Lesson 3: Advanced combinatorial design

Multi-module design, functional units and iterative networks

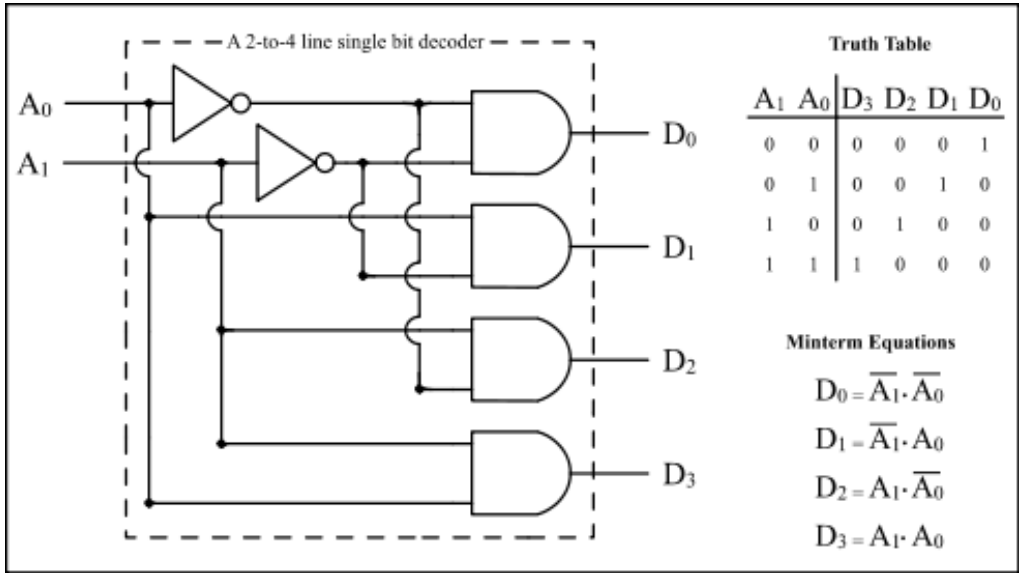
Outline

1. Combinatorial modules
2. Sharing resources
3. Iterative networks
4. Techniques to improve the performance
   1. Tree networks
   2. Anticipation: fast adders
5. Integer arithmetic

Decoder

- Combinatorial circuit with `n` inputs and 2^n outputs. Each output is one of the *minterms* that can be generated with n variables:

$$D_i = \begin{cases} 1 & \text{if } A = i, \text{ where } A = \sum_{j=0}^{n-1} A_j 2^j \text{ for } 0 \le i \le 2^n - 1 \\ 0 & \text{otherwise} \end{cases}$$



Decoder, VHDL coding

```
entity decoder is
    port ( sel: in  std_logic_vector (2 downto 0);
           res: out std_logic_vector (7 downto 0) );
end decoder;
```

```
-- Code of a decoder with fixed length
architecture rtl_1 of decoder is
begin
    res <= "00000001" when sel = "000" else
           "00000010" when sel = "001" else
           "00000100" when sel = "010" else
           "00001000" when sel = "011" else
           "00010000" when sel = "100" else
           "00100000" when sel = "101" else
           "01000000" when sel = "110" else
           "10000000";
end rtl_1;
```

```
-- Vivado does not infer a decoder. It generates logic
-- Generic parameters can be easily used
architecture rtl_2 of decoder is
begin
    process(sel)
    begin
        res <= (others => '0');
        res(to_integer(unsigned(sel))) <= '1';
    end process;
end rtl_2;
```

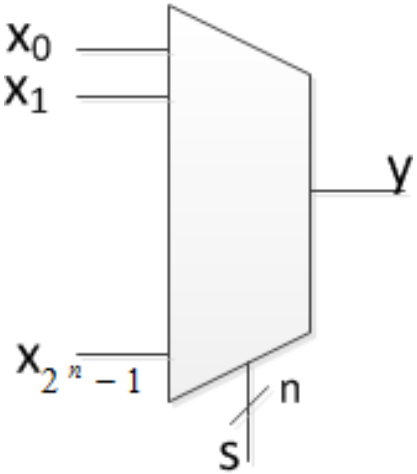
Encoder, VHDL coding

```
-- Vivado synthetizes the priority encoder with logic
entity priority_encoder is
    port ( sel:  in  std_logic_vector (7 downto 0);
           code: out std_logic_vector (2 downto 0) );
end priority_encoder;

architecture rtl of priority_encoder is
begin
    code <= "111" when sel(7) = '1' else
            "110" when sel(6) = '1' else
            "101" when sel(5) = '1' else
            "100" when sel(4) = '1' else
            "011" when sel(3) = '1' else
            "010" when sel(2) = '1' else
            "001" when sel(1) = '1' else
            "000" when sel(0) = '1' else
            "---";
end rtl;
```

Multiplexer, VHDL coding

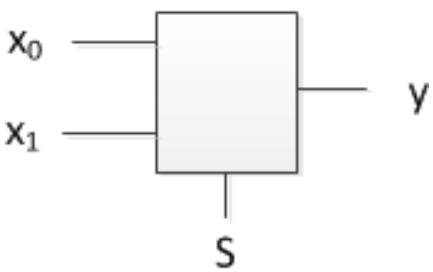
- High-level description

$$y = x_s \quad , \textit{where} \quad s = \sum_{j=0}^{n-1} x_j 2^j$$



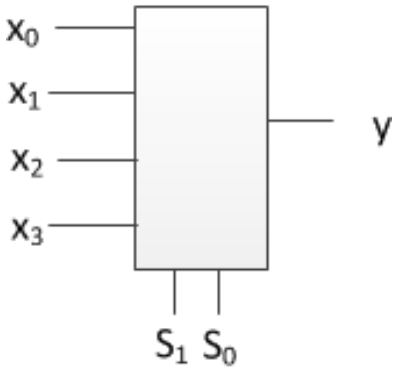
- Implementation

2:1



$$y = x_0 \cdot \bar{s} + x_1 \cdot s$$

4:1



$$y = x_0 \bar{s}_1 \bar{s}_0 + x_1 s_1 \bar{s}_0 + x_2 \bar{s}_1 s_0 + x_3 s_1 s_0$$

Multiplexer, VHDL coding

```
entity MUX8_1 is
    port( sel:  in  std_logic_vector(2 downto 0);
          data: in  std_logic_vector(7 downto 0);
          z:    out std_logic );
end MUX8_1;
```

```
architecture rtl_1 of MUX8_1 is
begin
    with sel select
       z <=  data(0) when "000",
             data(1) when "001",
             data(2) when "010",
             data(3) when "011",
             data(4) when "100",
             data(5) when "101",
             data(6) when "110",
             data(7) when others;
end rtl_1;
```

```
architecture rtl_2 of MUX8_1 is
begin
    Z <= data(to_integer(unsigned(sel)));
end rtl_2;
```

Adder, VHDL coding

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

entity adder is
    generic( n: natural := 8 );
    port( a   : in  std_logic_vector(n-1 downto 0);
          b   : in  std_logic_vector(n-1 downto 0);
          ci  : in  std_logic;
          sum : out std_logic_vector(n-1 downto 0);
          co  : out std_logic );
end adder;

architecture rtl of adder is
    signal sum_u: unsigned(n downto 0);
begin
    sum_u <= unsigned("0" & a) + unsigned("0" & b) + unsigned'("0" & ci);
    sum   <= std_logic_vector(sum_u(n-1 downto 0));
    co    <= sum_u(n);
end rtl;
```

Adder/Subtractor VHDL coding

```
entity adder_sub is
    generic( n: natural := 8 );
    port( a   : in  std_logic_vector(n-1 downto 0);
          b   : in  std_logic_vector(n-1 downto 0);
          op  : in  std_logic;
          res : out std_logic_vector(n-1 downto 0) );
end adder_sub;
architecture rtl of adder_sub is
    component adder
        generic( n: natural := 8 );
        port( a   : in  std_logic_vector(n-1 downto 0);
              b   : in  std_logic_vector(n-1 downto 0);
              ci  : in  std_logic;
              sum : out std_logic_vector(n-1 downto 0);
              co  : out std_logic );
    end component;
    signal op_v : std_logic_vector(n-1 downto 0);
begin
    op_v <= b xor (b'range => op);
    U_ADD: adder
        generic map(n)
        port map(
            a   => a,
            b   => op_v,
            ci  => op,
            sum => res,
            co  => open );
end rtl;
```

Comparator VHDL coding

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

entity comparator is
    generic( n: natural := 8 );
    port( a   : in  std_logic_vector(n-1 downto 0);
          b   : in  std_logic_vector(n-1 downto 0);
          cmp : out std_logic );
end comparator;

architecture rtl of comparator is
begin
    cmp <= '1' when a >= b else '0';
end rtl;
```

Comparator VHDL coding

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

entity comparator is
    generic( n: natural := 8 );
    port( a   : in  std_logic_vector(n-1 downto 0);
          b   : in  std_logic_vector(n-1 downto 0);
          cmp : out std_logic );
end comparator;

architecture rtl of comparator is
begin
    cmp <= '1' when a >= b else '0';
end rtl;
```

```
[un]signed(a) >= [un]signed(b)
```

Outline

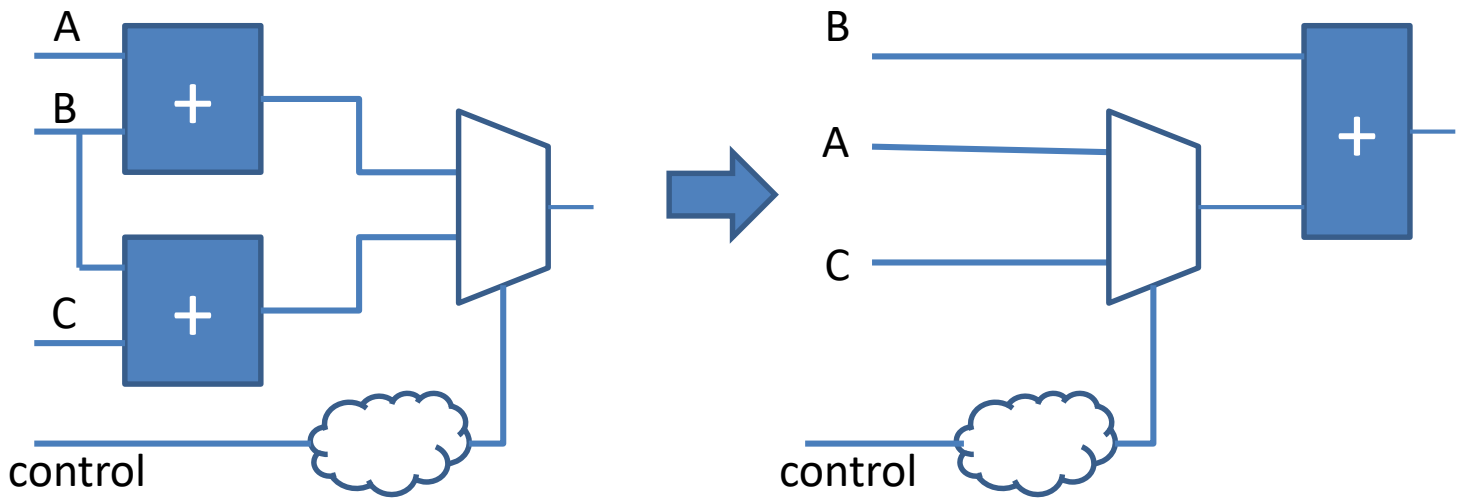
1. Combinatorial modules
2. Sharing resources
3. Iterative networks
4. Techniques to improve the performance
   1. Tree networks
   2. Anticipation: fast adders
5. Integer arithmetic

Basic idea

- Study the structure of the design to reduce the amount of HW resources, sharing logic between operations
- Main ideas:
  - Reuse of operators
  - Avoid multiplexing operations
  - Identify non-concurrent operations

Example I

- Do no multiplex operations



Vivado is able to identify some of these situations and optimize

Example I

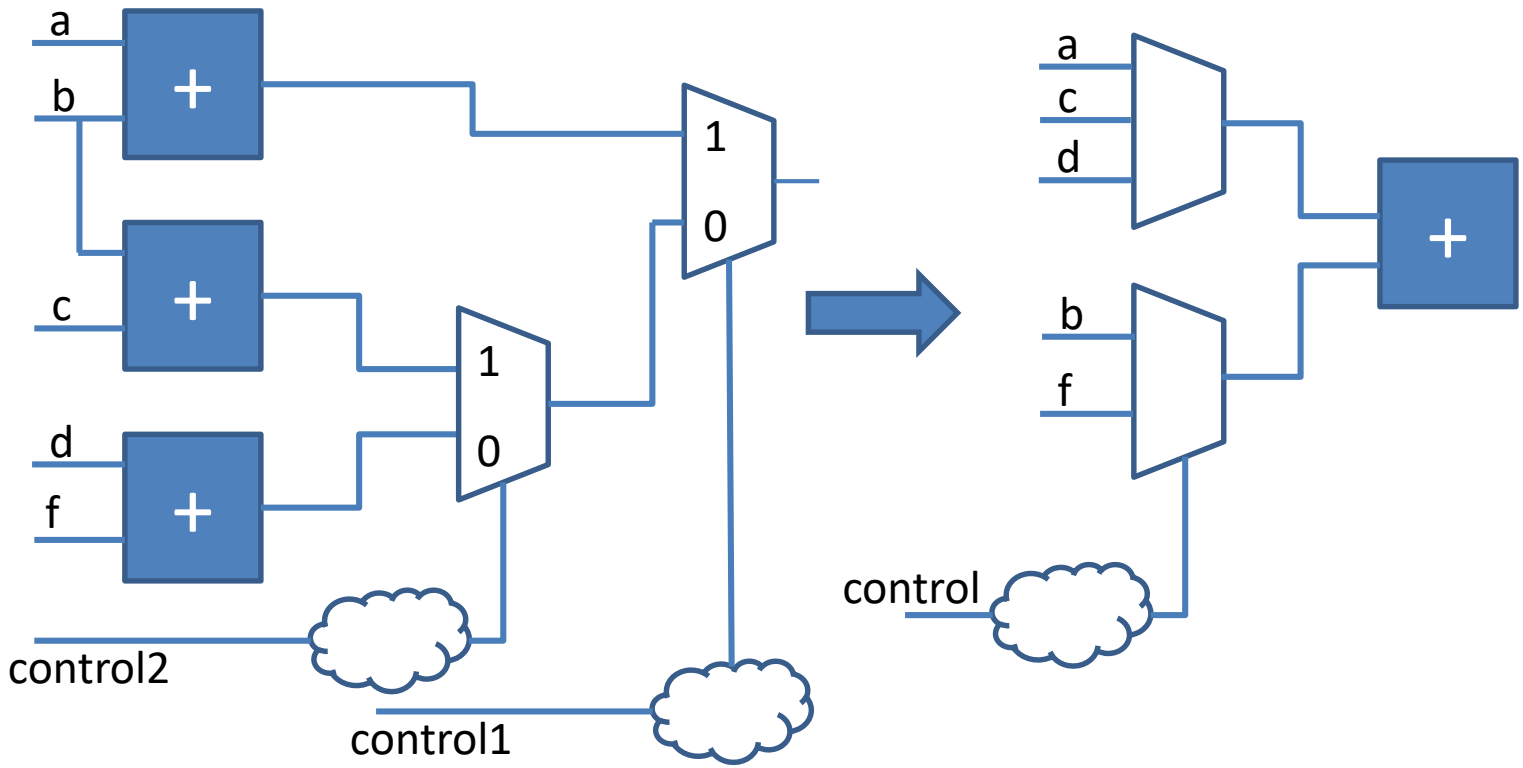
```
entity dsp is
  generic( n: natural := 8 );
  port( a, b, c : in  std_logic_vector(n-1 downto 0);
        ctrl    : in std_logic;
        res     : out std_logic_vector(n-1 downto 0) );
end dsp;

architecture rtl_dsp of dsp is
  signal res_u : unsigned (n-1 downto 0) ;
begin
  p_op : process (a,b,c,ctrl)
  begin
    case ctrl is
      when '1'    => res_u <= unsigned(a) + unsigned(b);
      when others => res_u <= unsigned(b) + unsigned(c);
    end case;
  end process p_op;
  res <= std_logic_vector(res_u);
end rtl_dsp;
```

Logs in Vivado - INFO: [Synth 8-5818] HDL ADVISOR - The operator resource <adder> is shared.

```
Detailed RTL Component Info :
+---Adders :
             2 Input      8 Bit       Adders := 1
+---Muxes :
             2 Input      8 Bit        Muxes := 1
```

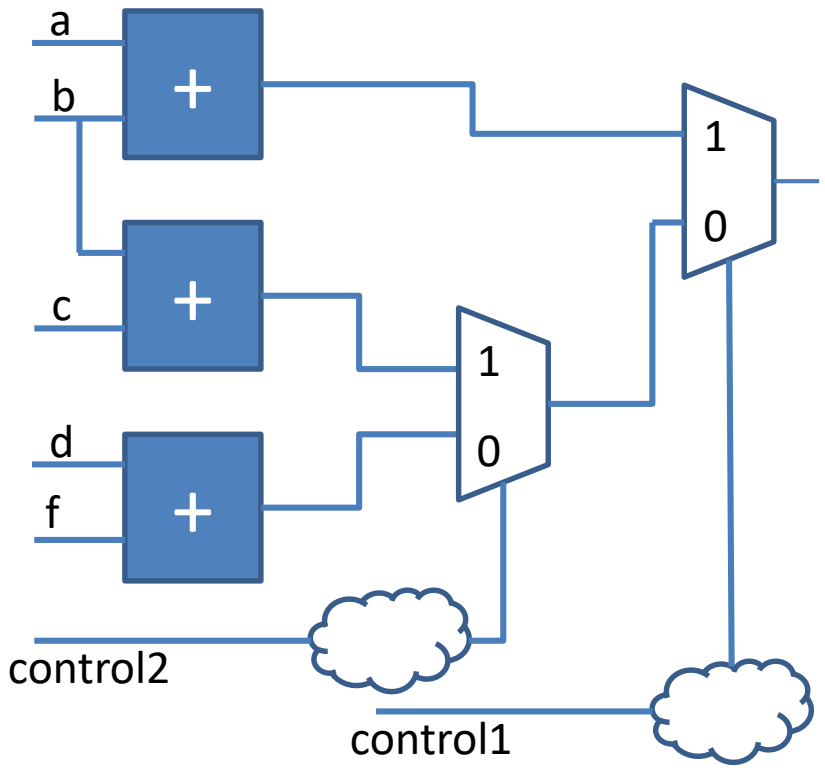
Example II

- How can this design be implemented using just one adder?



Example II

- Yes, it can!




```
p_op : process (a,b,c,d,f,ctrl1,ctrl2)
begin
case ctrl1 is
  when '1' =>
    res_u<=unsigned(a)+unsigned(b);
  when others =>
    case ctrl2 is
      when '1' =>
        res_u<=unsigned(b)+unsigned(c);
      when others =>
        res_u<=unsigned(d)+unsigned(f);
    end case;
  end case;
end process p_op;
```

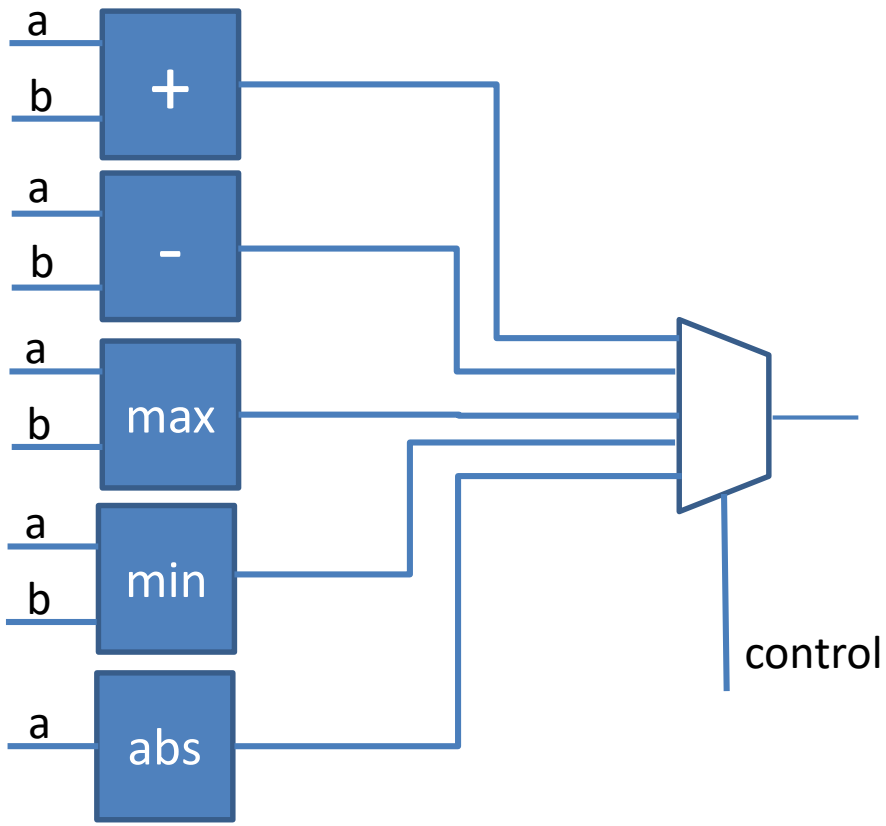
Logs in Vivado - INFO: [Synth 8-5818] HDL ADVISOR - The operator resource <adder> is shared (this log appears twice)

Share Functional Units (FUs)

- Group simple FUs into more complex FUs:
  - Multifunction Unit
- When?
  - The Multifunctional Unit plus interconnection cost is less than the cost of the simple UFs
  - The operations are NOT simultaneous
- How? By using a compatibility graph

Example III

- Implement a Multifunctional Unit : add, subtract, maximum, minimum and absolute value



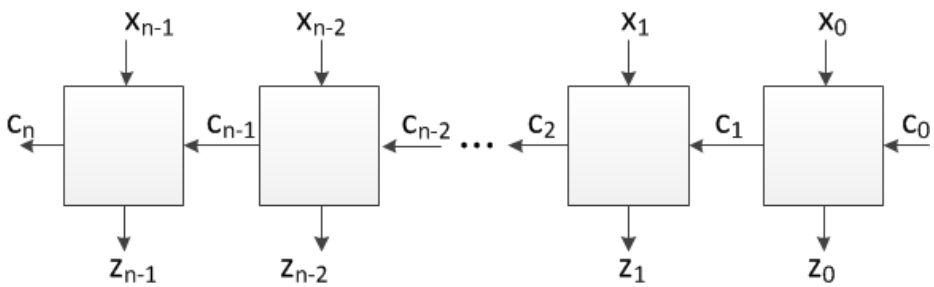
Outline

1. Combinatorial modules
2. Sharing resources
3. Iterative networks
4. Techniques to improve the performance
   1. Tree networks
   2. Anticipation: fast adders
5. Integer arithmetic

Iterative networks: Definition

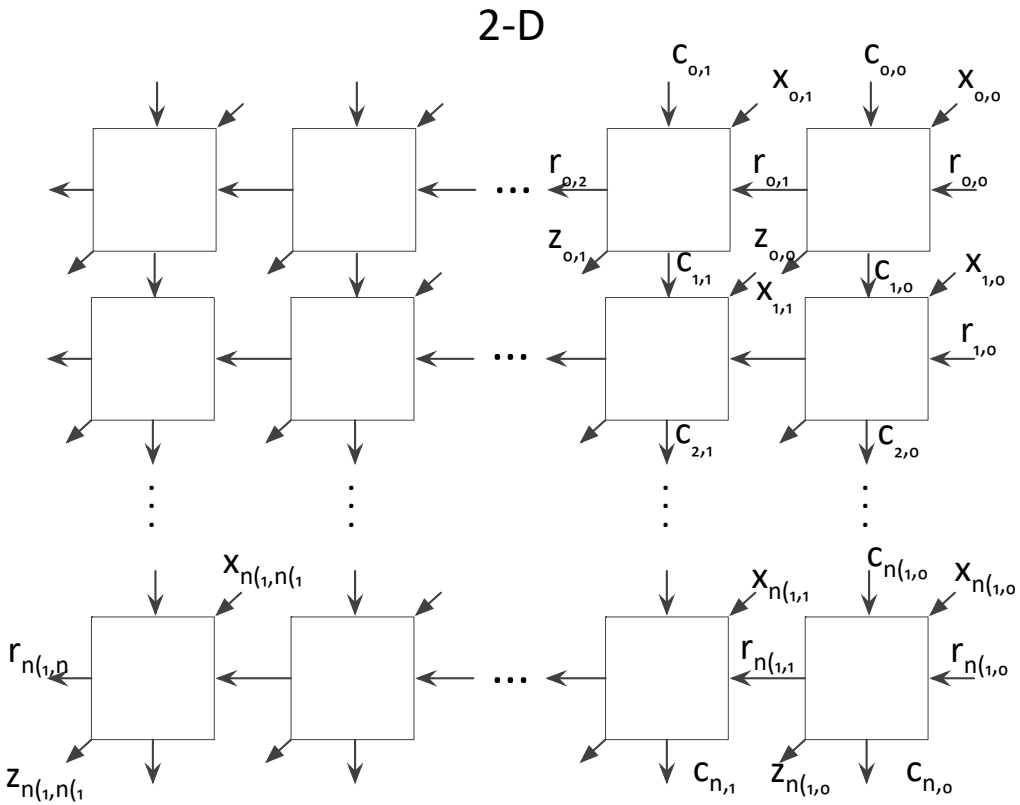
- Set of identical modules.
- Each module connected exclusively with its neighbors

1-D



Iterative networks: Definition

- Set of identical modules.
- Each module connected exclusively with its neighbors



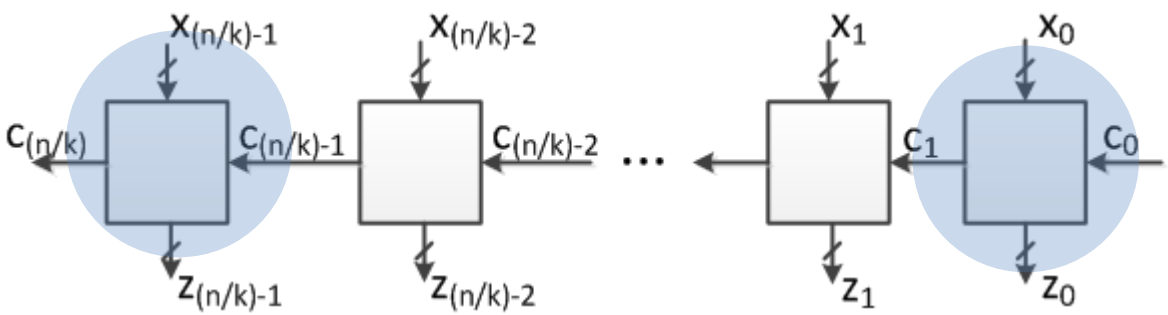
Iterative networks: Definition

- A 1-D iterative network of order *k* is an implementation of a *n-variable* function. *k* is the number of entries that are processed by a single cell.
- They feature (*n/k*) identical cells, that produce *n* outputs, with:
  - External inputs, x_i, and internal ones, c_i
  - External outputs, z_i, and internal ones, c_{i+1}

$$c_{i+1} = G(x_i, c_i)$$

$$z_i = F(x_i, c_i)$$

- The boundary cells can be simplified if the boundary conditions are taken into account
  - Boundary condition 1: Value of c_0
  - Boundary condition 2: Are the outputs $c_{(n/k)}$ and/or $z_{(n/k)-1}$ part of my final output?



Design

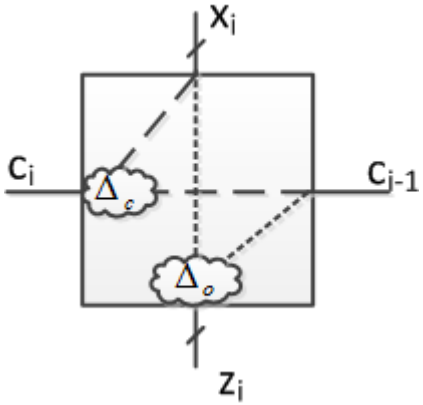
- Procedure for design
  1. Determine a good value for *k* (number of entries that are processed by a single cell)
     - Trade-off between the complexity of the cells (number of entries) and the number of cells (total delay).
  2. Determine the values that the intermediate modules must transmit (internal outputs) and the last output value of the last module.
  3. High-level description of the intermediate blocks.
  4. Implementation of the logical functions of the blocks.
  5. Simplify the boundary cells (previous slide).

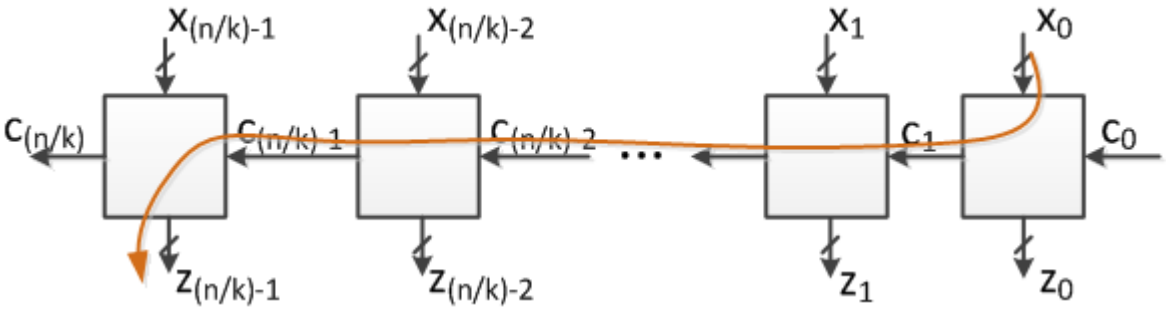
Timing

- Delay (D)
  - It depends on the delay of the external outputs of each cell, D_o
  - It depends on the delay of the intermediate outputs of each cell, D_c

$$D = \left(\frac{n}{k} - 1\right) D_c + \max\left(D_o, D_c\right)$$

As *k* increases, the number of cells decrease, but the delay of each individual cell may be higher





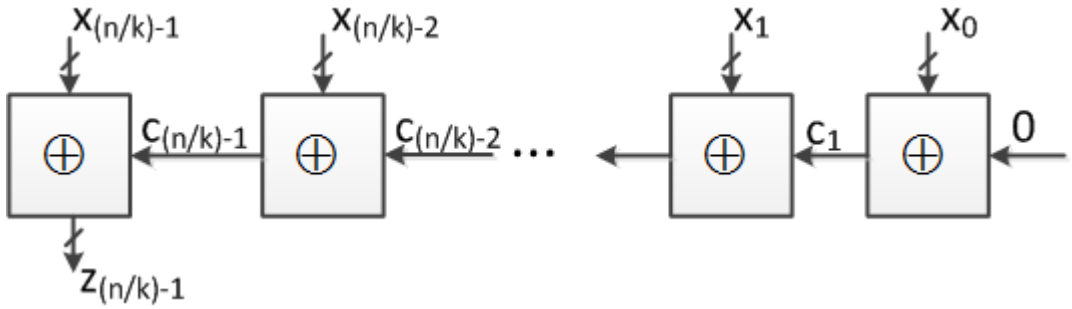
Example 1

- Example: Parity of a number of *n* bits

$$z = x_0 \oplus x_1 \oplus x_2 \oplus \cdots \oplus x_n$$

- Iterative network of grade *k*=1

$$\begin{aligned} G: \quad c_{i+1} &= x_i \oplus c_i \\ F: \quad z &= x_{n-1} \oplus c_{n-1} \end{aligned}$$



Example 2

- Comparator of two unsigned integers
  - *a* and *b* are n-bit vectors.
  - Grade of the iterative network, *k*=2. External input i^{th} module: a_i, b_i
  - Which information does the cell *i* transfer to the cell *i+1*?
    - Which number is greater until position *i*

$$c_{i+1} = \begin{cases} g & if\ a > b \text{ until position } i \\ e & if\ a = b \text{ until position } i \\ s & if\ a < b \text{ until position } i \end{cases}$$

Example 2

- Comparator of two unsigned integers

c_i \ $a_i b_i$	00	01	11	10
g	g	s	g	g
e	e	s	e	g
s	s	s	s	g

Codification 1
"one-hot"

$$g = 100$$
$$e = 010$$
$$s = 001$$

$$c_{i+1}^0 = c_i^0\left(a_i b_i + \bar{a}_i \bar{b}_i\right) + a_i \bar{b}_i$$
$$c_{i+1}^1 = c_i^1\left(a_i b_i + \bar{a}_i \bar{b}_i\right)$$
$$c_{i+1}^2 = c_i^2\left(a_i b_i + \bar{a}_i \bar{b}_i\right) + \bar{a}_i b_i$$

Codification 2

$$g = 10$$
$$e = 00$$
$$s = 01$$

$$c_{i+1}^0 = c_i^0\left(a_i + \bar{b}_i\right) + a_i \bar{b}_i$$
$$c_{i+1}^1 = c_i^1\left(\bar{a}_i + \bar{b}_i\right) + \bar{a}_i b_i$$

$$z = c_n$$

Examples 3 and 4

- Example 3:
  - Design an iterative network for priority resolution with *n* inputs (X_{n-1}, X_{n-2}, …, X_0) and *n* outputs (Z_{n-1}, Z_{n-2}, …, Z_0) . The output Z_i=1 if X_i=1 and X_j=0 $\forall j>i$.

- Example 4:
  - Design an iterative network for a system with *n* inputs (X_{n-1},…X_0) and an output Z that returns '1' if and only if $\exists$ i $0 \leq i \leq n$ such that X_i=1, X_{i+1}=0 and X_{i+2}=1.

Example 3 in VHDL (I)

- Example 3: Design an iterative network for priority resolution with n inputs (X_{n-1}, X_{n-2}, …, X_0) and n outputs (Z_{n-1}, Z_{n-2}, …, Z_0) . The output Z_i=1 if X_i=1 and X_j=0 $\forall$j>i.

Code of the basic cell

```
entity cell is
  port (x, c_in : in std_logic;
        c_out, z: out std_logic);
end cell;

architecture arch of cell is
begin
  c_out <= x or c_in;
  z     <= x and (not(c_in));
end arch;
```

Example 3 in VHDL (II)

- We need to generate an array with a loop to replicate the cell *N* times

```
gen1: for i in 0 to n-1 generate
   u: cell port map(x(i),c(i+1),c(i),z(i));
end generate gen1;
```

In this case, cells interchange information from LEFT to RIGHT

- In addition, we have to generate the boundary conditions for the initial cell:

```
c(n) <= boundary_value;
```

Example 3 in VHDL (III)

Code for the network

```
entity network is
  generic (n: natural := 4);
  port ( x: in  std_logic_vector(n-1 downto 0);
         z: out std_logic_vector(n-1 downto 0) );
end network ;

architecture arch of network is
   component cell
        port( x, c_in : in  std_logic;
              c_out, z: out std_logic );
   end component;
   signal c: std_logic_vector(n downto 0); --internal inputs/outputs
begin
   gen1: for i in 0 to n-1 generate
      u: cell port map(x(i),c(i+1),c(i),z(i));
   end generate gen1;
   c(n)<='0';                                --boundary condition
end arch;
```

Example 4 in VHDL (I)

- Design an iterative network for a system with *n* inputs ($X_{n-1},\ldots X_0$) and an output Z that returns '1' if and only if $\exists$ i $0 \leq i \leq n$ such that $X_i=1$, $X_{i+1}=0$ and $X_{i+2}=1$.

Package definitions

```
package definitions is
  constant g_width_data: natural:=8;
  type t_patt is (no_patt, first_bit, second_bit,patt_rec);
  type t_patt_vec is array (g_width_data downto 0) of t_patt;
end package definitions;
```

Cell code

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

entity cell is
  port ( x     : in  std_logic;
         c_in  : in  t_patt;
         c_out : out t_patt );
end cell;
```

Example 4 in VHDL (II)

```
architecture rtl of cell is
begin
  p_patt: process(c_in,x)
  begin
   case c_in is
     when no_patt=>
       if x='0' then
         c_out<=no_patt;
       else
         c_out<=first_bit;
       end if;
    when first_bit=>
       if x='0' then
         c_out<=second_bit;
       else
         c_out<=no_patt;
       end if;
...
```

```
...
       when second_bit=>
      if x='0' then
        c_out<=no_patt;
      else
        c_out<=patt_rec;
      end if;
    when patt_rec=>
      c_out<=patt_rec;
   end case;
  end process;
end rtl;
```

Example 4 in VHDL (III)

Network code

```
entity network is
   generic (g_width_data: natural:=8);
   Port(x : in  STD_LOGIC_VECTOR(g_width_data-1 downto 0);
        z : out STD_LOGIC);
end network;

architecture arch_net of network is
  component cell is
    port(
      x     : in  std_logic;
      c_in  : in  t_patt;
      c_out : out t_patt
    );
  end component;
  signal pattern: t_patt_vec;
begin
```

```
 gen_cells:
    for i in 0 to g_width_data-1 generate
      i_cell: cell port map (
        x(i),pattern(i),pattern(i+1) );
    end generate gen_cells;

  --boundary condition
  pattern(0)<=no_patt;

  --Output
  z <= '1' when pattern(g_width_data)=patt_rec
    else '0';

end arch_net;
```

Outline

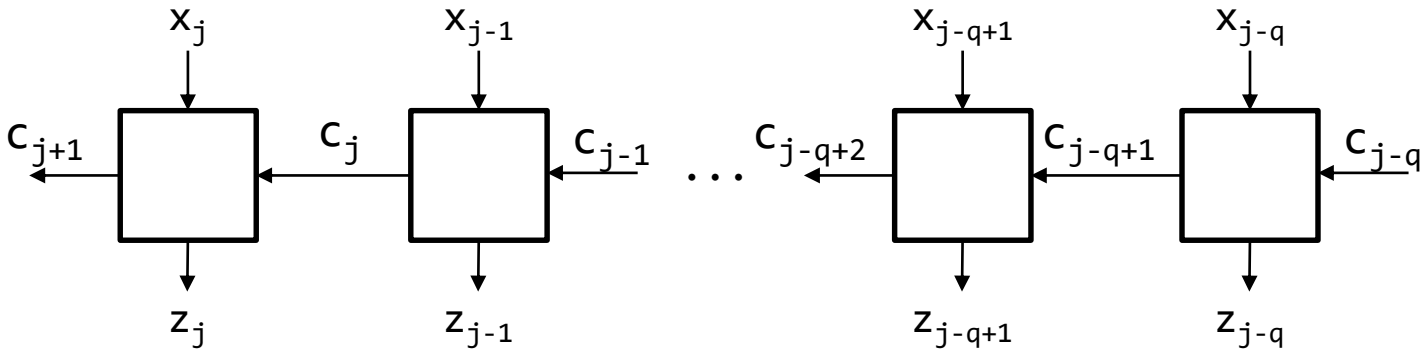
1. Combinatorial modules
2. Sharing resources
3. Iterative networks
4. Techniques to improve the performance
   1. Tree networks
   2. Anticipation: fast adders
5. Integer arithmetic

Techniques to improve the performance

- When designing iterative networks with a very high number of cells (for instance, when the input vector has 32 bits or more), the delay of said network is very high since the critical path has to traverse many cells.
- This happens because the output of the last cell of the iterative network cannot generate its output until it has received its intermediate signal.
  - That signal will have traversed the remaining `n-1` cells of the design.
  - Hence, the total delay will very high if `n` is very high as well

Anticipation networks

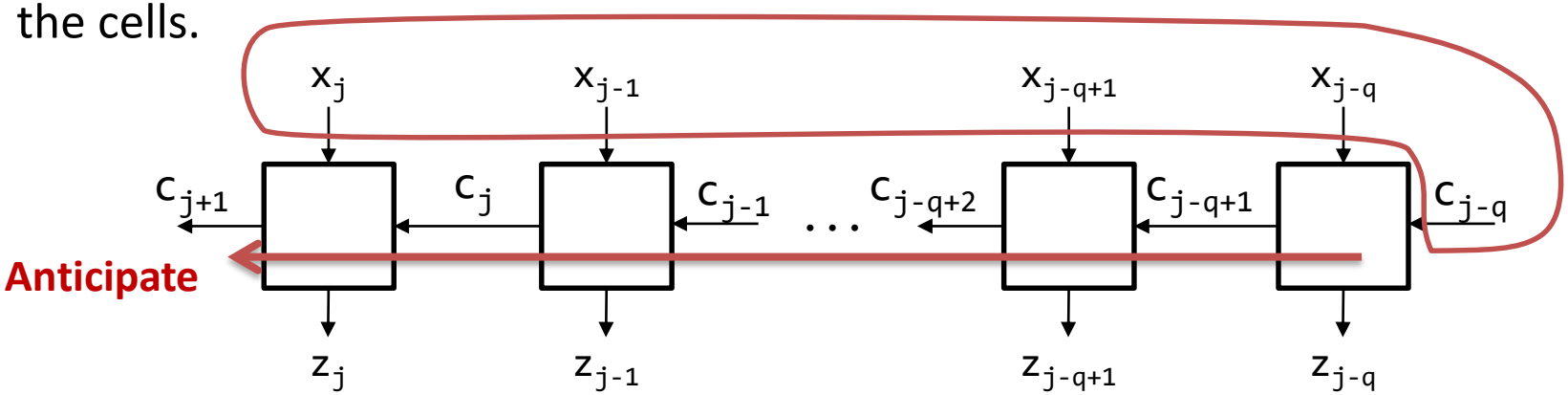
- Objective: reduce the delay of iterative networks.
- How? Anticipating the value of the signal that is propagated through all the cells.



$$c_{j+1} = G(x_j, c_j)$$
$$c_j = G(x_{j-1}, c_{j-1})$$
$$c_{j-1} = G(x_{j-2}, c_{j-2})$$
$$\vdots$$
$$c_{j-q+1} = G(x_{j-q}, c_{j-q})$$

Anticipation networks

- Objective: reduce the delay of iterative networks.
- How? Anticipating the value of the signal that is propagated through all the cells.



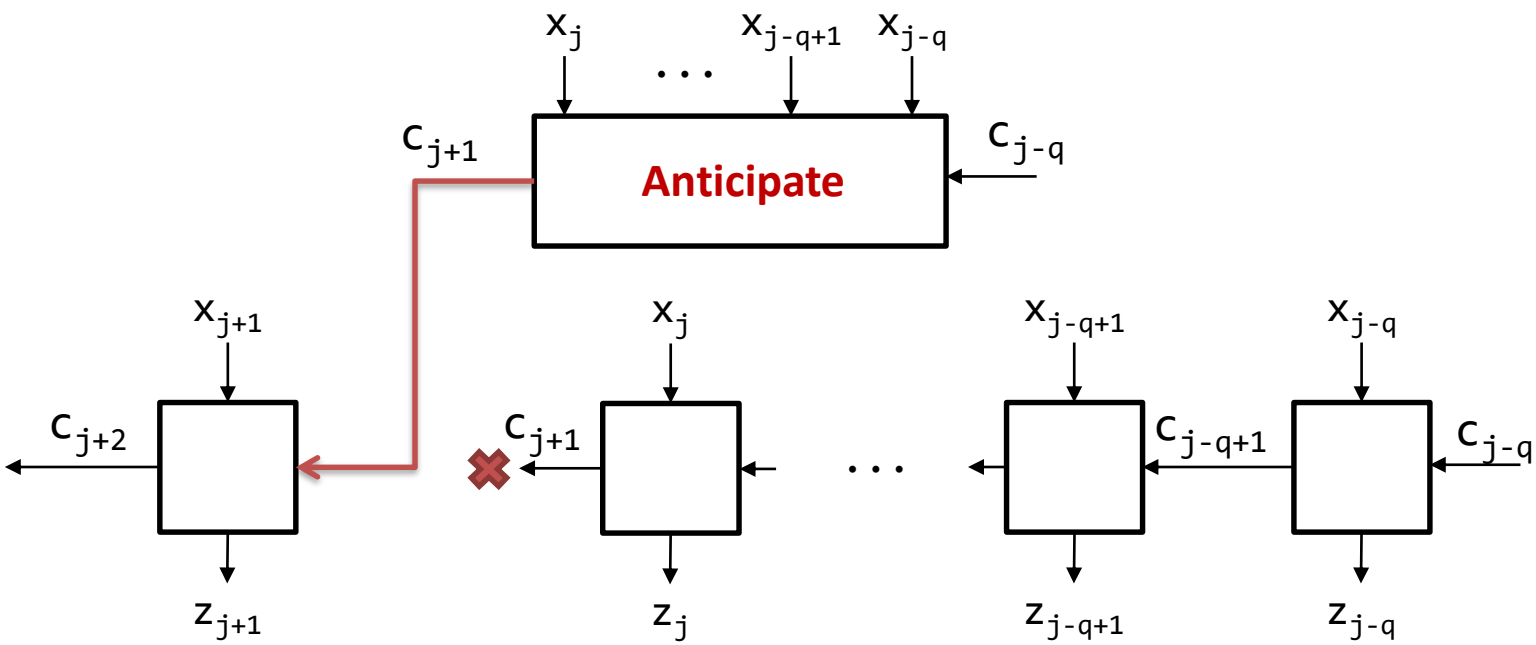
$$\left.\begin{aligned} c_{j+1} &= G(x_j, c_j) \\ c_j &= G(x_{j-1}, c_{j-1}) \\ c_{j-1} &= G(x_{j-2}, c_{j-2}) \\ &\vdots \\ c_{j-q+1} &= G(x_{j-q}, c_{j-q}) \end{aligned}\right\} \Rightarrow c_{j+1} = G(x_j, G(x_{j-1}, G(x_{j-2}, G(\cdots G(x_{j-q}, c_{j-q}))))))$$

$$\Downarrow \text{ Anticipate}$$

$$c_{j+1} = H(x_j, x_{j-1}, x_{j-2}, \cdots, x_{j-q}, c_{j-q})$$

Anticipation networks

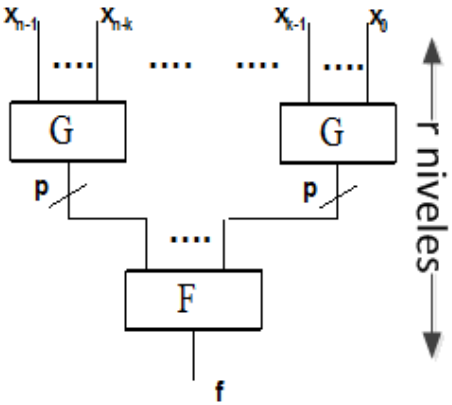
$$c_{j+1} = H\left(x_j, x_{j-1}, x_{j-2}, \cdots, x_{j-q}, c_{j-q}\right)$$



- The anticipation module has to be faster than the normal propagation of the intermediate signal; otherwise, this will not be useful!!

Tree networks

- Implement a function with *n* inputs using blocks with *k* inputs.

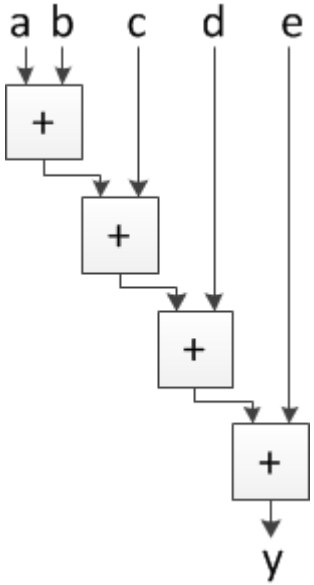


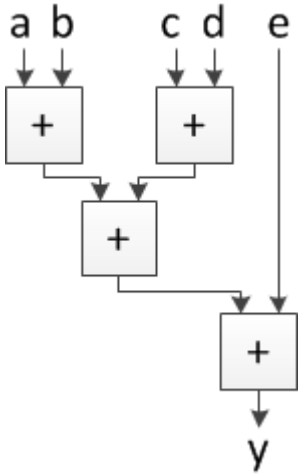
  - Module G: *k* inputs and *p* outputs ➡ $\frac{n}{k}$ modules
  - Module F: $p\frac{n}{k}$ inputs
  - Delay: $(r-1)\Delta_G + \Delta_F$

Tree networks. Examples

- Decoders tree
- Multiplexers tree
- Adders

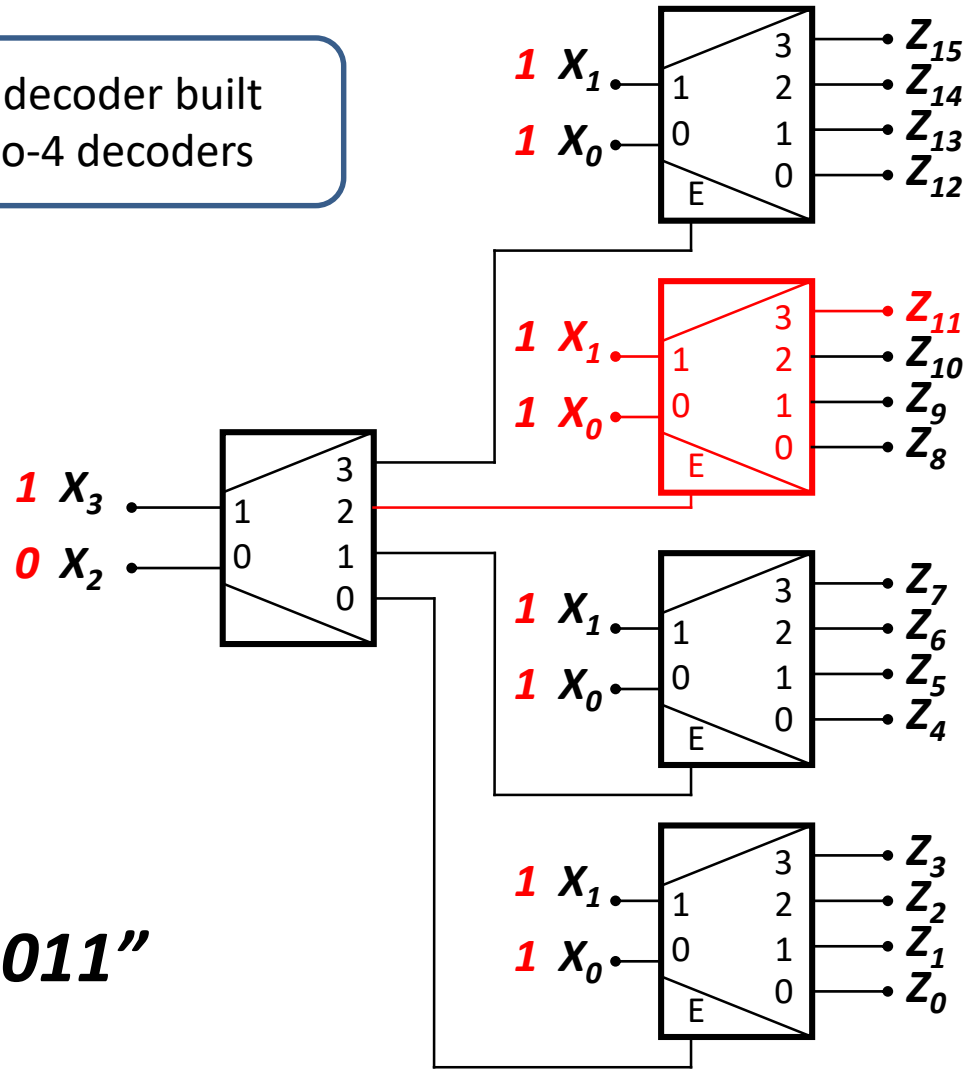
$$y = a + b + c + d + e$$





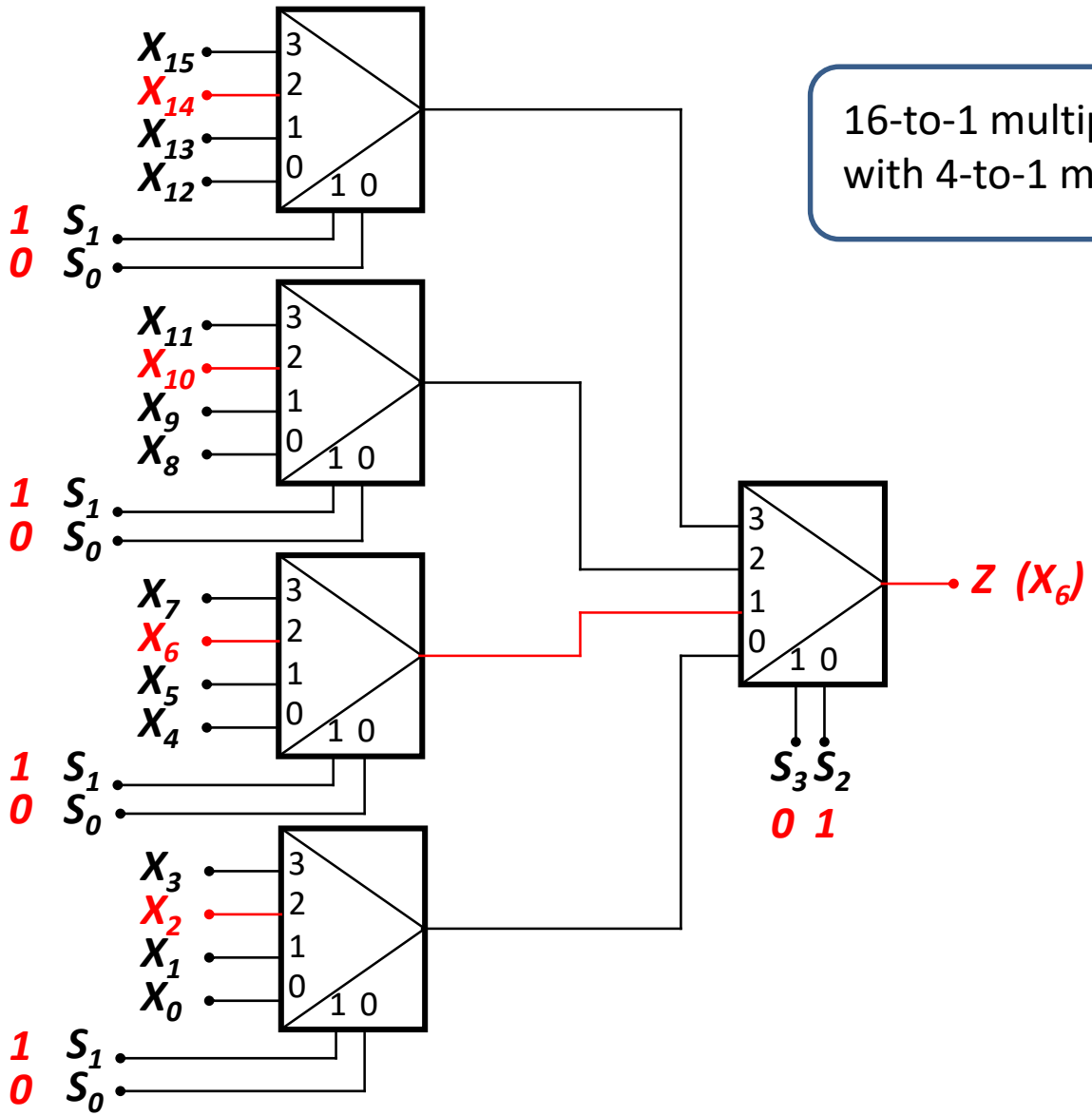
Decoders tree

4-to-16 decoder built with 2-to-4 decoders



X = "1011"

Multiplexers tree



16-to-1 multiplexer built with 4-to-1 multiplexers

S = "0110"

Tree networks. More examples

- Design a system that receives as input 16 2-bit numbers, and it returns the lowest number.

- Design a system that calculates the parity of a number by means of a tree of XOR gates.

Outline

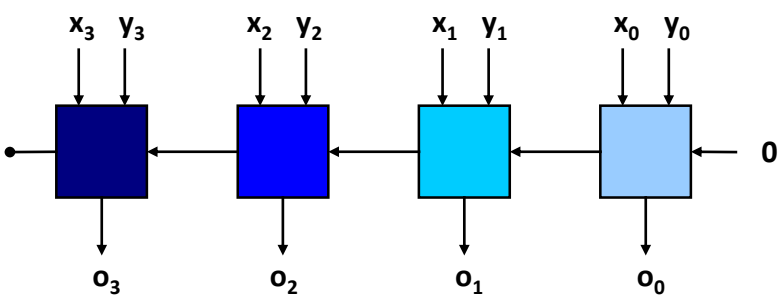
Adders

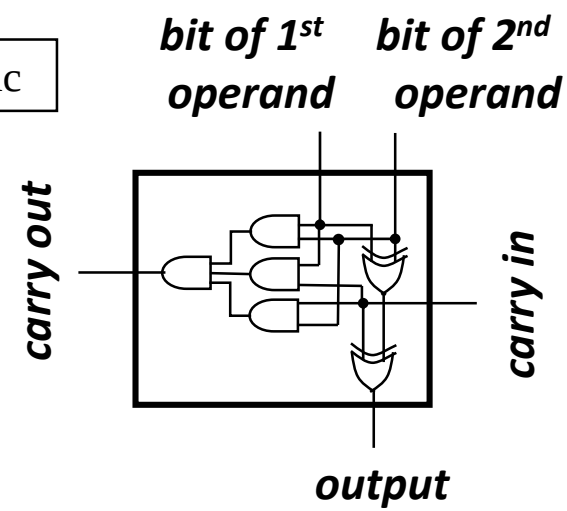
- Design a combinatorial cell that, taking two inputs and a possible additional carry, generates an output result and a carry-out
- Create as many cell instances as there are bits in the input numbers

$$o_i = (x_i \oplus y_i) \oplus c_i$$

$$c_{i+1} = x_i \cdot y_i + x_i \cdot c_i + c_i \cdot y_i = x_i \cdot y_i + (x_i \oplus y_i) \cdot c_i$$

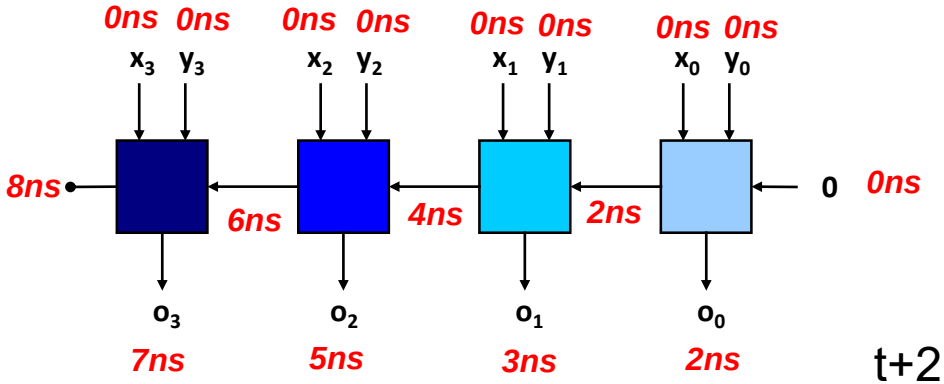
2 levels of logic

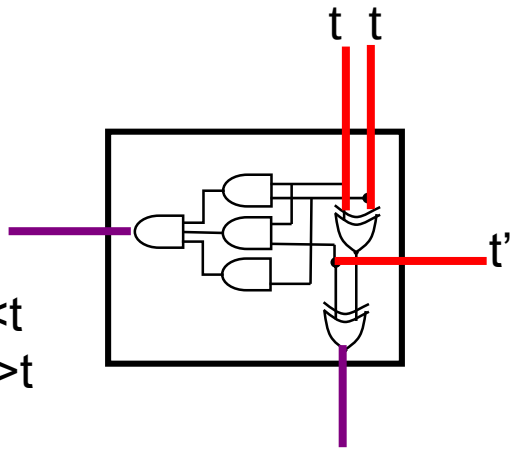




Adders

Assuming that each logic gate has a delay of 1 ns:





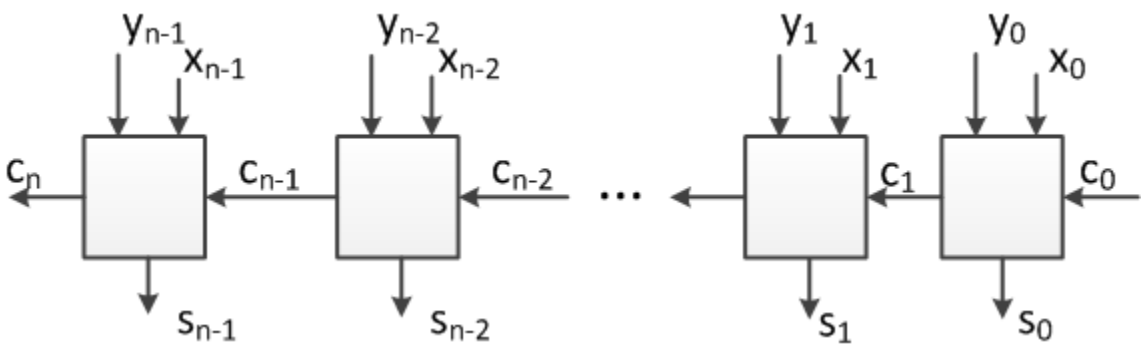
t+2ns if t'<t
t'+2ns if t'>t

t+2ns if t'<t+1ns
t'+1ns if t'>t+1ns

results in worst-case scenario

Adders

- This design is a *carry-ripple adder*.



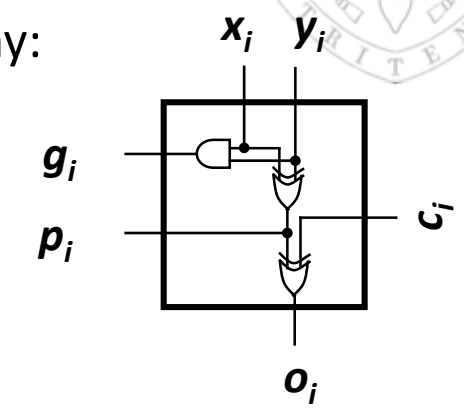
- Drawback: Since the carry signals traverse all the cells that constitute the adder, it is very slow.

$$D_{total} = (n-2)\cdot D_{cin\to cout} + MAX(D_{x,y\to cout}, D_{cin\to cout}) + MAX(D_{cin\to s}, D_{cin\to cout})$$

Carry-lookahead adder

- By means of carry anticipation, we want to reduce the delay:

$$c_{i+1} = x_i \cdot y_i + (x_i \oplus y_i) \cdot c_i = g_i + p_i c_i$$



  - $g_i = 1$ if the cell **generates** a carry; i.e., $x_i = y_i = 1$
  - $p_i = 1$ if the cell **propagates** a carry; i.e., $x_i \oplus y_i = 1$

$$o_i = (x_i \oplus y_i) \oplus c_i = p_i \oplus c_i$$

$$c_{i+1} = x_i \cdot y_i + x_i \cdot c_i + c_i \cdot y_i = x_i \cdot y_i + (x_i \oplus y_i) \cdot c_i = g_i + p_i \cdot c_i$$

  - The new equations for c_1, c_2, c_3 and c_4 look like as follows:

$$c_1 = g_0 + p_0 \cdot c_0$$

$$c_2 = g_1 + p_1 \cdot c_1$$

$$= g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0$$

$$c_3 = g_2 + p_2 \cdot c_2$$

$$= g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0$$

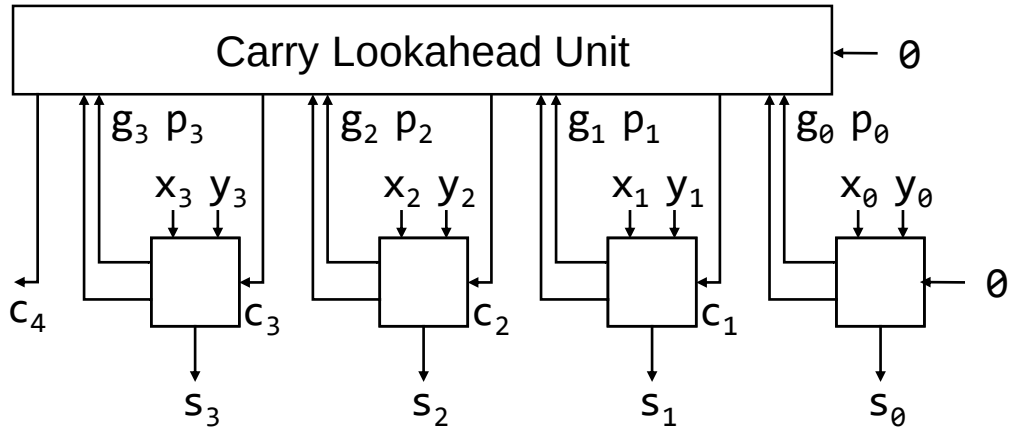
$$c_4 = g_3 + p_3 \cdot c_3$$

$$= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0$$

2 levels of logic

Carry-lookahead adder

- If we use a circuit to anticipate the carry, we don't have to wait for the carry to be propagated through the network. It can be calculated IN ADVANCE by a new circuit.
  - We will name this circuit "Carry Lookahead Unit".



propagation

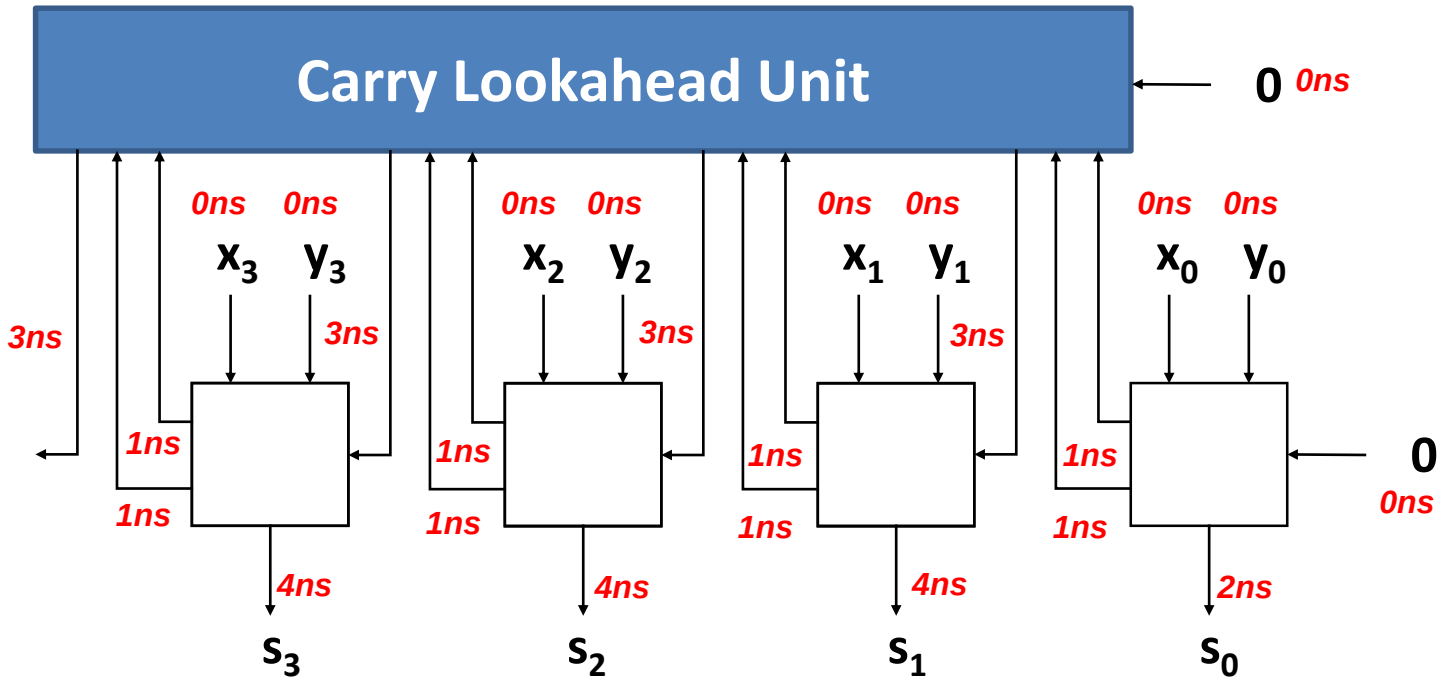
generation

input carry

  - Inputs: g_i, p_i and c_{in}
  - Outputs: c_4, c_3, c_2 and c_1, all of them calculated in parallel.

Carry-lookahead adder

- In order to obtain any bit in the result, it is necessary to go through ONLY 4 levels of gates (instead of 8).



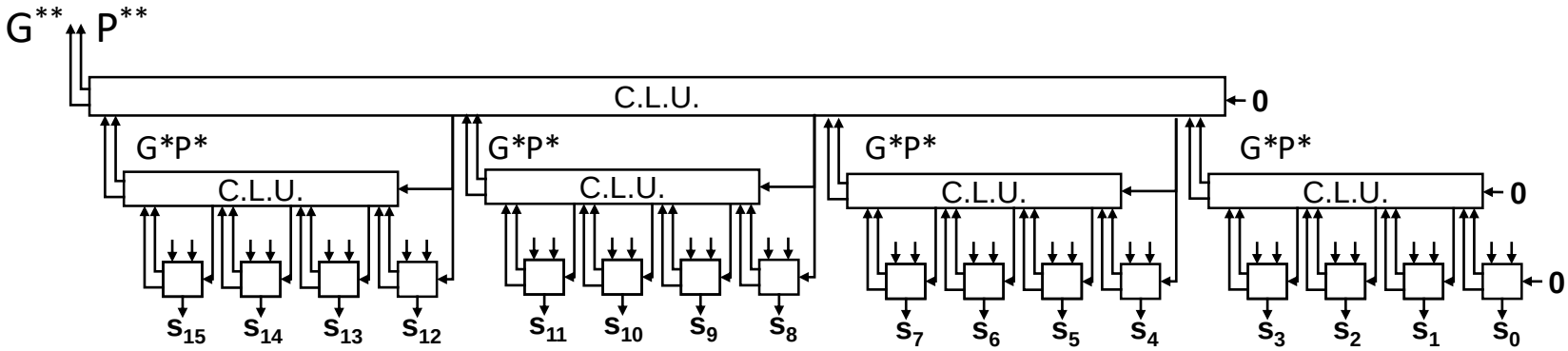
Assuming that each logic gate has a delay of 1 ns:

Carry-lookahead adder

- Problem
  - As the number of bits in the input numbers grow, the number of multiplications and factors grows too much.
  - For instance, for 32 bits, the calculation of c_{32} involves 33 multiplications and 33 factors in the worst case.
  - Actually gates do not have a constant delay:
    - The bigger the gate is, the greater its delay
- Solution
  - **Multilevel carry lookahead.**

Carry-lookahead adder

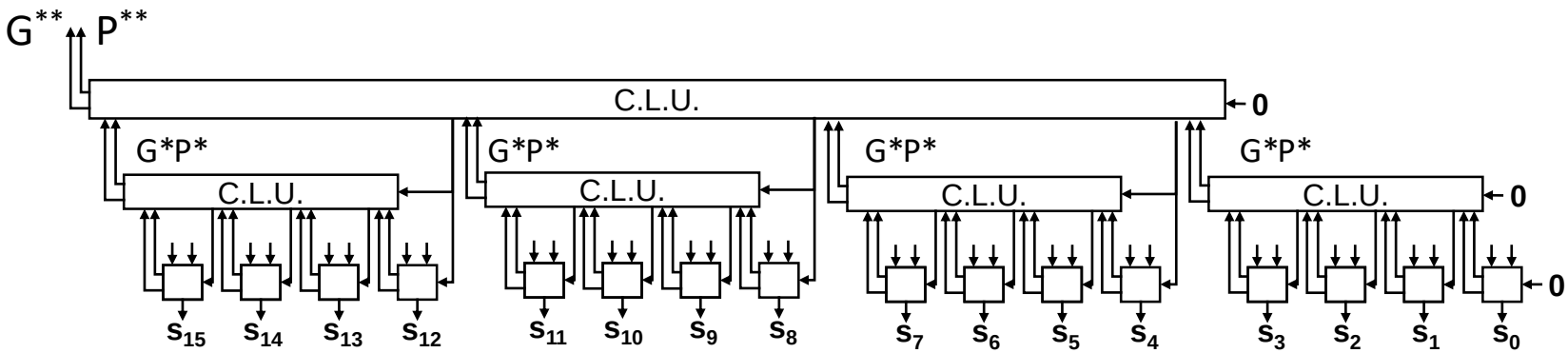
- Multilevel carry lookahead



  - Extended approach: several levels of carry lookahead.
  - The level-0 C.L.U.'s are directly connected to the cells of the adder, whereas the C.L.U.'s of upper levels *i* are connected to the C.L.U.'s of level *i-1*.
  - This approach is recommendable if the anticipation involves very large input numbers. However, it leads to a higher resources consumption.

Carry-lookahead adder

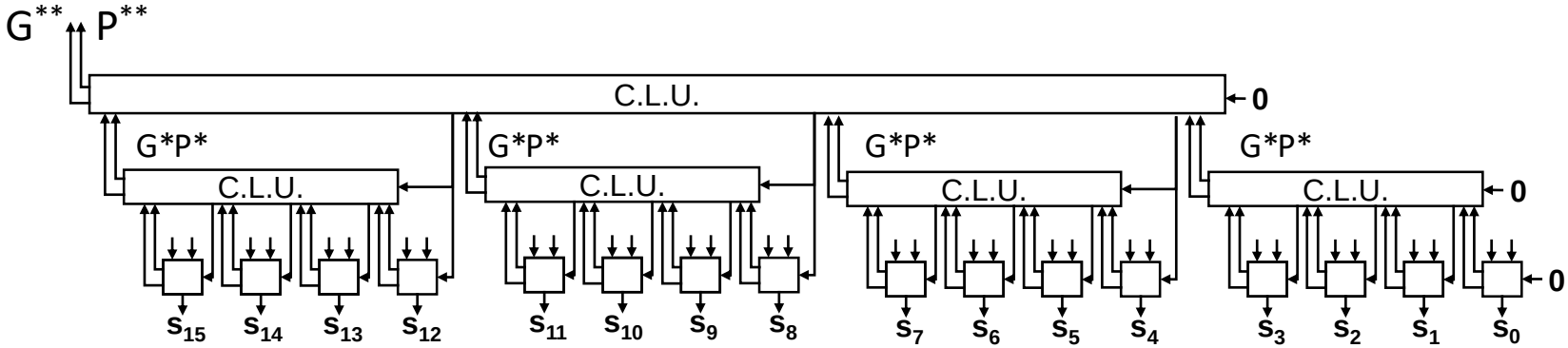
- Multilevel carry lookahead



  - A module **generates** a carry if any of its internal cells generates a carry and it is propagated to the output.
  - A module **propagates** a carry if the input carry is ‘1’ and all its intermediate cells propagate it.

Carry-lookahead adder

- Multilevel carry lookahead



– Equations of a C.L.U. of 4 bits:

$$c_1 = G_0 + P_0 \times c_0$$

$$c_2 = G_1 + P_1 \times c_1$$

$$= G_1 + P_1 \times G_0 + P_1 \times P_0 \times c_0$$

$$c_3 = G_2 + P_2 \times c_2$$

$$= G_2 + P_2 \times G_1 + P_2 \times P_1 \times G_0 + P_2 \times P_1 \times P_0 \times c_0$$

$$\cancel{c_4 = G_3 + P_3 \times c_3}$$

$$\cancel{= G_3 + P_3 \times G_2 + P_3 \times P_2 \times G_1 + P_3 \times P_2 \times P_1 \times G_0 + P_3 \times P_2 \times P_1 \times P_0 \times c_0}$$

$$G^* = G_3 + G_2 \times P_3 + G_1 \times P_2 \times P_3 + G_0 \times P_1 \times P_2 \times P_3$$

$$P^* = P_3 \times P_2 \times P_1 \times P_0$$

New signals G* and P*

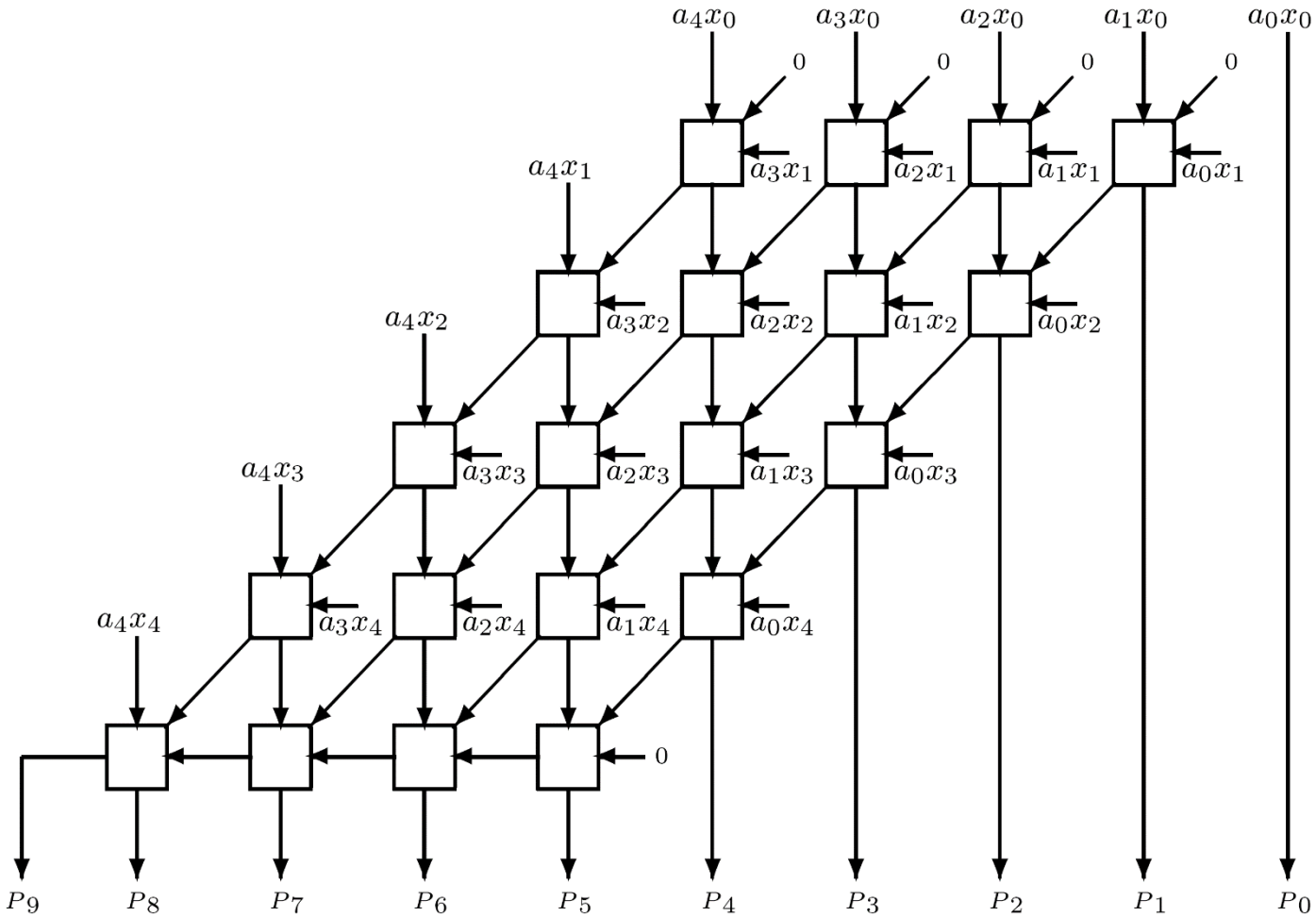
c_1, c_2 and c_3 are generated identically as the previous version of the C.L.U.
c_4 no longer exists

Multiplication of unsigned integers

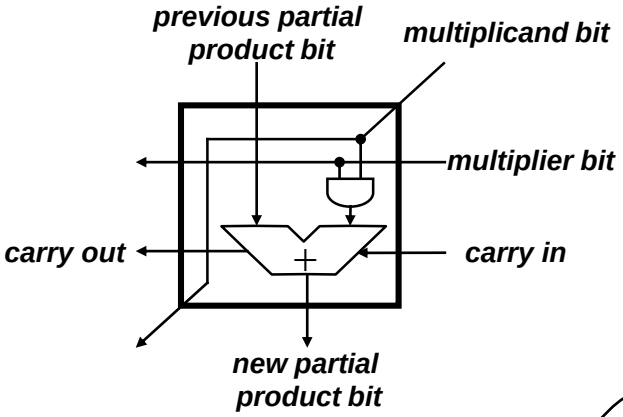
					a_4	a_3	a_2	a_1	a_0	Multiplicand (M)
					x_4	x_3	x_2	x_1	x_0	Multiplier (m)
					a_4x_0	a_3x_0	a_2x_0	a_1x_0	a_0x_0	Partial products (PP)
				a_4x_1	a_3x_1	a_2x_1	a_1x_1	a_0x_1		
			a_4x_2	a_3x_2	a_2x_2	a_1x_2	a_0x_2			
		a_4x_3	a_3x_3	a_2x_3	a_1x_3	a_0x_3				
	a_4x_4	a_3x_4	a_2x_4	a_1x_4	a_0x_4					
P_9	P_8	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	

				1	1	1	0	1
				1	1	0	1	1
				1	1	1	0	1
			1	1	1	0	1	
		0	0	0	0	0		
	1	1	1	0	1			
1	1	0	0	0	0	1	1	1	1

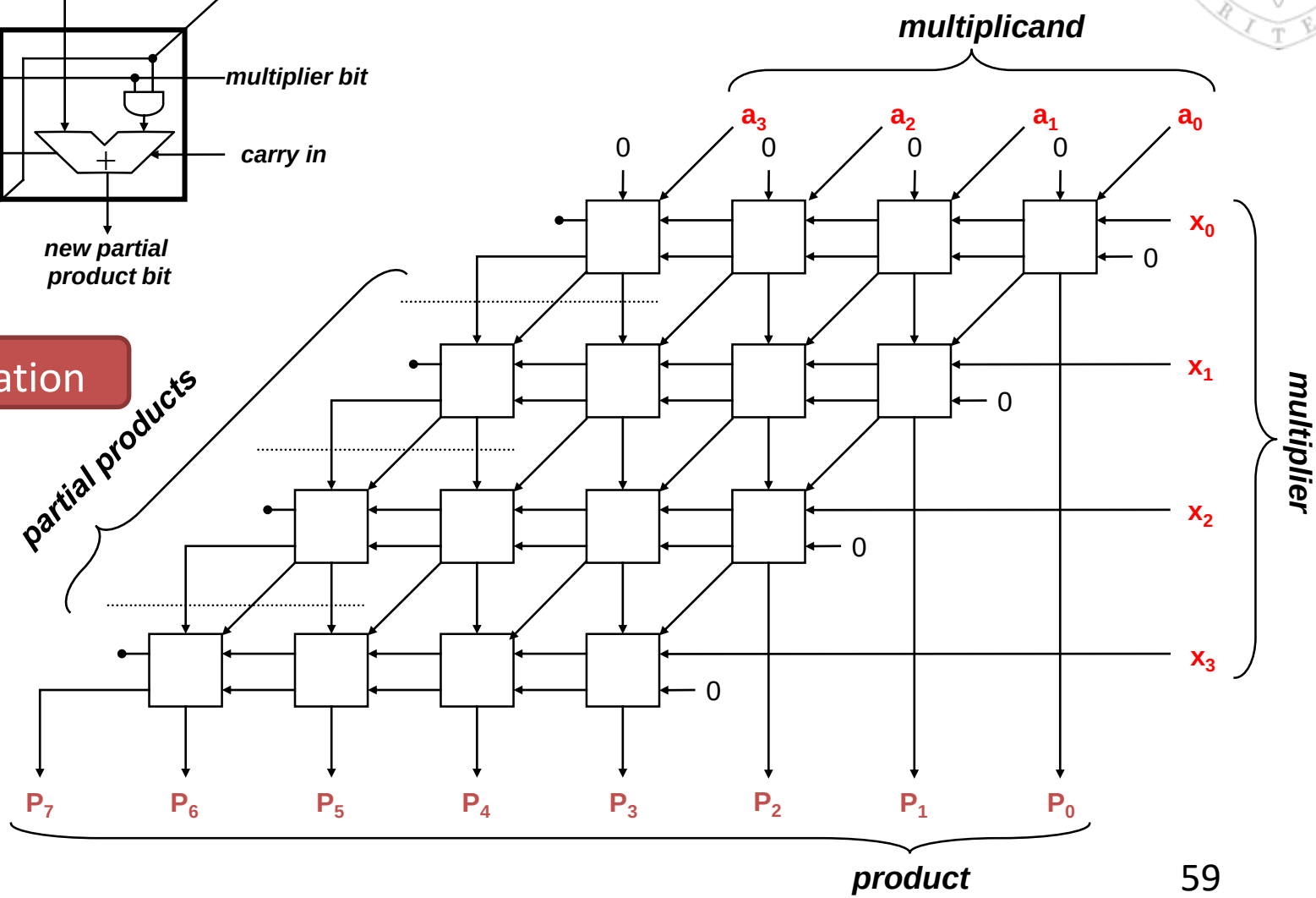
Combinatorial multiplier



Combinatorial multiplier



Modification



Combinatorial multiplier

- Example