AN INTERACTIVE APPROACH FOR TRADING
TEST TIME, AREA, AND FAULT COVERAGE IN
TESTABLE SYNTHESIS

by

HOUSSAM TOUMA

Submitted in partial fulfillment of the requirements
for the Degree of Masters of Science

Thesis Advisor: DR. HAIDAR HARMANANI

Department of Computer Science
LEBANESE AMERICAN UNIVERSITY
April 2006
I grant to the LEBANESE AMERICAN UNIVERSITY the right to use this work, irrespective of any copyright, for the University's own purpose without cost to the University or to its students, agents and employees. I further agree that the University may reproduce and provide single copies of the work, in any format other than in or from microforms, to the public for the cost of reproduction.

[Signature]

[Name]
Thesis Approval

Student Name: HOUSSAM TOUMA
Thesis Title: AN INTERACTIVE APPROACH FOR TRADING TEST TIME, AREA, AND FAULT COVERAGE IN TESTABLE SYNTHESIS
I.D.#: 199731050

Program: Computer Science
Division /Dept: Computer Science and Mathematics
School: School of Arts and Sciences, Byblos

Approved by:
Thesis Advisor: HAI DAR M. HAR MAN ANI

Member: DANIELLE AZAR
Member: MOUNJED MOUSSALLAM

Member

Date: MARCH 24, 2006
To my Family
Contents

Chapter 1 Introduction ........................................................................... 1
Chapter 2 Background and Review of Literature ........................................... 4
  2.1 Background .............................................................................. 4
  2.2 Software and hardware design: parallelism ...................................... 10
  2.3 Software development Tools ..................................................... 11
  2.4 Hardware development tools ..................................................... 12
  2.5 Summary .................................................................................. 14
Chapter 3 Synthesis and Layout Algorithms .................................................. 15
  3.1 Data Flow Graph ....................................................................... 16
  3.2 Scheduling ................................................................................. 19
    3.2.1 ASAP (As Soon As Possible) scheduling ............................... 20
    3.2.2 ALAP (As Late As Possible) scheduling ............................... 22
    3.2.3 ASAP Scheduling with Resource Constraint ....................... 24
  3.3 Initial Data Path (IDP) ................................................................ 26
  3.4 Channel Routing ....................................................................... 29
Chapter 4 Allocation/Binding ..................................................................... 33
  4.1 Structure ................................................................................. 33
  4.2 Module Allocation Graph ................................................................ 35
  4.3 Gain and Loss in TFBs Merging .................................................. 36
  4.4 Cost function ........................................................................... 37
  4.5 Allocation Algorithm .................................................................. 37
    4.5.1 Module Allocation ............................................................. 40
    4.5.2 Mux Allocation .................................................................. 40
  4.6 Experimental results ................................................................... 44
Chapter 5 Conclusion .............................................................................. 46
Appendix A GUI functionalities and screenshots ............................................. 48
  A.1 Considering Java ..................................................................... 48
  A.2 GUI screenshots ....................................................................... 49
    A.2.1 Menu toolbar ................................................................... 50
    A.2.2 Main functions screenshots .............................................. 53
Bibliography .................................................................................. 64
List of Figures

Figure 2.1 (a) VHDL Code, (b) Data Flow Graph .................................................. 7
Figure 2.2 (Technological library xml format) .......................................................... 8
Figure 2.3 (diffeq Datapath) .................................................................................... 9
Figure 2.4 (Process of software & hardware development) ........................................ 11
Figure 3.1 (High level synthesis design flow) ............................................................ 15
Figure 3.2 (Parsing and transforming) ...................................................................... 17
Figure 3.3 (Simplification VHDL code) .................................................................... 18
Figure 3.4 (VHDL_to_DFG algorithm) .................................................................... 18
Figure 3.5 (ASAP schedule of diffeq) ...................................................................... 20
Figure 3.6 (ASAP algorithm) .................................................................................... 21
Figure 3.7 (ALAP schedule of diffeq) ...................................................................... 22
Figure 3.8 (ALAP Algorithm) ................................................................................... 23
Figure 3.9 (ASAP with resource constraints, *=2, +=1, -=1) ..................................... 25
Figure 3.10 (Initial Data Path Algorithm) ................................................................. 26
Figure 3.11 (Initial Datapath of ASAP DFG diffeq) .................................................. 28
Figure 3.12 (channel routing problem) ..................................................................... 29
Figure 3.13 (left-edge algorithm) ............................................................................. 31
Figure 3.14 (data path) ............................................................................................ 32
Figure 4.1 (Datapath Allocation Algorithm) ............................................................... 41
Figure 4.2 (Merging Two TFBs) .............................................................................. 41
Figure 4.3: (a) Testable Functional Block, (b) Self Testable ALU with Self-Adjacency, (c) Non-Observeable ALU due to Self-Adjacency ..... 42
Figure 4.4 (Biquad filter DFG) ................................................................................ 42
Figure 4.5 (Biquad Module Allocation graph) ............................................................ 43
Figure 4.6 (Initial data path (IDP) for the biquad example) ....................................... 43
Figure 4.7 (Intermediate solution after merging TFBA0 and TFBA1) ......................... 44
Figure A.1 (main window) ....................................................................................... 49
Figure A.2 (File menu) ............................................................................................ 50
Figure A.3 (Scheduling menu) ................................................................................ 51
Figure A.4 (Options menu) ...................................................................................... 52
Figure A.5 (Look & Feel menu) .............................................................................. 52
Figure A.6 (File > New, new work sheet) ................................................................. 53
Figure A.7 (dragging a component into the drawing window) ................................... 54
Figure A.8 (adding a wire between 2 components) ................................................... 55
Figure A.9 (showing property sheet) ....................................................................... 56
Figure A.10 (file chooser dialog) ............................................................................ 57
Figure A.11 (the datapath of an rtl file) ................................................................. 58
Figure A.12 (Simple Image Editor) ......................................................................... 59
Figure A.13 (file chooser dialog) ............................................................................ 60
Figure A.14 (ASAP DFG graph) .............................................................................. 61
Figure A.15 (initial data path) ................................................................................ 62
Figure A.16 (MAG graph) ...................................................................................... 63
List of Tables

Table 4.1 Biquad.................................................................45
Table 4.2 Diffeq.................................................................45
Table 4.3 Poly_design..........................................................45
Table 4.4 Elliptic.................................................................45
ACKNOWLEDGEMENTS

First, I would like to thank God for his blessings and protection without which I would not have made it this far.

I would like to thank my thesis advisor, Dr. Haidar Harmanani for his assistance throughout this research project and for his guidance throughout my M.S studies.

I would also like to express my greatest appreciation to Mr. Munjid Musallem for all of his help and guidance throughout my B.S and M.S studies.

I would also like to thank Dr. Danielle Azar for her help and for being on my thesis committee.

I would like to express my sincere gratitude to the Lebanese American University whose financial support during my graduate studies made it all possible.

Finally, I would like to express my thanks and love to my parents, to my wife and son and to my brothers and sisters for their love and patience.
ABSTRACT

by

Houssam Touma

Over the past decade circuits' complexity has increased while the transistor count on a single chip increased tremendously. Research on computer-aided design (CAD) tools also progressed as circuit complexity increased. This increase in circuits' complexity led to the appearance of High-Level synthesis. High-Level synthesis can capture system specification at a higher level of abstraction, which will reduce the circuit design complexity. In this Thesis, we develop a High-level synthesis prototype that is flexible and that allows designers to interact with the system through a graphical user interface. This tool can be easily extended and new features can be implemented and plugged in without difficulties. The goal for a High-Level synthesis tool is the generation of a datapath from the circuit's behavioral description. This thesis presents a method for interactive test synthesis for RTL designs. The method provides a mechanism to redesign RTL datapath into testable ones while performing the allocation of functional units, registers, and interconnects. Our tool implements the main functions in a high-level synthesis and it proposes techniques for the allocation task that uses an iterative improvement optimization method that minimizes the area and power consumption. The system has been implemented using Java and favorable results are reported.
Chapter 1

Introduction

The first microchip was invented in 1959 with a transistor count of one. In 1964, complexity grew up to 32 transistors, and in 1965, a chip in the Fairchild R&D lab had 64 transistors [26]. In the year 1970 the number of transistors on a single chip increased from around 10,000 to 100,000. It went up to 1 million in the 1980s and reached 40 million transistors per chip in the 1990s. Clock frequency ranged from 200KHz to 2MHz in the 1970s. It increased to 20MHz in the 1980s then to 800MHz in the 1990s [18]. Today, chip density like “Pentium Prescott” can reach approximately 150 million transistors per chip, with a clock speed of 3.8GHz [16]. According to International Technology Roadmap for Semiconductor (ITRS) projections, the number of transistors per chip and the clock frequencies for high-performance microprocessors will continue to grow exponentially in the next 10 years too. The 2003 ITRS predicts that by 2014 microprocessor’s clock frequency will rise to almost 30 GHz with a transistor count of 1 billion transistors per chip [17]. This growth is the outcome of several factors, the evolution of process technology and design & development CAD tools [26].

From the 1970s till now, researches on computer-aided design (CAD) tools progressed as circuit complexity increased. These researches lead to the creation of High-level Synthesis tools in late 1980.

The first definition of High-Level Synthesis was stated by McFarland et al. [19]:

"High-Level Synthesis task is to take a specification of the behavior required of a
system and a set of constraints and goals to be satisfied, and to find a structure that implements the behavior while satisfying the goals and constraints.”

In other words “High-level Synthesis is the process of mapping a behavioral description at the algorithmic level to a structural description in terms of functional units, memory elements and interconnections.” [11]

High-level Synthesis tools are capable of capturing system specifications at a high level of abstraction. These tools take a textual algorithmic specification and generate a register transfer level RTL. Hardware Description Languages (HDL) such as VHDL (Very high speed integrated circuit Hardware Description Language) and Verilog [3] are input to the High-level Synthesis tool. HDL code is composed of sets of interconnected components. Circuit designers really appreciated the High-level synthesis tools and especially the matured ones.

With the help of High-level synthesis tools, designers are not obliged to use Boolean expressions and low level logic tools. Instead, all they need to do is to describe the circuit behavior into some form of HDL code and this HDL will then be transformed by the high-level synthesis tool into a circuit.

Thus, logic design is becoming more like software programming. Software programmers concentrate on the behavior of their applications and not on how the processor is processing the commands. In the same way, with the use of high-level synthesis tools, circuit designers will not concentrate on logic level details. Instead, they will only need to write the behaviors of the circuits in an HDL code. Also, designers who only have minimum experience in hardware design will be able, with the use of high-level synthesis tool, to design digital circuits.
This thesis is structured as follows: Chapter 2 presents a brief overview of previous high-level synthesis tools and a brief description of high-level synthesis concepts. It also provides a comparison between software and hardware synthesis and between software and hardware development tools. Chapter 3 presents a deep study of high-level synthesis concepts described in the literature such as data representation and scheduling. In addition, it contains the implementations of some algorithms used in this tool. Chapter 4 presents the study of allocation problems and optimization heuristics for solving it. Chapter 5 summarizes the findings, presenting concluding remarks and the problems we faced.
Chapter 2

Background and Review of Literature

2.1 Background

High level synthesis tools have been heavily investigated in the past years. Some are commercial, others academic. The commercial tools are of high quality like IBM BooleDozer [14], Synopsys Behavioral Compiler [27], and Cadence Synergy.

In academia, there are also some tools like Stanford Olympus [8] and U.C. Berkeley Hyper [23], which are of less quality and functionalities than the commercial ones. We have to consider several factors while constructing an RTL (Register Transfer Level) implementation like area, performance, testability and routability. Synthesis tools like CMU-DA [21] and the Yorktown Silicon Compiler [5] consider few factors in generating the RTL design: area and delays of functional units. Nowadays, high level synthesis tools are more advanced and take into consideration testability and other issues. As mentioned in the introduction, the input for High Level Synthesis tools are textual algorithmic description in the form of procedural statements similar to those of a high level programming language. VHDL and Verilog [3] are the languages used for hardware algorithmic description. In our tool, we use a simplified form of VHDL input code. The output of these tools will be an RTL (register transfer level) circuit which is made up of a network of registers, Arithmetic Logic Units (ALU), multiplexers, and busses [13]. For easier machine representation, the VHDL code is transformed to a data flow graph (DFG), and this DFG will then undergo all the processes of high level synthesis until the generation of an RTL.
Figures 2.1(a), 2.1(b) and 2.2, show an example of the VHDL code, the data flow graph and the technology library, respectively. These figures will be explained later in this thesis. The technology library is also used as input to the synthesis tool, which represents the description of primitive modules.

Figure 2.3 represents the register transfer level (RTL) that was generated by our high level synthesis tool. The most important tasks in synthesis tools were followed in mapping the internal description of the VHDL algorithm to a hardware configuration (RTL) that obeys the hardware model of the synthesis system [11] are:

- **Operations Scheduling:** The scheduling problem consists of placing operations from the data flow graph into specific order or steps to satisfy data-dependencies and global resource constraint.

- **Data Path Allocation (binding):** The allocation problem consists of choosing which type of resources found in the technology library (function units, register and multiplexers) will execute the operations scheduled in the data flow graph.

Although the register-transfer level implementation generated by following these previous tasks satisfies all constraints, it is neither unique nor optimal. The RTL depends on the scheduling algorithm used, the allocation algorithm and the technology library.

The scheduling and allocation algorithms have been shown to be NP-Hard [9]. Thus, our synthesis tool uses some well known scheduling algorithm and some heuristics in solving the allocation problem and the merging process. Different solutions can be found for these algorithms and as a result different RTL implementations will be generated.
The high-level synthesis process involves navigation through the design space making tradeoffs until the best solution satisfying the constrained is reached [13].

A lot of previous research has been done in high-level synthesis that focus on optimizing test time and overhead [29]. Papachristou et al [20] proposed a method that generated self-testable designs without self-adjacent registers. The method was later improved in Harmanani et al [13]. Avra [4] worked on a different allocation method to minimize the number of self-adjacent registers in the datapath and resolves the conflicts due to remaining self-adjacency by using CBILBOs. Another method was introduced by Parulkar et al. [22] that minimizes the sharing of registers in order to decrease BIST (Built-In Self Test) area cost. The aim of the method is to ensure that the functional area is not compromised in the quest for low BIST area overhead.

Bukovian et al. [6] proposed an allocation for BIST method that optimizes cost/quality trade-off relation during testable circuit synthesis.

Our goal in this Thesis is the generation of an RTL data path which satisfies all user constraints and where the cost of registers and the power consumption are minimized. We will go over these issues in detail in Chapter 4. In the next section we will discuss the similarities between software and hardware developments and review the history of software development tools. This helps clarify how to design better hardware tools.
DPS Description of: diffeq.vhdl
Input: uinport yinport aport dexport ximport
Output: woutport youtport xoutport
InOut:
Preset Variables:
Preset Values:
Constants:
Constant Values:
Bit Width:

\[ t_1 := u_{\text{inport}} \cdot d_{\text{export}} \]
\[ t_2 := 3 \cdot x_{\text{import}} \]
\[ t_3 := 3 \cdot y_{\text{import}} \]
\[ t_4 := t_1 \cdot t_2 \]
\[ t_5 := d_{\text{export}} \cdot t_3 \]
\[ t_6 := u_{\text{import}} - t_4 \]
\[ u_{\text{var}} := t_6 - t_5 \]
\[ y_1 := u_{\text{var}} \cdot d_{\text{export}} \]
\[ y_{\text{var}} := y_{\text{import}} + y_1 \]
\[ x_{\text{var}} := x_{\text{import}} + d_{\text{export}} \]
\[ x_{\text{output}} := x_{\text{var}} \]
\[ y_{\text{output}} := y_{\text{var}} \]
\[ u_{\text{output}} := u_{\text{var}} \]

Figure 2.1 (a) VHDL Code, (b) Data Flow Graph
<?xml version="1.0" encoding="UTF-8"?>
<Library>
   <Components>
      <ComponentType>alu</ComponentType>
      <ComponentName>multiplier</ComponentName>
      <Function>\</Function>
      <Bits>1</Bits>
      <Area>10</Area>
      <Delay>11</Delay>
      <Power>12</Power>
   </Components>
   <Components>
      <ComponentType>alu</ComponentType>
      <ComponentName>muladd</ComponentName>
      <Function>+</Function>
      <Bits>1</Bits>
      <Area>10</Area>
      <Delay>11</Delay>
      <Power>12</Power>
   </Components>
   <Components>
      <ComponentType>alu</ComponentType>
      <ComponentName>sub</ComponentName>
      <Function>-</Function>
      <Bits>1</Bits>
      <Area>10</Area>
      <Delay>11</Delay>
      <Power>12</Power>
   </Components>
   <Components>
      <ComponentType>alu</ComponentType>
      <ComponentName>divide</ComponentName>
      <Function>/</Function>
      <Bits>1</Bits>
      <Area>10</Area>
      <Delay>11</Delay>
      <Power>12</Power>
   </Components>
   <Components>
      <ComponentType>alu</ComponentType>
      <ComponentName>divsub</ComponentName>
      <Function>-</Function>
      <Bits>1</Bits>
      <Area>10</Area>
      <Delay>11</Delay>
      <Power>12</Power>
   </Components>
</Library>

Figure 2.2 (Technology library xml format)
2.2 Software and hardware design: parallelism

When a programmer wants to build a software system he/she starts by analyzing software specifications and then typically illustrates these specifications, by writing programs using high level languages such as Java, C, etc. Nowadays, there exist some higher level ways for describing specifications than writing programs, example as visual GUI tools or visual UML tools. These tools allow any person, not necessarily a programmer, to describe specifications captured. They are becoming more and more popular and maybe some day they will become the standard way for building software system. However, these tools will almost always need a lot of tweaking and lack of standardization. So, the typical and standard way for describing specifications remains, for now, writing high level language code.

After writing the program, a compiler will translate the high level code into machine language or into binary code (or bytecode in the case of java). Between these two steps (writing the code and translating it into binary code) there exist intermediate steps that are responsible for assembly code generation and optimization. The route of software development is described in Figure 2.4(a).

As for circuit designers, similar to software programmers, they will start by analyzing the system specification and describe them by using an HDL language, like VHDL or some simplified form of HDL like the one previously showed Figure 2.1(a). The HDL code will then be inserted into the high level synthesis compiler where it will be translated into an intermediate form such as Data Flow Graph (DFG). The DFG will then undergo several steps of optimization such as scheduling and allocation. The output of the high level synthesis tool will be a set of interconnected components (netlist) that obeys the technology library rules (Figure 2.2) described previously. The
netlist will pass through the last phase in which it will be drawn and the components will be connected and routed together. The route of hardware development is described in Figure 2.4(b).

![Diagram](image)

*Figure 2.4 (Process of software & hardware development)*

Because of the similarities between software and hardware design, reviewing the changes in software development will lead us to where the hardware development tools should be heading to. In the next section, we will briefly go through the software development evolution and try to deduce the evolution of the hardware development.

### 2.3 Software development Tools

When programmable machines were created, programmers needed to turn switches on or off to make some changes in the system. These machines were usually rooms containing wires and some electronics. In those days, keyboards did not even exist. Punch-cards were next introduced in order to improve the input method, while the output was presented using cathode-ray-tubes (CRT). "*Keyboard equipped machines were used to turn human-readable text into punch-cards.*" [24]

When computers and microprocessors were introduced in the last century, programmers had to use binary code to describe their algorithms or simple system specifications. A little later assembly language was introduced, and more complex
specifications. A little later assembly language was introduced, and more complex system specifications could then be implemented. Move, jump, and compare are some assembly language statements that allow programmers to describe higher levels of behaviors.

Many softwares have been written in assembly, and some even still exist in our days, but writing a complex software system in assembly is not easy. The solution for this problem was with the introduction of higher level programming languages like Basic, C, and others. With a high level language like C, programmers could describe software specifications at a higher level of abstraction, which in turn decreases the software complexity level by reducing the number of statements to write and manage.

In the 80s, graphical user interfaces (GUI) were first introduced by Xerox Labs, Palo Alto. While GUIs are user friendly and essential in our days, programming them is more complicated than console applications. For example in C, it would take several lines of code to display a window or some push buttons. With the rise of object oriented programming concept, GUI coding became easier. GUI functions or components such as opening a window, creating menus, etc., were considered as objects. These complex objects were as easy to use as simple type objects like arrays, vectors or others. This helped programmers build GUIs faster and reduced a lot the number of statements and the code that needs to be managed.

Later on, visual programming tools started to emerge. And as mentioned earlier in this chapter, these tools will allow non programmers to describe specifications captured in a higher level than writing codes. Such tools like GUI builders and UML tools are becoming familiar but still need some time to be standardized and matured.
2.4 Hardware development tools

Using transistors, capacitors and resistors, circuit synthesis can capture the most primary functions of all digital circuits which are ON and OFF. AND, OR and NOT are some basic logic functions that can be considered of a higher level than ON and OFF. Logic design theories are based on these three basic logical functions. With the increase of circuit complexity it is impossible for a designer to use transistors to represent his design. With AND, OR and NOT designers are capable of representing many more transistors than simple ON and OFF. Even with this evolution, a designer trying to design a non trivial circuit will end up using thousands of these logic gates and the circuit would become unmanageable. The obvious solution to this problem is the increase of the levels of abstraction by introducing functional units (FU). A functional unit is defined as follows: "a combinatorial or sequential logic circuit that realizes some Boolean function, such as an adder, a multiplier or an arithmetic logic unit (ALU)." [11]

High level synthesis also deals with binding basic functions: add, subtract, divide and multiply. These functions are equivalent to the basic operators found in software programming languages. The high level synthesis tool will take these functions and perform some tasks on them (scheduling and allocation) in order to get as an output the functional units.

Only one transistor can be constructed using primitive forms “ON and OFF”. Almost 10 transistors can be represented using AND, OR and NOT logic gates [30]. Using functional units and high level synthesis, designers can represent transistors in order of millions. With the continuous increase in circuits’ complexity, we will still face problems with the use of high level synthesis. The HDL code that acts as the input for the high level synthesis tool will greatly increase and become almost unmanageable.
Chapter 3

Synthesis and Layout Algorithms

In this section we present some of the algorithms implemented in this synthesis tool.
We will define the problem, present the solution and show our implementation
developed for our tool.

First, let us review the main functions of a synthesis tool in Figure 3.1

![Diagram of synthesis design flow]

Figure 3.1 (High level synthesis design flow)

One of the main goals in high level synthesis is the translation of the behavioral
specifications into an RTL description. In Figure 3.1 we can see that the system
specification is the first level of the design flow. This form of description is usually
given in plain English and consists of the abstract design representation. The
functional description of the design is given in the next behavioral description level. Behavioral specification is described in some HDL language like VHDL or Verilog. We are going to use a simplified form of the VHDL code as input to our synthesis tool (shown previously in Figure 2.1(a)). The simplified VHDL code will then be translated into a Data Flow Graph (DFG). "A DFG is expected to capture all the control and data flow information of the original simplified VHDL description while preserving the various dependencies." [2] This DFG will then undergo several refinements as it moves from one stage of high-level synthesis to another. Finally, the High Level synthesis tool will output a register transfer level (RTL) to the behavioral specification.

3.1 Data Flow Graph

The Data Flow Graph is a directed acyclic graph $G(V,E)$ where,

- the set of vertices $V = \{v_i; \ i=0,1,...,n\}$ correspond to set of operations,
- the set of edges $E = \{(v_i, v_j); \ i,j=0,1,...,n\}$ represent data dependencies between vertices.

As said earlier, the DFG is the graphical representation of the behavioral description given in the form of VHDL. So the first step in the generation of DFG is parsing the VHDL code. Actually, what we are doing in this step is building a compiler, similar to the high level language compiler, that takes the VHDL as input and parse it into a stream of tokens. The compiler will transform the source program to another representation. Each token consists of a sequence of characters that have an identifier, an operator and operands. This method is similar to the lexical analysis phase of a compiler methodology [1].
Figure 3.2 (parsing and transforming) shows a sample of the stream of tokens that will be used to construct the DFG graph.

![Diagram](image)

**Figure 3.2 (parsing and transforming)**

Each operator from the parsed tokens will be transformed into a node. The operand will be the input to this node and the result will be the output, example Figure 3.2. Every node is an object that has 4 members (*id, type, inputs* and *outputs*). The *id* member is the unique identifier of that node. In our tool the nodes will have *ids* in the format of \( V_n \). The *type* member is the type of the node (**,\(+\),\(-\), ...). The *inputs* member is a vector of all inputs to the node. These inputs could be either variables/ constants or expressions derived from the output of other nodes. The *outputs* member denotes the output element from the node which is the left part of the expression or the result of the operation.

As mentioned previously, there are 2 different types of inputs to the nodes: variables/ constants and expressions derived from the output of other nodes. The variables and constants are obtained from the header of our simplified VLSI input file. We will show in Figure 3.3 an example and then we will present the pseudo-code of the VHDL-DFG algorithm Figure 3.4.
Figure 3.3 (Simplified VHDL code)

<table>
<thead>
<tr>
<th>DFG Description of</th>
<th>: difeq.vhdl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>: uinport yinport aport dport xinport</td>
</tr>
<tr>
<td>Output</td>
<td>: uoutput youtput ooutput</td>
</tr>
<tr>
<td>InOut</td>
<td>: youtput</td>
</tr>
<tr>
<td>PreSet Variables</td>
<td>: yinport</td>
</tr>
<tr>
<td>PreSet Values</td>
<td>: yinport</td>
</tr>
<tr>
<td>Constants</td>
<td>: t3</td>
</tr>
<tr>
<td>Constant Values</td>
<td>: t3</td>
</tr>
<tr>
<td>Bit Width</td>
<td>: 4</td>
</tr>
</tbody>
</table>

---

\[ t1 := uinport + dport \]
\[ t2 := 3 + xinport \]
\[ t3 := 3 + yinport \]
\[ t4 := t1 + t2 \]
\[ t5 := dport * t3 \]
\[ t6 := uinport - t4 \]
\[ u_var := t6 - t5 \]
\[ y1 := u_var * dport \]
\[ v_var := yinport + y1 \]
\[ x_var := xinport + dport \]
\[ woutport := x_var \]
\[ youtport := v_var \]
\[ ooutput := u_var \]

Inputs to the nodes derived from the result of an other expression

---

Figure 3.4 (VHDL_to_DFG algorithm)

```c
Inputs: VHDL_file.
Outputs: Data Graph Model (DFG)

VHDL_to_DFG(File) {
    Graph g;

    // reading the header of the VHDL file
    1. while (not end of the header) {
        1. read values from Input and Constants fields (from the file) and insert them into the "inputs" vector.
    }

    2. add a parent node P to the graph. P has no input, and has as output the elements in vector "inputs".

    // reading the rest of the VHDL file
    3. while (not EOF) {
        1. parse expressions
        2. get tokens
        3. create node object for each operator
        4. set its inputs and outputs
        5. set types
        6. set names
        7. add it to the graph g
    }

    // linking nodes together in a way that respects dependencies
    4. for each node N_i in the graph g {
        1. get N_i inputs and check if they exist in the "inputs" vector.
        2. if they do, link the parent node P to this node N_i
    }

    5. for each node N_i in the graph g {
        for each node K_i in the graph g {
            1. check if outputs of K_i are inputs to N_i
            2. if they do, link N_i to K_i
        }
    }
}
```
The output of this algorithm is the DFG graph that must be drawn.

3.2 Scheduling

Scheduling is an important task in high level synthesis. It is the task of determining the instants at which the execution of the operations in the DFG will start. It takes the DFG graph (behavioral description) that contains the operations that must be performed by the hardware and partitions it into control steps. Each set of operations will be partitioned into a time step in which they will start execution.

After scheduling the DFG graph and partitioning it into time steps, each operation or node will be transformed into a functional unit (FU). In Chapter 4 we are going to see that nodes in the scheduled DFG graph or their corresponding functional units found in the same time step cannot be merged, unlike nodes in different time steps. This fact proves that if we scheduled more nodes into the same time step then the number of needed FUs will be bigger and the number of time steps will be smaller. On the other hand, if the number of scheduled nodes in a time step is smaller then the number of time steps needed to finish will be bigger and the number of needed FUs will be smaller. "Scheduling is an important task in high level Synthesis because it impacts the tradeoff between design cost and performance."[12]

The most commonly used scheduling algorithms are ASAP (As Soon As Possible) and ALAP (As Late As possible), and ASAP with resource constraint. Next we, explain these 3 algorithms.
3.2.1 ASAP (As Soon As Possible) scheduling

A simple method is to schedule operations "as soon as possible" (ASAP). ASAP is the simplest way to find a solution to a scheduling problem with precedence constraints.[11]

Emerald Facet system [28] from Carnegie Mellon University and CATREE system [10] from the University of Waterloo use this ASAP scheduling method.

Figure 3.5 shows an ASAP schedule for \textit{diffeq} DFG, generated by our synthesis tool. The pseudo-code for the ASAP algorithm is presented in Figure 3.6.

![ASAP Schedule Diagram](Figure 3.5)

Finding the ASAP schedule for an acyclic DFG graph is equivalent to finding the longest path in a directed acyclic graph. The algorithm starts by assigning the first node (the node that has no parents) to a time step $T_1$. All successors of this node will
then be assigned to increasing time steps. The rule to follow is that every parent node must be scheduled before its children. This algorithm computes the earliest time at which an operation can be executed. In other words, the result schedule will have the least number of time steps but it will not be optimized to resource constraints.

```plaintext
Inputs: Data Graph Model (DFG)  
Outputs: ASAP Scheduled DFG

ASAP_Schedule(Graph g) {
    // every node in the DFG graph has several members.  
    // one of them is the integer TS time stamp;  
    // another one is boolean sch;
    // initialize all nodes in g and schedule the parent node
    1. for each node N_i in the graph g {
        1.1. if (N_i is the parent node - has no inputs) {
            1.1.1. set TS of N_i to 0;
            1.1.2. set sch of N_i to true;
            1.1.3. } else {
            1.1.4. set TS of N_i to -1;
            1.1.5. set sch of N_i to false;
            1.1.6. }
        1.2. }
    // schedule all the un-scheduled nodes
    2. for each un-scheduled node N_i in the graph g {
        2.1. get TSs of N_i parents;
        2.2. set N_i TS to the MAX (of its parents' TSs + 1);
        2.3. set N_i sch to true;
        2.4. }
    3. sort all nodes in the graph g depending on their TS;
}
```

**Figure 3.6 (ASAP algorithm)**

The input to the algorithm is a DFG graph and the output is the same DFG graph but scheduled using the ASAP algorithm. The ASAP scheduled graph, like the DFG graph, will draw itself but this time the nodes will be positioned into different time levels depending on their time stamp (TS).

The ASAP scheduling can be used as part of more sophisticated algorithms such as those discussed next.
3.2.2 ALAP (As Late As Possible) scheduling

As said earlier, the ASAP scheduling computes the earliest time at which an operation can be executed. The ALAP scheduling does the exact opposite and computes the latest time at which an operation can be scheduled. Figure 3.7 shows an ALAP schedule for the *diffseq* DFG.

We implement the ALAP algorithm in the same way we did for ASAP except that the ALAP must start from the last node (the node that has no children) in the DFG and proceeds upwards.

In contrary to ASAP, the ALAP output schedule will be the slowest possible and will have the maximum number of time steps.

![Figure 3.7 (ALAP schedule of *diffseq*)](image)
The ALAP algorithm starts with the last nodes (nodes that have the largest time steps) in the ASAP schedule and assigns time steps in decreasing order as it proceeds backwards. Every node’s predecessor (parents of the node) will have a time step TS smaller by one.

Figure 3.8 shows the pseudo-code for the ALAP scheduling algorithm.

```
Inputs: ASAP Scheduled DFG graph
Outputs: ALAP Scheduled DFG graph

ALAP_Schedule(Graph G) {
    // every node in the DFG graph has several members.
    // one of them is the integer TS time step;
    // another one is boolean sch;
    1. lt = last time step in the ASAP graph;
    // initialize all nodes in the graph except the last nodes
    // in the ASAP graph, they must stay scheduled.
    2. for each node Ni in the graph G {
        1. if (Ni is not in the last time step - (last nodes)) { 
            2. set TS of Ni to -1;
            3. set sch of Ni to false;
            4. }
    3. }
    // schedule all the un-scheduled nodes
    4. for each un-scheduled node Ni in the graph G {
        1. get Ni that has no successors;
        2. set its TS to Last Time Step lt derived from ASAP;
        3. set Ni sch to true;
        4. get Ni whose successors have already been scheduled;
        5. set Ni TS to the MTD of its Children TSs - 1;
        6. set Ni sch to true;
    5. }
    6. sort all nodes in the graph G depending on their TS;
}
```

Figure 3.8 (ALAP Algorithm)

The input for the ALAP_Schedule algorithm is the output of the ASAP_Schedule algorithm. ALAP_Schedule takes a scheduled graph using ASAP and schedule it using ALAP. The algorithm starts by getting the last time step TS in the ASAP graph, and then unschedule each node in this graph, except the nodes that are in the last time step (nodes that have the largest TS). After that, we schedule the nodes that have no successors or children and set their TSs (time step) to the largest TS in the ASAP
graph. Finally, we schedule all unscheduled nodes and set their TS: \( TS(\text{node}) = \text{MIN}(TS(\text{children of the node}))-1 \).

The output is a scheduled DFG graph using ALAP algorithm.

The ASAP and the ALAP algorithms presented above are considered for unconstrained scheduling. When considering unconstrained scheduling, algorithms do not have to worry about satisfying constraints (resourced or time), they only have to satisfy sequencing relations.

### 3.2.3 ASAP Scheduling with Resource Constraint

Resource constraints scheduling is useful when the circuit design is restricted by silicon area, and where dedicated resources are not available.

The outcome schedule of this algorithm will eventually generate the best performance design while still satisfying the resource constraints.

The schedule is constructed in a step by step manner taking into consideration resource constraints and data dependencies of operations. Each time control will contain a number of operations that is not bigger than the number of FUs available.

In our application we chose to implement the ASAP with resource constraint algorithm. This algorithm works exactly the same way as the ASAP algorithm except it also has to satisfy the resource constraints given by the user.

Figure 3.9 shows an example of an ASAP schedule with resource constraints for the \text{diffeg} DFG with resource constraints: \((*) = 2, (+) = 1, (-) = 1\).
As said before, the "ASAP with resource constraints" algorithm works in the same way as the ASAP algorithm, with the addition of resource constraints. If we compare the ASAP graph in Figure 3.5 and the ASAP with constraints graph in Figure 3.9, we can see that the node v3 from the ASAP graph is moved down to time level 2 and this is because in level 1 the resources constraints are violated (only 2 (*) in a time level). Note also, that the node v5 from the ASAP graph is moved down to level 3 but this time not because of resource constraints violation but because of dependencies violation. Node v3 is a child for node v3, and the later has moved down to level 2 so node v5 must follow it and must be scheduled in the next level. When looking at Figure 3.9 we can notice that no more that 2(*) and 1(+) and 1(-) are scheduled into the same level time.
3.3 Initial Data Path (IDP)

In the next chapter we are going to present in detail the allocation and binding step, but for now we are only going to present the **Initial data path** algorithm. The **Initial data path** graph is generated from the scheduled DFG graph. Every node in the DFG graph will be transformed to a Functional Unit like an ALU and the input edges to this node will be transformed to registers. For example, node v1 in Figure 3.9 will be transformed to an ALU1 and will have 2 registers as input and one register as output. Figure 3.11 in page 28 shows the **Initial data path** generated from the ASAP DFG graph in Figure 3.5. We can see in this figure that many simplifications and a lot of merging could be done, but we do not do merging in the **Initial data path**. Merging algorithms will be presented in the next chapter.

Figure 3.10 shows the pseudo-code for the ALAP scheduling algorithm.

```
inputs: Scheduled DFG graph
outputs: Initial data path

Initial_data_path(Graph g) {
    // the graph g is the scheduled DFG graph.
    1. create the input ComponentModel inputs;
    2. create the output ComponentModel output;
    // looping over all the nodes Ni and creating ALUs end Registers components
    // and linking them together
    3. for each node Ni in the graph g {
        1. if (Ni is not the nop node (the parent node)) {
            2. Create an ALU componentModel A for Ni;
            3. for each parent of Ni {
                1. create a register componentModel R;
                2. link the output port of register R to the input port of ALU A;
                3. link the input port of register R to the output port of the Ni, parent;
            }
        } else if (Ni is in the last Time Step (has no outputs)) {
            1. create a register componentModel R;
            2. link the input port of register R to the output port of ALU A;
            3. link the output port of the register R to the input port of outputs;
        }
    }
}
```

*Figure 3.10 (Initial Data Path Algorithm)*
The input for the Initial Data Path algorithm is the scheduled DFG graph. The algorithm starts by creating the input and output components and then loops over all the nodes in the DFG graph, transforming these nodes to ALUs. The parent node nop will be transformed to the input component. All inputs for the ALUs will be transformed to Registers, and then these registers will be linked to the ALUs' parents. The nodes that have no children or outputs will be linked to Registers, and these registers will be linked to the output component.

The pseudo-code presented here for this algorithm is a part of the real implementation. The real implementation also contains the layout and routing algorithms. For the layout problem, we tried to keep it simple by putting every component model into a different column. As you can see in Figure 3.11 all ALUs and almost all Registers are positioned in different columns. A lot of improvement could have been done in the layout or placement problem by implementing some algorithms like constructive placement, iterative placement, and partitioning [26].

To link all components together without wires overlapping, we implemented the channel routing problem using the left-edge algorithm [11]. The algorithm is described next.
Figure 3.11 (Initial Datapath of ASAP DFG diff eq)
3.4 Channel Routing

"Channel Routing occurs as a natural problem in standard cell and building block layout styles, but also in the design of printed circuit boards (PCBs)" [11]. As said earlier, we are using channel routing to route and link components (ALUs, Registers...) without wires overlapping. We are going to implement the channel routing problem using left-edge algorithm defined by Sabih H.Gerez in his book "Algorithms for VLSI Design" and then rotate the outcome by 90 degrees clock wise to obtain a result as the one shown in Figure 3.11.

Channel routing consists of routing nets across a rectangular channel, as in Figure 3.12.

![Figure 3.12 (channel routing problem)](image)

In this figure we want to route all terminals that have the same number. Notice that we needed 3 rows to route all terminals without wire overlapping.

The “classical” model for channel routing is as follows:

- All lines run along orthogonal grid lines with uniform separation
- There are two wiring layers
- Horizontal segments are put on one layer and vertical segments on another
- For each net (or route), the wiring is realized by one horizontal segment with two vertical segments connecting it to all terminals on the net.
If horizontal segments belonging to different nets are put into the same layer or row, the segments should not overlap; otherwise we will not be solving the overlapping problem. For example, in Figure 3.12, net 1, 5 and 2 share the same layer but do not overlap. This restriction is called *horizontal constraint*.

Hashimoto and Stevens introduced the *left-edge algorithm* that solves the channel routing problem with horizontal constraint optimally, except with the existence of vertical constraints [11].

A net $i$ in a channel routing problem without vertical constraints can be characterized by an interval $[x_{\text{min}}, x_{\text{max}}]$, corresponding to the left-most and right-most terminal positions on the net. For example, in Figure 3.12, we can represent the nets by the following intervals: $i_1 = [1,4]$, $i_2 = [12,15]$, $i_3 = [7,13]$, $i_4 = [3,8]$, $i_5 = [5,10]$, $i_6 = [2,6]$, $i_7 = [9,14]$ where $i$ is the net, $i_j$ is the net for terminals “$j$”, and the interval $[1,4]$ is the position of the horizontal segment. By representing nets into intervals, we simplified the problem and now the algorithm has to assign a row position in the channel for each interval.

Our first step in the *left-edge algorithm* is to create these intervals and store them in a list $\text{netList}$ in order of their increasing left coordinates. For this, we created an object $\text{net}$ that contains the coordinates (the left and right coordinates) of every net or route in our problem. Then, we added this $\text{net}$ object to a list or vector ($\text{netList}$), ordering them in increasing left coordinates. This $\text{netList}$ that contains all the intervals or $\text{nets}$, will be the input to our *left-edge algorithm*.

The code for this algorithm is found in Figure 3.13.
```java
public void leftEdgeAlgorithm(Vector netlist) {
    int track = 0;
    boolean done = false;
    while (!done) {
        track++;
        done = true;
        // place unrouted nets in current track if they fit
        for (Enumeration e = netlist.elements(); e.hasMoreElements(); ) {
            Net n = (Net)e.nextElement();
            if (n.isRouted()) continue; // skip nets already done
            if (canRoute(n, track)) {
                n.setTrack(track);
            } else done = false; // at least one net (this one) remains unrouted!
        }
    }
}
```

Figure 3.13 (left-edge algorithm)

This algorithm consists of two main loops, an outer and an inner loop. The outer loop will stop only when all nets or intervals are positioned into tracks or rows. The inner loop will go over the `netList` vector and check if the current net can be placed on the first track. If not, it will be placed on the next track. The `canRoute` method determines, by scanning all other nets, whether we can assign net `n` to track `track`. It checks whether net `n` overlaps with other nets by checking the coordinates.

This algorithm solved our routing problem in Figure 3.12.

As mentioned earlier, we implemented the left-edge algorithm to solve the problem for routing components, and then we rotated the outcome by 90 degree clock wise.

Figure 3.14 shows another example of data path where we used channel routing to route components.
In Figure 3.14 we applied the channel routing between the Reg column and the Mux one, and between the Mux and Alu. If a wire must exist between Alu and Reg, then we cannot use channel routing. We simply let the wires pass from the bottom of the circuit.
Chapter 4

Allocation/Binding

Allocation is the last step in our synthesis tool. Allocation maps each operation in the scheduled DFG to a specific functional unit on which the operation will be executed [11]. It is also concerned with mapping storage values to registers and data transfers to interconnections and multiplexers.

Our data path allocation starts with a scheduled data flow description of a circuit and uses an iterative improvement optimization method. The scheduling can be accomplished using any method reported previously in chapter 3. In what follows, we describe our allocation method.

4.1 Structure

Given a scheduled data flow graph (DFG), a node corresponds to

1. An operator that must be assigned to a functional unit during the control step in which it is scheduled;

2. A value that must be assigned to a register for the duration of its life time.
   Thus, overlapping life times cannot be assigned to the same register.

3. Finally, data transfers are assigned to some path of connections, buses and multiplexers.
Consider a DFG node associated with a variable instance \( V \), its corresponding operation \( O(V) \), and life span \( L(V) \). Then the TFB that corresponds to \( V \) is the 3-tuple:

\[
TFB_v = [V; O(V); L(V)]
\]

For the example shown in Figure 4.4, the TFB of \( v_3 \) is: \([v_3; \text{Addition}; \{2,4\}]\). It follows from this definition that each DFG node represents a TFB whose input edges are connected to the outputs of other TFBs. A more complex TFB is generated by merging two or more TFBs. The objective of our scheme is to allow the mapping of DFG nodes into TFBs which are the building blocks of the data path generated by the proposed allocation.

Two TFBs are compatible if there is no resource conflict between the operations of the DFG nodes; that is, the nodes are assigned to different time steps in the schedule. However, in addition to the above restriction, there are two additional rules that should be satisfied for the successful merging of two TFBs:

- **Rule 1**: Do not merge the TFBs if such merging will result in a module that does not exist in the technology library in Figure 2.2. For example, if the resulting ALU has an operation set such as \(+,\times,\div\) that is not in the library, such merging cannot take place from the technology library standpoint.

- **Rule 2**: Allow self-adjacent registers only if the resulting structure is testable.

Thus, we would not generate designs such as the ones depicted in Figure 4.3(c), while designs as in Figure 4.3(b) are testable and thus acceptable.
4.2 Module Allocation Graph

In order to illustrate the compatibility relations among the DFG nodes, we use a special graph which we call module allocation graph (MAG). This is a directed levelized graph with its nodes corresponding to ones in the DFG operations and levels to the DFG schedule. An edge from node \( A \) to node \( B \) in the MAG indicates that both nodes are compatible; however, node \( A \) is scheduled before node \( B \). A given path in the MAG corresponds to a list of compatible TFBs that can share resources. Clearly, nodes at the same level are not compatible.

The motivation for the MAG is twofold. First, by having directed edges, a top-bottom optimization process is possible by considering two levels at a time, pruning the number of paths in the MAG; thus, reducing the number of merging possibilities. Second, we associate a local cost function with every path in the MAG. The cost function will guide the algorithm in order to select the best merging possibility at every iteration. The MAG is technology dependent and driven by the system library or the xml library discussed previously. Thus, for different libraries, we have different MAGs, and consequently different bindings due to Rule 1, discussed earlier.

The construction of the MAG is quite simple: we assign each node in the MAG to the level at which it was scheduled. We add directed edges between the compatible nodes using a top-bottom fashion by considering two levels at a time, \( k \) and \( k+1 \). The reason is to prune the number of paths in the MAG and thus reduce the number of edges in the graph. For example, in Figure 4.5, path \( v1/v6 \) (corresponding to merging operations \( v1 \) and \( v6 \)) is redundant since it is covered by path \( v1/v3/v6 \). Edges are added to the MAG as long as they do not cause resource conflicts. This basically means that the MAG contains the schedule edges (excluding the redundant ones) in addition to any edges that indicate no resource conflicts. The module allocation graph for the DFG in
Figure 4.4 is shown in Figure 4.5 where dotted edges correspond to incompatibilities due to library considerations.

We assign to each compatible edge in the module allocation graph a gain and a loss value. The purpose is to estimate the local gain of the datapath resulting from merging two TFBs while minimizing the loss. Next, at every iteration, two TFBs are selected for merger based on a set of heuristics that are described next.

### 4.3 Gain and Loss in TFBs Merging

Each TFBs merger corresponds to a move in the design space from point $S_1$ to $S_2$, where $S_1$ and $S_2$ are the states of the data path before and after the merge. We use the gain as a measure of the reduction in the datapath cost as a result from merging TFB$_A$ with TFB$_B$. We define the gain, $Gain(TFB_{AB})$, as the difference between the cost of the two datapath states: $Gain(TFB_{AB}) = Cost(S_1) - Cost(S_2)$. Taking into consideration the cost formulation we have:

$$Gain(TFB_{AB}) = Cost(TFB_A) + Cost(TFB_B) - Cost(TFB_{AB})$$

where $Cost(TFB_{AB})$ is the cost of the merged TFB$_{AB}$ and $Cost(TFB_A)$, $Cost(TFB_B)$, are the costs of TFB$_A$ and TFB$_B$ respectively.

The loss in datapath cost gives an indication of how the merge will affect future merges. The loss is more global than the gain because it takes into consideration the neighboring nodes. We define the loss, $Loss(TFB_A, TFB_B)$, as the sum of the gains of all edges that were removed as a result of merging TFB$_A$ and TFB$_B$, except the gain of the edge TFB$_{AB}$. The loss can be calculated as follows:

$$Loss(TFB_{AB}) = Gain_{out} + Gain_{in} - 2 * Gain(A, B)$$

where $Gain_{out}$ is the sum of Gains of all edges going out of TFB$_A$ and $Gain_{in}$ is the sum of Gains of all edges going into TFB$_B$. 
4.4 Cost function

As explained earlier, a TFB consists of a variable instance $V$, its corresponding operation $O(V)$, and life span $L(V)$. Thus, the gain and loss values consist of the following:

\[ Gain_{local}(C) = gain_{ALU} + gain_{Max} + gain_{Reg} \]
\[ Loss_{global}(C) = Loss_{ALU} + Loss_{Max} + Loss_{Reg} \]

where the individuals' gain and loss are based on the above gain and loss concepts and will be next described in detail.

4.5 Allocation Algorithm

The merging or optimization algorithm is illustrated in Figure 4.1. Briefly, we start with the nodes in the first two levels of the levelized compatibility graph (MAG) and find the level with the least outgoing edges. Obviously, on this level, the nodes with the least number of outgoing edges are more restricted than the others. Among these restricted nodes, we choose the nodes with a minimum loss. The local gain functions $g_{local}$ can be used to break any ties. We thus create at every merging step a more complex TFB. After exhausting all the nodes at the level into consideration, we proceed to the following one and update the local cost functions so as to reflect the new structure in the data path. We repeat this process until all nodes in the MAG have been processed. The complexity of the allocation algorithm, described in Figure 4.1, is in the worst case, of the order $O(n^2)$ where $n$ is the number of nodes in the DFG.

The allocation method is based on an incremental algorithm. The algorithm maps DFG nodes onto individual TFBs by constructing an initial design point: Initial Datapath Structure (IDP). The IDP is an initial design point that will be iteratively
improved in order to find the best design and test point subject to the initial constraints. The initial data path for the biquad example of Figure 4.4 is shown in Figure 4.6.

Each positive gain we obtain from the module allocation graph has the effect of reducing the data path cost by the amount of that gain. The objective is to maximize the datapath gain while minimizing the negative effect on future merges. There are two negative effects that are likely to occur:

1. One or more edges are removed, and the gain associated with those edges can no longer be obtained by the future merges.
2. The number of possible future merges is reduced as a result of removing some edges, and thus removing node compatibilities.

If \( \text{Out}(i, i + 1) \) denote the number of nodes in level \( i \) which have outgoing edges to nodes in level \( i + 1 \) and \( \text{In}(i, i + 1) \) denote the number of nodes at level \( i + 1 \) which have incoming edges from nodes at level \( i \), then by merging levels \( i \) and \( i + 1 \), the maximum number of merges is upper bounded by \( n \), where \( n = \min(\text{Out}(i, i + 1), \text{In}(i, i + 1)) \). We present next three heuristics that will guide our algorithm in selecting the pair of nodes that, when the corresponding TFBs are merged, will have as small negative effect as possible on future merges.

**Heuristic 1 — Priority level:** This heuristic is used to determine if merging the most restricted nodes in one level, will have an effect on increasing the number of future merges. Thus, choose the priority level when merging two levels \( i, i + 1 \) as follows:

1. If \( \text{Out}(i, i + 1) < \text{In}(i, i + 1) \) then the merging priority is given to the nodes at level \( i \)
2. If $Out(i, i + 1) > In(i, i + 1)$: merging priority given to the nodes at level $i + 1$.
3. If $Out(i, i + 1) = In(i, i + 1)$: we give the nodes in both levels the same priority.

- **Heuristic 2 — Merging Order:** This heuristic aims to reduce the first negative effect. Restricted nodes do not have the flexibility of merging with many other nodes. Thus, by merging those nodes first, we increase the possibility of maximizing the number of future merges. Choose the merging order for the nodes in the priority level in descending order of their restrictions, i.e., the most restricted nodes are merged first. Restriction of a node $A$ is measured by:
  
  1. the number of outgoing edges if $TFB_A$ is in level $i$.
  2. the number of incoming edges if $TFB_A$ is in level $i + 1$.

Clearly, the *most restricted node* is the one with the minimum number of incoming or outgoing edges. Nodes that are not in the priority level are considered least restricted.

- **Heuristic 3 — Edge Selection:** This heuristic aims to reduce the second negative effect, as edges with minimal loss are merged first. Choose the edge to be merged, as follows. Out of the edges that involve one or more of the most restricted nodes, in both levels, and choose the edges with minimum loss. If there are ties, then among the edges with equal loss choose the edge with maximum gain (any further ties could be broken arbitrarily).

These heuristics are taken from a previous work for Dr Haidar Harmanani.
4.5.1 Module Allocation

Every time we merge two ALUs that are associated with two different TFBs, we are reducing the data path cost by the cost of one ALU. Assuming that the combined operation set, $ALU_1 \cup ALU_2$, already exists in the library, we define the local gain function as:

$$g_{ALL} = \text{Cost}(ALU_1) + \text{Cost}(ALU_2) - \text{Cost}(ALU_1 \cup ALU_2)$$

where $\text{Cost}(ALU_1), \text{Cost}(ALU_2)$ and $\text{Cost}(ALU_1 \cup ALU_2)$ are provided by the library.

4.5.2 Mux Allocation

When mapping a commutative operation to a TFB, there are two possible configurations to assign the input ports to the TFB, left or right mux. For non-commutative operations such as subtraction, the configuration is unique. In order to obtain a good MUX cost estimation, we use incremental operands alignment. We explain the method next.

When considering two TFBs for merging, we assign the non-commutative operations to the multiplexers at the input ports of the resulting TFB first. The reason is that these signals cannot be swapped in order to explore sharing possibilities with other signals. The remaining assignment is done so as to reduce the number of multiplexer inputs, right or left, through register alignment. If $\text{Cost}(MUX_1)$ and $\text{Cost}(MUX_2)$ are the multiplexers cost of the original TFBs and $\text{Cost}(MUX)$ is the multiplexer cost of the resulting one provided by the library, then the local gain function is defined as:

$$g_{MUX} = \text{Cost}(MUX_1) + \text{Cost}(MUX_2) - \text{Cost}(MUX).$$
Input: Module Allocation Graph (MAG) and an Initial Data Path (IDP).  
Output: Optimized Self-Testable RTL Structure.

\[ \begin{align*} 
& i = 1 \\
& \text{while } (i < \text{number of levels in the MAG}) \text{ do } \{ \\
& \quad \text{Repeat } \{ \\
& \qquad 1. \text{ Compute the cost functions for all edges between levels } i \text{ and } i+1. \\
& \qquad 2. \text{ Do not take into consideration the edges between level } i \text{ and } i+1 \text{ that we cannot merge based on the library restriction (Rule 1).} \\
& \qquad 3. \text{ Choose the level that has the least number outgoing edges. If both levels have the same number of outgoing edges, choose either one arbitrarily. (Heuristic 1).} \\
& \qquad 4. \text{ Of the chosen level in the above step, select the nodes with the minimum number of incoming or outgoing edges. (Heuristic 2).} \\
& \qquad 5. \text{ Of the selected nodes, choose the nodes with minimum loss and merge the corresponding TFBs in the IDP. (Heuristic 3).} \\
& \quad \} \text{ Until all nodes at levels } i \text{ and } i+1 \text{ have been processed.} \\
& \text{Move remaining nodes to level } i+1 \\
& \text{Increment } i \\
\end{align*} \]

Figure 4.1 (Datapath Allocation Algorithm)

\textbf{Input:} TFBs A and B  
\textbf{Output:} New TFB C

- Construct the new ALU operation set such that:  
  \[ \text{Oper}(C) = \text{Oper}(A) \cup \text{Oper}(B). \]

- Construct two MUXes on top of TFB C. Assign the input registers of TFBs A and B to the new MUXes. Try to share common signals by swapping signals which belong to Commutative operation.

- Construct a list of the register set of TFBs A and B. Apply the Left Edge Algorithm and merge the registers which 1) don't have overlapped life spans and 2) do not create 'bad' self-adjacent registers (Figure 5-3(c)). Next, maintain the connections of these registers to the other TFBs.

Figure 4.2 (Merging Two TFBs)
Figure 4.3: (a) Testable Functional Block, (b) Self Testable ALU with Self-Adjacency, (c) Non-Observable ALU due to Self-Adjacency

Figure 4.4 (Biquad filter DFG)
Figure 4.5 (Biquad Module Allocation graph)

Figure 4.6 (Initial data path (IDP) for the biquad example)
4.6 Experimental results

We implemented the proposed tool using Java on a Pentium 2.4 GHz with 512 MB of RAM running Windows. The parsing, scheduling and allocation methods were very fast and all reported results we produced in at most two CPU minutes including all synthesis steps, drawing and graphs generation. We ran some examples that include biquad, differential equation diffeq, polynomial design poly_design and the elliptic wave filter.

For every example, in the following tables, we show the number of clock cycle, the number of functional units, the number of registers, the number of multiplexers, and the number of multiplexers input, library constraints, and the used scheduling method.
### Table 4.1 Biquad

<table>
<thead>
<tr>
<th>Clock Cycle</th>
<th>ALUs</th>
<th>REGs</th>
<th>MUX</th>
<th>MUX In</th>
<th>Library constraints</th>
<th>Scheduling constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4(*) 2(+)</td>
<td>13</td>
<td>1</td>
<td>2</td>
<td>(+), (*)</td>
<td>ASAP</td>
</tr>
<tr>
<td>4</td>
<td>2(<em>) 2(</em>+)</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>(+), (<em>), (+</em>)</td>
<td>ASAP</td>
</tr>
<tr>
<td>5</td>
<td>2(*) 1(+)</td>
<td>7</td>
<td>1</td>
<td>3</td>
<td>(+), (*)</td>
<td>1(+), 2(*)</td>
</tr>
<tr>
<td>7</td>
<td>1(*) 2(+)</td>
<td>7</td>
<td>3</td>
<td>6</td>
<td>(+), (*)</td>
<td>1(+), 1(*)</td>
</tr>
<tr>
<td>7</td>
<td>1(<em>) 1(</em>+)</td>
<td>6</td>
<td>2</td>
<td>7</td>
<td>(+), (<em>), (+</em>)</td>
<td>1(+), 1(*)</td>
</tr>
</tbody>
</table>

### Table 4.2 Diffreq

<table>
<thead>
<tr>
<th>Clock Cycle</th>
<th>ALUs</th>
<th>REGs</th>
<th>MUX</th>
<th>MUX In</th>
<th>Library constraints</th>
<th>Scheduling constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>4(*) 2(+) 1(-)</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>(+), (*)</td>
<td>ASAP</td>
</tr>
<tr>
<td>6</td>
<td>3(*) 1(++) 1(+) 1(-)</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>(+), (<em>), (+</em>)</td>
<td>ASAP</td>
</tr>
<tr>
<td>6</td>
<td>2(<em>) 2(+) 1(</em>)-</td>
<td>13</td>
<td>2</td>
<td>5</td>
<td>(+), (<em>), (-</em>)</td>
<td>ASAP</td>
</tr>
<tr>
<td>8</td>
<td>2(*) 2(+/-) 1(+)</td>
<td>12</td>
<td>2</td>
<td>5</td>
<td>(+), (*)</td>
<td>1(+), 1(+), 1(-)</td>
</tr>
<tr>
<td>8</td>
<td>1(<em>) 1(+) 1(-) 1(</em>)+</td>
<td>9</td>
<td>2</td>
<td>5</td>
<td>(+), (<em>), (+</em>)</td>
<td>1(+), 1(+), 1(-)</td>
</tr>
<tr>
<td>8</td>
<td>1(<em>) 2(+) 1(</em>)-</td>
<td>14</td>
<td>2</td>
<td>11</td>
<td>(+), (<em>), (-</em>)</td>
<td>1(+), 1(+), 1(-)</td>
</tr>
</tbody>
</table>

### Table 4.3 Poly_design

<table>
<thead>
<tr>
<th>Clock Cycle</th>
<th>ALUs</th>
<th>REGs</th>
<th>MUX</th>
<th>MUX In</th>
<th>Library constraints</th>
<th>Scheduling constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4(*) 3(+)</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>(+), (*)</td>
<td>ASAP</td>
</tr>
<tr>
<td>4</td>
<td>1(<em>) 2(</em>+)</td>
<td>7</td>
<td>2</td>
<td>4</td>
<td>(+), (<em>), (+</em>)</td>
<td>ASAP</td>
</tr>
<tr>
<td>4</td>
<td>2(*) 1(+)</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>(+), (*)</td>
<td>2(+), 2(*)</td>
</tr>
<tr>
<td>4</td>
<td>2(<em>) 1(</em>+)</td>
<td>7</td>
<td>2</td>
<td>5</td>
<td>(+), (<em>), (+</em>)</td>
<td>2(+), 2(*)</td>
</tr>
<tr>
<td>6</td>
<td>1(*) 2(+)</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>(+), (*)</td>
<td>1(+), 1(*)</td>
</tr>
<tr>
<td>6</td>
<td>1(<em>) 1(</em>+)</td>
<td>6</td>
<td>2</td>
<td>6</td>
<td>(+), (<em>), (+</em>)</td>
<td>1(+), 1(*)</td>
</tr>
</tbody>
</table>

### Table 4.4 Elliptic

<table>
<thead>
<tr>
<th>Clock Cycle</th>
<th>ALUs</th>
<th>REGs</th>
<th>MUX</th>
<th>MUX In</th>
<th>Library constraints</th>
<th>Scheduling constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>7(*) 11(+)</td>
<td>34</td>
<td>5</td>
<td>14</td>
<td>(+), (*)</td>
<td>ASAP</td>
</tr>
<tr>
<td>14</td>
<td>3(<em>) 6(+) 3(</em>)+</td>
<td>26</td>
<td>10</td>
<td>29</td>
<td>(+), (<em>), (+</em>)</td>
<td>ASAP</td>
</tr>
<tr>
<td>16</td>
<td>6(*) 8(+)</td>
<td>27</td>
<td>10</td>
<td>26</td>
<td>(+), (*)</td>
<td>2(+), 2(*)</td>
</tr>
<tr>
<td>16</td>
<td>1(<em>) 6(+) 2(</em>)+</td>
<td>24</td>
<td>6</td>
<td>26</td>
<td>(+), (<em>), (+</em>)</td>
<td>2(+), 2(*)</td>
</tr>
<tr>
<td>27</td>
<td>5(*) 2(+)</td>
<td>23</td>
<td>4</td>
<td>23</td>
<td>(+), (*)</td>
<td>1(+), 1(*)</td>
</tr>
<tr>
<td>27</td>
<td>4(<em>) 2(</em>+)</td>
<td>21</td>
<td>6</td>
<td>32</td>
<td>(+), (<em>), (+</em>)</td>
<td>1(+), 1(*)</td>
</tr>
</tbody>
</table>
Chapter 5

Conclusion

We have presented and discussed a computer-aid design (CAD) tool for the successful design of a circuit with a large complexity. The proposed tool synthesizes and captures circuit specifications at a higher level of abstraction, which will reduce the system complexity and shorten the delivery time.

The code is structured in a way that it can be extended without difficulty and new features and algorithms can be easily added.

This tool takes a behavioral description in a simplified form of VHDL and generates the corresponding datapath. The system first starts by parsing and compiling the VHDL input into a Data Flow Graph (DFG) which is the graphical representation of the behavioral description. Next, the DFG will be scheduled using several schedule algorithms in order to maximize operations concurrency. We implemented in our system the ASAP, ALAP, and ASAP with resource constraints scheduling algorithms.

After scheduling the DFG, the allocation task will transform operations to functional units, values to registers and finally, multiplexers will connect all components together. We also presented the allocation method which is based on the Module allocation graph (MAG) that is generated from the DFG. From the MAG we derived the Initial Data Path (IDP) and then we introduced a cost function. This cost function is then used in the heuristics we found to simplify and merge functional units in the initial data path. Finally, the output will be the optimized data path for the input VHDL code. In addition, after generating the datapath the user can interact with it to redesign the components and experiment more tradeoffs.
We implemented the tool using Java and Swing as the graphical user interface (GUI) platform. The GUI provided an efficient way to let the user interact with the system, and sees what is going on in every synthesis step. The scheduled DFG graph and the allocated datapath can be represented graphically. Also, we have presented some placement algorithms to place the nodes in the DFG graph and to place components in the datapath. In addition, we presented a routing algorithm to route all components together in the datapath. This problem is called channel routing problem and we solved it using left-edge algorithm.

The most challenging issues we faced were the scheduling and allocation algorithms implementation, in addition to the drawing of DFGs and Datapaths. We didn’t only want to implement the scheduling and allocation algorithms but we also needed to illustrate these algorithms in a neat way that can be understood by users. Also, these drawings are not simple drawings or pictures. Every component or node in these drawings is considered as a movable object which the user can move and change some of its properties. In brief, we wanted the drawing not to be rigid datapaths and schedules but rather dynamic and interactive.

Our synthesis tool is not a fully functional system, it can be more considered as a prototype. A lot of basic features in this tool can be extended and improved in a number of ways. For example, more scheduling algorithms like Force directed scheduling, Iterative rescheduling, List based scheduling [12], and others could be implemented and plugged into this tool.

Also, the placement problem in the datapath could be solved by implementing algorithms like constructive placement, iterative placement, and partitioning [11]. In addition, we can also add a simulation function to this synthesis tool. The simulation task will help user identify any errors in the design.
Appendix A

GUI functionalities and screenshots

In this appendix we discuss the high level language and the development environment we used in writing our synthesis tool. Then, we present a preview of the GUI using screenshots.

A.1 Considering Java

We want to develop a High-level Synthesis tool that can be expanded by others to include more features. For this, we had to think carefully about which high level language we will be using in developing such an application and how to structure our code to be easily readable and understandable by other programmers aiming to continue or expand this tool. We have decided on Java for several reasons and we will state some of them.

Java is a modern, object-oriented language based on public standards. In our case, building a tool of around 200 classes and 25,000 lines of code can never be done and managed without a good object-oriented design. In other words, objects can give your program a degree of modularity which makes it easier to understand, manage and maintain. It is possible to provide programmers with a consistent, public API (Application Programming Interface) and abstract classes to be extended while keeping the implementation details private.

Java is a standardized language and has a rich collection of core functions. Most developers are aware that Java applications can execute with little or no change on
multiple hardware platforms. We wanted our synthesis tool to be platform independent.

Also, there are excellent, free Java development tools such as the Eclipse IDE (Integrated Development Environment), NetBeans or JBuilder. When building "not a small" software program, these IDEs are useful to manage and debug your code. Finally, there is a huge community of Java programmers, many of whom make their work available in open source software. By using these open source software we will be saving a lot of precious development time by not reinventing the wheel.

A.2 GUI screenshots

In this section, we present some GUI screenshots of the synthesis tool and comment briefly on each one.

![Figure A.1 (main window)](image-url)
We used a JDesktopPane as our main window, it is considered as a work desktop where users can open and work on more than one window.

### A.2.1 Menu toolbar

![Interactive High Level Synthesis tool](image)

Figure A.2 (File menu)

The File menu item contains several actions:

- **New**: opens a new visual Data Path work sheet where users can add components and link them together.
- **Open**: opens a file chooser dialog to choose an RTL file and opens it as a visual Data Path in a work sheet where it can be edited.
- **Save .rtl & .attribute**: saves the edited visual Data Path and the Attributes of each component in an .rtl and .attribute respectively (textual format).
- **Save as ISCAS**: saves the Data Path as ISCAS format.
- **Export**: exports the Data Path as picture to .PDF or .GIF.
- **Image Editor Viewer**: opens a simple image editor to edit the Data Path as image before saving it.
- **Exit**: exit the application.
The Scheduling menu item contains several actions:

- **Open ASAP sch**: This will open a file chooser dialog box that allows users to choose the modified VHDL input code for our synthesis tool (.dfg file). The HDL code will be transformed into a DFG graph that will be scheduled by the As Soon As Possible (ASAP) algorithm.

- **Open ALAP sch**: Similar to the above, but the DFG graph will be scheduled using the As Late As Possible (ALAP) algorithm.

- **Open ASAP sch**: Similar to the above, but the DFG graph will be scheduled using ASAP with constraints.

- **Open ASAP Compatibility**: Draw an ASAP compatibility graph from the HDL input file.

- **Open ALAP Compatibility**: Same as above with ALAP algorithm.

- **Open ASAP Compatibility + constraint**: same as above with ASAP and constraints algorithm.
The Options menu item contains 4 actions:

- **Change Background Color**: this will allow users to change the background color of the Data Path work sheet.
- **ToolTips**: a checkbox that allows users to choose between allowing tooltips to appear on components and wires or not.
- **Wires from Top and Wires From bottom**: a group of two radio buttons that allows users to choose between drawing wire in the Data Path from the top or bottom.

This menu item (Look and Feel) contains some look and feels that users can choose from to change the look of the application.
A.2.2 Main functions screenshots

We will show now some screenshots of the main functions in this synthesis tool.

1) File > New:

![Interactive High Level Synthesis tool](image)

Figure A.6 (File > New, new work sheet)

This is the work sheet for the Data Path design. Users can drag and drop components from the left tool bar to the work sheet.
Figure A.7 (dragging a component into the drawing window)

This picture shows how to drag and drop the components into the work sheet.
Figure A.8 (adding a wire between 2 components)

This screenshot shows how to add a wire between the two components by selecting the source port and dragging the mouse pointer to the destination port.
By clicking on a component, a property table appears on the left. This property table allows users to edit and save properties to each component (properties like function, delay, area...). Each type of components (like ALUs or MUXs) has its own property table.
2) File > Open

![File chooser dialog](image)

**Figure A.10 (file chooser dialog)**

A file chooser dialog that allows users to choose .rtl and .attribute files. The rtl file contains a textual description about the data path (components and connections between them). The attribute file contains the properties and attributes of the components found in the rtl file.
The data path of the opened .rtl file can be edited and properties of each component can be changed.
3) File > Image Editor Viewer

Figure A.12 (Simple Image Editor)

This is a simple image editor that allows users to manipulate the data path image before saving it.
4) Scheduling > Open ASAP sch

![Please choose .dfg file dialog](image)

**Figure A.13 (file chooser dialog)**

File chooser dialog allows users to select a simplified VHDL input code that will be compiled into a DFG graph that will be scheduled using ASAP algorithm.
This is the scheduled DFG graph using ASAP. The toolbar of this window has server buttons. Starting from left to right:

- The first 3 buttons are zooming buttons.
- The second 2 buttons allow users to export to gif or pdf files.
- Generate initial data path graph button allows users to generate the initial data path graph form the DFG scheduled graph.
- Choose Component Library button allows user to choose the library file (xml format).
- MAG button allows users to generate the Module Allocation Graph.
Figure A.15 (initial data path)

This is the initial data path work sheet generated from the ASAP DFG graph.
The Modular Allocation Graph generated by the ASAP DFG graph is seen above.

In this section we presented screenshots of some of the features from our synthesis tool but not all of them.
Bibliography


