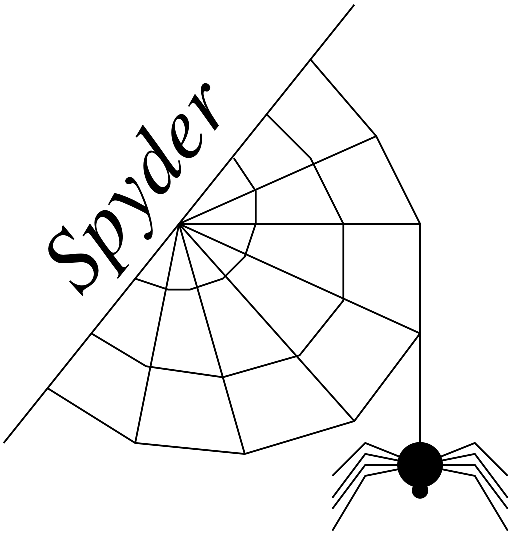

A Reconfigurable Processor Development System

THÈSE N° 1476

Présentée au Département d'informatique
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
pour l'obtention du grade de Docteur ès sciences techniques

par

Christian Iseli
ingénieur informaticien diplômé EPF
originaire de Messen (SO)

acceptée sur proposition du jury:

Prof. E. Sanchez	directeur de thèse
Dr R. Beuchat	corapporteur
Prof. A. A. Jerraya	corapporteur
Dr T. Kean	corapporteur
Dr C. Piguet	corapporteur
Dr R. Taylor	corapporteur

Lausanne, EPFL
1996

Abstract

The Spyder project consists of the development of a reconfigurable processor as well as its application development environment. The name Spyder is an anagram of the first letters of “REconfigurable Processor Development SYstem”, where the term reconfigurable means that the hardware of the processor can be specifically tailored for each application.

Augmenting the performance of a processor implies either increasing its clock frequency or modifying its architecture. In the latter case, the solution usually adopted is to endow the processor with multiple execution units working in parallel (superscalar processors). The main problem with this kind of processors is to locate, in the sequential list of instructions of a program, batches of instructions susceptible of being executed in parallel by the different execution units of the processor.

Thanks to the advent of field-programmable gate array (FPGA) circuits, new kind of superscalar processor architectures can be considered. In particular, the Spyder processor features multiple reconfigurable execution units, which can be redesigned to fit each application, a feature which greatly increases the opportunities to perform parallel computations, particularly when working with small data elements (e.g., 16 Boolean data can be packed into a single 16-bit data word and processed in parallel by a specifically designed operation in the execution unit).

All the resources of the Spyder processor operate in parallel and are controlled by a very large instruction word (VLIW) 128 bits wide. The VLIW architecture allows the use of the full parallelism available to superscalar processors without requiring the complex dispatch unit needed by such processors to handle sequential scalar instructions.

The goal of the Spyder project is to design and implement a superscalar processor with multiple reconfigurable execution units, as well as the software development tools (i.e., C++ compilers and a VLIW assembler) necessary to generate applications for this processor.

A prototype of the Spyder processor has been implemented on a VME board, which has been installed in a VME rack along with a SPARC board acting as host computer. A C++ compiler generating netlist files for the ViewLogic CAD tools has been developed and has generated a lot of interest in the scientific community. It is now available by anonymous ftp on the Internet.

Image processing applications and cellular automata simulations have been programmed on the Spyder processor. The Spyder prototype has demonstrated its ability to achieve a high level of performance with these applications.

Résumé

Le projet Spyder (SYstème de Développement à Processeur REconfigurable) vise à développer un processeur reconfigurable, adaptable à chaque application, ainsi que son environnement de programmation.

Pour augmenter la puissance de calcul d'un processeur sans changer sa fréquence d'horloge, il est nécessaire de modifier son architecture. L'approche généralement choisie est l'exécution de plusieurs opérations en parallèle. Cette approche superscalaire présente certains problèmes particuliers telle que la détection, dans un programme donné, des opérations susceptibles d'être effectuées en parallèle par les différentes unités de traitement du processeur.

L'apparition récente d'un nouveau type de circuit programmable à haute complexité, le FPGA (Field-Programmable Gate Array), permet d'envisager une nouvelle solution: la conception d'un processeur superscalaire avec des unités de traitement reconfigurables, taillées sur mesure pour chaque application. Cette capacité de reconfiguration des unités de traitement permet également d'augmenter le parallélisme lorsque le processeur travaille avec des données de petite taille (par exemple des valeurs booléennes) qui sont compactées sur des mots de données de taille standard (16 valeurs booléennes par mot de 16 bits, par exemple).

Les différentes unités de traitement sont commandées en parallèle, directement par les bits de l'instruction: on parle alors d'une architecture de type VLIW (Very Large Instruction Word). Aussi, on peut assurer une occupation maximale des unités de traitement sans avoir à lire plusieurs instructions consécutives et à les réordonner (tâche effectuée par une unité spécialisé complexe dans les processeurs superscalaires standards).

Le but du projet Spyder est d'une part la conception et réalisation d'un processeur superscalaire avec plusieurs unités de traitement reconfigurables travaillant en parallèle, et d'autre part le développement de compilateurs spécialisés traduisant un programme écrit dans un langage de haut niveau, en l'occurrence C++, en code objet et fichiers de configuration pour les unités de traitement.

Le processeur Spyder a été réalisé sur une carte VME implantée dans une station de travail qui sert de machine hôte. Un compilateur, transformant du code C++ en schémas logiques directement utilisables par les outils de conception assistée par ordinateur de type ViewLogic, a été développé et, vu l'intérêt manifesté, est distribué sur Internet. Des algorithmes de traitement d'images, ainsi qu'un simulateur du jeu de la vie ont été implémentés sur le processeur Spyder avec de très bons résultats quant à la vitesse d'exécution des algorithmes.

Acknowledgments

Preparing a Ph.D. thesis is a long and time consuming process. During this period I have received a lot of help and support from my family and colleagues, and I would like to take this opportunity to express my gratitude towards them.

First, I want to thank my wife Christina and my son Abraham for their love, support, patience, and understanding for the long hours I spent away from them.

I thank my parents, Fritz and Rosemarie, and the other members of my family for their love, support, and for providing me with a good and broad education.

I thank Eduardo Sanchez, my thesis advisor, for his advice, help, trust, friendship, and good humor.

I thank Gianluca Tempesti, my friend, colleague, and English language expert, for the time and effort spent proofreading the manuscript of this thesis.

I thank André Badertscher, my friend, colleague, soldering wizard, and wildlife expert, for his help building the prototype boards of the Spyder processor, and for his enthusiastic descriptions of bobcats, beavers, falcons...

I thank the ACORT team, Peter Brühlmeier and Georges Vaucher, for their help in the design of the prototype Spyder boards.

I thank Daniel Mange, our lab director, Marlyse Taric, our dedicated and charming secretary, and all my other colleagues at the Logic Systems Laboratory for their friendship, help, and support.

I thank René Beuchat, Ahmed Jerraya, Tom Kean, Christian Piguet, and Richard Taylor for the time and effort spent as experts for this Ph.D. thesis.

I thank Nelly and Jacques Lesne for their love, for helping me balance physical and mental exercising, and for teaching me what "martial arts" really means. Tai Chi helped me relax during these stressful times.

Remerciements

Telle l'ascension d'une montagne, la préparation d'une thèse de doctorat est un travail de longue haleine, avec sa marche d'approche, ses haltes pour reprendre des forces, et finalement l'escalade des dernières pentes abruptes qui mènent au sommet. Maintenant que j'ai gravi les derniers mètres, je veux prendre le temps de remercier tous ceux qui, de près ou de loin, m'ont aidé, encouragé et soutenu tout au long de ce parcours.

Tout d'abord, je remercie ma femme Christina et mon fils Abraham pour l'amour et le soutien qu'ils m'ont témoigné, ainsi que la patience et la compréhension dont ils ont fait preuve durant mes longues absences de la maison familiale.

Je remercie mes parents, Fritz et Rosemarie, ainsi que tous les membres de ma famille pour leur amour, leur soutien, et pour m'avoir offert la possibilité de recevoir une éducation solide et ouverte d'esprit.

Je remercie mon directeur de thèse, Eduardo Sanchez, pour son aide, ses conseils éclairés, sa confiance, son amitié et sa bonne humeur permanente et contagieuse.

Je remercie mon ami, collègue et expert de langue anglaise, Gianluca Tempesti, pour tout le temps qu'il a passé à relire et corriger le manuscrit de cette thèse.

Je remercie mon ami, collègue, artiste du fer à souder et expert de nos contrées sauvages, André Badertscher, pour son aide dans la construction des cartes prototypes du processeur Spyder, ainsi que pour ses discussions enthousiastes et ses superbes photos de lynx, castors, faucons et autres hérons de nos régions.

Je remercie l'équipe de l'ACORT, Peter Brühlmeier et Georges Vaucher, pour leur aide dans la conception des cartes prototypes du processeur Spyder.

Je remercie Daniel Mange, le directeur du Laboratoire de Systèmes Logiques, Marlyse Taric, notre dévouée et charmante secrétaire, ainsi que tous mes autres collègues pour leur aide, leur soutien et leur amitié.

Je remercie René Beuchat, Ahmed Jerraya, Tom Kean, Christian Piguet et Richard Taylor qui ont très aimablement accepté la charge d'expert lors de mon examen de doctorat.

Je remercie Nelly et Jacques Lesne pour leur affection, leur aide dans la recherche d'un équilibre entre le travail corporel et mental, ainsi que pour leur enseignement des valeurs authentiques des arts martiaux. Le Tai Chi m'a aidé à me détendre durant les périodes les plus éprouvantes.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Spyder project started as the dream of designing a machine capable of adapting its hardware to any given problem, in order to solve it more efficiently than a conventional machine.

Until recently, transformable hardware was a notion usually confined to the realms of science-fiction. Now, however, a new kind of programmable logic circuits, called Field Programmable Gate Arrays (FPGA), has made this notion into a reality.

This thesis describes the conception of such a transformable machine and its accompanying software development environment. The project name "Spyder" is an anagram of the first letters of "REconfigurable Processor Development SYstem".

This report is divided into 12 chapters and 5 appendices, the first chapter being the present introduction.

Chapters 2 and 3 are provided for the sake of completeness of this report, and do not represent work done as part of this thesis.

Chapter 2 presents an overview of field programmable gate array circuits. These circuits make up the heart of the Spyder processor and are the means used to achieve transformable hardware.

Chapter 3 gives some essential characteristics of the different kind of microprocessors commonly found in computers, and describes how these microprocessors relate to each other in terms of computing power.

Chapter 4 presents a global view of the Spyder project. It describes the main characteristics of the Spyder processor, how this processor is coupled with a host computer, and the means used to develop an application using the Spyder processor.

Chapter 5 describes the architecture and implementation of the Spyder processor in detail. It also describes the low-level communication signals between the host computer and the Spyder processor, as well as the VME interface.

Chapter 6 provides a step-by-step example showing how a Spyder application can be programmed and executed. It is probably necessary to read through the description of the Spyder processor architecture and of its operation in chapter 5 before reading this chapter.

Chapter 7 describes the hardware synthesizer tool. This tool, named `nlc` for netlist compiler, is used to transform software functions written in C^{++} into hardware operators. These hardware operators are then implemented by the configurable hardware of the Spyder processor.

Chapter 8 describes the VLIW compiler that generates the object code of the Spyder programs. The object code uses the hardware operators synthesized by the `nlc` compiler.

Chapter 9 describes the VLIW assembler that generates the object code of the Spyder programs.

Chapter 10 describes the UNIX interface between the host computer and the Spyder processor.

Chapter 11 presents four applications that have been implemented on the Spyder processor: two cellular automata simulations and two image processing programs. The performance of the Spyder processor on these applications is compared with the performance of a conventional computer.

Chapter 12 concludes this thesis report.

The five appendices contain the schematics of the Spyder processor. The source listings of the tools developed for the Spyder project have not been included with this report because of their size. They represent around 30000 lines of C^{++} source code and would require approximately 300 pages to print.

Chapter 2

An Introduction to FPGA Circuits

Field-programmable gate array (FPGA) circuits [Jen94, Tri94, BFRV92] belong to the same broad class of programmable logic devices (PLD) as the PAL, PLA, GAL, and CPLD circuit families [Bur90, Lal90]. The names and acronyms used to designate the different types of circuits in the literature can be quite confusing and sometimes inaccurate.

PAL, PLA, and GAL circuits consist of a matrix of and gates, which computes product terms from the input signals, followed by a matrix of or gates, which sums selected product terms to generate the output signals of the circuit. There is no signal routing in these devices.

CPLD circuits consist of a matrix of logic cells, implementing a configurable logic function, and an interconnection network which allows the routing of signals from the input pins, through the logic cells, to the output pins. However, the interconnection network is highly constrained: only a predefined number of connections can be made and there exists at most one possibility of connection between any two points in the circuit.

FPGA circuits are very similar to CPLD circuits, also consisting of configurable logic cells and an interconnection network. The main difference lies in the interconnection network which, in the case of FPGA circuits, is composed of several kinds of routing resources and allows great flexibility in the selection of a path between any two points in the circuit.

2.1 General Aspects of FPGA Architectures

All FPGA circuits consist of three types of configurable components: logic cells, input/output cells, and interconnection resources. The logic cells are used to implement combinatorial and sequential logic functions, the input/output cells to provide the interfacing between the chip and the outside world, and the interconnection resources to route signals throughout the circuit.

FPGA circuits can be divided into two broad categories, namely coarse-grained and fine-grained, according to the complexity of their logic cells. The logic cells of fine-grained FPGA circuits consist of either a few functionally complete logic gates (e.g., nand gates), or a low-complexity universal function (e.g., a multiplexer with 2 or 3 command variables). The logic cells of coarse-grained FPGA circuits, on the other hand, usually implement the universal function of several input variables (e.g., the universal function of 4 variables, implemented with a look-up table), as well as one or two flip-flops. Coarse-grained logic cells tend to waste silicon surface, especially when used to implement simple logic gates, but require fewer signals to be routed through the cells. Fine-grained logic cells, conversely, avoid the waste of silicon area at the cost of more signals to route.

FPGA circuits can also be divided into two other categories, RAM-based and antifuse-based, according to the technology used to configure the circuits. Each programmable resource in the circuit is controlled either by a static RAM cell in the case of RAM-based FPGA, or by antifuses in the case of antifuse-based FPGA. An antifuse is an electric device, connecting two points, that irreversibly changes from a high to a low resistance when a given programming voltage is applied across its terminals. The main advantage of RAM-based FPGA circuits is that they can be reprogrammed, while the disadvantages are that a static RAM cell is much larger than an antifuse and must be configured upon power-up. The advantages of antifuses are a very small size and short signal propagation delays, the drawbacks being the irreversibility of their programming and the substructure required to apply the programming voltage.

2.2 Application Design Process

The description of the function to be implemented by an FPGA circuit can be specified using a hardware description language (usually VHDL), a set of equations (e.g., Abel), or schematics. This input description is then processed by CAD tools to produce a hardware netlist representing the function to be implemented by the FPGA circuit. Each family of FPGA circuits has a specific hardware components library to be used in the netlist description.

The netlist is then processed by software tools, again specific to each family of FPGA circuits, to produce the configuration file required to program the FPGA circuit. The software tools perform three tasks: splitting the hardware components used in the netlist into parts that fit in one logic cell of the FPGA circuit, placing the parts in the available logic cells on the circuit, and routing the signals between cells.

2.3 Xilinx XC4000 FPGA Circuits

Xilinx was the first manufacturer of FPGA circuits, starting in 1984, and is currently the leader of the market, offering several families of FPGA circuits [Xil94]. Only the XC4000 family will be presented here.

The XC4000 family of FPGA circuits has a coarse-grained RAM-based architecture, where the logic cells are organized in a rectangular array, surrounded by the input/output cells. The routing network occupies the space between the cells.

Each logic cell (shown in figure 2.1) can implement two universal functions of 4 variables, one universal function of 5 variables, or, for some particular types of functions, one function of up to 9 variables. The functions are implemented by three look-up tables. Rather than implementing a logic function, two of the look-up tables can also be used

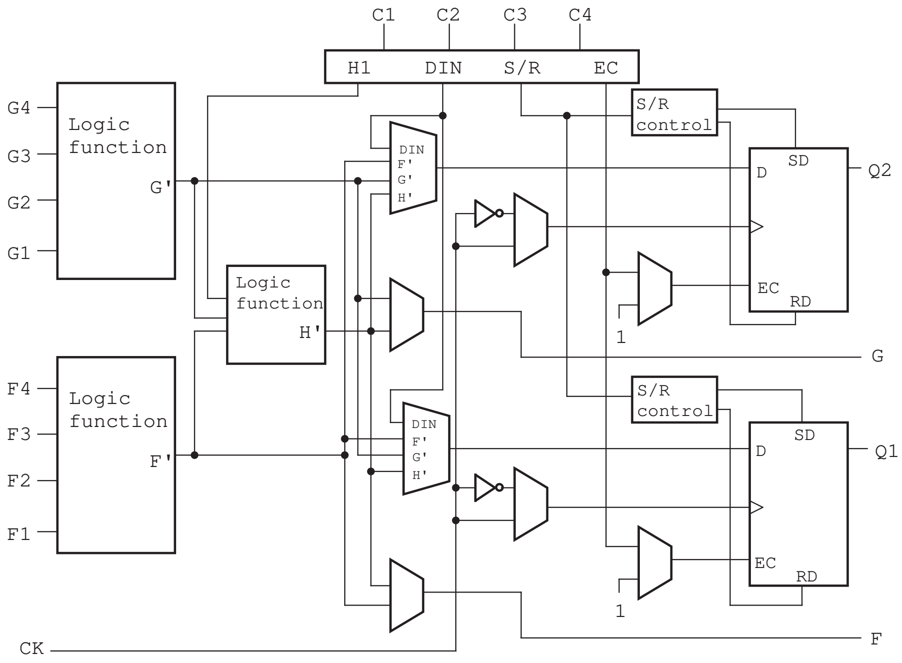

Figure 2.1: XC4000 logic cell architecture

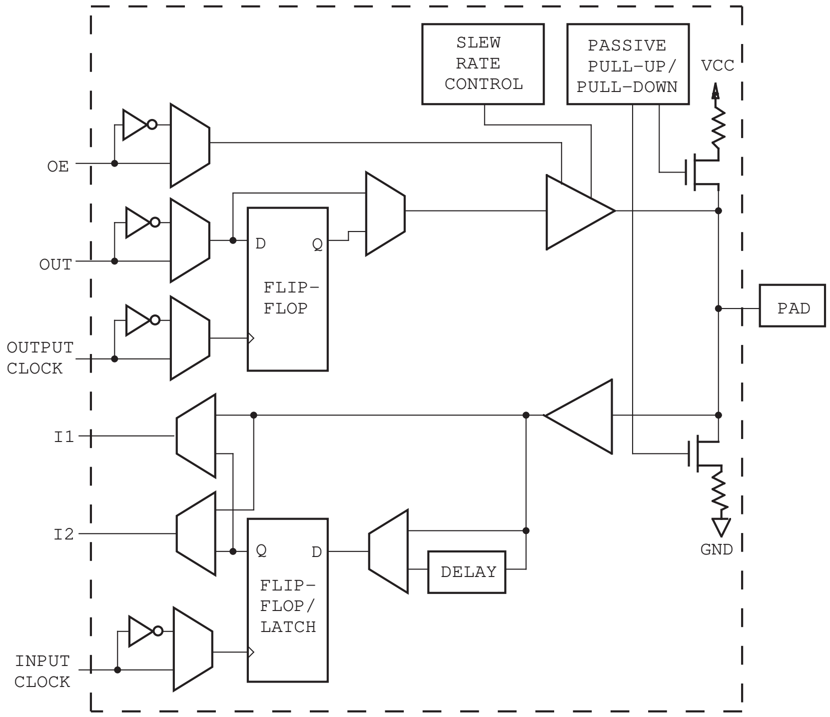

Figure 2.2: XC4000 input/output cell architecture

as static RAM elements. Each logic cell also contains two flip-flops with clock-enable, set, and reset signals.

Each pin of the circuit is connected to an input/output cell, shown in figure 2.2,

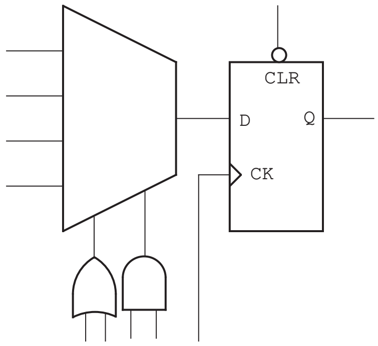

Figure 2.3: ACT2 logic cell architecture

containing one output flip-flop, one input flip-flop (also usable as a latch) a tri-state buffer, a pull-up resistor, and a pull-down resistor.

The routing network offers several type of connections: direct connections from one logic cell to its neighbors, short lines spanning one or a few logic cells and linked by switch matrices, and long lines spanning the entire circuit horizontally or vertically. There are also a few global nets for the distribution of clock and reset signals for the flip-flops.

There are two major programming modes for the XC4000 circuits: master or slave. These modes are further subdivided into parallel and serial submodes. In master mode the XC4000 circuit controls the loading of its own configuration, while in slave mode the control is performed by some other device (e.g., a computer). In parallel mode 8 bits of configuration data are loaded by the circuit in parallel (but the data is serialized before being used to configure the circuit). The entire FPGA circuit must be configured in a single loading operation (i.e., it is not possible to configure only a part of the circuit), a process which takes a few milliseconds.

2.4 Actel ACT2 FPGA Circuits

The ACT2 FPGA circuits are manufactured by Actel [Act94]. The ACT2 family of FPGA circuits has a fine-grained antifuse-based architecture. The logic cells are organized in rows separated by routing nets. The input/output cells are located on the edge of the circuit. The rows of routing nets are joined by vertical lines.

The logic cell of the ACT2 family, shown in figure 2.3, is basically a multiplexer with 2 command variables. Some, but not all, logic cells contain a flip-flop.

The input/output cells are extremely simple, containing a tri-state buffer but no storage element.

There are two types of routing resources: horizontal and vertical lines. The horizontal lines are located between the rows of cells and can be connected to the inputs of the cells (see figure 2.4) and to the vertical lines. The vertical lines are used to join horizontal lines between rows. The output of the logic cells is connected to a vertical line. The horizontal and vertical lines can be of various lengths, some vertical lines span the whole height of the circuit, in which case they are called long vertical tracks. Actel guarantees that a connection between any two points in the circuit crosses at most four

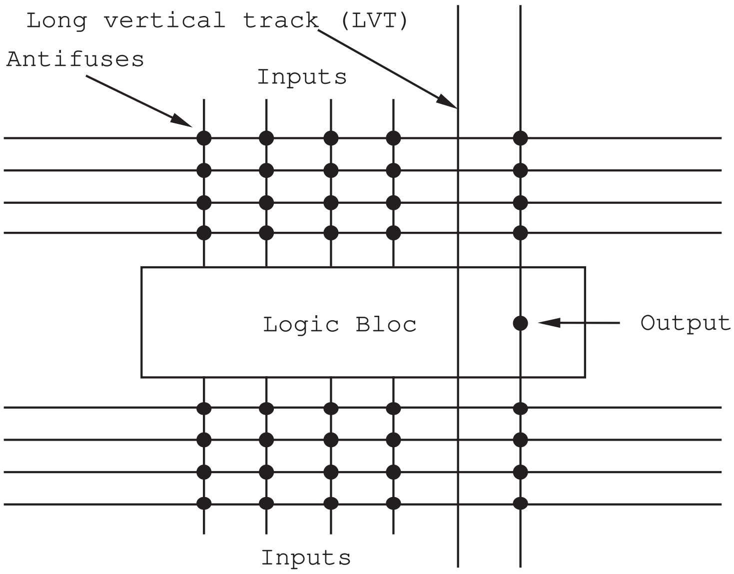

Figure 2.4: ACT2 logic cell inputs and output

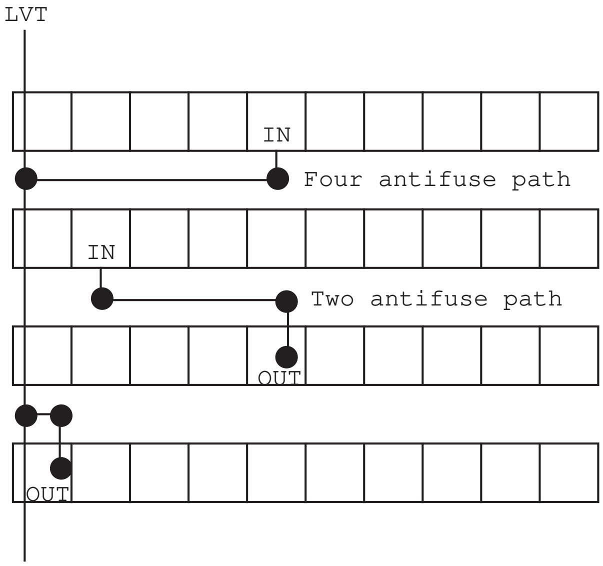

Figure 2.5: ACT2 signal routing examples

antifuses, as shown in figure 2.5.

Programming an ACT2 circuit requires the use of a special device, driven by a workstation. Once programmed, it is impossible to make any modification to the FPGA circuit.

Chapter 3

Processors and Performance

The purpose of a computer application can be roughly described as the generation of a set of output data O by applying some function, usually called a program, Ψ to a set of input data I, as summarized by equation 3.1.

$$O = \Psi(I) \tag{3.1}$$

The generation of the output data requires a certain amount of time T. The performance of a processor is determined by the time $T_{\Psi(I)}$ required to execute a program Ψ on a given set of input data I.

3.1 General-Purpose Processors

A general-purpose processor must be able to run any program Ψ, however complex. This is achieved by splitting Ψ in a series of basic instructions $(\psi_1, \psi_2, \ldots \psi_n)$, where each instruction ψ_i belongs to the instruction set of the processor, i.e., the processor is capable of executing it directly. Thus, the time taken by a general-purpose processor to run the program can be computed as in equation 3.2, where n is the number of instructions and t_{ψ_i} is the time taken by instruction ψ_i to execute.

$$T_{\Psi(I)} = \sum_{i=1}^{n} t_{\psi_i} \tag{3.2}$$

There are three ways to increase the performance of a processor, of which two are readily apparent from equation 3.2: decreasing the number of instructions required to execute a program and decreasing the time required to execute an instruction. The third way is to execute more than one instruction at the same time, in parallel. Unfortunately these parameters are not independent. Decreasing the number of instructions in the program usually means that the instructions will be more complex and require more time to execute. Finally, trying to use parallelism, like multiple-issue processors do, presupposes the ability to find independent instructions susceptible of concurrent execution, which requires more hardware resources and complicates the control of the processor.

The silicon technology used to manufacture the processors plays a great role in the raw execution speed of the instructions. Also, the architecture of the processor has a huge number of tunable parameters and whole books have been, and are still being, written on the subject [HP96, Fly95, Joh91, RF93].

3.2 CISC and RISC Processors

General-purpose processors can be divided into two broad categories: CISC and RISC, where CISC stands for Complex Instruction Set Computer and RISC stands for Reduced Instruction Set Computer.

Generally speaking, CISC processors try to optimize n in equation 3.2, while RISC processors favor very simple instructions which require less time to execute (but tends to increase the number of instructions needed to execute a program). In the former case, the time t_{ψ_i} can vary greatly from one instruction to the next, whereas in the latter case, all instructions require the same amount of time[1] and equation 3.2 can be rewritten as equation 3.3, where t_ψ is a constant.

$$T_{\Psi(I)} = nt_\psi \tag{3.3}$$

The choice of RISC designers to have simple, fast instructions has several consequences, the major ones being single instruction format, load-store architecture, and easier pipelining. The single instruction format means that all the instructions are coded using the same number of bits. Almost all known commercial RISC processors use 32 bits. Single format is desirable because it allows the processor to know what amount of memory to read to obtain a given number of instructions and because it simplifies the decoding of the instructions.

Another major difference between CISC and RISC processors is the way the instructions access data in memory. CISC instructions are able to operate directly on main memory data, which complicates their decoding, rendering the single instruction format very impractical, and the time required to execute these instructions further depends on how rapidly the main memory is able to supply the operands to the processor. RISC processors can only access the main memory through load and store operations (which move data in and out of their internal registers) and their instructions can only operate on their internal registers. Thus the load and store instructions are the only ones that can cause delays if the main memory cannot supply the data fast enough.

3.3 Pipelining

One way to obtain some gain from parallel execution without having to find independent instructions susceptible of executing concurrently is to use pipelining. The principle of pipelining is to split each basic instruction ψ_i into several sub-parts and to execute sub-parts of several consecutive instructions concurrently. Any instruction can be decomposed, in an intuitive and natural way, into a sequence of phases. These phases

[1] This is not exactly true of superscalar RISC processors, i.e., RISC processors which have several execution units (integer, floating-point, etc.) executing in parallel. In that case, each execution unit can have its own pipeline, the number of stages of each pipeline can be different, and thus different kind of instructions can require different amount of time to complete.

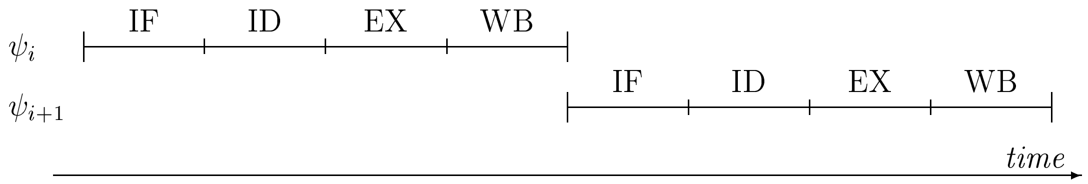

Figure 3.1: Sequential execution

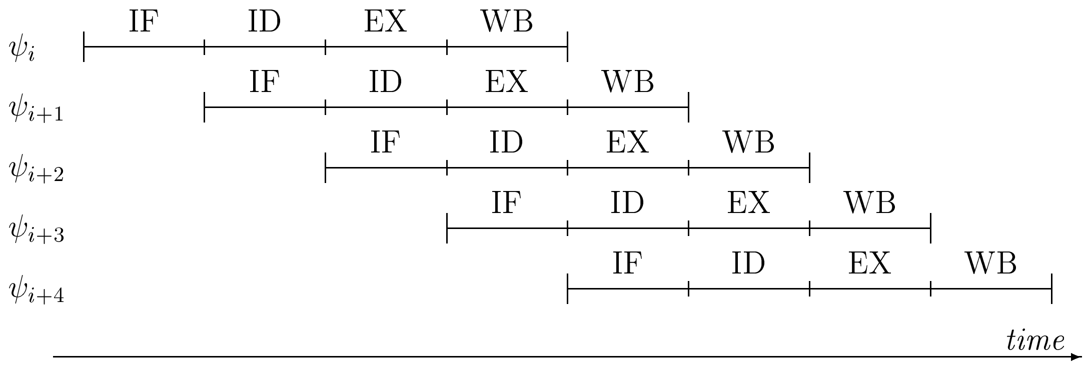

Figure 3.2: Pipelined execution

usually consist of an Instruction Fetch (IF) phase where the next instruction is fetched from memory into the processor, and Instruction Decode (ID) phase where the meaning of the instruction and the location of the operands are decoded, an Execute (EX) phase where the actual computing takes place, and a Write Back (WB) phase where the results of the computation are saved in the destination location.

Thus, if we admit the decomposition given above, to execute instruction ψ_i the processor can perform the IF phase, then the ID phase, then the EX phase, and finally the WB phase in sequence, as shown in figure 3.1.

However, each of these different phases can be executed by independent hardware resources, and the intermediate results saved (memorized) in registers. Thus, as soon as the IF phase of instruction ψ_i is complete it is possible to start the IF phase of instruction ψ_{i+1} while the instruction ψ_i proceeds in the ID phase, etc. This pipelined mode of execution is shown in figure 3.2. RISC processors are better suited to use pipelining since all their instructions take the same time to execute and the accesses to the main memory are restricted to the load and store operations, whereas the variable execution time of CISC instructions causes serious problems. Each instruction still requires time T_ψ to execute, but once the pipeline is full l instructions execute in parallel and the apparent execution time of an instruction becomes T_ψ/l, where l is the number of pipeline stages ($l = 4$ in our example above). If we admit that the time taken to fill the pipeline is negligible with respect to the total time taken to run a program, equation 3.3 can be rewritten as equation 3.4.

$$T_{\Psi(I)} = \frac{n}{l} t_\psi \tag{3.4}$$

Generally speaking, the different phases of execution of the pipeline are timed by a clock signal of period λ and l is chosen such that $l\lambda = T_\psi$. We can thus further rewrite equation 3.4 as equation 3.5.

$$T_{\Psi(I)} = n\lambda \tag{3.5}$$

Pipelines introduce some hazards [HP96, pages 139–178], which must be dealt with either at compile or execution time, including conditional branches and attempts to read a value that has not yet been written because the instruction that writes it is still in the pipeline. Moreover, splitting an instruction into phases adds some time and hardware overhead to save the intermediate results between the different phases.

3.4 Multiple-Issue Processors

Multiple-issue processors contain additional hardware to execute several instructions in parallel. They are split into two categories: superscalar and Very Large Instruction Word (VLIW) processors [Fly95, pages 456–498].

Superscalar processor use a dynamic scheduling method. They read several instructions of the program at the same time and determine which ones can be executed in parallel. This allows them to execute programs compiled for a single-issue processor of the same family, but requires a considerable amount of instruction scheduling hardware.

VLIW processors require the compiler to statically assemble operations to be executed in parallel in a single, large instruction. They are thus unable to reuse code compiled for a single-issue processor, but their hardware control is much simpler.

Good compilers for both categories tend to be equally difficult to write, since in both cases the compiler will want to rearrange operations in a way that will maximize their parallel executions.

The architecture of multiple-issue processors is basically similar to the architecture of RISC processors, but they have several execution units. For example, the PowerPC 620 processor has 6 execution units: two simple integer units, one complex integer unit handling multiply and divide operations, one load-store unit, one floating point unit, and one branch unit [HP96, pages 335–349].

Multiple-issue processors also use pipelining techniques to augment their performance. Thus, equation 3.5 can be rewritten as equation 3.6, where N is the average number of operations executed in parallel when program Ψ is applied to the set of input data I, and $1 \leq N \leq U$, where U is the number of available execution units.

$$T_{\Psi(I)} = \frac{n}{N}\lambda \tag{3.6}$$

3.5 Special-Purpose Processors

Special purpose processors try to minimize the time $T_{\Psi(I)}$ taken to run a program Ψ by providing special instructions tailored specifically for that particular task. In other words, they try to minimize n in equation 3.2 by providing custom-tailored instructions so that program Ψ is split into fewer instructions. The penalty for this approach is more hardware and greater development costs for the dedicated processors.

The best-known examples of special-purpose processors are floating-point coprocessors and graphics coprocessors. With the exception of universities, where they are built for research purposes, special-purpose processors on the market are rare, mainly because of their high development costs and their limited market share.

Chapter 4

Spyder Basics

The basic goal of the Spyder project was to develop a reconfigurable coprocessor with its accompanying development environment. The word reconfigurable is used by many people for different purposes. In the case of Spyder, reconfigurable means that the hardware of the processor is tailored specifically for each application.

The annual IEEE symposium on FPGAs for Custom Computing Machines in Napa, California, sees each year a growing number of proposals of configurable processors. Among the best-known are the Splash project [ABD92], and the DEC-Perle project [BRV89]. The Spyder project has some very distinctive features: the reconfigurable hardware is structured, in the form a VLIW processor, and a lot of efforts have been put in the construction of an application development environment in which a software developer can feel comfortable without being a hardware expert.

The development environment of Spyder should provide the same user interface as a typical workstation software development environment. In other words, given the source code of a typical application that can be compiled and run on a workstation, the ultimate goal of the Spyder development tools is to take this same source code and compile it, producing the configuration of the Spyder hardware and using the configured hardware to run the application.

The most difficult task is to split the code into parts: the part that becomes the hardware configuration, the part that directly controls this hardware, and the part that runs on the host workstation and uses the Spyder coprocessor. This task is actually left to the programmer. The Spyder development tools do, however, provide a unified development environment which uses the C++ language to describe all the parts.

4.1 The Spyder Processor

At this time, the only hardware devices which can provide both reconfigurability and sufficient computing power are RAM-based FPGA chips, presented in chapter 2.

To keep the Spyder coprocessor and the host workstation fairly independent, the former was endowed with its own program and data memories, so that there is no bus contention and both Spyder and the workstation can work in parallel without hindering each other.

4.1.1 Multiple Execution Units

When the Spyder project started, the choice of available FPGA chips was limited, as was the number of pins per chip. To provide enough computing power, it was decided that more than one chip was needed to perform the computation, and that they would all run in parallel: the superscalar architecture was born. Given the number of pins available per chip, a data width of 16 bits was chosen.

A common problem with processors having multiple execution units is the high bandwidth required to connect the execution units and the data memory. The solution adopted was to use a dual-port data memory and two four-port register banks, the latter being the memory chips with the highest number of independent ports available on the market; since one of the four ports is needed to connect the register bank with the data memory, there remain three usable ports and thus three execution units.

4.1.2 Register Windowing

The execution units read their input data and write their computed results exclusively into the register banks; in some sense the register banks act as a cache memory between the execution units and the main data memory. The rule of thumb says that the bigger the cache, the better. However, such a large monolithic block of registers is not very convenient: better to split it into smaller, independent chunks which can be accessed through an indexing mechanism. This access method is usually called register windowing.

The windowing mechanism defines a current window pointer, which is used to reference the current registers window, the previous registers window and the next registers window; a fourth registers window, called the global window, is also defined and is independent of the current window pointer, i.e., the global window stays the same at all times. The Spyder registers are always accessed through these four windows.

The current window pointer can be manipulated by three commands: increment, decrement and reset. The standard mode of operation is to increment the pointer on subroutine calls and to decrement it when the subroutines return, like in the SPARC processor [WG94], but on Spyder the three operations are under complete control of the programmer.

4.1.3 Programming

All the resources of Spyder processor operate in parallel and are controlled by a very large instruction word (VLIW) 128 bits wide. This VLIW code very closely resembles microcode, since it directly commands the Spyder resources without further interpretation or decoding. However, it is not microcode, as it is not interpreting the operations of a higher level assembly language: it is the assembly language itself. The control program is stored in its own memory, which means that the Spyder processor has a Harvard type architecture. The execution control is performed by a sequencer. The sequencer can perform conditional jumps based on status information computed by the execution units; it contains loop counters and a return address stack for subroutine calls.

The Spyder processor uses a pipelined execution mode to achieve higher processing speed. The pipeline is composed of four stages: instruction fetch, address computation, execution, and write back.

4.1.4 Host Workstation Coupling

From the host workstation point of view, Spyder appears as a loosely coupled coprocessor connected on the VME bus. There are four communication channels: one is used to download the execution units configurations, another to download the control program in the program memory, a third is used to access the data memory, and the fourth is the control channel.

A typical application run on a workstation with an attached Spyder board consists of the following steps: first, the workstation configures the Spyder processor by downloading the configuration of the execution units and the control program; then the workstation stores the data to be processed into the Spyder data memory and starts its execution; when Spyder has completed its task, the workstation reads back the results it needs, stores new data to be processed, restarts Spyder, and the cycle is repeated.

4.2 The Development Environment

The development of an application using the Spyder coprocessor basically consists of writing three separate pieces of source code: one to run on the host workstation, one to configure the execution units, and one for the control program. All three pieces can be written in C++, making it possible to compile them together with a standard C++ compiler to obtain a working application, running entirely on the host workstation, that can be debugged with standard debugging tools. When the application runs as expected, the pieces of code aimed at Spyder can be processed with the Spyder development tools and the application can be run using the Spyder coprocessor.

4.2.1 Host Workstation Program

The program running on the host workstation is responsible for loading the data to be processed by Spyder into the Spyder data memory, starting the coprocessor, waiting for it to complete its computations, and getting back the results. The program must also handle the user interface, and the interface to the operating system in general.

A C++ interface class for the Spyder coprocessor is provided by the development system. This interface class hides all the low-level details of communication with the coprocessor. It is thus easier to develop this program in C++, although it is quite possible to use another language.

4.2.2 Execution Units Configuration

The execution units configuration must describe all the operations that must be implemented by the execution units hardware. This description can be done either in C++ (the preferred solution) or with ViewLogic schematic tools.

The C++ description consists of providing the functional description of the required operations by writing C++ functions that implement these operations. Each of these functions will then be transformed by the hardware synthesizer into a hardware component that can be configured in the execution unit.

The main advantages of using C++ for the description of the hardware operators are simplicity and being able to simulate (or rather to emulate) the generated hardware operator as part of a larger C++ program.

4.2.3 VLIW Program

The VLIW program is charged with feeding data to the execution units, moving it from the main memory through the register banks, and storing back the results of the computation. Two tools are available to help the programmer develop this program: a C++ compiler and an assembler.

The C++ compiler performs several complex tasks: assigning the hardware operators to the execution units, parallelizing the code to use the three execution units and the data transfer operators in parallel, managing the data in the two large register banks, detecting and avoiding pipeline hazards.

The assembler allows complete control of the Spyder processor resources, at the expense of more work for the programmer, who is fully responsible for performing the assignment of the hardware operators to the execution units, controlling the parallel use of the available resources and handling the pipeline hazards.

Chapter 5

Spyder Architecture

The Spyder coprocessor is composed of three main parts: a bus interface, a control part and an execution part. Its architecture is presented in figure 5.1. It is based on standard microcode [Hab88] and VLIW [RF93] processor architectures, with some very distinctive characteristics: reconfigurable execution units and multiple shared register banks.

From its superscalar and VLIW architecture heritage, Spyder draws several execution units (currently three) working in parallel. These execution units have no predefined functionality and must be configured for each application. The instructions and the data reside in two separate memories (Harvard architecture), partly to alleviate constraints in the execution pipeline, but mostly because the widths of instruction and data word do not match: a data word is 16-bit wide whereas an instruction word is 128-bit wide. These word widths are implementation trade-offs, and future implementations of Spyder could have a 32-bit data word or additional execution units, in which case the instruction word would probably have to be wider.

Like RISC processors, the execution units of Spyder operate only on data in the registers. The data memory and the registers exchange data exclusively through load and store operations [FR94].

There are two register banks. Each bank is shared among all the execution units. The register banks have four ports that can be accessed simultaneously. Three of the ports are connected to the three execution units and the last one is connected to the data memory. The data memory is a 64K $\times$ 16 bits dual-port SRAM, allowing two load/store operations in parallel, one for each register bank.

5.1 Principle of Operation

The pipelining technique [PH94] has been used to increase the overall throughput of the Spyder processor. One basic instruction is executed in 4 phases. One instruction is issued at each clock cycle and requires 4 clock cycles to complete, as shown in figure 5.2.

During the first phase, the sequencer puts the address of the next instruction (PC) on the address bus of the program memory.

In the second phase, the program memory loads the instruction into its pipeline register and puts it on the command lines of the processor. The memory controller

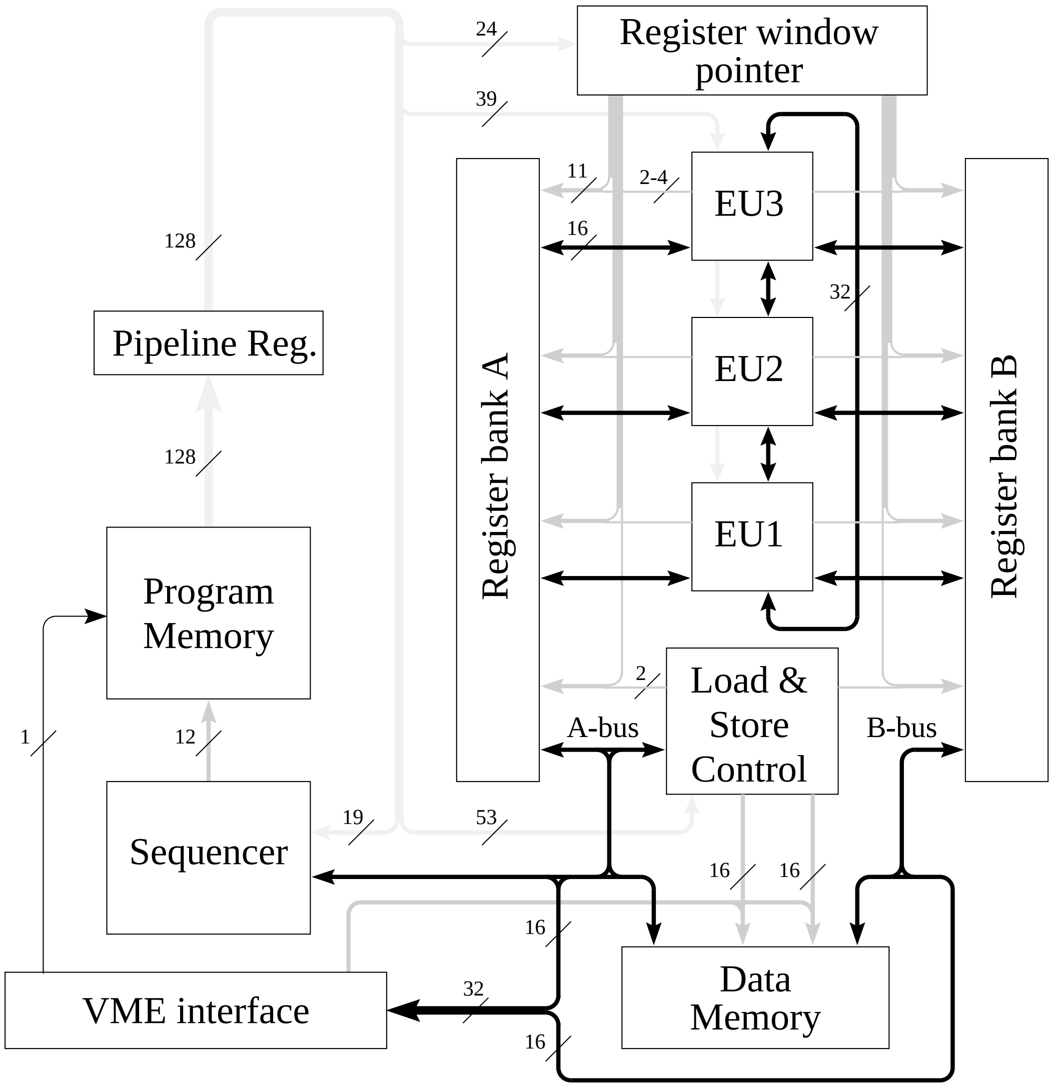

Figure 5.1: Spyder architecture

computes the data memory addresses used by the instruction. The register windowing controller and the execution units compute the address of the registers used by the instruction.

During the third phase, the data memory addresses computed in the previous phase are put onto the data memory address busses and the data transfers between the memory and the register banks are performed. The register addresses computed during the second phase are put onto the address busses and the execution units read the data coming from the registers halfway through this cycle (on the high to low transition of the clock). The execution units then start to execute the requested operation on the data they just read.

Halfway through the fourth phase (one complete clock cycle after the execution units started executing their operation), the address of the registers in which the result of the operation will be stored are put on the address busses and the results of the operations are saved.

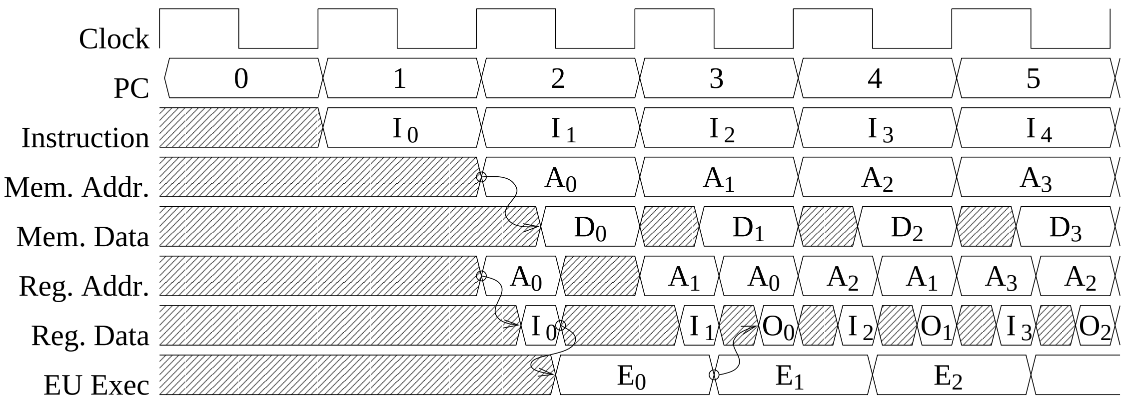

Figure 5.2: Instruction execution timing

		Write						Register Address								Window Select			
127	126–106	105	104	103	102	101	100	99–98	97–96	95–94	93–92	91–90	89–88	87–84	83–80	79–78	77–76	75–74	73–72
Stop	EU Commands	Register 3 B	Register 3 A	Register 2 B	Register 2 A	Register 1 B	Register 1 A	Register 3 B	Register 3 A	Register 2 B	Register 2 A	Register 1 B	Register 1 A	Register 0 B	Register 0 A	Register 3 B	Register 3 A	Register 2 B	Register 2 A

Window Select				Memory Bus B					Memory Bus A						Sequencer	
71–70	69–68	67–66	65–64	63–48	47	46–44	43	42–41	40–25	24–23	22–20	19	18	17–16	15–12	11–0
Register 1 B	Register 1 A	Register 0 B	Register 0 A	Displacement	Immediate	Base Register	Save Address	Direction	Displacement	Control	Base Register	Save Address	Register Select	Direction	Command	Jump Address

Figure 5.3: Spyder instruction word

Note that while data are moved to and from the data memory in the third phase, they are moved from the execution units back to the registers one clock cycle later in the fourth phase.

5.2 Structure of an Instruction

To take advantage of the parallelism provided by the multiple execution units, all the hardware components need to be controlled individually and simultaneously. This is the basic principle of a very large instruction word (VLIW) architecture. Each instruction commands all the resources of the processor in parallel, without any coding, in a way similar to horizontal microcoding [Hab88]: there is thus no decoding of the instruction and the parallelism is not constrained in any way. The format of an instruction is shown in figure 5.3.

Table 5.1: Sequencer and window pointer operations

Command				Window pointer	Counter	Action
I_{15}	I_{14}	I_{13}	I_{12}			
0	0	0	0	—	—	Jump
0	0	0	1	—	decrement	Repeat loop if counter $\neq 0$
0	0	1	0	—	—	Jump if flag $\equiv 0$
0	0	1	1	increment	decrement	Repeat loop if counter $\neq 0$
0	1	0	0	increment	—	Call
0	1	0	1	increment	—	Jump
0	1	1	0	cond. increment	—	Call if flag $\equiv 0$
0	1	1	1	increment	—	Jump if flag $\equiv 0$
1	0	0	0	—	pop	Jump
1	0	0	1	reset	—	Jump
1	0	1	0	—	—	Jump if flag $\neq 0$
1	0	1	1	decrement	decrement	Repeat loop if counter $\neq 0$
1	1	0	0	decrement	—	Return
1	1	0	1	decrement	—	Jump
1	1	1	0	cond. decrement	—	Return if flag $\equiv 0$
1	1	1	1	decrement	—	Jump if flag $\equiv 0$

The 128 bits of an instruction directly control 6 different operations, as described in the following sections.

5.2.1 Sequencer

The program sequencer is controlled by the instruction bits $I_{15:0}$. There are four categories of operations, controlled by bits $I_{15:12}$: jump, subroutine call, return from subroutine, and loop. All these operations can be executed unconditionally or conditionally using the flag provided by the execution units. The detailed description of the 16 available operations is given in table 5.1. The jump address is given explicitly by the 12 bits $I_{11:0}$. The return address of subroutine calls is saved using a built-in stack, currently providing 64 levels.

The loop mechanism is implemented using a stack of counters. Upon entry in a loop, the program pushes the number of iterations onto the stack[1]. At the end of the loop body, the program decrements the counter and conditionally jumps back to the start of the loop. When the counter reaches zero, it is automatically popped from the stack by the decrement operation. The current implementation of the sequencer provides a counter stack of 32 levels.

[1]This operation is actually an operation performed by the load/store unit (see section 5.2.2) and thus does not appear in table 5.1.

Table 5.2: Spyder memory address computation

$$address = \mathsf{A}_{base} + \mathsf{A}_{displacement} \quad (5.1)$$
$$address = \mathsf{A}_{base} + displacement \quad (5.2)$$

Table 5.3: Summary of the load and store operations

$\mathsf{M}_{address} \leftrightarrow \mathsf{R}^a_n$	(5.3)		$constant \rightarrow \mathsf{A}_n$	(5.8)
$\mathsf{A}_{base} \leftrightarrow \mathsf{R}^a_n$	(5.4)		$\mathsf{M}_{address} \rightarrow Sequencer$	(5.9)
$constant \rightarrow \mathsf{R}^a_n$	(5.5)		$\mathsf{A}_{base} \rightarrow Sequencer$	(5.10)
$\mathsf{M}_{address} \leftrightarrow \mathsf{A}_n$	(5.6)		$constant \rightarrow Sequencer$	(5.11)
$\mathsf{A}_{base} \leftrightarrow \mathsf{A}_n$	(5.7)		$\mathsf{M}_{address} \leftrightarrow \mathsf{R}^b_m$	(5.12)

5.2.2 Load and Store

The load and store unit controls the two busses that link the data memory (M), the register banks, and the sequencer. The data memory has two input/output ports. The first bus, called the A-bus, links the first port of the data memory, the register bank R^a, the sequencer, and the load/store unit itself. The second bus, called the B-bus, links the second port of the data memory and the register bank R^b.

The load and store unit contains 8 address registers (A). There are two addressing modes to access the data memory, as shown by equations 5.1 and 5.2 in table 5.2. These addressing modes are similar to the addressing modes used by the SPARC processor [WG94].

The computed address can be saved in the register A_{base}. The *base* and *displacement* parameters come directly from the instruction. The *displacement* parameter can be either a 16-bit constant or a register tag, selected using the 3 least significant bits.

The A-bus allows bidirectional data transfers between the data memory, the address registers, and the register bank R^a, transfers of a constant value to the register bank R^a, an address register, or the sequencer (to push a new loop counter), and transfers from the data memory or an address register to the sequencer. These operations are summarized by equations 5.3 through 5.11 in table 5.3. They are controlled by the instruction bits $I_{40:16}$ and their encoding is specified in table 5.4, where *base* corresponds to the instruction bits $I_{22:20}$, n to $I_{83:80}$, and *displacement* to $I_{40:25}$. The computed

Table 5.4: A-bus transfer operations

Direction I_{17}	Direction I_{16}	*Select* I_{18}	Control I_{24}	Control I_{23}	Action
0	0	ϕ	ϕ	ϕ	Idle
0	1	0	0	0	$\mathsf{M}_{\mathsf{A}_{base}+\mathsf{A}_{displacement}} \rightarrow \mathsf{R}^a_n$
0	1	0	0	1	$\mathsf{M}_{\mathsf{A}_{base}+displacement} \rightarrow \mathsf{R}^a_n$
0	1	0	1	0	$\mathsf{A}_{base} \rightarrow \mathsf{R}^a_n$
0	1	0	1	1	$displacement \rightarrow \mathsf{R}^a_n$
0	1	1	0	0	$\mathsf{M}_{\mathsf{A}_{base}+\mathsf{A}_{displacement}} \rightarrow \mathsf{A}_n$
0	1	1	0	1	$\mathsf{M}_{\mathsf{A}_{base}+displacement} \rightarrow \mathsf{A}_n$
0	1	1	1	0	$\mathsf{A}_{base} \rightarrow \mathsf{A}_n$
0	1	1	1	1	$displacement \rightarrow \mathsf{A}_n$
1	0	0	0	0	$\mathsf{M}_{\mathsf{A}_{base}+\mathsf{A}_{displacement}} \leftarrow \mathsf{R}^a_n$
1	0	0	0	1	$\mathsf{M}_{\mathsf{A}_{base}+displacement} \leftarrow \mathsf{R}^a_n$
1	0	0	1	0	$\mathsf{A}_{base} \leftarrow \mathsf{R}^a_n$
1	0	0	1	1	Illegal
1	0	1	0	0	$\mathsf{M}_{\mathsf{A}_{base}+\mathsf{A}_{displacement}} \leftarrow \mathsf{A}_n$
1	0	1	0	1	$\mathsf{M}_{\mathsf{A}_{base}+displacement} \leftarrow \mathsf{A}_n$
1	0	1	1	0	$\mathsf{A}_{base} \leftarrow \mathsf{A}_n$
1	0	1	1	1	Illegal
1	1	ϕ	0	0	$\mathsf{M}_{\mathsf{A}_{base}+\mathsf{A}_{displacement}} \rightarrow sequencer$
1	1	ϕ	0	1	$\mathsf{M}_{\mathsf{A}_{base}+displacement} \rightarrow sequencer$
1	1	ϕ	1	0	$\mathsf{A}_{base} \rightarrow sequencer$
1	1	ϕ	1	1	$displacement \rightarrow sequencer$

memory address can be saved in the address register A_{base} by setting bit I_{19}.

The B-bus allows data to be transferred between the data memory and the register bank R^B. This operation is summarized by equation 5.12 in table 5.3. It is controlled by the instruction bits $I_{63:41}$ and its encoding is specified in table 5.5, where *base* corresponds to the instruction bits $I_{46:44}$, m to $I_{87:84}$, and *displacement* to $I_{63:48}$. The computed memory address can be saved in the address register A_{base} by setting bit I_{43}.

It should be noted that the idle operation on both the A-bus and the B-bus can be used to do computations in the address registers, i.e., no data transfer occurs but the computed memory address is saved back in the base register.

5.2.3 Register Window Control

The registers banks R^a and R^b are accessed through a windowing mechanism, shown in figure 5.4. The set of available registers is divided into windows of 4, 8 or 16 registers. Four windows are accessible at any given time: the global window (G), the current window (C), the previous window (P), and the next window (N). The global window is always accessible, while the others are accessed through a current window pointer. The

Table 5.5: B-bus transfer operations

Direction I_{42}	I_{41}	Immediate I_{47}	Action
0	0	ϕ	Idle
0	1	0	$\mathsf{M}_{\mathsf{A}_{base}+\mathsf{A}_{displacement}} \rightarrow \mathsf{R}^b_m$
0	1	1	$\mathsf{M}_{\mathsf{A}_{base}+displacement} \rightarrow \mathsf{R}^b_m$
1	0	0	$\mathsf{M}_{\mathsf{A}_{base}+\mathsf{A}_{displacement}} \leftarrow \mathsf{R}^b_m$
1	0	1	$\mathsf{M}_{\mathsf{A}_{base}+displacement} \leftarrow \mathsf{R}^b_m$
1	1	ϕ	Illegal

Table 5.6: Register window selection

I_{j+1} ($j = 78, 76, 74, 72, 70, 68, 66, 64$)	I_j	Window select
0	0	Current
0	1	Next
1	0	Previous
1	1	Global

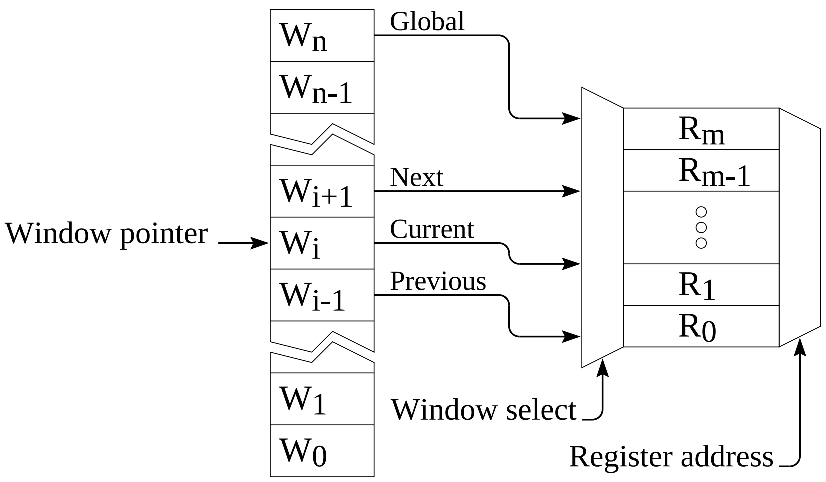

Figure 5.4: The register windows

control mechanism of the window pointer is somewhat similar to that of the SPARC architecture [WG94]: the current window index is incremented at each procedure call and decremented at each return, but can also be manipulated directly by the program. By convention, the global window has the highest index.

The instruction bits $I_{79:64}$ are used to select the register windows used by the operations dealing with data registers, described in sections 5.2.4 and 5.2.2. The encoding is specified in table 5.6. The same instruction bits $I_{15:12}$ that control the sequencer are

Table 5.7: Execution units instruction format

$$\mathsf{R}_i^a \leftarrow \mathsf{O}_1(\mathsf{R}_i^a,\ \mathsf{R}_j^b,\ \zeta) \tag{5.13}$$
$$\mathsf{R}_j^b \leftarrow \mathsf{O}_2(\mathsf{R}_i^a,\ \mathsf{R}_j^b,\ \zeta) \tag{5.14}$$

used to control modifications to the current window pointer, according to the operations specified in table 5.1.

5.2.4 Execution Units

The execution units take their operands from two register banks, labeled R^a and R^b. Each execution unit can perform either one or both operations shown by equations 5.13 and 5.14 in table 5.7, where O_1 and O_2 are operations configured by the user, and ζ is some context, kept in the execution unit, which can depend upon previous operations. The instruction bits $I_{105:100}$ specify if the result of the operation is actually written back in the corresponding register or is discarded.

The operations performed by the three execution units are controlled by 21 common bits, $I_{126:106}$. Their distribution and function is entirely open and depends on the configuration of the units.

Each execution unit can work with a configurable number of registers per window: two sets of 4, 8, or 16. When using the minimum number of registers (two sets of 4) the instruction provides directly the required address bits in $I_{99:88}$. To use more registers per window, a portion of the 21 common control bits must be used as additional address lines.

The execution units can produce one condition bit, user-specified and configured, sent to the sequencer through a common, single line.

5.2.5 Special

Bit I_{127} is used to set breakpoints and to stop the processor. When Spyder reaches an instruction which has bit I_{127} set to 1, it will stop right after the second phase of the instruction, as described in section 5.1.

5.3 Hardware Implementation

Spyder has been implemented as a coprocessor attached to a SPARCstation. It uses a standard double-Europe VME board which fits inside the host computer and supports the standard 32-bit data VME slave interface [Mot85].

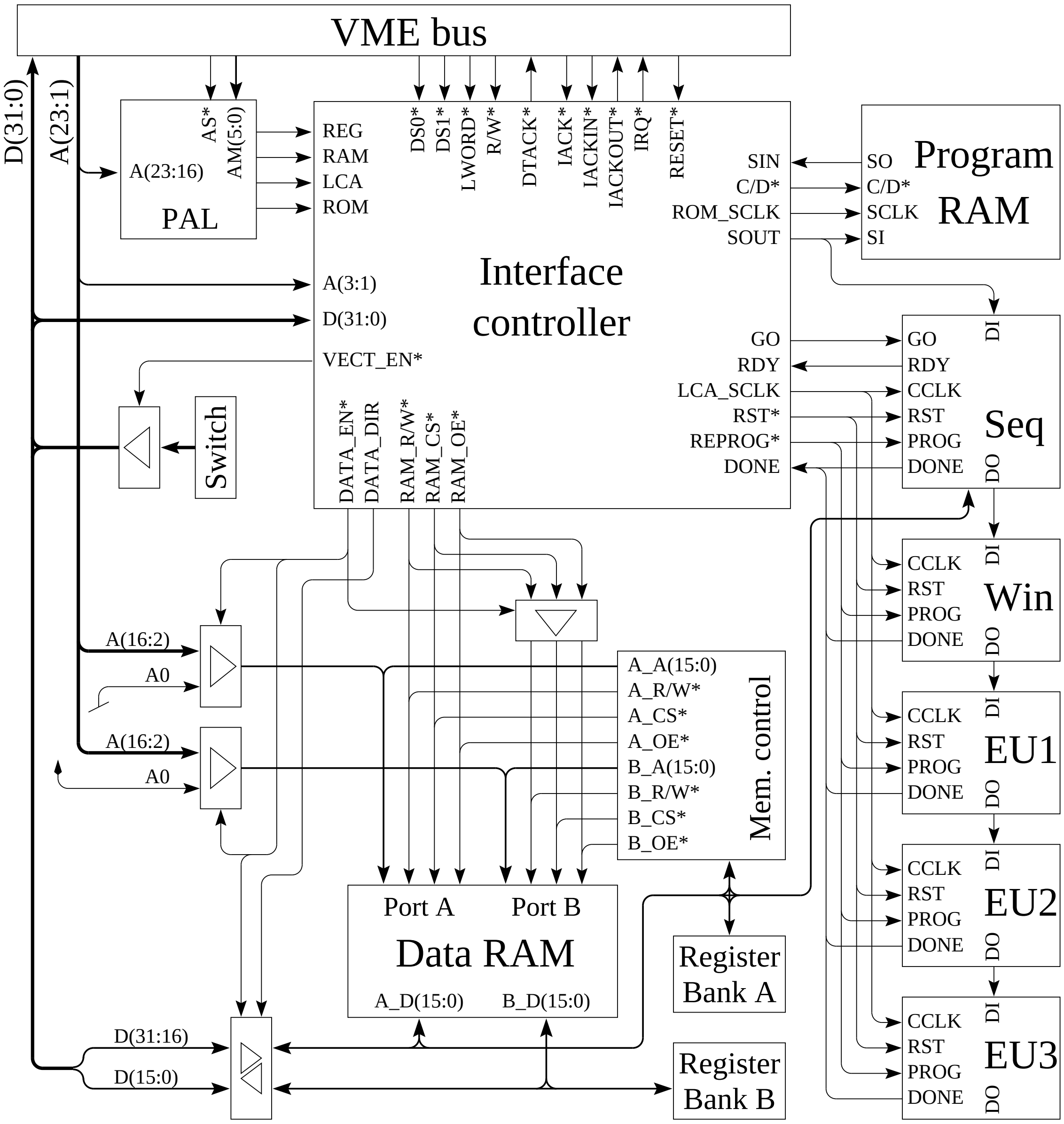

Figure 5.5: Block diagram of the VME interface and Spyder control signals

The current implementation of Spyder operates at 8 MHz, a limitation imposed not by the processor architecture, but rather by the choice of components determined by budget constraints.

5.3.1 The VME Controller

The purpose of the VME controller is to provide an interface between the VME bus and the Spyder processor, as shown in figure 5.5. The controller provides the following functions: implementing the VME 32-bit slave protocol, configuring the Xilinx FPGAs, reading and writing the Spyder program memory using its serial protocol channel, reading and writing the Spyder data memory, and, finally, generating and controlling the Spyder

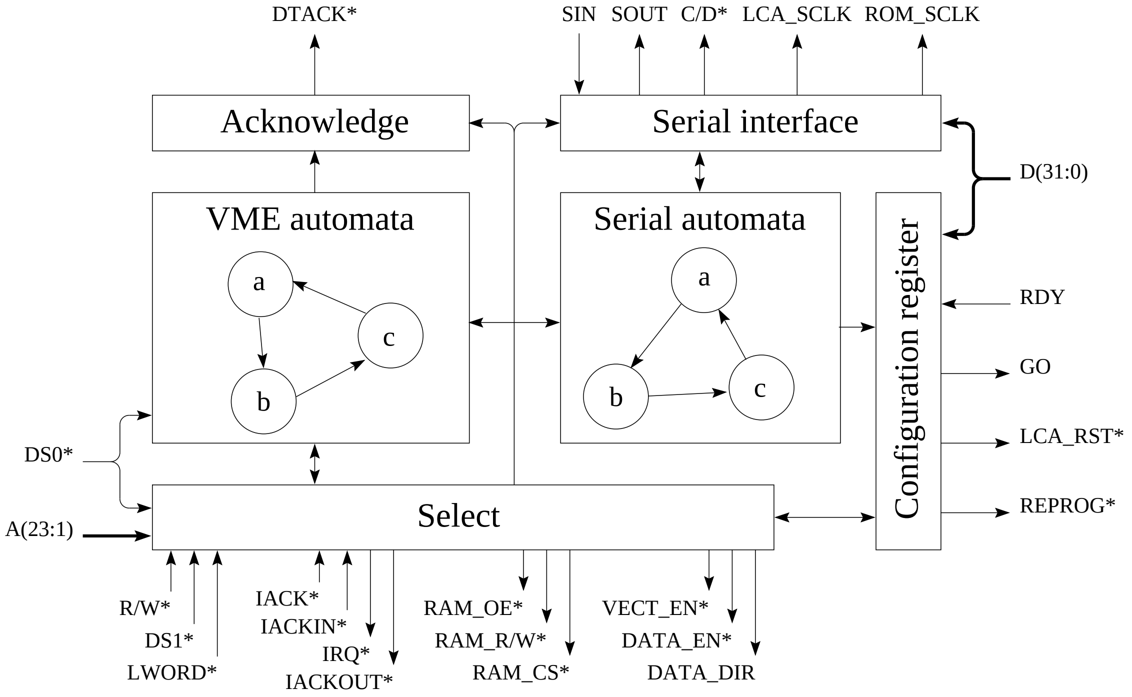

Figure 5.6: Block diagram of the VME controller

control signals.

The VME controller is implemented using an A1225 FPGA circuit from Actel. In addition, a fast PAL is used for the board address decoding, along with some line drivers and bidirectional bus drivers. The block structure of the complete controller is shown in figure 5.6. The controller is clocked by the master clock at 32 MHz and requires a cycle time of about 30 ns. The rest of the processor runs four times slower. See section 5.3.2 for details about the clock signals.

An Actel was selected because it does not need to be configured upon the power-up of the host computer and is instantly ready.

The controller uses the standard 24-bit address space with 32-bit data. It decodes the VME control signals $\overline{DS0}$, $\overline{DS1}$, $\overline{LWORD}$, $R/\overline{W}$, $\overline{AS}$, $AM_{5:0}$ and produces the acknowledge signal $\overline{DTACK}$. It is also capable of handling interrupt requests, using the $\overline{IRQ}$, $\overline{IACK}$, $\overline{IACKIN}$ and $\overline{IACKOUT}$ lines. The controller is reset by the VME $\overline{RESET}$ line. For a detailed description of the design of this VME controller, the reader is referred to T. Meyer's diploma report [Mey94].

The VME controller provides direct access to four components on the board. Each component is assigned a base address, as specified in table 5.8. The 128 Kbytes of data memory are directly accessible from the host workstation, as is the 8-bit control and status register. The program memory is accessed, in read and write mode, through a 64-bit shift register. The configuration of the FPGAs is performed through the same 64-bit shift register, but only write mode is allowed.

The data memory is accessed using 32-bit data width only. Both 16-bit ports of the IDT 7MB1006 circuit are accessed simultaneously, at addresses i and $i+1$, to provide the 32-bit data. The VME controller drives the $\overline{CS}$, $\overline{OE}$ and $R/\overline{W}$ lines of the memory.

Table 5.8: VME address of Spyder components

Component	Address	Size in bytes
Data memory	0x000000	0x20000
Program memory	0x020000	4
FPGAs configuration	0x030000	4
Control and status register	0x040001	1

Table 5.9: Control and status register description

Bit	Description	Write	Read
CSR_0	Interrupts handling	0: disable interrupts 1: enable interrupts	0: no interrupt 1: interrupt pending
CSR_1	Processor state	0: stop processor 1: start processor	0: processor running 1: processor stopped
CSR_2	Serial interface config.	0: write mode 1: read mode	
CSR_3	Serial interface state	0: no action 1: start transfer	0: busy 1: idle
CSR_4	$C/\overline{D}$ control	0: $C/\overline{D} \leftarrow 0$ 1: $C/\overline{D} \leftarrow 1$	
CSR_5	Unused	ϕ: no action	always 0
CSR_6	FPGA control	0: normal state 1: clear FPGAs config.	0: FPGAs not configured 1: FPGAs configured
CSR_7	FPGA control	0: normal state 1: reset processor	always 0

During such accesses, the Spyder processor must be stopped to avoid collisions.

The control and status register can be accessed using 8-bit data transfer only. This register controls and monitors all the functions provided by the VME controller. The description of the function of each bit of the register is summarized in table 5.9.

Bit CSR_0 can be set to enable VME interrupt requests by the Spyder processor. An interrupt is requested when Spyder reaches a breakpoint in the program. This same bit can be tested to see if an interrupt is pending. Bit CSR_1 is used to start and stop the processor, and it can be read to check if the processor has stopped running, i.e., has reached a breakpoint. Note that when the processor halts because it has reached a breakpoint, it must be stopped by setting bit CSR_1 to 0 before it can be restarted, as explained in section 5.3.2.

Bits $CSR_{4:2}$ control the serial interface. This interface is used both to configure the five Xilinx FPGAs and to load the program memory. The FPGAs are daisy-chained on the board: the sequencer chip is the head of the chain, followed by the windowing controller, the first and second execution units, and, finally, at the end of the chain, the third execution unit. The FPGAs are configured in slave serial mode, according to the Xilinx data book [Xil94].

The method used to configure the FPGAs is to download a configuration file to the FPGAs through the serial interface, as explained in chapter 6. This is done by repeating the following three steps:

1. wait for the serial interface to be idle, polling bit CSR_3;
2. write two long words to the FPGAs configuration address shown in table 5.8, thus filling the 64-bit shift register of the serial interface;
3. start the serial transfer by setting bit CSR_3.

These steps are repeated until the whole configuration has been downloaded. Padding bits after the end of the configuration file should be set to 1.

Before attempting to download a new configuration in the FPGAs, the old configuration must be cleared. This is accomplished by first setting and then resetting bit CSR_6.

As can be seen in figure 5.5, the FPGAs and the program memory share the same serial data line. The selection between the two devices is done by activating the requested serial clock. The controller decides which clock to activate based on which address was last used to access the shift register.

Loading the program memory is accomplished in a similar, but slightly more complicated way, described in section 5.3.3. Note that, usually, the user will just use the provided Unix driver and will not be required to handle all these details.

Setting and then resetting bit CSR_7 resets the Spyder processor, keeping the current FPGAs configuration. Note that this operation does not clear the content of the instruction pipeline register. The method used to clear this pipeline register is described in section 5.3.3.

5.3.2 The Sequencer

The sequencer is implemented in a Xilinx XC4005 chip [Xil94]. Its main purpose is to generate a 12-bit address for the program memory at each rising edge of the system clock (see section 5.2.1 for a detailed description of its functionality). The sequencer chip also implements a few other functions concerned with the generation of the clock signals and the control of the run/stop state of the processor. The complete schematic of the sequencer chip is given in appendix B, starting on page 145.

The master clock signal CLK is generated by an oscillator at 32 MHz. The VME controller uses this clock directly, to provide faster response times. However, such a frequency is too high for the rest of the processor. In particular, the load and store control unit is one of the most limiting component, requiring a cycle time of about 100 ns. Thus the system clock SCK is obtained by dividing the frequency of the master

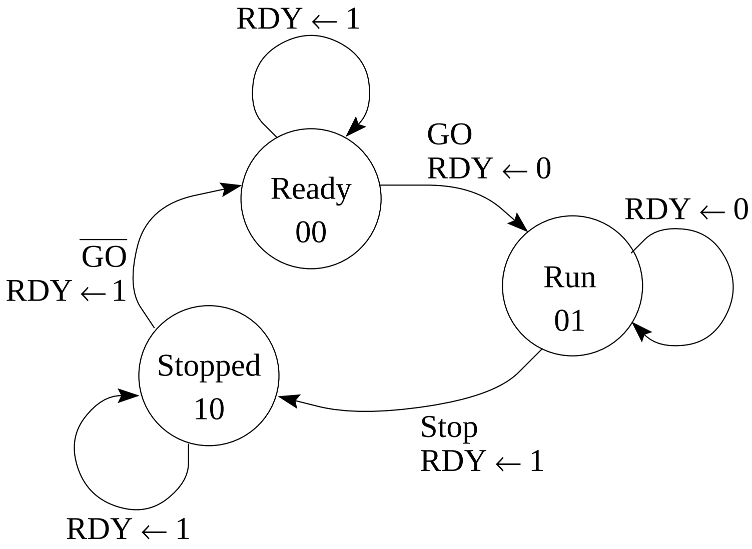

Figure 5.7: Run/stop state machine

clock by four, as shown in figure 5.9. The inverse of this clock, *SCKB*, is used as the output enable signal for the three ports of the data registers connected to the execution units (see section 5.3.6). Another clock, labeled the enable clock *ECK*, running at twice the speed of the system clock (half the speed of the master clock), is also used to generate clean write pulses for the register banks. The enable and system clocks are generated by a simple 4-state state machine. The Spyder processor is stopped by halting the state machine that generates the clocks, thus halting the clocks. Both clocks are stopped in the high state to allow the write operations to complete.

The state diagram of the run/stop state machine is given in figure 5.7. The *GO* line is controlled by bit CSR_1 of the VME controller. The stop transition occurs either when the bit I_{127} of the current instruction is set or when the *GO* line goes to 0. The sequencer generates two signals, *RDY* and *ENA*, which are low while Spyder is running and high otherwise. The RDY signal is used to inform the VME controller that the Spyder processor is idle (ready for further commands), while the *ENA* signal is used internally by the Spyder processor to enable the memory controller (see section 5.3.4). A timing diagram showing the master clock, the two generated clocks, and how the processor is started and stopped is given in figure 5.9.

The sequencer itself, presented in figure 5.8, is composed of four main parts: a multiplexer with some control logic that selects the address of the next instruction to be executed, a program counter register with an incrementer, a return address stack, and a counter stack. The multiplexer selects the next address among three possible sources: the instruction jump address field $I_{11:0}$, the program counter register, or the top of the return address stack. The selection logic is combinatorial and decodes the instruction command bits $I_{15:12}$, the state of the execution units flag, and the state of the *Zero* flag from the counter stack to select the appropriate source, as described in table 5.1.

The program counter register loads the current address, incremented by one, at each rising edge of the system clock.

The return address stack contains 64 levels. It is implemented using the RAM

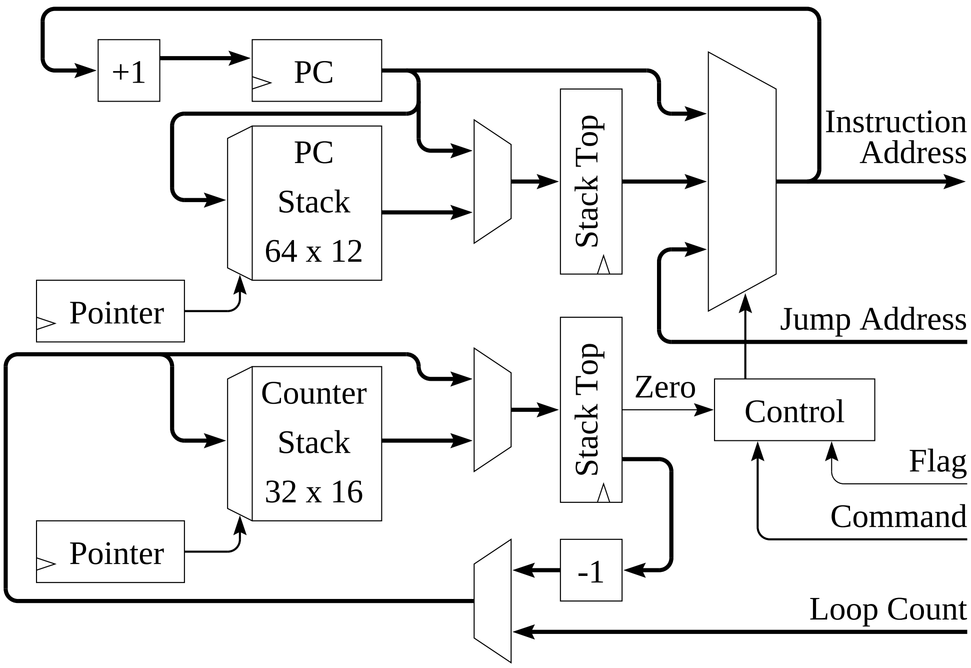

Figure 5.8: Block diagram of the sequencer

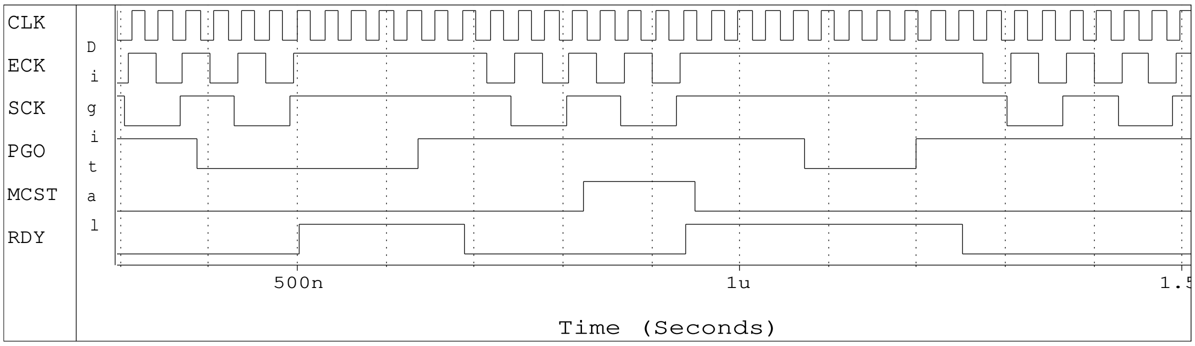

Figure 5.9: Timing diagram of the clocks and run/stop operations. At the beginning of the diagram, the processor is running. It is then stopped by the *GO* line (`PGO` in the figure) going low and restarted by the same line going high. The processor is then stopped by the breakpoint bit I_{127} (`MCST` in the figure) and restarted by cycling the *GO* line.

macros available in the XC4000 circuits, a stack pointer composed of a 6-bit up/down counter, and a register containing a copy of the return address on the top of the stack. The register provides a faster access to the return address when a return instruction is executed. Note that generating the write pulse for the RAM macros can cause some setup and hold time problems. In our case, these problems have been solved by generating short write pulses through combinatorial logic using the master, enable, and system clocks.

The counter stack provides 32 levels of counters. Its implementation is similar to the return address stack. This stack generates a *Zero* flag which is true when the current

Table 5.10: Serial protocol channel command codes

Command	Action
0	Read register
1	Write register
2	Read reg. and incr. counter
3	Write reg. and incr. counter
4–C	Reserved
D	Enter stub mode
E	Enter serial mode
F	No operation

Code	Register
0	Counter
1	RAM data
2	Pipeline register
3	Break mask register
4	Break data register
5	Control and status register
6	Data pins $I/O_{15:0}$ (Rd only)
7	RAM address (Rd only)
8–F	Reserved

counter will reach zero on the next decrement operation. A new counter value is pushed from the A-bus onto the stack when the A-bus transfer direction bits $I_{17:16}$ are both high.

5.3.3 The Program Memory

The program memory is implemented using 8 IDT 71502 chips [IDT92]. These chips feature 4K × 16-bit of static RAM and have two I/O ports: one 16-bit data, 12-bit address parallel port, and one serial port. The serial port is used to download the code from the host computer into the program memory, which avoids the need to multiplex the 32-bit data bus of the host computer with the 128-bit bus of the program memory. The IDT 71502 has a built-in pipeline register that is used by the Spyder architecture (see section 5.1). These chips also offer parity checking and data comparison with breakpoint capability, but these features are currently unused by the Spyder architecture.

The serial port is labeled Serial Protocol Channel (SPC) in the data book. It uses 4 wires: serial clock $SCLK$, data in SI, data out SO, and a control line $C/\overline{D}$. These 4 lines are handled by the VME controller. A complete description of the functions of the SPC can be found in the data book, but, for the sake of completeness, some of these functions are described below.

The SPC command register $SPC_{7:0}$ is 8-bit wide and is composed of two parts: $SPC_{7:4}$, the command code, and $SPC_{3:0}$, the register code. A description of these codes is given in table 5.10.

Downloading an object code file into the program memory through the serial interface is achieved by performing a number of operations. First, the processor is stopped and reset through the following two steps:

1. stop and reset the processor by writing 0 to bit CSR_1 and 1 to bit CSR_7;

2. return to normal state by writing 1 to bit CSR_7.

Then, the pipeline register is cleared. First set 0 in the SPC serial data register:

3. wait for the serial interface to be idle, polling bit CSR_3;

4. write 0 twice to the program memory configuration address shown in table 5.8, thus filling the 64-bit shift register of the serial interface with zeroes;

5. make sure the $C/\overline{D}$ line is in data mode by writing 0 to bit CSR_4 and start the serial transfer by setting bit CSR_3;

6. repeat operations 3 to 5 once.

At this point, the 128 bits of the serial data register of the SPC will contain 0. Now we have to send the write pipeline register command:

7. wait for the serial interface to be idle, polling bit CSR_3;

8. write 8 times the command 12, i.e., two long words containing 12121212, to the program memory configuration address;

9. make sure the $C/\overline{D}$ line is in command mode by writing 1 to bit CSR_4 and start the serial transfer by setting bit CSR_3;

10. wait for the serial interface to be idle, polling bit CSR_3, then reset the $C/\overline{D}$ line in data mode by writing 0 to bit CSR_4.

The high to low transition on the $C/\overline{D}$ line starts the command execution by the SPC. However, the command must be completed by at least one transition on the serial clock. Thus, after the last command has been sent to the SPC, we must arrange for at least one transition to occur on the serial clock, for example by performing steps 3 to 5 once again.

The next operation to perform is to set the counter register to 0. This is done by performing steps 3 to 10 once more, replacing the command 12 in step 8 by the command 10.

We can now start downloading the object code file, using the following steps:

11. wait for the serial interface to be idle, polling bit CSR_3;

12. write two long words from the object code file to the program memory configuration address (the least significant word is written first);

13. make sure the $C/\overline{D}$ line is in data mode by writing 0 to bit CSR_4 and start the serial transfer by setting bit CSR_3;

14. repeat operations 11 to 13 a second time to complete the 128-bit instruction word in the serial data register of the SPC;

15. wait for the serial interface to be idle, polling bit CSR_3;

16. write 8 times the command 31, i.e., two long words containing 31313131, to the program memory configuration address;

17. make sure the $C/\overline{D}$ line is in command mode by writing 1 to bit CSR_4 and start the serial transfer by setting bit CSR_3;

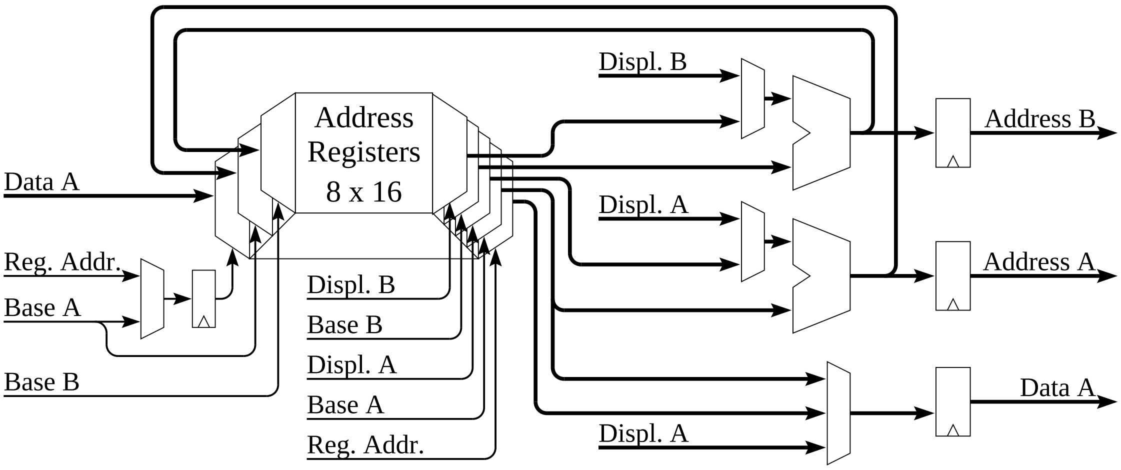

Figure 5.10: The load and store control unit

18. wait for the serial interface to be idle, polling bit CSR_3, then reset the $C/\overline{D}$ line in data mode by writing 0 to bit CSR_4.

Steps 13 to 18 are repeated until the whole configuration has been downloaded. Note that, as mentioned above, we must still arrange for at least one transition to occur on the serial clock.

5.3.4 The Load and Store Unit

The load and store control unit is implemented using an A1280 circuit from Actel [Act94]. This circuit is responsible for controlling the data transfers between the data memory and the data registers and for loading the loop counters in the sequencer. It contains 8 address registers and generates the addresses for both ports of the data memory. It has access to the A-bus to transfer the content of its address registers and to provide 16-bit constant values. A block diagram of the load and store control unit is given in figure 5.10. The complete schematic of the chip can be found in appendix C, starting on page 153.

An Actel chip was chosen to implement this circuit because of the high number of 16-bit wide multiplexers needed. The Actel technology seemed more appropriate for this kind of application. This circuit does not need to (and cannot) be reconfigured.

The load and store control unit generates two memory addresses per clock cycle: one for the A-bus transfer operation and one for the B-bus. Each address is the sum of the content of an address register, called the base register, with either the content of another address register or a 16-bit constant, called the displacement. Both computed addresses can be written back in their corresponding base register. Since the actual data transfer between the data memory and the data registers occurs during the third stage of the instruction pipeline (as described in section 5.1), both addresses are output through a register. Note that, as shown in figure 5.10, the write back of the computed address in the base register occurs during the second stage of the instruction pipeline.

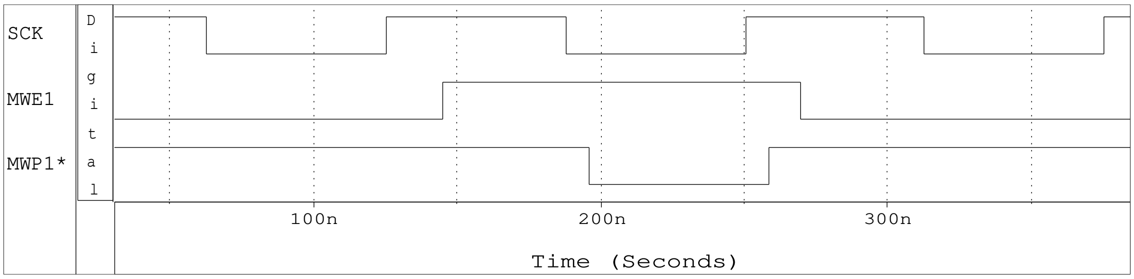

Figure 5.11: Timing diagram of the data memory write pulse

The circuit can load and store the content of any address register through the A-bus. It can also output the 16-bit *displacement* value, bits $I_{40:25}$ of the instruction, on the A-bus.

The address and control output pins of the circuit are set to the high-impedance state whenever the processor is not running, i.e., when the *ENA* signal generated by the sequencer chip is high.

The load and store control unit generates the chip select, output enable, and $R/\overline{W}$ signals for the data memory and for the corresponding I/O ports of the register banks. However, the actual write pulses are generated through a fast PAL 22V10 [AMD92], since the Actel circuit is not fast enough to generate correct write pulses directly. The PAL receives a write enable signal from the load and store control unit and synchronizes the write pulse with the system clock, as shown in figure 5.11.

5.3.5 The Data Memory

The data memory is implemented using an IDT 7MB1006 dual-port 64K $\times$ 16-bit SRAM module [IDT92]. This module provides 64K words of 16-bit data which can be accessed by two independent parallel I/O ports (the module also provides upper and lower byte access, semaphores, and interrupt mechanisms, but these features are not used by Spyder). Should a new version of Spyder be built, it will have a single-port data memory, accessed twice per cycle (once per register bank). The global behavior would be the same, and the data memory would be bigger and cheaper.

5.3.6 The Register Banks

The register banks are implemented using IDT 7052 four-port 2K $\times$ 8-bit SRAM chips [IDT92]. Two chips are used in parallel (one for each register bank) to provide 16-bit data words. These chips feature 2 Kbytes of static RAM with four independent input/output ports, allowing each register bank to be connected to the three execution units and to the data memory.

As explained in section 5.2.3, the registers are accessed through a windowing mechanism. Thus, the address of a register is decomposed into two parts: the most significant bits of the address specify the window and are generated by the window controller, while the least significant bits specify one particular register in the window and are generated by the execution units, the load and store control unit, or the window controller

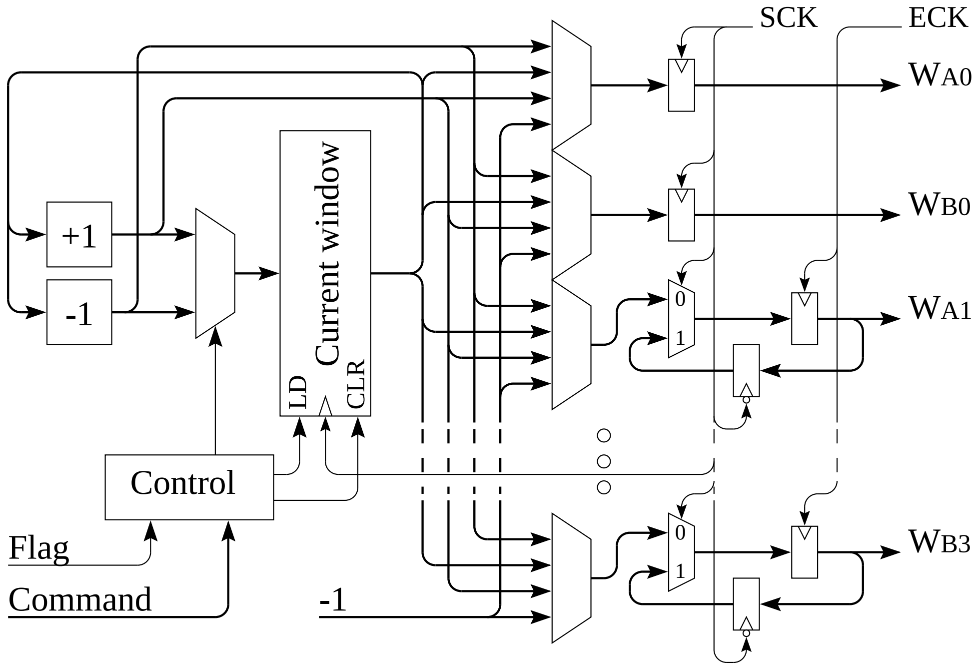

Figure 5.12: The window controller block diagram

(depending on which device is attached to the corresponding port).

The window size can be 4, 8 or 16 registers, thus each register bank contains 512, 256 or 128 windows respectively.

5.3.7 The Register Window Controller

The register window controller is implemented with a Xilinx XC4005 chip. This circuit is responsible for generating part of the register addresses, as described in section 5.3.6. The register window controller must generate eight times 9, 8 or 7 bits of the address (corresponding to 512, 256 or 128 windows respectively) at each clock cycle. Each address width determines a corresponding configuration of the window controller chip. The block diagram of the window controller is given in figure 5.12 and the complete schematic of the chip can be found in appendix D, starting on page 157.

The current window pointer is modified according to the command bits $I_{15:12}$ and the state of the execution units flag, as specified in table 5.1. The command is decoded through simple combinatorial logic. Each of the window components of the eight register addresses is selected, according to the instruction bits $I_{79:64}$, among four possible values: the current window pointer, the same pointer decremented or incremented by one, or the global window (i.e., all bits set to 1 to obtain the highest index).

There are two different cases for the output of the selected addresses. The first case is the simplest and deals with the output of the addresses for the transfers of data to/from the data memory. The transfers occur during the third stage of the instruction pipeline (as described in section 5.1) and thus the address bits W_{A0} and W_{B0} are output through a simple register.

The second case is the output of the addresses for the transfers of data to/from the

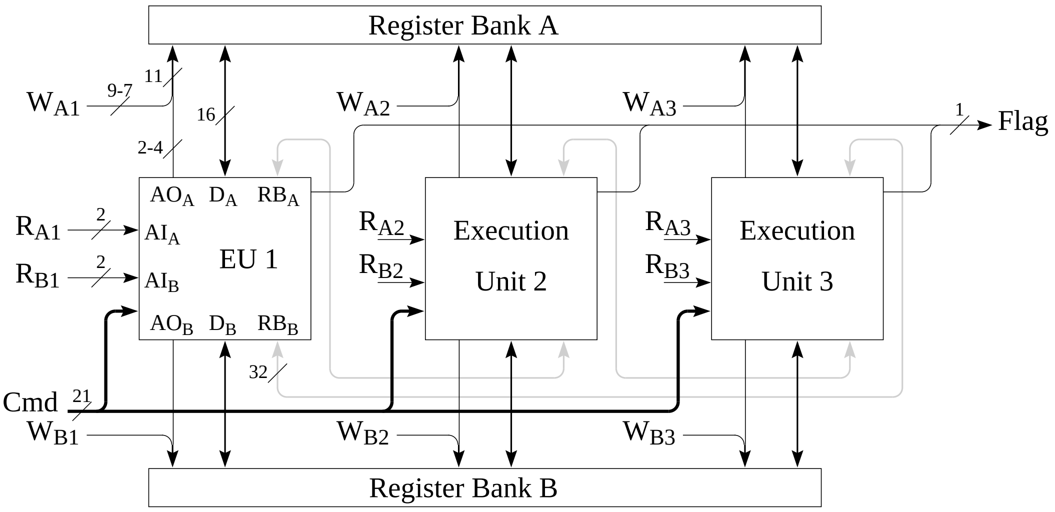

Figure 5.13: The execution units

execution units. There are actually two transfers: the first one occurs during the first half of the third pipeline stage, when the execution units read their operands from the registers. The second transfer occurs during the second half of the fourth pipeline stage, as the execution units write back their results in the registers. In this case, a register is used to keep a copy of the output addresses and the register addresses are output through a second register, clocked at twice the system clock speed. A multiplexer, controlled by the current state of the system clock, is used to select which address to output.

Since the number of registers per window can be configured, and since the load and store control unit cannot be reconfigured, the window controller receives bits $I_{83:82}$ and $I_{87:86}$, the two most significant bits of the register addresses for the memory data transfers, and outputs them as part of the register bank addresses when there are more than 4 registers per window. The two least significant bits of the register addresses for the memory data transfer are always output by the load and store control unit.

5.3.8 The Execution Units

The three execution units are implemented using Xilinx XC4008 circuits, which could be replaced, should the need arise, by pin-compatible XC4010 circuits. As shown in figure 5.13, they receive one 16-bit word of data from each of the register banks and can produce two 16-bit words of result, one for each register bank. They generate a 1-bit condition flag and two 4-bit addresses for the registers. The three execution units are also connected together in a ring through two 16-bit busses.

The input and output of the hardware operations implemented by the user are done through a standard register interface shown in figure 5.14. This interface should not be modified by the user, as it reflects the Spyder architecture as it was implemented on the printed circuit board. The schematic of this interface can be found in appendix E, starting on page 163.

The execution unit interface also generates the $R/\overline{W}$ signals corresponding I/O ports of the register banks. However, the actual write pulses are generated through a

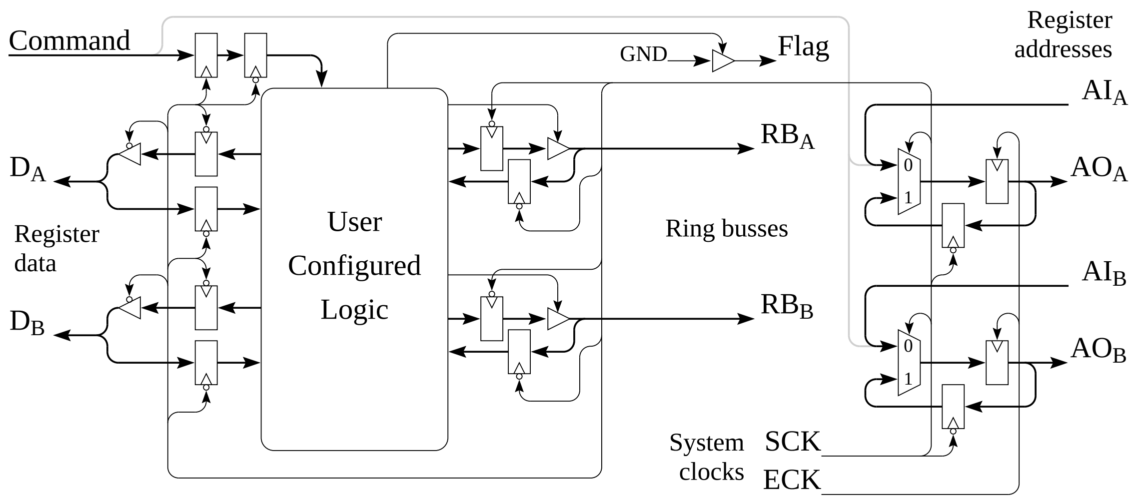

Figure 5.14: Block diagram of an execution unit

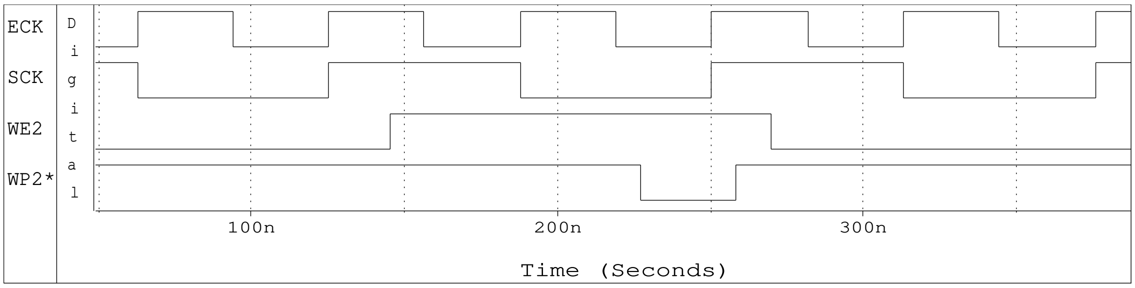

Figure 5.15: Timing diagram of the data register write pulse

fast PAL 22V10 [AMD92], since the Xilinx circuit is not fast enough to generate correct write pulses directly. The PAL receives a write enable signal from the execution unit and synchronizes the write pulse with the system clock, as shown in figure 5.15.

Chapter 6

Application Development Overview

This chapter provides a step-by-step description of the design of a complete application for the Spyder coprocessor. As an example, the following sections describe how to implement a fast simulation of a cellular automaton implementing the game of Life [BCG82]. The reader can refer to section 10.2 for details on the board and driver installations.

6.1 Application Overview

The game of life, as defined by the English mathematician John Conway [BCG82], is certainly one of the best-known cellular automata. A cellular automaton [TM87] can be described as an array of identical cells where each cell is a finite state machine. The state of the cells evolves as a function of time, the current state of the cell and the current state of the cell's neighbors. The time is divided into time steps, and at each step all the cells of the array simultaneously compute their future state as a function of their own current state and the state of some of their neighbors. A given cellular automaton is characterized by the number of possible states of the cell, the size of the neighborhood, and the transition rules used by the cells to compute their future state. There is no theoretical limit to the number of cells in an automaton, but in practice only finite rectangular arrays are used.

In the game of Life, each cell has only two states, dead or alive, and the future state depends of the current state of the cell itself as well as that of the 8 immediate neighbors, according to the following two simple rules: a living cell remains alive if it has 2 or 3 living neighbors, otherwise it dies; a dead cell becomes alive if it has exactly 3 living neighbors, otherwise it stays dead. As for the boundary conditions on the border of the array of cells, we will suppose that the cellular array is surrounded by immutable dead cells.

Despite the simplicity of these rules, the game of Life automaton can display very complex emergent behaviors which are due to interactions among cells. This complexity born out of simplicity best explains the high popularity of this automaton.

A program called `xlife`, simulating the game of Life automaton, is freely available by anonymous ftp on the host `ftp.x.org`. This program runs on the X window system [CGO92, Fou93, DDRS92, Man90] to graphically display the cellular array. The `xlife`

1. *Read an input file containing a representation of the initial state of the cells of the cellular automaton and store this initial state in the internal representation.*

2. *Display the state of the cellular automaton.*

3. *Compute the next state of the cellular automaton, according to user input, and go back to 2.*

Figure 6.1: Main algorithm of the cellular automaton simulator

program will be used as a starting point for the generation of our example. More precisely, we will use the same user interface and concentrate on rewriting the core of the program, i.e., the portion that performs the actual computation of the cellular array, to run on the Spyder coprocessor.

Since the cells have only two states, a single bit can be used to store the state of a cell, and thus 32 cells can be packed in a 32-bit word. The visualization of the state of the cellular automaton can be achieved by a simple bitmap, where dead cells are represented as black pixels and live cells as white pixels. As a consequence, the internal representation of the state of the cellular automaton and of the bitmap can be the same, removing any need of conversion between the two formats. The operation of the program can thus be summarized as in figure 6.1.

6.2 Splitting the Application

As remarked in chapter 4, the simplest way to make an application run with the Spyder coprocessor would be to let an intelligent compiler split the application into two parts, with one part running on the host workstation and the other on Spyder. Unfortunately, this compiler is not currently available and thus the user is forced to split the application by hand.

In this example, the division is relatively simple: the program running on the host workstation will handle all the user interface code, i.e., read the input file, wait for user input, and display the results; the code running on Spyder will perform the actual computation of the next state of the cells in the cellular automaton. Thus, the code running on the workstation has to provide the cellular automaton data to Spyder, instruct Spyder to compute the next state, and read back the new state from Spyder memory.

6.2.1 Accessing Spyder

The interface between the two parts of the program is provided by the C++ class Spyder, described in figure 6.2. This class provides ways to access the Spyder memory and to control the operation of the Spyder processor.

```
class Spyder {
public:
  unsigned long *ramPtr;
  unsigned char *ctrlPtr;
  virtual int WriteData(unsigned int start,
                        unsigned int length, void *data) = 0;
  virtual int ReadData(unsigned int start,
                       unsigned int length, void *data) = 0;
  virtual void Start(void) = 0;
  virtual void Stop(void) = 0;
  virtual bool RunningP(void) = 0;
  virtual void WaitAndStop(void);
};
```

Figure 6.2: The Spyder interface class

The `ctrlPtr` allows direct access to the Spyder control and status register. However, the preferred way to control the operation of the Spyder processor is through the `Start`, `Stop`, `RunningP` and `WaitAndStop` methods (see section 5.3.1 for a description of the control and status register). The Spyder processor can be in one of three states: ready, computing, or stopped. The `Start` method instructs the Spyder processor to start running, i.e., it changes the state of the processor from ready to computing. Once the processor has completed its computation, it will automatically enter the stopped state. The `Stop` method stops the processor and puts it back in the ready state, regardless of its current state. The `RunningP` method returns true if and only if the Spyder processor is still in the computing state. The `WaitAndStop` method waits for Spyder to finish its computation, i.e., enter the stopped state, and then puts it back in the ready state.

The Spyder memory should only be accessed while the processor is in the ready state. There are two ways to access the Spyder memory: through the `ramPtr` pointer and through the `WriteData` and `ReadData` methods. The `ramPtr` pointer gives direct access to the Spyder memory, without any overhead, but is a pointer to `long` and thus allows access only to 32-bit data words. The `WriteData` and `ReadData` methods provide a slightly higher-level interface to the Spyder memory: the `start` and `length` parameters are, respectively, the starting address and number of bytes to be read or written in the Spyder memory, while the `data` pointer points to the corresponding storage space in the workstation program memory.

6.3 Host Workstation Program

According to the subdivision defined in the previous section, the program running on the host workstation must do four things:

1. read the input file and initialize the data structure containing the description of the state of the cellular automaton; since the Spyder processor will need to operate on this data structure, it can reside in the Spyder memory;

2. initialize the window system environment which will display the state of the cellular automaton;

3. accept instructions from the user and instruct the Spyder coprocessor to compute the requested number of time steps;

4. wait for the Spyder coprocessor to finish the computation, read back the new state of the cellular automaton from the Spyder memory and display it.

The user communicates with the program using the workstation's three-button mouse. The program can be running either in single step or in continuous mode. While in single step mode, a click on the left mouse button tells the program to let Spyder compute the next state of the cellular automaton, display it, and then wait for further instructions; a click on the middle mouse button tells the program to enter the continuous mode with a time step of one; a click on the right mouse button also tells the program to enter the continuous mode, but with a time step of n, where n defaults to 10 but can be set by a command line switch.

While in continuous mode the program tells Spyder to compute the next state (or n next states, depending on how the program entered the continuous mode) of the cellular automaton, reads the new state from the Spyder memory, and then tells Spyder to again start computing while it displays the new state. A click on any mouse button returns to the single step state.

Since the purpose of this example is to explain how to use the Spyder processor, we will not provide further details about designing the X11 user interface and reading the input file. Only the code specific to the handling of the Spyder coprocessor will be described.

6.3.1 Sharing Data

As explained above, the computation of the new state of the cellular automaton will be performed by the Spyder coprocessor. Thus, the actual data representing the state of the cellular automaton must be stored in the Spyder memory by the host workstation. The Spyder coprocessor will also require some other information, for example the size of the array and the number of time steps to compute.

The programmer has to decide how to store all this information in the Spyder memory, taking into account the fact that the current implementation of Spyder has 16-bit wide data paths, i.e., that each memory word contains a 16-bit datum. For this example, the chosen layout is shown in table 6.1. The first few words contain the width and height of the array of cells and the number of iterations to compute per Spyder run; the width is expressed in number of 16-bit words, i.e., the width will be 1 for a 1 to 16 cells-wide array, 2 for a 17 to 32 cells-wide array, etc. The rest of the memory is used to store the state of the cells of the automaton. The description of the values stored at addresses 1 and 2 will be given a little later.

Table 6.1: Content of the Spyder memory

Spyder memory address	Content
0	zero
1	array width minus one
2	half of array height
3	array width
4	array height
5	number of iterations
$6 \ldots N$	cellular array state

```
1  extern Spyder *spyder;
2  char *curCells;
3  int boardHeight, boardBytes;
4
5  // Setup Spyder RAM
6  unsigned long data = (boardBytes >> 1) - 1;
7  spyder->ramPtr[0] = data;
8  data = ((boardHeight >> 1) << 16) | (boardBytes >> 1);
9  spyder->ramPtr[1] = data;
10 data = (boardHeight << 16) | 1;
11 spyder->ramPtr[2] = data;
12 unsigned long size = boardBytes * boardHeight;
13 spyder->WriteData(12, size, curCells);
```

Figure 6.3: Setting up the Spyder memory

Note that the memory module used on the current implementation of Spyder seems to have a problem when the workstation reads data from the Spyder memory: the content of address 0 is sometimes changed to 0xFFFF. As a consequence, this location will not be used to share data between the workstation and Spyder.

A code sample setting up the content of the Spyder memory according to the layout presented above is given in figure 6.3. Line 1 gives access to the Spyder object; line 2 declares a pointer to the cellular automaton state data (this data is allocated and filled according to the input file read by the simulation program); line 3 declares two integer variables containing the height of the cellular array and the number of bytes per line of the cellular array, i.e., the width of the cellular array divided by eight. Lines 6 to 11 use the `ramPtr` pointer to store data in the Spyder memory, while lines 12 and 13 use the `WriteData` method for the same purpose. Since the `ramPtr` pointer is a pointer to long words, i.e., 32-bit data elements, some computation has to be performed to set

```
1 // Wait for Spyder to terminate the computation
2 spyder->WaitAndStop();
3 // Get the new state
4 spyder->ReadData(12, boardBytes * boardHeight, curCells);
5 generations += (spyder->ramPtr[2] & 0xffff);
6 if (state == STOPREQ) {
7   state = STOP;
8 }
9 if (state != STOP) {
10   // Start a new computation
11   spyder->Start();
12 }
```

Figure 6.4: Getting the results of the Spyder computation

the 16-bit data, required by Spyder, in their proper places (the reason for the size of the data elements pointed to by `ramPtr` is that our implementation of the VME bus interface requires 32-bit data to access Spyder memory).

The code that reads back the new state of the cellular automaton is given in figure 6.4. Line 2 waits for Spyder to finish its computation; line 4 reads the new state of the cells of the cellular automaton; line 5 adds the number of time steps just computed by Spyder to `generations`, a variable holding the total number of steps computed so far. Lines 6 to 12 decide if a new state has to be computed, in which case line 11 tells Spyder to start again.

6.4 Programming Spyder

According to the job division presented in section 6.2, the task of the Spyder processor is to compute the next state of the cellular automaton. The method selected to perform this computation is a very simple scanning window algorithm.

6.4.1 Scanning Window Algorithm

Computing the next state of the cellular automaton means computing the next state of all the cells in the array. The data required to compute the next state of any cell are the state of that particular cell and the state of its eight surrounding neighbors. Thus, to compute the new state of the cell we need only look at a very small portion of the entire array; that portion is called a window. The basis of the algorithm is to move this window across the whole array, iteratively, starting from the top left corner of the array and proceeding from left to right and from top to bottom.

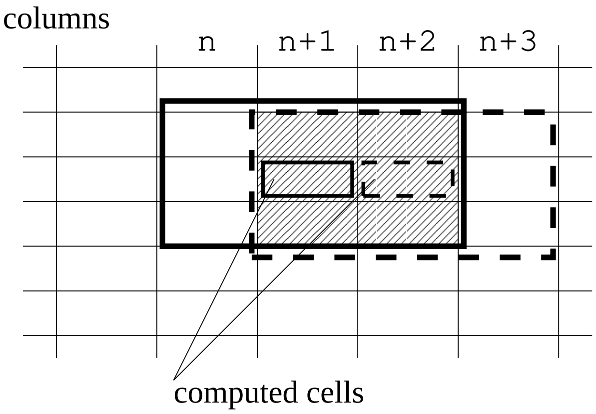

Figure 6.5: Reuse of six words in the execution unit

6.4.2 Splitting Again

The task running on Spyder has to be split again, by the programmer, in two parts, where one part will become the configuration of the execution units and the other will become the VLIW object code.

The execution units are used to perform the actual computation, which means, in this case, that they will be used to compute the new state of a cell given its present state and the state of its neighboring cells. The VLIW object code controls the operation of the execution units and is used to move data around between the Spyder memory, the register banks and the execution units.

6.5 Execution Units Configuration

To compute the next state of a cell, the execution unit needs to know the current state of the cell and the current state of its eight neighboring cells. Since the Spyder processor works with 16-bit data words, 16 cells of the cellular automaton are packed in a single data word and thus the execution unit will compute the next state of those 16 cells in parallel, which means that the execution unit must read nine data words to compute the next state of the cells in the central one.

Taking into account the fact that we use a scanning window algorithm, scanning the array from left to right, we can reduce to three the number of data words to be read to compute the next state of 16 cells. The reason is that to compute the next state of two consecutive words, six of the nine words required are the same for both words, as shown in figure 6.5. Thus, by preserving these six words in the execution unit, only three additional words need to be read.

It is now possible to define the interface of the `life` hardware operator, as shown in figures 6.6 and 6.7. The operator has three inputs `top`, `mid` and `bot` and one output `f`. Note that as the three words of column n are input to the operator, the output corresponds to column $n-1$.

The C^{++} code shown in figures 6.6 and 6.7 will be transformed into a hardware netlist by the `nlc` synthesizer, which is described in detail in chapter 7. Note that,

```
1 void
2 sumcol(bitvector l1 = bitvector(1), bitvector l2 = bitvector(1),
3        bitvector l3 = bitvector(1), bitvector &r = bitvector(2))
4 {
5   r = l1 + l2 + l3;
6 }
7
8 void
9 nextstate(bitvector c1 = bitvector(2), bitvector c2 = bitvector(2),
10           bitvector c3 = bitvector(2), bitvector c = bitvector(1),
11           bitvector &n = bitvector(1))
12 {
13   bitvector sum(4, 0);
14   sum = c1 + c2 + c3;
15   if (sum == 3)
16     n = 1;
17   else if (sum == 4)
18     n = c;
19   else
20     n = 0;
21 }
```

Figure 6.6: Definition of the hardware operator in C++, part 1

because of limitations in the ViewLogic system that will be used to process the output file of the `nlc` synthesizer, only lowercase letters should be used for function and variable names in the source code.

The data type used for describing the data handled by the hardware operators is called a `bitvector`. A `bitvector` variable can be viewed both as an integer variable and as an array of bits. Line 13 of figure 6.6 declares a 4-bit `bitvector` initialized to 0. Given that declaration, the value `sum` refers to the integer value represented by the 4 bits of the variable taken as a whole, while the value `sum[0]` refers to the value of the least significant bit of this variable. Lines 2 and 3 of figure 6.6 specify that the `sumcol` function takes three 1-bit wide input parameters and one 2-bit wide output parameter. A `bitvector` of unspecified length, like the parameters of the `life` function defined on line 2 of figure 6.7, is assigned a default width of 16.

The task of the `life` hardware operator is to compute the next state of the 16 cells of the central word according to the state transition rules given in section 6.1. The method adopted is to compute the number of live cells among the current cell and its eight neighbors: if this number is three the next state of the cell is alive and if the number is four the next state is the same as the current state, otherwise the next state

```
1 void
2 life(bitvector top, bitvector mid, bitvector bot, bitvector &f)
3 {
4   static bitvector la(17, 0), lb(17, 0), lc(17, 0);
5   bitvector c(2)[18];
6   bitvector i;
7
8   sumcol(top[15], mid[15], bot[15], c[0]);
9   for (i = 0; i < 17; i++) {
10    sumcol(la[i], lb[i], lc[i], c[i + 1]);
11  }
12  for (i = 0; i < 16; i++) {
13    nextstate(c[i], c[i + 1], c[i + 2], lb[i], f[i]);
14  }
15  // Store old values in registers
16  la[16] = la[0]; lb[16] = lb[0]; lc[16] = lc[0];
17  for (i = 0; i < 16; i++) {
18    la[i] = top[i];
19    lb[i] = mid[i];
20    lc[i] = bot[i];
21  }
22 }
```

Figure 6.7: Definition of the hardware operator in C^{++}, part 2

of the cell is dead.

To avoid repeating some additions, the task is divided into two steps: the first step is to compute the number of live cells per column, the second step is to add the column sums three by three and examine the result.

Since we need to compute the next state of 16 cells, we have to compute the sums for 18 columns: 16 central columns plus one on the left and one on the right. These 18 sums are kept in the `c` array declared on line 5 of figure 6.7, the left column having the highest index and the right column the lowest. The computation of the sums is performed by lines 8 to 11. The column on the right is provided by the most significant bit of the three inputs of the operator, and its sum is computed on line 8. The central columns, along with the left column, are memorized by the operator in the `la`, `lb` and `lc` static variables declared on line 4 of figure 6.7. The `sumcol` operator, defined on lines 1 to 6 of figure 6.6, simply adds its first three parameters, which are 1-bit values, and returns their sum, a 2-bit value, in the fourth parameter.

The second step is implemented on lines 12 to 14 of figure 6.7. The `nextstate` operator, defined on lines 8 to 21 of figure 6.6, takes the sums of three columns and the

```
life(bitvector top, bitvector mid, bitvector bot, bitvector &f)
{
  static bitvector la(17, 0), lb(17, 0), lc(17, 0);
#ifdef __NLC__
  bitvector c(2)[18];
#else
  bitvector c[18];
#endif
  ...
}
```

Figure 6.8: Modifications for regular C++

current state of the cell as inputs and computes the next state of the cell.

The last part of the `life` operator code, i.e., lines 16 to 21 of figure 6.7, stores the values of the six words of the two right-hand columns for use in the next computation. Since only the least significant bit of the words of the leftmost column are used, the rest of the word is discarded. The remaining bits are kept in `la[16]`, `lb[16]` and `lc[16]`.

6.5.1 Testing the Hardware Operators

Now that the `life` hardware operator has been designed, it must be tested. To this purpose, it will be compiled with a regular C++ compiler and linked with a small test program that should behave as intended.

A library implementing the `bitvector` class in C++ has been developed. It allows the use of a standard C++ development environment to simulate and debug the source code. It also allows the simulation and debugging of the complete application on the host computer before it is run on the Spyder coprocessor.

Some small modifications are required for the source code given in figure 6.7 in order to compile with a standard C++ compiler. In particular, modifications are needed in the array declarations, where the size specification of the `bitvector` variables must be removed. The size specification is required for the netlist compiler to be able to know the width of the components of the array. A regular C++ compiler has a different approach, which involves using dynamic objects, and thus does not absolutely require that kind of information.

In our example, the definitions of the `sumcol` and `nextstate` functions must be modified, along with the declaration of the `bitvector` array `c`. To facilitate this task, the netlist compiler defines a symbol `__NLC__` while compiling a file, making it possible to check which compiler is using the source code, and define things accordingly. The modifications are shown in figure 6.8.

A small test program is given in figure 6.9. After compiling this test program, the definition of the `life` hardware operator, and the `bitvector` class implementation

```
#include "bitvector.h"
extern void life(bitvector top, bitvector mid,
                 bitvector bot, bitvector &f);
int
main(int argc, char *argv[])
{
  bitvector top(16, 0), mid(16, 0), bot(16, 0), res(16, 0);
  life(top, mid, bot, res);
  bot = "0011100111001110";
  life(top, mid, bot, res);
  cout << "res: " << res << " (should be 0)\n";
  bot = 0;
  life(top, mid, bot, res);
  cout << "res: " << res << " (should be 0001000010000100)\n";
}
```

Figure 6.9: Test program for the `life` hardware operator

```
res: 0000000000000000 (should be 0)
res: 0001000010000100 (should be 0001000010000100)
```

Figure 6.10: Output of the test program of figure 6.9

(contained in a file named `bitvector.cc`) with a standard C++ compiler and linking the resulting object files with the required system libraries, an executable program should be obtained. Running this program should produce the output shown in figure 6.10.

6.6 Generating the VLIW Object Code using C++

The main task of the executable program is to provide data to the hardware operators and to transfer the results back to the Spyder memory. The programmer can describe this task using either C++ or an assembler. Both methods will be described in the following sections.

The `mcc` compiler accepts the same subset of C++ as the `nlc` compiler used to synthesize the hardware operators, with the addition of the `short` integer data type.

The `mcc` compiler requires three kinds of information: a description of the interface of the hardware operators, a description of the content of the Spyder memory, and the algorithm describing how the data must be fed to the operators and the result moved

```
1  extern void life(bitvector top, bitvector mid,
2                   bitvector bot, bitvector &f);
3  struct memory {
4    short zero;
5    short widthM1;
6    short heightH;
7    short width;
8    short height;
9    short iter;
10   short board[0];
11 };
12 extern struct memory *memory;
13 extern bitvector dummy;
14 short count;
```

Figure 6.11: Cellular automaton control program in C^{++}, declarations

back in the memory.

The control program for our example of the Life cellular automaton is given in figures 6.11 and 6.12. It uses a single hardware operator, namely `life`, which is defined on lines 1 and 2 of figure 6.11; the content of the memory, which we defined according to table 6.1, is described on lines 3 to 12 of figure 6.11. The `dummy` variable, defined on line 13 of that same figure, is used to discard unused results of operator calls (see for example line 10 of figure 6.12). Line 14 of figure 6.11 declares a global variable `count` (note that this variable will not be stored in the Spyder memory and will not be accessible from the host workstation).

The algorithm itself is given in figure 6.12 and implements the scanning window method described in section 6.4.1. It starts with some initializing steps, lines 7 to 10, and then computes the next state of the array of cells `iter` times (lines 11 to 36).

The `buffer` array is necessary to store the old state of the cells of line $n-1$ while line n is computed, to avoid it being overwritten by the new state which is stored in place. The size of 64 was chosen arbitrarily, allowing Spyder to handle arrays up to 1024 $(64 \cdot 16)$ cells wide. The array is initialized to zero, as we suppose that the cellular array is surrounded by immutable dead cells.

Lines 12 to 24 of figure 6.12 compute the next state of all the lines of the array, except the last one. The next state of the last line is computed by lines 25 to 35.

6.6.1 Using the Compiler

The source file of the program is transformed into a Spyder VLIW object code file by the `mcc` compiler. In addition to generating the loadable binary file of code, the `mcc` compiler

```
void
MCmain()
{
  short buffer[64];
  short i, j;
  bitvector res;
  for (i = 0; i < memory->width; i++) {
    buffer[i] = 0;
  }
  life(0, 0, 0, dummy);
  for (count = 0; count < memory->iter; count++) {
    for (i = 0; i < memory->height - 1; i++) {
      life(buffer[0], memory->board[i * memory->width],
           memory->board[(i+1) * memory->width], dummy);
      buffer[0] = memory->board[i * memory->width];
      for (j = 1; j < memory->width; j++) {
        life(buffer[j], memory->board[i * memory->width + j],
             memory->board[(i+1) * memory->width + j], res);
        memory->board[i * memory->width + j - 1] = res;
        buffer[j] = memory->board[i * memory->width + j];
      }
      life(0, 0, 0, res);
      memory->board[i * memory->width + j - 1] = res;
    }
    life(buffer[0], memory->board[i * memory->width],
         0, dummy);
    buffer[0] = 0;
    for (j = 1; j < memory->width; j++) {
      life(buffer[j], memory->board[i * memory->width + j],
           0, res);
      memory->board[i * memory->width + j - 1] = res;
      buffer[j] = 0;
    }
    life(0, 0, 0, res);
    memory->board[i * memory->width + j - 1] = res;
  }
}
```

Figure 6.12: Cellular automaton control program in C++, implementation

produces three C++ source files, which contain the description of each execution unit and must be compiled by `nlc` to generate the configuration files for the execution units.

6.6.2 Simulating the Whole Program

At this stage, it is possible to simulate the entire application without using the Spyder processor. This simulation requires the following: the C++ source code of the application that runs on the host computer, which we will suppose is stored in a file named `main.cc`; the C++ description of the hardware operator, stored in the file `lifeOP.cc`; the C++ version of the control program, stored in the file `life.cc`; the `bitvector` class implementation found in files `bitvector.h` and `bitvector.cc`; and a software simulation version of the Spyder processor, stored in files `SpyderSim.h` and `SpyderSim.cc`. By compiling all these files with a regular C++ compiler and linking the resulting object files, we obtain an executable program. Running this program on the workstation should produce the same results as the version for the Spyder coprocessor, albeit at a much lower execution speed. Since this program is run entirely on the host workstation, it is possible to use the usual debugging tools to locate potential problems.

6.7 Writing Real VLIW Assembler

The control program can also be written directly using an assembler, which gives the programmer full control over the operation of the Spyder processor, and is probably the only way to generate truly optimal VLIW object code.

When writing the code using the assembler, the programmer must also take over the other tasks of the `mcc` compiler, namely assign each hardware operator to a particular execution unit and define its unique set of operation codes.

6.7.1 Hardware Operator Assignment

Hardware operator assignment encompasses several tasks, of which only the first is obvious: decide which execution unit will implement which operator. As will be explained later, for our example we will implement the `life` operator twice: once in the first execution unit and once in the second. However, the `life` operator requires three inputs, while each execution unit has only two. Thus the three inputs have to be provided in two steps: the first step handles two inputs, which must be stored in registers, while the second step takes care of the third input and produces an output result. This, in turn, implies that the execution unit has to decode several operation codes and decide what to do accordingly. The programmer is responsible for defining these operation codes and implementing the appropriate decoding in the execution unit configuration.

The complete description of the first execution unit is given in figure 6.13. Line 1 defines which hardware library of operators to use for the hardware synthesis (in this case the library for the Xilinx XC4000 chip) and line 2 includes the `life`, `nextstate` and `sumcol` operators defined in section 6.5. Lines 4 to 8 define the interface of the execution unit, which is fixed by the Spyder architecture and is composed of two 16-bit inputs, a 21-bit command word, a 1-bit condition flag, and two 16-bit outputs.

Line 10 declares the two static variables which will hold the two inputs provided during the first step of the computation; these inputs are stored on lines 14 and 15 when bit 0 of the command word is set. The `life` operator itself is called on line 17 when bit 1 of the command word is set. The output of the execution is assigned on line 20. When bit 0 of the command word is set, i.e., during the first step of the computation, the output `aout` gets the input `bin`. This mechanism is used to move data from one register to another. During the last step of the computation, the output `aout` gets the

```
1 #include "x4000.h"
2 #include "lifeOP.cc"
3 void
4 eu_main(bitvector ain, bitvector bin,
5         bitvector cmd = bitvector(21),
6         bitvector &flag = bitvector(1),
7         bitvector &aout = bitvector(16),
8         bitvector &bout = bitvector(16))
9 {
10   static bitvector ntop(16, 0), nbot(16, 0);
11   bitvector res;
12   // Store first two values in registers
13   if (cmd[0] == 1) {
14     ntop = ain;
15     nbot = bin;
16   } else if (cmd[1] == 1) {
17     life(ntop, bin, nbot, res);
18   }
19   // Set the output
20   aout = (cmd[0] == 1) ? bin : res;
21   flag = 0;
22 }
```

Figure 6.13: Complete C++ description of execution unit 1

result of the `life` operator. In our example, the condition flag, used to pass a condition to the sequencer, is not used and is tied to zero on line 21.

The contents of the second execution unit are almost identical to those of the first. Its operation is controlled by bits 2 and 3 of the command word. The third execution unit is unused in this example.

6.7.2 Writing the Assembler Source File

Writing VLIW assembler is very similar to writing standard assembler. The main difference is that, instead of defining a single operation per line, the programmer must specify an operation for each execution unit of the processor, to be executed in parallel.

As explained in section 6.4.1, the control program must implement the scanning window algorithm, feeding data to the execution units in the proper sequence. To compute the next state of one row of cells, the processor must read the current state of three rows of cells. Moreover, two of these three rows will be needed again to compute the following row. Since Spyder has plenty of register space, it makes sense to keep the state of these two rows in the registers and thus avoid reading them from memory again.

```
1 // The address registers:
2 //  0 - stays to 0 to access global memory data
3 //  1 - the width of the array
4 //  2 - the width of the array minus 1
5 //  4 - reading row of first EU
6 //  5 - reading row of second EU
7 //  6 - writing row of first EU
8 //  7 - writing row of second EU
9 //
10 // The global registers:
11 //  1A - zero
12 //  1B - zero
13 Masks EU1 cmd(1,0) regA() regB()
14 Masks EU2 cmd(3,2) regA() regB()
15 Masks EU3 cmd() regA() regB()
```

Figure 6.14: Registers assignment

To compute the next state of the cells contained in one data word, nine data words are required. Of these, six are already stored in the execution unit and two are in registers, which leaves a single word which must be fetched from memory and another move from register to memory to store the computed result for each computation of the `life` operator. Since two instructions are needed for one call of the `life` operator, as explained in section 6.7.1, that leaves two free moves between the Spyder memory and the registers (since the memory has two ports). Thus, it is possible to compute two `life` operators in parallel by using two execution units.

The first step is to define how data will be stored in the registers. In our example, the scheme is defined through comments at the start of the assembler source file, as shown in figure 6.14.

6.7.2.1 Defining the Opcode Masks

The operations performed by the three execution units are controlled by 21 common bits. To aid the programmer in the task of setting these 21 bits properly, they can be split in nine sections through the use of masks. Each execution unit has three masks: one for its operation code and two allowing additional register address lines, needed when the register windows contain more than four registers (see section 6.7.3.1 below for a description of the register windows). The masks for our example are given in lines 13 to 15 of figure 6.14, where only the masks for the opcodes of execution units one and two are defined. Execution unit one uses bits one and zero for its opcode and execution unit two uses bits three and two; the third execution unit is unused.

```
//label: Memory A       , Memory B        , Exec unit 1   , Exec unit 2   , EU3 [, Sequencer]
start:  a[0]<-0         ,                 , NOP           , NOP           , NOP
        a[0]->M[a[0]+0] ,                 , NOP           , NOP           , NOP
        a[1]<-M[a[0]+3] ,                 , NOP           , NOP           , NOP
        a[2]<-M[a[0]+1] ,                 , NOP           , NOP           , NOP
        cntr<-M[a[0]+5] ,                 , NOP           , NOP           , NOP
        g[1]<-M[a[0]+0] , g[1]<-M[a[0]+0] , NOP           , NOP           , NOP
// Set up address regs and line counter
cont:   a[4]<-5         ,                 , NOP           , NOP           , NOP
        a[6]<-5         ,                 , NOP           , NOP           , NOP
        cntr<-M[a[0]+2] ,                 , NOP           , NOP           , NOP
// Here, we fill the registers in the windows before beginning the main loop
        cntr<-M[a[0]+3] ,                 , NOP           , NOP           , NOP
loop1:  c[0]<-0         , c[0]<-M[a[4]+1]*, NOP           , NOP           , NOP
        a[5]<-a[4]      ,                 , NOP           , NOP           , NOP, djnz loop1, W+
                        ,                 , NOP           , NOP           , NOP, jump, W0
// Load the column counter, put zeroes in the EUs and adjust some pointers
        a[7]<-a[4]      , M[a[5]+a[1]]*   , 1(g[1] ,g[1] ), 1(g[1] ,g[1] ), NOP
// First stages of the main loop
stage1: cntr<-M[a[0]+3] ,                 , 2(g[1], g[1] ), 2(g[1] ,g[1] ), NOP
        c[1]<-M[a[5]+1]*, c[1]<-M[a[4]+1]*, NOP           , NOP           , NOP
        n[1]<-M[a[5]+1]*, n[1]<-M[a[4]+1]*, 1(c[0]*,c[1] ), 1(c[1] ,c[0]*), NOP
                        ,                 , 2(c[1] ,c[0] ), 2(c[0] ,c[1] ), NOP, djnz stage2, W+
                        ,                 , NOP           , NOP           , NOP, jump finish1
stage2: n[1]<-M[a[5]+1]*, n[1]<-M[a[4]+1]*, 1(c[0]*,c[1] ), 1(c[1] ,c[0]*), NOP
                        ,                 , 2(c[1]*,c[0] ), 2(c[0] ,c[1]*), NOP, djnz main, W+
                        ,                 , NOP           , NOP           , NOP, jump finish2
// Go for the main loop
main:   n[1]<-M[a[5]+1]*, n[1]<-M[a[4]+1]*, 1(c[0]*,c[1] ), 1(c[1] ,c[0]*), NOP
        p[1]->M[a[6]+1]*, p[1]->M[a[7]+1]*, 2(c[1]*,c[0] ), 2(c[0] ,c[1]*), NOP, djnz main, W+
// Finish stages of the main loop
finish2:                , c[1]<-M[a[0]+0] , 1(g[1] ,g[1] ), 1(g[1] ,g[1] ), NOP
        p[1]->M[a[6]+1]*, p[1]->M[a[7]+1]*, 2(c[1]*,g[1] ), 2(g[1] ,c[1]*), NOP
finish3:M[a[4]+a[2]]*   , M[a[5]+a[2]]*   , NOP           , NOP           , NOP
        c[1]->M[a[6]+1]*, c[1]->M[a[7]+1]*, NOP           , NOP           , NOP, jump, W0
        M[a[6]+a[1]]*   , M[a[7]+a[1]]*   , NOP           , NOP           , NOP, djnz stage1
// Done one iteration
                        ,                 , NOP           , NOP           , NOP, djnz cont
*                       ,                 , NOP           , NOP           , NOP
                        ,                 , NOP           , NOP           , NOP, jump start
//
finish1:                , c[1]<-M[a[0]+0] , 1(g[1] ,g[1] ), 1(g[1] ,g[1] ), NOP
                        ,                 , 2(c[1]*,g[1] ), 2(g[1] ,c[1]*), NOP, jump finish3
```

Figure 6.15: Spyder control program in assembler

6.7.3 Assembler Structure

The assembler source code itself is shown in figure 6.15. Each instruction must be on a single line and is composed of eight fields: a breakpoint, a label, two memory transfers, three execution unit operations and a sequencer command. The file is passed to the C++ preprocessor before being assembled. Comments are thus defined as in C++: they start with a double slash (//) and extend to the end of the line.

The breakpoint is represented by a star (*): when Spyder encounters a breakpoint it will stop and signal the host workstation that it has completed its task.

The label is any string followed by a colon (:) and is used by the programmer to indicate a symbolic target for jump instructions.

Table 6.2: The available kinds of registers

Designator	Register kind	Available number
a	Address	8
g	Global	4, 8 or 16
c	Current	4, 8 or 16
n	Next	4, 8 or 16
p	Previous	4, 8 or 16
cntr	Loop Counter Stack	1

6.7.3.1 Memory Transfers

The memory transfer is composed of three parts: a register description, a direction, and the memory description. The register is defined by a letter followed by a number between square brackets, where the letter designates the register type according to table 6.2 and the number selects one particular register of that type (e.g., `a[4]` designates the address register number 4). The transfer direction is represented by an arrow pointing left (`<-`) or right (`->`). The memory description specifies the memory address of the transfer, obtained either by adding the content of two address registers (as in `M[a[0]+a[1]]`) or by adding the content of one address register with a constant (as in `M[a[0]+1]`). The resulting address can be saved back to the address register noted on the left by following the memory description with a star (e.g., `M[a[0]+1]*`).

The code letters for the registers of the global, current, next, and previous window, as well as for the address registers and the loop counter stack, are defined in table 6.2. The global window is the same at all times but the other three change according to a current window pointer, as described in section 5.2.3.

The address registers and the loop counter stack are separate entities from the two register banks.

The first memory transfer, on the assembler line, deals with the register bank R^a, the address registers, and the loop counter. The second memory transfer deals exclusively with the register bank R^b. The latter only accepts the register types `g`, `c`, `n`, and `p` as its left side. The first one accepts all register kinds as its left side and also allows its right side to be an address register or a constant, instead of the content of a memory address (e.g., `cntr<-5` pushes the constant 5 on the top of the loop counter stack and `a[5]<-a[4]` copies the content of address register 4 into address register 5).

6.7.3.2 Execution Units Operations

The specification of the execution unit operations is also split into three parts: the code of the operation to execute, one register from the bank R^a, and one register from the bank R^b. The code 0 should always be a no-operation for the execution unit (it is also possible to specify the reserved keyword `NOP`). The registers can be of the four kinds `g`, `c`, `n`, and `p`. If the value output by the execution unit must be written back, the register specification is followed by a star (`*`). For example, `2(c[1]*,g[1])` specifies operation

Table 6.3: Sequencer operations

Opcode	Operation	Window pointer operation
`call`	Call subroutine	Increment
`cbra`	Conditional jump	As specified
`ccal`	Conditional subroutine call	Conditional increment
`cret`	Conditional return	Conditional decrement
`djnz`	Decrement counter, jump if not zero	As specified
`jump`	Jump	As specified
`nbra`	Negative conditional branch	—
`retn`	Return	Decrement

code 2 applied to register `c[1]` from the bank R^a and `g[1]` from the bank R^b; the value output by the execution unit for the bank R^a must be written back in register `c[1]`.

As explained in section 5.1, Spyder uses a pipelined mode of execution: the values written back by the execution units into the register banks are not available during the next instruction: i.e., if instruction n specifies that the execution unit must write back a result in register `c[1]`, the value in this register during instruction $n+1$ will be the old value, while the new value will only be available for instruction $n+2$.

6.7.3.3 Sequencer Operations

The sequencer command is the last field of the assembler line. It is optional and, if omitted, control will simply pass on to the next instruction. The eight possible sequencer commands are given in table 6.3, together with their effect on the current window pointer. Three commands, namely `cbra`, `djnz`, and `jump`, allow the programmer to directly increment (`W+`) or decrement (`W-`) the current window pointer. For example, the command `djnz main, W+` will decrement the loop counter and jump to the label `main` if the counter is not zero and the current window pointer will be incremented regardless of the value of the counter. The `jump` command allows two more options: `W0`, which resets the current window pointer to zero, and `pop`, which pops the current loop counter from the loop counter stack.

If the target of the jump is omitted, it is assumed to be the next instruction. For example, line 16 of figure 6.15 resets the current window pointer to zero and control simply flows through to the next instruction.

The conditional branches are taken if the flag produced by the execution units is true; otherwise, control passes on to the next instruction. The `nbra` instruction applies the reverse condition, i.e., the branch is taken if the flag is false.

The sequencer contains a loop counter stack, so as to simplify the implementation of nested loops. Upon entering a loop, a new loop counter is pushed onto the stack using a memory transfer operation. For example, to loop five times we push five onto the loop counter stack with the instruction `cntr<-5` as the first memory transfer on the assembler line. At the end of the loop, the current loop counter, i.e., the loop counter on the top

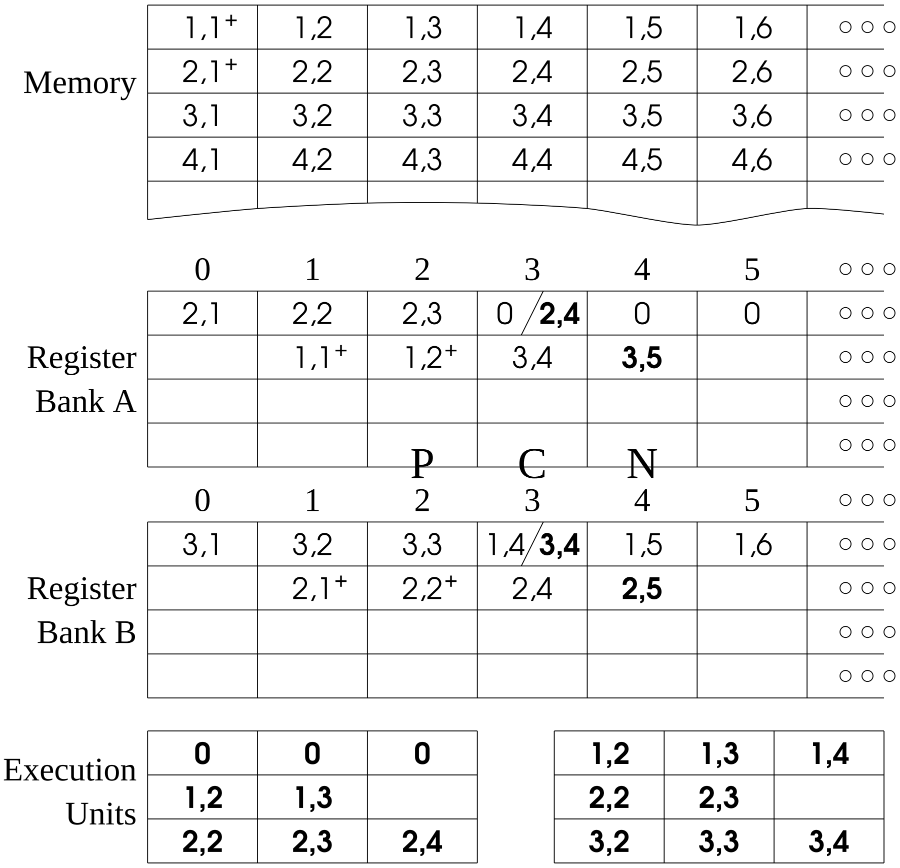

Figure 6.16: Main loop of the windowing algorithm, first instruction

of the stack, is decremented by a `djnz` instruction: if the new value of the counter is not zero, control jumps to the specified address, while if the new value is zero, the counter is popped from the stack and control flows through to the next instruction. It is possible to exit prematurely from the loop by popping the current loop counter from the stack with the `pop` option of the `jump` instruction described above.

6.7.4 Implementation of the Windowing Algorithm

The program given in figure 6.15 is essentially a hand-compiled version of the C++ program of figure 6.12. Lines 9 to 38 represent the loop that is repeated `iter` times to compute `iter` time steps of the cellular automaton.

The loop at lines 14 and 15 moves a line of zeroes and the first line of the cellular array in the register banks. The registers `c[0]` of each window of bank R^a contain the line of zeroes and the registers `c[0]` of each window of bank R^b contain the first line of the cellular array. Line 16 resets the current window pointer to zero.

Lines 18 to 27 are preparations to enter the main loop of the program, found at lines 29 and 30 of figure 6.15 and composed of two instructions. The effect of these instructions is shown in figures 6.16 and 6.17 and is described below in some detail. We assume that the program is computing the next state of the first two lines of the

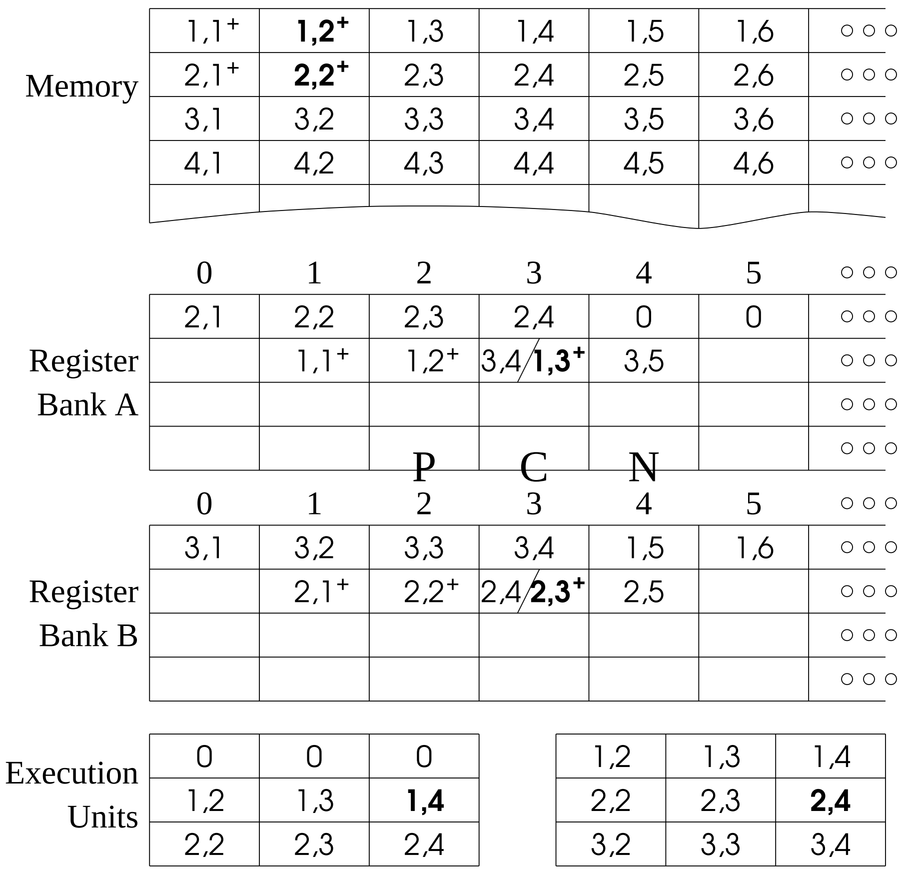

Figure 6.17: Main loop of the windowing algorithm, second instruction

cellular array. Each word of the memory contains 16 cells and is referenced by its row and column position in the cellular array (e.g., word 1,2 is the word on the first line and second column of the array). The current, previous, and next register windows are represented by `C`, `P` and `N` respectively. Values that have just been moved are shown in bold face.

6.7.4.1 First Instruction of the Main Loop

The first instruction is found on line 29 of figure 6.15 and its effects are shown in figure 6.16. It selects operation 1 of the first and second execution units, which shifts to the left the columns of cells in the two execution units.

Each of the two execution units reads one word of the row preceding and one word of the row following the current row from the current window (`C`). The contents of the following row are moved over the old state of the preceding row, so that the data are ready in the registers for the computation of the next two rows.

The next word of the two rows following the current rows are moved from the data memory into the registers in the next window (`N`). In this example the current rows are the first and second, and so words from the second and third rows are moved. The memory pointers `a[4]` and `a[5]`, which point at the data to be read, are incremented by one.

6.7.4.2 Second Instruction of the Main Loop

The second instruction is found on line 30 of figure 6.15 and its effects are shown in figure 6.17. It selects operation 2 of the first and second execution units, which causes them to compute the next state of the central cells. Each of the two execution units reads one word from its current row in the current window (`C`), computes the new states of the 16 central cells, and saves the results back in the registers in the current window. For example, the first execution unit reads its data from register `c[0]` in bank R^a and writes its result in register `c[1]` in bank R^b. Because of the pipeline, the value of cell 1,4 is still available in register `c[0]`, and the new value 3,4 will only become visible for the next instruction.

The new state of the cells computed during the previous iteration, available in the previous window (`P`), are written back into the data memory.

The window pointers are advanced, the loop counter decremented, and, if it has not reached zero, control returns to the beginning of the loop.

Lines 32 to 36 and 42 to 43 cleanly terminate the work of the main loop in all possible cases. When the requested number of time steps have been computed, the Spyder processor will stop on line 39. When it is restarted, line 40 ensures that the program restarts from the beginning.

6.7.5 Using the Assembler

Assuming that the assembler source code for the cellular automaton that has just been described is stored in a file called `life.spy`, the task of the assembler is to generate a binary file that can be loaded directly into the program memory of the Spyder processor. Invoking the assembler with the command `imac life.spy life` will produce two output files `life.bin` and `life.mic`: the former is a binary file containing the produced object code which can be downloaded directly into the Spyder program memory, while the latter is a textual representation of the object code.

6.8 Synthesizing the Execution Units

At this stage, the complete source code, in C++, of each execution unit should be available. For this example, only two execution units are being used, and the source code is found in the files `EU1.cc` and `EU2.cc`, which were generated either automatically by the `mcc` compiler or manually by the user. Both files use the file `lifeOP.cc`, which contains the source code of the `life` hardware operator, developed in section 6.5. A synthetic view of the global design flow of a Spyder application is given in figure 6.18.

The `nlc` compiler produces two kind of output files for the ViewLogic CAD tools: symbol files and netlist files. ViewLogic uses projects to define a complete design entity, where a project is composed of a directory containing a file `viewdraw.ini`, which defines basic properties of the project such as the components library to use, and three subdirectories: `sch`, `sym` and `wir`, which contain the project's schematic, symbol, and netlist files respectively.

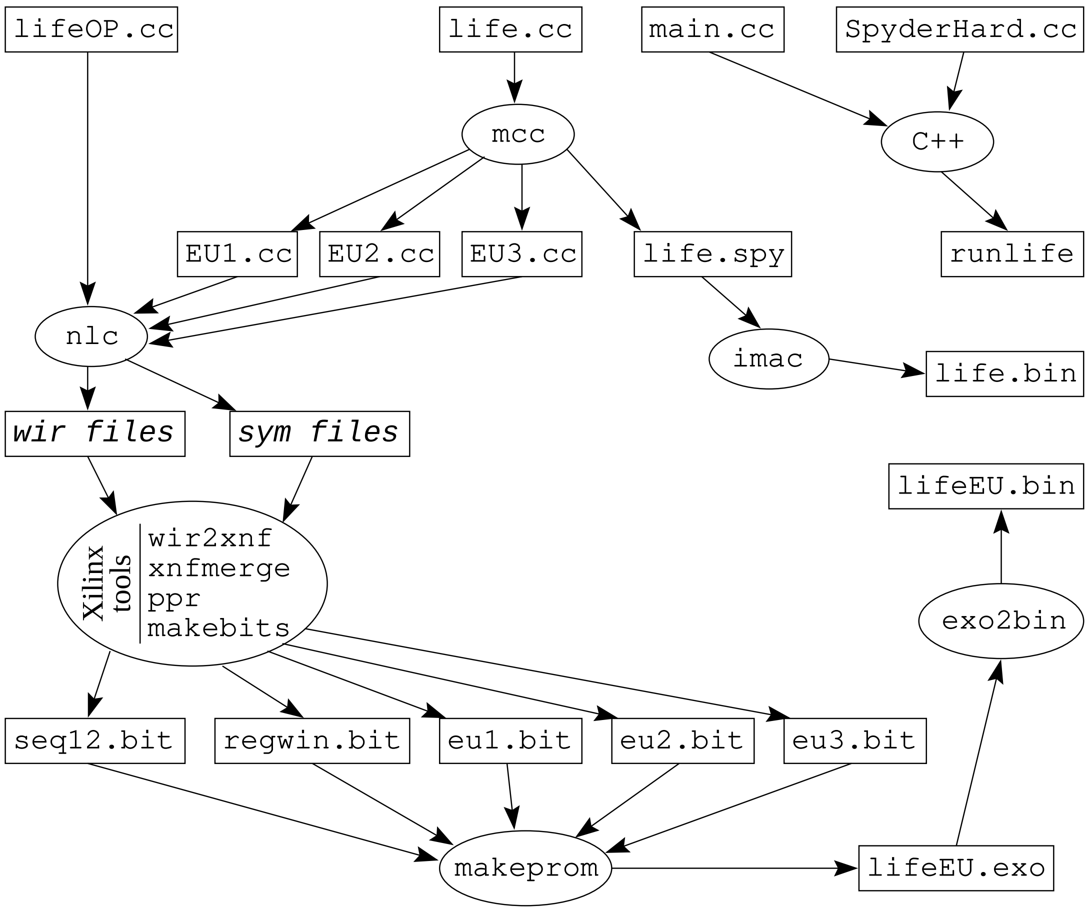

Figure 6.18: Overall design flow of a Spyder application

The `sch`, `sym`, and `wir` subdirectories should already contain the files describing the hardware interface of the Spyder execution units, described by the files `bio.1`, `eu.1`, `eupins.1`, `io.1`, and `outra.1` in each of the subdirectories. The main project directory should also contain a file named `eu.cst`, which is the pins placement constraint file for the Xilinx tools.

The hardware operators are synthesized by the following command:

```
nlc -O -N project/wir -S project/sym -I library EU1.cc
```

where `project` is the ViewLogic project directory and `library` is the directory containing the `x4000.h` file that describes the hardware library components for the Xilinx XC4000 family. The `-O` switch tells `nlc` to try to optimize the generated netlist. This command should produce files named `eu_main.1`, `life.1`, `nextstate.1`, and `sumcol.1` in the `wir` and `sym` subdirectories.

The next step is to produce a file `eu.bit`, which contains the bitstream needed to configure a Xilinx chip, using the Xilinx CAD tools `wir2xnf`, `xnfmerge`, `ppr`, and `makebits`. A description of these tools can be found in the Xilinx manuals [Xil91].

The file `eu.bit` is stored in a safe place, the directories are cleaned, and the same operations are repeated a second time to produce the bitstream for the second execution unit, which will be saved in the file `eu2.bit`.

The Spyder coprocessor contains five Xilinx chips which must be configured together. It is thus necessary to produce a combined bitstream containing the configuration information for all five chips. This is accomplished by using the `makeprom` Xilinx tool:

```
makeprom -s 128 -f exo -o lifeEU -v -u 0
         seq12.bit regwin.bit eu.bit eu2.bit eu2.bit
```

where `-s 128` specifies a size, `-f exo` specifies the output file format, `-o lifeEU` specifies the name of the output files, `-v` means verbose, and `-u 0` specifies to load the following files starting from position 0 and moving upward. The command loads the file `seq12.bit`, which contains the bitstream for the sequencer, the file `regwin.bit`, which contains the bitstream for the register windowing controller, and the bitstream for the execution units. Since a bitstream must be provided for all the execution units, the bitstream for the second execution unit is used twice. The `makeprom` command above produces a file `lifeEU.exo` which must be converted to binary format with the following command:

```
exo2bin lifeEU.exo > lifeEU.bin
```

the file `lifeEU.bin` can then be used to configure the Spyder execution units.

6.9 Running the Program

We assume that the file `life.bin` contains the binary object code of the control program, that the file `lifeEU.bin` contains the binary configuration for the Spyder execution units, and finally that the user is logged on the host workstation containing the Spyder board.

The configuration of the Spyder execution units is achieved by the command:

```
cat lifeEU.bin > /dev/spyder.lca
```

Loading the object code into the Spyder program memory is achieved by the command:

```
cat life.bin > /dev/spyder.rom
```

As a side effect, loading the program memory resets the Spyder processor, a feature which can be useful if the processor is stuck in an infinite loop.

The Spyder processor is now ready and the user can launch the control program on the host workstation.

Chapter 7

The Hardware Synthesizer nlc

The C++ to netlist compiler, or `nlc`, is used to create the configuration of the execution units from a high-level description of their functionality defined using a subset of C++ [Str91]. This chapter presents the source language translated by this compiler and describes the algorithms used by the compiler to synthesize hardware. For an explanation of the usage of this compiler, see chapter 6.

7.1 Hardware Synthesis: an Example

Hardware synthesis is the translation of a source file, written in some high-level language such as VHDL or C++, into a form suitable to be implemented into a hardware circuit. In this chapter we present `nlc`, a compiler able to translate a source file written in C++ into a netlist, which is then processed by CAD tools from ViewLogic [Vie95] and Xilinx [Xil94] to become a bitstream, i.e., a file used for the configuration of an FPGA circuit.

This section presents an overview of hardware operators from a CAD tool point of view, as well as a simple example of hardware synthesis.

7.1.1 Hardware Operators

In ViewLogic, a hardware operator, also called a component, is composed of two parts: an interface and an implementation. The interface, also called the symbol, is a file that describes the input and output ports, also called the pins, of the component. It contains various attributes about the component as a whole, or about individual pins, as well as graphical information, i.e., how the symbol should be drawn by the CAD tool. The implementation is stored in another file, called the netlist or the wirelist, which describes how the hardware operator is implemented. This file contains the list of elements that are needed to implement the operator and how these elements are wired together and to the inputs and outputs of the operator.

7.1.1.1 The Symbol of a Hardware Operator

The symbol file contains three kind of informations: the name of the input and output ports of the component, drawing information for the component, and various attributes of the component as a whole and of the input and output ports.

The input and output ports are called pins in the symbol file jargon. Each pin represents one input or output port and can be a single bit or an n-bit bus. Each pin must have a unique name, which is used to determine the width of the port (i.e., the name `FOO` represents a single bit pin and the name `FOO[15:0]` represents a 16-bit bus, where the bits of the bus are named `FOO15`, `FOO14`, ..., `FOO0`).

Attributes are used for many different purposes. An attribute can either be attached to a single pin, or to the symbol as a whole. An attribute is composed of a name and a value. For example, each pin has an attribute that defines if the pin is an input or an output port. The name of this attribute is `PINTYPE` and its value is `IN` for inputs and `OUT` for outputs.

7.1.1.2 The Netlist of a Hardware Operator

The netlist file is composed of four parts. The first part is a header that defines the name of the component and the name of the global signals used, e.g., the ground `GND` signal.

The second part of the file is the description of which hardware elements are used to implement the operator (logic gates, flip-flops, etc.). Each element is defined by its name and a set of attributes. Most of these attributes are the same as the attributes defined in the corresponding symbol file, but at least one is specific to the netlist file: the `PINORDER` attribute which defines the name and the order of the input and output ports of the element.

The third part defines how each instance of the library elements is connected. The specification is implemented by listing all the wire names to which the element is connected, in the order specified by the `PINORDER` attribute described above.

The fourth and final part defines the wiring of the input and output ports.

7.1.2 Hardware Operators Library

To synthesize a netlist, the `nlc` compiler requires that a set of library elements (e.g., `and` gates) be defined. The programmer must therefore include the definitions of the library elements with the source file being compiled. The information needed to write these definitions must be gathered by the programmer in the symbol files provided by the FPGA chip manufacturer for each FPGA architecture the programmer intends to use.

For example, the symbol file for a hardware inverter of the Xilinx 4000 chip family is given in figure 7.1. From this file, the programmer can construct the description of the inverter hardware operator needed by `nlc`, as given in figure 7.2. Please refer to section 7.2.2 for a precise description of the syntax of the hardware operators description.

7.1.3 Example

This section presents a very simple example of hardware synthesis: the synthesis of a counter.

```
V 50                                P 2 0 20 20 20 0 2 0
K 419705751400 INV                  L 10 20 10 0 9 0 0 0 I
Y 0                                 A 10 20 10 0 7 0 PINTYPE=IN
D 0 0 70 40                         l 2 50 20 20 5
Z 10                                l 2 50 20 20 35
i 2                                 l 2 20 5 20 35
P 1 70 20 50 20 0 3 1               U 0 -80 10 0 1 0 @ODLY1=1NS
L 60 20 10 0 3 0 0 0 0 O            U 0 -60 10 0 1 0 @ODLY0=1NS
A 60 20 10 0 1 0 PINTYPE=OUT        U 0 -40 10 0 1 0 @IDLY1=0
A 70 30 10 0 3 0 PARAM=INV          T 30 -5 10 0 5 INV
U 0 -120 10 0 1 0 DEVICE=INV        U 0 -20 10 0 1 0 @IDLY0=0
U 0 -100 10 0 1 0 LEVEL=XILINX      E
```

Figure 7.1: Symbol file of an XC4000 hardware inverter

```
asm ~ {
  "x4000:inv" <"LEVEL=XILINX" "DEVICE=INV" "@%iDLY0=0"
              "@%iDLY1=0" "@%oDLY0=1NS" "@%oDLY1=1NS">,
  ("i"), ("o" <"PARAM=INV">)
}
```

Figure 7.2: Description for `nlc` of the inverter of figure 7.1

The purpose is to generate a 3-bit hardware counter whose value must be incremented by one at each clock cycle and which must count from zero to five inclusive.

One possible solution is given in figure 7.3. Line 1 includes the definitions of the available hardware operators in the Xilinx XC4000 chip library. The file `x4000.h` contains definitions similar to the one shown in figure 7.2. Line 3 defines the name and the interface of the hardware operator to be synthesized: the name is `count6a` and it has one 3-bit wide output port. Line 5 states that the operator needs a 3-bit hardware register, initialized to zero, to store the current value of the counter. Line 6 increments the value of the counter by one, and lines 7 and 8 check that the value of the counter does not go above 5. Line 9 assigns the current value of the counter to the output of the operator.

For completeness, it is assumed that an external device provides a clock signal to the hardware register; i.e., the state of the static `state` variable is updated once every clock cycle.

The result obtained by synthesizing the source code given in figure 7.3 for the Xilinx XC4000 chip family is given in figure 7.4. The hardware operators shown in figure 7.4

```
#include "x4000.h"
void
count6a(bitvector &out(3))
{
  static bitvector state(3, 0);
  state++;
  if (state == 6)
    state = 0;
  out = state;
}
```

Figure 7.3: Source code of a 3-bit, zero to five counter

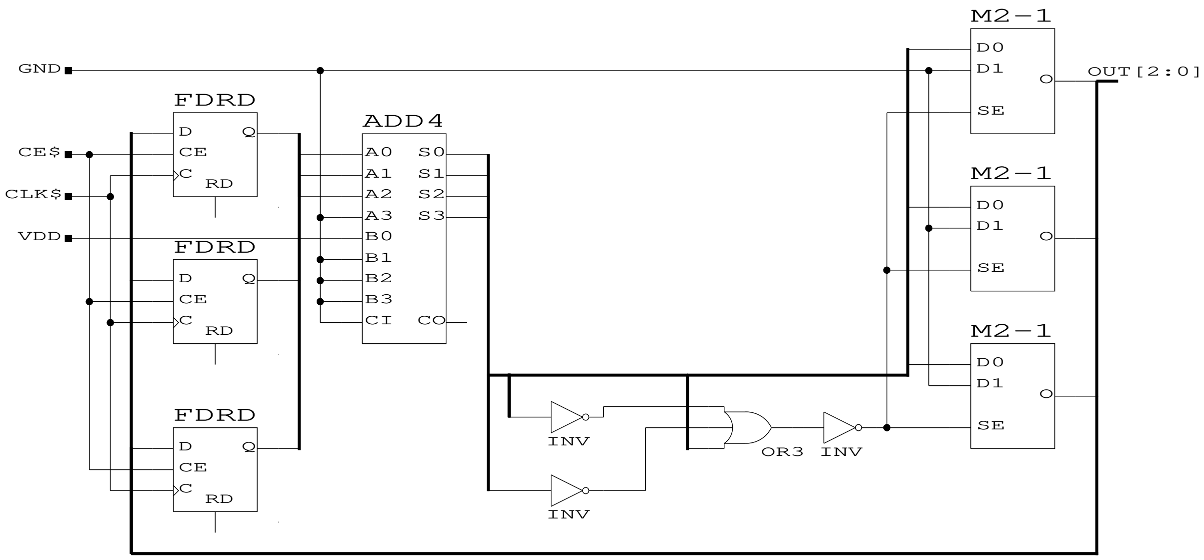

Figure 7.4: Synthesized 3-bit counter from figure 7.3

are those found in the operators library of the XC4000 chips.

Note that, because of limitations in the ViewLogic system that will be used to process the output file of the `nlc` synthesizer, only lowercase letters must be used for function and variable names in the source code.

7.2 The Language

The language of choice is a small subset of C^{++}, with a few extensions. It is a subset in the sense that there is no support for classes and inheritance, there are no pointers and the array type is limited to one-dimensional arrays. The extensions allow the definition of the target FPGA building blocks (basic library elements) and provide for a little more

```
bit a[16], b[8], c[16];
c = a + b;
```

Figure 7.5: Sample code using arrays

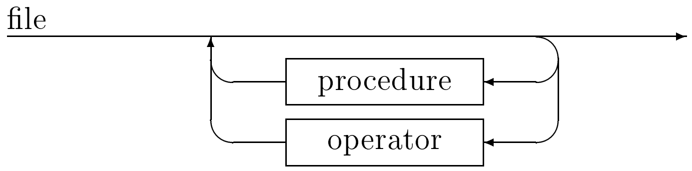

Figure 7.6: Syntax of a complete program

flexibility in variable declarations. The philosophy adopted is that everything happens in parallel. All loops are expanded. It is assumed that there is some external control device that calls the top level routine once every clock cycle.

Compared to standard C^{++}, the extensions are needed for the netlist compiler to be able to know the width of array elements. A regular C^{++} compiler can handle things differently by using dynamic objects, but a hardware component must have a completely static content.

7.2.1 Data Types

The only basic data type is the `bitvector`, and the only compound data type is one-dimensional arrays of `bitvector`. A `bitvector` is roughly equivalent to the standard `int` data type, but can also be viewed as an array of bits. A default `bitvector` contains 16 bits, but its size can be changed by a compiler switch and `bitvectors` of custom length can also be created by specifying the size explicitly to the constructor.

It would have been somewhat neater, and perhaps more intuitive, to have a bit as the basic data type and then declare `bitvector` as an array of bits. This approach would work for the netlist compiler, but unfortunately not for a regular C^{++} compiler (which is used for simulation purposes), mostly because C^{++} arrays contain no information about their length. Thus, if we consider the piece of code shown in figure 7.5, the add operator has no way of knowing the size of its operands.

The global syntax of a program source file is given in figure 7.6. It consists of the description of the hardware operators library and of one or more functions to be translated into a netlist. There are no global variables. The source file is preprocessed by a C preprocessor [Sta92] before being read by the `nlc` compiler.

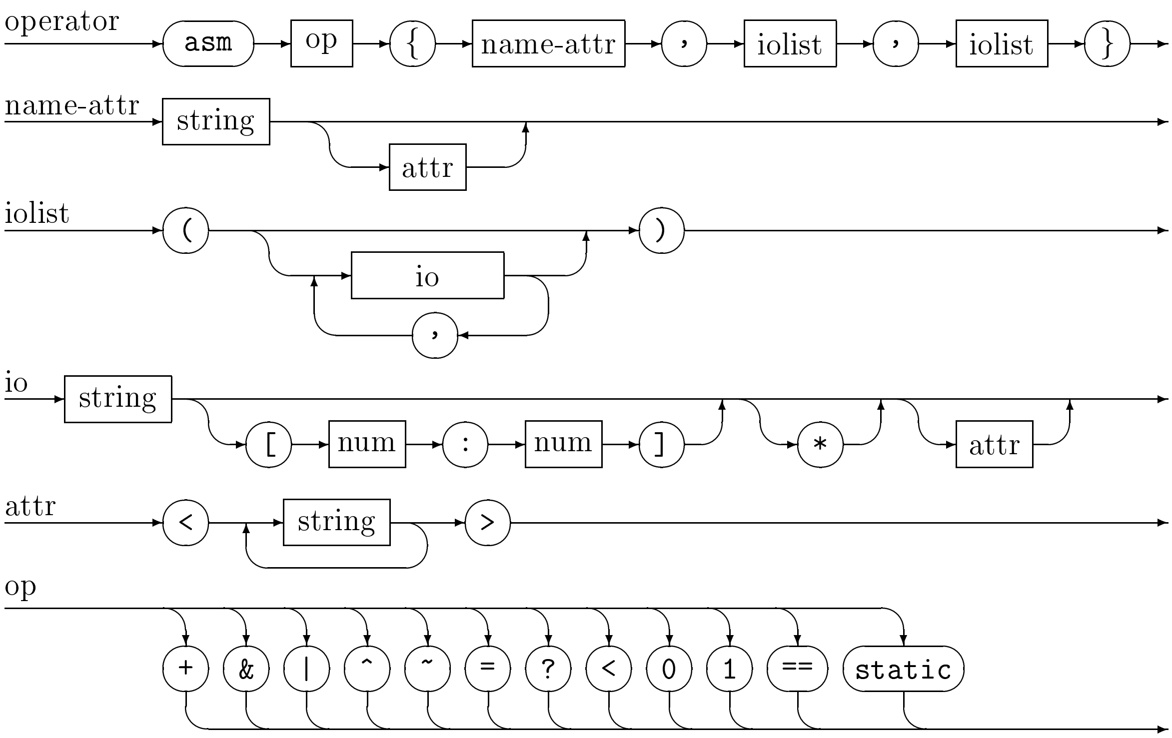

Figure 7.7: Syntax of an operator description

7.2.2 Hardware Operators Description

The hardware operators used by the compiler to generate the netlist must be described in the source file (or included by the source file). The syntax of the description is given in figure 7.7.

The description starts with the keyword `asm` and the symbol of the operator being described. The description itself consists of three parts: the name of the operator, as defined in the CAD software, followed by global attributes of the operator; the list of input ports of the operator; the list of output ports. The lists of input and output ports are composed of the names of these ports. Each port name can be followed by a list of attributes pertaining to that port alone.

The attributes of an operator have no special meaning to the `nlc` compiler, and are merely expanded and copied in the generated netlist file. Attributes are used by the simulators, the placement and routing tools, and the other CAD tools that work with the generated netlist files.

The `nlc` compiler knows how to generate netlists for the ViewLogic CAD tools (`.wir` files) [Vie95]. The `PINORDER` and `PINTYPE` attributes for these ViewLogic netlists are generated automatically by the compiler. Any other attribute must be specified in the operator description. It is unfortunately likely that the hardware operators library description will be dependent on both the FPGA family and the CAD tools used.

Since there are often global attributes that must be repeated for each of the inputs or outputs of an operator, the `nlc` compiler provides an automatic expansion facility

```
asm & {
  "x4000:and5" <"LEVEL=XILINX" "DEVICE=AND" "@%iDLY0=0"
                "@%iDLY1=0" "@%oDLY0=1NS" "@%oDLY1=1NS">,
  ("1", "2", "3", "4", "5"), ("o")
}

asm | {
  "x4000:or3" <"LEVEL=XILINX" "DEVICE=OR" "@%iDLY0=0"
               "@%iDLY1=0" "@%oDLY0=1NS" "@%oDLY1=1NS">,
  ("1", "2", "3"), ("o")
}

asm ^ {
  "x4000:xor4" <"LEVEL=XILINX" "DEVICE=XOR" "@%iDLY0=0"
                "@%iDLY1=0" "@%oDLY0=1NS" "@%oDLY1=1NS">,
  ("1", "2", "3", "4"), ("o")
}
```

Figure 7.8: Descriptions of a 5-inputs `and`, a 3-inputs `or`, and a 4-inputs `xor` gates

for these attributes. For example, the 3-input `|` operator of the XC4000 device family, shown in figure 7.8, has the global attributes `@1DLY0=0`, `@2DLY0=0` and `@3DLY0=0` (i.e., one for each of its inputs `1`, `2` and `3` respectively). The expansion mechanism allows to write a single global attribute `@%iDLY0=0` for the operator and the `%i` substring will be expanded once for each of the inputs of the operator. Similarly, a `%o` substring will be expanded once for each of the outputs. A single `%` character in the generated attribute can be obtained by writing `%%` in the source code.

Each recognized hardware operator will now be presented in more detail, with a sample description taken from the description of the XC4000 chips library. The complete description of the XC4000 operators library can be found in [Xil94], while an in-depth description of the theory explaining the inner working of these hardware operators can be found in [Man86] and [Wak90].

7.2.2.1 Basic Logic Gates

The `&`, `|` and `^` operators have a 1-bit output which is, respectively, the logical `and`, the logical `or`, and the logical `xor` of all their inputs. An example of description for these hardware operators is given in figure 7.8.

```
asm ~ {
  "x4000:inv" <"LEVEL=XILINX" "DEVICE=INV" "@%iDLY0=0"
              "@%iDLY1=0" "@%oDLY0=1NS" "@%oDLY1=1NS">,
  ("i"), ("o" <"PARAM=INV">)
}

asm = {
  "x4000:buf" <"LEVEL=XILINX" "DEVICE=BUF" "@%iDLY0=0"
              "@%iDLY1=0" "@%oDLY0=1NS" "@%oDLY1=1NS">,
  ("i"), ("o")
}
```

Figure 7.9: Descriptions of a 1-bit not and a 1-bit buffer hardware operators

7.2.2.2 Buffers and Inverters

The `~` and `=` operators have the same number of input and output bits. The outputs are, respectively, the logical not, and the buffered copy of the corresponding inputs. A buffer is used whenever a signal must have more than one name. An example of description for these hardware operators is given in figure 7.9.

7.2.2.3 Numerical Comparators

The `==` and `<` comparison operators take two sets of inputs of the same width. The `==` hardware operator is the equality comparator, which produces one bit of result which is true if the numbers represented by the two set of inputs are the same. The `<` hardware operator is the less than comparator, which produces one or two outputs (the second being optional). The first output is true if the number represented by the first set of inputs is less than the number represented by the second set of inputs. The optional second output is true if the first number is greater than the second (see section 7.3.4.6 for an explanation of its use). An example of description for these hardware operators is given in figure 7.10. The notations `"a"[15:0]` and `"b"[15:0]` in figure 7.10 indicate that the 16-bit inputs of the hardware operator are bus ports.

7.2.2.4 Multiplexers

The `?` multiplexer operator has three sets of inputs and one set of outputs. The first input is 1-bit wide and represents the control variable that selects among the other two sets of inputs: the second set of inputs is selected if the control variable is false, the third if the variable is true. The second and third sets of inputs must be the same width as the set of outputs. An example of description for the `?` hardware operators is given in figure 7.11.

```
asm == {
  "mx4000:comp16" <"LEVEL=MXILINX">,
  ("a"[15:0], "b"[15:0]), ("eq")
}

asm < {
  "mx4000:compm4" <"LEVEL=MXILINX">,
  ("a3", "a2", "a1", "a0", "b3", "b2", "b1", "b0"),
  ("lt", "gt")
}
```

Figure 7.10: Descriptions of a 16-bit equality and a 4-bit less than hardware operators

```
asm ? {
  "mx4000:m2-1" <"LEVEL=MXILINX">,
  ("se", "d0", "d1"), ("o")
}
```

Figure 7.11: Description of a two to one multiplexer

7.2.2.5 Registers

The `static` register operator takes three or four sets of inputs and has one set of outputs. The first set of inputs is the data to be stored in the register, the second input is the clock signal, and the third input is the clock enable signal. The optional fourth input is the clear signal. The set of outputs is the output of the register. An example of description for the `static` hardware operators is given in figure 7.12. In this example, the star (*) near the fourth input signal `"rd"` indicates that this signal is not used, a mechanism needed to be able to use symbols of the hardware components library that have more signals than needed by the `nlc` compiler.

7.2.2.6 Constant Values

The `0` and `1` constant operators have a single output which is the logical value 0 and 1 respectively. An example of description of these hardware operators is given in figure 7.13.

```
asm static {
  "mx4000:rd8" <"LEVEL=MXILINX">,
  ("d"[7:0], "c", "ce", "rd"*),
  ("q"[7:0])
}
```

Figure 7.12: Description of an 8-bit register

```
asm 0 {
  "x4000:gnd" <"LEVEL=XILINX">,
  ( ), ("o")
}

asm 1 {
  "x4000:vcc" <"LEVEL=XILINX">,
  ( ), ("o")
}
```

Figure 7.13: Descriptions of the logical values `0` and `1`

```
asm + {
  "mx4000:add2" <"LEVEL=MXILINX">,
  ("a1", "a0", "b1", "b0", "ci"),
  ("co", "s1", "s0")
}
```

Figure 7.14: Description of a 2-bit adder

7.2.2.7 Adders

The `+` hardware operator has three sets of inputs and two sets of outputs. The first and second sets of inputs are the two operands to be added and the third input is the carry in bit. The first output is the carry out bit and the second set of outputs is the result of the addition. The width of the input operands and the width of the result must be the same. An example of description for an adder is given in figure 7.14.

The `&`, `|`, `^`, `~`, `=`, `?`, and `static` operators are mandatory if they are used by the

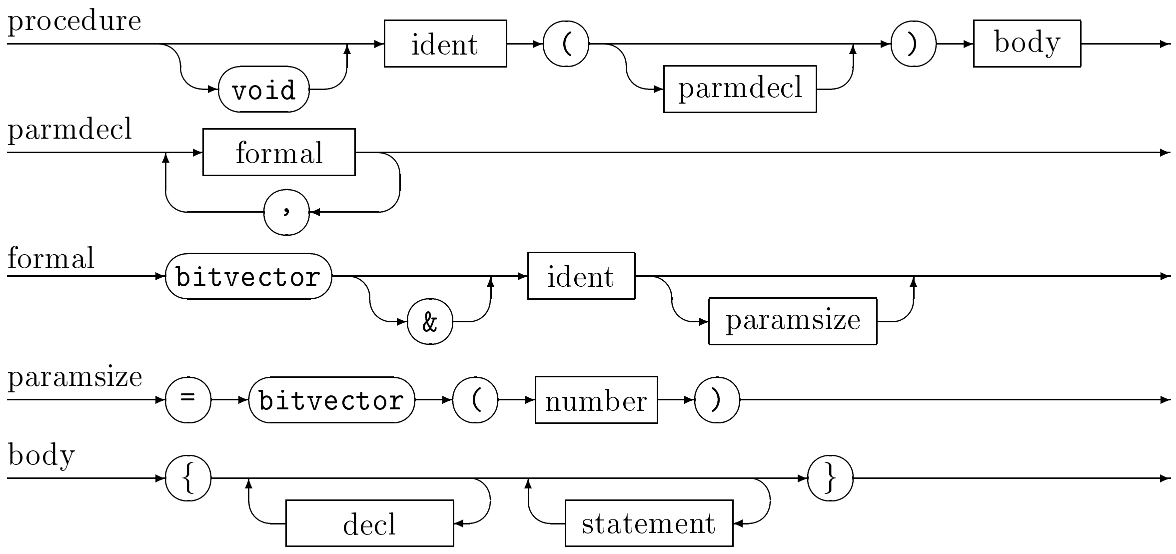

Figure 7.15: Syntax of a procedure description

```
void
foo(bitvector inA, bitvector inB = bitvector(6),
    bitvector &out = bitvector(8))
{
  // procedure body
}
```

Figure 7.16: Example of procedure declaration

source code being compiled. The others can be generated by the compiler using basic gates, as needed. If the `0` and `1` operators are not defined, the compiler will use the global signals `GND` and `VDD` of ViewLogic.

7.2.3 Procedure Description

The syntax of a procedure description is given in figure 7.15. The description starts with the optional keyword `void`, followed by the interface and the implementation of the procedure. The interface is composed of the name of the procedure and the set of input and output parameters, while the implementation is the body of the procedure. There are two kind of parameters: parameters passed by value and parameters passed by reference. Parameters passed by value are input parameters, while parameters passed by reference are output-only parameters, i.e., the output parameters do not have exactly the same semantics as C^{++} reference parameters, a feature that seldom causes problems. The body of a procedure is composed of variable declarations followed by statements.

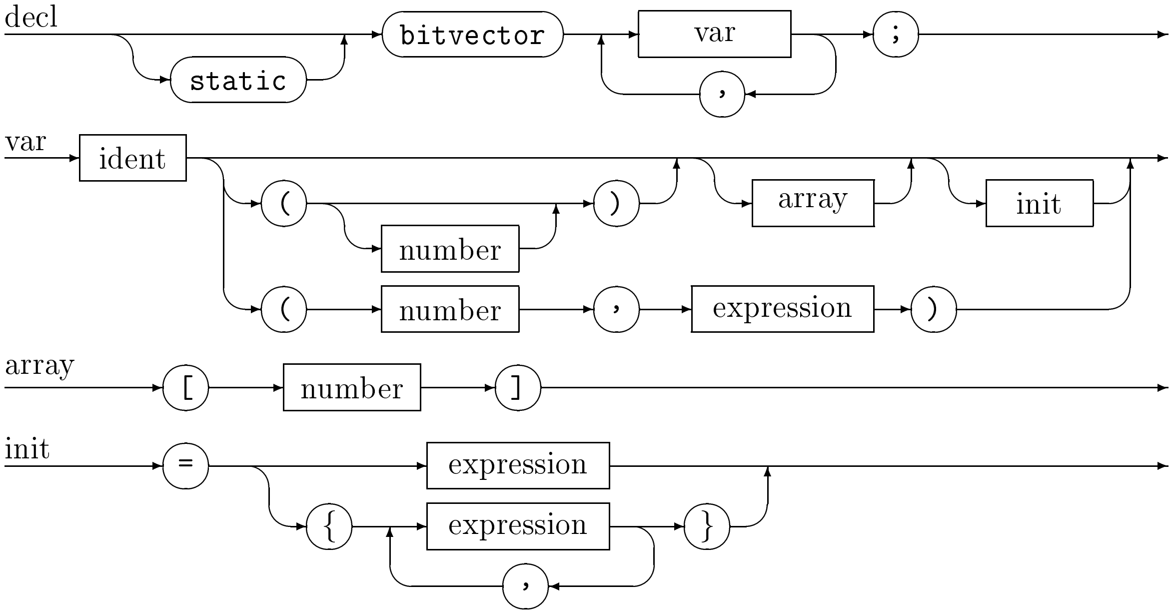

Figure 7.17: Syntax of the variable declarations

```
static bitvector la(17, 0), lc(17, 0);
bitvector nla(16, la);
bitvector nlc(16) = lc;
bitvector c(2)[18];
bitvector n;
```

Figure 7.18: Example of variable declarations

An example of procedure declaration is given in figure 7.16: the `inA` and `inB` parameters are respectively a 16-bit and a 6-bit input parameter, while `out` is an 8-bit output parameter. Procedures cannot return a value and thus are of type `void`. Note that the purpose of this seemingly strange way of specifying the width of the parameters is to assure that the declaration is accepted as-is by standard C^{++} compilers.

7.2.4 Variable Declarations

The syntax of variable declaration is given in figure 7.17. It is similar to the standard C^{++} syntax, but has a few extensions to allow more flexibility for initializers. A variable declaration starts with the optional keyword `static` followed by the keyword `bitvector` (the only valid data types are `bitvector` and array of `bitvector`). The variable declaration itself consists of its name and, optionally, its width and initial value. If the variable is an array, the name and optional width are followed by the size of the array

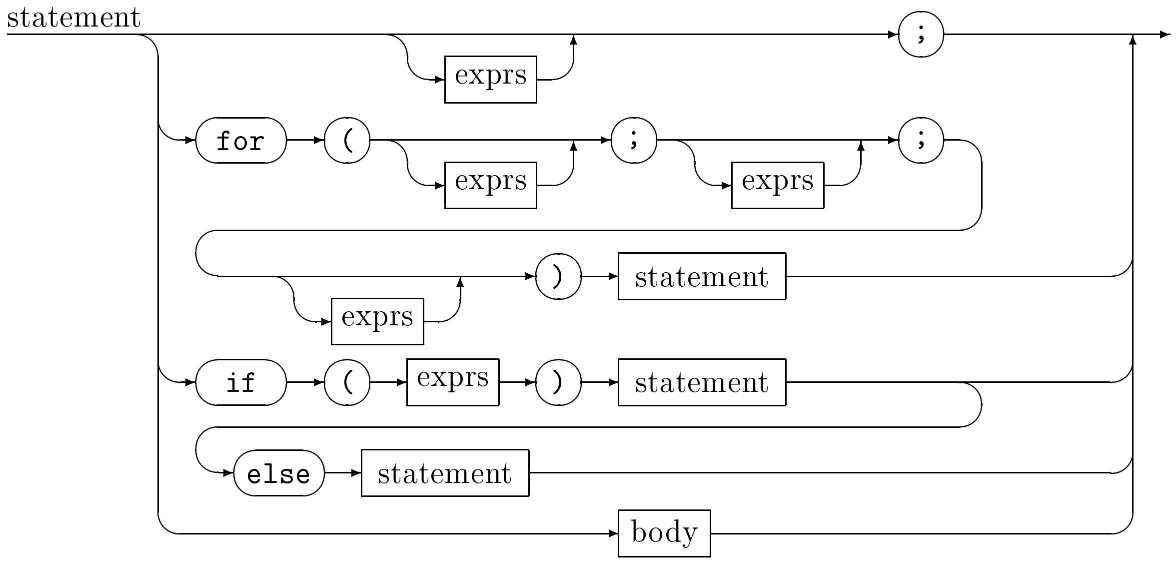

Figure 7.19: Statement syntax

between square brackets. If the width is not specified, the `nlc` compiler assigns either the default width of 16 or a width specified as an option when the compiler was called.

An example of variable declarations is given in figure 7.18. Line 1 declares two static `bitvector` variables 17-bit wide initialized to 0. Lines 2 and 3 declare two `bitvector` variables 16-bit wide: `nla`, initialized with the value of variable `la`, and `nlc`, initialized with the value of variable `lc`. The preferred way to declare a variable is that shown in lines 1 and 2, compatible with standard C++. Line 4 declares an array of 18 `bitvector` variables, each variable being 2-bit wide, and line 5 declares a `bitvector` variable of default width, i.e., 16-bit wide. Note that the declarations of lines 3 and 4 will not be accepted as-is by a standard C++ compiler (see section 6.5.1 on page 50).

Static variables are implemented using hardware registers. They keep their values from one call of the containing procedure to the next, as in standard C++. Static variables can only be initialized to 0, as the hardware registers implementing them will be set to 0 upon reset.

7.2.5 Statements

The statement syntax is given in figure 7.19. There are five kind of statements: empty statements, expressions, loop statements, conditional statements, and blocks.

There are two control flow constructs: `for` loops and `if` ... `then` ... `else` constructs. The `for` loop bounds must be known at compile time, since all loops must be statically unrolled. There are no `while` loops, but they can be easily implemented using `for` loops, as shown in figure 7.20.

```
bitvector i = 0;
for ( ; i < 3; ) {  // while (i < 3) {
  out[i] = in[i];
  i += 1;
}
```

Figure 7.20: Example of statements

7.2.6 Expressions

The expressions syntax is given in figures 7.21 and 7.22. A general expression can be composed of several expressions separated by commas. A simple expression is either a primary, a unary expression, a binary expression, or a ternary expression.

The operators obey the same precedence and associativity rules as in C and C++ [HSJ91, pp. 166–167]. All the available operators are given in figure 7.22.

A primary is either a number, an expression in parentheses, an array reference, a function call, or an identifier, optionally post-incremented or post-decremented.

7.3 The Compiler

The front-end of the `nlc` compiler is based on standard compilation techniques, as described in the Dragon book [ASU86], while the back-end is very different from a standard compiler, since it deals with wires instead of registers and generates a netlist instead of assembler instructions. The compiler is written in C++ with the help of two publicly available classes libraries: the GNU C++library [Lea92] and the LEDA library [NU95]. The C++ compiler used was the GNU gcc compiler [Sta95].

The global flow of operations of the `nlc` compiler is shown in figure 7.23. Each function is, in turn, parsed, analyzed, and transformed into a netlist, i.e., the compiler constructs the parse tree representing the complete function, processes this tree, and generates the corresponding netlist. The parse tree is then discarded and the process is repeated with the following functions. The processing is described in detail in the following sections.

7.3.1 Parsing and Tree Construction

A context-free grammar describing the syntax of `nlc` was developed, and a LALR parser [ASU86, pp. 195–266] for that grammar was constructed with the help of the parser generator Bison [DS95].

The parser performs three tasks: creation of the symbol table, creation of the list of available hardware components, and construction of a parse tree.

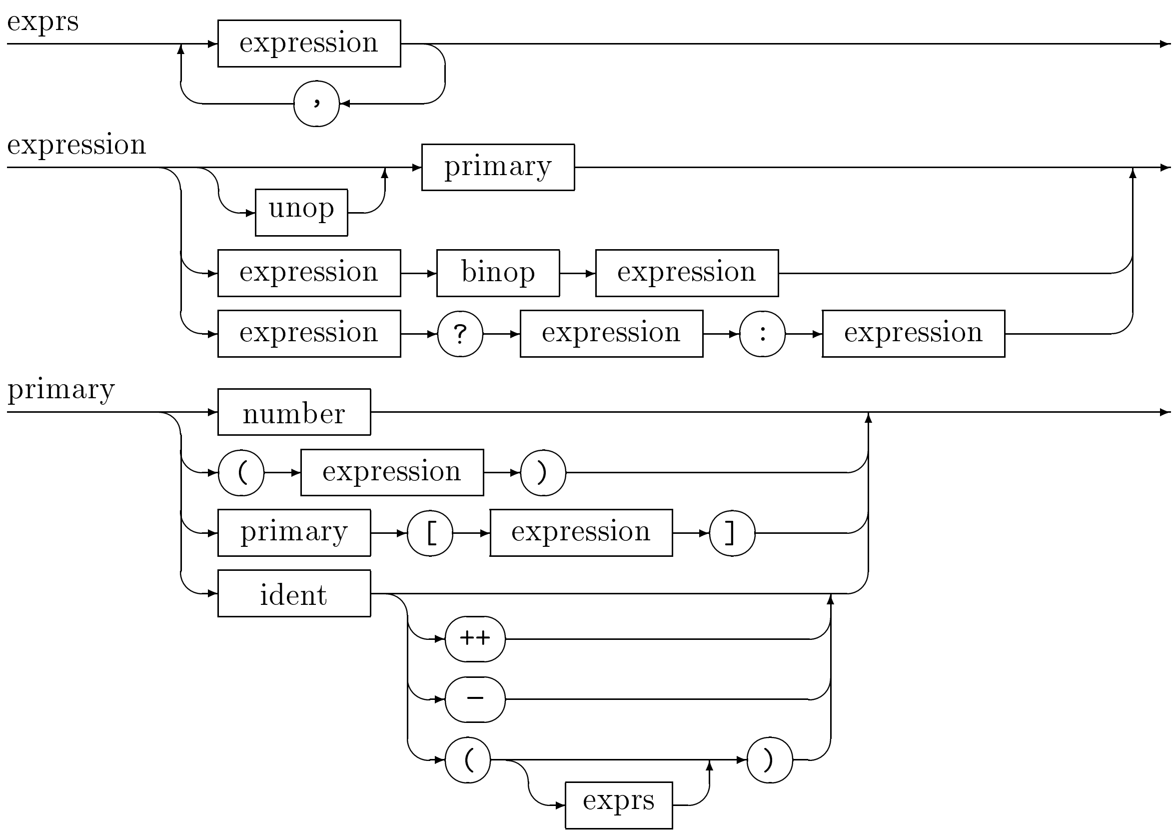

Figure 7.21: Expressions syntax

7.3.1.1 The Symbol Table

The symbol table is implemented using a LEDA dictionary, which associates a symbol name record (class `SNRec`) with the name of the symbol, and the symbol classes shown in figure 7.24.

The symbols are split into four categories: bitvector variables, handled by the `VSymRec` class; arrays of bitvector, handled by the `ASymRec` class; functions, handled by the `FSymRec` class; function inputs and outputs, handled by the `IOSymRec` class. These four classes are descendants of the general `SymRec` class.

7.3.1.2 Hardware Library Components

The hardware components defined in the source code are stored in objects of one of the subclasses of the `OpDescr` class shown in figure 7.25.

7.3.1.3 The Parse Tree

The parse tree is constructed using node objects belonging to the subclasses of the `TreeRec` class shown in figure 7.26. Each subclass corresponds to a specific language construct, and their purpose is described below. Each `TreeRec` node object has references

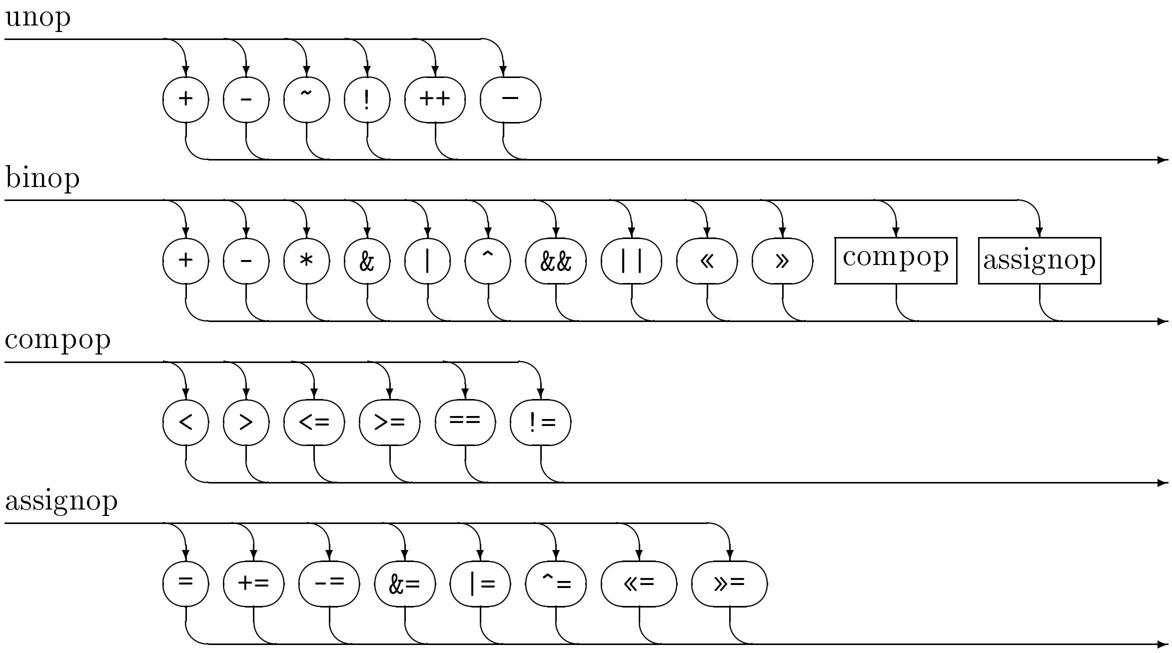

Figure 7.22: Available operations

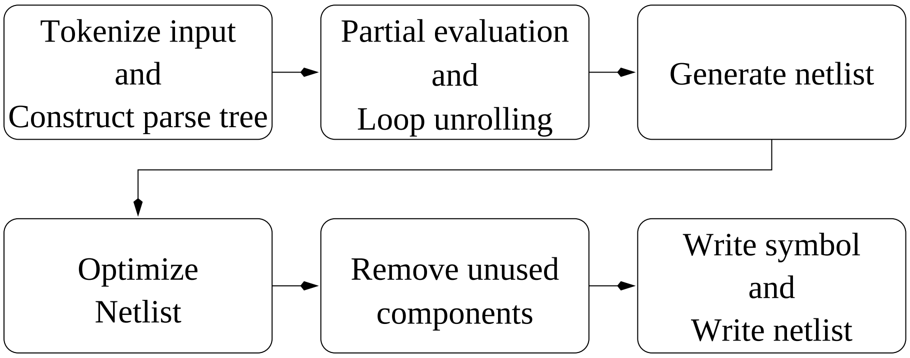

Figure 7.23: Compiler flow of operations

to its subtrees, if any, and also to its ancestor in the tree structure, i.e., the node of which it is a subtree.

7.3.1.4 Data Components

There are four data components: constants, which are represented by `ConstTreeRec`; accesses to array components, represented by `ArrayTreeRec`; accesses to a whole bitvector, represented by `VectorTreeRec`; accesses to a single bit of a bitvector, represented by `BitTreeRec`.

The `ConstTreeRec` simply contains the integer value of the constant it represents.

The `ArrayTreeRec` can represent both the access to a whole bitvector of the corresponding array, or the access to a single bit of a bitvector of the array. The array is

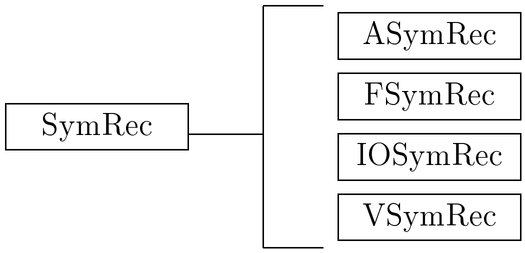

Figure 7.24: The symbol classes hierarchy

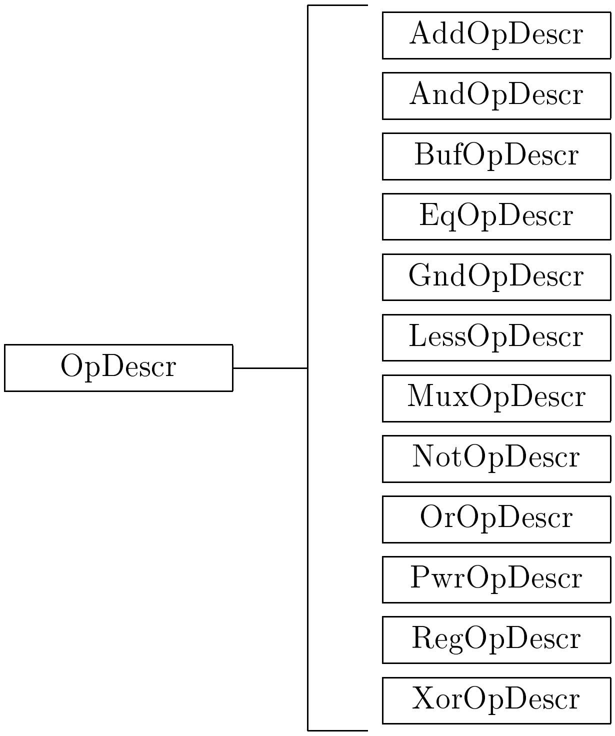

Figure 7.25: The operators description classes hierarchy

described by an `ASymRec` object. The index of the array element and the optional bit number, used if a single bit is referenced, are trees, where the latter tree is null if a whole bitvector is referenced.

The `VectorTreeRec` references a whole bitvector, described by a `VSymRec` object.

The `BitTreeRec` references a single bit from a bitvector. The bitvector is described by a `VSymRec` object and the bit number by a tree.

7.3.1.5 Operators

The operators are divided into three categories: the unary operators, which are descendants of the `UnaryTreeRec` class, as shown in figure 7.26; the binary associative operators, which are descendants of the `AssocTreeRec` class; the last category has no special

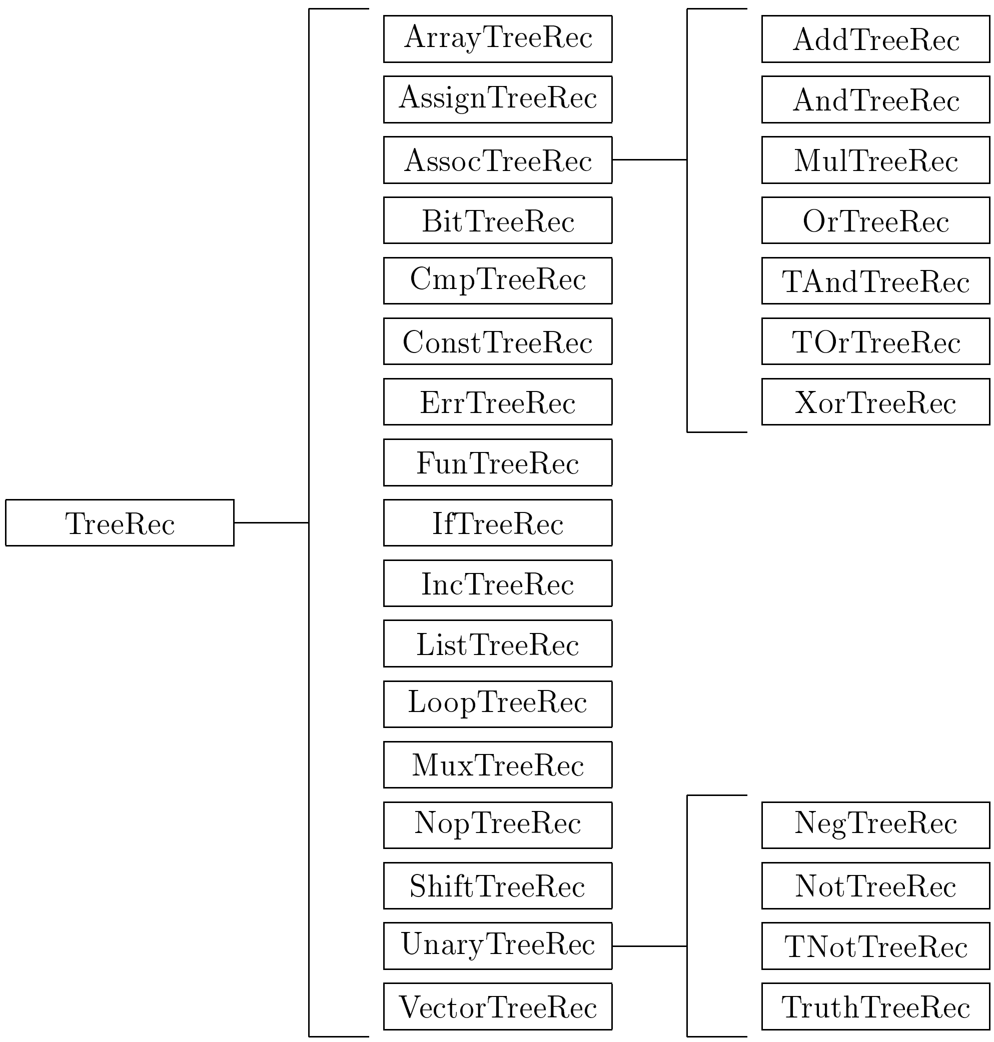

Figure 7.26: The parse tree classes hierarchy

property and its members, `CmpTreeRec`, `IncTreeRec`, `MuxTreeRec`, and `ShiftTreeRec` descend directly from the `TreeRec` class.

The `CmpTreeRec` represents the comparison, i.e., the equality, inequality, greater than, etc., of two values. These two values are represented by two trees, one left and one right. The result of the comparison is a Boolean value.

The `IncTreeRec` represents a general form of auto-increment of a value. The increment can either be a pre-increment (i.e., the value is incremented before being used) or a post-increment (i.e., the value is incremented after being used). The value to be incremented is represented by a `VSymRec` object, and the increment is a tree.

The `MuxTreeRec` is used to represent the C conditional operator `?`. The condition and both expressions are represented by trees.

The `ShiftTreeRec` is used to represent left and right shift operations. Both the left and right expressions are trees.

7.3.1.6 Unary Operators

The `UnaryTreeRec` class has four descendants. They are all unary operations and they apply to an expression that is represented by a tree.

The `NegTreeRec` implements the negation of an integer value, while the `NotTreeRec` implements the binary logical negation of a value.

The `TruthTreeRec` represents the truth value of an expression as defined by C, i.e., an expression is true if and only if its value is different from zero. Thus the value returned by a `TruthTreeRec` object is a Boolean value which is true if the expression referenced by the `TruthTreeRec` object is different from zero and false otherwise.

The `TNotTreeRec` represents the negation of a `TruthTreeRec`, i.e., it returns true if the expression referenced by the `TNotTreeRec` object is equal to zero.

7.3.1.7 Binary Associative Operators

The seven descendants of the `AssocTreeRec` class all represent binary associative operations. They operate on two or more expressions, which are represented by trees. These trees are kept in a list. If one of the expressions is a constant, it is kept as the first element of the list. If several expressions are constants, they are evaluated and replaced by the single constant value obtained by applying the operation implemented by the object to the constant expressions.

The `AddTreeRec` and `MulTreeRec` implement the arithmetic addition and multiplication respectively. Subtractions are implemented by adding the negated value of the second operand to the first, and thus are also implemented through `AddTreeRec` objects.

The `AndTreeRec`, `OrTreeRec`, and `XorTreeRec` implement the binary logical and, or, and xor operations respectively.

The `TAndTreeRec` and `TOrTreeRec` implement, respectively, the and and or operations of the truth value, as defined in section 7.3.1.6, of their child expressions.

7.3.1.8 Statements

There are four kind of statements: the assignment, the function call, the conditional statement, and the loop statement.

The assignment is implemented by the `AssignTreeRec` class. The right-hand side value is assigned to the left-hand side and both are trees.

The function call is implemented by the `FunTreeRec` class. The parameters of the call are described by a tree list.

Conditional statements are implemented by the `IfTreeRec` class. They contain three parts: the condition expression, the statement to be executed if the condition is true, and the statement to be executed if the condition is false. All three parts are represented by trees.

Loop statements are implemented by the `LoopTreeRec` class and contain four parts: the initializing expression, the looping condition expression, the increment expression, and the loop body. The three expressions are represented by trees, while the loop body is implemented by a tree list.

```
for (i = 0; i < 2; i++) {          i = 0;
  in_left[i] = in[i + 2];          in_left[0] = in[2];
  in_right[i] = in[i];       →     in_right[0] = in[0];
}                                  in_left[1] = in[3];
                                   in_right[1] = in[1];
```

Figure 7.27: Effects of `Expand` and `CleanupExpand` methods

7.3.1.9 Other Tree Components

The `ListTreeRec` object contains an ordered list of trees. It is used by other tree objects (e.g., the binary associative operators) to keep a list of expressions. It is also used to keep a list of statements. For example, the body of a function is represented by a `ListTreeRec` object.

The `NopTreeRec` object is used to represent a no-operation statement. For example, in a conditional statement with no `else` clause, the statement to be executed if the condition is false will be set to a `NopTreeRec` object.

The `ErrTreeRec` object is used when the parser detects a syntax error.

7.3.2 Partial Evaluation

During the second phase of the compilation process, the compiler unrolls all the loops contained in the function and evaluates all the values that can be computed at compile time. These transformations are performed by modifying the parse tree and are implemented by two methods of the `TreeRec` class: `Expand` and `CleanupExpand`. An example of these transformations is shown in figure 7.27.

The `Expand` method operates by interpreting the parse tree, computing all the values that can be statically determined, and expanding all the loops. The computed values are stored in the `sVal` component of the `TreeRec` object.

The `CleanupExpand` method goes through the parse tree to replace all subtrees which have a statically known value (i.e., whose `sVal` field was set to a defined value by the `Expand` method) with a `ConstTreeRec` object. Statements which have no side effects are deleted.

7.3.3 Preparations for Hardware Synthesis

The third phase of the compilation process consists of generating the netlist from the modified parse tree. The internal representation of the netlist is made up of hardware operators objects belonging to the descendents of the `OpRec` class shown in figure 7.28.

Each hardware operator has one or more output busses and zero or more input busses. A bus is composed of one or more wires. The output busses are implemented by objects of class `BusOutRec`, which define the name of the bus, its width, its use count, and have a reference to the hardware operator that owns or drives the bus. Each bit

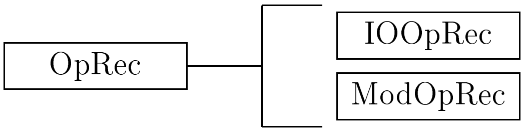

Figure 7.28: The modules classes hierarchy

of the output bus is further implemented by an object of class `BitOutRec`. The input busses are implemented by objects of class `BusInRec`: for each bit of the input bus, an object of class `BusInRec` simply references the corresponding `BitOutRec` of a particular `BusOutRec` (i.e., the one that drives the wire).

There are two subclasses of the `OpRec` class: the `IOOpRec` class implements the inputs and outputs of a function and the `ModOpRec` class implements all the hardware operators used for the internals of the function. In other words, the `IOOpRec` objects contain a single `BusOutRec` object which is the corresponding input or output port of the function. The `ModOpRec` objects contain at least one `BusInRec` object and one `BusOutRec` object.

The purpose of the hardware synthesis phase of the compiler is to translate the parse tree into a set of `ModOpRec` objects and to wire them together and to the input and output ports of the function. This process is divided into several steps which are described in detail in the following sections.

7.3.3.1 Expression Width Computation

The first step of the synthesis is to compute the width of all the expressions by calling the `ComputeWidth` method of the `TreeRec` class. The width of bitvector references is derived directly from the bitvector variable declaration, but the width of add or shift expressions must be computed from the width of the operands (e.g., the addition of two 4-bit wide operands can produce a 5-bit result).

The compiler must know the width of each expression node to choose a corresponding hardware operator.

7.3.3.2 Variables Initialization

The second step of the synthesis is to initialize the state of all the variables used by the current function. The current state of a variable is defined by its `curValue` field, which is an array of pointers to `BitOutRec` objects. There are four kinds of variables: the input parameters of the function, the output parameters, the static local variables, and the other local variables.

The initialization of the input and output parameters is handled by the `GenerateIO` method from the `VSymRec` class. This method is called for all the function parameters. It generates an `IOOpRec` object for each input parameter and also sets the `curValue` of the `VSymRec` objects to point to the corresponding bits of the `IOOpRec` objects. The `curValue` of output parameters is set to null (i.e., it is undefined).

The initialization of local variables is handled by the `GenerateLocals` method from the `VSymRec` class. This method is called for all the local variables of the function. For each static local variable, a corresponding hardware register is created and the `curValue` of the variable is set to the output of the register. The compiler needs the description of at least one kind of hardware register with clock enable to be able to synthesize static variables. The `curValue` of the other local variables is set to null.

7.3.4 Hardware Synthesis

The hardware synthesis of the function body is implemented by the `Generate` method of the `TreeRec` class and its subclasses. The basic approach is to walk through the tree, generating hardware operators corresponding to the traversed nodes, setting the input of the hardware operators to reference the corresponding operands, and setting the affected variables to reference the output of the operators. To implement this approach, each `TreeRec` node has a `val` field that references the `BitOutRec` objects that define its value, i.e., the `val` field is set to the output of the hardware operator that computes the value of the node.

7.3.4.1 Hardware Operator Selection

Since there may be more than one hardware operator of a certain kind (e.g., there can exist 2, 3 and 4-input and gates), the compiler has to choose which particular hardware operator to use for each case. The general rule is to use the smallest operator that has enough inputs. For example, if there was a choice of 1, 2, 4, and 8-bit adders and the compiler has to synthesize the addition of two 3-bit values, the chosen operator would be the 4-bit adder since it is the smallest of the available adders that can fit the 3-bit values. The seemingly wasted material will be automatically removed by the tools that perform the placement and routing for the FPGA chip. Considering the same example, if only 1 and 2-bit adders were available, the compiler would use one 2-bit adder and one 1-bit adder to implement the same operation.

The following sections describe in more detail the particular synthesis method used for each kind of node.

7.3.4.2 Shift Operators Synthesis

The shift operators, described by `ShiftTreeRec` objects, are the easiest to synthesize, simply becoming wires.

7.3.4.3 Simple Logic Gates Synthesis

The logical not operator, described by `NotTreeRec` nodes, is implemented using hardware inverters described by objects of class `NotOpDescr`. At least one hardware inverter must be available to the compiler for not operators to be used.

Objects of classes `AndTreeRec`, `OrTreeRec`, and `XorTreeRec` describe logical and, or, and xor operators respectively. They are implemented using hardware and gates, described by objects of class `AndOpDescr`, or gates, described by class `OrOpDescr`, and xor gates, described by class `XorOpDescr`. At least one hardware gate of each kind must be available to the compiler for the corresponding operators to be used.

7.3.4.4 Truth Conditions Synthesis

The truth of an expression is obtained by or-ing all of its bits. Thus, `TruthTreeRec` objects are implemented using or gates. `TNotTreeRec` are implemented by taking the truth value of the subtree and negating it with an inverter.

`TAndTreeRec` and `TOrTreeRec` are implemented by taking the truth value of all their subtrees and than and-ing, respectively or-ing, these truth values together.

The same conditions of library hardware operators availability explained in the previous section for the simple logic gates synthesis apply.

7.3.4.5 Arithmetic Expressions Synthesis

Additions and subtractions, described by `AddTreeRec` nodes, are implemented using hardware adders, if available, and simple logic gates. The principle of the synthesis is to divide the subtrees to be added in two groups, the positive and the negative, to generate the additions for each group and then subtract the negative result from the positive result. For example, the add expression $a + (-b) + c + (-d)$ is split into the two groups $G_+ = a + c$ and $G_- = b + d$ and the result computed as $G_+ - G_-$.

The add operator itself is synthesized either using hardware adders described by objects of the `AddOpRec` class, if available, or by using and, or, and xor logic gates. The subtraction of the negative group value from the positive group value is implemented by inverting all the bits of the value of the negative group and adding them to the value of the positive group, with the carry-in bit of the adder set to one.

Arithmetic negations, represented by `NegTreeRec` objects, are implemented by inverting all the bits of the value using hardware inverters and then adding one with an adder.

Multiply and divide operators have not been implemented.

7.3.4.6 Arithmetic Comparisons Synthesis

There are six possible arithmetic comparisons of two expressions a and b, described by `CmpTreeRec` nodes. All six are implemented using equality comparators, less than comparators, and not gates, as shown in table 7.1.

The first two comparisons are equality comparisons and are handled using an equality hardware operator and a not gate, if needed. The compiler uses hardware equality comparators described by objects of the `EqOpDescr` class, if available. If no such library components are available, the compiler constructs them using simple xor and or gates.

The last four comparisons are implemented using an hardware less than comparator and a not gate, if needed. The compiler uses hardware less than comparators described by objects of the `LessOpDescr` class, if available. If no such library components are available, the compiler constructs them using simple gates.

The compiler is able to use two different kind of hardware less than comparators: the first kind has a single output which is true if the first input value is less than the

Table 7.1: Implementation of arithmetic comparisons

Source	Implementation
$a = b$	$a = b$
$a \neq b$	$\mathsf{not}(a = b)$
$a < b$	$a < b$
$a \leq b$	$\mathsf{not}(b < a)$
$a > b$	$b < a$
$a \geq b$	$\mathsf{not}(a < b)$

second input value; the second kind has two outputs, one which is true when the first input value is less than the second input value, and the other when the first input value is greater than the second input value. The compiler uses this second output when it has to build bigger comparators using smaller ones, in which case it applies the relation given by equation 7.1. When this second output is not available, the compiler uses a second comparator to compute $a_{i\ldots j} > b_{i\ldots j}$.

$$(a_{i\ldots 0} < b_{i\ldots 0}) \Rightarrow (a_{i\ldots j} < b_{i\ldots j} \text{ or } (\mathsf{not}(a_{i\ldots j} > b_{i\ldots j}) \text{ and } a_{(j-1)\ldots 0} < b_{(j-1)\ldots 0})) \tag{7.1}$$

7.3.4.7 Conditional Expression Synthesis

Conditional expressions, described by nodes of class `MuxTreeRec`, are implemented using multiplexers, described by objects of the `MuxOpDescr` class. At least one hardware multiplexer must be available to the compiler for this operator to be used.

7.3.4.8 Conditional Statement Synthesis

The conditional statements are the most complicated to implement. The basic approach is to determine which variables are modified by the conditional statement and what is the final value of those variables, according to the value of the condition.

The method used to implement this approach is to keep a snapshot of the state of the variables before the conditional statement is entered. Then the compiler generates the statements that are to be executed when the condition is true, at which point a second snapshot of the state of the variables is taken representing the resulting state of the variables when the condition is true. The state of the variables is then restored to the value it had before entering the conditional statement and the compiler proceeds by generating the statements that are to be executed when the condition is false, at which point a third snapshot of the variables state is taken. The state of the variables after the conditional statement is then obtained using multiplexers, controlled by the condition value, which choose the correct value among those of the various snapshots. An example of conditional statement generation is shown in figure 7.29.

```
bitvector s(4);
bitvector c(1);
bitvector n(1);

if (s == 3)
   n = 1;
else if (s == 4)
   n = c;
else
   n = 0;
```

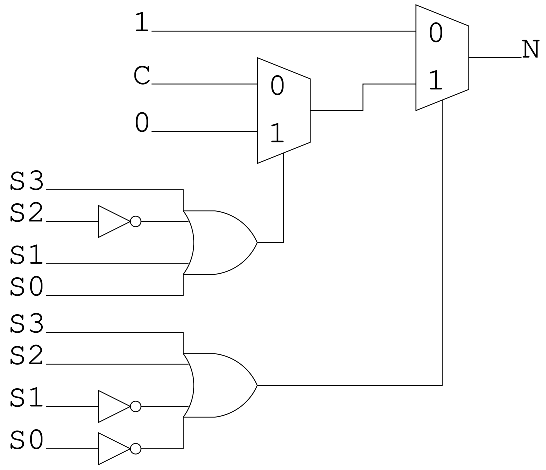

Figure 7.29: Implementation of a conditional statement

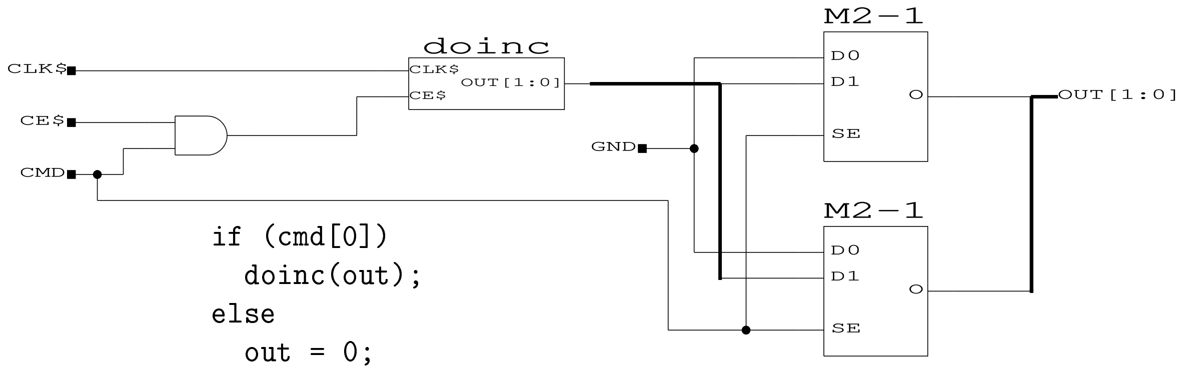

Figure 7.30: Synthesis of a function call

7.3.4.9 Function Call Synthesis

Since functions are transformed into hardware operators, a function call is transformed into the instantiation of the corresponding hardware operator.

Special care has to be taken when the function is called inside a conditional statement, since if the function called contains static variables, the corresponding registers might be incorrectly modified. Therefore, the compiler passes the condition that originates to the call to the called function: if the condition is false, the called function must avoid modifying its static variables. An example of function call synthesis is given in figure 7.30.

7.3.5 Feedback of Static Variables

The last value that was assigned to a static variable is the one that is fed to the register that implements it. This process is implemented by the `FeedbackStatic` method of the `VSymRec` class.

Since a function can be called from inside a conditional statement, care has to be

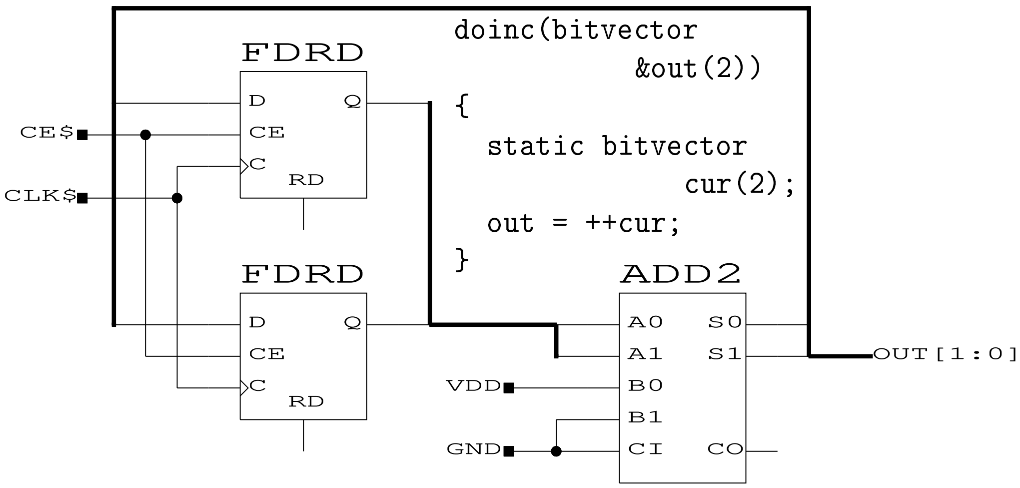

Figure 7.31: Feedback of the static variables

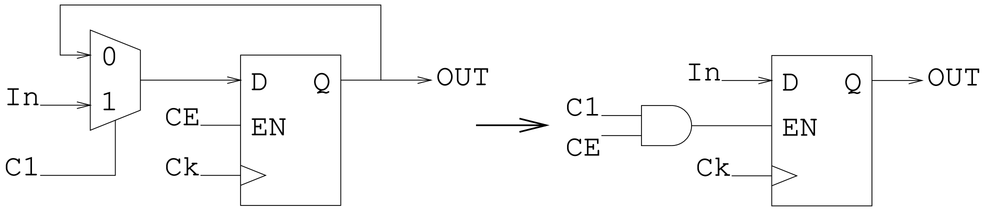

Figure 7.32: Multiplexer optimization

taken not to modify the content of these static variables if the condition under which the function is called is not true, as explained in section 7.3.4.9. This is the reason why each function that either contains static variables or calls a function that contains static variables has a clock enable input `CE$` which must be tied to the clock enable input of the hardware registers that implement the static variables, as shown in figure 7.31.

7.3.6 Hardware Optimizations

Two kind of optimizations are performed while the compiler is generating the netlist. The first is to avoid generating multiplexers which have twice the same input. The second handles additions of the form $x + C$, where $C = n \cdot 2^i$, $i > 0$ (i.e., C is a constant and its least significant bits are zero), in which case the addition can be optimized by computing $y+n$, where y represents the appropriate most significant bits of x and simply passing through the least significant bits of x.

Further optimizations are performed only if the user explicitely requests them. They are implemented by the `Optimize` method of the `OpDescr` class and its subclasses. There are two implemented optimizations.

The first optimization tries to remove unnecessary chains of inverters: when an input is driven by two inverters in a row, the input is redirected to the output that drives the first inverter.

The second optimization tries to remove unnecessary multiplexers, a condition which occurs mainly when a register is fed by a multiplexer, in which case the compiler tries to replace the multiplexer by commanding the clock enable input of the register, as shown in figure 7.32.

7.3.7 Output Wiring and Netlist Cleanup

The last stage of the netlist generation process is to wire the outputs of the function: the last value that was assigned to the output parameters of the function is wired to the output ports. This process is performed by the `WireOutput` method of the `VSymRec` class.

The compiler then removes unnecessary hardware components from the netlist, so as to remove all modules whose outputs are unused. This process is repeated as long as new modules can be deleted. Such unused modules usually result from the optimizations described in section 7.3.6.

7.3.8 Symbol and Netlist Output

Once the netlist is complete, the compiler writes two files: the first one describes the symbol of the generated function. This symbol describes the interface of the generated hardware operator and is used by the CAD tools. It is written by calling the `OutputSym` method of the `OpDescr` class.

The second file is the netlist itself, written by calling the `OutputWir` method of the `OpDescr`, `ModOpRec`, and `IOOpRec` classes.

7.3.9 Concluding Remarks

There are still some limitations to the current implementation. Some operators like multiply and divide are unimplemented. The mechanism for the library description could be generalized to allow more flexibility. The compiler could handle more cases of undefined operators and more cases of addition and comparison operators. The generated netlist could be better optimized (e.g., some information about the speed and size of the different library components could be added to their description, thus enabling the compiler to choose a fast or small implementation).

7.4 Comparison with VHDL

To compare the `nlc` compiler with current VHDL hardware synthesizers, the execution unit computing the "game of life" cellular automaton [BCG82] has been implemented in both C++ and VHDL: the C++ code can be found in figures 6.6, 6.7, and 6.13 of chapter 6 and the VHDL code is given in figures 7.33 and 7.34. The results of synthesizing this execution unit, for a Xilinx XC4000 chip, with `nlc`, Exemplar, and Synopsis are presented in table 7.2. It should be noted that the VHDL code is probably not written in an optimal way for hardware synthesis. However, it is a simple translation of the C++ code and probably resembles what a C programmer would have written in a first attempt.

Table 7.2: Summary of synthesized Life execution unit for a Xilinx XC4000 chip

Compiler	Language	Optimize	CLBs	Delay	Run time	Machine
`nlc`	C++	no	172	63 ns	00:00:01	SPARCstation 5
`nlc`	C++	yes	131	63 ns	00:00:01	SPARCstation 5
Exemplar	VHDL	Pass 5	145	86 ns	00:02:07	SPARCstation 5
Exemplar	VHDL	Pass 7	148	70 ns	00:02:26	SPARCstation 5
Synopsis	VHDL	Medium	71	63 ns	00:55:00	HP 700
Synopsis	VHDL	Hard	60	58 ns	01:05:00	HP 700

As can be seen in table 7.2, `nlc` gives slightly better results than Exemplar, but uses about twice as much configurable logic blocks (CLBs) than Synopsis. However, as far as compile time is concerned, `nlc` is the clear winner, part of the speed gain being explained by the lack of Boolean minimization tools in `nlc`. It can also be noted that if area or delay minimization are really important, the output of `nlc` can be fed to a tool like Synopsis for stronger optimization.

To obtain a working circuit, the output of `nlc` (or Exemplar or Synopsis) must be compiled using the chip vendor's tools. In this case, using Xilinx XC4000 circuits, this takes roughly 4 minutes on a SPARCstation 5 running at 85 MHz. This means that the time needed for the complete compilation process to get a working circuit, i.e., a running custom processor, is in the same order of magnitude as the time needed to compile a purely software program.

The reasonable compilation time and the fact that a single language can be used to describe a complete application are the main reasons which lead us to conclude that a C++ to hardware compiler is a tool that could have a bright future and help promote FPGAs as custom computing machines.

```
library synth;
use synth.std_logic_1164.all;
use synth.exemplar_1164.all;
entity life is
  port (
    clk  : in  std_logic;
    ain  : in  std_logic_vector(15 downto 0);
    bin  : in  std_logic_vector(15 downto 0);
    aout : out std_logic_vector(15 downto 0);
    bout : out std_logic_vector(15 downto 0);
    cmd  : in  std_logic_vector(20 downto 0)  );
end life;
architecture behavior of life is
  type intarray is array(17 downto 0) of integer range 0 to 3;
  signal la  : std_logic_vector(16 downto 0);
  signal lb  : std_logic_vector(16 downto 0);
  signal lc  : std_logic_vector(16 downto 0);
  signal nla : std_logic_vector(15 downto 0);
  signal nlc : std_logic_vector(15 downto 0);
  signal n   : std_logic_vector(15 downto 0);
  function sumcol(l1, l2, l3 : in std_logic) return integer is
    variable i : integer range 0 to 3;
  begin
    if (l1 = '1') then i := 1; else i := 0; end if;
    if (l2 = '1') then i := i + 1; end if;
    if (l3 = '1') then i := i + 1; end if;
    return i;
  end sumcol;
  function nextstate(c1, c2, c3 : in integer;
                     c : std_logic) return std_logic is
    variable sum : integer range 0 to 9;
    variable n : std_logic;
  begin
    sum := c1 + c2 + c3;
    if (sum = 3) then n := '1';
    elsif (sum = 4) then n := c;
    else n := '0';
    end if;
    return n;
  end;
```

Figure 7.33: Life execution unit written in VHDL, part1

```
  alias ph0 : std_logic is cmd(0);
  alias ph1 : std_logic is cmd(1);

begin

  seq : process
  begin
    wait on clk, bin, cmd, nla, nlc, la, lb, lc;
    if (clk'event and clk = '1') then
      if (ph1 = '1') then
        la(16) <= la(0);
        lb(16) <= lb(0);
        lc(16) <= lc(0);
        la(15 downto 0) <= nla(15 downto 0);
        lb(15 downto 0) <= bin(15 downto 0);
        lc(15 downto 0) <= nlc(15 downto 0);
      elsif (ph0 = '1') then
        nla <= ain;
        nlc <= bin;
      end if;
    end if;
  end process;

  aout <= bin when (ph0 = '1') else
          n;

  compute : process
    variable c : intarray;
  begin
    wait on nla, bin, nlc, la, lb, lc;
    c(0) := sumcol(nla(15), bin(15), nlc(15));
    for i in 0 to 16 loop
      c(i+1) := sumcol(la(i), lb(i), lc(i));
    end loop;
    for i in 0 to 15 loop
      n(i) <= nextstate(c(i), c(i+1), c(i+2), lb(i));
    end loop;
  end process;

end behavior;
```

Figure 7.34: Life execution unit written in VHDL, part2

Chapter 8

The VLIW Compiler mcc

The purpose of the VLIW compiler is to produce the control program responsible for feeding data to the execution units of the Spyder processor and storing the computed results back in its main data memory. Another task of the compiler is to assign the hardware operators to the available execution units and then produce the source code defining the entire contents of each of the execution units.

This chapter will present a brief example of control program, followed by a description of the source language accepted by the `mcc` compiler and a presentation of the algorithms used by the compiler to transform this source language into Spyder instructions.

8.1 Example of Control Program

The goal of this very simple example program is to compute the average of two numbers stored in the Spyder data memory and store back the result of the computation. For this example, it will be assumed that a hardware operator computing the average of two numbers is available.

The source code for the program is given in figure 8.1. Line 1 declares the `average` hardware operator, specifying that it takes two inputs, labeled `a` and `b`, and produces one output, labeled `f`. Lines 2 to 7 define the contents of the Spyder data memory, starting from address 0 and moving upward: in this example, the first two locations of the memory contain the two values of which we wish to compute the average and the result will be stored in the third location. Lines 8 to 14 define the control program, whose main routine must be named `MCmain`: in this example `MCmain` simply calls the `average` operator with the two values contained in the memory (line 12) and stores the result back in the memory (line 13).

The `mcc` compiler does not make a distinction between the `bitvector` data type and the `short` data type. For a standard C^{++} compiler, however, these types are entirely different, which is why the temporary variable `res` is used to store the result of the hardware operator, before it is moved into the memory.

```
extern void average(bitvector a, bitvector b, bitvector &f);
struct memory {
  short n1;
  short n2;
  short avg;
};
extern struct memory *memory;
void
MCmain()
{
  bitvector res;
  average(memory->n1, memory->n2, res);
  memory->avg = res;
}
```

Figure 8.1: Example of program

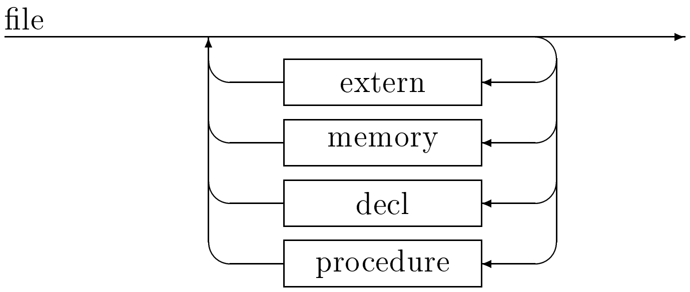

Figure 8.2: Syntax of a complete program

8.2 The Language

The language chosen is roughly the same subset of C^{++} as the one defined for `nlc` in section 7.2. There is no support for classes and inheritance, no pointers, and the array type is limited to one-dimensional arrays.

There is a single data type available, the 16-bit integer. It can be defined both as `short` and `bitvector`. The two definitions are completely equivalent as far as `mcc` is concerned. However, distinctions are made by standard C^{++} compilers and these two definitions were required for a standard C^{++} compiler to be able to compile the programs.

The global syntax of a program source file is given in figure 8.2. It consists of the description of the available hardware operators, of the description of the contents of the memory, of the global variables, and of the procedures. The source file is preprocessed by a C preprocessor [Sta92] before being read by the `mcc` compiler proper.

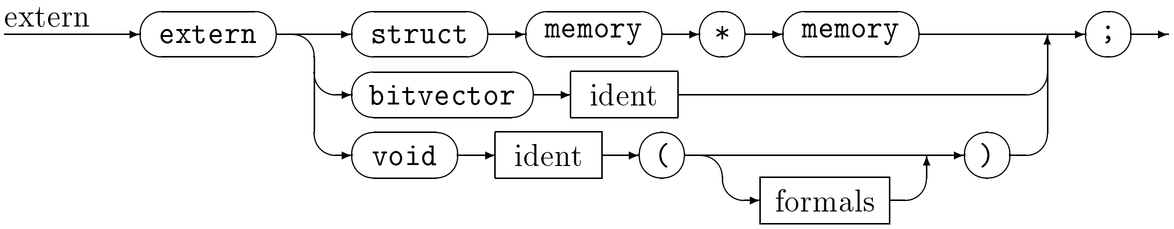

Figure 8.3: Extern declarations

```
extern void life(bitvector top, bitvector mid,
                 bitvector bot, bitvector &f);
```

Figure 8.4: Example of hardware operator description

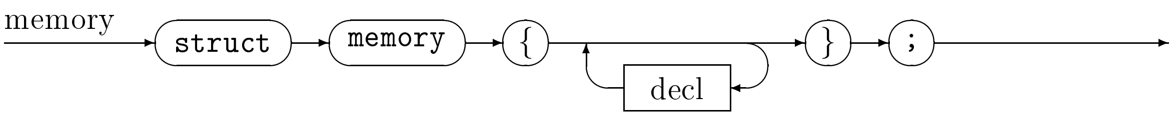

Figure 8.5: Memory content definition

8.2.1 Hardware Operators Definition

The hardware operators used by the program must be defined using the syntax given in figure 8.3, which consists of the name of the operator, along with its input and output parameters. The hardware operators appear like a function returning `void`, i.e., returning no value. Although it is of no consequence to `mcc`, the parameters should be of type `bitvector` to conform to the C++ conventions followed in the rest of the system.

Figure 8.4 gives an example of hardware operator description: the figure defines an operator named `life` that takes three inputs, named `top`, `mid`, and `bot`, and produces one output named `f`.

The other `extern` declarations shown in figure 8.3 allow the definition of the Spyder memory as an external element of the C++ program and the definition of an external `bitvector` variable named `dummy`. These declarations are necessary only for the sake of conventional C++ compilers. The purpose of the `dummy` variable will be explained later.

8.2.2 Memory Content Definition

The contents of the Spyder data memory must be described to the `mcc` compiler. The syntax of this description, which appears as a conventional C structure definition, is shown in figure 8.5. The memory can contain values of type `short` and arrays of `short` values. The description defines the content of the memory starting from address zero

```
1 struct memory {
2   short width;
3   short height;
4   short board[0];
5 };
```

Figure 8.6: Example of memory content definition

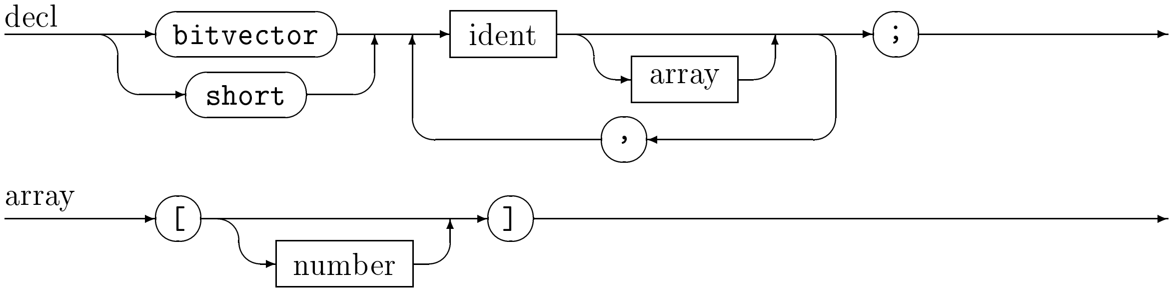

Figure 8.7: Syntax of the variable declarations

```
1 bitvector res;
2 short i, j;
3 short buf[64];
```

Figure 8.8: Example of variable declarations

and moving upward. The last element of the description can be an array of unspecified size.

Figure 8.6 shows an example of memory content definition: line 2 specifies that the address 0 of the memory contains the data named `width`; line 3 specifies that address 1 contains `height`; line 4 specifies that the rest of the memory is occupied by the array named `board`, which is of unspecified size.

8.2.3 Variable Declaration

The syntax of a variable declaration is shown in figure 8.7. The `mcc` compiler allows global variables and variables local to a procedure. Variables are stored in the registers of the Spyder processor and are never automatically assigned to a memory location.

An example of variable declarations is given in figure 8.8. Line 1 declares a variable labeled `res` and line 2 declares two variables labeled `i` and `j` respectively. The `mcc`

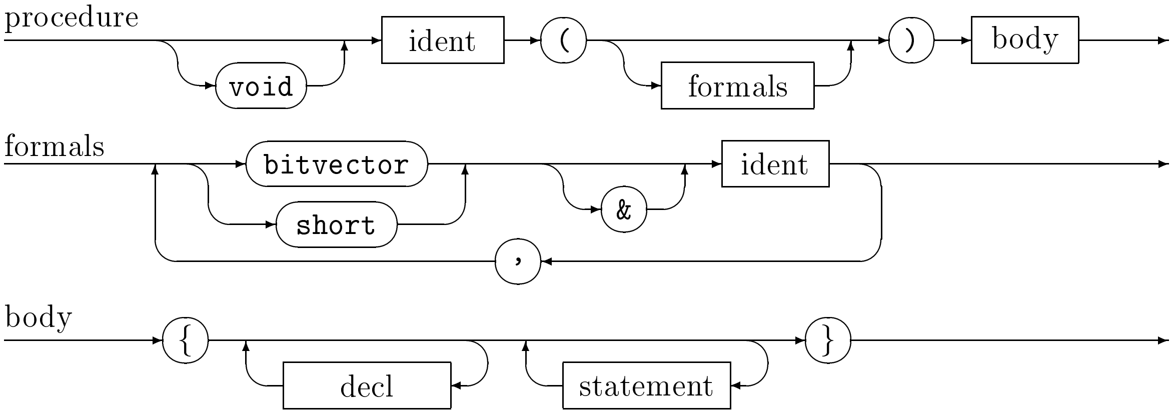

Figure 8.9: Syntax of a procedure description

```
void
foo(short inA, short &out)
{
  // procedure body
}
```

Figure 8.10: Example of procedure declaration

compiler makes no distinction of type between these three variables, but a conventional C++ compiler does. Line 3 declares an array of 64 values labeled `buf`.

8.2.4 Procedure Definition

The syntax of a procedure description is given in figure 8.9. The description starts with the optional keyword `void`, followed by the name, the formal parameters, and the body of the procedure. There are two kind of parameters: parameters passed by value and parameters passed by reference. Parameters passed by value are input parameters, while parameters passed by reference are output-only parameters, i.e., the output parameters do not have exactly the same semantics as C++ reference parameters, a difference which seldom causes problems. The body of a procedure is composed of variable declarations followed by statements.

An example of procedure declaration is given in figure 8.10: `in` is an input parameter and `out` is an output parameter. Procedures cannot return a value and thus are of type `void`. The main routine of the program must be named `MCmain`.

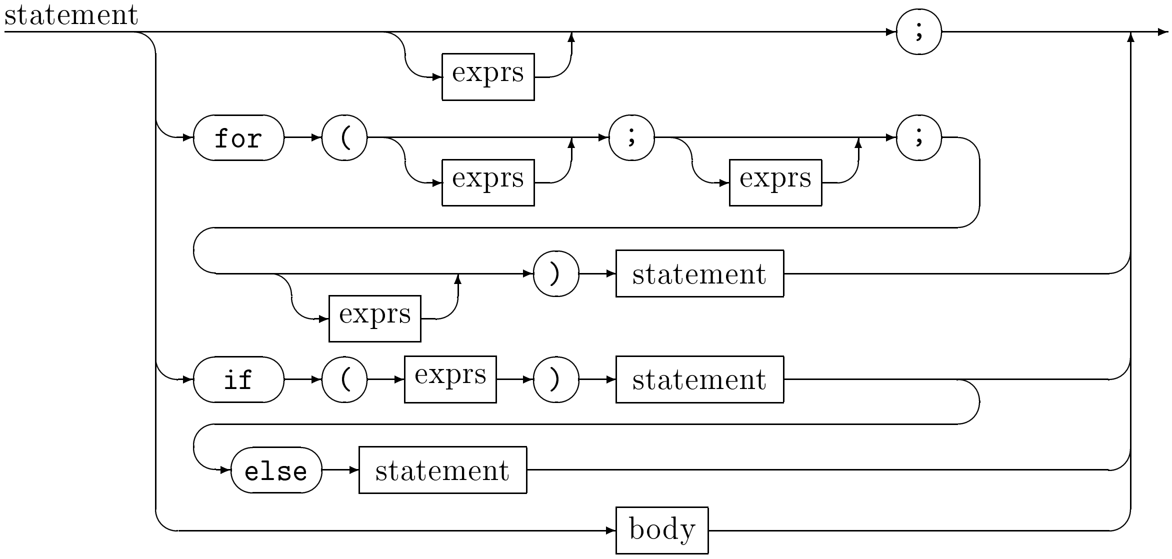

Figure 8.11: Statement syntax

```
for (i = 0; i < memory->size; i++) {
  memory->table[i] = 0;
}
```

Figure 8.12: Example of statements

8.2.5 Statements

The statement syntax is given in figure 8.11. There are five types of statements: empty statements, expressions, loop statements, conditional statements, and blocks.

The conditional statements allow decisions based on the flag computed by the execution units of the Spyder processor.

An example of statements is given in figure 8.12 (it is assumed that the accessed values have been properly defined).

8.2.6 Expressions

The expressions syntax is given in figures 8.13 and 8.14. A general expression can consist of several expressions separated by commas. A simple expression can be either a primary, a unary expression, or a binary expression.

The operators obey the same precedence and associativity rules as in C and C++ [HSJ91, pp. 166–167]. All the available operators are given in figure 8.14. The choice of operators is limited since the actual computation should be performed in the hardware operators.

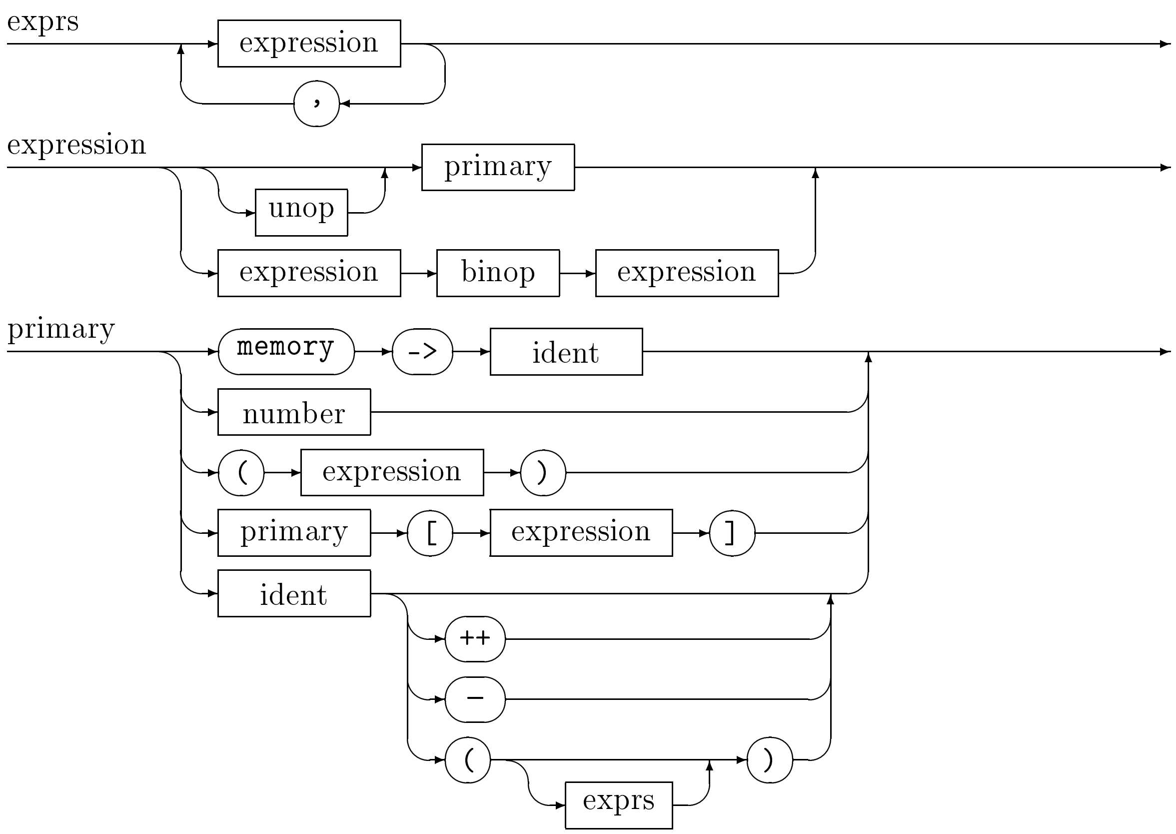

Figure 8.13: Expressions syntax

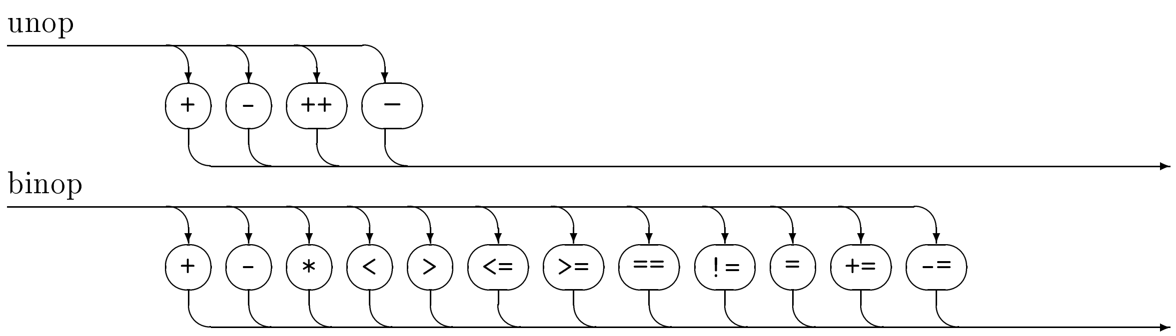

Figure 8.14: Available operations

A primary is either a reference to a memory element, a number, an expression in parenthesis, an array reference, a function call, or an identifier, optionally post-incremented or post-decremented.

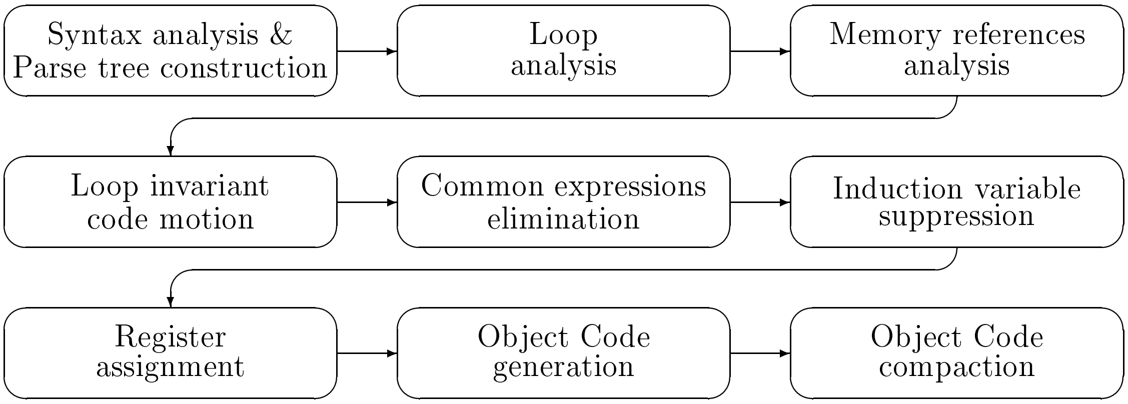

Figure 8.15: Flow of operation of the `mcc` compiler

8.3 The Compiler

The `mcc` compiler is based on standard compilation techniques, as described in the Dragon book [ASU86]. The compiler is written in C++ with the help of two publicly available classes libraries: the GNU C++library [Lea92] and the LEDA library [NU95]. The C++ compiler used was the GNU gcc compiler [Sta95].

The global flow of operations of the `mcc` compiler is shown in figure 8.15. Each function is, in turn, parsed, analyzed, and transformed into object code, i.e., the compiler constructs the parse tree representing the complete function, processes this tree, and generates the corresponding object code, at which stage the parse tree is discarded and the process is repeated with the next functions. The processing is described in detail in the following sections.

8.3.1 Parsing and Tree Construction

A context-free grammar describing the syntax of `mcc` has been developed, and a LALR parser [ASU86, pp. 195–266] for it has been constructed with the help of the parser generator Bison [DS95].

The parser performs two tasks: the creation of the symbol table and the construction of a parse tree.

8.3.1.1 The Symbol Table

The symbol table is implemented using a LEDA dictionary, which associates a symbol name record (class `SNRec`) with the name of the symbol, and the symbol classes shown in figure 8.16.

The symbols are represented by descendants of general `SymRec` class and are split into three categories: hardware operators, represented by elements of class `OpSymRec`, functions, represented by class `FSymRec`, and variables, represented by class `VSymRec`. This last category is further subdivided into two classes: `ASymRec` objects represent array variables and `SSymRec` represent scalar (i.e., simple) variables.

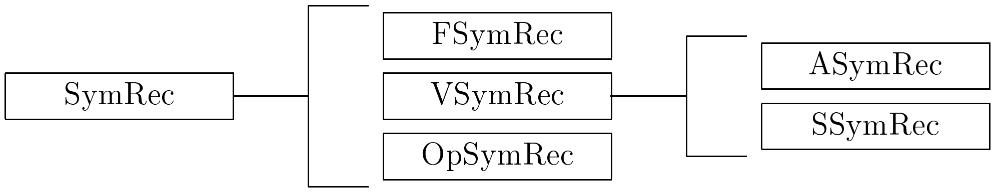

Figure 8.16: The symbol classes hierarchy

8.3.1.2 The Parse Tree

The parse tree is constructed using node objects belonging to the subclasses of the `TreeRec` class shown in figure 8.17, where each subclass corresponds to a specific language construct. The purpose of each subclass is described below. Each `TreeRec` node object has references to its subtrees, if any, as well as a reference to its ancestor in the tree structure, i.e., the node of which it is a subtree.

8.3.1.3 Data Components

There are four data components: constants, represented by `ConstTreeRec`; accesses to array components, represented by `ArrayTreeRec`; accesses to scalar variables, represented by `VarTreeRec`; accesses, through a pointer, to variables stored in the memory, represented by `RefTreeRec`.

The `ConstTreeRec` objects simply contain the integer value of the constant they represent. The `ArrayTreeRec` objects represent the access to an element of an array described by an `ASymRec` object, where the index of the array element is a tree. The `VarTreeRec` objects reference a scalar variable described by a `SSymRec` object. The `RefTreeRec` objects reference a value from the Spyder memory through a pointer, represented by a tree.

8.3.1.4 Operators

There are three kind of operators available: arithmetic operators, comparisons, and auto-increments. The arithmetic operators are represented by objects of classes `AddTreeRec`, `MulTreeRec`, and `NegTreeRec`. The `NegTreeRec` implements the negation of an integer value, represented by a tree.

The addition and multiplication operate on two or more expressions, represented by trees stored in a list. If one of the expression is a constant, it is stored as the first element of the list, while if several expressions are constants, they are evaluated and replaced by the single constant value obtained by applying the operation implemented by the object to the constant expressions. The `AddTreeRec` and `MulTreeRec` implement, respectively, the arithmetic addition and multiplication. Subtractions are implemented by adding the negated value of the second operand to the first, and thus are also implemented through `AddTreeRec` objects.

The `CmpTreeRec` objects represent the comparison (i.e., equality, inequality, greater than, etc.) of two values, represented by two trees, one left and one right. The result of

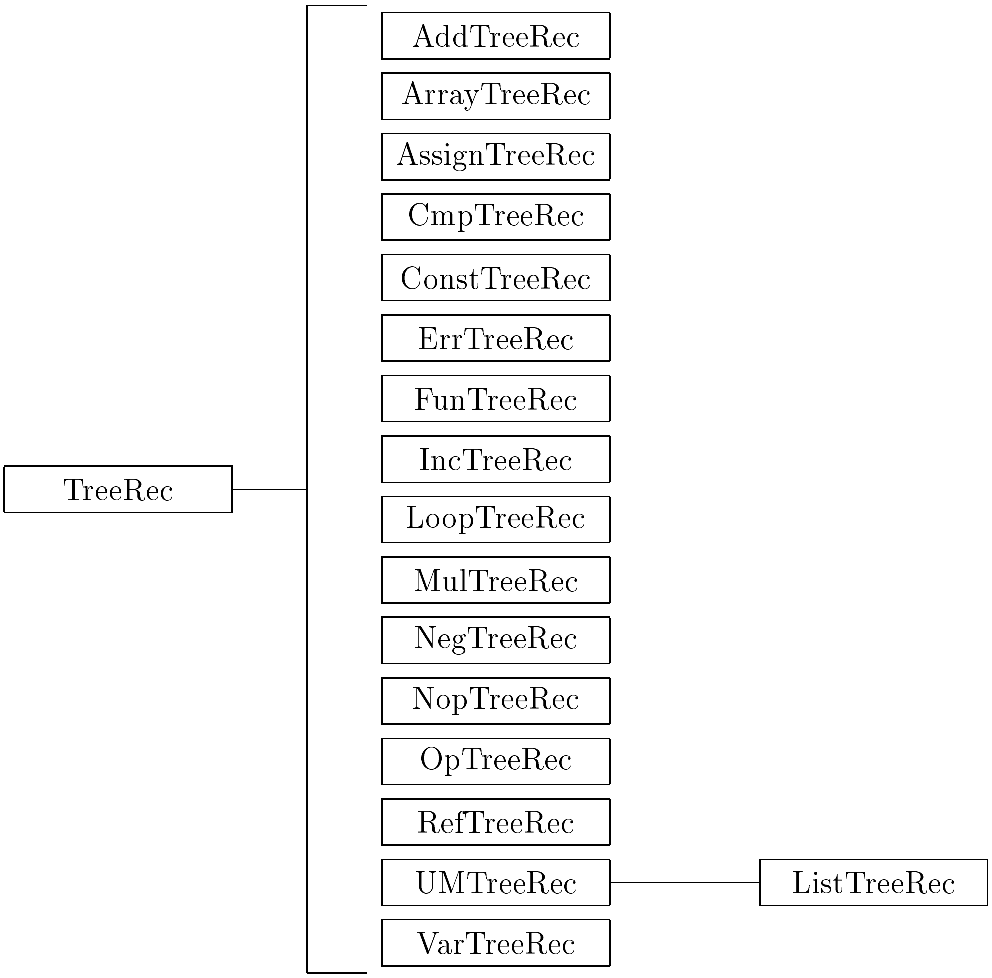

Figure 8.17: The parse tree classes hierarchy

the comparison is a Boolean value.

The `IncTreeRec` objects represent a general form of auto-increment of a value. The increment can either be a pre-increment (i.e., the value is incremented before being used) or a post-increment (i.e., the value is incremented after having been used). The value to be incremented is represented by an `SSymRec` object, and the increment is a tree.

8.3.1.5 Statements

There are five kind of statements: the assignment, the function call, the hardware operator call, the conditional statement, and the loop statement.

The assignment is implemented by the `AssignTreeRec` class. The right-hand side value is assigned to the left-hand side. Both are trees.

The function call is implemented by the `FunTreeRec` class. The function to call is represented by a `FSymRec` object and the parameters of the call are described by a tree list.

The hardware operator call is implemented by the `OpTreeRec` class. The operator to call is represented by an `OpSymRec` object and the parameters of the call are described by a tree list.

Loop statements are implemented by the `LoopTreeRec` class and contain four parts: the initializing expression, the looping condition expression, the increment expression, and the loop body. The three expressions are represented by trees, while the loop body is implemented by a tree list.

8.3.1.6 Other Tree Components

The `ListTreeRec` object contains an ordered list of trees and is used by other tree objects (e.g., the addition operator uses a tree list to store a list of expressions). It is also used to keep a list of statements (for example, the body of a function is represented by a `ListTreeRec` object).

The `NopTreeRec` object is used to represent a no-operation statement. For example, if a loop statement has no initialization sequence, that initialization sequence will be set to a `NopTreeRec` object.

The `ErrTreeRec` object is used when the parser detects a syntax error.

8.3.2 Source Level Optimizations

Once the parse tree of a whole function has been constructed, it is subjected to a number of transformations. The modified parse tree still represents valid C^{++} code, and therefore these transformations are called source level transformations. The `mcc` compiler is able to output to a file this modified parse tree as valid C^{++} source code, which can be compiled by a standard C^{++} compiler for verification.

The purpose of all the applied transformations is twofold: optimize the generated code and fit the source code to the Spyder architecture. Most of the work is performed on loops, where several well-known code transformations are applied: loop-based strength reduction, induction variable elimination, loop-invariant code motion [BGS94].

A small code sample, given in figure 8.18, will be used throughout this section to describe the various transformations applied. The example deals with a two-dimensional array, named `board`, stored in the memory. The dimension of the array is `height` lines of `width` columns and it is accessed line-by-line by the loops shown in lines 11 to 16 of figure 8.18.

8.3.2.1 Loop Analysis

The loop analysis, implemented by the `FindLoopBounds` method, tries to determine the number of iterations performed by each loop statement. For example, given the inner loop statement of figure 8.18, the analysis will determine that the loop is repeated `memory->width` times, a piece of information used to load the loop counter stack available in the sequencer of the Spyder processor. Other information extracted from the loops includes: the name of the loop control variable (also called the induction variable), its initial and final values, and the amount by which it is incremented at each iteration.

```
extern void foo(bitvector top, bitvector mid);
struct memory {
  short width;
  short height;
  short board[0];
};
MCmain()
{
  short i, j, k;
  k = memory->board[0];
  for (i = 0; i < memory->height - 1; i++) {
    for (j = 0; j < memory->width; j++) {
      foo(memory->board[i * memory->width + j],
          memory->board[(i+1) * memory->width + j]);
    }
  }
}
```

Figure 8.18: Sample source code

```
k = mem[2];
for (j = 0; j < memory->width; j += 1) {
  Temp_B_0[j] = mem[2 + j];
}
for (i = 0; i < -1 + memory->height; i += 1) {
  for (j = 0; j < memory->width; j += 1) {
    Temp_D_0 = Temp_B_0[j];
    Temp_B_0[j] = mem[2 + (1 + i) * memory->width + j];
    Temp_D_1 = Temp_B_0[j];
    foo(Temp_D_0, Temp_D_1);
  }
}
```

Figure 8.19: Finding common memory references

8.3.2.2 Memory References Analysis

The purpose of the analysis of memory references is to determine if several statements read the same memory location, in which case the memory is read once and the value

```
1 for (j = 0; j < memory->width; j += 1) {
2   Temp_B_0[j] = mem[2 + j];
3 }
4 Temp_T_0 = 2 + memory->width;
5 for (i = 0; i < -1 + memory->height; i += 1) {
6   Temp_T_1 = i * memory->width + Temp_T_0;
7   for (j = 0; j < memory->width; j += 1) {
8     Temp_D_0 = Temp_B_0[j];
9     Temp_B_0[j] = mem[j + Temp_T_1];
10    Temp_D_1 = Temp_B_0[j];
11    foo(Temp_D_0, Temp_D_1);
12  }
13 }
```

Figure 8.20: Loop invariant code motion

is stored in a register for the subsequent accesses. The task is implemented by the `ArrayToRef` and `FindCommonMemRef` methods.

The effect of applying memory reference analysis to the source code of figure 8.18 is illustrated in figure 8.19. The effect of the `ArrayToRef` method is to transform the accesses to the `board` array into accesses through pointers: since the `board` array starts at position 2 in the memory, the array access on line 10 of figure 8.18 is transformed into the pointer reference `mem[2]` shown on line 1 of figure 8.19, where `mem` is an array that represents the entire Spyder memory and 2 is the offset.

The `FindCommonMemRef` method has determined that the values referenced by line 14 of figure 8.18 will be reused in the next iteration of the outer loop. Thus, these values are stored in the temporary `Temp_B_0` array, initialized by lines 2 to 4 of figure 8.19. Lines 11 to 16 of figure 8.18 are transformed into lines 5 to 12 of figure 8.19.

8.3.2.3 Loop Invariant Code Motion

This step of the compilation process moves outside the loop those pieces of code that do not depend on the loop control variable (i.e., computations that yield constant results across all the iterations of the loop). Some arithmetic transformations of product terms distribution and a small amount of elimination of common expressions are performed before applying the loop invariant code motion, but most of the common expression analysis is performed later and explained below.

The code motion is performed by the `UprootLoopInvariants` method. The result of applying the method to the sample source code of figure 8.18 (after having applied the preceding code transformations) is given in figure 8.20. The pointer computation on line 8 of figure 8.19 has been split into three pieces: one piece has been moved to line 4 of figure 8.20, another to line 6, and the last piece left in place, on line 9. Note that

```
x = a * b + 2;              temp = a * b;
y = c + b * a;      ⟹      x = temp + 2;
                            y = c + temp;
```

Figure 8.21: Example of common expressions elimination

```
x = a * b + 1;      ⟹      x = a * b + 1;
y = a * b + 2;              y = x + 1;
```

Figure 8.22: Example of similar expressions simplification

```
for (j = 0; j < memory->width; j += 1) {
  Temp_A_0 = 2 + j;
  Temp_B_0[j] = mem[Temp_A_0];
}
Temp_A_1 = Temp_A_0;
for (i = 0; i < -1 + memory->height; i += 1) {
  for (j = 0; j < memory->width; j += 1) {
    Temp_D_0 = Temp_B_0[j];
    Temp_A_1 += 1;
    Temp_B_0[j] = mem[Temp_A_1];
    Temp_D_1 = Temp_B_0[j];
    foo(Temp_D_0, Temp_D_1);
  }
}
```

Figure 8.23: Common expressions elimination

line 1 of figure 8.19 has disappeared in figure 8.20 since the value of `k` was never used.

8.3.2.4 Common Expressions Elimination

A great deal of effort is spent by the compiler to find common and similar expressions. Two statements are said to contain common expressions when they contain subexpressions which are identical: e.g., figure 8.21 shows two statements which contain the common expression `a * b`. This common expression is computed once and stored in a temporary variable, which is subsequently used in place of the expression, as shown in figure 8.21.

```
Temp_A_0 = 1;
for (j = 0; j < memory->width; j += 1) {
  Temp_A_0 += 1;
  Temp_B_0[j] = mem[Temp_A_0];
}
Temp_A_1 = Temp_A_0;
for (i = 0; i < -1 + memory->height; i += 1) {
  for (j = 0; j < memory->width; j += 1) {
    Temp_D_0 = Temp_B_0[j];
    Temp_A_1 += 1;
    Temp_B_0[j] = mem[Temp_A_1];
    Temp_D_1 = Temp_B_0[j];
    foo(Temp_D_0, Temp_D_1);
  }
}
```

Figure 8.24: Induction variable suppression

Two expressions are said to be similar if their difference is a simple value, i.e., a constant or the value of a variable. In such cases, the compiler replaces the second expression by the sum of the value of the first expression with the computed difference, as shown in figure 8.22. The purpose of this transformation is to simplify the code and create opportunities to use the adders available in the memory control unit of Spyder.

The `PropagateKnownValues` method is devoted to finding common expressions, while the `PropagateSimilarValues` method to finding similar expressions. The latter method is able to find similar expressions across loop iterations. It is assumed that each loop body is executed at least once. The result of applying the common expressions elimination methods to the example code of figure 8.18, after having applied all the previous code transformations, is shown in figure 8.23.

In figure 8.23, the pointers to the memory array have been assigned to temporary variables of the form `Temp_A_n`, where `A` stands for address. Thus, line 2 of figure 8.20 has been replaced by lines 2 and 3 in figure 8.23, and line 9 of figure 8.20 by lines 9 and 10 in figure 8.23.

The elimination of common expressions has caused line 4 of figure 8.20 to be replaced by line 5 in figure 8.23, and the computation of the memory address used in line 9 of figure 8.20 (lines 6 and 9 of that figure) to be reduced to line 9 in figure 8.23.

8.3.2.5 Induction Variable Suppression

The purpose of induction variable suppression is to avoid using the loop control variable in evaluated expressions. Since the loop execution is controlled by the loop counters in the sequencer of the Spyder processor, the value of that variable will not be readily available

```
Temp_A_0 = 1;
for (j = 0; j < memory->width; j += 1) {
  Temp_A_0 += 1;
  Temp_B_0[j] = mem[Temp_A_0];
  /* inc wptr by 1 */
}
/* reset wptr */
for (i = 0; i < -1 + memory->height; i += 1) {
  for (j = 0; j < memory->width; j += 1) {
    Temp_D_0 = Temp_B_0[j];
    Temp_A_0 += 1;
    Temp_B_0[j] = mem[Temp_A_0];
    foo(Temp_D_0, Temp_B_0[j]);
    /* inc wptr by 1 */
  }
  /* reset wptr */
}
```

Figure 8.25: Source code ready for object code generation

to the processor, which is the main reason for trying to suppress references to the value of that control variable. The suppression task is performed by the `SuppressIndexRef` method.

The result of this transformation on the sample code of figure 8.18 is shown in figure 8.24. Line 3 of figure 8.23 is replaced by lines 1 and 3 in figure 8.24. The fact that the loop control variable `j` is still used on lines 4, 8, 10, and 11 of figure 8.24 does not constitute a problem, since the `Temp_B_0` array will be implemented by the register windowing mechanism (and thus the value of `j` will not be actually used).

8.3.3 Register Assignment

During the register assignment phase, the compiler decides where to store the global and local variables, as well as the temporary variables that were created during the preceding source code transformations. The assignment is performed in two steps. First, the `SetVarLocation` method verifies that each variable is positioned in the correct register set, i.e., that variables used as pointers are stored in address registers and variables used by the hardware operators are stored in the correct register banks. The second step uses standard graph coloring techniques [ASU86] to assign each variable to one register. The graph coloring is used so that more than one variable can be assigned to a given register.

The result of the register assignment phase on the sample code of figure 8.24 is shown in figure 8.25. Line 6 of figure 8.24 has been suppressed since both variables `Temp_A_0` and `Temp_A_1` can be assigned to the same hardware register. Line 12 of figure 8.24 was

```
#include "fooOP.cc"
void
eu_main(bitvector ain, bitvector bin,
  bitvector cmd = bitvector(21), bitvector &flag = bitvector(1),
  bitvector &aout = bitvector(16), bitvector &bout = bitvector(16))
{
  static bitvector t0reg(16,0);
  flag = 0;
  aout = bin; bout = ain;
  if (cmd[0] == 1) {
    foo(ain, bin);
  }
}
```

Figure 8.26: Source code configuration of execution unit 1

```
void
eu_main(bitvector ain, bitvector bin,
  bitvector cmd = bitvector(21), bitvector &flag = bitvector(1),
  bitvector &aout = bitvector(16), bitvector &bout = bitvector(16))
{
  static bitvector t0reg(16,0);
  flag = 0;
  aout = bin; bout = ain;
}
```

Figure 8.27: Source code configuration of execution unit 2

also suppressed since the temporary variable `Temp_D_1` it defined was used only once. The array `Temp_B_0` is implemented in a register bank through the register windowing mechanism, and thus lines 5, 7, 14, and 16 manipulate the register window pointer to access the correct element of the array.

8.3.4 Hardware Operator Assignment

At this stage, the compiler scans the generated Spyder instructions to determine which hardware operators must be implemented by each execution unit. Each hardware operator is then assigned a unique selector, or opcode, and the resulting selection source code is written out for each execution unit. This generated source code is then used by the

```
void
eu_main(bitvector ain, bitvector bin,
  bitvector cmd = bitvector(21), bitvector &flag = bitvector(1),
  bitvector &aout = bitvector(16), bitvector &bout = bitvector(16))
{
  static bitvector t0reg(16,0);
  flag = 0;
  aout = bin; bout = ain;
}
```

Figure 8.28: Source code configuration of execution unit 3

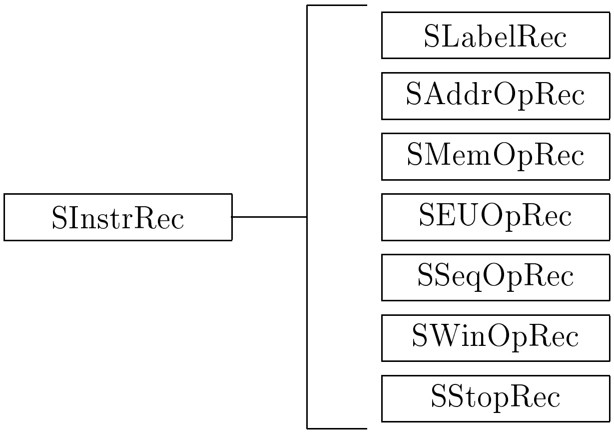

Figure 8.29: The instruction description classes hierarchy

`nlc` compiler (see section 7.3) to synthesize the hardware description of each execution unit.

The hardware operators were either explicitly defined in the source code (e.g., line 1 of figure 8.1), or implicitly generated by the compiler (e.g., moving a value from one register bank to the other).

The source code of the three execution units corresponding to the example of figure 8.1 is given in figures 8.26, 8.27, and 8.28. The `foo` hardware operator, described on line 1 of figure 8.1, has been assigned to execution unit 1 with the operation code 1. The default action of each execution unit is to swap the values coming from the banks R^a and R^b.

8.3.5 Object Code Generation

The object code generation phase of the compilation process transforms the parse tree into a linear list of simple Spyder instructions. These instructions are called simple

```
L_0: a[7]<-0    ,                 ,NOP     ,NOP        ,NOP
 a[0]<-1        ,                 ,NOP     ,NOP        ,NOP
 cntr<-M[a[7]+0],                 ,NOP     ,NOP        ,NOP
L_1:            ,c[0]<-M[a[0]+1]*,NOP      ,NOP        ,NOP,djnz L_1,W+
 a[7]<-M[a[7]+1],                 ,NOP     ,NOP        ,NOP,jump,W0
                ,                 ,NOP     ,NOP        ,NOP
 M[a[7]-1]*     ,                 ,NOP     ,NOP        ,NOP
 cntr<-a[7]     ,                 ,NOP     ,NOP        ,NOP
 a[7]<-0        ,                 ,NOP     ,NOP        ,NOP
L_2:            ,                 ,NOP     ,NOP        ,NOP
 cntr<-M[a[7]+0],                 ,NOP     ,NOP        ,NOP
L_3: M[a[0]+1]* ,                 ,NOP     ,0(c[0]*,c[0]),NOP
                ,c[0]<-M[a[0]+0]  ,NOP     ,NOP        ,NOP
                ,                 ,1(c[0],c[0]),NOP    ,NOP,djnz L_3,W+
                ,                 ,NOP     ,NOP        ,NOP,djnz L_2,W0
                ,                 ,NOP     ,NOP        ,NOP
                ,                 ,NOP     ,NOP        ,NOP
                ,                 ,NOP     ,NOP        ,NOP
*               ,                 ,NOP     ,NOP        ,NOP
                ,                 ,NOP     ,NOP        ,NOP,jump L_0
```

Figure 8.30: Generated object code

because they concern a single resource of the Spyder processor (e.g., the sequencer or the first execution unit). These simple instructions are represented by objects belonging to the subclasses of the `SInstrRec` class shown in figure 8.29.

At this stage of the compilation process, there exists a simple mapping between the nodes of the parse tree and the Spyder instructions. This transformation is performed by the `Compile` method of the `TreeRec` class.

8.3.6 Object Code Compaction

The object code compaction phase of the compiler tries to combine several of the simple instructions produced by the object code generation phase into one VLIW instruction, making use of the instruction-level parallelism available. An instruction dependency graph is constructed, taking into account the pipelined execution of the Spyder processor. Two instructions are said to be dependent when one must be executed before the other, in which case an edge, going from the former to the latter instruction, is added to the graph. The instructions are then sorted, using this dependency graph, by a topological sort algorithm [Sed92].

The topological sort is used to create clusters of simple instructions. The clusters are ordered (i.e., all the instructions of cluster i must be executed before starting to execute the instructions of cluster $i + 1$), but the instructions inside a cluster are independent from each other. The object code compaction phase combines the simple instructions of each cluster into the smallest possible number of Spyder instructions.

The object code corresponding to the example of figure 8.1 is given in figure 8.30. Lines 2 to 4 of figure 8.30 implement the loop of lines 1 to 6 of figure 8.25, lines 5 to 7 of figure 8.30 compute the counter value for the loop starting at line 8 of figure 8.25, while the two nested loops are implemented by lines 10 to 15 of figure 8.30. Lines 16 to 18 of figure 8.30 empty Spyder's pipeline before stopping the processor on line 19.

Chapter 9

The VLIW Assembler imac

The `imac` assembler has been developed to transform a source file containing the description of each very large instruction of a Spyder program into the corresponding binary object code format to be downloaded into the Spyder program memory.

This chapter describes the format of the instructions in the source file and describes the algorithms used by the assembler. See chapter 6 for an example of the use of this assembler.

9.1 The Assembler Source Syntax

The source file is preprocessed by a C preprocessor [Sta92] before being parsed by the assembler. It is thus possible to define symbolic names for the operations of the execution units, as well as any other macro expansion that can simplify the task of writing the program. The C preprocessor also handles the removal of comments (i.e., the syntax of comments in the assembler source file is the same as C^{++} comments).

The global syntax of a program source file is given in figure 9.1. It consists of the specification of the three execution unit command masks, followed by the description of the instructions that compose the program. Each instruction must be written on a single line.

9.1.1 The Execution Units Command Masks

The masks are used by the assembler to properly set the 21 common bits that control the three execution units. The syntax of the mask definition is given in figure 9.2.

Each mask specification defines three bit fields: one for the command of the execution unit, one for the register specification of bank R^a, and one for the register specification of bank R^b. The three execution units are defined by the three identifiers `EU1`, `EU2`, and `EU3`. Each bit field is specified by using the position of its bits among the 21 command bits. Considering the masks specification example given in figure 9.3, the bit field containing the command for the first execution unit is composed of the two bits at position zero and one of the 21 common bits.

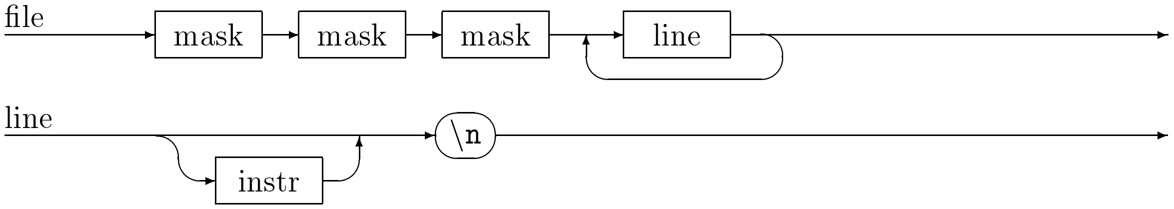

Figure 9.1: Syntax of a complete assembler program

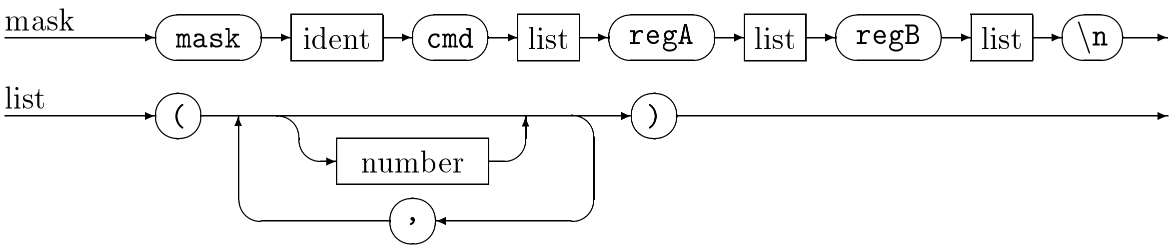

Figure 9.2: Syntax of the masks specification

```
Masks EU1 cmd(1,0) regA(20) regB(19)
Masks EU2 cmd(3,2) regA(18) regB(17)
Masks EU3 cmd(5,4) regA(16) regB(15)
```

Figure 9.3: Example of mask specification

Table 9.1: Example of mask computation, using the masks defined in figure 9.3

Operation	Bit field
1(c[3],c[4])	0 1 0 0 0 0 000000000 00 00 01
2(c[1],c[5])	0 0 0 1 0 0 000000000 00 10 00
3(c[7],c[0])	0 0 0 0 1 0 000000000 11 00 00
All together	0 1 0 1 1 0 000000000 11 10 01

In the default configuration, each register window contains four registers, and the register address is encoded in two dedicated register address lines. When more registers per window are needed, the additional register address lines are borrowed from the 21 common command bits of the execution units. The two register fields are used to define these additional address lines. The masks example given in figure 9.3 defines one additional register address line (i.e., there are 8 registers per window).

Examples of execution unit operations are given in table 9.1. The first three lines

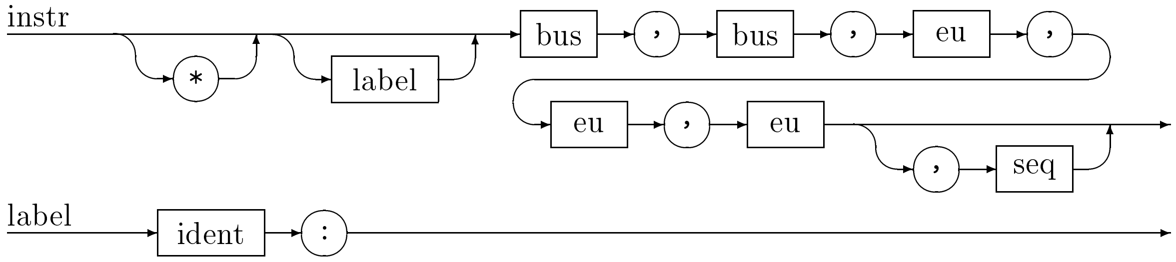

Figure 9.4: Instruction syntax

of the table define, respectively, the three operations of the three execution units, using the masks specified in figure 9.3. The last line represents the resulting 21 common bits of the execution units command word.

9.1.2 Instruction Syntax

Each instruction is composed of eight fields: a breakpoint, a label, two memory transfers, three execution unit operations, and a sequencer operation. The instruction syntax is given in figure 9.4.

A breakpoint causes the Spyder processor to stop and is mainly used to inform the host processor that Spyder has completed its task. It is indicated by a star (`*`) at the beginning of an instruction.

A label is any string followed by a colon (`:`). It is an optional field, used as target for sequencer operations.

The first data transfer command applies to the data memory port connected with the sequencer and with the register bank R^a, while the second data transfer command applies to the data memory port connected with the register bank R^b. Each of the three execution unit commands refers to the corresponding execution unit of the Spyder processor. All these fields are separated by a comma (`,`).

9.1.3 The Data Transfer Commands

The syntax of the data transfer commands is given in figure 9.5. The command is composed of three parts: the left part is a register specification, the middle part specifies the direction of the transfer, and the right part is either an address register, a constant, or a memory address specification. There are two data transfer commands on one assembler line: the first one applies to the A-bus of the Spyder processor and the second one applies to the B-bus.

The registers are defined by a one-letter register selector, followed by a number between square brackets to define one specific register among the selected type. The exception is the loop counter stack, which is defined by the name `cntr`. The list of available selectors is given in table 9.2.

The memory address is computed by adding the content of a first address register, called base address, with either a constant or the content of a second address regis-

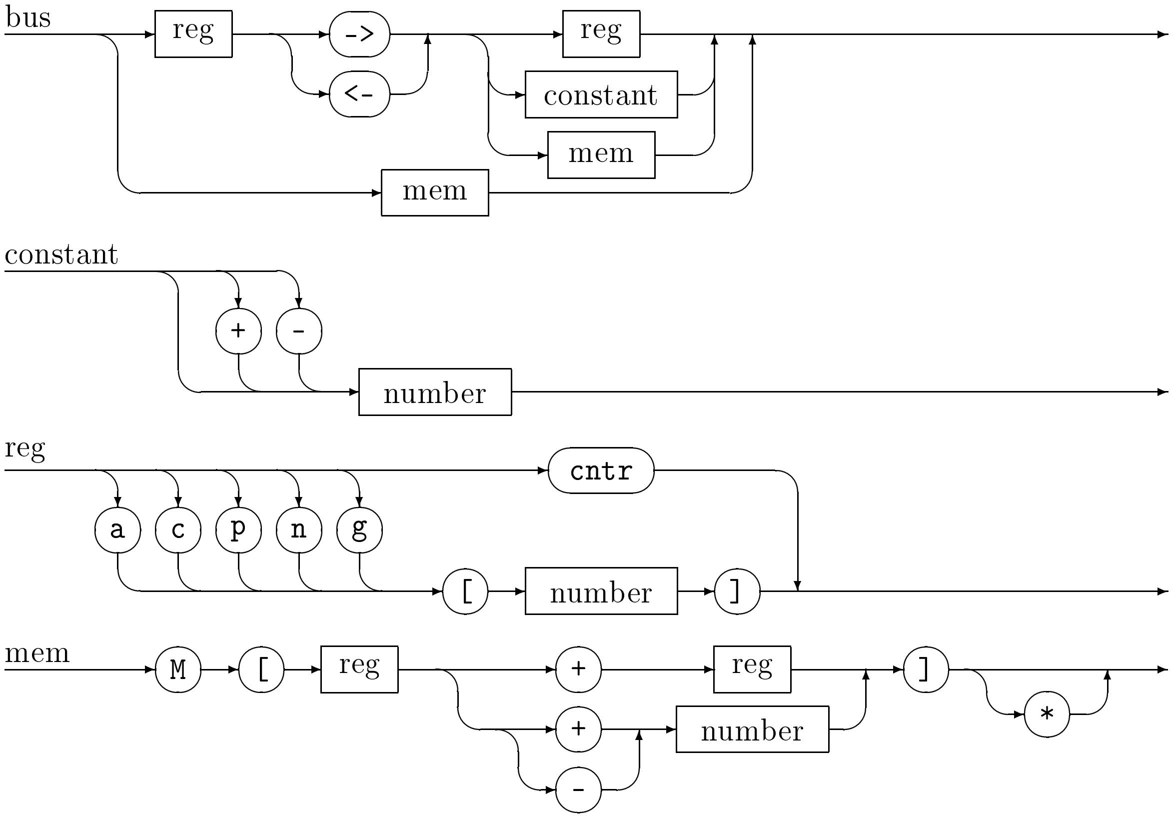

Figure 9.5: Syntax of the memory transfers

Table 9.2: The register selector

Selector	Kind
a	Address register
g	Global window
c	Current window
n	Next window
p	Previous window

ter, called displacement. When the memory specification is followed by a star (*), the resulting computed address is stored back in the base address register.

The A-bus offers more transfer possibilities than the B-bus. In fact, while for the A-bus the register on the left part of the transfer specification can be any register and the right part of the transfer command can specify an address register, a constant, or the content of a data memory location, for the B-bus the register on the left part of the transfer specification must be a member of the data registers (i.e., only the g, c, n, and p selectors are allowed) and the right part of the transfer command must specify the content of a data memory location.

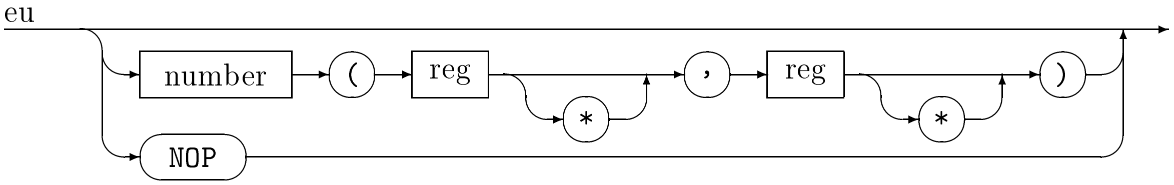

Figure 9.6: Syntax of an execution unit operation

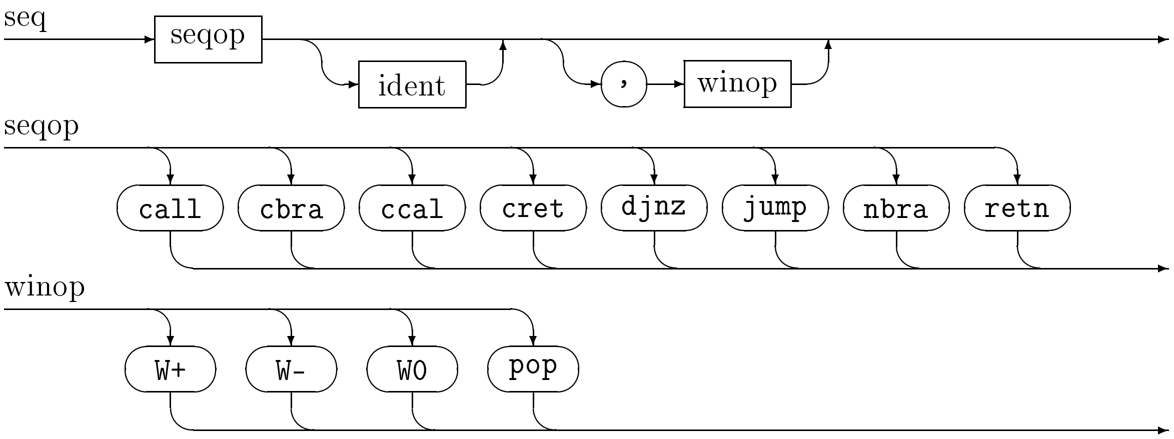

Figure 9.7: Syntax of the sequencer operations

9.1.4 The Execution Unit Commands

The syntax of the execution unit commands is given in figure 9.6. An empty command and the command `NOP` have the same meaning (no operation).

The execution unit operation is specified by a command number, followed by the specification of the two registers to which the command is applied. The first register is selected from register bank R^a and the second one from register bank R^b. Both registers must be data registers (i.e., only the `g`, `c`, `n`, and `p` selectors are allowed).

If the result produced by the execution unit operation must be written back into either or both registers, the corresponding register description is followed by a star (`*`).

9.1.5 The Sequencer Commands

The syntax of the sequencer commands is given in figure 9.7. The command consists of an opcode followed by an optional target label. The opcodes of the sequencer commands are given in table 9.3. When the target of a sequencer operation is omitted, the default is the next instruction.

The sequencer operation can be followed, optionally, by a window pointer command or a counter stack command. The meaning of these additional commands is described in table 9.4. Note that, strictly speaking, the `pop` command does not concern the current window pointer, but was included among the window pointer commands for convenience.

Table 9.3: Sequencer opcodes definition

Opcode	Operation
`call`	Call subroutine
`cbra`	Conditional jump
`ccal`	Conditional subroutine call
`cret`	Conditional return
`djnz`	Decrement counter and jump if not zero
`jump`	Jump
`nbra`	Negative conditional jump
`retn`	Return

Table 9.4: Window pointer and counter stack commands

Opcode	Operation
`W+`	Increment the current window pointer
`W-`	Increment the current window pointer
`W0`	Reset the current window pointer
`pop`	Pop the top of the loop counter stack

9.2 The Assembler

The assembler is written in C++ with the help of two publicly available classes libraries: the GNU C++library [Lea92] and the LEDA library [NU95]. The C++ compiler used was the GNU gcc compiler [Sta95].

The assembler uses two passes to produce the object code corresponding to the instructions contained in an input source file. The first pass parses the input and generates an internal representation of each instruction. A context-free grammar describing the syntax accepted by the assembler has been defined, and an LALR parser [ASU86, pp. 195–266] for that grammar has been constructed with the help of the Bison parser generator [DS95].

During the parsing of the input file, all the fields of the defined instructions are filled, except the jump address field of forward branching instructions. The assembler constructs a list of all the defined labels with their corresponding address and a list of the instructions with an undefined jump address field.

The second pass of the assembler fills the jump address field of the forward branching instructions using the two lists defined during the first pass.

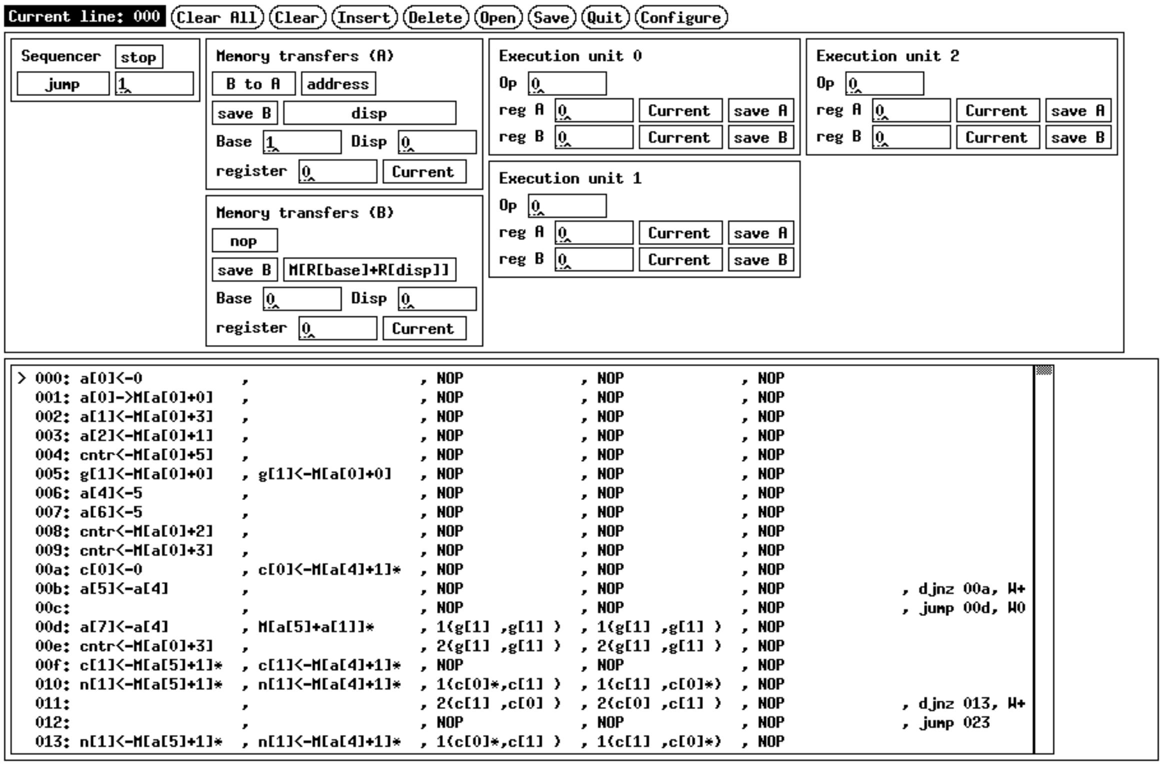

Figure 9.8: The interactive assembler interface

9.3 Related Tools

A program, named `imalist`, has been developed to perform the disassembly of a Spyder program. The output of this disassembler can be directly reassembled by the `imac` assembler.

An interactive assembler, with a graphical user interface, was also developed to allow the user to easily make small modifications to existing code (such as inserting a breakpoint). Figure 9.8 shows a snapshot of the user interface. The user can modify all the components of a Spyder instruction using menus, buttons, and the keyboard to enter values. The interactive assembler is named `ima` and was developed using the Athena toolkit of the X window system [Pet94].

Chapter 10

The UNIX Interface

In order to let the UNIX operating system access the Spyder coprocessor, the UNIX kernel must load a device driver. The driver consists of two files: a configuration file and the file containing the code of the device driver. This chapter describes the features provided by the device driver code and explains how to install the Spyder board and its driver into the host workstation.

10.1 The Spyder Device Driver

A device driver consists of a collection of C functions that are required by the UNIX operating system to access a given device. The functions must perform tasks concerned with installing and uninstalling the driver, defining the entry points of the device, opening and closing access to these entry points, reading and writing data into the device, and controlling the device operations [Sun93].

The device drivers on the Solaris operating system, which is the variant of UNIX used in this project, are organized into a tree structure. For instance, all the device drivers for VME devices, of which Spyder is a member, are attached to a general VME device driver which implement basic operations common to all VME device drivers. Thus the Spyder device driver must implement its routines in terms of the functions provided by the general driver [FOR94a].

The Spyder device driver defines four entry points into the Spyder coprocessor: one for the data memory, one for the program memory, one for the configuration of the FPGA circuits, and one to access the control and status register. The remainder of this section will describe each of these entry points without describing the implementation in too much detail. If needed, a detailed description can be found in the source code of the driver.

10.1.1 The Data Memory

The Spyder data memory can be accessed through the `/dev/spyder.ram` file. The driver implements the `open`, `close`, `read`, `write`, `lseek`, and `mmap` system calls. The user can map the Spyder data memory in the address space of its program by using the `mmap` system call, in which case the Spyder memory must be accessed as an array of 32-bit words, i.e., the address of the access must be a multiple of four bytes.

When the `read` and `write` calls are used, the driver handles the addressing automatically. The main reason for using the `mmap` call is that the access to the Spyder memory becomes much faster, since the driver code no longer needs to be called for reading and writing data.

10.1.2 The Program Memory

The program memory is accessed through the `/dev/spyder.rom` file. The driver implements the `open`, `close`, and `write` system calls. Opening this access point causes the Spyder processor to be stopped and reset and fills the pipeline register with zeroes. The only possible operation is a sequential write into the program memory, starting at address zero.

While this entry point is open it is not possible to open the FPGA configuration entry point.

10.1.3 The FPGA Configuration

The configuration of the FPGA circuits on the Spyder board is performed through the `/dev/spyder.lca` file. The driver implements the `open`, `close`, and `write` system calls. Opening this access point stops the Spyder processor, clears the current configuration of the FPGA chips, and fills the pipeline register of the program memory with zeroes. The only possible operation is a sequential write of the configuration data for the FPGA chips.

While this entry point is open it is not possible to open the program memory entry point.

10.1.4 The Control and Status Register

The Spyder control and status register can be accessed through the `/dev/spyder.csr` file. The driver implements the `open`, `close`, `read`, `write`, and `mmap` system calls. The user can map the register in the address space of its program by using the `mmap` system call. The size of the register is one byte and, for some unidentified reason, the pointer returned by the `mmap` system call must be incremented by one to actually point to the register.

10.2 Installing the Spyder Board

The Spyder processor is implemented on a standard double Europe VME board [Mot85] which must be plugged in a VME backplane. The Spyder processor is implemented as a slave device, i.e., there must be another board acting as bus master on the backplane. For example, the Spyder board is currently plugged in a VME backplane together with a SPARC CPU-5CE from Force computers [FOR94c, FOR94d, FOR94a, FOR94b], which is fully compatible with the SPARCstation computers from Sun Microsystems.

```
type=ddi_spyder \D.\M0
```

Figure 10.1: Spyder device links information

Once the board is correctly inserted in place, a driver for this board must be configured in the system. It is assumed that the board is plugged in a Sun SPARC-compatible workstation and that the operating system is Solaris 2 [Sun94]. Should this not be the case, the driver installation instructions might be different.

10.2.1 Installing the Driver Files

The driver consists of two files: `spyder` and `spyder.conf`. The first contains the code of the driver itself and the second contains configuration information. These two files must be copied in the `/usr/kernel/drv` directory, for the operating system kernel to be able to locate them.

10.2.2 Configuring the Entry Points

In order for the system to create the appropriate links in the `/dev` directory, the line shown in figure 10.1 must be added to the `/etc/devlink.tab` file.

10.2.3 Configuring the System

Once everything is in place, the command `add_drv spyder` can be used to inform the operating system that a new device driver has been added. It might be necessary to check and set up the access permissions of the device driver entry files created in the `/devices` directory. These files are pointed to by the `/dev/spyder.*` links. If no problems arise, the Spyder driver will announce itself in the system log file. Otherwise, the system log file should provide information about the problems encountered.

Chapter 11

Evaluation

This chapter presents four applications that have been implemented on the Spyder processor. Two applications are based on cellular automata simulation: John Conway's game of Life and an experiment in dendrite growth modeling. The other two applications concern low level image processing: thinning and edge detection.

11.1 Life Cellular Automaton

The sequential nature of standard processors and their fixed data size render them unsuitable to efficiently simulate cellular automata. Therefore, many dedicated processors have been designed and implemented to handle this kind of applications [PD84, TM87].

The implementation of the Life cellular automaton on the Spyder processor, described in detail in chapter 6, has been tested against `xlife`, a program running on the X window system, available from anonymous ftp on the host `ftp.x.org`.

Both programs have been used to simulate a block pusher pattern (file `blockpusher`, included with `xlife`) on a 608×608 array of cells for 13160 iterations. The block pusher pattern was selected because it presents a fairly complicated and stable pattern (where by stable we mean that the pattern does not tend to grow quickly out of the array). Figure 11.1 shows the block pusher pattern in its initial state on the left side of the figure and after 13160 iterations on the right side. The pattern is called a block pusher because it pushes a small block of cells (visible at the edge that is closest to the bottom right corner of the initial pattern) to the bottom right corner of the cellular array, as can be seen on the right side of figure 11.1.

The simulation was run on both Spyder and a SPARCstation 5 computer from Sun Microsystems. A comparison of the results is given in table 11.1.

The `xlife` program is highly optimized and uses fairly complicated algorithms and data structures to avoid computing the new state of cells in regions of the array where nothing will change during the current iteration. In the block pusher pattern, there is an average of 2800 live cells at any given time (out of a total of 369664 cells). The ratio of live to dead cells is thus less than 1 to 100 and `xlife` can significantly reduce the amount of computation by avoiding to recompute cell states in regions where all the cells are dead.

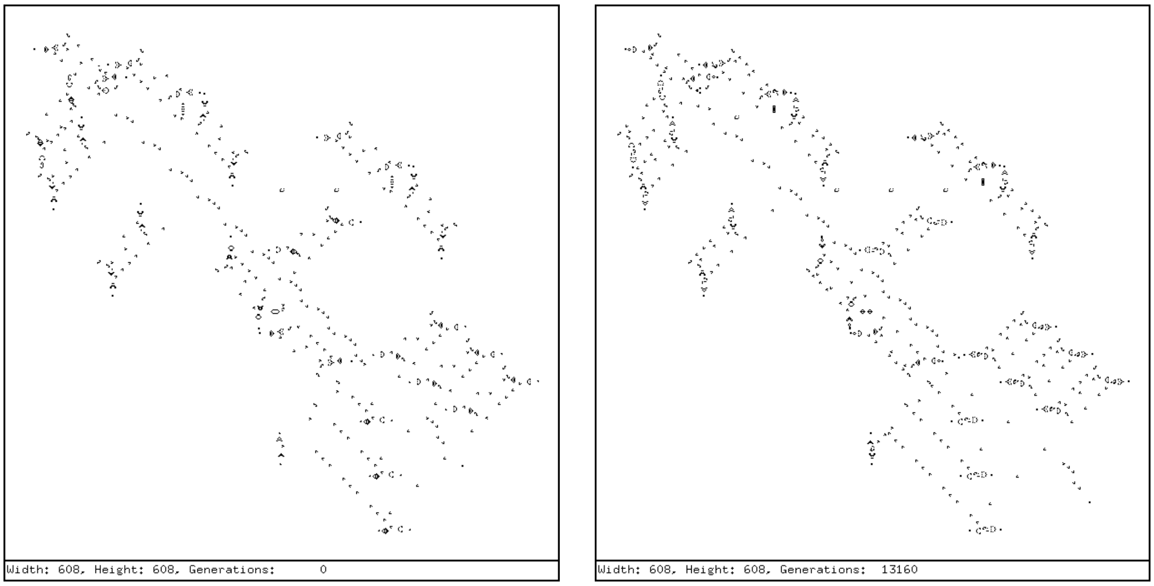

Figure 11.1: Block pusher pattern simulation

Table 11.1: Execution time for the Life cellular automaton

Processor	Clock frequency	Computation time
microSPARC 2	85 MHz	55.3 s
Spyder	8 MHz	41.6 s

The Spyder implementation, on the other hand, uses a simple algorithm and always computes the new state of all the cells of the array at each iteration.

To summarize, Spyder can compute the future state of about 115 million cells per second (369664 cells $\times$ 13160 iterations/41.6 s) with a clock speed of 8 MHz, compared to the approximately 6.5 million cells per second (27000 cells $\times$ 13160 iterations/55.3 s, where 27000 is the maximum number of computed cells based on the 3000 live cells and their 8 surrounding neighbors) computed by the SPARCstation's microSPARC 2 processor with its 85 MHz clock. If Spyder had 32-bit data words (there is room enough in the execution units), it could compute about 227 million cells per second with the same clock speed. An additional limiting factor is the bandwidth between the data memory and the registers: if more ports between the data memory and the registers were available in the current implementation, all three execution units could work in parallel and a speed of about 172 million cells per second could be achieved.

11.2 Dendrite Growth Modeling

The purpose of this application is to try to simulate, using cellular automata, the growth of a dendrite (see [OA74, HT94] for examples of real dendrites). The method was pro-

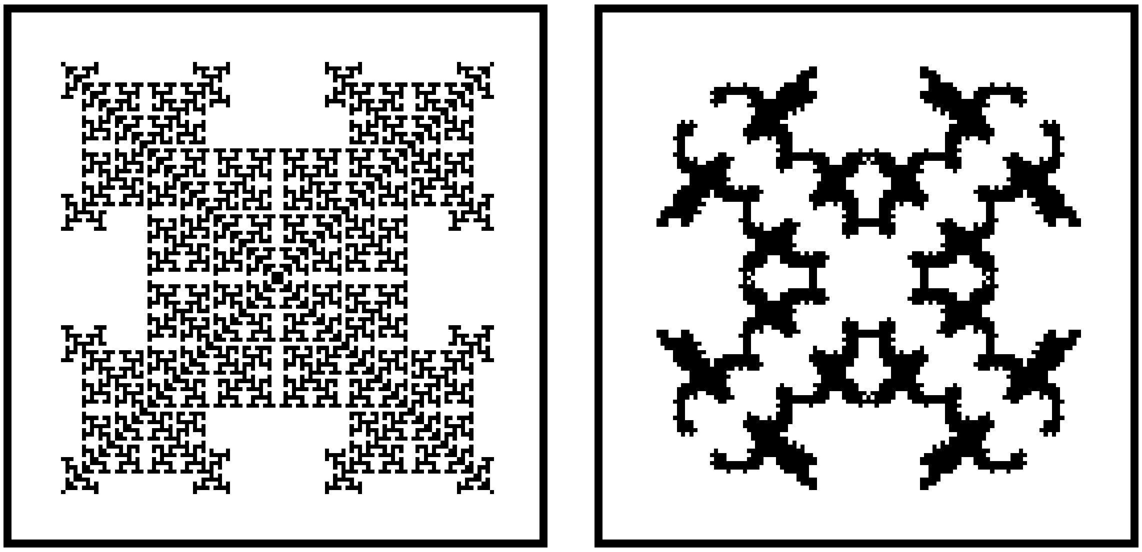

Figure 11.2: Cellular automaton state after growth, and melting steps

posed by professor Rappaz of the material sciences department of the EPFL, and the implementation was realized by a student as a semester project in our laboratory.

Each cell of the automaton can be in either of two states: melted or solid. The basis of the method is the use of two different sets of rules for a cellular automaton consisting of 2-state cells: the first set of rules causes the crystal to grow, while the second set of rules causes part of the crystal to melt back. These two antagonistic sets of rules are applied by turns, as seems to happen in real dendrite growth.

Both set of rules were taken from [TM87]. The growth rule causes a new solid cell to appear if it sees exactly one solid cell among its eight neighbors. The left image on figure 11.2 shows the effects of the growth rule, starting with a single central solid cell, after 52 time steps. The melting rule, known as Vichniac rule, states that a cell is solid if there are 4, 6, 7, 8, or 9 solid cells (including itself) in its immediate neighborhood, and melted in the other cases. The image on the right side of figure 11.2 shows the effects of 5 steps of the melting rule on the left side image.

This is an ongoing research project. There are a lot of possibilities to explore concerning the proper balance between applications of the two sets of rules and their sequencing, and it is not clear if the rules should be applied as they are, or modified to better emulate what happens at the molecular level in a melted alloy. The high speed of simulation offered by the Spyder processor permits to quickly explore different approaches. The current implementation uses two execution units working in parallel. Both execution units are configured to compute the growth and Vichniac rules of 16 cells in parallel, in the same manner used for the game of Life.

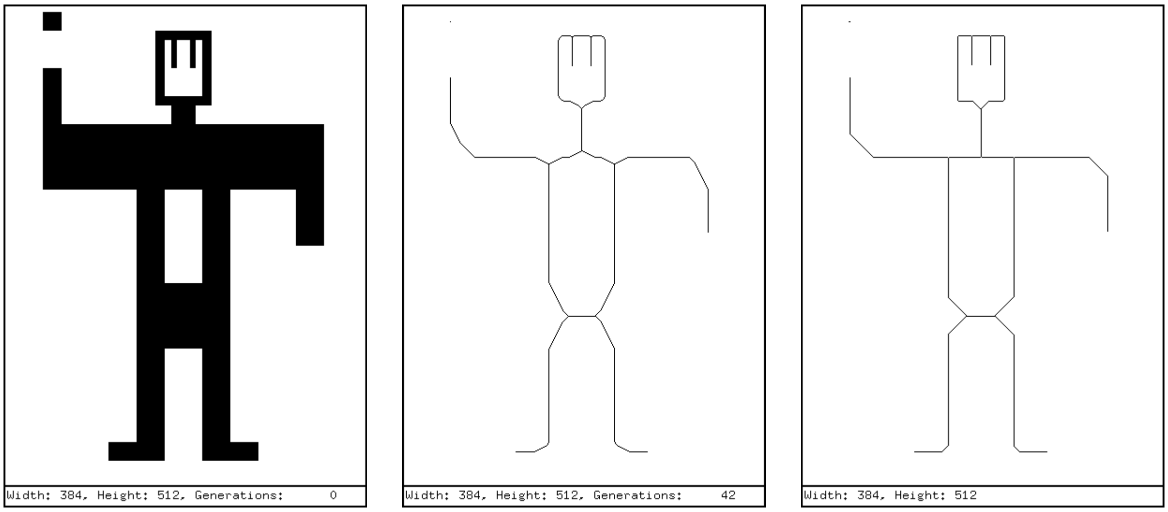

Figure 11.3: Robot image before and after skeletonization

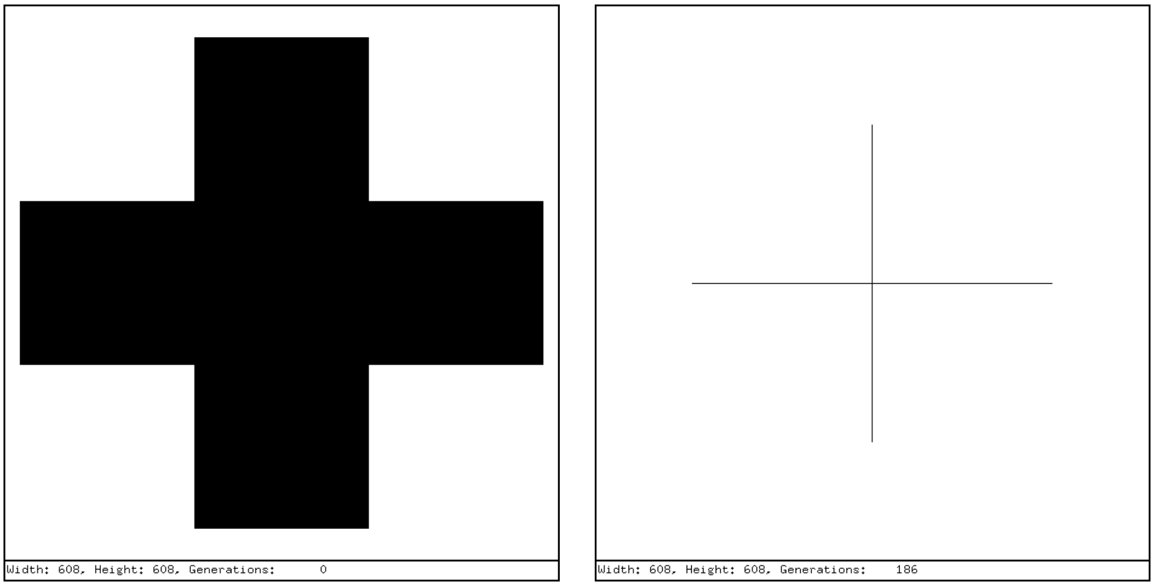

Figure 11.4: Cross image before and after skeletonization

11.3 Thinning

Thinning is a computation-intensive process used in many image processing and pattern recognition algorithms. Its purpose is to reduce the elements composing the image to lines one pixel wide, usually for further processing. For example, a character recognition algorithm will start by thinning the image of each character before attempting to recognize it.

A parallel thinning with two-subiteration algorithm, described in [GH89], has been implemented on the Spyder processor. This algorithm uses a scanning window technique

Table 11.2: Execution times for the skeletonization algorithms

Processor	Clock frequency	Robot Image	Cross Image
microSPARC 2	85 MHz	7.8 s	34.7 s
Spyder	8 MHz	0.154 s	1.17 s
ratio		50.6	29.7

to erode the elements of the image until a fix point is obtained. The window is 3×3 pixels square, and the computation in the window involves applying logic operations to the pixels, as well as performing some sums. The algorithm implemented on the Spyder processor uses two execution units working in parallel. Both execution units are configured to compute the thinning algorithm for 16 pixels in parallel.

The thinning implemented on the SPARCstation computer uses a different algorithm: Rosenfeld's parallel thinning algorithm [RK82] taken from [Cyc94], an efficient binary image thinning using neighborhood maps. This algorithm uses some computations and a table lookup mechanism to erode the elements of the image until a fix point is obtained.

Figures 11.3 and 11.4 show two examples of image thinning. The center image in figure 11.3 was obtained by Spyder, while the image on the right by the SPARCstation. The execution time for the thinning of each of these images is given in table 11.2. The difference in the speedup obtained by Spyder is due to the contents of the image, and probably reflects the difference in the thinning method used by the algorithms (as can be observed in figure 11.3, the resulting thinned images are not quite identical.

11.4 Edge Detection

Edge detection is an important image processing step when trying to determine the movement of objects in an image. Image movement detection is used, for example, for video image compression, where the edge detection step is used to locate objects in two consecutive images before trying to determine if those objects moved between one image and the other.

Doctor Domingo Benitez Diaz, during a one-month stage in our lab, implemented an edge detection algorithm on the Spyder processor [Dia96]. The purpose of the algorithm is to take a gray-level image as input and produce a black and white image as output, where pixels belonging to an edge are white and the others are black.

The method used to determine if a pixel belongs to an edge consists of computing the average brightness of its neighbors and determining if there is a sharp difference between the brightness of the pixel and the average brightness. This method uses a scanning window mechanism, where the window is 4 pixels high by 4 pixels wide. The algorithm implemented on the Spyder processor uses all three execution units in parallel. Each execution unit is configured to compute the edge detection algorithm for 2 pixels in parallel.

Table 11.3: Run time of different processors running the edge detection algorithm

Processor	Clock frequency	Time (ms)	Time/Time Spyder
Spyder	6.25 MHz	25	1
Notebook Pentium	90 MHz	400	16
HP 750	75 MHz	1400	56
SPARCstation 5	85 MHz	1400	56
SPARCstation 10	40 MHz	1800	72
SPARC Classic	50 MHz	2800	112
SPARCstation 2	40 MHz	3500	140

Table 11.3, copied from [Dia96] by permission, compares the execution speed of the edge detection algorithm running on Spyder and on several other common workstation processors. The input image used is a 256-level grayscale image containing 256 lines of 256 pixels (each pixel is thus 8-bit deep). The output image is black and white.

Chapter 12

Conclusions

A prototype of the Spyder processor has been implemented on a VME board, which has been installed in a VME rack along with a SPARC board acting as host computer. The SPARC board is compatible with the SPARCstation computers from Sun Microsystems and runs the Solaris operating system. A Solaris device driver for the Spyder board has been developed and allows the host workstation to easily communicate with the Spyder board.

The software environment needed to develop applications for the Spyder processor has been implemented and used by students.

The Spyder project has been presented at several international conferences [IS93b, IS93a, IS94a, IS94b, DI93] and an extended presentation has appeared in a special issue of the Journal of Supercomputing on field-programmable gate arrays [IS95b].

The `nlc` hardware synthesizer has also generated interest on its own right in the scientific community. It has been presented at two international conferences [IS95a, IS95c] and made available by anonymous ftp on the Internet.

At the start of the Spyder project, reconfigurable and VLIW processors were rarely the subject of serious consideration. To the best of my knowledge, Spyder was the first reconfigurable processor with a VLIW processor architecture.

But nowadays barely a month passes without new reconfigurable processor architectures being proposed, and some processor manufacturers appear to have plans to release VLIW general-purpose processors. When asked what lies beyond the next generation of superscalar processors, Nick Tredennick, Altera's chief scientist, proposed reconfigurable accelerators [Gwe94]. In his own words: "If lack of parallelism in the instruction stream seems to be the problem, let's quit executing instructions."

The Spyder prototype has demonstrated its ability to achieve a high level of performance, even when running with a slow clock. The best performances were achieved in low-level image processing applications and cellular automata simulations, which require a large amount of local data computations.

It is interesting to note that Spyder allows to directly apply simple, intuitive algorithms, instead of forcing the user to find complicated means to achieve greater speed because of the lack of the proper elementary operations in general purpose processors.

FPGA chips follow the same technology curve as microprocessors, and as time passes, bigger and faster FPGA chips become available. Thus, if a reconfigurable processor can outperform a general-purpose microprocessor today, there is no reason to believe that it will not do so in the future.

Appendix A

Spyder Schematic

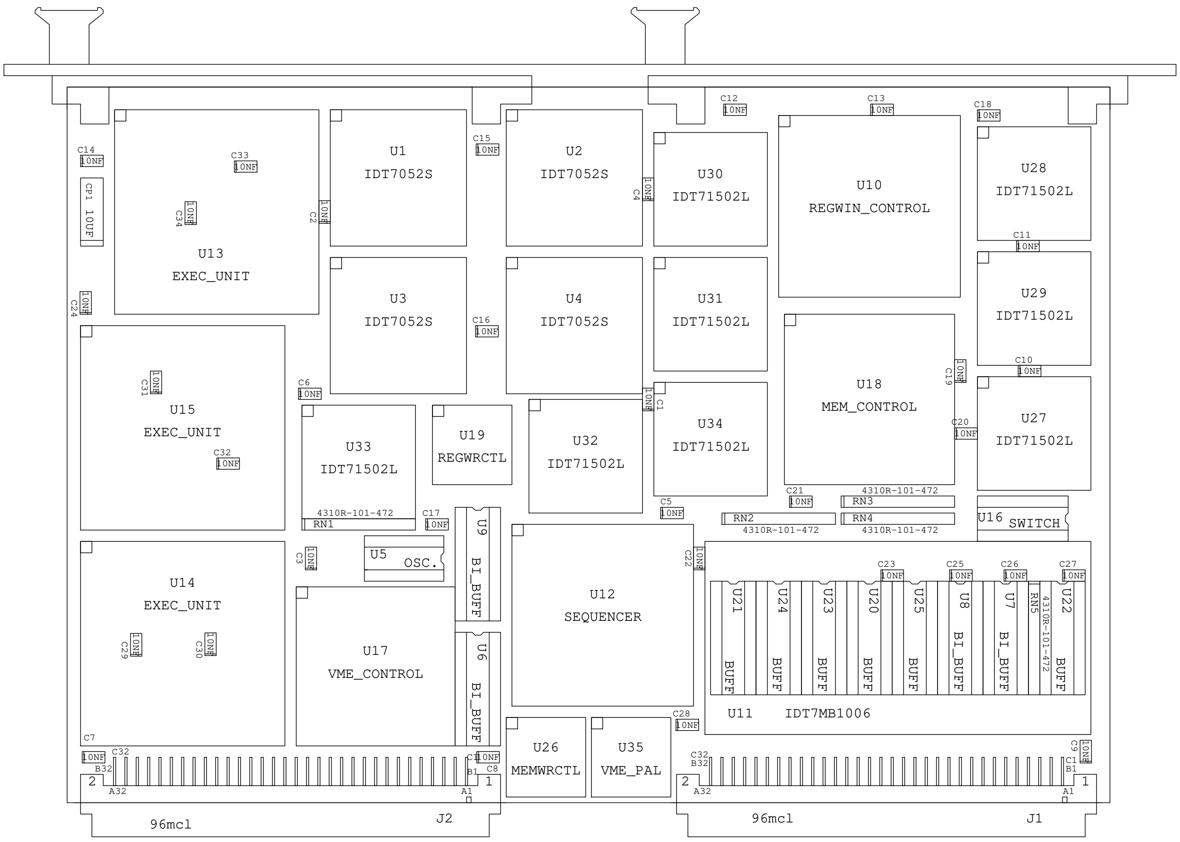

Figure A.1: Spyder VME board

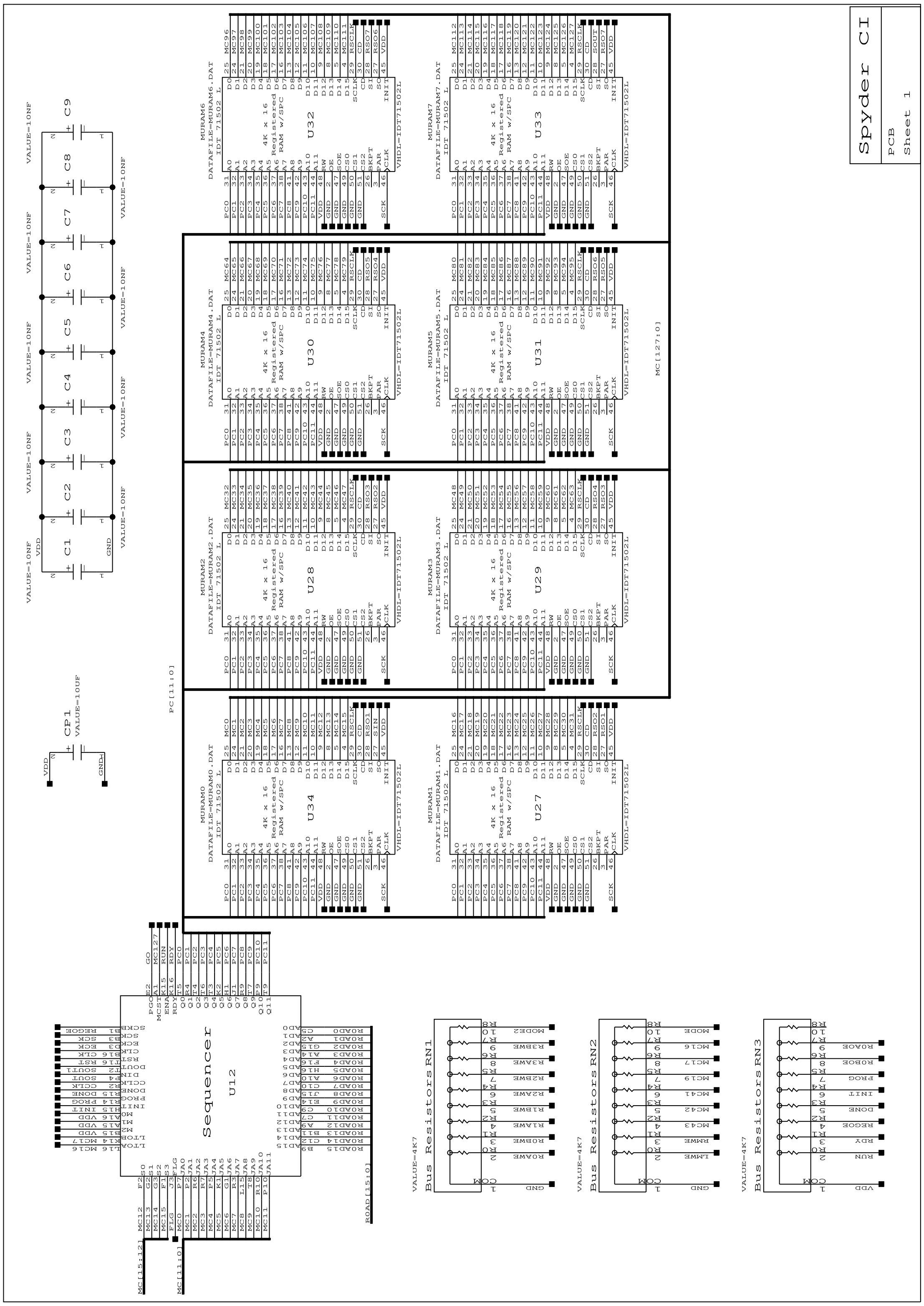

Figure A.2: Spyder program memory and sequencer

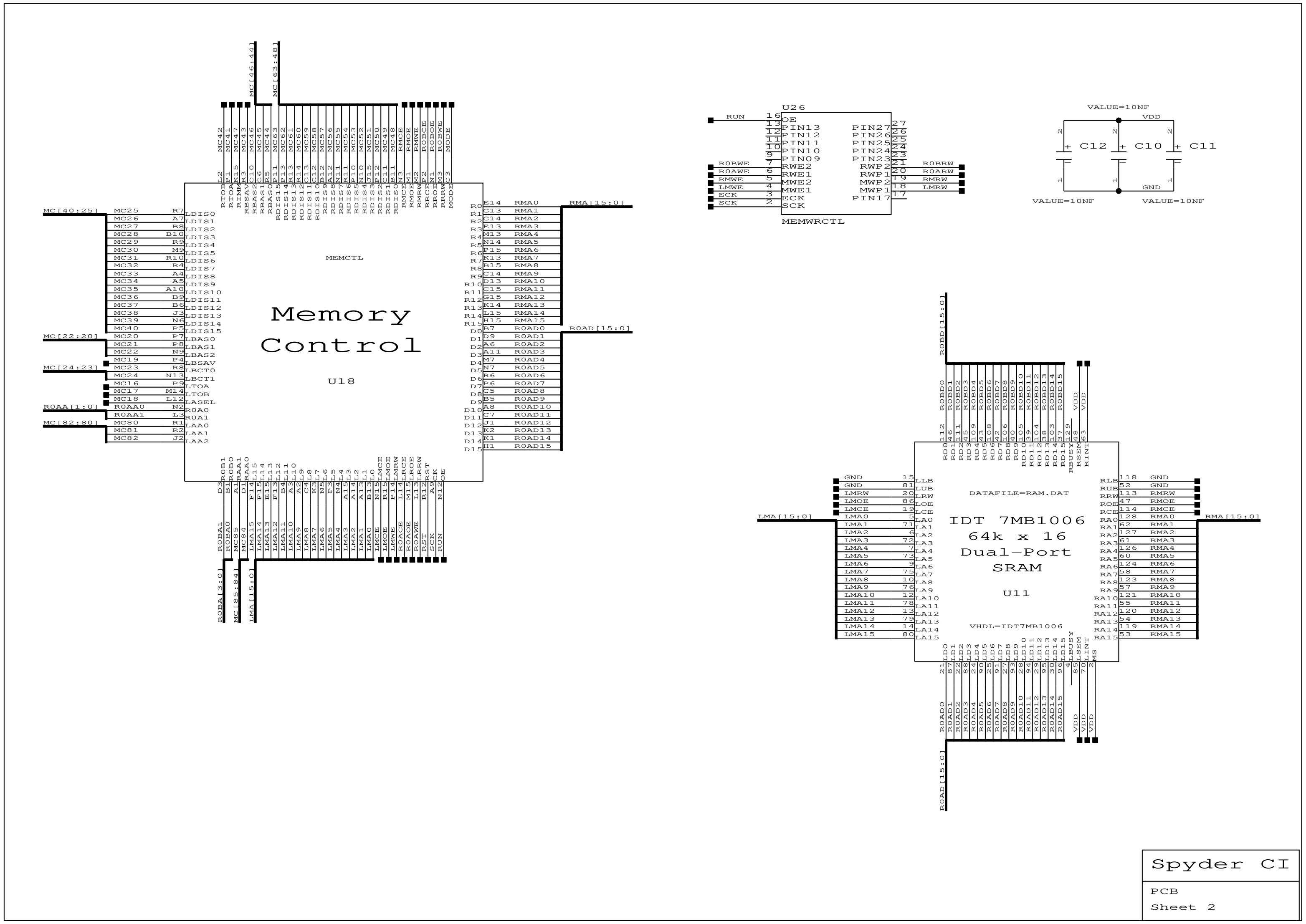

Figure A.3: Spyder data memory and controller

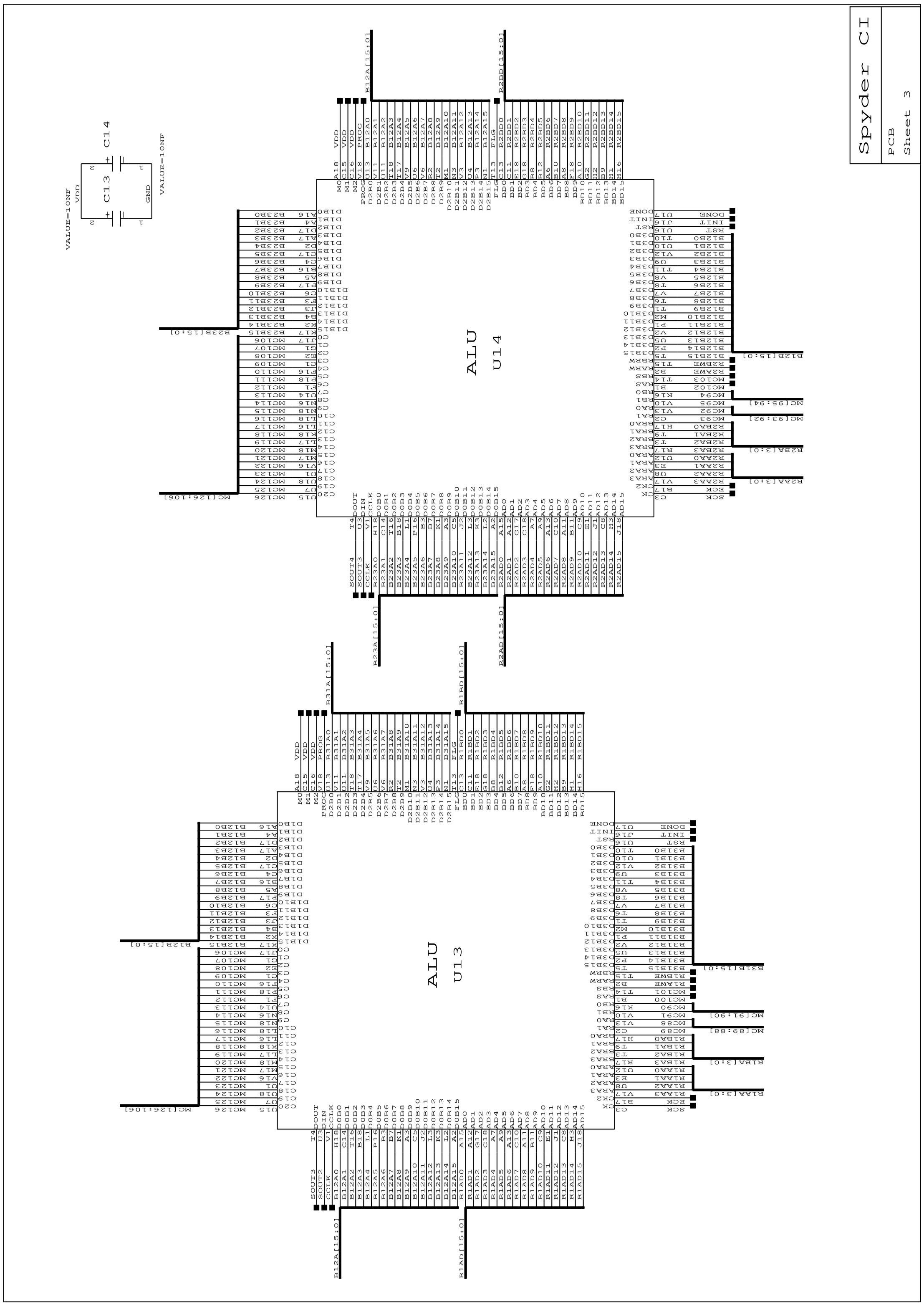

Figure A.4: Spyder execution units 1 and 2

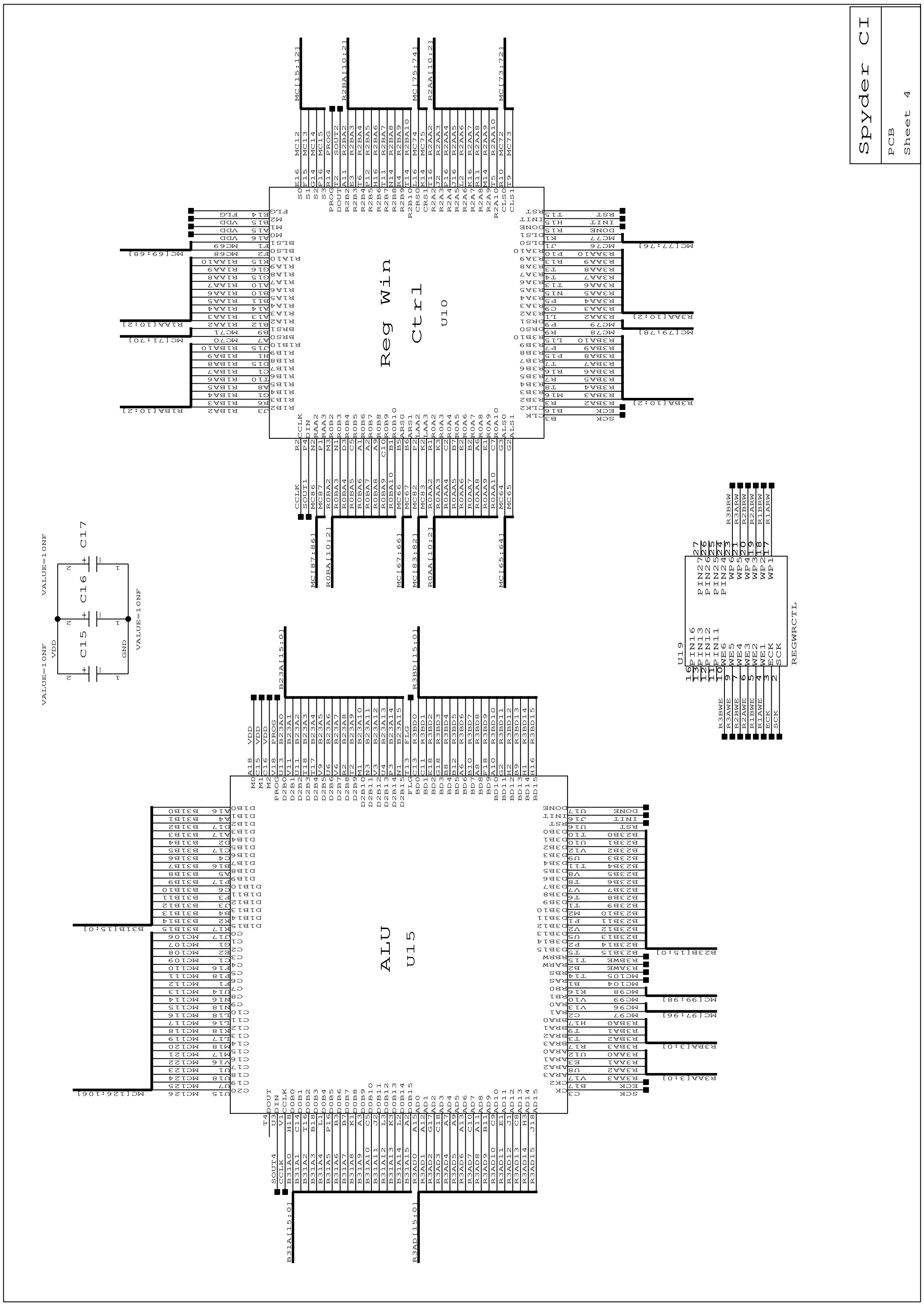

Figure A.5: Spyder execution unit 3 and window controller

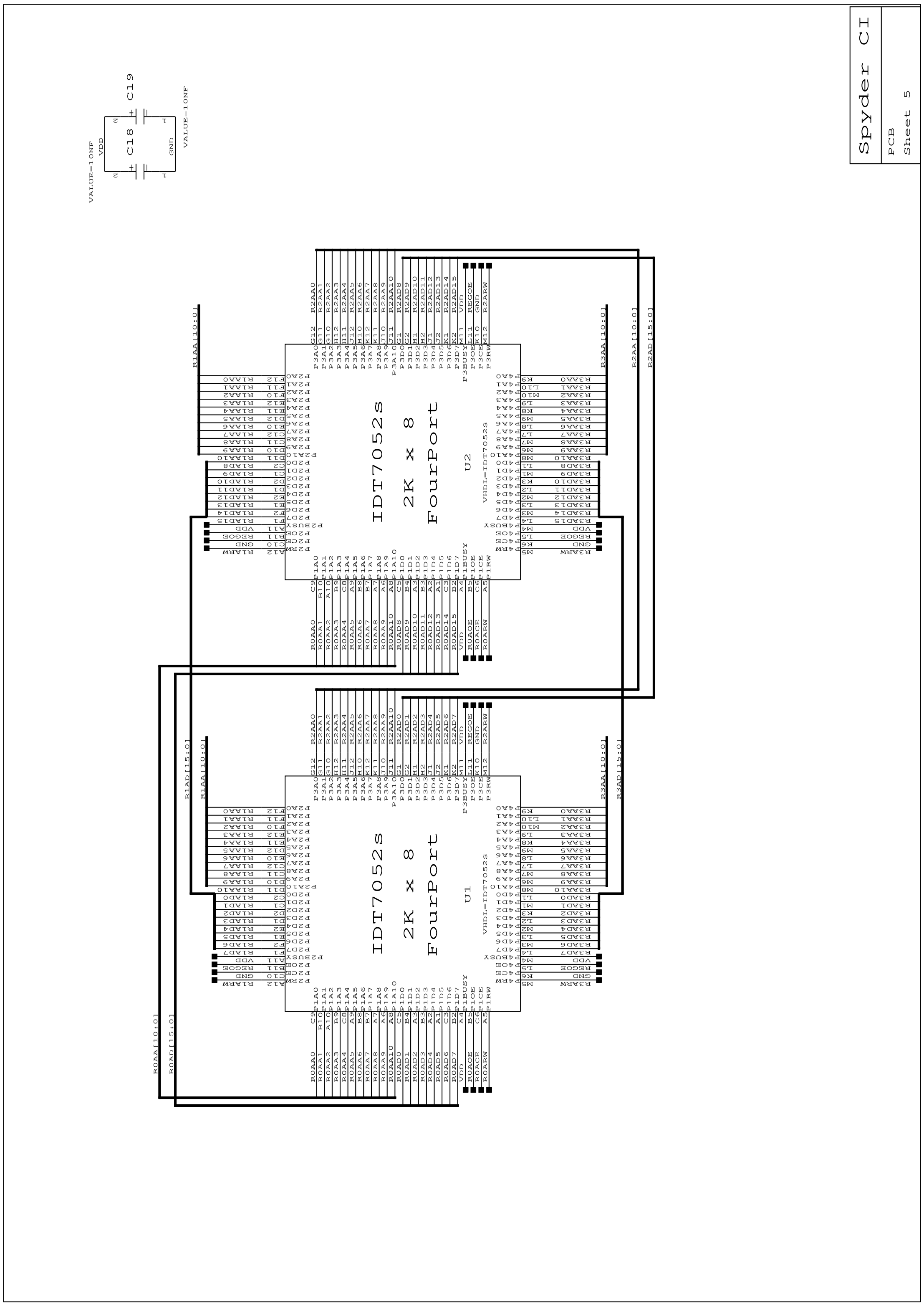

Figure A.6: Spyder register bank R^a

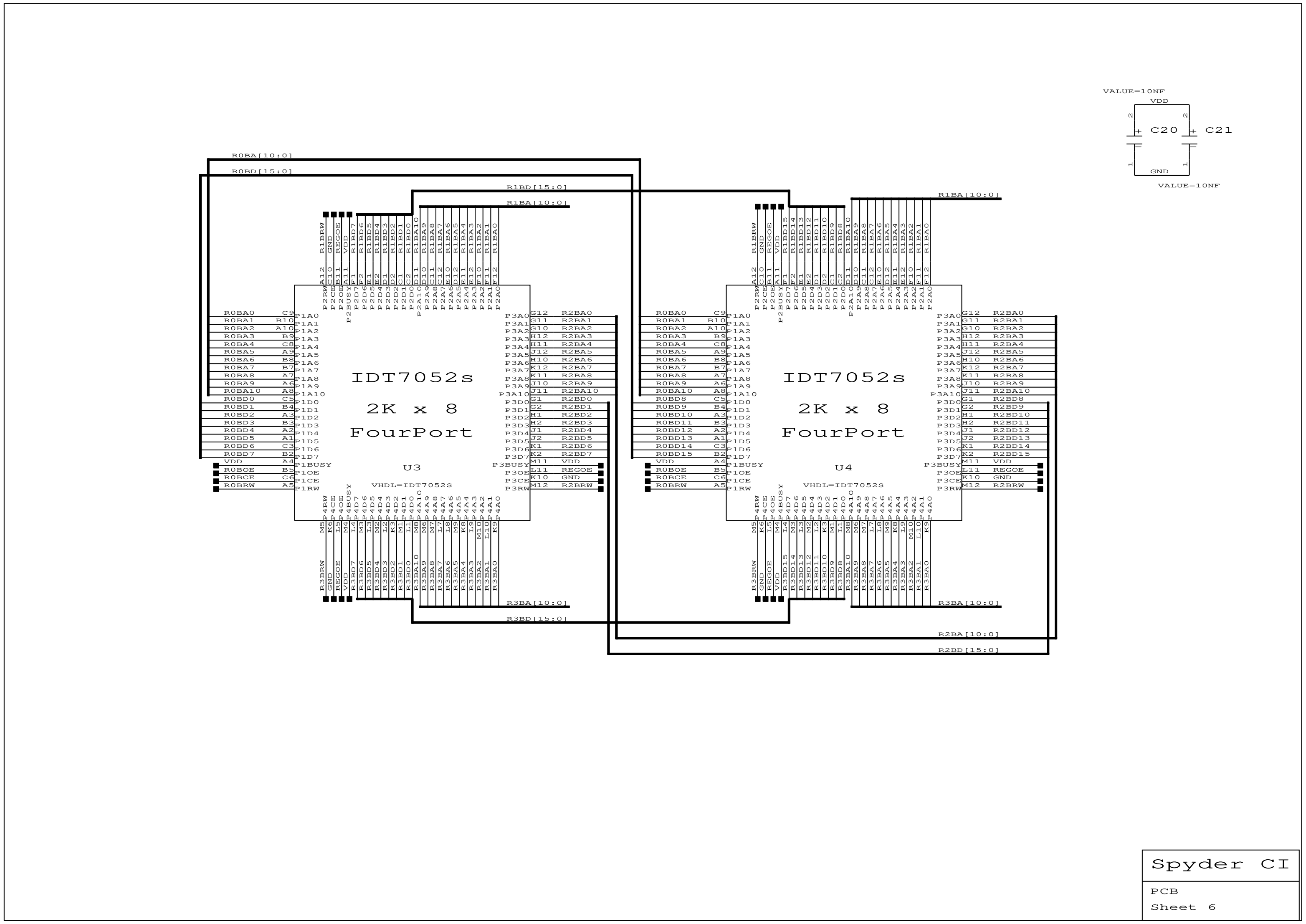

Figure A.7: Spyder register bank R^b

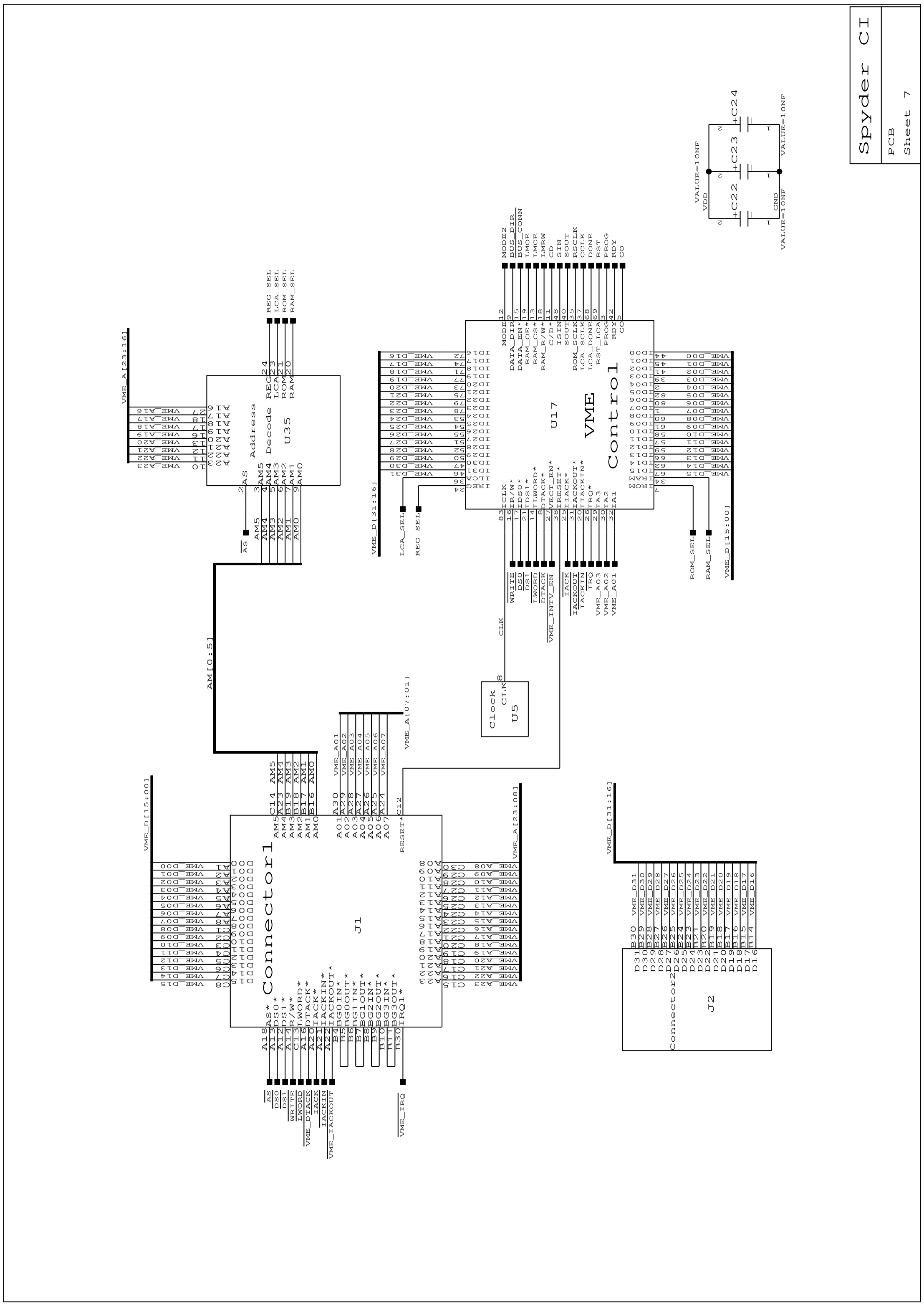

Figure A.8: Spyder VME interface and controller

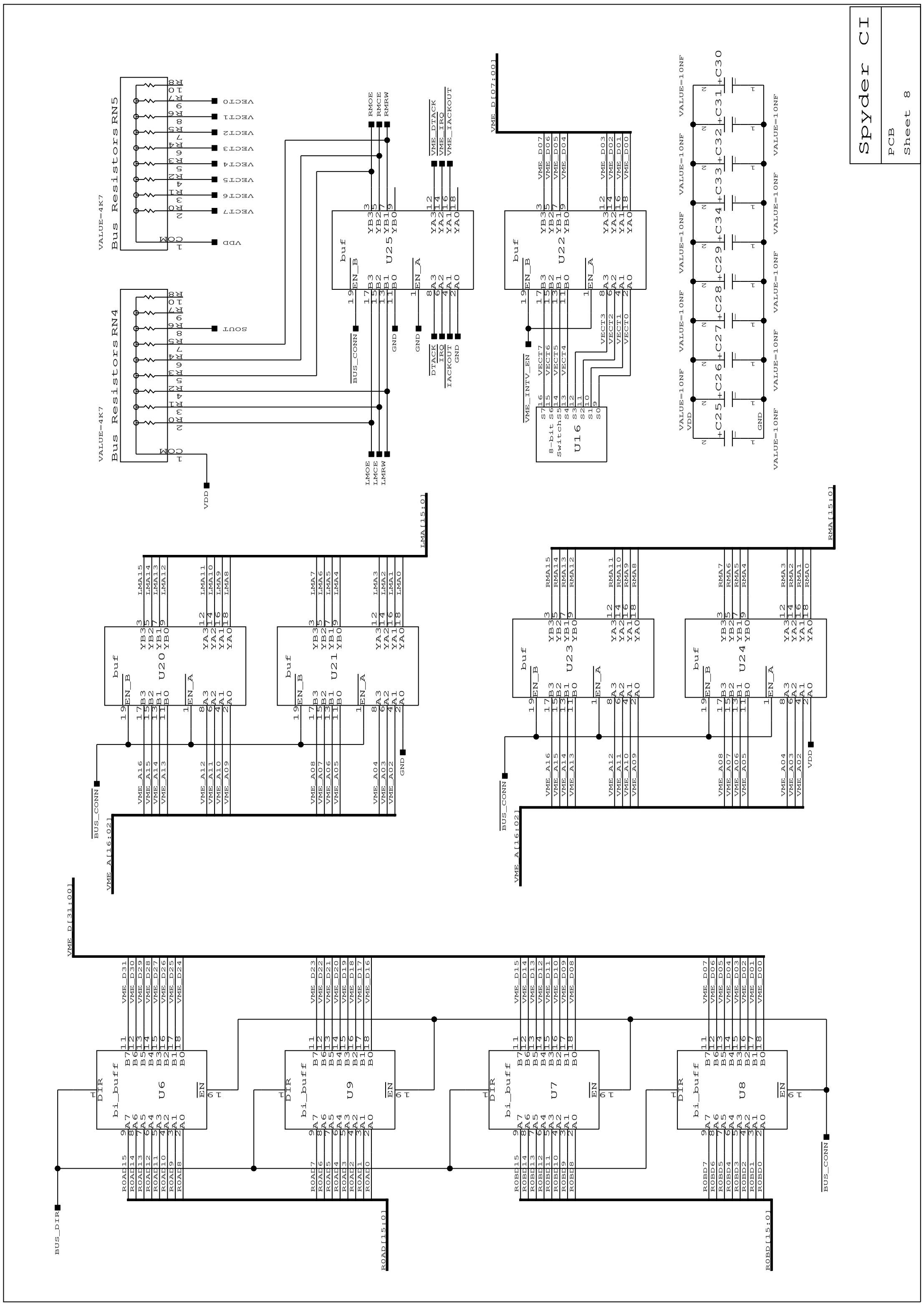

Figure A.9: Spyder VME bus and line drivers

Appendix B

Sequencer Schematic

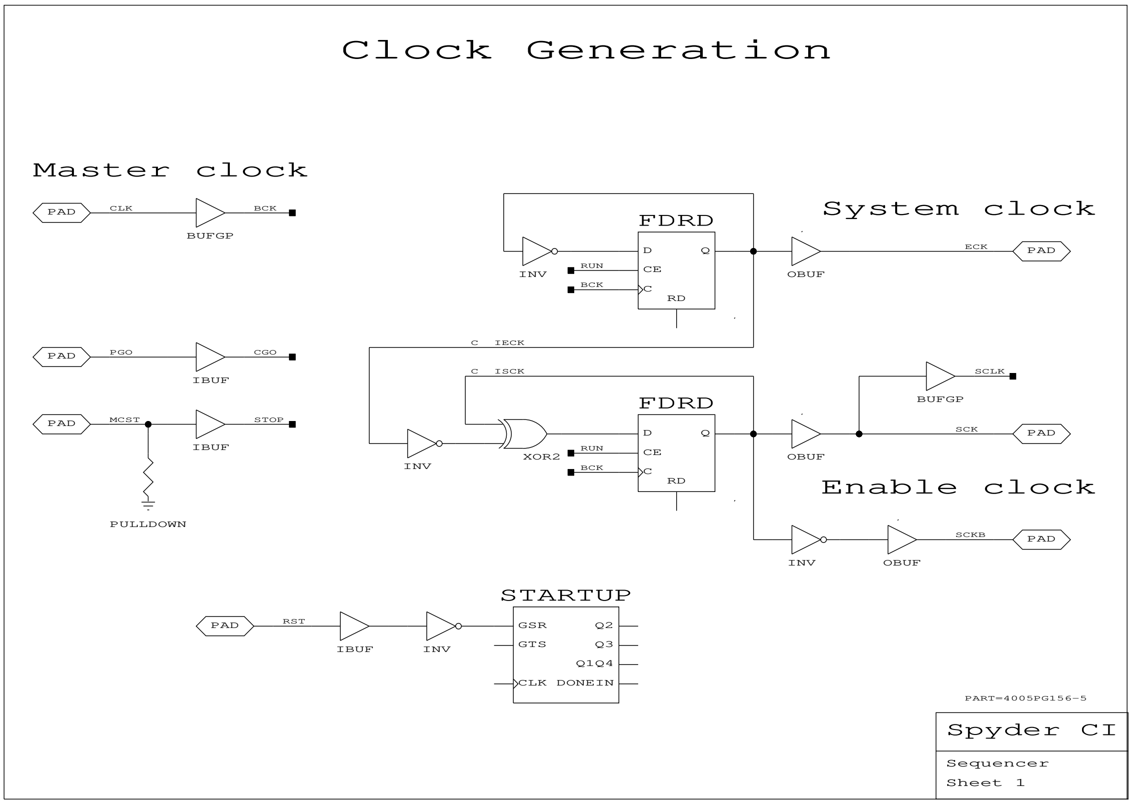

Figure B.1: System clock generation

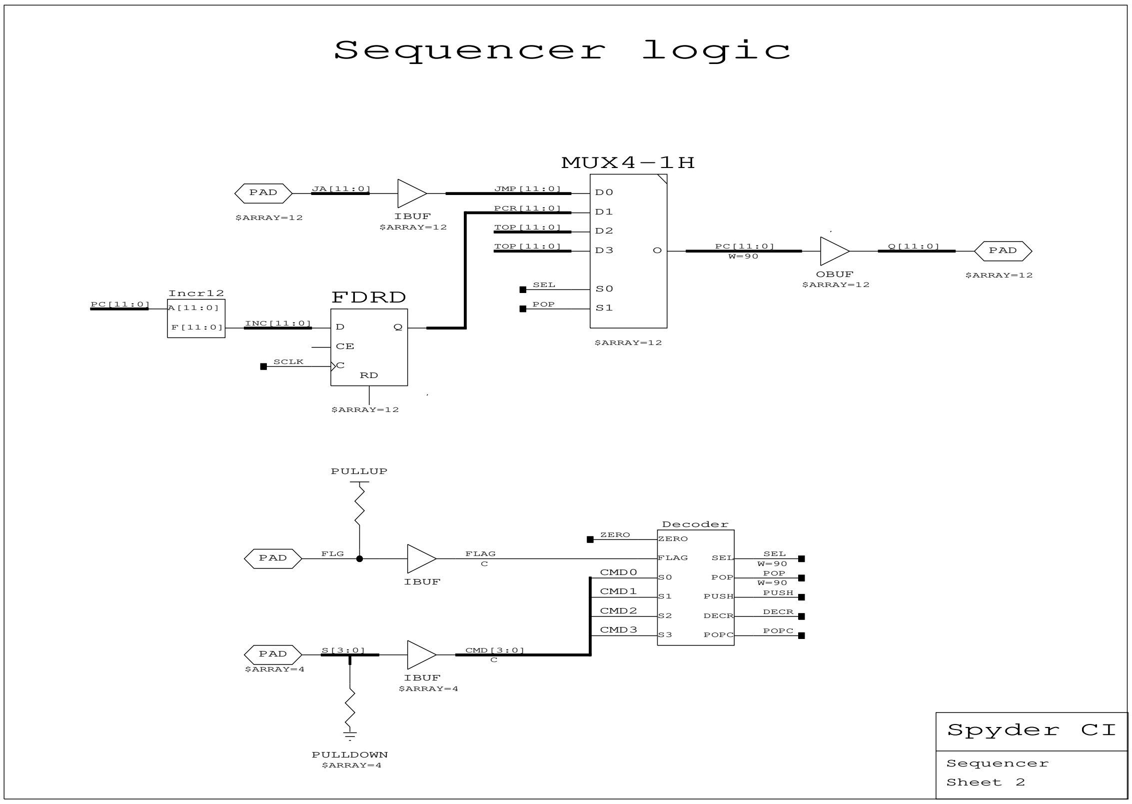

Figure B.2: Sequencer logic core

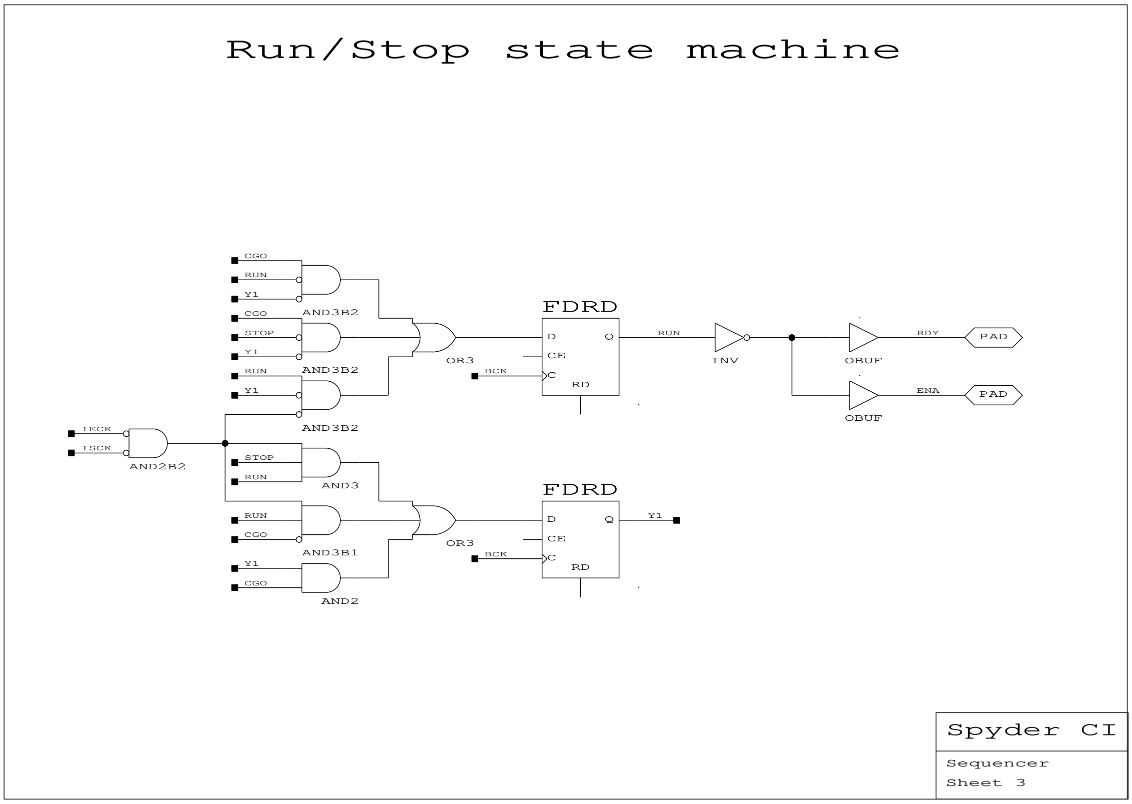

Figure B.3: Run/Stop state control

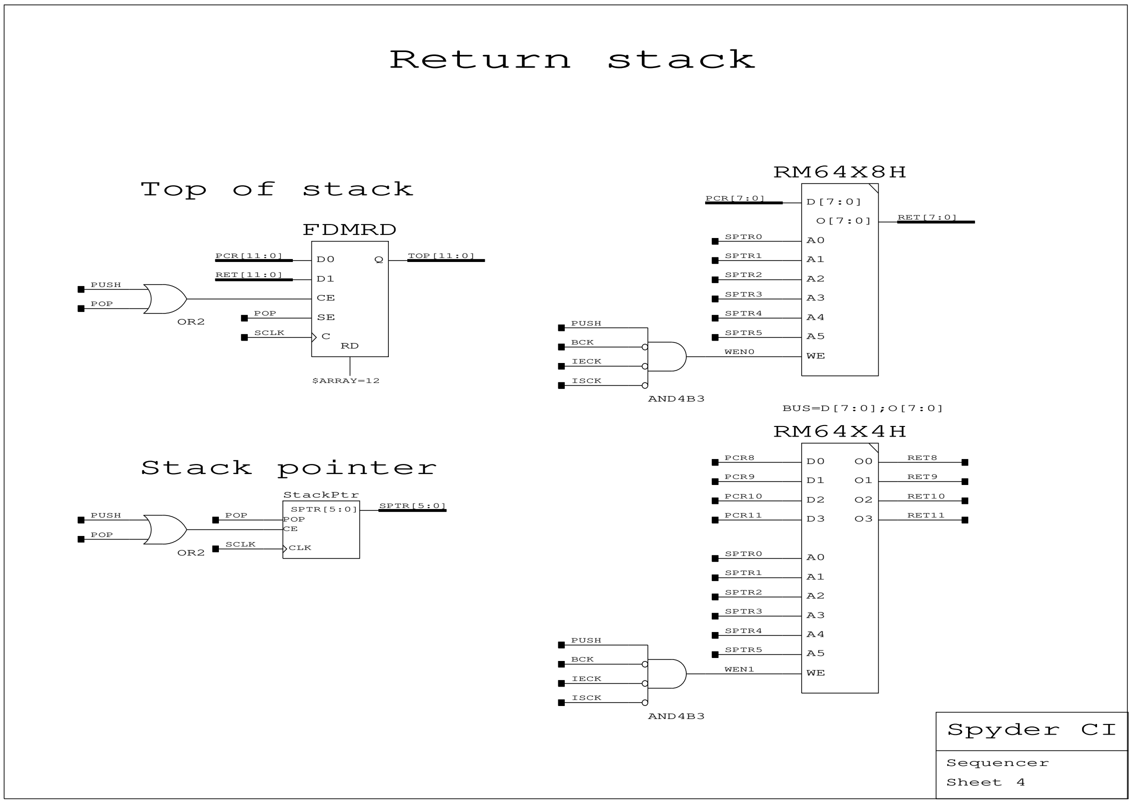

Figure B.4: Return address stack

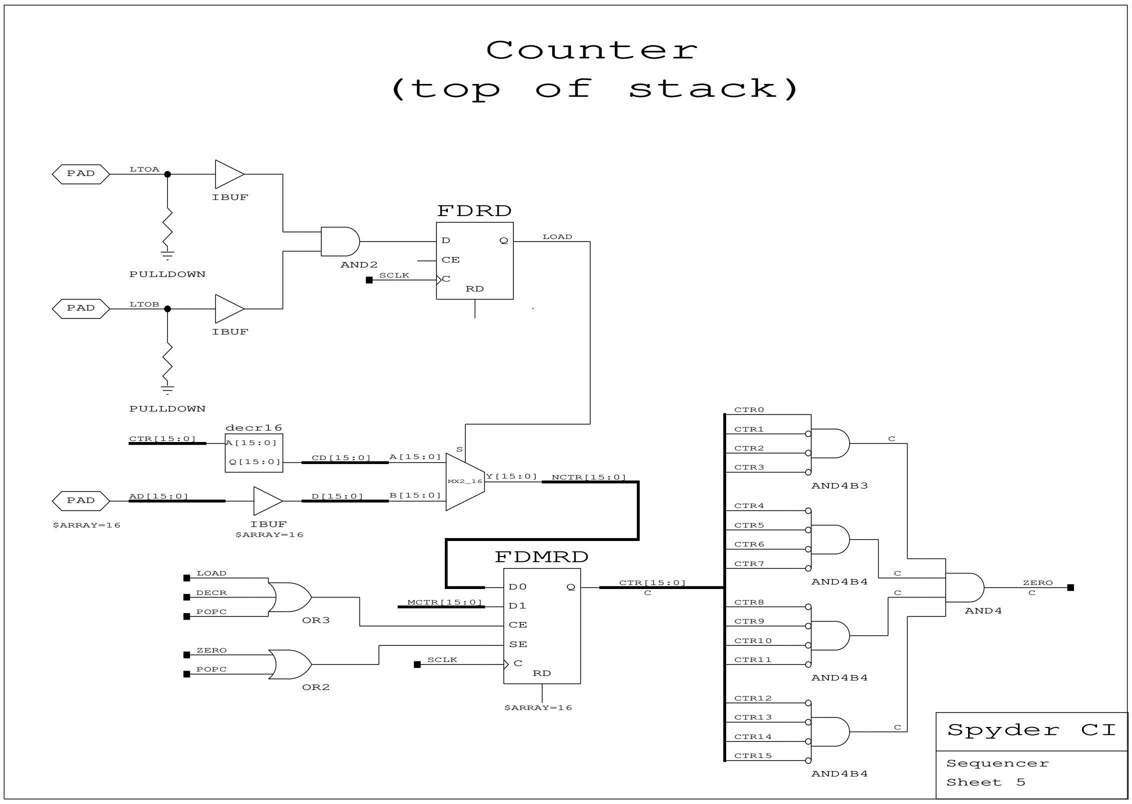

Figure B.5: Counter core

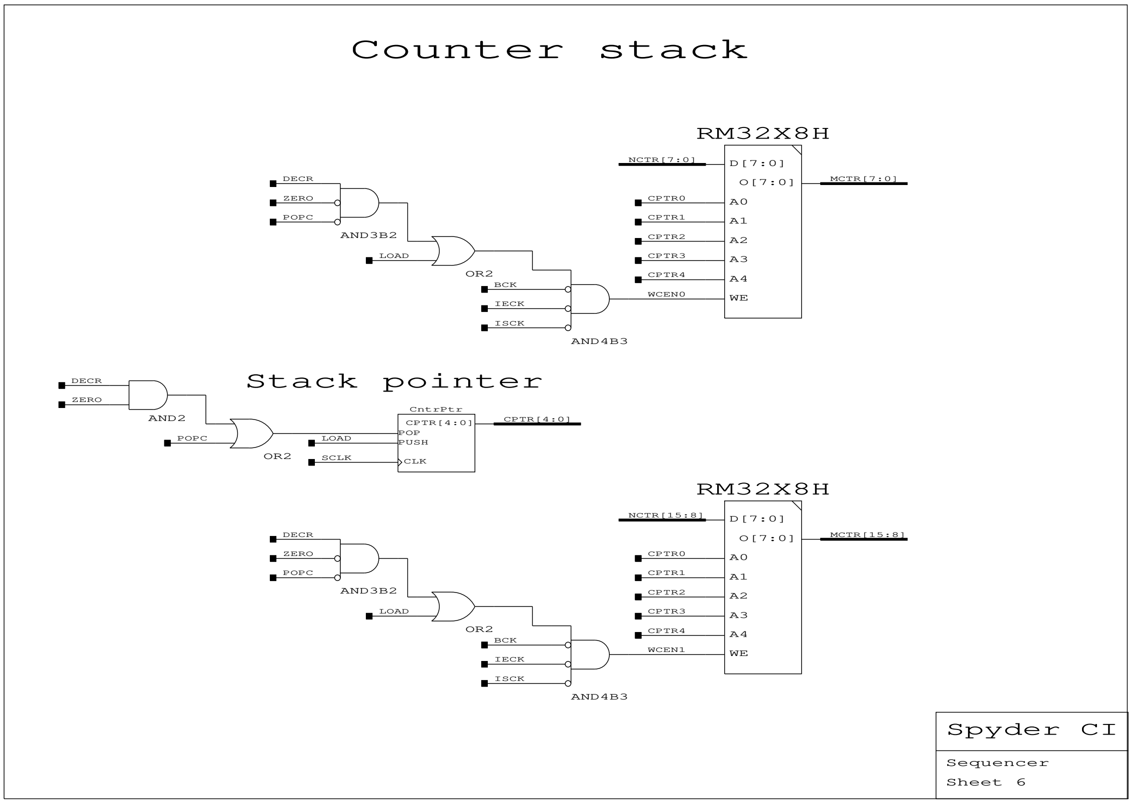

Figure B.6: Counter stack

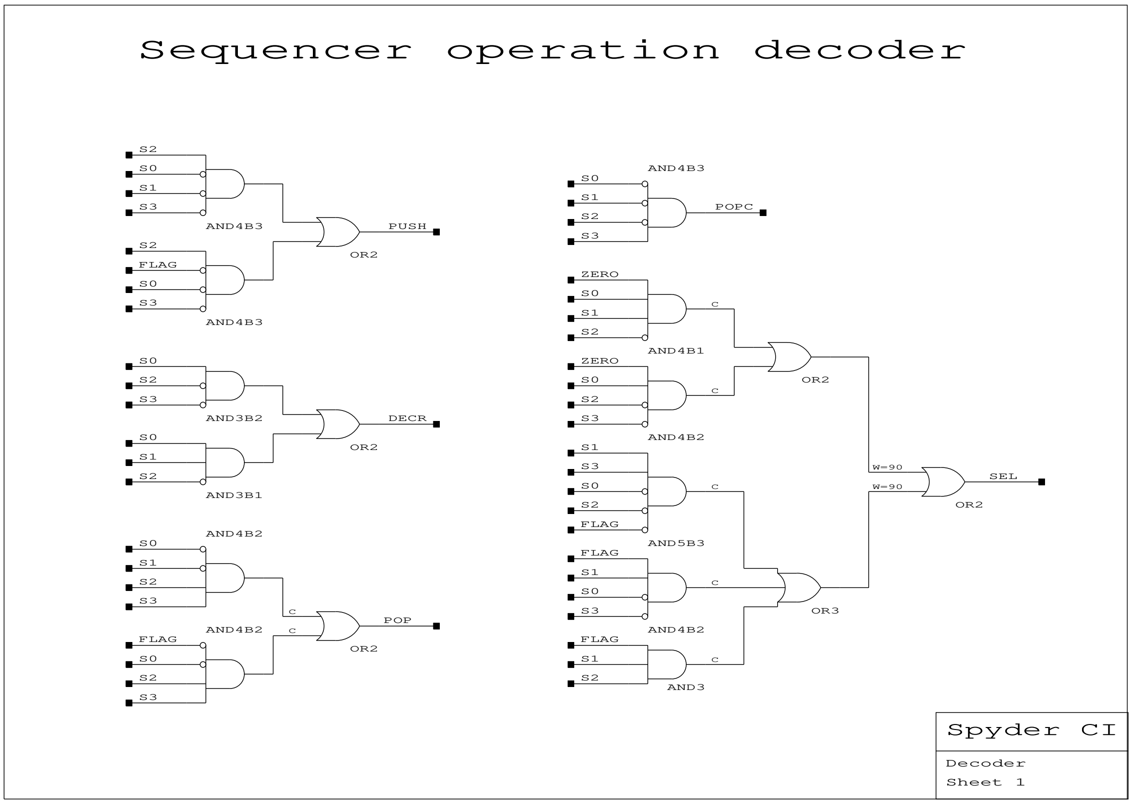

Figure B.7: Sequencer instruction decoder

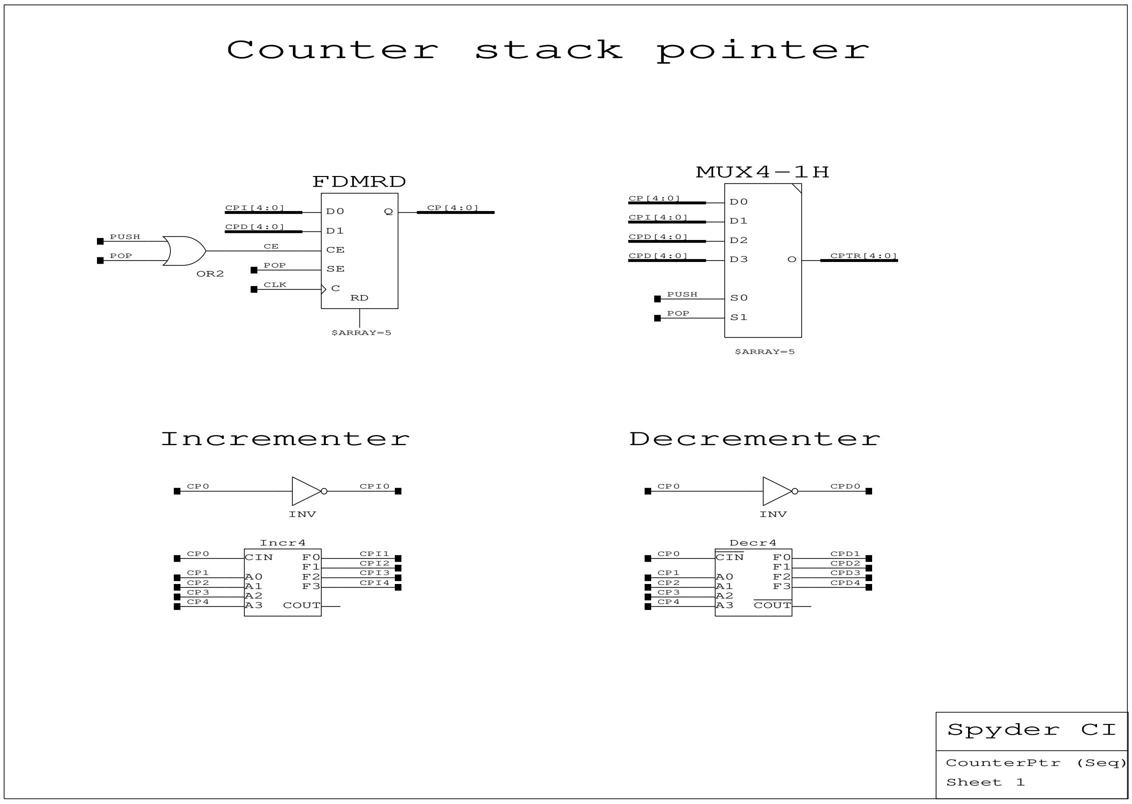

Figure B.8: Sequencer counter stack pointer

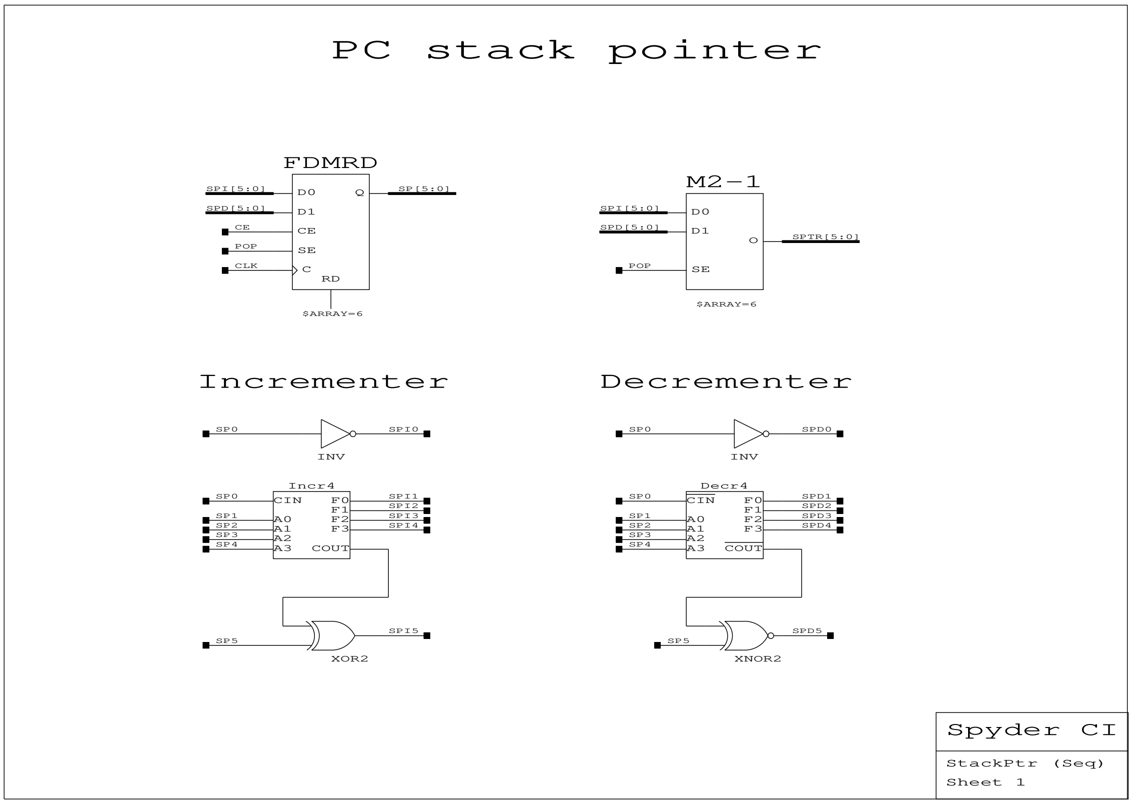

Figure B.9: Sequencer return stack pointer

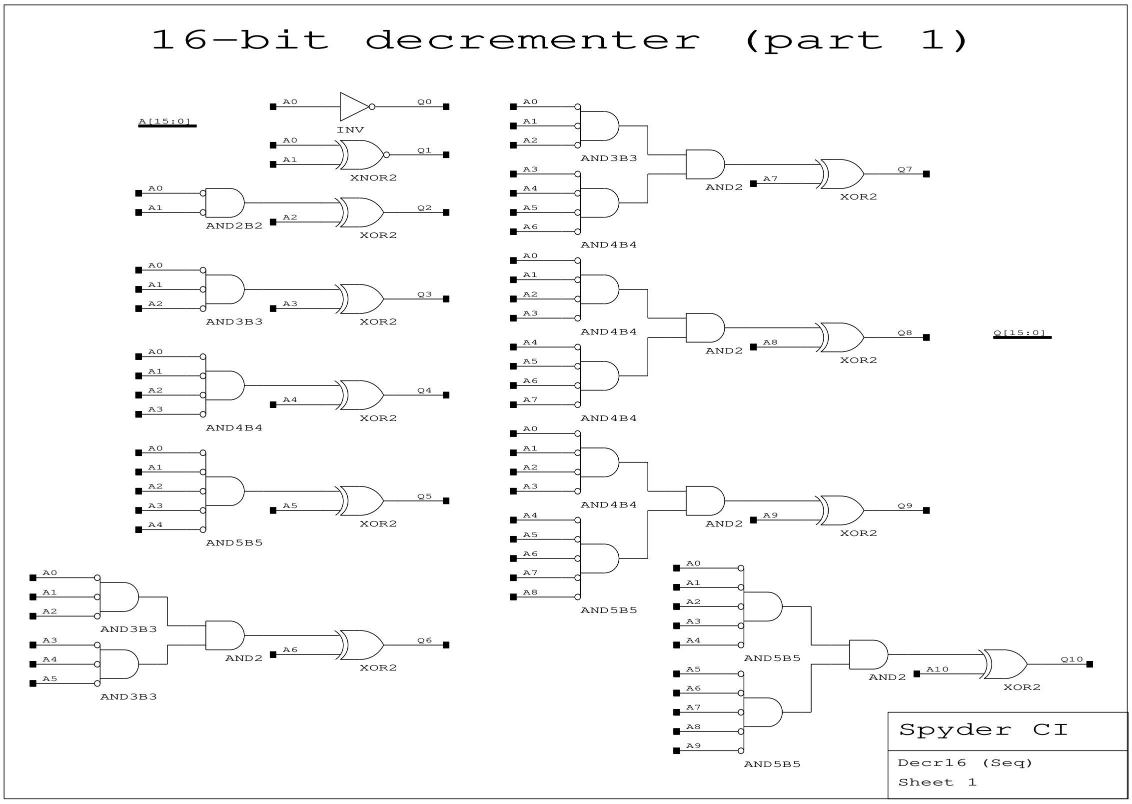

Figure B.10: Fast 16-bit decrementer, part 1

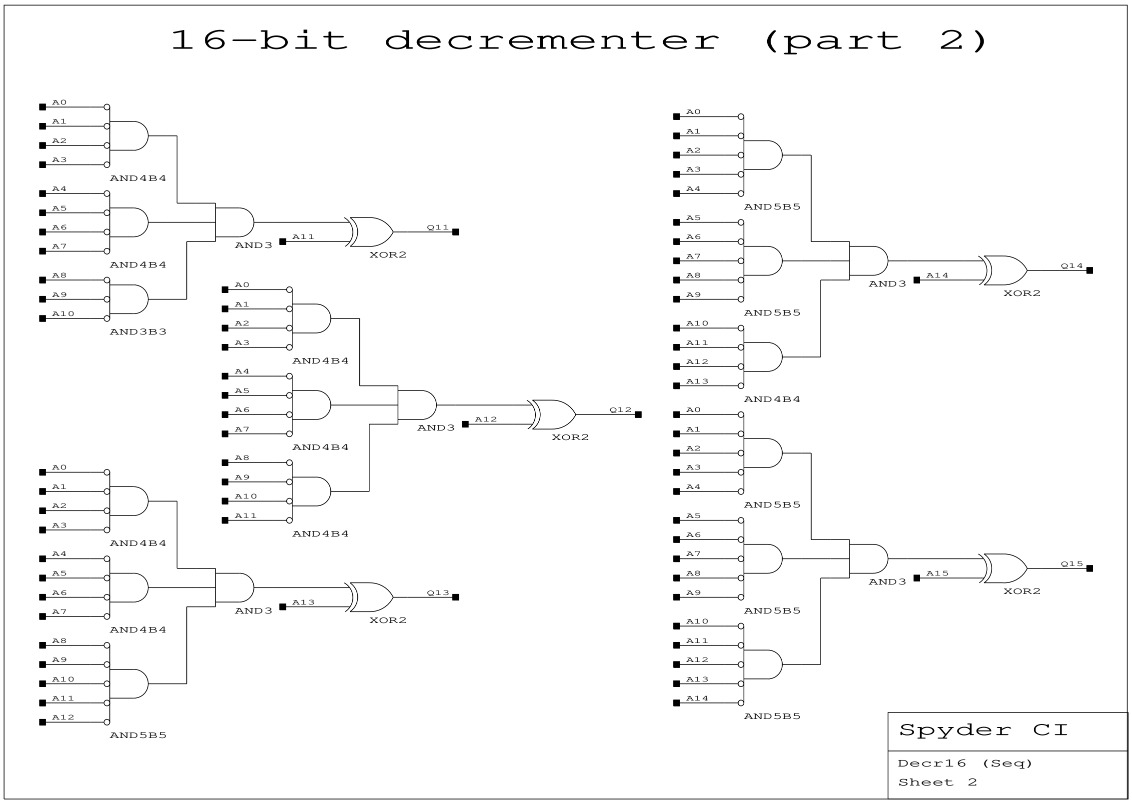

Figure B.11: Fast 16-bit decrementer, part 2

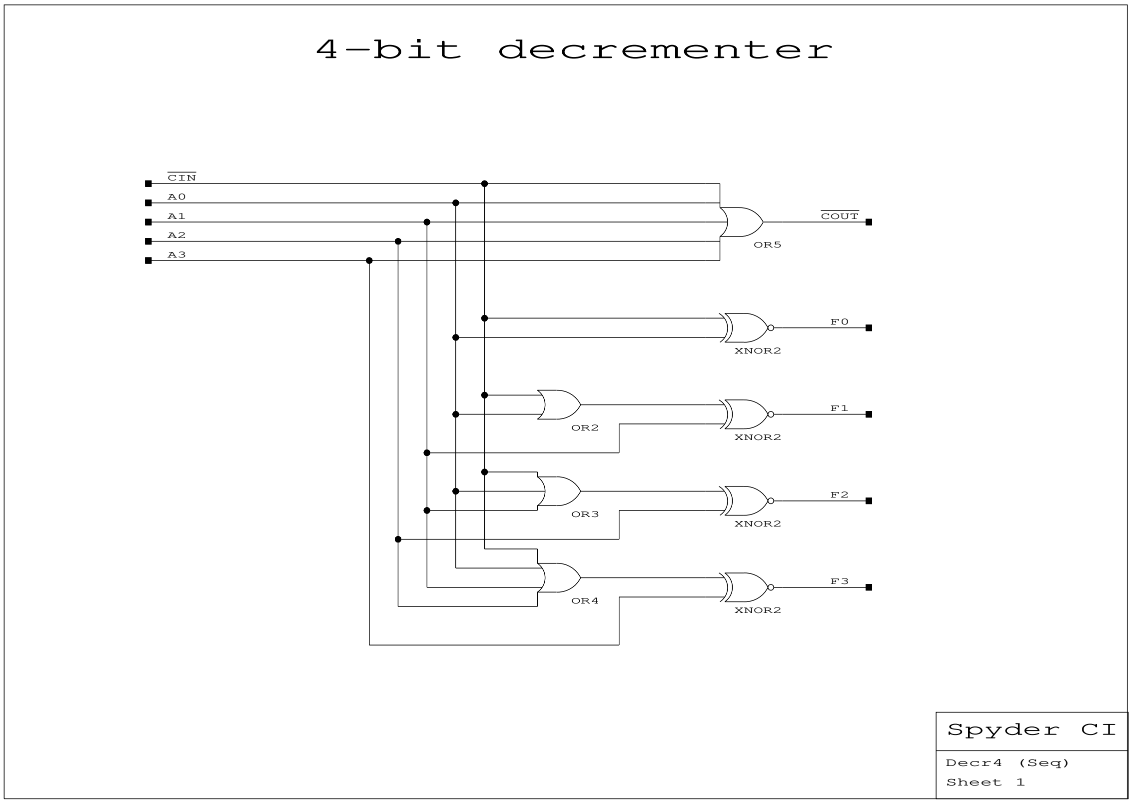

Figure B.12: Fast 4-bit decrementer

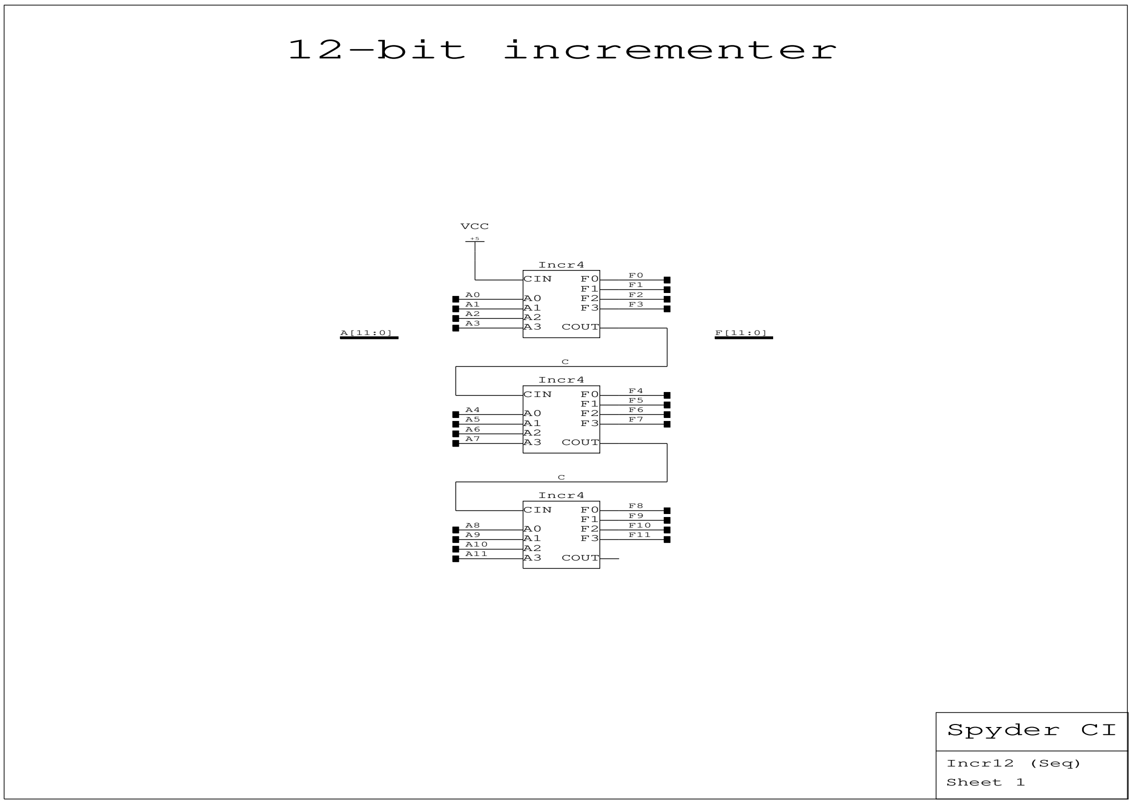

Figure B.13: Fast 12-bit incrementer

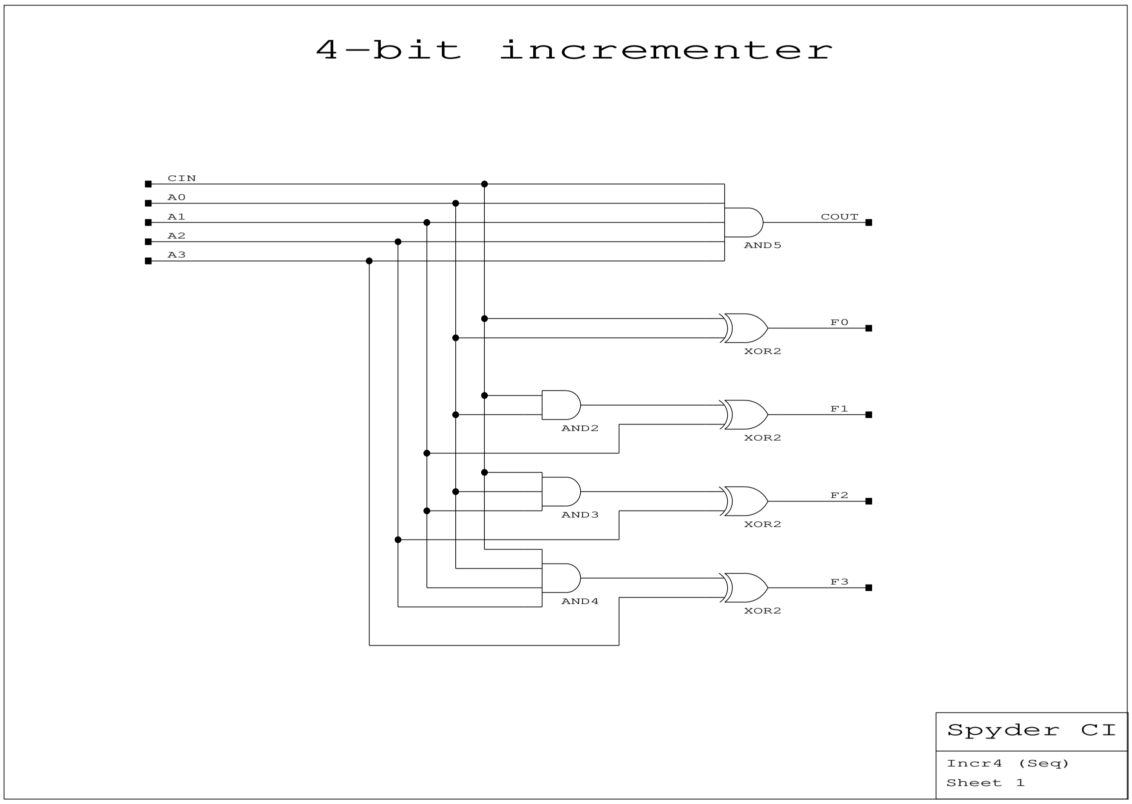

Figure B.14: Fast 4-bit incrementer

Appendix C

Load and Store Unit Schematic

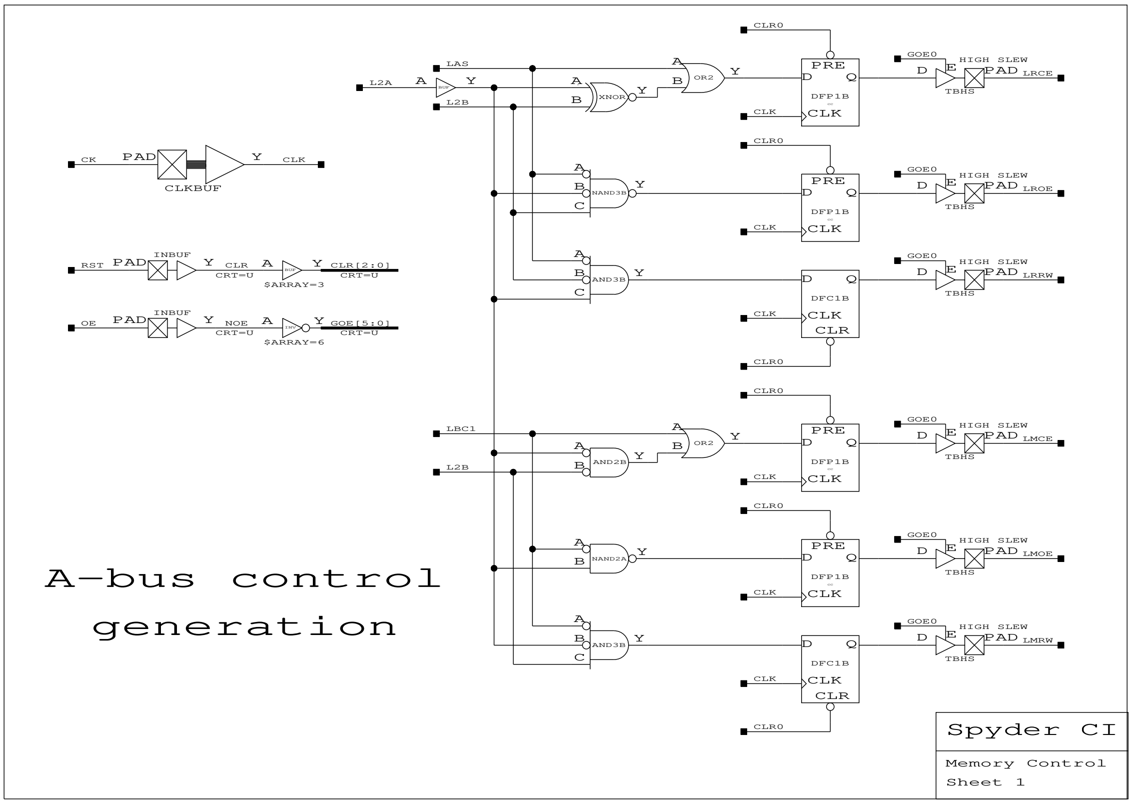

Figure C.1: Memory bus A control signals generation

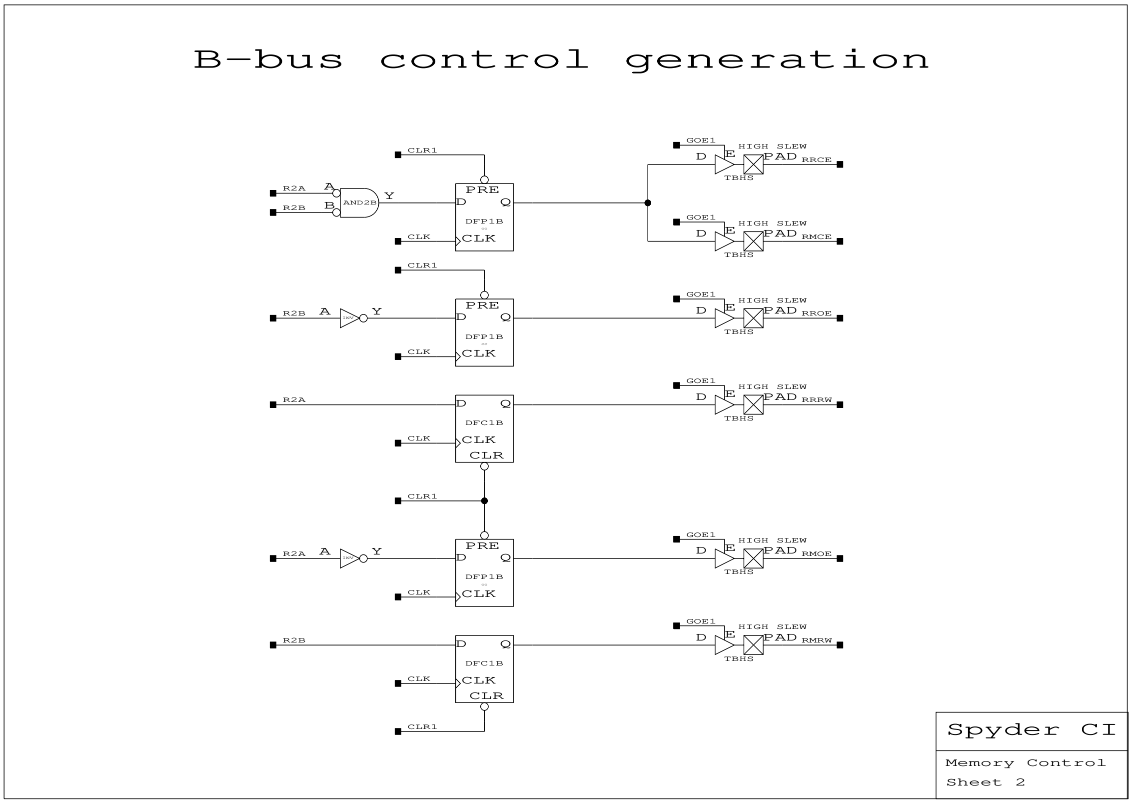

Figure C.2: Memory bus B control signals generation

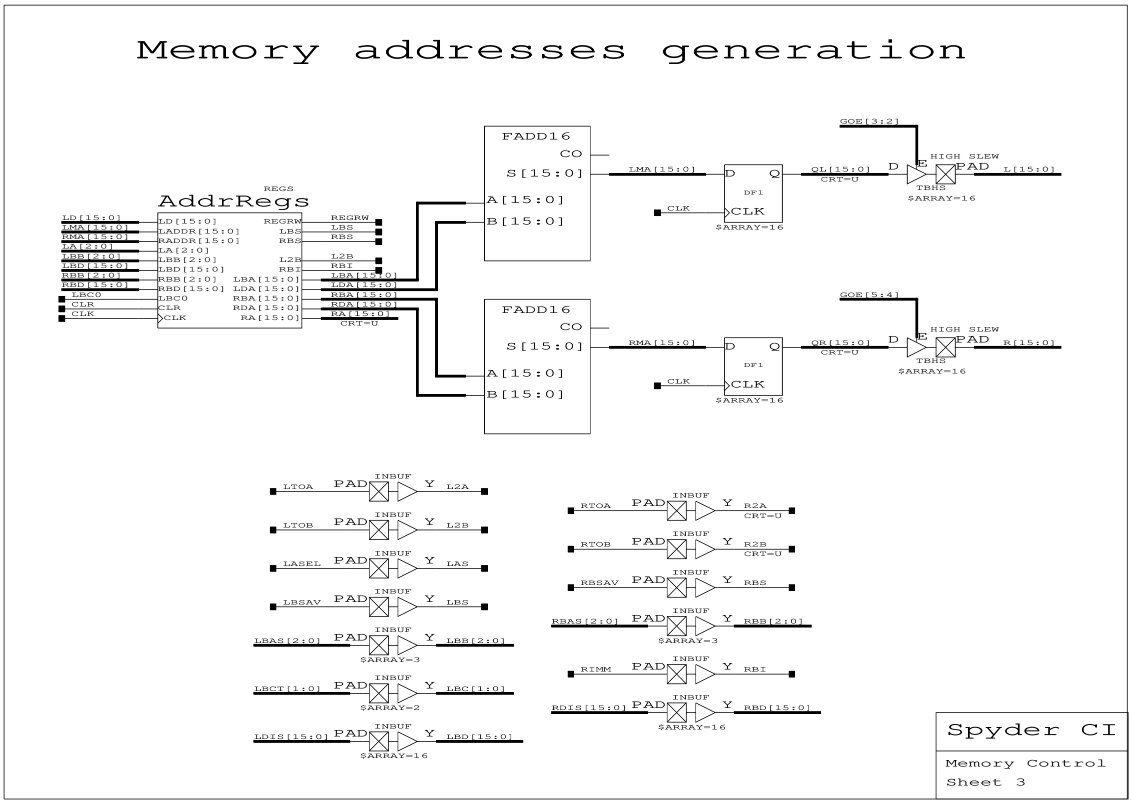

Figure C.3: Memory addresses generation

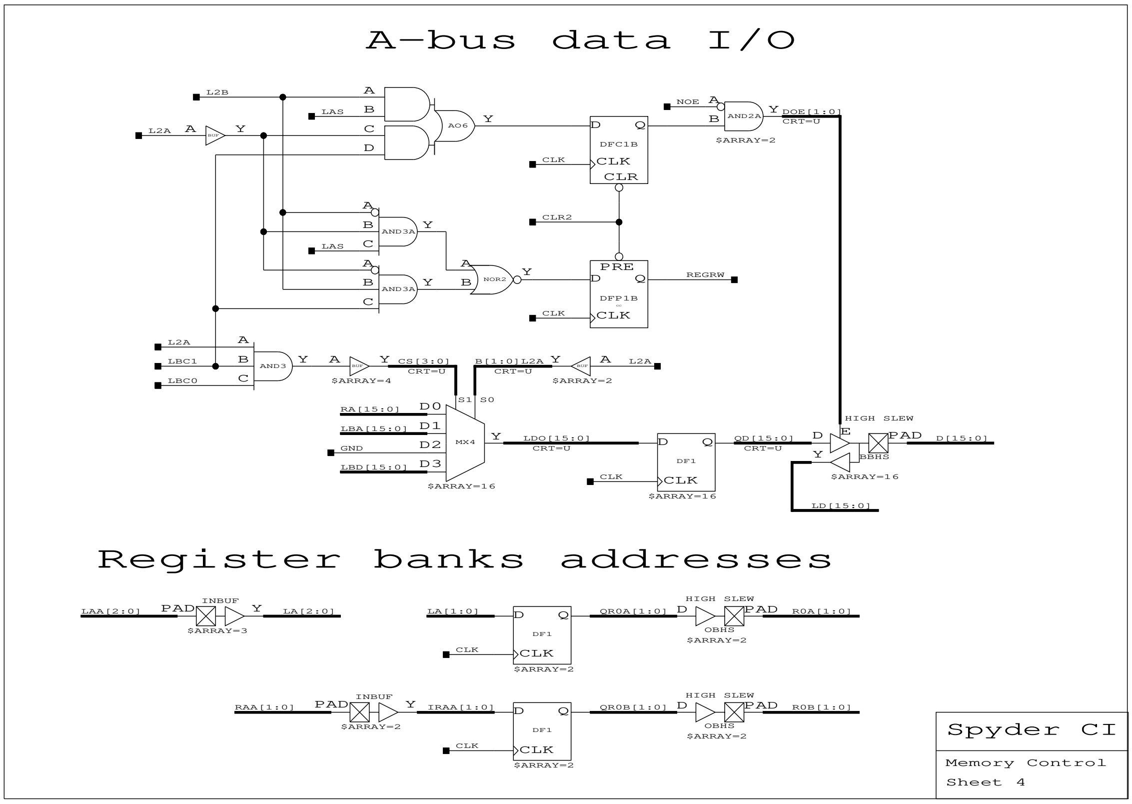

Figure C.4: Bus A data generation

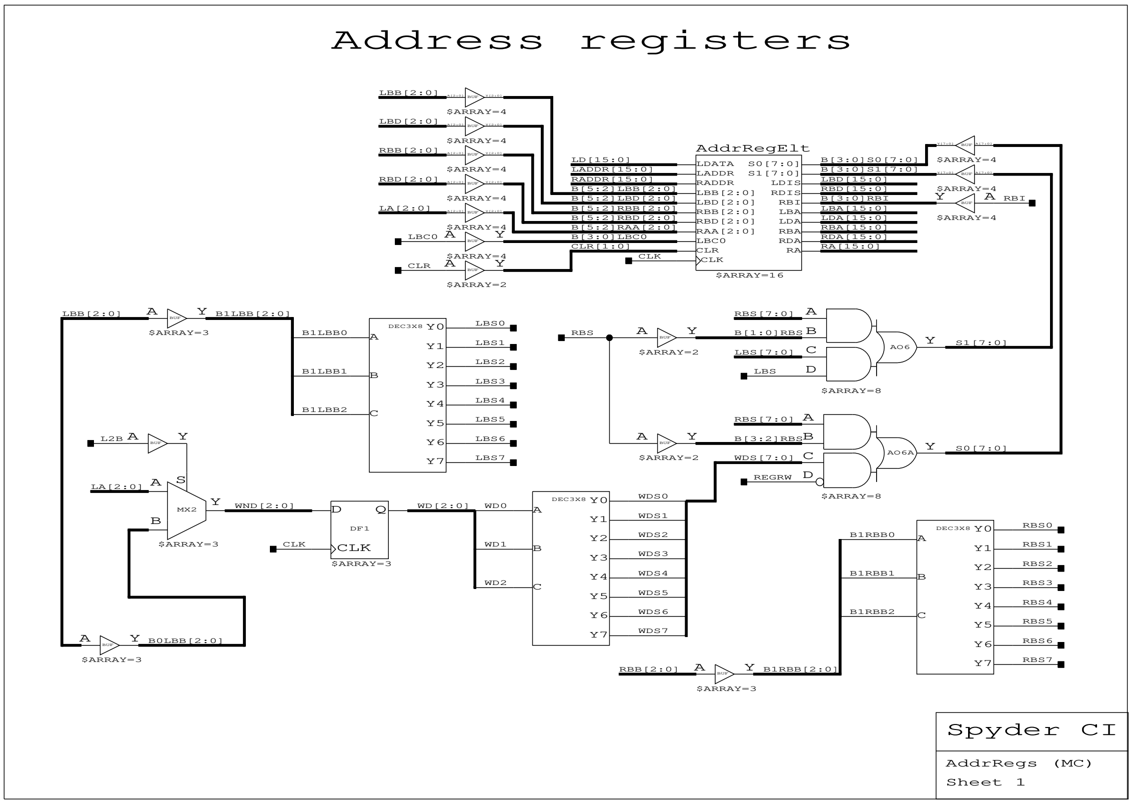

Figure C.5: 8 × 16-bit address registers

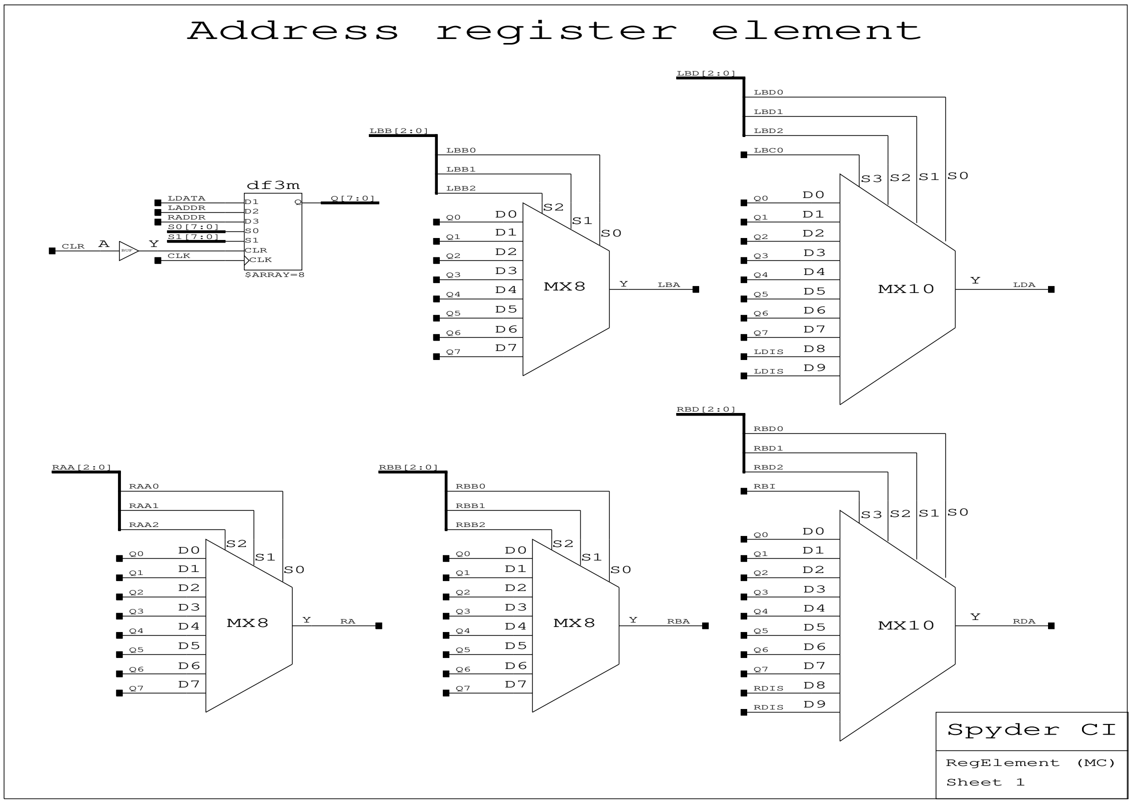

Figure C.6: 8 address registers element

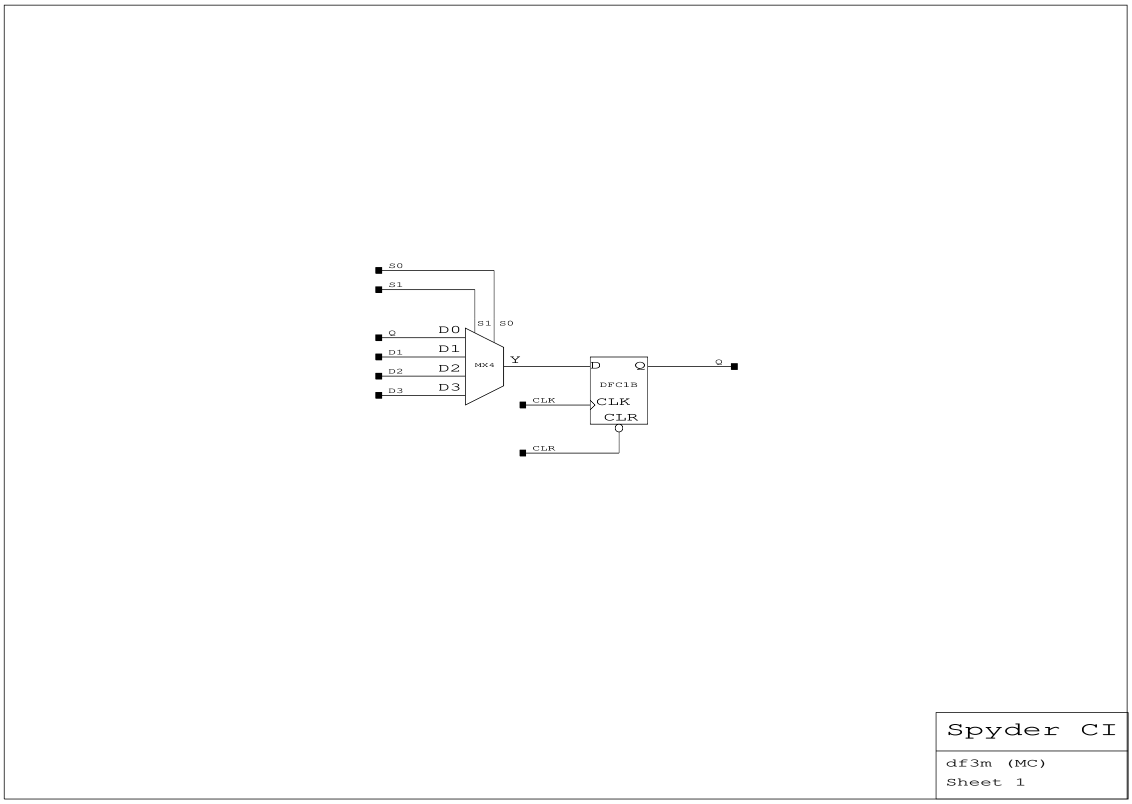

Figure C.7: Address register core element

Appendix D

Register Window Controller Schematic

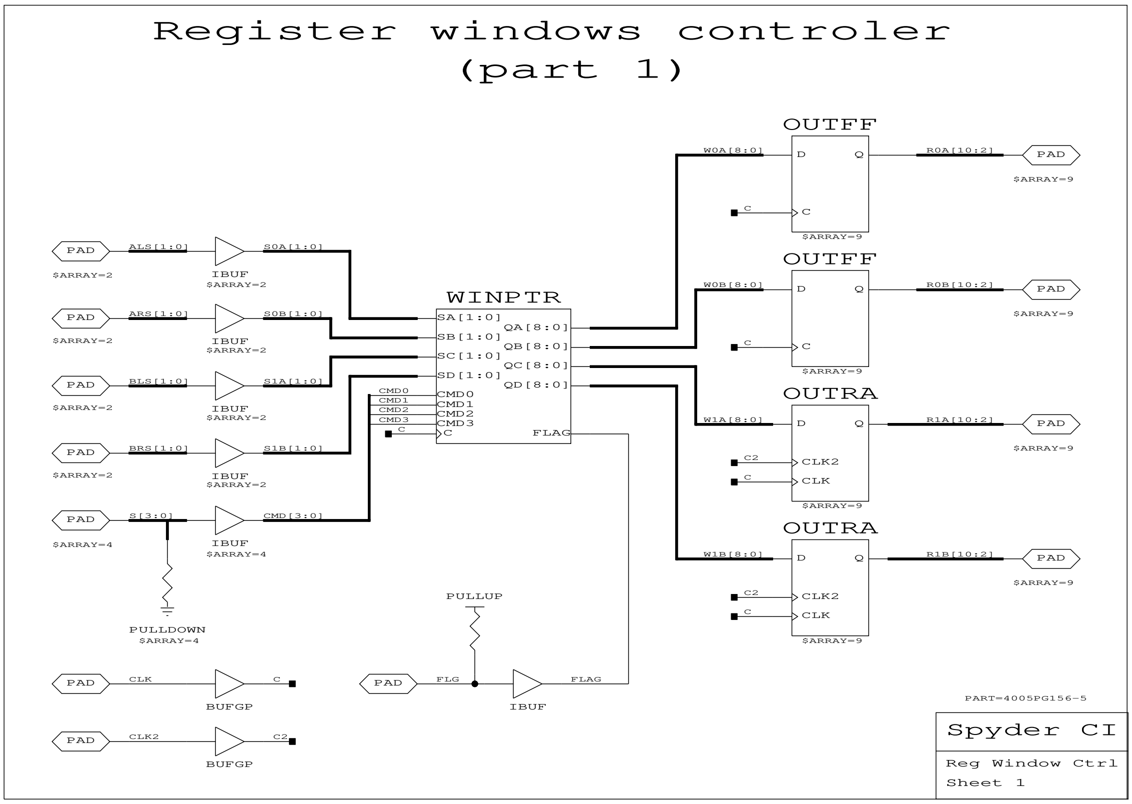

Figure D.1: Register window control, part 1

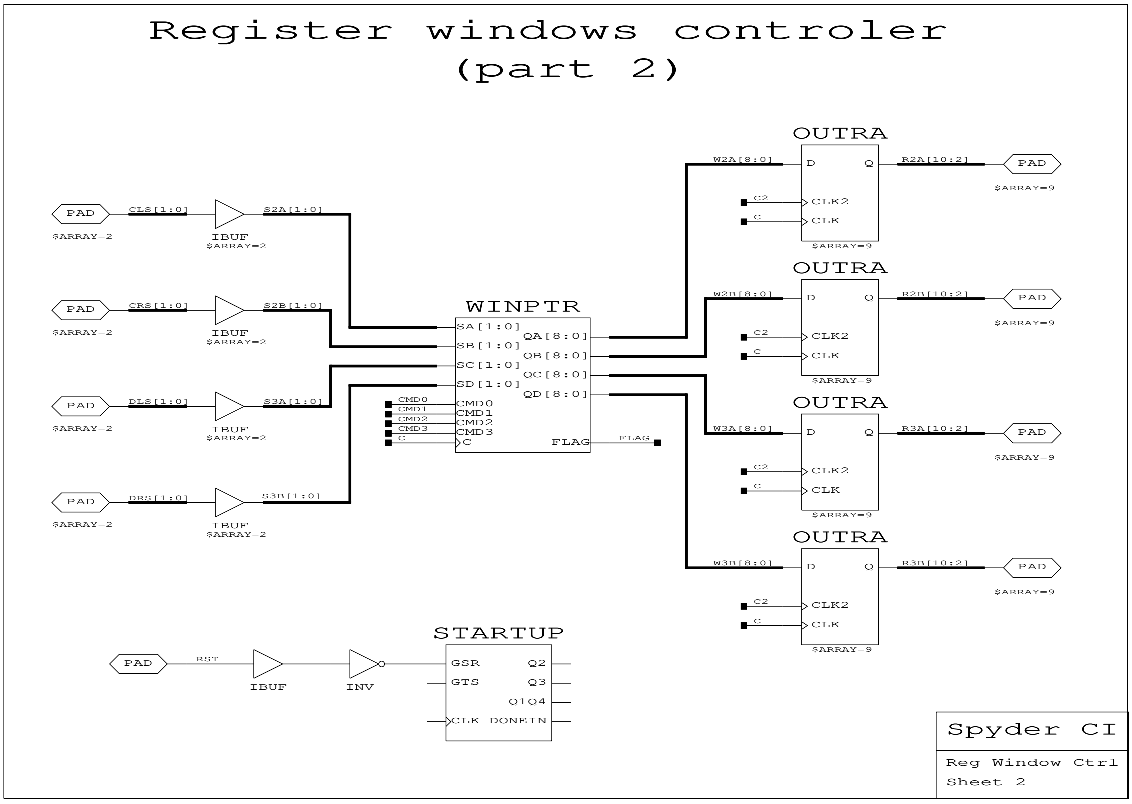

Figure D.2: Register window control, part 2

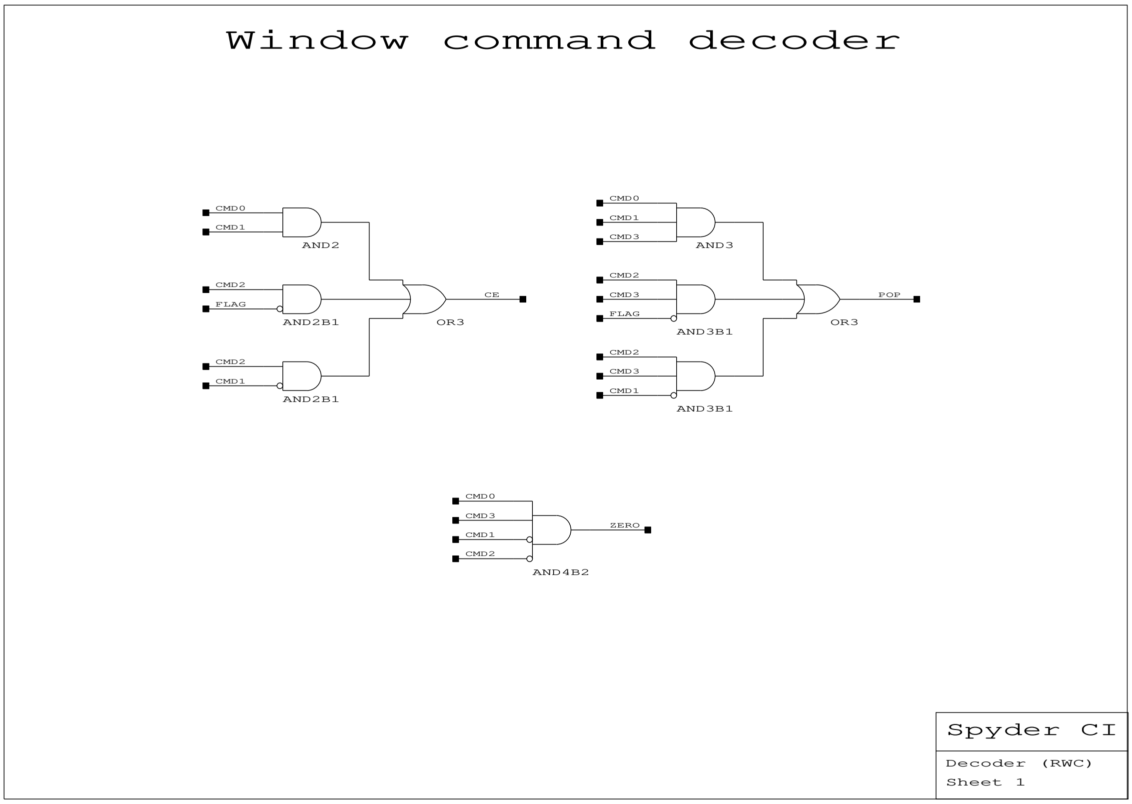

Figure D.3: Window command decoder

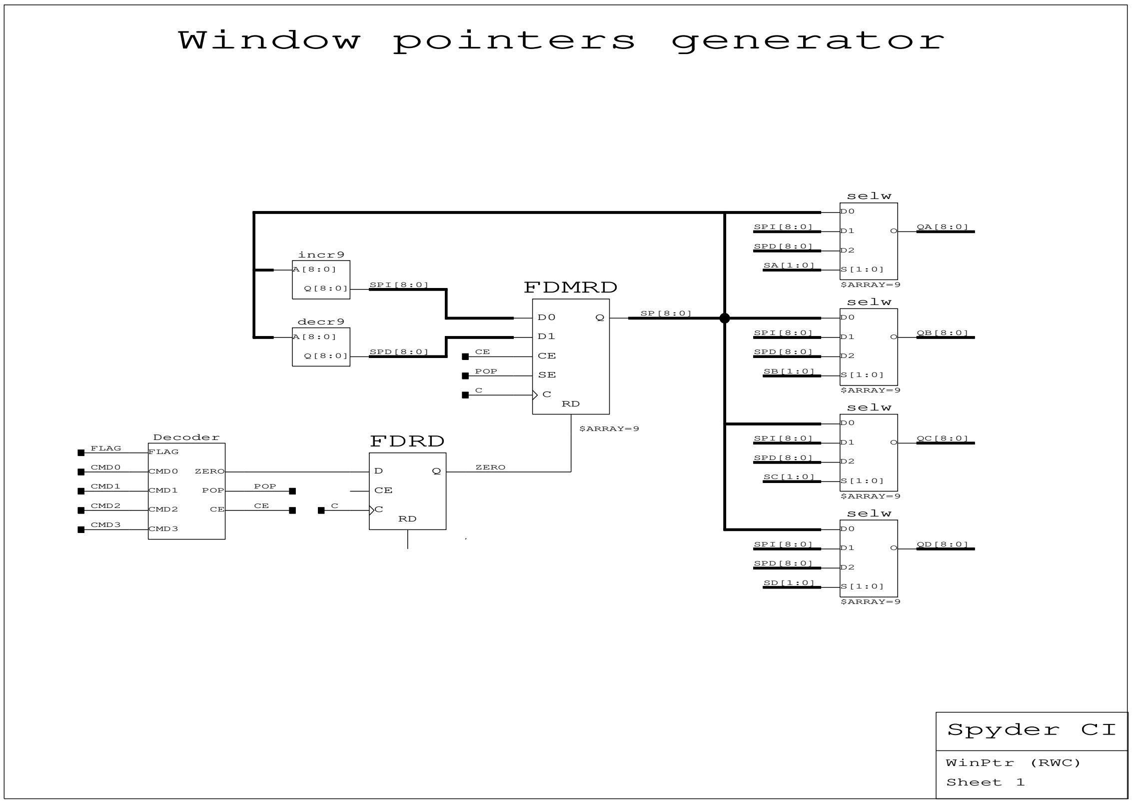

Figure D.4: Window pointers generator

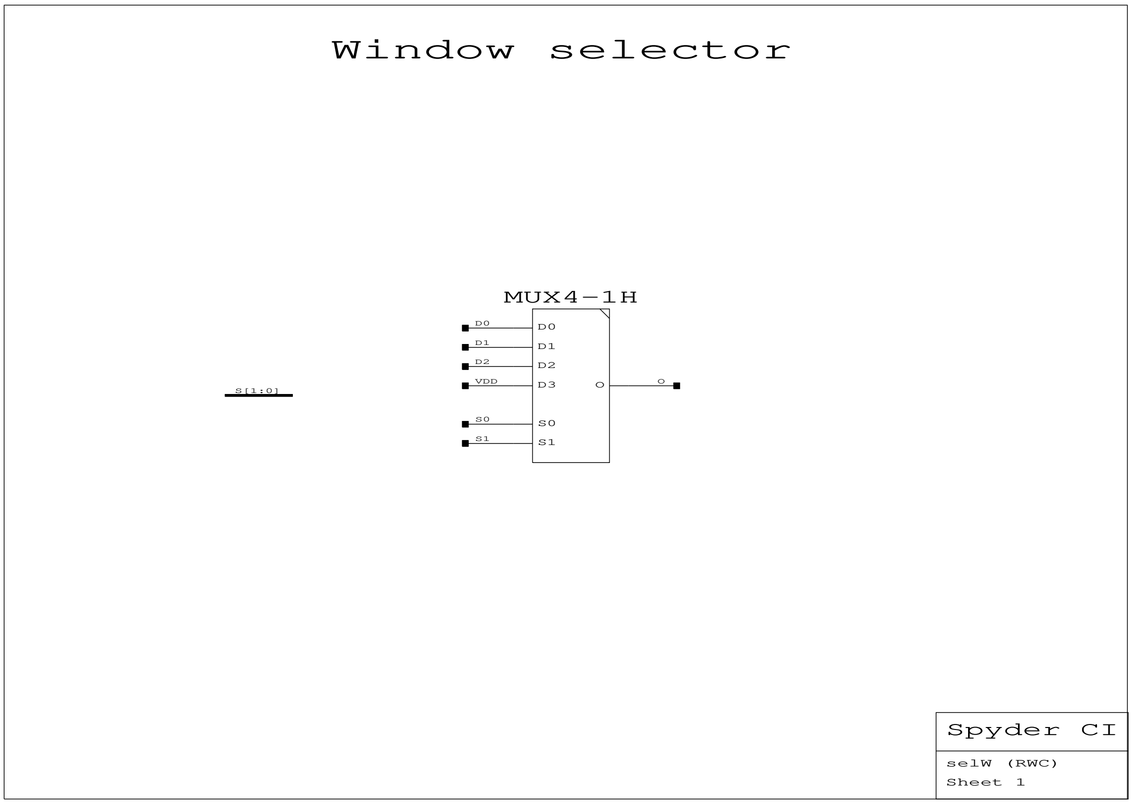

Figure D.5: Window selector

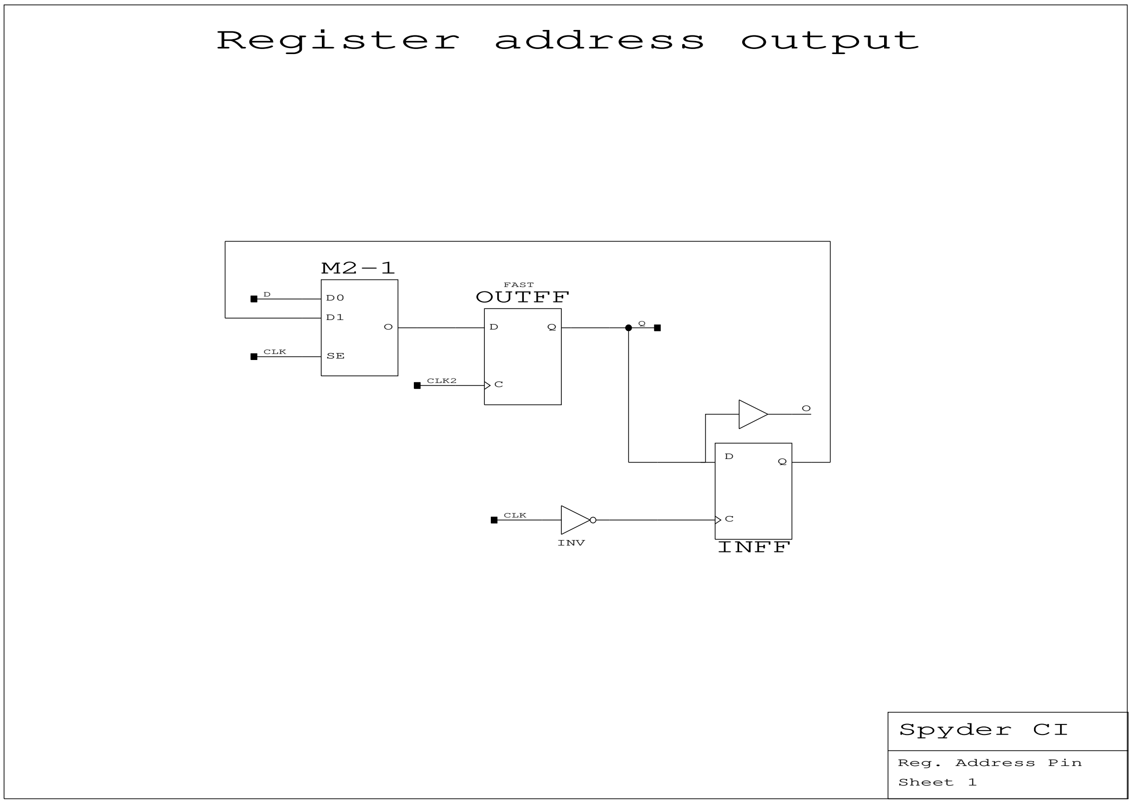

Figure D.6: Register address output pad

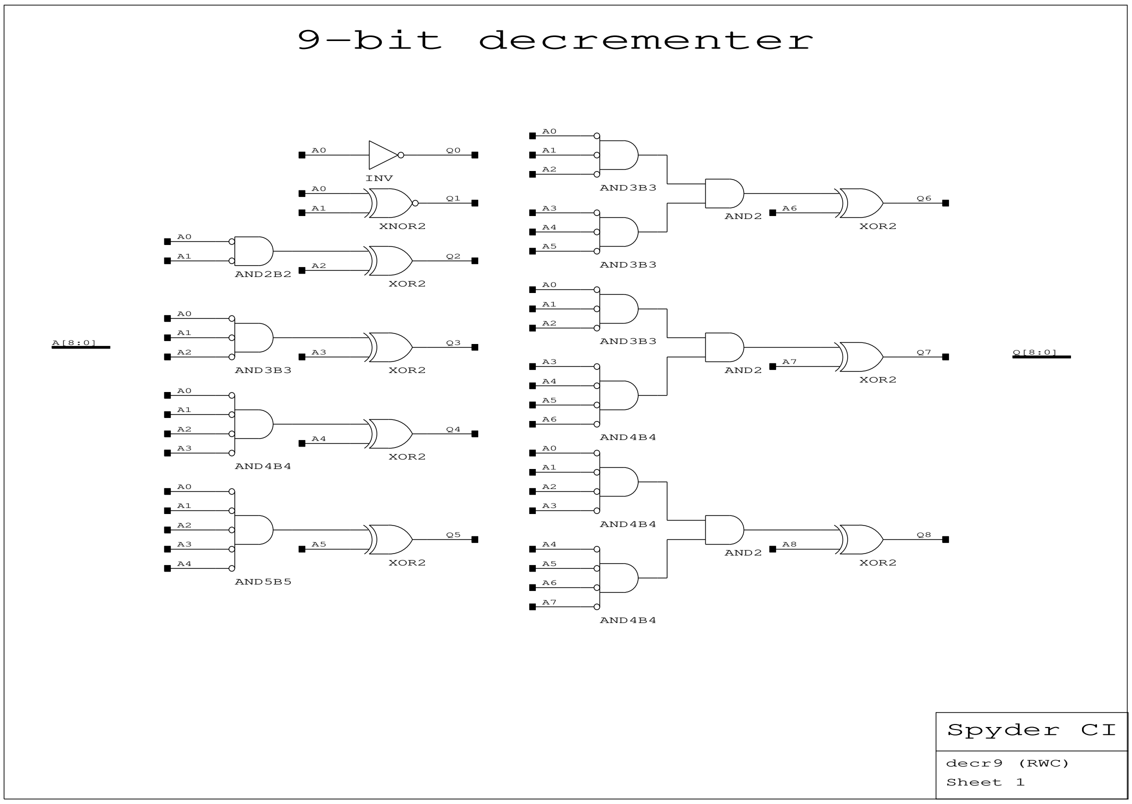

Figure D.7: Fast 9-bit decrementer

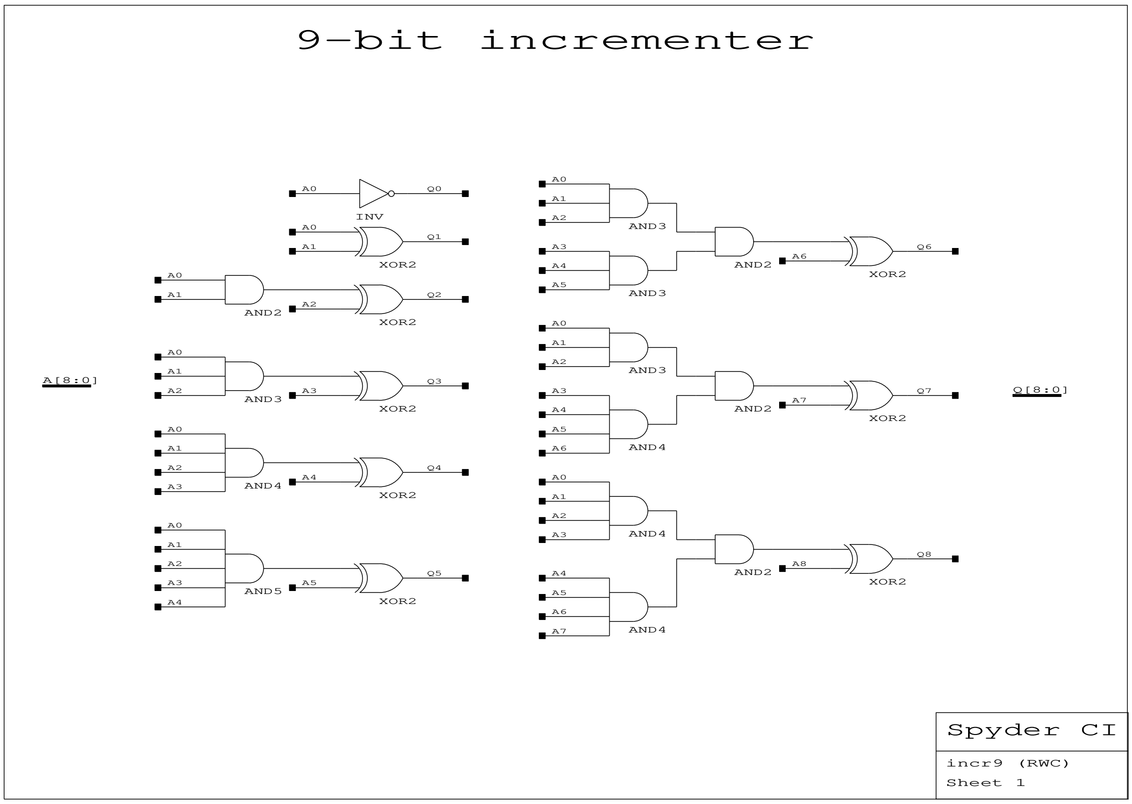

Figure D.8: Fast 9-bit incrementer

Appendix E

Execution Unit Schematic

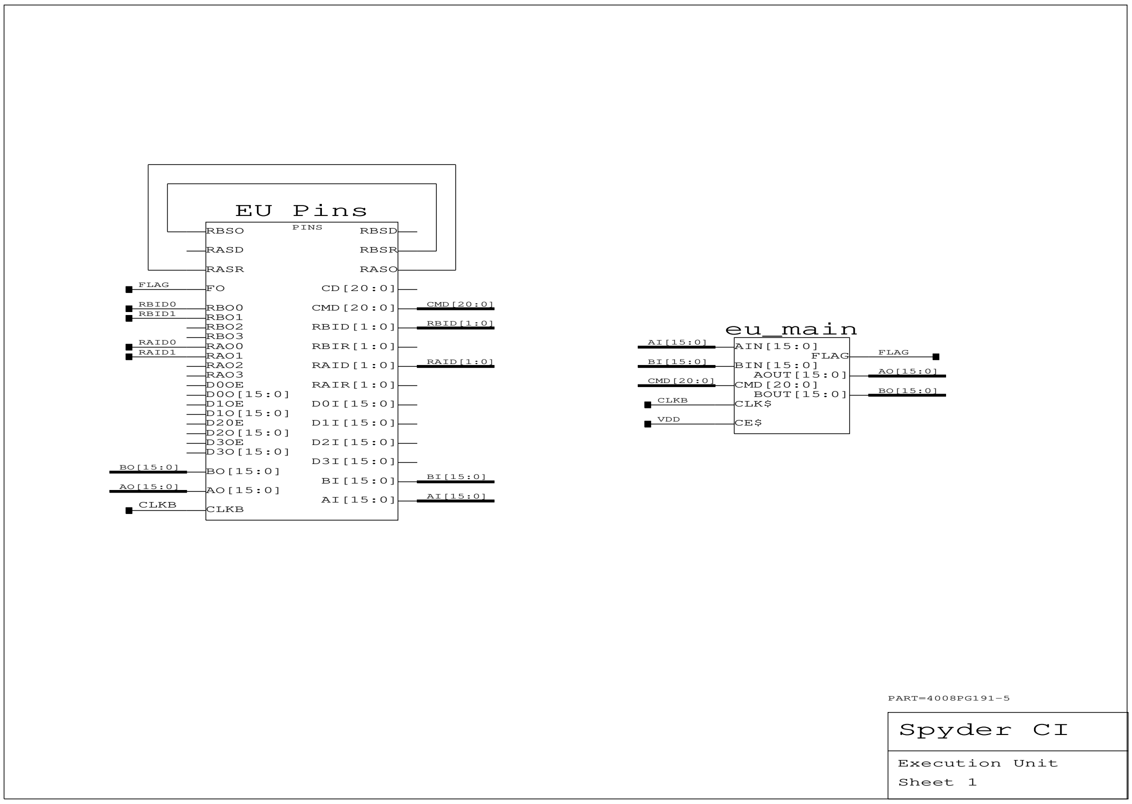

Figure E.1: Typical execution unit schematic

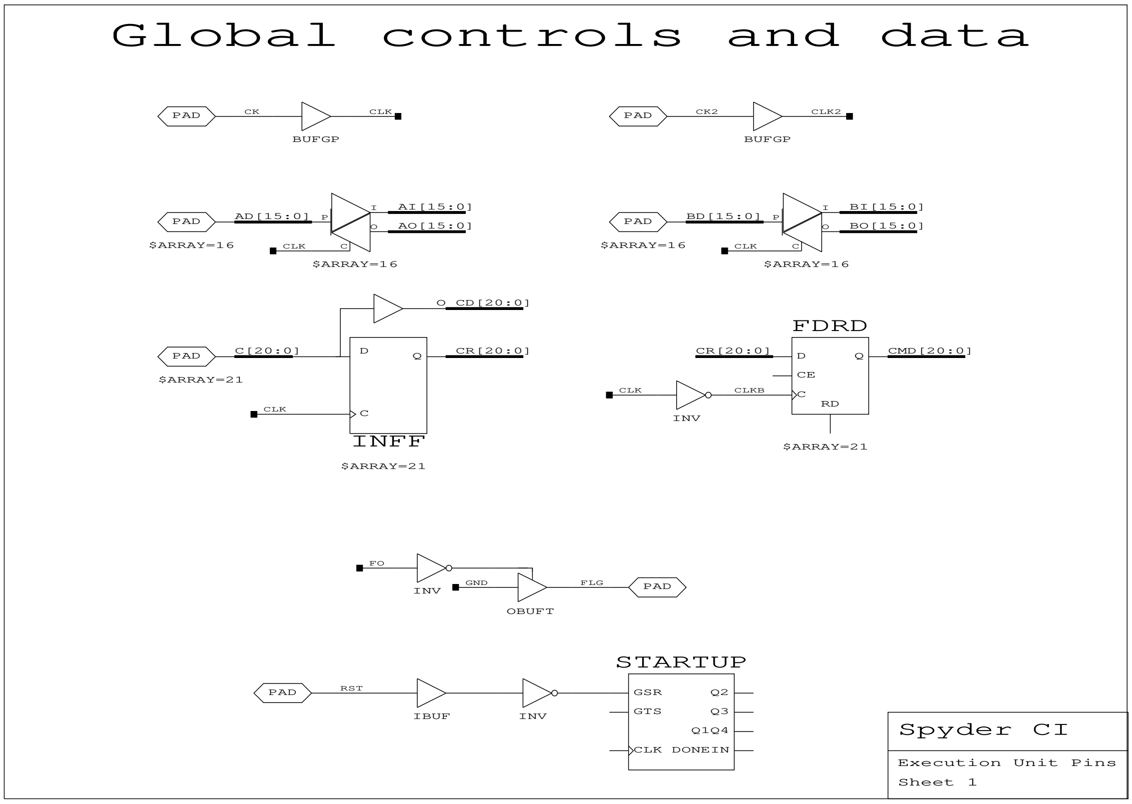

Figure E.2: Global control and data

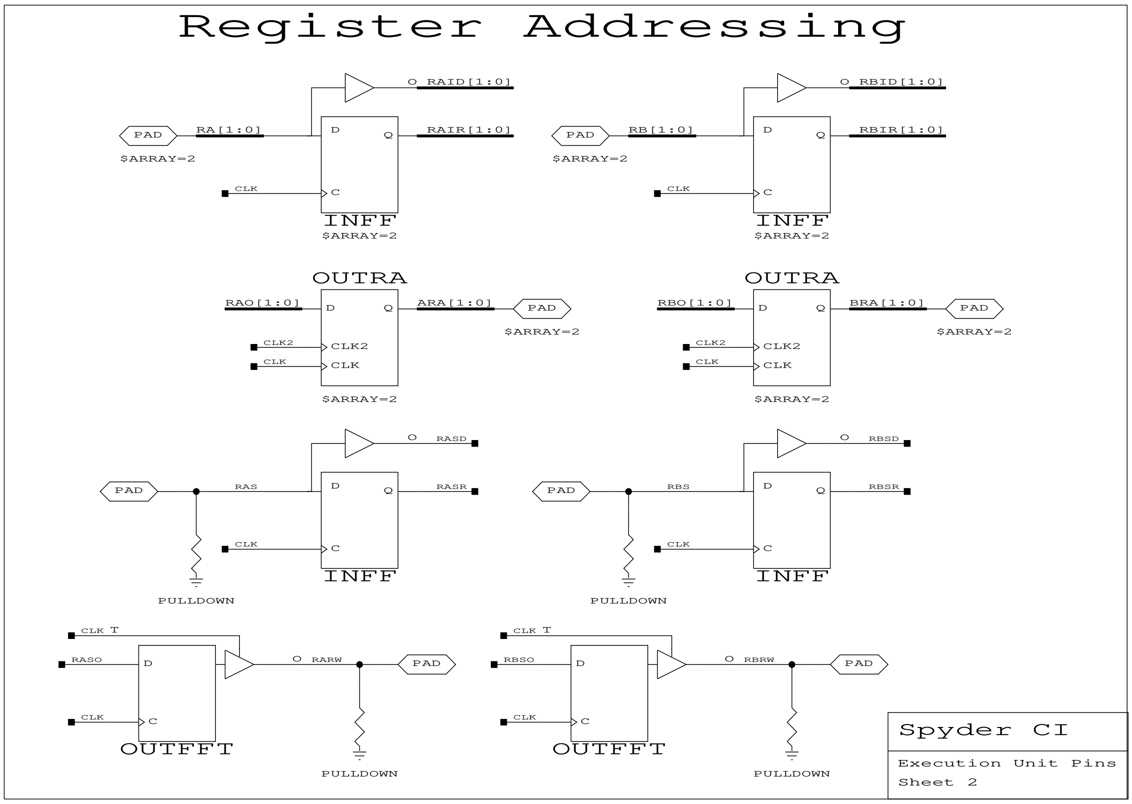

Figure E.3: Register addressing

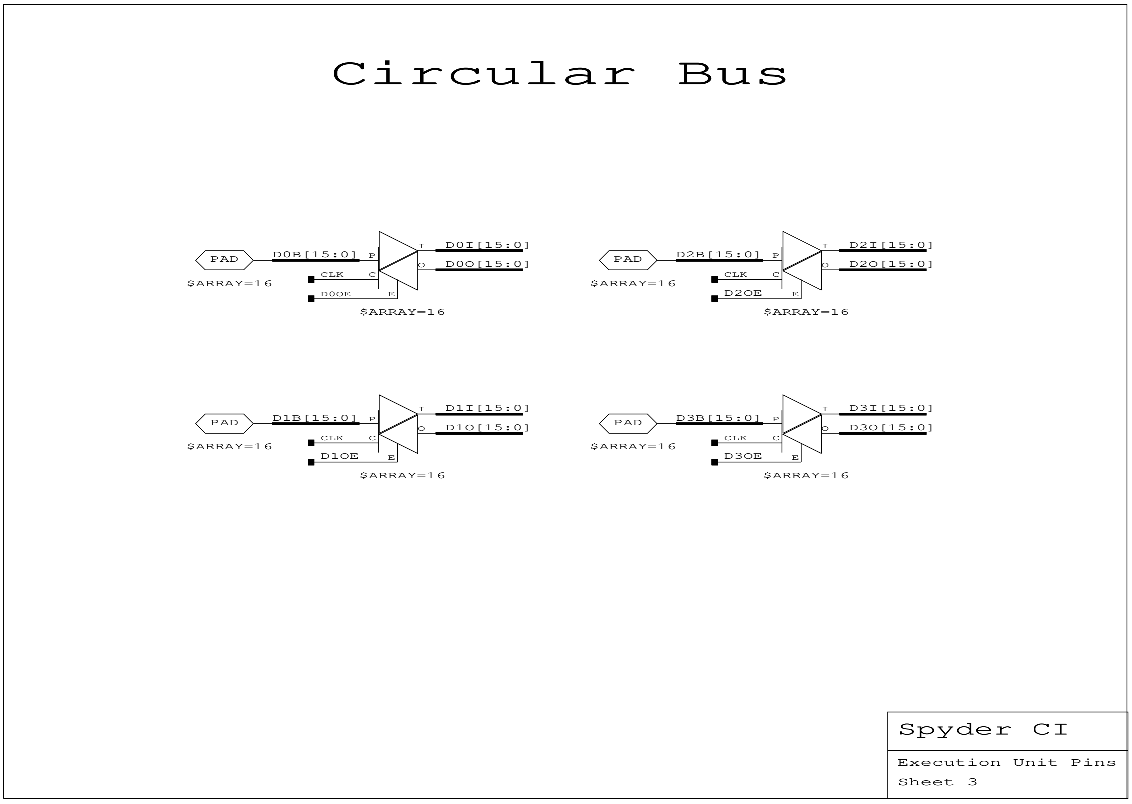

Figure E.4: Circular busses

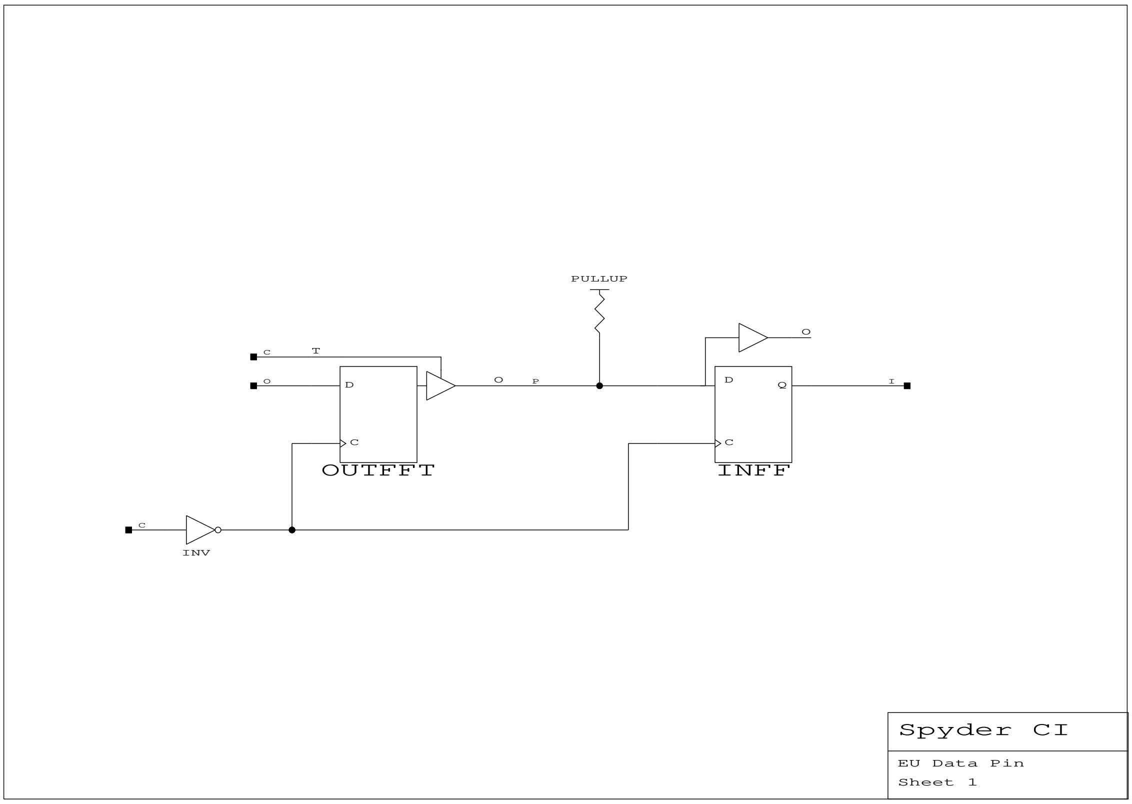

Figure E.5: Execution unit data pin

Bibliography

[ABD92] J. M. Arnold, D. A. Buell, and E. G. Davis. Splash 2. In *Proc. 4^{th} Annual ACM Symp. on Parallel Algorithms and Architectures*, pages 316–322, 1992.

[Act94] Actel. *FPGA Data Book and Design Guide.* Actel Corporation, Sunnyvale, 1994.

[AMD92] AMD. *PAL Devices Data Book.* Advanced Micro Devices, Sunnyvale, 1992.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[BCG82] E. R. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways*, volume 2. Academic Press, London, 1982.

[BFRV92] Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field–Programmable Gate Arrays.* Kluwer Academic Publishers, Boston, 1992.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[BRV89] P. Bertin, D. Roncin, and J. Vuillemin. Introduction to programmable active memories. Technical Report 3, DEC Paris Research Laboratory, June 1989.

[Bur90] Von L. Burton. *The Programmable Logic Device Handbook.* TAB Professional and Reference Books, Blue Ridge Summit, PA, 1990.

[CGO92] Ellie Cutler, Daniel Gilly, and Tim O'Reilly, editors. *The X Window System in a Nutshell.* O'Reilly and Associates, second edition, 1992.

[Cyc94] Joseph M. Cychosz. Efficient binary image thinning using neighborhood maps. In Paul Heckbert, editor, *Graphics Gems IV.* Academic Press, London, 1994.

[DDRS92] Allan Davison, Kieron Drake, William Roberts, and Mel Slater. *Distributed Window Systems: A Practical Guide to X11 and OpenWindows.* Addison-Wesley, 1992.

[DI93] Serge Durand and Christian Iseli. Developing a reconfigurable coprocessor on an SBus board. In *Sun User Group 1993 Proceedings*, pages 5–9, San Jose, California, December 1993.

[Dia96] Domingo Benitez Diaz. Image processing on spyder's VLIW architecture. Technical report, Laboratoire de Systèmes Logiques, EPFL, Lausanne, 1996.

[DS95] Charles Donnelly and Richard Stallman. *Bison*. Free Software Foundation, Cambridge, MA, May 1995.

[Fly95] Michael J. Flynn. *Computer architecture: pipelined and parallel processor design*. Jones and Bartlett, Boston, 1995.

[FOR94a] FORCE Computers Inc. *CPU-3CE/5CE Solaris 2.x VMEbus Driver User's Manual*, 2 edition, October 1994.

[FOR94b] FORCE Computers Inc. *Set of Data Sheets for the SPARC CPU-5CE Technical Reference Manual*, 1 edition, September 1994.

[FOR94c] FORCE Computers Inc. *SPARC CPU-5CE Installation Guide*, 1 edition, December 1994.

[FOR94d] FORCE Computers Inc. *SPARC CPU-5CE Technical Reference Manual*, 1 edition, September 1994.

[Fou93] Open Software Foundation. *OSF/Motif User's Guide*. Prentice Hall, Englewood Cliffs, NJ, 1.2 edition, 1993.

[FR94] J. M. Feldman and C. T. Retter. *Computer Architecture, A designer's text based on a generic RISC*. McGraw-Hill, New York, 1994.

[GH89] Zicheng Guo and Richard W. Hall. Parallel thinning with two-subiteration algorithms. *Communications of the ACM*, 32(3):359–373, March 1989.

[Gwe94] Linley Gwennap. Architects debate VLIW, single-chip MP or something completely different—reprogrammable accelerators. *Microprocessor report*, 8(16):20–21, December 1994.

[Hab88] Stanley Habib, editor. *Microprogramming and firmware engineering methods*. Van Nostrand, New York, 1988.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, San Francisco, second edition, 1996.

[HSJ91] Samuel P. Harbison and Guy L. Steele Jr. *C: a Reference Manual*. Software Series. Prentice Hall, Englewood Cliffs, NJ, third edition, 1991.

[HT94] S. H. Han and R. Trivedi. Primary spacing selection in directionally solidified alloys. *Acta Metallurgica*, 42, 1994.

[IDT92] IDT. *Specialized Memories and Modules.* Integrated Device Technology, Inc, Santa Clara, 1992.

[IS93a] Christian Iseli and Eduardo Sanchez. Beyond Superscalar Using FPGAs. In *IEEE International Conference on Computer Design*, Cambridge Mass., October 1993.

[IS93b] Christian Iseli and Eduardo Sanchez. Spyder: A reconfigurable VLIW processor using FPGAs. In *IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, April 1993.

[IS94a] Christian Iseli and Eduardo Sanchez. Augmentation du parallélisme par la reconfigurabilité. In L. Bougé, M. Cosnard, and P. Fraigniaud, editors, *Actes des* $6^{ème}$ *Rencontres Francophones du Parallélisme, RenPar'6*, pages 3–6, Lyon, June 1994. École normale supérieure.

[IS94b] Christian Iseli and Eduardo Sanchez. A superscalar and reconfigurable processor. In *Field-Programmable Logic Architectures, Synthesis and Applications*, volume 849 of *Lecture Notes in Computer Science*, pages 168–174, Prague, Czech Republic, September 1994. Springer-Verlag.

[IS95a] Christian Iseli and Eduardo Sanchez. A C^{++} compiler for FPGA custom execution units synthesis. In *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa, CA, April 1995.

[IS95b] Christian Iseli and Eduardo Sanchez. Spyder: A SURE (SUperscalar and REconfigurable) processor. *The Journal of Supercomputing*, 9(3):231–252, 1995.

[IS95c] Christian Iseli and Eduardo Sanchez. Synthèse automatique d'unités de traitement parallèles. In G. Libert, J.-L. Dekeyser, and P. Manneback, editors, *Actes des* $7^{ème}$ *Rencontres Francophones du Parallélisme, RenPar'7*, page 243, Mons, Belgique, May 1995. Faculté Polytechnique de Mons.

[Jen94] Jesse H. Jenkins. *Designing with FPGAs and CPLDs.* Prentice Hall, Englewood Cliffs, NJ, 1994.

[Joh91] Mike Johnson. *Superscalar microprocessor design.* Innovative Technology. Prentice Hall, Englewood Cliffs, 1991.

[Lal90] Parag K. Lala. *Digital System Design Using Programmable Logic Devices.* Prentice Hall, Englewood Cliffs, 1990.

[Lea92] Doug Lea. *User's Guide to the GNU* C^{++} *Library.* Free Software Foundation, Cambridge, MA, 2.0 edition, April 1992.

[Man86] Daniel Mange. *Analysis and Synthesis of Logic Systems.* Artech House, Norwood, MA, 1986.

[Man90] Niall Mansfield. *The X Window System: A User's Guide*. Addison-Wesley, 1990.

[Mey94] Thomas Meyer. Réalisation du processeur reconfigurable Spyder sur carte VME. Diploma report, École Polytechnique Fédérale de Lausanne, 1994.

[Mot85] Motorola. *VMEbus Specification Manual, Revision C*. Motorola, February 1985.

[NU95] Stefan Näher and Christian Uhrig. *The LEDA User Manual*. Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany, 3.3 edition, 1995.

[OA74] D. E. Ovsienko and G. A. Alfintsev. Kinetics and shape of crystal growth from the melt of substances with low l/kt values. *Journal of Crystal Growth*, 26, 1974.

[PD84] Kendall Preston Jr. and Michael J. B. Duff. *Modern Cellular Automata*. Plenum Press, New York, 1984.

[Pet94] Chris D. Peterson. *Athena Widget Set – C Language Interface*. X Consortium, 1994.

[PH94] David A. Patterson and John L. Hennessy. *Computer Organization & Design, The Hardware/Software Interface*. Morgan Kaufmann Publishers, San Mateo, 1994.

[RF93] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallelism: History, overview, and perspectives. In B. Ramakrishna Rau and Joseph A. Fisher, editors, *Instruction-Level Parallelism*, pages 9–50. Kluwer Academic Publishers, Boston, 1993.

[RK82] A. Rosenfeld and A. Kak. *Digital Picture Processing*, volume 2. Academic Press, New York, 1982.

[Sed92] Robert Sedgewick. *Algorithms in C++*. Addison-Wesley, 1992.

[Sta92] Richard M. Stallman. *The C Preprocessor*. Free Software Foundation, Boston, MA, 1992.

[Sta95] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Boston, MA, 2.7.1 edition, November 1995.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, second edition, 1991.

[Sun93] SunSoft, Mountain View, CA. *SunOS 5.3 Writing Device Drivers*, 1993.

[Sun94] SunSoft, Mountain View, CA. *Peripherals Administration*, 1994.

[TM87] Tommaso Toffoli and Norman Margolus. *Cellular Automata Machines*. The MIT Press, Cambridge, 1987.

[Tri94] Stephen M. Trimberger, editor. *Field-Programmable Gate Array Technology.* Kluwer Academic Publishers, Boston, 1994.

[Vie95] Viewlogic Systems, Inc., Marlboro, MA. *ViewDoc*, 5.3.1 edition, January 1995.

[Wak90] John F. Wakerly. *Digital Design Principles and Practices.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[WG94] David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual.* Prentice Hall, Englewood Cliffs, NJ, version 9 edition, 1994.

[Xil91] Xilinx. *The XACT4000 Design Implementation Reference guide.* Xilinx, San Jose, 1991.

[Xil94] Xilinx. *The Programmable Logic Data Book.* Xilinx, San Jose, 1994.

Curriculum Vitæ

Christian Iseli was born on September 20, 1963, in Orbe, Switzerland. Upon completion of his studies at the Swiss Federal Institute of Technology in Lausanne (EPFL) and at Carnegie Mellon University in Pittsburgh (CMU), he received the Computer Engineer diploma from the Swiss Federal Institute of Technology in 1988.

After a year as a visiting assistant researcher at the Hitachi Central Research Laboratory in Tokyo, he moved to Lausanne where he started working on his Ph.D. on reconfigurable processors in 1990. He is expected to receive his degree from the Swiss Federal Institute of Technology in Lausanne in early 1996.

Currently, Mr. Iseli holds a position as a research and teaching assistant with the Logic Systems Laboratory at the Swiss Federal Institute of Technology in Lausanne.