Evolving Sorting Networks using Genetic Programming and Rapidly Reconfigurable Field-Programmable Gate Arrays

John R. Koza
Computer Science Dept.
Stanford University
Stanford, California 94305-9020
koza@cs.stanford.edu
http://www-cs-faculty.stanford.edu/~koza/

Forrest H Bennett III
Visiting Scholar
Computer Science Dept.
Stanford University
Stanford, California 94305
forrest@evolute.com

Jeffrey L. Hutchings
Convergent Design, L.L.C.
3221 E. Hollyhock Hill
Salt Lake City, UT 84121
hutch@Convergent-Design.com

Stephen L. Bade
Convergent Design, L.L.C.
3221 E. Hollyhock Hill
Salt Lake City, UT 84121

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

David Andre
Computer Science Division
University of California
Berkeley, California
dandre@cs.berkeley.edu

ABSTRACT

This paper describes ongoing work involving the use of the Xilinx XC6216 rapidly reconfigurable field-programmable gate array to evolve sorting networks using genetic programming. We successfully evolved a network for sorting seven items that employs two fewer steps than the sorting network described in a 1962 patent and that has the same number of steps as the seven-sorter devised by Floyd and Knuth subsequent to the patent.

1. Introduction

Genetic programming is an extension of John Holland's genetic algorithm (1975). In genetic programming, the population consists of computer programs of varying sizes and shapes (Koza 1992, 1994a, 1994b; Koza and Rice 1992). Recent work on genetic programming is described in Kinnear (1994), Angeline and Kinnear (1996), Koza, Goldberg, Fogel, and Riolo (1996), and Koza et al. (1997).

The dominant component of the computational burden of solving a problem with the genetic algorithm or genetic programming is the task of evaluating the fitness for each of the thousands of individuals in the evolving population for each of the hundreds of generations in the run. Other tasks, such as the creation of the initial population and the execution of the genetic operations (e.g., Darwinian reproduction, crossover, and mutation) are relatively fast.

Rapidly reconfigurable computing devices (Sanchez and Tomassini 1996 and Higuchi 1997) open the possiblity of greatly accelerating the fitness evaluation task of genetic algorithms by translating each individual of the evolving population *into hardware* and then exploiting the parallelism of the hardware for the fitness evaluation task.

Section 2 describes the minimal sorting network problem. Section 3 describes rapidly reconfigurable field-programmable gate arrays and the Xilinx XC6216 chip. Section 4 outlines the preparatory steps for applying genetic programming to the problem of evolving a sorting network. Section 5 outlines the mapping of the fitness evaluation task for sorting networks onto the chip.

2. Minimal Sorting Networks

A *sorting network* is an algorithm for sorting items consisting of a sequence of comparison-exchange operations that are executed in a fixed order. Figure 1 shows a sorting network for four items.

Figure 1 Minimal sorting network for 4 items.

The to-be-sorted items, A_1, A_2, A_3, A_4, start at the left on the horizontal lines. A vertical line connecting horizontal line *i* and *j* indicates that items *i* and *j* are to be compared and exchanged, if necessary, so that the larger of the two is on the bottom. In this figure, the first step causes A_1 and A_2 to be exchanged if $A_2 < A_1$. This step and the next three steps cause the largest and smallest items to be routed down and up, respectively. The fifth step ensures that the remaining two items end up in the correct order. The correctly sorted output appears at the right. A five-step network is known to be minimal for four items.

Even though sorting networks are oblivious to their inputs in the sense that they always perform the same fixed sequence of comparison-exchange operations, they are of considerable practical importance because they are more efficient for sorting small numbers of items than the well-known non-oblivious sorting algorithms such as Quicksort. Thus, there is considerable interest in sorting networks with a minimum number of comparison-exchange operations.

As the number of items to be sorted increases, construction of a minimal sorting network becomes increasingly difficult. There has been a lively search over the years for smaller sorting networks (Knuth 1973). In U. S. patent 3,029,413, O'Connor and Nelson (1962) described sorting networks for 4, 5, 6, 7, and 8 items using 5, 9, 12, 18, and 19 comparison-exchange operations, respectively.

During the l960s, Floyd and Knuth devised a 16-step seven-sorter and proved it to be the minimal seven-sorter. They also proved that the four other sorting networks in the 1962 O'Connor and Nelson patent were minimal.

The 16-sorter has received considerable attention. In 1962, Bose and Nelson devised a 65-step sorting network for 16 items. In 1964, Batcher and Knuth presented a 63-step 16-sorter. In l969, Shapiro discovered a 62-step 16-sorter and, in the same year, Green discovered one with 60 steps.

Hillis (1990, 1992) used the genetic algorithm to evolve 16-sorters with 65 and 61 steps – the latter using co-evolution of a population of sorting networks competing with a population of fitness cases. In this work, the first 32 steps of Green's 60-step 16-sorter was incorporated as a fixed beginning for all sorters.

Juille (1995) used an evolutionary algorithm to evolve a 13-sorter with 45 steps thereby improving on the 13-sorter with 46 steps presented in Knuth (1973). Juille (1997) has also evolved networks for sorting 14, 15, and 16 items having the same number of steps (i.e., 51, 56, and 60, respectively) as reported in Knuth (1973).

Verification of the validity of a network (through analysis, instead of exhaustive enumeration) grows in difficulty as the number of items to be sorted increases. A sorting network can be exhaustively tested for validity by testing all $n!$ permutations of n distinct numbers. However, thanks to the "zero-one principle" (Knuth 1973, page 224), if a sorting network for n items correctly sorts n bits into non-decreasing order (i.e., all the 0's ahead of all the 1's) for all 2^n sequences of n bits, it necessarily will correctly sort any set of n distinct numbers into non-decreasing order. Thus, it is sufficient to test a putative 16-sorter against only $2^{16} = 65{,}536$ combinations of binary inputs, instead of all $16! \sim 2 \infty 10^{13}$ inputs. Nonetheless, in spite of this "zero-one principle," testing a putative 16-sorter consisting of around 60 steps on 65,536 different 16-bit input vectors is a formidable amount of computation when it appears in the inner loop of a genetic algorithm.

3. Field-Programmable Gate Arrays and the Xilinx XC6216

A *field-programmable gate array* (FPGA) is a type of digital chip that contains a regular two-dimensional array of thousands of logical function units and a regular network of interconnection lines for connecting the function units in which both the functionality of each logical function unit and the connectivity between the logical function units can be programmed by the user in the field (rather than at the chip fabrication factory) (Trimberger 1994).

FPGAs were commercially introduced in the mid 1980s by Xilinx. They are primarily used to facilitate rapid prototyping of new electronic products – particularly those for which time-to-market or low product volume precludes the fabrication of a custom application-specific integrated circuit (ASIC). Thus, when first marketed, many new electronic products contain an FPGA. If the product's sales volume is sufficient, an ASIC may be used in later versions of the product after the design has stabilized.

Ignoring the *anti-fuse* type of FPGA that is irreversibly programmed by the user in the field by the one-time application of a high voltage, we focus on the *infinitely reprogrammable* type of FPGA where the functionality of each function unit and the connectivity between the function units is stored in memory within the device, such as static random access memory (SRAM).

Engineers working with FPGAs typically employ a computer-aided design (CAD) tool to design and optimize their circuits. First, the engineer conceives the design (which often uses subcircuits from a library). Second, the engineer's design of the desired circuit is captured by the CAD tool in the form of Boolean expressions, a schematic diagram, or a general-purpose high-level description language such as VLSI Hardware Description Language (VHDL). Third, a *technology mapping* converts the description of the circuit into logical function blocks of the particular type that are present on the particular FPGA chip that is to be used. Fourth, the logical function blocks are *placed* into particular locations on the FPGA. Fifth, a *routing* using the FPGA's limited interconnection resources is created between the logical function units on the chip. Sixth, hundreds of thousands of *configuration bits* (which, for almost all commercially available FPGAs, are both highly complex and confidential) are created. Seventh, the configuration bits are downloaded into the FPGA's memory.

An engineer may spend several days in designing a circuit and perhaps an hour in entering the design into the CAD tool (the first two steps above). The CAD tool may require hours (or, at best, many minutes) to compile a single design (the next four steps above). The downloading of the configuration bits for a single design into memory may take a half a second. For almost all FPGAs, all the configuration bits must be reloaded if even one bit changes.

Note that all of the above times compare very favorably with the weeks or months that may be required to produce an ASIC for a single design (i.e., a time advantage of around two orders of magnitude).

Once an FPGA is configured, its thousands of logical function units operate in parallel at the chip's clock rate.

Since the major component of the computational burden of executing a genetic algorithm is the fitness evaluation task, the massive parallellism of FPGAs raises the possiblity that an FPGA could be used to accelerate the fitness evaluation task. However, this alluring possiblity is precluded as a practical possiblity for almost all commercially available FPGAs for several reasons. First, the technology mapping, placement, and routing tasks required to map each individual of each generation of the evolving population onto the chip is complex and consumes

so much time (typically hours or, at best, minutes) as to preclude practical use of an FPGA in the inner loop of an genetic algorithm. Second, the serial downloading of the configuration bits *alone* consumes so much time (typically about half a second) as to preclude practical use of an FPGA in the inner loop of a genetic algorithm. Third, the encoding scheme for the configuration bits are confidential.

As will be seen below, the new Xilinx XC6200 series of FPGAs minimizes or eliminates the above obstacles to the rapid reconfigurability required for a practical run of a genetic algorithm. If a problem can be successfully mapped onto this type of FPGA, reconfigurability can be accelerated by about 6 orders of magnitude – enough to make it practical for the inner loop of a genetic algorithm.

The Xilinx XC6216 chip contains a 64 ∞ 64 two-dimensional array of identical cells. Each cell is capable of performing any two-argument Boolean function (as well as many useful three-argument Boolean functions) and contains a flip-flop for storing one bit of information (Xilinx 1997). The functionality of each of these 4,096 cells is controlled by 24 configuration bits whose meaning is both straightforward and public.

Each cell can directly receive inputs from its four neighbors (as well as certain more distant cells). The interconnection between cells is controlled by additional configuration bits (also straightforward and public). Finally, additional configuration bits are used to establish interconnections between cells and 256 input-output units located on the periphery of the chip.

Unlike other FPGAs, the 6200 can be randomly accessed, and the memory containing the configuration bits is directly memory-mapped onto the address space of the host processor. Thus, it is possible to change single bits.

Most important, the Xilinx XC6216 FPGA is designed so that no combination of configuration bits for function cells can cause internal contention (i.e., conflicting '1' and '0' signals simultaneously driving a destination) and potential damage of the chip. Specifically, it is not possible for two or more signal sources to ever simultaneously drive a routing line or input node of a function cell. This is accomplished by obtaining the driving signal for each routing line and each input node from a multiplexer. Thus, only a single driving signal can be selected regardless of the choice of configuration bits. In contrast, in most other FPGAs, the driving signal is selected by multiple independently programmable interface points (pips). (Care must still be taken with the configuration bits that control the chip's I/O cells because an outside signal connected to one of the chip's input pins can potentially contend with a signal generated on the chip).

A PC board containing the XC6216 chip with a PCI interface and SRAM and supporting tools is available from Virtual Computer Corporation (`www.vcc.com`).

Thompson (1996) used a genetic algorithm to evolve a frequency discriminator on a Xilinx XC6216 reconfigurable digital gate array operating in analog mode.

4. Preparatory Steps

Before applying genetic programming to a problem, the user must perform six major preparatory steps, namely (1) identifying the terminals, (2) identifying the primitive functions, (3) creating the fitness measure, (4) choosing control parameters, (5) setting the termination criterion and method of result designation, and (6) determining the architecture of the program trees in the population.

For the problem of evolving a sorting network for 16 items, the terminal set, $\mathcal{T}$, is

$\mathcal{T}$ = {`D1`, ..., `D16`, `NOOP`}.

Here `NOOP` is the zero-argument "No Operation" function.

The function set, $\mathcal{F}$, is

$\mathcal{F}$ = {`COMPARE-EXCHANGE`, `PROGN2`, `PROGN3`, `PROGN4`}.

Note that none of these functions have return values.

Each individual in the population consists of a constrained syntactic structure composed of primitive functions from the function set, $\mathcal{F}$, and terminals from the terminal set, $\mathcal{T}$ such that the root of each program tree is a `PROGN2`, `PROGN3`, or `PROGN4`; each argument to `PROGN2`, `PROGN3`, and `PROGN4` must be a `NOOP` or a function from $\mathcal{F}$; and both arguments to every `COMPARE-EXCHANGE` function must be from $\mathcal{T}$ (but not `NOOP`)

The `PROGN2`, `PROGN3`, and `PROGN4` functions respectively evaluate each of their two, three, or four arguments sequentially.

The two-argument `COMPARE-EXCHANGE` function side-effects the current state of the vector of to-be-sorted bits. The result of executing a `(COMPARE-EXCHANGE i j)` is that the bit currently in position i of the vector is compared with the bit currently in position j of the vector. If the first bit is greater than the second bit, the two bits are exchanged. That is, the effect of executing a `(COMPARE-EXCHANGE i j)` is that the two bits are sorted into non-decreasing order. Table 1 shows the two results R_i and R_j produced by executing a `(COMPARE-EXCHANGE i j)`. Note that column R_i is the Boolean AND function and column R_j is the Boolean OR function.

Table 1 The `COMPARE-EXCHANGE` function.

Two Arguments		Two Results	
Ai	Aj	Ri	Rj
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

The fitness of each individual program in the population is based on the correctness of its sorting of 2^{16} = 65,536 fitness cases consisting of all possible vectors of 16 bits. If, after an individual program is executed on a particular fitness case, all the 1's appear after all the 0's), the program is deemed to have correctly sorted that particular fitness case.

Because our goal is to evolve small (and preferably minimal) sorting networks, we ignore exchanges where $i = j$

and exchanges that are identical to the previous exchange. Moreover, during the depth-first execution of a program tree, only the first C_{max} = 65 COMPARE-EXCHANGE functions (i.e., five more steps than in Green's 60-step 16-sorter) in a program are actually executed (thereby relegating the remainder of the program to be unused code).

Hits are defined as the number of fitness cases for which the sort is performed correctly.

The fitness measure for this problem is multi-objective in that it involves both the correctness and size of the sorting network. Standardized fitness is defined in a lexical fashion to be the number of fitness cases (0 to $16 \infty 2^{16}$) for which the sort is performed incorrectly plus 0.01 times the number (1 to C_{max}) of COMPARE-EXCHANGE functions that are actually executed. For example, the fitness of a 16-sorter with 60 COMPARE-EXCHANGE functions (such as Green's) is 0.60 while the fitness of an imperfect network with 60 COMPARE-EXCHANGE functions that correctly handles all but 12 fitness cases (out of $16 \infty 2^{16}$) is 12.60. Note that we used tournament selection.

The population size was 1,000. The percentage of genetic operations on each generation was 89% one-offspring crossovers, 10% reproductions, and 1% mutations. The maximum size, H_{rpb}, for the result-producing branch was 300 points. The other parameters for controlling the runs were the default values specified in Koza 1994a (appendix D). The architecture of the overall program consisted of one result-producing branch.

5. Mapping the Fitness Evaluation Task onto the Xilinx XC6216 Chip

The problem of evolving sorting networks was run on a host PC Pentium type computer containing a Virtual Computer Corporation "HOT Works" PCI board containing a Xilinx XC6216 field-programmable gate array. This combination permits the field-programmable gate array to be advantageously used for the computationally burdensome fitness evaluation task while permitting the general-purpose host computer to perform all the other tasks.

In this arrangement, the host PC begins the run by creating the initial random population (with the XC6216 waiting). Then, for generation 0 (and each succeeding generation), the PC creates the necessary configuration bits to enable the XC6216 to measure the fitness of the first individual program in the population (with the XC6216 waiting). Thereafter, the XC6216 measures the fitness of one individual. Note that the PC can simultaneously prepare the configuration bits for the next individual in the population and then polling to see if the XC6216 is finished). After the fitness of all individuals in the current generation of the population is measured, the genetic operations (reproduction, crossover, and mutation) are performed (with the XC6216 waiting). This arrangement is beneficial because the computational burden of creating the initial random population and of performing the genetic operations is small in comparison with the fitness evaluation task.

The clock rate at which a field-programmable gate array can be run on a problem is considerably slower than that of a contemporary serial microprocessor (e.g., Pentium or PowerPC) that might run a software version of the same problem. Thus, in order to advantageously use the Xilinx XC6216 field-programmable gate array for the computationally burdensome fitness evaluation task, it is necessary to find a mapping of the fitness evaluation task onto the XC6216 that exploits at least some of the massive parallelism of the 4,096 function cells of the XC6216.

Figure 2 shows our placement on 32 horizontal rows and 64 vertical columns of the XC6216 chip of eight major computational elements (labeled A through H). The figure does not show that a second such $32 \infty 64$ area operates in parallel on the chip. The figure also does not show the ring of input-output blocks (OIBs) that surround the $64 \infty 64$ area of function cells or the physical input-output pins that connect the chip to the outside.

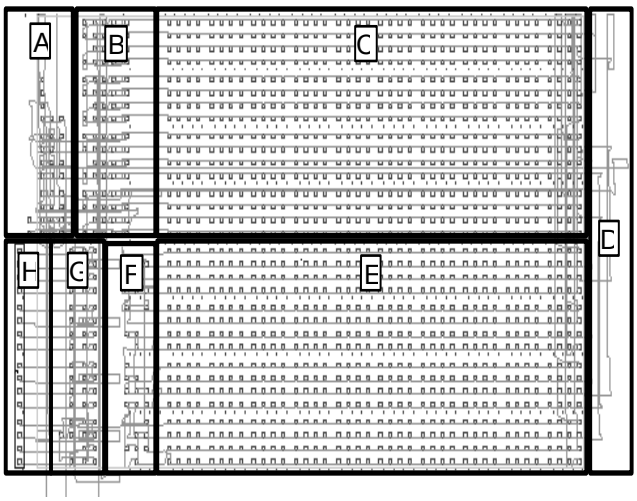

Figure 2 Arrangement of major elements A through H on a $32 \infty 64$ portion of the Xilinx XC6216 chip.

For a k-sorter ($k \leq 16$), a 16-bit counter B (near the upper left corner of the chip) counts down from 2^k - 2 to 0 under control of control logic A (upper left corner). The vector of k bits resident in counter B on a given time step represents one fitness case of the sorting network problem. The vector of bits from counter B is fed into the first (leftmost) $16 \infty 1$ vertical column of cells of the large $16 \infty 40$ area C. Each $16 \infty 1$ vertical column of cells in C (and each cell in similar area E) corresponds to one COMPARE-EXCHANGE operation of an individual candidate sorting network. The vector of 16 bits produced by the 40th (rightmost) sorting step of area C then proceeds to U-turn area D and is channeled into the first (rightmost) column of the large $16 \infty 40$ area E. The final output from area E is checked by answer logic F for whether the individual candidate sorting network has correctly rearranged the original incoming vector of bits so that all the 0s are above all the 1s. The 16-bit accumulator G is incremented by one if the bits are correctly sorted. Note that the 16 bits of accumulator G are sufficient for tallying the number of

correctly sorted fitness cases because the host computer starts counter B at 2^k - 2, thereby skipping the uninteresting fitness case of consisting of all 1s (which cannot be incorrectly sorted by any network). The final value of raw fitness is reported in 16-bit register H after all the 2^k - 2 fitness cases have been processed.

The logical function units and interconnection resources of areas A, B, D, F, G, and H are permanently configured to handle the sorting network problem for $k \leq 16$.

The two large areas, C and E, together represent the individual candidate sorting network. The configuration of the logical function units and interconnection resources of the 1,280 cells in areas C and E become personalized to the current individual candidate sorting network.

For area C, each cell in a 16 ∞ 1 vertical column is configured in one of three main ways. First, the logical function unit of exactly one of the 16 cells is configured as a two-argument Boolean AND function (corresponding to result R_i of table 1). Second, the logical function unit of exactly one other cell is configured as a two-argument Boolean OR function (corresponding to result R_j of table 1). Bits i and j become sorted into the correct order by virtue of the fact that the single AND cell in each 16 ∞ 1 vertical column always appears above the single OR cell. Third, the logical function units of 14 of the 16 cells are configured as "pass through" cells that horizontally pass their input from one vertical column to the next.

For area E, each cell in a 16 ∞ 1 vertical column is configured in one of three similar main ways.

There are four subtypes each of AND and OR cells and four types of "pass through" cells. Half of these subtypes are required because all the cells in area E differ in chirality (handedness) from those in area C in that they receive their input from their right and deliver output to their left.

If the sorting network has fewer than 80 COMPARE–EXCHANGE operations, the last few vertical columns of area E contain 16 "pass through" cells. Note that the genetic operations are constrained so as to not produce networks with more than 80 steps.

Within each cell of areas C and E, the one-bit output of the cell's logical function unit is stored into a flip-flop. The contents of the 16 flip-flops in one vertical column become the inputs to the next vertical column on the next time step.

The overall arrangement operates as an 87-stage pipeline (the 80 stages of areas C and E, the three stages of answer logic F, and four stages of padding at both ends of C and E).

Figure 3 shows the bottom six cells of an illustrative vertical column from area C whose purpose is to implement a (COMPARE–EXCHANGE 2 5) operation. As can be seen, cell 2 (second from top of the figure) is configured as a two-argument Boolean AND function (*) and cell 5 is configured as a two-argument OR function (+). All the remaining 14 cells of the vertical column (of which only four are shown in this abbreviated figure) are "pass through" cells. These "pass through" cells horizontally convey the bit in the previous vertical column to the next vertical column. In addition, each "pass through" cell (3 and 4) that lies between the AND and OR cells (1 and 5) is configured so that it conveys one signal vertically upwards and one signal vertically downwards as "fly over" signals. These "fly overs" of the two intervening "pass through" cells (3 and 4) enable cell 2's input to be shared with cell 5 and cell 5's input to be shared with cell 2. Specifically, the input coming into cell 2 horizontally from the previous vertical column (i.e., from the left in the figure) is bifurcated so that it feeds both the two-argument AND in cell 2 and the two-argument OR in cell 5 (and similarly for the input coming into cell 5).

Notice that when a 1 is received from the previous vertical column on horizontal row 2 and a 0 is received on horizontal row 5 (i.e., the two bits are out of order), the AND of cell 2 and the OR of cell 5 cause a 0 to be emitted as output on horizontal row 2 and a 1 to be emitted as output on horizontal row 5 (i.e., the two bits have become sorted into the correct order).

The remaining "pass through" cells (i.e., cells 1 and 6 in the figure and cells 7 through 16 in the full 1 ¥ 16 vertical column) are of a subtype that does not have the "fly over" capability of the two "intervening" cells (3 and 4). The design of this subtype prevents possible contention with the unpredictable signals available from the input-output blocks (IOBs) that surround the main 64 ¥ 64 area of the chip. All AND and OR cells are similarly designed since they necessarily sometimes occur at the top or bottom of a vertical column.

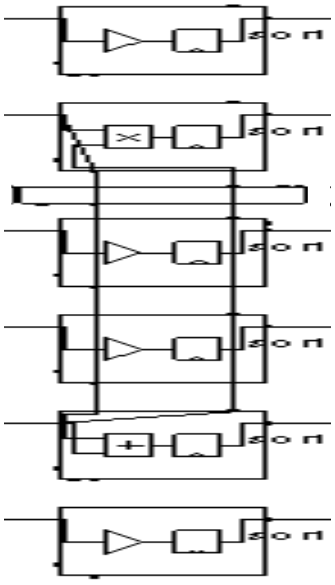

Figure 3 Implementation of (COMPARE–EXCHANGE 2 5).

Note that the intervening "pass through" cells (cells 3 and 4 in the figure) invert their "fly over" signals. Thus, if there is an odd number of "pass through" cells intervening vertically between the AND cells and OR cells, the signals being conveyed upwards and downwards in a vertical column will arrive at their destinations in inverted form.

Accordingly, special subtypes of the AND cells and OR cells reinvert (and thereby correct) such arriving signals.

When the XC6216 begins operation for a particular individual sorting network, all the 16 ∞ 80 flip-flops in C and E (as well as the flip-flops in three-stage answer logic F, the four insulative stages, and the "done bit" flip-flop) are initialized to zero. Thus, the first 87 output vectors received by the answer logic F each consist of 16 0's. Since the answer logic F treats a vector of 16 0's as incorrect, accumulator G is not incremented for these first 87 vectors.

A "past zero" flip-flop is set when counter B counts down to 0. As B continues counting, it rolls over to $2^{16} - 1$, and continues counting down. When counter B reaches $2^{16} - 87$ (with the "past zero" flip-flop being set), control logic A stops further incrementation of accumulator G. The raw fitness from G appears in reporting register H and the "done bit" flip-flop is set to 1. The host computer polls this "done bit" to determine that the XC6216 has completed its fitness evaluation task for the current individual.

The flip-flop toggle rate of the chip (220 MHz) provides an upper bound on the speed at which a field-programmable gate array can be run. In practice, the speed at which an FPGA can be run is determined by the longest routing delay. The FPGA can be run at 20 MHz for the current unoptimized version of the design.

The above approach exploits the massive parallelism of the XC6216 chip in five ways. First, the Boolean AND functions and OR functions of each COMPARE-EXCHANGE operation are performed in parallel (in the vertical columns of areas C and E). Second, numerous operations are performed in parallel in counter B, accumulator G, control logic A, and especially in answer logic F. The FPGA relieves the host computer of performing these operations in software in serial. Third, most importantly, the 87-step pipeline (80 steps for areas C and E and 7 steps for areas F and G) enables 87 fitness cases to be processed in parallel in the pipeline. Fourth, there are two separate 32 ∞ 64 areas operating in parallel on the chip. Fifth, the XC6216 evaluates the 2^k fitness cases independently of the activity of the host PC Pentium type computer (which simultaneously can prepare the next individual for the XC6216).

6. Results

A 16-step 7-sorter was evolved that has two fewer steps than the sorting network described in O'Connor and Nelsons' patent (1962) and that has the same number of steps as the 7-sorter that was devised by Floyd and Knuth subsequent to the patent and described in Knuth 1973.

Acknowledgments

Phillip Freidin of Silicon Spice provided invaluable information concerning FPGAs and helpful comments on this paper. Stefan Ludwig of DEC and Steve Casselman and John Schewel of Virtual Computer Corporation provided helpful assistance concerning operation of the XC6216. Simon Handley made helpful comments on this paper.

References

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

Higuchi, Tetsuya (editor). 1997. *Proceedings of International Conference on Evolvable Systems: From Biology to Hardware (ICES-96)*. Lecture Notes in Computer Science. Volume --- Berlin: Springer-Verlag.

Hillis, W. Daniel. 1990. Co-evolving parasites improve simulated evolution as an optimization procedure. In Forrest, Stephanie (editor). *Emergent Computation: Self-Organizing, Collective, and Cooperative Computing Networks*. Cambridge, MA: The MIT Press.

Hillis, W. Daniel. 1992. Co-evolving parasites improve simulated evolution as an optimization procedure. In Langton, Christopher, Taylor, Charles, Farmer, J. Doyne, and Rasmussen, Steen (editors). *Artificial Life II, SFI Studies in the Sciences of Complexity*. Volume X. Redwood City, CA: Addison-Wesley. Pages 313-324.

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

Juille, Hugues. 1995. Evolution of non-deterministic incremental algorithms as a new approach for search in state spaces. In Eshelman, L. J. (editor). *Proceedings of the Sixth International Conference on Genetic Algorithms*. San Francisco, CA: Morgan Kaufmann. 351 – 358.

Juille, Hugues. 1997. Personal communication.

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

Knuth, Donald E. 1973. *The Art of Computer Programming*. Volume 3. Reading, MA: Addison-Wesley.

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., Deb, Kalyanmoy, Dorigo, Marco, Fogel, David B., Garzon, Max, Iba, Hitoshi, and Riolo, Rick L. (editors). 1997. *Genetic Programming 1997: Proceedings of the Second Annual Conference, July 13–16, 1997, Stanford University*. San Francisco, CA: Morgan Kaufmann.

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.

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

O'Connor, Daniel G. and Nelson, Raymond J. 1962. *Sorting System with N-Line Sorting Switch*. United States Patent number 3,029,413. Issued April 10, 1962.

Sanchez, Eduardo and Tomassini, Marco (editors). 1996. *Towards Evolvable Hardware*. Lecture Notes in Computer Science, Volume 1062. Berlin: Springer-Verlag. 76 – 98.

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.

Trimberger, Stephen M. (Editor) 1994. *Field Programmable Gate Array Technology*. Boston, MA: Kluwer.

Xilinx. 1997. *XC6000 Field Programmable Gate Arrays: Advance Product Information*. January 9, 1997. Vers. 1.8.