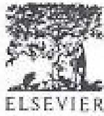
Sadiq M. Sait, Ali S, Benten MST•Scheduling and allocation in high-level synthesis using stochastic techniques •MICROELECTRONICS JOURNAL 27 (8): 693-712 NOV 1996 •



93–712
.imited
served
$15.00

Research paper

PII: S0026-2692(96)00001-8

Scheduling and allocation in high-level synthesis using stochastic techniques

Sadiq M. Sait, Shahid Ali and Muhammad S.T. Benten

Computer Engineering Department, King Fahd University of Petroleum and Minerals, KFUPM Box 673, Dhahran 31261, Saudi Arabia. Tel: 966(3)-8602217. Fax: 966(3)-8602366. E-mail: sadiq@ccse.kfupm.edu.sa

High-level synthesis is the process of automatically translating abstract behavioral models of digital systems to implementable hardware. Operation scheduling and hardware allocation are the two most important phases in the synthesis of circuits from behavioral specification. Scheduling and allocation can be formulated as an optimization problem. In this work, a unique approach to scheduling and allocation problem using the genetic algorithm (GA) is described. This approach is different from a previous attempt using GA (Wehn *et al.*, *IFIP Working Conference on Logic and Architecture Synthesis*, Paris, 1990, pp. 47–56) in many respects. The main contributions include: (1) a new chromosomal representation for scheduling and for two subproblems of allocation; and (2) two novel crossover operators to generate legal schedules. In addition the application of tabu search (TS) to scheduling and allocation is also implemented and studied. Two implementations of TS are reported and compared. Both genetic scheduling and allocation (GSA) and tabu scheduling and allocation (TSA) have been tested on various benchmarks and results obtained for data-oriented control-data flow graphs are compared with other implementations in the literature. (A discussion on GSA was presented at the European Design Automation Conference Euro-DAC'94 in Grenoble, France, and TSA at the International Conference on Electronics, Circuits and Systems – ICECS'94 in Cairo, Egypt.) A novel interconnect optimization technique using the GA is also realized. Copyright © 1996 Elsevier Science Ltd.

1. Introduction

High-level synthesis (HLS) [1] is the process of automatically translating abstract behavioral models of digital systems to implementable hardware. Operation scheduling and hardware allocation are the two most important phases in the HLS of circuits from behavioral descriptions. While scheduling distributes the execution of operations throughout time steps, allocation assigns hardware to operations and values. Allocation of hardware cells includes functional unit allocation, register allocation and bus allocation. Allocation determines the interconnections required. Early techniques for scheduling and allocation were simple, such as the as-soon-as-possible (ASAP) algorithm [1, 2]. These techniques tend to minimize schedule length while ignoring the hardware costs. Since

then different techniques have been developed to minimize hardware costs as well. These techniques are either constructive, transformational or exact. Constructive techniques usually use greedy heuristics which do not guarantee that an optimal solution will be found. Examples of such a technique are list scheduling [3] and force-directed scheduling [4]. Transformational techniques alter an existing schedule to find new low-cost schedules. Examples of transformational techniques for scheduling and allocation are simulated annealing [5, 6], genetic algorithm [7, 8] and simulated evolution [9]. Exact techniques are usually based on integer linear-programming.

Scheduling and allocation are closely interrelated, but are usually dealt with separately because of the complexity involved. Optimizing them separately may give suboptimal results because the possibility that the best designs (in terms of overall cost) may require suboptimum schedules and/or allocations may not be considered. Thus, one can combine scheduling and allocation in an attempt to optimize a cost function that includes both the number of control steps and the hardware.

This paper describes unique approaches to scheduling and allocation using genetic algorithm (GA) and tabu search (TS) for data-oriented control/data flow graphs (CDFGs). The remainder of this paper is organized as follows. Synthesis process and its complexity are illustrated in Section 2. After an introduction to GA in Section 3, genetic scheduling and allocation (GSA) [8, 10] is described in Section 4. Section 5 provides an introduction to TS. Section 6 discusses tabu scheduling and allocation (TSA) [8, 11]. Genetic algorithm for data path synthesis (DPS) [8] is described in Section 7. Section 8 presents the results of scheduling and allocation, and data path synthesis on various benchmarks. Finally, in Section 9 we present conclusions.

2. Synthesis process and problem illustration

A simplified example of synthesis is illustrated in Fig. 1. The behavioral description (Fig. 1a) is compiled into an intermediate form (IF) which is then transformed into optimized IF. The CDFG shown in Fig. 1b corresponds to this optimized IF. Note that the common subexpression $S \times T$ is computed only once. Scheduled CDFG (Fig. 1c) shows one of the possible schedules for a given CDFG. The small circles correspond to registers. It is assumed that input values S, T, U, V and W should be available even after the computation of output values X and Y. Therefore, the corresponding registers can not be used. Note that a temporary register Z is used to store $S \times T$, whereas the register for Y is temporarily used to store $U \times V$. Figure 1d

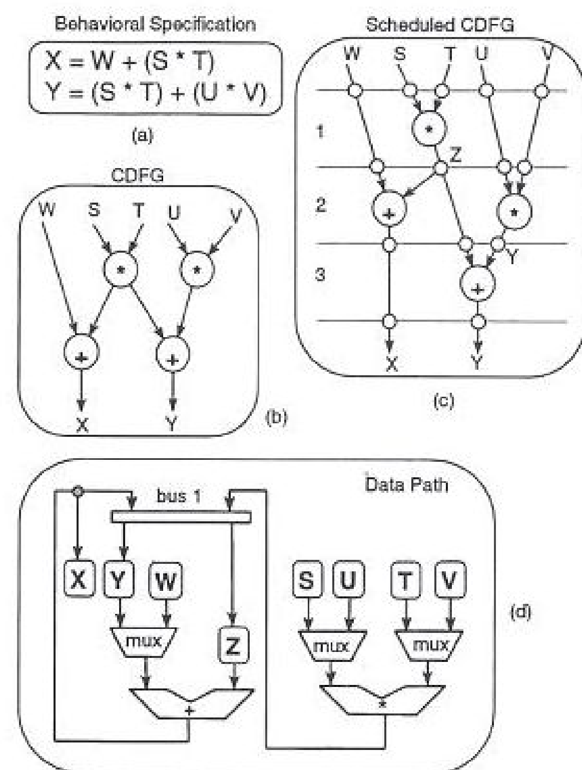
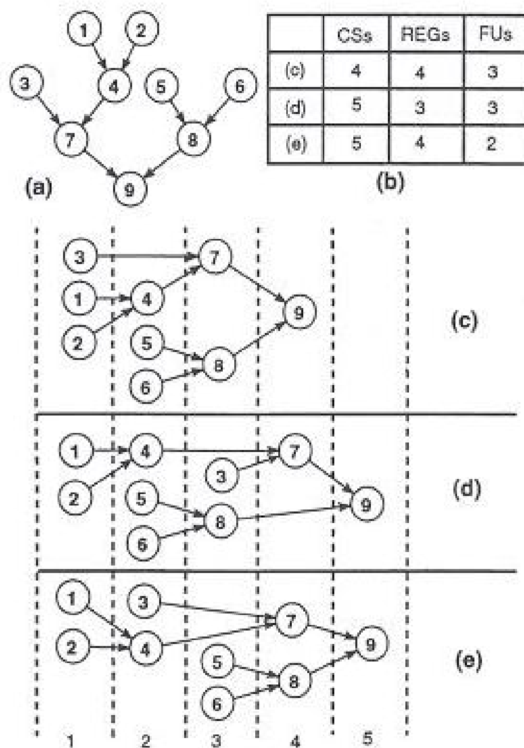


Fig. 1. Example of synthesis process.

shows the data path for this schedule after the allocation of functional units, registers and buses is performed.

As the transformation of a behavioral specification into a structural design, with limited resources, is an NP-hard problem, one tries to simplify the search for efficient approximations by subdividing the general problem into subproblems of scheduling and allocation. As stated earlier, this approach gives suboptimal results. This is because the two subproblems are highly interdependent. The strong interdependence between the allocation of resources and the scheduling of operations into time steps can be illustrated with the help of the simple example of Fig. 2. The CDFG is shown in Fig. 2a. Figure 2b gives the number of control steps, the minimum number of registers needed and the minimum number of functional units required for the different schedules of Figs. 2c–2e. The example makes it clear how the number of required control steps is mutually dependent on the number of available registers and functional units. It also shows how every partial problem is interdependent within the allocation, namely, register allocation and functional unit allocation.
The effect of functional unit allocation and register allocation on interconnection cost is illustrated with the help of the example of Fig. 3. Figure 3a shows a simple CDFG with three additions and one multiplication for the behavioral specification given in Fig. 3b. There are seven variables involved. The number of control steps for which the variable is active is called its lifetime. The lifetime analysis of variables used is shown in Fig. 3c. Variables with disjoint lifetimes can be stored in the same register. Thus we can form groups of ($v1$, $v6$) or ($v3$, $v6$) and



	CSs	REGs	FUs
(c)	4	4	3
(d)	5	3	3
(e)	5	4	2

Fig. 2. Interdependence between resource allocation and scheduling.

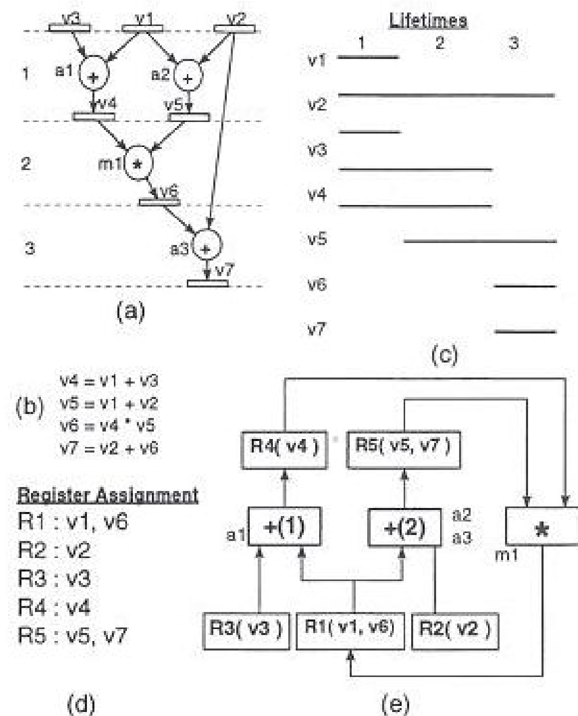


Fig. 3. Effect of functional unit and register allocation on interconnection cost.

($v4$, $v7$) or ($v5$, $v7$). (For simplicity, it will be assumed that a register can not be used to store a new value if it is involved in an operation during a particular control step.) The register assignments are shown in Fig. 3d. Variables $v1$ and $v6$ are grouped into register **R1** and $v5$ and $v7$ into **R5**. The data path using this assignment is shown in Fig. 3e. It can be seen that operation $a3$ can be assigned to adder 1 or 2. No more interconnections are needed if it is assigned to adder 2, as is done in Fig. 3e. Had it been assigned to adder 1, an extra interconnection from register **R2** to adder 1 would have been necessary which would also necessitate a multiplexer at one of the inputs of adder 1. It can also be seen that if variable $v6$ were grouped with $v3$, an extra interconnection and a multiplexer can not be avoided whether we assign operation $a3$ to adder 1 or 2. Grouping $v7$ with $v5$ does not require any extra interconnection as both are produced by adder 2. This clearly illustrates the effect of functional unit allocation and register allocation on the interconnection cost.

3. GA: an introduction

GA [12] works by emulating the natural process of evolution as a means of progressing toward the optimum. The algorithm starts with a population which consists of several solutions to the optimization problem. A member of a population is called an individual. A fitness value is associated with each individual. Each solution in the population or an individual is encoded as a string of symbols. These symbols are known as genes and the solution string is called a chromosome. The values taken by genes are called alleles. Several pair of individuals (parents) in the population mate to produce offspring by applying the genetic operator crossover. Selection of parents is done by repeated use of a choice function. A number of individuals and offspring are passed to a new generation such that the number of individuals in the new population is the same as in the old population. A selection function determines which strings form the population in the next generation. Each surviving string undergoes mutation and inversion with a specified probability.

The overall picture of a GA is as follows. Encoding is devised for a problem in hand. A population of encoded solutions is created. Fitness of each solution is found using an evaluation function. Two parents are selected for crossover which results in two offspring. Offspring are then mutated with a very low probability. After the crossover is applied a specified number of times, we get a population of offspring along with the old population of size n. A selection function is used to select individuals from these two populations to obtain the new population of size n. The above steps are then repeated for a specified number of generations. The best solution in the final population is the result of GA.

4. Scheduling and allocation using GA

In order to formulate scheduling and allocation as an optimization problem, a suitable cost function is required. The optimization technique will then attempt to optimize the value of this function. Since we want to optimize scheduling and allocation tasks jointly we need to incorporate both time related and hardware related terms in our cost function. The cost function C that will be optimized by the genetic algorithm is given and explained below:

$$C = W_{cs} \times N_{cs} + W_{reg} \times N_{reg} + W_{bus} \times N_{bus} + W_{fu} \times N_{fu} + W_{ic} \times N_{ic} \quad (1)$$

where W is the weight assigned and N is the number. The subscripts cs, reg, bus, fu and ic correspond to control steps, registers, buses, functional units and interconnections.

The algorithm starts with a specified upper bound on the number of control steps. During the optimization process the operations are

assigned to control steps and functional units. Each functional unit has two inputs labeled as 1 and 2. Besides assignment of operations to control steps and functional units, variables are assigned to functional unit inputs. Constants are always assigned to the same input as this helps in optimizing the number of interconnections. The number of registers and buses are optimized. Allocation of variables to registers and data transfers to buses is not actually made. The numbers of registers and buses as given by the final solution are optimal for the given schedule. Only a compromised estimation of the interconnection cost is used.

4.1 Chromosome

GAs work on the coding of the problem rather than on the actual problem. This coding is known as chromosomal representation. Devising a good coding is particularly necessary for better design space exploration by the GA. A given high-level specification of the description of the circuit is compiled using *lex* and *yacc* Unix utilities. A CDFG is then obtained from the compiled version. Any schedule should satisfy the precedence constraints implied by CDFG.

Since we want to combine scheduling and allocation into one optimization problem, the coding has to reflect this. This can be done only to a certain extent. Finding an encoding for all the parameters is nearly impossible as there are too many constraints. The coding that is adopted is shown in Fig. 4. Each gene has three values— control step number, functional unit number and the number of the functional unit input to which the left variable of the operation is assigned. The first row in the figure gives the operation number to which the above three values correspond. This coding will be manipulated by the genetic operators. Figure 4 corresponds to the schedule shown in Fig. 5. It is necessary to see why this coding is good enough to optimize scheduling and allocation tasks. With this representation the three subproblems are solved completely, namely, control step

Op Num	1	2	3	4	5	6	7	8	9
Ctrl Step	1	1	3	2	2	2	4	3	5
Func Unit	3	1	2	1	2	3	3	3	2
FU input	1	1	1	2	2	1	1	2	1

Fig. 4. Chromosome.

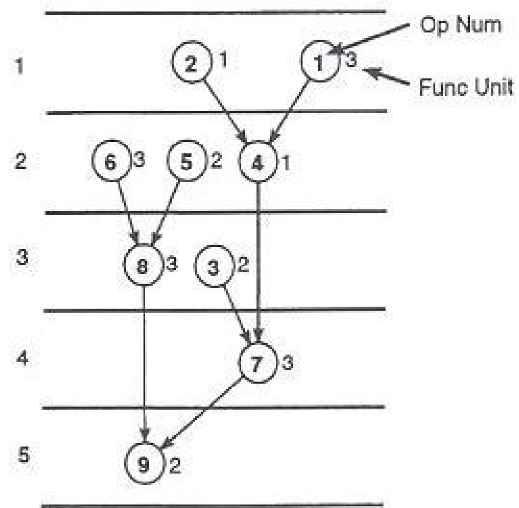


Fig. 5. Schedule corresponding to the chromosome of Fig. 4.

assignment, functional unit assignment and functional unit input assignment. Given this information, the exact number of registers and buses can be found, whereas only a fair estimation of interconnection cost can be obtained. The chromosome in [7] has operation number in depth-first order and alleles corresponding to mobility values (see below) that are filled constructively. Special genes at the end of the chromosome give the number of each type of functional unit.

4.2 Initial population

A good initial population is necessary for proper functioning of the GA reported in this research. GAs work by adopting good structures from the population to generate better individuals.

Therefore, initial population should be as diverse as possible. In this implementation the members of the initial population are created by using the following four scheduling schemes: (1) as soon as possible (ASAP) scheduling [13]; (2) as late as possible (ALAP) scheduling; (3) mobility-down variation of ASAP; and (4) mobility-up variation of ALAP.

ASAP scheduling assigns the operations in the earliest possible control steps, whereas ALAP scheduling assigns the operations in the latest possible control steps. For a given limit on control steps, mobility of each operation, which is the difference between ASAP and ALAP control step values, is calculated. The term mobility-down scheduling as used here means that operations are scheduled in ASAP manner within their mobility range taking care of the precedence constraints. Note that in a CDFG, mobility is used from upper nodes to lower nodes. If we reverse this sequence, we will get mobility-up scheduling. Functional units for each control step are assigned sequentially and randomly perturbed in the end. Assignment of the left variable to the functional unit input is done randomly.

4.3 Choice function

The first step to get a new generation is to select parents on which genetic operators are to be applied. The selection of parents is an important step which affects the population in the new generation. Selection of fittest parents leads to premature convergence. Thus an appropriate choice function is required. This depends on how the fitness of a member of the population is calculated.

4.3.1 Fitness calculation

The GA works naturally on the maximization problem whereas our cost function has to be minimized. Thus the cost minimization problem is converted to a fitness maximization problem as follows. The maximum cost C_{max} in the entire population is determined and each cost c_i is subtracted from this value to get the fitness f_i of individual i. Fitness scaling is used to avoid premature convergence. One method is linear scaling [12]. Linear scaling runs into problems in later runs of the GA when most of the fitness values are close to each other and some lethal members have very low fitness values. This leads to negative fitness values. To avoid this situation sigma (σ) truncation was proposed [12]. All the fitness values are preprocessed to calculate modified fitness values f'_i as follows:

$$f'_i = f_i - (f_{avg} - C_{mult} \times \sigma) \qquad (2)$$

where σ is the standard deviation of the population and C_{mult} is the multiplying constant between one and three. The negative values ($f'_i < 0$) are arbitrarily set to zero. After this truncation, linear scaling can proceed without the danger of negative results.

4.3.2 Sample space

Based on the scaled fitness value a probability is calculated for each individual. This is multiplied by the size of the population n to get expected number of times an individual should be selected (e_i) as parent:

$$e_i = \left(f'_i / \sum_{i=1}^{n} f'_i \right) \times n \qquad (3)$$

A sample space is defined based on e_i values. It consists of an array of records with two fields—a member identification number field and a probability field. For example, if $e_j = 2.6$, then individual j will receive three slots $(j, 1.0)$, $(j, 1.0)$ and $(j, 0.6)$ in the sample space. Assume that there is a total of m slots in the sample space. To select a parent a random number is generated between 1 and m and the individual corresponding to that slot is selected as parent with the probability of that slot. This process is repeated until a parent is selected. According to this scheme the fitter individual will get more slots in the sample space and have a high chance of being selected. Note that the scheme still maintains

diversity in the population because the selection is random over the sample space.

4.4 Crossover

The nodes in CDFG have precedence constraints that should not be violated when the crossover operator is applied. In [7], a simple two point crossover followed by a modified ASAP scheduling was proposed. This technique can produce schedules which are longer than the specified control step limit and is thus believed to take longer to find good schedules. Note that scheduling is performed each time the crossover is applied. We opted to have a crossover that would always give a valid schedule rather than a crossover where scheduling has to be done separately. Given the coding as described in an earlier section, it is a difficult proposition to be resolved. If we fix the order of nodes in the chromosome, a simple one or two point crossover will result in an invalid offspring chromosome. The following schemes developed in this research alleviate the above problems and always generate valid offspring.

4.4.1 Alternating crossover

The term alternating crossover as used here means that given the same order of genes in both parents, we take genes from the two parents in the alternating sequence such that whenever there is a violation of precedence constraint we take the gene from the other parent but maintain the alternating sequence. It is found that if we put the genes in the reverse depth-first order such that successors are always on the left-hand side of their predecessors, we can use the alternating crossover to generate valid offspring. It works because whenever we take a node that is to be scheduled all of its successors are already scheduled and thus we can check for any violations.

A working example of the alternating crossover is shown in Fig. 6. Figure 6a shows the two selected parents (p_1 and p_2) for crossover and Fig. 6b shows the resulting offspring (*os*) with

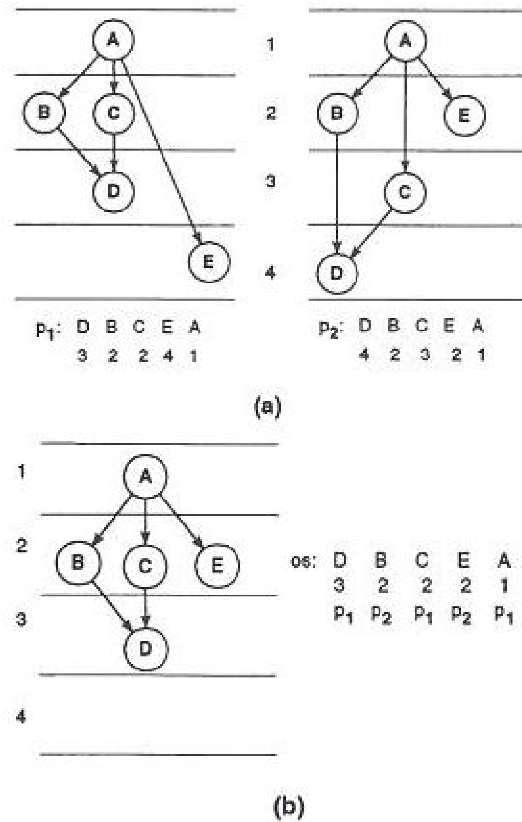


Fig. 6. Alternating crossover example with no scheduling violations: (a) parents; (b) offspring.

genes labeled with the parent tag from which it is taken. It can be seen that there are no scheduling violations in this example. An example which results in such a violation is shown in Fig. 7. Figure 7a shows two parents. As indicated in Fig. 7b, during crossover we take alternating genes from each parent. At one point we can not take a gene from parent 1 so this gene is taken from parent 2, but the alternating sequence is maintained and the next gene is also taken from parent 2.

4.4.2 Order crossover

It is found that alternating crossover is not able to inherit good structures from the parents. The main reason for this is that it works bottom up

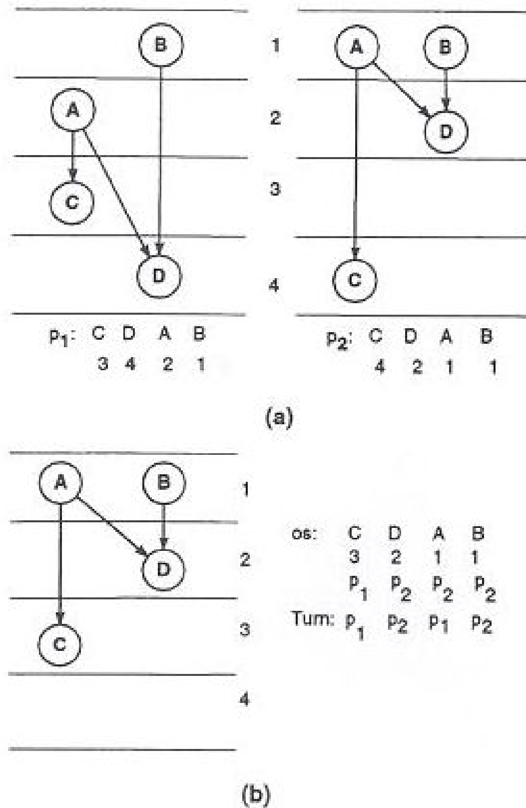


Fig. 7. Alternating crossover example with schedule violations: (a) parents; (b) offspring.

and things become constrained for upper operations. Thus the chance of mixing the genes becomes less. For this reason we started looking for a better crossover operator. Let us remove the restriction on the order of the genes in the chromosome. A simple order crossover works as follows. A cross point is randomly generated and genes on the left side of one parent are copied to offspring in those positions. The other parent is scanned from left to right and these genes are stored in the remaining positions of the offspring in that order (Fig. 8). This ensures that no genes are duplicated or missed.

Using this simple order crossover will of course give invalid schedules. The technique we adopted to avoid invalid schedules is as follows. The cross

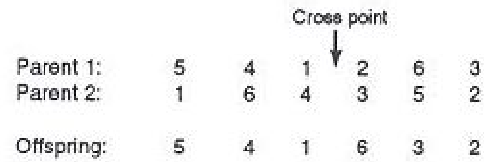


Fig. 8. Simple order crossover.

point is randomly generated and left genes of one parent are copied to the offspring. This determines the schedule for some operations. Given a schedule for some operations in CDFG, the ASAP schedule for the remaining operations can be determined. Those genes from the other parent which do not violate the precedence constraints are copied to the offspring and those which do violate are taken from the first parent. The ASAP values are used to check any violations. An example of this is shown in Fig. 9. The cross point is between the third and fourth genes of parent p_1. The left three genes (A, C, F) from parent p_1 are copied to the offspring and the ASAP schedule for the remaining genes as induced by genes (A, C, F) is determined. Since none of the remaining genes from the other parent violate the precedence constraints they are copied without any trouble. This crossover is able to group together good structures in an offspring which is passed from generation to generation.

4.5 Functional unit violation

Functional unit and functional unit input assignments are also taken from the same parent. One can easily see that sometimes this will result in concurrent assignment of the same functional unit to two or more operations in the same control step. One way to resolve this situation is to include a violation term in the cost function of eq. (1). The cost function then becomes:

$$C = W_{cs} \times N_{cs} + W_{reg} \times N_{reg} + W_{bus} \times N_{bus} + W_{fu} \times N_{fu} + W_{ic} \times N_{ic} + W_{viol} \times N_{viol} \quad (4)$$

where viol refers to violation. The other way around is to reassign the functional units for

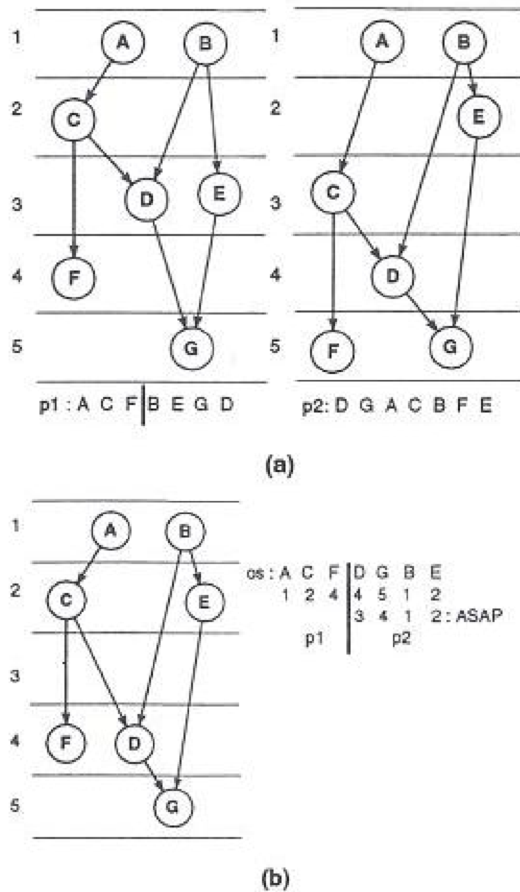


Fig. 9. Example of order crossover tailored for scheduling: (a) parents; (b) offspring.

violating operations only. The advantage that one can think of for the first scheme is that one would expect the functional unit assignment to improve genetically. But if there are too many violations then it will undermine any genetic improvement. This is indeed the case as found by experiments. Thus the second scheme looks practical and is used in our current implementation.

4.6 Normalization of functional unit assignment

Besides functional until violation there is one more problem that is to be handled when we apply the crossover operator. Suppose that the maximum number of functional units in one or both parents is three. The crossover can produce an offspring that uses only two functional units. Since we inherit the assignments from the parents unless there is a violation, it may happen that some operations are assigned to functional unit 3 whereas the maximum number of functional units used in offspring is only two. This means that one or both of the functional units are free in the control steps corresponding to those operations. Thus the functional unit assignments for these operations are performed again within the maximum range.

4.7 Mutation

Three types of mutation operators are used in the present implementation. Control step mutation is the most important type of mutation. An operation is selected randomly. An attempt is made to move it either up or down. The direction is generated randomly. If it does not result in any violation, its control step value is changed. Control step mutation has very far-reaching effects. It can produce a better schedule and reduce the number of functional units, buses and registers. The second type of mutation is the functional unit assignment mutation. An operation is selected randomly and a new functional unit number is generated. If this one is not used by any other operation in that control step then the functional unit assignment of the operation is changed to this one, otherwise another mutation attempt is made. In both cases mutation attempts are made a limited number of times. The last type of mutation is the functional unit input mutation. An operation is selected randomly and if it is a commutative operation then the assignment of its left variable to the functional unit input is changed. The last two types of mutation help in reducing the estimated number of interconnections.

4.8 Selection

Crossover is applied on the population with a specified rate. After the application of crossover is complete, we get an increased population

consisting of parents and offspring. We opted to have a fixed population size. Thus the next step is to transfer some of the individuals among parents and offspring to the next generation. This is done by a selection function based on fitness value. We create another sample space in the same manner as discussed in Section 4.3 for the increased population. Thus the selection function is the same as the choice function. This is applied as many times as population size to get the new population. It is found that good results can be obtained if this scheme is combined with one or more of the following schemes: (1) always selecting the best individual in the population; (2) selecting a specified quantity of the best individuals; and (3) selecting some specified quantity randomly. These schemes help in improving the search and maintaining the diversity in the population, which is necessary for search space exploration, and avoids premature convergence to the local optimum.

5. TS: an introduction

A general iterative technique, called tabu search (TS), was proposed by Glover [14, 15] for finding good solutions to combinatorial optimization problems. This technique is conceptually simple and elegant. It is a higher level heuristic which can be superimposed on any procedure which works by making moves to go from one trial solution to another. It has also proven itself to be very useful in providing good solutions for many NP-hard problems in a reasonable amount of time. Examples include VLSI placement [16] and circuit partitioning [17].

Like simulated annealing, TS does not resort to pure randomization to conquer intractability, nor does it take the conservative approach that a proper rate of descent will lead us to a good local optimum which may be close to a global one. TS uses a flexible attribute-based memory structure to exploit historical search information more thoroughly than by techniques using rigid memory structures (such as branch-and-bound and A* search) or by memoryless systems (such as simulated annealing). Using these memory structures, TS employs a mechanism of control which constrains and frees the search process. These correspond to tabu restrictions and aspiration criteria. TS takes the aggressive exploration approach which seeks to make the best move possible subject to available choices, performance and certain constraints.

5.1 Tabu restrictions

TS goes from one trial solution to another by making moves. It makes several candidate moves and selects the move producing the best solution among all candidate moves for current iteration. This best candidate solution may not improve the current solution. With this strategy, it is possible to reach the local optimum, ascend, and then come back to local optimum in case of a minimization problem. Thus there is a possibility of cycling back to the same state. Tabu restriction is a device to avoid such cycling by making selected attributes of these moves tabu (forbidden) to avoid move reversals. Tabu restrictions allow the search to go beyond the points of local optimality while still making the best possible move in each iteration. Selecting the best move (which may or may not improve the current solution) is based on the supposition that good moves are more likely to reach the optimal or near-optimal solutions. The set of admissible solutions achieved through different moves form a candidate list. TS selects the best solution from the candidate list. Candidate list size is a trade-off between quality and performance.

Tabu restrictions are enforced by a tabu list which stores the move attributes to avoid move reversals. Tabu list has an associated size. It can be visualized as a window on accepted moves. The moves which tend to undo moves within this window are forbidden (Fig. 10).

5.2 Aspiration criteria

The aspiration level component of TS introduces diversification in the search. It temporarily over-

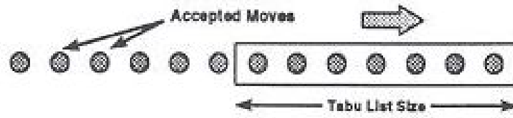


Fig. 10. Tabu list can be visualized as a window over accepted moves.

rides the tabu status if the move is sufficiently good. If a move is made tabu in iteration i and its reversal comes in iteration j, where $i < j + c$, then it is possible that the reverse move may take the search into a new region because of the effects of c intermediate moves. The aspiration criterion must make sure that reversal is leading to a solution which is better and is not the same as the previous one, otherwise cycling can occur.

The simplest aspiration criterion is to override the tabu status if the reversal produces a solution better than the best obtained thus far. Another approach is to use the same attribute of the move which is used to identify the tabu status and associate an aspiration level value with it. The reversal has to do better than this historical aspiration level. It is found to be useful in some applications to give aspiration level a tenure that parallels the tenure of the tabu list. This means that the aspiration level of the selected attribute is updated whenever that move is made tabu and whenever the aspiration level criterion is passed.

6. Scheduling and allocation using TS

TS implementation for scheduling and allocation in HLS of digital systems will be discussed in this section. As mentioned earlier, this is a transformational technique. The main tasks to formulate scheduling and allocation for TS are as follows: (1) starting with a proper initial solution; (2) defining a neighborhood for a given solution; (3) generation of moves; (4) formulation and maintenance of tabu list; (5) defining a proper aspiration level criterion; (6) finding a good tabu list size; and (7) an efficient way to accept moves. We will discuss these one by one in the remainder of this section.

6.1 Initial, current and best solution

Although in theory the initial solution can be any feasible solution, it is found that TS may take longer if given a poor initial feasible solution. We may start with the ASAP or ALAP schedule with a specified limit on the number of control steps. In both of these scheduling techniques only a few operations can be disturbed or rescheduled in the beginning of the search. Thus, it is found to be better to use either mobility-up or mobility-down scheduling [8]. These scheduling schemes have been discussed earlier. In the final implementation mobility-up scheduling is used for the initial solution. As TS proceeds we keep two solutions — one is the current solution and the other is the best solution found so far. The best solution found in n iterations is the output of TS, where n is specified by the user.

6.2 Generation of moves

Given a solution s, the generation of moves in the neighborhood of this solution is an important step in TS. The following three kinds of moves are defined for this purpose: (1) moves based on changing the control step of an operation; (2) moves based on changing the functional unit assignment; and (3) moves based on changing the functional unit input assignment of variables.

The first move is intended to optimize the number of control steps, functional units, registers and buses, whereas the last two are intended for the optimization of interconnections. Probabilities are assigned for each of these moves in accordance with the importance of each move toward optimizing the cost. Thus move type is selected probabilistically and N_{cm} moves of that type are generated, where N_{cm} is the number of candidate moves. The solutions obtained by each move are evaluated using eq. (1).

The moves are generated such that the new solutions are always feasible, but some or all of the moves may be tabu or may not pass the

aspiration criterion. In terms of first type of move, a feasible solution means that the precedence constraints are never violated. An operation is moved up or down where the direction is generated randomly. Functional unit changes are only performed if there are free functional units for the control step in which that operation is scheduled. Functional unit input changes are performed only for commutative operations.

Another way of generating neighborhood solutions is by making more than one type of move. This approach has less chances of finding the global optimum as the solution may be disturbed too much and, in fact, it might not be in the neighborhood of the present solution. Thus this approach is not used.

The algorithm proceeds as discussed in Section 4.

6.3 Tabu lists

Formulation of the tabu list is one of the main steps in mapping a problem for TS. Since we have three types of moves, the decision needs to be made whether to use one tabu list or three tabu lists. It has been suggested by Glover [18] that when the solution depends on multiple parameters it is appropriate to use more than one tabu list. Maintaining multiple tabu lists helps in generating search paths with different characteristics.

In the present implementation we used three tabu lists — one for each type of move. TS continues for the maximum number of specified iterations, n. Since separate tabu lists are maintained, we need to count how many times the particular type of move is performed. This number corresponds to the iteration number for that type of move. When the sum of iterations for three types of moves becomes n, TS stops. Attributes used for storing the control step move in the tabu list are discussed next. A two-dimensional array *csTabuList* is maintained for the tabu list. The first dimension corresponds to the total number of operations and the second dimension corresponds to the possible control steps to which an operation can be assigned. If an operation op_i is moved from (say) control step cs_i to a new control step cs_j, we store the current iteration number corresponding to the control step moves in *csTabuList* $[op_i][cs_i]$. Note that the reverse move is stored as this makes it easier to check the tabu status of future moves.

Similarly, one-dimensional arrays *fuTabuList* and *fuInpTabuList* are maintained for other types of moves. The dimension corresponds to the number of operations. If the effected operation in such moves is (say) op_j then the iteration number for that type of move is stored in *fuTabuList*$[op_j]$ or *fuInpTabuList*$[op_j]$. All these recordings of tabu status are found effective and good results are obtained.

6.4 Aspiration level criteria

After all the candidate moves of a particular type are generated for iteration itr for that kind of move, the best of these is selected, which may not be better than the current solution. If it is not tabu, it is accepted. The accepted move becomes the current solution for the next iteration. The tabu list size (T_{size}) is an important parameter in TS. In the present implementation the magic number seven is used for T_{size} and is the same for all three lists. Since we store the iteration number in order to check the tabu status, a move is tabu if the difference of itr and stored iteration number is less than or equal to T_{size}. If it is tabu, its aspiration level is checked as described below.

A common aspiration level (AL) criterion is used for all the three moves. ALs are associated with each of the operations and are initialized to infinity. If a move m_i affects an operation op_i and m_i is tabu (forbidden), the AL(op_i) value is checked against $c(m_i)$, the cost of the solution achieved by move m_i. If $c(m_i) < \text{AL}(op_i)$, the move is accepted and AL(op_i) is set to $c(m_i) - 1$. Otherwise another set of the same type of candidate moves

is generated. A maximum limit on regenerations is specified after which a new type is selected for candidate moves. Note that the AL of an operation is updated only when it overrides the tabu status of that operation. Thus aspiration level is not the best historical value. It may allow one to go to a previous solution reached by a tabu move, but this can happen only once. Thus cycling is avoided. This aspiration level criterion gives more freedom to explore the search space and is found to be effective for scheduling and allocation.

6.5 Alternative implementation

In an alternative implementation we tested with separate tabu list sizes for each type of move. Two AL criteria were used — one for the control step moves and the second for other types of moves. For control step moves there is a separate AL for each operation for each of its possible control steps. The AL corresponding to an operation for a particular control step is one less than the cost of the solution obtained when that operation was last assigned to that control step. It is updated each time a move is accepted and is thus not a historical value. It serves to override tabu status to explore new search paths. The AL for an operation to the other two moves is one less than the interconnection cost obtained when one of these moves was last applied to that operation. The value of interconnection cost is used for the AL because these two moves are intended to optimize interconnection cost. In this implementation a candidate list is prepared and consists of solutions reached through non-tabu moves of the same type or, if tabu, they passed the corresponding aspiration criterion. The best among these is selected. The candidate list size is kept between five and ten.

Although both implementations were able to find good solutions, it should be noted that aspiration criteria are more strict in the second implementation than the first one. The first implementation is not strict in selecting moves and the aspiration criterion is easy to pass. Because of the use of the candidate list strategy the second implementation is able to find good solutions more quickly than the first. An instance where the second implementation has achieved a better result than the first implementation is that of the discrete cosine transform benchmark for seven control step limits with a pipelined multiplier option.

7. Data path synthesis using GA

Interconnection of registers, buses, multiplexers and ALUs is called data path and the process of forming such an interconnection is called data path synthesis (DPS). Allocation using a GA as described in previous sections has only done part of the actual allocation. Functional units are allocated to operations, and variables involved in the operation are assigned to the functional unit inputs. Numbers of registers, buses and interconnections are optimized in an attempt to solve scheduling and allocation as a combined problem. The minimum number of registers and buses for the given schedule is known. The interconnection cost is only optimized. We still do not know the exact data path. Mappings of variable to register and data transfer to bus still need to be done. Both these mappings have a profound effect on the interconnection cost. Different mappings will give different interconnection costs and this can also be formulated as an optimization problem. This is the subject of this section.

After we map the variables to registers and data transfers to buses, we have the following information at our disposal about the high-level description from which we started.

- An operation is scheduled during which control step.
- An operation is performed on which functional unit.

- A variable goes to which input of the functional unit.
- A variable is stored in which register during any specific control step.
- A data transfer is performed on which bus.

Once we know all this information we can easily generate the data path for the given high-level description.

The architecture used for optimizing the number of interconnections is shown in Fig. 11. Outputs of functional units and registers are connected to buses. Multiplexers are provided at the inputs of functional units and registers if the input comes from more than one bus. A direct connection is provided in case the input comes from only one bus. The interconnection cost is estimated by number of multiplexer inputs. In the remainder of this section the formulation of the problem for the genetic algorithm is described [8]. As mentioned earlier, two mappings are to be performed. A chromosome needs to be devised that can incorporate both these mappings. The fitness of each individual is based on finding the exact number of interconnections, which involves calculating the total number of multiplexer inputs. A suitable initial population has to be created. Choice and selection functions are somewhat similar, as discussed in Sections 4.3 and 4.8. A crossover operator which produces valid mappings is to be found. A suitable mutation function is required. In the following sections we address these problems.

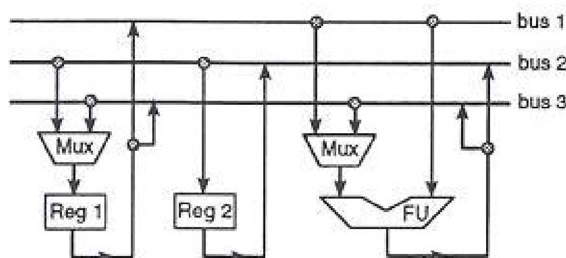


Fig. 11. Architecture on which the data path is mapped.

7.1 Initial population

Since we have to perform two types of mapping, the chromosome has to incorporate both these mappings. Thus the chromosome has two parts — one for the variable to register mapping and the other for data transfer to bus mapping.

7.1.1 Chromosome part for variable to register mapping

Given the lifetime analysis of the variables, the left-edge algorithm [19] can be used to map all variables to registers optimally for the given schedule. (It should be mentioned here that if a variable is regenerated then multiple lifetimes are kept for each regeneration as it may help in optimizing the number of interconnections.) One problem with the left-edge algorithm is that it does not take the interconnection cost into consideration while grouping variables. This is because the left-edge algorithm considers left edges in the sorted order. This problem is illustrated in Fig. 12. Lifetimes are shown in Fig. 12a. Assume that sorted order is (v_1, v_2, v_3, v_4). The left-edge algorithm will give the grouping of Fig. 12b. Another grouping is given in Fig. 12c. It may happen that the grouping of Fig. 12c may result in fewer interconnections than the grouping of Fig. 12b. Thus there is a possibility of interconnection cost optimization by considering various optimal (in terms of number of registers) groupings.

We can utilize this fact to create an initial population for this part of the chromosome. Given

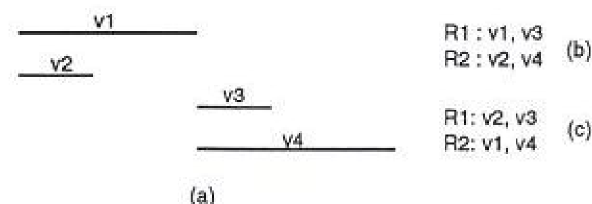


Fig. 12. Grouping variables into registers.

the lifetime analysis we can perturb the order of segments that starts from the same control step. This will not affect the sorted order of the left edges. Applying the left-edge algorithm on this configuration one can get a different grouping. In this way, the initial population for this part of the chromosome is created.

7.1.2 Chromosome part for data transfer to bus mapping

A list of data transfers for the scheduled CDFG can be made using depth first search. All data transfers during control step (say) i can take place on any available bus, but only one data transfer can take place on any bus at one time. As mentioned earlier, different mappings will give a different number of interconnections. An easy way to make a chromosome out of this situation is to think of any bus as consisting of segments for each of the control steps as shown in Fig. 13. Data transfers can be assigned to segments or slots. A list of data transfers is made and the bus chromosome is filled with the particular index to the data transfer list. Some slots will remain empty, meaning that there is no data transfer on that bus during that particular control step.

To create the initial population a sample chromosome is prepared. This can be done by noting that data transfers can be assigned to buses by using the left-edge algorithm. Data transfers in each column of the sample chromosome (Fig. 14) can be interchanged randomly to generate an initial population.

7.1.3 Complete chromosome

The complete chromosome is shown in Fig. 15. The left part is the bus chromosome and the right part is the register chromosome. One can think of the register chromosome as hooked to the bus chromosome. Crossover is only applied to the bus chromosome whereas mutation is applied on both bus and register chromosomes. One should think of the bus chromosome as if all buses are put next to each other one after another.

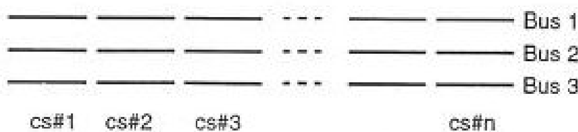


Fig. 13. Structure of the bus chromosome.

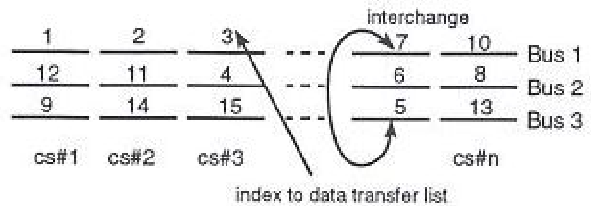


Fig. 14. Sample bus chromosome.

Fig. 15. Complete chromosome.

7.2 Fitness calculation

The first step in calculating the fitness value for each individual is to calculate the number of multiplexer/bus inputs. We keep the following information for each data transfer:

(1) Functional unit involved.

(2) The functional unit input to which it goes.

(3) The type of transfer (input or output).

(4) The control step in which it takes place.

(5) Register in which the variables involved are stored or to be stored.

(6) Bus on which it will take place.

Given this information, one can calculate the number of multiplexer/bus inputs required. Since this is also a minimization problem, the number of multiplexer inputs is subtracted by a specified maximum number of interconnections.

The resulting numbers for different individuals will not be far apart, so they are multiplied by a large number to create some difference between them. Following this, sigma truncation and linear scaling is applied as usual to obtain the final fitness value. The sample space is created and choice and selection functions take the same form as discussed in Section 4.3.

7.3 Crossover

As mentioned earlier, crossover is only applied to the bus part of the chromosome. A necessary property for the crossover operation is that the data transfers should not change their control step during crossover operation. With reference to Fig. 14, it means that they should remain in the same column. Another required property for the crossover is that the data transfers should not be duplicated or missed.

Different crossovers were considered that failed to satisfy one or both of these properties. Simple one or two point crossover will change the data transfer control steps as well as duplicate them. Order crossover or partially mapped crossover (PMX) [12] will not duplicate data transfers but will change the control steps.

It is seen that cycle crossover has an interesting property that can be utilized here. In cycle crossover an offspring inherits genes from one parent or the other in the same position as the corresponding parent. An example of a cycle crossover is illustrated in Fig. 16. There are two cycles: 3-1-6-3 in parent 1 and 4-2-5-4 in parent 2. We randomly start with parent 1. During the first cycle, offspring 1 get genes 3, 1 and 6, and offspring 2 gets genes 1, 6 and 3. For the second cycle we start with parent 2. During this cycle, offspring 1 gets genes 4, 2 and 5, and offspring 2 gets genes 2, 5 and 4. Note that the net effect of the second cycle is to swap the genes in the two parents as they are passed to offspring.

Now, consider the two parent bus chromosomes shown in Fig. 17. Assume that there are four control steps and thus there are three buses. The non-negative numbers are indices to the data transfer list and −1s indicates that there are no data transfers in those control steps. One would notice that we can not apply cycle crossover directly on this bus chromosome. The reason is that the gene's values (alleles) are not distinct. In order to have a bus chromosome on which cycle crossover can be applied, we fill the segments having no data transfer (in the sample chromosome) with the numbers greater than the total number of data transfers. In Fig. 17, there are eight data transfers. Thus −1s positions are filled with numbers 9, 10, 11 and 12 as shown in Fig. 18a. After cycle crossover is applied the resulting offspring are shown in Fig. 18b.

Parent 1:	3	2	1	5	4	6
Parent 2:	1	4	6	2	5	3
Offspring 1:	3	4	1	2	5	6
Offspring 2:	1	2	6	5	4	3

Cycle 1: 3 - 1 - 6 - 3 (in parent 1)
Cycle 2: 4 - 2 - 5 - 4 (in parent 2)

Fig. 16. Cycle crossover example.

3	-1	8	6	-1	4	1	5	7	-1	2	-1
6	7	-1	1	8	-1	2	-1	5	3	-1	4

Fig. 17. An example of a bus chromosome.

7.4 Final data path

As mentioned in Section 7.2, we store enough information for each data transfer that can be used to find the number of multiplexer inputs. Using the same information one can easily generate the final data path for the high-level description. This information is as follows:

- Functional unit number for each operation.
- Functional unit input number for each input variable.

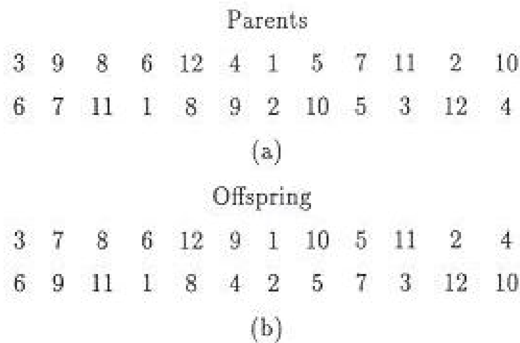


Fig. 18. (a) Bus chromosome suitable for cycle crossover. (b) Offspring resulting from cycle crossover.

- Variable to register mapping for each control step.
- Data transfer to bus mapping for each data transfer.

Multiplexers are provided only if there are multiple inputs coming to the two inputs of a functional unit or the input of a register. A direct interconnection is implied if there is only one input to a bus.

8. Experimental results

GSA and TSA are tested on various benchmarks. Table 1 shows the results for the differential equation benchmark. Table 2 shows the results for a more complicated fifth order elliptic wave filter (EWF) benchmark. The STAR system [20] uses parallel data transfers, so that the bus comparison with this system is of little significance. The results shown for 17_{LU} control steps are for loop unfolding. Table 3 shows the results obtained for the discrete cosine transform (DCT) benchmark. Two results of TSA for seven control steps with pipelined multipliers option correspond to the results obtained from two implementations of TSA discussed earlier. The results are compared with simulated evolution (SE) [9], the HAL system [4], SALSA II [6], the STAR system [20], the EMUCS system [3], and the CATREE system [2]. Comparisons are given for number of control steps (CS), adders (+), multipliers (*), functional units capable of performing addition and subtraction (+/−), registers (Reg), and multiplexers or buses (Mx). The cost column indicates the cost achieved by TSA and GSA. The costs of control steps, adders, multipliers, registers, multiplexers and interconnections are derived from [5]. The p in the (*) column stands for pipelined multiplier. It is assumed that addition takes one control step whereas multiplication takes two control steps. The cost column is included for the sake of comparison between GSA and TSA. The cost for other systems can not be computed because the interconnection cost is not known. The data path synthesis result using the GA for differential equation benchmark is shown in Table 4. The result is comparable with the best known systems.

TABLE 1 Differential equation results

System	CS	ALU	*	Reg	Mx	Cost
GSA	8	1	1p	5	4	1710
TSA	8	1	1p	5	4	1710
HAL	8	1	1p	5	4	-
SE	8	1	1p	5	5	-

9. Conclusions

The GA is a promising optimization technique. This work presents its application to scheduling and allocation in HLS. The work involves finding an appropriate string encoding or chromosomal representation. The initial population of solutions is constructed to get better results. Two scheduling techniques, mobility-up and mobility-down, are used for this purpose. Two new crossover operators (alternate crossover and order crossover for scheduling) are presented which can find

TABLE 2 EWF results

System	CS	+	*	Reg	Mx	Cost
GSA	17	3	3	11	10	4786
TSA	17	3	3	11	10	4804
SE	17	3	3	11	11	-
HAL	17	3	3	-	-	-
SALSA II	17	3	3	-	-	-
GSA	17	3	2p	11	10	3986
TSA	17	3	2p	11	10	4006
SE	17	3	2p	11	12	-
HAL	17	3	2p	-	-	-
CATREE	17	3	2p	12	-	-
SALSA II	17	3	2p	-	-	-
GSA	17_{LU}	2	1p	10	8	2638
TSA	17_{LU}	2	1p	10	8	2638
STAR	17_{LU}	2	1p	11	5*	-
GSA	18	2	2	11	8	3484
TSA	18	2	2	10	8	3424
SE	18	2	2	10	9	-
SALSA II	18	3	2	-	-	-
GSA	19	2	2	10	7	3370
TSA	19	2	2	10	7	3340
SE	19	2	2	10	11	-
HAL	19	2	2	12	-	-
EMUCS	19	2	2	12	12	-
GSA	19	2	1p	11	8	2644
TSA	19	2	1p	10	7	2558
SE	19	2	1p	11	9	-
HAL	19	2	1p	12	6	-
STAR	19	2	1p	11	4*	-
SALSA II	19	2	1p	-	-	-

TABLE 3 DCT results

System	CS	+/-	*	Reg	Mx	Cost
GSA	7	6	8	19	18	10894
TSA	7	6	8	19	18	10904
SALSA II	7	6	8	-	-	-
GSA	7	8	4p	21	21	8788
TSA_1	7	8	4p	21	21	8824
TSA_2	7	6	5p	19	17	8442
SALSA II	7	8	4p	-	-	-
GSA	8	5	6	15	17	8712
TSA	8	5	6	15	16	8660
SALSA II	8	5	6	-	-	-
GSA	8	5	4p	17	15	7030
TSA	8	5	4p	16	16	7110
SALSA II	8	5	4p	-	-	-
GSA	9	4	6	15	14	8076
TSA	9	4	6	14	15	8158
SALSA II	9	4	6	-	-	-
GSA	9	4	3p	13	14	5626
TSA	9	4	3p	13	14	5660
SALSA II	9	4	3p	-	-	-

TABLE 4 Data path synthesis results for differential equation benchmark

System	Mux Inputs
DPS	14
Splicer	16
HAL	13
SE	12

applications in many other areas. The GSA approach is different from a previous attempt using GA [7] in many respects. The contributions include: a new chromosomal representation for scheduling and two subproblems of allocation; and two novel crossover operators to generate legal schedules.

TS is another promising optimization technique. This paper presents its application to scheduling and allocation in HLS. Investigations are done to find a good initial solution to start with, to define a neighborhood for a given solution, generation of moves, formulation and maintenance of tabu list(s), defining a proper aspiration level criterion, finding a good tabu list size and an efficient way to accept moves. Two implementations of TS are reported and compared.

GSA and TSA are tested on three benchmark circuits, namely differential equation, fifth order elliptic wave filter, and DCT. The results obtained are comparable with those obtained by other systems.

A novel interconnect optimization approach using the GA is also reported in this research. It can be used to optimize a number of interconnections for a given schedule and functional unit allocation. It tries to find genetically good mappings for variables to registers and data transfers to buses with the aim of optimizing interconnection.

Future work will focus on designing a complete data path synthesis system using GA. Efforts will be directed to include facilities such as chaining and loop winding. The data path synthesis should be able to take high-level description and produce register-transfer level description of the circuit. Research can also be directed to finding more effective implementation for TS and designing a complete data path synthesis system using TS with the above-mentioned facilities.

Acknowledgements

The authors wish to acknowledge King Fahd University of Petroleum and Minerals for support under project No. COE/DESIGN/145.

References

[1] M.C. McFarland, A.C. Parker and R. Camposano, The high level synthesis of digital systems, *Proc. IEEE*, 78(2) (Feb. 1990) 301–318.

[2] C.H. Gebotys and M.I. Elmasry, A VLSI methodology with testability constraints, in *Proc. 1987 Canadian Conf. on VLSI*, Winnipeg, Oct. 1987.

[3] C.Y. Hitchcock and D.E. Thomas, A method of automatic data path synthesis, in *Proc. 20th Design Automation Conf.*, June 1983, pp. 484–489.

[4] P.G. Paulin and J.P. Knight, Force-directed scheduling for the behavioral synthesis of ASICs, *IEEE Trans. Computer-Aided Design*, 8(6) (June 1989) 661–679.

[5] S. Devadas and A. Richard Newton, Algorithms for hardware allocation in data path synthesis, *IEEE Trans. Computer-Aided Design*, 8(6) (1989) 768–781.

[6] M.R. Rhinehart and J. Nestor, SALSA II: a fast transformational scheduler for high-level synthesis, in *1993 IEEE Int. Symp. on Circuits and Systems*, 1993, pp. 1678–1681.

[7] N. Wehn, M. Glesner and M. Held, A novel scheduling/allocation approach for datapath synthesis based on genetic paradigms, in *IFIP Working Conference on Logic and Architecture Synthesis*, Paris, 1990, pp. 47–56.

[8] S. Ali, Scheduling and allocation in high-level synthesis using genetic algorithm, Master Thesis, King Fahd University of Petroleum and Minerals, 1994.

[9] T.A. Ly and J.T. Mowchenko, Applying simulated evolution to high level synthesis, *IEEE Trans. Computer-Aided Design*, 12(3) (March 1993) 389–409.

[10] S. Ali, S.M. Sait and M.S.T. Benten, GSA: scheduling and allocation using genetic algorithm, in *European Design Automation Conference – Euro-DAC'94*, Grenoble, France, Oct. 1994, pp. 84–89.

[11] S. Ali, S.M. Sait and M.S.T. Benten, TSA: scheduling and allocation using tabu search, in *Int. Conf. on Electronics, Circuits and Systems – ICECS'94*, Cairo, Egypt, Dec. 1994, pp. 423–428.

[12] D.E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, USA, 1989.

[13] S. Davidson, D. Landskov, B.D. Shriver and P.W. Mallet, Some experiments in local microcode compaction for horizontal machines, *IEEE Trans. Computer-Aided Design*, 30(7) (1981) 460–477.

[14] F. Glover, Tabu Search — Part I, *ORSA J. Comput.*, 1 (1989) 190–206.

[15] F. Glover, Tabu Search — Part II, *ORSA J. Comput.*, 2 (1990) 4–32.

[16] L. Song and A. Vannelli, VLSI placement using tabu search, *Microelectronics J.*, 17(5) (1992) 437–445.

[17] S. Areibi and A. Vannelli, Circuit partitioning using a tabu search approach, in *1993 IEEE Int. Symp. on Circuits and Systems*, 1993, pp. 1643–1646.

[18] F. Glover, Artificial intelligence, heuristic frameworks and tabu search, *Managerial and Decision Economics*, 11 (1990) 365–375.

[19] A. Hashimoto and J. Stevens, Wire routing by optimizing channel assignment within large apertures, in *Proc. 8th D. A. Workshop*, Las Vegas, 1971, pp. 155–169.

[20] Fur-Shing Tsai and Yu-Chin Hsu, STAR: an automatic data path allocator, *IEEE Trans. Computer-Aided Design*, 11(9) (Sept. 1992) 1053–1064.