Automated Analog Circuit Design Using Genetic Algorithms

ECE1352 Reading Assignment

Navid Azizi
984301910

Department of Electrical and Computer Engineering
University of Toronto

1 Introduction

Analog circuits, while being replaced by digital circuits in many cases, remain very important in high-speed applications such as communications. Analog circuit synthesis is very challenging, and has traditionally been performed by specialists who have a wealth of experience and intuition. Recently, much progression has been made in automating analog circuit synthesis using optimization algorithms. A particular optimization algorithm that has been applied to the task of automating analog circuit synthesis is the Genetic Algorithm (GA).

This paper will give an overview of how genetic algorithms have been developed and used for the synthesis of analog circuits. Section two will give a general overview of the GA. Section three will concentrate on how the GA is being applied to create custom analog circuits. Some issues that complicate the use of GAs will be presented in Section four. Section five will explain how GAs are being used with programmable analog hardware to alleviate the problems outlined in section four. Finally, Section six concludes with possibilities in the future use of GAs in analog circuit design.

2 Genetic Algorithm Overview

The genetic algorithm is one of many techniques used to find appropriate solutions for optimization problems. The GA tries to mimic the process of biological evolution where, over successive generations, individuals who are best suited to survive in an environment live on and reproduce, while other individuals die off [1]. Eventually, over many generations, the population has been optimized and only individuals who are suited to the environment live on. The GA tries to generate a viable solution to a problem through the successive evolution of substandard solutions.

GAs, in one way or another, all follow the pseudo-code document in Table 1. The algorithm starts with a set of generated solutions, which can be random or chosen to be close to the desired solution. The solutions are modeled as chromosomes and usually encoded in bit strings. Each bit (or group of bits) represents a gene. Thus a group of genes, each encoded as a bit string, can represent the characteristics of the solution in the same way that human genes represent a person's characteristics. After the first generation is generated the chromosomes are decoded into their actual representation, analyzed and given a scalar fitness value to characterize how close to the ideal solution they reside.

```
chromosome genericGeneticAlgorithm() {

   initializeGeneration();
   bestValue = calculateFitnessOfIndividuals();

   while (bestValue < desiredValue) {
      Selection();
      Crossover();
      Mutation();
      bestValue = calculateFitnessOfIndividuals();
   }

   return(chromosoneOf(bestValue));
}
```

Table 1: Generic Genetic Algorithm

Using the generated fitness values of each chromosome, a subset of the chromosomes are selected to mate and reproduce so that a new generation can be created. Selection can take many different forms including Tournament Selection, and Roulette Wheel Selection. Roulette Wheel Selection will be explained further.

In Roulette Wheel Selection each chromosome is given a proportion of the roulette wheel equal to the proportion of its fitness value to the sum of all fitness values of the generation. Then a roulette wheel is spun a number of times and the chromosome whose slot the ball lands in is selected to reproduce. An extension to the Roulette Wheel Selection always adds the best chromosome to the next generation so that the best solution cannot be lost through the generations.

Chromosomes reproduce by the process of crossover. Two parents are chosen randomly from the set of selected chromosomes and each chromosome (bit string) is cut into two at the same location. The halves are interchanged to create two children chromosomes to be included in the next generation. The parent chromosomes may or may not also be included in the next generation. Crossover can also be performed with multiple cuts, thus allowing genes in the middle of the chromosome to be interchanged.

As a final step, to avoid the set of solutions from falling into a local minimum in the overall optimization problem, some chromosomes may have their genes mutated [1]. Mutations can include a simple bit inversion in the chromosome or other more complex operations such as complete gene modifications.

The process of selection, crossover, and mutation is iterated on many times until a chromosome with a fitness value close to the desired value is found or until a maximum number

of generations have been created. The chromosome with the best fitness value at the end of the algorithm is decoded into the solution.

3 Custom Analog Circuit Design using Genetic Algorithms

Custom Analog Circuit Design implies creating a circuit topology and choosing component values and sizes to create a circuit with a desired functionality. To apply the GA to the analog circuit design problem the GA steps of Initialization, Crossover, and Mutation must be made specific and the circuit representation and fitness functions must also be defined. This section will explain the different GA flows that have been used in the synthesis of analog circuit, then it will enumerate and evaluate the different circuit representations used as chromosomes that appear in the literature, followed by the design of appropriate fitness functions for analog circuits. Finally the possibilities and tradeoffs in the steps of Crossover, and Mutations for analog circuits will be discussed.

3.1 GA Flow in Analog Circuits

GAs used for custom analog circuit design usually follow the general GA flow. The initialization step is used to create the first generation population. The population is usually generated randomly, but there are cases where the first generation is populated with solutions that are known to have characteristics close to the desired solution. Populating the first generation with known circuits has the advantage that it may lead to a quicker convergence on an appropriate solution by reducing the search space, but at the same time it may limit the GA from exploring novel designs that humans may have never conceived of [2]. The next step is to decode the population of chromosomes into a format recognizable by a circuit simulator such as SPICE. The circuit simulator will simulate each solution, and then the fitness function is applied on the simulator output data to produce a fitness measure for each solution. As in the general GA, the steps of selection, crossover, mutation, and the reevaluation of fitness measures are applied iteratively until a solution that meets the specifications is found. The general flow can be seen in Figure 1.

Before an in depth analysis of many of the GA steps used for analog circuit design are discussed, two examples contrary to the above flow for analog circuit synthesis found in [3] will be presented. One altered flow uses two GAs in succession; first a GA is used to create a circuit

topology with acceptable component values and then with the topology set a second GA is used to optimize the component values. This breaks the difficult analog circuit synthesis problem into two smaller steps. A second innovation presented in [3] is that the number of circuit components available to the GA is initialized to a small value. The GA, limited to the number of

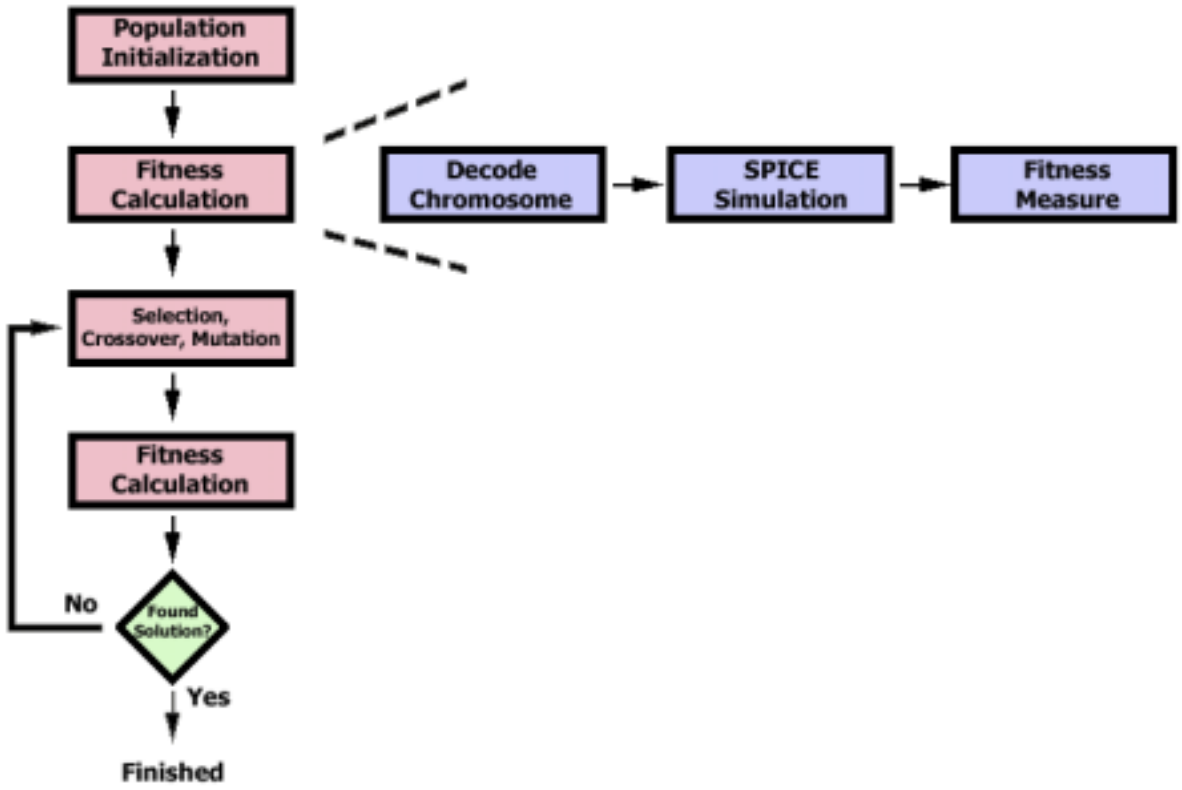

Figure 1: Genetic Algorithm Flow for Analog Circuit Synthesis

components, iterates through many generations trying to find an acceptable solution. If an acceptable solution is not found, the number of components is incremented and the whole process is repeated. This scheme ensures that not only is the final generated circuit a solution that meets the criteria, but that it is optimal in terms of its size.

3.2 Circuit Representations

One of the most important aspects of Genetic Programming is the choice of how to encode a solution, in this case a circuit representation, as a chromosome. The encoding can directly affect the ability of the iterative process to converge on an appropriate solution. In terms of analog circuits there are two pieces of information a chromosome must encode: the circuit topology and the component values such as resistor values and transistor sizes. Furthermore, for

the GA to be efficient and produce creative solutions, the circuit representation must allow almost any circuit to be encoded, and at the same time to be able to only represent valid circuits so computational effort is not wasted on useless chromosomes [3].

The problem statement can be so difficult that many analog synthesis algorithms have been developed where the topology is set, and only the component values are selected during the GA phase. In one such case [4], a State Variable Filter with a low-pass frequency response was being designed such that DC gain, cutoff frequency and selectivity factor had to achieve certain characteristics. The circuit had six resistors and two capacitors, and thus the searching the complete search-space for an appropriate solution would be an impossible task. Furthermore, the availability of only 'preferred' values for resistors and capacitors made the problem even more difficult. In the conventional design procedure many other restrictions are placed on the component values to make the problem tractable. The GA used to solve the problem introduced an 8-gene chromosome. Each gene represented one of the circuit components and had two fields: the first two bits indicated a decade value for the component from 10^3 to 10^6, and the final four bits to signify one of the 12 preferred values that a component may take in a selected decade. A sample gene can be seen in Figure 2a. With this encoding the GA was able to choose

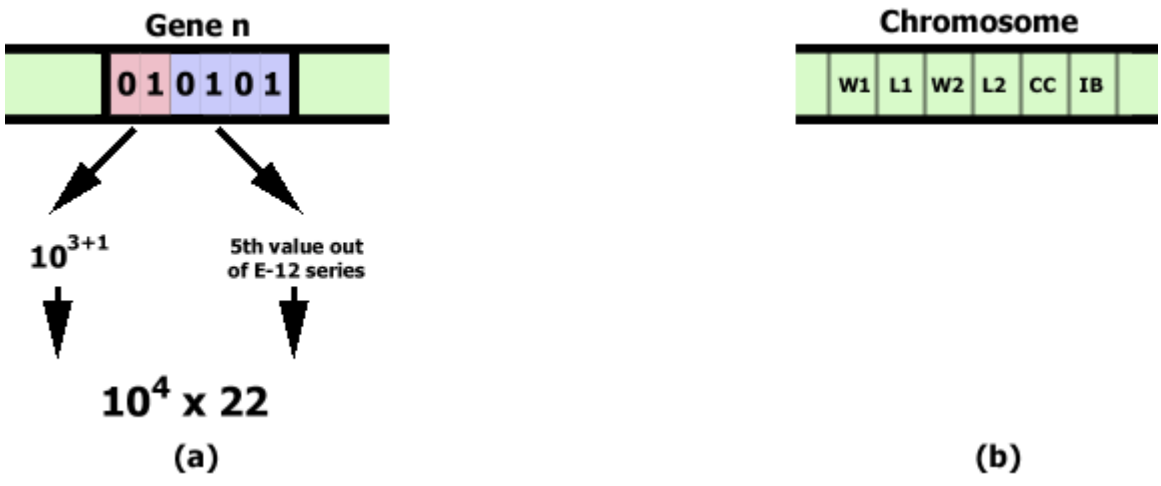

Figure 2: Different Chromosome Encodings for Component Values
(a) Encoding for a 22E4 resistor value (b) Encoding for Opamp variables

appropriate values for all components. In another example [5], in which an opamp was being designed, the GA used a 10-gene chromosome; each gene was a scalar value representing either a

transistor width, transistor length, bias current, or compensating capacitor value. The chromosome structure used in [5] can be seen in Figure 2b.

While the encoding of only the component values into the chromosome has shown to be useful, this circuit representation technique is limited to problems where a specific circuit topology is known to solve a problem well.

To be able to solve a more general class of solutions such as arbitrary filters, amplifiers with many design restrictions and analog computational circuits a GA must be able to modify the circuit topology as well as the components. The most basic way of encoding a circuit is in a netlist, which is a list of circuit components including their interconnections. One paper [3] describes a circuit representation where each gene encoded a circuit component. The gene had four fields: type, start point, end point, component value. The available component types included null, resistor, capacitor, and inductor. The null type was introduced to allow circuits of all sizes while keeping the chromosome length constant. The start points and end points were integers representing different nodes in the circuit. An example chromosome is shown in Figure 3. While this representation allows for only two-terminal passive components, a simple

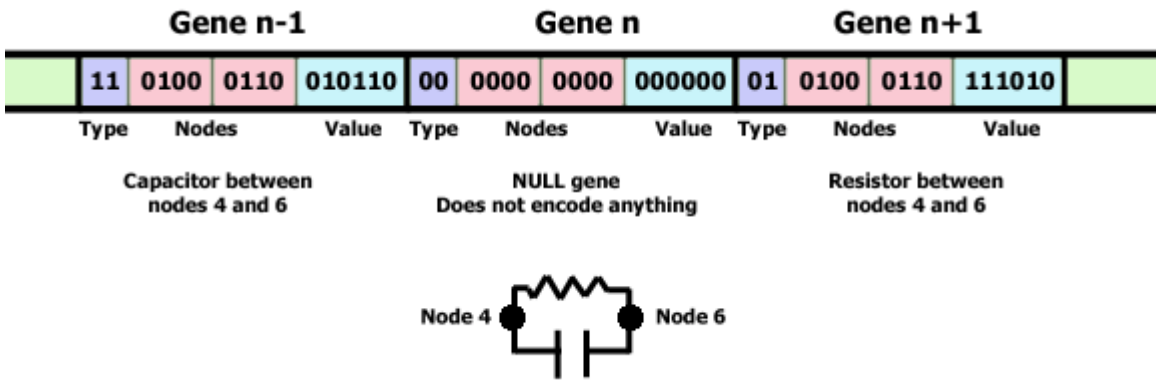

Figure 3: Netlist type circuit representation

extension would allow it to include active elements. The netlist representation is very general; any circuit imaginable can be encoded into a netlist, but at the same time a netlist can create invalid circuits. For example, a resistor having an end point that is not connected to any other circuit element. Thus, extra computational effort has to be performed when the chromosome is being decoded to prune circuit elements that make a circuit invalid. Using this netlist representation, the paper was able to create two filters within 100 generations [3].

Another way of encoding a complete circuit is to develop a circuit using a list of instructions programmed in a circuit-building language. As an assembly program is composed of a list of many different instructions to create a complete programming solution, a list of circuit-building instructions can create a circuit. In terms of the GA, each instruction is considered a gene, while the whole circuit-building program is the chromosome. Thus, the GA, through evolution, modifies the program to find a circuit that solves the problem statement. To decode the chromosome and create the final circuit, an initial circuit consisting of a wire connecting the input and output node is produced, and then each successive gene (instruction) is applied to the circuit producing a new circuit until all instructions have been processed. Two circuit-building languages used in GA will be described below.

The language presented in [6] uses 4 bytes to represent an instruction (gene): the first byte being the opcode and the final 3 bytes encoding a component value for the component used in the instruction. Five different types of instruction are available and are documented in Table 1. The instructions work by creating a new component between the active node and an outgoing node. The Active node may or may not change. The 'x' in the instruction type indicates which type of component is being added and may include resistors, capacitors, inductors and transistors.

Instruction	Outgoing Node	Active Node
x-move-to-new	New Node	New Node
x-cast-to-previous	Previous Node	Unchanged
x-cast-to-ground	Ground Node	Unchanged
x-cast-to-input	Input Node	Unchanged
x-cast-to-output	Output Node	Unchanged

Table 2: Instruction Types

Transistors can be used in this scheme even though they are three terminal devices by indicating which node the third terminal is connected to within the 'x' portion of the instruction. For example a "emitter to previous-move-to-new" instruction would create a transistor with its base connected to the active node, its collector connected to the newly created node and its emitter connected to the previous node. To cover all transistor possibilities, there must be fifty-four different transistor configurations for the 'x' portion of the instruction [6]. Some include 'emitter to GND', 'collector to GND', 'emitter to VDD', 'collector to Output', and 'collector to base'. A desirable characteristic of this approach to representing circuits is that it can only encode valid circuit graphs and thus computational effort is not wasted in creating and decoding useless chromosomes [6]. The circuit representation technique, however, does not allow all possible circuits to be constructed, but [6] argues that it allows for many circuit topologies to be created

including many seen in hand-designed circuits. Figure 4 illustrates the decoding of a gene within the above framework.

Another circuit building language is described in [7] is very different to the one described in [6]. The circuit-building process resembles the use of threads within a computer program in that many parts of the circuit may be expanding concurrently. When a new circuit element is added, computation on the previous element does not end, but instead works on the old element and the new element continue in tandem. Furthermore, component values are not set when a component is placed; rather the value is computed within a separate thread [7]. The language

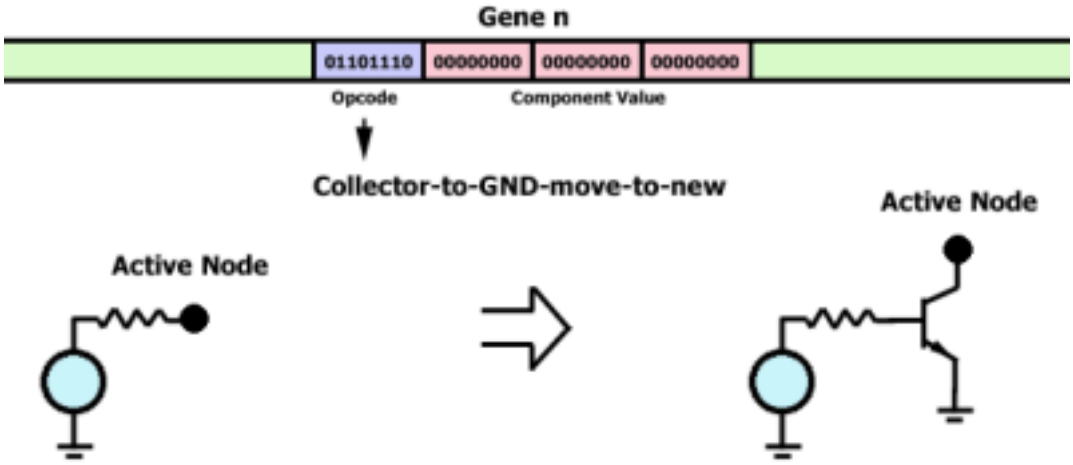

Figure 4: Instruction Type Circuit Representation

also has five types of instructions that are listed and described in Table 3.

This type of representation allows for the creation of any circuit structure starting with a single wire, but it also allows for the creation of invalid circuits. Invalid circuits are corrected by connecting floating nodes to the output through a large resistor [7].

Instruction Type	Description
Topology Modifying	Copies selected resource and places it in series or in parrallel to itself, creates wires to gnd, vdd, between arbitrary nodes
Component Creating	Changes selected resource (including wires) to a new component
Development Controlling	Allows for thread creation and termination
Arithmentic Performing	Used to caluclate component values

Table 3: Instruction Types

There are many different circuit representations used to create analog circuits within a GA framework with each having their advantages and their disadvantages and the correct choice depends on many other factors with the whole system.

3.3 Fitness Function

To be able to continuously improve solutions from generation to generation a fitness function is needed that accurately describes how close a possible solution is to the desired specification. The main challenge of creating fitness functions for the optimization of multiple objectives is the need to integrate a vector of performance measures into a single scalar value [5]. Within the scope of analog circuit design with genetic programming two types of circuits repeatedly come under question: filters and amplifiers. Possible fitness functions for these two types of circuits will be described below.

Filters have the characteristic that they must allow frequency within a passband to go through a circuit unattenuated while frequencies in the stopband must be completely attenuated. Furthermore there are frequencies between the passband and the stopband that are not important in the filter characteristic. A simple fitness function used for filter design is: $\sum_{i=1}^{n} \omega_i |S(f_i) - O(f_i)|$ where S is the ideal transfer function, O is the actual transfer function and ω is a weight [8]. Thus the fitness function is a weighted sum of the deviations between the ideal gain and the actual gain over a set of n points. The weight values are used to make deviations in the stopband and passband more important than deviations in the region between the stopband and passband. A simple extension to this idea found in [7] converts the weightings from simple scalars to a function depending, not only on the frequency, but also on the deviation. Deviations larger than an acceptable ripple along the passband and stopband are weighted more than deviations within the acceptable ripple. This extension gives far worse fitness results to solutions that are far from an acceptable characteristic allowing for faster convergence upon a solution [7].

Fitness functions for amplifier designs, commonly, are also a weighted sum of deviations. In [6] the weighted sum includes parameters such as dc gain, dc bias, power dissipation and linearity. This solution has shown to work well in many cases, however, other fitness functions that may allow for faster convergence to an acceptable solution have been designed. One such method in [5] assigns large weights to objectives for which the average value of a constraint for the whole population is far from the target value, and assigns low weights to objectives whose average values are around the desired ones. The GA search is then driven by the requirements that are harder to achieve within the specific population [5]. The exact fitness functions can be seen in Table 4. The overall fitness is a weighted sum of normalized finesses for each objective.

The weight for each objective is determined by comparing the average fitness of the characteristic with two user-defined values, a desired value and an acceptable value.

$$Fitness = \sum \omega_i Fn_i$$

$$Fn_i = \frac{F_i}{\bar{F}_i}$$

$$\omega_i = \frac{100 user_i}{\bar{F}_i} \text{ if } \bar{F}_i < accep_i$$

$$\omega_i = \frac{10 user_i}{\bar{F}_i} \text{ if } accep_i < \bar{F}_i < user_i$$

$$\omega_i = 1 \text{ if } \bar{F}_i > user_i$$

Table 4: Example Fitness Function for an Opamp

Three other techniques for developing fitness functions have been examined in [6] where the fitness function changes through the generations. The motivation for this technique is that dynamic fitness functions can guide the population of circuits through the solution space to an acceptable answer [6]. The first method increases the user requirements for the circuit from easy to difficult in each successive generation by a set value, trying to guide the population to the final answer. A second version, similar to the first, increases the difficulty of each requirement only when the majority of the population has met the current requirement. Finally, a third method allows the fitness requirements to evolve just as the population evolves. For example, for an amplifier circuit, the algorithm decreased the gain requirement over the first few generations because it was the hardest requirement to meet, and as the generations evolved and the other requirements were being met, the gain requirement was made more difficult [6]. The first two methods did not work well, but the evolving fitness function produced very good results. [6] explained the differences by indicating that the evolution of the fitness function keeps the "level of the problem difficulty near the leading edge of circuit proficiency," while the other two fitness functions where designed arbitrarily [6].

3.4 Selection, Crossover and Mutation for Analog Circuits

Throughout the literature, selection and crossover methods used for analog circuit synthesis vary greatly and include Tournament Selection, Uniform Selection, and Roulette-Style Selection, but the selection process is no different in GAs used for analog circuit synthesis compared to any other GA. Crossover techniques used in GAs for analog circuit synthesis are

also no different to crossover techniques used within other GAs. The single-point crossover was the most predominant form of crossover for analog circuits, but its rate varied from 0.3 to 0.8. [3,4].

The process of mutation, as opposed to selection and crossover, can differ for analog circuit synthesis GAs compared to other GAs. While many algorithms still perform mutation at the bit level for each chromosome (each bit has a low probability of switching), mutation can also happen at the circuit level. Possible mutation can include series mutations, where a new component is inserted in series with the mutated gene, or a change-element mutation where the component type is changed [3]. While these types of mutations can only be used with certain circuit representations, their use can lead to mutations that are known to create valid circuits thus saving computational effort.

4 Issues involved with Analog Circuit Synthesis using GAs

Genetic algorithms can create filters, amplifiers, controllers and many other analog circuits, but their use has still not become mainstream. GAs have some drawbacks including the computational time and effort needed to obtain a solution and with their inability to deal with the non-idealities of components and process variations that can make a circuit behave unexpectedly.

A GA can create a circuit with appropriate transistor sizings and component selections, and while the circuit behaves well under simulation when it is fabricated the non-idealities and process variations cause the circuit to fail in meeting its specifications [9]. It can seam that the GA has over-optimized the circuit's parameters to only work in a certain configuration. One possible solution is to replace ideal components with their non-ideal equivalent models before the circuit is simulated. This technique forces the GA to create a circuit that is more robust to component non-idealities. In [9] non-idealities were included in the simulation and it was showed that after the circuit was created it still met the specifications. Furthermore, a circuit can be simulated under various process variations, with its fitness measure being a weighted average of the individual fitness measures, so that a circuit that is resistant to process variations will reproduce more often.

Another problem with GAs used for analog circuit synthesis is that a large amount of computational resources are needed. While all GAs tend to be computationally intensive, GAs for circuit synthesis need increased computational resources since the simulation of circuits may

take a considerable amount of time. Take for example a population of 200 chromosomes evolving over 100 generations with each simulation only taking one second, then the time spent only on simulations can take 5.5 hours. Thus, for the synthesis of complex circuits the GA may take an extraordinary amount of time. One solution is to perform the simulations in a distributed manner; a master program that performs the selection, crossover and mutation can hand over chromosomes to other computers which can simultaneously simulate and analyze the chromosome and return the fitness function back to the master program. This can cut down the computational time needed, but more hardware is needed.

A further possibility is that many worker programs all work simultaneously on a breeding population. Each worker program randomly picks two members, performs a crossover, and places the offspring back in the population (but only if their fitness is higher than the average fitness) [10]. The offspring do not immediately become part of the breeding population, but must wait until a master program places them in the breeding population. The master program has the responsibility of removing members with low fitness from the breeding population and bringing in new offspring with high fitness [10]. This configuration was tested in [10] with 20 Sun IPC workstations and a speedup of 15 times was observed over a non-parallel version of the algorithm. A scenario with this framework can be seen in Figure 5.

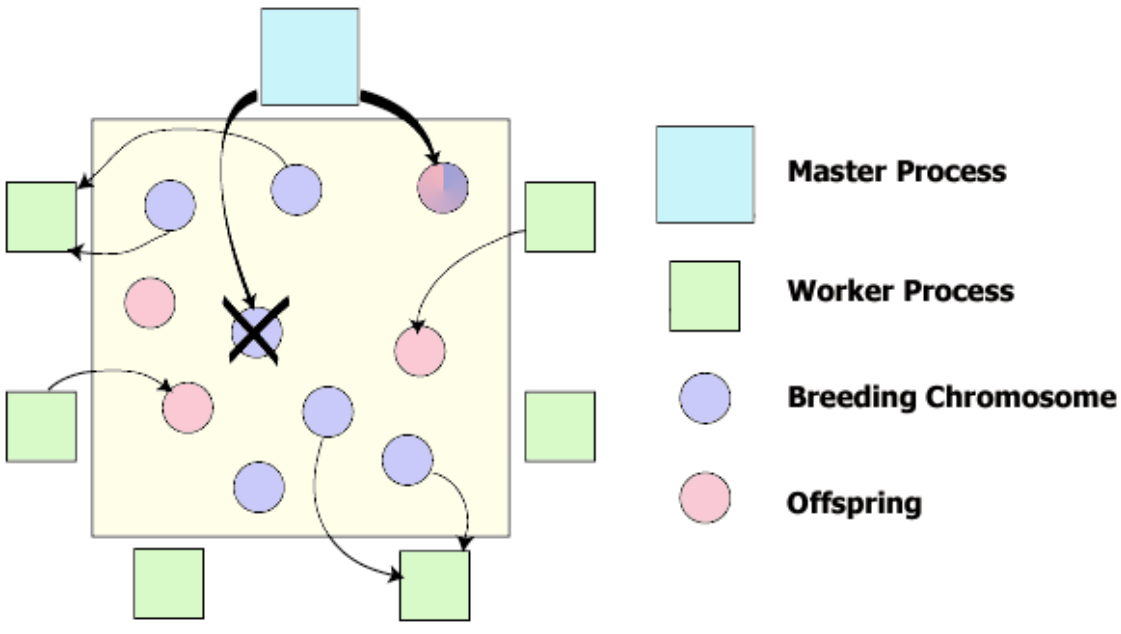

Figure 5: Parallel Genetic Algorithm

5 Analog Circuit Design with GAs and Programmable Hardware

Another application of the GA for analog circuit synthesis is its use with programmable hardware. The GA's use with programmable hardware actually removes many of the limitations involved with the custom circuit synthesis problem. Firstly, the circuit can be tested outright on the actual hardware thus removing the process variations and circuit non-idealities; these are factored in the test outputs and consequently the chromosomes' fitness. Furthermore, testing the actual circuit takes much less time than through simulation thus saving computational resources.

The GA used with programmable hardware remains very much the same as the GA described in section three; fitness functions, crossover rates and most other parameters do not change. The circuit representation in the chromosome, however, changes from being an encoded circuit in the custom circuit design process to using the architecture bits of the programmable hardware. By using the architecture bits there is no need for a decoding step, but rather the architecture bits are directly downloaded to the hardware and then test inputs are applied to the circuit (see Figure 6).

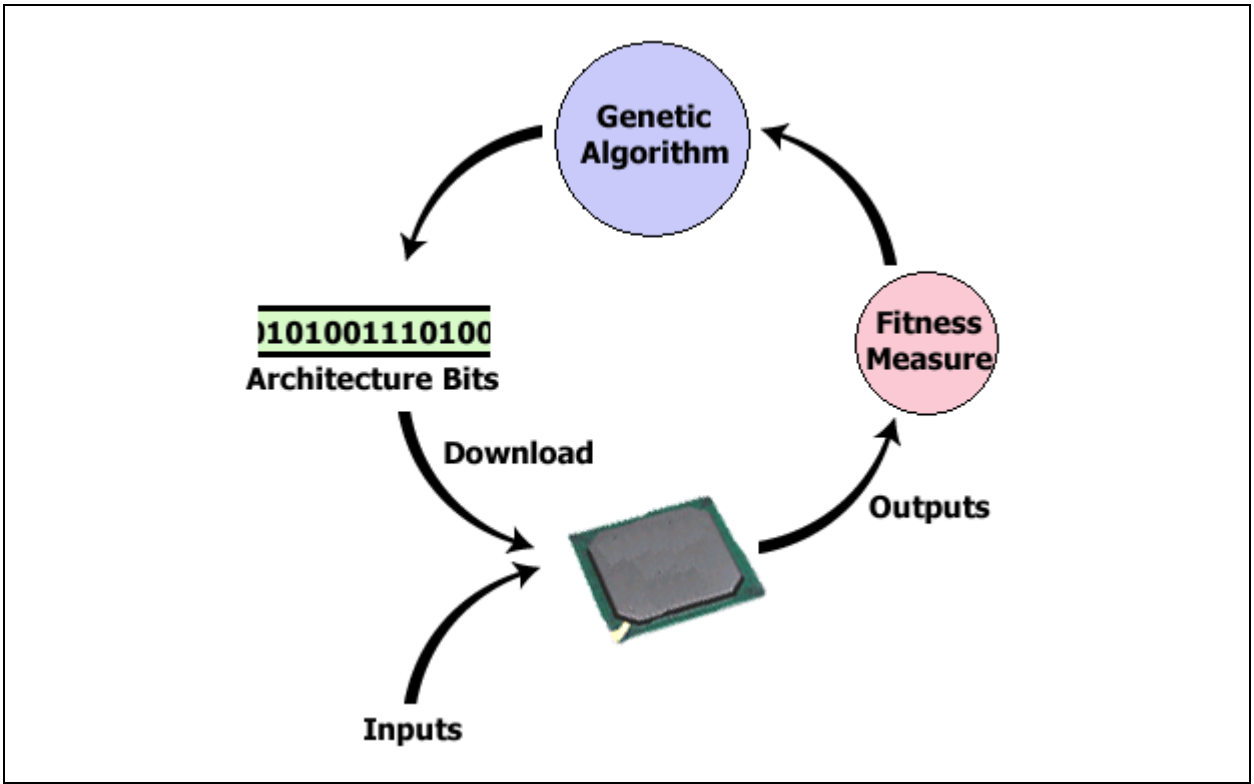

Figure 6: The Genetic Algorithm used with Programmable Hardware

In [8] an analog programmable chip, whose architecture bits controlled various transconductances within the circuit was used to create an intermediate frequency filter.

Surprisingly Field-Programmable-Gate-Arrays (FPGAs), which are digital programmable hardware chips, can also be used to create analog functions with the help of GAs. A frequency detecting circuit was constructed on an FPGA using its architecture bits as the chromosome within a GA [11]. The realized circuit detected signals at 10 kHz by output a logic '1'. To try to understand the nature of the solution, the authors decomposed the circuit into logic gates, but could not understand how the circuit worked. They, finally, tried probing the internal nodes of the circuit but the probing destroyed the circuit's functionality. In the end, the authors could not fully understand the realized circuit, and concluded that the circuit used the FGPAs inherent analog nature to produce a correct output [11]. This final example shows the power of the GA to create analog circuits.

6 Conclusion

Genetic algorithms can be powerful tools for automated analog circuit synthesis. Many successful examples exist where filters, amplifiers with many requirement, and analog computational circuits among other analog circuits have been designed with no human intervention. With the advancement of analog programmable hardware, the use of genetic algorithm for analog circuit synthesis will surely increase.

Bibliography

[1] M. Mitchell and S. Forrest, “Genetic Algorithms and Artificial Life,” [Online document], 1993 November, [cited 2001 Oct. 21], Available HTTP: http://www.santafe.edu/~mm/GA.Alife.ps

[2] N. Fujii and H. Shibatu, “Analog Circuit Synthesis by Superimposing of Sub-Circuits,” Proceedings of the International Symposium on Circuits and Systems, vol. 5, no. 1, pp. 427-430, 2001.

[3] C. Goh and Y. Li, " GA Automated Design and Synthesis of Analog Circuits with Practical Constraints," Proceedings of the 2001 Congress on Evolutionary Computation, vol. 1, no. 1, pp. 170-7, 2001.

[4] D.H Horrocks, and M.C Spittle, " Component Value Selection for Active Filters Using Genetic Algorithms," Proceedings of IEE Workshop on Natural Algorithms in Signal Processing, vol. 1, no. 1, pp. 131-136, 1993. Available HTTP: http://citeseer.nj.nec.com/89190.html

[5] R.S Zebeulum, M.A. Pacheco and M. Vellasco, " Synthesis of CMOS Operational Amplifiers Through Genetic Algorithms," Proceedings of the XI Brazilian Symposium on Integrated Circuit Design, vol. 11, no. 1, pp. 125-8, 1998.

[6] J.D. Lohn, G.L. Haith, S.P. Colombana and D. Stassinopoulos, " Towards Evolving Electronic Circuits for Autonomous Space Applications," Proceedings of the 2000 IEEE Aerospace Conference, vol. 5, no. 1, pp. 476-86, 2000.

[7] F.H. Bennet, M.A. Keane D.Andre, and J.R Koza, " Automatic Synthesis of the Topology and Sizing for Analog Electrical Circuits Using Genetic Programming," Evolutionary Algorithms in Engineering and Computer Science, 1 ed., Ed. K Miettinen, P. Neittaanmaki, M.M. Makela, J. Périaux, John Wiley & Sons Ltd., 1999, pp. 199-229.

[8] M. Murakawa, S. Yoshzawa, T. Adachi, S. Suzuki, K. Takasuka, M. Iwata and T. Higuchi, " Analogue EHW Chip for Intermediate Frequency Filters," Evolvable Systems: From Biology to Hardware, Proceedings of the Second International Conference, 1 ed., Ed. M. Sipper, D. Mange, and A. Pérez-Uribe, Lusanne, Switzerland: Springer, 1998, pp. 134-143.

[9] D.H Horrocks, and Y.M.A. Khalifa, " Genetic Algorithm Design of Electronic Analogue Circuits Including Parasitic Effects," Proceedings of the First Online Workshop on Soft Computing, vol. 1, no. 1, pp. 274-278, 1996. Available HTTP: http://citeseer.nj.nec.com/horrocks96genetic.html

[10] M. Davis, L. Liu, J.G. Elias, " VLSI Circuit Synthesis Using a Parallel Genetic Algorithm," Proceedings of the First IEEE Conference on Evolutionary Computation, vol. 1, no. 1, pp. 104-9, 1994.

[11] A. Thompson, P. Layzell, " Analysis of Unconventional Evolved Electronics," Communications of the ACM, vol. 42, no. 4, pp. 71-9, 1999.