Combinational Circuit Design

This slide set covers

- Derivation of efficient HDL description
- Operator sharing
- Functionality sharing
- Layout-related circuits
- General circuits

Operator Sharing

Sharing of resources often results in a performance penalty at the benefit of area savings

<table>
<thead>
<tr>
<th>width</th>
<th>nand</th>
<th>xor</th>
<th>&gt;a</th>
<th>&gt;d</th>
<th>=+1a</th>
<th>+1d</th>
<th>+a</th>
<th>+d</th>
<th>mux</th>
</tr>
</thead>
<tbody>
<tr>
<td>area (gate count)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>22</td>
<td>25</td>
<td>68</td>
<td>26</td>
<td>27</td>
<td>33</td>
<td>51</td>
<td>118</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>44</td>
<td>52</td>
<td>102</td>
<td>51</td>
<td>55</td>
<td>73</td>
<td>101</td>
<td>265</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>85</td>
<td>105</td>
<td>211</td>
<td>102</td>
<td>113</td>
<td>153</td>
<td>203</td>
<td>437</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>171</td>
<td>212</td>
<td>398</td>
<td>204</td>
<td>227</td>
<td>313</td>
<td>405</td>
<td>755</td>
</tr>
</tbody>
</table>

| delay (ns) |
| 8    | 0.1  | 0.4 | 4.0 | 1.9 | 1.0  | 2.4 | 1.5 | 4.2 | 3.2 | 0.3 |
| 16   | 0.1  | 0.4 | 8.6 | 3.7 | 1.7  | 5.5 | 3.3 | 8.2 | 5.5 | 0.3 |
| 32   | 0.1  | 0.4 | 17.6| 6.7 | 1.8  | 11.6| 7.5 | 16.2| 11.1| 0.3 |
| 64   | 0.1  | 0.4 | 35.7| 14.3| 2.2  | 24.0| 15.7| 32.2| 22.9| 0.3 |
Operator Sharing

Ideally, synthesis software can identify these opportunities automatically, but in practice, this is rarely the case.

There are usually lots of opportunities to share resources using the basic VHDL constructs b/c in many cases, operations are \textit{mutually exclusive}.

The value expressions in priority network and multiplexing network are mutually exclusive in the sense that only one result is routed to the output.

Conditional signal assignment (same for \textit{if stmt})

\begin{verbatim}
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 ...
\end{verbatim}

Selected signal assignment (same for \textit{case stmt})

\begin{verbatim}
with select_expression select
    sig_name <= value_expr_1 when choice_1, value_expr_2 when choice_2, ... value_expr_n when choice_n;
\end{verbatim}
**Operator Sharing**

If same operator is used in different expressions, it can be shared.

Example 1: Original code:

```
r <= a+b when boolean_exp else
   a+c;
```

Revised code: here the operands are routed, not the output of the adder

```
src0 <= b when boolean_exp else
   c;

r <= a + src0;
```

Original: **Area**: 2 adders, 1 MUX, **Delay**: max($T_{adder}$, $T_{boolean}$) + $T_{MUX}$

With Sharing: **Area**: 1 adders, 1 MUX, **Delay**: $T_{boolean}$ + $T_{MUX}$ + $T_{adder}$

(a) Original diagram

(b) Diagram with sharing
Operator Sharing

Example 2: Original code:

```vhdl
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:

```vhdl
process (a, b, c, d, ...) begin
    if boolean_exp_1 then
        src0 <= a;
        src1 <= b;
    elsif boolean_exp_2 then
        r <= a+c;
    else
        r <= d+1;
    end if
end process;
```
Operator Sharing

```
elsif boolean_exp_2 then
    src0 <= a;
    src1 <= c;
else
    src0 <= d;
    src1 <= "00000001"; -- MUX with constants can be
                        -- optimized by synthesis,
                        -- yielding more area savings
end if;
end process;
```

```
Original: Area: 2 adders, 1 inc, 2 MUX
With Sharing: Area: 1 adder, 4 MUX
```
Operator Sharing

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",
        d when others;

with sel_exp select
    src1 <= b when "00",
        c when "01",
        "00000001" when others;

r <= src0 + src1;
```
Operator Sharing

Note that the revised implementation has longer delay because of:

- The increased number of cascaded components in some cases
- The restriction on the available parallelism in other cases

Original: **Area**: 2 adders, 1 inc, 1 MUX
With Sharing: **Area**: 1 adder, 2 MUX

Note that in the revised scheme, the *sel exp* Boolean logic MUST be evaluated first before the addition takes place -- this is not the case in the original version.
Operator Sharing

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;
```

Original: **Area**: 2 adders, 2 MUX
Operator Sharing

Revised code:

```vhdl
process(a, b, c, d, ...)
begin
    if boolean_exp then
        src0 <= a;
        src1 <= b;
        x <= sum;
        y <= (others=>'0');
    else
        src0 <= c;
        src1 <= d;
        x <= (others=>'1');
        y <= sum;
    end if;
end process;

sum <= src0 + src1;
```
Operator Sharing

Worst case situation in which operator has **no** common sources or destinations

Is the sharing worthwhile in this case?

1 adder saved in original version but 2 MUX added in revised scheme

It depends on the size of the adder -- if optimized for speed, then it can be significantly larger than 2 MUX

Summary

- Merit of sharing depends on the complexity of the operator and the routing circuit
- Complex operators provide a lot of area savings
- Cost is increased propagation delay because of serial operation

**With Sharing: Area:**

1 adders, 4 MUX
Functionality Sharing
A large circuit such as a microcontroller includes a lot of functions

Several functions may be related and can share a common circuit

Identifying these opportunities is more difficult and is something synthesis software can NOT do

Done in an ad hoc manner by designer and is based on his/her expertise

Consider add-sub circuit

<table>
<thead>
<tr>
<th>ctrl</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a + b</td>
</tr>
<tr>
<td>1</td>
<td>a - b</td>
</tr>
</tbody>
</table>

Straightforward translation into VHDL:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity addsub is
```
Functionality Sharing

```vhdl
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
  signal src0, src1, sum: signed(7 downto 0);
begin
  src0 <= signed(a);  
  src1 <= signed(b);  
  sum <= src0 + src1 when ctrl='0' else  
      src0 - src1;  
  r <= std_logic_vector(sum);
end direct_arch;
```
Functionality Sharing

This version is translated such that it includes an adder, subtractor and MUX

\[
\begin{align*}
   a + \bar{b} + 1
\end{align*}
\]

```vhdl
architecture shared_arch of addsub is
    signal src0, src1, sum: signed(7 downto 0);
    signal cin: signed(0 downto 0); -- carry-in bit
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;
```
Functionality Sharing

The ’+ 1’ is implemented by setting the carry-in bit to ’1’ of the adder

Most synthesis software should deduce that the ’+ cin’ is one bit and can be implemented in this fashion

Alternatively, you can manually describe the carry-in in the adder by adding an extra bit to the adder and operands

Original operands $a_7a_6a_5a_4a_3a_2a_1a_0$ and $b_7b_6b_5b_4b_3b_2b_1b_0$

New operands $a_7a_6a_5a_4a_3a_2a_1a_01$ and $b_7b_6b_5b_4b_3b_2b_1b_0c_{in}$
Functionality Sharing
After the addition, discard the low order bit

```vhdl
architecture manual_carry_arch of addsub is
  signal src0, src1, sum: signed(8 downto 0);
  signal b_tmp: std_logic_vector(7 downto 0);
  signal cin: std_logic; -- carry-in bit
begin
  src0 <= signed(a & '1');
  b_tmp <= b when ctrl='0' else not b;
  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;
```
Functionality Sharing

As we know, \texttt{ieee.numeric\_std} provides signed and unsigned, with signed in 2’s complement format.

Here, addition and subtraction operations are identical and therefore, the same hardware can be used for either data type.
Functionality Sharing

Unfortunately, this is not true for the relational operators, and therefore we need to craft a control signal into the VHDL code

```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;
begi
```
**Functionality Sharing**

```vhdl
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;
```

To create an opportunity for sharing, we need to handle the sign bit separately for signed operands.

If the sign bits are **different**, then the positive number is larger for signed.

If they are the **same**, compare the n-1 bits (without MSB) using normal comparison.

Consider 1111 (-1), 1100 (-4), 1001(-7) -- after removing the MSB (sign bit)

111 > 100 > 001

Which is consistent with -1 > -4 > -7, so we can share the LSB compare logic.
Functionality Sharing

architecture shared_arch of comp2mode is

signal a1_b0, agtb_mag: std_logic;

begin

a1_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 (a(7)=b(7)) else
       a1_b0 when mode=’0’ else -- unsigned mode
       not a1_b0; -- signed mode

end shared_arch;

Rules are

• If \(a\) and \(b\) have same sign bit, compare in regular fashion
• If \(a\)’s sign bit is ’1’ and \(b\)’s sign bit is ’0’, \(a\) is greater than \(b\) when in unsigned mode and \(b\) is greater than \(a\) in signed mode
• If \(a\)’s sign bit is ’0’ and \(b\)’s sign bit is ’1’, reverse the previous result

New version is about 1/2 size of dual mode version
Functionality Sharing
Assume we need a full comparator, i.e., one that provides greater-than, equal-to and less-than -- straightforward approach

```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;
```
Functionality Sharing

An obvious change is to share the resources of two of the compares to derive the third.

Another optimization is to recognize that the equal-to comparator is faster and smaller than the other two.

```vhdl
architecture share2_arch of comp3 is
  signal eq, lt: std_logic;
begin
  eq <= '1' when a = b else '0';
  lt <= '1' when a < b else '0';
  aeqb <= eq;
  altb <= lt;
  agtb <= not (eq or lt);
end share2_arch;
```

Text covers a

- **absolute difference** circuit
- **three function barrel shifter** circuit
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, but it can control the general "shape"

In general, "square" or 2-D circuits are better than a 1-D 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

1-D has long delay

Better, 2-D has shorter delay
Layout-Related Circuits

Consider the reduced-xor circuit (covered before), in which all input bits of the operand are XOR’ed to produce the output.

The previous 1-D schematic is described as follows:

```vhdl
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 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 cascade1_arch;
```
Layout-Related Circuits

We can also use an 8-bit internal signal $p$ to represent intermediate results

```vhdl
architecture cascade2_arch of reduced_xor is
  signal p: std_logic_vector(7 downto 0);
  begin
    p(0) <= '0' xor 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 cascade2_arch;
```

The repetitive nature allows for a more compact vector form as shown below
Layout-Related Circuits

```vhdl
architecture cascade_compact_arch of reduced_xor is
    constant WIDTH: integer := 8;
    signal p: std_logic_vector(WIDTH-1 downto 0);
begin
    p <= (p(WIDTH-2 downto 0) & '0') xor a;
    y <= p(WIDTH-1);
end cascade_compact_arch;
```

Although this design uses the minimal number of XOR gates, it suffers from long propagation delay

Although the synthesis tool is likely to produce a 2-D structure given the simplicity of this circuit, the following is one way to guarantee it

```vhdl
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 tree_arch;
```
Layout-Related Circuits

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

Consider a reduced-xor-vector circuit

\[
\begin{align*}
    y_0 &= a_0 \\
    y_1 &= a_1 \oplus a_0 \\
    y_2 &= a_2 \oplus a_1 \oplus a_0 \\
    y_3 &= a_3 \oplus a_2 \oplus a_1 \oplus a_0 \\
    y_4 &= a_4 \oplus a_3 \oplus a_2 \oplus a_1 \oplus a_0 \\
    y_5 &= a_5 \oplus a_4 \oplus a_3 \oplus a_2 \oplus a_1 \oplus a_0 \\
    y_6 &= a_6 \oplus a_5 \oplus a_4 \oplus a_3 \oplus a_2 \oplus a_1 \oplus a_0 \\
    y_7 &= a_7 \oplus a_6 \oplus a_5 \oplus a_4 \oplus a_3 \oplus a_2 \oplus a_1 \oplus a_0
\end{align*}
\]

Here, all combinations of the lower bits of the input signal are xor’ed to produce 8 outputs, $y_i$

The straightforward (no sharing) implementation is given below
library ieee;
use ieee.std_logic_1164.all;

entity reduced_xor_vector is
    port(
        a: in std_logic_vector(7 downto 0);
        y: 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);
end direct_arch;
Layout-Related Circuits

\[
y(6) \leq a(6) \text{xor} \ a(5) \text{xor} \ a(4) \text{xor} \ a(3) \text{xor} \ a(2) \text{xor} \ a(1) \text{xor} \ a(0); \\
y(7) \leq a(7) \text{xor} \ a(6) \text{xor} \ a(5) \text{xor} \ a(4) \text{xor} \ a(3) \text{xor} \ a(2) \text{xor} \ a(1) \text{xor} \ a(0);
\]

end direct_arch;

This requires 28 xor gates if implemented un-optimized -- there are lots of common sub-expressions that can be shared

Code that shares is very similar to the reduced-xor code given earlier except all intermediate results are used as outputs

architecture shared1_arch of reduced_xor_vector is
  signal p: std_logic_vector(7 downto 0);
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);
Layout-Related Circuits

\[
\begin{align*}
    p(4) &= p(3) \oplus a(4); \\
    p(5) &= p(4) \oplus a(5); \\
    p(6) &= p(5) \oplus a(6); \\
    p(7) &= p(6) \oplus a(7); \\
    y &= p;
\end{align*}
\]

As before, the pattern of assignments can be coded more efficiently

\begin{verbatim}
architecture shared_compact_ar of reduced_xor_vector is
    constant WIDTH: integer := 8;
    signal p: std_logic_vector(WIDTH-1 downto 0);
begin
    p <= (p(WIDTH-2 downto 0) & '0') \oplus a;
    y <= p;
end shared_compact_ar;
\end{verbatim}

All of these designs specify a 1-D structure, with the critical path to \( y(7) \)
Text gives version with parenthesis to force 2-D tree-type design
Layout-Related Circuits

An *ad hoc* version that reduces both propagation delay to 3 gates AND uses only 12 *xor* gates

```
architecture optimal_tree_arch of reduced_xor_vector is
  signal p01, p23, p45, p67, p012,
    p0123, p456, p4567: std_logic;
begin
  p01 <= a(0) xor a(1);
  p23 <= a(2) xor a(3);
  p45 <= a(4) xor a(5);
  p67 <= a(6) xor a(7);
  p012 <= p01 xor a(2);
  p0123 <= p01 xor p23;
  p456 <= p45 xor a(6);
  p4567 <= p45 xor p67;
  y(0) <= a(0);
  y(1) <= p01;
  y(2) <= p012;
  y(3) <= p0123;
```
Layout-Related Circuits

\[
\begin{align*}
y(4) & \leq p0123 \text{ xor } a(4); \\
y(5) & \leq p0123 \text{ xor } p45; \\
y(6) & \leq p0123 \text{ xor } p456; \\
y(7) & \leq p0123 \text{ xor } p4567;
\end{align*}
\]

end optimal_tree_arch;

```
```
Layout-Related Circuits

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**: $\log_2 n$, **Coding**: not so easy to modify

- **Optimal tree**
  
  **Area**: $O(n\log_2 n)$ xor gates, **Delay**: $\log_2 n$, **Coding**: difficult to modify

Unlike the previous example, synthesis is not able to convert cascading chain to the optimal tree (parallel-prefix)

Next consider a **barrel shifter** -- direct implementation

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity rotate_right is
  port(
    a: in std_logic_vector(7 downto 0);
```
Layout-Related Circuits
(Barrel shifter)

```vhdl
amt: in std_logic_vector(2 downto 0);
y: out std_logic_vector(7 downto 0);
end rotate_right;

architecture direct_arch of rotate_right is
begin
with amt select
y<= a when "000",
a(0) & a(7 downto 1) when "001",
a(1 downto 0) & a(7 downto 2) when "010",
a(2 downto 0) & a(7 downto 3) when "011",
a(3 downto 0) & a(7 downto 4) when "100",
a(4 downto 0) & a(7 downto 5) when "101",
a(5 downto 0) & a(7 downto 6) when "110",
a(6 downto 0) & a(7) when others; -- 111
end direct_arch;
```
Layout-Related Circuits

The barrel shifter rotates the input \(a\) by the amount specified, from 0 to 7 rotates 8-bit 8-to-1 MUXs.

The code is realized using eight 1-bit 8-to-1 MUXs.
Layout-Related Circuits

Problems with this include:

- Wide MUXs cannot be effectively mapped to certain device technologies
- Input wires, \( a \), route to all MUXs, so loading and congestion grows as \( O(n^2) \)
- The 'single narrow strip' shape makes place and route difficult

Better to do the wiring in levels

In each level, each bit of the \( amt \) signal determines if we rotate or pass through

Note that the rotate amounts are different depending on the bit’s position

After passing through all three levels, the number of rotations performed is equal to the sum of those performed at each level

\[
\text{amt}(2) \times 2^2 + \text{amt}(1) \times 2^1 + \text{amt}(0) \times 2^0
\]
Layout-Related Circuits

architecture multi_level_arch of rotate_right is
  signal le0_out, le1_out, le2_out:
    std_logic_vector(7 downto 0);

begin
  -- level 0, shift 0 or 1 bit
  le0_out <= a(0) & a(7 downto 1) when amt(0)=’1’ else a;

  -- level 1, shift 0 or 2 bits
  le1_out <= le0_out(1 downto 0) & le0_out(7 downto 2)
    when amt(1)=’1’ else le0_out;

  -- level 2, shift 0 or 4 bits
  le2_out <= le1_out(3 downto 0) & le1_out(7 downto 4)
    when amt(2)=’1’ else le1_out;

  y <= le2_out;
end multi_level_arch;
Layout-Related Circuits
Comparison for n-bit shifter
• Direct implementation
  \( n \) n-to-1 MUX
  Vertical strip with \( O(n^2) \) input wiring
  Code not so easy to modify
• Staged implementation
  \( n \times \log_2 n \) 2-to-1 MUX (8 2-to-1 MUXs at each level for a total of 24)
  Rectangular shaped
  Code easier to modify

General Examples
• Gray code counter
• Signed addition with status
• Simple combinational multiplier

Gray code is a special sequence of values where only one bit changes in any two successive code.
Gray Code

Thus it minimizes the *number of transitions* that a signal switches between successive words.

<table>
<thead>
<tr>
<th>binary code</th>
<th>gray code</th>
<th>gray code</th>
<th>incremented gray code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_3b_2b_1b_0$</td>
<td>$g_3g_2g_1g_0$</td>
<td>0000</td>
<td>0001</td>
</tr>
<tr>
<td>0001</td>
<td>0001</td>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>0011</td>
<td>0010</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>0010</td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>0110</td>
<td>0110</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>0111</td>
<td>0111</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>0101</td>
<td>0101</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>0100</td>
<td>0100</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>1100</td>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>1101</td>
<td>1101</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>1111</td>
<td>1111</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>1110</td>
<td>1110</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>1010</td>
<td>1010</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>1011</td>
<td>1011</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>1001</td>
<td>1001</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>1000</td>
<td>1000</td>
<td></td>
</tr>
</tbody>
</table>

Need to implement the gray code incrementer on the right.

Straightforward way is to translate the table into a *selected signal assignment* stmt.
Gray Code

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

entity g_inc is
  port(
    g: in std_logic_vector(3 downto 0);
    g1: out std_logic_vector(3 downto 0)
  );
end g_inc;

architecture table_arch of g_inc is
begin
  with g select
    g1 <= "0001" when "0000",
         "0011" when "0001",
         "0010" when "0011",
         "0110" when "0010",
         "0111" when "0110",
         "1111" when "1111",
         "1011" when "1010",
         "1001" when "1000",
         "1000" when "1001",
         "1010" when "1110",
         "1011" when "1111",
         "1101" when "1100",
         "1100" when "1101",
         "1110" when "1110",
         "0100" when "0101",
         "0101" when "0111",
         "0111" when "0110",
         "0011" when "0010",
         "0010" when "0001",
         "0000" when "0000"
  );
end table_arch;
```
Gray Code

<table>
<thead>
<tr>
<th>Binary</th>
<th>Gray Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101</td>
<td>0111</td>
</tr>
<tr>
<td>0100</td>
<td>0101</td>
</tr>
<tr>
<td>1100</td>
<td>0100</td>
</tr>
<tr>
<td>1101</td>
<td>1100</td>
</tr>
<tr>
<td>1111</td>
<td>1101</td>
</tr>
<tr>
<td>1110</td>
<td>1111</td>
</tr>
<tr>
<td>1010</td>
<td>1110</td>
</tr>
<tr>
<td>1011</td>
<td>1010</td>
</tr>
<tr>
<td>1001</td>
<td>1011</td>
</tr>
<tr>
<td>1000</td>
<td>1001</td>
</tr>
<tr>
<td>0000</td>
<td>others; -- 1000</td>
</tr>
</tbody>
</table>

end table_arch;

Although this VHDL code is simple, it is not scalable b/c revisions take on order O(n^2)

Unfortunately, there is no easy algorithm to derive the next Gray code directly

However, an algorithm does exist to convert between binary and Gray code, and therefore one implementation is to use a binary incrementer
Gray Code

So the algorithm is to 1) convert a Gray code to binary, 2) increment binary and 3) covert back

The conversion is based on the following

The $i$th bit ($g_i$) of the Gray code word is '1' if the $i$th bit and $(i + 1)$th bit, i.e., $b_i$ and $b_{i+1}$ of the corresponding binary word are different

<table>
<thead>
<tr>
<th>binary code</th>
<th>gray code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_3b_2b_1b_0$</td>
<td>$g_3g_2g_1g_0$</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>0010</td>
<td>0011</td>
</tr>
<tr>
<td>0011</td>
<td>0010</td>
</tr>
<tr>
<td>0100</td>
<td>0110</td>
</tr>
<tr>
<td>0101</td>
<td>0111</td>
</tr>
<tr>
<td>0110</td>
<td>0101</td>
</tr>
<tr>
<td>0111</td>
<td>0100</td>
</tr>
<tr>
<td>1000</td>
<td>1100</td>
</tr>
<tr>
<td>1001</td>
<td>1101</td>
</tr>
<tr>
<td>1010</td>
<td>1111</td>
</tr>
<tr>
<td>1011</td>
<td>1110</td>
</tr>
<tr>
<td>1100</td>
<td>1010</td>
</tr>
<tr>
<td>1101</td>
<td>1011</td>
</tr>
<tr>
<td>1110</td>
<td>1001</td>
</tr>
<tr>
<td>1111</td>
<td>1000</td>
</tr>
</tbody>
</table>

Binary to Gray

$$g_i = b_i \oplus b_{i+1}$$

Gray to Binary

$$b_i = g_i \oplus b_{i+1}$$

Note recursive expansion is possible

Very similar to reduced-xor-vector
Gray Code

```vhdl
architecture compact_arch of g_inc is
  constant WIDTH: integer := 4;
  signal b, b1: std_logic_vector(WIDTH-1 downto 0);
begin

  -- Gray to binary
  b <= g xor ('0' & b(WIDTH-1 downto 1));

  -- binary increment
  b1 <= std_logic_vector((unsigned(b)) + 1);

  -- binary to Gray
  g1<= b1 xor ('0' & b1(WIDTH-1 downto 1));

  end compact_arch;
```

This code is independent of the input size (revision time $O(1)$) and uses a binary adder (of which there are many to choose from!)
Signed Addition with Status

The default ’+’ VHDL operator does not allow for status signals

Common status signals reflecting the result of the addition include
- zero: Is the result 0?
- sign: What is the sign of the result?
- overflow: Did the result overflow?

Also, carry signals (carry in and out) pass information between successive additions
Needed, for example, when you build a 64-bit adder from 8-bit adders

Given these status signals, it is important to note that overflow must be checked first
because the other status signals are invalid in this case

The following deductions can be made
- If the two operands have different sign bits, then overflow is NOT possible
- If the two operands and the result have the same sign, then overflow did not occur
- If the two operands have the same sign but the result has a different sign, overflow occurred
Signed Addition with Status

The following logic expression captures the last condition, with \( s_a, s_b \) and \( s_s \) representing the signs of the \( a \) and \( b \) operands and the sign of the sum, \( s \)

\[
\text{overflow} = (s_a \cdot s_b \cdot \overline{s_s}) + (\overline{s_a} \cdot \overline{s_b} \cdot s_s)
\]

Note that \( \text{zero} \) may be true when it shouldn’t be if \( \text{overflow} \) occurred, e.g., summing "1000" and "1000" using a 4-bit adder produces "0000"

Similar arguments hold for \( \text{sign} \) -- sign of the above is '0' but it should be '1’, therefore, if overflow occurs, then the \( \text{sign} \) signal should be inverted

The \( \text{carry}_\text{in} \) and \( \text{carry}_\text{out} \) signals are appended to the right and left, resp.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity adder_status is
  port (a, b: in std_logic_vector(7 downto 0));
```
Signed Addition with Status

```vhdl
    cin: in std_logic;
    sum: out std_logic_vector(7 downto 0);
    cout, zero, overflow, sign: out std_logic
);
end adder_status;

architecture arch of adder_status is
    signal a_ext, b_ext, sum_ext: signed(9 downto 0);
    signal ovf: std_logic;
    alias sign_a: std_logic is a_ext(8);
    alias sign_b: std_logic is b_ext(8);
    alias sign_s: std_logic is sum_ext(8);
    begin

        -- bit extend the operands on both sides
        a_ext <= signed('0' & a & '1');
        b_ext <= signed('0' & b & cin);
        sum_ext <= a_ext + b_ext;
```
Signed Addition with Status

\[
\text{ovf} \leq (\text{sign}_a \text{ and } \text{sign}_b \text{ and } \neg \text{sign}_s) \text{ or } \\
(\neg \text{sign}_a \text{ and } \neg \text{sign}_b \text{ and } \text{sign}_s);
\]

\[
\text{cout} \leq \text{sum}_\text{ext}(9);
\]

-- Invert sign if overflow occurred
\[
\text{sign} \leq \text{sum}_\text{ext}(8) \text{ when ovf='0' else } \\
\neg \text{sum}_\text{ext}(8);
\]

-- zero is invalid is overflow occurred
\[
\text{zero} \leq '1' \text{ when (sum}_\text{ext}(8 \text{ downto } 1)=0 \text{ and ovf='0'}) \\
\text{else '0'};
\]
\[
\text{overflow} \leq \text{ovf};
\]
\[
\text{sum} \leq \text{std_logic_vector(sum}_\text{ext}(8 \text{ downto } 1));
\]
\[
\text{end arch;}
\]
Simple Combinational Multiplier

A simple multiplier can be constructed using *first principles*

<table>
<thead>
<tr>
<th>×</th>
<th>a3</th>
<th>a2</th>
<th>a1</th>
<th>a0</th>
<th>multiplicand</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b3</td>
<td>b2</td>
<td>b1</td>
<td>b0</td>
<td>multiplier</td>
</tr>
<tr>
<td></td>
<td>a3b0</td>
<td>a2b0</td>
<td>a1b0</td>
<td>a0b0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a3b1</td>
<td>a2b1</td>
<td>a1b1</td>
<td>a0b1</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>a3b2</td>
<td>a2b2</td>
<td>a1b2</td>
<td>a0b2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a3b3</td>
<td>a2b3</td>
<td>a1b3</td>
<td>a0b3</td>
<td></td>
</tr>
</tbody>
</table>

| y7 | y6 | y5 | y4 | y3 | y2 | y1 | y0 | product |

Simple algorithm:

- Multiply the digits of the multiplier, $b_3b_2b_1b_0$ by the multiplicand by $A$, one at a time to obtain $b_3*A$, $b_2*A$, $b_1*A$ and $b_0*A$

  Given $b_i$ can only be ‘0’ or ‘1’, the product can only be ‘0’ or $A$

  Multiplication is performed using the **and** operation, i.e., $b_i*A = (a_3b_i, a_2b_i, a_1b_i, a_0b_i)$
Simple Combinational Multiplier

Simple algorithm:

- Shift $b_i \cdot A$ to the left $i$ positions
- Add the shifted $b_i \cdot A$ terms to obtain the final product

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity mult8 is
  port(
    a, b: in std_logic_vector(7 downto 0);
    y: out std_logic_vector(15 downto 0)
  );
end mult8;

architecture comb1_arch of mult8 is
  constant WIDTH: integer:=8;
  signal au, bv0, bv1, bv2, bv3, bv4, bv5, bv6, bv7: unsigned(WIDTH-1 downto 0);
```

Simple Combinational Multiplier

    signal p0, p1, p2, p3, p4, p5, p6, p7, prod:
          unsigned(2*WIDTH-1 downto 0);

    begin

        au <= unsigned(a);
        bv0 <= (others=>b(0));
        bv1 <= (others=>b(1));
        bv2 <= (others=>b(2));
        bv3 <= (others=>b(3));
        bv4 <= (others=>b(4));
        bv5 <= (others=>b(5));
        bv6 <= (others=>b(6));
        bv7 <= (others=>b(7));

        p0 <="00000000" & (bv0 and au);
        p1 <="0000000" & (bv1 and au) & "0";
        p2 <="000000" & (bv2 and au) & "00";
        p3 <="00000" & (bv3 and au) & "000";
        p4 <="0000" & (bv4 and au) & "0000";
Simple Combinational Multiplier

```vhdl
p5 <= "000" & (bv5 and au) & "00000";
p6 <= "00" & (bv6 and au) & "000000";
p7 <= "0" & (bv7 and au) & "0000000";
prod <= ((p0+p1)+(p2+p3))+((p4+p5)+(p6+p7));
y <= std_logic_vector(prod);
end comb1_arch;
```

See text for alternative architecture, as well as examples of a Hamming distance circuit and programmable priority encoder.