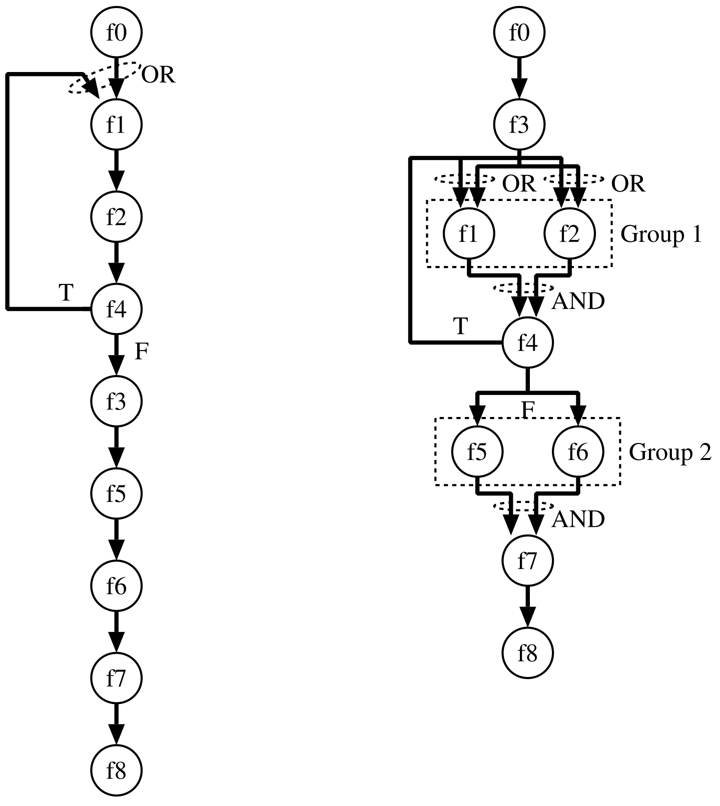
Chapter 13

Exercise 13.1

Graphs for cases (a) and (b) are shown in Figure **??**.



(a) sequential execution graph

(b) group sequential execution graph (max of 2 operations per group)

Figure 13.1: Execution Graphs for Exercise 13.1

The execution time for the concurrent graph, considering that the test done by node f_4 results true for n occurrences (and false after that), is computed as:

$$T_{conc} = 3T + 2nT + 2T = (5 + 2n)T$$

where $3T$ corresponds to the time to execute the path f_0 to f_4, f_5, and f_6. Each iteration that results from a True test at f_4 has a duration of $2T$, which is also the execution time of the last two tasks to finish the computation (f_7 and f_8).

The execution time of the sequential solution shown in Figure **??** (a), for the same loop condition used for the concurrent case is:

$$T_{seq} = 4T + 3nT + 5T = (9 + 3n)T$$

where $4T$ corresponds to the time to executed the path f_0 to f_4 for the first time. Each iteration takes $3T$ and the last sequence of tasks takes $5T$. **Exercise 13.3**

The algorithm presented in Example 13.7 is based on the recurrence:

$$z_{i+1} = z_i(2 - xz_i)$$

The algorithm stops when $|z_i x - 1| < 0.5\epsilon$.
When the inputs for the algorithm are:

$$x = \frac{2}{3} \text{ and } \epsilon = 10^{-5}$$

each variable will have the values shown in the next table:

step j	z_j	Error
0	1	
1	1.3333333	0.111111
2	1.4814814	0.001234
3	1.4997714	0.000015
4	1.4999999	2×10^{-8}

After 4 iterations, the result has an error that is smaller than 5×10^{-6}. **Exercise 13.5**

We want to compute $z = \sqrt{a}$ using the recurrence equation:

$$z_{i+1} = \frac{1}{2}(z_i + \frac{a}{z_i})$$

Since $z^2 = a$, the stop condition is defined as $z^2 - a < \epsilon$, where ϵ is the maximum error in the final result. The execution graph is presented in Figure **??**.
The VHDL code for the algorithm is given on Figure **??**.

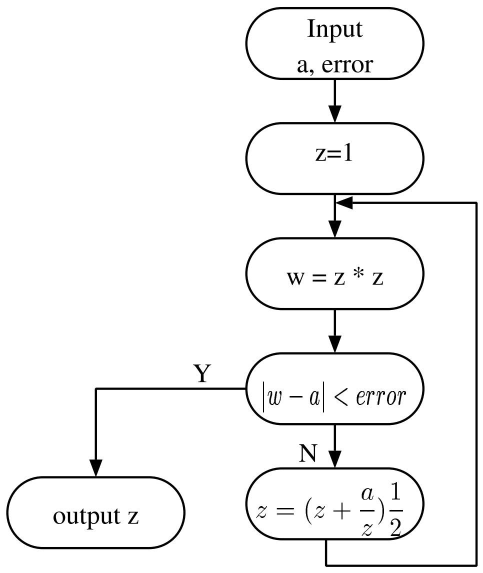


Figure 13.2: Execution graph - Exercise 13.5

Exercise 13.7

Execution graph of the algorithm that obtains the maximum value among n positive integers. All n integers are stored in a vector V. The vector is scanned from position 0 to position $n-1$. In this process, the maximal value is obtained. The computation graph is shown in Figure **??**.

```
-- VHDL code for MAX of N integers (in vector V) - Exercise 13.7
PACKAGE max_pkg IS
 CONSTANT N: INTEGER := 5;
 TYPE DatainT IS ARRAY (N-1 DOWNTO 0) OF INTEGER;
END max_pkg;
USE work.max_pkg.ALL;
ENTITY max IS
PORT ( V: IN DatainT := (-10,12,4,-2,34);
       M: OUT INTEGER;
       CLK: IN BIT);
END max;
ARCHITECTURE high_level OF max IS
BEGIN
 PROCESS (CLK)
 VARIABLE maxv: INTEGER;
 VARIABLE i: INTEGER RANGE 0 TO N;
 BEGIN
  IF  (CLK'event AND CLK='1') THEN
   maxv := V(0); i:= 1;
   WHILE (i < V'length) LOOP
    IF (maxv < V(i)) THEN maxv := V(i);
    END IF;
    i:=i+1;
   END LOOP;
   M <= maxv;
  END IF;
 END PROCESS;
```

```
-- VHDL code for Square-root computation - Exercise 13.5
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY sqroot IS
PORT (a_in: IN REAL RANGE 0.5 TO 1.0;
      Error: IN REAL RANGE 0.0 TO 0.1;
      z_out: OUT REAL RANGE 0.7 TO 1.0;
      CLK: IN BIT);
END sqroot;

ARCHITECTURE high_level OF sqroot IS
BEGIN
 PROCESS (CLK)
  VARIABLE a,w: REAL RANGE 0.5 TO 1.0;
  VARIABLE z: REAL RANGE 0.7 TO 1.0;
 BEGIN
   IF (CLK'event AND CLK='1') THEN
     z := 1.0;
     a:= a_in;
     w := 1.0;
     WHILE (ABS(w-a) > Error) LOOP
        z:=(a/z+z)/2.0;
        w:=z*z;
     END LOOP;
     z_out <= z;
   END IF;
 END PROCESS;
END high_level;
```

Figure 13.3: VHDL code for SQRT(a) - Exercise 13.5

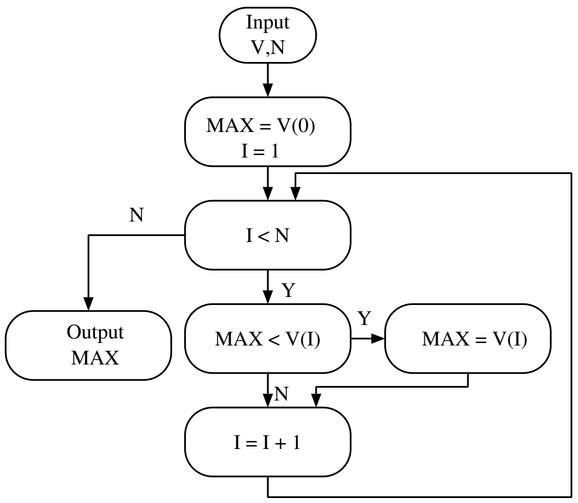


Figure 13.4: Execution graph - Exercise 13.7

```
END high_level;
```

Exercise 13.9

The networks generated for each case is shown in Figure **??**. **Exercise 13.11**

The state diagram for the system operation is shown in Figure **??**.

For $N = 5$ the trace of the computation is shown in the following table:

clock cycle	state	X	I	$ODD(X)$	$I > 0$
0	0	undef.	undef.	undef.	undef.
1	1	5	3	T	T
2	2	26	2	F	T
3	1	26	2	F	T
4	2	13	1	T	T
5	1	13	1	T	T
6	2	66	0	F	F
7	2	66	0	F	F

Thus the final value is $X = 66$.

For $N = 7$ the trace of the computation is shown in the following table:

clock cycle	state	X	I	$ODD(X)$	$I > 0$
0	0	undef.	undef.	undef.	undef.
1	1	7	3	T	T
2	2	36	2	F	T
3	1	36	2	F	T
4	2	18	1	F	T
5	1	18	1	F	T
6	2	9	0	T	F
7	2	9	0	T	F

Thus the final value is $X = 9$. **Exercise 13.13**

Control: Assume that E is connected to CNT input, and ldk is connected to the $LOAD$ input of the mod-4 counter. Based on these assumptions and the networks presented in Figure 13.31 of the text we obtain the following expressions:

$$\begin{aligned} E &= (ldk)' \\ ldK &= (A\_7.S\_1) + (c\_out.S\_2) \\ ldC &= clrB = c\_out.S\_2 \\ ldA &= K\_1'.K\_0' \\ S\_1 &= K\_1'.K\_0 \\ S\_2 &= K\_1.K\_0' \\ ldB &= K\_1.K\_0 \end{aligned}$$

The value of ldK for all possible outputs of the counter is given by the following table. This signal will force the counter to load a zero value.

counter state	ldK
0	0
1	A_7
2	c_out
3	0

Observe that the counter will be forced to zero in state 1 if A_7=1, or in state 2 if c_out=1. The state diagram for the control part is shown in Figure **??**.

The high-level description of the system in VHDL follows. The signals A, B, and C represent the output of each register.

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

entity p13_13 is
port (X: in STD_LOGIC_VECTOR (7 downto 0);
      CLK: in STD_LOGIC;
      Y: out STD_LOGIC_VECTOR (7 downto 0));
end p13_13;
architecture arch of p13_13 is
-- process description - Exercise 13.13
SIGNAL A,B,C: STD_LOGIC_VECTOR (7 downto 0):=(OTHERS=>'0');
SIGNAL state: NATURAL RANGE 0 TO 3 := 0;
SIGNAL c_out: STD_LOGIC := '0';
SIGNAL adder_out: STD_LOGIC_VECTOR (8 downto 0);
begin
 adder_out <=  ('0'&A) + ('0'&B);
 c_out <= adder_out(8);
CTR:PROCESS (CLK)
BEGIN
 IF  (CLK'event AND CLK='1') THEN
  CASE state is
   WHEN 0 => A <= X; state <= 1;
   WHEN 1 => IF (A(7) ='1')
               THEN state <= 0;
               ELSE state <= 2;
             END IF;
   WHEN 2 => IF (c_out='1')
              THEN C <= B;
                   B <= "00000000";
                   state <= 0;
              ELSE state <= 3;
             END IF;
   WHEN 3 => B <= adder_out(7 downto 0);
             state <= 0;
  END CASE;
  Y <= C;
 END IF;
END PROCESS;
end arch;
```

The system accumulates all entries with values less than 128, until the accumulated value cannot absorb another input $X < 128$ without overflowing. In this case, the internal accumulator value is registered as output and the internal state is reset. The trace of the system operation for a chosen sequence of inputs is shown in Figure **??**. **Exercise 13.15**

The high-level specification of the 32×32 serial-parallel multiplier for positive integers is given as follows, for $n = log_2 32 = 5$ bits:

Input: $x, y \in \{0, 1, 2, \ldots, 2^n - 1\}$

Output: $z \in \{0, 1, 2, \ldots, 2^{2n} - 2^{n+1} + 1\}$

Function: $z = x \times y$

A block diagram of the data path for the multiplier is shown in Figure **??**. The left multiplexer is used to select among the 4 multiples of X, that means: 0, X, $2X$, and $3X$. To enable the generation of the $3X$ value using the same adder used during the algorithms iterations, some gates were inserted in the path of the selection signals of this multiplexer to force the X value as an output. Another multiplexer was placed at the other input of the adder to receive the $2X$ value. After the value $3X$ is generated, the iterations will be executed.

The VHDL specification of the radix-4 multiplication algorithms is given next:

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

ENTITY radix4_multiplier IS
 GENERIC (n: NATURAL := 8); -- number of bits in the operands
 PORT (start: IN BIT;
       xin,yin: IN UNSIGNED (n-1 downto 0);
       clk: IN BIT;
       zout: OUT UNSIGNED (2*n-1 downto 0);
       done_out: OUT BIT);
END radix4_multiplier;

ARCHITECTURE behav OF radix4_multiplier IS
 TYPE stateT IS (idle,setup,gen3x,active,done);
 SIGNAL state: stateT := idle;
 SIGNAL x,y: UNSIGNED (n-1 downto 0);
 SIGNAL z: UNSIGNED (2*n-1 downto 0);
 SIGNAL XXX: UNSIGNED (n+1 downto 0);
begin
PROCESS (clk)
 VARIABLE zero_2n : UNSIGNED (2*n-1 DOWNTO 0); -- constant 0
 VARIABLE scale: UNSIGNED (n-1 DOWNTO 0);  -- aligning vector
 VARIABLE add_out: UNSIGNED (n+1 DOWNTO 0); -- adder output
 VARIABLE count: NATURAL RANGE 0 TO n;
 BEGIN
   zero_2n := (OTHERS => '0');
   scale := (OTHERS => '0');
   IF (clk'EVENT AND clk='1') THEN
     CASE state IS
       WHEN idle => done_out <= '0';
                    IF (start='1') THEN state <= setup;
                   ELSE state <= idle;
                    END IF;
       WHEN setup => x <= xin; y <= yin; z <= zero_2n; count:=0;
                     state <= gen3x;
       WHEN gen3x => XXX <= ('0'&x(n-1 downto 0)&'0') + ("00"&x);
                     state <= active;
       WHEN active => CASE conv_integer(y(count+1 downto count)) IS
                        WHEN 0 => add_out:="00"&z(2*n-1 downto n);
                        WHEN 1 => add_out:=("00"&z(2*n-1 downto n))+x;
                        WHEN 2 => add_out:=("00"&z(2*n-1 downto n))+('0'&x&'0');
                        WHEN 3 => add_out:=("00"&z(2*n-1 downto n))+XXX;
                        WHEN others => add_out := (others => '0');
                      END CASE;
                      z <=  add_out & z(n-1 downto 2);
                      IF (count /= (n-2))
                        THEN state <= active;
                             count := count + 2;
                        ELSE state <= done;
                             done_out <= '1';
                     END IF;
       WHEN done => IF (start='0')
                     THEN state <= done;
                     ELSE state <= idle;
                          done_out <= '0';
                   END IF;
     END CASE;
   END IF;
 END PROCESS;
zout <= z; -- update zout
END behav;
```

A trace of this code execution for two arbitrary inputs is shown in Figure **??**. The values are represented in octal.

For a 8×8 case, with the operand values $x = 135$ and $y = 115$, this is the result of the algorithm execution:

clk	state	count	X	Y	$3X$	Z
0	idle	x	xxxxxxxx	xxxxxxxx	xxxxxxxxxx	xxxxxxxxxxxxxxxx
1	setup	x	xxxxxxxx	xxxxxxxx	xxxxxxxxxx	xxxxxxxxxxxxxxxx
2	gen3x	0	10000111	01110011	xxxxxxxxxx	0000000000000000
3	active	0	10000111	01110011	0110010101	0000000000000000
4	active	2	10000111	00011100	0110010101	0110010101000000
5	active	4	10000111	00000111	0110010101	0001100101010000
6	active	6	10000111	00000001	0110010101	0110101110010100
7	done	6	10000111	00000000	0110010101	0011110010100101

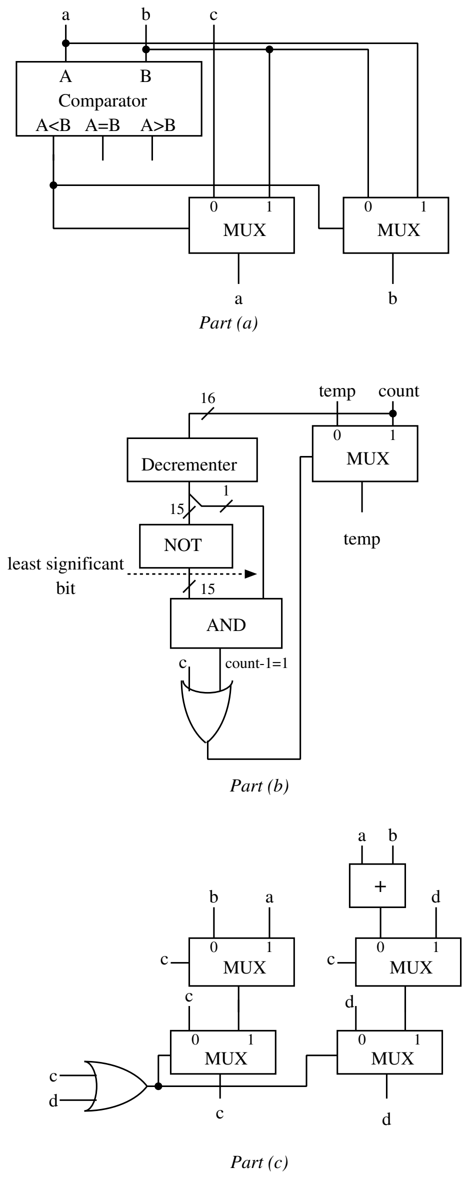


Figure 13.5: Networks for Exercise 13.9

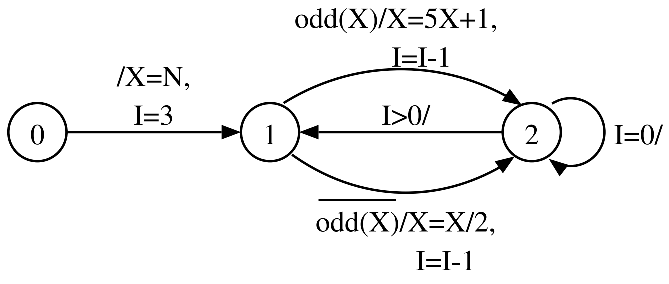


Figure 13.6: State Diagram for the system in Exercise 13.11

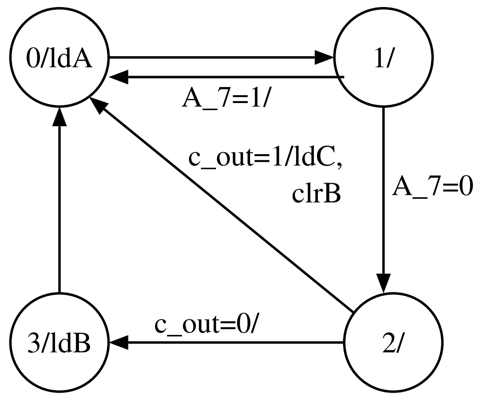


Figure 13.7: State diagram for Exercise 13.13

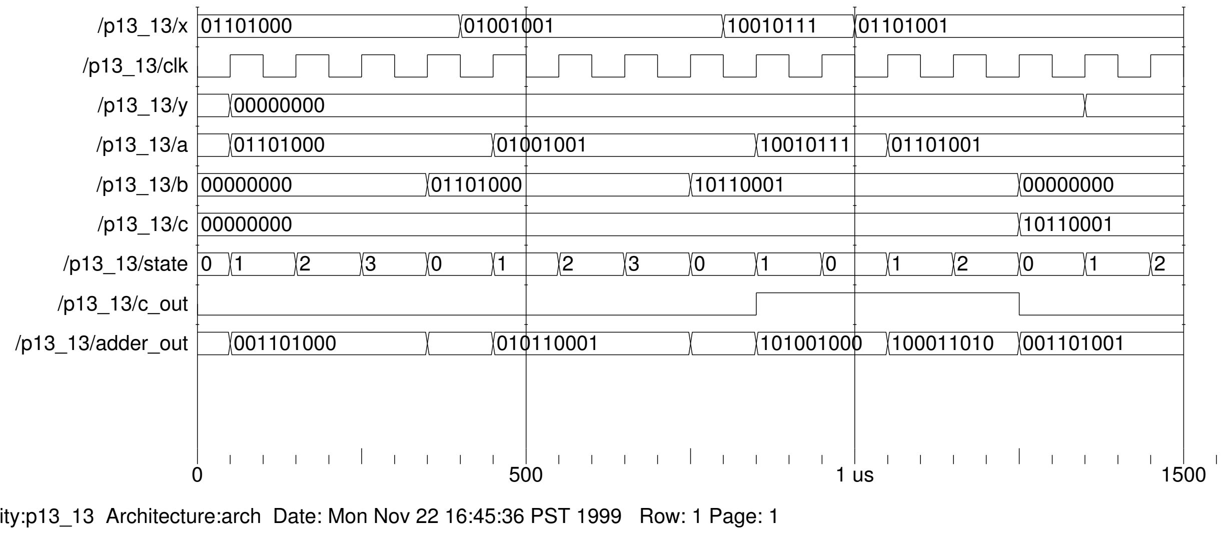
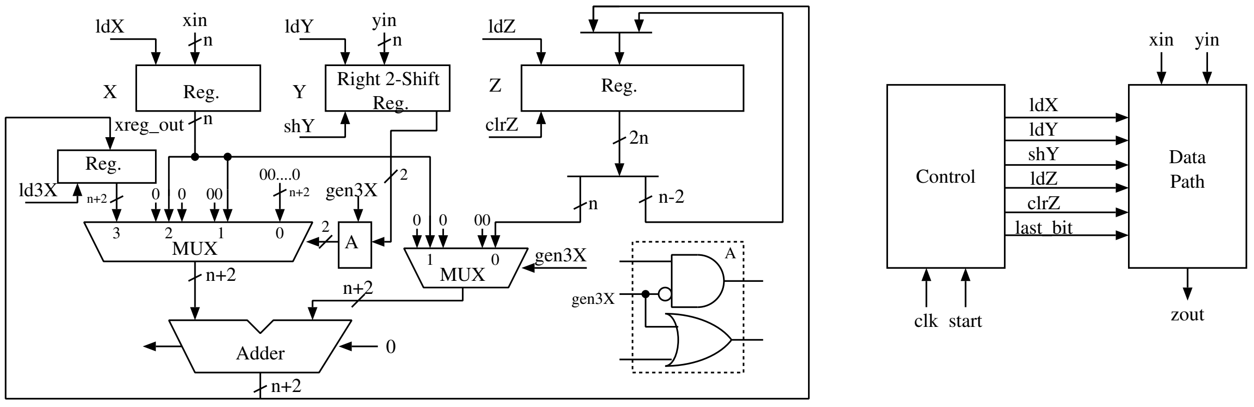


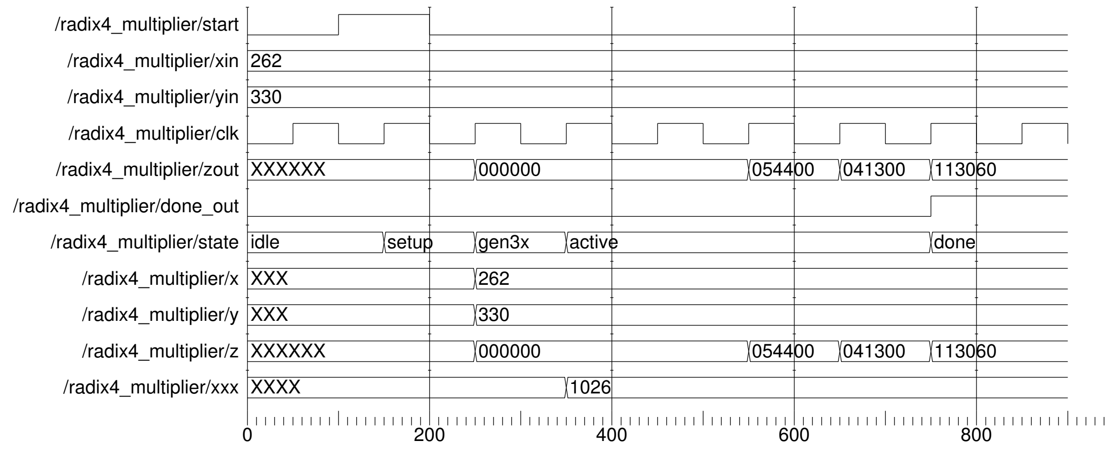
Figure 13.8: Trace of execution - Exercise 13.13



(a) Data path

(b) system's block diagram

Figure 13.9: Block Diagram for the Data path of the radix-4 multiplier



Entity:radix4_multiplier Architecture:behav Date: Sun Nov 21 20:24:21 PST 1999 Row: 1 Page: 1

Figure 13.10: Trace of execution - 8-bit radix-4 multiplier - Exercise 13.15