CHAPTER IV

DESIGN AND IMPLEMENTATION

The applications of GA to the high level synthesis were previously reported by many researchers [7,8,14]. These applications concentrate on the problem of functional unit scheduling and assignment to minimize the number of macro functional units formed as ALUs and the number of control steps. However no application has been done on combining the problem of functional unit scheduling and assignment with the problem of checkpoint insertion. Here we combine the problem studied in [7,8,14] with the problem proposed in [2]. We also include chaining and ALU delay as well as various ALU areas into consideration.

The synthesis algorithm consists of three main steps. The first step is to transform a behavioral description into a control-data flow graph (CDFG). The second step is to encode the CDFG into genes. The final step is to perform the evolution process on the set of generated genes. Among these steps, the second step is the most difficult one. In the last step, only two genetic operations, mutation and crossover, are considered. However the classical mutation and crossover operations [13] cannot be applied to this synthesis problem. New mutation and crossover operations suitable to this synthesis problem are introduced.

In detail, our approach is shown as Figure 4.1. The algorithm starts reading three inputs—the data-flow graph (DFG), the module library and the constraint requirements—and GA parameters which consists of the following:

- population size (*popsize*),
- newly created population (*replacepop*),
- probability of crossover (*probcross*),
- probability of mutation (*probmutate*), and
- conditions to perform evolution process which comprise the maximum number of generations denoted by *iteration*, the maximum similar population denoted by *maxsimilar*, and the maximum of generations that no change of the best fitness value of the population appears denoted by *iternochange*.

We have performed experiments on a set of problems with the following information; *popsize* = 100-500, *replacepop* = 60-80% of *popsize*, *probcross* = 0.65, *probmutate* = 0.05-0.1, *iteration* = 500, *maximilar* = 95 % of *popsize* and *iternochange* = 50-100. Then, we estimate the size of gene which will be discussed in section 4.1. Next we generate initial population by random and calculate their fitness value. The process is repeated until at least one of the conditions mentioned above is met.

As illustrated in Figure 4.2, we sort the population by their fitness values in decending order. Then we place in the new population the top of the sorted population equal to *popsize* minus *replacepop* (see in the Figure 4.2). The rest of new population is produced by crossover and/or mutation.

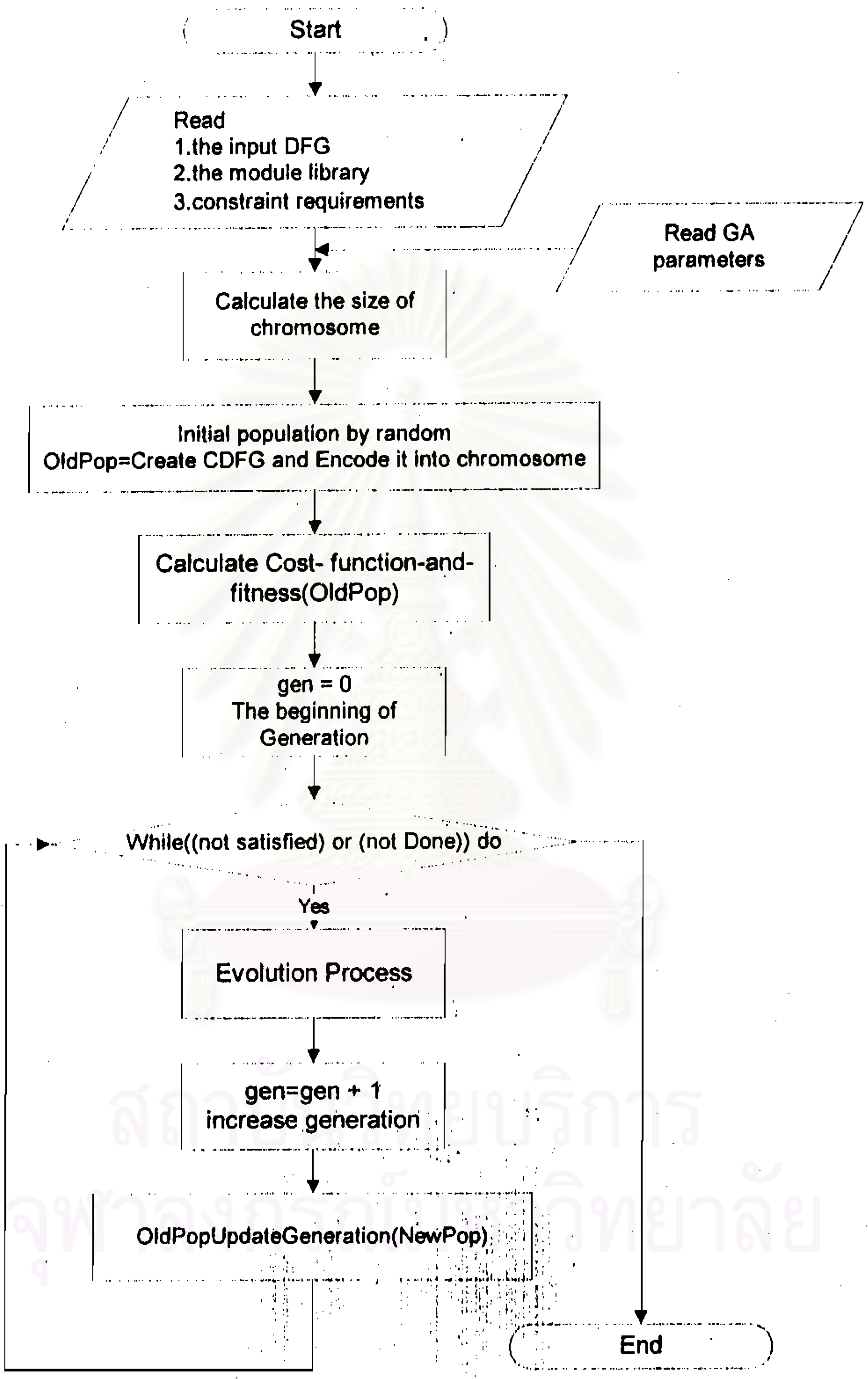

Start
Read
1.the input DFG
2.the module library
3.constraint requirements
Read GA parameters
Calculate the size of chromosome
Initial population by random
OldPop=Create CDFG and Encode it into chromosome
Calculate Cost- function-and-fitness(OldPop)
gen = 0
The beginning of Generation
While((not satisfied) or (not Done)) do
Yes
Evolution Process
gen=gen + 1
increase generation
OldPopUpdateGeneration(NewPop)
End

Figure 4.1 The flowchart of genetic algorithm process

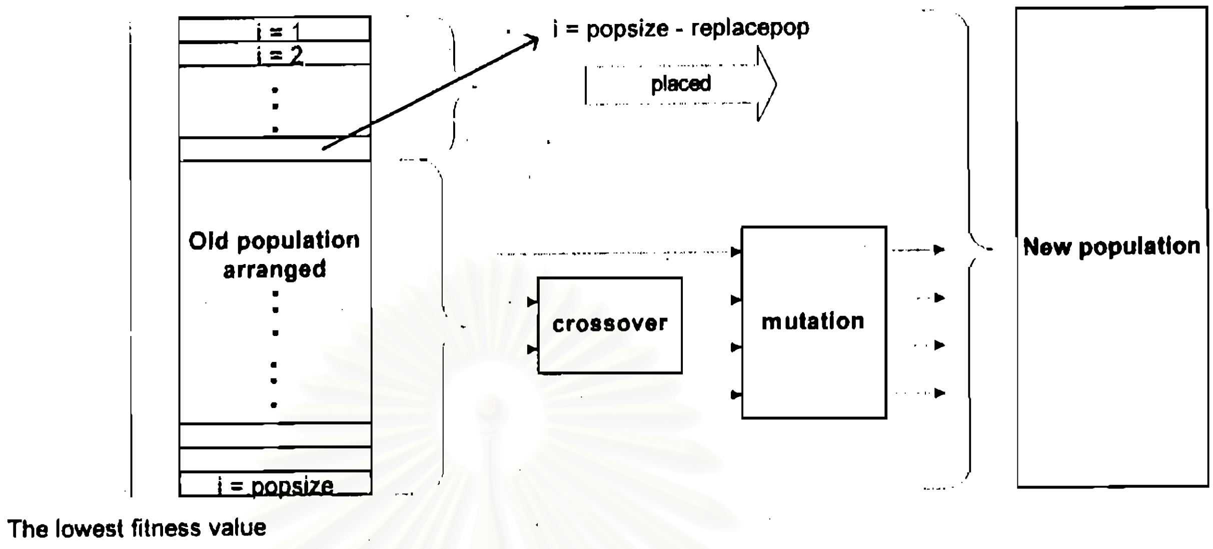


Figure 4.2 Evolution process

4.1. Encoding Scheme

The CDFG consists of a set of vertices representing functional units. Throughout the scheduling, it is noticed that the number of functional units in the CDFG never changes. Only the location of each functional unit may change. The number of functional units may be decreased after the functional unit assignment either in the form of a set of ALUs or by functional unit sharing. However this assignment will not alter the number of vertices in the CDFG. Therefore, there are three parts of information that we must retain. The first part is the location of each functional unit. The second part is the location of each checkpoint. The third part is the connection of vertices in CDFG. We call the first part ***resource information***, the second part ***rollback information***, and the third part ***data dependency information***. The resource information part consists of the following information.

1. Types of functional units. Generally, they are adder, multiplier, subtracter, divider, and ALU (Arithmetic Logic Unit). ALU can sometimes be used to replace a set of other functional units to reduce the area.
2. Delay times of functional units. The delay times are measured in the term of numbers of control steps.
3. Minimum and maximum numbers of control steps. The minimum number of control steps can be obtained by applying as-soon-as possible scheduling while the maximum number of control steps can be obtained by applying as-last-as possible scheduling.
4. Width and height of a given CDFG. The width and the height of a CDFG are defined in the following definitions.

Definition 1 The maximum height, H, of a CDFG is equal to the number of control steps obtained after applying the as-last-as possible (ALAP) scheduling to the CDFG.

Definition 2 Let F_i be the number of functional units in control step ith obtained by applying the as-soon-as-possible (ASAP) to the given CDFG. The maximum width, W, of a CDFG is equal to

$$W = \max_i(F_i)$$

As an example in Figure 4.3, F_1 is equal to five, the number of functional units in 1st control step. Therefore, the maximum width is five, $W = 5$.

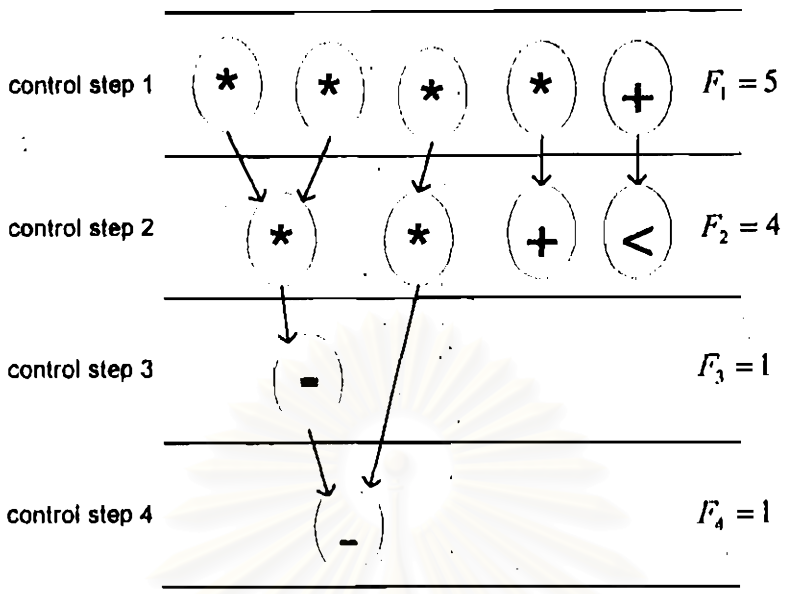


Figure 4.3 An example of as-soon-as-possible scheduling

The resource information concerns only the location of each checkpoint and the number of checkpoints which are generated by the genetic algorithm. The data dependency information expresses the dependency of each functional unit and its residing control step. Each functional unit is named by using an integer number. We use two 2-dimensional arrays to capture the encoded genes. The height of the array is set to the maximum height *H* and the width of the array is set to the maximum width *W* of the given CDFG. The first array stores the data dependency information and the second array stores the resource and rollback information. We call the first array ***data dependency array*** (DDA) and the second array ***resource and rollback array*** (RRA). In the DDA, each row corresponds to each control step while each column of the RRA corresponds to the assigned location of each functional unit in its residing control step. The value of each entry *(i,j)* of the DDA is set to the name of the functional unit assigned to this row *i* and column *j*. Similarly, each row of the RRA corresponds to each control step while each column of RRA corresponds to the assigned location of each functional unit in its residing control step. On the contrary, the value of each entry (*i,j*) is set to the type name assigned to the functional unit in its residing at the entry (*i,j*). In both arrays, we attach one additional column to store the location of each checkpoint.

Table 4.1 Module library

Operation	Type Name	Delay (number of cs)	Area
multiplier	1	2	64
	2	4	8
adder/substracter	3	1	16
	4	2	4

To understand this encoding scheme, let us consider an example whose control-data flow graph is shown in Figure 4.4. There are two subgraphs and five functional units (two subtracters, two multipliers, and one adder). Figure 4.4(a) shows the CDFG where each functional unit is labeled by its operation and Figure 4.4(b) shows the CDFG after each functional unit is named by a number. The information concerning the type names, delay times, and areas of each functional unit are assumed in the Table 4.1. We call this table Module Library.

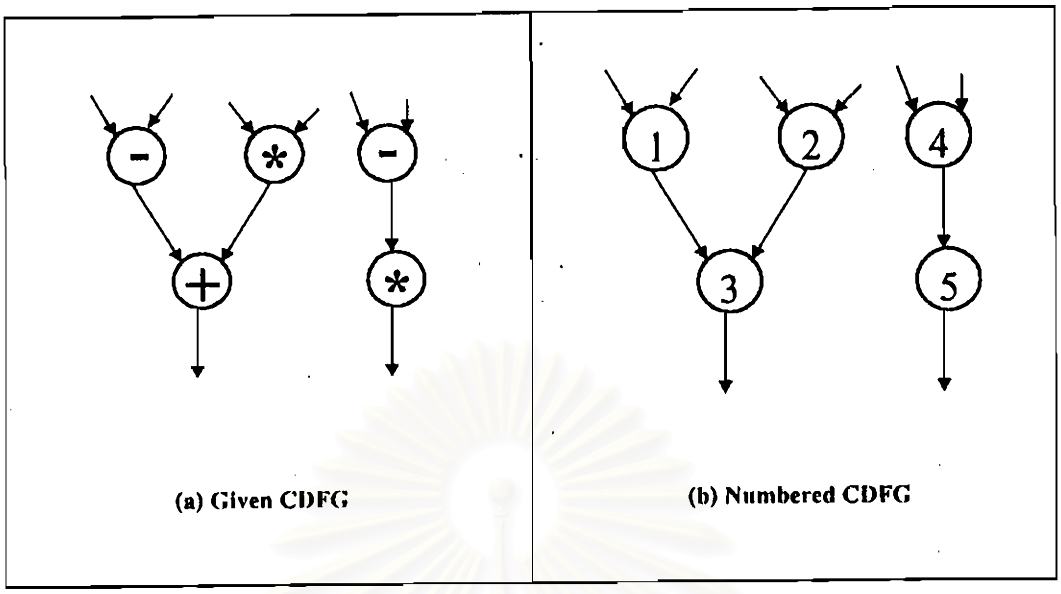


Figure 4.4 an example of a CDFG

Suppose that six control steps are used. Functional units 1, 2 and 4 are scheduled in one control step and functional units 3 and 5 are scheduled in another control step. Subtracters 1 and 4 are implemented by subtracters of type 4 with two unit delay times. Multiplier 2 is implemented by a multiplier of type 1 with two unit delay times. Adder 3 is implemented by an adder of type 3 with one unit delay time. Multiplier 5 is implemented by a multiplier of type 2 with four unit delay times. Figure 4.5 shows the CDFG with this implementation. Each number in Figure 4.5 denotes the functional unit name as given in Figure 4.4(b). The information in Figure 4.5 is encoded into a gene by using a DDA and a RRA as shown in Figure 4.6(a) and 4.6(b), respectively.

In Figure 4.6(a), there are six rows and four columns. The leftmost column is the first column. The numbers appearing in columns 1 to 3 refer to the names of the functional units as denoted in Figure 4.5. Number 0 means that no functional unit exists. The control step having a checkpoint inserted is indicated by number 1 at the corresponding row in column four. Here, the checkpoints are inserted at control steps 2 and 6. In Figure 4.6(b), the numbers in columns 1 to 3 denote the type names of the functional units shown in Figure 4.5. Number 0 means there is no functional unit at that location. The rightmost column captures the control step having a checkpoint indicated by number 1's. Number 0 in this column means there is no checkpoint.

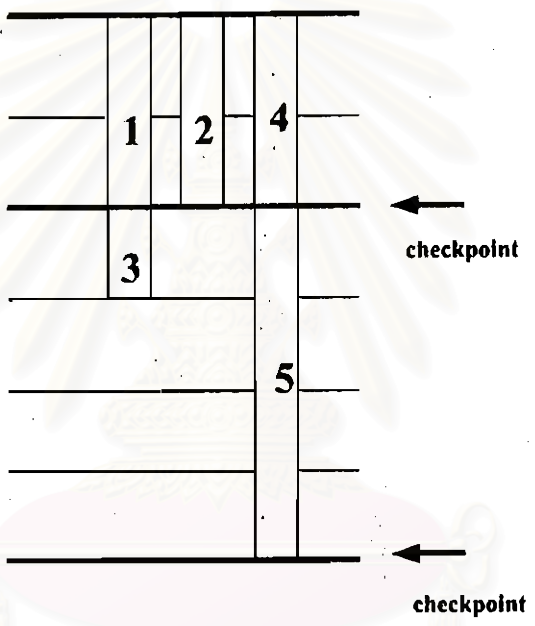


Figure 4.5 The control steps, the delay time of each functional unit, and the checkpoints of the CDFG

1	2	4	0
0	0	0	1
3	0	5	0
0	0	0	0
0	0	0	0
0	0	0	1

(a)

4	1	4	0
4	1	4	1
3	0	2	0
0	0	2	0
0	0	2	0
0	0	2	1

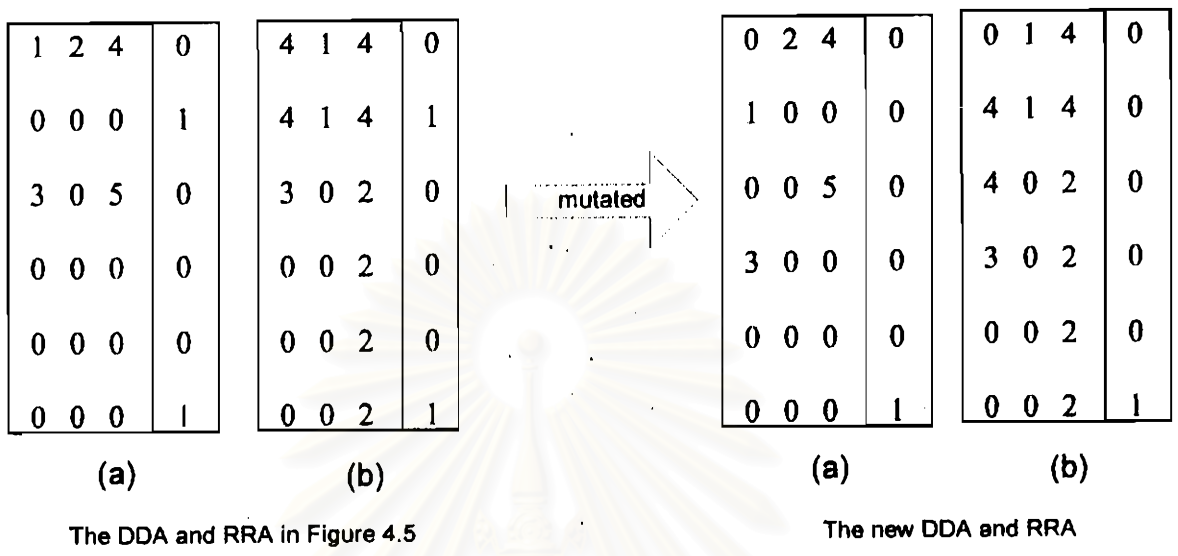
(b)

Figure 4.6 The data dependency array (DDA) of the CDFG in (a) and the resource and rollback array (RRA) of the CDFG in (b)

4.1. Mutation

Mutation can be performed on both data dependency array (DDA) and resource and rollback array (RRA). The mutations of both DDA and RRA must be collaborated. For the DDA, an encoded gene is mutated by sliding the functional unit names either up or down. The sliding is valid if it does not violate the original data dependency of the CDFG. Once the functional unit is slid either up or down in the DDA, its type name stored in the RRA must also be slid accordingly to the appropriate control step to preserve the data dependency and the actual delay time.

For example, suppose that the functional unit 1 is slid down from control step 1 to 2 in DDA. Since this unit has two unit delay times its descendent functional unit 3 must obviously be slid down from control step 3 to 4 in DDA. After the mutation is performed in DDA, the information in RRA must be updated. Figure 4.7 shows the mutated DDA and RRA of the gene in Figure 4.6, respectively.



1 2 4	0	4 1 4	0
0 0 0	1	4 1 4	1
3 0 5	0	3 0 2	0
0 0 0	0	0 0 2	0
0 0 0	0	0 0 2	0
0 0 0	1	0 0 2	1

0 2 4	0	0 1 4	0
1 0 0	0	4 1 4	0
0 0 5	0	4 0 2	0
3 0 0	0	3 0 2	0
0 0 0	0	0 0 2	0
0 0 0	1	0 0 2	1

Figure 4.7 An example of (a) mutated DDA and (b) RRA

For the RRA, the mutation can be performed by changing the type name of some functional unit. The new type name implies that the new area and the new delay time are assigned to the functional unit. In addition, the location of each checkpoint can also be changed to generate a new solution. However, not every control step can be a candidate location for a checkpoint. A checkpoint can only be inserted at the starting or ending control step of any functional unit. If there exist some functional units with multiple control step delay then a checkpoint cannot be inserted in between these control steps. Thus, after the mutation, some checkpoints must be properly relocated. In Figure 4.7, the checkpoint originally at control step 2 is mutated by setting the entry (2,4) of the RRA to 0.

4.2. Crossover Operation

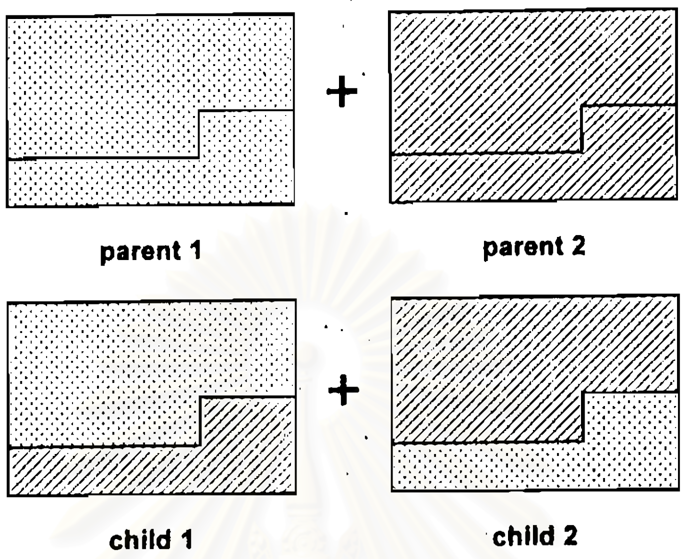
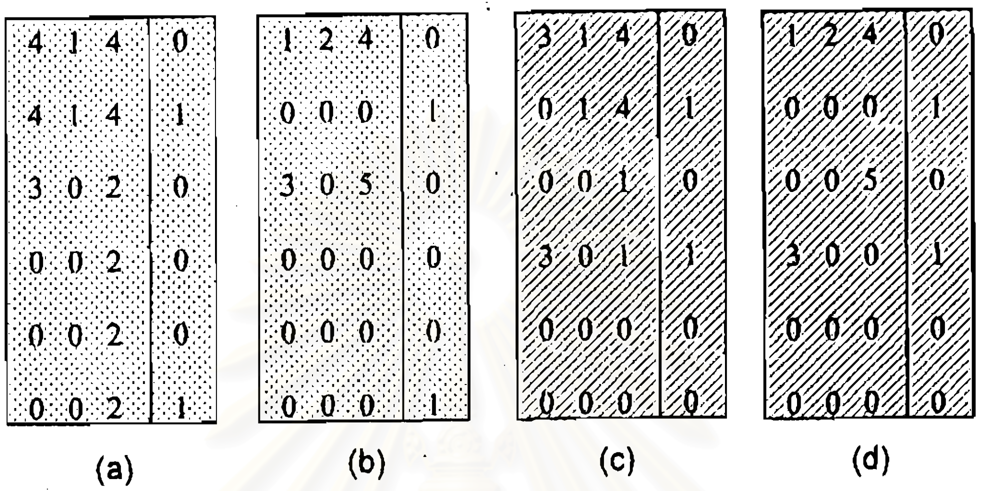


Figure 4.8 Crossover operation

This operation requires two parents. In our synthesis problem, we cannot directly apply the crossover operation suggested in [8,13,14]. In other words, our crossover operation is constrained by the data dependency of the CDFG and the multiple control step delay time of each functional unit. However, we still use the concept of a cutting line to cut each parent gene into two parts prior to the crossover operation. This crossing line is not a straight line as stated in [13]. We use a zigzag line to cut through the DDA and RRA under the condition that a cutting line cannot cut through a functional unit with multiple control step delay time. It can only cut through a functional unit at the starting or ending control step of that unit. After cutting, two parent genes exchange their DDA and RRA parts at the location guided by the cutting line. This exchanging process sometimes cannot be freely achieved as in the classical crossover operation because of the data dependency structures of two different CDFG.

Figure 4.8 shows an example of how a crossover operation for this synthesis is performed.



(a)

4	1	4	0
4	1	4	1
3	0	2	0
0	0	2	0
0	0	2	0
0	0	2	1

(b)

1	2	4	0
0	0	0	1
3	0	5	0
0	0	0	0
0	0	0	0
0	0	0	1

(c)

3	1	4	0
0	1	4	1
0	0	1	0
3	0	1	1
0	0	0	0
0	0	0	0

(d)

1	2	4	0
0	0	0	1
0	0	5	0
3	0	0	1
0	0	0	0
0	0	0	0

Figure 4.9 (a) RRA of parent 1, (b) DDA of parent 1, (c) RRA of parent 2 and (d) DDA of parent 2

Suppose we have two parents as shown in Figure 4.9. The gene of each parent is represented by two arrays, RRA and DDA. After the crossover operation, we obtain two children as shown in Figure 4.10.

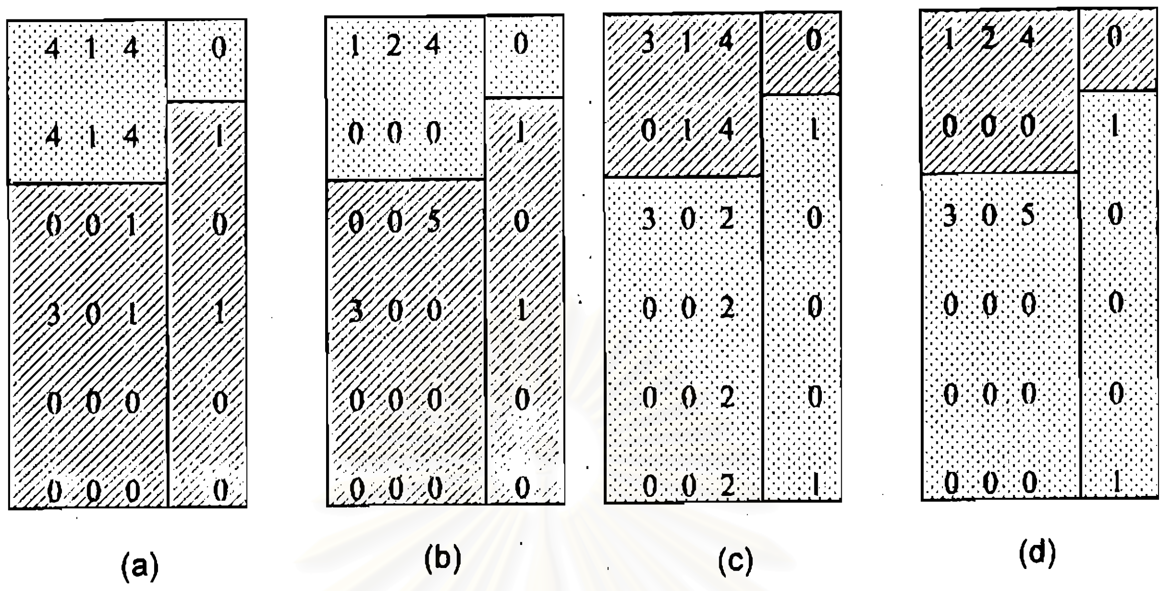


Figure 4.10 (a) RRA of child 1, (b) DDA of child 1, (c) RRA of child 2 and (d) DDA of child 2

4.4. Cost and Fitness Function

The cost function is the principle issue as it reflects the goal of optimization. The purpose of the simultaneous scheduling, allocation, module selection, and checkpoint insertion is to optimize the following three items; the hardware resources (i.e. functional units and registers), the number of control steps and the number of checkpoints. The cost function is defined as the following equations:

$$Constraint\_cost = \sum_{i=1}^{N} \alpha_i (Ccomponent[i] - Crequirement[i])^2 \qquad (1)$$

$$Min\_cost = \sum_{i=1}^{M} \beta_i Ccomponent[i] \qquad (2)$$

$$Total\_cost = Constraints\_cost + Min\_cost \qquad (3)$$

where

- *Ccomponent*[*i*] is the cost of i^{th} element in array A containing a set of constraint requirements,
- *Crequirement*[*i*] is the constant cost of i^{th} element in array A,
- An array B consists of the rest components not in set A,
- N and M are the size of arrays A and B, respectively,
- α_i and β_i are the arbitrary variables. They control whether the goal of the design is optimization for performance (speed) or for minimum area search.

Note that the total components are all constraints, i.e., the number of functional units, the number of registers, the number of control steps, the number of checkpoints, and the maximum recovery time. Furthermore, *Ccomponent*[*i*] is calculated by a gene in population but *Crequirement*[*i*] by the given constraints. For example, to define array A and B, if our problem is to optimize the number of control steps, array A contains a set of the type of each ALU, number of registers, number of checkpoints and the maximum recovery time. Therefore, array B has only one element, the number of control step.

The costs of elements in both array A and B mentioned above are computed as follows. The cost of the ALU of each type is equal to the number of ALUs of that type plus the size of the ALU type searched from the module library. The cost of register is derived from the number of registers plus the given size of register. The cost of the control step is equal to the number of control steps. The cost of the checkpoint and the maximum recovery time is equal to the amount of checkpoints and the size of the maximum recovery time.

The cost of each gene is computed using equation 3 and for fitness calculation of each gene using a ranking method and linear ordering [13]. in equation 4.

$$f(i) = k - \sigma \cdot rank(i) \qquad (4)$$

where k is the mean of *raw_fitness* and σ is the standard deviation of the same. Furthermore, *rank*(*i*) is the function that returns the rank of gene i in the population. This is a value between 1,....,*popsize* (the population size). The most fit gene has rank

1 and the least fit gene has rank *popsize*. Before above computation, we calculate the *raw_fitness* from the following equation.

$$raw\_fitness = \frac{1}{Total\_fitness} \qquad (5)$$

The cost and fitness calculation for the initial population is shown in Table 4.2.

Table 4.2 The cost and fitness function

Gene	Total_cost	Fitness
1	4.300000e+01	4.370648e-01
2	1.320900e+04	2.6488+7e-02
3	1.101400e+04	3.973316e-02
4	2.972000e+03	1.324439e-01
5	3.512000e+03	1.191995e-01
6	6.676000e+03	6.622193e-02