

Dynamic Partial Reconfiguration of an FPGA for Computational Hardware Support

Jens Thorvinger
Electroscience
Lund Institute of Technology
Supervisor: Thomas Lenart

June 2004



LUND INSTITUTE OF TECHNOLOGY
Lund University

Abstract

The VLSI development is continuously improving and new ways must be obtained to be able to fully take advantage and utilize the new technology. Reconfigurable hardware might be the next step which will give computer performance a new big leap forward. The idea is to use the, nowadays, high performance FPGA technology to adapt the hardware to the problem. This thesis intends to describe the development of a dynamically reconfigurable system which support multiple threads running concurrently, all with hardware support. A standard Xilinx FPGA is used to test the possibilities of loading partially new hardware configurations while other parts of the FPGA still are active. An example implementation is also realized in order to exemplify the possibilities within the subject.

To my grandfather Karl Berg, for his strong
and supportive interest in me taking my degree

Contents

Contents	7
List of Figures	11
Acknowledgments	13
1 Introduction	15
2 Scope of Thesis	19
2.1 Questions to be answered	19
2.2 Implementation to be realized	20
2.3 Further implementations	20
2.4 Material provided	20
2.4.1 Hardware	20
2.4.2 Software tools	20
2.5 Written report	21
3 Background and Motivation	23
3.1 Reconfigurable computing	23
3.1.1 Computing	23
3.1.2 New ideas can be introduced	26
3.1.3 The super-scalar processor	27
3.1.4 The reconfigurable computing	27
3.1.5 Combining processor and reconfigurable device	28
3.2 FPGA	28
3.2.1 CLB	28
3.2.2 Routing	30
3.2.3 Examples	30
3.3 Earlier FPGA based dynamic systems	30
3.4 The aims of the thesis	31

4	Design Considerations	33
4.1	Restrictions	33
4.2	Bus communication	34
4.3	Bus implementation	35
4.4	Extra area restriction	37
5	Suggested Architecture	39
5.1	Area layout	39
5.1.1	Static module	39
5.1.2	Dynamically reconfigurable modules	40
5.2	Bus macro	40
5.2.1	Architecture	41
5.2.2	Avoidance of simultaneous writing	43
5.2.3	Port interface	44
5.2.4	Bus protocol	45
5.3	External interface	47
6	Developed Methodology	49
6.1	Overview	49
6.2	Introduction	50
6.3	Reconfiguration flow	50
6.3.1	Overview of the flow	51
6.3.2	Project structure	52
6.3.3	Structural rules	52
6.3.4	Bus Macro	54
6.3.5	Local constants	54
6.4	Setup files	56
6.4.1	Bus macro	56
6.4.2	Top VHDL template	60
6.4.3	VHDL modules	62
6.5	Description of the build script	63
6.5.1	Usage	63
6.5.2	How it works	64
6.5.3	Other files	65
6.5.4	Reconfiguration	66
7	Retrospective	67
7.1	Reconfigurable floor plan	67
7.2	Bus macro	69

7.3	Tool support for dynamic reconfiguration	69
7.3.1	Custom made bus macro in FPGA_Editor	69
7.3.2	Another FPGA_Editor bug	70
8	Conclusions	71
9	Future Work	73
9.1	First generation — Software improvements	73
9.1.1	On chip reconfiguration	73
9.1.2	Module error detection	74
9.1.3	Master reconfigurable modules	74
9.2	Second generation — New ASIC	75
9.2.1	Identical reconfigured modules	75
9.2.2	Static module	75
9.2.3	Static bus	75
9.2.4	Optimizing the FPGA structure	75
9.2.5	Module error detection	76
9.3	Third generation — Processor–FPGA integration	76
9.4	Fourth generation	76
	Bibliography	77
A	Definition of Words	79
A.1	Terminology	79
A.2	Abbreviations	79
B	Source Code	81
B.1	The build script source code	81
B.2	The make bus macro source code	87
B.2.1	Shell script	87
B.2.2	Build XDL of bus macro with Perl	88
B.2.3	Synthesize top VHDL config file top.prj	101
B.2.4	Synthesize module VHDL config file module.prj	102
B.2.5	Program FPGA config file bitgen_v2_jtag.ut	103

List of Figures

2.1	The FPGA Virtex-II System Board, used in the thesis.	21
3.1	A data-flow graph for an expression	24
3.2	The functional units connected to the registers in a super-scalar processor.	25
3.3	Application specific pipeline	25
3.4	Basic structure of an FPGA	29
3.5	Configurable Logic Block (CLB) in an FPGA	29
3.6	Configurable routing in an FPGA.	31
4.1	An FPGA design with static and reconfigurable modules.	35
4.2	The implementation of Xilinx bus macro.	36
5.1	Suggested simplified model of a reconfigurable FPGA.	40
5.2	The architecture of the bus macro.	42
5.3	A write cycle diagram.	46
5.4	A read cycle diagram.	47
6.1	The physical implementation of a custom made bus macro	55
6.2	FPGA_Editor screen shot showing the internal bus structure.	58
6.3	FPGA_Editor screen shot showing the internal bus structure borders and module borders.	59
7.1	The first considered Xilinx FPGA structure	68
7.2	The real Xilinx FPGA structure	68
9.1	On chip reconfiguration.	74

Acknowledgments

The thesis you are holding in your hand would not be without several people and these deserve credit. First of all the people in the Digital ASIC corridor in Lunds Institute of Technology. In particular my supervisor Thomas Lenart for giving me encouragement, free hands and support when I needed it. Viktor Öwall — the chief of the corridor deserves credit as well for the approval and help with the thesis.

Thank you also, Henrik, Hugo, Fredrik, Matthias, and Joachim for making sure there is never a boring coffee break. Hugo deserves extra credit for sharing his soap opera life with us, which always reminds us of how trouble free our own lives are.

Outside of the corridor I would like to thank Per Foreby for the great \LaTeX -manual. Without it the thesis would take twice the amount of time to write.

Outside of the digital world I would like to thank my girlfriend Eva for not asking too hard questions about what I am really doing, and also for reminding me of the world outside.

Finally I would like to thank my parents for all their splendid support throughout my life.

Chapter 1

Introduction

Throughout the history of digital electronics the technology has improved exponentially over time. The performance of the devices is roughly doubling every 18 months because transistor size and cost of chips has shrunk in an impressive pace. This gives the microprocessor constructors roughly double the amount of available transistors every two years. Since there is no obvious way to scale the processors to make use of the extra transistors, processor architectures are under constant development and new ideas are rapidly put into practice. An architecture with an outstanding performance a few years ago may not even be sensible to implement in a modern technology and implementations of today were simply too expensive to even be considered back then. To foresee what the future holds simply by observing the current state of the art and extrapolate a likely development and choice of architectures is an almost impossible task. The choice of an architecture is dependent of the technological achievements regarding transistor size, transistor speed, chip size, cost etcetera.

The latest trend for using the ever increasing amount of available transistors are out-of-order super-scalar processors with large on-chip caches. The instruction pipeline is copied and multiple instructions can be processed in parallel. However, there is a limit of how many of the instructions can be processed in parallel due to data dependencies. The control overhead also grows quadratically with the number of pipelines.

Another trend is a larger cache, which is of course better than a small cache, but the overall speedup caused by a cache is not increasing linearly with the cache size. A cache is also occupying a large portion of the chip and the thought of using a part of this area for something improving speed more than the cache is quite appealing.

There are several good and bad ideas of how to use this available area. The

approach suggested in this thesis is based on *Field-Programmable Gate Arrays* — FPGAs. A given problem is most efficiently solved in a custom made ASIC¹ and *not* by using a program running on a general purpose microprocessor. The reason should be obvious: the ASIC can be created and optimized down to the last transistor for the specific problem. A general purpose microprocessor is always designed as a trade off between all possible problems it may encounter and how frequent it is going to encounter them. Some problems require heavy computational power during long times. In these cases it might be profitable to develop a supportive ASIC to free up resources in the microprocessor. A good example is a 3D graphic card in a modern workstation. Practically every problem is better solved with an ASIC but it is definitely not profitable or flexible to develop a new ASIC for every problem. A state-of-the-art ASIC is also becoming more advanced and expensive every day.

A new way of designing a chip is often an off-the-shelf solution, a chip to be configured into practically any digital arbitrary circuit, and thus replacing an ASIC. FPGAs are one kind of configurable chips, who has the ability to be reconfigured an infinite number of times, but the increasing flexibility comes at some costs. An FPGA does not have the speed of an ASIC, neither does it have the low power consumption of an ASIC, nor can it be as highly optimized, and the area cannot be fully utilized. However, the development within the area are taking rapid leaps forward and an FPGA today has a speed up to several 100 MHz, consists of millions of logical units, and can map quite large algorithms as a configuration. As seen before the best, i.e. fastest solution to a problem is an ASIC — custom made for the problem. The cheapest way is to use one or a few flexible chips, such as a general purpose processor. A combination of the abilities in an FPGA — configured specifically for the problem, but still flexible and reconfigurable for the next problem — seems to be a brilliant solution. If a microprocessor was supported with an FPGA, it could use this as a specialized application-specific circuit for each problem — very much like a 3D game today heavily depends on a good graphics card — it would significantly increase its performance. The configuration for the problem is downloaded into the FPGA and when it is finished it can be reconfigured for the next problem in line. An FPGA is configured for the problem by the programmer and therefore it scales much better than the automatic super-scalar technic described later.

There have been several studies within the subject reconfigurable computing the last decade and all kinds of suggested solutions have appeared. Most experiments have been made with commercial standard FPGAs connected to

¹All abbreviations are explained in the Appendix

the processor via a standard external interface such as the PCI bus. This approach is, however, not the way to achieve a higher performance. The idea with a supportive FPGA is that it is working as a unit for heavy calculations, hence needing a fast access to data. The PCI bus is by far too slow for this purpose. The most efficient way is of course an on-chip solution, where the FPGA is located on the same die as the processor itself. Other solutions could be to locate it on the processor–memory bus as a stand-alone-chip, or directly outside, giving it fast access to the memory through the DMA.

There are a number of different ways to implement FPGA support, e.g. an extra reconfigurable pipe in a super-scalar processor, as a small on-chip co-unit which must be stalled, saved, and restored in every context switch, or as a stand-alone unit which can be reserved by a process in the processor and is constantly working until it is flushed. An extended version of the last proposal will be considered in this thesis.

The idea is the use of an external FPGA connected to the memory bus and mapped as a memory area. The FPGA is divided into a few identical large blocks. A software program running on the processor can reserve and use one or more of the blocks in the FPGA depending on the needs, i.e. several applications can have different independent configurations running concurrently at the FPGA. The FPGA is also equipped with an IRQ signal to wake the processor when more data is needed, ready to be written back, or the process is finished.

Chapter 2

Scope of Thesis

The extent of a Master thesis is too limited in time to be enough to develop a functional system described earlier, no matter how tempting the development of such a project would be. A first trembling step is the development of a primitive system with off-the-shelf components. The system is expanded iteratively in order to come as far as possible to the suggested architecture without getting stuck too early, in other words, the problem is too complicated to implement in time. Therefore the task was limited to get a good start of a future development and get a feeling for the tools available and how they can be used within the scope of research.

2.1 Questions to be answered

- Is reconfiguration of the FPGA Virtex-II 1000 from Xilinx, possible to reconfigure partially and dynamically, i.e. while other parts are active?

If it is; there is a foundation for further research in order to use the Virtex-II 1000 as a dynamically reconfigurable element. The thesis is then extended with the following questions.

- How is the Virtex-II 1000 reconfigured dynamically?
- How is the Virtex-II 1000 divided and how can the reconfigurable elements be divided?
- How can independent reconfigurable blocks be developed and how does communication to these work?

2.2 Implementation to be realized

As stated before, an interesting question is *how* the reconfiguration flow works. This shall be answered by a simple example.

- At least two reconfigurable modules shall be implemented.
- When one is running, the other shall concurrently be loaded and started.
- Both shall be reached with commands and data.

2.3 Further implementations

Time is precious and in order to make sure the practical part of the thesis does not swell beyond the scope of time, the following are to be implemented as extra material.

- The FPGA is fully divided into reconfigurable blocks, each one reconfigurable with independent implementations.
- Processor tasks can address configured blocks through the address bus.
- All blocks are running concurrently towards block-local memories.
- IRQ signals can be generated as a wake-up for the processor.
- Blocks not needed are replaced by a DMA process.

2.4 Material provided

The following material was provided by the institution and it was to be used to solve the task.

2.4.1 Hardware

- *Vertex-II System Board*, by Memec Design [8].
- *Parallel Cable IV, Model DLC7*, by Xilinx.

2.4.2 Software tools

- *ModelSim SE PLUS 5.7g*, by Model Technology, <http://www.model.com>.
- *Synplify Pro 7.3.4*, by Synplicity, <http://www.synplicity.com>.
- *Xilinx ISE 6.1* is a set of tools by Xilinx Inc, <http://www.xilinx.com>.

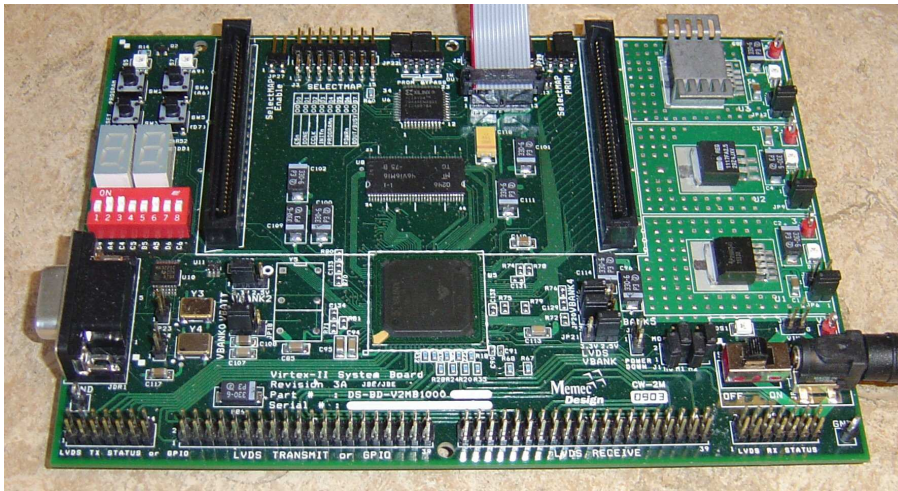


Figure 2.1: The FPGA Virtex-II System Board, by Memec Design, used in the thesis. The FPGA programmer is connected to the top and power form the right. The FPGA itself is the largest chip located below the center of the board.

2.5 Written report

Finally, the thesis ends with a written report covering the following points:

- Introduction to the subject.
- State of the art within this field of research.
- Description of the practical implementation problem.
- Solution to the practical problems.
- A good manual for a methodology solving the practical problem.
- Some reflections and thoughts through out the thesis.
- Suggestions to improve the realized system.

Chapter 3

Background and Motivation

The Background and Motivation chapter will make a short review of the basics of computing. It will also try to explain why reconfigurable computing has advantages over traditional general processor computing. There will also be a brief description of previous related work.

3.1 Reconfigurable computing

In this section the basic functionality of a super-scalar processor¹ is explained and compared with the effectiveness of a reconfigurable design.

3.1.1 Computing

A computation can always be represented as a data-flow graph with the nodes representing simple operations. The purpose of a computational unit is to evaluate such a graph to solve the expression. The graph itself is naturally not evaluated in a real processor, but represented by machine instructions executed one by one. A processor includes a functional unit (ALU) capable of performing the primitives given.

The modern super-scalar processors extracts the parallelism from the machine instructions in order to run instructions without data dependencies in parallel. An example is illustrated in Figure 3.1 where the data dependencies are much more obvious than in code. All the nodes are data dependent of the nodes with vectors pointing directly or indirectly at them. Therefore all the

¹The description is very brief and assumptions are being made that the reader is familiar with the basics of processor architecture. There is a very well written and interesting book on the subject [1].

nodes on a particular level can be concurrently evaluated. A node on a lower level can even be evaluated before all the nodes on a higher level are evaluated, as long as the nodes feeding the lower node with data are evaluated. A super-scalar processor is analyzing the instructions in order to perform as many instructions as possible in parallel. This requires multiple functional units and also gives the possibility to specialize them.

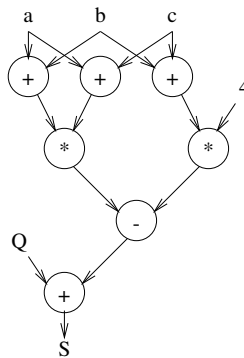


Figure 3.1: A data-flow graph for the expression $S = (a+b)(a+c) - (b+c)*4 + Q$

Having functional units is not enough. These must constantly be supplied with data from memory and other functional units, through the forwarding unit. The amount of space is limited due to the fact that the memory must reside very close to the functional units. This creates the need for a memory hierarchy, with the smallest and fastest memory, called registers, close to the functional units, and slower but larger further away. Moving data is therefore another time consuming task of the processor.

Figure 3.2 shows the concept of multiple functional units in a super-scalar processor. Every clock cycle the extracted data-flow graph is evaluated and the processor tries to use as many functional units as possible. The data is fetched from the registers or the forwarding unit – i.e. the data comes directly from another functional unit. Ultimately every functional unit is used every clock cycle, but unfortunately not realistic due to data dependencies. However, the flexibility of the system makes it capable of executing an arbitrary programs, thus making the processor *programmable*.

On the other end of the scale we have the hardware mapped solution. In this case we hardwire the circuit to our predefined specific problem. An example is the known expression in Figure 3.1 which can be layed out in an application specific pipeline as in figure 3.3. The advantage is a total utilization of the implementation, fully parallelized the data flow graph, less time and die con-

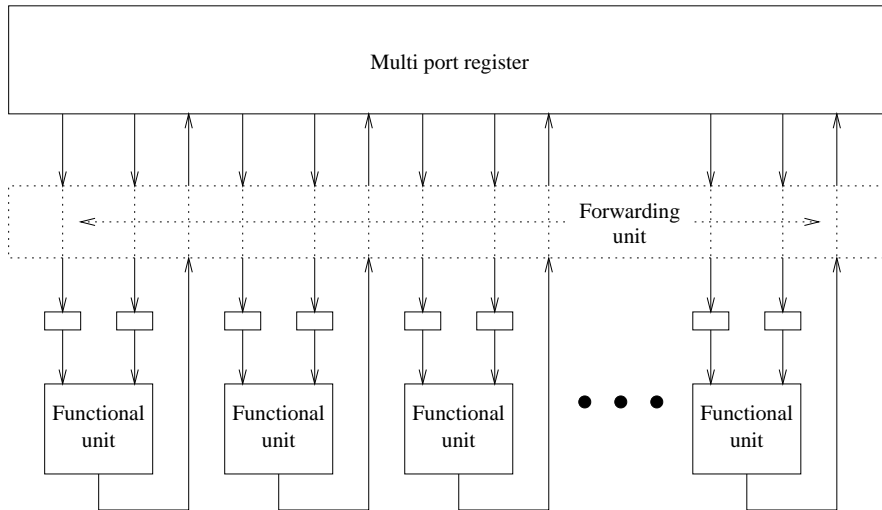


Figure 3.2: The functional units connected to the registers in a super-scalar processor.

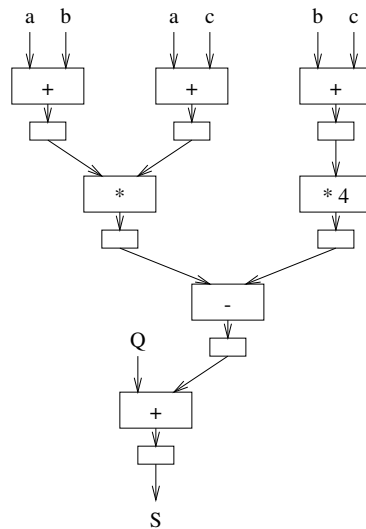


Figure 3.3: A pipeline specifically developed to solve the graph in Figure 3.1

suming overhead, and functional units specialized for its task, i.e. smaller and faster functional units.

A desirable architecture is a combination of the fast and simple ASIC in combination with the flexibility of a processor. An ASIC with the ability to be reprogrammed would do the job, and this is what an FPGA is. The question is; how to integrate it into a general purpose computer to take full advantage of the FPGA.

3.1.2 New ideas can be introduced

Every year the technology of electronic devices is rapidly improving, giving more die area, and more and smaller transistors for the same price. Since more transistors per chip are given every year, new ideas must constantly be introduced in order to utilize them as effectively as possible for each new processor generation.

Most of the real work is done in the functional units and with a larger amount of transistors a few things can be made. Individual functional units can be made faster, e.g. replacing an one bit iterative adder with a larger array adder. In a processor of today all calculations are laid out in parallel and on iterative loops are left to extract. New kinds of functional units can be introduced. Things like floating-point units and MMX are products of such approaches. The drawback is that the more innovative and specialized a functional unit is, the less likely it is to be used. A third approach is to create copies of existing functional units. The overhead created is however growing quadratically with the number of copies. The units can also only be used when parallelism is identified within the machine instructions.

The last alternative with an increasing amount of copies of the functional units is the approach of today, since the other two are developed to their full extent. The challenge is to find the parallelism in the code and take advantage of it. There are three kinds of parallelism that can be considered [2].

- Instruction-level parallelism (ILP) exists within a small group of instructions following each other. Figure 3.1 is an example of such a parallelism. All the primitive operations at one level can be executed concurrently, e.g. the three first additions.
- Inter-iteration parallelism exist when the iterations of a single loop are all independent. The loop can then be unwrapped and executed in parallel. This is also called data or vector parallelism, e.g. useful in matrix multiplication and FFT.

- Thread Parallelism exist between independent threads of execution. Usually the threads have no dependencies between them and can therefor be concurrently executed.

3.1.3 The super-scalar processor

The super-scalar processor is a reality since many years and the development has increased the number of parallel pipelines. The model presented in Figure 3.2 is the basic idea behind it, although the complexity is much larger. The challenge of many ports in the registry combined with a huge testing mechanism for finding data dependencies makes it quite clumsy. Not only the testing mechanism grows quadratically with the number of functional units, the complexity of the forwarding unit and the registry grows as well.

Despite the high complexity and efforts layed down into this technology, the achievements are limited. Far from all functional units can be fully utilized since the code is usually burdened with data dependencies.

The major flaw of the super-scalar technique is that it is only exploiting the first point (ILP) in the list above over different parallelisms. The one giving highest performance if taken advantage of, thread parallelism, is totally abandoned.

3.1.4 The reconfigurable computing

In the pure hardware mapped form of reconfigurable computing there exists no machine instructions and it does not execute code at all. Instead the device is configured with a complete specification of the function at once and set to be running a decent amount of time. Each configuration corresponds an application specific circuit. An example of a configuration was presented in Figure 3.3. This gives the possibility to take full control over the hardware and gives the programmer freedom to fit the hardware to the problem instead of running the software for the problem on a not-so-good processor.

The configuration of the reconfigurable hardware takes everything from a few clock cycles to thousands of cycles, which introduces a new bottleneck. There is a common statement, called the 90–10 rule, which asserts that 90% of the execution time is spent in 10% of the code. Those 10% are generally inner loops. In these cases when a longer execution is mapped down to a smaller code, the reconfigurable devices gets better performance, since the configuration time is small compared with execution time.

3.1.5 Combining processor and reconfigurable device

What about the 90% of the code executing only 10% of the time?

One disadvantage with the reconfigurable devices are an overhead in time due the reconfiguration time. This means that in practice a reconfigurable device is not faster than the processor since it will be slowed down with reconfiguration issues for the parts of the task that is hardly executed anyway. A good compromise would be a combination of both worlds where 90% of the code, which is not so time consuming anyway, is executed on an normal processor and the 10% of the code heavily used is executed in a reconfigurable device.

A sensible way to implement this would be to have the general processor as a master and the reconfigurable device as a sub-element, slave or in another way subordinated the processor. This architecture gives two major advantages. First, the processor takes care of all the control code, special cases code and can control the reconfiguration of the device. Second, the execution is normally executed on the general processor and the reconfigurable hardware is added as support. This approach opens up the possibility of having a fully backwards compatible system.

3.2 FPGA

The FPGA — *Field-Programmable Gate Array* is the reconfigurable device which is the most common today. It has two major fields of use, either as a prototyping platform or as a cheap, small volume alternative to ASICs. Figure 3.4 shows a model of a basic FPGA. The CLBs² – *configurable logic block* – are the functional units and are individually quite small. Between the CLBs there is a reconfigurable network, connecting the different CLBs with one another.

3.2.1 CLB

A simplified form of the CLB in (Xilinx Virtex series) is shown in Figure 3.5, where it is seen that the CLB consists of two four-way LUTs taking four bits inputs and generating one bit output. These can be used in a third three-way LUT with a third signal. The results can either be directly sent out of the CLB or stored in the provided latches. By filling the LUTs, configuring the MUXes and setting all other control bits the behavior of the CLB is defined. Studies throughout the years have proven four inputs per LUT as a good tradeoff

²CLB is not a general term, but a term used by Xilinx.

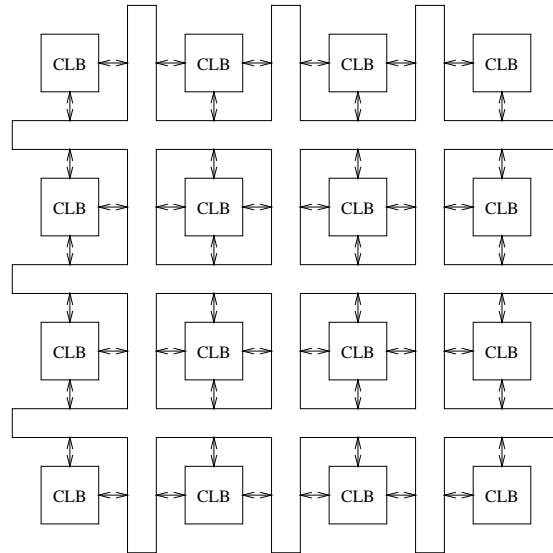


Figure 3.4: Basic structure of an FPGA

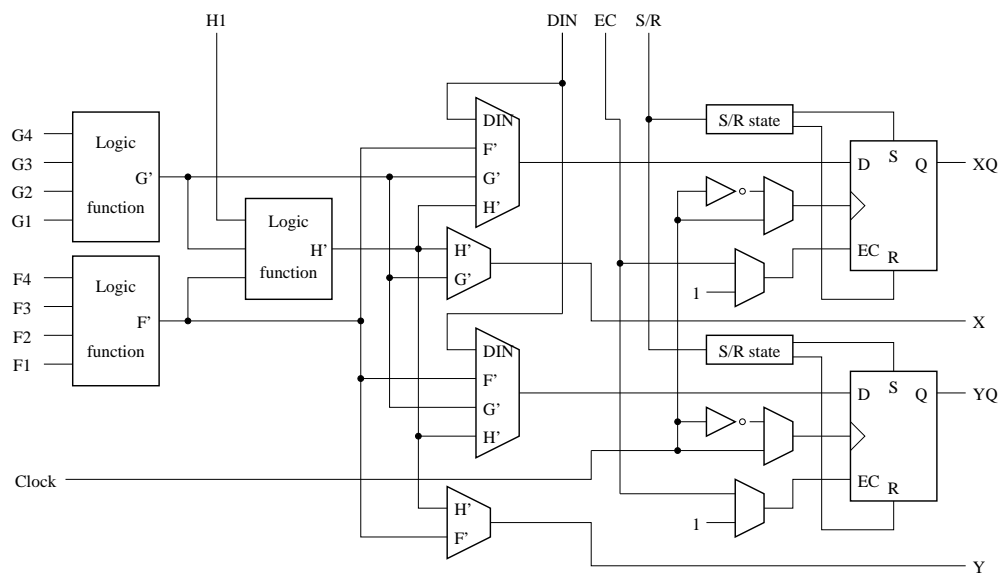


Figure 3.5: Configurable Logic Block (CLB) in an FPGA

between the flexibility (how powerful they are) compared with utilization (how large part can be used for an average problem).

3.2.2 Routing

Within the CLBs lies the intelligence of the FPGA, but without connections between them they are of no good. Layed out through out the structure are the interconnection resources [4]. Long and short wires connected to each other in switch matrixes and to the CLBs through switch nodes, shown in Figure 3.6. These connection nodes are to be reconfigured with the CLBs and are just as complex and memory demanding as the configuration of the CLBs.

3.2.3 Examples

Table 3.1 is an example of sizes of FPGAs from the largest FPGA manufacture, Xilinx Inc.

FPGA	Max System Gates	Logic Cells	Embedded RAM	Max I/Os
XC2V8000	8 M	104 832	3.024 Mbits	1 108
XC2V6000	6 M	76 032	2.592 Mbits	1 104
XC2V4000	4 M	51 840	2.160 Mbits	912
XC2V3000	3 M	32 256	1.728 Mbits	720
XC2V2000	2 M	24 192	1.008 Mbits	624
XC2V1500	1.5 M	17 280	864 kbits	528
XC2V1000	1 M	11 520	720 kbits	432
XC2V500	500 k	6 912	576 kbits	264
XC2V250	250 k	3 456	432 kbits	200
XC2V80	80 k	1 152	144 kbits	120
XC2V40	40 k	596	72 kbits	88

Table 3.1: A comparison between the larger FPGAs from Xilinx Virtex-II series in spring 2004

3.3 Earlier FPGA based dynamic systems

There are numerous examples of FPGA based adaptive systems over the last decade. All of them are more or less experimental or theoretical systems. They

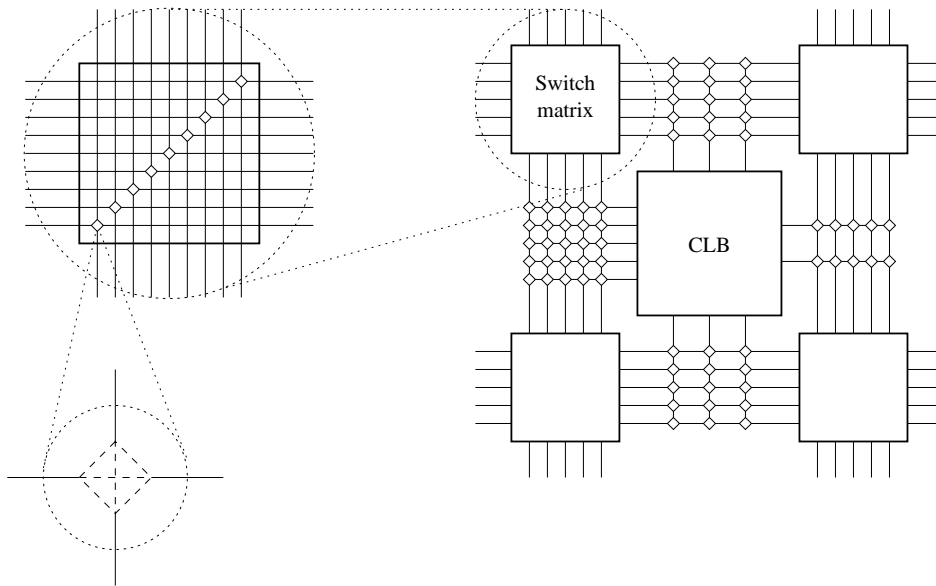


Figure 3.6: Configurable routing in an FPGA. The wires are connected to the CLB through switches. The connecting node in the switch matrix consists of six routing switches.

are however all more or less crippled by different practical problems. Most of the are running with the FPGA on an external PCI card, or they are using FPGAs in such large numbers that a comparison with a standard processor is unfair. A summarization of different systems can be found in [2] and [3].

The conclusions drawn from these previous works is that an approximate speedup of one to five times in the general program, two to twenty times in computational algorithms and up to thousand times faster in algorithms chosen to be much more effective in hardware.

3.4 The aims of the thesis

The dynamic reconfigurable computing is still in its starting phase and has the potential to reach far into higher effectiveness of computing. The aim of this thesis is to highlight one approach of dynamic reconfigurable computing which hopefully will be proven successful. No fully functional system described here will be developed, but the thesis will lay out a foundation for a future one. The reconfigurability of Xilinx FPGAs will be studied and tested. The most important task will be to actually create an example of the usage of reconfigurable

elements in an FPGA and describe the flow to get there.

Chapter 4

Design Considerations

The design developed is first of all restricted by the functionality of the FPGA. Consideration must be taken to the context the FPGA will be used in and how it is connected to the outside world. The internal structure between reconfigurable blocks must also be considered. Everything in this chapter is required according to Xilinx.

A feature in the Xilinx Virtex architecture is the ability to reconfigure a portion of the FPGA while the remainder of the design is in operation. This can not be done arbitrary and restrictions applies.

4.1 Restrictions

The following properties applies to reconfigurable modules and Figure 4.1 illustrates some of the rules below.

- The height of a reconfigurable module is always the full height of the FPGA, see Figure 4.1.
- The width and placement of a reconfigurable module is always multiples of 4 slices¹, e.g. placement $x = 0, 4, 8 \dots$ and width $w = 4, 8, 12 \dots$. However, extra restriction apply, see section 4.4.
- All logic resources occupying the space within the defined width of a reconfigured modules are belonging to this module, i.e. slices, TBUFs, block RAMs, multipliers, IOBs, and *all* routing resources.

¹The term slices will be used throughout the document. A slice is a CLB with an X- and Y-coordinate. Older versions of the Xilinx FPGA uses Rows and columns with R- and C-coordinates.

- Clocking logic is always separated from the reconfigurable module, clocks have separate bit stream frames.
- IOBs within the width (immediately above and below) of a reconfigurable module are belonging to this module.
- If a reconfigurable module occupies either the leftmost or rightmost slices, all IOBs on the specific edge are part of the specific reconfigurable modules resources.
- A reconfigurable module's boundary cannot be changed. The position and region occupied by any single reconfigurable module is always fixed.
- Reconfigurable modules communicate with other modules, both fixed and reconfigurable, by using special hardwired bus macros.
- No parts of the design must rely on any reconfigurable module during the reconfiguration of this module. Handshaking may be required.
- The state of the storage elements inside the reconfigurable module are preserved during the reconfiguration. Since global set/reset (GSR) logic cannot set/reset individual modules, the user must define own module specific set/reset signals if such are needed.

4.2 Bus communication

To conform to the requirements in the partial reconfiguration flow, a bus macro is needed. The communication between a reconfigurable module and another module must go through dedicated wires. In the non partial reconfiguration flow communication wires between the modules are all routed together which means that they can take an arbitrary path. In the partial reconfiguration flow the modules are routed unaware of the other modules. In order to establish a contact between them the two modules must use the *same* physical wires. This is exactly what the bus macro provides. The macro is a hardwired macro and always compiled into exactly the same place which gives the module a communication channel where they can be assured they have contact with each other.

The bus macro uses one TBUF longline per bit in the Virtex-II FPGA which are spanning from the very left to the very right. Every row² in the FPGA has four longlines spanning between the CLBs in the same row. Each row of the

²Slices with the same Y-coordinate.

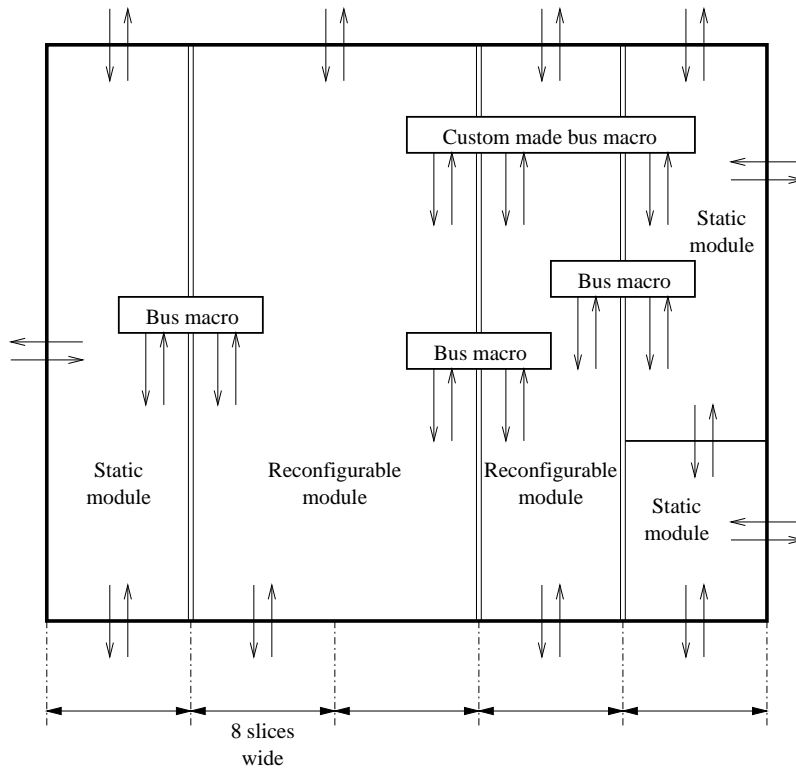


Figure 4.1: An FPGA design with static and reconfigurable modules communicating through bus macro. The short ones are the bus macro provided by Xilinx and the longer one is custom made to connect three modules.

Virtex device can support four bits of a bus macro. The bus macro position exactly straddles the dividing line between design A and B, using four slices of TBUFs on the A side, and four slices of TBUFs on the B side, see Figure 4.2. The longlines are going farther and can therefore continue into the next module C, D, and so on.

4.3 Bus implementation

Xilinx provides these bus macros through their web page³ and has one bus macro for every family of FPGAs. In this example the bus macro for Virtex-II is used. The current implementation uses eight tri-state buffers set up in an arrangement that allows four bits of information to travel either left-to-right or right-to-left, using one TBUF longline per bit, see Figure 4.2.

³<http://www.xilinx.com>

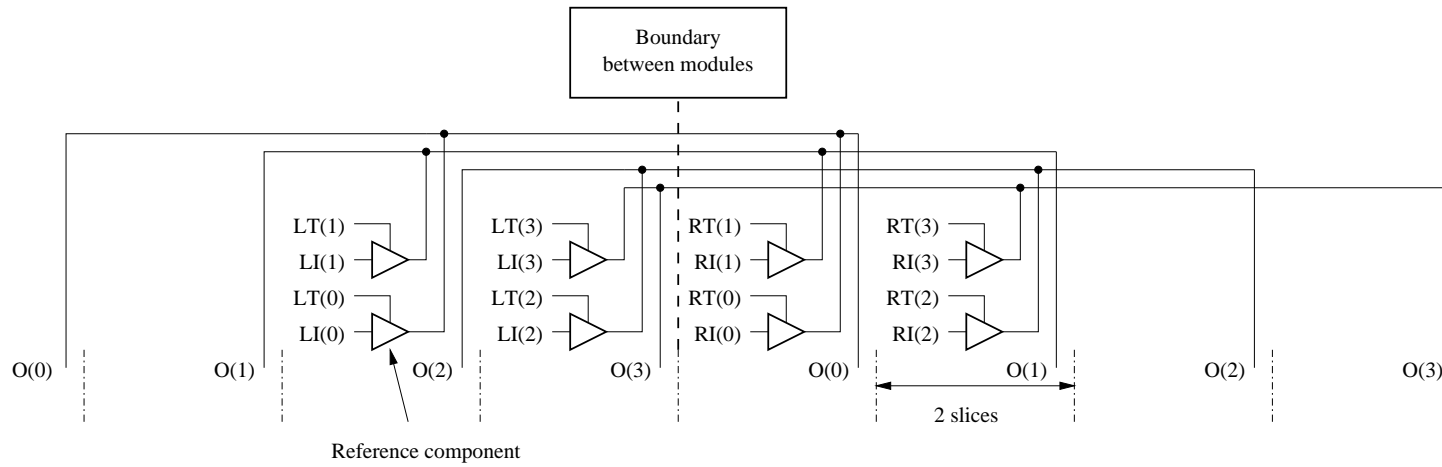


Figure 4.2: The implementation of Xilinx bus macro. The macro consists of four parallel longlines. One buffer at each end is connected to a longline. The $LT()$ and $RT()$ decides which side is the writing side of that longline. E.g. $LT(1)$ if configured as enable, means that $RT(1)$ must be disabled, the $LI(1)$ signal will be written on the bus and can be read at the $O(1)$ outputs.

Some things to think about in the implemented the bus design:

- All modules shall have the same interface. A common address and data bus design would be practical, since all the modules can share the address and data bus.
- The data bus consists of bidirectional lines and therefore multiple modules can simultaneously write on the data bus. This *must* be avoided to ensure no short circus occur. See section 5.2.2.
- Control signals must also be drawn to support the address data transfers.

4.4 Extra area restriction

To write to a four bit bus macro, the TBUFs in four slices in every module must be used. However, as seen in Figure 4.2, only one bit of four can be read every second slice. Hence, for a module to read all four bits on the bus the minimum width of the module is eight slices.

Chapter 5

Suggested Architecture

The design of reconfigurable modules are made by third party developers. In order to make the reconfigurable modules of the third party developers to work simultaneously without problem, a well defined layout of the FPGA is needed. This is divided into two parts. First, the reserved areas, i.e. where the reconfigurable modules are to be placed. Second, the communication between the outside world and the reconfigurable modules. The restrictions in this chapter are not set by Xilinx, but are needed in order to make the suggested architecture of this thesis work.

5.1 Area layout

As much of the area as possible should be used for the reconfigurable modules, since these are doing the desired work, but some other logic, here called static logic, is also required. A drawing of the suggested design can be seen in Figure 5.1.

5.1.1 Static module

Some control logic for the intermodule communication is needed, e.g. an arbitration mechanism to decide who may talk on the bus, an interrupt handler is needed to handle the interrupt requests from the reconfigurable modules, module reset signals, and programming handling. The internal bus protocol might not be compatible with the external interface protocol and therefore a translate layer must be provided.

All this must be implemented and is here implemented in a static module located on the left side of the FPGA. This is to protect the reconfigurable

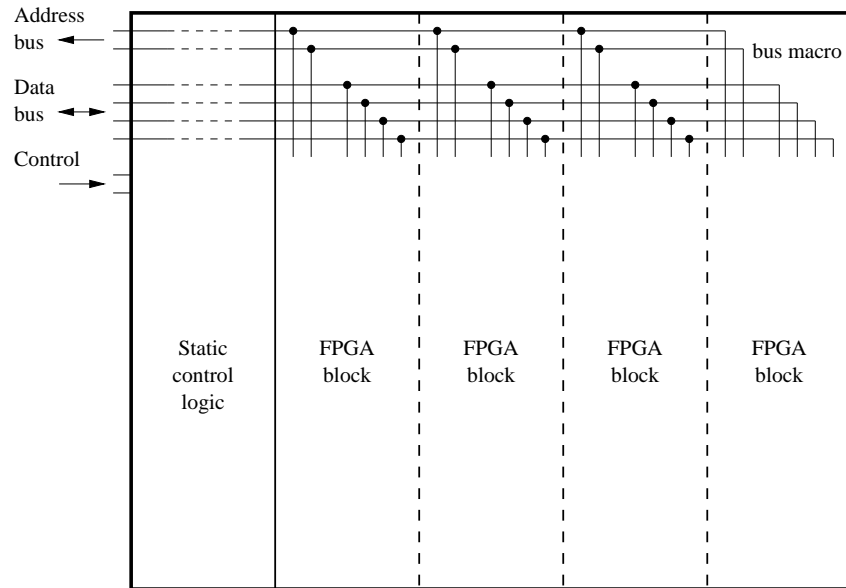


Figure 5.1: Suggested simplified model of a reconfigurable FPGA.

modules from the outside world and from each other and also to protect the FPGA from the reconfigurable modules.

The location of the static modules is chosen with respect to the external interface. The static module communicates with the outside world, hence it needs as many pins as possible. Here the left side was chosen, but the right side works just as well. Using both could also work, but due to simplicity one side is chosen.

5.1.2 Dynamically reconfigurable modules

The reconfigurable modules¹ are developed by third party vendors. They are provided with the information needed, i.e. FPGA to use, top VHDL file, bus macro, location of bus macro, bus protocol, area constraints, and timing constraints. This is enough information to go through a flow of building a reconfigurable block to use in the reconfigurable design.

5.2 Bus macro

The bus macro is a central part of a reconfigurable design. This transports commands and data in and out of the reconfigurable module and it is the only

¹A reconfigurable module always refer to a dynamically reconfigurable module.

way for the reconfigurable module to reach the outside world. *No* other lines must be used than the one provided by the bus macro, according to Xilinx.

The IOBs belonging to the reconfigurable module are also available for communication to the outside world, but in order to keep the suggested architecture as straight forward as possible, all communication is done over the bus.

There are some requirements for the communication, set by this thesis.

- High bandwidth. The idea behind a hardware support solution is to use the high computational speed of an FPGA. This is only possible if data is provided in a high speed as well. One word of data should be able to pass the bus every clock cycle.
- Simplicity. The bus macro must be simple for the ease of use and reduction of complexity.
- Compatibility. By using a well known protocol the use of the bus macro is simplified and the risk of errors are reduced.

5.2.1 Architecture

The suggested architecture is an address bus and bidirectional data bus to all the reconfigurable modules acting as slaves with the leftmost static module as master. This is extended with some control lines. The bus will be built up as a memory bus where data is read and written to the modules. There are perhaps better solutions but in order to keep it simple the memory bus is used. The focus of work is within dynamically partial reconfiguration and not bus protocols.

The reason of this architecture is the simplicity of communication. In the first generation of the design a standard processor is to be used and by mapping the FPGA into the regular address space no extra features of the processor are needed. By using a memory interface the translation in the static module between the internal bus and the external memory bus is also simplified.

Commands to the modules are sent by writing in special predefined memory mapped data areas. These are interpreted by the reconfigurable module as commands.

The bus is also equipped with two extra control bits to each reconfigurable module. One bit to reset the reconfigurable module when needed and one interrupt bit in the other direction to inform the master that data can be read. The entire architecture can be seen in Figure 5.2.

The architecture consists of:

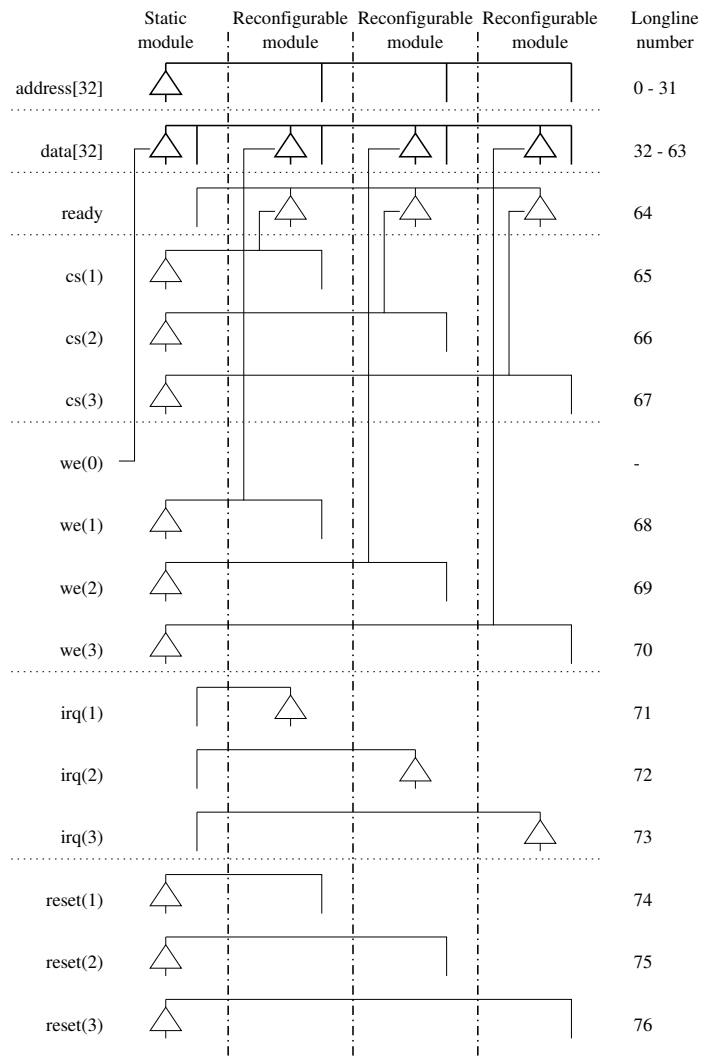


Figure 5.2: The architecture of the bus macro. Here shown with the support for four modules, 32 bits wide address bus, and 32 bits wide data bus. Notice the we(0), this does not exist but since it is not part of a physical longwire but it will be implemented in the top template VHDL file.

- `addr[]`: One address bus driven by the static module and read by the other reconfigurable modules.
- `data[]`: One bidirectional data bus driven by one of the modules at the time and read by the others. A module can only write to the data bus after getting a write enable (`we`) signal from the static module, see section 5.2.2.
- `ready`: One ready signal. This sends back a ready signal from the reconfigurable module currently communicating with the static module. A module can only write to the ready signal after getting a chip select (`cs`) signal from the static module, see section 5.2.2.
- `cs[]`: Chip select signal. There is one wire from the static module to every reconfigurable module which activated the communication for this module.
- `we[]`: Write enable signal. There is one wire from the static module to every reconfigurable module which activates the writing on the data bus for this module.
- `irq[]`: Interrupt signal. There is one wire from every every reconfigurable module to the static module which calls for attention.
- `reset[]`: Reset signal. There is one wire from the static module to every reconfigurable module which resets this reconfigurable module.

5.2.2 Avoidance of simultaneous writing

The reconfigurable modules are, as previously said, developed by third party developers. The third party developer is developing code, compiling the code into a configuration to fit into a predefined place in the FPGA. Many things can go wrong, and the final design cannot be proven stable since all parts are not available, or even developed yet. What can go wrong?

- The developer does not stick to the boundaries of a reconfigurable module place².
- The developer creates internal wires with multiple drivers, causing short circuit, and can result in a destroyed FPGA³.

²This can be tested and taken care of with the reconfiguration control built into the static module. See section 9.1

³A tool compiling HDL code to a configuration will discover this and not accept the code.

- The developer creates a protocol incompatible with the static module and the other reconfigurable modules⁴.
- The developer creates a falsely protocol where the drivers are being activated — data bus and ready signal — simultaneously as other reconfigurable modules are doing the same. This may result in a destroyed FPGA. This cannot be tested by the HDL compilers, since the compilers are unaware of the modules surrounding them.

The last point is the most fatal one and must be solved. The solution is to lift the responsibility for activating the drivers in the reconfigurable module from the module itself and let the static module take care of this. The reconfigurable module will not be able to write to neither the data bus, nor the ready signal. The module will only receive signal telling it when the writing ability has been activated.

The trick is to include the control signals of the tri-state buffers in the top VHDL file, i.e. out of reach of the third party developer. The signals are then controlled from the static module, hence within the safe verified code. It is the responsibility of the static module to make sure the situation of two simultaneous writings never occur.

5.2.3 Port interface

The described bus structure gives a fixed interface to all the reconfigurable modules. There will be two data busses, one `dataI` reading the bus and one `data0` writing to the bus when `we` is enabled. With an address bus width of 32 bits and a data bus width of 32 bits the interface will look like this:

```
addr  : in  std_logic_vector (31 downto 0);
dataI : in  std_logic_vector (31 downto 0);
data0 : out std_logic_vector (31 downto 0);
ready : out std_logic;
cs    : in  std_logic;
we    : in  std_logic;
irq   : out std_logic;
reset : in  std_logic;
gnd   : in  std_logic;
vcc   : in  std_logic
```

This can also be tested and taken care of with a complex testing module built into the static module. See section 9.1

⁴Not much to do but the mistake is easy to discover by the developer.

The static module has a larger interface to the bus macro. This module provides control signals for all the reconfigurable modules, i.e. the static module needs the same amount of control signals for every signal type as there are reconfigurable modules. The exception is the `we` signal since for the sake of consistency this is also provided for the static module itself. The static module controlling seven reconfigurable modules will be:

```
addr  : out std_logic_vector (31 downto 0);
dataI : in  std_logic_vector (31 downto 0);
dataO : out std_logic_vector (31 downto 0);
ready : in  std_logic;
cs     : out std_logic_vector (7  downto 1);
we     : out std_logic_vector (7  downto 0);
irq    : in  std_logic_vector (7  downto 1);
reset  : out std_logic_vector (7  downto 1)
```

5.2.4 Bus protocol

A well defined protocol is needed in order to make sure all units can communicate correctly at the bus. The protocol suggested is a memory bus protocol where the static module is acting as master and the reconfigurable modules are acting as slaves. The master is writing and reading data by addressing the reconfigurable modules with the address bus and `cs` signals. The `we` signals is used to decide whether a read or a write operation is taking place and the clock synchronizes the data transfer.

Write operation

A write operation is here exemplified with the Figure 5.3. All signals are measured at the bus from the masters point of view. Note the skew of the ready signal due to signal delays in the channel.

1. The master, i.e. the static module, starts a transfer at the high transition of the clock, by setting address, data, chip select (`cs`) for the reading module and write enable (`we`) for static module, i.e. itself. The ready signal is drawn low or high as soon as the chip select signal has reached the reconfigurable module and released the ready signal from the high impedance state. If it is drawn high the reconfigurable module *must* read the address and data the next positive flank and the master will give a new address and data. If drawn low the master will keep the address and data the next cycle.

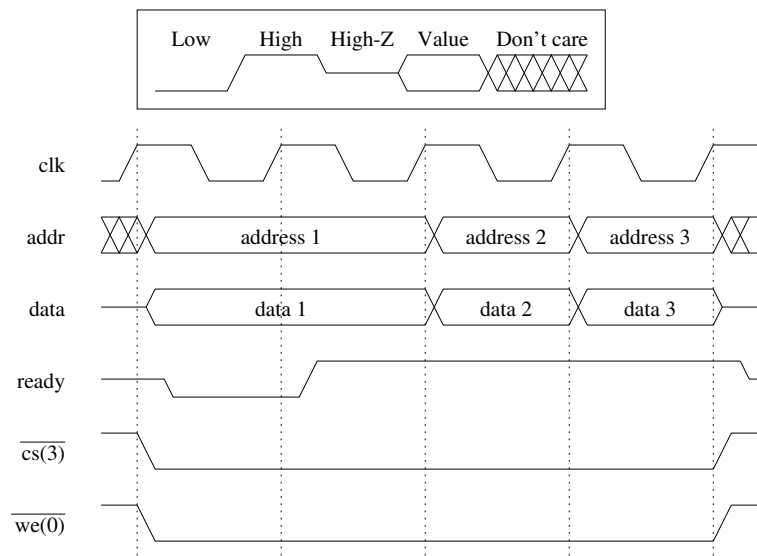


Figure 5.3: A write cycle diagram. The static module addresses the reconfigurable module 3 and writes three bytes. Notice the situation where the third byte is not accepted immediately.

2. When the final byte is transferred the signals chip select (**cs**) and write enable (**we**) are pulled high as soon as possible.

Read operation

A read operation is here exemplified with the Figure 5.4. All signals are here also measured at the bus from the masters point of view. Not the skew of the data bus and the ready signal due to signal delays in the channel.

1. The master, i.e. the static module, starts a transfer at the high transition of the clock, by setting address, chip select (**cs**) for the writing module and write enable (**we**) for writing module. The ready signal is drawn low as soon as the chip select signal has reached the reconfigurable module and released the ready signal from the high impedance state. The data bus is randomly set when released from high impedance by the write enable signal (**we**).
2. The slave *must* read the address on positive flank. If it can set the corresponding data immediately it sets the data on the bus and pulls ready high.
3. If the **ready** signal is high the data is read and a new address is given.

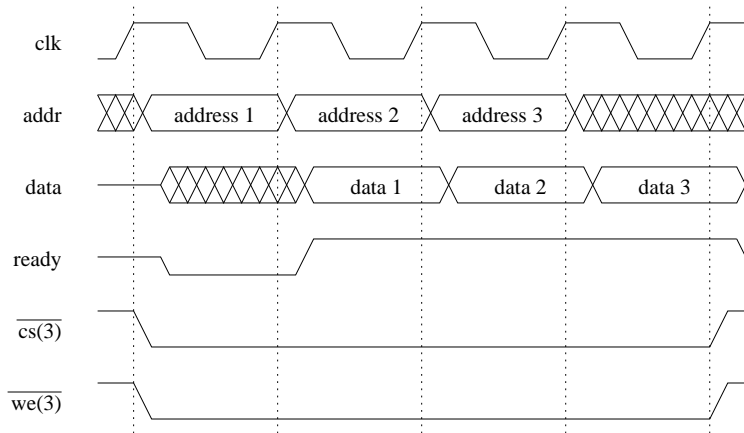


Figure 5.4: A read cycle diagram. The static module addresses the reconfigurable module 3 and reads three bytes.

4. When the final byte is transferred the signals chip select (\overline{cs}) and write enable (\overline{we}) are pulled high as soon as possible.

Note the major difference compared with the write cycle. The address is written in one cycle and the data at the address is given back in the next cycle.

This is a suggested protocol. As long as the static module supports other protocol and the other reconfigurable modules does not become confused, there may exist multiple protocols on the bus. This is however not recommended due to higher complexity.

5.3 External interface

In the normal case the FPGA would be attached somehow to the processor-memory bus. This would make it desirable to develop an interface suiting this bus, but a new motherboard had to be developed to support the FPGA in conjunction with all the regular part of a motherboard. This is a time and money consuming task and not suitable for these initial tests of a reconfigurable system.

In the thesis a Virtex-II System Development Board, by Memec Design, was used. The external interface of the FPGA is hereby quite fixed and the components surrounding the FPGA are those to be used. There exists eight switches whom can be used as inputs and a two figure LED-display to use as output. This limits the static module to be a full interface to the internal bus macro, but externally just communicate with switches and LEDs. In the static

module the testing algorithms has to be created and the result to be displayed at the LEDs⁵.

⁵The LEDs and switches are all connected to the upper row of external pins of the FPGA. Makes it necessary to limit the number of reconfigurable modules in the thesis to three, placed to the very right, since the static module must physically cover all the pins used externally. If the pins used would be placed at the left row of pins only, the maximum number of seven reconfigurable modules could be used.

Chapter 6

Developed Methodology

6.1 Overview

The results of the project is a methodology described in this thesis and implemented in the scripts belonging to this thesis. The idea is to use a standard Xilinx FPGA and create an environment where the development of dynamically reconfigurable modules in the FPGA can be implemented.

With the `make-bus` script it is possible to create a bus macro with the interface of a memory, with an address bus of arbitrary width, data bus of an arbitrary width, and for an arbitrary number of reconfigurable modules. The `make-bus` also creates a top template VHDL file as a top file for the entire FPGA. The user needs to modify the external connections and modify the interface to the static module located to the far left on the FPGA. The port definitions of the created bus macro and the reconfigurable modules are all premade in the top template VHDL file.

The next step is to develop reconfigurable modules. This can be done by the developer of the architecture or by third party developer. With the previously made bus macro and top VHDL file it is now possible to develop independent working reconfigurable modules. The third party developer can then use their modules concurrently, and without synchronization, with other developers modules in the same FPGA.

The creation of the reconfigurable modules are taken care of by the `build` script. The script is divided into three parts.

1. It supports the build of the static modules used in the top design.
2. It supports the design of all the reconfigurable modules in all possible locations in the FPGA. This part is the interesting part for third party

developers.

3. It supports a final assembly phase for all the static and reconfigurable modules for a power up design. The FPGA must be provided with an initial design when started. The developer of the architecture must at least create a dummy reconfigurable module to be placed in all reconfigurable locations before any other reconfigurable modules are dynamically loaded.

A deeper evaluation of how much the suggested architecture can handle and at what speeds has been left for further investigations.

6.2 Introduction

The first brief investigation on the subject came to the conclusion that dynamic partial reconfiguration is indeed supported in the FPGA, Virtex-II 1000 from Xilinx. The FPGA has the ability to be reconfigured in a portion of the FPGA while the reminder is still in operation. The support provided by Xilinx for this ability is to some extent limited. There is however one application note [6] from Xilinx covering the subject.

The thesis uses this application note as a foundation for the developed methodology. The methodology is implemented using the scripts `make-bus` and `build`. All rules and requirements referring to the scripts are therefore specific for this thesis.

6.3 Reconfiguration flow

The dynamic partial reconfiguration can be accomplished in either slave SelectMAP mode or Boundary Scan (JTAG) mode. The method used in this thesis is the Boundary Scan mode, but the SelectMAP mode is likely to work the same way. The kind of dynamic reconfiguration targeted here is the *Multi-Column Partial Reconfiguration, Communication Between Designs* which is the most complicated form of dynamic reconfiguration. This is the case where the reconfigurable blocks are not completely independent and there exist inter-block communication. The consistency of the connections are secured by a *bus macro*, see section 4.2 and 6.4.1. The other forms of dynamic reconfiguration are *Multi-Column Partial Reconfiguration, Independent Designs* and *Small Bit Manipulations*. In the first, no consideration has to be taken to the other modules when one is reconfigured, since there is no communication between the modules. In the later, only a very small number of bits are manipulated in order to change

elementary functionality in functional units. None of these two reconfigurations are considered here.

The partial reconfiguration is based on the Xilinx Modular Design methodology [7]. The part in this document *Modular Design* (Chapter 3) is highly recommended to read before trying to perform a partial reconfiguration.

6.3.1 Overview of the flow

The flow followed to create a reconfigurable modular design goes by these steps¹:

1. Run the `make-bus` script. This generates top template VHDL file and the bus macro needed.
2. Write VHDL code for every module according to the partial reconfiguration guidelines and this thesis.
3. Initial budgeting — Modify the generated top template VHDL file to fit the project, design template floor plan, constrain the logic, and create timing constrains.
4. Run the `build` script created for the thesis for all static modules and all reconfigurable modules in all desired locations.
5. Visually inspect all reconfigurable module designs using `FPGA_Editor` to make sure no connections exist in and out of reconfigurable modules unless routed through bus macros.
6. Run the `build` script in final mode for assembly phase implementation — Minimum is one full design as a initial power-up design. All reconfigurable module locations will be occupied with the reconfigurable module given.
7. Visually inspect the final assembled design using `FPGA_Editor` to make sure no connections exist in and out of reconfigurable modules unless routed through bus macros.
8. Download initial bit stream.
9. Download the reconfigurable bit streams dynamically during execution as desired.

¹All the steps except 5, 7, 8, 9 are developed or modified for this thesis.

6.3.2 Project structure

Xilinx guidelines for dynamic reconfiguration flow [6] uses a quite large and clumsy directory structure. In this thesis, the structure would be *much* larger since every reconfigurable module for every possible location must go through the flow. There must also be an equally large amount of top files (floor plans) created since the tools do not accept a module with the same name in two different locations. Therefore, there must be one top file for each reconfigurable module in each location. This is simplified by the `make-bus` script by generating a top template VHDL file with predefined dummy names for each location (`pos1`, `pos2`, `pos3...`). The `build` script uses `perl` to replace the dummy name for the location to be used with the name of the module to be `build` in the particular location.

The `build` script creates automatically a similar directory structure of the one used in Xilinx flow, but without manual work.

6.3.3 Structural rules

Partial reconfiguration have some requirements and to follow these some general structural rules are needed.

- Overall structure should be a top-level design with each functional module defined as a "black-box" level of hierarchy. Logic at the top level should be limited to I/Os, clocking logic, and the instantiations for the bus macros. There should be no other logic in the top-level design. This thesis requires the reconfigurable modules to be called `pos1`, `pos2`, `pos3...` to confirm to the `build` script. By using the generated top file from `make-bus`, these guidelines are followed.
- The first time the flow of dynamic reconfiguration is tested, it is recommended to use as few reconfigurable modules as possible. Preferably only two modules switching in one location. *Keep everything as simple as possible!*
- All modules, static and reconfigurable, should all be self contained blocks of the hierarchy. All modules must have port definitions of inputs and outputs. In this thesis, these can be copied from the generated top template VHDL file, generated by the `make-bus` script.
- Each reconfigurable module communicating with the surrounding modules (static or reconfigurable) may only do so by first passing through a

bus macro, see section 4.2 and 6.4.1. These macros are hard wired and provided by Xilinx or custom made. In this thesis, the `make-bus` script creates such a custom made bus macro.

- In this thesis, each reconfigurable module must use exactly the same interface in order to be placeable in all locations reserved for reconfigurable modules.
- There are two ways for signals to pass through a reconfigurable module:
 1. Signals passing through a reconfigurable module, connecting the modules on either side of the reconfigured module, must be wired through bus macros. The reconfigurable module must also provide the wires connecting the both sides. The wires passing through the reconfigurable module cannot be used during reconfiguration of the module. This is the method described in Xilinx guidelines for dynamic reconfiguration flow [6].
 2. Signals passing through a reconfigurable module, connecting the modules on either side of the reconfigured module, must be wired through a custom made bus macro. This is a hard wired macro reaching all three (or more) modules at the same time. Since it is hardwired it can be used for communication between the modules on either side of the module between them, even in the case when this module is being reconfigured. This is the method used in this thesis.
- There are possibilities of using multiple clocks [6] (Appendix C). However, for the sake of simplicity, these things should be avoided.
- All clocks must use global logic. The partial reconfiguration flow depends this to be able to keep clocks functional during reconfiguration.
- All signals except clocks must pass through bus macros when entering/leaving a reconfigurable module. This includes signals used in this thesis like reset, constants, enable signals, etcetera. There are possibilities for creating local constants, see section 6.3.5.
- The top-level is synthesized with I/O insertion enabled, producing a top-level net list.
- Each module is synthesized with I/O insertion disabled, producing a module-level net list for each module.

These guidelines are similar to those in [7] *Modular Design Flow*, with the exception of the things stated are belonging to this thesis.

6.3.4 Bus Macro

As described in section 4.2, a bus macro is needed to conform to the requirements in the partial reconfiguration flow. The communication between a reconfigurable module and another module must go through dedicated wires, i.e. in order to establish a contact between them the two modules must use the *same* physical wires. This is what the bus macro provides, see previous Figure 4.2.

The bus macro from Xilinx can however only establish communication between two modules next to one another. This makes it impossible to communicate from one module through a second to the third, without adapting the second module and let it provide wires for this operation. The communication between the first and the third module is broken during reconfiguration of the second one.

To avoid this problem a custom made bus macro can be made, shown in Figure 6.1. The dedicated wires are here extended to reach between three or more modules. The great advantage is that the communication is not broken during reconfiguration since the wires going through the module being reconfigured never change. To clarify; the bus going through the module being reconfigured is also reconfigured with the module, but since it will be placed in exactly the same way before, during, and after the reconfiguration, and since this is done glitch free, the bus will stay intact.

In order to create the bus specified previously a script (`make-bus`) was created. A deeper description of the script is in the next section 6.4.

6.3.5 Local constants

Local constants are needed for setting the direction of bus buffers in the modules. If a buffer is to drive a signal the buffer must be activated with a `gnd` signal, and vice versa. These constant signals must be generated within a module.

The simplest way to generate local constants (PWR and GND) for modules is to create the constants from otherwise unused IOBs and drive them into the modules as needed². Each module needs to have enough unused IOBs for this method to work. The alternative is to create "dummy" LUTs in each module for each constant needed. These both methods are taken directly from the Xilinx manual [6].

For example, to create a local `Logic_0` (GND) signal, instantiate a LUT1 primitive.

```
LUT1 mylut1 (.IO ( ), .0(Dummy_gnd));
```

²This technic is the one used in the thesis.

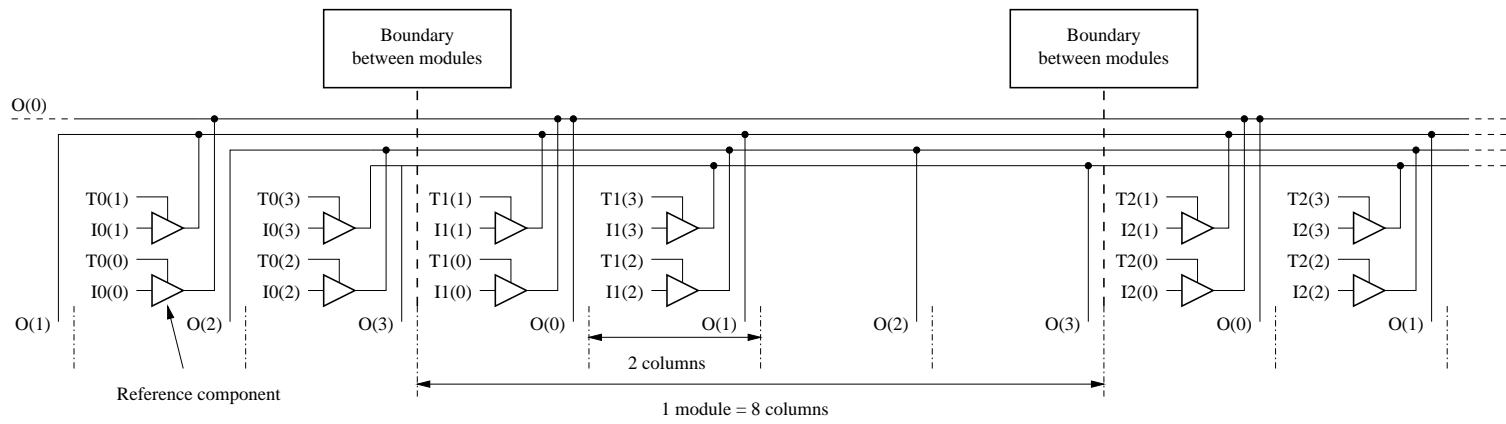


Figure 6.1: The physical implementation of a custom made bus macro. The macro provides communication between modules across multiple module boundaries.

Then, in the `.ucf` file for that module, add the following:

```
inst mylut1 LOCK_PINS; # prevent trimming of the instantiated LUT
inst mylut1 init=0; # initialize to drive a "0"
```

For a local `Logic_1` (VCC):

```
LUT1 mylut1 (.IO ( ), .0(Dummy_vcc));
```

Then, in the `.ucf` file for that module, add the following:

```
inst mylut1 LOCK_PINS; # prevent trimming of the instantiated LUT
```

6.4 Setup files

The described flow above requires a large amount of files. All these must be written according to the rules of the reconfiguration flow and the complexity of the files makes it an almost impossible task for the beginner.

The top file is simply a large amount of connections between the static modules and the bus macro and a large amount of connections between the bus macro and the reconfigurable modules. This can be automatically generated.

The bus macro is a small four bit bus between two modules and the manual creation of a larger bus macro is troublesome and time consuming. This can be automatically generated.

All the automatic generation of files is made by a perl script `mkbusxd1` found in appendix B.2.2. This is started by the script `make-bus` found in appendix B.2.1, both created for this thesis. The user specifies name of the bus, number of modules, address bus width, and data bus width. In the example below the following command was used:

```
make-bus bus 8 32 32
```

6.4.1 Bus macro

The bus macro is a binary macro file (with a `.nmc`-extension) created in `FPGA_Editor`, a graphical CAD-tool where every connection and CLB can be individually modified.

This can be converted to a XDL ASCII-file, describing the same thing, with the command:

```
xd1 -ncd2xd1 bus.nmc
```

By studying the XDL format the reversed process was accomplished. A perl script generates a custom bus in the XDL format and this is converted to a binary macro.

XDL format

The example below is a bus macro using 94 bits spanning over 8 modules and the final XDL-code is using 3924 (!) lines of code.

The format starts with specifying types, names, and the reference component (m0b0):

```
design "__XILINX_NMC_MACRO" x2v1000fg456-4 v2.38 ;
module "bus" "m0b0" , cfg "_SYSTEM_MACRO::FALSE" ;
```

Next the ports used externally access the bus macro are specified. Port name, location, and type (input, output, or tri-state buffer activation (T)). E.g:

```
port "addrI0(0)" "m0b0" "I" ;
port "addrT0(0)" "m0b0" "T" ;
port "addr0(0)" "m0b0" "0" ;
```

Next the location in the FPGA is specified and configuration of the tri-state buffer.

```
inst "m0b0" "TBUF" , placed R40C3 TBUF_X4Y0 ,
  cfg "TINV::T IINV::I _SUPERBEL::TRUE"
;
```

Next the connections between the modules to be connected are specified.

```
net "net0_addr" ,
cfg "
  _NET_PROP::IS_BUS_MACRO:" ,
  outpin "m0b0"          0          ,
  outpin "m7b0"          0          ,
  outpin "m6b0"          0          ,
  outpin "m5b0"          0          ,
  outpin "m4b0"          0          ,
  outpin "m3b0"          0          ,
  outpin "m2b0"          0          ,
  outpin "m1b0"          0          ,
;
```

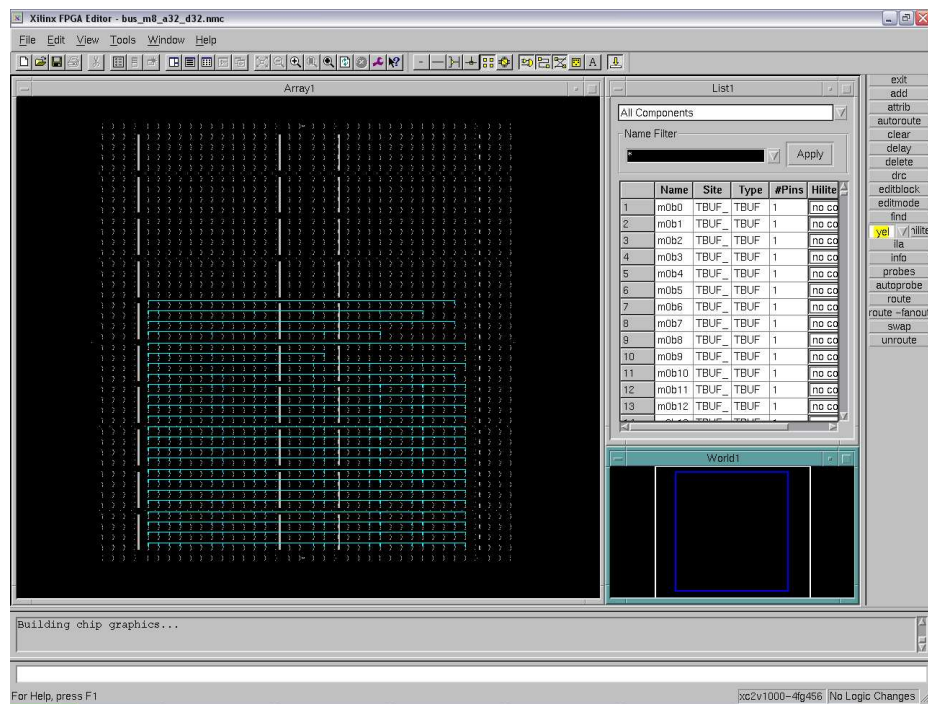


Figure 6.2: FPGA_Editor screen shot showing the internal bus structure. The bus is spanning over eight modules with 32 bit wide address bus, 32 bit wide data bus, 1 ready, 7 cs, 7 we, and 7 reset signals. The signals are grouped in groups of four and are going from bottom to top.

Finally everything ends with:

```
endmodule "bus" ;
```

FPGA_Editor

This is not enough. The XDL file, generated by the `make-bus` script, describes only ports, buffers used, and logical connections between buffers. The missing part is the physical routing between the buffers. This is not following a logical structure, thus cannot be automatically generated. However, the `FPGA_Editor` can do the routing. The routed bus can be seen in Figure 6.2 and with some explanations in Figure 6.3.

The XDL file is first converted from ASCII to binary and renamed to a macro file with the two commands:

```
xdl -xdl2ncd bus.xdl
mv bus.ncd bus.nmc
```

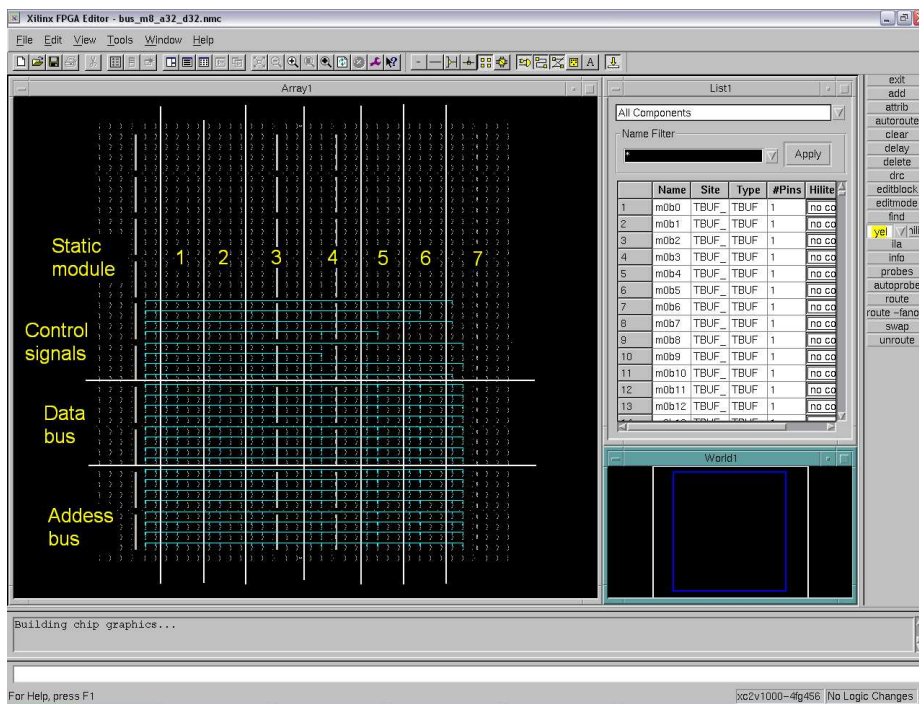


Figure 6.3: FPGA_Editor screen shot showing the internal bus structure borders and module borders. The vertical lines are showing the borders between the modules. The leftmost is the static module and the others are different reconfigurable modules, numbered 1 to 7. The horizontal lines are showing the signals of the bus.

Next the `FPGA_Editor` is started and a every buffer output on the same wire must be selected and a route must be performed. This is simplified by the support of scripting in the `FPGA_Editor`. The script is made in the same way the XDL file is made.

The outputs of the buffers to be connected are selected and the route command routes them. This is performed for every wire to be routed:

```
select pin "TBUF_X8Y0.0"
select pin "TBUF_X16Y0.0"
select pin "TBUF_X24Y0.0"
select pin "TBUF_X32Y0.0"
select pin "TBUF_X40Y0.0"
select pin "TBUF_X48Y0.0"
select pin "TBUF_X56Y0.0"
select pin "TBUF_X4Y0.0"
route
unselect -all
```

The script is finished with the lines:

```
save macro
quit!
end
```

The `FPGA_Editor` is started with the command below, where `bus.scr-script` is generated by the perl `mkbusxd1-script` above.

```
fpga_editor -p bus.scr bus.nmc
```

6.4.2 Top VHDL template

The top VHDL file instantiates all the reconfigurable module places, all the static modules, and the bus macro. The reconfigurable modules are all named by dummy names (`pos1`, `pos2`, `pos3...`) and must not be changed. The interfaces to the reconfigurable modules are also fixed by the definition of the bus. The bus macro itself is fixed and finally the signals connecting the bus macro with the reconfigurable module are fixed. Since all these are fixed and known when creating the bus macro, a top template VHDL file might as well be generated automatically by the perl `mkbusxd1-script` above.

The script generating the top template VHDL file also assumes there exists only one static module and this is fully connected to the bus macro. All the

signals for this is generated but may be changed by the user. Finally the user is responsible for the connections between the external ports and the static module (or modules if the designer decided to have more than one static module).

Top template VHDL file

The generated top template VHDL file is in this example called `comp-bus.vhd`, i.e. `comp-` followed by the busname and ending with `.vhd`. The best description of the top template VHDL file is by generating one and look at it. The file is filled with comments for easier understanding for the user. A good idea is to rename the VHDL file before editing to avoid unwanted overwriting of the file.

The generated top template VHDL file is built up of several parts.

External port definition:

```
entity top is
  port (

    -- Fill in top entity port specification

    ext_gnd      : in  std_logic;
    ext_vcc      : in  std_logic;
    ext_gnd_RM1  : in  std_logic;
    ext_vcc_RM1  : in  std_logic;
    ext_gnd_RM2  : in  std_logic;
    ext_vcc_RM2  : in  std_logic;
    ext_gnd_RM3  : in  std_logic;
    ext_vcc_RM3  : in  std_logic;
    ext_gnd_RM4  : in  std_logic;
    ext_vcc_RM4  : in  std_logic;
    ext_gnd_RM5  : in  std_logic;
    ext_vcc_RM5  : in  std_logic;
    ext_gnd_RM6  : in  std_logic;
    ext_vcc_RM6  : in  std_logic;
    ext_gnd_RM7  : in  std_logic;
    ext_vcc_RM7  : in  std_logic;

  );
end top;
```

The predefined ports are for creating a 1 and 0 for every module (static and reconfigurable). These are externally unconnected and connected internally

either to a pull-up if `vcc` or to a pull-down if `gnd`. They *must* be connected to a pin residing within the area of the module.

Next comes the bus definition. This defines all the ports in the bus macro and must not be changed. The bus macro definition is followed by the reconfigurable module definitions, which also must not be changed.

The last definition is the one of the static module. This is user defined and can be changed and split up into more modules if needed.

This is followed by the signal definitions. Most of them shall not be changed and some can be changed. Signals can also be added as needed. All information about this is included as comment in the VHDL file.

Finally the bus macro, the reconfigurable modules, and the static module are instantiated and connected. Only the static module may be changed here.

6.4.3 VHDL modules

The modules are the reconfigurable modules and the static module(s). The reconfigurable modules are using a fixed interface according to the bus macro. In this case with eight modules, 32 address bits, and 32 data bits the reconfigurable module port definition looks like this:

```
entity dummy is

    port (

        addr  : in  std_logic_vector (31 downto 0);
        dataI : in  std_logic_vector (31 downto 0);
        dataO : out std_logic_vector (31 downto 0);
        ready : out std_logic;
        cs    : in  std_logic;
        we    : in  std_logic;
        irq   : out std_logic;
        reset : in  std_logic;
        gnd   : in  std_logic;
        vcc   : in  std_logic
    );

end dummy;
```

Other than the interface the developer of a reconfigurable module is now free to make the desired implementation.

6.5 Description of the build script

In order to simplify the complicated flow of the *Multi-Column Partial Reconfiguration, Communication Between Designs*, a script was created. The source code of the script `build` is presented in appendix B.1.

6.5.1 Usage

The usage of the `build` script is divided into three parts.

Build static modules

All logic in the FPGA not belonging to a reconfigurable module must be stuffed into modules and may *not* lie in the top file to conform to Xilinx Modular Design methodology [7]. These static modules are to be built as separate modules just as the reconfigurable modules. They have however a fixed place in the floor plan and must be built only for one location.

The static module build mode is activated with the `-s` switch. The VHDL file to be built as the static module is also given as a parameter.

```
build -s <vhdl_module_name>
```

Build reconfigurable modules

This part is similar to the static build mode with one difference, a reconfigurable module must be built for every location where it is placeable, i.e. the script must run as many times as there are locations for the module. The `-p` switch directly followed by a number activates the reconfigurable build mode.

```
build -p <location> <vhdl_module_name>
```

Final assembly

The FPGA needs a configuration to start out with. The static modules must be programmed and even the reconfigurable modules must be setup to enable at least the buses. Therefore a *final assembly* phase is needed. The activation switch is `-f` and the VHDL file specified is a reconfigurable module already prebuilt into all possible locations. The specified reconfigured module is layed out in all reconfigurable locations. Preferable is the reconfigurable module some sort of "dummy" module.

```
build -f <vhdl_module_name>
```

General settings

The command `build -h` brings up a list of all available switches. The `-c` switch cleans up all steps between the VHDL files and the final modules. There are quite a large number of steps which generates a large number of files, and the `-c` switch minimizes this.

All the other switches are just for redefinitions of the default names and paths for all the files needed to run the `build` script.

6.5.2 How it works

This is a brief description for the main functionality. For a deeper understanding of how it works, look at the not-too-complicated script itself, appendix B.1.

Static module

The script copies the top file, describing how the modules are connected to each other and the external pins, to the created top synthesis directory. A project file describing the synthesis is also copied to the directory. The top file is there synthesized.

```
synplify_pro -batch top_${MODULE}$POS.prj
```

Where `$MODULE` is the name of the static module and `$POS` is empty.

The static module file is copied to the created module synthesis directory, and synthesized with the same command syntax as above. Both the synthesized `.edf` files are copied to the created source directory.

The top file is now built from the top source file (`.edf`).

```
ngdbuild -p xc2v1000-fg456-4 -modular initial top_${MODULE}$POS.edf
```

Where `-p xc2v1000-fg456-4` specifies the chip type.

The module file is built from the source file (`.edf`) and with a reference to the newly built top file.

```
ngdbuild -p xc2v1000-fg456-4 -modular module  
-active ${MODULE}$POS $TOPPATH/top_${MODULE}$POS.ngo
```

The module is then mapped with `map` and place-and-routed with `par`. The place-and-route program is the absolutely most time consuming part of the flow. After this the `pimcreate` publishes the routed design to the `pims`-directory. This will be used during final assembly phase late in the flow.

Reconfigurable module

There are three major differences in comparacing with the static flow. First the `$POS` is here set to `_posn` where `n` is the location to be configured.

The second is changes in the VHDL top file — the one describing connections between modules. Several different reconfigurable modules shall be configured for every location in the top file and to avoid the need for one top file for every reconfigurable module in every possible location, a template top file is used instead. All the locations are here called `pos1`, `pos2`, `pos3`... and by using `perl` the name of the right location is replace with the name of the reconfigurable module to be there.

The third is to copy a bit file to the bit-file directory. This is the file to be used when a module shall be dynamically loaded later.

Final assembly

All the routed and ready modules are published in a `pims` directory by the Xilinx tools. This stage requires all the static used modules to be present in the `pims` directory. Also the reconfigurable, usually a "dummy" module built for all locations, must be present here.

The script copies the top file, describing how the modules are connected each other and the external pins, to the created top synthesis directory. The top file is modified by `perl` in such a way that *all* template locations (`pos1`, `pos2`, `pos3`...) are replaced by the initial reconfigurable module for this location. A project file describing the synthesis is also copied to the directory. The top file is synthesized in the directory and the created `.edf` file is copied to the source directory.

```
synplify_pro -batch assembled_${MODULE}.prj
```

Where `$MODULE` is the name of the module.

The top file is now put together from the top source file (`.edf`) and the `pims`.

```
ngdbuild -p xc2v1000-fg456-4 -modular assemble  
-pimpath $PIMSPATH assembled_${MODULE}.edf
```

The module is then mapped with `map` and place-and-routed with `par`. After this the bit file is copied to the bit-file directory. This is the file to be used to initialize the FPGA.

6.5.3 Other files

Other files needed are found in the appendix B.2.3, B.2.4 and B.2.5

6.5.4 Reconfiguration

The final reconfigurable modules are stored as `.bit` files and downloaded into the FPGA through a programmer cable connected to a PC work station. All information about the location of the reconfigurable module is stored in the file and the download is therefore simple and straight forward.

Chapter 7

Retrospective

When an idea is transformed into a real implementation, the result never ends up the way thought of in the beginning.

7.1 Reconfigurable floor plan

The most falsely made assumption was the assumption that there existed some sort of random access to the FPGA, i.e. any configuration bit could be addressed and changed. This led to the first approaches in trying to build a structure looking like Figure 7.1. The idea was to reserve areas in the floor plan for reconfigurable elements. And surround the reconfigurable areas with static logic, supplying an interface to the surrounding world.

However, the Xilinx FPGAs are working in a different way. Parts of the chip *can* be addressed, but only in groups with the minimum area of four adjacent columns. The external pins are also working differently, since the pins located above or below a reconfigurable module are forced to be connected to the reconfigurable module. The result can be seen in Figure 7.2.

This gave a new problem that must be solved. It is now no longer possible to use a fixed static structure to interconnect the modules, since only the leftmost and rightmost reconfigurable modules do have contact with the static logic. The static logic on either side of the FPGA are in turn missing contact with each other. The solution is presented in the next section, 7.2.

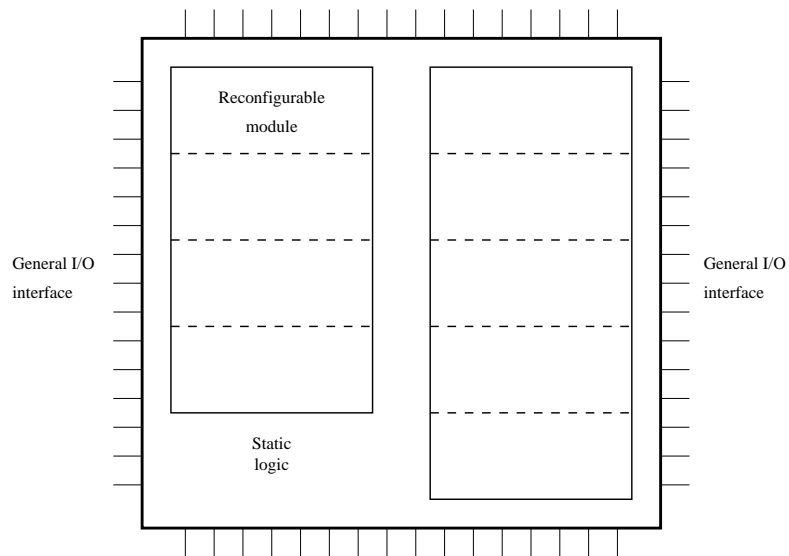


Figure 7.1: The first considered Xilinx FPGA structure. Reservations for reconfigurable elements are made and surrounded by static control logic.

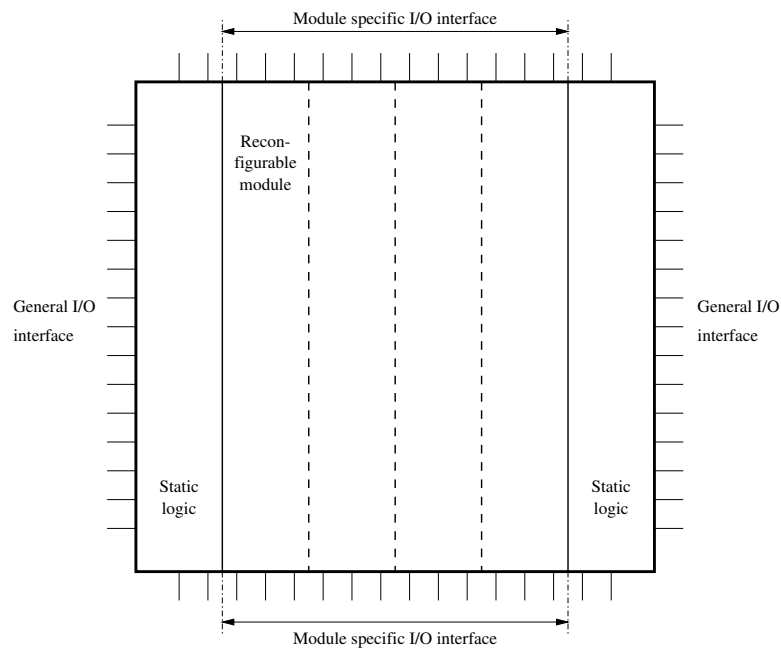


Figure 7.2: The real Xilinx FPGA structure. The reconfigurable elements are spanning from top to bottom with the static logic to the very right and left. External pins above and below a reconfigurable module are forced to be connected to the reconfigurable module.

7.2 Bus macro

To establish communication between reconfigurable blocks, static buses must be layed to be persistent when the surrounding reconfigurable logic is reconfigured. Xilinx provides a bus macro for the purpose of communicating between two adjacent modules, but this is not enough for communication between two modules with a reconfigurable module in between, when the module in between is being reconfigured. A custom made bus macro was therefore necessary.

7.3 Tool support for dynamic reconfiguration

One of the more time consuming parts of a new project is often the struggling battle with the tools. This work is of no exception. Looking back, a brief description of the more annoying problems and the solutions of those could be suitable.

7.3.1 Custom made bus macro in FPGA_Editor

The bus macro must be hardwired and have an exact predefined routing. Since it is not to complicated, the best tool to use is the *FPGA_Editor*. The *FPGA_Editor* shows the entire chip and all routing and configuration of CLBs can be manually made here. Creating a bus here looks easy with the help of the premade bus macro from Xilinx as a model. However, it will not work. The *FPGA_Editor* will forget to set some bus attributes at the bus, making it effectively non functional.

There are two solutions for the problem.

Start the *FPGA_editor* with the Xilinx bus macro loaded. Modify the bus macro and expand it into desired length and function. When the bus macro is saved it will *inherit* the attributes needed from the Xilinx functional bus macro. The problem is that the premade bus macro is made in the center of Xilinx smallest Virtex-II FPGA. Extending it into a large bus may be impossible.

The other solution is to start of with an empty FPGA layout. Create the bus as done before. When the macro is saved, it is saved to a binary `.nmc` file. This can be converted to a XDL-description wit the command:

```
xd1 -ncd2xd1 <bus_name>.nmc
```

Edit the file with a text editor. For each net, there should be a section where it specifies which PIPs to use and such. For each net there should be an attribute like this:

```
cfg "_NET_PROP::IS_BUS_MACRO"
```

FPGA_Editor "forgets" to put this in and must be manually set for the bus macro. The XDL file is converted back to an .ncd file with the command:

```
xdl -xdl2ncd bm_v2p_4b.xdl
```

This creates an .ncd file which must be renamed to an .nmc file to be able to work as a macro.

7.3.2 Another FPGA_Editor bug

After saving a macro in the FPGA_Editor something goes wrong without notice. The macro is saved correctly but *must not* be edited further. Instead exit the FPGA_Editor and reload the saved macro before continuing. If these actions are not taken the macro file will be corrupted the second time it is saved and be unloadable.

Chapter 8

Conclusions

This thesis presents a methodology to develop a dynamically reconfigurable FPGA based system as a support to a general processor for higher computational performance.

The advantage of a processor compared to an ASIC is the flexibility, and vice versa, the advantage of an ASIC compared to an processor is the higher performance. A combination of flexibility and high performance was introduced. 90% of the execution time is spent in 10% of the code, usually algorithms, i.e. high performance is needed. 90% of the code is utilizing 10% of the execution time, usually control code, exception handling, user interactions, i.e. high flexibility is needed.

This thesis proposes a methodology for setting up an environment for dynamically reconfigurable modules in an FPGA. These can be used as hardware algorithms for computational intense tasks. The processor is used for the tasks requiring high flexibility, mentioned before, and for control of the FPGA, i.e. loading and controlling the desired reconfigurable modules.

The focus of the thesis was the methodology setting up an architecture of the FPGA. How to create a bus throughout the FPGA, which will stay preserved during execution. How to divide the area between the reconfigurable modules. How to create an interface to the FPGA and reconfigurable modules within. And how to open up the possibility for third party developers to develop own modules working concurrently with others in the FPGA.

The described methods here are not only theoretical, but has been implemented on an FPGA. Seven reconfigurable modules have been implemented with independent reconfigurations. The blocks can be individually addressed through the address bus and reached with commands and data. All blocks are running concurrently. While blocks are in operation and communication on the

bus, other blocks can be reconfigured dynamically.

Hopefully, the methods described here will lead to a dramatical increase of performance in future computers, cell phones, and PDAs.

Chapter 9

Future Work

No project is ever finished and done. It is just halted on different stages in the development process and realized as products. As a designer it is an obligation to look forward some generations and make sure the design will be able to continue to improve. Some suggestions are given below.

9.1 First generation — Software improvements

This generation will keep all hardware and continue to improve the software part since much more work can be done.

9.1.1 On chip reconfiguration

The chip is reconfigured through an external interface. In this project the design was reconfigured with an external FPGA programmer connected to a workstation - hence slow.

This generation should have an implementation of the reconfiguration protocol in a part of the static module. The external processor would then be able to reconfigure the FPGA simply by writing the configuration into a special memory area handled by the static module. This will in turn reconfigure the reconfigurable module through external pins of the static module connected to the configuration pins of the FPGA itself, see Figure 9.1. The exact reconfiguration times are hard to know without further investigations. The data sheet states a programming speed of maximum 33 Mbytes/s. Since a configuration is around 20000 to 50000 bytes a theoretical reconfiguration would take 0.6 ms to 1.5 ms.

This will also give the static module knowledge of exactly when a module is available for the processor and when it is not due to reconfiguration.

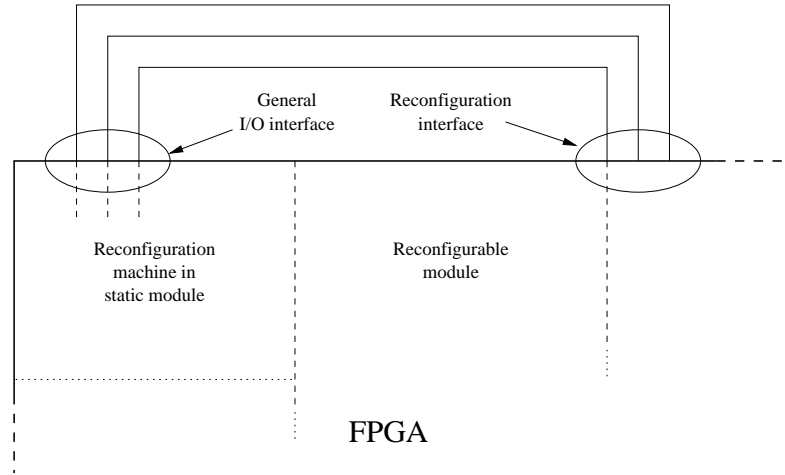


Figure 9.1: On chip reconfiguration where a reconfiguration machine is implemented in the static module. This has physical access to the configuration interface of the FPGA through external wires. Hence, giving the static module the possibility to reconfigure the reconfigurable modules of its own FPGA.

9.1.2 Module error detection

It is a hard task, but a module testing all the reconfigurable modules during download to make sure no lines exists with multiple drivers causing short circuit, would be practical. Due to the complexity of the FPGA this is perhaps better moved to the second generation 9.2, see section 9.2.5.

9.1.3 Master reconfigurable modules

A reconfigurable module uses a primitive memory interface and is acting as a slave. The only way of getting attention is by sending an interrupt and thereby be read or fed by the processor.

A reconfigurable module is usually implementing a computational intense algorithm and thereby requiring a high bandwidth to the memories. An alternative to the architecture of this thesis is to let the reconfigurable modules act as multiple masters act on the bus to read and write data to the memory as needed.

9.2 Second generation — New ASIC

Up till now a standard Virtex FPGA from Xilinx has been used. An FPGA is not optimized for these kind of tasks suggested in the thesis. In order to improve performance a more specific custom made FPGA must be constructed.

9.2.1 Identical reconfigured modules

FPGAs are not homogeneous since a homogeneous FPGA is not a effective design for routing and performance. This creates the need for one implementation of the same algorithm for every place it can be placed in, due to the fact that the places are not identical.

In a custom made FPGA this restriction can be lifted and one reconfigurable configuration can be placed in an arbitrary location. This reduces complexity since every algorithm must be built only once and saves external memory with fewer configurations to store.

9.2.2 Static module

The static module is configured as all other FPGA parts. This module will however never change and much space and speed would be gained by making it really static, e.i. an fixed design on the custom made FPGA chip.

9.2.3 Static bus

The same thought is about the bus spanning over the FPGA. This will also never change and would be better as an ASIC. This approach gives one more advantage. The designer must not more take the bus macro into consideration and make sure it is included into exactly the right place as before. It is now automatically included and impossible to remove.

9.2.4 Optimizing the FPGA structure

A standard FPGA is optimized for a general use and must be able to handle a lot of different situations. This is the reason why it is manipulation bits individually.

In this case we know the normal most used task will be mathematical operations on 32 or 64 bits wide data, since it is working as support to a processor. We can use this knowledge by grouping the bits and route the simultaneously in pairs, quadruples, or more. Studies have shown that optimal grouping is in pairs [2].

The data is also flowing mostly in one direction from the read buffer, through the computational part of the FPGA, and to the write buffers. Data is also floating between CLBs in the same bit row. By optimizing for this the area used for routing can be reduced, thereby giving more area and speed to the cost of some flexibility.

9.2.5 Module error detection

With a more homogeneous FPGA structure and the possibilities to make a fixed structure of the static module the module error tester can now be realized. The module tests the reconfigurable module configuration on the fly during reconfiguration and makes sure no lines exists with multiple drivers causing short circuit. A configuration containing errors is simply refused.

9.3 Third generation — Processor–FPGA integration

This generation will focus in the CPU–FPGA relationship. To get maximum speed and integration between the CPU and its hardware support, in the FPGA, a complete integration is of advantage. The CPU and the FPGA is now produced on the same chip and can share a lot of common things like:

- Common caches for data.
- Common MMU for data.
- More direct command communication between processor and FPGA.

9.4 Fourth generation

World domination...

Bibliography

- [1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, 2nd Edition*, Morgan Kaufmann Publishing Co., Menlo Park, CA. 1996
- [2] John R. Hauser, *Augumenting a Microprocessor with Reconfigurable Hardware*, UofC Berkeley, 2000
- [3] Dr.-Ing. Andreas Koch, *Adaptive Rechensysteme und ihre Entwurfswerkzeuge*, Braunschweig University of Technology
- [4] Dr.-Ing. Thomas Hollstein, *VLSI – Design of Integrated Circuits*, Darmstadt University of Technology, 2002
- [5] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins, *Interconnection Networks Enable Fine-Grain Dynamic Multi-Tasking on FPGAs*, IMEC vzw., Leuven, Belgium
- [6] Davin Lim and Mike Peattie *Two flows for Partial Reconfiguration: Module Based and Small Bit Manipulations*, Xilinx Inc., 2002
- [7] *Development System Reference Guide, ISE 4*, Xilinx Inc., ©2001
- [8] *Virtex-IITM V2MB1000 Development Board User's Guide, v3.0*, Memec Design, 2002
- [9] *MultiPoint Synthesis Using Synlify Pro for Xilinx*, Synplicity Inc., ©2002
- [10] Xilinx Inc., *Constraints Guide ISE 6.1i*

Appendix A

Definition of Words

A.1 Terminology

Dynamic reconfiguration The FPGA or parts of the FPGA are reconfigured in *run time*.

Reconfigurable module A part of an FPGA configuration which can be individually loaded into the FPGA independent of the rest of the FPGA configuration.

Partial reconfiguration Parts/sections of the FPGA are reconfigured.

A.2 Abbreviations

ALU Arithmetic Logic Unit

ASIC Application Specific Integrated Circuit

CLB¹ Configurable Logic Block

CPU Central Processing Unit

DMA Direct Memory Access

FFT Fast Fourier Transform

FPGA Field-Programmable Gate Array

GSR Global Set/Reset

HDL Hardware Description Language

IOB Input Output Block

ISP In System Programming

IRQ Interrupt Request

JTAG Joint Test Action Group

LED Light Emitting Diode

LUT Look-Up Table

MMU Memory Management Unit

MUX Multiplexer

PAR Place and Route

PCI Peripheral Component Interconnect

PIM Physically Implemented Module

PLD Programmable Logic Device

RAM Random Access Memory

TBUF Three-state Buffer

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

VLSI Very Large Scale Integration

XDL Xilinx Design Language

Appendix B

Source Code

B.1 The build script source code

```
#!/bin/sh
#
# Script to create:
# a build of the reconfigurable module s in position n
# a build of the static module s
# a build of final layout with module s in all reconfigurable positions
#

BUSMACRO=bm_3m.nmc
UCF=top.ucf
TOP=top.vhd
JTAG=bitgen_v2_jtag.ut
SYNTOP=top.prj
SYNMOD=module.prj

OPTIONS=""
VHDLFILE=empty
POS=""
MODE=empty
CLEAR=empty

STARTPATH='pwd'

while [ $# -gt 0 ]
do
  case "$1" in
    -h)  OPTIONS="$OPTIONS $1"
        echo ""
        echo "Usage: 'basename $0' -f | -p <position> | -s [-top <file>] [-bm <file>]" 1>&2
        echo "        [-ucf <file>] [-syntop <file>] [-synmod <file>] [-jtag <file>]"
        echo "        [-c] <input-file>"
        echo ""
        echo " -f          Final assembly. All modules needed must be ready."
        echo "          The <input-file> vhdl file will be place at all"
        echo "          reconfigurable positions."
  esac
  shift
done
```

```
echo " -p position    The module is configured as reconfigurable in position."
echo " -s             The module is configured as static."
echo " -top file      Redefine template top file."
echo "                Default: [TOP] in the same directory as input-file."
echo " -bm file       Redefine bus macro file."
echo "                Default: [BUSMACRO]"
echo " -ucf file       Redefine user constraint file."
echo "                Default: [UCF]"
echo " -syntop file     Redefine template synthesis top file."
echo "                Default: [SYNTOPTOP]"
echo " -synmod file     Redefine template synthesis module file."
echo "                Default: [SYNMOD]"
echo " -jtag file        Redefine JTAG file."
echo "                Default: [JTAG]"
echo " -c                Clean up files after compilation."
echo " input-file       The vhdl file to be configured."
echo ""
exit 0
;;
-top) OPTIONS="$OPTIONS $1 $2"
      TOP="$2"
      shift
      ;;
-bm)  OPTIONS="$OPTIONS $1 $2"
      BUSMACRO="$2"
      shift
      ;;
-ucf) OPTIONS="$OPTIONS $1 $2"
      UCF="$2"
      shift
      ;;
-syntop) OPTIONS="$OPTIONS $1 $2"
         SYNTOPTOP="$2"
         shift
         ;;
-synmod) OPTIONS="$OPTIONS $1 $2"
         SYNMOD="$2"
         shift
         ;;
-jtag)  OPTIONS="$OPTIONS $1 $2"
         JTAG="$2"
         shift
         ;;
-f)     if [ $MODE != "empty" ]; then
         echo "'basename $0': Syntax error: -f OR -p OR -s option must be set" 1>&2
         echo "Usage for help: 'basename $0' -h" 1>&2
         exit 1
       fi
       OPTIONS="$OPTIONS $1"
       POS="_pos"
       MODE="FINAL"
       ;;
-p)     if [ $MODE != "empty" ]; then
         echo "'basename $0': Syntax error: -f OR -p OR -s option must be set" 1>&2
         echo "Usage for help: 'basename $0' -h" 1>&2
         exit 1
       fi
```

```

OPTIONS="$OPTIONS $1 $2"
POS="_pos$2"
VHDLPOS="$2"
MODE="RECONF"
shift
;;
-s) if [ $MODE != "empty" ]; then
    echo "'basename $0': Syntax error: -f OR -p OR -s option must be set" 1>&2
    echo "Usage for help: 'basename $0' -h" 1>&2
    exit 1
fi
OPTIONS="$OPTIONS $1"
POS=""
MODE="STATIC"
;;
-c) OPTIONS="$OPTIONS $1"
CLEAR="TRUE"
;;
*) if [ $VHDLFILE != "empty" ]; then
    echo "'basename $0': Syntax error: Only one VHDL file allowed" 1>&2
    echo "Usage for help: 'basename $0' -h" 1>&2
    exit 1
fi
VHDLFILE="$1";;
esac
shift
done

if [ $VHDLFILE = "empty" ]; then
    echo "'basename $0': Syntax error: One VHDL file must be given" 1>&2
    echo "Usage for help: 'basename $0' -h" 1>&2
    exit 1
fi

if [ $MODE = "empty" ]; then
    echo "'basename $0': Syntax error: -f OR -p OR -s option must be set" 1>&2
    echo "Usage for help: 'basename $0' -h" 1>&2
    exit 1
fi

if [ ! -r $VHDLFILE ]; then
    echo "'basename $0': Cannot find or read module file: $VHDLFILE." 1>&2
    exit 1
fi

MODULE="'basename $VHDLFILE .vhd'"

if [ ! -r $TOP ]; then
    echo "'basename $0': Cannot find or read template top file: $TOP." 1>&2
    exit 1
fi

if [ ! -r $BUSMACRO ]; then
    echo "'basename $0': Cannot find or read bus macro file: $BUSMACRO." 1>&2
    exit 1
fi

if [ ! -r $UCF ]; then
    echo "'basename $0': Cannot find or read user constraints file: $UCF." 1>&2
    exit 1

```

APPENDIX B. SOURCE CODE

```
fi
if [ ! -r $SYNTOP ]; then
    echo "basename $0': Cannot find or read template synthesis top file: $SYNTOP." 1>&2
    exit 1
fi
if [ ! -r $SYNMOD ]; then
    echo "basename $0': Cannot find or read template synthesis module file: $SYNMOD." 1>&2
    exit 1
fi
if [ ! -r $JTAG ]; then
    echo "basename $0': Cannot find or read JTAG file: $JTAG." 1>&2
    exit 1
fi

FINALSYNTHPATH="$STARTPATH/synth/assembled_$MODULE"
TOPSYNTHPATH="$STARTPATH/synth/tops/top_$MODULE$POS"
MODSYNTHPATH="$STARTPATH/synth/$MODULE$POS"
SRCPATH="$STARTPATH/src"
FINALPATH="$STARTPATH/assembled_$MODULE"
TOPPATH="$STARTPATH/tops/top_$MODULE$POS"
MODPATH="$STARTPATH/modules/$MODULE$POS"
PIMSPATH="$STARTPATH/pims"
BITPATH="$STARTPATH/bitfiles"

# -----
# Assemble all modules into one bit file
# -----
if [ $MODE = "FINAL" ]; then
    # -----
    # Create custom made top VHDL file
    # -----
    mkdir -p $FINALSYNTHPATH
    cp $STARTPATH/$TOP $FINALSYNTHPATH/assembled_$MODULE.vhd
    perl -i -pe 's/pos1/'$MODULE\_pos1'/g' $FINALSYNTHPATH/assembled_$MODULE.vhd
    perl -i -pe 's/pos2/'$MODULE\_pos2'/g' $FINALSYNTHPATH/assembled_$MODULE.vhd
    perl -i -pe 's/pos3/'$MODULE\_pos3'/g' $FINALSYNTHPATH/assembled_$MODULE.vhd
    perl -i -pe 's/pos4/'$MODULE\_pos4'/g' $FINALSYNTHPATH/assembled_$MODULE.vhd
    perl -i -pe 's/pos5/'$MODULE\_pos5'/g' $FINALSYNTHPATH/assembled_$MODULE.vhd
    perl -i -pe 's/pos6/'$MODULE\_pos6'/g' $FINALSYNTHPATH/assembled_$MODULE.vhd
    perl -i -pe 's/pos7/'$MODULE\_pos7'/g' $FINALSYNTHPATH/assembled_$MODULE.vhd
    perl -i -pe 's/pos8/'$MODULE\_pos8'/g' $FINALSYNTHPATH/assembled_$MODULE.vhd
    perl -i -pe 's/pos9/'$MODULE\_pos9'/g' $FINALSYNTHPATH/assembled_$MODULE.vhd

    # -----
    # Synthesize the assemble top file
    # -----
    cp $SYNTOP $FINALSYNTHPATH/assembled_$MODULE.prj
    cd $FINALSYNTHPATH
    perl -i -pe 's/top_file_name.vhd/'assembled_$MODULE.vhd'/g' assembled_$MODULE.prj
    perl -i -pe 's/top_file_name.edf/'assembled_$MODULE.edf'/g' assembled_$MODULE.prj
    # echo "$STARTPATH/$UCF" | perl -i -pe 's/\\/\\\\/g' > temp.tmp
    # TEMPPATH='cat temp.tmp'
    # rm temp.tmp
    TEMPPATH='echo "$STARTPATH/$UCF" | perl -i -pe 's/\\/\\\\/g' '
    perl -i -pe 's/ucf_file.ucf/'$TEMPPATH'/g' assembled_$MODULE.prj
    synplify_pro -batch assembled_$MODULE.prj
    if [ $? != 0 ]; then exit; fi
```

```

mkdir -p $$SRCPATH
cp rev_1/assembled_$MODULE.edf $$SRCPATH/.
cd $STARTPATH

# -----
# Build the assembled top
# -----
mkdir -p $FINALPATH
cp $UCF $FINALPATH/assembled_$MODULE.ucf
cp $BUSMACRO $FINALPATH/.
cp $$SRCPATH/assembled_$MODULE.edf $FINALPATH/.
cp $JTAG $FINALPATH/.
cd $FINALPATH
ngdbuild -p xc2v1000-fg456-4 -modular assemble -pimpath $PIMSPATH assembled_$MODULE.edf
if [ $? != 0 ]; then exit; fi
map -pr b assembled_$MODULE.ngd -o assembled_$MODULE\_map.ncd assembled_$MODULE.pcf
par -w assembled_$MODULE\_map.ncd assembled_$MODULE.ncd assembled_$MODULE.pcf
bitgen -f $JTAG assembled_$MODULE.ncd
trce assembled_$MODULE.ncd assembled_$MODULE.pcf
mkdir -p $BITPATH
cp assembled_$MODULE.bit $BITPATH/.
cd $STARTPATH

if [ $CLEAR != "empty" ]; then
    cd $FINALSYNTHPATH
    CDIR='pwd'
    cd ..
    rm -rf $CDIR
    CDIR='pwd'
    cd ..
    rmdir $CDIR

    rm -f $$SRCPATH/assembled_$MODULE.edf
    cd $$SRCPATH
    CDIR='pwd'
    cd ..
    rmdir $CDIR
fi

exit 0
fi

# -----
# Create custom made top VHDL file
# -----
mkdir -p $TOPSYNTHPATH
cp $STARTPATH/$TOP $TOPSYNTHPATH/top_$MODULE$POS.vhd
if [ $MODE = "RECONF" ]; then
    perl -i -pe 's/pos'$VHDLPOS'/'$MODULE$POS'/g' $TOPSYNTHPATH/top_$MODULE$POS.vhd
fi

# -----
# Synthesize the top file
# -----
cp $$SYNTH $TOPSYNTHPATH/top_$MODULE$POS.prj
cd $TOPSYNTHPATH
perl -i -pe 's/top_file_name.vhd/'top_$MODULE$POS.vhd'/g' top_$MODULE$POS.prj

```

APPENDIX B. SOURCE CODE

```
perl -i -pe 's/top_file_name.edf/'top_${MODULE$POS}.edf'/g' top_${MODULE$POS}.prj
# echo "$STARTPATH/$UCF" | perl -i -pe 's/\\/\\\\/g' > temp.tmp
# TEMPPATH='cat temp.tmp'
# rm temp.tmp
TEMPPATH='echo "$STARTPATH/$UCF" | perl -i -pe 's/\\/\\\\/g','
perl -i -pe 's/ucf_file.ucf/'$TEMPPATH'/g' top_${MODULE$POS}.prj
synplify_pro -batch top_${MODULE$POS}.prj
if [ $? != 0 ]; then exit; fi
mkdir -p $SRCPATH
cp rev_1/top_${MODULE$POS}.edf $SRCPATH/.
cd $STARTPATH

# -----
# Synthesize the module file
# -----
mkdir -p $MODSYNTHPATH
cp $STARTPATH/$VHDLFILE $MODSYNTHPATH/$MODULE$POS.vhd
cp $SYNMOD $MODSYNTHPATH/$MODULE$POS.prj
cd $MODSYNTHPATH
perl -i -pe 's/module_file_name.vhd/'$MODULE$POS.vhd'/g' $MODULE$POS.prj
perl -i -pe 's/module_file_name.edf/'$MODULE$POS.edf'/g' $MODULE$POS.prj
synplify_pro -batch $MODULE$POS.prj
if [ $? != 0 ]; then exit; fi
mkdir -p $SRCPATH
cp rev_1/$MODULE$POS.edf $SRCPATH/.
cd $STARTPATH

# -----
# Build the top file
# -----
mkdir -p $TOPPATH
cp $UCF $TOPPATH/top_${MODULE$POS}.ucf
cp $BUSMACRO $TOPPATH/.
cp $SRCPATH/top_${MODULE$POS}.edf $TOPPATH/.
cd $TOPPATH
ngdbuild -p xc2v1000-fg456-4 -modular initial top_${MODULE$POS}.edf
if [ $? != 0 ]; then exit; fi
cd $STARTPATH

# -----
# Build the module
# -----
mkdir -p $MODPATH
cp $UCF $MODPATH/top_${MODULE$POS}.ucf
cp $BUSMACRO $MODPATH/.
cp $SRCPATH/$MODULE$POS.edf $MODPATH/.
cp $JTAG $MODPATH/.
cd $MODPATH
ngdbuild -p xc2v1000-fg456-4 -modular module -active $MODULE$POS $TOPPATH/top_${MODULE$POS}.ngo
if [ $? != 0 ]; then exit; fi
map -pr b top_${MODULE$POS}.ngd -o top_${MODULE$POS}_map.ncd top_${MODULE$POS}.pcf
par -w -ol 5 -n 3 -s 3 top_${MODULE$POS}_map.ncd mppr.dir top_${MODULE$POS}.pcf
cp mppr.dir/4_4_3.ncd $MODULE$POS.ncd
bitgen -d -f $JTAG -g ActiveReconfig:yes $MODULE$POS.ncd
trce $MODULE$POS.ncd top_${MODULE$POS}.pcf
mkdir -p $PIMSPATH
pincreate -ncd $MODULE$POS.ncd -ngm top_${MODULE$POS}_map.ngm $PIMSPATH
```

```
if [ $MODE = "RECONF" ]; then
  mkdir -p $BITPATH
  cp $MODULE$POS.bit $BITPATH/.
fi
cd $STARTPATH

if [ $CLEAR != "empty" ]; then
  cd $TOPSYNTHPATH
  CDIR='pwd'
  cd ..
  rm -rf $CDIR
  CDIR='pwd'
  cd ..
  rmdir $CDIR

  cd $MODSYNTHPATH
  CDIR='pwd'
  cd ..
  rm -rf $CDIR
  CDIR='pwd'
  cd ..
  rmdir $CDIR

  rm -f $SRCPATH/top_$MODULE$POS.edf
  rm -f $SRCPATH/$MODULE$POS.edf
  cd $SRCPATH
  CDIR='pwd'
  cd ..
  rmdir $CDIR

  cd $TOPPATH
  CDIR='pwd'
  cd ..
  rm -rf $CDIR
  CDIR='pwd'
  cd ..
  rmdir $CDIR
fi

exit 0
```

B.2 The make bus macro source code

B.2.1 Shell script

```
#!/bin/sh
#

if [ $# = 0 ] || [ $1 = "-h" ]; then
  echo ""
  echo "Usage: 'basename $0' <bus name> <modules> <address> <data>' 1>&2"
  echo ""
  echo " <bus name>      Name of the bus macro to be created."
  echo " <modules>       Number of modules to span over."
  echo " <address>       Width in bits of the internal address bus."
fi
```

```
        echo " <data>          Width in bits of the internal data bus."
        echo ""
        exit 0
    fi

    NAME="$1"
    MODULES="$2"
    ADDR="$3"
    DATA="$4"

    mkbusxdl $NAME $MODULES $ADDR $DATA
    xdl -xdl2ncd $NAME.xdl
    mv $NAME.ncd $NAME.nmc
    rm $NAME*.out
    fpga_editor -p $NAME.scr $NAME.nmc
```

B.2.2 Build XDL of bus macro with Perl

```
#!/usr/bin/perl -w

$name = $ARGV[0];

# signal widths

$modules = $ARGV[1];
$addr = $ARGV[2];
$data = $ARGV[3];
$ready = 1;
$cs = $modules - 1;
$we = $modules - 1;
$irq = $modules - 1;
$reset = $modules - 1;
$status = $ready + $cs + $we + $irq + $reset;
$total = $data + $addr + $status;

# signal offsets

$addroffset = 0;
$dataoffset = $addroffset + $addr;
$readyoffset = $dataoffset + $data;
$csoffset = $readyoffset + $ready;
$weoffset = $csoffset + $cs;
$irqoffset = $weoffset + $we;
$resetoffset = $irqoffset + $irq;

open XDL, "> $name.xdl" or die $!;

# code generation

print XDL 'design "__XILINX_NMC_MACRO" x2v1000fg456-4 v2.38 ;', "\n";
print XDL 'module ', $name, ' "m0b0" , cfg "_SYSTEM_MACRO::FALSE" ;', "\n";
```

```

# create ports

$m = 0;
while ($m < $modules) {
  $w = 0;
  $b = 0;
  while ($b < $addr) {
    print XDL ' port "addrI', $m, '(', $b, ')' " "m', $m, 'b', $w, ' "I" ;', "\n";
    print XDL ' port "addrT', $m, '(', $b, ')' " "m', $m, 'b', $w, ' "T" ;', "\n";
    if ($m == 0) {
      print XDL ' port "addrO(', $b, ')' " "m', $m, 'b', $w, ' "O" ;', "\n";
    }
    $b++;
    $w++;
  }
  $b = 0;
  while ($b < $data) {
    print XDL ' port "dataI', $m, '(', $b, ')' " "m', $m, 'b', $w, ' "I" ;', "\n";
    print XDL ' port "dataT', $m, '(', $b, ')' " "m', $m, 'b', $w, ' "T" ;', "\n";
    if ($m == 0) {
      print XDL ' port "dataO(', $b, ')' " "m', $m, 'b', $w, ' "O" ;', "\n";
    }
    $b++;
    $w++;
  }
  print XDL ' port "readyI', $m, ' " "m', $m, 'b', $w, ' "I" ;', "\n";
  print XDL ' port "readyT', $m, ' " "m', $m, 'b', $w, ' "T" ;', "\n";
  if ($m == 0) {
    print XDL ' port "readyO" "m', $m, 'b', $w, ' "O" ;', "\n";
  }
  $w++;
  $b = 0;
  while ($b < $cs) {
    if ($m == $b + 1) {
      print XDL ' port "csI', $m, ' " "m', $m, 'b', $w, ' "I" ;', "\n";
      print XDL ' port "csT', $m, ' " "m', $m, 'b', $w, ' "T" ;', "\n";
    }
    if ($m == 0) {
      print XDL ' port "csI', $m, '(', $b + 1, ')' " "m', $m, 'b', $w, ' "I" ;', "\n";
      print XDL ' port "csT', $m, '(', $b + 1, ')' " "m', $m, 'b', $w, ' "T" ;', "\n";
      print XDL ' port "csO', $b + 1, ' " "m', $m, 'b', $w, ' "O" ;', "\n";
    }
    $b++;
    $w++;
  }
  $b = 0;
  while ($b < $we) {
    if ($m == $b + 1) {
      print XDL ' port "weI', $m, ' " "m', $m, 'b', $w, ' "I" ;', "\n";
      print XDL ' port "weT', $m, ' " "m', $m, 'b', $w, ' "T" ;', "\n";
    }
    if ($m == 0) {
      print XDL ' port "weI', $m, '(', $b + 1, ')' " "m', $m, 'b', $w, ' "I" ;', "\n";
      print XDL ' port "weT', $m, '(', $b + 1, ')' " "m', $m, 'b', $w, ' "T" ;', "\n";
      print XDL ' port "weO', $b + 1, ' " "m', $m, 'b', $w, ' "O" ;', "\n";
    }
    $b++;
  }
}

```

```
$w++;
}
$b = 0;
while ($b < $irq) {
  if ($m == $b + 1) {
    print XDL ' port "irqI',$m,' "m',$m,'b',$w,' "I" ;', "\n";
    print XDL ' port "irqT',$m,' "m',$m,'b',$w,' "T" ;', "\n";
  }
  if ($m == 0) {
    print XDL ' port "irqI',$m,'($b + 1,')' "m',$m,'b',$w,' "I" ;', "\n";
    print XDL ' port "irqT',$m,'($b + 1,')' "m',$m,'b',$w,' "T" ;', "\n";
    print XDL ' port "irq0($b + 1,')' "m',$m,'b',$w,' "0" ;', "\n";
  }
  $b++;
  $w++;
}
$b = 0;
while ($b < $reset) {
  if ($m == $b + 1) {
    print XDL ' port "resetI',$m,' "m',$m,'b',$w,' "I" ;', "\n";
    print XDL ' port "resetT',$m,' "m',$m,'b',$w,' "T" ;', "\n";
  }
  if ($m == 0) {
    print XDL ' port "resetI',$m,'($b + 1,')' "m',$m,'b',$w,' "I" ;', "\n";
    print XDL ' port "resetT',$m,'($b + 1,')' "m',$m,'b',$w,' "T" ;', "\n";
    print XDL ' port "reset0',$b + 1,')' "m',$m,'b',$w,' "0" ;', "\n";
  }
  $b++;
  $w++;
}
$w++;
}

# place buffers

$m = 0;
while ($m < $modules) {
  $w = 0;
  while ($w < $total) {

    if (($m == 0) ||
        ($w >= $dataoffset && $w < $dataoffset + $data) ||
        ($w >= $addroffset && $w < $addroffset + $addr) ||
        ($w == $readyoffset) ||
        ($w == $csoffset + $m - 1) ||
        ($w == $weoffset + $m - 1) ||
        ($w == $irqoffset + $m - 1) ||
        ($w == $resetoffset + $m - 1)) {
      $r = 40 - int ($w / 4);
      $c = $m * 4 + 1 + int (($w % 4) / 2) + ($m == 0) * 2;
      $x = $m * 8 + 2 * int (($w % 4) / 2) + ($m == 0) * 4;
      $y = 2 * int ($w / 4) + ($w % 2);
      print XDL ' inst "m',$m,'b',$w,' "TBUF" , placed R',$r,'C',$c,' TBUF_X',$x,'Y',$y,' ,', "\n";
      print XDL ' cfg "TINV::T IINV::I _SUPERBEL::TRUE", "\n";
    }
  }
  $m++;
}
```

```

        print XDL '    ;', "\n";
    }
    $w++;
}
$m++;
}

# create connections

my @order = ();

$m = 1;
while ($m < $modules) {
    @order = ($m, @order);
    $m++;
}
@order = (0, @order);

$b = 0;
$w = 0;
while ($b < $addr) {
    print XDL ' net "net', $w, '_addr" ', "\n";
    print XDL ' cfg " ', "\n";
    print XDL ' _NET_PROP::IS_BUS_MACRO:" ', "\n";
    foreach $m (@order) {
        print XDL ' outpin "m', $m, 'b', $w, '" 0          ,', "\n";
    }
    print XDL '    ;', "\n";
    $b++;
    $w++;
}
$b = 0;
while ($b < $data) {
    print XDL ' net "net', $w, '_data" ', "\n";
    print XDL ' cfg " ', "\n";
    print XDL ' _NET_PROP::IS_BUS_MACRO:" ', "\n";
    foreach $m (@order) {
        print XDL ' outpin "m', $m, 'b', $w, '" 0          ,', "\n";
    }
    print XDL '    ;', "\n";
    $b++;
    $w++;
}
$b = 0;
while ($b < $ready) {
    print XDL ' net "net', $w, '_ready" ', "\n";
    print XDL ' cfg " ', "\n";
    print XDL ' _NET_PROP::IS_BUS_MACRO:" ', "\n";
    foreach $m (@order) {
        print XDL ' outpin "m', $m, 'b', $w, '" 0          ,', "\n";
    }
    print XDL '    ;', "\n";
    $b++;
    $w++;
}
$b = 0;

```

APPENDIX B. SOURCE CODE

```
while ($b < $cs) {
    print XDL ' net "net',$w,'_cs" ,'," \n";
    print XDL ' cfg "'," \n";
    print XDL ' _NET_PROP::IS_BUS_MACRO:" ,'," \n";
    foreach $m (@order) {
        if ($m == 0 || $m == $b + 1) {
            print XDL ' outpin "m',$m,'b',$w,'" 0 ,'," \n";
        }
    }
    print XDL ' ;'," \n";
    $b++;
    $w++;
}
$b = 0;
while ($b < $we) {
    print XDL ' net "net',$w,'_we" ,'," \n";
    print XDL ' cfg "'," \n";
    print XDL ' _NET_PROP::IS_BUS_MACRO:" ,'," \n";
    foreach $m (@order) {
        if ($m == 0 || $m == $b + 1) {
            print XDL ' outpin "m',$m,'b',$w,'" 0 ,'," \n";
        }
    }
    print XDL ' ;'," \n";
    $b++;
    $w++;
}
$b = 0;
while ($b < $irq) {
    print XDL ' net "net',$w,'_irq" ,'," \n";
    print XDL ' cfg "'," \n";
    print XDL ' _NET_PROP::IS_BUS_MACRO:" ,'," \n";
    foreach $m (@order) {
        if ($m == 0 || $m == $b + 1) {
            print XDL ' outpin "m',$m,'b',$w,'" 0 ,'," \n";
        }
    }
    print XDL ' ;'," \n";
    $b++;
    $w++;
}
$b = 0;
while ($b < $reset) {
    print XDL ' net "net',$w,'_reset" ,'," \n";
    print XDL ' cfg "'," \n";
    print XDL ' _NET_PROP::IS_BUS_MACRO:" ,'," \n";
    foreach $m (@order) {
        if ($m == 0 || $m == $b + 1) {
            print XDL ' outpin "m',$m,'b',$w,'" 0 ,'," \n";
        }
    }
    print XDL ' ;'," \n";
    $b++;
    $w++;
}

print XDL 'endmodule "',$name,'" ;'," \n";
```

```

# Create script to fpga_editor for routing

open SCR, "> $name.scr" or die $!;

@order = ();
$m = 1;
while ($m < $modules) {
    @order = (@order, $m);
    $m++;
}
@order = (@order, 0);

$w = 0;
$b = 0;
while ($b < $addr) {
    foreach $m (@order) {
        $x = $m * 8 + 2 * int (($w % 4) / 2) + ($m == 0) * 4;
        $y = 2 * int ($w / 4) + ($w % 2);
        print SCR 'select pin "TBUF_X', $x, 'Y', $y, '.0"', "\n";
    }
    print SCR 'route', "\n";
    print SCR 'unselect -all', "\n";
    $b++;
    $w++;
}
$b = 0;
while ($b < $data) {
    foreach $m (@order) {
        $x = $m * 8 + 2 * int (($w % 4) / 2) + ($m == 0) * 4;
        $y = 2 * int ($w / 4) + ($w % 2);
        print SCR 'select pin "TBUF_X', $x, 'Y', $y, '.0"', "\n";
    }
    print SCR 'route', "\n";
    print SCR 'unselect -all', "\n";
    $b++;
    $w++;
}
$b = 0;
while ($b < $ready) {
    foreach $m (@order) {
        $x = $m * 8 + 2 * int (($w % 4) / 2) + ($m == 0) * 4;
        $y = 2 * int ($w / 4) + ($w % 2);
        print SCR 'select pin "TBUF_X', $x, 'Y', $y, '.0"', "\n";
    }
    print SCR 'route', "\n";
    print SCR 'unselect -all', "\n";
    $b++;
    $w++;
}
$b = 0;
while ($b < $cs) {
    foreach $m (@order) {
        if ($m == 0 || $m == $b + 1) {
            $x = $m * 8 + 2 * int (($w % 4) / 2) + ($m == 0) * 4;

```

```
        $y = 2 * int ($w / 4) + ($w % 2);
        print SCR 'select pin "TBUF_X', $x, 'Y', $y, '.0"', "\n";
    }
}
print SCR 'route', "\n";
print SCR 'unselect -all', "\n";
$b++;
$w++;
}
$b = 0;
while ($b < $we) {
    foreach $m (@order) {
        if ($m == 0 || $m == $b + 1) {
            $x = $m * 8 + 2 * int (($w % 4) / 2) + ($m == 0) * 4;
            $y = 2 * int ($w / 4) + ($w % 2);
            print SCR 'select pin "TBUF_X', $x, 'Y', $y, '.0"', "\n";
        }
    }
    print SCR 'route', "\n";
    print SCR 'unselect -all', "\n";
    $b++;
    $w++;
}
$b = 0;
while ($b < $irq) {
    foreach $m (@order) {
        if ($m == 0 || $m == $b + 1) {
            $x = $m * 8 + 2 * int (($w % 4) / 2) + ($m == 0) * 4;
            $y = 2 * int ($w / 4) + ($w % 2);
            print SCR 'select pin "TBUF_X', $x, 'Y', $y, '.0"', "\n";
        }
    }
    print SCR 'route', "\n";
    print SCR 'unselect -all', "\n";
    $b++;
    $w++;
}
$b = 0;
while ($b < $reset) {
    foreach $m (@order) {
        if ($m == 0 || $m == $b + 1) {
            $x = $m * 8 + 2 * int (($w % 4) / 2) + ($m == 0) * 4;
            $y = 2 * int ($w / 4) + ($w % 2);
            print SCR 'select pin "TBUF_X', $x, 'Y', $y, '.0"', "\n";
        }
    }
    print SCR 'route', "\n";
    print SCR 'unselect -all', "\n";
    $b++;
    $w++;
}

print SCR 'save macro', "\n";
print SCR 'quit!', "\n";
print SCR 'end', "\n";
```

```

# Create template top VHDL-file

open VHDL, "> comp-$name.vhd" or die $!;

print VHDL "\n";
print VHDL '-----', "\n";
print VHDL '-- Top template with 1 static module 1 bus macro', "\n";
print VHDL '-- and ', $modules - 1, ' reconfigurable modules', "\n";
print VHDL '-----', "\n";
print VHDL "\n";
print VHDL '-----', "\n";
print VHDL '-- This file is an automatically generated VHDL', "\n";
print VHDL '-- template file.', "\n";
print VHDL '-- Do NOT change anything listed below:', "\n";
print VHDL '-- The bus macro component: ', $name, "\n";
print VHDL '-- The reconfigurable module components: top1 - top', $modules - 1, "\n";
print VHDL '-- The signals connecting these: signal bm_*', "\n";
print VHDL '-- DO use the defined signals not beginning with bm_*', "\n";
print VHDL '-- in order to connect to the bus macr, i.e. the', "\n";
print VHDL '-- reconfigurable modules.', "\n";
print VHDL '-- Add the following:', "\n";
print VHDL '-- Port definitions to connect to the external I/O', "\n";
print VHDL '-- Ports needed to the static module.', "\n";
print VHDL '-- Signals and other static modules as needed.', "\n";
print VHDL '-----', "\n";
print VHDL "\n";
print VHDL 'library ieee;', "\n";
print VHDL 'use ieee.std_logic_1164.all;', "\n";
print VHDL "\n";
print VHDL '-----', "\n";
print VHDL "\n";
print VHDL 'entity top is', "\n";
print VHDL ' port (' "\n";
print VHDL '     clk           : in std_logic;', "\n";
print VHDL "\n";
print VHDL '     -- Fill in top entity port specification', "\n";
print VHDL "\n";
print VHDL '     ext_gnd       : in std_logic;', "\n";
print VHDL '     ext_vcc       : in std_logic;', "\n";
$m = 1;
while ($m < $modules) {
    print VHDL '     ext_gnd_RM', $m, ' : in std_logic;', "\n";
    print VHDL '     ext_vcc_RM', $m, ' : in std_logic;', "\n";
    $m++;
}
print VHDL ' );', "\n";
print VHDL 'end top;', "\n";
print VHDL "\n";
print VHDL '-----', "\n";
print VHDL "\n";
print VHDL 'architecture top_behav of top is', "\n";
print VHDL "\n";
print VHDL '-- Declare modules at top-level to get port directionality', "\n";
print VHDL "\n";

```

APPENDIX B. SOURCE CODE

```
print VHDL '-- Special module: The Bus Macro!', "\n";
print VHDL '-- It is a Hard Macro dedicated for communication between the leftmost', "\n";
print VHDL '-- static module and all the reconfigurable modules.', "\n";
print VHDL "\n";
print VHDL ' component ', $name, "\n";
print VHDL ' port (' , "\n";

$m = 0;
while ($m < $modules) {
  print VHDL '      addrI', $m, ' : in std_logic_vector (' , $addr - 1, ' downto 0);', "\n";
  print VHDL '      addrT', $m, ' : in std_logic_vector (' , $addr - 1, ' downto 0);', "\n";
  if ($m == 0) {
    print VHDL '      addr0 : out std_logic_vector (' , $addr - 1, ' downto 0);', "\n";
  }
  print VHDL '      dataI', $m, ' : in std_logic_vector (' , $data - 1, ' downto 0);', "\n";
  print VHDL '      dataT', $m, ' : in std_logic_vector (' , $data - 1, ' downto 0);', "\n";
  if ($m == 0) {
    print VHDL '      data0 : out std_logic_vector (' , $data - 1, ' downto 0);', "\n";
  }
  print VHDL '      readyI', $m, ' : in std_logic;', "\n";
  print VHDL '      readyT', $m, ' : in std_logic;', "\n";
  if ($m == 0) {
    print VHDL '      ready0 : out std_logic;', "\n";
  }
  if ($m == 0) {
    print VHDL '      csI', $m, ' : in std_logic_vector (' , $cs, ' downto 1);', "\n";
    print VHDL '      csT', $m, ' : in std_logic_vector (' , $cs, ' downto 1);', "\n";
  }
  if ($m != 0) {
    print VHDL '      csI', $m, ' : in std_logic;', "\n";
    print VHDL '      csT', $m, ' : in std_logic;', "\n";
    print VHDL '      csO', $m, ' : out std_logic;', "\n";
  }
  if ($m == 0) {
    print VHDL '      weI', $m, ' : in std_logic_vector (' , $we, ' downto 1);', "\n";
    print VHDL '      weT', $m, ' : in std_logic_vector (' , $we, ' downto 1);', "\n";
  }
  if ($m != 0) {
    print VHDL '      weI', $m, ' : in std_logic;', "\n";
    print VHDL '      weT', $m, ' : in std_logic;', "\n";
    print VHDL '      weO', $m, ' : out std_logic;', "\n";
  }
  if ($m == 0) {
    print VHDL '      irqI', $m, ' : in std_logic_vector (' , $irq, ' downto 1);', "\n";
    print VHDL '      irqT', $m, ' : in std_logic_vector (' , $irq, ' downto 1);', "\n";
    print VHDL '      irq0 : out std_logic_vector (' , $irq, ' downto 1);', "\n";
  }
  if ($m != 0) {
    print VHDL '      irqI', $m, ' : in std_logic;', "\n";
    print VHDL '      irqT', $m, ' : in std_logic;', "\n";
  }
  if ($m == 0) {
    print VHDL '      resetI', $m, ' : in std_logic_vector (' , $cs, ' downto 1);', "\n";
    print VHDL '      resetT', $m, ' : in std_logic_vector (' , $cs, ' downto 1);', "\n";
  }
  if ($m != 0) {
    print VHDL '      resetI', $m, ' : in std_logic;', "\n";
  }
}
```

```

print VHDL '         resetT', $m, ' : in std_logic;', "\n";
print VHDL '         reset0', $m, ' : out std_logic';
if ($m != $modules -1) {
    print VHDL ' ';
}
print VHDL "\n";
}
if ($m != $modules - 1) {
    print VHDL "\n";
}
$m++;
}
print VHDL ' );', "\n";
print VHDL ' end component;', "\n";
print VHDL "\n";
print VHDL '-- Special module: The reconfigurable module!', "\n";
print VHDL '-- The interface to a reconfigurable module.', "\n";
print VHDL '-- Must not be altered.', "\n";
print VHDL "\n";

$m = 1;
while ($m < $modules) {
    print VHDL ' component pos', $m, "\n";
    print VHDL ' port (' , "\n";
    print VHDL '         clk : in std_logic;', "\n";
    print VHDL '         addr : in std_logic_vector (' , $addr - 1, ' downto 0);', "\n";
    print VHDL '         dataI : in std_logic_vector (' , $data - 1, ' downto 0);', "\n";
    print VHDL '         data0 : out std_logic_vector (' , $data - 1, ' downto 0);', "\n";
    print VHDL '         ready : out std_logic;', "\n";
    print VHDL '         cs : in std_logic;', "\n";
    print VHDL '         we : in std_logic;', "\n";
    print VHDL '         irq : out std_logic;', "\n";
    print VHDL '         reset : in std_logic;', "\n";
    print VHDL '         gnd : in std_logic;', "\n";
    print VHDL '         vcc : in std_logic;', "\n";
    print VHDL ' );', "\n";
    print VHDL ' end component;', "\n";
    print VHDL "\n";
    $m++;
}

print VHDL ' component static_module', "\n";
print VHDL ' port (' , "\n";
print VHDL '         clk : in std_logic;', "\n";
print VHDL '         addr : out std_logic_vector (' , $addr - 1, ' downto 0);', "\n";
print VHDL '         dataI : in std_logic_vector (' , $data - 1, ' downto 0);', "\n";
print VHDL '         data0 : out std_logic_vector (' , $data - 1, ' downto 0);', "\n";
print VHDL '         ready : in std_logic;', "\n";
print VHDL '         cs : out std_logic_vector (' , $modules - 1, ' downto 1);', "\n";
print VHDL '         we : out std_logic_vector (' , $modules - 1, ' downto 0);', "\n";
print VHDL '         irq : in std_logic_vector (' , $modules - 1, ' downto 1);', "\n";
print VHDL '         reset : out std_logic_vector (' , $modules - 1, ' downto 1);', "\n";
print VHDL "\n";
print VHDL '         -- Typically ports for external I/O are put here', "\n";
print VHDL '         -- :', "\n";
print VHDL '         -- :', "\n";
print VHDL "\n";

```

APPENDIX B. SOURCE CODE

```
print VHDL '    );','\n";
print VHDL ' end component;','\n";
print VHDL "\n";

print VHDL "\n";
print VHDL '-- User signals: Add signals below as needed.','\n";
print VHDL "\n";
print VHDL "\n";
print VHDL "\n";
print VHDL ' signal clk      : std_logic;','\n";
print VHDL "\n";
print VHDL '-- Bus signals: The signals from the used defined static module(s)','\n";
print VHDL '-- to the bus macro.','\n";
print VHDL '-- Recomending no changes.','\n";
print VHDL '-- Make use of these signals in order to communicate with the','\n";
print VHDL '-- bus macro, i.e. the reconfigurable modules.','\n";
print VHDL "\n";
print VHDL ' signal addr     : std_logic_vector (',$addr - 1,' downto 0);','\n";
print VHDL ' signal dataI    : std_logic_vector (',$data - 1,' downto 0);','\n";
print VHDL ' signal dataO    : std_logic_vector (',$data - 1,' downto 0);','\n";
print VHDL ' signal ready   : std_logic;','\n";
print VHDL ' signal cs      : std_logic_vector (',$modules - 1,' downto 1);','\n";
print VHDL ' signal we      : std_logic_vector (',$modules - 1,' downto 0);','\n";
print VHDL ' signal irq     : std_logic_vector (',$modules - 1,' downto 1);','\n";
print VHDL ' signal reset   : std_logic_vector (',$modules - 1,' downto 1);','\n";
print VHDL ' signal gnd     : std_logic;','\n";
print VHDL ' signal vcc     : std_logic;','\n";
print VHDL "\n";
print VHDL '-- Internal signals: The signals between reconfigurable','\n";
print VHDL '-- modules and the bus macro.','\n";
print VHDL '-- Must not be altered or used.','\n";
print VHDL "\n";
print VHDL ' signal bm_addr  : std_logic_vector (',$addr - 1,' downto 0);','\n";
print VHDL "\n";
$m = 1;
while ($m < $modules) {
    print VHDL ' signal bm_data0',$m,' : std_logic_vector (',$data - 1,' downto 0);','\n";
    print VHDL ' signal bm_ready',$m,' : std_logic;','\n";
    print VHDL ' signal bm_cs',$m,'   : std_logic;','\n";
    print VHDL ' signal bm_we',$m,'   : std_logic;','\n";
    print VHDL ' signal bm_irq',$m,'   : std_logic;','\n";
    print VHDL ' signal bm_reset',$m,' : std_logic;','\n";
    print VHDL ' signal bm_gnd',$m,'   : std_logic;','\n";
    print VHDL ' signal bm_vcc',$m,'   : std_logic;','\n";
    print VHDL "\n";
    $m++;
}

print VHDL "\n";
print VHDL 'begin','\n";

print VHDL "\n";
print VHDL '-- Instansiated Bus Macro:','\n";
print VHDL '-- Must not be altered.','\n";
print VHDL "\n";

print VHDL ' bus_macro : ','$name,"\n";
```

```

print VHDL '    port map(','\n";
$m = 0;
while ($m < $modules) {
  if ($m == 0) {
    print VHDL '        addrI',$m,' => addr,'\n";
    print VHDL '        addrT',$m,' => (others => gnd),'\n";
    print VHDL '        addr0  => bm_addr,'\n";
  }
  if ($m != 0) {
    print VHDL '        addrI',$m,' => (others => bm_vcc',$m,')', -- Dummy signal in,'\n";
    print VHDL '        addrT',$m,' => (others => bm_vcc',$m,')', -- High-impedance,'\n";
  }
  if ($m != 0) {
    print VHDL '        dataI',$m,' => bm_data0',$m,',' ,'\n";
    print VHDL '        dataT',$m,' => (others => bm_we',$m,') ,'\n";
  }
  if ($m == 0) {
    print VHDL '        dataI',$m,' => data0,'\n";
    print VHDL '        dataT',$m,' => (others => we(','$m,') ,'\n";
    print VHDL '        data0  => dataI,'\n";
  }
  if ($m != 0) {
    print VHDL '        readyI',$m,' => bm_ready',$m,',' ,'\n";
    print VHDL '        readyT',$m,' => bm_cs',$m,',' ,'\n";
  }
  if ($m == 0) {
    print VHDL '        readyI',$m,' => vcc,          -- Dummy signal in,'\n";
    print VHDL '        readyT',$m,' => vcc,          -- High-impedance,'\n";
    print VHDL '        ready0  => ready,'\n";
  }
  if ($m == 0) {
    print VHDL '        csI',$m,' => cs,'\n";
    print VHDL '        csT',$m,' => (others => gnd),'\n";
  }
  if ($m != 0) {
    print VHDL '        csI',$m,' => bm_vcc',$m,')', -- Dummy signal in,'\n";
    print VHDL '        csT',$m,' => bm_vcc',$m,')', -- High-impedance,'\n";
    print VHDL '        cs0',$m,' => bm_cs',$m,',' ,'\n";
  }
  if ($m == 0) {
    print VHDL '        weI',$m,' => we(','$modules - 1,' downto 1),'\n";
    print VHDL '        weT',$m,' => (others => gnd),'\n";
  }
  if ($m != 0) {
    print VHDL '        weI',$m,' => bm_vcc',$m,')', -- Dummy signal in,'\n";
    print VHDL '        weT',$m,' => bm_vcc',$m,')', -- High-impedance,'\n";
    print VHDL '        we0',$m,' => bm_we',$m,',' ,'\n";
  }
  if ($m != 0) {
    print VHDL '        irqI',$m,' => bm_irq',$m,',' ,'\n";
    print VHDL '        irqT',$m,' => bm_gnd',$m,',' ,'\n";
  }
  if ($m == 0) {
    print VHDL '        irqI',$m,' => (others => vcc),    -- Dummy signal in,'\n";
    print VHDL '        irqT',$m,' => (others => vcc),    -- High-impedance,'\n";
    print VHDL '        irq0  => irq,'\n";
  }
}

```

APPENDIX B. SOURCE CODE

```
if ($m == 0) {
    print VHDL '          resetI', $m, ' => reset, ', "\n";
    print VHDL '          resetT', $m, ' => (others => gnd), ', "\n";
}
if ($m != 0) {
    print VHDL '          resetI', $m, ' => bm_vcc', $m, ',           -- Dummy signal in', "\n";
    print VHDL '          resetT', $m, ' => bm_vcc', $m, ',           -- High-impedance', "\n";
    print VHDL '          resetO', $m, ' => bm_reset', $m;
    if ($m != $modules - 1) {
        print VHDL ', '
    }
    print VHDL "\n";
}
if ($m != $modules - 1) {
    print VHDL "\n";
}
$m++;
}
print VHDL '    ); ', "\n";
print VHDL "\n";

$m = 1;
while ($m < $modules) {
    print VHDL "\n";
    print VHDL '-- Instansiated Reconfigurable Modules: ', "\n";
    print VHDL '-- Must not be altered. ', "\n";
    print VHDL "\n";

    print VHDL '    reconfigurable_module', $m, ' : pos', $m, "\n";
    print VHDL '    port map( ', "\n";
    print VHDL '        clk    => clk, ', "\n";
    print VHDL '        addr    => bm_addr, ', "\n";
    print VHDL '        dataI    => dataI, ', "\n";
    print VHDL '        dataO    => bm_dataO', $m, ', ', "\n";
    print VHDL '        ready    => bm_ready', $m, ', ', "\n";
    print VHDL '        cs       => bm_cs', $m, ', ', "\n";
    print VHDL '        we       => bm_we', $m, ', ', "\n";
    print VHDL '        irq      => bm_irq', $m, ', ', "\n";
    print VHDL '        reset    => bm_reset', $m, ', ', "\n";
    print VHDL '        gnd      => bm_gnd', $m, ', ', "\n";
    print VHDL '        vcc      => bm_vcc', $m, "\n";
    print VHDL '    ); ', "\n";
    print VHDL "\n";
    $m++;
}

print VHDL "\n";
print VHDL '-- Instansiated Static module: ', "\n";
print VHDL '-- This is a template module. The ports used must be present', "\n";
print VHDL '-- but can be lifted out to more static modules or external I/O', "\n";
print VHDL '-- ports. Ports from static modules to external I/O-interface', "\n";
print VHDL '-- must also be added. ', "\n";
print VHDL "\n";

print VHDL '    static_module_pos0 : static_module', "\n";
print VHDL '    port map( ', "\n";
print VHDL '        clk    => clk, ', "\n";
```

```

print VHDL '          addr => addr,'"\n";
print VHDL '          dataI => dataI,'"\n";
print VHDL '          data0 => data0,'"\n";
print VHDL '          ready => ready,'"\n";
print VHDL '          cs    => cs,'"\n";
print VHDL '          we    => we,'"\n";
print VHDL '          irq   => irq,'"\n";
print VHDL '          reset => reset,'"\n";
print VHDL "\n";
print VHDL '          -- Typically ports for external I/O are put here,'"\n";
print VHDL '          -- :,'"\n";
print VHDL '          -- :,'"\n";
print VHDL "\n";
print VHDL '          gnd  => gnd,'"\n";
print VHDL '          vcc  => vcc,'"\n";
print VHDL '      );,'"\n";
print VHDL "\n";

print VHDL "\n";
print VHDL '-- Connect external gnd and vcc to the modules:'"\n";
print VHDL "\n";

print VHDL '  gnd <= ext_gnd;'"\n";
print VHDL '  vcc <= ext_vcc;'"\n";
$m = 1;
while ($m < $modules) {
  print VHDL '  bm_gnd',$m,' <= ext_gnd_RM',$m,';'"\n";
  print VHDL '  bm_vcc',$m,' <= ext_vcc_RM',$m,';'"\n";
  $m++;
}
print VHDL "\n";
print VHDL 'end top_behav;'"\n";
print VHDL "\n";

```

B.2.3 Synthesize top VHDL config file

top.prj

```

#-- Synplicity, Inc.
#-- Version 7.3.4
#-- Project file for tops
#-- Written on Thu Mar 04 10:42:31 2004

#add_file options
add_file -vhdl -lib work "top_file_name.vhd"
add_file "ucf_file.ucf"

#implementation: "rev_1"
impl -add rev_1

#device options
set_option -technology VIRTEX2
set_option -part XC2V1000
set_option -package FG456
set_option -speed_grade -4

```

```
#compilation/mapping options
set_option -default_enum_encoding default
set_option -symbolic_fsm_compiler 1
set_option -resource_sharing 1
set_option -use_fsm_explorer 0

#map options
set_option -frequency 1.000
set_option -fanout_limit 10000
set_option -disable_io_insertion 0
set_option -pipe 0
set_option -update_models_cp 0
set_option -verification_mode 0
set_option -fixgatedclocks 0
set_option -modular 1
set_option -retiming 0

#simulation options
set_option -write_verilog 0
set_option -write_vhdl 0

#automatic place and route (vendor) options
set_option -write_apr_constraint 1

#set result format/file last
project -result_file "rev_1/top_file_name.edf"

#implementation attributes
set_option -vlog_std v2001
set_option -auto_constrain_io 0
impl -active "rev_1"
```

B.2.4 Synthesize module VHDL config file module.prj

```
!-- Synplicity, Inc.
!-- Version 7.3.4
!-- Project file for modules
!-- Written on Thu Mar 04 13:44:24 2004

#add_file options
add_file -vhdl -lib work "module_file_name.vhd"

#implementation: "rev_1"
impl -add rev_1

#device options
set_option -technology VIRTEX2
set_option -part XC2V1000
set_option -package FG456
set_option -speed_grade -4

#compilation/mapping options
set_option -default_enum_encoding default
```

```
set_option -symbolic_fsm_compiler 1
set_option -resource_sharing 1
set_option -use_fsm_explorer 0

#map options
set_option -frequency 1.000
set_option -fanout_limit 10000
set_option -disable_io_insertion 1
set_option -pipe 0
set_option -update_models_cp 0
set_option -verification_mode 0
set_option -fixgatedclocks 0
set_option -modular 1
set_option -retiming 0

#simulation options
set_option -write_verilog 0
set_option -write_vhdl 0

#automatic place and route (vendor) options
set_option -write_apr_constraint 1

#set result format/file last
project -result_file "rev_1/module_file_name.edf"

#implementation attributes
set_option -vlog_std v2001
set_option -auto_constrain_io 0
impl -active "rev_1"
```

B.2.5 Program FPGA config file bitgen_v2_jtag.ut

```
-w
-l
-m
-g ReadBack
-g DebugBitstream:No
-g CRC:Enable
-g ConfigRate:4
-g CclkPin:PullUp
-g MOPin:PullUp
-g M1Pin:PullUp
-g M2Pin:PullUp
-g ProgPin:PullUp
-g DonePin:PullUp
-g DriveDone:No
-g PowerdownPin:PullUp
-g TckPin:PullUp
-g TdiPin:PullUp
-g TdoPin:PullNone
-g TmsPin:PullUp
-g UnusedPin:PullUp
-g UserID:0xFFFFFFFF
-g DCMSHutDown:Disable
-g DisableBandgap:No
-g StartUpClk:JtagClk
```

```
-g DONE_cycle:4
-g GTS_cycle:5
-g GWE_cycle:6
-g LCK_cycle:NoWait
-g Match_cycle:NoWait
-g Security:None
-g Persist:No
-g DonePipe:No
-g Encrypt:No
```