Synthesis with VHDL

by

Hatem Halawi

Submitted in partial fulfillment of the requirements
for the degree of Master of Science

Thesis Advisor: Dr. Haidar Harmanani

School of Arts and Sciences
LEBANESE AMERICAN UNIVERSITY
July 2001

LEBANESE AMERICAN UNIVERSITY

GRADUATE STUDIES

We hereby approve the thesis of

Hatem Halawi

Candidate for the Master of Science degree.

(signed) [redacted]
(Advisor: Dr. Haidar Harmanani)

(signed) [redacted]
(Co-advisor: Dr. Walid Keirouz)

(signed) [redacted]
(Co-advisor: Dr. Ramzi Haraty)

Date 17/7/2001

We also certify that written approval has been obtained for any propriety material contained therein.

[redacted]

Abstract

The need of high-level behavioral languages that make it easier to programmers to design hardware raised the issue of high-level synthesis. High-level synthesis is concerned with the design and implementation of circuits from behavioral description of some high-level languages that contain a set of goals and constraints.

Synthesis is defined as the translation of a behavioral description into a structural one. Doing this requires a synthesis tool that helps to get a good and efficient output design from a behavioral description.

A synthesis tool that takes a behavioral description and outputs a schedule is presented in this thesis. The synthesis tool is made of many two main components that also made of smaller ones. The first component is the translator that translates a behavioral code into an intermediate form that will be the input of the second component. The second component is the scheduler. The scheduler takes objects (nodes) and schedules them using some scheduling algorithms that are presented in the thesis.

To my parents

Acknowledgements

I would like to take opportunity to thank people who helped in making this thesis possible.
I would like to thank my advisor Dr. Haidar Harmanani for his continuous help all the times and with every thing possible through out my M.S. studies.
Thanks to the committee members Dr. Walid keirouz and Dr. Ramzi Haraty.
I would like to thank all those who indirectly helped me in writing my thesis and implementing my project.
Finally I would like to thank my family and friends for their long support.

Table Of Contents

Chapter 1 Introduction

1.1 Introduction

High-level synthesis is concerned with the design and implementation of circuits from behavioral description subject to a set of goals and constraints. High-level synthesis systems accept a behavioral specification and produce a register-transfer level design. The synthesis system generates an RTL description of a data-path and a controller from an algorithmic description that defines the precise procedure for the computer solution of a problem [1].

For a given behavioral specification, enormously many different register-levels can be produced. Instead of producing all and selecting the best one, synthesis systems use estimation in earlier design stages to prune the designs that are inferior or violate a given constraint [2].

In order to alleviate the design complexity, many commercial and educational tools for synthesis have emerged in the past decade. Recently, high-level synthesis tools with high level test considerations have emerged as well [1].

1.2 Definition of the problem

As have been mentioned in the introductory material, the main point behind synthesis is to find a better and more efficient design. To do so, synthesis tools are highly recommended to analyze the behavioral description of the design, find unnecessary items in the design, take advantage of all available resources than can parallelize and hence increase performance and efficiency, and finally create a new schedule for the design without loosing any of the goals the design was designed to reach.

A behavioral description will be the input to such tools (VHDL in this case). The input will be analyzed, parsed, translated into an intermediate form and finally given to the scheduler which schedule its processes according to some criteria (algorithm).

1.3 Motivation: Why high-level synthesis?

When dealing with this issue a question would be raised "*Why High-Level Synthesis?*"

The answer to this question would begin by answering the following question *Why high-level hardware design languages like VHDL?*

High-level hardware design languages are used for several reasons and some of them are:

- Have an easy design tool and make it easy for designers to design before switching to hardware.
- Provide organized designs.
- Provide simulation and testing of the design before implementing the hardware.
- Provide good understandability of the design and make it easier to control.

However, the development of these design languages has reached a to a high level where the program is now easy to be implemented but not very efficient any more.

This issue raised the idea of high-level synthesis where a synthesis tool takes a behavioral high-level hardware design language code and reschedule the tasks to have a good and efficient design.

1.4 Thesis outline

This thesis deals with high-level behavioral synthesis using VHDL. To do this, a lot of research and readings have been done on this issue, gathered information, exploring some tools that deal with this issue , and implementing a synthesis tool. The synthesis tool accepts behavioral VHDL design, translate it to an intermediate form, and schedule the processes in this form to give a proposed new design which is supposed to be more efficient and acquire less space.

Chapter 2 includes an introduction of the high-synthesis task which gives information and a background about the subject .

In chapter 3, a review of some of the literature written on this issue will be presented by summarizing some of the good papers that dealt with high-level synthesis using VHDL.

In chapter 4, the solution approach will be presented, the steps that have been followed, the tools that have been used to complete this work, and finally a small presentation of the synthesis tool and how it works by showing the layout of the tool, what does it do, and give small examples to give a better view and understanding of the tool. This section will appear as a small manual of the tool.

In chapter 5, results of some experiments and benchmarks that have been experimented using the synthesis tool.

The last chapter contains the conclusion of this thesis.

Chapter 2 High Level Synthesis

2.1 High-Level Synthesis Task

Synthesis is defined as the translation of a behavioral description into a structural one. The main difficulty in this process is that most behaviors have no implied architecture. Thus, it is rather impossible to develop synthesis algorithm that will generate the same quality from all possible design descriptions. One of the main tasks of high-level synthesis is to find the structure that best meets constraints while minimizing other costs [1].

2.2 The VHDL Synthesis Domains

System-Level Synthesis Domain: System-level synthesis tools partition the system description into hardware and software pieces based on the constraints specified by the designer. Interface synthesis is also performed to ensure that appropriate communication protocols are incorporated into the hardware and software portions. More details of the system level specification and synthesis process are available.

Chip-Level Synthesis Domain: Each of the chips derived from system-level synthesis must be synthesized into hardware. This is the function of the chip-level synthesis domain. The input to this domain is the behavioral specification of each chip.

Logic/Layout-Level Synthesis Domain: The RT level netlist that is output from the chip-level synthesis can be further synthesized using some of the commercially available logic and layout tools.

2.3 System Design: The Productivity Bottleneck

Nowadays one of the major objectives within VLSI domain is the improvement of the design quality and the designers' objectivity. This is due to the fact that the design process is characterized by two sets of factors and variable ones. For instance, typical large design budgets are usually fixed to around 10 to 15 persons over 18 months, and designers productivity has been evaluated to 10 projects per day. Controversially, ASIC design complexity forecast for year 2000 is 7 million transistors for 0.18 micron CMOS technology. It is expected that this exponential increase will continue until year 2010 in order to reach 40 million transistors[1].

At present, when using the most advanced logic and RTL (register transfer logic) synthesis tools, such a budget covers the design only 1 to 2 million transistors ASICs. This means that for year 2010 we still have a factor of 20 to 40 to close the gap between what present tools provide and what the technology will deliver.

It is then clear that we need improvement of the design quality and designers' productivity. This may be achieved in two ways:

- Providing higher level design tools allowing to start from a higher level of abstraction. After the success and widespread acceptance of logic and RTL synthesis, the next step is behavioral synthesis, commonly called architectural or high-level synthesis.
- Using more structured design methodologies allowing for an extensive reuse of existing components and subsystems. It seems that 70% of new designs correspond to existing components that cannot be reused because of a lack of methodologies and tools.

2.4 From Physical Design to System Design: Abstraction Levels

The arrival and acceptance of standard Hardware Design Languages (HDLs) such as VHDL and Verilog, have promoted high-level specification of electronic circuits. HDLs may be used for specification of whole systems, as well as subsystems that may then be assembled within a hierarchical (structural) description. The fact that various tools (for synthesis and simulation) have been developed this last decade has also helped high-level specification for VLSI to emerge and to gain more and more acceptance in the VLSI design community.

HDLs can be used for design specification at various abstraction levels, from gates to the behavioral level. Timing concepts will be used to fix the abstraction level of design specification.

Timing is the main issue during the process of designing an integrated circuit. In fact regardless of the abstraction level, the design process may be defined as the refinement of high-level concepts (operation, primitives, statements or constructs) into lower level concepts using more detailed timing units.

At the lowest level the basic timing unit is the delay. The design is specified in terms of gates and devices that are interconnected through nets. A typical specification at this level is a schematic. Of course this may be described using a HDL format such as VHDL or Verilog. At this level the components of the design (gates, nets, devices) are characterized by delays. Simulation and timing analysis tools are needed in order to compute the performance of the full design. The clock period will correspond to the longest path between two memorization elements. A path may include several components.

The next level is called logic or Register Transfer Level (RTL). The design is specified at the clock cycle level. Typical descriptions will state what to do at each clock cycle. A description is generally formed of a set of registers, operators, and transfers between registers and operators. Typical representations used by synthesis tools at this level are Boolean equations, FSMs and BDDs.

Such representation can be extracted automatically from HDLs. The main design steps applied starting from this level are logic optimizations, synthesis state encoding and technology mapping. The main role of these transformations is to fix the clock period and the gate count. The logic optimization is generally a trade-off between these two parameters. In VHDL, when writing statements an RTL description, we assume that all the computations between two wait statements can be transformed into a set of gates able to perform that computations within a clock period. The decomposition of the clock period delays is made automatically by RTL synthesis. Of course the designer can control the synthesis results through different writing styles or a set of constraints. This kind of description is also called synchronous description or cycle based description.

The next level is called behavioral or algorithmic level. A design is specified in terms of computation steps. The concept of operations and control statements (loop, wait) will be used to sequence the I/O events and the computation. At this level we have an event-driven specification. A typical description will be composed of a set of protocols to exchange data with the external world and internal computation. A computation step is composed of the set of operations executed between two successive I/O and/or synchronization points. A computation step may be take several clock cycles. The main function of behavioral synthesis is to split these computation steps into a set of clock cycles. Moreover, some of these computation steps may include data dependent computation implying a non-predictable (variable) computation time. A typical representation at this level is Behavioral Finite State Machine (BFSM), Control Flow Graph (CFG), Data Flow Graph (DFG) and Control-Data Flow Graph (CDFG). When writing a behavioral description we assume that all the computations between two synchronization points can be decomposed into a set of clock cycles that respects the communication protocol. The decomposition of a computation step into clock cycles is made automatically by behavioral synthesis. One of the major problems when using behavioral synthesis is to specify complex and precise protocols within behavioral description.

At the highest level, we have the system level specification. Such a specification includes distributed control and multi-thread computations. The basic timing unit at this level is the communication transaction. The basic primitive is the process. A description will be composed of a set hierarchical, parallel and communication process.

Most hardware description languages allow the first three abstraction levels (Gates, RTL, Behavior). For each of these languages, each level corresponds to a writing style using a subset of the language. One can note that we can describe and simulate a full system at the gate level or describe a gate at the behavioral level.

When describing large systems, it is often the case that all the specification levels have to be combined. In fact a system specification is seldom given in a unique specification level. Generally, it is composed of blocks described at different abstraction levels.

2.5 Behavioral Synthesis

Behavioral synthesis is the processes that from a behavioral or functional specification and produces an architecture able to execute the initial specification. The architecture is generally given as a Register Transfer Level (RTL) specification and is composed of a datapath and a controller. The behavioral description specifies the function to be performed by the design. It may be textual or graphical. A behavioral synthesis tool acts as a compiler that maps a high-level specification into an architecture. In order to modify the architecture you simply have to change the behavioral description and return it through the behavioral synthesis tool. The use of behavioral synthesis induce a drastic increase in productivity since behavioral descriptions are smaller and easier to write and modify.

High-Level synthesis is the bridge between what is called system design and CAD tools acting at the logic and RT Levels.

2.6 Behavioral Synthesis Tools

Several synthesis tools have been published in the literature. Only few of them have been applied for the design of VLSI chips[1]. While behavioral synthesis tools have been applied successfully to DSP algorithms, arithmetic computation and interfaces, behavioral synthesis was less successful for the off real-time controllers, complex heterogeneous design and data dependent computation. None of the existing behavioral synthesis tool has been recognized as a universal tool that may be efficient in all application domains and for all kinds of architectures. Existing behavioral synthesis tools may differ from several points of views. These correspond to the main choices that have to be made when designing a behavioral synthesis tool and more generally application generators.

2.7 Algorithms for behavioral synthesis

Behavioral synthesis produces an architecture starting from a functional specification. It is generally decomposed into five major transformations: generation of an intermediate form, scheduling, allocation binding and architecture generation. Only a brief outline of these steps will be given in this section :

1.Compilation of the behavioral description and generation of intermediate form

This step may include compiler-like transformations aimed at removing all the details related to the description language and the writing style. Such transformations include constants propagation, dead code, elimination, loop unfolding and procedure expansion.

2.Scheduling

Scheduling is the partitioning of the behavioral description into subgraphs, each one being executed in a single control step. A control step corresponds to a transition of an FSM. It may include several operations to be executed in parallel.

3.Allocation

Allocation fixes the amount and types of resources needed to execute the behavioral description.

4.Binding

This step decides which resources will be used by each operation of the behavioral description.

5.Connection allocation

This step fixes the resources needed for communication between the units of the datapath.

6.Architecture generation

This step produces an RTL description of the synthesis design.

2.8 VHDL for High-Level Synthesis

In typical design environment designers describe the design implementation using hardware description language such as VHDL. The design can be next synthesized using high-level synthesis and fed to lower level synthesis tools in order to generate the final physical design. This process can only succeed if the synthesis system can guarantee a consistency among abstraction levels[1].

Many different hardware designs can implement a given behavioral description, a subset of which also meet specified requirements such as cost, performance, and testability. Existing hardware synthesis systems typically use cost and performance as the main criteria for selecting the best hardware implementation, and even consider test issues during the synthesis process [3].

2.9 Using VHDL for synthesis

A VHDL description consists primarily of entities and architectures. An entity describes a device's interface with its environment through a list of input and output ports. The architecture describes the relationship between the device's inputs and outputs. One can model the device with any combination of the description levels mentioned earlier: architectural, register-transfer, logic, and structural.

Because VHDL was designed mainly as a simulation language, it presents some problems for synthesis. However by following modeling guidelines associated with each level of description, one can use it effectively as a synthesis language. Although a complete discussion of these guidelines is beyond the scope thesis.

The architectural level allows the designer to model design functionality with process statements. A process statement may include several sequential statements that provide the capabilities found in many high-level programming languages such as Ada, C, and Pascal. Although the Silicon 1076 synthesis environment supports most sequential statements at the architectural level, some restrictions apply like Wait statement time-out, Signal assignments are not permitted in all cases,... Most of these restrictions are not found in VHDL, which make more realistic and better for synthesis.

Chapter 3 Literature Review

This chapter will review some of the literature written in this field in the recent years. A summary of three good papers will be presented. These papers describe many aspects of high-level synthesis of behavioral VHDL and its importance in designing and simulating hardware. Moreover, these papers describe some tools that have been implemented or under construction for synthesizing behavioral VHDL.

3.1 The Design Process of Behavioral VHDL

3.1.1 AA-VSS system

The design process of behavioral synthesis from VHDL descriptions can be divided in two parts, (1) the architectural allocator (AA) that derives the appropriate type and number of resources that can satisfy the performance constraints imposed by the designer, and (2) the VHDL synthesis system (VSS) that synthesizes a RT-level netlist based on the allocation constraints. These two parts are combined together to form the AA-VSS system.

AA-VSS is based on a top-down design methodology that is divided into three domains[8]:

- System-level synthesis domain
- Chip-level synthesis domain
- Logic/layout-level synthesis domain

Each one of these deals with its domain and they are complementary. To synthesize systems we must make sure the chip level is working well and so on.

AA-VSS can be used to explore the design space to produce efficient chip design. The input to the system is a behavioral description written in VHDL. In order to facilitate the exploration of a large design space, many knobs are available on the system. The designer can control the synthesis process by changing the settings on the knobs. The knobs are listed as follows:

Partial resources (RAMs, register and register files, muxes and buses) [8]

- Clock period
- Performance constraints
- Component library
- Control pipelining
- Memory hierarchy

The designs synthesized by the AA-VSS system can be represented using the FSMD model of hardware. In FSMD model, the synthesized design consists of two parts[8]:

- The data path, which performs the storage of the data and computations on the stored data values. It consists of functional units, memory components, and buses.
- The controller, which performs the sequencing of various states, and controls the data path operations. It can be represented by a finite state machine, consists of a set of states and transmissions between states.

AA-VSS system accepts behavioral VHDL as input to the system. VHDL descriptions may contain sequential, process-level statements such as

- Sequential assignment statements
- Conditional statements (if, case)
- Loop statements
- Wait statements

AA-VSS does not accept the following language features in VHDL[8]

- Enumerated types
- Aliases
- CONSTANT declaration
- Null statement
- Procedures and functions
- Exit statements
- Return statements
- Loop statements with no iteration schemes

The first step in high-level synthesis is architecture allocation. Architecture allocation refers to the process of selecting the functional units, storage elements, and interconnects used to implement the datapath. Also, it defines the way in which data may be transferred between the datapath functional and storage units.

Architecture allocation consists of the following information:

- Level allocation: grouping the storage elements into different levels
- Storage allocation: This defines the number and types of memories used in the datapath.
- Functional unit (FU) allocation which specifies
    - The types of FUs used
    - The number of FUs of each type
    - The delay and data initiation rate of each FU
- Interconnect Allocation which defines:
    - The number of buses in the design
    - The delay per bus

Scheduling partitions all the operations in the CDFG into different sub-graphs such that each sub-graph is executed in one control step. However, the scheduling process must ensure that sufficient resources are available in each clock cycle to execute all the operations assigned to that control step[8].

Binding maps the variables and operations in the scheduled CDFG onto specific instances of the allocated functional units, storage components, and interconnect units while ensuring that the design behavior operates correctly.

3.1.2 Assessments

In this paper the authors is dealing with a synthesis system that, first derives the appropriate type and number of resources and second synthesize the RT-level netlist based on allocation constraints.

The authors describe a top-down design methodology to designing the system by going through three synthesis levels system, chip, and logic/layout levels. After that the authors goes over each step and describe it. However they did not a brief explanation of how binding and scheduling is done. Scheduling algorithms as well as binding criterion were not offered.

3.2 Synthesis in a VHDL Environment

The growth of system complexity has made gate-level hardware design increasingly difficult. To manage this complexity, computer designers are turning from traditional bottom-up methods to more hierarchical design practices. Hardware design languages such as VHDL now make it possible to use a single set of semantics to specify, simulate and design complex systems.

VHDL's advantage as a specification and synthesis language is that it can describe hardware at various levels of abstraction. The term architectural level describes the intended behavior of the hardware. The next level is the register-transfer level, which describes a system as a set of interconnected storage elements and functional blocks. At the logic level, a network of gates and flip-flops, the behavior is specified by logic equations. The lowest level is the structural level, the netlist that specifies what hardware components to use. RTL synthesis is the process of generating a netlist from an RTL description [11].

VHDL description of systems consists of a set of entities (device interface with its input and output) and architectures (relationships between devices input and output). However, VHDL was designed as a simulation language and it may present some problems. On the other hand, following some guidelines associated with each level of description, one can use VHDL effectively as a synthesis language.

Planning refers to the designer's task of making decisions that result in implementations exhibiting various sets of trade-offs. Architecture partitioning, exploration and evaluation of alternatives are the key steps in planning a design implementation.

As a first step, the architectural exploration tool groups atomic operations in the CDFG into clusters. It applies a hierarchical clustering algorithm to the operations to form a cluster tree. The algorithm assigns each operation in the CDFG to an individual cluster.

The goal of design space exploration is to find architectures that produce the intended behavior and comply with the user's area and timing goals. The exploration tool obtains different designs by considering the cluster sets at various levels of the tree, starting from the root and moving towards the leaves [11].

The tool chooses the hardware resources for each partition by assigning the minimum number of function units necessary to perform the operations in the cluster. Then the tool invokes a scheduler, which transform the operations into control steps using the appropriate hardware resources for the design.

The goal of architecture evolution is to predict the performance of an architecture found during design space exploration. To do so, the tool uses estimators that predict the area and speed of the selected architecture.

When scheduling is complete, the number of registers and multiplexers requires of each partition of an architecture are estimated [11].

3.2.1 Flexible architectural synthesis (Flexsyn)

Flexsyn, the architecture synthesis engine perform the following functions in the design environment: scheduling, connectivity binding, and register optimization.

Flexsyn's scheduling algorithm attempts to minimize the number of control steps, within the constraints of the clock frequency and module set.

Flexsyn uses a branch-and-bound algorithm for connectivity binding. For all operations in each state, the algorithm performs the following steps [11]:

1. Binds the registers carrying input variable to buses.
2. Binds buses to the inputs of function units.
3. Binds the output of function units to buses.
4. Binds buses to registers.

The connectivity-binding algorithm tries to reduce the number of registers in the design. It looks only at a portion of the CDFG at a time, so it may use more registers than necessary.

3.2.2 Assessments

In this paper, the authors are talking about synthesis using VHDL, hardware design and simulation language. They showed the importance of VHDL in synthesis and how it helps and make it easier when applying synthesis steps like clustering, partitioning scheduling, binding and finally optimizing a design. On the other hand, this paper did not show the advantage of VHDL over other hardware design languages. As a paper talking about VHDL, It did not show where are the functions that VHDL do and others do not.

3.3 Architectural synthesis Via VHDL

The term architectural synthesis is used to differentiate a class of design methodologies for large digital (VLSI) system. These design styles can be characterized by the following attributes:

- The system themselves are built from a "few" "large" objects rather than many small ones.
- The design is done with components from a design library.
- The design trade offs are in terms of structure. That is, we design by choosing components and interconnections strategies, rather than by choosing between algorithms for implementing functions.

The rational for this design styles based on the fact that the size of the design are such that optimization tool s take prohibitively long to run on the entire design. Rather a library of optimized (a possibly parameterized) components is used. We assume that the designer is primarily concerned with the design of the algorithm that will be implemented and the resources available (speed and area) rather than low level implementation issue. Therefore, the tradeoff s available to the designer are in terms of variations in the algorithm, component, interconnection style, and control style selection.

To build systems using this design method, the user (and the design tool) must have access to the characteristics of available components to make informed choices. If the library is a set of templates of parameterizes modules (rather than completed modules) the design tool must have ways to estimate the final size and speed of the system in order to guide the designer decision making..

The primary advantage of this design style is that it tends to produce working systems with minimum time from both the design synthesis tool and the designer. Systems are compositions of tested components interconnected using fixed techniques, usually with simple timing models. If the components are well characterized, and the design tool can perform placement and routing effectively, it is possible to generate working systems which meet specifications.

On the other hand, the primary draw back of this design style is that it tends to preclude a level of global system optimization which is possible when an entire design is synthesized at once. Generally, design tools can not perform this level of optimization on large systems. However, there are times when global optimization among components is the only way to get the desired performance from a system.

The goal in this work is to have a tool where we can get the benefits of a component based design style without sacrificing performance. We believe that this can come from removing the arbitrary boundaries between components which exist when they are just " cut and pasted" from a component library. We build a single hierarchical VHDL entity using components from a VHDL component library. At different stages in the synthesis process, redundant operators at the boundaries between the components are removed. The resulting monolithic design is then given to behavioral synthesis tools. However, to fully utilize the advantages of a library we needed to keep in the library (along with the VHDL description of the components and their performance characteristics) an abstract representation of their optimized layout. During the final synthesis phase of the design, the tools will use this information to speed up the synthesis of the full design.

3.3.1SandS

SansS (Slicer And Spicer) is a tool which performs architectural synthesis, the compilation of a high-level behavioral specifications into a register transfer level architecture. The input to SandS is a behavioral specifications written in a high-level language which describes an algorithm or set of algorithms that the user wants implemented on a chip. This input is generally more abstract than a VHDL structural/Dataflow description that would be used as the input to a logic synthesizer. An example simple file for a simple division algorithm we used throughout this paper is shown in the following Figure:

```
Program Divide( input, Output);
        Type integer = {0..7};
        Port in_y, in_x, out_q: integer;
        Var y, x, q: integer;

   Begin
     Y:= in_y; in_x; q :=0;
     If (y>= x)  then
       Repeat
          Y:= y-x;
          q:= q+1;
     when y>x ;
   out_q:= q
  end.
```

Division Algorithm

T havioral

sp e into a

flow-graph representation. The second is the allocation of operations to states or control steps of the machine (also called state binding). The third task is the allocation of hardware components to operations, register assignments, and the creation of multiplexors and/ or busses to complete the data path. This last task is also referred to as connectivity binding.

SandS builds an architecture based on constraints imposed by the user. The idea is to let the user perform architectural design tradeoffs, while the system synthesizes design that meets the user's constraints. This separates two basic type of knowledge needed to produce the design. The first type is the "What To Build" knowledge (the users Job) and the second is " how to build" it (the synthesizer job". The separation of knowledge in SandS frees the user from implementation details and presents the opportunity for creative exploration in the design space. The user can

Ask "what if" questions and have systems generated quickly to explore design alternative.

The constraints that the user specifies are control knobs for the system. By alternating the adjustments of the knobs, the user can direct SandS's region of operations within the design space. We have selected the following three major user constraints for three major user constraints for the architecture synthesizer:

1-*component selection:* the component selection mainly consists of the specification of the number of architectural components such as ALU's multipliers, adders, subtractors, etc. that are allowed in the design along with the component delay time.

2- *control style selection:* the control style selection specifies the type of micro-engine controlling the design. Ti may be of a PLA or random logic type and may have pipeline registers inserted within it.

3- *Interconnect style selection:* the interconnected style selection chooses between a tri-state bus based system or a system interconnected by multiplexors.

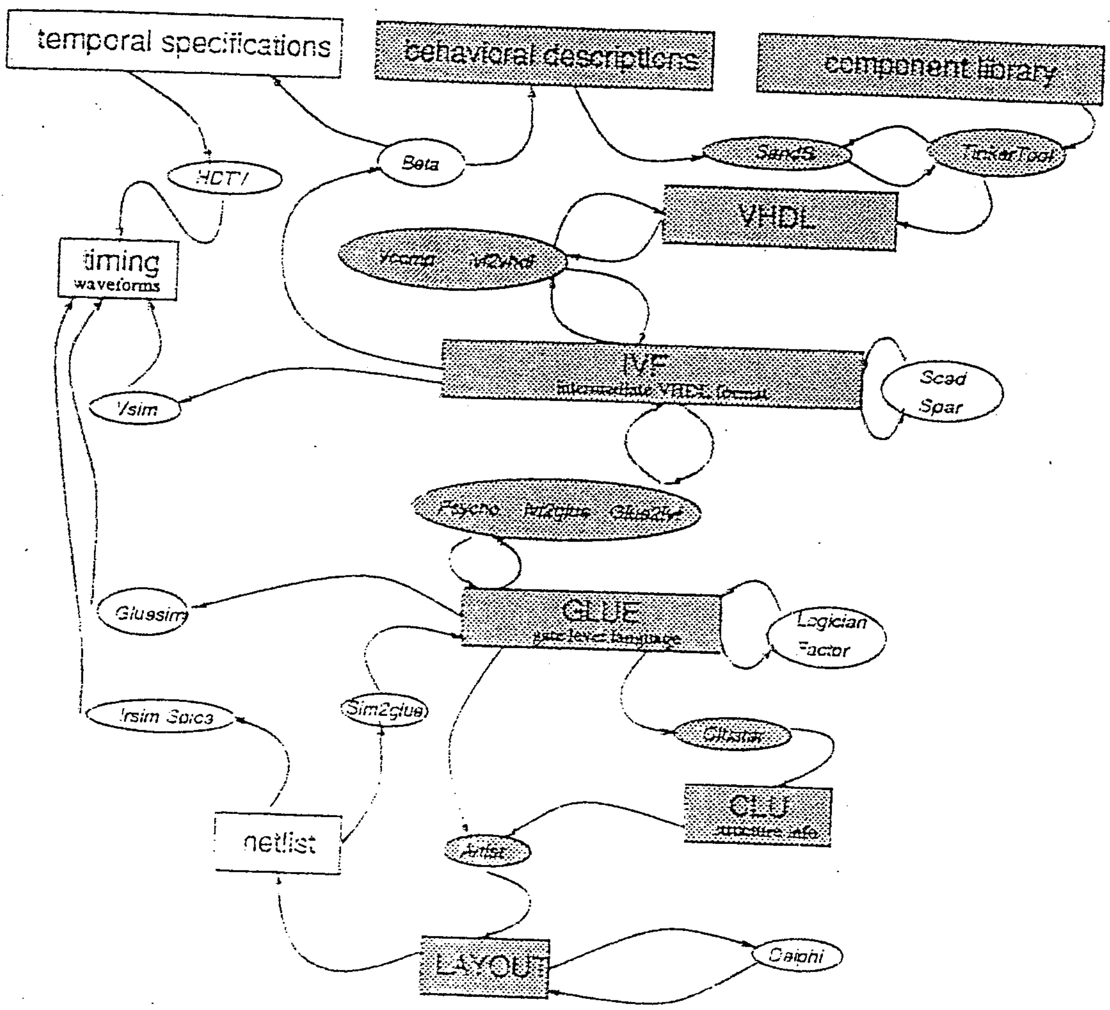
4- *Control cycle time*: the clock speed or control step period determines the amount of time allotted for each control step. The longer the period, the more time there is during each control step to perform each operation.

Information from the component library is necessary to produce reasonable estimates for the component and control delay times. These estimates could be guesses or result of VHDL simulations of the components. They could, in fact, be generated by running the library components individually through the module and layout synthesis tools, extracting the netlist from the layout, and simulating the result.

Early in the design process, the user must be able to obtain estimates quickly, in order to eliminate options of the design space that aren't worth exploring. Using estimates rather than real values from low-level simulation will help reduce the design space quickly. As we discuss below, it is possible, even desirable, to use "hypothetical" components at this phase of the design. As the design starts to approach a viable solution, the estimates must become more accurate and the designer (with the system) can spend more time deriving accurate estimate.

3.3.2 Keystone

Keystone is a multilevel design and synthesis system developed jointly by researchers at the University of Pittsburgh, and Pennsylvania State University. It is the result of several ongoing research efforts by researchers at both schools. Keystone consists of a suite of specification, synthesis, simulation, and analysis tools for designing VLSI systems. The following figure represent this system:



The Keystone System

The tools themselves are indicated by ellipses while each data abstraction level which we support is indicated by a box. The title of each box represents the format of the data within the abstraction level. The shaded boxes and ellipses indicate the path through the system which we will focus on later in this paper.

The tools are shown connecting the different design representations with which they interact. Most of the tools perform optimization within a design abstraction layer as well as transformation between representations. Some of the optimization tools perform restructuring or decomposition as well.

User supplied input to the design system can come at any of several abstraction levels. It can come from an architectural or temporal specification, it can be expressed in either behavioral or data-flow VHDL text or it can be given as a schematic graphical description of the architecture to bi implemented. System output can be generated at many abstraction levels. Perhaps the lowest levels of output of the design system are two dimensional gate matrix layout in a form compatible with MAGIC and timing waveforms from both the high-level tools and simulators.

As indicated by the bi-directional arrows, in addition to going from "higher" abstraction levels to lower ones, we support reverse transformation, e.g., from a layout to a VHDL or a schematic description of that layout. We support these various transformations in order to provide the user with a variety of input and output format options including netlists, mask designs, or logic equations fro other systems. By being able to move easily between representations in the design space, the system is able to more easily handle the design level interactions, user feedback operations, and support the user in the design process.

3.3.3 Assessments

This paper talks about how to produce working systems using design libraries and tested components in addition to the design synthesis tools.

Then the author present some previous work done on synthesis by writing about some of the synthesis tools and how they work.

What is not offered is, enough information and description about what are the good features of each tool and where each tool is better than the other.

Chapter 4 Solution Approach: Tool Construction

4.1 Introduction

As said before, high-level synthesis produces, from a behavioral description, a register- transfer-level design which satisfies a given set of hardware cost/execution time constraints. The major tasks in high-level synthesis are (i) translating the behavioral description, (ii) scheduling of computation steps, and (iii) hardware allocation. These steps will be described in the solution approach presented in this chapter.

The approach presented in this chapter differs from other approaches. The synthesis tool that is described in the next sections has two main features that might not found in many other synthesis tools. These features are:

- The tool provides conditional and nonconditional design schedules
- The tool provides schedules with resource and without resource constraints with number and type of resources set by the user when applying resource constraint scheduling.

To construct the synthesis tool, constructing algorithms steps have been followed. The input to the tool, as said before, is behavioral VHDL. The input is given to a compiler or translator to translate it into an intermediate language after getting rid of some redundant and unnecessary stuff in the input. This intermediate language is easy to read and schedule into a CDFG and give a new design from the original behavioral input design.

To implement the synthesis tool, JAVA was used as the programming language for several reasons:

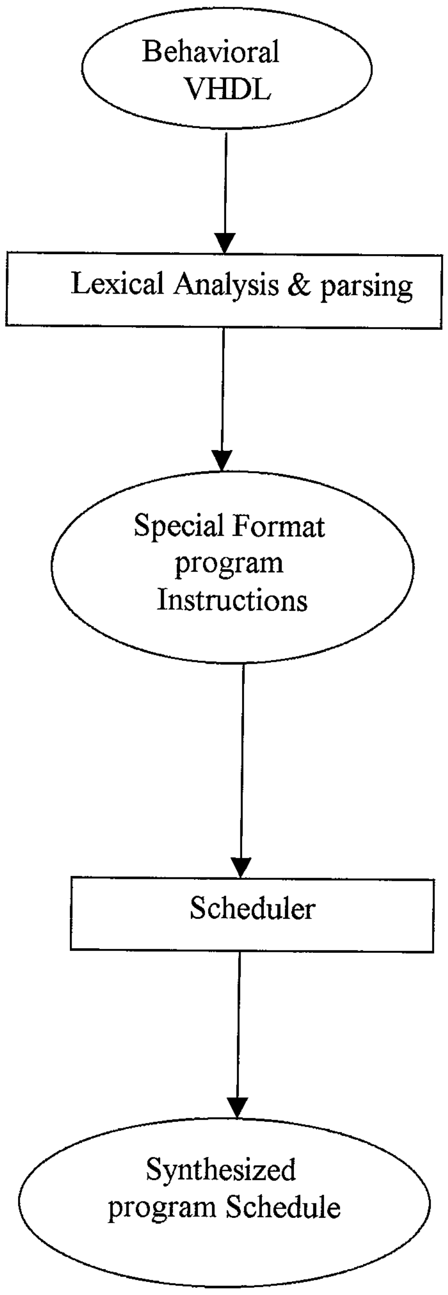
- It is portable

- It is object oriented which helps to visualize the intermediate form which are objects that contain processes and their information
- It has good graphical user interface features which make it easier to use the synthesis tool
- It will be easier to update the program or parts of it, or use it with other software

This chapter will go over each step followed to implement the tool. The steps are:

1. Behavioral VHDL translation to intermediate form: by building a special purpose compiler
2. Scheduling the intermediate form using different scheduling algorithms
3. Getting a new design schedule

The flow chart of the synthesis tool:


Behavioral VHDL
Lexical Analysis & parsing
Special Format program Instructions
Scheduler
Synthesized program Schedule

4.2 Behavioral VHDL Translation

Compilation or translation is not an easy issue. This is a field by itself and it needs a good understandability of some lower level issues that we usually, as high level programmers, do not deal with.

Compilers or translators are divided into two main parts: Lexical analyzers and parsers. Each of these parts has its job but they are absolutely complementary and compatible.

In the following sections I will be explaining each parts of the compiler and what does it do.

4.2.1 Lexical Analysis

4.2.1.1 Introduction

To translate a program from one language into another, a compiler must first pull it apart and understand its structure and meaning, then put it together in a different way. The front end of the compiler perform analysis; the back end does synthesis .

The analysis is usually broken up into:

Lexical analysis: breaking the input into individual words or "tokens";

Syntax analysis: parsing the phrase structure of the program; and

Semantic analysis: calculating the program meaning.

The Lexical Analyzer takes a stream of characters and produces a stream of names, keywords, and production marks; it discards white space and comments between tokens. It would unduly complicate the parser to have to account for possible white space and comments at every possible point; this is the main reason for separating lexical analysis from parsing.

I will not go very deeply in explaining and discussing the lexical analyzer since this is a field by itself but I will talk briefly about how I produced my lexical analyzer and the tool I used to produce it which is Jlex.

4.2.1.2 JLex: A Lexical Analyzer Generator

JLex is a Lexical Analyzer Generator that produces a Java program from lexical specifications. These specifications are written and specified into a file in a special format and is given as an input file to JLex which will produce the Lexical analyzer for the language specified in the file. The specification contains regular expressions and actions.

The JLex input file is organized into three sections, separated by double-percent directives("%%"). A proper JLex specification has the following format:

```
User code
%%
Jlex directives
%%
regular expression rules
```

In the next pages I will be showing the construction the lexical analyzer that was used in the translation process of the behavioral VHDL. First, I will show the specification file that was given to Jlex in order to generate the lexical analyzer. Then I will show the generated lexical analyzer.

Some of the issues might be ambiguous, however the explanation f these issues is out of the scope of this thesis as I myself faced some problems in having a good understandability in order to generate this lexical analyzer and later the parser.

4.2.2 Parsing

Definition: The way which words are put together to form phrases, clauses, or sentences.

Webster's Dictionary

4.2.2.1Context-Free Grammars

As has been said that a language is a set of strings; each string is a finite sequence of symbols taken from a finite alphabet. For parsing, the string is a finite source programs, the symbols are lexical tokens, and the alphabet is the set of token types returned by the lexical analyzer.

A context-free grammar describes a language. A grammar has a set of productions of the form:

Symbol-> symbol symbol .....

Where there are zero or more symbols on the right-hand side. Each symbol is either terminal, meaning that it is a token from the alphabet of strings in the language, or nonterminal, meaning that it appears on the left-hand side of some production. No token can ever appear on the left-hand side of a production.

The abbreviation mechanism in JLex, whereby a symbol stands for some regular expressions is convenient enough that it is tempting to use it in interesting ways.

4.2.2.2Using Parser Generators (CUP)

CUP (Construction of Useful Parsers) is a tool to generate a parser. A CUP specification as in Jlex has a preamble, which declares the list of terminal symbols, nonterminals, and so on, followed by grammar rules. The preamble also specifies how the parser is to be attached to a lexical analyzer and other such details.

As in the lexical analyzer section I will not go so much in this subject but I will insert in the next section the parser specifications I used to generate my parser.

Also as in the lexical analyzer, the input file for the parser generator is formatted and partitioned in many sections and this will be viewed in my input file to generate my parser using CUP.

The input file used to generate the parser must have the following specifications:

-Preliminaries to set up and use the scanner (lexical analyzer)

-Terminals (tokens returned by the scanner

-Non terminals

-Precedences

-The Grammar

Giving a file having these specification, CUP will generate a parser .

4.3 Scheduling

4.3.1 Introduction

Scheduling is a very important problem in architectural synthesis. The scheduling of the sequencing graph that prescribes operations dependencies, determine the precise start time of each task. Scheduling determines the concurrency of the implementation and has a major effect on its performance. The maximum number of concurrent operations of any type on any step is a lower bound on the number of hardware resources of these operation types.

In this section, I will present some models for the scheduling problem and describe the major algorithms for scheduling that I have used in my implementation. I consider first sequencing graph that are not hierarchical (no branching or iterative constructs) and representative of the model data flow. These graphs do not support pipelining.

4.3.2 A Model for The Scheduling Problem

The sequencing graph is a non-hierarchal polar acyclic graph $G_s(V,E)$, Where the vertex $V=\{v_i;\ i=0,1,\ldots,n\}$ and the edge set $E=\{(v_i,v_j);\ i,j=0,1,\ldots n\}$ represent dependencies. Let $D=\{d_i;\ i=0,1,\ldots n\}$ be the set of operation execution delays. We will assume that the delays are data-independent and known and that the delays for the source and sink vertices are both zero, i.e. $d_0=d_n=0$.

The sequencing graph requires that the start time of an operation is at least as large as the start time of each of its direct predecessor plus its execution delay $(ti \geq tj+dj)$

4.3.3 Scheduling without Resource Constraints

Unconstrained scheduling is applied when dedicated resources are used. Practical cases leading to dedicated resources are those when operations differ in their types or when their cost is marginal when compared to that of steering logic, registers, wiring and control.

Unconstrained scheduling is also used when resource binding is done prior to scheduling, and serializing the operations that share the same resource solves resource conflicts.

4.3.4 Scheduling Algorithms

4.3.4.1Unconstrained Scheduling: The ASAP Scheduling Algorithm

The unconstrained Scheduling minimum-latency scheduling problem can be solved in polynomial time by topological sorting the vertices of the sequencing graph. This approach is called *as soon as possible* (ASAP) scheduling since the start time of each operation is the least one allowed by the dependencies.

The ASAP Algorithm

```
ASAP(Gs (V,E)) {
  Schedule $v_0$ by setting $t_0^s = 1$;
  Repeat{
        Select a vertex $v_i$ whose predecessors are scheduled;
        Select $v_i$ by setting $t_i^s = \max(j: (v_j,v_i)\ c\ E)\ \ t_i^s + d_j$ ;
  }
  until ($v_n$ is scheduled);
  return ($t^s$);
}
```

4.3.4.2 Latency-Constrained Scheduling: The ALAP Scheduling Algorithm

We consider now the case in which a schedule must satisfy an upper bound on the latency, denoted by μ. This problem may be solved by executing the ASAP scheduling algorithm and verifying that $(t_n^s+t_0^s)\leq\mu$.

The *as late as possible* (ALAP) scheduling algorithm provides maximum values of start times. We denote by t_l the start times computed by the ALAP algorithm.

The ALAP Algorithm:

```
ALAP(Gs (V,E), $\mu$ ) {
    Schedule $v_0$ by setting $t_n^L = \mu + 1$;
    Repeat{
            Select a vertex $v_i$ whose successors are scheduled;
            Select $v_i$ by setting $t_i^L$ = min(j: $(v_j, v_i)$ c E)  $t_j^L + d_i$ ;
    }
    until ($v_n$ is scheduled);
    return ($t^L$);
}
```

4.3.4.3 Forced-Directed Scheduling

This algorithm was proposed by Paulin and Knight as a heuristic approach to solve the resource-constrained and the latency-constrained scheduling problems. Before describing the algorithm let me briefly describe the underlying concepts.

The time *frame (t_e^s t_i^l; i=0,1,..,n)*of an operation is the time interval where it can scheduled. t_e^s t_i^l these are the earliest and the latest times in a frame which can be computed by the ASAP & ALAP algorithms. The *operation probability($p_i(l)$)* is a function that is zero outside the corresponding time frame and is equal to the reciprocal of the frame inside it.

The *type distribution($q_k(l)$)* is the sum of the probabilities of the operations implementable by a specific resource in the set $\{1,2,..,n_{res}\}$ at any time step of interest.

In forced-directed scheduling the selection of a candidate operation to be scheduled in a given time step is done using the concept *force*. Forces attract/repel operations into/from specific schedule steps. Here the forces are related to its probability that was mentioned. The assignment of an operation to a control step corresponds to changing its probability. These forces are related to concurrency of operations of a given type. The larger the force the larger is the concurrency. Forces are categorized in two categories self-forces and predecessor/successor forces. These forces are computed using some formulas proposed by mathematicians.

I will not go further in this explanation since it is out of the scope of this thesis but I will write the force-Directed algorithm.

To take care of unconditional resource sharing and conditional resource sharing

–Transform a dataflow graph with conditional branches into dataflow graph without conditional branches (bottom-up)

–Apply scheduling algorithm

–Transform the schedule got for the original dataflow graph(top-down)

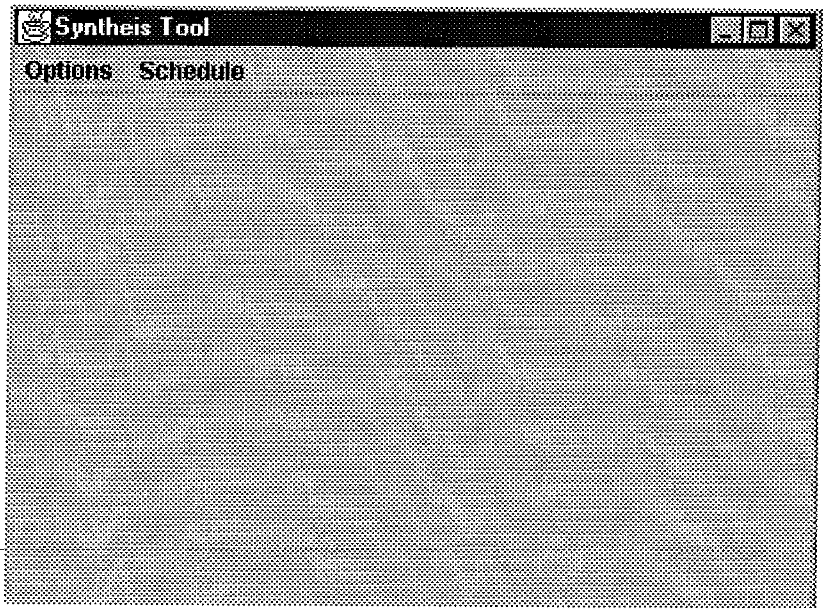
The FDS Algorithm

```
FDS(G(V,E), lat){
 repeat {
  compute the time-frame;
  compute the operation and type probabilities;
  compute the self-forces, pred/succ-forces and the total forces;
  Schedule the operation with the least force and update time-frame;
}
until ( all operations are scheduled)
return( t ) }
```

4.4 The Synthesis Tool

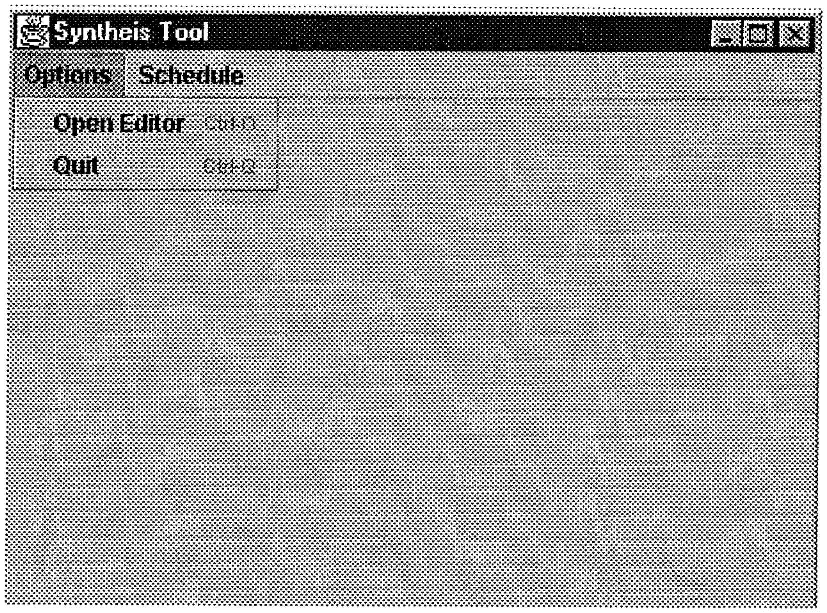
In this Section an overview will be given on the synthesis tool that have implemented. The section contain a presentation of what does the tool do and how does it do it with some figures showing the graphical user interface of the tool and examples of their use.

When we first run the program where the tool is implemented the layout shown in this figure will appear:



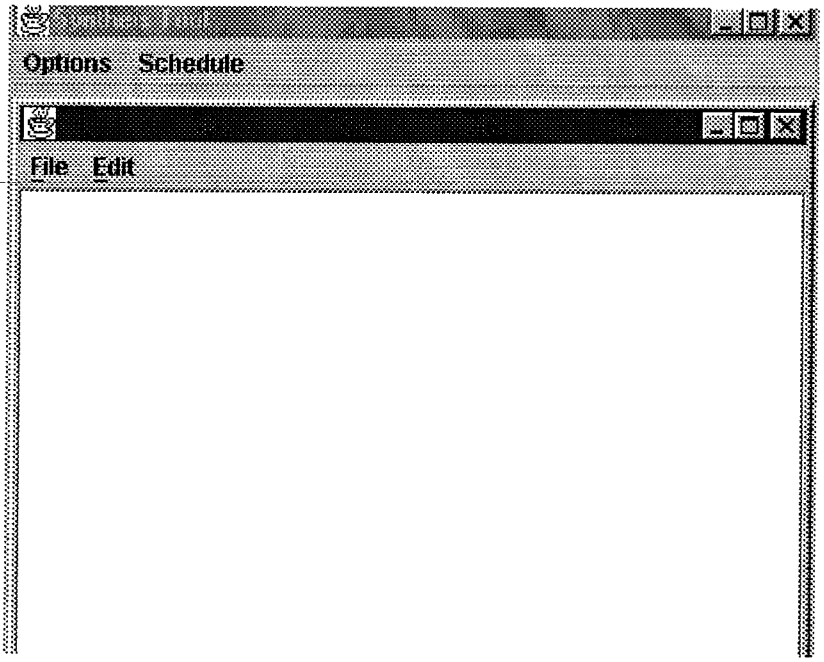
Two menus will appear the "Options" menu and the "Schedule" menu.

In the next figure we will see the options available in the "Options" menu:



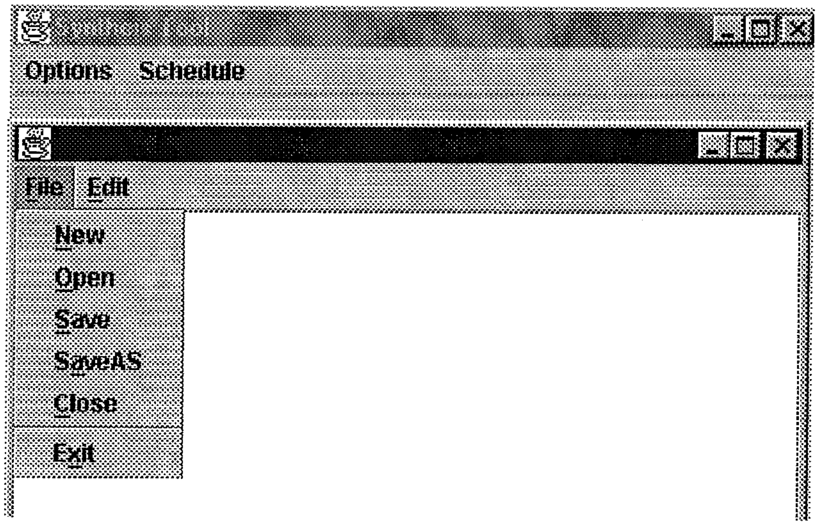
Here we can see the options available in the "Options" menu.

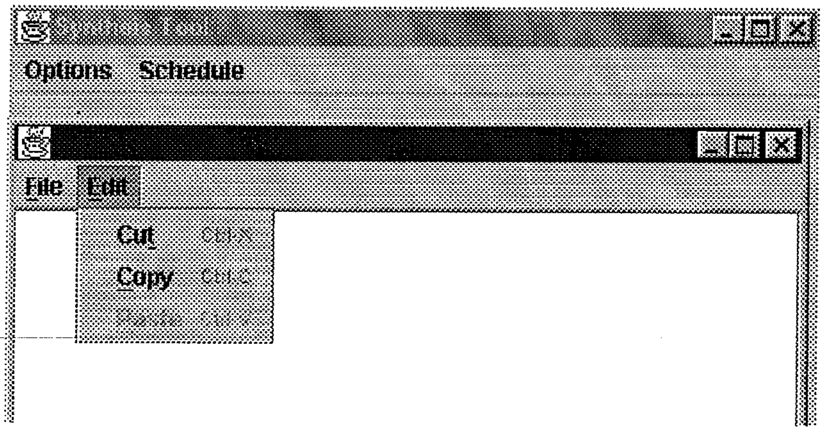
As will be shown in the next figures the first option provide the user with a text editor to edit his code that need to be synthesized and scheduled.



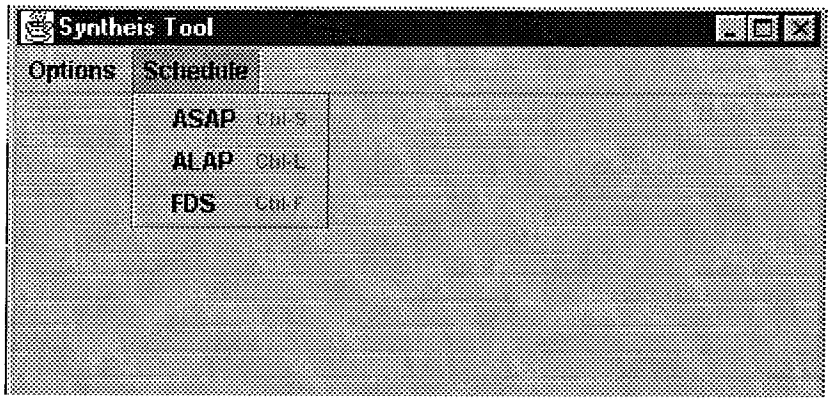
This is a text editor, which has two menus the "File" menu and the "Edit" menu. These menus almost provide the same functionalities as usual text editor similar menus.

The options of the text editor will be shown in the following two figures:



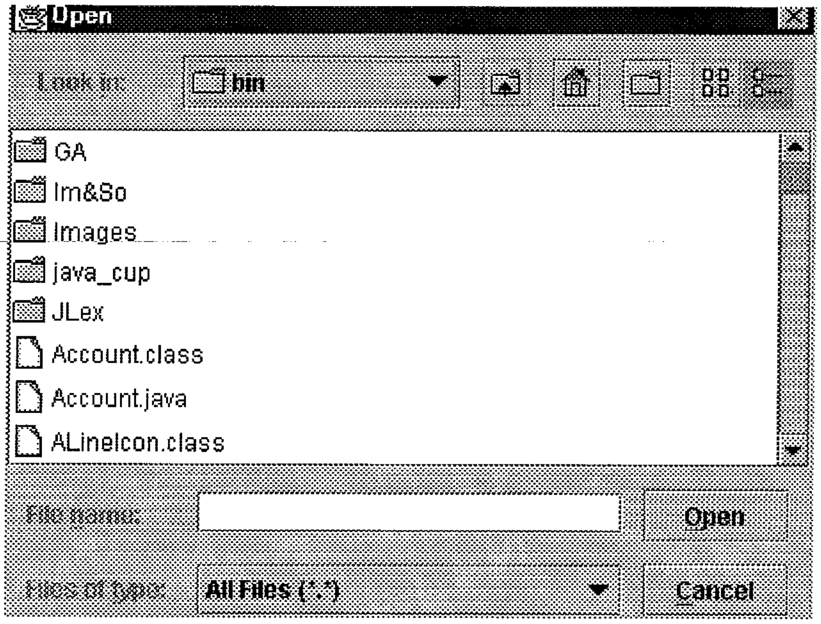


Now the next menu in the Main Synthesis tool frame is the one that do the main synthesis work before explaining what does the options in this tool I will show the options available in the "Schedule" menu in the following figure:



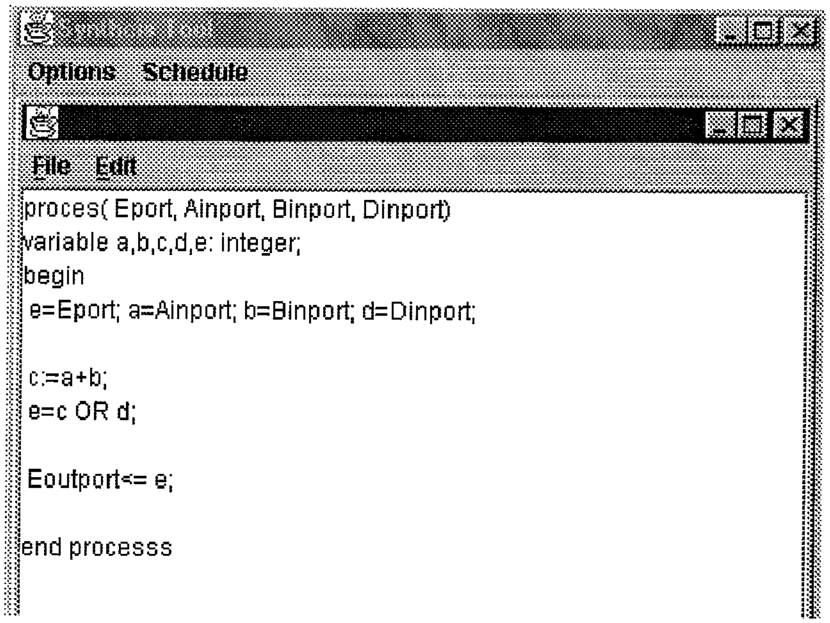
These three options are the abbreviations of the scheduling algorithms explained in chapter 3. ASAP means "As Soon As Possible", ALAP means " As Late As Possible", and FDS means " Forced Directed Scheduling".

Clicking on the ASAP, ALAP, or FDS the form shown in the following figure will appear:



This is a file chooser where a VHDL file can be chosen from any place and when Open button is cliked the file will be analyzed, parsed, and scheduled using the chosen scheduling algorithm (ASAP, ALAP, or FDS)

This is an example of an input to the tool:



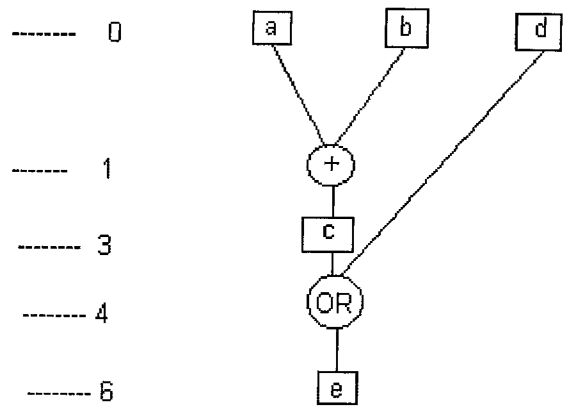
```
proces( Eport, Ainport, Binport, Dinport)
variable a,b,c,d,e: integer;
begin
 e=Eport; a=Ainport; b=Binport; d=Dinport;

 c:=a+b;
 e=c OR d;

 Eoutport<= e;

end processs
```

This is the output schedule of the above input:



Chapter 5 Experimental Results

5.1 Introduction

In this chapter I will be presenting some of the results that I got when experimenting the synthesis tool explained in chapter 4. I will test the tool with a behavioral HTML code that is written to solve a differential equation. I got this from [4].

This code will be written using the text editor in the synthesis tool and then will be fed to the tool using the ASAP algorithm to be scheduled.

To see the difference, I will present a serial schedule for the behavioral VHDL code and then another schedule got from the tool.

5.2 Experimental Results

Assumptions taken:

- Load and store operations take 1 time unit
- Add and Subtract operations take 2 time units
- Multiply operation take 4 time units
- Comparison operation takes 2 time units
- No constraints will be assumed. Resources needed are assumed to be available

Considering one iteration of the loop in the behavioral VHDL code

5.2.1 Simple Experiment

5.2.1.1 Input

Consider the following behavioral VHDL

```
Process (In1,In2)
Variable V1,V2: integer;
Begin
  V1= In1+In2;
  V2=In2+5;
  Output<=V1-V2
  End Process;
End;
```

5.2.1.2 Results

In the following table, a comparison between the serial design and the synthesis tool output design will be presented.

Serial Design time	Synthesis tool output design time
13 time units	6 time units

This shows 54% performance improvement

5.2.2 Complex Example Without Conditions

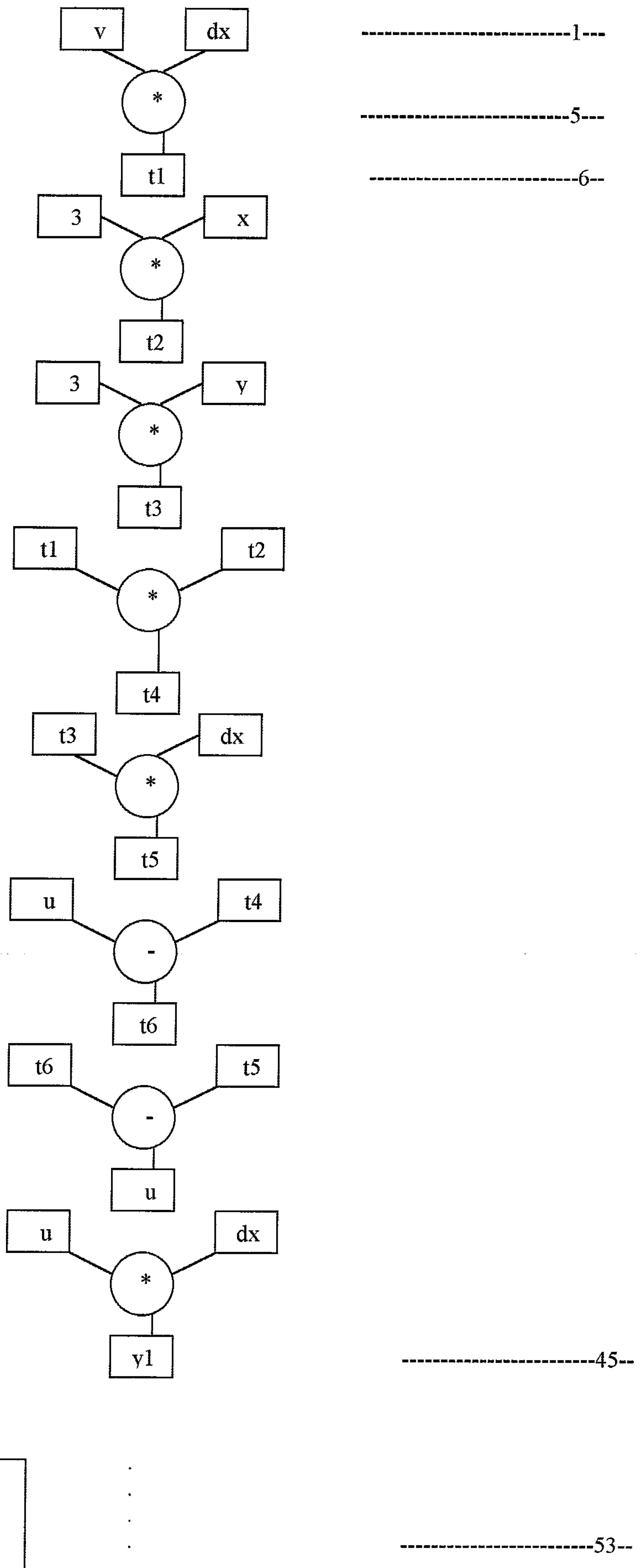
5.2.2.1 Input

The VHDL behavioral Description for solving the differential equation:

```
Process ( Aport, Dxport, Xinport, Yinport, Uinport)
variable x,y,u,a,dx: integer;
variable: x1,y1,t1,t2,t3,t4,t5,t6: integer;
Begin
x:= Xinport; y:=Yinport; u:=U inport;
a:= Aport; dx:=Dxport;
While (x<a) loop
  t1:=u*dx;
  t2:=3*x;
  t3:=3*y;
  t4:=t1*t2;
  t5:=dx*t3;
  t6:=u-t4;
  u:=t6-t5;
  y1:=u*dx;
  y:=y+y1;
  x:=x+dx
end loop;
Xoutport<=x;
Youtport<=y;
Uoutport<=u;
end process;
```

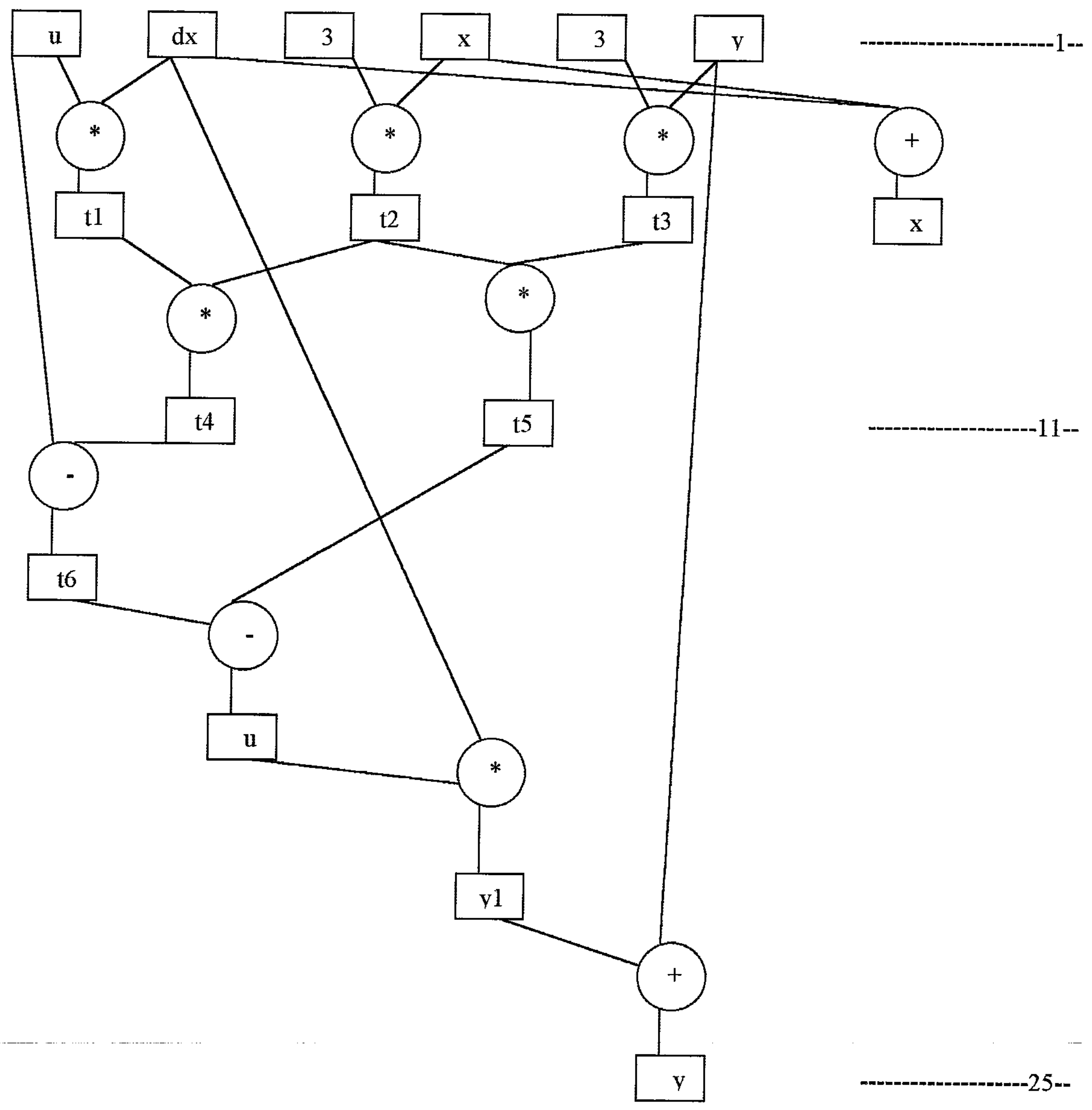
In the next pages the figures of the serial schedule and Synthesis tool output schedule will be shown.

Serial Schedule



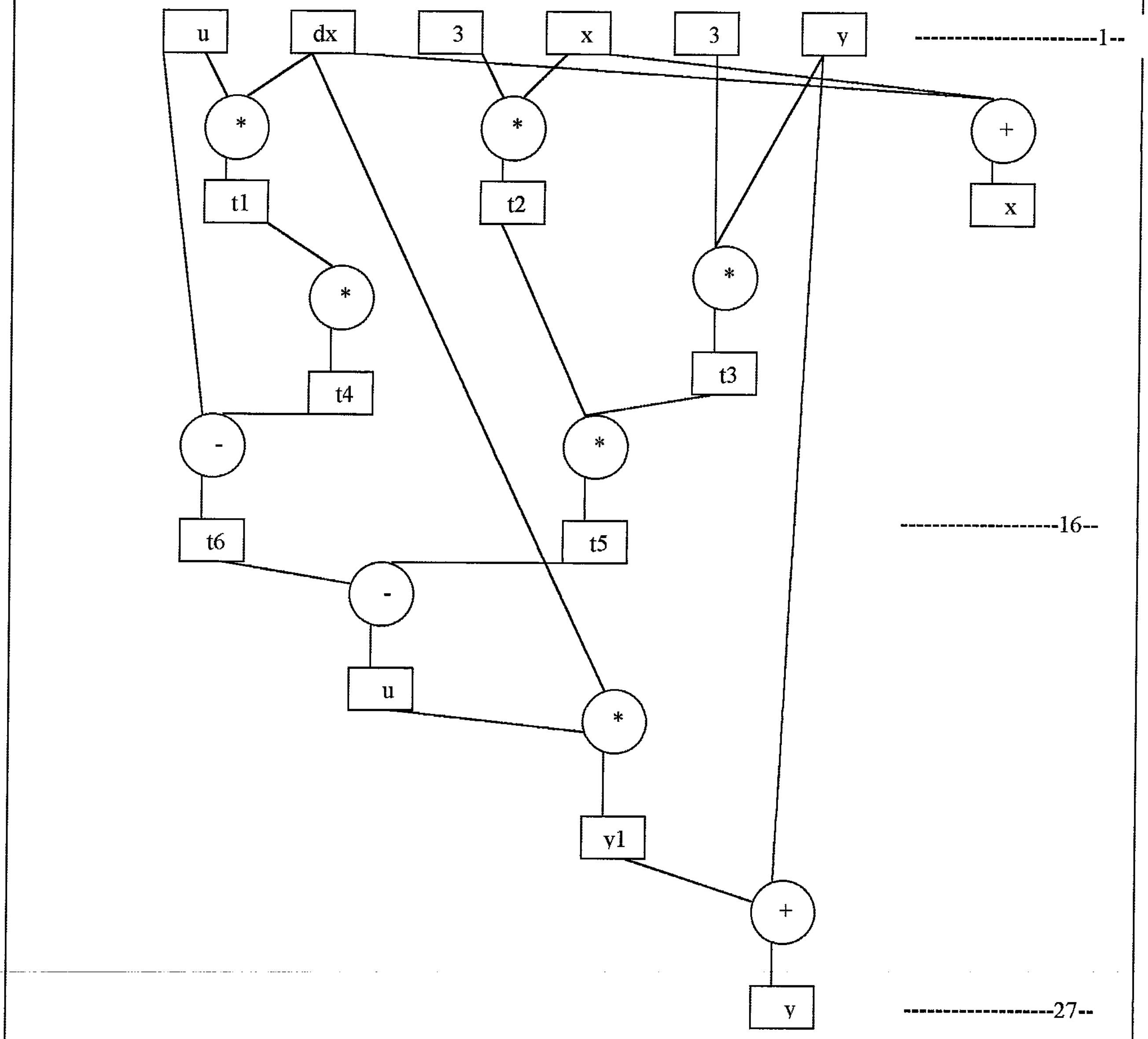
This will take 53 time
Units as shown in this
serial schedule

Synthesis Tool schedule without resource constraints



This new scheduled design will take 25 time Units

Synthesis Tool schedule with resource constraints (2 operations of each type)



This new scheduled design will take 25 time Units

5.2.2.2 Results

In the following table, a comparison between the serial design and the synthesis tool output design will be presented.

Serial Design time	Synthesis tool output design time without resource constraints	Synthesis tool output design time with resource constraints
53 time units	25 time units	27 time units

This shows 53% performance improvement without resource constraints and 49% performance improvement with resource constraints

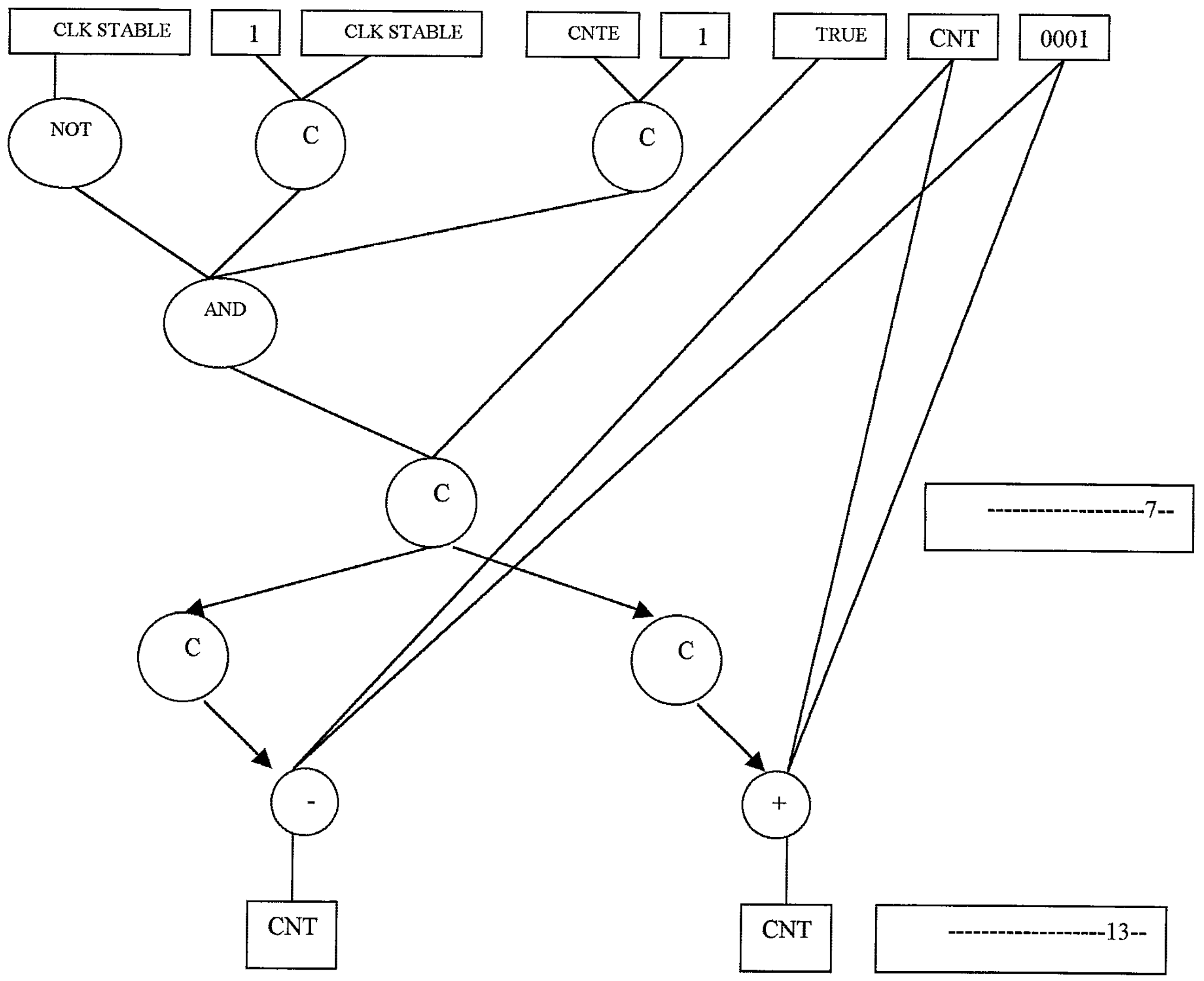
5.2.3 Complex Example With Conditions

5.2.3.1 Input

Input to the synthesis tool in this case is "ARMS_COUNTER.vhdl" a behavioral vhdl for the ARMS_COUNTER benchmark which is the Armstrong counter. It counts up or down and the counting is stops when limits are reached. It operates with the Clock and Strobe signals acting as triggers. The signal CON terminates the operation mode of the counter.

The content of the input file is found in Appendix B.

Synthesis Tool schedule(for one part, the others are done in parallel)



5.2.3.2 Results

In the following table, a comparison between the serial design and the synthesis tool output design will be presented.

Serial Design time	Synthesis tool output design time
31 time units	13 time units

This shows 58% performance improvement

5.3 Benchmarks

In this section, a table of results for testing some VHDL benchmarks will be presented. These benchmarks will be available in appendix B with their description.

Benchmark Name	Conditions	Nested Conditions	Serial Sched Time	Sched time without RC	Sched time with RC
Display.VHDL	yes	50%	51	25	38
Fancy.VHDL	yes	25%	65b	28b	40b
ARMS_Counter.VHDL	yes	30%	38	13	29
Elip.VHDL	no		152	37	61
Diffeq.VHDL	no		53((a-x)div dx +1)	25((a-x)div dx +1)	31((a-x)div dx +1)

RC= Resource Constraints

Chapter 6 Conclusion

From the examples in the previous chapter we can see the difference in performance. If we analyze the complex example (differential equation) we can see that in one iteration the first design took 53 time units (53*((a-x)div dx + 1) time units for all iterations)when manually scheduling the processes without applying the synthesis tool while the new scheduled design using the implemented synthesis tool took 25 time units without resource constraints and 27 time units with resource constrains which show 53% and 49% performance improvement respectively. The improvement is also obvious in the results regarding the benchmarks used to test the tool. Some show a very good improvement and others less. This is due to some code that cannot be parrallelized like nested conditions that must be done serially.

Off course there are some negative assumptions, like having no resources constraints in ASAP and ALAP algorithms, that might decrease our performance, however in the third algorithm FDS this was taken into consideration. On the other hand there are another positive assumptions, like considering some pipelined scheduling algorithms, that we did not also take which may almost overcome the decrease in performance. This shows what have been mentioned in chapter 1 about the importance of High-Level synthesis in hardware design and process scheduling.

What is good in this tool is the use of the three scheduling algorithms which give an efficient and good design of a behavioral description.

In this thesis, a good research has been done to show the importance of synthesis tools in hardware design nowadays. Moreover, a synthesis tool have been implemented for many reasons:

- To have and give a better knowledge of synthesis tools describe in detail how they operate.
- Have a good knowledge of each step have been followed so that future work to produce better tools, will be in considering each of these steps alone and make it better.
- This tool can be adapted in the future to be applied for procedural languages.
- Using Java, the object oriented programming language, to implement such tool by taking advantage of object orientation and java features in GUI and other things.

References

[1]Ahmed Amine Jerraya, Hong Ding, Polen Kission, Maher, Rahmouni. "*Behavioral Synthesis Component Reuse With VHDL*". Kluwer Academic Publishers. Boston, 1997.

[2]Andrew w. Appel. "*Modern Compiler Implementation in Java*". Cambridge University Press, 1998.

[3]De Michelli. "*Synthesis and Optimization of Digital Circuits*". McGraw-Hill, 1994.

[4]H. Harmanani, H. Halawi. " *A Synthesis for Test System using VHDL*". Lebanese American University, 2001.

[5]Ju Hwan Yi, Hoon Choi, In-Cheol Park, Seung Ho, Hwang, Chong-Min Kyung. "*Multipe Behavior Synthesis Based on Selective Groupings*". Dept. of EE, KAIST, Taejon, Korea.

[6]Kavin O'Brein, Maher Rahmouni, Ahmed Jerraya." *A VHDL-Based Scheduling Algorithm For Control-Flow Dominated Circuits*". Laboratoire TIM3/INPG. pp. 135-145. France.

[7]LaNae J. Avra, Laurent Gerbaux, Jean-Charles Giomi, Francoise Martinolle, Edward J. McCluskey. "*Synthesis for Test Design System*". Stanford University. Stanford, California, May 1994.

[8]Longanath Ramachandran, Nancy D. Holme, Daniel D. Gajski."*The Design Process for behavioral Synthesis from VHDl*". University of California, February 1994.

[9]Minjoong Rim, Rajvi Jain. " *Lower-Bound Performance Estimation for the High-Level synthesis Scheduling Problem*". pp. 451-458. IEEE, 1994.

[10]Piere G. Paulin, John P. Knight. "*Force-Directed Scheduling for Behavioral Synthesis of ASIC's*" . pp. 661-679. IEEE, 1989.

[11]Vijay Nagasamy, Neerav Berry, Carlos Dangelo." Specification, Planning, and Synthesis in a VHDL design Environment". pp. 58-68. IEEE, 1992.

[12]S. Narayan, F. Vahid, D.D. Gajski." *Incorporating VHDL Signal/Wait Semantics into Synthesis*". University of California. California, April, 1992.

[13] Taewhan Kim, Noritake, Jane Liu, C. Liu. " A Scheduling Algorithm for Conditional Resource Sharing". vol 13, No. 4,pp. 425-438, IEEE Transactions 1994

[14] "*Architectural Synthesis via VHDL*".

Appendix A

Specification files for analysis and parsing

A1.The following is my specification input file for my lexical analyzer:

```
/* User code*/

import java.awt.*;
import java.awt.event.*;
import java.io.*;

%%

/* Jlex directives  */

%{
private final int NUMIDENTS=19;

%}
%char
%line
%function nextToken
%type java_cup.runtime.Symbol
%class LexAna
%function Tokenize
%notunix
%ignorecase

alpha=[a-zA-Z]
digits=[0-9]
underbar=[_]
alphau=[a-zA-Z_]
alphanu=[a-zA-Z_0-9]

signs=[+-]
minus=[-]
beq= ":="
pas= "=>"
ipas= [<][=]

eoln=[\n]
white=[ \n\t]

other= .
dot= "."
quote=[\"]
squote=[']

name= [a-zA-Z_]({dot}?[a-zA-Z0-9_])?
```

```
string= [\"][other]*[\"]
number= {digits}+
float= [+-]?{number}({dot}{number})?([Ee][+-]?{number})?
comment="--".*"\n"

%%

/* regular expression rules  */

ARCHITECTURE {return new java_cup.runtime.Symbol(1,"ARCHITECTURE");}
BEGIN {return new java_cup.runtime.Symbol(2,"BEGIN");}
COMPONENT {return new java_cup.runtime.Symbol(3,"COMPONENT");}
DOWNTO {return new java_cup.runtime.Symbol(4,"DOWNTO");}
END {return new java_cup.runtime.Symbol(5,"END_");}
ENTITY {return new java_cup.runtime.Symbol(6,"ENTITY");}
GENERIC {return new java_cup.runtime.Symbol(7,"GENERIC");}
IF {return new java_cup.runtime.Symbol(8,"IF");}
IN {return new java_cup.runtime.Symbol(9,"IN");}
INOUT {return new java_cup.runtime.Symbol(10,"INOUT");}
IS {return new java_cup.runtime.Symbol(11,"IS");}
LIBRARY {return new java_cup.runtime.Symbol(12,"LIBRARY");}
MAP {return new java_cup.runtime.Symbol(13,"MAP");}
OF {return new java_cup.runtime.Symbol(14,"OF");}
OUT {return new java_cup.runtime.Symbol(15,"OUT");}
PORT {return new java_cup.runtime.Symbol(16,"PORT");}
SIGNAL {return new java_cup.runtime.Symbol(17,"SIGNAL");}
TO {return new java_cup.runtime.Symbol(18,"TO");}
USE {return new java_cup.runtime.Symbol(19,"USE");}
NOT {return new java_cup.runtime.Symbol(20,"NOT");}
AND {return new java_cup.runtime.Symbol(21,"AND");}
OR  {return new java_cup.runtime.Symbol(22,"OR");}
```

A2. An example of a specification file is the following:

```
/*imported libraries to be used in the generated parser class*/
import java_cup.runtime.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

/* Preliminaries to set up and use the scanner */
init with {: scanner.init();              :};
scan with {: return scanner.next_token(); :};

/*terminals*/
terminal Entity;
terminal IS;
terminal END_;
terminal GENERIC;
terminal PORT;
terminal DOWNTO;
terminal TO;
terminal ARCHITECTURE;
terminal OF;
terminal BEGIN_ ;
terminal COMPONENT;
terminal IN;
terminal OUT;
terminal IF;
terminal MAP;
terminal SIGNAL;
terminal LIBRARY;
terminal USE;
terminal INOUT;
terminal AND;
terminal OR;
terminal NOT;

/*non terminals*/
non terminal Identifier;
non terminal VectorType;
non terminal port_signal;
non terminal Bit;
non terminal declaration, declarations,entity_declaration,
architecture_declaration,library_declaration,use_statement;

/*An example of a Grammer*/
declarations ::= declaration
               |
               declarations declaration;

declaration ::= entity_declaration
              |
              architecture_declaration
              |
              library_declaration
              |
              use_statement;
```

Appendix B
Scheduler Implementation

B1. Scheduler Implementation

The following is the Java class representing the code in the intermediate language form and the scheduler use instances of these classes to represent the nodes of the final schedule.

```
public class gnode{
   String name;
   int succ[],  //array of successors
       pred[];  //array of predecessors
   int dur,     //duration
       st,      //schedul time
       type,    //node type 0 initially, 1 for operation, 2 for
                //operands
       succNo,
       predNo;


   public gnode(String na,int d)
   {
     name= new String(na);
     succ=new int[50];
     pred=new int[50];
     dur=d;
     st=-1;
     succNo=0;
     predNo=0;
     type=0;
   }

   //write a pointer to a node in another node to be its pred

    public void AddPreds(int i)
    {
      pred[predNo]=i;
      predNo++;
    }

   //write a pointer to a node in another node to be its pred


    public void AddSuccs(int i)
    {
      succ[succNo]=i;
      succNo++;
    }

    public int[] GetSucc() // gets the successors of n
    {
     int m[]=new int[50];
     int i;

     for(i=0;i<succNo;i++)
       m[i]=succ[i];

     return m;
    }

    public int[] Getpred() //gets the predecessors of n
    {
     int m[]=new int[50];
     int i;

     for(i=0;i<predNo;i++)
       m[i]=pred[i];

     return m;
    }
 }
```

The following is the scheduler implementation in a Java Class:

```
public class sgraph {

 gnode gn[]=new gnode[100];
 static int nodeNo=0;

 public int GetMaxTimePred(gnode n) //gets the max time of the pred
 {
  int pred[],i,max=0;
  pred=new int[50];
  pred=n.Getpred();
  for(i=0;i<n.predNo;i++)
  {
   if ( max < gn[pred[i]].st+gn[pred[i]].dur ) max=
gn[pred[i]].st+gn[pred[i]].dur;
  }
  return max;
 }

 public int GetMinTimeSucc(gnode n,int lat) //gets the min time of
the succ
 {
  int succ[],i,min=lat+1;
  succ=new int[50];
  succ=n.GetSucc();
  for(i=0;i<n.succNo;i++)
  {
    if ( min >= gn[succ[i]].st ) min= gn[succ[i]].st;
  }
  return min;
 }

 public boolean PredAreSched(gnode n) //test if predecessors are
scheduled
 {
  boolean b=true;
  int pred[],i;
  pred=new int[50];
```

```
   pred=n.Getpred();
   for(i=0;i<n.predNo;i++)
    if(gn[pred[i]].st==-1) b=false;
   return b;
  }

  public boolean SuccAreSched(gnode n) //test if successors are
 scheduled
  {
   boolean b=true;
   int succ[],i;
   succ=new int[50];
   succ=n.GetSucc();
   for(i=0;i<n.succNo;i++)
    if(gn[succ[i]].st==-1) b=false;
   return b;
  }

  public void AddNode(String str,int d) // adds a node of name s &
 duration d to the graph
  {
    gn[nodeNo]=new gnode(str,d);
    nodeNo++;
  }

  public int GetIndex(String s)
  {
    int i=0;
    int j=-1;

    while (i<nodeNo)
    {
    if ( gn[i].name.compareTo(s)==0)   j=i;
    i++;
    }
    return j;
  }

  public boolean SchASAP(gnode n) //schedules a node if possible using
 ASAP
  {
    boolean b=false;
    if (n.predNo==0)
    {
      n.st=1;
      b=true;
    }
    else if (PredAreSched(n))
    {
     n.st=GetMaxTimePred(n);
     b=true;
    }
    return b;
  }

  public boolean SchALAP(gnode n,int lat) //schedules a node if
 possible using ALAP
  {
    boolean b=false;
    if(n.succNo==0)
    {
     n.st=lat+1;
     b=true;
   }
   else if (SuccAreSched(n))
   {
    n.st=GetMinTimeSucc(n,lat)-n.dur;
    b=true;
   }
   return b;
 }

// as soon as posible algorithm (unconstrained scheduling)
public int[] ASAP()
{
 int l[]=new int[nodeNo];
 int count=1;
 int count1=1;
 boolean AllSched=false;

 // schedule Vo the 1st node
 gn[0].st=1;

 //for all Vi's schedule Vi if all preds are scheduled
 while(! AllSched)
 {
  if(gn[count1].st==-1)
     if(SchASAP(gn[count1]))
        count++;

  if(count==nodeNo) AllSched=true;

  count1=(count1+1) % nodeNo;

 }
 for(int i=0;i<nodeNo;i++) l[i]=gn[i].st;
 return l;
}

// as late as possible algorithm ( latency constrained)
public int[] ALAP(int lat)
{
 int l[]=new int[nodeNo];
 int count=1;
 int count1=nodeNo-2;
 boolean AllSched=false;

 // schedule Vn the last node t=lat+1
 gn[nodeNo-1].st=lat+1;

 //for all Vi's schedule Vi if all succs are scheduled
 while(! AllSched)
 {
  if(SchALAP(gn[count1],lat))
   count++;
  if(count==nodeNo) AllSched=true;
  count1=count1-1;
  if (count1==-1) count1=nodeNo-2;
 }

 for(int i=0;i<nodeNo;i++) l[i]=gn[i].st;
 return l; }
```

```
}
```

Appendix C

Benchmarks

This Appendix contain the VHDL code of the Benchmarks used to test the synthesis tool:

C1. Fancy.vhdl

The FANCY benchmark was created to illustrate the RTL optimization
Process.
"RT-Level Transformations for Gate Level Testability

```
--**VHDL**************************************************************
--
-- SRC-MODULE : FANCY
-- NAME       : fancy.vhdl
-- VERSION    : 1.0
--
-- PURPOSE    : Architecture of FANCY benchmark
--
-- LAST UPDATE: Wed May 19 13:03:48 MET DST 1993
--
--********************************************************************
--
-- Architecture of FANCY
--

PACKAGE types IS
  SUBTYPE nat8 is integer RANGE 0 TO 255;
END types;

USE work.types.all;

ENTITY fancy IS
PORT(reset    : IN bit;          -- Global reset
     clk      : IN bit;          -- Global clock
     startinp : IN boolean;
     ainp     : IN nat8;
     binp     : IN nat8;
     cinp     : IN nat8;
     eoc      : OUT boolean;
     f        : OUT nat8);
END fancy;

ARCHITECTURE algorithm OF fancy IS
BEGIN
  fancy: PROCESS
    VARIABLE   temp1a, temp1b, temp3, temp4, temp6a : nat8;
    VARIABLE   a, b, c, counter : nat8;
    VARIABLE   start            : boolean;
  BEGIN

    eoc <= true;
    f <= 0;

  RESET_LOOP : LOOP
    WAIT UNTIL clk = '1'; EXIT RESET_LOOP WHEN (reset = '1');
    a := ainp;
    b := binp;
    c := cinp;
    start := startinp;

    temp1a  := 0;
    counter := 0;
    eoc <= false;
    WHILE (counter < b) LOOP
        IF (a <= counter) THEN
            IF (a <= counter) THEN
                temp6a := b;
            ELSE
                temp6a := temp1a;
            END IF;
        ELSE
            temp6a := a;
        END IF;

        IF ((temp1a = 0) XOR (a > counter) XOR (a <= counter) XOR
start) THEN
            temp1b := c;
        ELSE
            IF (a > b) THEN
                temp1b := a;
            ELSE
                temp1b := b;
            END IF;
        END IF;

        IF start THEN
            temp4 := temp1b;
        ELSE
            temp4 := temp6a;
        END IF;

        IF (start XOR (a > b)) THEN
            IF (b > c) THEN
                temp3 := c;
            ELSE
                temp3 := b;
            END IF;
        ELSE
            temp3 := temp1b;
        END IF;

        IF ((temp1a = 0) XOR (a > counter) XOR (a <= counter)) THEN
            temp1a := temp4 + temp3;
        ELSE
            IF ((a > b) XOR (b > c)) THEN
                temp1a := temp4 + temp1a;
            ELSE
                temp1a := temp4 + a;
            END IF;
        END IF;
        counter := counter + 1;
        WAIT UNTIL clk = '1'; EXIT RESET_LOOP WHEN (reset = '1');
    END LOOP;

    f   <= templa;
    eoc <= true;

  END LOOP RESET_LOOP;

  END PROCESS fancy;
END algorithm;
```

C2. ARMS_COUNTER.vhdl

The Armstrong counter counts up or down and the counting is stops when limits are reached. It operates with the Clock and Strobe signals acting as triggers. The signal CON terminates the operation mode of the counter.

```
----------------------------------------------------------------------
-----------
--
-- Controlled Counter Benchmark
--
-- Source: "Chip Level Modeling with VHDL" by Jim Armstrong
(Prentice-Hall 1989)
--
-- Benchmark author: Joe Lis
--
--    Copyright (c) by Joe Lis 1988
--
-- Modified by : Champaka Ramachandran on Aug 24th 1992
--
-- Verification Information:
--
--                  Verified  By whom?                Date
Simulator
--                  --------  ------------            --------    -------
-----
--  Syntax           yes      Champaka Ramachandran  24/8/92     ZYCAD
--  Functionality    yes      Champaka Ramachandran  24/8/92     ZYCAD
----------------------------------------------------------------------
-----------

use work.BIT_FUNCTIONS.all;

entity ARMS_COUNTER is
  port (
        CLK: in BIT;
        STRB : in bit;
        CON: in BIT_VECTOR(1 downto 0);
        DATA: in BIT_VECTOR(3 downto 0);
        COUT: out BIT_VECTOR(3 downto 0));

end ARMS_COUNTER;

--VSS: design_style behavioural

architecture ARMS_COUNTER of ARMS_COUNTER is

   signal ENIT, RENIT: BIT;
   signal EN: BIT;
   signal CONSIG, LIM: BIT_VECTOR(3 downto 0);
   signal CNT : BIT_VECTOR(3 downto 0);

begin

----------------  The decoder -------------------------------------

DECODE: process (STRB, RENIT)

variable CONREG: BIT_VECTOR(1 downto 0) := "00";

begin

  if (STRB = '1') and (not STRB'STABLE) then
    CONREG := CON;

    case CONREG is
         when "00" => CONSIG <= "0001";
         when "01" => CONSIG <= "0010";
         when "10" => CONSIG <= "0100"; ENIT <= '1';
         when "11" => CONSIG <= "1000"; ENIT <= '1';
         when others =>
    end case;

  end if; -- Rising edge of STRB

  if (RENIT = '1') and (not RENIT'STABLE) then
    ENIT <= '0';
  end if;

end process DECODE;


----------------  The limit loader ----------------------------------
---

LOAD_LIMIT: process (STRB)

begin

  if (CONSIG(1) = '1') and (not STRB'STABLE) and (STRB = '0') then
    LIM <= DATA;
  end if;

end process LOAD_LIMIT;


----------------  The counter -------------------------------------

CTR: process (CONSIG(0), EN, CLK)

variable CNTE : BIT := '0';

begin

  if (CONSIG(0) = '1') and (not CONSIG(0)'STABLE) then
    CNT <= "0000";
  end if;

  if (not EN'STABLE) then
```

```
    if (EN = '1') then
      CNTE := '1';
    else
      CNTE := '0';
    end if;
  end if;

  if (not CLK'STABLE) and (CLK = '1') and (CNTE = '1') then
    if (CONSIG(2) = '1') then
      CNT <= CNT + "0001";
    elsif (CONSIG(3) = '1') then
          CNT <= CNT - "0001";
    end if;
  end if;

end process CTR;

---------------- The comparator ------------------------------------
-

LIMIT_CHK: process (CNT, ENIT)

begin

  if (not ENIT'STABLE) then
    if (ENIT = '1') then
      EN <= '1'; RENIT <= '1';
    else
      RENIT <= '0';
    end if;
  end if;

  if (EN = '1') and (CNT = LIM) then
    EN <= '0';
  end if;

end process LIMIT_CHK;

COUT <= CNT;

end ARMS_COUNTER;
```

C3. Display.vhdl

The DISPLAY chip is a driver for four seven-segment LED displays. The display units display ten minutes, minutes, ten seconds and seconds. After reaching 59 59, they reset on the next clock to 00 00.

```
ENTITY display IS
PORT(reset  : IN bit;                       -- Global reset
     clk    : IN bit;                       -- Global clock
     en     : IN boolean;
     unit0 : OUT bit_vector(6 DOWNTO 0);
     unit1 : OUT bit_vector(6 DOWNTO 0);
     unit2 : OUT bit_vector(6 DOWNTO 0);
     unit3 : OUT bit_vector(6 DOWNTO 0));
END display;
```

```
ARCHITECTURE algorithm OF display IS

  SUBTYPE nat4 is integer RANGE  15 DOWNTO 0;
  SUBTYPE nat3 is integer RANGE   7 DOWNTO 0;

BEGIN
  display: PROCESS
    VARIABLE   secs,  mins   : nat4; -- counters for seconds and
minutes
    VARIABLE   tsecs, tmins : nat3; -- counters for ten seconds and
ten minutes
  BEGIN

    -- Initialization

    secs  := 0;
    tsecs := 0;
    mins  := 0;
    tmins := 0;
    unit0 <= "1000000";
    unit1 <= "1000000";
    unit2 <= "1000000";
    unit3 <= "1000000";

    RESET_LOOP: LOOP
    WAIT UNTIL clk = '1'; EXIT RESET_LOOP WHEN reset = '1';

-- decoder part of the display circuit:
--
--                     0                               6543210
--                  -------                        0 :  1000000
--                 |       |       unitX(6..0)     1 :  1111001
--                5|       |1                      2 :  0100100
--                 |   6   |                       3 :  0110000
--                  -------                        4 :  0011001
--                 |       |                       5 :  0010010
--                4|       |2                      6 :  0000010
--                 |       |                       7 :  1111000
--                  -------                        8 :  0000000
--                     3                           9 :  0010000
--             0=light, 1=dark!
--

    CASE secs IS
        WHEN 0 => unit0 <= "1000000";
        WHEN 1 => unit0 <= "1111001";
        WHEN 2 => unit0 <= "0100100";
        WHEN 3 => unit0 <= "0110000";
        WHEN 4 => unit0 <= "0011001";
        WHEN 5 => unit0 <= "0010010";
        WHEN 6 => unit0 <= "0000010";
        WHEN 7 => unit0 <= "1111000";
        WHEN 8 => unit0 <= "0000000";
        WHEN 9 => unit0 <= "0010000";
        WHEN others => unit0 <= "0000000";
    END CASE;

    CASE tsecs IS
        WHEN 0 => unit1 <= "1000000";
        WHEN 1 => unit1 <= "1111001";
        WHEN 2 => unit1 <= "0100100";
        WHEN 3 => unit1 <= "0110000";
        WHEN 4 => unit1 <= "0011001";
        WHEN 5 => unit1 <= "0010010";
        WHEN others => unit1 <= "0000000";
    END CASE;

    CASE mins IS
        WHEN 0 => unit2 <= "1000000";
        WHEN 1 => unit2 <= "1111001";
        WHEN 2 => unit2 <= "0100100";
        WHEN 3 => unit2 <= "0110000";
        WHEN 4 => unit2 <= "0011001";
        WHEN 5 => unit2 <= "0010010";
        WHEN 6 => unit2 <= "0000010";
        WHEN 7 => unit2 <= "1111000";
        WHEN 8 => unit2 <= "0000000";
        WHEN 9 => unit2 <= "0010000";
        WHEN others => unit2 <= "0000000";
    END CASE;

    CASE tmins IS
        WHEN 0 => unit3 <= "1000000";
        WHEN 1 => unit3 <= "1111001";
        WHEN 2 => unit3 <= "0100100";
        WHEN 3 => unit3 <= "0110000";
        WHEN 4 => unit3 <= "0011001";
        WHEN 5 => unit3 <= "0010010";
        WHEN others => unit3 <= "0000000";
    END CASE;

    IF en THEN
      IF (secs = 9) THEN
        secs  := 0;
        IF tsecs = 5 THEN
            tsecs  := 0;
            IF mins = 9 THEN
                mins  := 0;
                IF tmins = 5 THEN
                    tmins := 0;
                ELSE
                    tmins := tmins + 1;
                END IF;
            ELSE
                mins  := mins + 1;
            END IF;
        ELSE
            tsecs  := tsecs + 1;
        END IF;
      ELSE
        secs  := secs + 1;
      END IF;
    END IF;

  END LOOP RESET_LOOP;

  END PROCESS display;
END algorithm;
```

C4. Ellip.vhdl

The elliptic filter has been used as a benchmark for many architectural synthesis packages, and is a part of the high-level synthesis benchmark suite. The elliptic filter belongs to the class of infinite Impulse Response (IIR) filters, because its response to an impulse input remains non-zero till infinite time in a theoretical sense. The particular filter we deal with here is a low pass filter, meaning that it filters off frequencies higher than a certain limit, called the cut-off frequency.

```
---------------------------------------------------------------------
-------
--
--
--                    Elliptical Wave Filter Benchmark
--
--
-- VHDL Benchmark author: D. Sreenivasa Rao
--                        University Of California, Irvine, CA 92717
--                        dsr@balboa.eng.uci.edu, (714)856-5106
--
-- Developed on 12 September, 1992
--
-- Verification Information:
--
--                   Verified     By whom?          Date
Simulator
--                   --------   ------------      --------      -----
-------
--  Syntax             yes          DSR           09/12/92
ZYCAD
--  Functionality      yes          DSR           09/12/92
ZYCAD
---------------------------------------------------------------------
----------

--use std.std_logic.all;
use work.bit_functions.all;

entity ellipf is
    port ( inp : in BIT_VECTOR(15 downto 0);
           outp : out BIT_VECTOR(15 downto 0);
           sv2, sv13, sv18, sv26, sv33, sv38, sv39 :
                 in BIT_VECTOR(15 downto 0);
           sv2_o, sv13_o, sv18_o, sv26_o, sv33_o, sv38_o, sv39_o :
                 out bit_vector(15 downto 0));
end ellipf;

architecture ellipf of ellipf is

begin

process (inp, sv2, sv13, sv18, sv26, sv33, sv38, sv39)

--  constant m1, m2, m3, m4, m5, m6, m7, m8 : integer :=
(1,1,1,1,1,1,1,1);
```

```
    variable n1, n2, n3, n4, n5, n6, n7 : BIT_VECTOR(15 downto 0);
    variable n8, n9, n10, n11, n12, n13 : BIT_VECTOR(15 downto 0);
    variable n14, n15, n16, n17, n18, n19 : BIT_VECTOR(15 downto 0);
    variable n20, n21, n22, n23, n24, n25 : BIT_VECTOR(15 downto 0);
    variable n26, n27, n28, n29 : BIT_VECTOR(15 downto 0);
--     constant i : integer := (1);

 begin
  while (i = 1) LOOP
    n1 := inp + sv2;
    n2 := sv33 + sv39;
    n3 := n1 + sv13;
    n4 := n3 + sv26;
    n5 := n4 + n2;
    n6 := n5 ;
    n7 := n5 ;
    n8 := n3 + n6;
    n9 := n7 + n2;
    n10 := n3 + n8;
    n11 := n8 + n5;
    n12 := n2 + n9;
    n13 := n10 ;
    n14 := n12 ;
    n15 := n1 + n13;
    n16 := n14 + sv39;
    n17 := n1 + n15;
    n18 := n15 + n8;
    n19 := n9 + n16;
    n20 := n16 + sv39;
    n21 := n17 ;
    n22 := n18 + sv18;
    n23 := sv38 + n19;
    n24 := n20 ;
    n25 := inp + n21;
    n26 := n22 ;
    n27 := n23 ;
    n28 := n26 + sv18;
    n29 := n27 + sv38;
    sv2_o <= n25 + n15;
    sv13_o <= n17 + n28;
    sv18_o <= n28;
    sv26_o <= n9 + n11;
    sv38_o <= n29;
    sv33_o <= n19 + n29;
    sv39_o <= n16 + n24;
    outp <= n24;
   end LOOP;
 end process;

end ellipf;

--configuration ellipcon of ellipf is
--  for ellip_beh
--  end for;
--end ellipcon;
```