Language and Compiler for FPGA

F.S. HIEW, K.H. KOAY
Faculty of Engineering and Technology
Multimedia University
Jalan Ayer Keroh Lama, 75450, Melaka
MALAYSIA

Abstract: - This paper presents a high-level, algorithmic, single-assignment programming language and its optimizing compiler for reconfigurable systems. The compiler is capable of accepting our proposed instruction sets and generating a set of synthesizable VHDL codes. Simulated annealing algorithm at the heart of this compiler determines the design speed and resource needed on Field Programmable Gate Array (FPGA). Language features are introduced and the structure of the compiler is discussed. In the paper, we particularly study the effects of simulated annealing schemes on our compiler.

Key-Words: - High-level, single assignment, VHDL, FPGA, compiler and simulated annealing.

1 Introduction

Since the introduction of Field Programmable Gate Array (FPGA), many companies rapidly come out with their own architectures to suit different applications. In order for designers to achieve the shortest system design time, as well as to attain the highest possible performance, FPGA companies always provide designers with their most comprehensive design tools for creating designs, such as Xilinx ISE and Altera Quartus II.

Consequently, nowadays FPGA designers can concentrate more on circuit design, instead of worrying about how internal logic assignment and place-and-route being done in FPGA. However, designing circuit using hardware description languages like VHDL and Verilog is often over-looked. Although these languages are suitable for chip design, they are time consuming tasks and heavily relied on designers' expert knowledge [1].

In this paper, a high-level, algorithmic, and single-assignment programming language and its compiler are proposed to reduce designers' loads in designing digital systems on FPGA. In our proposed method, design entry is a simple, readable, and C-like programming source code. The output is a set of synthesizable VHDL codes. In our proposed system, speed and area of the circuit are optimized in dataflow graph by using simulated annealing (SA) algorithm. In this paper, we particularly study the effects of simulated annealing schemes on our compiler. Experimental results show that simulated annealing is capable of finding the best possible implementation in an efficient manner and generating VHDL code for it. Fig.1 illustrates the overview of the proposed system.

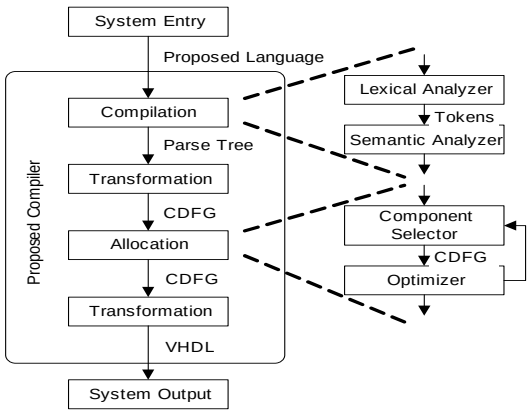


Fig.1. System overview

The main advantage of our proposed methodology is that they provide high-speed circuit design and verification capabilities to the circuit designers. Hence, it increases the designers' productivity and shortens the time-to-market. This methodology is also promoting component reusability while preserving its flexibility by outputting synthesizable VHDL codes. In other words, it can be easily extended to other architectures like ASICs and other devices that accept VHDL as the design input.

The rest of the paper is organized as follows: Section 2 reviews the related research. Section 3 provides a brief overview of our proposed language. Section 4 provides the details of flow-graph transformations and Section 5 discusses the simulated annealing used for component selection in our compiler. The transformation from our proposed source codes to VHDL codes is depicted in section 6 by using an example. Section 7 demonstrates the

experimental results. Concluding remarks are given in Section 8.

2 Related work

Simulated annealing (SA) is a general-purpose algorithm and applicable in combinational and function optimizations. Optimization by simulated annealing was introduced by Kirkpatrick, Gelatt and Vecchi [2] and widely used for path reduction purposes. The work in [3] applied SA to global wiring path routing for both idealized and actual designs of realistic size and complexity. Vecchi *et al*. reported that SA could achieve superior performance than the other sequential or greedy strategies those commonly employed in automatic wiring programs. Besides, SA was also utilized in rail network routing and the results demonstrated that SA was much preferable over genetic algorithm (GA) approach in which SA produces much better plans and easier parameter setting [4]. In our proposed compiler, SA is used for efficient design space exploration. Components in circuit must be selected correctly in such a way that the component meets the imposed throughput requirements.

3 The Proposed High-level Language

The proposed language is high-level and algorithmic, which simplifies circuit designing leading to shorter design time. All instructions are single-assignments and no pointer is involved for better compiler analysis and dataflow graph transformation [5]. Data types are the same as used in VHDL and the proposed language's variable name, type, and bit-width are user-specified. Operation assignment is similar to C programming as shown in Fig.2. Besides, timing and parallelism are excluded during system level design for hiding the details and intricacies of low-level hardware design.

```
#define : a,b,c,d,e,f,g,h,i,j,k : std_logic_vector := 8;
#define : l,m,n,o,r,s,t,u,v : std_logic_vector := 8;
#define : p,q : std_logic_vector := 12;
#define : w,x,y : std_logic_vector := 24;
e  = a + b;  f = c + d;   g = e + f;   i = g - h;
j  = k + l;   n = j + n;   o = i + m;
w = p*q;   y = w + x;
```

Bit Width

Type

Variable Name

Operation

Fig.2. Proposed high-level instruction set

4 Flow-graph Transformations

The flow-graph is obtained by parsing the instruction codes. This is achieved by first translating the source codes into a stream of tokens via lexical analyzer. This stream of tokens is further subjected to semantic analyzer which imposes a hierarchical structure on them to verify the correctness of the syntax. If all syntaxes are correct, a parse tree is formed. The parse tree is then further compressed to obtain a syntax tree. Finally, the syntax tree is transformed into a control and data flow graph (CDFG) that depicts the total flow of the control and data in the original description. Data dependencies that are inherent in the flow graph can be revealed through a full scan of the graph. Fig.3 gives a vivid description of the above methodology through an example. CDFG generated during control flow transformation just reveals its node operation and dependencies between nodes. Timing and other information will be added later on in the subsequent processes.

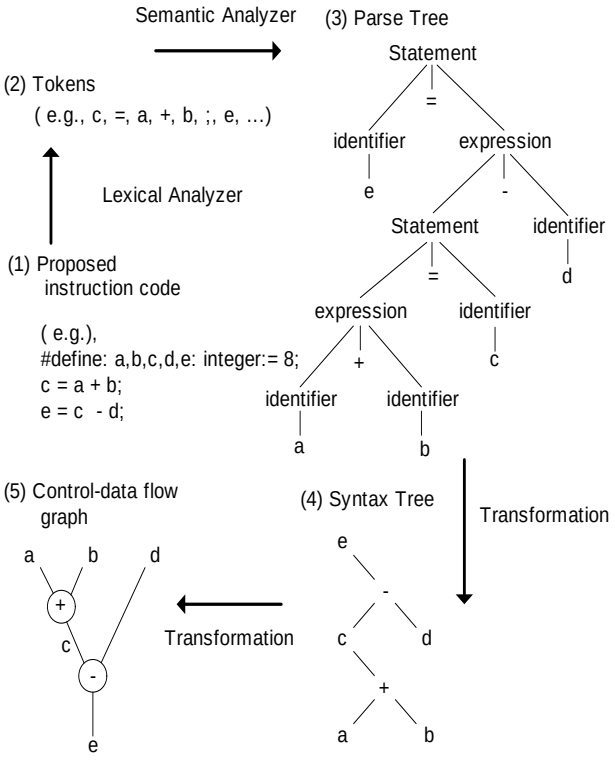


Fig.3. Transformation of CDFG from instruction set.

5 Component Selection

In this section, each node is assigned a VHDL component, which is chosen from a set of library, shown in Table 1. Note that timing delay (ns) and area (slice) in the table are generated by synthesizing the VHDL codes, from Zimmermann's arithmetic library

[6], in Xilinx Integrated Software Environment (ISE) series 5.2i.

Table 1. Components Library

Component	Preferences	Properties	4 bit	8 bit	……	64 bit
AddC	Slow	Area	6.00	12.00		
AddC	Slow	Delay	19.66	23.49		
AddC	Med	Area	6.00	12.00		
AddC	Med	Delay	19.66	23.49		
AddC	Fast	Area	6.00	12.00		
AddC	Fast	Delay	19.66	23.49		
AddCFast	Slow	Area	8.00	17.00		
AddCFast	Slow	Delay	18.99	23.50		
.						

When these VHDL components are assigned, the timing and resource information for a circuit can be estimated more easily. The critical-path circuit delay is

$$circuit\ delay = \sum_{i=1}^{c} delay_i \qquad (1)$$

In other words, the circuit delay is the summation of all delay times in the critical path. The total area is

$$circuit\ area = \sum_{j=1}^{n} area_j \qquad (2)$$

Circuit area is the total area in CDFG. From the example in Fig.4, we can see that the critical-path circuit delay is 93.14ns (24.06ns + 23.49ns + 22.77ns + 22.82ns) which is determined by the total delay in the critical path and the total area is the summation of all nodes' area in CDFG.

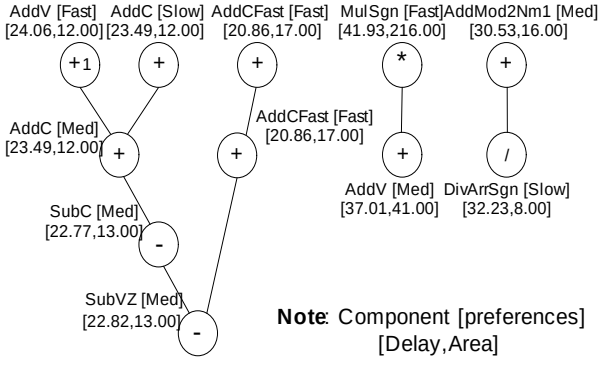


Fig.4. Nodes with VHDL component assigned

The VHDL component selection is based on four optimization mode selections. They are:

- ***Speed*** **(SD).** The speed of the circuit is set to maximum regardless of how large the area will be.
- ***Area*** **(AR).** The area of the circuit is set to minimum regardless of how slow the speed will be.
- ***Speed with acceptable area*** **(SAA).** The speed of the circuit is set to as high as possible with acceptable increasing in area size.
- ***Area with acceptable speed*** **(AAS).** The area of the circuit is set to as small as possible with acceptable decreasing in speed.

If optimization mode is set for ***speed*** (**SD**), the VHDL component with smallest delay (ns) will be chosen for all nodes in a particular circuit. If optimization mode is set for ***area*** (**AR)**, the VHDL component with smallest area (slices) is taken no matter how slow the component will be. On the other hand, if optimization mode is for ***speed with acceptable area*** (**SAA**) or ***area with acceptable speed*** (**AAS)**, optimization will be done by using simulated annealing (SA) algorithm.

SA performs speed and area optimizations in CDFG by randomly assigning various architecture of the defined VHDL component to the CDFG's nodes. The number of nodes to be assigned with new VHDL component at each iteration is based on the concept of *temperature reduction function* in SA. The number of nodes with new VHDL component assignment is high at high temperature and then it is decreased according to the temperature cooling schedule. When the temperature reaches freezing point, the process will stop. For example, the numbers of nodes changed in 12 iterations on CDFG in Fig.4 are 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, and 0. This concept is called as ***Component Reduction Function*** in our work. The **SAA** algorithm, in Fig.5, describes the modified SA algorithm, in which component reduction function is included to suit our needs. Fig.6 shows the relationship between the Temperature Reduction Function and Component Reduction Function.

In **SAA** mode, the delay of the critical path is evaluated every iteration. SA will just accept new component assignment if the iteration causes decrease in critical path delay. An increase in delay is accepted if the Metropolis criteria are fulfilled. In **AAS** mode, new assignment will be accepted if there is a decrease in total circuit area. On the other hand, an increase in area will be accepted if the Metropolis criteria are fulfilled.

6 VHDL Transformations

Assuming that the node with operator "$+_1$" in Fig.4 is generated from operation $e = a + b$ and is assigned with *AddC* (8-bits adder), this node is then transformed into VHDL module with input ports, output ports, clock, chip able, and reset signal. Note that, input ports in the module are the node's input

arcs (a and *b*) in CDFG. Whilst, the output port in the module is the node's output arc (*e*) in CDFG. Fig. 7 illustrates a part of VHDL (for operation *e* = *a* + *b*) source code.

In order to perform the addition operation, the AddC's VHDL component from Zimmermann's arithmetic library will be instantiated into the module. This process is repeated until every node is assigned with a module. Lastly, a top-level module with the same feature is created and all the modules formed previously are instantiated into this top module to form a complete VHDL module. Fig.8 depicts the abstract view of the top module formed. Each operator is instantiated with two registers at their input ports to store their input signals.

```
dTemp   = Number of temperature steps
dTrial   = Number of trials at each temperature

ComponentSelection ( ) {

   Initial temperature and component
   for i = 1…. dTemp do
      if SimulatedAnnealing ( ) then
         Temperature := TemperatureReduction ( );
         Component   := ComponentReduction ( );
      else
         break
      end if

         RouteLength := GetRouteLength ( );
         AreaSize       := GetTotalArea ( );

      if RouteLength < BestRouteLength or
         AreaSize       < BestAreaSize then
         store best route
      end if
       end for
}
SimulatedAnnealing ( ) {

   for j = 1….dTrial do
      randomly change Component  number of nodes
      trialCost := GetRouteLength ( );
      delta := current_cost - trial_cost

         if delta > 0 then
            make the change permanent
              bImprove := true
          else
            p  := random number[0…1]
              m := exp(delta / Temperature)
            if p < m then
           make the change permanent
                 bImprove := true
            end if
          end if
   end for
   if bImprove := true then
      return true
   end if
}
```

Fig.5. Pseudo code of modified SA

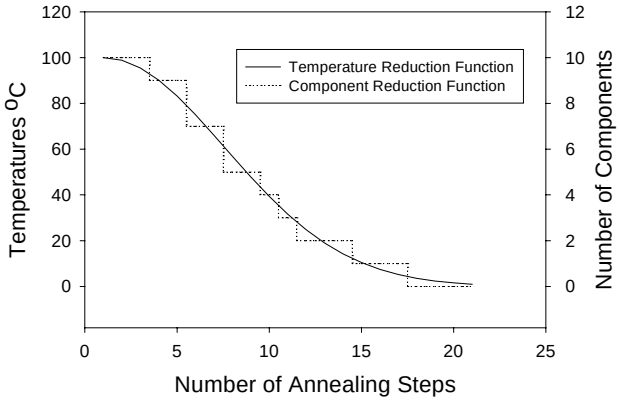


Fig.6. Temperature vs. Component Reduction Function

```
entity Module01 is
    port (          a_in   : in  std_logic_vector ( 7 downto 0 );
                    b_in   : in  std_logic_vector ( 7 downto 0 );
                    e_out  : out std_logic_vector ( 7 downto 0 );
                    ce, reset, clock : in  std_logic;          );
end Module01;

architecture Behavioral of Module01 is
        -- vhdl component declaration
        component AddC
        --signal declaration
        signal a,b,e: std_logic_vector ( 7 downto 0 );
begin
        -- vhdl component instantiation
        Comp01 : AddC
        --data in
        process( clock, reset )
        begin
                if reset = '1' then
                        a <= (a'range => '0');
                        b <= (b'range => '0');
                elsif clock'event and clock = '1' then
                        if ce = '1' then
                                a <= a_in;
                                b <= b_in;
                        end if;
                end if;
        end process;
        --data out
        e_out <= e;
end Behavioral;
```

Fig.7. VHDL module for instruction *e* = *a* + *b*.

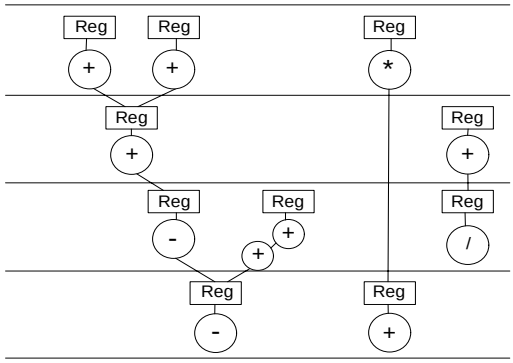


Fig.8. Abstract representation of the top module.

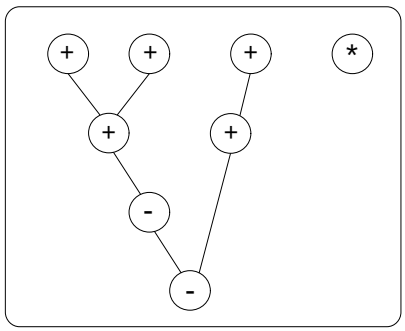
7 Experimental Results

The language and compiler have been implemented and tested based on the design illustrated in Fig.9, which contains eight operation nodes and the critical path formed by four operation nodes. In order for the SA algorithm to perform efficiently in our compiler, a suitable ***component reduction function*** must be chosen. A series of experiments has been done to compare the results from various ***component reduction functions*** in our compiler. These functions are shown below and the result is shown in Table 2.

```
#define:a,b,c,d,e,f,g,h,i,j,k,l,m,n,o:integer:=8;
#define:p,q:integer:=12;
#define:w:integer:=24;

e=a+b;
f=c+d;
g=e+f;
i=g-h;
j=k+l;
m=j+n;
o=i-m;

w=p*q;
```



(a) (b)

Fig.9. Example (a) Instructions set (b) CDFG

$$T_1 = T_o - i\left((T_o - T_N)/N\right) \quad (3)$$

$$T_2 = T_o\left(T_N/T_o\right)^{i/N} \quad (4)$$

$$T_3 = T_o - i^A, A = \ln\left(T_o - T_N\right)/\ln\left(N\right) \quad (5)$$

$$T_4 = (1/2)(T_o - T_N)\left(1 - \tanh\left((10i)/N - 5\right)\right) + T_N \quad (6)$$

$$T_5 = \left(T_o - T_N\right)/\cosh\left((10i)/N\right) + T_N \quad (7)$$

$$T_6 = A/(i+1) + T_o - A, A = \left((T_o - T_N)(N+1)\right)/N \quad (8)$$

$$T_7 = \left(T_o - T_N\right)/\left(1 + e^{3(i-N/2)}\right) + T_N \quad (9)$$

$$T_8 = (1/2)\left(T_o - T_N\right)\left(1 + \cos\left((i\pi)/N\right)\right) + T_N \quad (10)$$

$$T_9 = T_o e^{-Ai}, A = (1/N)\ln\left(T_o/T_N\right) \quad (11)$$

$$T_{10} = T_o e^{-Ai^2}, A = \left(1/N^2\right)\ln\left(T_o/T_N\right) \quad (12)$$

From Table 2, we can observe that T_5 is superior as it obtains the highest probability of achieving minimum critical path delay. Besides, the delay time obtained in various trials is always less than 90.3 ns. Fig.10 shows the delays obtained from our **SAA** experiments. These critical path delays are decreasing along the component reduction (temperature) steps. Some increases in delay time are accepted to avoid being stuck at local minimum.

Table 2. Results (dalay) of **SAA** for 15 trials with various component reduction functions

Trial	Delay time (ns)									
	Eqn T_1	Eqn T_2	Eqn T_3	Eqn T_4	Eqn T_5	Eqn T_6	Eqn T_7	Eqn T_8	Eqn T_9	Eqn T_{10}
1	92.2	88.2	93.2	88.9	90.0	88.1	88.1	91.5	88.8	90.4
2	90.8	90.1	90.7	88.0	88.4	89.1	90.3	91.6	88.1	**87.6**
3	**87.6**	**87.6**	90.4	92.3	88.7	**87.6**	88.8	88.9	89.4	88.6
4	91.0	90.6	90.7	88.8	90.9	88.9	88.9	91.6	91.5	91.2
5	94.1	88.7	90.8	88.2	**87.6**	92.8	91.1	91.8	89.1	89.4
6	88.8	**87.6**	91.5	92.2	88.2	90.1	90.9	89.5	90.9	89.7
7	91.9	93.3	92.1	88.2	88.2	**87.6**	88.1	**87.6**	93.4	88.2
8	**87.6**	88.2	89.5	89.8	90.3	93.0	91.6	90.8	89.1	89.9
9	91.4	89.5	89.2	88.2	88.4	88.7	92.1	88.8	92.7	93.1
10	91.8	**87.6**	90.7	92.5	88.2	94.0	**87.6**	91.5	89.6	90.7
11	**87.6**	**87.6**	91.1	88.4	**87.6**	90.1	90.2	92.4	90.2	92.9
12	88.4	89.4	**87.6**	89.3	**87.6**	89.5	**87**.6	**87.6**	88.7	89.4
13	92.6	88.4	91.9	89.8	90.1	90.5	91.2	88.9	88.4	89.5
14	90.7	88.3	91.4	90.9	88.2	91.4	89.4	90.4	87.6	90.3
15	90.8	90.9	**87.6**	89.5	**87.6**	89.3	89.1	91.3	90.2	89.4

In our subsequent experiment, the design is simulated based on T_5 as ***component reduction function***. The simulated delay and area of the abovementioned optimization modes are compared. From Table 3, it is shown that optimization with **SD** mode outperforms the others in term of speed which obtains minimum delay time (87.6ns). Whilst, optimization with **AR** mode is superior in term of area because area size (274 slices) obtained is smaller compared with the others. The simulation results for **SAA** and **AAS** modes are in between **SD** and **AR** optimization modes' results.

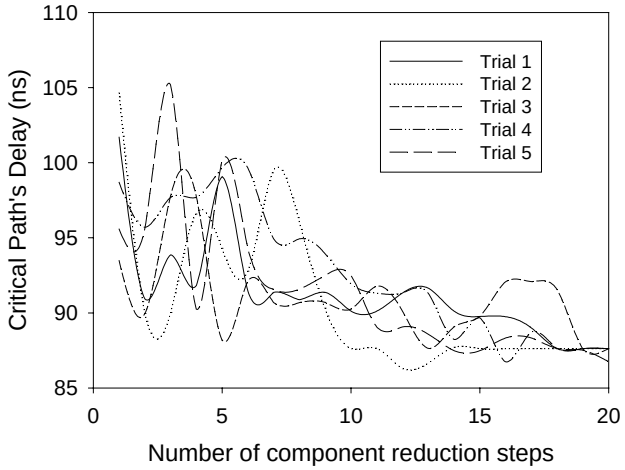


Fig.10. Critical path's delay obtained in a **SAA** trials

Table 3. Simulation results

Modes	Speed (**SD**)				Area (**AR**)			
Delay (ns)	**87.6**				99.4			
Area (slices)	318				**274**			
Modes	Speed with acceptable area (**SAA**)				Area with acceptable speed (**ASS**)			
Trials	1	2	3	4	1	2	3	4
Delay (ns)	90.3	88.0	91.8	90.4	93.8	94.9	92.6	98.9
Area (slices)	299	296	307	289	278	282	281	285

The results simulated by our compiler were compared with the results generated by Xilinx ISE 5.2i. The parameters were set as follow:

device family	: Spartan2
device	: xc2s200
package	: pq208
speed grade	: -5
optimization goal	: speed
optimization effort	: normal
place & route effort	: default (low)

Table 4 shows the results of the selected component in our four trials. From the table, we can see that the time delay and area size generated by using Xilinx ISE are greater than the simulated results. This is because registers are added in front of all modules for latching their input signals. Besides that, delay time and the number of resource used are depended on the place-and-route algorithm of the Xilinx ISE.

Table 4. Simulation results

Trial 1			Trial 2		
Component	Delay	Area	Component	Delay	Area
AddCFast[8,3]	20.86	17	AddCFast[8,2]	23.00	17
AddCFast[8,3]	20.86	17	AddV[8,1]	23.49	12
AddCFast[8,3]	20.86	17	AddC[8,2]	23.49	12
AddCFast[8,3]	20.86	17	AddC[8,2]	23.49	12
AddCFast[8,3]	20.86	17	AddV[8,3]	24.06	12
Sub[8,2]	21.57	11	Sub[8,2]	21.57	11
Sub[8,2]	21.57	11	SubC[8,1]	23.49	12
MulUns[12,3]	39.87	211	MulUns[12,2]	41.67	210
Our proposed system	***39.87**	**^318**	Our proposed system	***41.67**	**^298**
Xilinx ISE	**43.4**	**365**	Xilinx ISE	**43.1**	**354**

Trial 3			Trial 4		
Component	Delay	Area	Component	Delay	Area
AddCFast[8,3]	20.86	17	AddC[8,3]	23.49	12
AddCFast[8,3]	20.86	17	AddC[8,1]	23.49	12
AddC[8,1]	23.49	12	AddC[8,2]	23.49	12
AddMod2Nm1S0[8,2]	25.45	19	AddMod2Nm1[8,3]	30.65	15
AddV[8,1]	23.49	12	AddMod2Nm1S0[8,3]	31.20	16
SubV[8,3]	22.31	13	Sub[8,3]	21.57	11
SubVZ[8,3]	23.02	13	Sub[8,1]	21.71	9
MulUns[12,3]	39.87	211	MulUns[12,2]	41.67	210
Our proposed system	***39.87**	**^314**	Our proposed system	***41.67**	**^297**
Xilinx ISE	**42.8**	**367**	Xilinx ISE	**44.6**	**389**

Note: * = Largest and ^ = Total

8 Conclusion

We have presented a high-level, algorithmic, and single assignment language and its compiler. We have demonstrated, through an example, that the compiler is capable of generating synthesizable VHDL code for circuit design using our proposed programming language. Simulated annealing approach is used for speed and resource optimization. Future effort will be concentrated on refinement of the presented techniques.

References

[1] W. A. Najjar, W. Bohm, B. A. Draper, J. Hammes, R. Rinker, M. Chawathe, and C. Ross, High-level language abstraction for reconfigurable computing**,** *Computer*, vol.36, issue.8, 2003, pp.63 – 69.

[2] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, Optimization by simulated annealing, *Science*, vol.220, 1983, pp.671-680.

[3] M. P. Vecchi and S. Kirkpatrick, Global Wiring by Simulated Annealing, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol.2, issue.4, October 1983, pp.215-222.

[4] D. E. Brown, C. L. Huntley, B. P. Markowicz, and D. E. Sappington, Rail network routing and scheduling using simulated annealing, *IEEE International Conference on Systems, Man and Cybernetics*, vol.1, 1992, pp.589-592.

[5] J. Hammes, R. Rinker, W. Najjar, and B. Draper, A High-level, Algorithmic Programming Language and Compiler for Reconfigurable Systems, *The 2nd International Workshop on the Engineering of Reconfigurable Hardware/ Software Objects (ENREGLE), part of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Las Vegas, NV, June 2000.

[6] R. Zimmermann, VHDL Library of Arithmetic Units, *in Proc. First Int. Forum on Design Languages (FDL'98)*, Lausanne, Switzerland, Sept. 1998.