# Chapter 6

# Hardware Implementation of GA

**Matti Tommiska** and **Jarkko Vuori**

Helsinki University of Technology
Otakaari 5A, FIN-02150 ESPOO, Finland
E-mail: Matti.Tommiska@hut.fi, Jarkko.Vuori@hut.fi

**Abstract.** A genetic algorithm has been designed with Altera Hardware Description Language (AHDL). The design has also been simulated and implemented with programmable logic devices of Altera's Flex 10K Field Programmable Gate Array (FPGA) family. The genetic algorithm is run on a PC card, which is connected to the central processing unit (CPU) through high-performance Peripheral Component Interconnect (PCI) bus.

Due to the easy reconfigurability of programmable logic devices, experimentation with variable population sizes and various fitness functions is greatly facilitated. This is accomplished by rewriting the AHDL code on the host computer and then reprogramming the chip on the fly through PCI bus.

The main advantage of a hardware-based genetic algorithm is its inherent speed advantage over software-based methods. This speed advantage makes hardware-based genetic algorithm a prime candidate for real time applications, for example optimization of routing in telecommunications networks.

**Keywords:** programmable hardware, FPGA, hardware GA

The complete title of this article: "Implementation of Genetic Algorithms with Programmable Logic Devices". The article is available via anonymous ftp at site `ftp.uwasa.fi` directory `cs/2NWGA` as file `Vuori.ps.Z`.

## 6.1 Introduction

Genetic algorithms (GAs) [420] have aroused an intense interest due to their flexibility in solving problems which traditional optimization methods and machine learning find difficult.

Due to their iterative problem solving method, the need for computational power is immense. Traditional microprosessors are not very efficient in running these genetic algorithms, especially good high-speed random number generation has been difficult to implement. By using traditional microprocessors it is also very difficult to fully exploit the inherent parallelism in genetic algorithms. Currently the complexity of programmable hardware has been evolving to the phase where large high-speed digital systems can be implemented on a single programmable logic chip. The potential benefit of using a genetic algorithm hardware is that it allows **both** the huge parallelism and extremely efficient atomic operations just suited to random number generation, crossover, mutation and fitness evaluation.

As the system support logic increases in complexity, it becomes harder to craft the circuitry without incorporating small-to-medium-sized blocks of memory (single-port or multi-port SRAMs, FIFO buffers, etc.). These integrated memory blocks both simplify system design and allow systems to operate at faster speeds, since off-chip references can be avoided or different-speed system buses can all be tied together [551].

Unlike the high memory densities achievable in gate arrays and custom designs, implementing blocks of memory on previous-generation field-programmable gate arrays (FPGAs) has not been very efficient. In addition, since memory cells have to be formed from the programmable logic, they have typically been slow, generally offering access times of 30 to 50 ns at best [125].

Genetic algorithms need large memory banks to store the population and this has made the hardware implementation of GAs very inefficient. But that is changing. Several recent FPGA family releases, especially ones that employ distributed memory such as the XC4000E family from Xilinx or the 10K family from ALTERA which has arrays of dedicated RAM blocks, now allow users to implement blocks of SRAM to their FPGA designs.

We selected ALTERA's 10K family [3] for our implementation because of the good availability of the largest chips from the 10K family. Design efficiency of the 10K family seems to be better than standard gate arrays or other RAM based FPGA families. The largest chip in the 10K family contains 62 000–158 000 gates, depending on the application. Logic in the 10K family is divided inoto logic array blocks (LABs) and into the embedded array blocks (EABs). LABs can only implement logic but EABs can implement either 2048 bits of SRAM memory or 100–250 gates towards specialized tasks like multipliers, ALUs, and DSP functions. The chips of the 10K family can be programmed with Altera's special AHDL description language which resembles the widely used VHDL hardware

description language.

### 6.1.1 Related work

There have been few reported studies on GA hardware implementations, one VHDL description has been announced in [81] but no performance estimations were made. In [247] hardware description of GAPA system containing multiple FPGA chips and multiple digital signal processors was given. In our work we have implemented a GA in hardware using AHDL hardware description language and simulated the performance of the system with real timing information from a readily available FPGA chip.

## 6.2 Hardware implementation

Our hardware consists of a Pentium microprocessor based base unit which has four high-speed PCI bus slots available. These slots may contain one to four special purpose GA hardware boards. One board contains the high-speed PCI bus interface, a white noise generator and an A/D converter and a couple of Altera's 10K family FPGA chips. FPGA chips can be configured by the host Pentium processor via the PCI bus. It is also possible to build the algorithm which resides on the FPGA chips in such a way that the main operating parameters are fully parametrized. These parameters can then updated via the PCI bus and no AHDL rewriting and compilation is needed.

The population resides on the FPGA chip in the EABs, which are a flexible RAM and are therefore ideally suited for the storage of individual chromosomes. The fitness function is evaluated on the same chip in LABs, which can be configured for various arithmetic operations.

### 6.2.1 Random number generation

Good random number generation is of great importance to the proper operation of the GA. Normally random numbers are made by using linear shift-register (LSHR) based random number generators. These kind of generators are easy to implement and produce fairly good pseudo-randomness. In order to get the periodicity of the pseudo-random numbers sufficiently long, three shift-registers with appropriate lengths are coupled together.

Seed to this LSHR based random number generator is generated with the help of a noise diode, an amplifier and a high-speed (10 megasamples per second) 12-bit analog-to-digital converter. This produces natural gaussian noise and true randomness, which is very important in genetic algorithms. A sample from the noise diode is used as the seed and is added periodically to the LSHR random number generator. Random numbers are needed at 50 ns intervals and these seeds

are received at 100 ns intervals which means that every other random sample is from the noise diode and every other is synthetically produced in the shift register.

### 6.2.2 Mutation

Mutation is implemented by toggling one randomly selected bit. The mutation rate is now 100 %, but this can be varied easily. The code in figure 6.1 implements the mutation part of our genetic algorithm.

```
TITLE "mutation";

SUBDESIGN muta002(
    rand[4..0]          : INPUT;
    string_in[31..0]    : INPUT;
    string_out[31..0]   : OUTPUT;
)

VARIABLE
    xor_partner[31..0] : NODE;

BEGIN
    TABLE
        rand[4..0] => xor_partner[31..0];

        H"00"    => B"00000000000000000000000000000001";
                 .
                 .
                 .
        H"1E"    => B"01000000000000000000000000000000";
        H"1F"    => B"10000000000000000000000000000000";
    END TABLE;

    string_out[] = string_in[] xor xor_partner[];
END;
```

**Figure 6.1:** AHDL code for the mutation operation.

### 6.2.3 Operation of the Pipeline

The genetic algorithm is run in a pipelined fashion, figure 6.2. The pipeline comprises of four stages, which are separated by register banks. The register banks are necessary for the synchronization of the pipeline and the preservation of the addresses of the chromosomes. This guarantees that the same RAM addresses which the chromosomes were read from are also written to after the pipeline has processed and evaluated the chromosomes and their offsprings.

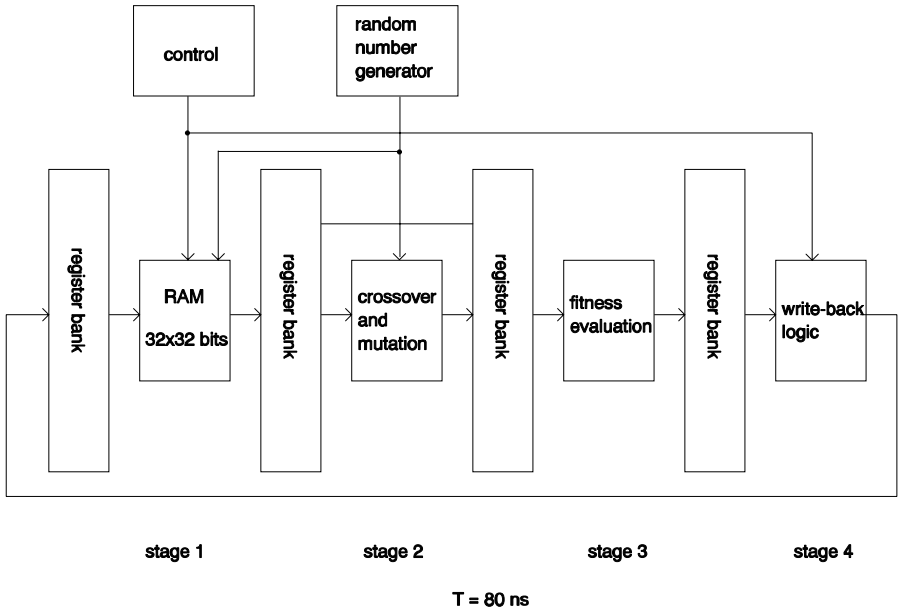


**Figure 6.2:** GA pipeline operates in four stages.

At the first stage of the pipeline, two chromosomes are selected at random from a 32x32 RAM (random access memory) block. The RAM is implemented as a synchronous memory with separate read and write ports. This facilitates the implementation of the control block, since no bidirectional data busses are required.

At the second stage of the pipeline, the selected chromosomes are subjected to crossover and mutation. The selected chromosomes are also passed over to the next stage in their original unchanged form. The crossover site between the two chromosomes is selected at random and the resulting offsprings are subjected to mutation, which is implemented as an inversion of a randomly selected bit in the 32-bit long chromosome. The random numbers used in the selection of the crossover site and mutation are uncorrelated to each other. In this design, the crossover probability was set to 100 % and the offspring chromosomes were subjected to mutation with a probability of 3.1 %. Both the crossover logic and mutation probability can be changed on the fly by rewriting the AHDL code.

At the third stage of the pipeline, the four chromosomes — the original two and their offsprings — are evaluated. The fitness function used in this design was a simple comparison between unsigned 32-bit binary numbers. The four chromosomes were compared with each other in a round-robin fashion. Since every chromosome must be compared with all other three chromosomes, a total of six 32-bit comparisons were required. The best two chromosomes were selected for write-back in the next stage of the pipeline. The flexibility of Altera's 10K internal architecture allows the updating of the fitness function without seriously affecting the operating speed of the pipeline. For example, if the fitness func-

tion included multiplications with a constant value, experimenting with different constant values would be easy by simply rewriting the AHDL code.

At the fourth and last stage of the pipeline, the best two chromosomes from the previous stage are written back to the same addresses from which either the chromosomes themselves or their parents were read from four clock cycles earlier. The control logic of the pipeline sets the appropriate control signals of the RAM blocks and of the address selection logic. The pipeline is active during four consecutive cycles and inactive during the next four clock cycles, giving it an efficiency of 50 per cent.

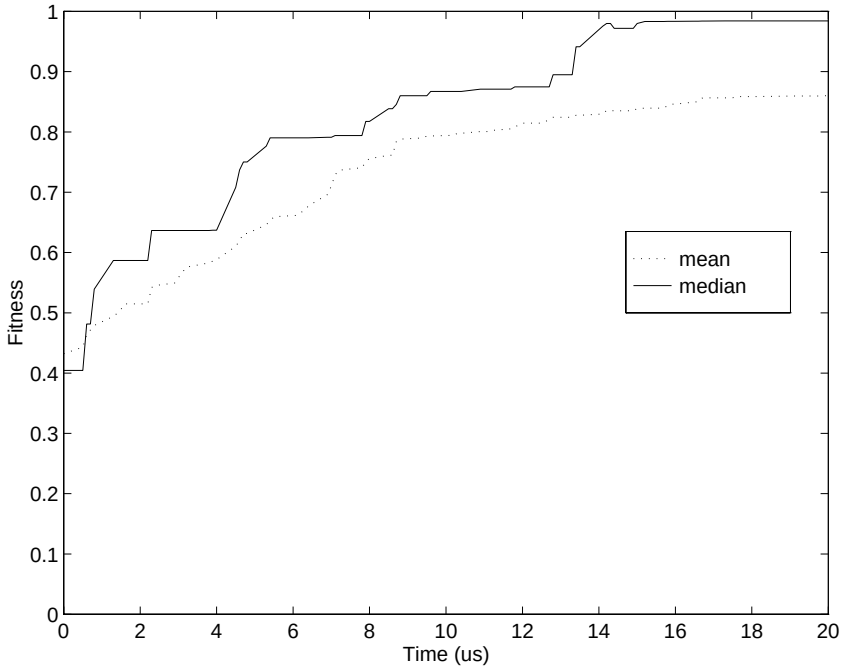


**Figure 6.3:** Fitness convergence of the hardware implemented genetic algorithm.

## 6.3 Simulations and performance

The design was simulated with Altera's Max+Plus II software simulator. The chromosomes residing in the RAM block were initialized with random values. The simulation results demonstrated both rapid convergence and robust overall performance. Algorithm was targeted to the Altera FLEX 10K50 chip, which contains 50000 usable gates. This implementation used 5% of the memory capasity and 51% of the logic cells.

Both the mean and median values of the population increased steadily during the first 20 $\mu$s of simulation. When the maximum achievable hexadecimal value FFFFFFFF was normalised to 1, the median value of the population was 0,98 and

the mean value of the population was 0,86 at the end of the 20 $\mu$s long simulation period, figure 6.3. The best hexadecimal value of an individual chromosome was FFF73267. The most remarkable feature of the simulated pipeline was its speed when compared to software-based genetic algorithms.

The largest achievable operation speed was 12.5 MHz, giving the design a cycle time of 80 ns. Since the pipeline operates at a 50 per cent efficiency, the selection of two chromosomes, their crossover operation and mutation, fitness evaluation and write-back requires 160 ns of processing time. Because the chromosomes are selected at random, successive generations overlap. The pipelined algorithm processes 32 chromosomes in 2.56 $\mu$s, which can be regarded as the time in which the number of chromosomes processed equals the population size.

The same algorithm was coded in C language and compiled and run on a Linux system with 120 MHz Pentium* processor. In this case the complete selection, crossover and mutation fitness evaluation cycle took 34 $\mu$s. This means that our hardware was $34/0.160 = 212$ times faster than the software solution. The same algorithm was also run on a HP C110 workstation (SPECint92 167) with HPUX operating system. In this case the complete cycle took 44 $\mu$s.

## 6.4 Conclusions and future

We have demonstrated the feasibility of using modern FPGA chips to speed-up the operation of genetic algorithms. In our case we got an improvement of roughly 200 compared to the software solution. This can be easily further improved by a factor of four by adding three additional parallel operating calculation units to the same FPGA chip. In addition, there can be four calculation machines of this kind on the same PCI bus. The total speed-up factor compared to the software implementation will then be 3200, if the speedup factor is linear. This can be even further improved by using logic devices of higher-speed and larger FPGA chips to be announced in the near future.

Several additional features can be added to this basic pipelined algorithm. These include different and more complicated fitness functions, larger populations with longer chromosomes and the use of non-overlapping generations.

Cost function evaluation in our system was extremely simple. More complicated functions can be implemented by combining logic gates and RAM tables. It seems that this combinatorial optimization problem can be solved using evolutionary programming methods. It is also possible to evaluate the fitness function in the host computer, but this may severely degrade the operating speed of the system.

Potential applications for hardware-based genetic algorithms are numerous. We have studied the use of genetic algorithms in the optimization of routing in ATM (Asynchronous Transfer Mode) networks. The initial results are promising.

*Pentium is a registered trademark of Intel Corporation

Other real-time applications which require rapid and robust optimization can also be tackled with a hardware-based genetic algorithm.

## 6.5 Bibliography

See chapter bibliography at the end of this proceedings.