OneChip: An FPGA Processor With Reconfigurable Logic

by

Ralph D. Wittig

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

OneChip: An FPGA Processor With Reconfigurable Logic

Ralph D. Wittig

Master of Applied Science, 1995

Department of Electrical and Computer Engineering
University of Toronto

Abstract

This thesis describes a processor architecture called *OneChip*, which combines a fixed logic processor core and reconfigurable logic resources. Using the variable components of this architecture, the performance of speed-critical applications can be improved by customizing OneChip's execution units or flexibility can be added to the glue logic interfaces of embedded controller type applications.

This work eliminates the shortcomings of other custom compute machines by tightly integrating the reconfigurable resources into a MIPS-like processor. The details of the core processor, the fixed to reconfigurable logic interface and the actual reconfigurable structures are described.

To study OneChip's feasibility, a 32-bit processor as well as several performance enhancement and embedded controller type applications are implemented on the Transmogrifier-1 field programmable system. It is shown that application speedups of over 40 are achievable. However, the design flexibility introduced with the use of less dense, reconfigurable structures carries an area penalty of no less than 3.5 times the size of the custom silicon design implementation.

Acknowledgments

I would like to thank my supervisor, Professor Paul Chow, for all his counselling and advice. His friendly and informal help with technical and every day problems made this work a truly enjoyable experience.

I am tremendously grateful to my parents Gisela and Detlef for their unconditional love, their guidance and their financial support. Moreover, I truly appreciate my better half Reena for her dedication, endless love and unconditional understanding throughout the hard work of this thesis. Thanks also to my sister Manon and her man Florian for being a great family to me.

I am very thankful to David Galloway for providing me with steadfast advice on Transmogrifier related problems and to many fellow grads who helped me out with any other technical questions. Especially my close friends Mark, Vaughn, Sam and Dave Y., but also the rest of LP392 provided a very enjoyable research environment, which often stretched far beyond the engineering aspects.

Thanks also to Pete, Mike, Dan, the Horizon Bunch and other fellow downtowners for being a great group of friends. I truly enjoyed all the fabulous times we had.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Motivation

Although field programmable gate arrays (FPGA) were introduced a decade ago, they have only recently become more popular . This is not only due to the fact that programmable logic saves development cost and time over increasingly complex ASIC designs, but also because the gate count per FPGA chip has reached numbers that allow for the implementation of more complex applications.

Many present day applications utilize a processor and other logic on two or more separate chips. However , with the anticipated ability to build chips with over ten million transistors, it will become possible to implement a processor within a sea of programmable logic, all on one chip. Such a design approach would allow a great degree of programmability freedom, both in hardware and in software: CAD tools could decide which parts of a source code program are actually to be executed in software and which other parts are to be implemented with hardware. The hardware may be needed for application interfacing reasons or may simply represent a coprocessor used to improve execution time.

Most computationally complex applications spend 90% of their execution time in only 10% of their code [1]. The basic instructions executed in this 10% of the code of a given program naturally differ from application to application. These observations make the idea of a fast, yet general purpose CPU seem inconsistent. The custom compute machine (CCM), which can be customized on a per application basis, appears to be the solution to the contradiction of general purpose computing and high performance processing.

1.2 Objectives

Various research organizations have recently studied the benefits of using reconfigurable logic for building CCMs. Many interesting systems have been designed, spanning a broad range of interconnection architectures and usable gate counts. They can be broadly categorized by their degree of processor - coprocessor coupling and by the size of the applications to be executed on the reconfigurable part of the CCM. An overview of representative systems will be given in the next chapter .

Independent of the size of the reconfigurable compute element(s), all of these early CCMs are loosely coupled systems with a clearly identifiable processor - coprocessor frontier which always crosses the boundary of one or more chips. However , loose coupling is one of the main limitations for obtaining high speed ups from reconfigurable compute engines. From previous work [2] it is known that achieving reasonable performance from CCMs hinges on two crucial issues which can be identified as:

- the inflexible coprocessor access protocol (control flow)
- the processor - coprocessor bandwidth limitation (data flow)

The objective of this thesis is to investigate a coupling scheme that links the fixed microprocessor and the reconfigurable logic closer than in any of the current CCM systems. The effects of this tight integration scheme on the above speed-up limiting factors will be studied. As illustrated in Figure 1-1, it is envisioned to place both types of logic onto a single

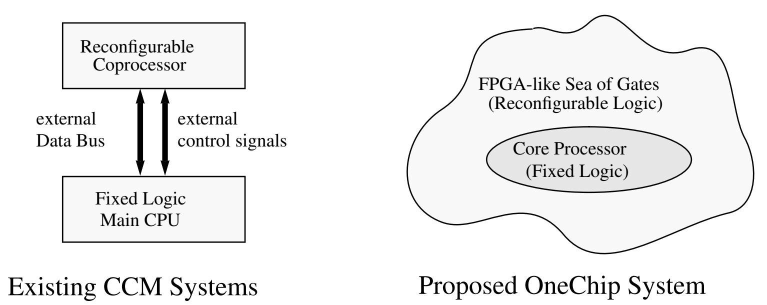


Fig. 1-1: CCM Reconfigurable Logic Integration Scheme

chip - a system called *OneChip* . The investigation will not only include ways to reduce the interfacing bottleneck, but will also address the issue of where the frontier of fixed and reconfigurable logic should be drawn or which parts of a CCM system are always needed and which other parts can be left flexible.

Programmable logic need not only be used for application speed-up, it can also be employed as intelligent glue logic for custom interfacing purposes such as in embedded controller applications. Current single-chip embedded processors attempt to provide very flexible interfaces that can be used in a large number of applications. However, they can often result in interfaces that are less efficient than intended. Furthermore, it might be desirable to perform some bit-level data computations in-between the main processor and the actual I/O interface. This thesis also investigates the requirements for providing a general purpose field-configurable interface for embedded processor applications.

The OneChip system, in today's technology requires custom silicon to be developed. Nonetheless, noting that the architectural issues involved in a tightly coupled system may be evaluated using UofT's Transmogrifier-1 (TM-1) board [3], which is a collection of programmable gate arrays, a working prototype system modelling the envisioned OneChip CCM was developed for feasibility and benchmarking purposes.

Along with the investigations outlined above, several CAD tools for FPGA design implementation and simulation were used. Many of them are either not fully functional, at a very early evolutionary stage, or not directly suited for a given task. As part of the prototyping process, this thesis was also concerned with studying and developing a feasible logic design process. Included in this tool study are a VHDL for synthesis design methodology as well as tool integration issues involved in the flow from a high level VHDL description to a fully simulated, synthesized, partitioned, placed and routed, multi-FPGA logic design.

With the above objectives set, the only element missing from a functional, tightly-coupled CCM is the software required to program the system. However, to design a fully automated CAD tool that identifies and separates a high-level language program into separate software and hardware images and to program the reconfigurable and the fixed parts of the CCM with these respective images is beyond the scope of this thesis, and is left as a challenge for future work.

1.3 Thesis Organization

This thesis is divided into six chapters. Chapter 2 provides the reader with some background information on existent CCM systems, with information on software issues involved in programming reconfigurable compute engines and with information on existent processor architectures. Chapter 3 introduces a VHDL for synthesis design methodology and discusses issues involved in the integration of CAD tools needed for the VHDL to FPGA design flow. Chapter 4 discusses the design of the 32-bit RISC CPU used as the fixed processor and issues involved in its implementation on the TM-1 board. Chapter 5 addresses interfacing issues involved in the integration of reconfigurable compute and glue logic elements into the fixed processor and describes two applications. Chapter 6 concludes this thesis and offers recommendations for future work.

Chapter

2

Background

This chapter is intended as a brief review for the reader already familiar with the major concepts and terminologies of the work addressed by this thesis - the areas of processors, field programmable gate arrays, and custom computing applications involving the former. By reviewing relevant related work and by pointing out the importance of some of the successes and shortcomings of the previous work, the reader is introduced and critically prepared for the ideas and their implications presented in the following chapters.

First, a survey of existent and proposed custom compute machines (CCM) is given. This overview is not intended to be complete, but will rather identify the most successful CCMs to date, categorized by the type of fixed to reconfigurable logic interface. Only two or three candidates per category are analyzed in detail, however, most existent reconfigurable compute engines can be matched to one of the presented interfacing classes.

Having introduced some of the hardware issues involved in CCMs, the second section of this chapter addresses the software requirements of CCMs. Although this thesis deals only with the hardware problems of custom computing, the reader must still understand the complex software issues involved in programming a compute engine and in executing code on such a specialized processing environment to be able to judge the decisions made when designing the corresponding hardware.

The last section of this chapter is a review of issues concerning processor performance and a comparison of the CISC and RISC processing paradigms. With this analytical comparison the reader is introduced to the architectural problem of tightly coupling reconfigurable logic to a fixed main processing element.

2.1 Custom Compute Machines Survey

Ever since FPGAs allowed for the implementation of more complex functions representing more than just simple sequential circuits, researchers have been utilizing these reconfigurable devices to speed up applications that are ill-suited for computation on general purpose processing elements by implementing algorithms or parts thereof in custom processing hardware. FPGAs have been employed for virtually every type of application ranging from very fine grained, highly repetitive to more coarse grained, general purpose computational tasks [2][4][5][6][7][8]. The following is a survey of the most successful CCM systems broken down into the following categories:

- systems loosely linking reconfigurable logic to a fixed, front end host computer
- systems loosely linking reconfigurable logic to a fixed, integrated CPU
- systems closely linking reconfigurable logic to a fixed, integrated CPU

One could further categorize CCM systems according to the interconnection of the FPGAs in the reconfigurable array , the number of usable gates per reconfigurable block and the grain size of the applications to be executed on the reconfigurable array . However , the three categories listed above are sufficient to study reconfigurable to fixed logic interfacing issues identified as speedup limiting factors by Jeschke [2]. Furthermore, it should be noted that, to the best of the authors' knowledge, at the present time no working hardware system other than the one presented in this thesis exists for the last of the listed categories.

2.1.1 Loosely Coupled CCM Systems W ith Front End Host

CCM systems of this category follow the configuration depicted in Figure 2-1. The three systems discussed are chosen because they have been very successful at speeding up a broad range of applications.

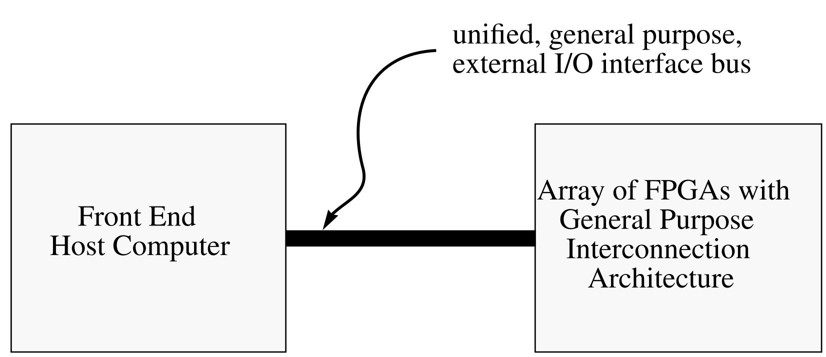


Fig. 2-1: Loosely Coupled CCM W ith Front End Host

DEC PeRLe

One of the early applications of FPGAs as a universal, reconfigurable hardware coprocessor came from the DEC Paris Research Laboratory [9]. The DEC PeRLe-0 and DEC PeRLe-1 [10] of DEC's Programmable Active Memories (PAM) group are CCM systems that attach an FPGA board to the system I/O bus of a host workstation. For the PeRLe-0, twenty-five Xilinx 3020 FPGAs [11] are hard-wired in a 5x5 two-dimensional nearest-neighbor connection mesh to each other and on a global bus to two 32-bit wide memory banks. The PeRLe-1 follows the same approach, only employing larger FPGAs (twenty-four Xilinx 3090 [11]) and larger RAM. Typical applications running on the PeRLe-1 CCM are computationally complex and require low communication overhead: a Long Integer Multiplier, a Discrete Cosine Transform (DCT) and an RSA Decrypter, for example, running from 25 to 33 MHz and achieving 250 to 264 Gbops (billion binary operations per second) [10].

SRC Splash

Inspired by the performance results of the PAM group, the Supercomputing Research Centre (SRC) developed the Splash series of compute engines [12][13]. The Splash-2 is an improved version of the earlier Splash-1 system. Its FPGA board employs sixteen Xilinx 4010 FPGAs [11] that are linked with each other in a linear array. Each FPGA also directly attaches to a 256k x 16 RAM and a 16 x 36 bit bi-directional crossbar. The memories are also linked amongst themselves on a global data bus. One or more of such FPGA boards are linked through a special bus to a front end host computer, essentially forming a systolic array. Again, applications running on this CCM are computationally complex on a per FPGA board basis, requiring a rather low front end host <-> back end FPGA board and inter FPGA board communication overhead. Applications include text searching, capable of scanning 50 million characters per second [5]. and genome sequence matching, achieving search rates of 12 million characters per second [6].

NCSU AnyBoard

A somewhat smaller FPGA board-based CCM is the AnyBoard system built by the North Carolina State University (NCSU) [8]. It employs only five Xilinx 3090 FPGAs [11] which are connected in a linear array through local buses. Three of the FPGAs are each coupled to the data port of a 128k x 8 bit RAM, while a forth FPGA supplies the three SRAMs with the address. The fifth FPGA is not attached to the memories in any way, but rather to the parallel bus of a PC, providing the programming and the control flow for the first four FPGAs. Overall, each of the five FPGAs attaches to a global data bus, which also connects the whole board to the host’s data bus. AnyBoard applications include neural networks, motor controllers and micro-sequencers, to name a few. However, performance enhancement applications are not presented in [8], noting that the system’s slow PC to FPGA board interface is a bottleneck that does not allow for the implementation of fast processing applications.

2.1.2 Loosely Coupled CCM Systems With Integrated CPU

Employing a somewhat closer degree of coupling than the systems presented in the previous section, the CCMs of this category utilize separate, dedicated interconnect resources for control and data flow signals. Also, as illustrated in Figure 2-2, both types of signals no longer pass through a general I/O interface, but rather directly attach to the local bus and/or dedicated pins of the integrated CPU.

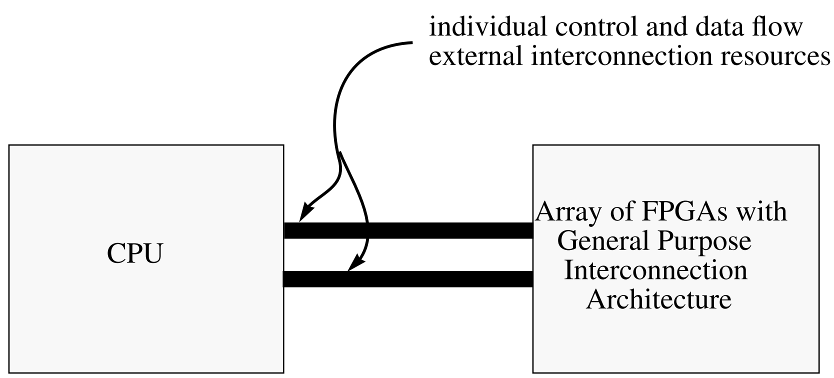


Fig. 2-2: Loosely Coupled CCM With Integrated CPU

BU PRISM

The first reconfigurable compute engine to directly interface FPGAs to a microprocessor was Brown University’s PRISM I system [4]. The project has evolved into the PRISM II CCM [14], overcoming some of the limitations of the earlier version. The PRISM concept links a Motorola 68010 microprocessor running at 10 MHz to a board based FPGA array consisting of four Xilinx 3090 FPGAs [11]. The PRISM concept does not intend to perform entire applications in hardware like the PeRLe and Splash projects, but rather concentrates on augmenting the general purpose Motorola CPU with a hardware instruction set representing only some subset of C functions of the program executing on the CCM. Speedup factors of the hardware implementation of functions range from 5 to 50 [4]. Complete program speedup numbers are limited by the processor - coprocessor interface, even though the communication is limited to data flow consisting of the arguments and corresponding computed results to and from the hardware C function.

UofT Reconfigurable Coprocessor

Another CCM linking its reconfigurable resources directly to a stand-alone microprocessor was developed at the University of Toronto [2]. It is designed to be interfaced to an Intel 80286 microprocessor through the external math coprocessor (80287) port. The system uses seven Xilinx 3042-100 FPGAs [11], two of which act as coprocessor interface chips, another one is used for board programming and debugging purposes, one is dedicated to accessing the system's 1M x 9 bit DRAM and the remaining three FPGAs provide the freely programmable logic. All the FPGAs are linked to vertically and horizontally adjacent neighbors and to a coprocessor board internal data bus. Demonstrated applications include a drug diffusion simulation and a DES program [2]. Performance results for the hardware implementation of some subroutines found in these applications result in speedups from 1.33 to 2. However, overall application speedup could not be achieved with this early prototype due to the system's inefficient data and control flow processor - coprocessor interface.

2.1.3 Closely Coupled CCM Systems With Integrated CPU

To the best of the authors' knowledge, at the time of preparation of this thesis no working closely coupled CCM with integrated CPU exists besides the one presented later in this work. In general, systems of this category attempt to link reconfigurable resources to some internal connection point within the core CPU. As depicted in Figure 2-3, the tight integration of this interfacing scheme allows the reconfigurable resources to be placed onto the same chip as the fixed logic processor.

MIT DPGA Coupled Microprocessor

The work presented by DeHon [15] is mostly concerned with providing an overview of technology trends, existing CCMs, their shortcomings and possible future reconfigurable systems. It attempts to apply the architecture of a recently presented FPGA, called the DPGA [16], to a microprocessor integrated reconfigurable system. While it does not present a

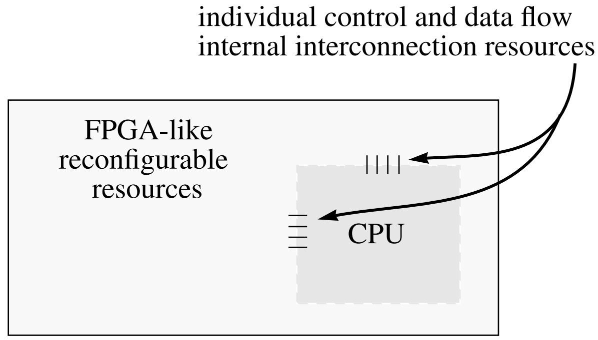


Fig. 2-3: Closely Coupled CCM W ith Integrated CPU

detailed new fixed / reconfigurable logic interface, it clearly identifies the communication bandwidth and latency of such an interface as the throughput limiting factors and stresses that a successful CCM, which integrates the DPGA with a microprocessor , will be closely coupled

Harvard PRISC

While this proposed CCM system has only been explored in simulations [17], unlike the work presented in [15], its architecture is clearly defined. PRISC takes a fast, 200 MHz RISC CPU and augments the datapaths existent functionality . The reconfigurable logic is integrated into the microprocessor by adding one programmable functional unit (PFU) in parallel to existing functional units (FU) like the arithmetic logical unit (ALU). A PFU is implemented as a regular structure of interconnects and look-up tables (LUT). The complexity of functions to be implemented in a PFU is such that their latency does not exceed the cycle time of the microprocessor . Context switching amongst precompiled PFU images is supported and different hardware functions are addressed through extensions to the existent instruction set of the fixed-logic CPU. The microprocessor to reconfigurable logic interface enables PFUs to make use of the existent datapath functionality like source and result operand handling as well as hazard detection and forwarding. Through this tight integration of the PFU into the CPU, data and control flow overhead are kept at a lower cost than in any of the previously discussed CCMs. The purely combinational nature of the PFU requires the automatic hardware function extraction software tool to generate hardware images with a small granularity . Nonetheless, simulated SPECint92 benchmarks showed speedups ranging from 1.06 to 1.91 for the complete program executed on PRISC over a software only evaluation on a RISC machine without the extra PFU [17].

2.1.4 Comments On CCM Architectures

While the above sampling of CCM systems is by no means complete - an up to date list can be obtained from [18] - most of the existent reconfigurable compute engines can be included into one of the presented fixed / reconfigurable logic interface categories. When analyzing the architectural concepts and the shortcomings of the three classes, several interesting observations can be made.

Amongst the loosely integrated systems, application granularity is particularly important. Significant speedups and super -computer like performance are only achieved with systems where the amount of reconfigurable hardware available for hardware function implementation is relatively large compared to the required communication overhead of the functions of the particular application being realized in hardware. In order to quantify this relative measure one must consider the following equation. It must be satisfied for any speedup to occur:

$$T_H + T_{OV} < T_S$$

T_H = time to execute function in hardware
T_{OV}= time to communicate data and control overhead
T_S = time to execute function in software (or main funct. unit)

This equation may be rewritten as follows:

$$\frac{T_H}{T_S} + \frac{T_{OV}}{T_S} < 1$$

The first fraction represents the actual hardware computational speedup, the second fraction is indicative of the granularity of a given application on a given CCM. A small T_{ov}/T_s ratio (< 0.1) indicates a larger grain size, while a larger T_{ov}/T_s ratio (> 0.5) represents a smaller grain size. The actual ratio depends on the characteristics of a given application running on a given CCM system. The sum of both T_h/T_s and T_{ov}/T_s is indicative of the overall speedup - the smaller this sum, the larger the overall speedup of a particular application on a particular CCM system (for sums less than one).

Clearly, CCM systems with a small T_{ov}/T_s ratio require only a smaller hardware speedup (T_h/T_s) to achieve the same overall speedup $(T_h + T_{ov}) / T_s$ than CCM systems whose T_{ov}/T_s is larger, similar to Amdahl's law [1]. In closely coupled CCMs, where the communication overhead is lower, the grain size of the functions implemented in hardware can be reduced. To enhance the performance of as wide a variety of applications as possible, the minimum required grain size for application speedup must be allowed to be relatively small by keeping the pertinent communication overhead associated with the CCM architecture at a minimum.

The architecture and structure of the reconfigurable logic required for CCMs has not been studied in detail. Most CCMs described above utilize general purpose FPGA structures to build their flexible logic resources. Only the recently proposed tightly integrated CCM systems stray from this path. Issues like configuration context switching for multitasking and time-sharing applications [15][16][17] and reconfigurable structures for special purpose applications ranging from very fine grain [19] to datapath oriented [20] designs have to be considered.

The explicit use of reconfigurable logic in embedded controller type applications has not yet been explored by present CCM systems. Issues involved in the attachment of customizable preprocessing glue logic to a microprocessor have yet to be identified and addressed.

2.2 CCM Programming Issues

Without sophisticated software to make use of the customizable features available through the added reconfigurable logic, the CCM system cannot easily achieve performance gains over a stand-alone microprocessor system. An experienced hardware and software designer might be able to generate the hardware images of coprocessor functions, program the reconfigurable resources with them and rewrite the application software to properly make use of the added hardware. However, the common user will hardly be able to accomplish such a challenging task, nor will anyone want to spend the time to hand-code the CCM system for every new application. To be useful, the CCM system must be programmable like any other computer and must handle the reconfigurable processing engine in a user transparent fashion. The following will describe an ideal system, an existent system and the challenges facing the ideal CCM software environment.

The general-purpose, user-friendly reconfigurable processing system includes a mature software package, shown in Figure 2-4, which automatically uses the reconfigurable resources to execute a set of given application programs under the most efficient system configuration. The user would code application software just like for any standard microprocessor system. Any high level language should be appropriate.

Following this initial step, the CCM preprocessor analyzes the code and, using profile information, identifies possible candidate procedures, functions or complex instructions for hardware implementation. Functions identified in this step are of various granularities, not necessarily orthogonal, and taken in one set, exceed the logic amount available in the

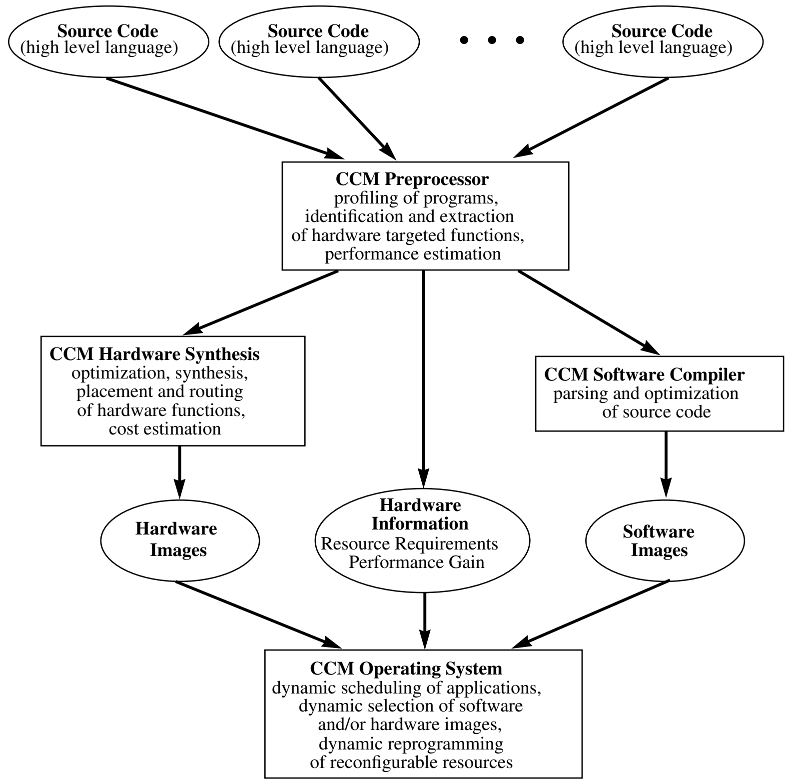


Fig. 2-4: The Ideal CCM Software Environment

reconfigurable resources. Once found to be providing a significant performance improvement at a calculated hardware cost, the functions are parsed, optimized, synthesized, pre-placed and pre-routed, readied for hardware implementation. The software side is prepared in a similar manner . The optimized hardware and software images, not yet scheduled, are passed on to the CCM operating system (OS) along with information of which software blocks are replaceable by which hardware images.

The OS then takes all this information of several programs to be executed and schedules them in a way that allows for the best overall performance of all pending applications. Such a scheduler must use the performance gain and hardware cost information of all the candidate functions of all programs and minimize the required dynamic reprogramming of the reconfigurable logic resources while maximizing the performance of individual tasks. Possible large performance gains for one particular application, achieved by using a large grain hardware image, might have to be traded for somewhat smaller gains by using a reduced grain size hardware image to more fairly share the reconfigurable resources across the numerous applications. Hardware images are thus precompiled and can be dynamically swapped in and out like in a virtual memory handler .

Overall, users can run their standard high level language applications on an automatically configuring, dynamically adjusting, multitasking CCM system just like on a standard, fixed microprocessor , while benefiting from a system-specific optimal use of the performance enhancing hardware execution of selected functions handled in total transparency by the compiler and the OS.

Existent CCM software environments, in contrast to the ideal one just described, achieve a more limited degree of automation. As an example, the PRISM configuration compiler [4] is described here. Its function identification and extraction process, which operates on the source program that is coded in C, has been automated to the point where the user is presented a list of synthesizeable function candidates and must select the most desired ones. Hardware image synthesis is fully automated. This process checks the high level language syntax, parses, optimizes, partitions and synthesizes the functions identified in the earlier step into their usable hardware images. Only certain limited language constructs are recognized by the compiler . Other CCM software environments also utilize standardized object libraries of frequently used hardware constructs to aid the hardware image synthesis process [8]. Optimization and merging of the software and hardware images, again, is fully automated. The PRISM system is limited to single program execution and does not allow multitasking, very large grain, sequential nor very small grain, sub-procedural, bit-level hardware targeted functions.

Even though automated CCM software environments have been built, some very challenging tasks need to be overcome to construct the ideal compiler and OS described above. Undoubtedly , adding multitasking capabilities to CCM systems will be one of the biggest challenges. This problem will require the clever handling of hardware image context switches along with the proper handling of any computational states associated with a given image. Furthermore, just like in virtual memory systems, fragmentation and thrashing must be considered and can be expected to represent areas where performance gains can be lost. The grain size question for hardware targeted function selection will depend on the reconfigurable logic architecture and on the maximum degree of multitasking achievable before thrashing and fragmentation eat up CCM performance gains. As the variety in grain sizes amongst the existent CCM systems suggests, the problem will not be in building a perfect grain size compute engine, but rather in selecting the perfect granularity .

This short discussion of some of the issues involved in the design of the system software is included only to inform the reader that the design of the underlying hardware is only a small part of the work involved in building a complex, general CCM system. The work presented here is only concerned with hardware issues.

2.3 Processor Architecture Issues

When attaching reconfigurable logic to a fixed microprocessor for application speedup purposes, performance gain over a comparable CPU without the flexible logic is the main goal. It has been pointed out by Jeschke [2] and others [4][15] that the achievable speedup from a CCM system can be severely limited by the fixed - reconfigurable logic interface. However , besides suggesting that the fixed CPU of a CCM should be as fast as a state of the art microprocessor such as the 300 MHz DEC 21 164 Alpha [21], no existent work has studied the choice of fixed CPU architectures in detail. This section will elaborate the trade-offs involved in selecting a suitable microprocessor for the high performance, tightly coupled OneChip system with integrated CPU.

The Complex Instruction Set Computer (CISC) is a general purpose microprocessor that provides a rich set of operations immediately usable through its instruction set. Several addressing modes are supported, the instruction format is flexible to allow for the requirements of the numerous operations, and even some complex functions are provided.

A Reduced Instruction Set Computer (RISC) differs from the CISC processor in that it usually limits the available addressing modes, provides only a fixed instruction format and leaves the programmer with the task of implementing complex instructions out of the few provided basic ones. The main idea for limiting the hardware resource is to optimize and pipeline the datapath to significantly improve the throughput and thus the overall performance of the CPU.

Both CISC and RISC CPUs typically represent general purpose platforms with a relatively complete set of instructions available for the execution of general purpose applications. CISC processors usually provide more instructions, however , are also less structured than a highly pipelined RISC CPU.

The requirements of the fixed microprocessor of the OneChip CCM system include speed and interfacing flexibility . The processor must be fast at executing instructions from the basic, fixed logic instruction set and at providing support for the instructions custom built in the reconfigurable logic. Some very basic instructions that always occur to a significant degree in a wide range of applications must be implemented in fixed logic, because a per application basis hardware compilation of instructions into reconfigurable logic carries a speed penalty of a factor of approximately three [22]. Also, the support and interfacing logic provided by the fixed processor must not be slower than the computational delay of the instructions implemented in reconfigurable hardware. Furthermore, the fixed microprocessor must be able to easily integrate and to address instructions implemented on the reconfigurable logic of the CCM system. This interface must be general enough to handle a broad range of control and data flow signals.

From the above discussion of the requirements it becomes apparent that OneChip's fixed CPU should be a very fast RISC CPU with a minimal basic instruction set and a flexible interface to tightly integrate the reconfigurable logic into the system. Complex instruction sets are not required for a CCM system as instructions can be optimized on a per application basis. The design of the instruction set of a CCM is addressed by the configuration software every time the reconfigurable part is reprogrammed. The pipelined structure of RISC CPUs makes them very regular, organized processor designs with many ideal access points to the main datapath to integrate reconfigurable logic resources.

2.4 Summary

This chapter reviews several existent custom compute machine (CCM) systems and classifies them according to the type of interfacing found in-between the fixed and the reconfigurable logic. The need for a closely integrated system is stressed. Also, issues involved in the design of an ideal, fully automated compiler and operating system (OS) for a CCM system are introduced. Their complexity is noted and some challenges limiting existing system software from being more ideal are identified. Finally, the question of the type of the fixed microprocessor for the CCM system is addressed through analyses of the CISC versus RISC computing paradigms and the problems of an instruction set design. It is recommended to use a very fast, easily interfaceable RISC CPU with a minimal fixed instruction set.

Chapter 3

Tool Issues

Rapid prototyping of both the architecture of reconfigurable compute engines as well as their applications is required during the early design stages of CCM system development. The trade-offs of various architectures must be studied in simulations, evaluated and refined before the more successful designs can be identified. For prototyping purposes it is important to have a stable tool suite that does not delay or distract the engineer from the actual task of designing the CCM system. This chapter addresses some of the problems a designer faces when dealing with current generation CAD tools for logic synthesis.

The first section addresses the selection of a high level language logic synthesis tool. It was noted from early work undertaken on the Transmogrifier-1 (TM-1) field-programmable system [23][24] that dealing with complex schematic based designs can be time consuming. A more structured, high level hardware description language (HDL) allows for easier changes to designs exhibiting a highly repetitive, parallel nature found in modern processors with wide datapaths. However, a suitably fast and efficient synthesis tool producing good quality (small area, minimal combinational delay) FPGA targeted logic must be used. The first section of this chapter contains a comparative study that evaluates the relative strengths of available HDL logic synthesis tools.

After the CAD tool of choice is determined, it is noted that the state of today's synthesis tool development is not very advanced and that the efficient use of the tools is limited by some of their shortcomings. The second section of this chapter identifies problems encountered while using the HDL synthesis tool of choice along with other CAD tools needed to implement designs on the TM-1 field-programmable system and presents work around solutions as well as actual fixes in the form of custom-made utility applications.

Finally, the last section of this chapter deals with the special design approach required when using the VHDL hardware description language [25] to target FPGA-like logic devices. Language constructs suitable for synthesis are identified and a design methodology for dealing with the individual logic structures found on FPGAs is described.

3.1 Synthesis Tool Selection

Implementing a given logic design can be done in many different ways. Possible mechanisms include schematic entry, high-level language synthesis and waveform dependent logic generation. The first and the last of these design methods are somewhat less suitable than the second when fast and efficient logic generation is required. Schematic entry can be time consuming for bigger designs and can require extensive work when regular, repetitive structures must be rewired. Waveform dependent logic generation is faster since it is fully automated, but relies on the required CAD tool for design optimization, which, in general, produces less efficient designs than manual design entry can.

The compromise amongst the just described methods of specifying and generating a design comes in the form of the high-level language synthesis tool. Being fully automated, it produces designs or changes to designs faster than the manual procedure can. At the same time, language synthesis is fairly efficient in the quality (area, speed) of the logic generated since the hardware oriented features of such hardware description languages (HDL) like VHDL can be exploited. Other high-level languages such as Pascal or C are less suitable candidates for HDLs, because they are designed for other purposes than hardware description and must be severely restricted for synthesis purposes, as pointed out in [26].

3.1.1 Study Outline

Even after deciding to use an HDL based logic synthesis tool for the purpose of specifying and generating logic designs, there are more choices to be made in selecting amongst the most suited of these CAD tools. The presented study is limited to VHDL synthesizing tools, because this hardware description language is widely accepted and commonly used in the industry. The investigation focuses on the following tool characteristics:

- synthesis time
- synthesis quality (logic area, combinational delay)
- integration with FPGA technology specific tools

Furthermore, the availability of a tool for use at the University of Toronto was considered. Initially, the three tools to be studied were Exemplar's *Core* [27], Synopsys' *Design Analyzer* [28] and Viewlogic's *ViewSynthesis* [29], the major VHDL logic synthesis tools available. However, after some initial investigation, the Synopsys tools were dropped from the study, because the required libraries for Xilinx FPGA specific technology mapping (required when targeting the TM-1 field-programmable system) were not available.

Obtaining, installing and configuring the missing library was not deemed to be feasible within the time frame of this work.

3.1.2 Benchmark Circuit Characteristics

The selection of the circuits used to evaluate the synthesis tools included in this study was made to cover a range of designs with varying characteristics. Initially, somewhat smaller circuits, not necessarily representing any real-world designs, were crafted to experiment with the synthesis tools and to obtain an understanding for their strengths and weaknesses. However, for the actual study, the following circuits were selected as real-world tool benchmarks:

32-bit ALU

This circuit represents the arithmetic logic unit (ALU) of the CPU implemented as part of this thesis. It is a purely combinational circuit that takes two 32-bit operands, computes their logical AND, OR, sum or difference and returns the result on another 32-bit wide bus. The actual VHDL code does not specify any structural way of implementing the logical and arithmetic operations. It leaves the optimization of these functions up to the synthesis tool.

16-bit multiplier

The second of the benchmark circuits, a Wallace tree multiplier was gratefully obtained from Qiang Wang [30], a PhD candidate in the Dept. of Electrical and Computer Engineering at the University of Toronto. It is another purely combinational circuit, significantly bigger than the first one, with a structure fully described as part of the VHDL code. Such building blocks as half adders and compressors are coded behaviorally and their interconnection to form the 16 x 16 multiplier is given structurally. Minimal optimization is possible on behalf of the synthesis tool.

CPU pipeline stage

The first of the benchmark circuits to be sequential in nature is a pipeline register that extensively uses storage elements. Only few combinational gates are employed. It includes one set of flip flops that have an enable and whose 32-bit wide input is muxed from one of two possible sources. Another set of 32 flip flops, always enabled and not preceded by a mux, is also part of this pipe stage. The VHDL code is written with standard constructs such that the mux and the flip flops are easily recognized by the synthesis tools.

Memory Controller

Another sequential circuit, the memory controller module, is taken from the design of OneChip' s core processor . It contains several simple state machines. The VHDL code of this application represents the interface that handles the generation of the control signals required by the CPU or the front end host to access the memory found on the TM-1 field-programmable system. The state machines are relatively small (few states, minimal combinational next-state logic), while the state information decoding, and the memory data qualifying combinational and latching logic account for most of the gates used. Again, the VHDL code is written using standard constructs easily recognized by the synthesis tools.

UART

The last of the tool study benchmarks is a universal asynchronous receiver and transmitter (UART) with five-level-deep send and receive buffers. This circuit contains more complex state machines than the previous one. State decoding logic is minimal as individual states correspond to unique events. A significant amount of logic is used for the muxing and the storage of parallel data in the buffering FIFOs. As before, the VHDL code constructs are easily recognized by the synthesis tools. The overall complexity of the UART logic is two to three times bigger than the complexity of the memory controller hardware.

3.1.3 Study Results

Findings of the comparative synthesis tool study are presented in this section. The results of the analysis of the tool characteristics described in Section 3.1.1 for the synthesis of the benchmark circuits presented in the previous section can be found in Table 3-1. The VHDL source code needed to be slightly adjusted for each tool to comply with the vendor specific VHDL syntax. However, this minor syntactic difference does not change the general coding style and should therefore not bias the results. While ViewSynthesis only allows the choice between optimizing area or delay, Exemplar's Core tool also provides adjustable optimization effort settings. The two choices “safe” and “quick” were tried amongst the available settings. In the “safe” mode several runs trying several optimization strategies are tried. The “quick” mode performs only one optimization run.

The results summarized in Table 3-1 allow for the quantifying of trends already observed as notions during initial experimentation with these logic synthesis tools:

• synthesis cpu time (min:sec) • CLB function gens. • CLB flip flops • max. CLB delay levels	Exemplar Core V2.1.9 optimizing area (safe)	Exemplar Core V2.1.9 optimizing area (quick)	Exemplar Core V2.1.9 optimizing speed (safe)	Exemplar Core V2.1.9 optimizing speed (quick)	Viewlogic ViewSynthesis V2.4 optimizing area	Viewlogic ViewSynthesis V2.4 optimizing speed
ALU	21:44 267 0 39	2:13 567 0 69	24:19 298 0 36	3:52 586 0 64	0:42 200 0 35	0:43 200 0 35
MULTIPLIER	235:09 662 0 32	66:02 990 0 35	245:20 726 0 24	70:04 990 0 32	2:17 723 0 33	2:16 723 0 33
PIPE STAGE	0:44 32 64 1	0:05 32 64 1	0:45 32 64 1	0:04 32 64 1	0:06 32 64 1	0:05 32 64 1
MEMORY-CONTROLLER	2:48 70 74 2	0:14 70 74 2	2:47 70 74 2	0:14 70 74 2	0:17 73 69 4	0:18 73 69 4
UART	17:34 204 180 5	1:58 204 180 5	13:09 393 196 5	1:17 297 180 6	8:42 288 188 7	8:49 288 188 7
all synthesis tool tests were performed on equivalent Sparc5 systems						

Table 3-1: Comparison of Synthesis Tools.

- for purely combinational circuits Viewlogic's tool outperforms the one from Exemplar in synthesis time and logic density, while matching it on the combinational delay.
- for sequential circuits Viewlogic's tool takes somewhat longer to synthesize equally dense and comparably fast logic as Exemplar's tool does.
- for flip-flop intensive sequential circuits neither tool fares better than the other.

These trends point at choosing Viewlogic's synthesis tool over Exemplar's for the significantly faster synthesis of combinational circuits while accepting minimally worse performance for sequential circuits. The quality of the synthesized logic seems to be similar with both tools, and it must be remembered that Xilinx's *XACT 5.0* software [31], required for the final stages of the design implementation, also performs further logic optimizations. Another significant factor already outlined in Section 3.1.1, yet not listed in Table 3-1 is the integration of the synthesis tool with FPGA specific software. In particular, the ability to take timing information from the output of the FPGA place and route tool and to feed this information into a simulator integrated into the synthesis tool suite is a very important option. Without this capability only a highly unrealistic unit delay simulation of the synthesized, but not yet placed and routed logic design is possible. The Viewlogic tool suite has the option of performing back-annotated timing simulations, the Exemplar package does not.

As a note, it is interesting to observe that all the benchmark VHDL code is written using only certain constructs easily recognized by the synthesis tool. Even Exemplar's tool operating in the "safe" optimization effort mode, where several optimization strategies are tried in numerous runs, can execute for hours and still produce slow and complex circuits when not fed standard, recognized code constructs. It thus suffices to use the one strategy, one optimization run Viewlogic tool or the Exemplar software in "quick", one run mode, along with VHDL code written with synthesis in mind to quickly obtain qualitative results. Section 3.3 deals with VHDL code structured for synthesis as well as with a design methodology for targeting FPGAs with a hardware description language.

Amongst the three logic synthesis tools available to the author, only two are capable of synthesizing into Xilinx FPGA specific logic. Between the remaining two, the Viewlogic synthesis tool is selected for its significantly better performance on combinational circuits and optional capability to perform realistic timing simulations when integrated with the Viewlogic tool suite and the Xilinx specific FPGA place and route software.

3.2 Tool Problems and Solutions

Ideally, an automated design flow starting with the VHDL synthesis tool at the front end ultimately targeting the FPGAs of the TM-1 field-programmable system at the back is envisioned. As part of this thesis, a script called *MakeMe* was developed to allow for the automation of the TM-1 design implementation. The complete design flow is illustrated in Figure 3-1.

The script takes some required information from an initialization file such as the names of the VHDL modules that make up the design, whether or not the preplaced and prerouted, parametrized *xblox* library modules [32] or the TM-1 external memories are used before it executes the CAD tools along the steps of the design flow (i.e. synthesis, global partitioning, local partitioning and placement and FPS programming data generation). However, several problems obstructing the integration and automation of the design implementation process were encountered. The problems were fixed either by working around them or by coding

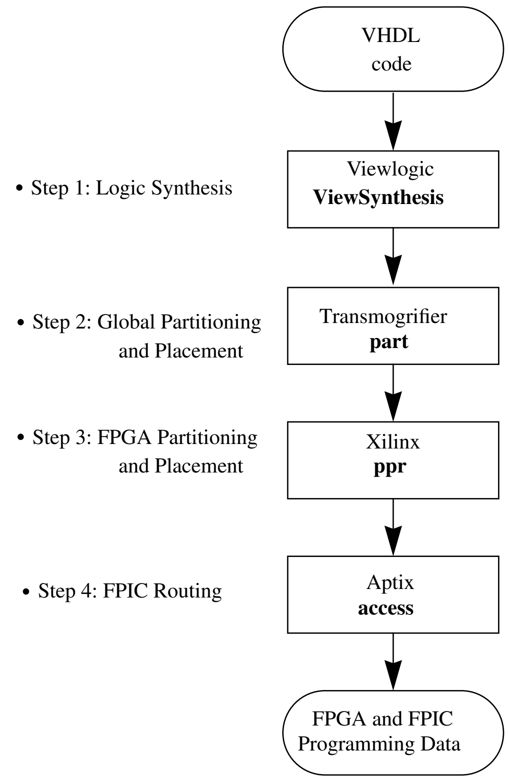


Fig. 3-1: MakeMe VHDL to TM-1 Design Flow

custom-made helper utilities.

3.2.1 ViewSynthesis Problems

Even the synthesis tool itself was not ready for integration into the automated design flow handled by the MakeMe script. Two major problems were encountered whose existence was not documented by Viewlogic. They required an extensive investigation to localize followed by the coding of custom-made helper utilities to solve them:

Bus Order Reversal

In an early version of the synthesis tool, the ordering of bus labels on sub-module symbols was reversed in the Viewlogic wir netlist file. Due to this reversal, the proper interconnection of modules and logic was not being made. An early work-around solution to this problem was to flatten the whole design into one VHDL module from the start to completely avoid the generation of a hierarchical netlist with sub-modules. However, as Table 3-2 illustrates, the long synthesis time for a flattened design is unacceptable.

To avoid time consuming hand fixes to the hierarchical netlists and to automate the synthesis design flow for appropriate integration with the MakeMe script, a utility program was coded. The program *fixBus* detects the case where the labels of buses on sub-module symbols are in a different order than buses attached to these particular sub-module pins. The appropriate bus reordering is performed in this case. The complexity of handling all special cases made it too difficult to implement fixBus as a shell script. For this reason the utility was coded in C. The program is fast enough to execute on the complete hierarchical description of the 32-bit CPU in less than one second.

Coding Style	Synthesis Time
Flat	~ 36 hours
Hierarchical	12 minutes
all tests run on zeep.eecg. FPGA resource utilization: 1157 CLB FGs., 889 CLB FFs	

Table 3-2: Hierarchical vs. Flat Synthesis of 32-bit CPU

It is interesting to note that continued calls to Viewlogic about this particular bus reversal problem resulted in a subsequent fix of the problem in their synthesis tool. Furthermore, Viewlogic's customer support offered to the University of Toronto increased noticeably in the form of frequent tool updates. Nonetheless, the fixBus utility was extensively used for a period of four months before the bus reordering problem with hierarchical netlists was solved in the actual synthesis tool.

Buffer and Pad Insertion

When targeting FPGA logic, the synthesis tool needs to insert the appropriate combination of input or output buffers and pads for the top-level I/O pins of a given design. Figure 3-2 illustrates the problem. A design whose top-level ports are only attached to pins and not to pads is said to be unattached or floating. The Xilinx specific CAD tools eliminate logic not attached to pads. The early version of ViewSynthesis did not have an option to insert Xilinx specific pads and buffers.

To avoid having Xilinx's place and route software optimizing away the synthesized yet unattached logic, the designer can apply two work-around solutions. It is possible to

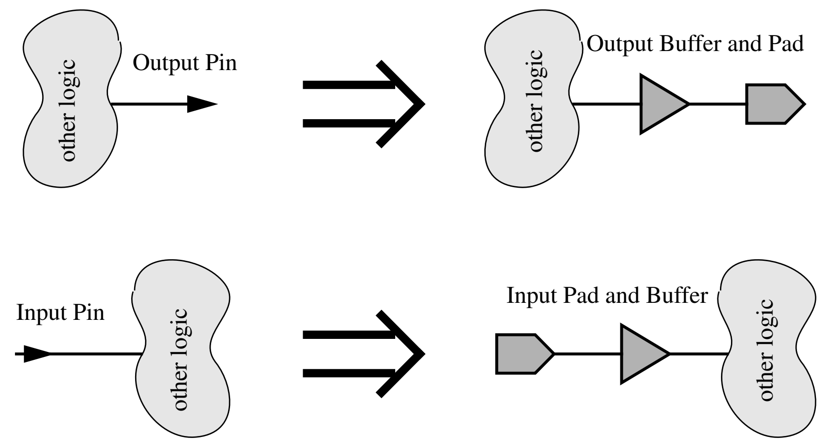


Fig. 3-2: Pin Replacement With Buffer And Pad

generate schematics from the synthesized design's netlists in which the top-level pins can be manually replaced by buffers and pads. Or the designer can go through the netlist files and again manually replace all top-level pin instantiations with the proper code for buffers and pads. Neither procedure can be recommended as both manual methods are time consuming and error prone.

The solution to the buffer and pad insertion problem is the custom made utility program called *placeBufs*. It automates the pin with buffer and pad replacement procedure. placeBufs is started after ViewSynthesis has finished with the generation of a top-level netlist file. In two passes over this given netlist, it gathers the relevant pin attachment information of the complete hierarchy and performs the required pin to buffer and pad replacements on the netlist. Typically, placeBufs does not noticeably add to the overall synthesis time requirements. ViewSynthesis requires twelve minutes to generate the netlist files of a 32-bit CPU (1 157 CLB function generators, 889 CLB flip flops), placeBufs takes six seconds to modify it. Presently, placeBufs is Xilinx XC4000 technology specific, but can easily be extended to be more general.

To make the best use of the functionality provided inside the input/output blocks (IOB) of Xilinx's XC4000 FPGA family, placeBufs also performs some logic optimization. IOBs provide extra flips flops and tri-state gates to handle the interfacing between the core logic and the I / O pads of the FPGA, which are already buffered. Designs whose top-level I/O pins directly attach to core flip flops might find these migrated into IOB flip flops. Core logic tri-state buffers, whose outputs are top-level pins, will also be migrated into their IOB

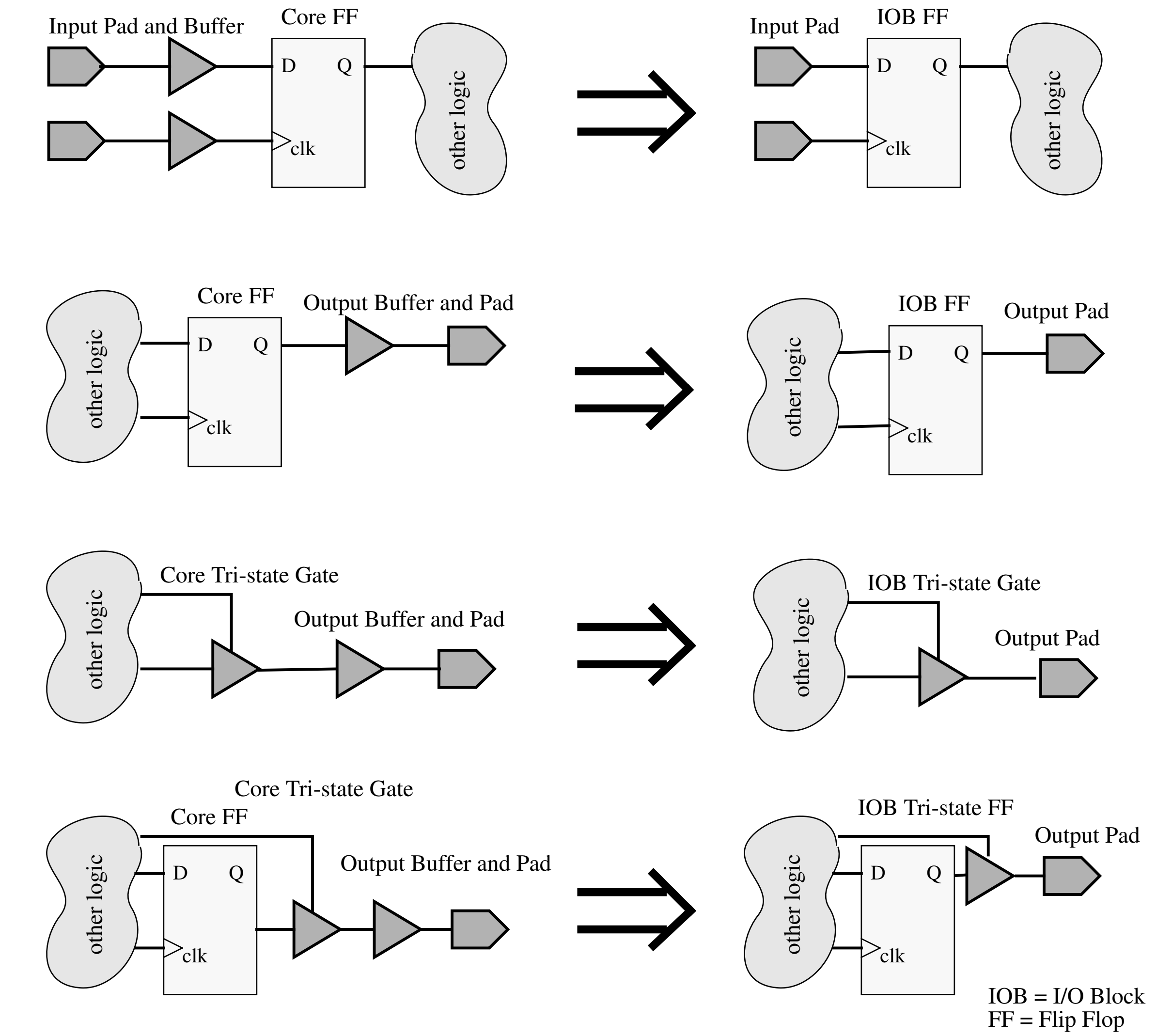


Fig. 3-3: placeBufs XC4000 Specific Logic Optimization

equivalent logic. Even such constructs as a core logic flip flop, followed by a core logic tri-state gate attached to a top-level pin are recognized and replaced by a single IOB tri-state flip flop. See Figure 3-3 for an illustration. Through a better utilization of FPGA resources, placeBufs' logic optimization makes designs more routeable.

It is interesting to note that just when the placeBufs utility had been coded and fully tested, Viewlogic offered an update to their synthesis tool. With the improved version 2.4 of ViewSynthesis the option to perform Xilinx specific pad and buffer insertion was added [29]. However, hierarchical netlists are still not supported. This makes the use of ViewSynthesis for designs requiring modularization for synthesis time reasons (see Table 3-2) unacceptable. placeBufs properly handles the parsing of hierarchical netlists. Another problem causes the synthesis tool to terminate right after pad insertion. It is thus not possible to integrate ViewSynthesis into the MakeMe automated VHDL to TM-1 design flow script when the pad and buffer placement option is used. Subsequent calls to Viewlogic have confirmed these noted limitations. Another tool update was promised for June 1995, but has not yet arrived.

3.2.2 TM-1 Specific Problems

Besides the problems with the synthesis tool just described above, the integration and automation of the Transmogrifier-1 (TM-1) design flow was also hindered by some TM-1 specific CAD tool problems. First two minor problems and their solutions are discussed, followed by the identification of two more significant design flow automation limiting factors and suggestions for working around them.

Aptix Netlist Generation

For the fourth step of the VHDL to TM-1 design flow (see Figure 3-1), the field-programmable interconnect (FPIC) routing, several netlists files are required. The FPIC routing software needs these files to know which pins on the TM-1 board are to be connected with others. The location and net-name of every pin of every component on the board are identified in this way. Utility programs to generate the netlist files for the four FPGAs and fixed components other than the host-interface / debugging connectors and the memories already exist [33]. The generation of the netlists for the connectors and the memories had been left to be done by hand.

To automate these last two tasks, the utilities *labelMem* and *labelCons* have been designed. They both take the name of the top-level module of a given design as their argument, which they use to read in top-level pin information from a file generated by placeBufs. labelMem sets up the FPIC routing netlist for the four 32k x 8-bit TM-1 SRAMs connected in parallel, to be used as one 32k x 32-bit wide memory. When the use of this memory is required, the designer must simply use standard names for the memory pins in the VHDL code and set the MakeMe initialization file memory usage flag to true. The TM-1 CAD tools, including labelMem, automatically generate the necessary TM-1 programming information. Similarly, labelCons sets up the FPIC routing netlist file for the interfacing and debugging connectors of the TM-1 system. It takes the names of the remaining top-level pins found in the placeBufs generated pin-name file and generates the FPIC connectors netlist.

While labelMem and labelCons are the last two utilities required to fully automate the basic VHDL to TM-1 design flow using the MakeMe shell script, some unsolved problems still exist. The Transmogrifier-1 specific global partitioning tool *part* developed by David Galloway [33] does not yet partition across bi-directional buses driven by tri-state gates and the Xilinx to Viewlogic back-annotation interface does not allow for multi-chip timing simulation.

Global Partitioning

Designs that use tri-state driven bi-directional buses and exceed the logic count available in a single TM-1 XC4010 FPGA, must be partitioned by hand at the Xilinx netlist file level. Alternatively, an intelligent split of the VHDL code into several modules that can be synthesized independently is possible. This later approach does not interrupt the design flow and can thus use the convenient MakeMe script. An initial probe into changing the partitioner to handle the special bus constructs showed that a significant amount of time will be required.

Multi-Chip Timing Simulation

The other presently unsolved Transmogrifier-1 specific problem arises from the Xilinx to Viewlogic tool interface. The generation of netlists for simulations on a unit delay per gate basis are easily integrated into the automated design flow. Upon the detection of the user flag specifying unit delay simulation, the MakeMe script uses a Viewlogic tool to generate a simulation netlist from the synthesized, but not yet globally partitioned design netlist. However, when a timing simulation is required, the partitioned designed must first be placed and routed in each of the required FPGAs to obtain the required timing information. This timing information can be back-annotated onto the Viewlogic netlist from where the simulation netlist can be prepared.

The actual problem stems from the fact that the Xilinx to Viewlogic interfacing tools, which allow for the timing information to be passed on to the Viewlogic netlist, operate on a per FPGA basis only. A multi-chip back-annotation procedure is not available. The work-around solution is to generate a new Viewlogic design netlist with the schematics editor, using properly interconnected, back-annotated symbols for each of the four TM-1 FPGAs with timing values from the Xilinx tools. However, the task of manual back-annotation and manual schematic editor net-stitching is very time consuming. The possibility of crafting a utility that automatically generates the new back-annotated Viewlogic design netlist from

FPIC routing information has been fully explored, but could not be further pursued due to time limitations.

3.3 Design Methodology

The use of VHDL for logic synthesis has several advantages over other design implementation methods, especially when reconfigurable logic as found in FPGAs is targeted, because designs can be rapidly prototyped. However, as noted in Section 3.1.3, the semantics of the VHDL code targeted for synthesis can have a great effect on the synthesis time. The fact that certain code constructs are more easily recognized than others also suggests that some non-standard piece of code might cause the synthesis tool to try a non-optimal implementation algorithm, producing synthesized logic of lower quality. Furthermore, targeting the predefined logic structures of FPGAs, which have fixed properties and allow very little freedom for RTL level logic optimizations, requires a special design approach when writing the high level VHDL code.

This section follows two main themes. First, the logic design approach that must be used when targeting FPGAs from a high level hardware description language like VHDL is formalized. General, basic logic structures feasible for implementation on FPGAs are introduced and their use for building more complex systems is stressed. Having presented the reader with this necessary FPGA design methodology, the second part of this section introduces VHDL code constructs that synthesize into the generalized logic structures allowing for the specification of logic designs using a high level language.

3.3.1 FPGA Logic Design Strategies

The Xilinx XC4010 FPGAs found on the Transmogrifier-1 field-programmable system contain flip flops, function generators (FG) capable of modelling many discrete gates and interconnect resources. Using these building blocks, various types of circuits can be constructed. However, the very nature of these predefined structures also imposes certain restrictions. The following discussion of the design of combinational, asynchronous and sequential circuits motivates the formulation of the required FPGA logic design strategy.

FPGAs are well suited for the implementation of combinational circuits. The vast abundance of function generators on the XC4010s allows for the implementation of complex functions with up to nine inputs and four outputs, totalling close to 10,000 equivalent logic gates [11]. As with the design of any combinational circuit, the issue requiring careful consideration is the overall sequential delay. The output of a combinational circuit might change while the value of its intermediate nodes is computed and must not be read or used before the final value has stabilized. Also, it must be remembered that the combinational delays of several integrated sub-circuits should be kept balanced, because the longest delay path determines the achievable clocking frequency .

Having discussed the basic properties of combinational circuits, their use as building blocks for more complex FPGA designs is discussed next. Typical large-scale logic designs often contain finite state machines and other feedback circuits. One can distinguish amongst asynchronous and synchronous feedback systems.

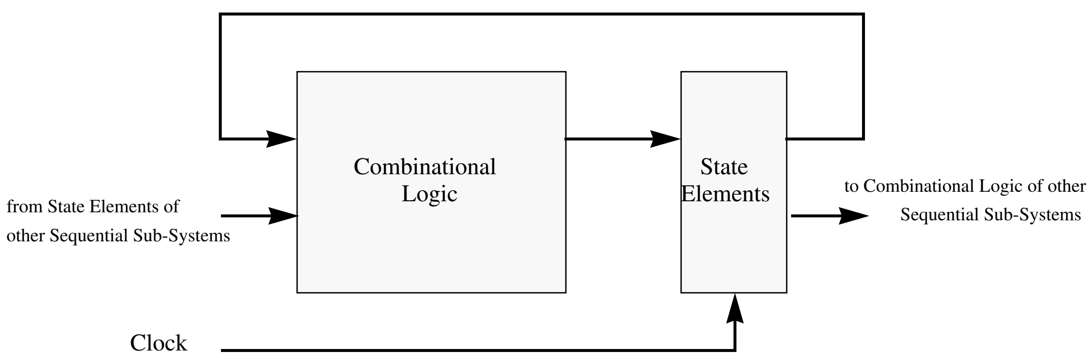


Fig. 3-4: Sequential Sub-System

The asynchronous design approach should be avoided when designing with FPGA-like logic structures. Any type of feedback circuit depends on a synchronizing event for the transition from one state to another . In case of the asynchronous circuit, this event must be generated by some form of intrinsic combinational logic and must arrive at a predetermined point in time. However , FPGA designs requiring the exact knowledge of combinational delays are problematic, because it is not possible for a designer to determine the precise delay of any given path in a particular design until it has been placed and routed inside the FPGA. Furthermore, the delay of a given combinational circuit may vary from one FPGA implementation to another , since the place and route tools do not necessarily use the same combination of function generators and routing resources for the implementation of a given function every time they are executed.

To become circuit delay independent, a synchronizer in the form of an FPGA external signal must be used. The transition from one state to another should be triggered by the rising or falling edge of a clock. With this design strategy , the intrinsic delay of the combinational elements can be determined after logic placement and routing is completed.

The period of the clock can then be adjusted to allow for the slowest combinational path to be traversed completely . With the capability to determine the exact triggering point of the synchronizing event after its required timing has been determined, the placement independent design of complex FPGA based logic systems becomes possible.

The format of a general synchronous, sequential circuit is depicted in Figure 3-4. The integration of combinational logic in between state elements, which are controlled by a synchronous event, represents the only proper FPGA logic design strategy . Complex systems are thus constructed from a collection of interconnected state machines whose elementary building block is the combinational module.

3.3.2 VHDL Constructs For FPGA Logic Design

Having defined the logic design approach that must be employed when targeting FPGAs, the corresponding high level language coding methodology can be described. This proper FPGA logic design strategy must also be considered when writing VHDL code: the required coding style interleaves combinational logic blocks amongst state elements, each of which are described using non-overlapping clearly identifiable code constructs. As Section 3.1.3 notes, the use of VHDL constructs readily recognized by the CAD software results in a significantly shorter synthesis time and qualitatively better logic. It is important to be at ease with coding components from these two major groups in VHDL to be able to trace and debug the results the synthesis tool produces. A description of the Viewlogic VHDL code for the most frequently encountered gates, broken into combinational and memory components follows. It should be noted that Viewlogic VHDL code slightly differs from the standard IEEE 1076 and 1164 packages.

Figure 3-5 gives the code and the equivalent logic symbol for such combinational gates as AND, OR, NOT as well as the slightly more complex 4-1 multiplexer and tri-state buffer . The code uses buses that are four bits wide for all data signals (*a, b, c, d, in, out*), a two-bit wide bus for the select input (*sel*) of the 4-1 mux and a single-bit line for the enable (*en*) input to the tri-state gate. These signal widths can be arbitrarily adjusted and are just meant as a general example.

It is important to observe that ViewSynthesis requires all combinational code constructs to be placed outside of process statements. The use of the process keyword automatically causes ViewSynthesis to introduce a state element in the form of a flip flop, even though this

```
SIGNAL a, b, c, d, in, out : VLBIT_1D(3 DOWNTO 0)
SIGNAL sel : VLBIT_1D(1 DOWNTO 0);
SIGNAL en : VLBIT   ;

out<= a AND b;

out <= a OR b;

out <= NOT in;

out <= in WHEN en='1'
         ELSE "ZZ";

WITH sel SELECT
    out <= a WHEN "00"
      ELSE b WHEN "01"
      ELSE c WHEN "10"
      ELSE d;
```

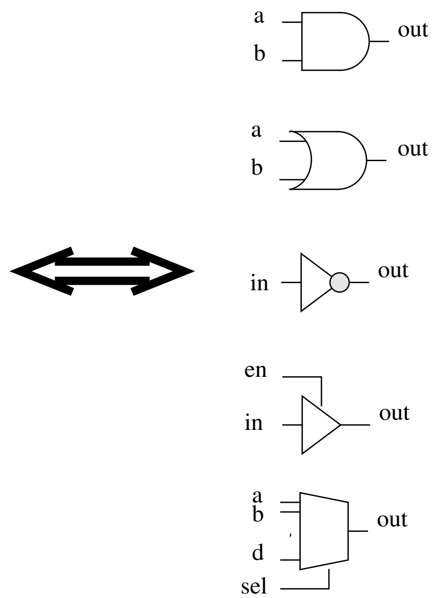


Fig. 3-5: Sample VHDL Combinational Logic Constructs

is not needed in combinational constructs. Exemplar's and Synopsys' VHDL synthesis tools are intelligent enough to recognize that certain code constructs inside of process statements, which intrinsically describe an event triggered piece of hardware, do not necessarily require state elements in the form of flip flops.

For the same reason, if-then-else or case constructs, which by the definition of the VHDL language can only be used inside process statements, should not be used to describe combinational logic. Again, ViewSynthesis would force the insertion of an unwanted flip flop. The when statement can be used as an alternative should a selective assignment be required.

As all the memory elements on Xilinx XC4000 FPGAs are edge-triggered, D-type flip flops with enable, asynchronous preset and reset inputs, all state element Viewlogic VHDL code examples presented in Figure 3-6 target only this type of memory device. Level-sensitive latches and other types of flip flops do not exist on the Xilinx XC4000 series FPGA nor in the Viewlogic Xilinx technology library and should therefore be avoided. The bus width of four bits for the data signals (*in*, *out*) is an arbitrary, general choice. Only the width of a single bit for the control signals (*clock*, *enable* and *reset*) is fixed.

```
SIGNAL in, out : VLBIT_1D (3 DOWNTO 0)
SIGNAL clock, enable, reset : VLBIT      ;

 - -

PROCESS
BEGIN
   WAIT UNTIL PRISING (clock);
   out <= in;
END PROCESS;
- -
PROCESS
BEGIN
   WAIT UNTIL PF   ALLING (clock);
   IF (enable='1') THEN
       out <= in;
   END IF;
END PROCESS;
 - -

PROCESS
BEGIN
   WAIT UNTIL PRISING (clock) OR reset='1';
   IF (reset='1') then
       out <= "00";
   ELSE
       IF (enable='1') THEN
           out <= in;
       END IF;
   END IF;
END PROCESS;
```

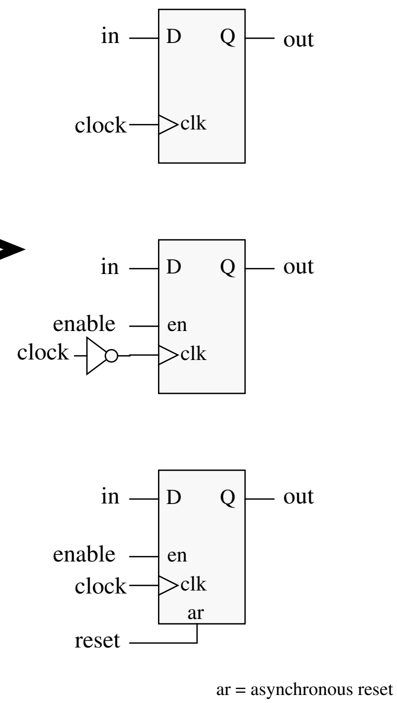


Fig. 3-6: Sample VHDL Flip Flop Constructs

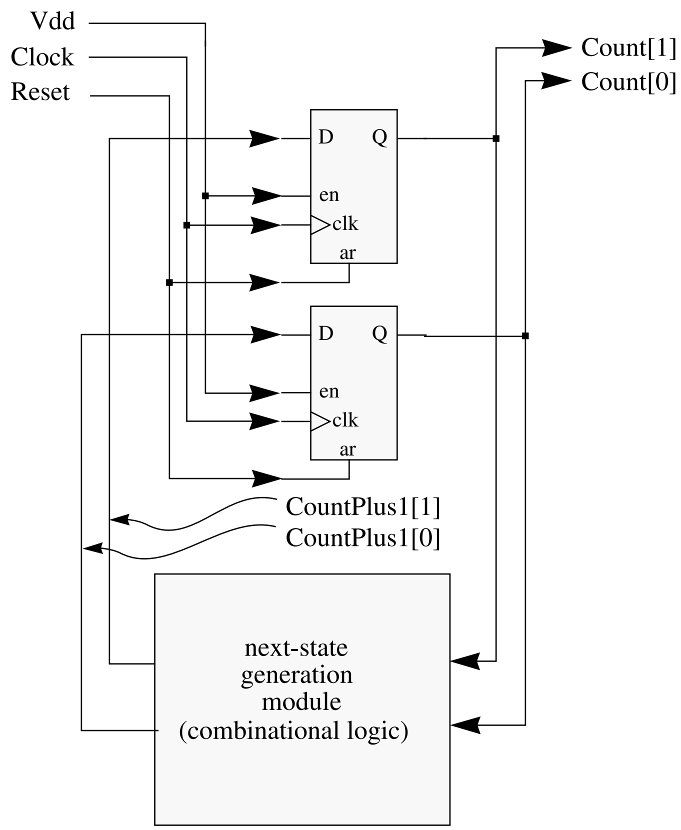
Any slight variation on the code of Figure 3-6 will result in extra combinational logic added in to achieve the required functionality . For example, if the enable line in the flip flop VHDL code is not properly detected, the synthesis tool might generate an extra 2-1 multiplexer in front of the flip flop' s D input. When set up such that the multiplexer selects between feedback from the present flip flop state and the new data input, a flip flop with enable feature is modelled. It should therefore be stressed again that V iewSynthesis is very strict with the syntax of the VHDL code to be synthesized.

VHDL code constructs not at all synthesized by V iewSynthesis include variables and for loops. V ariables are not supported as their valid range, required for determining the width of the equivalent hardware bus, is not known to the synthesis tool. The work-around solution requires a replacement of all variables with signals of the appropriate width. 'For ' loops, which use variables to specify the changing entity , must be unrolled and all variable references must, again, be replaced by their signal equivalent.

This concludes the discussion of typical code constructs needed to produce combinational and memory type logic required for the VHDL to FPGA logic design strategy . As an example

```
----- LIBRARIES
library synth;
use synth.stdsynth.all;

----- ENTITY
entity two_bit_counter is
port ( clk : in vlbit;
        Reset : in vlbit;
        count : out vlbit_1d(2 downto 0);
        );
end two_bit_counter;


architecture behave of two_bit_counter is

signal count_int : vlbit_1d(1 downto 0);
signal countPlus1 : vlbit_1d(1 downto 0);

begin
        present_state : process
        begin
                wait until prising(clk) or
Reset='1';
                if (Reset='1') then
                count_int <= "000";
                else
                count_int <= countPlus1;
                end if;
        end process;

        -- next_state generation logic

        with v1d2int(count(2 downto 0)) select
                countPlus1 <= "01" when 0,
                    "10" when 1,
                    "11" when 2,
                    "00" when 3,
                    "00" when others;
end behave;
```

VHDL Code



Synthesized Hardware

Fig. 3-7: Two-Bit Counter VHDL Design Example

of an application coded using the above code constructs, the Viewlogic VHDL code of a three bit counter with asynchronous reset is presented in Figure 3-7. The synthesized hardware construct is shown for comparison purposes. Note the clear separation of the purely combinational next-state logic and the encapsulated present-state flip flops inside the process statement.

3.4 Summary

This chapter deals with the CAD tools needed to synthesize a VHDL circuit description and to implement the generated logic design on the TM-1 field-programmable system. Issues affecting the selection of a VHDL synthesis tool are discussed and with the results of a study on tool characteristics, the Viewlogic tool suite is chosen over its competitors. An in-depth analysis of the VHDL to TM-1 design flow revealed several problems with the CAD tools involved. Custom made utility programs along with work-around solutions to these problems are presented. With these fixes, the whole design flow is integrated and automated by a shell script. Finally, an appropriate design methodology for targeting FPGAs from VHDL code is presented. An analysis of some FPGA architectural trademarks leads to the identification of the necessary FPGA logic design strategy. The implications of this strategy on the VHDL coding style are presented using some sample code constructs.

Chapter 4

Processor Implementation

Although the available logic count of FPGAs has been increasing since their conception over ten years ago, the implementation of large designs still requires the use multiple devices. Presently, the largest announced and soon-to-be available FPGA, Altera's Flex 10k part, provides more than 50,000 usable gates [34]. However, these state-of-the-art devices have not yet been incorporated into a multi-chip field programmable system (FPS). The Transmogrifier-1 (TM-1) FPS uses four of the somewhat older Xilinx XC4010 FPGAs with a total usable gate count of approximately 40,000 [11]. At the start of the OneChip project, it was expected that this capacity would suffice for the implementation of a basic CPU and its expansion with reconfigurable resources.

As outlined in chapter two, for the evaluation of reconfigurable to fixed logic interfacing, both the reconfigurable resources and the CPU of the OneChip CCM were implemented using FPGAs. A custom silicon implementation would certainly reap the speed and density benefits of fixing the CPU logic. Only the reconfigurable resources of the compute engine will still use FPGA like structures. However, for this exploratory work, implementing the CPU in reconfigurable logic allows the design space to be more easily investigated. This chapter deals with issues involved in the TM-1 implementation of the OneChip CPU.

First, a general description of the architecture of the processor is presented. The major building blocks of the design are identified and their function is described. The discussion focuses on breaking up the structure of the CPU such that it can easily be described in Viewlogic VHDL targeted for synthesis and subsequent TM-1 implementation, employing the design methodology described in the previous chapter.

The second section deals with TM-1 specific implementation issues typically encountered in big designs requiring the use of an FPS. Employing the OneChip CPU as an example for a big logic design, clocking, partitioning and pin constraint problems are discussed.

To round up the discussion, testing procedures and their results are presented. Multi-chip simulation and real system debugging experiences are described, followed by a brief discussion of the speed limiting factors and actual performance figures of the OneChip CPU.

4.1 Processor Description

The discussion of chapter two on the selection of a suitable CPU for the fixed part of a CCM system recommends the use of a highly structured, easily interfaced RISC processor . An initial investigation into RISC architectures showed that processors have become increasingly complex since the design of the first MIPS CPUs [21][35]. However , for the purpose of studying interfacing issues, only a very basic processor is required. Furthermore, the architecture of the basic MIPS-like RISC CPUs is well documented and can therefore be easily described using VHDL. The processor implemented as part of this work closely resembles the one described by Patterson and Hennessy (P&H) [36]. At this point, the author would like to thank Robert J. Gluss from the University of California in Davis for providing the initial VHDL model of the processor [37]. While being suitable for simulation, the code required extensive rewriting to conform to the V iewlogic VHDL syntax, to be synthesizeable, and to take advantage of such TM-1 specific features as the system' s external memory . The functionality and the architectural features of the processor are described in the following two sections.

4.1.1 Functional Description

The processor described by P&H and implemented as part of this work is a minimal MIPS-like RISC CPU. Its functionality is a subset of other MIPS machines with which it also shares the instruction format. Following the design of P&H, the only deviation from real MIPS CPUs is the branch delay . Functional highlights include:

- 32-bit instruction and data paths

After an initial feasibility study involving FPGA resource utilization estimates, it was determined that the bus width of 32 bits for the main datapath would fit in the logic and routing resources available in the TM-1 field-programmable system. The consideration to scale datapath and major functional blocks to 16 bits did not have to be undertaken.

- MIPS instruction format and binary code compatibility

The decision to retain the wide datapths allows for an implementation of the OneChip processor that is fully compatible with other MIPS machines for the instructions actually implemented. This binary code compatibility should allow for an easy porting of the gnu C compiler (GCC) [38].

- Nine basic instructions

The limited availability of logic resources on FPGAs only allows for the implementation of some basic instructions. Arithmetic functions include ADD, ADDI, SUB, AND as well as OR. Branch operations are limited to BEQ and BNE. Memory access is provided by L W and SW . Even though this instruction set is minimal, many more complex instructions can easily be modelled as a combination of the ones listed above.

- 31 general-purpose data registers

Just like in real MIPS processors, the abundance of general purpose data registers provides applications with plenty of local buffering space. Register zero is hardwired to zero. Although 32 registers are addressable, only six have been implemented due to limited flip flop availability .

- Assume-not-taken branching strategy

As already mentioned above, the branch algorithm used is not the same as the one or two delay slot approach found in commercial MIPS machines. Instead the simpler strategy of the P&H design is implemented. It assumes that branches are not taken, but should a branch actually proceed, the three delay slots are flushed. This simpler branch strategy saves some extra comparison hardware required for early branch taken/not taken evaluations required by more complex branch algorithms.

- Hazard elimination

Complex forwarding hardware ensures that all data hazards are eliminated. Load and store hazards are also detected and eliminated either through forwarding or through the stalling of the pipeline for one clock cycle.

4.1.2 Architectural Description and VHDL Coding

In this section, the architecture of the OneChip core processor is discussed in detail and the chosen grouping into functional blocks, each representing an individual VHDL module, is explained. The architecture of the CPU is clearly dominated by the characteristic pipeline structure of basic RISC machines. The five different pipe stages are easily identified in Figure 4-1. They include the instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM) and write-back (WB) sections. As will be seen shortly , this pipeline structure is well suited for implementation on FPGAs.

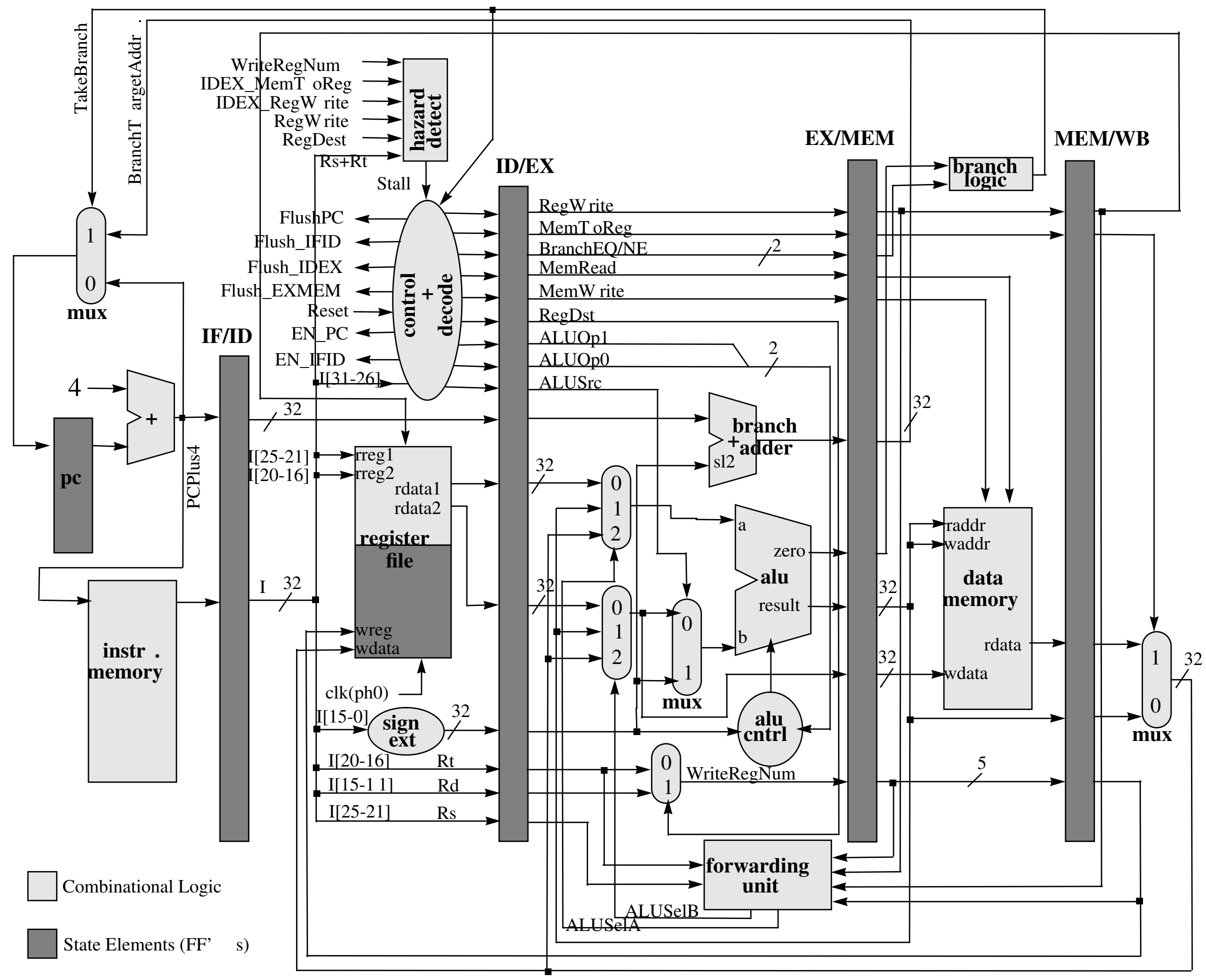


Fig. 4-1: Core CPU Block Diagram

From the discussion of the necessary FPGA logic design strategy in the previous chapter, it is clear that a design must be broken into combinational components and flip flops. A grouping of components of the same category leads to the formation of functional blocks. Again referring to Figure 4-1, several major functional blocks are clearly identifiable. Furthermore, the shading of the blocks indicates the component type of each group. With the exception of the register file, which is the combination of two functional blocks of opposite component types, the shading of a particular block is unique.

The architecture of the CPU can be compared to the one of a complex sequential circuit. Combinational logic operates in between the state elements of the pipe stages. Each stage takes data from the previous one (viewed in space and/or time) and performs some computation. In this fashion data is passed on every clock cycle. For a detailed description of the function of the individual pipe stages refer to P&H [36].

The implementation of VHDL modules follows the functional components of the block diagram on a one-to-one basis with the exception of the memory blocks and the register file. Combinational blocks are implemented using the constructs of Figure 3-5, clocked blocks employ the flip flop constructs of Figure 3-6. In the VHDL code, the instruction and data memory module are merged, employing both combinational and clocked logic, as they access one unified main memory . The code for the register file distinguishes between the combinational reading out of data from the registers and the clocked storing of new data. The register file is driven on the opposite edge of the clock than the pipeline registers (simulated two-phase clocking) to be able to pass data just stored in the register file onto the EX stage within the same clock cycle.

Clearly, the outlined break-down into functional blocks of both combinational components and flip flops follows the FPGA logic design strategy outlined in the previous chapter . Furthermore, the principle of pipelining the processor design with its alternating combinational and clocked component blocks matches the design approach of sequential circuits. The architecture of the basic RISC CPU can even be viewed as an integration of several sequential circuits, each consisting of one pipe stage, both in space (directional data flow from left to right in Figure 4-1) and in time (forwarding of data required out of directional order).

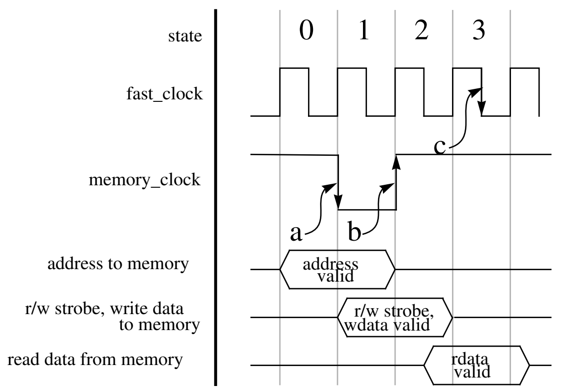
4.2 Transmogrifier -1 Implementation Issues

As outlined in the previous section, the design of the VHDL code of the processor only considers the technology-independent problem of synthesizing a large logic design from an HDL description into FPGAs. For an implementation of the processor on the TM-1 system several board specific issues need to be addressed. Up to this point no consideration was given to the memory interface, to the programming of the instruction memory and to the global partitioning of the processor design. It will be demonstrated that these issues require a TM-1 specific solution, before the implementation of the processor can proceed.

4.2.1 Memory Issues

A typical CPU contains at least three major components that require storage elements: the register file, the instruction memory and the data memory . With the introduction of the Xilinx XC4000 FPGA family , for the first time a reconfigurable device had the capability to provide on-chip storage elements other than flip flops. However , the availability of this memory comes at the expense of function generators whose storage elements can only be used for either programming the look-up table or for implementing arrays of RAM cells. To overcome this limitation, the TM-1 system provides four external 32k x 9-bit SRAM chips. In anticipation of the complexity and logic resource requirements for a big design like the processor, it was decided to implement the two larger storage blocks, the instruction and data memories, in the external RAM. The four SRAMs are used in parallel to obtain a 32-bit wide memory word. Only the smaller 32 x 32-bit register file is built with the storage elements available inside the actual FPGAs.

Using the external SRAMs on the TM-1 requires a special memory accessing scheme. As illustrated in Figure 4-2, the memory chips require their own clock (*memory_clock*). Address and write data along with the strobe indicating a read or write operation are synchronous inputs. Read data is returned asynchronously as the only output of the RAMs. The necessary



Event	Description
a	memory latches address
b	memory latches r/w strobe, write data (if access=write)
c	CPU latches read data (if access=read)

Fig. 4-2: TM-1 Memory Access

set-up and hold times for the inputs are not shown. A finite state machine with four states is employed to properly satisfy the timing requirements. The clocking scheme of Figure 4-2 meets these requirements for a *fast_clock* period of no less than 5 ns. It is the minimum *memory_clock* low pulse-width requirement of 5 ns that actually determines the maximum clocking frequency (*fast_clock*) of the memory controller module.

In states zero and one, the address of the data to be accessed is provided. It is latched in by the RAM chips on the falling edge of *memory_clock*, a signal also provided by the memory controller. States one and two are used to provide the RAMs with the strobe indicating a read or write operation as well as with the write data in case a memory write is selected. This information is latched in by the RAM chips on the rising edge of *memory_clock*. In the case of a memory read operation, the data coming from the RAM will be available sometime during state two, depending on the frequency of *fast_clock*, and can only considered to be stable in state three. The memory controller module latches the read information on the falling edge of the *fast_clock* of state three.

The discretization of the SRAM clock timing requirements into four distinct states determines the special clocking strategy employed for the processor. To simplify the design of the memory controller module, it was decided to unite the instruction and the data memory spaces. However, provisions must be taken to allow for two complete memory accesses per processor clock (*CPU_clock*) cycle - one required instruction fetch (read operation) and one possible memory read or write. The clocking scheme employed in the processor is illustrated in Figure 4-3. Recalling that four *fast_clock* cycles are required for a complete TM-1 SRAM

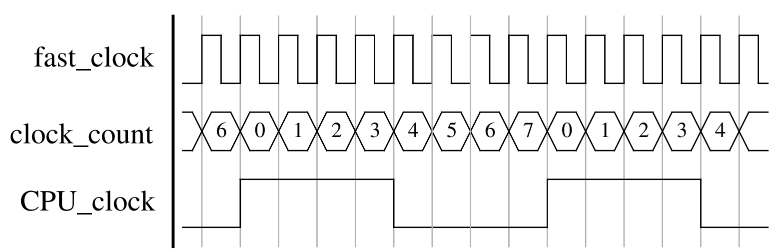


Fig. 4-3: Processor Clocks

access, one *CPU_clock* cycle with two memory accesses requires eight *fast_clock* cycles. The processor must thus be driven with an external clock operating at eight times its desired pipeline clocking frequency. A version of this *fast_clock* is used to drive the memory controller directly, while another *fast_clock* version, divided by a factor of eight, becomes the processor's *CPU_clock* that is used to drive the pipe stages.

At first, significant problems with the distribution of the *fast_clock* and the *CPU_clock* signals were encountered. The synchronization of events triggered by the two clock signals across an FPGA was not achieved. Low skew global signal distribution buffers are automatically inserted only for primary inputs with high fan-outs detected by placeBufs. High fan-out internal nodes are not yet considered by the utility program. It was thus necessary to manually manipulate the netlist files of the TM-1 FPGAs to add special global buffers to such signals as the output of the clock divider.

To download programs or to debug the memory of the processor from a front end host, a simple extension to the memory controller module of the CPU is needed. The interface that links the Sparc 5 host to the Transmogrifier-1 field-programmable system operates asynchronously. For this reason, the host cannot provide the clocking for the TM-1 memory chips. However, with the addition of a handshake protocol to the memory controller module of the processor, the required memory clocking can be provided by the CPU.

As shown in Figure 4-4, the front end host directly provides the address and the data information required to access the TM-1 RAMs. The signals *StallReq*, *Ready*, *Go* and

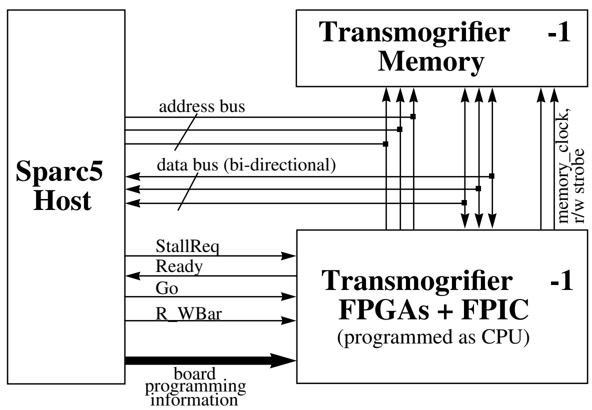


Fig. 4-4: Host - Processor Interface

R_WBar, however, attach to the memory controller module of the processor. Their function is to halt the regular operation of the processor, to asynchronously handle the memory accesses of the front end host and to select between reads and writes, respectively. In stalled mode, the processor thus only functions as the generator of the *memory_clock* and the read/write strobe signals. It is the responsibility of the front end host to hold the address and data information stable while the memory controller module provides the required clocking.

Due to FPGA pin constraint problems (further discussed in the following section), the address and data buses from the host to the TM-1 memory could not be passed through the memory controller module to the memories. The sourcing of two signals onto one bus therefore requires the use of tri-state bus drivers. The signal *StallReq*, when asserted, causes the address and data bus drivers of the processor to be tri-stated and the drivers of the host to be enabled. When not asserted, the host's drivers are disabled and the processor's are active.

4.2.2 Partitioning Issues

The use of multiple FPGAs to implement designs exceeding the amount of logic resource available on a single chip brings along several partitioning problems that need to be addressed. Several factors determine the relative success of a given partition. The amount of logic resources, the number of I/O pins and the achievable clocking frequency of each partition are examples for the measure of success. The difficulties encountered during the implementation of the MIPS-like RISC CPU on the TM-1 field-programmable system are discussed below .

The main challenge encountered involved balancing the amount of available logic resources with the amount of available I/O pins. The task was to find a successful partitioning strategy for the 1 157 CLB function generators and the 889 CLB flip flops of the basic, unenhanced processor without violating the constraints given by the 157 usable I/O pins, 800 available CLB function generators and 800 available CLB flip flops of each XC4010 FPGA found on the TM-1 system. The TM-1 automatic partitioner failed to find a successful partition that did not exceed either the available logic or the available pin count. An analysis of the design shown in Figure 4-1 indicated that a partitioning strategy removing one or more modules from the logic resource intensive pipe stages ID, EX and MEM would always result in the violation of the available I/O pin count of an FPGA due to the significant quantity of 32-bit wide buses leading to and from these building blocks. It was thus necessary to take an unconventional approach.

Inspired by work done at MIT [39], it was decided to time-division multiplex several 32-bit wide signals onto one bus to decrease the need for I/O pins. Figure 4-5 gives an example. MIT's virtual wires project includes an automated software environment to partition a large logic design across several FPGAs, to determine the number of logical wires required per physical wire and to generate the required muxing and demuxing logic constructs. This environment was not available for the OneChip project, requiring the manual execution of the above three steps. The choice of eight virtual wires per physical wire comes from the fact that the eight discrete intervals involved in the handling of two memory accesses per CPU clock cycle are also used to drive the time-division multiplexer of the virtual wires module. This sharing of clocking information is necessary as the number of global clock distribution buffers on the XC4000 FPGAs is limited.

Using the virtual wires approach, the ALU and the register file of the processor could be placed into separate FPGAs. For each component, the three 32-bit wide signals leading to

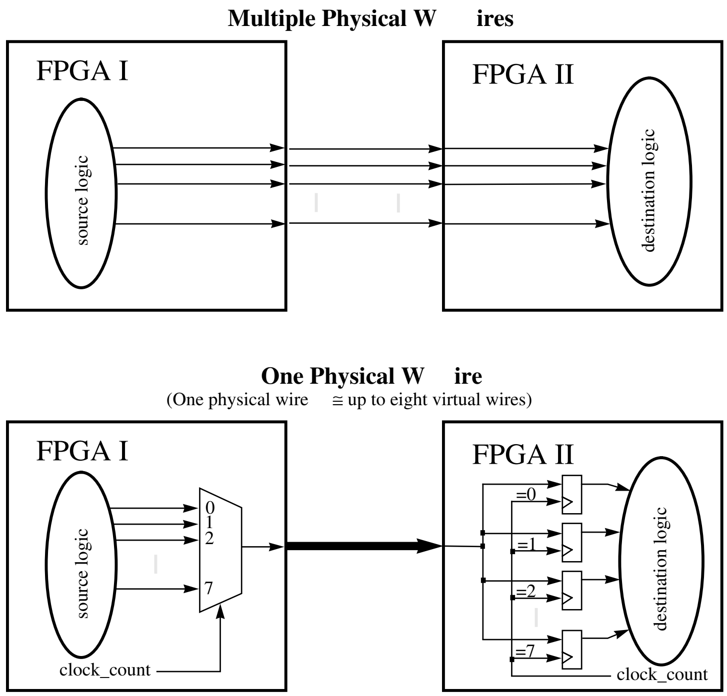


Fig. 4-5: Time-Division Multiplexing Physical Into Virtual Wires

and from the logic could be merged into one 32-bit wide bus. The combination of input and output signals onto one bi-directional bus required a slightly more complex approach than illustrated in Figure 4-5. However, this extra reduction of the I/O pin requirement was necessary to have sufficient pins available to be able to cut and connect both the ALU and the register file from the processor core to make the remaining, still complex, core logic fit into one FPGA.

The use of bi-directional bus constructs employing tri-state drivers is not yet handled by the automatic partitioning tool *part* (Section 3.2.2). The processor design needed to be partitioning manually. Figure 4-6 illustrates the partitioned processor design along with the logic resource utilization for each of the TM-1 FPGAs. It is interesting to note that even with only 86% of the I/O pins and 66% of the F&G function generators used, it was extremely difficult to route FPGA 1. Xilinx’s *ppr* tool had to be executed with the advanced setting for both placer and router efforts and required over five hours to finish. An earlier version of the FPGA 1 design containing two extra flip flops, which could eventually be optimized away, failed to route altogether. By contrast, FPGAs 2 and 3 could easily be routed within 15 minutes each. In anticipation of the addition of extra reconfigurable resources into the main

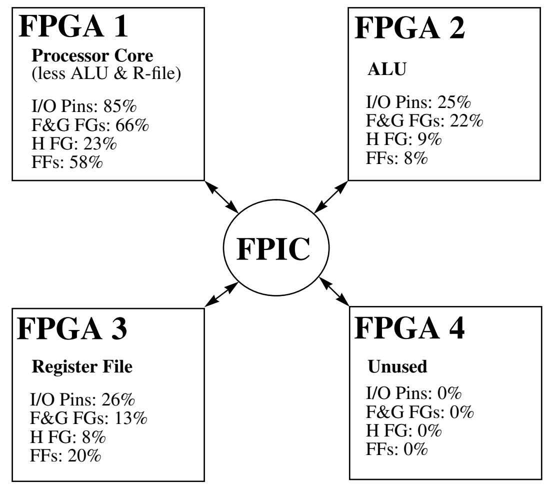


Fig. 4-6: TM-1 Processor Partitioning And Device Utilization

functional unit of the processor, the ALU, to prototype CCM interfacing architectures, the logic utilization of FPGA 2 is kept low. Also, the register file of FPGA 3 presently only implements five 32-bit registers, which keeps the device utilization low and the place and route times short. This small number of registers is sufficient for the testing performed as part of this thesis. Extra registers, up to 32 in total, can easily be added through a minor change of the relevant VHDL code.

With the partitioning of the processor design into multiple chips and its subsequent interconnection using slow, external wires, the synchronization of clocked events across the chip boundary became a problem. As the TM-1 field-programmable system does not provide any board-level, low-skew signal distribution network, similar to the global buffers available in XC4000 FPGAs, it was necessary to replicate the clock divider circuit onto every FPGA. The initial version of the partitioned processor design, in which the clock_count signal was generated in FPGA 1 only and distributed to FPGAs 2 and 3, exhibited clock_count skew problems in the form of synchronization failures amongst virtual wire multiplexing modules on any two FPGAs. However, the fact that the fast_clock signal driving the clock divider modules on FPGAs 1, 2 and 3 (refer to Figure 4-6) is routed through the FPIC interconnect, which does not guarantee equal delays on all of its internal routing paths, did not seem to have any impact on the proper operation of the partitioned design at the clock frequencies (see next section) used.

4.3 Testing and Performance Results

Only continuous testing of large logic designs, like the processor, at every stage of their development phase can guarantee a working circuit. Especially when designing with multi-chip FPGA systems, the frequent simulation of a logic design is essential, because the debugging of designs by means of monitoring actual signals on the I/O pins of the chips on field-programmable system like the TM-1 is a very tedious and time consuming process. The following two sections will describe the testing strategy employed throughout various phases of the design of the MIPS-like RISC processor and the speed critical sections identified to be responsible for the observed performance figures.

4.3.1 Testing Strategies

Simulations and tests verifying the proper operation of the CPU were run at several design stages. For the design flow of the processor from VHDL code to the TM-1 implementation, four major design levels were identified:

- Ideal, delay independent, order of event-occurrence modelling level
- Functional, unit delay, gate primitive modelling level
- Timing, back-annotated delay, placed and routed logic modelling level
- Real, actual delay, physical implementation modelling level

The simulation of VHDL code using Viewlogic's ViewSim environment represents the ideal, delay independent design level. At this design level, the VHDL simulator uses an internal scheduling of events based on order of occurrence to model sequential circuits. A set of test vectors was generated to exercise the complete functionality of the CPU described in Section 4.1.1. With this kind of ideal, timing independent simulation it was possible to evaluate the proper semantic operation of the processor. Any kind of timing related effects like glitches or maximal combinational delays could not be detected at this logic modelling level. However, only few, minor semantic problems were discovered in the early VHDL code of the CPU.

After the proper functionality of the ideal VHDL code had been determined in the event driven simulation, a netlist of the CPU was generated using Viewlogic's ViewSynthesis tool. In this netlist the functionality of the VHDL code is rebuilt using Xilinx gate primitives. The timing information of each gate instance is characterized by a unit delay. Using the same set of test vectors conceived for the earlier logic modelling level, it was thus possible to perform a ViewSim gate-level logic simulation in which some basic timing related effects like glitches due to unqualified signals were discovered. Some extra circuitry was added to obtain a glitch free unit delay simulation of the processor .

Up to this point the details of the actual FPGA hardware had not yet been considered. However , with the back-annotated timing simulation most of the TM-1 hardware dependent timing details are addressed. For this design stage, the synthesized logic of the processor design was partitioned (Section 4.2.2) and the individual FPGA partitions were subsequently placed and routed. From the actual configuration of the three routed FPGAs, back-annotated netlists were generated. In turn, these three netlists were merged into one using a schematics editor (Section 3.2.2) and simulated using the same test vectors already employed for simulations at earlier design levels. Even though this timing simulation did not take the delays introduced by the field-programmable interconnect (FPIC) into consideration, FPGA hardware dependent timing effects such as clock skew and the maximum combinational delay of sequential paths could be observed. The main problem discovered through the simulation at this design stage involved clock skew within individual FPGAs. As described in Section 4.2.1, global clock buffers for high fan-out internal nodes had to be added by hand to overcome this problem.

Once the back-annotated timing-level simulation produced results indicating a properly functioning circuit, the final tests on the real hardware could be performed. As noted earlier , due to the asynchronous nature of the front end host to TM-1 interface it was not possible to use this connection for clocking purposes. Hand coded programs of MIPS instructions had to be downloaded into the memory of the processor using the host-processor interface depicted in Figure 4-4 and described in Section 4.2.1. The programs could then be executed by reseting and disabling the stall on the processor , now clocked by the crystal found on the TM-1 system. The instructions executed in this manner exercised the same functionality as the test vectors employed earlier . Results were written back into the unified memory of the CPU from where they could be recovered using the same host-processor interface used to program the memory of the CPU earlier . Two problems were encountered in this final testing stage. The first one was caused by skew on clocks distributed in between FPGAs. As a result the synchronization of the time-division multiplexers of buses (Section 4.2.2) failed. The problem could be fixed by replicating the clock divider circuitry described earlier on a per FPGA basis. The second problem was due to previously unconsidered pad and interconnect delays. It was identified to be the bottle-neck of the TM-1 processor implementation.

4.3.2 Performance Figures

From the simulations and the tests performed at the back-annotated timing and the real implementation design levels, respectively , performance figures for the delay critical paths of the TM-1 implementation of the processor were obtained. The following discussion focuses on determining the most critical of these paths, which also determines the maximum clocking frequency and thus the overall performance of the processor .

FPGA description	maximum combinational delay path (ns)	maximum clocking freq. (MHz)
Processor Core	108.0	9.259
ALU	118.3	8.453
Register File	78.0	12.8

Table 4-1: Maximum Theoretical Combinational Delay Paths

From the routing information of the partitioned design it is possible to obtain delay information on the combinational paths within a particular FPGA. Using the Xilinx tool *xdelay* , the maximum combinational delay for each of the three FPGAs utilized by the processor design were determined. T able 4-1 lists this delay information. From the data it is clear that the arithmetic logical unit exhibits the longest delay path of all the combinational circuits found inside any of the FPGAs used in the design of the processor . This observation is not surprising as the 32-bit adder inside the ALU is a big combinational circuit, even when implemented using the optimized logic structures found in *xblox* modules. The preplaced and prerouted macro-modules of the *xblox* library represent commonly used logic function building blocks that have been optimized to use Xilinx FPGA specific architectural features like fast carry chains.

While the above delay information can be trusted to be fairly accurate, it must be remembered that it was only obtained from theoretical calculations performed by CAD tools. For this reason, the performance of the working design was tested again on the real hardware of the TM-1 system. Due to the fact that no clock generator with a continuously adjustable frequency could be used, the maximum clocking frequency of the TM-1 processor implementation could only be determined to an approximate range. As illustrated in Table 4-2, it was found that the system worked up to 10.0 MHz on the processor's *fast_clock* and that the next available discrete frequency of 16.67 MHz caused improper behavior of the test programs.

processor operating?	fast_clock freq. (MHz)	fast_clock period (ns)	CPU_clock freq.(MHz)
√	10.0	100	1.25
x	16.67	60	2.08

Table 4-2: Maximum Processor System Clocking Frequencies

A detailed analysis of the delay paths in the processor is required to determine the critical link that is responsible for the overall performance of the system. In anticipation of a long combinational delay on behalf of the ALU, it was decided to allocate three of the eight *fast_clock* steps of every *CPU_clock* cycle to allow for its complete operation. However, 300 ns, corresponding to the three *fast_clock* steps available for the ALU to finish its calculations, represent a longer combinational delay than the theoretically required 118.3 ns (Table 4-1). Even when taking typical interconnect and pad delays into consideration (Table 4-3), a worst case combinational delay of only 189.8 ns (= theoretical ALU delay + Aptix delay + slow input pad delay + slow output pad delay) is calculated. The difference between the available 300 ns and this worst case delay is more than 30%, which implies that the combinational delay of the ALU cannot be the critical path of the processor.

Aptix FPIC	XC4010 input pad slow | fast		XC 4010 output pad slow | fast	
~ 30	setup: 20.0	setup:1.0	21.5	16.5
	hold: 0.0	hold: 6.5		

Table 4-3: Typical TM-1 Interconnection Related Delays (ns)

Instead, it must be accepted, that the critical path runs through the virtual wire muxing logic of either the ALU or the register file bi-directional bus. For either path, the logic signal must traverse muxing logic, an output pad, the FPIC interconnect, an input pad and a flip flop in one given *fast_clock* cycle (see Figure 4-5). The total interconnect delay of such a path is about 70 ns. Slow pads were used in the processor design as they are the easily implemented default pad type. The muxing logic adds about 16 ns, assuming a two-level combinational construct for the multiplexer and a CLB combinational delay of 8ns while the setup and hold times for a single CLB flip flop add another 10 ns. The total delay of a given time-division multiplexed signal is 96 ns. Which is very close to the maximum period of 100 ns for the *fast_clock* of the real system.

From the discussion above, it becomes clear that any complex TM-1 design requiring the use of the virtual wires, time-division multiplexing technique to overcome pin-constraint problems will be limited in performance by interconnect related delays. Even when fast input and output pad drivers are used in the FPGAs, the total delay for each signal on a multiplexed, inter-FPGA bus is at least 50 ns, assuming ideal delays for the FPIC and the absorption of the demuxing flip flop into an I/O block. The clocking period of such a system can be no smaller than n x 50 ns, where n is the number of virtual wires per physical wire. For the processor, where $n = 8$, a lower bound of 400 ns for the *CPU_clock* period implies a maximum processor operation of 2.5 MHz if the required extensive, time-consuming interconnect delay path logic optimizations are performed. The presented system, however, utilizes a 10 MHz clock for its memory controller and the virtual wire, time-division multiplexing logic, which results in a 1.25 MHz operation of the main system.

4.4 Summary

In this chapter, the reader is introduced to the issues involved in the implementation of the MIPS-like CPU of the OneChip project on the Transmogrifier-1 (TM-1) field-programmable system. A general description of the functionality of the processor followed by a more detailed discussion of the main architectural features and their VHDL analogs is presented in the opening section. Next, some of the detailed TM-1-specific design strategies like multi-level clocking and clock synchronization, global design partitioning and the solution to the FPGA pin-constraint problem are discussed. Finally, a description of the simulation and testing procedures employed throughout the four design phases of the processor implementation motivates a discussion of the performance limiting hardware constructs found in the design. The bottleneck is identified to be the TM-1 interconnect, which limits the operation of the OneChip processor to 1.25 Mhz.

Chapter 5 Architecture and Applications

Having already covered the general idea of reconfigurable computing, the CAD tools required to prototype CCM engine designs and the implementation of a minimal processor core on the Transmogrifier-1 field-programmable system, this chapter concludes the discussion of the OneChip design with the actual integration of reconfigurable logic. However, issues involved in the design of the fixed to reconfigurable logic interface need to be addressed before prototype applications can be built and used to evaluate the presented CCM architecture.

For this reason, the first section of this chapter will only be concerned with architectural issues. Motivated by some of the limitations of existing CCM systems, the problem of how to interface fixed and reconfigurable logic is discussed. The highlights of the FPGA implementation of OneChip's architecture are presented. Issues involved in a custom silicon, physical implementation of the fixed logic core processor and of the reconfigurable logic are also discussed.

Building on the OneChip architecture developed in the first section, sample CCM applications are presented. The implementation of a universal asynchronous receiver and transmitter (UART) is developed to illustrate the usefulness of the presented CCM architecture for embedded controller type applications. More importantly, however, the design of an inverse discrete cosine transform (IDCT) functional block is used to demonstrate the success of the OneChip architecture in eliminating the interfacing bottleneck found in earlier CCM architectures.

5.1 Architectural Issues

As outlined in chapter one, the motivation behind this work is to investigate the benefits of a tight integration of reconfigurable resources into fixed logic, inspired by the shortcomings of existent loosely-coupled CCM systems. However, when confronted with the opportunity to add reconfigurable resources into the heart of a processor, instead of just to an external bus, the question of where and how to do so arises. The next three sections deal with architectural challenges, the actual FPGA implementation of the OncChip architecture and with issues involved in an envisioned custom silicon OneChip implementation.

5.1.1 The Interfacing Problem

During the investigation of possible ways to tightly integrate reconfigurable logic into the basic MIPS-like processor core described in the previous chapter , many strategies were developed and subsequently dismissed for a particular limitation. The following discussion will thus point out which interfacing strategies should be avoided. This leads to the identification of an interfacing solution which is not limited by the identified shortcomings. The following interfacing strategies should be avoided:

- reconfigurable logic attached to the memory bus or any other kind of processor external bus
- a problem specific interfacing solution requiring reconfiguration of the complete processor
- reconfigurable resources not controllable by the standard scheduling and forwarding modules of the core processor
- ‘creative’ approaches not compatible with the standard MIPS instruction format

The data flow and control flow bandwidths of both fixed and reconfigurable execution units must be matched closely . Any kind of interfacing scheme that attaches reconfigurable resources to a slow bus operating either asynchronously or at a lower clocking frequency than the internal datapath of the processor will always be a bottleneck for the computing operations of a CPU. Using a bus that operates slower than the processor ’s internal datapath as an interfacing point must be avoided.

Only those processor resources required to enhance the computing performance of a particular application should be built using reconfigurable logic. Customizing the whole processor and not just some execution units as suggested in [40] can be costly , as the development time, if done manually , for customized control and datapath structures is considerable. Also, on a direct comparison basis, the implementation of common structures required by most processors like the program counter and the register file using reconfigurable resources is less dense and slower than a custom silicon design, unless such structures are also optimized on a per application basis. Furthermore, FPGAs do not yet have the amount of logic resources required to implement large logic designs on one chip. This fact has caused the designers of a low resource processor to stress the need for minimizing the use of reconfigurable resources [41].

With the addition of reconfigurable logic to the instruction decode (ID) or the memory access (MEM) stages of the MIPS processor pipeline it would be possible to perform some limited amount of pre and / or post EX stage data processing. The main computations would still be handled by the execution unit in the EX stage. However, extra control logic will be required to manage ID and MEM stage processing and extra routing resources must be used to provide for the forwarding of data to these new processing elements. It remains questionable whether this extra logic and routing hardware justifies the small performance improvement expected from ID and MEM stage processing. Until a detailed architectural study investigating the cost/performance trade-off has been performed, interfacing reconfigurable logic into a MIPS-like processor in a way that does not make use of the standard control and forwarding features should be avoided.

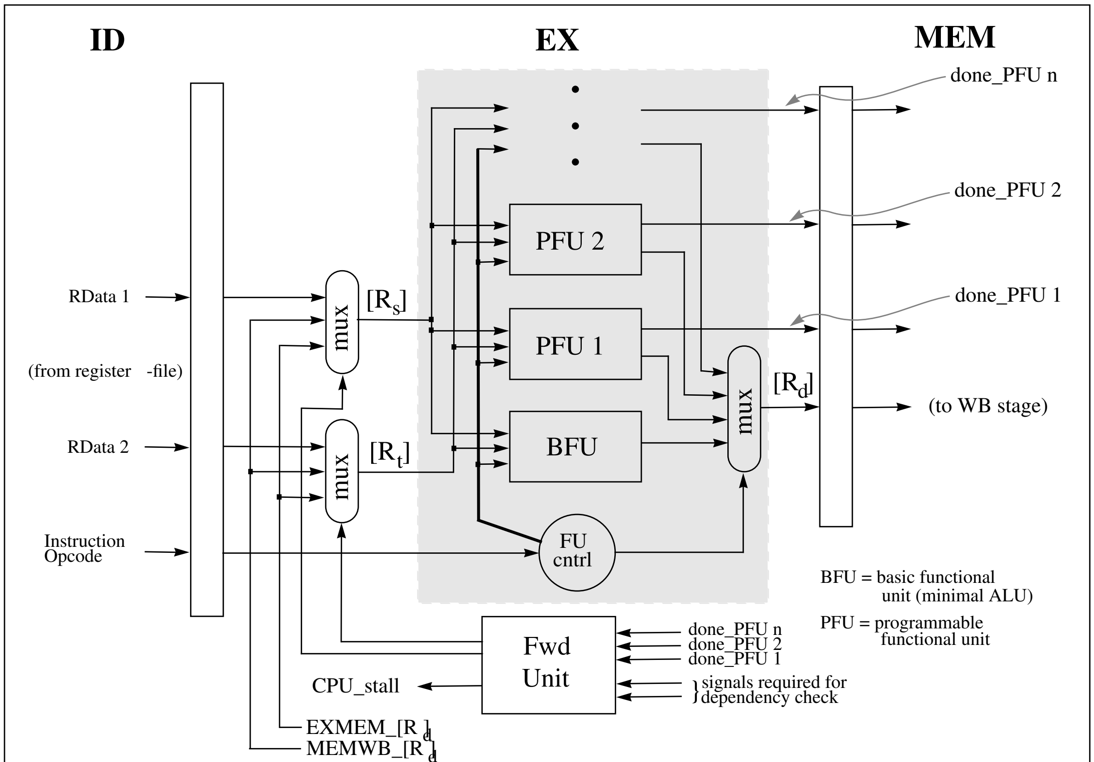


Fig. 5-1: Integration of Reconfigurable Logic into Programmable Functional Units

Apart from integrating reconfigurable resources into the regular datapath of the existing pipeline stages of the CPU, other more creative interfacing strategies have also been considered. However, they all require a divergence from the MIPS register-based, load-store scheme. Extra immediate bits were needed in the instruction word to address data sources and destinations other than the register file. The standard MIPS-like six-bit opcode had to be sacrificed to allocate space in the 32-bit instruction word for the required extra immediate data. This change in instruction format and the resulting loss of binary compatibility with basic, unenhanced MIPS executables is not desirable. It should not be required to recompile any code, unless a performance gain through the use of the new reconfigurable features is desired.

All the experience gained from the above considerations have led to the selection of an interfacing scheme as illustrated in Figure 5-1. The reconfigurable resources are added in parallel to the already existing basic functional unit (BFU) in the form of programmable functional units (PFU). The BFU is responsible for some elementary, arithmetic and logical operations required by many programs. It is built from fixed logic. The PFUs, however, can implement many application specific functions: any combinational or sequential circuit. They can also be used as programmable glue logic with some minimal pre- and/or post I/O processing features. Multiple PFU instances in Figure 5-1 illustrate the fact that run-time reconfiguration, the switching amongst several pre-compiled hardware images on one PFU, is expected to be slower than the multiplexing amongst several fixed, compile-time configured PFUs. The above interfacing strategy has the following advantages:

- Tight integration:
  - The data bandwidths of the standard and of the reconfigurable logic-based execution units (BFU, PFU) are equal
- Makes use of standard MIPS datapath and functional modules:
  - Logic other than reconfigurable resources can be made small and fast
- Employs standard control and forwarding schemes:
  - The control logic (FU cntrl) can be easily extended to address reconfigurable resources; the data dependency analysis check and the forwarding are performed as in a standard MIPS CPU; even multi-cycle PFU latency is handled at the minimal cost of an extension of the forwarding unit (fwd unit)
- Complete binary code compatibility with standard MIPS processors:
  - The BFU logic plus a provided default PFU configuration can implement all standard functions found in a regular MIPS CPU.
  - The default PFU configuration can be changed to better meet the requirements of a given application or can be traded for completely different computational or interfacing requirements; the lost default functionality must be modeled using other BFU and PFU functions by the compiler system

A similar reconfigurable logic interfacing scheme can be found in Harvard's PRISC project [17]. However, their PFU's can only implement small combinational functions that have an overall logic delay limited to one CPU clock period. This limitation makes their architecture only interesting for applications exhibiting opportunities for bit-level optimizations. Also, the lack of flip flops limits the usefulness of the PFUs as glue logic in micro-controller applications, because the frequently required synchronization of I/O signals cannot be performed.

5.1.2 FPGA Implementation

Having discussed the characteristics of a successful reconfigurable logic integration scheme in the previous section, the highlights of the actually implemented OneChip architecture are described below . Given the limited amount of logic resources available on the T ransmogrifier -1 field-programmable system, only one or two parallel PFUs of small functional granularity can be implemented in addition to the core processor .

It had been considered to implement a superscalar version of the OneChip system, however , this idea was deemed to be unfeasible given the TM-1 specific problems encountered even with the implementation of a single-issue MIPS processor (Section 4.2). The extensions to the datapath as well as the already complex memory system would have undoubtedly been impossible to realize on the TM-1 system.

For these reasons, the architecture implemented on the TM-1 to show the feasibility of the PFU integration scheme is only a scalar design, based on the MIPS processor described in the previous chapter . It has the following architectural features:

- scalar , single issue CPU
- single, fixed, 32-bit wide datapath
- two read, one write ported register file
- single read and write ported unified memory
- one functional unit (FU) with one basic fixed (BFU) and one or more programmable sub-components (PFUs, only one of which is active per clock cycle)
- variable, multiple clock cycle latency on PFU components
- forwarding module to handle dependency check and to schedule in-order instruction execution

Sample applications that demonstrate the feasibility of the PFU based reconfigurable logic interfacing scheme of this architecture are described in Section 5.2.

5.1.3 Custom Implementation

The prototype of the OneChip system, developed as part of this work, is implemented on the Transmogrifier-1 field-programmable system using commercially available FPGA technology. However, in a custom silicon implementation, TM-1 specific issues need no longer be considered and problem specific physical features can be crafted. The following discussion deals with physical implementation issues involved in:

- the design of the reconfigurable logic structures
- the relative placement of the pins and the various fixed and flexible logic blocks
- the structuring and layout of the routing resources
- the silicon area requirements relative to a XC4010

The FPGA implementation of the OneChip system described in the previous section employs several discrete PFU blocks, because XC4010 FPGAs do not allow for the run-time reconfiguration of their resources. This requires multiplexing between several pre-compiled and run-time fixed PFU images. However, a custom silicon implementation of OneChip can provide for a means of switching between PFU contexts as a new PFU image is required. Time-domain multiplexed FPGA architectures have also been proposed by Jones [42] and Bolotski, et al [16]. A tightly packed configuration memory could be used to efficiently store the pre-compiled PFU images. This approach is much more area efficient than the selective run-time disabling of pre-compiled, fixed PFU images using the general-purpose Xilinx logic structures. Jones [42] reports area savings by a factor of four over Xilinx look-up table mapped designs. It is anticipated that a custom silicon implementation of OneChip will employ this area efficient, run-time reconfiguration feature.

In addition to these optimizations of the configuration memory, a restructuring of the circuit-state or computational memory may further reduce the required silicon area. As mentioned in Section 5.1.1, the PFUs may implement any arbitrary sequential circuit. Should a given circuit require larger amounts of storage elements, the distributed style of memory found in the XC4010s does not represent a very area efficient design approach. This issue has been realized by the designers of Altera Corporation, who have included imbedded circuit-state memory arrays into their new Flex 10k CPL device [34]. A custom silicon implementation of the OneChip processor will employ an embedded, area efficient circuit-state memory array to be shared by all PFUs.

Another possible way to reduce the area requirements of reconfigurable resources has been proposed by Cherepacha [20]. He notes that it is possible to provide special routing structures for logic designs exhibiting datapath-like features. For such designs, replicated control structures could be absorbed and only the outputs of their shared programming signals need to be distributed. A reduction in the use of silicon area of up to 50% over an FPGA architecture not employing programming bit sharing has been estimated. OneChip's custom silicon structures will thus use a mixture of conventional XC4010 style reconfigurable resources employed for the implementation of single bit-based control constructs as well as datapath-style resources utilized for repeated logic structures where programming bit sharing can be employed.

It is expected that variations in the grain size of the basic logic blocks can influence the effective device utilization. By closely matching the grain size requirements of a particular application, the unused parts of individual logic blocks can be reduced. This reduction in the internal fragmentation of logic blocks results in more unused or free logic blocks, which effectively corresponds to a reduction in the required silicon area. However, a future study must first investigate the granularity of functions as well as the proportions of control-style, datapath-style and embedded RAM-type logic of typical OneChip applications before the total available silicon area can be divided amongst these individual logic structures.

Having described the structural options for a custom implementation of OneChip's reconfigurable logic, the general logic and pin placement will be discussed next. For a graphical illustration the reader is referred to Figure 5-2.

In Section 5.2.1 it will be shown that logic designs implemented in reconfigurable structures are approximately twenty times less dense than the equivalent custom silicon design. Furthermore, from design experiences with the implementation of the MIPS processor core (see previous chapter), it is known that, besides any caches or other memory blocks, the basic functional unit (BFU), the ALU, is the single largest logic module found in the processor. For these two reasons it is expected that the area of the PFUs implementing complex functions in reconfigurable logic resources will be much larger than the area of the fixed logic structures inside the processor core. This observation leads to a central placement of the smaller processor core immersed in the middle of a relatively larger sea of reconfigurable logic, where the distances between points in the fixed and in the reconfigurable structures are symmetrically balanced.

For the case in which only one PFU is configured at any time, the placement of the access point from the reconfigurable to the fixed logic is not important. However, should a superscalar architecture be envisioned or a time-division multiplexing of PFU images on a single physical PFU not be possible, more than one fixed to reconfigurable logic access point

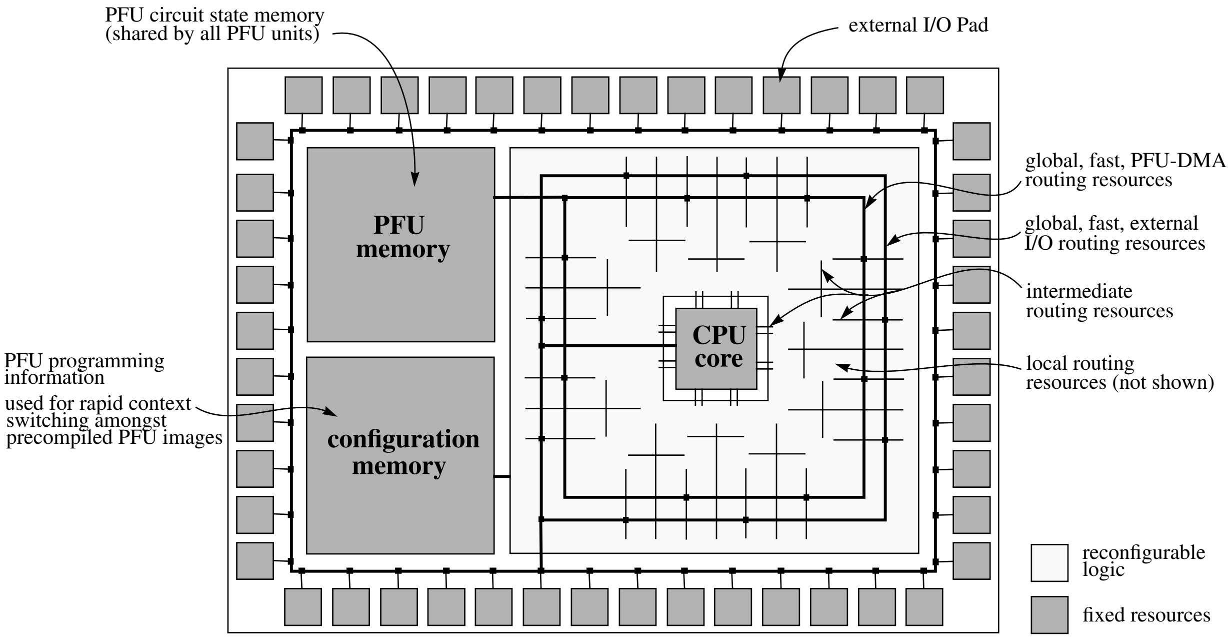


Fig. 5-2: Physical Layout of the OneChip Logic Resources

must be provided. In general, it must be expected that an arbitrary number of PFUs, n, will on average divide the available reconfigurable resources into n equal-sized logic chunks, with a certain logic locality present amongst structures occupied by a particular PFU. Thus, the PFU input and output access points should be placed equally spaced all around the fixed logic core with inputs and outputs belonging to one PFU placed next to each-other .

Other fixed logic structures include the PFU circuit state memory , the PFU configuration or context memory as well as general external I/O pads required for glue logic applications as well as external main memory accesses by the processor 's fixed core logic. The two internal memory blocks are tightly packed, regular structures. They are best placed anywhere to the side of the reconfigurable resources block where they can be accessed most easily by the routing resources interconnecting them with the sea of reconfigurable logic. The large number of external I/O pads will most likely be placed on the circumference of the block outlined by the previously mentioned structures.

Now that the fixed and reconfigurable logic structures have been assigned a relative location and the main interconnection points have been identified, the routing resources issue can be addressed. Fixed and programmable interconnects of varying lengths and propagation delays will be featured in the custom OneChip routing architecture.

Similar to existing Altera or Xilinx devices, OneChip employs fast, global nets for the distribution of signals required at many points across the chip. These long routing lines are needed to connect the sea of gates and the memory controller module of the processor core to the external pins for embedded controller applications and for external main memory access, respectively. Due to varying distances from the sea of gates to the numerous external pads, a separate pad routing network surrounding the main logic blocks of OneChip is envisioned to become pin-placement independent.

This network has an extension into the sea of gates in the form of a ring. A similar routing ring extends into the reconfigurable logic block from the PFU circuit-state, embedded memory array. Connection points to these rings are provided by intermediate routing resources at frequent, evenly spaced intervals. Intermediate length routing resources span the distance of several local interconnect resources combined, however, do not have the extensive reach of the global routing resources. Less frequent direct access points to the fast, global rings would only be useful if their location could be pre-determined and quickly reached from the basic logic blocks with local interconnect. However, changing access point locations may result in longer paths through the slow, local interconnect. To avoid this scenario, the custom OneChip architecture employs frequently and evenly spaced, intermediate-length routing resources.

Intermediate routing resources are also found initially uncommitted within the sea of reconfigurable gates and as dedicated connection points to the core processor's fixed logic. The embedded general purpose intermediate interconnect, with its horizontal and vertical lines all throughout the chip, can be used to connect basic logic blocks whose interconnection distance is more than a few local lines. Those intermediate lines leading to and from the fixed core logic are used as modular interfacing points. They can be reassigned amongst neighbor PFUs to balance the amount of interconnectivity required by one given PFU (the width of PFU buses, made up of intermediate interconnect, may vary) or they can be used in series with uncommitted intermediate-length lines to provide access for those PFU signals with a far away sea of gates connection point. This kind of modularity of the intermediate routing resources core logic interface would be harder to achieve if the faster, long interconnect lines had been employed instead.

Even though local routing resources have been referred to already, they are not explicitly shown in Figure 5-2. They are employed to either connect a basic logic block to its immediate neighbor or to intermediate routing resources which, in turn, link the basic blocks to other siblings.

Having described the relative placement of the major OneChip components, an area estimate for the relative dimensions of the fixed logic core processor and the reconfigurable structures will given next. Assuming we start with a device that has a flexible logic area, which includes the configuration and circuit-state memories, similar to that of a XC4010 [11]. The die area of a 0.8 μm Xilinx FPGA has been measured from a broken device and found to be approximately 144 mm^2. The area of the basic MIPS-like core processor can be estimated from an early 32-bit MIPS processor implementation described in [35]. In a 2.0 μm process, the MIPS-X processor measures approximately 6 mm x 6.5 mm. Scaled by a factor of 0.8/2.0, to account for the improved process technology, this area becomes 2.4 mm x 2.6 mm or 6.24 mm^2.

From the above area figures it can be seen that a complete MIPS-X processor will fit into only 4.33% (6.24 / 144) of the area required by a XC4010. In the custom implementation of the OneChip system similar relative area requirements of the fixed versus the flexible logic resources can be expected. The dimensions of the core processor may vary slightly from the ones of the MIPS-X CPU, depending on the amount of functionality provided in the BFU. The area of OneChip's reconfigurable resources may also differ from that of a XC4010. The use of optimized structures for both the reconfigurable logic, the circuit-state and the configuration memories may reduce the area requirements. The addition of extra logic resources to facilitate larger PFU images will increase the area requirements. Overall, it is expected that a custom implementation of the OneChip system will allocate less than 10% of its silicon area to its fixed logic core processor.

5.2 Application Implementation and Feasibility Issues

Having introduced a promising interfacing scheme for integrating reconfigurable logic resources into a fixed, MIPS-like processor, its feasibility can now be evaluated by building sample applications utilizing OneChip's reconfigurable features. With its general interfacing approach, the OneChip system may be employed for two types of applications:

- embedded controller type problems requiring custom glue logic interfaces
- application specific accelerators utilizing customized computation hardware

For both types of applications, the TM-1 field-programmable system is configured with the basic MIPS-like processor, whose implementation is described in Section 4.2, augmented with one or more programmable functional units (PFU). For complexity reasons, the TM-1 implementation of the OneChip CCM system only employs the single issue architecture described at the end of Section 5.1.2. Prototyping with the TM-1 system also does not allow for the use of any OneChip optimized reconfigurable logic or routing structures recommended for implementation of a custom silicon design in the previous section. The PFUs of the processor are configured uniquely at the time of TM-1 power-up, depending on the needs of the given application. For either application type, a multiplexer is used to select amongst the basic functional unit (BFU) and the PFU images.

5.2.1 Embedded Controller Type Applications

Well established, cheap, easily programmable CPUs like Motorola's 68000 series have become the basic building block for the cost sensitive yet large and lucrative embedded controller applications market. With the availability of reconfigurable logic in between the external pads and the processor core, application specific glue logic interfaces are easily implemented on the OneChip system.

To demonstrate the usefulness of OneChip's PFU-based reconfigurable logic integration scheme for embedded controller type applications, several glue logic configurations are considered. For comparison purposes, only commercially available glue logic blocks are chosen for implementation. The Motorola MC68306 applications [43] are singled out, because VHDL code describing their functionality could be obtained from another project undertaken in the same research department [44]. Simple models of the parallel port, the DRAM controller and the universal asynchronous receiver and transmitter (UART) modules of the MC68306 micro-controller are implemented in the reconfigurable structures of a Xilinx 4010 FPGA to obtain area estimates. However, only the UART has been fully integrated into the OneChip processor core as a PFU application to demonstrate a completely operational system.

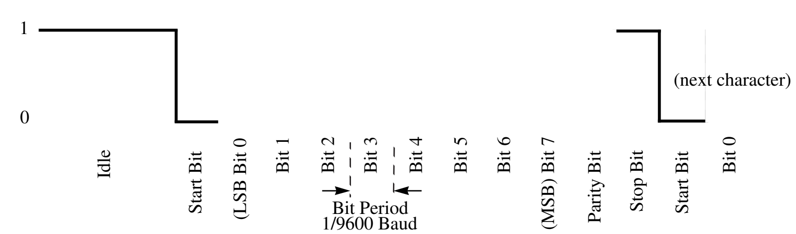


Fig. 5-3: Serial Character Transmission Characteristics

UART Implementation

The implementation of the UART uses transmission characteristics that are determined at hardware image synthesis-time. Instead of providing several usable baud rate, character length, parity and stop bit settings, these parameters are fixed, considerably reducing the UART logic count over the more complex MC68306 module. Should varying settings be required at run-time, several pre-compiled PFU images must be employed.

The basic OneChip UART implementation uses the serial transmission characteristics shown in Figure 5-3. Eight data bits are followed by one odd parity bit and one stop bit, at a transmission rate of 9600 Baud. The UART has two serial channels, one of which is set up for receiving, with the other set for sending characters. The operation of either channel is buffered by a four-deep FIFO, which smoothes out possible latencies in the processor's access of the UART module due to slower clocking rates of the UART over the core processor. Using these FIFOs, the UART handles the sending and receiving of data asynchronously from the rest of the processor whenever the send buffer contains data or the receive buffer is empty.

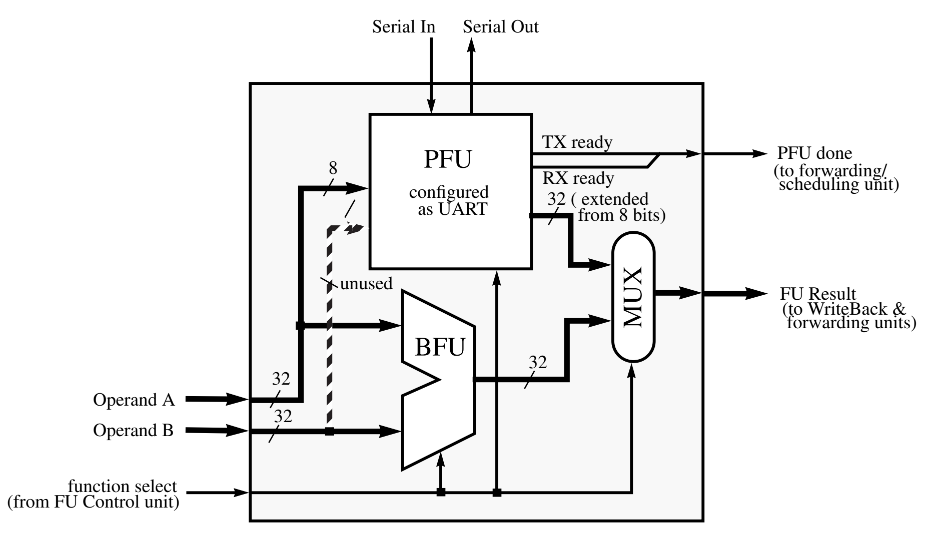


Fig. 5-4: Functional Unit With PFU Programmed As UART

Figure 5-4 shows the functional unit of the OneChip system programmed with the UART module. The block containing both the BFU and the PFU images, the extended functional unit of the OneChip system, nicely fits into FPGA 2 (see Figure 4-6) after the CCM system is partitioned for TM-1 implementation purposes. The two signals *Serial In* and *Serial Out* are links to the outside world. The UART itself is addressed through the use of two new instructions: UARTR and UARTW. They are implemented as a simple extension to the core processor instruction set. Their format is shown in Figure 5-5. Upon the issue of these new instructions, the forwarding unit will first check the signals *TX ready* and *RX ready*, which indicate the state of the transmit and receive buffers, before the processor can read or write data to and from the functional unit. If the required buffers are not available, a pipeline stall may be asserted.

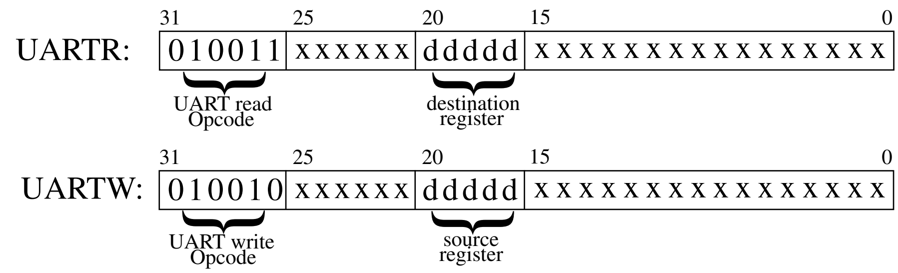


Fig. 5-5: UART Instruction Format

In the OneChip implementation, only eight-bit data words are passed to and from the UART PFU. However, using reconfigurable resources, other configurations can be realized quickly. Figure 5-6 illustrates the format of UART data. After a character has been read from the UART receive buffer, the bits *frame error* and *parity error* may be checked using bit operations of the core processor to detect transmission errors.

Test programs have been coded to verify the functionality of the UART described above. One application links the OneChip UART system to a Volker-Craig VC5220 ASCII terminal. Using this setup, the processor echoed data typed on the terminal's keyboard back onto the terminal's screen.

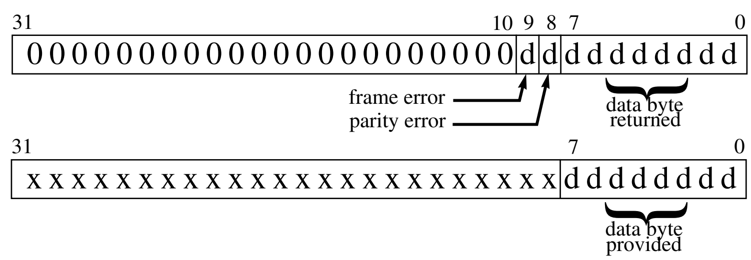


Fig. 5-6: UART Data Format

DRAM controller Implementation

The main functionality of the DRAM controller module is to provide the column and row access strobes (CAS, RAS) as well as the chip select signals on eight sets of 1Mbyte or four sets of 16Mbyte memory spaces. Similar to the design of the UART, the programmable functionality of the MC68306 DRAM controller has been modified for implementation with reconfigurable logic. In its original version, the memory controller has a variable refresh delay and the number of wait states can also be programmed. These two parameters are now to be fixed at hardware synthesis time and again, similar to the TM-1 implementation of the UART, a new PFU image must be generated for each set of refresh times and wait states.

The synthesized hardware image of the DRAM controller has not been integrated into the OneChip system for two reasons. First, its FPGA implementation is mainly intended for area comparison purposes with a fixed, custom silicon design and secondly, the MIPS core processor already contains its own TM-1 specific memory controller module (Section 4.2.1).

Parallel Port Implementation

In its original MC68306 glue logic version, this interfacing module has two eight-bit ports which can be individually programmed as either inputs or outputs. For each bit on the port, one register is used to latch input data, another one is used to latch output data, while a third register is employed to control the directionality. Furthermore, in its original version the parallel port module is memory addressed, requiring extra address decode logic. To show the benefits of using reconfigurable logic to reduce the required amount of logic resources and silicon area, several parallel port versions have been implemented on the TM-1. The original version, described above, is included as an upper limit for logic and area requirements. Other versions have their address decode logic stripped - they are expected to be instruction addressed, similar to the TM-1 UART implementation - and have the number of port-pins and their directionality fixed at hardware synthesis time.

With their varying port-pin counts, the parallel port controller-glue module versions are excellent examples to show the benefits and flexibility of customizing logic to the needs of a particular application. Again, none of the TM-1 implemented parallel port versions have been integrated into the OneChip system yet, since they are only studied to compare silicon area requirements of the fixed and the reconfigurable logic implementations. An integration of the parallel port module into the OneChip architecture would proceed analogous to the integration of the UART module described above.

Feasibility Considerations

When evaluating the feasibility of using reconfigurable logic resources to build custom glue logic interfaces for embedded controller-type applications, the ease of implementation and the logic resource requirements must be considered. Clearly, the major advantage of using reconfigurable controller glue logic lies in the flexibility of being able to customize interfacing logic on an application basis and in the possibility to reuse one CCM system for several different embedded controller-type applications. However, the implementation of logic in reconfigurable structures is not as dense as in full custom layouts. The customization of glue logic interfaces can reduce the overall logic resource requirements by omitting unused structures found in general, fixed interfaces, but an overall penalty of a larger silicon area for the glue logic interface must be paid when reconfigurable resources are utilized for its implementation.

To determine the severity of this penalty, the area of equivalent custom silicon and reconfigurable implementations of the presented applications must be compared. It was not possible to directly obtain information on area requirements of the MC68306 modules from Motorola. However, to make approximate order of magnitude area comparisons, the minimal VHDL code constructs of the controller-glue applications were synthesized into a 0.8 µm BiCMOS process available at the University of Toronto using Synopsys' Design Analyzer.

The area of the unrouted custom silicon implementation of a design could thus be obtained. From past work [45] it is known that the area of designs synthesized using Synopsys' tool grows by a factor of approximately 50% when these are placed and routed using the Cadence tool suite. This factor applies for combinational, non-combinational and sequential circuits. The area of the custom implementation of the above applications listed in the second column of Table 5-1, has already been adjusted to account for the routing resource requirements.

The resource requirements of the FPGA implementations of the given applications are listed in the third and fourth column of Table 5-1. They represent the required number of configurable logic blocks (CLB) and the equivalent FPGA silicon area. The former was obtained from the reports generated by Xilinx's partition, place and route tool *ppr*. The latter was calculated by multiplying the total FPGA die area by the ratio of the number of actually required CLBs to the number of available CLBs. The die area of a 0.8 µm XC4010 was measured from a broken FPGA, while the total number of CLBs for a XC4010 was given by Xilinx to be 400. A direct area comparison of custom versus FPGA implemented glue logic applications is thus feasible.

It becomes apparent that a significant reduction in the amount of logic resources required can be achieved when XC4010 FPGA technology is employed to customize a particular glue logic application (see right-most column of Table 5-1). In the case of the parallel port module, a switch from a memory-based to an instruction-based addressing scheme in conjunction with the fixing of the directionality can reduce the area requirements from 19.08 mm^2 to 2.880 mm^2. This represents an approximate ten-fold area improvement.

Application	Custom Silicon Implementation	FPGA Implementation	
	area (mm^2) [1]	# of packed CLBs	area (mm^2) [2]
UART	2.879	124	44.64
DRAM Controller	2.174	141	50.76
Parallel Port memory addressed, 16x1-bit I/O ports bit-wise programmable	0.804	53	19.08
Parallel Port instruction addressed, 8 input, 8 output ports, fixed directionality	0.231	8	2.880
Parallel Port instruction addressed, 9input, 19 output ports, fixed directionality	0.394	14	5.040
Parallel Port instruction addressed, m input, n output ports fixed directionality, $(m+n) \leq 32$	$\approx$ 0.014 x (m+n)	(m+n) / 2	0.180 x (m+n)

[1] Synopsys' Design Analyzer does not include routing resources in its area estimates, however, the listed data has been scaled by a factor of approximately two to include the area of routing resources (Section 5.2.1).
[2] obtained by multiplying the die area by a factor of (# of packed CLBs utilized / 400). Die area was measured from a broken XC4010 to be approximately 144 mm^2.

Table 5-1: Logic Resource Utilization of Embedded Controller-Type Applications

However , a comparison of the areas of custom silicon versus FPGA implementations of unoptimized glue logic applications (refer to individual rows of columns two and four of Table 5-1), shows that the penalty for using reconfigurable logic is considerable. This penalty ranges from a factor of 15.5, calculated as 44.64 mm^2 divided by 2.880 mm^2 for the UART , to a factor of 23.7, corresponding to 19.080 mm^2 divided by 0.804 mm^2 for the parallel port. The fluctuation of the penalty is a result of the varying relative mix of different sized combinational and non-combinational cells within the particular modules.

Considering the logic optimization achievable through the use of reconfigurable resources, the implementation related area penalty can be reduced. A comparison of the areas of the unoptimized custom implementation of the parallel port versus the area of the optimized FPGA implementation of the same module results in the following calculation. The area penalty is evaluated to be 3.58, corresponding to 2.880 mm^2 divided by 0.804 mm^2. Clearly, the design flexibility introduced with the use of reconfigurable logic can be employed to significantly reduce the implementation technology related area penalty from a factor of 23.7 to another factor as low as 3.58 with the benefit of being able to implement the exact logic interface required.

It should also be noted that designing optimized glue logic applications is a simple, fast procedure, which can be directly accredited to the use of FPGA-like structures. All the presented glue logic applications were coded in VHDL in less than two working days. Furthermore, the complete design process of the UART module including integration into the OneChip system as well as functionality testing required less than five working days. Minor hardware changes to accommodate a faster baud rate could be implemented in less than one hour. Overall, the OneChip architecture allows for the fast customization of embedded controller interfaces as well as the possibility to reuse the same system for numerous different applications at the cost of a slightly larger silicon area than required by the fixed, custom silicon implementation of a given controller glue application.

5.2.2 Performance Enhancement Applications

With the availability of tightly integrated reconfigurable logic resources within the core of the OneChip architecture, certain functions not well suited for evaluation using the regular functional unit may be processed faster. Similar performance enhancement CCM systems have been reviewed in chapter two, however, most of them only achieve significant speedups for large grain functions with a relatively small communication overhead. In this section it will be shown that the tight integration of reconfigurable resources into OneChip's architecture makes it useful for enhancing the performance of even those applications in which the required communication bandwidth is relatively high and the grain size is relatively small.

First, an introductory discussion deals with the selection of one particular application that can benefit from the OneChip architecture. The implementation of this application is described next. For performance evaluation purposes, the execution times of the given application, as implemented on the TM-1 and on a MIPS R4400 system, are determined. Finally, detailed speed-up calculations are given.

Application Selection

For the selection of applications targeted for speedup on the OneChip system, an analysis of code constructs not efficiently evaluated on fixed logic processors is useful. While a fixed-logic MIPS architecture must evenly distribute its hardware resources amongst functional units most utilized by a broad range of applications, a certain class of programs does not execute well on this general-purpose processor architecture. Applications ill-suited for the evaluation on OneChip's fixed-logic core processor alone, which could greatly benefit from the use its reconfigurable features, include:

- programs performing operations not utilizing the full bit-width of given functional units (ex: addition, subtraction and other operations on nibbles [46][47])
- code constructs requiring multiple clock cycles to complete on basic functional units (ex: multiplication using ALU add + shift, cosine approximations, DSP transforms)
- programs requiring more local store than the general purpose register file provides (ex: array operations like sort and search algorithms)

Several applications exhibiting one or more of the above features were considered for implementation on the OneChip system, however, the discrete cosine transform (DCT), found in the heart of every JPEG [48] and MPEG [49] coding algorithm shown in Figure 5-7,

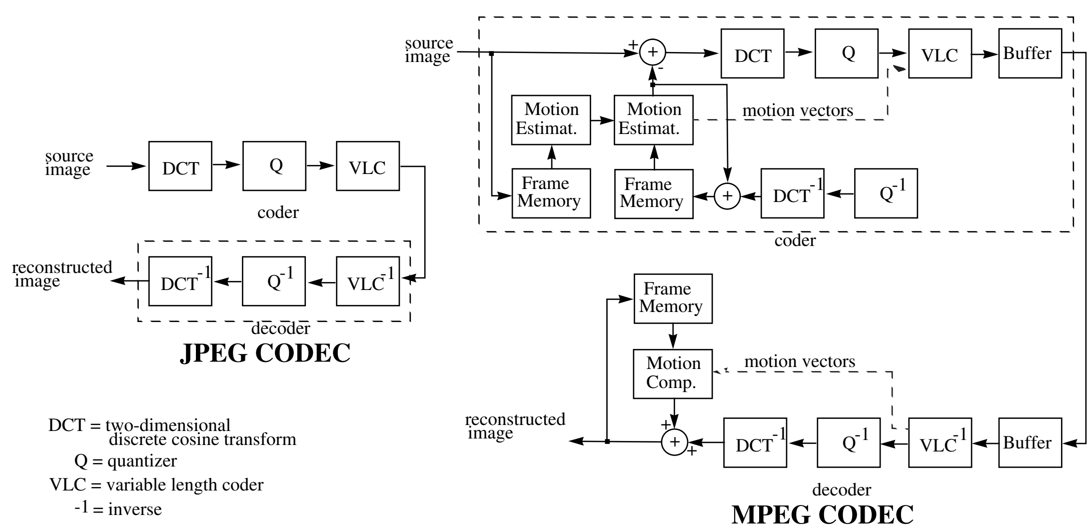


Fig. 5-7: JPEG and MPEG coder-decoders
(as illustrated in [49])

has been selected. In its one-dimensional form, it performs several sequential add and multiply operations on eight-bit quantities, requires multiple clock cycles to complete and never fully utilizes the 32-bit datapath found in OneChip's core processor. As illustrated in Figure 5-8 the convolution of the two-dimensional DCT may be separated and implemented as two sets of eight one-dimensional DCTs separated by a transpose [50], an operation exceeding the amount of local store provided by the register file of OneChip's core. The two-dimensional DCT is responsible for most of the computations performed in typical JPEG coders and still amounts to about 40% of the computations performed during MPEG playback [46].

Clearly, the DCT represents an application that can benefit from customized hardware constructs. Furthermore, the grain sizes of the one-dimensional and the two-dimensional versions represent a small grain and a large grain problem in one application, respectively. The former can be used to demonstrate that OneChip's tightly integrated architecture does not suffer from an I/O bandwidth bottleneck and the latter can be used to demonstrate significant performance enhancements, which are achievable with customized execution units of larger functional grain sizes.

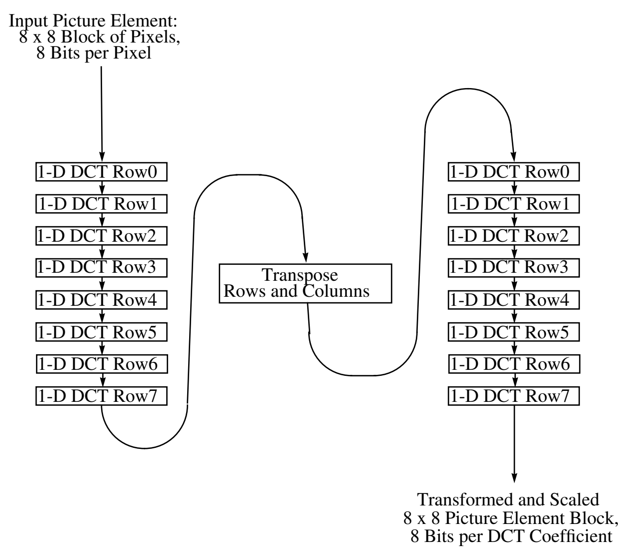


Fig. 5-8: Two-Dimensional DCT Block Diagram

Application Implementation

The primary interest in determining achievable processing performance gains through the use of OneChip's reconfigurable architecture is targeted in the implementation of the DCT. However, to achieve this goal, the details of the DCT algorithm must be analyzed before a PFU image best suited for extracting maximum processing performance from the

OneChip system can be generated. A direct implementation of the DCT is not possible as this would require a large amount of hardware, which is simply not available on the TM-1. Researchers have proposed several optimized algorithms that are suitable for a VLSI implementation. The algorithm presented in Kovac and Ranganathan’s work [48] was found to use a minimal amount of hardware resources and has thus been chosen for the OneChip DCT implementation.

The presented algorithm breaks the complex convolution summing computations into two linear , row based operations as shown in Figure 5-8. Since the operations performed resemble walking over a two-dimensional array, first in row major and then in column major order , the algorithm has been referred to as the two-dimensional DCT (2-D DCT), which involves simpler one-dimensional DCT (1-D DCT) computations. Hence the identification of a fine grain and a large grain problem. Figure 5-9 is an illustration of the six discrete computational steps involved in the 1-D DCT described by Kovac and Ranganathan.

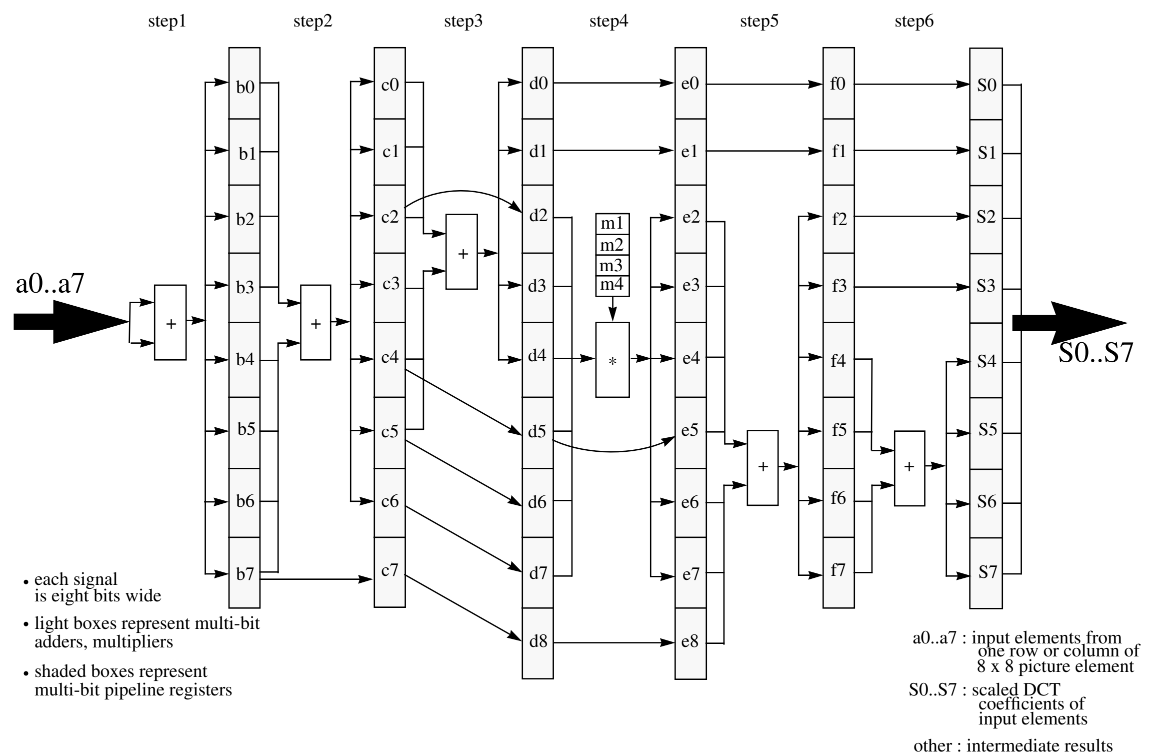


Fig. 5-9: One-Dimensional DCT Block Diagram
(as illustrated in [48])

To build a complete 2-D DCT with 16 dedicated 1-D DCTs on the TM-1 system there simply are not enough logic resources. For this reason, only the 1-D DCT is targeted for hardware implementation. The complete 2-D DCT is computed by a software program that calls the hardware implementation of the 1-D DCT 16 times and performs the required transpose operation (Figure 5-8).

The required 1-D DCT computations include 17 additions, 12 subtractions and one multiplication. To limit the latency of these computations, the design of the 1-D DCT is broken into six pipeline stages, each intended to complete within one core processor clock period. All major blocks identified in Figure 5-9 are described in VHDL and synthesized into OneChip's FPGA technology. The adders are of ripple-carry type and the multiplier is a Wallace tree version adapted from a design coded by a Qiang Wang, a PhD candidate at the University of Toronto [30].

The implementation of the 1-D DCT module has not yet been integrated into OneChip's PFU, because the objective of determining the maximum achievable application speedup does not require this. Instead, Xilinx' delay estimation tool *xdelay* [31] is employed to determine the slowest combinational path in the 1-D DCT module. Using delay information obtained in this manner in addition to other OneChip performance data obtained from the working prototype, the performance of the small grain 1-D DCT and the large grain 2-D DCT OneChip implementations can be calculated.

An actual integration of the generated 1-D DCT image would use a similar approach as detailed for the UART implementation in Section 5.2.1 and illustrated in Figure 5-4. The only difference would be a simplified forwarding and scheduling unit, because each DCT PFU pipeline stage always finishes its calculations within one processor clock cycle.

Performance Evaluation Methodology

For comparison purposes, it is desired to evaluate the performance of hardware-only OneChip DCT variants with the equivalent software-only DCT implementations. Realizing that the TM-1 based OneChip system with its extremely slow processor clocking frequency does not represent a commercially viable system, the performance comparison will consider the following two CCM systems:

- the OneChip system configured for DCT computations vs. the basic OneChip system
- the MIPS R4400 based machine enhanced with reconfigurable features configured for DCT computations versus the fixed MIPS R4400 machine

The former comparison is included to present results achieved with the OneChip architecture prototype, while the latter comparison is used to demonstrate expected performance results on an envisioned commercial system. Such a system is expected to employ an existing, realistically fast MIPS R4400 processor that has been augmented with the reconfigurable structures used by the OneChip architecture.

The methodology required to perform the above comparisons involves a break down of the DCT operation into a memory access required to load and store DCT operands and the actual DCT computation. This break down is necessary to calculate the performance of the optimized CCM systems shown in Table 5-2.

System		Memory Access	DCT Computation
1 a)	basic OneChip prototype	TM-1 specific (slow)	TM-1 general-purpose ALU (slow)
1 b)	optimized OneChip prototype	TM-1 specific (slow)	TM-1 DCT optimized PFU (fast)
2 a)	commercial R4400	R4400 specific (fast)	R4400 general-purpose ALU (slow)
2 b)	envisioned commercial OneChip	R4400 specific (fast)	TM-1 DCT optimized PFU (fast)

Table 5-2: Overview Of CCM Systems Used In Performance Evaluation

The performance of systems 1a) and 1b) can be calculated from the delay information obtained from Xilinx's *xdelay* tool. The performance of system 2a) can be measured by averaging the execution time of a number of DCT evaluations on a MIPS R4400 based computer. However, the performance of system 2b), which only exists as a theoretical architecture, must be calculated using the measured memory access and DCT evaluation times from systems 2a) and 1b), respectively.

Performance Measurement Setup

The pertinent performance information required for the comparisons outlined above are the execution times of both a memory access and the actual DCT operation of systems 1a), 1b) and 2a). These three systems have the following characteristics:

- the prototype OneChip environment described in chapter 4, utilizing the basic ALU (assumes single-cycle-latency multiplier is available)
- the prototype OneChip environment configured with the hardware 1-D DCT PFU image described above

- an SGI Indy workstation using a 150Mhz R4400 CPU with 512 MB main memory, 16 kB instruction and data caches and a 1MB unified L2 cache.

The OneChip system is taken to be running at 2.5 Mhz (=400ns clock period) which is the maximum upper bound for the CPU clock achievable with the optimized TM-1 processor as discussed in Chapter 4. The R4400 from the SGI Indy system was selected as the core processor of the envisioned commercial OneChip system, because other MIPS based SGI and SPARC based Sun workstations also available at the University of Toronto were found to be less powerful. Their execution times for the evaluation of the software DCT ranged anywhere from a few micro seconds up to an order of magnitude slower than the R4400 based machine.

The required memory access and DCT evaluation information is determined for several different DCT versions to uniquely identify the effects of two other optimization features besides the already mentioned hardware evaluation of the DCT algorithm. These additional features include loading or storing numerous bytes per memory operation (four bytes on the 32-bit prototype OneChip system, eight bytes on the 64-bit R4400 based systems) and the use of 512 (= eight rows times eight columns of eight-bit intermediate picture element data) PFU internal registers to reduce memory access requirements. The latter can be used to store the results of the computations of the first set of one-dimensional DCTs on the two-dimensional array of eight-bit numbers illustrated in Figure 5-8. The core processor's register file is not large enough to locally store this intermediate data. The resulting register spill would require several memory accesses. Thus, both of the additional DCT optimization features improve the memory access bandwidth of the PFU implemented DCT. Incorporating the two new features, the following five unique DCT versions can be studied:

- a one-dimensional DCT that loads or stores one byte per memory access
- a one-dimensional DCT that loads or stores numerous bytes per memory access (four bytes for a 32-bit system, eight bytes for a 64-bit system).
- a two-dimensional DCT that load or stores one byte per memory access and performs the transpose operation using the regular processor registers
- a two-dimensional DCT that load or stores numerous bytes per memory access (four bytes for a 32-bit system, eight bytes for a 64-bit system) and performs the transpose operation using the regular processor registers
- a two-dimensional DCT that load or stores numerous bytes per memory access (four bytes for a 32-bit system, eight bytes for a 64-bit system) and performs the transpose operation using the PFU internal registers

Memory Access Time

The memory access times of the above DCT versions as executed on the three different systems are shown in Table 5-3. They are determined as follows. The computations of the basic 1-D DCT involve 17 additions, 12 subtractions and one multiplication operation, each requiring the fetching of two operands and the storing of one result, for a total of 90 memory accesses. Employing a OneChip clock period of 400 ns (see above), the 1-D OneChip software DCT requires 90 x 400 ns or 36 µs for memory accesses. The simple hardware optimized 1-D OneChip DCT only requires one PFU operation to compute its result (with a latency of six clock cycles), but it requires the reading and writing of eight data bytes each. These 16 memory accesses can be performed in 6.4 µs. In the case where the optimized 1-D OneChip DCT is able to extract four bytes from the 32-bit memory access word, the required number of memory operations is further reduced to only two loads and two stores, for a total of 1.6 µs.

Processor System	Memory Access Time (in µsec/applicable DCT version)				
	1-D DCT (1 byte / register)	1-D DCT (4/8 bytes / register)	2-D DCT (1 byte / register) (regular transpose)	2-D DCT (4/8 bytes / register) (regular transpose)	2-D DCT (4/8 bytes / register) (PFU int. transp.)
OneChip Prototype (software DCT memory accesses)	36	*	576	*	*
OneChip Prototype (hardware DCT memory accesses)	6.4	1.6	102.4	25.6	12.8
envisioned commercial OneChip (hardware DCT memory accesses);	0.317	0.131	6.829	3.845	0.317

Table 5-3: Memory Access Time For DCT Data Load And Store Operations

The two-dimensional DCT, as implemented in this work, consists of two sets of eight one-dimensional DCTs with a transposition of the 8 x 8 data coefficient array in between the two sets. For this work it is assumed that the transpose operation can be completely absorbed by selective store and load memory operations on the intermediate array, termed "regular transpose" in Table 5-3. The basic 2-D DCT thus requires 16 x 90 memory accesses or 1440 operations, for a total of 576 µs. The simple hardware optimized 2-D DCT brings the memory operations of a single DCT from 90 down to 16 and the overall memory accesses to 16 x 16 or 256 operations resulting in 102.4 µs. The step to accessing four bytes during each memory operation for the optimized 2-D OneChip DCT brings the required memory accesses down by another factor of four to a total of 25.6 µs. Finally, the introduction of extra transpose registers into the DCT PFU, termed "PFU internal transpose" in Table 5-3, can eliminate all the intermediate stores and loads which are required between the two sets of one-dimensional DCTs. This additional factor of two brings the total memory access time for the most optimized 2-D OneChip hardware DCT to only 12.8 µs.

To calculate the performance of the envisioned commercial OneChip system, its memory access times must be determined. The necessary test runs were performed using programs that model the required number of memory accesses of the DCT versions as executed on the envisioned commercial OneChip system. The memory operations of the basic hardware optimized 1-D include eight loads and stores each, like on the OneChip prototype. To determine the memory access time of these 16 operations on a R4400 processor, the execution time of several runs of a C program that performs eight loads and eight stores was averaged. The memory access time obtained was 0.317 µs. A similar program was coded to determine the R4400 access time for one load and one store operation. This case represents the hardware optimized 1-D DCT with eight bytes per 64-bit memory access and required 0.131 µs. For a simple, hardware optimized 2-D DCT two sets of eight one-dimensional DCT memory accesses or 2x8x16 operations are required. An R4400 program modelling this scenario required 6.829 µs. When eight bytes can be used per 64-bit word, the required number of memory operations drops from 256 to 32 and the modelled R4400 memory access time reduces to 3.845 µs. With the introduction of the PFU internal transpose registers, another reduction of the required memory accesses by a factor of two is realizable, bringing the modelled R4400 memory access time to 0.317 µs.

DCT Evaluation Time

The evaluation time of the five different DCT versions is determined next. The data presented in Table 5-4 includes the memory access times already described above. Again, employing a clocking period of 400 ns for the OneChip prototype system, the total DCT evaluation time can be calculated as follows. The simple 1-D DCT requires 30 arithmetic operations (see above), each of which require two load and one store operation, for a total of 120 CPU operations or 48 µs. The simple hardware optimized 1-D OneChip DCT requires a single PFU DCT operation and eight load and store memory accesses. These 17 operations are performed in 6.8 µs. Employing memory accesses with four bytes per 32-bit word, the required operations can be reduced to a total of five which corresponds to 2.0 µs.

The simple two-dimensional DCT requires two sets of eight one-dimensional DCTs with a total of 2x8x120 or 1920 CPU operations corresponding to 768 µs. When executed in hardware, the required DCT evaluations can be reduced to 16 PFU computations, which each require eight loads and eight stores. In total, 272 operations are required resulting in an execution time of 108.8 µs. With the improved multiple byte per memory word accessing scheme, the number of CPU operations of the hardware optimized 2-D DCT can be reduced to 16 PFU operations, each requiring only two load and two store operations, for a total of 80 operations or 32 µs. Employing the PFU internal transpose register optimization, the required number of memory accesses can be reduced even further. Only eight sets of two load and two store memory accesses are required in addition to the 16 basic PFU operations. The total of which comes to 48 CPU operations or 19.2 µs.

Processor System	**Execution Time (in µsec/applicable DCT version)**				
	1-D DCT (1 byte / register)	**1-D DCT** (4/8 bytes / register)	**2-D DCT** (1 byte / register) (regular transpose)	**2-D DCT** (4/8 bytes / register) (regular transpose)	**2-D DCT** (4/8 bytes / register) (PFU int. transp.)
OneChip (software DCT)	48	*	768	*	*
OneChip (hardware DCT)	6.8	2.0	108.8	32	19.2
MIPS R4400 (software DCT)	1.754	*	29.818	*	*

* multiple bytes encoded into one data word require bit-masking instructions on fixed processor systems to separate - such an operation incurs significant computational overhead; special transpose registers can also not be realized on fixed processor systems

Table 5-4: Total Execution Time For DCT Evaluation (includes memory access time)

To determine the performance of the DCT algorithms on the R4400 based system, only two cases need to be considered. Since the processor's hardware is fixed and cannot be optimized, the performance measurement is limited to the basic one and two-dimensional DCTs. Again, the algorithms were coded and an average of numerous timed executions was taken. The observed figures can be found in Table 5-4.

From the performance figures given in Tables 5-3 and 5-4, it can be seen that the memory accesses time overhead is more significant for the hardware optimized OneChip prototype system than for the MIPS R4400 based machine relative to their overall DCT execution times. However, the appropriate use of the full memory bandwidth by reading or writing four eight-bit words per register (in the case of the 32-bit datapath of OneChip's prototype) or the introduction of local store within the DCT PFU can significantly reduce the required number of memory accesses and with it the DCT memory access time.

Speedup Calculations

Having determined the execution times of several DCT versions run on three different systems, the speedup resulting from application specific hardware optimizations can be calculated. The performance data presented above is used for calculating the speedups of the hardware optimized DCT implementations over their software-only equivalents as executed on the OneChip prototype and the envisioned commercial OneChip system.

The first comparison is included simply to show the speedups achievable with the presented architecture as implemented completely in TM-1 technology. The second comparison evaluates a more realistic system in which a much faster MIPS R4400 is employed as the fixed core processor which, however, is augmented by the reconfigurable features found in the presented OneChip architecture. For the latter system it can be expected that improvements in FPGA technology in conjunction with the architectural optimizations recommended for the reconfigurable structures of CCM systems in Section 5.1.3 will allow the DCT PFU pipeline to be clocked at faster frequencies than the quoted 2.5 Mhz.

1-D DCT (1 byte / register)	**1-D DCT** (4 bytes / register)	**2-D DCT** (1 byte / register) (regular transpose)	**2-D DCT** (4 bytes / register) (regular transpose)	**2-D DCT** (4 bytes / register) (PFU int. transp.)
7.059	24.00	7.059	24.00	40.00

Table 5-5: DCT Speedup Using OneChip Prototype

The calculations leading to the speedup figures achievable with the OneChip prototype, presented in Table 5-5, are performed as follows. Using the data from Table 5-4, the execution time of the software-only OneChip DCT implementation is divided by the appropriate time required by the hardware enhanced equivalent. Two optimized DCT versions are available for the one-dimensional algorithm, while the two-dimensional DCT is optimized in three different ways.

Obtaining speedup numbers for the envisioned commercial OneChip system, with the core processor running at 150 Mhz and the DCT PFU clocked at three different frequencies, is somewhat more complicated. As outlined in the methodology description, the performance evaluation approach includes combining the memory access times required by the numerous DCT versions modelled on the basic R4400 system with the stand-alone DCT evaluation times of the OneChip prototype. The former are listed in T able 5-3, while the latter are

DCT PFU Pipeline Clocking Frequency (Mhz)	1-D DCT (1 byte / register)	1-D DCT (8 bytes / register)	2-D DCT (1 byte / register) (regular transpose)	2-D DCT (8 bytes / register) (regular transpose)	2-D DCT (8 bytes / register) (PFU int. transp.)
2.5	2.449	3.303	2.254	2.911	4.439
25	4.913	10.257	3.992	6.648	31.158
50	5.205	11.616	4.171	7.159	46.810

Table 5-6: DCT Speedup Using Envisioned Commercial OneChip System

implicitly given as part of the total execution times of the hardware OneChip DCT versions in T able 5-4. The actual speedups of T able 5-6 are obtained by dividing the execution times of software-only R4400 execution times by the sum of the R4400 memory access and the OneChip stand-alone DCT evaluation times. Again, two optimized cases are considered for the one-dimensional algorithm and three speedup calculations are obtained for the two-dimensional case. Figure 5-10 gives a graphical illustration of the results of these calculations for the envisioned commercial OneChip system.

Using the speedup results just calculated, several interesting observations can be made. Clearly , the clocking frequencies of the fixed and the reconfigurable parts of a OneChip like CCM system must be matched. It makes little sense to augment a 150 Mhz CPU with extra computational hardware operating only at 2.5 Mhz. The speedups of the OneChip system with the slow processor are limited and may only be improved upon by increasing the clocking frequencies of both the core processor and the PFU logic. The performance gains of the enhanced R4400, OneChip-like machine, however , can be further raised as faster programmable logic resources become available. Figure 5-10 suggest diminishing returns as the operating frequencies of the core processor and the reconfigurable structures get closer to one-another .

A more significant observation concerns the grain size of the enhanced application. It is noted that even for the one-dimensional DCT , which represents the small grain application,

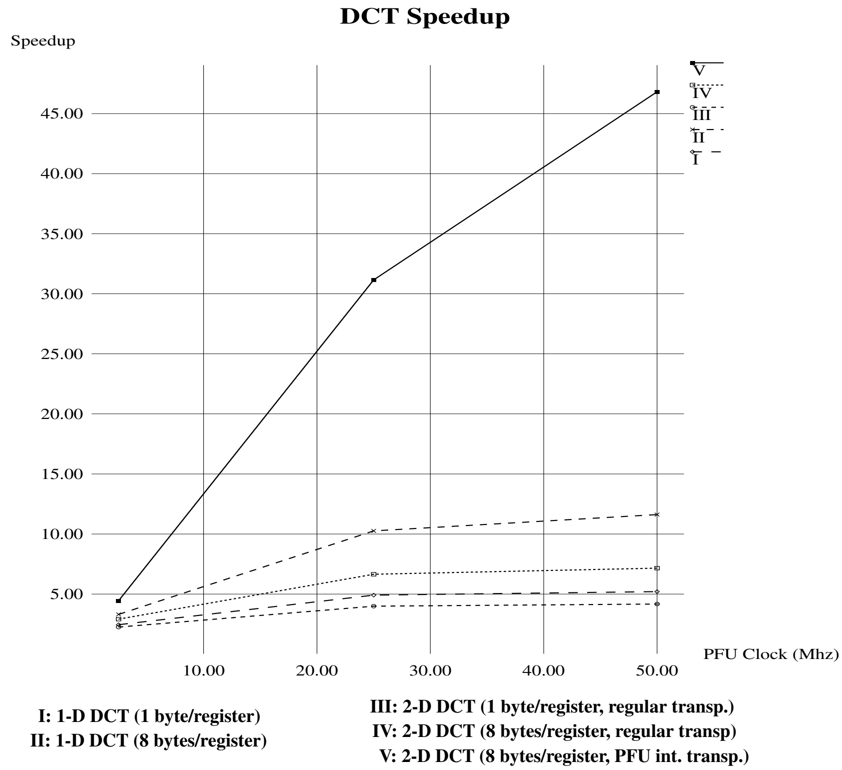


Fig. 5-10: DCT Speedup Using Enhanced R4400 System (Graph)

speedups of more than a factor of ten are achieved with the tightly integrated OneChip architectures. Such optimizations as the use of multiple numbers per memory data word, only realizable with a tight reconfigurable logic interfacing scheme make the PFU model interesting for the speedup of even relatively small size problems. The extension of the DCT PFU with local store for intermediate results brings the achievable speedup to a factor of more than forty while the granularity of the given problem is increased. However , the complexity of the hardware of the two-dimensional DCT implementation presented in this section, which fits into two or three Xilinx 4010 FPGAs, represents a much smaller grain size than required by typical applications of more loosely coupled systems [5][6], which employ 16 Xilinx 4010 FPGAs for their implementation.

It has thus been demonstrated that the interfacing scheme discussed in this work succeeds at being useful for smaller grain size applications. Furthermore, with the presented prototype and the detailed performance study , it has been shown that a OneChip-like system can be designed using present-day technology .

5.3 Summary

The first part of this chapter deals with architectural issues involved in the integration of reconfigurable logic resources into a fixed logic core processor . An interfacing scheme to tightly couple reconfigurable and fixed logic structures is introduced. The details of the programmable functional unit (PFU) interface are developed for a single issue machine before the interfacing scheme is generalized for use with superscalar architectures. Based upon the scalar system, physical implementation issues involved in the design of the reconfigurable logic structures and in the relative layout and the interconnection of the main functional blocks are discussed.

The second part of this chapter is concerned with demonstrating the feasibility of the presented interfacing scheme for embedded controller and processing performance enhancement applications. The integration of a UART module into the OneChip system is used to demonstrate the ease of customizing glue logic applications. The implementation of other interface modules is used to show that customizing glue logic constructs may reduce the required amount of hardware resources, but that the use of less dense reconfigurable structures to build these interfaces requires a slightly larger silicon area than fixed, full custom designs overall. Finally, it is demonstrated that the presented interfacing scheme succeeds at being interesting even for the speedup of small grain computational problems and that a custom OneChip system implementation is possible using today's technology.

Chapter 6 Conclusions and Future Work

This chapter contains a conclusion of the work presented as part of this thesis as well as starting points for future explorations.

6.1 Conclusion

Motivated by similar work performed in the area of reconfigurable computing and by promising speedup figures quoted for related custom compute machines (CCM), this thesis is concerned with studying architectural issues involved in the design of a reconfigurable processor. Its main goal is to overcome the bandwidth limitations present in existing CCM systems by tightly integrating a fixed logic processor core and reconfigurable logic resources into a single system called *OneChip*.

As part of this work a suitable logic design strategy required for the rapid prototyping of different CCM architectures on the Transmogrifier-1 (TM-1) reconfigurable system has been developed. VHDL synthesis tools have been studied and the coding style required for efficient logic synthesis has been described. Several custom made utilities allow for an automation of the VHDL to TM-1 design flow.

The design and TM-1 implementation of the 32-bit MIPS-like OneChip processor core are described in detail. Due to TM-1 specific problems such as FPGA pin limitations, multi-chip design partitioning and special memory-subsystem clocking, the presented processor only runs at 1.25 Mhz. The interconnect has been identified as the limiting bottleneck.

Backed with proper CCM prototyping means, OneChip's interfacing architecture is developed. The presented approach links reconfigurable structures into a functional block called a programmable functional unit (PFU). It is tightly integrated into the processor pipeline in parallel to the basic functional unit (BFU). Physical implementation issues and the extension of the OneChip architecture for superscalar systems are described.

The implementation of sample PFU images has been used to demonstrate the flexibility of OneChip's architecture for embedded controller applications. The silicon area penalty to be paid for this flexibility can be kept at a factor of less than 3.5 the area of custom silicon implementations for fixed logic applications. The OneChip architecture is also successful at overcoming bandwidth limitations of earlier CCM system for performance enhancement applications. In an example using the DCT as the application, it was shown that a speedup of over 40 could be achieved using state-of-the-art technology to implement the OneChip system.

6.2 Future Work

This thesis has only investigated one small part of the area of reconfigurable computing. Before a user-friendly OneChip processor will be found in the heart of future commercial products the following hardware and software issues must be addressed.

The architectural requirements of OneChip's core processor should be analyzed in greater detail. For the initial work presented in this thesis, a basic MIPS-like processor was employed for two reasons. First, its architectural description was readily available from P&H [36], which allowed for a detailed VHDL description and subsequent synthesis of the processor design. Secondly, to study a tightly integrated CCM system, the only requirement of its core processor had been to be easily interfaceable. This requirement is satisfied by the clearly identifiable functional unit of the highly structured MIPS machine. Additional work is required to determine the architectural requirements of the OneChip core processor beyond its fixed to reconfigurable logic interfacing point. A simpler, more compact core processor than the MIPS-like CPU employed in this work might be better suited for the use in the OneChip system.

The architecture developed as part of this work has only been prototyped on the inherently slow TM-1 system. To overcome the prototype's interconnect bottleneck, to achieve up-to-date performance with OneChip's core processor, and to allow for the use of the optimized reconfigurable structures described in the physical implementation section, custom silicon must be designed. Also, due to TM-1 resource limitations, the present implementation of the core processor only features a very minimal instruction set and no multi-level cache / memory system. In a custom implementation, the core processor would definitely incorporate more of these and other basic features typically found in present-generation CPUs.

The extension of the basic scalar OneChip architecture into the superscalar version must be considered. It can be expected that the large amount of logic resources required by a superscalar processor can be reduced by an application specific optimization of the functional unit and other parts of the datapath. The sharing of customized hardware constructs across a set of processes can contribute further reductions in the resource requirements. However, for the application of the OneChip architecture in multitasking environments some means of handling the computational state of several processes sharing one particular PFU must be provided.

Finally, the software environment required for the programming and execution of application programs on the OneChip system has yet to be developed. The system software must be able to automatically configure the programmable resources of the architecture for a reconfigurable computer to be useful. Experimental systems requiring human interaction and an extensive knowledge of the underlying hardware are not suited for the common user.

Existing CCM software environments have been fairly successful at targeting scalar architectures programmed with one or more power-up time fixed hardware images. However, the design of a run-time operating system which properly handles the dynamic compilation and context switching amongst multiple PFU images on a superscalar, time-shared system appears more challenging.

B Bibliography

[1] J. A. Hennessy, D. L. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kauffmann Publishers, San Mateo, CA, 1990.

[2] R. Jeschke, “An FPGA-Based Reconfigurable Coprocessor for the IBM PC”, M.A.Sc. Thesis, University of Toronto, 1994.

[3] D. Galloway, D. Karchmer, P. Chow, D. Lewis, J. Rose, “The Transmogifier: The University of Toronto Field-Programmable System”, *Second Canadian Workshop on Field-Prgrammable Devices*, Kingston, ON, June 1994. Available as CSRI Technical Report 306 via anonymous ftp as ftp://ftp.csri.toronto.edu/csri-technical-reports/306/.

[4] P. M. Athanas and H. F. Silverman, “Processor Reconfiguration Through Instruction-Set Metamorphosis”, *Computer*, March 1993, pp. 11-18.

[5] D. Pryor, M. Thistle, N. Shirazi, “Text Searching on Splash 2”, *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines (FCCM’93)*, April 1993.

[6] D. T. Hoang, “Searching Genetic Databases on Splash 2”, *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines (FCCM’93)*, April 1993, pp. 185-191.

[7] J. Arnold et al., “The Splash 2 Processor and Applications”, *Proceedings International Conference on Computer Design*, October 1993.

[8] D. Van den Bout et al., “Anyboard: An FPGA-Based Reconfigurable System”, *IEEE Design and Test of Computers, September 1992*, pp. 21-30.

[9] P. Bertin, D. Roncin, J. Vuillemin, “Introduction to Programmable Active Memories”, *DEC Paris Research Laboratory Report #3*, 1989.

[10] P. Bertin, D. Roncin, J Vuillemin, “ Programmable Active Memories: A Performance Assessment”, *Research on Integrated Systems: Proceedings of the 1993 Symposium*, MIT Press, 1993.

[11] Xilinx Inc., *The Programmable Logic Data Book*, San Jose, CA, 1994.

[12] M. Gokhale et al., “SPLASH: A Reconfigurable Linear Logic Array”, *Proceedings International Conference on Parallel Processing*, August 1990, pp. 526-532.

[13] D.A. Buell, "A Splash 2 Tutorial", *Technical Report SRC-TR-92-087*, Supercomputing Research Center, Bowie, Maryland, December 1992.

[14] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, S. Gosh, "PRISM-II Compiler and Architecture", *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'93)*, April 1993, pp. 9-16.

[15] A. DeHon, "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century", *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'94)*, April 1994.

[16] M. Bolotski, A. DeHon, T. F. Knight Jr., "Unifying FPGAs and SIMD Arrays", *2nd International ACM/SIGDA Workshop on FPGAs (FPGA '94)*, February 1994.

[17] R. Razdan, M. D. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units", *Micro 27*, November 1994, pp. 172-180.

[18] S. Guicionne, "List of FPGA-based Computing Machines", Hyper text link: http://uts.cc.utexas.edu/~guccione/HW_list.html

[19] Xilinx Inc., *The XC6200 Fine Grain FPGA*, San Jose, CA, 1995.

[20] D. Cherepacha, "A Field-Programmable Gate Array Architecture Optimized For Datapaths", M.A.Sc. Thesis, University of Toronto, 1994.

[21] J. H. Edmondson, P. Rubinfeld, R. Preston, V. Rajagopalan, "Superscalar Instruction Execution in the 21164 Alpha Microprocessor", *IEEE Micro*, April 1995, pp. 33-43.

[22] D. Lewis, M van Ierseel, D. Wong, "A Field Programmable Accelerator For Compiled-Code Applications", *Proc. Int. Conference on Computer Design*, October 1993.

[23] D. Yeh, P. Chow, G. Feygin, "A Multiprocessor Viterbi Decoder Using Xilinx FPGAs", *Canadian Workshop on Field-Programmable Devices*, June 1994.

[24] J. Lee, "A Field-Programmable System Emulation of an LNS Processor", B.A.Sc. Thesis, University of Toronto, 1994.

[25] D.R. Coelho, *The VHDL Handbook*, Kluwer Academic Publisher, Norwell, MA, 1989.

[26] D. Galloway, "The Transmogrifier C Hardware Description Language and Compiler for FPGAs", *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'95)*, Napa, CA, April 1995.

[27] Exemplar Logic Inc., *Core User Manual*, Berkeley, CA, 1994.

[28] Synopsys Inc, Synopsys Online Documentation V3.1, Mountain View CA, 1993.

[29] Viewlogic Systems Inc., *ViewSynthesis User's Guide*, Marlboro, MA, August 1994.

[30] Q. Wang, “16 x 16-Bit Wallace Tree Multiplier FPGA Implementation”, *VHDL Module Source Code*, University of Toronto, 1994.

[31] Xilinx Inc., *Viewlogic Interface User Guide*, San Jose, CA, 1994.

[32] Xilinx Inc., *X-BLOX User Guide*, San Jose, CA, 1994.

[33] D. Karchmer, “A Field-Programmable System with Reconfigurable Memory”, M.A.Sc. Thesis, University of Toronto, 1994.

[34] Altera Corporation, *Data Book*, San Jose, CA, 1995.

[35] P. Chow, *The MIPS-X RISC Microprocessor*, Kluwer Academic Publishers, MA, 1989.

[36] D. A. Patterson, J. L. Hennessy, *Computer Organization & Design: The Hardware/Software Interface*, Morgan Kauffman Publishers, San Mateo, CA, 1993.

[37] R. J. Gluss, “Modelling MIPS Pipelined RISC CPU in VHDL”, *ECE 280 Term Project*, University of California, Davis, CA, 1993.

[38] R. M. Stallman, *Using and Porting GNU CC*, Free Software Foundation Inc., 1990.

[39] J. Babb, R. Tessier, A. Agarwal, “Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators”, *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines (FCCM’93)*, Napa, CA, April 1993.

[40] J. Davidson, “FPGA Implementation Of A Reconfigurable Microprocessor”, *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines (FCCM’93)*, Napa, CA, April 1993.

[41] M. J. Wirthlin, B. L. Hutchings, K. L. Gilson, “The Nano Processor: A Low Resource Reconfigurable Processor”, *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines (FCCM’94)*, Napa, CA, April 1994.

[42] D. Jones, D. M. Lewis, “A Time-Multiplexed FPGA Architecture for Logic Emulation”, *Proceedings IEEE Custom Integrated Circuits Conference (CICC’95)*, Santa Clara, CA, May 1995.

[43] Motorola, Inc, *MC68306 Integrated EC000 Processor User’s Manual*, Motorola Literature Distribution, Phoenix, AZ, 1993.

[44] R. Jeschke, P. Chow, “MC68306 Based Rapid Prototyping Board”, *Design Project*, University of Toronto, 1995.

[45] B. Chan, D. D’Mello, “Starburst ATM Input Buffer Chip”, *Technical Report*, University of Toronto, July 1995.

[46] R. B. Lee, “Accelerating Multimedia with Enhanced Microprocessors”, *IEEE Micro*, April 1995, pp 22-32.

[47] B. Case, “Philips Hopes to Displace DPSs with VLIW: Trimedia Processors Aimed at Future Multimedia Embedded Apps”, *Microprocessor Report*, Dec. 1995, pp 12-15.

[48] M. Kovac, N. Ranganathan, “JAGUAR: A Fully Pipelined VLSI Architecture for JPEG Image Compression Standard”, *Proceedings of the IEEE*, Vol.83, No.2, February 1995, pp 247-258.

[49] P. Pirsch, N. Demassieux, W. Gehrke, “VLSI Architectures for Video Compression - A Survey”, *Proceedings of the IEEE*, Vol.83, No.2, February 1995, pp 220-246.

[50] W. B. Pennebaker, J. L. Mitchell, *JPEG Still Image Data Compression Standard*, Van Nostrand Reinhold, New York, NJ, 1993.