Intermediate Forms in High-Level Synthesis

BY

Essam Mohammad Khair Hubbi

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
In
Computer Engineering

November 1994

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN 31261, SAUDI ARABIA
COLLEGE OF GRADUATE STUDIES

This thesis, written by

Essam Mohammad Khair Hubbi

under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

Thesis Committee:

Sadiq Sait. M

Dr. Sadiq M. Sait (Chairman)

Dr. Habib Youssef (Co-Chairman)

Dr. Muhammad S. T. Benten (Member)

Department Chairman

Dean, College of Graduate Studies

Date

COLLEGE OF GRADUATE STUDIES

To my family

whose support and prayers

led to this achievement.

Acknowledgments

All praise be to Allah for his limitless help and guidance. Peace and blessings of Allah be upon his prophet Mohammad.

Acknowledgement is due to King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, for the generous help and support for this research.

I would like to express my profound appreciation to my advisor, Dr. Sadiq M. Sait, Associate Professor of Computer Engineering, for his guidance and patience throughout this thesis.

I would also like to thank co-chairman, Dr. Habib Youssef Assistant Professor of Computer Engineering, whose continuous encouragement can never be forgotten, and Dr. Muhammad S. T. Benten, Associate Professor of Computer Engineering for his consistent support and valuable suggestions. I also wish to thank faculty, research assistants, graduate assistants, and the staff members of the Computer Engineering Department for their support.

The encouragement and good wishes of the following friends in CCSE and other departments is also worthy of acknowledgement. They are: Hazem Abu-Saleh and Saleh Al-Meshari.

Finally, special thanks must be given to my family for their encouragement and moral support.

Contents

List of Figures

List of Tables

Abstract

Name: Essam Mohammad Khair Hubbi.
Title: Intermediate Forms in High-Level Synthesis.
Major Field: Computer Engineering.
Date of Degree: November, 1994.

High-level synthesis (HLS) is the automatic translation of a behavioral description into a structural description. This translation process is very complicated, therefore, it is broken into several tasks. Each task is performed with the help of some intermediate forms. In this work, a survey of various intermediate forms used in known high-level synthesis systems is introduced. Then, a classification framework that classifies intermediate forms into two main classes (primary and secondary) is introduced. After that, essential and desirable features of primary intermediate forms are identified. Finally, a new primary intermediate form called the Generic Control-Data Flow Graph (GCDFG) and its GCD-List notation are introduced. This GCDFG has all the desirable features of primary intermediate forms and facilitates synthesis tasks like scheduling and allocation. This GCDFG is expressed in the GCD-List notation which has a lisp-like format. This makes it rich and flexible format since attributes can be easily added and hence more constraints can be accomodated. Moreover, it is stored in ASCII text format which makes it portable and machine processible.

Master of Science Degree
King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia
November, 1994

الخلاصة

الإسم: عصام محمد خير حبي.

العنوان: الصيغ الوسيطية في التصميم عالي المستوى.
التخصص: هندسة حاسب آلي.
تاريخ منح الدرجة: نوفمبر ١٩٩٤ م.

التصميم عالي المستوى هو عملية التحويل الآلي للمواصفات العامة لسلوكيات الدوائر المتكاملة إلى التصميم التركيبي لتلك الدوائر. و لأن عملية التصميم عالي المستوى عملية معقدة، فإنه يتم تقسيم هذه العملية إلى عدد من المراحل. و تقوم هذه المراحل مجتمعة بعملية التحويل بالإستعانة ببعض الصيغ الوسيطية.

يقوم هذا البحث بدراسة عدد من الصيغ الوسيطية في أنظمة التصميم عالي المستوى المعروفة. كما يقوم هذا البحث بوضع إطار تصنيفي تصنف على أساسه هذه الصيغ الوسيطية إلى صنفين (أساسية و ثانوية). كما يتم التعرف على السمات الضرورية و الإضافية للصيغ الوسيطية الأساسية.

كما يقدم هذا البحث صيغة وسيطية أساسية مبتكرة تسمى رسم إنسياب بيانات التحكم العام تخزن على شكل لغة تسمى لائحة إنسياب بيانات التحكم العامة. و تشبة لائحة إنسياب بيانات التحكم العامة إلى حد كبير لغة "ليسب" مما يجعلها لغة غنية و مرنة ومساعدة في مراحل التصميم عالي المستوى المختلفة مثل: جدولة العمليات و توزيع المكونات. بالإضافة إلى كل ماذكر تخزن لائحة إنسياب بيانات التحكم العامة بشكل نصي مما يجعلها سهلة النسخ و النقل و قابلة للمعالجة الآلية .

ماجستير في علوم هندسة الحاسب الآلي
جامعة الملك فهد للبترول و المعادن
الظهران - المملكة العربية السعودية
نوفمبر ١٩٩٤ م

Chapter 1

Introduction

Design automation is the automatic synthesis of a physical design from some higher-level behavioral specification. There are three levels of synthesis namely: high-level synthesis, logic synthesis and layout synthesis. Logic synthesis converts a structural design, in terms of an interconnected set of register transfer level components, into combinational logic, and maps that logic onto cells from a library of a particular technology. Layout synthesis converts an interconnected set of cells, which describes the design structures, into the physical geometry (layout) of the design. An integrated system that contain all three synthesis levels is called *silicon compiler* (see Figure 1.1).

In this work, various intermediate forms from known high-level synthesis systems are surveyed. A classification framework that classifies intermediate forms into two main classes (primary and secondary) is introduced. Essential and desir-

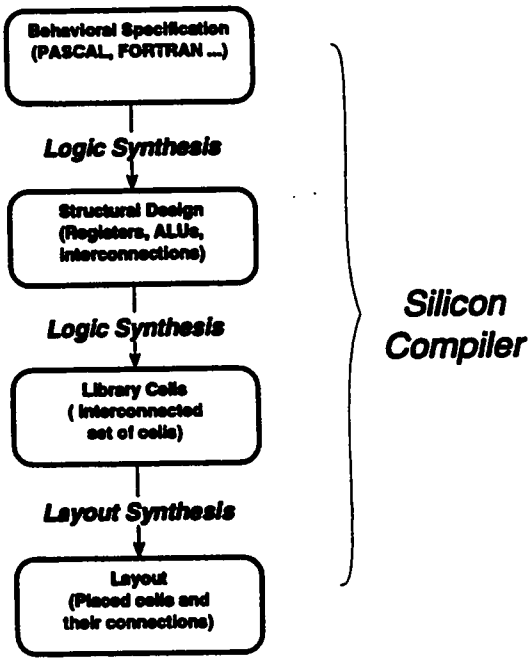

Figure 1.1: Synthesis levels and silicon compiler.

able features of primary intermediate forms are identified. Finally, a new primary intermediate form called Generic Control-Data Flow Graph (GCDFG) is introduced. This GCDFG has all the desirable features of primary intermediate forms and facilitates synthesis tasks like scheduling and allocation. This GCDFG is stored in ASCII, lisp-like format which makes it portable, machine processible, flexible and rich format.

This work is divided into five chapters. In chapter one, high-level synthesis is defined and basic terminology like: DFG, CFG, CDFG, transformation, scheduling, allocation etc., are explained. In the second chapter, some known high-level synthesis systems that use various intermediate forms are surveyed. Following the survey, a general classification framework, that classify intermediate forms into two classes: *primary* and *secondary*, is introduced. The essential and desirable features that should be possessed by a *primary* intermediate form are identified. Finally, a new intermediate form that is called the GCDFG is introduced. Based on this new intermediate form, an attributed format called GCD-List is introduced.

In this introductory chapter, abstraction levels and high level synthesis are defined. Following that, basic terminology like: data flow graph (DFG), control flow graph (CFG) and control-data flow graph (CDFG) are introduced and illustrated by examples. Finally, the main tasks of high-level synthesis are defined and explained briefly.

1.1 Definition of High-level Synthesis

Digital systems are represented at three abstraction levels: the behavioral level, the structural level and the physical level. The highest level of abstraction which is the *behavioral level*, corresponds to the high-level description of the system. Behavior means the way the system or its components interact with their environment [4]. At this level the system can be described in a high-level language like: FORTRAN [27], PASCAL[29], C... etc. The next lower level of abstraction is the *structural level*. At this level the system is described by its hardware components (registers, ALUs ...etc.,) and their interconnection (netlist). This level is also called the *Register Transfer Level* (RTL). Finally comes the lowest abstraction level which is called the *physical level*. At this level the digital system is interpreted by the chip layout [6].

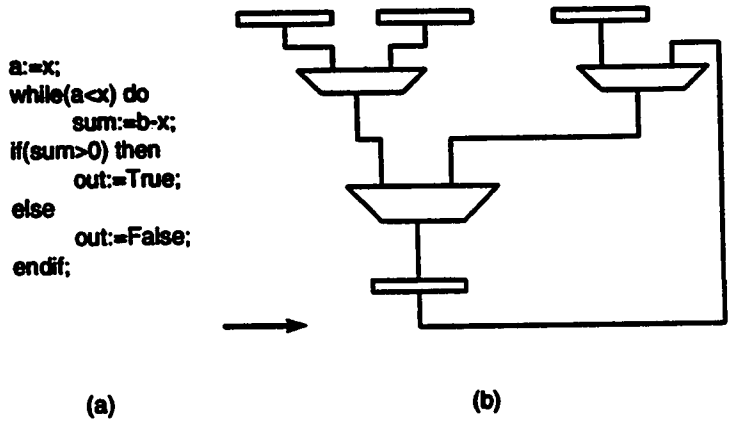

Figure 1.2: Digital system representation. (a) Behavioral level. (b) Structural level.

In general terms, high-level synthesis (HLS) can be defined as the automatic

translation of a high-level description (*behavioral level*) to a lower abstraction level (*structural level*). This translation is usually subject to specific constraints like (speed, cost, space ... etc.,) [5, 17, 10, 2](see Figure 1.2).

High-Level Synthesis (HLS) is a very complex process therefore, it is divided into tasks. The high-level description is first *transformed* into an internal representation (intermediate form (IF)). Next, operations are *scheduled* into control steps. Then, hardware resources like ALUs, registers ...etc., are *allocated* to execute these operations in the required control steps. Finally, a controller is synthesized to generate the control signals needed to invoke the allocated hardware components (see Figure 1.2). To perform the aforementioned tasks some intermediate forms are used to help the synthesis process. The most widely used intermediate forms are flow graphs like: data flow graphs (DFG), control flow graphs (CFG)... etc. DFG, CFG and CDFG intermediate forms are explained in detail below.

1.1.1 Date Flow Graph (DFG)

Data Flow Graphs are used to express the data flow of straight codes or basic blocks only. It can be defined as a sequence of statements that has one entry point and one exit point with no iterative, conditional or unconditional jumps. A Data Flow Graph consists of nodes and directed edges. Nodes represent operations as well as operands. The directed edges connect operand nodes to operation nodes or vice versa. A directed edge connecting a variable node to an operation node means this

operand is used as source and each incoming edge means this operand is used as a destination for that operation.

Mathematical and logical operations are either unary or binary. Unary operation nodes have one incoming edge. Binary operation nodes have two incoming edges (the two operands). If more than one operation are performed in the same instruction, some intermediate results have to be stored. These intermediate results are stored in intermediate variable nodes which are represented by small black dots (see Figure 1.3). In non-commutative binary operations like divide and subtract, a convention should be followed. For example $(c - d)$ in Figure 1.3, the first variable c is represented by an incoming edge from the left and the second variable d is represented by an incoming edge form the right.

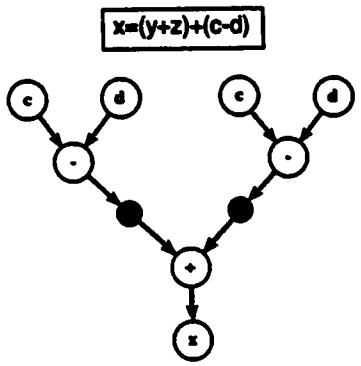

Figure 1.3: Data flow graph.

1.1.2 Control Flow Graph (CFG)

A Control Flow Graph (CFG) is a directed graph, where nodes correspond to operations and edges represent predecessor/successor relationships. There is one to one correspondence between the DFG operation nodes and the CFG nodes. An outgoing edge from node A to node B indicates that operation A is executed before operation B because node B is data dependent on node A (a destination variable in operation A is used as a source variable in operation B). The CFG is used to express conditional and iterative constructs. For Example, to implement an *if-then-else* statement, a *fork* node is used. The *fork* node, which carries the *if* condition, has two labeled outgoing edges, one for the *true* branch and one for the *false* branch. The condition of the *if* statement is implemented in the DFG. Then based on the condition value the corresponding branch is chosen (see Figure 1.4). Loops are implemented by feedback edges as in Figure 1.5.

1.1.3 Control-Data Flow Graph (CDFG)

A high-level synthesis system called ESPRIT [28] translates the high-level language into a Control-Data Flow Graph (CDFG) called *extended DFG* in which data and control flow graphs are combined. This *extended DFG* consists of nodes and directed edges. There are three main types of nodes: operation, operand and control nodes. Edges represent predecessor/successor relationships. Example of control edges are

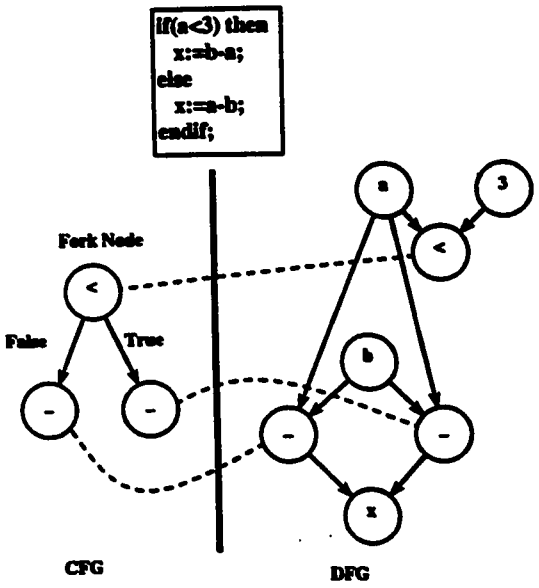

Figure 1.4: The CFG and the DFG representing an *if-then-else* statement.

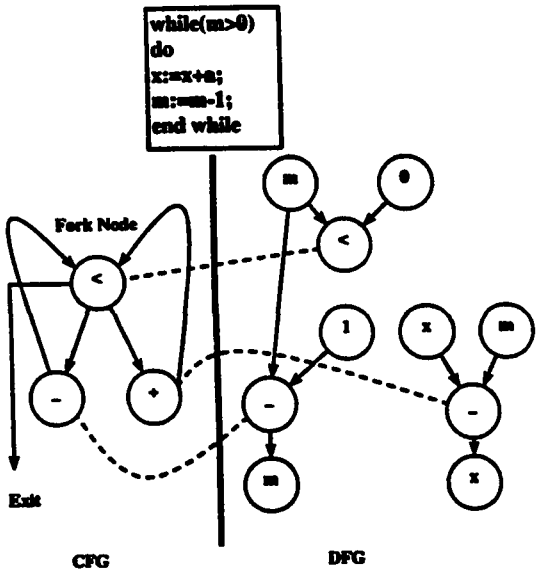

Figure 1.5: The CFG and the DFG representing a *while* loop.

feedback edges.

The CDFG nodes represent operations in the behavioral specification, and the edges model the transfer of values between operations. A single data value instance is defined to be a token. An operation is executed when a token passes from the incoming edge and exits via the outgoing edge. Thus, operations can be executed concurrently without violating data dependencies. Several node types are defined in this CDFG. These types of nodes allow the CDFG to support various high-level constructs. The node types are:

1. **Operation nodes.** These carry mathematical operations symbols like: $*, -, +, /$ or boolean like *and, or.*

2. **Input/Output nodes.** Every graph requires at least one node of type input and one of type output. Nodes of type input have no incoming edges and nodes of type output have no outgoing edge as in Figure 1.6. Input/Output nodes represent source variables and destination variables.

3. **Constant nodes.** These produce constant data values at their outgoing edge. These nodes have an incoming dashed edge to indicate the timing at which the constant should be generated.

4. **Branch and Merge nodes.** These are used in realizing conditional constructs like *if-then-else* statement. A branch node passes the token from the incoming edge to one output port, which is selected by the value of the token

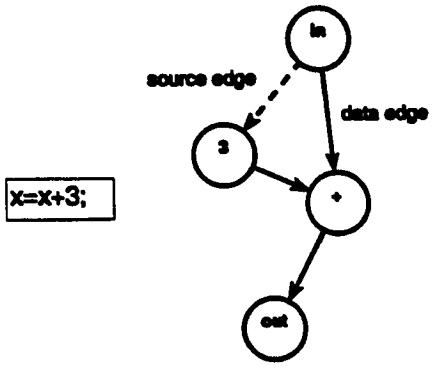

Figure 1.6: IN, OUT and constant nodes.

on the control input. In other words, the control input carries one value either 0 or 1, based on that the corresponding branch is selected. For example, in Figure 1.7 when $q = 1$, the left BR node passes the token to the increment node where a is incremented. When $q = 0$, the right BR node passes the token to the increment branch and b is incremented.

5. **Exit and Entry nodes** are similar to branch and merge nodes, however, they are used to build loop constructs. The loop body and the loop testing condition cause cycles in the CDFG. Using *exit* and *entry* nodes makes the identification of the cycles and hence the loop body and the loop condition easier (see Figure 1.8).

6. **Get and Put nodes** provide a mechanism for communication protocols with the outside world. These nodes are linked in a sequential chain to set the order in which read and write operations should appear on the port. These nodes

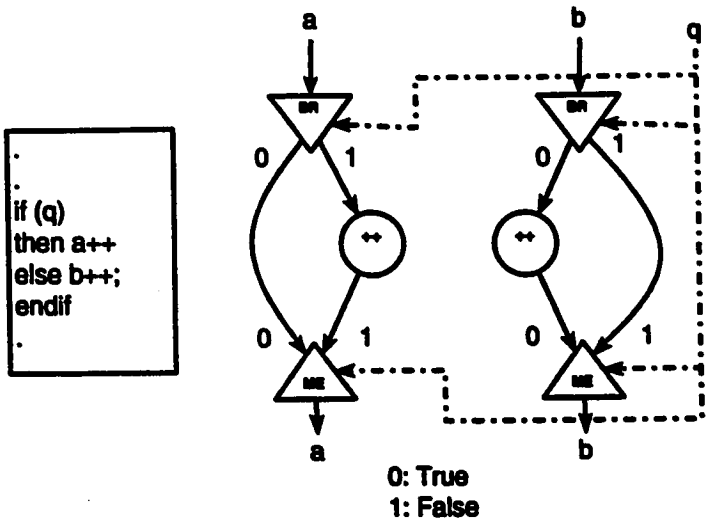

Figure 1.7: *If-then-else* construct.

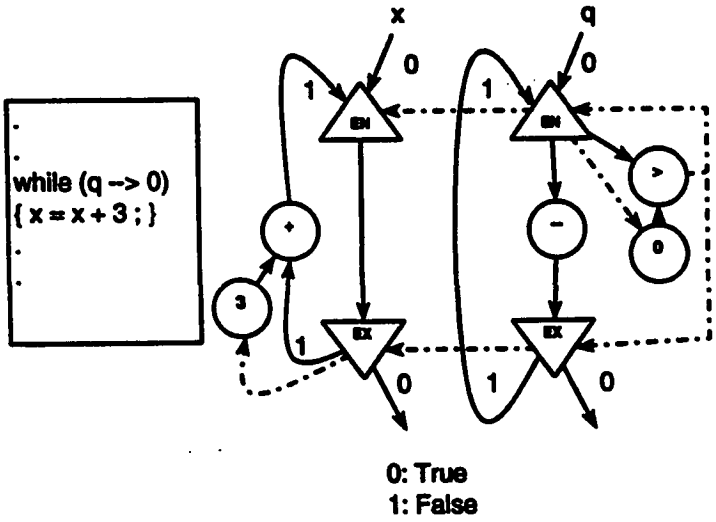

Figure 1.8: *While* loop with Entry and Exit nodes.

can for instance model pads on the chip boundary (see Figure 1.9).

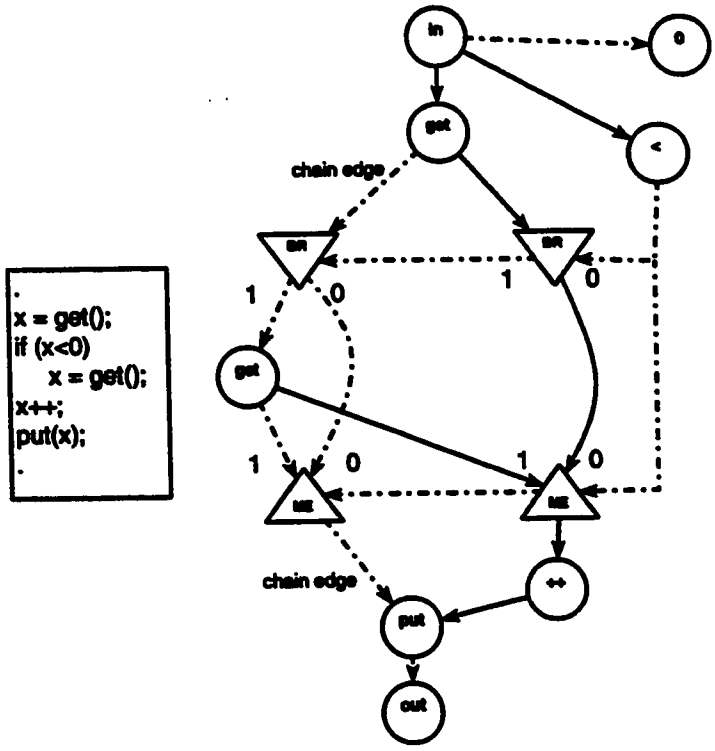

Figure 1.9: Example of Get and Put nodes.

7. **Array nodes** are used to manipulate array data values. There are basically three types of nodes: *Array* type is used for declaring the array size and initial values. It has outgoing chained edges which connect it to the other two types namely: *retrieve* and *update*. The *retrieve* type node is used to read data from the array. The *update* type node is used to write values to the array and it has a data input port that carries the value to be written (Figure 1.10). Chain edges specify the ordering in which the *retrieve* and *update* operations should

take place.

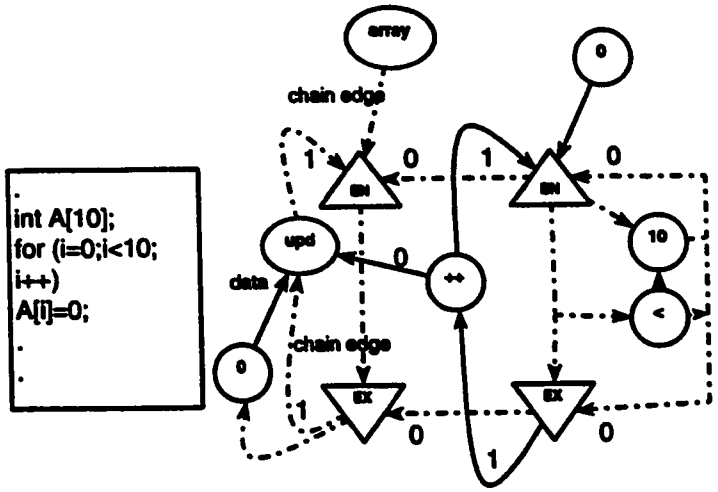

Figure 1.10: Array example.

1.2 The Main Tasks in HLS

High-level Synthesis (HLS) involves four main tasks namely:

1. Transformation or compilation of input specifications.

2. Scheduling.

3. Allocation and module binding.

4. Synthesis of control path and the merging of the control and data paths.

1.2.1 Transformation

This is the first step in HLS. It consists of parsing the high-level input description like behavioral VHDL [25], FORTRAN etc., and transforming it into an internal representation (intermediate form) such as DFG and CFG, CDFG, Value Trace etc. There are two main approaches in high-level language transformation:

- Two graphs are generated: one for Data (DFG) and one for Control (CFG).
- A Combined Control and Data Flow Graph (CDFG).

1.2.2 Scheduling

Scheduling means assigning operations to control steps so as to minimize a given objective function while meeting constraints. These constraints are usually speed and cost. For example, operations are assigned to control steps so as to maximize hardware resources sharing. Examples of some known scheduling approaches are listed below:

- Exhaustive search. In this method, all scheduling possibilities are tested and the best solution is found. This method finds the optimum solution, however, it has a very high complexity. It can be improved using some branch and bound techniques.
- As Soon As Possible (ASAP). Operations are assigned to their corresponding earliest possible control steps [22].

- As Late As Possible (ALAP). Operations are assigned to the latest possible control step [22].
- List Scheduling. The critical path is scheduled first. Operations to be scheduled in the next control step are ordered in a list based on some priority function like path length and urgency [20].
- Dynamic List Scheduling. DLS is oriented to control-flow dominated designs. It is a modified version of the known path-based scheduling approach [3]. The algorithm takes VHDL as input and produces a Finite State Machine (FSM) as output. Each transition in this FSM corresponds to a control step [9].
- Freedom-based scheduling. Operations on the critical path are scheduled first, then operations that have less freedom, and so on.
- Force Directed. In force-directed scheduling, a global time constraint is specified and the algorithm tries to minimize the resources required to meet that time constraint. By scheduling similar operations in different control steps, it is ensured that the functional units have high utilization and hence, concurrency is balanced [22, 23].

1.2.3 Allocation and Module Binding

Allocations means assigning operations, variables and interconnection to different hardware resources like registers, ALUs, multiplexers etc., while meeting constraints

to match a specific objective function such as minimizing total interconnection length, minimizing total hardware cost, minimizing critical path delays, minimizing design area and maximizing total throughput.

Some of the reported allocation algorithms are:

1. Clique partitioning. This method consists first of constructing a conflict graph, then using node coloring or clique partitioning techniques to allocate the minimum possible number of resources [31].

2. Cost function allocation. In this method each ALU is assigned a cost. After each allocation a cost formula is used to evaluate the cost of the overall ALU allocation. This process is repeated until a satisfactorily low cost allocation is reached. Allocation is conducted so as to minimize the overall cost [21].

1.2.4 Control Path Synthesis and Merging Data and Control Path

Scheduling and allocation produce a data-path and a finite state machine (FSM). In the FSM, the states correspond to control steps and the edges correspond to the conditions that cause transition from a state to another. From the FSM and the data path, the controller is synthesized. This controller is then interfaced with the data path to generate the signals that drive the data-path resources.

Chapter 2

Intermediate Forms and High-Level Synthesis Tasks

This chapter discusses the synthesis tasks and the intermediate forms used in these tasks in several high-hevel hynthesis systems. Different high level synthesis systems have been chosen to illustrate the various types of intermediate forms that are in use. It consists of three main sections. The first section surveys intermediate forms used in transformation. The second section discusses intermediate forms used in scheduling and the third section discusses intermediate forms used in allocation.

2.1 Intermediate Forms Used in Transformation

In this section, high-level languages and the intermediate forms used with them are surveyed. In high-level synthesis, the target design [12, 11] can be specified using:

1. a hardware description language like behavioral VHDL [26];

2. a high-level language: C, FORTRAN [27], PASCAL [29]... etc.

The aforementioned categories of languages share common characteristics. They describe data manipulation in terms of assignments of variables that keep their values until they are overwritten. Statements are *sequentially* organized in blocks linked by control transfer constructs like *conditional* constructs, and *looping*. *Hierarchy* is achieved by dividing large programs into subprograms (subroutines and procedures).

The study conducted here includes procedural languages only. However, there have been some experiments with nonprocedural specification languages, such as applicative (LISP) and declarative or rule based languages such as PROLOG [7]. In what follows, some HLS systems, their high-level languages and the supporting intermediate forms are surveyed. Three high-level synthesis systems are discussed. They are: HARP, Classical, and HAL. As will be seen later, these three are chosen because they use different approaches.

2.1.1 Transformation in HARP

The Hardware Architecture Ruling Processor (HARP) synthesis system uses a subset of the ANSI FORTRAN 77 [27] as a behavioral specification language. The FORTRAN code is translated into a DFG. Since DFG can only express basic blocks, the following assumptions and restrictions are observed:

1. Subroutines and functions can be used in the high-level description, however, they are flattened in the DFG.

2. Loops with indefinite iterations are not allowed and loops with finite length are enrolled.

3. Intrinsic (built-in) functions are not supported.

Before the FORTRAN code is translated into a DFG all procedures have to be expanded in-line and loops have to be unrolled. The FORTRAN code of Figure 2.1 is translated into a DFG in which nodes represent operations as well as operands and edges represent input/output relationships (see Figure 2.2).

DFG exposes the maximum potential parallelism of a high-level specification. For example, the basic block of Figure 2.1 can be realized by the DFG of Figure 2.2. From this DFG an equivalent parallel basic block can be written as in Figure 2.3. Instructions that are written in the same line are data independent and hence can be executed in parallel. Transforming a straight piece of code into a DFG is a

```
DATA I1,I2,I3,I4,I5,I6.I7,I8,I9,
& I10,I11,I12,I13,I14,I15
/ 1,2,3,4,5,6,7,8,9,
& 10,11,12,13,14,15 /
I3 = I1 + I2
I5 = I3 - I4
I7 = I3 * I6
I8 = I3 + I5
I9 = I1 + I7
I11 = I10 / I5
I12 = 100
I13 = I3
I12 = I1
I14 = IAND(I11,I8)
I15 = IOR(I12,I9)
I1 = I14
I2 = I15
STOP
END
```

Figure 2.1: High-level specification (basic block).

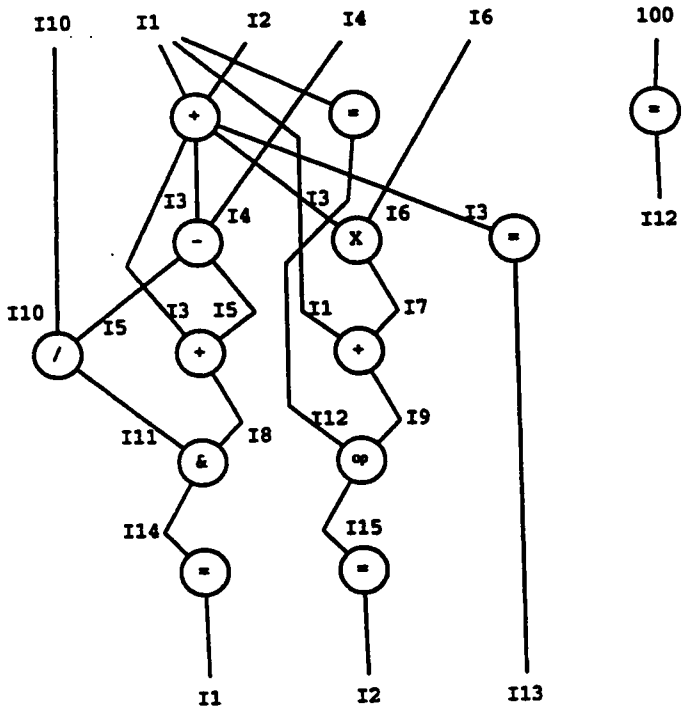

Figure 2.2: The corresponding DFG of the FORTRAN code of Figure 2.1.

```
I3 = I1 + I2;          I12 = I1;          I12 = 100;
I5 = I3 - I4;          I2 = I3 * I6;      I13 = I3;
I8 = I3 + I5;          I11 = I10 / I5;    I9 = I1 + I7;
I14 = IAND(I11,I8);    I15 = IOR(I12,I9);
I1 = I14;              I2 = I15;
```

Figure 2.3: Parallel code reconstructed from the DFG of Figure 2.2.

reversible process. In other words, an equivalent description of the original code can be recovered from the DFG.

2.1.2 Transformation in the Classical System

The classical system is a generic HLS system introduced in [32] by Camposano, where synthesis starts from a behavioral VHDL description. A DFG and a CFG are built, then scheduling and allocation are performed. The data-path is constructed from the DFG and the corresponding control path is constructed with the help of a FSM. Finally, the data-path and the control-path are merged together to form the RTL design of the input specification.

Behavioral VHDL can be represented graphically by an intermediate form called DAG (Directed Acyclic Graph). The graph nodes represent procedures and edges represent the calling relationships. Nodes and edges have attributes, examples of node attributes are procedure name, type of hardware (combinational or sequential).

Each VHDL procedure is transformed into a Data Flow Graph (DFG) and a

Control Flow Graph (CFG). DFG is a graph where nodes represent operations as well as operands and edges represent input/output relationships as in Figure 2.4. The small filled nodes in the DFG are those variables that do not exist in the high-

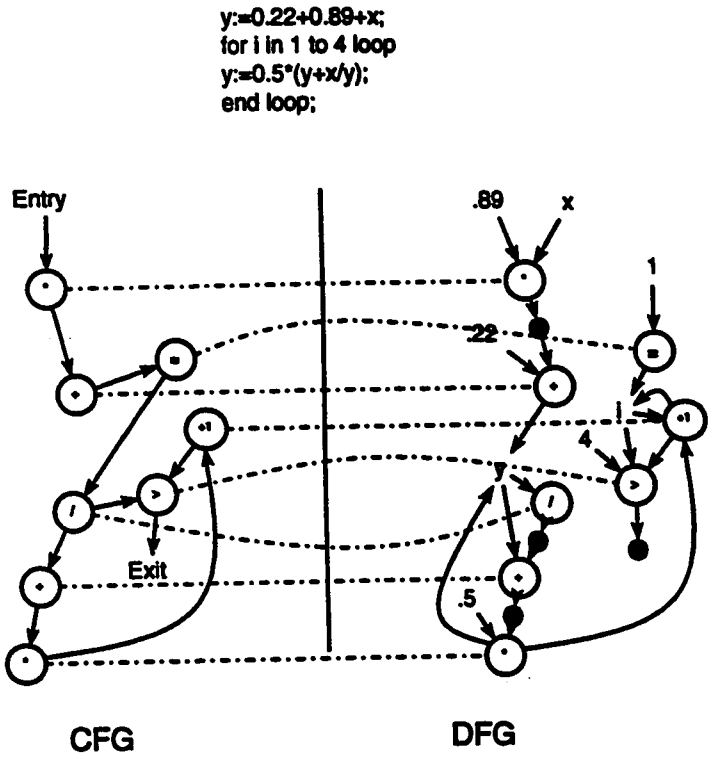

Figure 2.4: The corresponding DFG of the FORTRAN code of Figure 2.1.

level code. Operation nodes have operation type as attributes and variable nodes have type of variable as attribute. CFG is a directed graph where nodes represent operations and edges represent predecessor/successor relationship (Figure 2.4).

2.1.3 Transformation in HAL

In HAL [22] synthesis starts from a high-level language (HLL). The high-level language HLL which has a PASCAL like syntax and semantics is translated into a combined Control Data Flow Graph (CDFG) (see Figure 2.5). In the CDFG, nodes represent operations, constants, and variables. Edges represent input/output relationships and data dependency. Moreover, dummy timing nodes can be inserted between two nodes (operation); these are used to enforce a given execution order of the corresponding operations like forcing a specific operation to be executed before or concurrently with another operation. Conditional constructs like: *if-then-else* and *case* can be represented in the CDFG by attaching the *condition* to the edge. These require fork and join nodes as we will see later.

2.1.4 Summary

The results of the above study are summarized in Tables 2.1 and 2.2. Table 2.1 shows the high-level languages versus the features they support. In the Classical system the input description is VHDL which supports all the necessary high-level language features. In HAL synthesis system a generic PASCAL like language is used. The HLL in HAL supports all the necessary high-level language features except hierarchy where it is not clear from the literature whether it is supported or not. In HARP FORTRAN 77 is used as an input description language. All necessary high-level

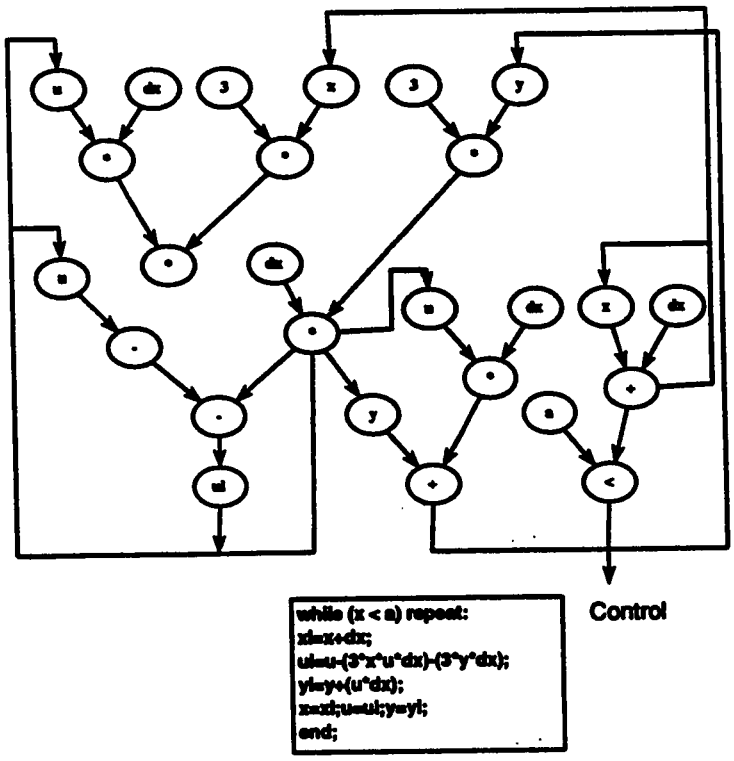

Figure 2.5: Combined Control and Data Flow Graph.

language features are supported in the FORTRAN 77 except concurrency. Finally, In ESPRIT a generic C-like language is used. All necessary high-level language features are supported except hierarchy and concurrency.

System	Classical	HAL	HARP	ESPRIT
Language	VHDL	HLL	FORTRAN	HLL
Sequencing	Yes	Yes	Yes	Yes
Hierarchy	Yes	-	Yes	-
Conditional	Yes	Yes	Yes	Yes
Looping	Yes	Yes	Yes	Yes
Concurrency	Yes	No	No	-

Table 2.1: High-level languages and their supporting features.

In Table 2.2, we summarize the features supported by each intermediate form. As seen clearly sequencing and concurrency of operations are supported by all types of flow graphs. Conditional constructs and looping are supported by DFG+CFG, CDFG and the extended DFG. Hence, DFG representation alone is not enough to realize conditional constructs and looping. It was not stated in most of the literature whether hierarchy is supported or not. However, in HARP it has been clearly stated that hierarchy is not supported and all subroutines should be in-line expanded before translation.

System	Classical	HAL	HARP	ESPRIT
Intermediate Form	DFG+CFG	CDFG	DFG	Extended DFG
Sequencing	Yes	Yes	Yes	Yes
Hierarchy	-	-	No	-
Conditional	Yes	Yes	No	Yes
Looping	Yes	Yes	No	Yes
Concurrency	Yes	Yes	Yes	Yes

Table 2.2: Intermediate forms and their supporting constructs.

2.2 Intermediate Forms Used in Scheduling

In this section, some scheduling techniques and their intermediate forms used in some high-level synthesis systems are discussed. The scheduling techniques that will be explained are as soon as possible (ASAP), path-based [3] and force-directed scheduling [22]. Scheduling means assigning operations to control steps so as to minimize a given objective function while meeting constraints. These constrains are usually speed and cost. Examples of speed constraints are number of control steps and length of the control step. Examples of cost constraints are number of functional units, registers, interconnections, buses and multiplexers.

There are two basic types of scheduling algorithms:

- **Transformational.** These are the ones that begin with a default schedule (maximally parallel or serial). Then several transformations are applied to get the final schedule. Examples of transformational scheduling algorithms are: the exhaustive search and the ad-hoc heuristics such as the one used in HARP

[27].

- **Iterative/Constructive.** This type adds operations one at a time until all operations are scheduled. For example: ASAP, ALAP and force-directed.

2.2.1 ASAP and ALAP Scheduling

In ASAP scheduling, operations are assigned to the earliest possible control step as shown in Figure 2.6(a). In As Late As Possible (ALAP), operations are assigned to the latest possible control step (see Figure 2.6(b)).

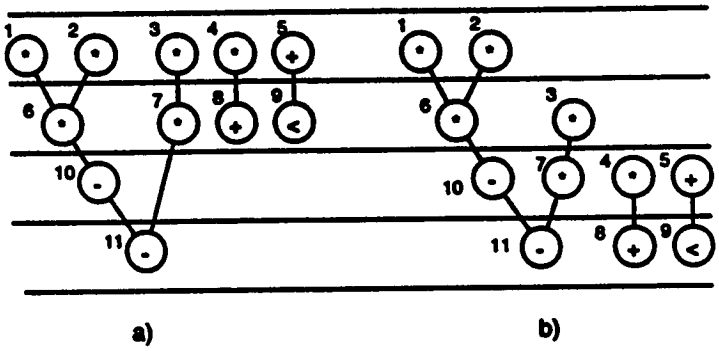

Figure 2.6: (a) ASAP scheduling. (b) ALAP scheduling.

2.2.2 Force-Directed Scheduling in HAL

In HAL synthesis system a scheduling technique called force-directed is used. In force-directed scheduling [23], a global time constraint is specified and the algorithm tries to minimize the hardware resources required to meet that time constraint. By

placing similar operations in different control steps, it is ensured that the functional unit has high utilization and the concurrency is balanced. The force-directed scheduling proceeds as follows:

1. Determine Time Frame:

   A time frame corresponds to an operation in the CDFG. The length of the time frame is calculated by evaluating the ASAP and the ALAP schedules as in Figure 2.6. The difference between the two schedules is equal to the length of the time frame (see Figure 2.7). The width of a time frame is equal to $1/length$. Hence the area of each time frame is always one. The time frame width indicates the probability of scheduling the corresponding operation in the control step(s) covered by the time frame. For example, in Figure 2.7 the ASAP schedule of the " $<$ " operation is in step 2 and its ALAP is in control step 4. So the time frame of the operation " $<$ " spans three control steps (control step 2 to control step 4) and its width is $\frac{1}{3}$. The time frame of operation " $<$ " has a length of 3 and a width of $\frac{1}{3}$, and hence, its area is equal to: $3 * \frac{1}{3} = 1$.

2. Create Distribution Graph (DG):

   Operation types are separated into disjoint sets. For each set the time frames are added to form a Distribution Graph. In this example there are four types of operations, namely: multiply, add, subtract and compare. They are separated

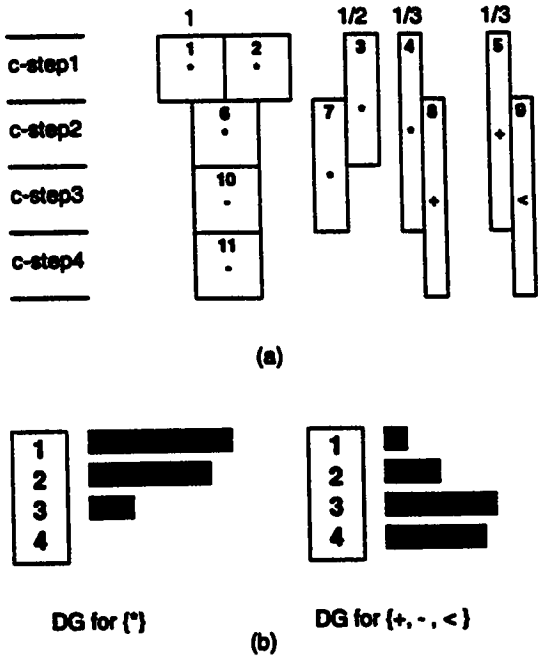

Figure 2.7: Time frames and distribution graph. (a) For multiply operation. (b) For add, subtract and compare operations.

into two disjoint sets: { *multiply* } and { *add, subtract, compare* }. For example the DG of the first set of operations { *multiply* } in control step 2 is equal to the summation of the time frames of *multiply* operations in step 2 and in this case they are operations number: 4, 6 and 7. Which makes the DG for step 2 $= 1 + \frac{1}{2} + \frac{1}{3} = 1.8$ (see Figure 2.7).

3. Calculate the force associated with each control step assignment for each operation as follows:

$F(op) = \sum_{j=1}^{N}(DG(j)) * x(op, j)$

j = *control step*

op = *operation*

$x(op,j)$: *change of probability of scheduling operation op in control step j*

For example, to schedule the multiply (operation 3) in step 1 then, the probability of this operation will change from $\frac{1}{2}$ to 1 in step 1 and from $\frac{1}{2}$ to 0 in step 2. This will result in a force equal to:

$$\begin{aligned} F(*_3) &= (DG(1) * x(3,1)) + (DG(2) * x(3,2)) \\ &= (2.833 * 0.5) + (2.333 * (-0.5)) \\ &= 0.25 \end{aligned}$$

The force is positive, because control step 1 is congested, hence scheduling operation 3 in control step 1 will have an adverse effect on the overall balance.

4. Choose the force with the largest negative value and schedule the corresponding set of operation(s) in the corresponding control step.

5. Update time frames and distribution graph as in Figure 2.8 and repeat from step (1) until all operations are scheduled.

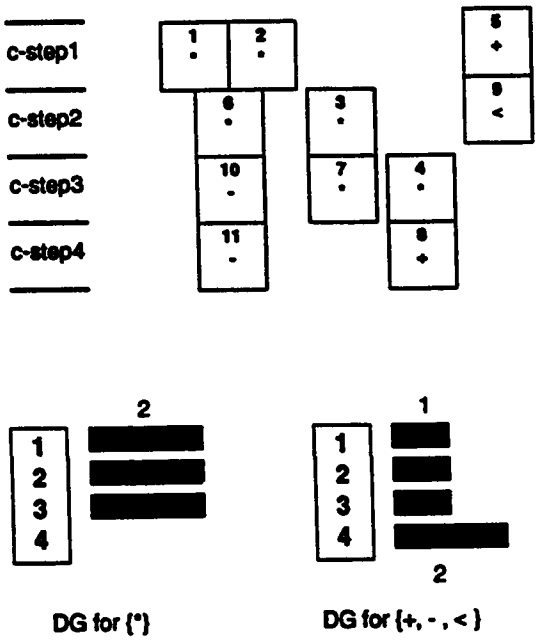

Figure 2.8: Final time frames and distribution graph.

Scheduling Conditional Branches

Conditional Constructs such as *if-then-else* statements cause forks in the CDFG as shown in Figure 2.9. Operations in different branches of a fork are mutually exclusive. When operations in different branches can be executed on the same type of FU, they can be scheduled into the same control-step without increasing the required number of FUs. Therefore, the same FU will be shared by those operations since they will never execute concurrently. To take advantage of this observation, the following is performed. For each control-step in which the time frames of the mutually exclusive operations intersect, only the highest probability of these is added to the corresponding DG. For example, for the "+" operation in Figure 2.9, which has a time frame spanning two control steps (steps 1 and 2), the distribution graph is made equal to 1 rather than $1+\frac{1}{2} = 1.5$. This is illustrated in Figure 2.9. Without special treatment of the mutually exclusive additions, the total distribution would be 1.5 in both control-steps. The unscheduled addition would then have an equal probability of being assigned to either control-step. It is obviously preferable to schedule it in the first control-step, since in this case only one adder will be required. This is due to the reduced distribution in the first control-step.

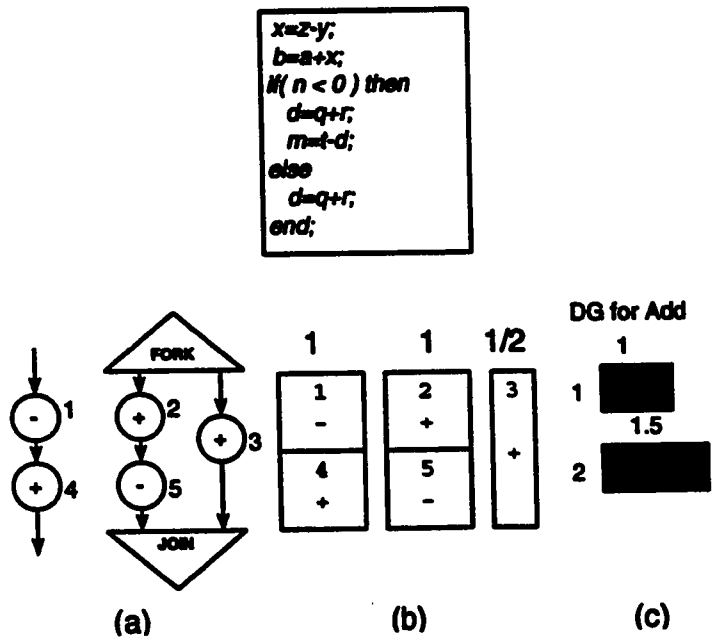

Figure 2.9: Behavioral description with an *if* statement and the corresponding: (a) CDFG. (b) Time frames. (c) DG.

2.2.3 Path-Based Scheduling

Path based scheduling [3] minimizes the number of control steps under given constraints. Conditional branches and conditional constructs are taken care of in path-based scheduling technique. Scheduling is applied on a CFG where nodes represent operations and edges represent precedence relationship. Edges have attributes (the condition of *if* and *while* statements), as illustrated in Figure 2.10.

```
entity prefetch is
 port(branchpc, ibus      : in bit 32;
      branch, ire         : in bit;
      ppc, popc, obus     : out bit 32);
end prefetch;
architecture behavior of prefetch is
begin
process
variable pc, olddpc : bit32 := 0;
begin
 ppc <= pc;                --1
 popc <= oldpc;            --2
 obus <= ibus + 4;         --3
 if (branch = '1')         --4
  then
    pc:= branchpc;         --5
 end if;                   --6
 wait until (ire = '1')    --7
 oldpc := pc;              --8
 pc := pc + 4;             --9,10
 end process;
 end behavior;
```

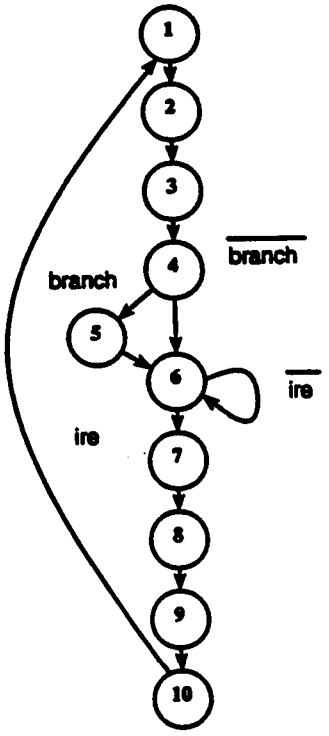

Figure 2.10: Behavioral description of an instruction fetch unit for a micro-processor, and the corresponding CFG.

The algorithm is applied in four steps:

1. The CFG of Figure 2.10 is transformed into a directed acyclic graph (DAG).

This is done by removing the feedback edges from the CFG.

2. All possible paths in the DAG are identified and scheduled independently in ASAP manner. A path represents a possible sequence of operations.

3. For each path all constraints are computed. Constraints are as follows:

   (a) Variables can be assigned only once in a control step.

   (b) I/O ports can be read or written only once in a control step.

   (c) FUs can be used only once in a control step.

   (d) The control step duration limits the number of operations that can be chained in that control step.

Each constraint can be interpreted as an interval that covers a set of operations, hence a constraint can be interpreted as a set of operations. For instance, the variable *pc* is written twice between operations 5 and 10, so constraint (a) indicates that path 1 has to be cut between operations 6 and 10, this is illustrated in Figure 2.11.

An interval graph is formed for each set of constraints. In the interval graph, each node represents an interval and an edge indicates that the two intervals overlap as in Figure 2.11. The number of cliques in a minimum clique cover corresponds to the minimum number of control steps. Each clique corresponds to a *cut*. This illustrates which operation will be executed in the corresponding

control step. Beside that, one cut (*cut*0) is added to the first operation in the CFG (see Figure 2.11). These cuts give the minimum number of control steps needed to execute that path.

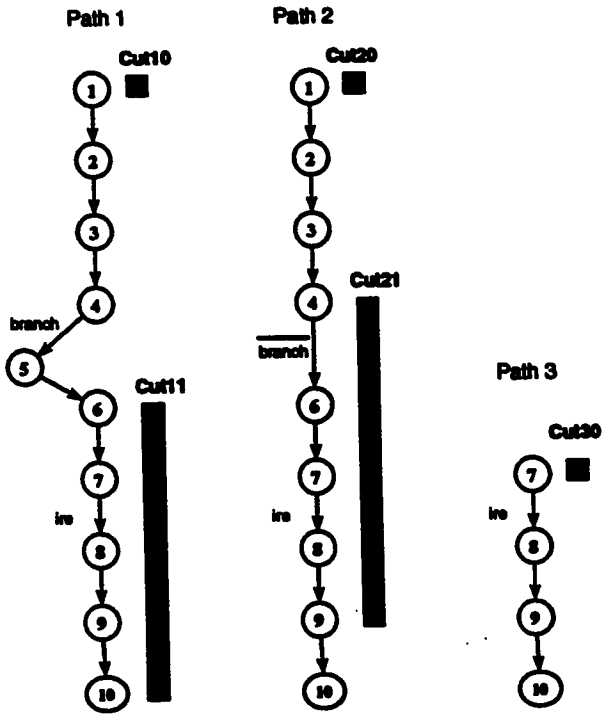

Figure 2.11: Constraints and interval graph.

4. Schedules of all paths, which are created in step 2, are overlapped so as to minimize the number of control steps. A graph is formed where the nodes correspond to the cuts generated in step 3 and the edges join nodes corresponding to overlapping cuts as in Figure 2.11. A minimum clique cover of this graph gives the least set of cuts and hence the minimum number of control steps (states) that satisfy the ASAP schedule for all paths.

5. The finite state machine (FSM) with the minimum number of control steps (states) that is calculated from step 4, is built (see Figure 2.12).

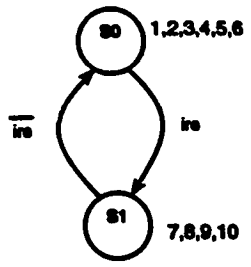

Figure 2.12: The control finite state machine.

2.2.4 Summary

A study of several intermediate forms and scheduling techniques has been conducted. ASAP and ALAP scheduling techniques can be easily applied to DFGs. As shown previously, force-directed scheduling can be applied on CDFG after obtaining the ASAP and the ALAP schedules. Hence, force-directed scheduling can also be applied to CDFGs. Force-directed Scheduling takes care of loops and conditional constructs. Path-based scheduling, operates on CFGs and takes care of loops and conditional constructs. The aforementioned results are summarized in Table 2.3.

	CDFG	DFG	CFG
ASAP and ALAP	Yes	Yes	Yes
Force-directed	Yes	Yes	No
Path-based	Yes	Yes	Yes

Table 2.3: Intermediate forms and scheduling techniques.

2.3 Intermediate Forms Used During Allocation

In this section various intermediate forms that are used during allocation are surveyed. Allocations can be defined as assigning operations, variables and communication paths to different hardware resources like: ALUs, registers, buses, multiplexers... etc., while meeting constraints like: total interconnections length, total hardware cost, critical path delays and design area [30]. In what follows, we describe how allocation is performed in various HLS systems together with the intermediate forms used for this task.

2.3.1 Allocation in FACET Synthesis System

FACET is a HLS system designed at CMU [31]. It uses a unified allocation procedure. FACET is not surveyed in the previous sections (transformation and scheduling), because the synthesis starts from a scheduled code sequence (straight code) which can be thought of as a scheduled DFG. The objectives are to minimize the number of FUs, storage elements, and interconnection units [31]. The synthesis process starts from a scheduled code sequence (straight code) as in Figure 2.13.

Step No.			
1	*V3 = V1 + V2;*	*V12 = V1;*	
2	*V5 = V3 - V4;*	*V2 = V3 * V6;*	*V13 = V3*
3	*V8 = V3 + V5;*	*V9 = V1 + V7;*	*V11 = V10 / V5*
4	*V14 = V11 and V8;*	*V15 = V12 OR V9;*	
5	*V1 = V14;*	*V2 = V15;*	

Figure 2.13: A code sequence.

Registers Allocation

The code sequence contains many variables. The purpose here is to combine as many variables as possible in the least possible number of registers. A variable is said to be *live* between the time of its first definition and last use. A variable is said to be *dead* between the time of its last use and its next definition. Table 2.4 shows the variables and their life-times (L=*Live* and D=*Dead*). If the *live* periods of two variables do not overlap then, they have disjoint life-time. Obviously two variables can be combined and hence stored in the same register if they have disjoint life-times. Relaxing this restriction, two variables are combinable if their life-times overlap in a step in which one of them is assigned to the other.

To combine variables in a minimum possible number of registers, a compatible graph is formed. In this compatible graph, nodes correspond to variables. Each two combinable registers are joined by an edge. To form a compatible graph, a complete graph is formed first, then edges that join two variables with overlapping life-times are deleted. The resulting graph is represented in Figure 2.14 where each

(1,9) (1,13) (1,14) (2,3) (2,5) (2,7)*
(2,8) (2,9) (2,11) (2,13) (2,15) (3,8)*
(3,13) (3,14) (3,15) (4,13) (5,8) (5,11)
(5,13) (5,14) (5,15) (6,13) (7,9) (7,13)
(7,14) (7,15) (8,13) (8,14) (9,13) (9,15)
(10,13) (11,13) (11,14) (12,13) (12,15) (13,14)
(13,15)

Figure 2.14: The edge list of the compatible variable graph.

pair represents two nodes (variables) connected by an edge. Edges that are tagged with "*" indicate pure data transfers like (*V1 = V14*).

Time	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10	v11	v12	v13	v14	v15
Etry	L	L	D	L	D	L	D	D	D	L	D	D	D	D	D
1	L	L	L	L	D	L	D	D	D	L	D	L	D	D	D
2	L	D	L	L	L	L	L	D	L	L	D	L	D	D	D
3	L	D	L	L	L	L	L	L	L	L	L	L	D	D	D
4	D	D	D	L	D	L	D	L	L	L	L	L	D	L	L
5	L	L	D	L	D	L	D	D	D	L	D	D	D	L	L
Exit	L	L	D	L	D	L	D	D	D	L	D	D	D	D	D

Table 2.4: Life-time table.

The compaction procedure works as follows. First, variables that are related by pure data transfers are combined. This is done by merging the nodes corresponding to these variables. This might lead to a reduction in the number of control steps. For example step 5 is cancelled because it contains two instructions which are pure data transfers (*V1 = V14, V2 = V15*). After that, the clique partitioning algorithm is applied on the graph of Figure 2.14. The algorithm partitions the set of vari-

ables into a minimum number of disjoint subsets. The resulting subsets (cliques) are {1,14}, {2,7,9,15}, {3,8,13}, {4}, {5,11}, {6}, {10} and {12}. The variables in each of these subsets can be assigned to one physical register, as follows:

Register 1 holds variables V1 and V14.

Register 2 holds variables V2, V7, V9 and V15.

Register 3 holds variables V3, V8 and V13.

Register 4 holds variable V4.

Register 5 holds variables V5 and V11.

Register 6 holds variable V6.

Register 7 holds variable V10.

Register 8 holds variable V12.

Therefore, only eight registers are needed to store these fifteen variables.

Functional Unit Allocation

To allocate the minimum number of FUs, similarities between the code sequence instructions are identified. For example, if two instructions have the same operation and the same operands but different destinations, then the FU and the connections from the source registers can be shared between these two instructions. Only a decoder has to be inserted at the FU output to select the desired destination. By looking at the input code sequence of Figure 2.13 it is clearly seen that the relation

between any two instructions has to be one of the following:

1. The operations and all three variables (destination and two sources) are different.

2. The operations are the same. All three pairs of variables are different.

3. One pair of the variables is the same. The operations and the other two pairs of variables are different.

4. The operations and one pair of variables are the same. The other two pairs of variables are different.

5. Two pairs of the variables are the same. The operations and one pair of variables are different.

6. The operations and two pairs of variables are the same. The third pair of variables is different.

7. The operations are different but the three pairs of variables are the same.

8. The operations and the three pairs of variables are all the same.

First a complete graph, whose nodes represent FUs, is constructed. Edges that connect simultaneously used FUs (nodes) are deleted to form the compatible graph. Then clique partitioning is applied on Figure 2.15. Hence, all operations are combined in three FUs as follows:

(1,2)	*(1, 3)*	*(1,4)*	*(1,5)*	*(1,6)*	*(1,7)*	*(1,8)*
		(2,4)	*(2,5)*	*(2,6)*	*(2,7)*	*(2,8)*
		(3,4)	*(3,5)*	*(3,6)*	*(3,7)*	*(3,8)*
					(4,7)	*(4,8)*
					(5,7)	*(5,8)*
					(6,7)	*(6,8)*
						(7,8)

Figure 2.15: The edge list of the FUs compatible graph.

1. FU1 implements " + " in step 1, " * " in step 2, " + " in step 3, and "*or*" in step 4.

2. FU2 implements " − " in step 2, " + " in step 3, and "*and*" in step 4.

3. FU3 implements "/" in step 3.

Interconnections Allocation

Interconnections that are never used simultaneously can be grouped into one bus. The interconnections problem consists of grouping all the interconnections into the minimum number of busses. This problem is very similar to the registers and the FUs allocation problem and it is solved in the same manner. A complete graph is formed where nodes represent interconnections between hardware resources (registers and FUs) (Table 2.5). Edges that connect simultaneously used interconnections are deleted. The clique partitioning algorithm is applied on the compatible graph of Figure 2.16 and the interconnections are combined into eight buses. All the 17

Source Name	Destination Name	Indexing Integer
V1	V12	1
V1	ALU1.In1	2
V2	ALU1.In2	3
V3	ALU1.In1	4
V3	ALU2.In1	5
V4	ALU2.In2	6
V5	ALU2.In2	7
V5	ALU3.In2	8
V6	ALU1.In2	9
V10	ALU3.In1	10
V12	ALU1.In1	11
ALU1.Out	V2	12
ALU1.Out	V3	13
ALU2.Out	V1	14
ALU2.Out	V3	15
ALU2.Out	V5	16
ALU3.Out	V10	17

Table 2.5: Indices of interconnections.

(1,2) (1,4) (1,5) (1,6) (1,7) (1,8)
(1,9) (1,10) (1,11) (1,12) (1,14) (1,15)
(1,16) (1,17) (2,4) (2,6) (2,9) (2,11)
(2,14) (2,16) (3,4) (3,6) (3,9) (3,16)
(4,5) (4,7) (4,8) (4,10) (4,11) (4,13)
(4,14) (4,15) (4,17) (5,13) (6,7) (6,8)
(6,10) (6,11) (6,13) (6,14) (6,15) (6,17)
(7,8) (7,9) (7,13) (7,16) (8,9) (8,11)
(8,13) (8,14) (8,16) (9,10) (9,11) (9,13)
(9,14) (9,15) (9,17) (10,11)(10,13)(10,14)
(10,16)(11,13)(11,15)(11,16)(11,17)(12,13)
(13,14)(13,15)(13,16)(13,17)(14,15)(14,16)
(14,17)(15,16)(16,17)

Figure 2.16: The edge list of the interconnections compatible graph.

interconnections of Table 2.5 are combined into 8 buses as follows:

Bus No. 1 replaces interconnections {13,14,15,16}.

Bus No. 2 replaces interconnections {1,2,4,11}.

Bus No. 3 replaces interconnections {6,7,8}.

Bus No. 4 replaces interconnections {3,9}.

Bus No. 5 replaces interconnections {5}.

Bus No. 6 replaces interconnections {10}.

Bus No. 7 replaces interconnections {12}.

Bus No. 8 replaces interconnections {17}.

2.3.2 Allocation in HARP

In HARP allocation is conducted in three phases: ALUs allocation, then registers allocation, and finally, interconnections allocation. straight

ALU Allocation

The High-Level Description in HARP is a basic block (straight code) written in FORTRAN 77 (see Figure 2.1). Scheduling and ALUs allocation are iterated in HARP and they are conducted as follows:

1. The FORTRAN code of Figure 2.1 is translated into a DFG as in Figure 2.2.

2. The DFG is scheduled in an ASAP way as shown Figure 2.17.

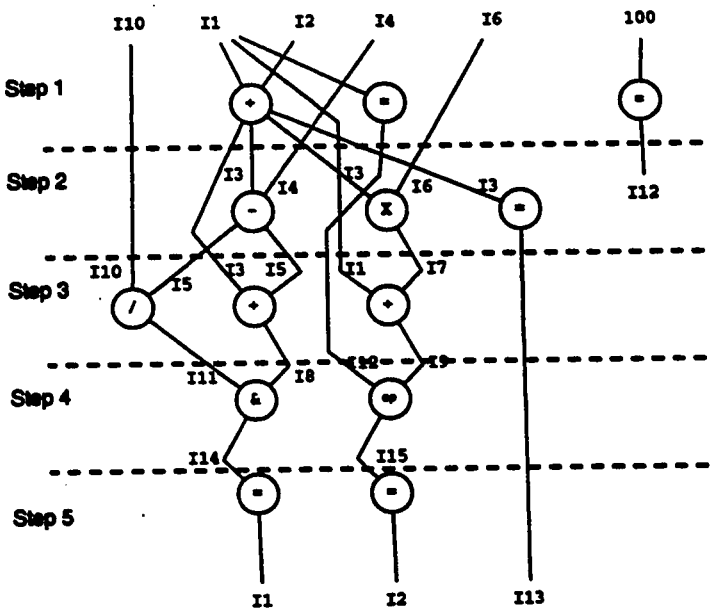

Figure 2.17: Initial schedule for the FORTRAN code of Figure 2.1.

3. Initial allocation is conducted by allocating one ALU for every operation.

4. A used/unused matrix is constructed in which each column represents an ALU and each row represents a control step. If a specific ALU is used in a control step then 1 is filled in the corresponding column/row location otherwise 0 is filled, (see Table 2.6). The used/unused matrix is constructed as follows:

$$U_{ns} = \begin{cases} 1, & \text{if ALU } n \text{ is used at step } s \\ 0, & \text{otherwise} \end{cases}$$

	A	L	U		Function Unit Definition
	1	0	0	1	1	ALU1:{+,-,AND}
s	1	1	0	1	0	ALU2:{*,/}
t	1	1	1	0	0	ALU3:{OR,+}
e	1	0	1	0	0	ALU4:{=}
p	0	0	0	1	1	ALU5:{=}

Table 2.6: Used/unused matrix.

5. From the used/unused matrix, a mutual correlation matrix is obtained as in Table 2.8. The mutual correlation matrix elements are computed as follows:

$$M(i,j) = \sum_{s=1}^{STEP} U_{is}.U_{js},$$

where STEP is equal to the number of control steps.

Group Number	Merge Table Operators
1	*,/
2	+,-,AND,or,
3	=

Table 2.7: Restriction database.

6. The mutual correlation matrix is searched for the minimum number and by referring to a restriction database (Figure 2.7). Mergeable ALUs can be detected and merged. For example, element (2,5) which indicates ALU2(*,/) and ALU5(=) is found to be the minimum. By referring to the restriction database, it is found that it is not possible to merge {*,/} and {=}. Next,

element (1,3) is found and ALU3 and ALU1 are mergeable, so they are merged to form ALU1 (see Table 2.8).

					Next function unit definition
4	2	2	2	1	
2	2	1	1	0	ALU1:{+,-,AND, OR}
2	1	2	0	0	ALU2:{*,/}
2	1	0	3	2	ALU4:{=}
1	0	0	2	2	ALU5:{=}

Table 2.8: Mutual correlation matrix.

7. After each ALU merge, the DFG is re-scheduled according to the new allocation. This leads to an increase in the number of control steps (see Figure 2.18).

8. Steps 4 through 7 are repeated until a predetermined maximum number of control steps is reached. In this case this predetermined number is 7 (see Figure 2.18).

Registers allocation

A life-time table is constructed. The columns of the life-time table correspond to variables and the rows correspond to control steps. In Table 2.9, "*d*" corresponds to a variable definition and "*r*" corresponds to a variable reference. For each variable (column), L (Live) is put in the interval [d,r], D (Dead) is put in the interval [r,d] and U (Unknown) otherwise (see Table 2.10). In Table 2.11 entries that contain

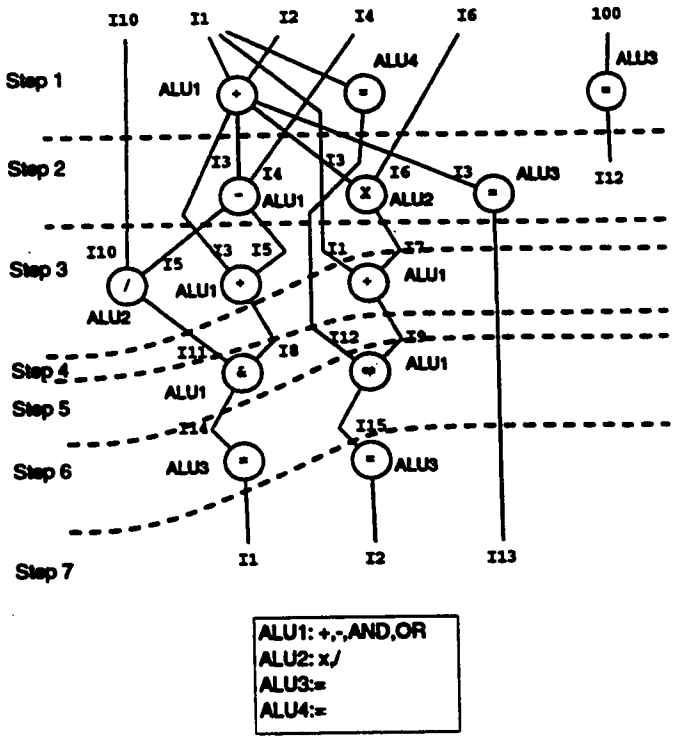

Figure 2.18: Final schedule and the allocated ALUs.

"U" are determined precisely and they are changed to either "*l*" for Live or "*d*" for Dead. Variables that do not have overlapping life-times can be allocated to the same register. Example of that is merging I13 with I12, I2 with I7, I5 with I15...etc. The heuristic that combines these registers is the left-edge algorithm which is widely used for channel routing.

Step	12	1	3	2	13	7	6	5	4	11	10	8	9	14	15
entry															
1	d	r	d	r											
2			r		d	d	r	d	r						
3			r					r		d	r	d			
4		r				r							d		
5										r		r		d	
6	r	d											r	r	d
7				d											r

Table 2.9: Definition and reference history.

Step	12	1	3	2	13	7	6	5	4	11	10	8	9	14	15
entry	D	L	D	L	D	D	L	D	L	D	L	D	D	D	D
1	L	L	L	L	D	D	L	D	L	D	L	D	D	D	D
2	L	L	L	D	U	L	L	L	L	D	L	D	D	D	D
3	L	L	L	D	U	L	U	L	U	L	L	L	D	D	D
4	L	L	U	D	U	L	U	U	U	L	U	L	L	D	D
5	L	D	U	D	U	U	U	U	U	L	U	L	L	L	D
6	L	U	U	D	U	U	U	U	U	U	U	U	L	L	L
7	U	U	U	U	U	U	U	U	U	U	U	U	U	U	L

Table 2.10: Preliminary life-time table.

Step	12	1	3	2	13	7	6	5	4	11	10	8	9	14	15
entry	D	L	D	L	D	D	L	D	L	D	L	D	D	D	D
1	L	L	L	L	D	D	L	D	L	D	L	D	D	D	D
2	L	L	L	D	d	L	L	l	L	D	L	D	D	D	D
3	L	L	L	D	d	L	D	l	l	L	L	L	D	D	D
4	L	L	d	D	d	L	l	d	l	L	l	L	L	D	D
5	L	D	d	D	d	d	l	d	l	L	l	L	L	L	D
6	L	l	d	D	d	d	l	d	l	d	l	d	L	L	L
7	d	l	d	l	d	d	l	d	l	d	l	d	d	d	L

Table 2.11: Final life-time table.

Interconnection Units Allocation

Resources with more than two outputs are merged into one data bus where possible. This is possible only if both the connections are not used simultaneously. After that multiplexers are inserted where needed.

2.3.3 Discussion

Many intermediate forms that are used in allocation have been discussed in this section. In FACET the allocation process starts from a scheduled straight code. A *life-time table* is built and a *conflict graph* is constructed. In HARP allocation also starts from a DFG and then a *used/unused matrix* is produced and then *mutual correlation matrix* is computed. A *life-time table* for compacting registers is obtained. It is concluded that flow graphs are ideal for the discussed allocation techniques. Moreover, it is observed that many additional intermediate forms are used to accomplish the allocation task. This is because allocation is usually conducted on all

hardware resources like: functional units (FUs), registers and interconnections.

2.4 Conclusion

In this chapter, several scheduling and allocation techniques and the intermediate forms used with them have been discussed.

Overall, the following statements can be made:

- Most intermediate forms are data flow based.
- Sequencing and concurrency are supported in all studied intermediate forms, since hardware systems are inherently concurrent.
- Hardware designs are concurrent in nature. In other words, operations in hardware systems are executed as concurrent as possible.
- Some intermediate forms can highlight some hidden features of the system behavior that even the high-level specification language cannot express. For example, concurrency is not highlighted in the high-level specification language while it is clearly shown in some intermediate forms like CDFGs.
- This chapter shows the inter-dependencies between the various high-level synthesis tasks which is the basis of the classification that will be introduced in the next chapter.

- Each scheduling and allocation technique can operate on a specific intermediate form.

Chapter 3

Classification of Intermediate Forms

In this chapter we briefly overview high-level synthesis systems and their corresponding intermediate forms. From these high-level synthesis systems, a classification framework which classifies intermediate forms into two main classes: *primary* and *secondary* is proposed. Primary intermediate forms (PIF) and secondary intermediate forms (SIF), objectives and constraints of each system are highlighted.

3.1 Constraints

In the Webster dictionary "constraint" is defined as "compulsion". When the term speed constraint is mentioned, it means that the design is synthesized in such a way

so as to maximize its speed. The way of increasing speed can be achieved by making the design more parallel. This takes place while scheduling or by allocating fast hardware components, which takes place during allocation. Moreover, the design speed can be increased by avoiding using multiplexers, which will be on the expense of increasing the number of registers and interconnections. Another example is cost. Reducing cost means reducing the design area, which can be achieved by reducing the number of the hardware components (ALUs, registers, buses and muliplexers). Satisfying one constraint will usually be on the expense of another, that is why compromise solutions are considered. Constraints are important and heavily influence the scheduling and allocation algorithms.

3.2 Classes of Intermediate Forms

The high-level synthesis process starts as follows. The high-level language is first transformed into an intermediate form (DFG, CDFG). This translation is necessary because the high-level language is not a suitable format. Therefore, it is transformed to a processible format (*intermediate form*) that can capture the specifications of the high-level language and lend itself to synthesis tasks like scheduling and allocation. The intermediate form that results from transforming the high-level language is referred to as *primary intermediate form (PIF)*. Synthesis tasks like scheduling and allocation extract a subset of the specification from the *primary intermediate form*

and build other intermediate forms referred to as *secondary intermediate forms.*

Therefore, intermediate forms can be classified according to the synthesis task they are used in. We classify intermediate forms into two main classes:

- Primary intermediate forms (PIF).

  These are the ones produced from *transformation*, since they inherit all the system behavior and specifications from the high-level description.

- Secondary intermediate forms (SIF).

  Those that extract a subset of the system specification from the PIF to perform a specific task in synthesis. These are usually used in *scheduling* and *allocation.*

High-level synthesis systems follow different approaches. Each high-level synthesis system synthesizes the input behavior under specific constraints to match specific objectives. In the following sections, some high-level synthesis systems are briefly surveyed. Primary intermediate forms (PIF), secondary intermediate forms (SIF), objectives and constraints of each system are highlighted when possible.

3.3 HAL Synthesis System

In HAL [22] the synthesis procedure starts with a high-level description. Some speed and cost constraints are imposed like: number of control steps and functional unit types. A scheduling technique called *force-directed* scheduling is used. This scheduling method reduces the number of functional units, registers and buses. It

places similar operations in different control steps so as to balance the concurrency of operations so that each hardware resource has high utilization. This in turn reduces the number of hardware resources required and hence reduces the area without increasing the number of control steps. The synthesis procedure starts as follows: first, a combined *control-data flow graph (CDFG)* is generated from the high-level description. Then, scheduling is iterated with allocation and the data path is generated (see Figure 3.1). This is achieved with the help of *secondary intermediate* forms like: Time Frames and a Distribution Graph.

In HAL, the intermediate forms used are:

- The primary intermediate form:

  Control Data Flow Graph (CDFG).

- Secondary intermediate forms:

  - Time Frames: this intermediate form shows the probability of scheduling each type of operation in each control step.
  - Distribution Graph: this intermediate form is generated for each type of operation. The force is calculated for each (operations set, control step) pair. An operation is assigned to the control step that has the minimum overall force, where the strength of the force is inversely proportional.

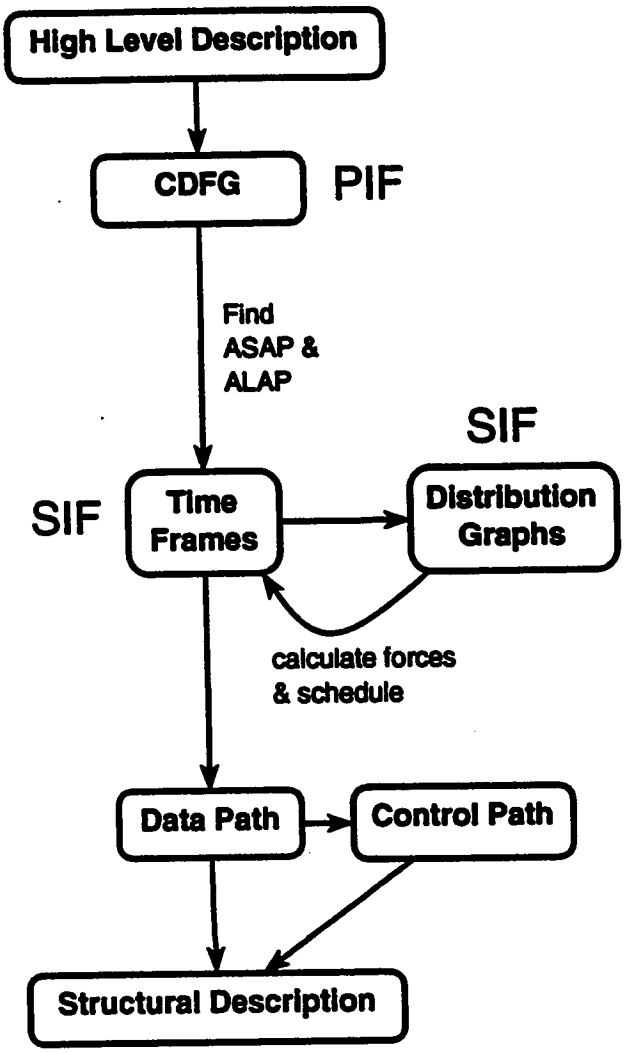

Figure 3.1: HAL high-level synthesis system.

3.4 FACET Synthesis System

FACET high-level synthesis system uses a unified procedure to accomplish the synthesis tasks [31]. Constraints are to minimize the number of FUs, storage elements, and interconnection units. The system synthesizes a *code sequence* (straight code). Then a *complete graph*, where nodes corresponding to registers, is established. Moreover a *life-time table* is established from the code sequence. By referring to the *life-time table*, edges that connect any pair of combinable registers are deleted. Then, the clique partitioning algorithm is applied on the remaining *compatible graph*. The resulting cliques represent the needed registers and the elements (nodes in each group) represent the variables that use the same register. The same procedure is applied to FUs where operations are numbered and a *conflict graph* is established and partitioned. FUs that are in the same partition are merged together. The same procedure is used in allocating interconnection units.

In this system intermediate forms can be categorized as follows:

- The primary intermediate form:

  Code sequence (straight code) in which each line represents a control step and operations that are in the same line are scheduled in the same control step.

- The secondary intermediate form:

  - Compatible/conflict Graph.

- Life-time table.

3.5 HARP Synthesis System

In HARP [27] the synthesis starts from a FORTRAN 77 plain code (see Figure 3.2). A *DFG* is generated. After that, the minimum number of single function ALUs are allocated. A *used/unused matrix* is generated and then a *mutual correlation matrix* is generated. Then with the help of a *restriction database* ALUs are merged. Merging ALUs leads to an increase in the number of control steps required. Therefore, the ALU merging process continues until a limit (a maximum number of control steps) is met [27]. Using this scheduled DFG, a *life-time table* for variables is built and then using the left edge heuristic, variables are merged to give the final number of needed registers. After that, interconnections buses are synthesized and multiplexers are inserted when needed. After generating the data path the control path is synthesized easily (see Figure 3.2). The intermediate forms used in HARP are:

- The primary intermediate form is:
  - DFG.
- The secondary intermediate forms are:
  - Used/unused matrix.
  - Mutual correlation matrix.

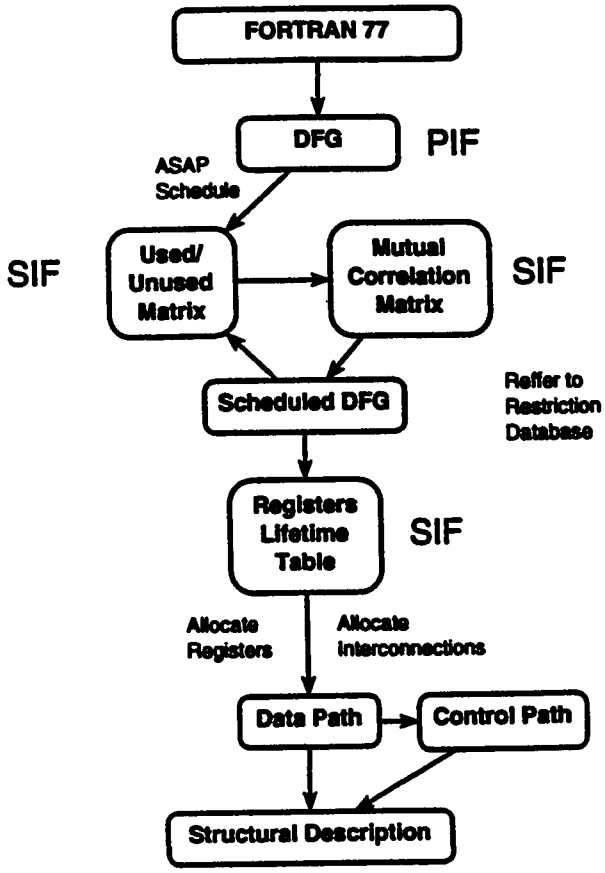

Figure 3.2: HARP high-level synthesis system.

- Restriction database.

- Registers life-time table

3.6 Path-Based Synthesis System [3]

In this system the synthesis starts from a *behavioral VHDL*. A CFG is built from the VHDL code. After that, the CFG is transformed into a *directed acyclic graph (DAG)*. This is done by removing the feedback edges from the CFG. All possible paths in the DAG are identified and scheduled independently in ASAP manner. A path represents a possible sequence of operations. For each path all constraints are computed. Each constraint can be interpreted as an interval that covers a set of operations. An *interval graph* is formed for each set of constraints. In the interval graph, each node represents an interval and an edge indicates that the two intervals overlap. The number of *cliques* in a minimum clique cover corresponds to the minimum number of control steps. Each clique corresponds to a *cut*. These cuts give the minimum number of control states needed to execute that path. From these cliques, a *finite state machine*, which indicates which operation will be executed in the corresponding control step, is constructed (see Figure 3.3). The intermediate forms used in the Path-Based system are:

- The primary intermediate form is:

  - CFG.

- The secondary intermediate forms are:
  - Directed acyclic graph (DAG).
  - Interval graph.
  - Finite state machine (FSM).

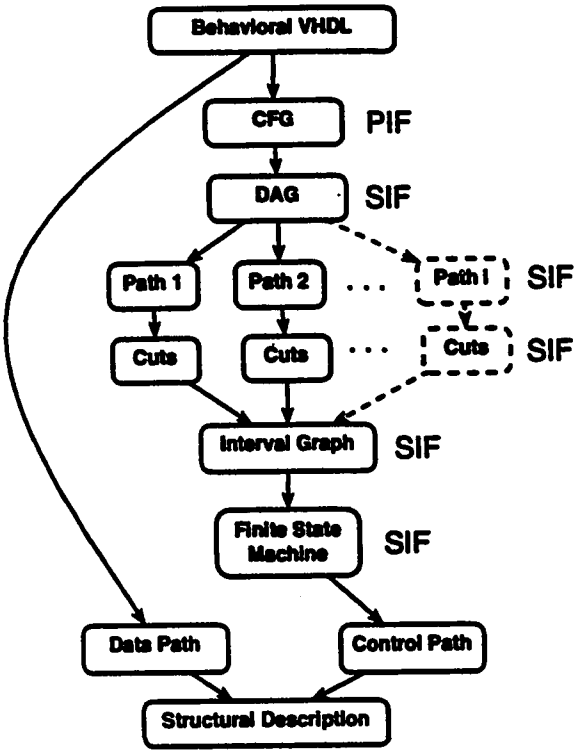

Figure 3.3: Path-based high-level synthesis system.

3.7 Conclusion

In this chapter a classification of intermediate forms has been presented. It classifies intermediate forms according to the synthesis tasks into two main categories namely: *primary* and *secondary*. In Table 3.1 *primary* and *secondary* intermediate forms for the presented systems are listed. Primary intermediate forms are complete. In other words, an equivalent high-level specification can be rebuilt from the PIF. Sometimes these PIFs help in highlighting information and details not explicit in the high-level description. For example, the DFG used in HARP synthesis system shows the maximum potential parallelism in the FORTRAN 77 code. This parallelism is not explicitly stated in the FORTRAN 77 code.

	BASIC	HAL	FACET	HARP
PIF	DFG + CFG	CDFG	Code Sequence	DFG
SIF	DPG CAG	Time Frames, Distribution Graph.	Compatible Graph, Conflict Graph, Life-time Table	Used/Unused Matrix Mutual Correlation Matrix, Life-time Table

Table 3.1: Primary and secondary intermediate forms.

From this chapter we conclude that an ideal PIF should posses the following features:

- Completeness. Which means that PIF should be able to capture all the information in the original specification.

- Extendibility. It should allow easy accommodation of additional user defined constraints.
- Should provide a suitable representation across all synthesis tasks.
- Independent of any input language.
- Should not be tied to a particular architectural style.
- Not restrictive. It should explore all possible concurrency so as not to restrict the search space.
- Should have simple syntax and can be easily manipulated.

Chapter 4

Generic Intermediate Form

Input behavioral specifications are typically compiled into flow graphs. The reason is mainly because graphs are powerful mathematical abstractions which allow the capture of all information in the original specification. Moreover, flow graphs are very powerful structures and have been extensively studied by scientists and engineers in various disciplines. As we have seen, most HLS tasks can easily be modeled as graph problems (e.g. clique partitioning).

New trends in high-level synthesis systems are pointing towards using a combined control and data flow graph (CDFG) as a primary intermediate form (PIF) [16] rather than using two separate flow graphs; one for data flow (DFG) and one for control flow (CFG) [15]. Using two separate graphs (DFG and CFG) results in the following:

- redundancy,

- complexity in handling and applying synthesis algorithms, and
- limit on the search space for some synthesis algorithms.

The type of representation used has an important impact on the final design [1].

In the following sections a new PIF is proposed. The proposed PIF is called *Generic CDFG* or *(GCDFG)*. It is called so to differentiate it from other CDFGs. First, the structure of this generic CDFG is defined, then a detailed explanation of the constructs it supports follows. Illustrative examples are given when necessary. A complexity analysis of this Generic CDFG is also presented. The complexity analysis covers the derivation or (compilation) complexity as well as the space complexity. After that an example is given to illustrate how the GCDFG is built. An example is also given to illustrate how this intermediate form is used in synthesis tasks like scheduling and allocation. Finally, a study that compares this proposed intermediate form with others is conducted.

4.1 Introduction and Definition

The GCDFG is an enhanced CDFG. It is a directed graph G(V,E), where nodes are connected by directed edges. There are two main types of nodes:

1. **Operation nodes.** These are the nodes that correspond to operations (logical and arithmetic) and they are labeled by their corresponding operation symbols.

2. **Control nodes.** A control node can be one of the following types:

   (a) S node. Start node that is used to indicate the beginning of a block.

   (b) E node. End node that is used to indicate the end of a block.

   (c) *IN/OUT* nodes. These types of nodes correspond to an Input/Output operations. For example, Read/Write operations can be implemented using IN/OUT nodes.

   (d) Fork node. It is used to implement *if-then-else* statements, and *For*, *While* and *Repeat* loops.

   (e) Join node. This type is used with the *fork* node to indicate the end of an *if-then-else* statement.

Edges in the GCDFG are of the following types:

1. **Variable Edges.** These edges correspond to variables.

2. **Control Edges.** These (dashed) edges are used for two purposes:

   (a) To connect control nodes to other nodes.

   (b) To connect two operation nodes which will force the source node operation to be executed before the destination node operation.

while **condition_body** *do*
loop_body
end while

(a)

*if (***condition_body** *) then*
if_body
else
else_body
endif;

(b)

Figure 4.1: Blocks in: (a) *while* loop (b)*if-then-else* statement.

4.2 Generic Control-Data Flow Graph (GCDFG) Transformation

To verify the power of this Generic CDFG, some examples from previous chapters are expressed using this intermediate form. The Generic CDFG supports arithmetic and logical operations, hierarchy, concurrency, control transfer like looping, conditional constructs and special data structures such as arrays. The input behavioral specification consists of blocks connected by control construct. Blocks can be loop bodies, loop conditions, procedures, *if-then-else* statements bodies and conditions (see Figure 4.1). Control constructs can be loops *(for, while, repeat-until)*, conditional constructs *(if-then-else)*, procedure calls.

4.2.1 Transforming Blocks

As mentioned earlier, the GCDFG consists of nodes and directed edges. Nodes are labeled and they represent operations or input/output relations. The directed edges

which represent variables are labeled by the variable name. If the node represent an operation then, its label is the mathematical symbol of the operation it represents such as $=$, $+$, $-$, $*$, $/$, *and*, *or* ... etc. Operands can be variables or constants. Constants are represented by an edge labeled by the constant value generated from a constant generator node *C.Gen.* For example in Figure 4.2, $c = a + b$, a and b are incoming edges to the node $+$, and c is an outgoing edge.

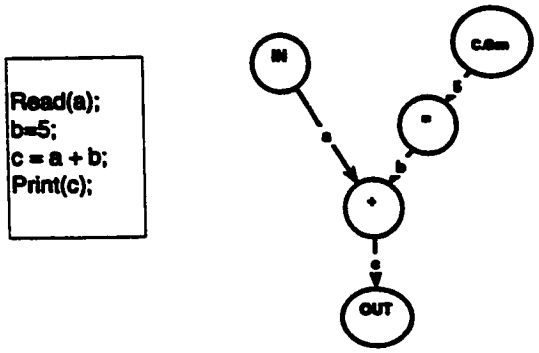

Figure 4.2: The GCDFG of c=a+b.

In binary operations like "$-$" and "/", a convention is followed, the edge coming from the left is the first operand and the right edge is for the second operand. This GCDFG shows all the potential parallelism of a behavioral description. For example, the basic block in Figure 4.3 is transformed into the GCDFG of Figure 4.4. This GCDFG shows the maximum potential parallelism in this basic block which can be rewritten as in Figure 4.5. All nodes and edges are labeled by a unique identification number. These numbers will be used to build node-list and edge-list as we will see later.

```
Read (I1)
Read (I2)
I3=I1+I2;
Read( I4)
Read( I6)
I5=I3-I4;
I7=I3*I6;
Read(I10);
I13=I3;
Write(I13)
I8=I3+I5;
I9=I1+I7;
I11=I10/I5;
I12=I1;
I14= (I11 AND I8);
I15= (I12 OR I9);
I1=I14;
I2=I15;
Write( I1, I2)
END
```

Figure 4.3: FORTRAN code.

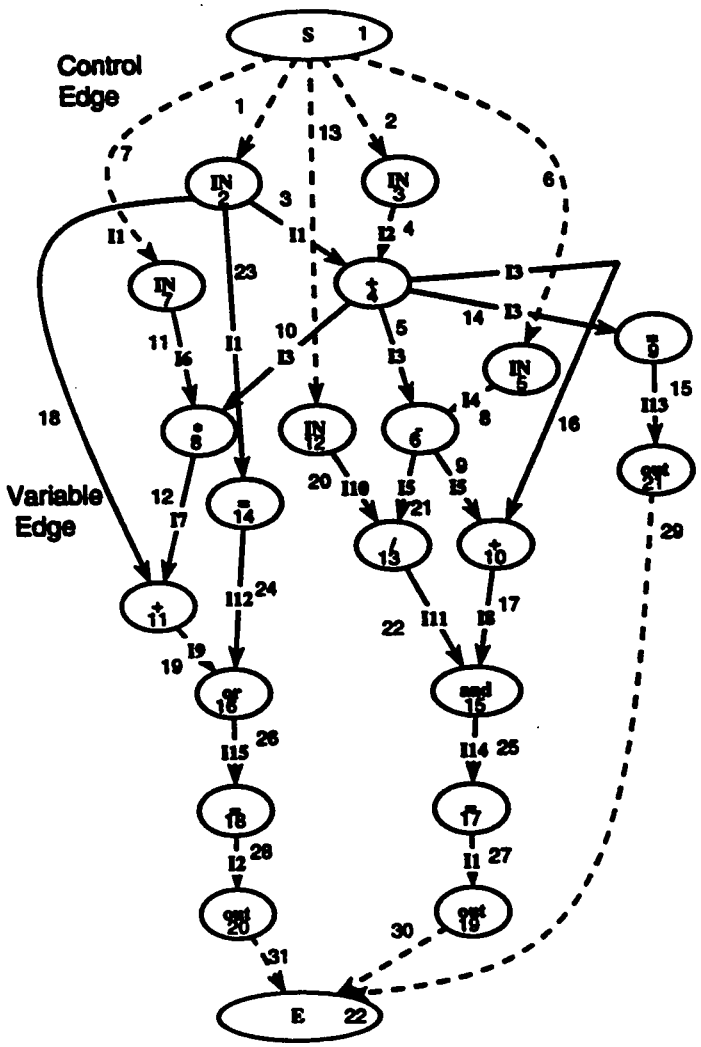

Figure 4.4: A GCDFG of the basic block in Figure 4.3.

```
Read(I1);          Read(I2);          Read(I4);          Read(I6);
I3=I1+I2;          I12=I1;
I5=I3-I4;          I7=I3*I6;          I13=I3;
I8=I3+I5;          I9=I1+I7;          I11=I10/I5;
I14=IAND(I11,I8);  I15=IRO(I12,I9);
I1=I14;            I2=I15;
Write(I13);        Write(I1);         Write(I2);
STOP
END
```

Figure 4.5: Parallel FORTRAN code as per the GCDFG in Figure 4.4.

When transforming blocks, a *start* node (S) and an *end* node (E) are added at the beginning and at the end of each block. These two nodes are added to indicate the beginning and the end of each block. Moreover, the S node is connected to all independent operations (have no predecessor) in the block, like *Read(I1)* and *Read(I2)* in Figure 4.3. The E node shows all operation that have no successor operation in a block, like *Write(I1)* and *Write(I2)* in Figure 4.3. Adding S and E nodes helps in scheduling and optimization as will be seen later.

4.2.2 Transforming Control Constructs

The GCDFG consists of blocks and control nodes connected by control edges. Control edges and control nodes are labeled. To differentiate control edges from variable edges, control edges are made dashed. Control edges specify operations execution order. A control edge connects two operation nodes, or an operation node to a

control node. For example, if node A is connected to node B by a dashed (control) edge, then operation A is executed before operation B even if they are not data dependent. So control edges force execution order. Solid edges, which represent variables, implicitly indicate data dependencies which in turn specifies execution order as well.

Transforming Loops

For loops, *while* loops and *repeat-until* loops have a condition to be satisfied. Based on this condition, the loop body is either repeated or halted. In our GCDFG notation, loops are specified with the help of *fork* nodes. A *fork* node has two incoming edges (condition and feedback) and two outgoing edges; the edge that starts the loop is labeled as *true* and the edge that exits the loop is labeled as *false*. The *fork* node receives the condition value (true or false) from the incoming condition edge, and based on that, a branch is selected (loop body branch (true) or the exit branch (false)). An example of a *for* loop transformation into GCDFG is shown in Figure 4.6. An example of a *while* loop that is transformed into GCDFG is shown in Figure 4.7. In the same way, the *repeat-until* loop construct is illustrated in Figure 4.8.

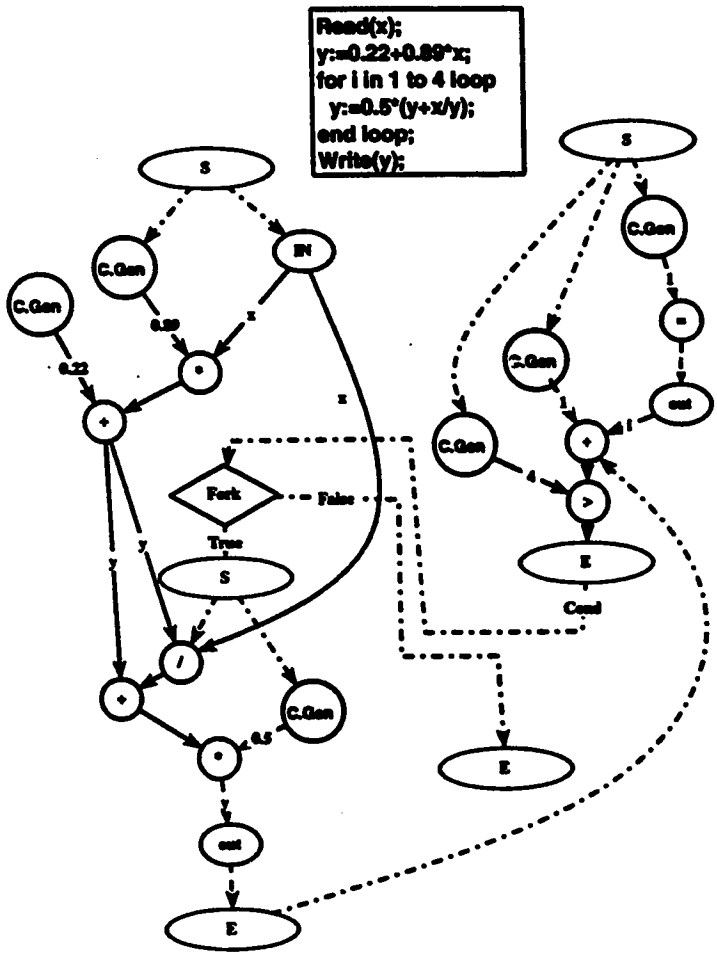

Figure 4.6: The GCDFG of a *for* loop.

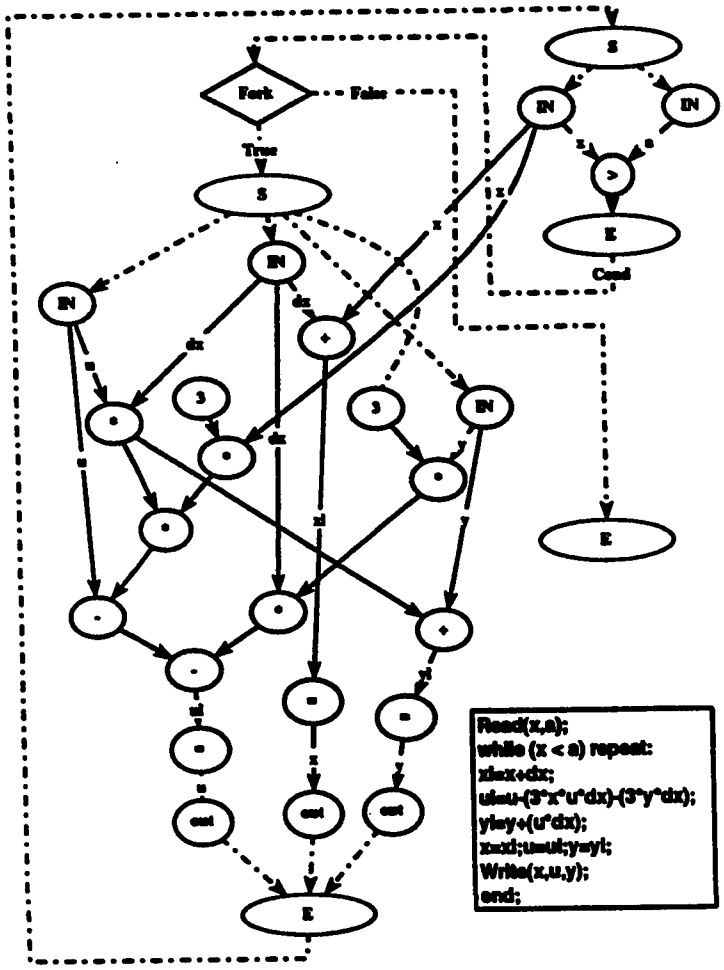

Figure 4.7: The GCDFG of a *while* loop.

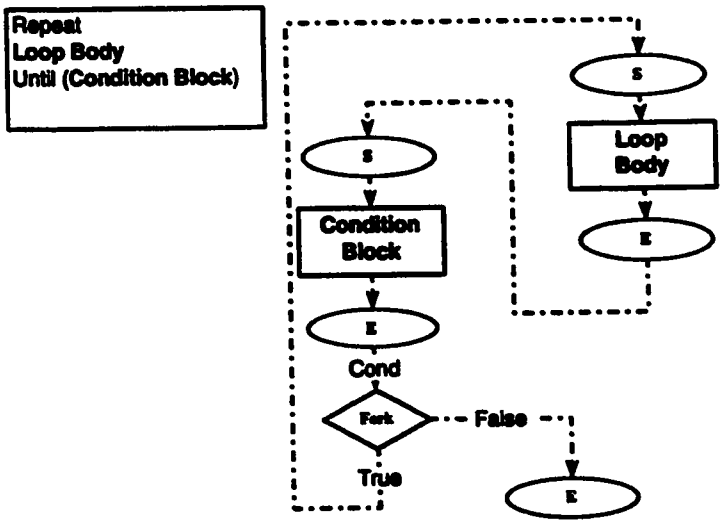

Figure 4.8: The GCDFG of a general *repeat-until* loop.

Conditional Constructs

Conditional constructs representations can be classified into two main categories [24].

- **Control select (C-select).** In this representation condition select is performed before executing any conditional operation from any branch (see Figure 4.9). So the decision as to which operations share resources is postponed until scheduling or allocation is done.

- **Data select (D-select).** In this representation all conditional branches are executed separately in parallel and correct data values are selected at the end of the conditional branch, which might generate faster design (see Figure 4.9). However, since mutual exclusive operations (operation in different branches)

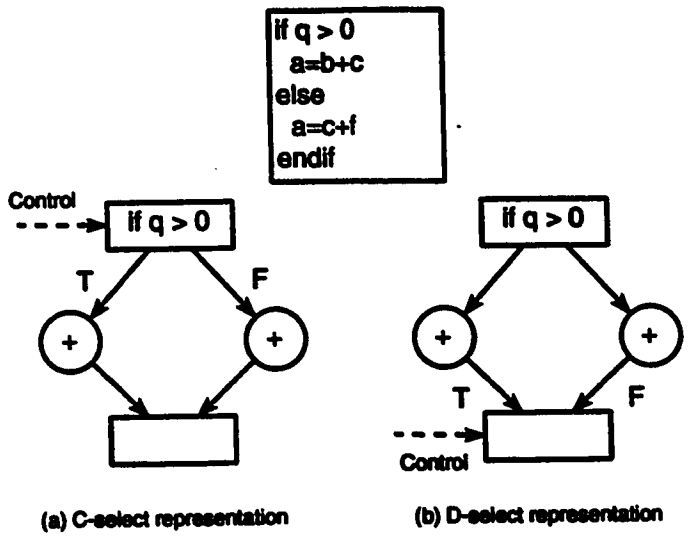

Figure 4.9: Conditional representation.

here are executed together, more ALUs might be allocated which might result in an expensive design.

In GCDFG C-select representation is used. In GCDFG, the *if-else* construct consists of three blocks: the *condition block*, the *if block* and the *else block*. The condition block evaluates the condition and passes the result (*true or false*) via the *condition edge* to the *fork* node which in turn selects the corresponding branch (true or false) (see Figure 4.10). The *fork* node has one incoming control edge and two outgoing edges. The incoming edge carries the condition value. Based on the condition value, one of the branches is executed.

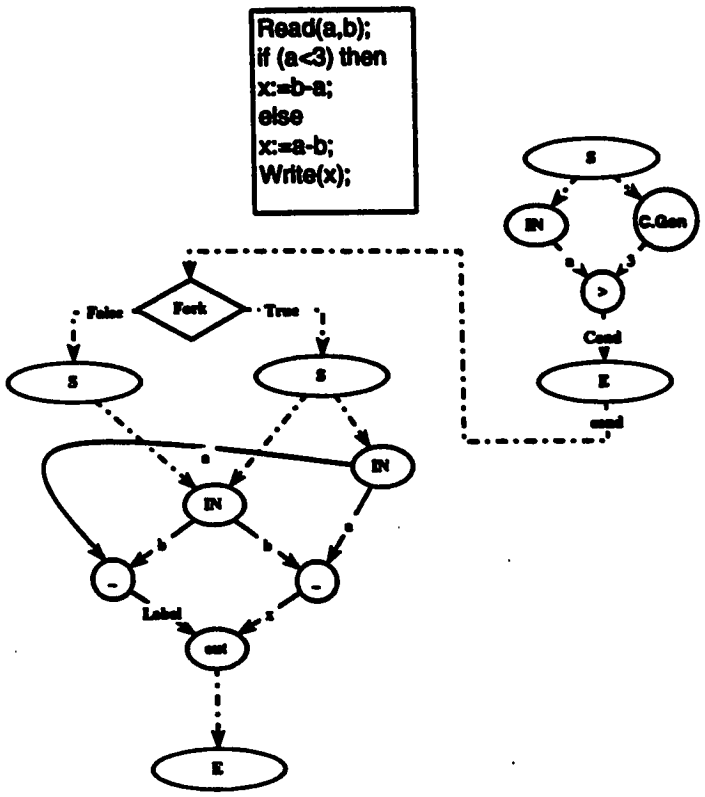

Figure 4.10: A GCDFG of an *if-then-else* statement.

4.2.3 Hierarchy

Procedures and subroutines in the high-level language are transformed first and then are handled as blocks. S and E nodes are added at the beginning and end of each procedure/subroutine. A *procedure call* statement is implemented by establishing a control edge labeled *call* from the main program to the S node of the procedure and an outgoing edge labeled *return* from the procedure E node to the calling program (see Figure 4.11).

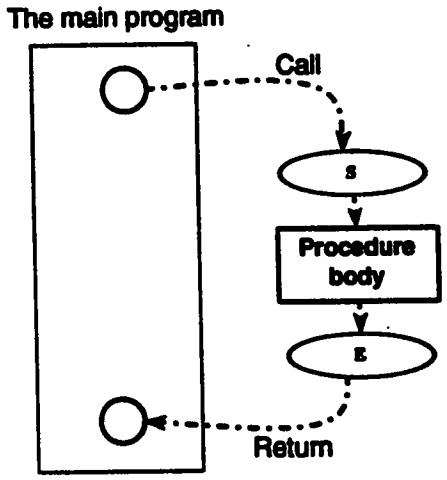

Figure 4.11: A procedure call.

4.2.4 Arrays

To transform arrays into GCDFG, In-Array *(I.Array)* and Out-Array *(O.Array)* nodes are used. When an array cell is read or written, we need to know the array name and the cell index. If the array is a destination operand (a value is written

to it) like $A[3] = 5 + 6$, then the *I.array* node takes two edges as inputs, the array index which is 3 and the array name which is A (see Figure 4.12). If the array is a source operand (a value is read from it) like $Q = A[5] + 6$ then the *O.Array* node takes the array index which is 5 in this case as an input edge and output one edge as the array cell (see Figure 4.12).

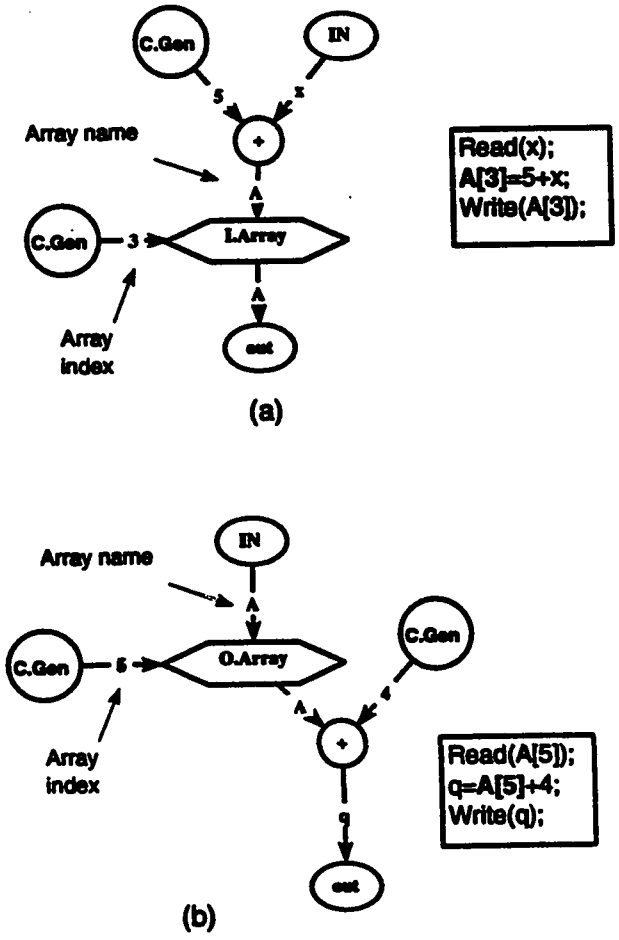

Figure 4.12: (a) The GCDFG for write to array operation. (b) The GCDFG for read from array operation.

4.3 Complexity Analysis

In this section, derivation and space complexity are analyzed for the GCDFG. To build a GCDFG from a high-level language (behavioral description), the parsing algorithm in Figures 4.13, 4.14 and 4.15 is applied. This algorithm parses the behavioral description and builds two lists; one for nodes and one for edges.

The parsing algorithm reads the input behavioral specification and works as follows.

- When a beginning of a block is encountered, a S node is added to the edge-list. After that, instructions within the block are transformed as follows: operations are added to the node-list, variables instances are added to the edge-list. This process continues until an end of a block is encountered; if so, then an E node is added to the node-list.

- When a control constructs like: *if-then-else*, *for*, *while* and *repeat until loops* or *procedures calls*, is encountered then:

  - If an *if-then-else* statement is encountered, we add a *fork* node and three control edges. These are: one incoming edge labeled as *'Cond'* to connects the *if condition* block and two outgoing edges; one to the *true* branch and one to the *false* branch. To indicate the end of the *if* statement, a *join* node and two edges one from each branch, are added (see Figure 4.10).

- If *for, while or repeat-until* loops are encountered, then a *fork* node and three edges are added at the beginning of the loop (in case of *repeat-until* it is added at the end): an incoming edge to the *fork* node comes from the loop condition block which is labeled as *'Cond'*. Two outgoing edges from the *fork* node one to the loop body labeled as *True* and one to loop exit labeled as *False*.

- If a *procedure* is encountered then, as in blocks, a S node is added at the beginning of the procedure then the procedure body is transformed. Finally, an E node is added at procedure end.

4.3.1 Space Complexity

As stated earlier, the aforementioned algorithm parses the behavioral description to build the GCDFG which consists of two lists; *node-list* and *edge-list*. The node-list has the following fields: *Node ID, Label, Type (operation or control)*. Each edge connects two nodes (a source node and a destination node), so the edge-list has the following fields: *Edge ID, Label, Type, Source Node ID, Destination Node ID*. Nodes and edges are given unique identification numbers. Each block with m operations creates $m + 2$ nodes in the GCDFG because a S and an E nodes are added to the node-list. The same applies for *procedures*. Each *if-else* statement in the high-level specification creates a *fork* node and a *join* node in the GCDFG. Each *loop* in the high-level specification creates a *fork* node in the GCDFG. Assuming that the

behavioral description has n operations, b blocks, f if-else conditions, l loops and p procedures, then the corresponding GCDFG has $(n+2b+2f+l+2p)$ nodes in the node-list.

Since the number of *procedures, if-then-else statements, for loops, while loops* and *repeat until loops* is negligible with respect to n (the number of operation), then the space complexity of the node-list is $O(n)$.

The number of GCDFG edges is not straightforward to calculate, since one edge is created each time a variable is used. Hence, the number of edges in a block is equal to the number of data transfers (variable instances). Besides, some control edges are added with some control nodes. When a S node is added to indicate the beginning of a block then, some edges are added between this S node and some operation nodes. The same applies for the E node. In the worst case, all the n operations of a block are connected to the S node and to the E node which creates $2n$ control edges. Moreover, a *feed-back* edge, a *condition*, *true* and *false* edges are added when a loop is encountered. Two branching edges (true branch and false branch), and two *joining* edges, are added whenever an *if-then-else* statement is implemented. A *call* edge and a *return* edge are added for each procedure call.

Assuming a behavioral description with e data transfers, n operations, f if-then-else conditions, l loops and c procedure calls in a behavior description, then the corresponding GCDFG contains $(e+2n+4f+4l+2c)$ entries in the node-list. Since the number of control constructs like: *if-then-else* statements, *for loops, while*

loops ... etc., is negligible with respect to e and n, then the number of edges in the GCDFG is $O(n+e)$. So the space complexity of the edge-list is also $O(n+e)$.

4.3.2 Derivation Complexity

The derivation complexity of the parsing algorithm is mainly the complexity of building the node-list and the edge-list. To build an edge-list of size $n+e$, at least $n+e$ steps are required. Each time an edge is added to the edge-list, the node-list is searched twice, once to get the *source node ID* and once to get the *destination node ID*, and since it takes n steps to search the node-list, the complexity of building edge-lists is equal to $O(n*(n+e)) = O(n^2+n*e)$.

4.4 GCD Example

In this section, a behavioral VHDL description is transformed into a GCDFG. The GCDFG is presented graphically. The node-list and the edge-list are constructed. Figure 4.16 shows an example of a behavioral specification. It contains a VHDL program fragment, demonstrating assignment, arithmetic operations, a *while* loop and an *if-then-else* statement. The node and edge-lists are shown respectively in Tables 4.1, 4.2, 4.3 and 4.4.

The parsing algorithm works as follows:

- A statement is read form the behavioral description. It is identified as a *while*

```
Procedure Process_Block
Begin
Add S node
Inner:=Inner+1;
    While not ((Control construct encountered) or (End of Block)) do
    Begin
        Add_Operations_to_Nodes_List;
        Add_Variables_to_Edges_List;
    End{ While}
if (End of Block) then
Begin
add an E node;
Inner:=Inner-1;
End;
else
Return;
End {Procedure}

Procedure Process_If_Statement
Begin
    Process_Block; {Transforming the condition}
        add fork node to the Node-List;
        add true edge to Edge-List;
    Process_Block;
if ('else' encountered) then
begin
    add false edge to Edge-List;
    Process_Block;
end;
add joint node and join edges to Nodes and Edge-Lists;
End;
```

Figure 4.13: This algorithm parses the input and creates the node-list and edge-list, (continued. next page).

```
Procedure Process_For_Loop
Begin
    Process_Block; {Transforming the condition}
        add fork node to the Node-List;
        add loop body edge to Edge-List;
    Process_Block;
add exit edge;
End;

Procedure Process_While_Loop
Begin
    Process_Block; {Transforming the condition}
        add fork node to the Node-List;
        add loop body edge to Edge-List;
    Process_Block;
add exit edge;
End;

Procedure Process_Repeat_Loop
Begin
        add fork node to the Node-List;
        add loop body edge to Edge-List;
    Process_Block;
    Process_Block; {Transforming the condition}
add exit edge;
End;
```

Figure 4.14: The parsing algorithm (continued from Figure 4.13).

```
Procedure Process_Procedures_Calls
Begin
        add 2 control edges to Edge-List;
        (one or calling and one for returning);
End;

Main
    While not (EOF(behavioral input)) do
    begin
    Read a word from the input behavioral description;
    if (a beginning of a block encountered) then
    Process_Block;
    else {(beginning of a control construct encountered)}
        Case the control construct of:
    'if' then
        Process_If_Statement_Body;
    'else' then
        Process_Else_Body;
    'while' then
        begin
        Process_While_Loop_Condition;
        Process_While_Loop_Body;
        end;
    'for' then
        Process_For_Loop;
        'repeat' then
        Process_Repeat_Loop;
        'procedure' then
        Process_Procedure;
    end;{While}
End.
```

Figure 4.15: The parsing algorithm, (continued from Figure 4.14).

```
entity EXAMPLE is
                        port (IN1, IN2, IN3 : in integer);
                        com : in bit;
end gcd;

architecture BEHAVIOR of EXAMPLE is
begin
                        process
                              variable A, B, C, D, E, F : integer;
                        while (E>F)
                        loop
                              if ( com = '1' ) then
                                    E:= IN1;
                                    F:= IN1;
                                    A:= D + IN2;
                                    B:= C - IN3;
                                    C:= A + B;
                                    D:= C + 3;
                               else
                                    E:= IN2;
                                    F:= IN3;
                               end if;
                         end loop;
```

Figure 4.16: A behavioral specification fragment in VHDL.

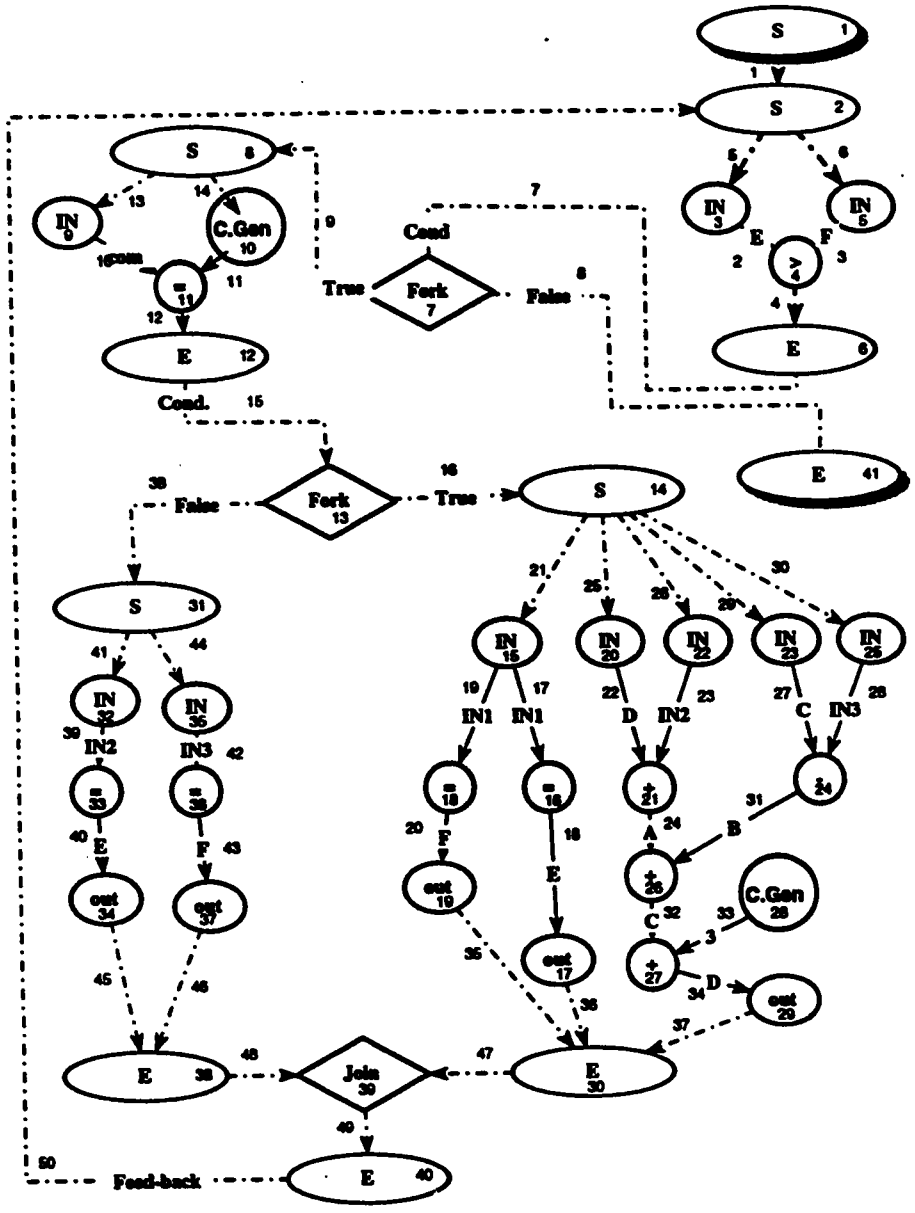

Figure 4.17: The GCDFG of the VHDL behavioral specification of Figure 4.16.

statement. A S node is added to the edge-list to indicate the beginning of *while* condition block. The while condition ($E < F$) is realized as follows. The values of E and F are read by two IN nodes. Then they are are compared by the operation "<". This results in adding three nodes to the node-list (IN, IN and "<") and two edges to the edge-list (an edge that represent F and an edge that represent E).

- An E node is added to the node-list to indicate the end of the *while condition block.*

- A *fork* node is added to the node-list to branch to the *while* loop body (true branch) or exit (false branch), and an edge labeled with *Cond* is added to connect the E node from the condition block to the *fork* node.

- A S node is added to indicate the beginning of the while loop body. An edge that is labeled *true* connects the *fork* node to the S node is added to the edge-list. An E node and an exit edge that is labeled *false* is added to the node-list and edge-list.

- An *if* statement is encountered. A S node is added to indicate the beginning of the *if* block.

- The condition of the *if* statement is transformed in the same way as the *while* condition.

- A *fork* node, that implements the *if-then-else* statement, is added to the node-list.
- A S node that indicates the beginning of the *true* branch is added. An edge that is labeled *true* is added to the edge-list to connect the *fork* node to the S node. The same is done to the *false* branch.
- The *if* statement body (true branch) is transformed. Then, the *else* body (false branch) is transformed.
- A *join* node is added to indicate the end of the *if-then-else* statement. Two edges that connect the end of the true block and the end of the false block to the *join* node are added to the edge-list.
- Finally, an E node that feeds back to the *fork* node is added to the node-list. A feed back edge, that connects the E node to the *fork* node to restart the *while loop*, is added to the edge-list.

4.5 Optimization

The GCDFG as a PIF facilitates high-level optimization. For example, *useless code* can be detected and eliminated easily. This can be achieved as follows. While transforming the high-level description into GCDFG, the GCDFG is checked for *dangling edges*. *Dangling edges* correspond to variables calculated or assigned values

Node ID	Node Label	Node Type
1	S	C
2	S	C
3	IN	O
4	>	O
5	IN	O
6	E	C
7	FORK	C
8	S	C
9	IN	O
10	C.Gen	O
11	=	O
12	E	C
13	FORK	C
14	S	C
15	IN	O
16	=	O
17	OUT	O
18	=	O
19	OUT	O
20	IN	O
21	+	O
22	IN	O
23	IN	O
24	-	O
25	IN	O

Table 4.1: The node-list.

Node ID	Node Label	Node Type
26	+	O
27	+	O
28	C.Gen.	O
29	OUT	C
30	E	C
31	S	C
32	IN	O
33	=	O
34	OUT	O
35	IN	O
36	=	O
37	OUT	O
38	E	C
39	JOIN	C
40	E	C
41	E	C

Table 4.2: Continued..., the node-list

but never used. These *dangling edges* and their successor nodes that have no path to the last E node in the block can be eliminated. This is done by scanning the edge-list for edges that have a source node but no destination node (see Figures 4.18 and 4.19).

4.6 Scheduling and Allocation Example

In this section an example that illustrates how can the GCDFG be used in scheduling and allocation is shown. First, the behavioral description of Figure 4.3 is transformed into a GCDFG of Figure 4.4. After that, an ASAP scheduling is applied to the

ID	Label	Type	Source	Destination
1	S	C	1	2
2	E	V	3	4
3	F	V	5	4
4	-	C	4	6
5	-	C	2	3
6	-	C	2	5
7	Cond	C	6	7
8	False	C	7	41
9	True	C	7	8
10	com	V	9	11
11	1	V	10	11
12	–	C	11	12
13	–	C	8	9
14	–	C	8	10
15	Cond	C	12	13
16	True	C	13	14
17	IN1	V	15	16
18	E	V	16	17
19	IN1	V	15	18
20	F	V	18	19
21	–	C	14	15
22	D	V	20	21
23	IN2	V	22	21
24	A	V	21	26
25	–	C	14	20

Table 4.3: The edge-list, continued in the next page.

ID	Label	Type	Source	Destination
26	–	C	14	22
27	C	V	23	24
28	IN3	V	25	24
29	–	C	14	23
30	D	V	14	25
31	B	V	24	26
32	C	V	26	27
33	3	V	28	27
34	D	V	27	29
35	–	C	19	30
36	–	C	17	30
37	–	C	29	30
38	False	C	13	31
39	IN2	V	32	33
40	E	V	33	34
41	–	C	31	32
42	IN3	V	35	36
43	F	V	36	37
44	–	C	31	35
45	–	C	34	38
46	–	C	37	38
47	–	C	30	39
48	–	C	38	39
49	–	C	39	40
50	Feedback	C	40	2

Table 4.4: Continued ..... the edge-list.

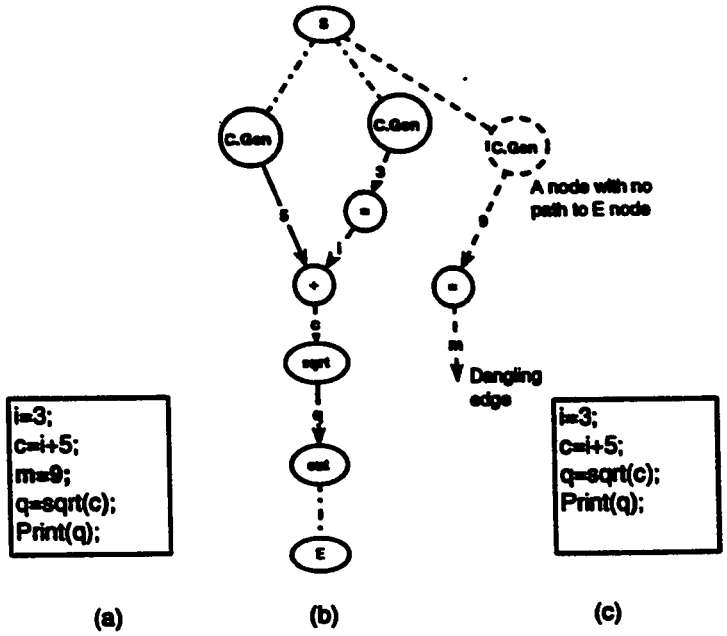

Figure 4.18: High-level optimization: (a) High-level code with useless statement. (b) The corresponding GCDFG and the dangling edge. (c) The high-level code after eliminating useless statements.

Search the Edges List;
If an edge is found with no destination node ID then
Delete this edge from the Edges List;
While (Not reached to a S node) Do
begin
Delete the Source node from the Nodes List
Delete all edges that
have this node ID in their source
end;

Figure 4.19: Useless code detection and elimination algorithm.

GCDFG. Finally, registers allocation is performed using a life-time table extracted from the GCDFG.

4.6.1 Transformation

The behavioral description of Figure 4.3 is transformed into GCDFG of Figure 4.4 and two lists are built; a node-list and an edge-list. The lists are built as follows:

- The first instruction is *"Read"* and since it is not a control construct, then a block is encountered.
- An S node is added to the node-list to indicate the beginning of a block.
- The first instruction "*Read*($I1$)" is transformed. An "IN" node is appended to the node-list and an edge labeled with $I1$ is added to the edge-list. The same is done for all *Read* instructions.
- The instruction $I3 = I1 + I2$ is transformed. $I1$ and $I2$ are read (using IN nodes), then the "$+$" operation node is appended to the node-list and its ID is filled at the $I1$ and $I2$ destination node ID in the edge-list. A new edge labeled $I3$ is added to the edge-list.
- All instructions are transformed in the same way. Operation nodes that are not dependent on any other operation are connected to the S node.

- A "*Write*" operation is transformed using OUT node that is added to the nodes list and an edge pointing to that node to the edge-list.

- Operation nodes that have no predecessor are connected to the E node through control edges.

The node-list and the edge-list are shown in Tables 4.5 and 4.6.

Node ID	Node Label	Node Type
1	S	C
2	IN	O
3	IN	O
4	+	O
5	IN	O
6	-	O
7	IN	O
8	*	O
9	=	O
10	+	O
11	+	O
12	IN	O
13	/	O
14	=	O
15	AND	O
16	OR	O
17	=	O
18	=	O
19	OUT	O
20	OUT	O
21	OUT	O
22	E	C

Table 4.5: The node-list.

ID	Label	Type	Source	Destination
1	–	C	1	2
2	–	C	1	3
3	I1	V	2	4
4	I2	V	3	4
5	I3	V	4	6
6	–	C	1	5
7	–	C	1	7
8	I4	V	5	6
9	I5	V	6	10
10	I3	V	4	8
11	I6	V	7	8
12	I7	V	8	11
13	–	C	1	12
14	I3	C	4	9
15	I13	V	9	21
16	I3	V	4	10
17	I8	V	10	15
18	I1	V	2	11
19	I9	V	11	16
20	I10	V	12	13
21	I5	V	6	13
22	I11	V	13	15
23	I1	V	2	14
24	I12	V	14	16
25	I14	V	15	17
26	I15	V	16	18
27	I1	V	17	19
28	I2	V	18	20
29	–	C	21	22
30	–	C	19	22
31	–	C	20	22

Table 4.6: The edges-list.

4.6.2 Scheduling

In this section ASAP scheduling is performed on the GCDFG example of Tables 4.5 and 4.6. The ASAP scheduling algorithm is shown in Figure 4.21. Then ASAP schedule is shown in Figure 4.20. The schedule is stored in the GCDFG by adding

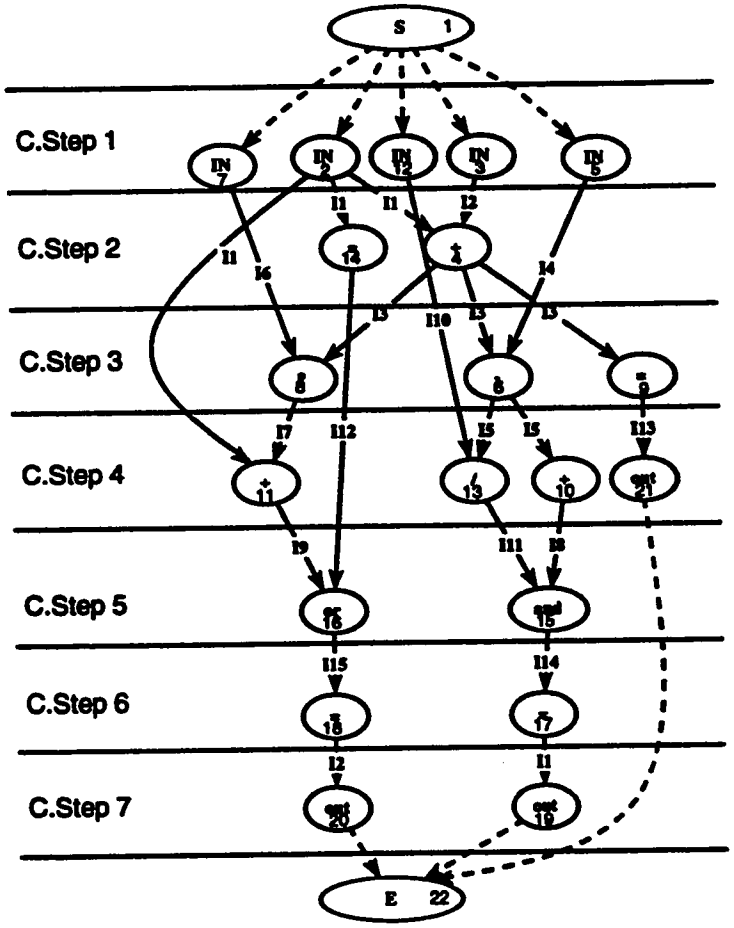

Figure 4.20: An ASAP schedule of the GCDFG of the behavior in Figure 4.3.

an extra field called *Control_Step* to the node-list. This field holds the control step

```
From the Node List Read the 1st S node ID
i:=0;
Put S node ID in CONTROL_STEP[i];
While not (all nodes marked) do
    Begin
        Procedure Search_Edge_List(CONTROL_STEP[i])
        i=i+1;
end;

Procedure Search_Edge_List(CONTROL_STEP[i]);
    begin
        While not EOF(Edges List) do
            begin
            for all nodes in (CONTROL_STEP[i]) do
        if ( Destination Node ID = node in CONTROL_STEP[i] ) then
                begin
                Add Destination Node ID to CONTROL_STEP[i+1]
                Mark added nodes
            endif;
    end;
End;
```

Figure 4.21: ASAP scheduling algorithm.

number for each operation (see Table 4.7).

4.6.3 Allocation

In this section, register allocation is performed using the technique of the FACET system [23]. Register life-time table is constructed by scanning the edge-list. As explained earlier, the columns of the life-time correspond to variables and the rows correspond to control steps. A register is live in the interval between its first defini-

Control Step	Node ID	Node Label	Node Type
–	1	S	C
1	2	IN	O
1	3	IN	O
2	4	+	O
1	5	IN	O
3	6	-	O
1	7	IN	O
3	8	*	O
3	9	=	O
4	10	+	O
4	11	+	O
1	12	IN	O
4	13	/	O
2	14	=	O
5	15	AND	O
5	16	OR	O
6	17	=	O
6	18	=	O
7	19	OUT	O
7	20	OUT	O
4	21	OUT	O
–	22	E	C

Table 4.7: The scheduled GCDFG.

tion and last reference. The life-time table construction algorithm is given in Figure 4.22 and works as follows. The control step in which a variable is first defined is detected by searching the edge-list for all edges that are labeled with the variable name. For each edge, the source node ID is read and the node with the smallest control step number is where the variable is first defined. For example, the variable "*I3*" has three occurrences in the edge-list.

- First occurrence is in record 3:
  - Source node ID is 4, which is in control step number 2
  - Destination node ID is 6, which is in control step number 3
- Second occurrence is in record 9:
  - Source node ID is 4, which is in control step number 2
  - Destination node ID is 9, which is in control step number 3
- Third occurrence is in record No. 11:
  - Source node ID is 4, which is in control step number 2
  - Destination node ID is 10, which is in control step number 4

Therefore, variable "*I3*" is first defined in control step 2 and referenced in control step 4. Therefore, variable "*I3*" is *live* in the interval [2,4]. In the column corresponding to "*I3*", **L** is filled in rows 2, 3 and 4 which correspond to control steps 2,

```
for all edges in the Edges List
    if the edge belong to a variable then
        Begin
        Read(Destination Node ID);
        Search the Node-List for(Destination Node ID)
            if (Destination Node) is of type "O" then
            Read ( Control Step No.) ;
    { This is where the variable is first referenced }

        Read(Source Node ID);
        Search the Node-List for(Source Node ID)
            if (Source Node) is of type "O" then
            Read ( Control Step No.) ;
    { This is where the variable is first defined }
```

Figure 4.22: Algorithm for constructing life-time table from GCDFG.

3 and 4. This last procedure is repeated for all variables. The edge-list is scanned and the life-time table (Table 4.8) is filled accordingly. Variables that do not have overlapping life-times can be allocated the same register. For example "*I2*" and "*I8*" can share the same register since they have non-overlapping life-times. After

Step	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10	I11	I12	I13	I14	I15
1	L	L													
2	L	L	L	L		L	L					L			
3	L		L	L	L	L	L			L		L	L		
4	L		L		L			L	L	L	L	L	L		
5								L	L		L	L		L	L
6														L	L
7	L	L													

Table 4.8: Life-time table.

constructing this life-time table, heuristics can be applied to compact and allocate

registers. Example of these heuristics are: *clique partitioning* algorithm and the *left-edge* algorithm which is widely used for channel routing.

4.7 GCD-List

The GCDFG is stored as an ASCII text file called GCD-List file which is a format for exchanging GCDFG descriptions. This GCD-List file has a lisp-like format and offers the following advantages:

- It allows adding user constraints (or what we refer to as constraints).
- Since it is ASCII text file, then it is machine processible.
- It can be transferred between different platforms.

The GCD-List is nothing but the node-list and the edge-list. The GCD-List file is organized as follows (see Figure 4.23):

- The file starts with a header which states the file name, date and time (see the example in Figure 4.24).
- The header is followed by an input/output section. All input/output ports, their ID, node number, source node and destination node are included (see Figure 4.24).
- The I/O section is succeeded by the node-list follows (see Figure 4.25).

- Finally, the last section gives the edge-list (see Figure 4.26).

Lines starting with "%" are comments.

In the following, an example that shows how this attribute format help the GCDFG accommodate constraints. A number, which represent the maximum allowed delay time for an operation, can be added to each node (operation). The delay of an operation is specified as a number at the end of the list of each operation node (see Figure 4.27).

4.8 Conclusion

After presenting the GCDFG notation, we conclude the following:

- Flow graphs are ideal intermediate forms in high-level synthesis since they are powerful mathematical abstraction that can capture all the information in the behavioral description.
- Behavioral specification is transformed to flow graph in two ways. Either two separate flow graphs (control and data) or a combined control-data flow graph.
- A new flow graph called GCDFG is proposed. This graph is used as a primary intermediate form since it can capture all necessary high-level constructs like loops and conditional jumps.

```
%GCD_LIST  FILENAME   DATE       TIME
(DECLARATION
(VARIABLES
(NAME                ATTRIBUTE1          ATTRIBUTE2 ...)
.
.
.
)
(PORTS
          (INPUT
          (ID1       ID2        ID3        ID4        ID5)
          )
          (OUTPUT
(ID1      ID2        ID3        ID4        ID5)
)
)
)
(NODE_LIST
(         ID         ATTRIBUTE1          ATTRIBUTE2 ...)
(         .          .                   .                )
(         .          .                   .                )
)
(EDGE_LIST
(         ID         ATTRIBUTE1          ATTRIBUTE2 ...)
(         .          .                   .                )
(         .          .                   .                )
)
```

Figure 4.23: The format of the GCD-List ASCII file.

```
%GCD_LIST EXAMPLE2.GCD  JUNE 01, 1994 14:15
(DECLARATION
(VARIABLES
(       I1      <8>     )
(       I2      <8>     )
(       I3      <8>     )
(       I4      <8>     )
(       I5      <8>     )
(       I6      <8>     )
(       I7      <8>     )
(       I8      <8>     )
(       I9      <8>     )
(       I10     <8>     )
(       I11     <8>     )
(       I12     <8>     )
(       I13     <8>     )
)
)
(PORTS
        (INPUT
        (2      3       5       6       9)
        )
        (OUTPUT
        (11 20 21)
)
)
```

Figure 4.24: An example of a GCD-List ASCII file.

```
(NODE_LIST
%     ID     Label  Type
(     1      S      C     )
(     2      IN     O     )
(     3      IN     O     )
(     4      +      O     )
(     5      IN     O     )
(     6      -      O     )
(     7      IN     O     )
(     8      *      O     )
(     9      =      O     )
(     10     +      O     )
(     11     +      O     )
(     12     IN     O     )
(     13     /      O     )
(     14     =      O     )
(     15     AND    O     )
(     16     OR     O     )
(     17     =      O     )
(     18     =      O     )
(     19     OUT    O     )
(     20     OUT    O     )
(     21     OUT    O     )
(     22     E      C     )
)
```

Figure 4.25: Example of GCD-List file (continued).

```
(EDGE_LIST
%     ID    LABEL TYPE SRC  DST
(     1     -     C    1    2    )
(     2     -     C    1    3    )
(     3     I1    V    2    4    )
(     4     I2    V    3    4    )
(     5     I3    V    4    6    )
(     6     -     C    1    5    )
(     7     -     C    1    7    )
(     8     I4    V    5    6    )
(     9     I5    V    6    10   )
(     10    I3    V    4    8    )
(     11    I6    V    7    8    )
(     12    I7    V    8    11   )
(     13    -     C    1    12   )
(     14    I3    C    4    9    )
(     15    I13   V    9    21   )
(     16    I3    V    4    10   )
(     17    I8    V    10   15   )
(     18    I1    V    2    11   )
(     19    I9    V    11   16   )
(     20    I10   V    12   13   )
(     21    I5    V    6    13   )
(     22    I11   V    13   15   )
(     23    I1    V    2    14   )
(     24    I12   V    14   16   )
(     25    I14   V    15   17   )
(     26    I15   V    16   18   )
(     27    I1    V    17   19   )
(     28    I2    V    18   20   )
(     29    -     C    21   22   )
(     30    -     C    19   22   )
(     31    -     C    20   22   )
)
```

Figure 4.26: Example of GCD-List file (continued).

```
(NODE_LIST
%     ID     Label  Type  Delay
(     1      S      C     -     )
(     2      IN     O     5     )
(     3      IN     O     5     )
(     4      +      O     4     )
(     5      IN     O     5     )
(     6      IN     O     5     )
(     7      -      O     11    )
(     8      *      O     4     )
(     9      IN     O     5     )
(     10     =      O     1     )
(     11     OUT    O     5     )
(     12     +      O     9     )
(     13     +      O     9     )
(     14     /      O     12    )
(     15     =      O     1     )
(     16     AND    O     5     )
(     17     OR     O     8     )
(     18     =      O     1     )
(     19     =      O     1     )
(     20     OUT    O     5     )
(     21     OUT    O     5     )
(     22     E      C     -     )
)
```

Figure 4.27: A part of the GCD-List file showing constraints.

- Using the GCDFG as a primary intermediate form facilitates the two main high-level synthesis tasks (scheduling and allocation).

- The GCDFG notation is expressed in a lisp-like format called GCD-List. This makes it able to accommodate user stated constraints. For example, a user constraint which states that some operations should be bounded by a time unit.

- The GCD-List is stored in ASCII text file which makes it exchangable and machine processible.

Chapter 5

Comparative Study of Flow Graphs

In this section, the GCDFG is compared to other *primary intermediate forms* (PIFs) like Value Trace (VT), SALSA and the exchange DFG. The structure of each PIF is studied (edges, nodes, types of nodes, types of edges ... etc.,). There are some essential features that should be possessed by an intermediate form to make it suitable for high-level synthesis. One is, the PIF should inherit all the necessary constructs and details from the high-level specification. It should also support all high-level constructs such as arithmetic and logical operations, hierarchy, conditional constructs and looping, some common data structures like arrays and user defined constraints. However, there are other features that are desirable but not necessary like supporting specific application oriented constructs.

5.1 Value Trace (VT)

Value trace is an intermediate form used in CMU-DA HLS system [19]. In CMU-DA synthesis system the ISPS description which is very similar to structured programming languages is used as input. ISPS allows the specification of certain structures like word length, sequencing methods, and control transfer [8, 14, 18]. ISPS is translated into an intermediate form called VT which specifies all the data flow and control flow information in the original ISPS description.

The VT is a "directed acyclic graph very much like a data flow graph except that control constructs have been retained and translated into their equivalents" [18]. In short, the VT consists of a collection of graphs (VT-bodies) linked by conditional or concurrent control constructs. The blocks in ISPS (procedures, loop bodies, labeled blocks) are translated into VT-bodies. Each VT-body is a set of operations which can be invoked or left as a unit. The nodes in VT correspond to operations (activities). The control nodes, such as subroutine call operation, are called (instantiations). Edges in VT correspond to values (variables). The VT is interpreted as a database. The VT basically has two parts: (i) descriptor for all entities declared in the ISPS description. (ii) operations (VT-bodies). The compiler parses the ISPS description and generates a file with extension 'GDB' (Global DataBase), which is in the form of a tree. This 'GDB' file is translated into VT which is then stored as an ASCII file.

The VT intermediate form supports all high-level constructs like control sequence, loops, conditional constructs and special data structures like arrays. Since it is a flow graph, it also supports concurrency.

In addition to the aforementioned necessary features, the VT has the following desirable features:

- The VT allows user defined constraints such as choosing the design style; use off-the-shelf TTL chips or customs LSI chip.
- The VT representation is unified and this facilitates the following: (i) it allows communication between various levels (ii) it avoids repeating the verification done at the high-level and some optimizations that are applicable to all levels (iii) it avoids the overhead of translating between data structures.
- It contains various control constructs that allow synchronization such as control nodes (STOP, DELAY, WAIT).
- Several types of optimizations can be conducted on the VT like:
  - Dead code elimination can be detected and eliminated by using the use-list for output values. If an operator does not affect synchronization by causing a WAIT, DELAY, or STOP and does not cause a jump in the control, and its values are not referenced anywhere then it is a dead code. This dead code can be eliminated by taking the operator out of

the sequencing lists and eliminating its inputs from the use-lists of the values they reference.

- Constant folding can also be achieved easily in VT. For example if $I = 7$ is an operator in the ISPS description, then in the VT it can be detected that 7 is a constants and all occurrences of the variable I can be replaced by the constant 7.

- The third optimization technique involves moving operations in the VT without violating data dependency. Moving operations will be for the sake of achieving more parallelism. An operator can be moved backward as long as: (i) it does not move into another branch or VT-Body, (ii) it does not move past any of the operators that produce values which it requires as input, and (iii) it does not move past any operator that requires synchronization.

- Code motion is also applied in VT. Operations that are not related to a branch of a FORK or a loop body can be detected and moved outside the loop or outside the branch.

5.1.1 The MIN example

In this section an example that illustrates most of the constructs and features supported by the VT is presented. Figure 5.1 gives an example of ISPS description.

```
min:= BEGIN
                    ** INPUTS **
                    n<5:0> ! number of words in the array
                    S[0:127]<15:0> ! the array and scratchpad

                    ** OUTPUT **
                    z<15:0> ! the minimum value

                    ** INDEX **
                    i<5:0>

                    ** The Algorithm **
start:= BEGIN
                    i<-n next
                    loop:= BEGIN
                        IF i EQL #00 => (z<-S[0] next stop() ) next
                        DECODE (S[2*i-1] LEQ S[2*i]) =>
                            BEGIN
                            0/false:= S[i-1]<-S[2*i],
                            1/true := S[i-1]<-S[2*i-1]
                            END next
                        i<-i-1 next
                        RESTART loop
                        END END END
```

Figure 5.1: ISPS of MIN [17]

This example is represented graphically in Figure 5.2 and the VT database file shown in Figures 5.3, 5.4 and 5.5. The ISPS description in Figure 5.1 finds the minimum of an array. The array is sorted in the upper half of S, beginning at S[n+1.]

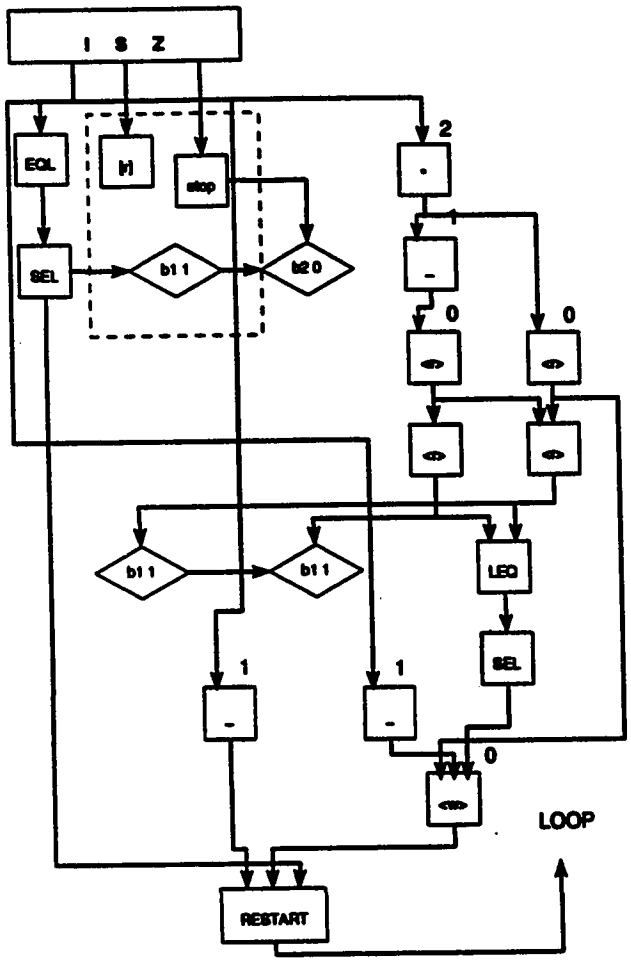

Figure 5.2: VT representation of the MIN example of Figure 5.1

```
!          ENTITY DECLARATIONS
! Typ/ID Decl in FLAGS          Word Structure  Bit Structure   Na
! MAP    To Wd Fact Ed Offs     Bit Offs
s1   *    0000000000000001000   *               *              M;
r1   %s1  0000000000000010000   *               <5:0>          N;
r2   %s1  0000000000000010000   [0:127]         <15:0>         S;
r3   %s1  0000000000000010000   *               <15:0>         Z;
r4   %s1  0000000000000010000   *               <5:0>          I;
v5   %s1  0000000000001000100   *               *              ST
v6   %v5  0001000010100010O     *               *              LO

!                  CONSTANT
!                  ID          Value       Size
                   c1          0           <6>
                   c2          0           <2>
                   c3          0           <7>
                   c4          2           <3>
                   c5          1           <2>
                   c6          0           <8>

!                  VTBODIES
!ID    OPCODE      (INPUTS)
       OUTPUT:     ID          CARRIER  SIZE      NAME

v6   %v5  0001000010100010O     *               *              LO
```

Figure 5.3: The Value Trace of the MIN example of Figure 5.1

```
                i1          r4          <6>       I
                i2          r2          <16>      S
                i3          r3          <16>      Z
x1      EQL     (v6. i1; 1) (*.c1=0)
        p1      *           <1>         *
x2      SELECT
@b1     %x5     (OTHERW)
x3      [r]     (v6.i2:s)   (*.c2=0)
                p1          r3          <16>      z
x4      STOP
=b1     %x5     (x3.p1:z)
@b2     %x5     [0]
=b2     %x5     (v6.i3:z)
x5      ENDSEL  (x1.p1)
                p1          r3          <16>      Z
x6      *       (*.c4=2)    (v6.i1:I)
                p1          *           <9>
x7      -       (x6.p1)                 (*.c5=1)
                p1          *           <10>      *
x8      <r4>    (x7.P1)     *           (*.c6=0)
                p1          *           <7> *
x9      [r]     (v6.12.S)*  (x7.p1)
                p1          *           <16>      *
x10     <r>     (x6.p1)     (*.c6=0)
                p1          *           <7>
x11     (r)     (v6.12.S)   (x6.p1)
                p1          *           <16>      *
```

Figure 5.4: The Value Trace of MIN (Continued from 5.3).

```
x12     LEQ       (x9.p 1)    (x11.pl)
                  pl          *          <1>       *
x13     SELECT
@b1     =b1       %x14        (x11.pl)
@b2     %x14      [1]
=b2     %x14      (x9.pl)
x14     ENDSEL    (x12.pl)
        pl        *           <16>       *
x15     -         (v6.il:I)                (*.c5=1)
                  pl          *          <7>       *
x16   [w]         (v6.i2.S)   (x15.pl)    (x14.pl)  (*.c6=0)
                  pl          r2         <16>      S
x17   -           (v6.il:1)   (*.c5=1)
                  pl          r4         <6>       1
x18   RESTART     V6:LOOP     (x17.pl:l) (x16.pl:S) (x5.pl:Z)
                  o1          r3         ¡16¿      Z
                  o2          r2         ¡16¿      S
                  o3          r4         ¡6¿       I

v5  %s1  000000000001000100      *    *    ST

                  i1          r4         <6>       1
                  i2          r2         <16>      S
                  i3          r3         <16>      Z
x1      CALL      @v6:LOOP    (v5.i1:1)  (v5.12:S) (v5.13:Z)
                  p1          r3         <16>      Z
                  p2          r2         <16>      S
                  p3          r4         <6>       1
LEAVE   @vS:STAR  (xl.p3:l)   (xl.p1:Z)  (xl.pz:S)
                  o1          r4         <6>       1
                  o2          r3         <16>      Z
                  o3          r2         <16>      S
```

Figure 5.5: The Value Trace of MIN (Continued from 5.4).

5.2 SALSA

SALSA is a modified Control-Data Flow Graph (CDFG) that allows alternative schedules to be quickly explored while maintaining timing constraints. SALSA is a directed graph in which nodes represent operations and edges represent ordering dependencies between operations. *Source* and *sink* nodes represent the beginning and end of activities in the graph. Edges between nodes represent three different types of ordering dependencies. They are:

1. Data edges, which represent the flow of data from one operation to another, implying an ordering relationship because the data must be computed before it is used.

2. Control edges, which represent ordering relationships associated with control operations such as conditional constructs.

3. Timing edges, which represent constraints between two operators that must be satisfied in a correct design. There is a minimum timing constraint and a maximum timing constraint.

Formally, SALSA can be represented by a directed graph G(V,E) where V represent the set of nodes and E represent the set of edges. The set of nodes include a source node v_{src}, a sink node v_{sink} and operation nodes v_1 to v_n. Each of these nodes has a delay attribute to help explore different schedules.

SALSA supports all necessary high-level constructs like hierarchy, conditional constructs and looping. It is not clear whether it supports arrays or not. Conditional constructs in SALSA are implemented as shown in Figure 5.6. A list of input conditions is attached to each operation representing the condition under which it is activated. Each row of this list represents a set of input conditions encoded as 0, 1 or X (don't care). The universal condition (XXX) is attached to unconditional operations. Conditional operations, that produce data values require a multiplexer node to select the proper branch (see Figure 5.6). Control operators that change

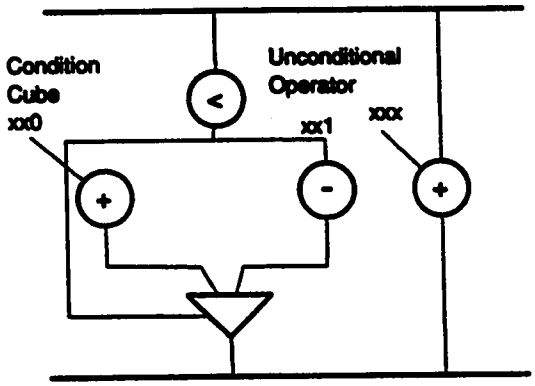

Figure 5.6: Conditional operations in SALSA.

control flow (like loop restart) do not require a multiplexer. Each branch of a conditional construct has a list of operations. If the intersection of the two branch lists is empty then, the two branches are mutually exclusive and can share the same functional unit.

SALSA allows the use of subroutines. A single instance of the subroutine is transformed. Then, the subroutine can be called as many times as needed. Each subroutine call node (see Figure 5.7) calls the transformed subroutine by a pointer pointing to the location of the transformed subroutine instance. Loops are treated

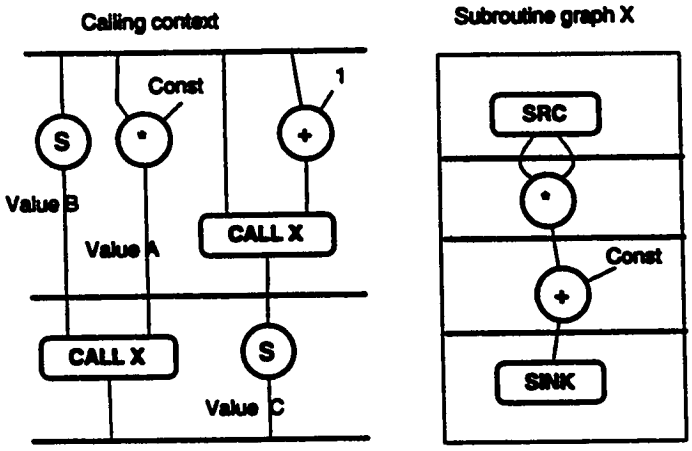

Figure 5.7: Subroutines in SALSA.

as a special case of subroutines. Each loop is represented by a separate graph. Loop execution is initiated using a CALL node and new iterations are initiated using a RESTART operation. Like in VT, WAIT operations are used for external synchronization.

The main desirable feature of the SALSA intermediate form is its ability to explore the scheduling space by trying different schedules.

5.3 Exchange DFG

As seen in Chapter 1, in ESPRIT [28] synthesis system the high-level language is translated into an *extended DFG* in which data and control flow graphs are combined. The DFG nodes represent operations in the behavioral specification, and the edges model the transfer of values between operations. Several node types are defined in this DFG. These types of nodes allow the DFG to support various constructs (see Chapter 1).

5.4 Generic CDFG

As seen in Chapter 4, GCDFG supports all high-level constructs like hierarchy, looping, conditional constructs and arrays.

The proposed Generic Control and Data Flow Graph (GCDFG) has the following extra features over ordinary CDFGs:

- Adding a control edge (dashed edge) between two operations (nodes at the same level) will force one to be executed before the other. This facilitates forcing a specific execution order of operations.

- User defined constraints can be easily specified. For example, adding a *delay* field to the node-list makes it possible to limit the operation delay time, which means facilitating constraints compulsion. In fact, any user defined constraints

on nodes (operation) or edges (variables) can be added easily. This is due to the attribute format of the GCD-List.

- It is complete. In other words, given the GCDFG of a behavioral description, an equivalent behavioral description can be rebuilt from the corresponding GCDFG. This allows some high-level optimization techniques to be directly applied on the intermediate from. For example *"code motion"* can be easily applied on the GCDFG, this is done easily by just modifying few entries in the edge-list.

- Independent of any input language.

- The GCDFG is simple, only four types of control nodes are used to express all control constructs like: *if-then-else, while* loops, *for* loops, *repeat-until* loops, *procedures* and *procedure calls.* These control nodes are: *S, E, fork* and *join.*

- Scheduling information can be stored using this intermediate form. This has been illustrated by an example and it is achieved by adding a *control step* field to the node-list. This field is filled for operation nodes only and it contains the control step number in which the corresponding operations are scheduled.

- Information necessary for allocation can also be extracted from the GCDFG. For example, life-time table of variables is built from their life-time intervals. A life-time interval of a variable can be obtained from the GCDFG as seen

earlier.

- The GCDFG is stored in a GCD-List file. This GCD-List file can be used in all synthesis procedures. This is because the attribute format allows it to accommodate as many details as needed. For example, schedules can be interpreted using the same GCD-List file. This is done, as seen earlier, by adding a control step field to the node-list.

- It is not tied to a particular architectural style.

Table 5.4 compares the various intermediate forms discussed in this chapter.

Intermediate Form	Exchange DFG	SALSA	Value Trace	Generic CDFG
Representation	Graph	Graph	Language, Graph	Graph
Structure				
Nodes	Operations IN,OUT, BR,ME, GET,PUT, ARRAY,UPD	Operations CALL, SRC, SINK	Operations BR, SELECT, LEAVE, STOP, DELAY, WAIT,	IN,OUT, S,E, FORK, JOIN ARRAY
Edges	transfer control	Conditions		variables, control
Constructs				
Concurrency	Yes	Yes	Yes	Yes
Arrays	Yes	No	Yes	Yes
Hierarchy	No	Yes	Yes	Yes
Conditional	Yes	Yes	Yes	Yes
Looping	Yes	Yes	Yes	Yes

Table 5.1: Intermediate Forms Comparison.

Chapter 6

Conclusion and Future Work

This research has introduced high-level synthesis, its main tasks and the intermediate forms used. The intermediate form is an internal representation of a specification that is more suitable for automatic handling. Then, a comparative study discussing synthesis tasks (transformation, scheduling and allocation) versus several intermediate forms has been conducted.

In Chapter 2, a classification framework has been introduced to classify intermediate forms according to the synthesis tasks. Two main categories are identified: *primary* and *secondary*. Primary intermediate forms (PIF) are those that inherit all the details from the high-level specification and result from the transformation step. Secondary intermediate forms (SIF) are those that extract specific information from the primary intermediate form to perform a specific synthesis task (scheduling and allocation).

Finally, a generic, modular and flexible primary intermediate form, that is called generic control-data flow graph (GCDFG), has been introduced. It is generic because it supports common high-level constructs like, arithmetic and logic operations, control transfer (looping, conditional constructs, hierarchy) and special data structure like arrays. This PIF is expressed in a lisp-like format called the GCD-List. The Lisp-like format gives the GCDFG flexibility because attributes can be added as required. The GCD-List is stored as an ASCII file which makes it machine processible and portable to different platforms [13].

Since this PIF is a flow graph then, it detects the potential parallelism in a behavioral specification. A complexity analysis has been conducted to show the space and the transformation complexities of the proposed intermediate form.

A comparative study that compares the GCDFG with Value Trace, SALSA and the Exchange DFG has been conducted. Necessary and desirable features of each intermediate form has been surveyed. From the comparative study we conclude that the proposed intermediate form supports all the generic features in other forms. However, it is simpler and easier to manipulate.

From the above we conclude that flow graphs are the ideal intermediate forms in high-level synthesis because:

- They inherit all necessary details and constructs from the behavioral specification.

- They carry all necessary details down to secondary intermediate forms to perform the rest of the synthesis tasks.

- Most scheduling and allocation techniques are applicable to flow graphs.

The proposal of this new CDFG, can be the stepping stone in defining new standards and solid definitions of primary intermediate forms that are used in high-level synthesis. As future work, experimental evaluation should be conducted on the GCDFG. Moreover, the GCD-List format needs more fine tuning.

Bibliography

[1] Said Amellal and Bozena Kaminska. Functional Synthesis of Digital Systems with TASS. *IEEE on Computer-Aided Design of Integrated Circuits and Systems*, 13(5):537–552, May 1994.

[2] William P. Birmingham, Anurag P. Gupta, and Daniel P. Siewiorek. The MICON System for Computer Design. In *26th ACM/IEEE Design Automation Conference*, pages 135–154, 1989.

[3] Raul Camposano. Path-Based Scheduling for Synthesis. *IEEE Transactions on Computer Aided Design*, 10(1):85–92, January 1991.

[4] Raul Camposano, Michael McFarland, and Alice C. Parker. The High-Level Synthesis of Digital Systems. *Proceedings of IEEE*, 78(2):301–317, February 1990.

[5] Raul Camposano and Wolfgang Rosentiel. Synthesizing Circuits from Behavioral Descriptions. *IEEE Transactions on Computer Aided Design*, 8(2):171–179, February 1989.

[6] Raul Camposano and R.M. Tabet. Design Representation for the Synthesis of Behavioral VHDL Models. In J. A. Darringer and F. J. Rammig, editors, *Computer Hardware Description Languages and Their Application*, pages 49–59, North-Holland, 1990. Elsevier Science Publishers B.V.

[7] M. Fujita, S. Kono, H. Tanaka, and Moto-Oka. Aid to hierarichal and structured logic design using temporal logic and PROLOG. *IEEE Proceedings*, 133(5):283–294, September 1986.

[8] Daniel D. Gajski. *Silicon Compilation*. Addison-Wesley Publishing Company, Inc., New Jersey, 1988.

[9] Daniel D. Gajski. Essential Issues and Possible Solutions in High-Level Synthesis. In Raul Camposano and Wayne Wolf, editors, *High-Level VLSI Synthesis*, pages 1–26, Boston/Dordrecht/London, 1991. Kluwer Academic Publisher.

[10] Baher S. Haroun and Mohamed I. Elmasry. Architectural Synthesis for DSP Silicon Compilers. *IEEE Transactions on Computer-Aided Design*, 8(4):431–446, April 1989.

[11] A. A. Jerraya and K. O'Brien. Bridging the Gap between Case Tools and Hardware Design Tools. Technical report, Institute National Polytechnique de Grenoble, 46, avenue Felix Viallet - 38031 GRENOBLE CEDEX, France, 1992.

[12] A. A. Jerraya and K. O'Brien. SOLAR: An Intermediate Format for System Level Design and Specification. Technical report, Institute National Polytechnique de Grenoble, 46, avenue Felix Viallet - 38031 GRENOBLE CEDEX, France, 1992.

[13] David W. Knapp and Alice C. Parker. A Unified Representation for Design Information. In C.J. Koomen and T. Moto-oka, editors, *Computer Hardware Description Languages and their Applications*, pages 337–353. Elsevier Science Publisher B.V., North-Holland, 1985.

[14] T. J. Kowalski and D. E. Thomas. The VLSI Design Automation Assistant: What's in a Knowledge Base. In *22nd Design Automation Conference*, pages 252–258, 1985.

[15] Randy K. Lind and K. Vairavan. An Experimental Investigation of Software Metrics and Their Relationship to Software Development Effort. *IEEE Transactions on Software Engineering*, 15(9):649–653, May 1989.

[16] Th. Krol J. V. Meerbergen, C. Niessen, W. Smits, and J. Huisken. The Sprite Input Language: An intermediate format for High-Level Synthesis. *IEEE*, pages 186–192, 1992.

[17] Barry Michael, Pangrle, and Daniel D. Gajski. Design Tools for Intelligent Silicon Compilation. *IEEE Transactions on Computer-Aided Design*, 6(6):1098–1112, November 1987.

[18] C. Michael and S. J. Mc Farland. The Value Trace: a Database for Automated Digital Design. Technical report, National Science Foundation, Carnegie-Mellon University, Pittsburg, Pennsylvania 15213, December 1978.

[19] Andrew W. Nagel, Richard Cloutier, and Alice C. Parker. Synthesis of Hardware for the Control of Digital Systems. *IEEE Transactions on Computer-Aided Design*, 1(4):201–212, October 1982.

[20] I. Park, K. O'Brien, and A. A. Jerraya. A VHDL-Based Scheduling Algorithm For Control-Flow Dominated Circuits. Technical report, Institute National Polytechnique de Grenoble, 46, avenue Felix Viallet - 38031 GRENOBLE CEDEX, France, 1992.

[21] I. Park, K. O'Brien, and A. A. Jerraya. AMICAL: Architectural Synthesis Based on VHDL. Technical report, Institute National Polytechnique de Grenoble, 46, avenue Felix Viallet - 38031 GRENOBLE CEDEX, France, 1992.

[22] Pierre G. Paulin and John P. Knight. Algorithms for high-level synthesis. *IEEE Transactions on Computer Aided Design*, 8(4):18–31, December 1989.

[23] Pierre G. Paulin and John P. Knight. Force-Directed Scheduling for the Behavioral Synthesis of ASIC's. *IEEE Transactions on Computer Aided Design*, 11(8):661–678, June 1989.

[24] Minjoong Rim and Rajiv Jain. Representing Conditional Branches for High-Level Synthesis Applications. In *29th ACM/IEEE Design Automation Conference*, pages 106–111, 1992.

[25] Sadiq M. Sait, Habib Youssef, Muhammad S. T. Benten, and Faisal M.Z. Soleja. Automated VHDL Composition from AHPL. Computer Engineering Department, KFUPM, Saudi Arabia.

[26] Sadiq M. Sait, Habib Youssef, Muhammad S.T Benten, and Faisal M.Z Soleja. Design and Implementation of an Integrated Design Automation Environment for High-Level Silicon Compilation, May 1991. Computer Engineering Department, KFUPM, Saudi Arabia.

[27] Toshika Tanaka, Tsutomu Kobayashi, and Osamu Karatsu. HARP: Fortran to Silicon. *IEEE Transactions on Computer Aided Design*, 8(6):649–660, June 1989.

[28] Jos T.J. and Leon Stok. A Data Flow Graph Exchange Standard. *IEEE Transactions on Computer Aided Design*, pages 193–199, 1992.

[29] H Trickey. FLAMEL: A High-Level Hardware Compiler. *IEEE Transactions on Computer Aided Design*, 8(6):259–269, March 1987.

[30] Fur-Shing Tsai and Yu-Chin Hsu. STAR: An Automatic Data Path Allocator. *IEEE Transactions on Computer Aided Design*, 11(9):1053–1064, September 1992.

[31] Chai-Jeng Tseng and Daniel P. Siewiorek. Automated Synthesis of Data Paths in Digital Systems. *IEEE Transactions on Computer Aided Design*, 5(3):379–394, July 1986.

[32] Robert A. Walker and Raul Camposano. *A Survey of High-Level Synthesis Systems*, volume 8. Kluwer Academic Publishers, February 1989.

Vita

- Essam Mohammad Khair Hubbi.
- Born in Damascus, Syria.
- Received Bachelor's degree in Computer Engineering from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia in January, 1991.
- Completed Master's degree requirements at King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia in November, 1994.