Journal of Circuits, Systems, and Computers
Vol. 14, No. 2 (2005) 347–366

World Scientific
www.worldscientific.com

AN EVOLUTIONARY ALGORITHM FOR THE ALLOCATION PROBLEM IN HIGH-LEVEL SYNTHESIS

HAIDAR M. HARMANANI
Department of Computer Science,
Lebanese American University,
Byblos, 1401 2010, Lebanon

RONY SALIBA
ABEO, Ideal Communication,
P. O. Box 11-6876, Beirut, Lebanon

Received 10 August 2003
Revised 9 July 2004

This paper presents an evolutionary algorithm to solve the datapath allocation problem in high-level synthesis. The method performs allocation of functional units, registers, and multiplexers in addition to controller synthesis with the objective of minimizing the cost of hardware resources. The system handles multicycle functional units as well as structural pipelining. The proposed method was implemented using C++ on a Linux workstation. We tested our method on a set of high-level synthesis benchmarks, all yielding good solutions in a short time. An integration path to Field Programmable Gate Arrays (FPGAs) is provided through VHDL.

Keywords: High-level synthesis; datapath allocation; genetic algorithms.

1. Introduction

The enormous progress in VLSI and CAD technology to support automated high-level synthesis has helped to shorten the time to market digital integrated circuits. *High-level synthesis* is the process of transforming a behavioral description into a structural one. From the input specification, the synthesis system produces a description of a datapath, that is, a network of registers, functional units, multiplexers and buses. The synthesis must also produce the specification of the control path. There are many different structures that can be used to realize a given behavior. One of the main tasks of high-level synthesis is to find the structure that best meets the constraints while minimizing other costs. For example, the goal might be to minimize area while achieving a certain required processing rate.[1]

The system to be designed is usually represented at the algorithmic level by a hardware description language such as Verilog or VHDL. The algorithmic

description is parsed and represented by a Data Flow Graph (DFG) that preserves data and control flow information. The nodes in such graphs represent the operations that are performed on the data (edges) coming into them. Arcs or edges leaving the nodes correspond to the results produced by the operations represented by the nodes. High-level synthesis involves two NP-complete optimization problems.[2] The first is a scheduling problem that involves assigning the operators in the DFG to control steps that represent the clock cycles in the final design. The second is an allocation problem which is concerned with assigning operations and values to hardware so as to minimize the amount of hardware needed. Thus, registers are allocated for variables, operations are assigned to functional units (FUs), and connections which are multiplexers, buses, or a combination of both, are established between them. The allocation phase is constrained by the schedule that it implements.

There has been during the last decade a growing interest in algorithms that are based on the principal of evolution. A common term refers to such techniques as *evolutionary computation*[3]; *genetic algorithms* and *evolutionary programming* are among the best known approaches within this class. An important advantage of evolutionary computation is that they only need an evaluable objective function. They do not need special pre-knowledge about the problem space. Furthermore, they are global in scope, and can handle nonlinear, discrete, continuous or mixed search spaces, thus making them especially suitable for tackling difficult and complex optimization problems. Though they do not always find the optimum solution, they usually find a *good approximation*, surprisingly quickly.

This paper presents an *evolutionary algorithm* to solve the datapath allocation problem, described as follows:

> Given a behavioral description of a digital circuit and a set of design constraints, use an *evolutionary approach* in order to generate a datapath and a controller pair that: (1) implements the original behavior and (2) meets the initial design constraints.

The work is motivated by the following issues:

- Rapid exploration of the complex design space using an *evolutionary computation.*
- A global optimization method that is based on a realistic technology library.
- Bridging the gap between behavioral and logic synthesis. This is accomplished using VHDL as an interface vehicle and targeting a FPGA family for rapid prototyping.

The rest of the paper is organized as follows. Section 2 describes our *evolutionary datapath synthesis* along with our chromosomal representation, the genetic operators, and the cost function. Section 3 discusses the bridge to lower level synthesis. Experimental results are presented and discussed in Sec. 4. Finally, we conclude with remarks in Sec. 5.

1.1. *Related work*

There has been various deterministic approaches to solve the datapath synthesis problem.[4–9] However, fewer probabilistic methods were reported. For example, Devadas[10] used a simulated annealing approach to solve the simultaneous cost/resource constrained allocation of functional units, registers, and interconnects by improving one solution at a time. In fact, a simulated annealing approach is similar to a genetic algorithm approach with a population of size *one*; however, genetic algorithms perform a multidimensional search by maintaining a population of potential solutions. Wehn[11] proposed a scheduling and allocation approach based on genetic algorithms. The system formulates the scheduling/allocation problem as a quadratic assignment problem and applies GA to find an assignment based on a given cost function. The system uses a special crossover operator in order to avoid illegal solutions.

Ly[12] used a simulated evolution approach where scheduling and allocation are performed independently. The approach is based on an analogy with the process of natural evolution but it is different from GA. The method explores the design space by repeatedly ripping parts of a design in a probabilistic manner and then reconstructing these parts using application specific heuristics. Martin[13] used genetic algorithms in order to solve the scheduling problem. Dhodhi[14] proposed a problem space genetic algorithm for datapath synthesis where the problem is altered using a genetic algorithm and then transformed into solution space by means of a heuristic thus avoiding unfeasible solutions. Blickle[15] used an evolutionary approach for system-level synthesis in order to optimally map a task-level specification onto a heterogeneous hardware/software architecture. Recently, Mandal[16] proposed a technique for datapath allocation using a genetic algorithm. The approach is based on force directed datapath binding.

2. Evolutionary Datapath Synthesis

The allocation method starts with a scheduled data flow description of a circuit. The scheduling can be accomplished using any method; however, we use the method reported by Paulin *et al.*[7] In what follows, we describe our evolutionary algorithm that solves the datapath allocation problem with reference to the scheduled data flow graph of the *simple biquad filter*, shown in Fig. 1.

2.1. *Problem formulation*

Given a scheduled data flow graph (DFG), a node corresponds to (1) an operator that must be assigned to a functional unit during the clock cycle in which it is scheduled and (2) a value that must be assigned to a register for the duration of its life time.[a] In order to optimize the datapath cost, functional units

[a]The life span of a variable is the interval between the time the variable was first introduced and last used.

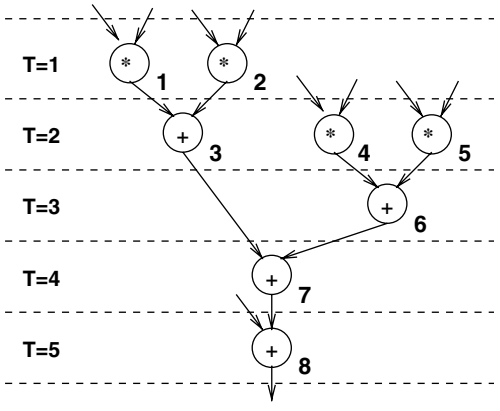


Fig. 1. Simple biquad DFG.

can be reused by different operations at disjoint clock cycles while variables with nonoverlapping life time can share the same register. Thus, the optimization problem has multiple objectives; i.e., to minimize functional units, registers, and the number of buses or multiplexers. We consider here design optimization as the combined minimization of area and maximization of performance. Optimization is motivated not only by the desire of maximizing the circuit quality, but also by the fact that synthesis without optimization would yield noncompetitive circuits at all.[17]

The basic idea of our algorithm is to merge data flow variables, operations and connections variables *simultaneously* in a single merging process using the genetic operators defined shortly. The motivation is as follows. In most of the existing allocation techniques, there is a lack of merging coordination, i.e., merging of variables, operations and connections are performed in separate. However, the merging order may adversely affect the cost — doing register merging first may increase the overhead of operation merging, and vice versa.

The specific objective of our scheme is to allow the mapping of DFG nodes into genes that are the building blocks of our datapath. Genes are merged in order to explore the reuse of registers and functional units while minimizing the number of multiplexers inputs. There are two conditions that should be satisfied for the successful merging of two DFG nodes:

(1) There is no conflict in the use of the two operators, that is, they are assigned to different clock cycles in the schedule.
(2) The merger of the nodes results in a module that exists in the library.

Formally, two nodes that can be merged under the above conditions are called *compatible*.

2.2. *Chromosomal representation*

A chromosome in the population encodes all the information that needs to be optimized. Each chromosome represents a candidate datapath solution that implements the original DFG. The chromosomal representation consists of a vector whose length is equal to the number of nodes in the DFG.

In order to perform the encoding, we number the DFG nodes one level at a time, top–down and left to right as shown in Fig. 1. The numbering ensures that different hardware instances for the same resource are numbered differently. Next, each DFG node is encoded as a gene in the chromosome. Thus, a gene or, to be precise, its position in the chromosome represents (Fig. 2):

- The actual hardware resource (ALU or a multiplier).
- The registers that are at the input and output ports of the hardware resource.
- The connectivity data that is stored with each gene so that the chromosome is an exact representation of the DFG.
- The DFG primary inputs and outputs are stored in a special look-up table.

Figure 3 illustrates the correspondence between a gene and datapath hardware resources while Fig. 4 shows the datapath after the merger of two nodes, *node 3* and *node 6*. Whenever two nodes are merged and assigned to the same hardware resource, the index of the genes are updated as follows:

- The functional unit fields in the corresponding genes are assigned the same index. For example, in Fig. 4, *gene 3* and *gene 6* have both the number 3 in the functional unit field indicating that both genes share the same adder.
- If the life spans of the output registers that correspond to the merged nodes do not overlap, then the corresponding register indexes are assigned the same index as well. Otherwise, they maintain their original numbering. In Fig. 4, the

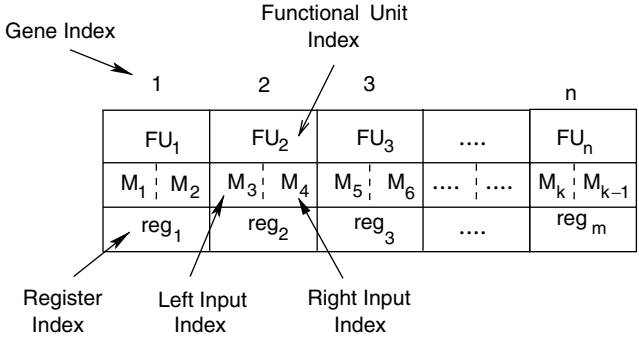


Fig. 2. General chromosome representation.

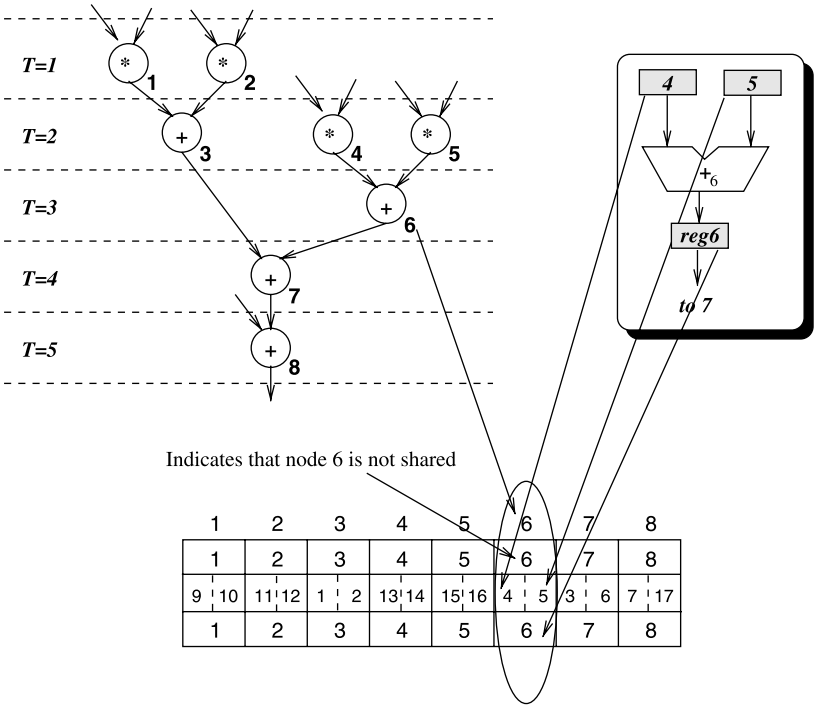


Fig. 3. Sample chromosome mapping. The shown chromosome corresponds to a datapath where there is no resource sharing. Thus, each gene (example gene 6) corresponds to an operator and inputs and output registers.

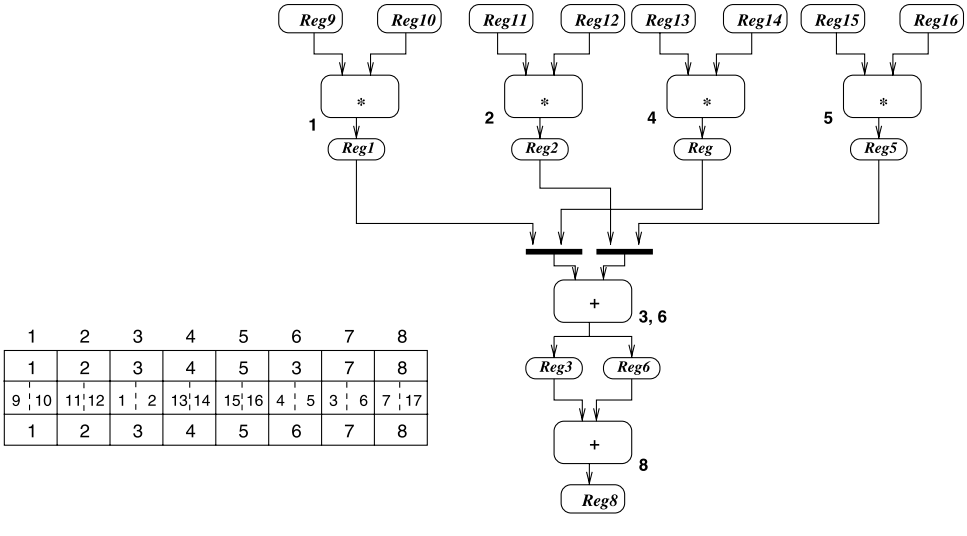


Fig. 4. Sample chromosome for the biquad example after the merger of gene 3 and gene 6. This is illustrated through the merger of functional units indexes. Registers are not merged due to variables life span overlap.

life spans of the output registers for *gene 3* and *gene 6* overlap. Therefore, the registers *cannot* be shared and the indexes of the register fields are not updated.

- The multiplexers of the corresponding nodes are updated accordingly in order to maintain the original data flow.

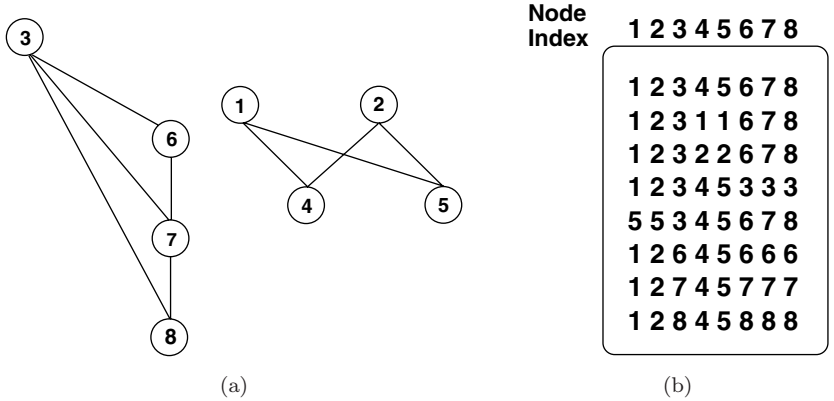
2.3. *Initial population*

At the beginning of each run, an analysis of the DFG is performed. Thus, compatibility relations among DFG nodes as well as registers life spans are analyzed. Compatibility relations are stored in a *compatibility graph*, $G_{\text{comp}}(V, E)$, that consists of vertexes V denoting operations and edges E denoting the compatibility relations among DFG nodes. Nodes that are connected with edges in the graph correspond to a *partial binding*. The variables life spans are stored in a two-dimensional matrix and are used later during register merger and allocation. The compatibility graph for biquad example is shown in Fig. 5(a).

The initial population is then generated based on problem specific data in two steps. First, chromosomes are generated through random enumeration of partial feasible bindings that are directly derived from the compatibility graph $G_{\text{comp}}(V, E)$. Then, the remaining chromosomes in the initial population are generated through random perturbation of the population using the *crossover* and *mutation* operators. The number of chromosomes in each of the above *sets* of the initial population is specified by the user.

2.4. *Technology library*

Minimizing the number of components is not sufficient to guarantee a good design since some components may be more expensive than others under a given technology. We use a *technology library* as an input to our system. The technology library



(a) (b)

Fig. 5. (a) Compatibility graph for the biquad example and (b) sample population of size 8.

Table 1. 0.35 μm 8-bit ASIC component costs.

Component	Area μm^2
Adder	2912
Multiplier	17 648
Register	3056
Mux (2:1)	1016
Mux (4:1)	2184
Mux (8:1)	4952

provides information about the layout area of every individual component along with the delay propagation and was generated based on 0.35 μm technology from austria *micro*systems[18] with some component costs shown in Table 1. The library components are parametrized by their bit length. This provides a good estimate of the datapath cost and controller cost during optimization. The controller cost is estimated dynamically during the allocation process, subject to the allocation of registers and multiplexers.

2.5. *Objective function*

The objective function measures the fitness of each chromosome in the population. The fitness of an individual is obviously crucial for the transmission of its gene information to the next generation. The register-transfer level (RTL) structure is based on a datapath and a controller pair. Datapath components are registers, multiplexers, and functional units. Thus, the cost of the datapath is simply the sum of the cost of individual components given as follows:

$$A_{\text{datapath}} = \sum_i A_{\text{fu}}(i) N_{\text{fu}}(i) + A_r N_r + \sum_j M_{\text{mux}}(j) M_j \,,$$

where $N_{\text{fu}}(i)$ is the number of functional units of type i; $A_{\text{fu}}(i)$ is the area of functional units of type i. N_r is the number of registers while A_r is the area of a register. Finally, $M_{\text{mux}}(j)$ is the number of multiplexers of type j with M_j the corresponding area.

The cost of the controller is a bit more complex and depends on the style one uses. If we assume a PLA implementation, the controller area maybe computed as: $Area_{\text{Controller}} = Width_{\text{PLA}} * Height_{\text{PLA}}$. Based on the area estimation model discussed by Gajski *et al.*,[19] the width could be estimated as the sum of the width of the input AND array, the width of product-term buffers, and the width of the OR array. The height of the PLA is computed as the sum of a latch height, a buffer height, and the height of the AND–OR plane. Thus, the PLA area reduces to:

$$A_{\text{Cont.}} = ((n+m) * \text{Max}(l_w, b_w) + W_p) * (l_h + b_h + r * p) \,,$$

where n and m are the number of PLA inputs and outputs, respectively; W_p is the width of the product-term buffers; b_h, b_w are the height and width of a buffer;

l_h, l_w are the height and width of a latch; r is the transistor pitch while p is the number of distinct product terms. It should be noted that the above controller cost gives an upper bound on the controller area since many optimization procedures such as folding maybe applied in order to further optimize the cost.

Based on the above, the fitness of chromosome i is given as follows:

$$fitness(i) = \frac{1}{\alpha * A_{\text{datapath}} + (1 - \alpha) * A_{\text{Cont.}}}, \tag{1}$$

where α is a parameter that controls the desired trade-off between datapath and controller area.

2.6. *Selection and reproduction*

There are many approaches for selecting parent chromosomes for reproduction. We use the commonly used technique, *roulette wheel selection*, where the selection is based on spinning the roulette wheel N times where N is the population size. A random number between 0 and 1 is generated and compared with the respective cumulative probability q for the individual under consideration. If the random number falls in the interval $q_{i-1} < r \leq q_i$, or if it is less than the cumulative probability of the first individual when considering it, then the individual is selected. It is very clear from the selection process that some "healthy" individuals maybe selected more than once while weaker individuals have a very small chance of getting selected.

2.7. *Genetic operators*

In order to explore the design space, we use two genetic operators: *mutation* and *crossover*. The genetic operators are applied iteratively and by taking turns with their corresponding probabilities. It should be noted that we have also experimented with an *inversion* operator; however, its impact on the optimization process was not noticeable. It is clear that in order to obtain a meaningful coding, one has to make sure that the binding suggested by the operators is feasible. Therefore, we repair unfeasible bindings by breaking apart genes that cause conflicts.

2.7.1. *Mutation*

Mutation is a generic operator that is used for finding new points in the search space. Thus, it selects a DFG node at random and replace it with a different function. We propose a novel mutation operator based on a *shift right* fashion, as illustrated in Fig. 6(a). We select two cut-points randomly between 1 and L where L is the number of DFG nodes. We next perturb the values, except those between the cut-points, by *shifting the genes to the right,* subject to the mutation probability P_m. While shifting, we ensure that nodes that are being merged are compatible.

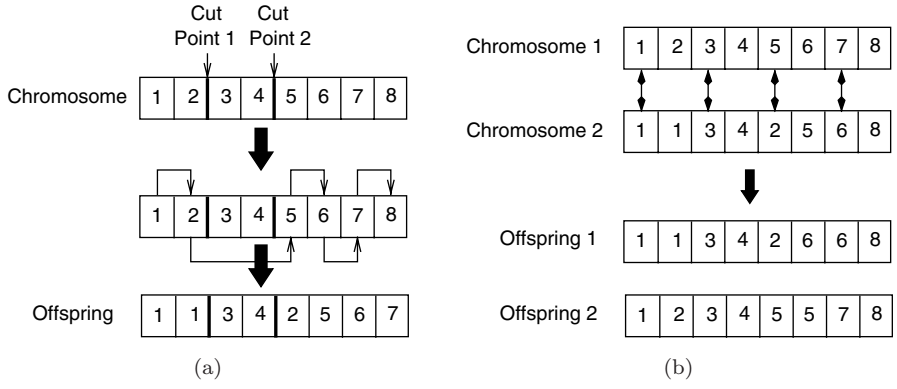


Fig. 6. (a) Mutation operator and (b) crossover operator.

2.7.2. *Crossover*

In order to increase the number of optimal solutions, one needs to overcome the information loss that occurs when the GA converges to a solution. This was done through the use of multiple point uniform crossover. Crossover is a reproduction technique that mates two parent chromosomes and produces two child chromosomes. Given two chromosomes, we apply *multiple-point uniform crossover* with a high probability P_c. Thus, two offspring are created from two parents by selecting each gene from either parent with probability P_c (Fig. 6(b)). It should be noted it has been shown that uniform crossover is capable of generating more diverse new offspring than the traditional one-point or two-point crossover.[3]

2.8. *Algorithm*

Every chromosome represents an intermediate datapath that has different number of registers, multiplexer inputs, functional units and controller cost. During every generation, chromosomes are selected for reproduction, resulting in new datapaths. This is accomplished by merging compatible nodes within each chromosome. The algorithm, shown in Fig. 7, ensures the following:

(1) The merged nodes are compatible. This is done simply by checking if the nodes that are being merged are connected in G_{comp}.
(2) Variables whose life spans do not overlap are merged thus reducing the number of registers in the resulting datapath.
(3) As nodes are merged, inputs are allocated to new multiplexers that are connected to the input ports of the resulting ALU.

2.8.1. *Functional unit allocation*

Every time we merge two genes in a chromosome, we are reducing the datapath cost by the cost of one ALU. Assuming that the combined operation set of the genes

```
Evolutionary_Synthesis(Scheduled_DFG)
{
   ■ Read the scheduled DFG and the resource library. Build the DFG.
   ■ Get the population size (N) and the number of generations (N_g) from the user.
   ■ Generate an initial population, current_pop.
   ■ Evaluate(current_pop)
   ■ Keep_the_best()
   for i = 0 to N_g do
   {
     ■ Select two chromosomes from current_pop for mating based on their fitness.
     ■ if (i % 2 == 0) apply crossover with probability P_xover
       else apply mutation with probability P_m
     ■ Perform register alignment and apply the Left Edge Algorithm in order to optimize
       the number of multiplexers inputs as well as the number of registers.
     ■ Evaluate the cost and the fitness for each chromosome using Eq. (1).
     ■ Save the fittest current solution.
     ■ Repeat the above for the entire current_pop forming a new population, new_pop.
     ■ current_pop ← new_pop
   }
   ■ Generate VHDL description for the scheduled DFG
   ■ Generate VHDL description for the datapath and the controller
   return;
}
```

Fig. 7. Evolutionary synthesis algorithm.

already exists in the library, the cost of a chromosome is reduced by $Cost(ALU_1) + Cost(ALU_2) - Cost(ALU_1 \cup ALU_2)$.

2.8.2. *Mux allocation*

When two genes are merged and assigned to the same chromosome cell, there exist two possible configurations to assign the input ports to the multiplexers, *left* or *right* mux. For noncommutative operations such as subtraction, the configuration is unique. In order to obtain a good MUX cost estimation, we use *incremental registers alignment.* Thus, when considering two genes for merger, we assign the noncommutative operations to the multiplexers at the input ports of the resulting gene first. The reason is that these signals cannot be swapped in order to explore sharing possibilities with other signals. The remaining assignment is done so as to reduce the number of multiplexer inputs, right or left, through register alignment.

2.8.3. *Registers allocation*

A DFG node corresponds to a value that must be assigned to a register for the duration of its life time. Thus, registers can be merged if their life spans do not overlap. Merging two or more genes implies merging their output registers as well. It is possible to have several binding with the same functional units cost but differ in register configurations. In order to minimize the number of registers, we apply locally the left edge algorithm (LEA)[20] incrementally at the output ports of the

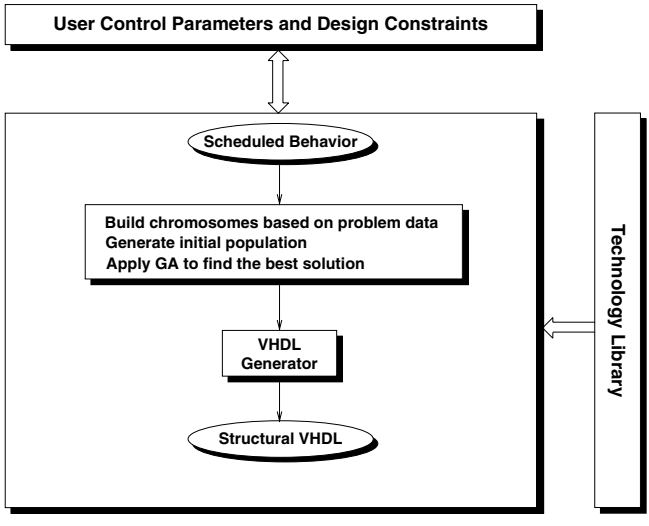


Fig. 8. GenSys design flow.

resulting gene. The LEA running time is extremely fast since we consider a small set of registers at a time.

3. VHDL Post Synthesis Module

The VHDL generator *automatically* generates the resulting structural *datapath* and *controller* in VHDL. Thus, the VHDL generator describes the datapath in terms of structural components available in the design library in addition to the controller. The above process involves:

- Creating the two main components: the datapath and the controller.
- Creating the controller by extracting necessary control signals from the datapath. Feedback latches are added in the finite state machine.
- Creating the datapath by collecting sub-components such as registers, functional units, and multiplexers selected by the allocator.
- Expanding the above components at the register-transfer or logic level, depending on the output option. Thus, creating a simulateable design.

The advantages of the VHDL output is that it can be used to verify the functionality of the generated architecture. Furthermore, since we are targeting Field Programmable Gate Arrays (FPGAs), then a hardware implementation can be easily implemented. Thus, providing an integration path to production CAD synthesis tools. Currently, we are targeting the Altera Flex 10K FPGA family. These devices were chosen primarily due to the sufficient gate count of the family. It should be noted that the simulation of the design can be carried out. In our case, it was shown to be correct through comparison with the scheduled VHDL DFG.

The datapath is described hierarchically using components that are lower level entities contained in top-level entity descriptions. Each component requires a model that describes its behavior to the simulator and synthesis tools. These models are generated dynamically for each instance of the hardware components in the datapath.

The datapath controller is described using a finite state machine (FSM) model that has two processes. The first process declares the state register while the second process declares the combinational part of the design. Each state in the FSM corresponds to a clock cycle. The controller has a partially asynchronous operation using the *reset* signal. Depending on the value of the *state* and the *rising clock edge,* the appropriate micro-operations are executed.

Each state specifies the next state and a set of control signals. The present state is usually encoded as binary values $p_k \cdots p_1 p_0$ where $k \leq \lceil \log_2 \rceil - 1$ and m is the number of states. The next states are encoded as binary values $r_k \cdots r_1 r_0$. Each output signal, c_i, controls a functional unit, a storage element, or an interconnect component in the datapath. The next state are determined based on the next contiguous clock cycle, as implied by the schedule, except in the case of loops and branches. The control signals are determined during allocation.

4. Experimental Results

4.1. *Parameter tuning*

In order to apply an evolutionary algorithm successfully to a specific optimization problem, several parameters have to be adjusted. Most important are the control parameters such as crossover rate, mutation rate, population size N, and the number of generations N_g. We have performed experiments on a set of problems with a crossover rate of $1.0, 0.9, 0.8, 0.7$, and 0.6, and a mutation rate of $0.2, 0.15, 0.1, 0.05$, and 0.009. The population size was varied between 50 and 200 and the number of generations between 50 and 500. It was determined experimentally that for the problems at hand, a population size between 150 and 200 and number of generation between 200 and 250 was sufficient to arrive at good solutions. The $P_{\text{xover}} = 0.9$ and $P_m = 0.15$ performed quite well. Figure 9 depicts the fitness of the best chromosome in a population of 500 such chromosomes as well as the mean fitness of all parents, the minimum and maximum, as a function of the number of generations. It is clear how quickly our method converges to a good solution before it saturates in less than 150 generations. Figure 10(a) illustrates the improvement in the datapath cost as the population size is steadily increased while Fig. 10(b) shows the variation in time as the the number of generations is increased.

4.2. *Benchmark results*

We implemented the described allocation method on a Pentium 266 PC running Linux. The synthesis system, GeneSys, is shown in Fig. 8. We tested our method

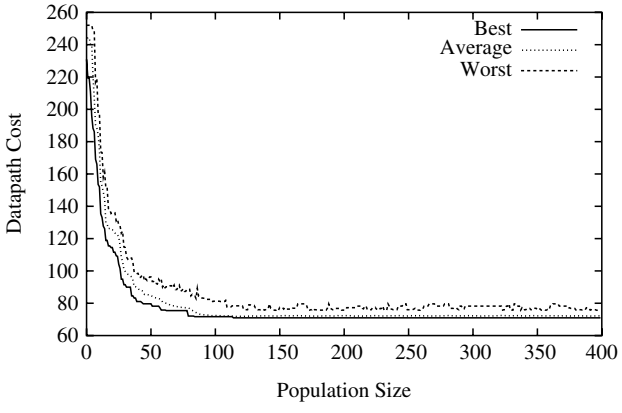


Fig. 9. Curves depict the best, worst and average fitness of all parent chromosomes in a population of 500 chromosomes for the elliptic wave filter benchmark. Note the quick convergence to a sub-optimum solution in less than 150 generations.

on a suite of design examples that include the differential equation example, the fifth order elliptical wave filter, the Facet example, the AR filter, and the discrete cosine transform (DCT) benchmark example. We compare our results to PSGA_Synth,[14] ADaPAS,[11] HAL,[7] simulated evolution method (SE),[12] Splicer,[21] Facet,[22] MABAL,[23] Salsa,[6] and GABIND.[16] All synthesis results have been produced for the 0.35 μm component library from austria *micro*systems[18] with some component costs shown in Table 1 for illustration purposes. Please note that the controller cost was not included since, to our best knowledge, no one has reported in the literature controller cost. Thus, it was difficult to compare.

For all attempted circuits, the system found the best results as implied by the schedule. The method is very fast and all reported results were produced in at most 1.87 CPU minutes including scheduling time. In what follows, we present and discuss these examples.

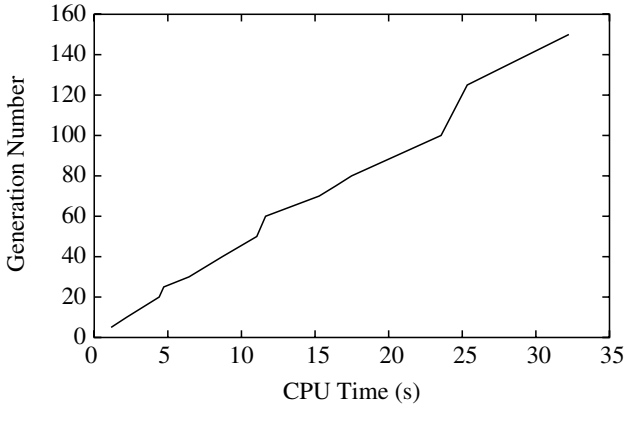
4.2.1. *The facet example*

This is one of the earliest examples, and was first introduced by Tseng *et al.*[22] The DFG has eight nodes. Results comparisons are shown in Table 2. For this example, our system generated two different solutions that differ with the functional units binding. The second binding slightly outperform all other systems.

4.2.2. *The differential equation example from HAL*

This example solves a second order differential equation, and was first introduced by Paulin *et al.*[7] The DFG has 10 nodes. Results comparisons are shown in Table 3.

(a)



(b)

Fig. 10. (a) Population size versus design cost and (b) number of generation versus CPU time for the elliptic wave filter benchmark.

Table 2. Results comparisons for the facet benchmark.

System	ALUs	# Reg	# Mux	# Mux in	CPU time (s)	Cost
Ours 1	(/)(−&+)(∗|+)	7	5	10	3.51	70 391
Ours 2	(+/)(−&+)(∗|)	6	5	10	2.86	67 332
HAL	(/)(−&+)(∗|+)	7	6	13	n/a	72 137
Facet	(/)(−&+)(∗|+)	8	7	15	n/a	76 357
Splicer	(/)(−&+)(∗|+)	7	4	8	1.4	69 227
GABIND	(/)(−&+)(∗|+)	6	5	11	28	67 917

Table 3. Results comparisons for the differential equation benchmark.

System	ALUs	Regs	Mux	Mux in	CPU time (s)	Cost
Ours 1	(*)(*)(+)(−)(>)	5	6	13	7.78	66 527
Ours 2	(*)(*)(+−)(>)	5	6	13	4.72	64 115
PSGA_Synth	(*)(*)(+)(−)(>)	5	6	13	n/a	66 527
HAL	(*)(*)(+)(−)(>)	5	6	13	140	66 527
GABIND	(*)(*)(+)(−)(>)	5	—	8	38	64 255
Splicer	(*)(*)(+)(−)(>)	5	5	11	1245	66 001

Table 4. Results from the AR filter benchmark.

Design characteristics	Clock cycles	ALUs	Regs	Mux	Mux in
Nonpipelined multicycled multipliers	11	4(*), 2(+)	8	12	46
	15	3(*), 2(+)	9	10	45
	18	2(*), 2(+)	8	6	36
	34	1(*), 1(+)	7	4	35
Pipelined multipliers	11	4 (*), 2(+)	8	12	43
	13	2 (*), 2(+)	8	8	39
	16	2(*), 1(+)	10	6	42
	20	1(*), 1(+)	9	4	38

Table 5. Results from the fifth-order elliptic wave filter benchmark.

Design characteristics	Clock cycles	ALUs	Regs	Mux	Mux in
Nonpipelined multicycled multipliers	17	3(*), 3(+)	10	8	32
	18	2(*), 3(+)	9	8	28
	19	1(*), 3(+)	9	5	28
	21	1(*), 2(+)	8	5	24
Pipelined multipliers	17	2(*), 3(+)	10	8	34
	18	1(*), 3(+)	10	7	28
	19	1(*), 2(+)	10	5	27
	21	1(*), 2(+)	8	5	24

For this example, our system also generated two different bindings with the second binding outperforming all other systems.

4.2.3. *The AR filter*

The AR filter was initially used by Jain *et al.*[5] We derived schedules using 11, 13, 15, 16, 18, 20, and 34 clock cycles. The schedules are based on multicycled and pipelined multipliers. Detailed results for this example are shown in Table 4. Note that it was not possible to report result comparisons for this example due to the lack of details in the literature regarding this example.

4.2.4. *Fifth-order wave elliptic filter*

This example was popularized by Paulin *et al.*[7] The example has 34 nodes and consists of addition and multiplication operators only. We derive six designs based

on four different schedules in 17, 18 19, and 21 clock cycles and based on multicycled and pipelined multipliers. In all ran cases, our system report the same number of functional units but allocated either fewer registers, fewer multiplexer inputs or both. Note that *PSGA_Synth* did not report the number of multiplexers inputs. Detailed results for this example are shown in Table 5. Tables 6 and 7

Table 6. Result comparisons from the wave (multicycled multipliers) benchmark.

Clock cycles	System	ALUs	# Reg	# Mux in	CPU time (s)	Cost
17	Ours	3(*), 3(+)	10	32	20.28	11 0864
	PSGA_Synth	3(*), 3(+)	10	—	10	—
	HAL	3(*), 3(+)	12	—	120	—
18	Ours	2(*), 3(+)	9	28	22.61	87 832
	PSGA_Synth	3(*), 2(+)	9	—	10	—
	HAL	3(*), 2(+)	12	—	240	—
19	Ours	1(*), 3(+)	10	27	21.72	75 570
	HAL	1(*), 2(+)	12	31	360	81 098
	SE	2(*), 2(+)	10	31	1791	122 634
21	Ours	1(*), 2(+)	8	24	14.91	61 888
	HAL	1(*), 2(+)	12	31	360	78 186
	SE	1(*), 2(+)	11	24	2870	71 056
	MABAL	2(*), 2(+)	11	43	—	99 762
	PSGA_Synth	1(*), 2(+)	10	—	10	—

Table 7. Result comparisons from the wave (pipelined multipliers) benchmark.

Clock cycles	System	ALUs	# Reg	# Mux in	CPU time (s)	Cost
17	Ours	2(*), 3(+)	10	32	24.40	102 816
	GABIND	2(*), 3(+)	13	29	210	110 238
	SE	2(*), 3(+)	11	31	1511	105 290
	HAL	2(*), 3(+)	12	31	120	108 846
	PSGA_Synth	2(*), 3(+)	10	—	10	—
	ALPS	2(*), 3(+)	11	—	—	—
	ADaPAS	2(*), 3(+)	10	—	—	—
18	Ours	1(*), 3(+)	10	28	18.37	78 040
	GABIND	1(*), 3(+)	11	31	251	82 842
	SE	1(*), 3(+)	10	24	2096	75 712
	HAL	1(*), 3(+)	12	31	240	85 898
	PSGA_Synth	1(*), 3(+)	10	—	10.25	—
	ALPS	1(*), 3(+)	11	—	—	—
	ADaPAS	1(*), 3(+)	9	—	—	—
19	Ours	1(*), 2(+)	10	27	17.21	74 546
	GABIND	1(*), 2(+)	14	27	255	86 770
	HAL	1(*), 2(+)	12	26	360	82 988
	SE	1(*), 2(+)	11	25	2096	79 350
21	Ours	1(*), 2(+)	10	24	14.91	72 800
	HAL	1(*), 2(+)	12	31	360	82 986
	SE	1(*), 3(+)	11	24	2870	78 768
	Splicer	1(*), 2(+)	11	43	55	86 914

provide comparison of our system with PSGA_Synth, MABAL, HAL, ADapAs, and SE.

4.2.5. *Discrete cosine transform*

The Discrete Cosine Transform (DCT) was first reported by Nestor *et al.*[6] It is a large benchmark that consists of 48 operators: 16 multiply by constant, 25 add, and 7 subtract. The DCT is used extensively in image coding and compression, and has been implemented in hardware for special purpose image processors. We generated four schedules for this example and we ran it with pipelined as well as with multicycled multipliers. We show the detailed results for this example in Table 8. Results comparisons are shown in Table 9. Note that in both reported cases our system slightly outperformed PSGA_Synth and Salsa.

5. Conclusion

A datapath allocation problem was presented based on an evolutionary algorithm. The proposed system can handle pipelined and multicycled operations and creates a VHDL description targeted for Altera FPGAs. The system provides the best solution in terms of the number and types of functional units, the number of registers,

Table 8. Results from the Discrete Cosine Transfer (DCT) benchmark.

Design characteristics	Clock cycles	ALUs	Regs	Mux	Mux in
Nonpipelined multicycled multipliers	10	5(*), 5(+−)	19	14	61
	11	5(*), 5(+−)	19	15	58
	14	4(*), 4(+−)	20	14	64
	18	2(*), 3(+−)	18	9	56
	19	2(*), 3(+−)	18	8	59
	25	1(*), 2(+−)	21	5	56
	34	1(*), 2(+−)	19	5	49
Pipelined multipliers	10	3(*), 6(+−)	22	13	64
	11	3(*), 3(+−)	18	9	60
	13	3(*), 4(+−)	18	10	59
	19	1(*), 3(+−)	14	7	47
	20	1(*), 3(+−)	18	9	54
	25	1(*), 2(+−)	21	5	56
	33	1(*), 2(+−)	20	5	52

Table 9. Results comparisons for the DCT benchmark.

System	Clock cycles	ALUs	# Reg	Cost
Ours 1	18	2(*), 3(+−)	16	92 922
Salsa	18	3(*), 2(+−)	13	98 496
Ours 2	19	2(*), 3(+−)	16	92 922
PSGA_Synth	19	3(*), 2(+−)	14	101 552

and the number of multiplexer inputs. The method was implemented on a Linux station using C++ and several benchmarks were attempted. Future work includes the incorporation of test consideration and test scheduling during the synthesis process.

Acknowledgments

The authors would like to thank Altera Corporation and Ms. Ralene Marcoccia for making their EDA tools available through the University Program.

References

1. M. McFarland, A. Parker and R. Compasano, The high level synthesis of digital systems, *Proc. IEEE* **78** (1990) 301–318.
2. M. Garey and D. S. Johnson, *Computer and Intractability* (W. H. Freeman, 1979).
3. D. Fogel, *Evolutionary Computation*: *Toward a New Philosophy of Machine Intelligence* (IEEE Press, 1995).
4. C. Gebotys and M. Elmasri, VLSI design synthesis with testability, *Proc. 25th Design Automation Conf.* (1988), pp. 16–21.
5. R. Jain, K. Kukukcakar, M. Mlinar and A. Parker, Experience with the Adam synthesis system, *Proc. 26th Design Automation Conf.* (1989).
6. J. Nestor and G. Krishnamoorthy, SALSA: A new approach to scheduling with timing constraints, *IEEE Trans. CAD* **12** (1993) 1107–1122.
7. P. Paulin and J. P. Knight, Forced-directed scheduling for the behavioral synthesis of ASIC's, *IEEE Trans. CAD* **8** (1989) 661–679.
8. J. Rabaey, H. D. Man, J. Vanhoof, G. Goossens and F. Catthoor, Cathedral II: A synthesis system for multiprocessor DSP, *Silicon Compilation*, ed. D. Gajski (Addison Wesley, 1988), pp. 311–360.
9. D. E. Thomas, E. M. Dirkes, R. A. Walker, J. V. Rajan and R. L. Blackburn, The System Architects Workbench, *Proc. 25th Design Automation Conf.* (1988), pp. 337–343.
10. S. Devadas and R. Newton, Algorithms for hardware allocation in data path synthesis, *IEEE Trans. CAD* **8** (1989) 768–781.
11. N. Wehn, M. Held and M. Glesner, A novel scheduling/allocation approach for datapath synthesis based on genetic paradigms, *Proc. TC10/W10.5 Workshop Logic and Architectural Synthesis* (1990), pp. 47–56.
12. T. Ly and J. Mowchenko, Applying simulated evolution to high-level synthesis, *IEEE Trans. CAD* **12** (1993) 389–409.
13. R. Martin and J. Knight, Genetic algorithms for optimization of integrated circuit synthesis, *Proc. 5th Int. Conf. GA and Their Applications* (1993), pp. 432–438.
14. M. Dhodhi, F. Hielscher, R. Storer and J. Bhasker, Datapath synthesis using a problem-space genetic algorithm, *IEEE Trans. CAD* **14** (1995) 934–944.
15. T. Blickle, J. Teich and L. Thiele, System-level synthesis using evolutionary algorithms, *J. Design Automation for Embedded Syst.* **3** (1998) 23–58.
16. C. Mandal, P. Chakrabarti and S. Ghose, GABIND: A GA approach to allocation and binding for the high-level synthesis of data paths, *IEEE Trans. VLSI* **8** (2000) 747–750.
17. G. De Micheli, *Synthesis and Optimization of Digital Circuits* (McGraw Hill, 1994).
18. 0.35 μm CMOS digital standard cell databook, *austriamicrosystems* (2001).

19. D. Gajski, N. Dutt, A. Wu and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design* (Kluwer Academic Publishers, 1992).
20. F. Kurdahi and A. Parker, REAL: A program for register allocation, *Proc. 24th Design Automation Conf.* (1987), pp. 210–215.
21. B. M. Pangrle, Splicer: A heuristic approach to connectivity binding, *Proc. 25th Design Automation Conf.* (1988), pp. 536–541.
22. C. Tseng and D. P. Siewiorek, Automated synthesis of data paths in digital systems, *IEEE Trans. CAD* **5** (1986) 379–395.
23. K. Kucukcakar and A. C. Parker, MABAL: A software package for module and bus allocation, Technical Report CRI-88-61, University of Southern California (1989).