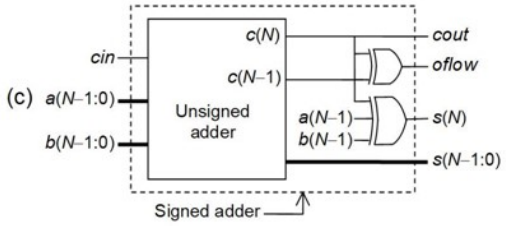
Errata, Clarifications, and Additional Details (rev.3)

1. ERRATA

Page	Correction
18	Replace figure 1.22c with this: 
63	Add the following at the end of the caption for figure 2.35: "The number of bits is for the worst-case scenario (arbitrary unsigned values)."
87	In the transitions of figure 3.19a, it should be *i* instead of *t*.
156, 158, 161	The maximum range for the type *integer*, which is from $-(2^{31}-1)$ to $+(2^{31}-1)$ (as seen in editions 1 and 2 of the book), was brought over incorrectly as -2^{31} to $+(2^{31}-1)$.
195	Below, the keyword "is" is missing before the parenthesis: `type type_name (type_values_list);`

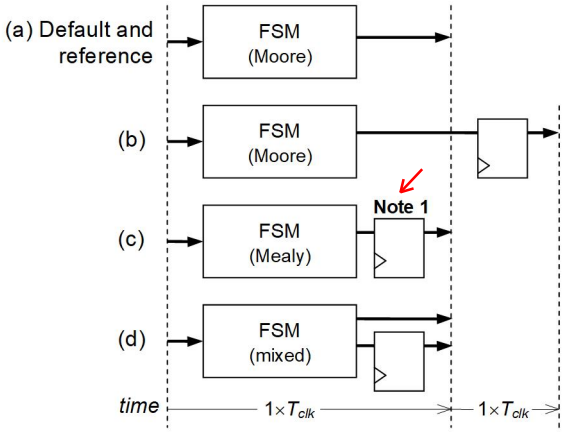
198	Remove the line marked below:
	`s <= ("0000", "1111", "0001");`
	~~`s(2) <= "0001";`~~
	`s(2, 0) <= '1';`

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 clarified and modified in sections 3.1 and 3.2 ahead.
337	In figures 13.7a-b, it should be $q_0q_1q_2$ and q_1q_2, 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."
456	*clk_vga* (line 17 of code) should be an internal signal, so place its declaration after line 25.
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	Replace the paragraph in the box below with that that follows. ~~Notice that this machine clearly falls in the situation described in section 3.3 – that is, the state diagram is useful to expose the problem but, at the same time, it tells us that a solution can be easily implemented without using the (formality of the) FSM approach.~~ This machine could be modified to avoid returning to idle when a request is waiting. Calling *Arb1* that in figure 3.21b and *Arb2* the new arbiter, and calling *n* and *t* the number of inputs and of transitions, respectively, the following is left as an exercise to the reader: (a) Draw the state diagram for *Arb2* (for *n*=3); (b) Write the equation for *t* in each arbiter; (c) Is it viable to sketch a state diagram in each case for *n*=8?
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.
240	Replace the entire description in exercise 9.4 with the following:

	You are given 12 equations of the form *y*=*f*(*a*,*b*), where *y*, *a*, and *b* are all of type INT, with *a* and *b* in the −16 to 15 range. Determine, for each equation, the minimum and maximum values of *y*, disregarding comments (1)-(7) of table 9.4 (the reason for that is that INT has no formal limits, so the comments are just suggested limits).
270	Some might find Example 11.3 too detailed; an alternative version is offered in section 3.3 ahead.
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 $N_i = N_x + N_b - 1 + \lceil log_2(i+2) \rceil$ $(0 \le i \le M)$. Replace parts (a) and (b) with the following: a) Implement it using VHDL, for arbitrary signed coefficients. Observe notes 3 and 4 above. b) Show simulation results, using the same parameters of example 13.3, with comments. Observe note 5 above.
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 Note 1 below in figure 15.4:



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×T_{clk}	(a)	Moore	T1 or T2	T10	----	----
All	2×T_{clk}	(b)	Moore	T3 or T4	T11	----	----
	1×T_{clk}	(c)	Mealy	T8 or T9	T14	T8 or T9	T14
Some	1×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.							

456	Since in example 17.6 the signal *clk_vga* is not used outside the code domain, its declaration (in line 17) could be replaced with an internal signal declaration, added to line 25: `signal clk_vga, Hactive, ...: std_logic;`

3. ADDITIONAL DETAILS

3.1 Number of bits in signed multiplier-adder arrays (pages 63, 320)

In figure 2.35 (page 63) and example 13.3 (page 320), the number of bits employed where 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. Table 1 presents the equations for *all* signed cases, followed by the adjusted code for example 13.3 in the next section.

Table 1. Minimum number of bits in signed multiplier-adder arrays (Fig. 2.35).

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 (Fig. 2.35b)	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)
			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)

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
N_i = Number of bits along the chain or tree
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)

3.2 Reviewed version of *Example 13.3: FIR filter with fixed coefficients* (page 320)

Using equations (3)-(5) of Table 1, we get N_b=4, N_0=8, and 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.

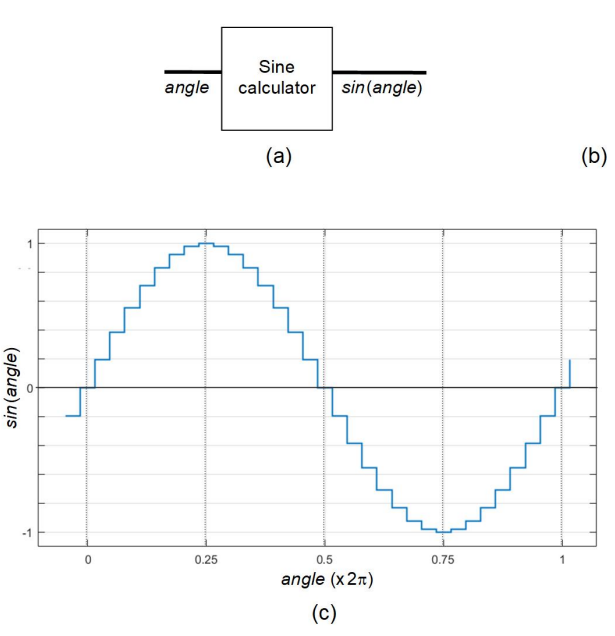
```
1  ----------------------------------------------------------------------------
2  library ieee;
3  use ieee.std_logic_1164.all;
4  use ieee.numeric_std.all;
   use ieee.math_real.all;
5
6  entity fir_filter is
7     generic (
8        NUM_COEF: natural := 11;    --number of filter coefficients
9        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  ---------------------------------------------------------------------------
```

3.3 Alternative version for *Example 11.3: Sine calculator* (page 270)

This example illustrates how continuous functions and ROM-type memories can be implemented in VHDL. For that, the sine calculator of figure 11.4a is constructed, which has *angle* (any integer in the 0-to-360 range) as input and *sin(angle)* as output.

This kind of design relies on two parameters: the number of coefficients employed to represent the sine wave and the number of bits used to represent each coefficient. A corresponding table can be easily derived manually or using a tool like Matlab, as illustrated for the latter in figure 11.4b, with 32 samples per period (=8 per quadrant). The last column shows the version with integers; since 10 bits are employed, they vary from $-(2^9-1) = -511$ (representing –1) to $2^9-1 = 511$ (representing +1). The way the data should be interpreted is illustrated in figure 11.4c. Because there is no relationship between the number of samples and the number of input values, a conversion from one range to the other is needed.



(b)

Matlab sine table for 32 samples/period.

Address (*addr*)	Float 64 bits	Fixed 10 bits	Integer 10 bits ($2^9 \times$ Float)
0	0	0	0
1	0.195090322016128	0.195312500	100
2	0.382683432365090	0.382812500	196
3	0.555570233019602	0.554687500	284
4	0.707106781186548	0.707031250	361
5	0.831469612302545	0.832031250	425
6	0.923879532511287	0.923828125	472
7	0.980785280403230	0.980468750	501
8	1	0.998046875	511
9	0.980785280403230	0.980468750	501
10	0.923879532511287	0.923828125	472
11	0.831469612302545	0.832031250	425
12	0.707106781186548	0.707031250	361
13	0.555570233019602	0.554687500	284
14	0.382683432365090	0.382812500	196
15	0.195090322016129	0.195312500	100
16	0	0	0
17	-0.195090322016128	-0.195312500	-100
18	-0.382683432365090	-0.382812500	-196
19	-0.555570233019602	-0.554687500	-284
20	-0.707106781186548	-0.707031250	-361
21	-0.831469612302545	-0.832031250	-425
22	-0.923879532511287	-0.923828125	-472
23	-0.980785280403230	-0.980468750	-501
24	-1	-1	-511
25	-0.980785280403230	-0.980468750	-501
26	-0.923879532511287	-0.923828125	-472
27	-0.831469612302546	-0.832031250	-425
28	-0.707106781186548	-0.707031250	-361
29	-0.555570233019602	-0.554687500	-284
30	-0.382683432365090	-0.382812500	-196
31	-0.195090322016129	-0.195312500	-100

Figure 11.4. Sine calculator of example 11.3.

A decision to be made here is whether to store in the ROM only the samples for one quadrant (saving memory) or for all four quadrants (reducing the need for comparators, improving the speed and also saving hardware). The VHDL implementation below employs the former (harder to do). The number of samples is 32 per quadrant (line 4) and the number of bits to represent each sample is 10 (line 5). *SINE_TABLE* (lines 15-19) is the local ROM-type memory that holds the 32+1 sample values

(it could be 32 values, but that would add another comparator). Notice that, for clarity, integers were employed in the circuit ports (it is left to the reader to change them to std_logic_vector). Corresponding simulation results are depicted in figure 11.5.

```
1  ---------------------------------------------------------------------------------
2  entity sine_calculator is
3     generic (
4        NUM_COEFF: natural := 32; --Attention: number of coefficients per quadrant
5        NUM_BITS: natural := 10);
6     port (
7        angle: in natural range 0 to 360;
8        sine: out integer range -2**NUM_BITS to 2**NUM_BITS-1);
9  end entity;
10
11 architecture with_sine_rom of sine_calculator is
12
13    type integer_array is array (0 to NUM_COEFF) of natural range 0 to 2**NUM_BITS-1;
14
15    constant SINE_TABLE: integer_array := (
16          0,  25,  50,  75, 100, 124, 148, 172,
17        196, 218, 241, 263, 284, 304, 324, 343,
18        361, 379, 395, 410, 425, 438, 451, 462,
19        472, 481, 489, 496, 501, 505, 509, 510, 511);
20
21 begin
22
23    with angle select
24       sine <=
25           SINE_TABLE((NUM_COEFF*(angle+1))/90) when 0 to 90,
26           SINE_TABLE((NUM_COEFF*(181-angle))/90) when 91 to 180,
27          -SINE_TABLE((NUM_COEFF*(angle-179))/90) when 181 to 270,
28          -SINE_TABLE((NUM_COEFF*(361-angle))/90) when others;
29
30 end architecture;
31 ---------------------------------------------------------------------------------
```

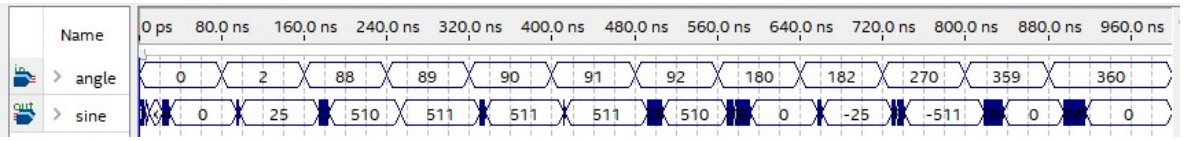


Figure 11.5. Simulation results from example 11.3.