REINALDO SILVEIRA

GUIDELINES FOR A NEW DIGITAL PROJECT METHODOLOGY

Thesis presented to the School Polytechnic of the University of São Paulo to obtain a title

of Doctor in Electrical Engineering.

São Paulo

2004

left blank

REINALDO SILVEIRA

GUIDELINES FOR A NEW DIGITAL PROJECT METHODOLOGY

Thesis presented to the School Polytechnic of the University of São Paulo to obtain the title of Doctor in Electrical Engineering.

Concentration area:

Microelectronics

Advisor: Prof. Holder Wilhelmus AM Van Noije

São Paulo

2004

CATALOG FORM

Silveira, Reinaldo

Guidelines for a New Digital Design Methodology / Reinaldo Silveira – São Paulo, 2004.

212 p.

Thesis (Doctorate) – Polytechnic School of the University from Sao Paulo. Systems Engineering Department Electronics.

1. CAD (tools) 2. VLSI Integrated Circuits 3. Computer architecture and organization I. University from Sao Paulo. Polytechnic School. Department of Electronic Systems Engineering II. t

I would like to dedicate this work

to my relatives and friends who

they knew how to support and encourage me

throughout this period.

left blank

Thanks

To Prof. Wilhelmus AM Van Noije for his valuable guidance and support. received during the performance of the work, and believing in its viability, even knowing that its realization could advance through fields of knowledge.

new cement for both of us.

To friend and colleague Jecel Assumpção for introducing me to the language Self and for guiding me through the first steps of object-oriented programming, Smalltalk and Self.

To the National Council for Scientific and Technological Development (CNPq), for the scholarship that enabled the maintenance and completion of this work. for the past three years.

My friend Profa. Noely Ielo de Campos, for her valuable advice on writing style and review of the work. To colleague and friend Gustavo Adolfo Cerezo Vásquez for the also valuable suggestions and help received.

And to all colleagues and professors at Poli (especially LSI), with the which I lived with for a long time and helped in my basic training in research that culminated in this work.

My sincere thanks.

left blank

Summary

This work presents a new digital design methodology, called “Metodolo-
Designer Oriented Technology”, or simply DO (“Designer Oriented”). The methodology
DO basically seeks to eliminate, from the project flow, concepts and operations that are foreign to the
application domain. This is done through the use of computational tools especially
projects whose mode of operation seeks to favor the user (hardware designer),
in order to always be as intuitive as possible and adapt to your needs. Of that
In this way, we seek to transform the design task into something more accessible, requiring less
training, consequently reducing the costs associated with it.

To demonstrate the methodology, we propose the SelfHDL system which is a system of
description of hardware implemented in Self language. System that uses graphic resources
cos and texts to elaborate the description of a digital system. Description this extremely
simple to be understood, allowing it to be used also as a didactic system for
support for teaching digital systems. The described system can also be simulated for-
interactive through a virtual environment in which the described system is emulated,
being able to interact with the user or with the computing environment that surrounds it as if the
described hardware actually existed. As it is implemented in Self, the SelfH-
DL does not need to be compiled, any modification takes effect immediately as if
was an interpreted system. This work presents the implementation of SelfHDL and the
its use through an example project.

The SelfHDL implementation of the DLX architecture is used as an example project.
In it we can compare a traditional implementation made in VHDL with the description
SelfHDL and see in practice the advantages of the new system. We will see that this new form
is very intuitive for the designer, normally used to dealing with representations and
graphic models of the elements of your domain and whose spirit is normally educated in

experimentation and manipulation. The SelfHDL system is a favorable environment for designers can evaluate several alternatives for their projects under development.

Finally, we concluded the work talking about the potential of the system and the works futures to which we intend to dedicate ourselves or even guide them as lines of research in order to to extend the system and make it more powerful.

Abstract

This work presents a new methodology for digital design, called “Designer Oriented Methodology” (DO). The main idea behind the DO methodology is to make the computer national tools closer to the domain of application. This is done by eliminating, from the design flow, tasks and concepts strange or superfluous to the final user. This methodology uses especially designed computer tools to avoid or hide unwanted aspects of the design from the user (hardware designer), The design process should interact with the system in a way that the designer's attention should be totally focused on the subject of his/hers work, as as much as possible. Like a player and his game are involved and absorbed by the reality of the game.

To demonstrate the methodology, it is presented the system SelfHDL which is a hardware description language written in the programming language Self. SelfHDL uses graphical and textual elements to elaborate a description of a digital system, that is extremely simple to be understood allowing its use as didactic support system for the education aid of digital systems. The described system can also be simulated in interactive mode, through a virtual environment where the described system is emulated, being able to interact with the user or the computer environment that surrounds it, as the real hardware would. Being implemented in Self, means that the SelfHDL system does not need to be compiled

led, so any modification has immediate effect as if the system were interpreted. this work presents the implementation of SelfHDL and its use in an example project.

The SelfHDL implementation of the DLX architecture is used as the example project. We compare two traditional DLX designs made in VHDL with the SelfHDL description and discuss the advantages of the new system. We will see that this new form is very intuitive for designers, normally used to deal with graphic models of the elements of their domain and whose spirit normally is educated in experimentation and manipulation. The SelfHDL system is a propitious environment where the designers can evaluate many alternatives for their projects under development. Finally, we conclude the work speaking of the potentialities of the new system and the future works we intend to dedicate or advise as a new research field in order to extend the system and to make it more powerful.

Contents

Page 12

Page 14

List of Figures

Page 16

List of Tables

Chapter 1

Introduction

The design of the electronic components was the critical factor in the project. for a long time The beginning of microelectronics, the complexity was relatively low, the correct dimension. time, this was the characteristic of the projects, relatively simple architectures to implement. constantly changing technologies (TTL, NMOS, CMOS, BiCMOS, etc.). Was It is natural that methodologies and tools accompany this development. The cost implementation of integrated circuits was relatively high and costly in time. that the allowable margin of error in a project was practically zero *. In this way, it was developed a vast set of verification and simulation tools in order to reduce the possibilities of logical/structural error, and thus to enable the implementation of a new integrated circuit.

Currently, the complexity of some systems reaches the order of tens of millions components on a single silicon wafer (Systems on a Chip) and are still used methodologies and tools conceived over twenty years ago. Obviously these tools have also evolved, but continue to act in the component's implementation part. There has been a great evolution in algorithms, data structures and automation, making the implementation task almost automatic in some cases. the conception of new architectures, however, it is still an almost pure work. intellectual.

Nowadays, most of the development time is spent on system design.

* Today, in the most sophisticated technologies, these costs are so high that errors need to be totally eradicated.

and in the functional/architectural/structural definition, while the implementation itself passes occupying only one-fifth of the total project time. In the past, this reason was inverse, hence the tremendous development experienced by tools in recent decades. O time is to review the way the design tools are thought, attacking the system design problem. Therefore, the purpose of this work is to outline the guidelines basics of a new digital design methodology that overcomes design difficulties of extremely complex digital systems and that is the starting point of development of a new set of tools that will support this new methodology.

In the early 1980s, there was great difficulty in adapting the designers of integrated circuits to the new scenario that was then configured. Due to the rapid development development of VLSI technologies, the pressure for more complex devices, at a level of higher integration, made the design task arduous, time-consuming and costly. the cost of implementation of the prototypes also made it necessary to raise the degree of con- reliability of the designed elements. This led Prof. Daniel D. Gajski [Gaj88] calling this period of the “design crisis” of the 1980s.

Two strategies were adopted to overcome the difficulties. One took into consideration reason that man was the main source of knowledge, therefore automation should provide tools that increased their productivity. The second considered that the knowledge needed to make a project could be “captured” and closed in program. but, the so-called "Silicon Compilers", which in turn could generate the VLSI devices automatically, from a “high-level” description. In fact these two strategies gave a big boost to VLSI system designs because they allowed designers to reach a level of abstraction where implementation and fabrication details were not most needed.

The term “Silicon Compilation” was first introduced by Dave Johannsen [Gaj88], when referring to the concept of parameterized assembly of Layout parts. O term became very popular and quickly took on a much broader connotation. than originally proposed. In reality, the silicon compilation is an extension of the Standard-cell concept, in which standardized cells are positioned and interconnected

along with special cells generated by cell compilers. More recently the generation of complex components featuring regular and/or symmetrical microarchitecture, such as ROMs,

RAMs, PLAs, ALUs, etc., have been associated with the functions of this class of programs. In generally speaking, it seems more convenient to understand the term by its broader aspect, that is, we must understand silicon compilation only as the process of obtaining the layout a from a high-level description.

Another concept, following the silicon compilers strategy, of increasing the program intelligence is that of high-level synthesis. In it, an architecture could be synthesized from a high-level description, possibly algorithmic. In this area there are some advances; however, because it is very difficult to define objectively which is really high level and also if you map all classes of problems/solutions, the existing tools ended up supporting only a few specific problems (very due to its regularity, such as digital signal processing), proliferating. especially in academic circles.

Looking deeper into the two strategies, we can see that their purpose The initial point was to attack the design problem at two different levels: the design problem, in this case. of silicon compilers and high-level synthesis; and the implementation, in the case of the other CAD tools. Unfortunately, both the compilation and the high-level synthesis, have been more successful in terms of implementation than in terms of the actual design, again, due to inconsistencies of what would be really high level or not.

1.1 Justification and Motivations

The process of creating an architecture (designing a hardware solution) passes

through several phases: Problem Definition/Analysis; System Project; element design components; and finally Implementation. Obviously, given the high complexity of the pro- problems that currently exist, computational tools are needed in each of its steps. The ultimate goal is to obtain a description that can be shared with the later stages of implementation. In the case of a digital system, this description is typically a description made using a hardware-level description language RTL (Register Transfer Level). From such a description, tools could be used methods of logical synthesis and silicon compilation to implement the prototype.

Depending on the degree of structuring of the designer, we can identify the same stages.

steps of developing an object-oriented software system, perhaps as a function of the very nature of the tools used. The truth is that the vast majority of fer- tools was developed by computer scientists and not by design engineers. digital/electronic. The predominant view of these tools is impregnated with the profile of the computing professionals and not with engineering professionals, the result is that the project flow and tasks in general do not follow a very intuitive pattern for the technical professional.

Let's take a simple example: When developing a project, we aim to ob- intending a system description in a hardware description language, for example VHDL, at RTL level (synthesizable). In the problem definition/analysis stage, it is common the need for experiments in order to better understand the problem that if you want to resolve; for this, hardware description languages offer a modality called “behavioral” to describe hardware modules not exactly as they must be implemented, but only as they are expected to behave external. This is done because the behavior is generally easier to describe than the real internal structure. This is called “Levels of Abstraction”, that is, abstracting the

structure and focus only on the function. Unfortunately, languages dedicated to the description of hardware are very limited in some aspects, it is very common for the designer to have to choose to use another, more flexible language to get results faster.
The engineer is required to assume the role of programmer to complete the task.

In the following Steps, System Design and Component Elements, follow more or minus the same steps in terms of tools. The behavioral description in a HDL (Hardware Description Language) any or in a general purpose language such as C or C++, plus all the associated overhead, makefiles, listings, includes, function libraries, compilations, simulations, analysis of results, corrections and another new cycle of iterations starts. That is, it is a typical software project using development tools of software to make a hardware project.

The final step: Implementation would be the most linked to the implementation itself said (and the objective of this type of development). The RTL description has more similarity with the physical structure of the component than its behavior. By following strict rules, the RTL description is often incompatible with the descriptions obtained so far in the steps. previous countries, and it is often necessary that they be rewritten for this

compliance is achieved. Even at this stage, the same tools and prior techniques, ie typically software development methodologies.

We can highlight several disadvantages to this type of approach. The most immediate is from need to transform an engineering professional into a computer professional. dog. Although engineering courses have a good amount of programming, it is still Good training of the professional is necessary in order to qualify him for this type of task. For a project of this type, it is not enough for the professional to have good algorithmic reasoning. mic, but it is also necessary that he can reason in terms of components and

logical gates.

Another disadvantage concerns the software project cycle. The cycle: editing, with-pilation, simulation and analysis, is intrinsically disconnected; was designed to work on batch or background at a time when processing time was expensive and Mainframes. We believe that a development must be iterative, continuous, uninterrupted in order to prevent the designer from straying into unrelated thoughts the task in process. Under current methodology, several minutes may elapse between a correction and the analysis of a particular detail, at which time the designer have to start another task or have a "coffee" to take advantage of the idle time, invariably straying from the important details of the project.

1.2 Objectives

This work proposes an alternative way of making the digital project, called “Method-Designer Oriented Methodology”, or simply DO (Designer Oriented Methodology). That methodology seeks to emphasize some fundamental principles, such as:

- The creation process needs to be iterative.
- The level of abstraction you want to achieve is the RTL, so that the later steps of implementation remain unchanged.
- The level of abstraction used in the intermediate phases must be indeterminate, so large as much as you want (or as needed).

- The hardware elements have autonomous characteristics, that is, they incorporate automatically the assigned behavior without the need for major intervention.

  of the designer.

- Just as graphic environments present the desktop metaphor (Desktop

  Metaphore) in today's computers, the new methodology must present the metaphor

  of the workbench.

- Use of computational tools to support the previous items. In this

  In this work, the SelfHDL system is implemented for the description of digital hardware, indi-

  especially for designing and exploring the architecture of digital systems.

In order to make possible the realization of this methodology, a great computational support.
nal was needed. The use of a high-level programming language, oriented to
objects, with characteristics of interactivity and accessibility to objects has been very
helps in the implementation of this computational support. We then chose the language Self
[US87] to fulfill this function. Self is an object-oriented programming language
based on prototypes and dynamic types developed in 1986 by David Ungar and Randall B.
Smith at Xerox PARC. Designed as an alternative to the Smalltalk-80 language [GR83],
Self seeks to maximize productivity through an exploitative programming environment.
at the same time keeping the language simple and pure without compromising the
expressiveness to malleability.

1.3 Conventions

In this work, we chose not to translate the technical terms commonly used in the field of
design of digital systems and CAD tools, with the exception of cases in which some
clarification if necessary. Therefore, all foreign language words are
displayed in italic characters, for example: bits, bytes, Hardware Description Language-
ge. During the presentation, we will also refer to software elements, such as
listings of programs, objects, etc. In this case we use cons- spacing characters.
tant and “courier” type. In this case, it is necessary to distinguish between message and
object, very peculiar when we describe elements of the Self or the SelfHDL. Therefore,

we conventionally refer to messages by enclosing them in quotation marks (""), in addition to the type "courier", when they appear in the middle of the explanatory text. For example: the circleMorph object has the “position” method that returns the position of the center of the circle in car coordinates. tesianas. The message “color: aColor” modifies the object's color to the new aColor value. Other times we can use the quotation marks simply to draw attention to the term, like in this sentence or when we talk about the “courier” type in this section. Another example of usage of quotation marks can be seen in section 2.2.5.1 , when we refer to the names of some objects. O name of some objects may consist of more than one word, so for larger clarity, we use quotation marks to make it stand out from the text that surrounds it, for example: "a point" is the name of an object of type point.

1.4 Organization of the Thesis

In chapter 2 we present a survey of the main tools currently in development in the area of design of digital systems and integrated circuits. We locate each development, compared to this work. In this chapter we present still the foundations of the Self language, in order to highlight the qualities that made it eligible to carry out this work and provide a basis for the understanding of SelfHDL implementation. This chapter should not be seen as a kind of tutorial, rather a reference for those who are unfamiliar with the philosophies used by a certain class of object-oriented languages, of which it makes part of the Self language.

In chapter 3 we will present the designer-oriented (DO) design methodology. besides some of the reasons that led us to propose this alternative methodology. Fan- we carry out a theoretical/philosophical development and argumentation based on certain principles

of hermeneutics, and we will begin the detailed description of the desired characteristics in tools. of this methodology. Finally, we conclude the chapter by superficially discussing the potential features of the SelfHDL system, used in this work to demonstrate the DO methodology.

In chapter 4 we present the implementation of the SelfHDL system, in which we show a detailing of the implementation of the main objects and their dynamics, as well

as the presentation of the first examples.

In chapter 5 we present the tests performed using the SelfHDL system. Show- s how the SelfHDL descriptions are superior in relation to the VHDL descriptions, through of the implementation of the DLX processor architecture. This implementation is compared to two other classical implementations and we'll see how the expressiveness level of the shapes traditional is inferior in relation to our proposal. We also see that the simulation features The interactive loop of the SelfHDL system make it an environment conducive to experimenting with architectures and as a didactic support tool.

Finally, in chapter 6 we present a summary of the general conclusions of this work. and future possibilities and motivations in relation to the SelfHDL methodology and system, the in order to make it more powerful and flexible.

Chapter 2

State of art

Not CAD minds for digital design in general. Our aim is to situate the work of this chapter we will make a presentation of hand tools development landscape within the various fields of research existing in recent years, as well as serving as basis for the concepts we will introduce.

The approach to tools will, in principle, have a demonstration character only, we will avoid, whenever possible, going into technical or very complicated details, which do not are the objective of this work. On the other hand, we will also present the language programming tool that will be used by us and that is responsible for many of the concepts that we will introduce. The presentation of the Self language is necessary, as different from In other languages, Self is very different in many ways. being generally unfamiliar the large portion of programmers.

2.1 State of the Art in Digital Design Tools

Since the appearance of the first integrated devices in the 60's until the days of today, the design of integrated circuits has been a big challenge. Some aspects of this challenge have changed over the years; however, all efforts seem to obey to what came to be known as “Moore's Law” [Cor04], enunciated by Gordon E. Moore (founder of the Intel Corporation) in 1965 [Moo65]. In this article, Moore predicts the importance of the integrated devices and observes that until then the integration rate had been doubled every two years.

The integrated circuit industry, in general, took this statement as a postulate, directing all your efforts towards keeping this premise true. For another On the other hand, there has also been an expectation, in the same sense, on the part of consumers, that pressure the demand for products with the same rates of improvement dictated

by that law. Intel, for example, is one of the companies that prides itself on showing the evolution of its main product line according to Moore's law [Moo03], see for example to figure 2.1 .

Figure 2.1: Moore's Law according to the Intel processor lineup.

Since then, technology has evolved according to these standards and at the expense of large investments, making, even today, the design and manufacture of circuits/systems integrated is out of reach for the vast majority of people. Philosophies apart, by for many years the technological cost was concentrated in the manufacturing processes of circuits. The instability of the processes and the low yields caused the cost of the devices get too high. At that time, the tools to help the electronic project. Although the circuits are well known, their size and the increasing functionality required countless hours of verification, as a CI that does not to function after fabrication was too expensive to tolerate. The breadboarding technique

was applied whenever possible during development, however, it was not a guarantee of success. The transition from the conceptual circuit to the micro circuit was invariably done under human intervention, resulting in a large number of potential errors.

Then came the first simulators and layout verification systems. In se- guide, schematic capture programs, since until then electrical and electronic circuits they were invariably represented in terms of schematic diagrams. This method of capture-simulation worked very well and has been popular for over three decades [gaj93]. Or- other important improvements were introduced such as the concept of hierarchical levels. and basic cells that could and would be used throughout the project, regular structures like memories and PLAs, and etc. [MC80 , WE88].

All these improvements along with the consolidation of the manufacturing processes made the pressure for more complex circuits to increase, and with it the difficulties, the time and costs of projects. Prof. Gajski called this period the “Design Crisis” of the 80s [Gaj88, DG90]. It was clear at that time that tools* to increase the productivity of the designers were not enough to deal with the demands that were coming up. These tools follow a line of thinking that says the process of development of an integrated circuit is very difficult to be conceived by automatic means. and that the human designer is the main source of knowledge for this purpose. Therefore, the tools must promote the productivity of the human designer through the automatic. tion of routine tasks and/or increased efficiency in others. They are generally tools capture, verification, analysis and optimization. For example: simulators, testers of layout rules, timing analyzer (timing), compressors and so on. are tools that automate repetitive, long and tiring tasks.

At that time, the idea emerged that it was possible to create synthesis and compilation capable of generating the layout of VLSI systems automatically. These tools would generate the layout from some “high-level” description of the system, which could be a symbolic layout, a schematic diagram of a circuit, a behavioral description of a microarchitecture, set of instructions or a processing algorithm.

signals. These tools became known generically as “Silicon Compilers”.

* We will normally use the term tool in the sense of computational tool, that is, programs developed specifically for certain tasks.

Both strategies had good results. Experience has shown us that the two approaches when used together can produce very efficient tools for the implementation of the circuits. However, throughout history, these tools were occupied exclusively with the implementation of the circuits in silicon, since until then the manufacturing was the most costly and determining factor in the success of a project.

In the late 1980s and early 1990s, other aspects became more important in the implementation of VLSI systems. The manufacturing processes were more or less equivalent. lenses, several manufacturers offered technologies that followed well-established trends and that implemented circuits with similar performance; so the only way to differ- to rate their products and make them more competitive was to employ improvements in circuit and the use of new architectures. Circuit techniques were mainly aimed at circuits that offered the maximum in terms of performance on the one hand [NN99, NN02], and low consumption on the other [CB95, Yea98], and the ideal was always the conjugation of two features.

In relation to architectures, mainly in the design of processors, it began to make the use of techniques hitherto used only on computers more common large-sized. It has become much more common to use superscalar architectures, large trace caches, standby stations, functional units with deep pipelines, data on-chip caches, out-of-order execution, speculative, etc. [PPE + 97, yes97]. of course way this was already predictable, in 89 [GGPY89] compared the evolution of microprocessors with mainframes, and predicted that their performance would outperform mainframes in

Few years. It was characterized in this way, the importance of architecture in the project final as a factor of competitiveness.

As a result, the 90's were characterized by an explosion of size and com-complexity of the integrated components. First, processors, followed by peripherals that have jumped from hundreds of thousands to tens of millions of components by integrated circuit, thanks mainly to the use of increasingly so-phisticated. However, there has been relatively little advance in the tools to assist in this type of development.

We will see in the next sections an overview of what was developed in terms of tools. Digital systems design tools from the mid 80's, 90's to the present day.

We will see the most important: that all the tools and methodologies presented have a lot of to more implementation character than conception, main objective of this work.
Even considering Hardware Description Languages, very fashionable nowadays.
te, we will see its limitations and difficulties in relation to the elaboration and conception aspect.
architecture of new systems.

2.1.1 Silence Compilation

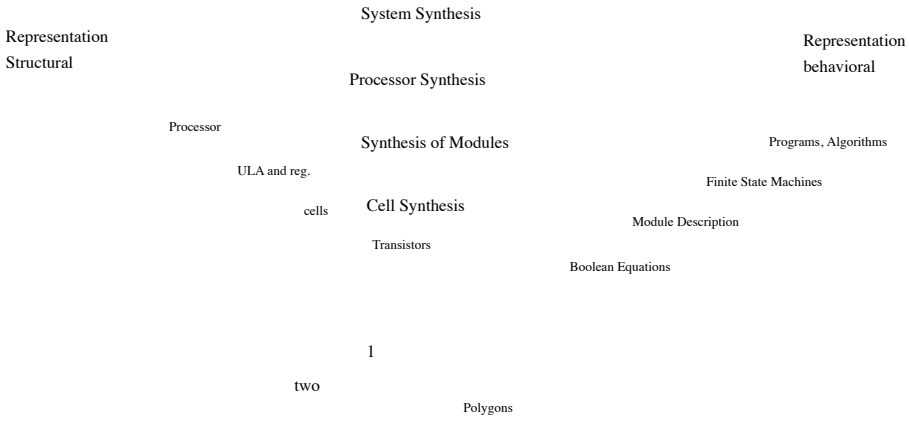
Silicon Compilation is the transformation of a high-level description of a given system.
ma in layout. By “high-level description” we mean any description that hides
user some level of detail. Usually the transformation process has
several steps, and it is customary to associate to each of these steps a corresponding compiler.
you. Therefore, we can define a logical compiler to transform a description into a
set of logic gates and flip-flops, or a microstructure compiler to transform
a given set of instructions in a set of registers, buses and functional units,

and so on.

In figure 2.2 we see a Y diagram, proposed by [gaj88, DG90], a classic representation of the compilation of silicon. In it, each of the axes represents the three domains of description: the behavioral, the structural and the physical (or geometric). along the axes the various levels of representation/description are represented. The level information becomes increasingly abstract as we move away from the center of the diagram. The tools of design are represented as arcs between the representation axes, and graphically denote what information the tool uses and what information is generated by the tool.

In the behavioral domain the interest is what the circuit does, not how it is constructed. Normally, the element is a black box containing inputs and outputs and a described function. the behavior of each output as a function of inputs and, eventually, of time.
The structural axis is the bridge between the behavioral and physical domains. The physical domain ignores as much as possible what the circuit should do, bringing the information to silicon structural design (physical/geometric design).

The transformations from the behavioral to the structural axis are called synthesis and the transformations from the structural to the physical axis are called physical implementations. Together, synthesis and implementation are called silicon compilation.



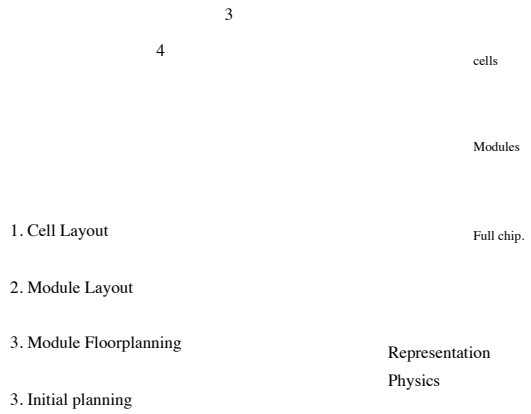


Figure 2.2: Y diagram.

In figure 2.2 are represented four compilers that would be needed in a system. of ideal silicon compilation. The system compiler decomposes programs and algorithms in a set of communicating processes. The processor compiler decomposes each process in a set of microarchitectural components or modules. the compilers of modules generate layouts of regular or irregular arrangement of cells and finally the cell compilers break cells down into gates, transistors and eventually into a set of polygons that represent the design in silicon of the synthesized element.

2.1.1.1 Cell Compilation

The cell compiler translates the behavioral description of the cell, usually a con- along with Boolean equations, in a mask layout. By cells we are referring to to single-bit functions, storage elements (memory bits), registers, microarchitectural components or even circuits with SSI or MSI complexity.

Cell layout synthesis is in general a very difficult task. It's a common practice simplify the problem by imposing a series of restrictions on the implementation architecture and on the

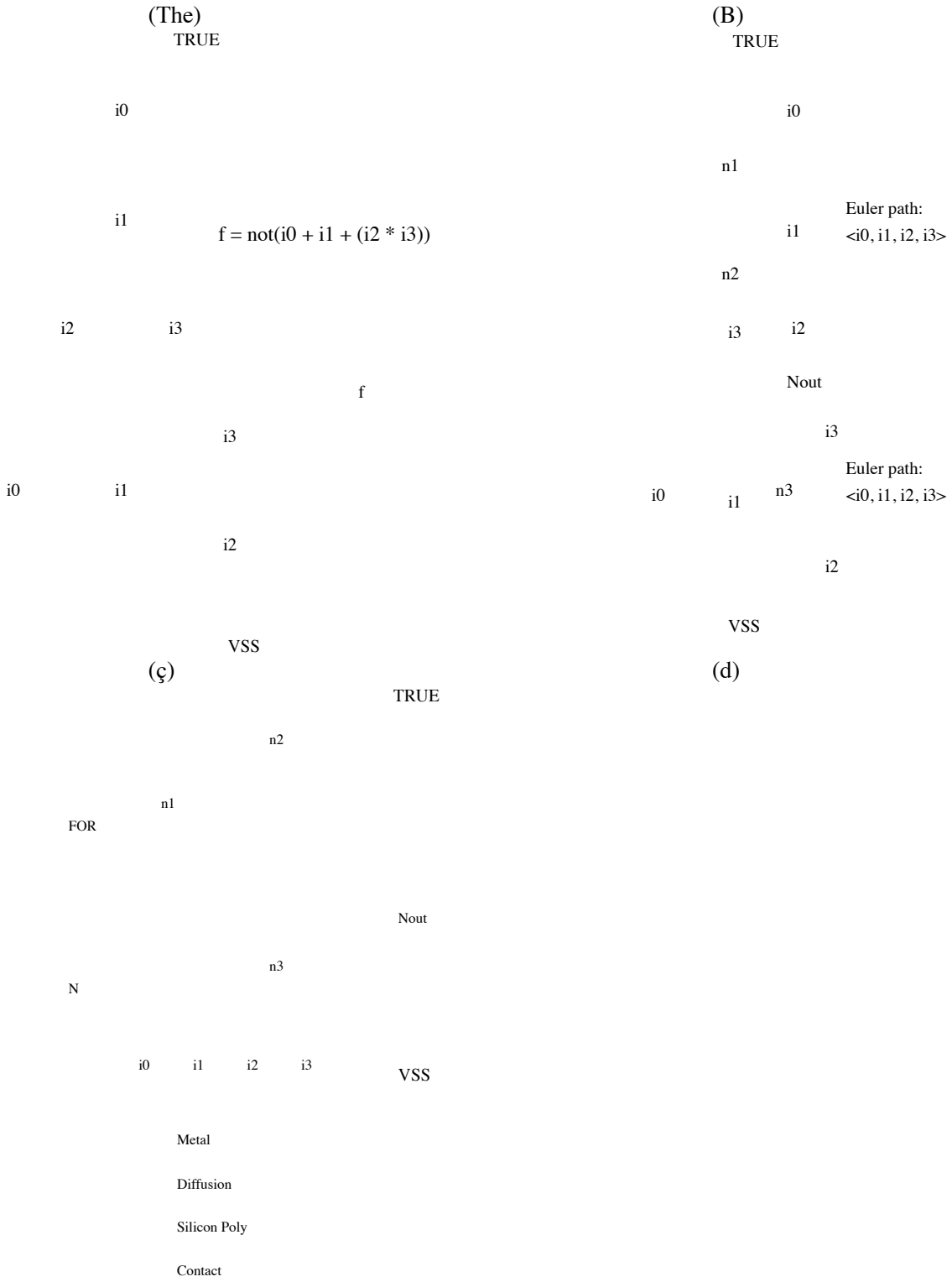


Figure 2.3: Example of CMOS cell synthesis with restricted layout to a single transistor P and one of N transistors. In (a) we see the functional specification of a complex CMOS cell and the corresponding scheme. In (b) the corresponding Euler Paths. In (c) we have the layout symbolic, and in (d) the actual layout.

dimensions of it. Obviously, this restricts the quality of the layout, but that's taken for granted. as a compromise between compiler complexity and design quality.

2.1.1.2 Compilation of Modules

The module compiler translates the behavior of a given module into a set of interconnected cells. The description of the module's behavior can be given from several ways like a set of Boolean equations or a feature specification which can be used to create a template for the module. Modules can be characterized by the type of logic they implement, can be random logic or modules matrix or regular.

Random logic modules are commonly used to implement functions of low hierarchy in the architecture and are generally implemented with PLAs, standard cells or custom logic. The behavioral specification of random logic generally does not produces an optimal layout, it is common to adopt the procedure of optimizing the description before proceeding with any kind of synthesis. The optimization strategy is composed of the following steps: minimization, factorization, mapping and optimization.

- Minimization: by minimization we seek to generate a minimum set of equations, in the sum-of-products form in order to minimize the number of transistors.
- Factorization: in factorization we can further reduce the number of transistors fa- making the equations in search of common elements. This technique however can decrease circuit performance as it introduces additional levels to the logic.
- Mapping: Some technologies have a pre-established set of cells to implement random logic. All subsequent implementation procedures

should take this set into account, so the intermediate descriptions

they must be “mapped” on the elements of that technology.

• Optimization: Finally a last optimization step with all elements

considered. Since there can be multiple types of logical gates in the library

of cells, we optimize the logic by replacing groups of ports with the corresponding ones.

more appropriate, through a procedure that involves certain rules and algorithms

Figure 2.4: Example of implementation of random logic in standard cells.

Figure 2.5: Example of regular module, 8-bit adder.

specials. A good reference on the multi-level minimization/optimization process can be found in [BHSV90].

Matrix or regular modules are commonly microarchitectural elements that de- they play specific functions. Examples of these modules are ROMs, RAMs, bank of registers, functional units, counters and data paths. A regular module is defined by a template and a set of cells that occupy predefined positions in this template. The specification of a regular module consists of the definition of the cells and their position in order to implement the desired functionality. Figure 2.5 is an example of this case, Figure 2.4 on the other hand presents a more generic version of logic implementation random that we call standard cells.

In a regular module the cell interface is made by simple juxtaposition, and the designer trust that any combination of cells is possible. Thus, no design rule is violated in the implementation of a given function. the circuit is

correct by construction.

In addition to the layout generation, module generators can also generate models for the interaction with other tools such as: symbols for schematic diagrams, models for different simulators, geometries for Floor planners, etc.

2.1.1.3 Processor Compilation

Processor compilation is the name of the process that translates a behavioral description abstract of a processing element in a set of architectural elements such as: regis- tractors, functional units and so on. The most used term for this class of tools is "High-Level Synthesis" and will be explored in more detail in section 2.1.2 .

This behavioral description consists of a black box with input channels and output, and a transformation function from inputs to outputs. Description that can be done, in principle, by any programming language. however for most of the languages, it is necessary to adopt a simplified model since these languages do not have the normally necessary notions of time, delay, performance or connectivity for a description of hardware. Only the order of execution is specified using the language control structures and sequential instructions. Description languages of Specific hardware for simulation and documentation of digital systems were conceived, they can also be used for synthesis purposes; however, we will see in section 2.1.3 that not always a suitable semantics for simulation is also suitable for synthesis of hardware.

In order to simplify the synthesis process, most processor compilers assume well-behaved and defined hardware architectures with synthesis target. One common model can be seen in figure 2.6 . In it, the system is considered as a set of connected processing elements (PE, Processing Elements). Each PE consists of a control unit (CU, Control Unit), and a datapath (DP). the control units are generically implemented as finite state machines, thus containing a internal state storage register and future state generation logic, control signals and communication with other PEs. Datapaths in turn consist of uni-storage facilities and functional units that are connected via buses. Storage elements can be registers, counters, register bank.

pains and memories; while, functional units are ALUs, shift registers, multipliers and so on.

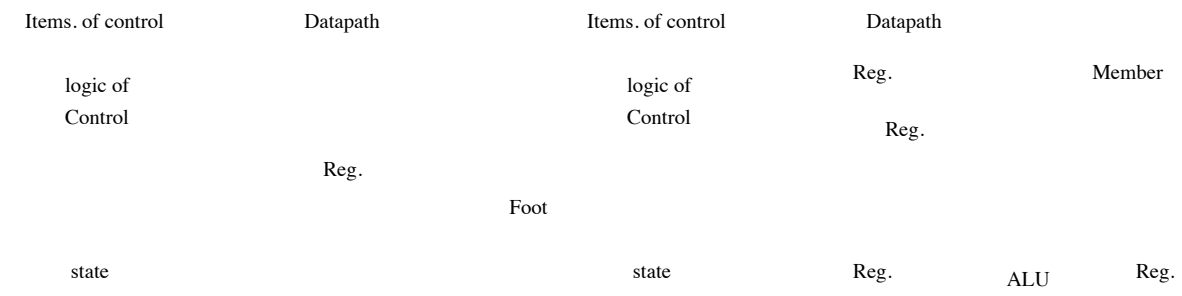


Figure 2.6: Processing Elements Model, PEs.

The compilation (or synthesis) process consists of mapping the language constructs. behavioral in architectural components. This process is generically called binding. Variables are allocated in storage elements (registers) or co- nexion (signals or buses), is the register binding. Operators, in turn, become functional units, is the unit binding. When variables are modified, you must establish connections to link registers to functional units to bring the variable to the unit corresponding and take it back to the appropriate record, ie the connection binding. IT'S I also need to convert the description language control structures – like loops and deviations – in sequential operations of the control units in the synthesized hardware. Each operation must in turn be associated with a time step or state. This process is called state binding.

Allocation of architectural elements does not cover the full range of generation possibilities hardware since many different frameworks can be designed by implementing the same functionality. The choice of which structure will be used is given by the constraints characteristics: performance, area and power consumption. Based on these restrictions, the process resource allocation creates an intermediate representation called a flow graph control/data flow graph, which eliminates unnecessary lin- input language, and can be easily manipulated to establish more structure. appropriate. We will see more details of this matter in section 2.1.2 .

2.1.1.4 Project System

An ideal design system based on silicon compilation combines the processes presented. presented in the previous sections, that is, it must use the processor compilers (synthesis level) and module and cell compilers. The user describes the circuit on a line. high-level graphics and lets the system take care of automatically generating the layout. In figure 2.7, a design system is presented as proposed by [DG90, gaj88].

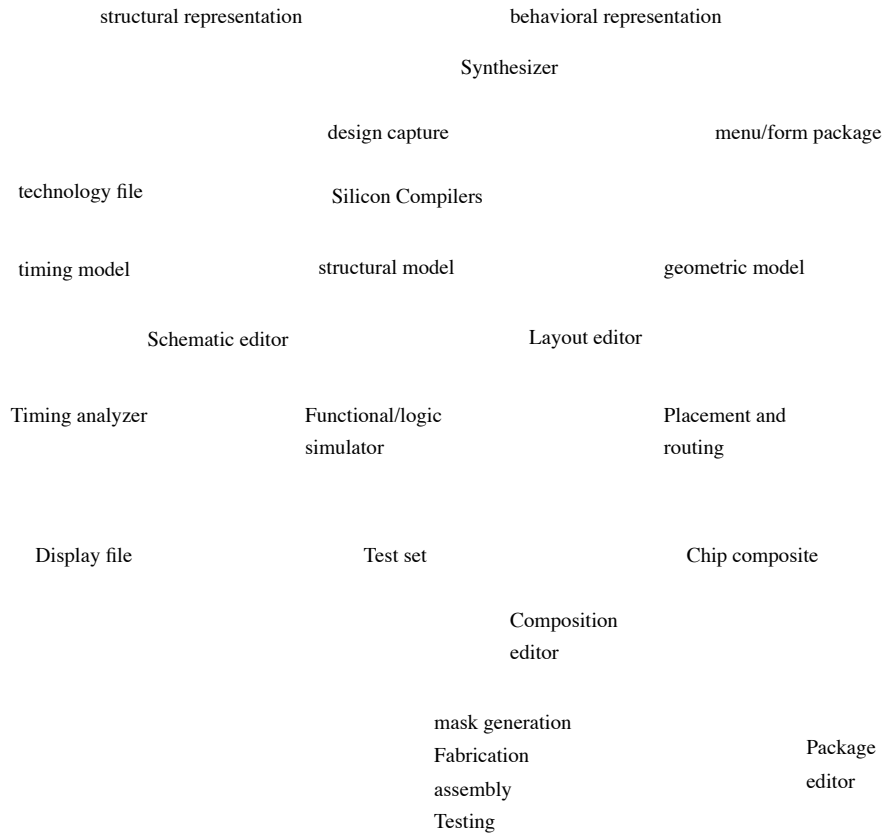


Figure 2.7: Design of systems based on Silicon Compilation.

We can see from the figure that the system can be specified either at the behavioral level. as well as structural, the bridge is made in the synthesis stage. The technology block contains all information relevant to the manufacturing process and design rules, used to generate the layout. In addition to the layout, the silicon compiler also generates time models and models. logical for each element of the structure. Then placement and routing tools

take care of the final composition. We also see that in almost all stages they are present editors that provide interactive access to the activities of the compiler, allowing the user interfere in the final result of the process.

The project system must also comprise a set of support tools,

which usually belong to one of the following groups: verification tools, analysis and optimization.

- Verification: verification tools verify the correctness of the composition descriptions. structural and structural. They usually consist of simulators of various types such as functional, logical, electrical and fault. Another form of evidence-based verification mathematics – called formal verification will be covered in more detail in the section 2.1.4.

- Analysis: analysis tools are used to determine or at least estimate the overall quality of the synthesized layout. It is often timing analyzers that determine the delay between ports or elements. By surveying these delays we can determine critical paths that, in turn, determine performance system maximum. Another common analysis is the testability analysis. For this, the controllability and observability of the circuit are calculated. Controllability is a measure that determines how difficult it is to control a node in the circuit, while observability measures the difficulty of being observed. Based on these measurements it is possible to plan the test strategies for the system.

- Optimization: The layout produced by the silicon compilation is not always optimal, but it is possible to improve it without compromising the other design constraints through of the optimization tools. At the geometric level we can mention the tools of

compaction, PLA folders, transistor resizing and so on. At intermediate levels quality optimizers help the description transformation processes.

2.1.2 High-Level Synthesis

As we saw in section 2.1.1.3 , high-level synthesis has been given different names throughout the time. “Processor Build”, “Behavioral Synthesis” or “High Level”, all if refer to the process of mapping a behavioral description, made in a language description of hardware, in a structural description or interconnected mesh of elements architectural, also called RTL description † . We will see below some fundamentals of techniques used in this field of research.

2.1.2.1 Internal Representation

High Level Synthesis (HLS) systems are unable to operate directly over the hardware descriptions, instead they use a representation internal where the transformations that make the behavioral description more close to RTL. This representation is known as data flow and control graphs, CDFG - control and data flow graphs, and consist of a better structured representation of the parse-tree generated by the parsing step of the language.

CDFGs exist in many different forms. In [Ber02] we are given an example that we reproduced in figure 2.8, in which a representation that highlights the control of the data. This representation is simpler to be understood without much explanation. and it does not present disadvantages in relation to the combined representations. The flow graph of control (CFG - Control Flow Graphs) represents the sequencing of the described operations in the specification language, including reordering operations, loop expansion and

unfolding. The data flow graph, in turn, represents the inter-
pending between data, operations and values. Note that for each node in the flow graph of
data, there is a corresponding in the flow of control; however, the reverse is not true. You
nodes that represent language ends have no counterparts in DFG.

2.1.2.2 Basic Operations

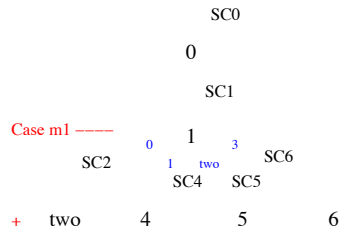
As stated in section 2.1.1.3 , the target model of the synthesis operations consists of unit-
processing conditions (PE), exemplified in Figure 2.6 . These units are nothing more.
which finite state machines associated with datapaths, or FSMD (Finite State Machine)
with Datapath). In principle, from the analysis of the data flow graphs, we can derive the
datapaths and, from the analysis of the control flows, we derive the control machine. Evident-
Mind you, real-life problems turn out to be much more complicated than we'd like.
that they were.

The high-level synthesis process must be based on design parameters, as per
example: cost, performance, power consumption and so on. These parameters should guide the
synthesis process, making it quite complicated, since the trivial solution rarely
satisfies project requirements. Thus, HLS systems adopted strategies for
implementation that allowed the exploration of these requirements during the synthesis process.

(The)

```
entity bde is
port ( clock: in bit;
        in1, in2, m1: in integer range 0 to 3;
        m2: in boolean;
        out1, out2, out3, out4: integer range 0 to 3 );
end bde;
Architecture behavior of bde is
begin
```

(B)



```
Process
    Variable A, B: integer range 0 to 3;
begin
    Wait until not clock'stable and clock='1';
    B := 0;                                  -- O0
    Case (m1) is                             -- O1
        when 0 => B := in1 + in2;            -- O2
                  A := A + B;                -- O3
        when 1 => B := 3;                    -- O4
        when 2 => A := in2;                  -- O5
        when 3 => A := 0;                    -- O6
                  B :=2;                     -- O7
    end Case;                                -- O8
    out1 <= A;                               -- O9
    If (m2)                                  -- O10
        then A := in1;                       -- O11
        else out2 <= B;                      -- O12
             out3 <= A + in1;                -- O13
    end If;                                  -- O14
    out4 <= A + B;                           -- O15
end Process;                                 -- O16
end behavior;
```

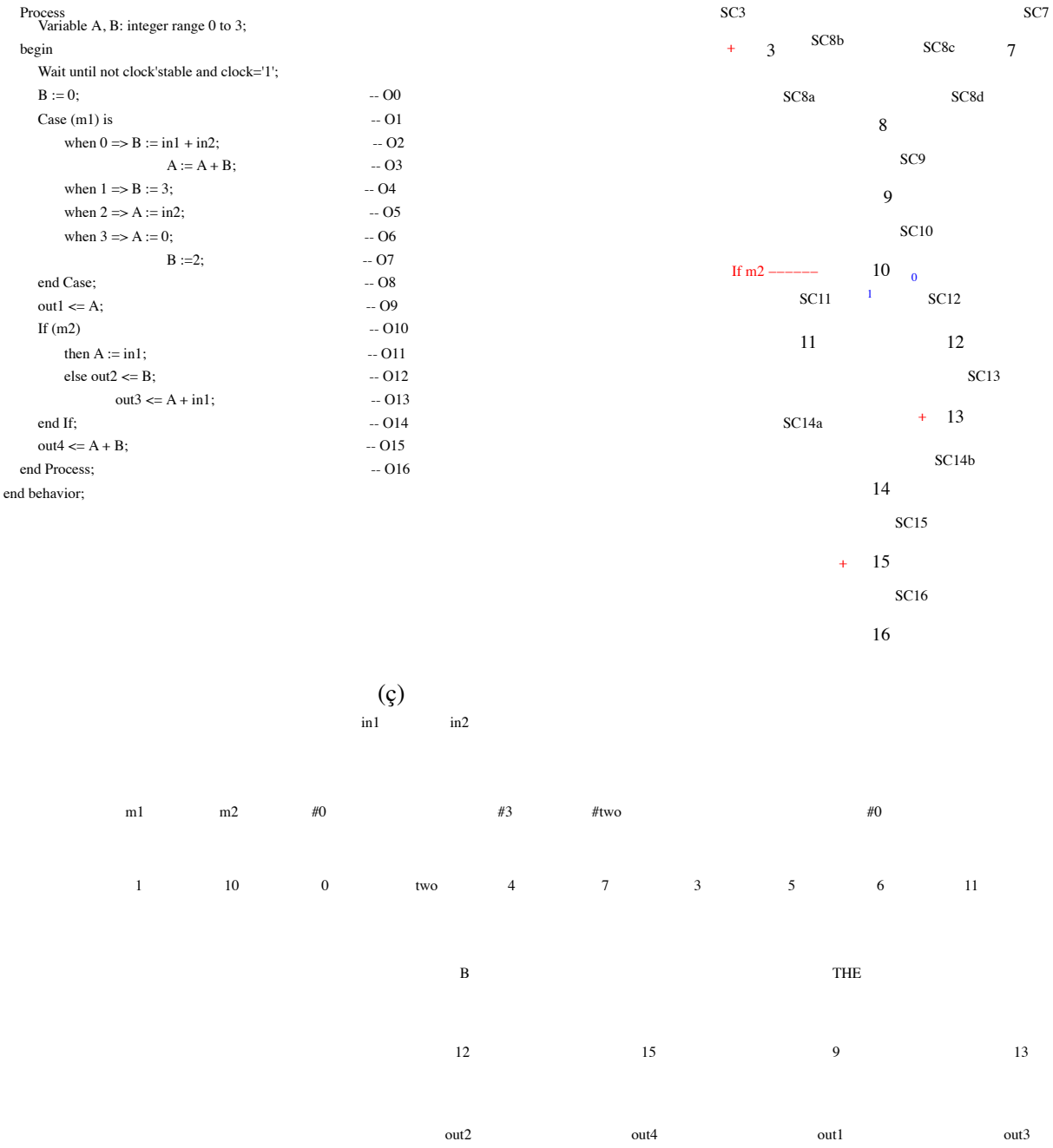


Figure 2.8: (a) Behavioral VHDL description. (b) Control flow graph. (c) Graph of data flow.

In general an HLS system can be divided into the following steps [Gov95]:

1. Compilation: in this phase the behavioral description is analyzed and translated into a intermediate representation (CDFGs) where future transformations will be applied.

2. Partitioning: at this stage the system is partitioned when necessary, for the purpose of reducing the problem to be analyzed and/or reducing the complexity.

3. Scheduling: the scheduling process consists of dividing the internal representation into control states in order to enable its future mapping in a machine of control states. This constitutes the most important phase of the synthesis. About her, can- if adopting strategies that take into account the design requirements (constraints). Scheduling algorithms can in turn be characterized as:

   • Basic: these are algorithms based more on common sense than on pro-requirements. said. They are also the starting point for several other strategies. Are they:

     (a) ASAP:"As Soon As Possible", determines the fastest allocation of operations per states as possible, hence the smallest number of control steps.

     (b) ALAP: "As Late As Possible", essentially opposite to the previous one producing the slowest "operations/states" allocation and the largest number of steps of control.

   • by Time Constraints: algorithms by time constraints are also key. called the "fixed number of steps approach". In general, they start from a solution ASAP/ALAP and then apply one of the following techniques:

     (a) Mathematical Programming: characterized by the ILP (Integer Linear) method Programming) that seeks to optimize the solution through searches, backtracking. This method presents very high complexity, long execution and solution excellent.

     (b) Constructive Heuristics: characterized by the FDS (Force Directed Sche- duling), this method seeks to equally distribute the operations of the same type. po on the set of control steps. It's the most popular – complexity

low, fast execution and almost optimal solution.

(c) Iterative Refinement: characterized by the IR (Iteractive Rescheduling) method,
it consists of modifying an allocation evaluating the cost of the new implementation.
The quality of the solution depends on the initial allocation, the complexity is high and
the execution time, average.

• by Resource Constraints: are indicated when there is a constraint in the silicon area.
Two methods stand out:

(a) List-Based Scheduling: This method maintains a priority list
for "ready" operations, ordered according to a priority function.
in. "Ready" operations are operations whose predecessors have already been allocated.
Once a list operation is allocated in a control step, many
others become "done" and must be included in the ordered list. O
The success of the scheduler will depend on the priority function adopted. The with-
The computational plexity of this method is high, but it produces results almost
always great. It is slower than the FDS method.

(b) Scheduling with Static Lists: despite dealing with lists, this method differs
of the previous one both in the means of allocation and in the way of ordering the lists.
The method starts with an ASAP and an ALAP solution to obtain the smallest
and the largest number of control steps (LCS - Least Control Step and GCS -
Greatest Control Step, respectively), for each operation. The algorithm uses
then these designations to sort a single large list that will be used
as a guide for allocation. Every time the resource limit is reached,
the allocation proceeds to a later control step. This method is a
slightly faster than the previous one and also produces good results.

• Other Methods: In addition to the methods presented, there are several others that are not fall into the above categories. In particular, two methods stand out, are they:

(a) Simulated Annealing: by this method, the allocation schedule operation/es- is represented by a two-dimensional table of control steps by functional units, then the problem is converted to a placement, where the positions of the table are filled with the operations. Eats-

Using an initial solution, the algorithm iteratively modifies the solution calculating the modification cost. A modification is accepted, associated with a certain probability, so that the solution space can be searched. in order to reach the minimums. Although robust, this method consumes long runtimes.

(b) Path-Based Scheduling: path-based scheduling seeks to minimize the number of control steps needed to execute the critical path of the CDFG. All execution paths are extracted and allocated independently. At the end, the different paths are combined to generate the solution.

4. Allocation: this phase works in intimate relation with the scheduling phase, and consists of in the partitioning of the intermediate representation with respect to space (resources hardware), also known as spatial mapping. It also determines the clock scheme, memory hierarchy and pipeline style. to satisfy everyone these requirements, the allocation must determine the exact area and performance of the units. allocated. An approximation of cost and performance can be given by the number of functional units and by the number of control steps. Rather than seeking self-

mathematically across the entire design space, most HLS tools current ones allow some degree of user interaction, offering metrics to help the designer to make the best choice.

5. Binding: This phase associates each operation and memory access, in each step of control, to a hardware element. A given resource, as a functional unit, storage or interconnection can be shared by different operations since that are mutually exclusive[GR94]. The binding phase consists of three sub-tasks, according to the type of unit:

   - Storage Binding, binds variables to storage units. Units of storage can be of many types, including registers, bank, registers and/or memory. Two variables that exist in control steps different ones can share the same registrar. Two variables that are not accessed simultaneously can, in a given state, be associated with it. port or memory.

   - Functional Unit Binding, binds each operation, in a given control step, to a functional unit. A functional unit or pipeline stage can only perform a single operation per clock cycle.
   - Interconnection Binding, associates interconnection units as multiplexers or buses for each data transfer between ports, functional units or storage.

Although we list these tasks independently, they are intrinsically re-related, and must be performed concurrently to obtain the best

results.

6. Control Generation: The last step is the generation of the control unit that will follow-up. will create the operations specified by the behavioral description and control the units functional datapath registers and memories.

2.1.3 Hardware Description Languages

Hardware Description Languages (HDL-Hardware Description Languages) are programming genes used to model the functioning of some piece of hardware, in general, digital. Two fundamental aspects must be present in this type of language: a Abstract behavioral modeling and Hardware structural modeling. The modeling is important to facilitate the behavioral description for the purpose of without prejudice to the structural or implementation aspects of the component. Per on the other hand, structural modeling is necessary in the specification refinement phases. when the component is described at a higher level of detail. So it is essential that an HDL is able to describe several levels of abstraction during the entire phase of project.

The description of digital systems through programming languages dates back to the decade 60 when it was first used to describe the IBM SYSTEM/360 system [SS98]. Designers and students use a wide variety of graphic symbols or symbols. textual explanations to describe digital systems. The formality in this type of description is essential for the documentation of a project. The use of a programming language it has two major advantages: it is naturally formal and unambiguous and can produce specifics.

executable cations, that is, it can produce programs that illustrate the described functionality (simulations).

During the 1940s and 1950s, computer architecture was relatively simple and the description of these systems was sufficient through logic diagrams and Boolean equations. The situation, however, changed with the introduction of the SYSTEM/360 system because it it consisted of a large family of implementations under the same set of instructions. There was a need to abstract the programming model from the implementation. In 1965 it was published the formal description of the SYSTEM/360 architecture, which consisted of a set of APL programs organized in two sections, a central processing system and an entry and exit system. The concept of abstraction levels was being characterized and consequently representation. Obviously a long time passed before there was a consensus in this area, the RTL level† (Register Transfer Level) was not well established, although there are many works and research activities in order to establish a universal order for this type of representation.

Over the years, many notations have been created for the purpose of describing the intermediate or lower levels of abstraction and came to be known as languages Description of Hardware (HDL). These notations were used for the purpose of documenting. mentation, design and simulation of digital computer systems. However, most of these languages was used only by research groups and academic environments such as input notations for simulation, analysis and synthesis tools. In the mid 80's, the industrial world started to demand stricter standards of representation that could be shared by diverse development groups. It was the beginning of the languages of description of traditional hardware.

2.1.3.1 VHDL - VHSIC Hardware Description Language

In 1980, the United States Department of Defense (DoD) wanted to make the pro- digital circuit project was self-documenting, following a common methodology and that

† Register Transfer Level (RTL): A level of description of digital systems in which the behavior synchronous system is described in terms of data transfer between storage elements, and that may imply passing through combinatorial logics that represent a computation or logical operation or arithmetic. RTL modeling allows for hierarchical design, which consists of a structural description of other RTL models [Soc99].

could be reusable in future projects. A hardware description language was created in the program of very high speed integrated circuits (VHSIC-Very High Speed Integrated Circuits) of the DoD. This language came to be known as VHDL. 1983 scored the beginning of VHDL development through the joint efforts of IBM, Texas Ins-instruments and Intermetrics. The experience shared by these companies in languages of high level and top-down design helped to define the VHDL and a simulation system associate.

In 1987, the future of the language was secured by its publication as a standard by IEEE (Institute of Electrical and Electronics Engineers), under the designation IEEE Standard 1076. Additionally, the DoD now requires that all contracted digital electronic systems from that year on were described in VHDL. The F-22 tactical fighter was one of the first major US government programs to intensively use VHDL as a tool of design and integration. Different subcontractors provided components and subsystems for the design of the plane and the interface between them needed to be impeccable. O The success of this project was critical to establishing the VHDL and the top-down methodology of project [smi97].

In 1993, there was the first major revision of the VHDL language by the IEEE. continued to be the most recent in 2002 [Soc02]. There are currently several co-commercials containing simulators and logical synthesis tools adhering to the VHDL standard. Additionally, the IEEE published language expansion standards through packages specific as: interoperability model for multivalued simulation system, IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std logic 1164) [Soc93]; model of mathematical functions, IEEE Standard VHDL Mathematical Packages [Soc96]; logical synthesis models, IEEE Standard VHDL Synthesis Packages [Soc97]; and RTL level synthesis model, IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis [Soc99].

2.1.3.2 Verilog

Verilog was developed by Gateway Design Automation, a company specialized in CAE tool development founded in 81 and launched in 1983 under the name of Verilog Hardware Description Language or just “Verilog HDL”. It consisted of the specific

language and an associated simulator. In 1985, both language and simulator underwent several improvements, renamed "Verilog-XL".

Due to its great flexibility and simulation power, the Verilog-XL system has become quickly very popular. The great increase in the complexity of integrated circuits and the large number of components easily surpassed the capacity of the other systems. mulators of that time. Then, in 1987, Gateway sought to link Verilog-XL to ASICs project looking for foundries approval, as their product was able to simulate easily over 100,000 ports. Another start-up, Synopsys, starts using the Verilog language as a behavioral description of its synthesis system. At that time the IEEE launched the VHDL standard, which solidified the concept of top-down design using a high-level language. All this contributed to the acceptance of Verilog-XL [Smi97].

In 1989, Gateway was purchased by Cadence. A year later, Cadence divides the language and the simulator, releasing Verilog HDL to the public domain. this was done with the clear purpose of competing with VHDL, which was a non-proprietary language. In this At that time the “Open Verilog International” (OVI) was created with the purpose of controlling the futures language improvements. OVI is a consortium of Verilog users and providers.

Almost all foundries supported Verilog and used Verilog-XL as a tool of main simulation. In a survey published in 1993, 85% of the projects were designed and submitted to foundries in Verilog. In 1995, the IEEE adopted it as a standard under the name

of “IEEE Standard 1364” [Soc01].

2.1.3.3 Programming Languages for Hardware Description

The use of general purpose programming languages for describing digital systems is a very old practice, as mentioned in section 2.1.3.. The advantages of specifications executables are undeniable [Syn02]:

- An executable specification avoids inconsistencies and errors and ensures the accuracy of the specification. This is because, when creating an executable specification, it is essential that the program behaves exactly like the hardware being designed.
- The executable specification also avoids ambiguous interpretations of the specification by have been done using formal language.

- The executable specification helps to validate the functionality of the system before wants another implementation to start.
- The executable specification helps to create system performance models and to validate the final performance of the project.
- The tests used to validate the specification can be used later to also validate the hardware that is under development.

Despite their great success, languages like Verilog and VHDL don't always offer the degree of flexibility required by certain development groups or, on the other hand, are out of reach because they are generally tools of an intrinsically commercial character. al and very high cost. For these groups, languages like C/C++ [Syn02, GKL99 , TB92, ZG01] or Java [KR00 , HO97] always seemed interesting alternatives because they are

widely available and are common knowledge of many professionals. However, such languages are not prepared to describe hardware in a convenient way when projetista. We will describe below some requirements for a programming language general purpose can adequately describe a digital system.

In [GL97] we are shown that for a language to be useful to the designer, its description and its hardware specification must meet the following requirements:

- Must be able to model the hardware correctly and unambiguously both under the behavioral and structural point of view at various levels of abstraction many different.
- Simulate the modeled hardware with the rest of the system, which may even contain com- software components.
- Finally, efficiently synthesize the hardware using CAD tools available.

These combined requirements mean a great deal of difficulty in using languages for general purpose, mainly because in general there is no support for integration these languages to CAD systems and vice versa. Even so, this subject has dis- great fascination among academic and research circles, which continue to bet in this path and in the advantages presented by it.

Traditional hardware description languages (VHDL and Verilog), are for the designer just a description for some kind of simulator designed for this language. Per on the other hand, when we are using a general purpose language, the description is compiled, generating an executable program that, in principle, should behave like the circuit described. The modeling of these components is commonly done through systems

reactive. A reactive system is one that is continuously interacting with its environment. environment, so we model a given component as an endless process (infinite loop) that it continually reacts to the stimuli of its environment. However, these systems must meet certain requirements to satisfy hardware development needs.

These requirements fall into two main categories: the semantic requirements, essential for a correct and unambiguous modeling; and the pragmatic requirements, dictated by the practice of projects and other implementation problems.

- Semantic Requirements:

1. Abstraction. The language must offer an abstraction mechanism, that is, it must allow complex systems to be implemented from systems minors or components. The interface must be independent of the component to allow for progressive refinement as the project progresses.

2. Reactive programming. As said before, the components are better. represented by reactive systems or independent processes. However, the Hardware simulation also requires basic synchronization and handling mechanisms. exception handling. Constructs like: wait(condition), watching(condition or terminal) and disable(name), or equivalents are common examples of elements control of these mechanisms.

3. Determinism. At any level of abstraction the simulation must be predictable, that is, a sequence of inputs must always lead to the same result. about to leave.

4. Concurrency or parallelism. Language must offer the illusion of simulation. tanneity of processes in order to reflect the parallelism of the implementation.

5. Timing. The language must also offer some branding mechanism. tion of time since the functioning of digital systems are linked

to a certain progression in time. Many systems offer support and times.
logical rather than real time, this is acceptable since real values
can be easily estimated from these first. Timekeeping
is fundamental for performance estimation.

- Pragmatic requirements:

1. Types of data. Variety of data types is extremely desirable as
makes the behavioral description much easier. Complex and composite types, co-
from vectors to arrays, they increase the abstraction capacity; however they can
make the synthesis and verification difficult.

2. Abstract interface. In compiled systems, the interface abstraction allows
that the declaration is totally independent of the component's description. That
facilitates the compilation process and can be implemented easily through the
classes in object-oriented languages. This is a very common feature in
VHDL and Ada.

3. Communication model. The communication model establishes the form with
that the components interact. In a compiled system, it means a part
essential in the simulation process. Can be implemented through variables
shared or specific objects.

4. Time and clock model. As stated earlier, the marking of
time is essential for some aspects such as performance and efficiency, however,
more specific refinements such as synchronization are often needed.
with real events (real time) or more elaborate clock disciplines.

5. Design tools. An entire project methodology must be associated with
given the description language. Auxiliary tools for analysis and synthesis, front
ends to help the compilation process and other features that help in the
productivity increases, are normally required when working with
very complex systems.

6. Multi-value logic. Multi-value logic support is very useful in bus modeling, tri-state, contention and so on. These phenomena, very

common in digital circuits are important because they can determine characteristics. fundamental tics in addition to correct functioning such as power consumption and performance.

In a general-purpose language, these requirements are not satisfied. It is necessary the development of an entire library of objects, procedures and functions to implement mention each of these aspects. Therefore, we can say that the use of a language of programming for the description and specification of hardware consists in the use of a series of function libraries that represent some aspect of a circuit's operation. digital. In chapter 3, we will explore in more detail the implementation techniques of these functions in the elaboration of a language for the description of digital systems.

2.1.4 Formal Verification

In this section we will present the verification of hardware design through formal methods. Before that we need to understand why design verification is so difficult. to estimate the possibility of the occurrence of an error in a digital circuit we can use equation 2.1 , proposed by [Keu91].

$$\frac{\text{logic transistors}}{\text{chip}} \times \frac{\text{lines in design}}{\text{logic transistors}} \times \frac{\text{bugs}}{\text{lines in design}} = \frac{\text{bugs}}{\text{chip}} \quad (2.1)$$

In this equation the term “logic transistors” refers to random logic transistors, since the number of transistors of regular components like RAMs, ROMs etc, they are much larger in number and do not usually interfere much in this calculation. a manager

of a project wishing to estimate the possibility of error in the project could use the following values, for example: a project that has 40,000 logic transistors and whose flat HDL model (no hierarchy) when synthesized produce an average of 5 logic transistors per line of code. If you consider that, historically, during the development process an error to every 8000 lines of HDL code escapes any kind of verification, the manager would complete that at the end of the project the circuit would still present an undetected error.

This is a picture that only tends to get worse once the current size of the circuits has increased considerably, getting tens to hundreds of times larger than the example. Note that the only way to reduce the amount of errors is to improve the ratio of

errors per line of code in equation 2.1 , since the synthesis ratio tends to hold constant. This means improving the productivity of verification methods.

The development process consists of successive specification and implementation steps. implementation, and implementation in one step becomes specification in the next step [CMP91]. The verification guarantees the correctness of the hardware in each one of the steps, being able to be done through simulation or formal proof. Simulation is still the tool common verification tool in the industry [Kum98]. However, its effectiveness lies in the con- reliability of a set of stimuli that are applied at the input of the circuit under test, propagated by logic and observed in the output. The verification, if the behavior matches with the specification, it is usually done manually and may lead to errors in interpretation. On the other hand, the formal verification does not require any kind of stimulus of input and even so guarantees 100% correctness of the verified hardware. The main concept main of this type of verification is in the word “formal”, which means that the verification is done in the form of a mathematical proof rather than an experimental one, as in the case of simulation.

There are basically two types of formal verification in use by the industry today:

are the property verification tools (or model verification) and the tools equivalence check. When checking properties, the specification is converted to some appropriate kind of internal representation. Then the designer asks questions that reflect the correct functioning of the system, such as if the protocol of a bus generates contentions or if an interface responds properly to an access. THE tool then checks whether these properties are satisfied by the implementation.

Equivalence verification, on the other hand, requires two models: one for reference and one for be verified. The tool determines whether the two models are equivalent or not. In case negative, the tool is capable of generating a set of stimuli that can be used for diagnosis. The verification by equivalence can still be of two types: equivalence combinational and sequential equivalence. In combinatorial equivalence, the tool is able to handle very large circuits as long as they both have the same number of states, that is, that they are not very different from each other. On the other hand, the sequential verification tools can handle projects and verify equivalence behavioral at any level of abstraction. The disadvantage is that the circuits are considerably smaller than in the case of the combinatorial equivalence tool.

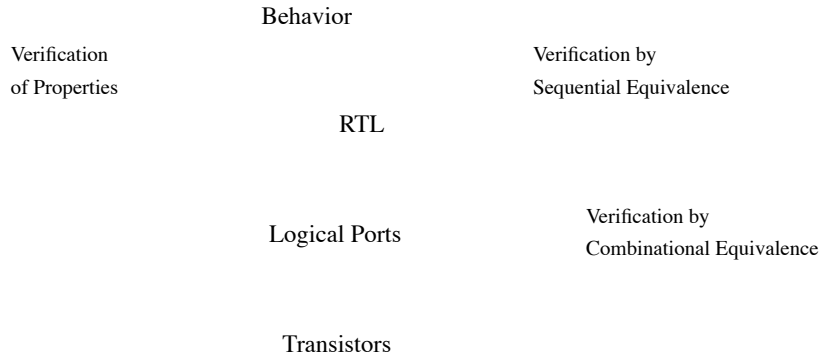


Figure 2.9: Design Flow using Formal Verification.

We see in figure 2.9 a design flow using formal verification. We can see that the property checking is usually used in the initial steps of the specification for certify that the concepts were correctly coded in the project description. One once a description is certified, it can be taken as a reference and used in the steps following in equivalence checks. Depending on the level of abstraction, you should use verification by sequential or combinational equivalence. Some of the tools of commercial verifications use Verilog and VHDL as input representations. then be easily integrated into a real design flow.

Despite all the advantages, formal verification is not the solution to all problems. It guarantees 100% correction, but this is a theoretical claim. If we consider that the tools are usually quite complex programs and in turn are not "formal-verified", we can conclude that the tools themselves are not infallible. Other reason is that the tools deal with models that are abstractions of reality, so they are intrinsically limited. Finally, the verification of properties depends on assumptions raised by the designer to determine the validity of a model. If any assumptions is false, the conclusion of the analysis will always be true. Another point to be considered is that formal verification is not a substitute for conventional simulation. The simulation still has a very important function, especially at higher abstraction levels when help the designer to understand and exercise problem.

2.1.5 Codesign Systems and Hardware/Software Specification

Historically, the evolution of CAD tools for designing integrated digital systems.
These have followed a cyclical pattern that we can characterize by the following aspects[FDK00]:
a methodology is adopted and well suited to a given problem; the problems become
nam more complex (usually complexity in terms of number of components);
new tools are adopted for specific problems; despite solving certain pro-
problems, these utilities end up harming productivity as a whole because the grade
of complexity becomes unsustainable; a new methodology emerges to deal with the new
set of premises; eventually a discontinuity is created in the market for the
users can migrate to the new techniques.

We observe this phenomenon when we analyze the evolution of specification techniques
and description of digital systems. In the beginning, designers used logical diagrams and diagrams.
chematics to describe the hardware. It was the most appropriate type of description because all
were used to thinking and understanding the problem in terms of logical gates and cir-
cares. Then with increasing complexity these diagrams were replaced by
hierarchical diagrams as the functionality could no longer be contained in a single sheet of
design. At the same time, we saw in section 2.1.3 that, in the area of computer projects
great progress was being made in terms of formal executable descriptions,
the first HDLs. So, in recent decades, the entire development methodology
digital involves some kind of hardware description language. But as it was predictable
other challenges emerged.

The first problem, like the schematic diagrams, was the degree of expression.
the description, or more technically speaking, at the “level of abstraction”. the advent of
first integrated circuits of many hundreds of thousands of components required dis-
long creations, time-consuming to be implemented and that gave rise to many errors of
codification. At the same time other types of implementations were emerging,
actions that demanded the reuse of previous projects and joint development
of hardware and software solutions.

Component reuse can be easily understood as it is a useful concept.
from the principles of digital systems engineering, it consists in expanding a given
system adding only the desired additional functionality while preserving what is already

it was working and proven to be tested. We did this first with the logic gates,
then with the MSI and LSI elements, now we're talking about VLSI systems and subsystems
complete. The problem is in the generation of descriptions for the components to be reused.
from generation to generation. Whenever there is a change in the type of description or
of the description language, it is necessary to redo the description of a system (or component),
so that it can be reused under the new methodology. sometimes it is possible
convert descriptions through specific programs; other times, I simply replaced
we use the complete description by another partial, ideal just to validate the interface between
the systems. Obviously this is a far from ideal procedure, however in the face of
market needs it is still widely used.

Software and hardware solutions are another class of application that has become very
popular in the mid-1980s and had a major development in the 1990s.
We believe that solutions involving software are quite flexible and can reach complexities.
extremely tall. With the proliferation of special purpose processors (and their ba-
prorating), this alternative has become a very viable solution to many problems.
However, the main disadvantage of this type of application is its limited performance.
the performance of the processor used and the algorithmic complexity. On the other hand, the
hardware solutions perform optimally in all applications. the problem here
is that for medium and high complexities the circuit size increases considerably,
also increasing the consequent problems, such as developmental difficulties,
juice and therefore cost. Another antagonistic aspect between these two solutions is that
the hardware has a definitive character, that is, once implemented there is no way to
change what has already been done, while in software these changes are always possible. Soon
the designers realized that these advantages/disadvantages could be combined to
meet the needs of market competitiveness.

Currently, these application classes are known by key terms, buzzwords or

"phrases" that call our attention to which aspect of the project, article, tool or methodology is of greatest interest. These most common terms are:

- System on a Chip, or SoC: refers to the possibility of implementing a system complete on a single silicon wafer, or Chip. Thanks to the integration capability today it is possible to integrate virtually any type of equipment or system into a

single chip. This can bring great benefits in terms of size, consumption of energy, weight and cost. The difficulties are exactly in the aspects raised above. secondly, deal with systems with millions of components and eventually balance hardware and software solutions.

- Embedded Systems, or embedded systems: refer to subsystems that generally make you part of larger systems and respond frequently to external events and interruptions that occur in real time‡. Some examples of this type of system are: protocol and bus controllers, robotic systems, answering machines, interface processors for electronic equipment (TVs, mini-systems, etc.), avionics control systems, etc. An important feature in this type of system is the balance between hardware and software as a solution.

- Hardware/Software Co-design: refers to joint software development and hardware to constitute the solution of a given system. Obviously this has a lot to see as embedded systems and SoCs.

We can see that these application areas are extremely interconnected, as they generally and embedded systems involve control operations of some kind and are implemented through microcontrollers, therefore, making use of Hardware/Software Co-

design. SoCs, in turn, can also encompass embedded systems or implement solutions that involve the integration of microprocessors in systems. The difference is in the idea that SoCs are, in general, much more complex systems that can reach some millions of components. The fact is that new development tools and methodologies need to deal with these new paradigms, both in terms of specification and in terms of verification. Only then will it be possible to enjoy the full potential of the technology currently available. These methodologies should enable development and software testing before the chip is produced, present verification mechanisms that ensure a high degree of operation reliability, present a high level of abstraction of

‡ Responding to real-time events is different from being a real-time system. By definition, a real time system or (Real Time System) is a general purpose computing system that offers high performance, capable of responding to real-time applications[Li96]. therefore having no relation to the subject of this work.

in order to reduce the specification time and decrease the probability of coding errors and allow the integration of modules developed by different teams without compromising the simulation, verification and problem tracking aspects.

2.1.5.1 Requirements for System Specification

System specification is a natural extension of hardware description. Measure that the hardware description was consolidated, the research was directed towards increase the level of abstraction of form and establish a new benchmark for the development of systems and tools. In [NG93, GV95 , Nar96] are raised some appropriate requirements for a systems development at a higher level of abstraction to traditional RTL and that in the future may characterize a methodology/language of

system-level description (System-Level Description Language - SLDL). we will present below are some of these requirements:

1. Hierarchy: Like HDLs, a development-oriented methodology at system level should offer the possibility of hierarchical description, both under the behavioral as well as the structural point of view. In the behavioral description, it is also necessary that there is the possibility of decomposing the behavior into "sub-behaviors", either through concurrent or sequential processes. also the possibility of activating or deactivating them at any time. in the hierarchy structural system is specified as a set of interconnections of components which in turn are also interconnections of even smaller components. He must there is the possibility of exchanging structural and behavioral components in any level of abstraction.

2. Competition: Hardware systems are more easily understood as they- autonomous/competitive entities that communicate with each other. Therefore the methodology must offer the possibility of representing this competition in the specification. This con- frequency may directly reflect the intrinsic parallelism of the hardware components or be characteristic of the constructions that make up the description, also known as statement-level parallelism.

3. Timing: here the term timing refers to the ability to account for passage of time of the real system for the purpose of performance measurement and even implementation modeling. Often the specification of a system is done. as a behavior over time; thus the modeling of time constitutes

a factor of great importance for the accuracy of the description as a whole.

4. Synchronization: If there is concurrency in a system, mechanisms are also needed
synchronization so that it is possible to exchange information and correct sequencing.
all processes. In my opinion, despite having been explicitly placed here,
the synchronization would be a consequence of the concurrency model adopted.

5. Inter-process Communication: This is a very important requirement when con-
we consider systems that involve the development of hardware and software in a way
competitor. The most common methods are: shared memory and passing
posts. At a more elementary level, we can also associate this requirement with the
way of implementing concurrency between processes. Also in this case, my
see, this would be a consequence of the tool's implementation.

6. Exception Management: refers to the ability to respond to external events.
us by the suspension or termination of the current action and the transfer of control to
another portion of the specification. It is also useful when considering development.
together with the software.

7. Programming Constructs: It is common for the designer to think about the behavior of
some components in algorithmic terms. In this case, programming constructs
common to programming languages are useful to help describe this type of
behavior.

8. State-based specification: A predominant feature of systems.
but embedded is functioning in terms of states. These systems work
by sequencing a set of predetermined states, in which some
action is taken or quantity evaluated. In general, they do not need algorithms with-
plexuses; however, this characteristic of functioning by states can make it difficult

Table 2.1: Evaluation of some hardware description languages.

Languages	Hierarchy	Competition	Actions	timing	Exceptions	transition from States	Synchronization	Communication	Detailing Structural
VHDL	declarative	at the level of Law Suit and commands	buildings of program bad	AF clause- HAVE command from WAIT	hard to represent sitting	representation laborious	events co- a few signs global	memory shared	Yes
HardwareC	declarative	at the level of Law Suit and commands	buildings of program bad	construction of Timing	hard to represent sitting	representation laborious	global ports message wait	memory share of sending posts	Yes
SDL	declarative	at the level of Law Suit	transitions of state	signs of Ti- meout	hard to represent sitting	diagram in States hierarchical	global signals	Shipping posts	Yes
StateCharts	hierarchy of states	in any level	states and transitions	timeout by bows and States	easy imple- mentation via arches	diagram in States hierarchical	Many resources	memory shared	Not
SpecCharts	hierarchy behave mental	behavioral in any level	buildings in program	AF clause- HAVE command from WAIT	easy imple- mentation via IT arcs	diagram of States	Many resources	memory share of sending posts	Yes

the specification if the development tool does not offer efficient means of represent it.

9. User Intervention for Synthesis: It is also desirable that it be available some kind of system control over the hardware that will be synthesized. The designer may want the system to follow some pre-established guidelines or even if it generates exactly what he is imagining. These controls can help the implementation process by reducing the universe of solution search of the tool.

In [NG93] is still shown a comparison of some description languages of hardware in view of the requirements presented above. The languages considered are: VHDL, already mentioned in section 2.1.3.1; o HardwareC, language-like description language C used for high-level synthesis by Stanford synthesis tools [KM90]; SDL (Specification and Description Language), standard established by the CCITT, widely used in telecommunication systems [MF00 , LHH02, ABS02]; StateCharts, developed as a extension of the concept of finite state machines for modeling reactive systems [DH89]; and SpecCharts, a system that combines the concepts of hierarchical state diagrams and competitors and VHDL statements [VNG95]. A summary of this comparison is shown in table 2.1 .

2.1.5.2 Hardware/Software Co-design

An aspect present in most digital systems is the presence of an element of processing and some software. Of course, we're not just referring to systems

of general purpose computing, even though they are part of a large portion of these systems. We are also including embedded/embedded systems and SoCs. Invariably, solutions involving software and hardware have become popular. rapidly over the last few decades and is a trend that should be consolidated in the future[dMG97].

Of the requirements presented in the previous section, some refer to the con- with systems involving hardware and software. However, the way it was presented, it may not have been clear this aspect in the development of systems. Therefore Therefore, in this section we will make some considerations about this type of development.

Hardware/Software Co-design (HSC) can encompass both relatively small systems. in very large systems, SoCs of millions of components. Take the example used by [Gup93], and reproduced in figure 2.10. It is a connected network controller. connected to a serial line and a memory. The purpose of controller is to receive and send data packets over a line using a given communication protocol, such as ethernet for example. The decision to map functionality into hardware or software is based on performance estimates and implementation cost of each of the parts. Traditional- mind, this partitioning is done in the early stages of the project and is mainly based in the experience of the designer(s). Hardware and software engineers work in each of its parts relatively independently or with minimal interaction. At disadvantages of this approach are predictable: not-so-optimal implementations, problems of integration of the parts, some uncertainty in the reliability of the implementation and high cost of implementation when considering the difficulties arising[Gup02]. This approach

can be called design-oriented.

As it could not be otherwise, it would not take long to try to use the techniques. synthesis cases/methodologies in order to improve development conditions in this area. This synthesis-oriented methodology, which was very successful in developing development of individual integrated circuits soon extended to the joint development of hardware/software.

The term Hardware/Software Co-design (HSC) is intended to characterize this type of development, however, it is necessary to distinguish the Co-development, which refer to the comprehensive development of hardware and software components; and Co-verification that re-

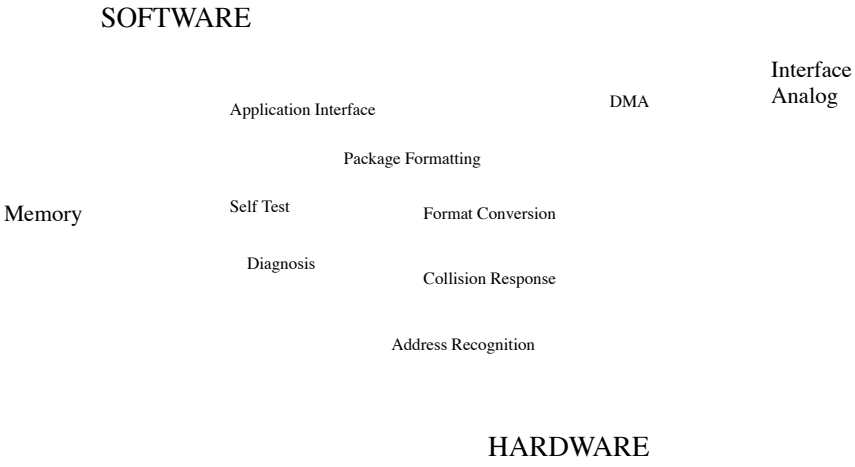


Figure 2.10: Specification of a system involving hardware and software

it hurts the joint verification of hardware and software, usually by joint simulation. HSC, on the other hand, it is the joint design of the system, at various levels of abstraction and partitioning. hardware and software components, including cost and constraints analysis.

Considering the example in figure 2.10, we can see in figure 2.11 the objectives of the HSC.

We are currently far from a consensus in this area, no HSC methodology proved to be effective for all applications. However, we can highlight the importance of the following steps in the success of this type of development[Li96]:

- Partitioning the behavioral description in hardware and software. This means define the boundary between these two domains, establishing precise evaluation metrics. tion of costs and commitments.
- Definition of an efficient communication interface between hardware and software for that the partitioning is not compromised by the implementation and that there is no future integration problems.
- Hardware synthesis and software generation. Ensuring speed, reliability, and greater autonomy of exploration of the design space.
- Performance analysis, which is necessary to assess whether the project adheres to the specifications performance, cost, etc.

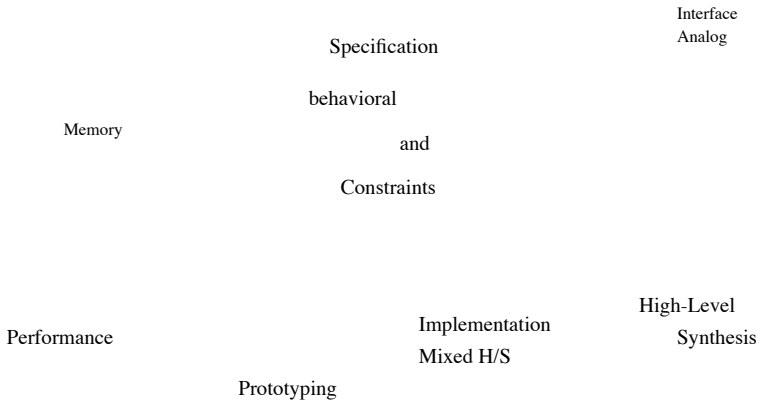


Figure 2.11: Hardware/Software Co-design Objectives

2.2 Self Programming Language

We have so far presented an overview of systems development tools that fit, in one way or another, into traditional design flows. THE in order to break the traditional line of thought, this work introduces concepts and techniques that go far beyond traditional techniques. The demonstration of these concepts and techniques only was made possible thanks to the use of the correct tool. In this case, the programming language Self. We will discuss in more detail in chapter 3 the reasons that led us to use of this language. We will see below a short introduction to the language and its main characteristics.

2.2.1 History

The philosophy behind the Self language dates back to the 1960s when the first ideas sobre objects began to take shape. At that time, the existing programming languages

were dedicated to the manipulation of processes or data. Object orientation emerged as a form of organization, in which data and processes were placed in a single structure. which came to be known as an object. Conceptually speaking, this organization can be done with any programming language, however, only the use of languages will allow you to explore the full potential of this methodology.

Only at the end of the 60s, it was possible to present the first language object-oriented, the Simula language. Although it has never become widely po- jumping, she was the archetype of several languages that followed. Simula was born from work by Ole-Johan Dahl and Hristen Nygaard [Sut99], who initially conceived a set of simulation procedures and a preprocessor for the ALGOL 60 language. mind, the work evolved into a compiler for a new language. This procedure ended up influencing other languages such as C++.

In the early 1970s, at the Xerox Palo Alto Research Center (PARC), a series began of projects that would dictate computing trends in later decades, among these projects was a programming language that promised to explore the concept of object-oriented programming like never before was Smalltalk. In this new language, which also included its own development system, was possible the inclusion of new classes, objects and behaviors with the system in operation. This language was responsible for several concepts with which we are familiar currently, such as graphical environments, windows, bytecodes, class hierarchy, etc. the de- Smalltalk development continued consistently with versions released every two from 1972 to 1978. The latest version of Xerox was released to the public domain at the beginning of the 1980s, Smalltalk-80 [GR83].

To date, many object-oriented languages have been released and played some role in the development of computer systems. However, we can say that the origin then objects has always been polarized in these two large branches, represented by the Smalltalk and Simula languages. Simulates, representing the traditional method of development. software development, and Smalltalk representing object-orientation in its aspects purer and more uniform. Without a doubt, the first group is by far the most popular, only because it performs better. The second group, on the other hand, is by far the one that implements higher-level concepts and can make the development task

easier and more enjoyable software use. Many works were prepared to solve the Smalltalk performance issues, some in hardware, some in software, getting quite satisfactory results. However, Smalltalk never came to threaten homeowners. traditional systems perhaps because it cannot lose the system label experimental or academic.

In 1986, David Ungar to Randall B. Smith, then also working at Xerox PARC, developed a new object-oriented, prototype-based, typed language. dynamics, it was called Self [ABC +00]. Self was conceived as an alternative to the lin- Smalltalk, Self seeks to maximize programmer productivity through a exploratory programming environment, keeping the language simple and pure, without however reduce its expressiveness and malleability.

2.2.2 Basic Principles of Language

Self is a pure object-oriented language, that is, all data are objects, all computation is done by receiving and sending messages. self merges the concepts of state and behavior, for example, invoking a method and accessing variables are indistinguishable, the object sending the message has no way of knowing whether its implement- tation is an access or a method, therefore, all code is independent of representation, that is, the same code can be reused by objects with structures totally different; as long as these objects correctly implement the protocol of expected messages. In other words, Self fully supports type abstraction of data. Only the object's interface is visible and all implementation details as structure, size and etc. are conveniently hidden. Therefore, as main attributes we can highlight:

• Self works with dynamic types, ie, unlike other pro-

gramming does not require the declaration of variables and types. The only restriction is that

a given object somehow implements the message being sent to

he. Determining the address of a given message during the execution of a

program is made dynamically at runtime, because during compilation it is

it is virtually impossible to determine which object will receive the message. Evi-

this feature makes the compilation work very difficult, however

techniques have been developed to overcome these difficulties [CUL89, CU90a].

• Self is based on prototypes rather than classes, this means that every object is self-

descriptive and can be modified, or “customized”, independently. This way,

it is possible to eliminate unintuitive concepts such as “meta-classes”, making the process

more natural development. In Self, objects inherit from other objects, des-

In this way it is possible to create a hierarchy of objects to factor behaviors and

common variables [UCCH91].

• Self has multiple inheritance, which can also change over time.

This allows the factorization not only of common behaviors to be possible.

also of "aggregable" behaviors, according to the convenience of the moment

[UCCH91, CUCH91].

• All control structures in Self are user-defined, that is, not

there are commands like if-then or while-do. In Self, the control structures

they are implemented with messages sent to special objects called blocks.

• Objects are automatically allocated and released from memory through a mem-

canism of “garbage collection”.

One of Self's main goals is to maximize programming productivity, to this the language offers some very interesting features, such as: Source-level semantics, the functioning of the system can always be explained in terms of code-source. Self programmer never has to confront cryptic messages like, “segmentation fault” or “arithmetic overflow”. These errors cannot be explained within the language definition therefore they are difficult to understand and to deal with. To avoid these problems, all Self primitives are safe against such flaws. direct execution semantics or interpreter semantics, that is, the system always behaves as if it were running directly to the source code, any modification becomes effective immediately. fast turn-around time: the programmer does not need to wait for long compilation periods, being update and compilation times are generally less than one second. Finally

as for efficiency: the programs developed in Self must present performance compatible with other more conventional languages.

Evidently, these attributes are difficult to be implemented, causing the Self is a better language for the programmer than for the machine itself. Fortunately, much of its development was devoted to improving development of techniques that allowed the realization of these attributes [CU91 , USCH92 , HU94a, hol94, HU94b].

2.2.3 The language

It is beyond the scope of this paper to present a comprehensive treatise on Self, however,

some fundamental notions will be very useful for a good understanding of the implementation. which we will present. Particularly [Cha92] summarizes the most common features. features of the language, and we find it interesting to reproduce them in this section. This will serve as basis for the discussions that will be made in the next chapters.

2.2.3.1 Object Model

Objects in Self consist of a set of pointers called slots, which are references to other objects. Some of these slots can be designated as parents or "parents" of that object. Objects can also have code associated with them, in these cases we call this object "method". The usual way to create an object in Self is by cloning another existing object called a "prototype".

In figure 2.12 we have an example of the object model present in Self, in it we see two objects point "A" and point "B" that represent Cartesian coordinates of a space two-dimensional. Each of these points has five slots: the parent*, which references the point traits object, parent § of point objects; the x and y slots that contain references to "integer" type objects and x: and y: slots that make references to the primitive function of attribution¶, represented in the figure by the left arrow (<-).

The point traits object, in turn, contains its own parent* slot, which points to another object not represented in the figure. The print and + slots are method-objects (which

§ In Self a slot is designated parent when the asterisk character "*" is added to the end of the name.

¶ We will detail the primitive functions in section 2.2.3.7.

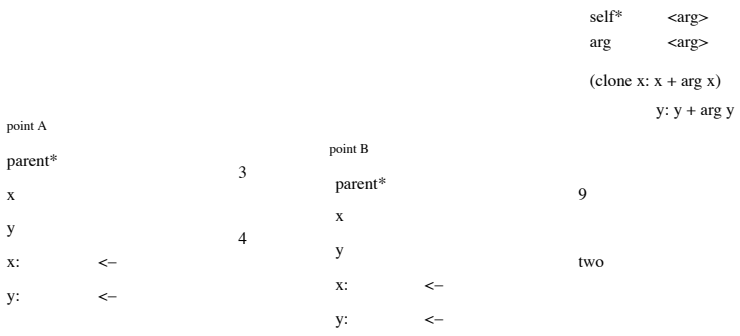


Figure 2.12: Self object model.

contains methods). A method differs from an ordinary object in that it has code associated with the even beyond the common slots. Each method has an implicit parent called self* that it also works as an argument and eventually ordinary arguments, as in the case of the method + which also takes the arg argument.

Self also has other types of objects: object arrays and byte arrays. To the Unlike normal objects, in arrays, elements (slots) are accessed not by a name, but by a number that works as an index. In byte arrays the elements are integers from 0 to 255, and are used in a more compact representation suitable for the interaction as elements external to the system. Primitive functions take care of your manipulation more efficiently.

2.2.3.2 Object Syntax

The programmer can describe an object in Self in a textual way simply by listing its slots and code in parentheses. Slots are listed between vertical bars placed at the beginning of the object description separated by a period “.”; the code is listed below through expressions also separated by a period. Each of these elements, slots or expressions, can be omitted. An example of the description of point “A” can be seen below:

```
(|
    parent* = traits point. "points to the traits point object"
    x <- 3. "left arrow (<-) indicates a modifiable slot"
    y <-4.
|)
```

A slot declaration is composed of three elements: the slot name, an "=" sign or "<-" is an expression for calculating the content of the slot, the last two being optional. If the statement omits the expression and the sign, the slot is created as modifiable and its value points to the nil object. The "=" sign indicates that the slot is constant and its value is the determined by the following expression, the sign "<-" creates a modifiable slot and additionally an almost homonymous slot that points to the assignment primitive. The name of this slot additional is formed by the modifiable slot name plus ":". Let's see below what the textual description of the traits point object.

```
(|
    parent* = ... "expression pointing to parent of traits point"
    print = ( x print.
                    '@' print.
                    y print ).
    + = (| :arg |
            (clone x: x + arg x)
                y: y + arg y ).
|)
```

The print and + methods were defined directly as contents of the respective slots. you. Argument slots are prefixed by a "colon (:)" and may or may not be initialized. Self still allows arguments to be declared as part of the name. of the slot. This feature allows the code to be clearer and more self-explanatory. In this case, the slot + statement would look like this:

```
+ arg = ( (clone x: x + arg x) y: y + arg y).
```

We see that the method declaration is similar to the object declaration, being the only one difference the inclusion of codes.

2.2.3.3 Message Evaluation

When a message is sent to an object in Self, the receiving object is scanned to check if it has a slot whose name matches the message received, in if so, the content of that slot is evaluated and the result is returned as a result.

of the message. For example, if we send message y to object point A, we will see that this message corresponds to a slot that refers to object 4, so as a result object 4 is returned. In this case, the message corresponds to an access to a variable

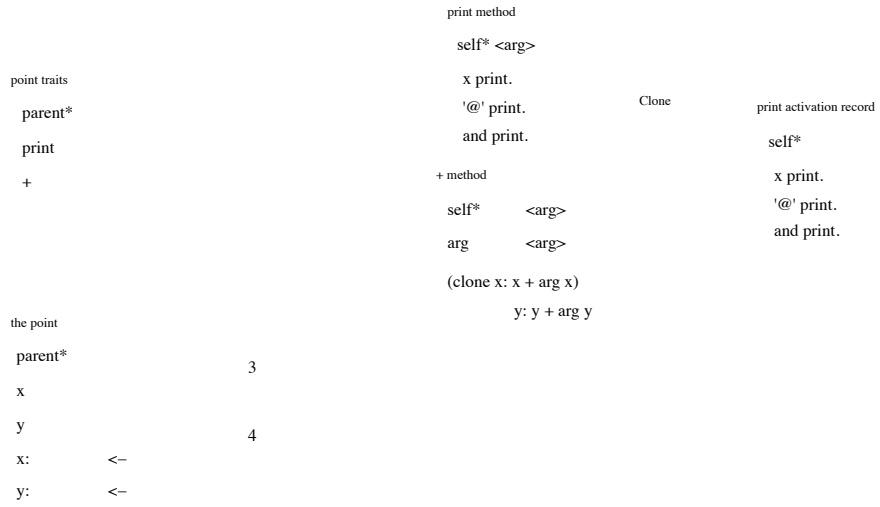


Figure 2.13: Evaluation of messages in Self.

from point A. If on the other hand, a suitable slot is not found, the search continues on object pointed to by the receiver's parent slot. For example, imagine now that the message were print. In this case, when analyzing point A, we verify that this message does not exist in the receiver, we then proceed to search on the receiver's parent, the point traits object.

The point traits object, on the other hand, implements the message through the print object method. The object-method is treated as an activation prototype. when the message is evaluated, the method-object is cloned and all processing takes place on this new object, called the “activation record”. The implicit self* slot is pointed to the receiver, which becomes the parent of the activation record method. Then the arguments (if any) they are replaced, and only then is the code executed. This process can be seen in the figure

2.13 .

Finally, when the message aims at modifying the content of a slot,
if special slots that reference the assignment primitive, as would be the case if the
message was x:7. In this case, the result of the message would be the replacement of the reference
to object 3 by reference to object 7. Note that in Self, even the parent slot can be
modified, as occurred in the process of activating a method-object. obviously, this
resource can be used for other purposes too, depending only on imagination.
of the programmer.

2.2.3.4 Message Syntax

The syntax of messages in Self is very similar to that of Smalltalk. Both define three
types of messages:

- Unary messages: unary messages are those that do not require argument.
  all. Syntactically, just write the name of the unary message after the name of the
  object or result of an expression (similar to postfix notation). Your
  name can be a sequence of letters or digits starting with a lowercase letter.
  it is not terminated by a “colon (:)”. Messages x, print are examples
  valid unary messages. Unary messages have maximum precedence between
  posts.

- Binary messages: Binary messages require only one argument. Feel-
  ethically, they appear between the receiver and the argument. are easily identified

for in general they are composed of a sequence of punctuation characters (with some but exceptions). The messages >, &&, = and + are valid examples of message names

binary. Binary messages have average precedence between messages and have none.

an associativity between them, that is, the programmer must make explicit the associativity

with parentheses when necessary. For example, the message “3 + 4 + 5” is full-

cool mind and means that 3 will be added to 4 resulting in 7, which in turn will be

added to 5 resulting in 12. The message “3 + 4 * 5”, also valid, will result in

7 multiplied by 5 resulting in 35, that is, the arguments are always evaluated from the

left to right. If we wanted the result to be 23 the message should-

would be: “3 + (4 * 5)”. In this way, the system interprets the message + being

sent to 3, having as argument another object (“4 * 5”).

• Keyword messages (Keyword): keyword messages require

rem one or more arguments, the name of these messages is formed by one or more

segments terminated by the character "colon (:)", each segment formed by a

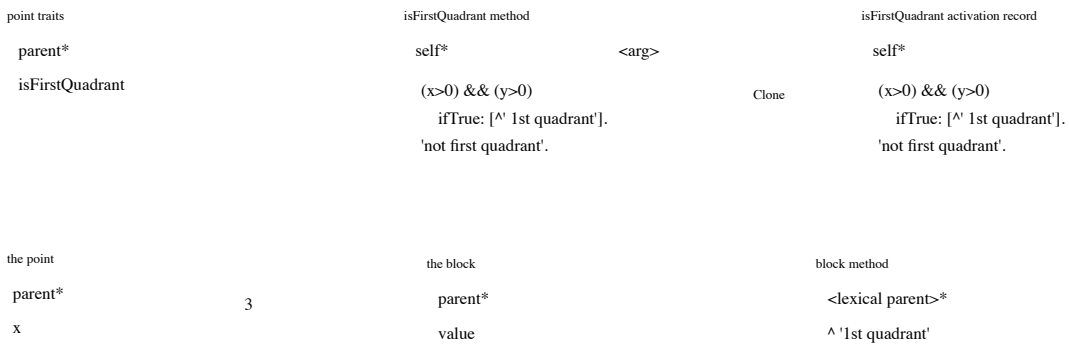
string of letters or numbers starting with a letter. The messages x:, ifTrue: and

ifTrue:False: are valid examples of keyword messages. In case of more

of an argument, they are placed between the segments of the message name,

for example, the message “page draw: aBox In: aPoint WithColor: myColor.”, the

Page 74



```
y                       4
x:          <-
y:          <-
```

Figure 2.14: Using Blocks to Create Control Structures.

message name is “draw:In:WithColor:” and uses aBox objects as arguments, aPoint and myColor. Keyword messages are used to make programs but more intelligible and more self-explanatory, in the example the meaning of the message: the message has been sent to the page object and requests that the aBox figure be drawn at the aPoint position with the color myColor. To simplify parsing and at the same time limit the need for parentheses, the first letter of the first segment of the message name must be lowercase and the first letter of the others segments must be capitalized. This type of message has the lowest precedence and the associativity is from right to left.

2.2.3.5 Blocks

Self allows the programmer to create their own program flow control structures. cessation through the use of special objects called blocks, or blocks. a block is a Self object that has the value slot that points to a special type of method-object. When the value message is sent to a block, it is evaluated as a "child" object the activation record from where the block was generated. Note that the block method does not have a parent self* slot like any other method-object, but a lexical parent that uses the object inheritance hierarchy to implement the lexical execution scope of the block. See the example in figure 2.14.

In this example, we have the isFirstQuadrant message added to the point traits object.

The message creates an activation record in the same way as explained above; in se-
Then, the execution of the expressions described by this method begins. This message verifies
if the point is in the first quadrant by testing whether both x and y are greater than zero. THE
evaluation of the expression "(x > 0) && (y > 0)" returns one of two possible types of objects,
true or false. The true object is an object of type boolean, which implements the message.
ifTrue: like: "ifTrue: aBlock = (aBlock value).". When the expression of x and y is
evaluates to true, the ifTrue message creates the block object represented in the figure and associates it.
o to the aBlock slot. The value message is then sent to it, as seen in
figure. If the expression of x and y evaluates to false, the block is not executed and the last one.
An evaluated expression returns as a result of the isFirstQuadrant message. A method
block can end with a non-local return by prefixing the return expression with the
character "^", thus the return returns not to the object that sent the value message,
but for the next higher lexical element. This feature has a similar effect to
C language return command.

2.2.3.6 Implicit Messages

Local variables and arguments are accessed in Self through implicit self messages,
that is, in the previous example when we evaluate the expression of x and y, in both cases these
messages do not have an explicit receiver and, therefore, the system interprets them as self x
or self y. This makes the search for the corresponding slot start from the object.
pointed to by the implicit self* slot of the method-object. As self* is the parent of activation
external record of execution at the moment, it is possible, through it, to have access to any
variable or method of it and its ancestors.

2.2.3.7 Primitive Functions

Most of the work in Self is performed by available primitive operations.
by the virtual machine. These operations are implemented below the language level.
Arithmetic operations, array access, input and output functions are examples of functions
primitives in Self. These functions are invoked in the same way that the
messages, except that the primitive functions are preceded by the underscore character
("_"). For example, the message "_IntAdd:", invokes the primitive sum of integers function.

In general, it is possible to pass to primitive functions a block with expressions for the case of function to show errors, just add the segment IfFail: and the desired error control block. For example:

3 _IntAdd: 'abc' IfFail: [| :code | ...]

, as the arguments of the primitive are not both properly formed the block that is given as the primitive's argument is invoked with the 'badTypeError' object as argument of the same.

2.2.4 Run Time Environment

Due to its nature, a Self system would have an implementation very similar to Small-talk, that is, a virtual machine that interprets the execution bytecodes and a file image that aggregated all system objects. However, such an implementation naive as that, little or nothing would contribute to the adherents of this new language. In In fact, Self still has these two components, image and virtual machine (VM). The difference The distrust is in the fact that the Self VM does not interpret the bytecodes, but compiles them directly for machine codes. Much of its development was spent on development. of these compilation techniques in order to achieve more satisfactory performances. for such a language. Currently, it is possible to offer performances from 2 to 4 times larger than commercial Smalltalk systems, which means nearly 50% of the performance of optimized C programs. The main techniques used in a Self system are:

- Customization: “customization” consists of extracting data type information programs that are exempt from these declarations, such as the Self. Many Sometimes, when we make a program, we use types and objects of a single type. Per

example, when we add two numbers we expect them to be formatted
compatible for the operation to be performed successfully. Likewise, it is possible,
analyzing the code, identifying the types of various objects and using them for the generation
of the compiled code. For each predicted type, a machine code "cus-
Tomized". Runtime tests are inserted to verify these predictions.
[Cha92].

- Type Analysis and Message Splitting: One of the Main System Problems
  as the Self is the low performance due to the fact that the subroutine calls
  cannot be determined until execution time when objects and addresses
  can be precisely determined. This is called dynamically-bound procedure
  calls. The best way to increase the performance of many subroutine calls
  is to compile them inline, inserting the code rather than actually making the call. En-
  however, without the type information this process is very difficult. Type analysis and
  the separation of messages extracts and manages the information better, maintaining a graph of
  annotated stream with type information of the compiled code. Iteratively, they are com-
  put variable types, allowing loops to be repeatedly recompiled
  and optimized for the most common type [CU90a].

- Type Feedback: is another technique that uses information obtained at runtime
  to eliminate the blurring of types. The information is stored and passed on to the
  compiler, which can now eliminate sending the message (called a subroutine)
  replacing it with an inline code. This technique can decrease sending frequency.
  of messages by a factor of 3.6 in relation to its non-use, resulting in a

performance increase of nearly 1.7 times [HU94a].

• Dynamic Recompilation: this feature basically allows the implementation of two previous ones, that is, when some important information is obtained the per- allows certain regions to be recompiled for performance more satisfying. However, all these optimizations and analysis increase the time to compilation in order to penalize the responsiveness of the system that should function as if it were interpreted. For an essentially interactive system, this might mean the your failure. However, Self's dynamic recompilation system uses compiled- different res to decrease the stall caused by the compilation. When a method is used for the first time, a non-optimized but very fast compiler is used. so that the compilation time is reduced. As the method becomes very used, your code is recompiled in a progressively optimized way so that your performance does not compromise the overall performance of the system [HU94b].

• Polymorphic Inline Caches: PICs allow the reduction of the overhead produced by highly polymorphic constructions and messages, that is, messages that can be sent to many different types of objects. PIC is another feature used in together with dynamic recompilation and consists of an extension of the inline caches of messages for multi-entry caches. Experience shows that less than 0.5% of messages have more than 4 different objects as receivers [Höl94], and that only 6% have between two and three receivers. Using PICs allows the compiler optimize even these messages. This optimization has represented an average of 27% increase in performance [HCU91].

In addition to the virtual machine, Self also works with a file that contains all the system objects, including the codes compiled and optimized by the system in the course of of usage, which is commonly called image. This image can be more easily experienced through its graphical interface, which is presented below.

2.2.5 Graphical User Interface

From the beginning, the initial goal of the Self language was to make the programming task something simple and natural. Historically, programming, which is nothing more than translating a sequence of operations in a form that is intelligible to the computer, is a extremely difficult task. Programmers are forced to deal with too many elements simultaneously, not to mention the interrelationship and dependence between them, requiring the limit the short-term memory capacity of programmers [CU90b].

Self's strategy for attacking this problem is, in addition to being a very pro- designed, to make an efficient implementation and a graphical interface that is firmly coupled with language. This interface allows you to access and inspect Self objects with combining object-based interface concepts with virtual reality. the interface emphasizes the objects of the problem domain rather than other elements that bypass the programmer's attention as seen in other window-based interfaces. In
In short, Self's graphical interface aims to make the interface invisible and objects and the world Real self elements.

2.2.5.1 Artificial Reality in Self

Self's artificial reality consists of Self objects inhabiting a simple bi-space.

dimensional. These objects are represented as a box that can be held by the mouse, as if it were a virtual hand, and dragged anywhere in the Self world. In figure 2.15, can be seen an example of these objects.

(The)

(B)

(ç) (d)

Figure 2.15: Example of Self objects.

The basic representation of any object in a Self world is similar to what appears in figure 2.15 (a). In it we see a block with a three-dimensional appearance to give us an impression. are of volume and solidity, and some knobs on their surface. These buttons perform some functions that are interesting for object manipulation. Their respective functions from right to left are: The “ ” button dismisses the object – when an object does not have more function or utility can be discarded by pressing this button. The “ ” button activates the evaluation window is actually a small text editor where messages can be edited and sent to the object to which it is associated. When this button is clicked, the object appearance changes as per figure 2.15 (c); Figure 2.15 (d) shows the composition. position of a message. The “ ” button invokes the parent object(s). The “ ” button invokes the comments, it is also a simple text editor where we can write comments about that object. By activating them, we have the appearance of figure 2.15(B). then we have

the object's name – not all objects have a name. In fact, the vast majority has no name, are created for temporary processing and discarded as soon as possible. followed. Finally, on the far left, we have a small triangle that represents the object's expansion state. An object is expanded when we can see the slots that compose it, as in figures 2.12 , 2.13 and 2.14 . In the previous examples, the object a shell is in its unexpanded state. We will have the opportunity to observe objects expanded later.

As in other modern graphic interfaces, in the Self's artificial reality, the cursor has the important function of representing the role of the virtual hand within the world of objects, the mouse buttons have functions also common to other interfaces. That simplifies the learning and adaptation process. The right mouse button is used to open a “meta-menu”, which is identical to all objects. This menu has functions like “dismiss”, “change color”, “resize” and “grab”. The “grab” function is useful because some objects special features like button and slider morphs redefine this original mouse function. The button left mouse exhibits natural grabbing, squeezing, and dragging behavior as in other interfaces. The middle button is used to open the object-specific menu under the cursor. Figure 2.16 shows an example of these actions and menus produced from of mouse.

In this artificial reality environment, objects can also be examined in your most basic composition. Take as an example an instance of the point object, remember that in Self there is no concept of classes and that objects can be created simply copying an object known as a “prototype”. In this example, we can write in the window evaluation of “the shell” the message “point copy” and press the button “Get it”. THE message will then be executed, that is, sending the copy message to the prototype object point will result in the creation of another object with similar characteristics to point and whose name will be “a point”, as we can see in figure 2.17(The).

As in the case of the “a shell” object, “a point” is initially in state

not expanded. We can invert this state by "clicking" on the small triangle on the left, causing the object's appearance to change to the expanded state, just like the small triangle. The appearance of “a point” when expanded is seen in figure 2.17(B). In it we can see the slots that make up “a point”: the parent* slot, which points to the

Figure 2.16: Mouse functions in Self's graphical environment.

object that “a point” inherits most messages/behavior; and the x and y slots that point to integers that represent the respective coordinates. Whenever possible, the content of the slot is shown to the right of the name; then there is a small button that, among other things, indicates the nature of the slot: parent* is indicated by the character “=” which means that the slot is a constant, that is, it cannot be modified. the x slots and y are represented by the character “:” which indicates that they are modifiable slots, that is, each one is actually two slots: one that returns the object's content and one that points to the slot modification primitive. A third type of button can be seen in figure 2.17(d), this one is characterized by a small square cut out resembling a window. This indicates that the slot points to a method object.

When the buttons to the right of the slots are pressed, the pointed objects are called.

(The)

(B)

(ç)

(d)

Figure 2.17: Creating and examining object a point.

to the graphical interface and the buttons jump from their original positions and cling to the called object, being linked to the respective slots by a line. This “dramatization” of the button action helps the user to understand what happened without being necessary verbal explanations or specific knowledge. These and other animation features will be briefly discussed in section 2.2.5.3. This behavior can be seen in Figure 2.17(ç).

When a button of an object-method is pressed, the behavior is different: A

a small text editor opens under the slot showing the code of the method it contains,
which can even be modified if necessary.

Note that the main feature of the Self's artificial reality is that the objective is aimed at
almost exclusively represent the objects unlike other graphical interfaces that
provide multiple tools to access them. This feature is called an interface
object-based and will be discussed below.

2.2.5.2 Object Based Interface

Traditional interfaces present themselves as a tool, a framework for in-
programmer's interactions with system objects. The interface is clearly identifiable.
There is no doubt that the programmer is in fact interacting with an interface that
translates its actions into operations on the objects of the problem domain instead of the pro-
programmer himself. In short, the interface is a barrier between the programmer and objects
of the system. The vast majority of systems operate this way, for example, on Smalltalk
[GR83], different tools are used for different activities: browsers to interact
with classes and methods, inspectors to show object instance variables, etc. That
UI style can be called “tool based” or “based
in activity".

The activity-based model emphasizes the manipulation of tools in the user interface.
user to allow access to the objects of the problem domain. On the model based
in objects, on the contrary, the objects of the problem domain themselves are manipulated in the
interface. Objects are made concrete in the interface and the programmer identifies them by representing them.
that they assume in the interface. This identification does not happen in methods based
in activity, because the same tool is used to give a view of different objects.
projects over time. The tools can actually be quite concrete for the

programmers, however, this is of little help since what really matters are the ob-
problem projects and not the tools. This ends up introducing a series of concepts and
specific knowledge, diverting the programmer's resources from the main objectives of the
its work, the objects of the problem domain. Force the user and keep in mind so many
different models interfere in the process of creating and solving problems [CU90b].

In the object-based model, object representations aim to identify them directly.
with the objects of the problem and, if this identification is good enough, the programmer
is led to think of the objects represented in the interface as the objects of the pro-
problem. From this point on, the behaviors displayed in the interface will contribute to the
programmer's mental model of programming language objects. So it's from
It is extremely important that the behavior of the objects in the interface are closely linked.
given to the semantics of language. Inconsistent behavior would only confuse
the programmer.

With that in mind, Self has carefully developed the language and interface.
and judicious in order to obtain a powerful development tool, helping the
programmer to better understand his work and the language he uses. How about
artificial reality, the ultimate goal is to make the user interface invisible, however
make the objects and the world Self as real as possible.

2.2.5.3 Animations

Self's graphical interface completely avoids the notion of windows, which means “to look”
for some things through some resource other than the eyes themselves, in favor of
a concrete artificial reality. The interface design shifts the focus from the "functionality of
tools” to that of “object personality”. In this sense, this concrete reality
must manifest itself in the behavior of objects in the interface, such as:

- Identity: in the Self world, an object that is unique must demonstrate this identity.
  when, for example, it is referenced multiplely. That is, instead of creating
  multiple representations of the same object, only one is allowed.

- Composition: a Self object is composed of a set of slots corresponding

`the messages the object can respond to. This is how objects in the world are presented

artificial Self.

• Uniformity: in Self everything is an object, even a simple integer.

Thus, in the Self world, all objects present themselves in a similar way.

and behave the same way.

• Connectivity: programs are networks of interconnected and interrelated objects

to exhibit a certain behavior or perform a certain operation.

dog. In the artificial world this interconnectivity and interdependence can be indicated

by links between objects. These links demonstrate their consistency when moving.

we connect objects and observe the connections following the movements performed.

• Referential: to understand the role of an object in a given operation is

need to know the sequence of messages in which it is involved. The world

artificial should allow the search for these cross-references as well as the search for the

object inheritance chain, in order to find the behavior

inherited, which parent, and so on.

The reaffirmation of this concrete reality is possible through the wide use of animation.

tions, which help the user to recognize and understand what is happening. most

user interfaces are based on static representations [CU93], events that occur during

during their operation are often reasons for fright and confusion. Obviously

users overcome these obstacles through experience. However, we cannot deny

that first contacts are often the worst; eventually they learn to compose

interface and begin to interact with it efficiently.

However, this effort of cognition of a new environment is against the propositions of the language and graphic environment of the Self. It is for this reason that any and all reactions in the artificial world of Self objects is accompanied by visual feedback that confirms the operation or reinforces the intention of the event so that there is no doubt for the programmer of the occurred. An example might be the one shown previously in figure 2.17(ç).

We said that when pressing the right button of the slot x the object it references is invoked for the graphical interface. This invocation is accompanied by an animation that moves the object from a position outside the view region to a position near the slot x. The movement is also accompanied by the connection line between the invoked object and the

slot. With that, there is no shadow of doubt about the deed and what happened. Likewise, when an object is dismissed it flies from its original position to somewhere outside the vision.

Thus, the environment hardly presents abrupt transitions that make it difficult to enter. trend of what happened. In Self, menus open smoothly, expansions are gradual and continuous, the motions are complete and not just contours. even when an object represented in the graphical interface receives a message it vibrates to demonstrate the operation. These small actions reaffirm the concrete character of the interface objects and maintains a closer relationship with the object model of the programmer problem. Unfortunately, in many other systems, actions like these still they are regarded as static modifications without value.

2.3 Conclusions

In this chapter, we took a look at the state of the art in development tools. digital systems, showing current trends and the lines of research that follow.

We can note that the emphasis is exclusively on solving specific problems, which makes the design flow a large cluster of scattered tools that are difficult to manage. development and difficult integration. We also show a oriented programming language to objects and based on prototypes and dynamic types (Self) which presents us with a for- innovative way of doing computation and that inspired the concepts we use for the realization of this work.

Chapter 3

Methodology

C digital systems, sometimes presenting the opposition in relation to the method
This section will present a new vis~ao development methodology
traditional. Many of the concepts presented here are a direct consequence of the language
of programming used in its implementation, the Self language. As was said before-
moreover, Self and Smalltalk are programming languages characterized by being
fully object-oriented and exploratory, that is, they are not just adaptations of the
concept of objects integrated into a common programming language like C++, but
on the other hand, they were conceived and implemented adopting the concept of object-message
for all elements of the language. Object orientation is already widely recognized
as a preferred paradigm for the development of complex systems by academics.
industrial, however, the exploratory character still appears in a timid or almost
nonexistent. We will see the importance of the exploratory factor in the proposed methodology. I believed
we believe that this factor can be the common denominator of new forms of development
in various fields of research.

In the next sections, we will see how the exploratory character in systems can be used.
hardware development themes. In section 3.1 we will talk about the flexibility factor.
As stated in section 2.1.5 , a system that intends to solve a large number of pro-
problems and surviving a long period of time needs to be flexible and adaptable to new
classes of problems as they arise. We'll see how this can be
done using an appropriate programming language. In section 3.5, we will talk how
the exploratory question can be addressed in order to captivate the operator/user in the task of

Abstract Concepts

prototypes

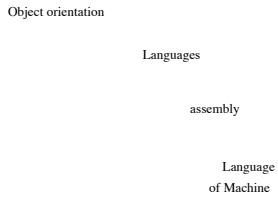


Figure 3.1: Man-Machine Approach through programming languages.

development. In section 3.6 he talks about how to implement the development “game”.

3.1 Programming Languages

Since the beginning of computing, different programming languages have been researched, in order to facilitate the arduous work of implementation and maintenance of systems operational and applications. Mathematically, we can say that under certain aspects all the languages are equivalent in that they can all be reduced, directly or indirectly, `the machine language of the processor that executes them. If we only considered the performance factor, we would all be programming in "assembly" that because it is closer the machine can be optimized practically to the limits of the hardware. However, this it does not just mean number of instructions or lines of code, but complexity. THE implementing complex or abstract algorithms in machine languages offers a level of complexity many times higher than that with which the human programmer is able to handle. If we were limited by this factor, we would certainly be in a degree of development equivalent to that of the beginning of the 70s and, certainly, we would not have personal computers available.

Programming languages emerged to free programmers' minds from machine execution details and allow the complexity to evolve into the field of

ideas (figure 3.1). The consequences of this release are undeniable. As well as other areas of knowledge, computer science explored regions and concepts as they were dictated by the needs of each era. Programming languages have come and gone. according to need or popularity. Interestingly, some languages have the ability to captivate users due to the degree of improvement they offer in relation to a previous generation, at other times due to the very concepts behind its implementation, as in the case of the FORTH [PW88] and Smalltalk [GR83 , PW88] languages. Other languages still had the good fortune to participate in a greater development, by demonstrate its usefulness and effectiveness as in the case of the C language, which acted decisively in the development of the UNIX system. Because UNIX has its own system of development, including its own C compiler, it became widely adopted as the preferred development language, quickly migrating to other systems. but operational and applications. This popularization ended up founding a culture and tradition very difficult to break. With the popularization of the ob- projects, the introduction of C++ occurred as a natural evolution of a path already then consolidated and secure.

Object orientation is certainly one of the most popular revolutions in computer science. expressive of the last decades, capable of establishing a new level of complexity to the computer systems. However, its adoption by conservative industry channels of software has limited its evolution and its wide use. The adaptation of this paradigm to an already existing language limited its power of action only to the scope of the project, leaving the program user alienated as to the potential of these new ideas. That means that for a simple program user, it doesn't matter if the program was developed by an object-oriented method or any other technique, as long as the program works, the result is pretty much the same. Current programmers certainly had their user phase, and as such, are impregnated with this old and retrograde view

of computer systems incapable of developing really innovative ideas/systems.
Evidently, the choice of this methodology can mean a reduction in the costs of
development and maintenance; however, this is not the only objective of the orientation to
objects. This distorted view ends up spreading the false idea that object-orientation is
a restricted knowledge of design to be known only by programmers and experts

developing, when in reality it consists of a powerful set of ideas that
should be learned as fundamentals of modern computer science.

In the early 1970s, when object-orientation made its debut, expectations
they were different. The potential of the new paradigm was well understood, however
the computational elements for its consolidation were lacking. Two streams had a
almost simultaneous development, one in Europe with the Simula language [Sut99] and
another in the US with Smalltalk [GR83]. Simula's lineage gave rise to C++, while
to Smalltalk, despite being way ahead of its time, took a back seat for a good deal of time.
time due to the poor performance of their implementations. The latter, however, does not
gave up none of the conceptual elements that make the object paradigm can
while its popular competitor (C++) has simplifications (deficiencies)
which end up limiting its evolutionary capacity as an instrument of development.
This work is far from being a criticism of the limitations of the C/C++ language.
well discussed in [Joy96]. The fact of having a consolidated position in the industry demonstrates the
its power and importance. However, we must keep in mind that if we want a great
evolution of the computational tools we need in our fields of research,
we must consider the use of alternatives that make this evolution possible.

Although it had limited development during the 1980s, Smalltalk
(ST) was responsible for many concepts that we currently use today

like graphical user interfaces and windowed environments, just to name a few. However, little commercial applications using the ST have been released, although academically a great deal of research was underway with the aim of improving their de-performance. Even with this relative stagnation, the TS remains current to this day. The emergence of the Self language in the early 90s marked a new stage for this line of implementation. Self adopts all that is purest in terms of object-oriented, inherited from its older brother the ST, in addition to adopting new concepts that make it simpler and more powerful than its predecessor, as shown in the section 2.2.2 .

Like the ST, Self brings the implementation objects to the user's domain, with this it is possible to extend the object-orientation beyond the design domains, making so that the “human agent” becomes part of the system as an active element and no longer

as a mere spectator. With these systems, it is possible to bring the development system for the application domain, the main objective of this work.

We saw in the previous chapter that, in Self, all elements are essentially objects, and that all are inserted in a context that allows them to be accessed, modified and reused at any time. The hierarchical composition structure, based on in prototypes and by delegating functions and behaviors, it allows objects to inherit directly from other objects and that are created from simple copies of objects prototypes. This makes the development proceed in a more linear, interactive way. and incremental, enabling better control throughout the development process. This peculiar implementation, combined with a carefully designed graphical interface of in order to be coherent with the philosophy of language, it allows the exploration of new concepts of use and development of programs and its possible application in the implementation of

a whole computer system.

For example, as we saw in section 2.2.5, the graphical interface of Self represents the objects in a concrete way, avoiding whenever possible the idea of a tool for a given goal. Their behavior reproduces the behavior of the objects of the material world, making the task of handling software objects so natural and intuitive as possible. The elimination of observation elements (windows and tools) allows that the interface is inhabited only by the objects of the problem domain, maintaining a a very close representation of the programmer's mental models. This gives more freedom to the mental process of creation because it drastically reduces the distractions and worries caused by elements that are part of the interface being used and that have nothing to do with the objects of the problem domain. In Self, we seek to eliminate the idea of a tool in favor of the idea of “personality” of objects.

Imagine, for example, that we have implemented a calculator in Self, such as that we find on our computers or pockets, where we type operations and we get the results in a viewer. Keeping in mind the personality of the objects and considering than in Self all computation is the result of sending a message to objects, that means that operations in the calculator are performed by sending messages to objects that represent the numbers. If we want to add a new type of number, (for example: complex numbers), it would be enough to create this object with all the associated arithmetic,

ted in a set of messages that keep a parallel with similar numbers real and ready. The new numbers would automatically work on the calculator. implemented without any need for reprogramming. This is the real potential of a language like the Self, the full use of object-oriented concepts, in this example: polymorphism and reuse.

3.1.1 Self Development

Another way to perceive the Self's potentialities is through its comparison with another conventional language. In [SM96] we can find a basic comparison. interesting, from where we take some important points that we will be presenting Next. Therefore, we will be comparing Self 4.1 language with C++, a based in prototypes and the other in classes, trying to highlight some important points.

3.1.1.1 Phases of Creating a Program

C++: Developing a C++ program generally involves these two phases:

- Editing the source program using a text editor, objects are created. using textual description through commands and statements that are available in the language.
- After the code is written, the programmer invokes the compile command. As the compilation takes place before the program is executed, it constitutes a totally static process. If the compilation ends successfully it is generated a file with the object code, which in turn must be "linked" (linked) with the functions of the standard C++ library. In the event of al- any problem, the source file must be modified and the entire process of compilation must be run again.

Self: Development in Self follows a different path. The programmer yes- It simply starts creating the necessary entities, objects and/or data structures using the Self programming environment or the graphical user interface, which as we have seen, it is an excellent tool for creating and visualizing objects. Huh- no command needs to be invoked to compile a sequence

of code. Instead, attributes, data, variables and states are created and set.
stored in memory immediately. Events are dynamically compiled against
as they are called for execution. Evidently this process takes place
completely transparent to the user. The programmer can modify
the code associated with an event or procedure without the need to recompile
the entire system.

3.1.1.2 Objects and Instances

C++: The class definition in C++ just creates a class template. This template
does not allocate memory space as no data or attributes currently exist.
An example of a class definition can be seen below:

```
Class phonebook {
        private:
char name[30];
char phone[15];
        public:
int insert(char *pname, char *pphone);
int get(char *pname, char *pphone);
int delete(char *pname);}
```

The creation of an instance of the class is done through a similar declaration
the declaration of a variable, with the class name in place of the variable's type,
as we see below:

```
phonebook PhoneBookInst;
```

At runtime, memory is dynamically allocated as the
instances are invoked and their values are known. This process is known
as “dynamic binding”. Memory is freed as soon as the program is finished.
of. If there is a need to release memory when the program is still
is working this must be done explicitly. To be able to use a die
object, it must have been previously defined in the source file; otherwise,
must be reedited and recompiled again.

Self: Objects can be created in Self as follows:

```
globals applications _AddSlotsIfAbsent: (| phonebook = () |)
```

```
phonebook _Define: (| parent* = traits clonable.
                                    name <- 'someone'.
                                    phone <- '555-5555' |
    insertName: n Phone: p = ( ... "Code for this method" ... ).
```

Page 94


```
    deleteName: n Phone: p = ( ... "Code of this other Method ...).
)
```

In this example, an object named phonebook is actually created as a slot. of the globals object. This allows this object to be referenced in the future. simply by your name. The phonebook object is in fact a fully-fledged object. operational and not just a template, occupying memory space, having functional attributes and methods. A new phonebook instance can be created through the message:

```
phonebook copy.
```

3.1.1.3 Data Types

C++: Like many other programming languages, C++ is based on checks. static typing, that is, whenever data is used, its type must be explicitly specified in the source code. At runtime the data is stored only in memory locations reserved for its respective type. It is not possible to change the data type dynamically.

Self: Self uses dynamic types, that is, there is no a priori verification of types. One variable can be defined and take any type of data, in addition the type can vary over time, for example:

```
globals _AddSlotsIfAbsent: (| test = () |)
_Define test: (| parent* = oddball traits.
                                myvar <- 3.142. |)
```

In this example the test object was created and it has the myvar slot, whose value

current is a real number. However, in Self it is perfectly legal to assign `to variable myvar not just another value, as a data of another type, with by example: test myvar: 'Hello World!'.

The message causes the character string to be stored in the myvar slot, and consequently that new messages referring to strings can be sent for that slot. This is the highest degree of polymorphism that can be achieved with an object-oriented language.

3.1.1.4 Language Commands

All types of program flow control or controlled loops are implemented. sent in Self through messages or events sent to objects, see 2.2.3.5 . In C++, on the other hand, they are implemented through fixed commands, similar to those present in non-object oriented languages. Certainly there is a cons- equivalent statement for every C++ control command, however Self has the advantage of being able to redefine any of these constructions or even implement others fully new as needed. This is the most that can be presented in terms of programming flexibility.

3.1.2 Conclusion

Self is a pure object-oriented, prototype-based, dynamic type language, can be considered an evolution of Smalltalk-80 [GR83] and whose goal is to maximize the programmer productivity through an exploratory programming environment. self

it does not have similar classes or concepts, it is purely object- and prototype-based, which they are objects considered as models. This means that each object can be changed individually, at any time, without interfering with the functioning of the others from the same family. This also eliminates unnecessary concepts like the "meta-classes" of the Smalltalk. Self allows for multiple inheritance, which is done directly between objects. O common behavior can be factored in by organizing the family tree of objects, optioptimizing the space used in the system. Another very interesting feature is unusual is variable inheritance. An object can have its inheritance relationship reset by over time, this feature proved to be very useful in the implementation of this work, as will be seen in chapter 4.

All computation in Self is done by sending messages. The majority of objects are not built-in constructions of the language, but written in the language itself. This means that almost everything in Self can be rewritten or redefined and, in consequence, all control structures, as stated in the previous section. Just like Smalltalk, Self works with a run-time environment called "Self world" or "Self world", composed of a virtual machine and a file (Snapshot) containing the work objects.

The virtual machine is responsible for system dynamics, the ”Snapshot” file works as a sea of "living" objects, that is, which can be inspected or modified anytime. When a new method is introduced into an object, its code is compiled into machine code and written to an appropriate position in the system. One very sophisticated compilation system ensures that the process is transparent to the user and guarantees a performance far superior to the traditional methods used for Smalltalk systems [HU94b, Hol94]. Self even offers a graphical system that explores very interesting ideas such as the concept of concrete objects [CUS95], widely used

In this job. In short, these reasons led us to consider the Self language to be the best.
candidate for the execution of this work.

3.2 User Oriented Application

Information technology or information technology reaches today a penetration in the various sectors of the society never seen before by another great technological advance. Despite this, and on the contrary from others, the dependence we have on computer services has an almost slaver. That is, we are practically in a master-slave relationship with respect to to the informatization elements and their suppliers. Users look on in wonder at the "advances" in information technology and let themselves be carried away by the promises of large corporations that offer products and services that fall far short of expectations and promised costs. In several technological areas, development and evolution lead to a cheaper product, increased reliability and productivity. In computing, this curiously does not and truth.

Ten years ago if we were looking for a state-of-the-art desktop device, we would find prices around $1500. If the option was for a notebook the cost would be $2500. If we look for equipment in these same classes today we will find pretty much the same prices. How is this possible? The "naive" defenders of this model may argue:

> "A desktop device today has the capability of a supercomputer ten years ago...", or,
>
> "Today an infinitely greater number of uses for the computer is possible

> people of what was there at that time."

This is all true, but the question is: “I really need a super-
computer to check my "emails", or write a memo, prepare a test,
keep my accounts up to date, listen to music, watch a movie?”. Personally,
I prefer listening to music on the stereo and movies on a big screen, and I believe that
I share this opinion with a large number of people. As for other activities,
a processor with architecture equivalent to the old 486, manufactured in a more up-to-date process.
would be perfectly capable of most of the tasks we are used to.
use our personal machines. If systems with the same specifications as those at that time
were manufactured with today's resources, we would certainly have personal computers to
prices possibly lower than current “palmtops”.

But real life is very different. The market is dominated by large corporations that
they are concerned only with the right profit. The consumer's opinion is not the most important element.
important. The more seductive and difficult to reproduce the product, the better,
for this will guarantee a “monopoly” that is difficult to be questioned. the attributes of
seduction can easily be achieved through a good marketing team. How much
To the difficulty of reproduction, the technological complexity itself takes care of most of the
her, helped by the computational model that seduces with the same advertisements.

3.3 Digital Systems Development Tools

Since the beginning of computer design, programming languages have been used
in the modeling of complex digital systems[SS98]. A large number of fo-
were designed to model the first systems, but each new development seemed
always a new experience, mainly due to the difficulty of using these nu-
such a specific task. There was a consensus that only one language especially
designed for the description of hardware could be of real use to the industry. Only-
In the mid-1980s, the first industrial language standards emerged.
Description of Hardware (HDL), with VHDL and Verilog. Soon, these patterns became
common in the main project flows. Unlike the schematic capture, the description
textual is much faster to be edited, unambiguous and able to describe the system in

different levels of abstraction. This caused levels of representation to be established appropriate for logical synthesis procedures (register transfer level - RTL), and levels of higher abstraction for representation and synthesis Following the HDLs, the tools of synthesis were the next big addendum to the design flows. First the tools of logical synthesis, then high-level synthesis, although the latter still does not enjoy the same popularity as the first.

Despite their great success in the industrial environment, standard HDLs have not been successful in fulfill all expectations, especially those of the academic community. Your difficulties problems in dealing with different data structures/types and the syntactic limitations in describing higher-level abstractions signaled the need for new types of description.
Many groups claim that the way is to use programming languages on purpose. general, as was done at the beginning, with a certain emphasis on the most popular languages, such as C/C++[Syn02 , TB92] or Java [KR00]. The idea is to use the power and flexibility of these languages to develop a subset (functions/objects) suitable for the descriptions of hardware. By standardizing these subsets, it would be possible to use the same structure of software development for systems design. Another important point: having been standardized these subsets of functions, the “reuse” and exchange of IPs would be guaranteed. Furthermore, as it is a general-purpose language, the integration in different flows of project would be practically guaranteed. Using object orientation to hide the complexity of the hardware models of the programmers/designers it would be possible to avoid the problems presented at the beginning of this practice.

The complexity of digital design has increased considerably in recent years. New classes of problems appear every day. Not to mention the decreasing time-to-market and the costs involved. The market is much more aggressive due to a new profile of consumer accustomed to technological innovations every year[ITR01]. The concern of

semiconductor industry has focused on problems arising from complexity
at the systems level, such as: “reuse”, verification and testing, project management, software
embedded, cost-based optimizations, and so on. Following these trends, the efforts of
means of research have been in increasing the productivity of aid tools (EDA)
and project flows, introducing high-level description models for digital and
new synthesis algorithms.

The result of this process is a great diversity of tools, models and interests
conflicts that is very difficult to keep manageable and up to date. Another serious problem is that
these tools were developed with computer science techniques aiming at a
very particular aspect of the problem and almost always without considering the forms of use
the same. These conditions, most of the time, do not match the reality of teams of
development. The designer's point of view has been systematically disregarded
when new methodologies are proposed. This state of affairs may not be intentional,
but the requirements of development teams that need to be trained have increased
in programming knowledge/concepts sometimes far beyond the scope of their
specializations. This ends up increasing the personnel cost even more.

Note that this is not a simple critique of computer science research in the field.
EDA tools. In fact they have made a remarkable contribution to the entire system of
project. The point is that research should pay more attention to the way in which the
tools with their users, their problems and limitations, and the way in which they are structured.
Data tures are processed and presented in the design flow. The triple of operations
“edit-compile-simulate” is not indicated when working with high levels of abs-
traction. At these levels, changes must be implemented and verified quickly.
way not to divert the designer's attention from the object of his work. despite all the

advances, the user's imagination still plays a determining factor in this “Game”. The tools must reflect this importance, as well as the entire project flow.

With all these points in mind, this paper proposes the foundation for what can become a new methodology for designing digital systems. A methodology that adopt the designer's point of view, where each tool assumes the human agent as a main source/guide to/from the project, can be achieved by applying the concept of Game (GLD) in the development of a common framework that integrates the most advanced tools, and also the more traditional ones; but that, above all, it is different in the how such tools interact with the human agent throughout the process of development.

3.4 Designer Oriented Development

In the previous section, we stated that the design support tools are developed under a very particular view of its creators. This can be easily seen when analyzed their operational mode. Take for example a hypothetical simulator. There are many types of simulation, the one we are interested in is the one that refers to to architectural and RTL simulations. In these cases, each hierarchical level corresponds to a set of primitives that constitute the basis of the simulation process [OHO98]. The simulates can also be of the Levelized Compiled Code (LCC) type or interpreted [Mau96]; or when the simulation inference type can be event-driven or cycle based. In all these cases, they share a common point: the (cultural) view of their servants.

res. Let's see how we use these simulators: type-independent, event-driven or cycle based, the compiled simulators start from a description of the circuit in terms of primitives, which is compiled generating a program that corresponds to the simulator for that circuit. This program is then executed as a function of inputs that match to the conditions under which the circuit must be verified. We can see here an operation pattern which, we must agree, makes a perfect parallel with how to develop programs, keeping the proper proportions.

This pattern is what appears in all program development cycles, or be "input-filter-result". “Inputs and results” correspond to files with func-special conditions, “filter” corresponds to a transformation agent that, operating on the input file, produces an output data sequence which is stored in “result. ted”. This can be observed in almost the entire universe of contemporary computing, being one of the foundations of UNIX systems. In the compilation, “input” corresponds to source file, the compiler to the “filter” and the executable program to the “result”. In the example p of the compiled simulator, we observe the same thing: “description-compile-simulator”, respectively. If we take an interpreted simulator as an example, we will see the same default: “description-simulator-output”. And so on. When we analyze others helper tools, we can identify in each case the above pattern, demonstrating that, however creative programmers try to be, they will still be tainted with a “modus operandi” that emerged more than 30 years ago and that does not correspond with the reality of users of your programs.

About 15 years ago, a more intense diffusion of graphical and “friendly” interfaces began. in computing systems. However, it seems that the programmers continued insensitive to the potential of this new technology. Maybe this happened in part by the technological limitations that were presented at the time, but the most likely is that they

still remain slaves of the old programming concepts. which, we believe,
it happens to this day. We can take as an example the "graphical" tools that
appeared at that time and are the foundation of many that we still use today.
We had the schematic editors, where we edited the circuit diagrams that were
under development. When it was necessary to do something really useful, such as by
example a verification (simulation), it was necessary to generate a netlist (“input”), apply it
to the simulator (“filter”) to obtain the results (“output”). And analyze it, taking these
results (“new input”) and apply it to a waveform generator (“filter”) so that
generated a graphical representation (“output”). That is, despite this great resource, the
programs have never stopped working that way. the traditional form of science
of computing:

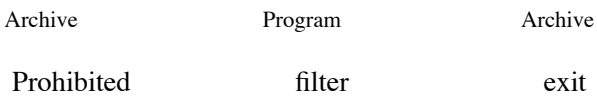


Figure 3.2: Traditional program operation scheme.

This form of operation has been shown to be effective in the last 40 years but has already shown
signs of tiredness over the past decade. This computing model emerged in a
a time when computers were machines practically inaccessible to common users
and that only top specialists could operate. At that time also the cost of
each equipment was so high that its existence could only be justified if its
utilization rate was 100%, 24 hours a day. Under these circumstances, it is fully
justified the above model, which, combined with the existing modest performances, allowed the
batch and background operations without operator intervention. However, what the
we have served in the last two decades was the appearance of machines for personal use, which
made it possible for the common user to access that strange equipment until then present only
mind in science fiction. However, the hopes of fiction remained only in fiction.
Despite the tremendous technological development, the vision of the programmers continued to immerse.

sa in the mainframe era. For the common user, programs are still difficult to be operated, not very intuitive and far from the reality of users.

The improvements we are currently witnessing are modest if we consider the real potential of the existing technology. The only drawback would be the interests of the corporations that dominate the sector. We will see in section 3.6, that only one change in implementation paradigms can open up a great possibility of development. development, without the inconveniences we have been pointing out so far.

The objective of this work refers mainly to the development methodology of digital systems in general, and it can be easily extended to other areas of knowledge. not linked to computer science but lacking computational help. The critic The main one concerns the operational mode of the tools currently available and the dis- conceptual tance between programs and users, which make the latter participate as mere observers in their fields of action.

The designer-oriented (DO) methodology consists of using a series of resources to approximate the application universe to the computational reality. This approximation is made using an object-oriented programming language (Self [US91], whose peculiar and advanced design allows the implementation of these features almost effortlessly. The program concept is diluted in a Self environment, what exists is only a set of objects that exhibit a dynamic among themselves, a behavior that translates into operations and processing. Self also features a graphical development environment[CUS95] which reinforces the concept of concrete and accessible objects, allowing programming be graphic and textual, depending on the convenience of the moment. This opens up a series of possibilities regarding the development of systems, the computational elements can be encapsulated in terms of elements common to the domain of specialization of the user, in order to facilitate its understanding and use. This can be graphically reinforced, in order to make development more pleasant and faster for the user. The second pillar of the methodology is the concept of Game applied to development, (GLD) “Game Like Development”. The Self language provides us with the computational model for the implementation

of methodology through the concept of concrete objects, the concept of Game, in turn, determines how applications should be presented in relation to the human agent, who does not more is considered as a separate element of the development process.

3.5 Game Concept in Development

In this section we will approach an important part of the methodology, which is the concept of Game applied to development (GLD, Game Like Development). To better understand the con-
concept of Game, we will borrow some concepts and observations from philosophy [Gad99] and also we will appeal to the common sense that each one of us possesses in relation to the subject. Game is not something strange to anyone, in fact it has been present in our lives since childhood.
part of the main stages of the individual's development. Including many
animals exhibit behaviors that indicate the existence of games with different purposes.
verses. Through games we develop specific qualities that are useful to us in life.
adult or that help us to get around difficult situations enabling relaxation, distraction.
tion, fun. In order not to get into infinite philosophical discussions, we will do some of the following.
comments that we think are pertinent in relation to the Game and the counterpart in relation to DO methodology.

- Despite its primarily playful character, the Game has its own seriousness that it does not depend on the elements that play it. This seriousness determines a world where unroll the actions and movements of the Game. Everyone must respect this formality, otherwise there is no Game. Formality is always a positive factor when related. to computation elements. A development system must provide this "world". Obviously, for the illusion of Game to be respected, this “world” must be as closed as possible and limit the number of extraneous elements to the dynamics

of development.

- The Game has a nature of its own, independent of the conscience of those who play it. The subject of the Game is not the players, but the Game itself which, through play, it simply gains representation. Like the Game, the result of the development development cannot be subject to the whims of designers. This means that the specifications of a project are part of the specifications of the Game environment. development, in order to limit and direct the actions of the designer(s) towards the common goal.

- When considering the meaning of the word Game, we often associate with it the idea of set of objects and movement between them. Likewise, digital systems

can be understood as objects that present a very specific dynamic. each other. The Game, on the other hand, must represent an order, in which the movements, in the same way that the dynamics of the system translates a behavior coherent.

- One of the playful attractions of a Game is its lightness, that is, the game's movements they should not require effort. This invites the player to explore possibilities and the man- focuses on playing. This is an essential point of the methodology, the inexistence of efforts. Each movement, translated by a certain action in the de- development, must allude to the lack of difficulty. That is, each action must be for the simple and immediate designer, allowing it to remain focused on the task under development. Obviously, this lightness doesn't need to have a real character, just an allusion to the lack of effort, as in a game of “Monoreal Estate” can be buy/build an industrial set based only on the result of a die.

• Another attraction of the Game is the playful character of the competition, where the coming and going of the mo-
effort-free, produces situations and conditions sometimes beyond the forecast
of your players. The Game surprises. Another important character of the methodology
is to react immediately to the modifications or movements made by the designers of
in order to offer as quickly as possible the information and consequences arising from the
same.

• The attraction of the Game, the fascination it exerts, resides in the fact that the Game masters itself
of the player. Likewise, the development environment must captivate the designer
in order to keep you in the "game" performing your duties until the goal is
Reached. The Game is what keeps the player on the way, that entangles him in the game, and that the
keeps in play.

• Finally, each game sets a task for the man who plays it. So it fits
the development system put this objective to the designer and reinforce it in each
moment that is possible, so that it is never lost or dispersed.

Figure 3.3: Example of a hardware description in which graphical and textual representations (SelfHDL) mix.

3.6 Implementation

We can then outline some features that are fundamental to the methodology designer-oriented: The system should exhibit a level of integration never seen before in the conventional frameworks; it must be interactive and interpreted, or at least interact with the user as it were; be facing the field of application; and finally, must make use of all available resources to keep the designer's attention on the object of your job. We will see below how we can implement such features in a way efficient.

The integration between tools, environment and user is a very complex problem. However, like any other problem involving complexity, this one can also be solved by adopting an adequate representation model. This model is offered by Self language. Using a special language to implement a framework Integration is not a new idea, we can mention Cadence's SKILL. We can appro- to veto the facilities of development and work with Self objects to promote the system integration. Self is a high-level language that has a conception of very peculiar object orientation which makes it also very simple. The fact that it is based on prototyping and having dynamic types also contributes greatly to this simplicity. All elements in Self are objects that can be accessed, tested, copied, etc.,

even during the execution of a program. This brings us to the second feature, the interactivity: in a Self program an object can be modified even when it is being used by a program. It's as if a mechanic could change the characteristics of a gear while the engine was running. This feature is fundamental in the implementation of the methodology because the objects can be conceived as real implementations of the development primitives, instantly displaying their functionality the instant they are instantiated, just like a real component is used on a test bench. To function as a bench, the system must also be ma offers an interactive simulation engine that has characteristics closer to emulation than simulation itself. In [Mau96] we see that an inter- blackda can be almost as efficient as a compiled one, although the example refers to logical simulation we can easily extend the concept to hierarchical functional simulation. ca with similar results. Finally, to create a more perfect illusion of this cybernetic reality of development, a lot of graphic resources should be used. In [CUS95] is presented a graphical implementation for Self that keeps programming focused on objects. Once again this concept can be extended given to the application domain in order to create the “virtual world” of development, suggested when we talk about the Game concept.

As an example, let's see how some fundamental objects of the method are implemented. theology. In a hardware development process, it is natural and desirable that the designer also think in terms of hardware. This is necessary because the closer to the objective. The end of the simplest process are the necessary help and synthesis tools. Following this principle, the objectThe main object of the methodology is the "Comp" object, if we compare with other HDLs, it would be equivalent to VHDL's “Entity” or Verilog's “Module”. Evidence- fearfully, in the methodology this object has a much larger scope than in these HDLs. Comp can be described in functional or structural terms. Functionally, this dis- creation can be done in several different ways: state diagrams, petri-nets, models symbolic and textual, SelfHDL being the description originally developed.

* We will make all descriptions in terms of objects, since in Self there is no concept of classes. Objects can inherit from other objects (multiple inheritance), and are used/created from copies of others special objects maintained especially for this purpose are called “prototypes”.

The structural description consists of an interconnection of other Comp objects, with the help of of Node objects. In the simulation process, these objects have the propagation function of events between the various components of the system. Nodes inherit their signal model from special objects implemented for this. As in Self, even inheritance can be attrib- dynamically, the Node's "parents" can be modified to reflect the sign that is most convenient. Currently, the system is being implemented with a mo- simple delo of two values (0,1), and a model compatible with the STD LOGIC 1164 standard [Soc93] of nine values to be compatible with the VHDL models. Evidently to that there are connections, it is necessary that there are input to output ports. All Comps have input and output ports, quantities vary with object functionality.

The evaluation of the state of a component is done by sending the message "step" for it, regardless of the type of description (functional/structural), the message starts the calculation of outputs as a function of current inputs. In the case of a description that works, In the end, this is done by directly following the procedure specified by the description; in case structural, a scheduling list will selectively coordinate the sending of new messages step for the subcomponents of the description. At each computed and modified output, a new event will be generated and added to the scheduling list. This scheme is typically event- driven, however we adopted optimizations suggested in [Mau96] that allow us results interesting.

Scheduling

Figure 3.4: Component evaluation scheduling scheme.

A Comp that is not changed by a given event need not be included.
in the scheduling list, this allows, for example, that cycle based simulations can be

made using the same mechanism. Customizing Comp objects and indicating that they are sensitive only to certain Ports (“clocks”), we can easily transform a simulation event-driven in cycle based, or even combining the two in a given system.

So far, the implementation doesn't look very different from other systems. En- however as we said, the big difference lies in the integration of tools and in the interaction with the user. Suppose, for example, the ideal workflow in the methodology DO and how it operates with the presented objects. Suppose we are working in a project using one of the available high-level descriptions, after the initial checks (simulations) a more effective (formal) verification is needed so that it is possible to pass for the next steps of development. This is done by simply sending a message. convenient gem to the Comp object. For example, checkProp. This message may consist of of a verification of properties previously configured at the beginning of the project. One time confirmed the consistency of the model, we would pass the high-level synthesis phase, in which based on the functional description, an architectural (structural) model of the composition would be generated. nent. Again this would be the result of a simple message, for example: archGen. The generated model would not create another object, but add to the original Comp a new description, this time structural. Comp can have as many descriptions as necessary,

the newly generated description would become the current description and the base would be the previous description. Once the new description has been exercised (simulated), a new step of formal rification would be necessary, this time comparing the two implementations, through the verify message. We could follow these steps successively until we get a description of the original Comp in terms of Comps at RTL level. At this point we could easily get a conventional HDL description through a message like VHDLGen, or VerilogGen, that would be in charge of generating a synthesizable HDL description of the designed component.

An important point to remember is that implementing the framework in Self is possible a functionality exactly the way it was proposed. each of these messages can be simply a button on the Comp representation icon, and that only available when and if the conditions necessary for its functioning were also available. available. No concepts such as files, filenames, directories, command syntax and etc, as everything would be accessible and organized automatically through of the Comp icon.

The interactivity and immersion characteristic pointed out in section 3.5 through the concept of Game, is obtained initially through the implementation and manipulation of the elements' icons of project. Graphic elements are very important in this methodology, not only as organizational elements as pointed out above, but also as essential elements for the creation of the project “virtual reality”. For example, through of the same icon, the designer could interact with Comp during a simulation. Using special objects, Observers, the designer could connect to the Ports of a component and observe the result of the simulation as it unfolds. obviously, a Observer could be customized to format the measurement point(s) accordingly. with the convenience of the designer, and thus it becomes a measuring instrument, a panel of

control and etc. This is also very interesting as it allows the designer to format and mask the implementation of a given system and allow presentations of real simulations for teams less concerned with hardware implementation details, such as: software development, marketing and business.

In the next chapter we will present the implementation of the SelfHDL system for examples. simplify the methodology presented here.

Chapter 4

SelfHDL

NTHIS chapter will present the basic set of objects that composes the system SelfHDL. We will see that this set was designed to describe and emulate elements. digital hardware, so that they can be used as a description language for hardware both as a means of exploration and modelling. We said "emulate" because how every Self object, the objects that make up a SelfHDL system are “live” objects with which we can interact from the moment of their creation. This makes the development process much more interactive and intuitive for the hardware designer. The system too allows the circuit described to be simulated within the environment itself without the need of auxiliary simulation programs.

4.1 Designer-Oriented Methodology and SelfHDL

We saw in the previous chapter that the designer-oriented (DO) methodology consists essentially of cially in eliminating from the design flow concepts and operations that are foreign to the domain of application, in this case, the design of digital systems. This is done through the use of a tool. computer minds specially developed to hide aspects from the end user undesirable effects of the project. During the use of these tools, the work should be so interactive.

as possible. The designer's attention must be captivated by the tool avoiding thus, that the sequence of your thoughts is interrupted by operations or information strange to the object of work, as a player is captivated by the Game universe that is being played, hence the analogy proposed in section 3.5 . In the DO methodology, we look for

avoid concepts such as files, intermediate steps (compilation or post-processing) and the notion of tool for the execution of specific tasks. Development must follow in a gradual and interactive way, giving the elaborated objects all the accessibility that is possible.

We chose SelfHDL to exemplify this methodology, because the hardware description plays a key role in the development process. For being the initial step of many design systems, there is nothing more logical than to present a hardware description that already adopts the concepts of the DO methodology and puts them into practice effectively.

SelfHDL stands for “Self Hardware Description Language”. he is essentially a digital hardware description system made entirely in the Self language [US91].

Figure 4.1 shows a simple example of a SelfHDL description, we can see that the description mixes graphic elements with textual descriptions. The interaction and dynamics of these elements will become more evident in the course of the presentation.

Figure 4.1: Example of a SelfHDL description/simulation.

A SelfHDL project is not just a description in yet another Description Language. Hardware creation (HDL), but an actual implementation. All elements described are “live” objects that can be handled and inspected immediately after their creation. Some have graphical representations that allow part of the description to be done much like the classic schematic captures. In the last years

we have observed a large migration to standard HDLs [Soc02, Soc01], in opposition `the classic forms of graphic capture. This was due to the greater flexibility that these The latter offered at the time, such as: the ease of describing and combining various levels of abs. traction, independence from technology, excellence as an input to synthesis tools automatic and formal verification[Sag00]. Paradoxically, several adaptation attempts from graphical tools to design flow [Vel00, AIG99] has been proposed. Descriptions textuals are very versatile; however, the understanding and documentation of these descriptions become extremely difficult in complex projects. Unfortunately, these proposals con- they only add steps to the design flow through the use of representations. intermediates and abstractions not always adequate to the development task, increasing also the maintenance costs of these systems.

The approach used in SelfHDL, combined with the DO methodology, eliminates these difficulties. and enjoy the best of both “worlds”. Descriptions can be textual when

necessary or convenient and also pictorial, becoming much more expressive than a obscure poorly documented text. In figure 4.20, we can see an example where text and structure are combined in a balanced way to provide the maximum amount of information in a single figure. Another advantage of SelfHDL is the fact that the simulation engine is naturally embedded in the elements of the description, being fully transparent to the user. Once a component or circuit is created, it can be immediately put into operation as if it were a real circuit. Still in Figure 4.20, we see a For example, “switches” and displays were connected to the circuit to inject signals and observe data from the circuit described. We'll see how this is done in more detail below.

4.2 Hardware Description Language in Self

SelfHDL is a collection of Self objects designed especially to describe, simulate, home/emulate digital hardware. We kept some similarities with traditional HDLs, especially VHDL in order to make its use more intuitive and easier for users to learn potential. SelfHDL is composed of a set of graphical and non-graphic objects, each which with a specific purpose in the system. The presentation of the main objects that make up the system is done next.

4.2.1 The comp object

The comp object is the prototype of all objects that describe hardware behavior in the SelfHDL system. This means that the entire description of hardware behavior starts by making a copy of the prototype comp and then modifying its properties. to reflect the behavior of the new component. This object can be compared to the VHDL “ENTITY” statement, with some differences, for example: the behavior

specified in an object of type comp refers only to sequential character statements.
of the description, that is, basically to those that appear between the statements “PROCESS” and
“END PROCESS” of the VHDL. Figure 4.2 shows the graphical appearance of the comp object. The object
comp is a fully operational implementation of a hardware component, ie,
it can be used in a real simulation like any other component, although this does not
be the most recommended. Normally, we try to preserve the prototypes so that they don't
accidentally corrupted during use.

Figure 4.2: Graphical representation of the comp object.

All objects of type comp are graphically represented as continuous blocks,
and internally divided into frames that are used to delimit the various fields in the
component representation block. Each of these frames has a meaning: those of
left represent entries; those on the right are the outputs; the center frame contains the
description of the behavior for the entity and finally on the upper side, there is a frame
non-visible which displays the name of the described component. The idea of concrete objects of the
Self is also used here, the block can be held and moved around the environment.
graphic through the mouse and connected to other components as if we were using
a classic schematic editor. The difference is that this graphic is not simply
a symbol but a fully functional hardware component.

Another idea inherited from the Self philosophy and which combines perfectly with the method-

ology DO is the absence of the idea of separate tools, that is, in Self we avoid the use tools to perform specific tasks, instead we expect the object itself provide the desired functionality. Following this trend, objects of type comp can be modified without major problems, with just a click of the mouse. Per example: if we want to change the behavior of a comp object, we use the do key. half-mouse over the central frame and choose the “edit behavior” option. The frame tral will become a simple text editor, allowing any changes to the behavior is made.

The creation of an object of type comp can also be done through a simple script, typed into a shell object like the ones in Figure 2.15. The implementation of a verifier parity checker can be seen below:

```
comp name: 'parity'
      Inputs: [| in <- nodeVector newType: std_unsigned Size: 8 |]
    Outputs: [| p |]
  Behavior: [| the |
                        o: (at to: 0).
                        (in length - 1) do: [| :i |
                            o: (a at: i) xor: o].
                        p setTo: o ].
```

The internal state of a comp object is computed by running the behavior method, shown in its center frame. The system knows that the status must be recalculated when one of the comp entries is modified by an event. Events are generated and transmitted through the message “setTo:” sent to an object of type node or nodeVector. Like was seen in section 2.2.3.3 and exemplified in figure 2.13 , when the message is sent “behavior” for a comp, an “activation record” object is created, containing a slot with the method to be executed and an implicit “self*” parent slot, which points to the comp object. who received the message. This mechanism establishes the message evaluation context. So that input and output ports and internal states can be referenced within the behavior method directly, each of these elements would need to be slots of the comp object itself, which would cause a lot of confusion in the internal structure of these objects. To avoid this problem we use an unusual feature of Self, the parent slot modifiable, that is, a parent slot can have its contents pointed to positions or objects different over time, as if it were a simple variable slot. A comp object

has three of these slots: inputs*, outputs* and state*. These slots point to objects

simple, which in turn have a slot for each reference they are intended for, that is, one for each input in the case of the inputs* slot, one for each output in the outputs*, and one for each state internal to state*. Because they are marked as parent slots, these new references enter the evaluation context of the behavior method when evaluated. The figure 4.3 shows this mechanism.

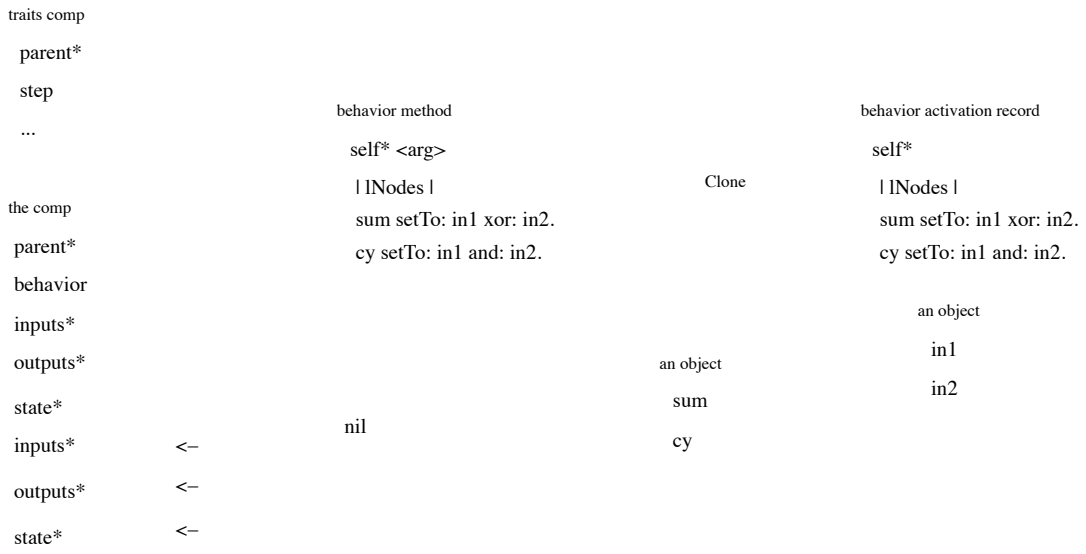


Figure 4.3: Behavior message evaluation scheme in a comp object.

4.2.2 The node object

The node object is the prototype for all objects that describe real * signals in the system SelfHDL. These objects can be used as variables for intermediate calculations. or as event distributors, spreading events throughout the system. are not objects

graphics as are the comp objects, since its use presupposes a high performance for guarantee the efficiency of the simulation. These objects are very simple. From the point of view of instance, a node object has only five internal states: type*, this slot points for an object that defines the node's type; value, contains the current value of the node; lastValue, contains the previous value the node owned; driveMe, contains the list of components that

* We refer by signs, electrical connections between real components. Sometimes we might use the term node, following the same context as electrical circuits.

can modify the value of that node, and finally iDrive, contains the list of components that should be updated if there is a change in the current value of the node.

For the purpose of demonstrating the methodology, two types of nodes were implemented, through the following objects: bit_logic and std_logic, both to make a parallel to “bit” and “IEEE Standard Logic 1164” types [Soc93] of the VHDL. as expected the bit_logic object defines a logic of two values and std_logic defines a logic of nine values. The type* slot is a modifiable parent slot, like the ones presented in the section previous. It can point to bit_logic or std_logic depending on the type of signal that we want to describe. This means that the node will inherit the behavior defined by one of these. objects. The usual methods defined by these objects are: logic operations, conversion of types and structures of control. A computation is done by sending an appropriate message for a node object, usually producing another node object as a result, by example: let a and b us and c be a modifiable slot, then the expression: “c: a and: b”, means that ac will be assigned the node resulting from the sending of the message “and: b” to a. note that node b is the message argument.

As event generators and distributors, node objects are activated by the message

"setTo:". It is usually one of the last expressions in a description of behavior, as can be seen in the previous examples. The message "setTo:" triggers a sequence of operations to ensure that all components that will potentially be affected by this event are updated. First it is checked if the message really change the value of node, that is, if there really is an event; then it is verified that the node is inside a schedulerMorph, that means it belongs to a hierarchical level. well-defined or interactive simulation; if so the node's iDrive list is copied to the end of the event list of the respective schedulerMorph and then to each iDrive component, the event that sensitized it (that is, the affected port and the new value).

4.2.3 The nodeVector object

nodeVector objects are the natural extension of node objects. As the name might suggest, they are vectors of nodes or buses, vectors of node objects. Most comments applied to nodes also applies to nodeVectors, the difference is that the operations

with vectors can include arithmetic operations and numerical conversions. This is done by bit_unsigned, bit_signed, std_unsigned, and std_signed special objects, which define the type of array indicated by slot type*, similar to the case of node. these objects define arithmetic, relational operations, conversions and size changes. An example nodeVector was used in the creation of the comp for parity checking. the slot value of a nodeVector points to a vector of node objects. The vectors in Self are indexed through natural numbers, that is, $n \geq 0$.

nodeVectors also function as event dispatchers, therefore, too respond to the message "setTo:". However, they don't have the iDrive or driveMe lists.

like the nodes. In fact, instead of managing each of the signals individually, the nodeVector does this collectively; because, most of the time, when we group several signals on a given bus, we are exactly indicating the cohesion between them. Of that In this way, we choose the “zero” order sign and use their lists that will be common for all other bus signals.

4.2.4 The connection object

Connection objects are the graphical counterpart to nodes and nodeVectors. Is-these objects are used to make the connection between components, when it is being used the graphical modality to compose structural descriptions. A connection object is created when the middle mouse key is used to connect two or more components. THE operation is done by placing the cursor over a component's output and selecting the “connect” option in the middle mouse button menu. A connection object is created with its rear end attached to the output of the component and its other end following the cursor. The connection is terminated by taking the end of the connection to a port † compa-and over it by releasing the arrow with a touch of the mouse. We can see that the operation is very similar to a classic schematic capture tool, but actually the object connection updates the status of the connection on the associated node or nodeVector object. That means that a real connection between the components has been established, providing the two objects a real communication channel that can be used immediately.

The graphical aspect of a connection object can be seen in figure 4.4 , note that

† Ports are objects used as transition elements between different hierarchical levels.

Figure 4.4: Graphic representation of two connection objects.

the thickness is proportional to the size of the associated node, simple node connections have two pixels thick (as seen on the right side of figure 4.4), vectors have connections pro-progressively thicker (as seen on the left of the figure). This figure also shows a port of an input vector and a port of an output node.

4.2.5 The schedulerMorph object

Objects of type schedulerMorph are responsible for coordinating the evaluation of the state. internal of the various entities connected to a structural component, see section 4.2.6 . These objects also have a graphical representation. Appear as a rectangle that extends over an area covering the circuit that is to be simulated or considered as an assembly, block or structural component. In fact, the circuit in question is inside the schedulerMorph rectangle. We'll see how this works later.

schedulerMorph objects can be used in two different ways: establish an interactive simulation/emulation environment for the designer or establish a hierarchical level, defining a structural component. The graphic environment of the Self implements has many interesting features that aim to make the programming experience more intuitive, among them the use of animations that resemble cartoons and that, through some effect visual, inform the programmer/user of some action that has been performed in the system. This feature is exploited by schedulerMorph objects to make circuits live and interactive. In the Self environment, there is a main scheduler that distributes the activities of the world Self among a group of objects noted on your activity list. these objects are generally those that need periodic updating, for example, a display of clock or even an animation. A schedulerMorph object can be included in the list. of activity sending the "start" message to yourself, to be deleted, just send

the message “stop”. Once included in the list, the main scheduler sends the message. "step" to each of the objects in the activity list, each object in turn updates its internal state and respective graphical representation and then return control to the scheduler main. The object that returned the message is then passed to the end of the list and a new object is chosen to receive the next “step”. So all objects have the opposite unity to be updated equally. It is the object itself that decides when it should leave the activity list, for example, an animation after it has its effect flagged it can be excluded from the list, not overloading the main system scheduler.

The activity list in the main scheduler is a simple list mainly because normal Self objects don't usually depend on each other. On the other hand, in a circuit description, the objects depend on each other and the evaluation must follow the topology of interconnections established. For this reason, schedulerMorph objects have lists dedicated to reflect the interconnect topology and the sequence of events. every time a schedulerMorph object receives the message “step”, selects comp or from its lists. sComp that must be updated at the moment and sends the message "step" to it, updating the internal state of that component. Once updated, the object is removed from the list, which means that its state no longer needs to be recalculated due to the current event. It makes make the scheduler work like a real bench circuit. The designer can probe, test or modify the circuit at will, working in a kind of virtual reality of the project. Figure 4.5 shows the event and dependency lists of an object of the schedulerMorph. Each position in the event list corresponds to a particular event and contains a list of components that depend on (or are affected by) this event. When the component is structural, control is passed to the schedulerMorph associated with the level lower, and then the new dependency lists are used to compute the state of the component at that level.

As the internal state of the components is computed, new events are generated. and added to schedulerMorph's list of events/dependencies. Under normal conditions,

the list increases and then decreases as a function of the convergence of the propagation effects. Thus, we can say that the effects of a given event have all been computed when schedulerMorph's list of events and dependencies is completely emptied.

Figure 4.5: Lists of events and dependencies of a schedulerMorph object.

4.2.6 The sComp object

sComp objects are a specialization of objects of type comp designed to describe structures. They also correspond to an "ENTITY" of the VHDL, but only for those parallel statements. In place of the behavior slot, sComp objects have a scheduler that points to a schedulerMorph object, which contains information about the level hierarchical structure to which the component corresponds. Following the same philosophy, a component structural can be created in different ways and between them we can make use of a script or draw the circuit like a classic schematic capture tool. Through scripting mode, we use a message that, sent to the prototype, provides the copy and the

initialization of the properties of the new sComp, as can be seen in the following example.

```
sComp name: 'FlipFlop'
        Inputs: [| sset. rset |]
      Outputs: [| q. qb |]
  Structure: [| g1 = comp nand portMap:
                                  [| a = 'sset'. b = 'qb'. c = 'q' |].
                        g2 = comp nand portMap:
                                  [| a = 'rset'. b = 'q'. c = 'qb' |] |]
```

This message creates a new object by copying the sComp prototype, then a new set of inputs and outputs is placed. In the new schedulerMorph of the new sComp puts the components listed in the argument of “Structure:” with the ports and associated connections. Positioning follows a very simple algorithm so that the result is not always visually pleasing, however, it can be reedited later if a good presentation is needed. The important point is that the component created is fully functional and can be used immediately.

Another way to create an sComp is by drawing a schematic diagram with the previously created comps, pulling connections from outputs to inputs as described previously. When schematic capture is complete, just call an empty sComp using the message “sComp copyEmpty” and the schedulerMorph handler associated through the middle mouse key over its center frame. The handler allows you to create ports from input and output or invoke the schedulerMorph of this sComp, it must be placed in the part left top of the circuit to be captured and then the “sched” key must be pressed. Finally, we drag the rectangle created over the circuit and click to mark the corner bottom right and finish the capture. schedulerMorph then captures the circuit below the covered area, which becomes part of its internal state and consequently of the respective sComp.

The graphical representation of an sComp object is seen in figure 4.6 . The central frame of the

graphical representation is a scale simplification of the real schematic diagram, only one figurative reminder of what this component represents. It is possible a very

easy between several hierarchical levels through the middle mouse menu, choosing the option

“go down” the entire window jumps to the respective schedulerMorph if it is in

Self world, otherwise schedulerMorph is appended to the cursor to be placed in

somewhere more convenient.

Figure 4.6: Graphic representation of an sComp object.

4.3 Hierarchy and Dynamics between Objects

In this section we will present the essential objects that make up a SelfHDL system. THE

object hierarchy will be presented using its own notation due to the difficulty

to frame the structure of the Self in conventional notations. Because we don't use

classes in Self, we find that the adoption of a representation such as diagrams

of UML classes [OMG00 , Lar00] could give the false idea of adopting this concept. THE

our notation is quite simple, in fact, it very much resembles the representation of

objects presented in the Self environment. It essentially consists of blocks representing

objects, the kinship (inheritance) is represented by the parent object(s) placed above the

reference object shifted slightly to the right and connected to it by a dotted line.
Main inheritance is represented by the first object `on the left, the others follow `a
right on the same level. The main hierarchy is usually the most important and is the only one
represented in the diagrams of this work. Whenever possible, objects that in
some point share of some inheritance sequence will be placed on a level
that allows you to mark the shared object with a horizontal line. see how
example the diagram in figure 4.7 . When an object is expanded to show some
of its properties, the block is drawn slightly thicker, with the properties
of interest listed just below the object's name. Everything in Self is object, every slot
points to an object that defines it, this is represented by an arrow between the slot and the object
pointed out. An example of this notation appears in figure 4.8 . The dynamic, however, does not
offers the same difficulty, so we will use UML sequence diagrams for
represent some dynamics in the most relevant cases.

4.3.1 Object Hierarchy

It would be very confusing if we presented the objects without a certain logical ordering, so
we group the objects that have the greatest affinity and place them on the same diagram.
In this way, we start with the objects that describe or are in some way associated
to components:

comp As previously stated in 4.2.1 , the comp object is used
to describe a component by its behavior.
All SelfHDL functionality is factored into the traits object
comp, the others come from its graphical inheritance through the object
frameMorph self traits ‡. The other objects of immediate inheritance

‡ In general, graphic objects in Self have the word “morph” in the name.

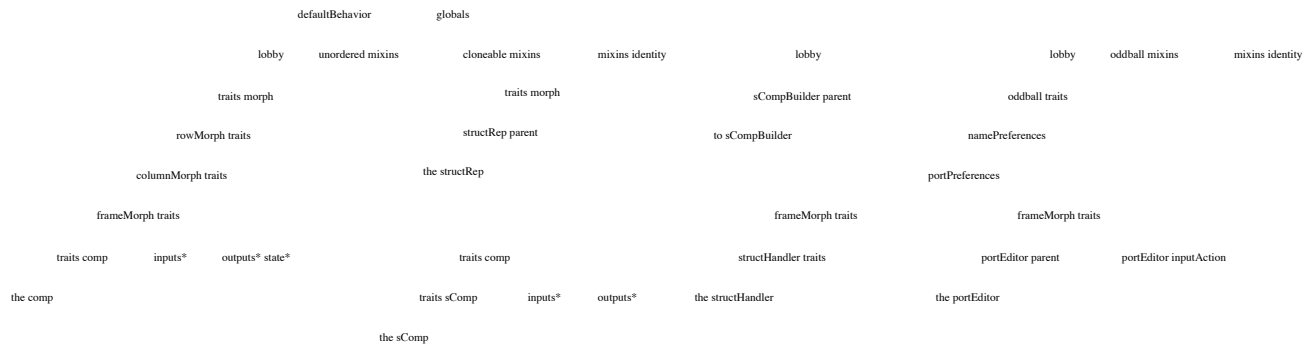


Figure 4.7: Hierarchy of comp and sComp objects and some other auxiliary objects.

diata are used to store the input, output and internal states when these exist.

sComp The sComp objects are used to describe a component through its structure, that is, through the connection of other components. It's a specialization of the comp object, how can you be seen in figure 4.7. Specialization is factored into the object traits sComp, note that just like comp, sComp too has in immediate inheritance objects to store the ports of input and output, however, there is no one that stores states internal. They are stored in the very elements that compute the sComp.

sCompBuilder sCompBuilder objects are auxiliary objects used in creating action of new sComp objects through textual descriptions. Those objects organize the information referring to the new sComp as internal components, connections, etc., and coordinate the creation of the new component.

structHandler The structHandler object is the rectangular object that appears in the upper left corner of a schedulerMorph object. He is used to aid in the graphical manipulation of skeletal components. structural. Through it we can create input and output ports and invoke the associated schedulerMorph to capture the circuit

newly edited.

portEditor portEditor is a form that helps you create ports
input and output when used by structHandler .

structRep The structRep object is a pictorial object that represents the struc-
contained in a structural component. It's that representation
that appears in the center frame of the graphical representation of
a sComp, see figure 4.6. It has the property of being a link
connection with the real scheme of the component and there is still the
possibility of modifying the representation scale, which can
get bigger or smaller according to convenience.

namePreferences The namePreferences object contains the style definitions of the re-
graphical presentations used in SelfHDL. Definitions such as
colors, fonts and basic actions in the case of line editors are defi-
contained in this object. It inherits from oddball traits, which means
that it is unique in the system and cannot be copied.

portPreferences The portPreferences object has the same kind of function, it is a
specialization of namePreferences special for ports. Tam-
well it is unique in the system.

Next, we will present the objects dedicated to component connections. Only one of them have graphical representation. Are they:

connection The connection objects are by far the most complex of this

group. They embody the connection graphic representation. Is-
we harvest to disassociate the graphical representation from the functionality
in the case of connections to reduce the computational weight of these
the last ones, which are used intensively in the evaluation of composition.
treatments. Connection objects graphically appear as
indicated in the lower right corner of figure 4.8 . Accompanying
the geometry of a component, the connection starts at a
point at any point and walk to the right. From there, the co-
nexon can follow several directions according to the position of the

Figure 4.8: Hierarchy of connection objects and types.

target-component and finally ends up following again
to the right ending in a "head" shaped like a tip of
arrow that indicates the propagation direction of the events. The con-
along segments that determine the path of a connection is
called branch. If a connection ends at more than one point
more than one branch is needed to characterize the link.
The branches of a connection are represented by objects
connection branch. The starting and ending points of a connection
are determined by the connection grip objects.

connection grip These objects determine the starting and ending points of a co-
connection. They belong to the circleMorph object family of the Self since
are represented by small 2-pi-dimensional circles.
xels. There is a grip for each branch of the connection and one more
which determines the starting point of the connection. The movement of

these objects determines the update of all the geometry of a
connection.

connection branch These objects determine the connection path between two points.
all. If there is only one branch on a connection, the path
is determined by the position of the two available grips: the grip
initial, pointed to by the tail slot of the connection object and by the
final grip, pointed by the head slot of the branch object. If it exists
more than one branch, the others determine the path from

some position among the segments of one of the previous branches.

flowers. This new starting position is determined by the offset slot

of branch. In this new starting position, a drawing is placed.

point (small circle) to indicate contact and not a mere cross.

different connections. The algorithm that determines the

segments of a branch is quite simple, so

fearfully, operator intervention is required to improve

the look of a circuit. The intention is exactly this, to avoid

great computational complexities when human intervention

mana is more efficient and unavoidable. The general rule says: when the

ending point is to the right of the starting point, are generated

one, two, or three segments as paths to the branch; if the

point is to the left four or five segments are generated.

node The node objects are in fact the communication channels between com-

components. Note that for each connection object there is a node

or nodeVector associated through the owningNode slot. The function

The basic functionality is factored into the traits node object and the speci-

signal type assignment is done through bit_logic and

std_logic. As stated earlier, our intention with

the creation of these two types was to create a model as close as possible.

possible of the VHDL to have a means of comparison between fer-

tools and also make our system familiar to users

of that language.

nodeVector The nodeVector objects, as their name suggests consist in a vector of nodes. The implementation logic follows closely the implementation of node with the difference that the types are more abundant, since signal vectors can be added. allowed or not as signed or unsigned integers. The basic types are determined by bit_vector objects. and std_logic_vector and the specializations through the objects: bit_signed, bit_unsigned, std_signed and std_unsigned. THE Basic functionality is factored into the traits nodeVec- objects tor.

bit_logic This object defines logic operations, conversions and control for nodes defined by only two logic levels.

std_logic This object defines logical operations, conversions and control for nodes defined by nine logic levels.

bit_vector This object indicates nodeVector formed by a vector of nodes of type bit_logic.

std_logic_vector This object indicates a nodeVector formed by a vector of nodes of type std_logic.

bit_signed This object indicates that the nodeVector is formed by a vector of nodes of type bit_logic and which can be considered as a signed integer for arithmetic purposes.

bit_unsigned This object indicates that the nodeVector is formed by a vector of nodes of type bit_logic and which can be considered as a unsigned integer for arithmetic purposes.

std_signed This object indicates that the nodeVector is formed by a vector of nodes of type std_logic and which can be considered as a signed integer for arithmetic purposes.

std_unsigned This object indicates that the nodeVector is formed by a vector of nodes of type std_logic and which can be considered as a unsigned integer for arithmetic purposes.

Other equally important objects, but which could not be grouped into categories. previous gorias are presented below. The implementation hierarchy of these objects is shown in figure 4.9 .

Figure 4.9: Hierarchy of other important objects.

inputPort The inputPort objects are input interfaces to hierarchies. the superiors. For each input port of an sComp there is a corresponding inputPort in the schedulerMorph that contains the circuit of that component. Sometimes when we create a simulation set, we put a pair of inputPort and outputPort dummies only so that the associated sComp appears in the correct proportions, as can be seen in figure 4.12 .

outputPort The outputPort objects are output interfaces to hierarchies. the superiors. The same remarks regarding the entries if apply here.

portFrame The portFrame objects are also related to the ports of generally speaking, they are their graphical representation. These objects are important as they are the only ones besides worldMorph authorized receiving a connection grip. These objects are also as- associated with a node or nodeVector and implement various functions.

interesting information via the middle mouse button menu. As it could not be otherwise, they are present in the objects inputPorts, outputPorts, comps and sComps.

schedulerMorph The schedulerMorph objects are the elements that determine the hierarchy in a SelfHDL system. As has been said, the same can be used as an interface for an interactive simulation or only as an element of hierarchy. Graphically your imple- mentation is quite simple as can be seen in the hierarchy. of figure 4.9 . It descends from an object of the Self system, the carpetMorph, used to capture and move blocks of objects within the Self environment.

navigator It could be said that the navigator object would be of the category of "cosmetic" objects of the SelfHDL system, however, our implementation is of paramount importance. As you know- mos, the environment Self consists of an infinite two-dimensional space and, as such, it must provide a way of "navigating" over that space. space so that we can move the visible area over the objects of interest. This function is done with the help of the Self object.

radarView and radarDisplayMorph. These objects allow the
movement of the visible area and create a pictorial representation.
rich (like a radar view) of the region around the area of
vision. However, this is somewhat confusing from a strategic point of view.
chematic, as this point of view always has as a reference
the center point of the screen. The browser object is a specialty.
tion of radarView, it divides the space into units (cells)
multiples the size of a visible window and allows you to move.
between cells, the reference being, in this case, the beginning of the
window. This makes the movement and occupation of space more
intuitive, as it resembles the use of independent sheets to
the design or documentation of schemas.

navigatorDisplay The navigatorDisplay object is a specialization of radarDisplay.
playMorph. It was designed to reflect the new behavior.
proposed by the browser. The browser and browser objects
Display can be seen in figure 4.10, in its representation
print shop.

Figure 4.10: Graphic appearance of a browser object.

Finally, we have the group of inspection and interaction objects, which are the elements of instrumentation of the SelfHDL system. In reality, this group can extend far beyond of the other groups, because, in fact, this is an open category in the system. The intention is we can easily adapt or create the instrumentation according to the convenience of the project. We will, however, present some basic elements by way of illustration. THE their hierarchy is seen in figure 4.11., where we can note that invariably are all specializations of the comp object.

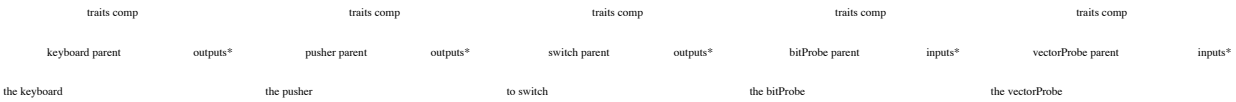


Figure 4.11: Hierarchy of inspection and interaction objects.

switch The switch objects are interacting elements with the vir-
simulation method. They are a specialization of the comp objects, in the
they are actually comps that have only one output port. The inte-
activity is achieved by a button that can be pressed.
by the operator (user), inverting the current state of the output. O
current state is also reflected in the color of this button.

pusher The pusher objects are the equivalents of switch to vectors.
of us. Its only output is a variable-size nodeVector
and in place of the switch button we have a line editor, where
we can directly type the value we want to inject by

exit. This value must be in simple binary format.

keyboard The keyboard object is an improvement on the previous two,
being also a simple example of how an object of interest.
ration can be modified to make the simulation more comfortable.
It is a pusher whose output is a nodeVector of four
bits and that, in place of the line editor, it has a small keyboard
hexadecimal.

bitProbe The bitProbe object is the simplest form of inspection object.
dog. As its name suggests, it is used to inspect the
state of a node. This is a comp with only one entry,
therefore not generating output events. However, your
alization implies changing the color of its flag element
as a function of the value of its input.

vectorProbe The vectorProbe object is the corresponding one for an input.
of vector type. The representation in this case is textual and binary,
updated also in the signaling element.

4.3.2 Dynamics of the simulation

It is unfeasible to present in this work all possible message sequences that
make up the SelfHDL system, so we chose the one that we think is the most relevant: the
interactive simulation sequence. Sequence of messages, contrary to the description of
Object hierarchy can be represented by UML sequence diagrams [OMG00 ,
Lar00], so this will be the symbology adopted in this section.

To describe an interactive simulation situation we do not need examples
very complicated, in fact, the principle is identical for any circuit complexity.

Figure 4.12: Typical simulation situation.

Thus, we will initially admit the example of figure 4.12 , where we can see a case simulation at a single hierarchical level.

4.3.2.1 Inserting an event in Interactive mode

Figure 4.13: Sequence diagram for inserting external events, such as interactions with the user.

Inserting an external event into a simulation event is done through a interaction component. We are assuming the switch element of Figure 4.12 , which appears. receives under the name “stimulus”. The sequence of events can be followed in the diagram. sequence of figure 4.13. The (human) actor initiates the interaction through a touch of the

mouse over the switch element, this action is translated in the Self environment as a series of events between objects normally transparent to the user. The first object affected

is the handMorph, the element that embodies the cursor of the Self environment. once noticed the event, it is classified and transferred to registered objects (suitable) to receive events of this nature. In this case, the element is the button of the switch component, represented in the system by the Self ui2Buttom object. The ui2Buttom object internally has another object that characterizes the action that the button should perform when pressed, this object is the buttomActionObject, which in turn calls the “behavior” method of the switch element. This method updates the color of the flag element and inverts the switch output by generating a event in the simulation environment.

4.3.2.2 Event propagation

Figure 4.14: Sequence diagram for event propagation through node objects.

As mentioned earlier, events are created when we send the message

“setTo:” for a node or nodeVector and usually this is one of the last messages from a “behavior” method. Well then, we start this new sequence from the point where the previous one was interrupted and, by the way, is exactly the message “setTo:” sent to your node of output with a new value for node as argument. In node objects this message has in effect propagate this event to all components this node is with connected. First it is checked if the circuit is inserted in a schedulerMorph. In if not the output is updated but no event is propagated as there is no object of simulation coordination available. If so, a new entry in the list of schedulerMorph's dependencies is created, in which the list of components that

are linked to this node. Characterized by the iDrive list of node. Then for each iDrive list element is sent an eventBinder, an event descriptor that characterizes the event being propagated. The descriptor is then placed in a queue that is in the eventQueue slot of the comp objects.

4.3.2.3 Start of the ”start” simulation

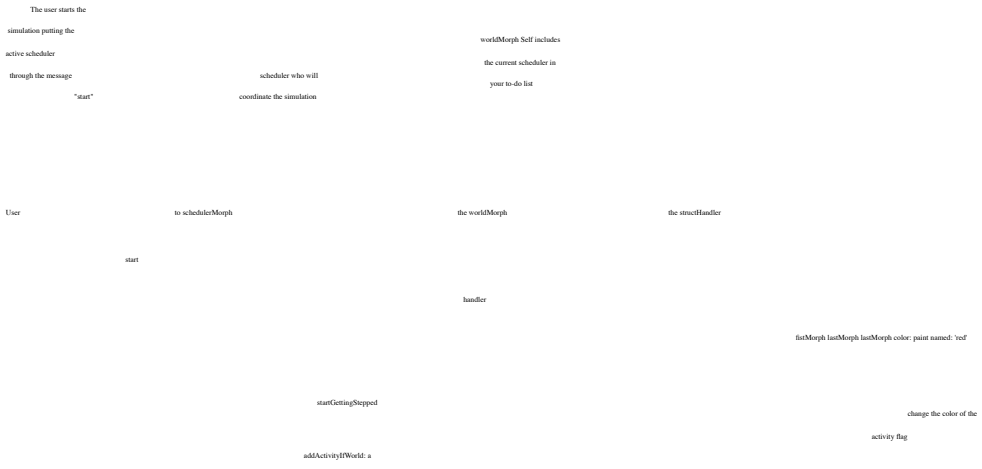


Figure 4.15: Sequence diagram for activating a simulation.

Another interesting sequence to be analyzed concerns the start of a simulation interactive. The interactive simulation is done by registering the highest level schedulerMorph in the list of updates of the Self environment, this is done by sending the message “start” to this schedulerMorph as shown in the sequence in figure 4.15. Upon receiving the message, the schedulerMorph's first step is to visually signal that it is "on", by changing the color of the flag element in your structHandler. the natural color of the structHandler is replaced by a bright red one. The next step is to request your registration next to the worldMorph object that contains it.

4.3.2.4 End of ”stop” simulation

The completion of the simulation is analogous to the previous sequence, see figure 4.16 . The function consists of in requesting the removal of the registration and this is done by sending the "stop" message to the schedulerMorph. As in the previous sequence, in the first step we return `the original color of the structHandler's flag element and then ask to remove the register to the worldMorph object that contains the simulation system.

Figure 4.16: Sequence diagram for disabling a simulation.

4.3.2.5 Advance of the ”stepping up” simulation

The most interesting sequences concern the simulation process itself
and will be analyzed below. In a first step, let's look at the case of a simulation num
single hierarchical level, that is, in which all components that make up the circuit are
elements described by their behaviors (comps), as is still the case in figure 4.12.

Figure 4.17: Sequence diagram for advancing the simulation of a single level.

The advance of interactive simulation starts by the initiative of the worldMorph object, which,

from time to time, it needs to update some objects it contains. These objects are in a list called activities. In the sequence of figure 4.17 we see the update of two objects: a gasTank, demonstrative object of the Self environment; and from a schedulerMorph. THE "step" message sent to the list objects takes care of all the updating. Upon receiving a "step", schedulerMorph selects the component from its dependency lists. to which the message must be forwarded, and so it does. The selected comp receives the "step" message and performs the following steps to update it: first remove a event descriptor from your event list (eventQueue), with this we guarantee that none event is missed and that all are evaluated in the correct order. These descriptors of events are used to effectively update the component's input ports. Per Lastly, the "behavior" method is executed, which finishes updating the internal state of this component.

To analyze a multi-level simulation situation, we need to consider a system. theme a little different from the initial one. In place of one of the internal components of the schedulerMorph we have a structural component, an sComp. As seen in the figure 4.18 .

Figure 4.18: Typical situation of a multi-level simulation.

In this new situation the component selected to receive the "step" message is a sComp, in this case the RS latch is structurally described. As in the previous case, the sComp needs to update its internal state to reflect the change in one of its entries. However, unlike comp objects, sComp does not have a "behavior" method. to that end, they are instead described by an internal schedulerMorph that has the description of the contained circuit. So the update also follows two steps: the first

step is also to remove an event descriptor from the component's event list and update the states of the circuit's input ports in the internal schedulerMorph. Lastly, the responsibility for distributing the “steps” rests with the internal schedulerMorph. This time, the component must be fully updated, so theoretically this schedulerMorph should receive as many "step" messages as necessary to guarantee the total propagation of the event in its internal components. This is done through the message "longStep", message that monitors schedulerMorph's dependency lists and generates “step” messages to yourself, until all lists are exhausted. This sequence of messages is seen in figure 4.19..

Figure 4.19: Sequence diagram for advancement in multi-level simulation.

A normal interactive simulation situation is therefore composed of the chain of several of these sequences in an appropriate order.

4.4 Description and Simulation

We will present some simple examples to quickly illustrate the expressiveness of the system.
SelfHDL theme, more complete and elaborate examples can be found in the next
chapter. The first point to note is that although the comps have some equivalence.
lence with “ENTITY” of the VHDL are not exactly the same thing. In SelfHDL there is

a clear distinction between behavior and structure, unlike VHDL. While
in VHDL parallel statements are also considered descriptions of behavior,
in SelfHDL they are structural in nature and are modeled by objects of type sComp.

4.4.1 Combinatorial and Sequential

Combinatorial circuits are easily described in SelfHDL.
in the description, all outputs are updated on the occurrence of an event, as it can be
seen in the “parity” example. If any output is not updated, it will function as a
memory of the previous state, for example:

```
comp name: 'PCreg'
      Inputs: [| Praça              <- nodeVector newType: std_unsigned Size: 32. clk |]
    Outputs: [| curr <- nodeVector newType: std_unsigned Size: 32.
                        next <- nodeVector newType: std_unsigned Size: 32 |]
  Behavior: [(isRising: clk) ifTrue:
                          [curr setTo: pc.
                              next setTo: pc + 4]]
```

In this example the outputs are only updated on the rising edge of the input signal.
clk. The component generated by this message is a 32-bit wide and sensitive register.
`the rising edge of the main clock. On the occurrence of this event, the curr output is loaded.
with the value presented in the input pc and the output next is loaded with that value plus four.

Other times we need to describe components with unassociated internal states to the outgoing ports. In these cases, the comp objects have a slot for storing state that can be used to store any kind of information. Slot initiation of status is made by the following message:

```
comp name: 'ram16x16'
    Inputs: [| adr <- nodeVector newType: std_unsigned Size: 4.
                    dat <- nodevector newType: std_unsigned Size: 16.
                    mrd. mwr. clk|]
   Outputs: [| dot <- nodeVector newType: std_unsigned Size: 16 |]
     State: [| mem <- vector copySyze: 16 FillingWith:
                                        ('0000000000000000' asNodeVectorType: std_unsigned)|]
Behavior: [| address |
                    (isRising: clk) ifFalse: [self].
                    address: adr toInteger.
                    mrd ifTrue: [dot setTo: (mem at: address)].
                    mwr ifTrue: [mem at: address Put: dat copy]]
```

Note that the difference between the two previous examples is the inclusion of the “State:” argument. In this argument, the type of information that should work as an internal state is specified.

Figure 4.20: First stage of DLX architecture pipeline.

of the component. In the previous example, the internal state is a vector of 16 positions each. of them containing a 16-bit wide nodeVector. The functioning of this component becomes clear when we look at the description of behavior.

4.4.2 Interactive Simulation

As stated earlier, the components of a SelfHDL description are live objects, once created and properly connected, these can be simulated as if they were the real circuit. A group of auxiliary objects called stimulators and observers are used to inject events and record the evolution of the state of nodes and components during the simulation. Unlike other simulators, these objects are not just registers. of results, but real elements of the interaction. Can be used as switches, keys, displays and many other things according to convenience. are able to give an emulation dimension to the simulation process.

Figure 4.20 shows the first pipeline stage of a DLX architecture as it is described in [HP96]. The figure shows a typical interactive simulation situation, where the circuit is evaluated before being admitted as the final sComp, as seen in figure 4.6. There is three stimulators and two observers in figure 4.20 : Two switches “mclk” and “branch”, and a single-line editor used to enter the branch address in format hexadecimal, are the stimulator objects. Switches can be turned on and off to advance the simulation step-by-step interactively. To enter an address

of branch, the new value must be typed in the block marked by “Branch Address”. You observers show the address used to access the instruction's memory, and the instruction read in the previous step. Instruction memory consists of a modified comp to receive an initialization of an external file, in this case a compiled test program. Like

can be seen, the text/schematic approach provides much more information about what is being described rather than a simple textual description.

4.5 Conclusion

In this chapter we present details of the SelfHDL implementation. This system was designed to be as powerful as standard HDLs and flexible enough to outperform also the difficulties of description methods based on general purpose languages. Your part textual, part graphical approach, allows a new dimension to be explored of hardware description, above all, it adheres perfectly to the principles advocated by the DO methodology. In the next chapter we will explore the creation process in more detail. using SelfHDL.

Chapter 5

Using SelfHDL

C of the Designer Oriented Methodology. Being a hard-coded description language. This section we will take a deeper evaluation of the potential and SelfHDL ware, nothing better than evaluating the performance of this system through a real project. Before of this we need to remember that the use of any hardware description system needs that guidelines are followed in code development and description of truly stable systems in any implementation. Section 5.1 reminds us of some of these guidelines, some can seem elementary but when followed they eliminate a very large amount of pro- problems in the next steps of the project. Then, in section 5.2, we will establish some practical conventions, which we have been collecting over time that simplify reading and understanding of a SelfHDL description. Section 5.3 will describe what the creation of com- special components, both of project instrumentation elements and of elements

who need to interact with the world outside the Self environment. Finally, in section 5.4 an implementation of the open architecture of the DLX processor will be described, described in [HP96] and compared to a traditional implementation. We conclude the chapter with some final considerations.

5.1 Rules and suggestions for a good hardware description

The rules presented in this section are, of course, familiar to experienced designers development and uses traditional hardware description languages, however, as this work also has a didactic character and presents a new form of description,

we think it is opportune to remember them. After all, they will also be useful in our system. These rules were collected and brilliantly presented by [Yan02] and will be summarized to follow.

5.1.1 Cell Library

Some decisions must be made at the beginning of an ASIC project. The use of sets or synchronous or asynchronous resets, activated when in high or low level. Records and Flip-flops sensitive to level or edge, etc. Many of these decisions usually take into account the personal preferences of each designer. However, it is wiser to take into account first the cell library that will actually be used and write the code so that the automatic synthesis takes into account these dedicated cells. imagine by example that in the library there are only registers with asynchronous set/reset and that by convenience we write the code considering the same synchronous. An additional charge of

circuit will be added only to satisfy the description, the original controls
are disabled. Therefore, a good recommendation is, whenever possible, to write the code taking into account the characteristics of the library cells and that these decisions are the same for the entire project.

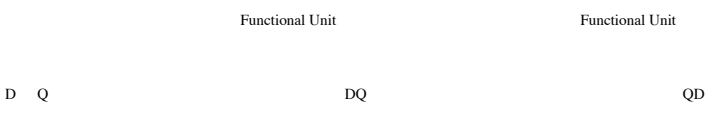
5.1.2 Data Transfer

Two clocks of different frequencies or of the same frequency but from different sources comes always to be considered as independent, that is, no pre-supposition as to timing between them can be done. The phase between them is simply unpredictable. Therefore, any data transfer involving these two systems must be done with very careful, generally taking into account the use of FIFOs when I pledge so to allow. When immediate transfer is required, the alternative is to secure in circuit the phase difference between the clocks.

5.1.3 Registered Inputs and Outputs

It is always convenient that the inputs of the various functional units come directly from registers and that their outputs are equally registered by a second set of

registrars. In this way, the timing of each stage can be controlled much more. easily. When the project allows, it is even possible to keep both inputs and outputs registered in the functional unit itself; thus, a data transfer between units can take an entire clock cycle, making traffic between distant units less critical as shown in figure 5.1



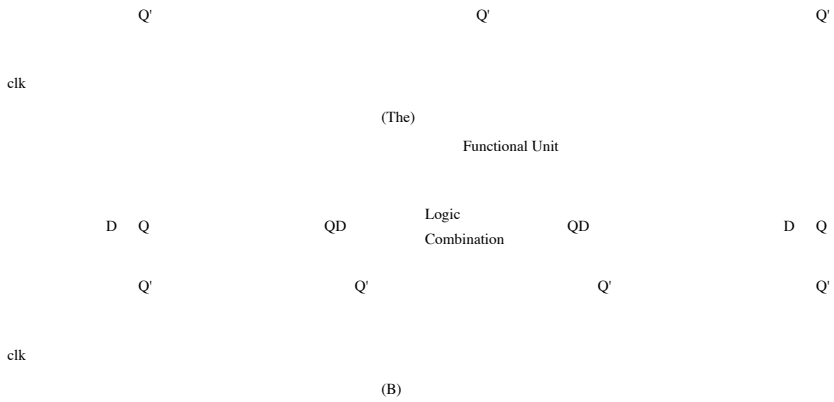


Figure 5.1: Inputs and outputs registered in different functional units help control better home the system timing: (a) All inputs coming from registered outputs; (B) Registering inputs and outputs in the functional unit itself.

5.1.4 Naming of Signals and Components

The naming of signals and components in a hardware project is very important because, in addition to documenting, it helps to understand its functionality. When the project is executed using a hardware description and automatic synthesis, a problem additional may occur. Many programs use the names given in the description. to flag and identify netlist elements. When the naming in the description is unfortunate, the generated netlist can present serious problems, for example, suppose that in a project there are two 16-bit bus and that one calls Databus and the other Databus1, suppose although somewhere we are referring to the signal “Databus1[5]”, that is, to the sixth element of the bus. The netlist generator might end up assigning to this signal the name “DATABUS15”, erroneously referring to the sixteenth signal of the first bus.

Other types of problems can also occur when using tools.
different natures, for example, when we jointly simulate systems described in
different languages, for example VHDL and Verilog. We must be careful not to use the
reserved words in a system as names or part of names of signals and components
on the other system. Other problems may involve special characters or the fact that a
system is case-insensitive, for example: VHDL is case-insensitive so it is very common
the designer assigns a signal such as MemChipSel designating a selection signal
of memory in a VHDL description. It is simple to read because the main words are
highlighted by capital initials. However, if this name is carried over to another
tool the name can be changed back to MEMCHIPSEL, which is more difficult to read.

Finally another confounding factor happens when, in order to document the most
a sign is clearly possible, we add words like “input” or “output” to the name, or part
of these words or equivalents in other languages. We must remember that the “InputMem” signal
is input from a given block but invariably will also be output from another. Re-
In summary, a common naming guideline for signals and components must be established.
for the entire project and must be strictly followed by all the designers of the group. Happily,
SelfHDL does not suffer from any of these problems internally, as no reference.
internal is done through names. However, as we intend to integrate this
system to other tools these tips are equally useful.

5.1.5 Not using Tri-States

The use of tri-state signals internally in integrated circuits must be followed by a
series of cautions by the designer. Signals that are left “floating” are sources of
a very wide range of potential problems. A “floating” sign is literally undiscovered.
known, this means non-definition of its logical state and consequently the consumption of
high power in certain regions of the circuit. So when this feature is needed
it is always recommended to use bus-holders on tri-state signals.

On the other hand, the cost of a failure of an already manufactured ASIC has increased steadily.
due to the evolution of manufacturing technologies, for this reason it is recommended
for the project to be more exhaustively verified through field-programmable gate arrays
(FPGAs). As these components generally do not have tri-state interconnection buses.

us, it is recommended that the original project also does not use this feature so that there is no adaptations between checks. Therefore, we should not use tri-state in a project, with exception of the ASIC's own input and output ports.

5.1.6 Simplification of Complex Operations

In a hardware description often port and connection delays are neglected. in function of the interest to be only in the logical verification of the implementation. It's common therefore, we describe expressions in which we combine sums and products of vectors very large as a single operation. For the synthesis tools, however, this is pro- problematic because often, in these situations, the latency of the operation exceeds the requirements of design clock. Therefore, it is always useful to keep in mind the synthesis method that will be adopted and future time constraints. It is therefore recommended that, in the case of complex actions, they are decomposed as if they were applied in a pipeline, even facilitating the understanding of these operations.

Another similar restriction concerns decisions: complex decisions tend to be synthesized in slow circuits. Therefore, the use of very complex decisions when describing a system.

5.1.7 Auxiliary and Test Circuits

As stated earlier, the cost of locating and fixing a defect after the component having already been manufactured has increased considerably in recent years. Not there is a foolproof way to ensure that a circuit is 100% correct if not with the adoption. tion of a very well elaborated system of tests and a good set of stimuli.

Often, it is necessary to include additional circuits to assist the test.
of integrated circuits (testability) and/or help access internal points of the com-

component in order to assist the fault diagnosis work. In these cases, there are also

several widely known and disseminated techniques of testing subsystems.

general recommendation that these techniques be defined at the beginning of the project and common to the entire

system so that there is uniformity in the verification methods and tools and, above

everything, compatibility and coverage between them.

5.2 Tips and conventions for using SelfHDL

In this section we will present some tips and conventions that we have adopted over time and

which has been shown to be useful in the description of components using SelfHDL. Some

conventions originate from the Self environment itself, others are features of the

of SelfHDL. The use of these guidelines will greatly facilitate future exchange

projects and the understanding of them by different teams. We hope that this section

be just a starting point for other norms and conventions as the system

SelfHDL is evolving.

5.2.1 Organizing a project

We know that a description in SelfHDL consists of a set of SelfH- objects

DL interconnected to describe and simulate certain digital hardware. As such, this

can be done in many ways and therefore a minimum convention must be established for

facilitate the exchange and understanding of this description by different teams. One more

instead the rules presented in this section and in the subsequent ones will be placed in the form of recognition.

mentions, since none of them interfere in the functioning of the SelfHDL system.

We can start talking about project organization and storage. In fact, the pro-
"Self world" itself constitutes the description and simulation environment of SelfHDL, therefore
when we "save" the environment at the end of a working day we will be saving the state
total environment up to that time. However, we notice that sometimes this is not yet
enough. We often use scripts to create new components and other
sometimes during the development itself, these scripts evolve into more elaborate versions
and/or complex. For this reason, whenever possible, it is recommended that the main
scripts of a project are stored in a special object, project repository. fu-
Naturally, this repository object can be associated with a Self module and distributed to
other images.

This is a practice that we have adopted since the beginning of development. An example
simple are scripts that define basic logic gates and some simple components. Those
scripts are stored in the traits comp object, so we can create one of these
components by simply sending a message with the script name to the comp object

for example, "comp nand". This message creates a component, a port nand of two
inputs, from a simple description of behavior. This feature was used in
several previous examples when we needed to quickly from a component to a
any purpose. In principle, scripts can be located anywhere in the environment.
Self and, in this case, the traits comp object serves as an anchor, as it establishes a point of
reference in the Self environment namespace. The comp object is known and how it is
heir to traits comp, scripts also become known.

Figure 5.2: Example of a repository object created in globals.

In the case of a generic project, it is recommended that a repository object be created. in the globals object. The globals object is the starting point from which all objects (prototypes) of the system are registered. That is, an object known as globals is co-known to almost all objects of the Self. The globals are organized into categories and subcategories, so we can add or remove slots from this object in a very orderly way. nothing. It is recommended, therefore, that a category be created for the project that will be developed and then a slot that reflects (or identifies) the design. In this way, every time a script needs to be reused, just send the message to this repository.

message “script name” and it will be executed. Figure 5.2 shows an example of a repository object created for the example that will be used later in this chapter.

5.2.2 Block coding conventions

When we describe a component through its behavior, there are some cautions.

of what we can take to make the simulation process more efficient. one of them says

specifically with respect to sequential components, typically components such as registers.

tracers, register bank, memories, and state machines evolve from state later

of an activity on a clock signal. Therefore, it is highly recommended that any

type of specific dependency is tested already in the first lines of the behavior description.

so that it is not necessary to evaluate the entire “behavior” message so that the object

is considered up to date. An example can be seen below:

```
comp name: 'PCreg'
      Inputs: [| pc <- nodeVector newType: std_unsigned Size: 32
                        clkld |]
    Outputs: [| curr <- nodeVector newType: std_unsigned Size: 32.
                        next <- nodeVector newType: std_unsigned Size: 32 |]
  Behavior: [ (isRising: clkld) ifTrue:
                          [curr setTo: pc.
                              next setTo: pc + 4 ]]
```

We saw in the example above that the first line of the behavior description is a test

about the load sign. It is a synchronous register with the rising edge of the

clock; therefore, if this was not the activation event, there is no need to evaluate

the message to its end, returning immediately after the message is evaluated

“ifTrue:”.

5.2.3 Circuit design

In this section we will give some tips on how to improve the appearance of the connections of a description.

structural relationship. Basically, we are referring to the use of connection objects. we know

from the previous chapter that a connection is formed by a set of branches, each branch.

ch connecting to a component input. The first branch always starts from the element

“drive” of the connection node, that is, of the component that determines the state of the node. The others

branches depart from one of the “elbows” of a previous branch. And finally, in this position a dot is placed to signal the division of “paths”. This feature when well used can make the connections look very nice. We will see below some care must be taken to ensure that it is used properly.

In figure 5.3, we see some connections. The first branch will always be 1, 3 or 5 segments according to the position of the first destination, figure 5.3(The). The other branches will be, by default, 2 or 4 segments. Each segment is counted from the beginning of the branch starting from 0. This value will be used to calculate the offset, origin point of the branch and position where the point will be drawn, later being stored in pairs with the branch number in the segmEvent slot of the connection objects, as can be seen in figure 4.8 .

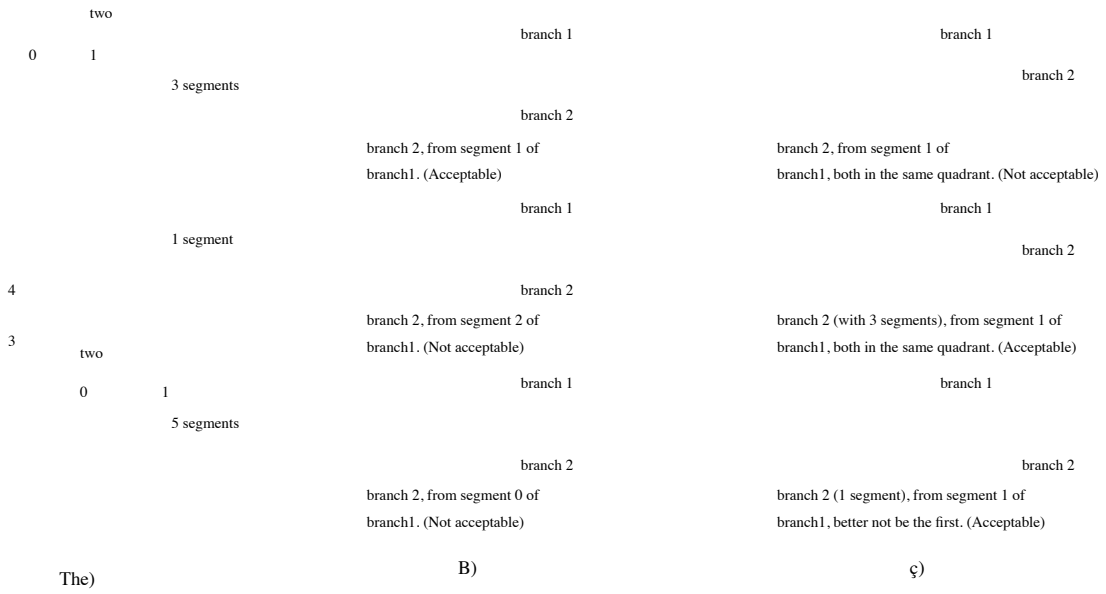


Figure 5.3: a) First branch has an odd number of segments. b) Second branch must start at an appropriate point. c) Examples of multiple branches.

We can see in figure 5.3 (b) that if an inappropriate segment is chosen to start the new branch, we might have a point in an odd position. Not that it interferes with the functionality of the connection, it is only a matter of aesthetic value. case the colon end are in the same quadrant, it is recommended that the new branch also have a

odd number of segments. This is done using the middle mouse button menu, "increaseSegs" option. In the opposite case, when possible, the number of segments by choosing the "decreaseSegs" option. In figure 5.3 (c) we see an example of this case. The two endpoints are in the same quadrant with respect to the point of origin,

the second branch originates from segment 1 of branch1, however, as it is not a primary branch, it receives only two segments which give it a weird final appearance. Through the "increaseSegs" menu option we can change branch2 from two to three segments, giving it a more natural appearance. On the other hand, when one of the end points is perfectly aligned with the origin (horizontally), it is recommended that the branch that makes this connection is not the first in the connection, if it is the first, it would only have one segment, so all other branches could only be drawn to from the very origin of the connection, which would also be a little strange. we remember more since these observations have a purely aesthetic character, and in any way that if branches of a connection are built, the connectivity between components will be always correct.

5.3 Implementation of special components

In this section we will present how the special components of the SelfH- system are created.

DL We refer by special to components that, in some way, deviate from the standard presented in the previous chapter. They are divided into two categories: basically they are the instrumentation elements, that is, the components we call "Observers" and "Es-stimulators" and specially adapted components with special features. Those components are all variations of the comp objects and their adaptations will be seen next.

5.3.1 Observers

Observers are the components used to “observe” the internal state of the elements.
of the SelfHDL system. In terms of implementation, we can say that are comps that
have only inputs and no outputs. Because they have no outputs, these
elements do not introduce additional events into a SelfHDL simulation system, only
record the events that are passed to them during the session. This record can be
of any kind possible and imaginable within the capabilities of the language Self. That is why,
the analysis potential of a SelfHDL system is only limited by the imagination.
and user programming capability, since the Self language is extremely
powerful. This work proposes some basic observers and others specially designed.

for use in the example project. In the future, it is intended to offer a wide range
of possibilities allowing a large number of possible analyses, thus reducing the
need for a great knowledge of Self on the part of designers. In this way it is-
we will, once again, reaffirm our belief in the importance and efficiency of the methodology
designer-oriented.

From the beginning we have highlighted the qualities of the interactive simulation of SelfHDL, because so much so, nothing more natural than to start talking about the most elementary type of observer, the bitProbe object. The hierarchy that a bitProbe object fits into has already been presented in figure 4.11. This is a good example, as all observers follow the same principle.
function: a parent* object is created for a particular type of observer and
it adapts the desired analysis functionality. In the case of bitProbe the func-
ality is to reflect, in color form, the logical level of the signal connected to its input. already the
vectorProbe objects, need to represent the state of an entire vector of nodes, so the

chosen means of representation was the text. Node states are read and their respective values are converted into a symbol and concatenated into a string of characters and written over the representation element. As we said, the kind of post-processing we can apply to inbound events is almost unlimited. For example, in figure 4.20 is shown an observer whose input is a 32bit vector, which we assume is a DLX instruction, and that we highlighted in Figure 5.4.

Figure 5.4: Detail of figure 4.20 where the observers of interest are shown.

So, for the kind of verification we wanted at that time it was more interesting, that instead of a bit vector represented in binary or hexadecimal, we could see exactly the instruction we were reading from memory. That's why this observer, the dlxInstrViewer, was created as a specialization of vectorProbe objects. The object dlxInstrViewer parent has a method that is basically a disassembler, the method

instrString. It composes a string representing the DLX instruction based on the value of the input vector (in), its implementation can be seen below:

```
instrString = (| fpcode.opcode.spcode.str |
    opcode: {in copyFrom: 0 UpTo: 5) toInteger.
    spcode: (in copyFrom: 26 UpTo: 31) toInteger.
    fpcode: (in copyFrom: 27 UpTo: 31) toInteger.
    opcode < 2 ifTrue: [
        opcode = 0 ifTrue: [
```

```
        str: (spcodes at: spcode), ' ', specialArgs: spcode.
    ] False: [
        str: (fpcodes at: fpcode), ' ', floatArgs: fpcode.
    ]] False: [
        str: (opcodes at: opcode), ' ', opcodeArgs: opcode.
    ].
str)
```

The opcodes, fpcodes and spcodes objects are string vectors also contained in dlxIntrViewer parent and contains the name of the instructions as specified in tables A.1 and A.2 . The functions “specialArgs:”, “floatArgs:” and “opcodeArgs:” compose the arguments (records, destinations, and immediate values) according to the respective code*.

5.3.2 Stimulators

Stimulators are objects designed to “inject” events into the simulation process. So like observers, stimulators are a specialization of comp objects, they have outputs (sources of events in the simulation environment) but no input. The "inputs" of the stimulators, are in fact elements external to the SelfHDL environment, such as stimulus files or direct interactions with the designer or other programs. In the figure 4.13 we saw how the user can generate an event from outside the simulation environment, in this case by pressing a button on the switch object. This example was remembered for being part of the interactive simulation process, one of the strengths of the SelfHDL system. However, there are other ways to promote a simulation, there are cases where the simulation must be done in the background or without user intervention, usually when it involves large amounts of input and/or output information, or when the operation extends for a very large number of cycles. In these cases it is also possible to use stimulators, this time associated with input files of stimuli and registered in the

* Note that a Self routine is so simple that there is almost no need to document it with comments.

worldMorph object to receive “steps” regularly. Figure 5.5 shows the diagram of sequence in this situation.

Figure 5.5: Sequence diagram of a non-interactive stimulator.

Figure 5.6: Sequence diagram of a non-interactive simulation.

Figure 5.6 shows the sequence diagram of the non-interactive simulation. Note that the non-interactive stimulator element is inside schedulerMorph, as it generates events from simulation in its list of dependencies. However, the stimulator does not run the risk of

receive the “step” message from this schedulerMorph because, because it has no entries

on the SelfHDL system, it will never be in that scheduler's dependency list.

The interaction with files external to the Self environment is also very simple, in the next section we will talk about other possibilities of special components where this feature is also very useful. We will leave to talk about this possibility in that section to avoid repetitions unnecessary.

A last comment regarding the instrumentation components would be the possibility of of mixing observers and stimulators in the same component. Nothing prevents that we “mix” the two functions in the same component as long as it is to operate on the interactive mode. This would be useful when there was a need to create a special interface. for the simulation system that involved the manipulation of non-technical individuals or unlearned in the SelfHDL system. The restriction to be in interactive mode can be explained easily by looking at the sequence diagram in figure 5.6.. In non-interactive, if the stimulator is eventually an observer too, there will be occasions in that it should receive “steps” by schedulerMorph instead of by worldMorph. Gives the way the system is currently implemented, there is no way to distinguish from where comes the message “step”, so all sorting provided by schedulerMorph would be compromised, thus compromising the simulation process.

5.3.3 Other Components

Another component that does not fit into the categories presented above are the memories. We saw that comp objects have a specific parent slot to store internal states not associated with the outputs. In section 4.4.1 we saw an example of an implementation. 16x16 RAM using this resource. In the case of simple components, this is a

very viable way to use SelfHDL and describe the component; however, in the case
of large RAMs or with special features this feature needs a
adaptation. A typical example will be used in the example project that will be presented in
next section. In the example of the DLX architecture, we have two memories one of instructions and
another of data. The use of distinct memories does not constitute a waste or an eccentricity,
it can be access to a specific cache of instructions and another of data. Anyway, the
instruction memory must contain a program to be executed in the implementation, for
on the other hand the data memory must offer the possibility of being able to store in a file

output the result of the simulation of a program. That is, both memories need
from having access to files external to the Self environment to read or write to those files. Per
on the other hand, memories tend to be quite large components and if we use them, as in
example from section 4.4.1 , an array with the exact size of memory we will be wasting
system resources and compromising its performance considerably. THE
The solution was the implementation of a special component, memoryFile.

In reality, the memoryFile implementation is very simple, like everything else in
SelfHDL. Once again, because there are no types in Self, the parent slot sta-
te* of comp can be used to store any object. In the example in section 4.4.1
a vector was used, in the memoryFile object we use a dictionary. the dictionary is
a Self object similar to the vector, but indexed by another object. internally he
it is composed of two vectors: one for the indices and one for the contents. the dictionaries
implement all regular value access and replacement operations as well as
insertion when necessary. This way, instead of storing all the positions
of a memory, we can store only the occupied memory positions. That is,
an assembler program that has some trap addresses and interrupts at positions

close to zero, but actually starting around address 2000h, will only store the actual occupied positions of memory. Because it allows the insertion of new values, memory can also be used as a writing element and its content can be stored in an output file.

Figure 5.7, shows a memoryFile in a test situation where the conditions present were evaluated. In this figure we can see the memoryFile object loaded initially with the test program presented in Appendix C . Handling input signals rd and wr, we can read the memory locations that will be seen through the dlxIntrViewer object, we can also write in already existing positions or positions not “occupied” by reading the result in the next read cycle. We can also see that the parent slot state* points to an object containing a single slot, the mem, which in turn points to a Dictionary of 32 elements. Each corresponding to a line of the object-program. As has been done so far, the specialization of a component is done using a parent object, where the specific methods are stored. Still in figure 5.7 we can see the memoryFile object parent* and the main implemented methods. The message that

Figure 5.7: Testing a memoryFile.

loads the program in the dictionary of this component is “load: fileName”, it is reproduced

Next:

```
load: fileName = (|f |
    f: os_file openForReading: fileName.
    [f atEOF] whileFalse: [| line. ad. val |
        line: f readLine.
        ad: addressPartOf: line.
        val: valuePartOf: line.
        mam at: ad Put: val ].
    (close)
```

We see that string objects themselves are used as dictionary elements.

In this way we optimize the component's operation, converting it to SelfHDL elements.

only during read accesses. For this, the access messages were defined:

"memoryAt: adr" and "memoryAt: adr Put: val", presented below:


```
memoryAt:adr Put:val = (
    mem at: (adr asHexString)
        Put: (val asHexString))
```

```
memoryAt:adr = (| lst.m.str |
    m: mem at: str asHexString
      IfAbsent: [| l <- '' |
                          (of length / 4) of:
                              [l:l, '0'].
                          l].
    lst: (m hexAsInteger) asDigitList: 16.
    str: ''.
    (m size) - (lst size) do: [lst: lst addFirst: 0].
    lst do: [| :d |
        str: str, (digitHexaString at: d)].
    ^str asNodeVectorType: std_unsigned)
```

With this we cover most of the features of the SelfHDL system. The next step

only then is it necessary to follow a complex project and demonstrate, in practice, the advantages of this

designer-oriented methodology.

5.4 Example Project: DLX Processor

In this section we will present an example project using the SelfHDL system. One question-

so somewhat complicated to be answered concerns the level of complexity that can be

we would use in this work. On the one hand, we have limited space to present

the problem and the solution, on the other hand we must use a problem complex enough to

convincingly demonstrate the proposed methodology. In principle, the SelfHDL system

is capable of handling any level of complexity. Our experience covers a range

relatively wide range of applications, ranging from telecommunication circuits to the architecture of

computers. The use of telecommunication circuits, however, would be problematic.

ca because not all possible people interested in this work have the same background in this

application field. A very detailed description of the problem would be needed before

that it would be possible to propose a solution. On the other hand, computer architecture is a

field of wide knowledge, both for students and researchers in the field of engineering

electrical and computer science. Therefore, the most logical choice is without a doubt the

computer architecture.

We chose the DLX processor architecture, described in [HP96]. This is a process

sador well known in academic circles, and there are several works involving this

architecture. The use of the DLX processor also allows a comparison to be possible
with other works or the use of SelfHDL as a didactic tool in courses already
established. We will be taking as a basis for comparison, the VHDL (RTL) implementation
proposed by [Ash04] And reproduced in part in Appendices B and C . The implementation of
[Ash04] is very didactic and easy to understand, so it will be a comparison
very good version of the SelfHDL system thus validating the proposed system.

5.4.1 DLX Architecture

The DLX processor is a RISC ("Reduced Instruction Set Computer") 32 architecture
bits proposed by [HP96] for strictly didactic purposes. It's a case study, in
which main concepts of computer architecture can be implemented and
checked. Not so it can be said that the DLX is "simplified", we can apply on the
its basic definition concepts as radical and modern as super-scalar and multi-
tithread. As main features we can mention: the use of registers on purpose
general in a typically load-store architecture; designed to be implemented effectively.
knowingly pipelined through the use of easy-to-decode instructions; architecture
RISC, designed to be efficiently exploited by compilers.

DLX has 32 general purpose registers of 32 bits each (GPRs), 32 registers
32-bit for single-precision floating-point numbers (FPRs), or 16 registers
64 bits each for double-precision floating-point numbers. The data types are
bytes (8 bits), half-word (16 bits) and words (32 bits), single-float (32 bits) and double-float (64
bits). Loads and Stores can also be done in 8, 16 or 32 bits. There are four modes
of addressing, being two real (16 bit immediate and offset) and two effective (relative
the register and absolute). The instructions have a fixed length of 32 bits and are encoded
in 6-bit fields, as shown in tables A.1 and A.2.

We chose to use the DLX pipeline implementation as an example. We believe that

in it we have the level of complexity necessary to demonstrate the qualities of the system SelfHDL without the need to fill the pages of this work with technical details of its functioning. Most DLX instructions can be implemented through a five-stage pipeline, shown in figure 5.8. We decided to leave out the operations of floating point by not adding anything new, apart from the work itself, in relation to

to the basic implementation.

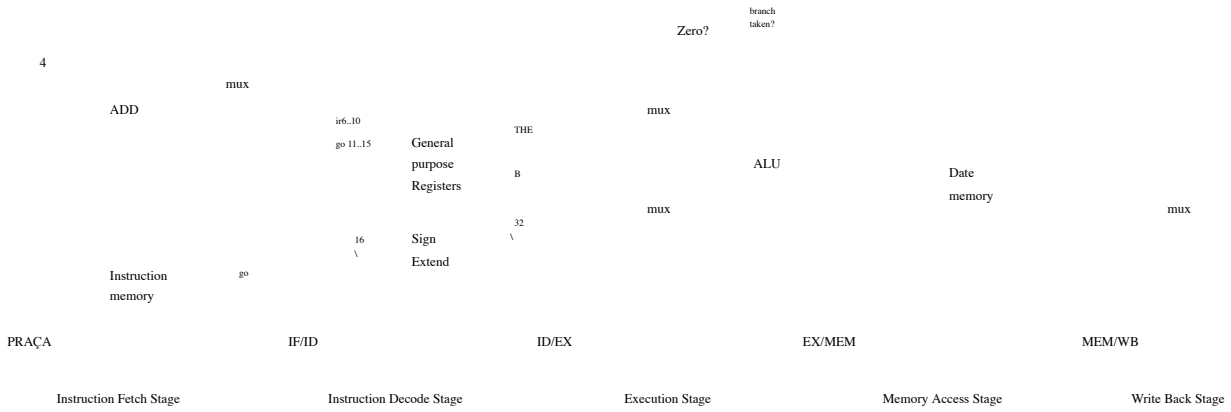


Figure 5.8: DLX pipeline architecture.

- Instruction Fetch: or instruction fetch is characterized by the following operations:

  IR ← Mem[PC]; NPC ← PC + 4 †. That is, it searches for an instruction from memory and stores it in the IR register which is to be used in subsequent stages.

  Stores the address of the next instruction in the NPC register; remembering that the DLX addresses bytes, the current value must be added by 4 to "point" to the next position.

- Instruction Decode and Register Fetch: or instruction decoding and fetch

of register: A ← Regs[IR $_{6..10}$], B ← Regs[IR $_{11..15}$], Imm ← ((IR $_{16}$) $_{16}$ ##IR $_{16..31}$). Decodes the instruction fields and accesses the register bank (GPR) storing the values read in the temporary registers A and B. Transforms the value 16-bit immediate into 32, sign-extending appropriately, the result is stored in the temporary Imm registry. Because the instructions have fields fixed, the decoding can be done in a totally parallel way.

• Execution or Effective Address: or Execution or effective address calculation. THE ULA (or ALU, "Arithmetic Logic Unit") uses the operands to perform one of the following operations:

– Memory Reference: ALUoutput ← A+Imm, ALU sums its operands

† We are following the notation suggested by [HP96].

to determine the effective memory access address; the result is stored in the ALUoutput temporary record.

– Operation with registers: ALUoutput ← AopB, the ALU performs the operation selected on the operands A and B; the result is stored in ALUoutput.

– Register and immediate operation: ALUoutput ← AopImm.

– Deviations: ALUoutput ← NPC + Imm, Cond ← (Aop0), the ALU sums the value of NPC with Imm to calculate bypass address; in parallel, the A record is checked to determine whether the deviation should be taken. The type of comparison is determined by the “opcode” field.

• Memory Access or Branch Completion: only DLX load operations, store and branches use this stage.

– Memory Access: LMD ← Mem[ALUoutput] or Mem[ALUoutput] ← B, if the instruction is read (load), the data memory is read in the position previously calculated and stored in ALUoutput and recorded in the temporary LMD register. If the access is write (store), the value of register B is written in the pointed position. by ALUoutput.

– Deviation: If (cond) PC ← ALUoutput else PC ← NPC, if the instruction is from bypass, the PC register is loaded with the new bypass address if the "cond" is true, or with the value of the next instruction if it is false.

• Write Back: writes the result in the register bank ending the instruction. The result can come from ALUoutput or LMD.

– Operation with registers: Regs[IR $_{16..20}$] ← ALUoutput.

– Register and immediate operation: Regs[IR $_{11..15}$] ← ALUoutput.

– Memory read operation: Regs[IR $_{11.15}$] ← LMD.

Figure 5.9 shows the filling of the pipeline stages in an ideal situation.

	clock cycle number								
Instruction Number	1	two	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction i + 1		IF	ID	EX	MEM	WB			
Instruction i + 2			IF	ID	EX	MEM	WB		
Instruction i + 3				IF	ID	EX	MEM	WB	
Instruction i + 4					IF	ID	EX	MEM	WB

Figure 5.9: Example of an ideal sequence of instructions in the DLX pipeline.

5.4.2 First stage of pipeline

One of the great advantages of the SelfHDL system is its ability to describe a system. digital textual and graphical form simultaneously. Taking advantage of this ability we can use the description in figure 5.8 as the basis for our SelfHDL description, thus the understanding of it becomes naturally more didactic and intelligible than a purely textual description. Figure 5.10 shows the SelfHDL description of the first stage DLX pipeline.

Figure 5.10: Instruction search stage in the DLX pipeline.

In the description of this stage we use four blocks described through the respective behaviors. They are: the PC and NPC register represented through the component PCreg, start of the DLX pipeline. Instruction memory, implemented by an object memoryFile, the output register of the stage that holds the newly read instruction and the address

of the next instruction, which can be used in the following stages, and the multiplexer of selection of the next address. Some generation scripts for these components are shown Next:

```
memoryFile name: 'Instruction Memory'
                Inputs: [| ad <- nodeVector newType std_unsigned Size: 32. rd |]
               Outputs: [| do <- nodeVector newType srd_unsigned Size: 32 |]
                 State: [| mem <- dictionary copyRemoveAll |]
              Behavior: [rd ifTrue:
                                [from setTo: memoryAt: ad]]

comp name: 'IFcontrol'
    Inputs: [| ir <- nodeVector newType: std_unsigned Size: 32.
                    pc <- nodeVector newType: std_unsigned Size: 32.
                    braddr <- nodeVector newType: std_unsigned Size: 32.
                    branch |]
   Outputs: [| npc <- nodeVector newType: std_unsigned Size: 32 |]
 behavior: [| i26. op |
                op: (go coptFrom: 0 Upto: 5) toInteger.
                i26: (go copyFrom: 6 UpTo: 31) toSigned.
                (op = 2) || [op = 3] ifTrue:
                    [^npc setTo: (i26 resizeTo: 32) + pc].
                (op = 17) ifTrue:
                    [^npc setTo: i26 resizeTo: 32].
                ifTrue branch:
                    [^npc setTo:braddr].
                npc setTo: pc]
```

The PCreg creation script was already presented in section 5.2.2, the output register of stage is analogous. Note that the next address selection multiplexer was also associated with an additional functionality to anticipate the decoding of the read instruction and determine in advance whether it is a branch or a trap. A typical test setup for this block was shown in figure 4.20.

5.4.3 Second stage of pipeline

As in the first stage, the operator search stage is also simple and can also be based on figure 5.8. The SelfHDL implementation is sitting in figure 5.11, in it we see four blocks as in the previous stage. The ID/EX block is the pipeline record of this stage, the GPR component would be the main block, it is the DLX architecture general purpose register bank. Another interesting block is which extends the immediate value to 32 bits, taking into account the sign of the

Prohibited. Finally, we have the IDcontrol block, which simply decomposes the fields of the instruction to obtain the addresses of the operands and/or retrieve the immediate value.

Figure 5.11: Stage of decoding and searching for operands in the pipeline registers of the DLX.

Some of the scripts used to create the component blocks are presented below:

```
comp name: 'GPR'
    Inputs: [| a1 <- nodeVector newType: std_unsigned Size: 5.
                a2 <- nodeVector newType: std_unsigned Size: 5.
                a3 <- nodeVector newType: std_unsigned Size: 5.
                wd <- nodeVector newType: std_unsigned Size: 32. rd. wr |]
```

```
    Outputs: [| rs1 <- nodeVector newType: std_unsigned Size: 32.
                rs2 <- nodeVector newType: std_unsigned Size: 32. |]
      State: [| mem <- vector copySize: 32
                        FillingWith: ('00000000000000000000000000000000'
                                asNodeVectorType: std_unsigned) |]
Behavior: [ rd ifTrue:
                [rs1 setTo: (mem at: a1 toInteger).
                 rs2 setTo: {mem at: a2 toInteger)].
            wr ifTrue: [| adr |
                 adr: a3 toInteger.
                 (adr = 0) ifFalse: [mem at: adr Put: wd copy]]]
```

Note that there are occasions when the GPR is read and written in the same cycle: in figure 5.9 cycle 5, while the instruction “i” performs a write back the instruction “i+3” performs a search

of operators. For the system to work correctly, we will agree that the readings are done in the second half of the clock cycle and that writings are done in the first.

Here are some more scripts.

```
comp name: 'Sign Extend'
     Inputs: [| imm <- nodeVector newType: std_signed Size: 16 |]
    Outputs: [| out <- nodeVector newType: std_signed Size: 32 |]
  Behavior: [ out setTo: (imm resizeTo: 32) ]

comp name: 'IDcontrol'
     Inputs: [| go <- nodeVector newType: std_unsigned Size: 32 |]
    Outputs: [| rs1 <- nodeVector newType: std_unsigned Size: 5.
                rs2 <- nodeVector newType: std_unsigned Size: 5.
                imm <- nodeVector newType: std_unsigned Size: 16 |]
  Behavior: [ rs1 setTo: go copyFrom: 6 UpTo: 10.
              rs2 setTo: go copyFrom: 11 UpTo: 15.
              imm setTo: go copyFrom: 16 UpTo: 31 ]
```

5.4.4 Third stage of pipeline

Figure 5.12 shows the execution stage and calculation of memory references of DLX.

This stage is the most complex of the implementation because a series of different operations

were implemented in various combinations of operands. In the figure, we can see the two sets of input multiplexers, the pipeline output register and the verification of “zero”. The most important blocks, however, are: the decoder block of control and the block that performs the DLX operations. The control decoder is a block generated by the script shown below.

```
comp name: 'EXcontrol'
      Inputs: [| go <- nodeVector newType: std_unsigned Size: 32 |]
    Outputs: [| iop <- nodeVector newType: std_unsigned Size: 5.
                        max. mxb. zrst. zinv |]
  Behavior: [| option |
                        opc: (go copyFrom: 0 UpTo: 5) toInteger.
                        (opc = 0) ifTrue: [ ^setForRtype ].
                        (opc > 31) && [opc < 44] ifTrue: [ ^setForLoadStore ].
                        (opc = 4) ifTrue: [ zinv setTo: node copyLow. ^setForBranch ].
                        (opc = 5) ifTrue: [ zinv setTo: node copyHigh. ^setForBranch ].
                        setForRImm].
```

We can observe an interesting phenomenon that also happens in SelfHDL and that already had previously aroused our attention in relation to textual descriptions. That is, textual descriptions tend to be long. According to the Self philosophy, a good method is that that can be “understood” in just a few lines of code. Therefore, the methods

Figure 5.12: Execution stage or address calculation in the DLX pipeline.

in Self are, in general, relatively small in relation to the listings of other languages.
of programming. We would like to apply the same principle to SelfHDL, however, this
It's not always possible. In implementing the EXcontrol object and some other objects
of this stage, the respective listings made the size of the components relatively
large because the description of the behavior is always placed in full in the representation
graphic of the same. One way around this problem is to factor the functionality into
helper methods, as was done in the EXcontrol component. One of the helper methods is
shown in the following listing, the others are very similar for this reason we do not judge
need to list them here.

```
setForRtype = (| one. sel. str. zero |
                    one: node copyHigh. zero: one not.
                    sel: (go copyFrom: 26 UpTo: 28) toInteger.
                    (sel = 0) ifTrue: [ str: '01' ].
                    (sel = 2) ifTrue: [ str: '10' ].
                    (sel = 5) ifTrue: [ str: '11' ]
                                        False: [ str: '00' ].
                    iop setTo: (go copyFrom: 29 UpTo: 31), str.
                    max setTo: zero. mxb setTo: zero.
                    zrst setTo: one. zinv setTo: zero. self)
```

The DLX operation block is a structural block essentially composed of three others.
blocks: The ALU, logical and arithmetic unit, the Shifter and the Condition Setter. Everyone is

shown in figure 5.13 .

Figure 5.13: Detail of the DLX Execution Unit with Integers.

We can clearly see the problem of the length of descriptions in the ALU and Condition blocks.

tion Setter. The other two components that appear in the description are a multiplexer

to select one of the three functional units and an operation decoder.

The operation of the DLX operation block is easy to understand, just follow

The figure. For the purpose of illustration, we present the ALU generation script below.

```
comp name: 'ALU'
    Inputs: [| aport <- nodeVector newType: std_unsigned Size: 32.
               bport <- nodeVector newType: std_unsigned Size: 32.
               sel <- nodeVector newType: std_unsigned Size: 3 |]
   Outputs: [| out <- nodeVector newType: std_unsigned Size: 32|]
 Behavior: [| op |
              op: sel toInteger.
              (op = 0) ifTrue: [^out setTo:
                                   ((aport toSigned) + (bport toSigned))].
              (op = 1) ifTrue: [^out setTo: aport + bport].
              (op = 2) ifTrue: [^out setTo:
                                   ((aport toSigned) - (bport toSigned))].
              (op = 3) ifTrue: [^out setTo: aport - bport].
              (op = 4) ifTrue: [^out setTo: aport and: bport].
              (op = 5) ifTrue: [^out setTo: aport or: bport].
              (op = 6) ifTrue: [^out setTo: aport xor: bport].
              (op = 7) ifTrue: [^out setTo: (bport copyFrom: 0 UpTo: 15),
                                                  '0000000000000000']]
```

5.4.5 Fourth stage of pipeline

The fourth stage of the DLX pipeline is relatively simple. It consists of three blocks: a data memory, the pipeline register and the instruction decoder. The implementation SelfHDL is shown in figure 5.14 .

Figure 5.14: DLX pipeline memory access stage.

The data memory and instruction decoder generation scripts are shown Next.

```
memoryFile name: 'Data Memory'
        Inputs: [| di <- nodeVector newType: std_unsigned Size: 32.
                   ad <- nodeVector newType: std_unsigned Size: 32.
                   rd wr |]
       Outputs: [| do <- nodeVector newType: std_unsigned Size: 32|]
      Behavior: ["Simple Memory"
```

```
                rd ifTrue:
                    [from setTo:memoryAt:ad].
                wr ifTrue:
                    [memoryAt:ad Put:di]

comp name: 'MEMcontrol'
      Inputs: [| go <- nodeVector newType: std_unsigned Size: 32 |]
     Outputs: [| rd wr |]
   Behavior: [| option |
                    opc: (go copyFrom: 0 UpTo: 5) toInteger.
                    (opc = 35) ifTrue:
```



```
                        [rd setTo: node copyHigh.
                          ^wr setTo: node copyLow].
                    (opc = 43) ifTrue:
                        [rd setTo: node copyLow.
                          ^wr setTo: node copyHigh].
                    rd setTo: node copyLow.
                    wr setTo: node copyLow ]
```

5.4.6 Fifth stage of pipeline

The writeback stage is the simplest of all. It consists of only two components: the stage instruction decoder and a multiplexer that selects the value that will be "written back" in the GPR register bank. Its SelfHDL implementation is shown in figure 5.15 .

Figure 5.15: Writeback stage of the DLX pipeline.

The stage instruction decoder generation script is shown below.

```
comp name: 'WBcontrol'
    Inputs: [| go <- nodeVector newType: std_unsigned Size: 32. mclk |]
   Outputs: [| rd <- nodeVector newType: std_unsigned Size: 5. wr. sel |]
 Behavior: [| option |
                opc: (go copyFrom: 0 UpTo: 5) toInteger.
                (opc = 35) ifTrue: [sel setTo: node copyHigh.
                                    ^wr setTo:mclk].
                sel setTo: node copyLow.
                (opc = 0) ifTrue:
```



```
                [| scd |
                  scd: (go copyFrom: 26 UpTo: 28) toInteger.
                  rd setTo: go copyFrom: 16 UpTo: 20.
                  (scd = 1) ifTrue: [^wr setTo: node copyLow].
                  (scd = 3) ifTrue: [^wr setTo: node copyLow].
                  (scd = 6) ifTrue: [^wr setTo: node copyLow].
                  (scd = 7) ifTrue: [^wr setTo: node copyLow].
                  ^wr setTo:mclk
               ] False: [rd setTo: go copyFrom: 11 UpTo: 15].
               (opc > 7) && [opc < 16] ifTrue: [^wr setTo: mclk].
               (opc > 19) && [opc < 30] ifTrue: [^wr setTo: mclk].
               (opc > 47) && [opc < 54] ifTrue: [^wr setTo: mclk].
               wr setTo: node copyLow ]
```

5.4.7 Implementation Assessment

The integration of the various stages of the DLX pipeline into SelfHDL is shown in figure

5.16 . It can also be seen a typical interactive test configuration, in which each instruction can be followed as it propagates through the pipeline.

In Appendix B , we have two VHDL implementations of the DLX architecture, one implementation. tion of the basic architecture proposed by [Ash04] (section B.1) and an implementation pipeline proposed by [MPS04] (section B.2). In the implementation of section B.1 we only see two listings ‡ that give us an idea of the difficulty of understanding the implementation when we deal only with purely textual descriptions. We can safely say, that an experienced VHDL programmer would take about an hour to unravel them all the implementation details, considering that he already knew the architecture beforehand. DLX. On the contrary, a SelfHDL implementation immediately provides an overview of the system because the graphical representation transmits in a more direct and unambiguous way the whole topology in a single image. As we said earlier, SelfHDL implementations they can use other previous representations as a starting point, as well as appro- we use the block diagram of the pipeline implementation proposed by [HP96]. Thereby, even an inexperienced SelfHDL user does not have great difficulties to understand implementation in a few minutes.

In figure 5.16, DLX's SelfHDL implementation was prepared to reflect a typical condition where we would like to evaluate DLX. As it is a didactic architecture, I like we would, as educators, demonstrate to students the conditions of exception (hazards) to

‡ The complete implementations, all files, can be found in the respective references.

Figure 5.16: Implementation of the DLX pipeline architecture.

which are subject to some processor implementations. The implementation was then prepared to show the propagation of instructions within the pipeline chain. Thereby, it is possible to more easily understand and demonstrate the peculiarities of the implement. tion without the need to use time signal diagrams (timing), very common to simulators for other languages; Also without the need for post-processing on the output data in case we want some special graphical treatment on the signals generated.

The implementation of [MPS04] is an implementation closer to the one we are presenting. sitting down. We can compare the two more closely and assess the level of difficulty in using of this or that implementation. For example, the SelfHDL implementation of the first stage would correspond to the VHDL listing presented in section B.2.1 , that of the second stage to section B.2.2 , and so on. Likewise, we do not publish all files so that we don't take up too much space in this work. All files can be found in [MPS04].

5.5 Conclusion

We present in this chapter important considerations regarding hardware descriptions in general and particularly referring also to the proposed SelfHDL system. we show how to extend the basic functionality of system objects to enable virtually any level of analysis that is required. And finally we show a higher-level implementation to demonstrate the potential of the system in an example. p that goes far beyond simple logic gates. DLX architecture was chosen as example so that we can also show the didactic potential of the SelfHDL system.

Chapter 6

Conclusions and Future Motivations

IN THIS work we present the main guidelines of a new methodology of design of digital systems that we call "Designer Oriented Methodology", or simply, DO (Design Oriented) methodology. By this methodology,

we seek to eliminate from the design flow concepts and operations that are foreign to the domain of

application. This makes the user better use their time for the benefit of the project, rather than wasting it on peripheral or minor tasks. This is done with the help of computational tools specially designed to hide the aspects end-user undesirables. Another important point of the DO methodology is to favor the user whenever possible, that is, the design flow must be intuitive in the domain of application and adapt to the user's needs, and not the other way around as it usually happens. This makes the design task more accessible, reducing costs by not requiring large investments in training for its use.

To demonstrate the methodology, the SelfHDL system was implemented, a system of description of digital hardware that uses graphic and textual resources to make the description. The system is implemented in Self language, hence its name. Choosing a system description of hardware is due to the exploration of architecture and experimentation of a project to be one of the first stages of implementation, usually using a conventional hardware description language. This step is fully covered by SelfHDL system, which, in addition to having the main features of conventional systems, features, it also has innovative and unusual features in CAD tools. traditional ones. Thanks to the use of the Self language, it was possible to implement a

Figure 6.1: Example of the use of labels in a SelfHDL description.

system that behaves like an interpreted system; that is, in no time, it is necessary to compile some part of the description. On the other hand, the com- dynamic pilation of the Self environment makes its performance superior to most of the other interpreted languages. As a general purpose language, Self still allows any type of post-processing to be integrated into the SelfHDL description for that the final analysis is as accurate and comfortable as possible for the user. all this inside from the same environment. The current alternative foresees the use of several programming languages. such as C++, TCL and Perl in the same system [McK01], making the infrastructure of complex and extremely expensive project.

We were able to see the advantages of a graphical/textual description provided by the SelfHDL system. The ease of understanding the descriptions produced is evident when we observe a similar description, done in VHDL for example. In the examples, in this work, labels were not included in the SelfHDL descriptions, this was done by purpose not to induce the false conclusion that labels were an effective part of the description. In fact, being a programming language, Self also allows them to be input. comments in the textual descriptions. On the other hand, graphic descriptions can also have elements that are “neutral” to the simulation, but that help in the documentation of the project. schedulerMorph objects can contain neutral objects, as does the structHandler. We can take advantage of this property of the system to further improve the qualities of the SelfHDL description by inserting the objects along the graphical descriptions labelMorph, as seen in Figure 6.1.

There are many advantages of using Self as an implementation language. Not
we had the opportunity to mention them so far because we were focused on the pro-
specific properties of SelfHDL. We think it is opportune to comment on them in this chapter because
greatly reduce the potential of the system when considering them. We saw that in SelfHDL
the descriptions are made within the graphical environment of the Self, this could bring some limits.
when there was more than one person working on the same project. However, the
that was not mentioned is that the same “world” Self can be shared among more
of a user, using the X11 platform. This means that more than one designer can
work simultaneously on the same project. Another advantage is that, in addition to the capabilities,
graphical features and versatility, Self also presents a communication infrastructure in
network, enabling communication between machines or processes, something quite simple.

Finally, the biggest advantage of all is that we are starting a new line of research
and CAD toolset, having full control and know-how over their development.
development. We will see in section 6.2 the potential of the new tool and the work of
research that we intend to continue after the conclusion of this work. Believe-
mos that the proposed subjects may yield some master's work and maybe some
doctorates, thus resulting in a powerful system of development.

6.1 Disadvantages of the SelfHDL system

Like any computer system, the SelfHDL system also has some disadvantages.
counts. However, we are safe to say that most of these disadvantages are
just circumstantial due to the technological moment in which we find ourselves.

The main disadvantages refer to the hardware needs for its use.
SelfHDL uses the Self language, so it is limited to those platforms that
support this language. Originally, Self was developed to run on the
SPARC platform (Sun Microsystems), being later "ported" to the platform
PowerPC (Apple) by the authors themselves. There are some independent port initiatives.
for the PC(x86), Linux and Microsoft platform [Gli04], but without the same level of
bility. In this work we used a PowerBook G4 with 400MHz and 256MB of memory
which, so far, has offered satisfactory performance in applications and examples.

plos that we have elaborated. However, one of the concerns that can be raised
would refer to performance and memory usage when using the system in projects
much more sophisticated. Another limitation refers to the intensive use of the graphical interface. In
SelfHDL, it is convenient to use displays of large proportions to be able to distribute
to fit the graphics more comfortably. Note that in this work some of the
implementations have very high graphic density. It happens out of necessity
we need to print the figures and scale them so they can be inserted in this work.
and still be legible. A real implementation could use a more display.
wide eliminating this problem. At the moment this work is being published, already
We have 2.5GHz dual 64-bit personal use Power PC platforms and memory available.
up to 8GB, offering more than ten times the performance we are currently using.
And 23” displays with resolution four times higher than the current one. So it's safe to say that
performance to run a system like the one we are proposing already exists.
As for the availability of different platforms, we still recommend using one of the
official platforms (SPARC or PowerPC). However, we believe that a serious application
in Self would contribute even more to the development of this language, giving encouragement
for other teams to develop and stabilize the Self for other platforms.

Finally, the least of the disadvantages we can present concerns the improvements in
graphical interface of the system, which still needs to be improved. In time, however,
this should not be a disadvantage as it is just an intermediate stage of the process.
of development.

6.2 Future Motivations

The conclusion of this work does not constitute, however, the conclusion of the SelfHDL system.

We have a number of ideas that we'd like to present and that we'd like to follow through. complementing the system. The most immediate need would be a performance evaluation. of the simulation system and the comparison with a traditional system. this is an aspect that we ended up not including in this work due to the lack of time for its realization. Other item excluded due to lack of time, is the evaluation of the use of the tool. Most likely would be developed in an undergraduate or graduate course, taking advantage of the facilities

didactics that the system offers. This work would need about one to two years for the collection of data and a careful evaluation if it had to be included.

As other immediate works we could highlight, cosmetic improvements so general to eliminate the last of the disadvantages pointed out in the previous section. Will improve representation for finite state machines, possibly proposing a representation graphic of the state diagram in a similar way to the schematic representation. Will improve structural description, making parallel statements recognized, generating and automatically instantiating components without having to generate them individually as happens currently. And finally implement a VHDL (or Verilog) code generator that can be synthesized, so that the system can be inserted into a real project flow and used more effectively.

As longer term works we highlight the elaboration of representations of more high level and, possibly, the inclusion of algorithms and routines for High Level Sinthesys. Another possibility would be the inclusion of infrastructure for parallel processing, approvating the availability of hardware that we are seeing. We believe that with a relatively small effort, we can add this feature and enjoy better the capacity of the parallel machines that are currently appearing on the market. One another possibility would be the inclusion of formal verification features for the descriptions.

SelfHDL. And finally, we also thought about extending the principles of the DO Methodology for other application areas, demonstrating other ways to implement programs applications.

Appendix A

DLX Architecture Instructions

A.1 DLX Instructions

In this section we present the encoding of the opcode fields and special-functions used in the DLX instructions. The format of DLX instructions is shown in figure A.1. The field opcode is a 6-bit field along the least significant (left) side of the instruction.
The field that defines the special-functions is also a 6-bit field next to the plus side.
(right) of Type-R instructions (Register to Register operations).
The coding used in this work is coherent with the “dlxasm” program used for generate test programs and shown in Appendix C .

The encoding of the opcode field is presented in table A.1, and the encoding of the functions-specials in table A.2 .

Table A.1: DLX instructions according to “Opcode” field.

Opcode C the		Instruction Implemented		Opcode N o		Instruction Implemented	
000000	0	special	Yes	100000 32	LB	not	
000001	1	ff arith	not	100001 33	LH	not	
000010	two	J	Yes	100010 34	-	-	
000011	3	JAL	Yes	100011 35	LW	Yes	
000100	4	BEQZ	Yes	100100 36	LBU	not	
000101	5	BNEZ	Yes	100101 37	LHU	not	
000110	6	BFPT	not	100110 38	LF	not	
000111	7	BFPF	not	100111 39	LD	not	
001000	8	ADDI	Yes	101000 40	SB	not	
001001	9	ADDUI	Yes	101001 41	SH	not	
001010 10		I climbed	Yes	101010 42	-	-	
001011 11		Climb	Yes	101011 43	SW	Yes	
001100 12		ANDI	Yes	101100 44	-	-	
001101 13		ORI	Yes	101101 45	-	-	
001110 14		XORI	Yes	101110 46	SF	not	
001111 15		LHI	Yes	101111 47	SD	not	
010000 16		RFE	Yes	110000 48	SEQUI	Yes	
010001 17		TRAP	Yes	110001 49	SNEUI	Yes	
010010 18		JR	Yes	110010 50	SLTUI	Yes	
010011 19		JALR	Yes	110011 51	SGTU	Yes	
010100 20		SLLI	Yes	110100 52	SLEUI	Yes	
010101 21		-	-	110101 53	SGEUI	Yes	
010110 22		SRLI	Yes	110110 54	-	-	
010111 23		MRS	Yes	110111 55	-	-	
011000 24		SEQI	Yes	111000 56	-	-	
011001 25		SNEI	Yes	111001 57	-	-	
011010 26		SLTI	Yes	111010 58	-	-	
011011 27		SGTI	Yes	111011 59	-	-	
011100 28		SLEI	Yes	111100 60	-	-	
011101 29		SGEI	Yes	111101 61	-	-	
011110 30		-	-	111110 62	-	-	
011111 31		-	-	111111 63	-	-	

Table A.2: DLX instructions according to the “Special Function” field.

Function C	the	Instruction	Implemented	Function N	o	Instruction	Implemented
000000	0	NOP	Yes	100000	32	ADD	Yes
000001	1	-	-	100001	33	ADDU	Yes
000010	two	-	-	100010	34	SUB	Yes
000011	3	-	-	100011	35	SUBU	Yes
000100	4	SLL	Yes	100100	36	AND	Yes
000101	5	-	-	100101	37	OR	Yes
000110	6	SRL	Yes	100110	38	XOR	Yes
000111	7	MRS	Yes	100111	39	-	-
001000	8	-	-	101000	40	SEQ	Yes
001001	9	-	-	101001	41	SNE	Yes
001010	10	-	-	101010	42	SLT	Yes
001011	11	-	-	101011	43	SGT	Yes
001100	12	-	-	101100	44	SLE	Yes
001101	13	-	-	101101	45	SGE	Yes
001110	14	-	-	101110	46	-	-
001111	15	-	-	101111	47	-	-
010000	16	SEQU	Yes	110000	48	MOVI2S	not
010001	17	SNEU	Yes	110001	49	MOVS2I	not
010010	18	SLTU	Yes	110010	50	MOVF	not
010011	19	SGTU	Yes	110011	51	MOVD	not
010100	20	SLEU	Yes	110100	52	MOVFP2I	not
010101	21	SGEU	Yes	110101	53	MOVI2FP	not

010110 22	-	-	110110 54	-	-
010111 23	-	-	110111 55	-	-
011000 24	-	-	111000 56	-	-
011001 25	-	-	111001 57	-	-
011010 26	-	-	111010 58	-	-
011011 27	-	-	111011 59	-	-
011100 28	-	-	111100 60	-	-
011101 29	-	-	111101 61	-	-
011110 30	-	-	111110 62	-	-
011111 31	-	-	111111 63	-	-

A.2 DLX Instruction Format

DLX instructions have a fixed width of 32 bit to facilitate decoding, in a typically RISC configuration. They are divided into three basic groups: the instructions of type-I, which encompass all operations involving Registers and Immediate values, the R-type instructions, which are the operations of Registers to Registers, and the J-type instructions, which are basically branch instructions. These formats can be seen below:

Type-I Instructions

Opcode	lol	rd	Immediate Value
6 bits	5 bits	5 bits	16 bits

Encodes Load and Store instructions of bytes, words and half words,
and all operations involving immediate values (rd <– rs1 immediate op)
Conditional branch instructions (rs1 is register, rd is not used)
Jump with register, jump and link with register. (rd = 0, rs1 = destination and im = 0)

Type-R Instructions

Opcode	lol	lol	rd	Special Function
6 bits	5 bits	5 bits	5 bits	11 bits

Register-to-register ULA operations: rd <– rs1 func rs2.

Special function encodes operation on data path: Add, Sub, ...

Read and Write to special registers and moves.

Type-J Instructions

Opcode	Displacement Added to PC
6 bits	26 bits

Jump and links

Trap and return exceptions.

Figure A.1: DLX Instruction Format.

Appendix B

VHDL implementations of DLX architecture

IN THIS appendix we present some files from two distinct implementations of DLX, which we use as a basis of comparison for the SelfHDL implementation sitting in this work. The first is a classic DLX implementation proposed by [Ash04], which is interesting because we can show the degree of difficulty in understanding that a VHDL description can achieve. From this implementation we take as an example only two files: the one that implements the integration of the various blocks, and the one that implements the controlley. We chose these files because they are extremely long and not very understandable. immediate. Unfortunately, this implementation is difficult to compare with the example project. in SelfHDL because it is not a pipeline.

In order to be able to compare more precisely, we use another implementation proposed by [MPS04]. This second implementation is pipeline and also seeks to follow the suggestion presented by [HP96]. It was initially conceived to be synthesized and implemented in FPGA and therefore uses reduced data/address pathways. Just right if possible, we adapted the implementation to contemplate 32-bit routes and be closer of the proposed SelfHDL implementation.

In this appendix we present only some files of these implementations. the files remaining can be found in the respective references.

B.1 RTL implementation of DLX

B.1.1 RTL implementation of DLX

```
---------------------------------------------------------- ------------------------
--
--      Copyright (C) 1993, Peter J. Ashenden
--      Mail: Dept. Computer Science
--      University of Adelaide, SA 5005, Australia
--      email: petera@cs.adelaide.edu.au
--
--      This program is free software; you can redistribute it and/or modify
--      it under the terms of the GNU General Public License as published by
--      the Free Software Foundation; either version 1, or (at your option)
--      any later version.
--
--      This program is distributed in the hope that it will be useful,
--      but WITHOUT ANY WARRANTY; without even the implied warranty of
--      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.                See the
--      GNU General Public License for more details.
--
--      You should have received a copy of the GNU General Public License
--      along with this program; if not, write to the Free Software
--      Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
--
---------------------------------------------------------- ------------------------
--
--      Entity specification for DLX processor
--

use work.dlx_types.all,
        work.mem_types.all;

entity dlx is
    generic (Tpd_clk_out : Time;                        -- clock to output propagation delay
                debug : boolean := false;               -- controls debug trace writes
                tag : string := "";
                origin_x, origin_y : real := 0.0);
    port (phi1, phi2 : in bit;                          -- 2-phase non-overlapping clocks
                reset : in bit;                         -- synchronous reset input
                a :out dlx_address;                     -- address bus output
                d : inout dlx_word_bus bus;             -- bidirectional data bus
                halt : out bit;                         -- halt indicator
                width : out mem_width;                  -- byte/haldword/word indicator
                write_enable : out bit;                 -- selects read or write cycle
                mem_enable : out bit;                   -- starts memory cycle
                ifetch : out bit;                       -- indicates instruction fetch
                ready : in bit);                        -- status from memory system
end dlx;

use std.textio.all,
        work.images.image_hex,
        work.bv_arithmetic.all,
        work.dlx_instr.all,
        work.alu_types.all;

architecture rtl of dlx is

    component alu
        port (s1 : in dlx_word;
                s2 : in dlx_word;
                result : out dlx_word;
                latch_en : in bit;
                func : in alu_func;
                zero, negative, overflow : out bit);
    end component;

    reg_file component
        port (a1 : in dlx_reg_addr;                     -- port1 address
                q1 : out dlx_word;                      -- port1 read data
                a2 : in dlx_reg_addr;                   -- port2 address
                q2 : out dlx_word;                      -- port2 read data
                a3 : in dlx_reg_addr;                   -- port3 address
                d3 : in dlx_word;                       -- port3 write data
                write_en : in bit);                     -- port3 write enable
    end component;

    component latch
        port (d : in dlx_word;
                q : out dlx_word;
                latch_en : in bit);
    end component;

    component reg_1_out
        port (d : in dlx_word;
          q : out dlx_word_bus bus;
                latch_en : in bit;
                out_en : in bit);
    end component;

    component reg_2_out
        port (d : in dlx_word;
          q1, q2 : out dlx_word_bus bus;
                latch_en : in bit;
                out_en1, out_en2 : in bit);
    end component;

    component reg_3_out
        port (d : in dlx_word;
          q1, q2, q3 : out dlx_word_bus bus;
                latch_en : in bit;
                out_en1, out_en2, out_en3 : in bit);
    end component;

    component reg_2_1_out
        port (d : in dlx_word;
          q1, q2 : out dlx_word_bus bus;
                q3 : out dlx_word;
                latch_en : in bit;
                out_en1, out_en2 : in bit);
    end component;

    component mux2
        port (i0, i1 : in dlx_word;
                y : out dlx_word;
                sel : in bit);
    end component;

    component go
        port (d : in dlx_word;                                                  -- instruction input from memory
          immed_q1, immed_q2 : out dlx_word_bus bus;
                ir_out : out dlx_word;                                          -- instruction output to control
                latch_en : in bit;
                immed_sel1, immed_sel2 : in immed_size;                         -- select 16-bit or 26-bit immed
                immed_unsigned1, immed_unsigned2 : in bit; -- extend immed unsigned/signed
                immed_en1, immed_en2 : in bit);                                 -- enable immed const outputs
    end component;

    component controller
        port (phi1, phi2 : in bit;
                    reset : in bit;
          halt : out bit;
                width : out mem_width;
                write_enable : out bit;
                mem_enable : out bit;
          ifetch : out bit;
                ready : in bit;
                alu_latch_en : out bit;
                alu_function : out alu_func;
                alu_zero, alu_negative, alu_overflow : in bit;
                reg_s1_addr, reg_s2_addr, reg_dest_addr : out dlx_reg_addr;
                reg_write : out bit;
                c_latch_en : out bit;
                a_latch_en, a_out_en : out bit;
                b_latch_en, b_out_en : out bit;
                temp_latch_en, temp_out_en1, temp_out_en2 : out bit;
                iar_latch_en, iar_out_en1, iar_out_en2 : out bit;
                pc_latch_en, pc_out_en1, pc_out_en2 : out bit;
                mar_latch_en, mar_out_en1, mar_out_en2 : out bit;
                mem_addr_mux_sel : out bit;
                mdr_latch_en, mdr_out_en1, mdr_out_en2, mdr_out_en3 : out bit;
                mdr_mux_sel : out bit;
          ir_latch_en : out bit;
                ir_immed_sel1, ir_immed_sel2 : out immed_size;
          ir_immed_unsigned1, ir_immed_unsigned2 : out bit;
                ir_immed_en1, ir_immed_en2 : out bit;
                current_instruction : in dlx_word;
                const1, const2 : out dlx_word_bus bus);
    end component;

    signal s1_bus, s2_bus : dlx_word_bus;
    signal dest_bus : dlx_word;
    signal alu_latch_en : bit;
    signal alu_function : alu_func;
    signal alu_zero, alu_negative, alu_overflow : bit;
    signal reg_s1_addr, reg_s2_addr, reg_dest_addr : dlx_reg_addr;
    signal reg_file_out1, reg_file_out2, reg_file_in : dlx_word;
    signal reg_write : bit;
    signal a_out_en, a_latch_en : bit;
    signal b_out_en, b_latch_en : bit;
    signal c_latch_en : bit;
    signal temp_out_en1, temp_out_en2, temp_latch_en : bit;
    signal iar_out_en1, iar_out_en2, iar_latch_en : bit;
    signal pc_out_en1, pc_out_en2, pc_latch_en : bit;
    signal pc_to_mem : dlx_word;
    signal mar_out_en1, mar_out_en2, mar_latch_en : bit;
    signal mar_to_mem : dlx_word;
    signal mem_addr_mux_sel : bit;
    signal mdr_out_en1, mdr_out_en2, mdr_out_en3, mdr_latch_en : bit;
    signal mdr_in : dlx_word;
    signal mdr_mux_sel : bit;
    signal current_instruction : dlx_word;
    signal ir_latch_en : bit;
    signal ir_immed_sel1, ir_immed_sel2 : immed_size;
    signal ir_immed_unsigned1, ir_immed_unsigned2 : bit;
    signal ir_immed_en1, ir_immed_en2 : bit;
```

```
the_alu : alu
    port map (s1 => s1_bus, s2 => s2_bus, result => dest_bus,
                latch_en => alu_latch_en, func => alu_function,
                zero => alu_zero, negative => alu_negative,
                overflow => alu_overflow);

the_reg_file : reg_file
    port map (a1 => reg_s1_addr, q1 => reg_file_out1,
                a2 => reg_s2_addr, q2 => reg_file_out2,
                a3 => reg_dest_addr, d3 => reg_file_in,
                write_en => reg_write);

c_reg : latch
    port map (d => dest_bus, q => reg_file_in, latch_en => c_latch_en);

a_reg : reg_1_out
    port map (d => reg_file_out1, q => s1_bus,
                latch_en => a_latch_en, out_en => a_out_en);

b_reg : reg_1_out
    port map (d => reg_file_out2, q => s2_bus,
                latch_en => b_latch_en, out_en => b_out_en);

temp_reg : reg_2_out
    port map (d => dest_bus, q1 => s1_bus, q2 => s2_bus,
                latch_en => temp_latch_en,
                out_en1 => temp_out_en1, out_en2 => temp_out_en2);

iar_reg : reg_2_out
    port map (d => dest_bus, q1 => s1_bus, q2 => s2_bus,
                latch_en => iar_latch_en,
                out_en1 => iar_out_en1, out_en2 => iar_out_en2);

pc_reg : reg_2_1_out
    port map (d => dest_bus, q1 => s1_bus, q2 => s2_bus,
                q3 => pc_to_mem, latch_en => pc_latch_en,
                out_en1 => pc_out_en1, out_en2 => pc_out_en2);

mar_reg : reg_2_1_out
    port map (d => dest_bus, q1 => s1_bus, q2 => s2_bus,
                q3 => mar_to_mem, latch_en => mar_latch_en,
                out_en1 => mar_out_en1, out_en2 => mar_out_en2);

addr_mux_mux : mux2
    port map (i0 => pc_to_mem, i1 => mar_to_mem, y => a,
                sel => mem_addr_mux_sel);

mdr_reg : reg_3_out
    port map (d => mdr_in, q1 => s1_bus, q2 => s2_bus, q3 => d,
                latch_en => mdr_latch_en,
                out_en1 => mdr_out_en1, out_en2 => mdr_out_en2,
            out_en3 => mdr_out_en3);

mdr_mux : mux2
    port map (i0 => dest_bus, i1 => d, y => mdr_in,
                sel => mdr_mux_sel);

instr_reg : go
    port map (d => d, immed_q1 => s1_bus, immed_q2 => s2_bus,
                ir_out => current_instruction,
                latch_en => ir_latch_en,
            immed_sel1 => ir_immed_sel1, immed_sel2 => ir_immed_sel2,
            immed_unsigned1 => ir_immed_unsigned1,
                immed_unsigned2 => ir_immed_unsigned2,
                immed_en1 => ir_immed_en1, immed_en2 => ir_immed_en2);

the_controller : controller
    port map (phi1, phi2, reset, halt,
                width, write_enable, mem_enable, ifetch, ready,
                alu_latch_en, alu_function, alu_zero,
                alu_negative, alu_overflow,
                reg_s1_addr, reg_s2_addr, reg_dest_addr, reg_write,

    debug_s2 : if debug generate
        s2_monitor : process (s2_bus)
            variable L : line;
        begin
            write(L, tag);
            write(L, string'(" s2_monitor: "));
            write(L, image_hex(s2_bus));
            writeline(output, L);
        end process s2_monitor;
    end generate;

    debug_dest : if debug generate
        dest_monitor : process (dest_bus)
            variable L : line;
        begin
            write(L, tag);
            write(L, string'(" dest_monitor: "));
            write(L, image_hex(dest_bus));
            writeline(output, L);
        end process dest_monitor;
    end generate;

end rtl;
```

B.1.2 Controller

```
------------------------------------------------------- -------------------
--
--      Copyright (C) 1993, Peter J. Ashenden
--      Mail: Dept. Computer Science
--      University of Adelaide, SA 5005, Australia
--      email: petera@cs.adelaide.edu.au
--
--      This program is free software; you can redistribute it and/or modify
--      it under the terms of the GNU General Public License as published by
--      the Free Software Foundation; either version 1, or (at your option)
--      any later version.
--
--      This program is distributed in the hope that it will be useful,
--      but WITHOUT ANY WARRANTY; without even the implied warranty of
--      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.                See the
--      GNU General Public License for more details.
--
--      You should have received a copy of the GNU General Public License
--      along with this program; if not, write to the Free Software
--      Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
--
------------------------------------------------------- -------------------
--
--      Entity declaration for DLX control section.
--

use work.dlx_types.all,
        work.dlx_instr.all,
        work.alu_types.all,
        work.mem_types.all;

entity controller is
    generic (Tpd_clk_ctrl, Tpd_clk_const : Time;
                debug : boolean := false;
                tag : string := "";
                origin_x, origin_y : real := 0.0);
    port (phi1, phi2 : in bit;
            reset : in bit;
              halt : out bit;
              width : out mem_width;
              write_enable : out bit;
              mem_enable : out bit;
              ifetch : out bit;
              ready : in bit;
              alu_latch_en : out bit;
```

```
                c_latch_en, a_latch_en, a_out_en, b_latch_en, b_out_en,
                temp_latch_en, temp_out_en1, temp_out_en2,
                iar_latch_en, iar_out_en1, iar_out_en2,
                pc_latch_en, pc_out_en1, pc_out_en2,
                mar_latch_en, mar_out_en1, mar_out_en2, mem_addr_mux_sel,
                mdr_latch_en, mdr_out_en1, mdr_out_en2,
            mdr_out_en3, mdr_mux_sel,
                ir_latch_en, ir_immed_sel1, ir_immed_sel2,
            ir_immed_unsigned1, ir_immed_unsigned2,
            ir_immed_en1, ir_immed_en2,
                current_instruction, s1_bus, s2_bus);

        debug_s1 : if debug generate
            s1_monitor : process (s1_bus)
                variable L : line;
                begin
                    write(L, tag);
                    write(L, string'(" s1_monitor: "));
                    write(L, image_hex(s1_bus));
                    writeline(output, L);
                end process s1_monitor;
        end generate;
```

```
            alu_function : out alu_func;
            alu_zero, alu_negative, alu_overflow : in bit;
            reg_s1_addr, reg_s2_addr, reg_dest_addr : out dlx_reg_addr;
            reg_write : out bit;
            c_latch_en : out bit;
            a_latch_en, a_out_en : out bit;
            b_latch_en, b_out_en : out bit;
            temp_latch_en, temp_out_en1, temp_out_en2 : out bit;
            iar_latch_en, iar_out_en1, iar_out_en2 : out bit;
            pc_latch_en, pc_out_en1, pc_out_en2 : out bit;
            mar_latch_en, mar_out_en1, mar_out_en2 : out bit;
            mem_addr_mux_sel : out bit;
            mdr_latch_en, mdr_out_en1, mdr_out_en2, mdr_out_en3 : out bit;
            mdr_mux_sel : out bit;
            ir_latch_en : out bit;
            ir_immed_sel1, ir_immed_sel2 : out immed_size;
            ir_immed_unsigned1, ir_immed_unsigned2 : out bit;
            ir_immed_en1, ir_immed_en2 : out bit;
            current_instruction : in dlx_word;
            const1, const2 : out dlx_word_bus bus);
    end controller;

    use work.bv_arithmetic.all, std.textio.all;
```



```
architecture behavior of controller is

begin -- behavior

    sequencer : process

        alias IR_opcode : dlx_opcode is current_instruction(0 to 5);
        alias IR_sp_func : dlx_sp_func is current_instruction(26 to 31);
        alias IR_fp_func : dlx_fp_func is current_instruction(27 to 31);
        alias IR_rs1 : dlx_reg_addr is current_instruction(6 to 10);
        alias IR_rs2 : dlx_reg_addr is current_instruction(11 to 15);
        alias IR_Itype_rd : dlx_reg_addr is current_instruction(11 to 15);
        alias IR_Rtype_rd : dlx_reg_addr is current_instruction(16 to 20);
        alias IR_immed16 : dlx_immed16 is current_instruction(16 to 31);
        alias IR_immed26 : dlx_immed26 is current_instruction(6 to 31);

        variable IR_opcode_num : dlx_opcode_num;
        variable IR_sp_func_num : dlx_sp_func_num;
        variable IR_fp_func_num : dlx_fp_func_num;

        variable result_of_set_is_1, branch_taken : boolean;
        variable L : line;

        procedure bus_instruction_fetch is
        begin
            -- use PC as address
            mem_addr_mux_sel <= '0' after Tpd_clk_ctrl;
            -- set up memory control signals
            width <= width_word after Tpd_clk_ctrl;
            ifetch <= '1' after Tpd_clk_ctrl;
            mem_enable <= '1' after Tpd_clk_ctrl;
            -- wait until phi2, then enable IR input
            wait until phi2 = '1';
            ir_latch_en <= '1' after Tpd_clk_ctrl;
            -- wait until memory is ready at end of phi2
            loop
                wait until phi2 = '0';
                if reset = '1' then
    return;
                end if;
                exit when ready = '1';
            end loop;
            -- disable IR input and memory control signals
            ir_latch_en <= '0' after Tpd_clk_ctrl;
            mem_enable <= '0' after Tpd_clk_ctrl;
        end bus_instruction_fetch;

        procedure bus_data_read(read_width : in mem_width) is
        begin
            -- use MAR as addresses
```

```
        end bus_data_write;

        procedure do_set_result is
        begin
            wait until phi1 = '1';
            if result_of_set_is_1 then
const2 <= X"0000_0001" after Tpd_clk_const;
            else
const2 <= X"0000_0000" after Tpd_clk_const;
            end if;
            alu_latch_en <= '1' after Tpd_clk_ctrl;
            alu_function <= alu_pass_s2 after Tpd_clk_ctrl;
            --
            wait until phi1 = '0';
            alu_latch_en <= '0' after Tpd_clk_ctrl;
            const2 <= null after Tpd_clk_const;
            --
            wait until phi2 = '1';
            c_latch_en <= '1' after Tpd_clk_ctrl;
            --
            wait until phi2 = '0';
            c_latch_en <= '0' after Tpd_clk_ctrl;
        end do_set_result;

        procedure do_EX_set_unsigned(immed : boolean) is
        begin
            wait until phi1 = '1';
            a_out_en <= '1' after Tpd_clk_ctrl;
            if immed then
                ir_immed_sel2 <= immed_size_16 after Tpd_clk_ctrl;
ir_immed_unsigned2 <= '1' after Tpd_clk_ctrl;
ir_immed_en2 <= '1' after Tpd_clk_ctrl;
            else
                b_out_en <= '1' after Tpd_clk_ctrl;
            end if;
            alu_latch_en <= '1' after Tpd_clk_ctrl;
            alu_function <= alu_subu after Tpd_clk_ctrl;
            --
            wait until phi1 = '0';
            alu_latch_en <= '0' after Tpd_clk_ctrl;
            a_out_en <= '0' after Tpd_clk_ctrl;
            if immed then
ir_immed_en2 <= '0' after Tpd_clk_ctrl;
            else
                b_out_en <= '0' after Tpd_clk_ctrl;
            end if;
            --
            wait until phi2 = '0';
            if immed then
                case IR_opcode is
                    when op_sequi =>
```

```
        mem_addr_mux_sel <= '1' after Tpd_clk_ctrl;
        -- set up memory control signals
        width <= read_width after Tpd_clk_ctrl;
        ifetch <= '0' after Tpd_clk_ctrl;
        mem_enable <= '1' after Tpd_clk_ctrl;
        -- wait until phi2, then enable MDR input
        wait until phi2 = '1';
        mdr_mux_sel <= '1' after Tpd_clk_ctrl;
        mdr_latch_en <= '1' after Tpd_clk_ctrl;
        -- wait until memory is ready at end of phi2
        loop
            wait until phi2 = '0';
            if reset = '1' then
return;
            end if;
            exit when ready = '1';
        end loop;
        -- disable MDR input and memory control signals
        mdr_latch_en <= '0' after Tpd_clk_ctrl;
        mem_enable <= '0' after Tpd_clk_ctrl;
    end bus_data_read;

    procedure bus_data_write(write_width : in mem_width) is
    begin
        -- use MAR as addresses
        mem_addr_mux_sel <= '1' after Tpd_clk_ctrl;
        -- enable MDR output
        mdr_out_en3 <= '1' after Tpd_clk_ctrl;
        -- set up memory control signals
        width <= write_width after Tpd_clk_ctrl;
        ifetch <= '0' after Tpd_clk_ctrl;
        write_enable <= '1' after Tpd_clk_ctrl;
        mem_enable <= '1' after Tpd_clk_ctrl;
        -- wait until memory is ready at end of phi2
        loop
            wait until phi2 = '0';
            if reset = '1' then
return;
            end if;
            exit when ready = '1';
        end loop;
        -- disable MDR output and memory control signals
        write_enable <= '0' after Tpd_clk_ctrl;
        mem_enable <= '0' after Tpd_clk_ctrl;
        mdr_out_en3 <= '0' after Tpd_clk_ctrl;
```

```
        result_of_set_is_1 := alu_zero = '1';
            when op_sneui =>
                result_of_set_is_1 := alu_zero /= '1';
            when op_sltui =>
        result_of_set_is_1 := alu_overflow = '1';
            when op_sgtui =>
                result_of_set_is_1 := alu_overflow /= '1' and alu_zero /= '1';
            when op_sleui =>
                result_of_set_is_1 := alu_overflow = '1' or alu_zero = '1';
            when op_sgeui =>
                result_of_set_is_1 := alu_overflow /= '1';
    when others =>
        null;
            end case;
        else
            case IR_sp_employee is
                when sp_func_sequ =>
        result_of_set_is_1 := alu_zero = '1';
            when sp_func_sneu =>
                result_of_set_is_1 := alu_zero /= '1';
            when sp_func_sltu =>
        result_of_set_is_1 := alu_overflow = '1';
            when sp_func_sgtu =>
                result_of_set_is_1 := alu_overflow /= '1' and alu_zero /= '1';
            when sp_func_sleu =>
                result_of_set_is_1 := alu_overflow = '1' or alu_zero = '1';
            when sp_func_sgeu =>
                result_of_set_is_1 := alu_overflow /= '1';
    when others =>
        null;
            end case;
        end if;
        --
        do_set_result;
    end do_EX_set_unsigned;

    procedure do_EX_set_signed(immed : boolean) is
    begin
        wait until phi1 = '1';
        a_out_en <= '1' after Tpd_clk_ctrl;
        if immed then
            ir_immed_sel2 <= immed_size_16 after Tpd_clk_ctrl;
ir_immed_unsigned2 <= '0' after Tpd_clk_ctrl;
ir_immed_en2 <= '1' after Tpd_clk_ctrl;
        else
```



```
                b_out_en <= '1' after Tpd_clk_ctrl;
            end if;
            alu_latch_en <= '1' after Tpd_clk_ctrl;
            alu_function <= alu_sub after Tpd_clk_ctrl;
            --
            wait until phi1 = '0';
            alu_latch_en <= '0' after Tpd_clk_ctrl;
            a_out_en <= '0' after Tpd_clk_ctrl;
            if immed then
      ir_immed_en2 <= '0' after Tpd_clk_ctrl;
            else
                b_out_en <= '0' after Tpd_clk_ctrl;
            end if;
            --
            wait until phi2 = '0';
            if immed then
                case IR_opcode is
                    when op_seqi =>
        result_of_set_is_1 := alu_zero = '1';
                    when op_snei =>
                        result_of_set_is_1 := alu_zero /= '1';
                    when op_slti =>
        result_of_set_is_1 := alu_negative = '1';
                    when op_sgti =>
                        result_of_set_is_1 := alu_negative /= '1' and alu_zero /= '1';
                    when op_slei =>
                        result_of_set_is_1 := alu_negative = '1' or alu_zero = '1';
                    when op_sgei =>
                        result_of_set_is_1 := alu_negative /= '1';
```

```
            procedure do_EX_arith_logic_immed is
            begin
                wait until phi1 = '1';
                a_out_en <= '1' after Tpd_clk_ctrl;
                ir_immed_sel2 <= immed_size_16 after Tpd_clk_ctrl;
                if IR_opcode = op_addi or IR_opcode = op_subi then
                    ir_immed_unsigned2 <= '0' after Tpd_clk_ctrl;
                else
        ir_immed_unsigned2 <= '1' after Tpd_clk_ctrl;
                end if;
                ir_immed_en2 <= '1' after Tpd_clk_ctrl;
                alu_latch_en <= '1' after Tpd_clk_ctrl;
                case IR_opcode is
                when op_addi =>
                    alu_function <= alu_add after Tpd_clk_ctrl;
                when op_subi =>
                    alu_function <= alu_sub after Tpd_clk_ctrl;
                when op_addui =>
                    alu_function <= alu_addu after Tpd_clk_ctrl;
                when op_subui =>
                    alu_function <= alu_subu after Tpd_clk_ctrl;
                when op_andi =>
                    alu_function <= alu_and after Tpd_clk_ctrl;
                when op_ori =>
                    alu_function <= alu_or after Tpd_clk_ctrl;
                when op_xori =>
                    alu_function <= alu_xor after Tpd_clk_ctrl;
                when op_slli =>
```

```
when others =>
    null;
        end case;
    else
        case IR_sp_employee is
            when sp_func_seq =>
    result_of_set_is_1 := alu_zero = '1';
            when sp_func_sne =>
                result_of_set_is_1 := alu_zero /= '1';
            when sp_func_slt =>
    result_of_set_is_1 := alu_negative = '1';
        when sp_func_sgt =>
            result_of_set_is_1 := alu_negative /= '1' and alu_zero /= '1';
        when sp_func_sle =>
            result_of_set_is_1 := alu_negative = '1' or alu_zero = '1';
        when sp_func_sge =>
            result_of_set_is_1 := alu_negative /= '1';
when others =>
    null;
        end case;
    end if;
    --
    do_set_result;
  end do_EX_set_signed;

  procedure do_EX_arith_logic is
  begin
    wait until phi1 = '1';
    a_out_en <= '1' after Tpd_clk_ctrl;
    b_out_en <= '1' after Tpd_clk_ctrl;
    alu_latch_en <= '1' after Tpd_clk_ctrl;
    case IR_sp_employee is
  when sp_func_add =>
            alu_function <= alu_add after Tpd_clk_ctrl;
        when sp_func_addu =>
            alu_function <= alu_addu after Tpd_clk_ctrl;
        when sp_func_sub =>
            alu_function <= alu_sub after Tpd_clk_ctrl;
        when sp_func_subu =>
            alu_function <= alu_subu after Tpd_clk_ctrl;
        when sp_func_and =>
            alu_function <= alu_and after Tpd_clk_ctrl;
        when sp_func_or =>
            alu_function <= alu_or after Tpd_clk_ctrl;
        when sp_func_xor =>
            alu_function <= alu_xor after Tpd_clk_ctrl;
        when sp_func_sll =>
            alu_function <= alu_sll after Tpd_clk_ctrl;
        when sp_func_srl =>
            alu_function <= alu_srl after Tpd_clk_ctrl;
        when sp_func_sra =>
            alu_function <= alu_sra after Tpd_clk_ctrl;
        when others =>
            null;
    end case;                                   --  IR_sp_empc
    --
    wait until phi1 = '0';
    alu_latch_en <= '0' after Tpd_clk_ctrl;
    a_out_en <= '0' after Tpd_clk_ctrl;
    b_out_en <= '0' after Tpd_clk_ctrl;
    --
    wait until phi2 = '1';
    c_latch_en <= '1' after Tpd_clk_ctrl;
    --
    wait until phi2 = '0';
    c_latch_en <= '0' after Tpd_clk_ctrl;
  end do_EX_arith_logic;
```



```
alu_function <= alu_sll after Tpd_clk_ctrl;
    when op_srli =>
        alu_function <= alu_srl after Tpd_clk_ctrl;
    when op_srai =>
        alu_function <= alu_sra after Tpd_clk_ctrl;
    when others =>
        null;
  end case;                                     --  IR_opcode
  --
  wait until phi1 = '0';
  alu_latch_en <= '0' after Tpd_clk_ctrl;
  a_out_en <= '0' after Tpd_clk_ctrl;
  ir_immed_en2 <= '0' after Tpd_clk_ctrl;
  --
  wait until phi2 = '1';
  c_latch_en <= '1' after Tpd_clk_ctrl;
  --
  wait until phi2 = '0';
  c_latch_en <= '0' after Tpd_clk_ctrl;
end do_EX_arith_logic_immed;

procedure do_EX_link is
begin
  wait until phi1 = '1';
  pc_out_en1 <= '1' after Tpd_clk_ctrl;
  alu_latch_en <= '1' after Tpd_clk_ctrl;
  alu_function <= alu_pass_s1 after Tpd_clk_ctrl;
  --
  wait until phi1 = '0';
  alu_latch_en <= '0' after Tpd_clk_ctrl;
  pc_out_en1 <= '0' after Tpd_clk_ctrl;
  --
  wait until phi2 = '1';
  c_latch_en <= '1' after Tpd_clk_ctrl;
  --
  wait until phi2 = '0';
  c_latch_en <= '0' after Tpd_clk_ctrl;
end do_EX_link;

procedure do_EX_lhi is
begin
  wait until phi1 = '1';
  ir_immed_sel1 <= immed_size_16 after Tpd_clk_ctrl;
  ir_immed_unsigned1 <= '1' after Tpd_clk_ctrl;
  ir_immed_en1 <= '1' after Tpd_clk_ctrl;
  const2 <= X"0000_0010" after Tpd_clk_const;       -- shift by 16 bits
  alu_latch_en <= '1' after Tpd_clk_ctrl;
  alu_function <= alu_sll after Tpd_clk_ctrl;
  --
  wait until phi1 = '0';
  alu_latch_en <= '0' after Tpd_clk_ctrl;
  ir_immed_en1 <= '0' after Tpd_clk_ctrl;
  const2 <= null after Tpd_clk_const;
  --
  wait until phi2 = '1';
  c_latch_en <= '1' after Tpd_clk_ctrl;
  --
  wait until phi2 = '0';
  c_latch_en <= '0' after Tpd_clk_ctrl;
end do_EX_lhi;

procedure do_EX_branch is
begin
  wait until phi1 = '1';
  a_out_en <= '1' after Tpd_clk_ctrl;
  const2 <= X"0000_0000" after Tpd_clk_const;
  alu_latch_en <= '1' after Tpd_clk_ctrl;
```

```
alu_function <= alu_sub after Tpd_clk_ctrl;
--
wait until phi1 = '0';
alu_latch_en <= '0' after Tpd_clk_ctrl;
a_out_en <= '0' after Tpd_clk_ctrl;
const2 <= null after Tpd_clk_const;
```

```
  pc_latch_en <= '0' after Tpd_clk_ctrl;
end do_MEM_branch;

procedure do_MEM_load is
begin
  wait until phi1 = '1';
```

```
        --
        wait until phi2 = '0';
        if IR_opcode = op_beqz then
branch_taken := alu_zero = '1';
        else
branch_taken := alu_zero /= '1';
        end if;
    end do_EX_branch;

    procedure do_EX_load_store is
    begin
        wait until phi1 = '1';
        a_out_en <= '1' after Tpd_clk_ctrl;
        ir_immed_sel2 <= immed_size_16 after Tpd_clk_ctrl;
        ir_immed_unsigned2 <= '0' after Tpd_clk_ctrl;
        ir_immed_en2 <= '1' after Tpd_clk_ctrl;
        alu_function <= alu_add after Tpd_clk_ctrl;
        alu_latch_en <= '1' after Tpd_clk_ctrl;
        --
        wait until phi1 = '0';
        alu_latch_en <= '0' after Tpd_clk_ctrl;
        a_out_en <= '0' after Tpd_clk_ctrl;
        ir_immed_en2 <= '0' after Tpd_clk_ctrl;
        --
        wait until phi2 = '1';
        mar_latch_en <= '1' after Tpd_clk_ctrl;
        --
        wait until phi2 = '0';
        mar_latch_en <= '0' after Tpd_clk_ctrl;
    end do_EX_load_store;

    procedure do_MEM_jump is
    begin
        wait until phi1 = '1';
        pc_out_en1 <= '1' after Tpd_clk_ctrl;
        ir_immed_sel2 <= immed_size_26 after Tpd_clk_ctrl;
        ir_immed_unsigned2 <= '0' after Tpd_clk_ctrl;
        ir_immed_en2 <= '1' after Tpd_clk_ctrl;
        alu_latch_en <= '1' after Tpd_clk_ctrl;
        alu_function <= alu_add after Tpd_clk_ctrl;
        --
        wait until phi1 = '0';
        alu_latch_en <= '0' after Tpd_clk_ctrl;
        pc_out_en1 <= '0' after Tpd_clk_ctrl;
        ir_immed_en2 <= '0' after Tpd_clk_ctrl;
        --
        wait until phi2 = '1';
        pc_latch_en <= '1' after Tpd_clk_ctrl;
        --
        wait until phi2 = '0';
        pc_latch_en <= '0' after Tpd_clk_ctrl;
    end do_MEM_jump;

    procedure do_MEM_jump_reg is
    begin
        wait until phi1 = '1';
        a_out_en <= '1' after Tpd_clk_ctrl;
        alu_latch_en <= '1' after Tpd_clk_ctrl;
        alu_function <= alu_pass_s1 after Tpd_clk_ctrl;
        --
        wait until phi1 = '0';
        alu_latch_en <= '0' after Tpd_clk_ctrl;
        a_out_en <= '0' after Tpd_clk_ctrl;
        --
        wait until phi2 = '1';
        pc_latch_en <= '1' after Tpd_clk_ctrl;
        --
        wait until phi2 = '0';
        pc_latch_en <= '0' after Tpd_clk_ctrl;
    end do_MEM_jump_reg;

    procedure do_MEM_branch is
    begin
        wait until phi1 = '1';
        pc_out_en1 <= '1' after Tpd_clk_ctrl;
        ir_immed_sel2 <= immed_size_16 after Tpd_clk_ctrl;
        ir_immed_unsigned2 <= '0' after Tpd_clk_ctrl;
        ir_immed_en2 <= '1' after Tpd_clk_ctrl;
        alu_latch_en <= '1' after Tpd_clk_ctrl;
        alu_function <= alu_add after Tpd_clk_ctrl;
        --
        wait until phi1 = '0';
        alu_latch_en <= '0' after Tpd_clk_ctrl;
        pc_out_en1 <= '0' after Tpd_clk_ctrl;
        ir_immed_en2 <= '0' after Tpd_clk_ctrl;
        --
        wait until phi2 = '1';
        pc_latch_en <= '1' after Tpd_clk_ctrl;
        --
        wait until phi2 = '0';
        bus_data_read(width_word);
        if reset = '1' then
            return;
        end if;
        --
        wait until phi1 = '1';
        mdr_out_en1 <= '1' after Tpd_clk_ctrl;
        alu_function <= alu_pass_s1 after Tpd_clk_ctrl;
        alu_latch_en <= '1' after Tpd_clk_ctrl;
        --
        wait until phi1 = '0';
        mdr_out_en1 <= '0' after Tpd_clk_ctrl;
        alu_latch_en <= '0' after Tpd_clk_ctrl;
        --
        wait until phi2 = '1';
        c_latch_en <= '1' after Tpd_clk_ctrl;
        --
        wait until phi2 = '0';
        c_latch_en <= '0' after Tpd_clk_ctrl;
    end do_MEM_load;

    procedure do_MEM_store is
    begin
        wait until phi1 = '1';
        b_out_en <= '1' after Tpd_clk_ctrl;
        alu_function <= alu_pass_s2 after Tpd_clk_ctrl;
        alu_latch_en <= '1' after Tpd_clk_ctrl;
        --
        wait until phi1 = '0';
        b_out_en <= '0' after Tpd_clk_ctrl;
        alu_latch_en <= '0' after Tpd_clk_ctrl;
        --
        wait until phi2 = '1';
        mdr_mux_sel <= '0' after Tpd_clk_ctrl;
        mdr_latch_en <= '1' after Tpd_clk_ctrl;
        --
        wait until phi2 = '0';
        mdr_latch_en <= '0' after Tpd_clk_ctrl;
        --
        wait until phi1 = '1';
        bus_data_write(width_word);
    end do_MEM_store;

    procedure do_WB(Rd : dlx_reg_addr) is
    begin
        wait until phi1 = '1';
        reg_dest_addr <= Rd after Tpd_clk_ctrl;
        reg_write <= '1' after Tpd_clk_ctrl;
        --
        wait until phi2 = '0';
        reg_write <= '0' after Tpd_clk_ctrl;
    end do_WB;

begin -- sequencer
    --
    ---------------------------------------------------------- --------------
    -- initialize all control signals
    ---------------------------------------------------------- --------------
    if debug then
        write(L, string'("controller: initializing"));
        writeline(output, L);
    end if;
    --
    halt <= '0' after Tpd_clk_ctrl;
    width <= width_word after Tpd_clk_ctrl;
    write_enable <= '0' after Tpd_clk_ctrl;
    mem_enable <= '0' after Tpd_clk_ctrl;
    ifetch <= '0' after Tpd_clk_ctrl;
    alu_latch_en <= '0' after Tpd_clk_ctrl;
    alu_function <= alu_add after Tpd_clk_ctrl;
    reg_s1_addr <= B"00000" after Tpd_clk_ctrl;
    reg_s2_addr <= B"00000" after Tpd_clk_ctrl;
    reg_dest_addr <= B"00000" after Tpd_clk_ctrl;
    reg_write <= '0' after Tpd_clk_ctrl;
    c_latch_en <= '0' after Tpd_clk_ctrl;
    a_latch_en <= '0' after Tpd_clk_ctrl;
    a_out_en <= '0' after Tpd_clk_ctrl;
    b_latch_en <= '0' after Tpd_clk_ctrl;
    b_out_en <= '0' after Tpd_clk_ctrl;
    temp_latch_en <= '0' after Tpd_clk_ctrl;
    temp_out_en1 <= '0' after Tpd_clk_ctrl;
    temp_out_en2 <= '0' after Tpd_clk_ctrl;
    iar_latch_en <= '0' after Tpd_clk_ctrl;
    iar_out_en1 <= '0' after Tpd_clk_ctrl;
    iar_out_en2 <= '0' after Tpd_clk_ctrl;
    pc_latch_en <= '0' after Tpd_clk_ctrl;
    pc_out_en1 <= '0' after Tpd_clk_ctrl;
    pc_out_en2 <= '0' after Tpd_clk_ctrl;
    mar_latch_en <= '0' after Tpd_clk_ctrl;
    mar_out_en1 <= '0' after Tpd_clk_ctrl;
```

```
mar_out_en2 <= '0' after Tpd_clk_ctrl;
mem_addr_mux_sel <= '0' after Tpd_clk_ctrl;
mdr_latch_en <= '0' after Tpd_clk_ctrl;
mdr_out_en1 <= '0' after Tpd_clk_ctrl;
mdr_out_en2 <= '0' after Tpd_clk_ctrl;
mdr_out_en3 <= '0' after Tpd_clk_ctrl;
mdr_mux_sel <= '0' after Tpd_clk_ctrl;
ir_latch_en <= '0' after Tpd_clk_ctrl;
ir_immed_sel1 <= immed_size_16 after Tpd_clk_ctrl;
ir_immed_sel2 <= immed_size_16 after Tpd_clk_ctrl;
ir_immed_unsigned1 <= '0' after Tpd_clk_ctrl;
ir_immed_unsigned2 <= '0' after Tpd_clk_ctrl;
ir_immed_en1 <= '0' after Tpd_clk_ctrl;
ir_immed_en2 <= '0' after Tpd_clk_ctrl;
const1 <= null after Tpd_clk_const;
const2 <= null after Tpd_clk_const;
--
wait until phi2 = '0' and reset = '0';
--
------------------------------------------------------- ---------------
-- control loop
------------------------------------------------------- ---------------
loop
    --
    ------------------------------------------------------- ---------------
    -- fetch next instruction (IF)
    ------------------------------------------------------- ---------------
    wait until phi1 = '1';
    if debug then
        write(L, string'("controller: instruction fetch"));
        writeline(output, L);
    end if;
    --
    bus_instruction_fetch;
    --
    ------------------------------------------------------- ---------------
    -- instruction decode, source register read, and PC increment (ID)
    ------------------------------------------------------- ---------------
    wait until phi1 = '1';
    if debug then
        write(L, string'("controller: decode, reg-read and PC incr"));
        writeline(output, L);
    end if;
    --
    IR_opcode_num := bv_to_natural(IR_opcode);
    IR_sp_func_num := bv_to_natural(IR_sp_func);
    IR_fp_func_num := bv_to_natural(IR_fp_func);
    --
    reg_s1_addr <= IR_rs1 after Tpd_clk_ctrl;
    reg_s2_addr <= IR_rs2 after Tpd_clk_ctrl;
    a_latch_en <= '1' after Tpd_clk_ctrl;
    b_latch_en <= '1' after Tpd_clk_ctrl;
    --
    pc_out_en1 <= '1' after Tpd_clk_ctrl;
    const2 <= X"0000_0004" after Tpd_clk_const;
    alu_latch_en <= '1' after Tpd_clk_ctrl;
    alu_function <= alu_addu after Tpd_clk_ctrl;
    --
    wait until phi1 = '0';
    a_latch_en <= '0' after Tpd_clk_ctrl;
    b_latch_en <= '0' after Tpd_clk_ctrl;
    alu_latch_en <= '0' after Tpd_clk_ctrl;
    pc_out_en1 <= '0' after Tpd_clk_ctrl;
    const2 <= null after Tpd_clk_const;
    --
    wait until phi2 = '1';
    pc_latch_en <= '1' after Tpd_clk_ctrl;
    --
    wait until phi2 = '0';
    pc_latch_en <= '0' after Tpd_clk_ctrl;
    --
    ------------------------------------------------------- ---------------
    -- execute instruction, (EX, MEM, WB)
    ------------------------------------------------------- ---------------
    if debug then
        write(L, string'("controller: execute"));
        writeline(output, L);
    end if;
    --
    case IR_opcode is
  when special_op =>
        case IR_sp_employee is
            when sp_func_nop =>
                sp_func_slt | sp_func_sgt |
                sp_func_sle | sp_func_sge =>
                do_EX_set_signed(immed => false);
                do_WB(IR_Rtype_rd);
            when sp_func_movi2s =>
      assert false
        report "MOVI2S instruction not implemented" severity warning;
        when sp_func_movs2i =>
            assert false
        report "MOVS2I instruction not implemented" severity warning;
        when sp_func_movf =>
            assert false
        report "MOVF instruction not implemented" severity warning;
        when sp_func_movd =>
            assert false
        report "MOVD instruction not implemented" severity warning;
        when sp_func_movfp2i =>
            assert false
        report "MOVFP2I instruction not implemented" severity warning;
        when sp_func_movi2fp =>
            assert false
        report "MOVI2FP instruction not implemented" severity warning;
            when others =>
      assert false
        report "undefined special instruction function" severity error;
      end case;
    when op_fparith =>
        case IR_fp_func is
        when fp_func_addf | fp_func_subf | fp_func_multif | fp_func_divf |
                fp_func_addd | fp_func_subd | fp_func_multd | fp_func_divd |
                fp_func_mult | fp_func_multu | fp_func_div | fp_func_divu |
                fp_func_cvtf2d | fp_func_cvtf2i | fp_func_cvtd2f |
                fp_func_cvtd2i | fp_func_cvti2f | fp_func_cvti2d |
                fp_func_eqf | fp_func_nef | fp_func_ltf | fp_func_gtf |
                fp_func_lef | fp_func_gef | fp_func_eqd | fp_func_ned |
                fp_func_ltd | fp_func_gtd | fp_func_led | fp_func_ged =>
            assert false
        report "floating point instructions not implemented" severity warning;
            when others =>
      assert false
        report "undefined floating point instruction function" severity error;
end case;
        when op_j =>
            do_MEM_jump;
        when op_jr =>
            do_MEM_jump_reg;
        when op_jal =>
            do_EX_link;
            do_MEM_jump;
            do_WB(natural_to_bv(link_reg, 5));
        when op_jalr =>
            do_EX_link;
            do_MEM_jump_reg;
            do_WB(natural_to_bv(link_reg, 5));
    when op_beqz | op_bnez =>
            do_EX_branch;
if branch_taken then
                do_MEM_branch;
            end if;
    when op_bfpt =>
            assert false
    report "BFPT instruction not implemented" severity warning;
    when op_bfpf =>
            assert false
    report "BFPF instruction not implemented" severity warning;
    when op_addi | op_upi |
                op_addui | op_subui |
                op_andi | op_ori | op_xori |
                    op_slli | op_srli | op_srai =>
            do_EX_arith_logic_immed;
            do_WB(IR_Itype_rd);
    when op_lhi =>
            do_EX_lhi;
            do_WB(IR_Itype_rd);
    when op_rfe =>
            assert false
    report "RFE instruction not implemented" severity warning;
    when op_trap =>
            assert false
    report "TRAP instruction encountered, execution halted"
    severity note;
            wait until phi1 = '1';
            halt <= '1' after Tpd_clk_ctrl;
```

```
                null;
            when sp_func_sequ | sp_func_sneu |
                    sp_func_sltu | sp_func_sgtu |
                    sp_func_sleu | sp_func_sgeu =>
                do_EX_set_unsigned(immed => false);
                do_WB(IR_Rtype_rd);
            when sp_func_add | sp_func_addu |
              sp_func_sub | sp_func_subu |
                sp_func_and | sp_func_or | sp_func_xor |
                sp_func_sll | sp_func_srl | sp_func_sra =>
                do_EX_arith_logic;
                do_WB(IR_Rtype_rd);
            when sp_func_seq | sp_func_sne |
```



```
                do_EX_load_store;
                do_MEM_load;
                exit when reset = '1';
                do_WB(IR_Itype_rd);
            when op_sw =>
                do_EX_load_store;
                do_MEM_store;
                exit when reset = '1';
            when op_lbu =>
        assert false
            report "LBU instruction not implemented" severity warning;
            when op_lhu =>
        assert false
            report "LHU instruction not implemented" severity warning;
            when op_sb =>
        assert false
            report "SB instruction not implemented" severity warning;
            when op_sh =>
        assert false
            report "SH instruction not implemented" severity warning;
            when op_lf =>
        assert false
            report "LF instruction not implemented" severity warning;
            when op_ld =>
        assert false
            report "LD instruction not implemented" severity warning;
            when op_sf =>
        assert false
            report "SF instruction not implemented" severity warning;
            when op_sd =>
        assert false
            report "SD instruction not implemented" severity warning;
            when op_sequi | op_sneui | op_sltui |
                    op_sgtui | op_sleui | op_sgeui =>
                do_EX_set_unsigned(immed => true);
                do_WB(IR_Itype_rd);
            when others =>
        assert false
            report "undefined instruction" severity error;
          end case;
          --
      end loop;
      --
      ------------------------------------------------------- ---------------
      -- loop exited on reset
      ------------------------------------------------------- ---------------
      assert reset = '1'
                    report "Internal error: reset code reached with reset = '0'"
      severity failure;
      --
      -- start again
      --
    end process sequencer;
end behavior;
```

B.2 Pipeline Implementation

```
    wait until reset = '1';
    exit;
      when op_seqi | op_snei | op_slti |
              op_sgti | op_slei | op_sgei =>
          do_EX_set_signed(immed => true);
          do_WB(IR_Itype_rd);
        when op_lb =>
    assert false
      report "LB instruction not implemented" severity warning;
        when op_lh =>
    assert false
      report "LH instruction not implemented" severity warning;
        when op_lw =>
```

```
--
-- Ifetch architecture
--

behavioral architecture of Ifetch is
  signal instruction                : std_logic_vector(0 to DATA_WIDTH -1);
  signal PC                         : std_logic_vector(0 to ADDR_WIDTH - 1);
  signal nextPC signal              : std_logic_vector(0 to ADDR_WIDTH - 1);
  signal PCCurr                     : std_logic_vector(0 to ADDR_WIDTH - 1);
  signal PCtemp                     : std_logic_vector(0 to ADDR_WIDTH - 1);
  signal addtemp                    : std_logic_vector(0 to ADDR_WIDTH - 1);
  signal NextPCadd                  : std_logic_vector(0 to ADDR_WIDTH);
  signal tempflush                  : std_logic;

-- Signals needed for RAM
-- signal nowrite: std_logic;

  begin
  inst_ram: lpm_rom
        GENERIC MAP (lpm_widthad => ADDR_WIDTH,
                     lpm_outdata => "UNREGISTERED",
                     lpm_address_control => "UNREGISTERED",
                     lpm_file => "instruct.mif",
                     lpm_width => DATA_WIDTH)
        PORT MAP (address => PCCurr(0 to ADDR_WIDTH - 1),
                  --inclock => clock,
                        q => Instruction(0 to DATA_WIDTH - 1));

-- Increment PC by 4
  PCout(0 to 7) <= PCCurr(0 to ADDR_WIDTH -1);
  NextPCAdd(0 to 8) <= PCCurr(0 to ADDR_WIDTH - 1) + 1;
  NextPC(0 to 7) <= NextPCAdd(1 to ADDR_WIDTH);
  tempflush <= '1' when (Branch_D(0 to 1) = "01")
                        or (Branch_D(0 to 1) = "10" And Zero_D='1')
                        or (Branch_D(0 to 1) = "11" and Zero_D='0')
                        or jump_D='1'
                   else '0';

  PCCurr(0 to ADDR_WIDTH - 1) <= AddResultBR_D(0 to ADDR_WIDTH - 1)
        when jump_D='1'
        else AddResult_D(0 to ADDR_WIDTH - 1)
             when tempflush='1'
             else PC(0 to          ADDR_WIDTH - 1);

  flush <= tempflush;

-- Load next PC
process
  begin
  wait until Clock'event and Clock='1';
    if reset='1' then
      PC <= (others => '0');
      PCAdd_D(0 to          ADDR_WIDTH - 1) <= (others => '0');
      Instruction_D(0 to DATA_WIDTH - 1) <= (others => '0');
    else
      PC(0 to ADDR_WIDTH - 1) <= NextPC(0 to ADDR_WIDTH - 1);
      PCAdd_D(0 to ADDR_WIDTH - 1) <= NextPC(0 to ADDR_WIDTH - 1);
      Instruction_D(0 to DATA_WIDTH - 1) <= Instruction(0 to DATA_WIDTH - 1);
    end if;
end process;
```

B.2.1 Instruction Search Stage

```
--
-- Ifetch module (provides the PC and instruction memory for the SPIM
-- computer)
--

library alter,IEEE,lpm;
use         alter.maxplus2.all;
use         IEEE.STD_LOGIC_1164.all;
USE         lpm.lpm_components.all;
use         IEEE.STD_LOGIC_ARITH.all;
use         IEEE.STD_LOGIC_SIGNED.all;

-- We can have up to 256 instructions with a PC address of 10 bits
-- however, we chose to only use 128 instructions at this time

entity Ifetch is
    Generic(ADDR_WIDTH : integer := 32; DATA_WIDTH: integer := 32);
    port(
        Instruction_D : out std_logic_vector(0 to DATA_WIDTH -                    1);
        PCadd_D             : out std_logic_vector(0 to ADDR_WIDTH - 1);
        Addresult_D         : in std_logic_vector(0 to ADDR_WIDTH - 1);
        AddResultBR_D : in std_logic_vector(0 to ADDR_WIDTH - 1);
        Branch_D            : in std_logic_vector(0 to 1);
        clock               : in std_logic;
        Reset               : in std_logic;
        Zero_D              : in std_logic;
        jump_D              : in std_logic;
        flush               :out std_logic;
        PCout               : out std_logic_vector(0 to          ADDR_WIDTH - 1)
    );
end entity Ifetch;
```

B.2.2 Decoding Stage

```
--
--      Idecode module (provides the register file for the DLX computer)
--

library alter,IEEE,lpm;
use         alter.maxplus2.all;
use         IEEE.STD_LOGIC_1164.all;
USE         lpm.lpm_components.all;
use         IEEE.STD_LOGIC_ARITH.all;
use         IEEE.STD_LOGIC_SIGNED.all;

entity Idecode is
  Generic(ADDR_WIDTH : integer := 32; DATA_WIDTH: integer := 32);
  port(
    rr1d_bus_D      :out    std_logic_vector(0 to DATA_WIDTH - 1);
    rr2d_bus_D      :out    std_logic_vector(0 to DATA_WIDTH - 1);
    Instruction_D : in      std_logic_vector(0 to DATA_WIDTH - 1);
    wrd_bus         : in    std_logic_vector(0 to DATA_WIDTH - 1);
    RegWrite_D      : in    std_logic;
    RegWrite_DD     : in    std_logic;
    RegWrite_DDD    : in    std_logic;
    RegDst          : in    std_logic;
    Zero_D          :out    std_logic;
    ADDResult_D     :out    std_logic_vector(0 to ADDR_WIDTH - 1);
    ADDResultBR_D : out     std_logic_vector(0 to ADDR_WIDTH - 1);
    PCadd_D         : in    std_logic_vector(0 to ADDR_WIDTH - 1);
    Extend_D        :out    std_logic_vector(0 to ADDR_WIDTH - 1);
    Func_D          :out    std_logic_vector(0 to 5);
```



```
    ALUSelA_D       :out    std_logic_vector(0 to 1);
    ALUSelB_D       :out    std_logic_vector(0 to 1);
    ALUResult_D     : in    std_logic_vector(0 to DATA_WIDTH - 1);
    switch          : in    std_logic_vector(0 to 7);
    regvalue        :out    std_logic_vector(0 to DATA_WIDTH - 1);
    jumpR           : in    std_logic;
    already         : in    std_logic;
    char_mode_D     :out    std_logic_vector(0 to 1);
    --      Accept_Key : in std_logic;
    --      Key_Data        : in std_logic_vector (0 to 5);
    --      Key_Stroke : in std_logic;
    clock           : in    std_logic;
    Reset           : in    std_logic
    );
end entity Idecode;

--
-- Idecode architecture
--

behavioral architecture of Idecode is

signal wraddress,wraddress_D                    : std_logic_vector(0 to 4)
signal wraddress_DD,wraddress_DDD : std_logic_vector(0 to 4);
signal Extend : std_logic_vector(0 to ADDR_WIDTH/2 - 1);

signal reg1,reg2,reg3,reg4,reg5                 : std_logic_vector(0 to DATA_WIDTH - 1);
signal reg6,reg7,reg8,reg9,reg10                : std_logic_vector(0 to DATA_WIDTH - 1);
signal reg11,reg12,reg13,reg14                  : std_logic_vector(0 to DATA_WIDTH - 1);
signal reg15,reg16,reg17,reg18                  : std_logic_vector(0 to DATA_WIDTH - 1);
signal reg19,reg20,reg21,reg22                  : std_logic_vector(0 to DATA_WIDTH - 1);
signal reg23,reg24,reg25,reg26                  : std_logic_vector(0 to DATA_WIDTH - 1);
signal reg27,reg28,reg29,reg30                  : std_logic_vector(0 to DATA_WIDTH - 1);
signal reg31                                    : std_logic_vector(0 to DATA_WIDTH - 1);
signal muxreg1,muxreg2,muxreg3                  : std_logic_vector(0 to DATA_WIDTH - 1);
signal muxreg4,muxreg5,muxreg6                  : std_logic_vector(0 to DATA_WIDTH - 1);
signal muxreg7,muxreg8                          : std_logic_vector(0 to DATA_WIDTH - 1);
signal muxreg9,muxreg10,muxreg11                : std_logic_vector(0 to DATA_WIDTH - 1);

behavioral end architecture;
```

```
    with rr2add_bus(0 to 4) select
    rr2d_bus(0 to DATA_WIDTH - 1) <=
                "00000000000000000000000000000000"      when "0000",
                reg1(0 to DATA_WIDTH - 1)               when "00001",
                reg2(0 to DATA_WIDTH - 1)               when "00010",
                reg3(0 to DATA_WIDTH - 1)               when "00011",
                reg4(0 to DATA_WIDTH - 1)               when "00100",
                reg5(0 to DATA_WIDTH - 1)               when "00101",
                reg6(0 to DATA_WIDTH - 1)               when "00110",
                reg7(0 to DATA_WIDTH - 1)               when "00111",
                reg8(0 to DATA_WIDTH - 1)               when "01000",
                reg9(0 to DATA_WIDTH - 1)               when "01001",
                reg10(0 to DATA_WIDTH - 1) when "01010",
                reg11(0 to DATA_WIDTH - 1) when "01011",
                reg12(0 to DATA_WIDTH - 1) when "01100",
                reg13(0 to DATA_WIDTH - 1) when "01101",
                reg14(0 to DATA_WIDTH - 1) when "01110",
                reg15(0 to DATA_WIDTH - 1) when "01111",
                reg16(0 to DATA_WIDTH - 1) when "10000",
                reg17(0 to DATA_WIDTH - 1) when "10001",
                reg18(0 to DATA_WIDTH - 1) when "10010",
                reg19(0 to DATA_WIDTH - 1) when "10011",
                reg20(0 to DATA_WIDTH - 1) when "10100",
                reg21(0 to DATA_WIDTH - 1) when "10101",
                reg22(0 to DATA_WIDTH - 1) when "10110",
                reg23(0 to DATA_WIDTH - 1) when "10111",
                reg24(0 to DATA_WIDTH - 1) when "11000",
                reg25(0 to DATA_WIDTH - 1) when "11001",
                reg26(0 to DATA_WIDTH - 1) when "11010",
                reg27(0 to DATA_WIDTH - 1) when "11011",
                reg28(0 to DATA_WIDTH - 1) when "11100",
                reg29(0 to DATA_WIDTH - 1) when "11101",
                reg30(0 to DATA_WIDTH - 1) when "11110",
                reg31(0 to DATA_WIDTH - 1) when "11111",
                "11111111111111111111111111111111" when others;
```

```
signal muxreg12,muxreg13,muxreg14 : std_logic_vector(0 to DATA_WIDTH - 1);
signal muxreg15,muxreg16,muxreg17 : std_logic_vector(0 to DATA_WIDTH - 1);
signal muxreg18,muxreg19,muxreg20 : std_logic_vector(0 to DATA_WIDTH - 1);
signal muxreg21,muxreg22,muxreg23 : std_logic_vector(0 to DATA_WIDTH - 1);
signal muxreg24,muxreg25,muxreg26 : std_logic_vector(0 to DATA_WIDTH - 1);
signal muxreg27,muxreg28,muxreg29 : std_logic_vector(0 to DATA_WIDTH - 1);
signal muxreg30,muxreg31                : std_logic_vector(0 to DATA_WIDTH - 1);

signal reg1wr,reg2wr,reg3wr,reg4wr                    : std_logic;
signal reg5wr,reg6wr,reg7wr,reg8wr,reg9wr : std_logic;
signal reg10wr,reg11wr,reg12wr,reg13wr                : std_logic;
signal reg14wr,reg15wr,reg16wr                        : std_logic;
signal reg17wr,reg18wr,reg19wr,reg20wr                : std_logic;
signal reg21wr,reg22wr,reg23wr,reg24wr                : std_logic;
signal reg25wr,reg26wr,reg27wr,reg28wr,reg29wr,reg30wr : std_logic;

signal rr1add_bus,rr2add_bus,mem_addr : std_logic_vector(0 to 4);

signal Func: std_logic_vector(0 to 5);

signal rr1d_bus,rr2d_bus          : std_logic_vector(0 to DATA_WIDTH - 1);
signal rr1d_bus2,rr2d_bus2 : std_logic_vector(0 to DATA_WIDTH - 1);
signal rr2d_bus3                  : std_logic_vector(0 to DATA_WIDTH - 1);

signal zero                        : std_logic;
signal AddResult, ResMux           : std_logic_vector(0 to ADDR_WIDTH - 1);
signal ALUSelA,ALUSelB, char_mode : std_logic_vector(0 to 1);
signal ADDResultBR                 : std_logic_vector(0 to ADDR_WIDTH - 1);
signal RegWrite_DDDout             : std_logic_vector(0 to DATA_WIDTH - 1);

signal Key_State : std_logic;
signal Wait_Key          : std_logic;

begin
    rr1add_bus(0 to 4) <= Instruction_D(11 to 15);
    rr2add_bus(0 to 4) <= Instruction_D(16 to 20)
                        when RegDst='1' else Instruction_D(6 to 10);
    wraddress(0 to 4)         <= Instruction_D(6 to 10);
    Function(0 to 5)          <= Instruction_D(26 to 31);
    Extend(0 to ADDR_WIDTH/2 - 1) <=
                        Instruction_D(ADDR_WIDTH/2 to ADDR_WIDTH - 1);
    char_mode(0 to 1)         <= Instruction_D(9 to 10);


-- Need to add additional register support later to support 32 registers
-- Read Register Operations
    with rr1add_bus(0 to 4) select
       rr1d_bus(0 to DATA_WIDTH - 1) <=
            "00000000000000000000000000000000"       when "0000",
                  reg1(0 to DATA_WIDTH - 1)          when "00001",
                  reg2(0 to DATA_WIDTH - 1)          when "00010",
                  reg3(0 to DATA_WIDTH - 1)          when "00011",
                  reg4(0 to DATA_WIDTH - 1)          when "00100",
                  reg5(0 to DATA_WIDTH - 1)          when "00101",
                  reg6(0 to DATA_WIDTH - 1)          when "00110",
                  reg7(0 to DATA_WIDTH - 1)          when "00111",
                  reg8(0 to DATA_WIDTH - 1)          when "01000",
                  reg10(0 to DATA_WIDTH - 1) when "01010",
                  reg11(0 to DATA_WIDTH - 1) when "01011",
                  reg12(0 to DATA_WIDTH - 1) when "01100",
                  reg13(0 to DATA_WIDTH - 1) when "01101",
                  reg14(0 to DATA_WIDTH - 1) when "01110",
                  reg15(0 to DATA_WIDTH - 1) when "01111",
                  reg16(0 to DATA_WIDTH - 1) when "10000",
                  reg17(0 to DATA_WIDTH - 1) when "10001",
                  reg18(0 to DATA_WIDTH - 1) when "10010",
                  reg19(0 to DATA_WIDTH - 1) when "10011",
                  reg20(0 to DATA_WIDTH - 1) when "10100",
                  reg21(0 to DATA_WIDTH - 1) when "10101",
                  reg22(0 to DATA_WIDTH - 1) when "10110",
                  reg23(0 to DATA_WIDTH - 1) when "10111",
                  reg24(0 to DATA_WIDTH - 1) when "11000",
                  reg25(0 to DATA_WIDTH - 1) when "11001",
                  reg26(0 to DATA_WIDTH - 1) when "11010",
                  reg27(0 to DATA_WIDTH - 1) when "11011",
                  reg28(0 to DATA_WIDTH - 1) when "11100",
                  reg29(0 to DATA_WIDTH - 1) when "11101",
                  reg30(0 to DATA_WIDTH - 1) when "11110",
                  reg31(0 to DATA_WIDTH - 1) when "11111",
            "11111111111111111111111111111111" when others;


muxreg1(0 to DATA_WIDTH - 1) <=r eg1(0 to DATA_WIDTH - 1) when reg1wr='0'
                                                                ELSE wrd_bus;
reg1wr <= '1' when ((wraddress_DDD(0 to 4)="00001") and (RegWrite_DDD='1'))
                  ELSE '0';

muxreg2(0 to DATA_WIDTH - 1) <= reg2(0 to DATA_WIDTH - 1) when reg2wr='0'
                                                                ELSE wrd_bus;
reg2wr <= '1' when ((wraddress_DDD(0 to 4)="00010") and (RegWrite_DDD='1'))
                  ELSE '0';

muxreg3(0 to DATA_WIDTH - 1) <= reg3(0 to DATA_WIDTH - 1) when reg3wr='0'
                                                               ELSE wrd_bus;
reg3wr <= '1' when ((wraddress_DDD(0 to 4)="00011") and (RegWrite_DDD='1'))
                  ELSE '0';

muxreg4(0 to DATA_WIDTH - 1) <= reg4(0 to DATA_WIDTH - 1) when reg4wr='0'
                                                                ELSE wrd_bus;
reg4wr <= '1' when ((wraddress_DDD(0 to 4)="00100") and (RegWrite_DDD='1'))
                  ELSE '0';

muxreg5(0 to DATA_WIDTH - 1) <= reg5(0 to DATA_WIDTH - 1) when reg5wr='0'
                                                                ELSE wrd_bus;
reg5wr <= '1' when ((wraddress_DDD(0 to 4)="00101") and (RegWrite_DDD='1'))
                  ELSE '0';

muxreg6(0 to DATA_WIDTH - 1) <= reg6(0 to DATA_WIDTH - 1) when reg6wr='0'
                                                                ELSE wrd_bus;
reg6wr <= '1' when ((wraddress_DDD(0 to 4)="00110") and (RegWrite_DDD='1'))
                  ELSE '0';

muxreg7(0 to DATA_WIDTH - 1) <= reg7(0 to DATA_WIDTH - 1) when reg7wr='0'
                                                                ELSE wrd_bus;
reg7wr <= '1' when ((wraddress_DDD(0 to 4)="00111") and (RegWrite_DDD='1'))
```



```
                  ELSE '0';

muxreg8(0 to DATA_WIDTH - 1) <= reg8(0 to DATA_WIDTH - 1) when reg8wr='0'
                                                                ELSE wrd_bus;
reg8wr <= '1' when ((wraddress_DDD(0 to 4)="01000") and (RegWrite_DDD='1'))
                  ELSE '0';

muxreg9(0 to DATA_WIDTH - 1) <= reg9(0 to DATA_WIDTH - 1) when reg9wr='0'
                                                                ELSE wrd_bus;
reg9wr <= '1' when ((wraddress_DDD(0 to 4)="01001") and (RegWrite_DDD='1'))
                  ELSE '0';

muxreg10(0 to DATA_WIDTH - 1) <= reg10(0 to DATA_WIDTH - 1)
                                        when reg10wr='0' ELSE wrd_bus;
reg10wr <= '1' when ((wraddress_DDD(0 to 4)="01010") and
                                    (RegWrite_DDD='1')) ELSE '0';

muxreg27(0 to DATA_WIDTH - 1) <= reg27(0 to DATA_WIDTH - 1)
                                        when reg27wr='0' ELSE wrd_bus;

reg27wr <= '1' when ((wraddress_DDD(0 to 4)="11011") and
                                    (RegWrite_DDD='1')) ELSE '0';
muxreg28(0 to DATA_WIDTH - 1) <= reg28(0 to DATA_WIDTH - 1)
                                        when reg28wr='0' ELSE wrd_bus;

reg28wr <= '1' when ((wraddress_DDD(0 to 4)="11100") and
                                    (RegWrite_DDD='1')) ELSE '0';
muxreg29(0 to DATA_WIDTH - 1) <= reg29(0 to DATA_WIDTH - 1)
                                        when reg29wr='0' ELSE wrd_bus;

reg29wr <= '1' when ((wraddress_DDD(0 to 4)="11101") and
                                    (RegWrite_DDD='1')) ELSE '0';
muxreg30(0 to DATA_WIDTH - 1) <= reg30(0 to DATA_WIDTH - 1)
```

```
muxreg11(0 to DATA_WIDTH - 1) <= reg11(0 to DATA_WIDTH - 1)
                                                    when reg11wr='0' ELSE wrd_bus;
reg11wr <= '1' when ((wraddress_DDD(0 to 4)="01011") and
                                                (RegWrite_DDD='1')) ELSE '0';
muxreg12(0 to DATA_WIDTH - 1) <= reg12(0 to DATA_WIDTH - 1)
                                                    when reg12wr='0' ELSE wrd_bus;

reg12wr <= '1' when ((wraddress_DDD(0 to 4)="01100") and
                                                (RegWrite_DDD='1')) ELSE '0';
muxreg13(0 to DATA_WIDTH - 1) <= reg13(0 to DATA_WIDTH - 1)
                                                    when reg13wr='0' ELSE wrd_bus;

reg13wr <= '1' when ((wraddress_DDD(0 to 4)="01101") and
                                                (RegWrite_DDD='1')) ELSE '0';
muxreg14(0 to DATA_WIDTH - 1) <= reg14(0 to DATA_WIDTH - 1)
                                                    when reg14wr='0' ELSE wrd_bus;

reg14wr <= '1' when ((wraddress_DDD(0 to 4)="01110") and
                                                (RegWrite_DDD='1')) ELSE '0';
muxreg15(0 to DATA_WIDTH - 1) <= reg15(0 to DATA_WIDTH - 1)
                                                    when reg15wr='0' ELSE wrd_bus;

reg15wr <= '1' when ((wraddress_DDD(0 to 4)="01111") and
                                                (RegWrite_DDD='1')) ELSE '0';
muxreg16(0 to DATA_WIDTH - 1) <= reg16(0 to DATA_WIDTH - 1)
                                                    when reg16wr='0' ELSE wrd_bus;

reg16wr <= '1' when ((wraddress_DDD(0 to 4)="10000") and
                                                (RegWrite_DDD='1')) ELSE '0';
muxreg17(0 to DATA_WIDTH - 1) <= reg17(0 to DATA_WIDTH - 1)
                                                    when reg17wr='0' ELSE wrd_bus;

reg17wr <= '1' when ((wraddress_DDD(0 to 4)="10001") and
                                                (RegWrite_DDD='1')) ELSE '0';
muxreg18(0 to DATA_WIDTH - 1) <= reg18(0 to DATA_WIDTH - 1)
                                                    when reg18wr='0' ELSE wrd_bus;

reg18wr <= '1' when ((wraddress_DDD(0 to 4)="10010") and
                                                (RegWrite_DDD='1')) ELSE '0';
muxreg19(0 to DATA_WIDTH - 1) <= reg19(0 to DATA_WIDTH - 1)
                                                    when reg19wr='0' ELSE wrd_bus;

reg19wr <= '1' when ((wraddress_DDD(0 to 4)="10011") and
                                                (RegWrite_DDD='1')) ELSE '0';
muxreg20(0 to DATA_WIDTH - 1) <= reg20(0 to DATA_WIDTH - 1)
                                                    when reg20wr='0' ELSE wrd_bus;

reg20wr <= '1' when ((wraddress_DDD(0 to 4)="10100")
                                        and (RegWrite_DDD='1')) ELSE '0';
muxreg21(0 to DATA_WIDTH - 1) <= reg21(0 to DATA_WIDTH - 1)
                                                    when reg21wr='0' ELSE wrd_bus;

reg21wr <= '1' when ((wraddress_DDD(0 to 4)="10101") and
                                                (RegWrite_DDD='1')) ELSE '0';
muxreg22(0 to DATA_WIDTH - 1) <= reg22(0 to DATA_WIDTH - 1)
                                                    when reg22wr='0' ELSE wrd_bus;

reg22wr <= '1' when ((wraddress_DDD(0 to 4)="10110") and
                                                (RegWrite_DDD='1')) ELSE '0';
muxreg23(0 to DATA_WIDTH - 1) <= reg23(0 to DATA_WIDTH - 1)
                                                    when reg23wr='0' ELSE wrd_bus;

reg23wr <= '1' when ((wraddress_DDD(0 to 4)="10111") and
                                                (RegWrite_DDD='1')) ELSE '0';
muxreg24(0 to DATA_WIDTH - 1) <= reg24(0 to DATA_WIDTH - 1)
                                                    when reg24wr='0' ELSE wrd_bus;

reg24wr <= '1' when ((wraddress_DDD(0 to 4)="11000") and
                                                (RegWrite_DDD='1')) ELSE '0';
muxreg25(0 to DATA_WIDTH - 1) <= reg25(0 to DATA_WIDTH - 1)
                                                    when reg25wr='0' ELSE wrd_bus;

reg25wr <= '1' when ((wraddress_DDD(0 to 4)="11001") and
                                                (RegWrite_DDD='1')) ELSE '0';
muxreg26(0 to DATA_WIDTH - 1) <= reg26(0 to DATA_WIDTH - 1)
                                                    when reg26wr='0' ELSE wrd_bus;

reg26wr <= '1' when ((wraddress_DDD(0 to 4)="11010") and
                                                (RegWrite_DDD='1')) ELSE '0';
                                                    when reg30wr='0' ELSE wrd_bus;
reg30wr <= '1' when ((wraddress_DDD(0 to 4)="11110") and
                                                (RegWrite_DDD='1')) ELSE '0';

--muxreg31(0 to DATA_WIDTH - 1) <= reg31(0 to DATA_WIDTH - 1)
--                                                  when reg31wr='0' ELSE wrd_bus;
--reg31wr <= '1' when ((wraddress_DDD(0 to 4)="11111") and
--                                                (RegWrite_DDD='1')) ELSE '0';

--          (For 8-bit data path model)

rr1d_bus2(0 to DATA_WIDTH - 1) <= wrd_bus(0 to DATA_WIDTH - 1) when
            (wraddress_DDD(0 to 4) = rr1add_bus(0 to 4) and RegWrite_DDD = '1')
        else rr1d_bus(0 to DATA_WIDTH - 1);

--SW forwarding still applies since this is the Register to store

rr2d_bus2(0 to DATA_WIDTH - 1) <= wrd_bus(0 to DATA_WIDTH - 1) when
            (wraddress_DDD(0 to 4) = rr2add_bus(0 to 4) and RegWrite_DDD = '1')
    else rr2d_bus(0 to DATA_WIDTH - 1);

rr2d_bus3(0 to DATA_WIDTH - 1) <= ALUResult_D(0 to DATA_WIDTH - 1) when
            (wraddress_DD(0 to 4)                   = rr2add_bus(0 to 4) and RegWrite_D = '1')
    else rr2d_bus2(0 to DATA_WIDTH - 1);

RegWrite_DDDout(0 to DATA_WIDTH - 1) <= "00000" & RegWrite_D &
                                                    RegWrite_DD & RegWrite_DDD;

  with switch(2 to 7) select
    regvalue(0 to DATA_WIDTH - 1) <= reg1(0 to DATA_WIDTH - 1) when "000001",
                                                    reg2(0 to DATA_WIDTH - 1) when "000010",
                                                    reg3(0 to DATA_WIDTH - 1) when "000011",
                                                    reg4(0 to DATA_WIDTH - 1) when "000100",
                                                    reg5(0 to DATA_WIDTH - 1) when "000101",
                                                    reg6(0 to DATA_WIDTH - 1) when "000110",
                                                    reg7(0 to DATA_WIDTH - 1) when "000111",
                                                    reg8(0 to DATA_WIDTH - 1) when "001000",
                                                    reg9(0 to DATA_WIDTH - 1) when "001001",
                                                  reg10(0 to DATA_WIDTH - 1) when "001010",
                                                  reg11(0 to DATA_WIDTH - 1) when "001011",
                                                  reg12(0 to DATA_WIDTH - 1) when "001100",
                                                  reg13(0 to DATA_WIDTH - 1) when "001101",
                                                  reg14(0 to DATA_WIDTH - 1) when "001110",
                                                  reg15(0 to DATA_WIDTH - 1) when "001111",
                                                  reg16(0 to DATA_WIDTH - 1) when "010000",
                                                  reg17(0 to DATA_WIDTH - 1) when "010001",
                                                  reg18(0 to DATA_WIDTH - 1) when "010010",
                                                  reg19(0 to DATA_WIDTH - 1) when "010011",
                                                  reg20(0 to DATA_WIDTH - 1) when "010100",
                                                  reg21(0 to DATA_WIDTH - 1) when "010101",
                                                  reg22(0 to DATA_WIDTH - 1) when "010110",
                                                  reg23(0 to DATA_WIDTH - 1) when "010111",
                                                  reg24(0 to DATA_WIDTH - 1) when "011000",
                                                  reg25(0 to DATA_WIDTH - 1) when "011001",
                                                  reg26(0 to DATA_WIDTH - 1) when "011010",
                                                  reg27(0 to DATA_WIDTH - 1) when "011011",
                                                  reg28(0 to DATA_WIDTH - 1) when "011100",
                                                  reg29(0 to DATA_WIDTH - 1) when "011101",
                                                  reg30(0 to DATA_WIDTH - 1) when "011110",
                                                  reg31(0 to DATA_WIDTH - 1) when "011111",

                                          regWrite_DDDout(0 to DATA_WIDTH - 1) when "100000",
                                                  wraddress_D(0 to 4)          when "100001",
                                                  wraddress_DD(0 to 4)         when "100010",
                                                  wraddress_DDD(0 to 4)        when "100011",
                                          AddResultBR(0 to ADDR_WIDTH - 1)     when "100100",
                                                AddResult(0 to ADDR_WIDTH - 1) when "100101",
                                          "00000000000000000000000000000000" when OTHERS;

-- 11/8 mikes 1:49 PM I think this is a problem but I'm not going to fix it right
-- now.          This following code does not use the forwarding provided by the above muxes.

--ResMux(0 to DATA_WIDTH - 1) <= rr1d_bus2(0 to DATA_WIDTH - 1) -
--                                                  rr2d_bus2(0 to DATA_WIDTH - 1);
--Zero <= '1' when ResMux(0 to DATA_WIDTH - 1) = "00000000" ELSE '0';

Zero <= '1' when rr2d_bus3="00000000" else '0';
muxreg31(0 to DATA_WIDTH - 1) <= PCadd_D(0 to DATA_WIDTH - 1)
                                          when Jal='1' else reg31(0 to DATA_WIDTH - 1);
```

```
AddresultBR(0 to DATA_WIDTH - 1) <= Instruction_D(24 to 31)
                                    when jal = '1'
                                    ELSE Reg31(0 to DATA_WIDTH - 1) when jumpr = '1'
                                    ELSE "00000000000000000000000000000000";
AddResult(0 to DATA_WIDTH - 1) <= Extend(0 to DATA_WIDTH - 1);

--ALUSelA should not use forwarding for LW/SW since the second data line
--is actually the memory address in disguise!
ALUSEL(0 to 1) <= "01" when
      (RegWrite_D = '1' and (wraddress_D(0 to 4) = rr1add_bus(0 to 4)) and
        rr1add_bus(0 to 4) /= "00000")
    else "10" when
      (RegWrite_DD = '1' and (wraddress_DD(0 to 4) = rr1add_bus(0 to 4)) and
      (RegWrite_D = '0' or (RegWrite_D = '1' and
        wraddress_D(0 to 4) /= rr1add_bus(0 to 4)))
        and rr1add_bus(0 to 4) /= "00000")
    else "00";

ALUSelB(0 to 1) <= "01" when
      (RegWrite_D = '1' and (wraddress_D(0 to 4) = rr2add_bus(0 to 4)) and
        rr2add_bus(0 to 4) /= "00000")
    else "10" when
      (RegWrite_DD = '1' and (wraddress_DD(0 to 4) = rr2add_bus(0 to 4)) and
      (RegWrite_D = '0' or (RegWrite_D = '1' and
        wraddress_D(0 to 4) /= rr2add_bus(0 to 4)))
        and rr2add_bus(0 to 4) /= "00000")
    else "00";

Process
    variable i : std_logic;
begin
    wait until clock'event and clock='1';
    if reset='1' then
        Addresult_D(0 to DATA_WIDTH - 1) <= "00000000";
        AddResultBR_D(0 to DATA_WIDTH - 1) <= "00000000";
        ALUSelA_D <= "00";
        ALUSelB_D <= "00";
    else
      --Write address is written back here and not needed in other modules
    wraddress_DDD(0 to 4) <= wraddress_DD(0 to 4);
        -- to get instruction back to idecode for write back
    wraddress_DD(0 to 4) <= wraddress_D(0 to 4);
        -- for when instruction is in memory
    wraddress_D(0 to 4) <= wraddress(0 to 4);
        -- for when instruction is in execute
    rr1d_bus_D(0 to DATA_WIDTH - 1) <= rr1d_bus2(0 to DATA_WIDTH - 1);
    rr2d_bus_D(0 to DATA_WIDTH - 1) <= rr2d_bus2(0 to DATA_WIDTH - 1);
    for i in 0 to (ADDR_WIDTH/2 - 1) loop
      Extend_D(i) <= Extend(0);
    end loop;
    Extend_D(ADDR_WIDTH to ADDR_WIDTH -1) <= Extend(0 to ADDR_WIDTH/2 - 1);
    Func_D(0 to 5) <= Func(0 to 5);
    AddResult_D(0 to DATA_WIDTH - 1) <= AddResult(0 to DATA_WIDTH - 1);
    AddResultBR_D(0 to DATA_WIDTH - 1) <= AddResultBR(0 to DATA_WIDTH - 1);

    ALUSelA_D(0 to 1) <= ALUSelA(0 to 1);
    ALUSelB_D(0 to 1) <= ALUSelB(0 to 1);
    end if;
end process;

Process
begin
  wait until Clock'event and Clock='1';
  if reset='1' then
    Zero_D <= '0';
    --These initial values were added for testing of the
    --        Change Board and chip only
    --They should be removed later
    reg1(0 to DATA_WIDTH - 1)                   <= (others => '0');
    reg2(0 to DATA_WIDTH - 1)                   <= (others => '0');
    reg3(0 to DATA_WIDTH - 1)                   <= (others => '0');
    reg4(0 to DATA_WIDTH - 1)                   <= (others => '0');
    reg5(0 to DATA_WIDTH - 1)                   <= (others => '0');
    reg6(0 to DATA_WIDTH - 1)                   <= (others => '0');
    reg7(0 to DATA_WIDTH - 1)                   <= (others => '0');
    reg8(0 to DATA_WIDTH - 1)                   <= (others => '0');
    reg9(0 to DATA_WIDTH - 1)                   <= (others => '0');
    reg10(0 to DATA_WIDTH - 1)                  <= (others => '0');
    reg11(0 to DATA_WIDTH - 1)                  <= (others => '0');
    reg12(0 to DATA_WIDTH - 1)                  <= (others => '0');
    reg13(0 to DATA_WIDTH - 1)                  <= (others => '0');
    reg14(0 to DATA_WIDTH - 1)                  <= (others => '0');
    reg15(0 to DATA_WIDTH - 1)                  <= (others => '0');
    reg16(0 to DATA_WIDTH - 1)                  <= (others => '0');
    reg17(0 to DATA_WIDTH - 1)                  <= (others => '0');
    reg18(0 to DATA_WIDTH - 1)                  <= (others => '0');
    reg19(0 to DATA_WIDTH - 1)                  <= (others => '0');
    reg20(0 to DATA_WIDTH - 1)                  <= (others => '0');
    reg21(0 to DATA_WIDTH - 1)                  <= (others => '0');
    reg22(0 to DATA_WIDTH - 1)                  <= (others => '0');
    reg23(0 to DATA_WIDTH - 1)                  <= (others => '0');
    reg24(0 to DATA_WIDTH - 1)                  <= (others => '0');
    reg25(0 to DATA_WIDTH - 1)                  <= (others => '0');
    reg26(0 to DATA_WIDTH - 1)                  <= (others => '0');
    reg27(0 to DATA_WIDTH - 1)                  <= (others => '0');
    reg28(0 to DATA_WIDTH - 1)                  <= (others => '0');
    reg29(0 to DATA_WIDTH - 1)                  <= (others => '0');
    reg30(0 to DATA_WIDTH - 1)                  <= (others => '0');
    reg31(0 to DATA_WIDTH - 1)                  <= (others => '0');
  else
    char_mode_D(0 to 1) <= char_mode(0 to 1);
    Zero_D <= Zero;
    reg1(0 to DATA_WIDTH - 1)                   <= muxreg1(0 to DATA_WIDTH - 1);
    reg2(0 to DATA_WIDTH - 1)                   <= muxreg2(0 to DATA_WIDTH - 1);
    reg3(0 to DATA_WIDTH - 1)                   <= muxreg3(0 to DATA_WIDTH - 1);
    reg4(0 to DATA_WIDTH - 1)                   <= muxreg4(0 to DATA_WIDTH - 1);
    reg5(0 to DATA_WIDTH - 1)                   <= muxreg5(0 to DATA_WIDTH - 1);
    reg6(0 to DATA_WIDTH - 1)                   <= muxreg6(0 to DATA_WIDTH - 1);
    reg7(0 to DATA_WIDTH - 1)                   <= muxreg7(0 to DATA_WIDTH - 1);
    reg8(0 to DATA_WIDTH - 1)                   <= muxreg8(0 to DATA_WIDTH - 1);
    reg9(0 to DATA_WIDTH - 1)                   <= muxreg9(0 to DATA_WIDTH - 1);
    reg10(0 to DATA_WIDTH - 1)                  <= muxreg10(0 to DATA_WIDTH - 1);
    reg11(0 to DATA_WIDTH - 1)                  <= muxreg11(0 to DATA_WIDTH - 1);
    reg12(0 to DATA_WIDTH - 1)                  <= muxreg12(0 to DATA_WIDTH - 1);
    reg13(0 to DATA_WIDTH - 1)                  <= muxreg13(0 to DATA_WIDTH - 1);
    reg14(0 to DATA_WIDTH - 1)                  <= muxreg14(0 to DATA_WIDTH - 1);
    reg15(0 to DATA_WIDTH - 1)                  <= muxreg15(0 to DATA_WIDTH - 1);
    reg16(0 to DATA_WIDTH - 1)                  <= muxreg16(0 to DATA_WIDTH - 1);
    reg17(0 to DATA_WIDTH - 1)                  <= muxreg17(0 to DATA_WIDTH - 1);
    reg18(0 to DATA_WIDTH - 1)                  <= muxreg18(0 to DATA_WIDTH - 1);
    reg19(0 to DATA_WIDTH - 1)                  <= muxreg19(0 to DATA_WIDTH - 1);
    reg20(0 to DATA_WIDTH - 1)                  <= muxreg20(0 to DATA_WIDTH - 1);
    reg21(0 to DATA_WIDTH - 1)                  <= muxreg21(0 to DATA_WIDTH - 1);
    reg22(0 to DATA_WIDTH - 1)                  <= muxreg22(0 to DATA_WIDTH - 1);
    reg23(0 to DATA_WIDTH - 1)                  <= muxreg23(0 to DATA_WIDTH - 1);
    reg24(0 to DATA_WIDTH - 1)                  <= muxreg24(0 to DATA_WIDTH - 1);
    reg25(0 to DATA_WIDTH - 1)                  <= muxreg25(0 to DATA_WIDTH - 1);
    reg26(0 to DATA_WIDTH - 1)                  <= muxreg26(0 to DATA_WIDTH - 1);
    reg27(0 to DATA_WIDTH - 1)                  <= muxreg27(0 to DATA_WIDTH - 1);
    reg28(0 to DATA_WIDTH - 1)                  <= muxreg28(0 to DATA_WIDTH - 1);
    reg29(0 to DATA_WIDTH - 1)                  <= muxreg29(0 to DATA_WIDTH - 1);
    reg30(0 to DATA_WIDTH - 1)                  <= muxreg30(0 to DATA_WIDTH - 1);
    reg31(0 to DATA_WIDTH - 1)                  <= muxreg31(0 to DATA_WIDTH - 1);
  end if;
end process;

-- Little process that will initialize the different flags once
-- a GetChar operation has been called.
-- Key_State will be the position of the character that has been
-- entered

--process(Accept_Key)
--        begin
-- Key_State <= '0';
-- Wait_Key <= '1';
--end process;

-- Supposes that two numbers are entered every time.
-- Thus the Key_State flag keeps count of position
-- of where the character is being entered ( set to zero means
-- most 'significant bit', set to 1 means least significant)

-- Problem ... Reg30 is 0 to 7, Key_Data is 5 downto 0, need two
-- sets of key therefore 10 bits...
-- So what value is going to be passed to Key_Data?
-- Should be the Hex value of the numbers...

--process(Key_Stroke)
--        begin
--            if Wait_Key ='1' then
--        if Key_State ='0' then
--          Reg30(2 to 3) <= Key_data(1 downto 0);
--          Key_State <='1';
--        else
--          Reg30(0 to 1) <= Key_Data(1 downto 0);
--
--        end if;
--end if;
--end process;

behavioral end architecture;
```

B.2.3 Execution Stage

```
--
-- Execute module (simulates SPIM (ALU) Execute module)
--

library alters,IEEE;
use             alter.maxplus2.all;
use             IEEE.STD_LOGIC_1164.all;
use             IEEE.STD_LOGIC_ARITH.all;
use             IEEE.STD_LOGIC_SIGNED.all;

entity Execute is
  Generic(ADDR_WIDTH : integer := 32; DATA_WIDTH: integer := 32);
  port(
```

```
        reg26(0 to DATA_WIDTH - 1)          <= (others => '0');
        reg27(0 to DATA_WIDTH - 1)          <= (others => '0');
        reg28(0 to DATA_WIDTH - 1)          <= (others => '0');
```

```
        Readdata1           : in    std_logic_vector(0 to DATA_WIDTH - 1);
        Readdata2           : in    std_logic_vector(0 to DATA_WIDTH - 1);
        rr2d_bus_DD         :out    std_logic_vector(0 to DATA_WIDTH - 1);
```



```
    Extend_D            : in    std_logic_vector(0 to ADDR_WIDTH - 1);
    Func_D              : in    std_logic_vector(0 to 5);
    ALUOp0_D            : in    std_logic;
    ALUOp1_D            : in    std_logic;
    ALUSrc_D            : in    std_logic;
    ALUResult_D         :out    std_logic_vector(0 to DATA_WIDTH - 1);
    video_date_D : out          std_logic_vector(0 to DATA_WIDTH - 1);
    wrd_bus             : in    std_logic_vector(0 to DATA_WIDTH - 1);
    ALUSelA_D           : in    std_logic_vector(0 to 1);
    ALUSelB_D           : in    std_logic_vector(0 to 1);
    char_mode           : in    std_logic_vector(0 to 1);
    clock               : in    std_logic;
    Reset               : in    std_logic
    );
end entity Execute;

architecture behavioral of Execute is

  signal input2                     :   std_logic_vector(0 to DATA_WIDTH - 1);
  signal Binput2                    :   std_logic_vector(0 to DATA_WIDTH - 1);
  signal input                      :   std_logic_vector(0 to DATA_WIDTH - 1);
  signal Binput                     :   std_logic_vector(0 to DATA_WIDTH - 1);
  signal ALUresult_Dout :           std_logic_vector(0 to DATA_WIDTH - 1);
  signal ResMux                     :   std_logic_vector(0 to DATA_WIDTH - 1);
  signal ALUctl                     :   std_logic_vector(0 to 2);
  signal ALUResult                  :   std_logic_vector(0 to DATA_WIDTH - 1);
  ALTB signal                       :   std_logic;
  signal Ainput6                    :   std_logic_vector(0 to DATA_WIDTH - 2);
  signal Binput6                    :   std_logic_vector(0 to DATA_WIDTH - 2);
  signal video_reg                  :   std_logic_vector(0 to DATA_WIDTH - 1);


begin
    Ainput(0 to DATA_WIDTH - 1) <= Ainput2(0 to DATA_WIDTH - 1);
    Binput(0 to DATA_WIDTH - 1) <= Binput2(0 to DATA_WIDTH - 1)
            when (ALUSrc_D='0') else Extend_D(0 to DATA_WIDTH - 1);

    Ainput2(0 to DATA_WIDTH - 1) <= ALUResult_Dout(0 to DATA_WIDTH - 1)
            when (ALUSelA_D(0 to 1) = "01")
            else wrd_bus(0 to DATA_WIDTH - 1) when (ALUSelA_D(0 to 1) = "10")
            else readdata1(0 to DATA_WIDTH - 1);

    Binput2(0 to DATA_WIDTH - 1) <= ALUResult_Dout(0 to DATA_WIDTH - 1)
            when (ALUSelB_D(0 to 1) = "01")
            else wrd_bus(0 to DATA_WIDTH - 1) when (ALUSelB_D(0 to 1) = "10")
            else readdata2(0 to DATA_WIDTH - 1);

    video_reg(0 to DATA_WIDTH - 1) <= "0011" & Ainput2(0 to 3)
            when (char_mode(0 to 1) = "00")
            else "0011" & Ainput2(4 to 7) when (char_mode(0 to 1) = "01")
            else Extend_D(0 to DATA_WIDTH - 1) when (char_mode(0 to 1) = "10")
            else "00000000000000000000000000000000";

-- This is where the ALU control bits are set

  ALUctl(0) <= (ALUOp0_D and (NOT ALUOp1_D) and (NOT Func_D(0))
          and (NOT Func_D(1))
          and (NOT Func_D(2)) and Func_D(3) and (NOT FUNC_D(4))
          and ( NOT Func_D(5))) or (ALUOp0_D and (NOT ALUOp1_D)
          and Func_D(0) and (NOT Func_D(1)) and (NOT Func_D(3))
          and Func_D(4) and (NOT Func_D(5))) or ((NOT ALUOp0_D) and ALUOp1_D);
  ALUctl(1) <= (Func_D(0) and (NOT Func_D(1)) and (NOT Func_D(2))
          and (NOT Func_D(3))
          and (NOT Func_D(5))) or (Func_D(0) and (NOT Func_D(1))
          and (NOT Func_D(3)) and Func_D(4) and (NOT Func_D(5)))
          or ((NOT ALUOp0_D) or ALUOp1_D);
  ALUctl(2) <= (ALUOp0_D and (NOT ALUOp1_D) and Func_D(0)
          and (NOT Func_D(1))
          and Func_D(2) and (NOT Func_D(3)) and Func_D(4) and (NOT Func_D(5)))
          or (ALUOp0_D and (NOT ALUOp1_D) and Func_D(0) and (NOT Func_D(1))
--          when "010",
--      Ainput(0 to DATA_WIDTH - 1) - Binput(0 to DATA_WIDTH - 1)
--          when "110",
--      To_stdlogicvector(X"00") when others;

tempinput(DATA_WIDTH - 2 to DATA_WIDTH - 1) := "00";
tempinput(0 to DATA_WIDTH - 3) := Ainput(2 to DATA_WIDTH - 1);

case ALUctl(0 to 2) is
  -- Select ALU operation
  -- ALU performs ALUresult = bus_A and bus_B
when "000" => ResMux(0 to DATA_WIDTH - 1) <= Ainput(0 to DATA_WIDTH - 1)
                                                    and Binput(0 to DATA_WIDTH - 1);
  -- ALU performs ALUresult = bus_A or bus_B
when "001" => ResMux(0 to DATA_WIDTH - 1) <= Ainput(0 to DATA_WIDTH - 1)
                                                    or Binput(0 to DATA_WIDTH - 1);
  -- ALU performs ALUresult = bus_A + bus_B and ADDI and LW and SW
when "010" => ResMux(0 to DATA_WIDTH - 1) <= Ainput(0 to DATA_WIDTH - 1)
                                                    + Binput(0 to DATA_WIDTH - 1);
  -- ALU performs SLL
when "100" => ResMux(0 to DATA_WIDTH - 1) <= tempinput(0 to DATA_WIDTH
                                                    - 1);
--Ainput SLL Binput;
  -- ALU performs SLT and SLTI
--when "111" => if (Ainput(0 to DATA_WIDTH - 1) < Binput(0 to DATA_WIDTH
                                - 1)) THEN
--              ResMux(0 to DATA_WIDTH - 1) <= To_stdlogicvector(X"01");
--          else
--          Resmux(0 to DATA_WIDTH - 1) <= To_stdlogicvector(X"00");
--          end if;
when "111" =>
    if (Ainput(0) ='1' and Binput(0)='0') or
          (Ainput(0) = Binput(0) and ALTB='1')
    then ResMux(0 to DATA_WIDTH - 1) <= "00000000000000000000000000000001";
    else ResMux(0 to DATA_WIDTH - 1) <= (others => '0');
    end if;

when Others => ResMux(0 to DATA_WIDTH - 1) <= (others => '0');
end case;
end process;

ALUResult(0 to DATA_WIDTH - 1) <= ResMux(0 to DATA_WIDTH - 1);
--dff(Zero,reset,clock,Zero_D);
--dff(ALUResult,reset,clock,ALUResult_D);
--dff(AddResult,reset,clock,AddResult_D);

process
begin
    wait until clock'event and clock='1';
    if reset = '1' then
        ALUResult_D(0 to DATA_WIDTH - 1)            <= (others => '0');
        ALUResult_Dout(0 to DATA_WIDTH - 1) <= (others => '0');
        rr2d_bus_DD(0 to DATA_WIDTH - 1)            <= (others => '0');
        video_data_D(0 to DATA_WIDTH - 1)           <= (others => '0');
    else
        rr2d_bus_DD(0 to DATA_WIDTH - 1)            <= Binput2(0 to DATA_WIDTH - 1);
        ALUResult_D(0 to DATA_WIDTH - 1)            <= ALUResult(0 to DATA_WIDTH - 1);
        ALUResult_Dout(0 to DATA_WIDTH - 1) <= ALUResult(0 to DATA_WIDTH - 1);
        video_data_D(0 to DATA_WIDTH - 1)           <= video_reg(0 to DATA_WIDTH - 1);
    end if;
end process;

behavioral end architecture;
```

B.2.4 Memory Access Stage

```
--
-- DMEMORY module (provides the data memory for the DLX computer)
--library Altera, IEEE;
```

```
            and (NOT Func_D(2)) and Func_D(3) and (NOT Func_D(4)) and Func_D(5))
            or ((NOT ALUOp0_D) and ALUOp1_D);

-- Select ALU output

--          ALUresult(0 to DATA_WIDTH - 1) <= To_stdlogicvector(B"0000000")
--                                              & Resmux(7) when ALUctl(0 to 2)="111"
--          else ResMux(0 to DATA_WIDTH - 1);

Ainput6(0 to DATA_WIDTH - 2) <= Ainput(1 to DATA_WIDTH - 1);
Binput6(0 to DATA_WIDTH - 2) <= Binput(1 to DATA_WIDTH - 1);
ALTB <= '1'
        when Ainput6(0 to DATA_WIDTH - 2) < Binput6(0 to DATA_WIDTH - 2)
        else '0';

Process (ALUctl,Ainput,Binput)
    variable tempinput : std_logic_vector(0 to DATA_WIDTH - 1);

begin
-- with ALUctl(0 to 2) select
-- ResMux(0 to DATA_WIDTH - 1) <=
--      Ainput(0 to DATA_WIDTH - 1) and Binput(0 to DATA_WIDTH - 1)
--          when "000",
--      Ainput(0 to DATA_WIDTH - 1) or Binput(0 to DATA_WIDTH - 1)
--          when "001",
--      Ainput(0 to DATA_WIDTH - 1) + Binput(0 to DATA_WIDTH - 1)
```

```
--use          Change.MaxPlus2.all;
--Library Synopsys, IEEE;
--use          Synopsys.attributes.all;
IEEE library;
use           IEEE.STD_LOGIC_1164.all;
use           IEEE.STD_LOGIC_SIGNED.all;
--use          work.mydff.all;

entity memory is
  Generic(ADDR_WIDTH : integer := 32; DATA_WIDTH: integer := 32);
  port(
    rd_bus_D          : out std_logic_vector(0 to DATA_WIDTH - 1);
    ra_bus            : in  std_logic_vector(0 to ADDR_WIDTH - 1);
    wd_bus            : in  std_logic_vector(0 to DATA_WIDTH - 1);
    MemRead_DD        : in  std_logic;
    memwrite_DD       : in  std_logic;
    MemtoReg_DD       : in  std_logic;
    clock             : in  std_logic;
    Reset             : in  std_logic
    );
end entity dmemory;

--
-- DMEMORY architecture
--
```



```
architecture behavioral of dmemory is

        signal mem0,mem1,mem2,mem3                  : std_logic_vector(0 to DATA_WIDTH - 1);
        signal mem4,mem5,mem6,mem7                  : std_logic_vector(0 to DATA_WIDTH - 1);
        signal mem8                                 : std_logic_vector(0 to DATA_WIDTH - 1);
--      signal mem9,mem10,mem11                     : std_logic_vector(0 to DATA_WIDTH - 1);
--      signal mem12,mem13,mem14,mem15 : std_logic_vector(0 to DATA_WIDTH - 1);
--      signal mem16,mem17,mem18,mem19 : std_logic_vector(0 to DATA_WIDTH - 1);
--      signal mem20,mem21,mem22,mem23 : std_logic_vector(0 to DATA_WIDTH - 1);
--      signal mem24,mem25,mem26,mem27 : std_logic_vector(0 to DATA_WIDTH - 1);
--      signal mem28,mem29 : std_logic_vector(0 to DATA_WIDTH - 1);
--      signal mem30,mem31 : std_logic_vector(0 to DATA_WIDTH - 1);
        signal mux : std_logic_vector(0 to DATA_WIDTH - 1);
        signal mem0wr,mem1wr,mem2wr,mem3wr                  : std_logic;
        signal mem4wr,mem5wr,mem6wr,mem7wr,mem8wr           : std_logic;
--      signal mem9wr,mem10wr,mem11wr,mem12wr               : std_logic;
--      signal mem13wr,mem14wr,mem15wr                      : std_logic;
--      signal mem16wr,mem17wr,mem18wr,mem19wr              : std_logic;
--      signal mem20wr,mem21wr,mem22wr,mem23wr              : std_logic;
--      signal mem24wr,mem25wr,mem26wr                      : std_logic;
--      signal mem27wr,mem28wr,mem29wr                      : std_logic;
--      signal mem30wr,mem31wr                              : std_logic;
        signal rd_bus : std_logic_vector(0 to DATA_WIDTH - 1);


begin
-- Read Data Memory
with ra_bus(3 to DATA_WIDTH - 1) select
    mux(0 to DATA_WIDTH - 1) <= mem0(0 to DATA_WIDTH - 1)                   WHEN "00000",
                                mem1(0 to DATA_WIDTH - 1) WHEN "00001",
                                mem2(0 to DATA_WIDTH - 1) WHEN "00010",
                                mem3(0 to DATA_WIDTH - 1) WHEN "00011",
                                mem4(0 to DATA_WIDTH - 1) WHEN "00100",
                                mem5(0 to DATA_WIDTH - 1) WHEN "00101",
                                mem6(0 to DATA_WIDTH - 1) WHEN "00110",
                                mem7(0 to DATA_WIDTH - 1) WHEN "00111",
                                mem8(0 to DATA_WIDTH - 1) WHEN "01000",
                            -- mem9(0 to DATA_WIDTH - 1) WHEN "01001",
                            --mem10(0 to DATA_WIDTH - 1) WHEN "01010",
                            --mem11(0 to DATA_WIDTH - 1) WHEN "01011",
                            --mem12(0 to DATA_WIDTH - 1) WHEN "01100",
                            --mem13(0 to DATA_WIDTH - 1) WHEN "01101",
                            --mem14(0 to DATA_WIDTH - 1) WHEN "01110",
                            --mem15(0 to DATA_WIDTH - 1) WHEN "01111",
                            --mem16(0 to DATA_WIDTH - 1) WHEN "10000",
                            --mem17(0 to DATA_WIDTH - 1) WHEN "10001",
                            --mem18(0 to DATA_WIDTH - 1) WHEN "10010",
```

```
mem8wr          <= '1'
            When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="01000"))
            ELSE '0';
--mem9wr           <= '1'
--          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="01001"))
--          ELSE '0';
--mem10wr <= '1'
--          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="01010"))
--          ELSE '0';
--mem11wr <= '1'
--          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="01011"))
--          ELSE '0';
--mem12wr <= '1'
--          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="01100"))
--          ELSE '0';
--mem13wr <= '1'
--          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="01101"))
--          ELSE '0';
--mem14wr <= '1'
--          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="01110"))
--          ELSE '0';
--mem15wr <= '1'
--          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="01111"))
--          ELSE '0';
--mem16wr <= '1'
--          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="10000"))
--          ELSE '0';
--mem17wr <= '1'
--          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="10001"))
--          ELSE '0';
--mem18wr <= '1'
--          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="10010"))
--          ELSE '0';
--mem19wr <= '1'
--          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="10011"))
--          ELSE '0';
--mem20wr <= '1'
--          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="10100"))
--          ELSE '0';
--mem21wr <= '1'
--          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="10101"))
--          ELSE '0';
--mem22wr <= '1'
--          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="10110"))
--          ELSE '0';
--mem23wr <= '1'
--          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="10111"))
--          ELSE '0';
```

```
                                        --mem19(0 to DATA_WIDTH - 1) WHEN "10011",
                                        --mem20(0 to DATA_WIDTH - 1) WHEN "10100",
                                        --mem21(0 to DATA_WIDTH - 1) WHEN "10101",
                                        --mem22(0 to DATA_WIDTH - 1) WHEN "10110",
                                        --mem23(0 to DATA_WIDTH - 1) WHEN "10111",
                                        --mem24(0 to DATA_WIDTH - 1) WHEN "11000",
                                        --mem25(0 to DATA_WIDTH - 1) WHEN "11001",
                                        --mem26(0 to DATA_WIDTH - 1) WHEN "11010",
                                        --mem27(0 to DATA_WIDTH - 1) WHEN "11011",
                                        --mem28(0 to DATA_WIDTH - 1) WHEN "11100",
                                        --mem29(0 to DATA_WIDTH - 1) WHEN "11101",
                                        --mem30(0 to DATA_WIDTH - 1) WHEN "11110",
                                        --mem31(0 to DATA_WIDTH - 1) WHEN "11111",
                                "11111111111111111111111111111111" WHEN others;

    -- Mux to skip data memory for Rformat instructions
    rd_bus(0 to DATA_WIDTH - 1) <= ra_bus(0 to DATA_WIDTH - 1)
                                WHEN (MemtoReg_DD='0') ELSE mux WHEN (MemRead_DD='1')
                                ELSE "00000000000000000000000000000000";

    -- write to data memory?
    -- The following code sets an initial value and replaces the next line
    -- dff_v(wd_bus,clock AND Memwrite_DD AND (ra_bus(0 to 2)="000"),mem0);

    memory      <= '1'
                When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="00000"))
                ELSE '0';
    mem1wr      <= '1'
                When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="00001"))
                ELSE '0';
    mem2wr      <= '1'
                When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="00010"))
                ELSE '0';
    mem3wr      <= '1'
                When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="00011"))
                ELSE '0';
    mem4wr      <= '1'
                When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="00100"))
                ELSE '0';
    mem5wr      <= '1'
                When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="00101"))
                ELSE '0';
    mem6wr      <= '1'
                When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="00110"))
                ELSE '0';
    mem7wr      <= '1'
                When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="00111"))
                ELSE '0';
    --mem24wr <= '1'
    --          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="11000"))
    --          ELSE '0';
    --mem25wr <= '1'
    --          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="11001"))
    --          ELSE '0';
    --mem26wr <= '1'
    --          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="11010"))
    --          ELSE '0';
    --mem27wr <= '1'
    --          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="11011"))
    --          ELSE '0';
    --mem28wr <= '1'
    --          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="11100"))
    --          ELSE '0';
    --mem29wr <= '1'
    --          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="11101"))
    --          ELSE '0';
    --mem30wr <= '1'
    --          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="11110"))
    --          ELSE '0';
    --mem31wr <= '1'
    --          When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="11111"))
    --          ELSE '0';

    process
    begin
    wait until clock'event and clock='1';
        if (reset = '1') then
            --mem0 <= To_stdlogicvector(X"55");
            --mem1 <= To_Stdlogicvector(X"55");
            --mem2 <= To_Stdlogicvector(B"10101010");
            rd_bus_D(0 to DATA_WIDTH - 1) <= "00000000000000000000000000000000";
        else

    -- We may want to change these to use their own flip-flops for
    -- performance reasons.                     see cmpe3510 dmemory module

        if mem0wr= '1'
            then mem0(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
            else mem0(0 to DATA_WIDTH - 1) <= mem0(0 to DATA_WIDTH - 1);
        end if;
        if mem1wr= '1'
            then mem1(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
            else mem1(0 to DATA_WIDTH - 1) <= mem1(0 to DATA_WIDTH - 1);
        end if;
        if mem2wr= '1'
            then mem2(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
```



```
            else mem2(0 to DATA_WIDTH - 1) <= mem2(0 to DATA_WIDTH - 1);
    end if;
    if mem3wr= '1'
        then mem3(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
        else mem3(0 to DATA_WIDTH - 1) <= mem3(0 to DATA_WIDTH - 1);
    end if;
    if mem4wr= '1'
        then mem4(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
        else mem4(0 to DATA_WIDTH - 1) <= mem4(0 to DATA_WIDTH - 1);
    end if;
    if mem5wr= '1'
        then mem5(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
        else mem5(0 to DATA_WIDTH - 1) <= mem5(0 to DATA_WIDTH - 1);
    end if;
    if mem6wr= '1'
        then mem6(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
        else mem6(0 to DATA_WIDTH - 1) <= mem6(0 to DATA_WIDTH - 1);
    end if;
    if mem7wr= '1'
        then mem7(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
        else mem7(0 to DATA_WIDTH - 1) <= mem7(0 to DATA_WIDTH - 1);
    end if;
    if mem8wr= '1'
        then mem8(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
        else mem8(0 to DATA_WIDTH - 1) <= mem8(0 to DATA_WIDTH - 1);
    end if;
```

```
                Out_Inst : out std_logic_vector(0 to 31);
            Video_Data_D : out std_logic_vector(0 to 7);
        Video_Write_DD : out std_logic;
                switch : in          std_logic_vector(0 to 7));
end TOP_SPIM;

architecture BEHAVIORAL of TOP_SPIM is

    component Ifetch
    port(Instruction_D : out std_logic_vector(0 to 31);
            PCadd_D             :out std_logic_vector(0 to 7);
            Addresult_D,AddResultBR_D : in                  std_logic_vector(0 to 7);
            Branch_D            : in    std_logic_vector(0 to 1);
            clock reset         : in    std_logic;
            Zero_D,jump_D : in          std_logic;
            flush               :out std_logic;
            PCout               :out std_logic_vector(0 to 7));
    end component;

    component Idecode
    port(rr1d_bus_D             :out std_logic_vector(0 to 7);
            rr2d_bus_D          :out std_logic_vector(0 to 7);
            Instruction_D : in          std_logic_vector(0 to 31);
            wrd_bus             : in    std_logic_vector(0 to 7);
            RegWrite_D,RegWrite_DD,RegWrite_DDD                 : in    std_logic;
            RegDst              : in    std_logic;
```

```
--if mem9wr= '1'
--        then mem9(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
--        else mem9(0 to DATA_WIDTH - 1) <= mem9(0 to DATA_WIDTH - 1);
--end if;
--if mem10wr= '1'
--        then mem10(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
--        else mem10(0 to DATA_WIDTH - 1) <= mem10(0 to DATA_WIDTH - 1);
--end if;
--if mem11wr= '1'
--        then mem11(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
--        else mem11(0 to DATA_WIDTH - 1) <= mem11(0 to DATA_WIDTH - 1);
--end if;
--if mem12wr= '1'
--        then mem12(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
--        else mem12(0 to DATA_WIDTH - 1) <= mem12(0 to DATA_WIDTH - 1);
--end if;
--if mem13wr= '1'
--        then mem13(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
--        else mem13(0 to DATA_WIDTH - 1) <= mem13(0 to DATA_WIDTH - 1);
--end if;
--if mem14wr= '1'
--        then mem14(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
--        else mem14(0 to DATA_WIDTH - 1) <= mem14(0 to DATA_WIDTH - 1);
--end if;
--if mem15wr= '1'
--        then mem15(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
--        else mem15(0 to DATA_WIDTH - 1) <= mem15(0 to DATA_WIDTH - 1);
--end if;
--if mem16wr= '1' then mem16 <= wd_bus; else mem16 <= mem16; end if;
--if mem17wr= '1' then mem17 <= wd_bus; else mem17 <= mem16; end if;
--if mem18wr= '1' then mem18 <= wd_bus; else mem18 <= mem18; end if;
--if mem19wr= '1' then mem19 <= wd_bus; else mem19 <= mem19; end if;
--if mem20wr= '1' then mem20 <= wd_bus; else mem20 <= mem20; end if;
--if mem21wr= '1' then mem21 <= wd_bus; else mem21 <= mem21; end if;
--if mem22wr= '1' then mem22 <= wd_bus; else mem22 <= mem22; end if;
--if mem23wr= '1' then mem23 <= wd_bus; else mem23 <= mem23; end if;
--if mem24wr= '1' then mem24 <= wd_bus; else mem24 <= mem24; end if;
--if mem25wr= '1' then mem25 <= wd_bus; else mem25 <= mem25; end if;
--if mem26wr= '1' then mem26 <= wd_bus; else mem26 <= mem26; end if;
--if mem27wr= '1' then mem27 <= wd_bus; else mem27 <= mem27; end if;
--if mem28wr= '1' then mem28 <= wd_bus; else mem28 <= mem28; end if;
--if mem29wr= '1' then mem29 <= wd_bus; else mem29 <= mem29; end if;
--if mem30wr= '1' then mem30 <= wd_bus; else mem30 <= mem30; end if;
--if mem31wr= '1' then mem31 <= wd_bus; else mem31 <= mem31; end if;
rd_bus_D(0 to DATA_WIDTH - 1) <= rd_bus(0 to DATA_WIDTH - 1);

    end if;

end process;
behavioral end architecture;
```

B.2.5 Integration of Blocks

```
-- TOP_SPIM module
--
-- VHDL synthesis and simulation model of MIPS machine
-- as described in chapter 5 of Patterson and Hennessey
-- NOTE: Data paths limited to 8 bits to speed synthesis and
-- simulation.                    Registers limited to 8 bits and $R0..$R7
-- Program and Data memory limited to locations 0.7

IEEE Library;
use IEEE.STD_LOGIC_1164.all;

entity TOP_SPIM is

port(reset,clock : in                    std_logic;
                    PC : out std_logic_vector(0 to 7);
                    Zero_D                :out std_logic;
                    ADDResult_D,AddResultBR_D : out std_logic_vector(0 to 7);
                    PCadd_D               : in    std_logic_vector(0 to 7);
                    Extend_D              :out std_logic_vector(0 to 7);
                    Func_D                :out std_logic_vector(0 to 5);
                    ALUSelA_D,ALUSelB_D : out std_logic_vector(0 to 1);
                    ALUResult_D           : in    std_logic_vector(0 to 7);
                    regvalue              :out     std_logic_vector(0 to 7);
                    switch                : in    std_logic_vector(0 to 7);
                    jumper, already       : in    std_logic;
                    char_mode_D           :out std_logic_vector(0 to 1);
--                  Accept_Key            : in    std_logic;
--                  Key_Data              : in    std_logic_vector (0 to 5);
--                  Key_Stroke            : in    std_logic;
                    clock reset           : in    std_logic);
end component;

component control
port(Op                   : in    std_logic_vector(0 to 5);
                    RegDst                :out std_logic;
                    ALUSrc_D              :out std_logic;
                    MemtoReg_DD           :out std_logic;
                    RegWrite_D,RegWrite_DD,RegWrite_DDD : out std_logic;
                    MemRead_DD            :out std_logic;
                    MemWrite_DD           :out std_logic;
                    Branch_D              :out std_logic_vector(0 to 1);
                    ALUop0_D              :out std_logic;
                    ALUop1_D              :out std_logic;
                    JumpR,jal,jump_D          :out std_logic;
--                  Accept_Key                :out std_logic;
                    Video_Write_DD            :out std_logic;
                    clock,reset,flush : in                  std_logic);
end component;

component          run
port(Readdata1                : in    std_logic_vector(0 to 7);
                    Readdata2             : in    std_logic_vector(0 to 7);
                    rr2d_bus_DD           :out std_logic_vector(0 to 7);
                    Extend_D              : in    std_logic_vector(0 to 7);
                    Func_D                : in    std_logic_vector(0 to 5);
                    ALUOp0_D              : in    std_logic;
                    ALUOp1_D              : in    std_logic;
                    ALUSrc_D              : in    std_logic;
                    ALUResult_D           :out std_logic_vector(0 to 7);
                    wrd_bus               : in    std_logic_vector(0 to 7);
                    ALUSelA_D,ALUSelB_D, char_mode : in                  std_logic_vector(0 to 1);
                    video_data_D : out std_logic_vector(0 to 7);
                    clock reset           : in    std_logic);
end component;

dmemory component
port(rd_bus_D                 :out std_logic_vector(0 to 7);
                    ra_bus                : in    std_logic_vector(0 to 7);
                    wd_bus                : in    std_logic_vector(0 to 7);
                    MemRead_DD, Memwrite_DD, MemtoReg_DD : in                  std_logic;
                    clock reset           : in    std_logic);
end component;

signal PCadd_D                :    std_logic_vector(0 to 7);
signal rr1d_bus_D,rr2d_bus_D :                  std_logic_vector(0 to 7);
signal rr2d_bus_DD            :    std_logic_vector(0 to 7);
signal Extend_D               :    std_logic_vector(0 to 7);
signal Func_D                 :    std_logic_vector(0 to 5);
signal Addresult_D,AddResultBR_D :                  std_logic_vector(0 to 7);
signal ALUresult_D            :    std_logic_vector(0 to 7);
Branch_D signal               :    std_logic_vector(0 to 1);
signal Zero_D,jump_d :               std_logic;
signal wrd_bus                :    std_logic_vector(0 to 7);
signal RegWrite_D,RegWrite_DD,RegWrite_DDD :                  std_logic;
signal RegDst                 :    std_logic;
signal ALUSrc_D               :    std_logic;
MemtoReg_DD signal            :    std_logic;
signal MemRead_DD             :    std_logic;
signal MemWrite_DD            :    std_logic;
                    wd_bus(0 to 7) => rr2d_bus_DD(0 to 7),
                    MemRead_DD            => MemRead_DD,
                    memwrite_DD           => MemWrite_DD,
                    MemtoReg_DD           => MemtoReg_DD,
```

```
    signal ALUop_D                  :   std_logic_vector(0 to 1);
    signal Instruction_D :              std_logic_vector(0 to 31);
    signal ALUSelA_D                :   std_logic_vector(0 to 1);
    signal ALUSelB_D                :   std_logic_vector(0 to 1);
    signal jumpr,jal,flush :                std_logic;
    signal char_mode                :   std_logic_vector(0 to 1);
    signal regvalue                 :   std_logic_vector(0 to 7);
-- signal accept_key                :   std_logic;

    begin
    Out_Inst(0 to 31) <= Instruction_D(0 to 31);

    IFE: Ifetch
    port map (Instruction_D                         => Instruction_D(0 to 31),
              PCadd_D                               => PCadd_D(0 to 7),
              Addresult_D(0 to 7)                   => Addresult_D(0 to 7),
              AddResultBR_D(0 to 7) => AddResultBR_D(0 to 7),
              Branch_D(0 to 1)                      => Branch_D(0 to 1),
              clock                                 => clock,
              reset                                 => reset,
              Zero_D                                => Zero_D,
              Jump_D                                => Jump_D,
              flush                                 => flush,
              PCout                                 => PC(0 to 7));

    ID: Idecode
    port map (rr1d_bus_D                            => rr1d_bus_D(0 to 7),
              rr2d_bus_D                            => rr2d_bus_D(0 to 7),
              Instruction_D(0 to 31) => Instruction_D(0 to 31),
              wrd_bus(0 to 7)                       => wrd_bus(0 to 7),
              RegWrite_D                            => RegWrite_D,
              RegWrite_DD                           => RegWrite_DD,
              RegWrite_DDD                          => RegWrite_DDD,
              RegDst                                => RegDst,
              Zero_D                                => Zero_D,
              ADDResult_D                           => ADDResult_D(0 to 7),
              AddResultBR_D                         => ADDResultBR_D(0 to 7),
              PCadd_D(0 to 7)                       => PCadd_D(0 to 7),
              Extend_D                              => Extend_D(0 to 7),
              Func_D                                => Func_D(0 to 5),
              ALUSelA_D                             => ALUSelA_D(0 to 1),
              ALUSelB_D                             => ALUSelB_D(0 to 1),
              ALUResult_D(0 to 7)                   => ALUResult_D(0 to 7),
              jumper                                => jumpr,
              already                               => already,
              regvalue                              => regvalue(0 to 7),
              switch                                => switch(0 to 7),
              char_mode_D                           => char_mode(0 to 1),
              clock => clock, reset => reset);

    CTL:            control
    port map (Op(0 to 5)                    => Instruction_D(0 to 5),
              RegDst                        => RegDst,
              ALUSrc_D                      => ALUSrc_D,
              MemtoReg_DD                   => MemtoReg_DD,
              RegWrite_D                    => RegWrite_D,
              RegWrite_DD                   => RegWrite_DD,
              RegWrite_DDD                  => RegWrite_DDD,
              MemRead_DD                    => MemRead_DD,
              MemWrite_DD                   => MemWrite_DD,
              Branch_D                      => Branch_D(0 to 1),
              ALUop0_D                      => ALUop_D(0),
              ALUop1_D                      => ALUop_D(1),
              jump_D                        => jump_D,
              jumper                        => jumpr,
              already                       => already,
--            accept_key                    => accept_key,
              Video_Write_DD => video_write_DD,
              clock                         => clock,
              reset                         => reset,
              flush                         => flush);

    EXE:        run
    port map (Readdata1(0 to 7) => rr1d_bus_D(0 to 7),
              Readdata2(0 to 7) => rr2d_bus_D(0 to 7),
              rr2d_bus_DD                   => rr2d_bus_DD(0 to 7),
              Extend_D(0 to 7)              => Extend_D(0 to 7),
              Func_D(0 to 5)                => Func_D(0 to 5),
              ALUOp0_D                      => ALUop_D(0),
              ALUOp1_D                      => ALUop_D(1),
              ALUSrc_D                      => ALUSrc_D,
              ALUResult_D                   => ALUResult_D(0 to 7),
              wrd_bus(0 to 7)               => wrd_bus(0 to 7),
              ALUSelA_D(0 to 1) => ALUSelA_D(0 to 1),
              ALUSelB_D(0 to 1) => ALUSelB_D(0 to 1),
              Video_date_D                  => video_data_D(0 to 7),
              char_mode                     => char_mode(0 to 1),
              clock => clock, reset => reset);

    MEM:        dmemory
    port map (rd_bus_D                      => wrd_bus(0 to 7),
              ra_bus(0 to 7) => ALUResult_D(0 to 7),
              clock                         => clock,
              reset                         => reset);
behavioral end;
```

Appendix C

DLX Test Programs

Example

AND THIS appendix presents an example program and introduces the dlxasm program. O dlxasm program was used in this work to generate files with programs for be used in testing the SelfHDL implementation. It is an assembler assembler created from the dlxsim program also developed by [HP96] and available via FTP anonymously in “max.stanford.edu” in the “pub/hennessy-patterson.software” directory. the dlxasm was also created by the team of [Ash04] being also available in the same reference.

We use dlxasm to create the image files that are used to initialize the objects.

memoryFile.

The program is very simple to operate. The command line for the dlxasm program is simply: dlxasm [-o file.out] file.s ....

In section C.1 we have an example program, and in section C.2 we have the generated code.

C.1 Sample Program

```
.text 0x0000                    ; Reset
        jal program
                                ; never executed
        trap        0x104; Wait until Write-Buffer is empty
        trap        0           ; Halt

.text 0x0100                    ; Store Transfer Error
        trap        0x104; Wait until Write-Buffer is empty
        trap        0           ; Halt

.text 0x0A00                    ; External Interrupt
        lw r30, -0xc0(r0) ; 0xffff_ff40, Interrupt Ack
                                        ; r30 can be used to determine
                                        ; the device that caused the
                                        ; interrupt
        nop
        rfe 0                   ; Assembler needs dummy parameter

.text 0x2000
Program:                        ; Evaluate some Fibonacci numbers
        add r1,r0,List + 4 ; r1: Pointer to list of
                                                ; numbers (Word)
        addui r2,r0,1                   ; initialise register
        sw -0x80(r0),r2                 ; 0xffff_ff80
```

C.2 Compiled Program

```
00000000 0c001ffc
00000004 44000104
00000008 44000000
00000100 44000104
00000104 44000000
00000a00 8c1eff40
00000a08 43fff5f4
00002000 24013004
00002004 2402001
00002008 ac02ff80
0000200c 00021820
00002010 9404205c
00002014 ac22fffc
00002018 ac230000
0000201c 00431020
00002020 ac220004
00002024 24210008
00002028 00431820
0000202c 2884001
00002030 1480ffe4
00002034 24073000
00002038 44000104
0000203c ac07ff00
00002040 24070014
```

```
                                        ; Interrupt-Enable reg.,                00002044 44000104
                                        ; enable Interrupt                      00002048 ac07ff04
        add r3,r0,r2                    ; initialise register                   0000204c 44000104
        lhu r4,CountLoops(r0) ; Run loop 3 teams                                00002050 ac00ff08
        sw -4(r1),r2                    ; Store first Fibonacci                 00002054 ac00f000
                                        ; number (1)                            00002058 4be00000
Loop:                           ; Two Fibonacci numbers per run                 0000205c 00030000
        sw 0(r1),r3
        add r2, r2, r3          ; Compute next Fibonacci number
        sw 4(r1),r2
        addui r1,r1,8           ; Increment pointer
        add r3,r2,r3            ; Compute next Fibonacci number
        sub r4,r4,1             ; Decrement counter
                ; Yes, it would be better to do this two
                ; earlier. But in this
                ; program I want to demonstrate unresolved
                ; branches and speculative execution.
        bnez r4,Loop            ; Run again?

; Print list to file, output file is 'dlx.dump'
        addui r7,r0,List
        trap 0x104                      ; Wait until Write-Buffer
                                        ; is empty, avoid merge
        sw -0x100(r0),r7 ; Start address of memory block
        addui r7,r0.20
        trap 0x104
        sw -0xfc(r0),r7 ; Number of elements (20)
        trap 0x104
        sw -0xf8(r0),r0 ; Start transfer of words

                ; Force Store Error
        sw -0x1000(r0),r0 ; The exception handler will
                                ; halt the DLX

        jr r31                  ; Done
CountLoops: .word 0x00030000

.text 0x3000
List:
```

Bibliography

[ABC + 00] Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, John Maloney, Randall B. Smith, David Ungar, and Mario Wolczko. The SELF 4.1 Programmer's Reference Manual. Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA, 2000.

[ABS02] Ahmad Alkhodre, Jean-Philippe Babau, and Jean-Jacques Schwartz. Modeling of real-time constraints using SDL for embedded systems design. Computing & Control Engineering Journal, 13(4):189–196, Aug 2002.

[AIG99] Dai Araki, Tadatoshi Ishii, and Daniel D. Gajski. Rapid prototyping with HW/SW codesign tool. In IEEE Conference and Workshop on Computer-Based Engineering Systems, 1999. Proceedings, ECBS'99., pages 114–121, 1999.

[Ash04] Peter J. Ashenden. DLX: Generic 32-bit RISC processor. http://www.eda.org-/rassp/vhdl/models/processors/dlx.tar.gz, 2004.

[Ber02] Reinaldo A. Bergamaschi. Bridging the domains of high-level and logic synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 21(5):582–596, May 2002.

[BHSV90] RK Brayton, GD Hachtel, and AL Sangiovanni-Vincentelli. Multilevel logic synthesis. Proceedings of the IEEE, 78(2):264–300, February 1990.

[CB95] Anantha P. Chandrakasan and Robert W. Brodersen. Low Power Digital CMOS Design. Kluwer Academic Publishers, Boston, 1995.

[Cha92] Craig Chambers. The Design and Implementation of the SELF Compiler, and Optimizing Compiler for Object-Oriented Programming Languages. PhD thesis, Department of Computer Science, Stanford University, March 1992.

[CMP91] Paolo Camurati, Tiziana Marcaria, and Paolo Prinetto. Formal verification of design correctness of sequential circuits based on theorem provers. In Proceedings of the 5th Annual European Computer Conference CompEuro'91. Advanced Computer Technology, Reliable Systems and Applications, pages 322–326, 1991.

[Cor04] Intel Corporation. Moore's law. http://www.intel.com/research/silicon/moore-slaw.htm, 2004.

[CU90a] Craig Chamber and David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementstation, pages 150–164, June 1990.

[CU90b] Bay-Wey Chang and David Ungar. Experiencing SELF objects: An object-based artificial reality. The Self Papers, Computer Systems Laboratory, Stanford University city, 1990.

[CU91] Craig Chambers and David Ungar. Making pure object-oriented languages practilime. In OOPSLA'91 Conference Proceedings, pages 1–15, October 1991.

[CU93] Bay-Wei Chang and David Ungar. Animation: From cartoons to the user interface. In User Interface Software and Technology UIST'93 Conference Proceedings, pages 45–55, November 1993.

[CUCH91] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. parents are shared parts: Inheritance and encapsulation in self. LISP AND SYMBOLIC COMPUTATION: An International Journal. Kluwer Academic Publishers, 4(3), June 1991.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee.

a dynamically-typed object-oriented language based on prtotypes. In OOPSLA'89 Conference Proceedings, pages 49–90, October 1989.

[CUS95] Bay-Wei Chang, David Ungar, and Randall B. Smith. Visual Object-Oriented Programming, chapter Getting Close to Objects: Object-Focused Programming Environments, pages 185–198. Prentice-Hall, 1995.

[DG90] Nikil D. Dutt and Daniel D. Gajski. Design synthesis and silicon compilation. IEEE Design & Test of Computers, 7(6):8–23, December 1990.

[DH89] Doron Drusinsky and David Harel. Using statecharts for hardware description and synthesis. IEEE Transactions on Computer-Aided Design, 8(7):798–807, July 1989.

[dMG97] Giovanni de Micheli and Rajesh K. Gupta. Hardware/software co-design. Proceed Dings of the IEEE, 85(3):349–365, March 1997.

[FDK00] Peter L. Flake, Simon J. Davidmann, and David J. Kelf. Measuring design languages for system-on-chip design. Technical report, Co-Design Automation, Inc., San Jose, CA 95113-1295 http://www.co-design.com/ , 2000.

[Gad99] Hans-Georg Gadamer. Truth and Method: Fundamental traits of a hermeneu- philosophical ethics. Editora Vozes, 1999.

[Gaj88] Daniel D. Gajski. Silicon Compilation. Addison-Wesley, 1988.

[Gaj93] Daniel D. Gajski. Design process beyond ASICs. In European Conference on Design Automation, 1993, with European Event in ASIC Design., pages 3–4, February 1993.

[GGPY89] Patrick P. Gelsinger, Paolo A. Gargini, Gerhard H. Parker, and Albert YC Yu.

Microprocessors circa 2000. IEEE Spectrum, October 1989.

[GKL99] Abhijit Ghosh, Joaquim Kunkel, and Stan Liao. Hardware synthesis from c/c++. In Proceedings of the Design, Automation and Test in Europe Conference and Exhibition 1999, pages 387–389, 1999.

[GL97] Rajesh K. Gupta and Stan Y. Liao. Using a programming language for digital system design. IEEE Design & Test of Computers, 14(2):72–80, April-June 1997.

[Gly04] Harald Gliebe. self/x86 for Linux/Microsoft. http://www.gliebe.de/self/index.html , 2004.

[Gov95] Sriram Govindarajan. Scheduling algorithms for high-level synthesis. Term Paper ECE831, Dept. of ECECS, University of Cincinatti, Cincinatti, OH 45221-0030, March 1995.

[GR83] Adele Goldberg and David Robson. Smalltalk-80: The Language and its Implementmentation. Addison-Wesley, 1983.

[GR94] Daniel D. Gajski and Loganath Ramachandran. Introduction to high-level synthsis. IEEE Design & Test of Computers, 11(4):44–54, Winter 1994.

[Gup93] Rajesh Kumar Gupta. Co-Synthesis of Hardware and Software for Digital Embedded Systems. Tech report csl-tr-94-614, Department of Electrical Engineering of Stanford University, December 1993.

[Gup02] Pallav Gupta. Hardware-software codesign. IEEE Potentials, 20(5):31–32, Dec-Jan 2002.

[GV95] Daniel D. Gajski and Frank Vahid. Specification and design of embedded hardware-

software systems. IEEE Design & Test of Computers, 12(1):53–67, Spring 1995.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. dynamically-typed optimizing object-oriented languages with polymorphic inline caches. In ECOOP'91 Confesence Proceedings, 1991.

[Höl94] Urs Hölzle. Adaptive optimization for Self: Reconciling High Performance with Exploratory Programming. PhD thesis, Computer Science Department of Stanford University, August 1994.

[HO97] Rachid Helaihel and Kunle Olukotun. Java as a specification language for hardware-software systems. In 1997 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers, pages 690–697, 1997.

[HP96] John L. Hennessy and David A. Patterson. Computer Architecture, A Quantitative Approach. Morgan Kaufmann Publishers, Inc, San Francisco, California, second edition, 1996.

[HU94a] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In Proceedings of the ACM SIGPLAN'94 Conference on Pro-Gramming Language Design and Implementation, 1994.

[HU94b] Urs Hölzle and David Ungar. The third-generation SELF implementation: Reconciling responsiveness with performance. In Proceedings of the ACM OOPSLA'94 Conference, 1994.

[ITR01] ITRS. International technology roadmap for semiconductors - design - 2001 edition.

http://public.itrs.net/Files/2001ITRS/Home.htm, 2001.

[Joy96] Ian Joyner. C++??: A critique of c++ and programming and languages trends of the 1990s (3rd edition). ftp://ftp.brown.edu/pub/c++/C++-Critique-3ed.PS.gz, 1996.

[Keu91] Kurt Keutzer. The need for formal verification in hardware design and what formal verification has not done for me lately. In International Workshop on HOL Theorem Proving System and Its Applications, pages 77–86, 1991.

[KM90] David Ku and Giovanni De Micheli. Hardwarec - a language for hardware design. version 2.0. Technical Report CSL-TR-90-419, Computer System Laboratory, Departs of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305-4055, April 1990.

[KR00] Tommy Kuhn and Wolfgang Rosenstiel. Java based object oriented hardware spepurification and synthesis. In Proceedings of the ASP-DAC 2000. Asia and South Pacific Design Automation Conference, pages 579–581, 2000.

[Kum98] Ramayya Kumar. Formal verification of hardware: Misconception and reality. In Wescon/98, pages 135–138, 1998.

[Lar00] Craig Larman. Using UML and Standards: An Introduction to Analysis and Design Object Oriented. Bookman, Porto Alegre, reprint 2002 edition, 2000.

[LHH02] Antti Laitinen, Marko Hännikäinen, and Timo Hämäläinen. Using SDL as a tool for system simulations. In IEEE International Symposium on Circuits and Systems, BAITS 2002, volume 5, pages 17–20, 2002.

[Li96] Jian Li. Timed decision tables: A model for embedded system representation and

optimization. MsC Thesis, Technical Report n the UIUCDCS-R-96-1971, University of Illinois at Urbana-Champaign, 1996.

[Mau96] Peter M. Maurer. Is compiled simulation really faster than interpreted simulation? In Proceedings of the 9th International Conference on VLSI Design, 1996, pages 303–306, 1996.

[MC80] Carver Mead and Lynn Conway. Introduction to VLSI Systems. Addison-Wesley, 1980.

[McK01] Michael D. McKinney. Integrating Perl, Tcl and C++ into simulation-based verification location environments. In Proceedings of the Sixth IEEE International Workshop on High-Level Design Validation and Test, pages 19–24, 2001.

[MF00] Annette Muth and Georg Färber. SDL as a system level specification language for application specific hardware in a rapid prototyping environment. In Proceedings of the 13th International Symposium on System Synthesis, pages 157–162, 2000.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. Electronics, 38(8), April 1965.

[Moo03] Gordon E. Moore. No exponential is forever... but we can delay“forever”. In IEEE International Solid-State Circuits Conference, ISSCC, February 2003.

[MPS04] Doug McAlister, Gauthier Phillippart, and Michael Sugg. DLX computer have design using Altera's MaxplusII. http://users.ece.gatech.edu:80/ hamblen/ALTERA/onedge/gatech/altera.htm , 2004.

[Nar96] Sanjiv Narayan. Requirements for specification of embedded systems. In Proceedings of the Ninth Annual IEEE International ASIC Conference and Exhibit, pages 133–137, 1996.

[NG93] Sanjiv Narayan and Daniel D. Gajski. Features supporting system-level specification in HDLs. In Design Automation Conference, 1993, wwith EURO-VHDL'93. Proceedings EURO-DAC'93, European, 1993, pages 540–545, 1993.

[NN99] Jo˜ao Navarro and Wilhelmus AM Van Noije. The 1.6-ghz dual modulus prescaler using the extended true-single-phase-clock CMOS circuit technique (E-TSPC). IEEE Journal of Solid-State Circuits, 34(1):97–102, 1999.

[NN02] Jo˜ao Navarro and Wilhelmus AM Van Noije. Extended TSPC structures with double input/output data throughput for gigahertz CMOS circuit design. IEEE Transaction on Very Large Scale Integration (VLSI) Systems, 10(3):301–308, 2002.

[OHO98] Kunle Olukotun, Mark Heinrich, and David Ofelt. Digital system simulation: Methodologies and examples. In Design Automation Conference, 1998. dings, 1998, pages 658–663, 1998.

[OMG00] OMG - Object Management Group. OMG Unified Modeling Language Specification, first edition - version 1.3 edition, March 2000.

[PPE + 97] Yale N. Patt, Sanjay J. Patel, Marius Evers, Daniel H. Friendly, and Jared Stark. One billion transistors, one uniprocessor, one chip. Computer, 30(9):51–57, September 1997.

[PW88] Lewis J. Pinson and Richard S. Wiener. An Introduction to Object-Oriented Programming and SmallTalk. Addison-Wesley Publishing Company, 1988.

[Sag00] So A. Sagahyroon. From ahpl to vhdl: A course in hardware description languages. IEEE Transaction on Education, 43(4):449–454, November 2000.

[Sim97] Dezs˜o Sima. Superscalar instruction issue. IEEE Micro, 17(5):28–39, September/October 1997.

[SM96] A. Shah and H. Mathkour. Developing an application using SELF programming language. In Workshop on Prototype Based Object Oriented Programming, ECO-

OP'96, 1996.

[Smi97] Douglas J. Smith. HDL Chip Design: A practical guide for designing, synthesizing and simulating ASICs and FPGAs using VHDL or Verilog. Doone Publications, 1997.

[Soc93] IEEE Computer Society. IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std logic 1164). The Institute of Electrical and Electronics Engineering Neers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, ieee std 1164-1993 edition, March 1993.

[Soc96] IEEE Computer Society. IEEE Standard VHDL Mathematical Packages. The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, ieee std 1076.2-1996 edition, September 1996.

[Soc97] IEEE Computer Society. IEEE Standard VHDL Synthesis Packages. The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, ieee std 1076.3-1997 edition, March 1997.

[Soc99] IEEE Computer Society. IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis. The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, ieee std 1076.6-1999 edition, September 1999.

[Soc01] IEEE Computer Society. IEEE Standard Verilog Hardware Description Language. The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New

York, NY 10016-5997, USA, ieee std 1364-2001 edition, September 2001.

[Soc02] IEEE Computer Society. IEEE Standard VHDL Language Reference Manual. The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, ieee std 1076-2002 edition, May 2002.

[SS98] Bruce Shriver and Bennett Smith. The Anatomy of a High-Performance Microprocessor: A System Perspective. IEEE Computer Society, 1998.

[Sut99] Jeff Sutherland. A history od objectct-oriented programming languages and their impact on program design and software development. http://jeffsutherland.com-/papers/Rans/OOlanguages.pdf , December 1999.

[Syn02] Inc. Synopsys. System C - Version 2.0 - User's Guide. http://www.systemc.org , 2002.

[TB92] Lars E. Thon and Robert W. Brodersen. C to silicon compilation. In Proceedings of the 1992 IEEE Custom Integrated Circuits Conference, pages 11.7.1–11.7.4, 1992.

[UCCH91] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. LISP AND SYMBOLIC COMPUTATION: An Inter-National Journal. Kluwer Academic Publishers, 4(3), June 1991.

[US87] David Ungar and Randall Smith. Self: The power of simplicity. In OOPSLA'87 Conference Proceedings, volume 22, pages 227–241, 1987.

[US91] David Ungar and Randall B. Smith. SELF: The power of simplicity. LISP AND SYMBOLIC COMPUTATION: An International Journal. Kluwer Academic Publishers, 4(3), June 1991.

[USCH92] David Ungar, Randall B. Smith, Craig Chambers, and Urs Holzle. object, same-sage, and performance: How they coexist in self. IEEE Computer, 25(10):53–64, October 1992.

[Vel00] Yosl Veller. Another approach to system level design. Proceedings of the VHDL International Users Forum, Fall Workshop, pages 25–31, 2000.

[VNG95] Frank Vahid, Sanjiv Narayan, and Daniel D. Gajski. SpecCharts: The VHDL front-end for embedded systems. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 14(6):694–706, June 1995.

[WE88] Neil HE Weste and Kamran Eshraghian. Principles of CMOS VLSI Design: a System Perspective. Addison-Wesley, reprinted in June 1988 edition, 1988.

[Yan02] Jan-Ti Yang. Rules and suggestions for ASIC design in HDL. Proceedings of ICSP'02, 2002.

[Yea98] Gary K. Yeap. Practical Low Power Digital VLSI Design. Klumer Academic Publishers, Boston, 1998.

[ZG01] Jianwen Zhu and Daniel Gajski. Compiling SpecC for simulation. In Design Au-Tomation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific, 2001., pages 57–62, 2001.