Military Technical College
Kobry El-Kobbah,
Cairo, Egypt

6th International Conference
on Electrical Engineering
ICEENG 2008

Implementation of Hardware Genetic Algorithm

By

Imbaby I. Mahmoud * May Salama** Asmaa Abdel Tawab*

Abstract:

This work presents a hardware implementation of a Genetic Algorithm. Hardware Genetic Operators are implemented in FPGA. Fitness evaluation, which is problem dependent, is left for implementation as S/W module or problem specific hardware design. This allowed a re-configurable general-purpose design, which is customized by application specific population generation and fitness evaluation solution. A 16 site Random Number Generator module is implemented in VHDL based on Hybrid Cellular Automata (CA). Selection, Crossover, and Mutation Operators are implemented as systolic architecture. For preserving locality & modularity of systolic arrays we separate selection array implementation from the crossover and mutation operators. The chromosomes are fed serially to allow variable length chromosomes. The Genetic Engine is targeted a Xilinx Vertex XC2V2000-5 device using Xilinx Foundation Environment. The simulation is carried out using ModelSim.

Keywords:

Genetic algorithms, FPGA and VLSI design

---

* Atomic Energy Auority, Cairo
** Shobra Faculty of Engineering, Benha Univ. Cairo

1. Introduction:

A genetic algorithm (GA) is a stochastic search and optimization technique based on the mechanics of natural selection. A population of candidate solutions (Chromosomes) is held and interacts over a number of iterations (Generations) to produce better solutions. In canonical GA, the chromosomes are encoded as binary strings. Driving the process is the fitness of the chromosomes, which relates the quality of a candidate in quantitative terms. The fitness function encapsulates the problem- specific knowledge. The fitness is used in a stochastic selection of pairs of chromosomes which are 'reproduced' to generate new solution strings. Reproduction involves Crossover, which generates new children by combining chromosomes in a process which swaps portions of each others genes. The other reproduction operator is called Mutation. Mutation randomly changes genes and is used to introduce new information into the search. Both crossover and mutation make heavy use of random numbers. In case of H/W GA, this random number needs to be generated by the device itself. There are aspects of GA approach attract H/W implementation. The operation of selection and reproduction are basically problem independent and involve basic string manipulation tasks. These can be achieved by logical circuits. The fitness evaluation task, which is problem dependent, however proves a major difficulty in H/W implementation. Another difficulty comes from that designs can only be used for the individual problem their fitness function represents. Therefore, in this work the genetic operators are implemented in H/W, while the fitness evaluation module is treated separately. It can be implemented as a S/W model. This allows a mixed hardware/software approach to address both generality and acceleration. In some cases the fitness evaluation can be simplified implemented in H/W.

The Genetic H/W Engine itself is composed of three modules as shown in Fig.1. FPGA is used for implementing these modules. Design and simulation results of these modules are presented. A random number generation module based on Cellular Automata CA is also designed and implemented to provide the other three modules with H/W generated random numbers. Simple fitness evaluation and population generation module is presented.

The RNG supplies pseudo-random bit strings to the selection module for scaling down the sum of fitness. The pseudo-random bit strings to the crossover and mutation modules to choose crossover and mutation points

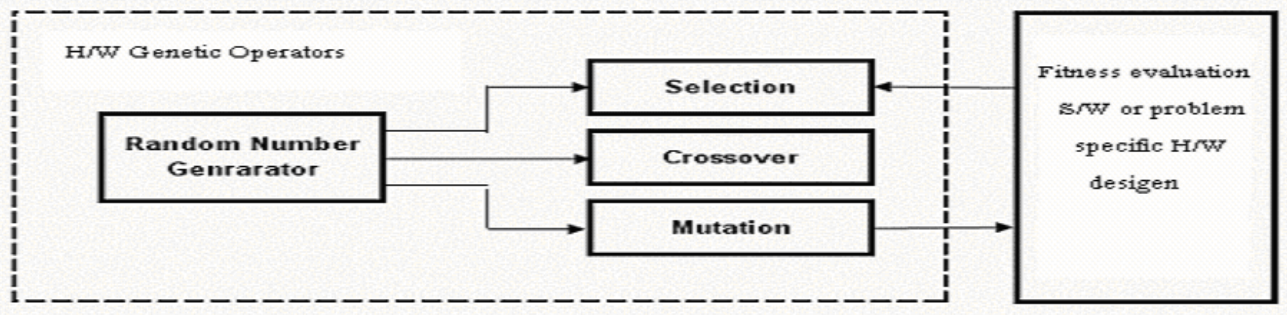

Figure (1): *Genetic Algorithm*

2. Random Number Generator :

The 1st module to be considered is a Random Number Generator (RNG). Hybrid Cellular Automata (CA) [1] is used due to its maximal length binary sequence production from each site. CA based generators compare favorably with the other types such as Linear Feedback Shift Registers (LFSR) and mixed congruential RNG in terms of quality of randomness and silicon area used. They use less silicon area in their implementation [1,2]. Four sites and 16 sites are implemented. Good results are obtained by forcing the neighbor to the last site to one. The boundary condition is not cyclic. 1st cell seeding, which can be all zeros, are applied only once. The output of the cell is passed out to provide the seed for next cell. Ref [1] applied random number tests of Kunth for CA RNG . It proved the good Randomness of this type of RNG.
Fig.2 shows the VHDL code of 4 site RNG using Xilinx Foundation 5.2i.

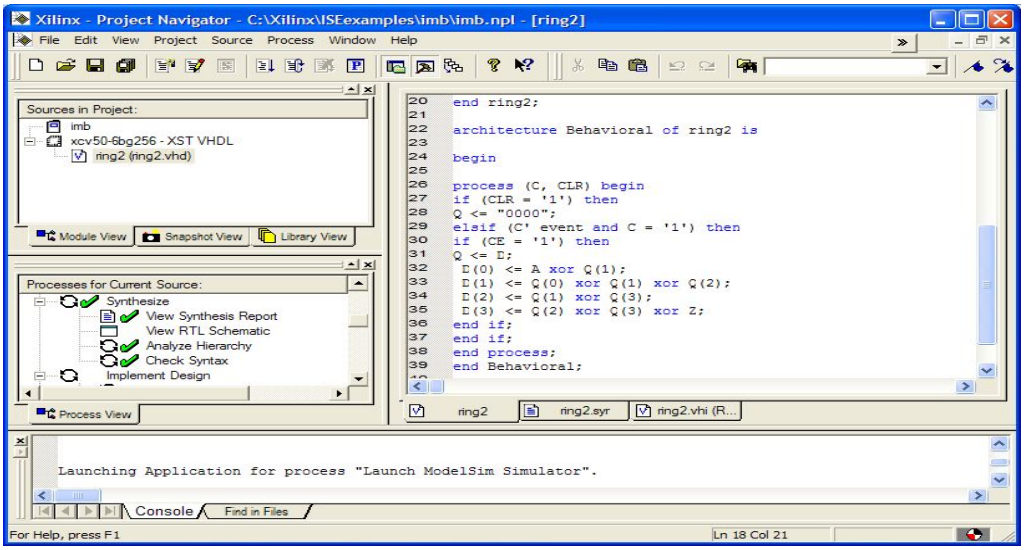

Figure (2): VHDL code for 4 sites Hybrid CA RNG

A 16 site RNG is implemented in VHDL. Fig.3 shows the simulation results of this RNG.

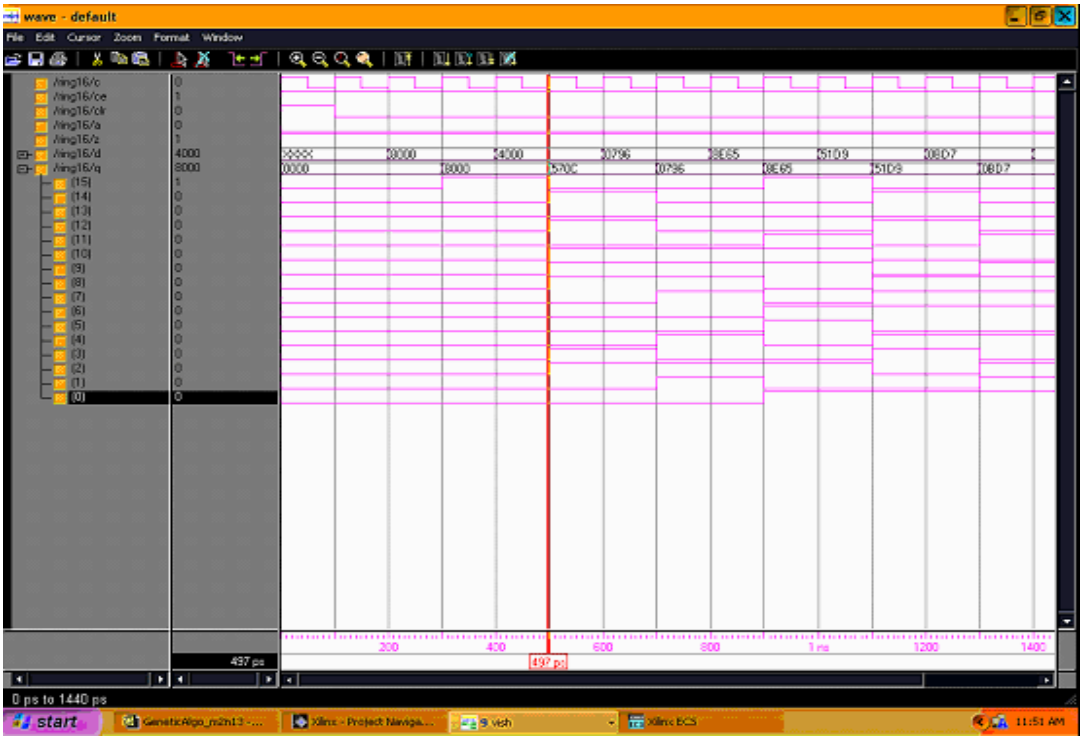

Figure (3): *Simulation of 16 sites Hybrid CA RNG*

Table (1): *Implementation results for 16 bit RNG based on CA A=0 Z=1*

Seed Q=	0	1	255	65535
1	32768	32771	33066	3690
2	16384	16388	17131	10923
3	8192	8202	9738	60072
4	61440	61467	64796	35500
5	47104	47152	44222	23206
6	1024	1112	10121	4797
7	35328	35476	65119	44685
8	23296	23266	43418	8405
9	4480	4631	12147	61761
10	43840	44838	58236	47691
11	10528	9213	38254	884
12	61168	62637	24641	34162
13	32824	45989	20643	18559
14	16452	7225	2484	15594
15	8362	48711	40722	53003
16	61867	2478	31663	62360

Our implementation of CA RNG set the bit 17 (the neighbor to the last site 16) to 1 so zeros can be used as initial value without producing zeros in the next cycles. Now the RNG will produce numbers in range from 0 to 2^{16}- 1 if it is given a number initially in that range. Also every number in that range will be visited every 2^{16}. If the initial value is zero or small value the next few numbers produced will not be ascending numbers. Table.1 demonstrates the implementation results of the implemented CA RNG with an initial value (0000000000000000) binary. It is noticed here that for a small seed, the generated numbers are freely randomized.

The synthesis summary of the 16 sites RNG is given in the following:

Device utilization summary:

Selected Device : v50bg256-6

Number of Slices:	18	out of 768	2%
Number of Slice Flip Flops:	32	out of 1536	2%
Number of 4 input LUTs:	17	out of 1536	1%
Number of bonded IOBs:	36	out of 184	19%
Number of GCLKs:	1	out of 4	25%

TIMING REPORT

Note: these timing numbers are only a synthesis estimate

For accurate timing information, refer to the trace report.

Generated after PLACE-and-ROUTE.

Clock Information:

Clock Signal	Clock buffer(FF name)	Load
c	BUFGP	32

Timing Summary:

Speed Grade: -6

Minimum period: 3.672ns (Maximum Frequency: 272.331MHz)

Minimum input arrival time before clock: 7.207ns

Maximum output required time after clock: 7.292ns

Maximum combinational path delay: No path found

This summary shows how small the amount of resources occupied by the RNG module. However to host the whole genetic operators, a larger chip will be used.

3. Selection:

The chromosomes emerge from the fitness evaluation and populations generation module enter the left of the select array [3,4]. A random number in the range of 0 to 1 (ball value) is input into the top of each column to be compared with the fitness values of the chromosomes. As the ball value descents, the fitness values are subtracted and the result is tested for zero crossing. Such an occurrence indicates selection. In the implemented design of selection module, the output selection is overwritten with the actual chromosome and passed to the next cell. Test is performed to ensure the selection is done only once for each column of selection array. Fig. 4 shows the simulation results of selection cell. Pseudo code of Selection cell can be written as follows:

```
If (ball_in < fit_in )
                {
                ball_out = Max;
                select_out = Chromo.gene_in;
                }
else            {
                ball_out = ball_in – fit_in;
                select_out = select_in;
                }
                Chromo.gene_out = Chromo.gene_in;
                fit_out = fit_in;
```

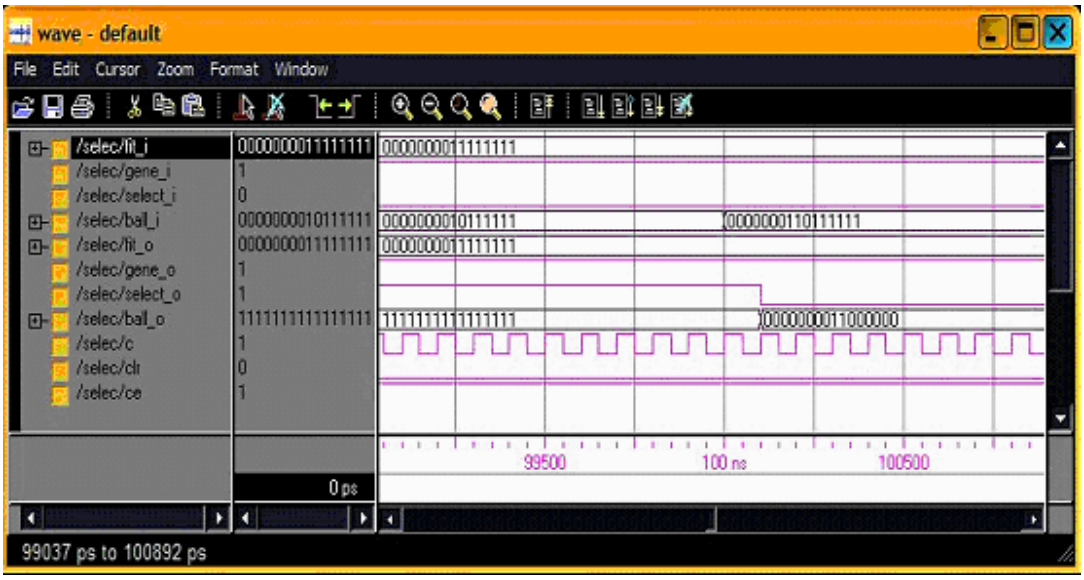

Figure (4): *Simulation of VHDL implementation of a Selection cell*

The implementation of 4 cell selection array is shown in fig. 5 while fig. 6 shows how the stagger is preserved between consecutive chromosomes.

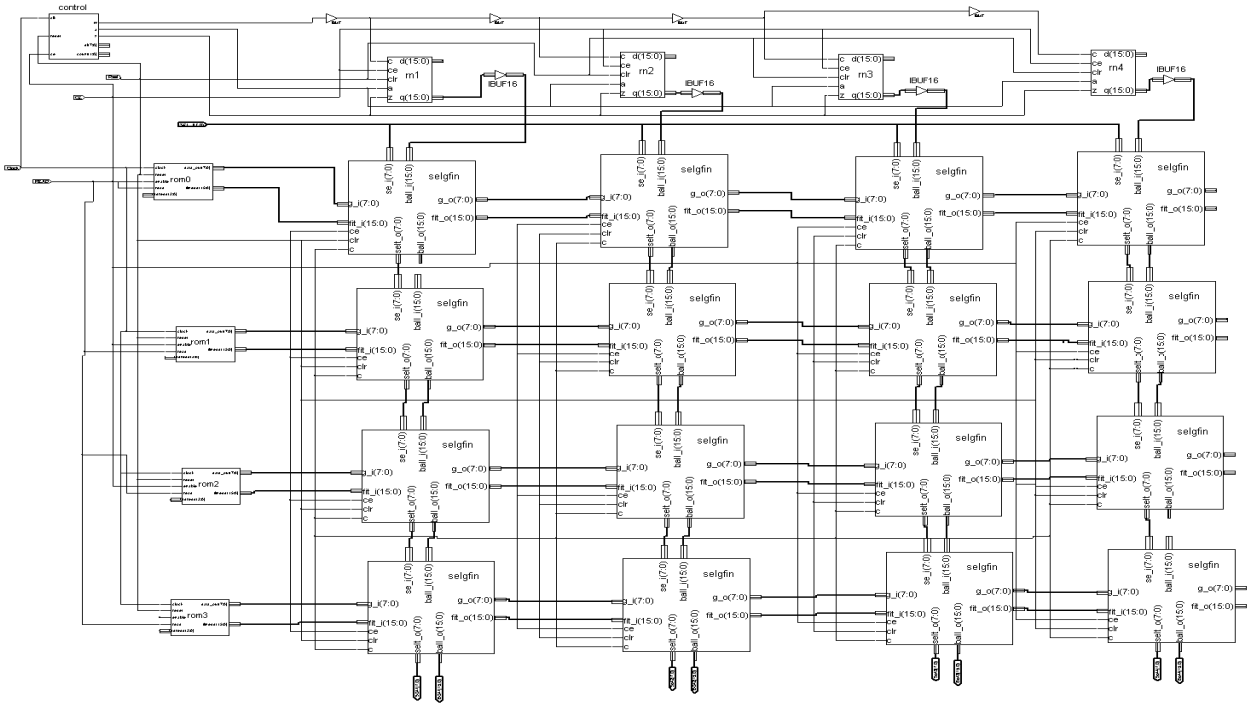

Figure (5): *Implemented Schematic of Selection Array*

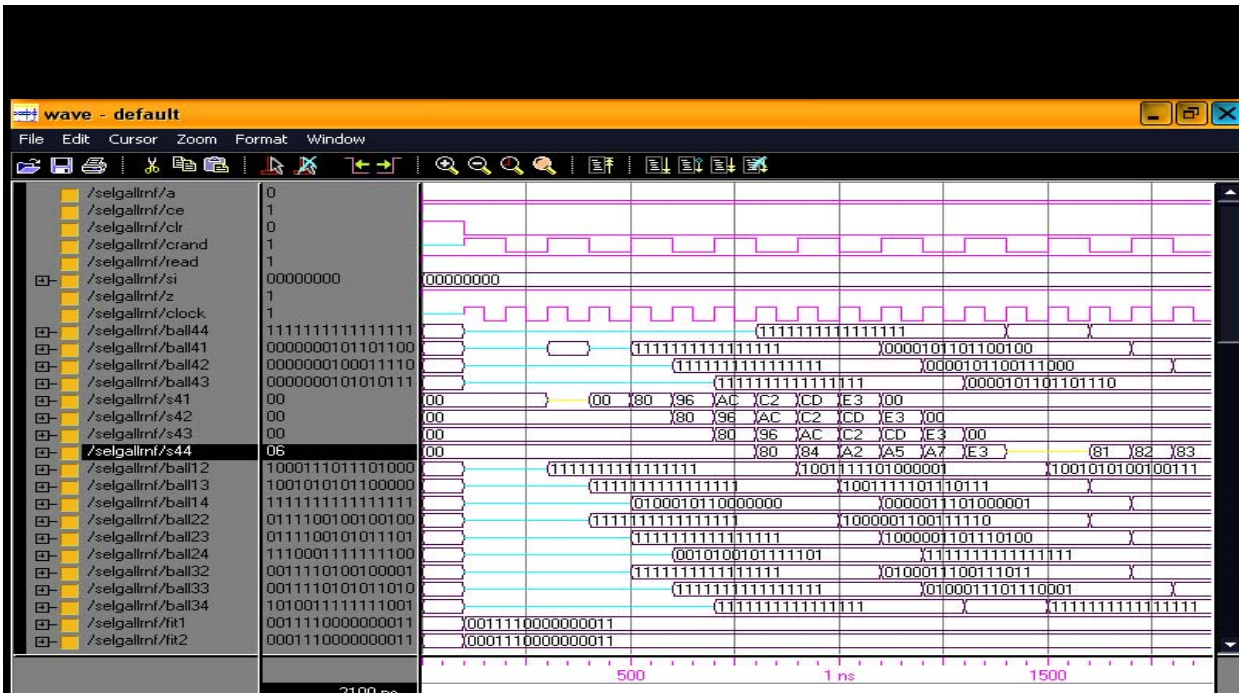

Figure (6): *Simulation of VHDL implementation of Selection Array*

3. Crossover:

Pseudo code of Uniform Crossover can be written as follows:

```
If rand = 1   {
              Chromo[I].gene[j] = Chromo[I+1].gene[j];
              Chromo[I+1].gene[j] = Chromo[I].gene[j];
              }
else          {
              Chromo[I].gene[j] = Chromo[I].gene[j];
              Chromo[I+1].gene[j] = Chromo[I+1].gene[j];
              }
```

Fig7 shows the VHDL implementation of 16 bit uniform crossover cell while the output is available in parallel.

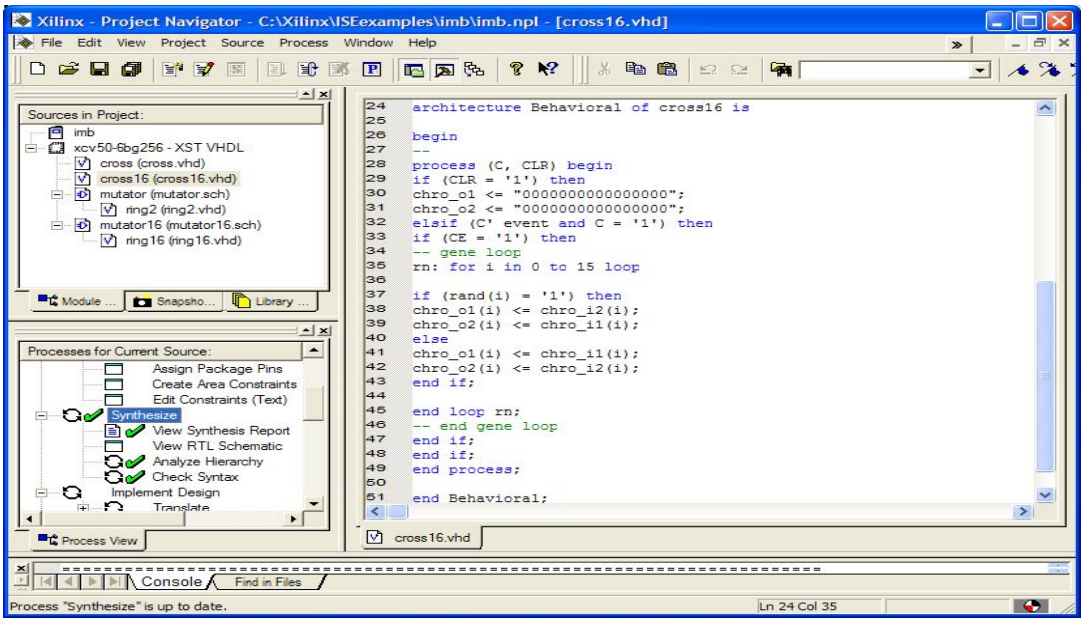

Figure (7): *VHDL implementation of 16 bit uniform crossover cell*

The simulation of *16 bit uniform crossover* cell is shown in fig 8.

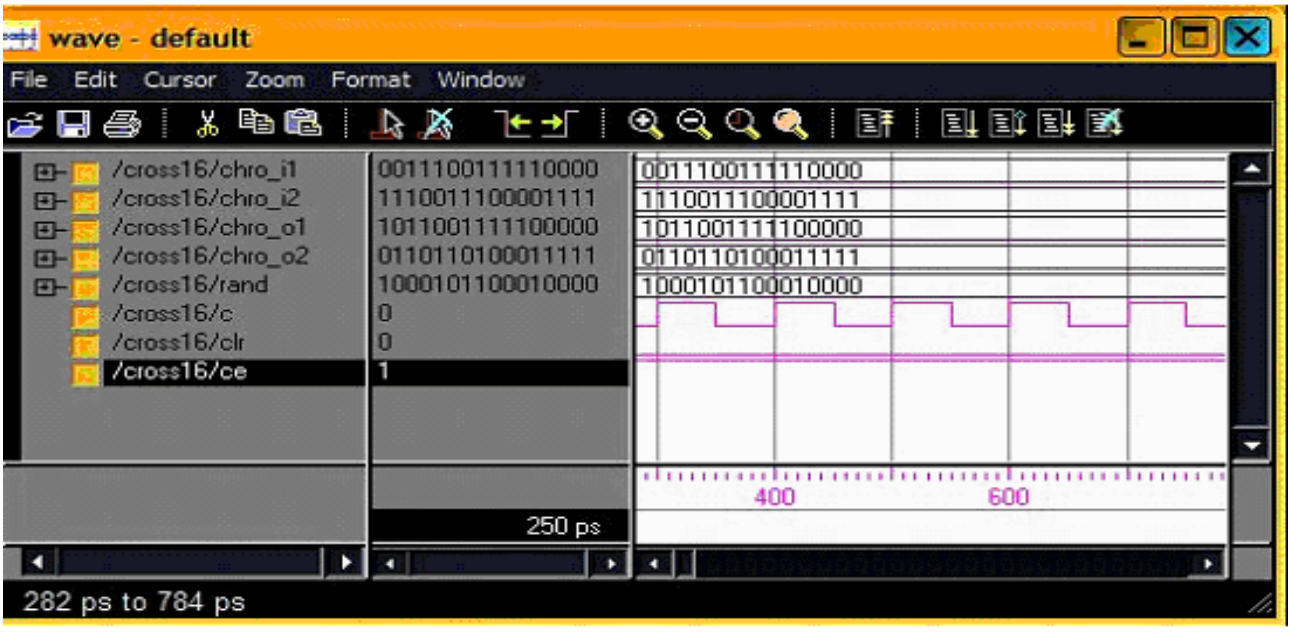

Figure (8): *Simulation of 16 bit uniform crossover cell.*

Since the process is to pump the chromosomes sequentially, a serialized version is implemented. Fig.9 shows the simulation of 16 bit uniform crossover cell while the

input and output are available in series.

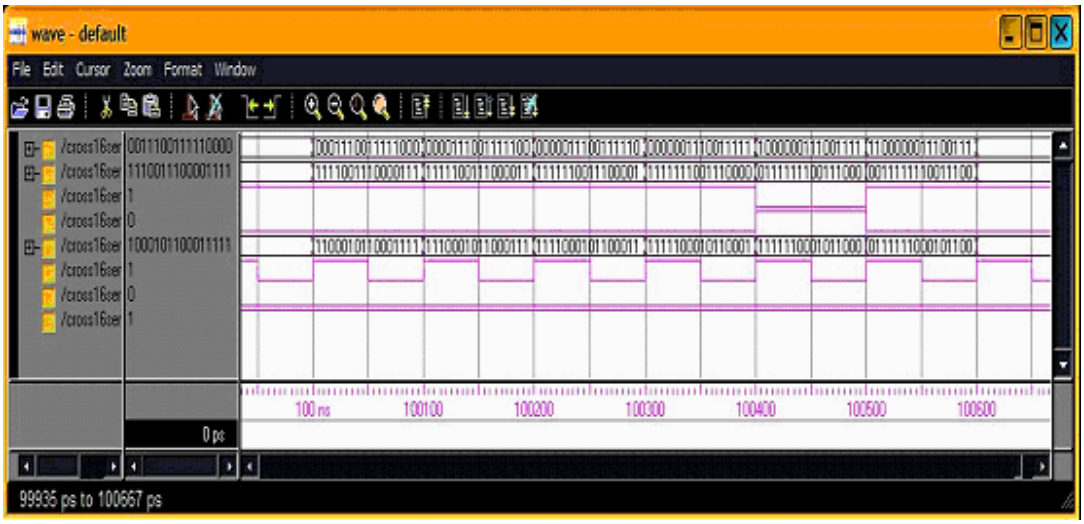

Figure (9): *Simulation of serial 16 bit uniform crossover cell*

Fig. 10 shows the schematic of crossover module which is composed from two cells.

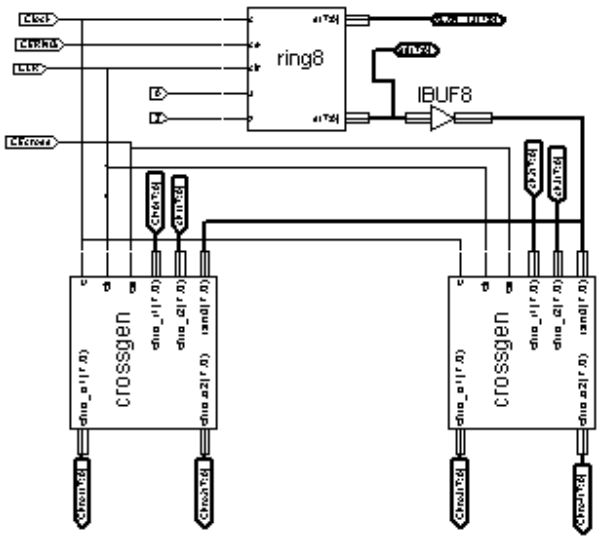

Figure (10): *Schematic of Crossover Module*

4. Mutation:

Once the random number has been generated, it is passed out to the mutation logic of Fig.11. The random number is compared with mutation probability P_{mut} using 16-bit magnitude comparator. The output of the comparator goes high if the random number is less than P_{mut} and this, together with the incoming gene, are fed into an XOR gate. If the comparator output is high the gene is inverted. Each mutation cell is written in VHDL. The simulation of mutation module is shown in Fig.12.

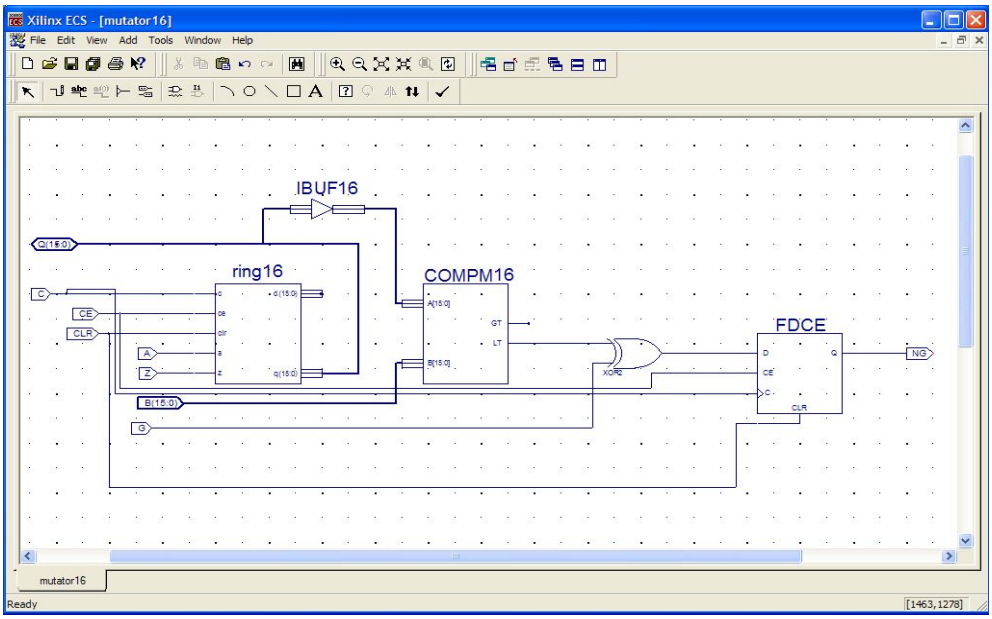

Figure (11): *Schematic of Mutation Module*

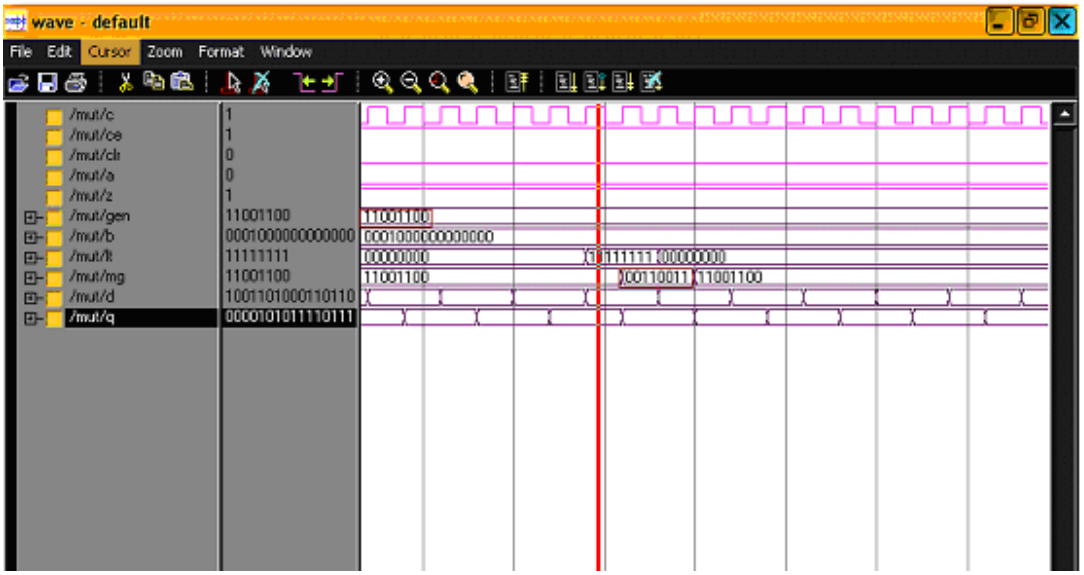

Figure (12): *Simulation of Mutation Module.*

5. Control and Storage module :

I. Memory Configuration

In our design, a memory module is used to store the chromosome which consists of 6 genes, every gene is 8bits in addition to its fitness value.
The memory consists of 4 blocks. Each block is accessed using a counter which is used as a pointer to read from and write to a specified location

II. Chromosome delivery control

This section generates master clock which is fed to each RNG module to keep the random number unchanged for the whole selection period (chromosome six gene). The control unit is responsible also for storing initial values

6. Implementation Results:

Since the Fitness evaluation module is a problem dependent, the total speed of the algorithm can not be determined. However, the speed measure for H/W Genetic Operators shows a significant improvement. Our implementation achieved 13.4 ns as maximum time consumed to traverse a single gene (processing). So, the implemented engine can process 74.6 million genes per second using Xilinx XC2V2000 with speed grade 5. Comparing with reference [5], 100% speed improvement is achieved.
The Device utilization summary for implementing both: Selection array and Crossover –

mutation arrays is reported.
- Device utilization summary:
Selected Device : 2v2000ff896-5

Number of Slices:	180	out of 10752	1%
Number of Slice Flip Flops:	218	out of 21504	1%
Number of 4 input LUTs:	257	out of 21504	1%
Number of bonded IOBs:	236	out of 624	37%
Number of GCLKs:	2	out of 16	12%

-Device utilization summary

Input File Name	: selection .prj		
Number of Slices:	644	out of 10752	5%
Number of Slice Flip Flops:	637	out of 21504	2%
Number of 4 input LUTs:	1028	out of 21504	4%
Number of bonded IOBs:	167	out of 624	26%
Number of TBUFs:	32	out of 5376	0%
Number of BRAMs:	3	out of 56	5%
Number of GCLKs:	1	out of 16	6%

The routs consume 37% and 26% for Selection array and crossover –mutation arrays respectively. This means that they can be hosted in a single partially configured chip.
Since placement and routing in FPGA from VHDL designs are not preserving locality, there is a major problem for implementation of systolic arrays in FPGA. For preserving locality and modularity of systolic arrays, we separate selection array implementation from the crossover and mutation operators. Partial configuration of recent FPGA allows this separation. Xillinx constraint editor allows for a careful placement and routing of each module.

7. Conclusions:

In this work a Hardware Implementation of a Genetic Algorithm is considered. Hardware Genetic Operators are implemented in FPGA. Fitness evaluation, which is problem dependent, is implemented as separate module. This allowed a re-configurable general-purpose design, which is customized by application specific population generation and fitness evaluation module. A 16 site Random Number Generator module based on hybrid Cellular Automata (CA) is implemented in VHDL. Selection, Crossover, and Mutation Operators are implemented and their simulation results are presented. The Genetic Engine is implemented in a Xilinx Vertex Xc2v2000-5 device using Xilinx Foundation Environment. H/W implementation of fitness evaluation module will be studied in future work.

References:

[1] P. Hortensius, R McLeod, and H. Card, "Parallel Random Number Generation for VLSI System Using Cellular Automata", IEEE Trans. On Computers, Vol.38, No.10, Oct. 1989.

[2] I. Bland and G. Megson, " Systolic Random Number Generation for Genetic Algorithms", Electronic Letters, Vol.32(12):1069, 1996.

[3] G. Megson and I. Bland, " Synthesis of a Systolic Array Genetic Algorithms" Proc. 12th Int. Parallel Processing Symp., 1998

[4] Hemmat Emam, "Genetic Algorithm Implementation using Hardware Tools", M. Sc. Thesis, Ain Shams University, 2001.

[5] I. M. Bland and G.M. Megson" The Synthesis Array Genetic Algorithm, An Example of Systolic Arrays as Reconfigurable Design Methodology" .IEEE Computer Society .In K.J. Pocek and J. M. Arnold, editor proc. of the IEEE Symposium on FPGAs for Custom Computing Machines ,pp.260-261, Los Alamitos, CA, USA, August 1998