Behavioral Exploration with RTL Library

Wenwei Pan
Peter Grun
Daniel D. Gajski

Technical Report #96-34
July 29, 1996

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
(714) 824-7063

wpan@ics.uci.edu
pgrun@ics.uci.edu
gajski@ics.uci.edu

Abstract

Behavioral synthesis that takes into consideration real components as well as timing constraints is necessary for the design of today's ASIC chips. In this report, we give a methodology for design space exploration under timing constraints. To illustrate our proposed methodology, we also give several designs that implement a Square Root Algorithm. We compare these designs and give their behavioral and structural description in the Appendix.

Contents

List of Figures

List of Tables

Behavioral Exploration with RTL Library

Wenwei Pan, Peter Grun, Daniel D. Gajski
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425

Abstract

Behavioral synthesis that takes into consideration real components as well as timing constraints is necessary for the design of today's ASIC chips. In this report, we give a methodology for design space exploration under timing constraints. To illustrate our proposed methodology, we also give several designs that implement a Square Root Algorithm. We compare these designs and give their behavioral and structural description in the Appendix.

1 Introduction

In this report, we give a detailed example showing how to design digital systems from behavioral description. The example is a custom ASIC for computing square root. Details of the design as well as the source listing of VHDL code are given in the following sections.

Generally, in the synthesis based design, the algorithm is represented as a Control/Data Flow Graph (CDFG). From CDFG, performing scheduling, allocation and binding, RT level design can be obtained. Different heuristics in each step and different component libraries will have a direct impact on the cost and performance of the final design.

By doing several manual designs we were able to formulate a methodology for exploring the design space.

Section 2 presents the specification of the SRA example, section 3 describes the library components needed. Next in section 4 we show three different designs, and in section 5 we present the methodology. After the conclusions in section 6, in the appendix we show the VHDL code for our optimal design.

2 Example Description

We have designed a custom ASIC which is to compute the square-root approximation (SRA) of 2 signed integers, a and b, by the following formula:

$$\sqrt{a^2 + b^2} \cong max((0.875x + 0.5y), x)$$

where $x = \max(|a|, |b|)$, $y = \min(|a|, |b|)$.

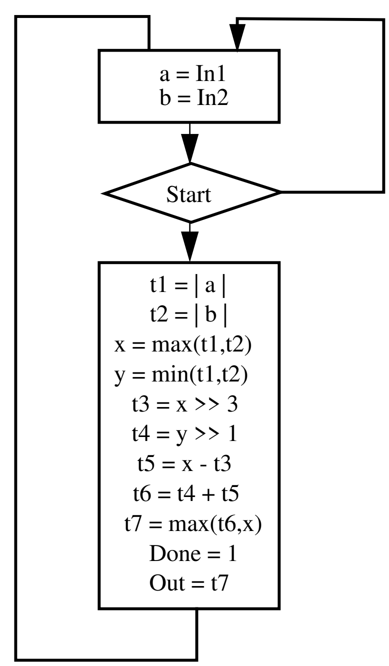


Figure 1: Flowchart of square root algorithm.

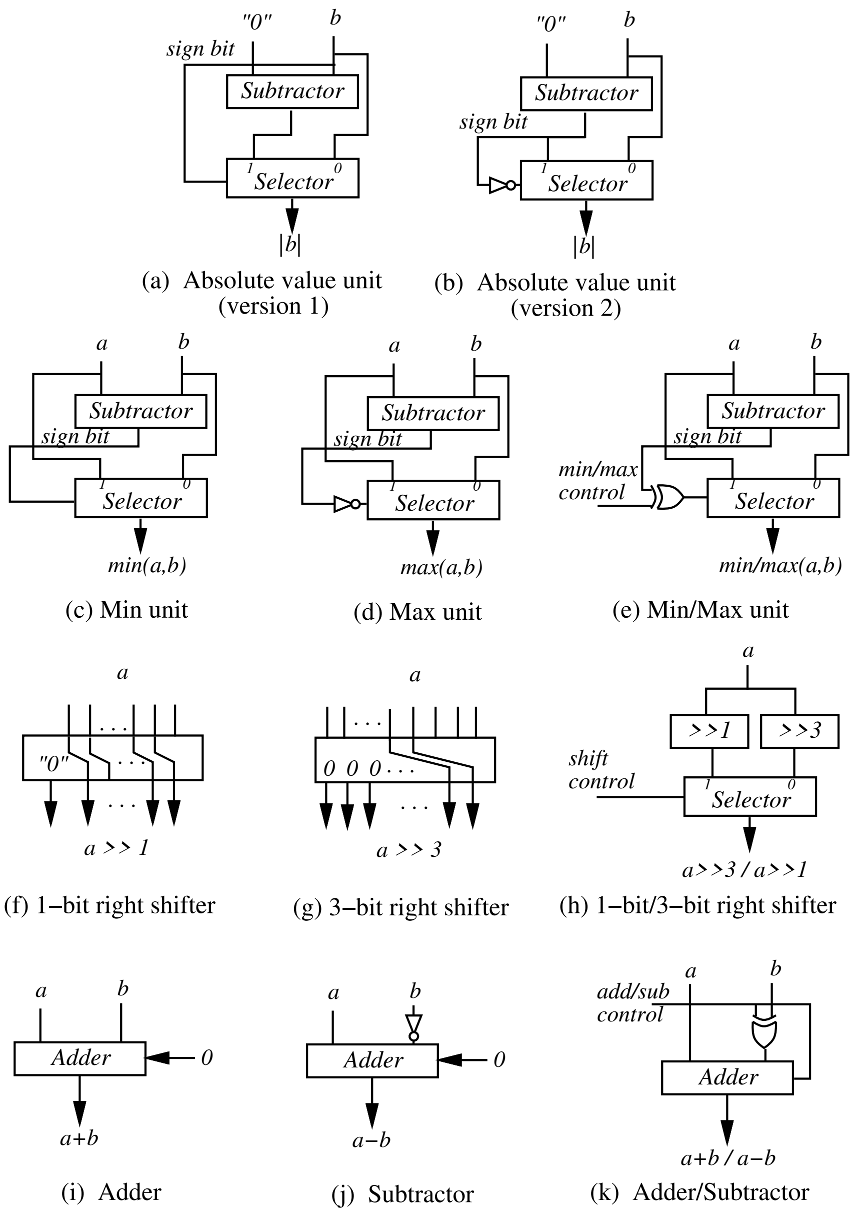
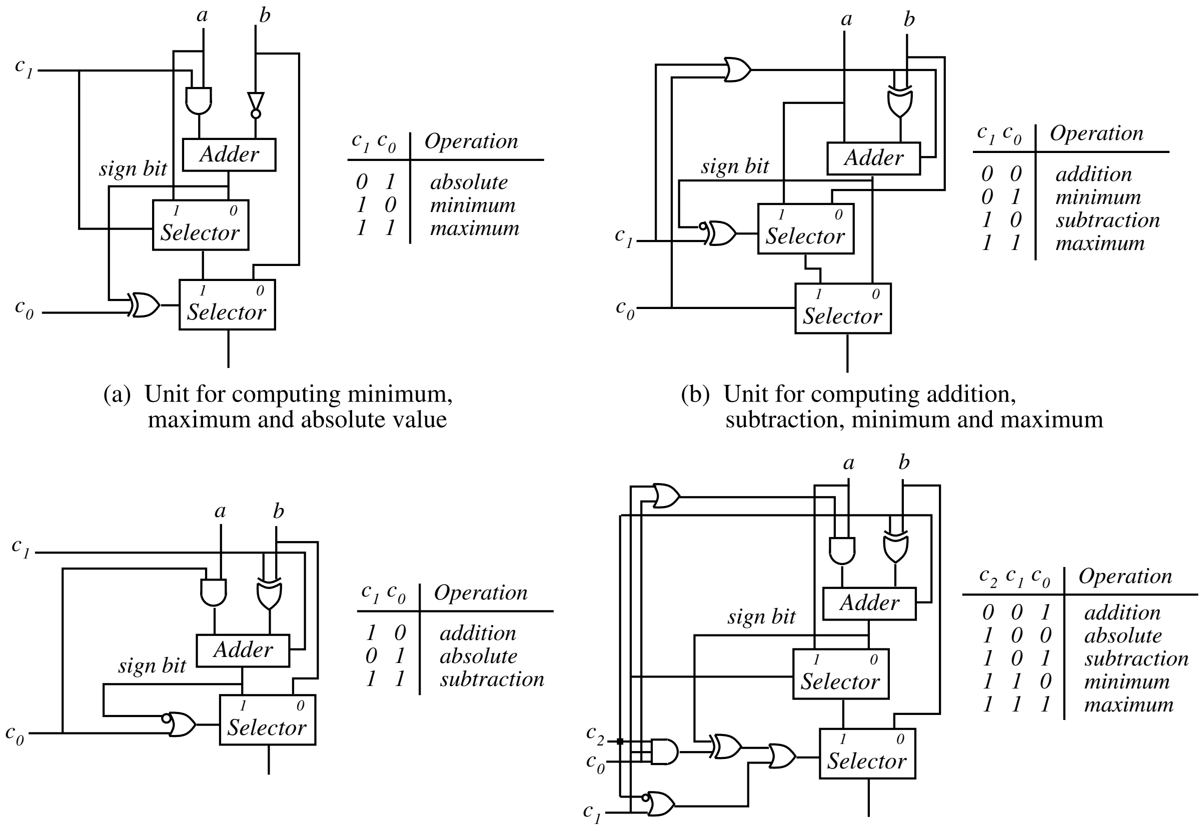


Figure 2: Simple library components.

According to Figure 1, the SRA ASIC has 2 input ports, *In1* and *In2*, which are used to read integers a and b, and one output port *Out* which outputs the result. In the flowchart, the ASIC reads the input ports and starts the computation whenever the input control signal *Start* becomes equal to 1. First it computes the absolute values of a and b and assigns the maximum of these 2 values to x and minimum to y. Then it shifts x three positions to the right to obtain $0.125x$ and y one position to the right to obtain $0.5y$. The ASIC calculates $0.875x$ by subtracting $0.125x$ from x. Next it adds $0.875x$ and $0.5y$, and computes the maximum of x and the expression $0.875x + 0.5y$. Finally the ASIC produces the result and makes it available through the *Out* port for one clock cycle.



(a) Unit for computing minimum, maximum and absolute value

(b) Unit for computing addition, subtraction, minimum and maximum

(c) Unit for computing addition, subtraction, and absolute value

(d) Unit for computing addition, subtraction, minimum, maximum and absolute value

Figure 3: Complex library components.

At the same time, it sets the control signal *Done* to 1, in order to signal to the environment that the data that has appeared at the *Out* port is a valid result.

3 Library Components.

In the implementation of the SRA example we use the library components from [1] (figure 8.12 and 8.21), replicated here for convenience in Figures 2 and 3. We describe 2 units as example. Other units are defined similarly. For brevity, we consider 4 bit versions of the input and output values in the following examples.

(1) Simple library component: *max* unit

Figure 2(d) is a functional unit performing the maximum of the 2 inputs. We present the functionality of this component here as an example:

Inputs: *A, B*: 4-bit 2's complement values
Outputs: *O*: 4-bit 2's complement value
Function: if *A* >= *B, O* = *A*; else *O* = *B*
Example: (1). *A* = 0111 (7), *B* = 0100 (4). A and B are the two inputs to the subtractor. Since the result of the subtraction is 0011, and the sign bit is 0, *A* is selected as output; (2). *A* = 1100 (-4), *B* = 0001 (1). The result of the subtraction is 1011, and the sign bit is 1, therefore, *B* is selected as output;

(2) Complex library component: *abs/min/max* unit

A library of complex functional units from [1] is presented in Figure 3. These units perform combinations of the +, -, *min, max,* and *abs* operations. In order to show how the components from the complex library work, we describe the *abs/min/max* unit from Figure 3(a) below.

Inputs: *A, B*: 4-bit 2's complement values
c1, c0: 1-bit mode control signal
Outputs: *O*: 4-bit 2's complement value
Function:
If *c1c0 = 01*: *O = abs(B)*;
If *c1c0 = 10*: *O = min(A. B);*
If *c1c0 = 11*: *O = max(A, B);*
Example: (1). Assume *c1c0* = 01, *A* = 0111 (7), *B* = 1100 (-4). Since *c1* = 0, the left input to the 4-bit adder is 0000. The right input is 0011 by inverting *B*(1100). So the result of the adder is 0100. Since *c1* = 0 this result is selected as the left input of the output selector. Since the sign bit of the adder result is 0, and *c0* = 1, the control bit for the output selector is 1. Therefore, *O* = 0100 (4) which is equivalent to *abs(B).*

(2). Assume *c1c0* = 10, *A* = 1110 (-2), *B* = 1011 (-5). Since *c1* = 1, the left input to the 4-bit adder is 1110. The right input of the adder is 0100. The result of the adder is 0011. Since the sign bit of this result is 0, and c0 = 0, the control bit of the output selector is 0. Therefore, *O* = *B* = 1011 (-5), which is equivalent to *min(A,B).*

In the Tables 1,2 and 3 we present the delays and the cost of the functional units in the library. The cost is expressed in number of transistors, whereas the delay is in *ns*.

No	Component	Delay from input (ns)	Delay from control (ns)	Cost (trans.)
1	and 2 input	2.4		6
2	and 3 input	2.4		8
3	and 4 input	3.2		10
4	or 2 input	2.4		6
5	or 3 input	2.4		8
6	or 4 input	3.2		10
7	xor 2 input	4.2		14
8	nand 2 input	1.4		4
9	nand 3 input	1.8		6
10	nand 4 input	2.2		8
11	inv	1		2
12	full adder	8.4		30
13	dff	4		18
14	dff with clear	4		26
15	mux 2 to 1	4.8	5.8	14
16	and_ 16	2.4		96
17	or_16	2.4		96
18	xor_16	4.2		224
19	inv_16	1		32
20	Ripple carry adder_16	50.4		494
21	CLA adder_16	21.6		1074
22	buffer_16	2		96
23	register_16	4		512
24	selector2_16	4.8	5.8	224

Table 1: Delays and costs of simple library components

No	Component	Delay from input (ns)	Delay from control (ns)	Cost (trans.)
25	+/-	54.6	54.6	718
26	abs1	56.2		750
27	abs2	57.2		752
28	min	56.2		750
29	max	57.2		752
30	min/max	60.4	10	764
31	shift1	0		0
32	shift3	0		0
33	abs/min/max	62.8	62.8	1084
34	+/-/abs/min/ max	69.6	70.2	1318
35	+/-/abs	63.8	63.8	1046
36	+/-/min/max	70.4	72.8	1188

Table 2: Delays and costs of complex library components using ripple-carry adder

No	Component	Delay From Input (ns)	Delay From Control (ns)	Cost (trans.)
25	+/-	25.8	25.8	1298
26	abs1	27.4		1330
27	abs2	28.4		1332
28	min	27.4		1330
29	max	28.4		1332
30	min/max	31.2	10	1344
31	shift1	0		0
32	shift3	0		0
33	abs/min/max	33.6	33.6	1664
34	+/-/abs/min/ max	40.4	40.8	1898
35	+/-/abs	34.6	34.6	1626
36	+/-/min/max	41	43.4	1768

Table 3: Delays and costs of complex components using CLA adder.

Table 3: Delays and costs of complex components using CLA adder.

Table 1 shows the simple components from the library (the components numbered 1 to 15 are 1-bit components, whereas 16 to 24 are 16-bit functional units and registers).

Table 2 shows the cost and delay for the complex functional units using the ripple-carry adder, whereas Table 3 shows the same components with 2-level CLA adder.

In the third/fourth column we show the delays from the data-inputs/control-inputs to the output. This is the worse case delay for single signal change from any input to any output.

For area we use 1 literal/2 transistors estimation for the basic gates (*nand, nor, inverter*), which we use subsequently in the rest of the components.

4 Design Space Exploration.

The problem addressed is starting from a behavioral description to generate RTL structural implementations of the algorithm satisfying a timing constraint, using RTL components from a library.

By using different components from a library, we can obtain a large spectrum of implementations for the given behavioral description. Intuitively, if we allocate more functional units (more potential for parallelism), or faster units, we can increase the speed. If the speed requirements are not important, we can use less or slower functional units, and obtain a cheaper implementation. Based on the requirements of a specific application, one can choose the implementation which best fits it's needs.

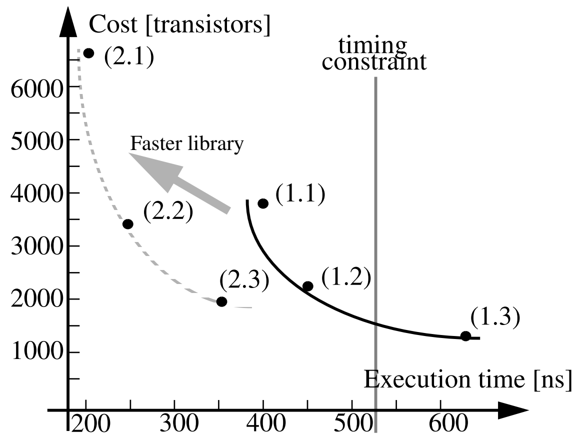


Figure 4: Design space exploration.

In the following we will present the way we derive two sets of three different implementations for the SRA algorithm: the first three of them are derived using the library of components shown in Tables 1 and 2 (ripple-carry adder version), and the next 3 use the library of components summarized in Tables 1 and 3 (carry-look-ahead adder version). In each set, the first implementation maximizes the speed of the resulting chip, the second one shows the best cost/performance trade-off, and the third implementation represents the lowest cost solution.

The points (1.1), (1.2) and (1.3) in Figure 4 represent the ripple-carry implementations, whereas the points (2.1), (2.2) and (2.3) show the CLA implementation in terms of *cost* and *execution time*. The best design in terms of both cost and performance should have minimal product of *cost* and *execution t*ime.

We tried many implementations but for the sake of brevity we only show 6 of them. In the following we show the three schedules generated for the slower library, and by only changing the cost and delay of the adders used, we recompute the cost and execution time of the implementations under the faster library.

In the Following three cases, we use the formula ***execution time*** = *clock cycle* x *number of states* to compute the execution time.

Case 1. Fastest schedule and allocation.

Starting from the flowchart representing the SRA algorithm, we derive the CDFG. In order to generate the fastest implementation, we assume initially that each operation is performed by the fastest component in the library. Later on, we optimize the description, by

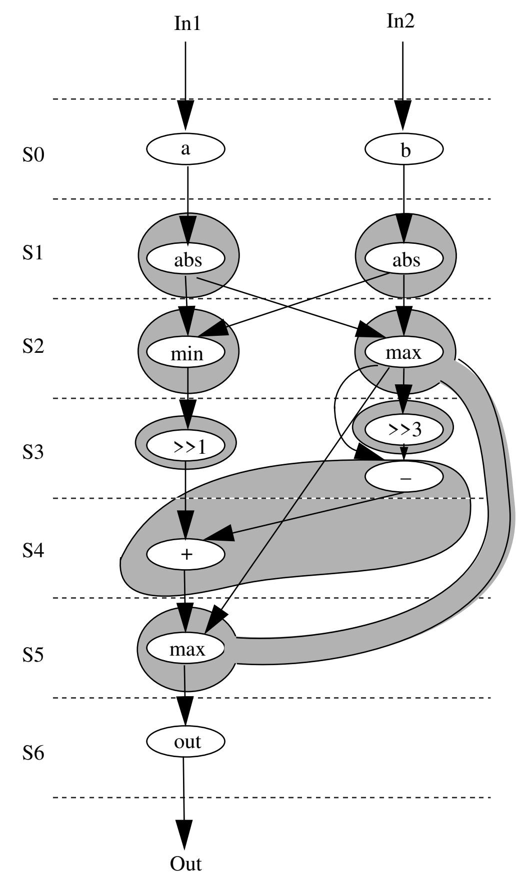


Figure 5: SRA schedule 1.

allowing operations that are not on the critical path to be performed by slower components, since no performance is lost by doing this. Also components which can be merged together without increasing clock cycle and the number of control steps (because they are not on the critical path, or the clock cycle is already determined by even slower components) are considered in order to lower the cost.

Knowing the smallest delays for implementing each operator in the CDFG (according to the library), we can determine the critical path. Obviously, the biggest of these delays determines the clock cycle (we don't consider multicycling at this step).

Based on the clock cycle we first schedule the operations on the critical path. Starting with the first operation in the CDFG we assign each node to the first state available. Consecutive operations which have the sum of delays smaller than the clock cycle can be assigned to the same state, provided the dependences allow it. The schedule generated for case 1 is shown in Figure 5.

The operations not on the critical path have the freedom to be moved to different states. This freedom can be used to optimize the usage of components, allowing us to lower the cost of the ASIC implementation (as long as the clock cycle is not affected). If two operations are not in the same state in the schedule, and there exists a functional unit in the library which can perform both of them, we say that we merge them by allocating the same library component to execute them. In our example we can merge together the + and - operations, because the delay of the library component for +/- is not longer than the clock cycle computed so far, and by doing this we decrease the cost of the implementation. Therefore in this case we will have the *max* library component implementing the *max* operators from the CDFG, the *min* for the *min* operators, 2 *abs* library components, and one +/- component. The shadings in the figures show the clustering of the operation nodes as the result of merging. For each cluster of operations we allocate one functional unit.
In the following, we show the cost and execution time for the designs. For the execution time we use the formula ***execution time*** = *clock cycle* x *number of states*.

(1.1) Slow library (ripple-carry adder)

cost = *1* x *Cost(min)* + *1* x *Cost(max)* + *2* x *Cost(abs)* + *1* x *Cost(+/-)* = *3720 transistors*
clock cycle = *57.2 ns*
number of states = *7*
execution time = *57.2* x *7* = *400.4 ns*

(1.2) Fast library (CLA adder)

cost = *1* x *Cost(min)* + *1* x *Cost(max)* + *2* x *Cost(abs)* + *1* x *Cost(+/-)* = *6620 transistors*
clock cycle = *28.4 ns*
number of states = *7*
execution time = *28.4* x *7* = *198.8 ns*

Case 2. Optimal cost/performance trade-off.

In order to get the best cost/performance trade-off, different schedules and merging of operations have to be attempted. By an iterative improvement technique, we can merge more operations in the CDFG, compromising the clock cycle and/or the number of states in the schedule against substantial cost improvements. If the cost improvement is higher than the performance loss, we obtain an overall cost/performance improvement.

As previously stated, we could get better cost/performance ratio, by trading off some performance loss against an improvement in area. To decrease the cost of the implementation we try to merge more operators into functional units. This additional merging will increase slightly the clock cycle.

We observe that the *>>1* and *>>3* operations have 0 delay, therefore, by chaining them with other operations would not affect the performance. On the other hand we can see that if we move the *min* operation from state S2 to state S3, we will be able to merge it with the *max* operation and use the same functional unit, which has a slightly higher delay, but it generates a good improvement in cost.

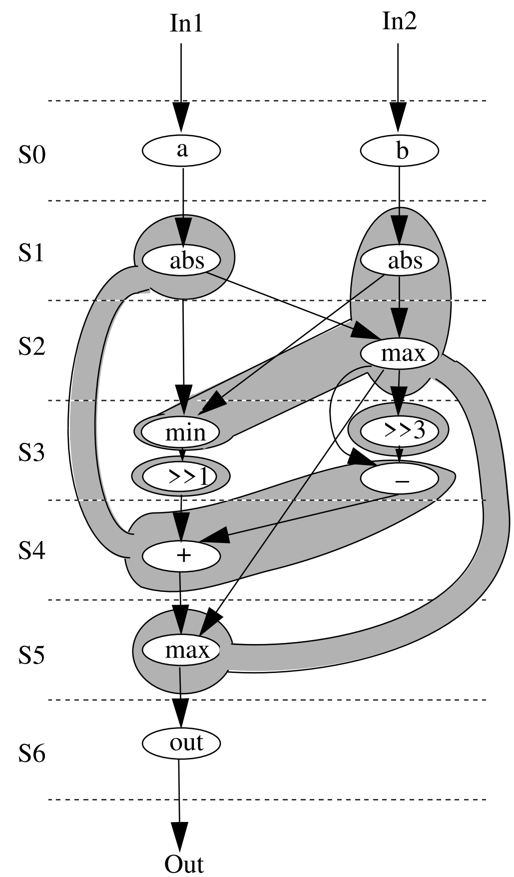


Figure 6: SRA schedule 2.

Figure 6 shows the final schedule for this implementation. The final allocation is one *abs/min/max* unit, and one *abs/+/- unit*, and this results in:

(2.1) Slow library (ripple-carry adder)

cost = *1* x *Cost(abs/min/max)* + *1* x *Cost(abs/+/-)* = *2130 transistors*
clock cycle = *63.8 ns*
number of states = *7*
execution time = *63.8* x *7* = *446.6 ns*

(2.2) Fast library (CLA adder)

cost = *1* x *Cost(abs/min/max)* + *1* x *Cost(abs/+/-)* = *3290 transistors*
clock cycle = *35 ns*
number of states = *7*
execution time = *35* x *7* = *245 ns*

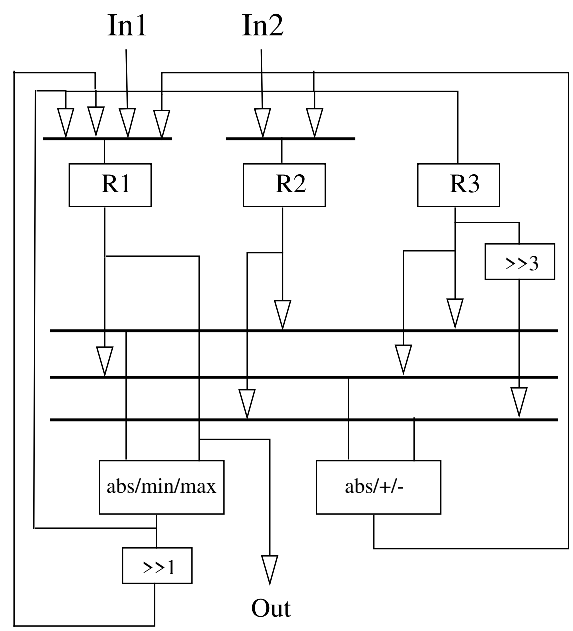


Figure 7: Datapath schematic

After scheduling and functional unit allocation, storage and interconnect allocation is done, generating the final RTL netlist. We present the datapath schematic for the case 2 in Figure 7.

Case 3. Minimal cost schedule and allocation.

In order to create the lowest cost implementation, we have to use the cheapest combination of functional units which still perform the desired operations.

In our case the *abs*, *min*, *max,* + and - operations have to be performed (besides the shifts which have cost 0). The best combination in this case is to use only one functional unit, the *+/-/abs/min/max* which has the cost of 1318 transistors.

When using only one functional unit, we cannot perform in the same state 2 different operations, therefore the schedule gets longer. Figure 8 shows that we need 9 states to perform the computation.

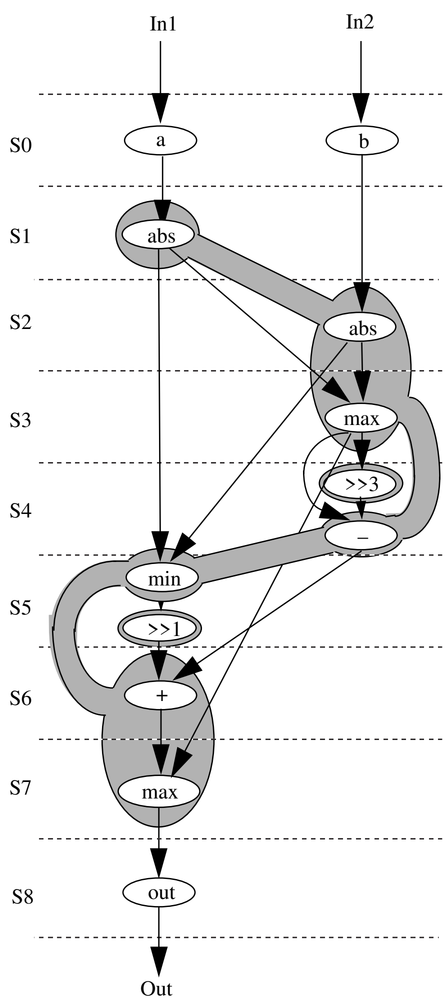


Figure 8: SRA schedule 3.

(3.1) Slow library (ripple-carry adder)

cost *= 1* x *Cost(+/-/abs/min/max) = 1318 transistors*
clock cycle *= 69.6 ns*
number of states *= 9*
execution time *= 69.6* x *9 = 626.4 ns*

(3.2) Fast library (CLA adder)
cost *= 1* x *Cost(+/-/abs/min/max) = 1898 transistors*
clock cycle *= 40.8 ns*
umber of states *= 9*
execution time *= 40.8* x *9 = 367.2 ns*

In Figure 4 we can see the differences in terms of cost and performance between the three implementations presented here for each library. By keeping the library of components the same and changing the schedule and allocation, we generate the points on the curves, as shown in the Figure 4. By changing the time and cost characteristics of the library (using faster components) and keeping the schedule and allocation the same, the implementations move along the dotted arrow.

Considering that we have a given time constraint, we can choose the lowest cost implementation which satisfies it.

5 Methodology.

As previously stated, we start with a behavioral specification of an algorithm (possibly in VHDL), and generate a set of implementations of different cost/performance ratios, that satisfy a timing constraint.

Therefore the starting point is an HDL high level description. First, we derive the CDFG representing this description, and we allocate to each operation the fastest component from the library (which implements that operation). Knowing the delays for performing each operation, we can find the critical paths. The clock

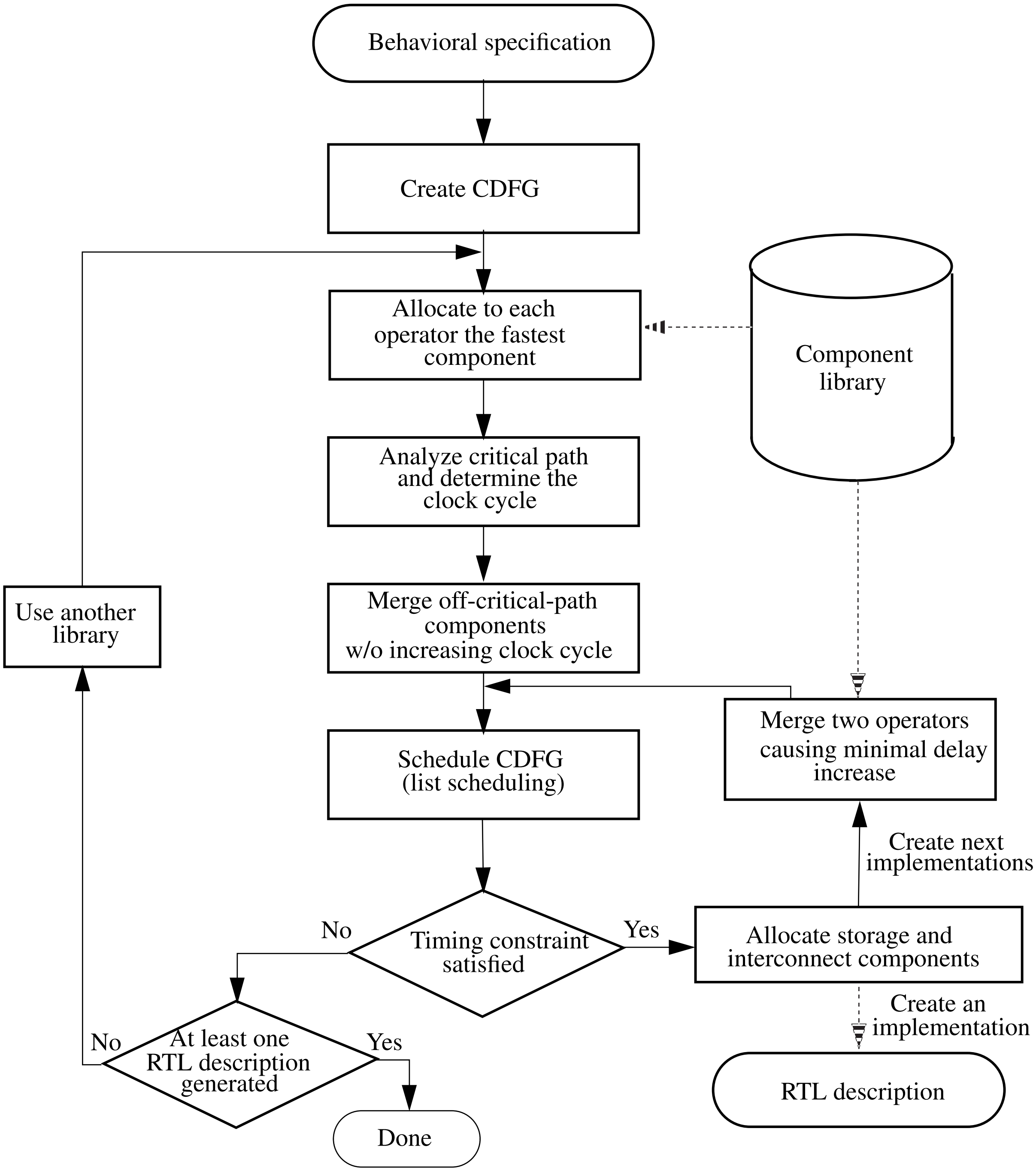


Figure 9: Methodology flowchart.

cycle is determined by the delay of the slowest component (considering that the operation is not performed in multiple cycles, chained with other units or pipelined) allocated to an operation.

We then try to cluster as many operations as possible, as long as we do not affect the clock cycle or the number of nodes on the critical path. For example if 2 operations have to be performed in different states, due to the exist-

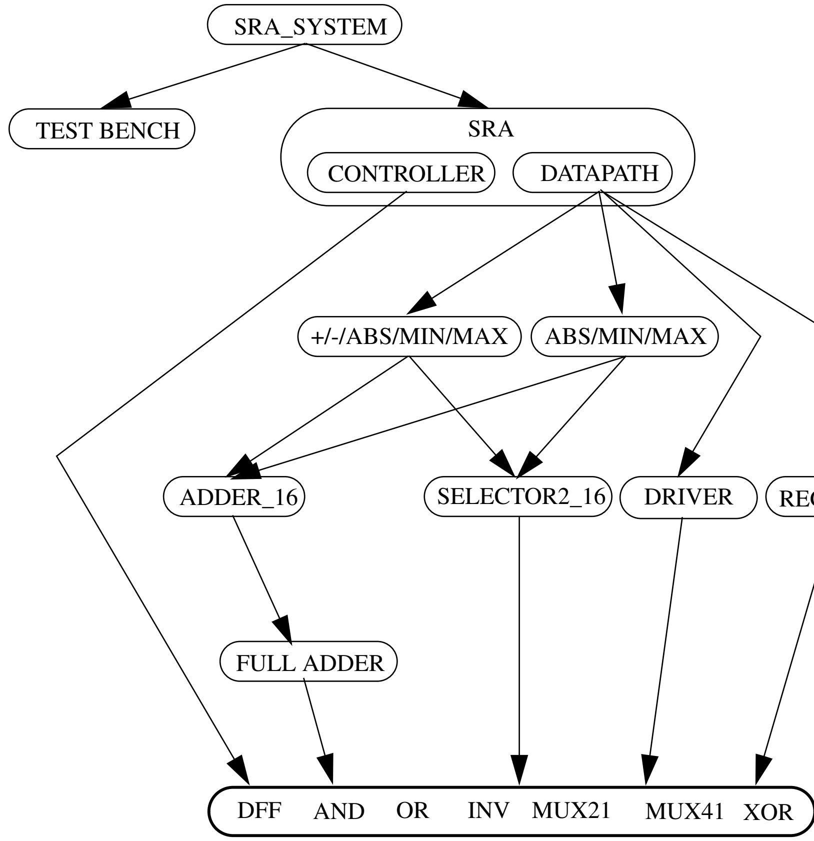


Figure 10: VHDL description hierarchy.

ence of a path between them in the CDFG then by allocating the same functional unit to perform them we do not increase the length of the schedule (the number of states needed).

We still have to make sure that we do not increase the clock cycle. This is true if the delay of the functional unit which performs both the operations is not greater than the delay of the slowest functional unit allocated so far to any other operation. In our example, all these conditions hold for the + and - operations (see Figure 5). Therefore we are able to group the + and - operations without compromising performance against the decrease of cost (the area of the unit which performs both + and - is less than the sum of the areas of the unit for + and the unit for -).

At this point we have a valid functional unit allocation, and we can generate a schedule using list scheduling. The allocated and scheduled CDFG corresponds to the fastest implementation, using the given library. If this implementation doesn't satisfy the timing constraint we have to use a library with faster components. If the implementation satisfies the constraint, we can allocate storage and interconnect to generate a complete RTL level implementation of the algorithm.

In order to generate the next implementations we will merge at each step the 2 operations which create the smallest performance degradation measured by the product of *clock cycle* and *number of states*. This allows us to trade off performance against the cost of the implementation. We keep on doing this until we pass the timing constraint. At each iteration of the methodology flowchart a new point in the cost/ execution time space is generated (see Figure 4). All the RTL descriptions generated so far represent points in the design space and are considered for use in the final implementation.

6 VHDL Models Hierarchy.

All the VHDL models are developed hierarchically in a bottom up fashion, as shown in Figure 10.

(1). The 1st level of hierarchy consists of the basic gates, *muxes* and *flip flops*. All the VHDL models in this level have only behavioral description. All the higher level components are composed of these basic entities.

The delay information for these gates and flip flops are shown in Table 1.

(2). The 2nd level of hierarchy consists of the 16-bit adders, selectors, bus drivers and registers. They appear as RT level components in the datapath. All the VHDL models in this level have both the behavioral and structural description. The structural VHDL models are simulated first and then the delay information is inserted into the behavioral models.

(3). The 3rd level of hierarchy consists of the complex library components such as *abs/min/max* and *abs/min/max/add/sub* units. They are also used in the datapath. For example, the *abs/min/max* consists of 2 16-bit selectors and 1 16-bit ripple carry adder. We have both behavioral and structural VHDL description for them.

(4). The 4th level of hierarchy consists of the datapath and controller. The datapath model is an RT level structural model. The controller has both behavioral model which generate appropriate control signals every cycle and structural model which is a gate level implementation including next-state logic, output logic and state flip flops. The testbench is also included in this level.

(5). The 5th level of hierarchy simply incorporates the SRA model and the testbench to be simulated.

7 Conclusions.

This report presents a methodology for time constrained functional unit allocation and scheduling, taking into account the area of the implementation. It shows how to perform design space exploration by selecting different library components, and creating different schedules of the CDFG. We also give a method how to achieve the best cost/performance trade-off, providing a good starting point for the next levels of synthesis. Our future plans are to extend this work for pipelined designs.

8 References.

[1] D. D. Gajski, “Principles of Digital Design”, Prentice Hall 1996.
[2] D. D. Gajski, N. Dutt, A. Wu, and S. Lin, “High Level Synthesis: Introduction to Chip and System Design”, Kluwer Academic Publishers, 1992.

9 Appendix.

9.1 SRA System.

```
////////////////////////// SRA system ///////////////////////////////////

-- square root approximation algorithm system - structural description
-- includes the whole system comprised of test bench component and the SRA.

library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_misc.all;
  use ieee.std_logic_arith.all;

entity sqr_system is
end sqr_system;

architecture schematic of sqr_system is

  signal      done  : std_logic;
  signal   sqr_out : std_logic_vector(15 downto 0);
  signal      reset : std_logic;
  signal          clk : std_logic;
  signal      start  : std_logic;
  signal         in2  : std_logic_vector(15 downto 0);
  signal         in1  : std_logic_vector(15 downto 0);

  -- test bench component
  component tb
    port (    done  : in     std_logic;
           sqr_out : in     std_logic_vector(15 downto 0);
                 clk : out   std_logic;
                in1  : out   std_logic_vector(15 downto 0);
                in2  : out   std_logic_vector(15 downto 0);
              reset : out   std_logic;
              start  : out   std_logic );
  end component;

  -- square root approximation component
  component sqr
    port (     clk  : in    std_logic;
              in1   : in    std_logic_vector(15 downto 0);
              in2   : in    std_logic_vector(15 downto 0);
             reset : in    std_logic;
             start  : in    std_logic;
             done  : out   std_logic;
         sqr_out  : out   std_logic_vector(15 downto 0) );
  end component;
  for all: sqr use entity work.sqr( schematic_optimal);

begin
  i_tb : tb
          port map ( done=>done, sqr_out=>sqr_out, clk=>clk, in1=>in1,
           in2=>in2, reset=>reset, start=>start );
  i_sqr : sqr
          port map ( clk=>clk, in1=>in1, in2=>in2, reset=>reset,
           start=>start, done=>done, sqr_out=>sqr_out );
end schematic;

configuration cfg_sqr_system_schematic of sqr_system is
  for schematic
  end for;
end cfg_sqr_system_schematic;
```

9.2 Test Bench Entity.

```
/////////////////////////// test bench //////////////////////////////////

--test bench for square root approximation algorithm -
behavioral description

library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_misc.all;
  use ieee.std_logic_arith.all;

entity tb is
    port (done   : in     std_logic;
          sqr_out  : in     std_logic_vector(15 downto 0);
               clk    : out   std_logic;
               in1    : out   std_logic_vector(15 downto 0);
               in2    : out   std_logic_vector(15 downto 0);
            reset    : out   std_logic;
             start    : out   std_logic );
end tb;

architecture behavioral of tb is
signal c : std_logic := '0';
constant period : time := 90 ns;
begin
  -- clock generation
  c <= not c after (period / 2.0 ) ;
  clk <= c;

-- test vectors generation
process
  type test_vector is array (1 to 10) of
  std_logic_vector(15 downto 0);
  constant in_1: test_vector:=
    ("1111111111111111",
      "0000000000000001",
      "0000000000000001",
     "0000000000000011",
      "0000000000000101",
      "1111111111111011",
      "1000000000000000",
      "0000000000001001",
      "0000000000000100",
      "0000000000000111"
    );
  constant in_2: test_vector:=
    ("0000000000000001",
      "0000000000000000",
      "0000000000000011",
      "0000000000000011",
      "1111111111110100",
      "1111111111110100",
      "0111111111111111",
      "0000000000011000",
      "0000000000000100",
      "0000000000001000"
    );
begin
     for i in in_1'range loop
       reset <= '0',
                     '1' after 150 ns;
       start  <= '0',
                    '1' after 150 ns,
                    '0' after 300 ns;
       in1 <= in_1(i);
       in2 <= in_2(i);
       wait for 900 ns;
     end loop;
end process;
end behavioral;
```

9.3 SRA Entity.

```
//////////////////////////// SRA entity ///////////////////////////////

-- Square Root Algorithm - behavioral and structural descriptions

library ieee;
   use ieee.std_logic_1164.all;
   use ieee.std_logic_misc.all;
   use ieee.std_logic_arith.all;

entity sqr is
      port (   clk : in     std_logic;
               in1 : in     std_logic_vector(15 downto 0);
               in2 : in     std_logic_vector(15 downto 0);
             reset : in     std_logic;
             start : in     std_logic;
              done : out    std_logic;
           sqr_out : out    std_logic_vector(15 downto 0) );
end sqr;

-- behavioral description of SRA
architecture nonscheduled_behavioral of sqr is
begin
  process(clk,reset)
        variable a,b,t1,t2,t3,t4,t5,t6,t7,x,y : integer;
        variable v_sqr_out: std_logic_vector(15 downto 0);
        variable v_done : std_logic:='0';
  begin
        if (reset = '0') then
                v_done := '0';
                v_sqr_out := "zzzzzzzzzzzzzzzz";
        elsif (clk = '1') and (clk'event) and (start = '1') then
                a := conv_integer(signed(in1));
                b := conv_integer(signed(in2));
                if (a<0) then
                        t1 := -a;
                else
                        t1 := a;
                end if;
                if (b<0) then
                        t2 := -b;
                else
                        t2 := b;
                end if;
                if (t1<t2) then
                        x := t2;
                        y := t1;
                else
                        x := t1;
                        y := t2;
                end if;
                t4 := y /2;
                t3 := x /8;
                t5 := x - t3;
                t6 := t4 + t5;
                if (t6 > x) then
                        t7 := t6;
                else
                        t7 := x;
                end if;
                v_done :='1';
                v_sqr_out := conv_std_logic_vector(t7,16);
        end if;
        if (reset'event) and (reset = '0') then
                done <= v_done after 6.4 ns;
                sqr_out <= v_sqr_out after 8.4 ns;
        end if;
        if (clk'event) and (clk = '1') and (start = '1') then
                done <= v_done after 7*88.8 ns;
                sqr_out <= v_sqr_out after 7*88.8 ns;
        end if;
end process;
end nonscheduled_behavioral;

-- SRA structural description (datapath and controller)
architecture schematic_optimal of sqr is
   signal      n_1 : std_logic;
   signal      n_2 : std_logic;
   signal      n_3 : std_logic;
   signal      n_4 : std_logic;
   signal      n_5 : std_logic;
   signal      n_6 : std_logic;
   signal      n_7 : std_logic;
   signal      n_8 : std_logic;
   signal      n_9 : std_logic;
   signal     n_10 : std_logic;
   signal     n_11 : std_logic;
   signal     n_12 : std_logic;
   signal     n_13 : std_logic;
   signal     n_14 : std_logic;
   signal     n_15 : std_logic;
   signal     n_16 : std_logic;
   signal     n_17 : std_logic;
   signal     n_18 : std_logic;
   signal     n_19 : std_logic;
   signal     n_20 : std_logic;

   component datapath_optimal
      port (     c1 : in    std_logic;
                 c2 : in    std_logic;
                 c3 : in    std_logic;
                 c4 : in    std_logic;
                clk : in    std_logic;
                 en : in    std_logic;
                in1 : in    std_logic_vector(15 downto 0);
                in2 : in    std_logic_vector(15 downto 0);
```

```
            load1 : in    std_logic;
            load2 : in    std_logic;
            load3 : in    std_logic;
               s1 : in    std_logic;
              s10 : in    std_logic;
              s11 : in    std_logic;
              s12 : in    std_logic;
               s2 : in    std_logic;
               s3 : in    std_logic;
               s4 : in    std_logic;
               s5 : in    std_logic;
               s6 : in    std_logic;
               s7 : in    std_logic;
               s8 : in    std_logic;
               s9 : in    std_logic;
         sqr_out : out   std_logic_vector(15 downto 0) );
   end component;

   component controller_optimal
      port (   clk : in    std_logic;
            reset : in    std_logic;
            start : in    std_logic;
               c1 : out   std_logic;
               c2 : out   std_logic;
               c3 : out   std_logic;
               c4 : out   std_logic;
             done : out   std_logic;
               en : out   std_logic;
            load1 : out   std_logic;
            load2 : out   std_logic;
            load3 : out   std_logic;
               s1 : out   std_logic;
              s10 : out   std_logic;
              s11 : out   std_logic;
              s12 : out   std_logic;
               s2 : out   std_logic;
               s3 : out   std_logic;
               s4 : out   std_logic;
               s5 : out   std_logic;
               s6 : out   std_logic;
               s7 : out   std_logic;
               s8 : out   std_logic;
               s9 : out   std_logic );
   end component;

begin
   i_1 : datapath_optimal
     port map ( c1=>n_11, c2=>n_10, c3=>n_9, c4=>n_8,
clk=>clk, en=>n_4,
           in1=>in1, in2=>in2, load1=>n_3, load2=>n_2,
load3=>n_1,
            s1=>n_20, s10=>n_7, s11=>n_6, s12=>n_5,
s2=>n_19,
             s3=>n_18, s4=>n_17, s5=>n_16, s6=>n_15,
s7=>n_14,
            s8=>n_13, s9=>n_12, sqr_out=>sqr_out );
   i_2 : controller_optimal
      port map ( clk=>clk, reset=>reset, start=>start,
c1=>n_11,
             c2=>n_10, c3=>n_9, c4=>n_8, done=>done,
en=>n_4,
              load1=>n_3, load2=>n_2, load3=>n_1,
s1=>n_20, s10=>n_7,
              s11=>n_6, s12=>n_5, s2=>n_19, s3=>n_18,
s4=>n_17,
             s5=>n_16, s6=>n_15, s7=>n_14, s8=>n_13,
s9=>n_12 );
end schematic_optimal;
```

9.4 Datapath Entity.

```
/////////////////////////// Datapath ////////////////////////////////
-- Datapath of Square Root Algorithm - structural description

library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_misc.all;
  use ieee.std_logic_arith.all;
  use work.all;

entity datapath_optimal is
    port (    c1 : in    std_logic;
              c2 : in    std_logic;
              c3 : in    std_logic;
              c4 : in    std_logic;
             clk : in    std_logic;
              en : in    std_logic;
             in1 : in    std_logic_vector(15 downto 0);
             in2 : in    std_logic_vector(15 downto 0);
           load1 : in    std_logic;
           load2 : in    std_logic;
           load3 : in    std_logic;
              s1 : in    std_logic;
             s10 : in    std_logic;
             s11 : in    std_logic;
             s12 : in    std_logic;
              s2 : in    std_logic;
              s3 : in    std_logic;
              s4 : in    std_logic;
              s5 : in    std_logic;
              s6 : in    std_logic;
              s7 : in    std_logic;
              s8 : in    std_logic;
              s9 : in    std_logic;
         sqr_out : out   std_logic_vector(15 downto 0) );
end datapath_optimal;

architecture schematic of datapath_optimal is

  signal    n_11 : std_logic_vector(15 downto 0);
  signal    n_12 : std_logic_vector(15 downto 0);
  signal    n_13 : std_logic_vector(15 downto 0);
  signal    n_14 : std_logic_vector(15 downto 0);
  signal     n_1 : std_logic_vector(15 downto 0);
  signal     n_2 : std_logic_vector(15 downto 0);
  signal     n_3 : std_logic_vector(15 downto 0);
  signal     n_4 : std_logic_vector(15 downto 0);
  signal     n_5 : std_logic_vector(15 downto 0);
  signal     n_7 : std_logic_vector(15 downto 0);
  signal     n_9 : std_logic_vector(15 downto 0);
  signal    n_10 : std_logic_vector(15 downto 0);

  component buffer16
    port (    c : in    std_logic;
              i : in    std_logic_vector(15 downto 0);
              o : out   std_logic_vector(15 downto 0) );
  end component;

  component shift3
    port (    i : in    std_logic_vector(15 downto 0);
              o : out   std_logic_vector(15 downto 0) );
  end component;

  component shift1
    port (    i : in    std_logic_vector(15 downto 0);
              o : out   std_logic_vector(15 downto 0) );
  end component;

  component addsubabs
    port (   c0 : in    std_logic;
             c1 : in    std_logic;
             i1 : in    std_logic_vector(15 downto 0);
             i2 : in    std_logic_vector(15 downto 0);
              o : out   std_logic_vector(15 downto 0) );
  end component;
   for all: addsubabs use entity work.addsubabs( structural);

  component register_16
    port (   clk : in    std_logic;
             din : in    std_logic_vector(15 downto 0);
            load : in    std_logic;
            dout : out   std_logic_vector(15 downto 0) );
  end component;

  component absminmax
    port (   c0 : in    std_logic;
             c1 : in    std_logic;
             i1 : in    std_logic_vector(15 downto 0);
             i2 : in    std_logic_vector(15 downto 0);
              o : out   std_logic_vector(15 downto 0) );
  end component;

begin

  i_20 : buffer16
    port map ( c=>en, i=>n_14, o=>sqr_out );
  i_1 : buffer16
    port map ( c=>s6, i=>n_7, o=>n_4 );
  i_2 : buffer16
    port map ( c=>s5, i=>in2, o=>n_4 );
  i_3 : buffer16
    port map ( c=>s4, i=>n_7, o=>n_5 );
  i_4 : buffer16
    port map ( c=>s3, i=>in1, o=>n_5 );
  i_5 : buffer16
    port map ( c=>s2, i=>n_9, o=>n_5 );
```

```
  i_6 : buffer16
    port map ( c=>s1, i=>n_10, o=>n_5 );
  i_7 : buffer16
    port map ( c=>s12, i=>n_12, o=>n_1 );
  i_8 : buffer16
    port map ( c=>s10, i=>n_13, o=>n_2 );
  i_9 : buffer16
    port map ( c=>s8, i=>n_13, o=>n_3 );
  i_10 : buffer16
    port map ( c=>s11, i=>n_11, o=>n_1 );
  i_11 : buffer16
    port map ( c=>s7, i=>n_11, o=>n_3 );
  i_12 : buffer16
    port map ( c=>s9, i=>n_14, o=>n_2 );
  i_13 : shift3
    port map ( i=>n_13, o=>n_12 );
  i_14 : shift1
    port map ( i=>n_10, o=>n_9 );
  i_15 : addsubabs
       port map ( c0=>c3, c1=>c4, i1=>n_2, i2=>n_1,
o=>n_7 );
  i_reg1 : register_16
        port map ( clk=>clk, din=>n_4, load=>load2,
dout=>n_11 );
  i_reg2 : register_16
       port map ( clk=>clk, din=>n_10, load=>load3,
dout=>n_13 );
  i_reg3 : register_16
        port map ( clk=>clk, din=>n_5, load=>load1,
dout=>n_14 );
  i_19 : absminmax
      port map ( c0=>c1, c1=>c2, i1=>n_3, i2=>n_14,
o=>n_10 );

end schematic;
```

9.5 Abs/min/max Entity.

```
/////////////////// abs/min/max entity //////////////////////////
-- abs/min/max library component - structural and
behavioral descriptions

library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_arith.all;
  use work.all;

entity absminmax is
    port (
        i1 : in   std_logic_vector (15 downto 0);
        i2 : in   std_logic_vector (15 downto 0);
       c0 : in   std_logic;
       c1 : in   std_logic;
        o : out  std_logic_vector (15 downto 0)
);
end absminmax;

architecture behavioral of absminmax is
begin
    process(c0,c1,i1,i2)
        variable a,b,c: integer;
    begin
        a := conv_integer(signed(i1));
        b := conv_integer(signed(i2));
        if (c1='0') and (c0='1') then
            if (b<0) then
                c := -b;
            else
                c := b;
            end if;
        elsif (c1='1') and (c0='0') then
            if (a < b) then
                c := a;
            else
                c := b;
            end if;
        elsif (c1='1') and (c0='1') then
            if (a > b) then
                c := a;
            else
                c := b;
            end if;
        end if;
        if (c0'event) then
            o <= conv_std_logic_vector(c,16) after 10
ns;
        end if;
        if (c1'event) then
                o <= conv_std_logic_vector(c,16) after
62.8 ns;
        end if;
        if (i1'event) then
                o <= conv_std_logic_vector(c,16) after
62.8 ns;
          end if;
        if (i2'event) then
              o <= conv_std_logic_vector(c,16) after 61
ns;
        end if;
end process;
end behavioral;


architecture structural of absminmax is

    signal s1,s2,s3,s4,s5 : std_logic_vector (15 downto
0);
    signal cout,overflow,s6,one: std_logic;

    -- adder_16 is a 16 bit ripple carry adder
    component adder_16
    port (    cin : in     std_logic;
                x : in     std_logic_vector (15 downto 0);
                y : in     std_logic_vector (15 downto 0);
             cout : out    std_logic;
         overflow : out    std_logic;
                s : out    std_logic_vector (15 downto 0) );
    end component;

    -- selector2_16 is a 16 bit 2 to 1 selector
    component selector2_16
    port (      i0 : in     std_logic_vector (15 downto 0);
                i1 : in     std_logic_vector (15 downto 0);
                 s : in     std_logic;
                 o : out    std_logic_vector (15 downto 0) );
    end component;

    -- and_16 is an array of 16 2-input and gates
    component and_16
    port (      i1 : in     std_logic_vector (15 downto 0);
                i2 : in     std_logic_vector (15 downto 0);
                 o : out    std_logic_vector (15 downto 0) );
    end component;

    -- inv_16 is an array of 16 inverters
    component inv_16
    port (       i : in     std_logic_vector (15 downto 0);
                 o : out    std_logic_vector (15 downto 0) );
    end component;

    -- xor2 is a 2 input xor gate
```

```
    component xor2
    port (      i1 : in     std_logic;
                i2 : in     std_logic;
                 o : out   std_logic );
    end component;

     for all : adder_16 use entity work.adder_16( sche-
matic );
     for all : selector2_16 use entity work.selector2_16(
schematic );
     for all : and_16 use entity work.and_16( schematic
);
    for all : inv_16 use entity work.inv_16( schematic );
    for all : xor2 use entity work.xor2( behavioral );

begin
    process(c1) begin
        for i in 0 to 15 loop
            s1(i) <= c1;
        end loop;
    end process;
    one <= ‘1’;
    u1 : and_16 port map(s1,i1,s2);
    u2 : inv_16 port map(i2,s3);
    u3 : adder_16 port map(one,s2,s3,cout,overflow,s4);
    u4 : selector2_16 port map(s4,i1,c1,s5);
    u5 : xor2 port map(c0,s4(15),s6);
    u6 : selector2_16 port map(i2,s5,s6,o);
end structural;
```

9.6 16 Bit Adder Entity.

```
/////////////////////////// 16 bit adder entity /////////////////////

-- 16 bit adder - behavioral and structural descriptions

library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_misc.all;
  use ieee.std_logic_arith.all;

entity adder_16 is
     port ( cin : in     std_logic;
              x : in     std_logic_vector(15 downto 0);
              y : in     std_logic_vector(15 downto 0);
           cout : out    std_logic;
       overflow : out    std_logic;
              s : out    std_logic_vector(15 downto 0) );
end adder_16;

architecture behavioral of adder_16 is
begin
      process(x,y,cin)
            variable sum: std_logic_vector(15 downto 0);
            variable carry: std_logic_vector(16 downto 0);
      begin
            carry(0) := cin;
            for i in 0 to 15 loop
                  sum(i) := x(i) xor y(i) xor carry(i);
                      carry(i+1) := (x(i) and y(i)) or (x(i) and
carry(i))
                                    or (y(i) and carry(i));
            end loop;
            if(cin'event) then
                  s <= sum after 46.2 ns; -- verified
                  cout <= carry(16) after 44.8 ns; -- verified
                    overflow <= carry(15) xor carry(16) after
46.2 ns;
            end if;
            if (x'event) or (y'event) then
                  s <= sum after 50.4 ns; -- verified
                  cout <= carry(16) after 49 ns;
                    overflow <= carry(15) xor carry(16) after
50.4 ns;
            end if;
      end process;
end behavioral;

-- 16-bit ripple carry adder
architecture schematic of adder_16 is

      signal     n_1 : std_logic;
      signal     n_2 : std_logic;
      signal     n_3 : std_logic;
      signal     n_4 : std_logic;
      signal     n_5 : std_logic;
      signal     n_6 : std_logic;
      signal     n_7 : std_logic;
      signal     n_8 : std_logic;
      signal     n_9 : std_logic;
      signal    n_10 : std_logic;
      signal    n_11 : std_logic;
      signal    n_12 : std_logic;
      signal    n_13 : std_logic;
      signal    n_14 : std_logic;
      signal    n_15 : std_logic;
      signal    cout_dummy : std_logic;

      -- adder_1 is a 1 bit full adder whose output is:
        -- s = (x xor y xor cin)
        -- cout = (x and y) or (x and cin) or (y and cin)
      component adder_1
      port (     cin : in     std_logic;
                   x : in     std_logic;
                   y : in     std_logic;
                cout : out    std_logic;
                   s : out    std_logic );
      end component;

      -- xor2 is a 2 input xor gate
      component xor2
      generic ( delay : time := 4.2 ns );
      port (      i1 : in    std_logic;
                  i2 : in    std_logic;
                   o : out   std_logic );
      end component;

begin

   cout <= cout_dummy;

       i_13 : adder_1
                   port map ( cin=>n_6, x=>x(4), y=>y(4),
cout=>n_5, s=>s(4) );
      i_14 : adder_1
                   port map ( cin=>n_7, x=>x(3), y=>y(3),
cout=>n_6, s=>s(3) );
      i_15 : adder_1
                   port map ( cin=>n_8, x=>x(2), y=>y(2),
cout=>n_7, s=>s(2) );
      i_16 : adder_1
                   port map ( cin=>n_9, x=>x(1), y=>y(1),
cout=>n_8, s=>s(1) );
      i_17 : adder_1
                   port map ( cin=>cin, x=>x(0), y=>y(0),
cout=>n_9, s=>s(0) );
      i_18 : adder_1
```

```
            port map ( cin=>n_5, x=>x(5), y=>y(5),
cout=>n_4, s=>s(5) );
    i_19 : adder_1
            port map ( cin=>n_4, x=>x(6), y=>y(6),
cout=>n_2, s=>s(6) );
    i_20 : adder_1
            port map ( cin=>n_2, x=>x(7), y=>y(7),
cout=>n_3, s=>s(7) );
    i_21 : adder_1
            port map ( cin=>n_3, x=>x(8), y=>y(8),
cout=>n_15, s=>s(8) );
    i_22 : adder_1
           port map ( cin=>n_15, x=>x(9), y=>y(9),
cout=>n_14, s=>s(9) );
    i_23 : adder_1
         port map ( cin=>n_14, x=>x(10), y=>y(10),
cout=>n_13, s=>s(10) );
    i_24 : adder_1
         port map ( cin=>n_13, x=>x(11), y=>y(11),
cout=>n_12, s=>s(11) );
    i_25 : adder_1
         port map ( cin=>n_12, x=>x(12), y=>y(12),
cout=>n_11, s=>s(12) );
    i_26 : adder_1
         port map ( cin=>n_11, x=>x(13), y=>y(13),
cout=>n_10, s=>s(13) );
    i_27 : adder_1
         port map ( cin=>n_10, x=>x(14), y=>y(14),
cout=>n_1, s=>s(14) );
    i_28 : adder_1
          port map ( cin=>n_1, x=>x(15), y=>y(15),
cout=>cout_dummy, s=>s(15) );
    i_12 : xor2
        generic map ( delay => 4.2 ns )
       port map ( i1=>cout_dummy, i2=>n_1, o=>over-
flow );

end schematic;
```

9.7 1-Bit Full Adder Entity.

```
/////////////////////// 1 bit adder entity ////////////////////////////

-- 1 bit full adder - behavioral and structural descriptions

library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_misc.all;
  use ieee.std_logic_arith.all;

entity adder_1 is
    port (    cin : in    std_logic;
                x : in    std_logic;
                y : in    std_logic;
             cout : out   std_logic;
                s : out   std_logic );
end adder_1;

architecture behavioral of adder_1 is
begin
    process(cin,x,y) begin
        if (x'event) or (y'event) then
            s <= (x xor y xor cin) after 8.4 ns;
             cout <= (x and y) or (x and cin) or (y and
cin) after 7 ns;
        end if;
        if (cin'event) then
            s <= (x xor y xor cin) after 4.2 ns;
             cout <= (x and y) or (x and cin) or (y and
cin) after 2.8 ns;
        end if;
    end process;
end behavioral;

architecture schematic of adder_1 is

    signal    n_1 : std_logic;
    signal    n_2 : std_logic;
    signal    n_3 : std_logic;

    -- nand2 is a 2-input nand gate
    component nand2
    generic ( delay : time := 1.4 ns );
    port (     i1 : in    std_logic;
               i2 : in    std_logic;
                o : out   std_logic );
    end component;

    -- xor2 is a 2-input xor gate
    component xor2
    generic ( delay : time := 4.2 ns );
    port (     i1 : in    std_logic;
               i2 : in    std_logic;
                o : out   std_logic );
    end component;

begin

    i_6 : nand2
         generic map ( delay => 1.4 ns )
        port map ( i1=>n_3, i2=>n_2, o=>cout );
    i_7 : nand2
         generic map ( delay => 1.4 ns )
        port map ( i1=>n_1, i2=>cin, o=>n_2 );
    i_8 : nand2
         generic map ( delay => 1.4 ns )
        port map ( i1=>y, i2=>x, o=>n_3 );
    i_5 : xor2
         generic map ( delay => 4.2 ns )
        port map ( i1=>cin, i2=>n_1, o=>s );
    i_4 : xor2
         generic map ( delay => 4.2 ns )
        port map ( i1=>y, i2=>x, o=>n_1 );

end schematic;
```

9.8 16-Bit Register Entity.

```
/////////////////////// 16 bit register entity ////////////////////

-- 16 bit register - behavioral and structural descriptions

library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_misc.all;
  use ieee.std_logic_arith.all;

entity register_16 is
      generic( setup: time := 2 ns;
       hold: time := 1 ns);
      port (     clk : in     std_logic;
                 din : in     std_logic_vector(15 downto 0);
                load : in     std_logic;
                dout : out   std_logic_vector(15 downto 0) );
begin
      process(clk)
      begin
           if(clk='1') and (clk'event) and (load = '1') then
                 assert ( din'stable(setup + 4.8 ns) )
                 report "setup time violation !"
                 severity warning;
                 assert ( load'stable(setup + 5.8 ns) )
                 report "setup time violation !"
                 severity warning;
                 assert ( din'stable(4.8 ns - hold) )
                 report "hold time violation !"
                 severity warning;
                 assert ( load'stable(5.8 ns- hold) )
                 report "hold time violation !"
                 severity warning;
            end if;
      end process;
end register_16;


architecture behavioral of register_16 is
      constant delay: time := 4 ns;
begin
      process(clk)
             variable result: std_logic_vector(15 downto 0)
:= "0000000000000000";
      begin
           if(clk='1') and (clk'event) and (load = '1') then
                 result := din;
                 dout <= result after delay;
            end if;
      end process;
end behavioral;

architecture schematic of register_16 is

      signal      n_4 : std_logic;
      signal      n_7 : std_logic;
      signal     n_10 : std_logic;
      signal     n_13 : std_logic;
      signal     n_16 : std_logic;
      signal     n_19 : std_logic;
      signal     n_22 : std_logic;
      signal     n_25 : std_logic;
      signal     n_28 : std_logic;
      signal     n_31 : std_logic;
      signal     n_34 : std_logic;
      signal     n_37 : std_logic;
      signal     n_40 : std_logic;
      signal     n_43 : std_logic;
      signal     n_46 : std_logic;
      signal     n_49 : std_logic;
      signal     dout_dummy : std_logic_vector(15 downto
0);

      -- mux2 is a 1-bit 2 to 1 selector
      component mux2
      port (      i0 : in     std_logic;
                  i1 : in     std_logic;
                   s : in     std_logic;
                   o : out   std_logic );
      end component;

      -- dff is a D flip flop
      component dff
     port (     clk : in    std_logic;
                  d : in    std_logic;
                  q : out   std_logic );
      end component;

begin

      dout <= dout_dummy;

      i_1 : mux2
               port map ( i0=>dout_dummy(0), i1=>din(0),
s=>load, o=>n_49 );
      i_2 : mux2
               port map ( i0=>dout_dummy(1), i1=>din(1),
s=>load, o=>n_46 );
      i_3 : mux2
               port map ( i0=>dout_dummy(2), i1=>din(2),
s=>load, o=>n_43 );
      i_4 : mux2
               port map ( i0=>dout_dummy(3), i1=>din(3),
s=>load, o=>n_40 );
      i_5 : mux2
```

```
        port map ( i0=>dout_dummy(4), i1=>din(4),
s=>load, o=>n_37 );
    i_6 : mux2
        port map ( i0=>dout_dummy(5), i1=>din(5),
s=>load, o=>n_34 );
    i_7 : mux2
        port map ( i0=>dout_dummy(6), i1=>din(6),
s=>load, o=>n_31 );
    i_8 : mux2
        port map ( i0=>dout_dummy(7), i1=>din(7),
s=>load, o=>n_28 );
    i_9 : mux2
        port map ( i0=>dout_dummy(8), i1=>din(8),
s=>load, o=>n_25 );
    i_10 : mux2
        port map ( i0=>dout_dummy(9), i1=>din(9),
s=>load, o=>n_22 );
    i_11 : mux2
        port map ( i0=>dout_dummy(10), i1=>din(10),
s=>load, o=>n_19 );
    i_12 : mux2
        port map ( i0=>dout_dummy(11), i1=>din(11),
s=>load, o=>n_16 );
    i_13 : mux2
        port map ( i0=>dout_dummy(12), i1=>din(12),
s=>load, o=>n_13 );
    i_14 : mux2
        port map ( i0=>dout_dummy(13), i1=>din(13),
s=>load, o=>n_10 );
    i_15 : mux2
        port map ( i0=>dout_dummy(14), i1=>din(14),
s=>load, o=>n_7 );
    i_16 : mux2
        port map ( i0=>dout_dummy(15), i1=>din(15),
s=>load, o=>n_4 );
    i_17 : dff
                port map ( clk=>clk, d=>n_46,
q=>dout_dummy(1) );
    i_18 : dff
                port map ( clk=>clk, d=>n_49,
q=>dout_dummy(0) );
    i_19 : dff
                port map ( clk=>clk, d=>n_43,
q=>dout_dummy(2) );
    i_20 : dff
                port map ( clk=>clk, d=>n_40,
q=>dout_dummy(3) );
    i_21 : dff
                port map ( clk=>clk, d=>n_34,
q=>dout_dummy(5) );
    i_22 : dff
                port map ( clk=>clk, d=>n_37,
q=>dout_dummy(4) );
    i_23 : dff
                port map ( clk=>clk, d=>n_31,
q=>dout_dummy(6) );
    i_24 : dff
                port map ( clk=>clk, d=>n_28,
q=>dout_dummy(7) );
    i_25 : dff
                port map ( clk=>clk, d=>n_22,
q=>dout_dummy(9) );
    i_26 : dff
                port map ( clk=>clk, d=>n_25,
q=>dout_dummy(8) );
    i_27 : dff
                port map ( clk=>clk, d=>n_19,
q=>dout_dummy(10) );
    i_28 : dff
                port map ( clk=>clk, d=>n_16,
q=>dout_dummy(11) );
    i_29 : dff
                port map ( clk=>clk, d=>n_10,
q=>dout_dummy(13) );
    i_30 : dff
                port map ( clk=>clk, d=>n_13,
q=>dout_dummy(12) );
    i_31 : dff
                port map ( clk=>clk, d=>n_7,
q=>dout_dummy(14) );
    i_32 : dff
                port map ( clk=>clk, d=>n_4,
q=>dout_dummy(15) );

end schematic;
```

9.9 D Flip Flop Entity.

```
////////////////// D flip flop entity ////////////////////////////////

-- D flip-flop - behavioral description

library ieee;
   use ieee.std_logic_1164.all;

entity dff is
       generic (  delay : time := 4 ns;
                  setup : time := 2 ns;
                   hold  : time := 1 ns );
       port (    clk : in     std_logic;
                  d : in     std_logic;
                  q : out   std_logic );
begin
       process(clk)
       begin
             if (clk = ‘1’) and (clk’event) then
                   assert ( d’stable(setup) )
                   report “setup time violation !”
                   severity warning;
             end if;
       end process;

       process(d) begin
             if (clk = ‘1’) then
                   assert ( clk’stable(hold) )
                   report “hold time violation !”
                   severity warning;
             end if;
       end process;

end dff;

architecture behavioral of dff is
begin
       process(clk)
             variable state : std_logic := ‘0’;
       begin
             if (clk = ‘1’) and (clk’event) then
                   state := d;
                   q <= state after delay;
             end if;
       end process;
end behavioral;
```

9.10 2 Input And Gate Entity.

```
/////////////////// 2 input and gate entity ///////////////////////

-- 2 input and gate - behavioral description

library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_misc.all;
  use ieee.std_logic_arith.all;

entity and2 is
      generic ( delay : time := 2.4 ns );
      port (      i1 : in    std_logic;
                  i2 : in    std_logic;
                   o : out   std_logic );
end and2;

architecture behavioral of and2 is
begin
      o <= i1 and i2 after delay;
end behavioral;
```