Topics for today:

Handshaking
'Concurrent' and 'Sequential' statements
(Another example: a counter)
Yet another example: a small ALU
Brief discussion of resource usage

Handshaking
Required when multiple lines of input are involved
This is a 'full handshake' Note that both positive going and negative going transitions are important in both directions

A Less Elaborate handshake
This is often used in things like UARTs which must deal with asynchronous data streams that they do not control

Handshaking
Required when multiple lines of input are involved
This is a 'full handshake' Note that both positive going and negative going transitions are important in both directions

We should be able to describe the sending and receiving agents as simple finite state machines. Here is the FSM at the Sending end: (Full handshake)

sender stabilize data and sets DAV
receiver reads data and clears RDAV
sender de-asserts data and clears DAV
Typically, sender does not wait for RDAV before setting new data. This can be used for detecting 'overrun' errors.

library ieee;
use ieee.std_logic_1164.all;

entity fullsend is
  generic (size: integer := 4);
  port (rdy, 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;
And here is the FSM for the receiving end:

```
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;
          if dav = '0' then
            state <= w_dav;
          else
            state <= r_dav;
          end if;
        end case;
      end if;
    end process handshake;
  end;
end;
```

Here is an alternative way of writing an emulator for the '163 counter

This is a register which can hold 4 bits
Counts when \(P=T=1\), holds when \(P\times T=0\)
Loads data when /LD = 0
Clears data when /CL = 0
All of these are synchronous: occur only on clock edges (positive edges)
Daisy-chaining is possible: RCO connects to \(T\) of next most significant ctr
RCO is \(T \times Q_3 \times Q_2 \times Q_1 \times Q_0\)

Here is an entity statement for this part

```
-- '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;
```
architecture behavioral of cnt is
signal cnt_int : std_logic_vector(size - 1 downto 0); -- internal count
signal int_cnt : std_logic_vector(size - 1 downto 0); -- internal count
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 p = '0' or t = '0' then
      int_cnt <= cnt_int;
    else
      int_cnt <= int_cnt + 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;

Note two processes here:
One has the combinatorics
associated with the logic in the part.
The other has the state transition
dynamics associated with the clock
dynamic.

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

RESOURCE ALLOCATION (11:32:34)

Information: Macrocell Utilization.

<table>
<thead>
<tr>
<th>Description</th>
<th>Used</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dedicated Inputs</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Clock/Inputs</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Enable/Inputs</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Output Macrocells</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

14 / 18 = 77 %

Information: Output Logic Product Term Utilization.

<table>
<thead>
<tr>
<th>Node#</th>
<th>Output Signal Name</th>
<th>Used</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>count_3</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>13</td>
<td>count_2</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>14</td>
<td>count_1</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>15</td>
<td>count_0</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>rco</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>17</td>
<td>Unused</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>18</td>
<td>Unused</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>19</td>
<td>Unused</td>
<td>0</td>
<td>8</td>
</tr>
</tbody>
</table>

23 / 64 = 35 %

This was implemented on a 16V8

Here are some numbers relating to how much of the resources of that part we used.

These are just about what you would have expected.
Note a lot of fluff has been optimized away.
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
begin
-- signal counter : std_logic_vector(3 downto 0);
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;

Now we are going to consider a strictly combinatoric circuit: an Arithmetic Logic Unit (ALU).

It takes 2 numbers (quite narrow in this case: 2 bits each)
(Plus a carry-in bit)
And can add, subtract and shift left
This can be done in more than one way. Consider addition:

1. \[ \begin{align*}
    a_{int} &= '0' \& a \\
    b_{int} &= '0' \& b \\
    \text{if } c_{in} = 0, c &= a_{int} + b_{int} \\
    \text{if } c_{in} = 1, c &= a_{int} + b_{int} + 1
\end{align*} \]

2. \[ \begin{align*}
    a_{int} &= '0' \& a \& c_{in} \\
    b_{int} &= '0' \& b \& c_{in} \\
    c_{int} &= a_{int} + b_{int} \\
    c &= c_{int}(\text{width downto } 1)
\end{align*} \]

These have differences in the way they are implemented, and when we get to actual implementation of the full alu we will find yet another one.

Simulation of Tri-State as an Output

Note that data(3 downto 0) are white (meaning an input) when oe is low.

Here is one architecture for the adder

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;

Note the carry bit is
used to determine
which expression to
evaluate: in logic it is a
kind of multiplexor.
The ‘opcode’ is another
multiplexor: in this
case a 3:1
These have overhead.
Here is the second architecture: does the same thing...

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;

This may or may not have more overhead. Note that by adding Cin to both inputs (the first place after the binary point is of value ) we save at least some notational overhead. It cancels on subtract. But we have to discard the rightmost bit at the end

Here is a schematic of the way the first of these schemes is implemented. The final selection is a 9:3 MUX, while there are two 6:3 MUXes ahead of it. And ahead of that are some simple combinatoric circuits to generate the sums.

Here is the second of the two schemes
There still is a MUX: this time of 12:4, but it still has only two 'steering' bits.

We discard a bit in the final result
And we also have to construct the signals that come to this MUX from the left.

Here our final operation is just an addition, so we do more work in the earlier stages, such as doing the two's complement for the negative of b. We use the same trick for concatenating the carry in bit to both halves of the addition.
This is the second of the three schemes

Information: Macrocell Utilization.

<table>
<thead>
<tr>
<th>Description</th>
<th>Used</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dedicated Inputs</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Clock/Inputs</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>I/O Macrocells</td>
<td>6</td>
<td>64</td>
</tr>
<tr>
<td>Buried Macrocells</td>
<td>2</td>
<td>64</td>
</tr>
<tr>
<td>PIM Input Connects</td>
<td>12</td>
<td>312</td>
</tr>
</tbody>
</table>

\[
25 / 445 = 5 \%
\]

 Required Max (Available)

<table>
<thead>
<tr>
<th>Description</th>
<th>Required</th>
<th>Max (Available)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOCK/LATCH ENABLE signals</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Input REG/LATCH signals</td>
<td>0</td>
<td>69</td>
</tr>
<tr>
<td>Input PIN signals</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Input PINs using I/O cells</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Output PIN signals</td>
<td>4</td>
<td>62</td>
</tr>
</tbody>
</table>

Total PIN signals            11  69
Macrocells Used              6  128
Unique Product Terms         28  640

This is the third of the three schemes

Information: Macrocell Utilization.

<table>
<thead>
<tr>
<th>Description</th>
<th>Used</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dedicated Inputs</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Clock/Inputs</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>I/O Macrocells</td>
<td>5</td>
<td>64</td>
</tr>
<tr>
<td>Buried Macrocells</td>
<td>1</td>
<td>64</td>
</tr>
<tr>
<td>PIM Input Connects</td>
<td>8</td>
<td>312</td>
</tr>
</tbody>
</table>

\[
19 / 445 = 4 \%
\]

 Required Max (Available)

<table>
<thead>
<tr>
<th>Description</th>
<th>Required</th>
<th>Max (Available)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOCK/LATCH ENABLE signals</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Input REG/LATCH signals</td>
<td>0</td>
<td>69</td>
</tr>
<tr>
<td>Input PIN signals</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Input PINs using I/O cells</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Output PIN signals</td>
<td>3</td>
<td>62</td>
</tr>
</tbody>
</table>

Total PIN signals            10  69
Macrocells Used              4  128
Unique Product Terms         28  640