Evolution of a Low-Distortion, Low-Bias 60 Decibel Op Amp with Good Frequency Generalization using Genetic Programming

John R. Koza
Computer Science Dept.
258 Gates Building
Stanford University
Stanford, California 94305
koza@cs.stanford.edu

David Andre
Visiting Scholar
Computer Science Dept.
Stanford University
Stanford, California 94305
andre@flamingo.stanford.edu

Forrest H Bennett III
Visiting Scholar
Computer Science Dept.
Stanford University
Stanford, California 94305
fhb3@slip.net

Martin A. Keane
Econometrics Inc.
5733 West Grover
Chicago, Illinois 60630
makeane@ix.netcom.com

ABSTRACT

Genetic programming was used to evolve *both* the topology and the sizing (numerical values) for each component of a low-distortion, low-bias 60 decibel (1000-to-1) amplifier circuit with good frequency generalization. The evolved circuit was composed of two types of transistors (active elements) as well as resistors and capacitors.

1. Introduction

The problem of circuit synthesis involves designing an electrical circuit that satisfies user-specified design goals. A complete design of an electrical circuit includes both its topology and the sizing of all its components. The *topology* of a circuit consists of the number of components in the circuit, the type of each component, and a list of all the connections between the components. The *sizing* of a circuit consists of the component value(s) of each component.

Evolvable hardware is one approach to automated circuit synthesis. Some of the early pioneering work in this field includes that of Higuchi, Niwa, Tanaka, Iba, de Garis, and Furuya (1993), Hemmi, Mizoguchi, and Shimohara (1994); Mizoguchi, Hemmi, and Shimohara (1994); and the work presented at the 1995 workshop on evolvable hardware in Lausanne (Sanchez and Tomassini 1996).

The design of analog circuits and mixed analog-digital circuits has not proved to be amenable to automation (Rutenbar 1993). In DARWIN (Kruiskamp and Leenaerts 1995), a CMOS op amp circuit was evolved using a genetic algorithm. However, the topology of each op amp was one of 24 hand-designed topologies. Thompson (1996) used a genetic algorithm to evolve a frequency discriminator on a Xilinx 6216 reconfigurable processor in analog mode.

2. Genetic Programming

Genetic programming is an extension of the genetic algorithm described in John Holland's pioneering *Adaptation in Natural and Artificial Systems* (1975). The book *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (Koza 1992) provides evidence that genetic programming can solve, or approximately solve, a variety of problems. See also Koza and Rice 1992. The book *Genetic Programming II: Automatic Discovery of Reusable Programs* (Koza 1994a, 1994b) describes multi-part programs consisting of a main program and one or more reusable, parameterized, hierarchically-called subprograms. See Kinnear 1994, Angeline and Kinnear 1996, Koza, Goldberg, Fogel, and Riolo 1996 for recent work.

Gruau's *cellular encoding* (1996) is an innovative technique in which genetic programming is used to concurrently evolve the architecture, weights, thresholds, and biases of neurons in a neural network.

3. Evolution of Circuits

Genetic programming can be applied to circuits if a mapping is established between the kind of rooted, point-labeled trees with ordered branches found in the world of genetic programming and the line-labeled cyclic graphs encountered in the world of circuits.

Developmental biology provides the motivation for this mapping. The starting point of the growth process used herein is a very simple embryonic electrical circuit. There is a circuit-constructing program tree of the type ordinarily used in genetic programming. An electrical circuit is then developed by progressively executing the functions in the program tree. These functions manipulate the embryonic circuit (and its successors) so as to produce valid electrical circuits at each step.

The functions are divided into four categories:

(1) connection-modifying functions that modify the topology of circuit (starting with the embryonic circuit), and

(2) component-creating functions that insert components into the topology of the circuit,

(3) arithmetic-performing functions that appear in arithmetic-performing subtrees as argument(s) to the component-creating functions and specify the numerical value of the component, and

(4) automatically defined functions in function-defining branches.

Each branch of the program tree is created in accordance with a constrained syntactic structure. Branches are composed from construction-continuing subtree(s) that continue the developmental process and arithmetic-performing subtree(s) that determine the numerical value of the component. Connection-modifying functions have one or more construction-continuing subtrees, but no arithmetic-performing subtrees. Component-creating functions have one construction-continuing subtree and typically have one arithmetic-performing subtree. This constrained syntactic structure is preserved by using structure-preserving crossover with point typing (Koza 1994a).

3.1. The Embryonic Circuit

The developmental process for converting a program tree into an electrical circuit begins with an embryonic circuit.

Figure 1 shows a one-input, one-output embryonic circuit that serves as a test harness for the evolving circuits. VSOURCE is the incoming signal. VOUT is the output signal. There is a fixed 1,000 Ohm load resistor RLOAD and a fixed 1,000 Ohm source resistor RSOURCE. Because we are evolving an amplifier, there is also a fixed 1,000,000 Ohm feedback resistor RFEEDBACK, a fixed 1,000 Ohm balancing source resistor RBALANCE_SOURCE, and a fixed 1,000,000 Ohm balancing feedback resistor RBALANCE_FEEDBACK. This arrangement limits the possible amplification of the evolving circuit to the 1000-to-1 ratio (which corresponds to 60 dB) of the feedback resistor to the source resistor.

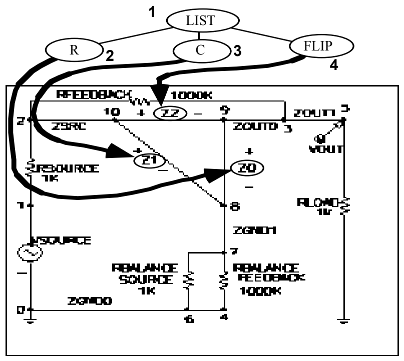

Figure 1 Embryonic circuit.

There are three modifiable wires Z0, Z1, and Z2 arranged in a triangle so as to provide connectivity between the input, the output, and ground. All of the above elements (except Z0, Z1, and Z2) are fixed and not modified during the developmental process. At the beginning of the developmental process, there is a writing head pointing to (highlighting) each of the three modifiable wires. All development occurs at wires or components to which a writing head points.

The top part of this figure shows a portion of an illustrative circuit-constructing program tree. It contains a resistor-creating R function (labeled 2 and described later), a capacitor-creating C function (labeled 3), and a polarity-reversing FLIP function (labeled 4) and a connective LIST function (labeled 1). The R and C functions cause modifiable wires Z0 and Z1 to become a resistor and capacitor, respectively; the FLIP function reverses the polarity of modifiable wire Z2.

3.2. Component-Creating Functions

Each circuit-constructing program tree in the population contains component-creating functions and connection-modifying functions.

Each component-creating function inserts a component into the developing circuit and assigns component value(s) to the component. Each component-creating function has a writing head that points to an associated highlighted component in the developing circuit and modifies the highlighted component in some way. The construction-continuing subtree of each component-creating function points to a successor function or terminal in the circuit-constructing program tree.

The arithmetic-performing subtree of a component-creating function consists of a composition of arithmetic functions (addition and subtraction) and random constants (in the range –1.000 to +1.000). The arithmetic-performing subtree specifies the numerical value of a component by returning a floating-point value that is, in turn, interpreted, in a logarithmic way, as the value for the component in a range of 10 orders of magnitude (using a unit of measure that is appropriate for the particular type of component involved). The floating-point value is interpreted as the value of the component as described in Koza, Andre, Bennett, and Keane (1996, 1997)

The two-argument resistor-creating R function causes the highlighted component to be changed into a resistor. The value of the resistor in kilo-Ohms is specified by its arithmetic-performing subtree.

Figure 2 shows a modifiable wire Z0 connecting nodes 1 and 2 of a partial circuit containing four capacitors.

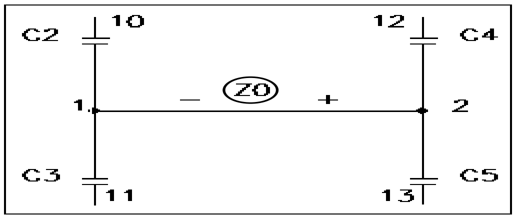

Figure 2 Modifiable wire Z0.

Figure 3 shows the result of applying the R function to the modifiable wire Z0 of figure 2.

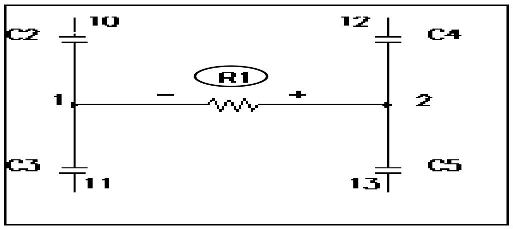

Figure 3 Result of applying the R function.

Similarly, the two-argument capacitor-creating C function causes the highlighted component to be changed into a capacitor. The value of the capacitor in nano Farads is specified by its arithmetic-performing subtree.

Space does not permit a detailed description of each function herein. See Koza, Andre, Bennett, and Keane

(1996, 1997), and Koza, Bennett, Andre, and Keane (1996a, 1996b, 1996c, 1996d) for details.

The one-argument Q_D_PNP diode-creating function causes a diode to be inserted in lieu of the highlighted component, where the diode is implemented using a PNP transistor whose collector and base are connected to each other. The Q_D_NPN function inserts a diode using an NPN transistor in a similar manner.

There are also six one-argument transistor-creating functions (called Q_POS_COLL_NPN, Q_GND_EMIT_NPN, Q_NEG_EMIT_NPN, Q_GND_EMIT_PNP, Q_POS_EMIT_PNP, Q_NEG_COLL_PNP) that insert a transistor in lieu of the highlighted component. For example, the Q_POS_COLL_NPN function inserts a NPN transistor whose collector is connected to the positive voltage source.

The three-argument transistor-creating Q_3_NPN function causes an NPN transistor (model Q2N3904) to be inserted in place of the highlighted component and one of the nodes to which the highlighted component is connected. The Q_3_NPN function creates five new nodes and three new modifiable wires. There is no writing head on the new transistor. Similarly, the three-argument transistor-creating Q_3_PNP function causes a PNP transistor (model Q2N3906) to be inserted.

Figure 4 shows the result of applying the Q_3_NPN0 function, thereby creating transistor Q6 in lieu of modifiable wire Z0 of figure 2.

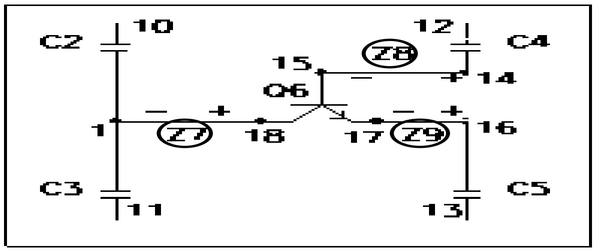

Figure 4 Result of applying Q_3_NPN0 function.

3.3. Connection-Modifying Functions

Each connection-modifying function in a program tree points to an associated highlighted component and modifies the topology of the developing circuit in some way.

The one-argument polarity-reversing FLIP function attaches the positive end of the highlighted component to the node to which its negative end is currently attached and vice versa. After execution of the FLIP function, there is one writing head pointing to the component.

The three-argument SERIES division function creates a series composition consisting of the highlighted component, a copy of it, one new modifiable wire, and two new nodes (each with a writing head).

Figure 5 illustrates the result of applying the SERIES division function to resistor R1 from figure 3.

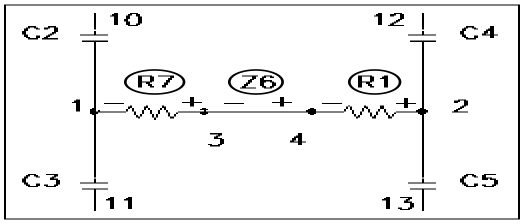

Figure 5 Result after applying the SERIES function.

The four-argument PSS and PSL parallel division functions create a parallel composition consisting of the original highlighted component, a copy of it two new wires, and two new nodes. Figure 6 shows the result of applying PSS to the resistor R1 from figure 3.

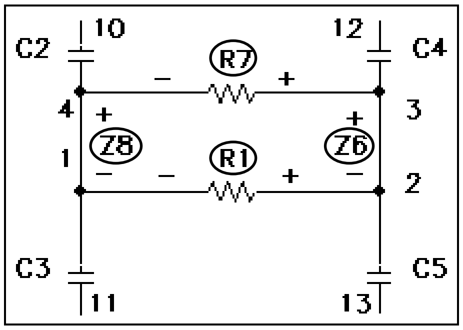

Figure 6 Result of the PSS parallel division function.

There are six three-argument functions (called T_GND_0, T_GND_1, T_POS_0, T_POS_1, T_NEG_0, T_NEG_1) that insert two new nodes and two new modifiable wires and make a connection to ground, positive voltage source, or negative voltage source, respectively. Figure 7 shows the T_GND_0 function connecting resistor R1 to ground.

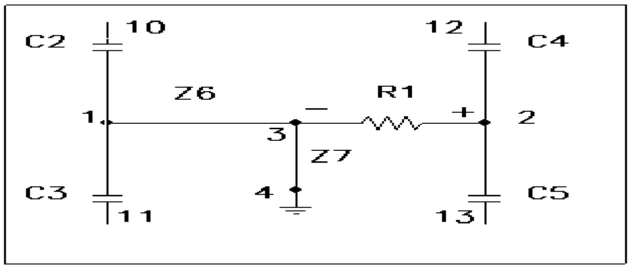

Figure 7 Result of applying the T_GND_0 function.

The three-argument PAIR_CONNECT_0 and PAIR_CONNECT_1 functions enable distant parts of a circuit to be connected together. The first PAIR_CONNECT to occur in the development of a circuit creates two new wires, two new nodes, and one temporary port. The next PAIR_CONNECT to occur (whether PAIR_CONNECT_0 or PAIR_CONNECT_1) creates two new wires and one new node, connects the temporary port (TEMP) to the end of one of these new wires, and then removes the temporary port.

The one-argument NOP function has no effect on the highlighted component; however, it delays activity on the developmental path on which it appears in relation to other developmental paths in the overall program tree.

The zero-argument END function causes the highlighted component to lose its writing head. The END function causes its writing head to be lost – thereby ending that particular developmental path.

The zero-argument SAFE_CUT function causes the highlighted component to be removed from the circuit provided that the degree of the nodes at both ends of the highlighted component is three (i.e., no dangling components or wires are created).

4. Preparatory Steps

Our goal in this paper is to evolve a design a 60 decibel amplifier with low distortion and low bias. Before applying genetic programming to circuit synthesis, the user must perform seven major preparatory steps, namely

(1) identifying the embryonic circuit that is suitable for the problem, (2) determining the architecture of the overall circuit-constructing program trees, (3) identifying the terminals of the to-be-evolved programs, (4) identifying the primitive functions contained in the to-be-evolved programs, (5) creating the fitness measure, (6) choosing certain control parameters (notably population size and the maximum number of generations to be run), and (7) determining the termination criterion and method of result designation. The feedback embryo of figure 1 is suitable for this problem.

Since the embryonic circuit has three writing heads – one associated with each of the result-producing branches – there are three result-producing branches (called RPB0, RPB1, and RPB2) in each program tree. We decided to include two one-argument automatically defined functions (called ADF0 and ADF1) in each program tree and that there would be no hierarchical references among the automatically defined functions. Thus, there are two function-defining branches in each program tree. Consequently, the architecture of each overall program tree in the population consists of a total of five branches (i.e., two function-defining branches and three result-producing branches) joined by a LIST function.

The terminal sets are identical for all three result-producing branches of the program trees. The function sets are identical for all three result-producing branches. For the three result-producing branches, the function set, $\mathcal{F}_{\text{ccs-rpb}}$, for each construction-continuing subtree is

$\mathcal{F}_{\text{ccs-rpb}}$ = {ADF0, ADF1, R, C, SERIES, PSS, PSL, FLIP, NOP, T_GND_0, T_GND_1, T_POS_0, T_POS_1, T_NEG_0, T_NEG_1, PAIR_CONNECT_0, PAIR_CONNECT_1, Q_D_NPN, Q_D_PNP, Q_3_NPN0, ..., Q_3_NPN11, Q_3_PNP0, ..., Q_3_PNP11, Q_POS_COLL_NPN, Q_GND_EMIT_NPN, Q_NEG_EMIT_NPN, Q_GND_EMIT_PNP, Q_POS_EMIT_PNP, Q_NEG_COLL_PNP}.

For the three result-producing branches, the terminal set, $\mathcal{T}_{\text{ccs-rpb}}$, for each construction-continuing subtree consists of

$\mathcal{T}_{\text{ccs-rpb}}$ = {END, SAFE_CUT}.

For the function-defining branches (automatically defined functions), the function set, $\mathcal{F}_{\text{ccs-adf}}$, for each construction-continuing subtree is

$\mathcal{F}_{\text{ccs-adf}} = \mathcal{F}_{\text{ccs-rpb}}$ – {ADF0, ADF1}.

The terminal sets are identical for both function-defining branches (automatically defined functions) of the program trees. The function sets are identical for both function-defining branches (automatically defined functions). For the function-defining branches, the terminal set, $\mathcal{T}_{\text{ccs-adf}}$, for each construction-continuing subtree is

$\mathcal{T}_{\text{ccs-adf}} = \mathcal{T}_{\text{ccs-adf}} \approx$ {ARG0},

where ARG0 is the dummy variable (formal parameter) of the automatically defined function.

The terminal set, $\mathcal{T}_{\text{aps}}$, for each arithmetic-performing subtree consists of

$\mathcal{T}_{\text{aps}} = \{\leftarrow\}$,

where ← represents floating-point random constants from –1.0 to +1.0.

The function set, $\mathcal{F}_{\text{aps}}$, for each arithmetic-performing subtree is,

$\mathcal{F}_{\text{aps}}$ = {+, -}.

each taking two arguments.

The evaluation of fitness for each individual circuit-constructing program tree in the population begins with its execution. This execution applies the functions in the program tree to the embryonic circuit, thereby developing the embryonic circuit into a fully developed circuit. A netlist describing the fully developed circuit is then created. The netlist identifies each component of the circuit, the nodes to which that component is connected, and the value of that component. Each circuit is then simulated to determine its behavior. The 217,000-line SPICE simulator was modified to run as a submodule within the genetic programming system. SPICE (Simulation Program with Integrated Circuit Emphasis) is a large simulation program written at the University of California (Quarles et al. 1994).

An amplifier can be viewed in terms of its response to a DC input. An ideal amplifier circuit would receive a DC input, invert it, and multiply it by the amplification factor. A circuit is flawed to the extent that it does not achieve the desired amplification; to the extent that the output signal is not centered on 0 volts (i. e., it has a bias); and to the extent that the DC response of the circuit at several different DC input voltages is not linear.

Thus, for this problem, we used a fitness measure based on SPICE's DC sweep. The DC sweep analysis measures the DC response of the circuit at several different DC input voltages. The circuits were analyzed with a 5 point DC sweep ranging from –10 millvolts to +10 MV, with input points at –10 MV, –5 MV, 0 MV, +5 MV, and +10 MV. SPICE produced the circuit's output for each of these input values.

Fitness is based on the four penalties derived from these five DC output values It is the sum of the amplification penalty, the bias penalty, and the two non-linearity penalties.

First, the amplification factor of the circuit is measured using the overall value for gain of the circuit as measured by the slope of the straight line between the output for –10 MV and the output for +10 MV (i.e., between the outputs for the endpoints of the DC sweep). If the amplification factor is less than the target (which is 60 dB for this problem), there is a penalty equal to the shortfall in amplification.

Second, the bias is computed using the DC output associated with a DC input of 0 volts. There is a penalty equal to the bias times a weight (1.0 for this problem).

Finally, the linearity is measured by the deviation between the slope of each of two shorter lines and the overall amplification factor of the circuit. The first shorter line segment connects the output value associated with an input of –10 MV and the output value for –5 MV. The second shorter line segment connects the output value for +5 MV and the output for +10 MV. There is a penalty for each of these shorter line segments equal to the absolute value of the difference in slope between the respective shorter line segment and the slope of the straight line between the output for –10 MV and the output for +10 MV.

Many of the circuits that are randomly created in the initial random population and that are subsequently created by the crossover and mutation operations are so bizarre that they cannot be simulated by SPICE. Such circuits are assigned a high penalty value of fitness (10^8).

The population size, *M*, was 640,000. The crossover percentage was 89%; the reproduction percentage was 10%; and the mutation percentage was 1%. A maximum size of 300 points was established for each of the branches in each overall program. The other minor parameters were the default values in Koza 1994a (appendix D).

This problem was run on a medium-grained parallel Parsytec computer system consisting of 64 80 MHz Power PC 601 processors arranged in a toroidal mesh with a host PC Pentium type computer. The distributed genetic algorithm was used with a population size of *Q* = 10,000 at each of the *D* = 64 demes. On each generation, four boatloads of emigrants, each consisting of *B* = 2% (the migration rate) of the node's subpopulation (selected on the basis of fitness) were dispatched to the four toroidally adjacent processing nodes. See Andre and Koza 1996.

5. Results

The best circuit (figure 11) from generation 0 achieves a fitness of 986.1 and has nine transistors, three capacitors, and two resistors (and five resistors of the embryo).

About 45% of the circuits of generation 0 cannot be simulated by SPICE. However, the percentage of unsimulatable programs drops to 8% by generation 1. Moreover, this percentage does not exceed 2% after generation 16 and does not exceed 1% after generation 58.

The best circuit (figure 12) from generation 49 achieves a fitness of 404.0 and has 16 transistors, no capacitors, and four resistors (and five resistors of the feedback embryo).

The best circuit (figure 13) from generation 109 has 22 transistors, no capacitors, and 11 resistors (and five resistors of the feedback embryo). It achieves a fitness of 0.178. Its circuit-constructing program tree has 40, 98, and 27 points, respectively, in its first, second, and third result-producing branches and 33 and 144 points, respectively, in its first and second automatically defined functions.

The amplification of an op amp can be measured from the DC sweep (figure 8). The amplification in decibels is the 20 times the common logarithm of the ratio of the change in the output divided by the change of the input (i.e., 20 millivolts here) . The amplification is 60 dB here (i.e., 1,000-to-1 ratio). There is a bias of 0.2 volts. Notice the linearity of the DC sweep in this figure.

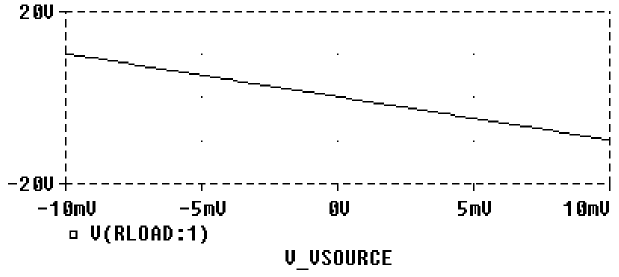

Figure 8 DC sweep of best circuit from generation 109.

Figure 9 shows the time domain behavior of the best circuit from generation 109. The vertical axis is voltage from –10 volts to +10 volts. The input is the 10 millivolt sinusoidal signal; however, this sinusoidal input signal appears as a nearly straight line because of the scale of this figure. At 1,000 Hz, the amplification is 59.7 dB; the bias is 0.18 volts; and the distortion is very low (0.17%).

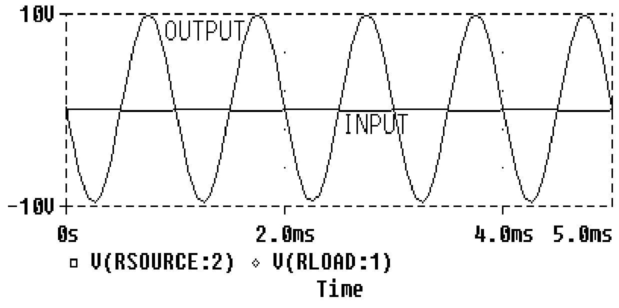

Figure 9 Time domain behavior of best of generation 109.

The amplification of an op amp can also be measured from an AC sweep. Figure 10 shows that the amplification at 1,000 Hz is 59.7 dB. The amplification is within 3 dB of the nominal (flatband) level of 60 dB up to 79, 333 Hz.

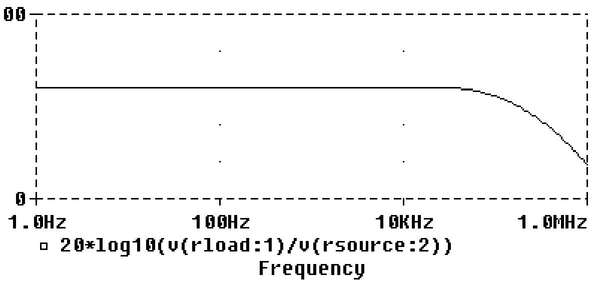

Figure 10 AC sweep for best circuit from generation 109.

In summary, the best circuit of generation 109 achieves 60 dB amplification (based on the DC sweep), has almost no bias or distortion, and generalizes well in the frequency domain.

We then tested whether the genetically evolved 22-transistor best circuit from generation 109 provided more than 60 dB amplification when embedded in a test harness that is appropriate for testing application of up to 80 dB. The amplification is 80.15 dB when SPICE's DC sweep is applied. The amplification is 77.79 dB at 1,000 Hz in the time domain. When the AC sweep is applied, the circuit delivers over 80 dB of amplification from 1 Hz to 36 Hz; the circuit delivers over 77.86 dB from 36 Hz to 55,000 Hz.

6. Conclusion

Genetic programming successfully evolved a 22-transistor amplifier that delivers a gain of 60 dB amplification (based on the DC sweep), that has almost no bias or distortion, and that generalizes well in the frequency domain. Moreover, this genetically evolved 60 dB amplifier generalizes in such a way as to deliver 80 dB of amplification (as measured by the DC sweep) when it is embedded in a test harness that allows 80 dB of amplification.

Acknowledgments

Simon Handley made helpful comments on various drafts of this paper.

References

Andre, David and Koza, John R. 1996. Parallel genetic programming: A scalable implementation using the transputer architecture. In Angeline, P. J. and Kinnear, K. E. Jr. (editors). 1996. *Advances in Genetic Programming 2*. Cambridge: MIT Press.

Angeline, Peter J. and Kinnear, Kenneth E. Jr. (editors). 1996. *Advances in Genetic Programming 2*. Cambridge, MA: The MIT Press.

Gruau, Frederic. 1996. Artificial cellular development in optimization and compilation. In Sanchez, Eduardo and Tomassini, Marco (editors). 1996. *Towards Evolvable Hardware*. Lecture Notes in Computer Science, Volume 1062. Berlin: Springer-Verlag. Pages 48 – 75.

Hemmi, Hitoshi, Mizoguchi, Jun'ichi, and Shimohara, Katsunori. 1994. Development and evolution of hardware behaviors. In Brooks, R. and Maes, P. (editors). *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*. Cambridge, MA: MIT Press. Pages 371–376.

Higuchi, T., Niwa, T., Tanaka, H., Iba, H., de Garis, H. and Furuya, T. 1993. Evolvable hardware – Genetic-based generation of electric circuitry at gate and hardware description language (HDL) levels. Electrotechnical Laboratory technical report 93-4, Tsukuba, Ibaraki, Japan.

Holland, John H. 1975. *Adaptation in Natural and Artificial System*. Ann Arbor, MI: University of Michigan Press.

Kinnear, Kenneth E. Jr. (editor). 1994. *Advances in Genetic Programming*. Cambridge, MA: The MIT Press.

Koza, John R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: MIT Press.

Koza, John R. 1994a. *Genetic Programming II: Automatic Discovery of Reusable Programs.* Cambridge, MA: MIT Press.

Koza, John R. 1994b. *Genetic Programming II Videotape: The Next Generation*. Cambridge, MA: MIT Press.

Koza, John R., Andre, David, Bennett III, Forrest H, and Keane, Martin A. 1996. Use of automatically defined functions and architecture-altering operations in automated circuit synthesis using genetic programming. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: The MIT Press.

Koza, John R., Andre, David, Bennett III, Forrest H, and Keane, Martin A. 1997. *Genetic Programming III.* In preparation.

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996a. Toward evolution of electronic animals using genetic programming. *Artificial Life V: Proceedings of the Fifth International Workshop on the Synthesis and Simulation of Living Systems*. Cambridge, MA: The MIT Press.

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996b. Four problems for which a computer program evolved by genetic programming is competitive with human performance. *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*. IEEE Press. Pages 1–10.

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996c. Automated design of both the topology and sizing of analog electrical circuits using genetic programming. In Gero, John S. and Sudweeks, Fay (editors). *Artificial Intelligence in Design '96*. Dordrecht: Kluwer. 151-170.

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996d. Automated WYWIWYG design of both the topology and component values of analog electrical circuits using genetic programming. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: The MIT Press.

Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: The MIT Press.

Koza, John R., and Rice, James P. 1992. *Genetic Programming: The Movie*. Cambridge, MA: MIT Press.

Kruiskamp, Wim and Leenaerts, Domine. 1995. DARWIN: CMOS opamp synthesis by means of a genetic algorithm. *Proceedings of the 32nd Design Automation Conference*. New York, NY: Association for Computing Machinery. Pages 433–438.

Mizoguchi, Junichi, Hemmi, Hitoshi, and Shimohara, Katsunori. 1994. Production genetic algorithms for automated hardware design through an evolutionary process. *Proceedings of the First IEEE Conference on Evolutionary Computation.* IEEE Press. Vol. I. 661-664.

Quarles, Thomas, Newton, A. R., Pederson, D. O., and Sangiovanni-Vincentelli, A. 1994. *SPICE 3 Version 3F5 User's Manual.* Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA. March 1994.

Rutenbar, R. A. 1993. Analog design automation: Where are we? Where are we going? *Proceedings of the l5th IEEE CICC.* New York: IEEE Press. 13.1.1-13.1.8.

Sanchez, Eduardo and Tomassini, Marco (editors). 1996. *Towards Evolvable Hardware.* Lecture Notes in Computer Science, Volume 1062. Berlin: Springer-Verlag.

Thompson, Adrian. 1996. Silicon evolution. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University.* Cambridge, MA: MIT Press.

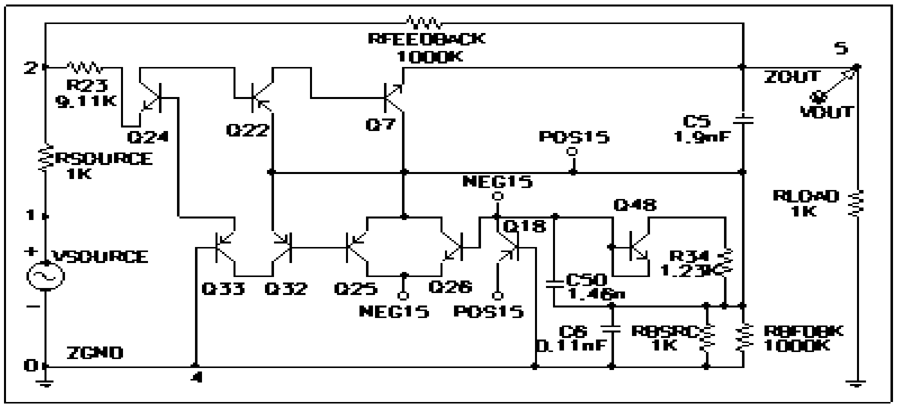

Figure 11 Best circuit from generation 0.

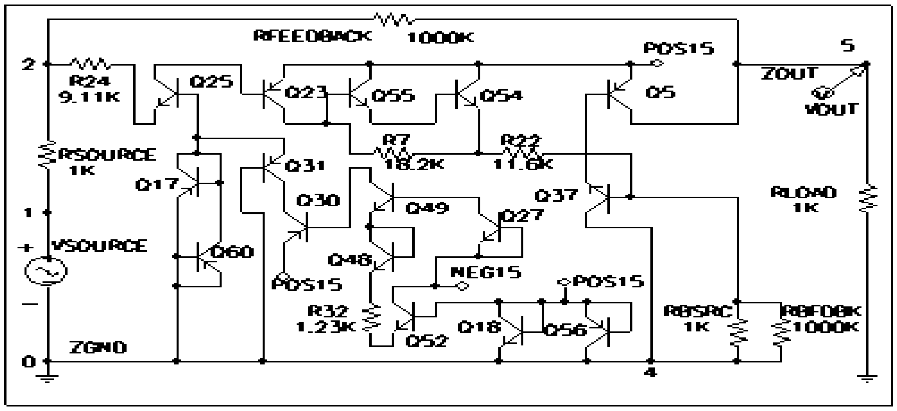

Figure 12 Best circuit from generation 49.

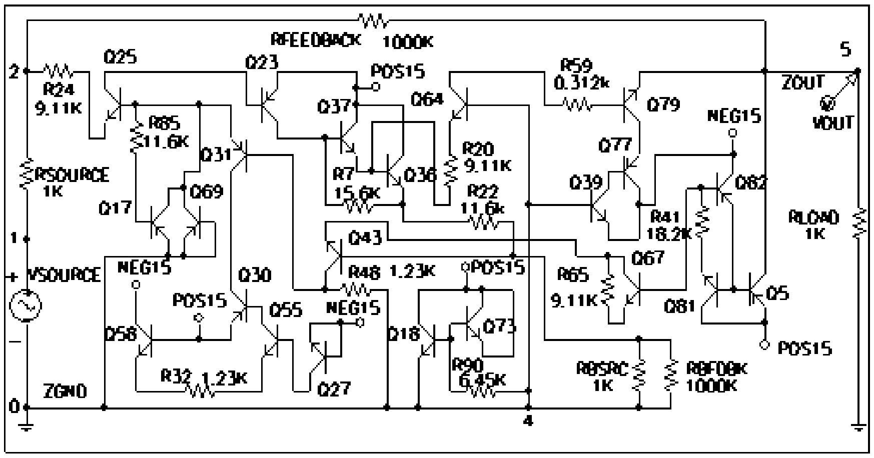

Figure 13 Best circuit from generation 109.