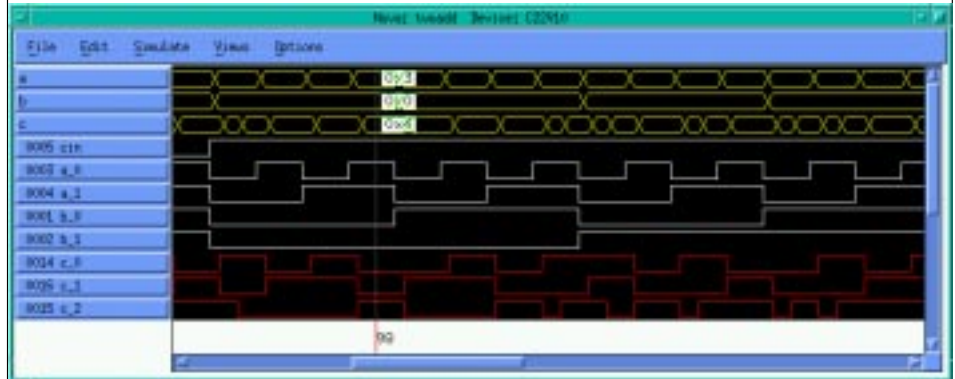
11.9 Testing the Adder by Simulation

- First, we will look at the part where cin is zero.



11.10 More Testing by Simulation

- Next, we will look at the part where cin is one.



11.11 What Did it Take?

- Not really for reading, BUT notice the rather large number of product terms and the cascading for c₂.

```

c_2 = cin * a_1 * a_0 * /b_1 * /b_0 * /MODULE_2_g1_a0_g0_u0_ga_g1_u0
+ cin * /a_1 * a_0 * b_1 * /b_0 * /MODULE_2_g1_a0_g0_u0_ga_g1_u0
+ cin * a_1 * /a_0 * /b_1 * b_0 * /MODULE_2_g1_a0_g0_u0_ga_g1_u0
+ cin * /a_1 * /a_0 * b_1 * b_0 * /MODULE_2_g1_a0_g0_u0_ga_g1_u0
+ /a_0 * /b_0 * MODULE_2_g1_a0_g0_u0_ga_g1_u0
+ a_0 * b_0 * MODULE_2_g1_a0_g0_u0_ga_g1_u0
+ /a_1 * /b_1 * MODULE_2_g1_a0_g0_u0_ga_g1_u0
+ a_1 * b_1 * MODULE_2_g1_a0_g0_u0_ga_g1_u0
+ /cin * MODULE_2_g1_a0_g0_u0_ga_g1_u0
c_1 = a_1 * /a_0 * /b_1 * /b_0
+ /cin * a_1 * /b_1 * /b_0
+ /a_1 * /a_0 * b_1 * /b_0
+ /cin * /a_1 * b_1 * /b_0
+ /a_1 * a_0 * /b_1 * b_0
+ cin * /a_1 * /b_1 * b_0
+ a_1 * a_0 * b_1 * b_0
+ cin * a_1 * b_1 * b_0
+ /cin * a_1 * /a_0 * /b_1
+ cin * /a_1 * a_0 * /b_1
+ /cin * /a_1 * /a_0 * b_1
+ cin * a_1 * a_0 * b_1
c_0 = cin * /a_0 * /b_0
+ /cin * a_0 * /b_0
+ /cin * /a_0 * b_0
+ cin * a_0 * b_0
MODULE_2_g1_a0_g0_u0_ga_g1_u0 = a_0 * b_1 * b_0
+ a_1 * a_0 * b_0
+ a_1 * b_1
    
```

11.12 Another Way to Look at Resources

```

c22v10
b_0 = | 1 | |24|* not used
b_1 = | 2 | |23|= (MODULE_2_g1..
a_0 = | 3 | |22|* not used
a_1 = | 4 | |21|* not used
cin = | 5 | |20|* not used
not used * | 6 | |19|* not used
not used * | 7 | |18|* not used
not used * | 8 | |17|* not used
not used * | 9 | |16|= c_1
not used * |10| |15|= c_2
not used * |11| |14|= c_0
not used * |12| |13|* not used
    
```

Information: Output Logic Product Term Utilization.

| Node# | Output Signal Name | Used | Max |
|-------|--------------------|------|-----|
| 14 | c_0 | 4 | 8 |
| 15 | c_2 | 9 | 10 |
| 16 | c_1 | 12 | 12 |
| 17 | Unused | 0 | 14 |
| 18 | Unused | 0 | 16 |
| 23 | MODULE_2_g1_... | 3 | 8 |
| 25 | Unused | 0 | 1 |

28 / 121 = 23 %

11.13 A Simpler Adder

- We create two internal signals that are two bits longer than the input signals by concatenating a zero on the left and cin on the right.
- We then create the output signal by using the left four bits of the sum of these two internal signals.

```

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

architecture justright of oneadd is
  signal a_int, b_int, c_int : std_logic_vector(3 downto 0);
begin
  a_int <= '0' & a & cin;
  b_int <= '0' & b & cin;
  c_int <= a_int + b_int;
  c <= c_int(3 downto 1);
end architecture justright;

```

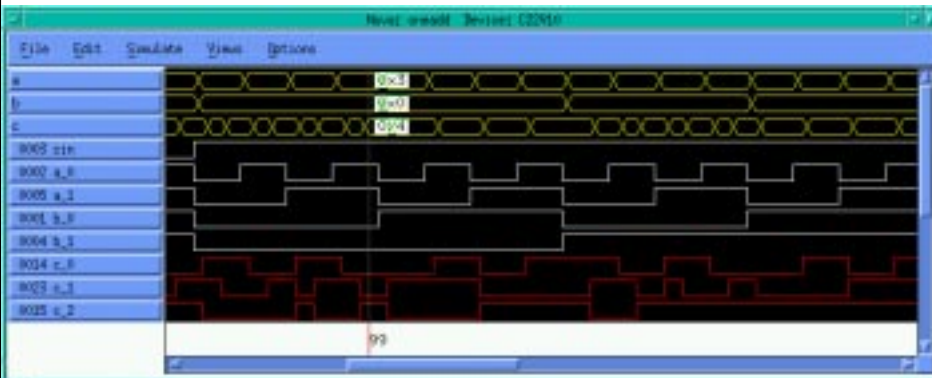
11.14 Testing the Simpler Adder

- We had better test the simpler adder to make sure it is right!
- First, we will look at the part where cin is zero.



11.15 More Testing of the Simpler Adder

- Next, we will look at the part where cin is one.



- We are lucky! This is right.
- But, then, isn't this the way you would solve this problem with hardware?

11.16 What Did the Simpler Adder Take?

- Again, not really for reading, BUT a lot fewer product terms.

```

c_2 = b_1 * MODULE_1_g1_a0_g0_u0_ga_g1_u0
      + a_1 * MODULE_1_g1_a0_g0_u0_ga_g1_u0
      + a_1 * b_1

c_1 = a_1 * /b_1 * /MODULE_1_g1_a0_g0_u0_ga_g1_u0
      + /a_1 * b_1 * /MODULE_1_g1_a0_g0_u0_ga_g1_u0
      + /a_1 * /b_1 * MODULE_1_g1_a0_g0_u0_ga_g1_u0
      + a_1 * b_1 * MODULE_1_g1_a0_g0_u0_ga_g1_u0

c_0 = cin * /a_0 * /b_0
      + /cin * a_0 * /b_0
      + /cin * /a_0 * b_0
      + cin * a_0 * b_0

MODULE_1_g1_a0_g0_u0_ga_g1_u0 = a_0 * b_0
                                + cin * b_0
                                + cin * a_0

```

11.17 A More Realistic Example

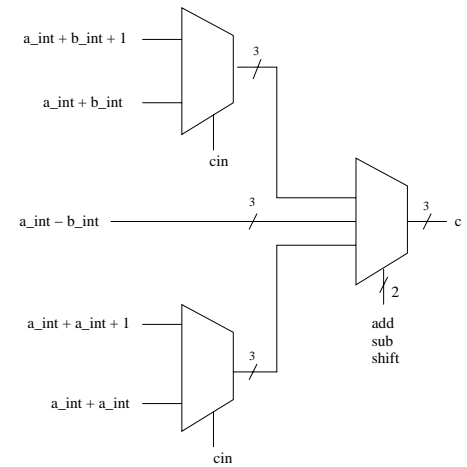
□ Switch to emacs.

□ See the files in the directory

`/mit/6.111/vhdl/warp/IMPLIED.ADDERS/`

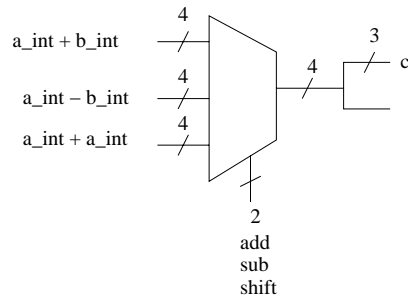
11.18 Many

MANY



11.19 Better

BETTER



11.20 Best

BEST

