Combinational Circuit Design: Practice

Outline
1. Derivation of efficient HDL description
2. Operator sharing
3. Functionality sharing
4. Layout-related circuits
5. General circuits

1. Derivation of efficient HDL description
   • Think “H”, not “L”, of HDL
   • Right way:
     – Research to find an efficient design (“domain knowledge”)
     – Develop VHDL code that accurately describes the design
   • Wrong way:
     – Write a C program and covert it to HDL

Sharing
   • Circuit complexity of VHDL operators varies
   • Arith operators
     – Large implementation
     – Limited optimization by synthesis software
   • “Optimization” can be achieved by “sharing” in RT level coding
     – Operator sharing
     – Functionality sharing

An example 0.55 um standard-cell CMOS implementation

<table>
<thead>
<tr>
<th>width</th>
<th>madd</th>
<th>xor</th>
<th>add</th>
<th>mul</th>
<th>div</th>
<th>shr</th>
<th>emux</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>area (gate count)</td>
<td>delay (ns)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3.5</td>
<td>8</td>
<td>0.5</td>
<td>4.2</td>
<td>2.2</td>
<td>0.7</td>
<td>2.8</td>
</tr>
<tr>
<td>16</td>
<td>101</td>
<td>91</td>
<td>6.5</td>
<td>5.5</td>
<td>4.3</td>
<td>2.3</td>
<td>3.5</td>
</tr>
<tr>
<td>32</td>
<td>203</td>
<td>211</td>
<td>115</td>
<td>131</td>
<td>405</td>
<td>171</td>
<td>437</td>
</tr>
<tr>
<td>64</td>
<td>406</td>
<td>412</td>
<td>408</td>
<td>408</td>
<td>408</td>
<td>408</td>
<td>408</td>
</tr>
</tbody>
</table>

2. Operator sharing
   – “value expressions” in priority network and multiplexing network are mutually exclusively:
   – Only one result is routed to output
   – Conditional sig assignment (if statement)
     ```vhdl```
     ```
     sig_name <= value_expr_1 when boolean_expr_1 else
                 value_expr_2 when boolean_expr_2 else
                 value_expr_3 when boolean_expr_3 else
                        . . .
                 value_expr_n;
     ```
– Selected sig assignment (case statement)

with select_expression select
  sig_name <= value_expr_1 when choice_1,
  value_expr_2 when choice_2,
  value_expr_3 when choice_3,
  . . .
  value_expr_n when choice_n;

Example 1

• Original code:
r <= a+b when boolean_exp else
  a+c;

• Revised code:
  src0 <= b when boolean_exp else
  c;
  r <= a + src0;

Example 2

• Original code:
  process(a,b,c,d,...)
  begin
    if boolean_exp_1 then
      r <= a+b;
    elsif boolean_exp_2 then
      r <= a+c;
    else
      r <= d+1;
    end if
  end process;

• Revised code:
  process(a,b,c,d,...)
  begin
    if boolean_exp_1 then
      src0 <= a;
      src1 <= b;
    elsif boolean_exp_2 then
      src0 <= a;
      src1 <= c;
    else
      src0 <= d;
      src1 <= "00000001";
    end if
  end process;
  r <= src0 + src1;
Example 3

• Original code:
  
  ```vhdl
  with sel select
  r <= a+b when "00",
      a+c when "01",
      d+1 when others;
  ```

• Revised code:
  
  ```vhdl
  with sel_exp select
  src0 <= a when "00"|"01",
           b when "00",
           c when "01",
           "00000001" when others;
  r <= src0 + src1;
  ```

Example 4

• Original code:
  
  ```vhdl
  process(a,b,c,d,...)
  begin
    if boolean_exp then
      x <= a + b;
      y <= (others=>'0');
    else
      x <= (others=>'1');
      y <= c + d;
    end if;
  end process;
  ```

• Revised code:
  
  ```vhdl
  begin
    if boolean_exp then
      src0 <= a;
      src1 <= b;
      y <= (others=>'0');
    else
      src0 <= c;
      src1 <= d;
      y <= sum;
    end if;
  end process;
  ```

• Area: 1 adder, 4 mux

• Is the sharing worthwhile?
  - 1 adder vs 2 mux
  - It depends . . .
Summary

- Sharing is done by additional routing circuit
- Merit of sharing depends on the complexity of the operator and the routing circuit
- Ideally, synthesis software should do this

3. Functionality sharing

- A large circuit involves lots of functions
- Several functions may be related and have common characteristics
- Several functions can share the same circuit.
- Done in an "ad hoc" basis, based on the understanding and insight of the designer (i.e., "domain knowledge")
- Difficult for software since it does not know the "meaning" of functions

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity addsub is
  port (a, b: in std_logic_vector(7 downto 0);
        ctrl: in std_logic;
        r: out std_logic_vector(7 downto 0));
end addsub;
architecture direct_arch of addsub is
begin
  signal src0, src1, sum: signed(7 downto 0);
  architecture direct_arch of addsub is
  begin
    src0 <= signed(a);
    src1 <= signed(b);
    sum <= src0 + src1 when ctrl='0' else
           src0 - src1;
    r <= std_logic_vector(sum);
  end direct_arch;
end addsub;
```

- Observation: \( a - b \) can be done by \( a + b' + 1 \)

```vhdl
architecture shared_arch of addsub is
begin
  src0 <= signed(a);
  src1 <= signed(b) when ctrl='0' else
           signed(not b);
  cin <= "0" when ctrl='0' else
         "1";
  sum <= src0 + src1 + cin;
  r <= std_logic_vector(sum);
end shared_arch;
```

• Manual injection of carry-in:
  • Append an additional bit in right (LSB): 

```vhdl
architecture manual_carry_arch of addsub is
begin
  b_tmp <= b when ctrl='0' else
            '0';
  cin <= '0' when ctrl='0' else
         '1';
  src1 <= signed(b_tmp & cin);
  sum <= src0 + src1;
  r <= std_logic_vector(sum(8 downto 1));
end manual_carry_arch;
```

- Carries are added with 
- Manual injection of carry-in:
  • Append an additional bit in right (LSB):

```vhdl
architecture manual_carry_arch of addsub is
begin
  b_tmp <= b when ctrl='0' else
            '0';
  cin <= '0' when ctrl='0' else
         '1';
  src1 <= signed(b_tmp & cin);
  sum <= src0 + src1;
  r <= std_logic_vector(sum(8 downto 1));
end manual_carry_arch;
```
e.g., sign-unsigned comparator

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity comp2mode is
  port(a,b: in std_logic_vector(7 downto 0);
    mode: in std_logic);
  agtb: out std_logic);
end comp2mode;

architecture direct_arch of comp2mode is
  signal agtb_signed, agtb_unsigned: std_logic;
begin
  agtb_signed <= '1' when signed(a) > signed(b) else '0';
  agtb_unsigned <= '1' when unsigned(a) > unsigned(b) else '0';
  agtb <= agtb_unsigned when (mode='0') else agtb_signed;
end direct_arch;
```

- Observation:
  - Unsigned: normal comparator
  - Signed:
  - Different sign bit: positive number is larger
  - Same sign: compare remaining 3 LSBs
    This works for negative number, too!
    E.g., 1111 (-1), 1100 (-4), 1001 (-7)
    ```
    111 > 100 > 001
    ```
  - The comparison of 3 LSBs can be shared

---

e.g., Full comparator

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity comp3 is
  port(a,b: in std_logic_vector(15 downto 0);
    agtb, altb, aeqb: out std_logic);
end comp3;

architecture direct_arch of comp3 is
begin
  agtb <= '1' when a > b else '0';
  altb <= '1' when a < b else '0';
  aeqb <= '1' when a = b else '0';
end direct_arch;
```

```vhdl
architecture shared_arch of comp2mode is
  signal s1_b0, agtb_mag: std_logic;
begin
  s1_b0 <= '1' when a(7)='1' and b(7)='0' else '0';
  agtb_mag <= '1' when a(6 downto 0) > b(6 downto 0) else '0';
  agtb <= agtb_mag when (s1_b0='1') else s1_b0 when code='0' else
    not s1_b0;
end shared_arch;
```

```vhdl
architecture share1_arch of comp3 is
  signal gt, lt: std_logic;
begin
  gt <= '1' when a > b else '0';
  lt <= '1' when a < b else '0';
  agtb <= gt;
  altb <= lt;
  aeqb <= not (gt or lt);
end share1_arch;
```
• Read 7.3.3 and 7.3.5

4. Layout-related circuits

– After synthesis, placement and routing will derive the actual physical layout of a digital circuit on a silicon chip.
– VHDL cannot specify the exact layout
– VHDL can outline the general “shape”

– Silicon chip is a “square”
– “Two-dimensional” shape (tree or rectangular) is better than one-dimensional shape (cascading-chain)
– Conditional signal assignment/if statement form a single “horizontal” cascading chain
– Selected signal assignment/case statement form a large “vertical” mux
– Neither is ideal
e.g., Reduced-xor circuit

\[ \overline{a_7 \oplus a_6 \oplus a_5 \oplus a_4 \oplus a_3 \oplus a_2 \oplus a_1 \oplus a_0} \]

```
library ieee;
use ieee.std_logic_1164.all;

entity reduced_xor is
  port (a: in std_logic_vector(7 downto 0); y: out std_logic);
end entity reduced_xor;
```

```
architecture cascade1_arch of reduced_xor is
begin
  y <= a(0) xor a(1) xor a(2) xor a(3) xor a(4) xor a(5) xor a(6) xor a(7);
end architecture cascade1_arch;
```

• Comparison of n-input reduced xor
  – Cascading chain:
    – Area: (n-1) xor gates
    – Delay: (n-1)
    – Coding: easy to modify (scale)
  – Tree:
    – Area: (n-1) xor gates
    – Delay: \( \log_2 n \)
    – Coding: not so easy to modify
  – Software should able to do the conversion automatically

```
architecture cascade2_arch of reduced_xor is
begin
  p(0) <= a(0); p(1) <= p(0) xor a(1);
  p(2) <= p(1) xor a(2);
  p(3) <= p(2) xor a(3);
  p(4) <= p(3) xor a(4);
  p(5) <= p(4) xor a(5);
  p(6) <= p(5) xor a(6);
  p(7) <= p(6) xor a(7);
  y <= p(7);
end architecture cascade2_arch;
```

```
architecture tree_arch of reduced_xor is
begin
  y <= ((a(7) xor a(6)) xor (a(5) xor a(4))) xor
       ((a(3) xor a(2)) xor (a(1) xor a(0)));
end architecture tree_arch;
```

e.g., Reduced-xor-vector circuit

\[
\begin{align*}
y_0 &= u_0 \\
y_1 &= u_1 \oplus u_0 \\
y_2 &= u_2 \oplus u_1 \oplus u_0 \\
y_3 &= u_3 \oplus u_2 \oplus u_1 \oplus u_0 \\
y_4 &= u_4 \oplus u_3 \oplus u_2 \oplus u_1 \oplus u_0 \\
y_5 &= u_5 \oplus u_4 \oplus u_3 \oplus u_2 \oplus u_1 \oplus u_0 \\
y_6 &= u_6 \oplus u_5 \oplus u_4 \oplus u_3 \oplus u_2 \oplus u_1 \oplus u_0 \\
y_7 &= u_7 \oplus u_6 \oplus u_5 \oplus u_4 \oplus u_3 \oplus u_2 \oplus u_1 \oplus u_0
\end{align*}
\]
• Direct implementation

```vhdl
entity reduced_xor_vector is
  port(
    a : in std_logic_vector(7 downto 0);
    p : out std_logic_vector(7 downto 0)
  );
end reduced_xor_vector;

architecture direct_arch of reduced_xor_vector is
begin
  y(0) <= a(0);
  y(1) <= a(1) xor a(0);
  y(2) <= a(2) xor a(1) xor a(0);
  y(3) <= a(3) xor a(2) xor a(1) xor a(0);
  y(4) <= a(4) xor a(3) xor a(2) xor a(1) xor a(0);
  y(5) <= a(5) xor a(4) xor a(3) xor a(2) xor a(1) xor a(0);
  y(6) <= a(6) xor a(5) xor a(4) xor a(3) xor a(2) xor a(1) xor a(0);
  y(7) <= a(7) xor a(6) xor a(5) xor a(4) xor a(3) xor a(2) xor a(1) xor a(0);
end direct_arch;
```

• Direct tree implementation

```vhdl
architecture direct_tree_arch of reduced_xor_vector is
begin
  y(0) <= a(0);
  y(1) <= a(1) xor a(0);
  y(2) <= a(2) xor a(1) xor a(0);
  y(3) <= a(3) xor a(2) xor a(1) xor a(0);
  y(4) <= (a(4) xor a(3)) xor (a(2) xor a(1) xor a(0));
  y(5) <= (a(5) xor a(4)) xor (a(3) xor a(2) xor a(1) xor a(0));
  y(6) <= (a(6) xor a(5)) xor (a(4) xor a(3) xor a(2) xor a(1) xor a(0));
  y(7) <= (a(7) xor a(6)) xor (a(5) xor a(4) xor a(3) xor a(2) xor a(1) xor a(0));
end direct_tree_arch;
```

• “Parallel-prefix” implementation

```vhdl
architecture parallel_prefix_arch of reduced_xor_vector is
begin
  p(0) <= a(0);
  p(1) <= p(0) xor a(1);
  p(2) <= p(1) xor a(2);
  p(3) <= p(2) xor a(3);
  p(4) <= p(3) xor a(4);
  p(5) <= p(4) xor a(5);
  p(6) <= p(5) xor a(6);
  p(7) <= p(6) xor a(7);
end parallel_prefix_arch;
```

• Functionality Sharing

```vhdl
architecture shared_arch of reduced_xor_vector is
begin
  p(0) <= a(0);
  p(1) <= p(0) xor a(1);
  p(2) <= p(1) xor a(2);
  p(3) <= p(2) xor a(3);
  p(4) <= p(3) xor a(4);
  p(5) <= p(4) xor a(5);
  p(6) <= p(5) xor a(6);
  p(7) <= p(6) xor a(7);
end shared_arch;
```

• Comparison of n-input reduced-xor-vector
  - Cascading chain
    - Area: (n-1) xor gates
    - Delay: (n-1)
    - Coding: easy to modify (scale)
  - Multiple trees
    - Area: $O(n^2)$ xor gates
    - Delay: $\log n$
    - Coding: not so easy to modify
  - Parallel-prefix
    - Area: $O(n \log n)$ xor gates
    - Delay: $\log n$
    - Coding: difficult to modify
    - Software is not able to convert cascading chain to parallel-prefix
e.g., Shifter (rotating right)

- **Direct implementation**
  ```vhdl
  entity rotate_right is
  port(
    a : in std_logic_vector(7 downto 0);
    y : out std_logic_vector(7 downto 0)
  );
  end rotate_right;

  architecture direct_arch of rotate_right is
  begin
    with y select
    y <= a when "000",
    a(6) & a(7) when "001",
    a(0) & a(7) when "010",
    a(1) & a(7) when "011",
    a(2) when "100",
    a(3) & a(7) when "101",
    a(3) when "110",
    a(0) & a(7) when others; -- (1)
  end direct_arch;
  ```

- **Better implementation**
  ```vhdl
  entity rotate_right is
  port(
    a : in std_logic_vector(7 downto 0);
    y : out std_logic_vector(7 downto 0)
  );
  end rotate_right;

  architecture multi_level_arch of rotate_right is
  signal l0_out, l1_out, l2_out:
  std_logic_vector(7 downto 0);
  begin
    -- level 0, shift 0 or 1 bit
    l0_out <= a and a(7) when a(0) = '1' else a;
    -- level 1, shift 0 or 2 bits
    l1_out <= l0_out(7 downto 0) & l0_out(7 downto 2) when a(1) = '1' else l0_out;
    -- level 2, shift 0 or 4 bits
    l2_out <= l1_out(3 downto 0) & l1_out(7 downto 4) when a(2) = '1' else l1_out;
    y <= l2_out;
  end multi_level_arch;
  ```

- **Comparison for n-bit shifter**
  - **Direct implementation**
    - n 1-to-1 mux
    - vertical strip with O(n^2) input wiring
    - Code not so easy to modify
  - **Staged implementation**
    - n log_2 n 2-to-1 mux
    - Rectangular shaped
    - Code easier to modify

5. General examples

- Gray code counter
- Signed addition with status
- Simple combinational multiplier
e.g., Gray code counter

<table>
<thead>
<tr>
<th>binary code</th>
<th>gray code</th>
<th>incremented gray code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0001</td>
</tr>
<tr>
<td>0001</td>
<td>0011</td>
<td>0011</td>
</tr>
<tr>
<td>0010</td>
<td>0110</td>
<td>0111</td>
</tr>
<tr>
<td>0011</td>
<td>0101</td>
<td>0100</td>
</tr>
<tr>
<td>0100</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>0101</td>
<td>1011</td>
<td>1010</td>
</tr>
<tr>
<td>0110</td>
<td>1100</td>
<td>1101</td>
</tr>
<tr>
<td>0111</td>
<td>1111</td>
<td>1110</td>
</tr>
<tr>
<td>1000</td>
<td>0000</td>
<td>0001</td>
</tr>
<tr>
<td>1001</td>
<td>0011</td>
<td>0010</td>
</tr>
<tr>
<td>1010</td>
<td>1010</td>
<td>1001</td>
</tr>
<tr>
<td>1011</td>
<td>1101</td>
<td>1100</td>
</tr>
<tr>
<td>1100</td>
<td>1110</td>
<td>1111</td>
</tr>
<tr>
<td>1101</td>
<td>1101</td>
<td>1100</td>
</tr>
<tr>
<td>1110</td>
<td>1000</td>
<td>1001</td>
</tr>
<tr>
<td>1111</td>
<td>1011</td>
<td>1010</td>
</tr>
</tbody>
</table>

• Direct implementation

```vhdl
architecture rtl_arch of g_inc is
begin
  g_inc <= std_logic_vector(2**WIDTH downto 0);
  if WIDTH = 4 then
    for i in 0 to 3 loop
      g_inc(i) <= (g_inc(i) + 1) mod 2;
    end loop;
  else
    for i in 0 to WIDTH - 1 loop
      g_inc(i) <= (g_inc(i) + 1) mod 2;
    end loop;
  end if;
end rtl_arch;
```

• Observation
  - Require 2^n rows
  - No simple algorithm for gray code increment
  - One possible method
    • Gray to binary
    • Increment the binary number
    • Binary to gray

• Binary to gray
  - g_i = b_i \oplus b_{i+1}
  - g_0 = b_0 \oplus g_0
  - g_i = b_i \oplus b_0
  - g_0 = b_0

• Gray to binary
  - b_i = g_i \oplus b_{i+1}
  - b_0 = g_0 \oplus b_0
  - b_i = g_i \oplus b_i
  - b_0 = b_0

e.g., signed addition with status

• Adder with
  - Carry-in: need an extra bit (LSB)
  - Carry-out: need an extra bit (MSB)
  - Overflow:
    • two operands has the same sign but the sum has a different sign
      \[ overflow = (s_0 \cdot s_b \cdot s) + (s_0^t \cdot s_b^t \cdot s) \]
    - Zero
    - Sign (of the addition result)
e.g., simple combinational multiplier

\[
\begin{array}{cccccccc}
\times & a_3 & a_2 & a_1 & a_0 & b_3 & b_2 & b_1 & b_0 \\
\hline
a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 \\
a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 \\
a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 \\
\end{array}
\]

\[\text{product} = a_3b_3 + a_2b_2 + a_1b_1 + a_0b_0\]

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity multi8 is
  port(
    a, b: in std_logic_vector(7 downto 0);
    y: out std_logic_vector(15 downto 0)
  );
end multi8;

architecture cpu1_arch of multi8 is
  constant WIDTH: integer := 8;
signal ao, bo0, bo1, bo2, bo3, bo4, bo5, bo6, bo7: unsigned(WIDTH-1 downto 0);
signal p0,p1,p2,p3,p4,p5,p6,p7,p8,p9: prod: unsigned(2*WIDTH-1 downto 0);
begin
  n0 <= unsigned(a);
  n1 <= (others=>b(0));
  n2 <= (others=>b(1));
  n3 <= (others=>b(2));
  n4 <= (others=>b(3));
  n5 <= (others=>b(4));
  n6 <= (others=>b(5));
  n7 <= (others=>b(7));
  p0 <= "00000000" & (bo0 and ao);
  p1 <= "00000000" & (bo1 and ao) & "00";
  p2 <= "00000000" & (bo2 and ao) & "00";
  p3 <= "00000000" & (bo3 and ao) & "00000000";
  p4 <= "00000000" & (bo4 and ao) & "00000000";
  p5 <= "00000000" & (bo5 and ao) & "00000000";
  p6 <= "00000000" & (bo6 and ao) & "00000000";
  p7 <= "00000000" & (bo7 and ao) & "00000000";
  prod <= (p0+p1+p2+p3+p4+p5+p6+p7); y <= std_logic_vector(prod);
end cpu1_arch;
```