Chapter 1

Introduction

During the 1970s, the transistor count for VLSI technology increased from 10,000 to 100,000 on one chip. In the 1980s it went up to 1 million. Clock frequency ranged from 200KHz to 2 MHz in the 1970s, increasing to 20MHz in the 1980s. Today, chip density can reach approximately 40 million transistors per chip, while clock frequency can reach 600MHz. According to predictions, by the year 2000, chip density will exceed 1 billion transistors and clock frequency will exceed 1 GHz [LiSh97].

From the 1970s through the 1980s, research for computer-aided design (CAD) progressed as circuit complexities increased, advancing from transistor level circuit synthesis to logic level synthesis. With the assistance of logic level synthesis tools, designers can capture system specifications at a higher level of abstraction. The advantage of working at a higher level of abstraction is that it reduces the number of objects the designer has to manipulate, enabling the designer to design larger and more complex systems in shorter periods of time. High-Level Synthesis (HLS) emerged in the late 80s, allowing the designer to capture system specification at an even higher abstraction level. Hardware Description Languages (HDLs), such as VHDL or Verilog [ArGr88] were used, generating structure descriptions composed of sets of interconnected components called netlists.
HLS has been actively investigated in the past decade. There are several high quality products used in industry, such as IBM BooleDozer, Synopsys Design Compiler (which has more than 90% of the market share), Synopsys Behavioral Compiler, and Cadence Synergy. In academia, two products used are Stanford OLYMPUS and U.C. Berkeley HYPER. As HLS tools have matured and have been gradually accepted by the circuit design community, today's logic designers do not have to handcraft the logic gates using graphic capturing tools, or capture the low level logic behaviors using Boolean expressions. In fact, logic design is becoming more and more like software programming. HLS allows logic designers to concentrate on describing the behaviors of the circuits and exploring more design alternatives. Another advantage of using HLS is that it enables designer/programmers who do not have hardware backgrounds to do circuit design. The programmer/designer can use an HDL to describe system behaviors, and then use an HLS tool to translate the behaviors into circuits, providing more opportunities for non-circuit experts to produce chips.

The preceding historical representation of the advance of VLSI technology and CAD tool development illustrates that as VLSI technology allowed designers to pack more transistors on one chip, CAD tools that allowed designers to capture a higher level of abstraction also emerged. Cycling time for today's microprocessors or top-of-the-line ASICs is about eighteen months. These complex components usually contain millions of transistors. If there were no CAD tools capable of capturing a higher level of abstraction, it would be impossible to design and manufacture chips with such complexity in such a short period of time. It is also hard to imagine that a group of designers could use only circuit schematics and Boolean equations to specify, document, and communicate the design with the complexity of today's chip.

Although current HLS is capable of capturing system specification at a higher level of abstraction, the HDL code used as an input to synthesize the system may become unmanageable or incomprehensible by designers. The VHDL code used to
document/synthesize the Pentium microprocessor, which contains about 7.5 million transistors, is said to have over one million statements. As stated before, the intent of HLS is to allow designers to work on higher levels of abstraction to reduce the number of objects under consideration. But with over one million statements of VHDL code, the advantage seems to be lost again. To cope with this problem, a new CAD tool that is capable of working at a higher level of abstraction than modern HLS tools is needed. However, what kind of abstractions are needed for VLSI technology? The following sections will discuss the similarity of software and hardware developments and review the history of tools used by software developers, which will provide clues to develop better hardware abstraction capturing tools.

1.1 Resemblance of Software and Hardware Synthesis

Software programs are translated into machine-dependent executable code. The programmers start by analyzing system specifications and then describe the specifications, e.g., writing programs, using high-level languages such as FORTRAN, C, or C++. A compiler is used to translate the programs into machine-dependent executable binary code. Intermediate steps include assembly code generation and optimization. This process of software development is illustrated in Figure 1.1 (a).

For circuit designers, the system specifications are captured using a Hardware Description Language, such as VHDL or Verilog. The HDL code is then put into an HLS Design Compiler. Next, the HLS Design Compiler translates the code into an intermediate form, such as a Control Data Flow Graph (CDFG) [GajD92]. The CDFG is then used by the Design Compiler to perform optimization tasks such as scheduling and allocation. The output of the HLS tool is a technology dependent netlist. This netlist contains the structure of the components and the component interconnection information. Extra steps, such as floor planning, placement, and routing, are needed to generate a complete physical layout, which can be sent to the manufacturer for fabrication. This process of hardware development is illustrated in Figure 1.1 (b).
1.2 Tools for Software Development

Because of the resemblance between software and hardware development, studying the evolution of software development should provide clues for where HLS should be heading in the increasing complexity of VLSI technology. When computers (microprocessors) were invented about a half century ago, programmers had to use machine code, which contains only binary information, to describe a simple algorithm or function, which is error-prone and very tedious. As the technology improved, computers could be used to implement more complex system specifications. Assembly languages are used to capture higher levels of behaviors/functions, such as move, shift, jump, and compare. If an assembly language is used to develop applications, such as an accounting system, the computer program may contain thousands or millions of statements, which is unmanageable by programmers. To alleviate this problem, High-level languages, such as FORTRAN and C, were introduced so that programmers would be able to describe system specifications at a much higher level of abstraction, which in turn reduces the number of statements to be considered and brings the program complexity to a manageable level.
In the mid-80s, Apple Computer brought Graphic User Interfaces (GUIs) to their product line. Before GUIs, computer users had to memorize dozens of commands in order to make computers perform the desired tasks. While GUIs are friendly to computer users, programming them is a challenge to computer programmers. It takes substantial effort and several hundred lines of code to open a window or lay out some simple push buttons. For more complicated GUIs, a programmer might spend most of his time programming the user interface instead of concentrating on programming system specifications. The appearance of GUIs dramatically increased the complexity of programming. To alleviate the complexity of GUIs programming, several new languages, like Visual Basic and Tk/Tcl, were invented to capture an even higher level of abstraction. Common GUIs functions, such as opening a window, creating radio buttons, or generating pop-up menus, are treated as objects. These complex objects are used just like simple objects such as integers, arrays, and strings. Building GUIs has now become a simple task because the new languages considerably reduce the number of objects that programmers have to think about.

With the maturity of the GUIs and object-oriented technology, programming tools called Visual Programming allow users to do programming without knowing any programming language. These tools capture the types of control flows and functions of system specifications using graphic interfaces. The types of control flows and functions can be system pre-defined or user defined, again reducing the number of objects that programmers need to handle. These tools include ProGraph [CoGi89] and LabView [JaWa93].

1.3 Tools for Hardware Development

Circuit design deals with three types of objects: transistor, resistor, and capacitor. Circuit synthesis uses these three types of objects to capture the most fundamental functions of all digital circuits: ON and OFF. Logic synthesis deals with three basic functions: AND,
OR, and NOT. These three functions are the basic functions for all modern logic design theories. AND, OR, and NOT are higher level abstractions than ON and OFF are, meaning AND, OR, and NOT are capable of representing more objects (transistors) than ON and OFF, a natural progression because of increasing design complexity. Designers need to represent the design at a higher level of abstraction to reduce the number of objects under consideration.

HLS deals with many more types of basic functions or basic scheduling blocks (BSB), which is a term borrowed from HW/SW co-design [ErHe93]. These basic functions are similar to the basic operators of software high-level languages. Basic functions include:

- basic arithmetic functions, such as add, subtract, multiply, and divide.
- control flow functions, such as if, while, when, and for.
- transfer operators, such as “:=” and “<=" in VHDL.

HLS uses the above functions as the BSBs to perform optimization tasks, like scheduling and allocation. In the allocation or binding steps, the BSBs are mapped to elementary functional units such as adders or ALUs for execution.

Using ON and OFF, we can represent a single transistor in the physical layout. Using AND, OR, and NOT, we can represent about 10 transistors in the physical layout [WeEs93]. Using the basic scheduling block in the HLS, we are able to represent several hundred transistors in the physical layout. With up to 40 million transistors in a single chip in modern VLSI technology, any design may require tens of thousands or even millions of lines of HDL codes as the HLS input for the design representation. This complexity becomes very difficult for human beings to manage. Capturing the abstractions at a higher level than HLS can do at present is urgently needed.

This phenomenon is very similar to that of ten years ago when software programmers tried to use high-level languages to build complex graphic user interfaces. Currently the
hardware designers manage to use HLS to design chips with millions of transistors. New tools or new languages were invented for software programmers to reduce program complexity. It is also necessary to provide new approaches for HLS to tackle the rising complexity in hardware design.

Because of the increasing VLSI design complexity, design reuse has been advocated to reduce design cycles. **Intellectual Property** (IP) is a term that emerged recently, which refers to synthesizable models/cores [EETi98a]. The IPs provided by commercial vendors usually include HDL source code, timing diagrams, simulation and verification packages, synthesis scripts, and complete documents. Currently, substantial efforts are being made to standardize IPs such that they can be fully synthesizable in different synthesis environments and portable between different technologies. The IPs available in the market presently range from simple Arithmetic Logic Units (ALUs) to complex microcontrollers [EETi98b]. It is expected that by the year 2000, ASIC and system-on-chip design will have as much as 80 percent of their contents coming from purchased IPs.

### 1.4 A New Optimization Algorithm

One of the major reasons that GUIs prevailed in software developments is that more powerful computers are readily available. GUIs would not be possible if the computer runs at a clock speed 8 MHz with 64 K bytes of main memory. The hardware design, especially HLS, should also take advantage of advanced computer technology and employ new algorithms whose application was once difficult or impractical, to explore tremendous design space.

There are many interdependent NP-hard problems associated with the HLS, like scheduling, allocation, and binding problems [GajD92]. Most HLS research investigated these NP-hard problems using constructive heuristics, integer linear programming (ILP), or simulated annealing (SA) [GajD92].
Most constructive heuristic algorithms are capable of solving only a single problem, because multiple interdependent problems are difficult to model using constructive heuristic approaches. Another major disadvantage is they get trapped in local optima. Integer linear programming can be easily applied to multiple interdependent problems in searching for the optimal solutions, but in practice the ILP approach can only be used in very small problems because the number of variables and inequalities grow exponentially when the problem size increases. The simulated annealing algorithm can be easily used to model a multiple interdependent optimization problem. Simulated annealing provides a mechanism to escape local optima and can produce good results. In fact, simulated annealing is extensively used in contemporary CAD tools to search for optimal solutions. The disadvantage of simulated annealing is that it suffers from very long run times [GajD92].

Genetic algorithms (GAs) were proposed by Holland in 1975 [Holl75]. Since then, genetic algorithms have been successfully applied in many research disciplines, such as biology, operations research, image processing, pattern recognition, and physical science [Gold89]. Genetic algorithms are non-deterministic algorithms that can be applied to search for optimal solutions to NP-hard problems. Unlike simulated annealing, which only looks randomly from point to point, genetic algorithms work with sets of points concurrently. They efficiently exploit historical search information to speculate on new solutions.

**Definition 1.1** A search point is a potential solution in the search space.

Genetic algorithms were applied in HLS to solve the interdependent NP-hard problems by several researchers [WeHe90, RaGu95, SaAl96, KiCh97] who demonstrated that the results are better than heuristics. They achieve the same results as simulated annealing but in much shorter run times. The main disadvantage of using GAs is that they require substantial computer resources, since they keep tens or even hundreds of search points in the computer memory simultaneously.
With increased computing power and memory capacity, keeping multiple solutions in the computer memory is feasible. A superior non-deterministic search technique like GAs should be applied, rather than continuing to use simulated annealing.

1.5 Synthesis Types

Table 1 illustrates the levels of abstraction that current synthesis tools are designed to capture. Contemporary synthesis methods are classified into three major categories: hardware synthesis, hardware/software co-synthesis, and software synthesis. Software synthesis does not require any hardware development and it uses only software languages to capture the system specifications, which can be implemented on different kinds of computer hardware. The system complexity that software synthesis can capture is the highest of the three types of synthesis. The applications include database management, network simulation, and information security.

Hardware/Software (HW/SW) co-synthesis partitions the captured system specification into software, which is then executed by microprocessors, and hardware, which is then executed by full-custom Application Specific Integration Circuits (ASICs) or synthesized Field Programmable Gate Arrays (FPGAs) [ThAd93, GuCo92, ErHe93]. The goal of HW/SW co-synthesis is achieving an optimal solution that satisfies the required constraints, such as performance, area, and cost. HW/SW co-synthesis requires both software and hardware developments which use both software languages and hardware languages. The granularity of the basic scheduling block is defined by the designers, which can be as fine as a microprocessor instruction (i.e., operation or function) or as coarse as a group of several hundred sets of instructions (i.e., task). The software part is usually optimized by tools like profilers and purifiers [ErHe93]. The hardware parts are usually optimized by HLS synthesis tools like Synopsys Design Compiler. The applications that can employ SW/HW co-design include Digital Signal Processing, Instruction-set processors, and Embedded system.
Table 1.1  Level of abstractions for various synthesis types.

<table>
<thead>
<tr>
<th>BASIC FUNCTION</th>
<th>IMPLEMENTATION METHOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>tasks, functions, operations</td>
<td>Software Synthesis</td>
</tr>
<tr>
<td>tasks, functions, operations</td>
<td>HW/SW Co-Synthesis</td>
</tr>
<tr>
<td>Floating point operations, LOGARITHM, MOD, COSINE, SINE, ARC-COSINE, ARC-SINE, SQRT, etc.</td>
<td>high level abstraction</td>
</tr>
<tr>
<td>Fixed point operations, ADD, SUBTRACT, MULTIPLICATION, COMPARE, etc.</td>
<td>low level abstraction</td>
</tr>
<tr>
<td>NAND, NOR, NOT, AND, OR</td>
<td>Logic Synthesis</td>
</tr>
<tr>
<td>ON, OFF</td>
<td>Circuit Synthesis</td>
</tr>
</tbody>
</table>

Hardware synthesis requires only hardware development so that all the system specifications are executed on full-custom designed integrated circuits. Hardware synthesis can be classified into four categories, according to the levels of abstraction that the synthesis types can capture. The types of hardware synthesis are shown toward the bottom of Table 1. As stated in previous sections, circuit synthesis is used to capture the abstractions of ON and OFF. Logic synthesis uses AND, OR, and NOT to capture system
specifications. The levels of abstraction for high-level synthesis can be further
categorized into low and high level of abstraction. In Table 1, the functions listed in low
level abstraction in high-level synthesis are basic arithmetic operations and control flows
that are currently used by HLS tools to capture system specifications. The functions
listed in high level abstraction are functions which future HLS tools may be able to use
them as basic functions.

1.6 Objectives and Contributions of this Thesis

The research work reported in this thesis had three goals: (1) classification of existing
complex functions so that HLS tools can use these functions to capture system
specifications, (2) creation of a new synthesis level employing complex functional units,
and (3) provision of a modeling technique based on a new type of Genetic Algorithm
called Problem Space Genetic Algorithm (PSGA). The purpose of a PSGA is the search
for optimal solutions.

Contributions of the research work in this thesis include: (1) a survey and classification of
the existing complex function devices, (2) construction of an interface to assist designers
in searching library components, (3) proposing a target architecture suitable for synthesis
tools that support complex functions, (4) demonstration of bounded instead of fixed
execution delays for optimization of complex functions, (5) invention of path vectors
that are used to identify data dependencies in CDFGs, (6) proposal of a set of metrics
used to evaluate the quality of functional partitioning, and (7) application of the Problem
Space Genetic Algorithm to digital system syntheses with complex functions.

This thesis is organized as follows: Chapter 2 presents a survey of high-level synthesis,
which includes issues such as data representation, scheduling, allocation, and
optimization algorithms. Chapter 3 presents a survey of existing off-the-shelf
components. The function types of the components are investigated and classified into
several categories. These functions can be used as the basic function types for future HLS
tools to capture higher levels of abstraction. Chapter 4 proposes a design style that incorporates complex functional units. The communication channels for the new design style are implemented using partitioned-buses and buffers. Local memory for each functional unit is organized into register files. Chapter 5 discusses the issues involved in synthesizing the new design style from a high-level representation. The issues include functional partitioning at the high-level design stage, local memory and communication buffer optimizations, system delay minimization, and interconnection topology optimization. Chapter 6 models the optimization issues addressed in Chapter 5 into a combinatorial optimization problem. This problem is then addressed using a heuristic optimization algorithm that is based on the Problem Space Genetic Algorithm. Chapter 7 presents the experimental results generated by the new optimization algorithm and also proposes a quality metric to evaluate the solution qualities. Chapter 8 summarizes the findings and presents concluding remarks.
Chapter 2

Survey on High-Level Synthesis

This chapter starts with an overview of High-Level Synthesis (HLS), which is developed by means of an example in Section 2.1. The example demonstrates several aspects of HLS technique:

- use of natural language to describe the system specifications,
- modeling of the specifications using VHDL [ArGr88, IEEE88],
- translation of the VHDL code into a graphic representation,
- scheduling of operations into control steps for execution,
- selection of functional units to implement the operations,
- allocation of operations to the functional units for execution,
- connection of the functional units and registers, and
- generation of a control unit to synchronize the operations.

After the step-by-step tasks for High-Level Synthesis are introduced, previous research work on the tasks of High-Level Synthesis, such as scheduling and allocation, is reviewed in Section 2.2. Different synthesis design styles are reviewed in Section 2.3. General optimization approaches for HLS are surveyed and discussed in Section 2.4.
2.1 Overview of High-Level Synthesis

The definition of High-Level Synthesis was first given by McFarland et al. [McPa88]. They stated that “High-Level Synthesis task is to take a specification of the behavior required of a system and a set of constraints and goals to be satisfied, and to find a structure that implements the behavior while satisfying the goals and constraints.” The procedures for transforming behaviors into structures have been extensively investigated in the past decade, and several sub-tasks of the transformation have been recognized [GajD92]. To show the general flow of High-Level Synthesis, the procedure used to compute the greatest common divisor (GCD) is described using natural language and VHDL modeling. The step by step process of synthesizing the VHDL codes into a register transfer circuit structure is then presented.

2.1.1 A High-Level Synthesis Example

Natural Language Description
The system specification for GCD is described using natural language as follows:

*If signal START is asserted to 1, variables A and B are loaded from A_port and B_port respectively. The GCD for A and B is computed by 1) if A is greater than B, A is equal to A minus B; otherwise, B is equal to B minus A. 2) if A is not equal to B, go to step 1; otherwise, the GCD is equal to A. The result is sent to output port GCD_out and signal DONE is set to 1.*

VHDL Model
The above behavioral description can be modeled using VHDL. The VHDL program is shown in Figure 2.1.
Control Data Flow Graph

The VHDL program is then translated into an internal graphic representation known as a Control Data Flow Graph (CDFG) [OrGa86, GajD92], which is shown in Figure 2.2. The CDFG representation contains two parts. The control flow graph represents the execution sequence, and the data flow graph shows the data dependencies. In the control flow graph, the nodes represent the control structures, such as loop, branching, and branching merge. In the data flow graph, the nodes represent the operations or functions. The operations are basic arithmetic operations such as add (+), subtract (-), or compare (>). The edges represent the flows of the variables generated by the operations. The dashed arrows that point from the control flow graph to the data flow graph express the associations of the control nodes to a block of operations in the data flow graph. The outputs of the operations/functions in the data flow graph may feed back to the control flow graph as status signals for controllers to make control flow decisions.

```vhdl
entity GCD is
  port(
    A_port, B_port: in INTEGER;
    GCD_out: out INTEGER;
    CLK: in BIT;
    START: in BIT;
    DONE: out BIT);
end GCD;

architecture BEHAVIOR of GCD is
begin
  process
    variable A,B: INTEGER;
  begin
    wait until (START = 1);
    A := A_port;
    B := B_port;
    DONE<='0';
    while (A /= B) loop
      if (A>B) then
        A := A - B;
      else
        B := B - A;
      end if;
    end loop;
    GCD_out <= A;
    DONE <= '1';
  end process;
end BEHAVIOR;
```

Figure 2.1 VHDL Model for GCD.
Scheduling and Allocation

Scheduling is binding the operations in the Data Flow graph to control steps for execution. Allocation is the process of binding the operations to functional units (i.e. components or cells) for execution. For the GCD example, both scheduling and allocation are trivial because of its simplicity. One of the possible solutions is shown in Figure 2.3.
If a comparator is chosen from the cell library to perform ‘=’ and ‘>’, and a subtractor is selected to perform ‘-‘, two functional units and three control steps (states) are needed to obtain the GCD for the two given variables A and B.

Although the scheduling and allocation tasks are trivial in the GCD example, the search space grows exponentially as the problem size grows. Scheduling and allocation are two interdependent NP-hard problems [GaJo79].
After the functional units are chosen, the number of registers required to store the variables and constants and the interconnections among the registers and functional units still need to be decided. This is an NP-complete problem on which numerous papers have been written [GajD92]. As stated before, Figure 2.3 is a possible solution for the example.

**Control Unit Generation**

After the datapath is completed, a control unit must be generated to synchronize the execution of the operations. The control unit can be random logic, a read-only memory (ROM), or a programmable logic array (PLA). In this example, a micro-code design style is shown in Figure 2.4. The shaded area indicates the control words stored in the ROM. In this design style a ROM, a state register, and an address generator are needed. The outputs of the ROM are the control signals for the functional units, registers, and steering logic (i.e., multiplexers). Each control word must also contain the next state information which is fed into the state register. The content of the next state register and the status signal, flag, from the datapath are used by the address generator to produce an address to retrieve a control word from the ROM for the next execution cycle.

For example, from Figure 2.4, if the current state is S0 and the status signal flag is ‘1’, the control word in address [1] will be output to the datapath and the next state signals which are a part of the control word are fed to the next state register. The contents of the next state register and the status signal flag are then used by the address generator to retrieve the control word from the ROM for the next execution cycle.

### 2.2 Typical High-Level Synthesis Tasks

The typical steps of High-Level Synthesis were presented in the previous section. Two important tasks, scheduling and allocation, have been subjects of extensive research [McPa88].
CDFGs are usually used as the input for the scheduling and allocation tasks, and are defined below. They will be used in the discussion throughout this dissertation.

![Figure 2.4 ROM style control unit for GCD.](image)

**Definition 2.1** CDFG is expressed as $G(V,E)$, where $V=\{v_1, v_2, \ldots, v_n\}$ is a set of vertices representing the operations/functions, and $E=\{(v_i, v_j) \mid v_i, v_j \in V\}$ is a set of edges connecting the vertices, which represent the data dependencies among the operations/functions.
2.2.1 Scheduling

In this section, several commonly used scheduling algorithms are reviewed briefly, such as As Soon As Possible (ASAP), As Late As Possible (ALAP), list scheduling, and Force-Directed Scheduling (FDS). Other scheduling techniques that utilize estimation algorithms to find the resource or timing lower bounds to prune the design space before scheduling are also reviewed. These include Multi-Schedule Scheduling, Lower-Bound Performance Estimation Scheduling, and Architectural Resources and Performance Estimation Scheduling.

**As Soon As Possible Scheduling**

If the timing given for an unscheduled CDFG is to be minimum and resources are not restricted, i.e., assume an unlimited amount of functional units are available, the scheduling problem can be solved using as soon as possible (ASAP) algorithm. An operation is scheduled in a control step as soon as all its predecessors are scheduled. The operations that do not have any predecessors are scheduled in the first control step.

The ASAP scheduling algorithm can be described by the following steps.

1. Set control step $C = 1$;
2. Select operation $v_i$ whose predecessors are all scheduled or who does not have any predecessors.
3. Schedule $v_i$ at $C$;
4. $C = C + 1$;
5. If there are any operations not scheduled, go to step 2; otherwise stop.

**As Late As Possible Scheduling**

Under the same constraints as ASAP, the scheduling problem can also be solved using as late as possible scheduling (ALAP). The step by step procedure is shown as follows:
1. If the timing constraint is given as $T$ control step, $C$ is set to $T$.
2. Select operation $v_i$ whose successors are all scheduled or who does not have any successors.
3. Schedule operation $v_i$ to control step $C$.
4. $C = C - 1$.
5. If there are any operations not scheduled, go to step 2; otherwise stop.

**Force-Directed Scheduling**

Force-Directed Scheduling (FDS) is a time-constrained scheduling algorithm published by Paulin and Knight [PaKn89]. The principle of FDS is to reduce the operation concurrency in a control step and distribute the operations evenly to each control step in order to minimize the number of functional units required. The algorithm starts by calculating the time frames of each operation. The time frame is also commonly called **mobility range**, which is the time interval in which an operation can be scheduled. The mobility range of an operation can be obtained by scheduling the CDFG using ASAP and ALAP. If an operation is scheduled at control step $cs$ by ASAP and is scheduled at control step $cl$ by ALAP, the mobility range is the difference between $cl$ and $cs$. The **probability** of an operation that can be scheduled in a control step which is in its time interval is the reciprocal of its mobility range. The probability for the operation to be scheduled outside the time frame is set to 0.

After the time frame, mobility range, and its probability in each control step of every operation in the CDFG are calculated, a **distribution graph** is created. The distribution graph shows the sum of the probabilities of each type of operation for each control step of the given CDFG. For the distribution graph of each operation type, the distribution in a control step $t$ can be represented as

$$d(t) = \sum_{\text{operation type}} \text{prob}(v, t)$$  \hspace{1cm} (2.1)
where \( \text{prob}(v,t) \) is the probability contribution of operation \( v \) to control step \( t \). The last step is to calculate the force for each operation in its time frame. For example, assume an operation \( v \) with a time frame from control step \( cs \) to control step \( cl \). The self force for \( v \) in control step \( t \), where \( cs \leq t \leq cl \), is calculated as

\[
\text{force}(t) = d(t) - \sum_{i=cs}^{cl} d(i) / (cs - cl + 1) \quad (2.2)
\]

The forces of \( v \)'s predecessors and successors must also be calculated by using Equation 2.2 whenever their time frames are affected by scheduling \( v \) to control step \( t \). The forces from predecessors and successors are called indirect forces. The total force for \( v \) to be scheduled in control step \( t \) is the sum of the self force and indirect forces. After all the forces of \( v \) in its time frame are calculated, \( v \) is scheduled to the control step that has the smallest force. The algorithm can be summarized as follows:

1. Select operation \( v \) for evaluation.
2. Evaluate the time frame for each operation.
3. Update distribution graph, using equation (2.1).
4. Calculate self forces for every control step in the operation’s time frame, using equation (2.2).
5. Add the predecessor and successor forces to self forces.
6. Schedule \( v \) to the control step that has the lowest force. The time frame of \( v \) is set to the selected control step.
7. If there is any operation not scheduled, go to step 1; otherwise stop.

**List Scheduling**

List scheduling is a resource-constrained scheduling algorithm. In many applications, area and cost are the major concerns so the number of functional units are given as the constraints. Under the resource constraints, list scheduling tries to find a schedule that
requires minimum system delay. Basically, list scheduling is a variant of ASAP scheduling, which uses a greedy approach to schedule as many operations as possible into a control step, subject to the constraint of the availability of functional units.

The computational complexity for list scheduling is \( O(n) \), and it is known to be very fast; However, a good priority function, which is used to decide which operations have higher priorities for scheduling at certain control steps, is difficult to find. Several commonly used priority functions are based on mobility ranges of the operations and the distances [Hu61] of the operations to the sinks. It happens very often that several operations have the highest priority but there are not enough functional units to accommodate all of them. In this case the commonly used approach is to pick the operations randomly. Because there are several ways to select the operations, some of the solutions may not be optimal.

The general list scheduling algorithm can be described as follows:

1. Set control step \( C = 1 \).
2. Get a list of operations that are ready to be scheduled.
3. Apply priority function to compute the priorities of the ready operations.
4. Schedule the first \( n \) operations and schedule them to \( C \), where \( n \) is the available number of functional units at \( C \).
5. \( C = C + 1 \).
6. If there are any operations not scheduled, go to step 2; otherwise stop.

**Multi-Schedule Scheduling**

Traditional high-level synthesis systems usually generate a single schedule. The single schedule is often obtained by using a heuristic approach by pruning a large number of search spaces. It is possible that better solutions are compromised. The multiple scheduling approach is proposed to produce a number of schedules [DaPi94]. During the scheduling stage, the functional unit costs for each schedule can be determined by calculating the maximum number of functional units needed in a control step. However, the costs for registers and interconnections cannot be determined during scheduling. The
reason is that the costs for registers and interconnections can only be determined when the operations are bounded to their functional units, which is performed during the allocation stage. Therefore, after these multiple schedules are obtained by using a depth first search algorithm with a partial schedule evaluation method, they are then used by an allocation algorithm to generate complete data paths for each schedule. The schedule with the lowest cost is then selected.

**Lower-Bound Performance Estimation Scheduling**
Rim and Jian [RiJi94] stated that after the resource constraint is defined for the scheduling, there is still an extremely large space for a synthesis system to search to find a best design. It is important for a synthesis system to use an estimation method in earlier designs to reduce the design space. An Integer Linear Programming (ILP) relaxation technique and a greedy algorithm are used to determine a lower-bound on the minimal number of control steps for a resource-constrained scheduling problem. The performance lower bound is then used to prune the search space for scheduling the operations in the CDFG.

**Recursive Scheduling**
Rim and Jian’s ILP relaxation technique was refined by Langevin and Cerny by using a recursive technique [LaCe93]. The method relies on the determination of an *as soon as possible under constraint* time value. The method computes a tighter lower bound but the speed is two times slower than Rim and Jian’s method.

**Architectural Resources and Performance Estimation Scheduling**
Sharma and Jian [ShRa93] proposed an algorithm that can be used to estimate the resource lower bound if a timing constraint is imposed, or to estimate performance lower bound if a resource constraint is imposed. From a given resource or timing constraint, the performance or resource lower bound can be obtained by iteratively computing the minimum operation overlaps in each time interval.
**Definition 2.2** If the critical path length for a CDFG is $t_{cp}$, a **time interval** is expressed as $[t_1,t_2]$, where $t_1 \leq t_2$ and $[t_1,t_2] \subseteq [0,t_{cp}]$.

**Definition 2.3** The **minimum operation overlap** in a time interval is the minimum number of functional units required to execute all the operations in the time interval.

After the performance or resource lower bounds are obtained, the scheduling problem is formulated and solved as an Integer Linear Programming (ILP) problem.

**2.2.2 Allocation**

Allocation is the task of assigning the operations to functional units, mapping the variables to registers, and interconnecting the functional units and registers. In this section, several register and functional unit allocation algorithms are reviewed. Interconnection allocation will be discussed in Section 2.3.

**Clique Partitioning**

Before applying the clique partitioning algorithm to bind variables to registers, which are generated by operations or input from the outside environment, a resource compatibility graph (RCG) must be created [SpTo90]. One variable can share the same register with another variable if their **lifetimes** do not overlap.

**Definition 2.4** A variable’s **lifetime** is the time period from the control step the variable is generated to the last control step the variable is used.

Let $RCG$ be defined as $G = (V,E)$, where $G$ is denoted as an undirected graph, $V$ is a set of vertices which represent variables, and $E$ is a set of edges. Edge $e_{ij} \in E$ is in the graph if the lifetimes of vertices $v_i$ and $v_j \in V$ do not overlap. A subgraph of $G$ is defined
as \( G' = (V', E') \), where \( V' \subseteq V \) and \( E' = \{ e_{ij} \mid e_{ij} \in E, v_i \text{ and } v_j \in V' \} \). The clique partitioning algorithm tries to find the minimal number of cliques that are complete subgraphs of the RCG. A clique is found if \( G' \) is a complete graph, which means for every pair of vertices in \( G' \), there exists an edge. Variables in the same clique are assigned to the same register. Because clique partitioning is an NP-complete problem [GaJo79], a heuristic algorithm is usually used. The following is a heuristic clique partition algorithm proposed by Tseng and Siewiorek [TsSi86].

The algorithm starts with a supergraph of \( G \), which is defined as \( SG = \{ SV, SE \} \), where \( SV = \{ \{ s \} \mid s \in V \} \) and \( SE = \{ \{ \{ s_i \}, \{ s_j \} \} \mid (s_i, s_j) \in E \} \). A supernode \( s_i \in SV \) is a common neighbor of the two supernodes \( s_j \) and \( s_k \in SV \) if there exist edges \( se_{ij} \) and \( se_{ik} \), where \( se_{ij} \) is the edge between \( s_i \) and \( s_j \), and \( se_{ik} \) is the edge between \( s_i \) and \( s_k \).

A procedure is used to return the set of supernodes that are common neighbors of \( s_i \) and \( s_j \), in which \( s_i \) and \( s_j \) are compatible, i.e., the variables in \( s_i \) and \( s_j \) can share the same register. Therefore, there is an edge between \( s_i \) and \( s_j \). The supernodes \( s_i \) and \( s_j \) which have the maximal number of common neighbors are united into a new supernode \( ns = \{ s_i \cup s_j \} \), and \( SV \) is modified as \( SV = (SV \cup ns) - s_i - s_j \). All the edges in \( SE \) that have links with \( s_i \) and \( s_j \) are deleted. New edges are added into \( SE \) that links \( ns \) with all the common neighbors of \( s_i \) and \( s_j \).

The above steps are repeated until there is no edge in the supergraph. The final supernodes contain sets of vertices of the original graph. All vertices in the same supernode form a clique and are assigned to the same register.

**Heuristic Coloring**

A resource conflict graph that is defined as \( G = (V,E) \), where \( V = \{ v_i, i=1,2,...,n \} \) is a set of vertices which is in one-to-one correspondence with the variables, and \( E = \{ e_{ij} \mid e_{ij} = (v_i,v_j), v_i,v_j \in V \} \) denotes the variable pairs whose lifetimes are overlapped.
The coloring algorithm starts with assigning colors to vertices such that no edge has both vertices of the same color. After all the vertices are assigned with colors, the vertices with the same color are bound to the same register. The problem of obtaining the minimum number of colors for the vertex coloring problem is intractable [GaJo79]. A simple heuristic vertex coloring algorithm [BasS78] can be as simple as scanning every vertex \( v_i \in V \) and assigning it with a color such that no vertex adjacent to \( v_i \) has the same color as \( v_i \).

**Left-Edge**

The left-edge algorithm was originally proposed for channel routing that is used for track assignment [HaSt71]. This idea is adopted by Kurdahi and Parker for register allocation [KuPa87]. To apply the Left-Edge algorithm for register allocation, a table has to be constructed in which the lifetime intervals of every variable are contained. The following is the step-by-step procedure:

1. Sort the intervals in the ascending order according to their left edges.
2. Assign the leftmost edge to the first register.
3. Find the first interval whose left edge is to the right of the last selected interval and assign it to the current register.
4. If no more intervals can be assigned to the current register, a new register is obtained and is started from Step 2.
5. Repeat the process until no intervals are left.

An alternative way of register assignment, which is similar to the Left-Edge algorithm, constructs and then colors an interval graph [MicG93]. Because the channel routing problem can be transformed to a problem that finds the minimum number of paths in the interval graph, which is not an NP-complete problem [DilR50], the Left-Edge algorithm has proven to yield the minimum number of registers needed for register allocation [HaSt71].
Multiport Memory Allocation

The previous allocation algorithms assume that register allocation binds the variables into isolated registers or register files. The advantages of using multiport memories as storage devices include more structured design and less chip area. Use of multiport memories as the storage devices in the datapath synthesis was first proposed by Marwedel [MarP84]. Marwedel mentioned the requirement of using multiport memories for simultaneous data access but did not show any methodology for implementing them. Balakrishnan et al. [BalM88] formulated the problem of grouping registers to form multiport memories as a 0-1 linear programming problem (ILP). After variables are assigned to registers, the largest group of registers is found to form a multiport memory. The other multiport memories are formed for the rest of the registers by repeating the same algorithm. Imtiaz and Chen [AhCh91] noticed that Balakrishnan’s algorithm does not result in a minimal number of multiport memories. Instead they proposed an approach to group all the variables into multiport memories simultaneously to minimize the multiport memories. They also formulated the minimization problem as a 0-1 integer linear programming problem which is based on the following criteria:

1. A variable can only be located in one memory, i.e., the variable does not have multiple copies.
2. Each variable must be mapped into a memory.
3. If a group of variables is allocated to an m-port memory, then no more than m of the variables are accessed simultaneously.

After the variables are grouped into several memories, a known register allocation algorithm, such as the Left-Edge algorithm, is applied to assign the variables into registers, and the registers in the same memory share the memory ports. Although Imtiaz and Chen’s method does get the minimal number of multiport memories, the execution time grows rapidly with the number of variables or the number of inequalities. In practice, the ILP approach is applicable to only small problems [GajD92].
Kim and Liu [KiLi93] addressed the problems of minimizing multiport memories and interconnection costs. They tried to minimize interconnection costs first and then multiport memory costs. It is a two-step procedure. First, assign variables to ports of a large virtual multiport memory and then connect the memory ports to functional units. Second, partition the variables and ports to form multiport memory modules. The results are then evaluated by a cost function. Steps 1 and 2 are repeated until the satisfactory cost is achieved.

**Local Memory Allocation**

Another interesting datapath architecture is that in which every functional unit has its own local memory device [LanD94, FraE95]. For the architectures with global registers or multiport memories, which all the algorithms previously addressed are based on, data are loaded to the buses at certain control steps when needed by functional units. For the architecture having functional units with local memories, there is flexibility for scheduling the data communications at some control steps. The exact algorithm for solving the communication scheduling, also known as data routing, was proposed by Frank *et al.* [FraE95] who formulated this problem as a network-flow problem. Although this architecture provides flexibility needed for control circuits to handle data reading and writing, neither the increasing complexities of the datapath and control circuit, nor minimization of the storage devices were addressed.

**Multi-function and Multi-output Functional Unit Allocation**

Attempts at using realistic register transfer (RT) components in functional unit allocation were studied by Ang and Dutt [AnDu94]. In their research, they address the shortcomings of the existing high-level synthesis algorithms that usually assume direct mapping of hardware description language operations to generic elementary RT components. The customized cells, previously designed components, and databook components are not explored. In their paper, an allocation algorithm that attempts to effectively reuse the more complex functional units, such as multi-function and multi-output functional units.
Ang and Dutt’s work focuses on binding behaviors into components that are capable of performing multiple functions and generating multiple outputs. Using a component/cell library with templates for the components, they define the library for multi-function and multi-output components and then show how such a library is used to allocate instances of components for a given behavioral description.

The approach of binding behaviors into functional units is a two phase task. The first phase binds the operations in the CDFG into a register transfer data flow graph (RTDFG). An RTFG is a combination of a global data flow graph and a state transition graph [HulH93] that is used to describe register transfer (RT) functions.

The second phase binds the RT functions into RT structures. An algorithm is used to select a node from the RTDFG as the starting node. For each starting node in the RTDFG, the algorithm binds the node to an already allocated RT unit or a new RT unit depending on a pre-defined component cost function. If the partial solution is not better than an existing solution, the partial solution is discarded. The algorithm is based on a branch-and-bound paradigm that recursively searches the component/cell library for better solutions.

### 2.3 Common Datapath Design Styles

As stated in the previous section, allocation consists of three sub-tasks. The interconnection allocation sub-task refers to connection of the functional units and registers. Interconnection can follow three major styles, which are multiplexer-oriented, bus-oriented, and partitioned-bus. These interconnection architectures will be discussed in this section.
2.3.1 Multiplexer-Oriented

Multiplexer-oriented interconnection architecture is also called **random topology** or **point-to-point** interconnection architecture. In Figure 2.3 the interconnections of the datapath are implemented by point-to-point connections. Each wire connects only two ports (i.e., one input, one output) of the functional units and/or registers. If more than one wire must be connected to the same port, a multiplexer is introduced. The multiplexer is used to select one of the multiple inputs and to transmit the selected input to the output. Point-to-point is the most popular interconnection topology in high-level synthesis because it simplifies the allocation algorithms. Although this method often leads to elegant algorithms, it usually results in unacceptable designs [GajD92]. Multiplexed datapaths have been shown to result in large routing area requirements compared with the bus-oriented architecture because of the large number of nets [Ewe90, MiPa91, MoRo91].

2.3.2 Bus-Oriented

In the Bus-oriented interconnection architecture the variables are transferred via busses. A common structure is shown in Figure 2.5. One of the advantages of the bus-oriented architecture is that the interconnections can be time-shared by different variables, a technique which reduces the routing area requirements. This is especially true if the datapath is designed in a bit-sliced fashion in which the data busses can be routed horizontally on the top of the functional units [FraE95]. Because busses generally connect functional units or registers over long distances, bus drivers are necessary to re-power the signals. More complex control units are also needed to handle the time-sharing of the bus usage. The number of connections can be further reduced if the registers are combined into multi-port register files or memories [BalM88, AhCh91] that support simultaneous data read or write. The architecture shown in Figure 2.5 has also been used by Hwang et al. [HwHs90] to determine the number of busses needed to optimize the datapath. Their experimental results show that even for simple benchmarks, this type of architecture can
easily exceed six or more busses, which is not acceptable for some design technologies [FrLe95a,b].

Figure 2.5  Bus-Oriented Interconnection Architecture.

Ellof [FraE95] pointed out that the large number of busses is caused by the high communication load between the functional units and the registers. He proposed a distributed memory bus-oriented architecture in which each functional unit is attached to a local register file. By using this type of architecture, the variables can be transferred to the functional units more flexibly, thereby reducing the number of busses required.

Another interconnection structure, which is generally used in processors or complex functional units, has the busses partitioned into segments so that different values can be transferred on the same bus simultaneously [EweC90]. Switches are placed between the segments. Simultaneous data transfer is achieved by turning the switches on or off. This type of interconnection can further reduce the number of busses if the operations or functions are allocated to each functional unit properly. A more detailed discussion of the partitioned-bus architecture will be presented in Chapter 4.
2.4 Optimization Approaches

So far the algorithms reviewed in this chapter have been based on either constructive heuristics or Integer Linear Programming. The major disadvantage of using constructive heuristics is that it is very difficult to model and to solve multi-objective optimization problems. It is usually used to tackle a single sub-problem while ignoring the other interdependent sub-problems. High-level synthesis optimization contains many interdependent tasks. If we bias the optimization of one task, other tasks may become impossible to optimize, thereby violating the given constraints. A good example of this is the interdependent relation between performance and resources in the scheduling and allocation tasks. Performance and resources are inversely related to each other in datapath optimization, i.e., we wish to achieve best performance with minimum resources. The constructive heuristic approach is not oriented toward solving this type of problem. Becoming trapped in local optima easily is another pitfall of using constructive heuristic approaches.

Integer Linear Programming (ILP) is an approach commonly used to solve small multi-objective optimization problems. The ILP formulation consists of a set of variables, a linear cost function, and a set of linear equality and inequality constraints. Integer values of the variables are chosen to minimize or maximize the linear cost functions. ILP is able to model the high-level synthesis optimization problems well. Once the ILP model is established, a computer ILP solving application is usually used to search for an optimal solution [HwHu91]. ILP is an effective way to solve small problems, but the number of variables grows exponentially as the problem size grows. Therefore, the computation time for solving the ILP increases rapidly [GajD92]. As a matter of fact, ILP is an NP-hard problem [BeTs97]. Although a variety of heuristic algorithms such as branch-and-bound and Lagrangian relaxation have been applied to find solutions, there are still disadvantages to the ILP approach, i.e., finding the optimal solution is still not guaranteed. Therefore, if we model a problem as an ILP and attempt to find the optimal
solution by solving the ILP, we will find that solving the ILP is an NP-hard problem itself!

2.4.1 Simulated Annealing

Simulated Annealing (SA) is a well developed and widely used randomized, probabilistic optimization algorithm that can include several objectives during optimization and can also provide a mechanism to escape local optima. The local optimum escape mechanism is achieved by not only accepting solutions with improved cost but also by probabilistically accepting solutions with deteriorated costs. Simulated annealing was first proposed by Kirkpatrick *et al.* in 1983 [KiGv83]. They observed that the optimization problem is analogous to the annealing process in physics. A physical material is melted and its minimal energy state is achieved by lowering the temperature so gradually that equilibrium is reached at each temperature. A solution in combinatorial optimization is equivalent to a state in the physical system, and the cost of the solution is analogous to the energy in the state. The objective is to attain a global optimum in which the annealing process attains a minimum energy state for materials.

In order to apply the simulated annealing algorithm, the problem has to be configured as a combinatorial optimization model. A cost function, initial solution, cooling schedule, frozen temperature, and initial temperature must also be established. The core of the simulated annealing algorithm is the Metropolis procedure [Metr53], which is explained in the following.

If the cost of the current solution is $C$ and the cost for a new solution is $C'$, the probability of accepting the new solution is defined as:

$$
\text{Prob(accept)} = \exp \left( \frac{-(C' - C)}{K \times T} \right),
$$

(2.3)
where \( K \) is the Boltzmann constant and \( T \) denotes temperature. If the new cost \( C' \) is smaller than the current cost \( C \), the new solution is always accepted. If \( C < C' \), then a random number is generated in the range between 0 and 1. If the random number is smaller than \( \text{Prob(accept)} \), the solution is accepted.

Simulated annealing starts with an initial temperature, and the Metropolis procedure is called repeatedly during each iteration. The equilibrium state can be said to have been reached for the current temperature when no further improvement occurs for a certain number of iterations. When such an equilibrium state is reached, a cooling schedule is applied to lower the temperature. The algorithm stops when the frozen temperature is reached.

The initial temperature for simulated annealing is set to a high temperature. However, if the initial temperature is set to high, valuable CPU times will be wasted. The ideal initial temperature should be set to a number that both the solutions with decreased and increased costs are all accepted. To address this problem, Wong and Liu [WoLi86] proposed an algorithm that uses the Metropolis function to determine the appropriate initial temperature. The algorithm is described as follows.

Before starting the simulated annealing procedure, a constant number, \( M \), of solutions that are in the neighborhood of the current solution are generated. Each solution is called a move [WoLi86]. The cost difference for each move \( i \), \( \Delta C_i \) is given as

\[
\Delta C_i = C_i - C_{i-1}.
\]

If \( M_u \) and \( M_d \) (\( M_u + M_d = M \)) are the number of improved and deteriorated solutions, the average for the differences of the costs for the improved solutions is given by

\[
\overline{\Delta C_u} = \frac{1}{M_u} \sum_{i=1}^{M_u} \Delta C_i .
\]  

(2.4)
Because we want to keep the probability of accepting the deteriorated solutions high in the early stage, the initial temperature $T_0$ can be estimated by substituting the high probability $P_0$ in the following expression that is derived from the Metropolis function:

$$T_0 = \frac{-\Delta C_u}{\ln(P_0)}, \quad (2.5)$$

$P_0$ can be set a number that is very close to 1, such as 0.999.

In general, simulated annealing can produce good results although the technique suffers from very long runtimes [GajD92].

### 2.4.2 Genetic Algorithms

Genetic algorithms (GAs) [Gold89] are powerful domain-independent search algorithms for optimization problems. In the GA approach, a solution of the optimization problem is encoded as a string of symbols called a chromosome. A number of possible solutions (population) are kept track of. During each iteration or generation, a selection algorithm is used to choose parents from the current population to produce offspring. Two operators, referred to as crossover and mutation, are involved in the reproduction process.

A fitness function must be provided to evaluate the fitness value of each chromosome. The fitness values are used to choose the chromosomes for reproduction. The more fit the chromosome, the higher the probability of its being selected. The iteration process stops when a given time limit is reached or too little solution improvement occurs. Using the GA approach can usually obtain very good results. In contrast with SA, GA is able to achieve better solution qualities in much shorter run times [RaGu95]. It also has been demonstrated to outperform some heuristic algorithms when applied to small problems in high-level synthesis [Dhod95, SaAl96].
Chapter 3

Taxonomy of Complex Functional Units

In this chapter, definitions for Basic Functions (BFs), elementary functions, complex functions, and complex functional units are introduced. Commercially packaged discrete components are surveyed and then categorized based on the functions that they can perform. These functions are then organized in an is-a hierarchy [Booc92]. Although the categorized functions are intended to support the capture of higher level abstractions by HLS tools, another use of the taxonomy is to support a search engine for designers to select components from a component library, which is proposed at the end of this chapter.

3.1 Basic Function

Definition 3.1 Basic Functions (BFs) are the basic operations/functions used in the system specification representation. In other words, basic functions are abstractions that can be used as basic building blocks to capture the system specifications.

To illustrate design capture with BFs, suppose we want to model a circuit to find the leakage resistance per unit length between the inner and outer conductors of a coaxial cable. The cable has an inner conductor of radius $a$, an outer conductor of inner radius $b$, and a medium with conductivity $\sigma$ [CheD89]. The solution can be expressed as:
\[ R = \frac{1}{2\pi \sigma} * \ln\left(\frac{b}{a}\right) \]

\[ = K * \ln\left(\frac{a}{b}\right). \]  \hspace{1cm} (3.1)

If the available BFs include functions \textbf{divide} (\(/\)) , \textbf{multiply} (\((\ast)\) , and \textbf{natural logarithm} (\(\ln\)) , a data flow graph for this computation can be represented as shown in Figure 3.1. The data flow graph is then used by a HLS algorithm to synthesize hardware. For each BF in the data flow graphs, there must be at least one component that can be used to execute the function in the component library. An example is shown in Figure 3.2, which illustrates a common synthesis rule structure that maps functions into components for HLS tools. If the BFs include only elementary arithmetic functions such as \textbf{add} (+) and \textbf{subtract} (-) , substantial efforts are required to model the floating point functions and \(\ln\) by using the existing elementary + and - functions. The data flow graphs would be much more complex.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure3.1.png}
\caption{Data flow graph for Equation 3.1.}
\end{figure}
In the current HLS methodology, only the elementary functions are used to capture the system specifications. In this thesis, more complex functions, which are additions to the set of elementary functions, are proposed for capturing higher levels of abstraction of the system specifications. The definitions of the elementary and complex functions are given below:

**Definition 3.2** *Elementary functions* are functions that are used to capture the logic operations, such as **and, or, nor, not, and xor**, or arithmetic operations, such as **add, subtract, compare** and **multiply**, that are currently used in the HLS.

**Definition 3.3** A *complex function* is a function that requires a sequence of at least two of the elementary functions.

For example, using Taylor series [Trim83], the **Sine** function can be approximated by
\[ \sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!}. \]
If Sine is not available in the system specification capturing methodology, the representation would require 4 multiplies, 2 divides, 1 add, and 1 subtract operations.

Definition 3.4 A complex functional unit can perform at least one type of complex function without external sequencing control signals.

3.2 Function Type Classification

As part of this research, a survey was conducted to collect the basic scheduling functions that have been implemented by off-the-shelf components [IHS98, DATA96]. These functions were then classified into a hierarchy using an is-a relation [Booc92]. The survey was intended to identify those functions with higher levels of abstraction that can be used as the BFs for HLS tools. The functions that have been implemented in packaged discrete components and put into mass production are those that are frequently used by circuit designers. With advancing VLSI technology, these discrete components can be implemented as macro cells or IPs (see Section 1.3) and stored in the component library. In this capacity they can be used by HLS tools to map complex functions in the data flow graph to these macro cells or IPs.

3.2.1 General Function Types

The most general function types that off-the-shelf components can perform are classified as operate, wait, and transfer [Cyre93]. All the other functions are specializations of these three general functions. The hierarchical relationships among these general function types and their sub-types are shown in Figure 3.3. More details about these function types will be addressed in the following sections.
3.2.2 Operate

Operate includes the sub-function types execute, calculate, logical, shift, rotate, map, serialize, parallel, compare, select, and seek. Typical components that are used to implement these functions will now be presented.

execute
The behavior of the execute function can be described as reading structured commands and then performing series of tasks. The components that can be used to implement the execute function are microprocessors, microcontrollers, arithmetic coprocessors, and digital signal processors.

calculate
Calculate is the general function type of add, subtract, multiply, divide, and more complicated computations, such as square root, factorial, logarithm, exponent, trigonometric, and hyperbola. The classified function type hierarchy is shown in Figure
3.4. The components that are used to perform **calculate** range from simple full adders, such as a TTL 74171, to sophisticated application specific arithmetic processors such as Motorola MC6881.

**Figure 3.4** Function type hierarchy for *Calculate*.

**logical**

The **logical** functions included in this category are usually recognized as basic Boolean type operations, such as **and**, **nand**, **or**, **nor**, **exclusive-or**, or **complement**. The components that are used to implement the **logical** functions can be simple logic gates, or complicated microprocessors. The TTL 7400 series are collections of components which provide most of the basic logic functions. Most microprocessors provide **logical** operations which can also be used to perform logic functions. The function type hierarchy for **logical** is shown in Figure 3.5.
**shift** and **rotate**

**Shift** operation provides a quick way of multiplying or dividing an operand by 2. It is also a very important function that is capable of retrieving a particular bit from a byte or a word. The mechanism of rotation is similar to **shift** except that the most significant bit is circulated to the least significant bit or the least significant bit is circulated to the most significant bit. The components that can be used to implement **shift** or **rotate** functions are shift registers, barrel shift registers, or shift counters. Microprocessors or microcontrollers that provide shift commands can also be used to perform these functions.

**map**

The behavior of **map** can be considered as transforming data or retrieving data that uses mapping functions. For example, decoders can be used to convert $n$-bit input formats to $2^n$-bit output formats, and encoders can be used to convert $2^n$-bit input formats to $n$-bit output formats. Seven-segment decoders map binary numbers to decimal representation onto Light-Emitting Diodes (LEDs)/Liquid-Crystal Displays (LCDs). Priority encoders are able to prioritize the input signals for service if more than one input is active.

Digital Signal Processors sometimes are implemented with pre-built lookup tables to **look up** the values for repeated complex computing. **Hash** is another type of table lookup mechanism using hash functions to obtain the entry point of the lookup table and then retrieve data from the table. The capability to **concatenate** two registers is usually important for microprocessors to achieve higher arithmetic precision or to get a more
powerful addressing mode. For example, the accumulator of the Intel 8086 microprocessor can be used separately as AH and AL and they also can be concatenated and used as AX. **Shuffle** is a function that takes two inputs and mixes them. Therefore, the size of the output is the sum of the two inputs. Unlike concatenation, the inputs of **shuffle** may not be distinguished from the output. The function type hierarchy for **map** is shown in Figure 3.6.

```
  map  [convert
       [lookup
            [hash
                [concatenate
                    [shuffle
                        [decode
                            [encode

Figure 3.6 Function type hierarchy for Map.
```

**serialize** and **parallelize**

The functions **serialize** and **parallelize** are usually implemented by components that support data communication, such as programmable peripheral interfaces and universal asynchronous receivers and transmitters (UARTs). Multiple parallel bits are converted into a serial series of signals for transmission. At the receiving end, a device that converts data from serial format to parallel data format must be provided to restore the original data. The behavior of conversion is different from the **map** function. The conversion of the map is with respect to space, which can be achieved without the presence of clocks, while the conversions of **serialize** or **parallelize** functions are done with respect to time for which clock signals are required.

**compare** and **verify**

Determining the control flow of a circuit executing sequences usually requires comparisons between two sets of data. For example, components that are used to implement **compare** functions are often incorporated within microprocessors in order to execute conditional jump instructions. To assure the integrity of data stored on the secondary storage devices, such as floppy disks or hard disks, the components that are used to implement **verify** functions are usually integrated within floppy disk controllers.
or hard disk controllers. This is done in order to read data on the storage devices and to verify those data against previously stored data. The basic mechanism of verifying is comparison between two numbers or two data streams.

**seek and search**
The components that are used to implement **seek** functions are often integrated within disk controllers. This is done so these controllers can force disk drives to move their read/write heads to positions required for reading in the data. When the data required by the central process unit (CPU) is not in memory, a situation sometimes called “page fault” or “cache miss”, the penalty is to retrieve data from secondary storage devices, which costs many clock cycles. Therefore, a good **search** algorithm is important for implementing **seek** functions. Content Addressable Memory (CAM) is a special type of memory that is equipped with a **search** mechanism for fast data retrieval.

**select**
**Select** is a general function for **multiplex, switch, connect, access**, or **address**. Components, such as multiplexers or bus controllers, operate by selecting data on multiple input lines and applying the chosen value to the output. The function type hierarchy for **select** is shown in Figure 3.7.

![Figure 3.7 Function type hierarchy for Select.](image)

3.2.3 **Wait**
The speeds of peripheral devices usually can not keep up with the speeds of microprocessors. For example, if a microprocessor needs data from disks, it has to wait
until the data transfer is completed and then start normal operations. Most of the microprocessors have “no operation” commands which are used to wait for certain events, such as NOP in Intel x86 series of microprocessors. The generation of time delays under software control is very common in microprocessor systems. The disadvantages of using software control are (1) it is difficult to get accurate timing and (2) the process increases the overheads of microprocessors. Some devices such as the Intel 8253 programmable interval timer have been implemented to solve this problem. This interval timer can generate the accurate delays required by the CPU. Shift registers can also be used for fixed delay generation.

### 3.2.4 Transfer

In order for components to execute their designated functions, most of them require input operands. The components then generate output operands. These input and output operands are usually stored in some type of memory device, such as a buffer. These memory devices are used only for transfer purposes and temporary storage rather than on data/operands’ processing. The function type hierarchy for transfer is shown in Figure 3.8.

![Function type hierarchy for Transfer.](image)

**Figure 3.8** Function type hierarchy for Transfer.
receive
The components in this category are implemented to receive data and store it. Examples are memories, stacks, queues, registers, latches, and flip-flops. Based on the different storage mechanisms of these components, their behaviors can also be described as read, fetch, pull, pop, load, send, or input. Receive represents their general behavior.

send
The components in this category are implemented to send out data from their storage elements, such as memories, flip-flops, registers, and latches. Send covers almost the same components as those in the receive category, but the behaviors that describe send components are different. Write, store, put, push, output, transmit, and save can be used to express the mechanism of sending data out.

move
The components in this category are implemented as agents for moving or swapping the data between storage areas. The components that can be used to perform move include DMA controllers, UARTs, floppy disk controllers, and hard disk controllers.

3.3 Application of the Taxonomy

The function taxonomy presented in the previous sections was obtained by surveying commercial components for their functionalities. An application of the taxonomy is in building a search engine for circuit designers and modelers. Such an engine assists circuit designers in choosing components from a component library.

Two typical component libraries are shown in Figure 3.9. It is very difficult for an inexperience designer to tell the functionality of a component by its “part number” such as MC68881. If the components were organized by their functionalities, it would be easier for designers to make component selections. Each function in the function type hierarchy presented in Section 3.2 is mapped to a set of components, as in Figure 3.2. A
graphic user interface was built, which is shown in Figure 3.10, based on the function type hierarchy to guide designers to select the components with the desired functionalities [LiCy96].

Figure 3.9 Component retrieval interfaces (a) WorkView (b) SYNOPSYS.
Figure 3.10 (a) Main menu with cascaded sub-menus of the component retrieving interface (b) List of components with their corresponding parameters and characteristics (c) Detailed information of the retrieved component.
Chapter 4

Design Styles

As stated in Chapter 2, high-level synthesis is a process of translating a behavioral description of the system specification into a structural description. The design styles discussed in Section 2.3, such as multiplexer- and bus-oriented architectures, are mainly developed for HLS optimization algorithms to translate the behavioral description having only elementary functions to target architectures with simple elementary functional units. Such architectures are not suitable for complex functions realized with complex functional units. This is due to the wide differences in execution delays and the sequential nature of complex functional units.

Definition 4.1  Design style is a term that refers to the principal qualitative features of a design, such as snooping data cache, bus-oriented datapath, or distributed memory device [GajD92].

Definition 4.2  Target architecture defines a design more precisely in terms of particular functional units, storage devices, and interconnections [GajD92].

In this chapter, Section 4.1 addresses the way in which the elementary and complex functions affect design styles. The rest of this chapter presents details of a design style proposed for HLS tools to optimize a system that contains complex functions.
4.1 Fixed vs. Bounded Delays

The general approaches of high-level synthesis that only exploit elementary functional units were discussed in Section 2.1. These approaches include steps such as data representation, scheduling, and allocation. One of the assumptions for current HLS tools that perform optimization during the scheduling and allocation steps is that the execution delays for each function are fixed. For example, simple functions such as add or subtract take 1 clock cycle to execute for elementary functional units such as adders or ALUs. Some functions, such as multiply, may take a few more cycles to execute, but they are still assumed to be fixed delays.

The design styles surveyed in Section 2.3 are used as the target architectures for current HLS tools, which assume that only simple functional units are used to realize elementary functions. The target architectures are optimized for the elementary functions that have fixed delays. However, execution delays for complex functions are bounded and usually take hundreds or thousands of clock cycles. Furthermore, the bounded delays are unknown during synthesis. They can only be determined after the design is implemented and then measured at the run time.

**Definition 4.1** A complex function has a bounded execution delay, \( d \), that is bounded between best case delay, \( d_{\text{min}} \), and worst case delay, \( d_{\text{max}} \), which can be expressed as \( d_{\text{min}} \leq d \leq d_{\text{max}} \). The exact delay is unknown during synthesis; and the exact delay can only be known at run time.

For example, the execution delay for function MOD (partial remainder) is given as 15 to 190 clock cycles, which are best case and worst case delays respectively, for an INTEL 8087 mathematics coprocessor [INTE91]. The exact delay depends on the values of the operands.
The multiplexer- and bus-oriented target architectures were proposed by prior researchers for synthesis with fixed or worst case delays. Therefore, the delays for complex functions have to be set to the worst case delays to use the current HLS optimization algorithms. The major disadvantage of fixing the delays in this way is that the functional units will be under-utilized. A better design style and optimization algorithms that exploit bounded delays are needed to tackle the problem. Such a design style will be proposed in the following sections, and the issues of optimizing such a target architecture will be addressed in Chapters 5 and 6.

4.2 Advantages of the Partitioned-Bus Architecture

The partitioned-bus architecture, first proposed by Ewering [EweC90], is derived from a bus-oriented architecture. The concept of partitioning a bus-oriented architecture was adopted later by several researchers [FrLe95, Fran95, Mont91, KiCh97]. Some other similar architectures are also reported [DeMH86, LanD94, DuHe93]. In this architecture the buses are partitioned into a number of segments, and each functional unit has its own local register file. The architecture is shown in Figure 4.1. The advantages of this structure include

1. transferring different values on the same bus at the same time but on different segments, which can reduce the number of buses, and
2. the bus drivers do not always have to drive the entire bus, which can reduce the power dissipation and transmission delays.

Another major advantage is that the interconnection can be generated systematically. Ewering showed that the areas of the synthesized circuits can be created 50% smaller than with the multiplexer-oriented architectures. This dramatic savings in area is due to the orderly interconnection structure.
Despite the above advantages, there are some disadvantages. Control units, for example, are more complex due to the control signals for bus switches. Therefore, it is more complicated to develop optimization algorithms. The optimization approaches or algorithms that have been developed for this type of architecture include Integer Linear Programming (ILP) [FrLe95], cluster partitioning [Ewe90], and Genetic Algorithms [KiCh97]. These are used to (1) minimize bus usage, (2) minimize register file sizes, (3) minimize the number of functional units, (4) minimize the number of ports for the local register files, and (5) optimize functional unit placement.

![Generic partitioned-bus architecture](image)

**Figure 4.1** Generic partitioned-bus architecture.

Although the partitioned-bus architecture was mainly developed for HLS tools with simple functional units, it is also suitable as the target architecture for synthesis with complex functional units because of the following advantages:

1. Storage devices are distributed in each functional unit as local register files. Global communication is needed only when a functional unit requires data generated by other functional units.
2. Functional units can be organized into \textit{loosely-coupled} configurations [LiGi86]. It is noted that these types of configurations are \textit{asynchronous} circuits, and their functional units have signals such as, END or READY, to indicate the completion of a function execution. Since each functional unit has its local storage devices, it is possible for different functional units to operate at different clock rates. Performance can be improved by assigning functions having many computational steps to functional units running at higher clock rates.

For example, in Figure 4.2, the delays for functions 1, 2, and 3 are 184, 168, and 800 clock cycles respectively. Suppose functions 1 and 2 are assigned to functional unit 1, and function 3 is assigned to functional unit 2 for execution. If both functional units 1 and 2 are running at clock rate of 2 Mhz, the system delay for Figure 4.2 is dominated by the execution delay of function 3, which is $800 \times 1/2\text{Mhz} = 800 \times 500\text{ns} = 400\text{ms}$. If a functional unit that is capable of running at 4 Mhz is used to execute function 3, the system delay is $800 \times 1/4\text{Mhz} = 800 \times 250 = 200\text{ms}$, which is a 50% reduction in system delay.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure42.png}
\caption{A CDFG with denoted functional unit allocation.}
\end{figure}
4.3 Revised Partitioned-Bus Architecture

In previous partitioned-bus research, the generic architecture, which is shown in Figure 4.1, is focused on synthesis with elementary functions. An improved version of this architecture that supports synthesis with complex functions better is shown in Figure 4.3. The major difference between the previous generic and the proposed partitioned-bus architecture is in the organization of the storage devices. For the proposed architecture, there are two types of register files, local register file and buffer, which address local and global variables respectively.

The proposed partitioned-bus architecture consists of a number of parallel buses that are divided by switches into several segments. The switches allow or inhibit data flow from one segment to another. Each switch is independently controlled. The structure can be defined as the Partitioned-bus structure \((B, S)\), where \(B\) is the number of buses and \(S\) is the number of segments on each bus. Note that all buses have the same number of segments, i.e., \(S\). The following notations or abbreviations will be used throughout this thesis:

- \(B_i\) is the \(i^{th}\) bus; \(0 < i \leq B\).
- \(B_{ij}\) is the bus segment \(j\) of the bus \(i\); \(0 < j \leq S\)
- \(S_{ij}\) is a switch between bus segment \(B_{ij}\) and \(B_{i,j+1}\)
- \(P_j\) is the set of all \(B_{ij}\) and is called a cross-section; \(i = 0, 1, \ldots, B\).
- A functional unit module \(FUM_j\) is attached to partition \(P_j\).

**Definition 4.2**  A functional unit module contains a complex functional unit, a local register file, a buffer, and internal control logic.

The structure of the functional unit module can be found within the dashed lines of Figure 4.3.
4.4 Operations on the Proposed Architecture

The proposed partitioned-bus architecture consists of bus segments and functional unit modules that attach to each partition. Complex functional units are used to perform the execution of complex functions. Buffers are used to store global variables that are transferred from other functional unit modules. Local register files are used to store local variables that are generated and consumed by the same functional unit. Local variables and global variables are defined as follows:

**Definition 4.3** A local variable is a variable that is generated and consumed by the same functional unit.
**Definition 4.4** A *global variable* is a variable that requires transfer through at least one bus switch. The transfer is required because the variable is used as an input operand by the other functional units.

The variables generated by functional units can be output to local register files or transferred to other functional units, or both. If the variables are used as input operands by the other functional units but are not needed by the functional unit that generates them, the variables are transferred to the target functional unit modules’ buffers. If the variables are needed both by the local functional unit, i.e. the functional unit that generated them, and the other functional units, the variables are sent to both the local register file and the target functional units’ buffers. There is control logic in front of each buffer which is used to select the register for storing the variables transferred from other functional units.

The two multiplexers in front of the functional units are used to select the sources of the input operands from local register files or buffers. Between the functional unit modules and the bus segments, there are tristate buffers. The tristate buffers are used to select a bus segment for sending the global variables that are to be transferred to other functional unit modules. Before transferring a global variable between functional unit modules, a bus has to be chosen and the switches have to be turned on to establish a communication channel by the control unit.

**Example 4.1**

This example is used to demonstrate how the proposed architecture works. The given dataflow graph is shown in Figure 4.4 (a). Suppose functions 1, 2, and 3 are assigned to Functional Unit 1 (FU_1) and function 4 is assigned to Functional Unit 2 (FU_2) for execution. The bounded delay for floating point multiplication (\( * \)) is 146-168 clock cycles, and for floating point addition (\( + \)) is 54-368 clock cycles [INTE91]. Initially, The variables \( a, b, \) and \( e \) are pre-loaded in the buffer of functional unit module \textbf{FUM}_1, and variables \( h \) and \( i \) are in the buffer of \textbf{FUM}_2, which is shown in Figure 4.4 (b).
Figure 4.4 (a) A data flow graph.
(b) The pre-loaded variables in FUM1’s and FMU2’s buffers and local register files.
(c) A snapshot of possible buffer and local register file contents before FU_1 executes function 2.
(d) A snapshot of possible buffer and local register file contents after FU_1 executes function 2.

Figure 4.4 (c) is a snapshot of the possible contents of the buffers and local register files of the two functional unit modules. The snapshot shows that variable c is output from FU_1 and stored in its local register file, and that variable f is generated by FU_2 and transferred to FUM1’s buffer. The contents of the buffers and local register files are
possible in the snapshot shown in Figure 4.4 (c) only when FU_1 finishes execution of function 1 before it begins executing function 2, and FU_2 finishes executing function 4. Because the execution delays for each function may vary in different runs, the times in which the variables are generated and stored in the buffers or local register files are uncertain.

Figure 4.4 (d) shows the contents of buffers and local register files after FU_1 executes function 2. The variable f in buffer and variable d in the local register file of FUM1 are used as input operands by FU_1 when it executes function 3.

4.5 Control Unit

After the datapath is synthesized, a control unit has to be implemented to sequence the FUM operations and switch the buses. Since the CDFG provides the control flow and the datapath provides the precise structural information, a finite state machine (FSM) can be modeled based on the CDFG and the datapath structure [GajD92]. The synthesis of control units has been the subject of extensive investigation by many researchers for several decades. It is usually called sequential synthesis [MicG94, MccE86]. The common approaches of sequential synthesis optimization include FSM modeling, state minimization, and state encoding. A state transition diagram or state transition table is usually used for FSM modeling. The optimized FSM is then mapped into logic circuits which consist of three major parts:

1. control logic that generates the control signals for the components in the datapath,
2. a state register to keep track of the current state, and
3. the next state logic that generates the next state by using the status signals input from the datapath and the content of the current state register [GajD92].

As stated in chapter 2, there are three types of implementation style for control units:
1. random logic,
2. programmable logic array (PLA), programmable array logic (PAL), field programmable gate array (FPGA), and
3. Read Only Memory (ROM).

If speed is the major concern, random logic and PLA implementations are the best options. For example, if the functional units in the datapath are only elementary functional units that take single or several clock cycles to execute a function/operation, the control unit has to meet the speed constraints, and random logic or PLA design styles are implemented mainly for speed considerations. For synthesis with complex functional units in which the execution delays usually take hundreds or thousands of clock cycles, use of the ROM implementation style is adequate.
Chapter 5

Issues in Complex Synthesis

In this chapter, the issues of translating the behavioral representation of the system specification into the proposed partitioned-bus architecture will be addressed. The issues focus on the functional partitioning and the metrics that are used to estimate the quality of the partitioning. The functions partitioned in the same block are assigned to the same complex functional unit for execution. Therefore, the number of complex functional units is equal to the number of blocks. By using the number of blocks as the constraint for functional partitioning, the metrics used to measure the partitioning quality include:

- the number of registers in the communication buffer,
- the number of registers in the register file,
- system delay,
- the number of buses,
- the number of links [DeNe89], and
- the number of multiplexers.

The complexity of the functional partitioning and the approaches proposed to address the problem are discussed in Section 5.1. Sections 5.2-7 present the algorithms that are used to address the metrics of partitioning quality in the complex synthesis. In Section 5.8, a cost function is formulated to evaluate the quality of the partition result.
5.1 Functional Partitioning

If the given system specification is represented by a Control Data Flow Graph (CDFG), functional partitioning is the task of partitioning the functions in the Control Data Flow Graphs into several blocks. Functions in the same block are mapped to the same functional unit for execution. If the Control Data Flow Graph $G = (V, E)$ consists of a set of functions which are represented by vertices $V = \{v_i | i = 1,2,..m\}$, and a set of data dependencies which are represented by edges $E = \{e_{ij} | e_{ij} = (v_i,v_j), \; v_i, v_j \in V\}$, the definition of functional partitioning is given as follows:

**Definition 5.1** The problem of functional partitioning is to partition $V$ into two or more interacting blocks, which is expressed as $P = \{P_i | i = 1,2,\ldots,N\}$, where $P_i = (V_i, E_i)$, $\cup V_i = V$, and $V_i \cap V_j = \emptyset$ if $i \neq j$.

For example, the functions in Figure 5.1 are partitioned into two blocks, which is expressed as $P = \{P_1, P_2\}$, where $P_1=\{1,3\}, \{a,b,c\}$, $P_2=\{2,4\}, \{a,b,c\}$.

![Figure 5.1 A CDFG with two partition blocks.](image-url)
For partitioning algorithms to perform functional partitioning, constraints, such as the number of functional units, the number of buses, or the clock speed, must be specified, and a cost function has to be established to evaluate the quality of partition results. The cost function may consist of several metrics, such as the metrics given on pages 77 and 78, which are used to estimate costs for the partition results. The cost functions are usually in the weighted-sum format that can take the varying importance of each metric into account. For example a cost function may be given as:

\[
\text{Cost} = W_1 \cdot (\text{number of registers}) + W_2 \cdot (\text{number of buses}) + W_3 \cdot (\text{number of multiplexers}),
\]

where \( W_1, W_2, \) and \( W_3 \) are the weights for adjusting the importance of the number of registers, the number of buses, and the number of multiplexers.

From the given CDFG, the constraints, and the cost function, a partitioning algorithm is used to search for the best partition, i.e., the partition with the lowest cost. For finding the lowest cost partition, an exhaustive search can guarantee the best result, but this approach is not practical even for small problems. For example, if we want to perform a four-way partition on a CDFG with 30 functions, the exhaustive search would require evaluation of \( 4^{30} \) partitions! As a matter of fact, partitioning is an NP-complete problem [GajD92], and there are many algorithms proposed to search for good partitions.

Partitioning algorithms are generally classified into two classes, which are known as constructive and iterative approaches [GajD92]. For the constructive approach, the algorithm starts grouping the functions gradually into the same blocks and ends at a complete partition. A closeness metric is used to evaluate the merits of assigning the functions into various blocks. For the iterative approach, the algorithm starts from a complete partition and then modifies the partition repeatedly in the hope that the partition result will be improved.
5.1.1 Constructive Approach

Hierarchical clustering is a commonly used constructive partitioning approach [LaTh91, McKo90, GajD92]. Initially, each function is initialized as a block, and closeness values are calculated for every pair of functions. The functions with high closeness value are merged into a new function. Closeness values are then recomputed for the new function paired with every other function. This function merging and closeness value computing continue until the stop criteria are reached, which can be when the functions are merged into a certain number of blocks.

Closeness values for every pair of functions are usually computed by counting the number of common edges (data dependencies) between them [GajD94]. The functions with high closeness values are to be merged because the inter-functional unit communication cost can be reduced when these functions are mapped to one functional unit for execution. Another commonly used method is proposed by McFarland [McKo90], which takes into account the number of each function’s input and output edges and the number of common edges between function pairs. The closeness value is computed by dividing the total edges of the two functions by the number of their common edges.

For the hierarchical clustering approach, only one closeness function is used to calculate closeness values throughout the partition process. For the Multi-stage clustering approach, several closeness functions are used [LaTh91]. An intermediate partition result, called a stage, is obtained by merging the functions according to closeness values calculated by a closeness function. A different closeness function can be used to compute closeness values of the intermediate partition result thereby generating a new stage. Stop criteria must also be provided for the Multi-stage clustering approach.
5.1.2 Iterative Refinement Approach

Before applying the iterative refinement approach for functional partitioning, an initial partition has to be established. The initial partition can be constructed by using constructive approaches or random mapping. The iterative approach takes the initial partition and then moves the functions among different blocks to improve the partition results. **Min-Cut** is a widely used iterative approach partitioning algorithm [KeLi70] because it can usually produce good results in short run times.

Min-Cut was originally developed for two-way partitioning. This algorithm exchanges the functions between the two blocks repeatedly to produce a maximal partitioning improvement. A cost function must be provided to evaluate the costs for the partition. A **gain** is defined to be the difference of costs before and after exchange of two functions. The exchange itself is referred to as a **move** between the two blocks. In each iteration, the algorithm starts by computing the gains from moving every pair of functions. The move with the maximum gain is accepted, and the moved function pair is then **locked**, meaning that they are not allowed to be moved in the following iterations.

The maximum gain from each iteration is stored. If the interchange of the first $k$ pairs of functions can achieve the maximum gain, then move the first $k$ pairs of functions, and the partition is saved as the final partition result. The two-way Min-Cut partitioning algorithm can be extended to multi-way partitioning by repeatedly performing two-way partitioning on the previous partitions until the number of desired blocks are obtained.

**Simulated Annealing** has been applied to the partitioning problem in a high-level synthesis environment [GajD92, DeNe89]. Devades and Newton stated that the greedy approaches used by the constructive partitioning algorithm may lead to local optimal solutions [DeNe89]. To avoid getting stuck in the local optima, a hill-climbing mechanism which may accept negative gains has to be provided during each partitioning iteration. The simulated annealing usually generates very good partition results but suffers
very long run times. The simulating annealing algorithm was discussed in Section 2.4.1. **Genetic Algorithms** can also be applied to address the partitioning problem, which will be discussed in the next chapter.

### 5.2 Communication Buffers

For the sake of simplicity and clarity, the term **buffer** indicates a set of registers used for communication between two blocks, and the term **communication buffer** denotes the union of all of the buffer sets. The number of registers in a buffer will be referred as **buffer size**, and sum of the buffer sizes is called the **communication buffer size**. The notations for buffer size and the communication buffer size will be introduced in Section 5.2.1. In this section, an algorithm is proposed to estimate the communication buffer size.

In the prior functional partitioning research [GajD92, GuCo92, KeLi70], the size of the edge cut-sets is usually used to estimate the communication cost. Since the edge-cut set size represents the inter-functional unit communications, it is desirable to reduce communications in order to minimize communication delays and their related inter-functional unit interface costs. In the proposed partitioned-bus architecture, the inter-functional unit communications are through buffers. Therefore, the numbers of registers in the buffers are used as a metric to estimate the communication cost [AgGu97].

However, estimating buffer sizes is not an easy task because complex functions have bounded delays and the inter-functional unit communication delays are unknown. The bounded execution delays and the unknown communication delays imply global variable lifetime uncertainties, indicating that the buffer sizes cannot be accurately estimated by the Left-Edge algorithm [KuPa87].
5.2.1 Problem Definition

The communication buffer size, $R$, can be expressed as:

$$R = \sum_{j=1}^{N-1} \sum_{i=j+1}^{N} (|b_{ij}| + |b_{ji}|),$$  \hspace{1cm} (5.1)

where $b_{ij}$ and $b_{ji}$ are the buffers required to support the two-way communication between block $i$ and $j$ of a partition, and $|b_{ij}|$ and $|b_{ji}|$ are denoted as the buffer sizes.

The buffer sizes between blocks can be estimated by tracing the data flows in the CDFG. The buffer size estimation algorithm is a two-step process that includes labeling the edges of the CDFG with path vectors (PVs) and then transforming the edges into a compatibility graph. The compatibility graph is then used to find the upper bound on the buffer size.

5.2.2 Path Vectors

Labeling the edges with path vectors is the method used to detect variables with non-overlapping lifetimes. Variables with non-overlapping lifetimes can share the same register in a buffer, which leads to buffer size reduction. For example, in Figure 5.2

![Figure 5.2](image)
functions 1 and 3 are assigned to block P1 and functions 2 and 4 are assigned to block P2. Variables a and c are non-overlapping because of the data dependencies among functions 1, 2, 3, and 4. Variables a and c can share a register in buffer b_{12}. Variable b is transferred from block P2 to block P1. A register is needed in buffer b_{21} to store variable b. Therefore, by using Equation 5.1, the buffer sizes for |b_{12}| and |b_{21}| are both 1, and R is 2, which is the sum of |b_{12}| and |b_{21}|. If edge cut-sets, C_{12} and C_{21}, are used to estimate the buffer sizes |C_{12}| and |C_{21}|, which are 2 and 1 respectively, three registers are needed, which is an over-estimate.

A path vector is a bit vector. The dimension of a path vector is the number of paths in the CDFG. The paths in the CDFG can be found by using all entry nodes, which are the nodes without predecessors, as the roots, and then performing Depth-First Search. For example, in Figure 5.3 (a), the roots for this CDFG are nodes 1, 2, 3, 4, and 5. After using the roots to find all the paths, which are shown in Figure 5.3 (b), each path is represented by a one-hot bit vector. For example, [00001] is used to represent path 1. All the edges that belong to that path are labeled with the same path vector. If an edge is traversed by more than one path, the edge is labeled with the bit-wise OR of the path vectors. For example, in Figure 5.3 (a), Edges f and h are traversed by paths 1 and 2 so that edges f and h are labeled with a path vector [00011].

**Definition 5.2** The path vector for an edge is used to represent paths that traverse the edge. The formal notation of the path vector for an edge is denoted as PV(e_{ij}), where e_{ij} ∈ E.

The path vectors are then used to determine whether the lifetimes of two variables may overlap. Two variables are lifetime non-overlapped if the bit-wise AND of their path vectors is not a zero vector. If the result of bit-wise AND of two path vectors is a zero vector, then the two variables are potentially lifetime overlapped.

For example, in Figure 5.3, the bit-wise AND of the path vectors of edges a and f is [00001], which is not a zero vector and it indicates that the lifetimes of a and f are not
overlapped. The bit-wise AND of the path vectors of edges \( a \) and \( b \) is a zero vector, which suggests that the lifetimes of \( a \) and \( b \) may overlap.

![Diagram of CDFG with labeled path vectors](image)

**Figure 5.3** CDFG with labeled path vectors.

### 5.2.3 Buffer Size Upper Bound

After all the edges in the CDFG are labeled with path vectors, compatibility graphs can be constructed to estimate the upper bound of the buffer sizes. In the constructed compatibility graphs, the nodes represent the variables of the given CDFG, and the edges represent the *compatibilities* of the variables. For example, if two variables can share the same register in a buffer, they are said to be *compatible* and there is an edge between them in the compatibility graph.

For two partition blocks \( P_i \) and \( P_j \), the edge cut-set, which is used to represent the variables generated by the functions in \( P_i \) and consumed by the functions in \( P_j \), is denoted as \( C_{ij} = \{ c_k | c_k = (u,v), \ u \in V_i, \ v \in V_j \} \). The compatibility graph for \( C_{ij} \) is constructed as follows:
For every edge pair \((c_i, c_j)\), where \(c_i, c_j \in C_{ij}\) and \(c_i \neq c_j\), if \(PV(c_i) \text{ AND } PV(c_j) \neq [0]\), then introduce an edge between \(c_i\) and \(c_j\).

For example, Figure 5.4 (a) shows a two-way partition for Figure 5.3. The edge cut-set and the path vectors for \(C_{12}\) are shown in Figure 5.4 (b). The compatibility graph for \(C_{12}\) is shown in Figure 5.4 (c).

**Figure 5.4**  (a) A two-way partitioning for Figure 5.3.  
(b) Edge cut-set and Path vectors for \(C_{12}\).  
(c) Compatibility graph for \(C_{12}\).

The buffer size upper bound for buffer \(b_{ij}\) is then transformed into a clique partition problem, which searches for the minimal number of cliques to cover the compatibility graph. Finding a clique partition is an NP-complete problem [MicG94]. An effective
heuristic algorithm proposed by Tseng and Siewiorek [TsSi86] may be used to find a good clique partition for the buffer size estimation algorithm. Analogously, the buffer size for $b_{ji}$ can be obtained by applying the above procedure, and the buffer size upper bound for block $i$ and $j$ is equal to $|b_{ij}| + |b_{ji}|$. The communication buffer size is attained by computing the buffer sizes of each block pair and then adding them up, which is expressed in Equation 5.1. The algorithm for the communication buffer size estimation is shown in Figure 5.5.

5.2.4 Complexity Analysis

The buffer size estimation algorithm contains two major sub-algorithms. The first sub-algorithm is the Depth-First Traversal algorithm. The complexity of the Depth-First Traversal depends on the data representation of the CDFG [TeLa90]. If the CDFG is represented as an adjacency list, which is adopted in the program implementation of this thesis, the complexity is $O(n + e)$, where $n$ is the number of nodes and $e$ is the number of edges. Because $e$ is usually greater than $n$, the complexity of Depth-First Traversal is often considered as $O(e)$ [TeLa90].

The second sub-algorithm is the heuristic clique partitioning algorithm proposed by Tseng and Siewiorek. Their algorithm includes processes for selecting pairs of nodes with the maximum number of common neighbors. The algorithm then merges these nodes into a super node. All the incident edges of the merged nodes are deleted, and new edges are added to their common neighbors. The algorithm repeats these processes until all the edges in the compatibility graph are deleted. For the worst case, in each iteration, there are $C(n,2) = (n^2/2 - n/2)$ ways to choose a pair of nodes to merge, which happens when there are edges between every two nodes, i.e., a complete graph. Therefore, in each iteration, the complexity is $O(n^2/2 - n/2)$, which can be simplified to $O(n^2/2)$, where $n$ is the number of nodes. $n-1$ iterations are needed for the worst case, which happens when the compatibility is a complete graph, so that the complexity of the heuristic partitioning algorithm is $O((n-1)n^2/2)$. 

71
**Algorithm** Communication_Buffer_Size_Estimation

// initialize data
Ne = |E|; // number of edges in the CDFG
So = set of entry nodes; // set of nodes that don’t have predecessors in the CDFG
Si = set of exit nodes; // set of nodes that don’t have successors in the CDFG
Path = 1;
N = number of blocks; // N-way partitioning
For each edge i, i ∈ E;
    PV(i) = [0]; // [0] is denoted as a 0 bit vector
End For;
// label the edges with path vectors
For each node i, i ∈ So;
    use i as root to perform Depth-First Traversal;
    if node j is visited ; j ∈ Si; then
        Path = Path + 1; // a path from Node i to Node j is found
        Label each traversed edge e, with PV(e) = PV(e) | To_Bit_Vector (Path) ;
        // '|' is denoted as a bit-wise OR operation
        // To_Bit_Vector() is a function to convert an integer to a bit vector
    End if;
End For;
// construct compatibility graphs for all block pairs and then do clique partitioning
For each block pair Pi and Pj
    Cij = Get_Cut_Set(Pi,Pj); // Cij is the edge cut-set between block Pi and block Pj
    For each edge pair ci and cj ; ci, cj ∈ Cij and ci ≠ cj
        if ( PV(ci) & PV(cj) ≠ [ 0] ) then // ‘&’ is denoted as a bit-wise AND operation
            introduce an edge between Node ci and Node cj
        End if;
    End For;
    |bij| = Get_Clique(Cij); // Get_Clique() is a clique partition algorithm proposed by
    // Tseng and Siewiorek, and it returns the number of cliques
    repeat the above steps for Cji in order to obtain |bji|
    Buffer_Size = Buffer_Size + ( |bij| + |bji| ) ;
End For;
Return Buffer_Size;
End Communication_Buffer_Size_Estimation.

**Figure 5.5** Algorithm for Communication Buffer Size Estimation.

For N-way partitioning 2C(N,2) compatibility graphs have to be established. The clique partitioning algorithm must then be applied 2C(N,2) = (N^2 - N) times. The complexity becomes O((n-1) n^2/2(N^2 - N)), which can be simplified to O(n^3N^2/2 ).
From the above complexity analyses, the algorithm for estimating complexity of the communication buffer size can be expressed as $O(e + n^3N^2/2)$, which allows us to obtain the communication buffer size estimation in polynomial time.

### 5.3 Register Files

The variables generated and consumed within a functional unit are stored in the functional unit’s local register file. The size of the local register file, which is referred to as the number of registers in the local register file, for each functional unit can be estimated by an algorithm which is similar to the communication buffer size estimation algorithm. The only difference is the edges that start and end in the same block are used to establish the compatibility graphs. For the sake of clarity in the following discussion, register file is the union of the local register files, and the register file size is referred to as sum of the numbers of registers in each local register file. The register file size, $L$, can be expressed as:

$$L = \sum_{i=1,N} |f_i|,$$  \hspace{1cm} (5.2)

where $|f_i|$ is the local register file size for functional unit $i$ which executes the functions in block $i$.

The local register file size estimation algorithm takes the CDFG with labeled path vectors as an input. A compatibility graph is constructed. If two variables are compatible, their lifetimes are not overlapped so that they can share the same register. There is an edge between these two variables in the compatibility graph. If there is no edge between two variables in the compatibility graph, the lifetimes of these two variables may be overlapped and they cannot share the same register. These two variables are said to be not compatible. For example, if functions $4$, $5$, $8$, and $9$ in Figure 5.3 are assigned to the same block, the path vectors for edge $e$ is $[10000]$ and for edge $d$ is $[01000]$. Because the bit-
wise AND of these two bit-vectors is a zero vector, local variables e and d are not compatible. There is no edge between e and d in the compatibility graph, which means that their lifetimes may be overlapped. Because of the possibility of lifetime overlapping, variables d and e cannot share the same register. Therefore, two registers are required.

After the compatibility graph for each partition is constructed and the clique partitioning algorithm is applied [TeSi86], the size of the local register file for each partition can be obtained. The size of the register file is the sum of each functional unit’s local register file size, which is defined in Equation 5.2.

5.4 System Delays

After each function in the CDFG is assigned to a block, the functions in the same block are to be executed by the same functional units. The delay for each functional unit is the sum of the functional execution delays, plus the time during which the functional unit is idle due to the data dependencies among the functions in the CDFG. A functional unit may have to wait for the required operands generated by the other functional units so that the functional unit can resume/start execution. The system delay, \( T \), is, therefore, the maximum delay over the functional unit delays. This relationship can be expressed as:

\[
T = \max_{j=1,N} \left( \sum_{i=1,|V_j|} T_i + \text{idle}(j) \right), \quad (5.3)
\]

where \( T_i \) is the execution delay for function \( v_i \), \( N \) is the number of blocks for \( N \)-way partitioning, and \( |V_j| \) is the number of functions in block \( j \).
Partition results have great impact on the system delay because of data dependencies among the functions. For example, in Figure 5.6, if functions $1$, $2$, $3$, and $4$ have the same execution delays, partition result (a) is preferred even though both of them have the same communication buffer size. Case (b) requires a longer execution delay, due to the data dependencies among the functions, and the partition blocks the functions are assigned to. Functions $1$ and $4$ both need variable $y$, which is the output of function $3$, as an input operand. In (a), functions $1$ and $4$ are in different blocks and can be executed by different functional units concurrently as soon as variable $y$ is available. In (b), functions $1$ and $4$ are in the same block. Functions $1$ and $4$ have to wait until variable $y$ is ready and then are executed sequentially by the same functional unit. Therefore, as shown in Figure 5.6, the delays for (a) is 2 clock cycles, and for (b) is 3 clock cycles.

The estimation of system delay can be treated as a resource-constrained scheduling problem [GajD92]. The number of blocks represents the available number of functional units. The scheduling algorithm tries to assign each function into a functional unit such that a schedule having minimum system delay is obtained. Resource-constrained scheduling is a known NP-complete problem. A heuristic scheduling algorithm, List scheduling, which is discussed in Chapter 2 can be used to estimate the system delay.
The priority function used by the List scheduling will be addressed in Chapter 6.

5.5 Buses

Variables can share the same bus if they are not transmitted at the same time. This principle, used to estimate the number of buses, is referred to as bus sharing in the temporal domain. For the proposed partitioned-bus architecture, the variables can also share the same bus at the same time if they are transferred on the non-overlapped bus segments, a principle referred to as bus sharing in the spatial domain. The bus sharing in these two domains will be presented in Sections 5.5.1 and 5.5.2 respectively, followed by an algorithm for estimating the number of buses needed.

5.5.1 Bus Sharing in the Temporal Domain

Buses are used to transfer variables among the functional unit modules. Variables can share the same bus if they are not transferred at the same time. If two variables can share the same bus, they are said to be compatible. A compatibility graph can be established to show bus sharing possibilities among the variables, and then the graph is input to the clique partitioning algorithm to estimate the number of buses needed [TiSe86].

Three methods are proposed for finding compatibilities among the variables:

1. tracing data dependencies in the CDFG,
2. tracking the blocks in which the variables are generated, and
3. identifying the variables that can be transferred on the same bus concurrently but over non-overlapped bus segments.
The first method and second will be illustrated and discussed with an example in this section because they look for compatibility in the temporal domain. The third one will be addressed in the next section, since it is used to find compatibilities in the spatial domain.

Figure 5.7 is a four-way partition for the CDFG shown in Figure 5.3 (a), in which variable c is to be transferred from block P1 to P4. Variables a, b, and d are to be transferred from block P2 to P3. Variable f is to be transferred to from block P3 to P4, variables g and h are to be transferred from block P4 to P3, and variable e is to be transferred from block P3 to P2.

Due to the data dependency between variables a and f, it is not possible to transfer variables a and f at the same time. Variable a has to be transferred to block P3 before variable f is generated. Variable a is then used as one of the input operands for function 6 to generate variable f. Therefore, variables a and f can use the same bus for transfers.
From Figure 5.3, a data dependency exists among variables \( a, f, \) and \( h \). Data dependencies also exist among variables \( b, f, \) and \( h \), and between variables \( c \) and \( g \). Therefore, the variables \( a, f, \) and \( h \) can use the same bus for transferring. Likewise, variables \( b, f, \) and \( h \), and variables \( c \) and \( g \) can also use the same bus for transfers.

The data dependencies among the variables can be identified by applying the bit-wise AND to their path vectors (See Figure 5.3). If two variables are data dependent, the bit-wise AND of their path vectors will be a non-zero bit vector. The bit-wise AND results for each pair of variables are used to construct a compatibility graph. For every two variables, an edge is formed if the bit-wise AND of their path vectors is not a zero-vector, i.e., if they are data dependent, an edge is formed between these two variables in the compatibility graph. The compatibility graph, \( G_d \), based on the data dependencies of Figure 5.7 is shown in Figure 5.8.

![Compatibility Graph](image)

**Figure 5.8** The compatibility graph based on the data dependencies of Figure 5.7.

In addition to that data dependencies can be used to identify variables which cannot transfer concurrently, variables generated in the same block also cannot transfer at the same time. For example, in Figure 5.7, the variables \( a, b, \) and \( d \) are generated in block \( P2 \). Their transfer times are not in conflict. Because \( a, b, \) and \( d \) are generated sequentially by the same functional unit, it is impossible to transfer \( a, b, \) and \( d \) at the same time. A compatibility graph can also be constructed for variables generated in the same block. In
Figure 5.9 (b), the compatibility graph based on block P2 is shown. Because the transfer times of variables a, b, and d are not in conflict, edges are formed among them to indicate they are compatible. By the same token, the compatibility graphs for blocks P1, P3, and P4 are shown in Figure 5.9 (a), (c) and (d) respectively.

Figure 5.9  (a) Compatibility graph for block P1,  (b) Compatibility graph for block P2,  (c) Compatibility graph for block P3, and  (d) Compatibility graph for block P4 shown in Figure 5.7.

After the compatibility graph derived from the variables’ data dependencies and the compatibility graphs derived from the variables generated in each block are constructed, these compatibility graphs are merged into a single compatibility graph. For each pair of variables, if at least one edge exists in any compatibility graph, an edge is formed
between the edge pair in the merged compatibility graph. If the compatibility graphs are represented in the adjacency matrix format as an \( n \times n \) 0-1 matrix, where \( n \) is the number of variables, the merged compatibility graph can be expressed as:

\[
G_T = G_d + G_p^1 + G_p^2 + \ldots + G_p^N,
\]

where \( G_T \) is the merged compatibility graph in the temporal domain, \( G_d \) is the compatibility graph based on the variables’ data dependencies, \( G_p^1 \) to \( G_p^N \) are the compatibility graphs derived from the variables that are generated in the same blocks, and “+” is denoted as the Boolean-OR operation. For example, Figure 5.10 (a) and (b) show the adjacency matrix format and the graphic representation of the merged compatibility graph respectively.

Figure 5.10  Compatibility graph with (a) adjacency matrix (b) graphic representation.

5.5.2 Bus Sharing in Spatial Domain

In accordance with a partitioned-bus architecture, the variables that are transferred concurrently still can share the bus if they are transferred on non-overlapped bus segments. For example, Figure 5.10 shows the variables that are compatible in the
temporal domain for Figure 5.7. Variable \( e \) is only compatible with variable \( f \) in the temporal domain. However, variable \( e \) is compatible with variables \( c, f, g, \) and \( h \) in the spatial domain if the linear placement of the functional unit modules, i.e., the order of placing the functional unit modules [Ewer90], is as shown in Figure 5.11. The reason is that, from Figure 5.7, variable \( e \) is transferred from block \( P_3 \) to \( P_2 \), variable \( c \) is transferred from block \( P_1 \) to \( P_4 \), variable \( f \) is transferred from block \( P_3 \) to \( P_4 \), and variables \( g \) and \( h \) are transferred from block \( P_4 \) to \( P_3 \). If the functions in blocks \( P_1, P_2, P_3, \) and \( P_4 \) are executed by functional units 1, 2, 3, and 4 respectively, the bus segments used to transfer variable \( e \) do not overlap with the bus segments used to transfer variables \( c, f, g, \) and \( h \) if the placement of the functional unit modules is as shown in Figure 5.11 in which functional unit module 4 is placed in the first position followed by functional unit modules 1, 3, and 2.

\[ \text{FUM}_4 \quad \text{FUM}_1 \quad \text{FUM}_3 \quad \text{FUM}_2 \]

**Figure 5.11** A Functional Unit Module placement for Figure 5.7.

Because the bus segments used by variable \( e \) do not overlap with variables \( c, f, g, \) and \( h \), variable \( e \) can share the same bus with variables \( c, f, g, \) or \( h \), as shown in Figure 5.12.
Figure 5.12  Variable e can share the same bus with variables c, g, or h in non-overlapped bus segments.

The example above shows a linear functional unit module placement in which variable e can share the same bus with variable c, f, g, and h, although variable e is not compatible with c, f, g, and h in the temporal domain. Finding an optimal linear placement for functional unit module is an NP-complete problem, and numerous heuristic algorithms have been proposed to address the problem [Ewer90, BeTs91, SaHa95]. In this thesis, the placement problem is addressed using Genetic Algorithms, which will be presented in Chapter 6.

After the linear placement of functional unit modules is determined, the compatibility of the variables for bus sharing in the spatial domain can be obtained. A compatibility graph is established to represent bus sharing at non-overlapped bus segments. From the partition given in Figure 5.7 and the placement given in Figure 5.11, the compatibility graph for the spatial domain, \( G_s \), is shown in Figure 5.13.
If the compatibility graphs $G_T$ and $G_S$ are represented as 0-1 matrices, the compatibility graph for estimating the number of buses, $G_B$, is:

$$G_B = G_T + G_S$$  \hspace{1cm} (5.4)$$

, where “+” denotes the Boolean-OR operation. $G_B$ is then input to the clique partitioning algorithm to find the minimum cliques, i.e., buses, that can cover all of the variables/nodes in the graph [TiSe86]. For the example given in Figure 5.7, and the linear placement shown in Figure 5.11, a possible solution is that two buses are needed. variables $a$, $b$, $c$, $d$, and $g$ are allocated to bus 1, while variables $e$, $f$, and $h$ are assigned to bus 2.

### 5.6 Links

After the variables are mapped to the buffer and to the buses, the number of input connections between the buses and the functional unit modules can be determined. The input connections are usually referred to as Links in other literature [DeNe91, SaAl96, Dhod96]. An example is shown in Figure 5.14 which is derived from Figure 5.7 and Figure 5.12. It shows the allocation of the variables to buses, which is obtained by

---

**Figure 5.13** Compatibility graph in spatial domain for Figure 5.7.
applying the algorithm presented in Section 5.5. It also show the allocation of the variables to the buffer of each functional unit, which is obtained by applying the algorithm proposed in Section 5.2.

**Definition 5.4** Links are the connections established for transferring variables from buses to functional unit modules.

The number of links is determined by allocation of variables to buses and to the buffers of functional unit modules. If there are any variables allocated to a bus that are also allocated to a buffer of a functional unit module, a link must be constructed between the bus and the functional unit module.

In some cases, one link between the bus and the functional unit module is not enough. An additional link is needed between a bus and a functional unit module whenever two variables on that bus are transferred at the same time to the functional unit module’s buffer from non-overlapped bus segments. For example, variables \( b \) and \( g \) at bus 1 are to be transferred to the buffer of functional unit module 3 which is shown in Figure 5.14. Variables \( b \) and \( g \) are not compatible in the temporal domain (See Figure 5.10 (b) ) but are compatible in the spatial domain. If there is only one link between bus 1 and Functional Unit Module 3, the transfer times of variables \( b \) and \( g \) might conflict. Therefore, two links are established between bus 1 and functional unit module 3 to ensure that variables \( b \) and \( g \) can be transferred simultaneously into the buffer of the functional unit module.

The total number of links is the sum of the links between each bus and every functional unit module. For example, six links are established among the buses and the functional unit modules as shown in Figure 5.14.
Figure 5.14  Allocation of variables to buses and buffers.

5.7 Multiplexers

Multiplexers are required for registers in the buffers in order to select among the input sources. For example, in Figure 5.14, variables $a$ and $h$ share the same register in the buffer of Functional Unit Module 3. Variable $a$ and $h$ are input from bus 1 and bus 2 respectively. A multiplexer is needed to select the input source for that register.

A multiplexer is needed for a register to select the input source whenever the variables sharing that register are input from different buses. After the allocation of the global variables to the buses and to the buffers, the number of multiplexers needed can be measured by checking Functional Unit Modules’ buffers. If a register in the buffer
contains more than one variable and the variables are input from different buses, a multiplexer is needed for that register. The total number of multiplexers is obtained by adding up the numbers of multiplexers required for each functional unit module’s buffer.

5.8 Cost Function

The metrics for the proposed partitioned-bus architecture were addressed in Sections 5.2-7. A cost function based on these metrics is formulated as follows:

\[
\text{Cost} = W_r R + W_l L + W_t T + W_b B + W_k K + W_m M,
\]

where

- \( R \) = the number of registers in the communication buffer,
- \( L \) = the number of registers in the register file,
- \( T \) = system delay,
- \( B \) = the number of buses,
- \( K \) = the number of links,
- \( M \) = the number of multiplexers, and

\( W_r, W_l, W_b, W_k, \) and \( W_m \) are the weights for these metrics. The cost function will be used to evaluate the quality of the partition results in Chapter 6.
Chapter 6

Optimization using Problem Space Genetic Algorithm

In this chapter, the implementation of the optimization approach used to address the synthesis task of the partitioned-bus architecture will be presented. The approach is based on a variation of genetic algorithms called the Problem Space Genetic Algorithm (PSGA) [StWu92]. Genetic algorithms were briefly introduced in Section 3.4.2. More details of genetic algorithms will be presented in Section 6.1. PSGA will also be discussed in Section 6.2.

6.1 Genetic Algorithms

Genetic algorithms (GAs) are powerful search techniques for optimization problems. Genetic algorithms were proposed by Holland in 1975 [Holl75]. Since then, numerous papers and dissertations in different research disciplines were published to establish the validity of GAs. As a result of these efforts, GAs are now widely regarded as efficient and effective search algorithms. The widespread use of GAs is due to the following advantages [Gold89, SaYo98]:

1. the computations are relatively simple,
2. the search spaces are not limited by restrictive assumptions such as continuity and the existence of derivatives, and
3. the algorithms search a set of points, i.e., solutions, simultaneously in the search space instead of a single point, a technique which can reduce the probabilities of being stuck in the local optima and speed up the searching process.

In the GA approach, solutions of optimization problems are encoded as strings of symbols called chromosomes. The symbols that make up a chromosome are known as genes. A group of possible solutions, referred to as a population, coexist in the optimal solution search space. During each iteration, referred to as a generation, a new population is produced by adopting some newly generated chromosomes and deleting some existing chromosomes. The reproduction process includes selecting pairs of chromosomes from the current population, performing crossover operations on the selected chromosomes, and carrying out mutation operations on the newly created chromosomes. The chromosomes chosen for reproduction are called parents, and the newly created chromosomes are their offspring. Each chromosome is associated with a fitness value, which is computed by a fitness function. The selection of the parents for reproduction is determined by their fitness values. This is because a chromosome’s fitness value indicates the cost/quality of the solution that the chromosome represents in the search space.

6.1.1 Fitness Function

A fitness function is used to calculate the fitness values of the chromosomes. A chromosome’s fitness value expresses the quality of the solution that the chromosome represents. The chromosomes with higher fitness values have higher solution qualities. Since the solution qualities for the combinatorial optimization problems are usually estimated by cost functions (see Section 5.1), a fitness value also indicates the cost of the solution represented by the chromosome. If the objective of the optimization problem is to minimize the cost, the chromosomes with lower costs should have higher fitness values. For example, in the resource-constrained scheduling problem, if delay is used to represent the cost of the solution, chromosomes with lower costs, i.e. shorter delays,
should be associated with higher fitness values. A simple fitness function for the resource-constrained scheduling problem can be expressed as the reciprocal of the delay, which is shown as follows:

$$f(i) = \frac{1}{\text{Delay}_i}, \quad (6.1)$$

where \(\text{Delay}_i\) is the delay for the \(i^{th}\) chromosome in the population.

### 6.1.2 Initial Population

Because genetic algorithms work on populations of solutions, a procedure has to be devised to generate the initial population. The procedure is usually called an initial population constructor [SaYo98]. The quality of the final solution is greatly affected by how the initial population is constructed as well as the size of the initial population. Some of the chromosomes in the initial population can be generated by known heuristic algorithms. Usually randomly generated chromosomes fill a pre-set population size. For example, in the functional partitioning problem, a heuristic partitioning algorithm such as Hierarchical Clustering or Min-Cut can be used to generate some initial solutions.

### 6.1.3 Crossover

A crossover operator is the main characteristic that distinguishes GA from other search algorithms [Gold89]. It is a simple mechanism intended to let the offspring inherit the properties of their parents. The crossover operator operates on two parents in order to generate two offspring. The process involves random number generations, string copies, and partial string exchanges. Many different crossover operators, such as Cut-and-Paste, Multiple-Point, Partially Matched Order (PMX), and Order Crossover (OX), have been proposed. The effectiveness of these different crossover operators depends on the combinatorial optimization problem being solved [SaYo98].
To illustrate how the crossover operator works, consider the following two examples that use the Order Crossover and Cut-and-Paste crossover operators to address the Traveling Salesman Problem (TSP) with nine cities. The nine cities are denoted as A through I [Chri75]. The first example is for the Order Crossover operator, and the second one is for the Cut-and-Paste operator.

**Example 6.1:**
Two routes (parents), which are represented by strings as (BIDCFGEHA) and (AGHCBIDEF), are chosen from the population. The order of the alphabets in the string expresses the sequence of the cities to be visited. The cut point is a randomly generated number, such as four. The first offspring is generated by copying the first four characters from the first parent whose partial result is denoted as (BIDC*****). The second parent is then scanned from left to right character by character. If a character is encountered that is not already in the partial result, that character is then used to fill an empty position. The scanning process ends when all the positions (nine) are filled. The result for the first offspring is (BIDCAGHEF).

The second offspring is generated similarly by copying the first four characters from the second parent. The partial result is (AGHC*****). The first parent is then scanned from left to right for un-used characters in the partial result to fill in the empty positions. The result for the second offspring is (AGHCBIDFE).

The offspring may not represent valid solutions. For example, if there is no edge between cities C and A, offspring 1 is not a valid solution. A procedure must be provided to check the validity of the newly generated offspring. The invalid offspring are discarded or a repairing algorithm must be used to fix them.

**Example 6.2:**
The Cut-and-Paste crossover operator starts from generating a random number as the cut point and then dividing each of the two parent chromosome into two parts. The offspring
are then produced by concatenating the segment of one parent to the left of the cut point with the portion of the other parent to the right of the cut point. Assume two parents, *(ABCDIHGFE)* and *(BCADEFIHG)*, are chosen for reproduction, and the cut point is four. The cut-and-paste operator combines the left portion of first chromosome, *(ABCD*****)* with the right portion of the second chromosome, *(****EFIHG)*. The resulting first offspring is *(ABCDEFIHG)*. The second offspring is obtained by combining the left portion of second chromosome *(BCAD*****)* with the right portion of the first chromosome *(****IHGFE)*. The resulting second offspring is *(BCADIHGFE)*. Computational simplicity is the major advantage of cut-and-paste. However, illegal offspring may be generated, such as the same city appears twice in the newly generated chromosomes for the TSP problem. The common approaches used to cope with the illegal offspring problem include discarding or providing a repair mechanism to fix the illegal offspring [SaAl96]. More discussion will be given in Section 6.2.

### 6.1.4 Selection of Crossover Parents

Pairs of chromosomes are chosen from the population to perform crossover operations. The choice is a probabilistic process. The chromosomes with higher fitness values are more likely to be chosen to mate than the ones with lower fitness values. The common way to simulate the selection process is to assign probabilities to each chromosome. Assignment of the probabilities to each chromosome is directly proportional to the fitness values of the chromosomes.

For example, if there are five chromosomes in the population, assume their fitness values are 10, 9, 9, 8, and 2 respectively. A simple way of assigning the probabilities to each chromosome is to divide each individual chromosome’s fitness value by the total of all the fitness values. Therefore, the probability for the first chromosome is 10/(10+9+9+8+2), which is 26.3%. The probabilities for the rest of the chromosomes are 23.7%, 23.7%, 21.0%, and 5.3%
After the probabilities of each chromosome are obtained, an algorithm has to be provided to select parents for reproduction. The **roulette-wheel algorithm**, currently in use by several researchers, has been demonstrated to be a simple and effective method for selecting parents [SaAl96, Gold89, Dhod95, Davi91]. In the roulette-wheel algorithm, a pie-chart like wheel, shown in Figure 6.1, is constructed. Each chromosome occupies a sector of the wheel. The sizes of the sectors are proportional to the chromosomes’ probabilities. To make a selection, the wheel is spun, and whichever chromosome comes up is selected as a parent. The pseudo-code of the roulette-wheel algorithm is shown as follows:

**Figure 6.1** An example of pie-chart for the roulette-wheel algorithm.

```
Algorithm Roulette-Wheel
    j = 0; // the jth chromosome in the population
    accumulation = 0; //
    r = random(0,1); // r is a random number between 0 and 1
                      // random() is function used to generate random numbers
    while(accumulation < r )
        j = j + 1;
        accumulation = accumulation + Probability(j);
                      // Probability(j) is the probability of
                      // the jth chromosome
    end while;
    return j;
End Roulette-Wheel
```
6.1.5 Mutation

Mutation is an operator that randomly selects a small number of offspring and then probabilistically alters the values of some genes in their chromosomes. If there are too many offspring chosen for mutation, there will be too much random disturbance and cause the offspring to lose the chances of inheriting the good characteristics from their parents. The effect of mutation is to perturb a certain chromosome and introduce new characteristics that are not currently present in any chromosomes in the population. Mutation is needed because, even though the crossover intentionally selects chromosomes with high fitness values and then recombines them, the crossover might occasionally miss some potentially useful genetic materials [Gold89]. The following example illustrates a weakness of the crossover operator:

Suppose a population contains four chromosomes,

\[ s_1 = (BIDCFGEHA) \]
\[ s_2 = (GHCBIDEFA) \]
\[ s_3 = (CFGEHBIDA) \]
\[ s_4 = (BIEHDCFGA) \]

It is noted that last cities are A in all four chromosomes. If the Order Crossover operator is used, in spite of the choice of the parents or the position of the cut point, the last city of the produced offspring will always be A. The only way to assure that the last city is a city other than A is by the operation of mutation.

6.1.6 Selection of Next Generation

In each generation, pairs of chromosomes are chosen as parents to perform crossover and new offspring are created. Because of the offspring, the population size grows. In order to keep the population size at a constant, a fixed number of chromosomes must be chosen.
from the initial population and from the generated offspring to form the population of the next generation. A commonly used strategy for selecting the chromosomes is based on their costs. If the population size of the new generation is set to be the same as the current population size, and the number of offspring generated is equal to the population size, the chromosomes with costs lower than the median cost of the current population and offspring population will be selected to form the next generation.

6.2 Problem Space Genetic Algorithm (PSGA)

In prior genetic algorithms research [SaAl96, RaGu95], the chromosomes in the population are encoded directly as the solutions of the combinatorial optimization problem. One major disadvantage for this type of chromosome encoding is that after crossover and/or mutation operations, the generated solutions may not be feasible. For example, in the high-level synthesis scheduling problem [McFa88], a function cannot be scheduled before its predecessors are scheduled. If the chromosomes themselves carry the clock cycles in which functions are scheduled for execution, infeasible solutions will be generated after crossover and/or mutation if a function is scheduled before its predecessors were scheduled. The infeasible solutions have to be fixed or discarded. An example of such a situation is illustrated as follows:

Suppose we want to optimize the resource-constrained scheduling problem in High-Level Synthesis with one available functional unit. The CDFG is shown in Figure 6.2 (a) and the chromosome representation is given in Figure 6.2 (b). Each gene represents the clock cycle in which the function is scheduled to be executed. The first row of Figure 6.2 (b) gives the function to which the gene corresponds.
Figures 6.2 and 6.3 show that two chromosomes in the population were chosen as two parents for reproduction. If the cut-and-paste crossover operator is used and the cut point, indicated by ‘|’ in Figure 6.4 (a), is between function B and C [Gold89], the two resulting offspring are shown in Figure 6.4 (b). Offspring 1 shows that functions A and E are scheduled in the first clock cycle, which violates the resource constraint in which only one functional unit is available. Furthermore, scheduling functions B and C in the same clock cycle violates both the data dependency and resource constraint. The solution
Figure 6.4  (a) The genes represent the clock cycles in which the functions are scheduled for execution.  
(b) The offspring are generated by performing the cut-and-paste crossover on parents 1 and 2.

represented by offspring 1 is not a feasible solution. Offspring 2 shows that functions A and C are scheduled in the second clock cycle, and functions B and E are scheduled in the
fourth clock cycle, which violates the resource constraint. Offspring 2 is not a feasible solution either. The infeasible solutions have to be discarded, or an algorithm has to be provided to correct the infeasible solutions [SaAl96].

To avoid infeasible solutions, PSGA takes an alternative approach by encoding the problem data not the solution data [StWu92]. The problem space information is used by a fast heuristic algorithm to map the problem information into solutions. The advantages of PSGA include:

1. The crossover operator can be constructed easily without having to fix infeasible solutions.
2. An existing fast heuristic algorithm, such as List scheduling for scheduling problems, can be selected to map the problem space to the solution. The existing problem specific heuristic algorithm has already gained insight knowledge of the problem. Embedding this heuristic algorithm within GAs is equivalent to incorporating problem specific knowledge into the search for the optimal solution.

For example, if the genes represent the scheduling priorities that are used by List scheduling as the priority to assign the functions in the ready list to the functional units [GajD92], Figure 6.5 shows the offspring generated by taking the chromosomes in Figures 6.2 and 6.3 as parents and performing the cut-and-paste crossover operation. The chromosome representations for the CDFGs in Figures 6.2 and 6.3 are shown in Figure 6.5 (a). The scheduling priorities are random numbers, and the cut point is between functions B and C. The scheduling priorities in the offspring are then used by List scheduling to perform scheduling. The results are shown in Figure 6.5 (b), which are both valid solutions.
Figure 6.5  (a) The genes represent the scheduling priorities of each function. (b) The offspring generated by performing the cut-and-paste crossover on parents 1 and 2. The schedulings are obtained by applying List Scheduling using Scheduling Priority as the priority function.
6.3. Problem Space Genetic Partitioning (PSGP)

The implementation of the functional partitioning algorithm, which is used to address the issues of complex synthesis tasks presented in Chapter 5, is presented in this section. The algorithm uses the PSGA technique. For illustration purposes, a equation used for modeling the superposition of reflected and incident uniform plane electromagnetic waves [Chen90] can be expressed as

\[ E(x,z,t) = -2E_{io} \left[ -\cos \theta \sin(\beta_1 z \cos \theta_1) \sin(\omega t - \beta_1 x \sin \theta_1) \hat{i} + \sin \theta \cos(\beta_1 z \cos \theta_1) \cos(\omega t - \beta_1 x \sin \theta_1) \hat{k} \right] = X \hat{x} + Z \hat{z} \]

The CDFG for this equation is shown in Figure 6.6.

**Figure 6.6** CDFG for the electromagnetic wave superposition model.
6.3.1 Chromosome Encoding and Initial Population

For optimizing the functional partitioning problem, a chromosome consists of three parts.

1. a list of integers, each integer represents the partition block to which each function/node in the CDFG is assigned,
2. a list of integers, each integer represents the work remaining (WR) [StWu92] of each function in the CDFG, and
3. a list of integers, each integer represents the placement priority of each functional unit module.

For example, Figure 6.7 shows three chromosome representations for Figure 6.6 which

<table>
<thead>
<tr>
<th>Function ID</th>
<th>Block assignment</th>
<th>Work remaining</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 1 1 2 1 1 2 1 2 1 1 1 1 2 1 2 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>31 30 29 3 2 1 0 30 28 2 32 31 30 3 2 1 1 2 0</td>
<td></td>
</tr>
</tbody>
</table>

Function Unit Module ID : 1 2
Placement priority: 70 98

<table>
<thead>
<tr>
<th>Function ID</th>
<th>Block assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 1 1 2 1 1 2 1 1 2 1 2 2 2 1</td>
</tr>
<tr>
<td></td>
<td>21 19 33 31 20 44 29 22 32 18 27 24 33 45 36 34 25 27 33</td>
</tr>
<tr>
<td></td>
<td>79 32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function ID</th>
<th>Block assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2 1 2 1 2 1 2 2 2 1 2 2 2 2 2 2</td>
</tr>
<tr>
<td></td>
<td>26 29 28 23 37 45 36 45 40 40 31 31 27 27 44 34 32 24 37</td>
</tr>
<tr>
<td></td>
<td>90 45</td>
</tr>
</tbody>
</table>

Figure 6.7 PSGP Chromosome representation for Figure 6.6.
contains 19 functions. These 19 functions are partitioned to two blocks. From the chromosome representation, the partitioned-bus architecture can be built. For example, Figure 6.8 shows the architecture for Figure 6.7 chromosome 1. The procedures of constructing each chromosome’s block assignment, work remaining, and placement priority, and constructing the initial population will be addressed in this section. Steps of building the partitioned-bus architecture from a given chromosome will be introduced in the next section.

Suppose a two-way partitioning for Figure 6.6 is to be performed. The first chromosome in the initial population is constructed as follows.

Note: The diagram is shown only with buffers, local register files, functional units, buses, switches, and links. Refer to Figure 4.2 for details.

**Figure 6.8 Partitioned-bus architecture for Figure 6.7 chromosome 1.**
The first part of the chromosome represents the block assignment for each function. Each function is randomly assigned to one of the two blocks. For example, chromosome 1 in Figure 6.7 shows that functions 1, 2, 3, 4, 6, 7, 9, 11, 12, 13, 14, 15, and 17 are assigned to block 1 and functions 5, 8, 10, 16, 18, and 19 are assigned to block 2.

The second part of the chromosome contains the scheduling priorities, which are referred as work remaining, used by list scheduling to estimate the system delay. Work remaining is a term borrowed from Operations Research. A function’s work remaining is used to indicate the minimum delay from when the function is finished to when all the functions are completed [StWu92]. The steps of calculating each function’s work remaining for the first chromosome in the initial population is given as follows:

1. perform ALAP scheduling on the given CDFG.
2. work remaining for function i is expressed as

   \[ \text{WR}_i = (\text{The total function execution delays in the critical path}) - (\text{The clock cycle that function } i \text{ is completed}). \]

For example, suppose the worst case execution delays for floating point addition is 2 clock cycles, multiplication is 1 clock cycle, cosine is 24 clock cycles, and sine is 26 clock cycles (These numbers are taken from Intel arithmetic processor unit 8231 and are normalized against multiplication). The critical path in Figure 6.6 is Path 11-12-13-9-18-6-7. The critical path can be found by using the Depth First Search to search for all the paths in the CDFG, and then select the path with the longest delay. The total delays in the the critical path is found to be 58 clock cycles. The ALAP scheduling result is shown in Table 6.1 (a). The work remaining for each function then can be calculated based on the critical path delay and the ALAP scheduling result. For example, in Table 6.1 (b), function 18, which is a sine, is scheduled to be executed starting from the 31st clock cycle and completed at the 56th clock cycle. Therefore, the work remaining for function 18 is equal to (58 - 56), which is 2.
Table 6.1 (a) ALAP scheduling result for Figure 6.6.
(b) Work remaining for each function.

<table>
<thead>
<tr>
<th>Clock Cycle</th>
<th>Function ID and type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11 ( sin )</td>
</tr>
<tr>
<td>4</td>
<td>1( cos )</td>
</tr>
<tr>
<td>27</td>
<td>12 ( * )</td>
</tr>
<tr>
<td>28</td>
<td>2 ( * ) , 8 ( * ) , 13 ( * )</td>
</tr>
<tr>
<td>29</td>
<td>3 ( * ) , 9 ( - )</td>
</tr>
<tr>
<td>30</td>
<td>4 ( sin )</td>
</tr>
<tr>
<td>31</td>
<td>18 ( sin )</td>
</tr>
<tr>
<td>32</td>
<td>14 ( cos )</td>
</tr>
<tr>
<td>33</td>
<td>10 ( cos )</td>
</tr>
<tr>
<td>56</td>
<td>5 ( * ) , 15 ( * )</td>
</tr>
<tr>
<td>57</td>
<td>6 ( * ) , 16 ( * ) , 17 ( neg)</td>
</tr>
<tr>
<td>58</td>
<td>7 ( * ) , 19 ( * )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function ID and Type</th>
<th>Start clock cycle</th>
<th>End clock cycle</th>
<th>Work Remaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (cos)</td>
<td>4</td>
<td>27</td>
<td>31</td>
</tr>
<tr>
<td>2 (*)</td>
<td>28</td>
<td>28</td>
<td>30</td>
</tr>
<tr>
<td>3 (*)</td>
<td>29</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>4 (sin)</td>
<td>30</td>
<td>55</td>
<td>3</td>
</tr>
<tr>
<td>5 (*)</td>
<td>56</td>
<td>56</td>
<td>2</td>
</tr>
<tr>
<td>6 (*)</td>
<td>57</td>
<td>57</td>
<td>1</td>
</tr>
<tr>
<td>7 (*)</td>
<td>58</td>
<td>58</td>
<td>0</td>
</tr>
<tr>
<td>8 (*)</td>
<td>28</td>
<td>28</td>
<td>30</td>
</tr>
<tr>
<td>9 (-)</td>
<td>29</td>
<td>30</td>
<td>28</td>
</tr>
<tr>
<td>10 (cos)</td>
<td>33</td>
<td>56</td>
<td>2</td>
</tr>
<tr>
<td>11 (sin)</td>
<td>1</td>
<td>26</td>
<td>32</td>
</tr>
<tr>
<td>12 (*)</td>
<td>27</td>
<td>27</td>
<td>31</td>
</tr>
<tr>
<td>13 (*)</td>
<td>28</td>
<td>28</td>
<td>30</td>
</tr>
<tr>
<td>14 (cos)</td>
<td>32</td>
<td>55</td>
<td>3</td>
</tr>
<tr>
<td>15 (*)</td>
<td>56</td>
<td>56</td>
<td>2</td>
</tr>
<tr>
<td>16 (*)</td>
<td>57</td>
<td>57</td>
<td>1</td>
</tr>
<tr>
<td>17 (neg)</td>
<td>57</td>
<td>57</td>
<td>1</td>
</tr>
<tr>
<td>18 (sin)</td>
<td>31</td>
<td>56</td>
<td>2</td>
</tr>
<tr>
<td>19 (*)</td>
<td>58</td>
<td>58</td>
<td>0</td>
</tr>
</tbody>
</table>

Note: Worst case execution delays for cosine is 24 clock cycles, sine is 26 clock cycles, floating point addition is 2 clock cycles, floating point multiplication is 1 clock cycle, and neg is 1 clock cycle.

The third part of the chromosome represents the placement priorities of the functional unit modules, which are used to determine the ordering of the functional unit modules in the linear placement topology that was discussed in Section 5.5.2. The ordering of the functional unit modules is important, because it affects the number of buses needed to transfer the variables.

The placement priorities are random numbers, and the random numbers are used as the priorities for PSGP to perform functional unit module linear placement. By using this approach, i.e., PSGA approach, the infeasible linear placement solutions which are generated by the crossover or mutation operator can be avoided. The reason is that if the chromosomes represent the physical locations of the functional unit modules, the
infeasible solutions such as placing two functional unit modules in the same positions may be generated by the crossover operator.

The functional unit module with highest priority is placed at the left-most location in the linear placement topology, followed by the functional unit module with the second highest priority, and so forth. For example, for chromosome 1 in Figure 6.7, functional unit module 2 has higher placement priority than functional unit module 1. Therefore, functional unit module 2 is placed in the first place and followed by functional unit module 1. Note that the block number to which the functions are assigned correspond to the functional unit module number, in which the functions are to be executed. For example, the functions assigned to block 1 are to be executed in functional unit module 1.

For the rest of the chromosomes in the initial population, the placement priorities are generated by randomly perturbing the placement priority of the first chromosome:

\[
PR_i = PR_{\text{max}} + \gamma_i \\
\gamma_i = [-b, b],
\]

(6.2a)

(6.2b)

where:
- \(PR_i\) is the placement priority for functional unit module \(i\) in a chromosome,
- \(PR_{\text{max}}\) is the maximum placement priority in the first chromosome,
- \(\gamma_i\) is a random number that is between \(-b\) and \(b\), and
- \(b\) is equal to \(PR_{\text{max}} / 2\).

The block assignments are generated using the same technique as that applied to the first chromosome. The work remaining values are generated by random perturbation of the work remaining of the first chromosome. The formulas are shown below [Dhod95]:

\[
WR_i = WR_{\text{max}} + \delta_i \\
\delta_i = [-a, a],
\]

(6.3a)

(6.3b)

where:
\( WR_i \) is the work remaining for Function \( i \) in a chromosome,
\( WR_{\text{max}} \) is the maximum work remaining in the first chromosome,
\( \delta_i \) is a random number that is between \(-a\) and \(a\), and
\( a \) is equal to \( WR_{\text{max}} / 2 \).

In Figure 6.7, chromosomes 2 and 3 are examples of chromosomes generated by applying Formulas 6.2 and 6.3.

### 6.3.2 Cost Estimation

After functions are assigned to blocks, the functions’ work remainings are calculated, and the functional unit modules’ placement priorities are computed, the metrics and algorithms presented in Chapter 5 are used to assess the qualities of the chromosomes. The qualities are assessed by using the cost function given in Equation 5.5. In this section, chromosome 1 in Figure 6.7 is used as an example to illustrate the procedures of computing the costs for chromosomes.

**Example 6.3:**

Since the first part of the chromosome represents each function’s block assignment, the result of two-way partition represented by chromosome 1 can be obtained and is shown in Figure 6.9 (a). Figure 6.9 (b) shows the paths in Figure 6.9 (a) and the path vectors for each variable. The steps of applying the algorithms used to obtain the communication buffer size, register file size, system delay, the number of links, and the number of multiplexers presented in Chapter 5 are addressed in the following paragraphs.
Non-shaded nodes are assigned to block P1, and shaded nodes are assigned to block P2.

(a) Path 000000001 : 1 - a' - 5 - e - 6 - f - 7
Path 000000010 : 1- a - 2 - b - 3 - c' - 14 - r - 15 - s - 16 - t - 19
Path 000000100 : 1 - a  - 2 - b -3 - c - 4 - d - 5- e - 6 - f - 7
Path 000001000 : 8 - h - 9 - j' - 10 - m - 16 - t - 19
Path 000010000 : 11- n' - 15 - s - 16 - t - 19
Path 000100000 : 11 - n - 12 - p -13- q - 9 - j' - 10 - m - 16 - t - 19
Path 010000000 : 11 - n - 12 - p -13 - q - 9 - j - 18 - k - 6 - f - 7
Path 100000000 : 17- u - 19

PV(a) = 000000110
PV(a') = 000000001
PV(b) = 000000110
PV(c) = 000001000
PV(c') = 000000010
PV(d) = 000001000
PV(e) = 000000101
PV(f) = 010010101
PV(h) = 000110000
PV(j) = 010010000
PV(j') = 001001000

(b) Figure 6.9 (a) Two-way partition represented by Figure 6.7 chromosome 1.
(b) Paths in the CDFG.
(c) Path Vectors for each variable.
Communication buffer

From Figure 6.9 (a), the edge cut-sets C_{12} and C_{21} are \{a', d, j, j', s, u\} and \{e, h, k\} respectively. The compatibility graphs for computing buffer sizes |b_{12}| and |b_{21}| can be constructed by the algorithm addressed in Section 5.2.3, which takes every edge pair in the edge-cut sets and then performs bit-wise AND on the edges' path vectors. If the result is not a zero vector, the variables are compatible and can share the same register in a buffer. The compatibility graphs are shown in Figures 6.10 (a) and (b). After applying the clique partition algorithm [TsSi86] to find the numbers of cliques in the compatibility graphs, the buffer sizes |b_{12}| and |b_{21}| are obtained as 6 and 2 respectively. The communication buffer size, R, is the sum of |b_{12}| and |b_{21}|, which is 8. The variables in the same clique are allocated to the same register in the buffers.

\[ C_{12} = \{a', d, j, j', s, u\} \]

(a)

PV(a') = 000000001
PV(d) = 000000100
PV(j) = 010010000
PV(j') = 001001000
PV(s) = 000100010
PV(u) = 100000000

(b)

PV(e) = 000000101
PV(h) = 000011000
PV(k) = 010010000

Figure 6.10 (a) Path vectors and compatibility graph for edge cut-set C_{12}. (b) Path vectors and compatibility graph for edge cut-set C_{21}. 
Register file

The local variables in blocks 1 and 2 are \{a, b, c, c’, f, n, n’, p, q, r\} and \{m, t\} respectively. Path vectors of the local variables are used to construct compatibility graphs that are used to estimate the register file size. The procedures of constructing the compatibility graphs from the path vectors were addressed in Section 5.3. The path vectors and the compatibility graph for the local variables in block 1 are shown in Figure 6.11 (a) and (b). The compatibility graph is then input to the clique partitioning algorithm [TsSi86]. Four cliques are found in Figure 6.11 (b), which are shown in Figure 6.11 (c).

The path vectors and compatibility graph for the local variables in block 2 are shown in Figure 6.12 (a) and (b). Figure 6.12 (c) shows that one clique is found in Figure 6.12 (b). The number of cliques is equal to the number of registers in the local register files. Therefore, the sum of these two local register files, which is denoted as register file size \(L\), is 5. The variables in the same clique are allocated to the same register in the register file.

\[
\begin{align*}
PV(a) &= 000000110 \\
PV(b) &= 000000110 \\
PV(c) &= 000000100 \\
PV(c') &= 000000010 \\
PV(f) &= 010010101 \\
PV(n') &= 000100000 \\
PV(n) &= 011000000 \\
PV(r) &= 000000010 \\
PV(p) &= 011000000 \\
PV(q) &= 011000000
\end{align*}
\]

\(\{n'\} \{a,b,c',r\} \{c,f\} \{n,p,q\}\)

\(\text{Figure 6.11 (a) Path vectors for the local variables in Figure 6.9 (a) block 1.} \\
(\text{b) Compatibility graph for the local variables.}) \\
(\text{c) Cliques found in (b).})\)
Figure 6.12 (a) Path vectors for the local variables in Figure 6.9(a) block 2. (b) The compatibility graph for (a). (c) The cliques found in (b).

System delay

List scheduling that uses chromosome’s work remaining as the priority function is applied to estimate the system delay. Note that functions with higher work remaining values have higher priorities. Two functional units are available and the functions in blocks 1 and 2 are executed by functional units 1 and 2 respectively. The scheduling result is in Table 6.2.

Table 6.2 A list scheduling result for Figure 6.6. with two available functional units and using Figure 6.7 chromosome 1’s work remaining as the priority function.

<table>
<thead>
<tr>
<th>Clock Cycle</th>
<th>Function ID and type in block 1</th>
<th>Function ID and type in block 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11 (sin)</td>
<td>8 ( *)</td>
</tr>
<tr>
<td>27</td>
<td>1 (cos)</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>12 ( * )</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>2 ( * )</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>13 ( * )</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>3 ( * )</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>9 ( - )</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>4 (sin)</td>
<td>10 (cos)</td>
</tr>
<tr>
<td>81</td>
<td>18 (sin)</td>
<td></td>
</tr>
<tr>
<td>83</td>
<td>14 (cos)</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>15 ( * )</td>
<td>5 ( * )</td>
</tr>
<tr>
<td>108</td>
<td>17 (neg)</td>
<td>16 ( * )</td>
</tr>
<tr>
<td>109</td>
<td>6 ( * )</td>
<td>19 ( * )</td>
</tr>
<tr>
<td>110</td>
<td>7 ( * )</td>
<td></td>
</tr>
</tbody>
</table>

Note: 1. Worst case execution delays for cosine is 24 clock cycles, sine is 26 clock cycles, floating point addition is 2 clock cycles, floating point multiplication is 1 clock cycle, and neg is 1 clock cycle.

2. Functions in blocks 1 and 2 are executed by functional units 1 and 2 respectively.
The table also shows that the last scheduled function, which is function 7, is scheduled at clock cycle 110 and completed at the same cycle. This is because function 7 is a floating point multiplication whose normalized delay is 1 clock cycle. Therefore, the system delay, $T$, is 110.

**Buses**

Variables $a'$, $d$, $j$, $j'$, $s$, and $u$ are transferred from block 1 to 2, and variables $e$, $k$, and $h$ are transferred from block 2 to 1. If the variables are not transferred at the same time, or transferred at same time but on different bus segments, the variables can share the same bus. The placement priority in the chromosome is used to place the relative locations of the functional unit modules. The placement of the functional unit modules can affect the variable bus sharing in the spatial domain, which in turn can affect the number of buses. The placement priority in Figure 6.7 chromosome 1 indicates that functional unit module 1 is placed at the leftmost position and then followed by functional unit module 2, because functional unit module 1 has a higher placement priority. A compatibility graph can be established by using the procedures presented in Section 5.5 to compute the number of buses. The compatibility graph is represented in adjacency matrix format shown in Figure 6.13 (a). The clique partitioning algorithm is used to search for the minimum number of cliques that can cover the compatibility graph. Two cliques, which are shown in Figure 6.13 (b), are found in the compatibility graph. The variables in the same clique are allocated to the same bus. For example, in Figure 6.13 (b), variables $\{a', d, j, j', s, u\}$ are allocated to bus 1, and variables $\{e, k, h\}$ are allocated to bus 2.
<table>
<thead>
<tr>
<th></th>
<th>a'</th>
<th>d</th>
<th>j</th>
<th>j'</th>
<th>s</th>
<th>u</th>
<th>e</th>
<th>k</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>a'</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j'</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>u</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>k</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\{a', d, j, j', s, u\} \{e, h, k\}

(a) (b)

**Figure 6.13** (a) Adjacency matrix representation for the compatibility graph used to estimate the number of buses required for Figure 6.7 chromosome 1.
(b) The cliques found in (a).

### Links and Multiplexers

After variables are allocated to buffers, local register files, and buses, the numbers of links and multiplexers can be obtained by using the procedures addressed in Sections 5.6 and 5.7. Links are used to transfer the variables from buses to functional unit modules. A link must be established between a bus and a functional unit module if there are any variables allocated to a bus that are also allocated to the buffer of a functional unit module. For example, in Figure 6.8, all the variables on bus 2 and none of the variables on bus 1 are in the buffer of functional unit module 1. Therefore, there must be a link between bus 2 and functional unit module 1. Analogously, there is a link between bus 1 and functional unit module 2. The total number of links, \(K\), is obtained as 2.

In front of a buffer’s register, a multiplexer is inserted to select the input sources if the register is shared by multiple variables and the variables are input from different buses. The only variables that share a register in the buffers are \(h\) and \(k\). Because variables \(h\) and \(k\) are transferred from the same bus, no multiplexer is needed for the register in which \(h\) and \(k\) are stored. The number of multiplexers, \(M\), needed for this example is 0.
After the communication buffer size, register file size, system delay, the number of links, and the number of multiplexers are obtained, the cost for chromosome 1 can be calculated by using Equation 5.5. If all the weights for these metrics are set to 1, the cost for chromosome 1 is $1 \times 8 + 1 \times 5 + 1 \times 110 + 1 \times 2 + 1 \times 2 + 1 \times 0 = 127$.

By applying the above steps, the costs for chromosomes 2 and 3 can be obtained as 128 and 154 respectively.

### 6.3.3 Crossover Operator

#### 6.3.3.1 Parent Selection

Crossover is the operation that selects two parent chromosomes from the population and produces two offspring. The selection of parents from the population is based on the fitness values of the chromosomes. The higher the fitness value, the higher the probability of a chromosome being chosen for reproduction.

The fitness value for each chromosome is calculated by using the following fitness function:

$$f(i) = \frac{(C_{\text{max}} - C_i + 1)^p}{\sum_{j=1,N} (C_{\text{max}} - C_j + 1)^p}, \quad (6.4)$$

where:

- $C_{\text{max}}$ is the maximum cost of any chromosomes in the population,
- $C_i$ is the cost for chromosome $i$,
- $N$ is the number of chromosomes in the population, and
- $p$ is a parameter used to determine the selectivity of the fitness function [StWu92].

The plus 1 in the denominator is needed for preventing a divided by 0 error if all the members in the population converge to an identical chromosome. $p$ is used to balance the diversity and convergence of the population, which is termed as selectivity by Storer et
If \( p \) is set too high, only the fittest few chromosomes will survive and the algorithm will prematurely converge to a population with many identical solutions, which means the diversity is lost. The value of \( p \) was experimented and established by Storer et al., and it is commonly set to an integer between 1 and 5.

The costs for the three chromosomes in Figure 6.7 are 127, 128, and 154 respectively. If the initial population contains only these three chromosomes and selectivity \( p \) is set to 2, their fitness values are calculated as follows:

\[
\begin{align*}
 f(1) &= \frac{(154 - 127 + 1)^2}{(154 - 127 + 1)^2 + (154 - 128 + 1)^2 + (154 - 154 + 1)^2} = \frac{784}{1514} = 0.52, \\
 f(2) &= \frac{(154 - 128 + 1)^2}{(154 - 127 + 1)^2 + (154 - 128 + 1)^2 + (154 - 154 + 1)^2} = \frac{729}{1514} = 0.48, \text{ and} \\
 f(3) &= \frac{(154-154+1)^2}{(154 - 127 + 1)^2 + (154 - 128 + 1)^2 + (154 - 154 + 1)^2} = \frac{1}{1514} \approx 0.00065.
\end{align*}
\]

After all the fitness values in the population are computed, the probability of each chromosome for being selected as the parents for reproduction can be obtained by using the method presented in Section 6.1.4. The probabilities for the three chromosomes in Figure 6.7 are \( \frac{0.52}{0.52 + 0.48 + 0} = 0.52, \frac{0.48}{0.52 + 0.48 + 0} = 0.48, \) and \( \frac{0}{0.52 + 0.48 + 0} = 0 \) respectively.

The roulette wheel algorithm is used to chose the chromosomes, which is based on the chromosomes’ probabilities, for crossover operation [Gold89]. The Cut-and-Paste crossover operator is used for the implementation of PSGP, which is addressed in the following section.

**6.3.3.2 Cut-and-Paste**

Suppose two chromosomes, \( M_m \) and \( M_f \), are selected and randomly generates a cut point \( i \). The first offspring is the concatenation of \( M_m(1:i) \) and \( M_f(i+1,|V|) \), and the second offspring is the concatenation of \( M_f(1:i) \) and \( M_m(i+1,|V|) \), where \(|V|\) is denoted as the number of functions in the CDFG.
For example, let chromosomes 1 and 2 in Figure 6.7 be selected for a Cut-and-Paste crossover operation. Because the chromosomes contain three parts, which are block assignment, work remaining, and placement priority, three cut points needed to be generated. If the randomly generated cut points are 12, 5, and 1 respectively, the two newly generated offspring are shown in Figure 6.14.

Figure 6.14 Two offspring generated by Cut-and-Paste crossover operator using cut points 12, 5, and 1 respectively.
6.3.4 Mutation Operator

The mutation operator randomly selects a small percentage of the offspring for mutation. The procedure used in the PSGP implementation to choose the offspring for mutation is described as follows:

1. Generate a random number for each offspring.
2. If the random number is smaller than the mutation probability $P_\mu$, the offspring is chosen for mutation.

For the offspring chosen for mutation, the mutation operator changes their block mappings, work remainings, and placement priorities. The block change re-maps the randomly selected functions into other blocks. The work remaining change re-assigns randomly selected functions to new work remainings. Equation 6.3 is used to compute values for the new work remainings. The placement priority change re-assigns new priorities to randomly selected functional unit modules. Equation 6.2 is used to compute values for the new placement priorities.

For example, if the mutation probability $P_\mu$ is set to 0.05, random numbers between 0 and 1 are generated for the two offspring in Figure 6.7, which are 0.78 and 0.02 respectively. Because the random number for offspring 2 is smaller than 0.05, offspring 2 is chosen to be mutated.

A random integer number between 1 and 19 is generated to select a gene in offspring 2’s block assignment part for mutation. Genes are the symbols or integers that make up a chromosome (see Section 6.1). Suppose the random number is 3. Another random integer number is generated to replace the old block assignment. If the random number is 1, gene 3’s block assignment value is changed to 1.
A random integer number between 1 and 19 is generated to select a gene in offspring 2’s work remaining part for mutation. Suppose the random number is 10. From Equation 6.3, it can be obtained that \( \text{WR}_{\text{max}} \) for offspring 2 is 32, and \( \delta_i \) is a random number between -16 and 16. A random number is generated for \( \delta_i \). If \( \delta_i \) is 8, the old work remaining value is changed from 2 to 40 for gene 10.

An integer number between 1 and 2 is generated to select a gene in offspring 2’s placement priority part for mutation. Suppose the random number is 2. From Equation 6.2, it can be obtained that \( \text{PR}_{\text{max}} \) for offspring 2 is 98, and \( \gamma_i \) is a random number between -49 and 49. A random number is generated for \( \gamma_i \). If \( \gamma_i \) is -30, the old placement priority value is changed from 98 to 68 for gene 2.

The result of offspring 2 is shown in Figure 6.15.

![Figure 6.15](image)

**Figure 6.15** The mutation result for Figure 6.14 offspring 2. The mutated genes are shown in bold.

### 6.3.5 Next Generation Selector

Pairs of chromosomes are selected from the population to perform crossover operations in order to produce offspring in each generation/iteration. A small percentage of the offspring are mutated. A fixed number of chromosomes is selected from the current population and the offspring to form the population for the next generation. The selection is based on their costs. For example, if the population size is \( N_p \), and \( N_o \) of offspring are generated, in order to keep the population size as a constant \( N_p \), the chromosomes in the
current population and their offspring are sorted according to their costs. The best \( N_p \) chromosomes are chosen to make up the next generation population.

Based on prior experimental work, Alander [Alan92] suggested that the population size, \( N_p \), is set to a value between the number of genes and twice of the number of genes in the chromosomes. For example, in Figure 6.6 the number of genes is 40, which is the sum of the number of genes in block assignment, work remaining, and placement priority, the suggested population size is between 40 and 80. However, for most of the GA applications, typical population sizes are between 10 and 50 [SaYo98], and the number of offspring, \( N_o \), is usually equal to the population size \( N_p \).

### 6.3.6 Stopping Criterion

The Genetic Algorithm is an iterative algorithm, and a stopping criterion must be provided to halt iterations at some point. The stopping criterion can be a function of the solution quality, the available runtime, the number of generations, or no reduction in cost for a certain number of generations. Using the number of generations is a common approach and a simple way to specify the stopping criterion. If the cost is not satisfactory, we can gradually increase the number of generations in each run until the desired solution is obtained.

For the examples given in this chapter, the number of generations and available runtime are both good candidates for the stopping criterion, because they are easy to apply and make the computer runtime predictable. If the results are not satisfactory, we can either increase the number of generations or runtimes to achieve better results. These two criteria are used for the experiments that will be illustrated in the next chapter.

To summarize the Problem Space Genetic Partitioning, the pseudo-code is given in Figure 6.16, in which the stopping criterion is given as the number of generations.
Algorithm  Problem Space Genetic Partitioning

\( N_p \) = number of chromosomes in the population  
\( N_g \) = number of generations  
\( N_o \) = number of offspring  
\( P_u \) = mutation probability

\[
P = \text{Init\_Population}(); \quad \text{// Init\_Population()} \text{ is a procedure to generate initial population}
\]

\[
\text{CalCost}(P); \quad \text{// CalCost()} \text{ is a function used to estimate the cost of each chromosome}
\]

\[
\text{Evaluate\_Fitness}(P); \quad \text{// Evaluate\_Fitness is used to calculate the fitness value of each chromosome}
\]

\[
\text{For } i=1 \text{ to } N_g \\
\quad \text{For } j=1 \text{ to } N_o/2; \quad \text{// each iteration generates 2 offspring so divide } N_o \text{ by 2}
\quad \quad \text{Mm = Select\_Parent}(); \quad \text{// select two parents for crossover operation}
\quad \quad \text{Mf = Select\_Parent}(); \quad \text{// based on their fitness values}
\quad \quad \text{Oj , Oj+1 = Crossover(Mm,Mf); \quad // 2 offspring Oj and Oj+1 are generated}
\quad \text{End For}
\quad \text{For } j=1 \text{ to } N_o
\quad \quad x=\text{random}(); \quad \text{// get a random number}
\quad \quad \text{if}(x<P_u) \text{ Mutation}(Oj); \quad \text{// if } x<P_u \text{, then mutate the offspring Oj}
\quad \text{End For}
\quad \text{CalCost}(O); \quad \text{// estimate the costs of each chromosome in the offspring}
\quad \text{P = Selection(P,O); \quad // select the chromosomes from the current population and}
\quad \text{\quad their offspring to form the next generation population}
\quad \text{Evaluate\_Fitness}(P); \quad \text{// The fitness values for each chromosome in the new population}
\quad \text{\quad have to be re-computed, because the maximum cost (C_max)}
\quad \text{\quad may be changed. See Equation 6.4.}
\quad \text{End For}
\]

\[
\text{End Algorithm}
\]

\[
\text{Return the chromosome with the lowest cost in } P
\]

\[
\text{End Algorithm}
\]

\[
\text{Figure 6.16 Pseudo-Code for Problem-Space Genetic Partitioning.}
\]
Chapter 7

Solution Quality Metric and Experimental Results

In Section 5.7, a cost function was presented for evaluating the quality of partition results for the partitioned-bus architecture. The cost function is also used by the Problem Space Genetic Partition algorithm (PSGP) to search for a partition result requiring minimum cost. PSGP is a heuristic algorithm; therefore, there is no guarantee that it will find an optimal solution.

For algorithms developed to address NP-complete problems, one way to compare the performance of different algorithms is by using well-known benchmarks as inputs and then comparing the output results. However, most of the prior research on partitioning only use the size of edge cut-sets between blocks in the cost function [GajD92, KeLi70]. It is not realistic to use their results to evaluate PSGP because PSGP takes many other issues into consideration. To cope with this problem, a model is constructed and test cases are generated to evaluate the solution qualities of the PSGP algorithm. The model and the test cases are presented in Section 7.1.

In Section 7.2, a comparative study of functional partitioning that uses PSGP and Simulated Annealing (SA) is presented. Several test cases derived from Electromagnetic Wave and Field, Digital Signal Processing, and Image Processing are used as inputs. The
complexities of these test cases, which are represented by CDFGs, range from 19 to 241 operations/functions, and from 32 to 240 data dependencies. Because simulated annealing is a widely used search algorithm for NP-compete problems, it is interesting to compare the results generated by PSGP and simulated annealing.

7.1 Solution Quality Metric

The purpose of proposing a solution quality metric is to evaluate the solution qualities obtained by using PSGP, and thereby validate the PSGP. Several artificially generated CDFGs with known optimal solutions are used as the test cases to verify the solution qualities that PSGP can attain.

7.1.1 Model for Solution Quality Measurement

Suppose that the optimal partition for a given CDFG is \( C^* \), and the partition result generated by PSGP is \( C \). A solution quality metric that can be used is:

\[
Q = (1 - \frac{C - C^*}{C^*})100\%
\]  

(7.1)

To obtain \( Q \), the optimal solution \( C^* \) must first be known. However, this is not feasible because obtaining \( C^* \) is an NP-complete problem itself. To get around this problem, artificially generated CDFGs for which the optimal solution is known a priori can be used [SaAl96]. In order to prevent Equation 7.1 from becoming a negative number, the cost \( C \) must be less than twice the cost of \( C^* \), i.e., \( C < 2C^* \). Otherwise, the solution is not acceptable and should be rejected because of the poor solution quality. The procedure for generating the CDFGs is described as follows.

From an artificially generated CDFG, if the functions in the CDFG are executed by only one functional unit, the system delay \( T \) is the sum of execution delays of all the functions,
which is denoted as $\lambda$. For an $N$_way partitioning, $N$ copies of the CDFG are used as the input (disjoint graph). Because there are no edges, i.e. inter-communication, among those identical CDFGs, it is obvious that the optimal solution is:

- communication buffer size $R = 0$,
- system delay $T = \lambda$,
- number of buses $B = 0$,
- number of links $K = 0$, and
- number of multiplexers $M = 0$ (see Equation 5.5).

Therefore, the cost $C^*$ for the optimal $N$_way partitioning becomes

$$C^* = W_r R + W_l L + W_t T + W_b B + W_k K + W_m M$$

(7.2)

$$= W_r 0 + W_l L + W_t \lambda + W_b 0 + W_k 0 + W_m 0$$

$$= W_l L + W_t \lambda.$$

The only unknown in Equation 7.2 is the size of the register file $L$. $W_r$, $W_l$, $W_t$, $W_b$, $W_k$, and $W_m$ are weights for adjusting the importance of the communication buffer size, register file size, system delay, number of buses, number of links, and number of multiplexers respectively. To obtain the optimal $L$ is an NP-complete problem itself [GajD92]. However, if $W_l$ is set to 0 in the cost estimation, Equation 7.2 becomes $C^* = W_t \lambda$. $C$ is obtained after using the same CDFG as the input to PSGP. The quality of the solution is then measured by using Equation 7.1.

### 7.1.2 Experimental Results

The PSGP algorithm is coded using C++ and compiled using GNU g++. The users must provide the following data as inputs for the program: (1) a CDFG, (2) the number of blocks of the partition, (3) a population size, (4) weights for $W_r$, $W_l$, $W_t$, $W_b$, $W_k$, and $W_m$, and (5) a stopping criteria, which can be a runtime or the number of generations. The
outputs of the program are a partition of the CDFG, the communication buffer and register file sizes, system delay, number of buses, number of links, number of multiplexers, and cost.

Eight artificially generated CDFGs, whose optimal solutions are known, are used to determine the best solutions PSGP can achieve for a given memory constraint, i.e., the population size, and a timing constraint, i.e., the number of generations. The complexities of the CDFGs are from 24 functions to 110 functions. These test cases were run on SUN UltraSparc Workstations, and the runtimes are in the order of minutes.

<table>
<thead>
<tr>
<th>CDFG</th>
<th>Number of Partition Blocks</th>
<th>Number of Nodes</th>
<th>Optimal</th>
<th>PSGP</th>
<th></th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Communication Buffer Size</td>
<td>System Delay</td>
<td>Cost ( C* )</td>
<td>Communication Buffer Size</td>
</tr>
<tr>
<td>gh_1</td>
<td>2</td>
<td>24</td>
<td>0</td>
<td>121</td>
<td>121</td>
<td>0</td>
</tr>
<tr>
<td>gh_2</td>
<td>2</td>
<td>48</td>
<td>0</td>
<td>241</td>
<td>241</td>
<td>5</td>
</tr>
<tr>
<td>gh_3</td>
<td>2</td>
<td>96</td>
<td>0</td>
<td>481</td>
<td>481</td>
<td>12</td>
</tr>
<tr>
<td>ph_1</td>
<td>2</td>
<td>22</td>
<td>0</td>
<td>275</td>
<td>275</td>
<td>2</td>
</tr>
<tr>
<td>ph_2</td>
<td>2</td>
<td>66</td>
<td>0</td>
<td>823</td>
<td>823</td>
<td>11</td>
</tr>
<tr>
<td>ph_3</td>
<td>2</td>
<td>110</td>
<td>0</td>
<td>1371</td>
<td>1371</td>
<td>12</td>
</tr>
<tr>
<td>ph_4</td>
<td>4</td>
<td>44</td>
<td>0</td>
<td>275</td>
<td>275</td>
<td>9</td>
</tr>
<tr>
<td>ph_5</td>
<td>4</td>
<td>88</td>
<td>0</td>
<td>549</td>
<td>549</td>
<td>31</td>
</tr>
</tbody>
</table>

In this experiment, the population size, Np, is set to 30, and the number of generations, Ng, is set to 1000. For the weights used in the cost function, W_r is set to 5, W_t is set to 1, and the other weights are set to 0. In this weight setting, the costs for C* and C can be denoted as λ and 5R + T respectively. The CDFGs for test cases, gh_1 to gh_3 and ph_1 to ph_5, are shown in Appendix Figure A.1-8. In Table 7.1, the sizes of the communication buffer and system delays that contribute to the cost evaluation are listed. The results show that all the test cases can reach about 87% of the optima for two-way partitioning. As for the four-way partitioning, the results are not as good as those for two-way partitioning. The reason is that the functions in the CDFG have more freedom to be
assigned into different blocks. This implies that evolution time, i.e., number of generations, or the population size has to be increased to get better solutions for four-way partitioning.

7.2 Problem Space Genetic Partitioning vs. Simulated Annealing

Simulated annealing, which was discussed in Section 2.4.1, is a heuristic algorithm that has been applied in many hard combinatorial optimization problems. Examples are the functional unit allocation problem in high-level synthesis and floorplan in VLSI design [Nedv89, LiWo86]. In this section, performance of a PSGP algorithm is compared with simulated annealing using the same cost function addressed in Section 5.8 to evaluate the solution qualities.

In the following sections, five test cases derived from real applications are used as inputs to PSGP and simulated annealing. The results are compared and discussed. For all the experiments, the population size was set to 50. The initial temperature for simulated annealing was determined by the procedure described in Section 3.4.1. The temperature cooling rate was set to 0.95, which indicates that the temperature used for each subsequent iteration is 95% of the current temperature. Fifty solutions were examined for each temperature. The weights were set to 1, 1, 10, 10, 1, and 1 for $W_r$, $W_l$, $W_t$, $W_b$, $W_k$, and $W_m$, respectively in the cost function, which indicates that the system delay and number of buses were more important than the others criteria.

7.2.1 Electrical Field Superposition

This test case is taken from Cheng’s book [Chen90], which is used to model the behavior of a uniform plane electromagnetic wave that is incident obliquely on a plane conducting surface. The CDFG which contains 19 functions/nodes is shown in Figure 6.6.
In this test case, the run times are given as 100 and 200 seconds respectively for two-way and four-way partitioning. The experimental results of two- and four-way partitioning for PSGP and SA are illustrated in Table 7.2. Table 7.2 shows that PSGP attains partitions with lower costs, i.e. better solution qualities, than SA can attain. Figure 7.1 and Figure 7.2 are the plots of the improvement of solution qualities for two- and four-way partitioning as the PSGP and SA proceed. Both Figure 7.1 and Figure 7.2 show that PSGP attains solutions with lower cost than those for SA.

Table 7.2 Comparisons of two- and four-way partitioning for PSGP and SA on Electrical Field Superposition.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Number of Partition Blocks</th>
<th>Algorithm</th>
<th>Cost</th>
<th>Delay</th>
<th>Register File</th>
<th>Communication Buffer</th>
<th>Bus</th>
<th>Link</th>
<th>Multiplexer</th>
<th>Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Electrical Field Superposition</td>
<td>2</td>
<td>PSGP</td>
<td>134</td>
<td>84</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SA</td>
<td>135</td>
<td>87</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Electrical Field Superposition</td>
<td>4</td>
<td>PSGP</td>
<td>130</td>
<td>60</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SA</td>
<td>163</td>
<td>59</td>
<td>6</td>
<td>6</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.1 Relative performance of PSGP and SA for two-way partitioning on Electrical Field Superposition.
Figure 7.2  Relative performance of PSGP and SA for four-way partitioning on Electrical Field Superposition.

7.2.2  Elliptic Filter

This test case is a fifth-order elliptic digital filter whose behavioral description contains 33 functions/nodes [KuWh85]. The CDFG for this test case is shown in Appendix Figure A.9.

In this test case, the run times are given as 300 and 600 seconds respectively for two-way and four-way partitioning. The experimental results of two- and four-way partitioning for PSGP and SA are illustrated in Table 7.3. Table 7.3 shows that PSGP attains better solution qualities in the given run times.
Table 7.3  Comparisons of two- and four-way partitioning for PSGP and SA on Elliptic Filter.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Number of Partition Blocks</th>
<th>Algorithm</th>
<th>Cost</th>
<th>Delay</th>
<th>Register File</th>
<th>Communication Buffer</th>
<th>Bus</th>
<th>Link</th>
<th>Multiplexer</th>
<th>Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elliptic Filter</td>
<td>2</td>
<td>PSGP</td>
<td>339</td>
<td>301</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>300</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SA</td>
<td>392</td>
<td>311</td>
<td>7</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>PSGP</td>
<td>301</td>
<td>231</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>600</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SA</td>
<td>390</td>
<td>231</td>
<td>5</td>
<td>10</td>
<td>4</td>
<td>10</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.3 and Figure 7.4 are the plots of the improvement of solution qualities for two- and four-way partitioning as the PSGP and SA proceed. Figure 7.3 and Figure 7.4 shows PSGP attains lower costs in both two- and four-way partitioning.

Two-way partitioning

![Two-way partitioning graph](image)

**Figure 7.3**  Relative performance of PSGP and SA for two-way partitioning on Elliptic Filter.
Figure 7.4 Relative performance of PSGP and SA for four-way partitioning on Elliptic Filter.

7.2.3 Fresnel Transition Function

The CDFG used to describe the behavior of the Fresnel Transition Function is given in Appendix Figure A.10 [Bala89]. There are 20 functions/nodes in the CDFG. The run times are given as 180 and 360 seconds respectively for two- and four-way partitions. The experimental results are illustrated in Table 7.4, which shows that PSGP converges to costs 153 and 138 and SA converges to cost 165 and 186 respectively in two- and four-way partitioning.

Figure 7.5 and Figure 7.6 are the plots of the costs of the solution for two- and four-way partitioning as the PSGP and SA proceed. In two-way partitioning, PSGP and SA reach
their lowest costs at about the same time. However, in both two- and four-way partitioning, PSGP obtains better results.

**Table 7.4** Comparisons of two- and four-way partitioning for PSGP and SA on Fresnel Transition Function.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Number of Partition Blocks</th>
<th>Algorithm</th>
<th>Cost</th>
<th>Delay</th>
<th>Register File</th>
<th>Communication Buffer</th>
<th>Bus</th>
<th>Link</th>
<th>Multiplexer</th>
<th>Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fresnel Transition Function</td>
<td>2</td>
<td>PSGP</td>
<td>153</td>
<td>101</td>
<td>11</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>180</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SA</td>
<td>165</td>
<td>115</td>
<td>9</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>PSGP</td>
<td>138</td>
<td>84</td>
<td>12</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>360</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SA</td>
<td>186</td>
<td>81</td>
<td>10</td>
<td>6</td>
<td>3</td>
<td>5</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Two-way partitioning**

![Two-way partitioning graph](image)

**Figure 7.5** Relative performance of PSGP and SA for two-way partitioning on Fresnel Transition Function.
7.2.4 Wedge Diffraction Coefficients

The CDFGs shown in Appendix Figure A.11(a)-(c) are used to represent the behavior of calculating the Wedge Diffraction Coefficients [Bala89]. The CDFGs contains 150 functions/nodes. Four- and eight-way partitioning are performed on this test case. The run times are given as 5 and 8 hours respectively. The experimental results are illustrated in Table 7.5. Table 7.5 shows that PSGP converges to costs 1218 and 1380 and SA converges to cost 1367 and 1563 respectively in four- and eight-way partitioning.

Figure 7.7 and Figure 7.8 are the plots of the costs of the solution for four- and eight-way partitioning as the PSGP and SA proceed. In four-way partitioning, SA converges to cost 1367 in 2.26 hours, and, after that, there is no further improvement. PSGP does not converge to an asymptote during the 5 hours run time. PSGP lowers the cost gradually until it reaches 1218 at the end of the run time. In eight-way partitioning, PSGP also attains better results than SA does.
Table 7.5  Comparisons of four- and eight-way partitioning for PSGP and SA on Wedge Diffraction Coefficients.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Number of Partition Blocks</th>
<th>Algorithm</th>
<th>Cost</th>
<th>Delay</th>
<th>Register File</th>
<th>Communication Buffer</th>
<th>Bus</th>
<th>Link</th>
<th>Multiplexer</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wedge Diffraction Coefficients</td>
<td>4</td>
<td>PSGP</td>
<td>1218</td>
<td>356</td>
<td>33</td>
<td>75</td>
<td>4</td>
<td>12</td>
<td>27</td>
<td>5 (hr.)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SA</td>
<td>1367</td>
<td>434</td>
<td>34</td>
<td>81</td>
<td>5</td>
<td>14</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>PSGP</td>
<td>1380</td>
<td>256</td>
<td>30</td>
<td>92</td>
<td>9</td>
<td>52</td>
<td>32</td>
<td>8 (hr.)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SA</td>
<td>1563</td>
<td>290</td>
<td>26</td>
<td>106</td>
<td>10</td>
<td>54</td>
<td>33</td>
<td></td>
</tr>
</tbody>
</table>

Four-way partitioning

Figure 7.7  Relative performance of PSGP and SA for four-way partitioning on Wedge Diffraction Coefficients.
Figure 7.8 Relative performance of PSGP and SA for eight-way partitioning on Wedge Diffraction Coefficients.

7.2.5 Edge Detector

This test case is taken from Haralick and Shapiro’s book [HaSh92], which applies the Laplacian of Gaussian zero-crossing operator to a digitized image in order to detect the edges. The mask for the Laplacian of Gaussian kernel is an 11 by 11 matrix. The behavior of the edge detector is described in Appendix Figure A.12 that shows 241 functions/nodes in the CDFG. Four- and eight-way partitioning are explored on this test case, and the run times are set to 5 and 8 hours respectively. The experimental results are shown in Table 7.6, which illustrates that PSGP can attain better results than SA can in the given run times in both four- and eight-way partitioning.
Table 7.6  Comparisons of four- and eight-way partitioning for PSGP and SA on Edge Detector.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Number of Partition Blocks</th>
<th>Algorithm</th>
<th>Cost</th>
<th>Delay</th>
<th>Register File</th>
<th>Communication Buffer</th>
<th>Bus</th>
<th>Link</th>
<th>Multiplexer</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge Detector</td>
<td>4</td>
<td>PSGP</td>
<td>949</td>
<td>191</td>
<td>62</td>
<td>62</td>
<td>5</td>
<td>16</td>
<td>10</td>
<td>5 (hr.)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SA</td>
<td>1091</td>
<td>185</td>
<td>49</td>
<td>78</td>
<td>5</td>
<td>16</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>PSGP</td>
<td>1241</td>
<td>154</td>
<td>42</td>
<td>89</td>
<td>10</td>
<td>48</td>
<td>7</td>
<td>8 (hr.)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SA</td>
<td>1387</td>
<td>159</td>
<td>27</td>
<td>104</td>
<td>9</td>
<td>56</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.9 and Figure 7.10 are the plots of the costs of the solution for four- and eight-way partitioning as the PSGP and SA proceed. In four-way partitioning, SA converges to a cost 1091 after 1.5 hours, and there is no further improvement after that. For PSGP, there is little improvement in the first 30 minutes, but the cost drops dramatically after that and then decreases gradually until the run time ends. In eight-way partitioning, SA shows very little improvement throughout the eight hours of run time. PSGP is able to gradually lower the costs and reaches cost 1241 when the run time ends. In four- and eight-way partitioning, PSGP attains lower cost solutions than SA can attain.
Figure 7.9 Relative performance of PSGP and SA for four-way partitioning on Edge Detector.

Figure 7.10 Relative performance of PSGP and SA for eight-way partitioning on Edge Detector.
7.2.6 Experiment-Result Summary

The summary of the performance comparisons between PSGP and SA on the test cases demonstrated in previous sections is shown in Table 7.7. The performance is calculated using the following formula:

\[
\left(\frac{\text{SA}_{\text{cost}} - \text{PSGP}_{\text{cost}}}{\text{SA}_{\text{cost}}}\right) \times 100\% \tag{7.3}
\]

where \(\text{SA}_{\text{cost}}\) and \(\text{PSGP}_{\text{cost}}\) are the costs that SA and PSGP can attain. Table 7.7 shows that PSGP outperforms SA in all the test cases, and the improvement attributable to PSGP can be as high as 25.8% over SA.

**Table 7.7 Performance comparisons for Problem Space Genetic Partitioning and Simulated Annealing.**

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Number of Partition Blocks</th>
<th>Cost</th>
<th>Performance PSGP over SA %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>PSGP</td>
<td>SA</td>
</tr>
<tr>
<td>Electrical Field</td>
<td>2</td>
<td>134</td>
<td>135</td>
</tr>
<tr>
<td>Superposition</td>
<td>4</td>
<td>130</td>
<td>163</td>
</tr>
<tr>
<td>Elliptic Filter</td>
<td>2</td>
<td>339</td>
<td>392</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>301</td>
<td>390</td>
</tr>
<tr>
<td>Fresnel Transition</td>
<td>2</td>
<td>153</td>
<td>165</td>
</tr>
<tr>
<td>Function</td>
<td>4</td>
<td>138</td>
<td>186</td>
</tr>
<tr>
<td>Wedge Diffraction</td>
<td>4</td>
<td>1218</td>
<td>1367</td>
</tr>
<tr>
<td>Coefficients</td>
<td>8</td>
<td>1380</td>
<td>1563</td>
</tr>
<tr>
<td>Edge Detector</td>
<td>4</td>
<td>949</td>
<td>1091</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1241</td>
<td>1387</td>
</tr>
</tbody>
</table>
Chapter 8

Summary and Conclusions

This thesis has shown approaches for coping with the problem of increasing complexity of VLSI design. The materials has focused on three aspects: 1) capturing a higher level of abstraction, 2) using a new target architecture, and 3) using a new optimization technique. These three aspects are summarized as follows.

The transistor count for today’s VLSI technology reaches 40 million transistors on one chip. In order to successfully design a system with such complexity, new computer-aided design (CAD) tools are needed to capture the system specifications at higher levels of abstraction. The advantage of working on a higher level of abstraction is that the number of objects that designers have to manipulate is reduced so that more complex systems can be delivered in shorter periods of time. In Chapter 3, the functions that can be used to capture higher levels of abstraction are surveyed and categorized into an is-a hierarchy. Another use of the function taxonomy is to build a search engine that can assist designers in searching for the desired components from component libraries.

The target architectures currently used by High-Level Synthesis (HLS) tools are optimized for a system that contains only simple functional units to realize elementary functions; however, these target architectures are not suitable for systems that consist of
complex functional units used to realize complex functions. This is due to the major difference in execution delays between elementary functions and complex functions. The execution delays for elementary functions are fixed; however, the execution delays for complex functions are **bounded**. Actual delay is data dependent and can only be determined after the systems are implemented and then measured at run time. The partitioned-bus architecture was proposed in Chapter 4 to tackle the problem of complex functions with bounded delays.

The issues involved in synthesizing complex functions using the partitioned-bus architecture were addressed in Chapter 5. These issues are focused on the functional partitioning problem. Algorithms that are used to optimize several metrics that affect the solution qualities of functional partitioning were presented. The metrics included communication buffer size, register file size, system delay, the number of buses, the number of links, and the number of multiplexers. These metrics are used to form a cost function, which is utilized by the Problem Space Genetic Partitioning algorithm (PSGP) to search for a good solution.

Functional partitioning is an NP-complete problem. Algorithms proposed to address this problem use constructive or iterative refinement heuristic approaches, or a stochastic approach such as simulated annealing (SA). The disadvantage of heuristic approaches is that they frequently get trapped in local optima; therefore, the heuristic approaches are unable to search globally for a good solution. The disadvantage of simulated annealing is that it requires very long run times to get good results. A genetic algorithm is another stochastic approach, which performs searches on multiple solutions at the same time. Because a genetic algorithm keeps track of a number of solutions simultaneously, it requires shorter run times than simulated annealing does to get good solutions. A Problem Space Genetic Algorithm (PSGA) was used to cope with the problem of infeasible solutions generated by the crossover and mutation operators. Problem Space Genetic Partitioning (PSGP) is based on PSGA, which is used to address the functional partitioning problem that arises from realizing the CDFG representations to the
partitioning-bus architecture. The details of PSGP were presented in Chapter 6, which includes chromosome representations, fitness functions, crossover operators, mutation operators, etc.

The experimental results are shown in Chapter 7, which includes a solution quality metric to evaluate PSGP and a comparative study of PSGP and SA. Artificially generated test cases with known optimal solutions are used to evaluate the solution qualities that PSGP can attain under run time (i.e., number of generations) and memory space (i.e., population size) constraints. The results show that PSGP can reach about 87% of the optima for two-way partitioning for the test cases used.

In order to compare the results obtained by PSGP with the solutions obtained by using simulated annealing, the functional partitioning using simulated annealing was also implemented. The complexities of the test cases used in the study range from 19 to 241 functions in the CDFGs. The results show that PSGP outperforms SA, and the solution quality of PSGP can be as high as 25.8% than simulated annealing can achieve.
Appendix

A. Control Data Flow Graphs

The CDFGs used as the test cases in Chapter 7 are illustrated in the following figures. The CDFGs include gh_1, gh_2, gh_3, ph_1, ph_2, ph_3, ph_4, and ph_5 referenced in Section 7.1, and Elliptical Filter, Fresnel Transition Function, Wedge Diffraction Coefficients, and Edge Detector referenced in Section 7.2.

NOTE: All the functions are assumed to have the maximum delay of 10 clock cycles

Figure A.1 CDFG for gh_1.
Figure A.2  CDFG for gh_2.

NOTE: All the functions are assumed to have the maximum delay of 10 clock cycles.
NOTE: All the functions are assumed to have the maximum delay of 10 clock cycles.

**Figure A.3** CDFG for gh_3.
NOTE: Functions with shaded nodes have maximum delay of 24 clock cycles, and functions with transparent nodes have maximum delay of 26 clock cycles.

Figure A.4 CDFG for ph_1.
NOTE: Functions with shaded nodes have maximum delay of 24 clock cycles, and functions with transparent nodes have maximum delay of 26 clock cycles.

**Figure A.5** CDFG for ph_2.
NOTE: Functions with shaded nodes have maximum delay of 24 clock cycles, and functions with transparent nodes have maximum delay of 26 clock cycles.

Figure A.6 CDFG for ph_3.
Figure A.7  CDFG for ph_4.
**Figure A.8** CDFG for ph_5.

**NOTE:** Functions with shaded nodes have maximum delay of 24 clock cycles, and functions with transparent nodes have maximum delay of 26 clock cycles.
Figure A.9  CDFG for Elliptic Filter.
Figure A.10  CDFG for the Fresnel Transition Function.
NOTE: Continued on Figure A.11(b)

Figure A.11(a)  CDFG for Wedge Diffraction Coefficients (Part 1/3).
Figure A.11(b)  CDFG for Wedge Diffraction Coefficients (Part 2/3).

NOTE: Continued on Figure A.11 (c)
Figure A.11(c) CDFG for Wedge Diffraction Coefficients (Part 3/3).
Function “+” is repeated 117 times

Figure A.12 CDFG for Edge Detector.
B. List of Publications

The research results presented in this dissertation contribute to five publications, which are listed as follows:


References


