Run-time Re-configurable DSP Parallel Processing System Using Dynamic FPGAs

Ciaron William Murphy

A thesis submitted in partial fulfilment of the
requirements of the Liverpool John Moores University
For degree of Doctor of Philosophy

June 2002

Abstract

Run-time Re-configurable DSP Parallel Processing System Using Dynamic FPGAs

Ph.D. Thesis **Ciaron William Murphy** **13th June 2002**

This thesis describes the inclusion of dynamic coprocessor and routing-hub capabilities within an existing TIM-40 standard Texas Instruments TMS320C40 parallel processing environment. This work was conducted both to develop dynamic hardware applications and assess the potential benefits of this technology within an existing high performance architecture.

To integrate dynamic hardware within the TMS320C40 multiprocessor environment, a custom designed run-time reconfigurable hardware development environment was designed and constructed (XC6200DS). This system used a Xilinx XC6200 family FPGA as the dynamic hardware resource. Custom XC6200DS development software tools (XC6200ADS) were also developed, enabling temporal and spatial examinations of sequential XC6200 designs, to generate configuration data, govern XC6200DS housekeeping functions, and facilitate XC6200 FPGA run-time hardware verification.

A new BinDCT algorithm was used to develop novel XC6200 FPGA based dynamic TMS320C40 DSP coprocessor applications. Dynamic BinDCT operation increased operand throughputs from 9260 to 18520 BinDCT one-dimensional transform operations per second. This was accomplished through dynamically swapping the BinDCT hardware configuration depending on the frequency content of each transforms input data. Results obtained indicated that compared to static XC6200 configurations, dynamic BinDCT operation also improved system accuracy in approximating true DCT operation.

Using the XC6200DS, a TMS320C40 communication channel routing-hub was developed. Data paths configured within the routing-hub were updated during run-time improving processing node connectivity. This novel concept was furthered by spatially partitioning processing and routing resources (Roberts Cross Edge Detector) within the hub. This allowed the creation of a new system topology that provided additional processing hardware or node bandwidth as depicted by system operation through reusing existing hardware.

Novel dynamic hardware applications and multiprocessor operating concepts have been explored by this research. Through continual improvements in run-time reconfigurable hardware technologies, the potential benefits demonstrated can be fully exploited.

Acknowledgements

I would like to express my appreciation to my principal research coordinator Dr. David Harvey for his guidance, support and enthusiasm throughout the research project and in preparation of this thesis.

I would like to express my gratitude to my secondary research coordinator Dr. Laurence Nicolson as well as Prof. Bill Mullarkey for their time, guidance and wisdom provided throughout the project.

Many thanks to the CEORG members, especially the past and present residents of Laboratory 523 for the friendly working atmosphere experienced and comradeship encountered through the joys and woes of postgraduate research.

I would also like to acknowledge the support provide by my family and friends throughout my studies.

Finally, I wish to thank the EDAM centre for providing funding to conduct this work and the Royal Academy of Engineering for awarding travel grants to present my work.

Contents

Symbols

ADDC	*Add integer with carry*
ADDI	*Add integer*
ADDI3	*Add integer, 3 Operands*
ALU	*Arithmetic Logic Unit*
ANN	*Artificial Neural Network*
ARAUs	*Auxiliary Register Arithmetic Units*
ASA	*Applications Specific Architecture*
ASIC	*Application Specific Integrated Circuit*
C40	*TMS320C40*
CISA	*Custom Instruction-Set Architecture*
CLB	*Configurable Logic Block*
CLC	*Configurable Logic Cell*
CORDIC	*Co-Ordinate Rotation DIgital Computer*
CPLD	*Complex Programmable Logic Device*
CPU	*Central Processing Unit*
CTR	*Compile Time Reconfiguration*
DCT	*Discrete Cosine Transform*
DISC	*Dynamic Instruction-Set Computer*
DMA	*Direct Memory Access*
DPC	*Dynamically Programmable Cache*
DPGA	*Dynamic Programmable Gate Array*
DRAM	*Dynamic RAM*
DSP	*Digital Signal Processor*
EDIF	*Electronic Design Interchange Forma*
EDRAM	*Enhanced DRAM*
EEPROM	*Electrically Erasable Programmable Read Only Memory*
EFPPA	*Embedded Field Programmable Processor Array*
EPROM	*Erasable Programmable Read Only Memory*
F+V	*Fixed Plus Variable*
FBinDCT-C1	*Forward BinDCT operation using Configuration C1*
FBinDCT-C9	*Forward BinDCT operation using Configuration C9*
FDC	*Flip-flop Device* with *Clear*
FDCP	*Flip-flop Device* with *Clear Preset*
$FDCT_i$	*Fast Discrete Cosine Algorithm*
$FDCT_{ii}$	*Forward DCT*
FFT	*Fast Fourier Transform*
FIFO	*First In First Out Register*
FIPSOC	*FIeld Programmable System On a Chip*
FPAA	*Field Programmable Analogue Array*
FPGA	*Field Programmable Gate Array*
FPID	*Field Programmable Interconnection Device*
FSM	*Finite State Machine*

FU	*Functional Unit*
G	10^9
Gbytes	10^9 *Bytes*
GMICR	*Global Memory Interface Control Register*
GUI	*Graphical User Interface*
HDL	*Hardware Description Language*
I/O	*Input / Output*
IC	*Integrated Circuit*
IDC	*Internal Device Connector*
IDROM	*IDentity ROM*
IOB	*Input /Output Block*
IRL	*Internet Reconfigurable Logic*
ISA	*Industry Standard Architecture*
ISP	*In-System Programming*
JPEG	*Joint Photographic Experts Group*
JTAG	*Joint Test Action Group*
k	10^3
kbytes	10^3 *Bytes*
kbytes/sec	10^3 *Bytes per Second*
LHS	*Linear Hardware Space*
LSH	*Logical Shift*
LUT	*Look Up Table*
M	10^6
Mbytes	10^6 *Bytes*
Mbytes/sec	10^6 *Bytes per Second*
MAC	*Multiply ACcumulate*
MATRIX	*Multiple Alu Architecture with Reconfigurable Interconnect eXperiment*
MIMD	*Multiple Instruction Multiple Data*
MSB	*Most Significant Bit*
MSE	*Mean Square Error*
MOPS	*Millions of Operations per Second*
MPYI	*Multiply Integer*
MPYI3	*Multiply Integer, 3 Operands*
msec	10^{-3} *Seconds*
N+	*Negative doped semiconductor material*
nsec	10^{-9} *Seconds*
ops/sec	*One-dimensional transform operations per second*
PAL	*Programmable Array Logic*
PAU	*Port Arbitration Unit*
PC	*Personal Computer*
PCB	*Printed Circuit Board*
PCI	*Peripheral Component Interface*
PE	*Processing Element*
PIA	*Programmable Interconnect Architecture*
PLD	*Programmable Logic Device*
PRISM	*Processor Reconfiguration through Instruction-Set Metamorphous*

PROM	*Programmable Read Only Memory*
RACE	*Reconfigurable and Adaptive Computing Environment*
RBinDCT-C1	*Reverse BinDCT operation using Configuration C1*
RBinDCT-C9	*Reverse BinDCT operation using Configuration C9*
RC	*Reconfigurable Cell*
RDCT	*Reverse Discrete Cosine Transform*
RISC	*Reduced Instruction-Set Computer*
RLC	*Reconfigurable Logic Coprocessor*
ROM	*Read Only Memory*
RPM	*Rapid Prototype for Multiprocessors*
RPFD	*Read-Only Protected Flip-flop Device*
RPTS	*Repeat Single*
RPU	*Reconfigurable Processing Unit*
RS	*Reconfigurable Supercomputer*
RTR	*Run Time Reconfiguration*
SIMD	*Single Instruction Multiple Data*
SRAM	*Static Random Access Memory*
SUBC	*Subtract Integer Conditionally*
SUBI	*Subtract Integer*
TM-2	*Transmogifer-2 Prototype Environment*
XC6200DS	*XC6200 Development System*
XC6200ADS	*XC6200 Application Development Software*
μsec	*10^{-6} Seconds*

Chapter 1

Introduction

The growth in complexity and application of image processing algorithms has been reflected by demand for ever-greater computing power. The development of more powerful processor architectures to satisfy this requirement itself encourages further application diversity. This repetitive cycle has been dependant upon the continual advancement of semiconductor technologies and construction of dedicated processing architectures. Existing general-purpose computing solutions whilst rapidly advancing still do not exhibit the necessary processing power required for many applications.

High-powered computing architectures have traditionally been constructed using two fabrication methods. The first method requires the development of *Application-Specific Integrated Circuits* (ASICs) to implement whole or part of the target algorithm directly in hardware. This process incurred high devolvement costs but generated the most efficient implementation, particularly for volume production, since the architecture was designed to accelerate a specific task.

The second construction method used multiple commercial instruction-set based processors operating concurrently. Within such architectures, individual processors typically communicated using fixed interconnection topologies. In comparison to ASIC construction, this method resulted in less efficient implementation of the application, but with reduced development costs.

Though both construction techniques provided high performance computing compared to general-purpose processing platforms, the flexibility and versatility of system operation was impeded through the construction methods used. Application diversity of the system was compromised to achieve high operand throughput. The computation of non-target algorithms was therefore inefficient, if at all feasible.

To merge the versatility of a general purpose processing architecture with the high performance of an application specific hardware, processor hardware must adapt and accelerate each specific task. In existing fixed processing topologies, the ability to optimise the architecture for each different application was restricted by the hardwired nature of the processing and routing resources. If these components were reconfigured for each application, the mapping efficiency, hence operand throughput would increase.

Through the development of programmable logic technology, the concept of high performance multipurpose computing architectures has been realised. Such architectures contain high performance processing resources within a common architecture that can be adapted to accelerate different algorithmic structures.

Construction of custom computing machines using this technique has revealed new operation taxonomies that can achieve ever more efficient hardware implementations. Efficient hardware design increases system throughput whilst reducing power consumption.

Many image-processing applications contain both primary and secondary processing operations. Within existing high performance computing architectures, the configuration of internal resources are fixed during the computation to accelerate the primary function. In applications where multiple functions occur, system throughput would be degraded by inefficient hardware implementations of the secondary functions.

One solution to this problem has been the continual development of larger semiconductors, with logic capacities that enable all functions within an application to be implemented efficiently. Eventually limitations within present-day semiconductor fabrication techniques will be reached restricting the development of higher capacity devices.

A different approach to increasing efficiency has been to examine the operating cycle of an application and partition the resultant hardware design into time independent

segments. This concept is known as *temporal partitioning*. Each temporal partition equates to a portion of the overall system operation accelerated through efficient hardware implementation.

To compute an application individual temporal partitions are activated as demanded by operational flow. Each successive partition reuses hardware resources that implemented the previous active partition. This concept of reusing logic within sequential temporal functions is known as *spatial partitioning*, and provides a hardware efficient approach to implementing high-performance computing architectures.

The realisation of the full performance benefits that this technology can provide is not yet apparent. Technological advances in programmable logic technology must occur prior to acceptance of temporal and spatial design implementation techniques within industrial applications. This interest however will only be generated through academia developing applications in which performance benefits occur through dynamic implementation. This is a symbiotic relationship since for the technology to mature industrial acceptance is required.

The technological contribution made through the research presented in this thesis has been the integration of dynamic hardware components within existing high performance multiprocessor topologies. This has been conducted to determine how the operation of each technology can be advanced, and to develop dynamic hardware applications.

The outcome of this work has been the development of dynamic coprocessor functions and communication routing hub topologies within a multiprocessor environment. A novel implementation of the BinDCT algorithm has also been developed. To conduct this work an industry standard (TIM-40) DSP MIMD parallel processing architecture has been upgraded, in conjunction with the development of a custom designed dynamic hardware prototype environment (XC6200DS) and associated software tools (XC6200ADS).

From the onset of the investigation, it was apparent that the operating characteristics of the dynamic semiconductor technology available (Xilinx XC6200 FPGA family) were inferior compared to existing hardwired technologies. The development of hardware throughout the project has therefore addressed dynamic implementation techniques, issues and operating concepts, rather than obtaining raw throughput. Newer emerging devices, naturally faster and with more computing power are however not yet run-time reconfigurable. This makes the Xilinx XC6200 family unique, and allowed real hardware to be investigated whilst researching dynamically reconfigurable architectures.

The thesis presented consists of nine chapters including this chapter, the *Introduction.* To provide a knowledge base on which to digest the concepts explored within this project, *Chapter-2* introduces the concept, history and development of configurable technology. The evolution of adaptive machine topologies and operating characteristics are then introduced, with examples given. Prominent examples of each classification are described, with more detail provided in *Appendix-II. Chapter-2* concludes by describing the present status of configurable logic technology, its limitations and future research directions.

To investigate merging dynamic hardware within a parallel processing environment, a *XC6200 FPGA Development System* (XC6200DS) was designed and constructed since no suitable tool was commercially available. To manage XC6200DS operation, perform in-circuit hardware verification and dynamic configuration data generation, a suite of custom software tools known as *XC6200 Application Development Software* (XC6200ADS) was constructed.

Chapter-3 describes the operation, construction, integration and configuration of both the XC6200DS and TIM-40 systems. The aim of this chapter has been to provide the reader within an insight into the function of XC6200ADS hardware development tools and XC6200DS configuration modes used during the development of the dynamic hardware applications presented later. Operational summaries of key components are

given with detailed explanations included within *Appendix-III*. Development cycles for both the parallel processing and dynamic hardware environments are explained.

Prior to the construction of complex hardware structures, the operating and performance characteristics of the XC6200DS had to be evaluated. This task is presented in *Chapter-4* and details the implementation of fundamental processing hardware within the XC6200 FPGA. The suitability of each function for XC6200 hardware implementation was assessed, including design techniques for temporal and spatial partitioning. The implementation strategies devised in *Chapter-4* were then applied during the development of dynamic hardware applications within *Chapter-6* and *Chapter-7*.

The development of a dynamic coprocessor configuration (*Chapter-3*) enabled the construction of a novel method of *Discreet Cosine Transform* (DCT) computation using the BinDCT algorithm. *Chapter-5* describes the operation of this algorithm with respect to the implementation of a traditional DCT and Chen's *Fast DCT* (FDCT) algorithm. The chapter describes experiments conducted to determine the suitability of the BinDCT for dynamic operation and how system operation was enhanced using run-time reconfiguration for one and two-dimensional BinDCT transforms. Presented within *Chapter-5* are dynamic BinDCT software simulated results, which are compared against XC6200 FPGA generated dynamic BinDCT hardware results in *Chapter-6*.

XC6200 FPGA implementations of dynamic BinDCT TMS320C40 DSP coprocessors are described in *Chapter-6*. This discussion describes the initial static hardware implementation methods used, incorporation of BinDCT hardware within the XC6200DS coprocessor configuration, and the inclusion of a custom dynamic configuration mechanism known as the self-configuration controller (described in *Section-3.4.4*). Hardware results presented demonstrate the advantages temporally partitioned hardware, reconfigured using dynamic configuration can provide.

The insertion of dynamic hardware within the TIM-40 communication topology has permitted the investigation of a multiple-purpose routing hub. *Chapter-7* describes how

the throughput of an existing multiprocessor architecture can be improved through incorporation of this technology. This concept is further expanded on by combining the computation of simple functions within the transfer of operands between system nodes. Within *Chapter-7*, the implementation of a Roberts Cross Edge Detector is described, as well as the construction and operation of the routing hub.

A summary of the content and specific conclusions for each task has been provided within each chapter. Resulting conclusions for the research presented are described in *Chapter-8*. Topics covered include the status of configurable technology and the contributions to the research field this project has made. These contributions include novel application development, and integration of dynamic hardware within multiprocessor architectures.

From these conclusions, recommendations for further work have been determined. These are described in *Chapter-9*. Improvements to the XC6200DS and XC6200ADS are suggested as well as addressing dynamic hardware and in-circuit verification strategies. Ideas for the integration of configurable hardware within commercial products are also discussed.

Chapter 2

Reconfigurable Computing Technology

Introduction

The aim of this chapter is to provide a sound knowledge base of reconfigurable computing techniques, architecture classification, and applications. *Section-2.1* introduces the basic concept of configurable computing, and details the advantages gained. *Section-2.2* builds on this with an example of the first configurable computing system devised, and explains how configurable computing architectures evolved through advances made in semiconductor technology.

Section-2.3 describes in detail the configuration mechanisms and performance issues of existing configurable architectures. *Section-2.4* discusses the classification of configurable machine architecture and related taxonomies, with *Section-2.5* describing systems applications and operational characteristics of each type. Detailed overviews of prominent architectures discussed are included in *Appendix-II*. *Section-2.6* reviews current research trends and the present status of the technology.

2.1 Configurable Computing Introduction

By their very nature image digital signal-processing algorithms are computationally intensive. To achieve high operand throughput, inherent concurrent operations within a task must be fully exploited. Until recently, high performance image-processing applications could only be achieved through using dedicated custom computing hardware known as *application-specific architectures* (ASAs), designed specifically to accelerate and compute a given task. ASAs effectively mimic the structure of an algorithm within hardware. This enables an ASA implementation to exhibit greater throughput and efficiency when compared to a general-purpose processing architecture.

Traditionally, ASAs have been constructed using either *application-specific integrated circuits* (ASICs) that implemented the whole or part of the target algorithm directly in

hardware [1], or by use of multiple specialised instruction-set based processors, communicating via a fixed interconnection topology [2]. Though such implementations provide high-performance solutions, the versatility of the system application was limited when computing an algorithm of different structure, if at all feasible. Application-specific architectures also suffered from high development costs compared to commercial processing engines, since the architecture and composite components were normally designed from scratch and constructed in small quantities. Historically however, these were the only solution for many high-performance applications.

Image-processing functions can be classified as being local or global type operations. Local operations consist of a large number of simple highly concurrent calculations such as binary thresholds, which are most suited for computation using an array of fine-grain *processing elements* (PEs). Global operations consist of fewer but more complex functions, exhibiting less inherent concurrent operations. Typically these operators consist of trigonometric mathematical functions, computed using CORDIC (*Co-Ordinate Rotation DIgital Computer*) [3] [4] based processing architectures. Within CORDIC calculations, functions such as *multiply and accumulates* (MACs) occur.

Local operators are normally used for image pre-processing, whereas global functions are used for extracting embedded information. Most image-processing applications normally consist of both types of operation. To merge ASA performance with the system flexibility of a general-purpose computing architecture, optimisation of the processing architecture for each application was required. The adaptation of a processing architecture during run-time, to accelerate each phase within an application, is the key concept within reconfigurable computing. This is illustrated in Figure 2.1.

As computer applications continue to grow in complexity, the requirement for more powerful processing architectures is ever present. This demand has been realised through the development of processor architecture technology, and through advances in semiconductor fabrication techniques that have enabled clock frequencies to increase. Traditional processing architectures function using a fixed instruction-set, with the

instructions providing limited concurrent execution. The efficiency with which such architectures can utilise parallel operations occurring within an application is therefore restricted. By increasing the clock frequency, the throughput of the architecture can be improved but not the efficiency by which concurrent functions are exploited. Alternative methods to achieve this are, inclusion of additional concurrent processing hardware within the architecture, or through reorganising existing resources.

For efficient exploitation of parallelism within multiple tasks, a processing architecture must be optimised through reconfiguration on demand. This area of processor architecture research is relatively new, therefore both the hardware architectures, and software development tools are still in their infancy, and require much more investigation to stimulate further development.

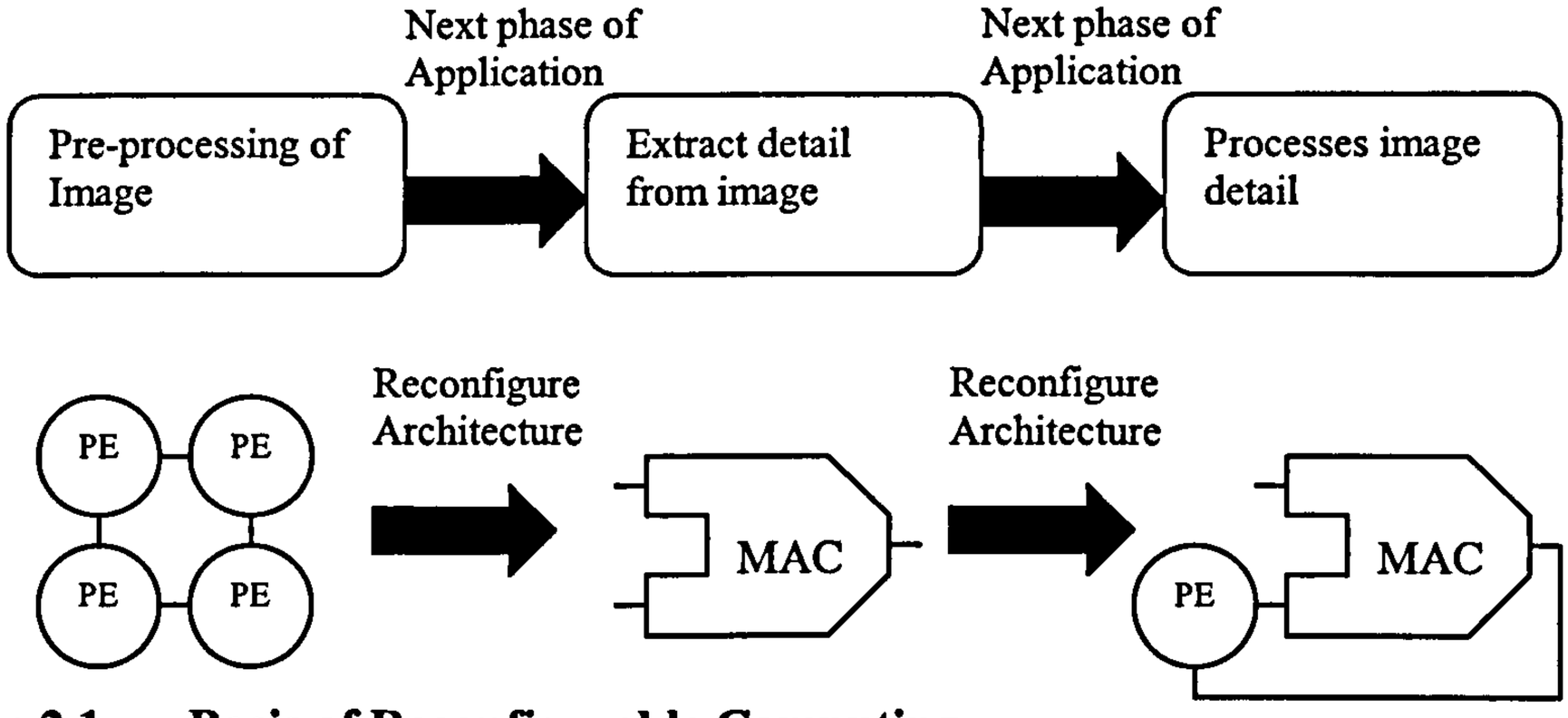


Figure 2.1 Basis of Reconfigurable Computing

2.2 Evolution Of Reconfigurable Computing

The ability to reorganize the structure of a processing architecture improves the task diversity and operating characteristics of the system. The development of such operating techniques however has been hindered through the unavailability of suitable implementation fabrics. These limitations have imposed operating constraints upon such architectures. The evolution of both configurable computing concepts and programmable semiconductor technologies are therefore linked and continue to influence each other.

2.2.1 The First Notion

G. Esterin of the University of California at Los Angeles proposed the first notion of this idea in 1959 [5]. Esterin's concept was of a *fixed-plus-variable* (F+V) computing architecture that would provide high performance and application diversity. Figure 2.2 shows a block diagram of the F+V architecture. The design consisted of a fixed general-purpose *central processor unit* (CPU) known as the F-Unit, tightly coupled to a coprocessor (V-Unit) configured for each application. A supervisory control unit governed interaction between the two units.

Esterin intended the V-Unit to be configured during system operation using electro-mechanical relays, which selected and activated circuit cards comprising application specific processing hardware. These circuit cards could also be manually replaced whilst the processor was inactive. Functions such as vector arithmetic and hyperbolic operations could be performed within this unit.

The limitations of early electronic technologies made it difficult to implement such architectures. The concept of configurable computing technology has therefore only become practically viable during the last fifteen years, through the introduction and development of in-circuit *programmable logic devices* (PLDs), primarily *Field Programmable Gate Arrays* (FPGAs). The subsequent technological development of these devices and that of configurable computing techniques have influenced each other.

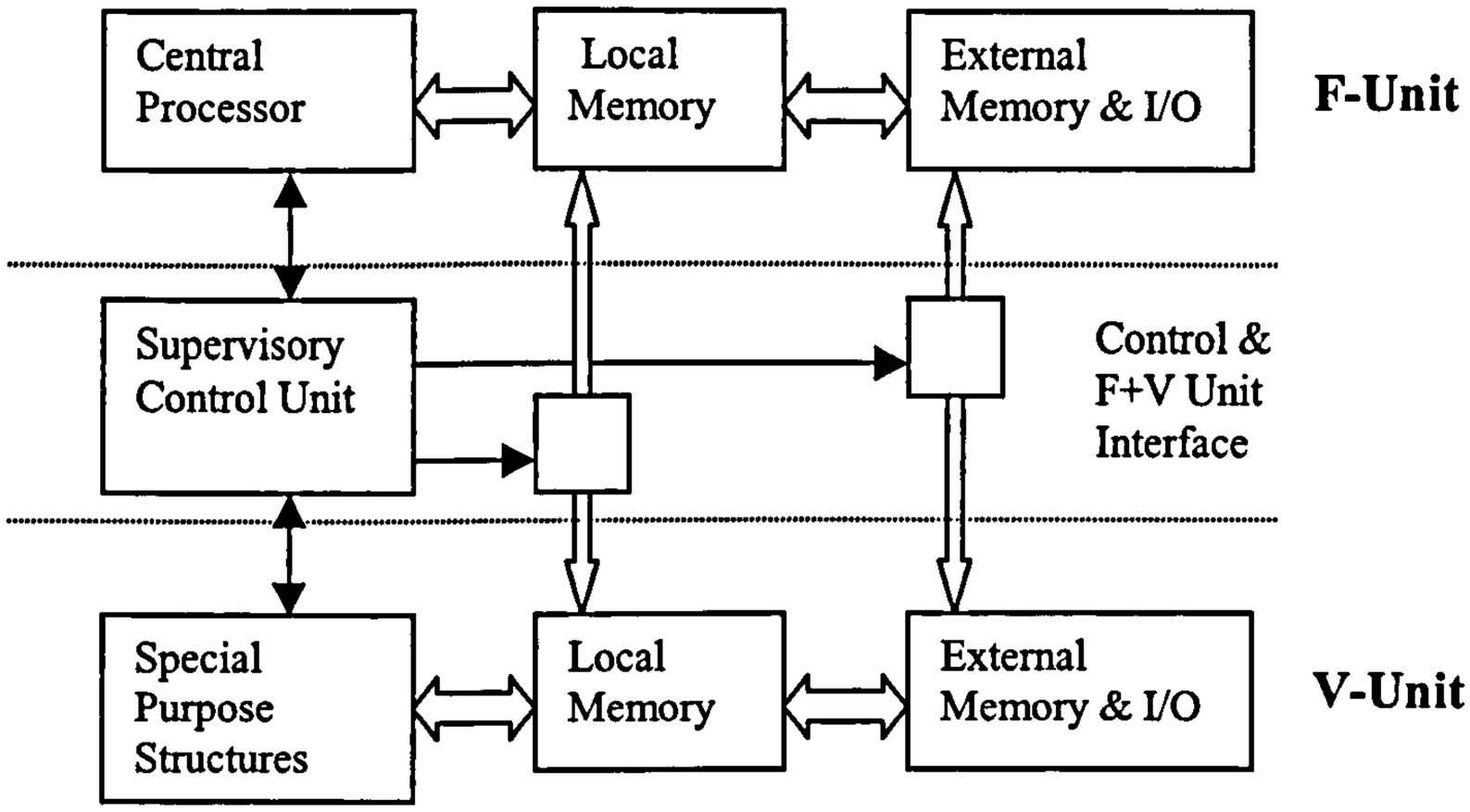


Figure 2.2 Esterin's Fixed Plus Variable Computing Architecture

2.2.2 FPGA Technology Review

An FPGA is a silicon chip in which the user determines the function. In 1986 Xilinx introduced the first FPGA (XC2000 family). FPGAs were developed because existing programmable logic called *complex programmable logic devices* (CPLDs) could not support the ever-increasing demand for greater on-chip logic capacity.

The fundamental problem within CPLD architectures was that the ratio of sequential logic resources (flip-flops) compared to combinatorial logic (logic gates) was small and insufficient for many tasks. With a typical CPLD such as the Vantis MACH111 [6], this ratio was one flip-flop to twelve product terms (two-input Boolean expression). This limitation can be accredited to the underlying architecture of a CPLD, in which logic functions were configured within multiple *programmable array logic* (PAL) units, interconnected via a *programmable interconnect architecture* (PIA). The relation of these components within a typical CPLD (MACH111) is shown in Figure 2.3. The MACH111 consists of two PALs interconnected using a PIA. A MACH111 PAL consists of sixteen macro-cells, each cell containing a programmable AND, fixed OR array matrix, and one flip-flop. Limited sequential logic resources within CPLD

architectures therefore hindered the migration to fabricating complex processing architectures within programmable devices.

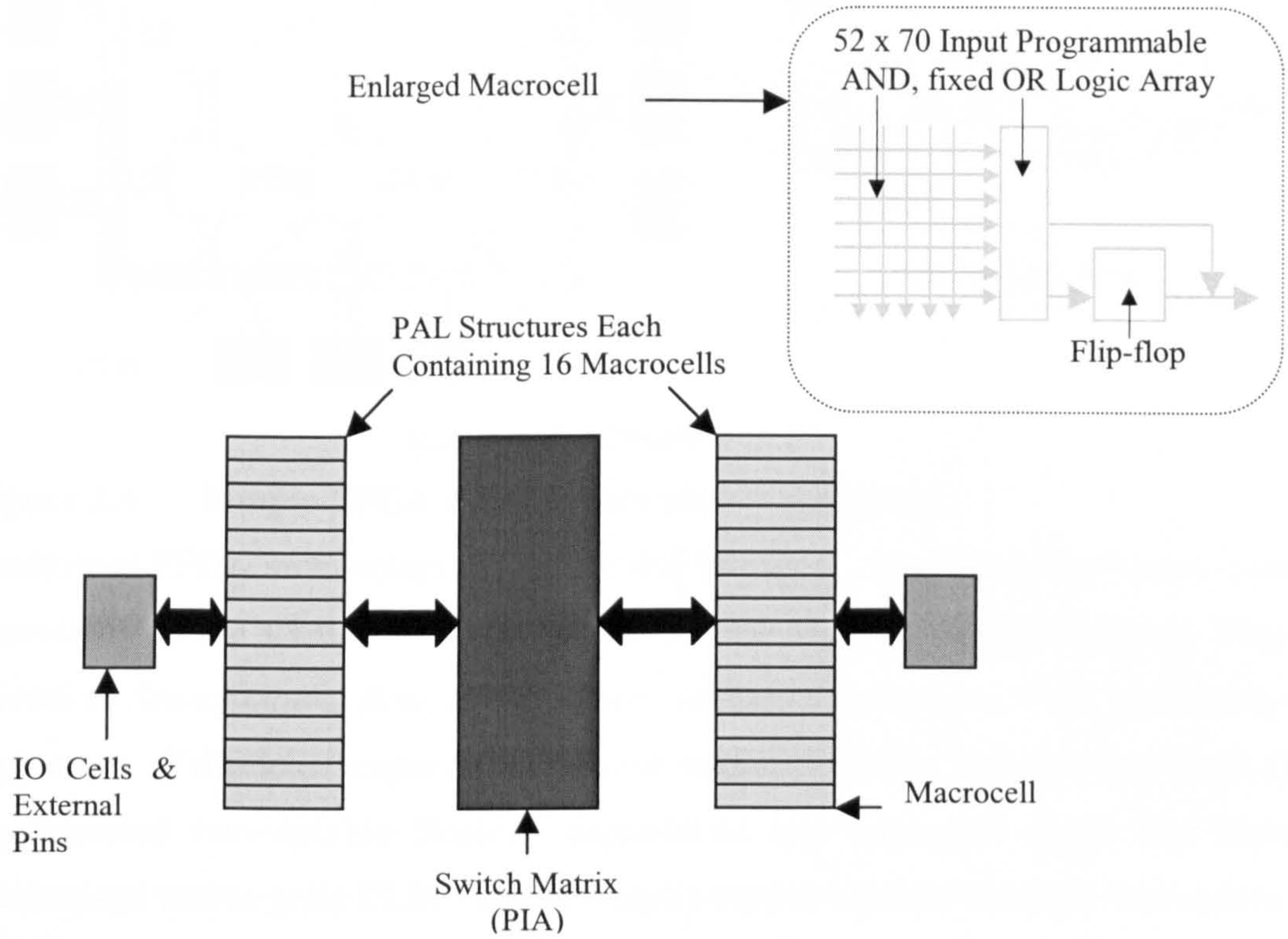


Figure 2.3 Simplified MACH111 CPLD Architecture

The basic outline architecture of an FPGA is shown in Figure 2.4. The architecture consists of an array of *configurable logic blocks* (CLBs), a programmable interconnection matrix, and *input/output blocks* (IOBs) connected to external pins on the chip carrier. Instead of using PALs, FPGA logic is implemented within CLBs. A basic CLB implements combinatorial logic using multiplexers and *look-up tables* (LUTs), and contains one or more flip-flop devices to implement sequential logic elements.

In Figure 2.4 the inputs to the CLB are noted as *A, B, C,* and *Clock*, with the output labelled *F*. The gate capacity of a CLB is normally less than that of a CPLD PAL. For example a CLB could implement up to three four-variable Boolean expressions [9] compared to the PALs twelve product terms [6]. Designs implemented upon an FPGA must be partitioned into a far greater number of logic elements than that implemented upon a CPLD. FPGAs therefore require and indeed have far greater routing resources than that of CPLDs.

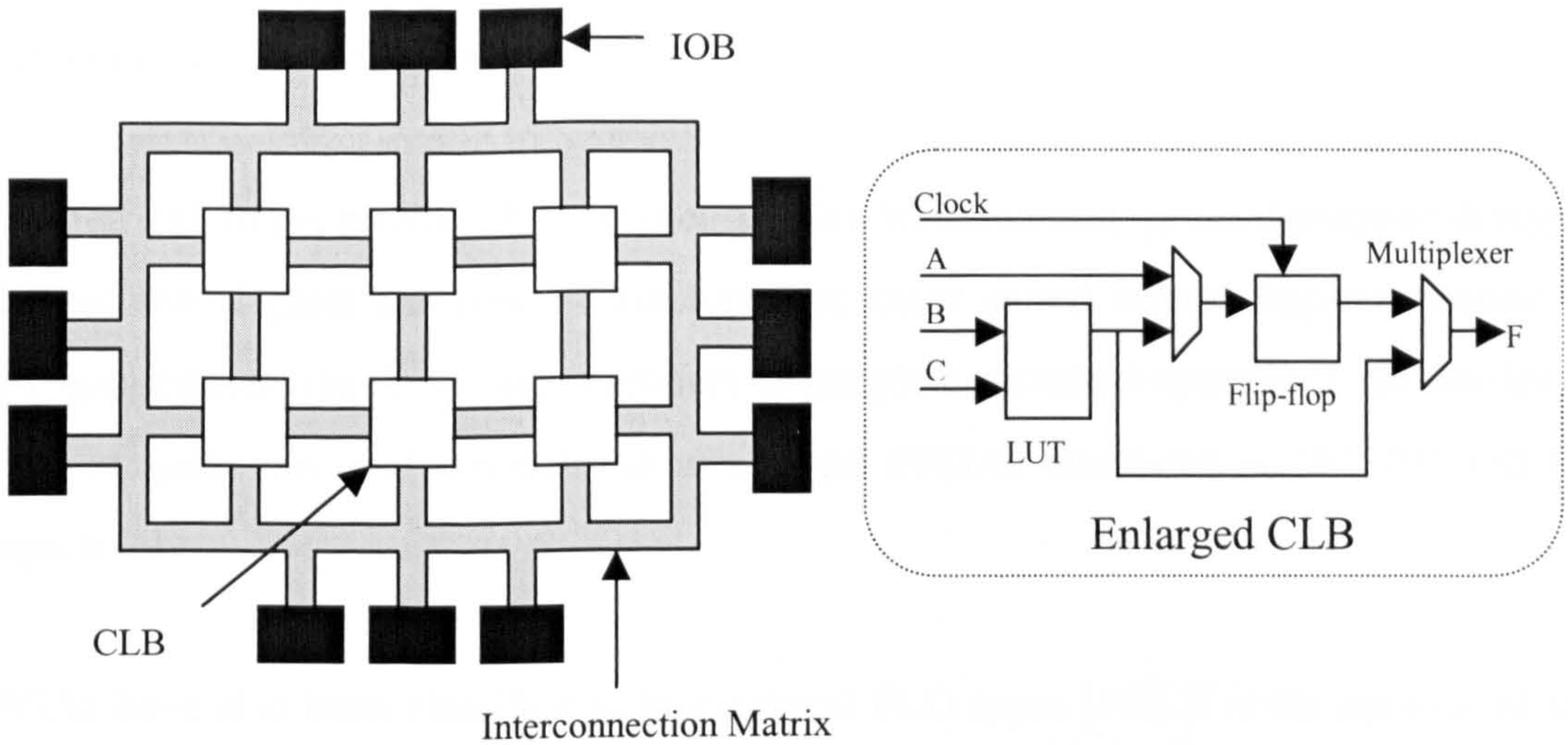


Figure 2.4 Simple FPGA Architecture and Components

Traditional FPGA architecture can be divided into three main categories based upon the granularity of the CLB and the complexity of the internal routing structure. They are known as *Sea-of-Gates, Row,* and *Symmetrical* type architectures. CLB granularity was a measure of the logic capacity configured within a device. Typical fine grain CLBs implemented two-variable Boolean expressions and contained single flip-flops. In comparison coarse grain CLBs typically implemented two four-variable expressions and contained two flip-flops.

Sea-of-gate FPGA architectures consist of fine-grain CLBs interconnected via an extensive local routing structure. If the CLB granularity is fine, a design has to be partitioned into a greater number of simple logic blocks. This means that neighbouring CLBs rely upon extensive local signal routing to share product results.

Row architectures consist of coarser CLBs, possessing local and global routing resources. With a coarse-grain FPGA, the design is partitioned into fewer but more complex logic blocks. CLBs would not require extensive local signal routing since the number of product terms shared by neighbouring CLBs would be reduced. Instead they require dedicated longer chip-wide routing resources to share product functions.

Symmetrical arrays have the coarsest CLB granularity and contain extensive chip-wide routing resources. Within the FPGA architectures a trade-off exists between local

routing capacity and CLB granularity.

Symmetrical arrays provide the best medium for implementing general-purpose designs, whereas sea-of-gates and row architectures are more suited for the implementation of DSP applications requiring large numbers of simple concurrent operations. Examples of a sea of gate, row and symmetrical array type FPGAs are listed in [8], [9] and [7] respectively.

FPGAs have also been classified as hierarchical PLD types [10]. It is the opinion of the author however, that devices of this category are CPLD hierarchical architectures incorporating aspects of FPGA technology.

Traditionally FPGAs functioned using *static random accesses memory* (SRAM) programmable technology, whereas existing CPLDs incorporated floating-gate technologies similar to that used in *erasable programmable read only memory* (EPROM) and e*lectrically EPROM* (EEPROM) technology. SRAM has the advantage over floating-gate technologies in that configuration times are reduced from seconds to milliseconds. Data written to the FPGAs configuration SRAM determines the configuration of the CLBs, IOBs, and programmable interconnection network. A disadvantage of SRAM technology is that it is volatile and upon power-up, SRAM based FPGAs must download configuration data from a external source (typically a PROM).

FPGAs have also been developed using one-time programmable non-volatile programmable technologies. The Act3-PCI FPGA [9] family manufactured by Actel is an example of such a device, and incorporates anti-fuse instead of SRAM programming technology. A comparison of PLD configuration technologies is detailed in *Appendix-I.*

The design process used in the development of an FPGA application is shown in Figure 2.5. The first stage of the process *(1)* requires generating the design entity using *hardware description languages* (HDL) such as VHDL and Verilog, or by using

schematic entry tools such as View Draw. The function of the design can be simulated and verified using software design tools, which enables the development cycle of the hardware to be conducted within software. The use of such tools allows errors be detected and corrected *(2)* before programming a PROM. Once the design has been proved correct the FPGAs configuration data can then be generated *(3)*. This is then programmed within a PROM *(4)* from which the FPGA downloads its configuration data upon system power-up *(5)*. The function of the FPGA can then be tested and verified in-circuit using the FPGAs JTAG [79] interface *(6)* (if applicable).

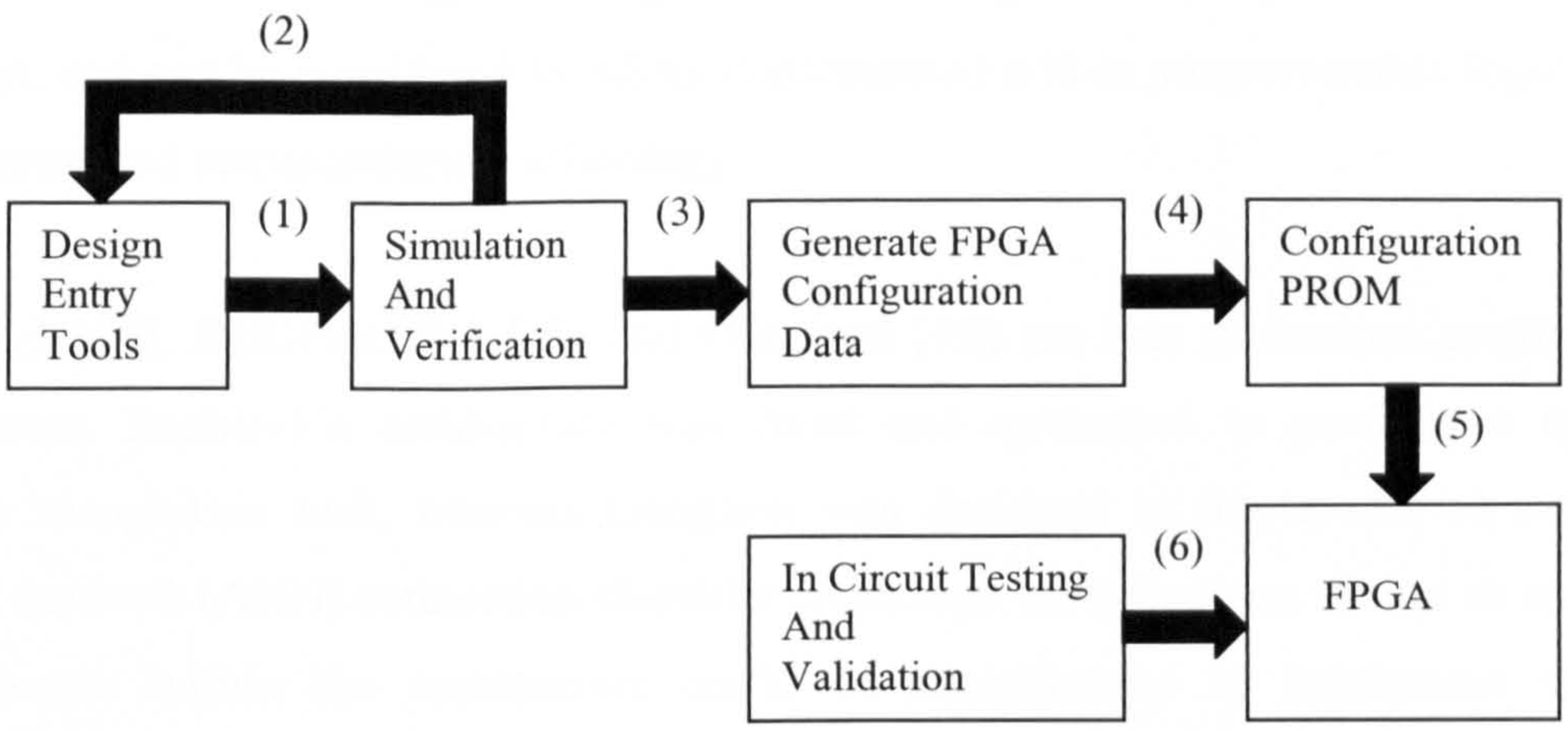


Figure 2.5 FPGA Application Development Cycle

Until recently the concept of user-programmable ICs has been restricted to the domain of synchronous digital logic. Motorola have developed a programmable analogue device known as a *field programmable analogue array* (FPAA) [11], whilst I-Cube have developed programmable switches known as *field programmable interconnect devices* (FPIDs) [12]. These devices utilise SRAM programming technology and exhibit similar configuration characteristics to traditional FPGAs.

Academic research groups have also developed FPGA type architectures for use in asynchronous digital logic applications. An example of an asynchronous FPGA is Montage [13]. Work has also been conducted to investigate the use of optical configuration mechanisms in FPGAs, rather than SRAM to reduce configuration delays [14]. The Virtual Wires project [15], has also addressed the issue of limited IOB

bandwidth caused by inadequate chip carrier pin resources through multiplexing multiple signals upon each pin.

2.2.1 First Generation Machines *(circa 1987-1993)*

FPGAs were initially developed for use as reusable prototype devices to reduce development costs of digital hardware. Through continual improvements in FPGA technology, it became evident that larger capacity FPGAs (e.g. Xilinx XC3000 family 1987, logic capacity 1000-6000 gates) [16]) could be used as alternatives to hardwired ICs within the final design. First generation configurable computers reflected this concept, and can be considered as ASAs implemented within programmable logic rather than hardwired semiconductor technology.

Enable-1 [17], DECPeRLE-1 [18] and Ganglion [19] are first generation configurable computers. Enable-1's architecture was fixed and optimised to perform a specific pattern recognition task, whereas Ganglion was designed to implement an *artificial neural network* (ANN) connection classifier. Although Ganglion was design as an ASA, components within the architecture could be reconfigured to implement specific weights, bias values and scaling parameters rather than use generic values for each application.

DECPeRLE-1 can also be considered first generation architecture, although not for the same reasons as Ganglion or Enable++. DECPeRLE-1 design was more generalised than an ASA and allowed greater adaptation of system architecture, hence facilitated broader application diversity. Applications including cryptography, stereovision, and neural networks highlighted the potential benefits configurable computing could offer.

2.2.4 Second Generation Machines *(circa 1993-1996)*

As FPGA technology developed, the capacity, structure and performance of devices improved enabling the concept of configurable computing to evolve. For example the Xilinx XC4000XL [9] and Altera Flex10K [10] series had gate capacities of up to 180,000 and 250,000 gates respectively. Through the development and evaluation of first generation architectures, coupled with the enhanced FPGAs architecture, four distinct types of configurable architectures began to emerge. These were *Prototype Environments*, *Configurable Supercomputers*, *Configurable Coprocessors*, and *Configurable Instruction-Set* architectures.

During system prototyping, a single FPGA could only support a limited volume of logic. By coupling multiple FPGAs together more complex designs could be implemented. To facilitate prototyping, such architectures were designed for flexibility and not high performance. Examples of this type of architecture were Transmogrifier-1 [20] and Springbok [21]. Transmogrifier-1 could be used to directly implement ASIC logic designs of up to 40,000 gates. Springbok however used configurable logic to provide signal routing between hardwired ICs implementing the prototype design. A further example was the *Rapid Prototype engine for Multiprocessors* (RPM) [22], designed specifically to emulate MIMD processor architectures.

Configurable supercomputers emerged that provided the high performance of an ASA, with the versatility of a general-purpose architecture. This was possible through the sheer scale of their configurable resources (typically hundreds of thousands of gates). The most prominent examples of such architectures were the Virtual Computer [23] and Splash-2 [24]. Both machines could be configured and optimised for different tasks, therefore exploiting the concurrent properties of each algorithm implemented.

Traditional instruction-set architectures also incorporated reconfigurable hardware through using FPGA based coprocessors. Examples of such architectures were Garp [25] and Harp [26]. Garp consisting of a processor and configurable array tightly

coupled upon a custom silicon die, whereas Harp consisted of a Transputer and commercial FPGA. In both architectures, application computation was partitioned between the two processing resources. The coprocessor performed instructions implemented inefficiently or not present within the main processor. This feature allowed optimization of a common architecture to accelerate different applications.

A different concept developed instruction-set architectures, where only the instructions actually used during the computation were configured. Early examples of this type of architecture were PRISM [27] and the Nano Processor [28]. Both devices functioned using fixed processing-core skeletons upon which optimal instruction-sets were configured.

2.2.5 Virtual Hardware

The development of FPGA technology was continuous, however the limited logic capacities of devices still restricted application development. Such limitations were more apparent in coprocessor and configurable instruction-set architectures as they contained far fewer reconfigurable logic resources, which restricted the number of custom instructions implemented concurrently.

The serial nature of instruction-set micro-coded operation implied that a small portion of the instruction-set would be active at any given moment. Logic resources implementing inactive instructions could therefore be reconfigured with active instructions. This feature provided *virtual instruction-set capacity* and *virtual hardware* capabilities [29].

The operation of virtual hardware can be compared to virtual memory within modern computers. Virtual memory implies that a computer possesses more memory than physically present in the system. Swapping data from the hard-drive to the physical memory only when required in the computation, and then transferring it back to hard-drive when inactive achieves this. In virtual hardware, instead of data being transferred

to and from memory, hardware configurations resident in a configuration store are swapped to and from a device during system operation. Therefore the device appears to exhibit greater hardware capacity than physically present. This idea is illustrated in Figure 2.6. The principal of virtual hardware has also been labelled *Multiple Context Configuration* [30] and *Cache Logic* [31].

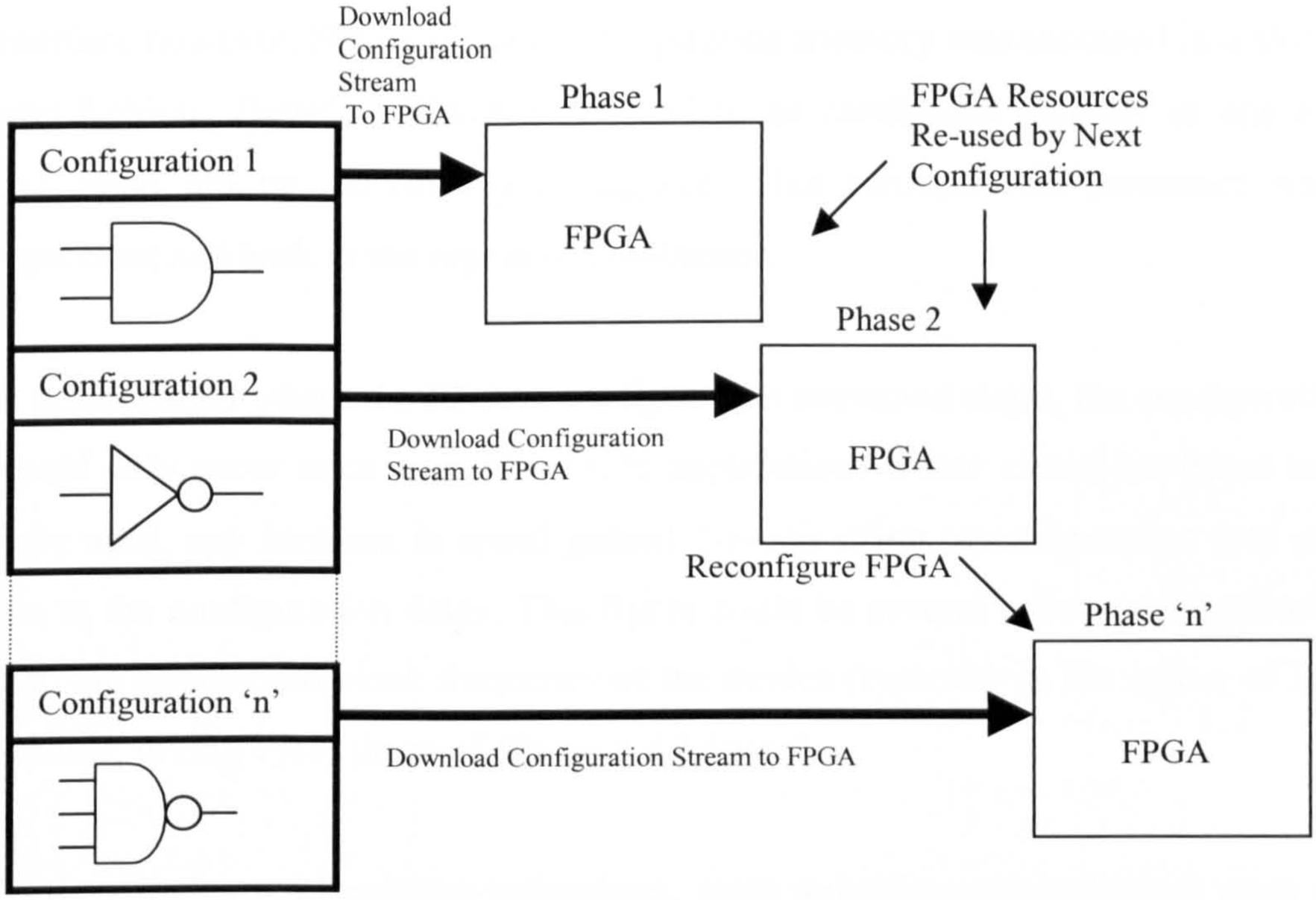


Figure 2.6 Concept of Virtual Hardware

To incorporate virtual hardware an FPGA's configuration must be updated concurrent to system operation. This was impractical with existing FPGAs such as Xilinx XC4000 family configured prior to start-up, therefore a new generation of FPGA devices both custom and commercial were developed. These new FPGA architectures reflected the change in FPGA applications from implementing static hardware, to use in configurable computing applications.

2.2.5 Dynamic FPGA Technology

Within traditional FPGA architectures the design emphasis placed upon the configuration mechanism was to minimise the number of chip-carrier pins used during the configuration process. In early FPGAs, configuration data was downloaded in a serial fashion, with later devices incorporated parallel interfaces. With both types of interface however, FPGA internal configuration memory was accessed in a shift-register type fashion. Therefore FPGA CLBs had to be configured together in one operation, and could not be individually configured. This configuration processes was device dependant and took in the region of 20-40msec.

In applications where the FPGAs configuration remained static, the configuration delay would only occur once on power-up. In applications where virtual hardware techniques were used, any increase in speed gained through using reconfiguration was eradicated due to the configuration delay. This figure could be several orders of magnitude greater than the operational clock frequency of the device (typically in the region of 20MHz to 80MHz, giving cycle times of 50nsec to12.5nsec).

To develop virtual hardware technology, more suitable semiconductors were required. The device configuration delay had to be reduced, and the granularity of the configurable logic cells structure improved. Since none were commercial available, academic research groups began to develop their own configurable computing architectures such as Matrix [33] and the DPGA *(Dynamically Programmable Gate Array*) [34].

Matrix (*Multiple ALU Architecture with Reconfigurable Interconnect eXperiment*) addressed the problem of inefficient CLBs. Matrix contained coarse-grained configurable units that could implement high-level operations such as multiplication, rather than simple Boolean expressions. The DPGA architecture primarily addressed the concept of virtual hardware. The DPGA could be reconfigured in one clock cycle (9nsec) by switching between configurations stored in a four-deep multiple context

configuration memory. Villasenor and Hutchings [35] however, indicated that the first research into multiplexing FPGA configurations was actually conducted by Xilinx in 1991, but remained the intellectual property of Xilinx until publication in 1997 [36].

In 1995 Xilinx introduced the XC6200 family of FPGAs [32] (formally Algotronix CAL series). These devices were designed specifically for use in configurable computing applications, and incorporated *partial* and *dynamic* configuration. Partial configuration enabled only specific areas of the FPGA to be configured. The advantage of this technique was the volume of configuration data required to reconfigure the FPGA was kept to a minimum. This reduced the configuration delay, which was proportional to the volume of configuration data.

Dynamic configuration was the ability to partially configure an FPGA without halting the operation of unaffected areas of the device. Within the XC6200 architecture partial and dynamic configuration was made possible through a parallel interface known as the FastMap™ illustrated in Figure 2.7. Configuration data was written during run-time using the address and data-buses. The address was interpreted by the FastMap™ interface as a pointer to the row and column location of a resource within the *Configurable Logic Cell* (CLC) array; CLC was the term given to XC6200 equivalent of traditional FPGA CLBs. The new configuration was then written from the data-bus to the configuration memory. This architecture is discussed in greater detail in *Appendix-III*.

The XC6200 architecture provided a structure to implement virtual hardware. The configuration mechanism employed was not ideal since the configuration delay was proportional to the volume of configuration data (approximately 30nsec per CLC). In comparison, the DPGA could be totally reconfigured in 9nsec by switching between memory contexts.

Unlike the XC6200 the DPGA did not support partial configuration. The capacity of the DPGAs context memory restricted the number of configurations to four [30]. In

comparison the XC6200 could support an unlimited number of configurations using partial configuration techniques. For the work presented in this thesis, XC6200 devices have been used for the dynamically reconfigurable elements, as they were the only suitable devices commercially available at the time.

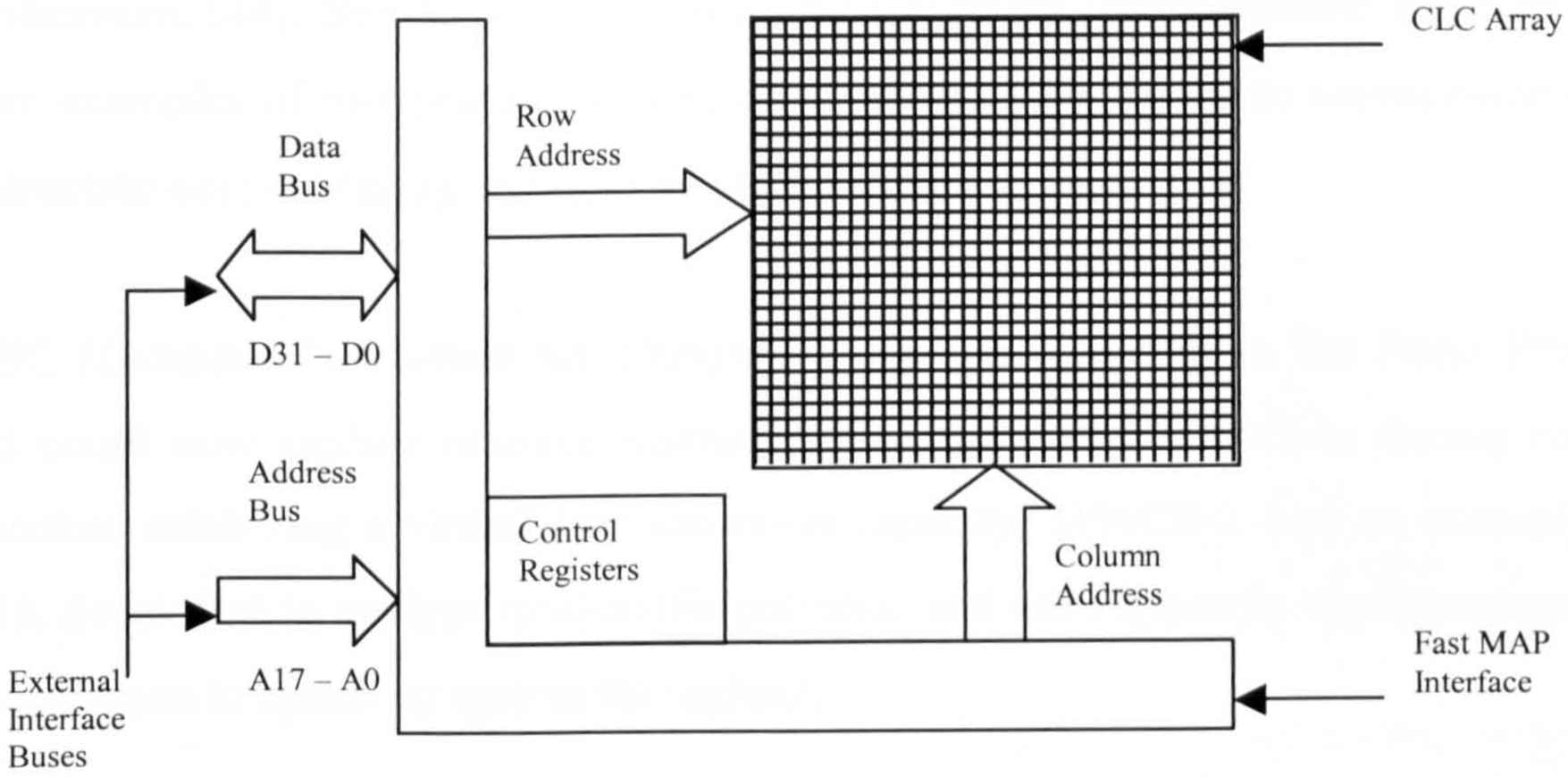


Figure 2.7 XC6200 FPGA FastMap™ Interface

Further prototype FPGAs developed were Atmel AT6000 family [31] and the National Semiconductor CLAy FPGA series [37]. All three devices were intended to be commercial products but only ever produced in small quantities, with restricted availability.

This can be attributed to the reluctance by industry to implement run-time adaptive architectures. Experiences gained through the development of the XC6200 architecture however, have been reflected within Xilinx Virtex family of FPGAs [38], their current flagship devices.

2.2.6 Third Generation Machines *(circa 1996-present date)*

The introduction of the XC6200 family as well as the Atmel and National Semiconductor dynamic FPGAs fuelled further research into configurable computing techniques. Primarily, these advances were based upon the incorporation of virtual hardware within existing computing architectures. Examples of such architectures were

Hades [39], Chimaera [40], Space-2 [41], DISC [42], and Morphosys [43].

Hades was an example of a dynamic coprocessor. This project involved coupling a Xilinx XC6200 FPGA to a custom instruction-set processing architecture. A further outcome of this project was the development of a suite of design tools for dynamic applications [44]. Similarly, Morphosys and Chimaera reconfigurable functional units were examples of merging a tightly coupled run-time configurable coprocessor with an instruction-set processing architecture upon a common silicon die.

DISC *(Dynamic Instruction Set Computer*) was the successor to the Nano Processor, and could now replace inactive instructions with active instructions during run-time, therefore exhibiting a virtual instruction-set capacity. SPACE-2 was an example of an ASA developed to analyse road-traffic patterns, and used dynamic configuration within its operation to speed-up system throughput.

2.2.8 Commercial Machines

The transfer of configurable technology from academia to industry has recently become apparent at system, programmable fabric, and development tool levels. At system level Triscend developed the first instruction-set processor incorporating a user-defined instruction-set, configured and optimised for each individual application [45]. Star Bridge Systems have also developed a dynamic configurable processing architecture called HAL-300GRW1 [46]. When first introduced in 1998, for certain applications HAL-300GRW1 could exceed the performance of the most powerful computer at the time, IBM's Pacific Blue.

Recently, the prospect of commercially available processor cores coupled to reconfigurable logic has become apparent. Xilinx and IBM have announced that they are working together to couple a PowerPC processor with a Virtex FPGA [47], whilst Altera are integrating ARM and MIPS processor architectures within its FLEX family of FPGAs. This new device will be known as Excalibur [48].

Commercial reconfigurable application development software is also under development. At Synopsys there is a project to determine the best method of including reconfigurable logic within their existing design flow tools. Celoxica (formally Embedded Solutions) have also developed software tools that can generate either configuration data for an FPGA implementation or binary code for a microprocessor using an initial HDL Handel-C design [61].

2.3 Configurable Computing Performance

Existing configurable computing architectures vary extensively. To evaluate and classify configurable computing machines, the operational characteristics of the programmable logic used, system configuration mechanics and granularity of configuration used must be assessed.

Configurable computing architectures can be divided into two categories based upon their configuration techniques. These categories are known as *Compile-Time Reconfiguration* (CTR) and *Run-Time Reconfiguration* (RTR) [49].

2.3.1 Compile-Time Reconfiguration

Compile-time reconfiguration was the first configuration mechanism developed. The name of this technique reflects the limitation caused through using first generation FPGA devices in constructing dynamic systems. FPGAs of this era had to be configured as a whole unit, independent of the proportion of the design requiring updating. This limitation forced the processing architecture to appear fixed during system operation, hence only supporting a single configuration (single context).

Initially this method of configuration did not introduce any performance limitations. Early configurable computing machines were designed to provide high performance processing platforms, optimised for different algorithmic structures on a task-by-task basis. For each individual task, the system configuration would be determined and downloaded prior to execution of the task. Since the configuration remained static

during system run-time, the initial configuration delay encountered would not affect overall system performance. This concept shown in Figure 2.8 was true for most early configurable computer architectures.

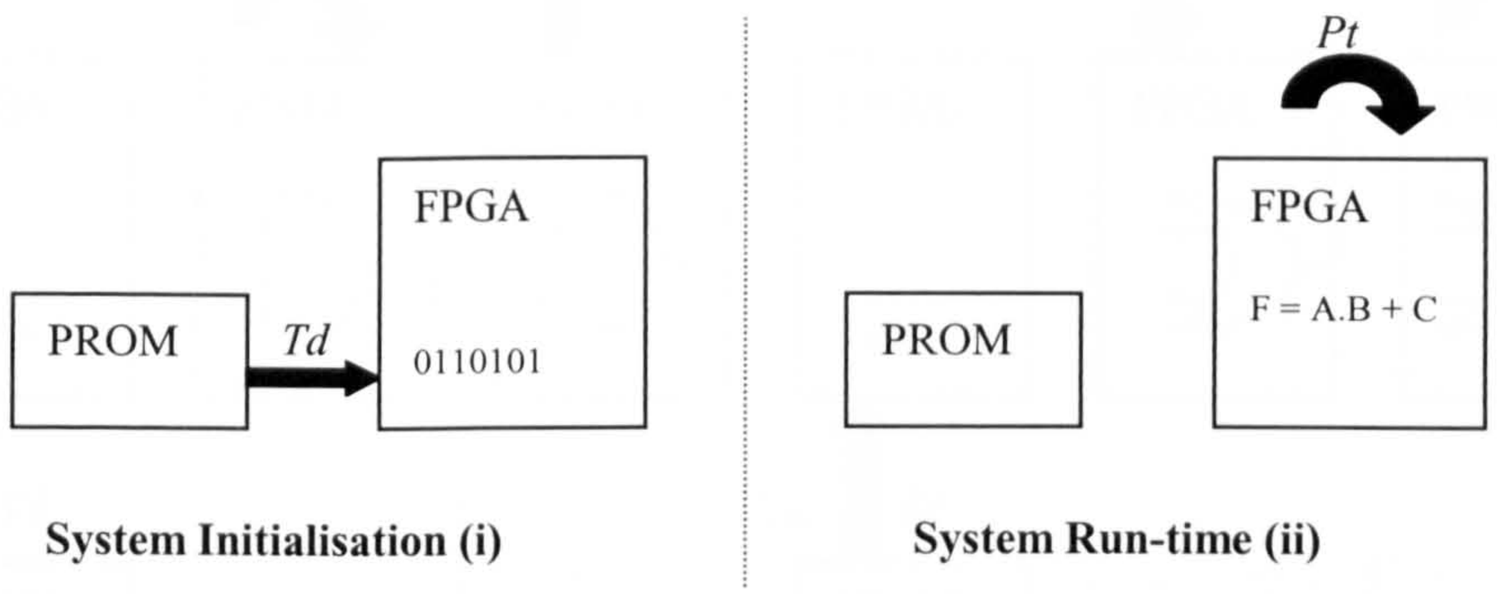


Figure 2.8 Simple CTR System Operation

Figure 2.8 illustrates how configuration data is downloaded (i) prior to system operation (ii). The configuration is single context therefore this delay occurs once (*Td*). To evaluate the performance of such systems, only the processing time (*Pt*) needs to be considered. The configuration delay (*Td*) can be disregarded as device power-up and initialisation delay, which are commonplace within electronic systems.

The limited gate capacity of early FPGA technology depicted the complexity of processing hardware developed. To construct larger designs, the required architecture could be partitioned into multiple configurations (temporal partitioning). This required analysing the design to determine at any given phase during system operation which hardware resources were active, and which were inactive. By swapping inactive with active logic, a design requiring a higher gate capacity than was physically present could be implemented. To achieve this system operation would be suspended whilst new configuration data was downloaded.

The CTR configuration mechanism constrained the functionality and ability of the architecture to be adapted efficiently during system run-time. Configuration delays incurred, therefore reduced any speed-up. This limitation was applicable to second-generation configurable machines and is illustrated in Figure 2.9.

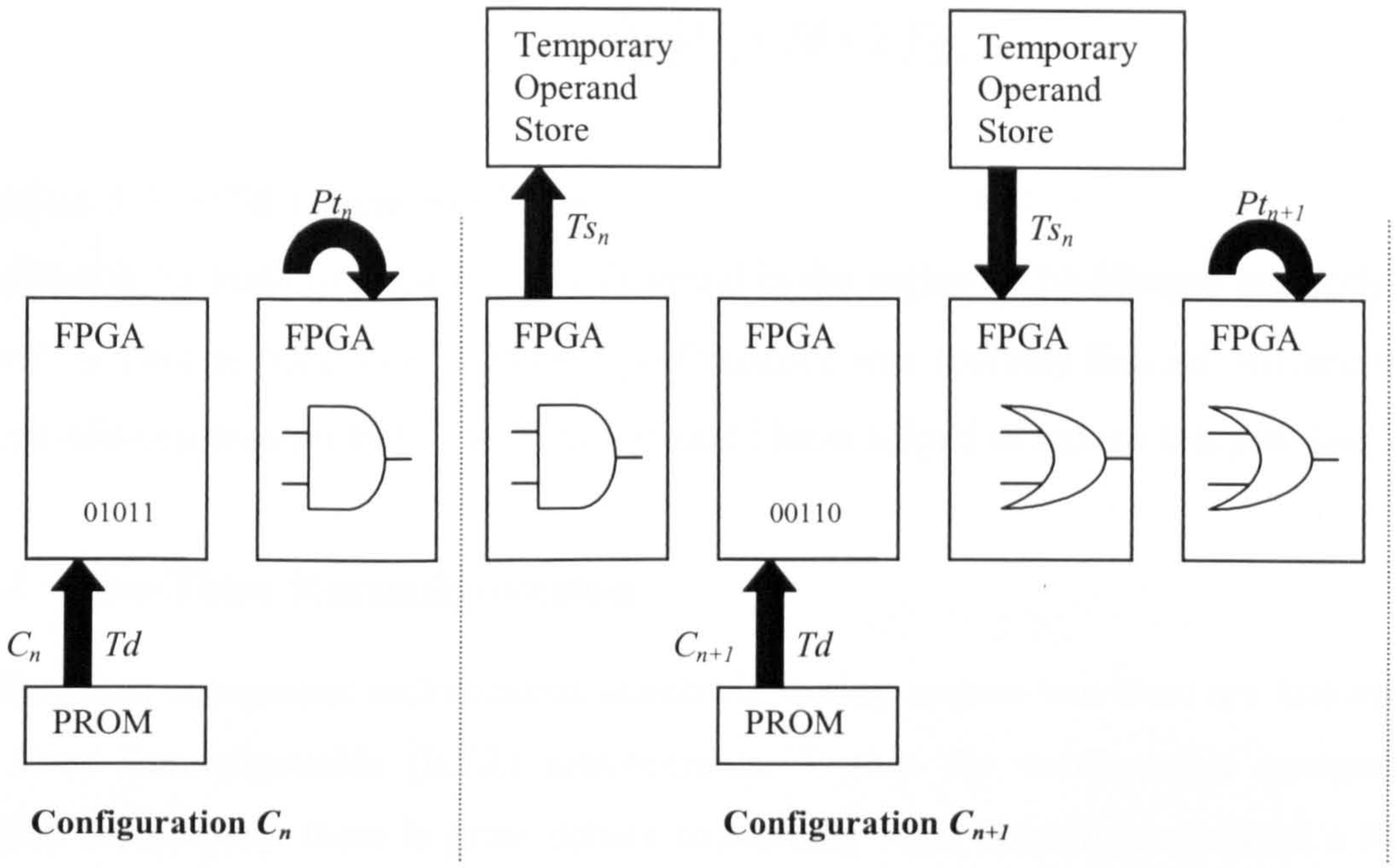


Figure 2.9 Multiple CTR System Operation

Figure 2.9 illustrates a multiple context temporally partitioned CTR design. Upon power-up the first configuration (C_n) was downloaded generating a configuration delay (*Td*). This initial delay can be ignored and attributed to system initialisation. Process C_n takes time Pt_n to complete. Before the next configuration (C_{n+1}) can be downloaded, data required by C_{n+1} and present in the C_n must be stored temporarily. This function takes time Ts_n, and must be completed before the next configuration can be downloaded.

When the FPGA is reconfigured, its content is overwritten hence intermediate data is erased. The configuration delay incurred is still *Td* since the device must be configured as a whole unit, independent upon the percentage of the architecture required updating. Before system operation recommences, operand data stored locally must be retrieved introducing a further delay of Ts_n. Configuration C_{n+1} commences, and is completed in time Pt_{n+1}. The cycle is then ready to be repeated. Ignoring the initial configuration delay, the total processing time (*Ptot*) for '*n*' configurations can be determined using Equation 2.1.

$$Ptot = Pt_0 + \sum_{0}^{n-1} Pt_n + Td + 2Ts_n$$

Equation 2.1 CTR Processing Time

Considering the configuration delay (*Td*) would in the region of 20-40msec and both Ts_n and Pt_n in μsec or nsec, overall system performance was severely limited. Advances in custom and commercial FPGA technology based have helped to reduce this problem.

2.3.2 Run-Time Reconfiguration

Configurable computing architectures adaptable during system run-time are known as *Run-Time Reconfigurable* (RTR) architectures. Within the configurable computing research community, there is great debate concerning what actually constitutes a RTR system. It has been argued that configuration overheads alone should determine whether or not a system is RTR. Typically, such systems must therefore be reconfigured within a couple of clock cycles at normal operating frequencies. This notion represents the ideal characteristics for an RTR system.

It is also argued that the action of partially updating a design without halting the remainder of the systems operation deems it to be RTR compatible. This is the opinion of the author. Using this approach, RTR can be implemented using partial and dynamic configuration. However system performance will be reduced through the configuration delay generated using partial reconfiguration. This reduction in performance will only be apparent if system throughput stalls whilst waiting for hardware currently being reconfigured. Within the research community, the function of RTR has also been referred to as *multiple-context switching*, *adaptive logic* and *virtual hardware*.

To perform RTR, the configuration data must be refreshed. Configuration data can either be located in an externally memory or on-chip, as illustrated in Figures 2.10 and 2.11 respectively.

Method-one (Figure 2.10) requires an external configuration store containing multiple configurations (PROM). When requested by the system, a new configuration is downloaded. The transfer of data introduces a configuration delay caused by the interface bandwidth bottleneck between the reconfigurable device and PROM. The configuration delay is a product of the configuration interface bandwidth and the volume of configuration data. Removal of this bottleneck reduces the configuration delay and increases system performance.

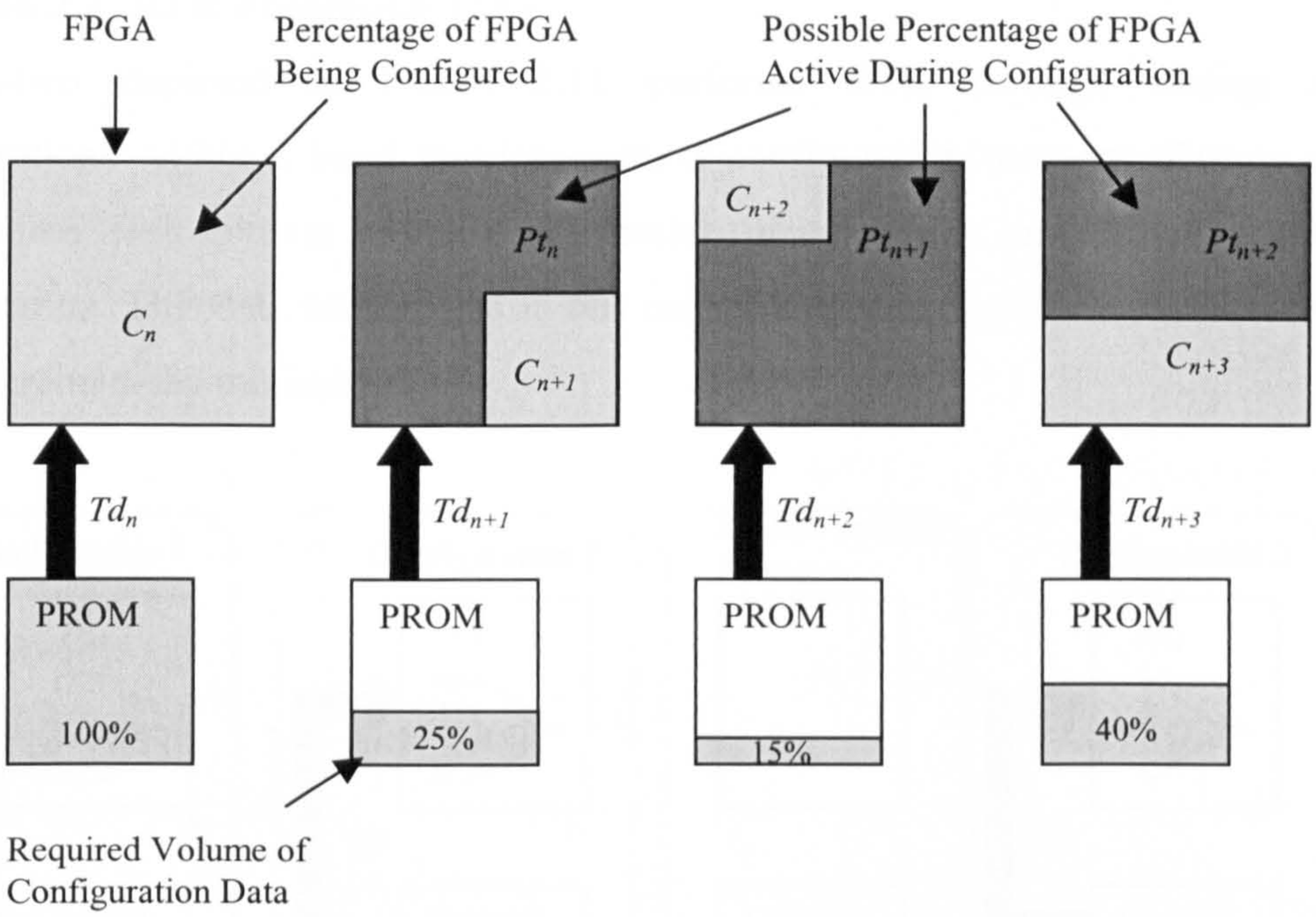


Figure 2.10 External Configuration Mechanism

Through using partial configuration the volume of configuration data is reduced. Reconfiguration however, cannot occur in one concurrent operation since configuration data is stored off-chip.

Using partial configuration, the configuration delay (Td_n) is proportional to the percentage of the device being reconfigured. During this delay, unaffected regions of the architecture can still function normally therefore exhibiting the architecture's RTR capabilities.

To calculate the performance of the architecture, the configuration delay between each partial configuration update (Td_{n+1}) needs to be compared against the current active processing time (Pt_n). The greater of the two delays (Tdp_n) defines the overall delay of a particular configuration (C_n). Ignoring the initial configuration delay Td_n, system performance can be calculated using Equation 2.2.

$$Ptot = Pt_n + \sum_{0}^{n-1} Tdp_{n+1} \qquad Tdp_n \begin{cases} Tdp_n = Pt_n, Pt_n > Td_{n+1} \\ Tdp_n = Td_{n+1}, Pt_n < Td_{n+1} \end{cases}$$

Equation 2.2 RTR Processing Time

Method-two depicted in Figure 2.11 performs RTR through storing system configurations within a local multiple-context configuration memory. During system initialisation each context of the configuration memory was loaded with a different configuration. This data was written in one concurrent operation during RTR, hence the configuration delay minimised.

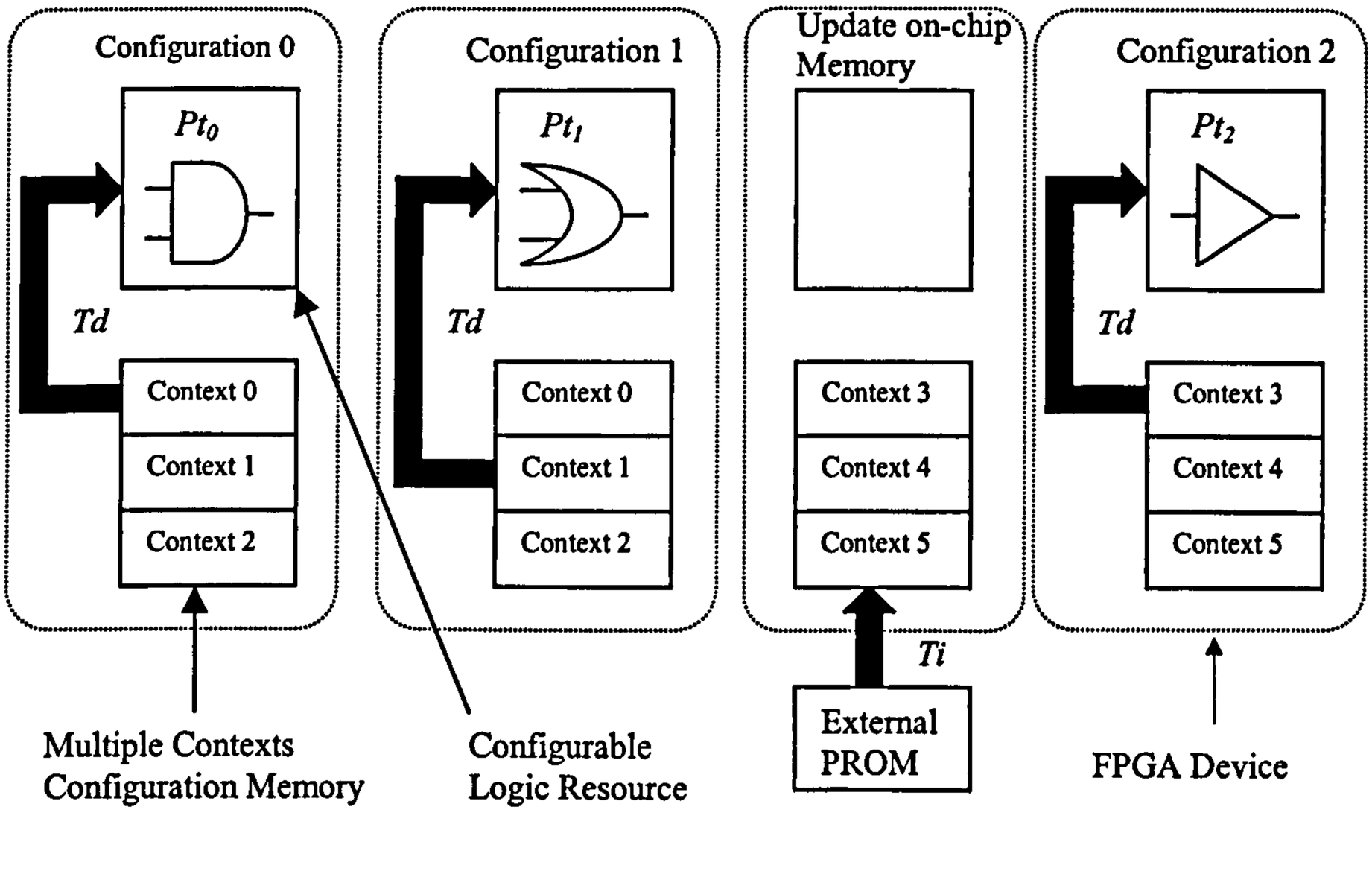


Figure 2.11 Internal Multiple Context Configuration Mechanism

The depth of the context memory limits the number of RTR configurations. This memory takes up large areas of the silicon chip therefore reducing the logic capacity of

the device.

This initialisation period (*Ti*) of multiple–context operation can be disregarded when calculating system performance, unless configuration memory download occurs again during run-time. This is only required if the total number of configurations required exceeds the capacity of the on-chip configuration memory.

If a configuration does not reside in the local configuration memory, it must be written from an external source. The total configuration delay would therefore be equal to the combined delays of *Td* and *Ti*, effectively halting RTR. However excluding *Ti*, RTR is achieved during normal operation since the configuration can be updated at system clock speeds.

Selecting a memory context reconfigures the device in one concurrent operation. This introduces a configuration delay (*Td*) that is constant for every context switch and far less than the equivalent off-chip configuration store delay.

2.4 Architecture Classification

The classification of configurable processing architectures using traditional topologies such as Flynn's [50] is inaccurate. In a reconfigurable architecture, the active configuration may exhibit properties related to that of Flynn's machine classification. This relationship however is only based upon the properties of the hardware configuration downloaded, and does not actually describe the underlying architecture of the system. Further inaccuracies are introduced due to the run-time adaptive nature of the configurable computing architectures. Therefore to derive a general taxonomy that encases configurable computing architectures, the type and application of architecture, configuration mechanism, and granularity of configuration must be addressed.

Section-2.2 illustrated that configurable processing architectures could differ considerably. To classify and compare the performance of configurable architectures

0was difficult due to differences in the configuration mechanism, granularity of configuration and system resources.

The first taxonomy proposed by Guccione was based upon the logic capacity of the configurable resources and the availability of local memory, and divided configurable processing architectures into four major categories [51]. These were *Application Specific Architectures* (ASA), *Reconfigurable Logic Coprocessor* (RLC), *Custom Instruction-Set Architectures* (CISA), and *Reconfigurable Super Computers* (RS). Within this taxonomy, the configurable resources are collectively known as the *Reconfigurable Processing Unit* (RPU). The relationship between local memory and RPU capacity is shown in Table 2.1.

	No local memory	**Local memory**
Small RPU	CISA	RLC
Large RPU	ASA	RS

Table 2.1 Guccione's Reconfigurable Computer Taxonomy

In this taxonomy, the names given to each category did reflect specific types of system. Overall however, it was too generalised and did not take into account the configuration mechanisms used nor the granularity of configuration. These characteristics, the RPU logic capacity, and memory structure were essential to accurately classify architecture types Classifications of real working systems are given in *Section-2.5*, according to these criteria.

The difficulties encountered in constructing a suitable taxonomy were also highlighted when trying to compare the relative performance of two different configurable computing architectures. DeHon proposed a method for comparing the performances of configurable architectures and instruction-set processors [52]. The basis of this method was to determine whether an architecture achieved superior performance through the use of RTR, or simply because the computation was implemented within greater logic resources. DeHon compares two architectures by using three factors *area*, *time*, and *energy*.

An architectures *area* was determined through calculating the volume of configurable logic, memory resources, and hardwired logic (if applicable) implementing the design. To provide a comparison with other semiconductor technologies, the transistor sizing parameter lambda (λ) was considered. Lambda determines the size and separation of transistors upon a silicon die, hence the area of silicon used by an application could be determined.

The second factor (*time*) was determined by considering the duration of an application, rather than comparison of device clock frequencies. Within different systems, the amount of work conducted per clock cycle was not uniform. Using this analysis, the operand throughput could also be used to calculate this factor.

The third factor considered the quantity of *energy* used to perform the computation. This was a valid factor when the power consumption of a design needed to be assessed. When determining the faster of two systems however, it was of minor relevance.

The characteristics of each type of computer listed in this taxonomy and others are discussed in *Section-2.5*. The determination and interpretation of architecture characteristics to be used in the development of a universal classification system for reconfigurable computers is a highly debated topic within the research community, with no clear answers so far commonly accepted.

2.5 Configurable Computing Applications

By analysing system operation and application, existing configurable computing architectures can be divided into five dominant types. Using the taxonomy names detailed in *Section-2.4* as they describe each functional class best but with a refined classification, these categories are:

(1) Application Specific Architectures

(2) Prototype Environments

(3) Reconfigurable Logic Coprocessor

(4) Reconfigurable Supercomputers

(5) Custom Instruction-set Architectures

2.5.1 Application Specific Architectures

Most early configurable computers were designed to accelerate specific applications and could be considered as traditional ASAs except the design had been implemented using configurable logic rather than hardwire ICs. In later ASA's, the processing architecture was designed to include reconfiguration within normal operation. This was to ensure that if the structure of the target algorithm evolved during run-time (e.g. evolutionary algorithms), the processing architecture could adapt and accommodate this.

It was possible to adapt ASAs to compute other functions, however system performance would degrade as the interconnection of system components such as hardware resources, local and shared memories were typically fixed and optimised to compute the original application.

An example of an ASA was Ganglion [19]. Ganglion was developed at IBM's research division, San Jose, USA in 1991. The aim of the project was to develop high performance processor architectures with reduced development cycles, to implement connection classifier *artificial neural networks* (ANNs).

2.5.2 Prototype Environments

Configurable architectures have been designed to facilitate the development of hardwired ICs [20], multiprocessor and DSP architectures [22], RTR application development and hardware-software co-design [53]. To facilitate wide ranging applications such as these, three distinct type of prototype environment exist. These are massive-scale CTR systems (millions of gates), small-scale CTR systems and small-scale RTR systems (thousands of gates).

2.5.2.1 Massive-Scale CTR Prototyping Environments

Within massive-scale CTR prototyping environments, multiple FPGA devices connect together forming an array and interact through dedicated fixed routing resources or FPIDs. Prototype systems of this type have limited or no local memory resources. A prominent example of this type of architecture was *Transmogrifier-2* (TM-2) prototyping system [78], with an architectural description provided in *Appendix-II*. TM-2 contained logic resources to implement designs with up to 1,000,000 gates.

2.5.2.2 Small-Scale CTR Prototyping Environments

The second types of prototype environment were small-scale CTR systems, typically consisting of one to three FPGA devices coupled to a host microprocessor. In this type of architecture, the FPGA and microprocessor accessed a shared memory resource. During system operation, the microprocessor provided additional prototyping resources through interaction with those configured upon the FPGA(s). The combined processing resources were loosely coupled, since their interaction was redefined for each application. Systems of this type were used for the partitioning of hardware-software co-design. An example of this type of system was Harp [26].

2.5.2.3 Small Scale RTR Prototyping Environments

The third type of prototype environment was small-scale RTR systems. The architecture and function of these systems can be considered similar to that of the small-scale CTR systems, except that now RTR is used. An example was RACE (*Reconfigurable and Adaptive Computing Environment*) developed at the University of Cincinnati [53], supporting up to 52,000 logic gates.

Common to all system types, the configuration of each programmable device was determined using development software that partitioned a design amongst the available resources. Individual configurations were then downloaded to each device via the host computer. The prototype hardware implemented could then be evaluated with software techniques or by traditional hands-on approach (e.g. using an oscilloscope).

2.5.3 Reconfigurable Logic Coprocessors

Configurable coprocessor architectures have been designed to accelerate applications by combining the processing resources of an instruction-set based processor and a CTR or RTR device. Partitioning the computational overheads upon two processing fabrics increases the throughput of an application. This is achieved through implementing portions of the process that can be executed faster on the configurable logic than on the main processor.

Configurable coprocessors have been fabricated using custom and commercial devices, with both possessing the same underlying architecture. The architecture of a typical configurable coprocessor is similar to that of a small-scale prototype environment. Both devices can access shared memory, but unlike the prototype environment, the coprocessor system instruction-set and configurable hardware resource are tightly coupled. Since the operation of the coprocessor is highly integrated with that of the primary processor, a high bandwidth communication interface is required between them.

The first example of this type of architecture was PRISM [27]. PRISM was an acronym for 'Processor Reconfiguration through Instruction-set Metamorphous' and was developed in 1992. Other examples include GARP [25] and Morphosys [43], with the Morphosys architecture being described in detail in *Appendix-II.*

2.5.4 Reconfigurable Supercomputers

Configurable supercomputers are massive-scale configurable architectures designed to accelerate applications by exploiting concurrent properties of a task directly in hardware. The underlying components of a reconfigurable supercomputer are similar to that of massive-scale prototyping environments, with application implementation being partitioned amongst a large number of interconnected FPGA devices (typically tens of FPGAs).

Within supercomputers a dedicated FPGA interconnection topology exists to minimise communication bottlenecks. Typically FPGA devices can access shared global and private local memory resources. Due to the sheer size of these architectures, complex application development software is required to efficiently partition the design upon the available resources. Prominent examples of configurable supercomputers were the Virtual Computer [23] and Splash-2 [24]. Splash-2 system architecture and operation is described in *Appendix-II.*

2.5.5 Configurable Instruction-Set Architectures

Configurable instruction-set computing is the ability to reconfigure a hardware resource with custom instructions during run-time, therefore exhibiting a virtual instruction–set. Configurable instruction-set computers could be considered as configurable coprocessor systems since both implement instructions within reusable hardware. However, configurable instruction–set machines do not contain a primary processor, but instead have only a skeleton architecture that is responsible for instigating the configuration of instructions as demanded by program flow. Once instructions have been used, they are removed and the hardware resources become available to implement new instructions, typically using RTR techniques.

An early example of a configurable instruction-set processing architecture was the Nano Processor developed 1994 [28]. The Nano Processor consisted of a fixed processing core, and a custom instruction-set compiled for a given task. Evolutions of the Nano processor were DISC and DISC-2 [42]. DISC and DISC-2 could configure instructions as demanded by the core processor during run-time. DISC's operation is described in detail in *Appendix-II.*

2.6 Summary

The configurable computing systems described represent a cross section of the current status of this technology. Configurable computing technology is constantly evolving with development tools, reconfigurable media, system architecture, and application development continually improving.

To develop more suitable configurable media, the granularity of configuration, speed of configuration, and overall control structure governing virtual hardware instigation must be addressed. The Kress Array project [54] is currently investigating the granularity of reconfigurable structures and the methodologies by which they can be reconfigured. This work has shown that coarse-grain CLBs implementing specific functions rather than product terms are most suitable for configurable computing applications.

Research is also being conducted to develop optically reconfigurable FPGAs configured using spatially modulated structured light [14] through an optical fibre interface. Since light is being used, the configuration delay of the device is dependent primarily upon the operational speed of the photo-electronic configuration cells. Compared to SRAM RTR technologies, optical reconfiguration should reduce configuration delays through the increase in configuration data transfer rate, and response times of photo-electronic cells compared to SRAM. This technology however is still in its infancy and as of yet, the author does not know of any functional device.

Developments in reconfigurable technology have provided a platform upon which the integration of traditional processing architectures and configurable computing concepts can evolve. An example of such is the *Dynamically Programmable Cache* (DPC) project [55]. This aims to reduce configuration overheads through integrating configurable hardware within the cache memory architecture of an instruction-set processor. The cache appears to the processor during run-time as either dedicated cache or a tightly coupled coprocessor.

A further example has been the development of a commercial FIPSOC™ (*FIeld Programmable System On a Chip*) by SIDSA [56]. This architecture combines a low power microprocessor core, dynamic programmable logic, and dedicated communication interface together specifically for use in multiprocessor applications.

In most applications, configurable logic has been primarily used to implement processing hardware. However, in the RENNS computer system, configurable logic has been used to implement an interconnection topology optimised for each task [57]. Another example ARMEN [58] consisted of a MIMD architecture incorporating FPGAs configured for each task to provide additional processing and inter-node routing resource.

An aspect of the research work presented in this thesis advances this idea by incorporating dynamic configurable media in a custom MIMD architecture. Similar to ARMEN each MIMD processing node has a coprocessor, but distinct from previous systems, each processing node can exploit virtual hardware capabilities. Within this thesis, a dynamic routing hub is presented that can be configured during run-time with additional processing resources, which is a novel concept.

Development tools and design verification methodologies also need to be improved. In comparison to commercial software tools used in developing traditional FPGA applications, configurable computing tools are very inefficient. It is inherent that different configurable architectures may require unique operating software to govern system operation. However, the strategies used to generate the system configuration could be unified into a common design language.

Similar to the programming language JAVA™ [59], a design could be described using a common syntax and only differ by how logic is mapped upon a particular architecture. Examples of such design languages are RUBY [60], Handel-C [61], and Lola [44]. Handel-C is the design language used to implement designs upon the Harp configurable coprocessor, and Lola is used with the Trianus set of design tools developed for the

Hades dynamic configurable coprocessor.

A further emerging discipline is evolutionary electronics. Evolutionary electronics are applications that use artificial evolution to generate hardware fulfilling a design criterion [62]. The resultant design is generated through multiple cycles in which the difference between the actual and the required output of the system are examined, and the existing architecture then modified accordingly. Dynamic FPGAs are used as the implementation media and utilise partial and dynamic configuration to update the architecture through each evolution in the design cycle. An example of evolutionary electronic hardware has been a design that could distinguish between two different frequencies [63]. This was implemented using a Xilinx XC6200 FPGA.

A problem with existing evolutionary electronic technology is that although it may implement a design very efficiently, how it actually functions can be difficult to interpret. Factors such as signal propagation and routing delays can influence the evolutionary cycle design and can vary between identical FPGAs. In traditional designs, these problems are removed through the use of synchronous design techniques.

At the present moment in time, configurable computing technology is still in its infancy and needs to mature before its incorporation into industry. FPGA vendors such as Xilinx and Altera, and the research community as a whole recognise this, but to accelerate the evolution of the new exciting computing concepts described, a 'killer' application is required. Such an application is sought to boost industrial interest in RTR technology, there by fuelling greater interest and the provision of more resources for the on-going development of reconfigurable computing technologies.

Within this thesis a novel application using RTR implementation has been developed. Even though the applications implementation was constricted by the limitations of existing dynamic FPGA technologies, resultant hardware has shown how dynamic configuration can be used to increase the operand throughput, compression ratio, and accuracy in approximating *Discrete Cosine Transform* (DCT) operation.

Chapter 3

Dynamic Hardware Development System

Introduction

The aim of this chapter is to describe the design and operation of the development system used during the research program. The chapter introduces the system components first, and then explains the construction of each, with combined system configurations detailed at the end of the chapter. Descriptions of key semiconductor devices used are provided here, with detailed explanations contained within *Appendix-III.*

3.1 Overview

To provide a platform for the evaluation and inclusion of RTR within a multiple processor environment a research and development system has been constructed. This incorporated a commercial parallel processing architecture, dynamic hardware platform, and software development tools.

The parallel processing architecture consisted of four TIM-40 standard TMS320C40 DSPs *(Section-3.2)*, chosen since they facilitated the insertion of additional hardware within the routing topology and memory address space. The dynamic hardware resource consists of a custom designed *XC6200 FPGA Development System* (XC6200DS) *(Section-3.3)*.

To facilitate the development of RTR applications, software has been written (XC6200ADS) that enabled XC6200DS hardware to be evaluated *(Section-3.3.5)*. This software also generated dynamic configuration data, and governed the transfer of operands to and from the XC6200DS.

An operational system consisted of one TIM-40 motherboard with up to four TMS320C40 processors installed, and up to three XC6200DS cards attached to it. To

implement a design upon the combined system architectures, the RTR hardware and TIM-40 system were programmed independently. Depending upon the system configuration, the XC6200DS could appear as a RTR coprocessor, routing hub, prototype environment and self-configuration system *(Section-3.4)*.

3.2 TMS320C40 Parallel Processor

The parallel processing system used was based upon the TIM-40 standard [64], developed by Texas Instruments in conjunction with a consortium of DSP related manufacturers. This standard enabled the development of multiple processor systems through use of a modular format consisting of processor modules *(Section-3.2.1)*, peripherals and host motherboards *(Section-3.2.2)*.

Processor modules were developed using the TMS320C40 [65] (C40 hereafter). This was a 32-bit floating-point based DSP designed specifically for use in multiple processor environments. Incorporated in C40 architecture were components dedicated to facilitate inter-processor communication without degrading overall system performance. These consisted of six high-speed communication ports used to implement inter-processor routing topology, and two external memory interfaces known as the *Global* and *Local* interfaces. More detailed information describing the C40 DSP is contained within *Appendix-III*.

3.2.1 TIM-40 TMS320C40 Processing Node

Transtech Parallel Systems TDM411 type TIM-40 modules were the DSP modules used. TDM411s consisted of a single C40 DSP with 4-Mbytes of *Enhanced DRAM* technology (EDRAM) mapped within both the Global and Local memory interfaces. Each 4-Mbytes was local to the C40 and accessed using signal *strobe0* of each interface. Figure 3.1 illustrates the structure and position of system components upon the TDM411.

The TDM411 itself did not constitute a fully functional system. Instead, it was connected to a TIM-40 standard motherboard using the *Primary* and *Secondary* connectors. Through these connections C40 resources such as memory interfaces and communication port signals could be accessed, as well as providing system house keeping functions and power supplies.

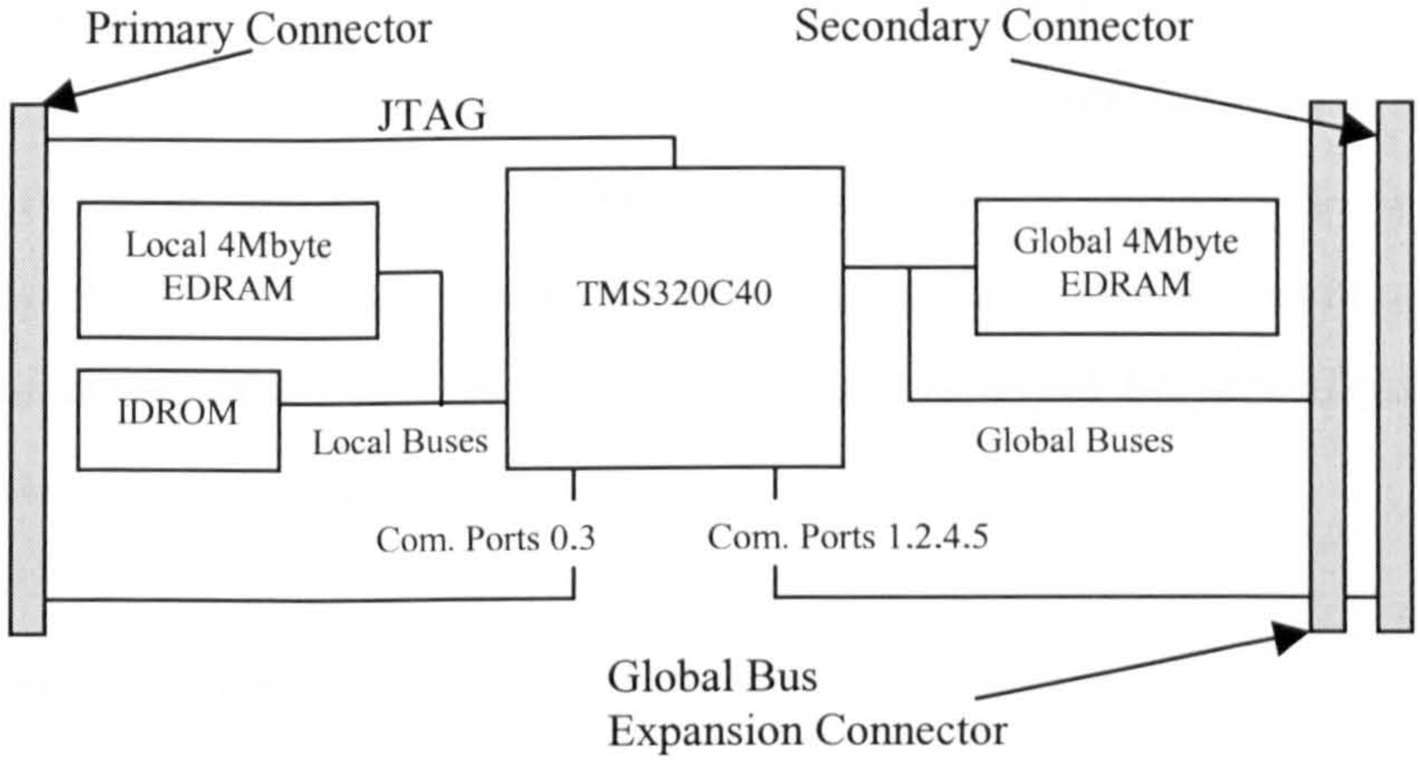


Figure 3.1 Transtech Parallel Systems TDM411 Processor Module

The TDM411 had a further connector known as the *Global Bus Expansion* connector. If the TIM-40 motherboard supported this connector (optional), additional or shared memory could be accessed using signal *GSTROBE1* of the Global interface. This connector allowed peripherals direct access to the control, data and address-buses of the Global interface.

In accordance with the TIM-40 standard, the TDM411 had an IDROM containing information that detailed the organisation of the C40s memory space. Upon system initialisation the content of the IDROM was downloaded to C40 configuration registers. Further, JTAG in-circuit evaluation and debugging was supported [68].

3.2.2 TIM-40 Motherboard

To provide a host platform for the TDM411 modules, a Transtech Parallel Systems TDMB412 TIM-40 motherboard was used. The TDMB412 was a full-length 16-bit PC ISA based peripheral, which could host up to four individual TDM411 modules. The role of the motherboard was not just to provide power and access to C40 signals, but also provide an underlying processor interconnection topology, a Texas Instruments XDS510 compatible JTAG interface, and an interface between the C40 system and host computer. The structure of the TDMB412 motherboard is shown in Figure 3.2, indicating the default data direction of each C40 communication channel. However, C40 communication channels were bi-directional and could be reconfigured as either uni-directional or bi-directional.

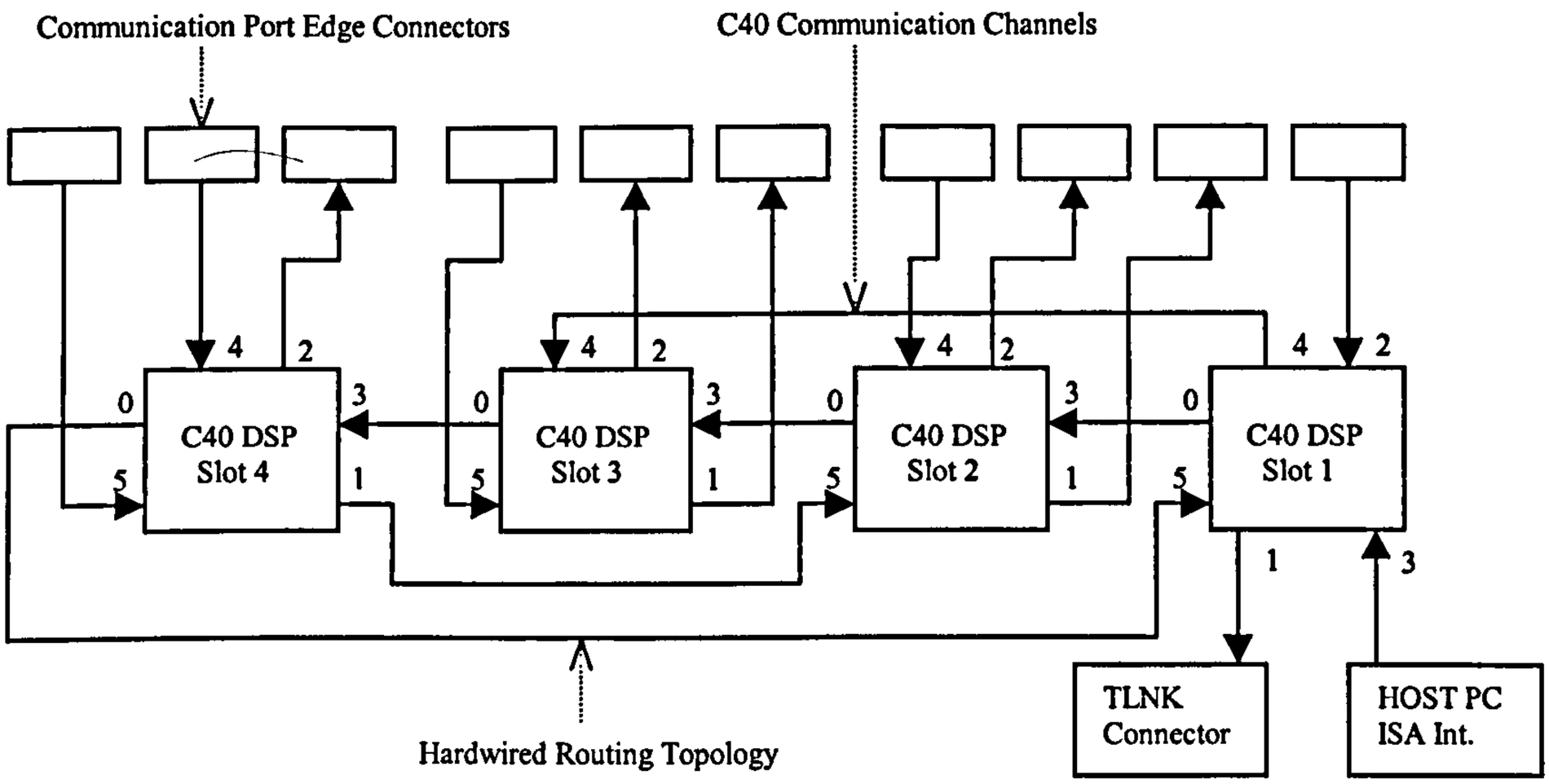


Figure 3.2 Transtech Parallel Systems TDMB412 TIM-40 Motherboard

The construction of a multiple processing environment was simplified using this modular approach. Although only four C40 modules could be implemented per motherboard, individual motherboards could be linked together forming larger systems.

Two communication channels of each C40 could be accessed through connectors situated on the edge of each motherboard. The four remaining channels of each C40

were connected between the other TDM411 positions. This interconnection topology was hardwired within the fabric of the motherboard.

The C40 installed in slot-one on the motherboard was regarded as the JTAG *root* processor, since it was the first processor encountered by the on-board JTAG debugger daisy chain. When additional TDM411 modules were plugged into the motherboard, they were inserted into this chain as *slave* devices. In a multiple motherboard systems, only one root processor could exist in the JTAG chain. Root processors on subsequent boards were therefore set to slave and the on-board JTAG controller disabled via switches on all but the root motherboard.

To link applications running on both the host computer and TIM-40 system, a hardware interface could be formed using communication channel three of the root processor. Slave TDM411s therefore communicated with the host computer via the root processor. Switches upon the TDMB412 permitted the base address location of this hardware to be relocated in the host development computer, as well as disabling the interface itself.

3.2.3 TMS320C40 Application Development

The software development tools supplied with the TIM-40 system were called PaCE [66]. The development process used throughout the project is shown in Figure 3.3. Designs were entered using either assembly language (prog.asm) or a C40 C programming language variant (prog.c), which were then compiled and checked for errors using software tools *asm30* and *cl30* respectively. The next stage of the process was to link the compiled application with the hardware resources of the target system using the *lnk30* software tool. The hardware resources of the target system were described within a command file (prog.cmd), that allocated system memory areas used by software structures, and determined the location and size of the heap and stack.

Once the design had been linked, resultant programs could be simulated, evaluated in circuit, or executed normally. Tool *sim4x* was used to simulate the execution of a

program and allowed the contents of C40 registers to be viewed, communication channels created, and single step execution of the program performed. To allow the simulation to function correctly, a command file describing the address space configuration of the TDM411 was required (siminit.cmd).

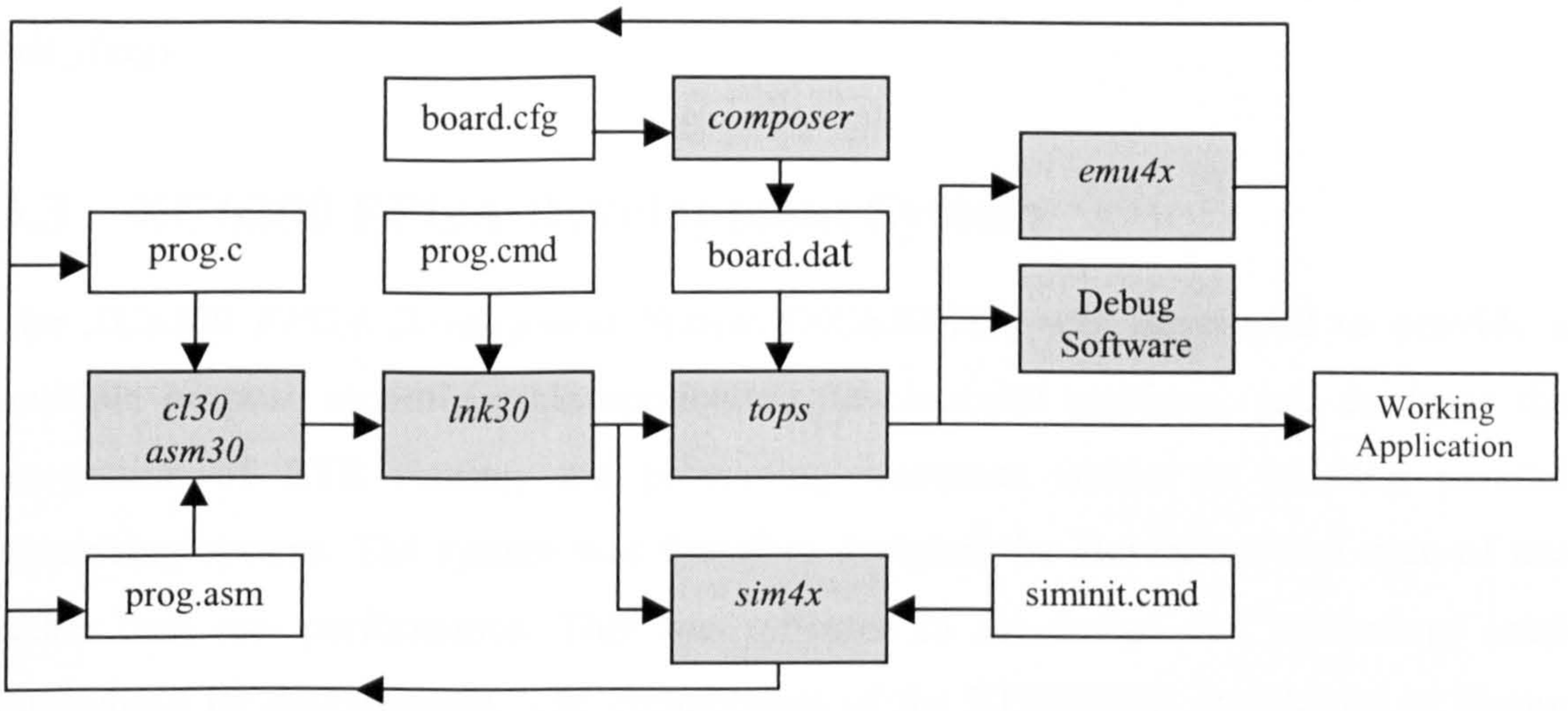


Figure 3.3 TMS320C40 PaCE Development Cycle

The C40 JTAG port enabled run-time system evaluation. Elements forming the TIM-40 system architecture had to be given an identity in order to generate a JTAG chain file (board.dat). This was performed using utility *composer*, which read an input file describing the architecture (board.cfg). C40 programs could then be downloaded to the system via the *tops* software tool. Next, the JTAG emulator *emu4x* was executed, which took control of the C40 and placed it in single-step execution debug mode. Using this software, internal registers and memory locations were accessed.

A further method used to evaluate and debug designs was the creation of custom test programs to display memory and register contents upon the host PC. However, this was only possible through the root processor of the motherboard.

The development procedure outlined was used to debug C40s individually. To develop parallel processing applications, software for each C40 was developed independently, with individual programs downloaded on-block to the system. Program downloads were

conducted using *tops*, which read a configuration file (prog.nd) indicating the appropriate program to download to each C40.

TIM-40 utilities were also supplied that enable the configuration of the TIM-40 system to be checked (*tcheck*), and JTAG chain related operations to be debugged (*jtagrst, xds_diag*).

3.3 XC6200 FPGA Development System

The *XC6200 FPGA Development System* (XC6200DS) was developed to provide a multiple purpose reconfigurable application development platform, and facilitate the integration of RTR routing and processing resources within an existing parallel processing system. The system was therefore designed for flexibility, and ease of use rather than raw performance. This was reflected in the design and technology used throughout its development. The components of the XC6200DS are shown in Figure 3.4. They integrate with the TIM-40 parallel processing system using external connection leads and host PC address space.

Initially, a FATHOTs XC6200 based development system purchased from the Virtual Computer Corporation [67] was intended to be used for RTR application development. After conducting several experiments with this system it was determined that the debugging tools, local memory and overall operation were generally unreliable. Therefore the XC6200DS was designed.

Development platform operation required both hardware (XC6200DS) and software components (XC6200ADS). Each hardware unit consisted of an ISA based PC card containing a XC6200 family FPGA, related interfaces, and control logic. The XC6200 FPGA was used as the reconfigurable logic, with the control circuitry providing the interface between the XC6200 and application development software upon the host PC (XC6200ADS).

XC6200 FPGAs were chosen since they could be reconfigured using partial and dynamic configuration capabilities. These were the only commercially available devices able to do this, and to the author's knowledge, still are. This was made possible through use of a novel hardware interface known as the FastMAP™ interface. The XC6200 family architecture is described in further detail in *Appendix-III*.

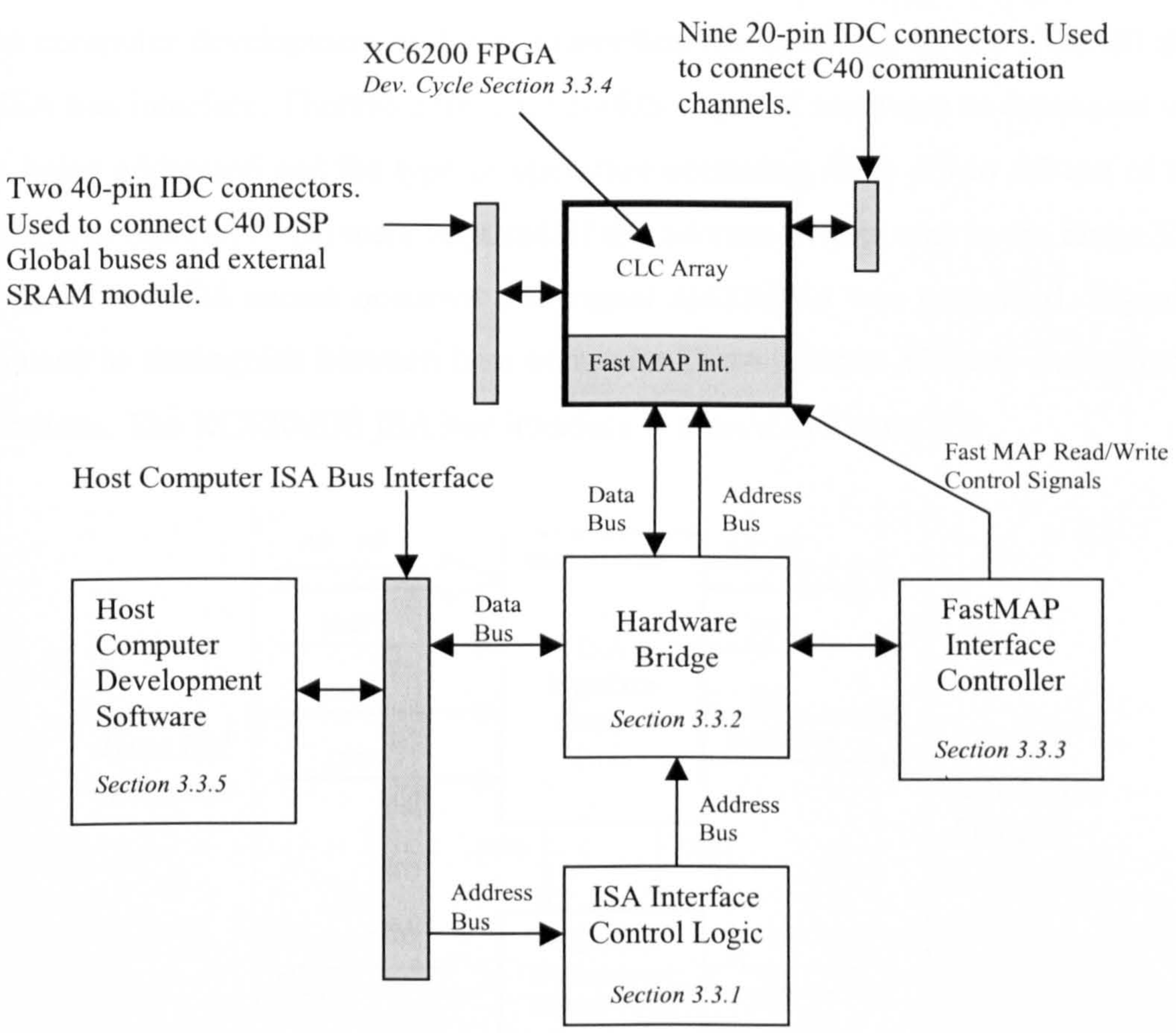


Figure 3.4 Components of XC6200 Development System

To construct the XC6200DS, FPGA devices XC6216 and XC6264 were purchased. Both possessed the same underlying architecture but had different logic capacities of 24,000 and 100,000 gates respectively [32]. Collectively, these devices are referred to as XC6200 throughout the discussion.

The XC6200 interacts with the host computer using an 8-bit ISA port. The decision to use an ISA rather than PCI specification interface was based upon the simplicity of the

interface, with the aim of reducing the development overheads. To fabricate the XC6200DS ISA card, a custom PCB was developed using Zuken CadStar PCB development tools. In total three identical PCBs were constructed which took considerable time, and all successfully commissioned.

3.3.1 Host Computer Interface

Host computer development software controlled the operation of the XC6200 through an ISA bus interface. Therefore the XC6200DS required hardware to determine when it was being addressed and the type of operation occurring; Only A9 to A0 out of the 20-bit address-bus (A19-A0) were required. If the address written was in the range 320_{16} to $32F_{16}$, XC6200DS access occurred and signal *DATAENA* was generated. Signal AEN was used to distinguish between host computer DMA (*Direct Memory Access*) and I/O operations. The XC6200DS ISA bus interface is shown in Figure 3.5.

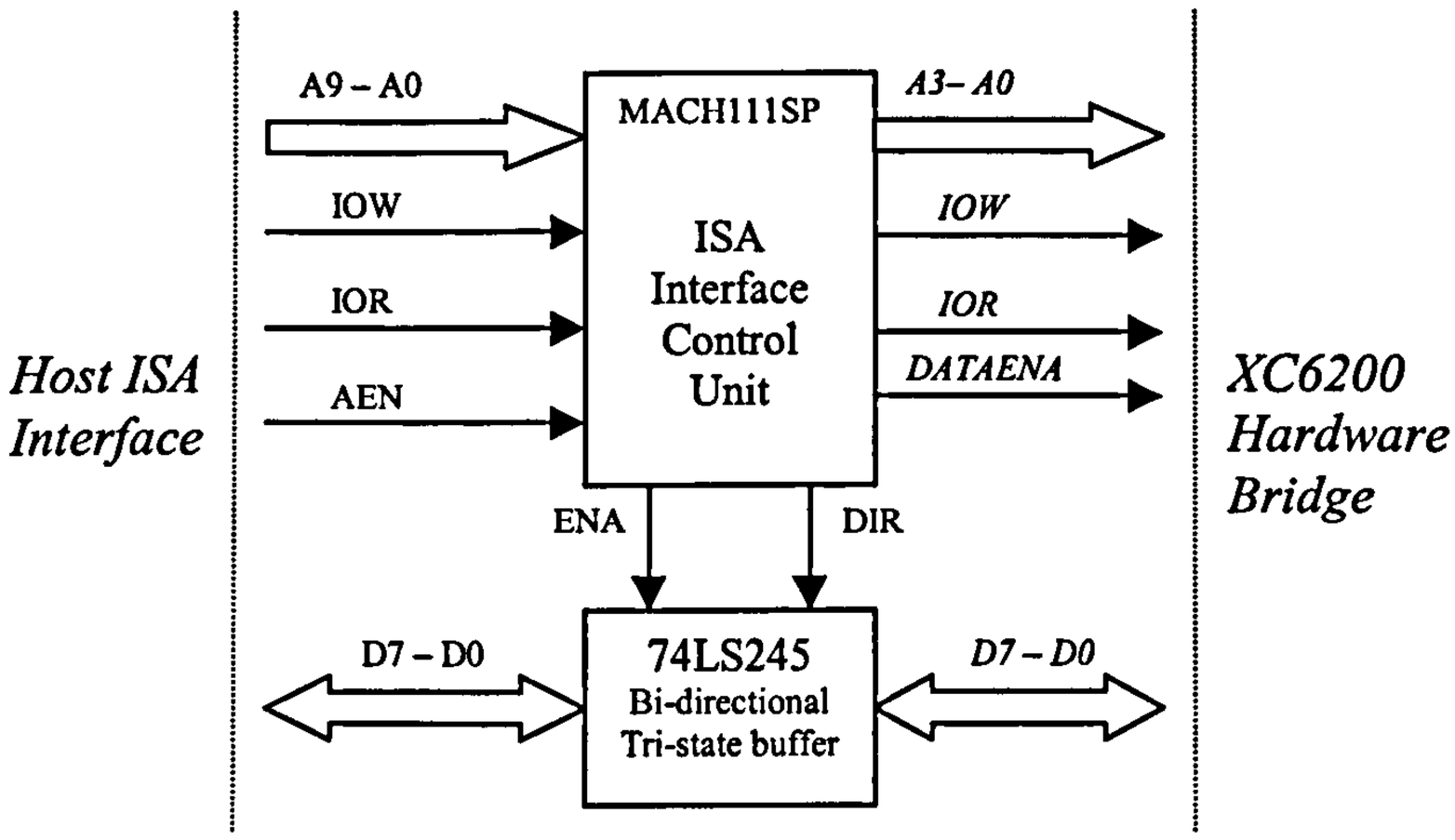
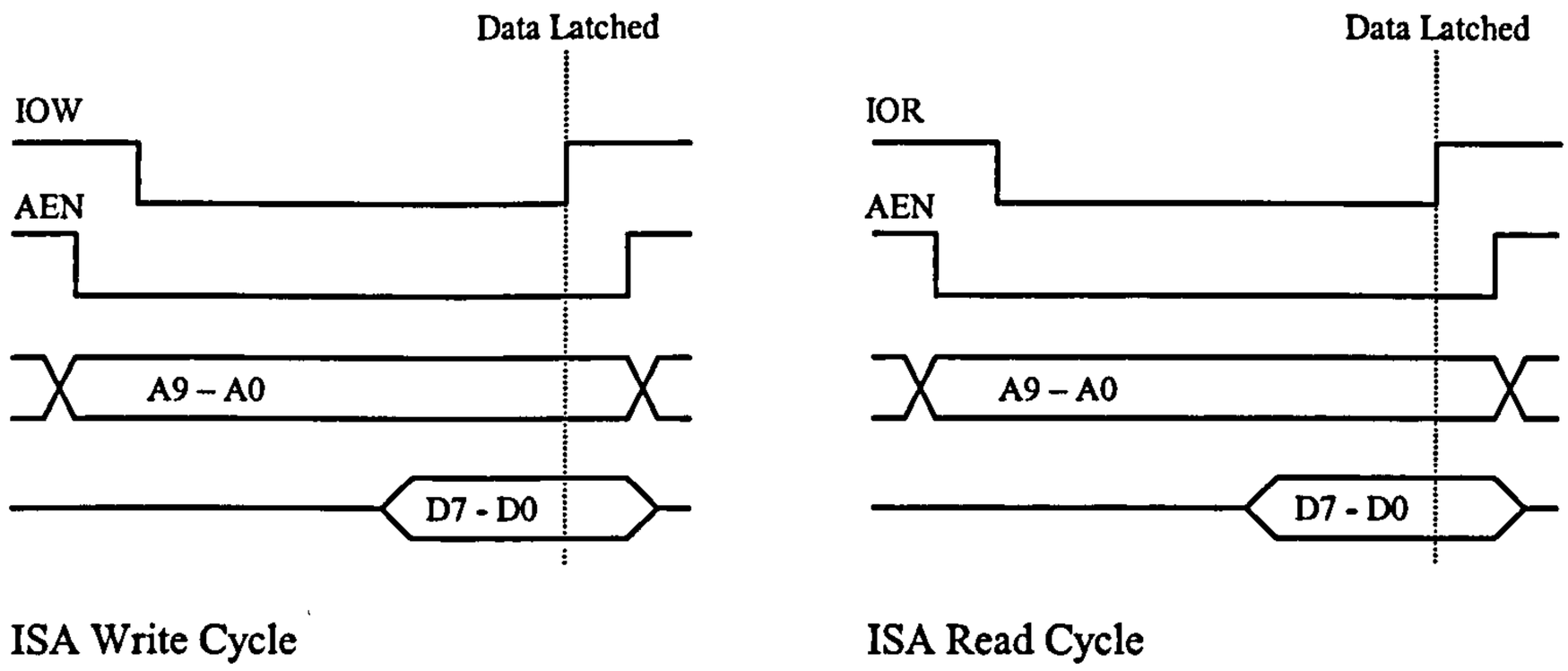


Figure 3.5 XC6200DS ISA Bus Interface

When addressed, ISA signals IOR and IOW (active low signals) determined whether a XC6200DS read or write operation was occurring. The control unit generated the enable (ENA) and direction (DIR) signal of the tri-state buffer accordingly. For both types of operations data was latched on the trailing positive edge of IOR or IOW respectively as shown in Figure 3.6. Control unit outputs *IOW*, *IOR*, and *A3–A0* are buffered versions of ISA bus signals, and routed to the XC6200 hardware-bridge.

The ISA control unit was implemented within a Vantis MACH111SP CPLD [6], chosen since it could be re-programmed using *In-System Programming* (ISP) techniques. The direction of the data-bus was controlled using a 74HC245 bi-directional tri-state buffer. Hardware configured within the CPLD was constructed using Vantis HDL PALASM (design is listed in *Appendix-IV)* with software development tool MACHXL. The control unit was initially tested using MACHXL simulation tools, and in-circuit using custom software that could read and write data to different address locations within the host PC.



ISA Write Cycle ISA Read Cycle

Figure 3.6 ISA Bus Interface Signals

3.3.2 XC6200DS Hardware-bridge

To achieve RTR, the XC6200 FPGA must be configured through its FastMAP™ interface enabling partial and dynamic configuration techniques to occur. The FastMAP™ interface consisted of a 32-bit data-bus and 18-bit address-bus. Configuration data was written to the XC6200 in address and data pairs. The address-bus width was fixed, but the data-bus could be 8, 16 or 32-bits wide.

The XC6200DS hardware-bridge provided the mechanism by which read/write operations issued by the host computer were instigated upon the XC6200 FPGA. Effectively this hardware managed the operation of XC6200 FastMAP™ interface. A block diagram of its architecture is shown in Figure 3.7.

The hardware-bridge register-set was accessed through the I/O address space of the ISA bus interface. The content of these registers contained either configuration data, or generated XC6200 related control signals, with all access performed at the ISA bus clock frequency (8.33MHz).

To access XC6200 address space, an 18-bit address was first generated. This took three ISA bus write operations (8-bit ISA interface), with data written to three separate registers. The registers accessed were determined through decoding the value of address bits *A3–A0*. The output of this decoder was only active when signal *DATAENA* indicated the XC6200DS was being addressed. Signal *IOW* ensured that only valid data was clocked into the registers.

Upon writing the third byte, all address bytes were latched into an external 18-bit address register. The output of this register was connected directly to the address-bus of the FastMAP™ interface. Depending upon the width of FastMAP™ data-bus one, two or four data bytes were then written to registers within the hardware-bridge. A further write to the hardware-bridge control register enabled the content of these registers to appear on the FastMAP™ data-bus, as well as generating a XC6200 write cycle through external state machine operation (XC6200 FastMAP™ Interface Controller).

An XC6200 read operation was similar except that once the address had been set, the control register instigated a XC6200 read cycle and the content of the FastMAP™ data-bus was then latched into hardware-bridge registers. For the host to access this 32-bit data, four ISA read cycles were generated, each addressing a separate 8-bit register. Signal *IOR* was used to ensure the validity of the data read by the ISA bus.

Through the control register other aspects of system operation can be governed. These features include assigning FastMAP™ interface control to internal XC6200 logic, and configuring the XC6200s primary clock source (*Gclk*). Dependant upon the particular design an external crystal oscillator could be selected as *Gclk*s source.

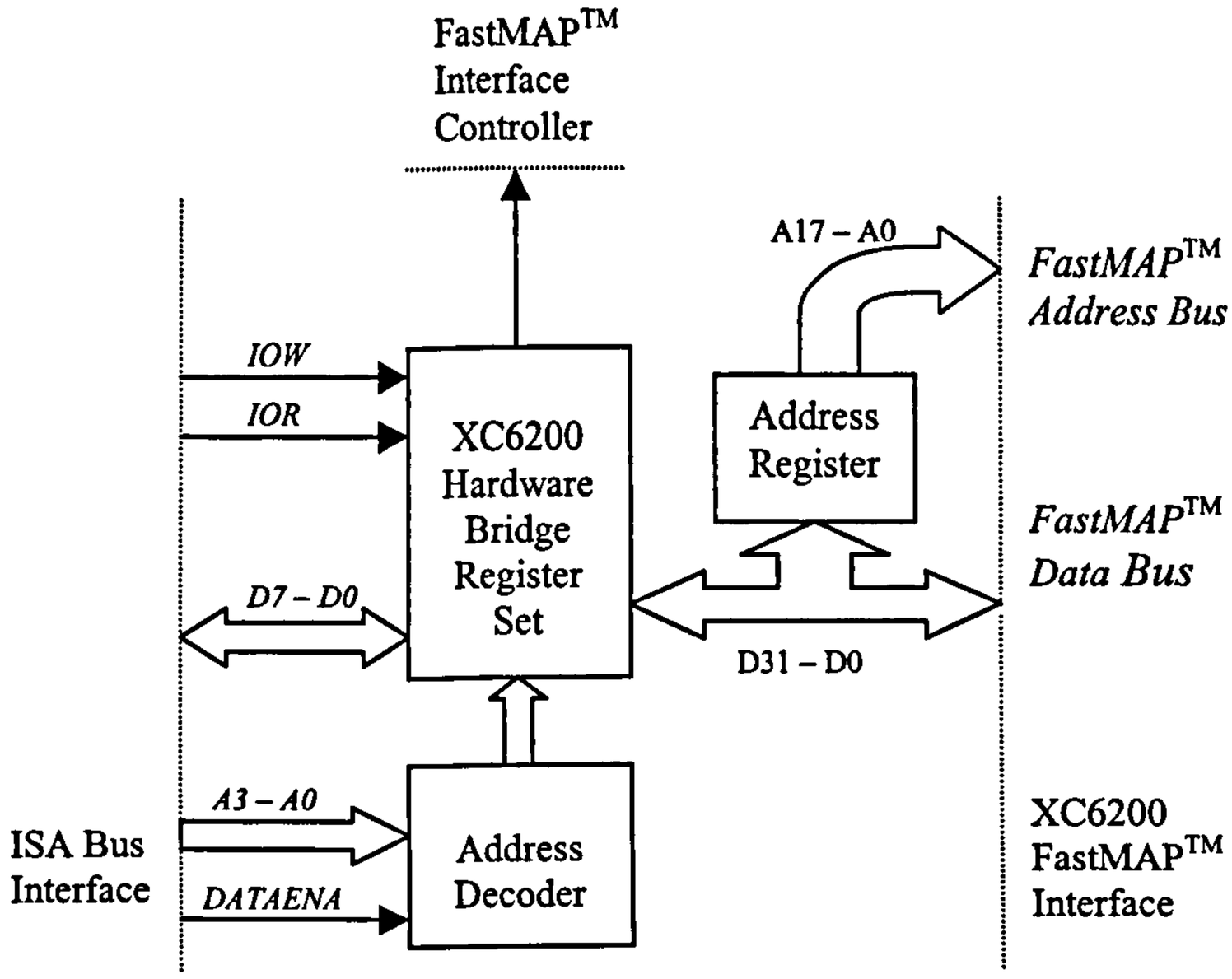


Figure 3.7 Block Diagram of XC6200 Hardware-Bridge

The XC6200 hardware-bridge was fabricated using a XC4005 FPGA [9] and three 74HC373 octal registers (address registers). External registers were required due to the limited available XC4005 I/O pins. The architecture was designed using schematic capture and VHDL design entry techniques within Xilinx Foundation development tools. Using this software the functionality of the control logic as well as the external latches and ISA interface hardware were simulated and assessed. After primary testing, in-circuit operation was verified using custom software tools developed upon the host PC.

During system development, configuration data was downloaded using a Xilinx XChecker Cable. The XC6200DS board also provided the facility to change the source of the XC4005 configuration from the Checker cable to a standard configuration PROM. The resultant design is illustrated in *Appendix-IV*.

3.3.3 FastMAP™ Interface Controller

Although the hardware-bridge instigated XC6200 memory access cycles, the FastMAP™ interface controller generated XC6200 signals *Gclk*, *CE* and *RW*. This hardware consisted of two state machines that generate the appropriate FastMAP™ control signals depending upon the type of access required. Their function is shown in Figure 3.8 with the basic XC6200 memory access cycles shown in Figure 3.9. Signal *Gclk* was the XC6200s primary global clock and all memory access operations had to be performed in synchronism with it. Therefore *Gclk* clocked the state machines.

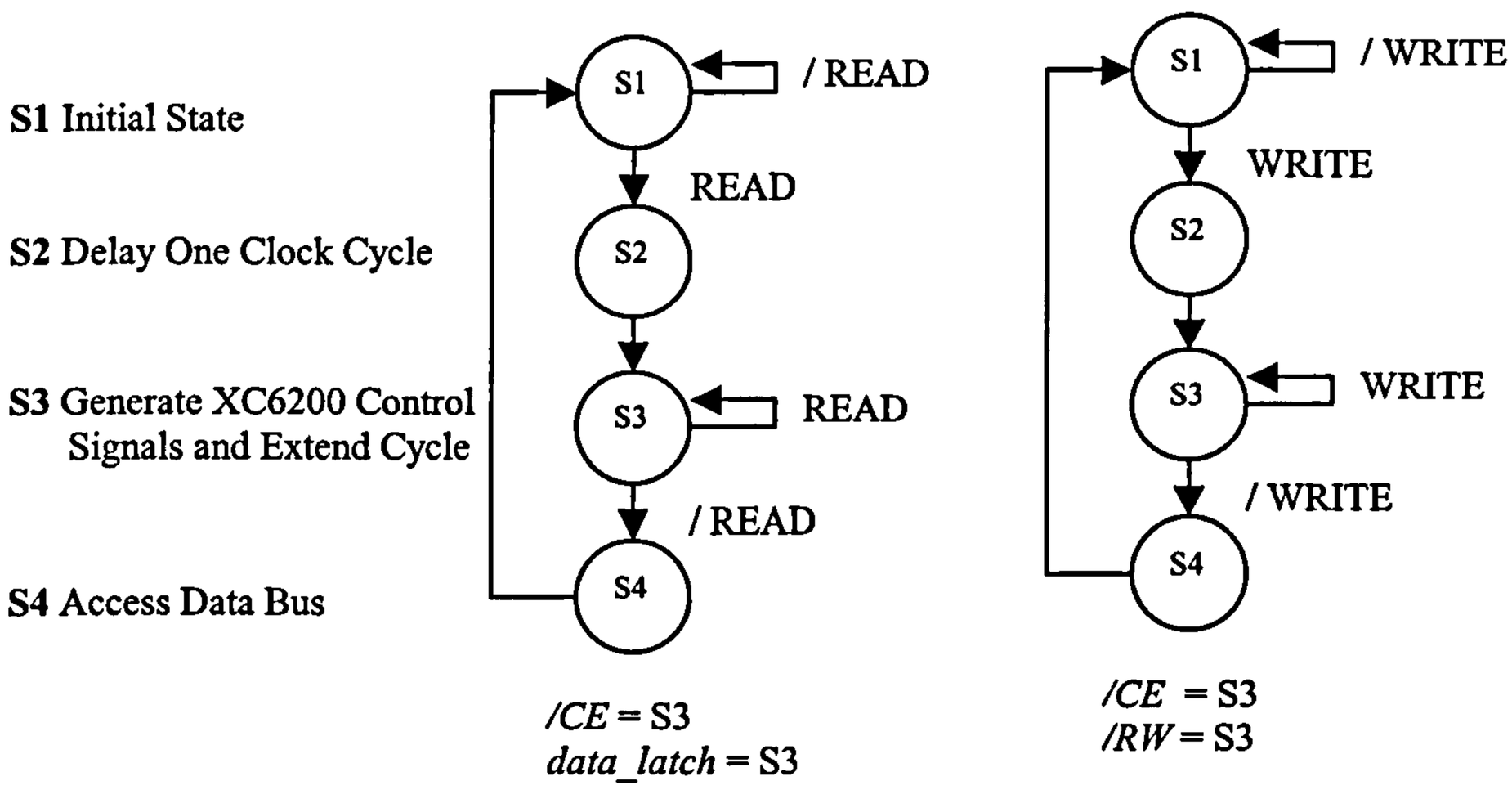


Figure 3.8 FastMAP™ Interface Controller State Machines

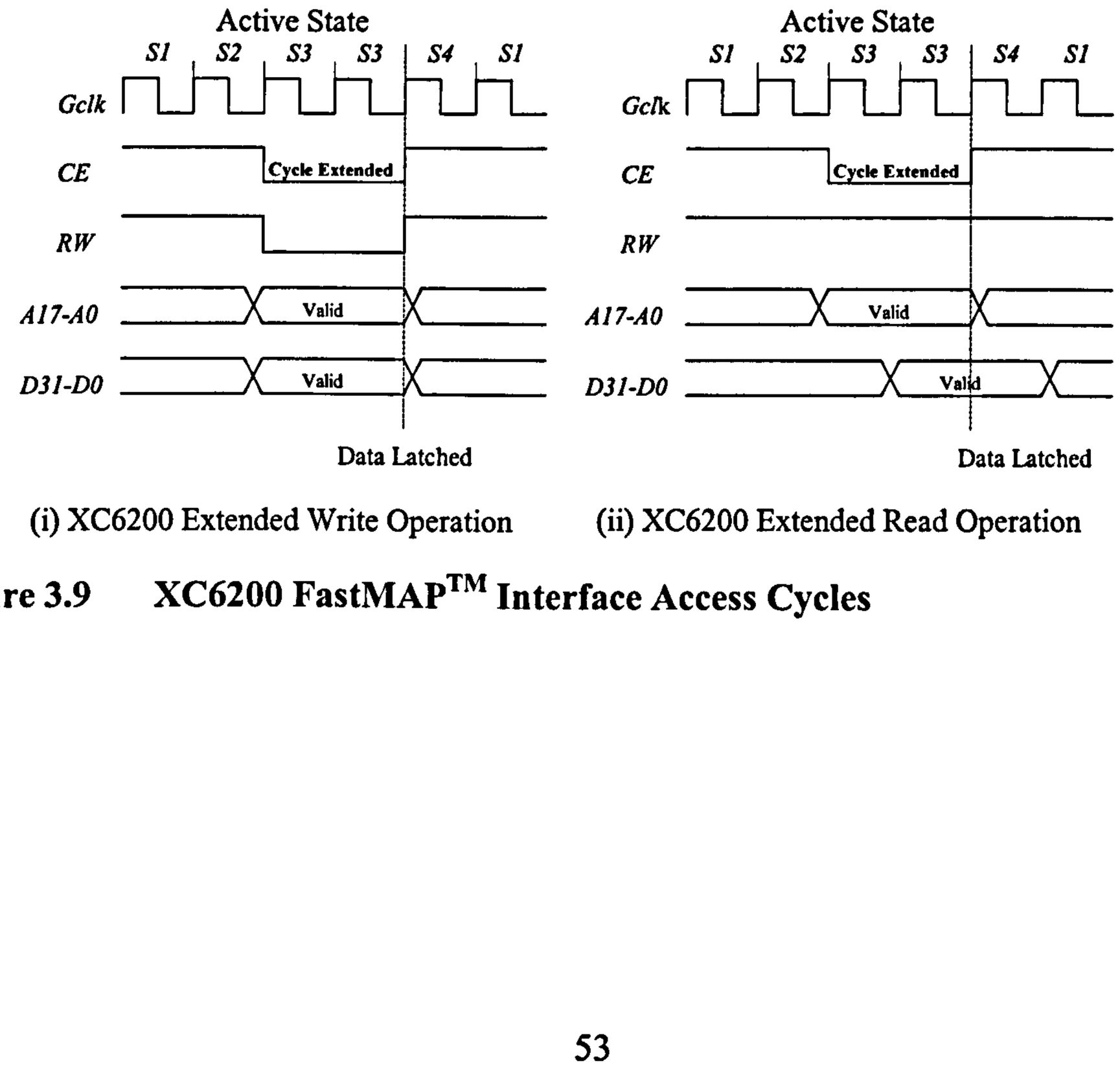
Figure 3.8 shows the structure of both state machines. These machines are similar, but have different input and output signals, with common output signals combined together. The signal *data_latch* was used to indicate when data read from the XC6200 FastMAP™ interface was valid, and could then be written to registers within the hardware-bridge.

The remote implementation of FastMAP™ interface control logic from the hardware-bridge was done to enable the XC6200 to function at higher clock frequencies than that of the ISA bus clock. This allowed interaction between XC6200 memory accesses

performed at XC6200 clock (*Gclk*) and ISA bus frequencies; Through experimentation, the maximum XC6200 flip-flop clock frequency was determined to be 88MHz. This gave an order of magnitude improvement over the ISA bus interface speed of 8.33MHz.

State machine operation was controlled using signals *READ* and *WRITE*, which were activated through the control register of the hardware-bridge. Using these signals XC6200 memory access cycles could be extended to remove differences in clock frequency between the XC6200 *Gclk* and ISA bus clock. To extend these cycles signal *CE* must be held low for more than one clock cycle. This operation corresponds to S3 of the state machines operation, as shown in Figure 3.9.

The FastMAP™ controller was implemented within a Vantis MACH111SP CPLD. The CPLD was programmed using PALASM, with the state machines described using net-lists. Simulation was conducted within MACHXL development environment, with further debugging performed in-circuit using custom software and traditional electronic test equipment. The resultant PALASM design file is listed in *Appendix-IV*.



(i) XC6200 Extended Write Operation (ii) XC6200 Extended Read Operation

Figure 3.9 XC6200 FastMAP™ Interface Access Cycles

3.3.4 XC6200ADS Development Software

The *XC6200 Application Development Software* tools (XC6200ADS) were developed to provide a platform for XC6200 CTR or RTR techniques, access of XC6200 address space and RTR application development. The structure and function of these custom tools is shown in Figure 3.10.

When the software was executed, its first task was to detect if a XC6200 FPGA was present upon the XC6200DS, and then determine the device type through reading the XC6200s *Device Configuration* register. During this operation the functionality of the XC6200DS hardware components were verified. This was because before any XC6200 register accesses could occur, a string of fifteen bytes had to be first written correctly to the XC6200s *Identity* registers. The XC6200 FPGA device type present also had to be determined since the address locations of control registers differed in XC6200 family members. Next, the user menu was displayed, which listed utilities to access XC6200 configuration registers, access user registers configured within the XC6200s CLC array, execute user defined function macros, and download configuration data using both CTR and RTR methods.

XC6200 configuration data was generated by XACT6000 software in the form of text files (known as *cal* files because of their filename extensions), consisting of address and data pairs. To program the XC6200 cal files were first parsed to remove text comments, and then address and data pairs converted from word to byte formats. The resultant file was then downloaded to the XC6200 using XC6200ADS functions. For both CTR and RTR techniques CLC configuration delay was calculated to be 2.4μsec. However, this value did not take into account host computer interrupt operations, and was measured to be in the region of 225μsec to 374μsec.

To perform RTR configuration both the existing and new FPGA configuration cal files were required. Initially both files had any XACT6000 text comments removed, and then RTR reduction occurred. This was a novel process that reduced the volume of

configuration data required for RTR to a minimum. Essentially, the two files were compared and only the differences between them downloaded to the XC6200 using partial and dynamic configuration techniques. To aid debugging, all ISA bus operations performed during configuration were written to a text file.

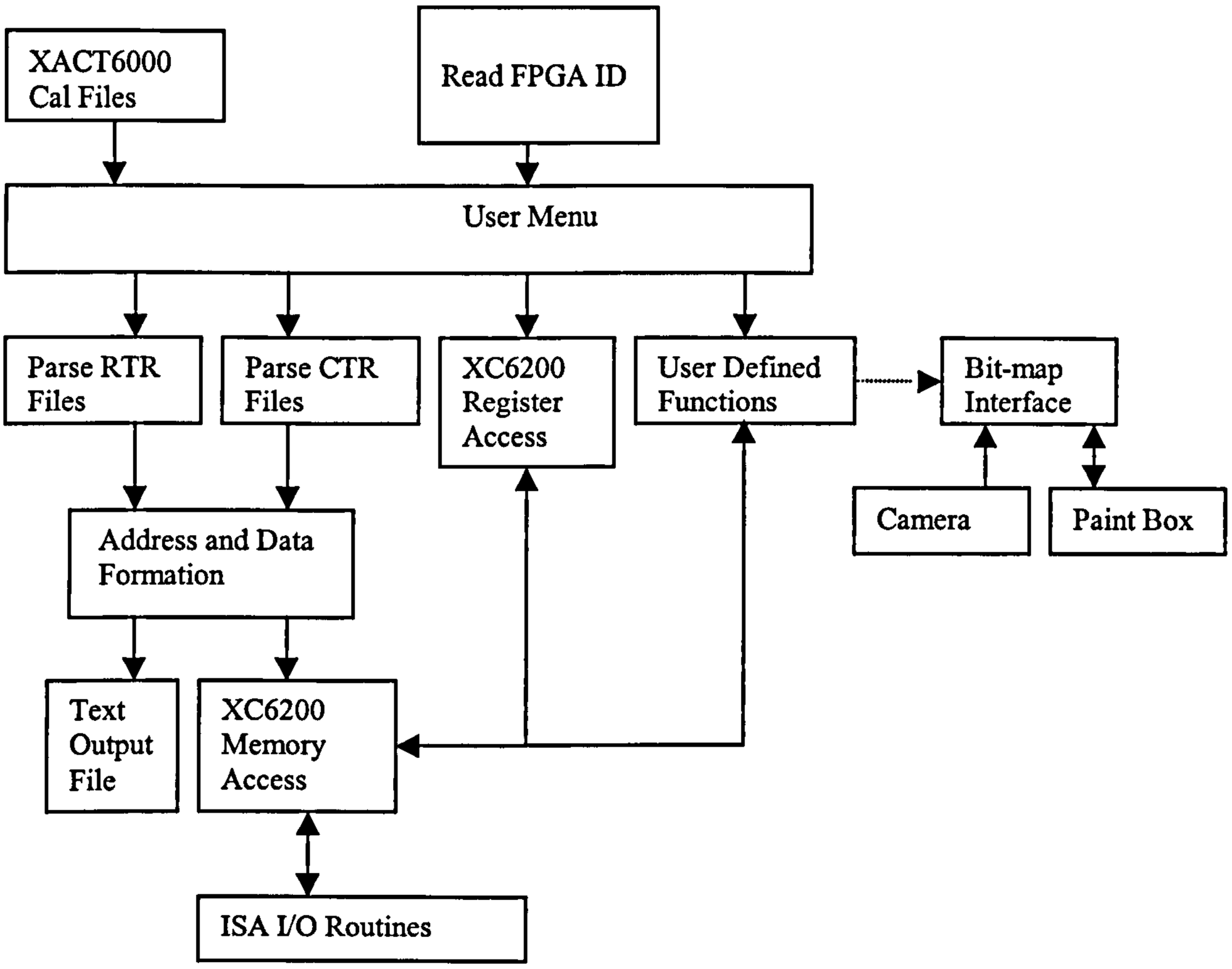


Figure 3.10 Structure of XC6200ADS Software Tools

The contents of registers within the CLC array and XC6200 control registers could be addressed and accessed. Using XC6200ADS functions XC6200 control registers could be accessed directly by entering their respective address location within the XC6200s memory map. Prior to accessing user registers configured within the CLC array, XC6200 *Mask* and *Map* control registers values had to be set accordingly. This was achieved through addressing the XC6200s Mask/Map control registers directly and updating their contents.

Mask and map register contents were determined by the row position and width of user configured CLC registers. Once configured, the column number where the user register was located was then entered into the XC6200ADS. A practical example of this mechanism is shown in *Section-4.1*.

The manual configuration of Mask and Map registers was time consuming. To speed up these processes user macros could be defined within the program source code. Macros were also constructed that enabled bit-map images to be downloaded to hardware within the XC6200. Results could then be written back to the host and displayed using Microsoft Windows Paint Box. Most images used were computer generated using Paint Box, but real images could also been obtained using a web camera.

The software itself was written using Microsoft Visual C++ development environment. For simplicity a text based and menu driven *Graphical User Interface* (GUI) was used, since the software was an evolving tool and constantly updated to facilitate debugging of new designs. A screen shot of the XC6200DS is shown in *Appendix-VII*.

3.3.5 XC6200 FPGA Hardware Development Cycle

The development process used to implement hardware within the XC6200 is shown in Figure 3.11. Designs were constructed using the HDL VHDL. Initially, the VHDL code was written to IEEE1164 standard to allow Xilinx Foundation simulation tools to be used to debug the design. Once the design had been proved functional, XC6200 specific VHDL attributes, including gate primitive libraries, placement and routing constraints were inserted.

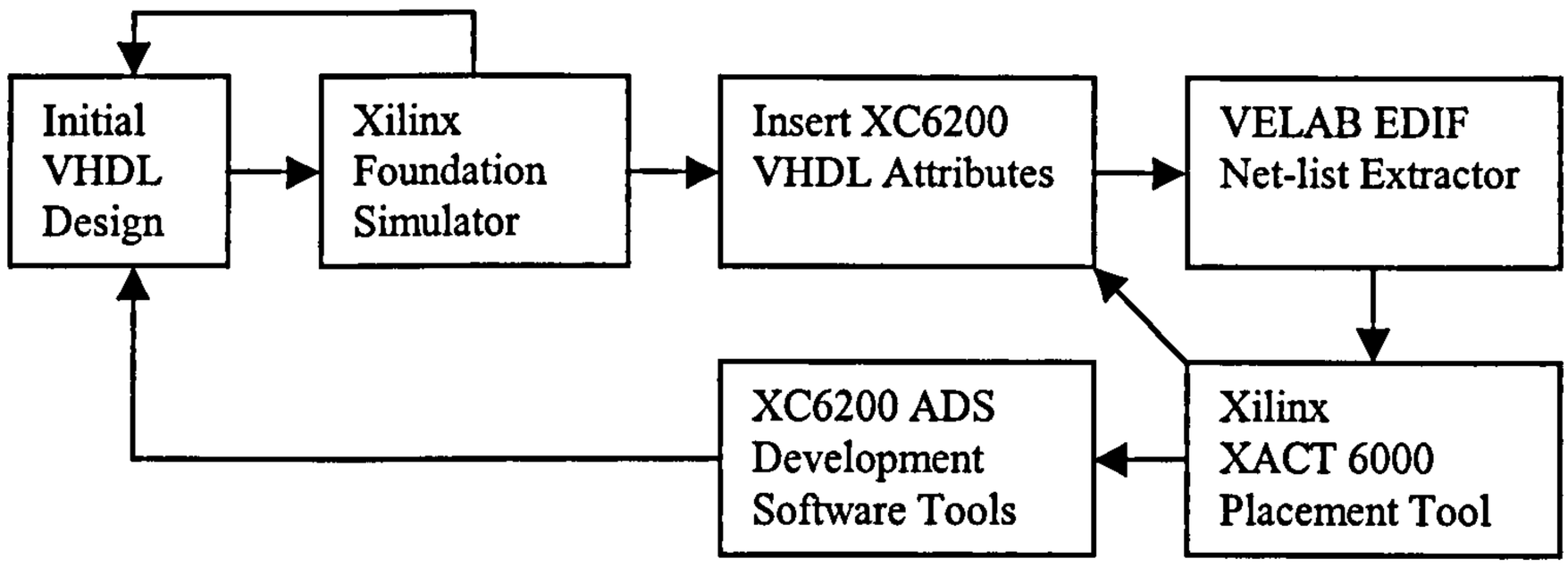


Figure 3.11 XC6200DS Hardware Development Cycle

Using software tool *VELAB*, an EDIF format net-list was generated, which was then read by the Xilinx XACT6000 XC6200 placement tool. This software was used to map and route the design (EDIF net-list) upon the XC6200 FPGA, and generate the cal configuration file (containing XC6200 configuration data address/data pairs). The XC6200 FPGA was then configured and evaluated in circuit using XC6200ADS user macros.

3.4 XC6200DS Configuration Topologies

The XC6200DS was constructed to provide a platform for the development of RTR applications, and insertion of such architectures within the routing and node structure of existing parallel processing architectures. Component PCB positions, system operation, and I/O resources of the XC6200DS were therefore designed with flexibility and multiple functions in mind. These design parameters enabled the XC6200DS hardware to operate in four modes. These were:

i. *Dynamic Prototype Environment*

ii. *TMS320C40 Dynamic Coprocessor*

iii. *TMS320C40 Communication Channel Routing Hub*

iv. *Self-configurable RTR Hardware*

A description of each mode is given in the following sections.

3.4.1 Dynamic Prototype Environment

The default XC6200DS configuration was a dynamic prototyping environment. In this topology, the XC6200DS and C40 parallel processing architecture were not attached and operated independently. This mode of operation facilitated XC6200 FPGA hardware development since the FPGA could be configured and debugged using its FastMAP™ interface. The system architecture is shown in Figure 3.12.

To assist in evaluating real-time performance of FPGA based processing hardware, a SRAM memory module could be connected to the XC6200DS using either one of the two I/O 40-pin IDC connectors. The memory module consisted of 256-kbytes of SRAM, connected to the XC6200DS through IDC ribbon cable. All control signals (*RW*, *CE*, and *OE*), address (*A18-A0*), and data-bus content (*D7-D0*) were controlled and generated by hardware configured within the XC6200. Through registers configured within the CLC array, the host computer could access the contents of this memory.

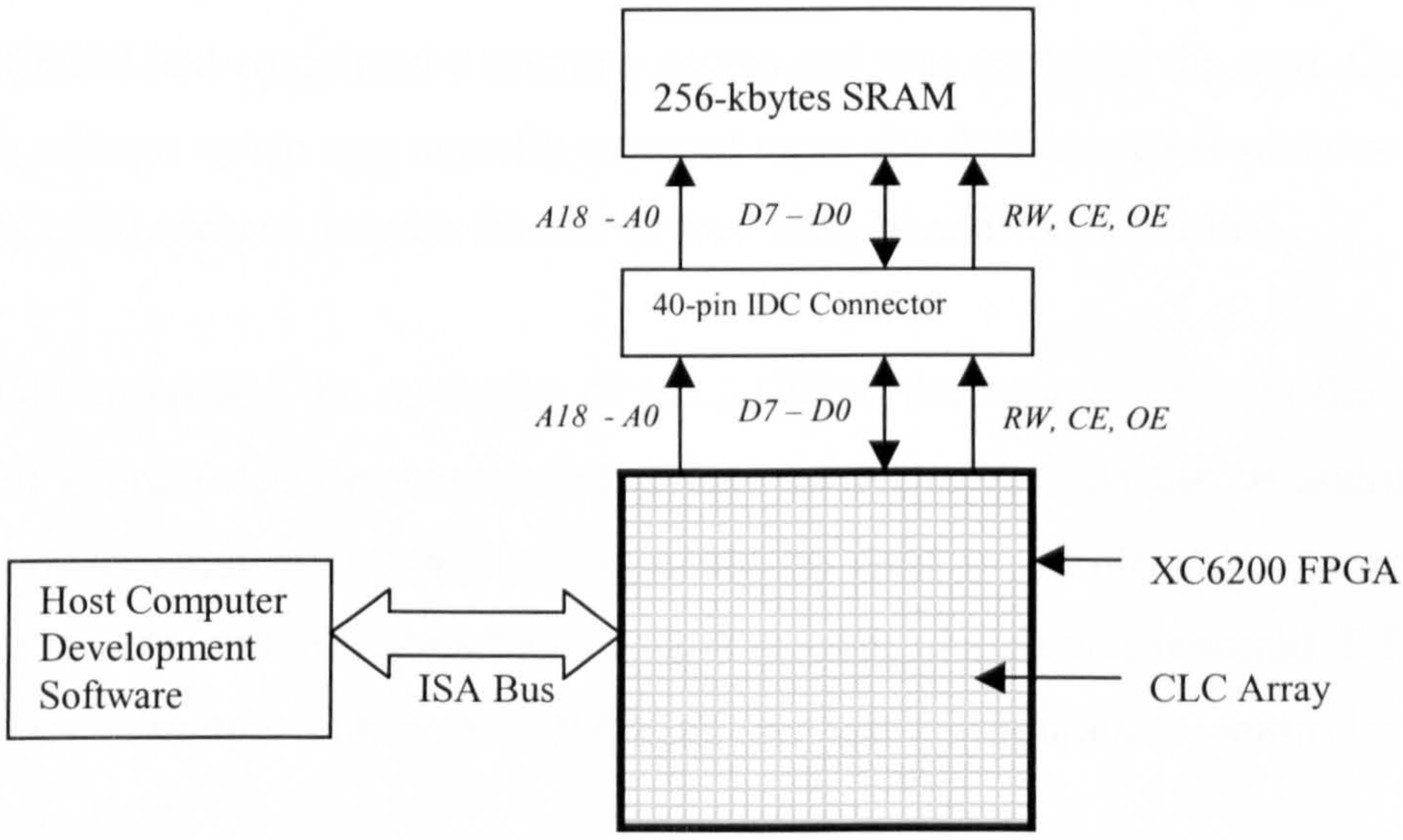


Figure 3.12 Dynamic Prototype Environment

Through the interface, operands could be downloaded to the memory module, processed by XC6200 hardware, results stored back in memory, and then uploaded upon request to the host computer.

3.4.2 TMS320C40 Dynamic Coprocessor

The second XC6200DS configuration connected the XC6200DS to the Global bus interface of a C40 DSP. In this topology the FPGA appeared as a loosely coupled coprocessor to the C40. This is illustrated in Figure 3.13, with actual XC6264 footprints contained within *Appendix-VI.* The memory map of the Global bus could be divided into two regions accessible by signals *GSTRB0* and *GSTRB1*, with each region having the ability to posses a separate memory configuration if required. The TIM-40 standard dictated that signal *GSTRB1* and its related signals must be used to access external peripherals. To configure this interface, the content of the *Global Memory Interface Control Register* (GMICR) required updating.

The Global interface was flexible and can be configured for different memory capacities, page sizes, and speed of operation. To perform XC6200 memory mapped transfers, *GSTRB1* of the Global Interface occupied address range 80100000_{16} to $FFFFFFFF_{16}$. Memory access control was governed using signal *RDY1* generated by hardware configured within the XC6200. This signal indicated when hardware within the XC6200 had completed a memory access and was ready for the next. Only a subset of this address space was actually required since the data-bus used was one byte wide, and XC6200 address decoder limited to four bits (16 address locations).

The C40 controlled the operation of the XC6200 during run-time via the output of a four-bit address decoder configured within the XC6200. The use of address decoder outputs was application specific and could be used to enable I/O registers, generate control signals, and indicate when XC6200 reconfiguration was required. RTR could be conducted through either XC6200ADS or self-configuration controller operation (*Section- 3.4.4*).

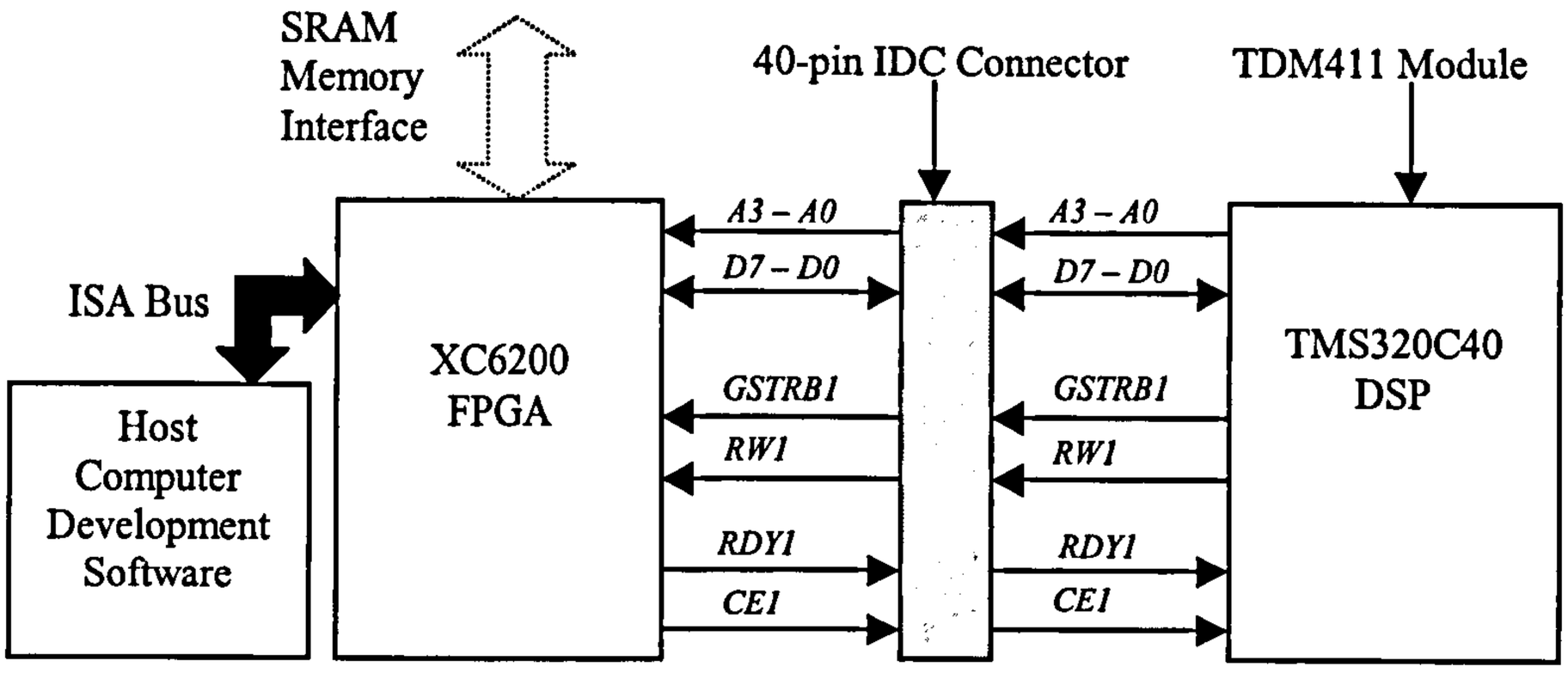


Figure 3.13 TMS320C40 RTR Coprocessor Configuration

The TIM-40 (C40) and XC6200 hardware interface was formed using either one of two XC6200DS 40-pin IDC connectors. Depending on the application, the unused connector could then be configured to support additional external memory. In their original forms, TDM411 modules did not support IDC type connections. Forty-pin IDC connectors were therefore attached to the modules and connections made to the appropriate C40 pins. It had originally been proposed to use the full width of the Global data-bus (32-bit), but difficulties encountered during this conversion process prevented this. Therefore 8-bit rather than 32-bit interfaces were used, which was perfectly good for prototyping development. With *Gclk* set to 8MHz, a C40/XC6200 I/O bandwidth of 2Mbytes/sec was obtainable (maximum 4.14Mbytes/sec, *Gclk* @16.57Mhz).

3.4.3 TMS320C40 Communication Channel Routing Hub

The third configuration of the XC6200DS facilitated the insertion of RTR hardware within the routing topology of the C40 parallel processing system. This hardware provided additional routing and processor resources as determined by the application.

The interconnection topology was formed using C40 DSP communication channels. Each TDM411 C40 module possessed six communication channels. Four were used to create a hardwired interconnection topology within the TDMB412 motherboard, whereas the remaining two were brought out through connectors situated on the TIM-40 motherboard (using 20-pin IDC 2-mm pitch type connectors). Through external cabling

these channels could be attached to the XC6200, with each FPGA having nine of these connectors situated around it. This distribution of I/O resources enabled efficient use of the CLC array when configuring routing channels within the FPGA. A block diagram of the XC6200 used as a routing-hub is shown in Figure 3.14.

To insert a routing-hub within the C40 communication topology, XC6200 FPGA hardware was required to interact with C40 communication channel *Port Arbitration Units* (PAU) *Finite State Machines* (FSM). XC6200 hardware was also required to manage 32-bit data transfers occurring in four-byte blocks. To manage data transfer two distinct XC6200 routing-hub control mechanisms were developed.

The first mechanism isolated the operation of the communication channel transfer protocol from the remainder of the XC6200s configuration. The channel interface consisted of a state-machine to govern the transfer and a FIFO to store four-byte data blocks. Hardware within the XC6200 accessed and instigated data transfer between the C40 and XC6200 via the FIFO.

The second mechanism used a self-arbitrating control unit. Data transfer occurred directly between registers within a XC6200 design, and not through intermediate FIFO buffers. The bytes constituting C40 communication channel words were transferred individually, rather than in four byte blocks as in the first control mechanism; The merits, comparisons, and detailed description of both communication control mechanisms are described in *Section-7.2.*

The C40 communication channels are bi-directional. It was originally intended that the data transfer direction could be reversed during run-time. Whilst implementing this function, it became apparent that XC6200 operational speeds were not fast enough to facilitate this function. During the directional transfer process, it was possible for both the XC6200 and C40 to be driving the same signals. This limited the maximum system flexibility, but was not a major problem since each C40 could exhibit bi-directional data transfer through using two communication channels.

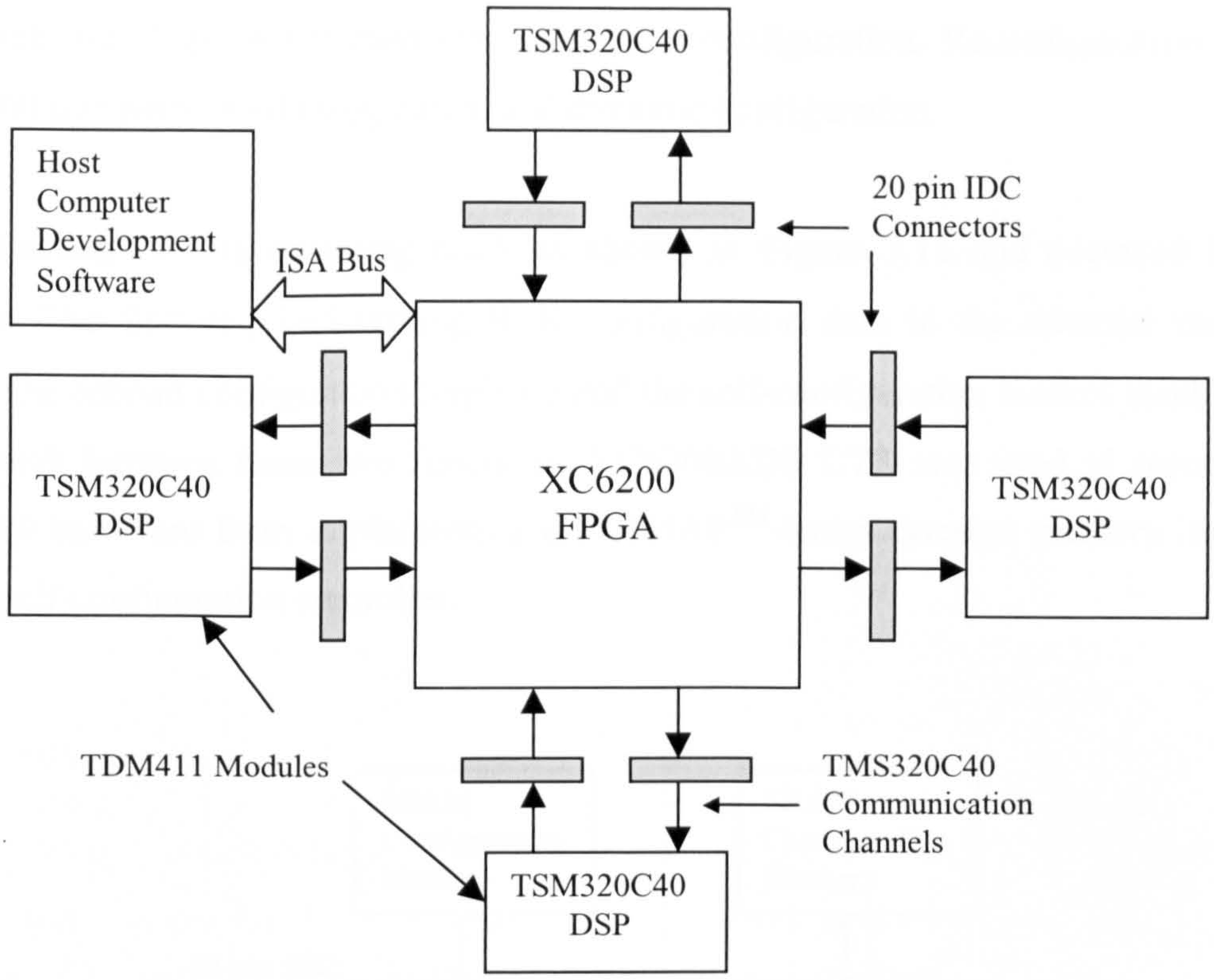


Figure 3.14 TMS320C40 Communication Channel Routing Hub

3.4.4 Self-Configurable RTR Hardware

The final XC6200DS mode of operation was self-configuration. In this mode, RTR was instigated within the CLC array (not off-chip), and performed through the FastMAP™ interface without intervention of the hardware-bridge. Configuration data was stored in external memory connected using a modified 40-pin IDC XC6200DS I/O socket. Since the original XC6200DS PCB required modification to support this, only one board supported this configuration mode.

Upon power up, RTR configuration data for up to sixteen different configurations was downloaded through the FastMAP™ interface. Using hardware configured within the XC6200, this data was written to a 256-kbytes external configuration memory. The address boundaries of individual configurations stored in memory were calculated, and later used by the control logic to activate a particular configuration for RTR. The self-

configuration control mechanism was self-contained within the XC6200 configuration, in which user logic determined the next active configuration. Reconfiguration of the XC6200 was performed using partial and dynamic configuration.

The function of this operating mode is shown in Figure 3.15 and occurred in two phases. The first required writing RTR configuration data to the external memory, whilst the second configuration implemented the self-configuration control mechanism. To switch between these two functions, XC6200ADS CTR was used to reconfigure XC6200 hardware from implementing a FastMAP™-based external memory interface to the self-configuration controller.

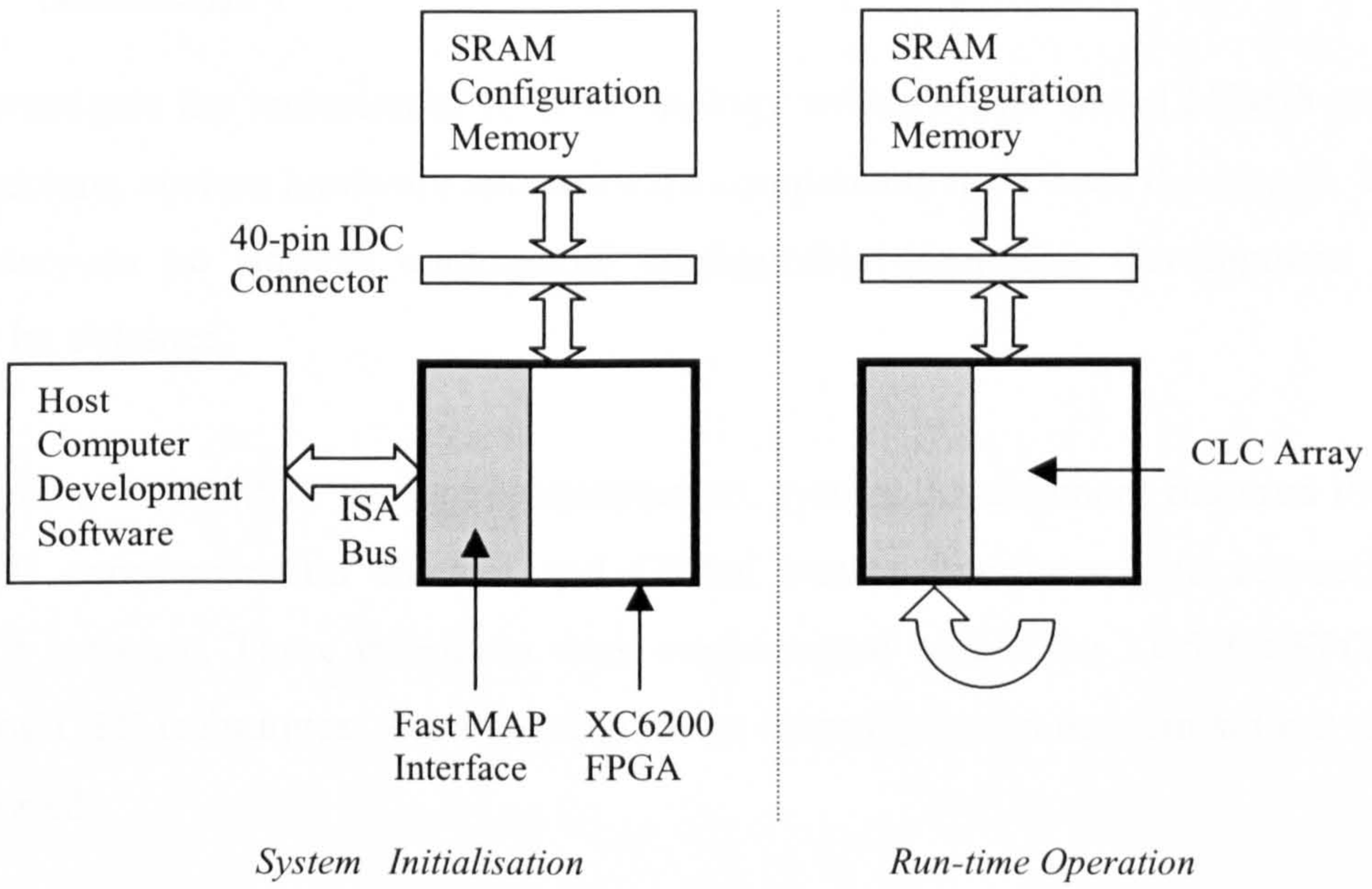


Figure 3.15 Self-Configuration Mode

FastMAP™ bus conflicts between the hardware-bridge and the self-configuration controller were prevented through the hardware-bridge control register. To read XC6200 registers using the host PC, the hardware-bridge required control of the FastMAP™ interface.

RTR configuration controlled by the host computer occurred at ISA bus operating speeds. To increase the speed of configuration, the XC6200 global clock frequency

must be increased (*Gclk*). This could be accomplished only when self-configuration was performed since ISA bus transfers were not required. Through the hardware-bridge control register, the source of *Gclk* could be selected as either an external crystal oscillator or the ISA bus clock. Using XACT6000 placement tools, the maximum frequency of *Gclk* was calculated to be 6.99MHz. The maximum delay however related to a non-critical signal and design operated normally at 8MHz. Configuration delay at 8MHz was calculated to be 1.8µsec per CLC and measured using external apparatus at 1.88µsec. Placement of the self-configuration control unit within a XC6264 FPGA is shown within *Appendix-VI*, with a description of it internal operation provided in *Appendix-IV-4*.

3.5 Summary

To investigate the inclusion of RTR technology within a DSP based MIMD processing architecture, custom hardware and software components have been developed. This was necessary as no suitable commercial configurable computing development systems could be obtained.

As well as XC6200DS hardware construction, system development required the design of C40 communication channel and Global busses interfaces, and respective C40 control software. These interfaces were implemented within the XC6200 FPGA using existing CTR techniques. A self-configuration reconfiguration mechanism has also been developed.

The philosophy reflected in the design and function of the system was for as simplistic and flexible operation as possible, since the XC6200DS was designed for multiple system configurations. This is apparent not only in hardware design, but also in the development of software tools and the resultant XC6200DS application development cycle.

Development of XC6200ADS tools has increased the versatility and suitability of the XC6200DS to be used for dynamic hardware prototyping. XC6200ADS functions permit real-time in-circuit probing of XC6200 FPGA designs, reduce the volume of configuration data required for dynamic configuration, as well as providing the facility for user defined macros. These have been configured to perform tasks such as analysing data transfers within the XC6200DS routing-hub, and the transferral of operands from the host PC to the XC6200DS and TIM-40 nodes.

From the project onset it was evident that the operational characteristics of RTR semiconductors commercially available (XC6200 FPGA family) were limited. Both logic gate capacity and operating frequency when compared to existing cutting edge technologies were poor. Therefore the integration and implementation of RTR hardware rather than trying to achieve raw processing power was the design goal.

The C40 MIMD system used to provide the multiprocessor environment was ideal since it facilitated the inclusion of additional hardware within both its node and routing topologies. Dynamic hardware has been inserted into the existing MIMD topology through using existing C40 communication channel connections, and the creation of custom sockets upon TIM-40 processor modules.

Development of the XC6200DS has provided the platform upon which RTR hardware can be developed, evaluated, and included within both the processing and routing topologies of high-performance parallel processing environments.

Chapter 4

XC6200 FPGA Hardware Investigation

Introduction

This chapter describes the development of XC6200 FPGA hardware implementation and in-circuit verification techniques. This work was conducted to generate design and test procedures, and a knowledge base upon which further XC6200 based DSP RTR coprocessor hardware and routing topologies could be constructed.

Section-4.1 describes XC6200 hardware verification techniques developed, with a practical example demonstrated. *Section-4.2* describes the development of both static and dynamic hardware implementation methodologies. These were determined through implementing fundamental processing hardware structures within the XC6200.

Section-4.3 details conclusions determined through implementing and evaluating hardware within the XC6200 FPGA using the XC6200DS and XC6200ADS. These conclusions provide the basis for operating procedures used in further XC6200 based hardware development.

4.1 XC6200 Design Verification

To develop RTR hardware using the XC6200DS, system implementation and hardware verification strategies have been developed. Using these techniques XC6200 hardware developed throughout the research program was designed and developed initially using the XC6200DS in its prototype environment configuration. This mode minimised component placement restrictions within the XC6200 and facilitated the insertion of test signals through interface connectors situated around the XD6200DS.

XC6200 FPGA designs were constructed using VHDL and initially evaluated using Xilinx Foundation's VHDL Simulator. Once proved functional the design was mapped onto the XC6200 architecture (using XACT6000) and analysed. The experiments

conducted to develop placement techniques for this task are described in *Section-4.2* of this chapter. The current section details the development and evaluation of in-circuit test procedures for hardware configured within the XC6200.

Within existing semiconductors technology, techniques such as JTAG (IEEE 1149.1 standard) [68] can be used to verify the operation and configuration of hardware. The XC6200 family data sheet [32] stated that the FPGA was compatible with the JTAG standard using library macrocells. However, these macros could not be obtained for hardware developed using VHDL. To debug active XC6200 designs internal signals had to be analysed through external pins or accessed through registers within the design via the FastMAP™ interface.

The FastMAP™ interface allowed accesses to the XC6200 SRAM configuration memory and user-defined registers within the CLC array. Register access was determined through the contents of Map and Mask control registers within the XC6200.

Figure 4.1 shows a simplified diagram of a design being evaluated within the XC6264. Included in Figure 4.1 are the relative row and column positions of the design within the CLC array. These positions must be known to correctly configure the XC6200 control registers.

The CLC design consists of three sections. These are a 4-bit RPFD register (column 10), the design under test (columns 11 and 12), and a 4-bit FDC register (column 13) with open *q* outputs. RPFDs (*Read-Only Protected Flip-flop Device*) are XC6200 specific components that enable data to be written to logic configured within the CLC array through the FastMAP™ interface. The function of an RPFD can be considered similar to the standard FDC VHDL entity (D-Type *Flip-flop Device* with *Clear*) except that the *d* input cannot be accessed via user logic but instead is tied internally to FastMAP™ interface control logic.

The XC6200 primary clock *Gclk* is used to clock RPFD and FDC registers accessed through the FastMAP™ interface. A further constraint imposed was that for concurrent access of register bits, CLCs forming the register had to be stacked vertically in the appropriate bit positions.

Each input test stimulus was written to the design through an RPFD register, via the FastMAP™ interface. Results generated by the design were first latched into an FDC register, and then read using the FastMAP™ interface.

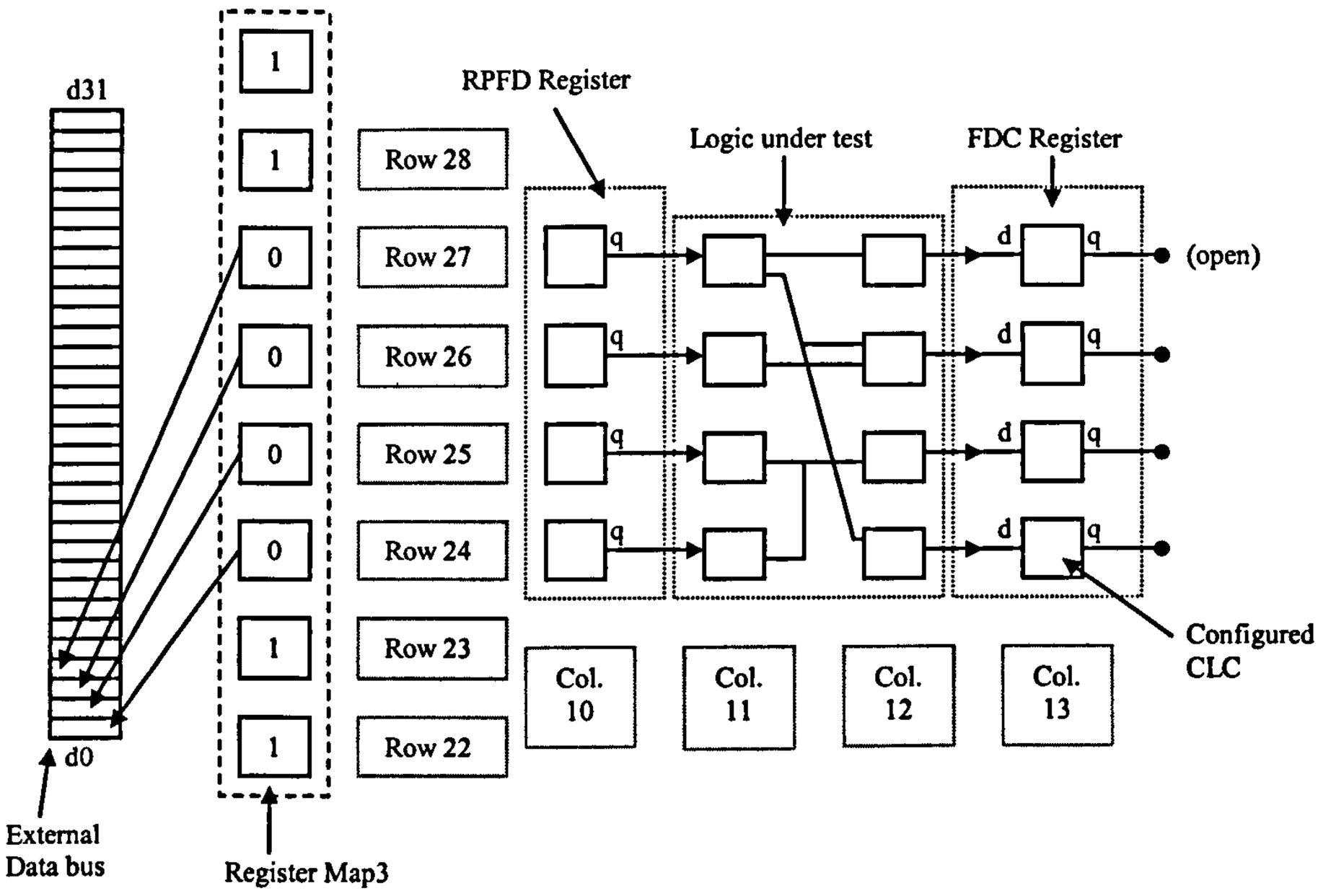


Figure 4.1 XC6200 In-Circuit Hardware Verification Method

The first XC6200 control register group configured were the *Mask* registers (*Mask-0*, *Mask-1*, *Mask-2*, and *Mask-3*). These registers did not select the actual register, but instead mapped the content of internal buses (data-paths) upon the external FastMAP™ data-bus. The address locations of Mask registers varied for different XC6200 family members.

In the example shown in Figure 4.1, a 4-bit data-path was required between the external data-bus and CLC array. This was because the RPFD and FDC registers were only 4-

bits wide. This data-path could be mapped onto any bits of the FastMAP™ data bus, but for practicality the four least significant bits (*d3-d0*) were chosen.

Bit-0 of *Mask-0* corresponded to bit *d0* of the FastMAP™ interface data-bus. Within the Mask registers, bits allocated to form CLC data-paths connected to the FastMAP™ data-bus were set to logic-zero. Mask register bits unused were set to logic-one. The Mask register configuration required is shown in Figure 4.2.

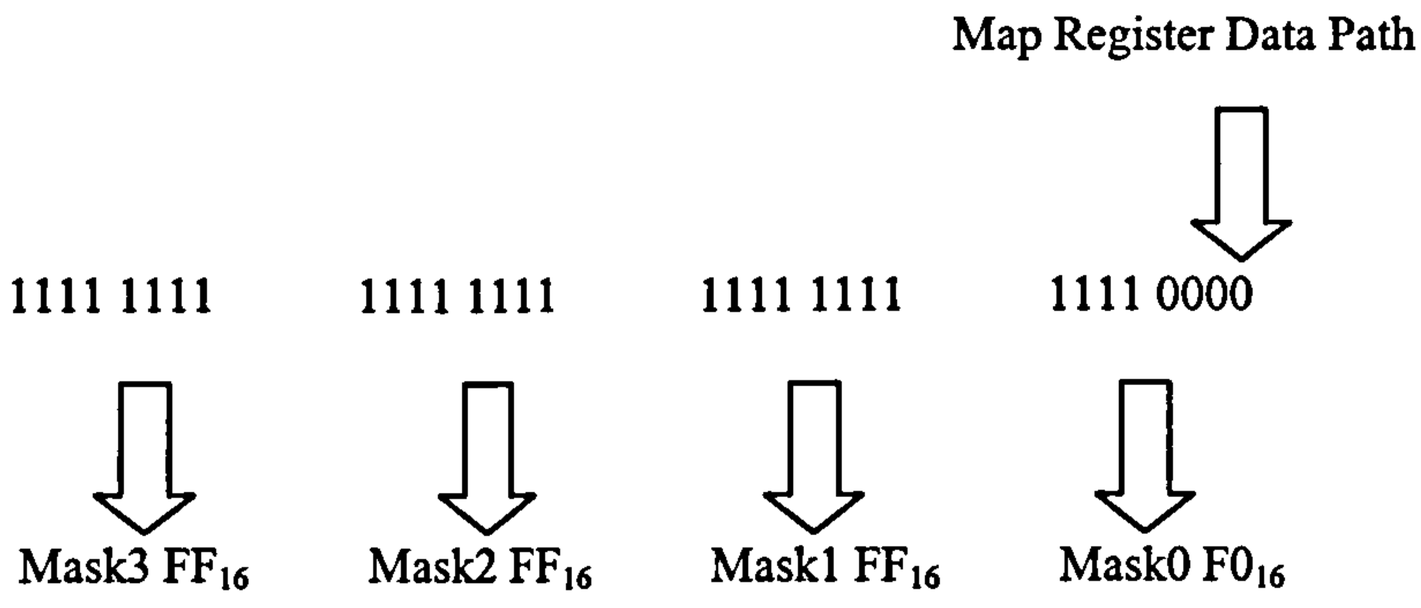


Figure 4.2 XC6200 Mask Register Configuration Mechanisms

The second group of XC6200 control registers configured were the *Map* registers. The number and address location of these registers were again dependant upon XC6200 FPGA family member used. The XC6264 FPGA (128x128 CLC array) used in the example had sixteen 8-bit Map registers (*Map-15 – Map-0*). The Map registers selected the row of the CLC array to be accessed (8-bit x 16 registers = 128 rows). Individual bits within each Map register corresponded to row positions within the XC6264. For example *Map-0*, bit-0 accessed row-0.

To configure the Map registers correctly the row position of the RPFD and FDC registers must be known. The example given in Figure 4.1 required register access on rows 24 to 27 inclusive. CLC row position 24 corresponds to *Map-3*, bit-0, therefore the lower nibble of register *Map-3* was set to 0000_2. The remaining bits within the Map registers were set to logic-one. The resulting content of the Map registers is listed in Table 4.1.

Once the control registers (*Mask & Map*) were configured, a data-path was established and registers in the design could be accessed. To access a register, its column address was written to the FastMAP™ interface address bus. The actual address location was obtained by using the column identity as an offset within the XC6200 memory map. Within XC6200ADS operation, the number of a column would be entered and the appropriate address generated automatically.

Register	Content	Row Range	Register	Content	Row Range
Map0	FF_{16}	0 to 7	Map8	FF_{16}	64 to 71
Map1	FF_{16}	8 to 15	Map9	FF_{16}	72 to 79
Map2	FF_{16}	16 to 23	Map10	FF_{16}	80 to 87
Map3	$F0_{16}$	24 to 31	Map11	FF_{16}	88 to 95
Map4	FF_{16}	32 to 39	Map12	FF_{16}	96 to 103
Map5	FF_{16}	40 to 47	Map13	FF_{16}	104 to 111
Map6	FF_{16}	48 to 55	Map14	FF_{16}	112 to 119
Map7	FF_{16}	56 to 63	Map15	FF_{16}	120 to 127

Table 4.1 XC6264 Map Register Contents

Test scripts compiled within the XC6200ADS were used to configure the Map and Mask registers, store and display test results, and automate analysis of results. Such features were useful when evaluating memory structures. Typically however, the operation of a design could only be determined through user interpretation of the results.

The hardware verification method constructed was nevertheless not ideal. However it has been proved practical and reliable in use throughout the project.

4.2 XC6200 Hardware Implementation

Prior to developing complex hardware, the XC6200 FPGAs operating characteristics and component placement strategies were determined. Fundamental processing operators were configured within the XC6200, which allowed the structure, performance and implementation methods used to be assessed. Addition, subtraction, division, multiplication units and memory structures were constructed, as well as simple RTR configurations used to develop dynamic hardware strategies. These investigations are detailed in *Sections-4.2.1* to *4.2.7*.

The FPGA used in these experiments was the XC6264. This was the largest XC6200 device with a specified gate capacity of 64000 to 100000 gates (128 x 128 CLC array) [32]. Previous investigations by the author had concluded this figure was 16384 two variable Boolean expressions combined with 16384 D-Type flip-flops.

To provide performance benchmarks for the VHDL designs, results generated using the XC6264 were compared with those produced for general-purpose XC4013 FPGA. The XC4013 FPGA (XC4000 family [9]) was chosen since its gate capacity of 10000 to 30000 (576 CLBs) was similar to that calculated for the XC6264. The XC4000 family has a CLB logic capacity approximately three times greater than the XC6200 CLC but is not run-time reconfigurable. To provide a comparison with C40 operation, the number of instructions required to compute a similar function to the VHDL code within the DSP was also assessed.

Experiments were conducted using the XC6200DS in the dynamic hardware prototype environment mode of operation. Test stimuli were applied and analysed using XACT6000 software and XC6200ADS test scripts as described in *Section-4.1*.

4.2.1 Addition Unit

Addition is a key operation in many processing tasks. The simplest method of adding two bits together is to use a full-adder, were two inputs (*a, b*) and a carry-in (*cin*) are

summed and a resultant sum (*sum*) and carry-out (*cout*) produced. To add two '*n*' bit words together, '*n*' full-adders connected in a 'daisy-chain' fashion can be used. In such a design, the carry-in signal of the least significant full-adder is tied to logic-zero. The 'daisy-chain' method is slow as propagation delays occur within the carry chain. Methods such as 'look-ahead' and 'fast carry' adders can reduce these delays [67].

a	*b*	*cin*	*cout*	*sum*
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1

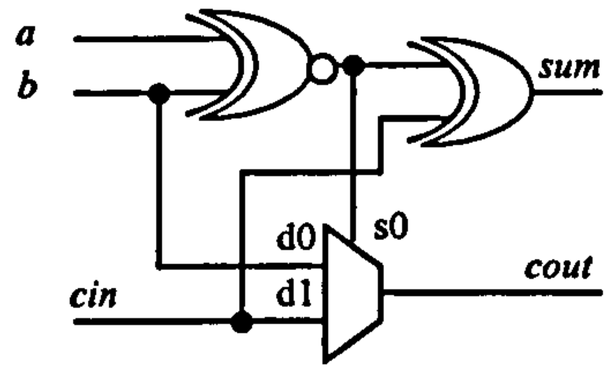


Figure 4.3 XC6200 VHDL Full-adder Design and Truth Table

To achieve high operand throughput carry signal propagation delays must be kept to a minimum. The full-adder architecture analysed was documented within a Xilinx tutorial [82]. The VHDL design of this adder reflected the structure of the CLC array, and therefore could be mapped efficiently within the XC6264. The design and truth table are shown in Figure 4.3. The full-adder is a 'fast carry' type as the carry output (*cout*) has the same propagation delay as the sum output (*sum*).

Using the combinational logic full-adder design (*Figure 4.3*), unsigned adders of width 8, 16, and 32-bits were configured within the XC6200 and the results generated compared against XC4013 FPGA implementation. The results obtained are listed in Table 4.2. Since a register-less full-adder implementation was used, the throughput of

the design (operating frequency) was determined through cascaded signal propagation delays.

	XC6264		XC4013	
Adder Width	*No. Of CLCs*	*Max. Freq. (MHz)*	*No. Of CLBs*	*Max. Freq. (MHz)*
8-bit	24	26.22	11	36.22
16-bit	48	15.847	23	20.68
32-bit	96	8.89	47	11.23

Table 4.2 XC6264/XC4013 VHDL Addition Unit

The addition-unit structure was designed for optimum mapping upon XC6264 architecture. To provide an additional performance benchmark, dedicated XC4013 based addition units were generated using Xilinx Foundation CORE module generator software. The results shown in Table 4.3 were obtained using an unsigned registered adder design. Unlike the register-less full adder design (*Figure 4.3*), operand throughput was calculated by multiplying the operating frequency by the width (bit(s)) of the unit.

Adder Width	No. Of CLBs	Max. Freq. (MHz)
8-bit	6	89.67
16-bit	10	67.72
32-bit	18	48.28

Table 4.3 XC4013 CORE Generator Addition Unit

The efficiency in which a common VHDL design entity could be mapped within different FPGA architecture varied. This factor related to software design tool issues, which were proven by constructing addition-units optimised for both the XC6200 and XC4000 FPGA architectures. The results generated concluded that within XC6200 VHDL designs, the structure and content of the design should reflect actual CLC placement and routing of the design. Compared to traditional VHDL techniques this implied XC6200 VHDL code appeared inefficient.

These results reinforced previous conclusions that programmable logic suffered from low operating frequencies when compared to hardwired logic and microprocessor

operation. In comparison the C40 DSP could implement one 32-bit addition using instruction ADDC (*Add integer with carry*) in one cycle (50nsec @40MHz) [65].

The addition-unit constructed was a 'bit-slice' type and it was predicted that as the bus width increased the maximum operating frequency would decrease in a linear fashion. Through analysis of the XC6200 results, it was evident that this assumption was incorrect. It was determined that the operating frequency was much more dependant upon the level of XC6200 routing hierarchy used than previously anticipated.

4.2.2 Subtraction Unit

Binary subtraction was similar to addition, and could be accomplished through modification of the adder unit described in *Section-4.2.1*, as illustrated in Figure 4.4. This design functioned by converting one operand into its twos-complement form and then adding it to the other, hence performing the subtraction operation. This operation took one clock cycle. Instead of *cin* and *cout*, subtraction units have borrow-in (*bin*) and borrow-out (*bout*) signals. For subtraction the least significant bit *bin* signal must be connected to logic-one.

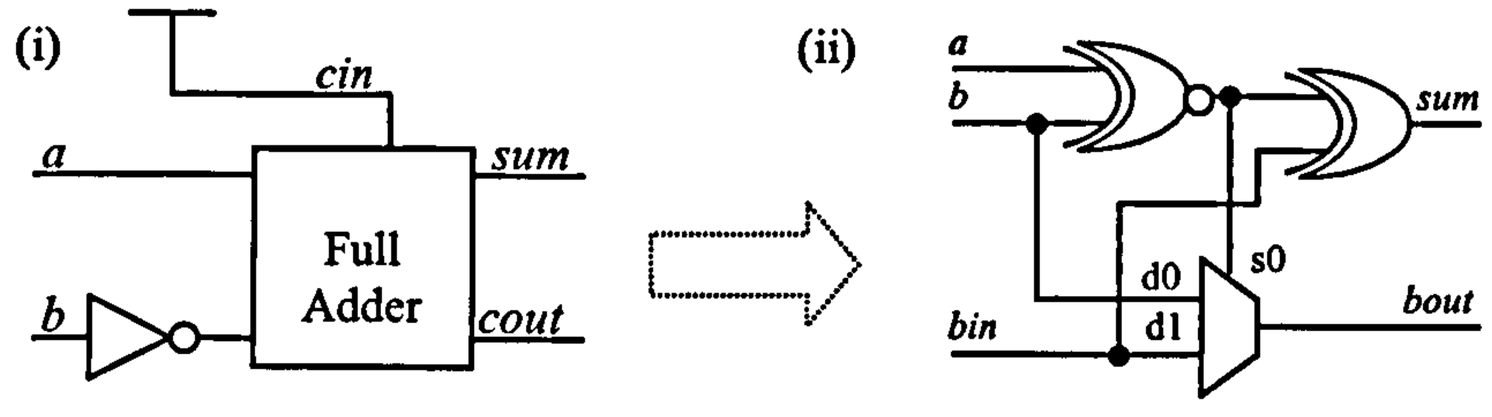


Figure 4.4 Modified Full-Adder *(i)* and Subtraction Unit *(ii)*

The subtraction-unit (unsigned) was configured for different bus widths, with the results obtained detailed in Table 4.4 and XC4013 optimised version in Table 4.5. Operand throughputs of results in Table 4.4 were equivalent to the operating frequency (calculated using 1/max signal propagation delay) since the subtraction unit was a register-less design. In comparison CORE generated designs (*Table 4.5*) were register

based, therefore operand throughput was equivalent to the clock frequency multiplied by the subtraction units width.

	XC6264		XC4013	
Sub. Width	*No. Of CLCs*	*Max. Freq. (MHz)*	*No. Of CLBs*	*Max. Freq. (MHz)*
8-bit	24	26.43	11	36.27
16-bit	48	15.92	23	20.68
32-bit	96	8.92	47	11.23

Table 4.4 XC6264/XC4013 VHDL Subtraction Unit

Sub. Width	No. Of CLBs	Max. Freq. (MHz)
8-bit	6	89.67
16-bit	10	62.72
32-bit	18	48.27

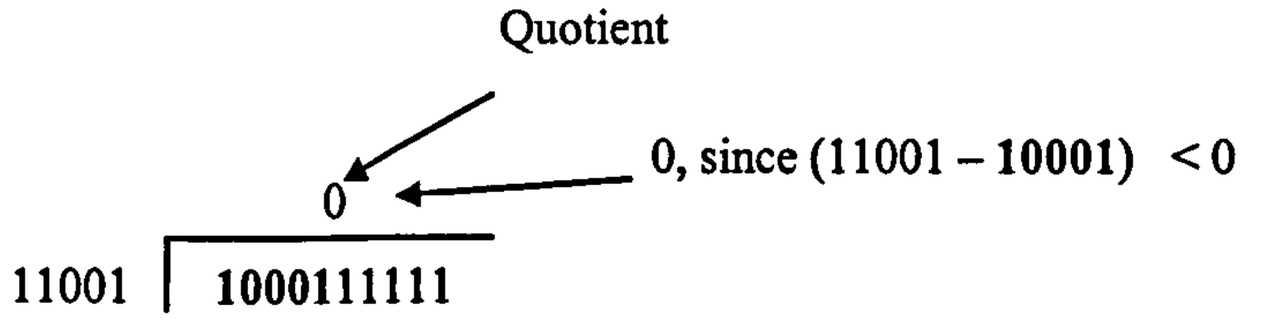
Table 4.5 XC4013 CORE Generator Subtraction Unit

Analysis of the results reinforced conclusions previously drawn in *Section-4.2.1*. Whilst developing the subtraction unit, it was discovered that when defining a logic-one signal within hardware, VHDL entity VCC should not be used. This was to minimise XC6200 routing as VCC required chip-wide routing resources. Instead, a logic-one would be generated using an OR2B1 gate with both inputs connected to a common local routed signal. In comparison to the C40, the FPGA configured subtraction units were inefficient. The C40 could execute integer subtraction instruction (SUBI) in one cycle (50nsec @40MHz) [65].

4.2.3 Division Unit

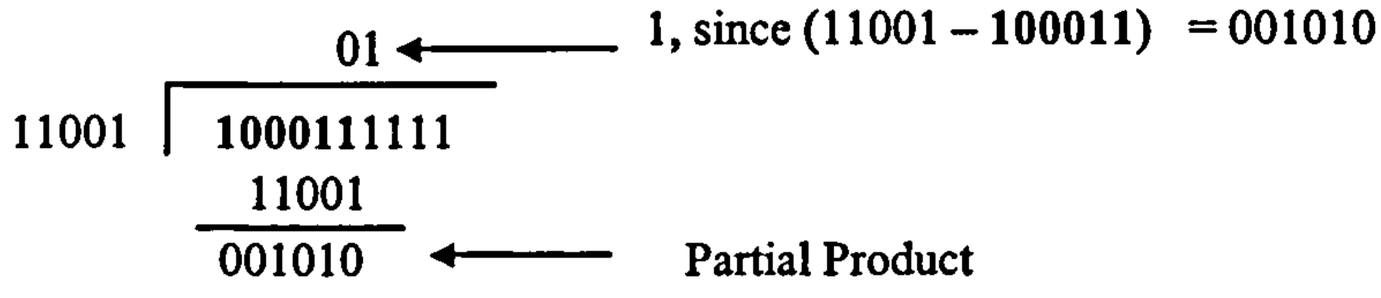
Binary division can be accomplished using a 'shift and subtract' approach similar to long division of decimal numbers. The procedure shown in Equations 4.1 to 4.4 divides a dividend of 1000111111_2 by a divisor of 11001_2. The basis of this procedure was to count the number of times the divisor could be subtracted from the dividend until a result of zero or negative value was obtained.

The divisor of '*n*' bits wide was first subtracted from the '*n*' most significant bits of the dividend. Since the result was negative a zero was entered into the quotient.



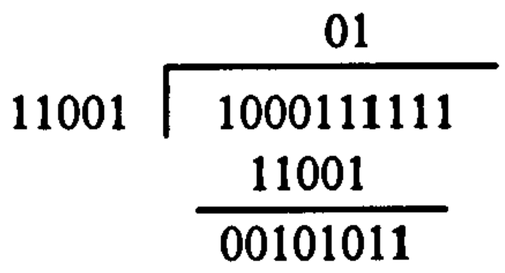
Equation 4.1

Because the result of the first subtraction was negative, the divisor was then subtracted from the next six bits of the dividend. This produced a positive result (partial product) of 001010_2 thus a one was entered into the quotient (*Equation 4.2*).



Equation 4.2

The next bit of the dividend was then included and added to the partial product as depicted by Equation 4.3.



Equation 4.3

The divisor was then subtracted from the partial product and produced a negative result. Therefore a zero was entered into the quotient. This process was repeated the number of times there were bits in the divisor, with the result stored in the quotient and the remainder located in the partial product. This is shown in Equation 4.4.

```
           010111   <------ Result
      _____________
11001 | 1000111111
         11001
        --------
        00101011
          11001
         -------
          100101
           11001
          -------
            11001
            11001
           -------
                0
                ↗
Remainder (partial product)
```

Equation 4.4

A XC6200 division unit was constructed with each stage of the process (*Equations 4.1-4.4*) relating to a shift-register, subtraction-unit and control logic within the design. This type of divider was known as a 'restoring divider' [69] since the original value of the partial product was restored when a subtraction operation generated a negative result. The results obtained are listed in Table 4.6, where the dividend is twice the divisor (*div*) width.

	XC6264		**XC4013**	
Div. Width	*No. Of CLCs*	*Max. Freq. (MHz)*	*No. Of CLBs*	*Max. Freq. (MHz)*
8-bit	167	3.87	53	6.13
16-bit	1104	1.99	240	1.19
32-bit	2081	1.24	N/A	N/A

Table 4.6 XC6200/XC4013 VHDL Division Unit

The Xilinx CORE module generator supported binary division up to divisor widths of 24-bits. However, CORE and XC6264 VHDL designs with respective divisors greater than 8-bit and 16-bits wide would not fit on to XC4013 FPGA architecture. Using the CORE generator an 8-bit divider was fabricated, requiring 129 CLBs and having a maximum frequency of 87.40 MHz.

For all divider implementations listed, the operand throughput was determined by multiplying the clock frequency by the number of cycles required to generate the result (width of dividend).

The C40 computed a division using repeated (via RPTS instruction) subtractions (SUBC instruction) and shifts (LSH instruction). The instruction SUBC was executed within a loop until a flag was set. SUBC and the shift operation instruction LSH took one cycle (50nsec @40MHz) to execute. The loop instruction RPTS took four cycles, however the number of times RPTS was executed was dependant upon the division calculation. In comparison to the XC6200 divider, the C40s operation still had greater throughput since the operating clock frequency of the DSP was 40MHz, independent of bus width.

The constructional techniques gained from developing this design proved beneficial. Previous designs had been regular structured bit-slice designs without any major supervisory state machines. Limitations within the XC6200s architecture, development tools, and the testability and verification of hardware configured within the XC6200 were exposed.

VHDL component FDCP (*Flip-flop Device* with *Clear Preset*) could not be mapped within a CLC. This component was present within the VHDL library supplied with XACT6000 software, yet XACT6000 would reject the component when compiled. This problem was overcome by creating a new entity formed using valid existing VHDL components.

When XACT6000 software compiled designs, it was discovered that signals would become inverted to simplify signal routing constraints. To prevent signal inversion from occurring a buffer component was inserted into the signal path.

For the design to be routed successfully, all system components were positioned manually within the CLC array using XC6200 specific VHDL attributes. The placement

methodology used created a regular footprint and minimised routing resources reliance. XACT6000s component placement strategy was to use the smallest area of CLC array possible. Often the component positions chosen could not be fully routed, therefore it was determined that manual placement of components using VHDL attributes would be used within designs.

4.2.4 Multiplication Unit

Multiplication can be achieved using a 'shift and add' approach, in which the multiplicand is added to the product the number of times the value of the multiplier. Supplied with the FATHOTs Development kit [67] were examples of multiplier designs constructed in VHDL. The aim of this experiment however was to determine the implementation and operation characteristics of such architecture, therefore a new 'shift and add' multiplier was designed and constructed.

Figure 4.5, illustrates the multiplier structure developed which was known as a *ripple carry array multiplier* [69], since the operation was performed by off-setting the product term in each stage by one bit with respect to the previous stage. Each row within the design was created using cascaded full-adders.

Within Figure 4.5 inputs (a3 to a0) and (b3 to b0) correspond to the multiplier and multiplicand respectively, and (p7 to p0) the partial product. For a registered full-adder based implementation, the multipliers output was generated in *n* clock cycles. If non-register based full-adders were used (shown in *Figure 4.3*), the output delay would be equal to the maximum signal propagation delay within the design.

The design was verified using a VHDL simulator, but XC6200 placement proved difficult. It was possible to implement an 8-bit multiplier but increasing the bus width incurred signal routing conflicts. This situation reflected implementation issues encountered previously. Instead, a multiplier example supplied with the FATHOTs kit was used to generate performance benchmarks (*Table 4.7).* This design was similar, but the distribution and placement of system components was more suited to generating a

regular placement footprint for array rows. This simplified signal routing constraints. The characteristics of the equivalent CORE generator multipliers are shown in Table 4.8. The XC4013 FPGA did not have sufficient CLB capacity to support implement 32-bit designs.

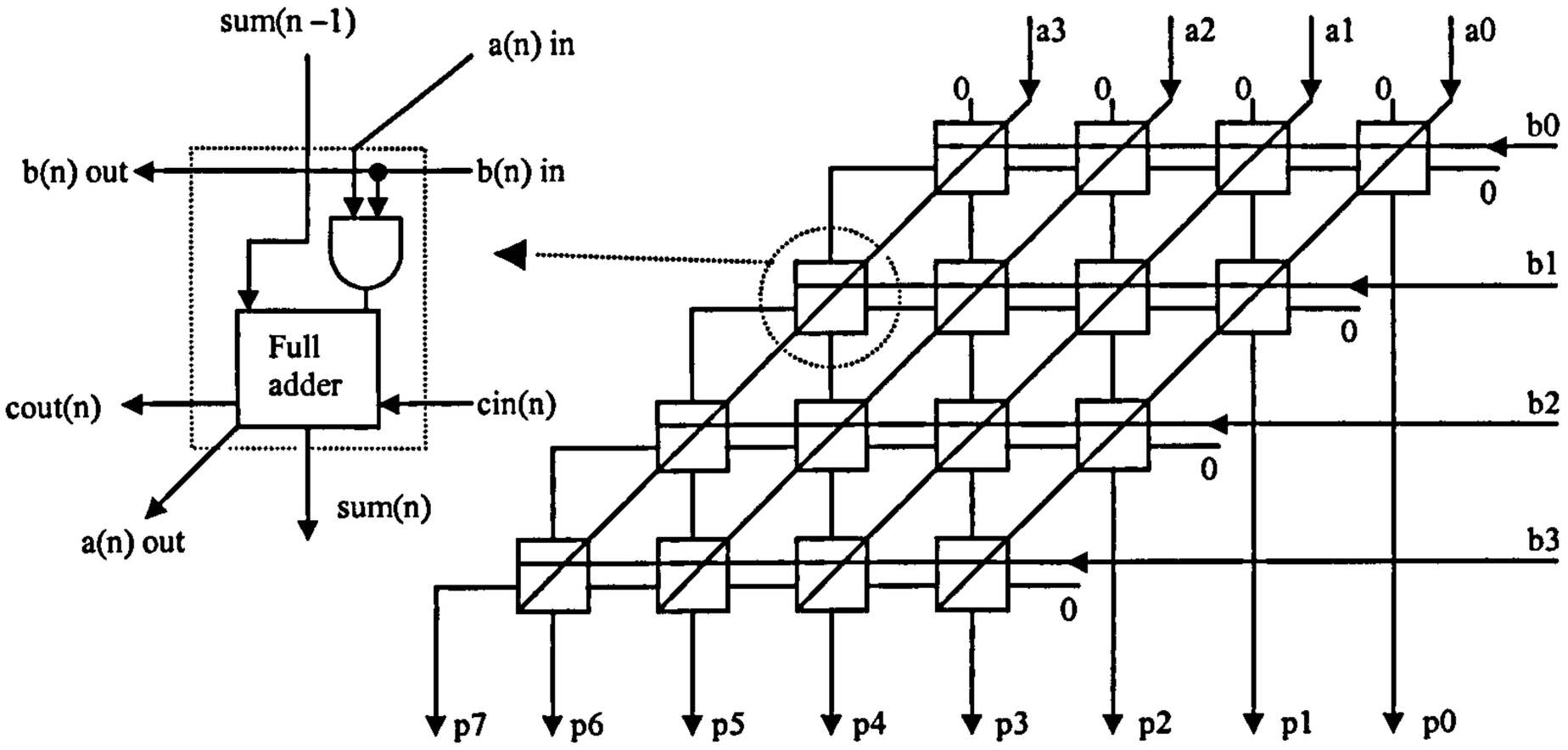


Figure 4.5 Ripple Carry Array Multiplier

Ripple carry array multipliers can be used to calculate twos-complement products. An alternative method is Booths algorithm [69] (*Table 4.9*), which differs in structure and method of operation. The multiplier, multiplicand, and partial product in Table 4.9 are known as (A), (B), and (P) respectively. This investigation was conducted to determine which type of multiplier architecture was most suited for XC6200 implementation.

	XC6264		XC4013	
Mult. Width	*No. Of CLCs*	*Max. Freq. (MHz)*	*No. Of CLBs*	*Max. Freq. (MHz)*
8-bit	167	10.08	78	7.66
16-bit	1104	7.71	344	3.33
32-bit	2081	2.44	N/A	N/A

Table 4.7 XC6264/XC4013 VHDL Ripple Carry Array Multiplier

Multiplier Width	No. Of CLBs	Max. Freq. (MHz)
8-bit	52	70.02
16-bit	208	32.68
32-bit	N/A	N/A

Table 4.8 XC4013 CORE Generator Multiplier

A	[A-1]	Meaning	Action Taken
0	0	Middle of string of 0's	P = P + 0
0	1	End of string of 1's	P = P + B
1	0	Start of string of 1's	P = P - B
1	1	Middle of string of 1's	P = P + 0

The initial value of [A –1] = 0

Table 4.9 Booths Multiplication Algorithm

Booths algorithm utilised the property that a string of logic-ones in the multiplier operand corresponded to several additions, which could be replaced by one subtraction or one addition. Using Booths algorithm, pairs of bits in the multiplier operand were examined against the properties listed within Table 4.9 and the respective operation performed. This was a cyclic operation with the position of the multiplier bit used [A] being right shifted by one bit each successive operation. The partial product became the final product upon the last iteration of the loop.

To implement a '*n*' by '*n*' multiplication operation the processing hardware required consisted of an *n*-bit register to store the multiplicand, and a $(2n + 1)$ bit register to hold the partial product. This register required a right shift function with sign extension. To perform the addition and subtraction operations an *n*-bit *Arithmetic Logic Unit* (ALU) was required. Governing overall operation was a control unit used to determine the ALUs function.

The operational characteristics obtained are detailed in Table 4.10. The operand throughput was calculated by multiplying this value by the number of cycles (*n*)

required to calculate the product. The largest multiplier width evaluated would not fit within the XC4013 FPGA, whilst no corresponding CORE generator module existed.

	XC6264		XC4013	
Mult. Width	*No. Of CLCs*	*Max. Freq. (MHz)*	*No. Of CLBs*	*Max. Freq. (MHz)*
8-bit	167	14.02	38	28.29
16-bit	1104	10.04	74	16.01
32-bit	2081	6.32	N/A	N/A

Table 4.10 XC6264/XC4013 VHDL Booths Algorithm Multiplier

Complementing previous design tasks, the positioning and routing of system components within the XC6200 CLC array proved time consuming and often resulted inefficient in use of available CLC logic resources.

Comparing the two multiplication methods, the Booths algorithm implementation was constructed using fewer CLCs than the ripple array, which was common for all bus widths. The operating clock frequency of the Booths multiplier was greater than the ripple array, however the operand throughput of the design was less through its cyclic operation. In comparison the C40 could perform similar integer multiplications using the MPYI instruction in one cycle (50nsec @40MHz) [65].

Through analysing the structures of the two multipliers it was concluded Booths algorithm was more suited for a microprocessor than FPGA based implementation. This was because of the irregular hardware design footprint and cyclic operation. This could be performed more efficiently within a loop-orientated architecture.

4.2.5 Multiply Accumulate Unit

Multiply Accumulate (MAC) is a common operation within processing applications. Individual C40 MACs took two clock cycles to compute using instructions MPYI and ADDI. Instructions MPYI3 and ADDI3 could be processed concurrently within the C40, however within an individual MAC operation this does not occur.

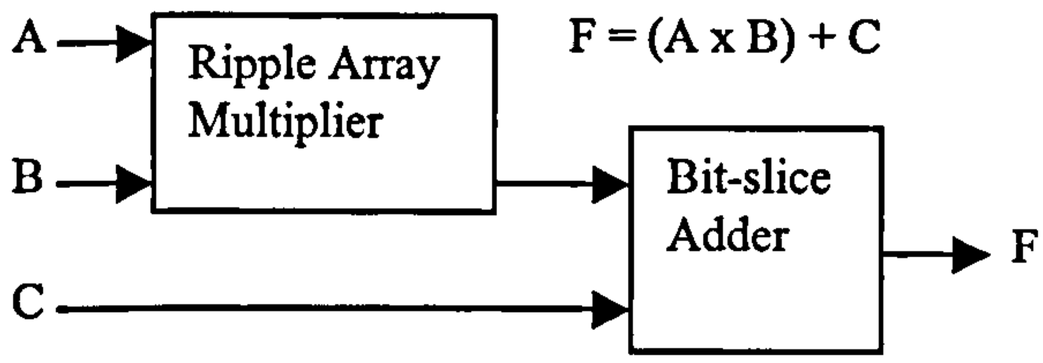


Figure 4.6 XC6264/XC4013 VHDL MAC Architecture

The structure of the XC6200 MAC shown in Figure 4.6 was fabricated using register-less ripple array multiplier and bit-slice adder designs described previously. The results obtained for XC6264 and XC4013 implementations of different MAC widths are shown in Table 4.11. No MAC functions were available within the version of CORE generator software used.

Once input operands were written, the MACs output was generated within one operating cycle. The duration of this cycle was equal to the maximum signal propagation delay path through the multiplier and adder. In comparison individual C40 MACs took two operating cycles. However, multiple C40 MAC operations could overlap within the C40s instruction pipeline and execute concurrently in one cycle.

	XC6264		XC4013	
MAC Width	*No. Of CLCs*	*Max. Freq. (MHz)*	*No. Of CLBs*	*Max. Freq. (MHz)*
8-bit	323	8.91	95	6.63
16-bit	1155	5.28	381	3.34
32-bit	3986	4.15	N/A	N/A

Table 4.11 XC6264/XC4013 VHDL Multiply and Accumulate

4.2.6 RAM Memory Structures

Within processing architectures local memory was often required to store operands temporally. This resource could either be discrete register or memory structure based. The simplest memory structure was an asynchronous RAM [71]. This structure (excluding the address decoder) is shown in Figure 4.7. A word was written to memory by writing data to the input (*Data_in*) and then toggling the appropriate write select

signal (*W[n]*). This signal was connected to the clock inputs of all registers forming the word, hence latched the word into the registers. To read a word from memory the respective read signal was set (*R[n]*), which forced the row multiplexor to select the register outputs. The word then appeared upon output (*Data_out*). FPGA RAM implementation characteristics generated using this design of different word size and depth are shown in Table 4.12.

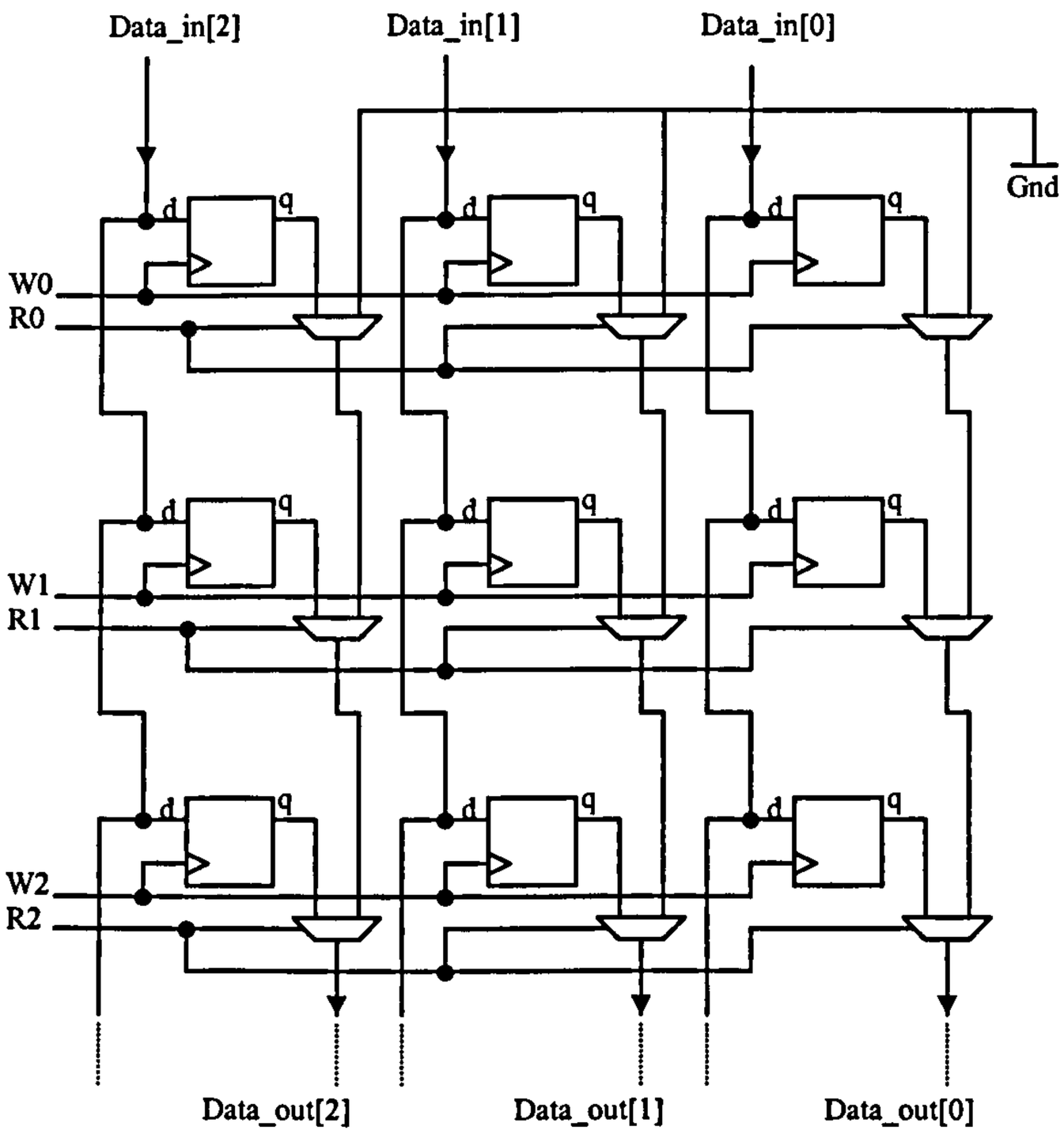


Figure 4.7 XC6264/XC4013 VHDL Asynchronous RAM

	Word Width (bits)					
	8		16		32	
RAM Depth	*No. Of CLCs*	*Access Time*	*No. Of CLCs*	*Access Time*	*No. Of CLCs*	*Access Time*
8-words	128	41.81 nsec	256	41.81 nsec	512	41.81 nsec
16-words	256	77.32 nsec	512	77.32 nsec	1024	77.32 nsec
32-words	512	148.33 nsec	1024	148.33 nsec	2048	148.33 nsec

Table 4.12 XC6264 VHDL Asynchronous RAM

Results generated indicated the memory access time increased as the number of words in the RAM (depth) increased. Access times for RAMs of same word depth but different word size were consistent. This statistic was caused through the multiplexers providing word selection introducing cascaded gate delays. The propagation delays encountered in each memory column (RAM depth) were far greater than those encountered in each row (RAM word size). In similar RAM designs, use of tri-state gates would eradicate this problem. Within the XC6200 tri-state gates could only be configured within IOBs. Routing problems were also encountered since the clock signal of each register had to be routed using standard local CLC routing rather than dedicated clock routes. This increased propagation delays, hence reduced access times.

A second RAM architecture known as a synchronous RAM was constructed. Within this design all registers were clocked by a global clock signal. With each clock pulse the output of each register (q) would be routed back into its input (d) via a multiplexer. The content of the register therefore appeared constant. Register contents were updated by the multiplexer switching between the row outputs and RAM input data-bus. Memory access times obtained were identical to those listed in Table 4.12 since the number of CLCs used to implement each memory type were identical and routing structure similar.

There are further memory structures such as dual port ram and FIFOs, however through implementing only asynchronous and synchronous RAM it was apparent that the XC6200 was suited only for the configuration of small and simple memory structures. ROM memory could also be generated using RPFD registers with the content written via the FastMAP™ interface.

Xilinx Foundation CORE generator software was used to develop synchronous RAM modules. From the results obtained it was apparent that coarse grain FPGA architectures were most suited for implementing memory structures. Results obtained through these experiments are shown in Table 4.13, and compare CORE generate RAM modules to that of XC6264 VHDL designs (*Figure 4.7*) implemented within XC4013 architecture.

These memory structures had a fixed depth of 32 words, since the CORE generator supported a minimum RAM depth of 16 words.

The results highlighted the inefficient method by which RAM was constructed upon FPGA architecture using discrete gates. In comparison the CORE generated RAM used fewer CLBs and had faster access times. This was because the RAM structure was fabricated directly within XC4013 FPGA CLB LUTs. XC4000 FPGA CLBs have two four variable LUTs. Each can be configured as a 4-bit x 4-word RAM. This feature cannot be replicated upon XC6200 FPGA architecture.

	XC6200 Design		**CORE Generator**	
Ram Size	*No. Of CLCs*	*Access Time*	*No. Of CLBs*	*Access Time*
8-bits x 32-words	192	80.14 nsec	8	11.60 nsec
16-bits x 32-words	384	76.19 nsec	16	12.71 nsec

Table 4.13 XC4013 CORE Synchronous RAM

4.2.7 Run-Time Reconfiguration

XC6200 FPGAs could be reconfigured through the Fast MAP™ interface during run-time. Configuration data required was generated through analysing sequential XC6200 configurations and performed within the XC6200ADS (*Section-3.4.4*). The volume of configuration data generated, hence the configuration delay was proportional to the difference between successive configurations. Therefore to minimise RTR delay sequential configurations should have similar structures as possible.

Compared to the FATHOTs mechanism [67] the XC6200DS configuration methods allowed more efficient use of CLC resources, hence reduced the volume of configuration data required. Within the FATHOTs method individual CLCs designated for reconfiguration (denoted using a VHDL attribute) could not be used to provide additional signal routing. The configuration data generated also contained all possible

CLC configurations. From this data only one configuration would be active at any one time.

In comparison the XC6200ADS mechanism determined only the difference between the present and next configuration of a CLC. If the CLCs configuration required updating, four bytes of configuration data were generated (three XC6200 SRAM addresses, and one data byte).

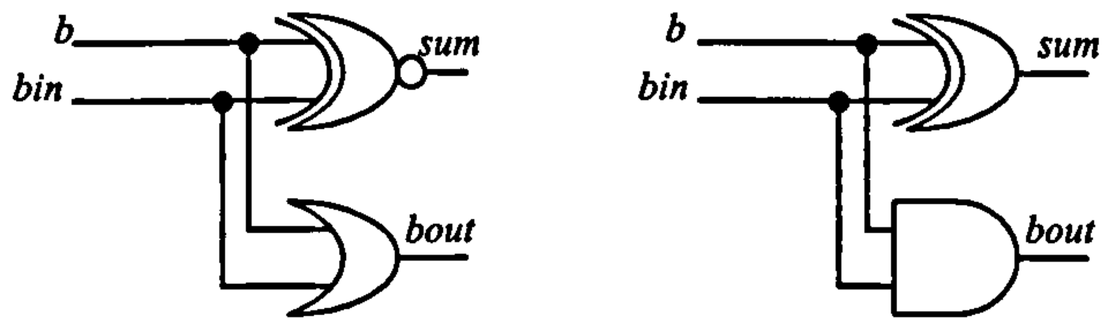
To achieve efficient RTR, the minimal volume of configuration data was required. When designing RTR hardware the position and structure of components within each active configuration was assessed and constructed to minimise differences between them. This approach resulted in inefficient design layouts but acceptable, since the goal of the project was to develop RTR hardware.

Neglecting the configuration delays, the conversion from static to dynamic hardware configuration typically resulted in higher operating frequencies. This was because the temporal partitions of the design contained fewer logic gates and signal routes, therefore reduced signal propagation delays.

Figure 4.8 illustrates the conversion of a static subtraction-unit into a RTR version through temporal partitioning. The original design was temporal partitioned using input (*a*) as the temporal divisor input. The RTR implementation consists of two configurations, with the difference between them relating to two CLC configurations. When input (*a*) was at logic-zero, configuration (i) was active, otherwise configuration (ii) was used.

To demonstrate this technique, an unsigned 8-bit wide subtraction unit was developed. This unit had to subtract either 11110000_2 or 11110001_2 (input *a*) from input (*b*). Within this design input (*a*) was hardwired within the design using the subtraction bit configurations illustrated in Figure 4.8 (i) and (ii)).

The resultant design consisted of two configurations that were swapped as required. Within the first configuration, input (a) was set to 11110000_2, whilst the second configurations value was set to 11110001_2.



Configuration (i)
Input $a = 0$

Configuration (ii)
Input $a = 1$

Figure 4.8 XC6200 Temporal Partitioned Subtraction Unit

To perform RTR configuration the XC6200ADS analysed both designs and generated configuration data consisting of XC6200 FastMAP™ interface address and data pairs. RTR configuration data generated is shown in Table 4.14 and 4.15; Omitted are configuration mechanism and XC6200ADS specific control values. Using this data RTR can commence either under control of the host computer (15. 2μsec) or using the self-configuration controller (1.54msec).

The configuration delays were measured using an external custom designed 24-bit counter, having a timing resolution of 25nsec (@ 40MHz). This was attached to the XC6200DS using flying leads. The counter functioned by using XC6200 FPGA generated signals to enable/disable the clock signal of the counter. These signals were activated/deactivated during the dynamic configuration processes. The delay incurred was then read from the counter in binary (3 bytes), and converted to decimal. This value was then multiplied by the timers clock frequency period (25nsec @40Mhz).

When measuring RTR timings generated through XC6200ADS, typically three measurements were recorded using the counter, then the mean value calculated. This processes was required due to hardware/software interrupt operation upon the host PC. These interrupts could not be masked due to the nature of host PC operating software. XC6200ADS XC6200 address/data-pair update delays excluding interrupt operation where calculated to take 2.4μsec (5 ISA bus write cycles @8.33Mhz). In comparison

self-configuration control timings measured were constant and similar to calculated values, since host PC operation was not required during dynamic configuration (measured at 1.88μsec per CLC).

FastMAP™ Address	FastMAP™ Data
20007_{16}	00_{16}
$2000B_{16}$	00_{16}
$0908F_{16}$	07_{16}
$00A88_{16}$	EE_{16}

Table 4.14 XC6264 Configuration Data, RTR design (i) to (ii)

FastMAP™ Address	FastMAP™ Data
20007_{16}	00_{16}
$2000B_{16}$	00_{16}
$0908F_{16}$	23_{16}
$00A88_{16}$	BE_{16}

Table 4.15 XC6264 Configuration Data, RTR design (ii) to (i)

Independent upon which XC6200DS RTR mechanism was used four XC6264 address locations were updated to switch between active configurations. Although one address-data pair determined the function of each CLC, four addresses were required. This was because the logic placement and mapping strategy used by XACT6000 software utilised signal inversion inherent within the XC6200s routing structure. This was to aid logic placement and could not be disabled within XACT6000.

Compared to fixed 8-bit subtraction (*Section-4.2.2*), the dynamic implementation reduced the number of gates required from 24 to 16 and simplified the designs routing structure. This resulted in a higher operating frequency of 31.2MHz when compared to the static design of 26.43MHz (*Table 4.4*). However this increase in performance excluded the reconfiguration time.

4.3 Summary

The work presented in *Chapter-4* determined the design implementation and hardware verification procedures for use with the XC6200DS. This work was vital before commencing development on more complex static and RTR processor and router structures described in the proceeding chapters.

The outcome of this chapter has been the development of such techniques as well as providing performance benchmarks to evaluate the system operation. Work conducted determined three problem areas.

VHDL can be used to construct XC6200 hardware. Although the format of VHDL code must comply with IEEE-1164 standard, for efficient placement XC6200 VHDL designs must reflect the structure of the resultant XC6200 design. XC6200 VHDL code must also be written at gate-level and not using 'process' or conditional operators, with exception of the 'for' statement. This restriction implies XC6200 VHDL designs appear structurally inefficient and time consuming to construct.

When constructing XC6200 designs using VHDL, additional components (primarily buffer components) must be inserted to provide signal path guides between different hierarchical levels within designs. This feature also limited signal inversion occurring during component placement. To minimise the occurrence of such problems, components were manually positioned within the XC6200 FPGAs CLC array using XC62000 specific VHDL attributes.

To evaluate and debug hardware configured within the XC6200 external I/O signals and internal registers were configured within designs. This method of hardware verification was functional and did not allow accurate analysis of signal timings. To minimise the occurrence of timing hazards, hazard reduction techniques must be applied throughout the design process.

The conclusions determined through using the XC6200 FPGA family and XACT6000 software design tools reflected their status as non-commercial products. Through commercially developing these products the performance of both components would have improved dramatically. The XC6200 gate capacity and clock frequencies obtainable were poor in comparison to similar FPGAs. The structure of the FastMAP™ interface as well could be improved. Although the FastMAP™ interface has a 32-bit data-bus, RTR could only be accomplished using 8-bit data. This interface however is unique among FPGA architectures as it allows dynamic configuration to occur. This was the primary reason why the XC6200 FPGA family was chosen for use in the project.

Chapter 5

The Dynamic BinDCT Algorithm

Introduction

This chapter describes investigations conducted to determine if the throughput and performance of the BinDCT algorithm could be improved through dynamic hardware implementation. This algorithm is a new integer friendly multiplier-less approximation of the DCT.

An overview of DCT operation is provided in *Section-5.1*, with the derivation of the BinDCT algorithm from the DCT described in *Section-5.2*. *Section-5.3* discusses experiments conducted to determine the benefits gained from an RTR implementation, whilst *Section-5.4* provides a summary of the conclusions derived from this work.

5.1. The Discrete Cosine Transform

The *Discrete Cosine Transform* (DCT) has been used extensively in signal and image-processing compression techniques. The DCT is related to the Fourier Transform but differs since only real numbers are generated in the computation. Effectively the DCT can be considered as the cosine operation within the Fourier Transform.

The DCT functions by converting a time-domain input sequence in to their respective frequency components. This operation is known as a forward DCT with the frequency composition dependant upon the length of input sequence and dynamic range of values. The output generated by a forward DCT consists of a DC component that is the average of the input sequence and AC coefficients dictating frequency content.

If the dynamic range of an input sequence is limited the frequency content of the DCT (AC coefficients) is small. Using compression methods such as run-length coding [72], AC coefficients at zero or near to zero can be quantisised compressed and represented using fewer digits.

To reconstruct compressed data a reverse DCT is applied. The difference between the original input sequence and output of the reverse DCT is minimal, as the smaller AC coefficients quantisised do not contribute major frequency components.

5.1.1 Transform Computation Methods

The forward DCT algorithm is defined in Equation 5.1, with the reverse DCT defined in Equation 5.2

$$Xc[k] = \alpha[\kappa] \sum_{n=0}^{N-1} x[n] \cos\left(\pi(2n+1)k / 2N\right)$$

For k = 0,1,.N-1

Equation 5.1 Forward DCT Algorithm

$$x[n] = \sum_{k=0}^{N-1} \alpha[k] Xc[k] \cos\left(\pi(2n+1)k / 2N\right)$$

For n = 0,1,.N-1

Equation 5.2 Reverse DCT Algorithm

Where:

$\alpha[k] = \sqrt{1/N}$ for k = 0

$\alpha[k] = \sqrt{2/N}$ for k = 1,2,..N-1

N = Length of input data set

The forward transform of an input data set of length 'N' and its respective reverse operation can be calculated directly using Equations 5.1 and 5.2. For an input sequence of eight samples calculated without simplification, the forward transform would take 320 multiplications, 128 additions, and 64 discrete division operations to compute. Similar, the reverse transform would require 384 multiplications, 64 additions, and 64 division operations. These values were calculated using XC6200ADS software functions implementing one-dimensional DCT transforms.

The DCT operates using a cosine function over a time period of 0 to 2π as a weight to determine the amplitude of frequency components (AC coefficients) for each operand within the input sequence. This operation is conducted upon the entire input data set as

the respective amplitudes of frequencies obtained are dependant upon the weighted (over period of 2π) average of the DC coefficient.

Figure 5.1 illustrates the cosine function over a period of 0 to 2π. Coefficient weights in the range of plus one to minus one are generated therefore can be of the same magnitude but opposing signs. Using this property the number of calculation required to compute a DCT can be reduced.

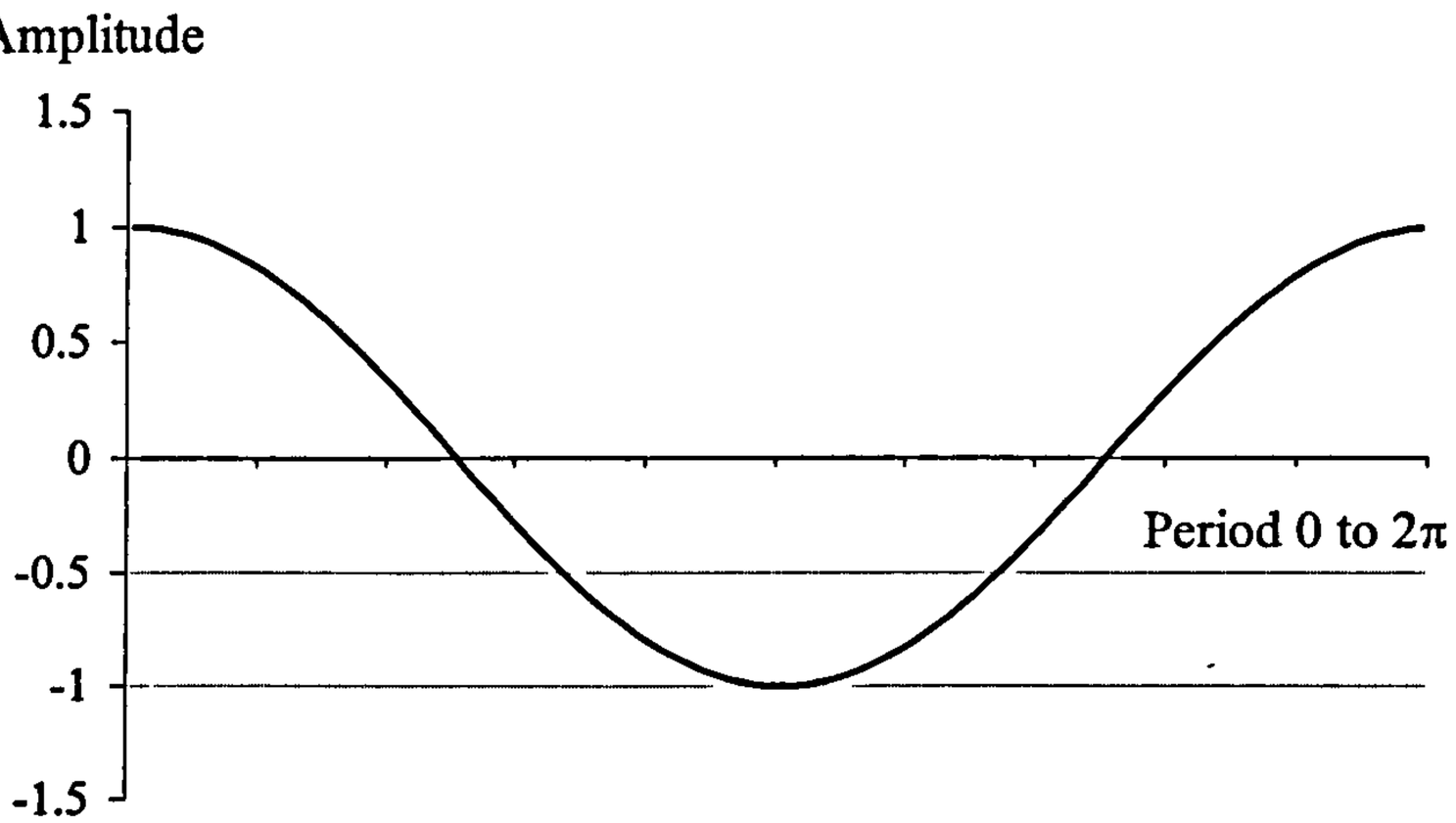


Figure 5.1 Cosine Function Through Time Period 0 to 2π

Using Equations 5.1 and 5.2, for an operation of length 'N' to reduce computation overheads, function $\cos(\pi(2n+1)k / 2N)$ can be calculated prior to system operation. For a DCT of input sequence length (N = 8), these values are shown in Table 5.1.

Coefficients in Table 5.1 show that many identical but opposing weights (symmetrical) are required to generate the DCT output. By formulating Table 5.1 as a matrix function the symmetry of composite DCT operations can be exploited and number of computations required reduced. Using this approach, Chen [73], developed a *Fast Discrete Cosine Algorithm* ($FDCT_i$) implementation that reduced computation overheads by almost one sixth compared to Equation 5.1 and 5.2.

	n *(0..N-1)*							
k *(0..N-1)*	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	0.7071	0.7071	0.7071	0.7071	0.7071	0.7071	0.7071	0.7071
1	0.9808	0.8315	0.5556	0.1951	-0.1951	-0.556	-0.8315	-0.9808
2	0.9239	0.3827	-0.3827	-0.9239	-0.9239	-0.3827	0.3827	0.9239
3	0.8315	-0.1951	0.9808	-0.5556	0.5556	0.9808	0.1951	-0.8315
4	0.7071	-0.7071	-0.7071	0.7071	0.7071	-0.7071	-0.7071	0.7071
5	0.5556	-0.9808	0.1951	0.8315	-0.8315	-0.1951	0.9808	-0.5556
6	0.3827	-0.9239	0.9239	-0.3827	-0.3827	0.9239	-0.9239	0.3827
7	0.1951	-0.5556	0.8315	-0.9808	0.9808	-0.8315	0.5556	-0.1951

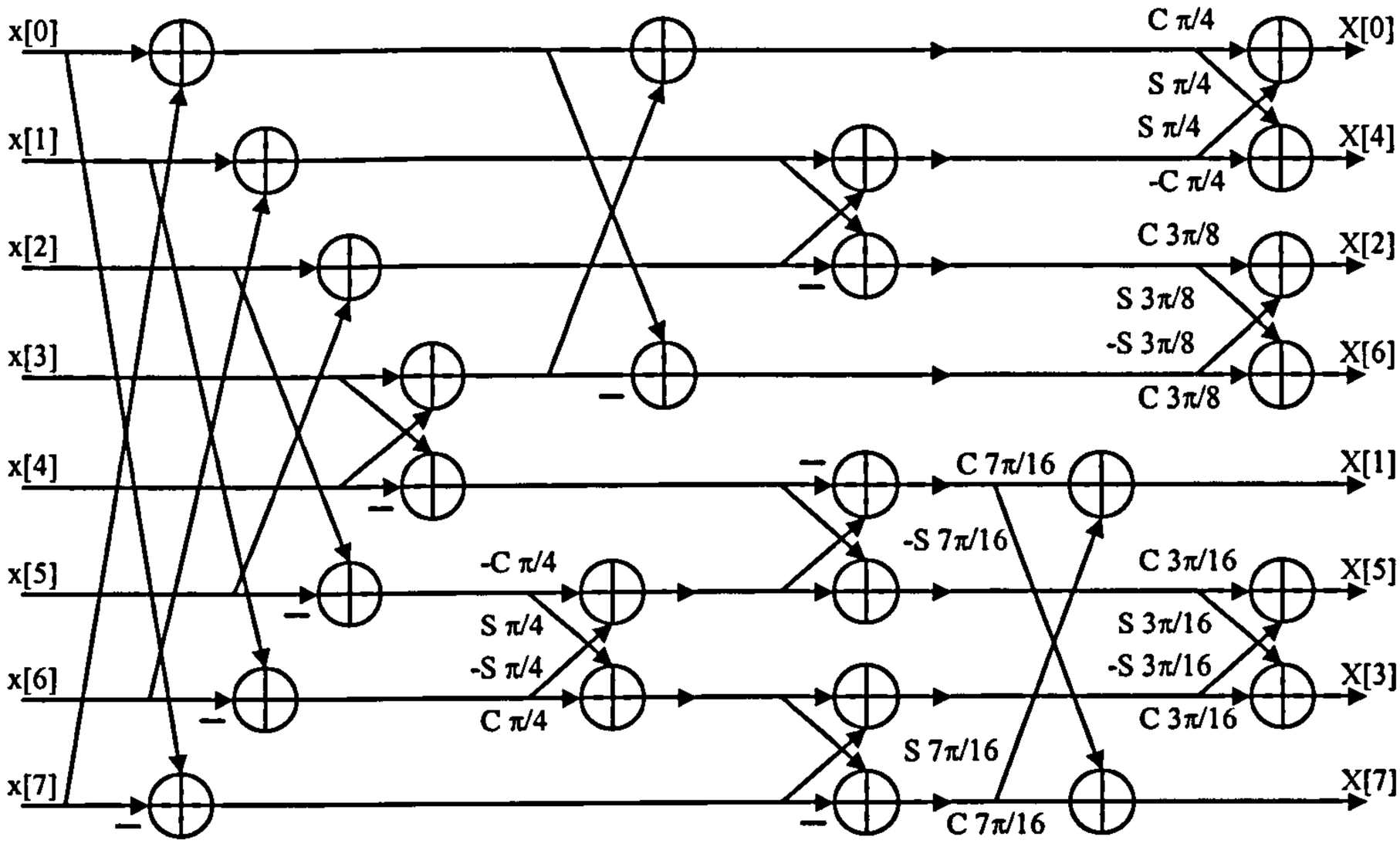
Table 5.1 Cosine Weights for Sample *(n)* of Input *(N)* at Frequency *(k)*

5.1.2 Chen's Fast DCT

Chen's FDCT implementation (*Figure 5.2*) resembled a *Fast Fourier Transform* (FFT) flow diagram, and was formed using butterfly computations and plane rotations [73]. These features correspond to the summation and rotation of weighting coefficient signs within Table 5.1. The mathematical functions represented by butterfly computations shown in Figure 5.2 are explained in Figure 5.3.

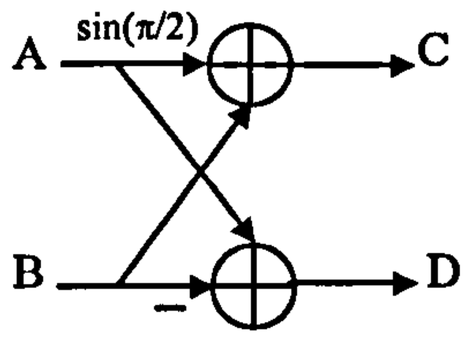
For an input sequence of length (N = 8) Chen's DCT took 20 multiplications and 26 additions to compute. These computation overheads have been reduced further using methods proposed by Hou [74] and Fieg [75], which reduce the number of computations through scaling input values and further exploitation of symmetry and redundancy within operations.

DCT algorithms such as these require calculation using floating-point multiplication and addition units. Such units require extensive silicon footprints and have lower operand throughput when compared to fixed-point implementations. Fixed-point arithmetic units can be used, but at the expense of introducing rounding errors within the result.



Where: Cn = cos(n), Sn = sin(n)

Figure 5.2 Chen's Fast Forward DCT



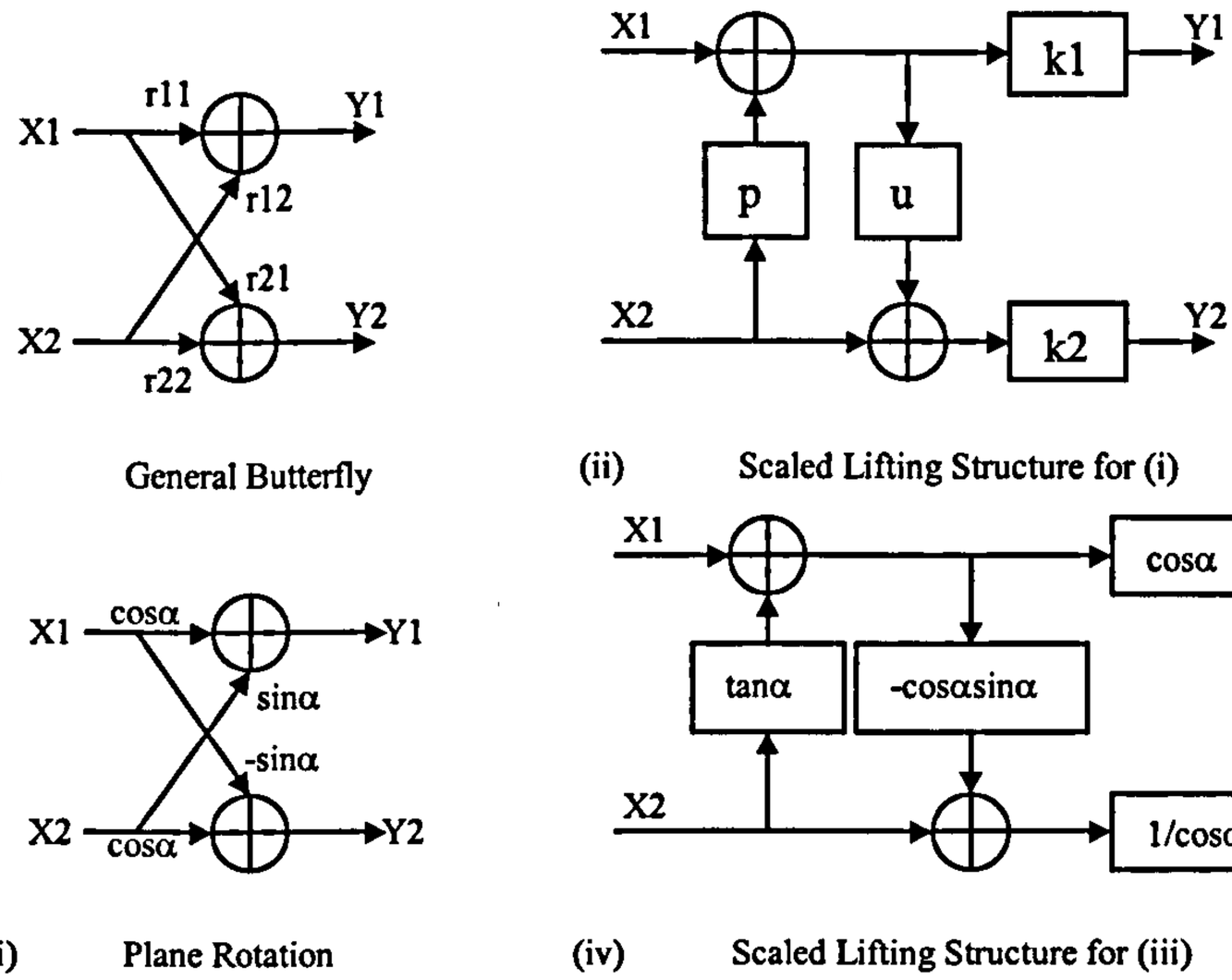
Where : C = Asin(π/2) + B
D = -B + A

Figure 5.3 Butterfly Operation

5.2 The BinDCT Algorithm

Tran and Liang [76] proposed a DCT mechanism more suited to fixed-point hardware implementation. This method was known as the *BinDCT*, and calculated forward and reverse transforms using a multiplier-less approximation of Chen's DCT. The basis of BinDCT function was to replace all plane rotations (e.g. C 3π/8, -S π/4) by a series of dyadic lifting-steps. Dyadic values are integer fix-point implementation friendly values of format $k/2^m$; Where k, m are integers.

The general butterfly structure and plane rotation within Figure 5.2 can be represented using lifting-structures as shown in Figure 5.4. Lifting-structures are also known as *shears* and *ladder structures*, and relate to the rotational operation of an inverse matrix.



(i) General Butterfly

(ii) Scaled Lifting Structure for (i)

(iii) Plane Rotation

(iv) Scaled Lifting Structure for (iii)

Figure 5.4 BinDCT Lifting-Structures

Figure 5.4-i illustrates that a butterfly computation can be represented (*Figure 5.4-ii*) using two lifting steps (p, u) and two scaling factors (k1, k2). Mathematically, the two lifting-step operations can be considered as two individual multiplication (p, u) and addition operations. The butterfly and lifting-step operations are shown in Equations 5.3 to 5.4.

$$Y1 = r11X1 + r12X2$$
$$Y2 = r21X1 + r22X2$$

Equation 5.3 DCT Butterfly Operation

$$Y1 = k1(X1 + pX2)$$
$$Y1 = k1X1 + k1pX2$$

$$Y2 = k2(u(X1 + pX2) + X2)$$
$$Y2 = k2uX1 + k2(1 + pu)X2$$

Equation 5.4 Lifting Step Operation

The dyadic values of (p) and (u) are calculated using Equation 5.5. Examples of dyadic values are listed in Table 5.2.

$$p = \frac{r12}{r11}$$

$$u = \frac{r11r21}{r11r22 - r21r12}$$

Equation 5.5 Calculation of Dyadic Coefficients

The outputs of Figure 5.3-ii (Y1, Y2) are adjusted by two scaling factors (k1, k2) as shown in Equation 5.6. Within the overall BinDCT structure, these individual scaling factors are absorbed into one operation either in the first or last stage depending upon whether a forward or reverse BinDCT calculation occurs.

$$k1 = r11$$

$$k2 = \frac{r11r22 - r21r12}{r11}$$

Equation 5.6 Calculation of Scaling Values

A Chen type plane rotation is shown in Figure 5.4-iii with the resultant scaled lifting structure depicted in Figure 5.4-iv. The dyadic values and scaling factors within Figure 5.4-iv are calculated using substitution within Equations 5.4 and 5.5. This process is shown in Equations 5.7 to 5.8 [76].

$$p = \frac{r12}{r11} \Rightarrow \frac{\cos(\alpha)}{\sin(\alpha)} = \tan(\alpha)$$

$$u = \frac{r11r21}{r11r22 - r21r12} \Rightarrow \frac{-\cos(\alpha)\sin(\alpha)}{(\cos(\alpha)\cos(\alpha)) - (-\sin(\alpha)\sin(\alpha))}$$

$$= \frac{-\cos(\alpha)\sin(\alpha)}{\cos(\alpha)^2 + \sin(\alpha)^2}$$

$$= \frac{-\cos(\alpha)\sin(\alpha)}{1} = -\cos(\alpha)\sin(\alpha)$$

Equation 5.7 Calculation of Dyadic Coefficients

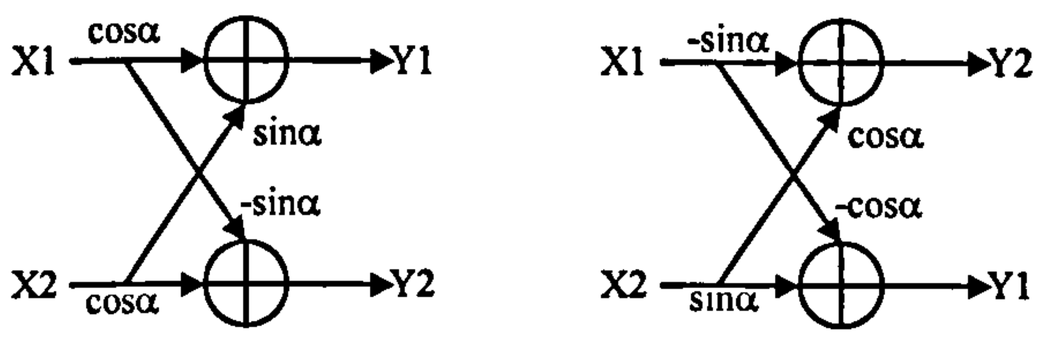
$$k1 = r11 \Rightarrow \cos(\alpha)$$

$$k2 = \frac{r11r22 - r21r12}{r11} \Rightarrow \frac{(\cos(\alpha)\cos(\alpha)) - (-\sin(\alpha)\sin(\alpha))}{\cos(\alpha)}$$

$$= \frac{\cos(\alpha)^2 + \sin(\alpha)^2}{\cos(\alpha)} = \frac{1}{\cos(\alpha)}$$

Equation 5.8 Calculation of Scaling Values

Using this substitution process (*Equations 5.7-5.8*) lifting structures for other plane rotation butterfly weights can be calculated. To enable integer fix-point implementation, BinDCT computations use dyadic values with limited fractional capability. This limitation causes the results generated by butterfly operations to be truncated, introducing a margin of error. To compensate for this effect, butterfly calculations resulting in small magnitude outputs, have their scaling weights transposed using trigonometric identities (*Figure 5.5*). This causes the output of the butterfly to be rearranged, which causes the forward BinDCT transform output to be out of sequence (*Figure 5.6*).



(i) Original Plane Rotation (ii) Transposed Plane Rotation

Where: $\cos^2(\alpha) > \sin_2(\alpha)$, use original plane rotation (i)

$\cos^2(\alpha) < \sin_2(\alpha)$, use transposed plane rotation (ii)

Figure 5.5 Transposition of Butterfly Operation

The use of dyadic coefficients (p, u) within the lifting structure enables a loss-less fixed-point approximation of the DCT. The approximation is loss-less since a characteristic of a lifting structure is that it can reconstruct an input from an output response without error if identical coefficient values are used in both operations. To ensure loss-less operations, the operand resolution of the hardware implementation must be at least that of the dyadic coefficients used. Rounding errors will otherwise be introduced into calculations, resulting in a reduction of BinDCT accuracy to approximating true DCT operation.

The structure of forward and reverse BinDCTs are shown in Figures 5.6 and 5.7, with the dyadic coefficients (u, p) determined for different accuracies in approximating DCT operation listed in Table 5.2. For clarity, input and output scaling factors were excluded from Figures 5.6 and 5.7 and are listed in Table 5.3. The raw output values generated by the reverse BinDCT operation however, have a magnitude four times greater than the actual value. This feature is caused through the summation of scaling factors.

Table 5.2 contains nine different configurations of dyadic lifting scheme coefficients (C1-C9). These values were inserted into lifting structures as indicated by identities (P_n) and (U_n). Each configuration generated an approximation of the DCT algorithm, but with varying degrees of accuracy. Configuration (C1) was the most accurate, with (C9)

being the least accurate. All nine of these configurations could be used to provide loss-less compression.

	BinDCT Configuration C1 - C9								
Coefficients	*C1*	*C2*	*C3*	*C4*	*C5*	*C6*	*C7*	*C8*	*C9*
P1	0.40625	0.4375	0.40625	0.4375	0.375	0.5	0.5	1	0
U1	0.34375	0.375	0.34375	0.375	0.375	0.375	0.5	0.5	0
P2	0.6875	0.625	0.6875	0.625	0.875	0.875	1	1	0
U2	0.48675	0.4375	0.48675	0.4375	0.5	0.5	0.5	0.5	0
P3	0.1875	0.1875	0.1875	0.1875	0.1875	0.1875	0.25	0	0
U3	0.1875	0.1875	0.1875	0.1875	0.1875	0.25	0.25	0	0
P4	0.40625	0.40625	0.4375	0.4375	0.4375	0.4375	0.5	0	0
U4	0.6875	0.6875	0.6875	0.6875	0.6875	0.75	0.75	0.5	0
P5	0.40625	0.40625	0.375	0.375	0.375	0.375	0.5	0.5	0

Table 5.2 BinDCT Coefficient Configurations

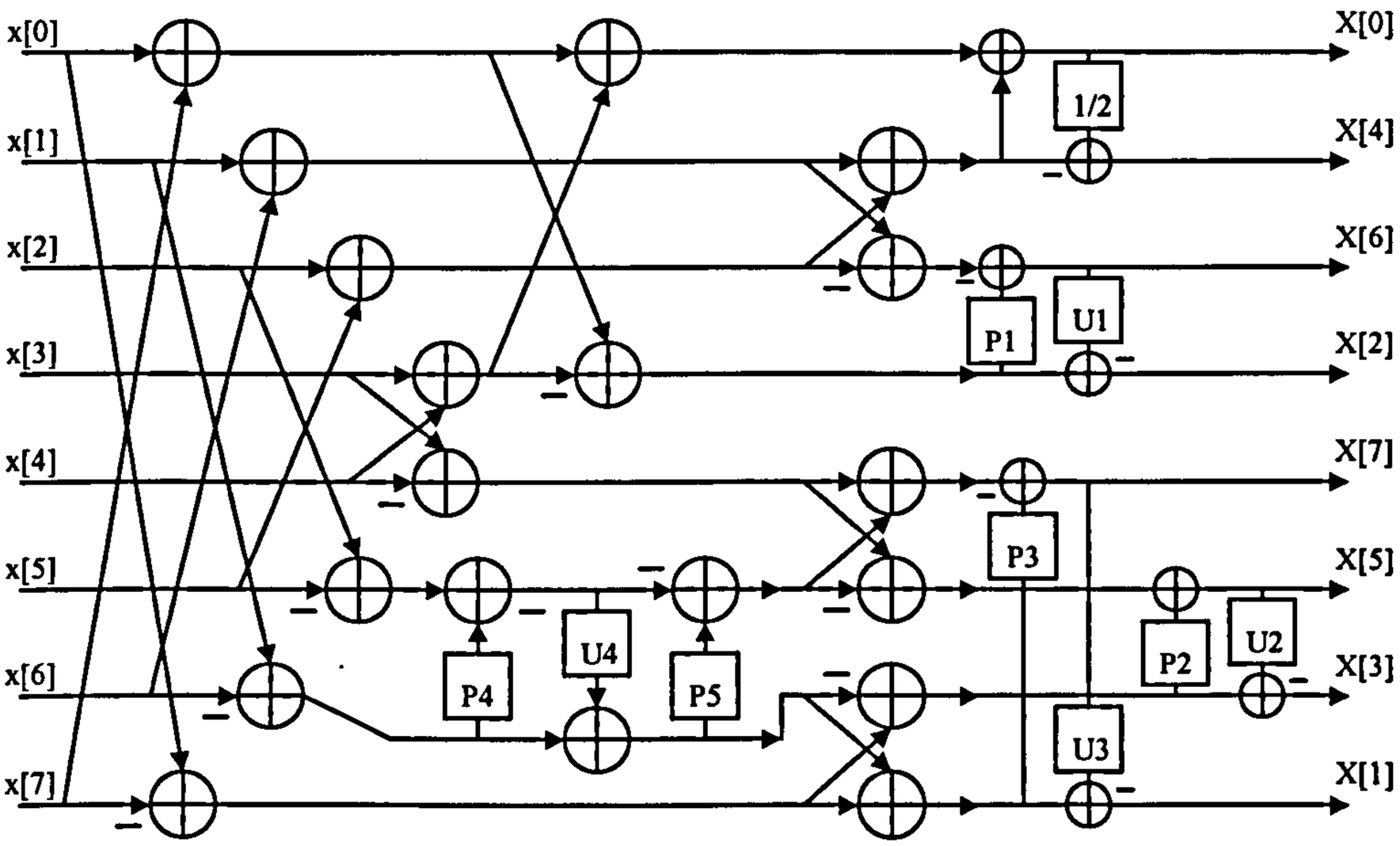


Figure 5.6 Forward BinDCT Flow Diagram

To compare the output of a forward BinDCT with that of the DCT algorithm, the results of the BinDCT must be scaled by the factors shown in Table 5.3. If scaled transform coefficients were then applied to the reverse BinDCT, they had to be re-adjusted (scaled) prior to computation. If forward BinDCT outputs were not scaled, their results could be applied directly to the reverse BinDCT. Regardless of whether result scaling occurred, the output of the reverse BinDCT had to be divided by four.

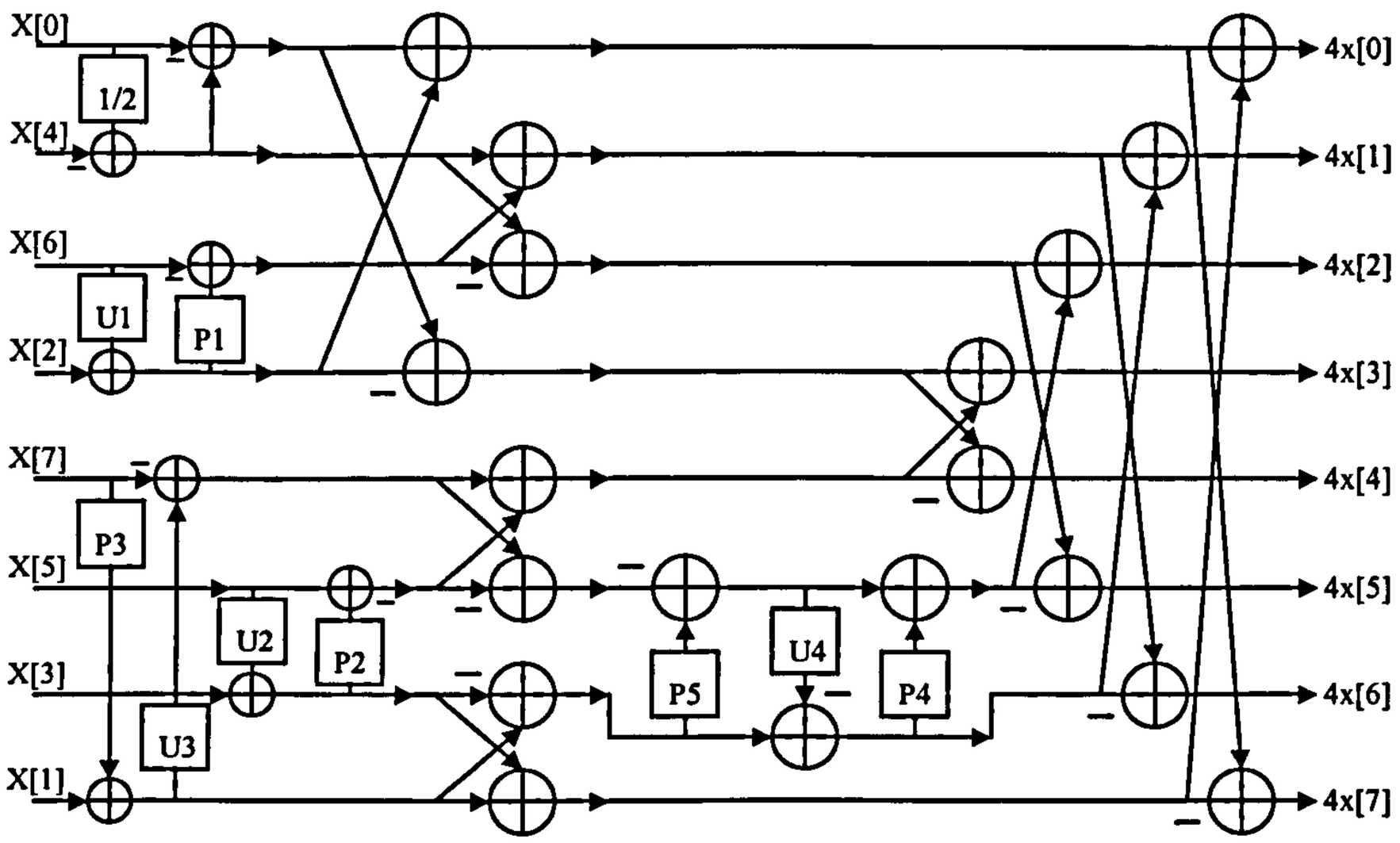


Figure 5.7 Reverse BinDCT Flow Diagram

Tran's paper [76], detailed how the BinDCT could be implemented without using multipliers, but instead using successive shifts and additions. This was possible since dyadic coefficients could be represented using fixed-point binary notation. However, work conducted previously in *Section 4.2.4*, determined that binary multiplication was best performed on an FPGA through successive shifts and additions. It was concluded therefore that the BinDCT was calculated using the shift and add method of distributed multiplication.

Compared to existing floating-point based DCT implementations, the integer friendly structure of the BinDCT appeared most suited for implementation within FPGA type architectures. Furthermore, the complexity of the hardware configuration was dependant

upon the BinDCT configuration used, which itself was determined by the accuracy of the DCT approximating required. Investigations conducted to determine the relationship of this feature and DCT compression ratios are described next in *Section-5.3*.

Forward BinDCT		**Reverse BinDCT**	
Applied to Output	*Scaling Factor*	*Applied to Input*	*Scaling Factor*
X[0]	(sinπ/4)/2	X[0]	2/(sinπ/4)
X[4]	sinπ/4	X[4]	1/(sinπ/4)
X[6]	(sin3π/8)/2	X[6]	2/(sin3π/8)
X[2]	1/(2sin3π/8)	X[2]	2sin3π/8
X[7]	(sin7π/16)/2	X[7]	2/(sin7π/16)
X[5]	(cos3π/16)/2	X[5]	2/(cos3π/16)
X[3]	1/(2cos3π/16)	X[3]	2cos3π/16
X[1]	1/(2sin7π/16)	X[1]	2sin7π/16

Table 5.3 BinDCT Scaling Factors

5.3 Dynamic BinDCT Investigation

The BinDCT implemented fixed-point multiplier-less approximations of DCT operation through use of lifting ladder structures. These calculations were computed with varying degrees of accuracy in approximating true DCT operation.

The operational characteristics of all BinDCT configurations were evaluated using five input sequences as illustrated in Figure 5.8. These sequences were generated to reflect the different frequency content and structure encountered within signal compression operations. Each input applied contained eight operands in the range of 0 to 255 (8-bit). Sequence-(i) represented a ramp function, (ii) a constant level, (iii) a Mexican hat function, (iv) a step function, and (v) a spike function.

Results for forward and reverse DCT and BinDCT operations for these input sequences were generated using custom software written in C++. The implementation of the transforms within software reflected the structure of processing architecture required for hardware implementation. Initially results were generated for all nine BinDCT

configurations but after preliminary analysis of the data work focussed upon configurations C1 and C9.

This decision was based upon configuration C1 providing the most accurate approximation of the DCT, whereas C9 required the least number of computations to calculate, hence provided the simplest hardware implementation.

5.3.1 Transform Characteristics

The forward and reverse transforms of the five input sequences generated through the software implementation of DCT and BinDCT configurations (C1 and C9) are listed in Tables 5.4 to 5.8. The results contain the original input sequence, the transform outputs (forward and reverse directions denoted by 'F' and 'R' respectively), and the calculated *Root Mean Square Error* (RMSE) (*Equation 5.9*, [77]) for each transform configuration and direction; For clarity, the results in the tables have be rounded to three and four (RMSE) decimal places.

$$RMSE = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(Xi - T)^2}$$

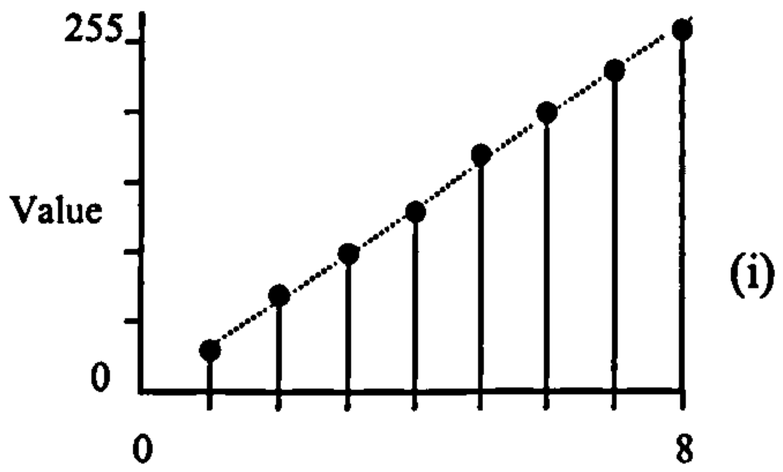
Where: Xi = ith value of group m values

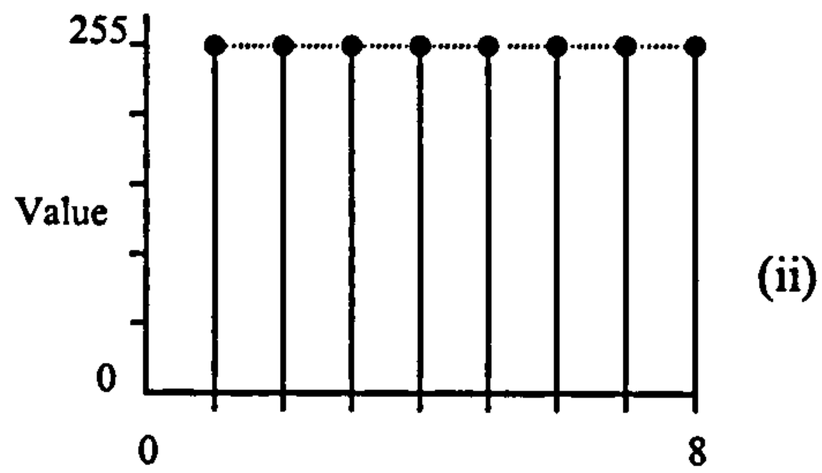
T = Target Value

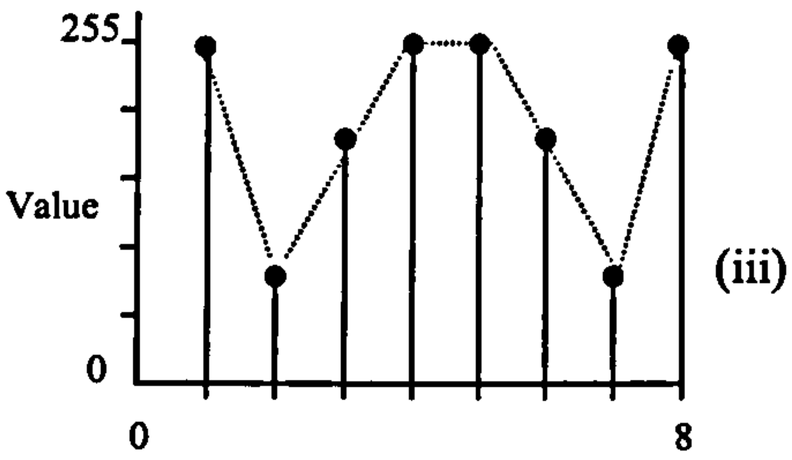
Equation 5.9 Root Mean Square Error (RMSE)

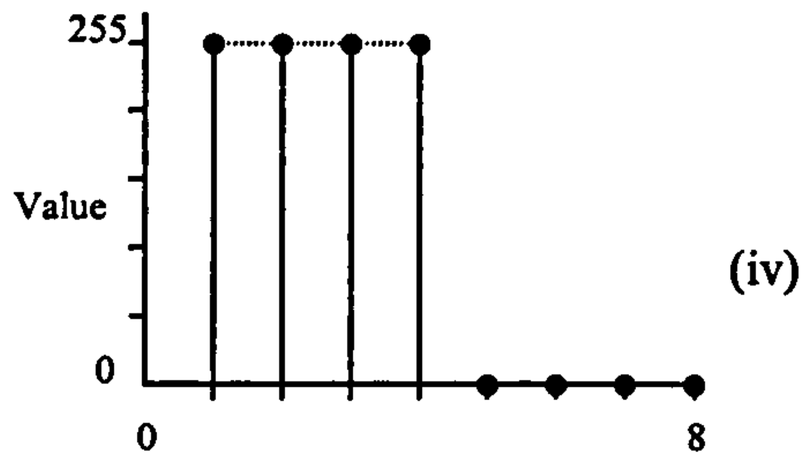
n	Input [0 to (N-1)]	FDCT	RDCT	FBinDCT-C1	RBinDCT-C1	FBinDCT-C9	RBinDCT-C9
0	31	404.819	31.000	404.819	31.000	404.819	31.000
1	63	-206.57	63.000	-206.920	63.000	-196.271	63.000
2	95	0.191	95.000	0.186	95.000	0.000	95.000
3	127	-21.453	127.000	-20.305	127.000	-37.885	127.000
4	159	-0.354	159.000	-0.354	159.000	-0.354	159.000
5	191	-5.938	191.000	-7.273	191.000	-53.214	191.000
6	224	-0.462	224.000	-0.462	224.000	-0.462	224.000
7	255	-1.345	255.000	-0.380	255.000	-31.385	255.000
RMSE		N/A	N/A	0.7206	0.0000	20.9569	0.0000

Table 5.4 Transform Outputs for Data Sequence-(i) Ramp Function









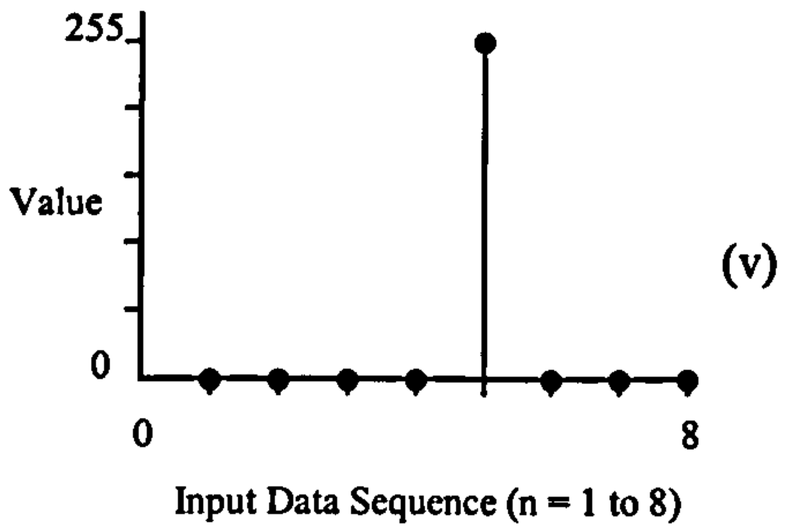


Figure 5.8 BinDCT Input Sequence Characteristics

n	Input [0 to (N-1)]	FDCT	RDCT	FBinDCT-C1	RBinDCT-C1	FBinDCT-C9	RBinDCT-C9
0	255	721.249	255.000	721.249	255.000	721.249	255.000
1	255	0.000	255.000	0.000	255.000	0.000	255.000
2	255	0.000	255.000	0.000	255.000	0.000	255.000
3	255	0.000	255.000	0.000	255.000	0.000	255.000
4	255	0.000	255.000	0.000	255.000	0.000	255.000
5	255	0.000	255.000	0.000	255.000	0.000	255.000
6	255	0.000	255.000	0.000	255.000	0.000	255.000
7	255	0.000	255.000	0.000	255.000	0.000	255.000
RMSE		N/A	N/A	0.0000	0.0000	0.0000	0.0000

Table 5.5 Transform Outputs for Data Sequence-(ii) Constant Level

n	Input [0 to (N-1)]	FDCT	RDCT	FBinDCT-C1	RbinDCT-C1	FBinDCT-C9	RBinDCT-C9
0	255	540.937	255.000	540.937	255.000	540.937	255.000
1	85	0.000	85.000	0.000	85.000	0.000	85.000
2	170	-32.528	170.000	-31.626	170.000	0.000	170.000
3	255	0.000	255.000	0.000	255.000	0.000	255.000
4	255	180.312	255.000	180.312	255.000	180.312	255.000
5	170	0.000	170.000	0.000	170.000	0.000	170.000
6	85	78.530	85.000	78.530	85.000	78.530	85.000
7	255	0.000	255.000	0.000	255.000	0.000	255.000
RMSE		N/A	N/A	0.0102	0.0000	13.2260	0.0000

Table 5.6 Transform Outputs for Data Sequence-(iii) Mexican Hat

n	Input [0 to (N-1)]	FDCT	RDCT	FBinDCT-C1	RBinDCT-C1	FBinDCT-C9	RBinDCT-C9
0	255	360.624	255.000	360.624	255.000	360.624	255.000
1	255	326.772	255.000	325.902	255.000	259.996	255.000
2	255	0.000	255.000	0.000	255.000	0.000	255.000
3	255	-114.747	255.000	-115.860	255.000	0.000	255.000
4	0	0.000	0.000	0.000	0.000	0.000	0.000
5	0	76.672	0.000	78.558	0.000	212.025	0.000
6	0	0.000	0.000	0.000	0.000	0.000	0.000
7	0	-64.999	0.000	-65.876	0.000	0.000	0.000
RMSE		N/A	N/A	0.8889	0	70.8618	0

Table 5.7 Transform Outputs for Data Sequence-(iv) Step Function

n	Input [0 to (N-1)]	FDCT	RDCT	FBinDCT-C1	RBinDCT-C1	FBinDCT-C9	RBinDCT-C9
0	0	90.156	0.000	90.156	0.000	90.156	0.000
1	0	-24.874	0.000	-24.375	0.000	0.000	0.000
2	0	-117.795	0.000	-118.733	0.000	-138.005	0.000
3	0	70.835	0.000	71.880	0.000	0.000	0.000
4	255	90.156	255.000	90.156	255.000	90.156	255.000
5	0	-106.012	0.000	-106.012	0.000	-106.012	0.000
6	0	-48.792	0.000	-47.854	0.000	0.000	0.000
7	0	125.050	0.000	125.050	0.000	125.050	0.000
RMSE		N/A	N/A	0.6226	0.0000	32.4527	0.0000

Table 5.8 Transform Outputs for Data Sequence-(v) Spike Function

Tables 5.4 to 5.8 list the RMSE of BinDCT configurations C1 and C9 compared to true DCT operation ($FDCT_{ii}$, RDCT). Results indicated that RMSEs were dependant not only upon the BinDCT configuration used, but also the frequency content of the input sequence.

The forward transform of BinDCT configuration C9 generated the largest RMSE, with a maximum error of 70.8618 determined (*Table 5.7, sequence-iv*). This high value was attributed to the reduced accuracy of DCT approximation of FBinDCT-C9 compared to FBinDCT-C1. In comparison transform FBinDCT-C1 largest RMSE was 0.8889 (*Table 5.7*), reflecting the increased accuracy of DCT approximation for FBinDCT-C1 compared to FBinDCT-C9.

The reverse transform RMSEs obtained for both configurations were zero, which was attributed the operational characteristics of lifting ladder structures. With lifting ladder operation, if identical coefficients were used within forward and reverse transforms, the original data was reconstructed without loss.

5.3.2 BinDCT Compression

By representing data using its frequency components, spectral frequency redundancy invisible in the time-domain can be extracted and used to compress the signal. Within the frequency domain, frequency components at zero can be removed without compromising the representation of information in the time-domain.

The compression ratio can be enhanced by quantisisation of the forward transform coefficients, to increase the number at zero. Table 5.9 lists number of frequency components generated at zero. These indicated that prior to quantisisation, overall BinDCT-C9 achieved the greatest data redundancy.

To investigate the DCT coding-gain [76] of each configuration, the forward transform coefficients generated were quantisised, with results generated listed in Tables 5.10 to 5.14. To provide comparisons, transforms were evaluated using the number of forward transform coefficients at zero as benchmarks (n:N), RMSE compared to true DCT operation, and maximum error of reconstructed data; Where 'n' is the number of coefficient(s) at zero, and 'N' is the original input sequence length.

	Ratio of Coefficients at Zero : Total Number of Coefficients		
Input Sequence	*DCT*	*BinDCT-C1*	*BinDCT-C9*
(i)	0:8	0:8	1:8
(ii)	7:8	7:8	7:8
(iii)	4:8	4:8	5:8
(iv)	3:8	3:8	5:8
(v)	0:8	0:8	3:8

Table 5.9 Inherent DCT/BinDCT Compression

The RMSEs generated for both DCT and BinDCT configurations represents the error obtained between the original and reconstructed data sequences. The maximum BinDCT error was determined by comparing BinDCT reconstructed output data against the original input.

Tables 5.10 to 5.14 indicate the ability of the DCT and BinDCT transforms to reconstruct data using quantisised forward transform coefficients. In Table 5.14 the maximum error (*Max Error*) of BinDCT-C9 is zero for three zero coefficients. This value is attributed to the fact that BinDCT-C9 originally had a loss-less zero coefficient ratio of 3:8.

	DCT		BinDCT-C1		BinDCT-C9	
Zero Coefficients	*RMSE*	*Max Error*	*RMSE*	*Max Error*	*RMSE*	*Max Error*
1	0.0675	0.088	0.0656	0.086	0.000	0.000
2	0.1420	0.213	0.1412	0.211	0.1250	0.125
3	0.2165	0.375	0.1950	0.398	0.2165	0.375
4	0.5222	0.784	0.2549	0.477	11.3158	16.125
5	2.1635	3.661	2.5540	3.993	15.8770	16.125
6	7.8874	10.825	7.7107	10.653	27.6420	48.125
7	73.4582	112.125	73.4582	112.125	73.4582	112.125

Table 5.10 Reconstructed Quantisised Data: Sequence-(i)

	DCT		BinDCT-C1		BinDCT-C9	
Zero Coefficients	*RMSE*	*Max Error*	*RMSE*	*Max Error*	*RMSE*	*Max Error*
7	0.000	0.000	0.000	0.000	0.000	0.000

Table 5.11 Reconstructed Quantisised Data: Sequence-(ii)

	DCT		BinDCT-C1		BinDCT-C9	
Zero Coefficients	*RMSE*	*Max Error*	*RMSE*	*Max Error*	*RMSE*	*Max Error*
5	11.5004	15.026	11.1501	14.609	0.000	0.000
6	30.0520	42.500	30.0520	42.500	30.0520	42.500
7	70.4783	106.250	70.4783	106.250	70.4783	106.250

Table 5.12 Reconstructed Quantisised Data: Sequence-(iii)

	DCT		BinDCT-C1		BinDCT-C9	
Zero Coefficients	*RMSE*	*Max Error*	*RMSE*	*Max Error*	*RMSE*	*Max Error*
4	22.9806	31.875	23.3158	32.403	0.000	0.000
5	35.5376	63.750	36.1132	64.419	0.000	0.000
6	53.9331	95.625	54.8250	97.534	90.1561	127.500
7	127.500	127.500	127.500	127.500	127.500	127.500

Table 5.13 Reconstructed Quantisised Data: Sequence-(iv)

	DCT		BinDCT-C1		BinDCT-C9	
Zero Coefficients	*RMSE*	*Max Error*	*RMSE*	*Max Error*	*RMSE*	*Max Error*
1	8.7943	12.198	8.6091	11.953	0.000	0.000
2	19.3629	32.880	19.0259	32.437	0.000	0.000
3	31.6564	64.186	32.0217	64.718	0.000	0.000
5	55.0831	95.189	55.2939	95.438	45.0781	63.750
6	66.6256	139.262	66.4961	138.644	63.7500	127.500
7	78.5712	193.676	78.5231	193.491	78.0775	191.250

Table 5.14 Reconstructed Quantisised Data: Sequence-(v)

Results indicated that as the number of forward transform zero coefficients increased the RMSE between original input and recovered output sequences increased. Furthermore, BinDCT-C9 output degradation was greater compared to DCT and BinDCT-C1 for each comparative quantisised step. This was attributed to the BinDCT-C9 approximation of the DCT being less accurate than that of BinDCT-C1.

For high frequency content inputs (sequences-i & -iii) for a given accuracy, greater forward transform coefficient redundancy was obtained using BinDCT-C1 compared to BinDCT-C9. The RMSE indicated that although errors were introduced, the reverse

transform output still contained the characteristics features of the original input (lossy compression). Compared to the BinDCT-C1, the BinDCT-C9 output for the same number of coefficients at zero was distorted. This is illustrated in Figure 5.9 for sequence-(i), with five forward transform coefficients set to zero. Figure 5.10 highlights the errors measured between original input and reconstructed data for each transform.

For low frequency content input sequences (sequences-iv & -v), BinDCT-C9 generated greater loss-less compression ratio than BinDCT-C1. This was true for the BinDCT-C1 even when quantisisation was used to increase the compression ratio. For example for both BinDCT and DCT configurations, Figure 5.11 represents sequence-v reconstructed with three forward coefficients set to zero.

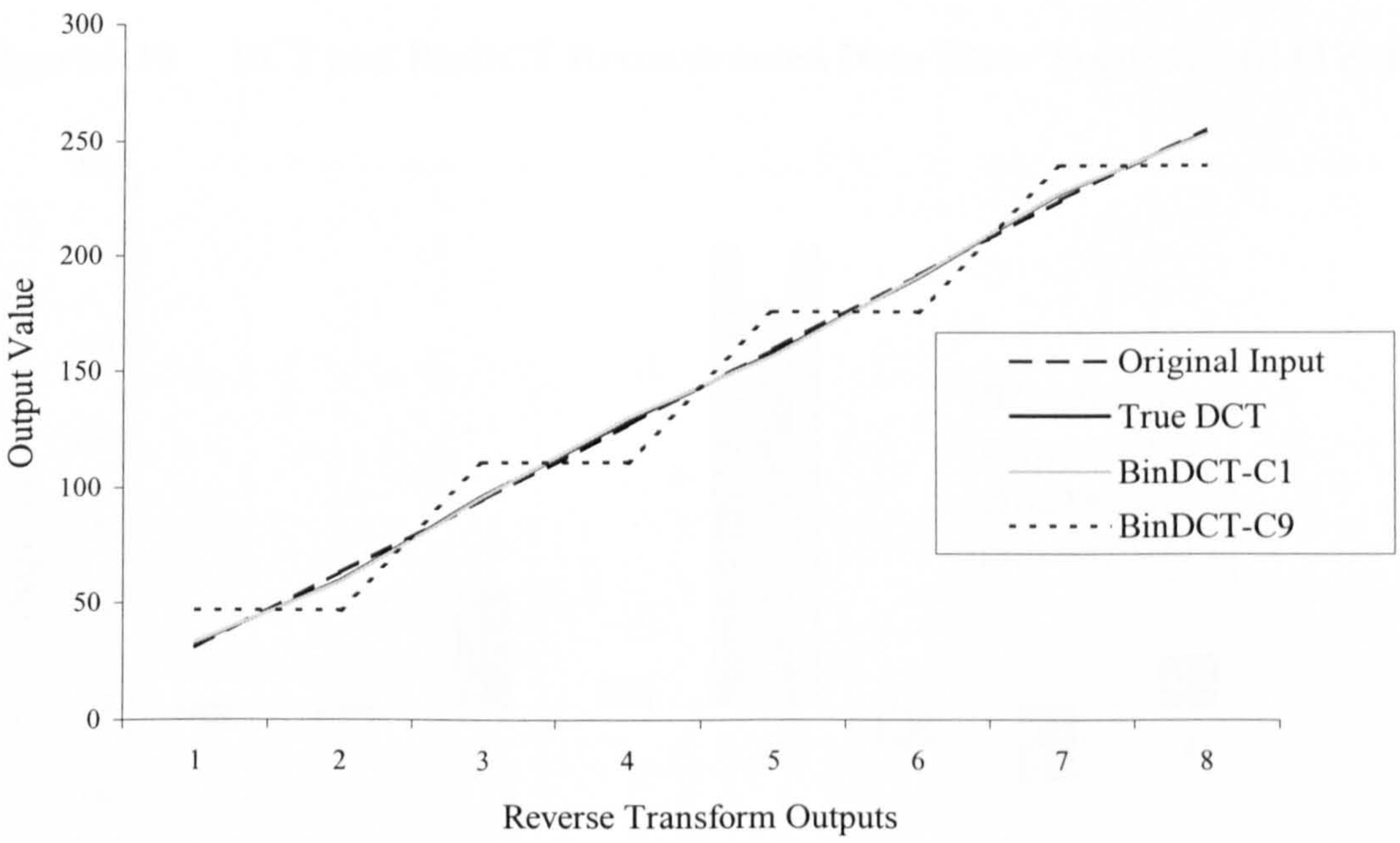


Figure 5.9 BinDCT Reverse Transform Output of Sequence–(i) (5 Zeros)

Figure 5.10 DCT and BinDCT Reconstructed Data Error Sequence–(i) (5 Zeros)

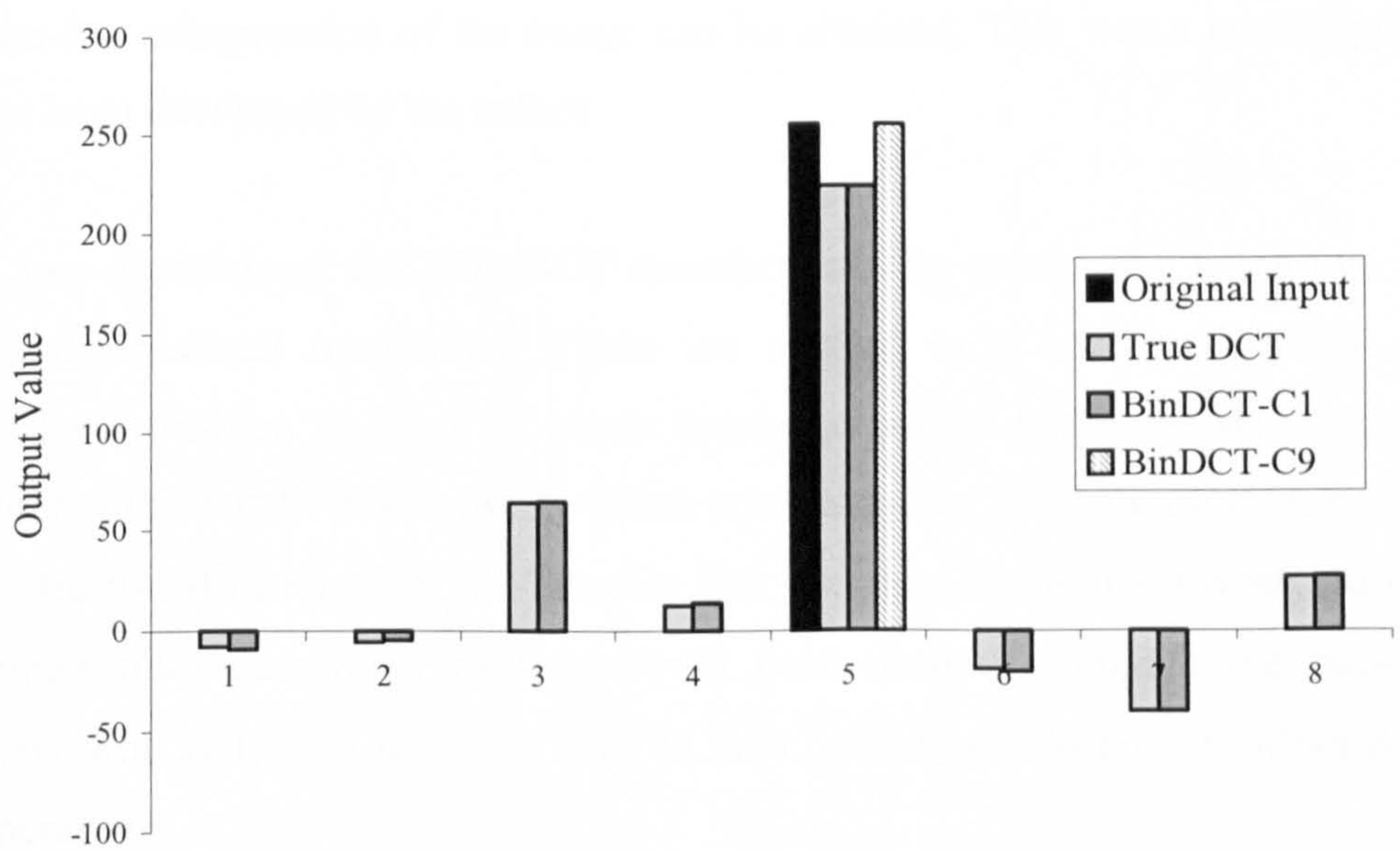


Figure 5.11 BinDCT Reverse Transform Output of Sequence-(v) (3 Zeros)

From these experiments it was determined that to achieve high BinDCT compression ratios, input sequences containing high frequency content should be compressed and quantisised using BinDCT-C1. Input sequences containing low frequency content

should instead use BinDCT-C9. Similar results could be obtained using BinDCT-C1 but at the expense of using more complex processing architecture than actually required.

To implement these concepts, the frequency content of input sequence must be known. In reality this does not occur, therefore to implement the most efficient BinDCT operation in terms of compression and complexity of design, the processing architecture must be able to switch between the BinDCT-C1 and BinDCT-C9 during run-time as dictated by system operation.

5.3.3 Two-Dimensional Dynamic BinDCT Operation

Within image-processing applications, images can be compressed using two-dimensional DCT transform operations, constructed using two individual one-dimensional transforms (*Equations 5.1 and 5.*2). Through updating the BinDCT configuration used for each two-dimensional transform during this process, the optimal loss-less compression of the image can be obtained. This was a novel application that has been developed by the author.

A two-dimensional DCT/BinDCT transform can be constructed using two independent one-dimensional transforms. These are applied in a horizontal (*row*) and vertical (*column*) fashion to block of pixels known as a *tile*. A tile can be of any size, but a common size is 64 pixels (8x8), which equates to row and column sizes of 8 pixels each as illustrated in Figure 5.12. This tile size was used during the investigation since one-dimensional transforms of size (N=8) were used to compute the two-dimensional transform. A typical image of size 512x512 pixels would contain 4096 8x8 pixel tile operations.

To compress an image, a two-dimensional forward transform operation could be applied. This was performed by first applying a one-dimension forward transform to either each row or column (choice was dependant upon the user), with the resultant

coefficients generated replacing the original data within the tile. This is shown in Figure 5.12.

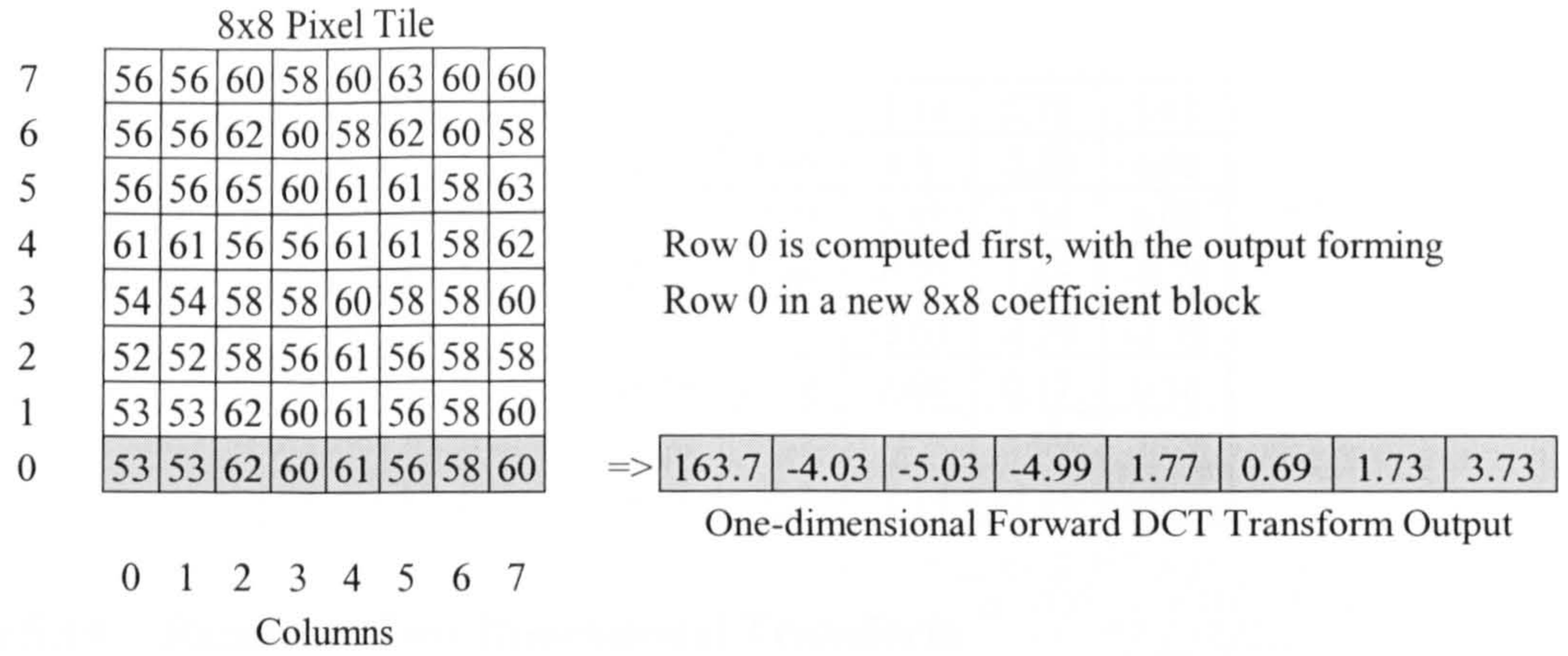


Figure 5.12 One-Dimensional Forward Transform (Row)

A second one-dimensional transform was then applied to the each column using the coefficient generated previous as the input as shown in Figure 5.13.

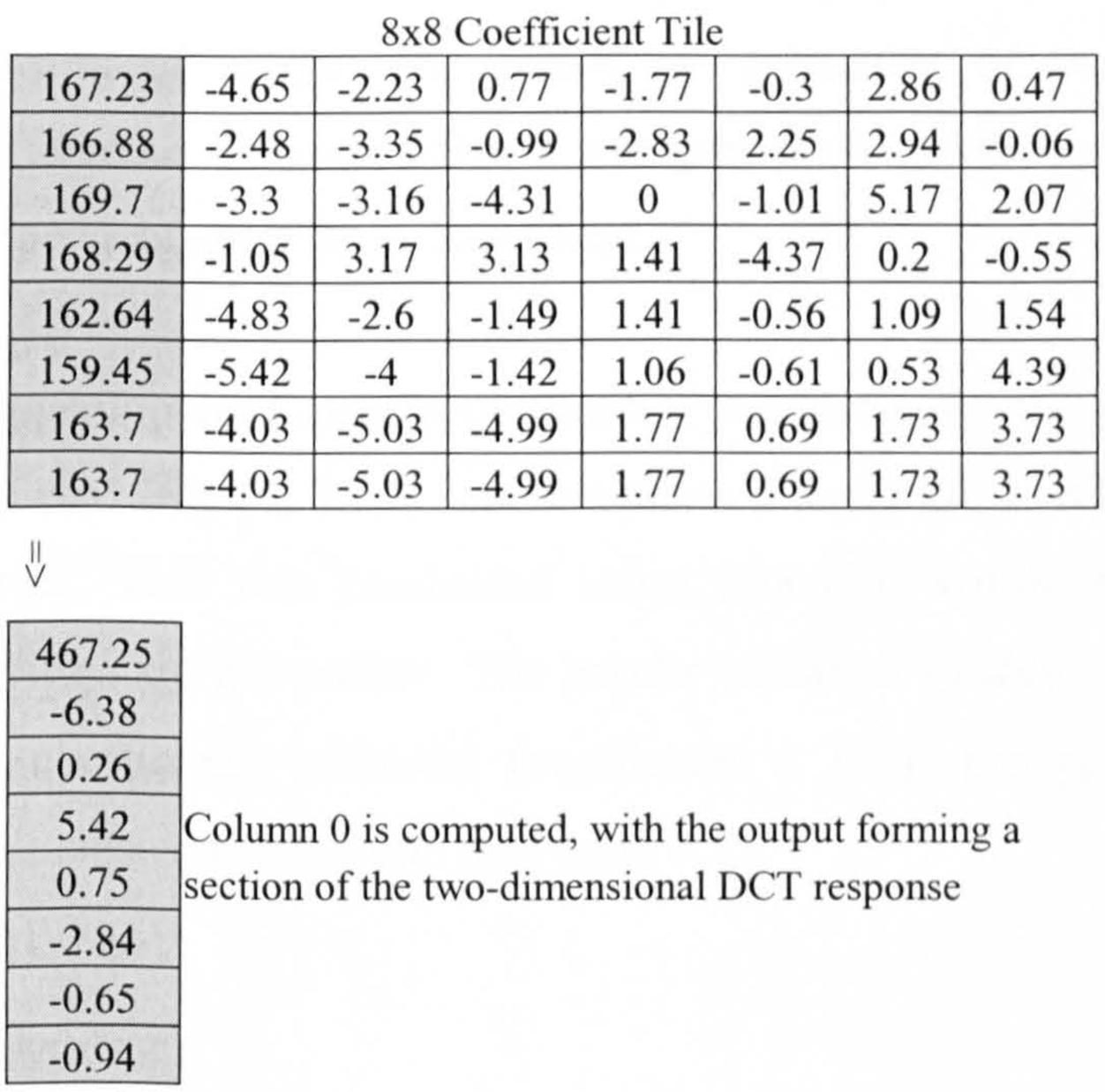


Figure 5.13 One-Dimensional Forward Transform (Column)

Completion of the forward two-dimensional DCT gives a row and column result as shown in Figure 5.14

467.25	-10.53	-7.87	-5.05	1	-1.14	5.75	5.41
-6.38	-1.28	-2.86	-4.18	3.96	0.3	-2.23	4.02
0.26	-0.89	-3.87	-2.77	-1.71	3.32	1.34	0.98
5.42	2.49	1.06	-2.05	0.48	-0.72	1.65	-0.75
0.75	0.24	3.13	3.23	1	-2.07	-1.59	-1.75
-2.84	-0.84	-2.45	-1.29	-1.16	2.68	0.17	0.16
-0.65	-1.55	-0.91	-0.98	0.45	-1.1	1.09	1.89
-0.94	1.47	2.69	4.05	-0.5	-1.18	-2.14	-0.8

Figure 5.14 Resultant Two-Dimensional Transform

To reconstruct the original image, one-dimension reverse transforms are applied to the coefficient tile, first upon each column then upon each row. Using two-dimension DCT transforms, images can be compressed through removal of zero and small AC frequency coefficients. Through removing only zero coefficients loss-less compression is achieved. However, if coefficient values are quantisised, errors will be introduced within the reconstructed image. The non-linear properties of the human eye in distinguishing differences between colours and grey-scale gradients, causes the error threshold to be dependent upon the source image.

To investigate dynamic two-dimensional BinDCT compression, the image processing standard image 'Lena' (*Figure 5.15*) was compressed using DCT, static, and dynamic BinDCT transforms. This was conducted using functions within the XC6200ADS software tools written by the author. The results obtained (*Table 5.15*) indicate the number of zero coefficients obtained (coefficient < 0.5) through performing the appropriate two-dimensional transform on Figure 5.15.

Figure 5.15 Lena Benchmark Image

	True DCT	BinDCT-C1	BinDCT-C9	Dynamic BinDCT
Zero Coefficients	38899	38777	33359	40891

Table 5.15 Loss-less Compression of Lena Image

From Table 5.15 it was determined that 529 out of 4096 (13%) tile operations exhibited greater inherent loss-less compression using configuration BinDCT-C9 than BinDCT-C1. To determine which BinDCT configuration generated the greatest compression for each tile operation, XC6200ADS software functions were written to analyse the tiles inherent coding gain for each BinDCT configuration. Each tile was then computed using the BinDCT configuration that generated the greatest number of forward transform zero coefficients. Using this information the distribution of BinDCT configurations within the source image (*Figure 5.15*) was determined. This is shown in Figure 5.16, with the corresponding BinDCT-C9 tile operation locations represented in black and BinDCT-C1 operations in white.

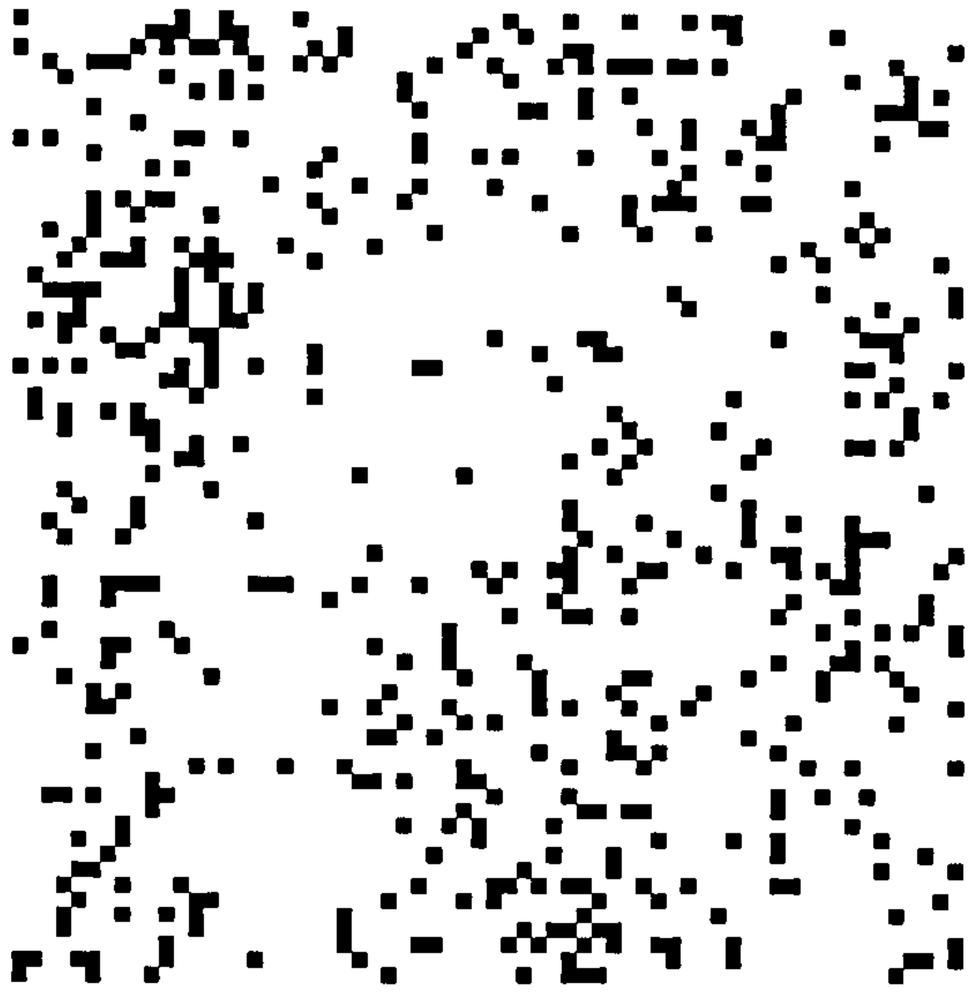


Figure 5.16 BinDCT Configuration Distribution

The source image was compressed by computing each tile using the appropriate BinDCT configuration. This technique resulted in an additional 1992 forward transform coefficients generated being at zero (*Table 5.15, 40891-38899*).

To reconstruct the original image the forward BinDCT transform configuration used for each pixel tile had to be known. This was required to allow coefficients to be reconstructed using the correct reverse BinDCT transform. This information was encoded within the compressed data using *Run Length Coding* techniques [72].

Through adapting the BinDCT configuration as required for each tile, the volume of transform coefficients at zero increased when compared to static BinDCT and traditional DCT implementations. The development of this technique provided the basis for dynamic hardware implementation of the BinDCT algorithm.

5.4 Summary

This chapter has described developing an application were system throughput and compression has been improved through dynamic hardware implementation. This application is a recently develop integer-friendly approximation of the DCT called the BinDCT.

To improve loss-less compression within an image, a dynamic two-dimensional BinDCT algorithm application has been developed. Through actively swapping the BinDCT configuration used to compute each 8x8 pixel tile, the compression ratio has been increased. Swapping active BinDCT configurations improves operand throughput since the computational complexity of BinDCT-C9 is approximately half that of BinDCT-C1.

To develop a hardware implementation of this application, dynamic hardware was required to switch between different BinDCT configurations. The development and operation of this architecture is described next in *Chapter-6*.

Chapter 6

Dynamic XC6264 BinDCT Coprocessor

Introduction

This chapter describes the construction of XC6264 based dynamic BinDCT hardware, inserted within the TIM-40 architecture. XC6264 hardware appeared as RTR C40 memory mapped coprocessor peripherals.

The XC6200DS configuration mode used and C40 processor interface operation is described in *Section-6.1*. Next the design and construction of the underlying XC6264 BinDCT architecture is described in *Section-6.2*. *Section-6.3* details the implementation of fixed BinDCT coprocessor operation, whilst *Section-6.4* expands to RTR dynamic operation.

Section-6.5 compares one and two-dimensional XC6264 BinDCT transform operations against software results generated in *Chapter-5*. Conclusions derived from the work present in this chapter are then presented in *Section-6.6*.

The CLC array floor plans of key XC6264 FPGA designs are detailed in *Appendix-VI*, whilst overviews of both XC6200 and C40 DSP device architectures provided in *Appendix-III*.

6.1 Design Overview

To interface dynamic FPGA and DSP processing hardware, both components must interact at hardware and user software levels. This capability was made possible through the development of the XC6200DS *(Section-3.3)* and upgrading existing C40 TIM-40 modules *(Section-3.4.2)*.

The dynamic BinDCT coprocessor architectures developed were constructed through combining both XC6200DS C40 coprocessor *(Section-3.4.2)* and self-configuration

(Section-3.4.3) modes of operation. The XC6200DS prototype environment mode *(Section-3.2.1)* was also used during system development, enabling real-time XC6264 hardware debugging (*Section-4.2).*

The base architecture of the XC6264 dynamic coprocessor consisted of three sections as shown in Figure 6.1. These were the self-configuration control mechanism, C40 Global buses interface, and coprocessor function (represented by unused CLCs in *Figure 6.1*). The BinDCT processor hardware developed was implemented within the coprocessor function area (approximately 77% of XC6264 CLCs available).

The C40 and XC6264 interacted through a parallel interface formed using the C40s Global buses interface and hardware configured within XC6264 CLC array. The XC6264 interface consisted of control state machines, a four to sixteen-bit address decoder (*A15–A0*), and bi-directional data-bus (*D7-D0*). Addresses (*A15-A0*) were used to control aspects of coprocessor operation including operand transfer and instigating RTR. C40 programs could manage such functions through accessing *A15-A0* (XC6264 address space) mapped within the C40s Global bus address space.

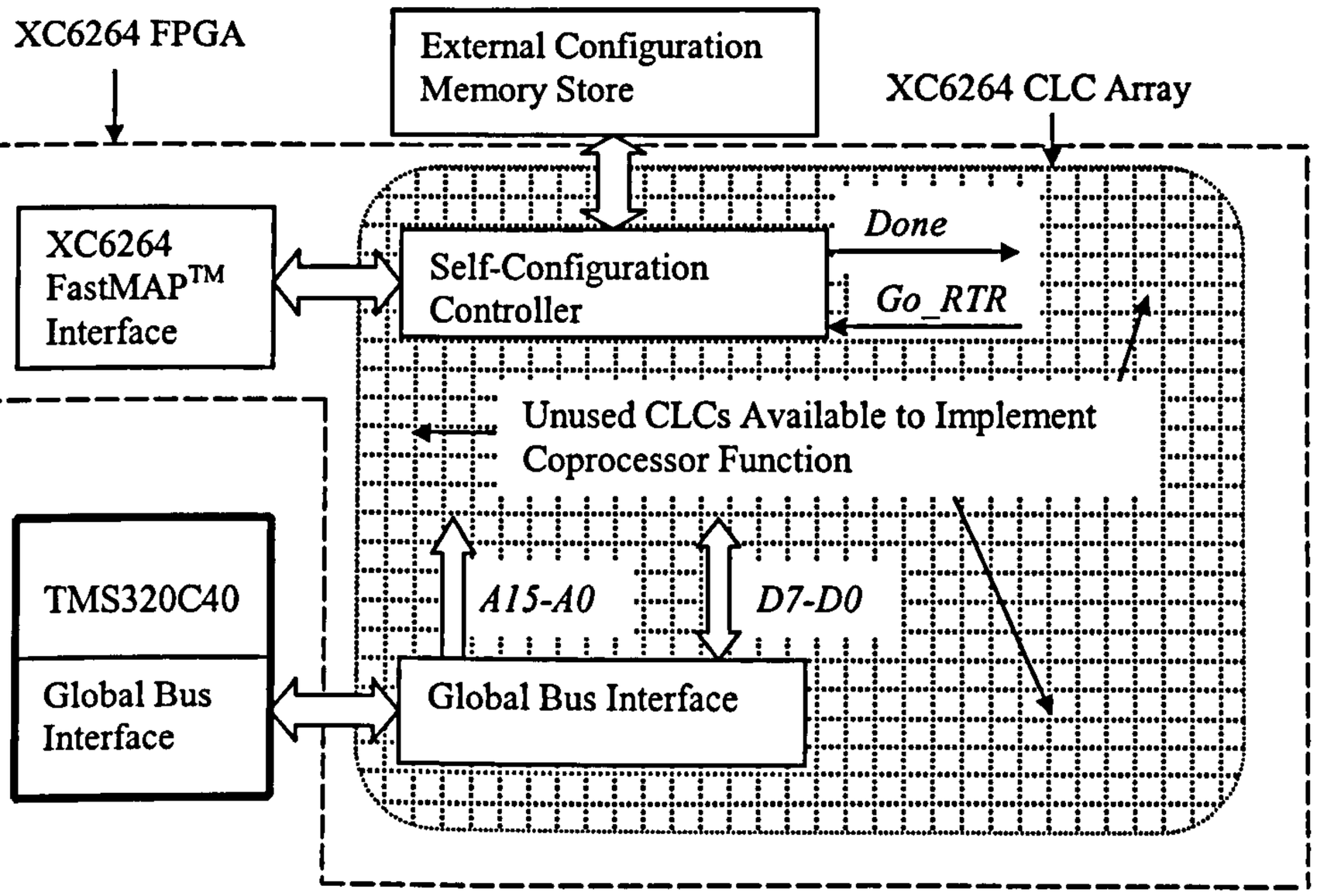


Figure 6.1 XC6200 based Dynamic Coprocessor Topology

6.1.1 TMS320C40 Coprocessor Management

To simplify design of the XC6264 Global buses interface, C40 control signal *GSTRB1* was required to change state for each coprocessor (XC6264) access; *GSTRB1* indicated that an address within a particular C40 memory page had been selected. When accessing common memory page addresses, mechanisms within the C40s DMA forced *GSTRB1* to a constant state. Prior to addressing the XC6264, a dummy address operation had to occur to ensure that a different memory page would be activated (*GSTRB1* toggled).

Run-time management of the coprocessor was performed through software executed upon the C40, written in a variant of the C language. Examples of program code used to access the XC6264 coprocessor are shown in Program 6.1. XC6264 hardware timings were also performed through utilising the C40s internal watchdog timers through software macros.

Program Code (C40 C)	**Comment/Action**
#include <stdlib.h> #include <stdio.h>	C Library Declarations
int main(int argc, char *argv[]) {	Start of Program
volatile long *bfield_pointer_dum = (volatile long*) 0x8000000f;	Define dummy address

volatile long *bfield_pointer_d0 = (volatile long*) 0x8ffffff0
volatile long *bfield_pointer_d1 = (volatile long*) 0x8ffffff1;
volatile long *bfield_pointer_d2 = (volatile long*) 0x8ffffff2;
volatile long *bfield_pointer_d3 = (volatile long*) 0x8ffffff3;
: : : : : :
volatile long *bfield_pointer_df = (volatile long*) 0x8fffffff;
	Define XC6200 Address *A0*
	Define XC6200 Address *A1*
	Define XC6200 Address *A2*
	Define XC6200 Address *A3*
	
	Define XC6200 Address *A15*

volatile long *glob_int_con = (volatile long*) 0x00100000;	Define *GMICR* Address
volatile long glob_int, data;	Define Program Variables

*bfield_pointer_dum = 0x00000007;
 	
*bfield_pointer_d2 = 0x23;
	Dummy Memory Access
	XC6200 Write Cycle
	23_{16} Written to Address *A2*

*bfield_pointer_dum = 0x00000000;
 	
data = *bfield_pointer_d8;
	Dummy Memory Access
	XC6200 Read Cycle
	Data Read From Address *A8*

return(0); }	End of Program

Program-6.1 C40 XC6200 Coprocessor Addressing

Within Program-6.1 pointers to XC6264 addresses *A15-A0* mapped within the C40s Global bus address space were listed first, and defined as *bfield_pointer_d0 to *bfield_pointer_df. These pointers corresponded to C40 Global bus address locations $8ffffff0_{16}$ to $8fffffff_{16}$.

Before data transfer occurred between the XC6264 and C40, XC6264 address space within the C40 Global bus memory map had to be configured. This was achieved through configuring the *Global Memory Interface Control Register* (GMICR). The value written was dependent upon the Global bus interface configuration and determined from the C40s data book [65]. The XC6264 coprocessor could then be addressed, with each access routine including a preliminary dummy memory accesses to ensure signal *GSTRB1* toggled during the actual coprocessor read/write operation.

6.1.2 XC6264 Dynamic Configuration

For dynamic RTR hardware operation to occur, the coprocessor (XC6264) had to be reconfigured during run-time. This was achieved by instigating self-configuration through the XC6264 address space. Once this occurred, the self-configuration controller then updated the coprocessors function using RTR independent of C40/host computer operation. Configuration data required for RTR was stored within and accessed from a local XC6264 coupled configuration memory (*Section-3.4.3)*.

Within the XC6264 self-configuration control unit, signal *Go_RTR* (*Figure 6.1*) initiated self-configuration, whilst *Done* indicated completion. The request, selection and generation of the next active configuration could be determined within the coprocessor function or by the C40 via *A15-A0*.

6.2 XC6264 BinDCT Construction

To investigate dynamic BinDCT operation, the forward and reverse transforms of configurations BinDCT-C1 and BinDCT-C9 were implemented (using VHDL) within the user-function area of the XC6200DS dynamic coprocessor configuration (*Figure*

6.1). This task required developing hardware implementations of each BinDCT configuration (*Section–6.2)* integrating the resultant processor architectures within the skeleton coprocessor (*Section–6.3)*, and incorporating RTR within design (*Section–6.4)* and application (*Section–6.5)*.

In total four BinDCT configurations were constructed. These were configurations FBinDCT-C1, FBinDCT-C9, RBinDCT-C1 and RBinDCT-C9 (F and R denote forward and reverse transforms respectively). The C40 interface and self-configuration control mechanism were developed prior to construction of XC6264 BinDCT hardware.

The four XC6264 BinDCT coprocessors developed functioned as eight concurrent twos-complement binary serial processing pipelines. System operands consisted of 20-bit fixed-point data. Using this scheme decimal numbers in the range of +-0.031225 to 16383.98765 could be represented. The operation, control and operand transfer within these pipelines was governed by the C40 through the XC6264s Global interface address space.

Each transform structure was divided into four operational stages. Using these partitions BinDCT transforms were developed in a modular fashion, allowing efficient replication of common processing elements. Figures 6.2 and 6.3, illustrated the simplified structure of C9 configurations compared to C1 (shown in *Figures 5.6* and *5.7*). Within C9 all lifting ladder coefficients were set to zero, resulting in lifting-ladder computations being reduced to simple addition or subtraction operations.

BinDCT coefficient scaling parameters were not included within the resultant XC6264 designs. It was envisaged to incorporate these parameters using CORDIC algorithms [3]. Through the development of XC6264 based CORDIC hardware, it was concluded that because of XC6264 signal routing limitations, CORDIC hardware was not suitable for configuration within large XC6264 designs. This was because CORDIC implementations required irregular XC6264 footprints, large volumes of CLCs, and extensive local and chip-wide routing resources.

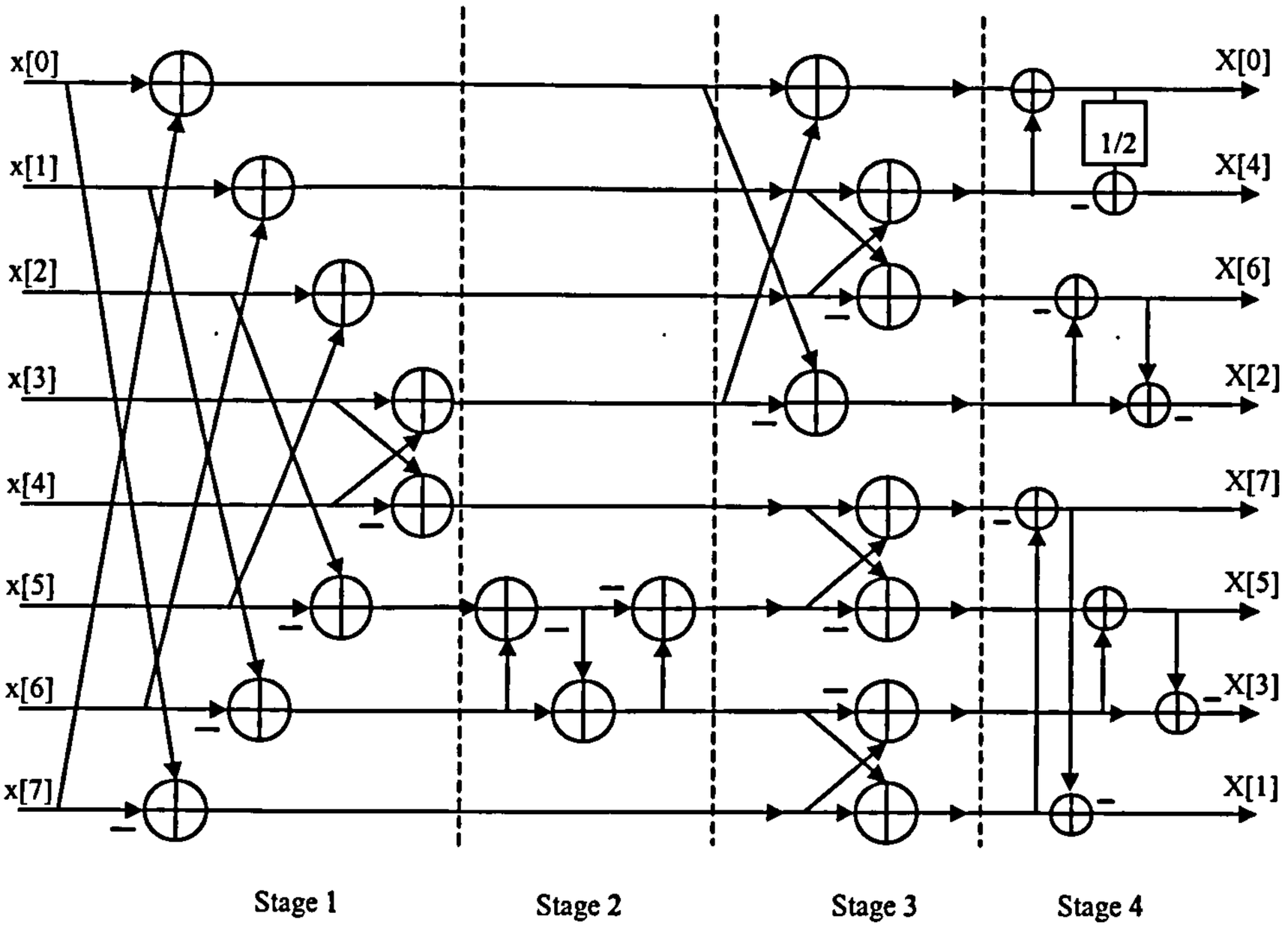


Figure 6.2 BinDCT C9 Forward Transform Flow Diagram

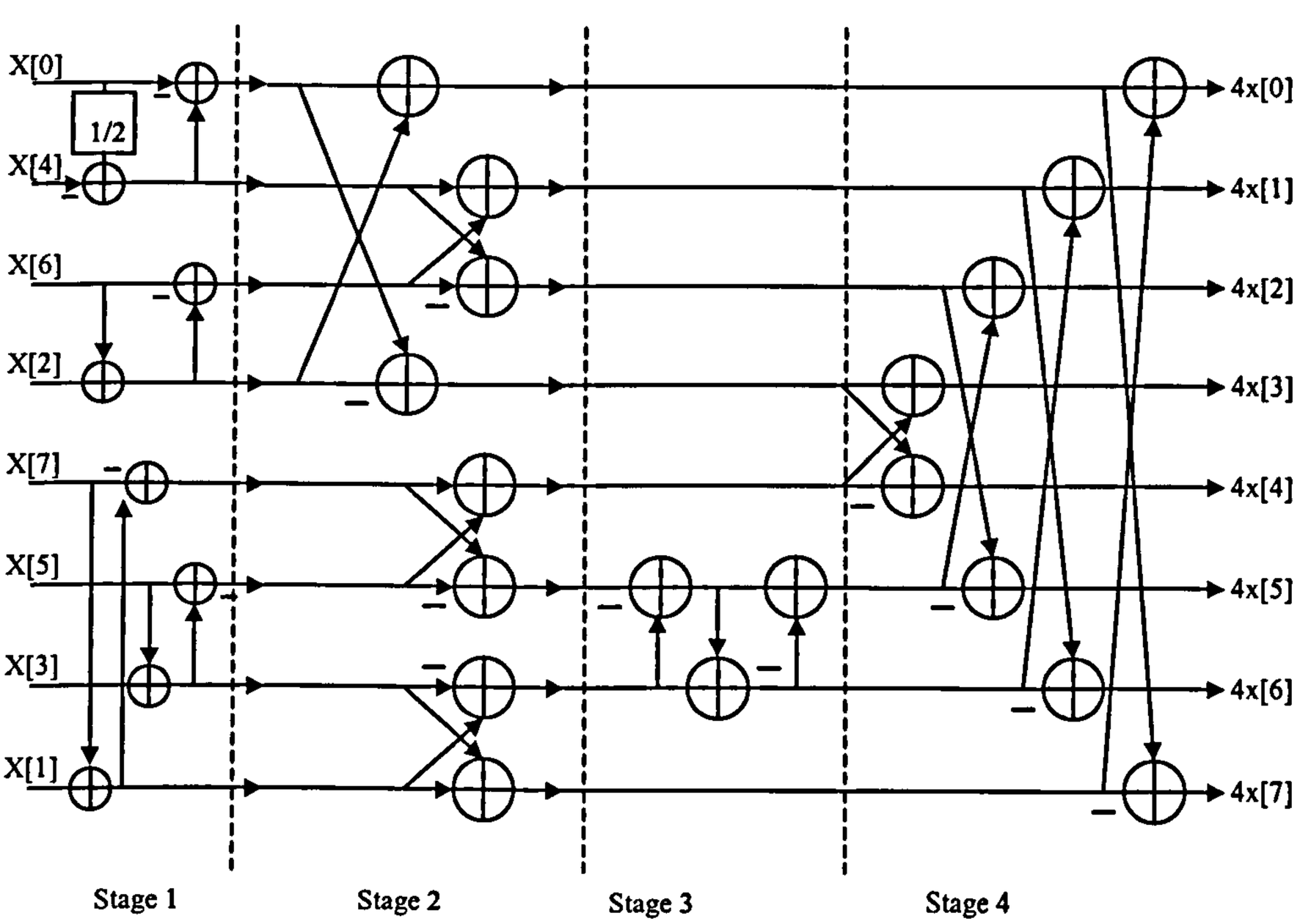


Figure 6.3 BinDCT C9 Reverse Transform Flow Diagram

Figures 5.6, 5.7, 6.2 and 6.3 illustrated how the construction of each BinDCT configuration could be divided into four sections. It was therefore logical to develop and implement each transform configuration in four stages. FBinDCT-C1 was the first configuration developed. Each stage within this transform was constructed and verified operational before commencing the next. This process is described in *Section-6.2.1*.

6.2.1 FBinDCT-C1: Stage-One

FBinDCT-C1 stage-one appeared similar to a general butterfly structure, and was constructed using twos-complement serial binary addition and subtraction units. Initially BinDCT hardware was constructed for maximum throughput using bit-slice designs and parallel data paths. After placement and routing it became apparent that using these methods BinDCT transform hardware would not fit within the XC6264 FPGA. Instead, serial based hardware implementations requiring less XC6264 CLCs and routing resources, but with reduced operand throughput were developed. Conversion of bit-slice to serial designs is illustrated in Figure 6.4.

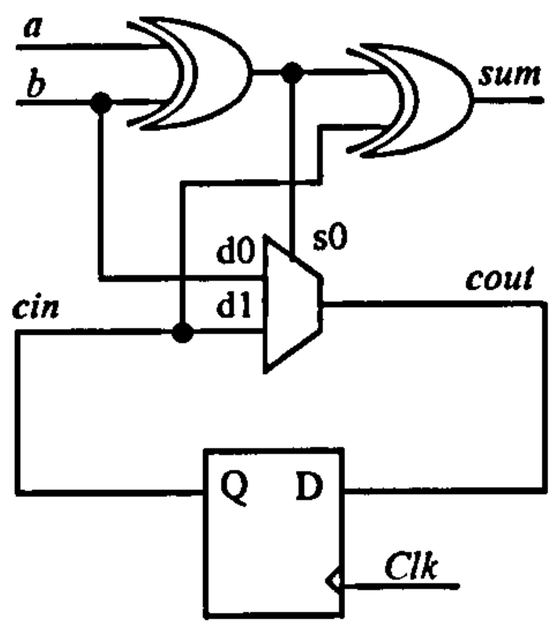


Figure 6.4 XC6264 Serial Adder Bit

The serial adder devised consisted of one adder-bit and a register used to store the *cout* output generated by the previous bits calculation. This value was then used as the present *cin* input. Serial subtraction units were developed using similar methods.

Within the operation of FBinDCT-C1 stage-one individual serial butterfly operations were processed concurrently. Within the resultant coprocessor designs, to govern

system operation, pipeline control signals would be generated by the C40 via the XC6264 coprocessor interface. However during this stage of development, control signal generation processes was performed using the FastMAP™ interface. Pipeline control signal generation is described in *Section-6.3.*

FBinDCT-C1 stage-one construction required 52 CLCs, with a routing footprint of 3 CLCs wide by 20 CLCs high. The design could operate at a maximum frequency of 102.81 MHz, having a pipeline cycle length of 194.5nsec (20-bit data).

Within the pipeline architecture, this design required one pipeline cycle to compute. When coupled to the other BinDCT stages, pipeline delays of individual stages overlapped. The resultant design of the stage-one is shown in Figure 6.5.

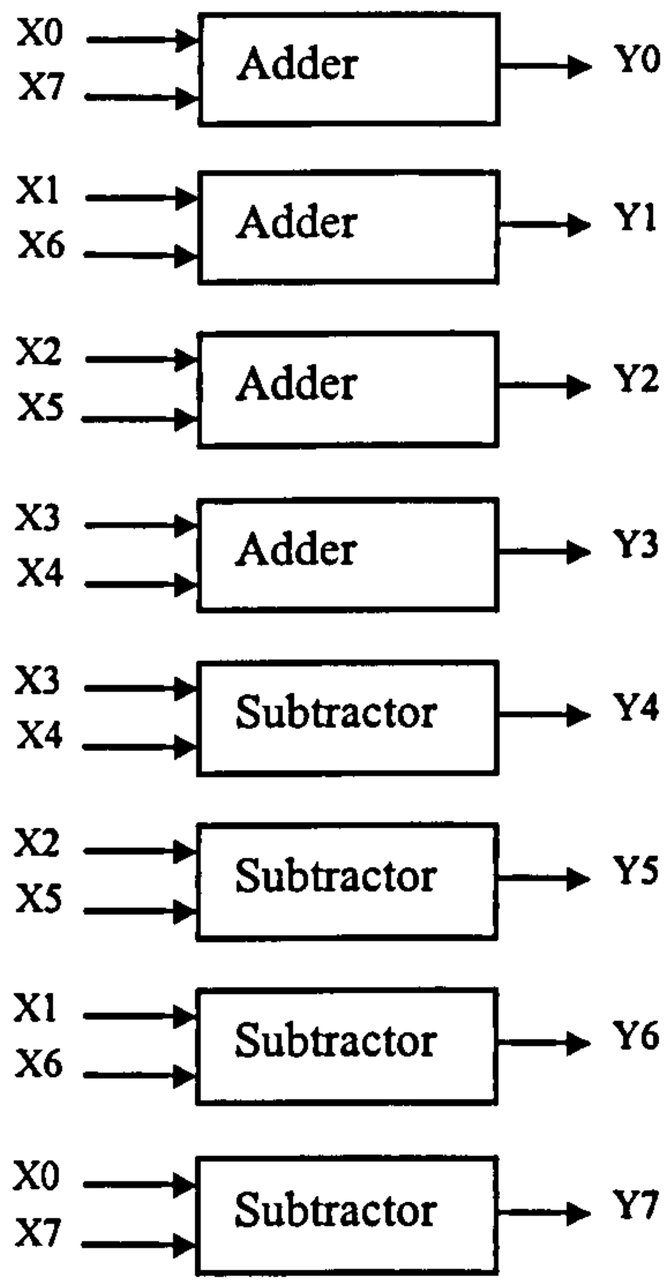


Figure 6.5 FBinDCT-C1 Stage One Architecture

6.2.2 FBinDCT-C1: Stage-Two

Stage-two of FBinDCT-C1 required construction of dyadic lifting-ladders. To compute this function a serial dyadic shift operation was devised. This consisted of a shift register, adder chain and control logic, as shown in Figure 6.6. This unit was common for each dyadic value required, with the actual value configured dependant upon inputs *Ena_0* to *Ena_4*.

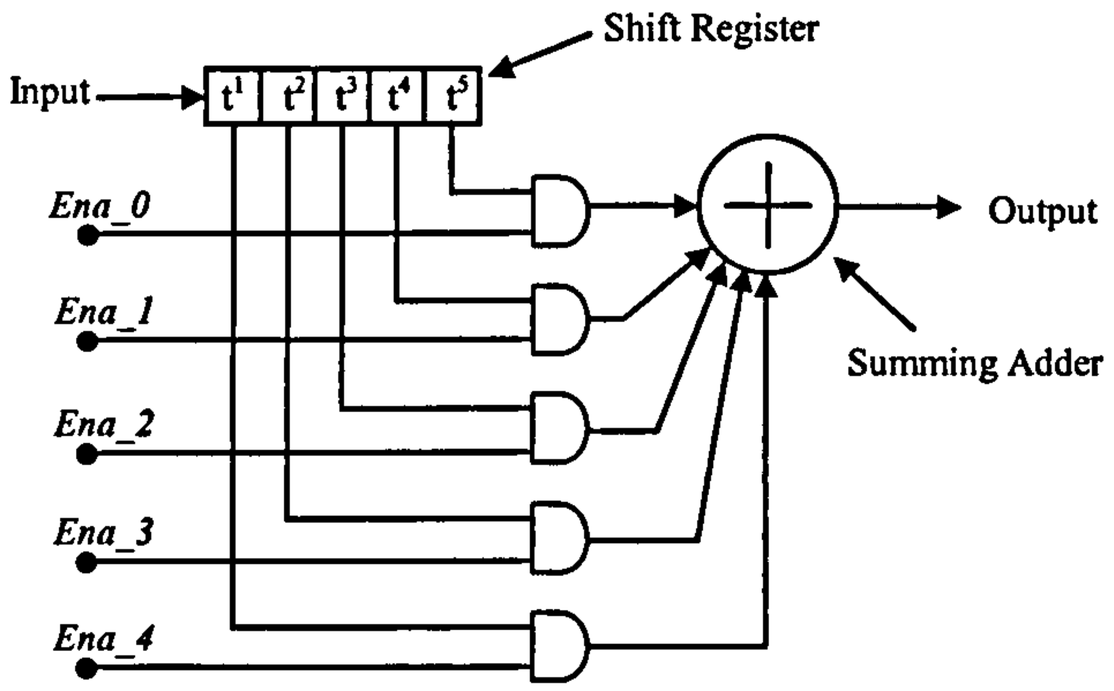


Figure 6.6 Serial Dyadic Shift Unit

To illustrate the operation of the serial dyadic shift unit, consider the following fixed-point binary number $0000\ 0001.0000_2$ scaled using dyadic values 0.5 to 0.34375.

Dyadic Value	Scaled Output Value	Shift Register Delay
0.5	$0000\ 0000.10000_2$	(t^{+1})
0.25	$0000\ 0000.01000_2$	(t^{+2})
0.125	$0000\ 0000.00100_2$	(t^{+3})
0.1875	$0000\ 0000.00110_2$	$(t^{+3}) + (t^{+4})$
0.34375	$0000\ 0000.01011_2$	$(t^{+2}) + (t^{+4}) + (t^{+5})$

Table 6.1 Dyadic Number Representations

A binary number can be scaled by dyadic values of 0.5, 0.25 and 0.125 by shifting the input by one, two, and three places respectively to the right. With respect to Figure 6.7, these operations relate to using output taps (t^{+1}, t^{+2}, t^{+3}) of the shift register. For a dyadic

value such as 0.1875, multiple shift register output taps (t^{+3}, t^{+4}) are summed together within the adder.

When using twos-complement data the sign-bit (MSB) must be included within each dyadic shift operation. Since the design in Figure 6.6 operated using serial operands each dyadic shift took two pipeline cycles to compute. Lifting-structures P4, P5 and U4 were constructed through coupling dyadic-shift units to addition (U4) or subtraction units (P4, P5), as dictated by the flow diagram (*Figure 5.4*).

Within FBinDCT-C1 stage-two lifting structures P4, U4 and P5 connected in series, allowing pipeline cycles of each lifting-structure operation to overlap. This feature reduced the delay of stage-two to 3 pipeline cycles. Consequently, stage-two operands unaffected by dyadic shift operations had to be delayed by 3 pipeline cycles to ensure output data coherency. The resultant structure of stage-two is shown in Figure 6.7.

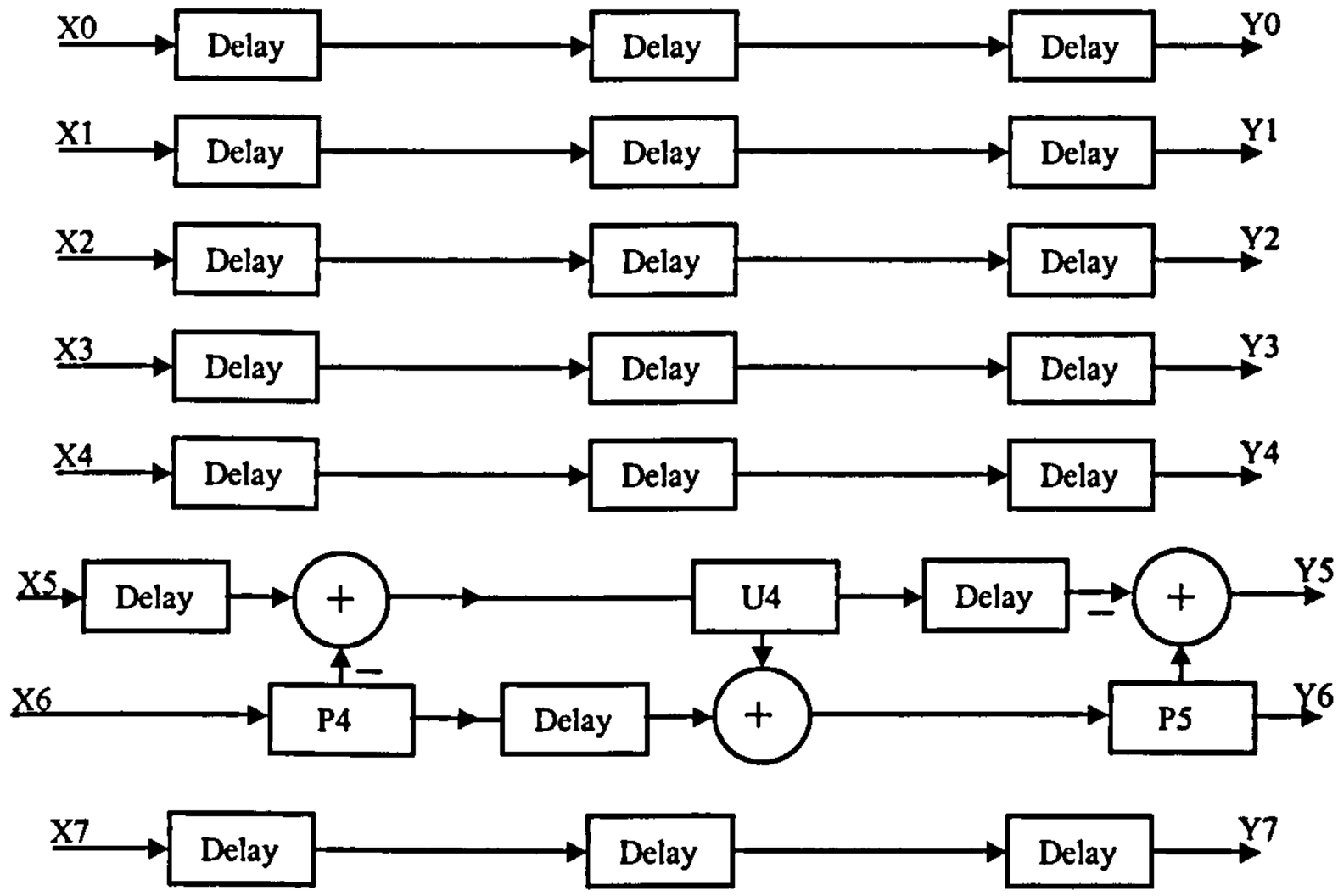


Figure 6.7 FBinDCT-C1 Stage Two Architecture

The implementation of this design required 648 CLCs (116 per lifting-structure, 20 per pipeline delay) and a routing footprint of 18 CLCs wide by 56 CLCs high. The maximum clock frequency of the design was calculated to be 8.38MHz. Using 20-bit

data the initial output of stage-two required three pipeline cycles to compute (7.16μsec). Once the pipeline was full, a result would be generated every cycle (2.39μsec).

6.2.3 BinDCT-C1: Stage-Three

The implementation of FBinDCT-C1 stage-three required addition and subtraction butterflies only, with no lifting-structures required. The fabrication of this stage was similar to stage-one (*Section-6.2.1*) and constructed using replicated stage-one component as shown in Figure 6.8. The XC6264 implementation properties obtained for this stage were therefore identical to those obtained for stage-one.

6.2.4 BinDCT-C1: Stage-Four

Stage-four comprised four pairs of concurrent operating lifting-structures. Each lifting structure was constructed using techniques described in *Section-6.2.2*. The flow diagram of the resultant design was identical to the original FBinDCT-C1 flow diagram and has therefore not been included. Through overlapping serial dyadic lifting steps, stage-four required two pipeline cycles to compute when empty. Once full, a result could be generated every pipeline cycle.

Stage-four was implemented within 928 CLCs, with a routing footprint of 48 CLCs high by 38 CLCs wide. The maximum operating frequency of stage-four determined was 8.8MHz. The initial output of stage-four took 4.54μsec to compute (two pipeline cycles) after this results were generated every 2.27μsec (one pipeline cycle).

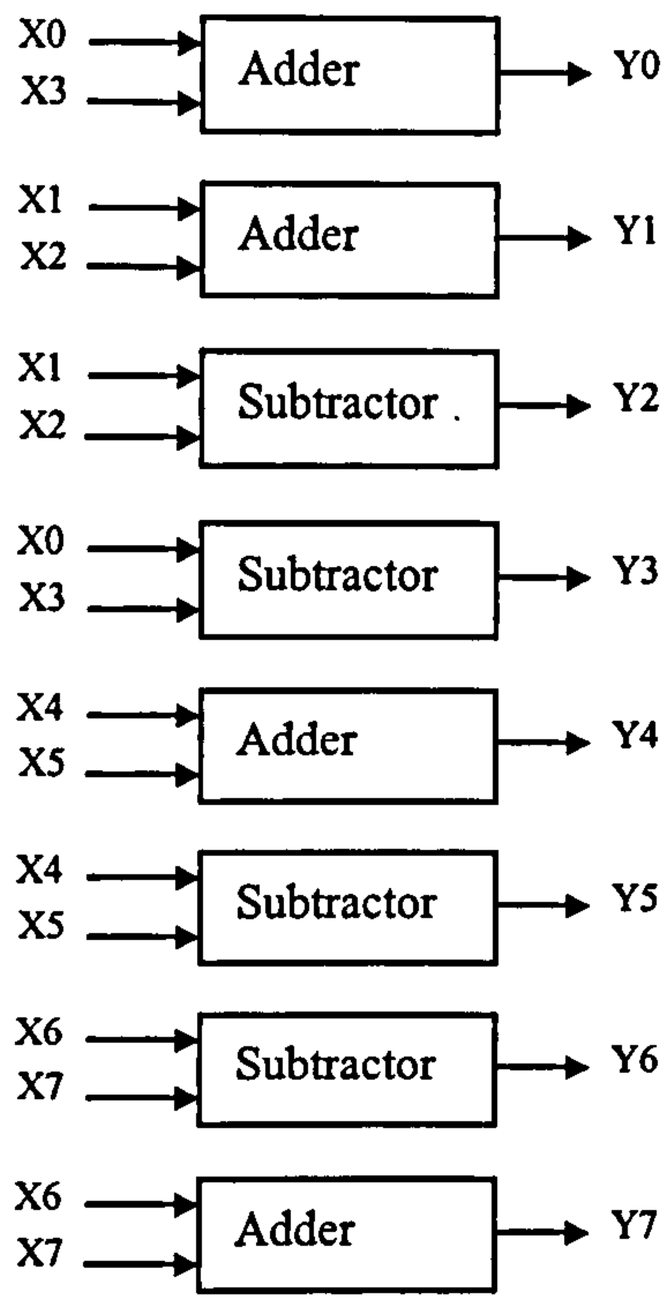


Figure 6.8 FBinDCT-C1 Stage Three Architecture

6.2.5 XC6264 BinDCT Hardware Characteristics

To verify the operation of each stage, operands and system control signals were applied through the FastMAP™ interface. Additional hardware configured within each stage converted operands from parallel to serial format and performed I/O transfer. Results generated were then observed using XC6200ADS tools.

The three remaining BinDCT transforms were each constructed and evaluated using the four-stage development concept presented. Through reuse and adaptation of hardware modules developed previously, development overheads were reduced and differences between transforms kept to a minimal, aiding RTR configuration data generation.

Within configurations FBinDCT-C9 and RBinDCT-C9, dyadic lifting structure coefficients were set to zero. This reduced the hardware implementation of the

equivalent FBinDCT-C1 and RBinDCT-C1 stages to only addition, subtraction, and units. This reduced the number of pipeline cycles required to generate each transform.

The XC6264 hardware characteristics of each BinDCT transform neglecting FastMAP™ interface overheads, coprocessor interface, and self-configuration controller delays are detailed in Tables 6.2-6.4. Results highlighted the reduction in hardware overheads and performance benefits gained through using BinDCT configuration C9 instead of C1.

	Stage 1	Stage 2	Stage 3	Stage 4	Max. Frequency
FBinDCT-C1	102.81MHz	8.37MHz	102.81MHz	8.8MHz	8.37MHz
FBinDCT-C9	102.81MHz	20.08MHz	102.81MHz	8.57MHz	8.57MHz
RBinDCT-C1	8.8MHz	102.81MHz	8.43MHz	102.81MHz	8.43MHz
RBinDCT-C9	9.12MHz	102.81MHz	19.82MHz	102.81MHz	9.12MHz

Table 6.2 BinDCT Hardware Characteristics: Operating Frequency

Table 6.2 indicates the maximum clock frequency obtained for each stage within the transforms and number of XC6264 CLCs required to implement each design. Identical results were obtained for several different stages since common hardware components were re-used for different transforms.

	Stage 1	Stage 2	Stage 3	Stage 4	Total CLCs Used
FBinDCT-C1	52	648	52	928	1680
FBinDCT-C9	52	14	52	379	497
RBinDCT-C1	928	52	648	52	1680
RBinDCT-C9	379	52	14	52	497

Table 6.3 BinDCT Hardware Characteristics: XC6264 CLCs Required

The maximum operand throughput (pipeline full) of each transform is listed in Table 6.4. These values were determined using the maximum clock frequency of the design, the pipeline cycle length (20-bit) and the number of pipeline cycles required to compute the result. The results generated include operand throughputs calculated when the pipeline was initially empty. Once the pipeline was full, a result was generated every pipeline cycle. If adjacent pixel tiles were processed by the same BinDCT configuration

(C1 or C9), the pipeline would remain full. Only when BinDCT configuration was switched did the pipeline have to be refilled.

	Number of Pipeline Cycles	Pipeline Cycle Duration	BinDCT Throughput (At Max Frequency)	
			Pipeline Full	Pipeline Empty
FBinDCT-C1	6	2.389μsec	418.58kBinDCT ops/sec	69.76kBinDCT ops/sec
FBinDCT-C9	3	2.333μsec	428.63kBinDCT ops/sec	142.88kBinDCT ops/sec
RBinDCT-C1	6	2.372μsec	421.56kBinDCT ops/sec	70.26kBinDCT ops/sec
RBinDCT-C9	3	2.192μsec	456.20kBinDCT ops/sec	152.07kBinDCT ops/sec

Table 6.4 BinDCT Performance Characteristics

Comparisons between XC6264 BinDCT hardware operation and software BinDCT simulation and true DCT operation are discussed within *Section 6.4*.

6.3 BinDCT Static Coprocessor Integration

Throughout the development phase of BinDCT processor hardware, operand transfers and pipeline control signal generation were conducted via the FastMAP™ interface. To integrate BinDCT configurations within C40 XC6264 coprocessors, operand transfer and pipeline control had to be performed by the C40. This occurred by accessing XC6264 coprocessor address space, located within the C40 Global bus memory map.

Operand transfer and initialisation of BinDCT pipeline cycles was governed by the C40 through accessing XC6264 memory mapped addresses. The interface itself consisted of an address decoder, control logic, data paths, and operand registers as shown in Figure 6.9. To perform XC6264 BinDCT operations, components within the pipeline were first reset to their initial conditions using signal *C40_clr* (*A7*). Operands were then applied to the input of stage-one by writing data to XC6264 address *A14* (*Write_input*). The pipeline was then clocked using signal *C40_clk* (*A8*), and the output of stage-four was then read via *A15* (*Read_output*). This cycle of writing input operands, generating *C40_clk*, and reading the output of stage-four was performed 20 times (20-bit system operand, using twos complement fixed point representation) for each block of 8 input coefficients.

To activate the dyadic scaling units, signal *Ena_shift* (A9) was activated after '*m*' *C40_clks*; *Where m = number of operand bits – (system accuracy + 2).* Since system operands were 20-bit, having 5-bit accuracy (*S*000 0000 0000 0000.00000$_2$, where '*S*' was the sign bit), 'm' was calculated to be 13 *(20-(5+2))*. Upon the completion of the pipeline cycle signal *Ena_shift* was reset, and the contents of serial adder/subtractor carry/borrow registers cleared and re-initialised via signal *Next_pipe* (A10). The global *C40_clr* signal could not be used for this purpose otherwise intermediate pipeline operands would be erased.

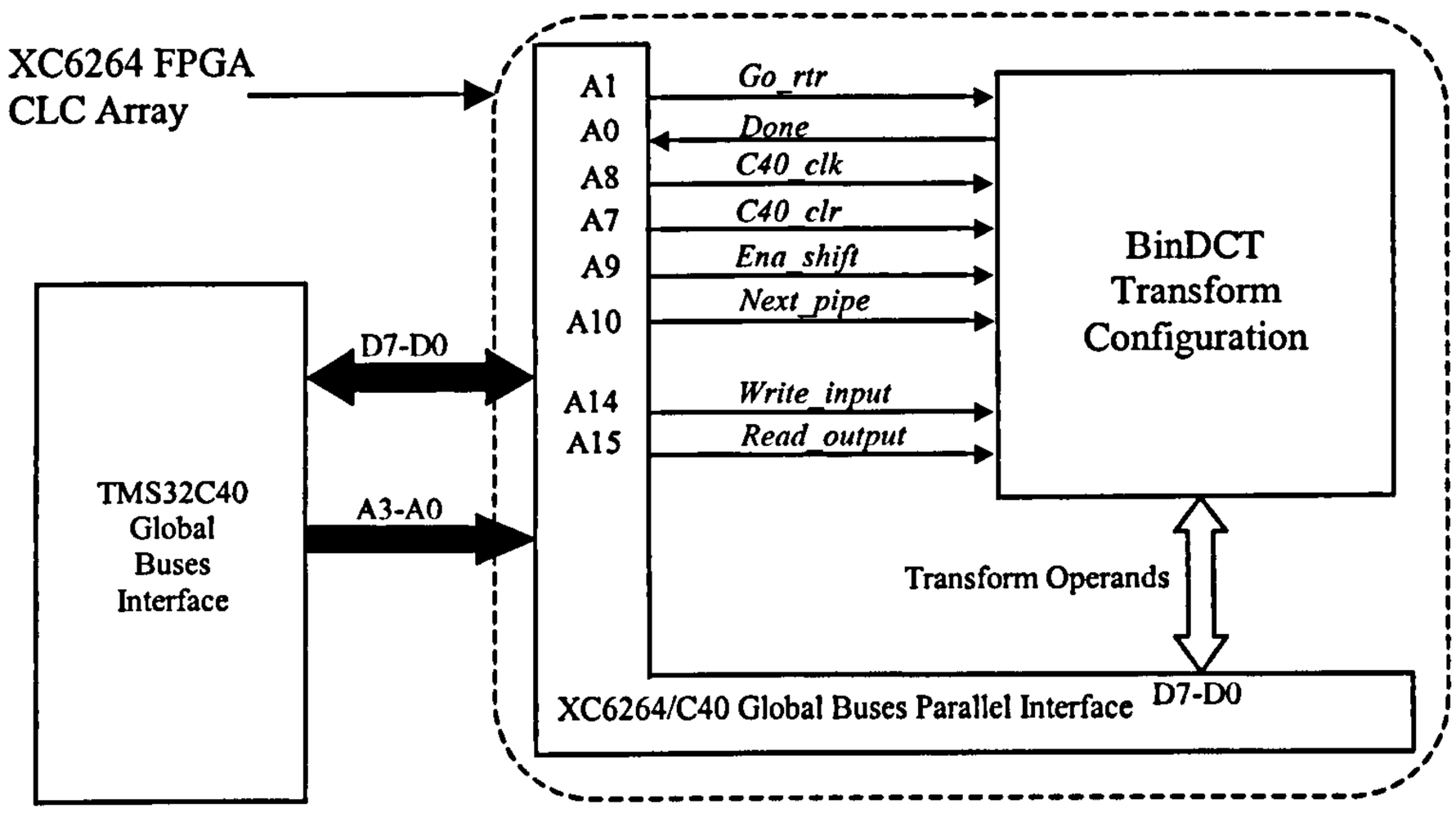


Figure 6.9 XC6264 BinDCT Coprocessor Integration

Blocks of eight parallel operands were computed by XC6264 BinDCT hardware using eight concurrent serial pipelines. Each of the eight individual operands were transferred to/from the C40 in a serial fashion bit by bit, accounting for a byte value upon the Global buses data-bus. All eight operands could therefore be transmitted in a concurrent serial fashion using this method. The byte values transferred to/from the C40 therefore related to the distributed serial implementation of eight operands. To transfer eight operands with value range +-0.03125 to 16383.98765, 20 C40 byte transfers were required.

This concept is illustrated in Table 6.5, using a block of eight input bytes (*original input*). The appropriate weighted value required for each input byte was determined by converting the C40 byte values into their XC6264 weighted value equivalents. The conversion of input coefficients *(0.5,2.03125,0,0,4,5,2.25,0.125*) to XC6264 weighted values was performed by placing a '1' in the weighted values columns, constituting the original input value (e.g. 2.03125 = 2 + 0.03125 => 010000001). To generate the required byte value to be transferred by the C40, the bit content of each value column was converted to hexadecimal format. In Table 6.5 only 8 weighted values are used, for true BinDCT operation 19 weighted values would be used (plus sign bit), generating 20 C40 bytes.

	XC6264 Bit Weighted Value							
Original Input	*4*	*2*	*1*	*0.5*	*0.25*	*0.125*	*0.0625*	*0.03125*
0.5	0	0	0	1	0	0	0	0
2.03125	0	1	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
4	1	0	0	0	0	0	0	0
5	1	0	1	0	0	0	0	0
2.25	0	1	0	0	1	0	0	0
0.125	0	0	0	0	0	1	0	0
Required C40 Output Value	**0x30**	**0x42**	**0x20**	**0x01**	**0x40**	**0x80**	**0x0**	**0x02**

Block of 8 Coefficients (0-7)

Block of 8 Hexadecimal C40 Bytes (0-7)

Table 6.5 C40/XC6264 Weighted Operand Conversion

This conversion process was initially conducted within XC6264 BinDCT hardware. However, it was later removed and incorporated within C40 operating software to aid BinDCT application development. Combined within this operation were routines that converted from C40 floating-point data to XC6264 two complements fixed point operand notation and vice-versa. C40 software also provided BinDCT coefficient scaling when comparing the operation of BinDCT transforms against true DCT operation (*Section-6.5*).

The maximum operating frequencies calculated for each transform are shown in Table 6.6. In comparison to Table 6.2, the maximum frequencies of the transforms were reduced. This feature was predicted since the XC6264 now contained greater logic resources than previously (BinDCT transform, plus coprocessor interface), requiring greater signal routing complexity.

	Maximum Frequency
Forward BinDCT C1	6.035MHz
Forward BinDCT C9	5.518MHz
Reverse BinDCT C1	5.827MHz
Reverse BinDCT C9	4.759MHz

Table 6.6 Maximum BinDCT Coprocessor Operating Frequencies

The maximum operating frequency of the XC6264/C40 coprocessor interface was calculated to be 16.58MHz using XACT6000, enabling a bandwidth of approximately 4.15Mbytes/sec. However, during normal operation this frequency was set to 8.0MHz resulting in a bandwidth of 2.0Mbytes/sec.

BinDCT hardware was clocked using *C40_clk* (generated by *A8*) at a frequency of 1.11MHz. This value related to two C40 memory accesses that addressed and wrote data to *A8* within the XC6264s address space (*C40_clk = 1, C40_clk = 0*). The XC6264 Global bus interface bandwidth limited the maximum frequency of *C40_clk* that could be synthesized. Initially it was intended to use *A8* directly as the clock frequency (generating a frequency of approximately 2MHz), however during construction routing limitations encountered within XC6264s C40 Global buses interface prevented this.

Results presented in Table 6.7 demonstrate the difference between the maximum and normal operational coprocessor characteristics. These results were calculated assuming that BinDCT pipelines were empty (result generated after *n* pipeline cycles; Where *n* is the pipeline cycle length). Full pipeline throughput was calculated by multiplying the pipeline *operand throughput* by the number of pipeline cycles (*cycle length*) shown in Table 6.7. Using this procedure, operand throughputs (@1.11MHz) of 55.5kBinDCT

one-dimensional transform operations per second (ops/sec) were obtained for each BinDCT configuration for full pipeline operation.

	Maximum Frequency of Design		Normal (1.11MHz) Operating Frequency	
	Cycle Length	*Operand Throughput*	*Cycle Length*	*Operand Throughput*
FBinDCT-C1	6	50.29kBinDCT ops/sec	6	9.25kBinDCT ops/sec
FBinDCT-C9	3	91.52kBinDCT ops/sec	3	18.5kBinDCT ops/sec
RBinDCT-C1	6	48.56kBinDCT ops/sec	6	9.25kBinDCT ops/sec
RBinDCT-C9	3	79.32kBinDCT ops/sec	3	18.5kBinDCT ops/sec

Table 6.7 Maximum/Normal BinDCT Coprocessor Throughputs

6.4 Dynamic Coprocessor Development

To enable dynamic coprocessor operation, a self-configuration control mechanism was inserted within the static coprocessors developed in *Section-6.3*. The resultant configurations were then temporally and spatially examined using XC6200ADS tools.

The integration of the self-configuration controller occurred in two stages. The first stage developed the control mechanism between the C40 parallel interface and the self-configuration control unit. Once this had been proved functional, the BinDCT transforms were then implemented using this mechanism. These stages are described in *Sections-6.4.1* and *6.4.2* respectively.

6.4.1 Dynamic Coprocessor Control Mechanism

The self-configuration controller interface consisted of three signals called *Go_rtr*, *Done* and *Dsel*. *Go_rtr* initiated the reconfiguration process, whilst *Done* indicated when the process had finished. *Dsel* (16-bit bus) determined the next active configuration downloaded from the configuration memory store.

For the XC6200DS to function as a dynamic coprocessor, self-configuration had to be instigated either internally within the coprocessor function itself, or by the master

processor (C40). The XC6264s gate capacity remaining available after BinDCT and coprocessor interface integration determined that the decision to conduct RTR would occur within the C40. (*Appendix-IV-4* describes the internal operation of the self-configuration control mechanism).

To develop this concept, a simple RTR application was developed. The XC6264 was configured with two temporally partitioned circuits generating two different clock frequencies. RTR instigation was conducted through the C40 Global bus interface as shown in Figure 6.10.

Signal *Go_rtr* was generated using XC6264 interface address *A1*. When a positive signal transition occurred on *Go_rtr*, RTR commenced. The resulting configuration delay was dependant upon the level of differences between successive configurations. To determine when this operation was complete, initially the status of *Done* was monitored by the C40 using XC6264 address *A0*. Through experimentation it was discovered that completion of RTR could also be determined by the C40, through halting XC6264 Global interface control state machine operation using signal *Done*. This concept proved more reliable than the previous configuration since *A0*s value could be inadvertently updated during RTR.

The bit set (logic one) within *Dsel* (16-bit) determined the next hardware configuration activated. *Dsel* bit values could be set via the C40 interface, but within this example were hardwired within each XC6264 design. The value of *Dsel* is shown in Figure 6.10 for each active configuration. Effectively the address pointer to the next configuration is stored within the current active configuration. This operation can be considered similar to linked-list structures within programming languages.

To swap between each configuration required 224 bytes of configuration data (56 address/data pairs) to be downloaded. Using a self-configuration controller clock frequency of 8MHz, this took approximately 107.8μsec to complete (measured externally, using the custom designed timer).

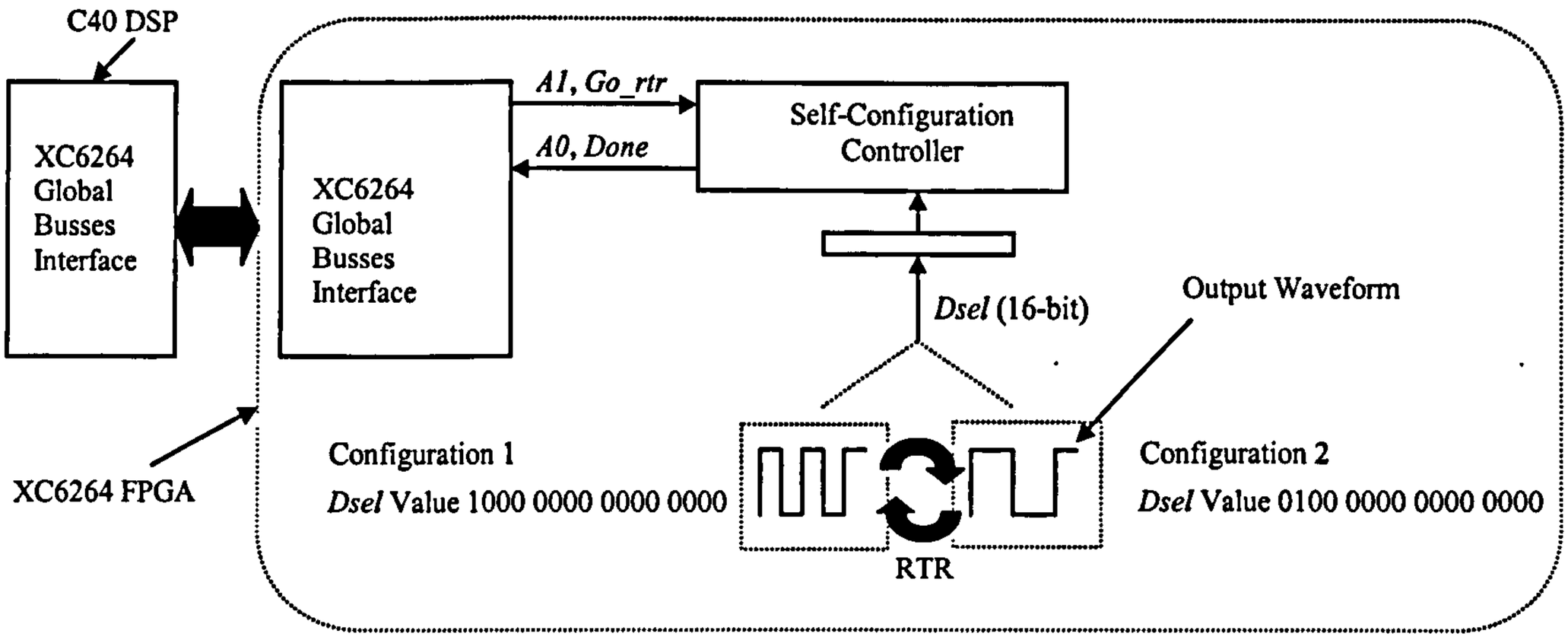


Figure 6.10 C40 Dynamic Coprocessor Operation

6.4.2 BinDCT Integration

To construct the dynamic coprocessor function each BinDCT configuration was inserted within the outline dynamic coprocessor mechanism developed in *Section-6.4.1* (*Figure 6.10*). In similar fashion the differences between each coprocessor configuration were determined using XC6200ADS tools.

Temporal and spatial partitioning was performed between each transform configuration. During this procedure the structures of each configuration pair were analysed and differences between them determined. To minimise differences between successive configurations, common components in each transform were located at identical CLC positions within the XC6264 FPGA. The volumes of XC6264 CLC array address locations required updating (*XC6264 CLCs)* and actual configuration data downloaded (*data bytes*), with the resultant minimum configuration delay for each transform update are shown in Table 6.8. The self-configuration controllers clock frequency was set to 8MHz, with configuration delays recorded using the custom designed external timer.

Configuration Number		Configuration Data Volume		Configuration Delay
Config. (N)	*Config. (N+1)*	*XC6264 CLCs*	*Data bytes*	*External Timer*
FBinDCT-C1	FBinDCT-C9	6413	25652	12.01msec
FBinDCT-C1	RBinDCT-C1	10781	43124	20.28msec
FBinDCT-C9	RBinDCT-C9	4373	17492	8.24msec
RBinDCT-C1	RBinDCT-C9	6746	26984	12.66msec

Table 6.8 Dynamic BinDCT Coprocessor Configuration Parameters

To evaluate dynamic switching capabilities, the clock frequencies of the C40 interface and self-configuration control mechanism were set to a clock frequency of 8.0MHz. The BinDCT pipeline was clocked at a frequency of approximately 1.11MHz. This signal was generated within the XC6264s C40 Global busses interface, through the C40 accessing XC6264 address location *A8 (*located within C40s Global memory map).

Using XACT6000 software the maximum clock frequencies determined for each transform are shown in Table 6.9. Table 6.9 lists the operand throughput (measured in kBinDCT ops/sec) for both maximum and normal operating frequency conditions (@1.11MHz). Operand throughputs were calculated for both full and empty pipeline conditions. Results indicated that the dynamic BinDCT operation had a maximum combined coprocessor operating frequency of 4.17MHz. However, during normal operation, all BinDCT configurations were clocked at 1.11MHz using signal *C40_clk*, as described in *Section-6.3*.

	Maximum Frequency			BinDCT @1.11MHz	
	Operating	Pipeline Condition		Pipeline Condition	
Configuration	Frequency	*Empty*	*Full*	*Empty*	*Full*
FBinDCT-C1	5.16MHz	43.00	257.79	9.25	55.5
FBinDCT-C9	5.3MHz	88.37	265.11	18.50	55.5
RBinDCT-C1	4.5MHz	37.54	225.27	9.25	55.5
RBinDCT-C9	4.17MHz	69.48	208.46	18.50	55.5

Table 6.9 BinDCT Dynamic Coprocessor Operating Frequency

6.5 XC6264 BinDCT Transform Characteristics

To evaluate the operational characteristics of XC6264 BinDCT hardware, transform operations were performed using input sequences defined in *Section-5.3.1*. The output responses recorded were then compared to results generated through software simulations of BinDCT and true DCT operations. This task was conducted for both one and two-dimensional transforms. The results recorded are presented in *Section-6.5.1* and *Section-6.5.2* respectively.

6.5.1 One-Dimensional XC6264 BinDCT Operation

Through applying the input data sequences defined in *Section-5.3.1,* the operational characteristic of XC6264 hardware transforms FBinDCT-C1, FBinDCT-C9, RBinDCT-C1 and RBinDCT-C9 were analysed. Results recorded are shown in Tables 6.10 to 6.14. Similarly to *Chapter-5*, the RMSE generated compared the difference between XC6264 BinDCT hardware and true DCT operations. Furthermore, BinDCT coefficient-scaling factors (*Table 5.3*) omitted from the XC6264 hardware were performed within the C40 DSP primary processor.

n	Input[0 to(N-1)]	FDCT	RDCT	FBinDCT-C1	RBinDCT-C1	FBinDCT-C9	RBinDCT-C9
0	31	404.819	31	404.785	30.984	404.785	30.969
1	63	-206.57	63	-206.96	62.961	-98.391	62.992
2	95	0.191	95	0.203	94.984	0.541	95.000
3	127	-21.453	127	-20.276	126.969	-77.573	126.961
4	159	-0.354	159	-0.397	158.969	-0.397	158.961
5	191	-5.938	191	-7.236	191.000	0.402	190.984
6	224	-0.462	224	-0.462	224.023	-0.447	223.992
7	255	-1.345	255	-0.368	254.984	-62.295	255.015
XC6264 BinDCT/DCT RMSE				0.7228	0.0243	48.2278	0.0240

Table 6.10 XC6264 BinDCT Outputs for Data Sequence-(i) Ramp Function

n	Input[0 to(N-1)]	FDCT	RDCT	FBinDCTC1	RBinDCTC1	FBinDCTC9	RBinDCTC9
0	255	721.249	255	721.249	255.000	721.249	255.000
1	255	0.000	255	0.000	254.992	0.000	254.992
2	255	0.000	255	0.000	254.992	0.000	254.992
3	255	0.000	255	0.000	255.000	0.000	255.000
4	255	0.000	255	0.000	255.000	0.000	255.000
5	255	0.000	255	0.000	254.992	0.000	254.992
6	255	0.000	255	0.000	254.992	0.000	254.992
7	255	0.000	255	0.000	255.000	0.000	255.000
XC6264 BinDCT/DCT RMSE				0.0000	0.0057	0.0000	0.0057

Table 6.11 XC6264 BinDCT Outputs for Data Sequence-(ii) Constant Level

n	Input[0 to(N-1)]	FDCT	RDCT	FBinDCTC1	RBinDCTC1	FBinDCTC9	RBinDCTC9
0	255	540.937	255	540.937	254.977	540.937	254.984
1	85	0.0000	85	0.000	84.984	0.000	85.000
2	170	-32.528	170	-31.625	170.015	-92.003	170.000
3	255	0.0000	255	0.000	255.008	0.000	255.000
4	255	180.312	255	180.312	255.008	180.312	255.000
5	170	0.0000	170	0.000	170.015	0.000	170.000
6	85	78.53	85	78.53	84.984	78.530	85.000
7	255	0.0000	255	0.000	254.977	0.000	254.984
XC6264 BinDCT/DCT RMSE				0.3193	0.0164	21.0276	0.0080

Table 6.12 XC6264 BinDCT Outputs for Data Sequence-(iii) Mexican Hat

n	Input[0 to(N-1)]	FDCT	RDCT	FBinDCTC1	RBinDCTC1	FBinDCTC9	RBinDCTC9
0	255	360.624	255	360.624	255.000	360.624	255.000
1	255	326.772	255	325.887	254.969	259.996	254.992
2	255	0.0000	255	0.000	254.969	0.000	254.992
3	255	-114.747	255	-115.759	254.984	0.000	255.000
4	0	0.000	0	0.000	0.015	0.000	0.000
5	0	76.671	0	78.560	0.015	0.000	-0.008
6	0	0.0000	0	0.000	0.015	0.000	-0.008
7	0	-64.999	0	-65.865	0.000	0.000	0.000
XC6264 BinDCT/DCT RMSE				0.8750	0.0189	58.8740	0.0057

Table 6.13 XC6264 BinDCT Outputs for Data Sequence-(iv) Step Function

n	Input[0 to(N-1)]	FDCT	RDCT	FBinDCTC1	RBinDCTC1	FBinDCTC9	RBinDCTC9
0	0	90.156	0	90.156	0.000	90.156	-0.015
1	0	-24.874	0	-24.374	0.000	-129.998	-0.008
2	0	-117.795	0	-118.707	0.031	0.000	0.008
3	0	70.835	0	71.88	0.008	153.343	0.000
4	255	90.156	255	90.156	254.961	90.156	254.984
5	0	-106.012	0	-106.012	-0.015	-106.012	-0.008
6	0	-48.792	0	-47.854	-0.015	-117.795	0.008
7	0	125.05	0	125.05	0.000	125.05	0.000
XC6264 BinDCT/DCT RMSE				0.6178	0.0194	67.5423	0.0096

Table 6.14 XC6264 BinDCT Outputs for Data Sequence-(v) Spike Function

Table 6.15 compares the software and hardware BinDCT RMSE values calculated with respect to true DCT operation. Results indicated that XC6264 BinDCT hardware generated results introduced greater errors when reconstructing original data compared to BinDCT software simulations (*Section-5.3.1)*. This error was expected since XC6264 BINDCT hardware was constructed using fixed-point binary twos-complement numbers (operand resolution of 0.03125), and BinDCT software simulations performed on a PC (operand resolution $1.2x10^{-38}$ [81]).

Differences in RMSEs calculated for FBinDCT-C9 operation were attributed to the XC6264 designs used for FBinDCT-C9 and RBinDCT-C9. Within FBinDCT-C9s XC6264 design, internal operands were scaled to simplify the design, with bit positions adjusted within FBinDCT-C9, complemented during RBinDCT-C9s operation. This feature is illustrated in Table 6.15 since RMSE error between software and hardware RBinDCT-C9 configurations were minimal.

	PC Software BinDCT RMSE				XC6264 Implemented BinDCT RMSE			
Seq.	*FBinDCT-C1*	*RBinDCT-C1*	*FBinDCT-C9*	*RBinDCT-C9*	*FBinDCT-C1*	*RBinDCT-C1*	*FBinDCT-C9*	*RBinDCT-C9*
i	0.7206	0.0000	20.9569	0.0000	0.7228	0.0243	48.2278	0.0240
ii	0.0000	0.0000	0.0000	0.0000	0.0000	0.0057	0.0000	0.0057
iii	0.3189	0.0000	11.5004	0.0000	0.3193	0.0164	21.0276	0.0080
iv	0.8889	0.0000	70.8618	0.0000	0.8750	0.0189	58.8740	0.0057
v	0.6226	0.0000	32.4527	0.0000	0.6178	0.0194	67.5423	0.0096

Table 6.15 BinDCT Hardware/Software RMSE Comparison

As in *Section-5.3.2,* the XC6264 BinDCT coding-gain was assessed, with the results generated shown in Tables 6.16 to 6.20. To provide performance benchmarks, the ability of BinDCT configurations to reconstruct data using compressed transform coefficients was assessed. Using true DCT compression as reference levels, (n:N, where 'n' is the number of DCT coefficient(s) at zero, 'N' is the original input sequence length), forward transform BinDCT coefficients were thresholded to obtain the same number of zero coefficients as the DCT. Error between original and BinDCT reconstructed compressed data was then compared to true DCT operation.

With respect to Tables 5.9-5.14 (*Section-5.3.2)* the XC6264 BinDCT-C1 hardware results obtained (*Tables 6.16-6.20*) displayed minimal differences compared to those obtained using software simulations. This was true when calculating the maximum difference in magnitude between input and output sequences, and RMSE between BinDCT transform and true DCT operation.

Results generated for BinDCT-C9 configuration were similar to those in *Section-5.* Similarly to Table 6.15, differences in RMSEs calculated for FBinDCT-C9 operation were attributed to the FBinDCT-C9 and RBinDCT-C9 XC6264 designs. Internal operands were scaled to simplify their design and operation, with bit positions adjusted within FBinDCT-C9, complemented during RBinDCT-C9s operation.

Within Tables 6.16-6.20, RMSE values of zero indicated that no difference occurred between original and reconstructed data for given numbers of forward transform zero coefficients.

	DCT		BinDCT-C1		BinDCT-C9	
Zero Coefficients	*RMSE*	*Max Error*	*RMSE*	*Max Error*	*RMSE*	*Max Error*
1	0.0676	0.0900	0.0733	0.1100	0.0000	0.0000
2	0.1421	0.2100	0.1458	0.2300	0.2176	0.3800
3	0.2165	0.3800	0.1918	0.3600	0.2800	0.6200
4	0.5224	0.7800	0.2516	0.5100	0.2800	0.3900
5	2.1636	3.6700	2.5539	4.0300	22.4524	31.6100
6	7.8873	10.8200	7.7110	10.8800	39.1932	63.8600
7	73.4582	112.1200	73.4582	0.0000	73.4582	112.1100

Table 6.16 Comparison of XC6264/True DCT Compression for Sequence-(i)

	DCT		BinDCT-C1		BinDCT-C9	
Zero Coefficients	*RMSE*	*Max Error*	*RMSE*	*Max Error*	*RMSE*	*Max Error*
7	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 6.17 Comparison of XC6264/True DCT Compression for Sequence-(ii)

	DCT		BinDCT-C1		BinDCT-C9	
Zero Coefficients	*RMSE*	*Max Error*	*RMSE*	*Max Error*	*RMSE*	*Max Error*
5	11.5004	15.0300	11.1313	14.2900	30.6520	42.4900
6	30.0520	42.5000	30.0520	42.5000	30.0520	42.5000
7	70.4783	110.2400	70.4746	106.2400	70.4746	106.2400

Table 6.18 Comparison of XC6264/True DCT Compression for Sequence-(iii)

	DCT		BinDCT-C1		BinDCT-C9	
Zero Coefficients	*RMSE*	*Max Error*	*RMSE*	*Max Error*	*RMSE*	*Max Error*
4	22.9806	31.8800	23.3163	32.4100	0.0000	0.0000
5	35.5370	63.7500	36.1147	64.4500	0.0000	0.0000
6	53.9331	95.6200	54.8274	97.5400	0.0000	0.0000
7	127.5000	127.5000	127.5000	127.5000	127.5000	127.5000

Table 6.19 Comparison of XC6264/True DCT Compression for Sequence-(iv)

	DCT		BinDCT-C1		BinDCT-C9	
Zero Coefficients	*RMSE*	*Max Error*	*RMSE*	*Max Error*	*RMSE*	*Max Error*
1	8.7943	12.2000	8.7943	11.9500	0.0000	0.0000
2	19.0348	32.8800	19.0348	32.4500	45.0780	63.7600
3	31.6564	64.1900	32.0313	64.7300	45.0780	63.7600
5	55.0832	95.4700	55.2994	95.4700	78.0742	127.4900
6	66.6257	139.2600	66.4937	138.6200	90.1532	127.4900
7	78.5712	193.6800	78.5172	193.4900	90.1532	191.2600

Table 6.20 Comparison of XC6264/True DCT Compression for Sequence-(v)

6.5.2 Two Dimensional XC6264 BinDCT Operation

Two-dimensional dynamic XC6264 BinDCT hardware configurations were developed, and operated using techniques described in *Section-5.5*. The optimal BinDCT configuration for each pixel tile (8x8 pixels) within the target image was determined using XC6200DS software tools. This information was then encoded within the source image.

The source image was then downloaded to the C40 DSP (coupled to the XC6264 co-processor). The C40 then converted the pixel values first from floating-point to XC6264 fixed-point twos-complement binary representation, and then from 20-bit parallel to serial notation. Depending upon the BinDCT configuration required, the C40 would (if applicable) update the XC6264 BinDCT configuration using RTR. Dynamic configuration updates used were FDinDCT-C1 to FBinDCT-C9, and RDinDCT-C1 to RBinDCT-C9. The C40 then transferred input operands to the XC6264, generated BinDCT pipeline control signals, and then read the resultant output.

When using XC6264 BinDCT hardware to perform two-dimensional transforms, after each row or column operation (one–dimensional transform), intermediate results had to be scaled and reordering due to the nature of dyadic lifting structures used within the design. This aspect was a feature of the XC6264 BinDCT hardware implementation.

The initial target image use was the 'Lena' benchmark (*Figure 5.15)*, with BinDCT configuration distribution calculated shown in Figure 5.16 (*Section-5.5*). XC6264 two-

dimensional BinDCT operation was assessed through performing a forward then reverse two-dimensional transform and then comparing the difference (RMSE) between the original and reconstructed images pixel values. For comparison identical static and dynamic BinDCT operations were performed within the C40 DSP. Results generated using 'Lena' are listed in Table 6.21.

	Total Summed Pixel Errors		Average Pixel Error		RMSE	
Configuration	*XC6264*	*TMS320C40*	*XC6264*	*TMS320C40*	*XC6264*	*TMS320C40*
Static BinDCT-C1	10652.3	12.367	0.04063	4.72x10-5	0.01607	1.65x10-5
Static BinDCT-C9	10046.3	12.1144	0.03832	4.62x10-5	0.01517	1.61x10-5
Dynamic BinDCT	10530.7	12.3252	0.04017	4.70x10-5	0.15895	1.64x10-5

Table 6.21 Comparison of XC6264/TMS32C40 2D-BinDCT Operation

For both C40 and XC6264 operations RMSE values obtained indicated that reconstructed data contained a degree of error compared to the original. However, using static configuration BinDCT-C1 as an example, the summed pixel error obtained was 10652.3 per 262144 (512x512) pixels, producing an average error per pixel of 0.0406. This tolerance would be acceptable within most image processing applications since pixel values are rounded to the nearest whole number.

The RMSEs of XC6264 compared to C40 BinDCT operations were greater. This was expected due to the maximum resolution of BinDCT coefficients being 0.03125. C40 computations used 40-bit floating-point representation, with a maximum resolution of $5.8x10^{-39}$ [65]. The difference in RMSE value for BinDCT-C1 and BinDCT-C9 configurations reflected the differences in approximation of true DCT operation.

For FPGA based dynamic BinDCT operations, RMSEs generated were less than the equivalent fixed implementations. This factor was a by-product of choosing the BinDCT configuration for each tile that generated the greatest inherent coding gain. Through having greater frequency components at zero, errors introduced through the rounding of intermediate coefficients (system resolution) were reduced.

To further demonstrate dynamic BinDCT operation, optical fringe patterns having differing frequency contents and structure were examined. The resultant BinDCT configurations generated and DC coding gain increases are shown in Figures 6.11 to 6.16. Comparing BinDCT distribution to the original image, BinDCT-C9 configurations (*black*) are most common in areas of low frequency content, whilst BinDCT-C1 (*white*) in areas of higher frequency contents.

Table 6.22 details BinDCT distribution information obtained through computing Figures 6.11-6.16. Speed-up figures calculated demonstrate the percentage of 8x8 pixel tile operations accelerated through using BinDCT-C9 hardware compared to BinDCT-C1 implementations. Through using dynamic BinDCT operation, the percentage of forward transform coefficients at zero (inherent coding gain) has been increased.

	BinDCT Distribution		**Dynamic BinDCT**	**Transform Zero Coefficients**	
Image	*Config. C1*	*Config. C9*	*Speed-up (%)*	*True DCT*	*Dynamic BinDCT*
Pattern(i)	2253	1843	45%	12.08%	20.87%
Pattern(ii)	2416	1680	41.02%	61.6%	75.5%
Pattern(iii)	781	3315	80.93%	12.8%	27.3%

Table 6.22 BinDCT Configuration Distributions

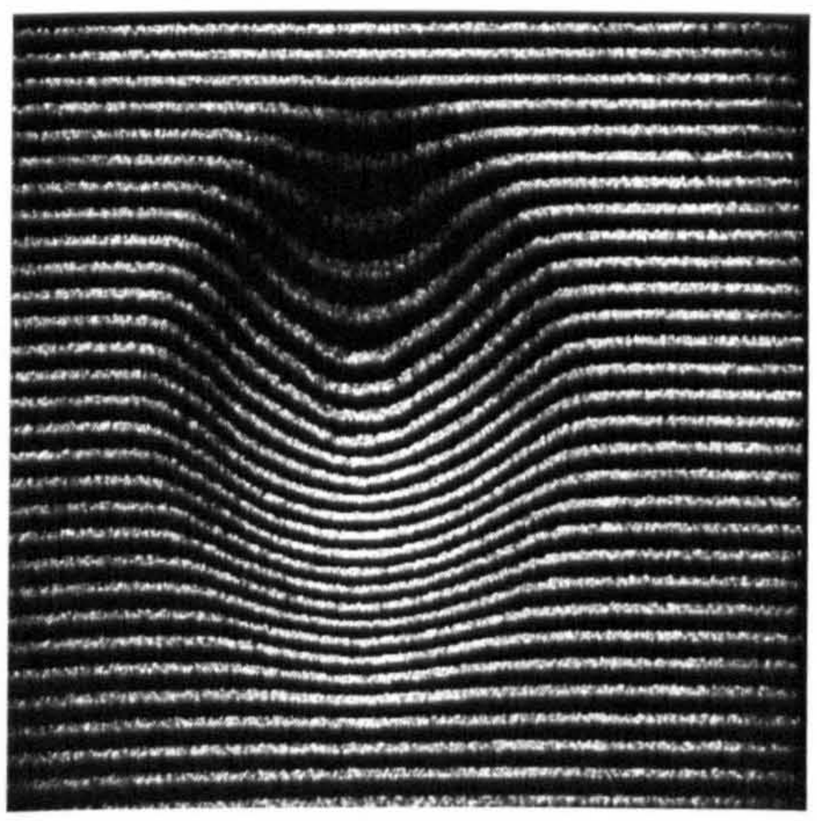


Figure 6.11 Optical Fringe Pattern-(i)

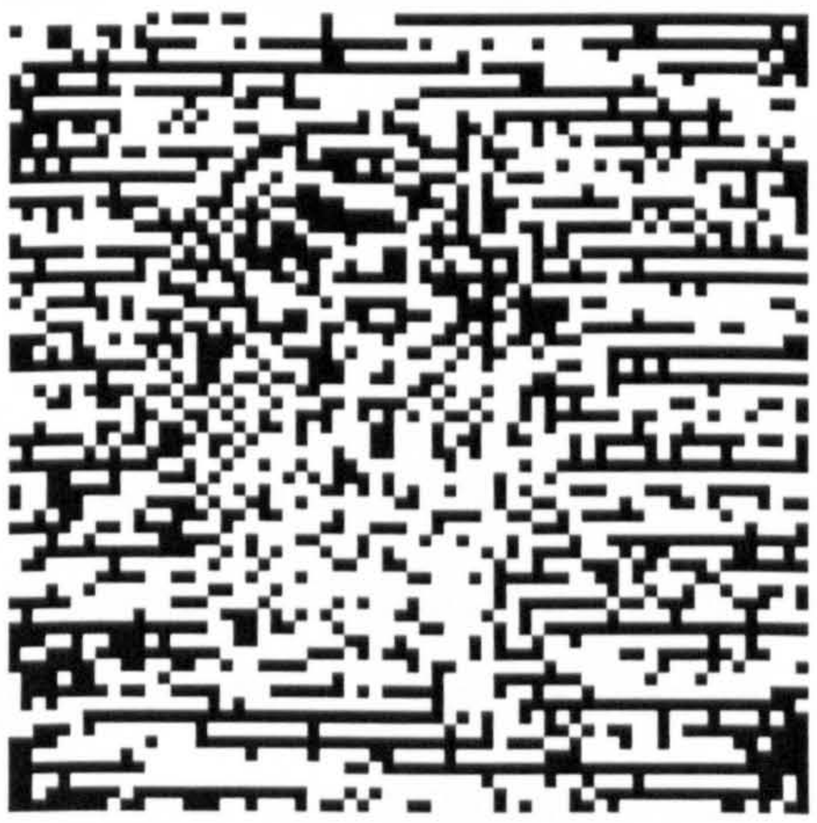


Figure 6.12 BinDCT Distribution

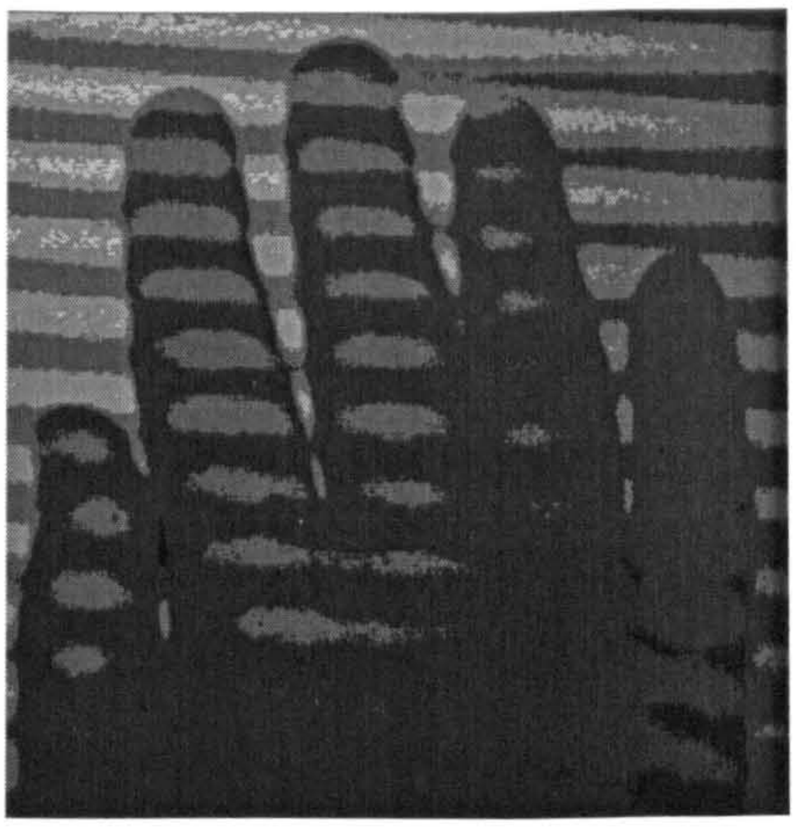


Figure 6.13 Optical Fringe Pattern-(ii)

Figure 6.14 BinDCT Distribution

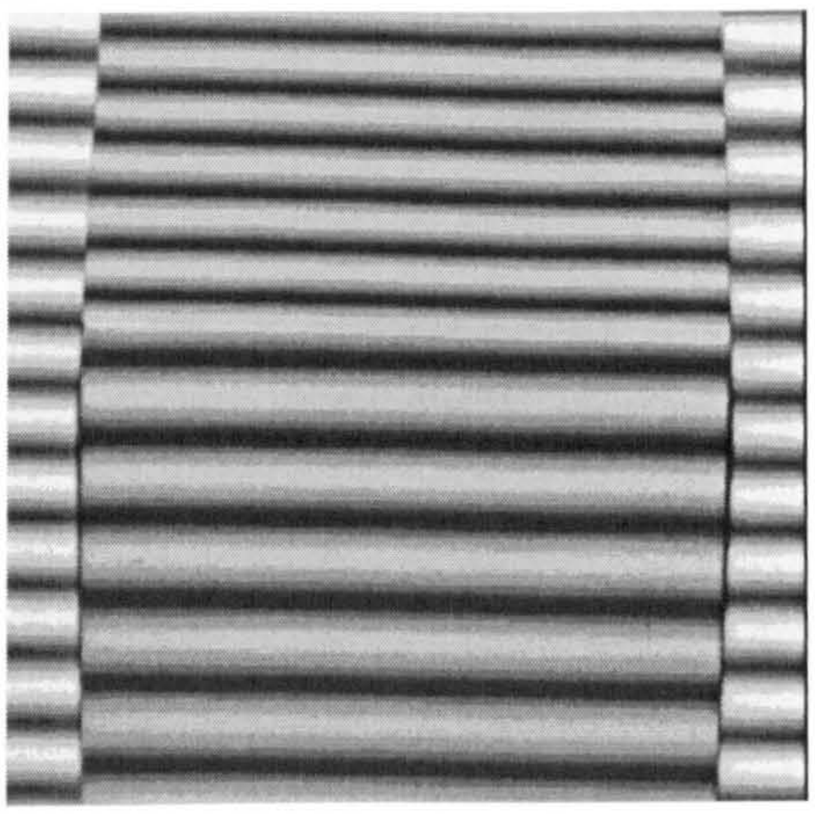


Figure 6.15 Optical Fringe Pattern-(iii)

Figure 6.16 BinDCT Distribution

6.6 Summary

Chapter-6 has described the development and construction of XC6264-based dynamic configurable BinDCT hardware, configured to appear as TMS320C40 memory mapped coprocessors. Through creating dynamic coprocessor topologies, this has permitted the C40 primary processor to exhibit virtual hardware characteristics. This work has demonstrated the potential benefits of including dynamic hardware within traditional fixed processing topologies.

Results generated within *Chapter-6* have demonstrated how BinDCT algorithm operating characteristics have benefited from dynamic hardware implementation. The inherent coding-gain was increased, whilst accuracy in reconstructing the original data compared to fixed BinDCT configurations improved. This concept of dynamic BinDCT implementation and system operation is a novel idea.

Hardware configuration statistics generated, proved the hypothesis that configuration C1 had reduced throughput compared to C9. It was also evident that the implementation of C9 required fewer logic resources than C1. The innovation of RTR hardware for configurations BinDCT-C1 and BinDCT-C9 has allowed not only more efficient BinDCT processing, but shown that by dynamically switching between the two BinDCTs based on the context of each 8x8 pixel data tile, a faster and more accurate transform has been created.

The design methodology used throughout *Chapter-6*, was to simplify routing and placement of BinDCT components within the XC6264 FPGA, at the expense of operand throughput. However, increases in BinDCT compression and accuracy obtained through dynamic compared to fixed operation would be present whatever the implementation method used.

Chapter 7

XC6264 Dynamic Routing Hub

Introduction

This chapter details the insertion of a RTR routing-hub within the C40 MIMD processing topology. The first section of the chapter (*Section-7.1*) explains how the XC6200DS and C40 communication channels were integrated. *Section-7.2* describes the development and function of initial static routing topologies. The next section (*Section-7.3*) expands upon the designs developed in *Section-7.2* and develops the concept of dynamic routing topologies. *Section-7.4* then describes how processing hardware can be configured within the routing-hub using RTR. A summary of conclusions derived through undertaking this work is presented in *Section-7.5*.

7.1 System Overview

The C40 DSP contains six bi-directional communication ports that facilitate processor-to-processor communication. To investigate the benefit of including dynamic hardware within a multiprocessor environment, XC6200 based routing topologies have been developed.

Using the XC6200DS *routing-hub* configuration mode (*Section-3.4.3*), up to nine C40 communication channels can be connected to the XC6200DS through external cabling. Any eight of these channels could be active at any one time. Collectively, the XC6200DS C40 channel hub interfaces were referred to as *comports*. Within this configuration the identity of each comport (range 5 to 13 respectively) reflected the PCB connector number assignments.

7.1.1 Communication Port Interface

C40 communication channels have a bandwidth of 20Mbytes/sec and can reverse their transfer direction within four instruction cycles (200nsec @40MHz) [65]. This

operation occurs through the interaction of C40 communication channel *Port Arbitration Units* (PAUs). To couple C40 channels to the XC6200DS, the PAU and communication channel interfaces were initially synthesized within the XC6200 FPGA.

The XC6200 PAU implementation however could not complete direction transfers within 100nsec. It was therefore decided that to prevent common signals from being driven by both the C40 and XC6200 PAUs during bidirectional transfers, communication channel direction would be fixed. This did not detract from the communication strategies, as enough uni-directional ports were available for the developed switching fabrics. The XC6200 PAU was therefore removed from the design, however the respective C40 control signals (CREQ, CACK) could not be left open-circuit and instead were tied to logic-one (*Figure 7.2*).

The data-bus of C40 communication channels was 8-bits wide. Data transfers occurred using four-byte words, since the internal C40 architecture was 32-bits wide. A C40 word transfer operation therefore consisted of four individual byte cycles, which were managed using signals CSTRB and CRDY. This control mechanism is shown in Figure 7.1, where CSTRB was generated by the interface transmitting data (*CSTRB_TX*), and CRDY by the receiving unit (*CRDY_RX*). The signal connections required between interface receiver and transmitter comports to C40 communication channels are shown in Figure 7.2.

Valid data at the receiving comport was determined using the negative-edge of signal CRDY. For transmission, data had to be presented before the negative-edge of signal CSTRB. These signals were generated through the interaction of C40 and XC6200 comport interface control state machines.

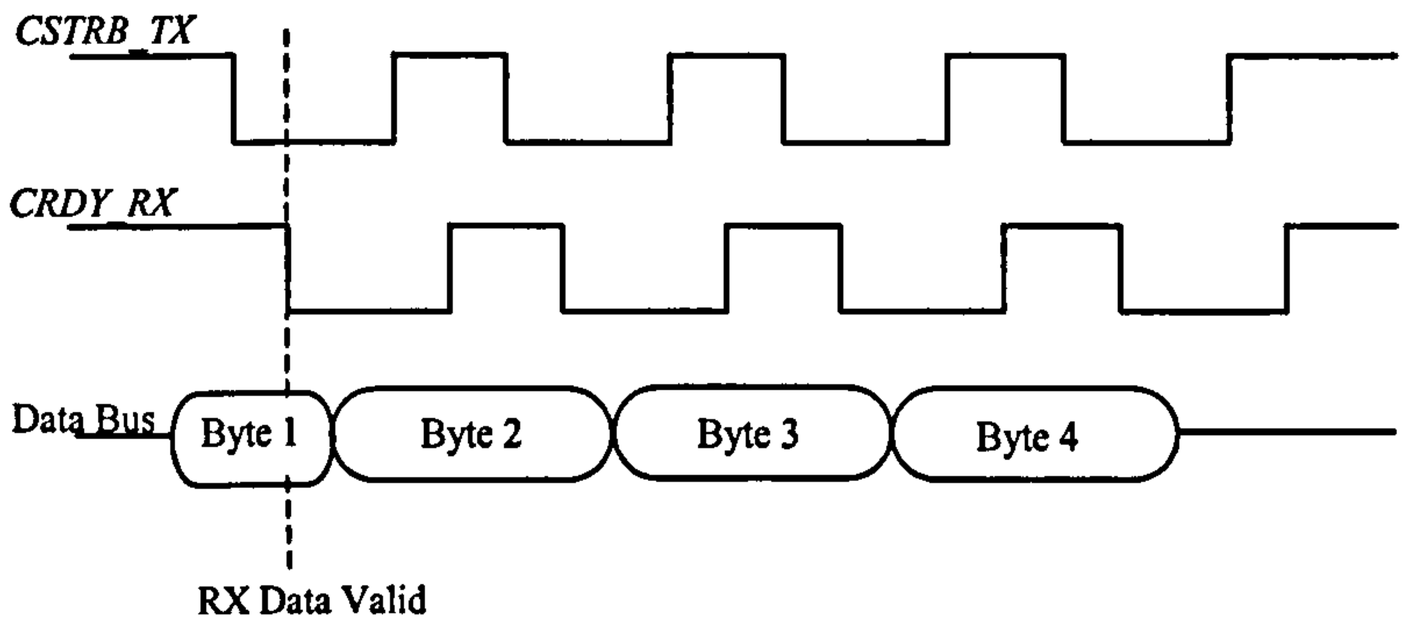


Figure 7.1 Communication Channel Byte Transfer Protocol

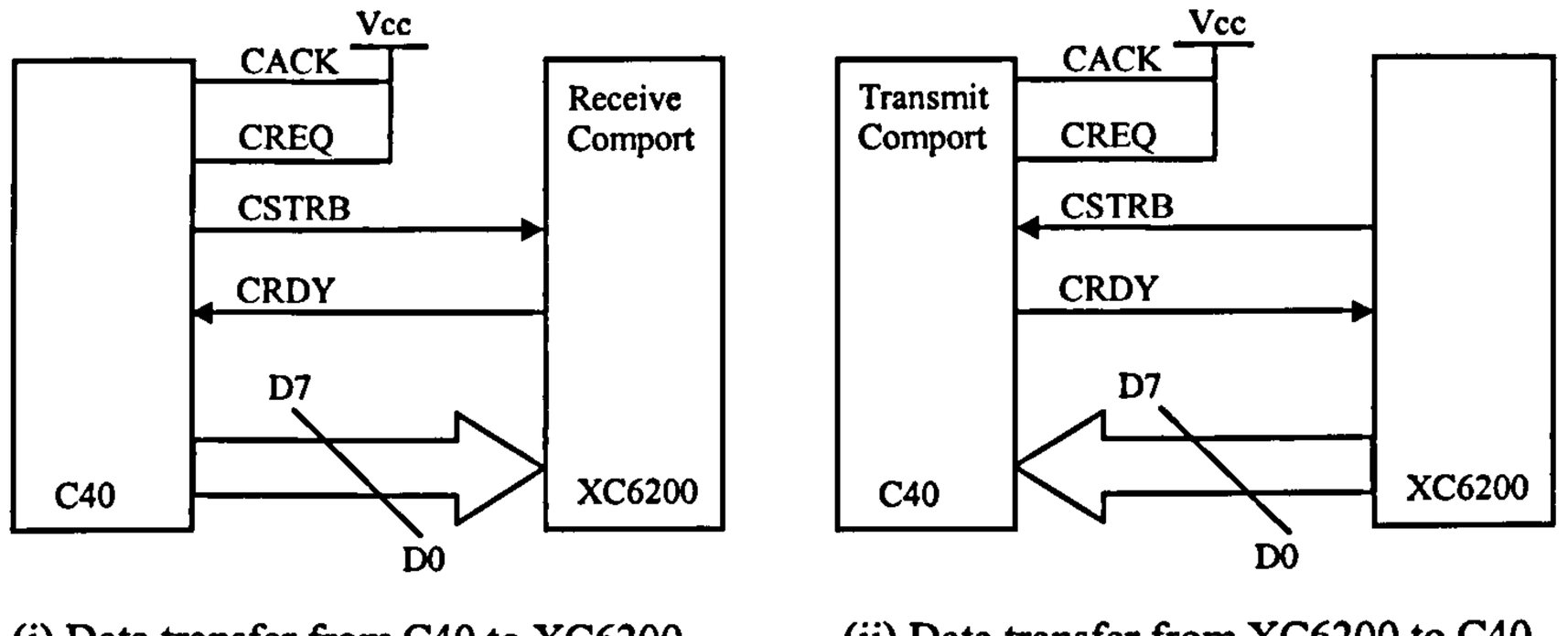


Figure 7.2 Communication Channel Signal Connections

7.1.2 Transfer Management

Port transfers within the XC6200 were initiated through discrete logic functions. Within the C40 this task was governed by software programs constructed using a C40 variant of the C programming language. The management of channel interfaces was invisible to user-code, with data transfer controlled using a library of pre-complied C functions. The formats of two common C functions are shown in Program 7.1.

```
out_word(datatx, 1);
printf ("Data TX is %x   \n", datatx);
datarx = (int) in_word(4);
printf ("Data RX is %x   \n", datarx);
```

Program 7.1 C40 Communication Channel Functions

Function *out_word(m, n)* was used to write data to a C40 communication port. The function had two arguments. *m* was the 32-bit word to be written to channel *n* (range 1 to 6). Function *in_word(n)* received data from C40 communication channel *n*, and returned the value read.

7.2 Comport Transfer Mechanisms

XC6200 comport designs consisted of state-machines and data-buses. The state-machines generated and managed control signal handshaking with the C40, whilst operand transfer occurred using the data-bus.

For the XC6200DS to appear as a transparent routing hub, data received from one C40 port had to be re-transmitted to another. To manage the transfer of data between each XC6200 comport, two control methods know as *FIFO Control* and *Self-Arbitration* were developed.

7.2.1 FIFO Control Unit

The structure of the FIFO Control mechanism developed is illustrated in Figure 7.3 and contained comport interfaces, a four-byte deep FIFO, and control logic. Comport connector interfaces consisted of signals D7-D0, CSTRB, and CRDY as shown previously in Figure 7.2.

The receiving comport was activated using signal *go_rx* (generates *er_w*). Individual bytes within C40 32-bit word (four bytes) were then transferred, with each byte in turn being latched into the FIFO upon the negative edge of *CRDY_RX* (*via clk_fifo*) (*Figures 7.1 and 7.3*).

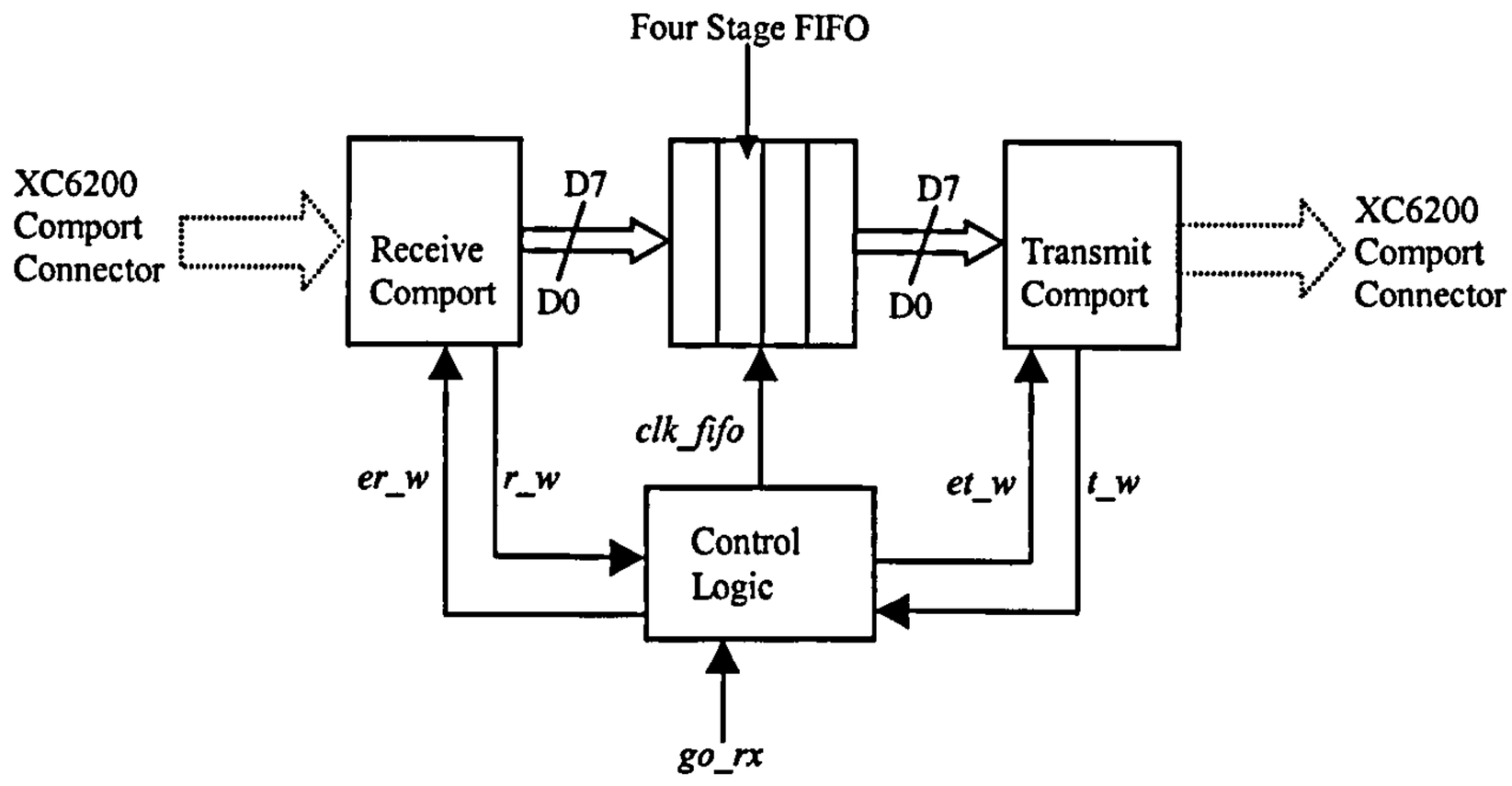


Figure 7.3 XC6200 FIFO Control Comport Management

The receiving comport indicated when it had accepted four bytes (32-bit word) with signal *r_w*. Router control logic then instructed the transmitting comport via *et_w* to write the contents of the FIFO to the next C40. The contents of the FIFO were incremented upon the transmission of each byte through signal *CSTRB_TX* (*via clk_fifo*) (*Figures 7.1 and 7.3*). Once the FIFOs contents were written signal *t_w* instructed the control logic that the FIFO was ready to receive another four bytes from the receiving comport.

The structure of FIFO control can be considered similar to C40 communication port operation. Each C40 communication port transfers 32-bit words as four individual bytes. To minimise communication bottlenecks, each C40 port had eight level 32-bit wide FIFOs. For dedicated processor-to-processor communication links this provided a buffer of 16 words. In contrast within the XC6200 FIFO control mechanism, the FIFO was used to buffer 4 bytes (one 32-bit word) during transfer within the communication hub, and not act as a buffer within the receiving and transmitting C40s. This was a very efficient use of limited FPGA resources.

7.2.2 Self-Arbitration Unit

The structure of the self-arbitration mechanism is shown in Figure 7.4. This system comprises both receiver and transmitter comports, with the FIFO replaced by a single register clocked using *CRDY_RX*. The function of the router control logic was replaced

by a handshaking operation between the receiver and transmitter comports (signals *go_tx* and *tx_byte*). Effectively self-arbitration functioned by receiving and transmitting each byte within a 32-bit word independently. This design was the minimum required, with maximum efficiency in terms of hardware resources used.

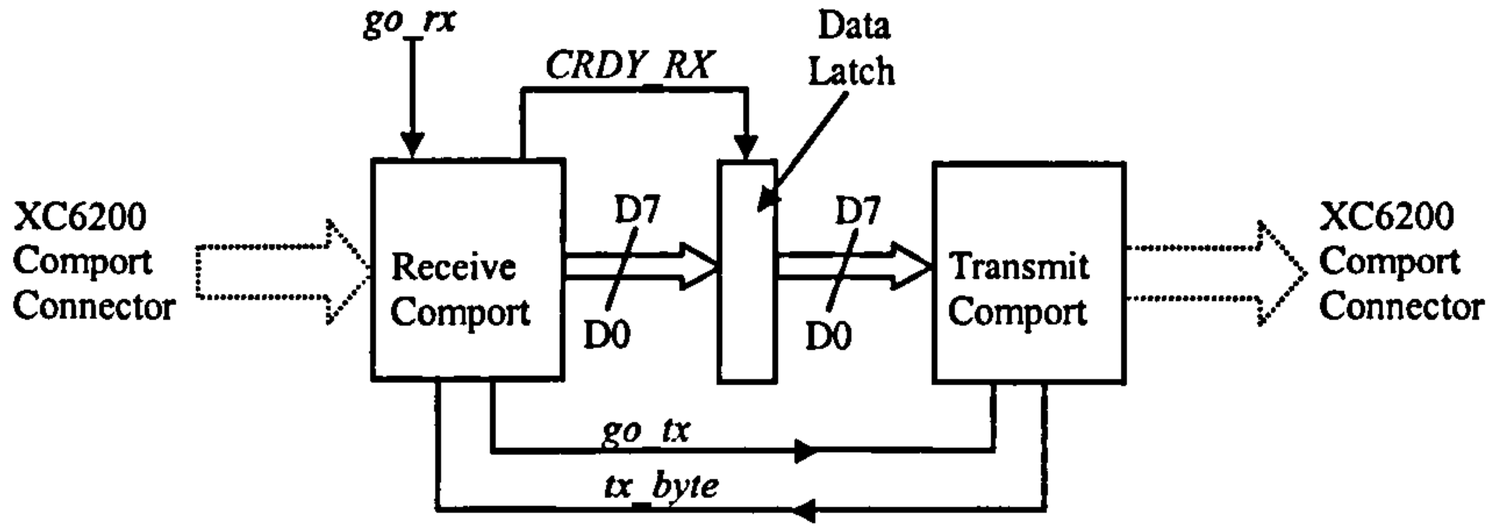


Figure 7.4 XC6200 Self-Arbitration Unit Comport Management

7.2.3 Transfer Protocol Analysis

The two protocols were evaluated through configuring identical routing topologies within a XC6264 FPGA. The routing structure used was a direct connection between comport-8 and comport-12 of the XC6200DS. These two comports were chosen since they were located on opposite sides of the XC6264 FPGA, and would encounter the greatest signal propagation delays inside the device. Timing and hardware characteristics generated using XACT6000 upon a XC6264 FPGA for both protocols are listed in Table7.1. These results compare the maximum signal delay when both designs are routed using local (*length-16*) and global (*chip-wide*) routing resources; Length-16 and chip-wide XC6200 FPGA routing resources are detailed in *Appendix-III-2*). The number of CLCs required in implementing both state machine control and data path components for both protocols are also listed.

	Max Signal Delay		CLC Volume	
	Local Routing	*Global Routing*	*State Machines*	*Data Path*
Self Arbitration	56.018nsec	58.947nsec	43	0
FIFO Control	56.02nsec	55.932nsec	94	24

Table 7.1 Transfer Protocol Hardware Characteristics

C40 software was written to verify the operation of routing topology as well as generating benchmark results. Operand transfer delays were generated for FIFO control, self-arbitration configurations, and for comparison direct C40 to C40 connections. The results obtained are shown in Table 7.2, with a XC6264 clock frequency of 8MHz.

	C40 Glue less Connection	FIFO Control Unit	Self-arbitration Unit
No. of Words	*Operand Transfer Delay*	*Operand Transfer Delay*	*Operand Transfer Delay*
1	4.8µsec	9.6µsec	8.4µsec
10	3.09msec	3.14msec	3.12msec
262144	22.14sec	23.18sec	22.16sec

Table 7.2 Transfer Protocol TMS320C40 Timings

Operand transfer timings were generated using the C40s internal timer. These results illustrate the difference in transfer delays introduced by the control mechanism within the C40 MIMD. The transfer delays generated however also constituted additional C40 instructions delays and C40/host PC interrupts. These were required by the C40 to access comport data, which was then written to external files for error checking purposes. However no errors were detected between transmitted and received data sets.

Using XACT6000 software, the maximum throughput of the FIFO control method was determined as 4.469Mbytes/sec (using global routing), whereas the self-arbitration unit was 4.462Mbytes/sec (using local routing).

The results obtained showed that the bandwidth obtained for each transfer protocol was similar. However, the simplicity and internal regulation of the self-arbitration was deemed superior. This was evident through the self-arbitration method not requiring additional control logic and FIFO buffers within the data path.

The FIFO protocol achieved highest bandwidths when chip-wide global XC6264 routing resources were used. Within XC6200 FPGAs these resources were limited, therefore in complex designs local routing had to be used, incurring greater propagation delays.

The self-arbitration method however was more suited to local than global routing implementations. Because of this factor and not requiring a FIFO to be present between each XC6200 comport route, the self-arbitration control mechanism was used within XC6200 router designs.

7.3 Static Routing-Hub Development

To investigate the advantages gained by including adaptable routing resources within the C40 MIMD architecture, XC6200DS hardware was developed that facilitated bi-directional operand transfer between four C40 DSP nodes. These designs were implemented within a single XC6264 FPGA.

7.3.1 Hub Construction

To develop router hardware compromises to channel bandwidth and node connectivity were made to aid XC6200DS placement, design and routing of integral components. Apart from low operating speeds, a disadvantaged of the XC6200 FPGA family was that tri-state gates could not be configured within XC6200 CLCs, resulting in fixed direction data buses. This limitation also forced the inefficient construction of crossbar switches using multiplexers, which would cause multiple delays to routed data.

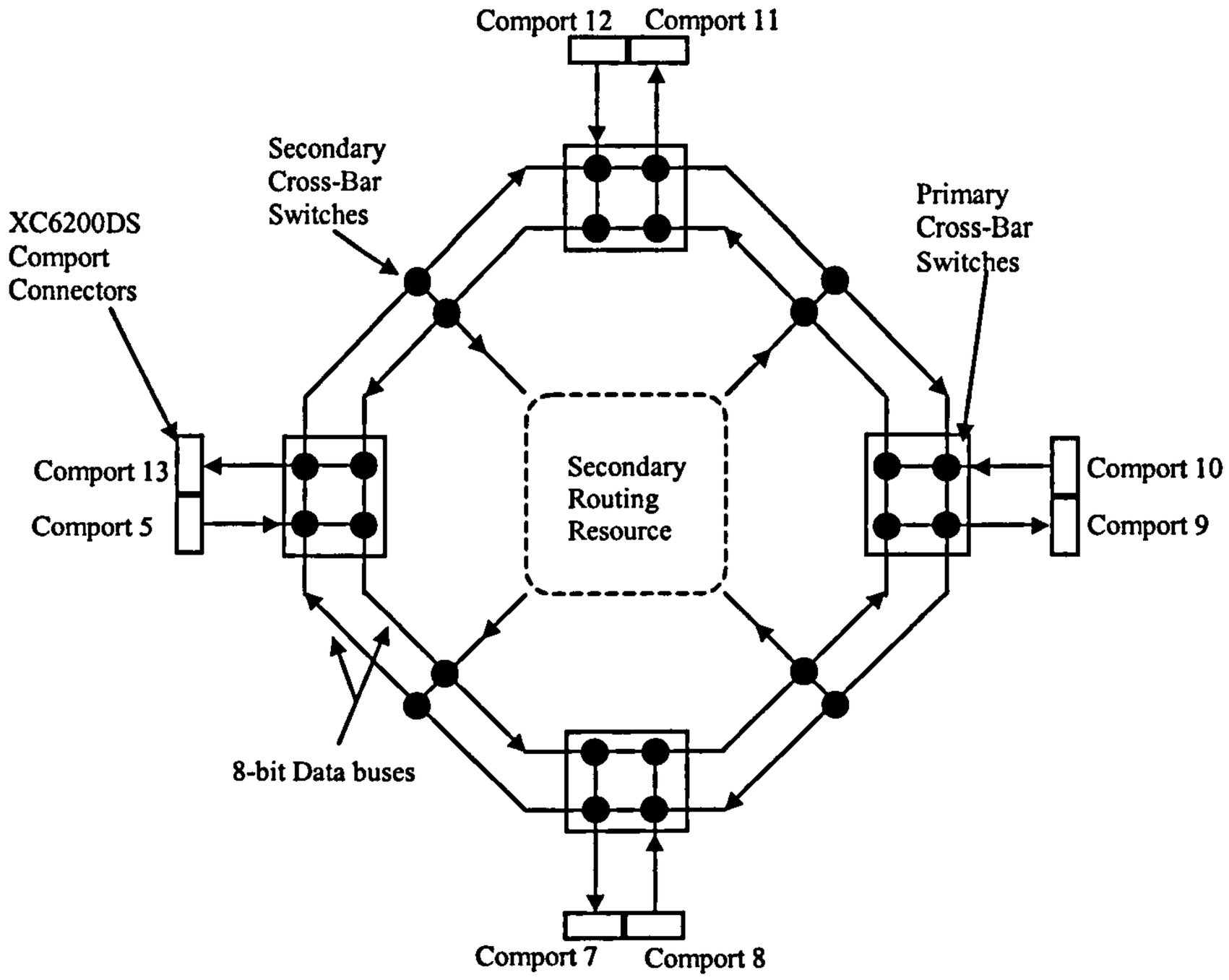


Figure 7.5 XC6264 Routing-hub Topology

The routing-hub topology developed is illustrated in Figure 7.5. The design was similar to the Chordal Ring architecture [80] since two unidirectional data buses link together four primary crossbar switches. Unlike traditional routing designs there existed a 'user' area were secondary routing resources and processing hardware could be configured. Expanding this concept, the four primary switches and unidirectional buses were considered as a base platform upon which further configurations could be developed.

Primary crossbar switches consisted of two XC6200DS comports (receive and transmit) and three banks of multiplexers. These were used to implement the crossbar switch illustrated in Figure 7.6, using a switch configuration table listed in Table 7.3. The data transfer direction of both comports however were fixed.

The secondary crossbar switches (*Figure 7.5*) provided accesses to unused CLC resources within the XC6264. Unused CLCs could be configured to appear as additional

routing resources (approximately 25% of XC6264 CLC array) within the topology or as processing hardware. This concept was further developed in *Section-7.4*.

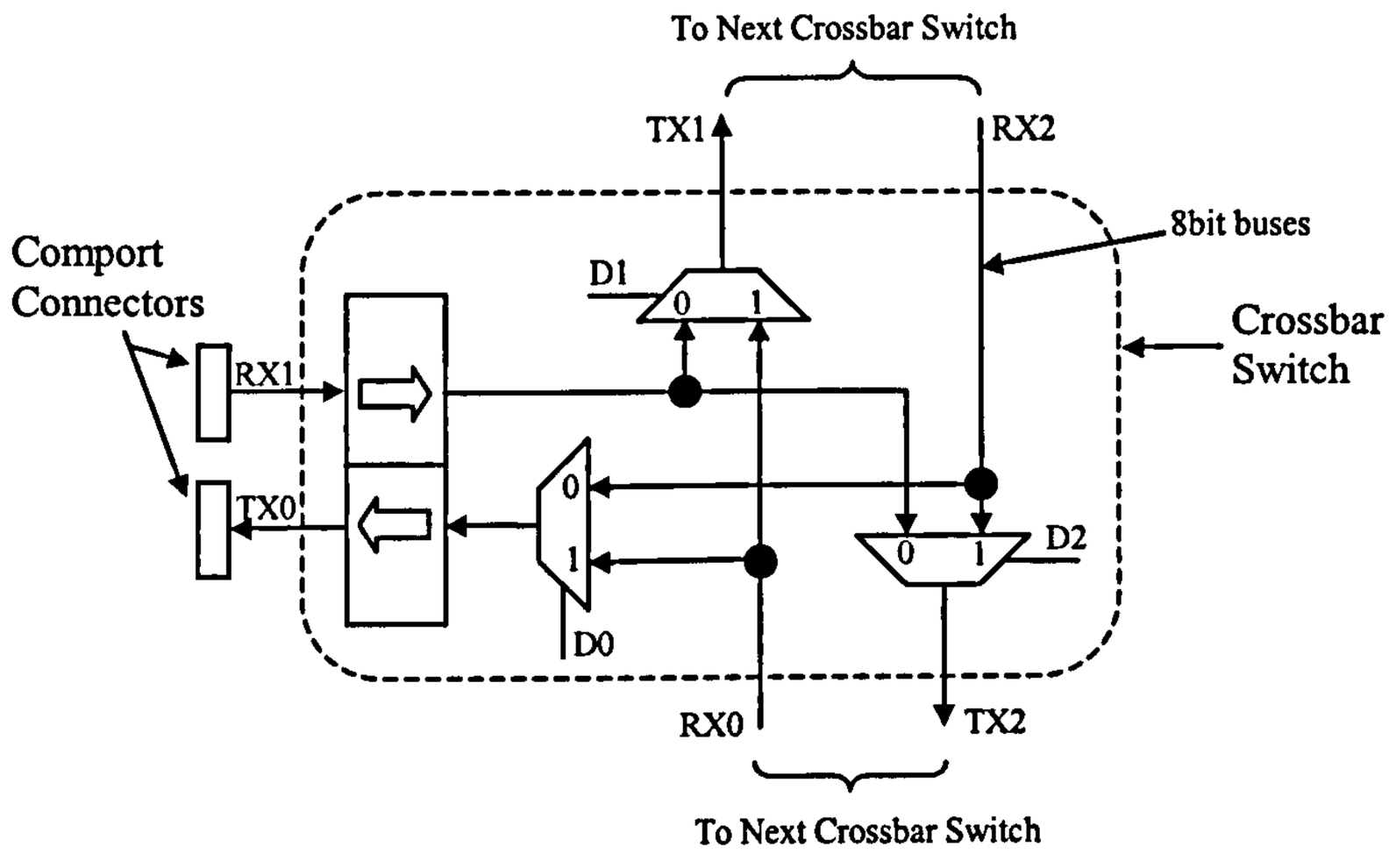


Figure 7.6 Crossbar Switch Construction

Crossbar Control Signals			Crossbar Output Busses		
D0	*D1*	*D2*	*TX0*	*TX1*	*TX2*
0	0	0	RX2	RX1	RX1
0	0	1	RX2	RX1	RX2
0	1	0	RX2	RX0	RX1
0	1	1	RX2	RX0	RX2
1	0	0	RX0	RX1	RX1
1	0	1	RX0	RX1	RX2
1	1	0	RX0	RX0	RX1
1	1	1	RX0	RX0	RX2

Where: RX0, RX1, & RX2 are Crossbar Input Busses

Table 7.3 Crossbar Switch Configuration Table

7.3.1 Hub Operating Characteristics

The operation of the routing-hub was verified using custom designed C40 software. Two C40s functioning in a MIMD configuration were used to transmit and receive data

through communication ports. Within this test procedure, different hub topologies were configured and the operation analysed. During this operation, the configuration of the routing-hub remained static during system operation, and reconfigured using CTR methods (*Section-2.3.1*).

The performance characteristics of the routing-hub were assessed for throughput and comport connectivity. During this process, the clock frequency of the design was set to 8MHz. The maximum signal propagation delay recorded was 275.21nsec. This value related to signal delay encountered through circumnavigating the routing topology (excluding secondary routing resources) resulting in individual channel bandwidths of 908.4kbytes/sec, and 7.26Mbytes/sec for the eight channels combined. When secondary routing resources were configured, the maximum signal propagation delay of the design increased to 389.92nsec, resulting in individual channel bandwidths of 641.16kbytes/sec and 5.13Mbytes/sec combined.

Signal routing delays encountered were excessive when compared to the clock frequency period (125nsec). Nevertheless the design still functioned correctly, since the XC6264 component placement used ensured that the self-arbitration control signals encountered similar routing delays to data buses.

This operating concept was not ideal for use in dynamic routing-hub development, and the XC6264 clock frequency was reduced to 2MHz. In comparison, the maximum operating frequencies obtained for each XC6264 comport was in the range of 35-39MHz.

Within the routing-hub design, a receiver comport could broadcast data to all transmitter comports. To perform this operation crossbar switches were configured accordingly and the self-arbitration control signals of each comport combined.

Through configuration of the primary crossbar switches, bi-direction transfers between adjacent comports could occur using two unidirectional data paths. For data transfer to

occur, self-arbitration unit handshaking signals *go_tx* and *go_rx* (*Figure 7.5*) belonging to each comport forming the communication channel had to be connected together.

The individual bandwidth of comports operating at a clock frequency of 2MHz was calculated as 500kbytes/sec. In comparison, the bandwidth of a C40 communication channel was 20Mbytes/sec. Inserting the routing-hub within a C40 MIMD structure substantially reduced operand throughput. This limitation was expected due to the architecture of the XC6264 FPGA, and accepted, since the aim of the experiment was to determine if such architectures were viable and if they improved the versatility of a multiprocessing architecture.

The routing-hub was implemented within a XC6264 FPGA. The design consisted of 562 CLCs, but required a placement footprint covering the whole CLC array (128 by 128 CLCs) (see *Appendix-VI* for XC6264 layout). Within the design, 258 CLCs were required to construct the primary and secondary switches. 128 CLCs formed guides for routing data-bus signals between individual crossbar switches. Individual transmitter and receiver comports required 20 and 24 CLCs respectively to implement. Comparing the volume of CLCs used to form crossbar switches and router guides (386 CLCs), to those actually implementing comport logic (176 CLCs), demonstrated how much the XC6264 FPGA in this application was pushed to its limits. Confined within the secondary routing resources approximately 4096 (25%) of XC6264 CLCs could be used to implement either additional data-paths or processing hardware.

Using this routing hub, dynamic routing-hub configurations, including processor functions were developed. These are detailed next in *Sections-7.4* and *7.5* respectively.

7.4 Dynamic Routing Topology

The XC6200 routing-hub topology developed in *Section-7.3* remained fixed during system operation and was reconfigured using CTR for each application. During run-time, this architecture incurred connectivity and operand bandwidth constraints associated with fixed multiprocessor architectures. To reduce these limitations RTR routing-hub strategies were investigated with the following solutions.

7.4.1 Configuration Mechanisms

RTR configuration can be performed within the XC6200DS using two methods. The self-configuration mechanism downloaded configuration data automatically without external intervention at a rate of 1.8μsec (@8MHz) per XC6200 address/data pair. The second mechanism requiring user intervention used XC6200ADS software functions, with the average configuration delay measured at 258μsec per XC6200 address/data pair (delay includes host PC interrupt operation).

For optimal performance dynamic configuration must occur without user intervention. The self-configuration mechanism permitted this feature but could not be used within the routing-hub due to its construction. This was because XC6264 comports and self-configuration memory interfaces (external memory and C40) used the same XC6264 pin locations.

If the self-configuration mechanism was implemented, comport-11, comport-12, and comport-13 could not be used. Similarly if the XC6200-C40 Global interface was configured comport-7 and comport-8 were inaccessible. Three comport connectors would therefore be available to implement the routing topology preventing bi-directional transfer (requires four comports) between two C40 DSPs.

The decision was therefore taken to perform RTR through XC6200ADS interaction. This implied adaptation of the routing topology could not be automated by the C40 (true

RTR), but increased operational diversity through the user being able to manually select the next active configuration.

The concept of automating active routing topology determination within the hub, through analyses of system bottlenecks was also assessed. It was concluded however, that this work was beyond the scope of this project and would be recommended for further investigation (*Chapter-9*).

7.4.2 Implementation Strategies

Dynamic routing topologies were configured within the XC6264 using two design approaches defined as *structured* and *non-structured.* Structured router architectures as the name suggest had defined skeleton architectures. An example of this type of architecture was the routing topology developed in *Section-7.3*.

Non-structured routing topology was the term given to designs that did not contain predefined data buses and crossbar switches. The routing of buses between system components was defined instead by XACT6000 during compilation. This methodology was used within the direct connection router designs developed in *Section-7.1.1*.

To compare the merits of structured and non-structured operation, RTR routing topologies consisting of two configurations were developed. The routing topology configurations devised are shown in Figure 7.7.

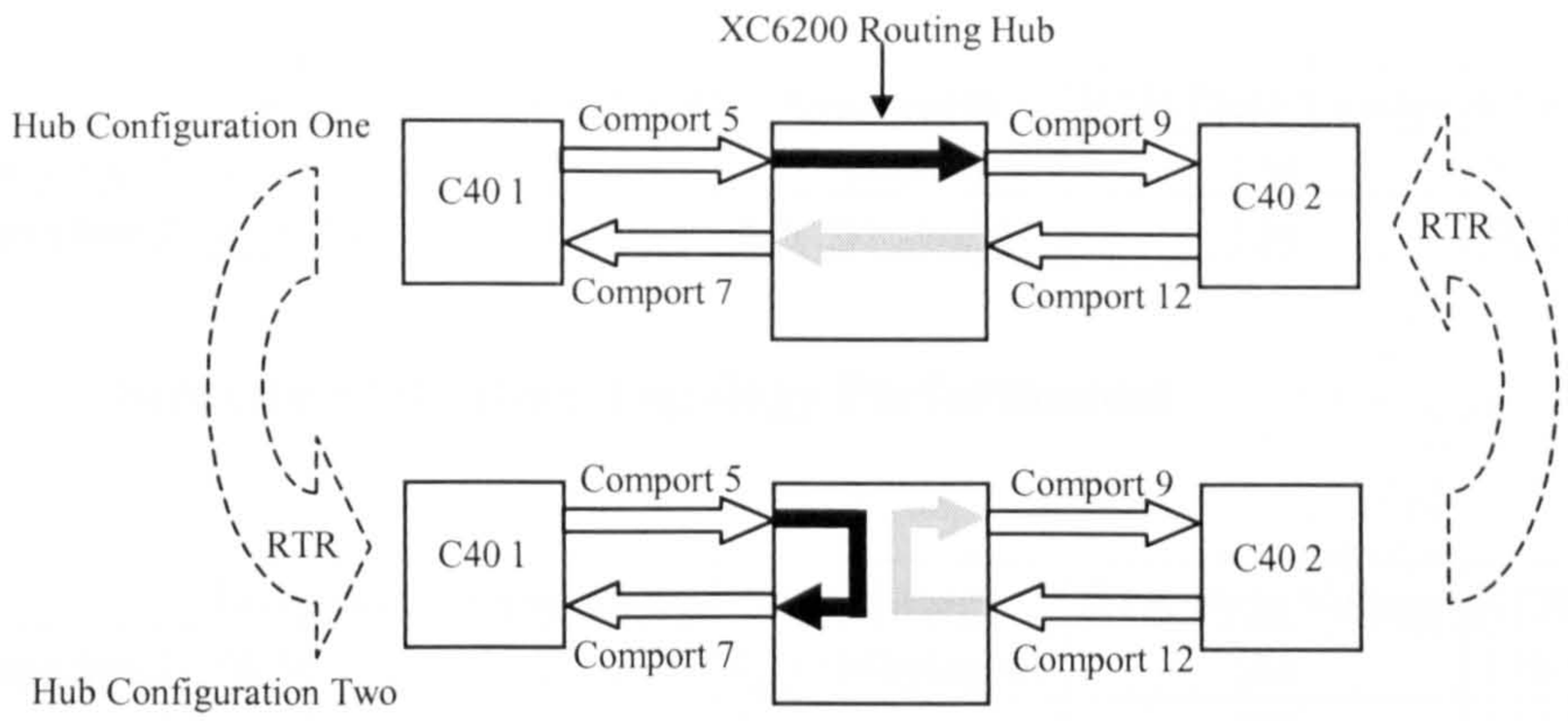


Figure 7.7 Dynamic Routing-hub Configurations

Each configuration required four comport connections interconnecting two TIM-40 (C40) nodes. The two configurations could reside within the XC6264 using temporal partitioning, performed through RTR. Configuration-one consisted of two data paths. Within the first path, C40-1 transmits to C40-2 via Comport-5 and Comport-9 of the routing hub, whilst the second path was formed by C40-2 transmitting to C40-1 using Comport-12 and Comport-7.

The second XC6264 configuration consisted of two data paths, with C40-1 and C40-2 both using the routing-hub channels to transmit data from one local comport back to another in closed loop fashion.

Using these configurations, both structured and non-structured routing implementations were developed. Each implementation method was spatially and temporally analysed using XC6200ADS tools. The operational characteristics of each configuration are listed in Tables 7.2 and 7.3. These contain the maximum operating frequency, signal propagation delay, channel bandwidths, volume of configuration data required to swap between configuration–one and configuration-two (XC6200 address/data pairs), and measured XC6200ADS configuration delay (using external timer). XC624 footprints for each design are shown in *Appendix-VI*.

	Frequency	Signal Delay	Bandwidth	RTR Data Volume	RTR Delay
Configuration 1	3.519MHz	284.171nsec	879.7kbytes/sec	136	50.59msec
Configuration 2	3.518MHz	284.260nsec	879.5kbytes/sec	136	50.59msec

Table 7.4 Structured Routing Topology Performances

	Frequency	Signal Delay	Bandwidth	RTR Data Volume	RTR Delay
Configuration 1	16.86MHz	59.313nsec	4.215Mbytes/sec	588	174.1msec
Configuration 2	16.794hz	59.544nsec	4.199Mbytes/sec	588	174.1msec

Table 7.5 Non-Structured Routing Topology Performances

Operating frequencies listed in Table 7.4 indicated that the signal propagation delay within the structured topology was almost constant for each configuration; Results indicated a difference of 0.089nsec in maximum signal propagation delay between configurations. The maximum signal propagation relates to the signal delay encountered throughout one cycle of each unidirectional ring and is attributed to the XC6264 signal routing topology.

Table 7.5 indicated that the communication bandwidth of non-structured designs was far greater than that of structured design (*Table 7.4*). This was expected since XACT6000 mapping of system components was not restricted (as in structured designs) and therefore placement was optimised for speed; XC6264 footprints in *Appendix-VI* highlight this feature.

With the structured architecture the data-bus paths used within configuration-one and two can be considered equal to a half and a quarter distance respectively of the total ring length. Using this approach, structured topology results were adjusted with the revised maximum operand throughputs calculated for each structured configuration shown in Table 7.6.

	Structured		Non-Structured	
	Configuration 1	*Configuration 2*	*Configuration 1*	*Configuration 2*
Mbytes/sec.	1.759	3.518	4.215	4.199

Table 7.6 Adjusted Routing Topology Bandwidths

The analysis indicated that non-structured designs exhibited greater operand throughput compared to structured designs. This difference in performance was predicted since crossbar switches were absent within non-structure designs. The results also indicated the volume of configuration data required for RTR of non-structured routing designs was far greater than identical structured topologies. This reflected the vast differences between individual non-structured configurations.

If sequential non-structured router configurations did not occupy common XC6264 resources, it was possible to implement multiple configurations within the XC6264. In effect multiple individual configurations would be combined within one static configuration, with the actual routing topology selection instead being performed through updating the C40s operating software.

To verify dynamic re-configuration of each routing topology, two TIM-40 C40 nodes executed test programs. Each C40 wrote different sequences of data to the routing hub. The active configuration of the hub then determined which sequence each C40 node would receive.

Within the TIM-40 motherboard architecture, a hardwired routing topology existed between individual C40 positions (*Section 3.2.2*). Since only the JTAG root node (C40-1) could communicate with the host PC, results recorded by C40-2 were transmitted to C40-1 using this existing communication network. The combined results were then displayed on the host computer.

Dynamic switching of each configuration was performed using XC6200ADS tools controlled through user intervention. In conjunction with C40 test programs, the dynamic switching capability of both design strategies was verified.

During this test procedure however, it was observed that positions of individual bytes within the C40 communication channel packets would be incremented causing data

transfer errors. This was an infrequent feature, but occurred upon completion of dynamic reconfiguration.

Byte position shifts were caused through the manual instigation of reconfiguration not being in synchronism with the free-running C40 operation. If the C40 governed RTR using the self-configuration control mechanism, byte shifts would not occur since RTR completion and C40 communication channel operations would be sequenced.

Analysis of non-structured configurations, determined that the allocation, distribution of routing resources and operand throughput was dependant upon the comport positions used. Through developing further explorative hardware configurations, it was apparent that routing constraints encountered when using non-structured data paths would prevent the insertion of processing elements within a routing hub.

It was concluded that to develop routing-hub based processing elements a structured design approach was desired. Fabricating a discrete routing topology using a non-structured design would generate greater operand throughput, but at the expense of incurring greater RTR delays.

7.5 Routing-Hub Processing Elements

The previous section investigated the merits of including dynamic routing resources within MIMD DSP architectures. In comparison to the existing router hardware, the operating characteristics of the XC6200 implementations were poor. This was attributed to limitations within the XC6200 FPGA architecture and not the operating principles of the design.

To further investigate this aspect of dynamic routing hubs, processing elements were configured within hub data paths. These processing elements implemented simple fine-grain local operations, computed inefficiently within the DSP architectures optimised to accelerate coarse-grain operations. Coarse grain functions typically require floating-

point calculations unlike local operators that can be processed using fixed-point notation.

Within digital image processing applications, the processing power associated with DSP multiprocessor architectures was normally harnessed to extract information from within an image [2]. Prior to performing this function, pre-processing of the image typically occurred and included functions such as binary threshold and edge detection.

Upon commencing system operation, operands and images within a multiprocessor environment must be distributed amongst system nodes via a routing topology. If local fine-grain type pre-processing functions were performed during operand/image transfer, computation overheads for each node would be reduced. Once pre-processing operations had finished, unused hardware within the routing-hub could then be re-adapted to increase communication bandwidth between system nodes. Within this concept, the routing-hub itself could appear as an additional processing node, or as shared memory, if routing-hub transfers were not required during phases of system operation.

Through constructing the routing hub, it was apparent that logic resources available to implement such processing elements would be limited. It was possible however to develop a simple applications to demonstrate this concept. The primary operation developed was the Roberts Cross Edge Detector. This algorithm is described next.

7.5.1 Roberts Cross Edge Detector

The Roberts Cross edge detector, as the name suggests is used to detect gradient changes (edges) within images. This function operated using two convolution masks as shown in Figure 7.8, with each mask scanned over the input image. The mask coefficients appear similar to the Sobel operator and are rotated so that mask-one determines gradient changes in the X-plane of the image, whilst mask-two the Y-plane.

Through combing the operation of the two masks, the contour information of the target image can be extracted. This can be considered a local-type operation since each calculation requires a pixel connectivity of four. Further only one addition and two subtraction operations are required within the computation. The simplistic operation of mask operation however makes the Robert Cross operator vulnerable to noise within the image.

P1	P2
P3	P4

Pixel Grid Positions

+1	0
0	-1

Mask One X-Plane

0	+1
-1	0

Mask Two Y-Plane

Figure 7.8 Roberts Operator Mask Coefficients

The combined functions of both masks are shown in Equation 7.1.

$$|G| = \sqrt{Gx^2 + Gy^2}$$

Equation 7.1

Equation 7.1 can be approximated using Equation 7.2, with the operation re-written in terms of the mask operations in Equation 7.3.

$$|G| = |Gx| + |Gy|$$

Equation 7.2

$$|G| = |P1 - P4| + |P2 - P3|$$

Equation 7.3

To determine the presence of an edge, the resultant gradient is quantisised. If the magnitude of gradient (G) is greater than the threshold an edge has been detected and a black pixel is written to the output image in the corresponding grid position of mask pixel P3.

In typical Roberts Cross applications, edge detection is the only function required. It is

also possible to determine the orientation of the edge within the output image. This operation is shown in Equation 7.4.

$$\phi = \tan^{-1}\left(\frac{Gy}{Gx}\right) - \left(\frac{3\pi}{4}\right)$$

Equation 7.4

To determine the edges orientation, the base angle was taken as the direction of maximum contrast running from left to right of the output image.

An example of the Robert Cross operation is shown in Figures 7.9 and 7.10. Figure 7.9 is the original image, whilst Figure 7.10 is the output image generated with an edge threshold of 40. The original image was a 512x512 pixel 24-bit colour optical fringe pattern, converted to 8-bit grey-scale using XC6200ADS tools. Within the output image (*Figure 7.10*) 99,475 pixels (represented in black) were greater than the threshold value.

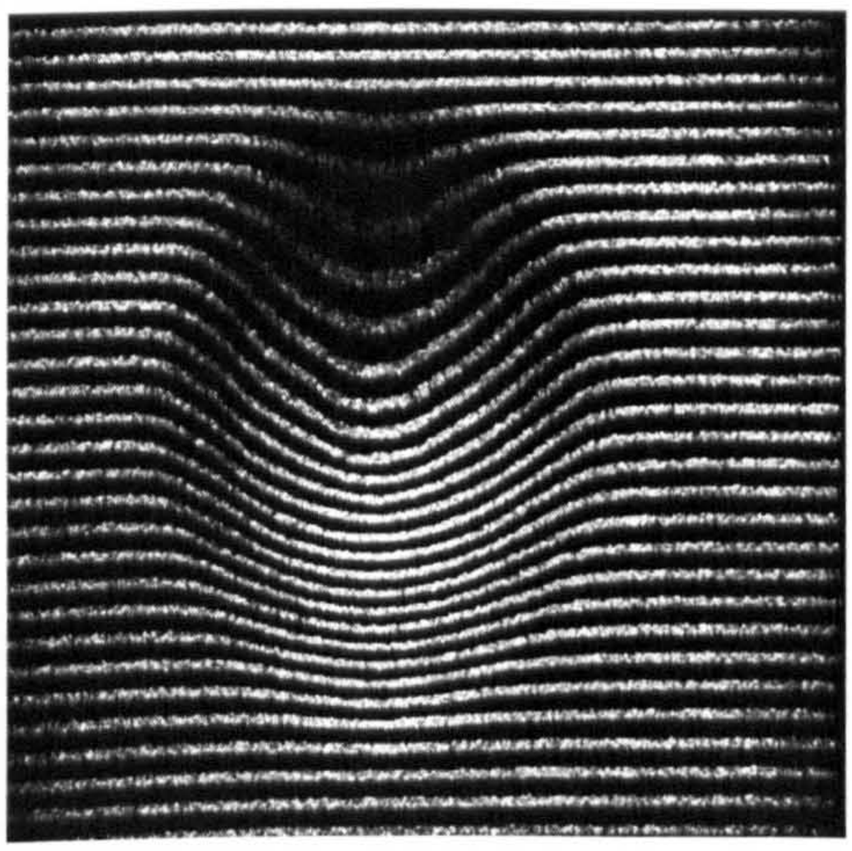


Figure 7.9 Original Image

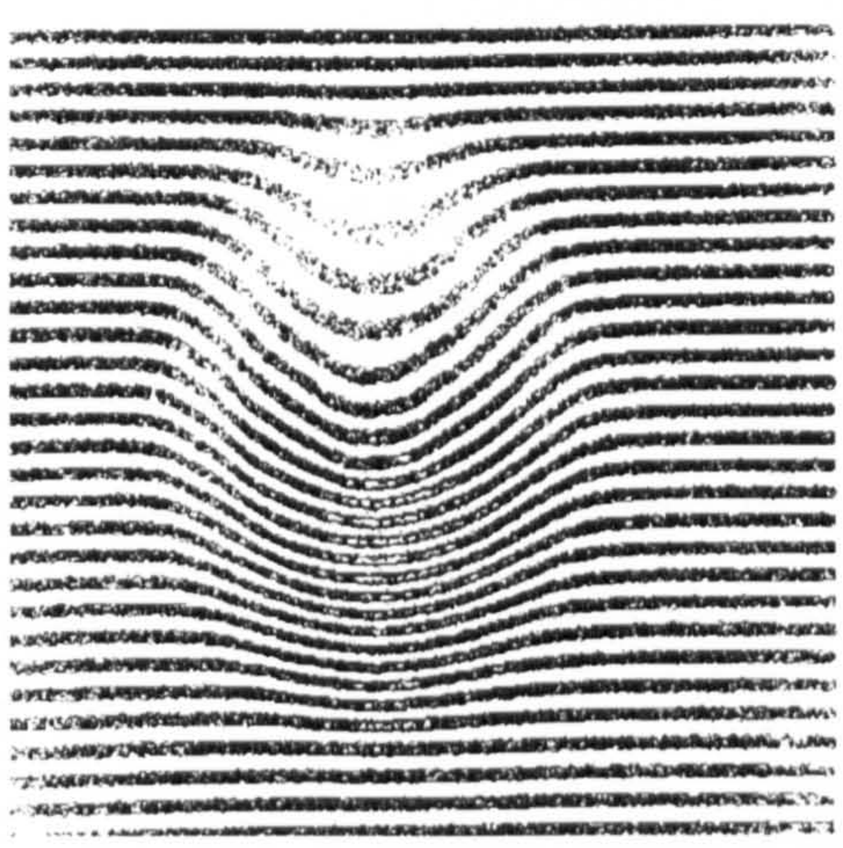


Figure 7.10 Roberts Operator Output

7.5.2 Roberts Operator Hardware Implementation

The Roberts Cross edge detector algorithm was implemented within a XC6264 FPGA using a 9-bit twos-complement bit-slice design. The block diagram of this hardware is shown in Figure 7.11. The design was constructed using arithmetic hardware developed in *Chapter-4*.

The design has four inputs *P1, P2, P3* and *P4* relating to the pixel position in Figure 7.8, and two outputs. Output *Sum* is the Roberts gradient (G) prior to quantisisation, with the threshold value determine by the content of a register within the design. The detection of an edge is indicated by signal *Edge*.

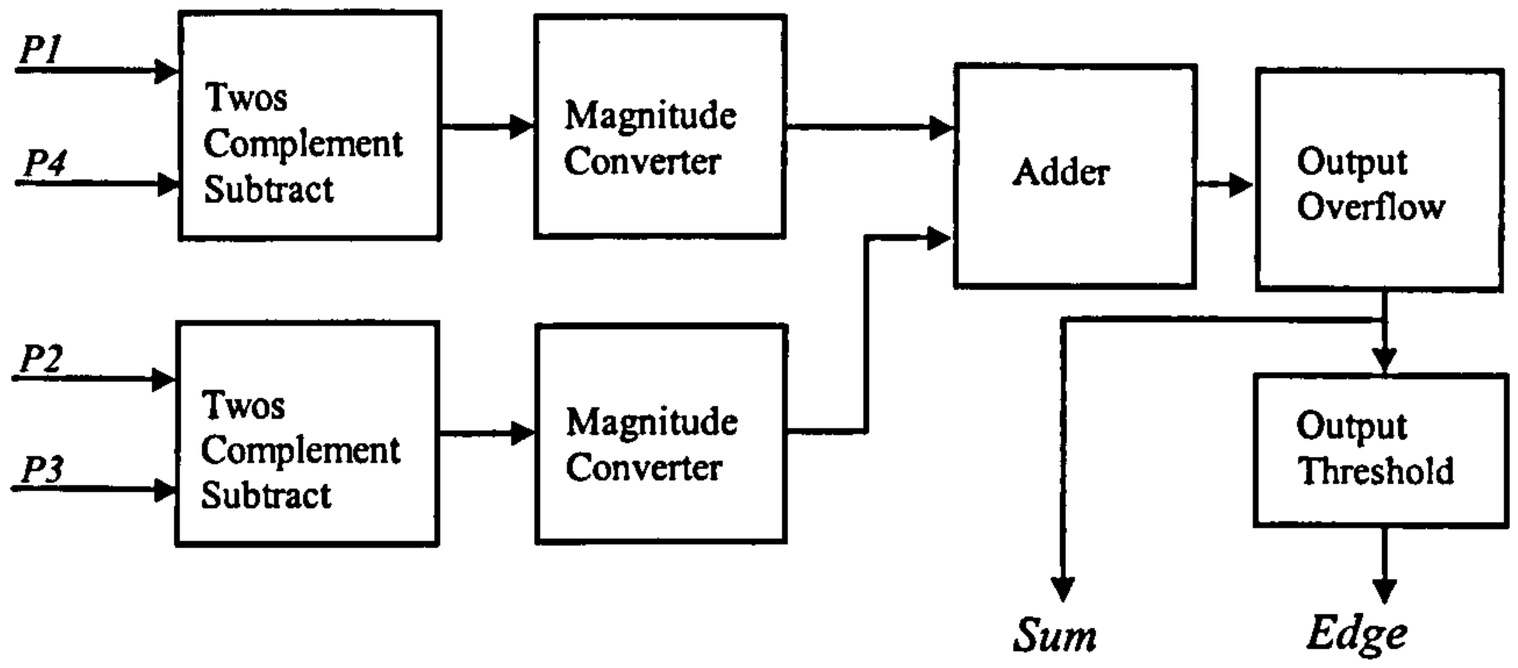


Figure 7.11 XC6264 Grey-scale Roberts Operator Hardware Implementation

The design functioned by first performing the subtractions operations within Equation 7.3. If the results generated by these operations were negative, they were then converted back to a positive magnitude. The addition operation was then calculated with the output magnitude limited to a value of 255 (8-bit grey-scale representation). Edge detection was performed through subtracting this value (*sum*) from the threshold value. If a negative result was generated an edge had been detected.

The complexity of the design could be reduced if monochrome (binary) input images were used. Within such images, pixel values were represented using one-bit data. Roberts operator hardware can therefore be simplified to three logic gates as shown in Figure 7.12.

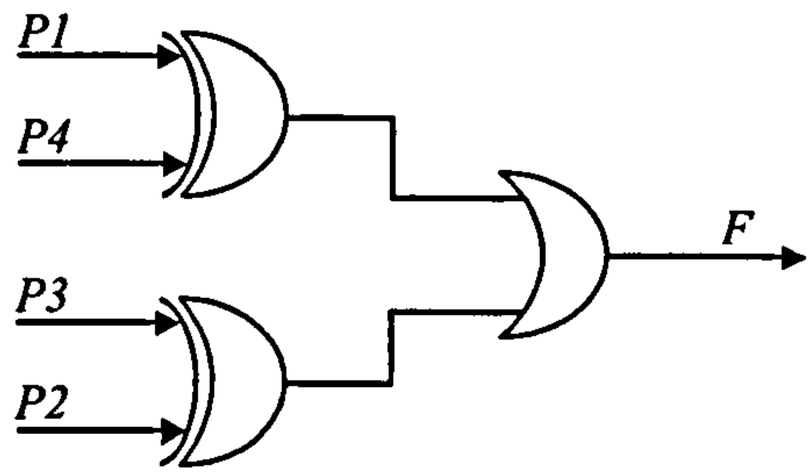


Figure 7.12 XC6264 Monochrome Roberts Operator Hardware Implementation

The implementation characteristics of each configuration are listed in Table 7.5. The results indicated that the binary Roberts operator had greater throughput than the grey-scale version. This was expected when considering the complexity of each operation.

	Number of CLCs	Maximum Frequency
Gray-scale	173	6.704MHz
Monochrome	3	62.697MHz

Table 7.7 XC6264 Grey-scale and Monochrome Operator Characteristics

To compare the output response of both grey-scale and binary Roberts operator implementations, a common input image (*Figure 7.9*) was used. This image was converted to binary before the monochrome operation commenced automatically within the XC6200ADS; A hardware version of this operation was also developed during the project.

The XC6264 output images obtained for an 8-bit grey-scale and monochrome Roberts operators are shown in Figure 7.13 and Figure 7.14. The monochrome implementation used a binary threshold of 130 prior to applying the Roberts operator, whilst the Grey-scale version had an edge threshold of 40.

Inspection of the images revealed that the 8-bit operator generated superior edges compared to the binary version. This deficit was expected and could be adjusted through employing histogram shifting prior to converting from grey-scale to monochrome image. Interpretation of the output generated however is dependant upon human visual perception.

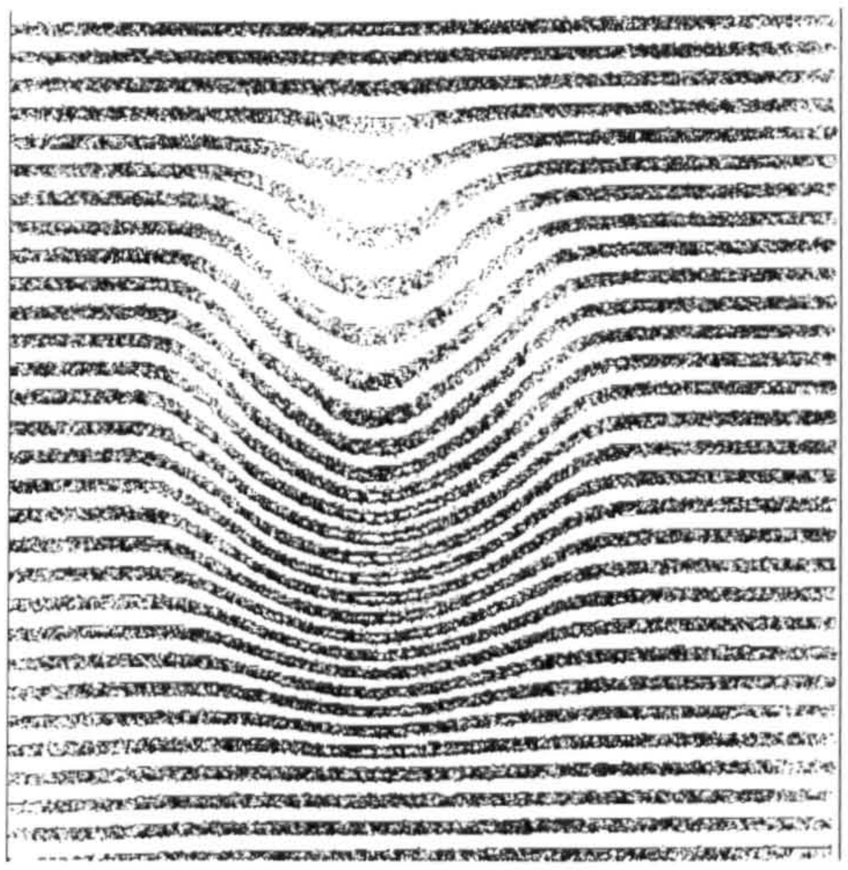


Figure 7.13 Grey-scale Output

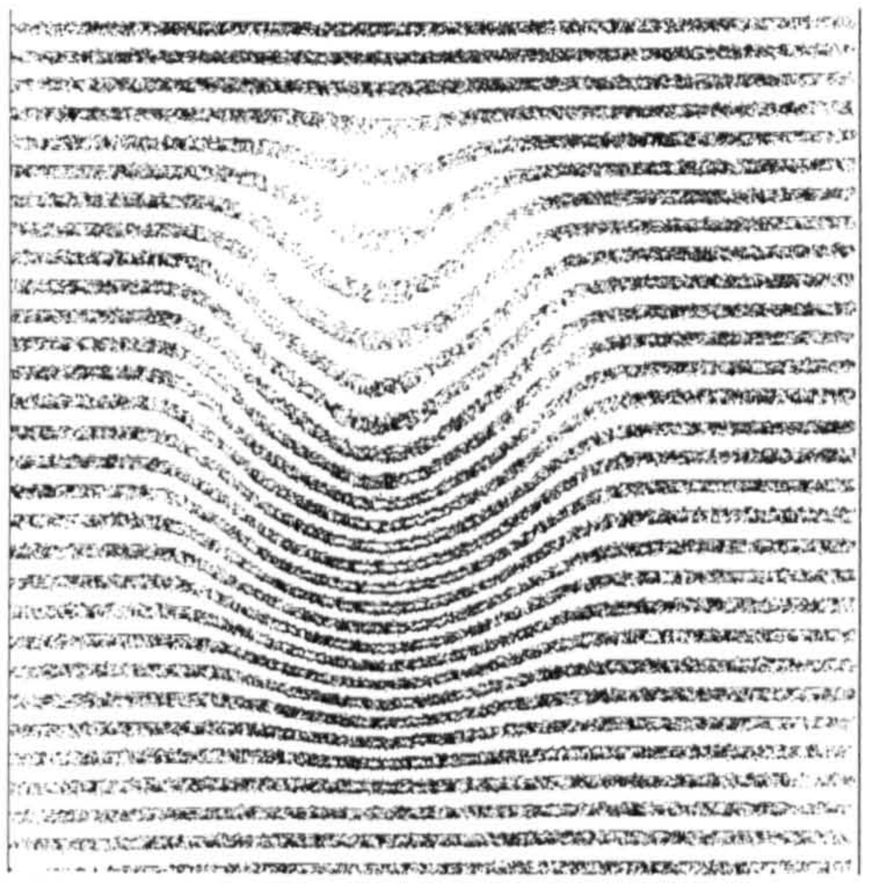


Figure 7.14 Monochrome Output

7.5.3 Roberts Operator Routing-Hub Integration

The implementation of the Roberts Cross operator within the routing-hub is shown in Figure 7.15. Data was communicated and controlled through the secondary crossbar switches enabling and disabling Roberts operator hardware as required. Input image pixels were written through comport-12 in one C40 word. Within the Roberts operator hardware a FIFO four-bytes deep was used to separate and apply each individual input pixel (*P1-P4*) to the processing elements.

Edge detection was determined using comport-9. Within the Roberts hardware implementation, signal *Edge* was represented by a single bit-value. Each bit within the word read by the C40 was set to the *Edge* value.

The Roberts threshold level was written through comport-8, and could be updated during system operation. One byte was required to implement this value but the method of applying input pixels constrained this value to be written within the fourth byte of the C40 word. This value could be controlled through observation of output signal *Sum* via comport-7.

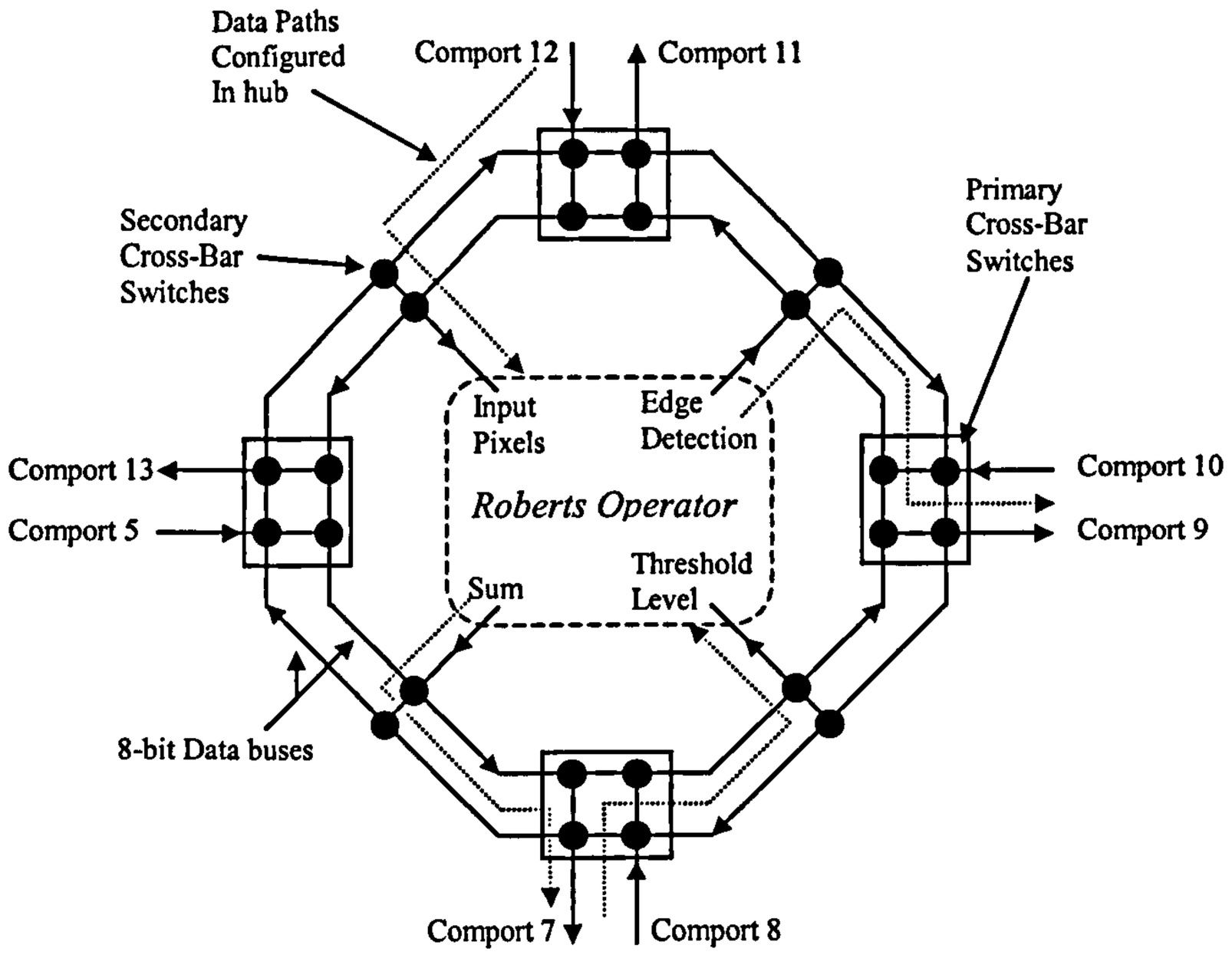


Figure 7.15 Router Hub Processing Element Configuration

To verify the operation of this topology, two TIM-40 nodes were attached. Similarly to *Section 7.4,* the internal routing topology of the TIM-40 motherboard was used to transfer test data between the secondary C40 and JTAG root nodes.

The primary node was connected to comport-12 *(P1, P2, P3 & P4*) and comport-9 (*Edge*), whilst the secondary node was connected to comport-7 (*Sum*) and comport-8 (*Threshold*). This topology allowed the threshold to be calculated and adapted during system operation. The feature could also be accomplished through a temporally partitioned version of the Roberts operator processing hardware.

Whilst evaluating the monochrome operator only one C40 node was required since no threshold value was used nor *sum* generated. Input data was written through comport-12 *(P1, P2, P3 & P4*) and the result read from comport-9 (*F*).

The optimum hardware characteristics obtained for each edge detector configuration are shown in Table 7.8. The volume of RTR configuration data generated (address/data pairs) was determined using the skeleton router hub architecture (*Figure 7.7*) as the configuration prior to the Roberts operator. Using XC6200ADS RTR, the configuration delay was measured externally.

Roberts Cross Configuration	Signal Delay	RTR Data Volume	Configuration Delay
Grey-scale Pixel Operator	303.24nsec	2951	1.1053sec
Binary Pixel Operator	290.64nsec	1562	0.627sec

Table 7.8 XC6264 Dynamic Router/Processor Hub Hardware Characteristics

During testing however, the XC6264 clock frequency was set to 2MHz. Inserting the grey-scale Roberts operator within the routing-hub actually increased operand throughput by 183.24kbytes/sec to 824.4kbytes/sec compared to a channel capacity of 641.16kbytes/sec, calculated when using both primary and secondary routing resources (*Section-7.3.1*). Prior to this experiment it was predicted that hub throughput would be reduced since the XC6264 configuration would be of greater complexity. Contradictory results however were obtained since hub secondary routing resource signal mappings (*Figure 7.5*) generated by XACT6000, differed for each hardware configuration. This further highlighted the limitations of XC6200 FPGA architecture and development tools.

Basic mechanisms for automated threshold calculation using the value of *Sum* were explored. If the value of *Sum* was small compared to the edge threshold then the kernel pixels (*P1-P4*) were in a region of low variation (no edges). However, if the value of sum was just less than the threshold, the active pixels were in a region of high frequency content and the threshold value would then be adjusted to detect the edge.

Using XC6200ADS software tools this concept was demonstrated using image 'lena'. Figure 7.16 shows the output of a Roberts Cross operator with an edge threshold of 30. Edge pixels were detected 23931 times within the input image. During the operation however, it was discovered that the *Sum* values of 12998 additional kernel operations

were below the edge threshold by a *deviation value* of 8 or less. The deviation value was chosen to be 8 through analyses of the *Sum* values generated. Exploring this concept however was beyond the scope of the project and required further investigation.

The Roberts Cross edge detector output using a primary threshold value of 30 and deviation value of 8 is shown in Figure 7.17. Figure 7.18 illustrates the output when threshold values of 38 excluding threshold deviations are used. With a threshold value of 30 and active deviation of 8, 36929 kernel operations detected edge pixels. In comparison a threshold value of 38 with no deviation detected 16398 edge pixels.

Figure 7.16 Edge Threshold 30 with no Deviation

Figure 7.17 Edge Threshold 30 with Deviation of 8

Figure 7.18 Edge Threshold 38 with no Deviation

7.6 Summary

This chapter has presented two methods by which dynamic hardware can be integrated into an existing multiprocessor topology to enhance system operation. Through inclusion of a dynamic routing-hub using dynamically and temporally partitioned routing structures, inter-node bandwidth can be adapted during system operation to speed-up operand transfer between nodes.

A dynamic hub has the potential to increases application diversity and system throughput, by enabling the structure of the processing topology to be adapted for optimal efficiently during each phase of an application. Through dynamic configuration the routing-hub could appear as an additional processing node during periods of limited operand transfer.

Expanding this idea further, a fine-grain pre-processing function (Roberts Cross) has been inserted into data-channels within the routing hub, with the aim to overlap operand transfer and computation overheads. This has increased system performance by offering 'front-end' fine grain processing and un-expectantly, increased data transfer rates.

The limited performance XC6200 based hardware developed to explore these concepts has not resulted in any increase in C40 MIMD performance. However they have

demonstrated the potential performance gains obtained from these ideas, and have ratified development strategies used.

Chapter 8

Conclusions

The objective of the work presented in this thesis has been an investigation into the integration of dynamic hardware resources within a DSP based multiprocessor architecture. This has been accomplished through implementing RTR hardware within a custom designed dynamic hardware development platform (XC6200DS). The outcome of this work has resulted in the construction of custom dynamic hardware and software tools, allowing the development of three novel aspects within configurable computing technology. These concepts have addressed dynamic hardware application development (BinDCT), dynamic coprocessor operation (self-configuration controller), and RTR routing-hub integration within multiprocessor architectures (routing-hub, Roberts Cross operators).

Limitations encountered within dynamic FPGA development tools have constricted the progress of this work to focus primarily on developing efficient RTR design principles, applications and dynamic operation, rather than increasing the raw operand throughput of the original parallel processing architecture.

Through exploiting redundant properties within the BinDCT algorithm during run-time, one and two-dimensional transform operations have been developed. Compared to static XC6200 FPGA implementation, dynamic hardware operation has increased operand throughput from 9.26 to 18.52 kBinDCT ops/sec per one-dimensional operation, improved the inherent DC coding-gain, resulting in increased accuracy in approximating true DCT operation.

Dynamic BinDCT operation was realised using a temporally partitioned C40 DSP XC6200 fixed-point dynamic coprocessor application (*Chapter-6*). To perform this operation a novel configuration controlled mechanism known as the self-configuration controller was developed. This concept enabled RTR to be performed without user

intervention, instead being instigated by the C40 DSP or XC6200 coprocessor function itself.

Through adapting the XC6200 coprocessors configuration during run-time, the throughput of two-dimensional BinDCT operations was increased by a factor of two for 8x8 pixel tiles possessing limited frequency contents. Increases in operand throughput however were masked by the RTR delay incurred. This factor was XC6200 FPGA specific and would decrease dramatically through improved dynamic semiconductor technologies. The implementations of the BinDCT algorithm using temporally portioned dynamic hardware were novel concepts.

The integration of dynamic coprocessors resources within the DSP multiprocessor architecture demonstrated how operand throughput could be increased through using re-usable application-specific hardware. The topology created provided each processing node with a hardware resource that could be configured and optimised to accelerate each computation during system operation. Using this technology each node exhibited virtual hardware capabilities.

The insertion of a dynamic routing hub within the TIM-40 multiprocessor communication topology has also revealed aspects of system operation that can be accelerated through dynamic hardware implementation (*Chapter-7*). Prior to inserting the routing hub, node operand transfer bandwidths were fixed during system operation. Through incorporating dynamic hardware, inter-processor bandwidths could be adapted during system operation and accelerate data transfers. Even with the limited bandwidth restrictions imposed through using XC6200 FPGA based hardware, the routing hub developed proved this aspect viable.

In constructing dynamic routing hardware, a trade-off between communication channel bandwidth and RTR delay (configuration data volume) existed. Although this factor appeared XC6200 FPGA specific, the development methodologies demonstrated during routing-hub construction have contributed to combating and rectifying this problem.

Incorporating elements within FPGA architectures such as predefined bus routes and dedicated crossbar switches have been explored.

To investigate the ability of RTR system nodes to implement router and processing resources concurrently, the XC6264 routing-hub possessed a secondary 'user' configurable area within its skeleton architecture. This was a novel concept within parallel processor routing topologies. To expand this notation, local-type fine-grain image processing operators were inserted within data-paths of the routing-hub. The aim was to overlap computation and operand transfer overheads by performing simple functions upon the data whilst in transit.

A Roberts Cross edge detector hardware implementation was developed to explore this feature (*Section-7.5*). Although this hypothesis proved beneficial to system architecture, the throughput obtained was again restricted by XC6200 FPGA hardware limitations.

Through inserting reconfigurable hardware within an existing fixed processing topology, the potential benefits to system operation were demonstrated. The optimal exploitation of these factors was not obtainable due to limitations imposed by the dynamic media.

The XC6200 FPGAs and development tools purchased were supplied for research purposes only. The effects of this restriction were visible through inefficient development tools and poor hardware operating characterises when compared to current FPGA architectures. The XC6200 FPGA family were used as the dynamic resources since no other suitable RTR devices were commercially available.

If developed commercially, the XC6200 family would have matured with gate capacity, operating frequency, dynamic configuration performance and quality of development tools. However, no industry standard dynamic applications existed, therefore demand for these semiconductors were low. It is hoped that application concepts demonstrated within this thesis may address this issue.

To exploit fully the benefits temporal partitioned hardware can offer, configuration delays must be reduced. This can be accomplished through advancements in RTR configuration mechanisms, and increases in reconfiguration granularity. The fine-grain granularity of the XC6200 requires each logic gate to be configured independently, generating large configuration files. If RTR could be performed on larger configuration tiles such as ALU units rather than individual gates, the volume of configuration data required would be reduced, hence configuration delay reduced.

Whilst constructing dynamic hardware, it became evident that in-circuit verification of dynamic hardware configurations was non-existent. This limitation was apparent for both functional testing and ensuring RTR had been successful. To address these problems in-circuit hardware verification methods were developed using the XC6200 FastMAP™ interface (*Section-4.1*). Although they proved reliable in operation, these methods were inadequate and poor compared to existing hardware design standards such as IEEE JTAG Boundary Scan [79].

The design philosophy used whilst developing dynamic hardware was to minimise differences between consecutive configurations. Although these techniques were XC6200 FPGA specific, it was evident that hardware designed for minimal RTR update was beneficial. RTR overheads and in-circuit hardware verification requirements would be reduced, and the design would exhibit greater operational reliability since fewer architectural changes occurred between successive configurations.

The aims of the project have been achieved, with three aspects of system operation enhanced through dynamic hardware. The potential performance benefits gained through using reconfigurable logic have been demonstrated. Before maximising these goals however, advancements must be made within configurable device architectures, software development tools, design strategies and in-circuit verification methods.

Progressions of these tasks has commenced within academia, and now starting to occur in industry [46] [47] [48]. The potential performance benefits offered by this technology

will be more apparent through further application development and semiconductor fabrication technologies reaching the limits of Moores law. Achieving high operand-throughputs would therefore show greater dependence upon efficient hardware implementation, rather than increased clock frequencies.

Through the inclusion of dynamic hardware resources within a traditional instruction-set parallel processing topology, the potential for increased application diversity and greater processing capacity has been demonstrated. The goals have been reached through exploitation of the concepts of virtual hardware and temporal application development.

Chapter 9

Recommendations For Future Research

Introduction

To advance the development and facilitate the inclusion of dynamic hardware within industrial applications, problem areas highlighted within *Chapter-8* must be rectified. The recommendations addressing these issues are divided in three categories. These are dynamic hardware technologies (*Section-9.1*), XC6200DS operation (*Section-9.2*) and dynamic hardware application development (*Section-9.3*).

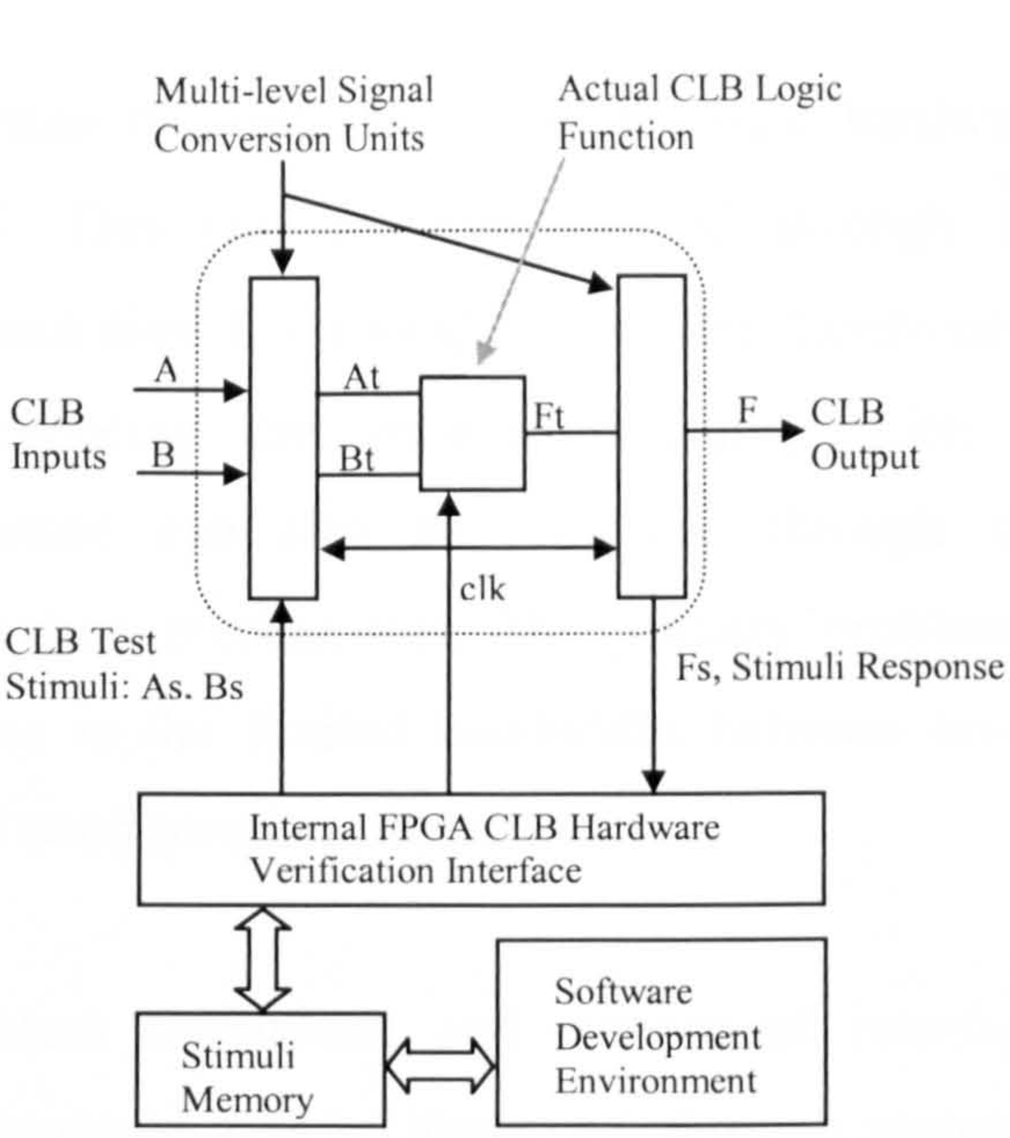
9.1 Configurable Logic Technology

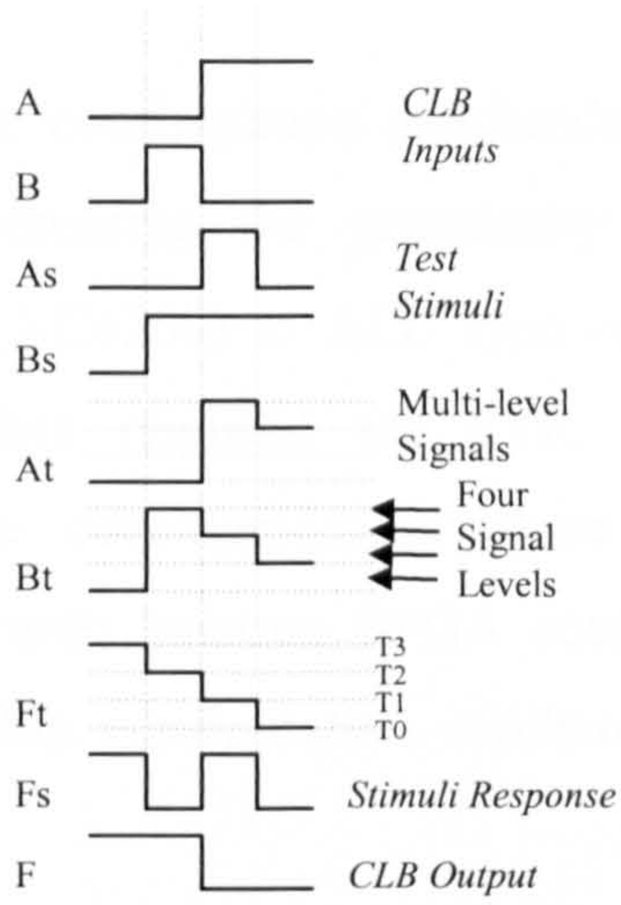
Whilst developing RTR hardware, it was evident that in-circuit test mechanisms for dynamic hardware did not exist. Limited hardware verification was only possible through using the XC6200 FPGA FastMAP™ interface. Improvements must be made to this aspect of hardware development if dynamic configuration is to be accepted as an industry standard.

Dynamic in-circuit verification must be present within a design to ensure that each run-time configuration used functions correctly. This task must be accomplished during run-time without inhibiting system throughput. To achieve this function, a multi-level real-time in-circuit hardware verification method, similar to the JTAG chain operation is proposed.

Within this concept hardware verification and operand throughput would occur concurrently through the use of four-level stimuli within FPGA CLBs as illustrated in Figure 9.1. The proposed FPGA architecture would consists of a dedicated on-chip CLB hardware verification interface unit, coupled to a (on-chip/external) memory containing specimen input and output test stimuli. The specimen results contained within the memory would be generated using software development tools prior to system

operation. During system operation, input test stimuli would be read from memory through the hardware verification interface and applied to the respective FPGA CLB.





System Waveforms

Figure 9.1 Run-Time Dynamic Hardware Verification Mechanisms

A	As	At
0	0	T0
0	1	T1
1	0	T2
1	1	T3

Where: T0-T3 correspond to signal levels in Figure 9.1

Table 9.1 Binary to Multiple Threshold Conversion Table

Through combing both the CLBs input data and test stimuli within a multi-level signal conversion unit, a four-level signal response is generated as shown in Table 9.1, where *A* is a CLB binary input, *As* the test stimuli, and *At* is the resultant multi-threshold signal. This response is then separated in to actual and test stimuli CLB outputs using a complementary multi-level signal converter. The FPGAs hardware verification unit would then compare the CLBs output test stimuli against pre-defined specimen results (located within the stimuli memory). If errors were detected remedial action would be taken including reloading defective CLBs or halting system operation and generating an error signal. Stimuli responses generated could also be transferred to and evaluated by

software tools through the stimuli memory software development environment interface.

To increase the performance of dynamic hardware configuration overheads must be reduced. This can be accomplished through increasing the granularity of RTR component tiles from single logic gate functions (XC6200) to ALU type operations, hence reducing the volume of configuration data required for RTR. Dynamic performance can also be improved through the development of more efficient configuration mechanisms. The primary problem with existing FPGA configuration interfaces is the limited bandwidth between on-chip configuration mechanisms and external configuration stores.

The system throughput and component interfaces between temporally partitioned hardware could also be improved, through incorporating self-timed design techniques within dynamic FPGA operation. For these concepts to be realised further investigation into the subject area is required.

Evolvable hardware has been toted as the solution for developing ever-more efficient digital designs. Presently, evolvable hardware configurations are generated by manipulation of FPGA configuration data sequences, updated through examining the FPGAs output compared to the desired hardware function. Through each evolutionary cycle, the FPGAs configuration stream is modified to adapt the FPGAs output to mimic more closely the required function.

The function and mechanisms of FPGA hardware configurations used within the evolutionary cycle do not follow any traditional synchronous digital design techniques. Instead features such as FPGA signal propagation delays and silicon electro-magnetic properties are utilised within the hardware's function and evolution. These factors however are device dependant and can vary with temperature and operating frequency. To address this issue, CLB configurations should be used as the evolutionary building

blocks, with updates of FPGA configuration performed using structured CLB configurations. This is in contrast to the present method of randomly modifying FPGA configuration bits, and then determining the effect this has had upon system output [63].

9.2 XC6200DS Operation

The XC6200DS was conceived as a multi-purpose development platform for RTR hardware. The system hardware and software tools were designed for easy operation, efficient construction and versatility, rather than raw performance.

To improve the XC6200ADS, the existing user interface should be replaced with a Windows ™ style menu GUI. The structure of the software also needs to be improved since it has continually evolved throughout the project. Its operation has now reached a standardised format, and therefore could be rewritten and optimised to improve operational performance.

The XC6200DS hardware design could be improved by replacing the external self-configuration RAM module (connected to XC6200DS using 40-pin IDC cable) with onboard RAM. This would improve the reliability of the self-configuration operation since the electrical noise encountered during configuration data transfer would be reduced.

To increase the bandwidth between the FastMAP™ interface and host computer, the XC6200DS should be redesigned as a PCI type peripheral or newer video interface standard, having 32-bit instead of 8-bit internal architecture. Inclusion of an on-board digital camera interface would also be beneficial to application development.

Further, modifications to the TIM-40 expansion port needs to be assessed. Although this proved operational, a more suitable construction method is required.

9.3 Application Development

The XC6264 coprocessor implementation of the BinDCT has shown how RTR can be used to improve the operand throughput, loss-less DC coding gain, and DCT approximation accuracy, compared to static hardware configurations. Work however is required to enhance the existing mechanisms used to determine the optimal BinDCT configuration for each pixel kernel. The present mechanism functions by applying each BinDCT configuration sequentially, and then calculating, which has generated the greatest inherent DC coding gain. Although it proved reliable in operation, this technique requires additional processing overheads that must be performed prior to BinDCT dynamic operation. One solution to this problem is to examine the variation of data within an input sequence block during computation, to determine if it has high or low frequency contents.

Experiments conducted have only assessed BinDCT operation using configurations BinDCT-C1 and BinDCT-C9. Further work is required to determine if including configurations BinDCT-C2 to BinDCT-C8 enhances dynamic operation.

To explore further the benefits of implementing BinDCT hardware within real applications such as JPEG compression, higher operand throughputs must be obtained. This is possible through implementing BinDCT hardware using parallel bit-wise implementation techniques instead of serial methods used within the XC6264 design. Before this can occur however the logic capacity of dynamic FPGA technologies must increase.

The concept of a multiprocessor dynamic routing hub can be advanced through developing automated system configuration strategies. The optimal configuration of the routing hub could be determined through analysing the bandwidth of data transfers and system bottlenecks occurring within the multiprocessor topology. Further, the concept of Roberts Cross edge detector threshold deviation should be explored in greater detail.

The final recommendation is to investigate the effects configurable logic could have on product design and operation. Design life spans could benefit from using configurable logic, and not just necessarily dynamic hardware.

Within electronic manufacturing, the lifespan of evolving commodity products such as mobile phones is limited. Such product designs become obsolete as new developments occur and consumers upgrade their hardware to be at the cutting-edge of technology. If products such as these were developed using reconfigurable hardware, users could upgrade to the latest protocols by downloading new configurations. This would help improve the lifespan of designs.

A further product concept is that of a multi purpose base-unit that a consumer reuses to perform multiple applications. Consumers would purchase each function as required, which are stored locally within a non-volatile configuration memory. Dependant upon the application required by the user, the appropriate hardware function would then be configured within in the base-unit.

The two concepts outlined will expand on the techniques of remotely generating, then downloading FPGA configuring data using the Internet known as *Internet Reconfigurable Logic* (IRL), as practiced today by Xilinx [82].

References

[1] J.D.Mellot, M.Lewis, F.Taylor, P.Coffield, "ASAP - A 2D DFT VLSI Processor and Architecture", *Proceedings IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 3280-3283, Atlanta, U.S.A., 7th-10th May 1996.*

[2] S.P.Kshirsagar, "High Speed Image-processing System Using Parallel DSPs", *Ph.D. Thesis, LJMU 1994.*

[3] J.E.Volder, "The CORDIC Trigonometric Computing Technique", *IRE Transactions Electronic Computing, Vol. EC-8, pp. 330-3345, 1959.*

[4] J.S.Walther, "A Unified Algorithm for Elementary Functions", *Proceedings Spring Joint Computer Conference, pp. 379-385, 1971.*

[5] G.Estrin, "Organisation of Computer Systems: The Fixed-Plus Variable Structure Computer", *Proceedings Western Joint Computer Conference, American Institute of Electrical Engineers, pp. 33-40, N.Y., U.S.A., 1960.*

[6] "MACH111SP CPLD Data Sheet", *Vantis Corporation, Publication Number 21120, Version B, 1997.*

[7] "54SX FPGA Family Data Sheet", *Actel Corporation, Version 3.0.1, May 2000.*

[8] "ACT™ 3 FPGA Family Data Sheet", *Actel Corporation, September 1997.*

[9] "XC4000E, XC4000X FPGA Family Data Sheet", *Xilinx, Version 1.6, May 1999.*

[10] "FLEX 10K Embedded Programmable Logic Family Data Sheet", *Altera, Publication Number A-DS-F10K, Version 4.02, May 2000.*

[11] "MPAA020 FPAA Advanced Information", *Motorola, Document Number MPAA020, April 1997.*

[12] "IQX FPID Family Data Sheet", *I-Cube, Version 6.2, February 2001.*

[13] S.Hauck, S.Burns, G.Borriello, B.Ebling, "An FPGA for Implementing Asynchronous Circuits", *IEEE Design and Test of Computers, Vol. 11, No. 3, pp. 60-69, 1994.*

[14] M.Valsilko, D.Ait-Boudaoud, "Optically Reconfigurable FPGAs: Is This a Future Trend ?", *Field Programmable Logic: Smart Applications, New Paradigms and Compilers, (FPL 96), Darmstadt, Germany, 23rd-25th September, 1996. Lecture Notes in Computer Science, Vol. 1142, pp. 270-279, Springer-Verlag, 1996.*

[15] J.Babb, R.Tessier, A.Argarwal, "Virtual Wires: Overcoming Pin Limitations in FPGA Based Logic Emulators", *IEEE Workshop on FPGAs for Custom Computing Machines, pp. 142-151, Napa, California, April 1993.*

[16] "XC3000 FPGA Family Data Sheet", *Xilinx, Version 3.1, November 1998.*

[17] H.Hogl, A.Kugel, J.Ludvig, R.Manner, K.Noffz, R,Zoz, "Enable++: A Second Generation FPGA Processor", *IEEE Symposium on FPGAs for Custom Computing Machines, pp. 45-43 ,Los Alamitos, U.S.A., April 1995.*

[18] J.E.Vuillemin, P.Bertin, D.Roncin, M.Shand, H.Toutati, P.Boucard, "Programmable Active Memories: Reconfigurable systems Come of Age", *IEEE Transactions on Very Large Scale Integration Systems, Vol. 4, No. 1, pp. 56-69, March 1996.*

[19] C.E.Cox, W.Ekkehard Blanz, "GANGLION – A Fast Field Programmable Gate Array Implementation of a Connectionist Classifier", *IEEE Journal of Solid State Circuits, Vol. 27, No. 3, pp. 288-299, March 1992.*

[20] D.Lewis, D.Galloway, D.Karchmer, D.J.Rose, P.Chow, J.Rose " The Transmogrifier: The University of Toronto Field-Programmable System", *Technical Report CSRI-30, Computer System Research Institute, University of Toronto, Toronto, Canada, June 1994.*

[21] S.Hauck, G.Borriello, C.Ebeling, "Springbok: A Rapid Prototyping System for Board Level Design", *ACM/SIGDA 2nd International Workshop on FPGAs, Berkley, U.S.A., 13th-15th February 1994.*

[22] L.Barroso, S.Iman, J.Jeong, K.Oner, K.Ramamurthy, M.Dubois, "RPM: A Rapid Prototyping Engine for Multiprocessor Systems", *IEEE Computer, pp. 26-34, February 1995.*

[23] S.Casselman, "Virtual Computing and the Virtual Computer", *IEEE Workshop: FPGAs for Custom Computing Machines, pp. 43-48, Napa U.S.A., 5th-7th April 1993.*

[24] P.M.Athanas, A.L.Abbot, "Real Rime Image Processing on a Custom Computing Platform", *IEEE Computer, pp. 16-24, February 1995, Vol.8 No.2.*

[25] J.R.Hauser, J.Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Co-processor", *Proceedings. IEEE Symposium Field Programmable Custom Computing Machines, (FCCM97), pp. 87-96, Napa Valley, U.S.A., 16th-18th April 1997.*

[26] I. Page, "Reconfigurable Processor Architectures", *Key Note Address, Heathrow PLD Conference, 1st April 1995, published in Microprocessors and Microsystems Special Issue on Co-Design, May 1996, Pub. Elsevier Science.*

[27] P.Athanas, H.Silverman, "Processor Reconfiguration Through Instruction Set Metamorphosis", *IEEE Computer, pp. 11-18, March 1993, Vol. 26, No.3.*

[28] M.Wirthlin, B.Hutchings, "The Nano Processor: a Low Resource Reconfigurable Processor", *IEEE Workshop on FPGAs for Custom Computing Machines, pp. 23-30, Los Alamitos, U.S.A., 11th April 1994.*

[29] S.Singh, P.Bellec, "Virtual Hardware for Graphics Applications Using FPGAs", *Field Programmable Custom Computing Machines, (FCCM94), pp.49-58, Napa Valley, U.S.A., April 1994.*

[30] A.DeHon. "Dynamically programmable gate Arrays: A Step Towards Increased Computational Density", *Fourth Canadian Workshop of Field Programmable Devices, (FPD 96), pp. 47-54, Toronto, Canada, 13th-14th May 1996.*

[31] "AT6000 Series Configuration Guide", *Atmel Corporation, Version 0436C, September 1999.*

[32] "XC6200 FPGA Family Data Sheet", *Xilinx, Version 1.10, April 1997.*

[33] E.Mirsky, A.DeHon, "Matrix: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources" *Proceeding. IEEE Symposium on FPGAs for Custom Computing Machines, pp. 157-166, Napa Valley, U.S.A., 17th-19th April 1996.*

[34] A.DeHon, M.Bolotski, T.F.Knigh, "Unifying FPGAs and SIMD Arrays, International *ACM/SIGDA Workshop on FPGAs (FPGA 94), pp. 1-10, Berkeley, U.S.A., 13th-15th February 1994.*

[35] J.Villasenor, B.Hutchings, "The Flexibility of Configurable Computing", *IEEE Signal Processing Magazine, pp. 68-84, Vol.15, September 1998.*

[36] S.Trimberger, D.Carberry, A.Johnson, J.Wong, "A Time Multiplexed FPGA", *Proceedings of IEEE Symposium on FPGA-Based Custom computing Machines, (FCCM 97), pp. 34-40, Napa Valley, U.S.A., April 1997.*

[37] CLAy Datasheet, *National Semiconductor Corporation, Santa Clara California, U.S.A., 1993.*

[38] Virtex II FPGA Family Data Sheet, *Xilinx, Publication Number DS03101, Ver. 1.5, April 2001.*

[39] S.Ludwig, "The Design of a Co-processor Board Using Xilinx's XC6200 FPGA -An Experience Report", *Proceedings 6th International Workshops on Field Programmable Logic and Applications, Springler-Verlag, Darmstact, Germany, 23rd-25th September 1996.*

[40] S,Hauck,T.W.Fry, M.M.Hoswler, J.P.Kao, "The Chimaera Reconfigurable Functional Unit", *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM 97), pp. 87-96, Napa Valley, U.S.A., April 1994.*

[41] B.K.Gunther, "Space 2 as a Reconfigurable Stream Processor", *Australasian Conference on Parallel and Real-Time Systems, pp. 286-297, Newcastle, Australia, September 1997.*

[42] M.J.Wirthlin, B.L.Hutchings, "DISC: The Dynamic Instruction Set Computer", *Proceedings of SPIE Conference: FPGAs for Fast Board Development and Reconfigurable Computing, pp. 92-103,Philadelphia, U.S.A., 25th-26th October 1995.*

[43] H.Singh, M.Lee, G.Lu, F.J.Kurdahi , N.Bagherzadeh, E.Chaves Filho, "Morphosys: An Integrated Reconfigurable System for Data Parallel Computation Intensive Applications", *IEEE Transactions of Computers, Vol. 49, No. 5, pp. 465-480, May 2000.*

[44] S.Gehring, S.Ludwig, "The Trianus System and its Application to Custom Computing", *Proceedings 6th International Workshops on Field Programmable Logic and Applications, Springler-Verlag, Darmstact Germany, 23rd-25th September 1996.*

[45] "E5 Configurable System-on-Chip (SOC) Data Sheet", *Triscend, Version 1.05, February 2001.*

[46] Star Bridge Systems Press release, *http://www.starbridgesystems.com, 5th February 1999.*

[47] "IBM and Xilinx Team to Create New Generation of Integrated Circuits", *IBM/Xilinx Joint Press Release", New York, U.S.A., July 25th 2000, http://www.xilinx.com/prs_rls/ibmpartner.htm.*

[48] "Excalibur Device Overview", *Altera, Publication A-DS-EXCARM, Ver.1.2, February 2001.*

[49] B.L.Hutchings, M.J.Wirthlin, "Implementation Approaches for Reconfigurable Logic Applications", *International Workshop on Field Programmable Logic and Applications, pp. 419-428, Oxford, England, 29th–31st August 1995.*

[50] M.J.Flynn, "Very High Speed Computing Systems", *Proceedings of the IEEE, Vol. 54, pp. 1901-1909, December 1966.*

[51] S.A.Guccione, M.J.Gonzalez, "Classification and Performance of reconfigurable Architectures", *International Workshop on Field Programmable Logic and Applications, pp. 439-448, Oxford, England, 29th–31st August 1995.*

[52] A.DeHon, "Comparing Computing Machines", *SPIE International Symposium on Voice, Video, and Data Communications, Conference No. 3526: 'Configurable Computing: Technology and Applications', pp.124-123, Boston, U.S.A., 1st-5th November 1998.*

[53] D.Smith, D.Bhatia, "RACE: Reconfigurable and Adaptive Computing Environment", *Lecture Notes in Computer Science, Vol. 1142, pp. 87–95, Springer-Verlag, 1996.*

[54] R.W.Hartenstein, M.Hertz, T.Hoffman, U.Nageldinger, "Using the Kress Array for Reconfigurable Computing", *SPIE International Symposium on Voice, Video, and Data Communications, Conference No. 3526: 'Configurable Computing: Technology and Applications', Boston, U.S.A., 1st-5th November 1998.*

[55] M.Nakkar, J.Harding, D,Schwartz, "Dynamically Programmable Cache", *SPIE International Symposium on Voice, Video, and Data Communications, Conference No. 3526: 'Configurable Computing: Technology and Applications', Boston, U.S.A., 1st-5th November 1998.*

[56] "FIPSOC™ Data Sheet", *SIDSA, San Francisco, U.S.A., 2001.*

[57] G.Mykleburst, J.G.Solheim, " RENNs – Utilizing a Reconfigurable Communication System", *Joint Conference on Information Sciences (JCIS'94), Duke University, U.S.A., November 1994.*

[58] C.Beaumont, "Using FPGAs as Control Support in MIMD Executions", *Field Programmable Logic and Applications, Springer-Verlag, pp. 44 –53, 1995.*

[59] J.Gosling, H.McGilton,"The JAVA™ Language Environment, A White Paper", *Sun Micro Systems, May 1996.*

[60] L.Rossen, "Ruby Algebra", *Designing Correct Circuits, Workshops in Computing, pp. 297-312, Oxford 1990, Pub. Springer-Verlag, 1990.*

[61] I.Page, "Constructing Hardware-Software Systems From a Single Description", *Journal of VLSI Signal Processing Systems for Signal Image and Video Technology, Vol. 12 No. 1, pp. 87-107, January 1996.*

[62] A.Thompson, I.Harvey, P.Husbands, "The Natural Way to Evolve Hardware", *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS 96), Vol. 4, pp. 37-40, 1996.*

[63] A. Thompson, "An Evolved Circuit, Intrinsic in Silicon, Entwined With Physics", *Proceedings International Conference on Evolvable Systems (ICES 96), Vol. 1259, pp. 390-405, Pub. Springer-Verlag, 1997.*

[64] "TMS320C4X Module Specification Version 0.232", *Texas Instruments, 1992.*

[65] "TMS320C4X User's Guide", *Texas Instruments, Publication Number 2564090-9721, Rev. A, May 1991.*

[66] "PaCE User's Manual", *Transtech Parallel Systems, Publication Number PAACM801, 1998.*

[67] J.Schewel, S.Casselman, "HOT User's Guide", *Virtual Computer Corporation, September 1997.*

[68] "Introduction to JTAG Boundary Scan", *Sun Microelectronics, White Paper, January 1997.*

[69] P.Pirsch, "Architectures for Digital Signal Processing", Pub. *John Wiley and Sons, ISBN: 0471971456.*

[70] IE.C.Ifeachor, B.W.Jervis ,"Digital Signal Processing", *Pub. Addison-Wesley ISBN: 020154413.*

[71] D.Van den Bout, "The Practical Xilinx Designer Lab Book", *Prentice Hall ISBN:013021617.*

[72] G.A.Baxes, "Digital Image Processing", *Pub. Prentice Hall ISBN:013214056.*

[73] W.Chen, C.Harrison-Smith, S.C.Fralick, "A Fast Computational Algorithm for the Discrete Cosine Transform", *IEE Transactions on Communications, pp. 1004-1009, Vol.25, No. 9, September 1977.*

[74] H.S.Hou, "A Fast Recursive Algorithm for Computing the Discrete Cosine Transform", *IEEE Trans. Acoustics, Speech and Signal Processing, Vol. Assp-035, pp. 1243-1245, December 1984.*

[75] E.Fieg, S.Winograd, "On the Multiplicative Complexity of Discrete Cosine Transform", *IEEE Trans. Information Theory, Vol.3, No.2, pp. 1387-13941, July 1992.*

[76] J. Liang, T. D. Tran, "Fast Multiplierless Approximations of the DCT with the Lifting Scheme", *IEEE Trans. on Signal Processing, Vol.49 No 12, pp. 3032-3054, December 2001.*

[77] G.J.Battaglia, "Mean Square Error", *AMP Journal of Technology, pp. 31-36, Vol.5, June 1996.*

[78]]D.Lewis, D.Galloway, M. van Ierssel, D.J.Rose, P.Chow, " The Transmogrifier-2: A One Million Gate Rapid Prototyping System", *ACM/SIGDA International Symposium. on FPGAs, FPGA-97, pp. 53-61, Monterey, U.S.A., 9th-11th February 1997.*

[79] IEEE,"*IEEE standard test access port and boundary-scan architecture", IEEE Std 1149.1-2001, IEEE Publication, 2001.*

[80] B.W.Arden, H Lee, "*Analysis of Chordal Ring Network". IEEE Transactions on Computers pp. 291-295, Vol.230, No.4, 1981.*

[81] J.Liberty, "Teach Yourself C++", *Pub. Sams Publishing, ISBN: 0672310708*

[82] "XC6200DS Tutorial 1", *Xilinx, Version 1.2, August 1997.*

[83] "Architecting Systems for Upgradability with IRL (Internet Reconfigurable Logic)", *Xilinx, Document Number XAPP412, Version 1.0, June 2001.*

Appendix I

Programmable Logic Device Technologies

Introduction

Existing PLDs can be fabricated using one of four mainstream technologies. These are *Fuse*, *Anti-Fuse*, *Floating-Gate* and *SRAM*. An overview and operational characteristics of each are described in *Sections I-1* to *I-4* respectively.

Appendix I-1 Metal Fuse Technology

Early PLD fabrication used fuse based programming technology. The fuse consisted of a metal strip connecting two signal routes, and when configured would appear as either a short or open circuit connection. Three typical fuse types are shown in Figure I.1. If a fuse was programmed to be an open circuit, a current source larger than the normal operating conditions had to be applied.

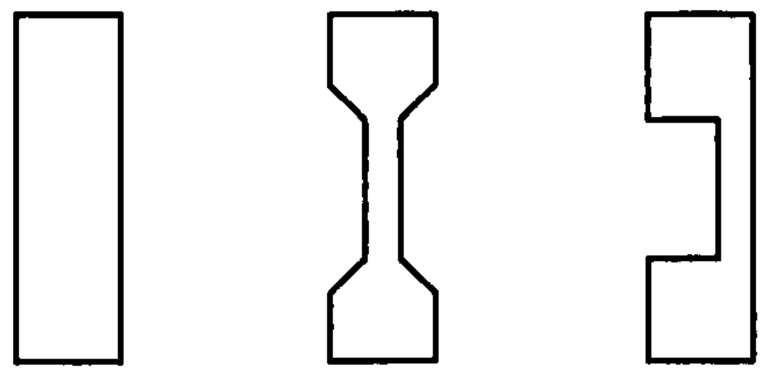


Figure I.1 PLD Metal Fuse Technology

Within the PLD architecture control circuitry (*Figure I.2*) was required to distinguish between programming and normal operating modes (via Zener diodes), and providing additional current (typically 50 -100 mA) to blow the metal fuses (using transistors).

Once a fuse had been made open circuit it could not be reverted back to a short circuit. Fuse based PLDs were therefore one-time programmable devices. This technology was non-volatile since the PLD retained its configuration after the supply voltage had been removed.

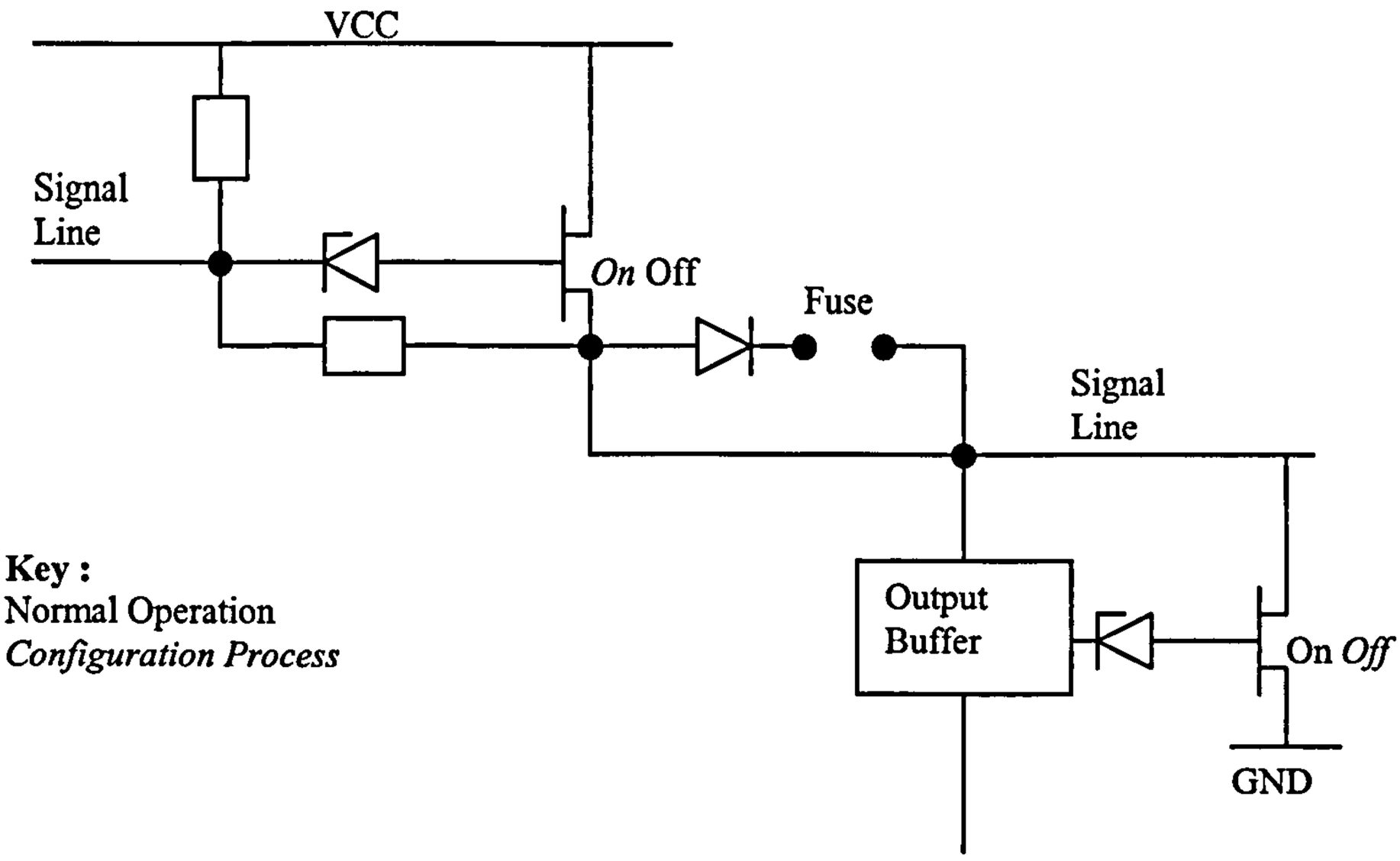


Figure I.2 PLD Configuration Control Circuit

Appendix I-2 Anti-Fuse Technology

Anti-fuse PLD technology exhibits similar operational characteristics to that of metal fuse devices. The difference is that an anti-fuse appears open circuit until it has been 'blown', and then it becomes a short circuit. This is the reverse operation of a metal fuse. Anti-fuse PLDs can be programmed only once and have a non-volatile configuration.

Figure I.3 illustrates the programmed and non-programmed states of an anti-fuse. An anti-fuse construction is similar to that of an MOS transistor, but with a dielectric inserted between the poly-silicon and the N+ diffusion region. Before configuration, the dielectric prevents current flow between the poly-silicon and the N+ diffusion region, thus the anti-fuse has a high resistance (appears open circuit).

Upon application of a programming voltage larger than the normal operational voltage, the dielectric melts, hence the anti-fuse has been 'blown'. This allows current flow

between the poly-silicon and the N+ diffusion region to occur (appears as a short circuit).

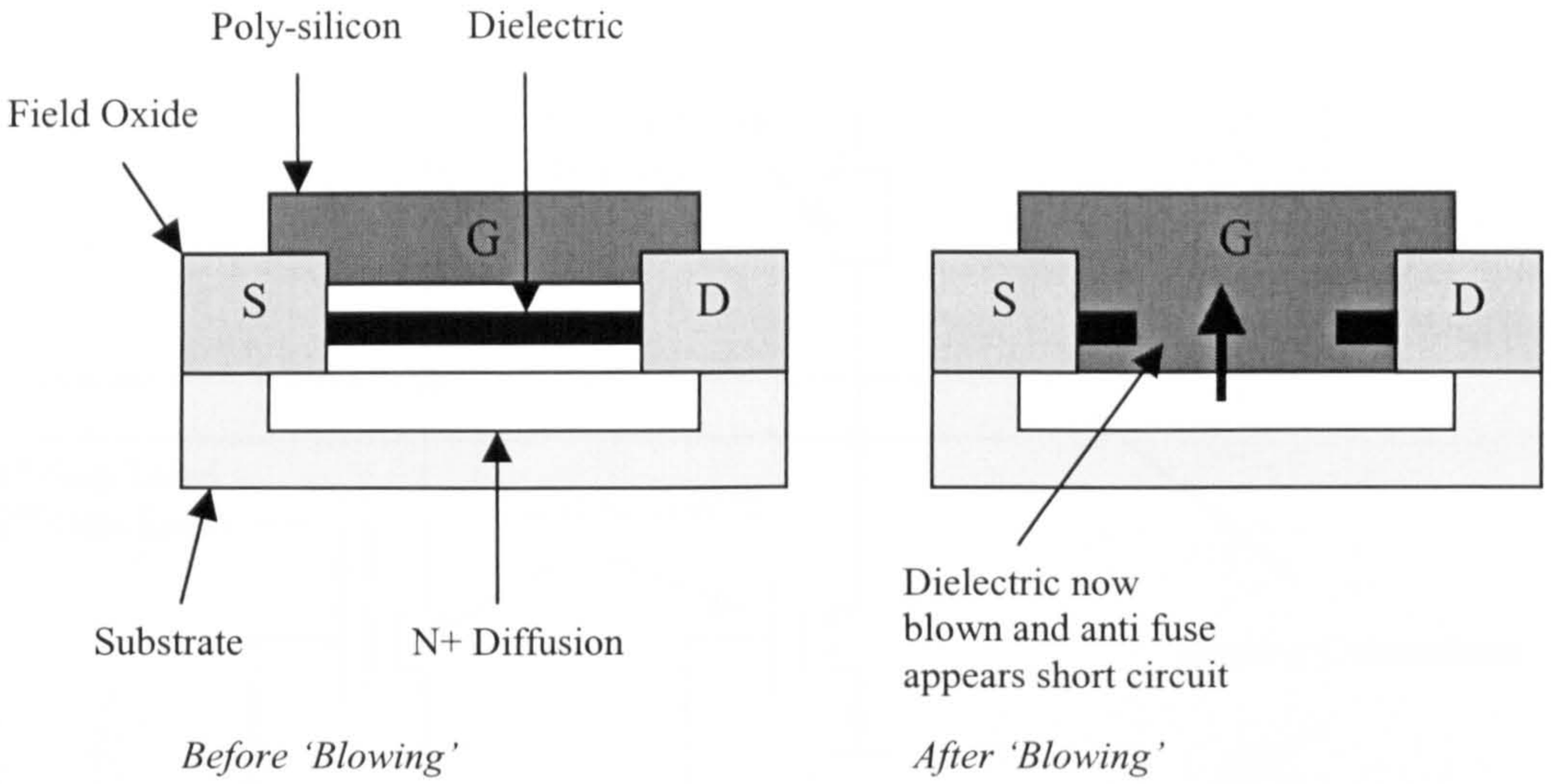


Figure I.3. PLD Anti Fuse Construction

Anti-fuse PLDs exhibited higher logic densities due to the smaller silicon footprint of anti-fuses compared to metal fuses. This technology is still used in FPGAs and CPLDs manufactured by vendors such as Actel and Quicklogic.

Appendix I-3 Floating-Gate Transistors

Metal and anti-fuse technologies were limited by the fact that they could only be programmed once. Further programming circuitry used up valuable silicon real-estate, hence reduced the gate capacity.

Reusable PLDs became viable through the introduction of floating-gate technology. Instead of using fuses, floating-gate technology incorporated transistors configured as pass-switches. By increasing the transistors gate threshold voltage above the supply voltage the operation of the transistor pass-switch could be disabled. This concept is shown in Figure I.4.

The application of a high potential between the gate and the drain regions of the transistors prevents the formation of a conduction channel, hence the transistor appears

open circuit (non-conductive). This is because the threshold level of the gate terminal has been increased to a value greater than the normal operating supply voltage

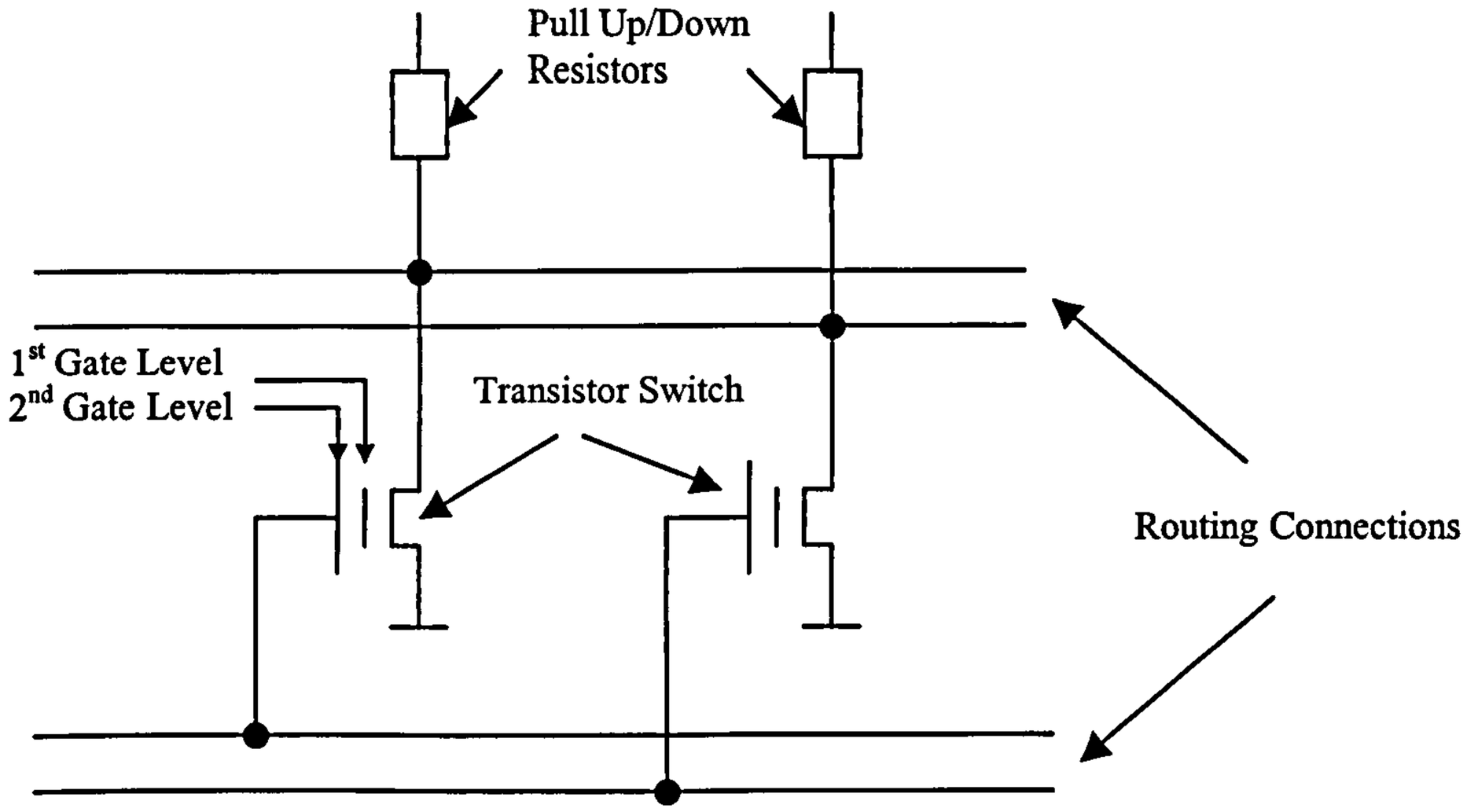


Figure I.4 Floating-Gate PLD Technology

The additional gate charge can be removed through exposing the gate region to *ultra-violet light* or using electrical methods. This provides a distinct advantage over previous technologies since floating-gate devices can be programmed multiple times. These two erasure methods are similar to that used in EPROM and EEPROM memory technologies. Examples of EPROM and EEPROM PLDs are Altera MAX5000 and Vantis MACH series respectively.

Appendix I-4 Static Random Access Memory

This programming technology uses the content of a SRAM memory to control the function of pass transistors (switches), multiplexers and memory LUTs. This concept is shown in Figure I.5.

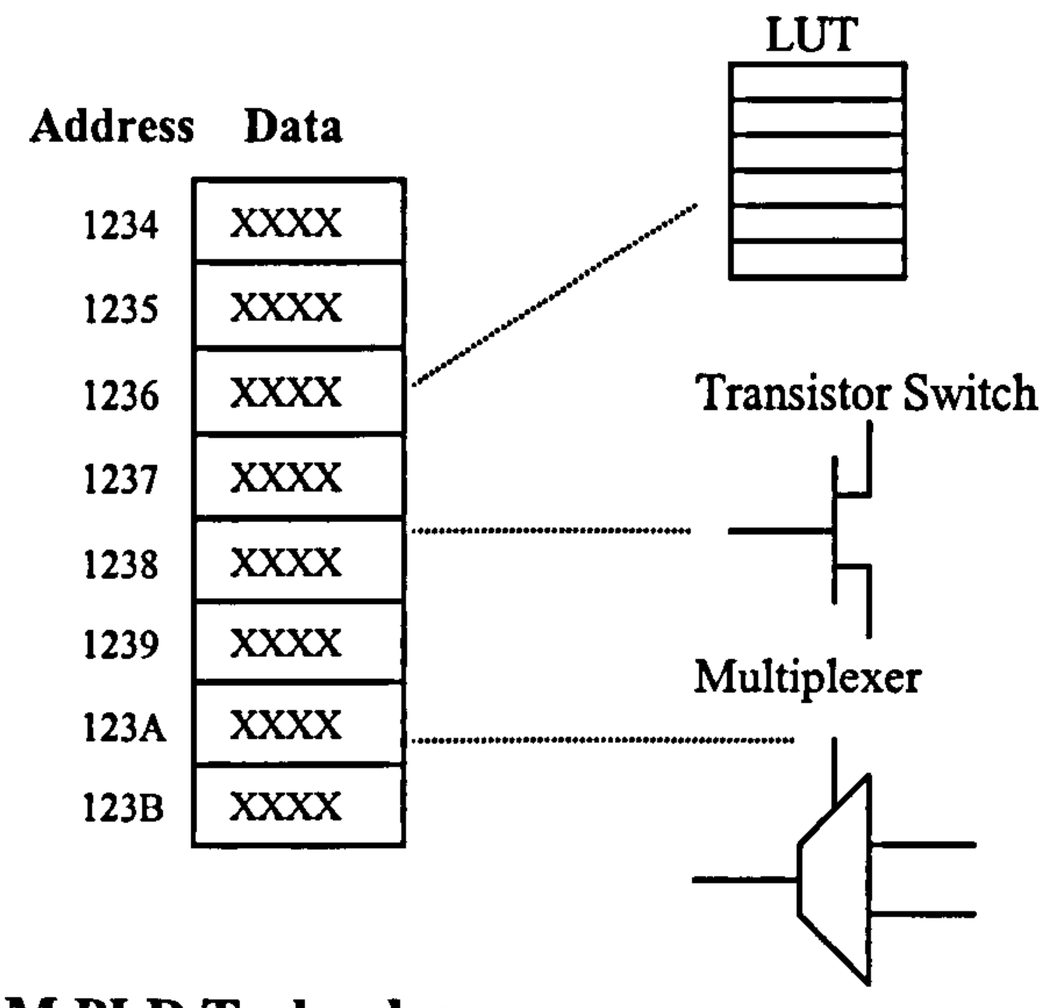


Figure I.5 SRAM PLD Technology

SRAM based PLDs are reconfigured by overwriting the contents of the configuration memory. The advantage of SRAM PLDs is that they can be configured much quicker than previous programming technologies. SRAM memory however is volatile and must always be configured upon power-up. Normally configuration data is stored within an external PROM.

A second disadvantage is that the implementation of SRAM requires large silicon real footprints when compared to floating gate and anti fuse technology. An example of SRAM based user programmable devices are the Xilinx XC4000 FPGA family.

Appendix II

Configurable Computer Architecture

Introduction

This section of the Appendix explains in greater detail the architecture and operation of configurable computing machines discussed in *Chapter-2*.

Appendix II-1 Transmogrifier-2

The *Transmogrifier-2* (TM-2) [78] was a second generation multiple FPGA based rapid prototyping system that could implement logic designs up to one-million logic gates in complexity. The system was modular and consisted of between one to sixteen TM-2 circuit boards, interconnected via a back plane to a host computer. For each design, prototype hardware was first manually partitioned upon the system and then automatically configured.

The outline architecture of a TM-2 board is shown in Figure II.1. Each board consisted of two Altera 10K50 FPGAs, dedicated local memory, hierarchical FPID based interconnection network, clock generation circuitry and housekeeping functions.

Designs implemented upon the TM-2 system were partitioned and configured amongst multiple FPGAs. Therefore a flexible and high bandwidth two-layer hierarchical crossbar FPGA interconnection topology was formed using I-Cube IQ320 FPIDs. This topology enabled FPGAs to share operands and memory resources upon each board, and at system level.

The lowest level of crossbar hierarchy was at board level. This is illustrated in Figure II.1, with the crossbar switch formed using FPID1, FPID2 and FPID3. The top level of this topology was at system level and formed through interconnecting FPID-4 of each board in the system via the back plane connector.

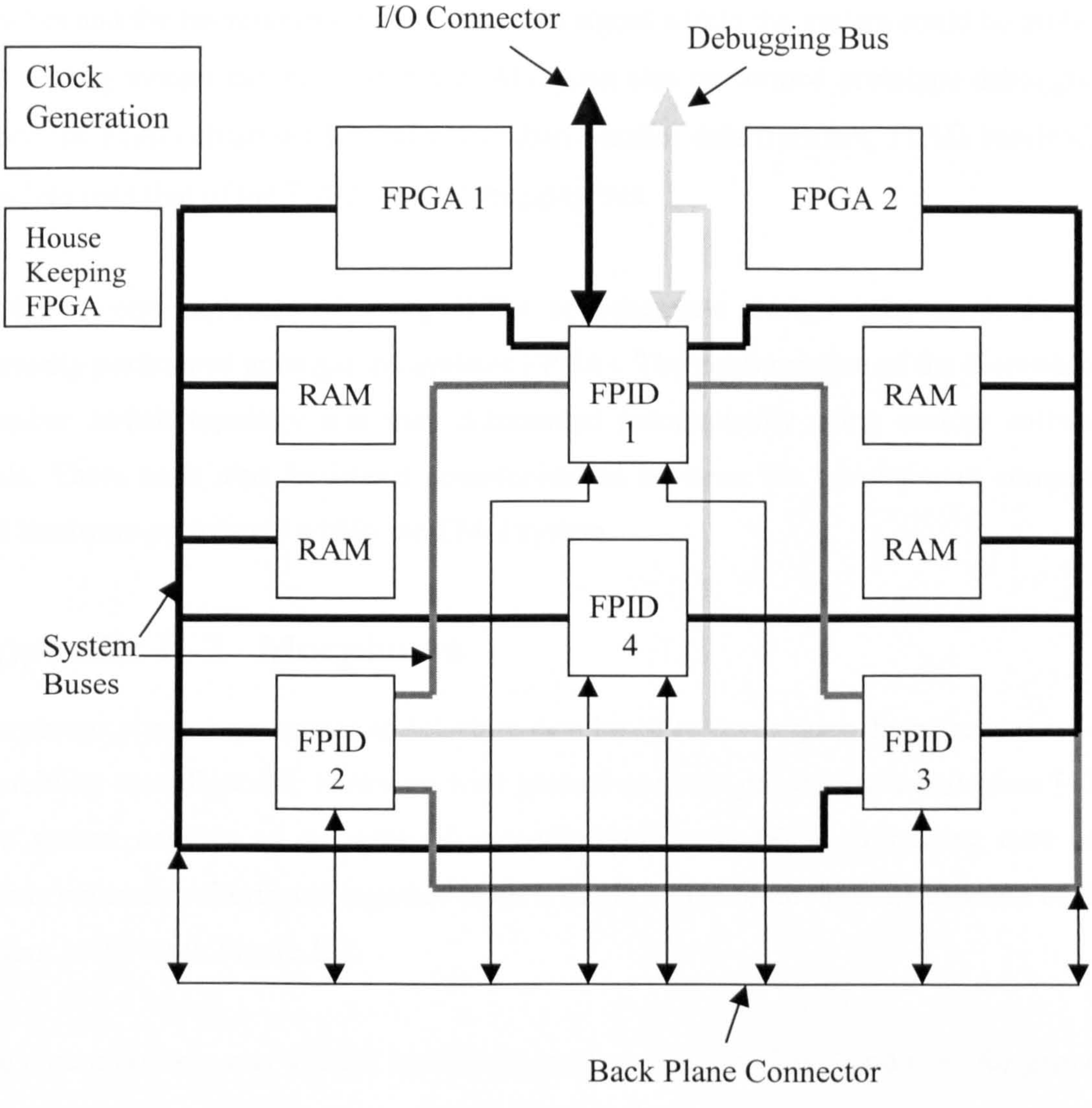


Figure II.1 Transmogrifier-2 Board Architecture

The housekeeping FPGA upon each TM-2 board supervised the downloading of configuration data to both FPIDs and FPGAs. It also detected short-circuits occurring between devices through monitoring device supply currents. Short-circuits could occur through errors introduced during the manual partitioning of designs, enabling multiple signals to drive a single net. When short-circuits were detected, the housekeeping FPGA disabled the output of the board and notified the host computer.

The TM-2 architecture was primarily designed to develop hardware and enable real-time debugging of prototype designs using a dedicated 32-bit bus. Through the action of this bus and the interconnection topology, any signal within the system could be probed. Connecting system devices within a JTAG chain also performed prototype debugging. Since the JTAG chain used serial rather than parallel data transfers, JTAG bandwidth was less than that of the TM-2 32-bit debugging bus.

Designs were implemented using HDLs or schematic design entry methods, but manually partitioned amongst the systems FPGAs. The configuration of the hierarchical crossbar switch topology was then determined automatically using custom software tools. These tools also facilitated communication between the development computer and hardware prototyped within the TM-2 system.

Appendix II-2 Morphosys

Morphosys was a coprocessor architecture developed to investigate the effectiveness of combining reconfigurable hardware with general-purpose processing architecture [43]. The system consists of an array of *reconfigurable cells* (RC), processing core and memory interface fabricated together upon a single silicon chip. The architecture of the system is shown in Figure II.2.

The core processor was a RISC type architecture called TinyRISC and used for general purpose operations and managing the RC array. During system operation, additional instructions were inserted into the core processor instruction-set to govern configuration of the RC array.

The configurable logic array within Morphosys was implemented using custom designed RCs. Compared to existing FPGA CLBs, the structure of an RC was coarser, with each containing an ALU, multiplier and register file. The configuration of an RC was determined using a word selected from a multiple-context configuration memory.

Using a global or a private context broadcast, the configuration of the whole array or individual RCs could be updated during run-time.

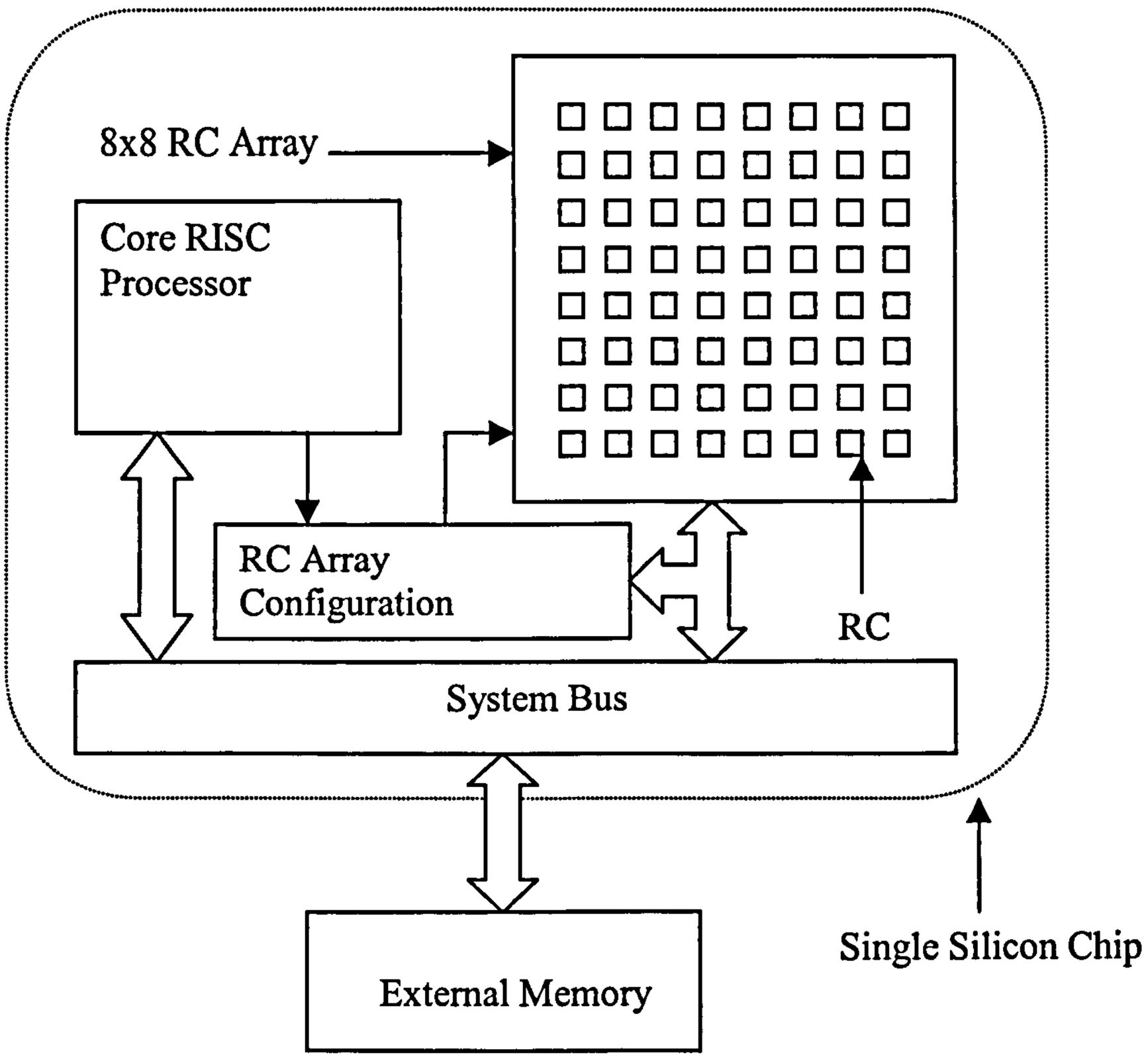


Figure II.2 Morphosys System architecture

Behavioural models for Morphosys were developed in both VHDL and C programming languages. Applications for Morphosys written in C were simulated using a custom designed simulation tool called *MuLate.* Designs implemented in the VHDL are simulated using a VHDL model of Morphosys called *MorphoSim* within QuickVHDL Simulation environment.

A Morphosys design consists of both instruction-set and configurable logic components. To map designs onto the RC array a custom development tool called *mView* was developed. The user inputted the function of each RC and the source and destination of

required operands. From this description the configuration of the RC array was generated.

To combine multiple RC array configurations and sequence context switching, software was developed that determined the most efficient sequence in which the context switches should occur. This tools inserted specific instructions within the core processor instruction-set to instigate context switches.

Currently, software is under construction that will allow system applications to be described in C. This software will then automatically map and generate the RC array configuration contexts and core processor instruction-set program. Applications mapped on to Morphosys have included video compression, automatic target recognition, and data encryption.

Appendix II-3 Splash-2

Splash-2 was a prominent example of second-generation reconfigurable supercomputer architecture [24]. Splash-2 was designed principally to compute high-performance linear systolic applications. Through the flexibility of its architecture Splash-2 could be reconfigured to perform other tasks. Splash-2 has been used to implement image-processing functions such as Hough transforms, fast Fourier transforms and morphological operations. The architecture of Splash-2 is shown in Figure II.3.

A Splash-2 system consists of between one to fifteen Splash-2 array boards connected in a daisy-chain fashion and interconnected to a host computer. Each array board consists of 16 PEs and a crossbar switch controlled by a further PE. Each PE itself consists of a Xilinx XC4000 series FPGA and locally coupled SRAM memory. The crossbar switch enables inter-board PE communication and can be configured during run-time via the control PE. For each application, the PEs configurations were determined using development software located upon a host computer.

Splash-2 array boards are connected to the host computer using a shared bus and a private bus, connecting the host to the last board in the chain. The shared bus was known as the SIMD bus, and used to distribute input data to the array boards. Using the private bus (Rbus) results were then passed back to the host. Each array board was connected to its neighbours using a local bus network in a daisy-chain fashion.

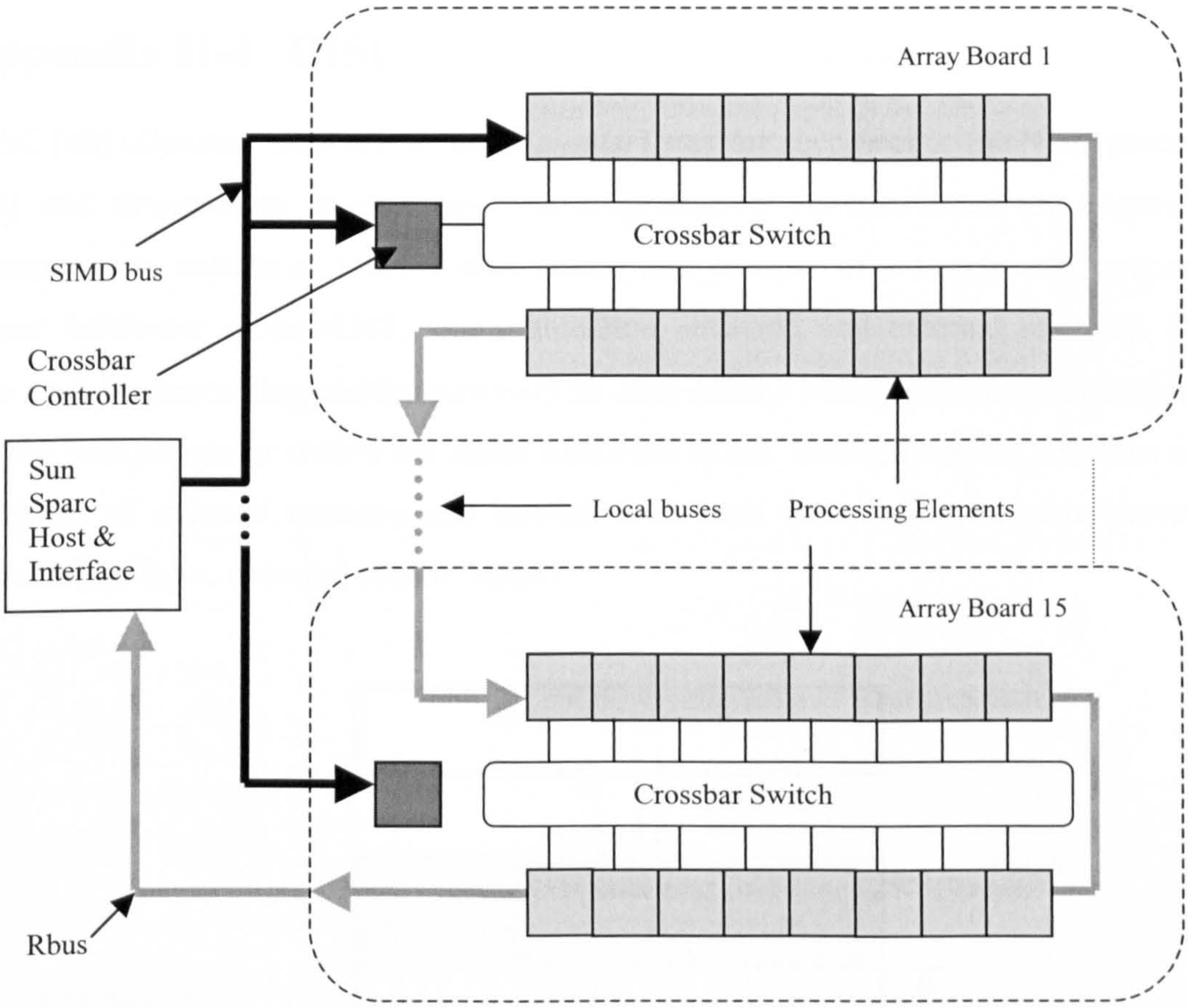


Figure II.3 Splash-2 System Architecture

Although Splash-2 was designed to implement a specific type of application, the flexibility of its architecture has enabled it to be used as a general purpose processing architecture.

The Spash-2 development cycle begins with modelling the design using VHDL. The design was then partitioned for placement amongst the available resources, which was a

manual process. A design was first partitioned between individual array boards, and then amongst individual PEs. Once completed, the configuration of the crossbar switch PE was determined. Individual PE configurations were then described in VHDL, with each being compiled, routed, and then downloaded separately to the appropriate PE. To evaluate and debug the design, Splash-2's operating environment contained a dedicated library of C programming language functions and interactive debugger tool called *T2*.

Appendix II-4 DISC

DISC [42] (*Dynamic Instruction-set Computer*) was the successor to the Nano processor [28] and designed to support run-time adaptation of its instruction-set. Figure II.4 illustrates the outline of DISC's architecture and consists of a simple core processor, *linear hardware space* (LHS), communication network, and external memory. DISC was an 8-bit processing architecture with all instructions being configured as demanded by the core processor within the linear hardware space. Instructions can accesses up to 32kbytes of external memory and interact with each other using the core processor extended address, data and control buses.

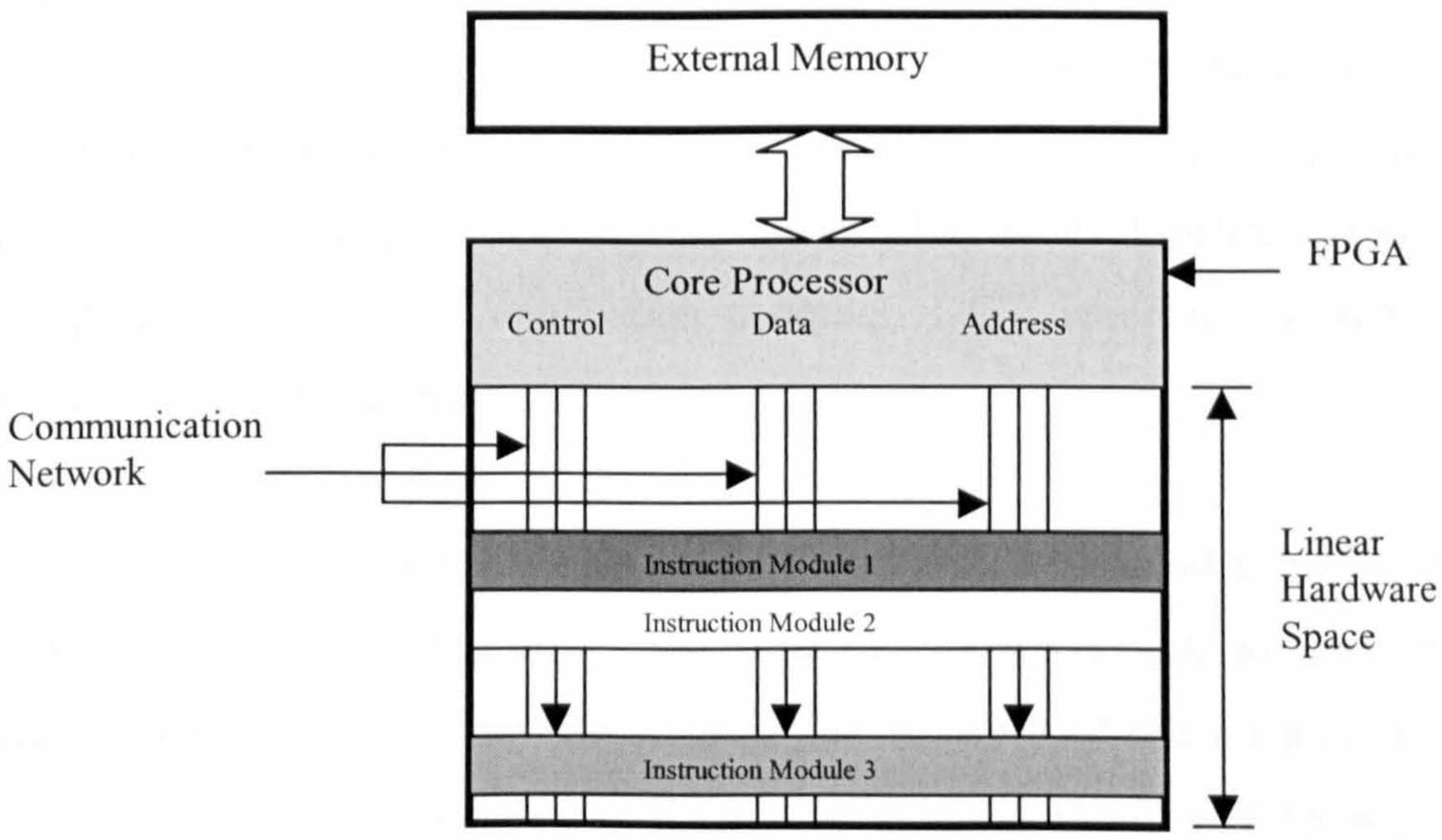


Figure II.4 DISC System Architecture

The LHS can be considered as a two-dimensional grid of CLBs used to implement active instructions. To provide an underlying structure for the run-time adaptation of instructions, each instruction was configured in a horizontal chip-wide fashion. Therefore multiple active instructions were stacked vertically. Instruction placement was not restricted to any particular location, however they could not overlap. Instructions no longer required were removed to free up hardware resources, whereas instructions required on a regular basis were allowed to remain configured. This concept was considered as *instruction caching*.

Instructions situated in the LHS were reconfigured during run-time using partial configuration. Instructions to be configured were determined by the processor core in a demand-driven manner as depicted by the application program. This program was essentially source code indicating the order of instruction execution, and contained configuration data to implement the instruction.

The instruction-set and underlying core architecture of DISC were developed using commercial HDLs and software development tools situated upon a host computer. Within this environment multiple instances of each instruction module were generated since each instruction can be configured at a different location within the LHS. Additions were made to the existing instruction-set by creating the hardware implantation of a new instruction and then including its function within the DISC development software and application compiler. DISC applications were constructed using a variant of the C programming language.

The initial DISC system was constructed within a single National Semiconductor CLAy FPGA, however, it was found that the logic resources available to implement custom instructions were inadequate. To increase the capacity of the custom instruction-set DISC2 was developed which partitioned the original DISC architecture upon three CLAy FPGAs. Improvements to the development computer interface were also made. DISC2 has been used implement image-processing functions such as low-pass filtering and binary threshold operations.

Appendix III

TMS320C40 and XC66200 Component Architectures

Introduction

This section of the Appendix provides detailed explanations of key components used within the TIM-40 multiprocessor topology and XC6200DS described in *Chapter-3*.

Appendix III-1 TMS320C40 DSP

Texas Instruments introduced the TMS320C40 (C40 hereafter) in 1989 [65]. The C40 DSP was a floating-point based processor designed specifically for use in multiple processor environments. Incorporated in the architecture were dedicated components that facilitated inter-processor communication without degrading overall system performance. The internal bus width of the C40 was 32-bit, and supported a memory map of up to 4-Gwords (16-Gbytes), with all peripherals and sub-components accessed through memory mapped I/O. A simplified block diagram of this architecture is shown in Figure III.1. The key components are the *Central Processing Unit* (CPU), *Direct Memory Access* (DMA) coprocessor, inter-processor communication ports and two external memory interfaces (*Local* and *Global*).

The C40 CPU consists of a floating point/integer multiplier, 32-bit barrel shifter, *arithmetic logic unit* (ALU), *auxiliary register arithmetic units* (ARAUs), thirty-two 32-bit registers and system buses. The multiplier can perform a 32-bit integer or 40-bit floating-point multiplication in one instruction cycle (50nsec @40MHz). The bus topology of the C40 enabled the ALU to perform single cycle 32-bit integer or 40-bit floating-point operations in parallel to the multiplier unit. This could only occur if arithmetic units did not share operands. A 32-bit barrel shifter within the ALU could also function concurrently to the multiplier.

The two ARAUs were used to generate the addresses of operands within the CPU when using displacement based addressing modes such as index addressing. Each ARAU

could generate an address location in a single instruction cycle and function concurrently to the ALU and multiplier.

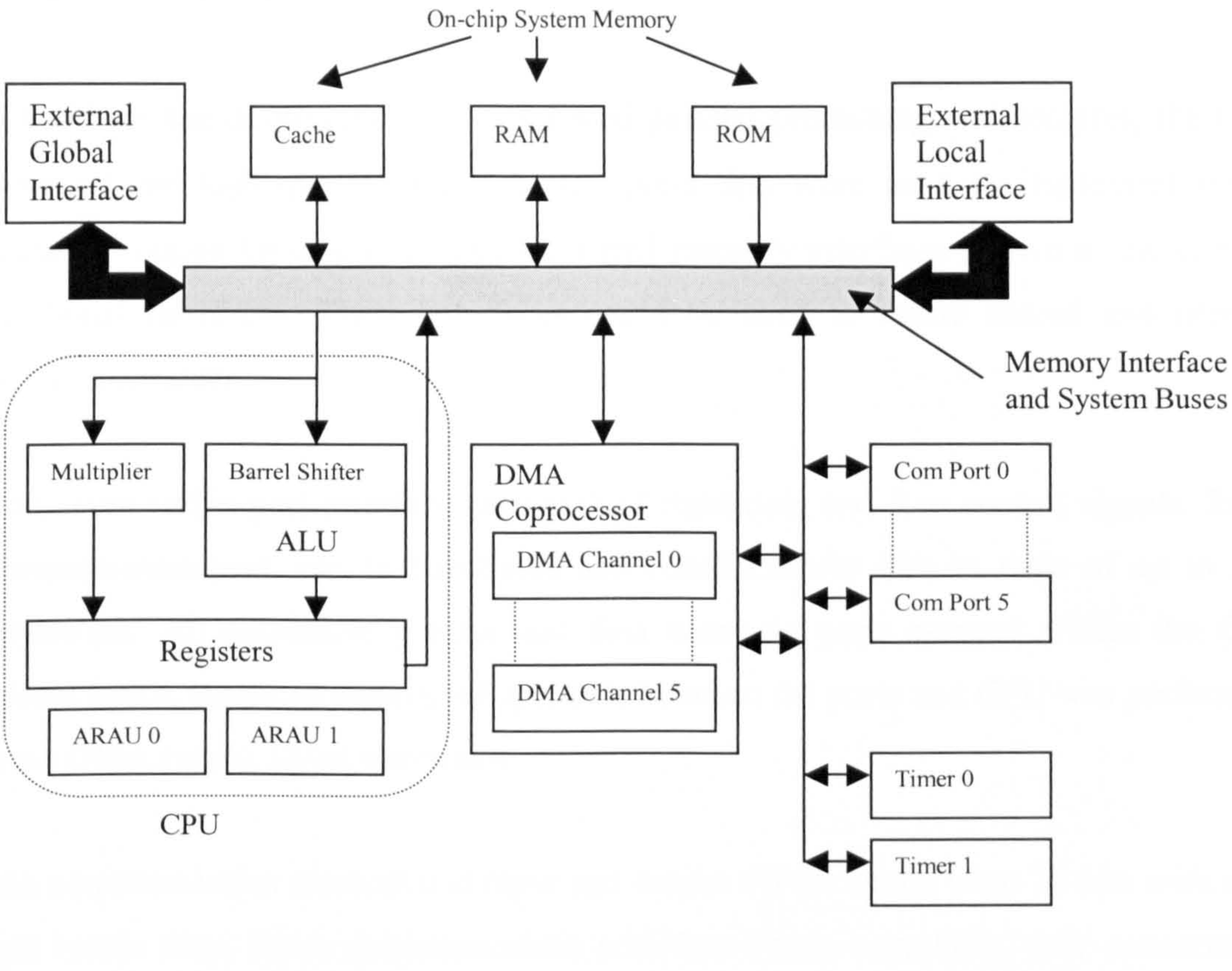


Figure III.1 TMS320C40 DSP Block Diagram

To reduce the CPUs burden of accessing operands from system memory the C40 included a DMA coprocessor. The DMA coprocessor operated in parallel to the CPU and could access any address within the C40s memory map. The DMA consisted of six channels with each able to initiate data transfers concurrently. Data transfer occurred using a dedicated system of DMA controlled buses that appeared as a multiplexed resource to the DMA communication channels. Contained within the DMA were address counters, address generation units, and synchronisation control, which enabled the DMA and CPU to function in parallel and transfer operands with minimal performance overheads.

Fully exploiting the C40 architecture, the CPU could process eight operations per instruction cycle (50nsec @40MHz), in conjunction with the DMAs three operations, giving the C40 a total performance of 275-MOPs.

To facilitate the development of C40 based parallel processing architectures, the C40 possessed six high-speed communication ports that were used to implement inter-processor routing topologies and two external memory interfaces known as the *Global* and *Local* interfaces. These interfaces could be used to create shared and private memory resources.

A communication port interface consisted of eight data and four control signals. Each communication port was bi-directional and could transfer data at rates of up to 20-Mbytes/sec. All respective control and data channels were mapped within the C40 address space, therefore transfer of operands between the ports and CPU was performed by the DMA (using 32-bit operands).

Each communication channel had input and output FIFOs which were 32-bits wide and eight levels deep. Since communication port data-buses were 8-bit, four consecutive byte transfers had to occurred to transfer a single C40 word. FIFOs provided a buffer for this operation, and in a single communication link, a combined buffer of 64 bytes existed (16 C40 words), therefore minimising the introduction of communication bottlenecks.

Connecting individual C40 communication channels together was a 'glue-less' procedure. The control of data transfer and the direction of the transfer for each communication channels was governed through the interaction of remote *Port Arbitration Units* (PAUs) using four control signals. The operation and interaction of individual PAUs upon a C40 can be considered effectively to be that of a *finite state machine* (FSM). During normal operation, this function was invisible to the user.

The Global and Local interfaces consisted of 32-bit data and 31-bit address buses and were mapped within different regions of C40 address space. Through these interfaces, single cycle operand transfer could occur from external memory to CPU (via DMA).

To provide a flexible external memory/ peripheral interface, Global and Local interfaces could be configured independently. Within both interfaces two memory strobe signals could be configured, to divide the memory space into two further subsets. Within each subset, the memory page size, interface control signal operation, and transfer rates (inclusion of read/write wait states) could be configured. This operation was conducted by writing appropriate data to interface control registers.

Contained within the C40 architecture were dedicated RAM, ROM, cache memory, and two peripheral timers that could be used to facilitate system performance benchmarking.

Appendix III-2 XC6200 Architecture

Introduced by Xilinx in June 1995, the XC6200 FPGA family reflected the changing role and application of FPGAs in electronic circuits from implementing simple peripherals to processor-based hardware [32]. The XC6200 utilises SRAM based programming technology and can be considered a second-generation sea-of-gates array architecture. XC6200 devices could operate at up to 220MHz and had logic capacities up to 100000 gates. The architecture also contained a novel interface to facilitate processor integration known as the FastMAP^{TM} interface. Through use of this interface, partial and dynamic configuration could be accomplished.

The XC6200 programmable media consists of three types of units. These were *configurable logic cells* (CLCs), *input/output blocks* (IOBs), and the routing resources; CLC was another term used by Xilinx to represent CLBs.

The CLC of a XC6200 was very different from that of previous FPGA CLBs, such as the XC4000 CLB. The XC6200 CLCs smaller granularity was a characteristic

consequence of sea-of-gate type architecture. Therefore the CLC structure and local routing resources were optimised to share logic resources with neighbouring CLCs.

The *functional unit* (FU) of the CLC consisted of five multiplexers and a D-type flip-flop. A CLC could implement a two-variable Boolean expression, compared to the XC4000 CLBs two four-variable expressions. Depending upon the expression and external dependencies, a CLC could implement both a combinatorial and sequential function. Through experience gained however, it was concluded that a CLC could on average implement only one logic or register function at any one time.

The CLC routing resources of the XC6200 were formed using a hierarchical layered routing topology formed upon blocks of 4x4 matrixes, with each level of this hierarchical structure containing its own routing resources. The first level of hierarchy consisted of a 4x4 matrix of CLCs (*length-4 routes*), with *local* interconnection routes confined within the 4x4 CLC matrix. The next level of hierarchy consisted of a 4x4 matrix of the first hierarchy of cells (*length-16 routes*), which in effect formed an array of a 16x16 CLCs. Depending upon the type of XC6200 device, hierarchy levels could be three or four layers deep. The top layer of routing hierarchy supported chip-wide routing resources known as *global* routes.

CLCs situated on the boundary edge of the highest hierarchy level were connected to IOBs, which simplified the partitioning, placement and routing of a design in multiple FPGA systems. XC6200 IOBs provided a means to route signals between CLCs and the pins on the FPGAs chip carrier. IOBs were connected to every CLC on the boundary edge of the CLC array. Not every IOB however was connected directly to an external pin, since there were more IOBs present than chip-carrier pins. Local IOB routing resources however did ensure that every IOB could be connected to at least one pin.

XC6200 family programming technology was SRAM based and could be programmed using a serial PROM like traditional FPGA applications. Unlike previous FPGAs, the XC6200 supported a dedicated processor interface, which enabled XC6200 SRAM

control and configuration memory to appear within the memory map of a host processor. Therefore the XC6200 appeared as a peripheral memory device, allowing its configuration to be updated by writing new data to the appropriate address location. This concept is shown in Figure III.2. Depending upon the XC6200 device used, the interface could support up to 18-bit address and 32-bit data buses.

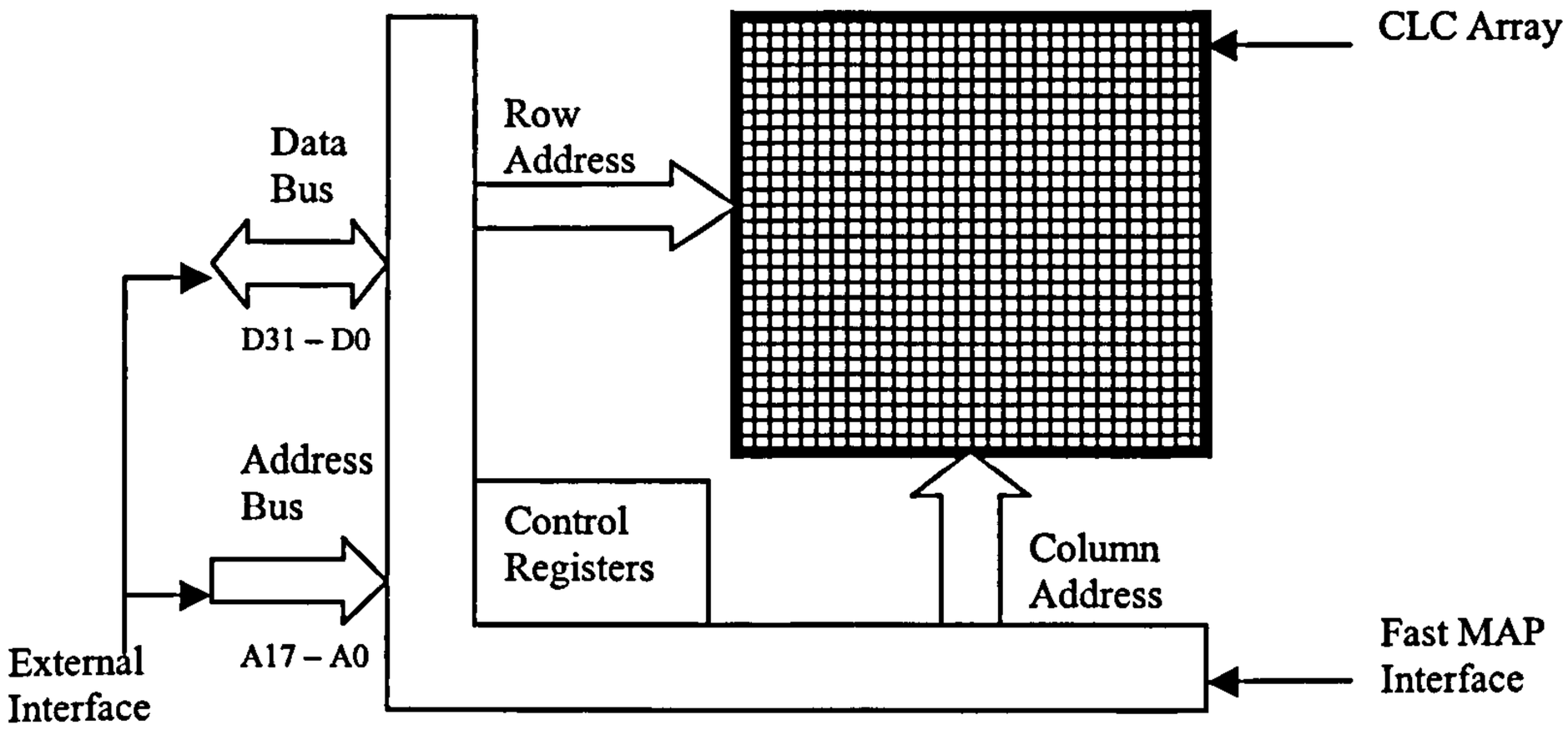


Figure III.2 XC6200 FastMAP™ interface

Within the FastMAP™ interface, routing switches and CLCs were allocated address locations within the XC6200 configuration memory. Configuration data was transferred to memory using the FastMAP™ 32-bit data bus. However, the bus width was flexible and configuration data could be written in 8,16 or 32-bit formats.

The FastMAP™ interface allowed the XC6200 to support partial configuration and dynamic configuration capability. This was because individual CLCs could be reconfigured without having to suspend the operation of unaffected CLCs. Through this interface, the content of CLC registers could be accessed. This provided a means to transfer data between host and CLC array and access internal control registers. Dynamic configuration could only be performed however using 8-bit FastMAP™ data bus transfers.

The XC6200 FPGA family never went into full production. The specifications of the devices obtained to construct the XC6200 ADS are detailed in Table III.1.Through experiments conducted however it became apparent that the gate capacities and maximum operating frequency listed could not be replicated.

Device	XC6216	XC6264
Typical gate count	16000-24000	64000-100000
Number of cells	4096	16384
Number of I/O blocks	256	512
Array matrix size	64x64	128x128

Table III.1 XC6200 FPGA Specifications

Appendix IV

XC6200DS Hardware

Introduction

This section of the Appendix contains Vantis Mach111 CPLD PALASM and Xilinx XC4005 FPGA schematic designs used in the construction of the XC6200DS.

Appendix IV-1 ISA Bus Interface

```
TITLE           isa_bus_int
PATTERN         A
REVISION        VER1
AUTHOR          C.MURPHY
COMPANY         CEORG
DATE            07/08/01

CHIP    isa_bus_int MACH111

PIN  18          IOR             COMBINATORIAL    ;    INPUT
PIN  17          IOW             COMBINATORIAL    ;    INPUT
PIN  16          AEN             COMBINATORIAL    ;    INPUT
PIN  [2..9]      ADD_IN[9..2]    COMBINATORIAL    ;    INPUT
PIN  [15..14]    ADD_IN[0..1]    COMBINATORIAL    ;    INPUT
PIN  [40..43]    ADD_OUT[3..0]   COMBINATORIAL    ;    OUTPUT
PIN  39          DATA_ENA        COMBINATORIAL    ;    OUTPUT
PIN  38          IOR_OUT         COMBINATORIAL    ;    OUTPUT
PIN  37          IOW_OUT         COMBINATORIAL    ;    OUTPUT
PIN  11          PC_CLK_IN       COMBINATORIAL    ;    INPUT
PIN  25          PC_CLK_OUT      COMBINATORIAL    ;    OUTPUT
PIN  29          TEST_P          COMBINATORIAL    ;    OUTPUT

NODE ?           ADD_NODE[3..0]   COMBINATORIAL   ;
NODE ?           ADD_IS_32X      COMBINATORIAL    ;
NODE ?           NOT_DMA         COMBINATORIAL    ;
NODE ?           IOR_IOW         COMBINATORIAL    ;
NODE ?           N_D_ENA         COMBINATORIAL    ;

EQUATIONS

reset = gnd;

ADD_NODE[3..0] = ADD_IN[3..0]
ADD_OUT[3..0] = ADD_NODE[3..0]
```

```
; Define ISA Address Range 0x32X
IF (ADD_IN[9..4] = #H32) THEN
        BEGIN
                ADD_IS_32X = VCC
        END
ELSE
        BEGIN
                ADD_IS_32X = GND
        END

NOT_DMA = ADD_IS_32X * /AEN
IOR_IOW = IOR * IOW

N_D_ENA = /IOR_IOW * NOT_DMA
DATA_ENA = /N_D_ENA

IOR_OUT = IOR
IOW_OUT = IOW

PC_CLK_OUT  = PC_CLK_IN


;Define test vectors
SIMULATION

TRACE_ON

FOR X:= 1 TO 20 DO
      BEGIN
        CLOCKF pc_clk_in
        END
TRACE_OFF
```

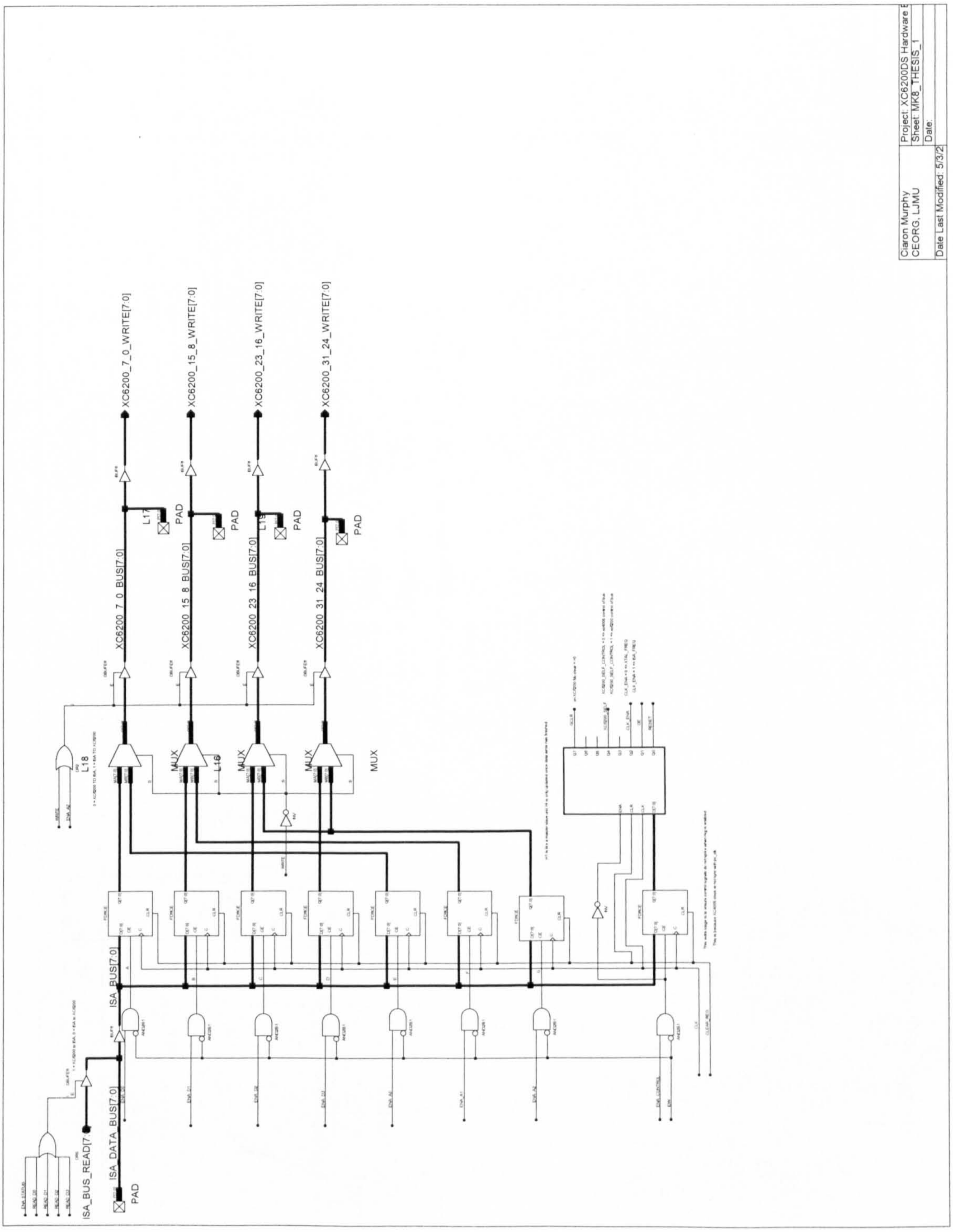
Appendix IV-2 FastMAP™ Interface Controller

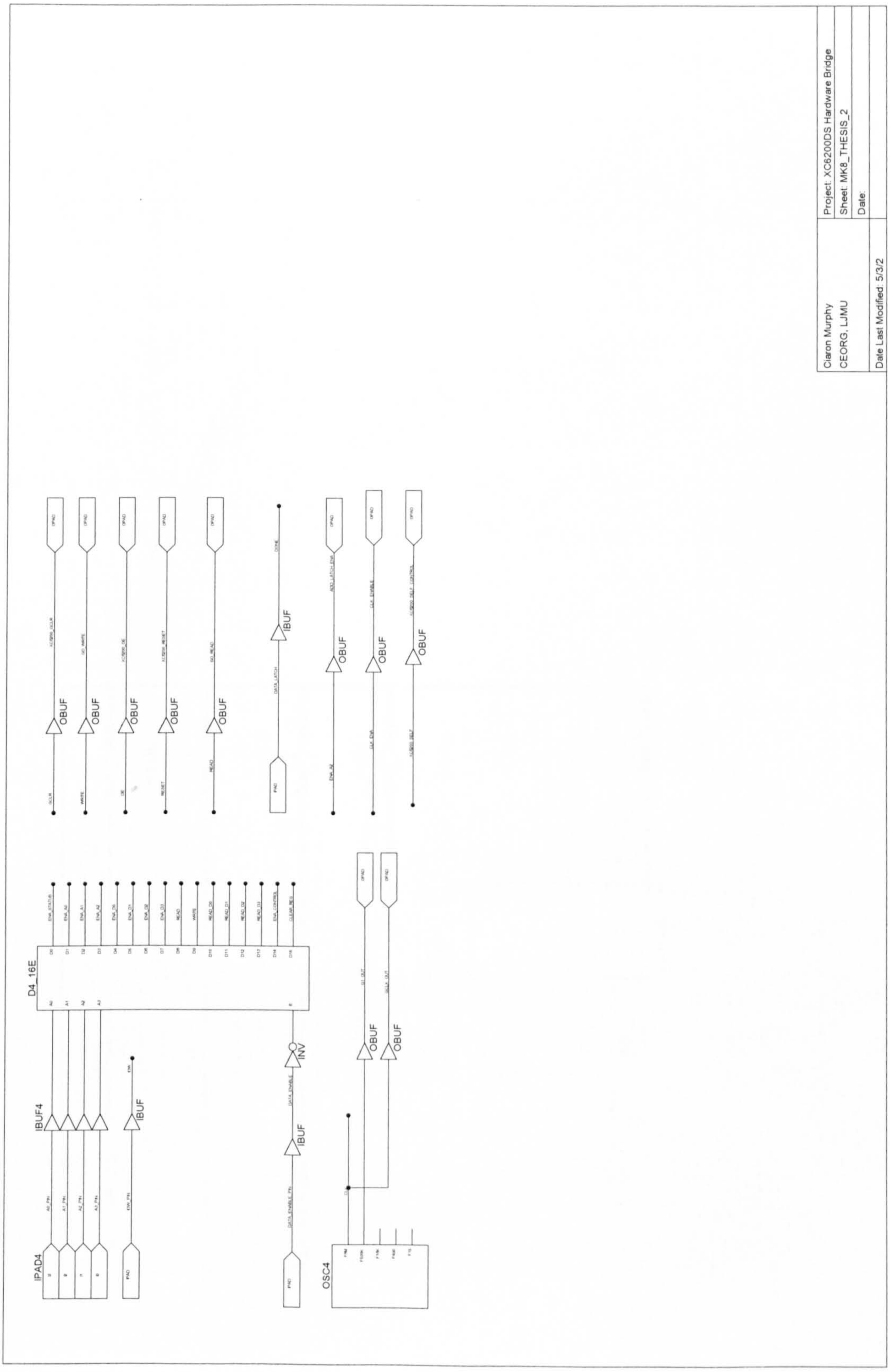
```
TITLE          FastMAP Int. STATE MACHINE
PATTERN        BOOLEAN NETLIST
REVISION       3..
AUTHOR         CIARON MURPHY
COMPANY        CEORG
DATE           13/03/01

CHIP           Fast_MAP MACH111

PIN 15         PC_CLK                                  ; INPUT
PIN 16         XTAL_CLK                                ; INPUT
PIN 5          CLK_CON                                 ; INPUT
PIN 17         CLOCK_OUT                               ; OUTPUT

PIN 6          X6200_XC4005_OE                         ; INPUT
PIN 11         CLOCK                                   ; INPUT
PIN 7          RESET                                   ; INPUT

PIN 8          GO_READ                                 ; INPUT
PIN 9          GO_WRITE                                ; INPUT
PIN 18         SELF_WRITE                              ; INPUT
PIN 24         XC4005_ENA                              ; INPUT
PIN 19         XC4005_A_LATCH                          ; INPUT
PIN 20         XC6200_A_LATCH                          ; INPUT
PIN 21         ADD_LATCH          COMBINATORIAL        ; OUTPUT
PIN 25         XC4005_ena_ext                          ; INPUT
PIN 4          XC6200_RW          COMBINATORIAL           ; OUTPUT
PIN 2          XC6200_CE          COMBINATORIAL           ; OUTPUT
PIN 14         XC6200_OE          COMBINATORIAL           ; OUTPUT
PIN 3          DLATCH             COMBINATORIAL           ; OUTPUT

NODE ?  Y1     REGISTERED
NODE ?  Y2     REGISTERED
NODE ?  X1     REGISTERED
NODE ?  X2     REGISTERED

NODE ? CLK1    COMBINATORIAL
NODE ? CLK2    COMBINATORIAL
NODE ? SELF_WRITE_ENA    COMBINATORIAL
NODE ? WRITE   COMBINATORIAL

NODE ? ADD1    COMBINATORIAL
NODE ? ADD2    COMBINATORIAL


EQUATIONS

X2.RSTF = /RESET
X1.RSTF = /RESET
X2.CLKF = CLOCK
X1.CLKF = CLOCK
```

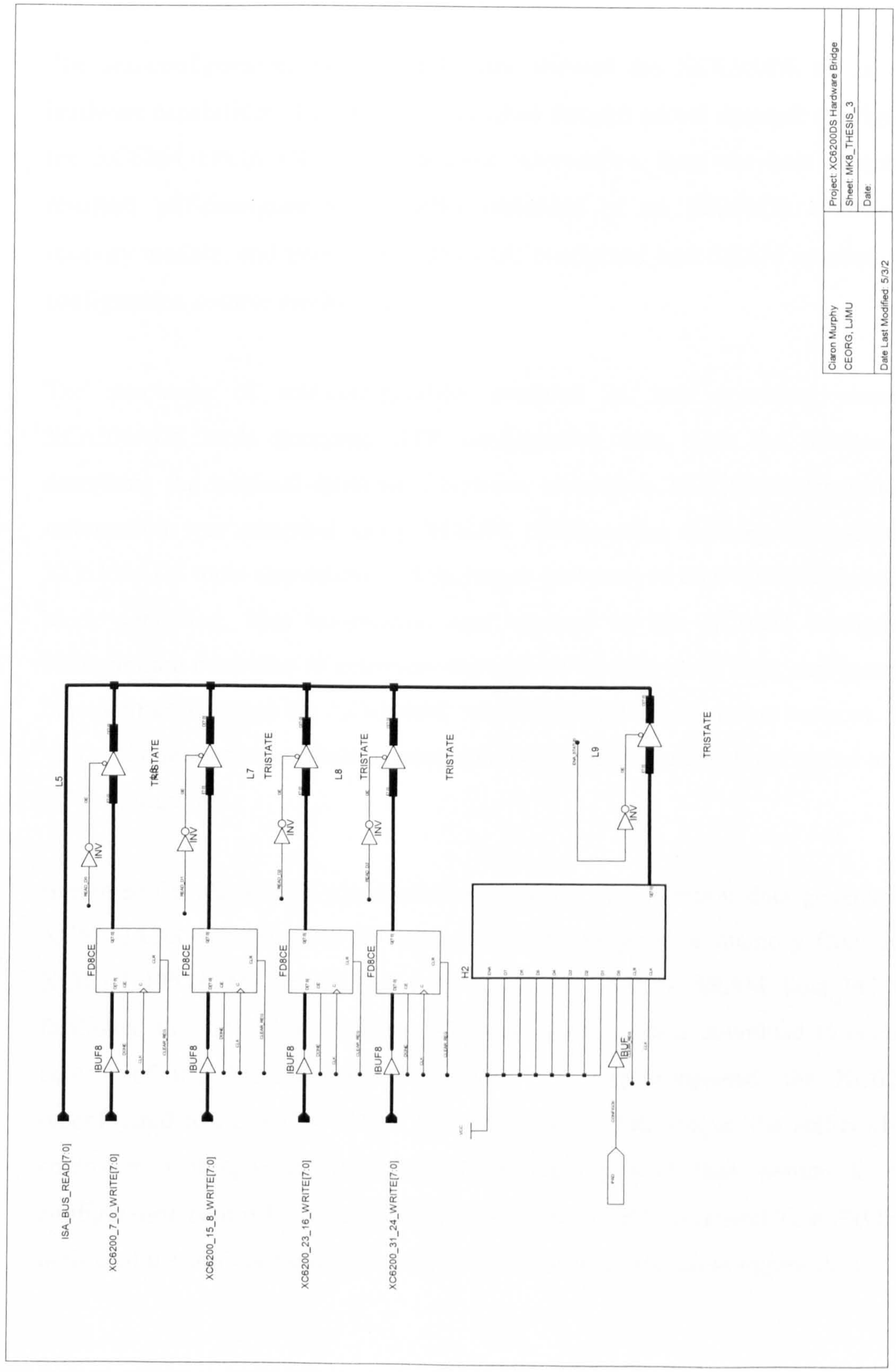
```
Y2.RSTF = /RESET
Y1.RSTF = /RESET
Y2.CLKF = CLOCK
Y1.CLKF = CLOCK

; Define clock selection mechanism

CLK1 =  /CLK_CON * XTAL_CLK
CLK2 =  CLK_CON * PC_CLK

CLOCK_OUT = CLK1 + CLK2  ;   CLOCK_OUT IS ACTUAL SIGNAL CLOCK, VIA HW
EXT.

; Define FastMAP Address bus latch
ADD1 =  /XC4005_ENA * XC4005_A_LATCH
ADD2 =  ( (/XC4005_ENA *  /XC4005_ena_ext) + XC4005_ENA )*
XC6200_A_LATCH

ADD_LATCH = ADD1 + ADD2

; State equations for FastMAP Read Operation
Y1 := Y2
Y2 := (/Y1 * Y2) + (/Y1 * GO_READ) + (Y2 * GO_READ)

; State equations for FastMAP Write Operation

SELF_WRITE_ENA = SELF_WRITE * ((/XC4005_ENA * /XC4005_ena_ext) +
XC4005_ENA)

WRITE = SELF_WRITE_ENA + GO_WRITE

X1 := (/X2 * X1) + (/X2 * WRITE) + ( X1 * WRITE)
X2 := (/X2 * X1) + ( X1 * WRITE)

; COMBINED STATE MACHINE OUTPUTS

XC6200_RW = (/X1 * /X2) + (X1 * X2) + (/X1 *  X2)
DLATCH = Y2
XC6200_CE = ((/X1 * /X2) + (X1 * X2) + (/X1 *  X2)) * /Y2

XC6200_OE = X6200_XC4005_OE

; Define hardware test vectors
SIMULATION

TRACE_ON  CLOCK

SETF /RESET
SETF RESET
CLOCKF CLOCK
CLOCKF CLOCK

SETF /XC4005_ENA
CLOCKF CLOCK
CLOCKF CLOCK
```

```
FOR X:= 1 TO 5 DO
      BEGIN
      CLOCKF CLOCK
      END

SETF /GO_WRITE

FOR X:= 1 TO 5 DO
      BEGIN
      CLOCKF CLOCK
          END
TRACE_OFF
```

Appendix IV-3 Hardware-Bridge




Project: XC6200DS Hardware Bridge
Sheet: MK8_THESIS_2
Date:
Ciaron Murphy
CEORG, LJMU
Date Last Modified: 5/3/2
OPAD
OBUF
SCLR
WRITE
OE
RESET
READ
IBUF
DONE
DATA_LATCH
PAD
DATA_LATCH_ENA
DATA_ENABLE
ENA_A2
CLEAR_REG
D4_16E
ENA_STATUS
ENA_A0
ENA_A1
ENA_A2
ENA_D6
ENA_D1
ENA_D2
ENA_D3
READ_D6
READ_D1
READ_D2
READ_D3
ENA_CONTROL
A0
A1
A2
A3
D0
D1
D2
D3
D4
D5
D6
D7
D8
D9
D10
D11
D12
D13
D14
D15
E
INV
IBUF4
IPAD4
A0_PIN
A1_PIN
A2_PIN
A3_PIN
ENA_PIN
DATA_ENABLE_PIN
OSC4


Project: XC6200DS Hardware Bridge
Sheet: MK8_THESIS_3
Date:
Ciaron Murphy
CEORG, LJMU
Date Last Modified: 5/3/2
ISA_BUS_READ[7:0]
XC6200_7_0_WRITE[7:0]
XC6200_15_8_WRITE[7:0]
XC6200_23_16_WRITE[7:0]
XC6200_31_24_WRITE[7:0]
L5
L7
L8
L9
TRISTATE
INV
IBUF8
FD8CE
H2
IBUF
PAD

Appendix IV-4 Self-Configuration Controller

The self-configuration control mechanism allowed the XC6200DS to exhibit RTR hardware capabilities. This was accomplished through partial dynamic configuration of the XC6264 FPGA CLC array, without intervention from the host computer. The resultant self-configuration controller consisted of an X6200DS external SRAM memory module, and internal XC6264 CLC configured host SRAM interface and self-configuration control mechanisms.

The processes of self-configuration occurred in two operating phases. First XC6200ADS tools generated RTR configuration data, with the resultant outputs describing the minimal differences between successive XC6264 configurations. This information was compiled using XC6264 configuration memory address/data pairs. XC6200ADS tools also determined the length (in bytes) of each RTR configuration data block generated. This information was required by the XC6264 configured self-configuration controller to determine the address boundaries of RTR configuration data blocks situated within the XC6200DS external SRAM configuration memory. This task was performed prior to dynamic operation, with result files generated stored on the host PC for future use.

Before self-configuration could commence, RTR configuration data generated by the XC6200ADS was written to the external SRAM configuration memory. Configuring the XC6264 FPGA to function as an external XC6200DS SRAM host PC interface facilitated this operation, enabling RTR configuration data download to occur, under control of the XC6200ADS. Once this task was completed, the XC6264 was reconfigured using X6200ADS CTR techniques to function as the self-configuration controller. Configuration data address boundaries were then written to the self-configuration controllers internal PROM (via FastMAP^{TM} interface) This PROM and an outline of the self-configuration control mechanism are shown in Figure IV-1.

Figure IV-1 illustrates the self-configuration control mechanism, that consisted of a 16x18-bit PROM, loadable incremental binary counter, control unit, and 8-bit register latches. These components were configured within the XC6264s CLC array. Complementing the XC6264 design components, an external 262144bytes SRAM module, three 74LS373 ICs (8-bit latches), and signal interfaces to XC6200DS Hardware Bridge and FastMAP™ Interface controller were also required.

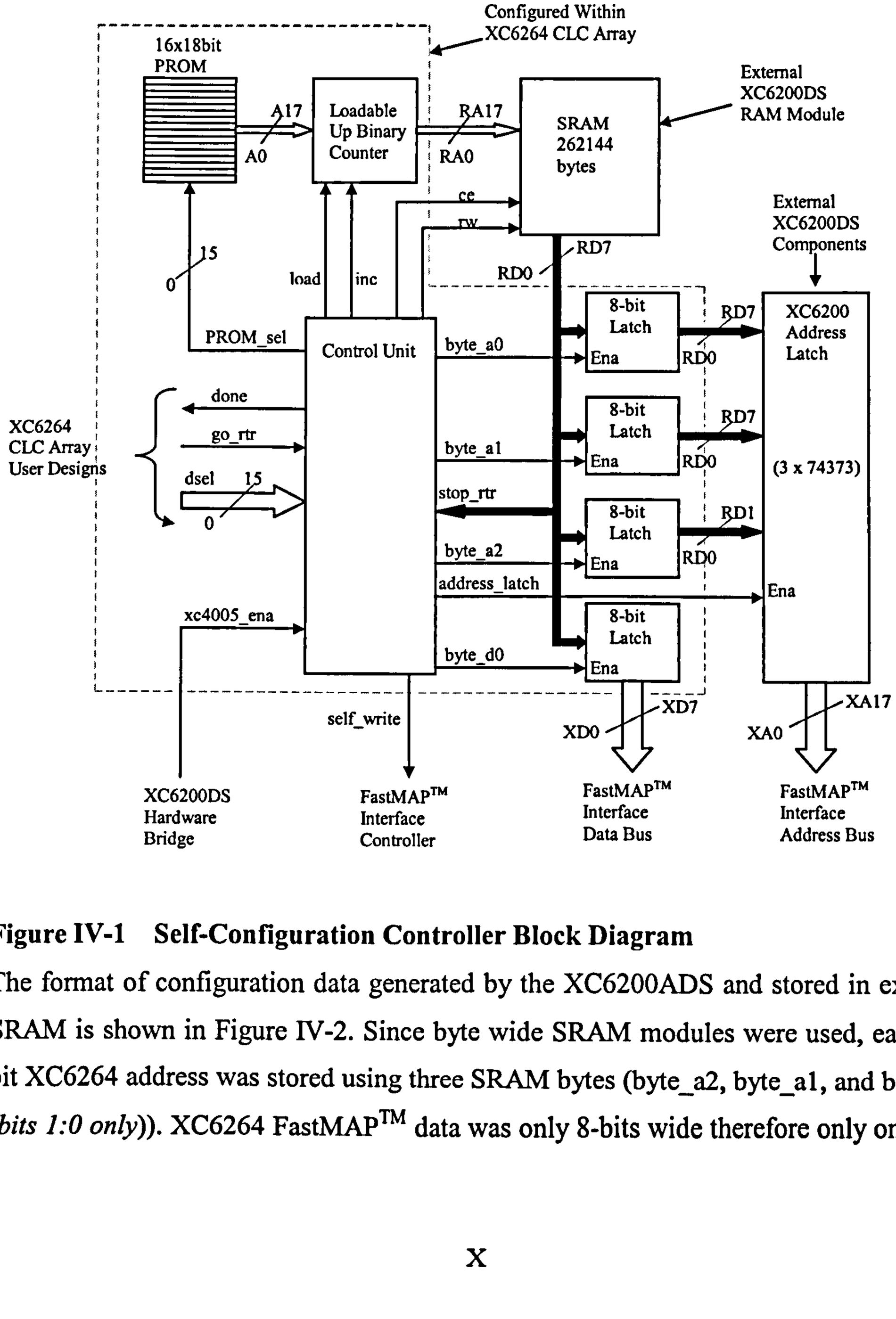


Figure IV-1 Self-Configuration Controller Block Diagram

The format of configuration data generated by the XC6200ADS and stored in external SRAM is shown in Figure IV-2. Since byte wide SRAM modules were used, each 18-bit XC6264 address was stored using three SRAM bytes (byte_a2, byte_a1, and byte_a2 (*bits 1:0 only*)). XC6264 FastMAP™ data was only 8-bits wide therefore only one byte

was required (byte_d0). Using this format, XC6264 configuration data address/data pairs were stored in the external configuration SRAM module using sequential memory locations. The self-configuration controller supported up to 16 different configurations, with the next active configuration determined using signal dsel(15:0) (for clarity only three start addresses are shown in *Figure IV-2*).

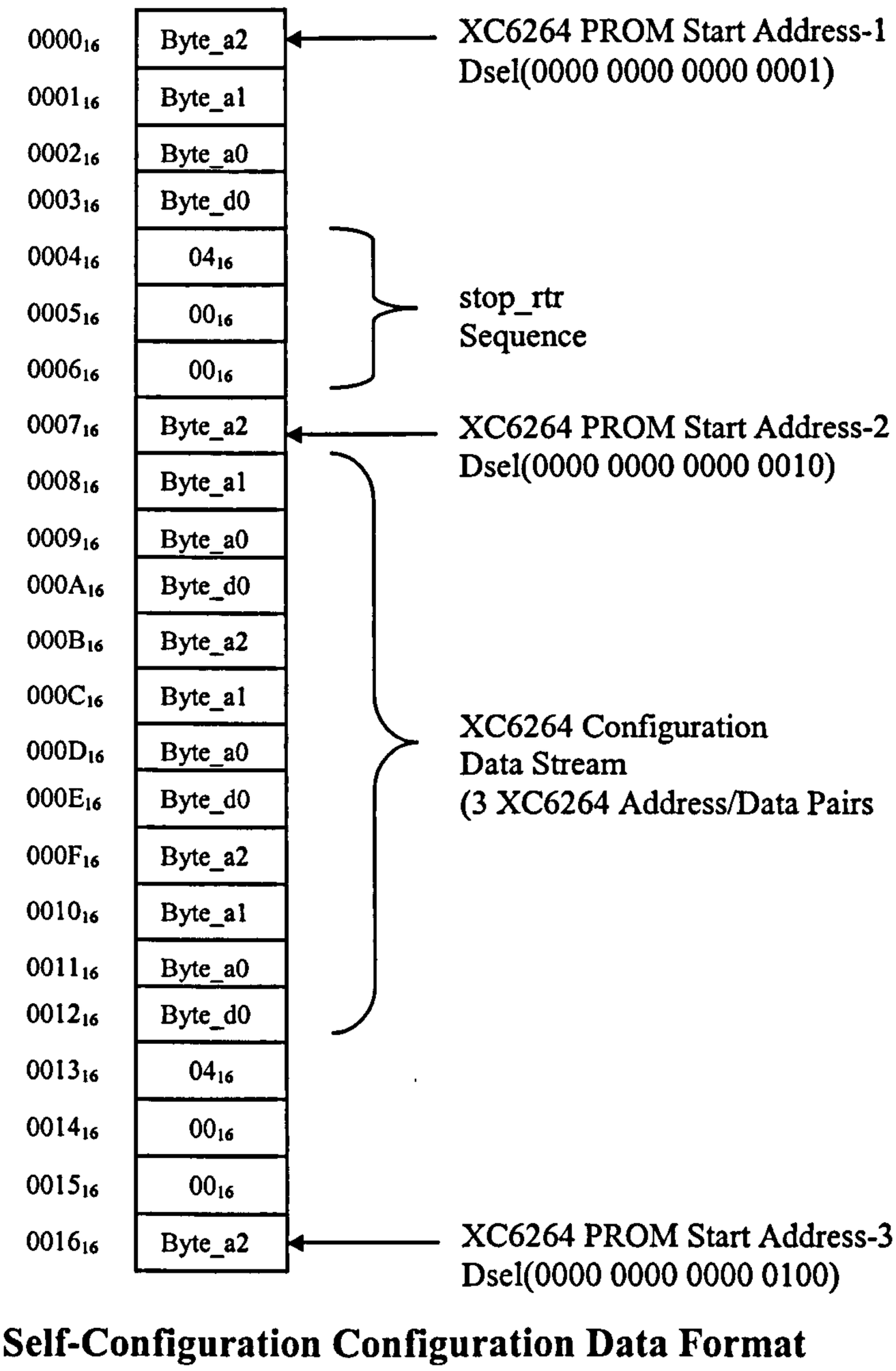


Figure IV-2 Self-Configuration Configuration Data Format

XC6200ADS tools generated the start address of each active configuration (16 values stored in PROM), with the end of each configuration block determined by the self-configuration control mechanism reading an address value of 40000_{16} (stored in byte_a2, byte_a1, and byte_a2). This action generated signal stop_rtr, which indicated

(via signal done) that the RTR update was complete. With respect to Figure IV-1 and the self-configuration controller control unit FSM operational states (Sn) (Figure IV-3), the internal operation of the self–configuration control mechanism is described next.

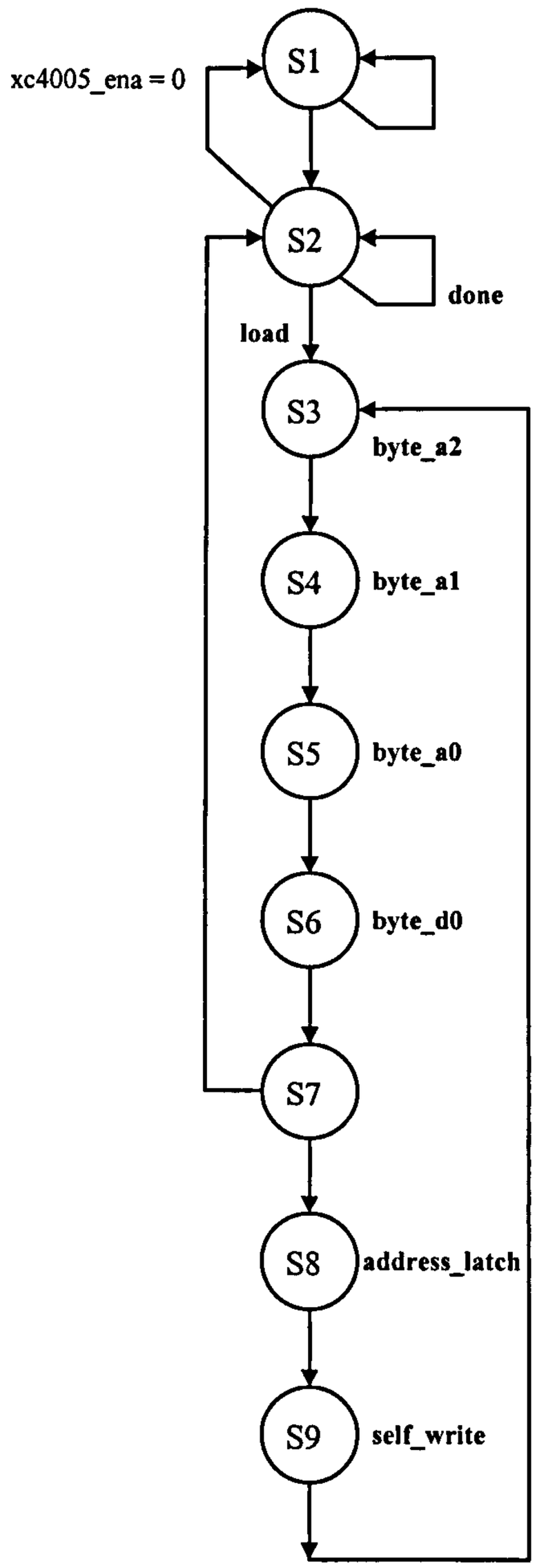


Figure IV-3 Self-Configuration Control Unit Finite State Machine

Prior to system operation RTR configuration data was written to external SRAM and the starting addresses of each configuration written to the self–configuration controller PROM. To enable the self-configuration control mechanism, XC6200ADS functions

were used to activate signal xc4005_ena within the control register of the XC6200DS Hardware Bridge (S1). Once enabled, the XC6264 could be dynamically reconfigured through activating signal go_rtr (S2), with the next active configuration determined by signal dsel(15:0), with signal done indicating that RTR was commencing. Done, go_rtr, and dsel provided the interface between the self-configuration controller mechanism and user defined hardware configured within the XC6264 CLC array.

During (S2), the contents of the PROM at the address location selected by PROM_sel (originates from dsel) were loaded into an incremental counter. The output of this counter was used as external configuration SRAM address bus value. This counter was incremented using signal inc after each SRAM memory access.

Four SRAM address locations were then read in succession (S3), (S4), (S5), and (S6), with the contents of each location stored temporally in 8-bit latches. The active latch enabled for each SRAM read operation (S3), (S4), (S5), and (S6), was determined through control unit FSM operation.

If the stop_rtr address value was read (40000_{16} determined by comparator operation in control unit), this indicated that RTR was completed, with signal done disenabled, and the control unit FSM operation re-entering state (S2). If 40000_{16} was not detected, address and data bytes read from SRAM were valid XC6264 RTR configuration data. This data was then downloaded to XC6264 configuration memory using its FastMAP™ interface.

This process required the XC6264s FastMAP™ address-bus to be first set up by loading bytes byte_a0, byte_a1, and byte_a2 into external latches (3x 74LS373s), using signal address_latch (S8). Byte_d0 was then written to the FastMAP™ data-bus (S9), with FastMAP™ interface control signals generated by the XC6200DS FastMAP™ Interface Controller. This operation was performed through the control unit generating signal self_write. Controller FSM operation then resumed back to state (S3), with the

cyclic operation of (S3) to (S9) repeated for the number of XC6264 address/data pairs that required updating.

Appendix IV-5 XC6200DS Signal Connectors

	XC6200DS		SRAM Module	
Signal	Connector J14	XC6200 Pin No.	IC1 Pin	IC2 Pin
D0	9	70	16	N/A
D1	10	71	17	N/A
D2	11	72	18	N/A
D3	12	73	19	N/A
D4	13	74	N/A	16
D5	14	76	N/A	17
D6	16	81	N/A	18
D7	15	77	N/A	19
A0	17	82	21	21
A1	19	87	22	22
A2	21	89	23	23
A3	20	88	24	24
A4	23	95	25	25
A5	22	94	26	26
A6	27	100	27	27
A7	29	105	1	1
A8	31	108	2	2
A9	32	109	3	3
A10	25	97	4	4
A11	33	110	5	5
A12	26	99	6	6
A13	28	104	7	7
A14	35	115	8	8
A15	34	111	9	9
A16	37	116	10	10
A17	38	120	11	11
OE	6	66	13	13
CE	7	68	12	12
R/W	8	69	15	15
VCC	1 , 39	N/A	28	28
GND	2 , 40	N/A	14	14

Table IV-1 XC6200DS SRAM Memory Module Pin Description

XC6200DS J1, J3	
Pin No.	**Signal**
1	TDI
2	VCC
3	GND
4	TDO
5	N/A
6	TCK
7	N/A
8	TNS
9	N/A
10	N/A

Table IV-2 XC6200DS Vantis MACH111 ISP Socket

XC6200DS J5-J13	
Pin No.	**Signal**
1	N/A
2	D0
3	D1
4	D2
5	D3
6	GND
7	D4
8	D5
9	D6
10	D7
11	GND
12	CREQ
13	GND
14	CACK
15	GND
16	CSTRB
17	GND
18	CRDY
19	GND
20	N/A

Table IV-3 XC6200DS TIM-40 TMS320C40 Comport Socket

	XC6200DS		SRAM Module	
Signal	**Connector J15**	**XC6200 Pin No.**	**IC1 Pin**	**IC2 Pin**
D0	19	202	16	N/A
D1	24	208	17	N/A
D2	38	228	18	N/A
D3	25	209	19	N/A
D4	26	210	N/A	16
D5	27	213	N/A	17
D6	28	218	N/A	18
D7	37	220	N/A	19
A0	37	226	21	21
A1	7	187	22	22
A2	36	225	23	23
A3	34	223	24	24
A4	4	183	25	25
A5	3	181	26	26
A6	35	224	27	27
A7	33	221	1	1
A8	30	216	2	2
A9	28	214	3	3
A10	29	215	4	4
A11	22	206	5	5
A12	23	207	6	6
A13	20	203	7	7
A14	21	205	8	8
A15	17	198	9	9
A16	15	195	10	10
A17	13	193	11	11
OE	12	192	13	13
CE	11	191	12	12
R/W	10	190	15	15
VCC	1 , 39	N/A	28	28
GND	2 , 40	N/A	14	14

Table IV-4 XC6200DS Self-Configuration SRAM Memory Module

	XC6200DS	
Signal	**Connector J17**	**Connector J15**
XC_AD_0	1	19
XC_AD_1	2	24
XC_AD_2	3	38
XC_AD_3	4	25
XC_AD_4	5	26
XC_AD_5	6	27
XC_AD_6	7	28
XC_AD_7	8	37
XC_A_8	9	37
XC_A_9	10	7
XC_A_10	11	36
XC_A_11	12	34
XC_A_12	13	4
XC_A_13	14	3
XC_A_14	15	35
XC_A_15	16	33
XC_A_16	17	30
XC_A_17	18	28

	Connection	
Signal	**From**	**To**
add_latch	IC10,P197	IC7,IC8,IC9, P11
xc4005_ena	IC14,P37	IC10,P184
xc6200_write	IC10,P189	IC1,P18

Table IV-5 XC6200DS Self-Configuration Operation Mode Modifications

Appendix V

Published Work and Awards Presented

Appendix-V-1 Conference papers published

Conference	SPIE International Symposium on Voice, Video, and Data Communications, Conference No. 3526: 'Configurable Computing: Technology and Applications',
Venue	Boston, Massachusetts, USA.
Date	1st-5th November 1998.
Title of paper	*A Low-Cost Reconfigurable DSP-based Parallel Image-Processing computer.*
Conference	IEE Informatics Colloquium on Reconfigurable Systems.
Venue	Glasgow, UK.
Date	10th March 1999.
Title of paper	*Low-cost TMS320C40/XC6200 Based Reconfigurable Parallel Image-Processing Architecture.*
Conference	IASTED International Conference Applied Informatics, International Symposium on Parallel and Distributed Computing and Networks.
Venue	Innsbruck, Austria.
Date	18th-21st February 2002.
Title of paper	*Dynamic Configurable DSP Parallel Processing Architecture.*

Photostat copies of these papers taken from the original conference proceedings are located at the back of the thesis.

Appendix-V-2 Awards

Title	Royal Academy of Engineering Travel grant
Award Fund	Royal Academy of Engineering
Purpose	To Present Paper at SPIE conference

Title	Royal Academy of Engineering Travel grant
Award Fund	Royal Academy of Engineering
Purpose	To Present Paper at IASTED Conference

Appendix VI

XC6264 Design Footprints

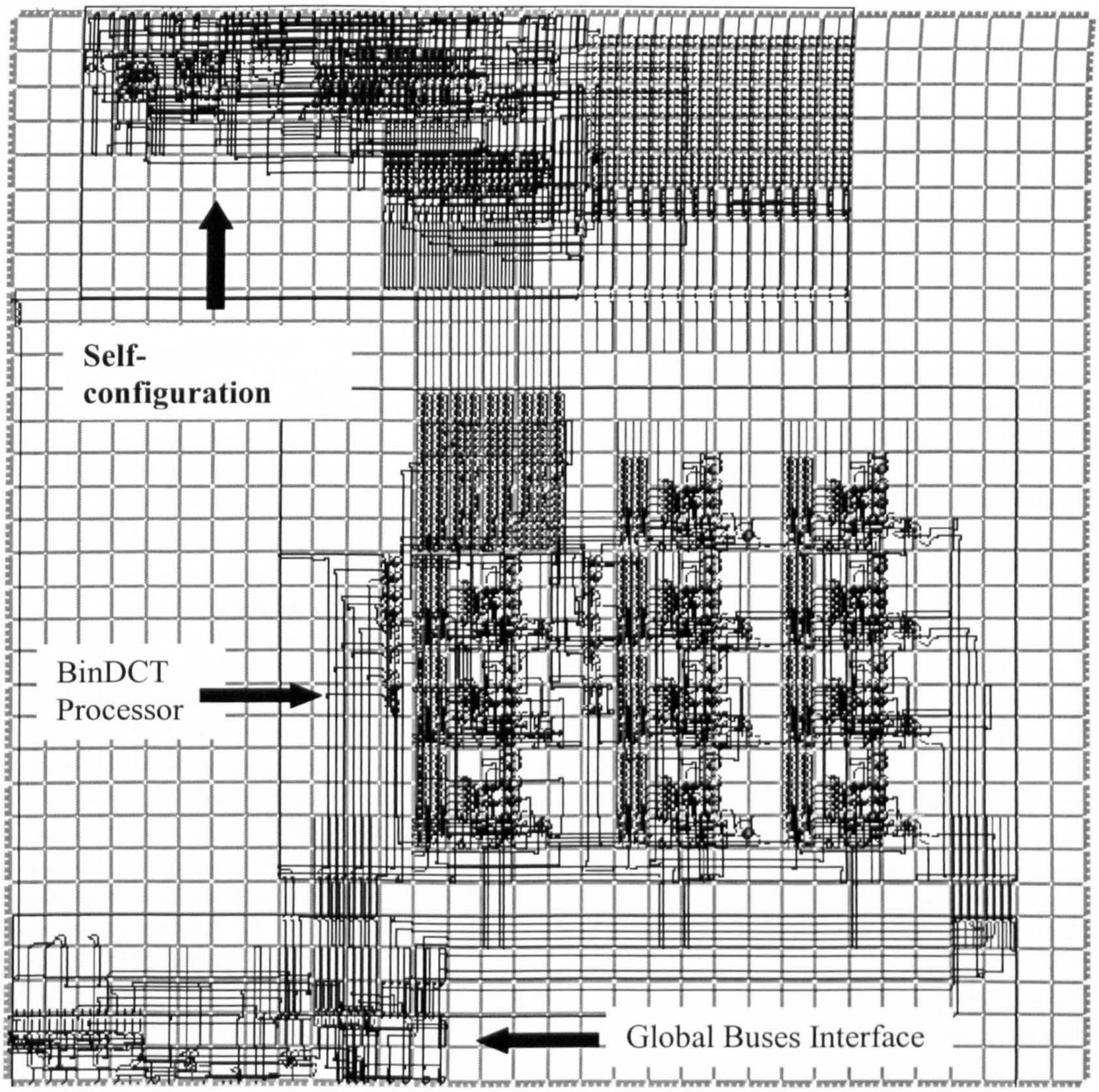


Figure VI-I FBinDCT-C1 XC6264 Footprint

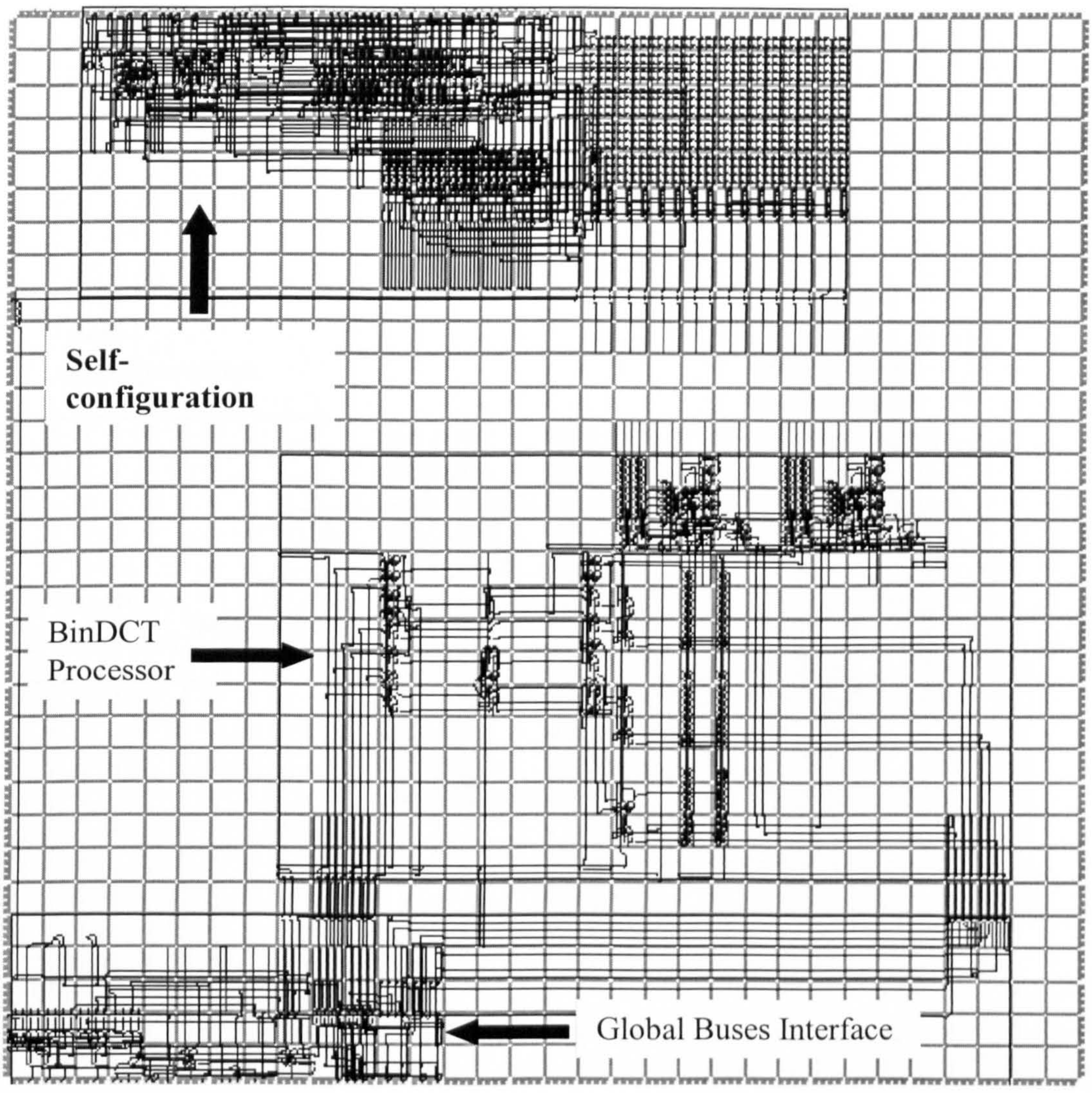


Figure VI-II FBinDCT-C9 XC6264 Footprint

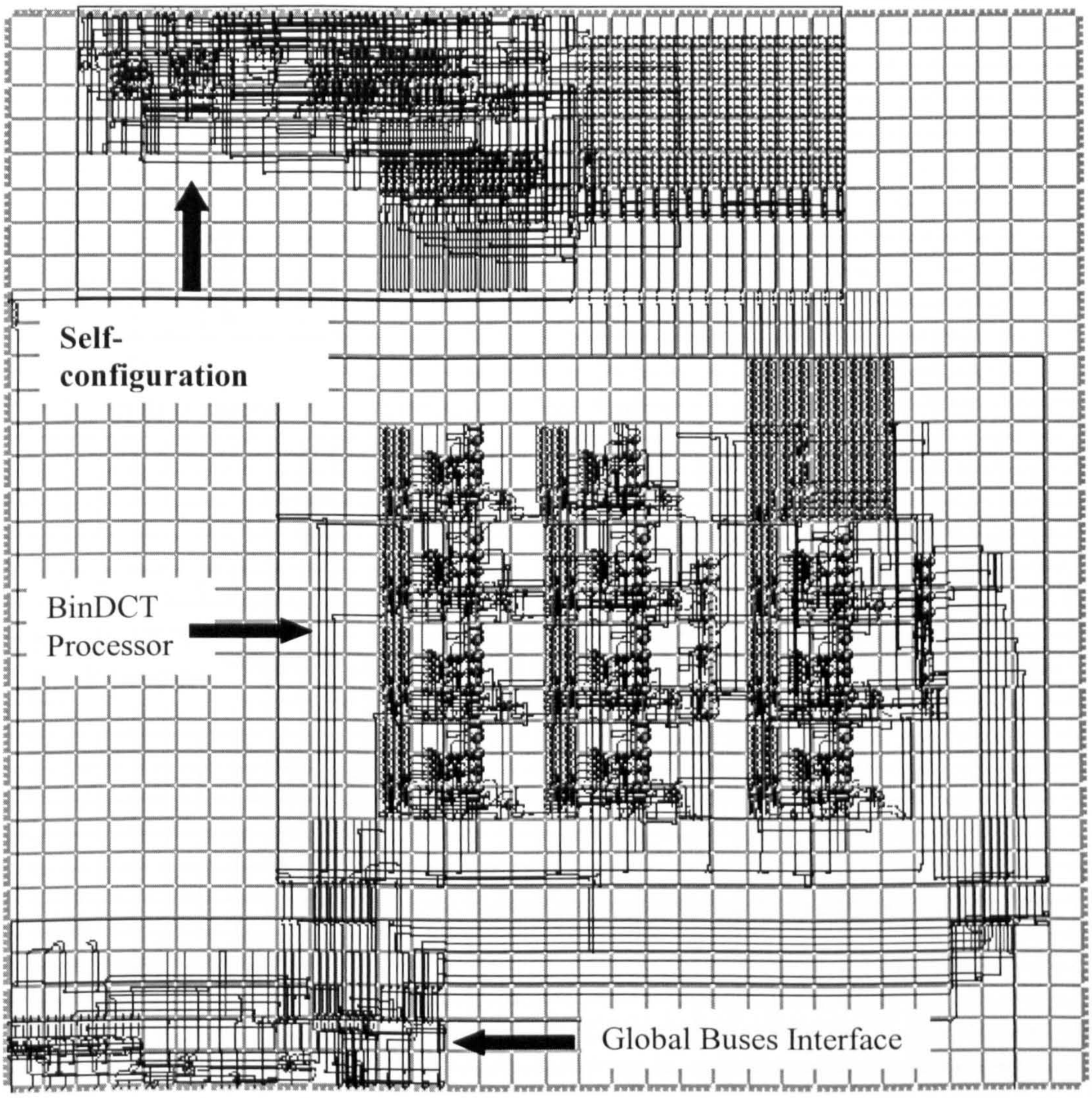


Figure VI-III RBinDCT-C1 XC6264 Footprint

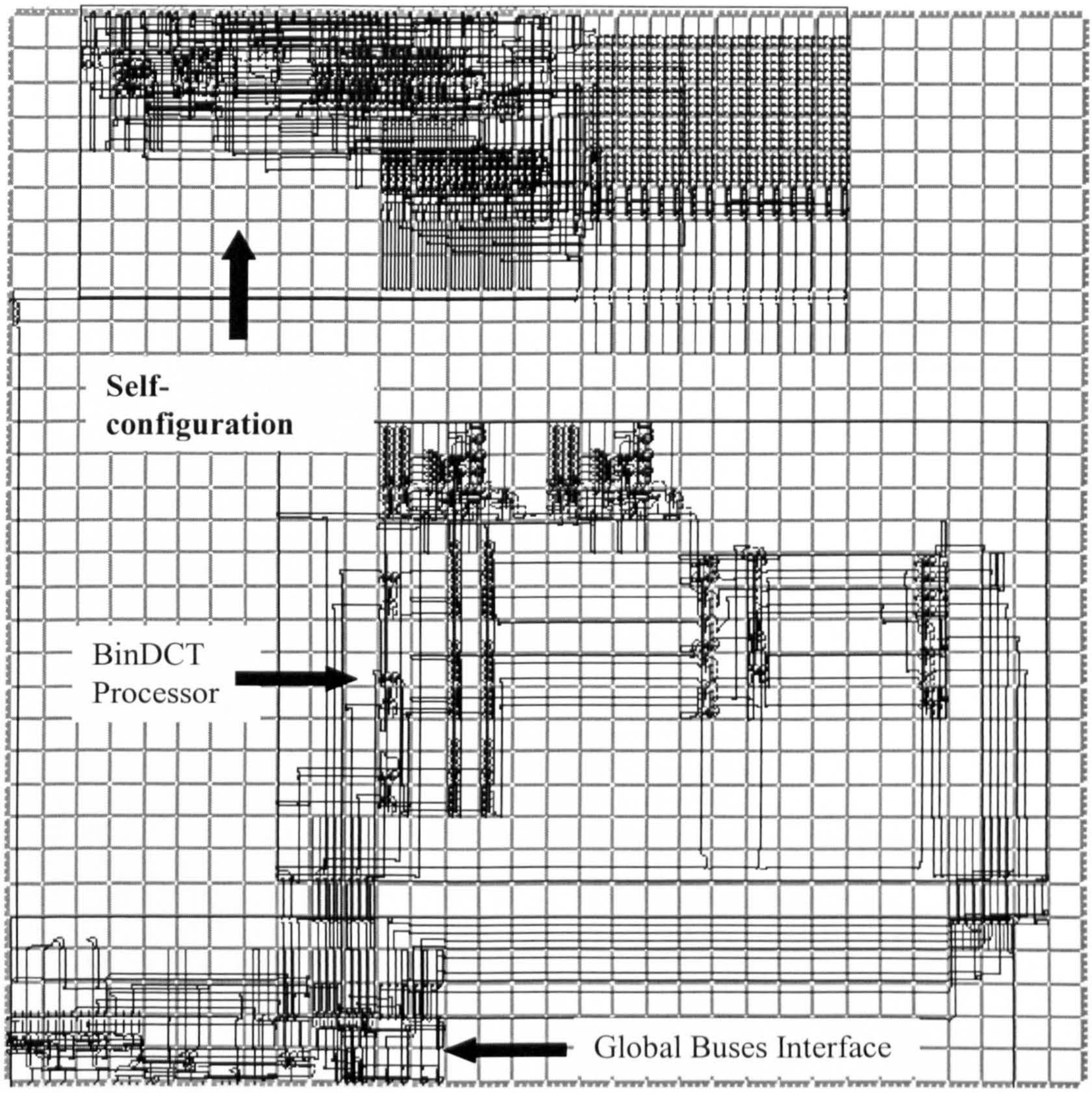


Figure VI-IV RBinDCT-C9 XC6264 Footprint

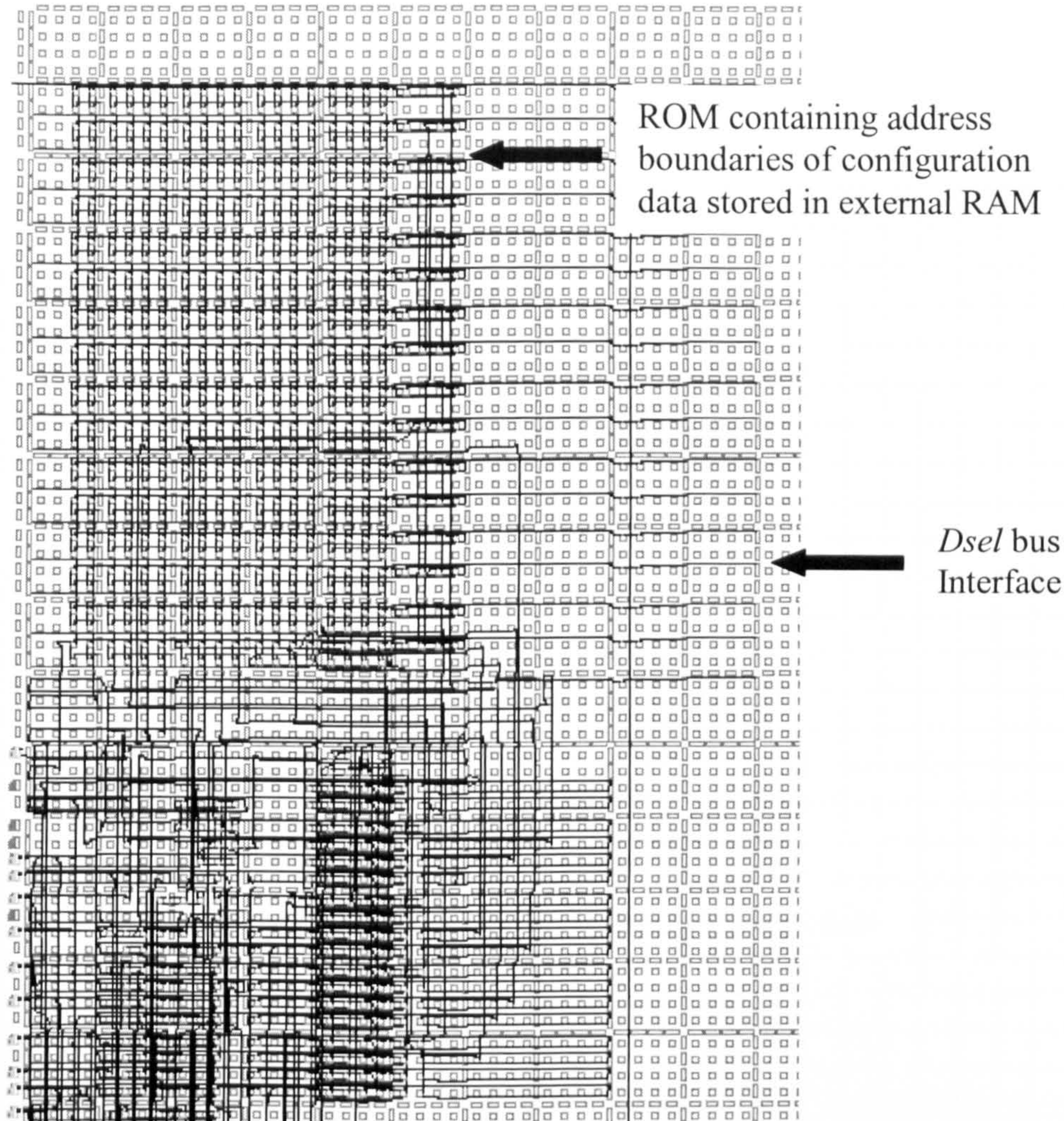


Figure VI-V Self-Configuration Controller XC6264 Footprint

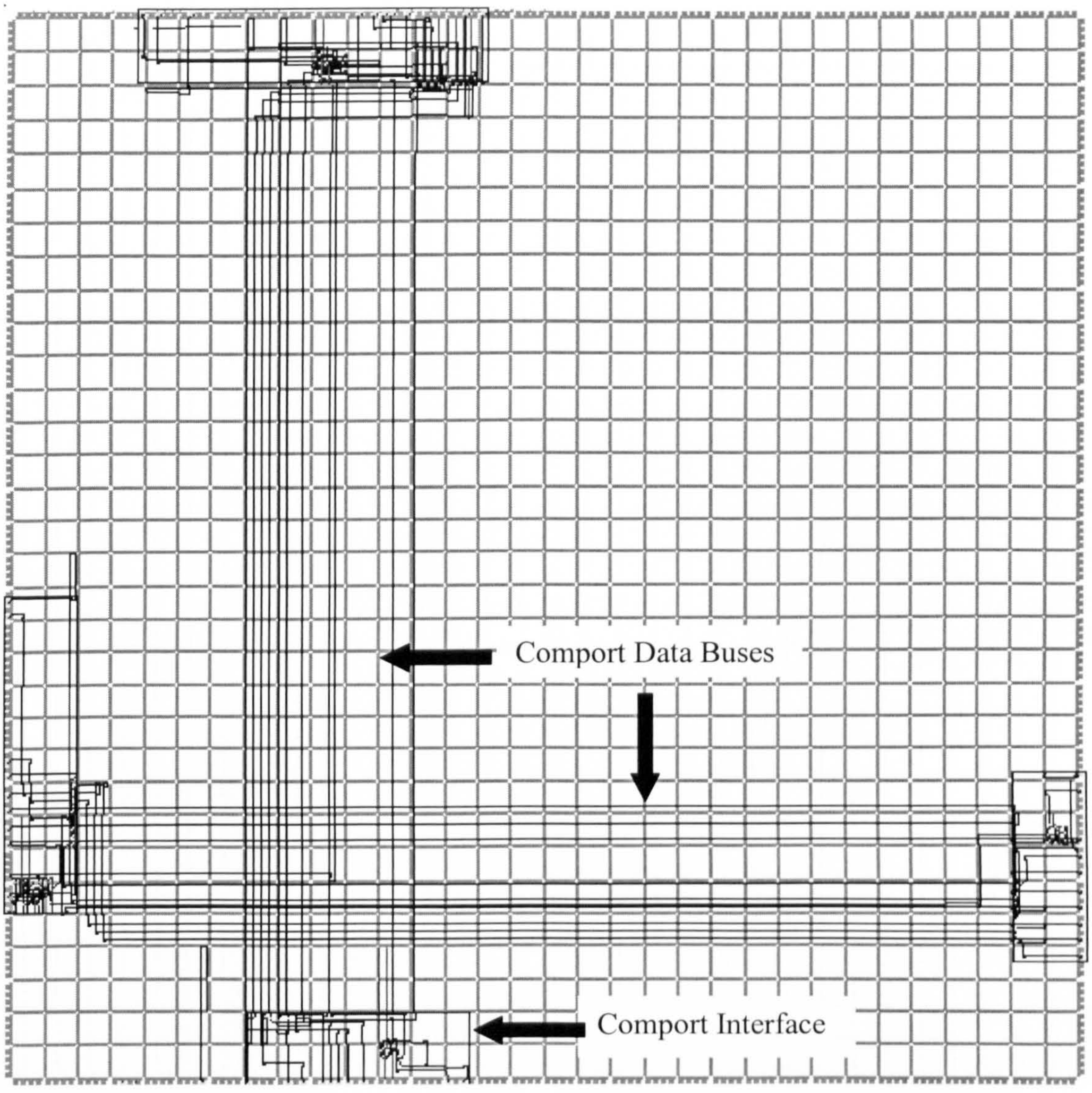


Figure VI-VI Non Structured Routing XC6264 Footprint (Configuration–1)

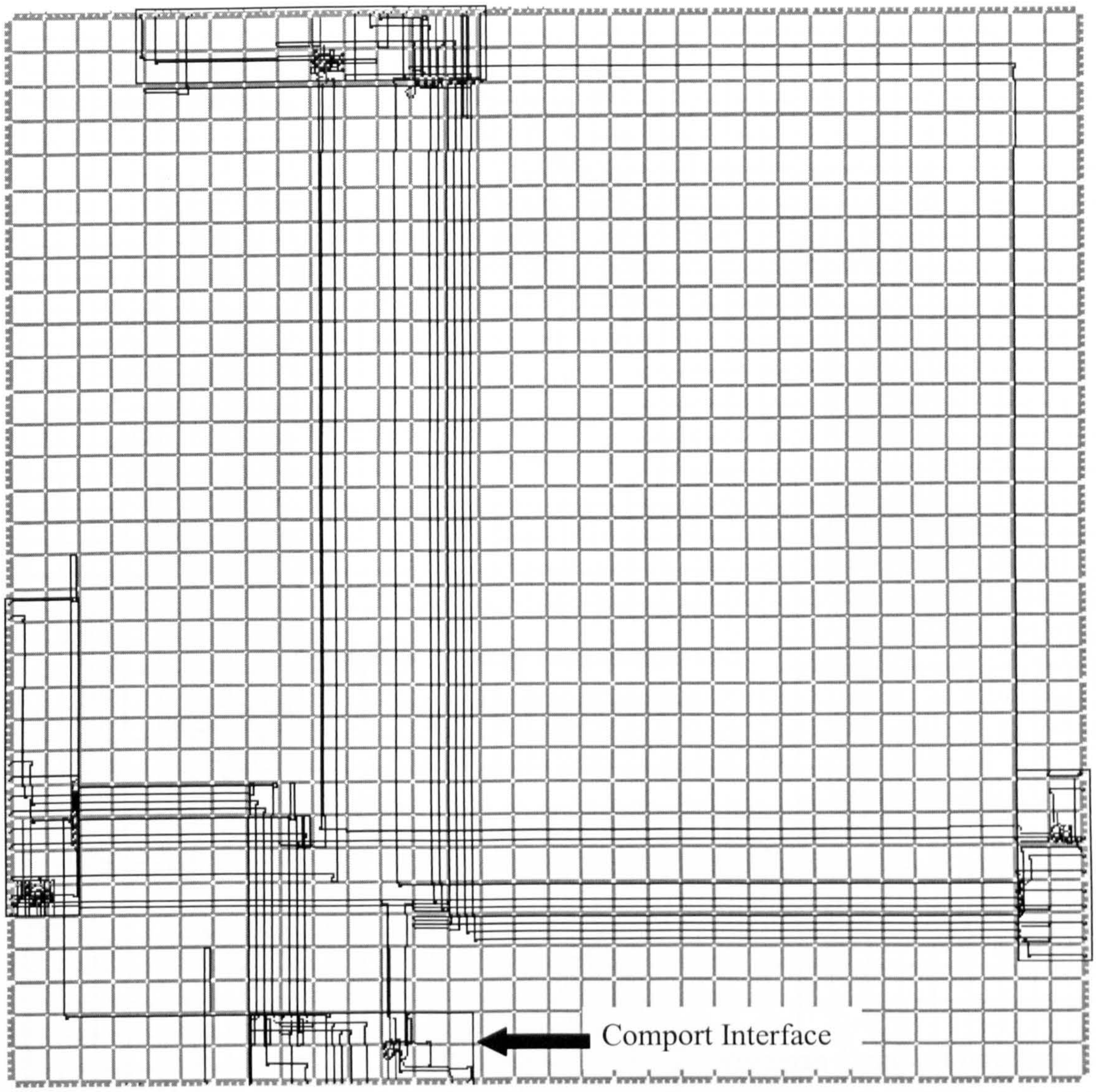


Figure VI-VII Non Structured Routing XC6264 Footprint (Configuration–2)

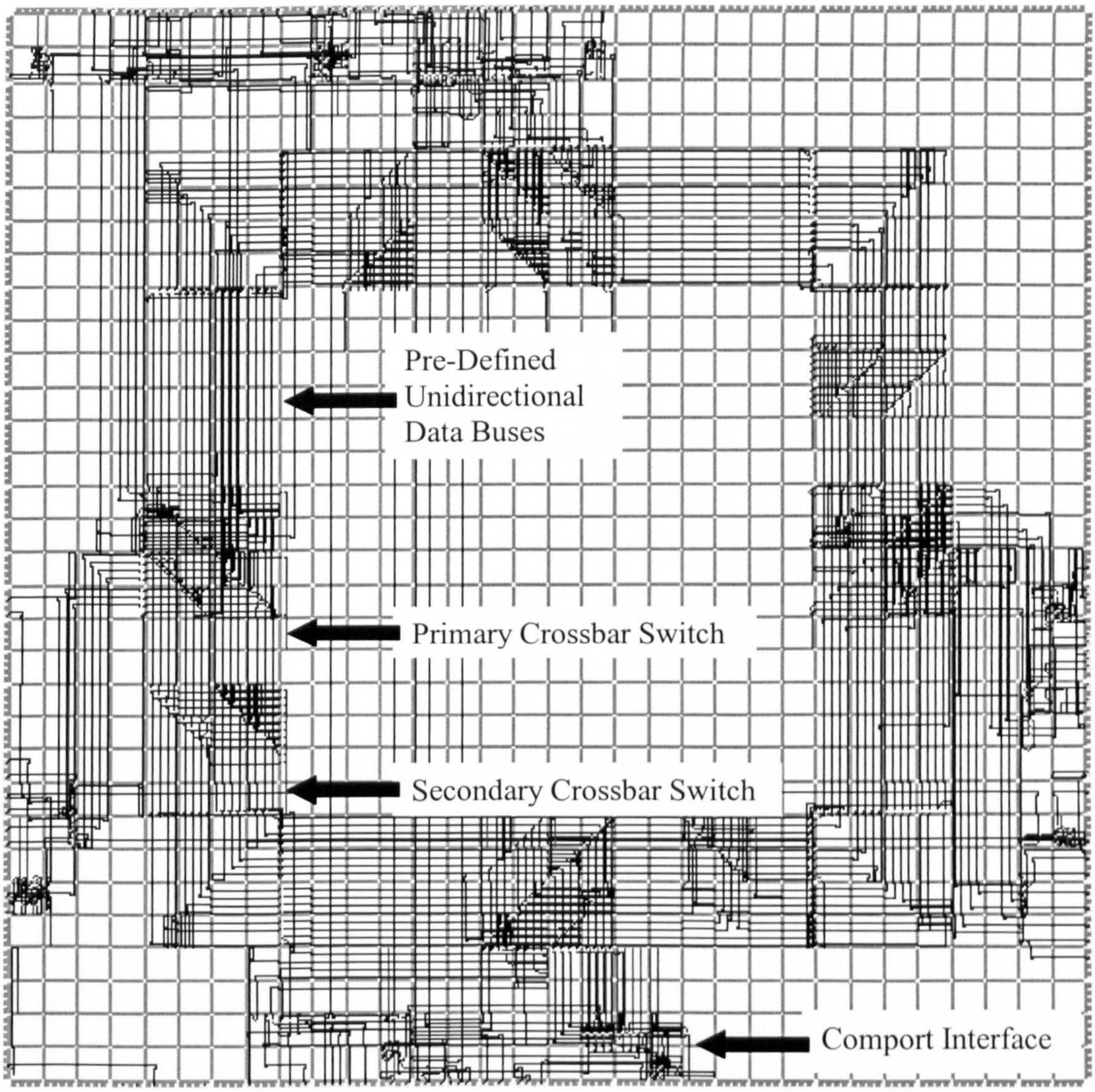


Figure VI-VIII Structured Routing XC6264 Footprint (Configuration-1)

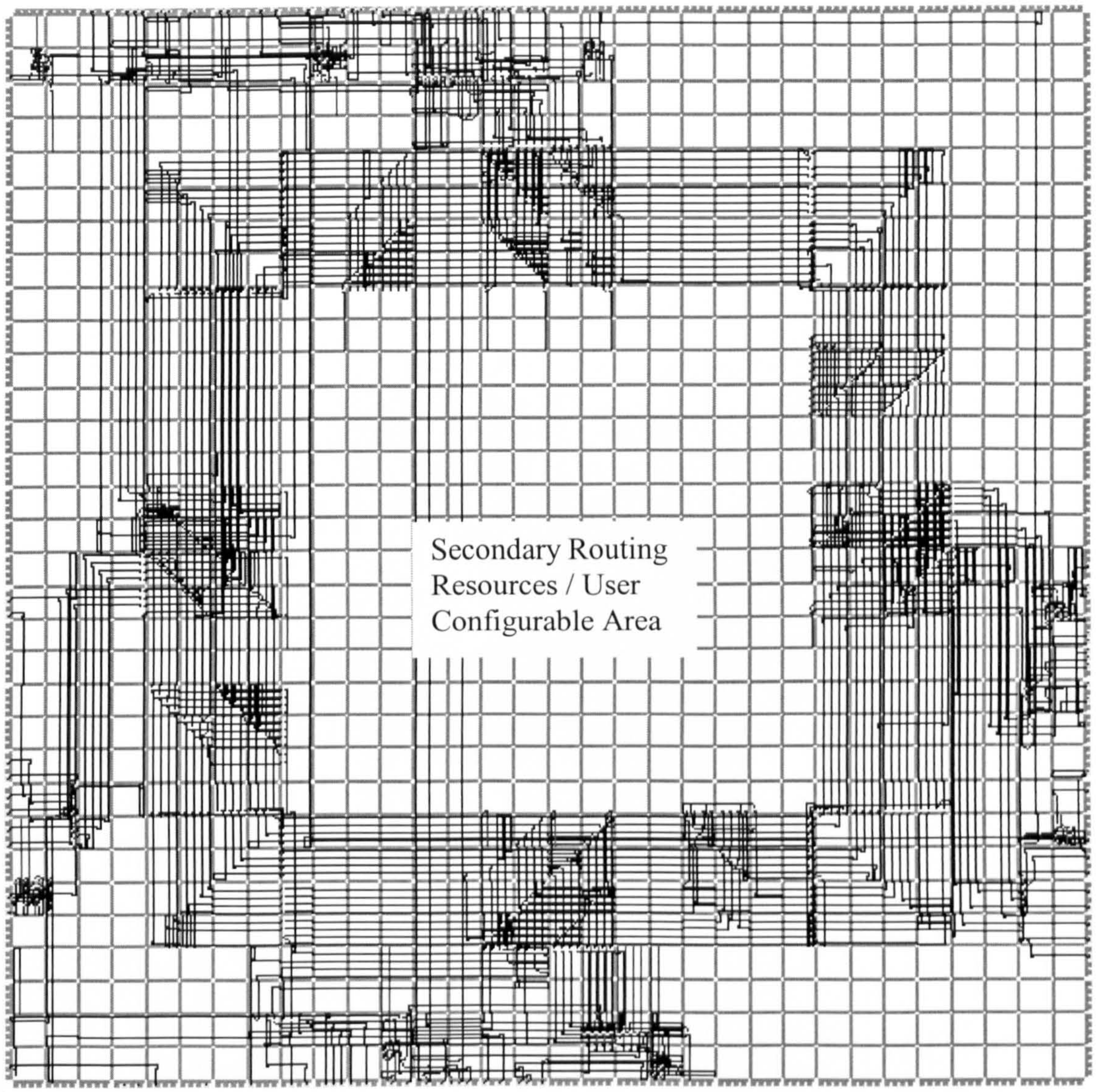


Figure VI-IX Structured Routing XC6264 Footprint (Configuration–2)

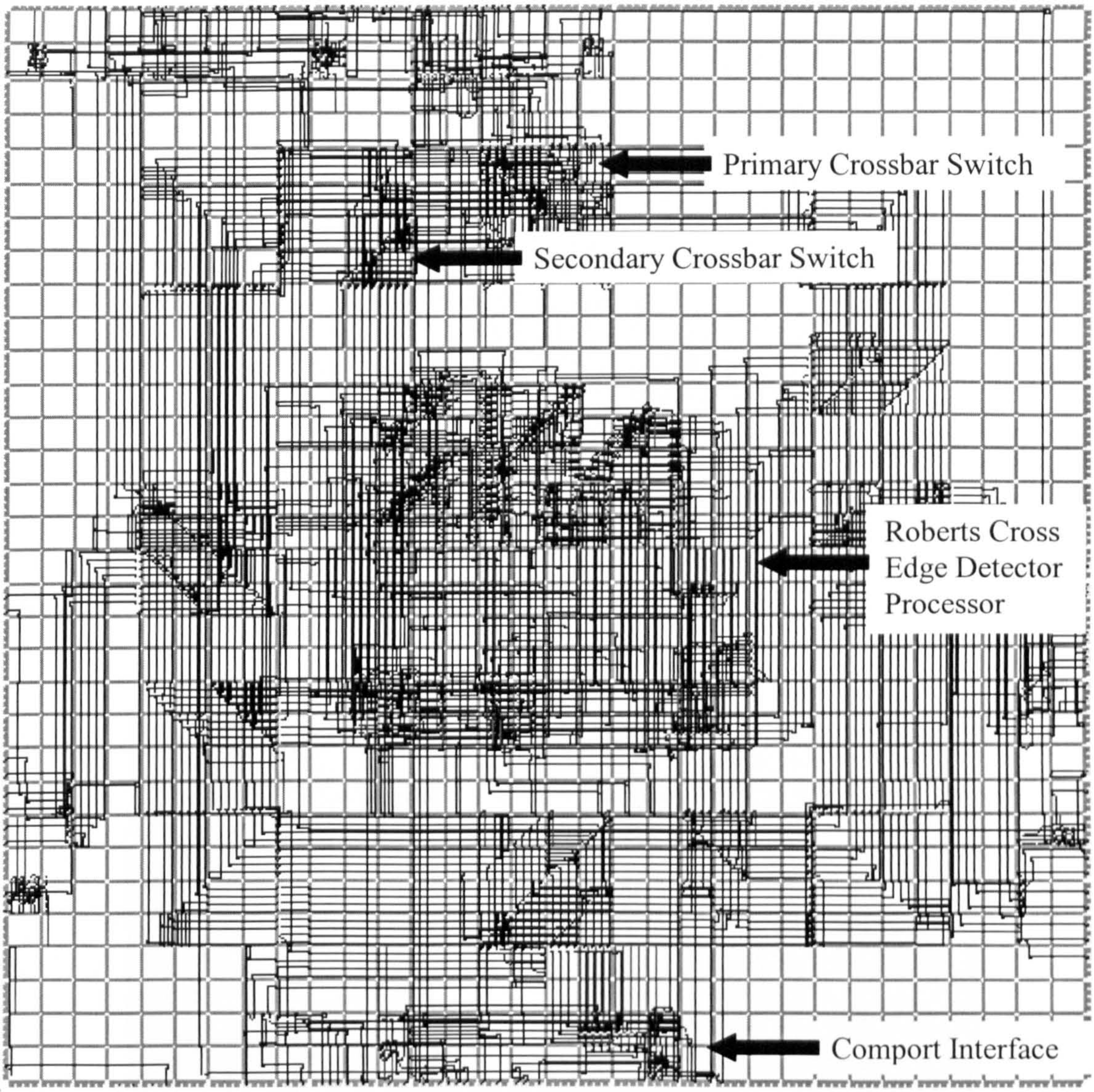


Figure VI-X Roberts Cross Edge Detector Routing Hub XC6264 Footprint

Appendix-VII

Development System Images

Figure VIII-1 **XC6264 FPGA**

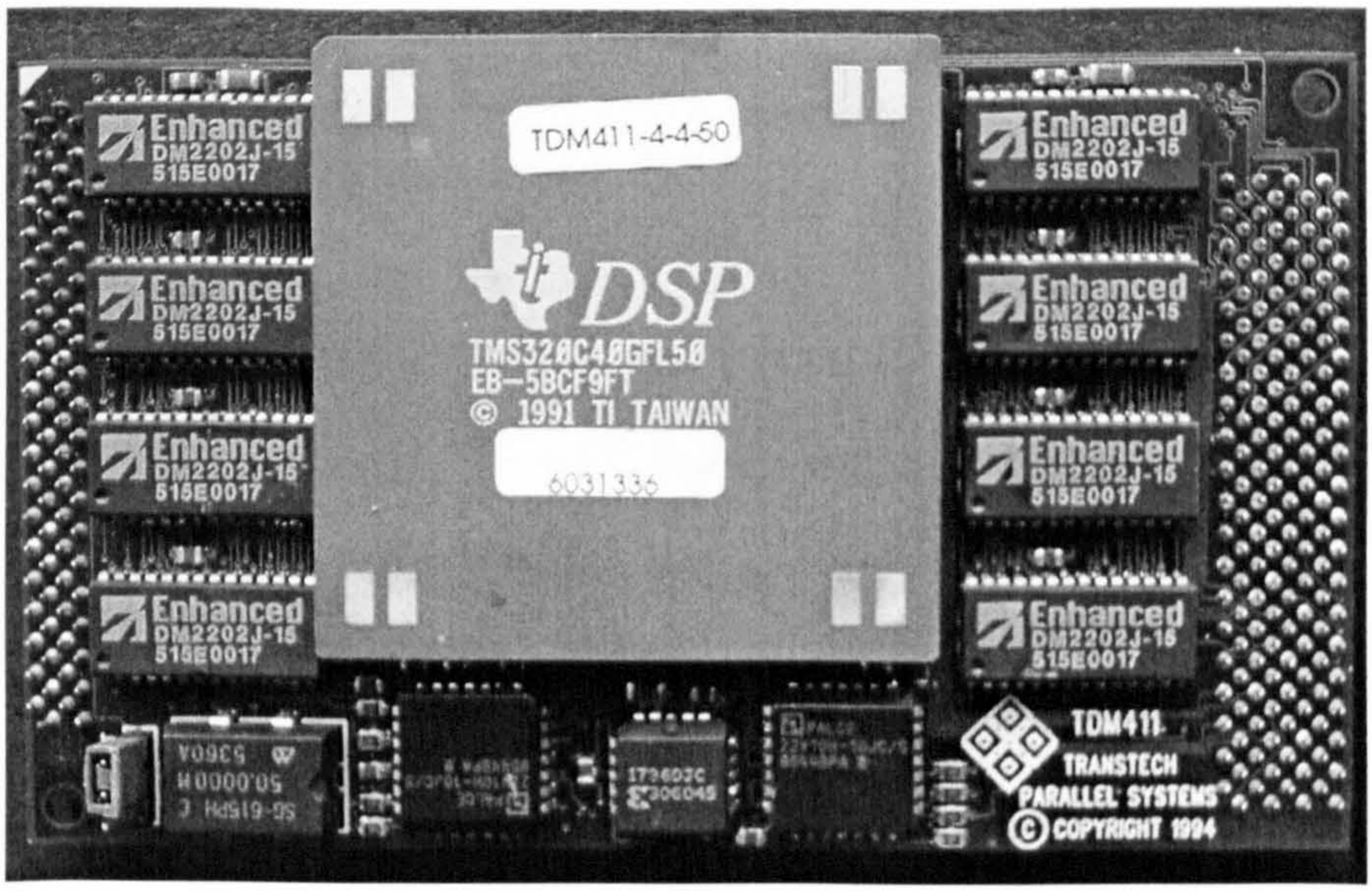


Figure VIII-2 **Transtech TDM411 TMS320C40 DSP Module**

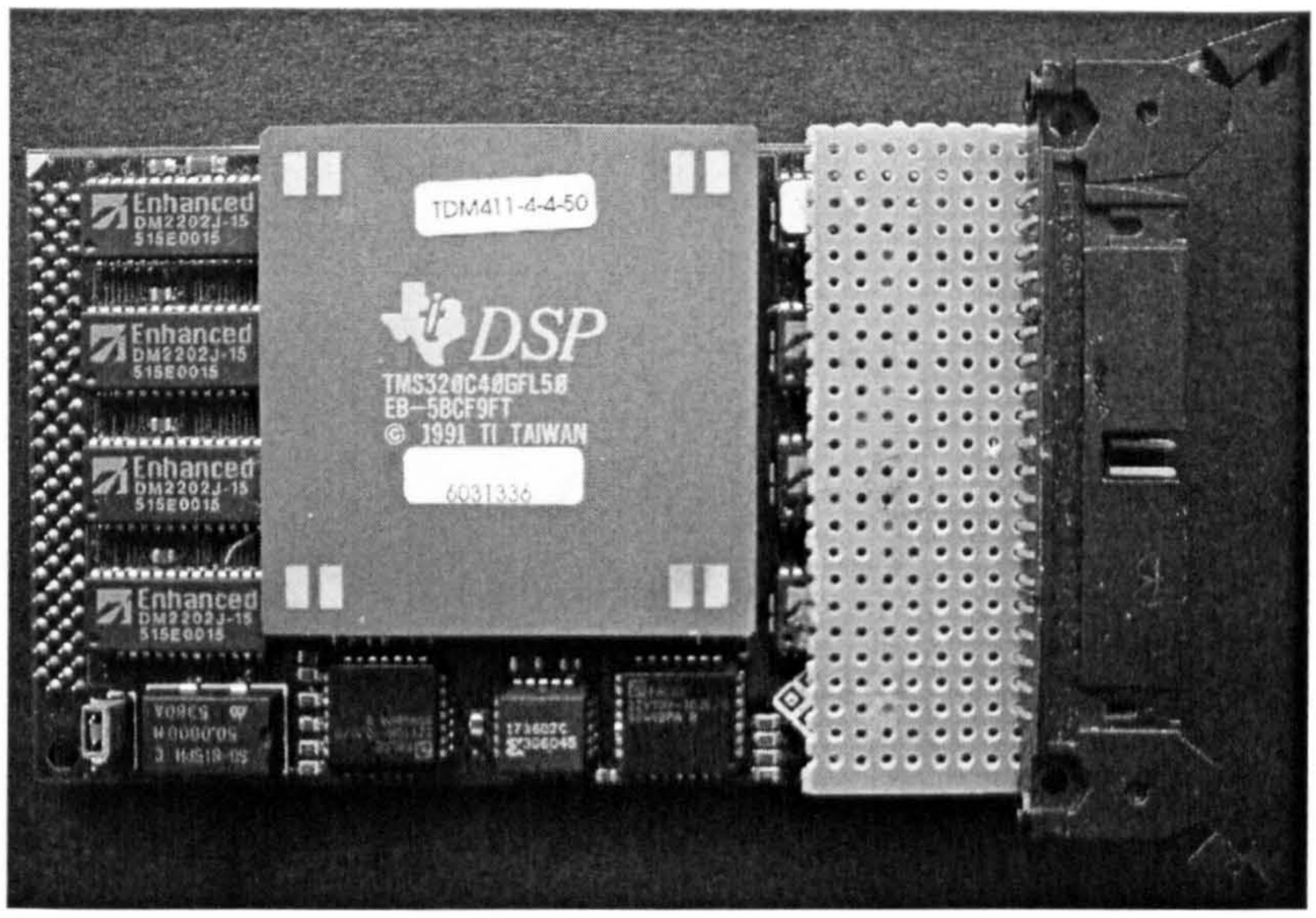


Figure VIII-2 TDM411 With XC6200DS Coprocessor Interface Connector

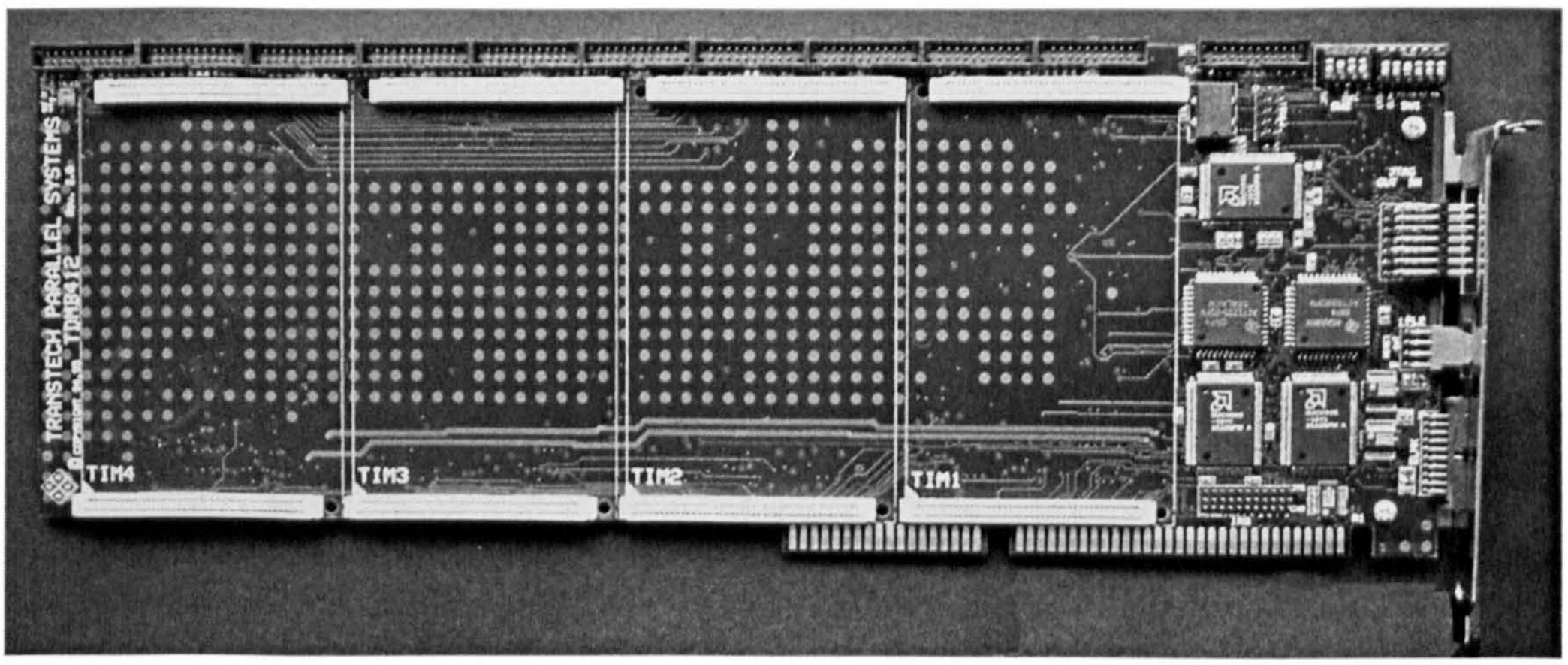


Figure VIII-3 TDMB412 Motherboard (No TDM411s Installed)

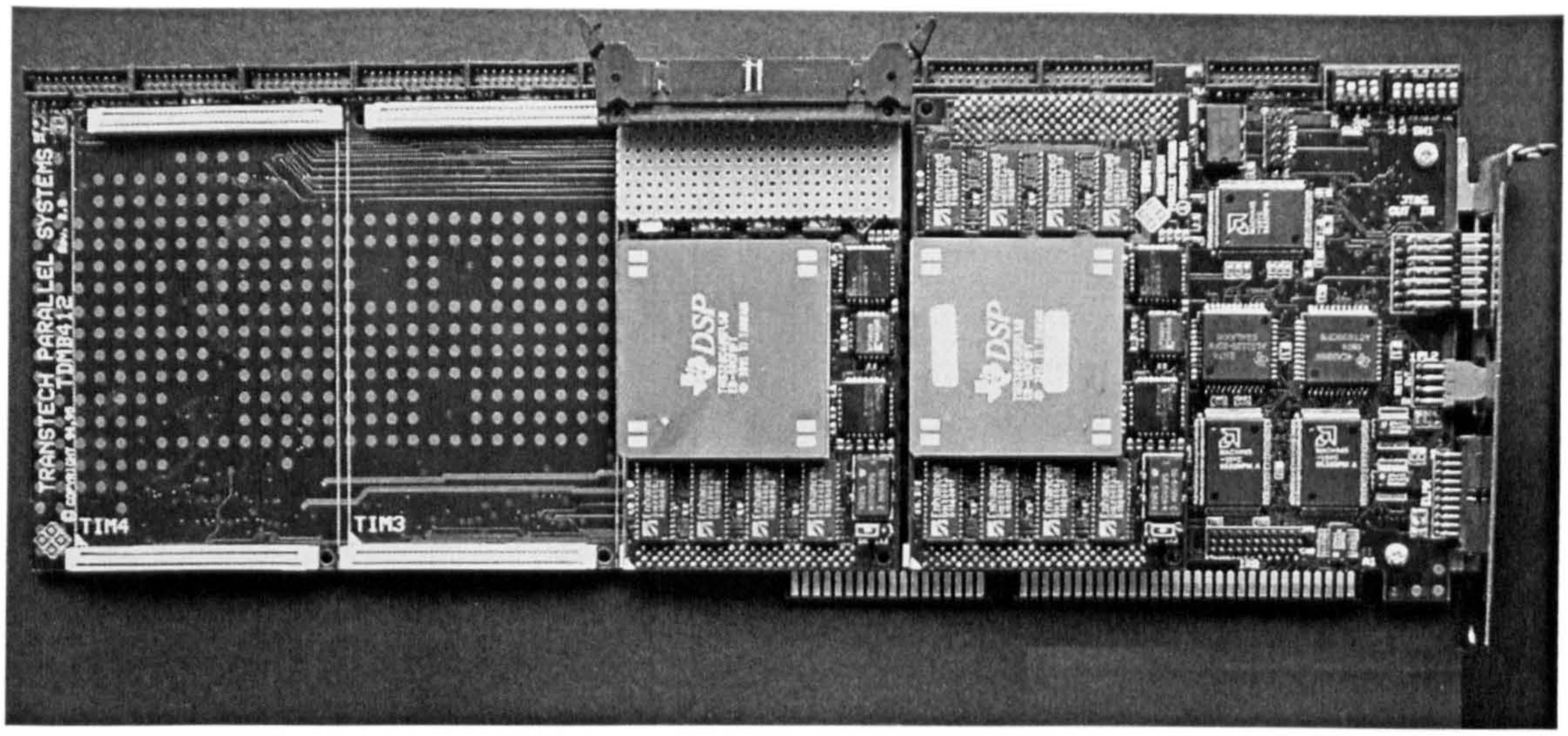


Figure VIII-4 TDMB412 Motherboard (TwoTDM411s Installed)

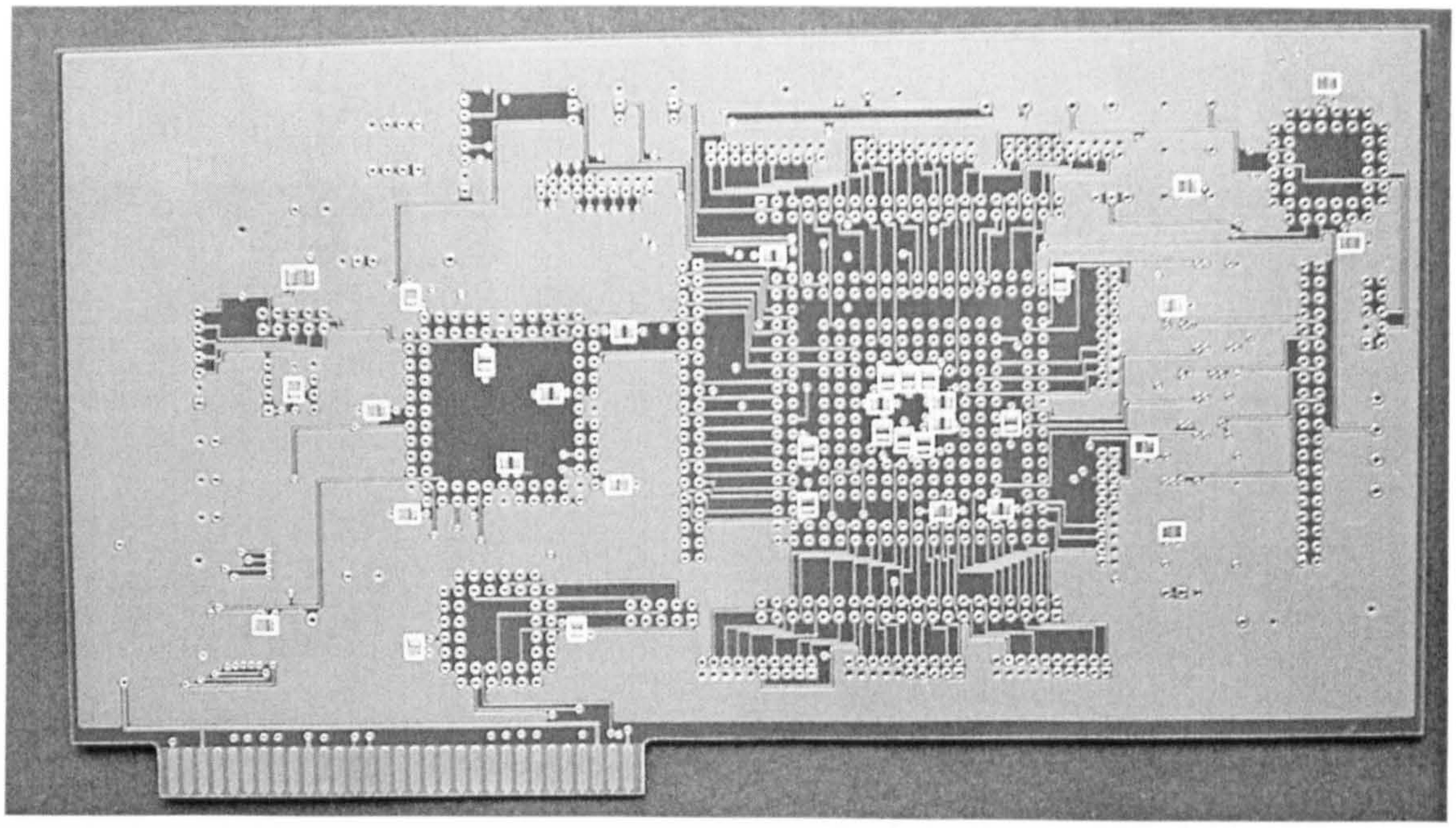


Figure VIII-5 Unpopulated XC6200DS PCB (Side 1)

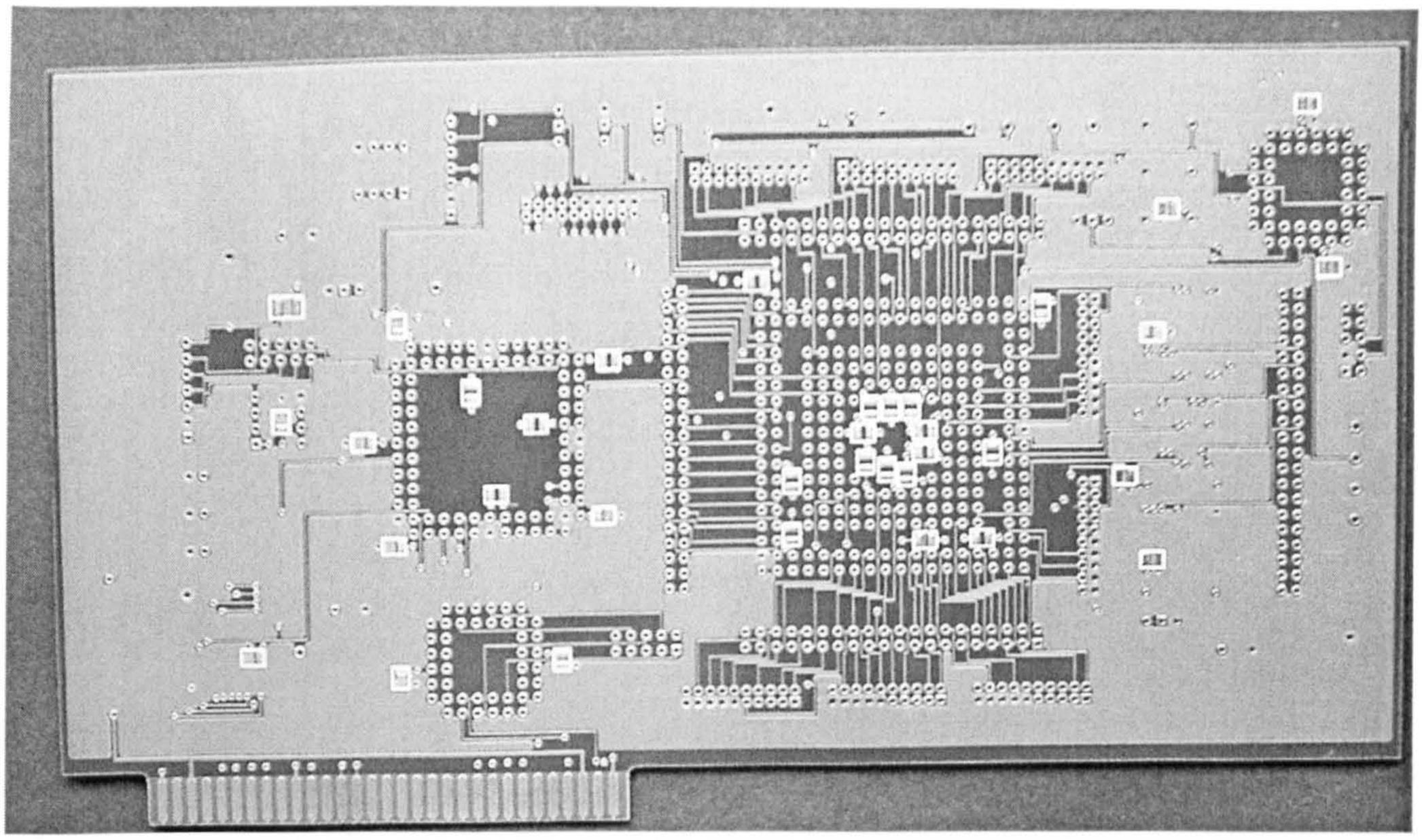


Figure VIII-5 Unpopulated XC6200DS PCB (Side 2)

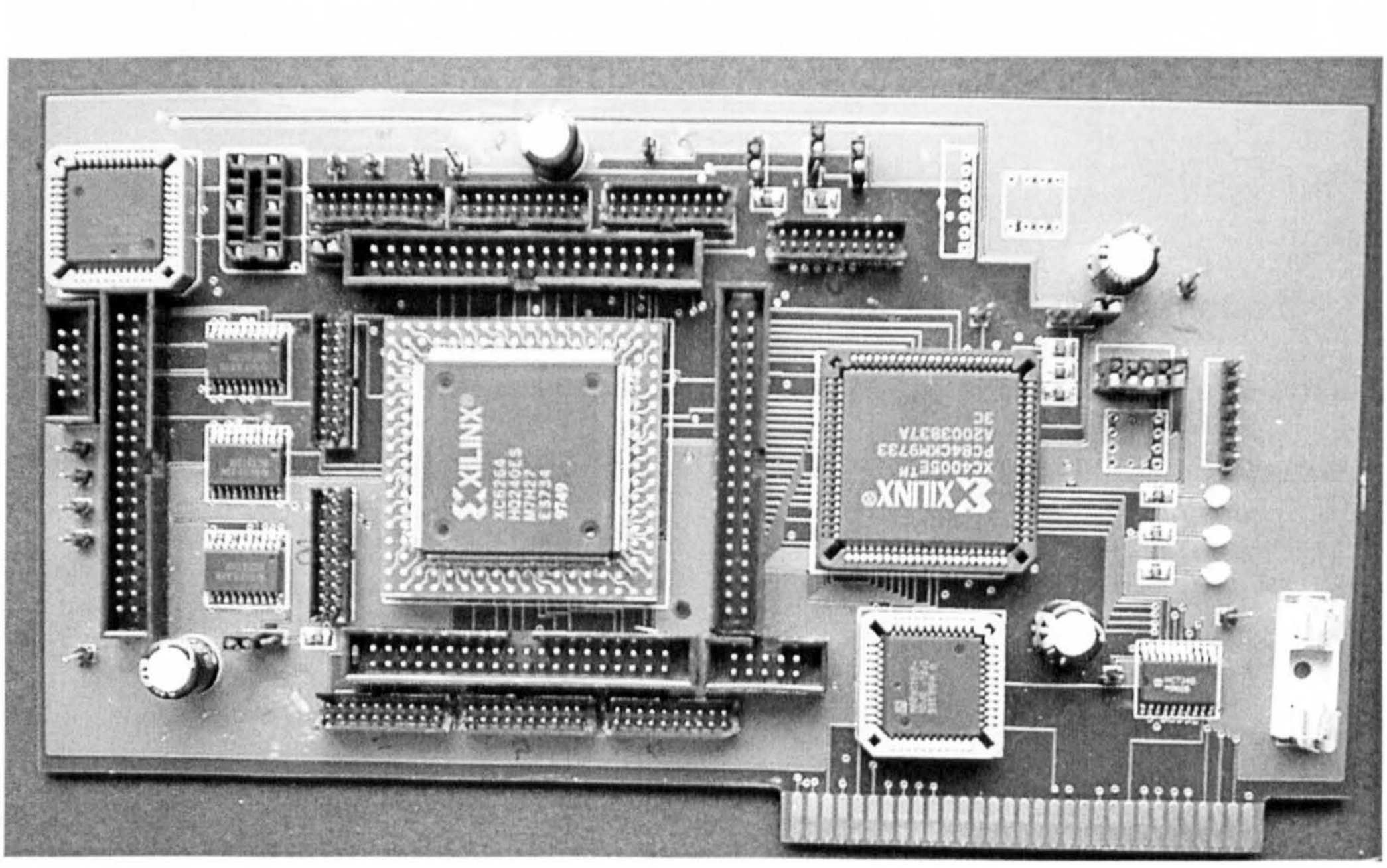


Figure VIII-6 Populated XC6200DS PCB

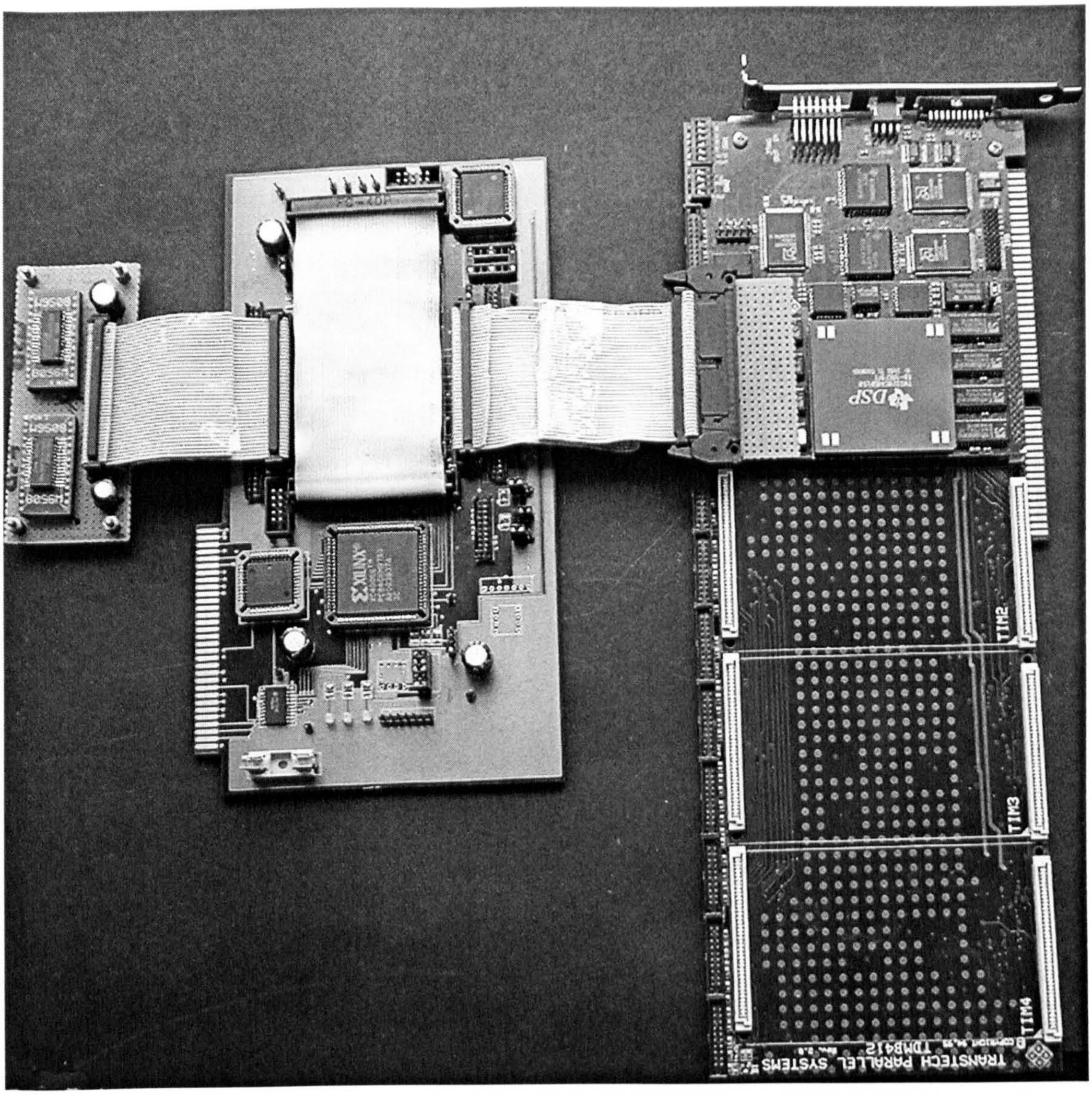


Figure VIII-7 XC6200DS/TDMB412 Dynamic Coprocessor Configuration

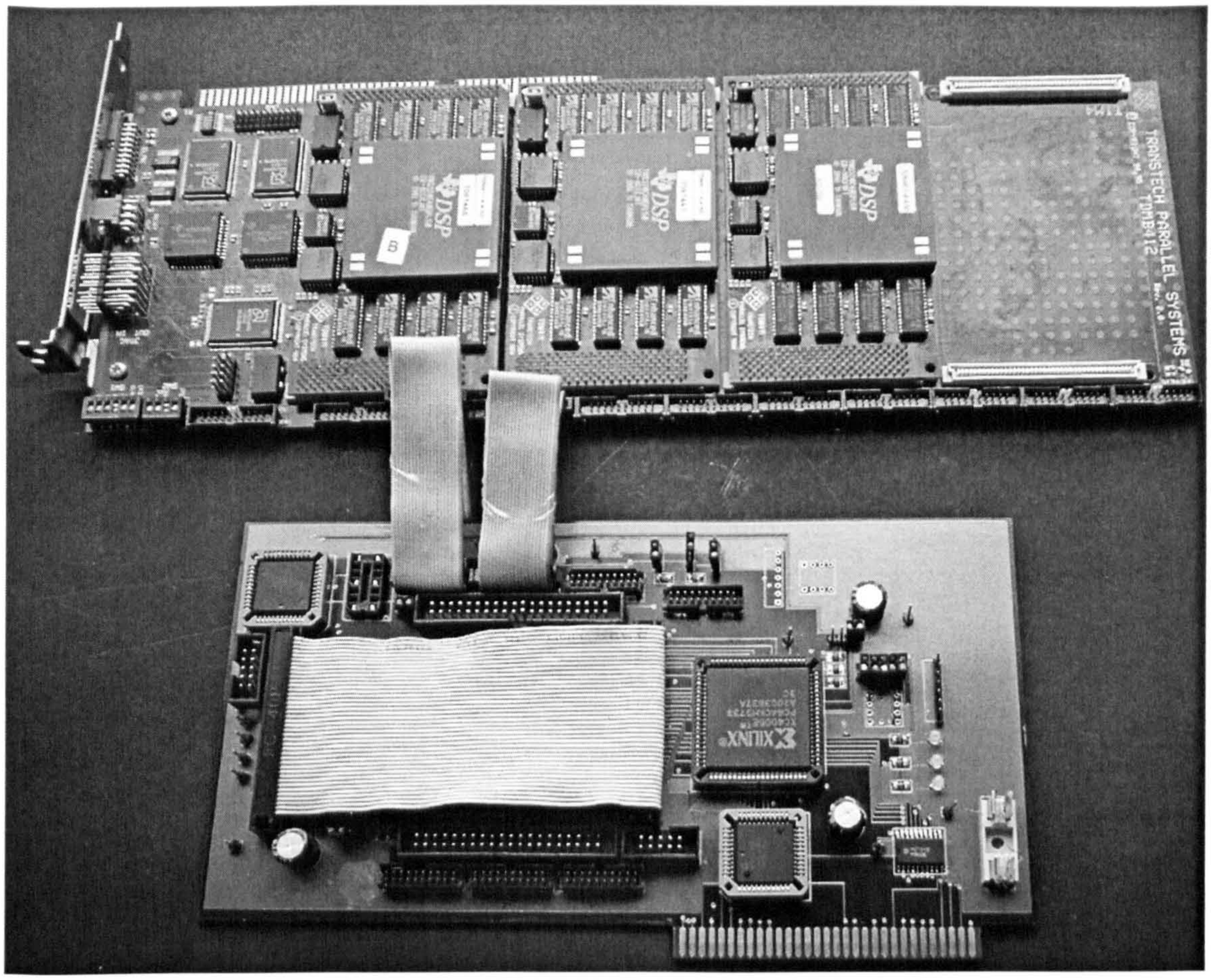


Figure VIII-8 XC6200DS/TDMB412 RTR Routing-Hub Configuration

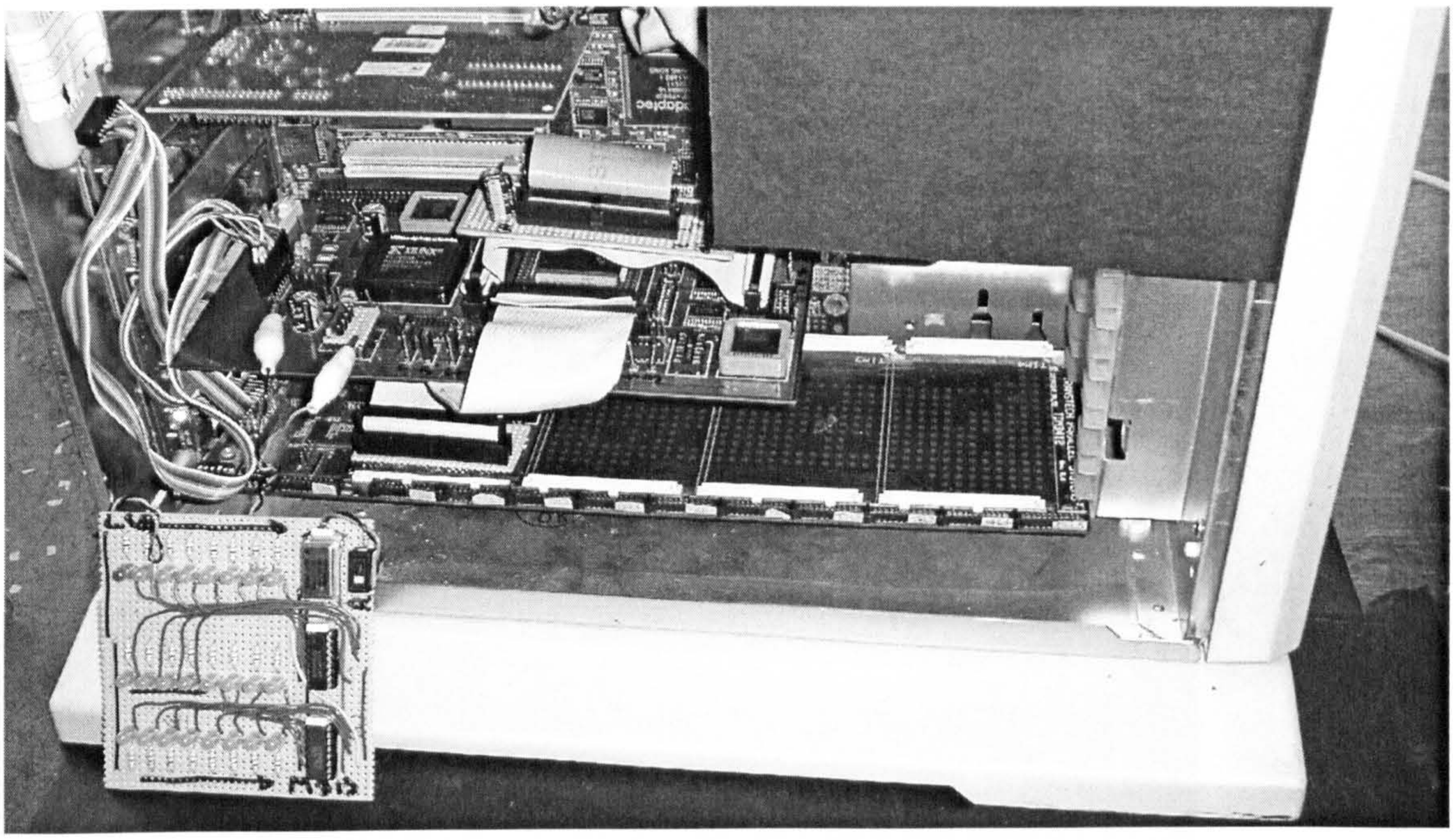


Figure VIII-9 Dynamic Coprocessor Host PC Integration

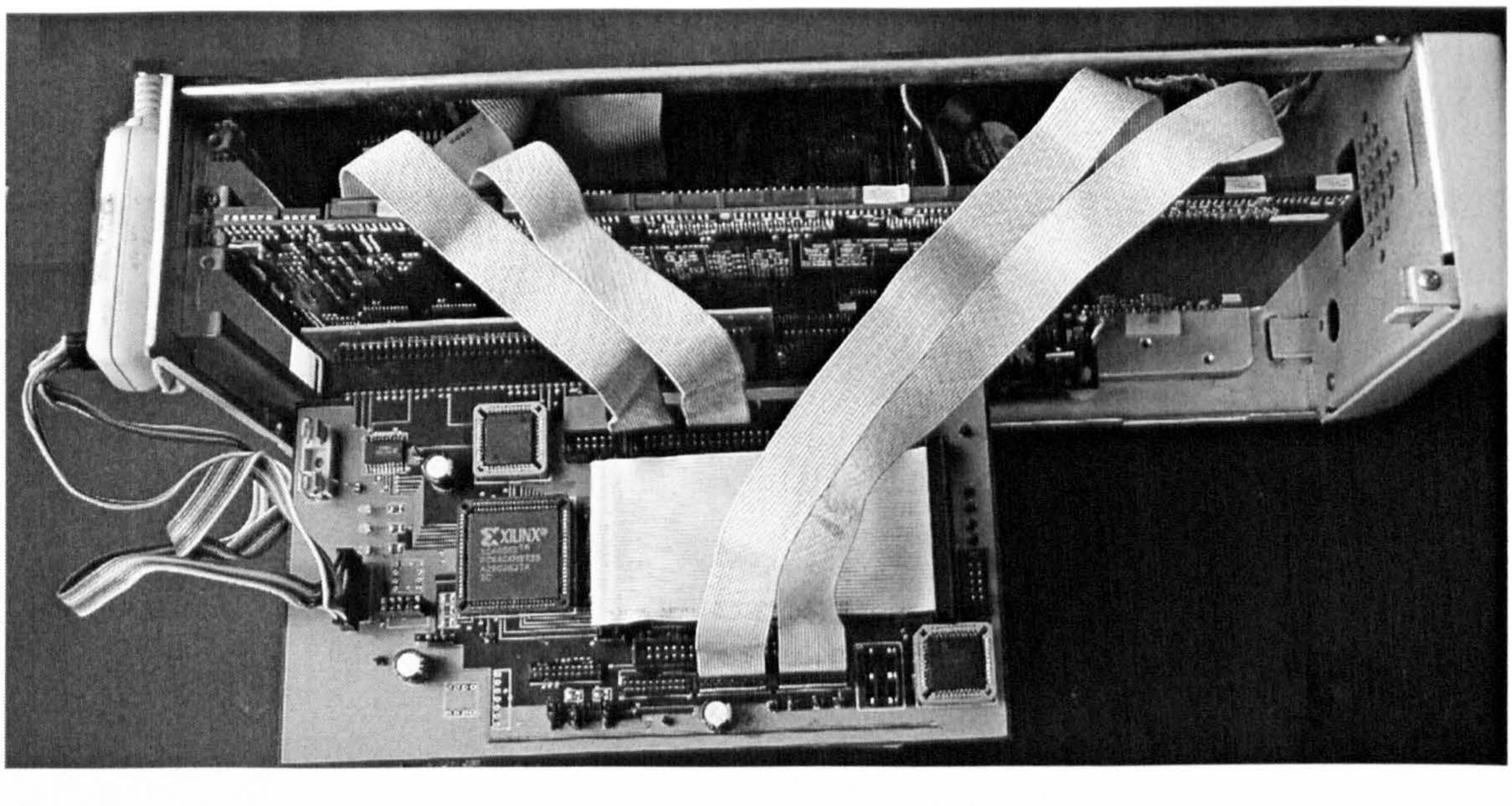


Figure VIII-11 RTR Routing-Hub Host PC Integration

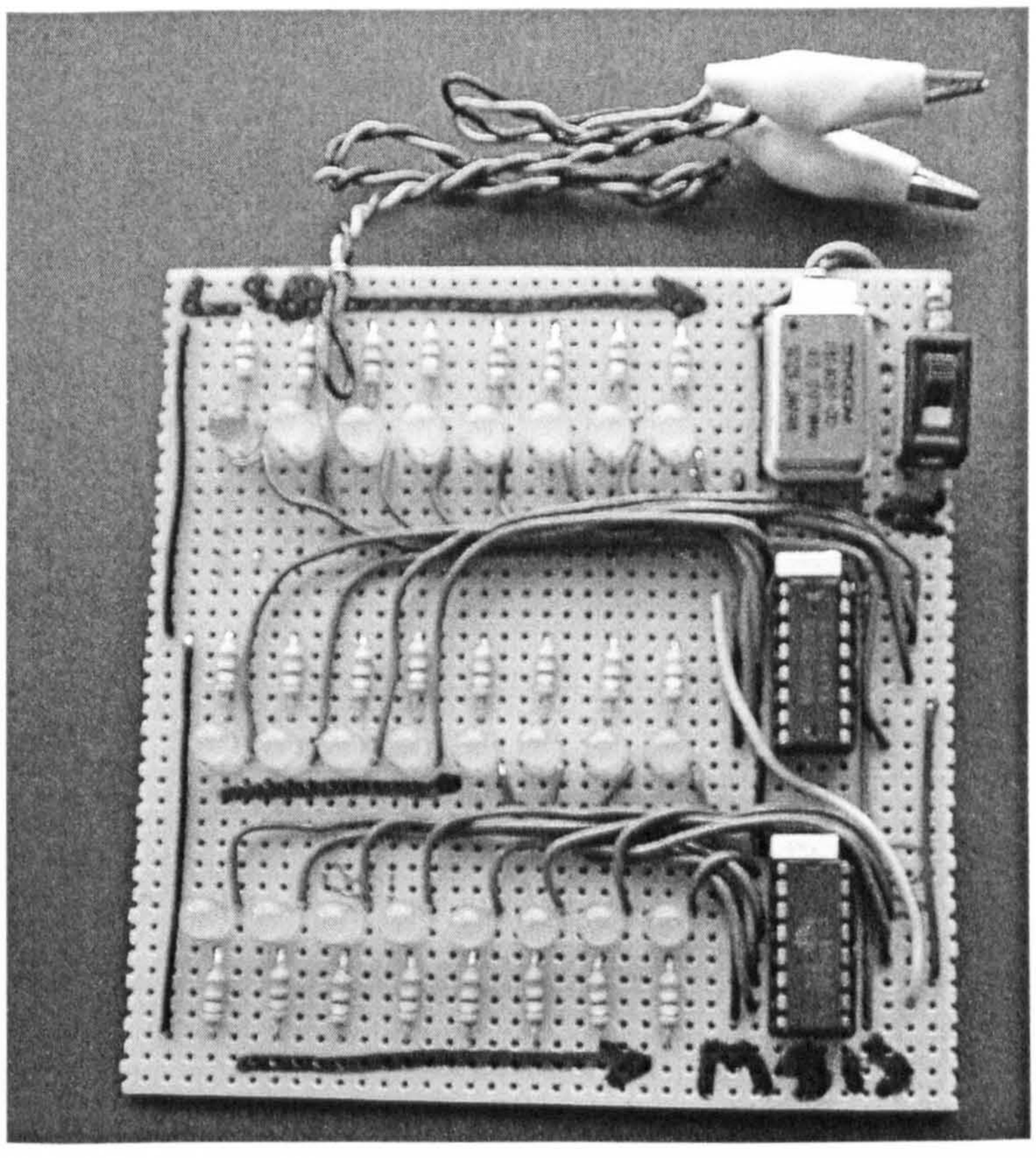


Figure VIII-12 External Dynamic Configuration Timer

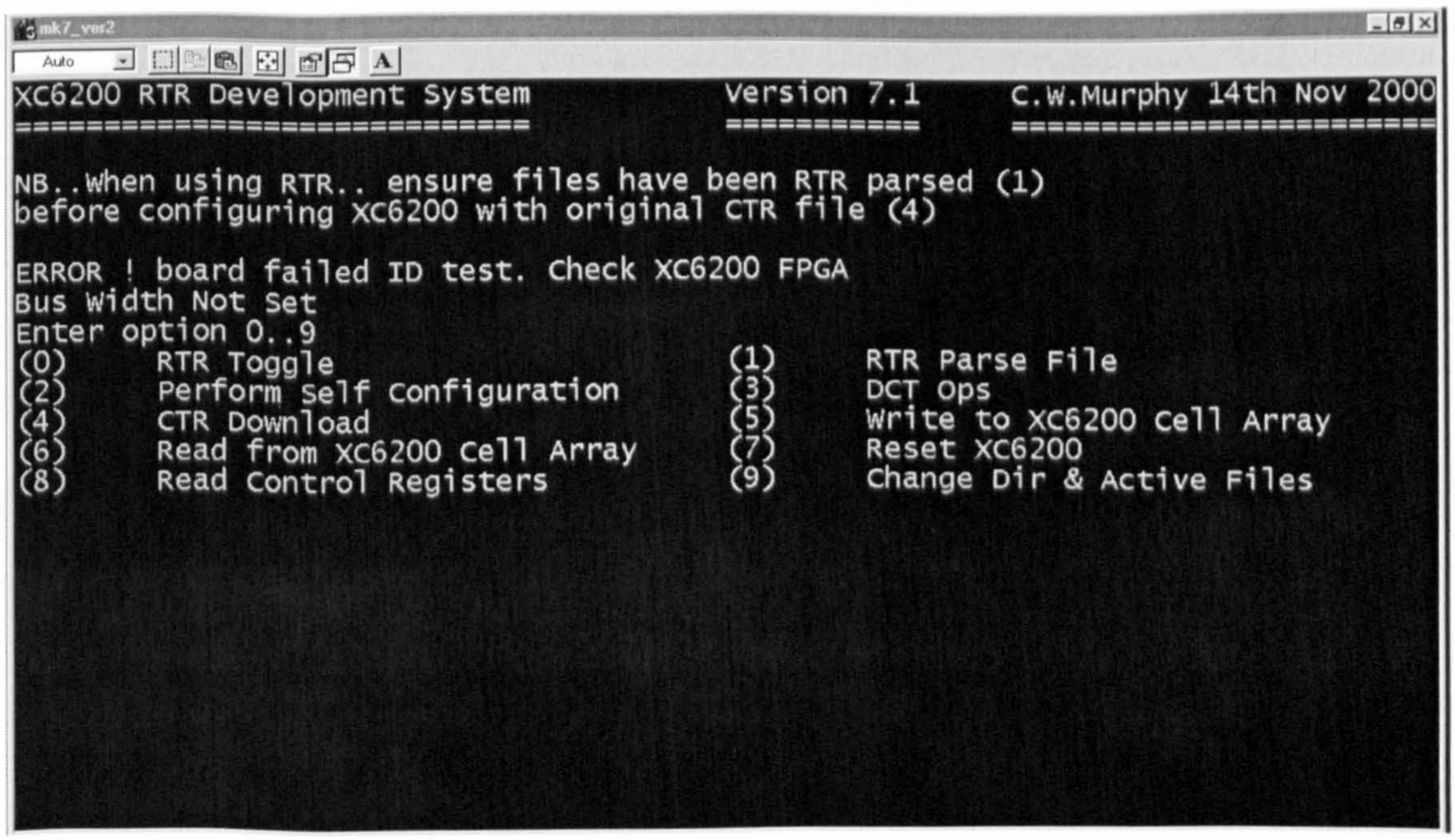


Figure VIII-13 Screen Shot of XC6200ADS

SOME PARTS EXCLUDED UNDER INSTRUCTION FROM THE UNIVERSITY