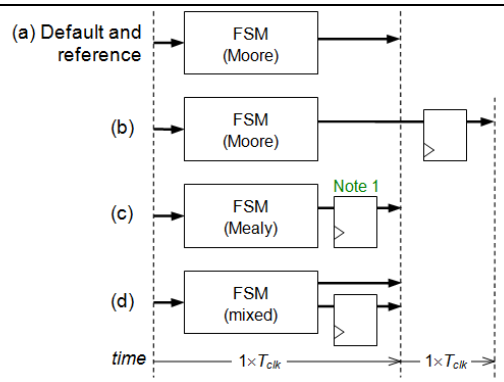
Errata and Clarifications (rev.1)

1. ERRATA

Page	Error
87	In the transitions of figure 3.19a, it should be *i* instead of *t*.
195	Below, the word "is" is missing before the parenthesis: **8.2.2 Enumeration Types** An enumeration type consists of a list of symbolic values. It can be declared as follows: `type type_name (type_values_list);`
291	In process P2, the line `count <= std_logic_vector(to_unsigned(i, BITS));` should be added between lines 29 and 30.
306	In process P1, the line `is_max <= '0';` should be added between lines 10 and 11.
320	In example 13.3, the number of bits of figure 2.35a (page 63) were employed, which are for the worst-case scenario, i.e., for *arbitrary unsigned* values. That was fixed and clarified in sections 3.1 and 3.2 ahead.
334	In figures 13.7a-b, it should be q_0-q_2 and q_0-q_1, respectively.
359	Below, it should be "functions", not "variables": **14.4 Procedure** Compared to functions, procedures are used to implement multi-output problems. Moreover, procedures are stand-alone statements, while variables are used as part of expressions.
403	About Mealy machines: See in the **Clarifications** (next table), the important Note to be included in figure 15.4.
421	In exercise 16.2, relax the requirement "*dout* must be guaranteed to be glitch free."
458	In line 105 of code, it should be `when 3 to 5`
534	In line 9 of both codes (for character 6), it should be `"0100000"`.

2. CLARIFICATIONS

Page	Comment
63	The number of bits in figure 2.35a are for the worst-case scenario, which is for *arbitrary unsigned* values. For *signed* values, with arbitrary or fixed coefficients, and for chain- or tree-type architecture, see section 3.1 ahead.
89	Figure 3.21 shows an arbiter for *n*=3 clients. The one-clock-period latency can be easily eliminated, but for higher *n* (=8, for example), the number of transitions becomes exceedingly large (see the new Arbiter exercise in **Exercises: Comments and Extensions**, at vhdl.us).
171	Table 7.8 includes all *functions* available in the *math_real* package, but it has a line repeated (**). There is also a *procedure* in that package, called *uniform*, useful for generating random numbers in simulation.
320	See the comments regarding example 13.3 in sections 3.1 and 3.2 ahead.
347	In exercise 13.37, the number of coefficients is 11 (so *M*=10) and the number of bits in the input and in the coefficients are N_x=N_b=4. The filter is *signed*, and the coefficients, being programmable, are arbitrary (as opposed to fixed). The number of bits along the chain (lower part of figure 2.36) is $N_i = N_x + N_b - 1 + \lceil log_2(i+2) \rceil$ $(0 \le i \le M)$. In the simulations, use the same coefficient values of example 13.3. **Suggestions:** *Solution 2:* Solve this exercise also for the *tree-type* architecture (figure 2.35b). The equations for the number of bits are in the table of section 3.1 below. *Solution 3:* Solve this exercise also for the *linear-phase* case (figure 2.35d).
374	In topic (3) of page 374, an important recommendation for Mealy is missing; it is for the implementation of recursive machines without latency, which leads to the construction of figure 15.4c.
403	To make section 15.6 clearer, include in figure 15.4 the Note in green below.



Suggested templates							
Registered outputs	Total time	Figure above	Machine type	Machine category			
				(1) Regular	(2) Timed	(3.1) Recursive	(3.2) Recursive-timed
None	$1\times T_{clk}$	(a)	Moore	T1 or T2	T10	----	----
All	$2\times T_{clk}$	(b)	Moore	T3 or T4	T11	----	----
	$1\times T_{clk}$	(c)	Mealy	T8 or T9	T14	T8 or T9	T14
Some	$1\times T_{clk}$	(d)	Mixed	T5 or T6	T12	T5 or T6	T12
Note 1: For the case in figure (c), with the output register removed, use template T7 if Regular or T13 if Timed.							

3. ADDITIONAL DETAILS

3.1 Number of bits in signed multiplier-adder arrays

In example 13.3 (page 320), the number of bits of figure 2.35a (page 63) were employed, which are for the worst-case scenario, i.e., for *arbitrary unsigned* values. The numeric values illustrating the implementation, however, include positive and negative coefficients, and they are stored in ROM-like memory, so a *signed* filter with *fixed* coefficients is in principle implied. This section presents the equations for *all* signed cases, followed by the adjusted code for example 13.3 in the next section.

The following notation is used in the equations:
M = Filter order (= number of coefficients − 1)
N_x = Number of bits in the input signal (x)
N_b = Number of bits in the filter coefficients (b_i, $0 \le i \le M$)
N_y = Number of bits in the output signal (y)
L = Number of sum layers in the tree-type array ($L=\lceil log_2(M+1) \rceil$)
i = Chain stage index, horizontal (i=0 to M, Fig. 2.35a)
j = Tree layer index, vertical (j=0 to L, Fig. 2.35b)

Minimum number of bits in signed multiplier-adder arrays (Fig. 2.35 of book).

Architecture	Polarity	Coeff.	Position	Equations	#
Chain-type (Fig. 2.35a)	Signed	Arbitrary	Bits along the chain	$N_i = N_x + N_b - 1 + \lceil log_2(i+2) \rceil \quad (0 \le i \le M)$	(1)
			Bits at the output	$N_y = N_x + N_b - 1 + \lceil log_2(M+2) \rceil$	(2)
		Fixed	Bits along the chain	$N_i = \begin{cases} N_x + N_b - 1 + \lceil log_2(i+2) \rceil & \text{while } N_i < N_y \\ \text{Else } N_y \text{ (equation below)} \end{cases}$	(3)
				Where $N_b = \begin{cases} \lceil log_2(\lvert b_{min} \rvert) \rceil + 1 & \text{if } \lvert b_{min} \rvert > b_{max} \\ \lceil log_2(b_{max}+1) \rceil + 1 & \text{if } \lvert b_{min} \rvert \le b_{max} \end{cases}$	(4)
			Bits at the output	$N_y = N_x + \left\lceil log_2\left(\sum_{i=0}^{M} \lvert b_i \rvert + 1 \right) \right\rceil$	(5)
Tree-type	Signed	Arbitrary	Bits along the tree	$N_j = N_x + N_b + j \quad \left(0 \le j < L,\ L = \lceil log_2(M+1) \rceil\right)$	(6)

(Fig. 2.35b)			Bits at the output	$N_y = \begin{cases} N_x + N_b + L & \text{if } M+1 \text{ is a power-of-two} \\ N_x + N_b + L - 1 & \text{otherwise} \end{cases}$	(7)
		Fixed	Bits along the tree	$N_j = \begin{cases} N_x + N_b + j & \text{while } N_j < N_y \ (0 \le j < L) \\ \text{Else } N_y & \text{(equation below)} \end{cases}$ Where N_b is given by eq. (4)	(8)
			Bits at the output	$N_y = N_x + \left\lceil log_2 \left(\sum_{i=0}^{M} \lvert b_i \rvert \right) + 1 \right\rceil$	(9)

3.2 Reviewed version of *Example 13.3. FIR filter with fixed coefficients*

We can now make use of the table above to adjust the circuit of example 13.3 to operate as *signed*, with *fixed* coefficients, in a chain-type architecture. The right equations are (3)-(5). From (4): N_b=4; from (3), for i=0: N_0=8; and from (5): N_y=10. Therefore, the number of bits along the chain starts with 8 and can be stopped when it reaches 10. This modification (which is the only real modification) is in line 11 of the code below. Lines 9-10 are just a splitting of the original line 10 to make it clear that N_x and N_b can be different. The rest are just adjustments to comply with the new parameter names.

```
1  ----------------------------------------------------------------------------
2  library ieee;
3  use ieee.std_logic_1164.all;
4  use ieee.numeric_std.all;
5
6  entity fir_filter is
7     generic (
8        NUM_COEF: natural := 11;    --number of filter coefficients
9        BITS_COEF: natural := 4;    --number of bits in the coefficients
10       BITS_IN: natural := 4;      --number of bits in the input signal
11       BITS_OUT: natural := 10);   --number of bits in the output signal
12    port (
13       clk, rst: in std_logic;
14       x: in std_logic_vector(BITS_IN-1 downto 0);
15       y: out std_logic_vector(BITS_OUT-1 downto 0));
16 end entity;
17
18 architecture fixed_coeff_chain_type of fir_filter is
19
20    --Filter coefficients (ROM-type memory with integer as base type):
21    type int_array is array (0 to NUM_COEF-1) of integer range
22       -2**(BITS_COEF-1) to 2**(BITS_COEF-1)-1;
23    constant coef: int_array := (-8, -5, -5, -1, 1, 2, 2, 3, 5, 7, 7);
24
25    --Internal signals (arrays with signed as base type):
26    type signed_array is array (natural range <>) of signed;
27    signal shift_reg: signed_array(1 to NUM_COEF-1)(BITS_COEF-1 downto 0);
28    signal prod: signed_array(0 to NUM_COEF-1)(BITS_IN+BITS_COEF-1 downto 0);
29    signal sum: signed_array(0 to NUM_COEF-1)(BITS_OUT-1 downto 0);
30
31 begin
32
33    --Shift register:
34    process (clk, rst)
35    begin
36       if rst then
37          shift_reg <= (others => (others => '0'));
38       elsif rising_edge(clk) then
39          shift_reg <= signed(x) & shift_reg(1 to NUM_COEF-2);
40       end if;
41    end process;
42
43    --Multipliers:
44    prod(0) <= coef(0) * signed(x);
45    mult: for i in 1 to NUM_COEF-1 generate
46       prod(i) <= to_signed(coef(i), BITS_COEF) * shift_reg(i);
47    end generate;
48
49    --Adder array:
50    sum(0) <= resize(prod(0), BITS_OUT);
51    adder: for i in 1 to NUM_COEF-1 generate
```

```
52          sum(i) <= sum(i-1) + prod(i);
53      end generate;
54      y <= std_logic_vector(sum(NUM_COEF-1));
55
56  end architecture;
57  --------------------------------------------------------------------------------
```