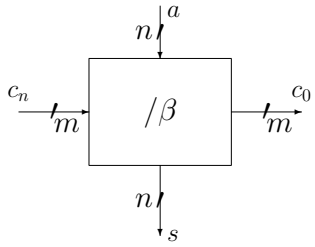
8 Design Example: A Division-by-Constant Combinational Circuit

8.1 A general case

A combinational circuit which divides n-bit binary number by a 'small' constant β has a modular structure of an iterative 1-D array circuit, similar to the structure of an incrementer or an adder. The carry propagates from left to right, and its values are limited by the divisor, β.



$(2^n \cdot c_n + a)$ – dividend
β – divisor
s – quotient
c_0 – remainder

β is a 'small' integer constant,
$a, s \in \{0, \ldots, 2^n - 1\}$ are n-bit integers,
$c_n, c_0 \in \{0, \ldots, \beta - 1\}$ are m-bit integers,
$2^m \geq \beta$

Input/output variables are related by the following equation which links divider, divisor, quotient and remainder:

$$\frac{2^n \cdot c_n + a}{\beta} = s + \frac{c_0}{\beta}$$

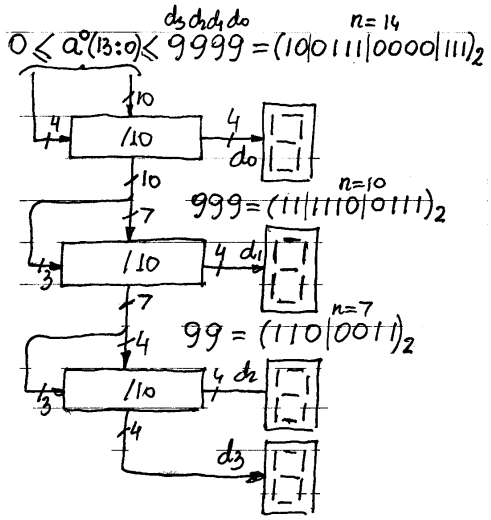
or

$$2^n \cdot c_n + a = \beta \cdot s + c_0$$

- The objective is to build a combinational circuit, which, given n-bit input a and possibly c_n, will generate the quotient s and the remainder, c_0.
- It is possible to build such a circuit using 1-bit cells.

8.2 Binary-to-decimal conversion

- The division-by-constant circuit can be used for binary-to-decimal conversion.
- It is the "division-by-target" radix method, therefore $\beta = 10$.

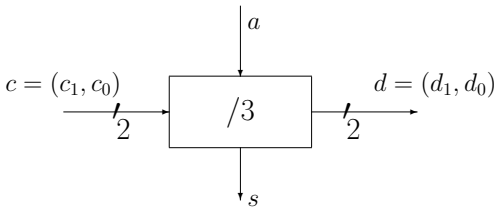


- In the example we consider conversion of a binary number to a 4-digit decimal number.
- The largest 4-digit decimal number, 9999, is represented by a 14-bit binary number.
- The first level division-by-10 circuit generate the first digit d_0 as a remainder and a 10-digit quotient that is equivalent to 3-digit decimal number not greater than 999.
- The final level division-by-10 circuit generate two last decimal digits, d_3, d_2

8.3 A 1-bit division-by-constant circuit

- Consider as an example a division-by-3 circuit. In this case, we have

$$n = 1; \quad \beta = 3; \quad m = 2; \quad \{2^m \geq \beta\}$$



- Input and output carry signals, c, d, are 2-bit numbers which are less than the divisor, $\beta = 3$, that is:

$$c,\ d \in \{0,\ 1,\ 2\}$$

- The I/O equation is now of the following form:

$$2 \cdot c + a = 3 \cdot s + d$$

Function Table

c	a	$2c+a$ / $3s+d$	s	d
0	0	0	0	0
0	1	1	0	1
1	0	2	0	2
1	1	3	1	0
2	0	4	1	1
2	1	5	1	2
3	0	6	–	–
3	1	7	–	–

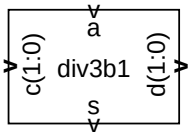
Truth Table

c_1	c_0	a	s	d_1	d_0
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	–	–	–
1	1	1	–	–	–

- From the tables the logic equations for the three outputs can be derived using the Karnaugh map technique.
- One possible SoP form is as follows:

$$\begin{aligned} s &= c_1 + a \cdot c_0 \\ d_1 &= \overline{a} \cdot c_0 + a \cdot c_1 \\ d_0 &= \overline{a} \cdot c_1 + a \cdot \overline{c_1} \cdot \overline{c_0} \end{aligned}$$

A possible VHDL implementations of the 1-bit cell, **div3b1** based on the derived logic equations is as follows:




```
LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

-- div3b1, 1-bit div-by-3 Entity Description
ENTITY  div3b1  IS
  PORT(
     c : IN  std_logic_vector (1 downto 0) ;
     a : IN  std_logic ;
     d : OUT std_logic_vector (1 downto 0) ;
     s : OUT std_logic
  );
END div3b1;

-- arch1 Architecture Description
ARCHITECTURE  arch1  OF  div3b1 IS

BEGIN
     s <=  c(1) or (a and c(0)) ;
  d(1) <=  (not a and c(0)) or (a and c(1)) ;
  d(0) <=  (not a and c(1)) or (a and not c(1) and not c(0)) ;
END arch1 ;
```

- At this stage we might like to avoid deriving the logic equations and use the truth table directly, as in the 1-bit adder example.
- We specify the **truth table as a constant** as it is illustrated in the next section.

8.4 An architecture with the truth table specification

- The truth table is simply specified as a constant **array of binary words**. The array consists of 2^3 3-bit words as in the truth table on page 8–3.

```
ARCHITECTURE  dv3tt  OF  div3b1 IS
   TYPE arr_vec IS ARRAY (natural range <>)
               OF std_logic_vector(2 downto 0);
   CONSTANT sd_ac : arr_vec := (
   -- sd1d0     ca
   --------------- the truth table:
      "000",  -- 0
      "001",  -- 1
      "010",  -- 2
      "100",  -- 3
      "101",  -- 4
      "110",  -- 5
      "---",  -- 6
      "---"); -- 7
   SIGNAL  sd : std_logic_vector (2 downto 0) ;
BEGIN
  -- reading from the truth table :
  sd <= sd_ac(conv_integer(unsigned(c & a)));
   d <= sd (1 downto 0) ;
   s <= sd (2) ;
END dv3tt ;
```

- The truth table is a constant sd_ac of the type arr_vec.
- The value of the constant is our truth table. Note that we can also specify the "don't care" values, '–'.
- To read the values from the truth table we need an assignment statement of the form

  ```
  sd <= sd_ac(ca);
  ```

- In the **array** specification we have implicitly specified that the address signal ac is of the **type integer**.
- Therefore, we first concatenate **(c & a)** into a 3-bit **std_logic_vector**, which is subsequently converted into a 3-bit **unsigned** vector that can be converted into an integer.

- The conversion functions are specified in the libraries **ieee.std_logic_1164** and **ieee.std_logic_arith**.

- We consider another architecture of the 1-bit division-by-3 circuit, based on the division and remainder operations.

  Such operators are available in the numeric_std IEEE library.

- The division '/' and remainder **rem** , like all other arithmetic operators, do not operate on signals of the type **std_logic_vector**.

  Instead, we can use signals of the type **unsigned** .

- The conversion function **std_logic_vector** converts signals back from the **unsigned** to **std_logic_vector** form.

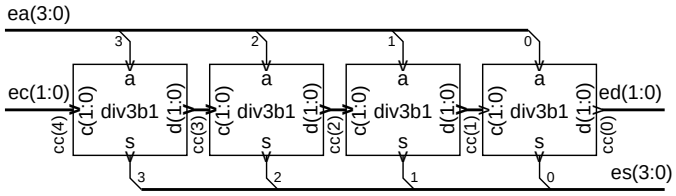
- Since ca is a 3-bit signal, the results of division and remainder operations are also 3-bit signals.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

-- div3b1, 1-bit div-by-3 Entity Description
ENTITY  div3b1b  IS
  PORT(
     c : IN  std_logic_vector (1 downto 0) ;
     a : IN  std_logic ;
     d : OUT std_logic_vector (1 downto 0) ;
     s : OUT std_logic
  );
END div3b1b;
ARCHITECTURE  dv3mod  OF  div3b1b IS
  SIGNAL     ca : unsigned (2 downto 0);
  SIGNAL ss, dd : std_logic_vector (2 downto 0);
BEGIN
 ca <= unsigned(c & a) ;
 dd <= std_logic_vector(ca REM 3) ;
  d <= dd(1 downto 0) ;
 ss <= std_logic_vector(ca/3) ;
  s <= ss(0) ;
END dv3mod ;
```

8.5 An n-bit division-by-three circuit

- In order to implement an n-bit division-by-3 circuit, we can instantiate the 1-bit component div3b1 using the **port map** and **generate** statements.
- The resulting structure of a 4-bit division-by-3 circuit has the following block-diagram:



- The numbers represented by the port signals are related by the following arithmetic equality:

$$16 \cdot ec + ea = 3 \cdot es + ed$$

- The internal carry signals, cc, form a 5 by 2 array of 2-bit signals.

The block diagram of the 4-bit division-by-3 circuit is equivalent to the following VHDL code:


```
LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY  div3b4  IS    ---   4-bit division-by-3
  GENERIC (N : natural := 3, m : natural := 1) ;
  PORT(
     ec : IN  std_logic_vector(m downto 0) ;
     ea : IN  std_logic_vector(N downto 0) ;
     ed : OUT std_logic_vector(m downto 0) ;
     es : OUT std_logic_vector(N downto 0) );
END div3b4;
ARCHITECTURE  strctrl  OF  div3b4  IS
  COMPONENT div3b1
    PORT(
      c : IN  std_logic_vector(1 downto 0) ;
      a : IN  std_logic ;
      d : OUT std_logic_vector(1 downto 0) ;
      s : OUT std_logic
        ) ;
  END COMPONENT div3b1 ;
  TYPE arr5b2 IS ARRAY(4 downto 0)
              OF std_logic_vector(1 downto 0);
  SIGNAL cc :  arr5b2 ;
BEGIN
  cc(4) <= ec ;
  glp: FOR i IN  ea'RANGE  GENERATE
    U1: div3b1
    PORT MAP ( c => cc(i+1),
               a => ea(i),
               d => cc(i),
               s => es(i)
             );
  END GENERATE ;
  ed <= cc(0) ;
END strctrl ;
```

Note that

- In the **generate** statement ea'range is equivalent to 4 downto 0.
- In the component instantiation statement, **port map** , we have used the association-by-name method.