CHAPTER III

THEORETICAL BACKGROUND

3.1. The Problem of High Level Synthesis with Micro-Rollback Capability

High level synthesis is a sequence of tasks that transforms a behavioral representation into a Register Transfer Level (RTL) design. The design consists of functional units such as ALUs and multipliers, storage units such as registers, and interconnection units such as multiplexers and buses. Before compiling into the RTL description, the behavioral description is transformed into intermediate graph-based representations depicting both a data flow and a control flow called a control-data flow graph.

The problem of Self-recovery Micro-rollback Synthesis (SMS) combines the problem of functional unit scheduling and assignment with the problem of checkpoint insertion and microprogram optimization. It is well known that those problems are inherently interdependent. Thus, the optimal results of all problems must be considered simultaneously.

The scheduling is a process that assigns operations in the data flow graph to control steps. Then the assignment process selects each operation to an appropriate type of functional unit from a module library to execute the operations specified in the data-flow graph.

Self-recovery micro-rollback is a technique for recovering from transient faults, which are temporary interruptions of the logic level of signals. A checkpoint is the state of a computation at a particular instant of time, i.e., the values of all live variables. Whenever an interruption occurs, there is no need to save the status of the machine immediately. The status that is saved into the registers at the previous checkpoint is loaded into the machine to re-execute from that state. If the checkpoint is uncorrupted and the fault is no longer disrupting the system, the new execution will proceed correctly.

For instance, in Figure 3.1, the self-recovery micro-rollback process operation is demonstrated. The checkpoints are inserted at control step 2 and 4. As indicated in the figure, execution proceeds until the end of control step 2. At that point, if an error is detected, all inputs in the beginning of CDFG are loaded into the machine to re-start. Otherwise, the execution proceeds to control step 3. At the end of control step 4, if an error is detected, the computation re-executes at control step 3 by loading the status of the variables which are stored in registers. If no error occurs, execution completes.

The values of all variables stored in registers at one control step are maintained until the next checkpoint. Therefore, the live variable graph which is drawn from the scheduled CDFG is the starting point of the checkpoint insertion process. In Figure 3.2, a live-variable graph shows the control step in which each variable in the CDFG is active or live. Each node at a particular control step in the live-variable graph represents a register to save the state of the variable at that control step. The lifetimes of variables without the checkpoints are presented in Figure 3.2 (a) and three registers are required. After checkpoint insertion, the number of registers becomes four as

shown in Figure 3.2 (b) since variables d and e are saved and maintained until the end of control step 4 to be re-executed if an error occurs.

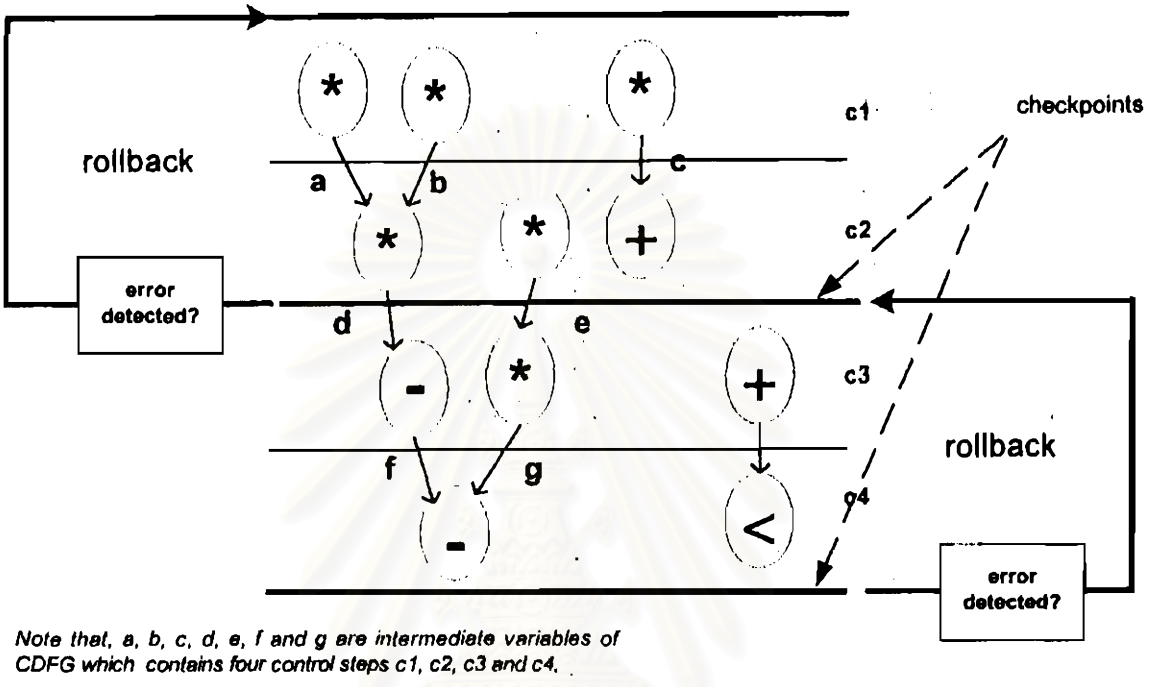

Figure 3.1. The self-recovery micro-rollback process operation.

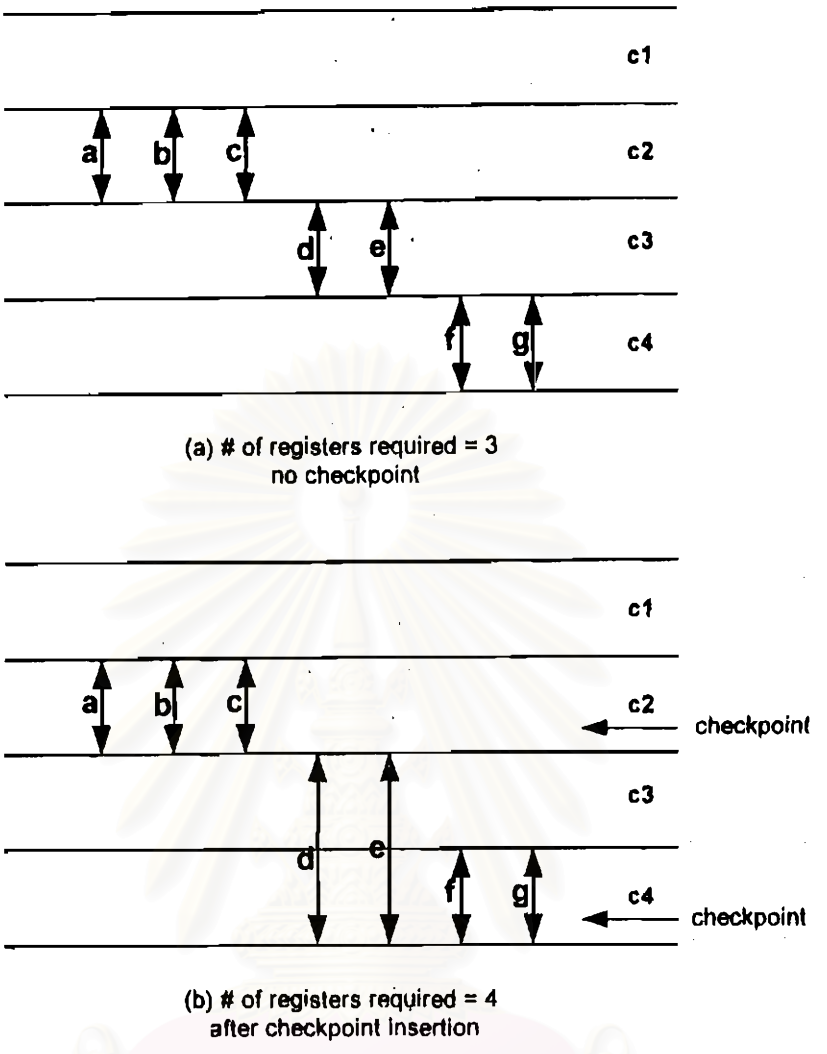

Figure 3.2 Lifetimes of intermediate variables for CDFG of Figure 3.1

3.2. Genetic Algorithm

Genetic Algorithm (GA) was first proposed by John Holland at the University of Michigan in 1975 [13]. He and his students investigated and proved that GA is a significant contribution for scientific and engineering application. Since then, the output of research work in this field has grown rapidly. Now, the GA development has

reached a stage of maturity [17]. This results from the effort of academics and engineers all over the world.

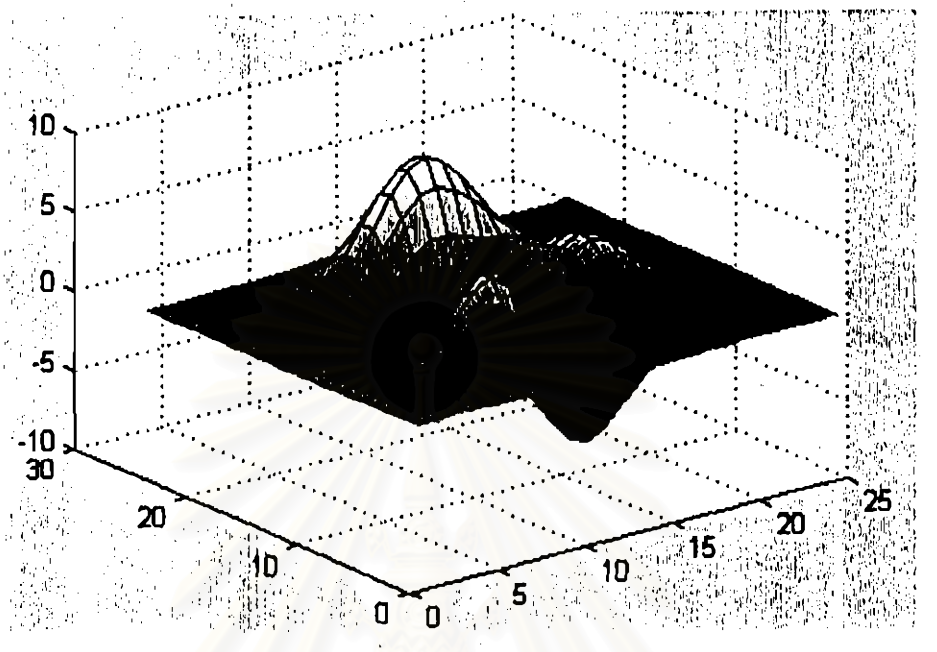

Figure 3.3 An error surface

GA is not a technique that requires the use of derivatives. The obtained optima are evolved from generation to generation without stringent mathematical formulation such as the traditional gradient type of optimizing procedure. Gradient descent which calculates the slope of error surface at the current position works well when the error surface is relatively smooth, with few local minima.

However, most real-world data has the distorted error surface by noise. Figure 3.3 illustrates an error surface which would prove difficult for gradient descent because of the local minima. GA is less sensitive to local minima because it constitutes a parallel search of the solution space, as opposed to a point-by-point search.

Hence, GA is usually applied to optimization problems that are difficult to solve or cannot be solved by stringent mathematical formulation. It is also used to resolve NP-hard and NP-complete such as traveling salesman, scheduling and design problems. It performs searching throughout the solution space to find the near optimal answer.

3.3. Genetic Algorithm Basis

Genetic Algorithm (GA) is a technique imitating biological process of natural selection (Darwin's rule) by which only good or fit being can survive [13]. The theory of Charles Darwin may be summarized as follows. (a) The individuals of a species show variation. (b) In general, more offsprings are produced than needed to replace their parents. (c) Populations cannot expand indefinitely and, on average, population sizes remain stable. (d) There must be competition for survival. (e) Therefore the best adapted variants (the fittest) survive.

GA uses a direct similarity of natural behavior following Darwin's theory. Above all, the problem to be solved by GA must be first encoded into a gene. There is no uniform encoding scheme for every problem. The encoding scheme varies from one problem to another problem. The appropriate encoding for the problem has to be devised.

Standard Genetic Algorithm ()

1. t =0.
2. Generate initial population (valid genes).
3. Calculate fitness values of each gene.
4. While the conditions are not satisfied and t<N do
   (a) t = t + 1
   (b) Select parents by random.
   (c) Recombine the population by cross-over and mutation operations.
   (d) Calculate fitness values of valid child genes.
   (e) Select the new population from the old population and the child population.

Figure 3.4 Structure of Genetic Algorithm

The structure of GA is illustrated in Figure 3.4. At the beginning, a set of the first generation of gene population is randomly produced. We also evaluate their fitness as different beings possess unequal capability to survive. After that, the genes in the set are randomly selected to produce the next generation genes with the high fitness cost. The genes with low fitness cost are eliminated. The producing process continues until the number of generations reaches the specified value or there is no new gene produced. The set of new genes is generated by three main gene operations, which are mutation, crossover, and inversion. Moreover, the conditions in the while loop depend on the problems to be solved. For example, the condition for the traveling salesman problem is the minimum total traveling distance. Variable t counts the number of generations whose maximum value is denoted by a constant N.

3.4. Principal Factors of Genetic Algorithm

The performance of the GA is controlled by the following factors.

3.4.1. Encoding Scheme

Encoding scheme is referred to as genes of a chromosome which can be commonly structured by three ways: binary string, gray code, and floating point. The binary scheme is traditionally used in GA but not appreciated in some problems such as the problem concerning many variables with large domain. The second scheme is the gray code which is slightly modified from the binary coding. Note that the gray coding has the property that any two points next to each other in the problem space differ by one bit only.

As we attempt to move GA closer to the problem space, we manipulate real-value directly with each chromosome to deal with real parameter problem. The implementation of each chromosome vector is encoded as the vector of floating point numbers. Generally, this implementation would be much better than binary scheme in many aspects. It is faster in computation and easier for designing other operations incorporating specific problem. In addition, it is more consistent from the basic of run-to-run.

3.4.2. Cost and Fitness Function

Cost function is the link between the GA and the problem to be solved. It is one of the most significant elements to assess the GA performance. The value of the cost function is calculated for an individual of population and fitness value is settled on its

basis. The interaction between a chromosome and a cost function provides a measure of its fitness that is used when carrying out reproduction. Its fitness is supposed to be proportional to the utility or ability of the individual which that chromosome represents.

3.4.3. Crossover and Mutation

Crossover occupies a special place in the heart of GA. In nature, crossover occurs when two parents exchange parts of their corresponding chromosomes. In GA, the crossover recombines the genetic material in two parent chromosomes to make two children. This is called by John Holland "one-point crossover" illustrated in Figure 3.5 (a).

In some situation, using one-point crossover is inefficient such as in schemata of long defining length (building block). A multipoint crossover can be submitted to overcome this problem and its performance of generating offsprings is satisfying. An example is demonstrated in Figure 3.5(b) where multiple crossover points are randomly selected.

Another operator is called "uniform crossover" which is similar to multipoint crossover. But it needs a randomly generated crossover template which is the pattern of crossover point. There is an example in Figure 3.5(c). The length of string 0-1 in the template is equal to the length of chromosome. Therefore, at 0 in the template, the gene of child 1 is placed by the gene of parent 1 and the gene of child 2 by the gene of parent 2. At 1 in the template, the gene of child 1 is placed by the gene of parent 2 and the gene of child 2 by the gene of parent 1.

Since the uniform crossover exchanges bits rather than segments, it can combine features regardless of their relative location. This ability may outweigh the disadvantage of destroying building blocks and make uniform crossover a superior operator for some problems.

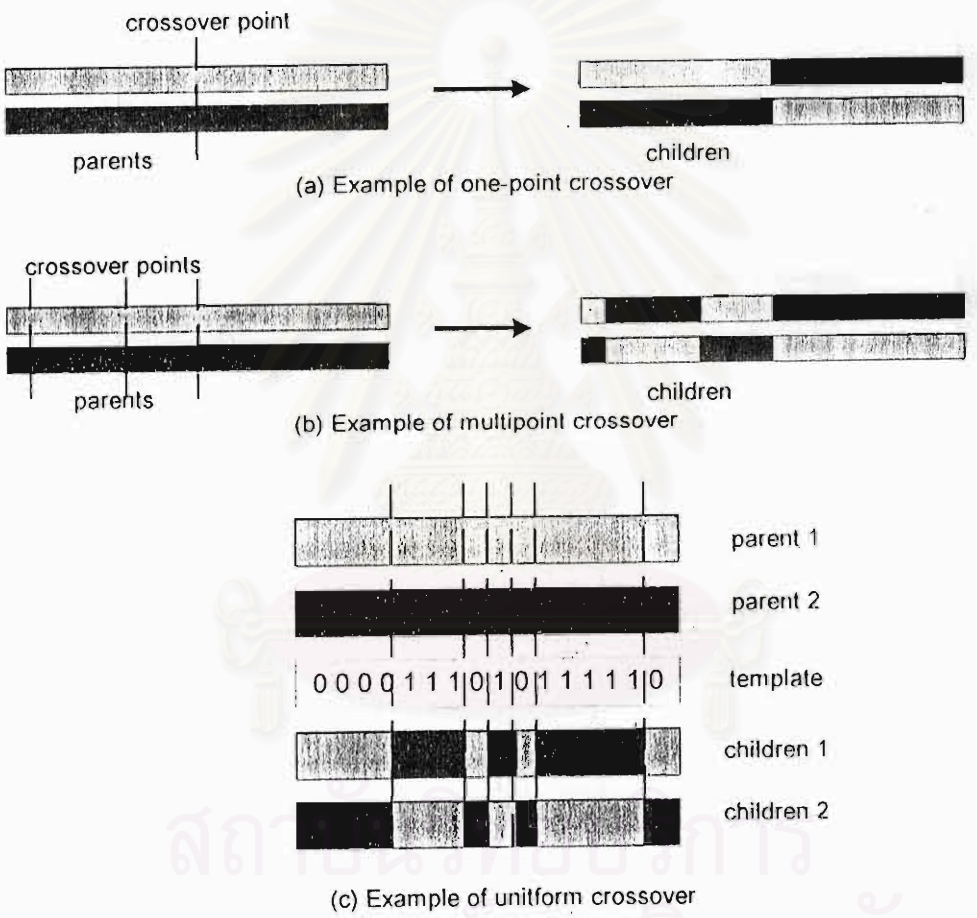

Figure 3.5 An example of crossover

Mutation is the process applied to each offspring individually after the crossover exercise. In GA, this operator creates new individuals by a small change in

a single individual by random selection. When mutation is applied to a bit string, it sweeps down the list of bits, and replaces each by a randomly selected bit if the probability of test passes. It is called "Bit Mutation" as shown in Figure 3.6. In addition, it has an associated parameter probability that is typically quite low.

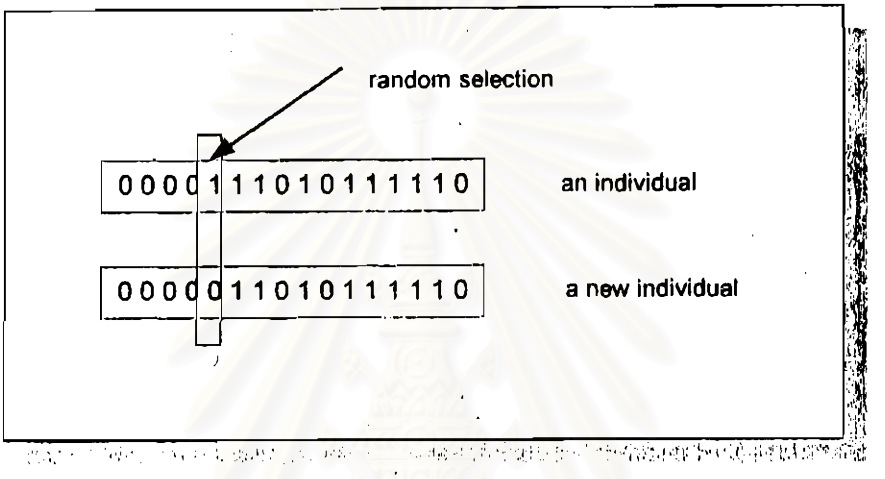

Figure 3.6 An example of mutation