HDL-LEVEL PARTITIONING OF CIRCUITS

by

Juan Humberto Rico Romo

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright by Juan Humberto Rico Romo 2001
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.
Abstract

HDL-Level Partitioning of Circuits

Juan Humberto Rico Romo

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2001

The development of the Transmogrifier 3 (TM-3) at the University of Toronto has resulted in a multi-FPGA system that could be used for logic emulation. To make it useful requires the development of a partitioning tool.

This thesis proposes manual partitioning of designs at their high-level language representation i.e. VHDL-language representation. This approach allows the designer to choose in an easy manner in which partition a component is to be placed. Consequently, should the design not behave/perform as desired, the VHDL files describing the corresponding components can be easily identified among all the other VHDL source files in the design.

The resulting partitioning tool is capable of flattening hierarchical designs, thus reducing the granularity of the netlist representing the design. This partitioning tool generates synthesisable VHDL code for each partition, and considers the TM-3 topology. To limit the scope of this work, only a subset of the complex VHDL language is supported.
Acknowledgements

I would like to thank my supervisor Professor Paul Chow for his support and guidance to accomplish the work presented in this thesis.

I also would like to thank my parents and my wife for their support and encouragement to pursue graduate studies. To my wife, I would like to give special thanks for her love and patience throughout this experience.

Finally, I would like to express my gratitude to Mrs. Sarah Cherian for her willingness to help me during special circumstances.
# Contents

Abstract .......................... ii

Acknowledgments ................. iii

List of Figures .................... x

1 Introduction ...................... 1

   1.1 Motivation ........................ 1

   1.2 Thesis Overview ................... 2

   1.3 Thesis Organisation .............. 5

2 Background Information .......... 7

   2.1 Multi-FPGA Systems ............... 7

      2.1.1 Circuit Partitioning Techniques .... 9

      2.1.2 Network Topologies ............. 10

      2.1.3 Transmogrifier 3 ............... 12

      2.1.3.1 Topology .................... 12

   2.2 VHDL Quick Reference .......... 14

      2.2.1 Port Signals ................... 15
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.1.1</td>
<td>Signal Mode</td>
<td>16</td>
</tr>
<tr>
<td>2.2.1.2</td>
<td>Signal Type</td>
<td>17</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Internal Signals</td>
<td>18</td>
</tr>
<tr>
<td>2.2.3</td>
<td>VHDL Subcircuits</td>
<td>18</td>
</tr>
<tr>
<td>2.2.3.1</td>
<td>Component Declarations</td>
<td>18</td>
</tr>
<tr>
<td>2.2.3.2</td>
<td>Instantiated Components</td>
<td>19</td>
</tr>
<tr>
<td>2.2.3.3</td>
<td>Hierarchical Designs and Levels of Abstraction</td>
<td>20</td>
</tr>
<tr>
<td>2.2.4</td>
<td>Other VHDL Constructs</td>
<td>23</td>
</tr>
<tr>
<td>3</td>
<td>VHDL-Design Analysis</td>
<td>24</td>
</tr>
<tr>
<td>3.1</td>
<td>Lexical and Syntactical Analysis</td>
<td>24</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Supported VHDL</td>
<td>25</td>
</tr>
<tr>
<td>3.2</td>
<td>Intermediate Representation</td>
<td>30</td>
</tr>
<tr>
<td>3.3</td>
<td>Graph Implementation</td>
<td>31</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Architecture and Entity Signals as Graph Nodes</td>
<td>31</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Node Set Definitions</td>
<td>34</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Properties of the Graph</td>
<td>35</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Component Declarations and Direction of Edges</td>
<td>37</td>
</tr>
<tr>
<td>4</td>
<td>VHDL-code Partitioning</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>39</td>
</tr>
<tr>
<td>4.2</td>
<td>Graph Partitioning</td>
<td>40</td>
</tr>
<tr>
<td>4.3</td>
<td>Signal Dependencies</td>
<td>41</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Same-Partition Signal Dependencies</td>
<td>43</td>
</tr>
</tbody>
</table>
C VHDL Source Codes And Schematics 98

C.1 First VHDL Design .......................................................... 98
  C.1.1 VHDL Source Code ...................................................... 98
  C.1.2 Schematic Diagram .................................................. 100
  C.1.3 VHDL Source Codes After Partitioning .......................... 101
    C.1.3.1 Partition One .................................................. 101
    C.1.3.2 Partition Two .................................................. 103
  C.1.4 Schematic Diagrams After Partitioning ....................... 105
    C.1.4.1 Partition One .................................................. 105
    C.1.4.2 Partition Two .................................................. 106

C.2 Alarm Clock Implementation ........................................... 107
  C.2.1 Top-Level VHDL File ................................................ 107
  C.2.2 Top-Level Schematic Diagram .................................... 109
  C.2.3 Flattened Top-Level VHDL File .................................. 110
  C.2.4 Schematic Diagram for Flattened Top-Level File ............ 114
  C.2.5 VHDL Source Codes After Partitioning .......................... 115
    C.2.5.1 Partition One .................................................. 115
    C.2.5.2 Partition Two .................................................. 118
    C.2.5.3 Partition Three ............................................... 121
    C.2.5.4 Partition Four ............................................... 124
  C.2.6 Schematic Diagrams After Partitioning .......................... 127
    C.2.6.1 Partition One .................................................. 127
    C.2.6.2 Partition Two .................................................. 128
C.2.6.3 Partition Three .............................................. 129
C.2.6.4 Partition Four ............................................... 130
C.2.7 Manually Generated VHDL File For Simulation Of Partitioned Design 131
C.2.8 Schematic Diagram for Manually Generated VHDL File For Simulation 135

Bibliography 136
# List of Figures

1.1 VHDL-code partitioning process ............................................ 4

2.1 Simple Crossbar Topology .................................................. 11

2.2 Simple Mesh Topology ....................................................... 11

2.3 TM-3 Block Diagram ......................................................... 12

2.4 Virtex’s VersaRing ........................................................... 13

2.5 A Possible Implementation Of An OR Gate’s Entity Construct .......... 14

2.6 Entity and Port-Signal Declarations ....................................... 16

2.7 Vector Signals Representing Two Buses Of 16 Wires Each ............... 17

2.8 Component Declaration ...................................................... 19

2.9 Component Instantiation Declaration ..................................... 19

2.10 Hierarchical Design ........................................................ 21

2.11 VHDL code in Structural Style for top-level hierarchy of Fig. 2.10 .... 21

2.12 VHDL code in Structural Style for “Component_with_Subcomponents” of Fig. 2.10 .............................................................. 22

2.13 VHDL code in Behavioural Style for “Component_with_Subcomponents” of Fig. 2.10 .............................................................. 22
2.14 VHDL code in Behavioural Style for MUX of Fig. 2.10 .......................... 23
2.15 Difficult-to-synthesise VHDL code ....................................................... 23
3.1 Non-supported VHDL instantiated in an upper-level VHDL file ............... 26
3.2 Non-supported VHDL Code BEFORE It Is Converted Into Supported VHDL . 27
3.3 VHDL Code AFTER It Is Converted Into Supported VHDL ....................... 28
3.4 VHDL Code for component "registercomponent" so it can be instantiated in
   Figure 3.3 and supported by the partitioning tool ................................. 29
3.5 Entity and Architecture Signals Represented as Edges of a Circuit's Graph . 32
3.6 Entity and Architecture Signals Represented as Nodes of a Circuit's Graph . 33
3.7 Node Sets' Graphical Representation .................................................. 33
3.8 Instantiated Components Interconnection ............................................. 36
3.9 No Fan-in Allowed Between Instantiated Components ........................... 36
3.10 Fan-out Between Instantiated Components .......................................... 37
4.1 Graph Partitioning ................................................................................ 41
4.2 Graph $G$ Before Partitioning With Entity And Architecture Signals As Nodes . 42
4.3 Partitioned Graph With Entity Signals ($ES_{cmmn}$) Demarcating Partitions $GP_1$
   and $GP_2$ ......................................................................................... 42
4.4 Two Instantiated-component Signals Dependent Upon Two Other Instantiated-
   Component Signals ............................................................................. 44
4.5 Instantiated-component signal dependents ............................................. 44
4.6 Output Signal's Inter-partition Dependence Example ............................. 49
4.7 Input Signal's Inter-partition Dependence Example Before Moving $ic_{2bm}$ ...... 50
4.8 Input Signal's Inter-partition Dependence Example After Moving $i_{c2bm}$  

5.1 A Hierarchical Design And Its Representation With A Tree Data Structure  

5.2 Instantiated Components Replaced By Their Children  

5.3 Graph being moved one level up inherits all the connections (ancestors and dependents) of the instantiated component and loses its interconnection signals (entity signals)  

5.4 Vector Signal with dependents of different widths  

6.1 Fan-out in inter-partition dependents  

6.2 Inter-partition dependents using different FPGA pins  

6.3 Inter-partition dependents using different FPGA pins and an Architecture Signal  

7.1 First VHDL Design Schematic Diagram  

7.2 Partition One of First VHDL Design  

7.3 Partition Two of First VHDL Design  

7.4 Alarm Clock Design Hierarchy  

7.5 Alarm Clock Block Diagram  

7.6 Top-Level File Selection  

7.7 Instantiated Components In Flattened Design  

7.8 Selected Instantiated Components Being Moved From Partition 1 to Partition 2  

7.9 Files Generation
Chapter 1

Introduction

1.1 Motivation

Over the past years, research in the area of logic circuit emulation has been increasing. The need for extensive verification of larger and more complex designs, and the need for reduction of time-to-market and costs in fabrication of VLSI chips has made emulation more popular.

The development of the Transmogrifier 3 at the University of Toronto has resulted in a multi-FPGA system that could be used for emulation. To make it useful requires the development of a partitioning tool.

Partitioning of electrical circuits is a topic widely and thoroughly researched in the VLSI field. However, most of this research is focused on the partitioning of a design after it has been synthesised and represented by the smallest elements of the device to which it is being targeted. The approach taken here is the partitioning of a design at its high-level language representation (i.e. VHDL-language representation) before it has been synthesised. Since the software developed in this thesis is a first-time implementation, an easy way of implementing
and testing it had to be chosen. This first implementation partitions a design manually i.e. the intervention of the designer is required to select the location of a design’s components.

Partitioning a design at this level allows the designer to know more easily which components reside in which partition. Consequently, in case the simulation or emulation of a design does not behave/perform as expected, the identification of the proper file(s) representing the component(s) to modify becomes easier. Moreover, since the partitioning approach taken here is manual, the designer has the option of placing the design’s components onto any desired partition (FPGA).

Nevertheless, there are some trade-offs in this approach. First, the granularity of the graph representing the circuit is affected. In a synthesised circuit’s graph every node usually represents primitive cells [1], whereas in a VHDL-design’s graph, its nodes represent instantiated components and entity and architecture signals. And second: since the partitioning is manual, the number of inter-partition crossing wires will not be optimal, although it is not necessary since this number need only be just good enough to achieve good partitions.

1.2 Thesis Overview

The purpose of this work is to develop a VHDL-code partitioning tool for the Transmogifier 3 (TM-3). The tool is dedicated to the TM-3 because it can divide the code into the same number of partitions as FPGAs on the TM-3: furthermore, it follows the considerations given by the TM-3’s topology. Since this is a first implementation of the tool, it gives support to only a subset of the VHDL language, though, a simple algorithm for supporting a broader range of VHDL constructs is proposed.
The partitioning tool reads the VHDL code given by the original design, partitions it, and translates it into one VHDL file per partition. During partitioning the original VHDL file passes through a series of steps depicted in Figure 1.1.

The first phase in the partitioning process is parsing the VHDL files that describe the original design. Once these files have been parsed, an intermediate representation is created. This intermediate representation is an internal format that allows the tool to be VHDL-parser independent. Thus, any VHDL parser can be modified to write out intermediate representation files that could be used by the partitioning tool.

The next step consists of forming a graph data structure from the intermediate representation files. If the design has been written hierarchically, all files in the design and their corresponding graphs are brought to their upper levels until the design contains only one level of hierarchy. Since all graphs are now merged into one single graph and form only one hierarchy, the user can move all components to any partition. In this phase the instantiated-component signal nodes representing vector signals (buses) are modified to represent single signals. This occurs only if those vector signal nodes are partially shared by different instantiated components.

After the entire design is represented by one single graph, the designer provides the tool with the set of components to be moved to each partition. This is done by means of a graphical user interface (GUI) that is fast and easy to use. The GUI shows the current location of the design's components, and allows the user to move them around partitions with a simple click of the mouse. The GUI also stores previous partitioning schemes selected by the user. This feature makes it easier for the user to try different combinations of components in different partitions, or to continue the partitioning process at a later time without having to move all
Figure 1.1: VHDL-code partitioning process
components to their previous position from the beginning.

Once the components have been placed in the given partitions, the VHDL files for each partition can be generated. In this phase, the subgraphs for each partition are translated into compilable VHDL code. When the VHDL code is being generated, it is also modified to consider the characteristics of the TM3's topology. Once generated, all the VHDL files can now be synthesised using any commercial tool, targeted and downloaded onto the corresponding FPGA. Since the partitioning tool does not tell the user whether the partitions fit into the corresponding FPGAs, the user needs to compile the generated VHDL files and get this information from the compiler. If the partitions fit, the partitioning process is finished. But if they don't fit, or if the user wants to try different partitioning approaches, VHDL files with new partitioning information have to be created. Each of the steps for partitioning a design will be described in the thesis.

1.3 Thesis Organisation

Chapter 2 provides a quick reference of the VHDL language related to this thesis. Also, some of the previous work in logic emulators and circuit partitioning approaches is given. Along with this, a brief description of the TM-3 is included. Chapter 3 focuses on the parsing of source files containing supported VHDL, their intermediate representation, and their translation into a graph data structure. Properties of the graph and its dependency definitions are discussed as well. Chapter 4 describes the actual partitioning of the VHDL code. Also discussed are signal dependencies, and why they must be considered for the partitioning of a design. Special cases such as hierarchical designs and vector signals, and how they are handled are presented in
chapter 5. In chapter 6, VHDL-code generation and how it is affected by the TM-3's topology is discussed. Chapter 7 shows the results of this thesis with a complete partitioning flow of a real design. The partitioning flow includes the use of the GUI and some snapshots of it. This chapter also presents the partitioning of a design containing a component with all dependency cases. Chapter 8 provides conclusions, along with a description of possible future work.
Chapter 2

Background Information

This chapter presents some background information about multi-FPGA systems and VHDL. In particular some details about the TM-3 are given. It is by no means intended to present an in-depth discussion of all existing systems and their software. The VHDL part assumes the reader has some knowledge of this language. It only presents an overview that serves as a quick reference for the subsequent chapters. A good starters’ reference is given by Brown [40].

2.1 Multi-FPGA Systems

Multi-FPGA Systems have a wide range of applications. One of the most important ones is logic emulation. The need for design verification has made FPGA-based logic emulators a growing area of research.

Emulation of logic circuits in these specialised hardware systems offer many advantages over software simulation and prototype fabrications. Very complex designs can be mapped onto these logic emulators with certain ease whereas in software simulation, as designs become
more complex, they also become more complicated to simulate. The full functionality of the ASIC being emulated is tested with real components on a prototype board, thus eliminating the inaccuracy of software component modelling. Since designs are implemented in hardware, clock speeds are several orders of magnitude faster than software simulators[2]. Furthermore, there are many cases in which software drivers are needed for the hardware being implemented but they cannot be written until a working first prototype has been implemented[3].

Another important advantage is money and time savings since prototype fabrications are quite expensive and slow to manufacture. With logic emulators, non-recurring engineering (NRE) costs are greatly reduced or eliminated when repeated prototype fabrications is required. Minor modifications to a design usually implies re-fabrication of the prototype. The major disadvantage of logic emulators against hardware prototypes is the lower frequency at which they can run.

As an array of FPGAs connected with a certain interconnection topology, FPGA-based logic emulators present many challenges and opportunities for research. The areas of interconnection topologies and circuit partitioning are the most explored since they are the major steps for building and using a logic emulator, respectively. Even FPGA architectures specialising in logic emulation have been proposed [4].

ASIC verification is commercially available in several models of multi-FPGA emulation systems from various companies [5, 6, 7, 8, 9]. These systems provide circuit partitioning at a very fine-grain level.
2.1.1 Circuit Partitioning Techniques

Current FPGA technologies can handle designs of several hundreds of thousands of equivalent gates [10], but due to the increasing size of designs, a single FPGA is still not sufficient for implementing many designs. The number of gates in typical ASIC designs is larger than the number of equivalent gates a single FPGA can hold. Because of this, designs need to be partitioned into multiple subpartitions, each fitting into a single FPGA. This is the first major step in doing logic emulation [11].

Logic partitioning is one of the critical issues in CAD for digital logic. Because of the increasing use of FPGA-based emulators and prototyping systems, partitioning is becoming even more critical. This has lead to a number of heuristic approaches such as Kernighan-Lin [12], Fiduccia-Mattheyses [13] and Simulated Annealing [14] among others. These approaches try to obtain optimal solutions for the amount of crossing wires between partitions as well as balancing the partitions. In fact, there have been so many approaches that it has been difficult to compare and contrast all of them. Nevertheless, evaluations of some of these approaches have been made [15, 16]. Some work has been done for partitioning a design at a high-level language level. Some approaches for it are presented in [17] and [18].

FPGA-based logic emulation systems are designed to hold millions of gates. These emulators used to be large structures so they could hold all the logic contained in an ASIC design [19]. Due to higher density integration, FPGAs have become faster and can hold larger designs. New algorithms have been developed to use as much as possible all of the logic an FPGA is capable of holding. These algorithms have tried to overcome some of the main problems in logic partitioning by increasing the pin intercommunication bandwidths [20], and by reducing
critical paths crossing several FPGAs with component replication [21].

2.1.2 Network Topologies

Logic emulators can be characterised by their interconnection topology. The way their FPGAs are arranged and how they share resources is the most important difference between existing logic emulators. Logic emulator topology impacts greatly the system speed, capacity and scalability.

There has been a lot of research in this area. Several chip-arrangements and sharing of resources have been proposed [22, 23]. Crossbar and mesh topologies are the most common topologies. Crossbar topologies use field programmable interconnect chips (FPIC) [24, 25] (cheap and small FPGAs can also be used) for board-level routing purposes. These FPICs are programmable switches that interconnect the main FPGAs in to which the designs are mapped. Figure 2.1 shows an example of this type of topology. Topologies of this type provide a predictable routing delay; the trade-off is chip utilisation and scalability. Nonetheless, several other attempts such as the TM-2’s topology[26] have been made to overcome these limitations.

Mesh routing topologies interconnect the emulator’s FPGAs in a nearest-neighbour fashion. The I/O pins of each FPGA are divided into proper subsets of equal size and are assigned to the nearest neighbour. Their main disadvantage versus crossbar topologies is the fixed amount of pins assigned to a neighbour, and their less predictable delay in systems with many chips. Figure 2.2 presents a simple example of a mesh topology in a system with many FPGAs.
Figure 2.1: Simple Crossbar Topology

Figure 2.2: Simple Mesh Topology
2.1.3 Transmogrifier 3

The Transmogrifier 3 is the successor of the TM-2 multi-FPGA rapid prototyping system built at the University of Toronto. The TM-3 takes advantage of state-of-the-art FPGAs from the Xilinx Virtex family to emulate designs of several hundred thousands of gates (depending on routing resources and pin utilisation). The TM-3 is comprised of four Virtex 1000 FPGAs[10], which can be easily replaced by Virtex 2000E. It contains external RAM, so designs using large amounts of RAM can be emulated. Figure 2.3 shows the TM-3 topology.

2.1.3.1 Topology

The TM-3 uses a mesh topology to interconnect the four main FPGAs. The I/O pins of each FPGA are divided into proper subsets of equal size. They interconnect to all neighbours. In

Figure 2.3: TM-3 Block Diagram
large FPGA-based logic emulators (large in number of FPGAs, not necessarily capacity) it may be better to interconnect FPGAs in a way such as proposed in [22]. In that approach pins are permuted in a north-south-west-east manner to improve routing resources in each FPGA and to reduce delay among paths crossing various chips. However, this approach leads to a more difficult and expensive board design since more layers are likely required. Moreover, the TM-3 uses only four FPGAs and all of them are interconnected; paths would cross only a few chips and its delay would not be greatly improved. Furthermore, the TM-3 uses Xilinx Virtex FPGAs with the VersaRing© architecture which is a routing ring with resources specific to the assignment of I/O pads. This architecture is supposed to reduce the effects of pre-assigning signals to I/O pins at the board-level (i.e. pin locking)[30].
2.2 VHDL Quick Reference

VHDL is a very complex high-level hardware description language. The initials stand for VHSIC (Very-High-Speed Integrated Circuit) Hardware Description Language. It was initially used only for documenting and simulating designs. Now it is recognised as a standard for describing digital circuits for synthesis by the IEEE (IEEE Standards 1076 and 1164[27]).

The VHDL language contains several constructs for describing a circuit. The construct for describing the external interfaces of a circuit is called an entity. An entity represents one level of the design’s hierarchy (the design may have only one hierarchy level). This construct defines the I/O signals of the circuit being implemented. Figure 2.5 illustrates the entity construct for a simple two-input OR gate.

```
ENTITY orgate IS
PORT (  
      a : IN bit;
      b : IN bit;
      c : OUT bit;
    );
END orgate;
```

Figure 2.5: A Possible Implementation Of An OR Gate’s Entity Construct

The actual implementation of the circuit is defined inside a construct called architecture. Entities can have different architectures (implementations) describing their behaviour or the subcomponents the entity uses. Subcomponents are placed inside an architecture by coding a component declaration construct and by calling it with a component instantiation construct. As many as desired instantiated components can use this declaration. Component declarations
can be present in the same file where they are being used, or in a package file being called by the entity using them.

Depending on their level of abstraction, modelling VHDL architectures can be categorised into three design styles[28, 29]: structural, behavioural and dataflow. Structural VHDL is the view closest to hardware since it describes it as a set of components interconnected by wires (i.e. a netlist). The components are represented by instantiated component constructs, and the wires by entity and architecture signals. The second group: Behavioural VHDL, as its name indicates, describes the behaviour or functionality of a circuit with program-like constructs. This style is comprised by much more high-level program-like constructs. These constructs have to be translated before they can represent a netlist of hardware components. These constructs need to be meaningful in hardware so they can be synthesised into it. Thus, their synthesis is not always possible; especially when designers forget they are not writing a program but describing hardware behaviour. Some behavioural constructs can infer hardware-like registers, multiplexers, etc, but others cannot be translated to hardware without more sophisticated techniques like behavioural synthesis. The third group, dataflow, is comprised by a combination of both previous styles in the same file.

2.2.1 Port Signals

Input and output signals of a VHDL design are defined in the entity declaration. These signals are described as port signals because they are the circuit interface to other designs.

Port signals must have a name, a mode and a type. The name gives them their identity. The mode specifies whether the signal is an input (IN), an output (OUT), both (INOUT), or a BUFFER. If a signal’s mode is omitted, by default, the signal is considered an input signal.
Figure 2.6 shows an example of port signal declarations in an entity declaration.

```
ENTITY component_name IS PORT (  
    formal_name_of_signal : IN anytypeofsignal;  
    formal_name_of_signal.2 : OUT anytypeofsignal  
);  
END component_name;
```

Figure 2.6: Entity and Port-Signal Declarations

### 2.2.1.1 Signal Mode

The mode of the port signals gives them certain properties and restrictions:

- **OUT** Output signals can only be assigned a value. They cannot be used inside the entity’s architecture i.e. they cannot have dependents since they cannot be read. They can have only one driver unless a *resolution function* [27] is written.

- **BUFFER** This is a special mode that allows the signal to be used as an output signal and as an internal signal driving other components (dependents). It can only be assigned a value, so this value can be used inside the architecture and also as an output. It can also have only one driver unless a *resolution function* is specified.

- **IN** Input signals can only be read. They can have multiple dependents, but they do not have drivers i.e. they cannot be assigned a value.
- **INOUT** A signal with this mode has all the characteristics and limitations of an input or an output signal when used as an input or as an output signal, respectively. To assign a value to these signals, it is necessary to use certain VHDL constructs (behavioural-VHDL constructs) stating the conditions in which the signal is going to be treated as an output, otherwise the signal is considered an input. Signals with **INOUT** mode require a high-impedance state, and its corresponding representation ('Z') must be assigned when the signal is being used as an input.

### 2.2.1.2 Signal Type

The type of a signal is a previously defined data type. It usually is defined in the *package(s)* or libraries being used. User-defined types can be created using the proper VHDL syntax. Very often they are subtypes of previously defined types.

The IEEE VHDL 1164 library provides some common data types such as *bit, std_logic, std_ulogic*, etc. that represent single signals. Each of these types have an associated type used to represent bus signals. The associated types are called vectors, and they represent a collection of signals of the original type. To represent this type of signal, the original type is appended the "_vector" string, and a width (number of signals in the group) must be specified (see Fig. 2.7).

```vhdl
formal_name_of_signal1 : IN std_logic_vector (15 downto 0);
formal_name_of_signal2 : IN bit_vector (0 to 15);
```

*Figure 2.7: Vector Signals Representing Two Buses Of 16 Wires Each*
2.2.2 Internal Signals

Signals interconnecting subcomponents (wires) are called *architecture signals*. Like port signals, their declaration contains a name and a type for the signal, but not a mode. Their declaration must be preceded by the `SIGNAL` keyword. They can have multiple dependents. They can also have multiple ancestors if they have an associated *resolution function*. However, for simplicity in this work, it is assumed that they can have only one driver (ancestor). These signals not only can be used to connect instantiated components but also can be used by VHDL program-like constructs to assign or read values.

2.2.3 VHDL Subcircuits

Subcircuits can be used in VHDL by instantiating components into the VHDL design file. These instantiated components can have other instantiated components in the VHDL design file describing them, thus forming a hierarchical VHDL design. To form a hierarchical design the “component declaration” and “component instantiation” constructs are necessary.

2.2.3.1 Component Declarations

Component declarations allow designs to use existing subcircuits. They define the subcircuit’s interface. They must be declared in the architecture part or in a package. This interface is almost identical to its entity declaration. An example of a component declaration is given in Fig. 2.8. This interface can be used by as many instantiated components as desired.

\(^1\)Another reason is that buses would not be possible to implement
COMPONENT component_name
PORT(
    formal_name_of_signal  : IN anytypeofsignal;
    formal_name_of_signal_2 : OUT anytypeofsignal
);
END COMPONENT;

Figure 2.8: Component Declaration

2.2.3.2 Instantiated Components

Once the component declaration (socket) is given, the component can be instantiated as a subcircuit using the proper constructs. A component instantiation syntax example is given in Fig. 2.9.

instance_name : component_name PORT MAP(
    formal_name_of_signal  => actual_name,
    formal_name_of_signal_2 => actual_name2
);

Figure 2.9: Component Instantiation Declaration

The name "formal_name_of_signal" represents the name of the port signal in the subcircuit. The name "actual_name" is the name of an entity or architecture signal in the code calling the component. This type of association is called named association because the formal part of the signal is present in the component instantiation. There is another type of association called positional association where the formal part is omitted. The actual parts are assigned the signal in the same order in which they were declared in the component declaration statement. This type of association is not supported by the partitioning tool since it is considered a poor design style. Besides the graph generation becomes simpler.
The mode of the signals is not provided in component instantiation constructs, therefore the signal must be associated to its corresponding signal in the component declaration part in order to know its mode. Knowing the mode of the signal helps to easily determine whether the signal can have dependents or ancestors.

2.2.3.3 Hierarchical Designs and Levels of Abstraction

The architecture of a VHDL subcircuit can be described using any of the various levels of abstraction. However, to make a hierarchical design, structural VHDL must be used. Hierarchies are built by instantiating components inside the architecture describing a component. Figure 2.10 shows a schematic diagram of a hierarchical design, and Figs. 2.11, 2.12 and 2.14 show the VHDL architectures for its components. Figure 2.13 shows another possible architecture for "component_with_subcomponents". Note that the "MUX" component architecture could also have been described by instantiating the necessary components to build a multiplexor. These components can be part of a library which is called from the VHDL design file that using the instantiated component.
ARCHITECTURE OF toplevel_in_STRUCTURAL_STYLE IS

COMPONENT andgate PORT(
  Sin1 : in bit; Sin2 : in bit; Sout : out bit)
END COMPONENT;

COMPONENT component_with_subcomponents PORT(
  A,B,C : in bit; E : out bit)
END COMPONENT;

SIGNAL C : BIT;
BEGIN
  andgatel : andgate PORT MAP (Sin2 => a, Sin1 => b, Sout => c);
  componentX : component_with_subcomponents PORT MAP (A => C, B => D, C => E, E => F);
END toplevel_in_STRUCTURAL_STYLE;

Figure 2.10: Hierarchical Design.

Figure 2.11: VHDL code in Structural Style for top-level hierarchy of Fig. 2.10
ARCHITECTURE OF component_with_subcomponents IS

COMPONENT MUX PORT(
    A,B,Sel :in bit; O :out bit
);
END COMPONENT;

COMPONENT NOT PORT(
    A :in bit; O :out bit
);
END COMPONENT;

SIGNAL D : BIT;
BEGIN
MUX1 : MUX PORT MAP (    
    A => A, B => B, Sel => C, O => D);
NOT1 : component_with_subcomponents PORT MAP (    
    A => D, O=>F);
END component_with_subcomponents;

Figure 2.12: VHDL code in Structural Style for “Component_with_Subcomponents” of Fig. 2.10

ARCHITECTURE OF BEHAVIOURAL component_with_subcomponents IS
SIGNAL D : BIT;
BEGIN
O <= B when Sel='l' else A;
-- If the mux’s output were not negated
-- this statement would look like this
-- O <= A when Sel='l' else B;
END BEHAVIOURAL component_with_subcomponents;

Figure 2.13: VHDL code in Behavioural Style for “Component_with_Subcomponents” of Fig. 2.10
2.2.4 Other VHDL Constructs

There are many more VHDL constructs. Most of these complex VHDL constructs usually describe the behaviour of the circuit rather than the instantiated components the circuit must use to perform the required task. These program-like constructs may be so complex (often in bad or very complex designs) that even some commercial tools will not be able to synthesise them. An example of difficult-to-synthesise code is presented in Fig. 2.15.

```vhdl
process(atmp)
    variable i,j : integer;
    variable btmp : std_logic_vector(WORD_WIDTH-1 downto 0);
begin
    for i in 0 to WORD_WIDTH-1 loop
        btmp(WORD_WIDTH-1-i) := '0';
        for j in 0 to i loop
            btmp(WORD_WIDTH-1-i) := btmp(WORD_WIDTH-1-i) or atmp(WORD_WIDTH-1-j);
            end loop;--j
        end loop;--i
    ctmp <= btmp;
end process;
```

Figure 2.15: Difficult-to-synthesise VHDL code
Chapter 3

VHDL-Design Analysis

The first step to partitioning a design is to analyse it and represent it in such a way that can be easily manipulated. The input to the partitioning software tool consists of a series of VHDL files describing a design. These files are read, analysed and then translated into an intermediate representation. From this intermediate representation a graph data structure is created in such a manner that facilitates partitioning and VHDL code generation. The type of analysis made on the source files is lexical and syntactical.

3.1 Lexical and Syntactical Analysis

The lexical and syntactical analysis of the VHDL source code is performed by means of a lexer and a parser taken from the base line of VAUL [31, 32], which is the front-end for the Free VHDL project [33]. The lexer and parser can analyse input files written in either VHDL’87 or VHDL’93 standards.

The parser uses BNF-like [34] syntax to describe VHDL syntax rules (also called constructs). Specifically, these rules are written for YACC [35] or Bison [36] (with minor modifications). After the definition of each rule, the programmer can append C code to the rule. This C code will tell the parser what operations to perform with the rule, so it can be useful. In this
case the rules are added code so an intermediate representation file can be created out of the VHDL source file.

### 3.1.1 Supported VHDL

The scope of this first implementation of the partitioning tool is to work with a subset of the VHDL language only. This subset is comprised by a subset of what some literature[28, 29] calls a *structural level of abstraction* for modelling VHDL design’s architectures. Therefore, only the rules related to that level of abstraction are useful, and no coding had to be added for the rest of the VHDL rules. As explained in chapter 2, a structural level of abstraction can be seen as the view closest to hardware. This view is presented as a netlist of *instantiated components* interconnected by *architecture* and *entity signals* (wires). The supported VHDL subset does not include inout signals, generics, generates, and high-level program-like constructs. Figures 2.10 and 2.11 depict an example of the structural level of abstraction.

The reason for at first not supporting all of the VHDL constructs is simple: the VHDL language is a rather complex language. It would take too long to give support to all of those constructs. But more importantly, some of the non-supported VHDL constructs need to be synthesised before they describe actual hardware that could be partitioned. It is not an objective of this work to build a VHDL compiler.

Nevertheless, some non-supported VHDL\(^1\) constructs can still be part of the VHDL design to be partitioned since supporting VHDL at the structural level of abstraction allows the designer to instantiate components previously compiled (or even without architectures yet). Thus, a VHDL design file containing non-supported VHDL constructs can form a component

---

\(^1\)In this section “non-supported VHDL” refers to high-level program-like VHDL constructs such as *process, if-then-else, case, etc.* statements but *not* to the generic, generate and inout signal constructs.
and then be instantiated in an upper-level hierarchy. Figure 3.1 shows a register written in non-supported VHDL being instantiated in a higher-level hierarchy.

![Diagram of a register with a non-supported VHDL example]

Figure 3.1: Non-supported VHDL instantiated in an upper-level VHDL file

It is important to note that the input VHDL design files must not contain a combination of both supported and non-supported VHDL. A combination of them would cause, either the tool to stop its parsing process, or its malfunctioning with some warning messages. If the original VHDL design to be partitioned has been written using both types of VHDL constructs (supported and non-supported), the non-supported VHDL constructs can be manually separated by placing them in a new design file, thus, forming a component. Then this new component can be
instantiated back into the original file where the rest of the supported VHDL constructs reside. Figure 3.2 shows the VHDL code of the circuit in Fig. 3.1 before the “Register” (non-supported VHDL) is separated from the supported VHDL in the original file to form an instantiated component. Figure 3.3 presents the non-supported VHDL code after it has been separated to create a new component so it can be supported. Figure 3.4 shows the code for the new component.

Note that VHDL files containing non-supported VHDL are only in the lowest levels of the design’s hierarchy. The reason for this is that component instantiation is the only way to allow a VHDL file to have a lower-level hierarchy, and component instantiation is part of the supported VHDL. An algorithm for automatically separating non-supported VHDL from supported VHDL and vice-versa is seen as future work.

... 
architecture non_supported_VHDL of TOP-LEVEL is
SIGNAL LD,XOR_TO_NOT : bit;
Begin
Process(clk,ld)
  clk’event and clk=’1’ then
    If ld=’1’ then
      Dataout<=Datain;
    Else
      DataIN<=DataIN;
    End if;
  End if;
End process;
xor1 : xorgate port map (
  IN1=>A, IN2=>B,
  OUT1=>XOR_TO_NOT);
not1 : notgate port map (
  IN1=>XOR_TO_NOT,OUT1=>LD);
end non_supported_VHDL;

Figure 3.2: Non-supported VHDL Code BEFORE It Is Converted Into Supported VHDL
... 

architecture supported_VHDL of TOP-LEVEL is
-- this component declaration is added so the
-- unsupported VHDL can be instantiated as a component
component registercomponent
port(    clk : in std_logic;
       ld : in std_logic;
       datain : in std_logic_vector(15 downto 0);
       dataout: out std_logic_vector(15 downto 0);
    );
end component;
SIGNAL LD,XOR_TO_NOT : bit;
Begin
register: registercomponent port map(
    clk=>clk,
    ld=>ld,
    datain=>datain,
    dataout=>dataout);

xor1 : xorgate port map (    IN1=>A,
                        IN2=>B,
                        OUT1=>XOR_TO_NOT);

notl : notgate port map (    IN1=>XOR_TO_NOT,
                        OUT1=>LD);
end supported_VHDL;

Figure 3.3: VHDL Code AFTER It Is Converted Into Supported VHDL
entity registercomponent is
port(
    clk : in std_logic;
    ld : in std_logic;
    datain : in std_logic_vector(15 downto 0);
    dataout: out std_logic_vector(15 downto 0);
);
end registercomponent;
architecture non_supported_VHDL of registercomponent is
Begin
    Process(clk,ld)
        clk'event and clk='l' then
            If ld='l' then
                Dataout<=Datain;
            Else
                DataIN<=DataIN;
            End if;
        End if;
End process;
end non_supported_VHDL;

Figure 3.4: VHDL Code for component "registercomponent" so it can be instantiated in Figure 3.3 and supported by the partitioning tool
3.2 Intermediate Representation

After a source file is parsed, it is translated and saved as an intermediate representation file. This occurs only if there does not already exist a file containing its intermediate representation and if the source file has not been modified since the last time the intermediate file was generated. It is faster to read an intermediate representation file than it is to parse the complete VHDL file since in the intermediate representation file there are simpler rules to recognise and they are all related only to the supported VHDL.

Intermediate representation of a VHDL source file also makes it faster for the partitioning tool to re-read in a design when only some of the VHDL source files have been modified. For simplicity in all the algorithms of the partitioning tool, it is assumed that one VHDL source file can describe the entity and architecture of only one component. Because of this, in a hierarchical design, where components can have subcomponents and subcomponents can have more subcomponents, and so on, there are as many files as components and subcomponents in the design's hierarchy tree. Consequently, should it be necessary to modify one source file, the other intermediate representation files would remain unchanged and their re-generation would not be required. Furthermore, the partitioning of a design is usually done when there is a working version of it, if not totally working, at least a testable version of it. In this phase VHDL source files (and consequently intermediate representation files) are not modified frequently.

Another important advantage of creating an intermediate representation of the source VHDL files is that it makes the partitioning tool independent of the VHDL parser. Therefore, any VHDL parser written in any language can be modified to produce an intermediate representation that follows the same format used by the partitioning tool.
3.3 Graph Implementation

The partitioning tool reads the intermediate representation file associated with the top-level of the hierarchy of the VHDL design to create a graph from it. This graph has all the information of the top-level VHDL file (components, signals, etc). If the design's instantiated components do not have subcomponents (i.e. the design has not been written hierarchically or is the lowest level in the hierarchy), or if the designer only wants to partition the design at its top-level, this is the end of the graph implementation. The graph obtained from this top-level intermediate representation file is the graph to be partitioned. However, if the design has been written hierarchically and the designer wants to have access to all subcomponents in the design and consequently a finer granularity in the partitioning process, a depth-first search algorithm is applied to the hierarchy tree to look for all the remaining intermediate representation files. A graph for each intermediate representation file containing subcomponents is then created. All of these files are merged with the top-level graph to form a single graph that represents the entire design in one level of hierarchy i.e. the design is flattened. All of these graphs are implemented in the same manner, and all of them have the same characteristics and properties described in this section.

3.3.1 Architecture and Entity Signals as Graph Nodes

A circuit $C$ described in VHDL could be represented by means of a directed acyclic\textsuperscript{2} hypergraph\textsuperscript{3} (called just a graph throughout this work) $G(V, E)$ where the vertices ($V$) represent the instantiated components ($IC$), and the edges ($E$), its interconnecting wires (entity and ar-

\textsuperscript{2}No signal starts and ends in the same component, and no fan-in is allowed

\textsuperscript{3}A signal can drive one or more components i.e. fan-out is allowed
architecture signals). Figure 3.5 depicts this approach.

![Figure 3.5: Entity and Architecture Signals Represented as Edges of a Circuit’s Graph](image)

However, in graphs built by the partitioning tool, not only the instantiated components (IC), but also the architecture signals (AS) and the entity signals (ES) are represented by vertices of the graph $G$, such that $V = ES \cup AS \cup IC$. This approach allows a faster VHDL code generation when the partitioning process is complete. The reason for this is that when generating the VHDL code for each partition, information regarding architecture signals (name, type, etc.) does not have to be deduced from the graph’s edges and component nodes because it is already present in a node. Furthermore, architecture signals may have to be converted to entity signals during the partitioning process and the required information for them will not have to be deduced from the nets of the graph because it already exists as node. As for the edges of the graph, they are only a directed connection between two vertices. They represent the connection between entity or architecture signals and instantiated-component signals. Figure 3.6 depicts a very simple graph implemented by the software tool for the circuit presented in Figure 3.5. Fig-
Figure 3.7 shows the meaning of the symbols in Figure 3.6.

Figure 3.6: Entity and Architecture Signals Represented as Nodes of a Circuit’s Graph

- **Entity Signals (ES_{in})** with IN mode (left of the circuit), or **Entity Signals (ES_{out})** with OUT mode (right of the circuit).
- **Entity Signals** with BUFFER mode.
- **Architecture Signals (AS).**
- **Instantiated Components (IC).**
- **Instantiated-Component Signals with mode** OUT or BUFFER ($S_{ob}$) in their component declaration.
- **Instantiated-Component Signals with mode** IN ($S_{in}$) in their component declaration.

Figure 3.7: Node Sets’ Graphical Representation
3.3.2 Node Set Definitions

The graph has other basic properties and assumptions. However, before describing them it is necessary to define the following node sets. The graphical representation of these nodes is provided in figure 3.7. Appendix A.2 gives a list of set notation and definitions.

1. \( ES_{in} = \{ es_1 \cdots es_{N_{ES_{in}}} \} \) is the set of port signals with mode \textit{IN}; where \( N_{ES_{in}} \) is the total number of port signals with this mode.

2. \( ES_{out} = \{ es_1 \cdots es_{N_{ES_{out}}} \} \) is the set of port signals with mode \textit{OUT}; where \( N_{ES_{out}} \) is the total number of port signals with this mode.

3. \( ES_{buffer} = \{ es_1 \cdots es_{N_{ES_{buffer}}} \} \) is the set of port signals with mode \textit{BUFFER}; where \( N_{ES_{buffer}} \) is the total number of port signals with this mode. Note that this set may be empty.

4. \( ES = ES_{in} \cup ES_{out} \cup ES_{buffer} = \{ es_1, \cdots, es_{N_{ES}} \} \) is the set of port signals with any mode; where \( N_{ES} \) is the total number of port signals in the given design, and

\[
N_{ES} = N_{ES_{in}} + N_{ES_{out}} + N_{ES_{buffer}}. 
\]

5. \( AS = \{ as_1, \cdots, as_{N_{AS}} \} \) is the set of architecture signals in the design, where \( N_{AS} \) is the total number of them.

6. \( IC = \{ ic_1, \cdots, ic_{N_{IC}} \} \) is the set of instantiated components in the design; where \( N_{IC} \) is the total number of them.

7. \( S_{in} = \{ s_{in_1}, \cdots \} \) is the set of input edges \( E(ic) \) at any given \( ic \in IC \). In other words, this represents the set of signals in an instantiated component whose mode in the component declaration is \textit{IN}.
8. $S_{ob} = \{s_{ob1}, \ldots \}$ is the set of output edges $E(ic)$ at any given $ic \in IC$. In other words, this represents the set of signals in an instantiated component whose mode in the component declaration is $OUT$ or $BUFFER$. Note that buffer signals are treated as output signals for assigning a direction to the edges at an instantiated component.

9. $S = S_{in} \cup S_{ob}$ is the set of edges $E(ic)$ at any given $ic \in IC$. In other words, this represents the set of signals in an instantiated component whose mode in the component declaration is $IN$, $OUT$ or $BUFFER$. Note that an instantiated component $ic$ could be seen as a set of instantiated-component signals.

### 3.3.3 Properties of the Graph

The following properties and considerations are mostly given by the syntax of the VHDL language. This is because it is desirable for the graph to have as much VHDL information as possible so the file generation process is easier and faster. Some other properties just make it easier for the partitioning and graph generation processes.

- $E(IC, IC) = \emptyset$ Direct edges to and from instantiated components are not allowed because the VHDL syntax requires the presence of an architecture or entity\(^4\) signal to interconnect instantiated components. Figure 3.8 provides a simple graphical explanation for this property, where components $l$ and 2 are being interconnected by an architecture signal ($x$) and a buffer signal ($y$). Note that edge $z$ is not allowed.

- $E(S_{ob}, as) \cap E(S_{ob}, as)$, and $E(S_{ob}, es_{out}) \cap E(S_{ob}, es_{out})$ are not defined because multiple instantiated components cannot drive one single signal i.e. no fan-in is allowed

\(^4\)Entity signal mode must be $buffer$
because there are no tri-state components inside an FPGA\textsuperscript{5} that could be synthesised.

Figure 3.9 depicts illegal circuits with fanin, where \( a \) and \( b \) show multiple drivers for their corresponding signals.

---

\[ E(es, S_{in}) \cup E(es, S_{in}) \text{ exists for all } as \in AS \text{ and for all } es \in ES_{in} \cup ES_{buff} \]

because fan-out is allowed (see Fig. 3.10).

\textsuperscript{5}Although Xilinx's Virtex FPGAs have some internal tri-state buffers, it is considered as a bad design methodology to use them, and therefore they are not supported.
3.3.4 Component Declarations and Direction of Edges

To assign a direction to the edges of the graph and build it, it is necessary to know the mode of all the signals in all given instantiated components. This mode is given in the component's declarative part of the design's VHDL source code. Therefore the component declaration must be linked with the instantiated component, adding more nodes to the graph. This can be seen as two graphs, the one just described, and another comprised by the instantiated components and their declarations. Their intersection is the set of instantiated components. However, for simplicity in the explanation of the algorithms, it is assumed that the instantiated component vertices already contain the mode of their signals.

So far, the partitioning tool does not support component declarations to be in a file other than the one being parsed. This implies that component declarations in packages is not supported (note that components in libraries are supported as long as their component declaration construct is present in the file being parsed). The main reason for this is simplicity in the algorithms. If an instantiated component in a VHDL file does not have its declaration in that file, all
packages specified in the VHDL file would have to be parsed until the component declaration is found. This would happen for all instantiated components. Thus, the process of assigning direction to the graph’s edges would become slower. Also package’s constructs would have to be supported in the parser. This process is seen as future work.
Chapter 4

VHDL-code Partitioning

4.1 Introduction

The partitioning of the VHDL code (being represented as a graph at this point) takes place after the user enters the partitioning information into the software tool. This tool expects the user to select a subset of the design’s instantiated components to be moved to any of the four FPGAs on the TM-3. All instantiated components can be chosen by writing a script file, or in the GUI by dragging and dropping the selected instantiated components onto the desired FPGA (partition). This set of components will form a new subgraph for the given partition.

The selection of instantiated components to be moved is based on a trial-and-error scheme. That is, if the entire design fits into one single FPGA, partitioning is not required. However, designs intended for ASICs generally contain more gates than the usable ones in a single FPGA, so partitioning is required. In this case, the user will try to split the design into the available resources by moving components around the different FPGAs until it fits\(^1\). Consequently, not only partitioning of the graph is required, but also its unpartitioning (i.e. re-union of components to an existing partition).

Once the user has defined the set of components to be moved, the graph representing the

\(^1\)The partitioning tool does not tell the user whether a design fits. To know whether it fits, the designer must compile and target the generated VHDL files to the type of FPGAs the logic emulator has.
design is ready to be partitioned. For both, partitioning and unpartitioning, it is necessary to perform a dependency analysis on the design's instantiated-component signals (only on the ones being moved and that have not been analysed previously in the current partitioning process). This analysis ensures that even though the components are in different partitions, their signal dependencies are respected so the partitioned design represents the same original design. If the design has not been previously partitioned, only signal dependencies within the partition where the components to be moved reside are analysed. But if the design has been previously partitioned and instantiated-component nodes in a partition are to be joined with other components in other partitions, dependency analysis of inter-partition signals is also required. This is explained in depth in Section 4.3.2. This chapter focuses in the generation of graph partitions, their signal dependence analysis, and the formal representation of both.

4.2 Graph Partitioning

In general, a partition can be represented as a subgraph \((GP_p)\) of the original graph \(G\) such that

\[
G = \bigcup_{i=1}^{p} GP_i \text{ and } GP_p \neq \emptyset \text{ and } GP_i \cap GP_j = \emptyset \text{ where } i \neq j \text{ and } GP_i, GP_j \in G [1].
\]

Figure 4.1 shows an example of a simple partitioning.

In the graph representation proposed in this work the intersection of any given partitions is the set of entity signals the partitions have in common (crossing signals). Recall, however, that entity and architecture signals are also represented as nodes as shown in Figure 4.2. Hence,

\[
GP_i \cap GP_j = ES_{cmmn}, \text{ where } ES_{cmmn} \text{ is the set of entity signals interconnecting partitions } i \text{ and } j.
\]

A similar approach is given in [37], but note that in the approach taken in this thesis a connecting signal \(es_{cmmn}\) is not considered one node, but two: the driving signal and the dependent signal. This representation allows a fast VHDL-code generation because both nodes
are present in both partitions at the time of generating the code. Having to deduce information from the connecting nodes to comply with the VHDL syntax is no longer necessary because all the required information is explicit in each node. Figure 4.3 shows an example of a graph after it has been partitioned with its interconnection signals represented as two interconnecting nodes. Figure 4.2 shows the same graph, but before it has been partitioned.

The new subgraph is comprised by the set of instantiated components and it’s corresponding entity and architecture signals so all subgraphs represent the original design. As can be seen in Fig. 4.3 new signals are created and some are just moved; this is defined by the dependencies of the instantiated components being moved to the new partition.

4.3 Signal Dependencies

When the graph is being partitioned (and components are being moved to and from partitions), the creation of new entity signals may be needed, and architecture signals may require trans-
Figure 4.2: Graph $G$ Before Partitioning With Entity And Architecture Signals As Nodes

Figure 4.3: Partitioned Graph With Entity Signals ($ES_{comm}$) Delimiting Partitions $GP_1$ and $GP_2$
formation into entity signals, or be moved to other partitions. To know what changes (if any) to perform to a given signal, or what attributes to give to a newly created one, it is necessary to know the dependencies of the components being moved.

When the component's dependencies are being analysed, it is actually the dependencies of each of the component's signals that is being analysed. The reason for this is that not only the instantiated-component node is being moved, but also the signals connecting it (architecture and/or entity signals). Furthermore, instantiated components do not have only one ancestor or dependent, but up to as many as signals the component has.

If the graph were to be partitioned only once, there would be only same-partition dependencies because there would not be dependencies to components in other partitions that require analysis. But since a subset of nodes of a partition or an entire partition can be rejoined with another partition (iteratively until the user finds a suitable -fittable- design partitioning), signal dependency analysis between partitions also needs to be considered. This is discussed in section 4.3.2.

4.3.1 Same-Partition Signal Dependencies

Let $IC_{2bm} \subseteq IC$ be the set of instantiated components selected by the user to be moved from partition $G_{P_k}$ to partition $G_{P_m}$, where $(k \neq m)$ and $G_{P_k}, G_{P_m} \in G$. Before moving an element $ic_{2bm} \subseteq IC_{2bm}$, the same-partition dependencies for each of its signals must be checked in order to know whether the signal in question has dependent or ancestor signals within the same partition. A formal definition for these dependencies is given in appendix B.2.1, and in a simple manner, they are defined as follows:

1. An instantiated-component signal $(A)$ is said to be dependent upon another signal if it is
directly affected by that instantiated-component signal (X) and both are within the same partition. Figure 4.4 shows two dependent signals and their ancestors.

![Diagram of two instantiated-component signals dependent upon two other instantiated-component signals]

Figure 4.4: Two Instantiated-component Signals Dependent Upon Two Other Instantiated-Component Signals

2. Similarly, an instantiated-component signal has as many dependents as signals it affects in the same partition. Figure 4.5 shows signal X having as dependents A and B.

![Diagram of instantiated-component signal dependents]

Figure 4.5: Instantiated-component signal dependents

It is important to note that for an instantiated-component signal to have dependents, it must be of mode OUT or BUFFER in its component-declaration part. Similarly, for it to
be dependent upon another signal (have an ancestor), it must be of mode \textit{IN}. Given this, an instantiated-component signal's dependencies are determined by its mode: input or output.\footnote{Mode \textit{buffer} for instantiated-component signals is treated as \textit{out} mode because they are an output from the instantiated component.}

### 4.3.1.1 Instantiated-Component Output Signal Dependency Cases

The set containing all instantiated components ($IC$) in a design can be divided into components to be moved to another partition ($IC_{2bm} \subseteq IC$) and components \textit{NOT} to be moved to another partition ($IC \setminus IC_{2bm} = IC_{N2bm}$). Thus, the dependents of an instantiated-component signal of an instantiated component to be moved can belong to any of the various $IC$ subsets defined previously: (1) instantiated components to be moved $IC_{2bm} \subseteq IC$, (2) instantiated components \textit{not} to be moved $IC \setminus IC_{2bm} = IC_{N2bm}$, or (3) to none of them. In this last case the dependent is an entity signal with mode \textit{out}.

More specifically, a signal may have dependents in $IC_{2bm}$ but not in $IC_{N2bm}$, or vice versa: or it may have dependents in both sets $IC_{2bm}$ and $IC_{N2bm}$, or it may not have dependents in either of these sets. These possibilities comprise the same-partition dependency cases for which the output instantiated-component signals must be checked. Each case implies different modifications to the graph and/or nodes' attributes such as name, mode and direction. All these cases are fully documented in a formal manner in appendix B.2.2.

### 4.3.1.2 Instantiated-Component Input Signal Dependency Cases

An instantiated-component input signal can be dependent upon another signal coming from different instantiated-components. This instantiated-component signal is called its ancestor or driving signal ($i_{c_{driv}}$). When analysing dependencies in instantiated-component signals with mode input, what is really being analysed are the dependencies of the ancestor signal. This
ancestor signal may be part of the set of instantiated components to be moved \((IC_{2bm})\); or part of the set of instantiated components \textit{NOT} to be moved \((IC_{N2bm})\); or part of none of them (in which case the ancestor signal is an entity signal with output mode). And more importantly, it may have other dependents besides the signal being analysed and for which it is ancestor; and those dependents can be among the instantiated components to be moved \((IC_{2bm})\) or in the instantiated components not to be moved \((IC_{N2bm})\). As with the output signals, there is a permutation of all the possible dependency cases. The dependency analysis will tell whether the entity or architecture signal connecting the ancestor and the dependent signal needs to be modified or just moved. The entire set of cases is described in appendix B.2.3.

The reason for analysing the instantiated-component input signals dependencies is because the partitioning algorithm moves one instantiated component at a time. Instantiated components are moved in the order the list of instantiated components to be moved was created. Because of this, the list may or may not be in the order of dependencies. That is, the ancestors are not always analysed before their dependents.

Signals already moved are marked so that when another instantiated-component signal making use of this signal is being moved, this signal’s dependencies are not re-analysed. Moving one component at a time with all its signals allows a simpler partitioning algorithm since dependence relationships between inputs and outputs are not derived recursively. The algorithm also avoids recursive revisiting of instantiated-component nodes.

### 4.3.2 Inter-partition Signal Dependencies

Since the partitioning of the circuit is not automated but selected by the designer, there will very likely be cases in which the designer after doing some partitioning still wants to move
components from a partition $GP_i$ to a partition $GP_j$. It is possible that a component in partition $GP_i$ may have originally been dependent upon a component already placed in partition $GP_j$. The instantiated component in $GP_i$ may be required to be moved back to partition $GP_j$ or to any other partition. In either case the dependencies of this component in other partitions need to be analysed. They are analysed to know whether inter-partition crossing signals and/or internal signals need to be created, eliminated, or modified. Dependency analysis in this subsection occurs only as a result of a previous partitioning. Since this is a new type of dependency, new dependency definitions are required. For a formal definition see appendix B.3.1. Stated in a simple fashion, these definitions and considerations are as follows:

1. Inter-partition dependencies can exist only among entity signals ($ES$) from different partitions. Therefore only entity signals can have inter-partition dependents or ancestors, which can only be entity signals in other partitions.

2. An entity signal in a given partition is said to have dependent(s) in other partition(s) if its mode is $out$ or $buffer$ and if there are other input entity-signals in other partitions being affected by it.

3. An entity signal in a given partition is said to be dependent upon another entity signal in another partition if its mode is $in$, and if that output signal directly affects it.

Note that although the TM-3 topology does not allow inter-partition fanout, in the graph it is allowed. The reason for this is simplicity in the partitioning algorithm. Nonetheless, when the actual VHDL file to represent a partition is generated, it is written in such a way that it complies with the TM-3’s topology (see chapter 6).
From the definitions given above, inter-partition signal dependencies of an instantiated component being moved to another partition can be divided into signals that can have dependents (output or buffer entity signals), and signals that can have ancestors in other partitions (input entity signals).

### 4.3.2.1 Output Signal Inter-partition Dependency Cases

An instantiated component to be moved (\(IC_{2bm}\)), besides the dependents it may have in the same partition, may also have dependents in other partitions. Each of its signals can have dependents in other partitions, and a signal’s dependents may be in the same partition the instantiated component is being moved to, or in other partitions. Analysing an output signal’s dependencies shows whether a signal being moved remains as an entity signal so it can feed dependents in other partitions other than the one it is being moved to; or whether it is left as an internal signal (architecture signal) in the partition it is being moved to (if no dependents in other partitions exist).

Specifically these possibilities are given by the permutation of certain cases. Let \(p_{from}\) be the partition in which the instantiated component to be moved resides; \(pto\) be the partition which the instantiated component is being moved to (\(pto \neq p_{from}\)); and \(e_{p_{from}} \in ES_{ob_{p_{from}}}\) be the entity signal being analysed with mode \(out\) or \(buffer\) that is to be moved to partition \(pto\). The signal being analysed \(e_{p_{from}}\) may or may not have dependents. If it does, these dependents may be only in the \(pto\) partition, or they may be only in partitions other than \(pto\) and \(p_{from}\), or they may be in both. These possibilities are the dependency cases for which a signal is analysed. Figure 4.6 presents an instantiated component to be moved from partition \(p_{from}\) to partition \(pto\) having dependents in partition \(pto\) and in other partitions. Note that there are
many variants to these cases depending on the mode of the signal or in whether the signal was originally an output (or buffer) signal, or was created by the partitioning tool. All cases and variants are formally explained in appendix B.4.

4.3.2.2 Input Signal Inter-Partition Dependency Cases

The second group in which the inter-partition dependency cases can be divided is the set of entity signals that can have ancestors (i.e. input entity signals). When the dependencies of an input entity signal are being analysed, what is actually being analysed, are the dependencies of the ancestor signal (if existent). Analysing the ancestor’s dependencies shows whether the ancestor is required to exist so it can feed dependents in other partitions (if existent).

An input signal’s dependency cases can be derived by permuting the possibilities in which a signal’s dependents may be spread around partitions. These possibilities may be summarised
as follows. An input entity signal \((e_{\text{spfrom}} \in ES_{\text{infrom}})\) in partition \(\text{pfrom}\) may or may not have an ancestor signal. If it does, this ancestor signal \((e_{\text{spdrv}})\) in a partition \(\text{pdrv}\) may or may not be in the same partition in which the signal being analysed is being moved to \((\text{pdrv} = \text{pto} \text{ or } \text{pdrv} \neq \text{pto}, \text{respectively})\). The ancestor signal \((e_{\text{spdrv}})\) may or may not have more dependents in partitions other than \(\text{pfrom}\). Moreover, there may be already an identical signal in the partition to which the signal being analysed is being moved to. All dependency cases with all their variants are formally described in appendix B.5. Figure 4.7 depicts a case where the ancestor of an instantiated-component signal being moved, \(ic_{2bm}\), has more dependents in other partitions. Figure 4.8 represents the same case after the component has been moved.

Note that both figures only show the result of the input entity-signal being moved.

Figure 4.7: Input Signal's Inter-partition Dependence Example Before Moving \(ic_{2bm}\)
Signal \( \text{cs} \text{from} \) is removed because it does not have more same-partition dependents.

Partition where \( \text{cs} \text{from} \) and \( \text{id} \text{from} \) are being moved to

Dependent(s) placed in partition \( \text{pt} \)

Dependent of \( \text{cs} \text{drv} \) now driving the former dependents of \( \text{cs} \text{from} \)

Other Partition With Dependents Upon \( \text{cs} \text{from} \)

Dependent(s) placed in other partition

Partition \( \text{pd} \text{rv} \) With Ancestor of \( \text{cs} \text{from} \)

\( \text{cs} \text{drv} \) (Ancestor)

\( \text{id} \text{from} \)

\( \text{cs} \text{from} \)

\( \text{pt} \)

\( \text{pt} \)

\( \text{pt} \)

Figure 4.8: Input Signal's Inter-partition Dependence Example After Moving \( \text{id} \text{from} \)

4.4 Partitioning Algorithm: Summary

The partitioning algorithm is based on the two types of dependency analysis described throughout this chapter, internal dependencies, and inter-partition dependencies.

Once the user inputs the list of instantiated components to be moved \( IC_{2bm} \), the algorithm moves one component at a time; then the dependencies for each of the signals in the component-being-moved are checked for internal dependencies first, and for inter-partition
dependencies later. This dependency analysis will define whether signals are moved and/or created. If a signal being moved has dependents among the components to be moved, or if it is dependent upon one of the components to be moved, it is marked as “already moved” so another component-to-be-moved does not move it nor checks its dependencies again.

The algorithm analyses the dependencies of a signal and deduces a permutation of possible dependency cases in which the signal can fall. Once the case is deduced, a specific action is taken. Each case leads to a different action to be taken. For these actions not only the dependencies of the signal are analysed, but also other factors, e.g. whether the signal was part of the original design or was created by previous partitioning processes. There are some dependency cases for which the action taken is very similar: in some of them only the mode of the signal is changed, or the number of dependents updated. In terms of the software implementation, those cases share functions.
Chapter 5

Flattening of Hierarchical Designs and Vector Signals

Almost any VHDL design can be described by a single file containing all the logic required for it. However, as designs become large, they are written hierarchically. That is, a top-level file instantiates components formed by subcomponents where these subcomponents can have more subcomponents and so on. Usually each subcomponent is described by one file so it can be used in another later design without modifying the original file. Writing hierarchical VHDL designs is a good design style, however for partitioning it is better to have a flat design where all the design’s components are located in a single file. The reason for this is that a hierarchical design in its top-level file usually contains the instantiation of large components (usually a cluster of smaller components in their lower hierarchies), thus leading to a coarse-grain netlist. Partitioning such a netlist could be impossible since one of its components may be so large that it would not fit in a single FPGA. Therefore, a netlist with smaller components (a finer-grain netlist) can more likely be partitioned i.e. its partitionability is increased. A way to achieve this finer granularity in a hierarchical design is by flattening it. In this manner, the partitioning tool, before showing the designer which components are available to be moved, flattens the design if it has been written hierarchically and if the user wants to flatten the design.
Another way of increasing a design’s partitionability is by flattening buses that are partially shared by different components in the design\(^1\). A vector signal (bus signal) that is not being partially shared by different components can be treated as a single signal of vector type since all signals comprised by the vector have the same dependencies. However, in partially-shared vector signals each of the signals in the vector group have their own dependencies. If the vector signal were not flattened, dependents of the vector signal would have to be moved together to another partition, or kept in the same partition to respect the design’s dependencies. Consequently, if those dependent components are so large that they cannot fit in a single FPGA, the design could not be realised because these components cannot be separated from each other to fit them in the available FPGAs. This bus flattening process occurs whenever there are partially-shared buses.

In this chapter Section 5.1 presents a scheme for flattening a hierarchical design. Section 5.2 shows how vector signals are separated into single signals when it is necessary.

### 5.1 Hierarchical Design Flattening

A VHDL hierarchical design can be represented by means of a tree data structure. The top-level file of the design represents the root of the tree and the instantiated components its children. The instantiated components inside other instantiated components are children of the children of the tree, etc. Figure 5.1(a), shows a hierarchical design, where the dashed squares represent the lowest hierarchy, and the thickest squares represent the highest hierarchy. This figure shows that child instantiated components may have the same name as the parent instantiated

\(^{1}\)In this context, flattening refers to separating bus signals into single signals
component\textsuperscript{2} or other components in the upper hierarchy. This is a problem when flattening the design. It is addressed in section 5.1.1. Figure 5.1(b), depicts the design’s representation with a tree data structure. The lowest levels of the hierarchy tree (leaves) represent instantiated components with no subcomponents.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5.1.png}
\caption{A Hierarchical Design And Its Representation With A Tree Data Structure}
\end{figure}

As can be seen in Fig. 5.2, to flatten a design it is necessary to replace parent components with their children components i.e. remove hierarchies. To do this, the flattening algorithm uses a depth-first search algorithm \cite{1} to find the lowest level of an instantiated component’s hierarchy. Once the lowest level of the hierarchy has been found it is recursively brought to its upper hierarchy level until it reaches the top-level file. Thus, every time the lower-level

\textsuperscript{2}If a child instantiated component has the same name as the parent instantiated component, it must describe a different design i.e. have another component declaration
members of a hierarchy are moved one level up, the parent instantiated component is removed from the tree and replaced with its child components and the graph $G_{IC}$ associated with these components. This graph is generated when the file containing the instantiated component’s description is parsed. The lowest level nodes are brought to the upper level before an instantiated component is replaced with its children. Figure 5.2(a) depicts the design in Figure 5.1 after the lowest-level components have replaced instantiated components in their upper-level hierarchy, i.e. the lowest level of Fig. 5.1 has been flattened. Figure 5.2 shows the design after it has been completely flattened.

![Diagram of flattened hierarchy](image)

**Figure 5.2: Instantiated Components Replaced By Their Children**

Each instantiated component is assumed to have a VHDL file describing it. When the depth-first algorithm finds an instantiated component, the file associated to it is opened, parsed and a new tree data structure representing it is generated. This new tree becomes the parent that is analysed and checked for instantiated components. The algorithm continues recursively.
for all instantiated components found; at the same time it generates the proper graphs from the files associated to them and joins them to the parent graph.

5.1.1 Merger of Children and Parent Graphs

Every time a file is opened and parsed, a graph $G_{IC}$ with properties and characteristics like those described in chapters 3 and 4 is created. This graph $G_{IC}$ is merged with its parent graph $G_{Parent}$ when the parent instantiated component is replaced with its subcomponents. This happens recursively until all the instantiated components of the original top-level file have been replaced with its subcomponents. Instantiated components in the current top-level file with no subcomponents are skipped and no graph representing them is generated.

The merged graph must contain information to comply with VHDL syntax, and most of all, it must represent the same design but without hierarchies. For this graph to keep representing the same design, the child graphs, $G_{IC}$, must inherit all ancestors and dependents the parent instantiated component had.

When the graph $G_{IC}$ is joined to the parent graph, $G_{Parent}$, such that $G_{Parent} = G_{Parent} \cup G_{IC} \setminus (ES \in G_{IC})$, all its entity signals are removed because they are no longer required. Entity signals are ports used to interface a component to external circuits. Since the graph $G_{IC}$ is being moved to an upper hierarchy level, it is no longer an external circuit to the parent graph, $G_{Parent}$, so its entity signals are no longer required. Now the instantiated components $IC \in G_{IC}$ also belong to the parent graph $G_{Parent}$ and their signals can have dependent or ancestor instantiated-component signals without boundary (entity) signals.

Figure 5.3 depicts an example of a graph $G_{IC}$ before and after being brought one level up in the hierarchy to where the parent component (CCS) resided in the parent graph $G_{IC}$. 
Figure 5.3: Graph being moved one level up inherits all the connections (ancestors and dependents) of the instantiated component and loses its interconnection signals (entity signals)
However, there are two main problems for the new parent graph $G_{\text{Parent}}$ to comply with VHDL syntax. First, the name of the architecture signals and instantiated components recently brought to the upper level; and second, the component declaration of the instantiated components in the upper and lower levels.

If there were architecture signals in the lower-level graph ($G_{IC}$) with the same name as the architecture signals in the parent graph, the VHDL file that the tool generates would not compile because duplicate identifiers would be found. This also applies to the names of the instantiated components in both graphs. Nevertheless, this problem can be easily overcome by pre-appending the name of the parent instantiated component to all components and signals in the lower-level graph and by checking that the new names do not already exist.

The second problem is overcome by bringing the component declarations one level up if there is not already an existing component declaration with the same characteristics. Only non-existent component declarations are moved to the upper level graph. The software tool does not currently support packages, however library components can be instantiated if their component declarations are in the same file they are being used.

### 5.2 Partially Shared Vector Signals

Designs frequently contain internal and external buses (e.g. for data and addresses). In VHDL, architecture and entity signals of *vector* type represent these kinds of buses. Signals with mode *buffer* can represent both internal and external buses because they are port signals that can also interconnect internal components. The width of the bus is specified in the signal declaration and it represents the number of signals grouped in a bus. Section 2.2.1.2 gives a brief explanation of this.
As with any other signal in VHDL, signals of vector type can have dependents or ancestors in other signals and their associated instantiated components. However, one vector signal can have dependents of different widths. Since a bus represents a collection of signals $S_{vector}\{s_1, \ldots, s_w\}$ where $w$ is the width of the vector signal, dependent components can make use of all or only some of the signals in the group. In other words, dependent instantiated components can have vector signals of various widths, $w_{sc}$, which are not necessarily equal to the width of the ancestor bus. In this case the bus is said to be split and shared by various instantiated components.

When a design is first read and parsed, the generated graph does not consider the type of signals being incorporated to the graph. For this reason, a signal of type vector is represented by the same kind of node representing a signal of a non-vector type. If a vector signal does not have various dependents with different bus widths, the graph can be partitioned in the same way as explained in chapter 4 because all the signals comprised in the bus would have the same dependencies.

If there are shared busses with different widths, the graph $G$ would have to be partitioned in a different way, imposing some restrictions to the way the graph could be partitioned. Figure 5.4 depicts a component ($IC_1$) having an output vector signal ($Data$) partially shared by other components ($IC_2$, $IC_3$, $IC_4$) each using a different part of the bus. A better option would be to modify the graph $G$ so the same algorithm can be used and no restrictions are imposed. This solution separates and treats all partially-shared vector signals as single signals.

The modification of vector signals implies modifying the entity signals, the architecture signals, and the instantiated-components nodes. This modification also implies assigning each signal node its own dependencies.
5.2.1 Entity and Architecture Signals Generation

When the parser realises the design has buses partially shared by different instantiated-component signals, the entity and architecture signals associated with them are marked. These marked signals are later separated into single signals. To represent a vector signal as a group of signals it is necessary to create as many nodes as the signal’s width minus one. The names of the signals are changed from the original name to the original name plus the index appended to it. The type of the signals is also changed. For example, if the type of a signal is "std_logic_vector", it is changed to the associated type that represents a single signal of the same type: "std_logic" in this case. This is applied to the most common vector types.
The algorithm does not currently support user-defined types. Its implementation would also require supporting the corresponding VHDL constructs for user-defined types. If these subtypes were defined inside a VHDL package, support for package constructs would also be required.

Table 5.1 shows two common signal types. The first column presents signals with vector types and the second column shows the newly acquired names and types for describing them as single signals. All of them are now different signals. Because of that, their dependencies must be changed accordingly.

Since the vector signal was created when building the graph, it already contains information regarding its dependencies. Who it depends upon and which are its dependents is stored in it. However, when building the graph, this information is not specific to each of the signals being represented by the vector signal because it is being treated as a single signal.

The reason for not considering single signals from the beginning is because it is \( w - 1 \) times faster to generate a single vector signal node than it is to generate \( w \) vector signals. Moreover, the dependencies of all single signals would have to be checked and added to the graph node. And in most cases these dependencies would be the same because split vectors do not appear in all instantiated components; in fact, they appear less often than fully-used instantiated-component vector signals.
The dependencies of the newly created single signals are assigned one by one. The mode of the signal defines whether dependents and/or ancestors are added to its corresponding list of dependents or ancestors. To assign a signal's dependents, besides the mode of the signal, it is also necessary to know whether the signal in question is being used by some signal in the list of dependents stored in the original vector signal. All dependents specific to the single signal will be added to its own list of dependents. As for the ancestor of the single signal, the original vector signal also has a list of the vector-signal’s drivers (ancestors).

### 5.2.2 Modification of Formal and Actual Parts in Instantiated Components

As described in section 2.2.3, the VHDL language allows instantiated components to be interconnected by architecture and entity signals using a formal and an actual part for each and every one of its declared signals. The formal part represents the signal in the instantiated component, and the actual part represents the entity or architecture signal to which it is actually being connected. For vector signals, the width of the bus must be specified if the formal and actual parts have different widths and/or order in the assignment. This width can be as small as one single signal for both formal and actual parts. In such a case, the number of formal parts must be repeated as many times as the width of the instantiated component signal (as shown in table 5.2).

Since the architecture and entity signals have been ungrouped and now are single signals

---

3 This list will not exist if the signal is an entity signal of mode `out`.

4 Because they represent a set of signals, vector signals are the only type of signals that can have a list of multiple drivers. If a vector signal has multiple drivers it implies that it is being shared by various instantiated components. Note that this list will not exist if the vector signal is an entity signal of mode `in`. However, if the signal's mode is `buffer`, both lists: ancestors and dependents can exist.

5 The formal part in an instantiated component is not strictly required by the VHDL language (see section 2.2.3.2), but for simplicity in the algorithms, the partitioning tool requires it.
Table 5.2: Vector signal transformation into a group of single signals

<table>
<thead>
<tr>
<th>signal declaration</th>
<th>formal part in instantiated component</th>
</tr>
</thead>
</table>
| formal part bus (2 downto 0) | \ldots, formal part bus (2) \rightarrow actual part signal Y,  
formal part bus (1) = actual part bus X (29),  
formal part bus (0) = actual part bus X (28), \ldots |

Table 5.3: Formal and actual parts in an instantiated component before and after they are flattened. Assume that the signal “formal” was declared as “signal: in std_logic_vector (0 to 3)” and the signal “actual.N” was declared as “signal actual: in std_logic_vector (7 downto 4)”

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>formal part</td>
<td>actual part</td>
</tr>
</tbody>
</table>
| formal => actual | formal (0) => actual_7,  
formal (1) => actual_6,  
formal (2) => actual_5,  
formal (3) => actual_4, | |

with different names and types, it is necessary to modify the formal and actual parts to comply with the VHDL syntax of instantiated components.

There are several forms in which a signal’s formal and actual parts could have been written by the designer. Before the ungrouping of a bus, and before modifying the actual and formal parts, it is necessary to recognize these cases. Once these cases have been identified, the formal part of an instantiated-component vector signals is separated into \( w - 1 \) signals. They are then assigned the corresponding entity and architecture signals that represent the actual part. Table 5.3 gives a brief example of the formal and actual parts of an instantiated-component after they have been ungrouped.

Let the bus going to components \( IC_3 \) and \( IC_4 \) in figure 5.4 be \( S_{\text{vector}} \). Note that both components make use of only some of its signals. This divides \( S_{\text{vector}} \) into two subsets \( S_{\text{vector}}_1 \subseteq S_{\text{vector}} \) and \( S_{\text{vector}}_2 \subseteq S_{\text{vector}} \). If \( S_{\text{vector}}_1 \) and \( S_{\text{vector}}_2 \) did not intersect, it may be a
good idea to only separate the signals into two sets instead of separating them into \( w \) subsets of only one element. However if they intersect, they would have to be separated into three subsets. Since other components could make use of the signal, \( S_{\text{vector}} \) must be separated in different sets than the ones required by \( IC_3 \) and \( IC_4 \), and the idea becomes more complex. If these sets still intersect, the idea would become almost impossible and slow. Therefore, it is faster to simply separate the signals into single-member sets and assign their corresponding dependencies.
Chapter 6

VHDL-Code Generation

The last step in the partitioning process is the actual VHDL code generation. Once the user has selected the components to be placed in each partition, VHDL files are generated out of the graphs representing the design's partitions. The information in each of the graphs is taken and structured in such a manner that the output files comply with the VHDL syntax. The generation of the VHDL files is done in a straightforward manner since all the information regarding components, instantiated components, signals, types, modes, names, etc. is explicit in the graphs. No information has to be deduced from the graphs. The only problem is that the VHDL files should consider the TM-3's topology. Among the graphs generated by the partitioning tool, inter-partition fan-out is allowed; however the topology of the TM-3 does not support it. Overcoming this situation is also described in this chapter.

6.1 VHDL files generation

Subgraphs \( G_{P_n} \) representing the partitions of the graph \( G \) contain detailed information regarding entity signals, architecture signals, component declarations, and instantiated components. Entity-signal nodes contain names, modes, types, and sizes for them; component-declaration nodes contain names for them and the signals plus the same information as the entity and ar-
chitecture signals for each of their own signals. The graph contains all the required information for writing a VHDL file.

As stated in chapter 4, representing entity and architecture signals as vertices of the graph makes the code-generation process faster and easier. It is faster to read a set of architecture signal nodes with all the information required by the VHDL syntax than it is to traverse all the instantiated-component nodes deducing information from its edges to generate the proper signals with the proper syntax for every partition. Because of the way the graph is built, creating the proper VHDL files is a matter of reading the corresponding graph nodes and writing them to a file.

To comply with VHDL syntax, the information contained in the graph nodes needs to be formatted when output to a VHDL file. Then the basic structure of a VHDL file is followed: entity and architecture declarations with their corresponding signals and components. One VHDL file is created for each partition.

6.1.1 No Inter-chip Fanout

The previously described method of creating the corresponding VHDL files for each partition applies to most of the graph nodes except to those of entity signals having dependents in more than one partition as shown in Figure 6.1. Subpartition graphs allow fan-out among their interconnecting entity signals, but the TM-3’s topology was not designed to support inter-chip fanout, i.e. no given wire connects to more than one FPGAs. Therefore the VHDL files must be written in a way that can be used by the TM-3.

To generate VHDL files compliant with the TM-3’s topology and the VHDL syntax, entity-signal nodes interconnecting more than one subpartition must be transformed into a set of
entity-signal nodes with only one inter-partition dependent.

The number of new entity signals equals the number of fanout dependents minus one. Their name is the same as the original signal with a number appended to them. Their corresponding dependent's name is also changed so the designer or the software can easily identify them when the pin assignment is performed\textsuperscript{1}.

A simple solution would be to change the mode of the signal to \textit{buffer} if it is \textit{out}. If it already is \textit{buffer}, it remains the same. This change in the signal's mode allows the signal to have internal dependents. Since it can now have dependents, its new dependents are the

\footnote{\textsuperscript{1}It is important to note that when the pin assignment is performed, the design must conform to the TM-3 topology i.e. the number of crossing pins must not exceed the number specified by the TM-3's topology.}
recently created entity signals. When generating the file, the original signal with mode buffer would be assigned to these entity signals. Figure 6.2 shows a signal having interpartition dependents after its entity signals have been set to use different FPGA pins.

![Figure 6.2: Inter-partition dependents using different FPGA pins](image)

Table 6.1 shows how in the actual VHDL code the entity signals are modified in both the driving and dependent partitions to adjust the files to the TM-3 topology.

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>ancestor partition</td>
<td>dependents in different partitions</td>
</tr>
<tr>
<td>signdep : out bit</td>
<td>signdep : in bit</td>
</tr>
<tr>
<td>signdep : in bit</td>
<td>signdep : in bit</td>
</tr>
<tr>
<td>signdep : in bit</td>
<td>signdep_2 : out bit</td>
</tr>
</tbody>
</table>

Table 6.1: VHDL signals with more than one inter-partition dependent signal
This solution works fine with the Altera Max+Plus II [38] VHDL compiler, but not for the Synopsys VHDL compiler [39]. Synopsys does not allow assigning entity buffer signals to instantiated-component signals whose mode declaration is other than buffer. Therefore a solution for overcoming this situation is declaring an architecture signal and assigning it to the newly created entity signals. Figure 6.3 depicts this solution.

Figure 6.3: Inter-partition dependents using different FPGA pins and an Architecture Signal

Once the VHDL files have been generated, the design is ready to be synthesised by any commercial tool. Each file describes the portion of the design corresponding to one FPGA. After each file has been synthesised, it can be mapped and downloaded onto its corresponding TM-3 FPGA.
Chapter 7

Partitioning of two VHDL designs

Having detailed all aspects of the partitioning software tool, this chapter presents the partitioning of two actual VHDL designs. The first design mainly tests the dependency analysis algorithm and the VHDL code generation process. The second design, unlike the first one, is a real design. It describes an alarm clock implementation. Partitioning of this design is intended to show an example of the entire partitioning process. The partitioning of this design shows step by step how the graphical user interface is used by providing some snapshots of it. This second design also tests all stages of the partitioning software tool since it is a hierarchical design and has vector signals being partially shared by various components.

7.1 First VHDL Design

The first step in testing the partitioning tool was to develop an input VHDL design that permitted a simple debugging process. However to test the basic functionality of the tool, this design should have a certain level of complexity. The first VHDL design that was used for testing purposes does not represent any actual circuit. It represents a netlist of instantiated components where one of them ("1" in figure 7.1) contains signals each containing a different same-partition dependency case. The dependencies of these signals represent all the same-
partition dependencies described in section 4.3.1 and appendix B.2. Figure 7.1 depicts the design's graph. In this figure all components to be moved to a different partition have the label $IC_{2bm}$, and the components that remain in the same partition are marked as $IC_{N2bm}$. The component having all same-partition dependency cases is marked as "uno". The actual VHDL code for this simple design is given in appendix C.1.1, and its schematic diagram is given in appendix C.1.2. Figure 7.2 and Figure 7.3 depicts the first VHDL design after it has been partitioned. The resulting VHDL code and the schematics for each partition are presented in appendices C.1.3 and C.1.4, respectively. As it can be seen from figures 7.1, 7.2 and 7.3 the design preserved its dependencies after it was partitioned. Note that for each signal of instantiated component "uno" a different action was taken according to the dependency case that each signal had.
Figure 7.2: Partition One of First VHDL Design

Figure 7.3: Partition Two of First VHDL Design
7.2 Partitioning Of A Real VHDL Design

After the basic dependency analysis algorithm was verified, a second more complex design was chosen to test the partitioning tool thoroughly. The first step for performing a more complete testing of the software tool was selecting a VHDL design to be partitioned. For the design to be considered it had to be simple enough to allow debugging of the software if required. But it should also have to be complex enough to test all steps of the entire tool, from recursively parsing files to the generation of flattened VHDL files for each partition.

7.2.1 Selected VHDL Design

The selected VHDL file to be partitioned had to have three main characteristics. First, it had to be a hierarchical design so flattening and recursive parsing of the design could be tested. Second, the design had to have vector signals being partially shared by different components so the bus flattening process could be verified. And third, test vectors should be provided by the designer so the resulting partitioned-design's simulation could be compared with the original design's simulation.

The selected VHDL design is an alarm clock implementation since it contains the three characteristics mentioned above. It was taken from the Synopsys Quick Start Guide[39]. Figure 7.4 depicts the hierarchy of this design.

![Figure 7.4: Alarm Clock Design Hierarchy](image)
Figure 7.5 presents a block diagram of the top-level file before it has been flattened. Appendix C.2.1 contains the VHDL source code for the top-level hierarchy of the design.

Figure 7.5: Alarm Clock Block Diagram

7.2.2 Parsing Of Files And Design Flattening

The first step to the partitioning process is selecting the VHDL file to be analysed. Figure 7.6 depicts the selection of the top-level file of the design through the graphical user interface (GUI). This GUI is written using the TCL/TK scripting language[44] and communicates with the rest of the partitioning software with pipes[46].

Once the top-level file is selected, the user is asked whether or not the design is to be flattened. Since in this example the design is to be flattened, all files in the design are parsed and brought to a new top-level file. The software tool, when searching for the design's files, assumes that all VHDL files have the same name as the entity they are describing. Figure 7.7
Figure 7.6: Top-Level File Selection

shows all the design's components without hierarchies. Appendix C.2.3 shows the VHDL source code for this flattened design, and appendix C.2.4 a schematic diagram for it.

Figure 7.7: Instantiated Components In Flattened Design
7.2.3 Design Partitioning and Generation of VHDL Files

At this point, the partitioning software expects the user to select a set of instantiated components to be moved to any of the four FPGAs of the TM-3. This set can be chosen in the graphical user interface (GUI) by dragging and dropping the names of the instantiated components to be moved into the wanted FPGA (partition). Figure 7.8 shows a set of instantiated components already placed in partition number three, and a set of selected instantiated components being moved from partition number one to partition number two.

![Figure 7.8: Selected Instantiated Components Being Moved From Partition 1 to Partition 2](image)

The actual graph partitioning occurs when the designer is ready to generate the corresponding VHDL files for each partition. Figure 7.9 shows how by selecting “Generate and Save All” the GUI is told to generate all VHDL files. The prefix \textit{FPGA}<N> is added to the name of the design, where \textit{<N>} is the name of the partition. In this manner, a design file called “top.vhd”
Chapter 7. Partitioning of two VHDL designs

generates the files FPGA1top.vhd, FPGA2top.vhd, etc., for each partition. These files are ready to be compiled using any commercial tool. Appendix C.2.5 and C.2.6 show the resulting VHDL files for each partition and their corresponding schematic diagrams, respectively.

Figure 7.9: Files Generation

7.3 Simulation Of Generated VHDL Files

To verify that the generated VHDL files for each partition represent the original design. All generated VHDL files can be instantiated as components in a new VHDL design file so it can be simulated. The simulation of the new VHDL design containing the four instantiated components (or less depending on the number of partitions required by the designer), where each component represents a partition, is compared to the original design. This is the reason why a design with test vectors provided by the designer was chosen to be partitioned.

Instantiating the components representing the partitions of the design into the new VHDL design is done manually. The new VHDL file has as entity declaration the same entity declaration the original design had. It can be copied and pasted. To interconnect the instantiated components, if a signal with the same name as the formal part does not exist in the entity signals, a new architecture signal, if it does not already exist, is created. An automated generation of this VHDL file is seen as future work. Appendix C.2.7 shows the new VHDL file used to simulate the partitioned design.
7.4 Iterative Design Partitioning

Once the VHDL files have been generated the process of partitioning a design is finished. After the VHDL files representing each partition have been compiled it is possible that the contained logic in one of the files does not fit in an FPGA. Or it is also possible that the designer wants to try another partitioning approach. In either case, redoing the entire partitioning process would be tedious, especially in circuits with many components. Because of this, the software tool is provided with a way of remembering which components were present in each partition at a given time. The option “gendatfile” in figure 7.9 will generate a data file with the name of the instantiated components present in each partition at the moment of generating the file. The option “readdatfile” will open those files and show the designer a previous partitioning approach.
Chapter 8

Conclusion and Future Work

8.1 Conclusion

The purpose of this work was to develop a VHDL-code partitioning tool for the Transmogrifier 3 (TM-3). The development of this work had three main objectives. First, reading designs written in the VHDL language. Second, presenting the design’s components to the user in a simple fashion so they could be manually placed in any desired partition. And third, writing output VHDL files with the partitioning information provided by the user so the generated output VHDL files could be synthesised and downloaded onto the corresponding TM-3’s FPGAs using any commercial tool. These objectives were met, although only a subset of the complex VHDL language is supported to limit the scope of the work. Supporting a broader range of VHDL is the first priority for future work.

Partitioning of a given VHDL design is manually performed. That is, the user has to provide the location of the design’s components in any of the four TM-3’s possible partitions. Manually partitioning a design allows the designer to choose in which specific FPGA a certain component is to be placed. For example in the TM-3 only FPGA number three has direct access to video I/O functions[45]. Should the user want to assign all components related to video generation to FPGA number 3 so the timing constraints in a design are met (or for any other
reason), it is only a matter of selecting the proper instantiated components and placing them into the corresponding partition. Because of this manual placement of the design's components, unlike most automated graph partitioning algorithms, the resulting partitions may not be balanced and the number of crossing edges not minimised. Nevertheless good partitions may be achieved since it is assumed that designer knows the design well enough to produce good enough partitions. Furthermore, since the software described in this thesis should be seen as a first step to creating a more complex and automated partitioning tool, this first step of manually partitioning a design permits an easier testing and debugging of the proposed algorithms than a more complex implementation.

The first step to partitioning a VHDL design was representing it with a graph data structure. The proposed graph $G$ contained internal signals (architecture signals) and primary inputs and outputs (entity or port signals) of the netlist as nodes ($V$) of the graph and not as edges ($E$) as it is commonly done. The reason for taking this approach was to make a simple file generation process. The resulting file generation process is simple since no architecture or entity signals information has to be deduced from the edges of the graph because this information is explicit as a node. However, making a simple file generation process leads to a more difficult partitioning algorithm since more nodes with more information had to created and/or modified during the partitioning process. It would have been interesting to try making the partitioning process simpler and the file generation process more complex. This might simplify the application of automated partitioning algorithms to the software tool.

During the hierarchical designs flattening process, files describing components in the hierarchy are assumed to have the same name of the component they describe. With this approach the designer only has to worry about giving the same name to both the file and the entity in it.
However this may not be very flexible if the designer wants to use another file to describe the component the previous file described. A more flexible way of searching for the design’s files could be asking the user for a list of files included in the design and perform the search process among those files. Currently the partitioning tool asks whether the design needs to be flattened and if so, flattens it all. An improvement to this could be flattening only the components that the user wants to flatten. This would require more communication between the software tool and the GUI with minor modifications. So far flattening the desired component is possible if the file for that component is used as the top-level of a hierarchy and then the generated files are used by the real top-level of the hierarchy as instantiated components.

The main contribution of this work is the partitioning of VHDL designs at a coarse-grain granularity level. This work assumes the use of large FPGAs where large components can be synthesised. In general it can be said that this is a good approach for partitioning circuits. However, if the designs to be partitioned contain components so large that don’t fit in a single FPGA this approach will fail to properly partition a design. Nevertheless, those designs can still be partitioned if they are synthesised and then translated back into VHDL. This process will reduce the size of the design’s components, thus leading to a fine-grain granularity partitioning. However, at this level of partitioning an automated partitioning algorithm would be indispensable since the number of small components will be so large that it may be impossible (let aside tedious) to create good enough partitions. Another solution for this may be synthesising only the large component that does not fit and replacing with it the large un-synthesised file. Nevertheless, it would be interesting to partitioning large (maybe commercial) designs with the proposed partitioning tool and compare its results with an automated fine-grain partitioning algorithm.
This partitioning tool is a first implementation of what could be a more complex partitioning tool. Many aspects of this work can be improved. And more interestingly, many other algorithms and approaches can be added to the basic implementation so it can become more useful. The following section proposes some directions for future improvements. Using and testing the tool described in this work may lead to further improvements.

8.2 Future Work

The scope of this work encloses only manual logic partitioning at the VHDL-code level. However the field of logic partitioning for logic emulators is very ample and many of the concepts presented in recent literature can be applied to the tool presented in this thesis. However, although it might not be necessary for certain projects, the first work to be done is supporting a broader range of the VHDL language such as packages, generics, generates, complex signal assignments and signals whose mode is \textit{INOUT}.

8.2.1 Inter-Partition Pin Multiplexing

Since the partitioning of the VHDL designs is done manually, the number of crossing wires between partitions is not minimised. Inter-chip communication has been proven to be one of the main reasons for not fully utilising all the logic an FPGA can hold. One of the efforts to overcome this problem multiplexes inter-partition pins, transmitting serially over one pin information that otherwise would be transmitted using various parallel pins[20]. A significantly simplified version of that work could be applied to the partitioning tool described in this thesis. This could be done by instantiating parallel-to-serial and serial-to-parallel converters (such as shift registers). A state machine controlling them would also be needed. A simple dependency
analysis would be required to schedule the transmitted signals. For example, if a bus crossing
chips were to be multiplexed, each of its signals would have to be assigned to a different
transmitter (and in the same position in all of them) so the data would be received without
alterations. Other signals could share the same transmitters. Many topics would still need be
investigated, such as the width of the transmitters and receivers so maximum clock performance
could be achieved, how much resources of the FPGA they use, etc. This pin multiplexing could
be followed by an automated logic partitioning.

8.2.2 Automated Partitioning

Automated partitioning could be done either after, or before the pin multiplexing technique.
However before any automated logic partitioning algorithm is implemented it would be inter-
esting to support constrained VHDL so the user could receive some information on whether
the timing requirements are being met. Furthermore, since the partitioning occurs at the high
level language there is no information about the size of a component (equivalent FPGA gates).
Therefore, it would be necessary to find a way of getting information on the size of the de-
sign's components so the user could know, before synthesising the design's partitions, whether
a partition fits in the designated FPGA. This would allow the automated partitioning algorithm
to assign a weight to all the design's components.

8.2.2.1 Critical Path Reduction

Critical paths crossing chips are one of the main reasons for logic emulators not performing
well at higher clock speeds. Two specific partitioning algorithms seem to be very beneficial for
reducing critical paths crossing various chips. A combination of both may be even better. The
first approach uses cone partitioning [41] from one primary output to the primary input signals
that affect that output; the second one uses component replication in partitions [21]. The first algorithm places the overlapping cones in one of the two partitions and an incomplete cone in the other partition. The combination of both algorithms would result in an algorithm that used cone partitioning with replication of the overlapping cones (i.e. the components in them).
Appendix A

Set Notation and Definitions

A.1 Set Notation and Logical Terms

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\emptyset</td>
<td>empty set</td>
</tr>
<tr>
<td>\cap</td>
<td>intersection</td>
</tr>
<tr>
<td>\cup</td>
<td>union</td>
</tr>
<tr>
<td>\setminus</td>
<td>set minus</td>
</tr>
<tr>
<td>\in</td>
<td>element in</td>
</tr>
<tr>
<td>\subseteq</td>
<td>subset</td>
</tr>
<tr>
<td>\subseteqq</td>
<td>subset or equal</td>
</tr>
<tr>
<td>\sum</td>
<td>sum</td>
</tr>
<tr>
<td>\exists</td>
<td>such that</td>
</tr>
<tr>
<td>\exists !</td>
<td>existential quantifier (exists)</td>
</tr>
<tr>
<td>\forall</td>
<td>universal quantifier (for all)</td>
</tr>
<tr>
<td>\lor</td>
<td>disjunction</td>
</tr>
<tr>
<td>\land</td>
<td>conjunction</td>
</tr>
<tr>
<td>\rightarrow</td>
<td>implication</td>
</tr>
<tr>
<td>\Rightarrow</td>
<td>implication</td>
</tr>
</tbody>
</table>

Table A.1: Set Notation and Logical Terms

A.2 Set Definitions

This appendix presents as a quick reference for the set definitions used throughout this thesis.

1. $ES_{in} = \{es_1 \cdots es_{N_{ES_{in}}}\}$ is the set of port signals with mode $IN$; where $N_{ES_{in}}$ is the total number of port signals with this mode.
2. \( ES_{out} = \{ es_1 \cdots es_{N_{ES_{out}}} \} \) is the set of port signals with mode \( OUT \): where \( N_{ES_{out}} \) is the total number of port signals with this mode.

3. \( ES_{buffer} = \{ es_1 \cdots es_{N_{ES_{buffer}}} \} \) is the set of port signals with mode \( BUFFER \): where \( N_{ES_{buffer}} \) is the total number of port signals with this mode. Note that this set may be empty.

4. \( ES = ES_{in} \cup ES_{out} \cup ES_{buffer} = \{ es_1, \cdots , es_{N_{ES}} \} \) is the set of port signals with any mode: where \( N_{ES} \) is the total number of port signals in the given design. and \( N_{ES} = N_{ES_{in}} + N_{ES_{out}} + N_{ES_{buffer}} \).

5. \( AS = \{ as_1 \cdots as_{N_{AS}} \} \) is the set of architecture signals in the design, where \( N_{AS} \) is the total number.

6. \( IC = \{ ic_1 . \cdots , ic_{N_{IC}} \} \) is the set of instantiated components in the design: where \( N_{IC} \) is the total number.

7. \( S_{in} = \{ s_{in_1}, \cdots \} \) is the set of input edges \( E(ic) \) at any given \( ic \in IC \). In other words, this represents the set of signals in an instantiated component whose mode in the component declaration is \( IN \).

8. \( S_{ob} = \{ s_{ob_1}, \cdots \} \) is the set of output edges \( E(ic) \) at any given \( ic \in IC \). In other words, this represents the set of signals in an instantiated component whose mode in the component declaration is \( OUT \) or \( BUFFER \). Note that buffer signals are treated as output signals for assigning a direction to the edges at an instantiated component.

9. \( S = S_{in} \cup S_{ob} \) is the set of edges \( E(ic) \) at any given \( ic \in IC \). In other words, this represents the set of signals in an instantiated component whose mode in the component
declaration is \textit{IN}, \textit{OUT} or \textit{BUFFER}. Note that an instantiated component \textit{ic} could be seen as a set of instantiated-component signals.

10. $IC_{2bm} \subseteq IC$ Instantiated components to be moved from a partition to another one.

11. $ic_{2bm} \subseteq IC_{2bm}$ Instantiated component being moved at a given time.

12. $ic_{driv} \subset IC$ Ancestor instantiated component.

13. $ES_{OB}$ Entity signals with mode \textit{OUT} or \textit{BUFFER}

14. $es_{pdrv_{ob}} \in ES_{OB_{pto}}$ Ancestor entity signal in partition \textit{pto}.
Appendix B

Signal Dependencies: Formal Analysis and Representation

B.1 Introduction

This appendix presents in a formal manner all the signal dependency cases described in chapter 4. Signal dependencies are divided into same-partition dependencies, and inter-partition dependencies.

B.2 Same-Partition Signal Dependencies

Let $IC_{2bm} \subseteq IC$ be the set of instantiated components selected by the user to be moved from partition $G_{P_k}$ to partition $G_{P_m}$, where $(k \neq m)$ and $G_{P_k}, G_{P_m} \in G$. Before moving each element $ic_{2bm} \subseteq IC_{2bm}$, same-partition dependencies for each of its signals must be checked to know whether the signal in question has dependent or ancestor signals within the same partition, and whether its dependents or ancestors are among the set of components to be moved $IC_{2bm}$.

B.2.1 Signal Dependency Definitions

1. An instantiated-component signal is said to have an ancestor in another signal if the ancestor signal directly affects it. Considering the properties of the type of graphs built by
the partitioning tool, an \textit{instantiated-component signal} \( s_y \in \text{ic}_y \) is said to be \textit{dependent} upon another \textit{instantiated-component signal} \( s_x \in \text{ic}_x \), if there exists a directed path \( P(s_x, s_y) \), and for this path there exists both: an edge \( E(s_x, as_z) \) and an edge \( E(s_y, as_z) \); or both: an edge \( E(es_{buffer_z}, s_y) \) and an edge \( E(ic_{P_z}, es_{buffer_z}) \). Signal \( s_x \) is the ancestor of \( s_y \).

2. Similarly, a signal \( s_x \in \text{ic}_x \) has \textit{N dependents} \( \{s_1, \ldots, s_N\} \) in \( G \) if for each \( N \) there exists a path \( P(s_x, s_N) \) and for each path there exists \( E(as_z, s_N) \) and \( E(s_x, as_z) \); or if for each \( N \) there exists a path \( P(s_x, s_N) \) and for each path there exists \( E(es_{buffer_z}, s_N) \) and \( E(s_x, es_{buffer_z}) \). Note that \( E(as_z, s_N) \) and \( E(es_{buffer_z}, s_N) \) can be part of a path going from one signal to several other because fan-out is allowed.

By these definitions it can be concluded that for an \textit{instantiated-component signal} to have dependents, it must be declared mode \textit{OUT} or \textit{BUFFER}. And similarly, for it to be dependent upon another signal (have an ancestor), it must be declared mode \textit{IN}.

\textbf{B.2.2 Same-Partition Instantiated-Component Output Signal Dependency Cases}

\textbf{Case 1:} \( IC_{dep} \cap IC_{2bm} \neq \emptyset \) \( \exists \) \( IC_{dep} \cap IC_{N2bm} = \emptyset \). The set of \textit{instantiated components} \( IC_{dep} \) dependent upon the output signal being analysed completely belongs to the set of \textit{instantiated components} to be moved, \( IC_{2bm} \), to another partition.

\textbf{Case 2:} \( (IC_{dep} \cap IC_{2bm} \neq \emptyset) \land (IC_{dep} \cap IC_{N2bm} \neq \emptyset) \). The output signal being analysed has dependents in both the set of \textit{instantiated components} to be moved to another partition \textit{and} in the set of \textit{instantiated components} not to be moved \( IC_{N2bm} \).
Case 3: \((IC_{dep} \cap IC_{2bm} = \emptyset) \land (IC_{dep} \cap IC_{N2bm} \neq \emptyset)\). The dependents of the output signal being analysed belong exclusively to the set of instantiated components not to be moved.

Case 4: \((IC_{dep} \cap IC_{2bm} = \emptyset) \land (IC_{dep} \cap IC_{N2bm} = \emptyset) \rightarrow IC_{dep} = \emptyset\). The output signal being analysed does not have dependents among the set of instantiated components. This means that the dependents are a subset of the entity signals with mode out \(ES_{out}\). Although signals with buffer mode could be considered output entity signals, they are not because the signal would have dependents, and signals having dependents are checked in the previous three cases.

B.2.3 Same-Partition Instantiated-Component Input Signal Dependency Cases

Case 1: \((IC_{dep} \subseteq IC_{2bm}) \land (IC_{dep} \cap IC_{N2bm} = \emptyset)\) All the signals dependent upon \(ic_{driv}\) belong uniquely to the set of instantiated components to be moved \(IC_{2bm}\). Nevertheless, the component driving the input signal being analysed may or may not be included in the set \(IC_{2bm}\). This instantiated component may not even exist. This adds three variants to this case:

**Variant 1:** \((ic_{driv} \subseteq IC_{2bm}) \ni (\exists ic_{driv})\) The instantiated component driving the input signal being analysed belongs to the set of instantiated components to be moved.

Hence it exists.

**Variant 2:** \((ic_{driv} \not\subseteq IC_{dep}) \ni (\exists ic_{driv})\) The instantiated component \(ic_{driv}\) driving the input signal in question exists and belongs to the set of instantiated components not to be moved.

**Variant 3:** \((\nexists ic_{driv})\) In this case there does not exist an instantiated component driving
the input signal being analysed. This implies that the signal is an input signal to the FPGA, i.e., the driving signal is an entity signal with mode IN.

**Case 2:** \((IC_{\text{dep}} \cap IC_{2bn} \neq \emptyset) \land (IC_{\text{dep}} \cap (IC_{N2bn}) \neq \emptyset)\) The ancestor signal of the signal being analysed has dependents in both instantiated components to be moved \(IC_{2bn}\), and in the instantiated components not to be moved. In the actual algorithm it is not checked whether the ancestor signal has dependents in the set of instantiated components to be moved because the input signal being analysed is among the set of instantiated components to be moved, and the driving signal will always have dependents in this set. The variants for this case are whether the driving signal belongs to the instantiated components to be moved, or to the instantiated components not to be moved, or whether it exists.

**Variant 1:** \((ic_{\text{driv}} \in IC_{2bn}) \ni (\exists ic_{\text{driv}})\) The component driving the signal being checked for dependencies exists and it is contained in the set of instantiated components to be moved.

**Variant 2:** \((ic_{\text{driv}} \in IC_{N2bn}) \ni (\exists ic_{\text{driv}})\) The driving component exists and it belongs to the instantiated components *not* to be moved.

**Variant 3:** \(\bar{ic}_{\text{driv}}\) There is no component driving the analysed signal. Its ancestor is an *entity signal* and it is an input signal to the design.

**B.3 Inter-partition Signal Dependencies**

Since the partitioning of the circuit is not automated but selected by the designer, there will very likely be cases in which the designer after doing some partitioning still wants to move
components from a partition $GP_i$ to a partition $GP_j$. It is possible that a component in partition $GP_i$ may have originally been dependent upon a component already placed in partition $GP_j$. The instantiated component in $GP_i$ may be required to be moved back to partition $GP_j$ or to any other partition. In either case the dependencies of this component in other partitions need to be analysed. They are analysed to know whether inter-partition crossing signals and/or internal signals need to be created, eliminated, or modified.

Dependency cases in this subsection occur only as a result of a previous partitioning. Since this is a new type of dependency, new dependency definitions are required.

### B.3.1 Signal Dependency Definitions

1. Inter-partition dependencies can exist only among entity signals $ES$ from different partitions.

2. For an entity signal $es_{P_x} \in ES_{P_x}$ in partition $P_x$ to have dependents in another partition(s), it must belong to the set of entity signals with mode output or buffer $ES_{P_xOB}$ in partition $P_x$.

3. For an entity signal $es_{P_x} \in ES_{P_x}$ in partition $P_x$ to be dependent upon another entity signal $es_{P_w}$ in another partition $P_w$, it must belong to the set of entity signals in partition $P_x$ with mode in $ES_{P_{xin}}$.

4. An $ES_{OB}$ signal can have one or more inter-partition dependents\(^1\).

5. A signal $es_{P_y}$ is said to be dependent upon another signal $es_{P_x}$ if and only if there is an edge $E(es_{P_x}, es_{P_y})$ joining them and they comply with the definitions given above.

\(^1\)This is allowed although the TM-3 topology does not allow inter-partition fanout.
6. A signal $es_{p_z}$ is said to have $N$ inter-partition dependents if and only if $es_{p_z} \in ES_{OB_{p_z}}$
and there exists a set of edges $E(es_{p_z}, (\bigcup_{i=1}^{N} ES_{p_{z1}}) \setminus ES_{p_{zX}})$ having $N$ members. Note
that each entity signal can have only one dependent per partition because the partitioning
algorithm declares only one entity signal for all dependents in one subpartition so there
are not input entity signals in a partition being driven by the same inter-partition ancestor
signal. This reduces the number of inter-partition crossing wires.

B.4 Inter-Partition Instantiated-Component Output Signal Dependency Cases

**Case 1:** $ES_{IN_{dep}} \cap ES_{pto} \neq \emptyset$ and $ES_{IN_{dep}} \notin \bigcup_{i=1}^{p} ES_{i_{IN}} \setminus (ES_{IN_{pfrom}} \cup ES_{IN_{pto}})$. The
signal has dependents only in the partition it is being moved to.

**Case 2:** $ES_{IN_{dep}} \cap ES_{pto} \neq \emptyset$ and $ES_{IN_{dep}} \subset \bigcup_{i=1}^{p} ES_{i_{IN}} \setminus (ES_{IN_{pfrom}} \cup ES_{IN_{pto}})$. The
signal has dependents in the partition it is being moved to and also has dependents in
other partitions.

**Case 3:** $ES_{IN_{dep}} \cap ES_{pto} = \emptyset$ and $ES_{IN_{dep}} \subset \bigcup_{i=1}^{p} ES_{i_{IN}} \setminus (ES_{IN_{pfrom}} \cup ES_{IN_{pto}})$. The
signal has dependents in other partitions but not in the partition where it is being moved
to.

**Case 4:** $ES_{IN_{dep}} \cap ES_{pto} = \emptyset$ and $ES_{IN_{dep}} \not\subset \bigcup_{i=1}^{p} ES_{i_{IN}} \setminus (ES_{IN_{pfrom}} \cup ES_{IN_{pto}})$. The
signal does not have inter-partition dependents $\Rightarrow ES_{IN_{dep}} = \emptyset$. It can be deduced
that this signal was an output signal with no dependents in the original design.
B.5 Inter-Partition Instantiated-Component Input-Signal Dependency Cases

Case 1: \((es_{pdrv} \in ES_{O_{\text{ptra}}}) \ni \exists es_{drv} \ni \) The input entity signal being checked for dependencies is dependent upon another entity signal, and this ancestor signal belongs to the same partition its dependent signal is being moved to. To perform the appropriate changes to a signal, it is necessary to know whether its driving signal has more dependents in other partitions, and whether that ancestor signal existed in the original design. This adds the following variants to this case.

**Variant 1:** \(ES_{IN_{dep}} \setminus \{es_{from}\} \neq \emptyset \text{ and } es_{pdrv} \in ES_{\text{buffer}_{pdrv}}\) The driving signal has more dependents in other partitions, its mode is buffer and was of this mode before any partitioning. This implies that the driving signal has dependents in the partition it resides.

**Variant 2:** \(ES_{IN_{dep}} \setminus \{es_{from}\} \neq \emptyset \text{ and } es_{pdrv} \in ES_{\text{buffer}_{pdrv}}\) The driving signal has more dependents in other partitions, its mode is buffer but its mode was made buffer by a previous partitioning. This implies that the driving signal has more dependents inside its own partition. In the actual partitioning algorithm, signals are checked for internal and inter-partition dependencies.

**Variant 3:** \(ES_{IN_{dep}} \setminus \{es_{from}\} \neq \emptyset \text{ and } es_{pdrv} \in ES_{\text{out}_{pdrv}}\) The driving signal has more dependents in other partitions, its mode is out and was of this mode before any partitioning. Although this case can never exist, it is necessary to list it in order to show all cases. The driving signal was obviously an architecture signal which at the time of moving its dependents to another partition(s) became an entity signal.
with mode \textit{out}.

**Variant 4:** $E_{S_{IN_{dep}}} \setminus \{e_{sp_{from}}\} \neq \emptyset$ and $e_{sp_{drv}} \in E_{S_{out_{pdv}}}$. The driving signal has more dependents in other partitions, its mode is \textit{out} but its mode was made \textit{out} by a previous partitioning.

**Variant 5:** $E_{S_{IN_{dep}}} \setminus \{e_{sp_{from}}\} = \emptyset$ and $e_{sp_{drv}} \in E_{S_{buffer_{pdv}}}$. The driving signal does not have more inter-partition dependents besides the signal being analysed, its mode is \textit{buffer} and was of this mode before any partitioning.

**Variant 6:** $E_{S_{IN_{dep}}} \setminus \{e_{sp_{from}}\} = \emptyset$ and $e_{sp_{drv}} \in E_{S_{buffer_{pdv}}}$. The driving signal does not have more inter-partition dependents apart from the signal being analysed, its mode is \textit{buffer} but was made of this mode after partitioning.

**Variant 7:** $E_{S_{IN_{dep}}} \setminus \{e_{sp_{from}}\} = \emptyset$ and $e_{sp_{drv}} \in E_{S_{out_{pdv}}}$. The driving signal does not have more inter-partition dependents other than the signal being analysed, its mode is \textit{out} and had this mode before any partitioning. This variant is not possible, an entity signal with mode \textit{out} cannot have dependents in the same partitions according to the VHDL grammar.

**Variant 8:** $E_{S_{IN_{dep}}} \setminus \{e_{sp_{from}}\} = \emptyset$ and $e_{sp_{drv}} \in E_{S_{out_{pdv}}}$. The driving signal does not have more inter-partition dependents other than the signal being analysed, its mode is \textit{out} and obviously was acquired after partitioning.

**Case 2:** $e_{sdrv_{ob}} \in \left( \bigcup_{i=1}^{p} E_{S_{i_{ob}}} \setminus E_{S_{ob_{pto}}} \setminus E_{S_{ob_{pfrom}}} \right)$ The signal being analysed does have an ancestor signal driving it. This driving signal is located in a partition other than the one the signal being moved resides ($p_{from}$). Although in this case the signal being analysed could be checked for all the variants described in the previous case, it is not necessary.
In this case it does not matter whether the signal is of buffer or output mode because its mode and entity attributes are not going to be affected. What is really important, is whether the driving signal already has dependents in the partition pto. This defines whether an input entity signal is required in the partition pto or not.

**Variant 1:** $ES_{I,N_{dep}} \cap ES_{pto} = \emptyset$ There are no dependents of the driving signal in the partition the signal being analysed is being moved to.

**Variant 2:** $ES_{I,N_{dep}} \cap ES_{pto} \neq \emptyset$ The driving signal has a dependent signal in the same partition the entity input signal is being moved to.

**Case 3:** $\exists es_{drv_{oo}}$ The input entity signal being moved does not depend on any other signal i.e. it does not have an ancestor. This implies that it was an input signal to the original design. However, the partitioning algorithm, in order to reduce the number of crossing wires between partitions, may replicate an original input entity signal in another partition. This creates two variants for this case.

**Variant 1:** $(es_{from} \in ES_{from}) \land (es_{from}' \in ES_{pto})$ An input entity signal with the same name and characteristics already exists in partition pto the only difference is their internal dependencies.

**Variant 2:** $es_{from} \in ES_{from} \not\ni \exists es_{from}'$ An input entity signal with the same name and characteristics does not exists in partition pto.
Appendix C

VHDL Source Codes And Schematics

This appendix presents the VHDL source codes for designs in chapter 7 before and after these designs have been partitioned.

C.1  First VHDL Design

C.1.1  VHDL Source Code
library IEEE;
use IEEE.std_logic_1164.all;

entity test is
  port (K11 : in std_logic;
        K12 : in std_logic;
        K13 : in std_logic;
        K14 : in std_logic;
        K15 : in std_logic;
        K16 : in std_logic;
        K07 : buffer std_logic;
        E08 : buffer std_logic;
        E09 : buffer std_logic;
        E10 : buffer std_logic;
        E11 : out std_logic;
        E12 : out std_logic;
        E13 : out std_logic;
        E14 : out std_logic;
        E15 : buffer std_logic;
        E16 : buffer std_logic;
        E17 : out std_logic;
        E18 : out std_logic;
        E19 : buffer std_logic;
  end test);

architecture structural of test is
  component tres45p
    port (in1 : in std_logic;
          out1 : buffer std_logic;
          out2 : buffer std_logic;
    end component;
  component siete8910
    port (in1, in2 : in std_logic;
          out1 : out std_logic;
    end component;
  component cumpuno
    port (in1, in2, in3, in4, in5, in6, in7, in8, in9, in10 : in std_logic;
          out1, out2, out3, out4, out5, out6, out7 : buffer std_logic;
    end component;
  component cumpdos
    port (in1, in2, in3, in4, in5 : in std_logic;
          out1 : out std_logic;
    end component;
  component uno
    port (in1 <= K11;
          in2 <= E11;
          in3 <= A1;
          in4 <= A2;
          in5 <= A3;
          in6 <= A4;
          in7 <= A5;
          in8 <= A6;
          in9 <= A7;
          in10 <= K12;
          out1 <= E11;
          out2 <= E07;
          out3 <= E08;
    end component;
  component dos
    port (in1 <= K11;
          in2 <= E11;
          in3 <= A1;
          in4 <= A2;
          in5 <= A3;
          in6 <= A4;
          in7 <= A5;
          in8 <= A6;
          in9 <= A7;
          in10 <= K12;
          out1 <= E11;
          out2 <= E07;
          out3 <= E08;
    end component;
  component tres
    port (in1 <= E11;
          in2 <= E07;
          in3 <= K12;
          out1 <= E11;
          out2 <= E07;
          out3 <= E08;
    end component;
  component cuatro
    port (in1 <= E11;
          in1 <= E07;
          in2 <= E08;
    end component;
  component cinco
    port (in1 <= K11;
          out1 <= E11;
          out2 <= A1;
          out3 <= A2;
    end component;
  component seis
    port (in1 <= E11;
          out1 <= A1;
          out2 <= A2;
    end component;
  component siete
    port (in1 <= K12;
          out1 <= E11;
          out2 <= A1;
          out3 <= A2;
    end component;
  component ochos
    port (in1 <= E11;
          in2 <= E07;
          out1 <= E08;
    end component;
  component nueve
    port (in1 <= E11;
          in2 <= E07;
          in3 <= K12;
          out1 <= E08;
    end component;
  component diez
    port (in1 <= E11;
          in2 <= E07;
          in3 <= K12;
          out1 <= E08;
    end component;
  component once
    port (in1 <= E11;
          in2 <= E07;
          in3 <= K12;
          out1 <= E08;
    end component;
begin
  tres : tres45p port map;
  cuatro : tres45p port map;
end structural;
C.1.2 Schematic Diagram
C.1.3 VHDL Source Codes After Partitioning

C.1.3.1 Partition One
library IEEE;
use IEEE.std_logic_1164.all;
entity FPGAtest is
  PORT ( 
       ED19 : IN STD_LOGIC;
       ED16 : IN STD_LOGIC;
       AT : IN STD_LOGIC;
       AF : IN STD_LOGIC;
       DF : IN STD_LOGIC;
       AD : IN STD_LOGIC;
       E08 : IN STD_LOGIC;
       E05 : BUFFER STD_LOGIC;
       E10 : BUFFER STD_LOGIC;
       E12 : OUT STD_LOGIC;
       E13 : OUT STD_LOGIC;
       E18 : OUT STD_LOGIC;
       ED18 : IN STD_LOGIC;
     );
end FPGAtest;
architecture structural of FPGAtest is
component ties456
  port (in1 : in std_logic;
        out1 : buffer std_logic;
        out2 : buffer std_logic);
end component;
component siete8910
  port (in1, in2 : in std_logic;
        out1 : out std_logic);
end component;
component compdos
  port (in1, in2, in3, in4, in5 : in std_logic;
        out1 : out std_logic);
end component;
begin
CUATRO : TRES456 PORT MAP ( 
      IN1 => E12,
      OUT1 => E09,
      OUT2 => E010);
SEIS : TRES456 PORT MAP ( 
      IN1 => E16,
      OUT1 => A3,
      OUT2 => A4);
OCHO : SIETE8910 PORT MAP ( 
      IN1 => A6,
      IN2 => A7,
      OUT1 => E012);
Diez : SIETE8910 PORT MAP ( 
      IN1 => E016,
      IN2 => E019,
      OUT1 => E018);
end structural;
DOS : COMPDOS PORT MAP ( 
      IN1 => E14,
      IN2 => A5,
      IN3 => A4,
      IN4 => E08,
      IN5 => E010,
      OUT1 => E013);
C.1.3.2 Partition Two
library IEEE;
use IEEE.std_logic_1164.all;

entity FPGA2test is
  port (E11 : IN STD_LOGIC;
        E15 : IN STD_LOGIC;
        E013 : OUT STD_LOGIC;
        E017 : OUT STD_LOGIC;
        E019 : BUFFER STD_LOGIC;
        E016 : BUFFER STD_LOGIC;
        E015 : BUFFER STD_LOGIC;
        E014 : OUT STD_LOGIC;
        A7 : OUT STD_LOGIC;
        A6 : BUFFER STD_LOGIC;
        E016 : IN STD_LOGIC;
        E09 : IN STD_LOGIC;
        E07 : BUFFER STD_LOGIC;
        E08 : BUFFER STD_LOGIC;
        A4 : IN STD_LOGIC;
        A3 : IN STD_LOGIC;
        A2 : BUFFER STD_LOGIC;
        E14 : IN STD_LOGIC;
        E13 : IN STD_LOGIC);
end FPGA2test;

architecture structural of FPGA2test is
  begin
    TRE5 : TRES456 PORT MAP (IN1 => E11, OUT1 => E07, OUT2 => E08);
    SIETE : SIETE910 PORT MAP (IN1 => A5, IN2 => E15, OUT1 => E011);
    NUEVE : SIETE910 PORT MAP (IN1 => E015, IN2 => E016, OUT1 => E017);
    UNO : COMPUNO PORT MAP (IN1 => E13, IN2 => E14, IN3 => A1, IN4 => A2, IN5 => A3, IN6 => A4, IN7 => E07, IN8 => E08, IN9 => E09, OUT1 => E010, OUT2 => ARCH_A6, OUT3 => A7, OUT4 => E014, OUT5 => E015, OUT6 => E016, OUT7 => E019);
    A0 <= ARCH_A6,
    end structural;
end architecture structural;

begin
  TRE5 : TRES456 PORT MAP (IN1 => E11, OUT1 => E07, OUT2 => E08);
  SIETE : SIETE910 PORT MAP (IN1 => A5, IN2 => E15, OUT1 => E011);
  NUEVE : SIETE910 PORT MAP (IN1 => E015, IN2 => E016, OUT1 => E017);
  UNO : COMPUNO PORT MAP (IN1 => E13, IN2 => E14, IN3 => A1, IN4 => A2, IN5 => A3, IN6 => A4, IN7 => E07, IN8 => E08, IN9 => E09, OUT1 => E010, OUT2 => ARCH_A6, OUT3 => A7, OUT4 => E014, OUT5 => E015, OUT6 => E016, OUT7 => E019);

  A0 <= ARCH_A6;
end
C.1.4 **Schematic Diagrams After Partitioning**

C.1.4.1 **Partition One**

![Schematic Diagram]

- **EI4**
- **EI6**
- **EI2**
- **EO8**
- **A6**
- **A7**
- **EO16**
- **EO19**
- **A2**

- **seis**
- **cuatro**
- **ocho**
- **diez**

- **A3**
- **A4**
- **EO13**
- **EO9**
- **EO10**
- **EO12**
- **EO18**
C.1.4.2 Partition Two
C.2   Alarm Clock Implementation

C.2.1   Top-Level VHDL File
entity TOP =
begin
architecture SCHEMATIC of TOP is
begin
  RING : RINGER_STATE_MACHINE
  PORT MAP (CLK => CL, COMPARE_IN => H1, TOGGLE_ON => TOGGLE_SWITCH,
             RING => RING_OUT);
  CONVERT : CONVERTER_CRT
  PORT MAP (HRS_IN => H1, MINS_IN => M1, DISPI => DISPI_OUT,
            OUTPUT_MIX => MUX);
  ALARM : ALARM_STATE_MACHINE
  PORT MAP (CLK => CL, HRS => H1, MINS => M1, RESET => RESET,
            TIME_HRS => TIME_HRS_OUT, TIME_MINS => TIME_MINS_OUT);
end architecture SCHEMATIC;
end TOP;

component RINGER_STATE_MACHINE
  Port (CLK : IN std_logic;
        COMPARE_IN : IN std_logic;
        TOGGLE_ON : IN std_logic;
        RING : OUT std_logic);
end component;

component CONVERTER_CRT
  Port (HRS_IN : IN std_logic_vector(5 downto 0);
        MINS_IN : IN std_logic_vector(5 downto 0);
        DISPI : OUT std_logic_vector(13 downto 0);
        OUTPUT_MIX : OUT std_logic_vector(13 downto 0);
        OUTPUT_MIX : OUT std_logic_vector(13 downto 0));
end component;

component ALARM_STATE_MACHINE
  Port (CLK : IN std_logic;
        ALARM_HRS : IN std_logic;
        ALARM_MINS : IN std_logic;
        TIME_HRS : IN std_logic;
        TIME_MINS : IN std_logic;
        COMPARE_OUT : OUT std_logic);
end component;

configured.cfg : SCHEMATIC OF TOP is
  for RING : RINGER_STATE_MACHINE
    use configuration WORK.CFG_RINGER_STATE_MACHINE_BEHAVIORAL;
  end for;
  for CONVERT : CONVERTER_CRT
    use configuration WORK.CFG_CONVERTER_CRT_BEHAVIORAL;
  end for;

configuration cfg_top_schematic of top is
  for RING : RINGER_STATE_MACHINE
    use configuration WORK.CFG_RINGER_STATE_MACHINE_BEHAVIORAL;
  end for;
  for CONVERT : CONVERTER_CRT
    use configuration WORK.CFG_CONVERTER_CRT_BEHAVIORAL;
  end for;
end configuration;

begin
end

C.2.2 Top-Level Schematic Diagram
C.2.3 Flattened Top-Level VHDL File
use IEEE.std_logic_arith.all;

library IEEE;
use IEEE.std_logic_arith.all;

entity flattopsplit is
  PORT ( 
    SPEAKER_OUT : OUT std_logic;
    DISP_13 : OUT std_logic;
    DISP_12 : OUT std_logic;
    DISP_11 : OUT std_logic;
    DISP_10 : OUT std_logic;
    DISP_9 : OUT std_logic;
    DISP_8 : OUT std_logic;
    DISP_7 : OUT std_logic;
    DISP_6 : OUT std_logic;
    DISP_5 : OUT std_logic;
    DISP_4 : OUT std_logic;
    DISP_3 : OUT std_logic;
    DISP_2 : OUT std_logic;
    DISP_1 : OUT std_logic;
    DISP_0 : OUT std_logic;
    DISP_13 : OUT std_logic;
    DISP_12 : OUT std_logic;
    DISP_11 : OUT std_logic;
    DISP_10 : OUT std_logic;
    DISP_9 : OUT std_logic;
    DISP_8 : OUT std_logic;
    DISP_7 : OUT std_logic;
    DISP_6 : OUT std_logic;
    DISP_5 : OUT std_logic;
    DISP_4 : OUT std_logic;
    DISP_3 : OUT std_logic;
    DISP_2 : OUT std_logic;
    DISP_1 : OUT std_logic;
    DISP_0 : OUT std_logic;
    CLK : IN std_logic;
    ALARM : IN std_logic
  );
end flattopsplit;

architecture behavior of flattopsplit is
  COMPONENT ZERO
    PORT ( 
      ZERO : OUT std_logic;
    );
  END COMPONENT;

  COMPONENT HOURS_FILTER
    PORT ( 
      TENS_OUT : OUT std_logic_vector(1 DOWNTO 0);
      TENS_IN : IN std_logic_vector(6 DOWNTO 0);
    );
  END COMPONENT;

  COMPONENT CONVERTER
    PORT ( 
      TENS_DIGIT : OUT std_logic_vector(6 DOWNTO 0);
      ODES_DIGIT : OUT std_logic_vector(6 DOWNTO 0);
      DATA_IN : IN std_logic_vector(5 DOWNTO 0);
    );
  END COMPONENT;

  COMPONENT TIME_COUNTER
    PORT ( 
      MINS_OUT : BUFFER INTEGER RANGE 0 TO 59;
      HRS_OUT : BUFFER INTEGER RANGE 1 TO 12;
      AM_PM_OUT : BUFFER std_logic;
      RESET : IN std_logic;
      INC_SECS : IN std_logic;
      INC_MINS : IN std_logic;
      INC_AM : IN std_logic;
      INC_PM : IN std_logic;
      CLK : IN std_logic;
    );
  END COMPONENT;

  COMPONENT TIME_STATE_MACHINE
    PORT ( 
      INC_SECS : OUT std_logic;
      INC_MINS : OUT std_logic;
      INC_AM : OUT std_logic;
      INC_PM : OUT std_logic;
      TIME_BUTTON : IN std_logic;
      MINS_BUTTON : IN std_logic;
      HRS_BUTTON : IN std_logic;
      CLR : IN std_logic;
    );
  END COMPONENT;

  COMPONENT ALARM_COUNTER
    PORT ( 
      MINS_OUT : BUFFER INTEGER RANGE 0 TO 59;
      HRS_OUT : BUFFER INTEGER RANGE 0 TO 12;
      AM_PM_OUT : BUFFER std_logic;
      RESET : IN std_logic;
      INC_MINS : IN std_logic;
      INC_HRS : IN std_logic;
      CLR : IN std_logic;
    );
  END COMPONENT;

  COMPONENT ALARM_STATE_MACHINE
    PORT ( 
      INC_MINS : OUT std_logic;
      INC_HRS : OUT std_logic;
      RESET : IN std_logic;
      MINS_BUTTON : IN std_logic;
      HRS_BUTTON : IN std_logic;
      CLR : IN std_logic;
    );
  END COMPONENT;

  COMPONENT alarm_block_comp
    PORT ( 
      MINS_OUT : BUFFER INTEGER RANGE 0 TO 59;
      HRS_OUT : BUFFER INTEGER RANGE 0 TO 12;
      AM_PM_OUT : BUFFER std_logic;
      CLR : IN std_logic;
    );
  END COMPONENT;

  COMPONENT comparator
    PORT ( 
      COMPARE_OUT : OUT std_logic;
    );
  END COMPONENT;
end behavior;
COMPONENT RING_STATE_MACHINE
PORT (RS : OUT std_logic;
    TIME: INTEGER RANGE 0 TO 59;
    COMPARE: IN INTEGER RANGE 1 TO 12;
    INC_MIN: => INC_MIN,
    INC_HRS: => INC_HRS,
    INC: => INC;
    CLK: IN std_logic);
END COMPONENT;

COMPONENT RINCER_STATE_MACHINE
PORT (RING: OUT std_logic;
    MSG: IN INTEGER RANGE 0 TO 59;
    ALARM: IN INTEGER RANGE 0 TO 12;
    ALARM_MIN: => ALARM_MIN,
    ALARM_HRS: => ALARM_HRS,
    ALARM: => ALARM;
    CLK: IN std_logic);
END COMPONENT;

BEGIN
A_BLOCK_ALARM_STATE_MACHINE: alarm_state_machine PORT MAP
    (INC_MIN: => INC_MIN,
    INC_HRS: => INC_HRS,
    INC: => INC;
    MSG: OUT => MSG,
    TIME: IN INTEGER RANGE 0 TO 59;
    COMPARE: IN INTEGER RANGE 1 TO 12;
    INC_MIN: => INC_MIN,
    INC_HRS: => INC_HRS,
    INC: => INC;
    CLK: IN std_logic;
    MSG_OUT: => MSG_OUT;
    TIME: IN INTEGER RANGE 0 TO 59;
    COMPARE: IN INTEGER RANGE 1 TO 12;
    INC_MIN: => INC_MIN,
    INC_HRS: => INC_HRS,
    INC: => INC;
    CLK: IN std_logic;
    MSG_OUT: => MSG_OUT;
    TIME: IN INTEGER RANGE 0 TO 59;
    COMPARE: IN INTEGER RANGE 1 TO 12;
    INC_MIN: => INC_MIN,
    INC_HRS: => INC_HRS,
    INC: => INC;
    CLK: IN std_logic;
    MSG_OUT: => MSG_OUT;
    TIME: IN INTEGER RANGE 0 TO 59;
    COMPARE: IN INTEGER RANGE 1 TO 12;
    INC_MIN: => INC_MIN,
    INC_HRS: => INC_HRS,
    INC: => INC;
    CLK: IN std_logic;
    MSG_OUT: => MSG_OUT;
    TIME: IN INTEGER RANGE 0 TO 59;
    COMPARE: IN INTEGER RANGE 1 TO 12;
    INC_MIN: => INC_MIN,
    INC_HRS: => INC_HRS,
    INC: => INC;
    CLK: IN std_logic;
    MSG_OUT: => MSG_OUT;
    TIME: IN INTEGER RANGE 0 TO 59;
    COMPARE: IN INTEGER RANGE 1 TO 12;
    INC_MIN: => INC_MIN,
    INC_HRS: => INC_HRS,
    INC: => INC;
    CLK: IN std_logic;
    MSG_OUT: => MSG_OUT;
    TIME: IN INTEGER RANGE 0 TO 59;
    COMPARE: IN INTEGER RANGE 1 TO 12;
    INC_MIN: => INC_MIN,
    INC_HRS: => INC_HRS,
    INC: => INC;
    CLK: IN std_logic;
    MSG_OUT: => MSG_OUT;
    TIME: IN INTEGER RANGE 0 TO 59;
    COMPARE: IN INTEGER RANGE 1 TO 12;
    INC_MIN: => INC_MIN,
    INC_HRS: => INC_HRS,
    INC: => INC;
    CLK: IN std_logic;
    MSG_OUT: => MSG_OUT;
    TIME: IN INTEGER RANGE 0 TO 59;
    COMPARE: IN INTEGER RANGE 1 TO 12;
    INC_MIN: => INC_MIN,
    INC_HRS: => INC_HRS,
    INC: => INC;
    CLK: IN std_logic;
    MSG_OUT: => MSG_OUT;
    TIME: IN INTEGER RANGE 0 TO 59;
    COMPARE: IN INTEGER RANGE 1 TO 12;
    INC_MIN: => INC_MIN,
    INC_HRS: => INC_HRS,
    INC: => INC;
    CLK: IN std_logic;
    MSG_OUT: => MSG_OUT;
    TIME: IN INTEGER RANGE 0 TO 59;
    COMPARE: IN INTEGER RANGE 1 TO 12;
    INC_MIN: => INC_MIN,
    INC_HRS: => INC_HRS,
    INC: => INC;
    CLK: IN std_logic;
    MSG_OUT: => MSG_OUT;
    TIME: IN INTEGER RANGE 0 TO 59;
    COMPARE: IN INTEGER RANGE 1 TO 12;
    INC_MIN: => INC_MIN,
    INC_HRS: => INC_HRS,
    INC: => INC;
    CLK: IN std_logic;
    MSG_OUT: => MSG_OUT;
    TIME: IN INTEGER RANGE 0 TO 59;
    COMPARE: IN INTEGER RANGE 1 TO 12;
    INC_MIN: => INC_MIN,
COMPARE_OUT => H.1,
TIME_MINS => TH,
TIME_HRS => TH,
TIME_AM_PM => TP,
ALARM_MINS => AM,
ALARM_HRS => AH,
ALARM_AM_PM => AP;

CONVERSION PORT MAP

for behavior for CONVERT_HRS_CONVERTER, CONVERT_MINS_CONVERTER, CONVERT_HRS_CONVERTER, CONVERT_MINS_CONVERTER use configuration WORK.CFG_CONVERTER_BEHAVIORAL;
end for;

for behavior for CONVERT: COMPARE use configuration WORK.CFG_COMPARATOR_BEHAVIORAL;
end for;

for behavior for OUTPUT_MUX: MUX use configuration WORK.CFG_MUX_BEHAVIORAL;
end for;

for behavior for T_BW: TIME_BLOCK use configuration WORK.CFG_TIME_BLOCK_SCHEMATIC;
end for;

for behavior for T_BW: TIME_COUNTER use configuration WORK.CFG_TIME_COUNTER_BEHAVIORAL;
end for;

for behavior for T_BW: TIME_STATE_MACHINE use configuration WORK.CFG_TIME_STATE_MACHINE_BEHAVIORAL;
end for;

for behavior for A_BLOCK: ALARM use configuration WORK.CFG_ALARM_STATE_MACHINE_BEHAVIORAL;
end for;

for behavior for A_BLOCK: ALARM use configuration WORK.CFG_ALARM_STATE_MACHINE_BEHAVIORAL;
end for;
C.2.4 Schematic Diagram for Flattened Top-Level File
C.2.5 VHDL Source Codes After Partitioning

C.2.5.1 Partition One
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY FPGAtop IS
PORT ( 
  CLK : IN STD_LOGIC;
  RESET : IN STD_LOGIC;
  MINS : IN STD_LOGIC;
  HRS : IN STD_LOGIC;
  TM : BUFFER INTEGER RANGE 0 TO 59;
  TH : BUFFER INTEGER RANGE 1 TO 12;
  TP : BUFFER STD_LOGIC;
  SET_TIME : IN STD_LOGIC;
  )
END COMPONENT;

ARCHITECTURE ARCHFPGAtop OF FPGAtop IS

COMPONENT RINGER_STATE_MACHINE
PORT ( 
  CLK : IN STD_LOGIC;
  COMPARE_IN : IN STD_LOGIC;
  TOGGLE_ON : IN STD_LOGIC;
  TIME_COUNTER : OUT STD_LOGIC;
  )
END COMPONENT;

COMPONENT COMPARATOR
PORT ( 
  ALARM_AM_PM : IN STD_LOGIC;
  ALARM_HRS : IN INTEGER RANGE 1 TO 12;
  ALARM_MINS : IN INTEGER RANGE 0 TO 59;
  TIME_AM_PM : IN STD_LOGIC;
  TIME_HRS : IN INTEGER RANGE 1 TO 12;
  TIME_MINS : IN INTEGER RANGE 0 TO 59;
  COMPARE_OUT : OUT STD_LOGIC;
  )
END COMPONENT;

COMPONENT MUX
PORT ( 
  ALARM : IN STD_LOGIC;
  ALARM_AM_PM : IN STD_LOGIC;
  ALARM_HRS : IN INTEGER RANGE 1 TO 12;
  ALARM_MINS : IN INTEGER RANGE 0 TO 59;
  TIME_AM_PM : IN STD_LOGIC;
  TIME_HRS : IN INTEGER RANGE 1 TO 12;
  TIME_MINS : IN INTEGER RANGE 0 TO 59;
  AM_PM : OUT STD_LOGIC;
  HRS_BIN : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
  MINS_BIN : OUT STD_LOGIC_VECTOR(5 DOWNTO 0);
  )
END COMPONENT;

COMPONENT ALARM_BLOCK_COMP
PORT ( 
  AM_PM_OUT_DUMMY : IN STD_LOGIC;
  MINS_OUT_DUMMY : IN INTEGER RANGE 0 TO 59;
  HRS_OUT_DUMMY : IN INTEGER RANGE 1 TO 12;
  AM_PM_OUT : BUFFER STD_LOGIC;
  HRS_OUT : BUFFER INTEGER RANGE 1 TO 12;
  MINS_OUT : BUFFER INTEGER RANGE 0 TO 59;
  )
END COMPONENT;

COMPONENT ALARM_STATE_MACHINE
PORT ( 
  ALARM_BUTTON : IN STD_LOGIC;
  CLK : IN STD_LOGIC;
  HRS_BUTTON : IN STD_LOGIC;
  MIN_BUTTON : IN STD_LOGIC;
  RESET : IN STD_LOGIC;
  INC_HRS : OUT STD_LOGIC;
  INC_MINS : OUT STD_LOGIC;
  )
END COMPONENT;

COMPONENT ALARM_COUNTER
PORT ( 
  CLK : IN STD_LOGIC;
  INC_HRS : IN STD_LOGIC;
  INC_MINS : IN STD_LOGIC;
  RESET : IN STD_LOGIC;
  AM_PM_OUT : BUFFER STD_LOGIC;
  HRS_OUT : BUFFER INTEGER RANGE 1 TO 12;
  MINS_OUT : BUFFER INTEGER RANGE 0 TO 59;
  )
END COMPONENT;

COMPONENT TIME_STATE_MACHINE
PORT ( 
  CLK : IN STD_LOGIC;
  INC_HRS : IN STD_LOGIC;
  INC_MINS : IN STD_LOGIC;
  INC_SECS : IN STD_LOGIC;
  )
END COMPONENT;

COMPONENT TIME_COUNTER
PORT ( 
  CLK : IN STD_LOGIC;
  INC_HRS : IN STD_LOGIC;
  INC_MINS : IN STD_LOGIC;
  INC_SECS : IN STD_LOGIC;
  AM_PM_OUT : BUFFER STD_LOGIC;
  HRS_OUT : BUFFER INTEGER RANGE 1 TO 12;
  MINS_OUT : BUFFER INTEGER RANGE 0 TO 59;
  )
END COMPONENT;

COMPONENT SIGNAL
PORT ( 
  DATA_IN : IN STD_LOGIC_VECTOR; 5 DOWNTO 0); ONE'S_DIGIT : OUT STD_LOGIC_VECTOR(6 DOWNTO 0); TEN'S_DIGIT : OUT STD_LOGIC_VECTOR(6 DOWNTO 0); TIME_COUNTER : OUT STD_LOGIC_VECTOR(6 DOWNTO 0); HOURS_FILTER : OUT STD_LOGIC_VECTOR(6 DOWNTO 0); BINARY_CODE : OUT STD_LOGIC_VECTOR(6 DOWNTO 0); )
END COMPONENT;

COMPONENT HOURS_FILTER
PORT ( 
  TEN'S_IN : IN STD_LOGIC_VECTOR(6 DOWNTO 0); TEN'S_OUT : OUT STD_LOGIC_VECTOR(6 DOWNTO 0); )
END COMPONENT;

COMPONENT ZERO
PORT ( 
  ZERO : OUT STD_LOGIC;
  )
END COMPONENT;

BEGIN
T_BLOCK_time_block_comp : alarm_block_comp PORT MAP ( 
  )
END
```
AM_PM_OUT_DUMMY => T_BLOCK_AM_PM_OUT_DUMMY,
MINS_OUT_DUMMY  => T_BLOCK_MINS_OUT_DUMMY,
HRS_OUT_DUMMY   => T_BLOCK_HRS_OUT_DUMMY,
AM_PM_OUT       => TP,
HRS_OUT         => TM,
MINS_OUT        => TM);

T_BLOCK_TCOUNT : TIME_COUNTER PORT MAP 
  CLK  => CLK,
  INC_HRS  => T_BLOCK_H_1,
  INC_MINS => T_BLOCK_M_2,
  INC_SEC  => T_BLOCK_S_3,
  RESET   => RESET,
  AM_PM_OUT => T_BLOCK_AM_PM_OUT_DUMMY,
  HRS_OUT  => T_BLOCK_HRS_OUT_DUMMY,
  MINS_OUT => T_BLOCK_MINS_OUT_DUMMY;

T_BLOCK_TSM : TIME_STATE_MACHINE PORT MAP 
  CLK  => CLK,
  HRS_BUTTON => HRS,
  MINS_BUTTON => MINS,
  RESET   => RESET,
  TIME_BUTTON => SET_TIME,
  INC_HRS  => T_BLOCK_H_1,
  INC_MINS => T_BLOCK_M_2,
  INC_SEC  => T_BLOCK_S_3;

END ARCHITECTURE;
```
C.2.5.2  Partition Two
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY FPGA2top IS
PORT (  
  RESET : IN std_logic;
  HRS : IN std_logic;
  CLS : IN std_logic;
  M.S : OUT std_logic;
  M.V : OUT std_logic;
  M.R : OUT std_logic;
  M.L : OUT std_logic;
  M.H : OUT std_logic;
  M.Z : OUT std_logic;
  AM_PM : OUT std_logic;
  TM : IN integer range 0 TO 59;
  TH : IN integer range 1 TO 12;
  TP : IN std_logic;
  AM : BUFFER integer range 0 TO 59;
  AN : BUFFER integer range 1 TO 12;
  AP : BUFFER std_logic;
  ALARM : IN std_logic
);  
END FPGA2top;
ARCHITECTURE ARCHFPGA2top OF FPGA2top IS
COMPONENT RINGER_STATE_MACHINE
PORT (  
  CLK : IN std_logic;
  COMPARE_IN : IN std_logic;
  TOGGLE_ON : IN std_logic;
  RING : OUT std_logic;
);  
END COMPONENT;
COMPONENT COMPARATOR
PORT (  
  ALARM_AM_PM : IN std_logic;
  ALARM_HRS : IN integer range 1 TO 12;
  ALARM_MINS : IN integer range 0 TO 59;
  TIME_AM_PM : IN std_logic;
  TIME_HRS : IN integer range 1 TO 12;
  TIME_MINS : IN integer range 0 TO 59;
  COMPARE_OUT : OUT std_logic;
);  
END COMPONENT;
COMPONENT MIX
PORT (  
  ALARM : IN std_logic;
  ALARM_AM_PM : IN std_logic;
  ALARM_HRS : IN integer range 1 TO 12;
  ALARM_MINS : IN integer range 0 TO 59;
  TIME_AM_PM : IN std_logic;
  TIME_HRS : IN integer range 1 TO 12;
  TIME_MINS : IN integer range 0 TO 59;
  AM_PM : OUT std_logic;
  HRS_BIN : OUT std_logic_vector(3 DISCONNECT 0);
  MINS_BIN : OUT std_logic_vector(5 DISCONNECT 0);
);  
END COMPONENT;
COMPONENT alarm_block_comp
PORT (  
  AM_PM_OUT_DUMMY : IN std_logic;
  MINS_OUT_DUMMY : IN integer range 0 TO 59;
  HRS_OUT_DUMMY : IN integer range 1 TO 12;
  AM_PM_OUT : BUFFER std_logic;
  HRS_OUT : BUFFER integer range 1 TO 12;
  MINS_OUT : BUFFER integer range 0 TO 59;
);  
END COMPONENT;
COMPONENT ALARM_STATE_MACHINE
PORT (  
  ALARM_BUTTON : IN std_logic;
  CLK : IN std_logic;
  HRS_BUTTON : IN std_logic;
  MINS_BUTTON : IN std_logic;
  RESET : IN std_logic;
  INC_HRS : OUT std_logic;
  INC_MINS : OUT std_logic;
);  
END COMPONENT;
COMPONENT ALARM_COUNTER
PORT (  
  CLK : IN std_logic;
  INC_HRS : IN std_logic;
  INC_MINS : IN std_logic;
  INC_SECS : IN std_logic;
  AM_PM_OUT : BUFFER std_logic;
  HRS_OUT : BUFFER integer range 1 TO 12;
  MINS_OUT : BUFFER integer range 0 TO 59;
);  
END COMPONENT;
COMPONENT TIME_STATE_MACHINE
PORT (  
  CLK : IN std_logic;
  HRS_BUTTON : IN std_logic;
  MINS_BUTTON : IN std_logic;
  RESET : IN std_logic;
  TIME_BUTTON : IN std_logic;
  INC_HRS : OUT std_logic;
  INC_MINS : OUT std_logic;
  INC_SECS : OUT std_logic;
);  
END COMPONENT;
COMPONENT TIME_COUNTER
PORT (  
  CLK : IN std_logic;
  INC_HRS : IN std_logic;
  INC_MINS : IN std_logic;
  INC_SECS : IN std_logic;
  AM_PM_OUT : BUFFER std_logic;
  HRS_OUT : BUFFER integer range 1 TO 12;
  MINS_OUT : BUFFER integer range 0 TO 59;
);  
END COMPONENT;
COMPONENT CONVERTOR
PORT (  
  DATA_IN : IN std_logic_vector(5 DISCONNECT 0);
  ONES_DIGIT : OUT std_logic_vector(6 DISCONNECT 0);
  TENS_DIGIT : OUT std_logic_vector(5 DISCONNECT 0);
);  
END COMPONENT;
COMPONENT HOURS_FILTER
PORT (  
  TENS_IN : IN std_logic_vector(6 DISCONNECT 0);
  TENS_OUT : OUT std_logic_vector(6 DISCONNECT 0);
);  
END COMPONENT;
COMPONENT ZERO
PORT (  
);  
END COMPONENT;
ZERO : OUT std_logic;

END COMPONENT;

SIGNAL A_BLOCK_H_1 : std_logic;
SIGNAL A_BLOCK_H_2 : std_logic;
SIGNAL A_BLOCK_MINS_OUT_DUMMY : INTEGER RANGE 0 TO 59;
SIGNAL A_BLOCK_AM_PM_OUT_DUMMY : std_logic;

BEGIN
A_BLOCK_ACOUNT : ALARM_COUNTER PORT MAP (
  CLK => CLK,
  INC_HRS => A_BLOCK_H_2,
  INC_MINS => A_BLOCK_H_1,
  RESET => RESET,
  AM_PM_OUT => A_BLOCK_AM_PM_OUT_DUMMY,
  HRS_OUT => A_BLOCK_MINS_OUT_DUMMY,
  MINS_OUT => A_BLOCK_MINS_OUT_DUMMY);

A_BLOCK_AM : ALARM_STATE_MACHINE PORT MAP (
  ALARM_BUTTON => ALARM,
  CLK => CLK,
  HRS_BUTTON => HRS,
  MINS_BUTTON => MINS,
  RESET => RESET,
  INC_HRS => A_BLOCK_H_2,
  INC_MINS => A_BLOCK_H_1);

A_BLOCK_alarm_block_comp : alarm_block_comp PORT MAP (
  AM_PM_OUT_DUMMY => A_BLOCK_AM_PM_OUT_DUMMY,
  MINS_OUT_DUMMY => A_BLOCK_MINS_OUT_DUMMY,
  HRS_OUT_DUMMY => A_BLOCK_HRS_OUT_DUMMY,
  AM_PM_OUT => AP,
  HRS_OUT => AH,
  MINS_OUT => AM);

OUTPUT_MUX : MUX PORT MAP (
  ALARM => ALARM,
  ALARM_AM_PM => AP,
  ALARM_HRS => AH,
  ALARM_MINS => AM,
  TIME_AM_PM => TP,
  TIME_HRS => TH,
  TIME_MINS => TM,
  AM_PM => AM_PM,
  HRS_BIN(0) => H_0,
  HRS_BIN(1) => H_1,
  HRS_BIN(2) => H_2,
  HRS_BIN(3) => H_3,
  MINS_BIN(0) => M_0,
  MINS_BIN(1) => M_1,
  MINS_BIN(2) => M_2,
  MINS_BIN(3) => M_3,
  MINS_BIN(4) => M_4,
  MINS_BIN(5) => M_5);

END ARCHITECTURE;
C.2.5.3 Partition Three
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY FPGA3Top IS
PORT :
  DISPLAY_6 : OUT std_logic;
  DISPLAY_5 : OUT std_logic;
  DISPLAY_4 : OUT std_logic;
  DISPLAY_3 : OUT std_logic;
  DISPLAY_2 : OUT std_logic;
  DISPLAY_1 : OUT std_logic;
  H3 : IN std_logic;
  H2 : IN std_logic;
  H1 : IN std_logic;
  H0 : IN std_logic;
  DISPLAY_12 : OUT std_logic;
  DISPLAY_11 : OUT std_logic;
  DISPLAY_10 : OUT std_logic;
  DISPLAY_9 : OUT std_logic;
  DISPLAY_8 : OUT std_logic;
  DISPLAY_7 : OUT std_logic;
END COMPONENT;
ARCHITECTURE ARCH_GPIO3Top OF FPGA3Top IS
COMPONENT SIGEN_STATE_MACHINE
PORT :
  CLK : IN std_logic;
  COMPARE_IN : IN std_logic;
  TOGGLE_ON : IN std_logic;
  RING : OUT std_logic;
END COMPONENT;
COMPONENT COMPARATOR
PORT :
  ALARM_AM_PM : IN std_logic;
  ALARM_HRS : IN INTEGER RANGE 1 TO 12;
  ALARM_MINS : IN INTEGER RANGE 0 TO 59;
  TIME_AM_PM : IN std_logic;
  TIME_HRS : IN INTEGER RANGE 1 TO 12;
  TIME_MINS : IN INTEGER RANGE 0 TO 59;
  COMPARE_OUT : OUT std_logic;
END COMPONENT;
COMPONENT MUX
PORT :
  ALARM : IN std_logic;
  ALARM_AM_PM : IN std_logic;
  ALARM_HRS : IN INTEGER RANGE 1 TO 12;
  ALARM_MINS : IN INTEGER RANGE 0 TO 59;
  TIME_AM_PM : IN std_logic;
  TIME_HRS : IN INTEGER RANGE 1 TO 12;
  TIME_MINS : IN INTEGER RANGE 0 TO 59;
  AM_PM : OUT std_logic;
  HRS_BIN : OUT std_logic_vector(3 DOWNTO 0);
  MINS_BIN : OUT std_logic_vector(5 DOWNTO 0);
END COMPONENT;
COMPONENT alarm_block_comp
PORT :
  AM_PM_OUT_DUMMY : IN std_logic;
  MINS_OUT_DUMMY : IN INTEGER RANGE 0 TO 59;
  HRS_OUT_DUMMY : IN INTEGER RANGE 1 TO 12;
  AM_PM_OUT : BUFFER std_logic;
  HRS_OUT : BUFFER INTEGER RANGE 1 TO 12;
  MINS_OUT : BUFFER INTEGER RANGE 0 TO 59;
END COMPONENT;
COMPONENT ALARM_STATE_MACHINE
PORT :
  ALARM_BUTTON : IN std_logic;
  CLK : IN std_logic;
  HRS_BUTTON : IN std_logic;
  MINS_BUTTON : IN std_logic;
  RESET : IN std_logic;
  INC_HRS : OUT std_logic;
  INC_MINS : OUT std_logic;
  INC_TIME : OUT std_logic;
END COMPONENT;
COMPONENT ALARM_COUNTER
PORT :
  CLK : IN std_logic;
  INC_HRS : IN std_logic;
  INC_MINS : IN std_logic;
  RESET : IN std_logic;
  ALARM_BUTTON : IN std_logic;
  INC_TIME : IN std_logic;
  INC_TIME : OUT std_logic;
  INC_MINS : OUT std_logic;
  INC_TIME : OUT std_logic;
END COMPONENT;
COMPONENT TIME_STATE_MACHINE
PORT :
  CLK : IN std_logic;
  RING : IN std_logic;
  TIME_BUTTN : IN std_logic;
  TIME_STATE : IN std_logic;
  TIME_STATE : OUT std_logic;
  RING : IN std_logic;
  TIME_BUTTN : IN std_logic;
  TIME_STATE : IN std_logic;
  TIME_STATE : OUT std_logic;
END COMPONENT;
COMPONENT TIME_COUNTER
PORT :
  CLK : IN std_logic;
  INC_HRS : IN std_logic;
  INC_MINS : IN std_logic;
  INC_TIME : IN std_logic;
  INC_TIME : OUT std_logic;
  INC_TIME : OUT std_logic;
  INC_TIME : OUT std_logic;
END COMPONENT;
COMPONENT CONVERTER
PORT :
  DATA_IN : IN std_logic_vector(5 DOWNTO 0);
  OOKS_DIGIT : OUT std_logic_vector(6 DOWNTO 0);
  TENS_DIGIT : OUT std_logic_vector(6 DOWNTO 0);
END COMPONENT;
COMPONENT HOUR_FILTER
PORT :
  TIME_IN : IN std_logic_vector(6 DOWNTO 0);
  TIME_OUT : OUT std_logic_vector(6 DOWNTO 0);
END COMPONENT;
COMPONENT ZERO
PORT :
  ZERO : OUT std_logic;
END COMPONENT;
SIGNAL CONVERT_A_6 : std_logic;
SIGNAL CONVERT_B_4 : std_logic;
SIGNAL CONVERT_B_3 : std_logic;
SIGNAL CONVERT_B_2 : std_logic;
SIGNAL CONVERT_B_1 : std_logic;
SIGNAL CONVERT_B_0 : std_logic;
SIGNAL CONVERT_S1 : std_logic;
SIGNAL CONVERT_S2 : std_logic;

BEGIN

CONVERT_HRS_CONVERTER : CONVERTOR PORT MAP (
  DATA_IN(5) => CONVERT_S1,
  DATA_IN(4) => CONVERT_S2,
  DATA_IN(3) => H_0,
  DATA_IN(2) => H_1,
  DATA_IN(1) => H_2,
  DATA_IN(0) => H_3,
  ONES_DIGIT(0) => DISP1_0,
  ONES_DIGIT(1) => DISP1_1,
  ONES_DIGIT(2) => DISP1_2,
  ONES_DIGIT(3) => DISP1_3,
  ONES_DIGIT(4) => DISP1_4,
  ONES_DIGIT(5) => DISP1_5,
  ONES_DIGIT(6) => DISP1_6,
  TENS_DIGIT(0) => CONVERT_B_0,
  TENS_DIGIT(1) => CONVERT_B_1,
  TENS_DIGIT(2) => CONVERT_B_2,
  TENS_DIGIT(3) => CONVERT_B_3,
  TENS_DIGIT(4) => CONVERT_B_4,
  TENS_DIGIT(5) => CONVERT_B_5,
  TENS_DIGIT(6) => CONVERT_B_6);

CONVERT_HRS_FILTER : HOURS_FILTER PORT MAP (
  TENS_IN(5) => CONVERT_B_5,
  TENS_IN(4) => CONVERT_B_4,
  TENS_IN(3) => CONVERT_B_3,
  TENS_IN(2) => CONVERT_B_2,
  TENS_IN(1) => CONVERT_B_1,
  TENS_IN(0) => TENS_CLR,
  TENS_OUT(0) => TENS_CLR,
  TENS_OUT(1) => TENS_CLR,
  TENS_OUT(2) => TENS_CLR,
  TENS_OUT(3) => TENS_CLR,
  TENS_OUT(4) => DISP1_10,
  TENS_OUT(5) => DISP1_11,
  TENS_OUT(6) => DISP1_12);

CONVERT_S1 : ZERO PORT MAP (ZERO => CONVERT_S1);

CONVERT_S2 : ZERO PORT MAP (ZERO => CONVERT_S2);

END ARCHITECT;

END ARCHITECT;
CHAPTER C. VHDL SOURCE CODES AND SCHEMATICS

C.2.5.4 Partition Four
ENTITY FPGAtop IS
  PORT ( 
    TM : IN INTEGER RANGE 0 TO 59 ;
    TP : IN std_logic;
    AM : IN INTEGER RANGE 0 TO 59 ;
    AH : IN INTEGER RANGE 1 TO 12 ;
    AP : IN std_logic;
    DISP2_13 : OUT std_logic;
    DISP2_12 : OUT std_logic;
    DISP2_11 : OUT std_logic;
    DISP2_10 : OUT std_logic;
    DISP2_9 : OUT std_logic;
    DISP2_8 : OUT std_logic;
    DISP2_7 : OUT std_logic;
    DISP2_6 : OUT std_logic;
    DISP2_5 : OUT std_logic;
    DISP2_4 : OUT std_logic;
    DISP2_3 : OUT std_logic;
    DISP2_2 : OUT std_logic;
    DISP2_1 : OUT std_logic;
    DISP2_0 : OUT std_logic;
    K_5 : IN std_logic;
    K_4 : IN std_logic;
    K_3 : IN std_logic;
    K_2 : IN std_logic;
    K_1 : IN std_logic;
    K_0 : IN std_logic;
    SPEAKER_OUT : OUT std_logic;
    TOGGLE_SWITCH : IN std_logic;
    CLK : IN std_logic ;
  );
END FPGAtop;
ARCHITECTURE ARCH FPGAtop OF FPGAtop IS
  COMPONENT ALARM_STATE_MACHINE
    PORT ( 
      CLK : IN std_logic;
      INC_HRS : IN std_logic;
      INC_MINS : IN std_logic;
      RESET : IN std_logic;
      HRS_OUT : BUFFER std_logic;
      MINS_OUT : BUFFER INTEGER RANGE 0 TO 59 ;
    );
END COMPONENT;
  COMPONENT TIME_COUNTER
    PORT ( 
      CLK : IN std_logic;
      INC_HRS : IN std_logic;
      INC_MINS : IN std_logic;
      RESET : IN std_logic;
      TIME : IN std_logic;
    );
END COMPONENT;
  COMPONENT ALARM_COUNTER
    PORT ( 
      CLK : IN std_logic;
      INC_HRS : IN std_logic;
      INC_MINS : IN std_logic;
    );
END COMPONENT;
  COMPONENT CONVERTOR
    PORT ( 
      DATA_IN : IN std_logic_vector(5 DOWNTO 0 );
      HOURS_DIGIT : OUT std_logic_vector(3 DOWNTO 0 );
      MINS_DIGIT : OUT std_logic_vector(3 DOWNTO 0 );
    );
END COMPONENT;
  COMPONENT COMPARATOR
    PORT ( 
      ALARM_AM_PM : IN std_logic;
      ALARM_HRS : IN INTEGER RANGE 1 TO 12 ;
      ALARM_MINS : IN INTEGER RANGE 0 TO 59 ;
      TIME_AM_PM : IN std_logic;
      TIME_HRS : IN INTEGER RANGE 1 TO 12 ;
      TIME_MINS : IN INTEGER RANGE 0 TO 59 ;
      COMPARE_OUT : OUT std_logic ;
    );
END COMPONENT;
  COMPONENT ALARM
    PORT ( 
      ALARM : IN std_logic;
      ALARM_AM_PM : IN std_logic;
      ALARM_HRS : IN INTEGER RANGE 1 TO 12 ;
      ALARM_MINS : IN INTEGER RANGE 0 TO 59 ;
    );
END COMPONENT;
  COMPONENT ALARM_BLACK_COMP
    PORT ( 
      AM_PM : OUT std_logic;
      HOURS : OUT std_logic_vector(3 DOWNTO 0 );
      MINS : OUT std_logic_vector(3 DOWNTO 0 );
    );
END COMPONENT;
END ARCHITECTURE ARCH FPGAtop ;
COMPONENT HOURS_FILTER
PORT (  
  TENS_IN : IN std_logic_vector(6 DOWNTO 0);  
  TENS_OUT : OUT std_logic_vector(6 DOWNTO 0));
END COMPONENT;

COMPONENT ZERO
PORT (  
  ZERO : OUT std_logic;)
END COMPONENT;

SIGNAL N_1 : std_logic;

BEGIN
  COMPARE : COMPARATOR PORT MAP (  
    ALARM_AM_PM => AP,  
    ALARM_HRS => AH,  
    ALARM_MINS => AM,  
    TIME_AM_PM => TP,  
    TIME_HRS => TH,  
    TIME_MINS => TM,  
    COMPARE_OUT => N_1);  
  CONVERT_MINS_CONVERTER : CONVERTER PORT MAP (  
    DATA_IN(0) => N_0,  
    DATA_IN(1) => N_1,  
    DATA_IN(2) => N_2,  
    DATA_IN(3) => N_3,  
    DATA_IN(4) => N_4,  
    DATA_IN(5) => N_5,  
    ONS_DIGIT(0) => DISP2_0,  
    ONS_DIGIT(1) => DISP2_1,  
    ONS_DIGIT(2) => DISP2_2,  
    ONS_DIGIT(3) => DISP2_3,  
    ONS_DIGIT(4) => DISP2_4,  
    ONS_DIGIT(5) => DISP2_5,  
    ONS_DIGIT(6) => DISP2_6,  
    TENS_DIGIT(0) => DISP2_7,  
    TENS_DIGIT(1) => DISP2_8,  
    TENS_DIGIT(2) => DISP2_9,  
    TENS_DIGIT(3) => DISP2_10,  
    TENS_DIGIT(4) => DISP2_11,  
    TENS_DIGIT(5) => DISP2_12,  
    TENS_DIGIT(6) => DISP2_13);  
  RSM : RINGER_STATE_MACHINE PORT MAP (  
    CLK => CLK,  
    COMPARE_IN => N_1,  
    TOGGLE_IN => TOGGLE_SWITCH,  
    RING => SPEAKER_OUT);  
END ARCHPPDA4top;
C.2.6  Schematic Diagrams After Partitioning

C.2.6.1  Partition One
C.2.6.2 Partition Two
C.2.6.3 Partition Three
C.2.6.4 Partition Four
C.2.7 Manually Generated VHDL File For Simulation Of Partitioned Design
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity FPGAAlltop is
PORT :
  SPEAKER_OUT : OUT std_logic;
  DISP2_13 : OUT std_logic;
  DISP2_12 : OUT std_logic;
  DISP2_11 : OUT std_logic;
  DISP2_10 : OUT std_logic;
  DISP2_9 : OUT std_logic;
  DISP2_8 : OUT std_logic;
  DISP2_7 : OUT std_logic;
  DOOR_SWITCH : IN std_logic;
  SET_TIME : IN std_logic;
  RESET : IN std_logic;
  MNS : IN std_logic;
  HRS : IN std_logic;
  CLK : IN std_logic;
  ALARM : IN std_logic;

END component;

architecture structural of FPGAAlltop2 is
component FPGAAlltop
PORT :
  HRS : IN std_logic;
  CLK : IN std_logic;
  MNS : OUT std_logic;
  HRS : OUT std_logic;
  MNS : OUT std_logic;
  ALARM : OUT std_logic;
  TM : IN INTEGER RANGE 0 TO 59;
  TH : IN INTEGER RANGE 1 TO 12;
  TP : IN std_logic;
  AM : BUFFER INTEGER RANGE 0 TO 59;
  AP : BUFFER INTEGER RANGE 1 TO 12;
  AP : BUFFER std_logic;
  ALARM : IN std_logic

END component;

component FPGAAlltop
PORT :
  TM : IN INTEGER RANGE 0 TO 59;
  TH : IN INTEGER RANGE 1 TO 12;
  TP : IN std_logic;
  AM : IN INTEGER RANGE 0 TO 59;
  AP : IN INTEGER RANGE 1 TO 12;
  AP : IN std_logic;
  TM : OUT std_logic;
  TH : OUT std_logic;
  TP : OUT std_logic;
  AM : OUT INTEGER RANGE 0 TO 59;
  AP : OUT INTEGER RANGE 1 TO 12;
  AP : OUT std_logic;
  TM : IN INTEGER RANGE 0 TO 59;
  TH : IN INTEGER RANGE 1 TO 12;
  TP : IN std_logic;
  AM : BUFFER INTEGER RANGE 0 TO 59;
  AP : BUFFER INTEGER RANGE 1 TO 12;
  AP : BUFFER std_logic;
  TM : OUT std_logic;
  TH : OUT std_logic;
  TP : OUT std_logic;
  AM : OUT INTEGER RANGE 0 TO 59;
  AP : OUT INTEGER RANGE 1 TO 12;
  AP : OUT std_logic;

END component;

end FPGAAlltop2;

architecture structural of FPGAAlltop2 in component FPGAAlltop
PORT :
  CLK : IN std_logic;
  RESET : IN std_logic;
  MNS : IN std_logic;
  HRS : IN std_logic;
  AM : IN INTEGER RANGE 0 TO 59;
  AP : IN INTEGER RANGE 1 TO 12;
  AP : IN std_logic;
  TM : OUT std_logic;
  TH : OUT std_logic;
  TP : OUT std_logic;
  AM : OUT INTEGER RANGE 0 TO 59;
  AP : OUT INTEGER RANGE 1 TO 12;
  AP : OUT std_logic;

END component;

component FPGAAlltop
PORT :
  RESET : IN std_logic;
  MNS : IN std_logic;

END component;
C.2.8 Schematic Diagram for Manually Generated VHDL File For Simulation
Bibliography


[31] Thomas Dettmer, Dortmund University Dept. of Computer Science. dettmer@ls1.informatik.uni-dortmund.de.


http://www.eecg.toronto.edu/~tm3/