# 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

| width | VHDL operator | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | nand | xor | $>_a$ | $>_d$ | = | $+1_a$ | $+1_d$ | $+_a$ | $+_d$ | mux |
| | | | | | **area (gate count)** | | | | | |
| 8 | 8 | 22 | 25 | 68 | 26 | 27 | 33 | 51 | 118 | 21 |
| 16 | 16 | 44 | 52 | 102 | 51 | 55 | 73 | 101 | 265 | 42 |
| 32 | 32 | 85 | 105 | 211 | 102 | 113 | 153 | 203 | 437 | 85 |
| 64 | 64 | 171 | 212 | 398 | 204 | 227 | 313 | 405 | 755 | 171 |
| | | | | | **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 |

# 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)

```
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;
```

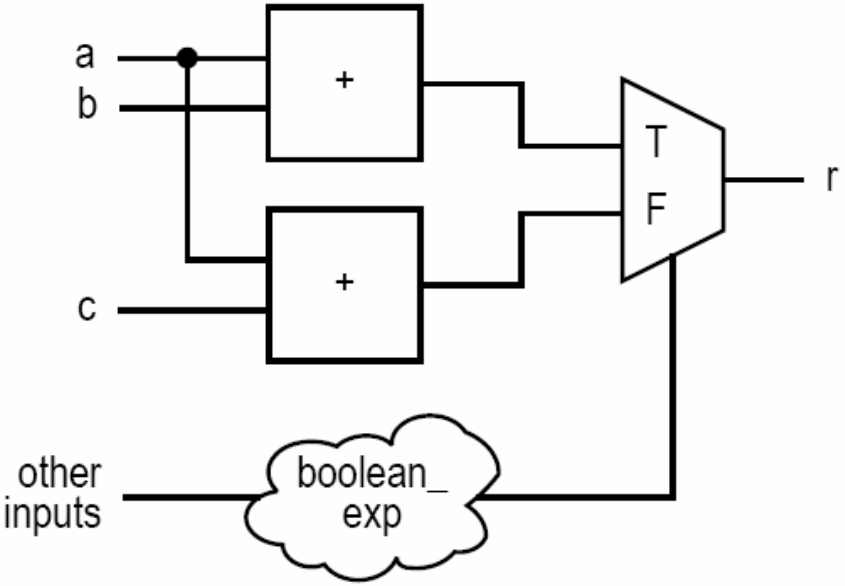
# Example 1

- Original code:

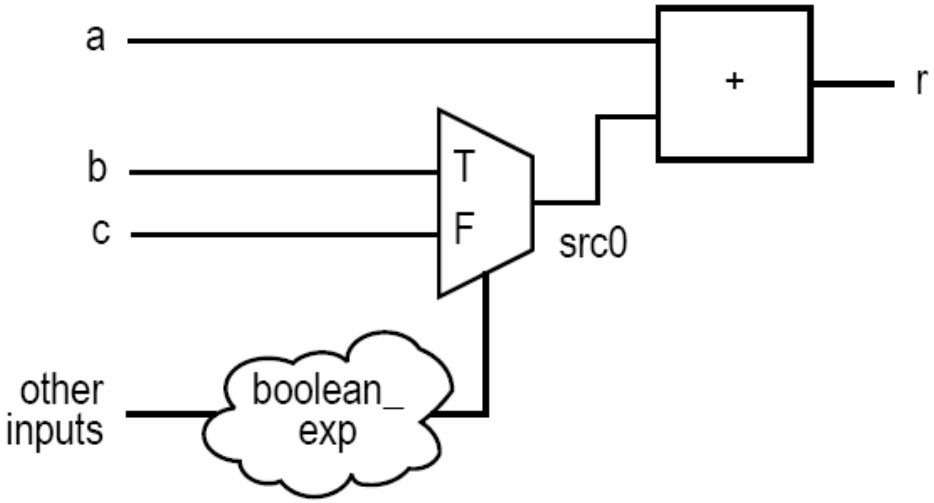
```
r <= a+b when boolean_exp else
     a+c;
```

- Revised code:

```
src0 <= b when boolean_exp else
        c;
r <= a + src0;
```




(a) Original diagram




(b) Diagram with sharing

Area: 2 adders, 1 mux

Delay:

$$\max(T_{adder}, T_{boolean}) + T_{mux}$$

Area: 1 adder, 1 mux

Delay:

$$T_{boolean} + T_{mux} + T_{adder}$$

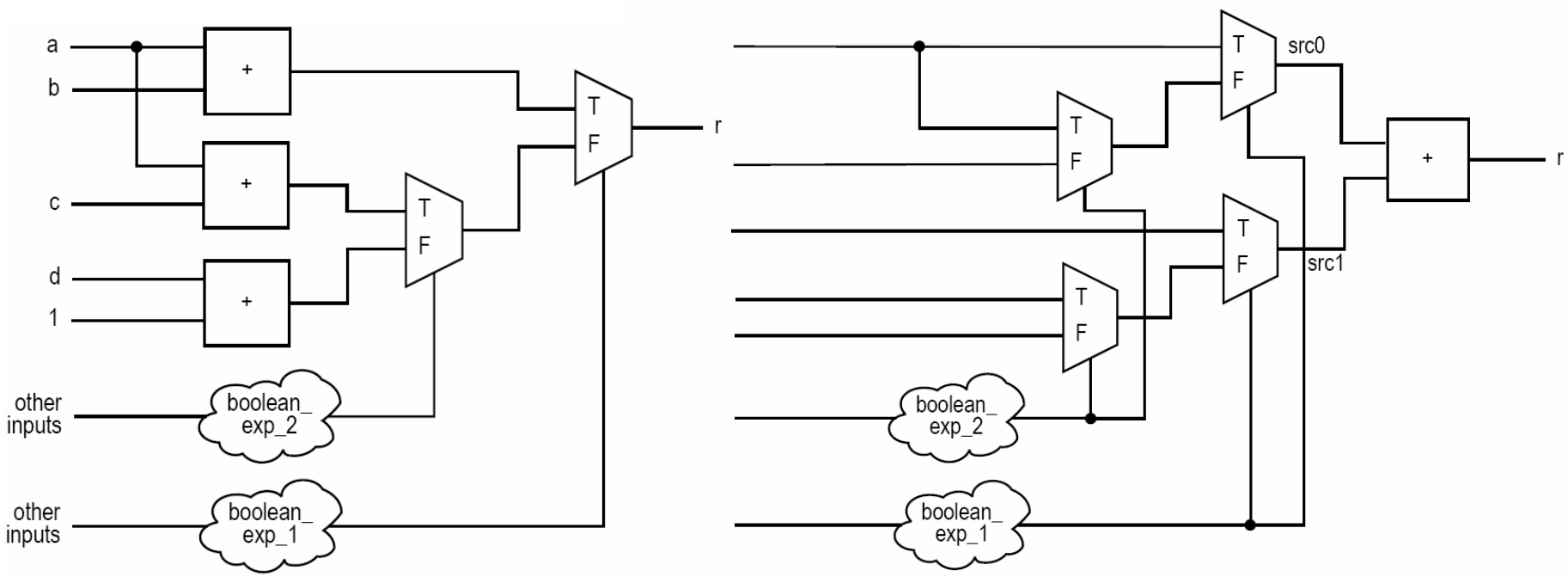
# Example 2

- 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;
```




Area:

2 adders, 1 inc, 2 mux

Area:

1 adder, 4 mux

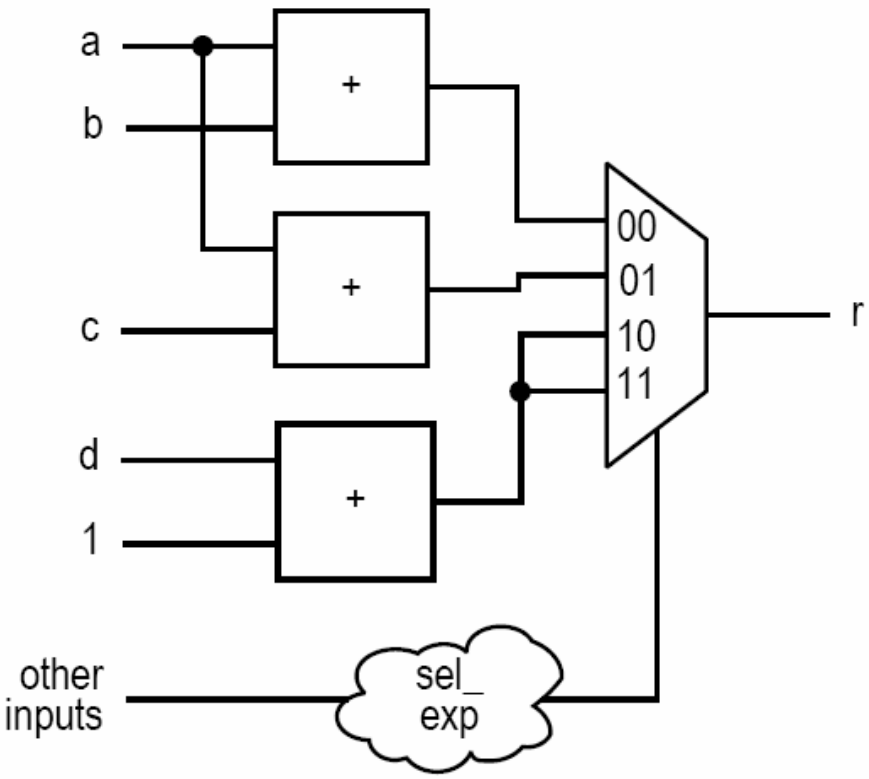
# Example 3

- Original code:

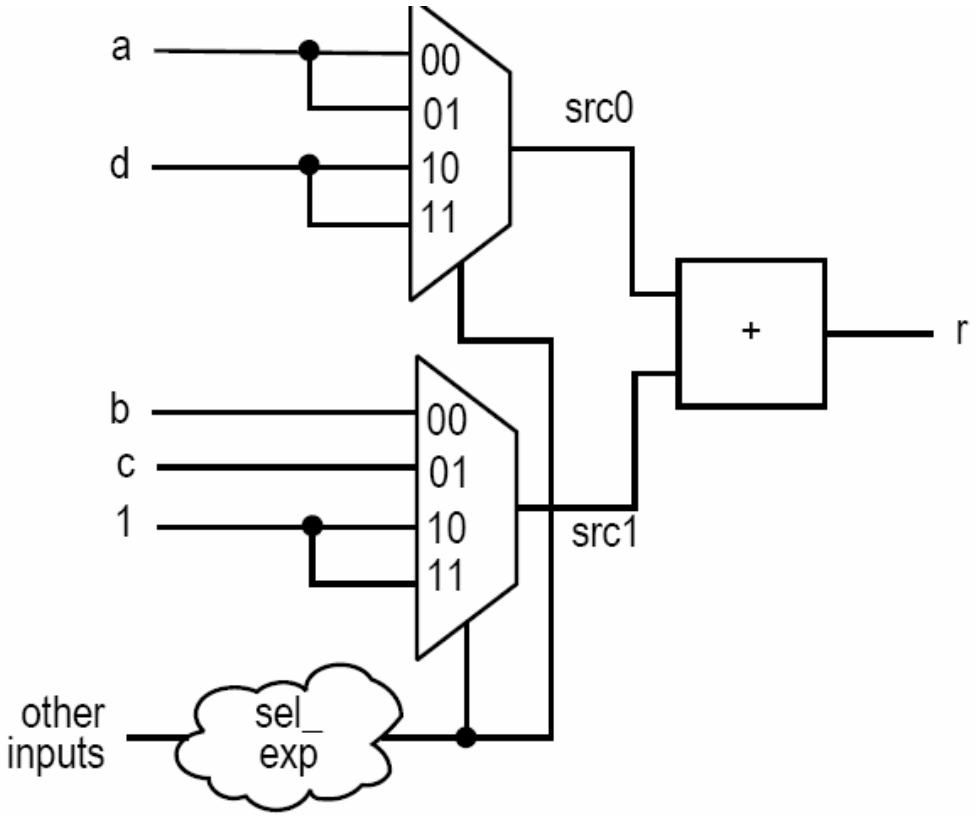
```
with sel select
   r <= a+b when "00",
        a+c when "01",
        d+1 when others;
```

- Revised code:

```
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;
```




(a) Original diagram




(b) Diagram with sharing

Area:

2 adders, 1 inc, 1 mux

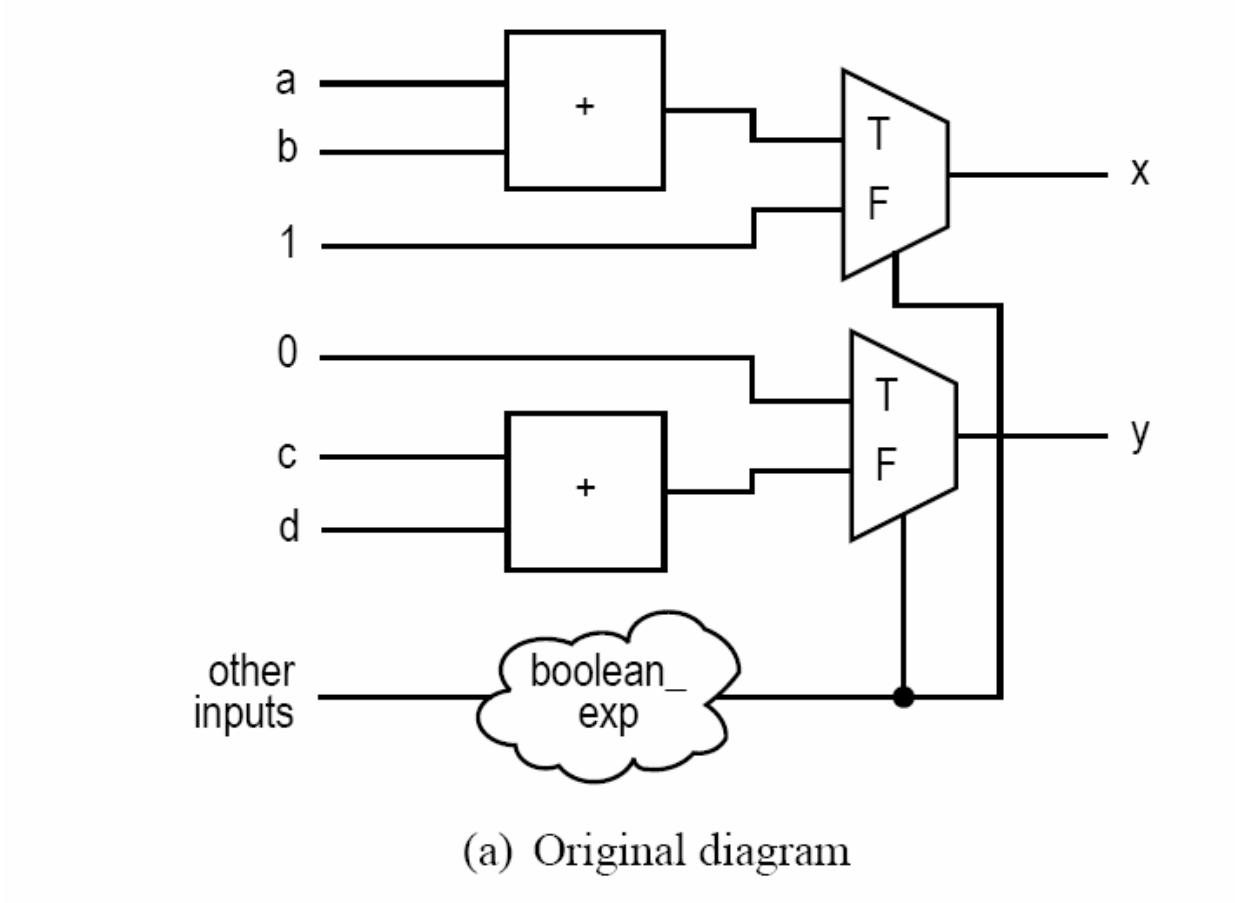
Area:

1 adder, 2 mux

# Example 4

- Original code:

```
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;
```



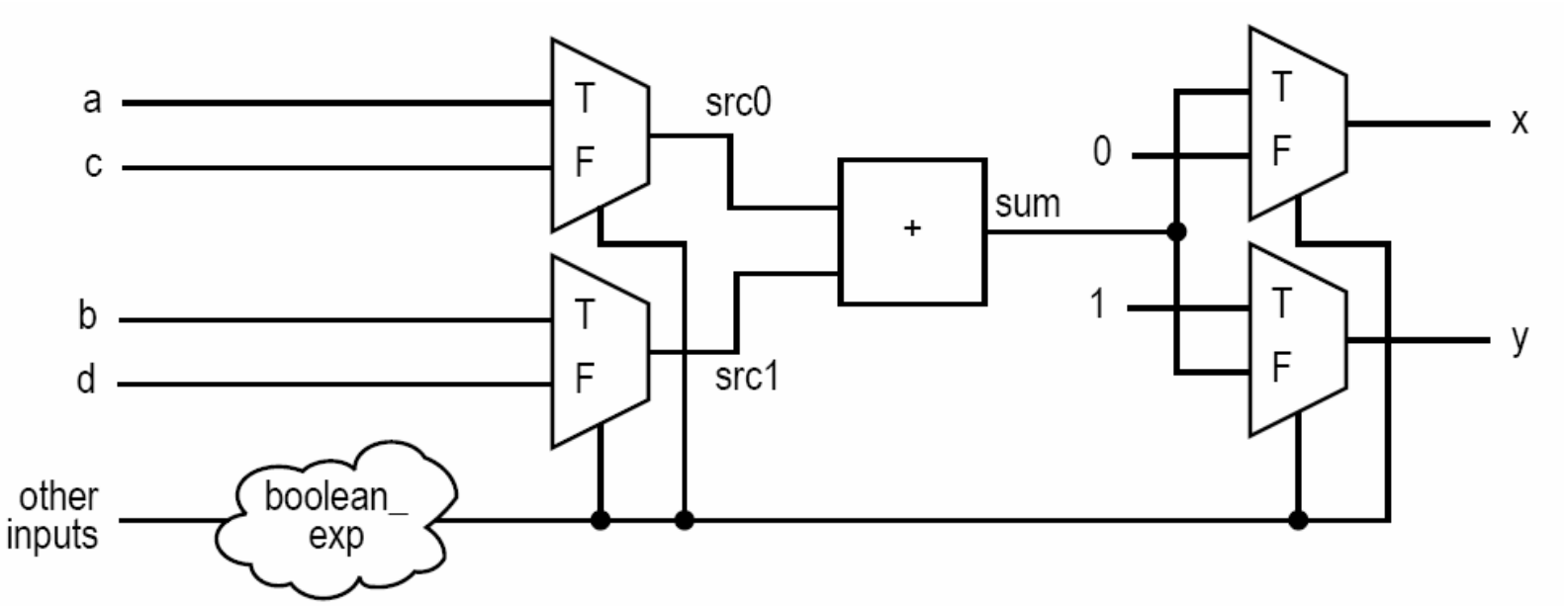

(a) Original diagram

Area:

2 adders, 2 mux

- Revised code:

```
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;
```




- 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 it since it does not know the “meaning” of functions

# e.g., add-sub circuit

```
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
    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;
```

| ctrl | operation |
|---|---|
| 0 | a + b |
| 1 | a - b |

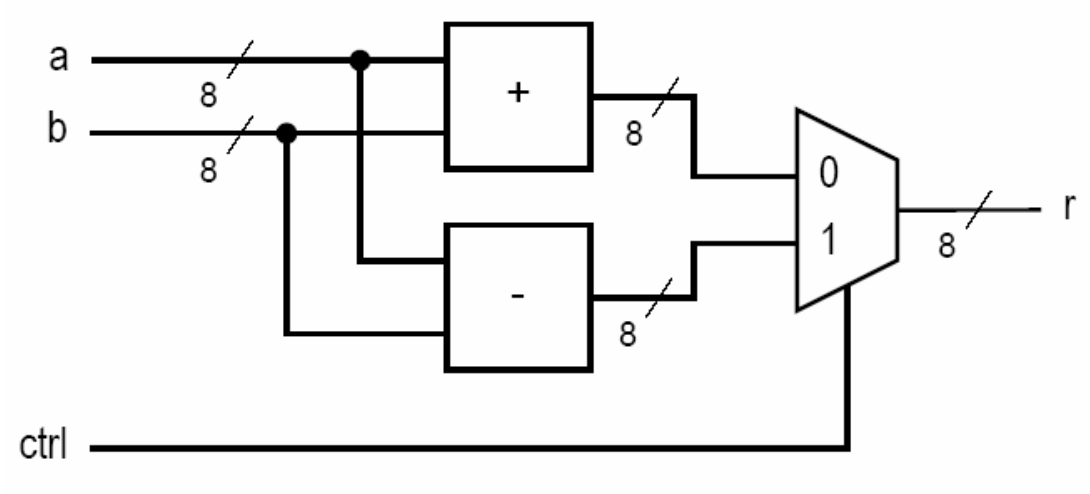
- Observation: a – b can be done by a + b' + 1

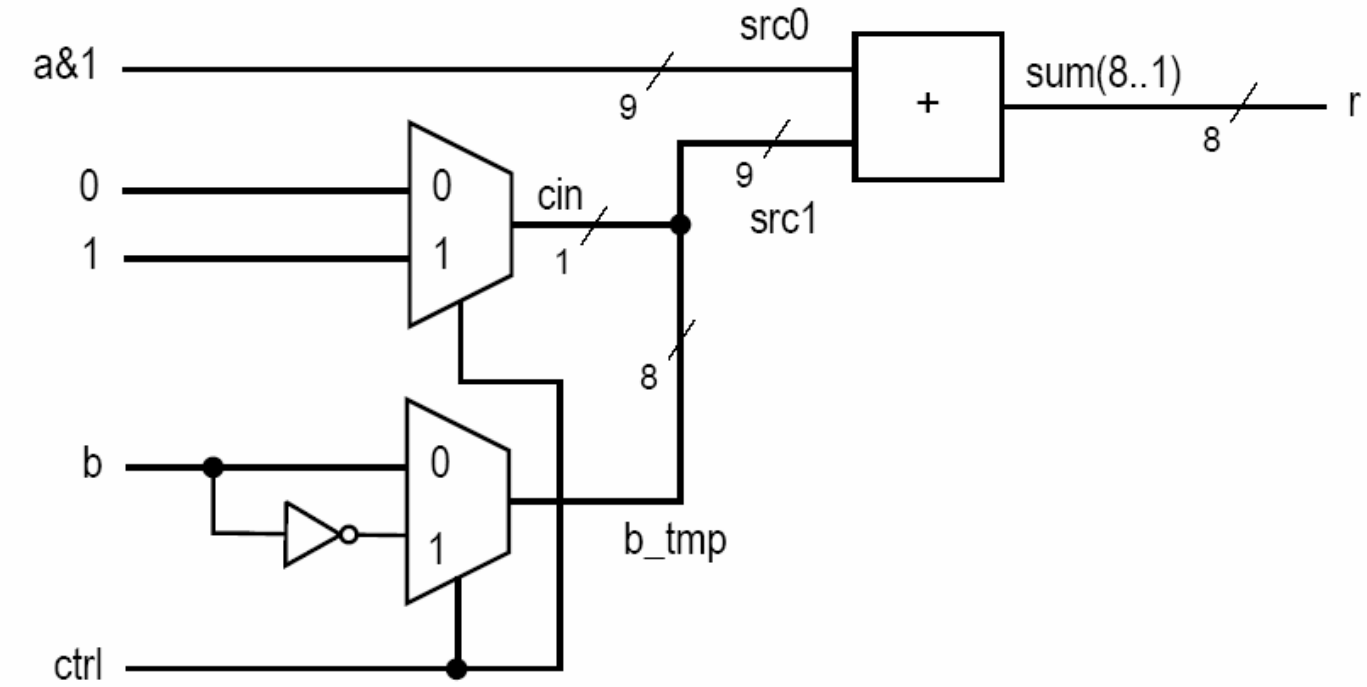
```
architecture shared_arch of addsub is
   signal src0, src1, sum: signed(7 downto 0);
   signal cin: signed(0 downto 0); -- carry-in bit
begin
   src0 <= signed(a);
   src1 <= signed(b) when ctrl='0' else
           signed(not b);
   cin <= "0" when ctrl='0' else
          "1";
   sum <= src0 + src1 + cin;
   r <= std_logic_vector(sum);
end shared_arch;
```

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

$x_7x_6x_5x_4x_3x_2x_1x_0 1$ and $y_7y_6y_5y_4y_3y_2y_1y_0c_{in}$

```
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;
```

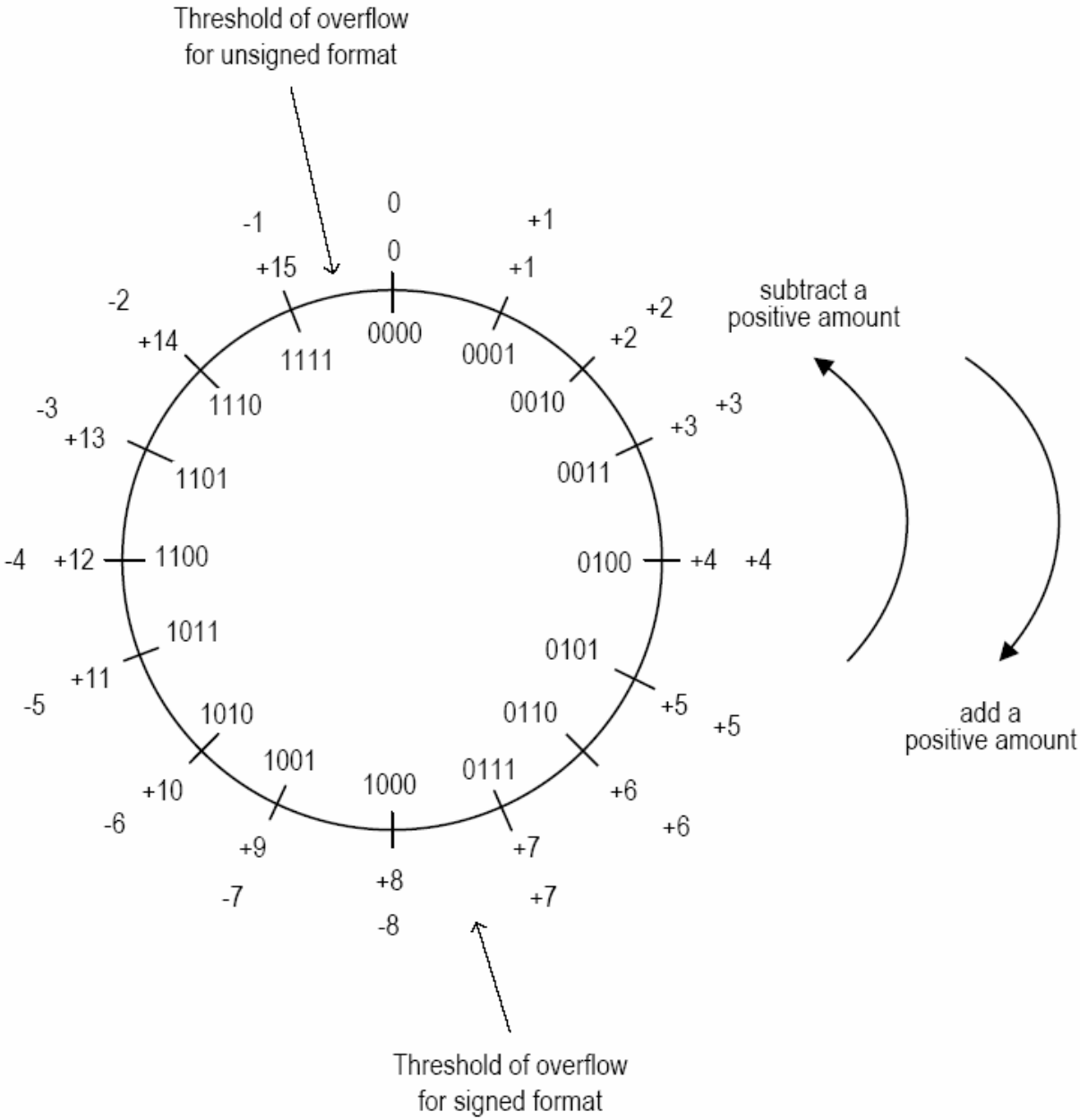

a
b
8
8
+
-
8
8
0
1
8
r
ctrl


a&1
9
src0
+
sum(8..1)
8
r
0
1
0
1
cin
1
9
src1
8
b
b_tmp
ctrl

# e.g., sign-unsigned comparator

```
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) els
                    '0';
   agtb <= agtb_unsigned when (mode='0') else
           agtb_signed;
end direct_arch ;
```

# Binary wheel



- 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

```
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
        not a1_b0;
end shared_arch;
```

# e.g., Full comparator

```
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;
```

```
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;
```

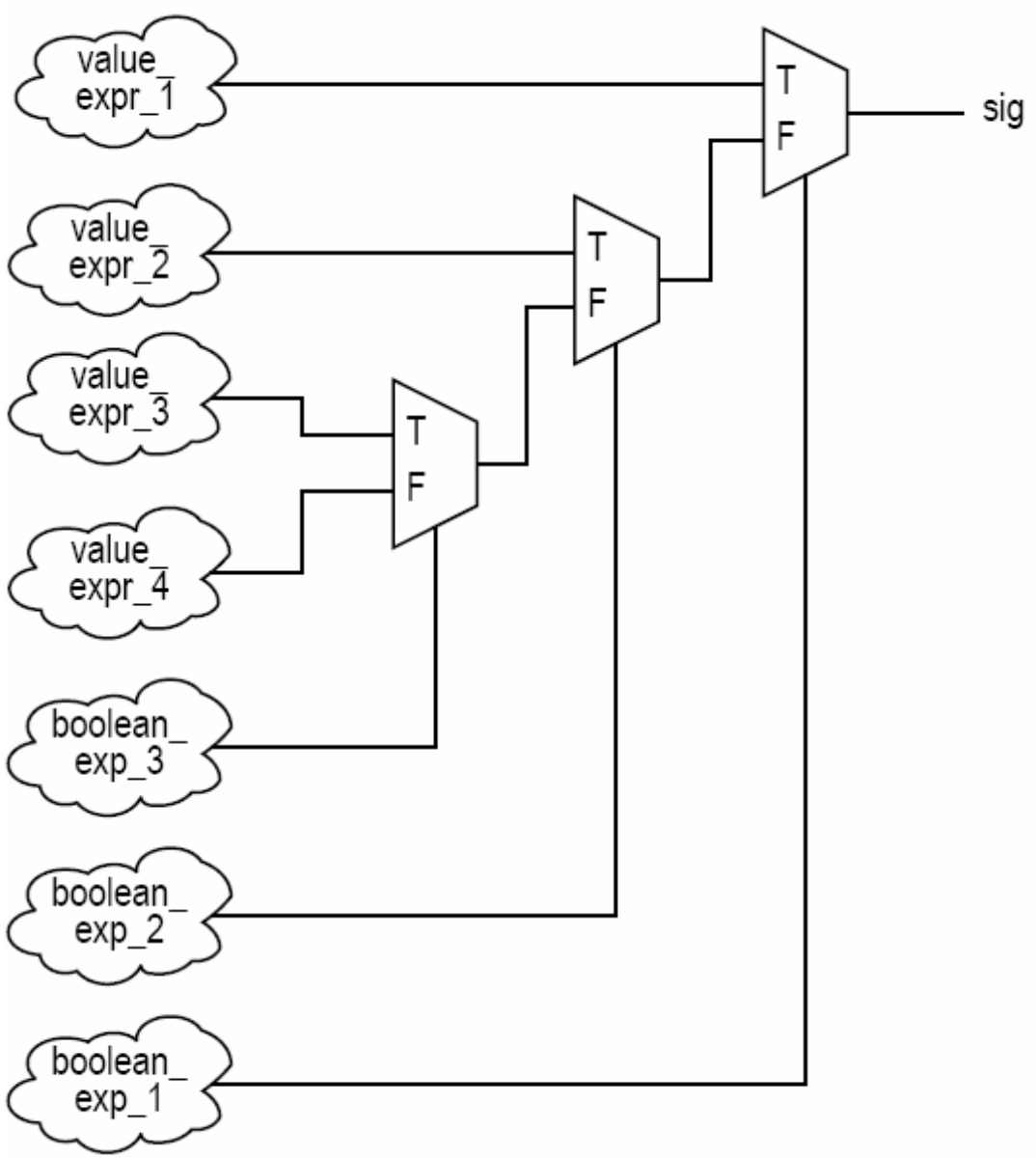
```
architecture share2_arch of comp3 is
    signal eq, lt: std_logic;
begin
    eq <= '1' when a = b else
          '0';
    lt <= '1' when a < b else
          '0';
    aeqb <= eq;
    altb <= lt;
    agtb <= not (eq or lt);
end share2_arch;
```

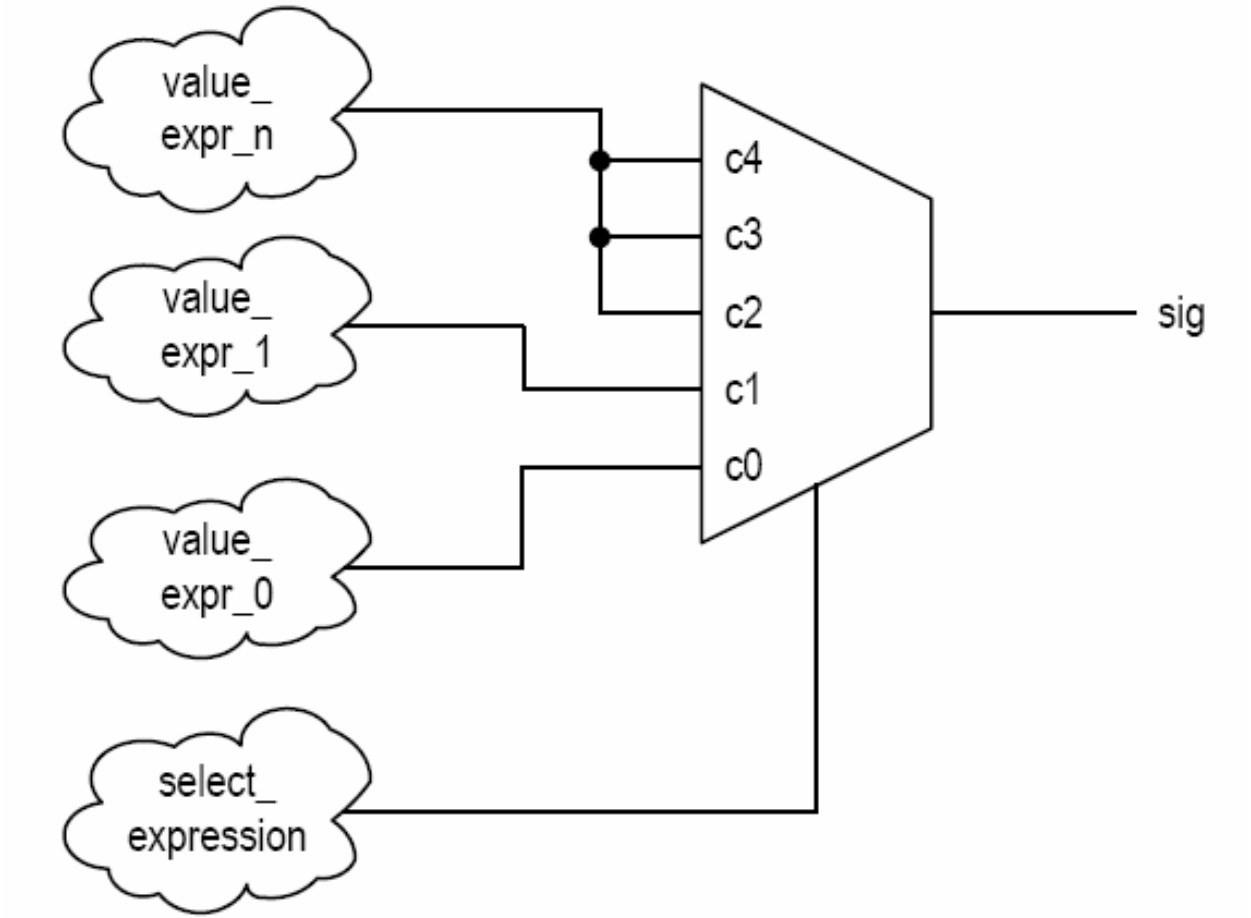
- Read 7.3.3 and 7.3.5

# 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


value_
expr_1
value_
expr_2
value_
expr_3
value_
expr_4
boolean_
exp_3
boolean_
exp_2
boolean_
exp_1
T
F
T
F
T
F
sig


value_
expr_n
value_
expr_1
value_
expr_0
select_
expression
c4
c3
c2
c1
c0
sig

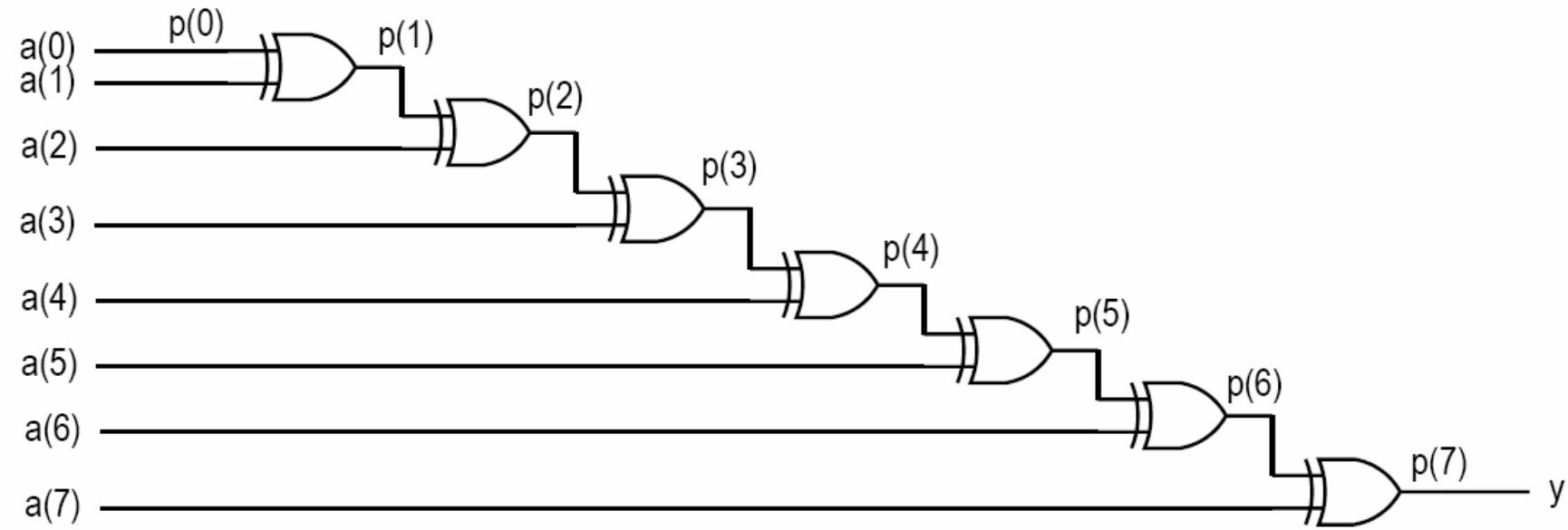
# e.g., Reduced-xor circuit

$$a_7 \oplus a_6 \oplus a_5 \oplus a_4 \oplus a_3 \oplus a_2 \oplus a_1 \oplus a_0$$

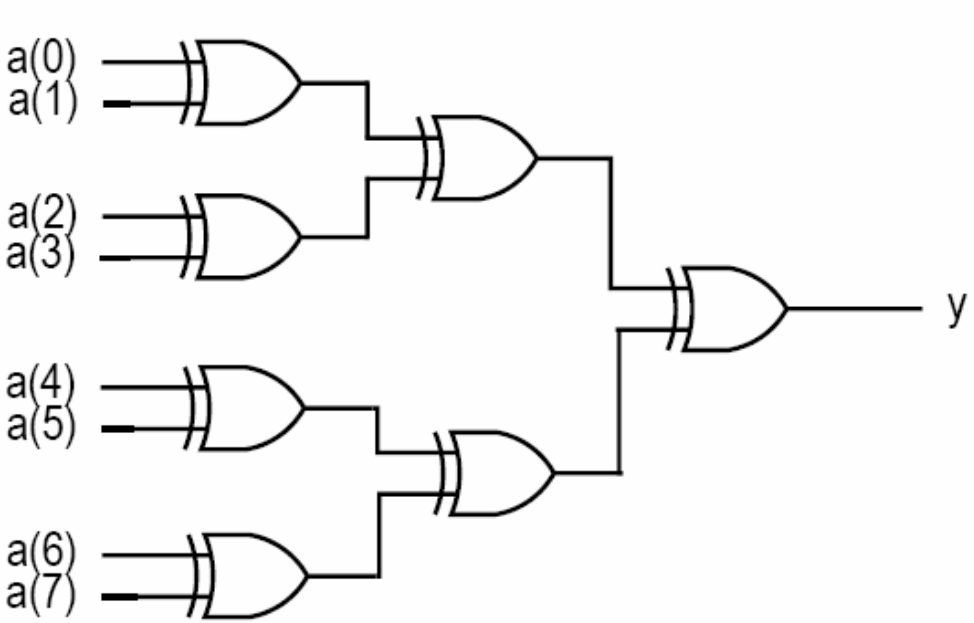
```
library ieee;
use ieee.std_logic_1164.all;

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

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

```
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;
```


a(0)
a(1)
a(2)
a(3)
a(4)
a(5)
a(6)
a(7)
p(0)
p(1)
p(2)
p(3)
p(4)
p(5)
p(6)
p(7)
y

```
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;
```



- 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

# e.g., Reduced-xor-vector circuit

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

- Direct implementation

```
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
   signal p: std_logic_vector(7 downto 0);
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;
```

- Functionality Sharing

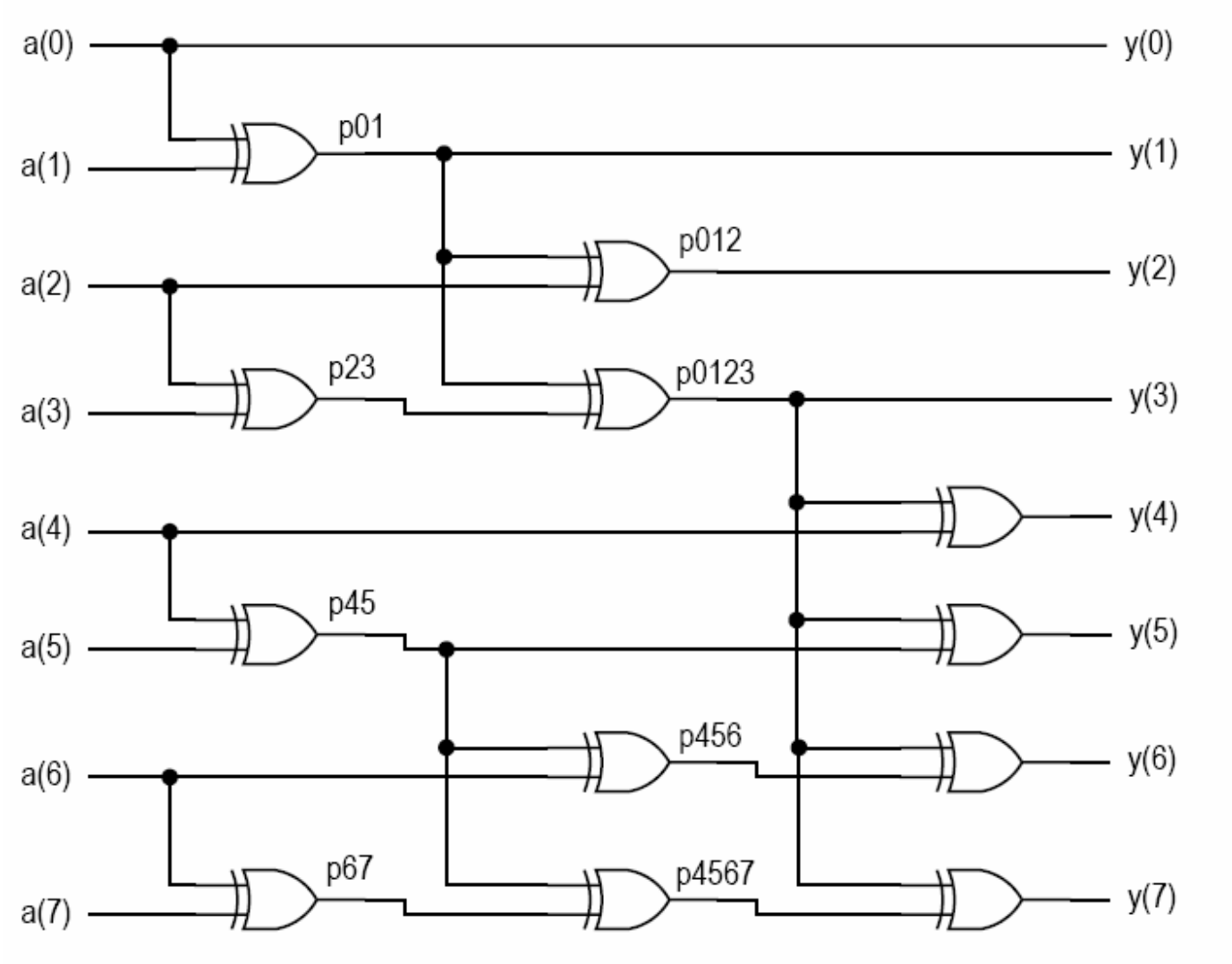
```
architecture shared1_arch of reduced_xor_vector is
   signal p: std_logic_vector(7 downto 0);
begin
   p(0) <= a(0);
   p(1) <= p(0) xor a(1);
   p(2) <= p(1) xor a(2);
   p(3) <= p(2) xor a(3);
   p(4) <= p(3) xor a(4);
   p(5) <= p(4) xor a(5);
   p(6) <= p(5) xor a(6);
   p(7) <= p(6) xor a(7);
   y <= p;
end shared1_arch;
```

```
architecture shared_compact_arch 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') xor a;
   y <= p;
end shared_compact_arch;
```

- Direct tree implementation

```
architecture direct_tree_arch of reduced_xor_vector is
   signal p: std_logic_vector(7 downto 0);
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



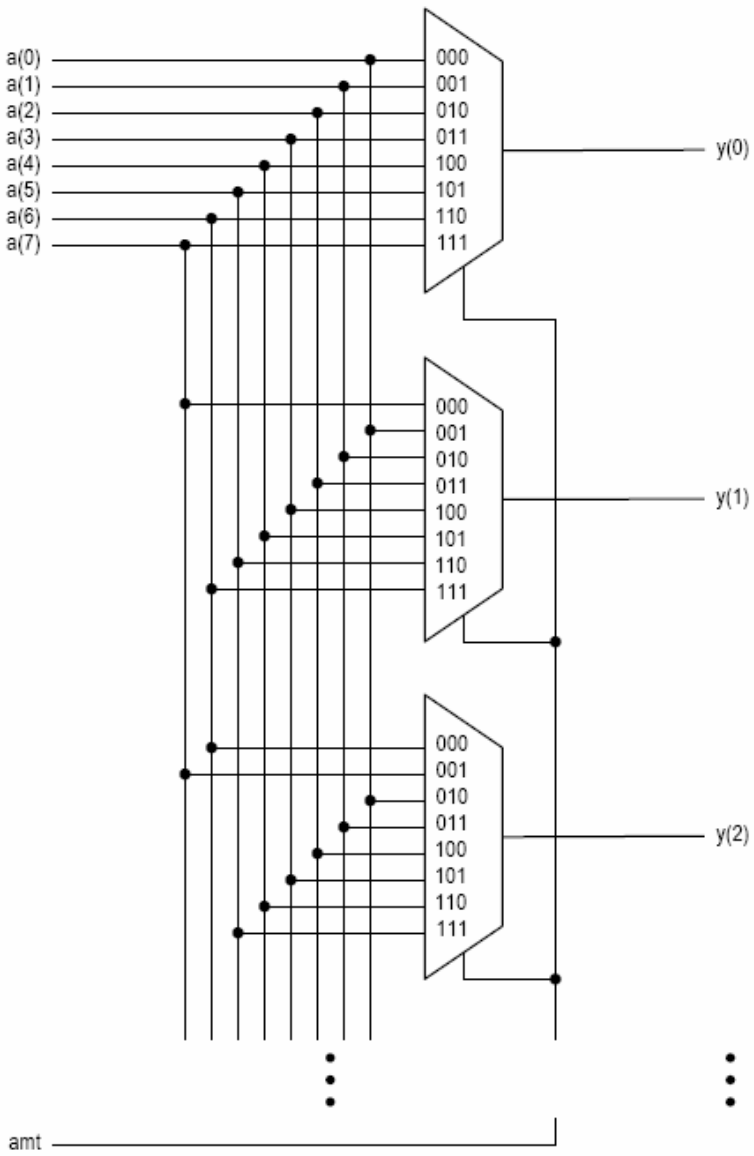
```
architecture optimal_tree_arch of reduced_xor_vector is
   signal p01, p23, p45, p67, p012,
          p0123, p456, p4567: std_logic;
begin
   p01 <= a(0) xor a(1);
   p23 <= a(2) xor a(3);
   p45 <= a(4) xor a(5);
   p67 <= a(6) xor a(7);
   p012 <= p01 xor a(2);
   p0123 <= p01 xor p23;
   p456 <= p45 xor a(6);
   p4567 <= p45 xor p67;
   y(0) <= a(0);
   y(1) <= p01;
   y(2) <= p012;
   y(3) <= p0123;
   y(4) <= p0123 xor a(4);
   y(5) <= p0123 xor p45;
   y(6) <= p0123 xor p456;
   y(7) <= p0123 xor p4567;
end optimal_tree_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: $\log_2 n$
    - Coding: not so easy to modify
  - Parallel-prefix
    - Area: $O(n \log_2 n)$ xor gates
    - Delay: $\log_2 n$
    - Coding: difficult to modify
  - Software is not able to convert cascading chain to parallel-prefix

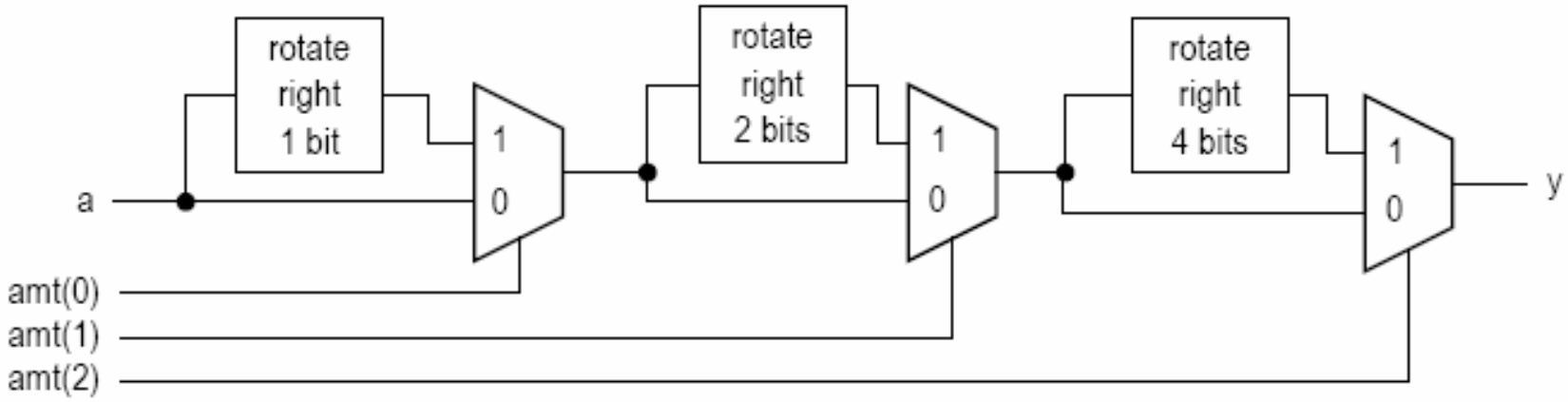
# e.g., Shifter (rotating right)

- Direct implementation

```
entity rotate_right is
   port(
      a: in std_logic_vector(7 downto 0);
      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;
```


a(0)
a(1)
a(2)
a(3)
a(4)
a(5)
a(6)
a(7)
000
001
010
011
100
101
110
111
y(0)
y(1)
y(2)
amt

- Better implementation



```
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;
```

- 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
    - $n*\log_2 n$ 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

| binary code $b_3b_2b_1b_0$ | gray code $g_3g_2g_1g_0$ |
|---|---|
| 0000 | 0000 |
| 0001 | 0001 |
| 0010 | 0011 |
| 0011 | 0010 |
| 0100 | 0110 |
| 0101 | 0111 |
| 0110 | 0101 |
| 0111 | 0100 |
| 1000 | 1100 |
| 1001 | 1101 |
| 1010 | 1111 |
| 1011 | 1110 |
| 1100 | 1010 |
| 1101 | 1011 |
| 1110 | 1001 |
| 1111 | 1000 |

| gray code | incremented gray code |
|---|---|
| 0000 | 0001 |
| 0001 | 0011 |
| 0011 | 0010 |
| 0010 | 0110 |
| 0110 | 0111 |
| 0111 | 0101 |
| 0101 | 0100 |
| 0100 | 1100 |
| 1100 | 1101 |
| 1101 | 1111 |
| 1111 | 1110 |
| 1110 | 1010 |
| 1010 | 1011 |
| 1011 | 1001 |
| 1001 | 1000 |
| 1000 | 0000 |

- Direct implementation

```
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",
                  "0101" when "0111",
                  "0100" when "0101",
                  "1100" when "0100",
                  "1101" when "1100",
                  "1111" when "1101",
                  "1110" when "1111",
                  "1010" when "1110",
                  "1011" when "1010",
                  "1001" when "1011",
                  "1000" when "1001",
                  "0000" when others; -- "1000"
      end table_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 code $b_3 b_2 b_1 b_0$ | gray code $g_3 g_2 g_1 g_0$ |
|---|---|
| 0000 | 0000 |
| 0001 | 0001 |
| 0010 | 0011 |
| 0011 | 0010 |
| 0100 | 0110 |
| 0101 | 0111 |
| 0110 | 0101 |
| 0111 | 0100 |
| 1000 | 1100 |
| 1001 | 1101 |
| 1010 | 1111 |
| 1011 | 1110 |
| 1100 | 1010 |
| 1101 | 1011 |
| 1110 | 1001 |
| 1111 | 1000 |

- binary to gray

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

$$\begin{aligned} g_3 &= b_3 \oplus 0 = b_3 \\ g_2 &= b_2 \oplus b_3 \\ g_1 &= b_1 \oplus b_2 \\ g_0 &= b_0 \oplus b_1 \end{aligned}$$

- gray to binary

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

$$\begin{aligned} b_3 &= g_3 \oplus 0 = g_3 \\ b_2 &= g_2 \oplus b_3 = g_2 \oplus g_3 \\ b_1 &= g_1 \oplus b_2 = g_1 \oplus g_2 \oplus g_3 \\ b_0 &= g_0 \oplus b_1 = g_0 \oplus g_1 \oplus g_2 \oplus g_3 \end{aligned}$$

```
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;
```

# 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

$$overflow = (s_a \cdot s_b \cdot s'_s) + (s'_a \cdot s'_b \cdot s_s)$$

  - Zero
  - Sign (of the addition result)

```
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);
      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
   a_ext <= signed('0' & a & '1');
   b_ext <= signed('0' & b & cin);
   sum_ext <= a_ext + b_ext;
   ovf <= (sign_a and sign_b and (not sign_s)) or
          ((not sign_a) and (not sign_b) and sign_s);
   cout <= sum_ext(9);
   zero <= '1' when (sum_ext(8 downto 1)=0 and ovf='0') else
           '0';
   overflow <= ovf;
   sum <= std_logic_vector(sum_ext(8 downto 1));
end arch;
```

# e.g., simple combinational multiplier

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | $a_3$ | $a_2$ | $a_1$ | $a_0$ | multiplicand |
| $\times$ | | | | | $b_3$ | $b_2$ | $b_1$ | $b_0$ | multiplier |
| | | | | | $a_3b_0$ | $a_2b_0$ | $a_1b_0$ | $a_0b_0$ | |
| | | | | $a_3b_1$ | $a_2b_1$ | $a_1b_1$ | $a_0b_1$ | | |
| | | | $a_3b_2$ | $a_2b_2$ | $a_1b_2$ | $a_0b_2$ | | | |
| $+$ | | $a_3b_3$ | $a_2b_3$ | $a_1b_3$ | $a_0b_3$ | | | | |
| | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ | product |

```
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);
    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";
    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;
```