UNIVERSITY OF SOUTHAMPTON

Synthesis of Multi-FPGA Systems with Asynchronous Communications

Volume 1 of 2

by

Tack Boon Yee

A thesis submitted for the degree of
Doctor of Philosophy.

School of Electronics and Computer Science,
University of Southampton

April, 2007

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

Synthesis of Multi-FPGA Systems with Asynchronous Communications

by Tack Boon Yee

MOODS (Multiple Objective Optimisation in Data and control path Synthesis) is a high-level synthesis system which provides the ability to synthesise a system level behavioural description into a structural representation. This thesis describes an enhancement to the original MOODS system that provides an automated mechanism to target a single behavioural input design onto heterogeneous re-configurable devices, forming a multi-FPGA system. This thesis focuses on some of the problems associated with multi-FPGA synthesis, in particular the area utilisation of target devices and input/output (I/O) constraints in a multi-FPGA system.

The multi-FPGA partitioning mechanism has added a new optimisation objective into the MOODS synthesis system. Not only does it provide an automated means of partitioning the design into separate blocks, the partitioning algorithm optimises the utilisation of device area and I/O taking into account the activity profile of the design, and allows performance and I/O utilisation trade-offs to be considered. Asynchronous channel-based communication and pipelining techniques in multi-FPGA synthesis can produce a multi-FPGA system with performance close to a single-device implementation.

The contribution of this work presented herein describes multi-FPGA synthesis with the insertion of asynchronous explicit and implicit subprogram communication channels between target devices in the synthesised multi-FPGA system without any user intervention. Experiments and simulation results of test examples and a hardware demonstrator presented in this thesis provide evaluation on the performance of the synthesised non-pipelined and pipelined multi-FPGA systems with asynchronous communications. Results showed that the multi-FPGA synthesis enhancement integrated within the MOODS environment provided a rapid realisation of pipelined multi-FPGA systems with asynchronous communication channels at the expense of an acceptable increase in area overhead and design latency.

Contents

List of Figures

List of Tables

List of Acronyms

256-AES	256-bit Advanced Encryption Standard
AES	Advanced Encryption Standard
CDFG	Control and Data Flow Graph
DES	Data Encryption Standard
DFG	Data Flow Graph
ETPN	Extended Timed Petri Net
ExC	Explicit Communication Channel
FIFO	First In First Out
FM	Fiduccia-Mattheyses
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GA	Genetic Algorithms
GALS	Globally Asynchronous Locally Synchronous
ICODE	Intermediate Code
IDCT	Inverse Discrete Cosine Transform
JFIF	JPEG File Interchange Format
JPEG	Joint Photographic Experts Group
KL	Kernighan-Lin
LUT	Look Up Table
MOODS	Multiple Objective Optimisation in Data and control path synthesis
MFI	Multi-FPGA Implementation
RTL	Register Transfer Level
SA	Simulated Annealing
SpC	Subprogram Communication Channel
SRAM	Static Random Access Memory
VGA	Video Graphics Array
VHDL	VHSIC Hardware Description Language

VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration
XBM	Extended Burst-Mode

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Mark Zwolinski for his invaluable help and guidance. Mark's unbiased opinions and support were very much appreciated and I am very grateful to him for sourcing out the development boards, which made realisation of the hardware demonstrator possible.

Thanks to all members of the Electronics Systems Design Group at the University of Southampton, and also Xilinx for donating the four development boards through the Xilinx University Program. I would also like to thank the senior tutor of the school, Eric Cooke for his advice and help in administrative issues.

Thanks also go to Andrew Chapman, Dr. Petros Oikonomakos, Donald Esrafili-Gerdeh, and Dr. Matthew Sacker for the numerous discussions made on the subject of high-level synthesis and partitioning.

I would like to thank my family for their unconditional support and love. I owe much to my aunties for providing the financial support during my final few months towards the completion of this work.

My special thanks go to my good friends, Dr. Yeng Leong Chong, Josephus Tan and his wife Claire for their support over these years.

Chapter 1

Introduction

1.1 Partitioning and high-level synthesis

Partitioning is an important issue in high-level synthesis, hardware/software co-design, VLSI CAD (Very Large Scale Integration Computer Aided Design) [1, 2]. With the ever-increasing complexity of digital designs, partitioning of the circuit or system into a collection of smaller, manageable components has become a central and critical design task. Partitioning is also used to divide a large design into several target devices to satisfy packaging constraints such as input/output pins and area. Partitioning of a design over multiple hardware targets can be performed at several levels of abstraction (these include system level, behavioural level, and structural netlist level). Partitioning a design at high levels with incomplete knowledge of the targeted technology, and the final hardware (or software) implementation of a component poses a difficult design decision. The task of partitioning a system at a high level with a coarse granularity (i.e. relatively few objects with moderate to high complexities) can still be done manually, based on the experience of the designers. However, as the complexity and size of the entire system increases, this difficult decision and design optimisation problem gets harder, to the point when it gets beyond the capabilities of human designers to solve.

High-level behavioural synthesis of a digital design takes the behavioural description and translates this into an optimised structural description of the same design. The design is described behaviourally using either hardware description languages or high-level programming languages. There has been a recent interest in electronic system level (ESL) [3-5] design with new high-level synthesis tools released in 2004 from major Electronic Design Automation vendors: Mentor Graphics *Catapult C Synthesis* [6], Bluespec Inc.

Bluespec Compiler [7], and Forte Design Systems *Cynthesizer* [8]. The ESL design methodology is an evolution of high-level modelling of complex systems and behavioural synthesis, extended to address additional needs of system-level design, such as architectural design, software development and Intellectual Property (IP) exchange and reuse. Design abstraction in the digital domain has changed from schematic to language-based and is migrating towards behavioural specifications.

MOODS (Multiple Objective Optimisation in Data and control path Synthesis) is the high-level synthesis tool developed in the University of Southampton. The MOODS synthesis system synthesises an input behavioural design and produces a structural implementation of the behavioural design with the advantages of a rapid development time and design space exploration providing many alternative optimised implementations (with differing area, delay, power characteristics).

The present configuration of the MOODS synthesis system can only handle single-chip digital designs. The behavioural description of the user's digital design is synthesised into a single, large structural output. A synthesised structural output too large to fit into a selected target FPGA must instead be targeted to a larger and more costly FPGA, or be split into pieces small enough to fit into multiple FPGAs. The latter requires the user to rewrite the behavioural design, breaking the design into smaller descriptions and manually assign the structural outputs of these smaller descriptions to multiple FPGAs and connect the inter-device signals. Consider partitioning a design with 15 blocks, of which any combination of 5 blocks can fit into a target device in the multi-FPGA system, there are a total of 3003 combinations of partitioning the 15 blocks (i.e. Number of combinations, ${}_nC^k = n! / [k! * (n-k)!]$, where in this case, n = 15 and k = 5). The required design effort is becoming a major limitation to system complexity and the FPGA partitioning process needs to be automated. Synthesis of a large complex behavioural design into a multi-FPGA system poses difficult partitioning questions that need to be answered:

- How to partition the design and will the smaller partitions fit the target devices?

- Which target device should a partitioned design be assigned to and how many target devices are needed in the multi-FPGA system?

- How many I/Os are available and required to connect up the multi-FPGA system?

- How are the target devices going to transfer information (data) to each other?

This thesis focuses on some of the problems associated with multi-FPGA synthesis, in particular the area utilisation of target devices and input/output (I/O) constraints in a multi-FPGA system. In this thesis, an evaluation is made of existing multi-FPGA synthesis systems and multi-FPGA partitioning techniques. This provided an insight on the pros and cons of the various approaches and techniques, adopting the best technique or combination of techniques towards the development of our partitioning extension to the MOODS synthesis system. The goal was to extend the MOODS synthesis system to support partitioning over multiple hardware targets taking into consideration the area and I/O resources of target devices. In pursuit of this goal, we also explored asynchronous and pipelining techniques to improve the performance of a partitioned design.

The underlying hypothesis of this research is that combining asynchronous channel-based communication and pipelining techniques in multi-FPGA synthesis can fully utilise the I/O constrained FPGA target devices and the performance of the synthesised multi-FPGA implementation (MFI) will be close to a single-device implementation.

This work presents asynchronous channel-based data transfer mechanisms into multi-FPGA systems and using design activity profile to guide the proposed partitioner in reducing inter-device data transfers. Behavioural design examples and a hardware demonstrator are synthesised using the multi-FPGA synthesis in MOODS and experiments on non-pipelined MFIs with subprogram communication channels (without explicit communication channels) and pipelined MFIs with explicit communication channels are presented.

The experiments and simulation results show that the proposed channel-based approach with pipelining in a multi-FPGA systems achieve significantly better performance (in terms of reduced area overheads and design latencies) over non-pipelined implementations. Experiments on the hardware demonstrator show that the multi-FPGA synthesis enhancement integrated within the MOODS environment can synthesise a large

and complex behavioural design and target the partitioned design to a pipelined multi-FPGA system, with an acceptable increase in area overhead and design latency.

1.2 Thesis structure

The thesis consists of three main parts. The introductory chapters present the background material on behavioural synthesis and multi-FPGA partitioning. Chapter 2 introduces high-level synthesis, followed by a detailed overview of the MOODS synthesis system. Chapter 3 introduces partitioning methodologies and multi-FPGA partitioning. The chapter also reviews current research on multi-FPGA partitioning and includes a detailed discussion on synthesis systems capable of synthesising and targeting multiple devices.

The second part of the thesis, Chapters 4 and 5 describe original work, which cover the multi-FPGA partitioning enhancement of the MOODS synthesis system and the communication cells used in inter-FPGA data transfers. Chapter 4 describes in detail the automatic partitioning mechanism that partitions a single design description, and the generation of multiple structural output files for configuring a multi-FPGA system. This chapter also introduces the channel-based approach to handle inter-device data in the synthesised multi-FPGA design. Chapter 5 covers the subprogram communication channel customised for asynchronous inter-FPGA subprogram data transfers in a multi-FPGA system. The chapter describes in detail the design of communication cells and arbiter cells, which are the building blocks of the communication channel.

Chapter 6 contributes to the third part of the thesis with experimental results on multi-FPGA synthesis in MOODS. Chapter 7 describes the design, synthesis and physical implementation of a hardware demonstrator, a multi-FPGA JPEG (Joint Photographic Experts Group) decoder. Implementation results and analysis of the performance of the non-pipelined and pipelined multi-FPGA JPEG decoder are presented.

Finally, the thesis concludes with a summary of the contributions of this research and a discussion of possible future work in Chapter 8.

A number of appendices are also included in this thesis. Appendix A contains a paper published in the proceedings of International Federation for Information Processing International Conference on Very Large Scale Integration 2005 (IFIP VLSI-SOC 2005). Appendix B contains detailed information on the hardware demonstrator and a full profile of test images and photographs of the test images decoded by the multi-FPGA JPEG decoder. Post-MOODS synthesis simulation results of the multi-FPGA JPEG decoder core are also included in this appendix.

Appendix C details the format of various data files used within the MOODS synthesis environment. VHDL code listings of behavioural VHDL design examples used in the experiments described in Chapter 6 and post-MOODS synthesis simulation results of the examples are given in Appendix D. Appendix E is a brief user's guide to performing multi-FPGA synthesis using MOODS with the multi-FPGA partitioning enhancement.

Chapter 2

High-level synthesis and the MOODS synthesis system

This chapter describes the background material of high-level synthesis used within the research project. Sections 2.1 to 2.5 give a general overview of high-level synthesis. Section 2.6 describes the MOODS (Multiple Objective Optimisation in Data and control path Synthesis) synthesis system, which is used in Chapters 4 to 7 for all the implementation and multi-FPGA synthesis results of this thesis. The post-processing stage of the MOODS synthesis system is covered within Section 2.7.

2.1 High-level synthesis

Behavioural, or high-level synthesis is the process of transforming an abstract specification (such as an algorithm description) of the behaviour of the system into an equivalent structural description that satisfies a set of user constraints and goals on factors such as area, delay and energy consumption. The interpretation of VHDL [9-11] for behavioural synthesis is substantially different from that of traditional RTL (*R*egister *T*ransfer *L*evel) synthesis. In the RTL synthesis interpretation [12], the execution of an operation triggered on a clock edge within a process will complete within a clock cycle and the mapping from RTL design to gate-level design is a cycle-accurate mapping preserving the simulation semantics of VHDL. However, behavioural synthesis interprets sequential statements as if they were normal software, each of which may take several clock cycles to execute a single line of code. At the statement level, the overall behaviour of the system is unchanged, only the cycle timing of each statement is altered. This

statement-accurate mapping feature allows the synthesis system the flexibility to adjust the timing of operations and to trade off timing against other factors such as total area.

The process of transforming a behavioural description of a digital design, described in some hardware description language or sequential language such as Verilog, VHDL or C, is illustrated in Figure 2-1. The design flow consists of a number of separate tasks and different synthesis systems may perform a number of these tasks concurrently. The output of the behavioural synthesis system is a mixture of structural and RTL description of the design and it is suitable for the targeted logic synthesis and layout tools.

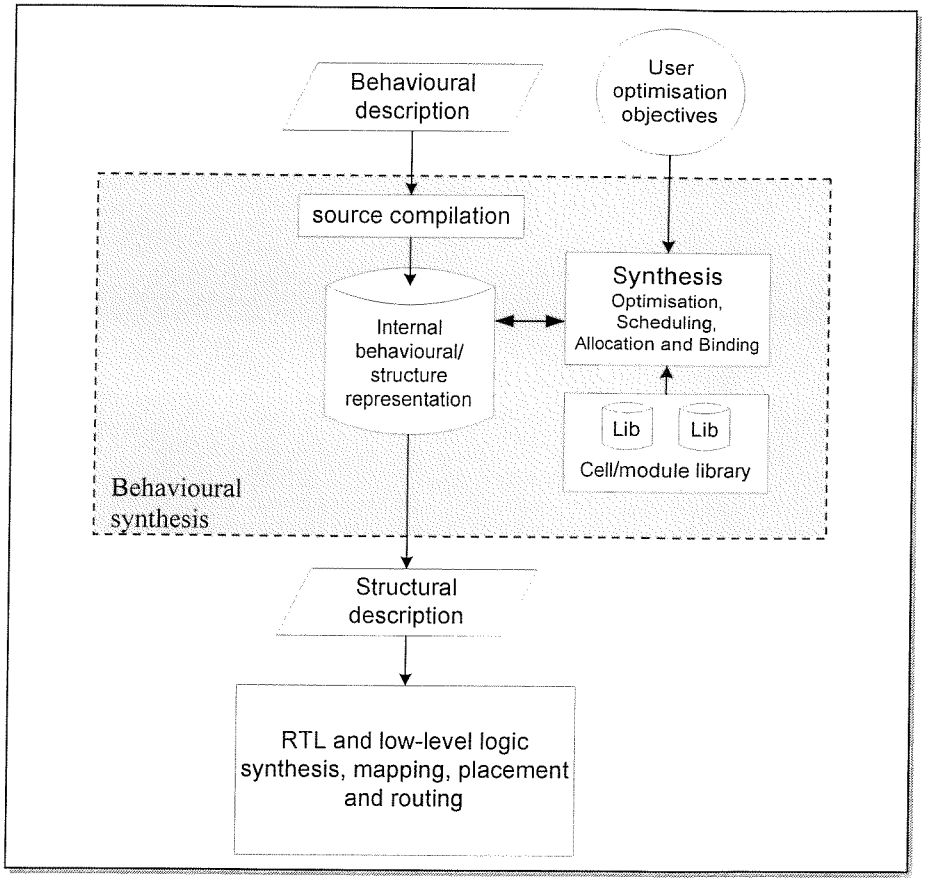


Figure 2-1 Design flow of a generic behavioural synthesis system

2.2 Input description languages

Having mentioned that VHDL can be the input hardware description language, it is only one of the many hardware description languages or sequential languages that can be used to describe a design behaviourally.

2.2.1 SystemC

SystemC [13-15] is an open C++ class library used for hardware system design and validation. SystemC is defined by the OSCI (Open SystemC Initiative) and the IEEE 1666-2005 SystemC standard [14] was ratified on December 2005. C++ or C is the language choice for software algorithms and interface specifications and most designers are familiar with these languages. The SystemC language and modeling platform provides the necessary constructs to model system architectures at various system levels of abstraction for digital design. The SystemC Class library extends the standard C++, without adding new syntactic constructs, to give hardware timing, concurrency, and reactive behaviour. This encourages systems and software designers with little or no knowledge of hardware description languages such as VHDL and Verilog to create digital designs, and to quickly simulate to validate and optimise the design according to the user objectives.

The SystemC design is compiled into an executable file and validation of the design is basically a run of the execution file. The execution of the run file is faster than a run of an HDL model that depends on the simulation and it does not require licenses as needed by most EDA tools.

2.2.2 Verilog

Verilog HDL [16, 17] is another hardware description language, other than VHDL, that is widely used, both academically and commercially. Verilog was designed in mid 1980s and the Open Verilog International (OVI) was formed in 1990 to manage the Verilog language, which was only then opened to the public domain. Verilog was later ratified as IEEE std. 1364-1995 [16]. The second ratification is the IEEE std. 1364-2001 [17], commonly called Verilog-2001. Verilog-2001 adds many significant enhancements to

IEEE 1364-1995 Verilog, which include greater support for re-usable, configurable models, Intellectual Property (IP) modeling, and very deep submicron timing accuracy [18]. SystemVerilog is the third generation of Verilog and it is built on Verilog-2001. Besides extending the Verilog language to give design features that VHDL already had in place for years [19], SystemVerilog has new constructs for verification to keep up with the increases in complexity of today's design and verification challenges [20].

2.2.3 VHDL

VHDL is an IEEE standard hardware description language [9-11, 21]. It is originally largely targeted towards simulation of digital systems at the various levels of abstraction. Synthesis use was introduced later, with the introduction of RTL (Register Transfer Level) synthesis tools first, then progressing into behavioural synthesis tools. VHDL was proposed as an IEEE standard in 1986 and it went through a number of revisions and changes before it was adopted as the IEEE 1076-1987 [10]. The first modification of VHDL was ratified in 1993 [11], and the latest in 2002 [9].

VHDL supports many different design methodologies (top-down, bottom-up, delay of detail) and is very flexible in its approach to designing hardware. VHDL provides technology independence and it contains levels of representation that can be used to represent all levels of description from the device level up to the system level. Figure 2-2 illustrates the hierarchy structure of VHDL. VHDL models a digital system using *entities* and *architectures* to define its interface and operation respectively. It is capable of describing concurrent blocks (a netlist of interconnected components) of sequential code, where the sequential elements describe the behaviour of the concurrent block at any abstraction level, via *processes*. Each design can be encapsulated by a library definition of its own interface, which highlights the ability of VHDL to describe a system in terms of modular concurrent components. A library of algorithmic descriptions can also be built from sequential blocks such as functions and procedures. In VHDL, *signals* are the only way to tie together elements of structural descriptions or to pass information directly between VHDL processes and entities, VHDL signals are declared in the VHDL architecture. VHDL *variables* are local to process and they are declared within VHDL processes.

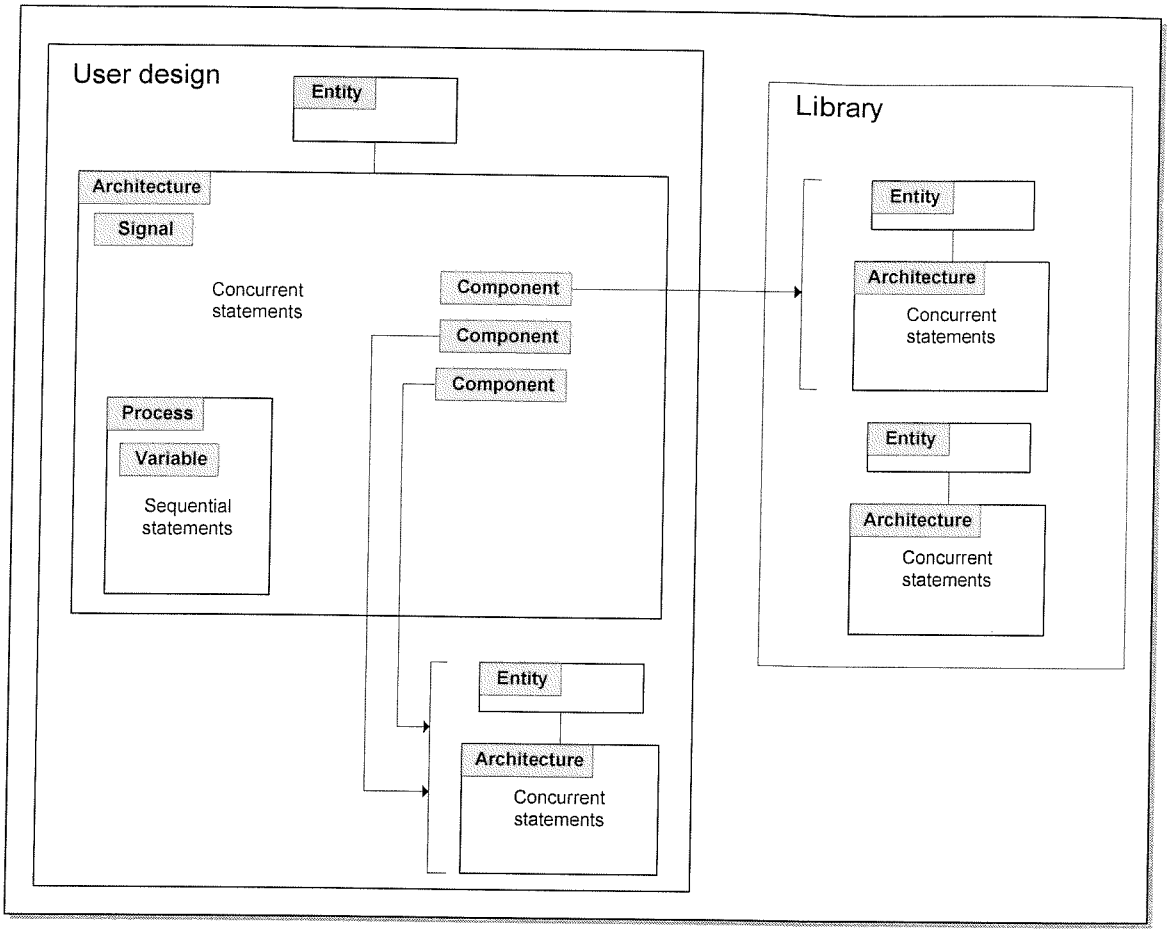


Figure 2-2 VHDL hierarchy structure

2.3 Compilation and internal representation

- *Behavioural input compilation and optimisation* – is the first synthesis task and it is concerned with the compilation of the behavioural description into an internal representation to which synthesis operations may be applied. A number of compile-time optimisations (procedural inlining, dead code elimination, loop unrolling) [22] may be performed. The result of compilation is the generation of a design specified in terms of a number of simple instructions, similar to a machine-readable software assembly language, often in some form of abstract data and control flow graphs.

The synthesis optimisation process is either performed during the construction of the data structures or during an iterative refinement process after the initial data structures are

created, or a combination of both methods. A number of behaviour-preserving internal data structures can be used to fully describe the design throughout the synthesis process. A simple *dataflow graph* (DFG) can be used to describe the system. The operations and data dependencies of a simple design (with no conditional and iteration constructs) can be represented with a dataflow graph as illustrated in Figure 2-3.

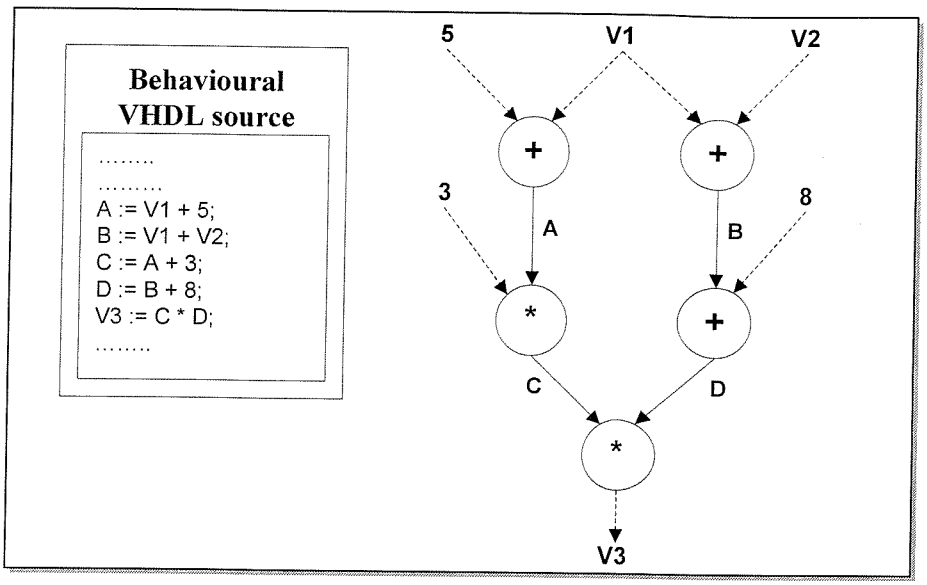


Figure 2-3 Data flow graph representation

In general, many applications contain a significant amount of conditional (*if-then-else*) and iteration (*loop*) constructs, and thus requires a more comprehensive representation of data, and the control flow information. The *Control and Data Flow Graph* (CDFG) models both the data and control flow information in a single hierarchical structure. This is done by extending the DFG representation to encapsulate control flow information for blocks of DFG sub-graphs within the parent graph. A major disadvantage of the CDFG representation is the basic blocks of DFG sub-graphs provide boundaries across which the scheduling of operations cannot pass, even if there are no dependencies restricting this schedule [23]. On the other hand, the *Extended Timed Petri Net* (ETPN) [24, 25] representation has no such block boundary restriction as the data and control flow is separated into two individual, but interrelated data structures, hence allowing more optimisation transforms to be performed. This separation of the control flow also makes the ETPN more suitable for designs with concurrency execution of operations and asynchrony inherent.

By way of an example, the associated CDFG and ETPN representations of the fragment of VHDL code are illustrated in Figure 2-4 and Figure 2-5 respectively. The CDFG in Figure 2-4 comprises four DFGs. The top one represents the three sequential addition assignments, the second two graphs represents the two conditional assignments, and the last DFG represents the last multiplication assignment.

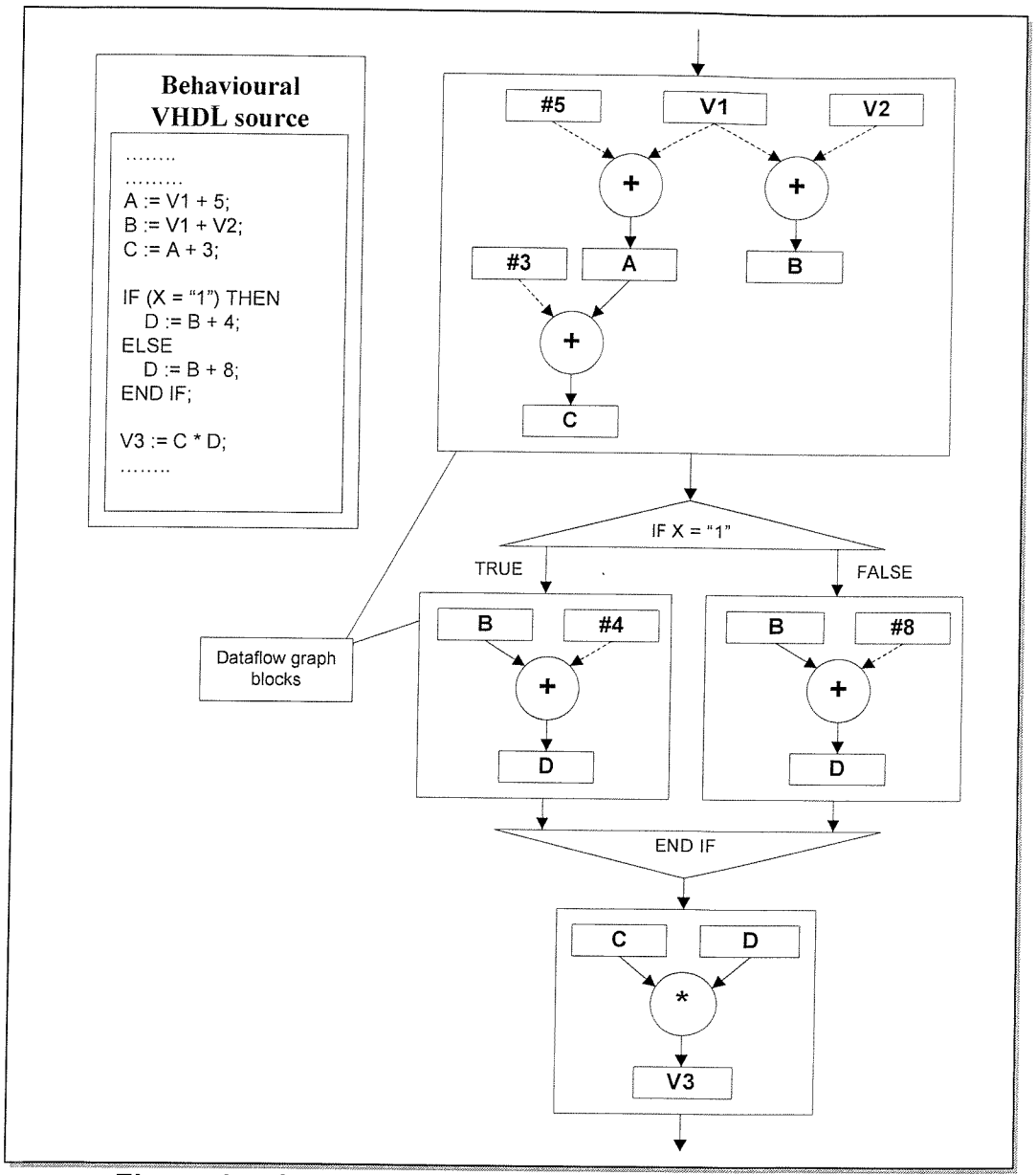


Figure 2-4 Control and data flow graph representation

ETPN represents the data path as a directed graph with nodes and conditional arcs. The nodes represent individual functional (operator) and storage (variable) units, while the arcs form the connections between nodes. These connections are only made if the arc is

activated by a control signal (S_n signals generated from the control part). The control flow of the design is described by the passage of tokens through a Petri net [26]. Each control path vertex represents a control state, which is activated when it receives a token, thereby activating the associated data path via its S_n signal. In Figure 2-5, the conditional block (if X = "1") is modelled by states S_4 and S_5, and the selection is based upon condition C_1, generated from the data path comparator (=) block.

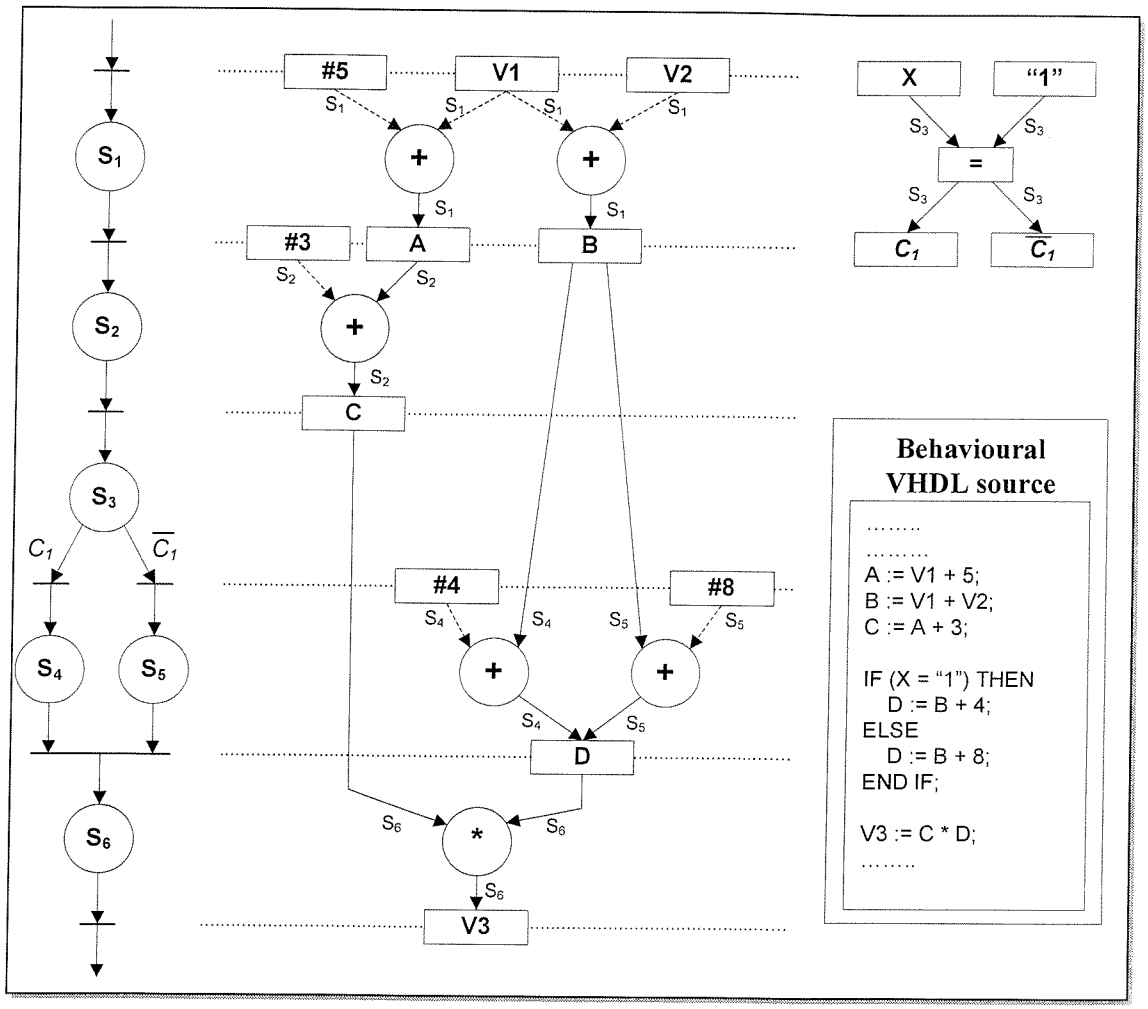


Figure 2-5 Extended timed petri net representation

2.4 Scheduling, allocation and module binding

The next three tasks form the core of the behavioural synthesis system. These tasks are concerned with performing scheduling, allocation and module binding according to user-defined optimisation objectives.

- *Operation scheduling* – is the task of assigning each operation to a particular time step. Schedules are optimised to achieve the objectives of the user, whilst satisfying both resource constraints, specified by a given target area or the maximum number of functional types within each time step, or time constraints, specified by the number of time (or control) steps for the operations.

Scheduling techniques [27] can be generalised into two main categories: *constructive scheduling* and *transformational scheduling*. Simple constructive algorithms include *As Soon As Possible* (ASAP) [28], *As Late As Possible* (ALAP) [28]. ASAP schedules operations in the earliest possible time step permitted by the data dependencies, while ALAP assigns operations to the latest possible time step. The main disadvantage of these two algorithms is that all operations are treated equally, with no priority given to the more critical ones. This can result in operations that are less critical to be scheduled first on a limited resource (e.g. a single multiplier unit), which may block critical operations scheduling and result in a performance degradation. Figure 2-6 illustrates an example of ASAP and ALAP schedules.

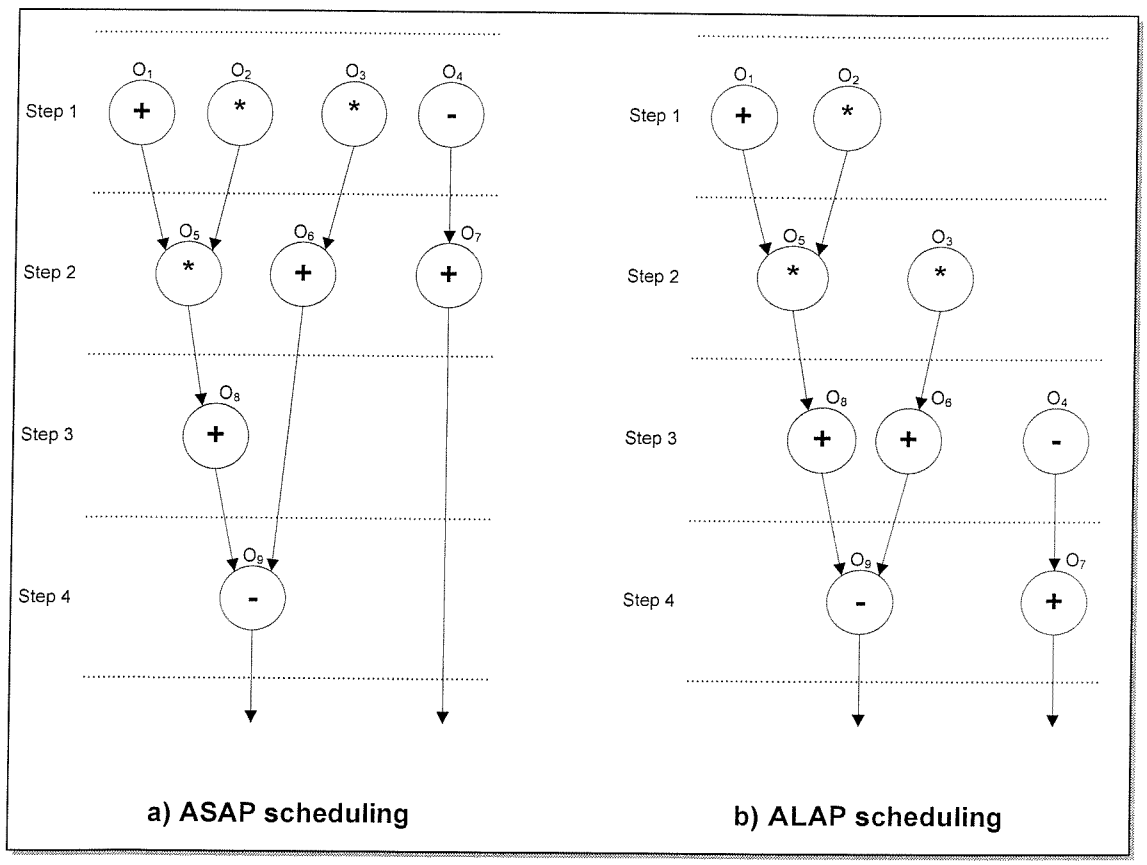


Figure 2-6 Example of ASAP and ALAP schedules

List scheduling [29] takes a more controlled approach in ordering the operations to be scheduled based on some priority function. At each control step, operations (O_n) are scheduled sequentially as long as the required resource is available, otherwise, operations are postponed according to their priority. Figure 2-7 shows list scheduling of a simple control graph, where the priority of each operator is defined as the length of the data path from the operation to the end of the block (marked in braces in Figure 2-7). Operation 3 (O_3) has a higher priority than operation 1 (O_1), and is therefore scheduled in control step 1, providing an optimal solution in this case.

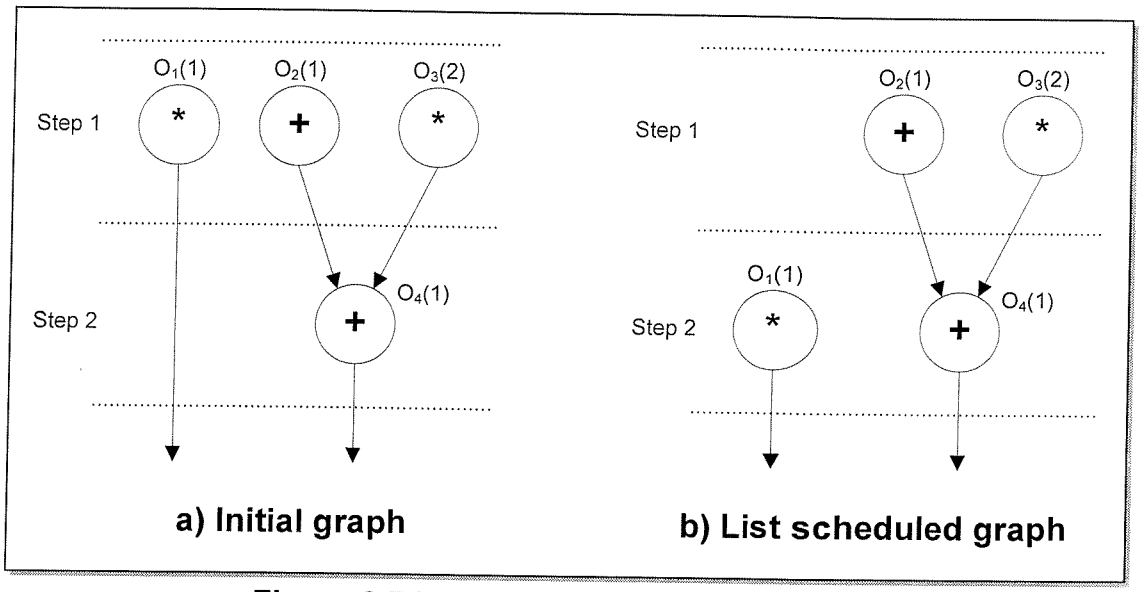


Figure 2-7 Example of list scheduled graph

All the above algorithms make decisions on local considerations, which may be optimal for one operation, but do not necessarily produce an overall optimal schedule. A constructive scheduling algorithm that makes global analysis of the operations and control steps when selecting the next operation to be scheduled is the force-directed scheduling [30]. The basic strategy of this algorithm is to balance the concurrency of operations to ensure that each functional unit has a high utilisation and therefore the number of units required is reduced. Force-directed scheduling is more computationally expensive than all the constructive algorithms mentioned previously, due to its global selection of the next operation to schedule.

In contrast to constructive scheduling which creates a schedule from scratch and adds operations one at a time until all operations are scheduled, transformational scheduling starts with an initial schedule, generally maximally serial or maximally parallel, and iteratively applies a set of local transformations, improving and guiding the design towards the objectives specified by the user. One important advantage of the transformational-based approach is that a complete schedule exists in each iteration and accurate estimation of the design in terms of different criteria (e.g. area or delay) can be made. This technique has been adopted in high-level synthesis systems such as Computer-Aided Modelling, Analysis and Design (CAMAD) [25, 31] and MOODS [32], where both systems combine the scheduling and allocation together as a general optimisation problem. The transformation-based approach employed in MOODS is described in more detail in latter sections.

- *Allocation* – involves the assignment of data variables and instructions into groups of data elements; storage units (registers, ROMs, RAMs, etc) used to hold data in the data path, functional units (adders, ALUs, multipliers, etc) that perform the operations depicted by the instructions and interconnect units (multiplexors) between storage units and functional units.

Allocation techniques can also be generalised into *constructive* and *global* algorithms. Iterative/constructive allocation algorithms select an operation and the data element to which it will be bound, one at a time in an iterative manner and builds up the allocation, typically minimising cost in terms of area whilst conforming to timing constraints of the schedule.

Global allocation techniques, on the other hand, deal with the data path as a whole, and attempt to allocate all its elements simultaneously. Allocation can be defined as a graph problem, where a clique-partitioning algorithm [33] builds a *compatibility graph* where vertices denote operations and edges denote the compatibility relation between the operations whereby edges connect mutually-exclusive operations that can share the same hardware. The problem is then reduced to finding a maximal partitioning of fully interconnected vertices, which represents a solution with the minimum hardware cost. Examples of other global allocation techniques include minimal graph-colouring algorithm, left-edge algorithm [34].

- *Module Binding* – is the process of selecting and assigning the allocated data path units from a list of technology-dependent hardware blocks implemented from units in the target cell/module library. Depending on the requirements of the synthesis system, the module library may contain exactly one implementation per functional unit, or a selection of implementations (such as ripple-carry and carry-lookahead adders) per unit, thus allowing a one-to-many mapping choice, in which case the chosen implementation will be based on user objectives. The low-level module characterisation data, generally in the form of area and delay estimates are used to guide the scheduling and allocation processes. A similar operation is performed on the control path, implementing the circuitry for the control path units, which activate and steer data in the data path via appropriate data path control signals (e.g. register load signal, multiplexer select signals). The module library can be extended, possibly into multiple module libraries where module cells are designed specifically for a particular design such as floating-point functional units or special communication protocol units.

2.5 Design space exploration

High-level synthesis is the process of transforming a behavioural description of a design, in the form of its initial internal representation within the synthesis tool, into a structural implementation, optimised according to objectives set by the designer. The synthesis process produces a range of implementations for a particular input design, and each of these implementations forms a single point in what is called the design space [35-37], which is defined as the *n*-dimensional space describing all possible implementations of a single input description, in terms of *n* design aspects.

Figure 2-8 shows a two-dimensional design space in terms of area and delay (latency). For any particular design and target technology, the design space consists of two regions where feasible implementations lie in the *achievable* region and infeasible implementations lie in the *unachievable* region. These two regions are separated by the *optimal design curve*, which comprises a set of discrete points representing the most efficient implementations. For a given system, only a portion of the achievable region may be obtained as indicated by the shaded *actual achievable region*. This actual achievable

region is dependent on a number of factors such as the optimisation algorithms and design space modelling methods [38] used.

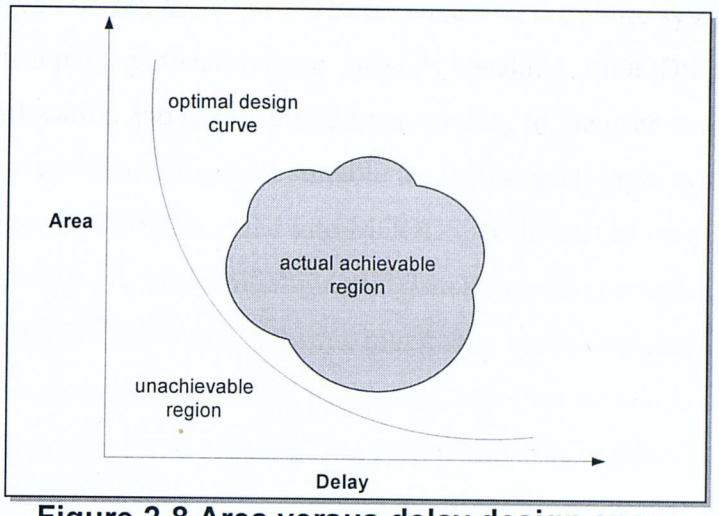


Figure 2-8 Area versus delay design space

2.6 MOODS

The MOODS (Multiple Objective Optimisation in Data and control path Synthesis) synthesis system is the behavioural synthesis system used and modified for the multi-FPGA partitioning research. The term MOODS refers to the entire synthesis system, however the core synthesis engine is also referred to as MOODS.

This section details the principles and operations of the original MOODS synthesis system (without multi-FPGA partitioning enhancements) [32, 35, 36, 39]. The entire synthesis system comprises a number of separate programs performing various tasks in the synthesis of a behavioural description from VHDL down to hardware FPGA implementation. These tasks communicate via a number of generated intermediate files.

Figure 2-9 illustrates the data flow of the original MOODS synthesis system before the multi-FPGA partitioning enhancements. The actions performed by the subcomponents are:

1. ***VHDL and ICODE assembler*:** The behavioural VHDL description is passed into the VHDL complier, 'VHDL2IC' and translated into a simpler intermediate description,

ICODE (Intermediate CODE). ICODE is an input language-independent description suitable for direct input into the core MOODS synthesis engine.

2. ***Synthesis engine – MOODS***: The core component in the entire system is the MOODS data and control path synthesis engine. Broadly speaking, MOODS performs scheduling, allocation and module binding according to the user-defined optimisation objectives, and produces an output suitable for the targeted logic synthesis and layout tools. The single ICODE file is fed into MOODS, with a set of user objectives and technology libraries. A naïve initial internal data structure is created by a direct translation of the ICODE input to form a maximally serial implementation (i.e. one control state/clock cycle per ICODE instruction, with the functionality of each instruction being bound to a separate data path node). The synthesis proceeds and iteratively modifies the data structures until the user objectives are met. The internal representation is converted into a technology-specific netlist using interface information stored in the library. The use of technology-specific estimates fed up from the cell libraries enables MOODS to make technology-dependent trade-offs, while maintaining overall technology (and layout system) independence within the bulk of the synthesis system.

3. ***Structural Linker – DDFLink***: The DDFLink (Design Data Format Link) linker is now used for the generation of the structural VHDL file output, previously generated directly from MOODS. The 'raw' structural VHDL description generated directly from MOODS is only suitable for debugging purposes, as it is rather unreadable, and contains unoptimised control to/from the data path glue logic. DDFLink performs a range of post-synthesis cleanup tasks (including optimisation of the glue logic to remove redundancy), and generates a highly commented and more readable structural VHDL output. The original intention of this back-end link stage was to take the output from several separately-synthesised blocks, and combine them in much the same way as a compiler link stage.

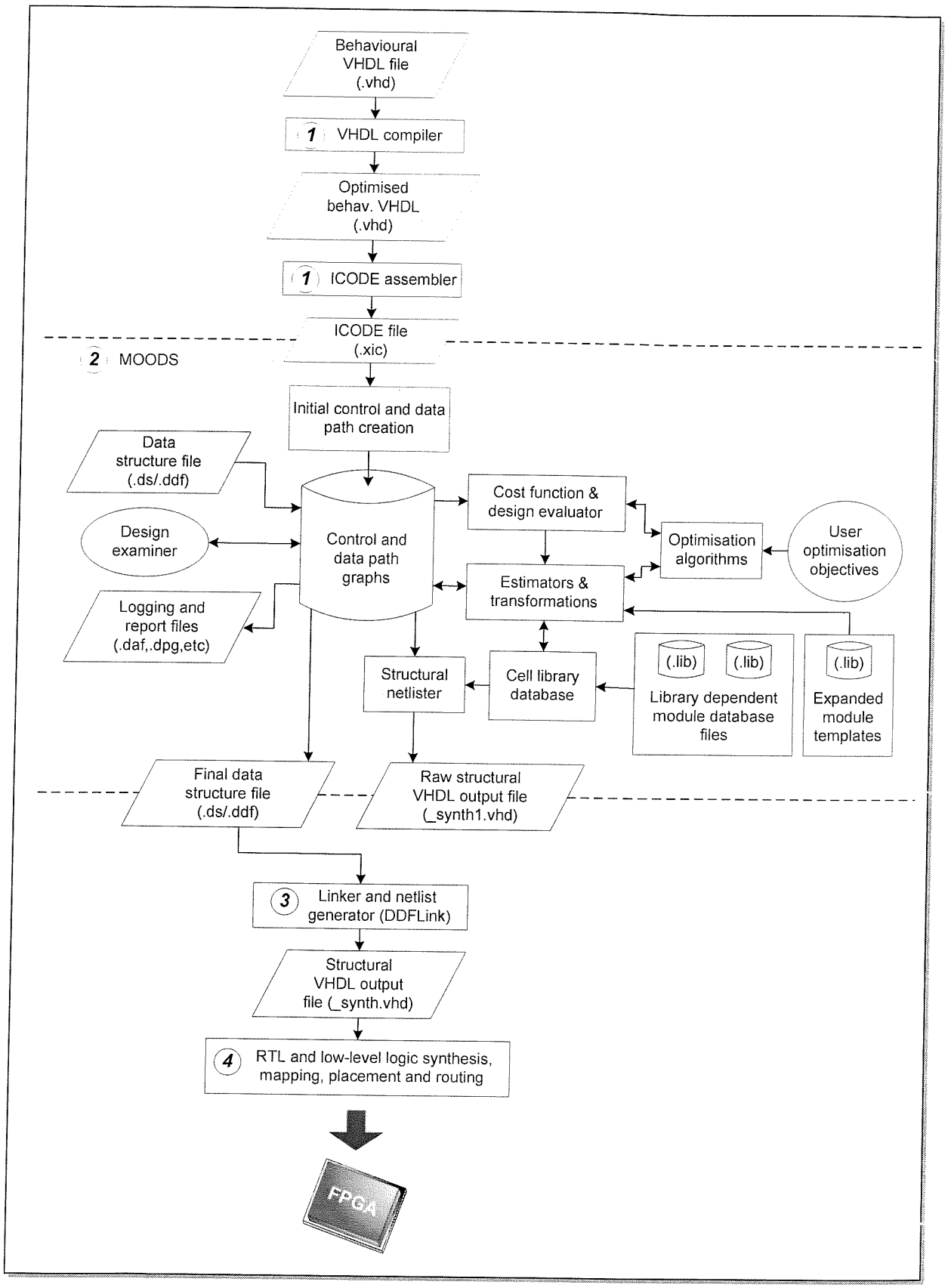


Figure 2-9 Original MOODS synthesis system design flow

4. ***Back-end synthesis, logic-level optimisation and technology mapping, placement and routing***: The final stage in the design flow is low-level optimisation and technology mapping, which utilises a number of third party tools, Synplicity Synplify Pro, Xilinx ISE (Integrated Software Environment), and Mentor Graphics LeonardoSpectrum. These tools take the structural VHDL description generated by MOODS as input. Each tool performs the low-level logic synthesis and technology mapping, which translates the design into a physical circuit to be implemented in an FPGA or ASIC as illustrated in Figure 2-10. For Xilinx FPGAs, the Xilinx-targeted EDIF (Electronic Design Interchange Format) output from RTL synthesis tools is processed by Xilinx ISE to generate a bitstream file to download onto a FPGA.

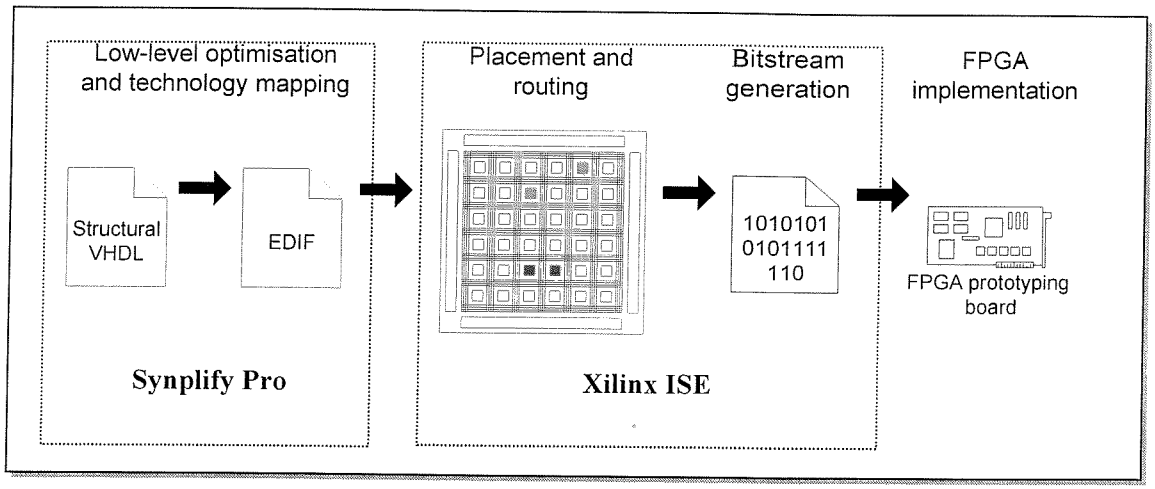


Figure 2-10 Back-end synthesis using third party tools

2.6.1 Synthesisable VHDL subset in MOODS

VHDL is used by the MOODS synthesis system described in the body of the thesis. VHDL, which has already been considered in Section 2.2.3, was initially designed as a simulation language. This leads to a number of problems when implementing VHDL in a synthesis environment. The general set of behavioural VHDL restrictions [24, 40, 41] in the context of synthesis imposes a set of constraints on the synthesisable VHDL [39, 42] in the MOODS synthesis system. The limitations are due to the difficulty of implementation of certain features within the VHDL language, and the relaxed timing

model utilised within behavioural synthesis. Examples of unsynthesisable VHDL are data types such as pointers and linked lists which are unrealisable in the context of hardware logic and gates because their size is dynamic, unsynthesisable constructs such as *assert* statements for simulation-only operations for error messages and anything to do with file I/O types due to lack of an operating system to deal with the file I/O operations such as opening and closing a file.

2.6.2 ICODE generation

The VHDL compiler that forms the front-end of the MOODS synthesis system parses and translates a single or a number of input VHDL files into a single language-independent ICODE file. Using the intermediate language ICODE as input to the MOODS synthesis core allows the use of different languages (Section 2.2) to describe the behaviour of the user design. ICODE describes the functionality, sequencing and connectivity of the design in a lower language level, similar to an assembly language representation of a software language, with additional control flow information. Complex statements (such as sqrt(dx*dx + dy*dy)) must be broken up because they cannot be sensibly represented as an atomic operation. ICODE is in a form suitable for direct mapping to the cell library, and employs simple two input operations to ensure technology independence.

VHDL processes, procedures, and functions are translated and mapped into a set of ICODE *modules*, with the main entity/architecture definitions mapped to the ICODE *program* module that forms the root of the system's control flow. The processes within the architecture definition are merged into the program module. The VHDL process is the only concurrent construct that is converted into ICODE. No other VHDL concurrent constructs are supported. The concurrent behaviour of VHDL processes is performed in the ICODE representation by the execution of the first ICODE *instruction* with a multiple instruction *activation list* on a system reset. Each entry in this first activation list activates the first instruction in each process and its control flow never re-converges.

Each module comprises a set of numbered ICODE instructions together with an associated *activation list*. The sequencing of operations is based on a Petri-net style token passing

mechanism, in which an instruction is only executed once it has been *activated* by another; the activation list specifying which instructions are to be executed once the current one has terminated.

VHDL signals and variables are translated into equivalent ICODE registers, aliases (bit-slices), memory blocks, counters or ports based on their declaration in the behavioural code. An ICODE counter is inferred from variables defined within a loop construct. RAM and ROM memory blocks are specified directly by the user. A port is used only to define the input ports within the I/O list of the module as output ports are defined as registers.

A simple example showing a fragment of behavioural VHDL code with its equivalent ICODE is shown in Figure 2-11. It outlines the key features of the generated ICODE description:

- An ICODE file can contain a number of '*MODULE*'s (ICODE lines 27 and 39), which are translations of VHDL subprograms (functions and procedures). Concurrent processes are merged into the main '*PROGRAM*' module. The first ICODE *NOOP* instruction with a multiple instruction *activation list* activates the first translated ICODE instruction in all the VHDL process on a system reset. For example, instruction *2*, which is the first ICODE instruction in process P_one, and instruction *10* which is the first ICODE instruction in process P_two.

- An ICODE instruction has the general form:

  label : OPERATION <inputs>, <outputs> <activation list>

  where '*.L*' precedes the instruction number. For example, instruction *1* is labelled '*.L0001*', instruction *2* is labelled '*.L0002*', and so forth.

- Each ICODE instruction can be activated by any number of other ICODE instructions. Upon completion of the execution of the current instruction, all the instructions in its activation list are activated. If no activations are listed for an instruction, then next instruction is activated. For example, instruction 2 (labelled '*.L0002*') activates a *conditional branch* instruction 3 (labelled '*.L0003*'), which then activates either

instructions *4* or *9*. While the absence of the activation list in instruction *5* results in an automatic activation of instruction *6*.

- Conditional branches are implemented as an *IF* instruction with two activation lists. One for the true condition (*ACTT*) and the other for the false condition (*ACTF*). The VHDL conditional statement ("*if start = "1" then*" in VHDL line 13) is implemented as two instructions *2* and *3*, with instruction *4* being activated if the condition is true ,and instruction *9* being activated if the condition is false.

- Complex expressions are split into a number of simpler ICODE instructions, with temporary variables (for example, *tmp1* in ICODE line 44) inserted to pass data through each operation.

- VHDL subprograms (functions and procedures) are implemented as separate modules and these modules are activated via a calling ICODE *MODULEAP* instruction, which halts the main execution and passes control to the called module. A subprogram module will return when the *ENDMODULE* instruction is activated.

A complete definition of the ICODE format is provided in Appendix C.1.

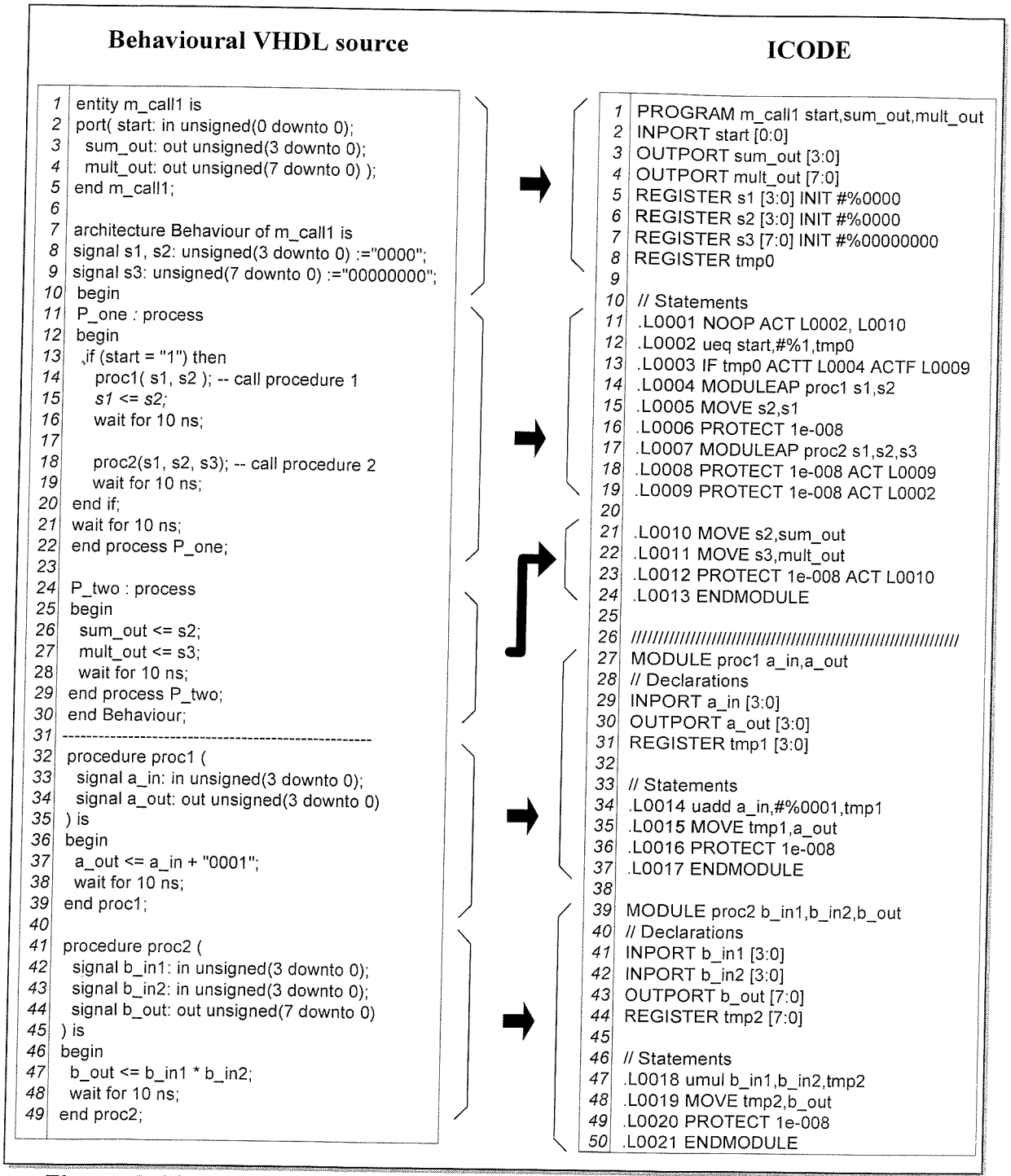


Figure 2-11 VHDL and the generated ICODE for a sum/multiply example

2.6.3 Data path and control path structure

The internal MOODS core data structures hold both the behavioural representation of the ICODE and a fully bounded structural implementation of the structural implementation of the behavioural data path and control path. MOODS models the control and data paths as two separate graphs, linked together via implementation links and control equations. The data path nodes implement the operations performed by the ICODE instruction and the storage elements in the data path stores the ICODE variables passed into the operations and the results from each execution. The control path holds a graph representation of every state within the controlling state machine. The MOODS synthesis process is the iterative process of applying multiple simple optimisation transformations to these data structures, controlled by a transformation selection algorithm. It is possible to output a structural representation of the system at any point within the synthesis process after the building of the control and data path graphs from the ICODE file.

Figure 2-12 shows the initial control and data path graphs created by MOODS for the example shown in Figure 2-11. The structural implementation is generated directly from the ICODE, and this naïve implementation of the behaviour has one control state node per instruction and a separate data path node for each functional ICODE operation and ICODE variable. At this stage, the initial structure has no shared operation and variable storage elements within the data path and since each data path node is activated by one control state node, it is possible to superimpose the schedules time steps over the data path graph, as illustrated in Figure 2-12. This combined view of the design is no longer feasible during synthesis when the functional units and storage elements are shared within the data path, altering the one-to-one direct correspondence between the two graphs.

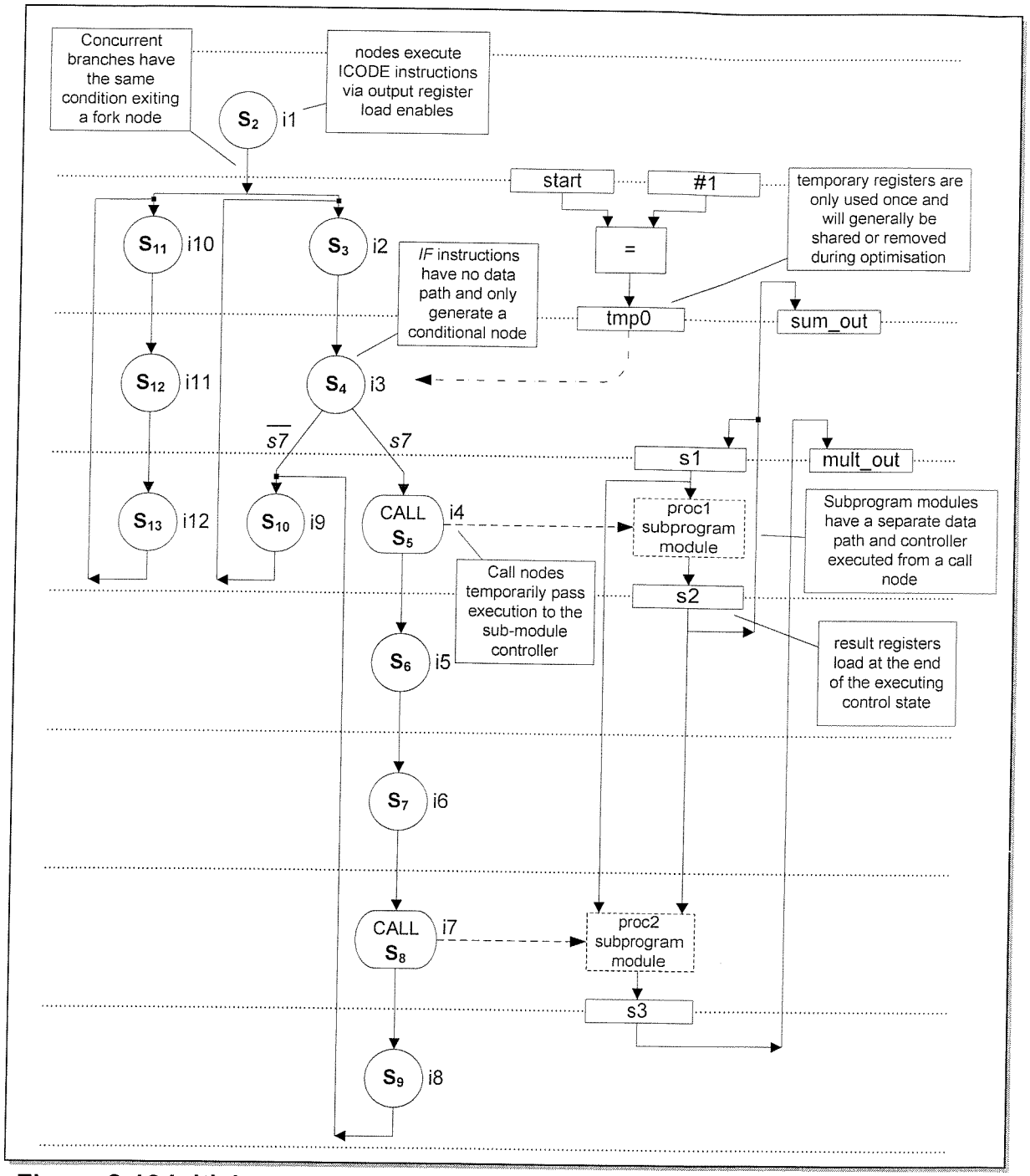


Figure 2-12 Initial control and data flow graphs for the sum/multiply example

2.6.3.1 The control path

The control path data structure is built within MOODS in a graph structure, where each graph node represents a single control state. Input and output control arcs between the graph nodes form the links to the previous and next control state node. These arcs describe

state transitions conditional on signals generated by the data path. For example, in Figure 2-12, the transition from state S_4 to state S_5 or S_{10} is conditional on the data path signal $s7$.

At present, a one-hot encoded token passing structure is implemented for the controller. The controller itself is a non-deterministic finite state machine, where each state conceptually executes one or more ICODE instructions. Each control state is built from a control cell that contains a single register that is activated for one clock cycle by one or more token inputs to the cell. These token signals are representative of the arcs connecting the control state nodes, and the registered state bit forms the state enable signals used to control the data path. The controller structure suits the register-rich FPGA architecture. It is entirely possible to implement the controller using alternative state encoding (e.g. binary, gray-coded) in platforms with limited registers, or use a micro-coded controller.

The instructions executed during a state are stored as an instruction list within the control state node data structure. A set of acyclic sub-graphs within this list divides the instructions into groups of dependent instructions, where each group is numbered and instructions within the group are executed sequentially. Instructions in each group are data independent with instructions in other groups, and hence the instructions can be executed concurrently. Within a group, the instructions are dependent on each other and they are executed sequentially. Figure 2-13 illustrates the execution of two concurrent instruction groups in a single control state. The two add instructions are data dependent and the result of chaining the two add instructions (*i1* and *i2*) in a single control state is that two separate adder data path units are required. The multiply instruction (*i3*) grouped separately from the addition instructions executes concurrently. The propagation delay for the data path units are used to estimate the minimum delay required executing the instructions in the control state. The *characterisation data* (i.e. inherent data) for each instruction is fed from links to the data path nodes implementing the relevant ICODE instructions. All data path nodes are fully bound to a physical technology-specific library cell during synthesis, from which the characterisation data is obtained and fed up to the synthesis optimisation process. The estimated delay is used to determine the maximum allowable clock rate for a design. With a synthesis constraint (e.g. setting the clock period to 20 ns) specified by the user, the propagation delay for the data path units determines if instructions implemented by the corresponding units can be chained to execute in one control state.

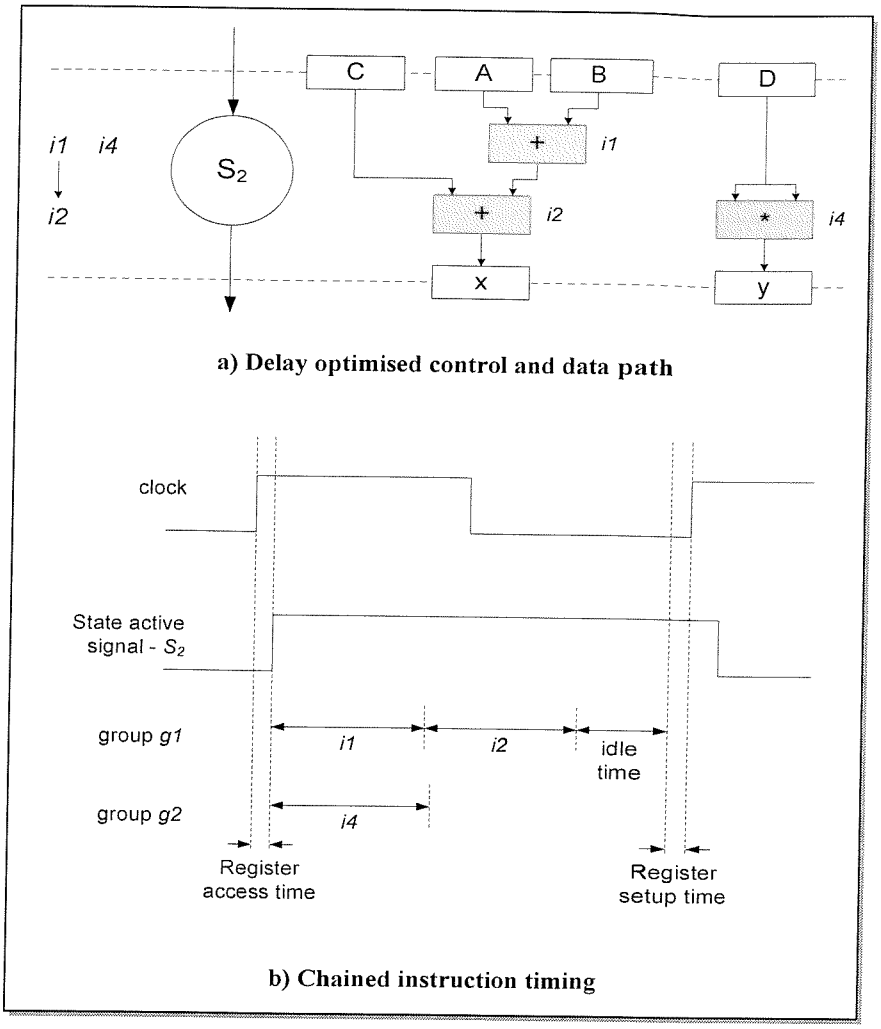


Figure 2-13 Execution of chain instruction in a single control state

The control nodes in the control path are categorised into six basic types as listed in Table 2-1. Scheduling transformations performed on the control graph data structure tends to merge control nodes together forming composites of the types listed below. This merging of control nodes does not apply to collect and call nodes. The collect node, however, can be completely removed by the parallel merge transformation.

Control node types	Description
***General* node**	This node has a single unconditional input and output arc and it can contain any ICODE instructions other than '*MODULEAP*', '*COLLECT*' or conditional instructions. A *general* node represents a simple sequential control state taking one cycle.

Control node types	Description
***Fork* node**	This node has a single input arc and multiple unconditional output arcs. It can contain the same ICODE instructions as the *general* node. *Fork* node forms the root of a concurrent branch that simultaneously activates all the successor nodes.
***Collect* node**	A *collect* node has two or more input arcs and a single output arc. The node contains a single ICODE '*COLLECT*' instruction only and the node will not activate the next control state node until a fixed number of input arcs (which may be less than the total number of inputs) are activated, thereby synchronising a set of concurrent branches. This node complements the *fork* node. Note that the synchronisation of the translated concurrent VHDL *processes* is not done with the *collect* node. The VHDL compiler no longer supports concurrent translation of sequential instruction activations, thus rendering the collect node obsolete. This 'collect' mechanism is still supported by the MOODS core, and thus the *collect* node is listed here for completeness.
***Conditional* node**	A *conditional* node has single input arc and two of more conditional output arcs. This node can contain any ICODE instructions supported by the *general* node, as well as conditional ICODE instructions such as '*IF*' and '*SWITCHON*' to form the conditional paths through the control graph.
***Dot* node**	This node has two or more input arcs and a single output arc. This node complements the *conditional* node. Any of the input arcs can activate the node. The *dot* node provides the re-convergence path for the conditional branch sections. It supports the same set of ICODE instructions as the general node.
***Call* node**	The *call* node forms the basis of the module calling mechanism. It has a single input and output arc. This node contains a single ICODE '*MODULEAP*' instruction only. The *call* node delays the execution of the next control state node until a single iteration of the called sub-module is executed. The *call* node activates the first control node in the separate control sub-graph of the called module and when the sub-module terminates, control is passed back to the *call* node and it activates the next control state node in the main graph.

Table 2-1 Descriptions of the six basic control node types

Once the control graph is optimised, the only distinct types of node are the call, fork, conditional and general node types. A call node is physically realised by the control *call* node described in Section 4.6, with all other nodes realised by a *general* control node.

2.6.3.2 The data path

The MOODS synthesis core generates the data path as a fully connected graph of data path node units, connected indirectly via data path nets. This level of indirection is used to determine the bit range connectivity of multi-bit nets used to connect the node units. The network of functional (adders, multipliers etc.), storage (registers), and interconnect (multiplexors) units in the data path graph implements the functionality of the ICODE instructions. The flow of data through this network is controlled by the control nodes in the control path. The data path consists of three main types of data path node units:

1. One storage unit (register) is initially created for each ICODE variable (both user specified and temporary variables). A number of different types of storage units exist and the selection of which of these optimised storage units depends on the operations performed on the variable. The general register type storage unit is implemented for the storage of data variables and temporary variables used in a number of instructions. Each operating instruction is performed by a separate or shared functional unit, which is connected to a register input via a multiplexor. A variable which is only reset and incremented (or decremented) is implemented by a counter register with a reset input, thereby removing the need for an adder unit for such instruction executions. A third type of storage unit is formed from a multi-bit array variable or constant, where a RAM-type or ROM-type storage unit is created respectively. During synthesis, register sharing and bypassing reduces the number of physical storage units required.

2. A functional unit implements ICODE operations such as additions, multiplications, and comparisons. These operations are purely combinational, with a combinatorial functional unit to produce the results. Characterisation data (Section 2.6.3.1) from cell libraries provides technology-specific performance data and this is used to estimate the area and maximum delay; time required for an input change to

propagate through to its output (i.e. the longest accumulated combinatorial delay). The functional nodes are not controlled directly from the control state nodes in the control path; they rely on the controller to feed the appropriate values to the inputs of the functional units and to read the results of the unit during specified control states. An exception to this rule is the use of ALU type functional units, which can perform more than one type of operation. The type of operation is selected from a set of input control signals, driven by control nodes in the control state machine. An example of an ALU unit is an add/subtract, which is used in place of a single add unit and a single subtract unit. Note that only one type of operation may be used within any single activated control state node.

3. The final type of data path unit is the interconnect unit. The library cell that implements the interconnect node is a multiplexor. The multiplexor selects the appropriate input amongst two of more input nets and drives the inputs of any shared data path unit. Interconnect units are not physically created until the post-processing phase in the synthesis process for code size efficiency and reducing synthesis runtime during data path optimisation. The MOODS synthesis tool, however, does take into account the delay and area factors of these implied multiplexors during the optimisation process.

Each data path node is a generic functional block performing the appropriate ICODE operations (or operations for ALUs). Functionality and characterisation data (such as area and delay) for each unit is obtained via a link into the cell library. The actual physical implementations of the data path elements are taken from a pre-defined set of parameterised structural/RTL components defined in a technology library file. The separation of the generic and physical aspects of the data path elements gives technology independence within the synthesis core without sacrificing the accuracy of the performance information as the technology specific cell information is used within the synthesis process.

The signals that link the control path nodes to the data path nodes are represented by Boolean logic equations. This abstraction of the control signal generation allows a number of different ways of implementing the linking signals, which includes a network of multi-level logic gates, ROM lookup, or combined with the control graph to form a micro-coded

controller. At present, these linkage signals are output as simple VHDL logic expressions, leaving Boolean optimisation and technology mapping to the low-level logic synthesis tools.

2.6.4 Transformations

MOODS formulates the synthesis optimisation process as an iterative optimisation of the initial naïve implementation of the design, where the synthesis task is divided into the traditional sub-tasks of scheduling, allocation, and module binding. This allows trade-offs to be made between the various synthesis sub-tasks, which are performed simultaneously. Optimisation is performed by applying a number of small local transformations on selected parts of the design using a dedicated optimisation algorithm.

Each local transformation is semantic preserving and *complete*, resulting in a valid design after each transformation applied to the design. Throughout the process, the internal representation describes a complete and fully bound design implementation, making use of the low-level characterisation information from the module libraries to provide accurate estimates for circuit performance. At present, MOODS has a set of fourteen different transformations, each performing slight changes to the design, adjusting the scheduling of the control state nodes in the control path, and the allocation and binding of data path nodes in the data path. The fourteen transformations include six inverse transformations to perform backwards steps to reverse previous design decisions, resolving the problem associated with premature binding decisions, which may produce non-optimal designs. Details of the four basic control state merging transformations, two inverse state-splitting transformations to undo the merging of states, and a clock period adjustment transformation, are given in Section 2.6.4.1. Details of the two data path unit sharing transformations, together with four of their associated inverse unsharing transformations, and a binding transformation to select an alternate functional unit are given in Section 2.6.4.2.

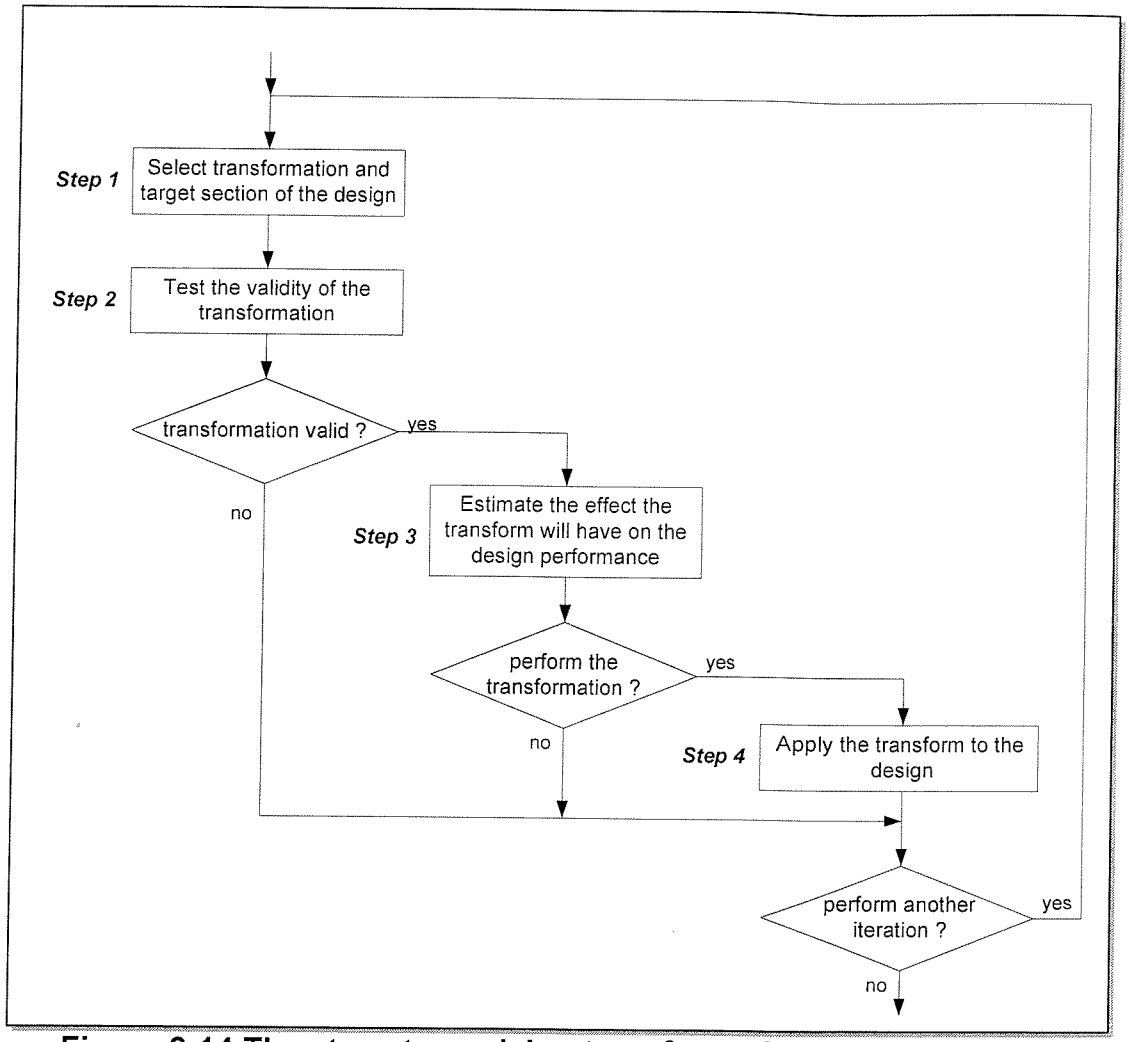


Figure 2-14 The steps to applying transformations in the iterative optimisation process

The selection and application of the transformations performed within each iteration of the optimisation process consists of four separate steps, as illustrated in Figure 2-14:

1. *Selection* - This initial phase selects a transformation from the fourteen available, and the portion of the design to which it should be applied. This selection is controlled by the optimisation algorithm in use.

2. *Testing* - The second step involves checking the validity of the given transformation on the selected portion of the design. It is possible for some transformations to alter the behaviour of the design (e.g. instruction dependency and mutual exclusivity). This

testing phase ensures that all transformations to be applied are valid and invalid ones are filtered out and aborted.

3. *Estimation* - This estimation step evaluates the effect of the given transformation on the system performance based on the user objectives (such as area, delay, power consumption, etc), without permanently altering the core data structures. This step *simulates* all the changes made to the design by the transformation and calculates the effect of these on the current system performance. The optimisation algorithm uses the results of the estimation to determine whether to perform the transformation (i.e. make the changes in the core data structures) or abort the transformation.

4. *Execution* - The last step applies the transformation, altering the internal data structures of the design.

2.6.4.1 Scheduling

Scheduling transformations perform control graph optimisation, whereby ICODE instructions are assigned to control state nodes and the number of control state nodes used to perform a number of ICODE operations is optimised, when more than one ICODE instructions are merged into a single control state. There are four basic state merging transformations, two inverse state-splitting transformations to undo the merging of states, and clock period adjustment transformation. These seven scheduling transformations and their effects are listed in Table 2-2. More details may be found in [39].

Transformation	Effects
Sequential merge	This merging transform merges two sequential control nodes (i.e. nodes executed sequentially) to form a single control node implementing multiple instructions, where the instructions in the second node are moved into the first. ICODE instructions with data dependency are chained together within an instruction group, thus bypassing intermediate data value registers. The second control node with no associated instructions is then removed from the control path.

Transformation	Effects
Parallel merge	This transform is applied to a concurrent branching fork node, where the first nodes within each branch are merged into a single successor node.
Merge fork and successor	Elements of the previous two are combined to form this third transformation, where the successor instructions contained within one branch are merged into the branching (fork or conditional) node.
Group instructions on register	This transformation is geared towards removing registers with a single input and output net, which is common in the temporary variable register storage units generated by the compiler. These variables are accessed by one read and write instruction. The transformation tries to bypass (remove) the data path register by merging the instruction group that contains the write instruction into the read instruction control node.
Ungroup into groups	This inverse scheduling transformation is a state-splitting transform, which moves groups of instructions within a control state node into two separate control nodes. The first control state node contains the single selected and extracted group of instructions, leaving the remaining groups in the second control node.
Ungroup into time slices	This second inverse scheduling transformation extracts instructions from a specified control node and places them into a number of new control state node, such that the time taken to execute the instruction groups in any of these old or new control nodes does not exceed a specified period.
Clock set / multi-cycling	This optimisation is a global optimisation that sets the maximum clock period for the entire design. This transformation makes use of the *ungroup into time slices* transformation, forcing all control nodes below a user specified clock period constraint.

Table 2-2 Scheduling transformations

2.6.4.2 Allocation and Binding

The second group of transformations acts upon the data path, performing allocation and binding optimisation, where the transformations are concerned with the sharing and unsharing of data path units. There are two sharing transformations, together with four of their associated inverse unsharing transformations. A further binding transformation is also provided to select an alternate functional unit to perform the same operation. These seven data path transformations and their effects are listed in Table 2-3. More details may be found in [39].

Transformation	Effects
Combine functional units	This transformation attempts to merge two functional units into one, where the operations performed by the two source functional units are not executed in the same time slice. This has the effect of time-sharing a functional unit between multiple operations. The resultant function unit will have number of inputs, which are selected by the multiplexor interconnect node. Control signals from control state nodes are used to drive the select signals of the inferred multiplexors, and the load signals of the required output registers. The availability of multi-function ALU units in the cell libraries allows two combined units performing different operations to merge into a single ALU unit. For example, merging an add and a subtract functional unit into a single add/subtract ALU unit.
Share registers	This second merging transformation tries to share a single register storage unit between multiple ICODE variables with non-overlapping lifetimes, or variables that occurs in mutually exclusive conditional branches. The register lifetime analysis also takes into account variable persistence around loops and through conditional branches.

Transformation	Effects
Uncombine instruction from unit	This first uncombine transformation undoes the merging of two functional units by the combine functional unit transformation. It takes a shared functional unit that implements two or more ICODE instructions, and removes one of these instructions into a new functional unit created to implement the extracted instruction. The transform makes use of the cell library to determine the type of unit to use for implementing the extracted instruction. The unit that was initially shared is re-evaluated and the cell library is used to select a replacement functional unit to perform the remaining instructions (i.e. minus the extracted instruction).
Uncombine unit fully	This transformation utilises the *uncombine instruction from unit* transformation to completely extract all ICODE instructions from a single functional unit into a number of functional units, one unit implementing one instruction from the original shared unit.
Unshare variable from register	Shared registers are unshared in a manner similar to the *uncombine instruction from unit* transformation using this transformation, which extracts one of the implemented variables from the shared variable and placed in a new separate register storage unit.
Unshare register fully	This transformation utilises the *unshared variable from register* transformation mentioned above to completely unshare all the ICODE variables being implemented by a single shared registered storage unit. This transformation creates a number of separate register storage units, one for each ICODE variable.
Alternate implementation	This is the only binding transformation provided by the MOODS synthesis core. For a functional unit that has two or more different available implementations in the cell library, this transformation attempts to replace the existing unit bound to the functional data path unit with an alternative implementation with a different area and delay characteristics. This attempt to use an alternative implementation changes the cost of the unit and the cost function used within the optimisation algorithm is used to determine whether to accept or abort the new unit binding.

Table 2-3 Allocation and binding transformations

2.6.5 Cost function

The *cost function* is a measure of the "goodness" of a change (e.g. merging of control states or sharing of data path units) in the design structure with an application of the selected transformations. This cost function is used during the estimation phase (step 3 in Figure 2-14) in the iterative optimisation process. The cost function evaluates a design configuration with respect to the target objectives specified by the user, where the multiple, possibly conflicting objectives form the weighted costs of each objective (dimension). These weighted costs are used by the cost function to produce a single value representation of the state of the design in an n-dimensional design space.

The MOODS cost function allows the user to specify objectives for a number of measurable design parameters such as area, delay, and power consumption. Each of these objectives is specified as a target value and user defined priorities are assigned to each objective. The priority level assigned to objectives determines the order in which targets are optimised, where the *primary objectives* with priority 1 (highest priority) are considered first before any other lower priority objectives.

An analogy for the cost function is the "energy" of a system. During optimisation, the effect of applying the selected transformation is predicted by evaluating its effect on the system "energy". For a single objective, the change is given by:

$$\Delta E = \frac{C_{estimate} - C_{previous}}{C_{initial}} \tag{2.1}$$

Where $C_{estimate}$ is the estimated cost of the design after applying the transformation, $C_{previous}$ is the cost of the design before the transformation, and $C_{initial}$ is the cost of the initial implementation. A negative average change ($\Delta E < 0$) indicates a general improvement in the design implementation with respect to the user objective.

The optimisation algorithm uses the value of the change in energy (ΔE) due to applying a single transform to decide whether or not to accept the transform. ΔE is calculated by summing the change in cost of each objective caused by applying a transform starting with the primary objective. If all primary objectives are met, whereby all $C_{previous} \leq C_{target}$ and

$C_{estimate} \leq C_{target}$ (where C_{target} is the target cost for the objective), then ΔE is calculated from the priority 2 objectives, and so on for other lower priorities.

2.6.6 Optimisation algorithms

The MOODS synthesis core currently provides two optimisation algorithms. Both methods can be used to control the optimisation process described in the previous section. The first key function of the optimisation algorithms is selection of the initial transformation and design portion to which the selected transformation should be applied in the data selection phase. The other function of the optimisation algorithm is to decide the number of transformation iterations to execute.

2.6.6.1 Simulated annealing

This first algorithm exploits a method with an analogy in metallurgy, where *annealing* is originally a process where the molten material is cooled down from the high-energy liquid phase to the minimal low energy solid phase in a controlled, usually slow, manner. A proper disciplined cooling schedule sets the final energy state at its globally minimum level at the end of the cooling process.

The simulated annealing algorithm [43] is a global optimisation method that is based on the Metropolis algorithm [44]. The simulated algorithm works by selecting a random transformation and design section to target the transformation. The resulting system energy change in the cost function is evaluated and the algorithm accepts both improving ($\Delta E < 0$) and degrading ($\Delta E > 0$) transformations. Transformations leading to cost improvements will automatically be accepted, whereas the probability of accepting a cost degrading transformation is given by:

$$P = \exp\frac{-\Delta E}{T} \quad : \quad \Delta E > 0 \tag{2.2}$$

where P is the resulting probability of accepting a degrading transformation, ΔE is the estimated positive change in energy given by the transformation and T is the temperature within the annealing algorithm. This ensures that the probability of acceptance of

degradation decreases when the temperature decreases. The decision to accept degradations is made from the comparison of the probability threshold value and a normalised random number, and acceptance being granted when the generated random value is the smaller value between the two.

Figure 2-15 illustrates a one-dimensional configuration space and it demonstrates how the simulated annealing algorithm avoids being trapped in local minima on the configuration path.

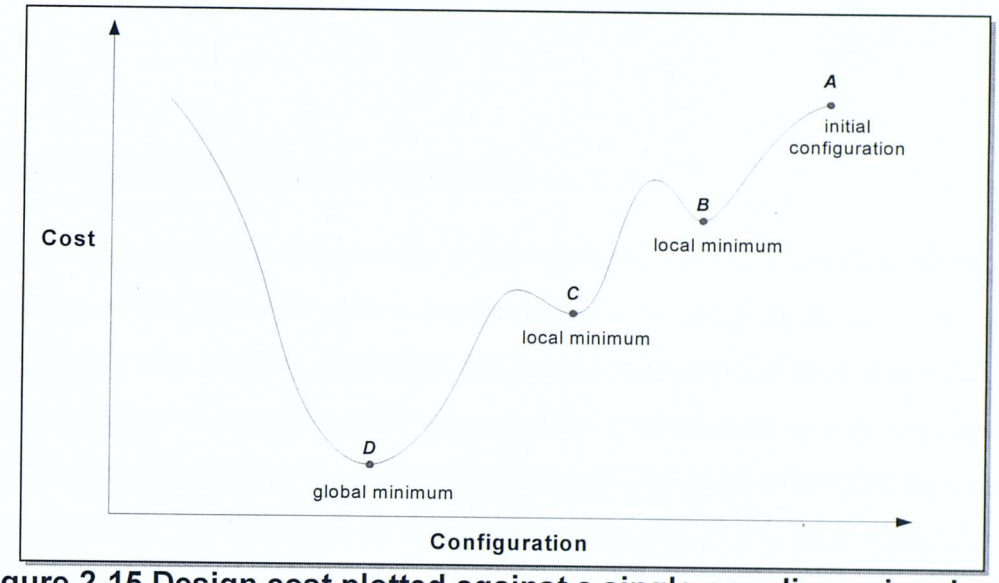


Figure 2-15 Design cost plotted against a single one-dimensional space

The design is initially represented by point *A*. An optimisation algorithm that accepts only transformations that result in an improvement will hit the local minimum (point *B*). The simulated annealing algorithm accepts degradation and hence allows the configuration to jump out of the local minima (points *B* and *C*) into the global minimum (point *D*).

The main advantages of the simulated annealing algorithm are its abstractness in terms of its application independence and its ability to find a global minimum. The optimisation process using the simulated annealing approach relies entirely on the cost function and transformation estimators to encapsulate the design space. This allows complex trade-offs to be made between multiple conflicting objectives, and permits the inclusion of further objectives (e.g. dynamic re-configurability or testability) with additional costing

mechanisms to the cost function, with characteristic information and models for the added objectives. However, there are a few disadvantages associated with the simulated annealing. Firstly, the simulated annealing uses a random approach in the selection of transformations, thus many iterations are required for a system to reach equilibrium, making the simulated annealing approach slower than heuristic methods. The abstract parameter values used to control the simulated annealing process requires manual selection by the user, these are often difficult to predict in advance, requiring considerable experience to obtain the solutions for each design. Generally, the optimisation speed is traded off against the quality of the resultant synthesised design.

2.6.6.2 Quasi-exhaustive heuristic

MOODS synthesis tool addresses some of the unpredictable and often slow nature of the simulated annealing algorithm with a quasi-exhaustive heuristic algorithm, which is both faster and more user friendly. Unlike the random selection method used in simulated annealing, this heuristic approach uses the same set of transformations, applied in a pre-defined schedule, guided by an analysis of the design. The quasi-exhaustive heuristic produces the same final structure for every optimisation run of any fixed design. The algorithm only performs area and delay optimisation, with knowledge of suitable trade-offs gained through an analysis of a number of test designs.

The quasi-exhaustive heuristic only accepts improving transformations; it uses the same sets of transformations as those used within the simulated annealing algorithm, apart from the inverse (i.e. degrading) transformations. Two basic routines are provided to optimise area and delay:

1. *Compact control path* - This routine utilises the scheduling transformations, merging control state nodes in the control path. This reduces delay by performing more instructions within a single control node and to a lesser extent, as temporary intermediate registers are bypassed and removed in the data path.

2. *Compact data path* - This second routine utilises the allocation and binding transformations, merging operations and sharing register storage units, hence optimising area.

There are two main disadvantages in using the quasi-exhaustive heuristic algorithm. Firstly, the inability to apply degrading transformations can lead to a sub-optimal synthesised design. The other reason is the heuristic approach only performs area/delay optimisations. To perform multi-dimensional trade-offs between multiple conflicting objectives, the heuristic approach needs to understand the interactions between all aspects of the design space in order to perform the most appropriate transformations. The algorithm which is faster than the simulated annealing algorithm provides the user with some general idea of what constitutes a realisable target and it may be used as a pre- or post-processing step in conjunction with simulated annealing for further optimisation.

2.7 Post-processing

The post-processing stage in the MOODS is used to complete the structural description of a design. This epilogue (finalisation) stage happens just before the generation of the raw structural VHDL output (*_synth1.vhd*) file, final data structure (*.ds*) file, and the DDF data structure (*.ddf*) file (the last processing stage in the synthesis engine – MOODS in Figure 2-9).

The first step of the post-processing stage is the bypassing of subprogram module output registers to implement pass by reference (See Chapter 4 for subprogram module modifications to change outputs of external modules to pass by value).

The next stage in the post-processing stage is to generate any multiplexors that are required. These interconnect data path nodes are completely implied during optimisation for efficiency reasons. A multiplexor is created and linked into the data path structure when multiple input nets drive a single data path node in. The net activation conditions which correspond to ICODE instructions that drive the input nets are later converted into multiplexor select signals.

The third stage in the post-processing stage involves the generation of control signals from the conditional signal list. Boolean expressions are generated and linked in to the appropriate nets and control signals within the control and data paths.

Other tasks performed within this post-processing stage include the tidy up of the data path, removing of unused data path units, which have been bypassed during the optimisation process. The control path is also tidied by removing redundant control states.

2.8 Summary

The beginning of this chapter gives an outline of the overall high-level synthesis process and the main sub-tasks within the process. The rest of the chapter describes the MOODS synthesis system in more depth, in particular the methods employed by MOODS to carry out the synthesis sub-tasks. MOODS develops and provides the user with multiple implementations from a single behavioural input description within the design space, which provides trade-off between several, possibly conflicting, objectives in the aim of producing an optimised implementation of the design.

Chapter 3

Multi-FPGA partitioning

3.1 Background

The previous chapter covered an overview of high-level synthesis and an in-depth description of the MOODS synthesis system, which forms one part of the multi-FPGA synthesis system. The other important part is the partitioning phase, more specifically *how* is the design partitioned and *when* to perform partitioning. The first aspect of *how* the design is partitioned deals with the design representation used for partitioning and the partitioning algorithm used. A common design representation for partitioning is based on graph notation, where a data flow graph, control and data-flow graph, or module call graph is partitioned with the goal to attain a partitioned design that fulfils optimisation criteria and constraints such as area, speed, power consumption, number of I/Os, etc. Partitioning can be performed at different abstraction levels and granularity, the second aspect of *when* to perform partitioning has a high impact on the quality of the structural output produced by the synthesis system.

The rest of this chapter provide the background information on multi-FPGA partitioning, giving an insight on *how* a design can be partitioned. Section 3.2 deals with partitioning methodologies and it provides an overview of partitioning algorithms. With an understanding of partitioning algorithms, Section 3.3 introduces multi-FPGA partitioning and describes synthesis systems with multi-FPGA partitioning features. Section 3.4 deals with some aspects of the data communication in the context of multi-FPGA systems. Section 3.5 describes techniques and issues related to data synchronisation across clock domains. An introduction of design activity profiling and how the obtained profile can be

used to guide the partitioning algorithm to achieve an improvement in the implementation solution is given in Section 3.6.

3.2 Partitioning methodology

Multi-FPGA systems [45] are often used in logic emulation, prototyping applications, and implementation of application specific integrated circuits (ASIC) of large system designs because of their programmability features, low costs and short production times [46]. The general partitioning problem is a well-known NP-complete problem [47]. Partitioning of a design over multiple devices can be performed at various levels of abstraction, with a multitude of techniques in partitioning multi-FPGA systems and the possible combinations could reach into the thousands.

3.2.1 Overview of partitioning algorithms

Partitioning algorithms can be classified broadly into two main categories [48, 49]. The first is *constructive* and the other *iterative*. Constructive algorithms determine a partition from the graph describing the circuit or system, whereas iterative methods aim at improving the quality of an existing partitioning solution.

Partitioning algorithms can also be labelled *deterministic* or *probabilistic*. Deterministic algorithms generate the same solution for the same set of inputs every time. Probabilistic algorithms, on the other hand, produce differing solutions as they are based on random numbers. One of the best-known, most widely referred and extended deterministic algorithm is the Kernighan and Lin (KL algorithm) [50] and its variant, the Fiduccia-Mattheyses (FM) heuristic [51]. Refinement and extensions to the basic FM heuristics given by Krishnamurthy [52], Huang and Kahng [53], Hauck and Borriello [54],Cong et al. [55, 56], Dutt and Deng [57], Kužnar et al. [58, 59] as well as many others. Probabilistic or stochastic algorithms includes the Simulated Annealing (SA) algorithm [43] and Genetic Algorithms (GA) [60].

Research on partitioning at a higher level of abstraction (e.g. behavioural partitioning) and hierarchical partitioning techniques were carried out by Vahid et al. [61-65], Digital

Design Environments Laboratory in the University of Cincinnati [66-68], Fang and Wu [69-71], Duncan et al. [72], Krupnova et al. [73] as well as many others.

Kernighan-Lin algorithm

The KL (Kernighan-Lin) [50] algorithm is an iterative improvement bipartitioning algorithm for a graph $G = (V, E)$, which starts with two initial partitions (usually randomly generated) of n elements each. Pairs of vertices are swapped between partitions until no further improvement can be achieved. The KL algorithm attempts to swap pair of vertices to reduce the cutsize or a move resulting in the smallest increase in cutsize, if no decrease is possible. A cost matrix $C = (c_{ij})$, where $i=j= 1, 2, 3, \ldots, 2n$, $i \neq j$ is associated with the graph. For each node $a \in A$, an *external* cost, E_a is defined by:

$$E_a = \sum_{y \in B} c_{ay} \qquad \text{where } c_{ay} \text{ is number of edges that cross the partition boundary} \tag{3.1}$$

and an *internal* cost I_a by:

$$I_a = \sum_{v \in A} c_{av} \qquad \text{where } c_{av} \text{ is number of edges that do not cross the partition boundary} \tag{3.2}$$

$D_a = E_a - I_a$ is the benefit of moving vertex a from A to B. The gain of swapping a vertex pair (a, b), where $a \in A$ and $b \in B$ is given by $D_a + D_b - 2c_{ab}$.

The first step of the KL algorithm arbitrarily partitions V into two equal subsets A and B. External costs, internal costs, and the difference between the two costs are then computed for all vertices. Step 3 of the algorithm is to choose the pair of vertices that will result in the highest gain value when the interchange occurs. The gain resulting from this move is stored and the selected pair of vertices is locked to prevent it from being considered for swapping again. The procedure continues until all n pairs of vertices are evaluated and locked, and the sequence of gains, $g_1, \ldots, g_n$ is generated (Step 4). The total gain of swapping the first vertex pairs is given by:

$$G_k = \sum_{i=1}^{k} g_i \qquad \text{where } 1 \leq k \leq n \tag{3.3}$$

Kernighan-Lin Algorithm

begin

Step1. V= set of $2n$ elements; A, B is the initial partition where $|A| = |B|$; $A \cap B = \varnothing$; and $A \cup B = V$;

Step2. Compute D_v for all $v \in V$; *queue* $\leftarrow 0$; and $i \leftarrow 1$;
$A' = A$; $B' = B$;

Step3. Choose $ai \in A'$, $bi \in B'$, which maximises

$$gi = D_{a_i} + D_{b_i} - 2c_{a_i b_i};$$

Lock a_i and b_i, and add the pair (a_i, b_i) to *queue*;
$A' = A' - \{ a_i \}$; $B' = B' - \{ b_i \}$;

Step4. **if** A' and B' are both empty **then Goto** Step5
else recalculate D - values for $A' \cup B'$;
$i \leftarrow i + 1$; **Goto** Step3;
end if

Step5. Find k to maximise the partial sum

$$G_k = \sum_{i=1}^{k} g_i;$$

if $G > 0$ **then**
Move $X = \{ a_1, \ldots, a_k \}$ to B;
Move $Y = \{ b_1, \ldots, b_k \}$ to A;
Goto Step2;
else STOP
end if

end

Figure 3-1 Kernighan-Lin algorithm

The last step of the algorithm (Step 5) interchanges the first k pairs of vertices for which G_k is maximal, making the interchange of $\{a_1, \ldots, a_k\}$ with $\{b_1, \ldots, b_k\}$ permanent. The KL algorithm stops when the best gain found in an iteration is less than or equal to zero, that is, no further improvements can be obtained from vertex pair swapping.

Fiduccia-Mattheyses algorithm

The FM (Fiduccia-Mattheyses) algorithm [51] is one of the best-known, most widely referred and extended partitioning algorithm. It makes two modifications to the KL algorithm to improve the time complexity. Firstly, instead of swapping and locking a pair

of vertices, the FM algorithm considers and moves a single vertex, one with the highest gain in a partition-to-partition move. Secondly, also one important feature of the FM algorithm, a *bucket data structure* (see Figure 3-2) keeps sorted lists of candidates (vertices) for moving to the other partition. The vertices are sorted by order of maximal gain in a move, where a positive gain is an improvement in the overall solution while a negative gain degrades it.

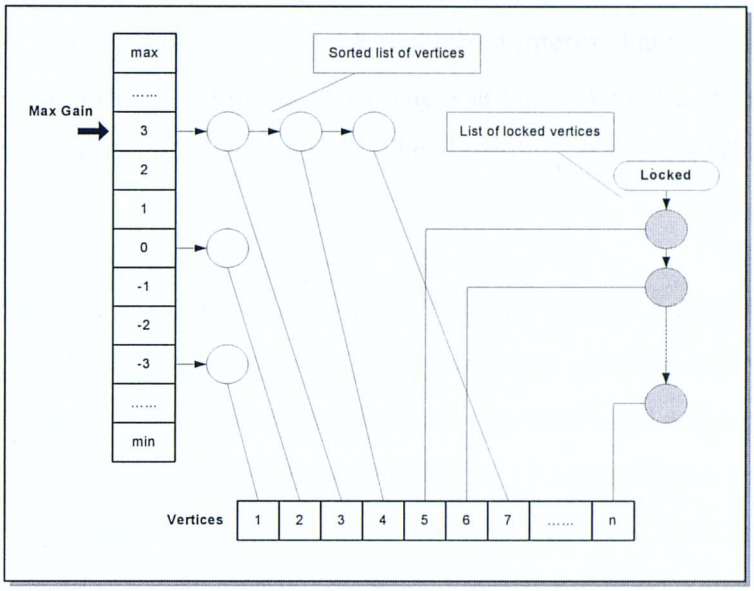


Figure 3-2 Bucket data structure in the FM algorithm

The FM algorithm is an iterative improvement algorithm, in that it starts with a random initial partition, and iteratively modifies the solution by a sequence of *moves* within a *pass*. To avoid having all vertices migrate to one partition, a balancing criterion is maintained. A user-specified balance factor r (called *ratio*), $0 < r < 1$, is used to ensure that only final partitions satisfying $|A| / (|A| + |B|) = r$ are acceptable, where $|A|$ and $|B|$ are the sizes of partitioned blocks A and B. A partition (A, B) is balanced if

$$(r * |V| - s_{max}) \leq |A| \leq (r * |V| + s_{max}) \tag{3.4}$$

where $|A| + |B| = |V|$, $s_{max} = \text{Max}[s(i)]$, and $i \in A \cup B = V$

All vertices are free to move initially, the move with the highest gain and does not violate the balance criterion, is selected and executed iteratively. The moved vertex (base vertex)

is locked and never selected and moved again during the pass, preventing the algorithm from selecting and moving the vertex moved in the previous iteration, thus the algorithm avoids executing in an infinite loop. The gain values of adjacent vertices affected by the base vertex move are updated after each move. The algorithm maintains a maximum gain index for each bucket data structure to keep track of the vertex with the highest gain. The algorithm continues with the execution of this iterative select-and-move sequence until no more unlocked nodes can be moved without violating the partition size constraints. At the end of a pass, the FM algorithm moves back to the best intermediate solution, allowing the algorithm to climb out of local minima. All vertices are unlocked and the best solution forms the starting partition for the next pass. The algorithm terminates when a pass fails to improve the solution.

A single pass of the FM algorithm using a graph with six vertices (labelled *M*-*R*) is illustrated in Figure 3-3. The algorithm starts with an initial partition in (a). The vertex with the highest gain (vertex *M* is the base vertex in (a)) is selected and moved to the next partition. The moved node is locked (shown shaded in the diagram) and the gain values of the adjacent vertices (vertices *O*, *Q*, and *R*) are updated. The select-and-move sequence ends in (g) and the intermediate result in (c) gives the best solution (with a cut-size of 3) forms the starting partition for the next pass.

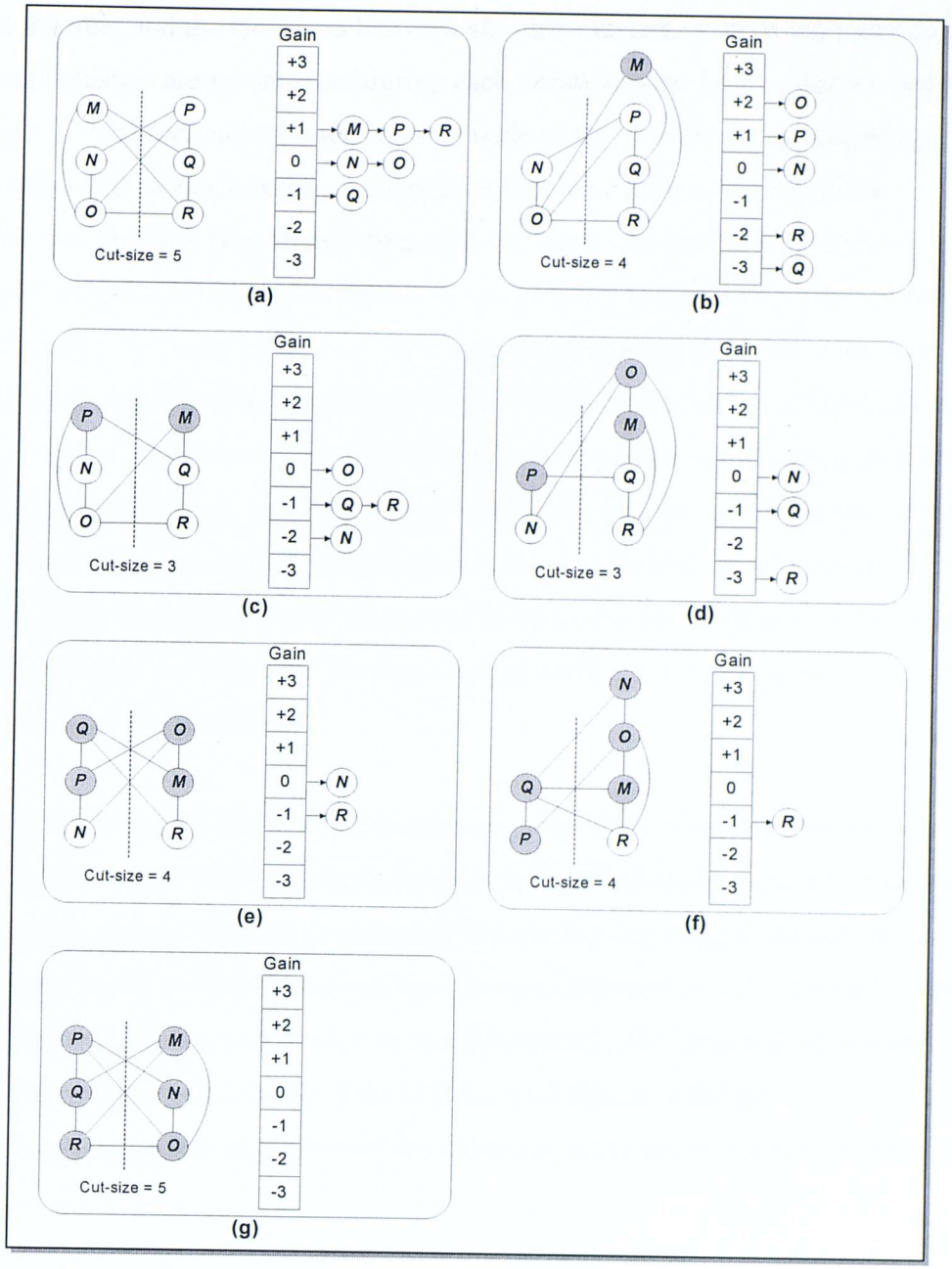


Figure 3-3 Example of a single pass in the FM algorithm

Clustering algorithm

A clustering algorithm groups a set of objects according to some measure of closeness. Strongly connected objects are merged into clusters; thereby condensing the overall design. A *hierarchical cluster tree*, with the original objects as the leaf nodes, is formed as the merging process is iterated until a single cluster is formed. The two 'closest' objects

(which can be individual leaf nodes or clusters resulting from previous iterations) are grouped together and the closeness between all other clusters, or between individual objects and clusters are recomputed during each iteration pass. Leaf nodes are considered to a height of zero and each non-terminating node of the tree has an associated height, which reflects the distance between the objects that have been merged into the corresponding clusters. Non-terminating nodes closer to the leaf nodes represent clusters in which the objects are strongly connected, and in contrast, non-terminating nodes with larger distance (i.e. closer to the root node) represents clusters in which objects are less strongly connected. *Cut lines* at different heights of the tree produces differing number and size of partitions, as each sub-tree below the cut line becomes one resulting partition. A small number of relatively large clusters are obtained when the cut line is close to the root, while a cut near the leaves will give a large number of relatively small clusters. The final partitions of the design are usually chosen by having cut lines at different levels and the resultant partitions from each cut are evaluated according to design criteria such as area or I/O utilisation of target devices.

Figure 3-4 illustrates the hierarchical partitioning algorithm using five vertices labelled *A* to *E*. Closeness values between pairs of objects are marked on the labelled on the edges connecting the objects. Objects or groups of objects that are merged in each succession are encompassed in the shaded cluster and the closeness between two clusters or between an individual object and a group of objects are recomputed. The closeness values can be the maximum, minimum, or average of the closeness of objects in the group. This closeness value has been estimated as the maximum closeness in the given example shown below.

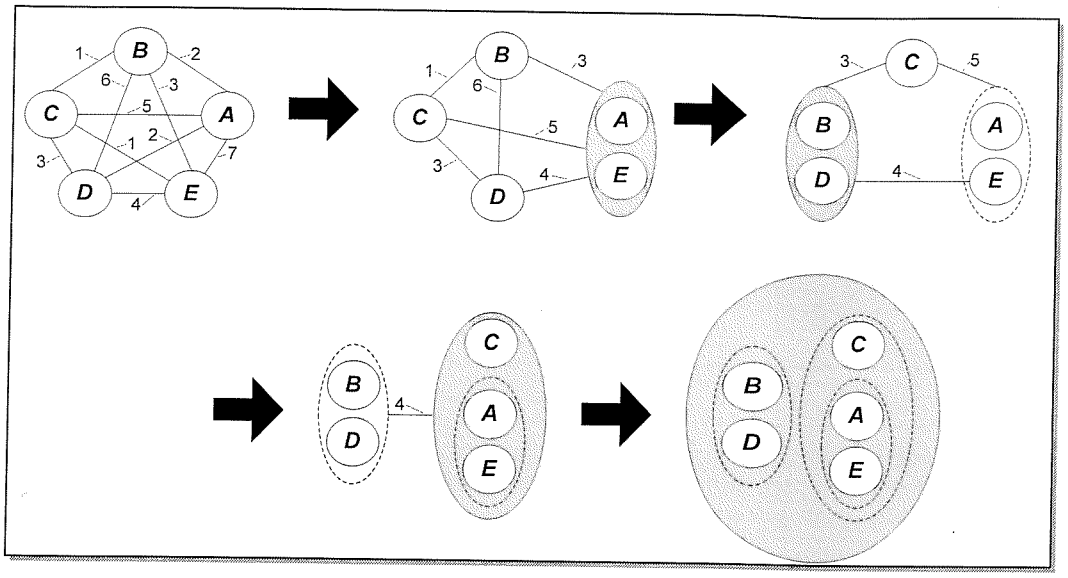


Figure 3-4 Successive steps in Hierarchical clustering

Figure 3-5 below illustrates the cluster tree produced by the hierarchical clustering algorithm. Partitions produced from each cut are shown the corresponding cut lines. The highest cut line, which is closest to the root node, produces a partitioning of two clusters with objects *A*, *E* and *C* in one cluster and objects *B* and *D* in the other. The lowest cut line produces a partitioning of five clusters with a single object in each.

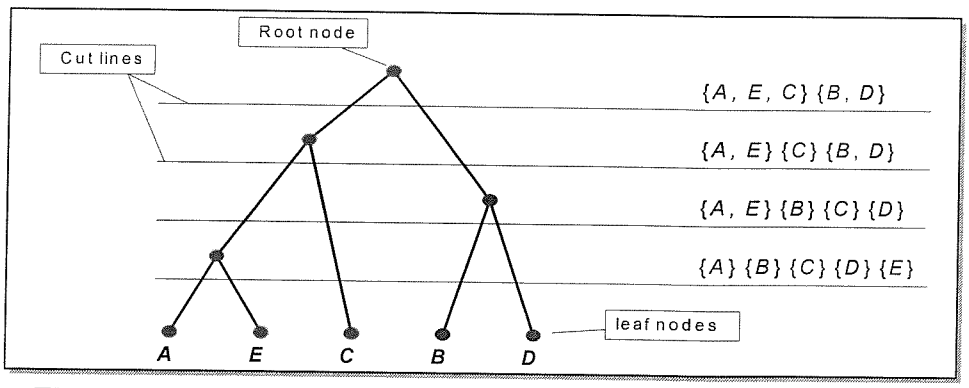


Figure 3-5 Cluster tree produced by Hierarchical clustering

Clustering algorithm can be applied at several levels of abstraction (i.e. gate netlist level, functional level, system level). Hierarchical clustering algorithms, which exploits the design structural hierarchy are reported in [70, 71, 73]. Fang and Wu [71] describes a hierarchical set-covering approach at the structural level for multiple-FPGA applications. The design is first converted into a three-level, *module*, *process*, and *function*, structural tree. An example of a structural tree with three modules (M1, M2 and M3), eight

processes (P1,1 to P1,3, P2,1,P2,2, and P3,1 to P3,3) and twenty functions (f1,1,1 to f3,3,2) is shown in Figure 3-6.

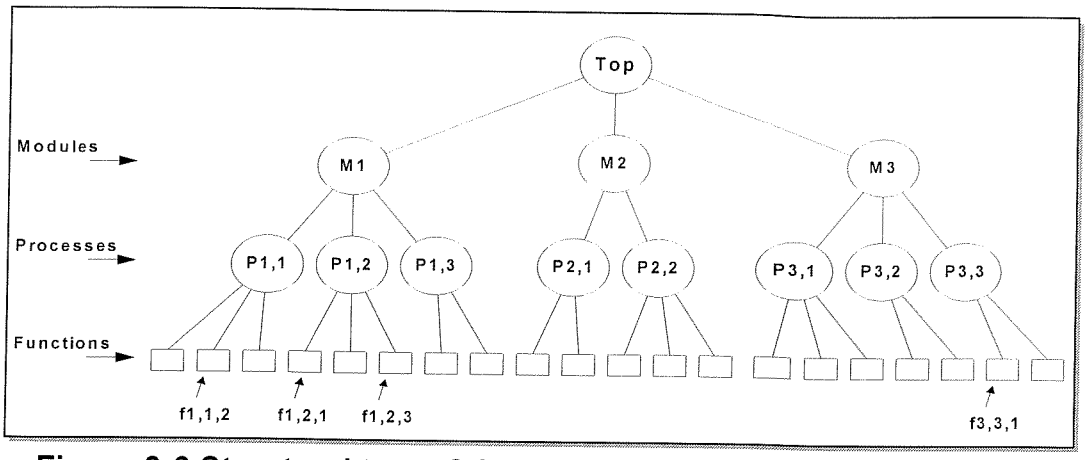


Figure 3-6 Structural tree of the hierarchical set-covering algorithm

The structural tree is next converted into a hierarchical connected graph illustrated in Figure 3-7 below. The covering process is performed on the hierarchical graph and it starts from the nodes with coarse granularity and then moves down to nodes with finer granularity when no more feasible covers can be found in the latter (higher) level. If modules M1 and M2 can be grouped into a set while satisfying the constraints, in this case, area and I/O of the target FPGA, then M1 and M2 can be merged into a set and targeted to the device. On the other hand, if the constraints are violated, then M1 and M2, then the set-covering algorithm tries to merge portions of one module with the other module to improve the covering size. For example, in Figure 3-7, module M2 and portions of M1 (process node P1,2 and functional node f1,3,1) are covered as a set.

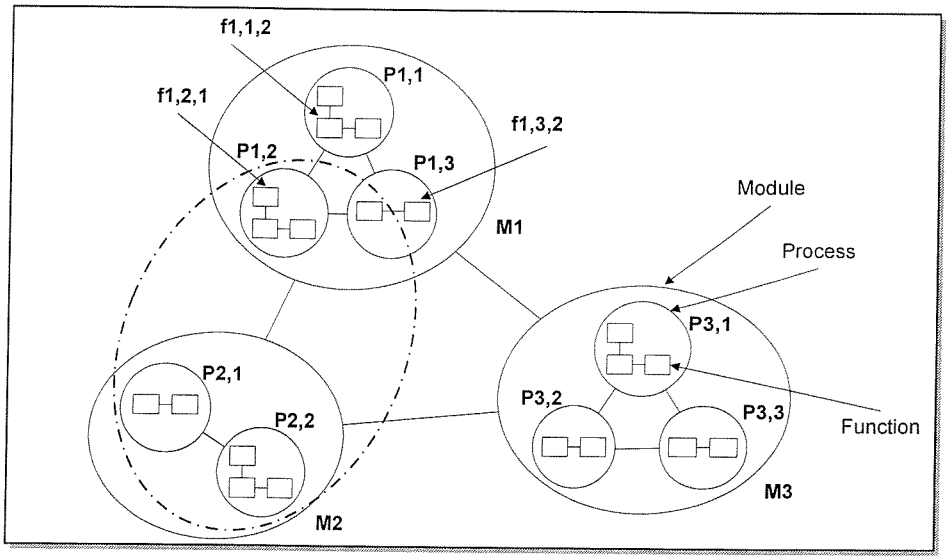


Figure 3-7 Hierarchical connected graph

Frank and Gajski [62] describes a *closeness metrics* for system-level functional partitioning, and a *N-way clustering* method is used to group close objects until there are only N groups remaining, where each group is then assigned to its own system (hardware or software) component. A clustering algorithm is often used with other partitioning heuristics to reduce the complexity of the design, thus reducing the computational effort, and even significantly improve the quality of the final solution [61, 74].

Simulated annealing algorithm

The simulated annealing (SA) algorithm works in a similar manner as described in Section 2.7.6.1, where the simulated annealing is one of the optimisation algorithms used within the MOODS synthesis core. In partitioning, the SA algorithm starts with a random partition, and iteratively improves the solution. A pair of vertices is selected from each partition randomly in each state, and compared with the previous state. The intermediate solution that results in an improvement in the overall solution is accepted and the move is made permanent. A predetermined number of moves are attempted at each temperature. When a move that degrades the overall solution is encountered, the probability of accepting the degrading move is given by:

$$P = \exp\frac{-\Delta E}{T} \quad : \quad \Delta E > 0 \tag{3.5}$$

where P is the resulting probability of accepting a degrading move, ΔE is the change in quality of the states, and T is the current temperature. This function ensures that the probability of accepting a degrading move decreases when the temperature decreases. The decision to accept degradations is made from the comparison of the probability threshold value and a normalised random number, and acceptance being granted when the generated random value is the smaller value between the two.

The simulated annealing algorithm generally produces good partitions but it is a very slow algorithm. The need to determine experimentally the several parameters of the SA

algorithm, such as the starting temperature, the cooling schedule, and the number of moves to perform at each temperature is another disadvantage in using the SA algorithm.

Genetic algorithm

Genetic algorithms (GAs) are inspired by Darwin's theory of evolution, where problems are solved by an evolutionary process that mimics the natural selection and genetics. The origins of GAs are often accredited to work carried out by J. Holland [60] in the early 70s. A genetic algorithm is a randomised parallel search method for a single or multi-objective function optimisation. A *population* of individuals is maintained by the genetic algorithm, where each individual is a potential solution for each generation. Each potential solution is evaluated to give some measure of its *fitness*. From this population, a new population is formed by selecting some of the fitter individuals (*selection*) and others are formed using genetic operators (such as *crossover* and *mutation*). After some generations the program converges and the best individual (hopefully) represents the optimum solution. The genetic partitioning algorithm given in Figure 3-8 is used in the partitioning of modules in a multi-FPGA system [75].

Genetic Partitioning Algorithm
K: population size (number of partitions in a generation)
S: percent of new generation produced by *selection*.
C: percent of new generation produced by *crossover*.
M: percentage of partitions *mutated*.

```
GA()
begin
Create a random set of K partitions
  Evaluate the fitness of each partition
    while (stopping criteria not satisfied)
      Create S percent of new population of partitions by selection
      Create C percent of new population of partitions by crossover
      Replace the current generation by new generation of partitions
      Mutate M percent of the current partitions
      Evaluate the fitness of each partition
      Save the partition with the best fitness
    end while
end
```

Figure 3-8 Pseudo code of the genetic algorithm

The GA starts with the generation of the initial population and a measure of the *goodness*, *fitness* (*f*) of a partition is quantified by an evaluation function.

$$f = \frac{1}{1 + \sum_{i=1}^{k} \Delta Pi + \sum_{i=1}^{k} \Delta Ai} \tag{3.6}$$

$$\Delta Pi = \begin{cases} 0 & \text{if } Pi < P\max \\ Pi - P\max & \text{otherwise} \end{cases}$$

$$\Delta Ai = \begin{cases} 0 & \text{if } Ai < A\max \\ Ai - A\max & \text{otherwise} \end{cases}$$

where k is the number of chips the design is partitioned into, P_i and A_i are the pin-count and area of partition i respectively, P_{max} and A_{max} are the constraints on the pin-count and area for the partitions. The fitness value is in the range 0.0 to 1.0; 0 indicates a bad solution and 1 indicates an excellent solution (i.e. all partitions satisfy all the constraints). The GA algorithm uses the following operators in order to produce the next generation of partition (population):

- ***Selection*** - This operator probabilistically selects highly fit individuals from the present generation and moves them into the new generation using the *roulette wheel* technique. Roulette wheel selection can be summarised in three steps [76] as shown in Figure 3-9.

Roulette Wheel Selection

1. Sum the fitness of all the population members and call the result total fitness.
2. Generate a random number n, between 0 and total fitness.
3. Return the first population member whose fitness, when added to the sum of fitness of preceding members, is greater than or equal to n (random number generated in step 2).

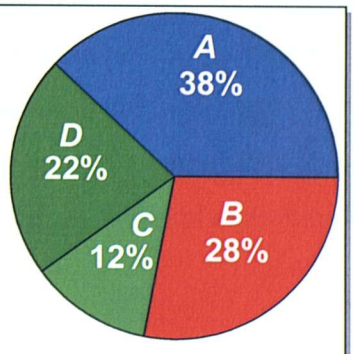


Figure 3-9 Selection using roulette wheel technique

The roulette wheel is an imaginary wheel which is split into as many parts as the population. On this wheel, each individual is assigned an area which is proportional to the relative fitness of the individual with respect to the overall population. Figure 3-9 gives an example of a roulette wheel with four individuals – *A*, *B*, *C*, *D*, and relative fitness of 38%, 28%, 12% and 22% respectively. If the wheel is spun, when the wheel stops, the probability that the arrow would be on *A* is 0.38, *B* is 0.28 and so on. This means that the probability of *C* (the predicted worst individual to lead to the optimal solution) being selected is the minimum and the probability of *A* (the predicted best individual to lead to the optimal solution) being selected in the maximum. The effect of selection ensures that good individuals in the search space are preserved and search continued from those individuals to look for a better solution. It is important to note that *selection* does nothing to explore the unexplored regions of the search space. Searching of unexplored regions is mainly achieved with the crossover operator and to a lesser extent with the mutation operator. *S* percent (typically 20 to 40 %) of the partitions in the new generation are created with this *select* operator.

- ***Crossover*** - This basic genetic operator probabilistically selects two highly fit (*parent*) partition structures from the current population, exchanges information between them and produces two offspring (*child*) structures. The significance of this is that the offspring structures represent two points (or solutions) different from the parent points in the search space, which probably represents some unexplored points in the design space. An example of *uniform crossover* is illustrated in Figure 3-10. Uniform crossover starts by selecting probabilistically two highly fit parent structures, *Parent 1* and *Parent 2* for mating. The second step is to generate a binary string template whose length is the same as the number of elements in the design. The bits in this crossover template are randomly selected to be either 1 or 0. The offspring of the parents are produced using the randomly generated crossover template. Figure 3-10 shows two parent structures, which are possible partitioning solutions for 10 components (R_0, R_1, R_2, …, R_9) into three target devices (Chip0, Chip1 and Chip2). An explanation on how the two offspring of the parents are produced is given below:

  Child 1 creation: If the i^{th} bit in the crossover template is a 1, then the i^{th} component of the design is placed in the same partition as it was in *Parent 1*, and if the i^{th} bit is a 0, then the component is placed in the partition as it was in *Parent*

2. For example, Bit 0 of the crossover template is a 0, component R_0 is placed in Chip2, the same partition as R_0 occurred in *Parent 2*. Bit 1 of the template is a 1, component R_1 is placed in Chip1 of *Child 1*, which is the same as it occurred in *Parent 1*.

Child 2 creation: The creation of Child 2 follows a similar process. If the i^{th} bit in the crossover template is a 1, then the i^{th} component of the design is placed in the same partition as it was in *Parent 2*, and if the i^{th} bit is a 0, then the component is placed in the partition as it was in *Parent 1*.

This *crossover* operator creates *C* percent (typically 60 to 80%) of the partitions in the new generation.

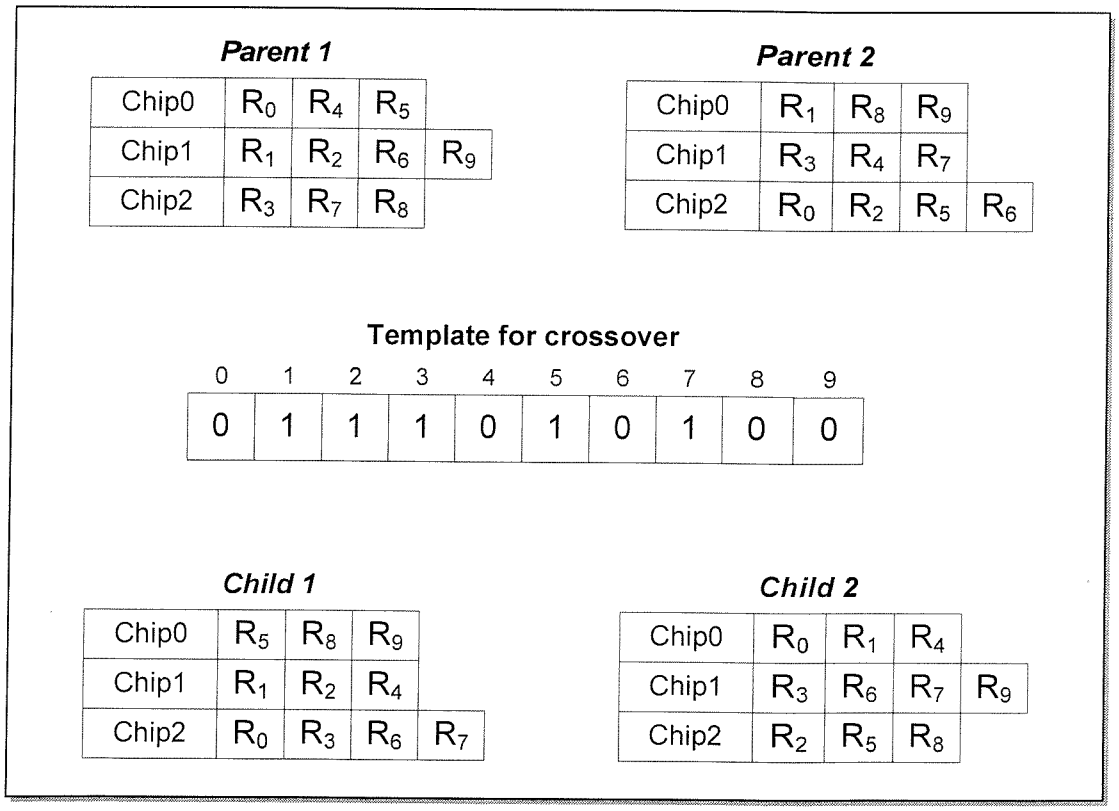


Figure 3-10 Example of uniform crossover

- ***Mutation*** – The *mutation* operator is introduced as a means to help the genetic algorithm avoid local optima. The mutation operator is invoked after selection and crossover. The mutation operator selects a partition structure probabilistically and

moves a design component from some randomly selected segment in the partition to another randomly selected segment. If the fitness of the mutated structure is low, it would most likely be eliminated in subsequent generations. However, if the fitness of the mutated structure is higher, then the probability that this structure will survive and lead to a better solution is high. This mutation operation is applied to M percent (typically 20 to 25%) of the partitions in the new generation.

The genetic algorithm terminates when a termination criterion (or a required fitness value) is met. Criteria such as computational time, number of generations to be searched or a limit on the global optimality such as the total number of interconnection wires or total number of chips are also specified. Multiple objectives can be assigned a weighting value to prioritise the user-defined objectives in the computation of the fitness value. Recent work on multi-FPGA partitioning using the GA algorithm can be found in [77, 78].

3.3 Multi-FPGA synthesis systems

The preceding section gives an introduction to the various methods and algorithms of partitioning in a general context. This section describes partitioning of multi-FPGA systems using some of these partitioning algorithms or a combination of algorithms used in multi-FPGA synthesis tools. A number of multi-FPGA synthesis systems exist, both commercial and academic. Some of the commercial systems are: Aptix Corporation Design Pilot™ [79], Auspy Development Inc. Auspy Partition System II [80], and Synplicity Certify® [81]. None of the mentioned commercial tools perform the partitioning at the behavioural level. All three tools perform partitioning at the register transfer level, and Auspy Partition System II also supports partitioning at the gate-level. Some academic tools are: COBRA-ABS [72, 82], SPARCS Project [68, 83] and related work in multi-component partitioning and synthesis [66, 67, 84], ISyn [70, 71, 85], SpecSyn [63, 65, 86, 87], CADDY-II [88].

3.3.1 COBRA-ABS

The COBRA-ABS (Column Oriented Butted Regular Architecture – Algorithmic Behavioural Synthesis) high-level synthesis tool developed at the University of Aberdeen has been designed to synthesise digital signal processing (DSP) algorithms specified in C, and target onto multi-field programmable gate array custom computing machines (FCCMs). The synthesis tool takes as input an FCCM architectural file and a datapath-library description file, in addition to the input algorithm description described in C. Information in the FCCM architecture file, which specifies the target FCCM, in terms of the FPGA devices, custom/ASIC arithmetic resources, inter-FPGA routing (point-to-point and bus based), FPGA-to-memory routing, and associated communications delay. The low-level datapath-library description file contains characterisation data about the RTL modules available and cost (in area) and timing characteristics (in clock cycles). The FCCM target information and the low-level library characterisation data are fed into the optimisation process driven by a simulated annealing algorithm.

The target architecture of the tool is based is on a partitioned VLIW (Very Long Instruction Word) style architecture, where each FPGA holds a single "RISC (Reduced Instruction Set Computer)-like" register-file based, load-store processor, with a bus-based architecture and a set of functional units.

The C function inputs forms the *basic blocks* in COBRA-ABS, each of which are represented by dataflow graph and controlled by a corresponding control-flow graph. The datapath space model [89] formed a three-dimensional space in which the lifetimes of variables are optimised. The optimised dataflow implies the hardware that is required to create, transfer, store and consume the data. The variable dimension represents the explicit and implicit *data* in the behavioural description. The processor dimension directly relates to the "RISC-like" processors and hence represents the *partitions*. The datapath space (dp-space) can therefore represent data flow in time and across partition boundaries.

The three-dimensional model was extended to a four-dimensional model to allow conditional branches and loops of the basic blocks. The overall datapath for each processor is the superposition of all the datapaths, which would be implied by each of the 3-D sub spaces isolation. The conceptual view of the superposition in the four-dimensional datapath space is illustrated in Figure 3-11.

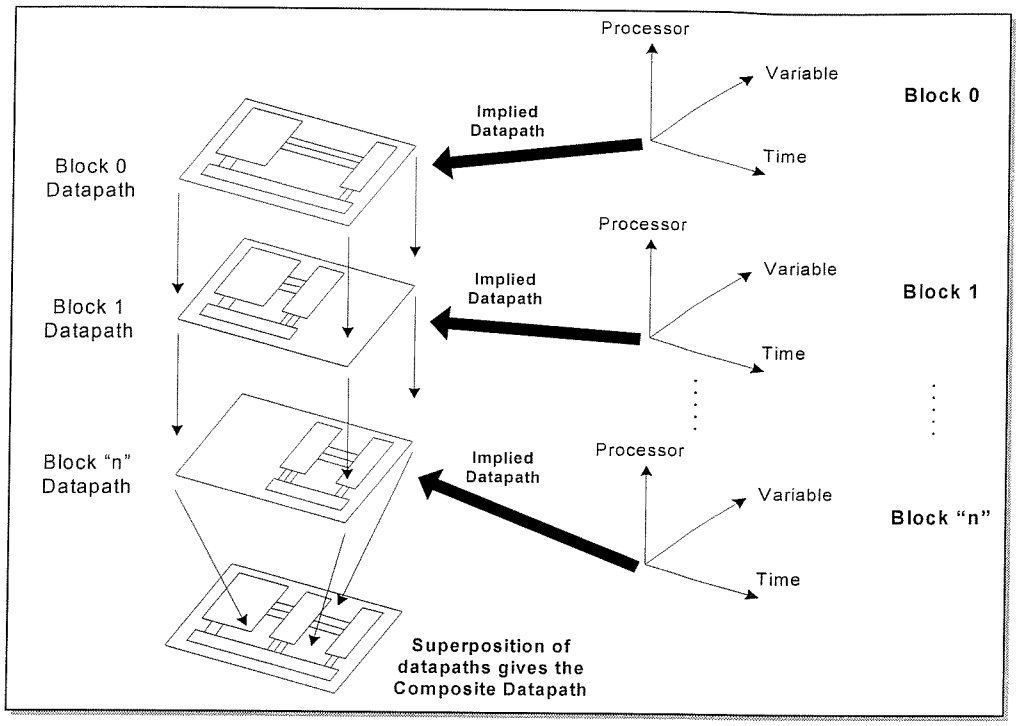


Figure 3-11 Conceptual view of superposition in 4-dimensional datapath space

The dp-space model is composed of a *behavioural layer* and a *structural layer*. A number of the "entities" representing the required behaviour are mapped to the 3-D dp-space. These entities are: input node, output node, functional-unit node, memory write, memory read, and global bus transfers. The DFG is transformed into a graph of interconnected dp-space entities and mapped to the *behavioural layer*. The *structural layer* administers the implication of hardware units and use of fixed FCCM resources. The cost of a dp-space configuration is measured in both the behavioural and structural layers, and the simulated annealing process adjusts the dp-space configuration, in the aim of finding the fastest implementation, which will fit on the FPGAs.

The synthesised output of the algorithm can be visualised as one 3-D block of dp-space flowing in time, into the next, with data passing seamlessly between blocks. The concept of the "pluggable block" was developed so that blocks, which can potentially "interface in time", have compatible interface on their dp-space variable-processor planes.

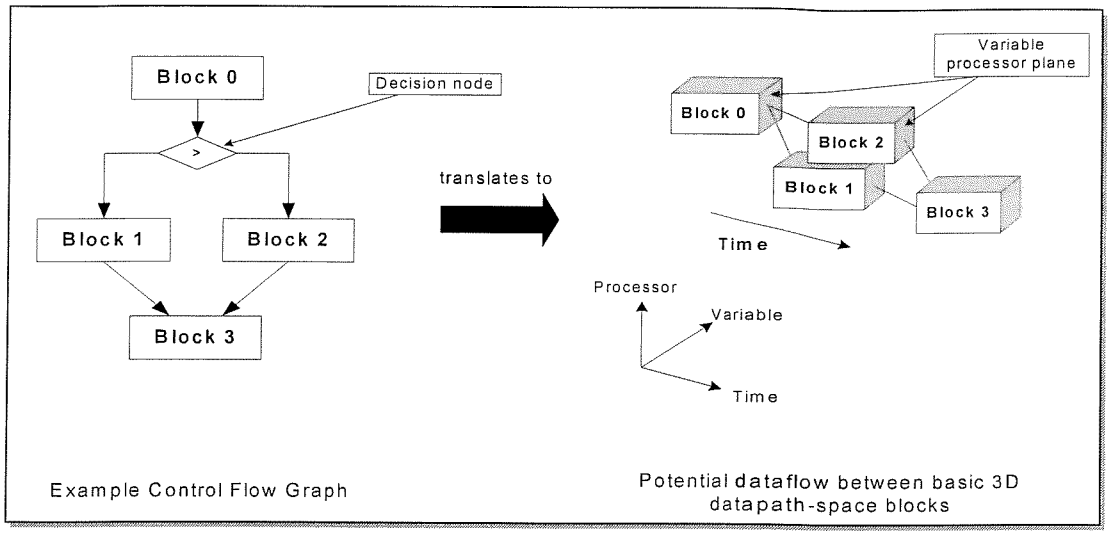


Figure 3-12 Pluggable 3-D block concept

COBRA-ABS provides a powerful high-level synthesis environment for DSP algorithms, specified in C. However, the run times reported (in [72, 82]) are rather high (> 10 hours) because of the simulated annealing algorithm, which forms the core of the synthesis optimisation process. The other point worth noting is the "pluggable block" concept is highly dependent on the number of I/O resources between each block that resides in different FPGAs, and this will impose an upper limit on the number of buses or point-to-point interconnects in the fixed board-level target architecture [90].

3.3.2 SPARCS

The Synthesis and Partitioning for Adaptive and Reconfigurable Computer Systems (SPARCS) [67, 68, 83] partitioning and synthesis framework was developed at the University of Cincinnati. The behavioural input designs are specified in either subsets of VHDL or C, and translated into an equivalent Control Data Flow *Block* Graph (CDFG), where each *block* contains a simple dataflow graph that captures the operations, and the edges between blocks represent the data and control flow across blocks. Each block is viewed as an *atomic* element that cannot be partitioned onto multiple FPGAs. The control flow at the end of the block can conditionally branch into one of the mutually exclusive blocks connected to it. The control flow also permits loops in the block call graph. The block call graph represents a single thread of control where all blocks are mutually

exclusive in time. Each of the partition CDFG contains a subset of blocks, which is synthesised into an RTL design for the corresponding device in the multi-FPGA architecture, with a single finite state machine controller and datapath resources shared by blocks within the same partition.

The "partitioner" that performs the partitioning of the CDFG is tightly integrated with the high-level exploration engine, whereby the partitioner always communicates any change in the partitioned configuration to the exploration engine. A four-dimensional design space model was used to represent the overall design so that the exploration engine has a partitioned view of the behaviour. Each partition segment, consisting of a set of operations, is represented by a traditional three-dimensional design space illustrated in Figure 3-13. The set of all partition segments of the design behaviour forms the fourth dimension. An example of the four-dimensional design space for design behaviour with two partitioned segments is illustrated below in Figure 3-13.

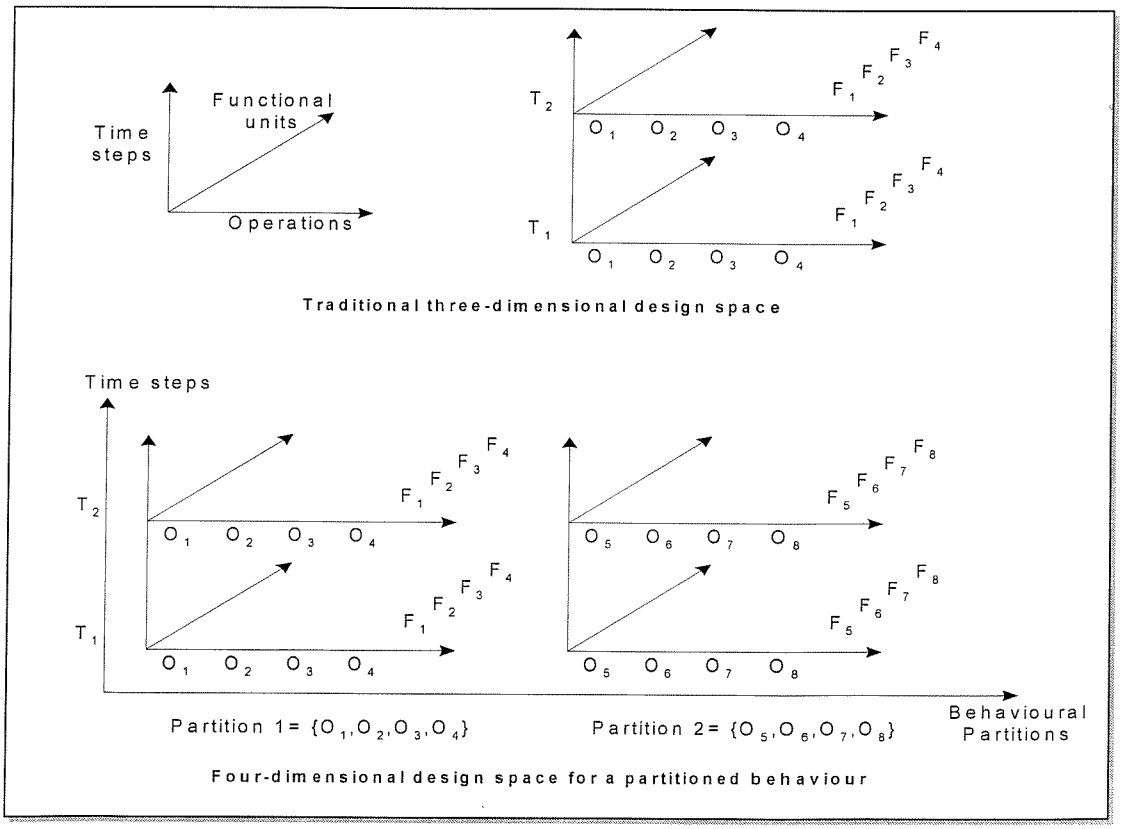


Figure 3-13 Four-dimensional design space for a partitioned behaviour

The functional units and operations are mutually exclusive between the partitions. However, the time steps span across the partition segments because the synthesised RTL output, one for each partition, is controlled by a synchronous FSM controller clocked by a single common (global) clock.

A multi-partition exploration algorithm performs an iterative exploration of blocks where the schedule of a block is either relaxed or tightened such that the design constraints are best satisfied. Relaxing (increasing) the schedule length could reduce the area of a partition and increase the latency of the entire design and tightening the schedule works vice versa. A collection of cost functions are used to sort and prioritise the blocks and guide the exploration engine to perform the area/latency exploration.

The exploration algorithm is independent of the partitioning algorithm used to obtain the partition segments. Synthesis results of SPARCS with partitioners using algorithms based on Fiduccia-Mattheyes (FM) partitioning algorithm and simulated annealing are given and it has been reported in [68] that the run times needed to find constraint satisfying solutions for a similar board architecture are much lesser than those reported in COBRA-ABS [72] described in the previous section.

The four-dimensional design space global technique with an integrated synthesis and partitioning model in SPARCS has provided a fast and efficient environment to generate constraint-satisfying solutions targeting a multi-FPGA architecture. In a similar manner to the COBRA-ABS, the implication of a partitioned design is explored with a fourth dimension. However, SPARCS also does not allow performance trade-off against the number of interconnecting I/O resources between the devices in the fixed architecture.

3.4 Data communications and communications synthesis

Data communications is fundamentally a simple operation, where data is sent from one point to another. A communications protocol is a specification of events and timing requirements in transferring information.

In a multi-FPGA system, data is sent from one FPGA device to another FPGA device. It is possible to have direct pin-to-pin connection mappings [91-93] on the FPGAs if both devices access the same signal. The signal value is changed in one device, passed on through the direct pin-to-pin connections, and updated in another device. However, the number of I/Os available on the I/O constrained FPGAs may not be sufficient to accommodate all the signals in the design. Another significant disadvantage of the multi-FPGA system is the lower speed of operation compared to a single chip implementation. The programmable features and the associated programming circuitry require a large amount of the chip area. The switches have significant resistance and capacitance, which account for the low speed of operation [49].

The Virtual Wires project [94] carried out in the MIT Computer Architecture Group explores methods to overcome pin limitation in FPGAs. *Virtual wires* are created by multiplexing and pipelining inter-device I/O signals. A virtual wire represents a single connection between a logical output on one FPGA partition and logical input on another FPGA partition. Shift registers in the sending and receiving FPGA are configured into shift loops, storing logical outputs into shift registers at the sending end, and shifting them into shift registers on the receiving FPGA.

A bus based approach to overcome the I/O limitation was proposed by Vahid [95]. The approach uses a single bus, the *FunctionBus*, for implementing function calls among FPGAs. The FunctionBus architecture is shown in Figure 3-14. Inter-FPGA data and control-encoded address are sent over the *AD* lines of the bus, with two additional bi-directional control lines, *Areq* and *Dreq*, used to indicate a valid address and a valid data

on *AD* respectively. Vahid also demonstrated techniques to trade off performance for even more I/O reductions using the FunctionBus.

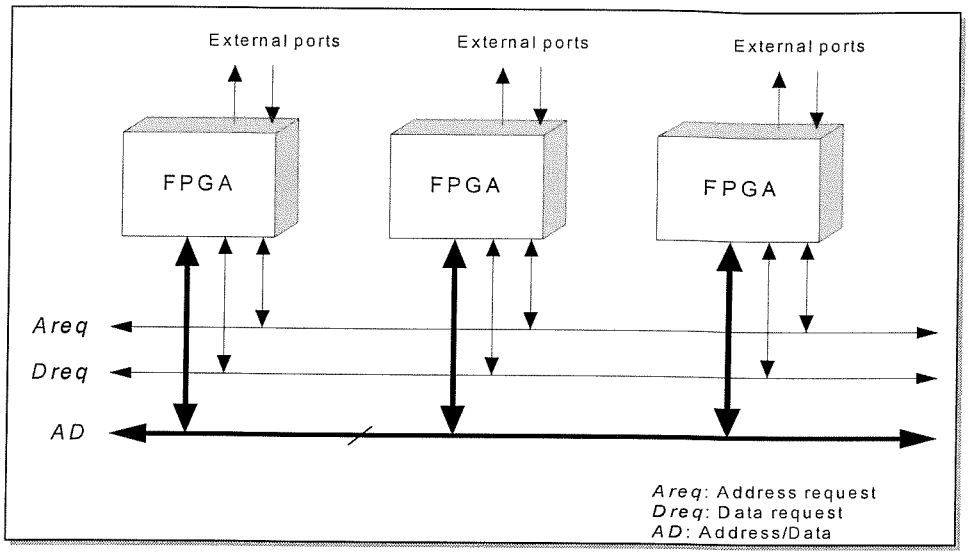


Figure 3-14 FunctionBus architecture

3.5 Data synchronisation over multiple clock domains

"Moving information from one clock domain to another is rather like descending into Dante's inferno. All sorts of evils lie in wait to beset the naïve." [96]. Data communication between two independently clocked domains can result in the data metastability [97-100]. Metastability can occur when an input to a register (flip-flop) is not synchronous to the clock, which can result in setup or hold time violations. Metastability is caused when the asynchronous input changes too close to the clock edge; this input to the register is not a stable high or low value during the register setup time. The flip-flop does not know if it is to change state or not, and may enter the metastable state, with the output not being logic High nor Low. Even though the flip-flop will eventually settle in a stable state after some period of time, this can still cause a system failure if the flip-flop has not left the metastable state by the end of the system's clock period. Figure 3-15 illustrates a simple two flip-flop (double buffer) synchroniser, which is typically sufficient to remove all likely metastability.

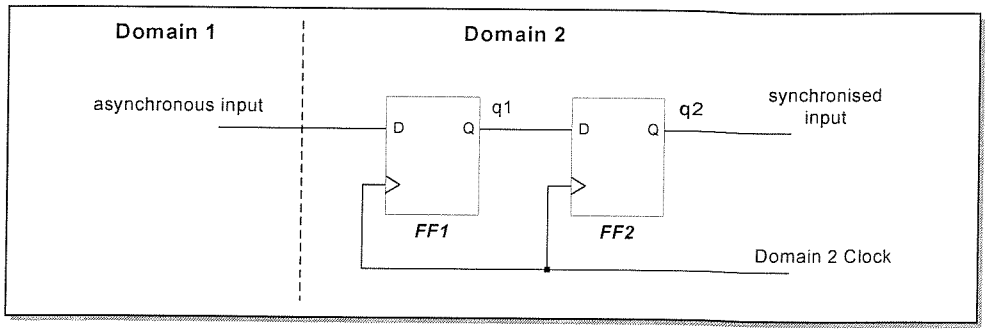


Figure 3-15 Double buffer synchroniser

It is still possible that a synchronisation failure can occur and this failure probability can only be determined statistically, and the generally accepted equation for Mean Time Between Failure (MTBF) [98] for a metastable flip flop is given by:

$$MTBF = \frac{e^{K_2 * \tau}}{F_1 F_2 K_1} \tag{3.7}$$

where K_2 is the register parameter that describes the speed with which the metastable condition is resolved. τ is the time delay for the metastability to resolve itself (resolution period), K_1 is another register parameter that represents the metastability-catching setup time window (i.e. the likelihood of the register going into the metastable state). F_1 is the clock frequency of the synchronisers and F_2 is the average frequency of the asynchronous input changes. Using values $K_1 = 10^{-10}$s and $K_2 = 19.4$/ns based on the Xilinx XC4005E-3 given in [98], this gives a MTBF of $0.0001\ e^{19.4*\tau}$ based on a clock frequency of 100 MHz and asynchronous input changes at a frequency of 1 MHz. With a τ value of 9 ns (a resolution period slightly less than the clock period), the MTBF value is $6.73 * 10^{71}$ seconds. The probability of failure increases rapidly when the number of asynchronous inputs and clock frequency increases. For example, a clock frequency of 1 GHz, with asynchronous input changes at a frequency of 100 MHz and a τ value of 0.9 ns, the MTBF value is only 3.83 seconds. Later results on the MTBF for newer Xilinx devices were published in [100] and the MTBF value exceeds millions of years when granted 2 ns of extra flip-flop settling delay. For the same operating conditions of clock frequency of 100 MHz and asynchronous input change at a frequency of 1 MHz, the MTBF of newer Xilinx Virtex-II Pro devices exceeds billions of years compared to the older Xilinx XC4005E-3 device.

Three common techniques of transferring data between clock domains are (1) pass data across clock domains using handshake signals, (2) use a Micropipeline or, (3) use an asynchronous FIFO (First In First Out memory) to transfer the inter-domain data.

3.5.1 Handshaking data between clock domains

Data is transferred across clock domains using additional handshaking control signals, where the sender places data onto a data bus and then asserts a request (*req*) signal to the receiver through a synchroniser. When the *req* signal is recognised in the receiving domain, the receiver clocks in the data into a register (or latch), and asserts acknowledge (*ack*) signal through a synchroniser to the sender in the domain of the sender. Handshaking of data is commonly used to pass data between asynchronous circuits, and two common signalling protocols are illustrated in Figure 3-16.

Figure 3-16(a) illustrates the two-phase signalling scheme, where the signal levels of the handshake signals are unimportant; it is the signalling event (i.e. a transition, either a rising edge, or a falling edge on the handshake signals) that is significant. The two-phase signalling protocol uses a non return-to-zero scheme. The four-phase signalling protocol illustrated in Figure 3-16(b) uses the signal levels of the handshaking signals to indicate the validity of data and its acceptance by the receiver. This protocol uses a return-to-zero scheme, where the *req* and *ack* signals end up in the same signalling level after a data transfer as they were before the transfer. This protocol thus uses twice as many signalling events for every data transfer as the two-phase counterpart.

Control logic for the four-phase protocol is often simpler than that needed in a two-phase system because the signalling lines can be used to directly drive the level-controlled latches (or registers) (discussed later in Section 5.5). It is also common that data lines are triple-buffered using triple buffer synchronisers. The extra buffering stage of the data lines ensures that valid data is 'definitely' on the data bus when the data request signal is asserted. This prevents a receiver that has an input request line with a shorter propagation delay from reading in the wrong data. The biggest disadvantage to using handshaking is

the latency required to pass and recognise all of the handshaking signals used for each data transferred.

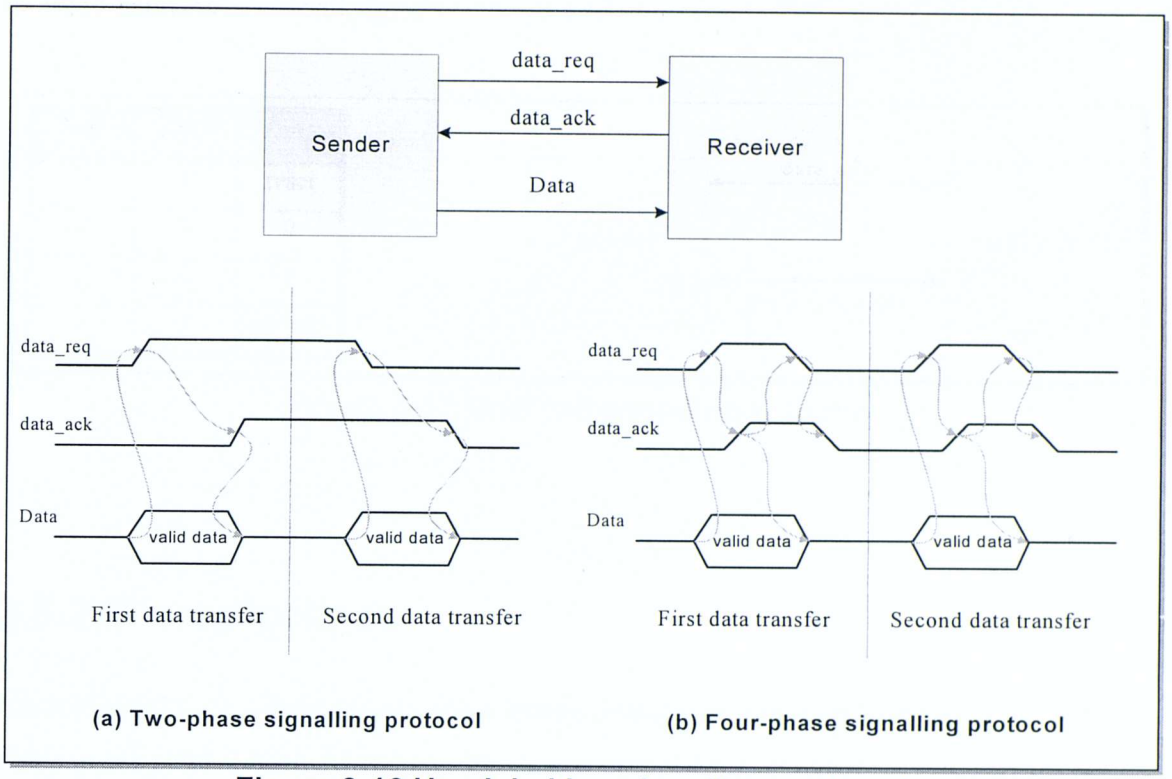


Figure 3-16 Handshaking signalling protocols

Single-rail and **dual-rail** encoding are two commonly used encoding schemes [101] for data representation. Single-rail encoding [102], which is conventionally used in synchronous designs, uses a single wire for each bit of information. Additional handshaking control signals are used to indicate data availability and its acceptance by the receiver. This scheme is also known as **bundled-data** approach. Dual-rail encoding [103] scheme uses two wires to represent each bit of information. Dual-rail circuits can have bundled control signals, however timing information is implicit in the code and the *req* signal required to indicate data readiness is thus not necessary. Figure 3-17 gives the list of values associated with the signal levels of the two wires (W0 and W1) in a dual-rail encoding circuit, and the corresponding interfaces between the sender and receiver.

The main advantage of dual-rail circuits is that they are delay-insensitive [103]. Delay-Insensitive (DI) circuits operate correctly regardless of delays in components and connections. They have the disadvantage of having a significantly larger area overhead in both the number of wires and the data transfer completion detection logic. Single-rail

circuits have the advantages of being smaller and faster as compared with their dual-rail counterparts. The disadvantage of single-rail circuits is they require tighter timing constraints (e.g. validity (*req*) control and data delays must be matched) when used with bundled handshake control signals.

W0	W1	Value
0	0	reset
0	1	0
1	0	1
1	1	unused

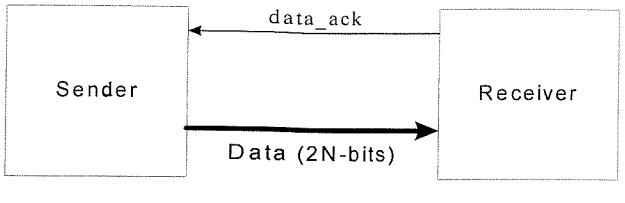
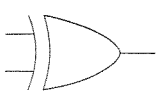


Figure 3-17 Dual-rail encoding scheme

3.5.2 Micropipelines

Micropipelines are a style of two-phase bundled-data pipeline introduced by Sutherland [104] in his 1988 Turing Award lecture. A micropipeline is an event-driven, self-timed asynchronous pipeline. Various simple event control module blocks are given in [104] to provide elemental functions such as merging and branching of the control flow. The micropipeline basic control modules are illustrated and described in Table 3-1.

Figure 3-18 shows a simple micropipeline without processing elements. The data path is composed of a set of event-controlled storage elements in series, while the string of Muller C-elements serves as its local timing control block. Delay elements (if required) ensures that the output request signals are asserted after the data is valid (e.g. R(1) is asserted only when data is ready at the output of the first storage element), so that the bundling constraint of the bundled-data protocol is met.

Event control module	Description
OR function 	The **OR** function for events is implementation using an exclusive-OR (XOR) gate. This is also known as a merge because it allows two event flows to merge into one. An event on either of the inputs causes a corresponding event to be seen on the output.

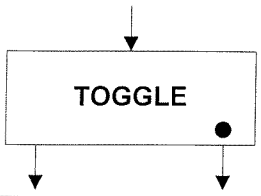
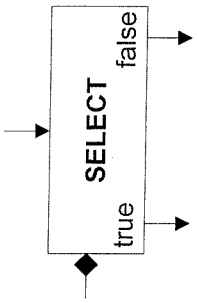
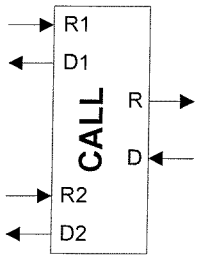
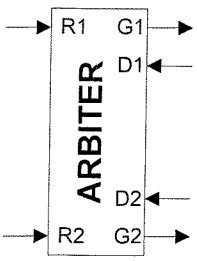
Event control module	Description
AND function 	The **AND** function for events is implemented using a MULLER C-element. A transition will occur on the output only when there has been a transition on both inputs. The Muller C-element is sometimes known as a rendezvous element because events are allowed to pass to the outputs only when all input events have arrived.
TOGGLE 	The TOGGLE steers input events alternately to the outputs. The first event is directed to the output marked with a dot after initialisation, the next event to the unmarked output and the sequence repeats.
SELECT 	The input event is steered to one of the two outputs depending on the Boolean input select value (indicated by the diamond head). The select signal must be available before the incoming event arrives, a similar requirement to the bundling constraint.
CALL 	The **call** module allows two mutually exclusive processes to access a shared resource (section of data path) or procedure, analogous to procedure calls in software. Unlike the previous modules, the call module operates on pairs of request/acknowledge (or done) handshaking signals. Incoming requests (either R1 or R2) are directed to the output request (R). The Call module remembers which of its inputs most recently received an event, and returns an acknowledge (done) event on the appropriate output acknowledge (either D1 or D2) signal. For the call module to operate properly, input request events have to be mutually exclusive.
ARBITER 	The **ARBITER** provides arbitration between two possibly concurrent asynchronous request events on its inputs (R1 and R2) and only passes one through at any time to the corresponding grant outputs (G1 and G2). Similar to a semaphore in software, it delays subsequent grants until it has received an event on the done wire (D1 or D2) corresponding to an earlier grant so that there is no more than one outstanding grant at a time.

Table 3-1 Description of the micropipeline event control modules

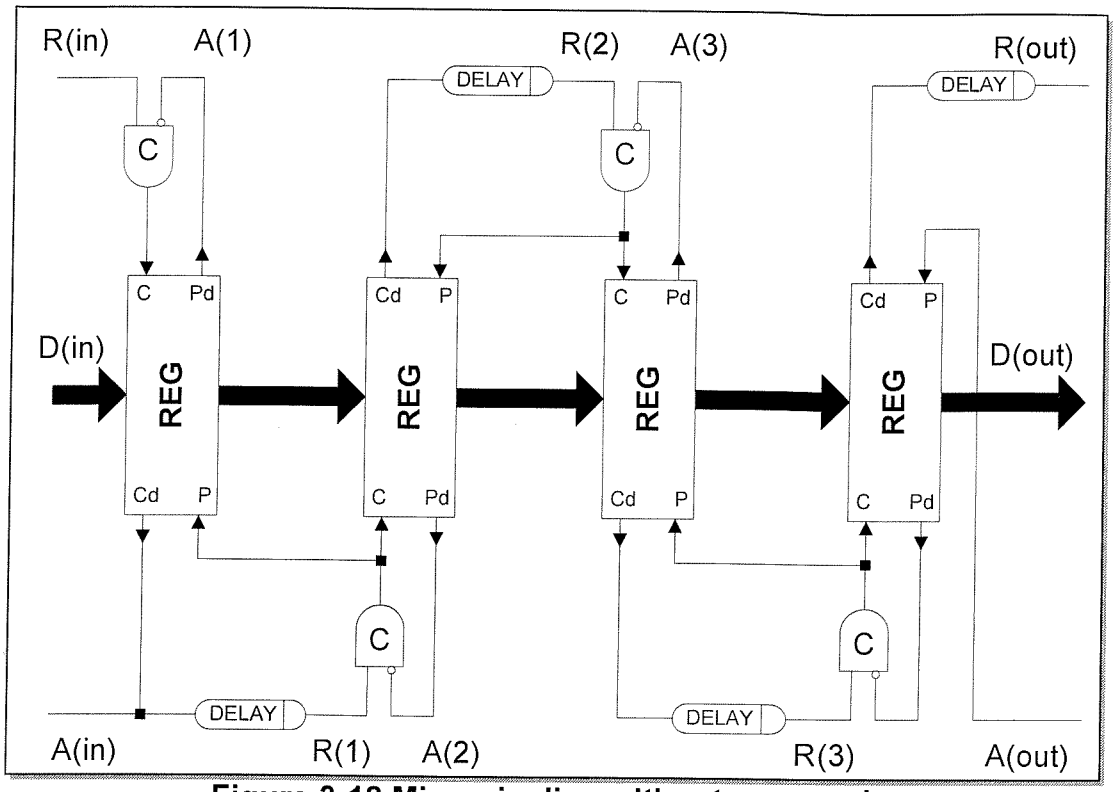


Figure 3-18 Micropipeline without processing

A major advantage of the micropipeline structure is the possibility of filtering out all the hazards in the logic blocks (i.e. removes the arbitration and synchronisation problem of two separate clocks at the input and output of the micropipeline). Another important feature is that micropipelines are automatically elastic. Data can be sent to or received from a micropipeline at arbitrary times. The basic event control modules of the micropipeline and the storage elements can be interconnected to form larger structures, which form the basis of more complex systems [105].

3.5.3 Dual port asynchronous FIFO

Another popular method of passing data between clock domains is using an asynchronous FIFO (First In First Out memory) [106-108]. A dual port memory is used for the FIFO memory.

The *write* port is controlled by the sender and data is written into the memory, one data word per write clock. The other port (*read* port) reads data out of memory, one data word per read clock. Two control signals are used to indicate if the FIFO is empty or full. The write and read *increment* signals are used to push data into the memory during a write cycle, or pop data from the memory during a read cycle.

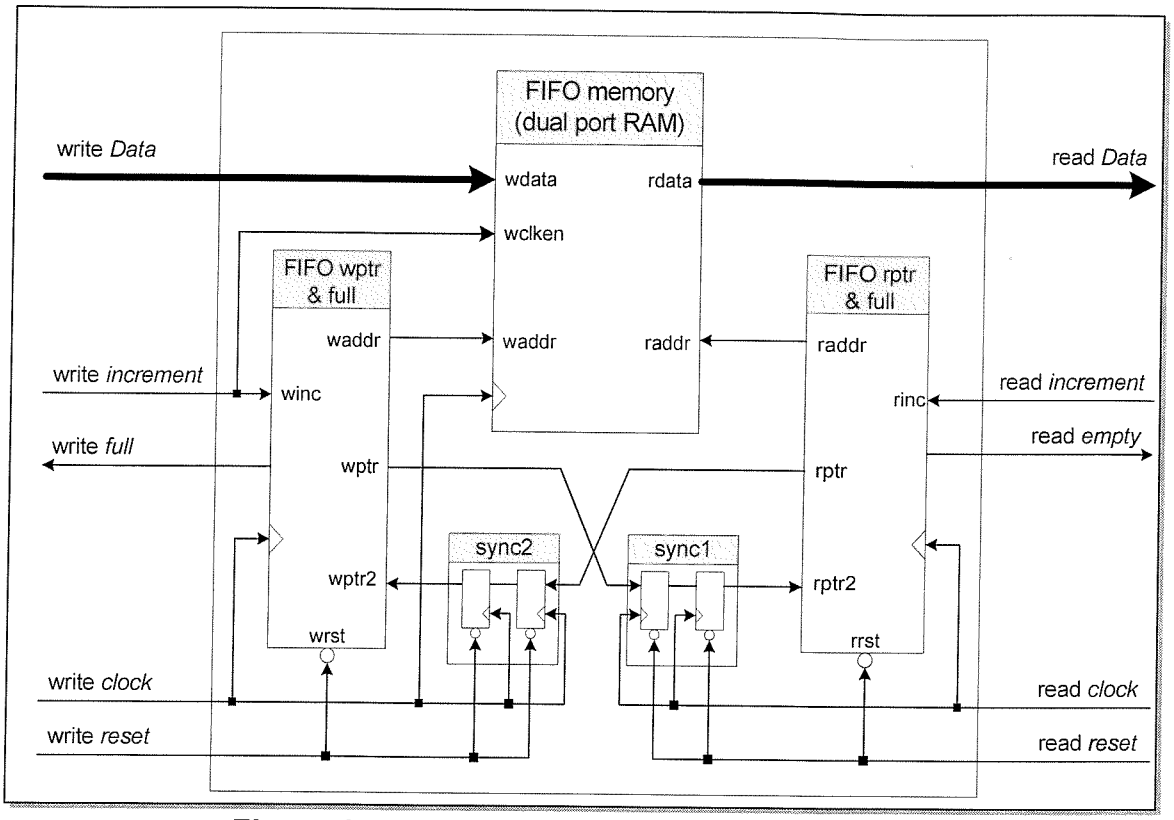


Figure 3-19 Asynchronous FIFO block diagram

Figure 3-19 above illustrates the blocks in the asynchronous FIFO design presented by Cummings of Sunburst Design Inc. in [106]. The five blocks in the asynchronous FIFO are:

- FIFO memory: This is a dual port RAM that is accessed by the write clock domain via the write port, and the read clock domain via the read port.

- FIFO write pointer and full (*wptr & full*): This block is mostly synchronous to the write-clock domain and it contains the logic for the FIFO write pointer (*wptr*) and

it generates a full (write *full*) signal to the write-clock domain when the FIFO is full. Gray coded addresses are created for writing to the memory and the FIFO write pointer is passed to the read-clock domain.

- FIFO read pointer and empty (*rptr & empty*): This block is similar to its write counterpart mentioned above. It is mostly synchronous to the read-clock domain and it contains the logic for the FIFO read pointer (*rptr*) and FIFO empty (read *empty*) signal generation. Gray coded addresses are created for reading from the memory and the FIFO read pointer is passed to the write-clock domain.

- Write-to-Read synchroniser (sync1): This block consists of a double buffer synchroniser that synchronises the write pointer (*wrptr*) into the read-clock domain.

- Read-to-Write synchroniser (sync2): This block is similar to its Write-to-Read counterpart described above. This block synchronises the read pointer (*rptr*) into the write-clock domain.

For a FIFO memory with (n-1)-bits address lines, giving a total of 2^{n-1} addressable locations, the read and write pointers are *n*-bits wide. The extra most significant bit (MSB) is used as a flag to determine if the FIFO is empty or full. When the pointers are equal, including the two MSBs, the FIFO is empty. The FIFO is full when the pointers are equal but not the MSBs.

The dual port memory asynchronous FIFO allows the sender to write data into the memory through the write port whilst the receiver reads stored data in the memory out from the read port concurrently. This has the advantage of reducing the latency in the overall system as the sender can send data into the FIFO independent of the receiver when the memory is not full. This reduces the possibility of blocking the sender if the receiver is not ready to receive the new data. However, careful speed matching of the sender and receiver and the depth of the FIFO have to be considered to reduce FIFO overflow and underflow conditions [109].

The main disadvantage of using the asynchronous FIFO in an I/O constraint multi-FPGA system for inter-device data transfers is the increased number of I/Os required for the

control, clock and reset signals compared to just a pair of handshaking signals in the bundled-data approach.

3.6 Design activity profiling

Design profiling is a process where a profiling tool generates and collects information on how a system operates and the resultant profile data is used to guide the profile-driven optimisation process to improve the system's performance.

Design activating profiling to obtain the usage and inter-communications between multiple processes is carried out with a full testbench of the system and the obtained information (profile) is used in the high-level synthesis and partitioning of the design itself. From the simulation of the structural VHDL design using a set of typical data to emulate the system, the profiler gathers the various event activities. The system is simulated with a testbench to generate activity information for all operations in the design and this information is used to guide the partitioner. This approach allows the user to provide the system with activity information in the most practical from, as a comprehensive test suite will almost certainly be created for most designs. Once a set of activity data has been generated, the operation need only be repeated if the behavioural design changes, and not on each synthesis run.

The activity data is fed into the partitioner during the partitioning stage and used in the assignments of *weights* on the edges of nodes in the partitioning graphs. Operations that interact intensively will have edges that are more heavily weighted and these edges are less likely to be cut by the partitioner. The atomic functional objects (processes, procedures, functions, shared variables, etc) that interact and communicate more often with each other are grouped into the same FPGA if the area permits. This reduces the off-chip interconnections and the inter-chip communication overheads associated with it.

3.7 Summary

This chapter focuses on the background material on multi-FPGA synthesis systems, with emphasis on partitioning and multi-FPGA synthesis systems. This chapter starts with an overview of the various partitioning algorithms, and an introduction of commercial and academic multi-FPGA high-level synthesis systems that exists. An introduction of techniques for inter-FPGA (cross clock domain) data transfers is also covered within this chapter. Multi-FPGA partitioning and the inter-domain data transfer forms the two main core components in the extension of the MOODS synthesis system to target multi-FPGA systems with asynchronous communications.

This chapter has covered the techniques and background material on *how* a design can be partitioned. The next chapter covers *when* to perform partitioning in the MOODS synthesis system. This deals with the implementation details of the partitioning enhancement in the MOODS synthesis system. Implementation details and signalling protocols to enable data transfers between clock domains are covered in more detail in Chapter 5.

Chapter 4

Multi-FPGA partitioning in MOODS

4.1 Introduction

Chapter 3 has provided an insight on *how* a design can be partitioned; this chapter starts with the selection of *when* to perform partitioning. The multi-FPGA partitioning enhancement to the MOODS synthesis system comprises two main stages: (1) High-level synthesis and partitioning, and (2) Interface generation. This chapter covers the generation of multiple structural VHDL outputs from a single behavioural VHDL description as illustrated in the shaded region of Figure 4-1.

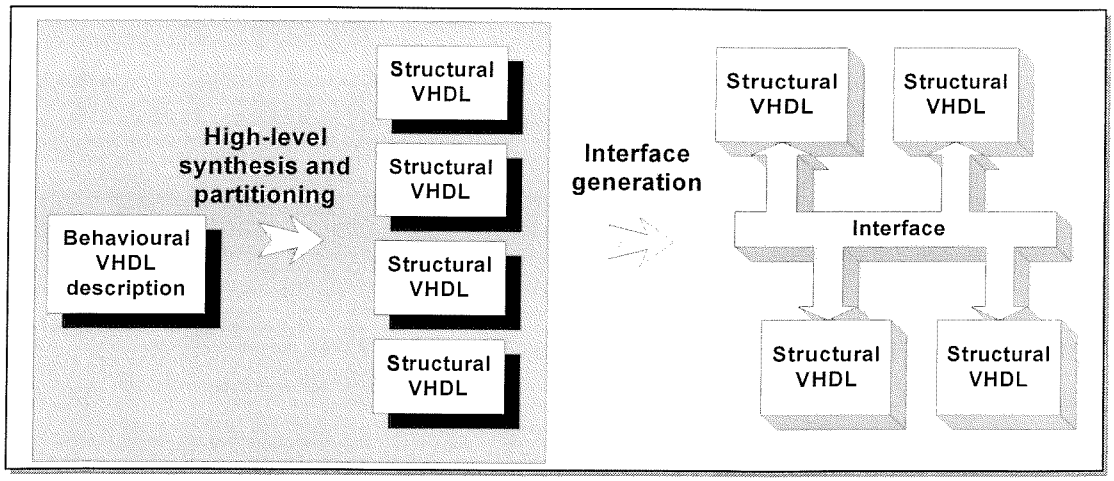


Figure 4-1 Generated system structure

Section 4.2 starts with a discussion on the various stages that the partitioning mechanism can be inserted and concludes with an insight on the partitioning granularity and insertion of the K-way partitioner as part of the partitioning enhancement in the MOODS synthesis

system. The section also presents the channel-based approach to handle inter-device data in the synthesised multi-FPGA design.

Section 4.3 introduces the module call graph representation and shows how a design is modelled using a module call graph. Implementation details and modifications of the partitioning algorithm are covered within Section 4.4. Section 4.5 describes design profiling in detail. Section 4.6 describes the modified ICODE modules, and the modifications made to the sub-module calling mechanism to support inter-FPGA module calls.

4.2 MOODS synthesis system with multi-FPGA partitioning

This section starts with the selection of the partitioning mechanism insertion into the MOODS synthesis system, which has an effect on the level of abstraction that the proposed partitioning algorithm is applied to. This affects the runtime and the granularity of the components that are being partitioned. Partitioning at the higher level of abstraction (e.g. at the system-level or algorithmic level), usually at a coarser granularity has fewer components to assign to partitions, compared to partitioning at the cell and netlist level.

4.2.1 Design partitioning phases in MOODS

The MOODS synthesis system comprises four separate sub-components, which perform the various tasks in synthesis as described in Chapter 2. There are several possible phases during the synthesis process where the partitioning mechanism (partitioner) can be inserted as shown in Figure 4-2.

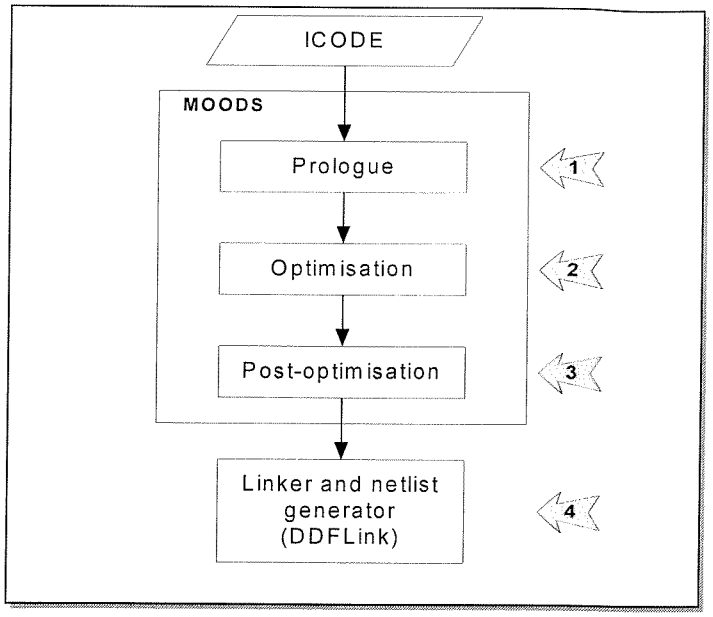


Figure 4-2 Insertion of K-way partitioner into the MOODS synthesis system

The following lists the four stages where the partitioner can be inserted into the MOODS synthesis system:

1. Prologue (Pre-MOODS optimisation):

Partitioning at this early stage provides the opportunity for the MOODS synthesis core to perform synthesis for each partition based on its own optimisation criteria. There are two different ways to target the partitioning at this stage. The first approach is to partition the ICODE file, where subprogram module sections of the original ICODE file are extracted and written to multiple enhanced ICODE (ICODE+) files, each ICODE+ file targeting a FPGA device. The ICODE+ files will contain extra partitioning information on the targeted partition, and communication interface details. The ICODE+ files are then synthesised separately to produce separate structural VHDL output files.

The second approach to partition the design is to partition the initial data and control path before applying the optimisation transforms to the partitioned data and control path structures.

An estimation mechanism is needed in both approaches to use the low-level information in the technology cell libraries to obtain an estimate of the size and delay of modules, which

is used by the partitioning algorithm. Partitioning the design at this stage with no information about the final optimised design means it is difficult to obtain an accurate and efficient partition, resulting in a low utilisation of the targeted FPGAs. It is possible to assign an FPGA with more modules than it can accommodate using the estimated sizes of the modules, in the hope that the optimisation stage in MOODS optimises the modules and the final design can fit into the allocated FPGA device. However, a design may have to go through multiple iterations of synthesis before each partition of the design can fit into the targeted devices.

2. MOODS optimisation:

MOODS optimisation is an iterative process whereby various transforms are used to modify the data structure and the optimisation algorithm controls the whole process, choosing which transforms to apply and where in order to achieve the user's target criteria.

Throughout the optimisation process, the low-level characterisation information from the technology cell library is used to provide accurate estimates for circuit performance. These figures are used by the optimisation algorithm to guide the selection and targeting of transformations in such a way as to move the implementation through the design space towards the cost objectives specified by the user. In a similar manner, these figures can also be used by the partitioning algorithm to guide the partitioning of the design and targeting of FPGA devices. Modules in the design may change and reduce in their sizes after each optimisation iteration and previous allocations of modules to partitions become inaccurate. The optimised design has to be re-partitioned within the optimisation loop, using the updated information of modules to guide the partitioning algorithm and allocate modules to partitions.

Using the existing simulated annealing within the MOODS synthesis core, the partitioning of modules over multiple target devices can be added as one of the objectives to be considered by the simulated annealing algorithm.

3. Post-MOODS optimisation:

This is the epilogue phase where MOODS "finishes" the design, converting any implicit and behaviour related parts of the data structure (such as multiplexers, and control/net

gating signal conditions) into explicitly described structures, and removes any redundant control or data path elements.

The partitioner is inserted at this stage to partition the optimised design and mark the control or data path elements needed for a partitioned design so that they are not removed when the control and data path are 'tidied' up in this stage. The extra logic created for the control and data transfers of a partition design may require some form of multiplexing logic, which is inserted together with the rest of the design. Thus the insertion of the partitioner at this stage removes the need for an extra stage to re-insert the control and data path elements, and the multiplexers required for a partitioned design.

4. Linker and netlist generation (DDFLink):

This is the last stage in the MOODS synthesis and the design is purely structural. The main disadvantage of inserting the partitioner at this late stage is in the breaking up of the structural design and the insertion of the extra logic needed for the control and data transfers in the partitioned design. The original control/net gating signal conditions has to be modified and updated to include the control conditions for inter-FPGA subprogram calls. The objective of partitioning at this cell/netlist is normally to group the allocated data path units and the synchronous FSM controller into partitions (which will fit on the targeted devices) and attempt to reduce the interconnections between devices.

4.2.2 Insertion of the partitioner into MOODS

The partitioner is not inserted in the pre-MOODS optimisation stage (stage 1 in Section 4.2.1) due to the lack of information about the final optimised design which makes it difficult to obtain an accurate and efficient partition. This can result in a low utilisation of the targeted FPGAs which will require multiple partitioning and synthesis iterations to get an optimised multi-FPGA implementation. The post-MOODS optimisation stage (stage 3 in Section 4.2.1) is not selected as the insertion of the partitioner in this stage does not allow further optimisation on selected modules after partitioning as the MOODS synthesis

core where design exploration and optimisation is performed in stage 2 (MOODS optimisation stage).

The partitioner is inserted in stage 2 and partitioning is performed at the module level which has a coarser level of granularity rather than at the cell/netlist level. VHDL processes and subprograms (functions and procedures) are treated as inseparable units during partitioning. Firstly, the number of components in the graph being partitioned is not too large when compared to the cell/netlist level (stage 4 – linker and netlist generation) partitioning. Unlike partitioning with a finer granularity where control lines could be running across partition boundaries from one target device to another via the board interconnections, control lines of the control path are kept in the same partition as the data path that it is controlling when partitioning at the module level since the individual modules has its own control path controlling the data path units within the module. Having the control path in its local clock domain reduces the number of cross-domain control signals and latency due to cross-domain data synchronisation. Partitioning in Stage 2 (MOODS optimisation stage) also allows further optimisation (i.e. an optimisation re-run) on the whole design or selected modules after analysing the partitioning configuration.

The K-way partitioner performs partitioning on the optimised ICODE modules and the subprogram communication channel optimisation if the design contains ICODE subprogram modules. The two-phase partitioning exploration is currently not integrated with the MOODS optimisation process but it does allow the user to re-run the MOODS optimisation stage after examining the partitioned design. It is possible to relax or tighten the schedule of the modules and iteratively improve the multi-FPGA solution using the current partitioning solution to guide the MOODS optimisation process. This has been left as possible future work (described in Chapter 8) due to the time restriction of this project.

Unlike the multi-FPGA high-level synthesis systems described in Section 3.3, the MOODS synthesis system does not take absolute timing (in the form of deadlines and release times that specifies some form of absolute timing on the start of operations) into consideration during the optimisation process. However, MOODS possesses a basic multicycling [32] capability based around the specification of a user-specified clock

period. The problem of single instructions with too large a delay is dealt with by spanning the instruction over a sufficient number of control states can be forced below a user specified clock period constraint if the clock period is specified. MOODS does not follow the strict timing of the VHDL standard, which specifies that time only passes in *wait* statements; thus parallel processes are kept in lockstep as they are all guaranteed to enter waits at the same time and implicitly synchronised at these points. However, MOODS allows processes complete independence, where synchronisation of processes is done through the use of handshaking via global signals [32, 39]. Channel-based communications [14, 110] in an abstract Communicating Sequential Process (CSP) [111, 112] manner between processes are also commonly used for process synchronisation.

4.2.2.1 Explicit communication channel (*ExC*)

An explicit channel-based approach for process synchronisation in MOODS was added by Sacker [109]. An ICODE expansion stage was added between the ICODE assembler and the MOODS synthesis core, which allows channel related ICODE instructions to be expanded and inlined by an ICODE module contained within expansion libraries. The ICODE expansion stage also generates concurrent "blackbox" components required for the explicit channel instantiation from the behavioural VHDL. This ICODE "blackbox" component contains only a VHDL entity and its behaviour is not defined. This allows the "blackbox" component to be synthesised as normal and the behaviour of the "blackbox" (in this case the explicit channel) inserted after synthesis. ICODE templates of varying channel widths (8-bits, 16-bits, 32-bits, etc) for the channel send and receive instructions and the channel body "blackbox" components are defined in ICODE expansion library files *mod_lib.xic* and *comp_lib.xic* respectively.

The final task performed by this ICODE expansion stage is the separation of ICODE segments (VHDL processes) from within the program module (recall Section 2.6.2) into separate ICODE process modules. Each individual ICODE process module has its own control path controlling the data path units within the module. This allows easy identification of concurrent process blocks in the design and more importantly it extends the number of objects for partitioning. The explicit communication channels introduce an implied pipeline structure whereby asynchronous channels connect pipelines stages in a

design. Channel handshaking ensures that the pipelines stages will work irrespective of the operation execution time of individual stages in the asynchronous pipeline [113]. The effects and the benefits of extracted process modules communicating through explicit communication channels are shown in the experimental results in Chapter 6.

4.2.2.2 Subprogram communication channel (*SpC*)

VHDL subprograms (procedures and functions) are translated into ICODE subprogram modules in MOODS. A hierarchical calling structure is used in MOODS, whereby the control path in each subprogram module starts its execution upon receiving the *activate* signal and it sends an *end* signal back to the calling (parent) module upon termination. This implicit design boundary provides good object granularity for partitioning and the hierarchical nature of the *activate-end* protocol works seamlessly with the handshaking between processes. Process modules can run independently and call subprogram modules existing in different partitions. An arbitration scheme is necessary to arbitrate calls to an ICODE subprogram module from different calling modules. Details on the modifications of the hierarchical subprogram module calling mechanism to support inter-device subprogram calls are described in Section 4.6. The asynchronous subprogram communication channel is inserted by MOODS automatically to handle the inter-device subprogram call. The underlying communication cells and the arbitration scheme to support inter-FPGA module calls are described fully in Chapter 5.

VHDL signals are declared in the VHDL architecture and they are seen as ***global*** to processes within the architecture. Whilst any number of processes may read from a VHDL signal, only one process is allowed to write to a signal as the current MOODS synthesis system does not support resolved signals [32, 41]. It is becoming common to use communication channels for multiple communication processes [110, 113] whereby inter-process data is sent in a unidirectional, point-to-point manner. A physical implementation of a simple channel is a bundle of wires; one request wire, one acknowledge wire, and one wire per data bit (recall the bundled-data approach in Section 3.5.1). Now, the explicit communication channels performs the synchronisation task of multiple communicating VHDL processes in MOODS which was previously done through explicit handshaking

global signals or semaphores [32, 42] as well as sending an updated value of the global signal (or channel data) to the VHDL process on the receiving end of the channel.

4.3 Module call graph representation

The input behavioural specification (described in VHDL) is translated into a corresponding intermediate code (ICODE), with VHDL processes and subprograms (functions and procedures) translated into ICODE modules, and modelled as a control and data path graphs within the synthesis core. Multi-FPGA partitioning assigns the ICODE modules among k target devices. This section describes the symbols and notations used in a module call graph for a better representation, where the type of node and edge in the call graph gives a clear distinction between process and subprogram modules and the type of communication channel between the modules respectively. This representation allows the modelling of subprogram calls from different modules in the design, with arbitrarily deep nesting of such calls.

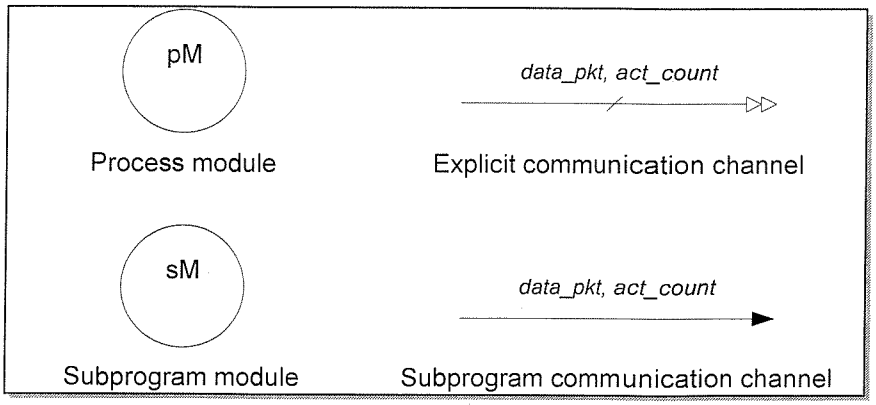


Figure 4-3 Types of nodes and edges in the module call graph

The symbol and annotation convention of the module call graph given in Figure 4-3 is used throughout the rest of the thesis unless specified otherwise. There are two types of nodes (labelled pM and sM) to represent the process module and subprogram module respectively. An explicit communication channel (see Section 4.2.2.1) is represented with an edge with two arrow heads pointing at the destination module. A subprogram call is represented by a subprogram communication channel and this is an edge with a single

filled arrow head pointing towards the called subprogram module. Both types of edges are annotated with *data packet*[1] and *activation count*[2] values.

Examples of the various basic types of connection in the module call graph are listed in Table 4-1. A node in the module call graph can have multiple edges connecting it to other nodes. The first and last types of connection in the table show two process modules (P1 and P2) connected with explicit communication channels, the second type of connection is a subprogram call (to subprogram proc1) initiated from a process module (P1), and the third type of connection is a nested subprogram call. In summary, an explicit communication channel is used for process-to-process communications and a subprogram communication channel is used to connect the destination subprogram module to a process module or a subprogram module, in the case of nested subprogram calls.

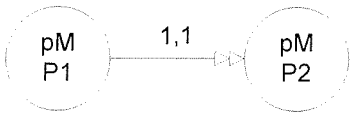
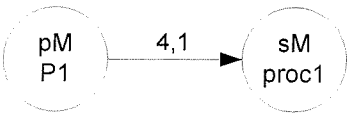
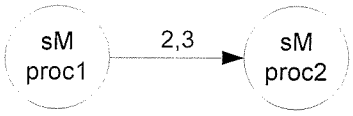
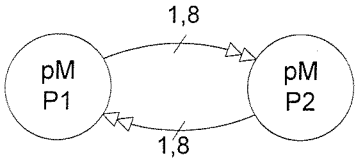
Connection type example	Description of the example
	Process module P1 sends data to process module P2 through an explicit communication channel. The channel has a single data packet count and activation count.
	A subprogram communication channel connects process module P1 to subprogram module proc 1. This subprogram call has 4 data packets and an activation count of 1.
	A subprogram communication channel connects subprogram module proc1 to subprogram module proc2. This nested subprogram call has 2 data packets and an activation count of 3.
	Process module P1 and process module P2 send and receive data via explicit communication channels. Both explicit channels have a single data packet count and an activation count of 8.

Table 4-1 Examples of types of connection in the module call graph

[1] Defined as the number of data packets transferred as parameters between a source and destination module

[2] Number of times the source module activates the destination module

4.4 Problem formulation

The module call graph is a weighted directed graph $CG = (N, E)$. Each node $n_i \in N$ represents a module in the design, the area of each module is denoted as $a(n_i)$, and the I/O pin count is denoted as $io(n_i)$, for $i \rightarrow 1$ to n_{total}, where n_{total} is the total number of modules in each partition. Each edge $e_i \in E$, $e_i = (nsrc, ndst, data\_pkt, act\_count)$, for $i \rightarrow 1$ to e_{total}, $nsrc \in N$, $ndst \in N$, $nsrc \neq ndst$ corresponds to either an explicit channel or subprogram communication channel from the source module *nsrc* to the destination module *ndst*. The data packet count *data_pkt* is the number of data packets transferred as parameters between *nsrc* and *ndst* during each call. The activation count *act_count* is the number of times *nsrc* calls *ndst* and this activation count value is obtain from the design activity profile.

A set of available m target devices is given by $D = \{d_1, d_2, \ldots, d_m\}$ where $m \geq k \geq 2$. Each device $d_i = (d\_area_i, d\_io_i)$ where d_area_i and d_io_i denote the area capacity and number of available I/O pins of device i.

The K-way partitioning problem finds a set of clusters $P = \{ p_1, p_2, \ldots, p_k \}$ such that $p_i \subseteq N$ for $i \rightarrow 1$ to k, $\bigcup_{i=1}^{k} p_i = N$ and $p_i \cap p_j = \varnothing$ for $i \rightarrow 1$ to $k, j \rightarrow 1$ to k, and $i \neq j$. The partitioning solution must satisfy a set of device constraints (area and I/O) and minimise the inter-partition data transfers.

The area constraint for this K-way partitioning problem is given by:

$$d\_area_k \geq \sum_{\forall n_i \in p_k} a(n_i) \qquad \text{for } k \text{ partitions where } i \rightarrow 1 \text{ to } n_{total}, \; n \in N, \text{ and } p_k \in P. \tag{4.1}$$

Let the cut-size c_{kj} be the number of interconnects crossing the partition boundary between partitions p_k and p_j. The I/O constraint is given by:

$$d\_io_k \geq \left(\sum_{\forall n_i \in p_k} io(n_i) \right) + c_{kj} \qquad \text{for } k \text{ partitions where } i \rightarrow 1 \text{ to } n_{total}, \; n \in N, p_k \in P, j \rightarrow 1 \text{ to } k, \text{ and } j \neq k. \tag{4.2}$$

4.4.1 Modified K-way partitioning in MOODS

The partitioning process of a single design onto multiple FPGAs is done in two phases. The first phase performs K-way partitioning on the modules in the design. The partitioning algorithm is outlined in Figure 4-4. The second phase deals with the assignment and optimisation of inter-FPGA subprogram communication channels to the partitions. Each subprogram communication channel is managed by communication (transmit and receive) cells and an arbiter cell. Figure 4-6 outlines the second optimisation algorithm that creates and optimises the subprogram communication channel(s) between target devices. More than one subprogram communication channel can be created and assigned to two or more modules in the design.

4.4.1.1 K-way partitioning algorithm

The inputs to the K-way partitioning algorithm include the module call graph of the design and the area constraint of the target devices. The algorithm starts with an initialisation stage where the input module call graph *CG* is checked to ensure that it is properly annotated with valid parameters, and all constraints such as number of target devices are set. An initial partition is generated and this forms the starting partition of the first pass.

The K-way partitioning algorithm is similar to the two-way FM algorithm (described in Section 3.2.1) with a few slight changes, such as the select-and-move process, and the balanced criteria. Unlike the two-way FM algorithm that only considers whether to move a node to the next partition (i.e. move the base node from partition *A* to *B*, or from partition *B* to *A*), the K-way algorithm considers *K*-1 possible partitions to move the base node and the *Gain_Array* that holds an array (*K*-1 in size) of gain values associated with moving a node from the current partition to another partition.

A selected base node (n_{base}) move from partition p_x to partition p_y is only allowed when it satisfies the balanced criterion given by:

$$\left(\left(\sum_{\forall n_i \in p_y} a(n_i)\right) + a(n_{base}) \le \frac{a(N)}{k} + w_{\min}\right) \wedge \left(\left(\sum_{\forall n_i \in p_x} a(n_i)\right) - a(n_{base}) \ge \frac{a(N)}{k} - w_{\min}\right) \quad (4.3)$$

for $i \rightarrow 1$ to n, $n \in N$, $p_x \in P$, $p_y \in P$, $x \neq y$, and w_{min} is the area of the smallest unlocked node (i.e. $w_{min} = \min_{1 \leq i \leq n} (a(n_i)_{\text{unlocked}})$).

K-way Partitioning Algorithm

CG: module call graph $CG = (N, E)$, N is a set of nodes and E is a set of edges
DevArea[]: Device area of each target device (FPGA)

```
KWay (CG, DevArea[])
begin
    Initialise K-way partitioning parameters;
    CurrentPartition ← Generate a legal initial partition;
    BestPartition ← CurrentPartition;
    BestCutcost ← CurrentCutcost;
    improved_cutcost ← True;
    /* ----- PASS MANAGER ----- */
    while ( improved_cutcost ) { /* run until no further improvement in the cutcost */
        /* ----- MOVE MANAGER ----- */
        step_number ← 0;
        /* True only when balance condition and device area constraints are satisfied */
        while ( kway_move_vertex(Gain_Array, CurrentPartition, DevArea[]) ) {
            step_number++;
            Update K-way Gain_Array, and CurrentCutcost;
            Update tentative_cutcost[], tentative_moves[], tentative_moved_to[];
            Update size of partition and lock moved node;
            if (tentative_cutcost[best_tentative_move] ≥ CurrentCutcost ) then
                best_tentative_move ← step_number;
            end if
        } end while
        for (i =1; i ≤ best_tentative_move; i++)
            Permanently move nodes in tentative_moves[i]to partition specified in tentative_moved_to[i]
        end for

        improved_cutcost ← False;
        if ( CurrentCutcost < BestCutcost ) then
            CurrentPartition ← BestPartition;
            CurrentCutcost ← tentative_cutcost[best_tentative_move];
            Improved_cutcost ← True;
        end if
    } end while
    return ( CurrentPartition ) /* Final partition */
end
```

Figure 4-4 Outline of the K-way partitioning algorithm

Similar to the two-way FM algorithm, the move with the highest gain is selected and executed iteratively until all free nodes are locked. The K-way algorithm continues with the execution of this iterative select-and-move sequence until no more unlocked nodes can be moved without violating the balanced criterion. At the end of a pass, the K-way algorithm moves back to the best intermediate solution. All nodes are unlocked and the best solution forms the starting partition for the next pass. The algorithm terminates when a pass fails to improve the cutcost. The cutcost is the total number of inter-FPGA data packets between all partitions and it is given by

$$\sum_{\forall e_i} e_i(data\_pkt) \cdot e_i(act\_count) \tag{4.4}$$

for $i \rightarrow 1$ to e_{total}, $e \in E$, $p_{e(nsrc)} \in P$, $p_{e(ndst)} \in P$, *and* $p_{e(nsrc)} \neq p_{e(ndst)}$.

4.4.1.2 Subprogram communication channel optimisation algorithm

The subprogram communication channel optimisation algorithm creates a subprogram communication channel or multiple channels, optimised to reduce the inter-FPGA data packets sent between partitions. Using the design activity profile to determine module calls that cause congestion in the communication channel, provided that the target device area and I/O constraints between these module calls are met, the algorithm creates and assigns the modules responsible for this bottleneck in data transfer to a new subprogram communication channel. The design activity profile is a temporal analysis of the module activation in the system over a series of time steps.

The algorithm uses a greedy-based strategy [114, 115] to reduce the bottleneck through the primary subprogram communication channel. A simple example of the greedy-based strategy in Figure 4-5 shows subprogram calls *A* to *D* and the block height of each call gives the number of data packets sent in each call (e.g. call *A* sends one data packet and call *B* sends three data packets).

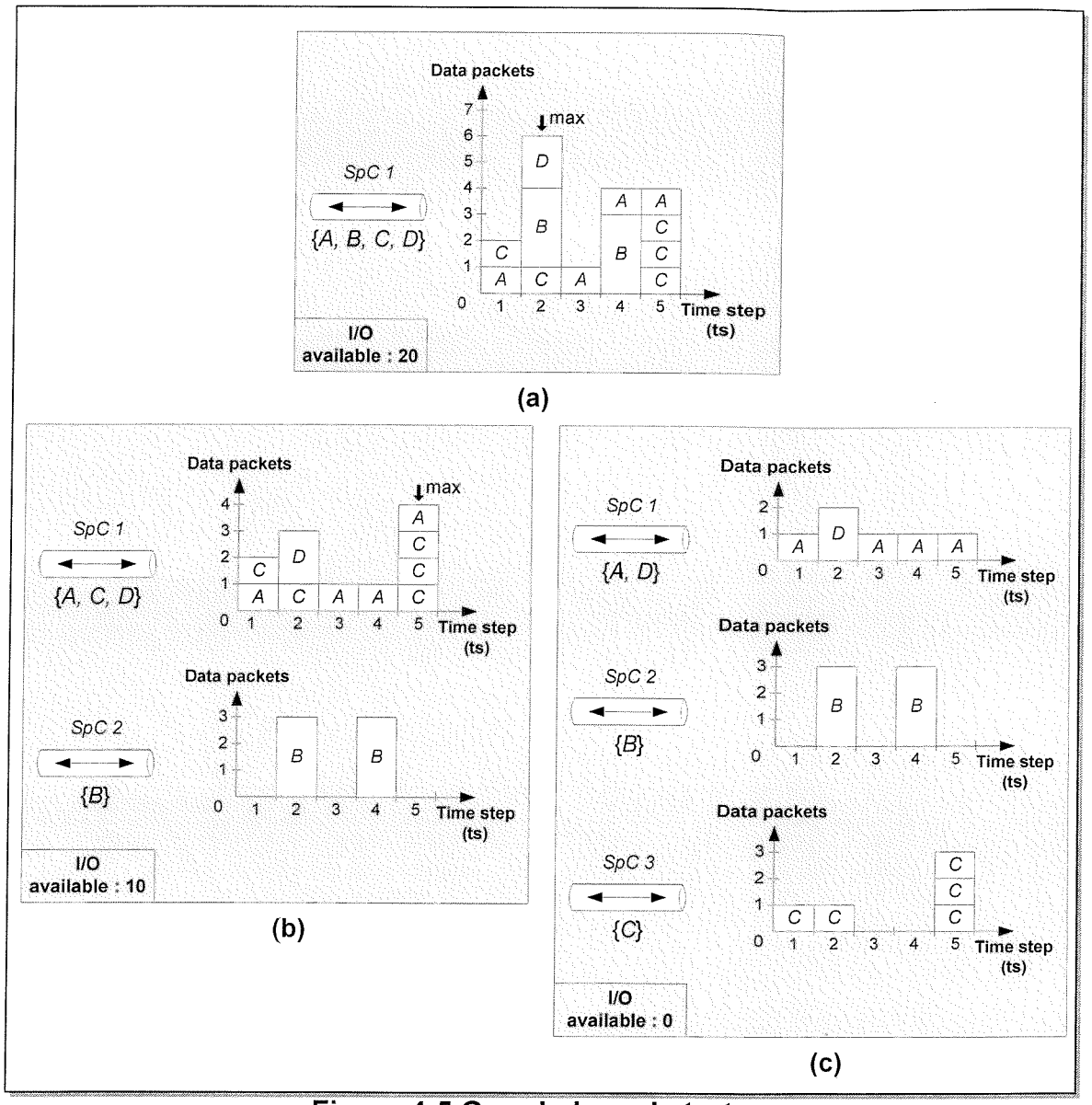


Figure 4-5 Greedy-based strategy

Figure 4-5(a) shows all the four subprogram calls assigned to *SpC 1* and 20 I/O pins available. Assume each *SpC* uses 10 I/O pins and the subprogram calls are mutually exclusive. The time-step with the maximum number of data packets is identified (in this case time step 2) and the subprogram call with the largest number of data packets is call *B*. Subprogram call *B* is extracted and allocated to a new subprogram communication channel (*SpC 2*) as shown in Figure 4-5(b). The number of I/O pins available reduces to 10 with inclusion of *SpC 2*. Now, the maximum number of data packets sent in any one time step reduces to 4 in time step 5, with subprogram call *C* called 3 times and contributing the

most number of data packets in time step 5. Subprogram call C is extracted and allocated to a new subprogram communication channel (*SpC 3*) shown in Figure 4-5(c). The algorithm terminates as no more I/O pins are available. Assume each data packet takes one unit delay. The original allocation of all four subprograms allocated to a single communication channel in (a) will take 6 unit delays. With additional I/O pins and the allocation of subprogram calls to more subprogram communication channels (3 channels in this example), the overall delay is reduced to 3 unit delays.

The subprogram communication channel optimisation algorithm (outlined in Figure 4-6) begins with all module call pairs assigned to the primary subprogram communication channel. If the option for multiple subprogram communication channels is not selected (i.e. *Multiple_Comm_Channel = false*), the algorithm terminates with all modules transferring inter-FPGA data using the primary communication channel. If enabled, the algorithm proceeds by first unlocking all module pairs. The following three steps are executed iteratively in sequence till no further improvements (*end_of_Opt = true*) can be made:

- **Step 1** - The inter-FPGA data transfers for all unlocked module call pairs are calculated (see Section 4.5).

- **Step 2** - The time steps are sorted according to the number of inter-FPGA data transfers and the unlocked module call pairs in each time step are sorted according to their inter-FPGA transfers in the temporal time step. The subprogram communication channel optimisation algorithm terminates when none of the target device area and I/O constraints for the module pairs is met.

- **Step 3** - The channel insertion routine *insert_comm_channel* is called and a new subprogram communication channel is inserted when there is an improvement (reduction) in the total number of inter-FPGA packets. The channel insertion routine returns a 0 when no improvement can be made, or when there is only a single module pair assigned to the subprogram communication channel. The area and I/O resources of the target devices containing the module pairs are updated if the channel insertion

routine returns a non-zero return value; else a zero return value terminates the subprogram communication channel optimisation algorithm.

Subprogram Communication Channel Optimisation Algorithm

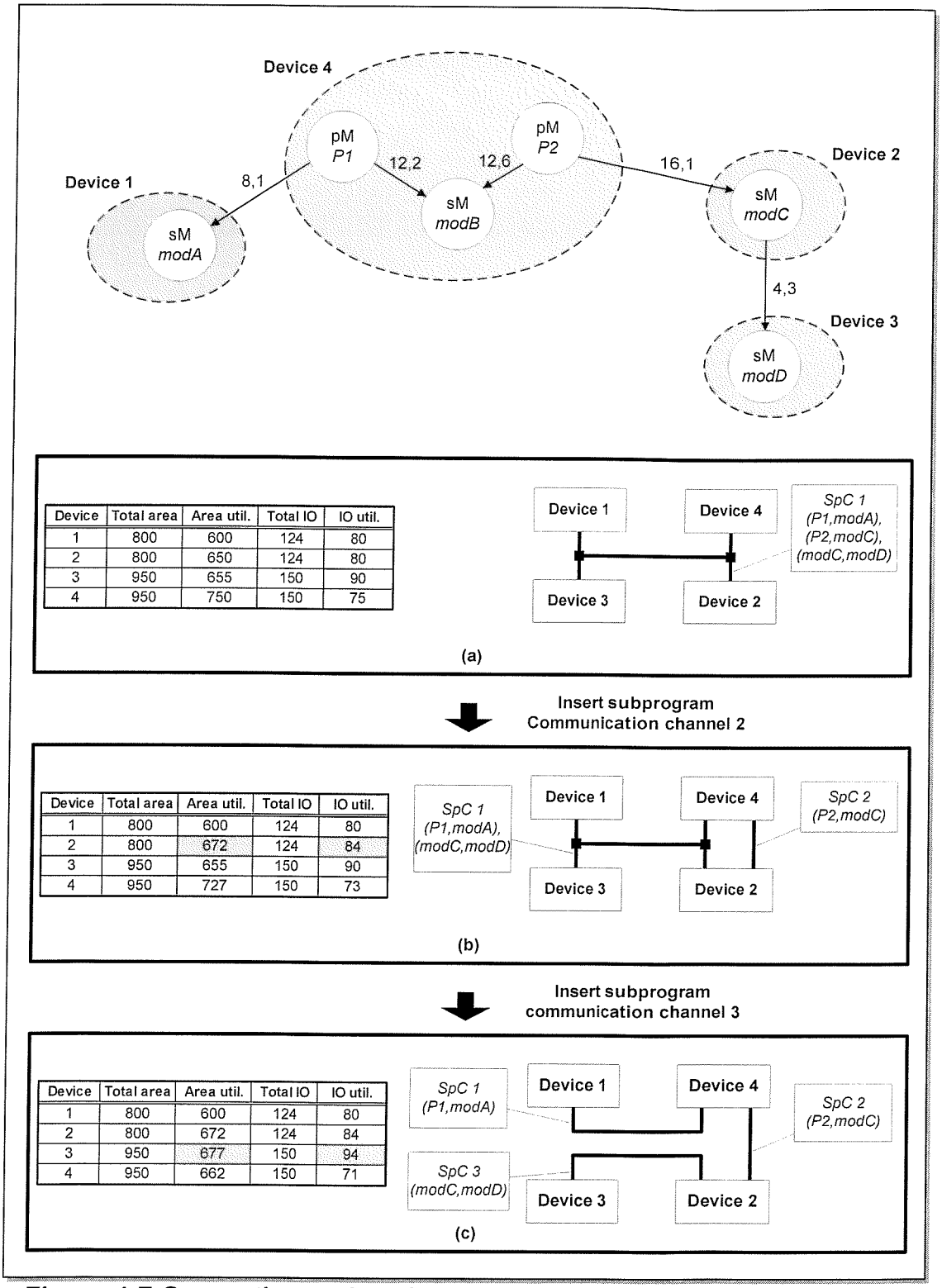
profile_info[]: profile information array containing information on the module call graph

```
Optimise_Comm_Channel (profile_info[])
begin
    Primary comm_channel ← all modules pairs;
    if ( Multiple_Comm_Channel ) then {
    Unlock all module call pairs (src,dst)
    end_of_Opt ← False;
    while ( end_of_Opt = False ) {/* run until no further improvement can be made */
        /* ----- Step 1 ----- */
        profile_info_sorted[] ←   Generate a sorted array of profile_info[], the generated array is sorted
                                  according to the inter-FPGA data transfers.
        Calculate inter-FPGA data transfers for all unlocked module pairs in each time step.

        /* ----- Step 2 ----- */
        Sort time steps in order according to 'traffic congestion', with unlocked module pairs in each time
        step sorted in order according to their inter-FPGA data transfers in the time step.

        if ( Area and I/O constraints not met ) then
            end_of_Opt ← True;
        else
            end_of_Opt ← False;
        end if

        /* ----- Step 3 ----- */
        bus_opt_status ← insert_comm_channel()
        if ( bus_opt_status ) then
            end_of_Opt ← False;
            Update Area and I/O resources of the device that the newly created bus arbiter is assigned to
        else
            end_of_Opt ← True;
        end if
    } end while
    end if
end
```

Figure 4-6 Outline of the subprogram communication channel optimisation algorithm

Figure 4-7 illustrates the generation and assignment of multiple subprogram communication channels (*SpC*s) to alleviate the delay due to devices sharing a common communication channel. Each row in the table shown in the figure gives the total area (in slices), the area utilised, total number of available I/Os, and the number of I/Os utilised by the devices in the multi-FPGA system. Each communication channel consists of transmit cell(s), receive cell(s), and a channel arbiter to ensure mutually exclusive access to the shared channel between the devices connected to it. The original partitioned design starts with a single communication channel (see Figure 4-7(a)), which connects up all the target devices and inter-FPGA data transfers are made through a single bi-directional communication channel. The arbiter for *SpC 1* is found in Device 4. With extra area and I/Os available in (a), the subprogram communication channel optimisation algorithm inserts a new communication channel, module call pairs with a high amount of *traffic* that cause congestion in the first communication channel are determined and assigned to a new communication channel. Further details on design activity profiling to determine the amount of inter-FPGA data transfers and how this affects the partitioning algorithm is covered in the next section.

In the module call graph given at the top of Figure 4-7, module call pair (*P2, modC*) is extracted from *SpC 1* and assigned to the newly created communication channel (*SpC 2*) to spread the inter-FPGA data transfers over two channels, and thus inferring a level of parallelism in inter-FPGA data transfers since the two communication channels can transfer data concurrently. Device area and I/O utilisation are traded off for the increase in parallelism. The area and I/O increase of Device 2 is shaded in the table in Figure 4-7(b); this increase is due to the insertion of a communication channel arbiter for *SpC 2*. The area utilisation of Device 4 is reduced to 727 units (as the arbiter for *SpC 1* described above is smaller) and its I/O utilisation reduced by two pins as a new arbiter (with an area of 22 units) generated in Device 2 handles the arbitration control of module call pair (*P2, modC*). In Figure 4-7(c), module call pair (*modC, modD*) is extracted from *SpC 1* and assigned to *SpC 3*. The communication channel arbiter for *SpC 3* is inserted into Device 3, thus increasing the area and I/O utilisation for Device 3. This again reduces the area and I/O utilisation of Device 4, and now the task of inter-device data transfers is distributed between three subprogram communication channels.



Device	Total area	Area util.	Total IO	IO util.
1	800	600	124	80
2	800	650	124	80
3	950	655	150	90
4	950	750	150	75

Device	Total area	Area util.	Total IO	IO util.
1	800	600	124	80
2	800	672	124	84
3	950	655	150	90
4	950	727	150	73

Device	Total area	Area util.	Total IO	IO util.
1	800	600	124	80
2	800	672	124	84
3	950	677	150	94
4	950	662	150	71

Figure 4-7 Generation and assignment of communication subsystems

4.5 Integration of the design activity profile and the K-way partitioning algorithm

The generation of the design profile and its integration into the K-way partitioning algorithm to guide the partitioner is covered within this section. This section starts with a look at how the data widths of input and output parameters affect partitioning. A subprogram module with larger input or output parameters data width tends to require more data packets than a sub-module with a smaller input and output parameter data width if the target devices are I/O limited. This is illustrated in Figure 4-8 with process module *P1* calling two subprogram modules, *proc1* and *proc2*.

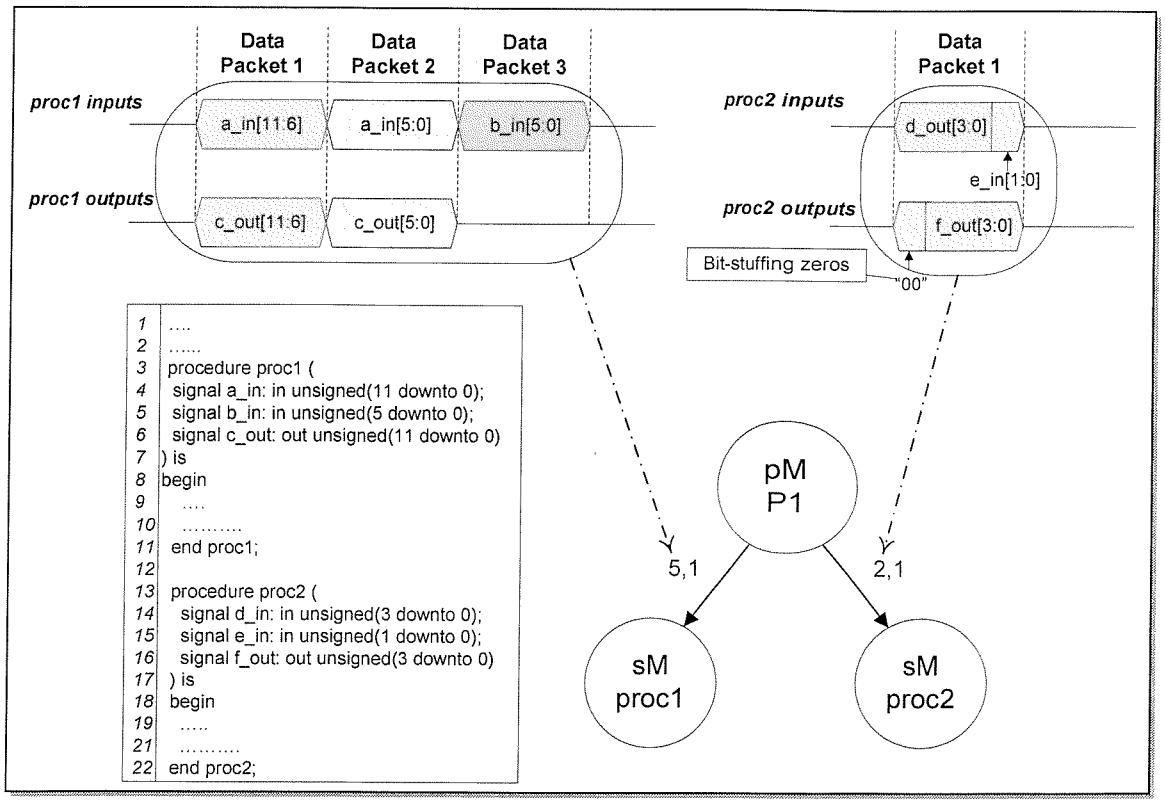


Figure 4-8 Example of I/O parameter sizes and data packet count

The example shows the number of data packets sent by each subprogram module using a common subprogram communication channel with a 6-bit channel width. Input parameter a_in of subprogram *proc1* is sent over the channel in two separate data packets and input b_in in a third data packet. Output parameter (result) c_out of subprogram *proc1* is sent over the channel in two separate data packets. Input parameters d_in and e_in of

subprogram proc2 are concatenated and sent in a single packet. Output parameter f_out of proc2 is bit-stuffed with zeros and sent in a single data packet. The subprogram communication channels connecting process module P1 to subprogram modules proc1 and proc2 have total data packet counts of 5 and 2 respectively (as shown in Figure 4-8).

The partitioning algorithm priorities the partitioning of subprogram modules based on the number of data packets sent, thus a subprogram module with a larger parameter bit-width is less likely to be partitioned onto a second FPGA compared to a subprogram module with a smaller parameter bit-width if both are being called by the same source module (i.e. assuming both subprogram modules are activated the same number of times) as a larger parameter bit-width will probably require more data packets when targeting I/O limited devices.

With the inclusion of the design activity profile, the subprogram module activation can be modelled more accurately and the profile data is used to guide the partitioner in producing a partitioned design with less inter-FPGA communication. The temporal 'traffic analysis' is extracted from the simulation of the design using a typical (or likely) set of values emulating a working system. The source-destination module pair has a *call-node* associated with each subprogram call. A module call list (*.mcl*) file is automatically generated in MOODS during synthesis. Definition of the module call list can be found in Appendix C.3. This module call list file lists all source-destination pairs and the call node that is activated for each subprogram module call. An example of a module call list file and a simulation of the activation of modules in a module call graph with two process modules (P1 and P2) and four subprogram modules (modA, modB, modC and modD) is shown in Figure 4-9.

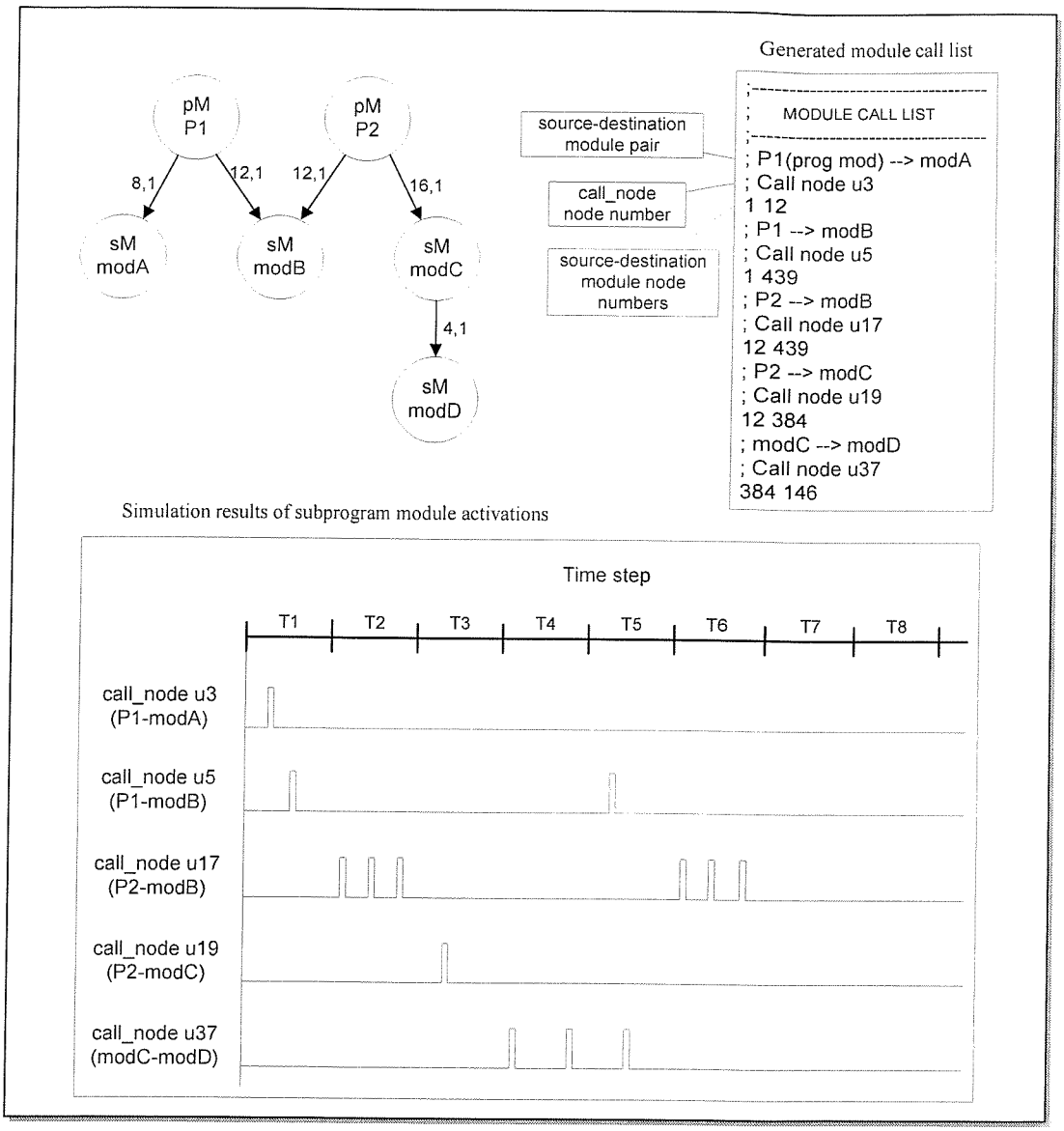


Figure 4-9 Example of module call list and simulation of subprogram module activations

A profile of activation counts of the call nodes is extracted from the simulation, and this design profiling data is fed into the partitioner using the partitioning information (*.par*) file. The profile data is modelled using a distribution graph, where the vertical axis corresponds to the summation of all module activation counts in a particular time step on the horizontal axis. An example of the distribution graph generated for the example above is illustrated in Figure 4-10.

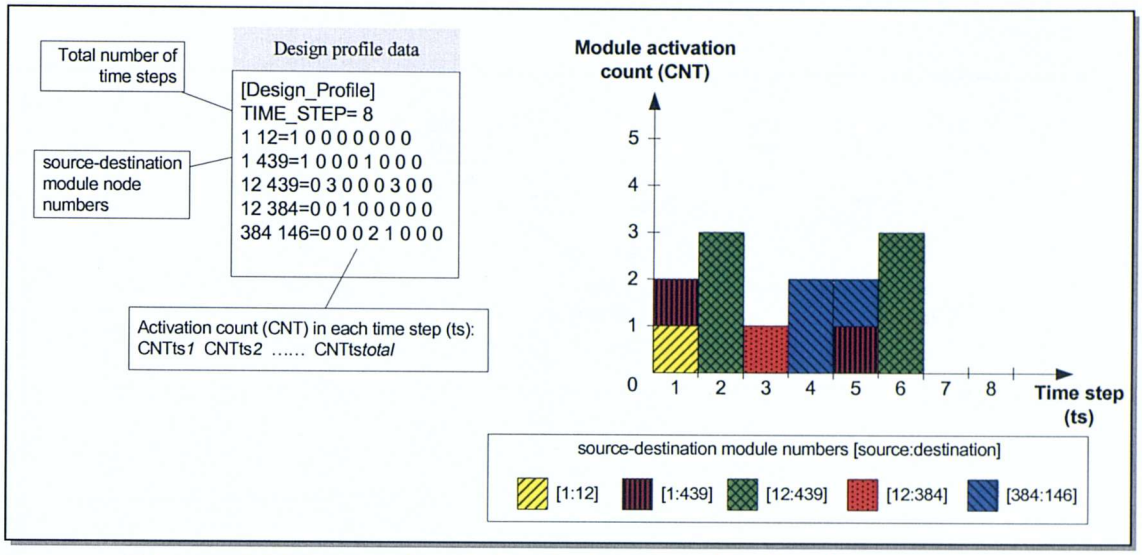
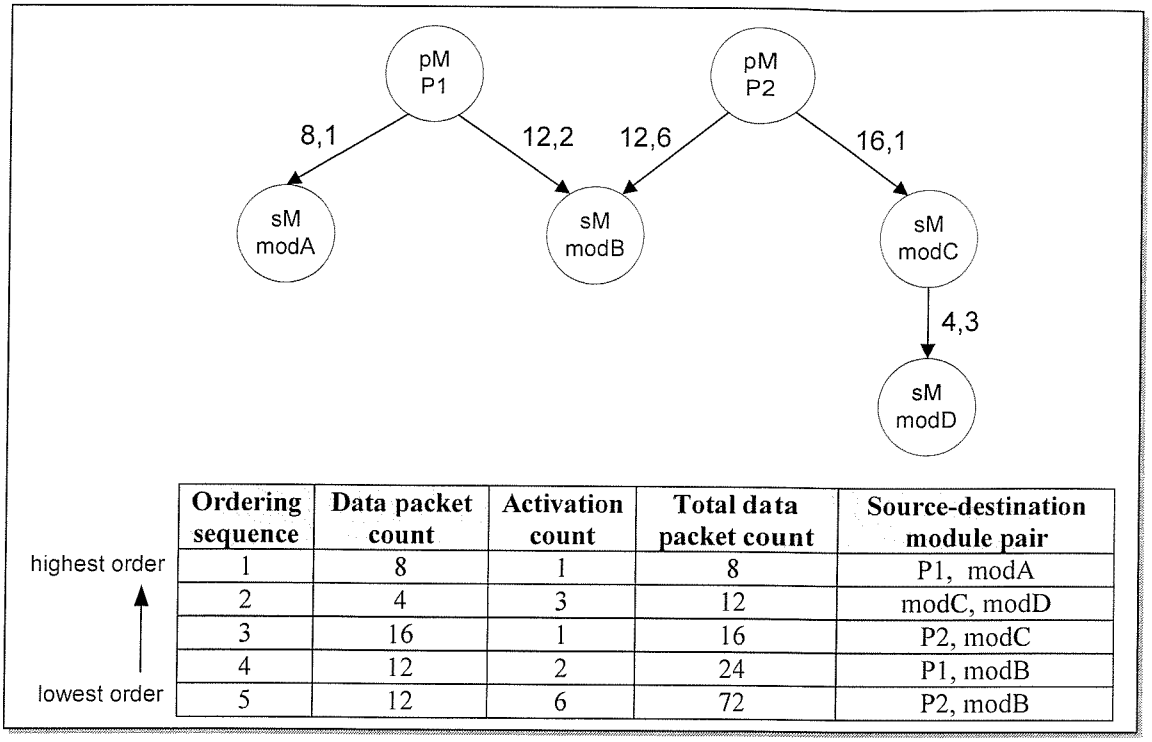


Figure 4-10 Example of the design profile distribution graph

A partitioning ordering sequence gives the likelihood of a source and destination module being partitioned onto separate FPGA devices. The data packet multiplied by the activation count is given in the total data packets column. The total number of data packets for the source, destination pair has an inverse relationship with the likelihood of the pair being partitioned onto separate FPGA devices. In other words, the greater the total data packet count, the more likely the pair will be partitioned onto the same device. Source-destination module pairs with a lower ordering sequence are less likely to be partitioned onto separate devices. Figure 4-11 gives the partitioning ordering sequence of the call graph example in Figure 4-9.

The total data packet count of the source-destination module pairs is now not only dependent on the I/O parameters data width but also the number of times the source module calls the sub-module. For example, the (*P2, modB*) pair has an activation count of 6 and a total data packet count of 72, and it has the largest total data packet count compared to the other module pairs. Modules *P2* and *modB* are most likely to be partitioned onto the same FPGA device, whereas modules *P1* and *modA* have the highest chance of being partitioned onto separate FPGAs because the (*P1, modA*) pair has the highest sequence order of 1. The activation count and the data packet count for the module

call pairs are fed into the K-way partitioner described in Section 4.4.1 using the partitioning information (.par) file.



highest order ↑ lowest order

Ordering sequence	Data packet count	Activation count	Total data packet count	Source-destination module pair
1	8	1	8	P1, modA
2	4	3	12	modC, modD
3	16	1	16	P2, modC
4	12	2	24	P1, modB
5	12	6	72	P2, modB

Figure 4-11 Partitioning ordering sequence with design profiling

4.6 ICODE Module modifications

Prior to the partitioning enhancement, the output values for subprogram modules are passed by reference. Now, modules are targeted onto two or more target devices, the output results are passed by value between the source module and the subprogram module called. Registers are required to hold the output parameters prior to sending the data back to the calling (source) module. The removal of output registers for inter-device subprogram modules are bypassed. Details of the modifications in the subprogram module call mechanism are covered within this section.

After the partitioning stage, 'call nodes' (*call_nodes*) associated to ICODE *MODULEAP* instructions for modules calling a subprogram module in a different partition are replaced with 'transmit call nodes' (*tcall_nodes*). The *tcall_nodes* are associated with ICODE

TX_CELL instructions, which replace the original ICODE *MODULEAP* instructions. This change allows MOODS to distinguish between the two types of calling methods. The instruction associated with ICODE subprogram modules is the ICODE *MODULE* instruction. Likewise, subprogram module that is called by modules in another partition has its module header instruction changed from the *MODULE* instruction into a new instruction defined for subprogram modules with inter-FPGA calls, ICODE *RXCELL* instruction. This change allows MOODS to determine which subprogram modules are called from modules in another FPGA device. The top of Figure 4-12 shows the original call node associated with an ICODE *MODULEAP* instruction, the *call_node* activates the start node in the subprogram module when it is being activated (i.e. when the main execution is paused and control is passed to the subprogram module controller). This hierarchical method of control passing and data passing is modified when the subprogram module is located in a separate FPGA device as illustrated in Figure 4-12.

In the source (calling module) partition, the original *call_node* is replaced by the *tcall_node* and the *tcall_node* now activates a 'transmit cell' (*txcell_node*) when inter-FPGA communications is required. The *txcell_node* is the communication cell that sends input parameters across the FPGA device and receives the results when the execution of the operation is complete at the destination subprogram module. Upon completion of the subprogram module execution, the *txcell_node* receives and loads the output results into the appropriate output result registers. Control is passed back to the main execution and this completes the subprogram module call.

In the called subprogram module, a 'receive cell' (*rxcell_node*) receives the input parameters sent by the *txcell_node* of the calling module. A 'receive call node' (*rcall_node*) is activated by the *rxcell_node* when the input parameters are received and loaded into the appropriate registers prior to the execution of the subprogram module. The *rcall_node* uses the same calling mechanism of a *call_node*, it activates the start node in the subprogram module. Upon completion of the subprogram module, results in the output registers are ready to be sent back to the source module in the other partition. The *rcall_node* activates the *rxcell_node*, which completes the subprogram module call when it sends the results to the corresponding *txcell_node* of the called module.

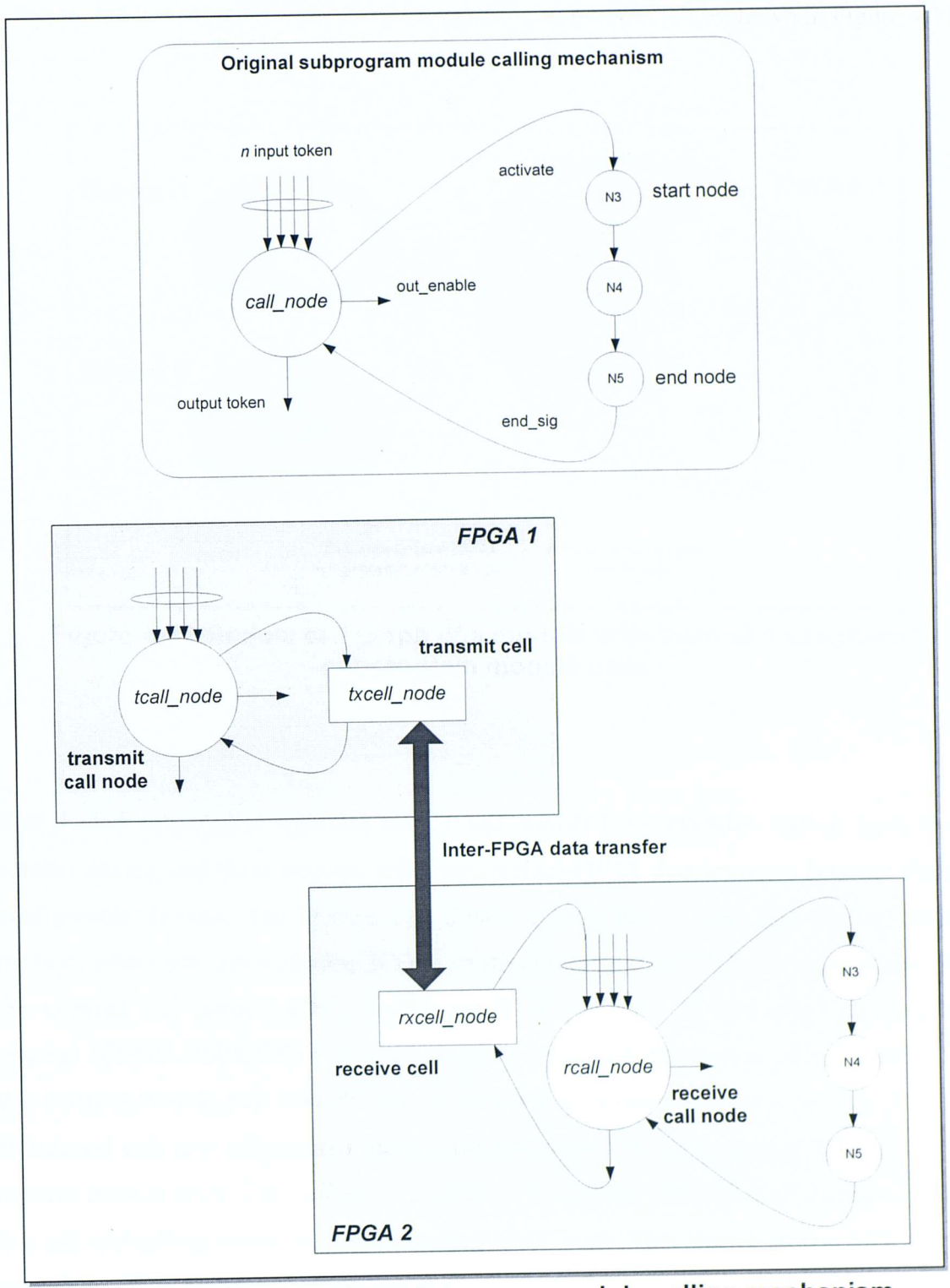


Figure 4-12 Inter-FPGA subprogram module calling mechanism

In a partitioned design, not all module calls require the inter-FPGA data transfer using the communication subsystem. Some of the module may only call other modules within the same device and the original module call method is used in such cases. An example of a module that has an internal and external subprogram module call is shown in Figure 4-13.

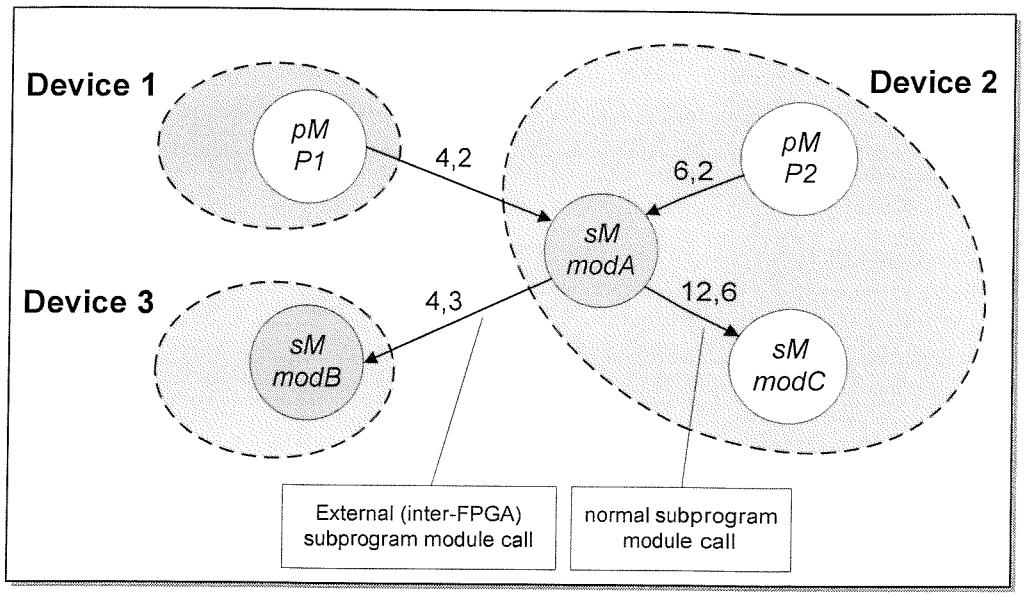


Figure 4-13 Module call graph of a module with internal and external subprogram module calls

The shaded subprogram modules (*modA* and *modB*) have modules calling them from another device, and these external calls require inter-FPGA data transfers between the re-configurable devices. The internal call (*modA* calling *modC*) uses the original calling method, where the corresponding ICODE instruction is the *MODULEAP* instruction. For the external call (*modA* calling *modC*), an ICODE *TX_CELL* instruction replaces the original ICODE *MODULEAP* instruction, and a set of communication cells is created for this external module call. Note the special case where an internal process module (*P2*) has an internal call to a subprogram module (*modA*), which is also activated by an external process module (*P1*). The *call_node* associated to ICODE *MODULEAP* instructions for this call (*P2* calling *modA*) is replaced with a *dcall_node*. This change allows MOODS to identify modules activated by both internal and external subprogram calling methods.

Details of the *txcell_node*, *rxcell_node*, and the modifications of I/O parameter registers are covered in greater detail in Chapter 5. Note the underlying structure of the final generated hardware uses pass-by-value instead of pass-by-reference for the subprogram I/O parameters as a local copy of the subprogram I/O parameters is needed in the target device of the external subprogram. Data packets which contain subprogram I/O parameters are sent to the external module and kept in local (duplicated) registers. Registers are required to hold the output parameters prior to sending the data back to the calling (source) module.

4.7 Summary

This chapter starts with a discussion on the implementation of the partitioning mechanism into the MOODS synthesis system, and the effect of performing partitioning for a multi-FPGA system at the various subcomponent stages within the MOODS synthesis system. A stage to insert the multi-FPGA partitioning mechanism and the level of granularity to perform the partitioning is selected considering the various factors that would affect the performance of the synthesis tool as well as that of the generated multi-FPGA system. Two types of communication channels are presented in this chapter; an explicit communication channel used for data transfers between ICODE processes (from VHDL processes) and a subprogram communication channel used for inter-device ICODE subprogram module (from VHDL procedures and functions) calls.

A formulation of the multi-FPGA partitioning problem is presented, a K-way partitioning algorithm and a subprogram communication channel optimisation algorithm are proposed as a two-phase solution. A module call graph representation used to model the data structures for partitioning is also presented within this chapter. The generation of the design profile and how this profile information is used to guide the partitioning algorithm is also covered within Section 4.5. The design profile and target technology information (number of target devices, area and I/O constraints) are passed into the synthesis system using a partitioning information (*.par*) file. Refer to Appendix C for the full detail of the partitioning information file. The new added features to synthesise a multi-FPGA system using the MOODS synthesis system are shaded in Figure 4-14.

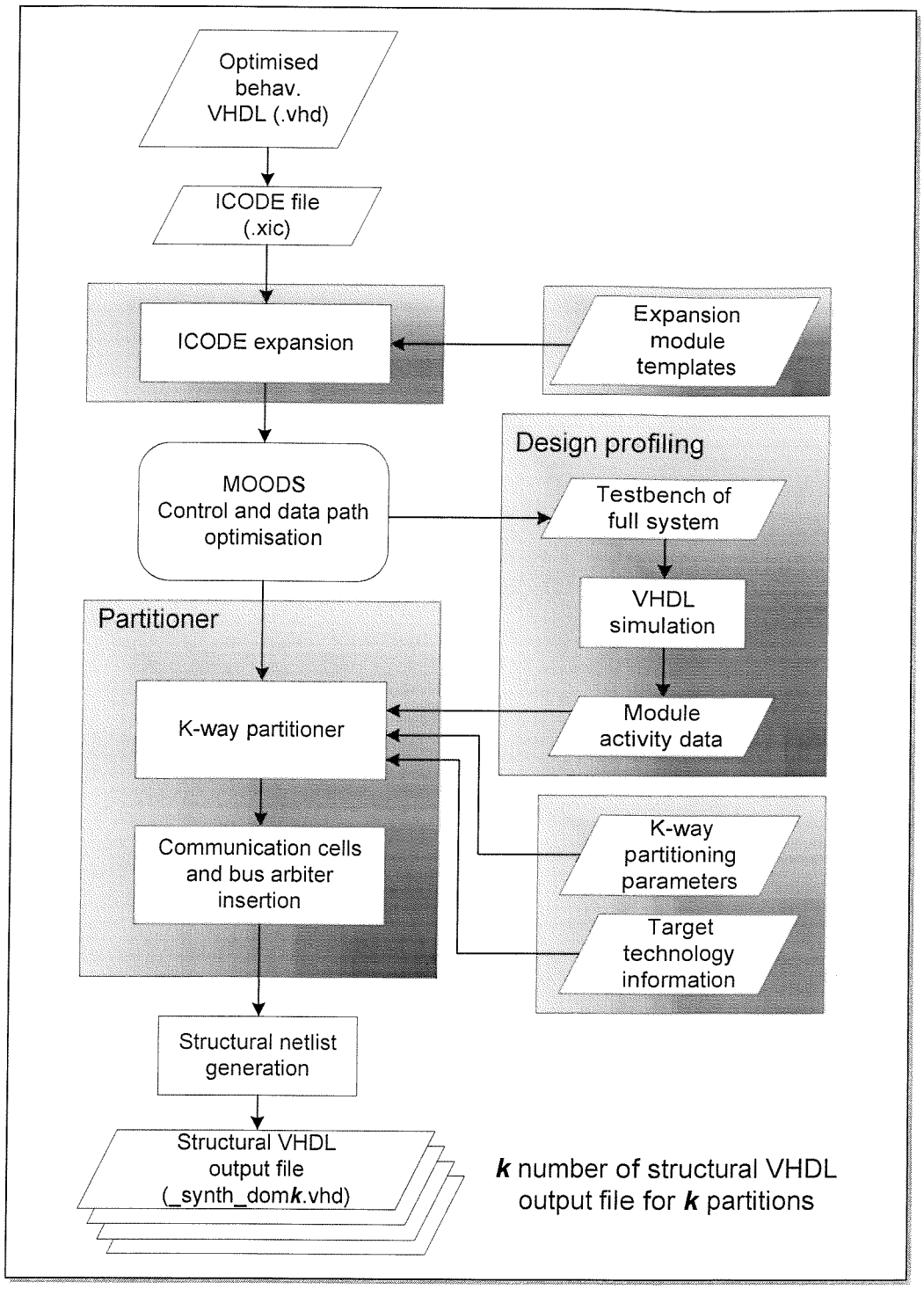


Figure 4-14 Modified MOODS synthesis system with multi-FPGA partitioning

The next chapter covers the implementation of the asynchronous communication channels in greater detail. Details covered within the next chapter include the asynchronous data communication channel used for inter-FPGA data transfers, the various communication

cells and arbiter cells which form the building blocks of the subprogram communication channel, and the structure of the final generated hardware synthesised design targeted onto multiple heterogeneous target devices. The ICODE list of instructions and the MOODS cell libraries are modified for the support of the multi-FPGA partitioning. The ICODE instruction database is extended to include the ICODE instructions associated with inter-FPGA data transfers (e.g. instructions such as *TX_CELL, RX_CELL*). Refer to Appendix C for the full ICODE instruction database description. The MOODS cell library database file (*.mlib*) is extended to include the communication cells, and latches to implement the subprogram communication channel. The parameterised structural/RTL components of these cells are added to the existing *MOODS_LIB2.6* (*.vhd*) file, and a new updated library file, *MOODS_LIB2.7* (*.vhd*) file, was created.

Chapter 5

Communication channels

5.1 Introduction

Once a single behavioural description has been synthesised and partitioned in MOODS, the next step is to look at the interface generation (shaded region in Figure 5-1) so that one FPGA device can communicate with other FPGA devices in the multi-FPGA system.

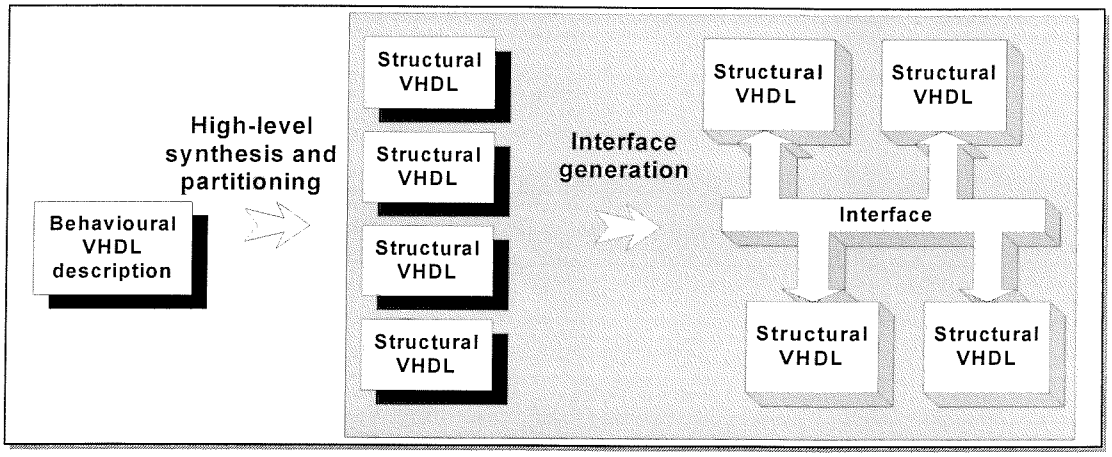


Figure 5-1 Generated system structure

The synthesis of a multi-FPGA system with heterogeneous devices with a single global clock becomes impractical, as the clock rate of the whole system is dependent on the slowest device connected. As the number of FPGA devices increases, the limiting problem becomes the distribution of the single clock without introducing intolerable clock skew. One approach to alleviate the above given problems is to synthesise a multi-FPGA system using a locally clocked, globally delay-insensitive approach [103, 116]. The partitioned design is targeted onto separate FPGA devices, where each device is clocked locally and the board-level devices communicate with one another using delay-insensitive signalling

methods. Asynchronous communication channels are used for data transfers between the partitions. The communication channel interface is presented in Section 5.2. Section 5.3 details the communication protocol of design targeted onto an arbitrary number of FPGA devices. Section 5.4 deals with the implementation details of the communication cells and arbiter cells, which are the building blocks of the subprogram communication channel. Section 5.5 deals with the hardware generation of the underlying structure to support data communications between the devices in the multi-FPGA system.

5.2 Communication channel interface

The ICODE expansion stage expands channel-related instructions and replaces the instructions with the corresponding expanded ICODE template by inlining. A simpler template shown in Figure 5-2 with a variable channel data-width defined by the channel data sent using the channel replaces the original ICODE templates of varying channel widths (8-bits, 16-bits, 32-bits, etc) in [109].

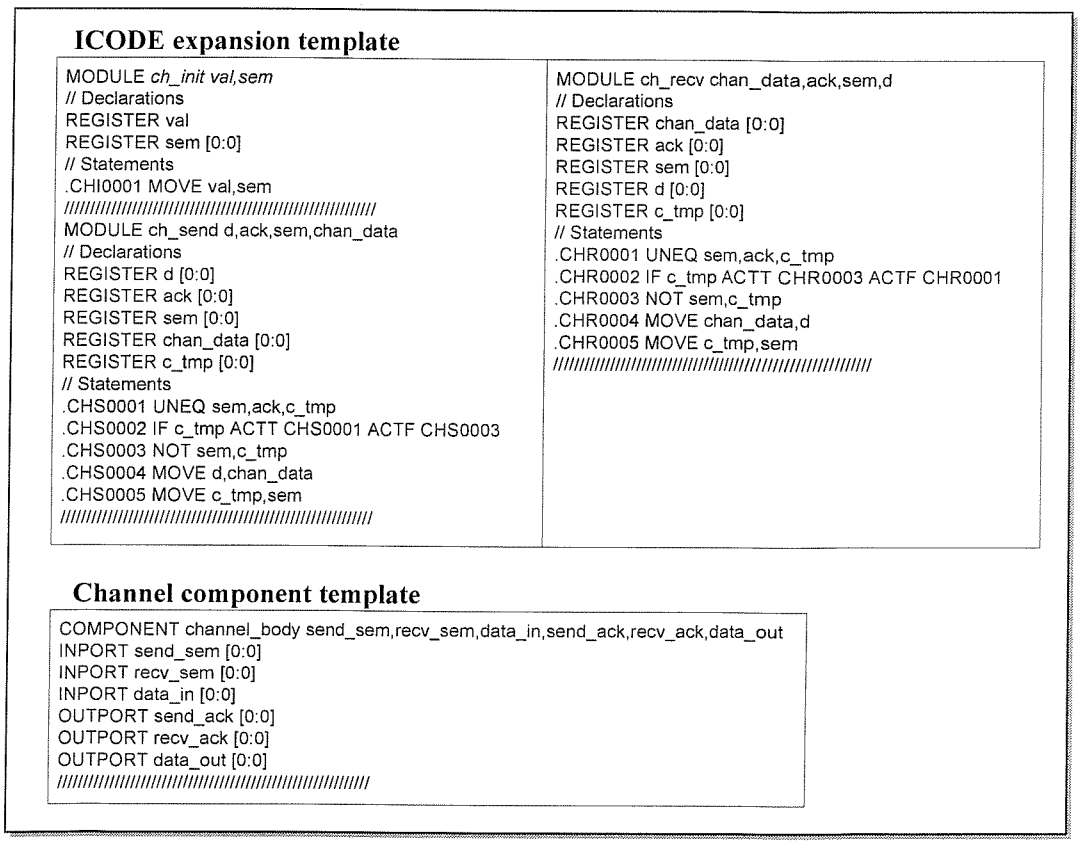


Figure 5-2 ICODE expansion and channel component templates

The ICODE *component* is equivalent to an ICODE *module* but only the interface is defined and not its behaviour. This allows concurrent VHDL "black box" components (e.g. the communication channel "black box" component in Figure 5-3(a)) with a VHDL *entity* and *architecture* (see VHDL *component*s in the VHDL hierarchy structure in Section 2.3.3) but no defined behaviour to be synthesised in MOODS and the behaviour of the component inserted after synthesis. The VHDL compiler front end does not support this "black box" concept; a dummy component consisting of a VHDL process and procedure (Figure 5-3(b)) is used, allowing the VHDL to ICODE compiler to translate an ICODE dummy component into an ICODE module activated by a MODULEAP instruction (.L000002 in Figure 5-4(a)). The generated ICODE for the behavioural example in Figure 5-3 is given in Figure 5-4(a)).

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.channel_package.all;

entity m_call1 is
  port( a: out std_logic_vector(7 downto 0) );
end m_call1;

architecture behaviour of m_call1 is
  signal c1_send_sem, c1_recv_sem: channel_sem;
  signal c1_send_ack, c1_recv_ack: channel_ack;
  signal c1_send_data, c1_recv_data: std_logic_vector(7 downto 0);
begin
  -- Communication channel blackbox component
  c1: entity work.SIMPLE_CHANNEL generic map (8)
       port map( c1_send_sem,c1_recv_sem,c1_send_data,
                 c1_send_ack,c1_recv_ack,c1_recv_data);

  -- send process
  process
    variable temp1: std_logic_vector(7 downto 0);
    variable data: unsigned(7 downto 0);
  begin
    init(c1_send_sem);
    data := (others => '0');
    forever: loop
      temp1 := std_logic_vector(data);
      send(c1_send_sem, c1_send_ack, c1_send_data, temp1);
      wait for 10 ns;
    end loop;
  end process;

  -- recv process
  process
    variable temp3: std_logic_vector(7 downto 0);
  begin
    init(c1_recv_sem);
    forever: loop
      recv(c1_recv_sem, c1_recv_ack, c1_recv_data, temp3);
      a <= temp3;
      wait for 10 ns;
    end loop;
  end process;
end behaviour;
```

(a) Behavioural VHDL design

```
library ieee;
use ieee.std_logic_1164.all;
entity SIMPLE_CHANNEL is
  generic (width: positive := 8);
  port (send_sem: in std_logic_vector(0 downto 0);
        recv_sem: in std_logic_vector(0 downto 0);
        send_data: in std_logic_vector(width-1 downto 0);
        send_ack: out std_logic_vector(0 downto 0);
        recv_ack: out std_logic_vector(0 downto 0);
        recv_data: out std_logic_vector(width-1 downto 0));
end SIMPLE_CHANNEL;
architecture structure of SIMPLE_CHANNEL is
  procedure channel_body(
    signal send_sem: in std_logic_vector(0 downto 0);
    signal recv_sem: in std_logic_vector(0 downto 0);
    signal send_data: in std_logic_vector(width-1 downto 0);
    signal send_ack: out std_logic_vector(0 downto 0);
    signal recv_ack: out std_logic_vector(0 downto 0);
    signal recv_data: out std_logic_vector(width-1 downto 0) ) is
    begin
      wait for 0 ns;
    end;
begin
process
begin
  channel_body(send_sem,recv_sem,send_data,send_ack,recv_ack,recv_data);
end process;
end structure;
```

(b) Communication channel dummy component

Figure 5-3 VHDL black box component

VHDL processes are translated and merged into the program ICODE module during the ICODE generation (Section 2.7.2). The ICODE expansion stage separates the ICODE statements (see Figure 5-4(a)) for each process into separate ICODE process modules (**p_MOD_1** and **p_MOD_2** in the given example - Figure 5-4(b)) activated by MODULEAP instructions (.L000002_0 and .L000011_0 in Figure 5-4(b)).

Figure 5-4 ICODE expansion example

The ICODE expansion stage replaces the dummy ICODE component module for the communication channel "black box" in Figure 5-4(a) with an ICODE **component** in Figure 5-4(b). Channel-related instructions (**ch_init**, **ch_send** and **ch_recv** in (a)) are replaced and inlined with the contents of expanded ICODE templates (Figure 5-2) in the ICODE expansion library.

Communication cells are inserted in the partitioning stage to handle inter-device subprogram calls; 'call_nodes' in modules calling inter-device subprogram modules are replaced with 'transmit call nodes', and 'transmit cells'. 'receive cells' are connected to 'receive call nodes' in the destination subprogram modules (described in Section 4.6). These communication cells are inserted automatically by the MOODS synthesis tool after the partitioning phase. The output structure of the partitioned design and its interface to the subprogram communication channels are not created by the user, but by the MOODS synthesis tool itself. Later sections look into the creation and the hardware connections of the subprogram communication cells and channel in greater detail.

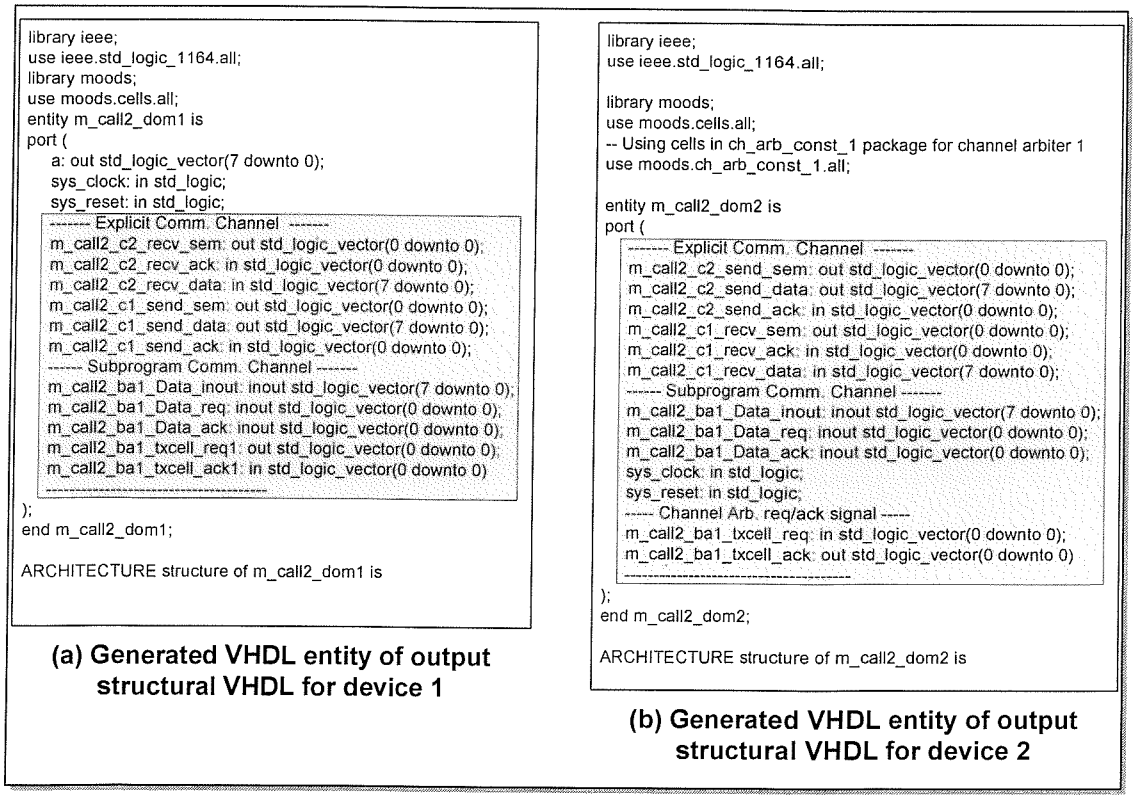


Figure 5-5 Generated VHDL entity with explicit and subprogram communication channel signal declaration

An example of the VHDL entity of two structural VHDL output files for a design partitioned into two devices is given in Figure 5-5. The interface ports that link to the structural implementation of the inter-device communication channel (both explicit communication and subprogram communication channels) are added automatically to the VHDL entity port list declaration of the generated structural VHDL design (shown shaded in Figure 5-5). The input and output signals in the VHDL entity port list declaration are grouped and mapped to the VHDL processes that access them and these signals are written to the structural VHDL output files that the processes are partitioned and assigned to. The plan was to perform most of the system enhancement through the insertion of the partitioning mechanism to partition the single design, and automatically insert the control and data path elements into the optimised design, requiring minor modifications to the MOODS synthesis core.

5.3 Communication protocol

The partitioning enhancement in the MOODS synthesis tool synthesises and generates a partitioned structural design for an arbitrary number of target FPGA devices. The communication cells in the partitioned design requires some form of arbitration as they are transferring data from one FPGA device to another via the shared subprogram communication channel. The key feature of the communication cells is in the usage of asynchronous communication techniques to transfer data between the FPGA devices. Communications synthesis [117-121], asynchronous logic synthesis [122-127] are well-researched areas and the current research of these areas investigating aspects of low-power design and system on chip design methodology [128-130]. None of the work has addressed the automatic generation and insertion of asynchronous communication channels/links during multi-FPGA system synthesis. The partitioning enhancement in MOODS utilises the principle of locality, where each FPGA device is implemented as individual processing units having an asynchronous communication interface. This concept is very similar to the Globally Asynchronous Locally Synchronous (GALS) paradigm [101, 104, 116, 131-135]. In this case, the multi-FPGA system is viewed as an

arbitrary number of FPGAs, or "locally synchronous islands" communicating asynchronously.

5.3.1 Asynchronous data transfer protocol

Subprogram channel communication between module and subprogram module is handled by the communication cells, comprising of a transmit cell (*txcell_node*), a receive cell (*rxcell_node*), and a communication channel arbiter cell (*arb*). Data transfers across clock domains use the single-rail bundled-data approach, where data is synchronised using two additional handshaking control signals (Section 3.5). The bundled-data approach uses fewer I/Os compared to an asynchronous FIFO channel.

The implementation of the data handshaking controller is not as complicated as the asynchronous FIFO and this simplicity facilitates the ease of device expansion. An arbitrary number of target devices in the multi-FPGA system, each with its local clock, can be connected to the asynchronous tri-state communication channel. An asynchronous FIFO channel forms a point-to-point unidirectional communication channel between two clock domains. Two such channels are needed to send and receive input and output (result) parameters between two domains respectively. Additional circuitry (i.e. address decoding, multiplexing control inputs, tri-state shared control signals) has to be added to the asynchronous FIFO so that the multi-FPGA system can be connected in a multipoint manner. One of the main multi-FPGA partitioning is the I/O constraints of the target FPGA devices. Additional FPGA devices or devices with more I/O pins may be required to accommodate all the signals in the design if an asynchronous FIFO channel is used.

Figure 5-6 shows an explicit communication channel and connections of the communication cells and arbiter cell generated for inter-FPGA subprogram communications through a subprogram communication channel. Each transmit cell and receive has a pair of request/acknowledge and activate/ready signals connected to the centralised communication channel arbiter respectively. To reduce I/O utilisation, the asynchronous handshaking and data signals in the subprogram communication channel are all tri-stated.

The communication channel arbiter serves dual functions in the communication protocol. Firstly, it handles the arbitration of the control of the shared communication channel for all transmit and receive cells that use the channel and it ensures a clean hand-over of ownership of the channel from one sender to another. Secondly, a lookup table in the arbiter provides a direct mapping of source modules and the corresponding destination modules to activate. Information on the creation and implementation of the communication cells are covered in greater detail in the subsequent sections.

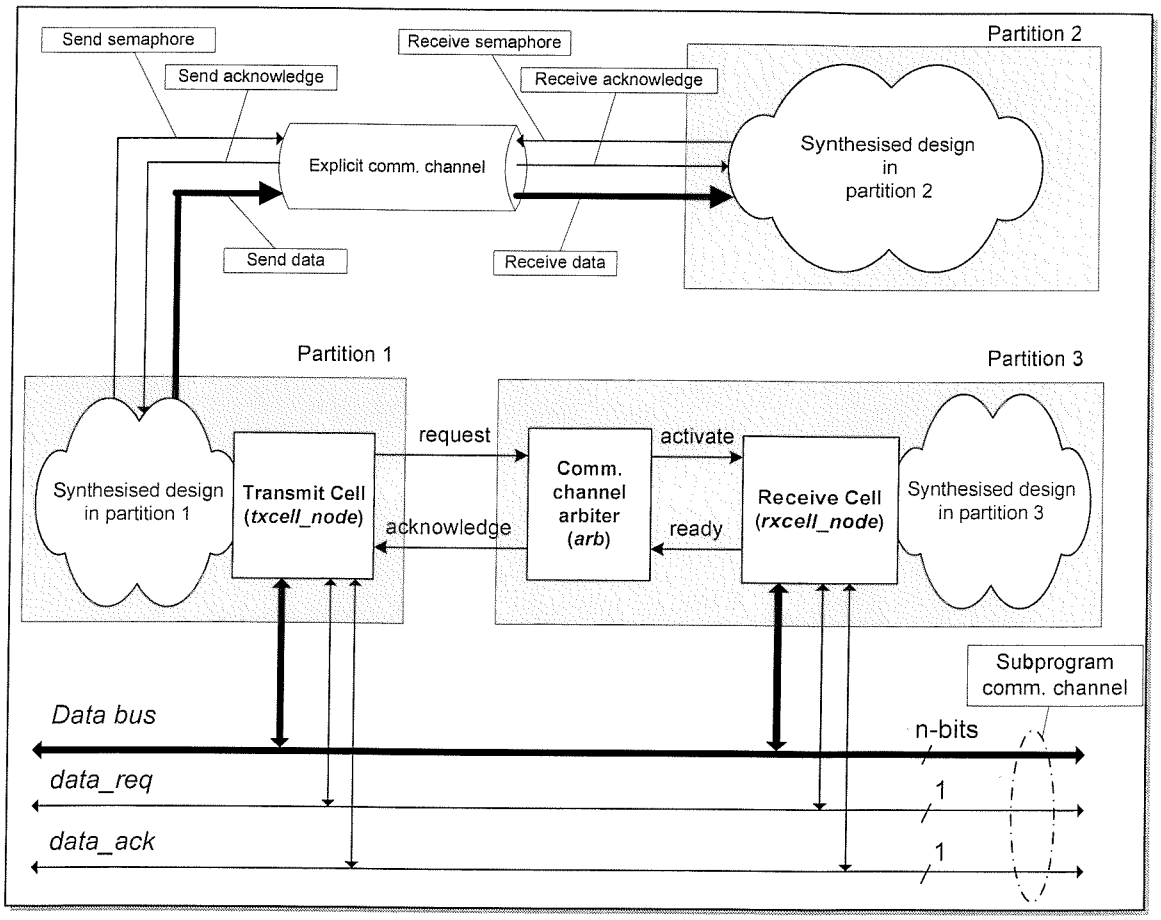


Figure 5-6 Communication cell connections in the multi-FPGA system

5.3.2 Extended burst mode state machines

The idea here is to automatically insert asynchronous data communication channels between the FPGA devices using the MOODS synthesis tool. The asynchronous channel controllers are specified using *extended burst-mode* (XBM) asynchronous state machines [123] and synthesised using the 3D synthesis system [136]. An extended burst-mode

asynchronous finite state machine is specified by a state diagram, which consists of a finite number of states, a set of labelled state transition arcs connecting pairs of states, and a start state. Each transition is labelled with a set of conditional signal levels and two sets of signal edges: an input burst and an output burst. An *input burst* is a non-empty set of input edges (terminating or directed don't care), where at least one of which must be specified. An *output burst* is a set of output edges. Figure 5-7 describes two XBM state machines for the asynchronous channel controllers for transmit and receive cells in the subprogram communication channel. Signals that are not enclosed in angle brackets and ending with + or – are *terminating edge signals* (e.g. *den*, *ack* in Figure 5-7(a) and *req*, *lastpack* in Figure 5-7(b)). The signals enclosed in angle brackets are conditionals, which are level signals whose values are sampled when all of the terminating signals associated with them have occurred. A conditional "if *lastpack* is high" represented by <*lastpack*+>, and "if *lastpack* is low" is represented by <*lastpack*->. A state transition only occurs when all the conditions are met and all the terminating signals have appeared. A slash (/) is used to delimit each input burst. A signal ending with an asterisk is a *directed don't care*. The following lists some of the labels on the state transitions in Figure 5-7:

- ***den*+ *lastpack**/ *req*+** denotes the state machine raises *req* when *den* rises regardless of the state of *lastpack*. This state transition changes from the current state *S0* to *S1* in (a).

- ***<lastpack+> ack*+ / *req- txdone*+** denotes if *lastpack* = 1 when *ack* rises, then the state machine lowers *req* and raises *txdone*. This state transition changes from current state *S1* to *S2* in (a).

- ***req- / ack-*** denotes the state machine lowers *ack* when *req* falls. This state transition changes from the current state *S2* to *S0* in (b).

Details on the formalisation of the extended burst-mode specifications can be found in [123, 137, 138].

Figure 5-7(a) describes an extended burst-mode specification for the asynchronous channel controller (*send_XBM*) that manages the protocol for sending inter-FPGA data packets, and Figure 5-7(b) describes the asynchronous channel controller (*receive_XBM*)

that manages the protocol for receiving the inter-FPGA data packets. The burst-mode specification described in (a) has four inputs (*den, lastpack, ack,* and *txdoneset*) and two outputs (*req,* and *txdone*), and (b) has three inputs (*req, lastpack,* and *rxdoneset*) and two outputs (*ack, rxdone*). Communication cells (transmit and receive cells) both have a pair of *send_XBM* and *receive_XBM* to deal with the asynchronous inter-FPGA data transfers.

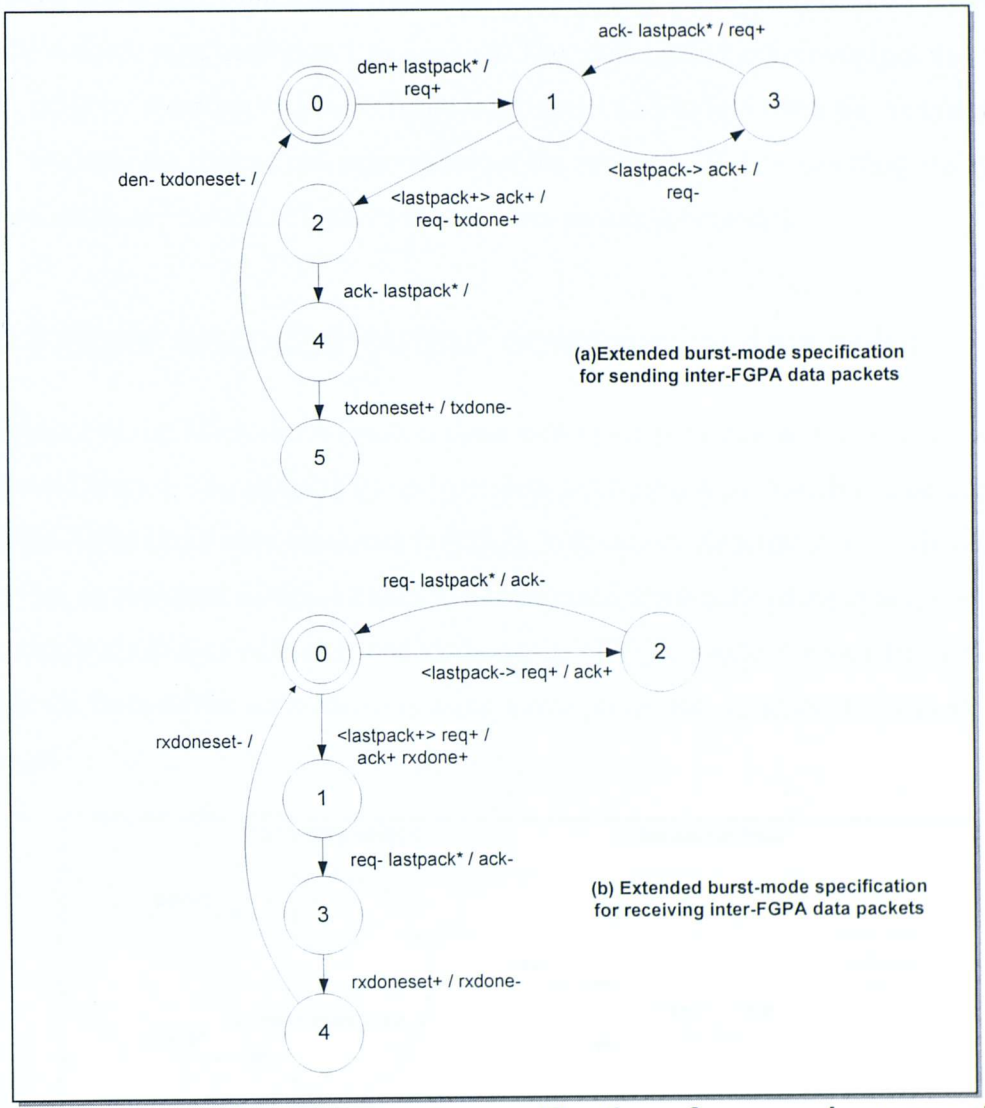


Figure 5-7 Extended burst-mode specifications for asynchronous channel controllers in communication cells

Initially, the transmit cell at the source device asserts *den* to enable inter-FPGA data on the data bus of the tri-state communication channel and its *send_XBM* asserts *req* as illustrated

in the transition from states 0 to 1 in Figure 5-7(a). When the receive cell at the destination device receives the data on the data bus, the *receive_XBM* acknowledges the sender by asserting *ack*: the transmit cell in turn negates *req*. *lastpack* is asserted when the current data packet is the last data transfer to be sent. If the data is not the last packet (<*lastpack-*>), the *send_XBM* (in the source device) and *receive_XBM* (in the destination device) continues with the four-phase handshaking protocol. When the last data packet (<*lastpack+*>) is placed on the data bus, the *receive_XBM* acknowledges the sender by asserting *ack* and *rx_done* (state transition 0 to 1 in (b)), *send_XBM* negates *req* and asserts *txdone* (state transition 1 to 2 in (a)). The transmit cell acknowledges the *send_XBM* by asserting *txdoneset* (state transition 4 to 5 in (a)) when the last data packet is sent, similarly the receive cell acknowledges the *receive_XBM* by asserting *rxdoneset* (state transition 3 to 4 in (b)) when the last data packet is received.

5.3.3 State encoded output communication cells

The output of the 3D synthesis system described in the previous section is a set of optimised hazard-free, technology-independent logic equations, which can be used to describe XBM finite state machines to handle inter-device data transfers. This section describes an alternative implementation of communication cells using synchronous finite state machines (FSM) with state encoded outputs [139] to produce glitch free FSMs to handle the inter-device data transfers using a two-phase data handshaking signalling protocol.

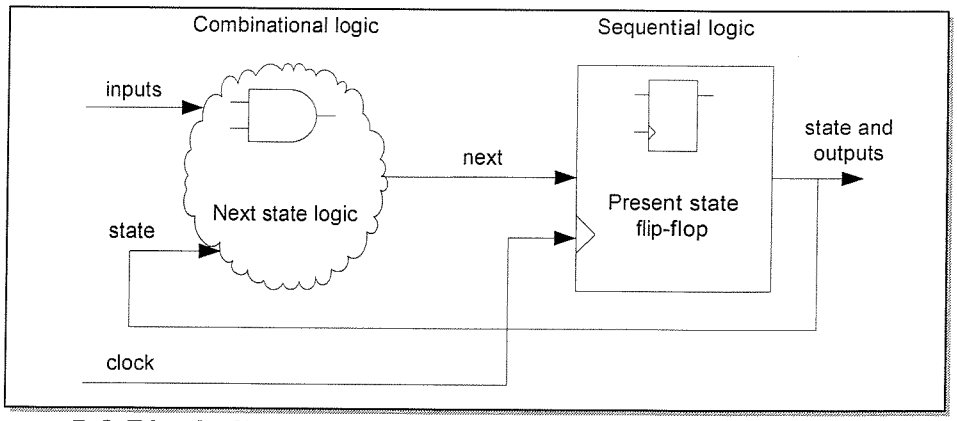


Figure 5-8 Block diagram of finite state machine with state encoded registered outputs

Figure 5-8 above shows the block diagram of the state encoded FSM, where state encodings are unique and the FSM outputs are registered and assigned directly from the

state-register bits. Finite state machines of communication cells and arbiter cells are described in subsequent sections.

5.3.3.1 Transmit cell finite state machine

This section describes the FSM of the transmit cell given in Figure 5-9 and the corresponding state encoding table in Table 5-1. The transmit cell FSM has a total of 11 states with 6 state encoded registered outputs and 2 additional state bits, *x1* and *x2*, so that all of the encodings are unique. The edge labels of the directed edges in Figure 5-9 specify the transition condition, and the corresponding effects (output values) are given in the state encoding table.

During an inter-device subprogram module call, the source module activates the transmit cell through the transmit call node (described in Section 4.4). The transmit call node asserts "*proc_en*" and the transmit cell FSM enters state *S1* and output "*transfer_req*" is asserted. When the destination receive cell is ready to receive data, the communication channel arbiter cell acknowledges the transmit cell by asserting "*transfer_ack*". The transmit cell FSM enters state *S2*, de-asserts "*transfer_req*", enables the tri-state data and handshaking signals in the communication channel with valid data by asserting transmit enable signal, "*TX_EN*" and "*data_req_out*" respectively. The destination receive cell acknowledges the receipt of the inter-device data with the assertion of the tri-state handshaking signal "*data_ack*" to complete the two-phase handshaking signalling protocol. The transmit cell FSM enters state *S4* if the preceding data packet sent is the last, else it enters state *S3* to initiate the transfer of the second data packet. The transmit cell FSM enters state *R1* when the communication channel arbiter cell de-asserts "*transfer_ack*" to complete the transfer of inter-device module input parameters.

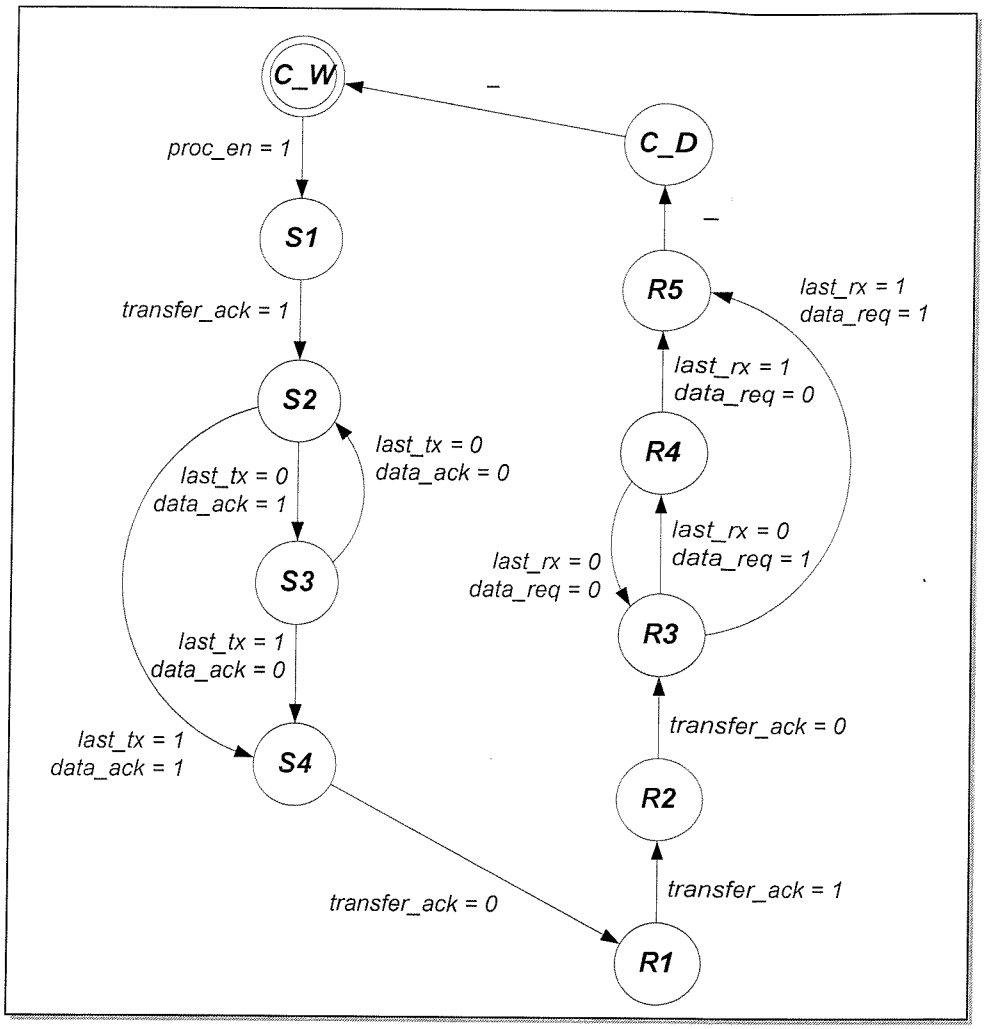


Figure 5-9 State diagram of the transmit cell FSM

The communication channel is available for other module calls while waiting for the destination module execution to complete. This non-blocking protocol is important as it allows the destination module to activate other inter-device modules without the need for a separate communication channel in the case of nested module calls.

The communication channel arbiter cell asserts the "*transfer_ack*" to indicate that the external module has completed execution and results are ready to be sent back to the source module. The transmit cell FSM enters state *R2*, output signal "*transfer_req*" is asserted and the tri-state handshake signal "*data_ack_out*" is set to logic '0'. The transmit cell FSM enters state *R3* when "*transfer_ack*" is de-asserted. The destination receive cell asserts handshake signal "*data_req*" and puts the data (results) on the tri-state communication channel. The transmit cell FSM loads in the data on the communication channel and asserts "*data_ack_out*" to acknowledge receipt of the data and enters state *S5* if the received data packet is the last, else it enters state *S4*. Output signal "*proc_done*" is

asserted in state *C_D* to end the inter-device module call and the FSM returns to state *C_W*.

State	*x2*	*x1*	*proc_done*	*transfer_req*	*RX_EN*	*TX_EN*	*data_req_out*	*data_ack_out*
CALL_WAIT (*C_W*)	0	0	0	0	0	0	0	0
SEND_1 (*S1*)	0	0	0	1	0	0	0	0
SEND_2 (*S2*)	0	0	0	0	0	1	1	0
SEND_3 (*S3*)	0	0	0	0	0	1	0	0
SEND_4 (*S4*)	0	1	0	0	0	1	0	0
READ_1 (*R1*)	0	1	0	0	0	0	0	0
READ_2 (*R2*)	0	0	0	1	1	0	0	0
READ_3 (*R3*)	1	1	0	1	1	0	0	0
READ_4 (*R4*)	1	0	0	1	1	0	0	1
READ_5 (*R5*)	1	1	0	1	1	0	0	1
CALL_DONE (*C_D*)	0	0	1	0	1	0	0	0

Table 5-1 State table of the transmit cell FSM

5.3.3.2 Receive cell finite state machine

The receive cell FSM complements the transmit cell FSM in the transfer of inter-device subprogram module data. The section describes the FSM of the receive cell given in Figure 5-10 and the corresponding state encoding table in Table 5-2. The receive cell FSM has a total of 12 states with 7 state encoded registered outputs and 2 additional state bits, *x1* and *x2*, so that all of the encodings are unique. The edge labels of the directed edges in Figure 5-10 specify the transition condition, and the corresponding effects (output values) are given in the state encoding table.

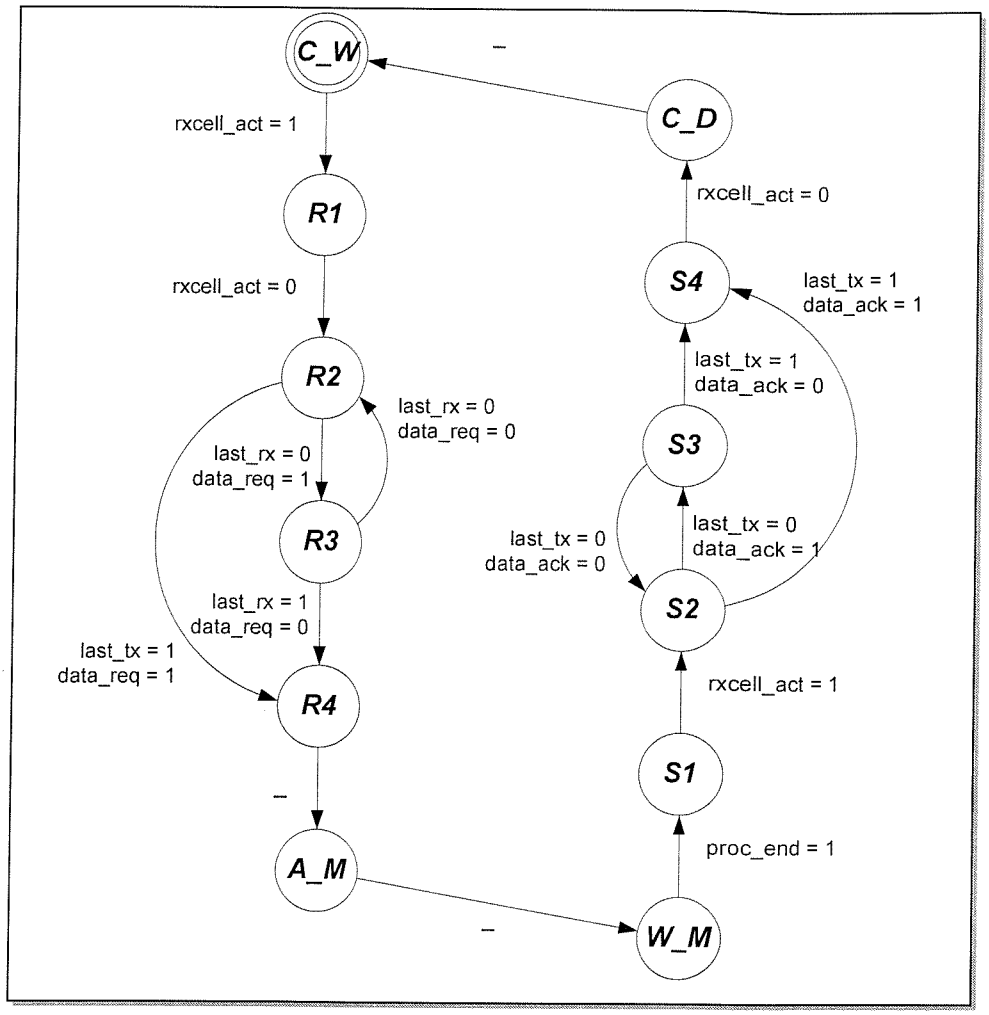


Figure 5-10 State diagram of the receive cell FSM

The receive cell FSM starts in state *C_W* and upon receive the assertion of "*rxcell_act*" from the communication channel arbiter, the FSM enters state *R1* and output signals "*rxcell_rdy*" is asserted and the tri-state handshake signal "*data_ack_out*" is set to logic '0'. The communication channel arbiter de-asserts "*rxcell_act*" when the source transmit cell is ready to send data, the receive cell FSM enters state *R2* and checks if the tri-state "*data_req*" signal is asserted to indicate valid data on the communication channel. The receive cell FSM loads in the data on the communication channel and asserts output signal "*data_ack_out*" to acknowledge receipt of the data and enters state *R4* if the received data packet is the last, else it enters state *R3*. The receive cell FSM enters state *A_M* and output signal "*proc_act*" is asserted to activate the receive call node (described in Section 4.4) and the receive cell FSM enters state *W_M* and waits till "*proc_end*" assertion by the receive call node to indicate the completion of module execution. The receive cell FSM enters state *S1* and asserts "*rxcell_rdy*". The communication channel arbiter asserts

"*rxcell_act*" to acknowledge the receive cell that the source transmit cell is ready to receive the output parameters (results) from the module execution. The receive cell FSM enters state *S2*, de-asserts "*rxcell_rdy*", enables the tri-state data and handshaking signals in the communication channel with valid data by asserting transmit enable signal, "*TX_EN*" and "*data_req_out*" respectively. The source transmit cell acknowledges the receipt of the inter-device data with the assertion of the tri-state handshaking signal "*data_ack*" to complete the two-phase data handshaking signalling scheme. The receive cell FSM enters *S3* and de-asserts "*data_req_out*". The receive cell FSM enters state *S4* if the preceding data packet sent is the last, else it enters state *S3* to initiate the transfer of the second data packet. The receive cell FSM enters state *C_D* when the communication channel arbiter cell de-asserts "*rxcell_act*" to complete the transfer of inter-device module input parameters. Output signal "*rxcell_done*" asserted in state *C_D* to activate the next control state node in the main control path and the receive cell FSM enters state *C_W* to await the next module call.

State	*x2*	*x1*	*proc_act*	*rxcell_done*	*rxcell_rdy*	*RX_EN*	*TX_EN*	*data_req_out*	*data_ack_out*
CALL_WAIT (*C_W*)	0	0	0	0	0	0	0	0	0
READ_1 (*R1*)	0	1	0	0	1	1	0	0	0
READ_2 (*R2*)	1	1	0	0	1	1	0	0	0
READ_3 (*R3*)	1	0	0	0	1	1	0	0	1
READ_4 (*R4*)	1	0	0	0	1	1	0	0	1
ACT_MOD (*A_M*)	1	0	1	0	0	1	0	0	0
WAIT_MOD (*W_M*)	1	0	0	0	0	0	0	0	0
SEND_1 (*S1*)	0	1	0	0	1	0	0	0	0
SEND_2 (*S2*)	1	0	0	0	0	0	1	1	0
SEND_3 (*S3*)	1	1	0	0	0	0	1	0	0
SEND_4 (*S4*)	0	1	0	0	0	0	1	0	0
CALL_DONE (*C_D*)	0	0	0	1	0	0	0	0	0

Table 5-2 State table of the receive cell FSM

5.3.3.3 Arbiter cell finite state machine

The communication channel arbiter cell provides arbitration for the shared subprogram communication channel. There are two types of communication channel arbiter cells, the *single-arbiter* and the *multi-arbiter* as shown in Figure 5-11. The single-arbiter (*s_arb*) cell, as the name suggests, provides arbitration to a single pair of communication cells (transmit and receive cells) using the bi-directional tri-state communication channel. The multi-arbiter (*m_arb*) cell provides arbitration to more than two communication cells using the shared bi-directional tri-state communication channel.

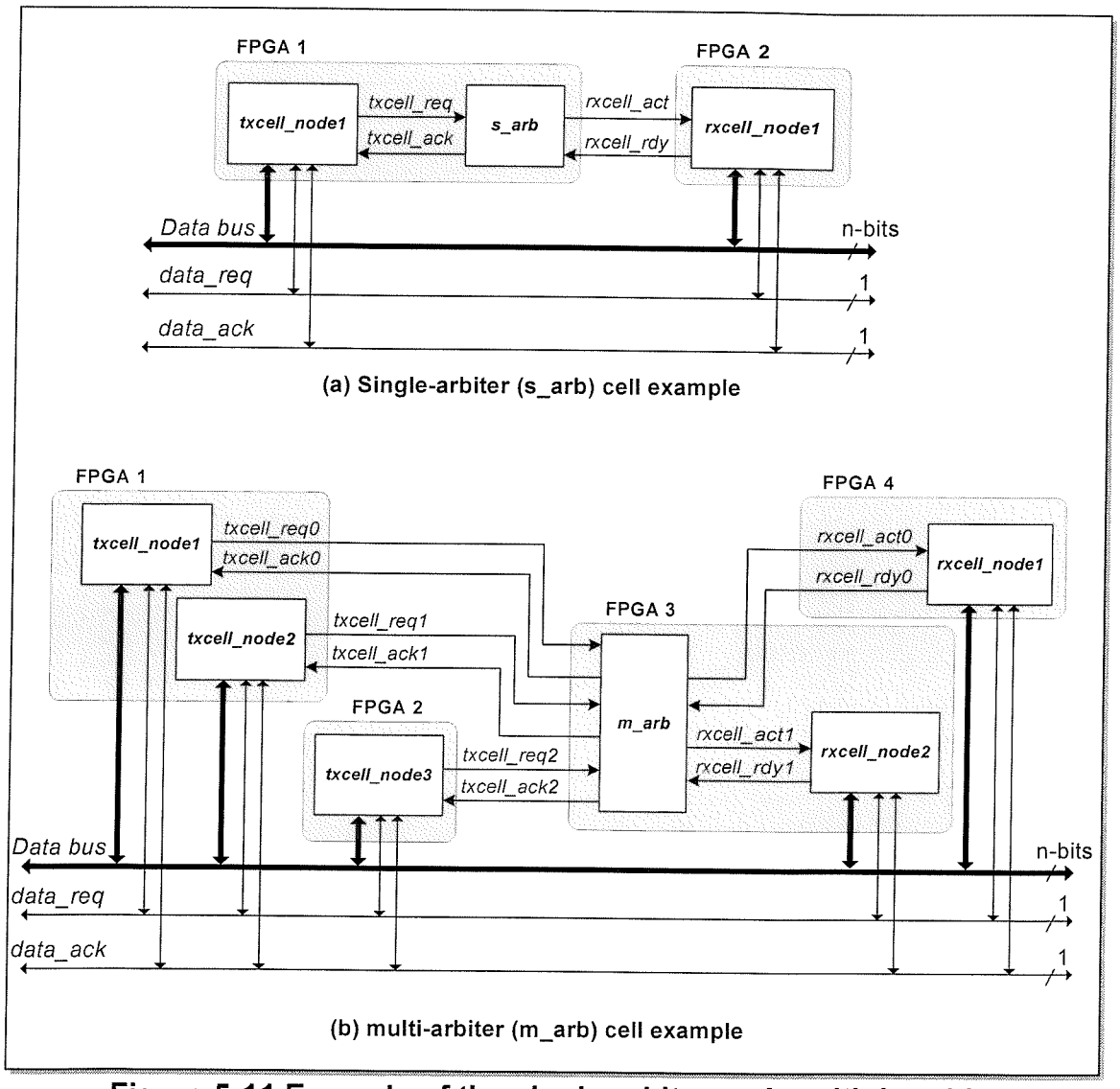


Figure 5-11 Example of the single-arbiter and multiple-arbiter

The example of a single-arbiter in Figure 5-11(a) shows a single source transmit cell (*txcell_node1*) and destination receive cell (*rxcell_node1*) connected to a single-arbiter (*s_arb*) that provides simple "one-to-one" communication channel arbitration between the

pair of communication cells. Figure 5-11(b) shows a multi-arbiter (*m_arb*) providing channel arbitration for communication cells in a "many-to-many" configuration, where transmit cells (*txcell_node1* and *txcell_node2* in FPGA 1, and *txcell_node3* in FPGA 2) sends inter-device data to receive cells (*rxcell_node1* in FPGA 4 and *rxcell_node2* in FPGA 3).

The FSM of the single-arbiter cell is given in Figure 5-12 and the corresponding state encoding table in Table 5-3. The transmit cell FSM has a total of 8 states with 2 state encoded registered outputs and an additional state bit, *x1*, so that all of the encodings are unique. The edge labels of the directed edges in Figure 5-12 specify the transition condition, and the corresponding effects (output values) are given in the state encoding table. The single-arbiter cell handles the simple "one-to-one" arbitration and a glitch-free handover of the communication channel between a pair of transmit and receive cells using a single bi-directional tri-state communication channel. The arbiter cell performs handshaking between the transmit and receive cells to ensure that the tri-state signals are enabled (set to a known level, logic '0' in this instance) by the corresponding communication cells before it acknowledges the source or destination cell to initiate the start of the inter-device transfer.

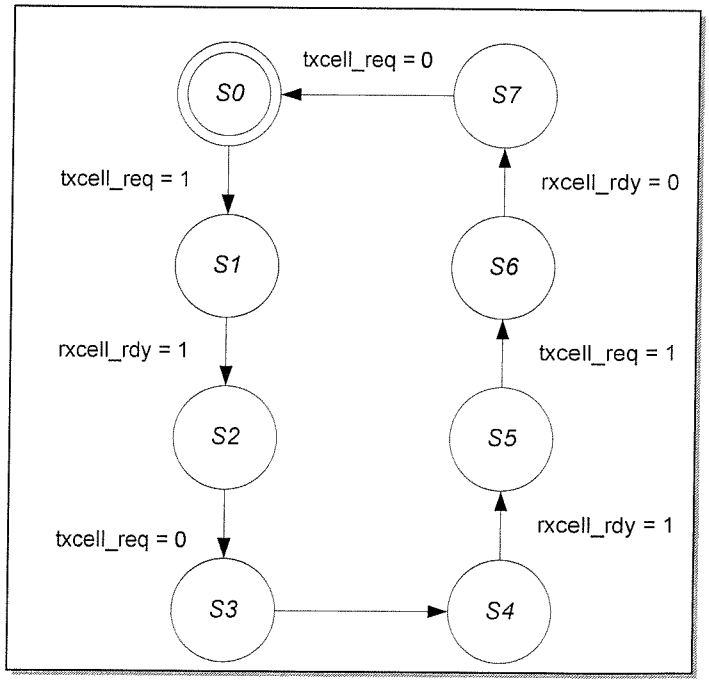


Figure 5-12 State diagram of the single-arbiter cell FSM

State	*x1*	*txcell_ack*	*rxcell_act*
S0	0	0	0
S1	0	0	1
S2	0	1	1
S3	0	1	0
S4	1	0	0
S5	1	1	0
S6	1	1	1
S7	1	0	1

Table 5-3 State table of the single-arbiter cell FSM

The multi-arbiter cell performs a similar task of communication channel arbitration as the single-arbiter cell. The multi-arbiter handles communication channel arbitration for "many-to-one" or "many-to-many" inter-device module call configuration. The multi-arbiter cell has a ROM Look-Up Table (LUT) block that holds the direct mappings of the source transmit cells and the corresponding receive cell(s) to activate. Figure 5-13 shows an example of the LUT mapping for three transmit cells and two receive cells given in Figure 5-11. The size of the LUT block is the same as the number of transmit cells connected to the multi-arbiter cell. The first and last transmit cells (*txcell_node1* and *txcell_node3*) calls *rxcell_node1* and the second transmit cell calls *rxcell_node2*. The resultant mapping in the LUT is a 0 in the first and third location of the LUT block, and a 1 in the second location of the LUT block.

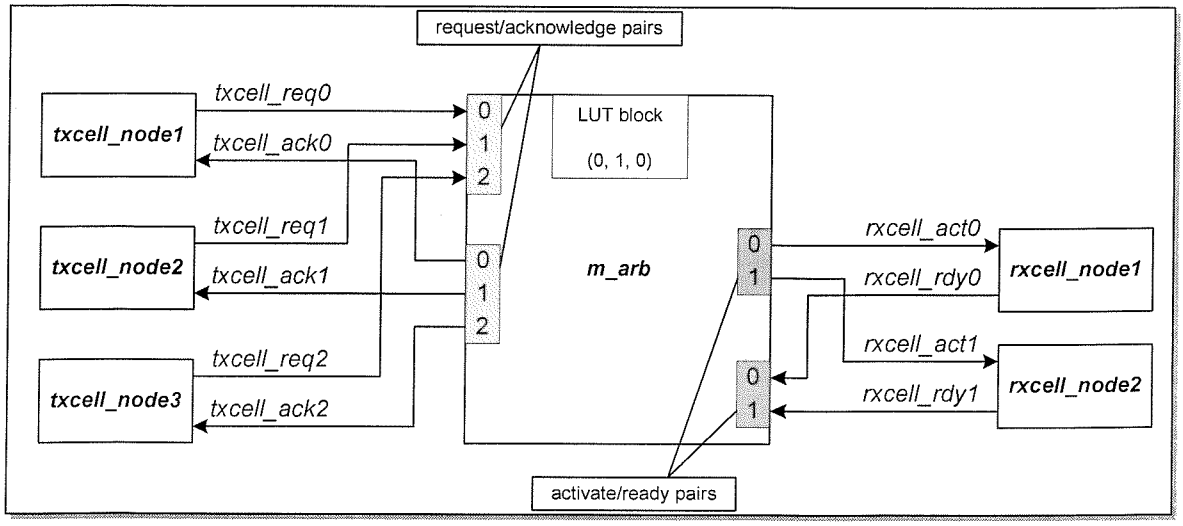


Figure 5-13 Example of LUT mapping of communication cells

The FSM of the multi-arbiter cell with a total of 10 states is illustrated in Figure 5-14. The multi-arbiter cell checks the connected transmit and receive cells in a round-robin manner. A token with an initial value of one is incremented by one every time the multi-arbiter cell FSM enters state *I_T*. The token is cleared to zero when the preceding token value is the maximum count value given by one less the maximum number of transmit cells connected to the multi-arbiter cell (For example, the total number of connected transmit cells given in Figure 5-11(b) is: 3, hence the maximum token count value is: 3 -1 = 2).

The multi-arbiter cell performs a prioritised condition check in state *C_S*, whereby condition A has a higher priority than condition B. Condition A checks the "*transfer_req*" input signal specified by the token value for an inter-device module call. The "*mod_active*" register in the multi-arbiter is set to '1' if the destination module is not available. The "*mod_active*" register bit of an activated receive cell is set to '1' and the transmit cell that activated the receive cell will have a '1' set in the "*call_reg*" register. Condition B checks the completion of execution from the destination module. This condition is true when the corresponding bits in the "*mod_active*" and "*call_reg*" are set and "*rxcell_rdy*" is asserted by the activated receive cell. The multi-arbiter cell FSM enters state *I_T* when conditions A and B are not met.

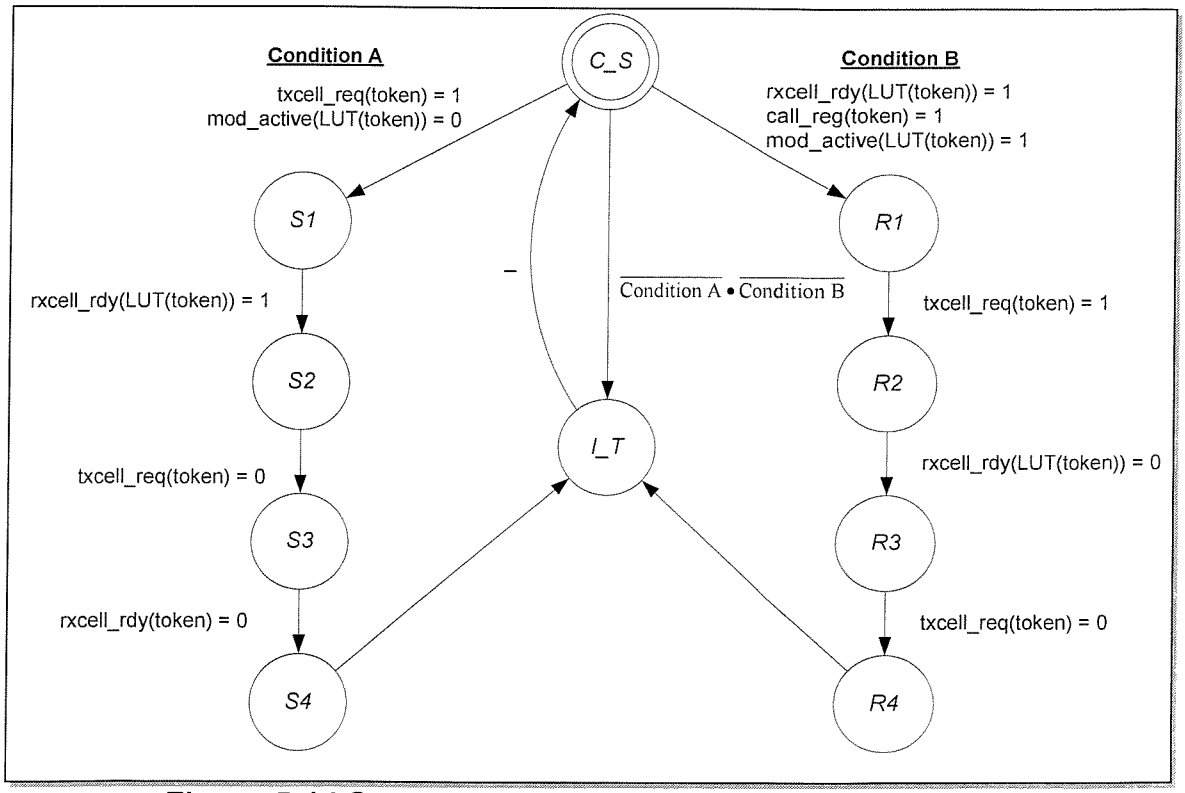


Figure 5-14 State diagram of the multi-arbiter cell FSM

States *S1* to *S4* in the multi-arbiter cell FSM handles the inter-device data (input parameters) transfers from the source transmit cell to the receive cell in the destination device. States *R1* to *R4* handles the inter-device data (results from the module execution) transfers from the receive cell to the source transmit cell. Status register bits to identify active transmit and receive cells in "*call_reg*" and "*mod_active*" respectively are set in state *S2*. The corresponding bits in the status registers are cleared in state *R3* to permit future inter-device module call activations. The registered output signals in the multi-arbiter cell are given in Table 5-4 below.

State	*txcell_ack(token)*	*rxcell_act(token)*
CHECK_SIG (*C_S*)	–	–
INCREMENT_TOKEN (*I_T*)	–	–
S1	0	1
S2	1	1
S3	1	0
S4	0	0
R1	1	0
R2	1	1
R3	0	1
R4	0	0

Table 5-4 Registered output signals in the multi-arbiter cell FSM

5.3.4 Data transfer protocol for communication cells

Four-phase signalling protocol is used in the handshaking of request and acknowledge signals in the subprogram communication channel arbitration, and two-phase signalling protocol is used to indicate data is valid on the tri-stated communication channel (data bus) and acknowledge the acceptance of data at the corresponding receiver cell. There are a total of sixteen events in the shared communication channel arbitration for each cross-domain subprogram call (illustrated in Figure 5-15). The first eight events (labelled 1 to 8 in Figure 5-15) correspond to the signalling of handshake signals used in the passing of input parameters from the source module through the activated transmit cell (*txcell_node*) to the destination module through the activated receive cell (*rxcell_node*).

The remaining 8 events (labelled 9 to 16) correspond to the returning of output parameters (results) from the destination module to the corresponding calling source module. The *rxcell_node* activates the subprogram module when all the input parameters are received and loaded at the destination domain (event 7). Subprogram execution completes and the *rxcell_node* asserts *ready* signal (event 9) to indicate that the results from the subprogram execution are ready to be sent back to the calling module.

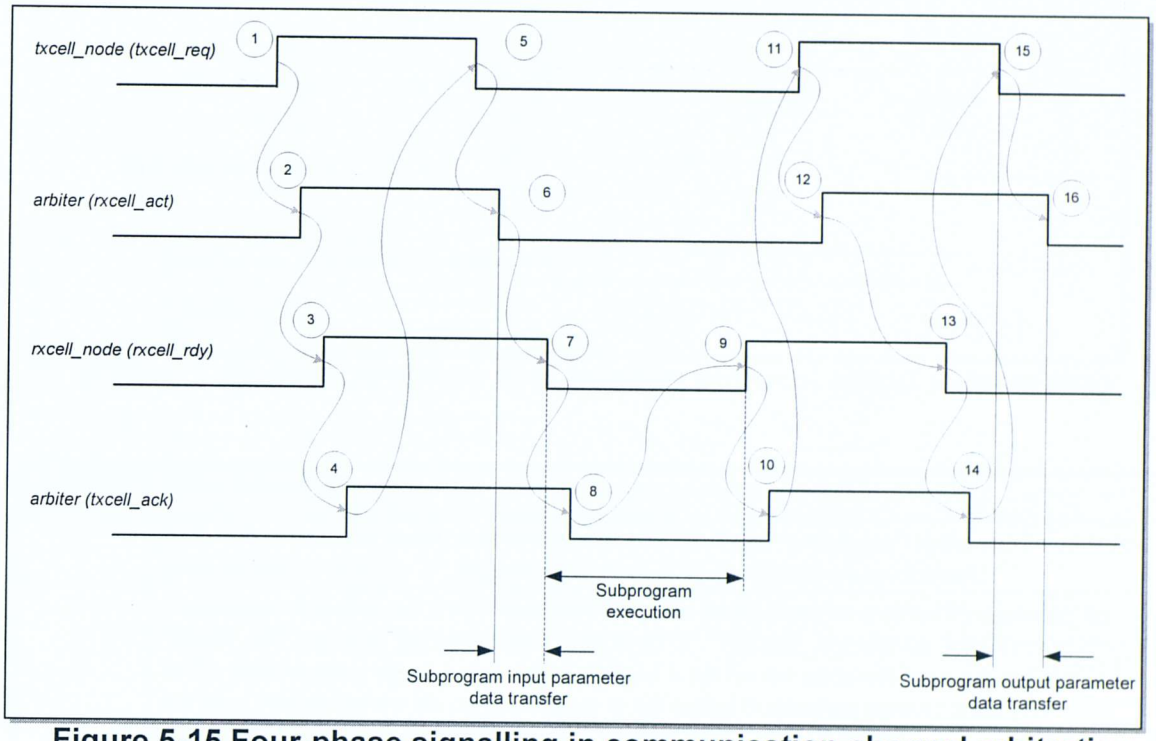


Figure 5-15 Four-phase signalling in communication channel arbitration

Figure 5-16 illustrates the sequence of events (labelled 1 to 8) in the passing of input parameters, with the asynchronous data handshaking signals (*data_req*, *data_ack*, and *Data*). A description of the sequence of events corresponding to the passing of input parameters marked in Figure 5-16 is listed in Table 5-5.

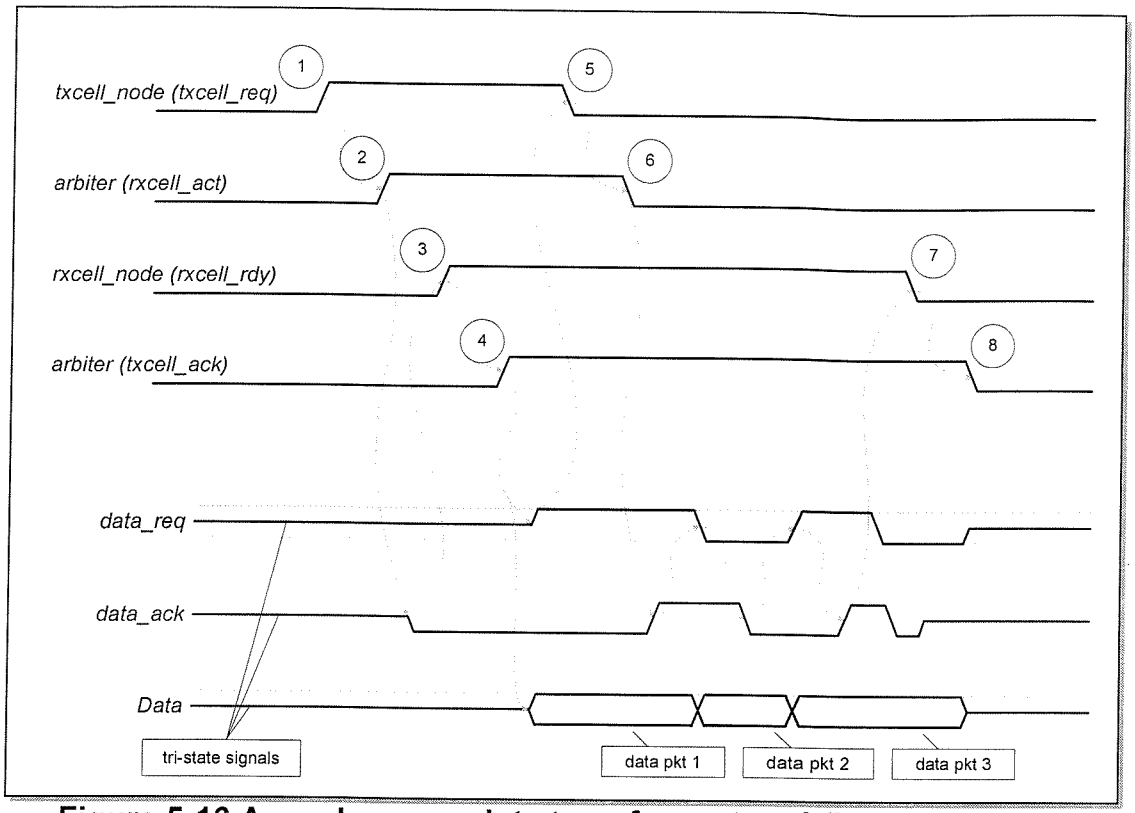


Figure 5-16 Asynchronous data transfer protocol (input parameters)

EVENT	DESCRIPTION
1	Transmit cell (*txcell_node*) is activated and request signal "*txcell_req*" to the communication channel arbiter is set to '1', requesting control of the communication channel.
2	Communication channel arbiter (*arbiter*) activates the destination module by asserting the "*rxcell_act*" signal of the destination module to '1'. The *call_reg* register bit corresponding to the input *request* signal being acknowledged is set (in the multi-arbiter cell) and similarly the *mod_active* register bit corresponding to the called destination module is set (see previous section for more details on *call_reg* and *mod_active* registers).
3	Receive cell (*rxcell_node*) initialises the tri-state *data_ack* handshaking line to '0' and acknowledges the communication channel arbiter activation by asserting its "*rxcell_rdy*" signal to '1'.
4	Communication channel arbiter acknowledges the *txcell_node* by asserting "*txcell_ack*" to '1'.
5	*txcell_node* enables the tri-state *Data* bus with first data packet is placed on the *Data* bus, and "*data_req*" handshaking signal is asserted to '1' to initiate the inter-device data transfer. *txcell_node* de-asserts "*txcell_req*" signal.
6	Communication channel arbiter sets "*rxcell_act*" signal to '0', telling *rxcell_node* in the destination domain that the tri-state data handshaking lines are initialised and inter-device data are ready to be received. The receive cell loads in and acknowledges the data packets sent by the transmit cell.
7	Receive cell loads in the last data packet and de-asserts "*rxcell_rdy*" signal to '0'. Receive cell releases control of the tri-state "*data_ack*" line.
8	Communication channel arbiter sets "*txcell_ack*" signal to '0' and this completes the data transfer protocol for the transfer of cross-domain input parameters.

Table 5-5 Sequence of events in the asynchronous data transfers protocol (input parameters)

Figure 5-17 below illustrates the sequence of events (labelled 9 to 16) in the passing of output parameters, with the asynchronous data handshaking signals (*data_req*, *data_ack*, and *Data*). A description of the sequence of events corresponding to the passing of output parameters marked in Figure 5-17 is given in Table 5-6.

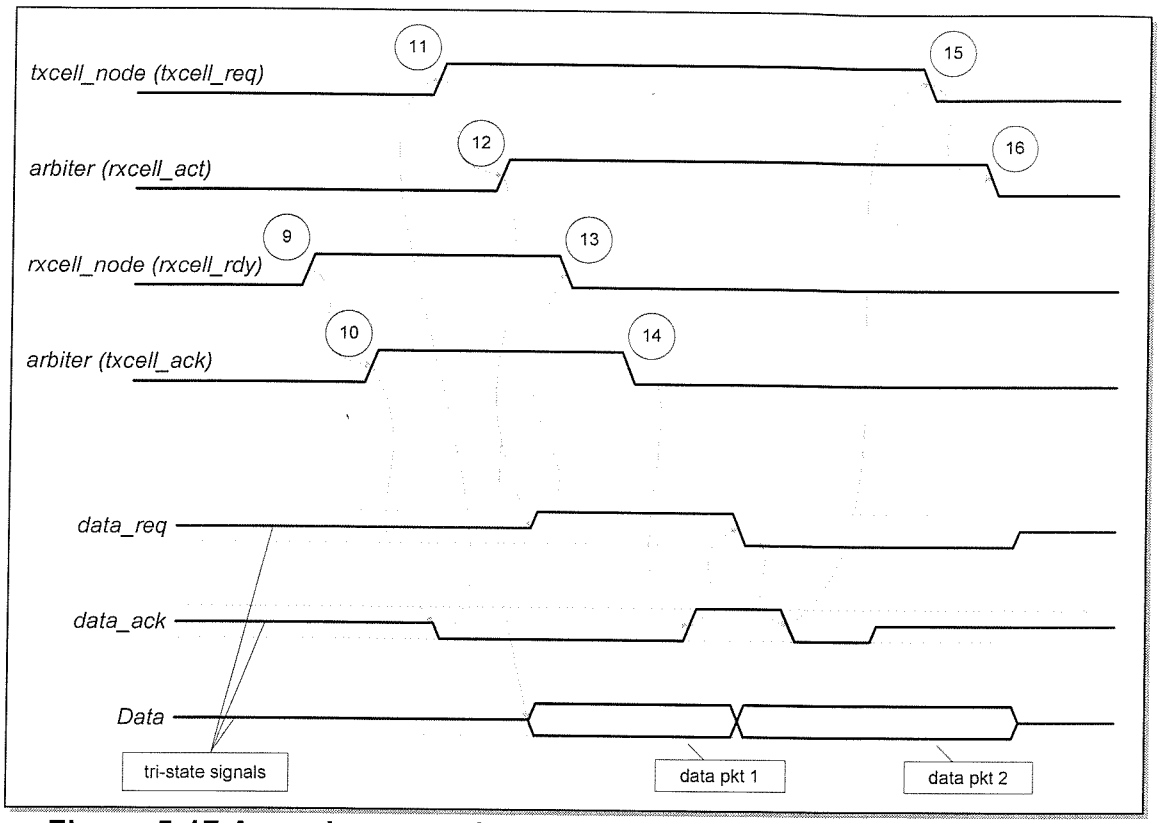


Figure 5-17 Asynchronous data transfer protocol (output parameters)

EVENT	DESCRIPTION
9	Upon completion of the subprogram execution, receive cell (*rxcell_node*) asserts its "*rxcell_rdy*" signal to '1'.
10	Communication channel arbiter (*arb*) asserts *"txcell_ack"* signal indicating to the *txcell_node* that the subprogram has completed its execution.
11	*txcell_node* initialises the *data_ack* handshaking line to '0' and asserts *"txcell_req"* to '1'.
12	The *call_reg* register bit corresponding to the input *request* signal being acknowledged is cleared (in the multi-arbiter cell) and similarly the *mod_active* register bit corresponding to the called destination module is cleared (see previous section for more details on *call_reg* and *mod_active* registers). Communication channel arbiter acknowledges the r*xcell_node* by asserting "r*xcell_act*" to '1'.
13	*rxcell_node* enables the tri-state *Data* bus with first data packet is placed on the *Data* bus, and "*data_req"* handshaking signal is asserted to '1' to initiate the inter-device data transfer. *rxcell_node* de-asserts "*rxcell_rdy*" signal.
14	Communication channel arbiter sets "*txcell_ack*" signal to '0'. This tells *txcell_node* in the source domain that the tri-state data handshaking lines are initialised and inter-device data are ready to be received. The transmit cell loads in and acknowledges the data (result) packets sent by the receive cell.

EVENT	DESCRIPTION
15	Transmit cell loads in the last data packet and de-asserts "*txcell_req*" signal to '0'. Transmit cell releases control of the tri-state "*data_ack*" line.
16	Communication channel arbiter sets the *"rxcell_act"* to '0' and this completes the data transfer protocol for the transfer of cross-domain output parameters.

Table 5-6 Sequence of events in the asynchronous data transfers protocol (output parameters)

5.4 Subsystem architecture

This section starts with the details on the creation and implementation of the asynchronous subprogram communication channels. Implementation details of the various communication subsystem interface cells; transmit and receive cells and the communication channel arbiter cell are covered within this section.

5.4.1 Transmit cell

The 'transmit cell' (*txcell_node*) is the inter-FPGA communication interface cell inserted into the source module that calls a destination module in another partition mapped onto a separate FPGA device. The original *call_node* associated with a subprogram call is replaced by the *tcall_node* if the called module is allocated a separate partition. For each *tcall_node*, a *txcell_node* is added into the structural output to handle the handshaking and transfer of I/O parameters across the communication channel.

The width of the communication channel is optimised by the MOODS synthesis tool based on the number of available user I/Os of the interconnected FPGAs and the width of the input and output parameters of the subprogram. The input and output parameters are concatenated and sent in data packets, where the size of each data packet is the width of the communication channel used to send the data. If the bit-width of the last data packet is not less than the width of the communication channel, the last data packet is bit-stuffed with zeros to the full bit width. A multiplexor is created to select the appropriate data for a multi-packet subprogram input parameter transfer. The multiplexor select signals are

driven from the *txcell_node*. Figure 5-18 shows the structure generated for a subprogram with five input parameters (A, B, C, D, and E) of varying bit widths and the communication channel has a width of 16 bits.

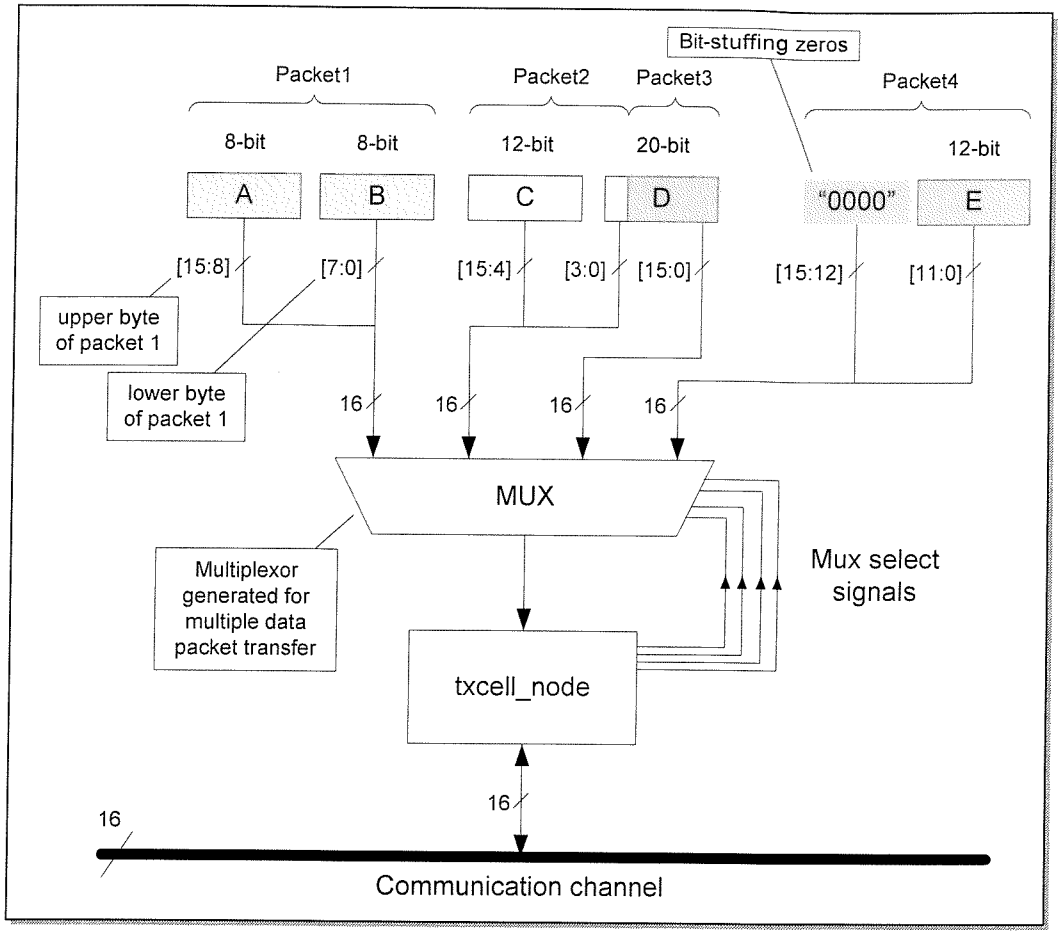


Figure 5-18 Generated structure for a multi-packet input data transfer via the *txcell_node*

If an output parameter is sent over the communication channel in multiple data packets, only the corresponding bits of the register are updated for each packet transferred. This is achieved in a similar way to the input parameters multiplexor select signals, where instead of using the load-enable signal directly for each register, the load-enable signal is 'ANDed' with the output parameter select signals driven from the *txcell_node*. Figure 5-19 shows the structure generated for a subprogram with 3 output parameters (X_out, Y_out, and Z_out) of varying bit widths and the communication channel has a width of 16 bits. Latches are used in place of registers to hold the output parameters when XBM finite state machine are used instead of FSM with state encoded outputs; details on the creation and register-to-latch modifications are covered in Section 5.5.1.

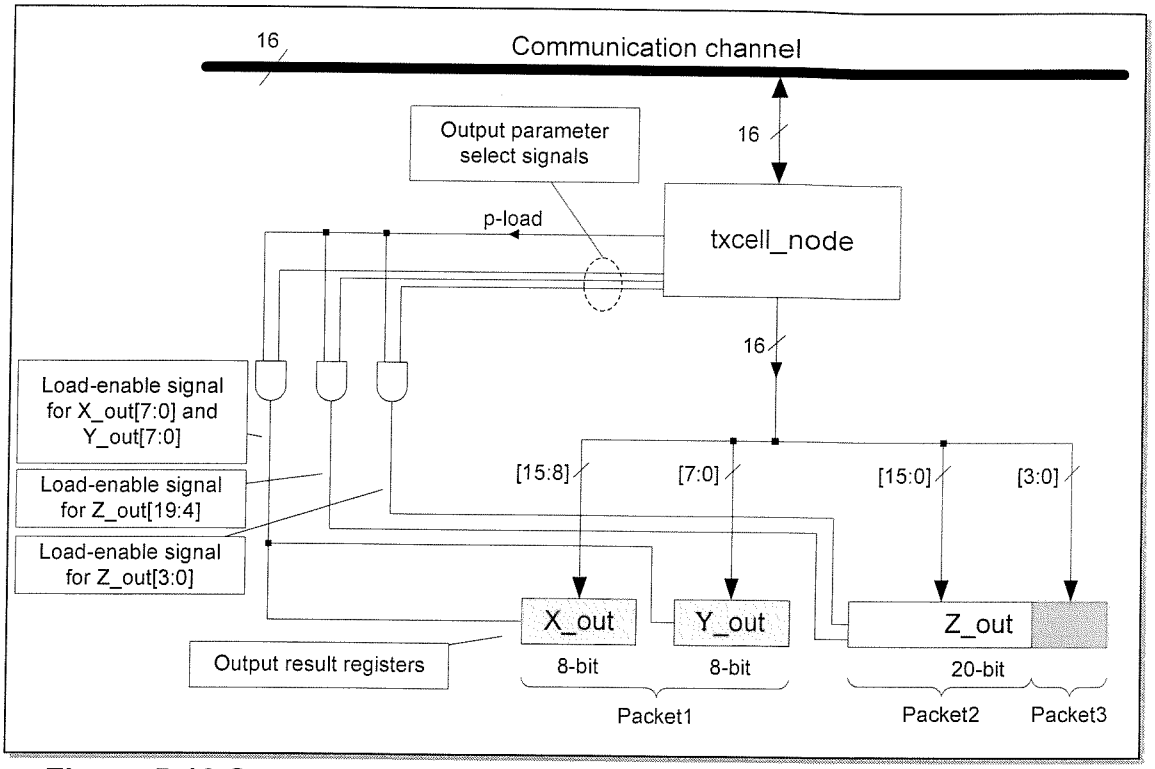


Figure 5-19 Structure generated for receiving a multi-packet output data transfer via the *txcell_node*

A single *txcell_node* is shared between *tcall_node*s, which are mapped to the same FPGA device, and calling the same destination module. For a *txcell_node* that is shared by two or more *tcall_nodes*, a multiplexor is created to select the input parameters associated with the activated *tcall_node*. Figure 5-20 shows the structure generated for a *txcell_node* shared by two subprogram calls (ICODE instructions '*.L0001*' and '*.L0005*').

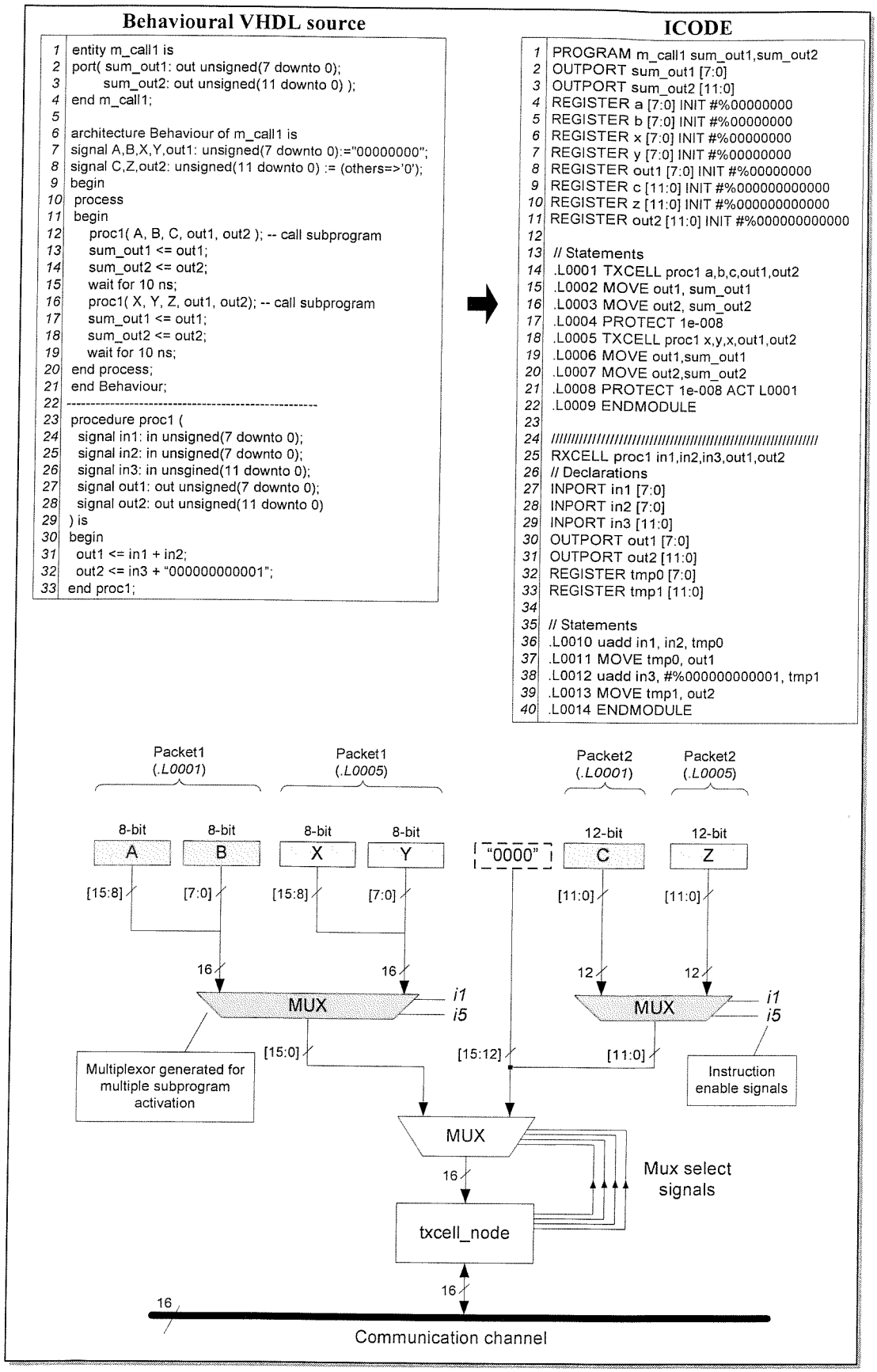


Figure 5-20 Generated structure for a shared *txcell_node*

5.4.2 Receive cell

The 'receive cell' (*rxcell_node*) is the inter-FPGA communication interface cell inserted into the subprogram module that is called by a source module in another partition, which is mapped onto a separate FPGA device.

The ICODE *RXCELL* instruction replaces the original ICODE *MODULE* module header instruction when the module is called by a module in another FPGA device. For each *RXCELL* module, a receive cell (*rxcell_node*) and a receive call node (*rcall_node*) is created to handle the inter-FPGA data transfer and initiating the execution of instructions within the module. The *rxcell_node* is added into the structural output to handle the handshaking and transfer of I/O parameters in data packets across the communication channel. The *rcall_node* has the same structure as the *call_node*, the only difference is the *rcall_node* is activated by the *rxcell_node*. Upon completion of the subprogram execution, control is passed back to the *rxcell_node*, which then initiates the return of the output results to the calling module.

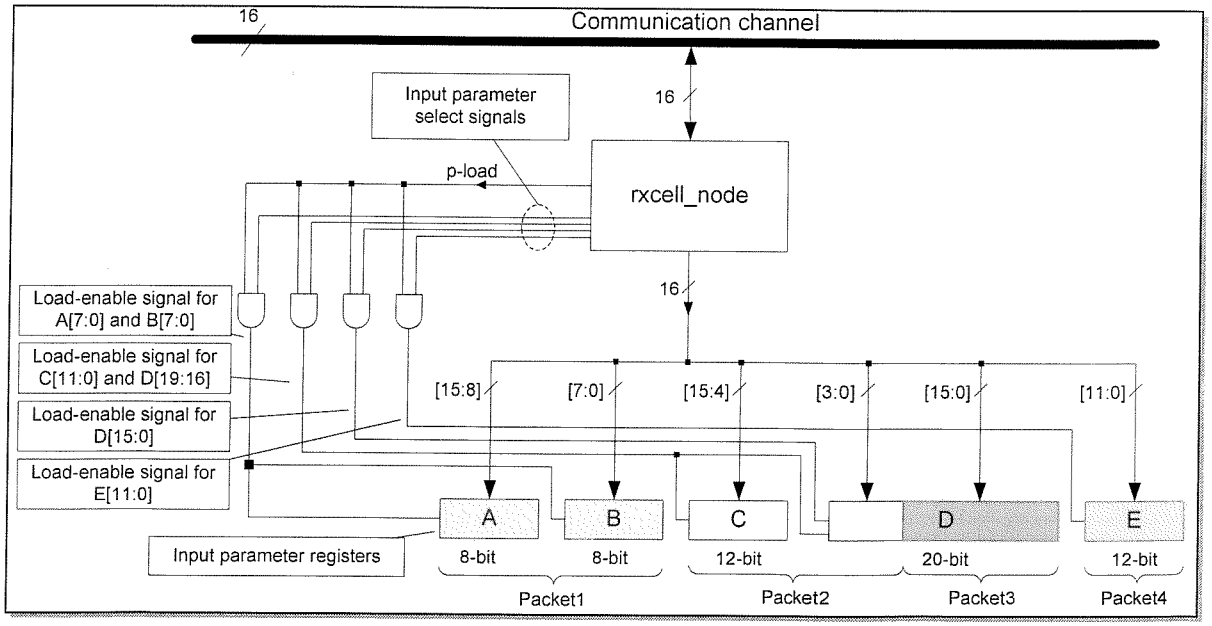


Figure 5-21 Structure generated for receiving a multi-packet input data transfer

The *rxcell_node* receives the concatenated input parameters sent by the *txcell_node* of the calling module. The input parameters in the data packets are then loaded into the corresponding input parameter registers. Recall the structure generated for a multi-packet input data transfer in Figure 5-18, Figure 5-21 illustrates the structure generated in the

destination sub-module to receive and load the input parameters. This structure is identical to the structure generated to load output results via the *txcell_node* in the calling module as described earlier.

Latches are used in place of registers to hold the input parameters when XBM finite state machine are used instead of FSM with state encoded outputs; details on the creation and register-to-latch modifications are covered in Section 5.5.1.

A multiplexor is created to select the appropriate data packet for a multi-packet subprogram output result transfer. The multiplexor select signals are driven from the *rxcell_node*. Recall the structure generated to receive a multi-packet output data transfer in Figure 5-19, Figure 5-22 illustrates the structure generated in the destination sub-module to send the results back to the *txcell_node* of the calling module. This structure is identical to the structure generated to load input parameters via the *txcell_node* in the calling module as described earlier.

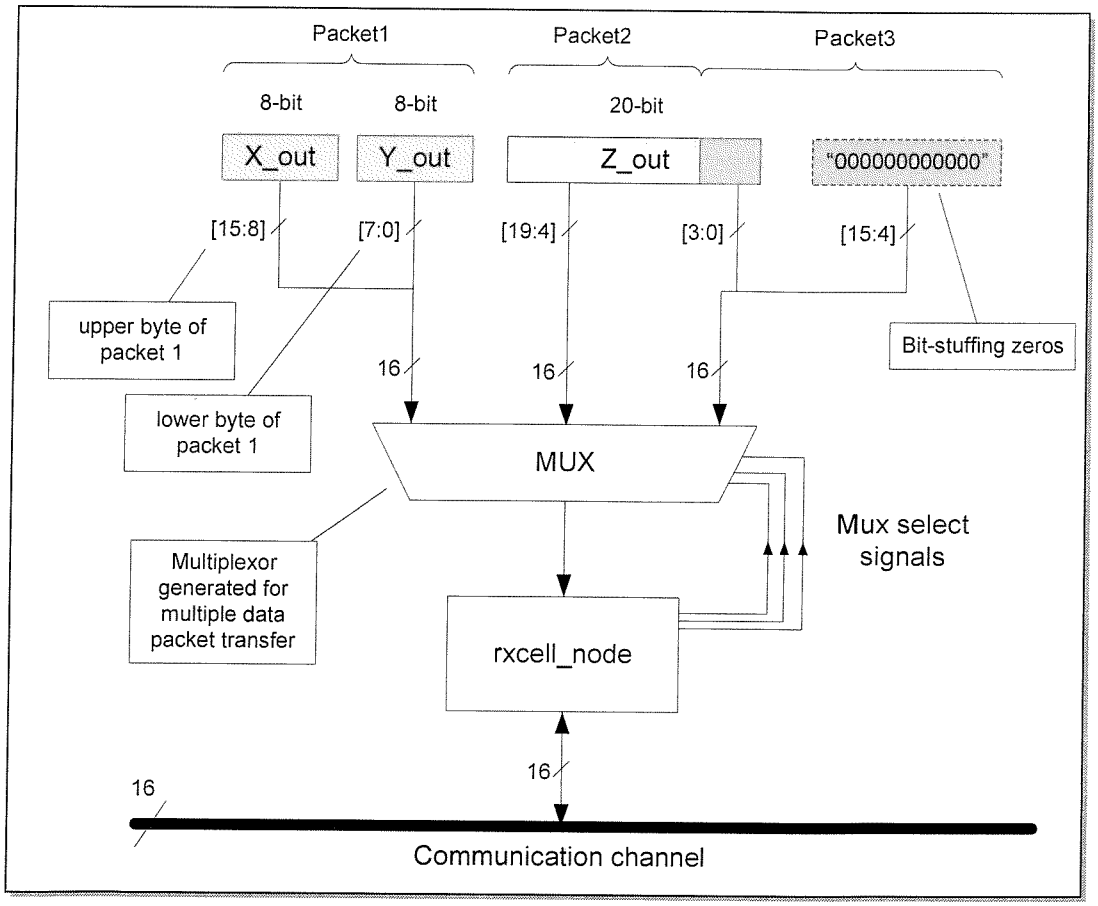


Figure 5-22 Generated structure for receiving a multi-packet output data transfer via the *rxcell_node*

5.4.3 Communication channel (data bus) arbiter

The communication cells (transmit cell and receive cell) send and receive data packets over a shared communication channel and these communication cells are connected to a centralised arbiter granting the usage of the communication channel. During synthesis, the mapping for each source-destination module pair in an inter-FPGA subprogram call is determined and this provides a direct mapping of the 'calling' module in one FPGA device and the 'called' module in another device. This mapping information provides the values to the ROM Look-Up Table (LUT) block in the arbiter. A round-robin (rotating) priority scheme is implemented in the communication channel arbiter, where request lines are polled in a rotating manner. The sequence of events is described in Table 5-5 and Table 5-6. The asynchronous data transfer protocol sets up the *txcell_node* and *rxcell_node* in different FPGA devices for the asynchronous inter-FPGA data transfer. Figure 5-23 illustrates the LUT and status registers (*call_reg* and *mod_active*) structure in the multi-arbiter.

The interface ports that link to the structural implementation of the arbiter are added automatically to the entity port list declaration of the generated structural VHDL design. The MOODS synthesis tool checks for communication cells (*txcell_nodes and rxcell_nodes*) that are in the same partition as the communication channel arbiter when the communication channel arbiters are created. These 'internal' communication cells are connected directly to the communication channel arbiter via internal signal nets. The synthesis tool determines the sizes of the external signals that interface with the arbiter, and resizes the interface ports.

The communication channel arbiter is defined within a VHDL package generated by the MOODS synthesis tool. There are two reasons for defining the communication channel arbiter as a separate package. Firstly, the size of the LUT block is determined when the communication channel arbiter is created during the post-processing phase of the MOODS synthesis tool, and these memory elements are customised in the structural/RTL communication channel arbiter component to support all the communication cells connected to its interface ports. The second reason is the creation of the LUT mapping of the communication cells, which is a direct one-to-one mapping with the input request lines from source *txcell_nodes*, and the LUT value addressed by the index corresponds to the

output activation lines to destination *rxcell_nodes*. The mapping information is only available after partitioning and allocation of arbiters to FPGA devices. The synthesis tool also checks for arbiters providing arbitration to just a single source-destination module pair, the LUT is not required and a single-arbiter is used instead.

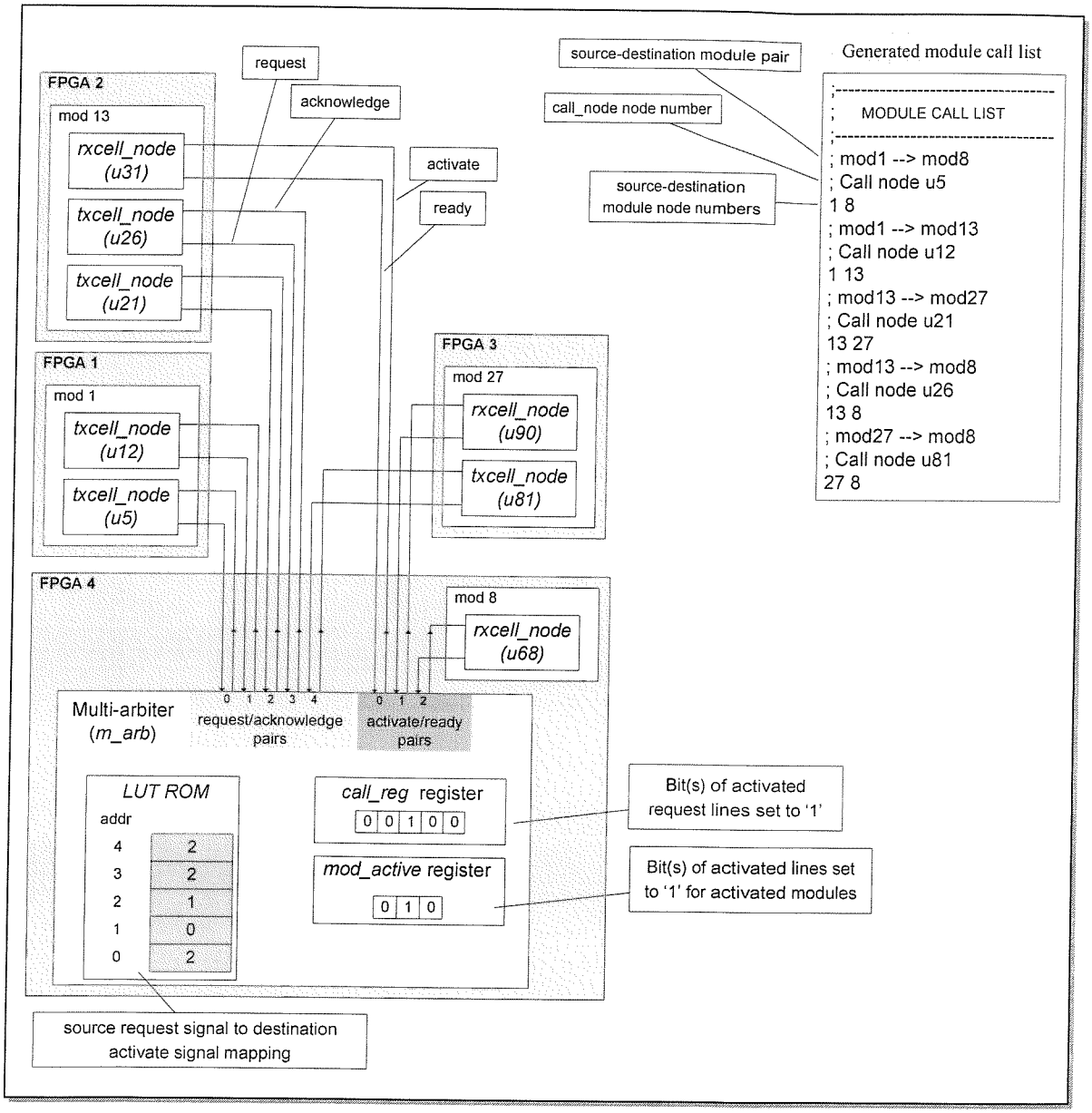


Figure 5-23 Look-up table block and status registers in the multi-arbiter cell

5.5 Hardware generation

With the inclusion of the communication subsystem providing the subprogram communication channel for the asynchronous transfer of data between multiple target

devices, various enhancements and modifications are made to the single output structure generated by the MOODS synthesis system. Firstly, the MOODS synthesis system now generates multiple output structural output files from a single input behavioural description; one for each target device. A new latch component is added into the MOODS technology (cell) libraries, latches are used in place of data-gated registers in some parts of the design where asynchronous data transferred over the communication channel is loaded independent of the system clock by XBM finite state machines. For communication cells using FSM with state encoded outputs, data-gated registers in the existing MOODS technology libraries are used.

5.5.1 Data latch generation and hardware duplication

The register arrangement for the original subprogram (module) I/O parameters is shown in Figure 5-24, where the original structure uses pass-by-reference for subprogram I/O parameters. Output results obtained from the subprogram execution are written directly to the corresponding output registers (*out_X* and *out_Y* in the figure). The data path storage units (registers) implemented for the subprogram output parameters are bypassed and optimised away (removed), as shown shaded in the figure.

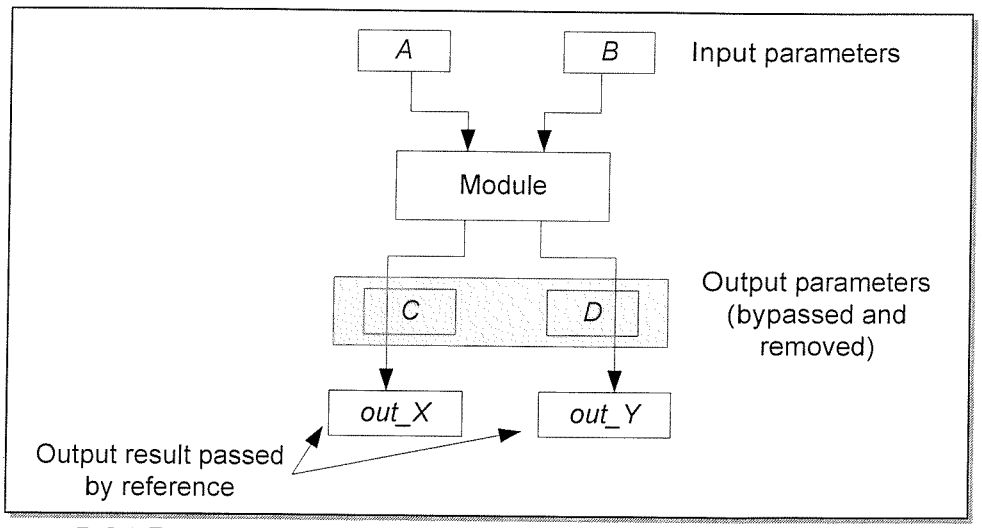


Figure 5-24 Register arrangement for original subprogram module I/O parameters

The communication subsystem transfers the input and output parameters of external subprogram modules asynchronously via a pair of transmit and receive communication cells. The subprogram module call mechanism is modified (described in Section 4.4) and the underlying structure of the final generated hardware uses pass-by-value instead of

pass-by-reference for the procedures I/O parameters. Figure 5-25 illustrates the latch and duplicated register arrangement for a subprogram module that is being called from a module in a different target device (FPGA) using communication cells with XBM asynchronous finite state machines. The figure shows two FPGA devices, where *FPGA 1* is the source device, which contains the calling module, and the called subprogram module is located in *FPGA 2*.

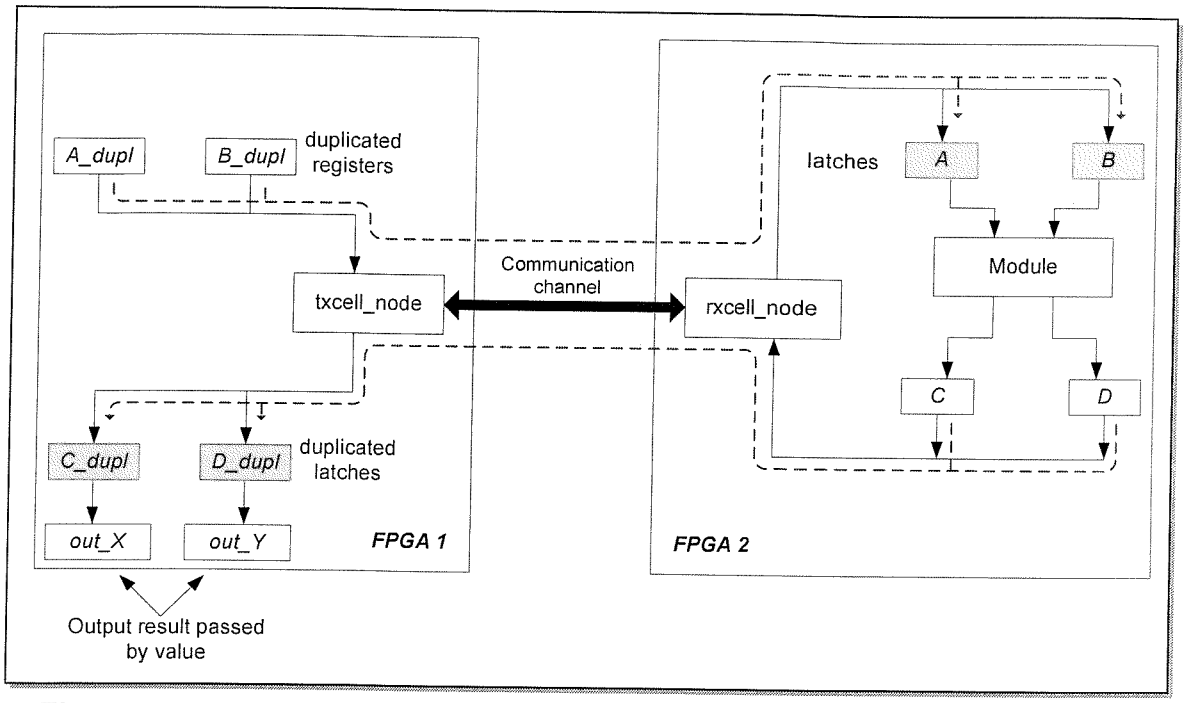


Figure 5-25 Latch and duplicated register arrangement for subprogram module I/O parameters across FPGA boundaries

The registers for the input parameters (*A* and *B*) of the called module in *FPGA 2* are replaced with latches (shaded in *FPGA 2*), and registers for the output parameters (*C* and *D*) are not bypassed, as they are needed to hold the valid results obtained from the subprogram execution for the rxcell_node to send the results back to the called module.

Duplicated registers (*A_dupl* and *B_dupl*) are generated and inserted into *FPGA 1* to hold the input parameters, which is sent to the called module by the corresponding txcell_node. Properties (data path unit bit-width, activation instructions, etc) for these duplicated registers are copied from the original set of registers (registers *A* and *B* in Figure 5-24 in this example). Similarly, duplicated latches (*C_dupl* and *D_dupl*) are generated and inserted into *FPGA 1* to latch in the result data packets put on the communication channel by the rxcell_node of the activated subprogram. At the end of the external module call, the

results in the duplicated latches are loaded into the appropriate output registers (*out_X* and *out_Y*).

5.6 Summary

The development of a multi-FPGA synthesis addition within the existing MOODS synthesis system has extended the MOODS synthesis system to perform optimisation and target multiple heterogeneous hardware devices, implementing a multi-FPGA system.

This chapter describes the asynchronous communication channel interface and the automatic generation and insertion of communication cells that form the building blocks of the subprogram communication channel, and inter-FPGA data transfers over asynchronous communication channels/links in a multi-FPGA system. The asynchronous data communication mechanism provided by the communication cells alleviates clock skew problems in a multi-FPGA system, as each re-configurable device is viewed as locally clocked processing units having an asynchronous communication interface.

The applicability of the two-phase partitioning algorithm and multi-FPGA synthesis and results of the multi-FPGA synthesis are demonstrated through a few design examples in the next chapter.

Chapter 6

Multi-FPGA implementation results

6.1 Introduction

The partitioning enhancement to the MOODS synthesis system provides the synthesis tool with an automated mechanism to partition a single behavioural design, which would not fit onto a single target device or which would be too costly to fit onto a target device with a large enough area capacity. With this partitioning add-on, the MOODS synthesis system can now target a single behavioural design onto two or more heterogeneous re-configurable devices (FPGAs) at the board level. One main objective in obtaining the partitions is to reduce interconnects (cutsets) and data transfers across boundaries. The K-way partitioning algorithm and communication subsystem optimisation algorithm described in Section 4.4.1 generates a partitioned design, with an optimised communication channel or multiple communication channels to improve the performance of the multi-FPGA system.

The two target technologies used for all the experiments in this section are given in Table 6-1. Target technologies listed in Table 6-1(a) show the Xilinx Spartan2 FPGAs, and Table 6-1(b) shows Xilinx Virtex FPGAs. The device parts in the target technology are listed in the first column of the tables and the package type of the device are given in the second column. The third column shows the total number of user I/Os available on the device. Four global clock pins on Spartan 2 or Virtex devices are usable as additional user I/Os when not used as global clock input pins. These pins are not included in the total user I/O counts given in the tables because these pins are normally connected to surface mounted clock oscillators or sockets for oscillator (e.g. global clock inputs GCK2 and GCK3 are connected to an on-board oscillator and a socket for a second oscillator in the D2-SB system board, see Appendix B.6). The fourth and fifth columns show the

maximum number of I/Os used and area of the devices in slices respectively. The information given in the fourth and fifth columns are used by the two-phase partitioning K-way algorithm (Section 4.4.1). The user I/O and area information of the target devices constitutes the target domain information of the input partitioning information (.par) file to the MOODS synthesis system. Detailed information on the two Xilinx target technologies are given in [140, 141].

(a) Xilinx Spartan 2 FPGA devices

Device	Package	Total user I/O	Max. user I/O	Max. area in slices
XC2S15	TQ144	86	80	768
XC2S30	TQ144	92	80	1200
XC2S50	FG256	176	150	1728
XC2S100	FG256	176	150	2352
XC2S150	FG456	260	250	3072
XC2S200	FG456	284	250	4800

(b) Xilinx Virtex FPGA devices

Device	Package	Total user I/O	Max. user I/O	Max. area in slices
XCV50	BG256	180	160	768
XCV100	BG256	180	160	1200
XCV150	BG352	260	250	1728
XCV200	BG352	260	250	2352
XCV300	BG432	316	300	3072
XCV400	BG432	316	300	4800
XCV600	BG560	404	400	6912
XCV800	BG560	404	400	9408
XCV1000	FG680	512	500	12288

Table 6-1 Target Xilinx FPGA technologies

Design examples are described using behavioural VHDL and synthesised using the MOODS synthesis system to generate un-partitioned and partitioned multi-FPGA implementations. The structural VHDL description files generated by MOODS are further processed by third party tools, Synplicity Synplify Pro and Xilinx ISE (Integrated Software Environment), which performs low-level logic synthesis and technology mapping. The Xilinx-targeted EDIF (Electronic Design Interchange Format) output from Synplify Pro is processed by Xilinx ISE to generate a single, or multiple, bitstream files to download onto a single, or multiple, FPGAs for an un-partitioned or a multi-FPGA design.

The first part of this chapter looks at experiments on subprogram communication channels in non-pipelined multi-FPGA systems (without explicit communication channels) in Section 6.2. The second part, Section 6.3, shows the inclusion of explicit communication channels and the overall performance of pipelined multi-FPGA systems.

6.2 Experimental results (without explicit communication channels)

This section contains the experimental results of five behavioural VHDL designs: (1) Quadratic equation solver (Quad eqs), (2) Cubic equation solver (Cubic eqs), (3) Inverse Discrete Cosine Transform (IDCT) module, (4) Triple-Data Encryption Standard (Triple-DES) core, (5) 256-bit Advanced Encryption Standard (AES256) core. The behavioural VHDL designs of all five examples and post-MOODS synthesis simulation results of the multi-FPGA implementations can be found in Appendix D.1. In this section, the non-pipelined multi-FPGA implementations of the VHDL examples are compared with the equivalent single-device implementation.

Synthesis result tables are given for each of the behavioural examples, where the first set of synthesis results are obtained from synthesised designs optimised in terms of area (i.e. with a high area optimisation priority) and the second set of results are obtained from synthesised designs optimised in terms of delay (i.e. with a high delay optimisation priority). Synthesis results for un-partitioned single device implementations using the original MOODS (before the partitioning enhancements were made) are shaded and given in the synthesis result tables. Subsequent rows list the multi-board FPGA implementations produced using various configuration of target FPGA devices. The * in the synthesis results denotes the implemented design or partition has exceeded either the maximum area, or the maximum number of user I/Os of the target device.

Column 1 of the synthesis results tables shows the number of targeted FPGA prototyping boards used to implement the synthesised design. Each FPGA board has a single Xilinx FPGA device, which is one of the devices in the target Xilinx FPGA technologies given in Table 6-1. For example, a s50 FPGA denotes a Xilinx Spartan 2 XC2S50 device in a FG256 package with a maximum user I/O of 150 pins and a maximum device area of 1728 slices, and a v200 FPGA denotes a Xilinx Virtex XCV200 device in a BG352 package with a maximum user I/O of 250 pins and a maximum device area of 2352 slices. The target FPGA devices are given in column 2 of the synthesis results tables. The MOODS synthesis optimisation priority (i.e. Area or Delay) is given in column 3.

The next four columns show the synthesis results of the implemented design after the Xilinx ISE placement and routing phase. These results show the final hardware implementation and not the MOODS synthesis estimate. Area (in slices), and I/O utilisation are given in columns 4 and 5 respectively. The maximum achievable frequencies (Freq) of the FPGAs are reported in column 6. Column 7 gives the area overhead (AO) of the multi-FPGA system with respect to the un-partitioned area-optimised or delay-optimised implementations (shaded row) of each example. The two-phase partitioning results are given in the last two columns of the synthesis result tables. Column 8 reports the initial number of inter-device data packet transfers and the final number of inter-device data packet transfers after the two-phase partitioning. Column 9 shows the number of explicit communication channels (*ExC*s) and subprogram communication channels (*SpC*s) inserted during synthesis with the data width of the channels in brackets.

6.2.1 Quadratic equation solver

The behavioural VHDL of the quadratic equation solver can be found in Appendix D.1.1. Figure 6-1 shows the module call graph representation of the quadratic equation solver, with a total of seven modules in the design.

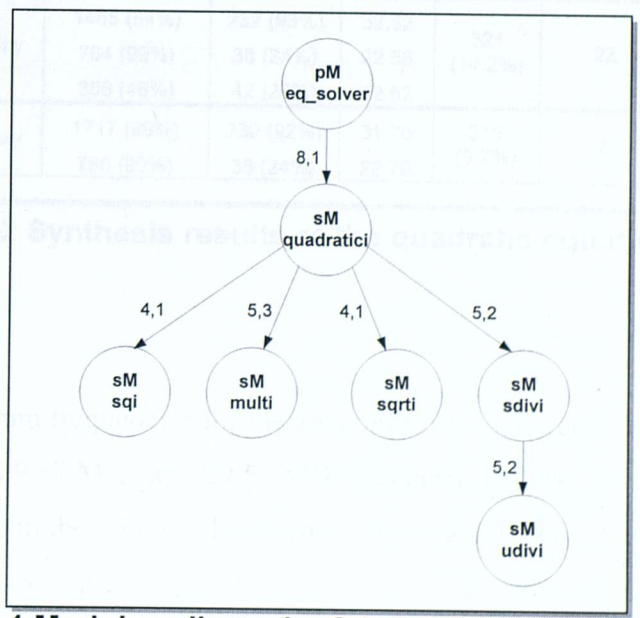


Figure 6-1 Module call graph of the quadratic equation solver

Synthesis results of the quadratic equation solver with high optimisation priority in area and delay are shown below in Table 6-2. These results in terms of area and maximum achievable frequency of the final implementation are obtained from the report files generated by post-Xilinx ISE placement and routing phase and not estimates obtained from the MOODS synthesis system. The Xilinx Virtex XCV200 (v200) is the smallest FPGA device in the targeted Xilinx Virtex technology, with sufficient area (in slices) for a single-chip implementation in both area and delay optimised quadratic equation solver examples.

Boards	FPGA	Synthesis results					Two-phase partitioning results	
		Opt. priority	Area in slices	I/O	Freq. (MHz)	AO in slices	Data pkts (initial → final)	Channels (Data widths)
1	v200	Area	2294 (97%)	194 (78%)	28.27	–	–	–
2	v150 v100	Area	1617 (93%) 865 (72%)	230 (92%) 38 (24%)	32.19 28.89	188 (8.2%)	29 → 6	1 *SpC* (32)
3	v150 v50 v50	Area	1483 (85%) 747 (97%) 355 (46%)	232 (93%) 38 (24%) 42 (26%)	33.66 23.51 40.58	291 (12.7%)	22 → 4	1 *SpC* (32)
2	v150 v50	Area	1726 (99%) 754 (98%)	230 (92%) 38 (24%)	30.30 22.69	186 (8.1%)	8 → 4	1 *SpC* (32)
1	v200	Delay	2264 (96%)	194 (78%)	28.43	–	–	–
2	v150 v100	Delay	1717 (99%) 771 (64%)	230 (92%) 38 (24%)	31.70 25.62	224 (9.9%)	29 → 2	1 *SpC* (32)
3	v150 v50 v50	Delay	1465 (84%) 764 (99%) 356 (46%)	232 (93%) 38 (24%) 42 (26%)	32.92 22.56 42.62	321 (14.2%)	22 → 2	1 *SpC* (32)
2	v150 v50	Delay	1717 (99%) 766 (99%)	230 (92%) 38 (24%)	31.70 22.70	219 (9.7%)	8 → 2	1 *SpC* (32)

Table 6-2 Synthesis results of the quadratic equation solver

The average maximum frequencies for the area optimised and delay optimised quadratic equation solver are 29.87 MHz and 29.52 MHz respectively. The least number of inter-device data transfers in the optimised implementations are 4 and 2 data packets in the area and delay optimised examples respectively. A single 32-bit subprogram communication channel (*SpC*) is inserted in all multi-FPGA implementation configurations. All the

constraint-satisfying partitioning solutions given in the table is found within 3 passes in the K-way partitioning algorithm in all cases.

It may appear strange that the area-optimised un-partitioned implementation has a larger area (in slices) than the delay-optimised un-partitioned implementation but a further look at the MOODS design space for both the area- and delay-optimised in Figure 6-2 shows that the MOODS estimation of the final implementation with a higher priority in delay in this case produced not only a synthesised design with a smaller delay, the area is also smaller than that of the final area-optimised implementation.

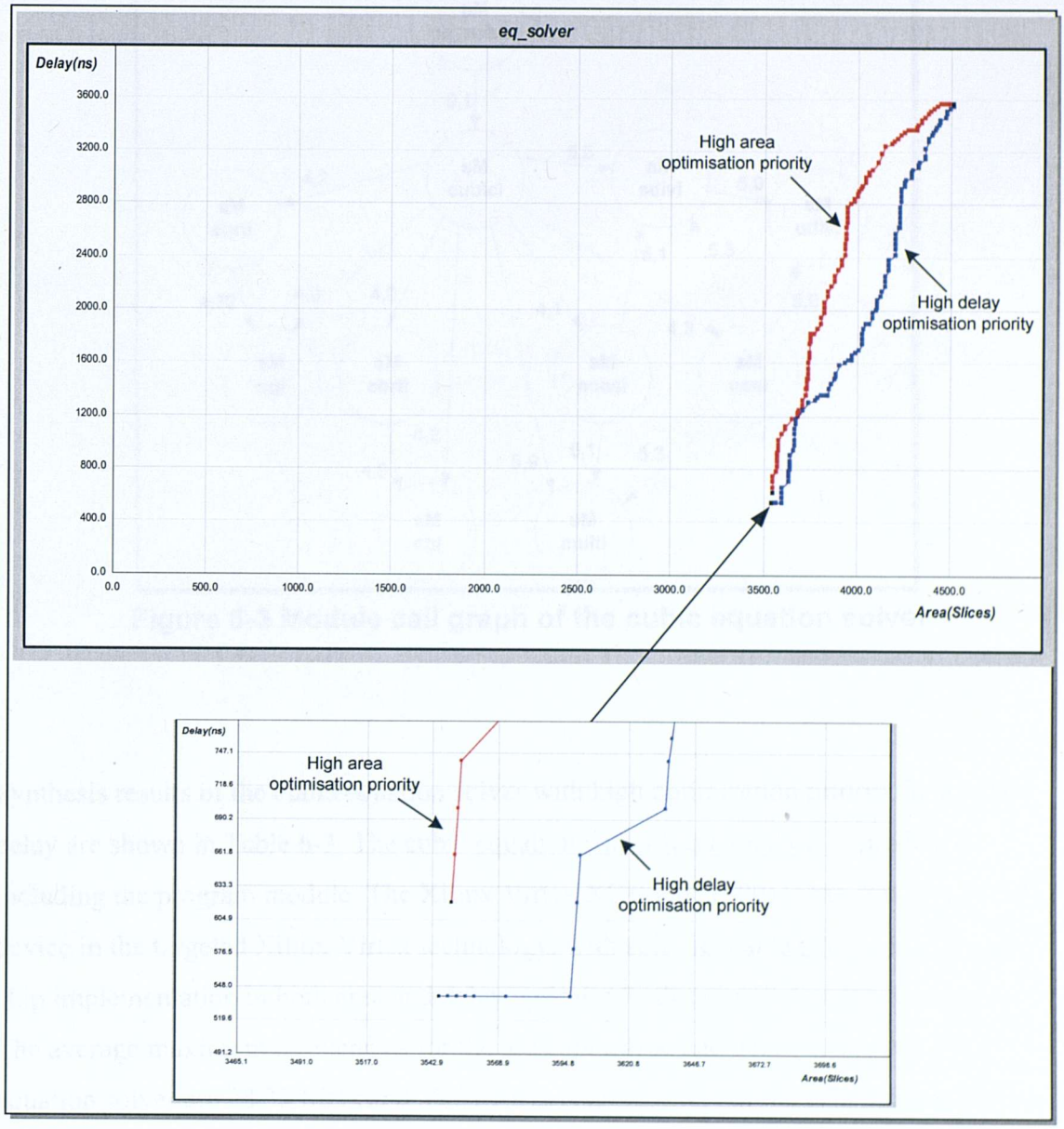


Figure 6-2 Design space of the un-partitioned quadratic equation solver

6.2.2 Cubic equation solver

The cubic equation solver uses similar VHDL subprograms as the quadratic equation solver but it is more complex. The module call graph representation of the cubic equation solver, with a total of 11 modules is given in Figure 6-3. The behavioural VHDL of the cubic equation solver can be found in Appendix D.1.2.

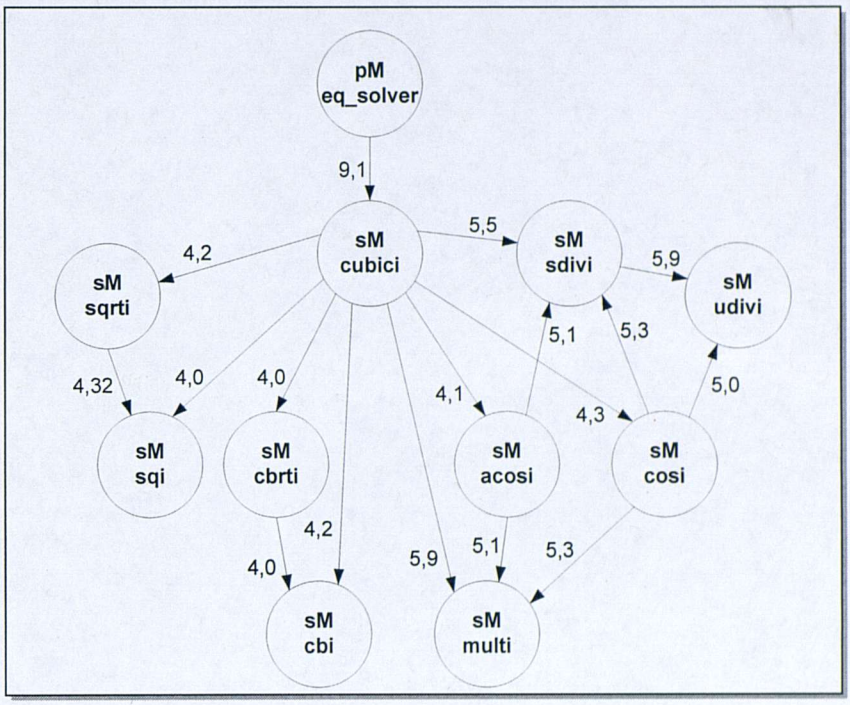


Figure 6-3 Module call graph of the cubic equation solver

Synthesis results of the cubic equation solver with high optimisation priority in area and delay are shown in Table 6-3. The cubic equation solver has a total of 11 modules including the program module. The Xilinx Virtex XCV400 (v400) is the smallest FPGA device in the targeted Xilinx Virtex technology, with sufficient area (in slices) for a single chip implementation in both area and delay optimised cubic equation solver examples. The average maximum frequencies for the area optimised and delay optimised cubic equation solver are 38.32 MHz and 32.51 MHz respectively. All the constraint-satisfying partitioning solutions given in the table is found within 3 passes in the K-way partitioning algorithm in all cases. The least number of inter-device data transfers in the optimised

implementations is 16 data packets and 2 optimised subprogram communication channels (*SpC*s) are inserted in all multi-FPGA implementation configurations. The maximum achievable frequency of the v300 device in the 4-board implementation, comprising one v300 and three v50 devices cannot be determined as the maximum device area utilisation has been exceeded.

Boards	FPGA	Synthesis results					Two-phase partitioning results	
		Opt. priority	Area in slices	I/O	Freq. (MHz)	AO in slices	Data pkts (initial → final)	Channels (Data widths)
1	v400	Area	3791 (78%)	226 (75%)	25.44	–	–	–
2	v300	Area	3070 (99%)	300(100%)	29.07	840 (22.2%)	100 → 36	2 *SpC* (32,24)
	v200		1561 (66%)	76 (25%)	36.53			
3	v300	Area	3070 (99%)	300(100%)	31.51	896 (23.6%)	100 → 36	2 *SpC* (32,24)
	v150		379 (21%)	38 (24%)	35.86			
	v150		1238 (71%)	78 (31%)	42.01			
4	v300	Area	3070 (99%)	300(100%)	32.86	866 (22.8%)	104 → 16	2 *SpC* (32,24)
	v100		379 (31%)	38 (24%)	35.17			
	v100		537 (44%)	58 (36%)	39.06			
	v100		671 (55%)	72 (45%)	57.28			
4	v300	Area	3070 (99%)	300(100%)	31.74	789 (20.8%)	40 → 16	2 *SpC* (32,24)
	v150		630 (36%)	56 (35%)	36.57			
	v50		406 (52%)	68 (43%)	57.90			
	v50		474 (61%)	40 (25%)	38.42			
4	v300	Area	3085* (101%)*	300(100%)	–	1096 (28.9%)	36 → 36	2 *SpC* (32,24)
	v50		567 (73%)	66 (41%)	39.90			
	v50		684 (89%)	70 (44%)	62.06			
	v50		551 (71%)	66 (41%)	38.95			
1	v400	Delay	3877 (80%)	226 (75%)	25.54	–	–	–
2	v300	Delay	3070 (99%)	294 (98%)	27.72	743 (19.2%)	36 → 36	2 *SpC* (32,18)
	v200		1550 (65%)	70 (28%)	24.54			
3	v300	Delay	3070 (100%)	300(100%)	31.29	804 (20.7%)	40 → 36	2 *SpC* (32,24)
	v150		624 (36%)	56 (35%)	34.58			
	v150		987 (46%)	72 (45%)	24.65			
4	v300	Delay	3070 (100%)	300(100%)	31.30	1024 (26.4%)	104 → 32	2 *SpC* (32,24)
	v100		393 (32%)	38 (24%)	34.62			
	v100		645 (53%)	68 (43%)	24.52			
	v100		793 (66%)	58 (36%)	40.25			
4	v300	Delay	3070 (100%)	300(100%)	24.51	786 (20.3%)	16 → 16	2 *SpC* (32,24)
	v150		868 (50%)	42 (26%)	34.19			
	v50		428 (55%)	68 (43%)	34.88			
	v50		297 (38%)	58 (36%)	61.83			
4	v300	Delay	3075* (101%)*	300(100%)	–	1030 (26.6%)	100 → 36	2 *SpC* (32,24)
	v50		633 (82%)	66 (41%)	25.21			
	v50		433 (56%)	66 (41%)	33.12			
	v50		766 (99%)	62 (39%)	42.04			

Table 6-3 Synthesis results of the cubic equation solver

6.2.3 Inverse discrete cosine transform

The inverse discrete cosine transform is a relatively simpler example with 3 modules (see module call graph representation in Figure 6-4) compared to the previous two design examples. The 2-D IDCT architecture is adapted from [142, 143] and the behavioural VHDL of the inverse discrete cosine transform can be found in Appendix D.1.3.

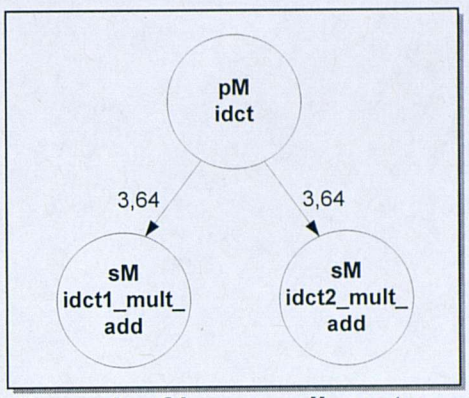


Figure 6-4 Module call graph of inverse discrete cosine transform example

Synthesis results of the inverse discrete cosine transform (IDCT) example with high optimisation priority in area and delay are shown in Table 6-4. The IDCT example has a total of 3 modules including the program module. The Xilinx Spartan 2 XC2S100 (s100) and XC2S150 (s150) are the smallest FPGA devices in the targeted Xilinx Spartan 2 technology, with sufficient area (in slices) for a single chip implementation in the area and delay optimised IDCT examples respectively. Place and route error (Par err) denotes incomplete low-level placement and routing of components by the Xilinx ISE (Integrated Software Environment) and the maximum frequencies of the post-placement and routed design are not given in such cases.

The average maximum frequencies for the area optimised and delay optimised IDCT examples are 30.02 MHz and 33.00 MHz respectively. The least number of inter-device data transfers in the optimised implementations are 128 and 192 data packets in the area and delay optimised IDCT modules respectively.

Boards	FPGA	Synthesis results					Two-phase partitioning results	
		Opt. priority	Area in slices	I/O	Freq. (MHz)	AO in slices	Data pkts (initial → final)	Channels (Data widths)
1	s100	Area	1018 (84%)	26 (17%)	28.27	–	–	–
2	s50 s50	Area	766 (99%) 527 (68%)	130 (92%) 121 (76%)	30.33 28.92	275 (27.0%)	320 → 128	1 *SpC* (91)
2	s30 s30	Area	652* (150%)* 430 (99%)	80 (100%) 56 (70%)	– 30.29	64 (6.3%)	320 → 128	1 *SpC* (91)
3	s30 s30 s30	Area	430 (85%) 430 (99%) 430 (99%)	80 (100%) 58 (73%) 54 (68%)	31.17 29.26 29.13	272 (26.7%)	320 → 320	1 *SpC* (91)
2	s50 s30	Area	766 (99%) 430 (99%)	104 (69%) 80 (100%)	30.33 30.85	178 (17.5%)	192 → 192	1 *SpC* (91)
1	s150	Delay	1476 (85%)	26 (10%)	28.43	–	–	–
2	s100 s100	Delay	1003 (83%) 865 (72%)	121 (81%) 97 (65%)	32.93 36.51	392 (26.6%)	192 → 192	1 *SpC* (91)
2	s50 s50	Delay	835* (108%)* 766 (99%)	121 (81%) 97 (65%)	– PAR err	125 (8.5%)	192 → 192	1 *SpC* (91)
3	s50 s50 s50	Delay	458 (59%) 766 (99%) 766 (99%)	123 (82%) 101 (76%) 97 (65%)	32.31 22.56 38.94	514 (34.8%)	320 → 320	1 *SpC* (91)
2	s100 s50	Delay	1002 (83%) 766 (99%)	123 (82%) 97 (65%)	33.56 PAR err	292 (19.8%)	192 → 192	1 *SpC* (91)

Table 6-4 Synthesis results of the inverse discrete cosine transform example

6.2.4 Triple-data encryption standard

The triple-data encryption standard core implements the triple data encryption algorithm (TDEA) in the electronic codebook (ECB) mode [144]. The behavioural VHDL of the triple-data encryption standard (triple-DES) can be found in Appendix D.1.4. Figure 6-5 shows the module call graph representation of the triple-DES, with a total of eleven modules in the design.

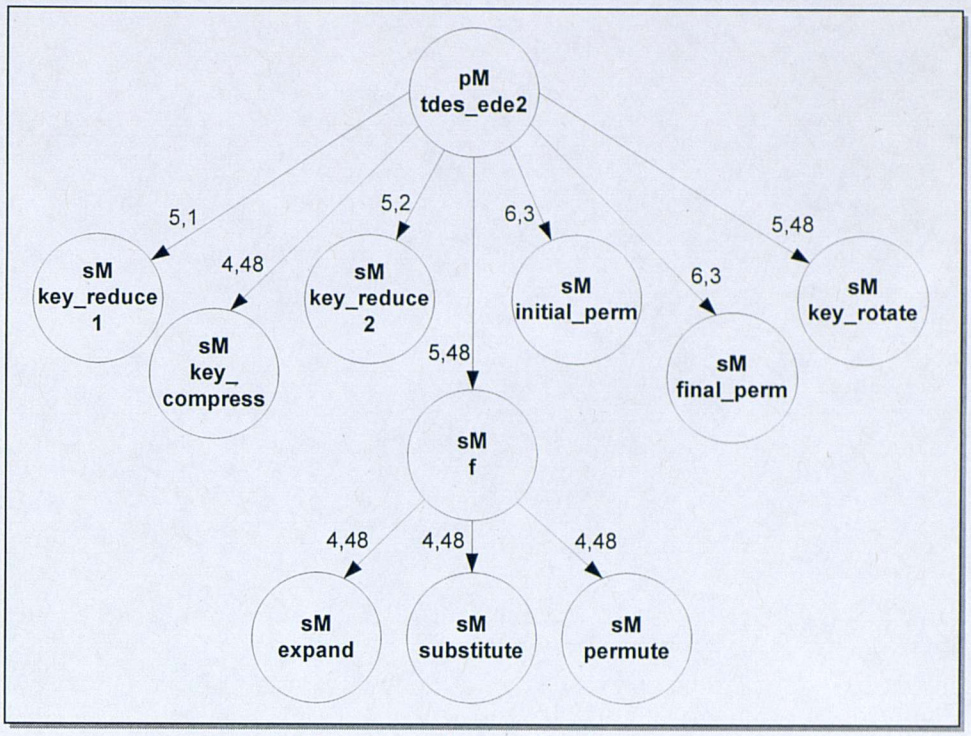


Figure 6-5 Module call graph of the triple-DES

Synthesis results of the triple-DES core with high optimisation priority in area and delay are shown in Table 6-5. The Xilinx Spartan 2 XC2S50 (s50) is the smallest FPGA device in the targeted Xilinx Spartan 2 technology, with sufficient area (in slices) for a single chip implementation in both area and delay optimised triple-DES core examples.

Boards	FPGA	Synthesis results					Two-phase partitioning results	
		Opt. priority	Area in slices	I/O	Freq. (MHz)	AO in slices	Data pkts (initial → final)	Channels (Data widths)
1	s50	Area	670 (87%)	99 (62%)	55.73	–	–	–
2	s30 s30	Area	633* (146%)* 430 (99%)	141*(176%)* 56 (70%)	– 30.29	393 (58.7%)	204 → 4	1 *SpC* (32)
2	s50 s30	Area	656 (85%) 380 (87%)	141 (94%) 40 (100%)	60.01 80.39	366 (54.6%)	204 → 4	1 *SpC* (32)
1	s50	Delay	670 (87%)	99 (62%)	59.03	–	–	–
2	s50 s30	Delay	746 (97%) 437* (101%)*	143 (95%) 42 (53%)	64.13 –	513 (76.6%)	396 → 204	1 *SpC* (32)
3	s50 s30 s30	Delay	655 (85%) 154 (35%) 256 (59%)	141 (94%) 42 (53%) 38 (48%)	63.86 69.47 56.36	395 (59.0%)	596 → 200	1 *SpC* (32)

Table 6-5 Synthesis results of the triple-DES core

The average maximum frequencies for the area optimised and delay optimised triple-DES core are 70.20 MHz and 63.23 MHz respectively. The least number of inter-device data transfers in the optimised implementations are 4 and 200 data packets in the area and delay optimised examples respectively. A single 32-bit subprogram communication channel (*SpC*) is inserted in all multi-FPGA implementation configurations.

6.2.5 256-bit advanced encryption standard

The 256-bit advanced encryption standard (AES) implements the Rijndael algorithm [145, 146], a symmetric block cipher that processes data blocks of 128 bits using a 256-bit cipher key. The algorithm is symmetric since the decryption algorithm is the exact reverse of the encryption algorithm. The 256-bit AES has a total of 5 modules. The module call graph representation is given in Figure 6-6. The behavioural VHDL of the quadratic equation solver can be found in Appendix D.1.5.

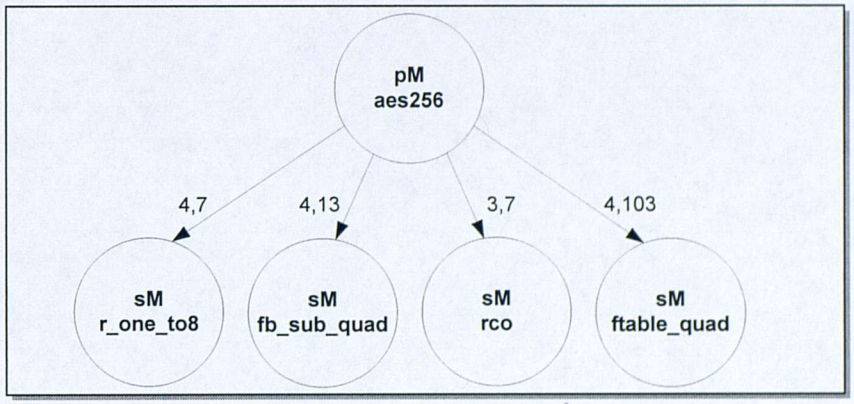


Figure 6-6 Module call graph of 256-bit advanced encryption standard

Synthesis results of the 256-bit AES core with high optimisation priority in area and delay are shown in Table 6-6. The 256-bit AES core has a total of 5 modules including the program module. The Xilinx Spartan 2 XC2S150 (s150) is the smallest FPGA device in the targeted Xilinx Spartan 2 technology, with sufficient area (in slices) for a single chip implementation in both area and delay optimised 256-bit AES core examples. The average

maximum frequencies for the area optimised and delay optimised 256-bit AES core are 42.60 MHz and 44.61 MHz respectively. The least number of inter-device data transfers in the optimised implementations are 28 and 232 data packets in the area and delay optimised examples respectively. A single 32-bit communication channel (*SpC*) is inserted in all multi-FPGA implementation configurations.

Boards	FPGA	Synthesis results					Two-phase partitioning results	
		Opt. priority	Area in slices	I/O	Freq. (MHz)	AO in slices	Data pkts (initial → final)	Channels (Data widths)
1	s150	Area	1445 (83%)	102 (68%)	36.85	–	–	–
2	s100 s50	Area	1138 (94%) 766 (99%)	144 (96%) 44 (29%)	39.79 45.41	459 (31.8%)	260 → 260	1 *SpC* (32)
2	s100 s15	Area	1459* (121%)* 140 (32%)	140 (93%) 40 (50%)	– 80.95	154 (10.7%)	54 → 28	1 *SpC* (32)
3	s100 s30 s30 s30	Area	1198 (99%) 152 (35%) 468* (108%)* 204 (47%)	144 (96%) 48 (60%) 38 (48%) 38 (48%)	40.54 79.48 – 56.10	577 (39.9%)	260 → 260	1 *SpC* (32)
1	s150	Delay	1476 (85%)	102 (68%)	39.43	–	–	–
2	s100 s50	Delay	1181 (98%) 759 (98%)	144 (96%) 44 (29%)	40.00 44.60	464 (31.4%)	260 → 260	1 *SpC* (32)
2	s100 s30	Delay	1060 (88%) 607* (140%)*	140 (93%) 40 (50%)	40.46 –	191 (12.9%)	246 → 232	1 *SpC* (32)
3	s100 s30 s30	Delay	1130 (94%) 271 (62%) 430 (99%)	142 (95%) 44 (55%) 38 (48%)	40.02 55.54 45.17	355 (24.1%)	260 → 246	1 *SpC* (32)

Table 6-6 Synthesis results of the 256-bit AES core

6.2.6 Discussion of results

The area overheads of the multi-FPGA implementations (MFIs) of the VHDL examples are due to various factors. The first and also the main factor that contributes most to the area overheads is the generation and inclusion of communication cells and arbiters, which are the building blocks of the communication subsystem. The other reason is the duplication of registers (or creation of latches) for the I/O parameters of external (cross boundary) subprogram modules (Section 5.5.1).

Figure 6-7 shows the area and I/O utilisation of the devices in different multi-board configurations for all 5 design examples listed in Table 6-2 to Table 6-6. All twenty-five configurations satisfy the target device constraints (in terms of device area in slices and I/O pins available) for all partitions in the MFI and the partitions are successfully mapped to their target devices.

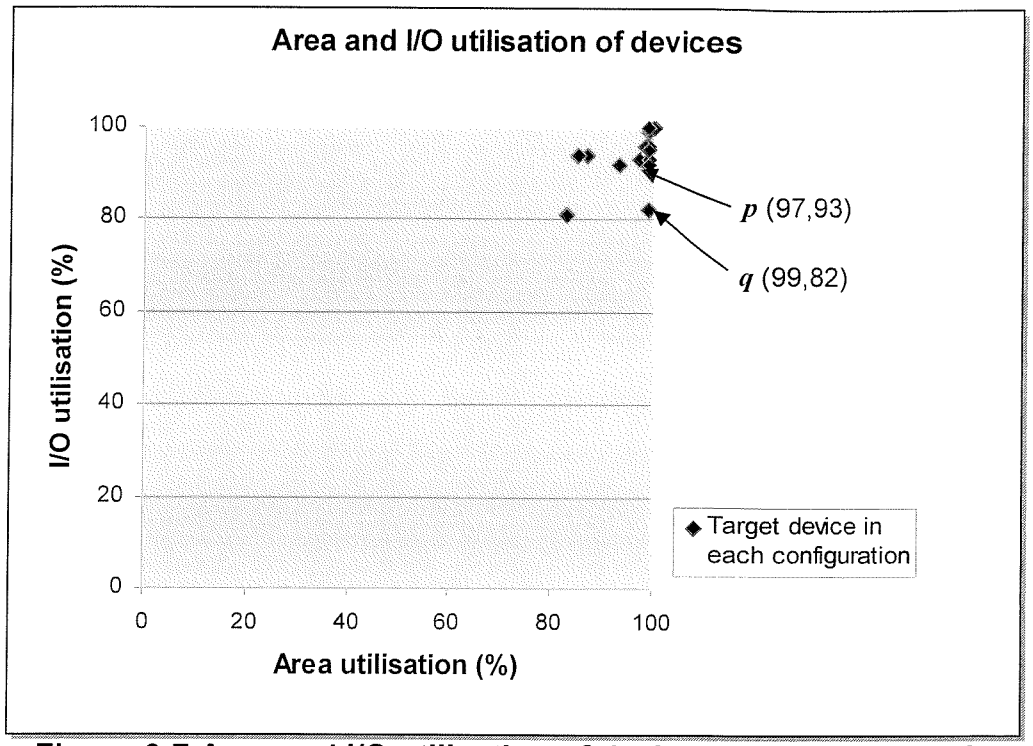


Figure 6-7 Area and I/O utilisation of devices in example designs

Each plotted value in Figure 6-7 gives the maximum area and I/O utilisation (area, I/O) amongst the devices in each of the configuration. Value ***p*** gives the maximum area and I/O utilisation of the 3-board MFI of the quadratic equation solver with high priority in area optimisation in Table 6-2 consisting of one v150 and two v50 devices, where the v150 gives the highest I/O utilisation at 93% and the first v50 device gives the highest device area utilisation at 97%. Value ***q*** gives the maximum area and I/O utilisation of the 3-board MFI of the delay-optimised IDCT example in Table 6-5 consisting of three s50 devices, where the first s50 device gives the highest device I/O utilisation at 82% and the other two s50 devices give the highest device area utilisation at 99%. The two-phase partitioning algorithm produces a high area and I/O utilisation of the FPGA devices in all

configurations of the five design examples, achieving over 90% in both area and I/O utilisations in at least one device in most cases.

Example	Inter-device data packets	Clock cycles		Freq (MHz)		Design latency (μs)	
		un-partitioned	partitioned	un-partitioned	partitioned	un-partitioned	partitioned
Quad_eqs	4	179	224	28.27	28.66	6.33	7.82
Cubic_eqs	36	755	1770	25.44	26.13	29.68	67.74
IDCT	192	831	4175	30.34	34.72	27.39	120.25
Triple-DES	200	524	3950	55.73	63.23	9.40	62.47
AES-256	246	814	5257	36.85	42.30	22.09	124.28

Table 6-7 Performance of example designs

The performances of example designs and the overheads (in terms of clock cycles) in multi-FPGA implementations are given in Table 6-7 above. The number of clock cycles given in the table gives the total number of clock cycles it takes to complete the application (e.g. the number of clock cycles for the equation solvers is calculated from the first clock cycle when the input data is received to the last clock cycle when the last output data is obtained). The increase in design latencies are mainly due to the setting up of the shared tri-state subprogram communication channels and synchronisation of the data packets during the inter-clock domain asynchronous data transfers. The tri-state data bus and data handshake signals allow I/O resource sharing between modules in different target devices. A point-to-point (PTP) unidirectional communication channel implementation requires a simpler circuitry, with possibly smaller overheads to send and receive inter-device data. Experiments in the next section look at the effects of point-to-point explicit communication channels together with subprogram communication channels in optimised multi-FPGA configurations.

6.3 Experimental results (with explicit communication channels)

This section contains experiments of three VHDL examples used in the previous section. The quadratic equation solver (Quad_eqs), inverse discrete cosine transform (IDCT) module and 256-bit Advanced Encryption Standard (AES256) core are modified slightly to include explicit communication channels (Section 4.2.2.1). Explicit communication channels are used to synchronise and transfer global VHDL signal data between VHDL processes. The three VHDL examples are re-written and pipelined to include explicit communication channels. The behavioural VHDL designs of all three pipelined examples and the Post-MOODS synthesis simulation results of the multi-FPGA implementations can be found in Appendix D.2.

6.3.1 Pipelined quadratic equation solver

The pipelined quadratic equation solver is a two-stage pipelined version of the quadratic equation solver given in Section 6.2.1. The behavioural VHDL of the pipelined quadratic equation solver can be found in Appendix D.2.1. Figure 6-8 shows the module call graph representation of the pipelined quadratic equation solver, with two process modules and five subprogram modules in the design.

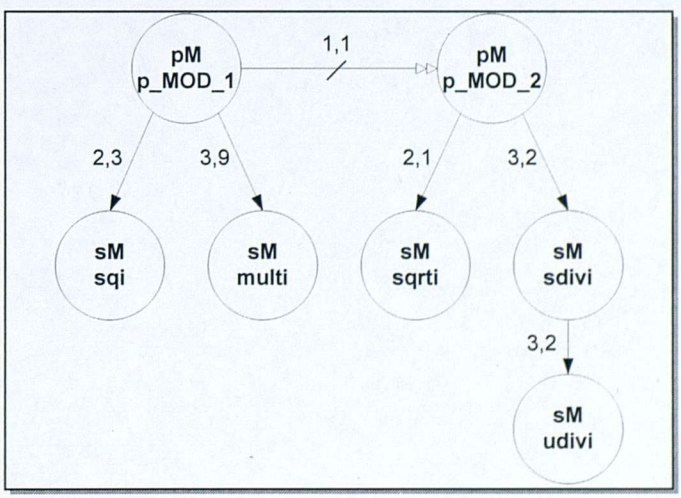


Figure 6-8 Module call graph of the pipelined quadratic equation solver

Process modules p_MOD_1 and p_MOD_2 are connected by an explicit communication channel (*ExC*) with a data width of 96-bits. The multi-FPGA pipelined quadratic equation solver implementation not only resulted in a lower area overhead for area and delay optimised implementations (7.4% and 8.9% respectively) compared to the results without explicit communication channels (8.2% and 9.7% respectively) given in Table 6-2, the number of inter-device data packet transfers is reduced to just the data sent across the pipelined stage through the explicit communication channel.

Boards	FPGA	Synthesis results					Two-phase partitioning results	
		Opt. priority	Area in slices	I/O	Freq. (MHz)	AO in slices	Data pkts (initial → final)	Channels (Data widths)
1	v200	Area	2294 (97%)	194 (78%)	28.27	–	–	–
2	v150 v150	Area	1726 (99%) 738 (42%)	196 (78%) 196 (78%)	32.33 34.03	170 (7.4%)	34 → 1	1 *ExC* (96)
2	v150 v100	Area	1726 (99%) 738 (61%)	196 (78%) 196* (123%)*	32.33 –	170 (7.4%)	34 → 1	1 *ExC* (96)
1	v200	Delay	2264 (96%)	194 (78%)	28.43	–	–	–
2	v150 v150	Delay	1726 (99%) 739 (42%)	196 (78%) 196 (78%)	29.07 34.21	201 (8.9%)	34 → 1	1 *ExC* (96)
2	v150 v100	Delay	1726 (99%) 738 (61%)	196 (78%) 196* (123%)*	29.07 –	200 (8.8%)	34 → 1	1 *ExC* (96)

Table 6-8 Synthesis results of the pipelined quadratic equation solver

The six 32-bit input and output signals in the VHDL entity port list declaration of the pipelined quadratic equation solver are grouped and mapped to process modules that access these signal, distributing the utilisation of I/O resources over two or more devices. This alleviates the problem of a single device in the multi-FPGA implementation exceeding the maximum number of usable I/Os whilst the I/O resources of other devices are under-utilised. Without this capability to distribute the signals in the VHDL entity port list declaration, a larger target device such as a Xilinx XCV300 with 300 usable I/O pins (see Table 6-1) has to be one of the targeted devices since a minimum I/O utilisation of 292 I/O pins (i.e. 194 pins for the signals VHDL entity port list declaration and 98 pins for the explicit communication channel with two semaphore signals and 96-bit data width) is needed.

discrete cosine transform (IDCT) core given in Section 6.2.3. The behavioural VHDL of the pipelined IDCT core can be found in Appendix D.2.2. Figure 6-9 shows the four modules in the module call graph representation of the pipelined IDCT core.

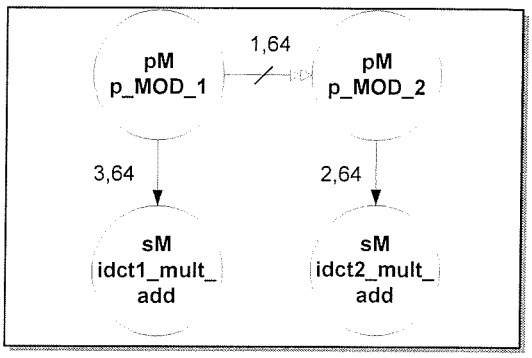


Figure 6-9 Module call graph of the pipelined inverse discrete cosine transform example

Boards	FPGA	Synthesis results					Two-phase partitioning results	
		Opt. priority	Area in slices	I/O	Freq. (MHz)	AO in slices	Data pkts (initial → final)	Channels (Data widths)
1	s100	Area	1018 (84%)	26 (17%)	28.27	–	–	–
2	s50	Area	511 (66%)	29 (19%)	29.22	129 (12.7%)	1 → 1	1 *ExC*
	s50		636 (82%)	25 (17%)	31.67			(11)
2	s50	Area	766 (99%)	104 (69%)	31.15	178 (17.5%)	1 → 1	1 *SpC*
	s30		430 (99%)	80 (100%)	28.72			(74)
3	s30	Area	447* (103%)*	29 (19%)	–	180 (17.7%)	3 → 3	1 *ExC* (11)
	s30		430 (99%)	57 (71%)	29.64			1 *SpC* (51)
	s30		321 (37%)	80 (100%)	35.11			
1	s150	Delay	1476 (85%)	26 (10%)	28.43	–	–	–
2	s100	Delay	754 (62%)	29 (19%)	34.63	46 (3.1%)	1 → 1	1 *ExC*
	s100		768 (64%)	25 (17%)	37.31			(11)
2	s50	Delay	754 (98%)	29 (19%)	31.82	44 (3.0%)	1 → 1	1 *ExC*
	s50		766 (99%)	25 (65%)	36.87			(11)
3	s50	Delay	754 (98%)	29 (19%)	32.97	455 (30.8%)	3 → 3	1 *ExC* (11)
	s50		766 (99%)	97 (65%)	38.93			1 *SpC* (91)
	s50		411 (53%)	120 (80%)	32.78			

Table 6-9 Synthesis results of the pipelined inverse discrete cosine transform example

Process modules p_MOD_1 and p_MOD_2 are connected by an explicit communication channel (*ExC*) with a data width of 11-bits. The multi-FPGA pipelined IDCT core implementation not only resulted in a lower area overhead for area and delay optimised implementations (12.7% and 3.0% respectively) compared to the results without explicit communication channels (27.0% and 26.6% respectively) given in Table 6-4, the number of inter-device data packet transfers is reduced to just the data sent across the pipelined stage through the explicit communication channel for the 2-board implementations. A single subprogram communication channel is generated for both area and delay optimised multi-FPGA implementations targeting three devices. The maximum I/O pin utilisation for one device is reduced to 29 pins for all two-board pipelined implementations with just an explicit communication channel in Table 6-9 compared to over 100 pins in the non-pipelined implementation (given in Table 6-4).

6.3.3 Pipelined 256-bit advanced encryption standard

The last pipelined VHDL example is a two-stage pipelined version of the 256-bit advanced encryption standard (AES) core given in Section 6.2.5. The behavioural VHDL of the pipelined 256-bit AES core can be found in Appendix D.2.3. Figure 6-10 shows two process modules and five subprogram modules in the module call graph representation of the pipelined 256-bit AES core.

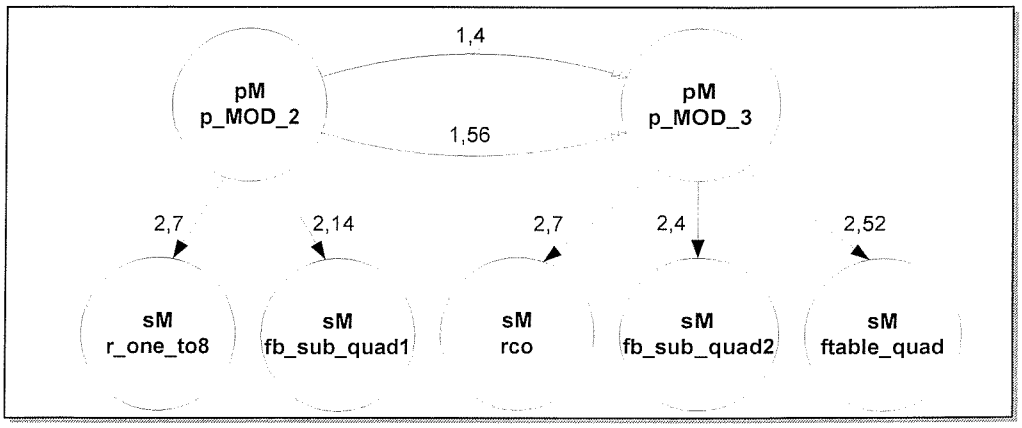


Figure 6-10 Module call graph of the pipelined 256-bit advanced encryption standard example

The two process modules (p_MOD_2 and p_MOD_3) in the pipelined 256-bit AES core are connected through two explicit communication channels with data widths of 32-bits. The number of inter-device data packet transfers is reduced to just the data sent across the pipelined stage through the two explicit communication channels. The two-board pipelined implementations given in Table 6-10 have an area overhead reduction of 10.0% for an area optimised implementation and 15.0% for a delay optimised implementation compared to the non-pipelined implementations given in Table 6-6.

Boards	FPGA	Synthesis results					Two-phase partitioning results	
		Opt. priority	Area in slices	I/O	Freq. (MHz)	AO in slices	Data pkts (initial → final)	Channels (Data widths)
1	s150	Area	1445 (83%)	102 (68%)	36.85	–	–	–
2	s100	Area	994 (82%)	104 (69%)	43.78	315 (21.8%)	1 → 1	2 *ExC*
	s50		766 (99%)	136 (91%)	34.76			(32,32)
3	s100	Area	1283* (106%)*	146 (97%)	–	577 (57.6%)	168 → 84	1 *SpC* (32)
	s30		625* (144%)*	40 (27%)	–			
	s30		203 (46%)	38 (48%)	56.75			
	s30		166 (38%)	52 (65%)	80.30			
1	s150	Delay	1476 (85%)	102 (68%)	39.43	–	–	–
2	s100	Delay	955 (79%)	104 (69%)	41.99	242 (16.4%)	1 → 1	2 *ExC*
	s50		763 (99%)	136 (91%)	34.47			(32,32)
3	s100	Delay	1198 (99%)	144 (96%)	34.63	611 (41.4%)	168 → 77	1 *SpC* (32)
	s30		74 (17%)	38 (48%)	70.46			
	s30		581* (134%)*	40 (27%)	–			
	s30		234 (54%)	50 (63%)	57.05			

Table 6-10 Synthesis results of the pipelined 256-bit AES core

6.3.4 Discussion of results

The performances of the pipelined example designs and the overheads (in terms of clock cycles) in multi-FPGA implementations (MFIs) are given in Table 6-11. The number of clock cycles given in the table gives the total number of clock cycles it takes to complete the application (i.e. the number of clock cycles is calculated from the first clock cycle

when the input data is received to the last clock cycle when the last output data is obtained). Design latency is calculated by multiplying the number of clock cycles by the clock period (1/Freq) of the design. Performance results of pipelined example designs are shaded in the table and results of the un-partitioned and non-pipelined taken from Table 6-7 are shown for completeness and ease of comparison.

When pipeline stages are targeted onto a separate device, inter-device data packets are sent via the explicit communication channels (*ExC*s) connecting the pipelined stages in the pipelined MFI. The *ExC* is a dedicated point-to-point communication channel that does not require channel resource arbitration and special communication cells (Section 5.4) to handle inter-device data packet transfers unlike the *SpC*. Inter-device data sent through the *ExC* also removes the need for *hardware duplication* (Section 5.5.1), hence reducing the area overheads. The inter-device data packets are reduced by at least a factor of three compared to the non-pipelined MFI. As a result, the pipelined IDCT and AES-256 MFIs only suffer a fraction of the design latency overhead compared to the non-pipelined MFIs.

Example	Inter-device data packets		Clock cycles			Freq (MHz)			Design latency (µs)		
	Partitioned (non-pipelined)	Partitioned (pipelined)	un-partitioned	Partitioned (non-pipelined)	Partitioned (pipelined)	un-partitioned	Partitioned (non-pipelined)	Partitioned (pipelined)	un-partitioned	Partitioned (non-pipelined)	Partitioned (pipelined)
Quad_eqs	4	1	179	224	189	28.27	28.66	31.64	6.33	7.82	5.97
IDCT	192	64	831	4175	1167	30.34	34.72	34.35	27.39	120.25	34.98
AES-256	246	60	814	5257	1137	36.85	42.30	38.23	22.09	124.28	29.74

Table 6-11 Performance of the pipelined example designs

Another advantage is the higher average achievable frequencies of the pipelined implementations of all three example designs compared to the un-partitioned implementations. In the case of the pipelined quadratic equation solver implementation, there is only a single inter-device data packet transfer in the pipelined MFI and it only

takes 10 clock cycles more than the un-partitioned implementation. With the higher achievable frequency, the pipelined quadratic equation solver has a lower design latency compared to both the un-partitioned and non-pipelined implementations.

The experiments in this section show that pipelined multi-FPGA systems can achieve performances comparable to single-device implementations. The user now has the choice of targeting a large behavioural design onto multiple smaller devices without having the need to get a larger and more costly target FPGA device if the design requirements are met with a multi-FPGA system. The user would be able to use existing FPGA devices or a number of FPGA development boards configured into a multi-FPGA system for design prototyping. This saving in design cost and flexibility in using existing development boards with a collection of smaller devices would not be possible otherwise if a single large behavioural design is not partitioned.

The asynchronous communication channels provide safe communication of inter-device data in the multi-FPGA system. The subprogram communication channel (*SpC*) allows multiple external subprogram modules to share a common channel, hence reducing the number of I/Os needed for inter-device data transfers in the I/O constraint multi-FPGA system. The explicit communication channel (*ExC*) itself is responsible for the synchronisation of VHDL processes (process modules) connected to it and the transfer of data between the process modules. Therefore, the user can concentrate on the behaviour of the design and not the complexities of how the target devices can safely communicate.

All experiments were run on a Intel Pentium M 1.5 GHz machine with 512 MB RAM. The multi-FPGA synthesis run times remain similar to the run times of single-device implementations using an original version of MOODS without the multi-FPGA synthesis enhancements as final partition solutions for all the VHDL examples are found within 3 passes of the modified K-way partitioning algorithm (Section 4.4.1). Run times are approximates due to the nature of Microsoft windows environment which the synthesis tool is running in (e.g. synthesis run times for the pipelined multi-FPGA 256-bit AES core example is 2 minutes and the single-device implementation is 1 minute and 56 seconds).

6.4 Summary

The applicability of the two-phase partitioning algorithm and multi-FPGA synthesis and results of the multi-FPGA synthesis are demonstrated through a few design examples in this chapter. Results of non-pipelined and pipelined multi-FPGA implementations are given in Sections 6.2 and 6.3 respectively. The pipelined multi-FPGA systems with explicit communication channels have reductions in area overheads (by up to 23% in the case of the pipelined IDCT example) and design latencies (up to 4 times faster) over non-pipelined multi-FPGA implementations. Results presented in Section 6.3 show that pipelined multi-FPGA systems can be synthesised to achieve performances comparable to single-device implementations.

Partitioning of a large behavioural design with incomplete knowledge of the targeted technology, and the final hardware implementation of a component poses a difficult design decision. As the complexity and size of the entire system increases, this difficult decision and design optimisation problem gets harder, to the point when it gets beyond the capabilities of human designers to solve. High-level synthesis of a large behavioural design into a multi-FPGA system reduces the design time and effort required by the user.

This chapter has demonstrated the automated process of multi-FPGA synthesis in MOODS to produce multi-FPGA systems with asynchronous communication channels (described in Chapters 4 and 5). All the multi-FPGA implementations given in this chapter have been synthesised, simulated, and proven correct, with comparisons to single-device implementations using an original version of MOODS without the multi-FPGA synthesis enhancements.

A further analysis of the performance of a multi-FPGA system is demonstrated through a hardware demonstrator of a large behavioural design in Chapter 7.

Chapter 7

Practical synthesis

7.1 Introduction

This chapter describes in detail the hardware demonstrator, a JPEG decoder synthesised using MOODS with the multi-FPGA synthesis enhancement. The goal of the hardware demonstrator is to assess the performance of a multi-FPGA JPEG decoder, as opposed to a JPEG decoder in a single chip implementation. Section 7.2 provides information on the hardware development boards used to implement the multi-FPGA system. Section 7.3 gives an introduction on the JPEG algorithm, and outlines the multi-FPGA implementation of the JPEG decoder. The implementation details of the partitioned hardware JPEG decoder targeting a multi-FPGA system are covered in the last section (Section 7.4). Section 7.4 discusses the performance of the non-pipelined multi-FPGA JPEG decoder, with a further analysis on the implementation results. It also covers a pipelined implementation of the multi-FPGA JPEG decoder using explicit communication channels.

7.2 FPGA-based development board

The multi-FPGA system created to demonstrate the multi-FPGA partitioning enhancement of MOODS is realised with the use of a number of FPGA-based development boards. This section starts with a brief insight on the development boards used to form the multi-FPGA system and the Input/Output VGA extension board built for this hardware demonstrator.

7.2.1 Hardware development board

The JPEG decoder synthesised using the MOODS synthesis system into a multi-FPGA system is targeted onto the Digilent D2-SB FPGA-based development boards [147]. The development board provides a complete and expandable development platform for hardware prototyping purposes. The D2-SB features a Xilinx Spartan 2E-200 FPGA in a PQ208 package that has gate capacity of 200,000 and over 200 MHz operation. The D2-SB provides a total of 143 user I/Os routed to six standard 40-pin connectors, and it contains a surface-mount 50MHz oscillator, and a socket for a second oscillator. The D2-SB has a JTAG port (see Appendix B.6) and it is used to program the Spartan 2E FPGA and the 18V02 configuration flash ROM, and any programmable devices on peripheral boards attached to the D2-SB development board.

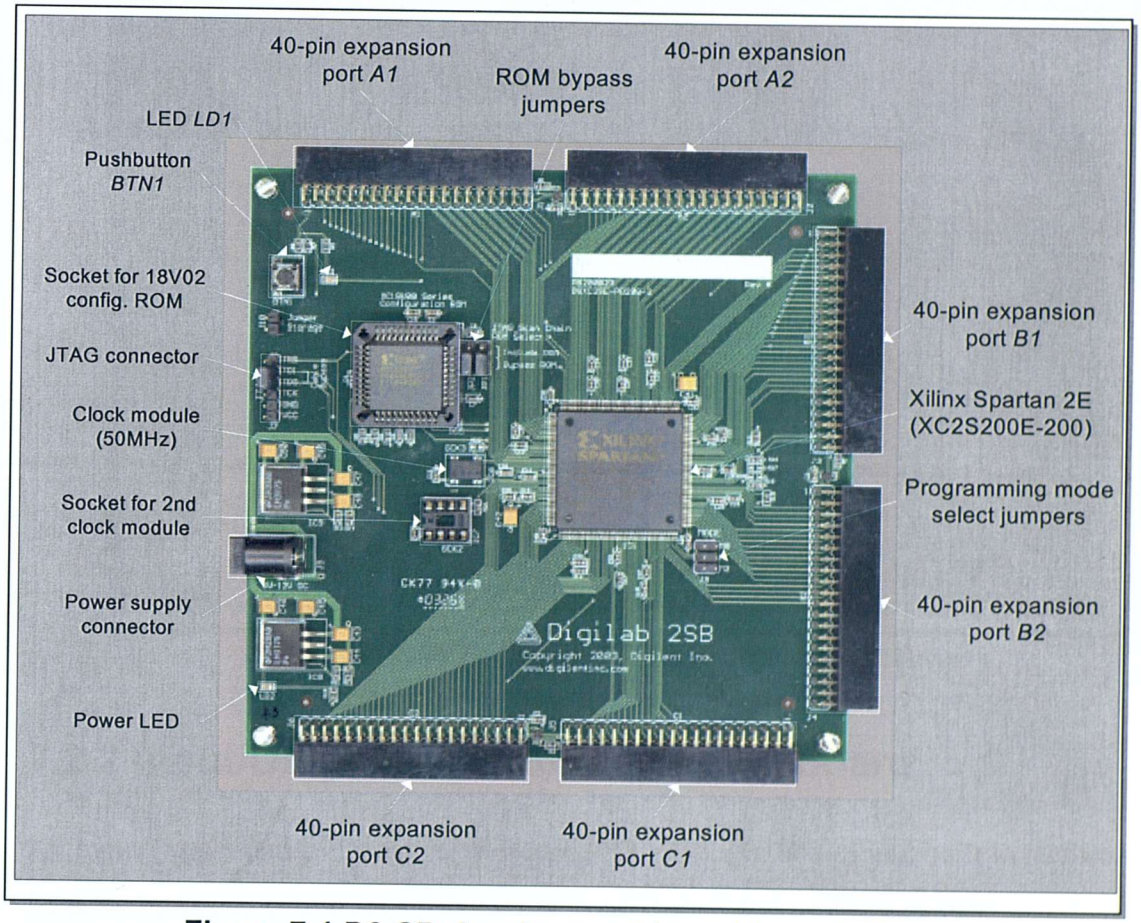


Figure 7-1 D2-SB development board layout picture

Figure 7-1 shows a picture of the physical board, with various devices and interfaces highlighted. Detailed description of the D2-SB development board can be found in Appendix B.6.

The Digilent DIO4 peripheral board [148] provides a fast and easy way to add several useful I/O devices to the D2-SB development board. The DIO4 provides a 4-digit seven-segment LED display, 8 individual LEDs, 5 pushbuttons with debouncing circuitry, 8 slide switches, a 3-bit VGA port, and a PS2 port. The DIO4 draws power from the main system board (i.e. the D2-SB development board), and signals from the various I/O devices are routed to individual pins on the system board connectors. A picture of the physical board layout of the DIO4 peripheral board is given in Figure 7-2, with various devices and interfaces highlighted. Detailed description of the DIO4 peripheral board can be found in Appendix B.7.

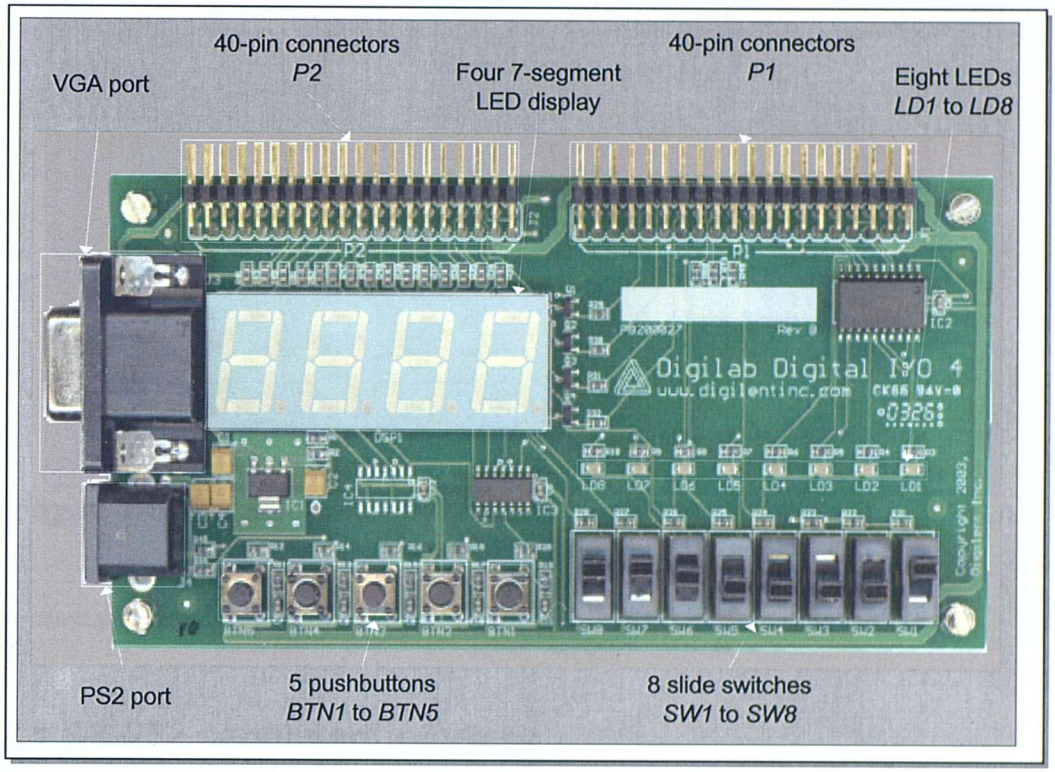


Figure 7-2 DIO4 digital I/O board layout picture

7.2.2 Input/Output and VGA extension board

An Input/Output and VGA extension board (I/O VGA ext. board) was built to facilitate both the input and output stages in the hardware demonstrator. The key components on the extension board include a serial (RS-232) port interface, a 4Mbyte Asynchronous SRAM (256K x 16 CMOS 15 ns) DIL module and a video Digital-to-Analogue Converter (videoDAC). The extension board was designed to be a general plug-in daughter board, which can be used with any other hardware development board to provide a serial port communication interface, four megabyte of fast (15 ns access time), asynchronous SRAM

memory, and a triple 8-bit videoDAC (BT121) to drive a VGA monitor. Figure 7-3 shows the key components and their corresponding locations on the top side of the I/O VGA ext. board.

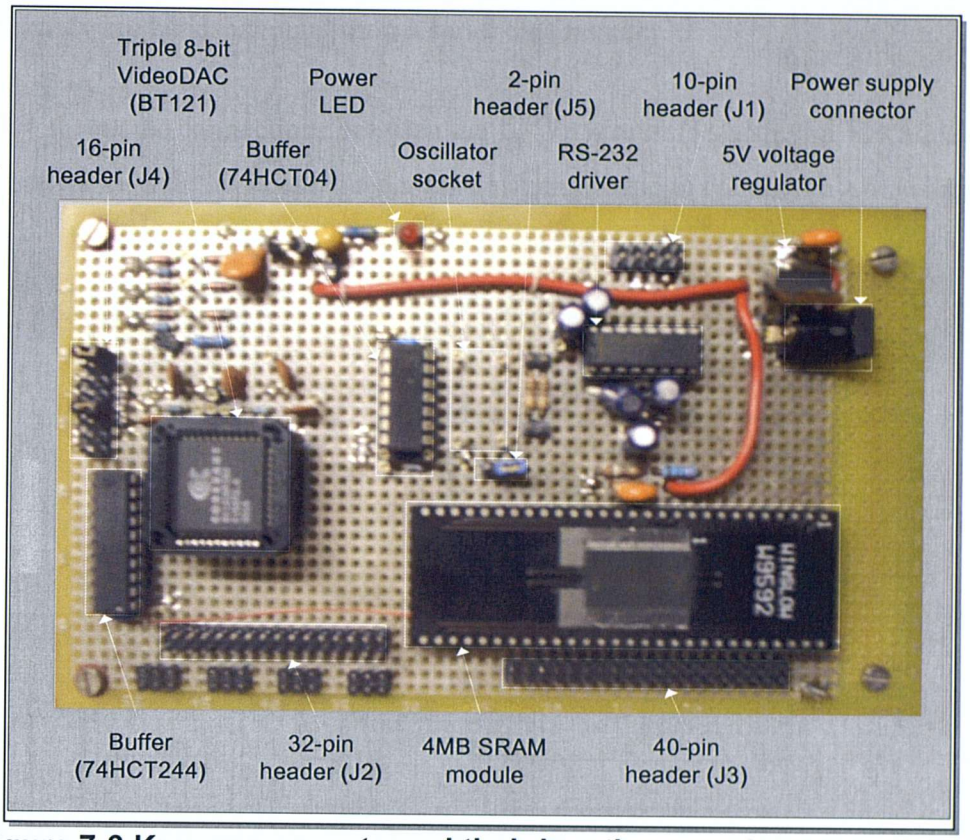


Figure 7-3 Key components and their locations on the I/O and VGA extension board

7.2.2.1 RS-232 serial port interface

The I/O VGA ext. board has a RS-232 serial port [149] interface that allows the extension board to connect a PC's serial port. The Maxim MAX232EPE RS-232 voltage converter takes serial data as TTL/CMOS levels from a connected development board via pin 29 of the 32-pin and coverts the logic level to the appropriate RS-232 voltage level and this is sent to a connected device via pin 2 of the 10-pin header (J1) located next to the Maxim device. Likewise, the Maxim device converts the RS-232 serial input data to TTL/CMOS levels and sends this to the development board via pin 30 of the 32-pin header (J2). A DB9 serial port connector can be connected to the 10-pin header next to the Maxim device and a standard-through or null-modem serial cable can be used to connect the I/O VGA ext. board to the PC's serial port. Two 100-ohm series resistors between the Maxim output

pins (pins 9 and 12) and pins 28 and 30 of header J3 protects against accidental logic conflicts. Note that the Xilinx Spartan 2E-200 FPGA on the Digilent D2-SB development boards are not 5 volts tolerant, the series resistors are necessary in this case. Control signals, *Request To Send* (RTS) and *Clear To Send* (CTS) are connected to header J2 to facilitate hardware handshaking during a serial data transfer.

Figure 7-4 shows the connections between the 10-pin header (J1), Maxim MAX232EPE RS-232 converter, and the 32-pin header (J3), which is used to connect to a development board.

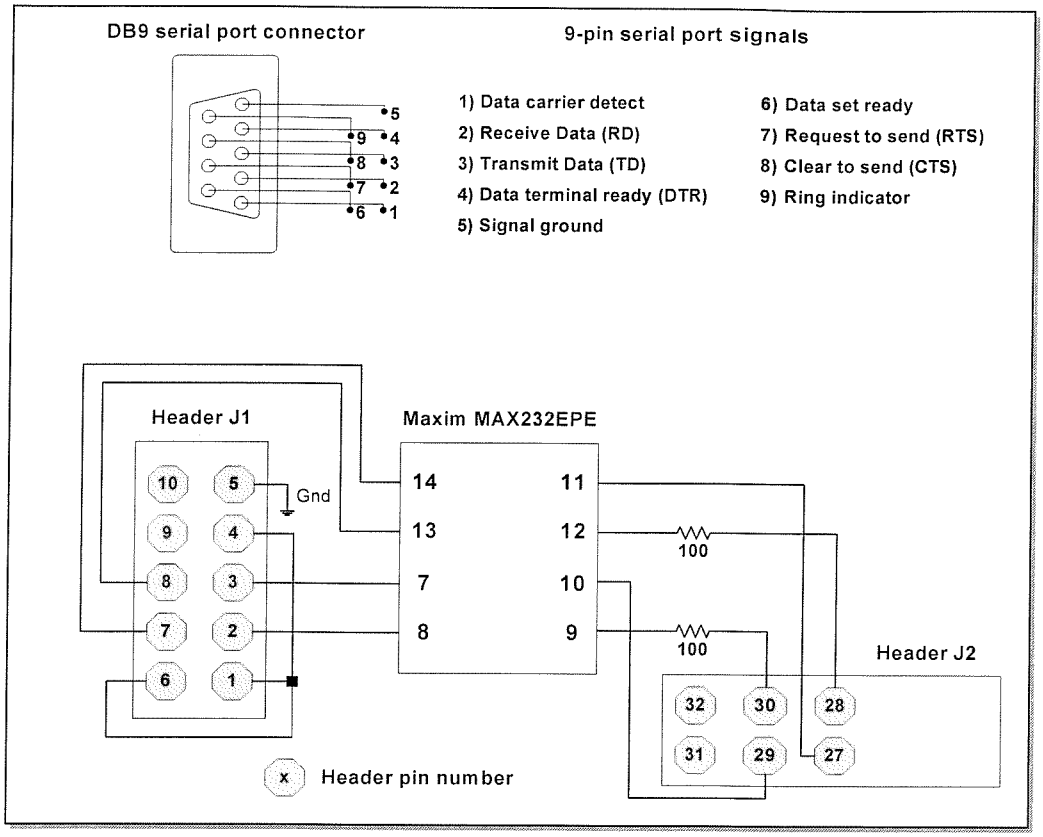


Figure 7-4 9-pin RS-232 serial port interface

7.2.2.2 Fast, Asynchronous SRAM module

The I/O VGA ext. board includes a four-megabyte Static Random Access Memory (SRAM) module (AS7C34098) designed for memory applications where fast data access, low power, and simple interfacing are desired. The surface-mount memory device in a 44-pin JEDEC 400-mil TSOP2 standard package sits on a TSOP-DIL adapter. The inputs and outputs are TTL- and CMOS-compatible with a high speed address access time of 15 ns and output enable access time of 7 ns. The memory is organised as 262,144 words X 16

bits. The SRAM module has separate byte enable controls, allowing individual bytes to be written and read. Control signal $\overline{LB}$ controls the lower bits, *I/O1* to *I/O8*, and $\overline{UB}$ controls the upper bits, *I/O9* to *I/O16*. Table 7-1 shows the 18-bit SRAM address, data and control signal connections between the SRAM and header J3.

Address bit	Header J3 pin number	Signal	Header J3 pin number
A1	1	I/O1	19
A2	2	I/O2	20
A3	3	I/O3	21
A4	4	I/O4	22
A5	5	I/O5	23
A6	6	I/O6	24
A7	7	I/O7	25
A8	8	I/O8	26
A9	9	I/O9	27
A10	10	I/O10	28
A11	11	I/O11	29
A12	12	I/O12	30
A13	13	I/O13	31
A14	14	I/O14	32
A15	15	I/O15	33
A16	16	I/O16	34
A17	17	$\overline{CE}$	35
A18	18	$\overline{WE}$	36
		$\overline{LB}$	37
		$\overline{UB}$	38
		$\overline{OE}$	39

Table 7-1 SRAM address, data and control signal connections to header J3

7.2.2.3 VGA interface

The VGA interface on the I/O VGA ext. board has a Conexant BT121 triple 8-bit videoDAC chip, with triple 8-bit digital to analogue converters for operations up to 80MHz and driving a monitor in 24-bit True colour (16.8 million colours) mode. Figure 7-5 shows the connections between the Conexant BT121 videoDAC chip, 74HCT244 buffer, and headers J2, J4 and J5. Components connected to the BT121 videoDAC are omitted for clarity. Detailed connections of all the components can be found in Appendix B.

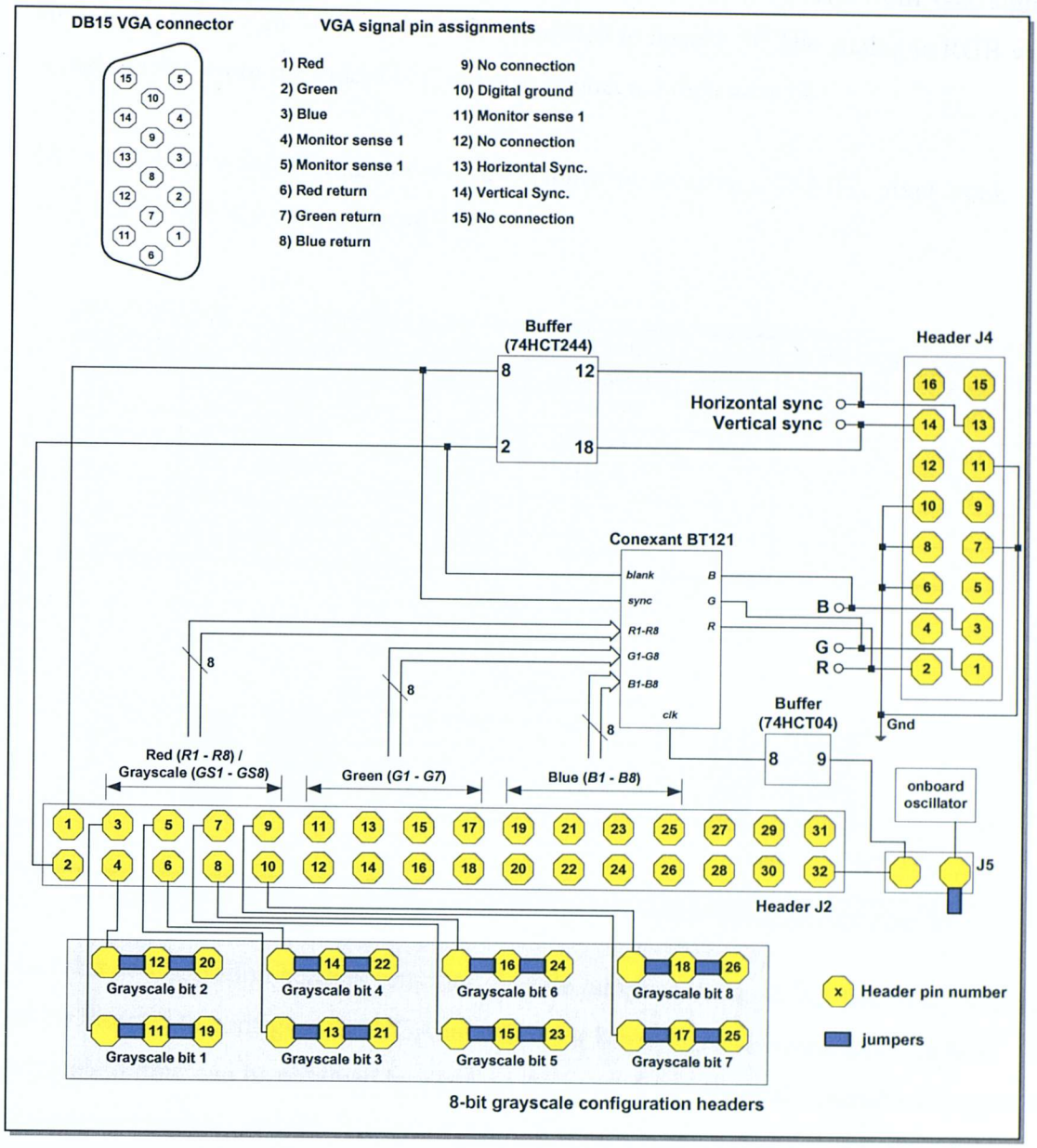


Figure 7-5 VGA interface connections

Red, Green, and Blue (RGB) input video digital data are sent to the Conexant videoDAC via a number of pins on header J2: pin 3 to 10 for the read (R) component, pins 11 to 18 for the green (G) component, and pins 19 to 26 for the blue (B) component. The pixel clock defines the time available to display one pixel of information. This pixel clock input is taken from pin 32 of header J2, or from an onboard oscillator if the two pins on header J5 are shorted with a jumper. The clock input to the videoDAC is a buffered (through 74HCT04) pixel clock signal. The vertical sync (VS) signal defines the "refresh" frequency of the display and this is taken from pin 2 of header J2. The number of lines to be displayed at a given refresh frequency defines the horizontal "retrace" frequency, and

this horizontal sync (HS) signal is taken from pin 1 of header J2. VS and HS signals are connected to the *blank* and *sync* inputs of the videoDAC respectively. Both sync signals are buffered (through 74HCT244) and connected to header J4. The analogue RGB video signal outputs from the videoDAC are also connected to header J4.

The VGA timing for a standard 640x480 display mode using a 25 MHz pixel clock and 60 +/- 1 Hz refresh is given in Figure 7-6.

Symbol	Parameter	Vertical sync			Horizontal sync	
		Time	Clocks	Lines	Time	Clocks
T_S	Sync pulse time	16.7 ms	416800	521	32 us	800
T_{disp}	Display time	15.36 ms	384000	480	25.6 us	640
T_{pw}	VS pulse time	64 us	1600	2	3.84 us	96
T_{fp}	VS front porch	320 us	8000	10	640 ns	16
T_{bp}	VS back porch	928 us	23200	29	1.92 us	48

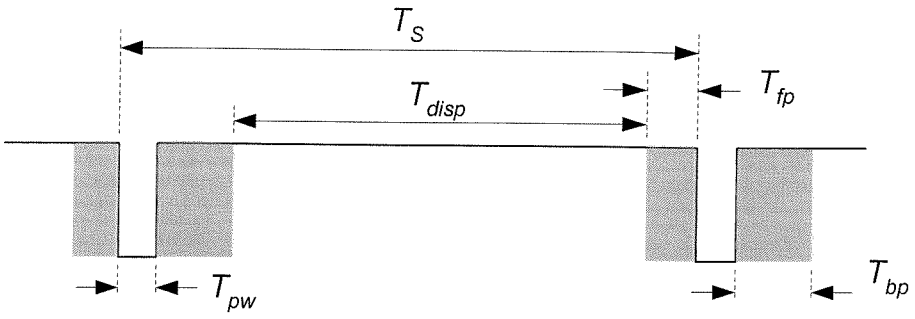


Figure 7-6 VGA timing for a standard 640x480 display mode

The 8-bit configuration headers provides flexible jumper setting so that removal jumpers can be inserted to configure the VGA interface for 8-bit grayscale operation. An 8-bit greyscale output can be obtained by sending the same 8-bit grayscale digital video data to all three components (RGB). The configuration headers are arranged in three headers per set, with a total of eight sets to correspond to the 24-bits RGB signals. Each set of the 3-pin headers corresponds to a bit of the grayscale value and they connect each bit of the individual colour component when the jumpers are inserted. For example, the set of headers for *grayscale bit 1(GS1)* are connected to bit 1 of red component (*R1,* pin 3 of header J2), bit 1 of green component (*G1,* pin 11 of header J2), and bit 1 of the blue component (*B1,* pin 19 of header J2). The 8-bit grayscale (*GS1* to *GS8*) digital input data is taken from pins 3 to 10 of header J2 and the jumpers in the configuration headers send the 8-bit grayscale values to all three components (R, G, and B).

The 8-bit configuration headers offers a two-fold advantage, first, the hardware demonstrator in this projects uses the 8-bit grayscale configuration, however the use of the configuration headers instead of a fixed wiring approach allows the 24-bit true colour VGA interface to be easily used in future projects and hardware demonstrators simply by removing the removal jumpers. The second advantage is the development board now only needs to send an 8-bit grayscale digital data to the I/O and VGA ext. board and the 8-bit value is wired to all three components through the configuration headers, thus reducing the physical I/O connections needed for an 8-bit grayscale operation.

7.3 JPEG decoder in a multi-FPGA system

JPEG (Joint Photographic Experts Group) is one of the most popular algorithms for still image compression. The formal name of the standard that most people refer to as 'JPEG' is the ISO/IEC 10918-1 | ITU-T Recommendation T.81 [150]. The basic JPEG standard defines many options and alternatives for compression of still images of photographic quality. There are four distinct modes of operation defined under which the various coding and decoding processes are defined: *sequential Discrete Cosine Transform-based* (or *sequential Baseline*), *progressive Discrete Cosine Transform-based*, *lossless*, and *hierarchical*. This section covers the information on the implementation of a hardware JPEG decoder to decompress and reconstruct a grayscale image data compressed using the JPEG sequential DCT- based compression. Further details on the JPEG algorithm using the other coding methods and decoding JPEG images can be found in [143, 150-152]. JPEG is a compression algorithm and does not define a specific file format for storing the final data values. The JPEG File Interchange Format (JFIF) [153] is a minimal file format which enables JPEG bitstreams to be exchanged between a wide variety of platforms and applications. JFIF is currently the industry standard file format for JPEG files.

7.3.1 Sequential baseline JPEG decoder

The main procedures in the encoding and decoding processes based on DCT are illustrated in Figure 7-7. The rest of this section only describes the decoding of a grayscale JPEG compressed image in the JFIF file, however, the encoding process is basically the same as

performing the decoding steps but in reverse, and in the opposite order as shown in Figure 7-7.

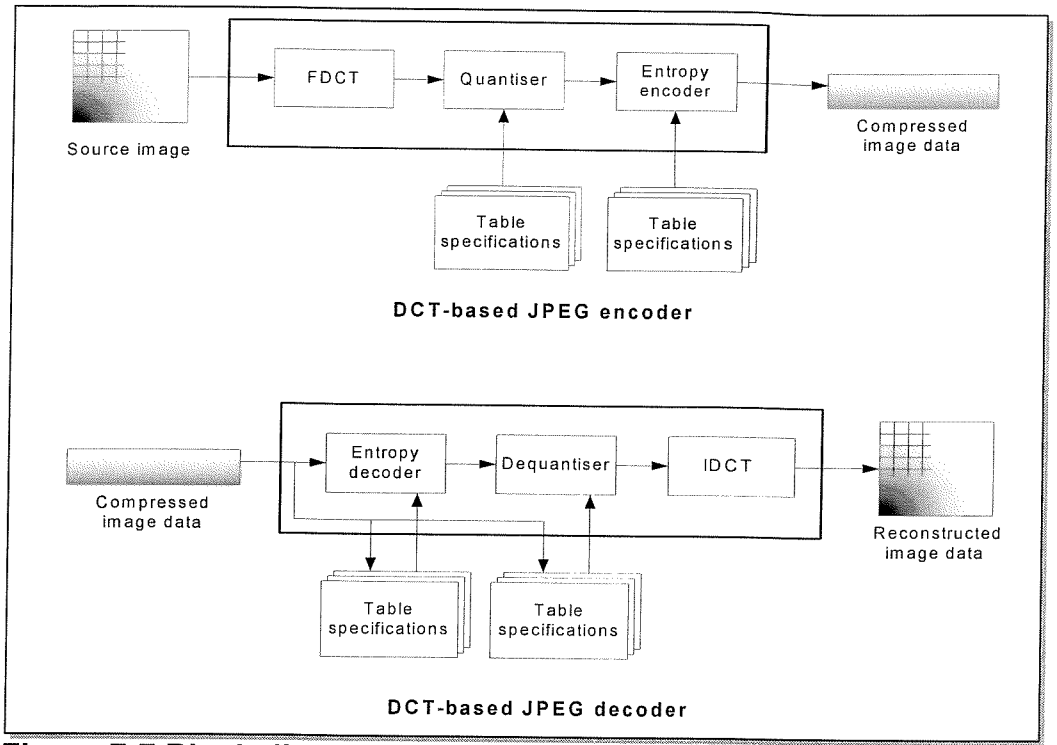


Figure 7-7 Block diagram of a DCT-based JPEG encoder and decoder

Entropy Decoder

The JPEG entropy decoder implemented in this hardware demonstrator is based on the programmable VLC decoder for JPEG described in [143, 154]. The entropy decoder consists of two main data decompression units: a variable length decoder (VLD) and a run-length decoder (RLD). In the JPEG encoder, the quantised DCT coefficients are pre-processed prior to entropy coding. The DC coefficient has a correlation among adjacent blocks and its value varies slightly between successive blocks. DC coefficients are coded using differential coding. DC_i and DC_{i-1} in denote the DC coefficients of blocks i and i-1 as shown below in Figure 7-8.

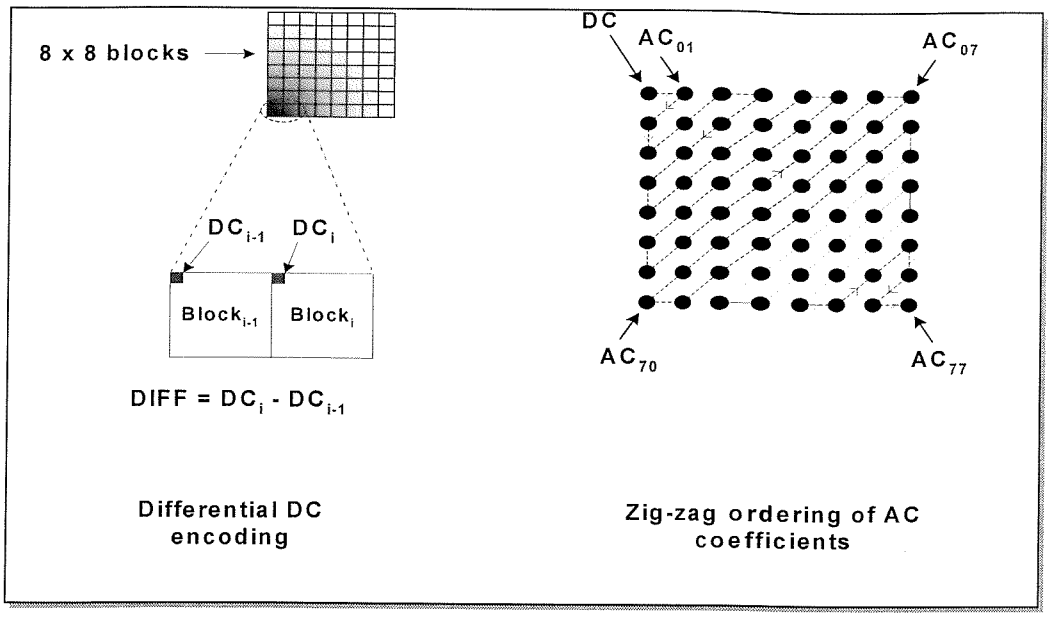


Figure 7-8 Zig-zag arrangement of the DC and AC coefficients

The coefficients are rearranged into a one-dimensional array using a zigzag pattern as illustrated in Figure 7-8, placing the low frequency AC coefficients at the start of the linear sequence and the high frequency coefficients at the end. This groups the zeros resulting from the high frequency AC coefficients together, increasing the consecutive runs of zeros for run-length coding. A run-length coder compresses the quantised DCT coefficients by representing consecutive zeros with a run-length value. Each AC coefficient is represented by two symbols, where *symbol1* is a combination of {run-length, size} values and *symbol2*, which is the quantised frequency value {amplitude}, is encoded with a variable-length integer (VLI). The size value is the number of bits needed to represent the second symbol. DC coefficients are also represented by two symbols, but *symbol1* has only the size value. The first symbol, with the {run-length, size} information is next encoded using a specified Huffman table.

The VLC decoding process begins with the retrieval of Huffman table values in the JFIF file and the entropy decoder decompresses and decodes the Huffman-coded data in the compressed image. The four least significant bits of the decoded *symbol1* specify the number bits used to encode *symbol2*. A one in the most significant bit (MSB) of *symbol2* denotes positive amplitude and the value of the extracted codeword represents the actual amplitude of the coefficient. A zero in the MSB denotes negative amplitude and the amplitude of the DCT coefficient is given by a one's complement of the extracted

codeword. The entropy decoder decodes the differential coded DC values by simply doing the opposite process, DC coefficients for each block are computed by adding the first coefficient value with the preceding DC coefficient. The entropy decoder is described in behavioural VHDL and synthesised using the MOODS synthesis system. An example of entropy decoding is given below in Figure 7-9, the image data is decoded using the standard Huffman tables for luminance components defined in [150].

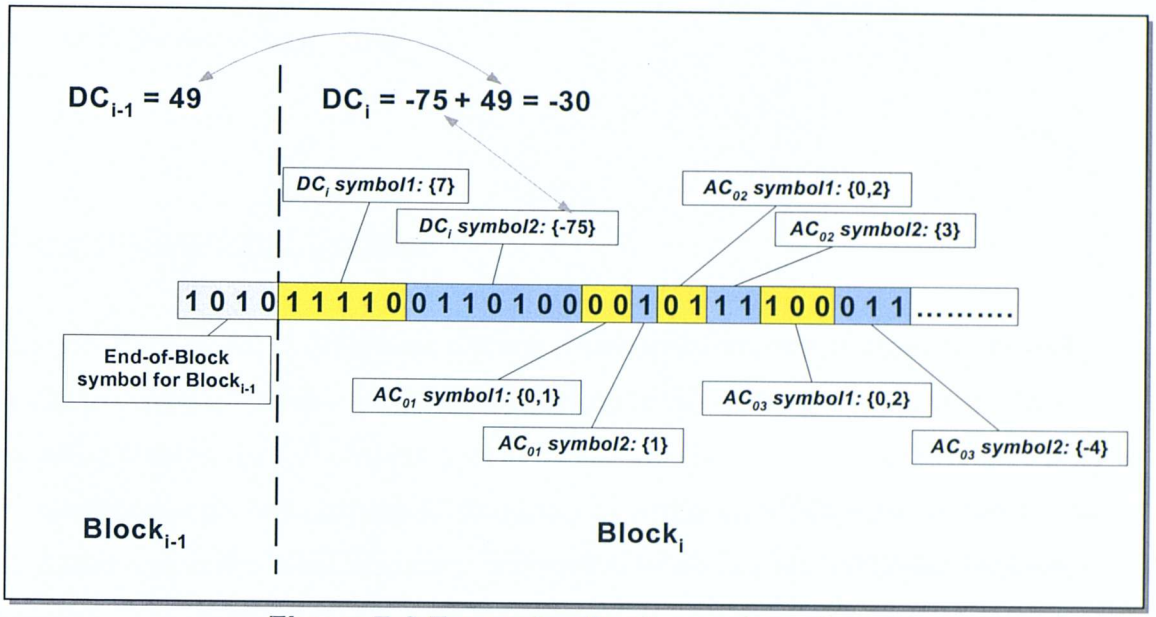


Figure 7-9 Example of entropy decoding

Dequantiser

The 8x8 blocks are dequantised by multiplying each DCT coefficient value in the blocks with the corresponding value in the 8x8 quantisation matrix specified in the JFIF file. The constant values specified in the quantisation matrix may be arbitrary, but generally these values are usually calculated based on the quality versus size factor. During the quantisation process in the JPEG encoding process, high constant values introduce more errors in the rounding up or down of the values obtained from the division of the DCT coefficients and the quantisation matrix. However, having high constant values also result in more high frequency DCT coefficients with small data values to become zero. Our human eyes are not sensitive to high frequency data information, thus the image will look very close to the original. The zig-zag arrangement described in the previous section tends to group the zeros together to form long *run* of zeros, thus allowing the entropy encoder to

further compress the data. A constant value of 1 will result in nearly lossless compression (loss will be due to the round-off errors), whereas a constant value of 255 is the maximum amount of loss for that coefficient. Since arbitrary constant values could be used during the quantisation process, the entire quantisation matrix is stored in the JFIF file so that the dequantiser will know the constant value to multiply each DCT coefficient by to obtain a dequantised 8x8 block. The final step is the decoding of the zig-zag ordered values to reconstruct the frequency domain 8x8 blocks that were originally obtained after the DCT process in the encoding process.

Inverse Discrete Cosine Transform

The two-dimensional (2-D) inverse discrete cosine transform is performed on the 8x8 blocks to convert data from the frequency domain to the spatial domain. In the JPEG encoding process, the 2-D Discrete Cosine Transform (DCT) [155] was performed prior to the quantisation phase to group high frequency information, which is not as sensitive to the human eye as the lower frequency information when they are minimised (or even removed). The coefficients of the resultant frequency domain matrix, or DCT matrix, contain integers in the range of –1024 to 1023. The upper left entry in the resultant DCT matrix, is the DC coefficient, which is the average of the entire block and the lowest frequency cosine coefficient. The higher frequency remaining 63 coefficients or the AC coefficients occur at the lower right of the matrix. The high frequency AC coefficient values are often significantly smaller than the lower frequency coefficients, small enough to be neglected with little visible distortion to the image. The JPEG compression takes advantage of this and typically, the entire lower right half of the matrix comprises only zeros after the quantisation phase.

The 2-D IDCT module implemented in this hardware demonstrator is based on the vector processing technique, which is widely used in hardware implementation of image processing and video coders and decoders because of the regular structure, simple control logic and a good balance between complexity of implementation and performance. The 2-D IDCT module is described in behavioural VHDL and synthesised using the MOODS synthesis system. The 2-D IDCT architecture is adapted from [142] and it is illustrated in Figure 7-10.

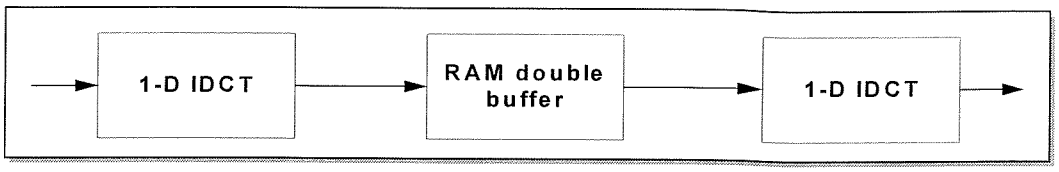


Figure 7-10 2-D IDCT architecture

The architecture is made up of a one-dimensional 8-point IDCT followed by an internal double buffer memory, followed by another one-dimensional 8-point IDCT. The algorithm used for the calculation of the 2-D IDCT is based on the equation (7.1).

$$XC_{pq} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} XN_{mn} \bullet \frac{c(p)c(q)}{4} \bullet \cos\left(\frac{\pi(2m+1)p}{2M}\right) \bullet \cos\left(\frac{\pi(2n+1)q}{2N}\right) \tag{7.1}$$

Equation (7.1) can be separated into the row part and column part as shown in equations (7.2) and (7.3). The 2-D IDCT is computed by first applying 1-D IDCT on the rows and then on the columns.

$$C = \mathrm{K} \bullet \cos \frac{(2 \cdot col\ number\ + 1) \bullet row\ \mathrm{number}\ \bullet \pi}{2 \bullet \mathrm{M}} \tag{7.2}$$

where $\mathrm{K} = \frac{\sqrt{1}}{\mathrm{N}}$ for row = 0, $\mathrm{K} = \frac{\sqrt{2}}{\mathrm{N}}$ for row $\neq 0$

$$C^{t} = \mathrm{K} \bullet \cos \frac{(2 \cdot row\ number\ + 1) \bullet col\ number\ \bullet \pi}{2 \bullet \mathrm{N}} \tag{7.3}$$

where $\mathrm{K} = \frac{\sqrt{1}}{\mathrm{M}}$ for col = 0, $\mathrm{K} = \frac{\sqrt{2}}{\mathrm{M}}$ for col $\neq 0$

During the JPEG encoding process, the image samples are level-shifted to a signed representation by subtracting 2^{P-1}, where P is the precision parameter of the image specified in the JFIF. For a grayscale image with 8 bits precision, the 8-bit signed values

are level-shifted back to the original sample values by adding 128 to each of the values in the 8x8 block resulting from the IDCT transform. Figure 7-11 shows an example of an 8x8 dequantised block input to the IDCT module and the corresponding 8x8 values obtained during the IDCT process.

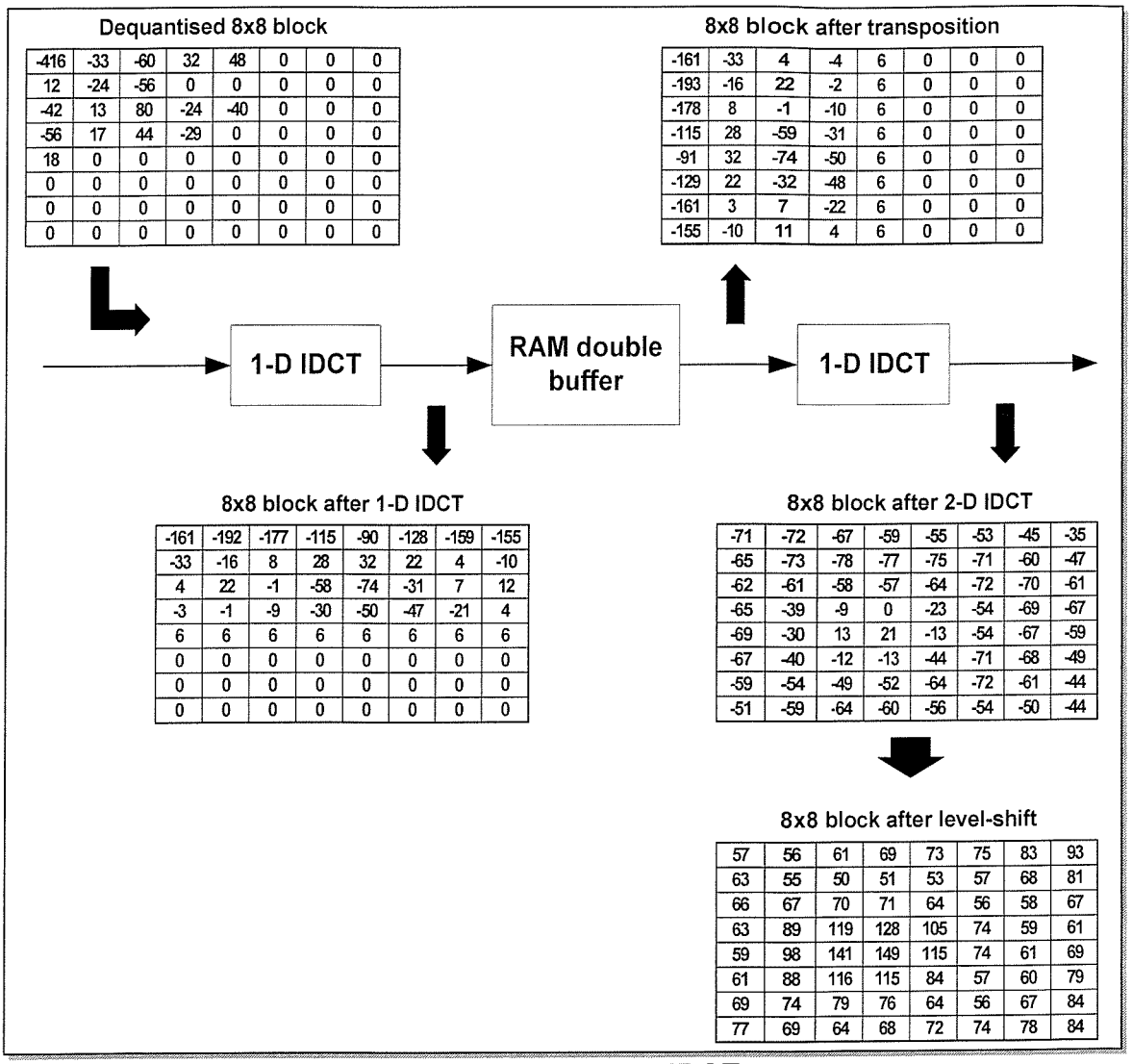


Figure 7-11 Example of the IDCT process

7.3.2 Partitioned JPEG decoder

The sequential DCT-based JPEG algorithm and the description of the key components are covered in the previous section. This section describes the partitioning of a JPEG decoder, which forms the core of the hardware demonstrator. Figure 7-12 illustrates the overview of

the hardware demonstrator system. There are three distinct phases in the multi-FPGA JPEG decoder: *Input phase, Output phase, JPEG decoding phase.*

A simple file I/O Graphical User Interface (GUI) [156] is used to select the JFIF file to be decoded in the *input phase*. This JFIF file in the source PC is then transferred serially to the RS-232 interface on the I/O VGA extension board using a serial (null-modem) cable. The *output phase* is the visual output of the decoded JPEG image on a VGA monitor. The high-speed 4-megabyte SRAM on the I/O VGA extension board is used as a frame buffer to store the decoded pixel data values in the 8x8 blocks in a raster-scan manner, suitable for a standard 640x480 pixel VGA display.

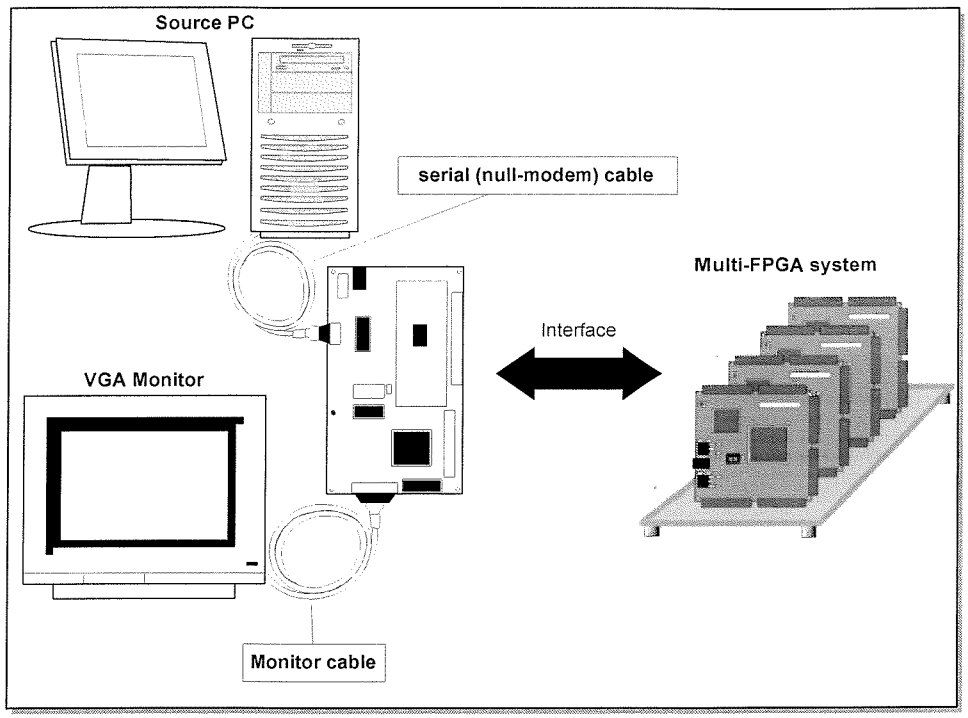


Figure 7-12 Overview of the hardware demonstrator system

The *JPEG decoding phase* is the core of the hardware demonstrator and is performed by a partitioned JPEG decoder in a multi-FPGA system, formed with a number of hardware development boards. The hardware implementation of the JPEG decoding algorithm is partitioned and synthesised using the K-way partitioning enhancements described in Chapter 4. The re-configurable device on each development board in the multi-FPGA system is viewed as a locally clocked processing unit performing part(s) of the JPEG

decoding algorithm, and inter-device data transfers performed using asynchronous techniques described in Chapter 5.

7.3.3 VHDL Design

The hardware demonstrator comprises VHDL modules written in two different styles of VHDL, behavioural VHDL and Register Transfer Level (RTL) VHDL. Modules communicating directly with the input/outputs (i.e. the I/O and VGA extension board) of the system are written in RTL VHDL, as strict timing requirements have to be met. For example, the VGA driver module that has to send the pixel data every 40 ns for a pixel clock of 25 MHz. Adapting a design with a mixture of modules coded using RTL and behavioural VHDL and not solely using the behavioural style of VHDL is not design limitation, however, it is more sensible and less time consuming to adopt such a design approach.

Figure 7-13 shows the overall VHDL modules in the hardware demonstrator. The *S data interface*, *UART*, *Frame buffer controller*, and *VGA driver module* are written in RTL VHDL, with the rest of the modules written in behavioural VHDL and synthesised using the MOODS synthesis system. The *UART* module communicates directly with the Maxim MAX232EPE described in Section 7.2.2.1. The *UART* module is part of the VHDL communications library in [156]. The *UART* sends and receives data serially from the Maxim device and bytes of data are passed to the *S data interface* VHDL module. This *S data interface* receives a byte of data from the *UART* and this byte of data is passed to the *Entropy decoder* if it is a compressed image data to be decoded, else to the JFIF file decoder if it is part of the header information. A detailed description on the JFIF file layout can be found in Appendix B.1. The *Block Transpose module* after the JPEG decoder core performs block transposition on the transposed 8x8 blocks of image data from the *IDCT module*.

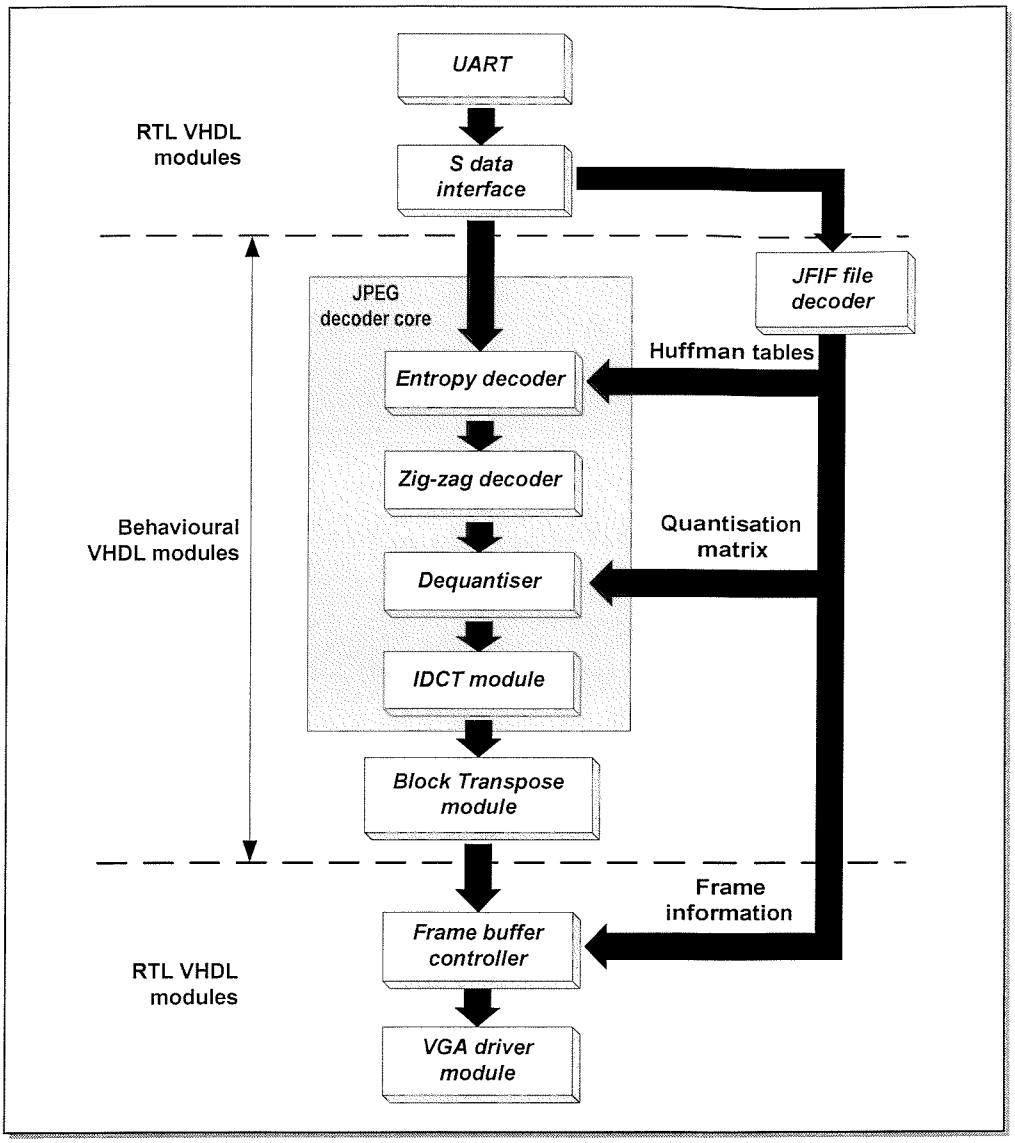


Figure 7-13 VHDL modules in the hardware demonstrator system

The sequential baseline JPEG decompression algorithm decodes the compressed image in 8x8 blocks and the decompressed image is stored in the frame buffer memory. The 4MB SRAM device on the I/O VGA ext. board is used as the frame buffer memory and Figure 7-14 illustrates how the decompressed 8x8 blocks of data are stored in the frame buffer memory.

The frame buffer memory mapping shows how each pixel, specified as an x-y co-ordinate relative to the top left of the VGA monitor display, maps to the memory location in the SRAM device. For an 8-bit grayscale image of up to 512 by 480 pixels, a total of 512 × 480 × 8 bytes (1.92 MB) are required. Blocks of decompressed image data ready to be displayed are sent to the *frame buffer controller* and two bytes of pixel data are stored in

each memory location. Decoded image data is only sent to the *VGA driver module* when a complete image is stored in the frame buffer. The *VGA driver module* generates the horizontal sync and vertical sync timing signals and it coordinates with the *frame buffer controller* to deliver a pixel data on each pixel clock to the I/O VGA ext. board. With the frame information (image height and image width), the VGA driver module sends a 'background' pixel data, filling regions larger than the image with a background colour (black, white, or a shade of grey for an 8-bit grayscale VGA interface).

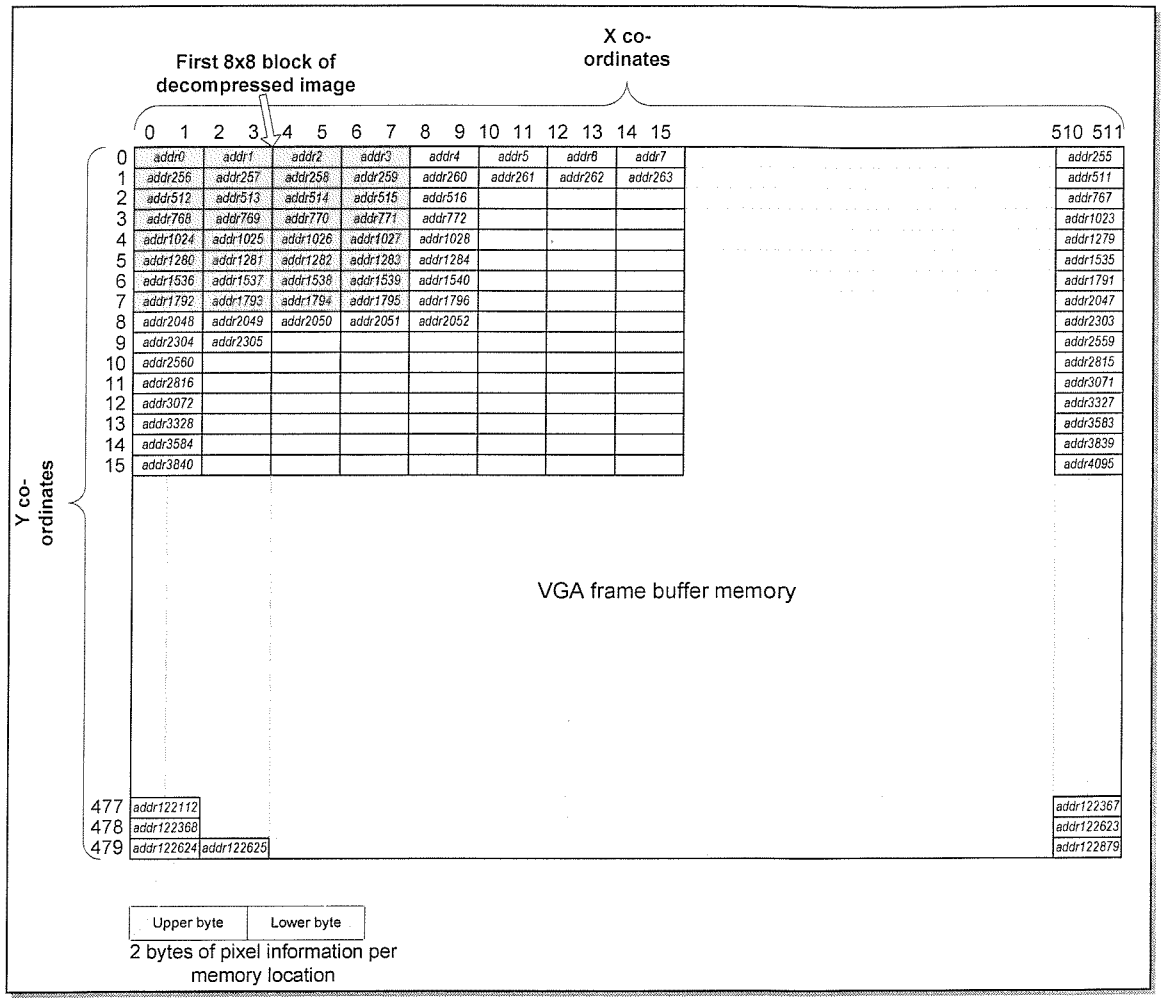


Figure 7-14 Frame buffer memory mapping of 8x8 blocks

7.4 Results and performance

The behavioural JPEG decoder core and block transpose modules are synthesised using the MOODS synthesis system to generate a multi-FPGA system. The multi-FPGA JPEG decoder is targeted onto three Digilent D2-SB FPGA-based development boards

(described in Section 7.2.1) with a Xilinx Spartan 2E-200 FPGA on each board. All the RTL VHDL modules (shown in Figure 7-13) and the JFIF file decoder module are targeted onto a single D2-SB development board and connected to the I/O VGA ext. board. The behavioural JPEG decoder and the block transpose module are synthesised and partitioned using the MOODS synthesis system with the partitioning enhancement described in Chapters 4 and 5.

The whole system is simulated at the gate level (based on the post-placed and route simulation model produced by Xilinx ISE) prior to downloading the multi-FPGA system onto the FPGA devices. After the verification of the multi-FPGA system, the prototyping boards are connected up to form the multi-FPGA JPEG decoder hardware demonstrator system. Examples of the multi-FPGA JPEG decoder in action can be found in the photographs of Figure 7-15 and Figure 7-16, which demonstrates the complete system in full working order.

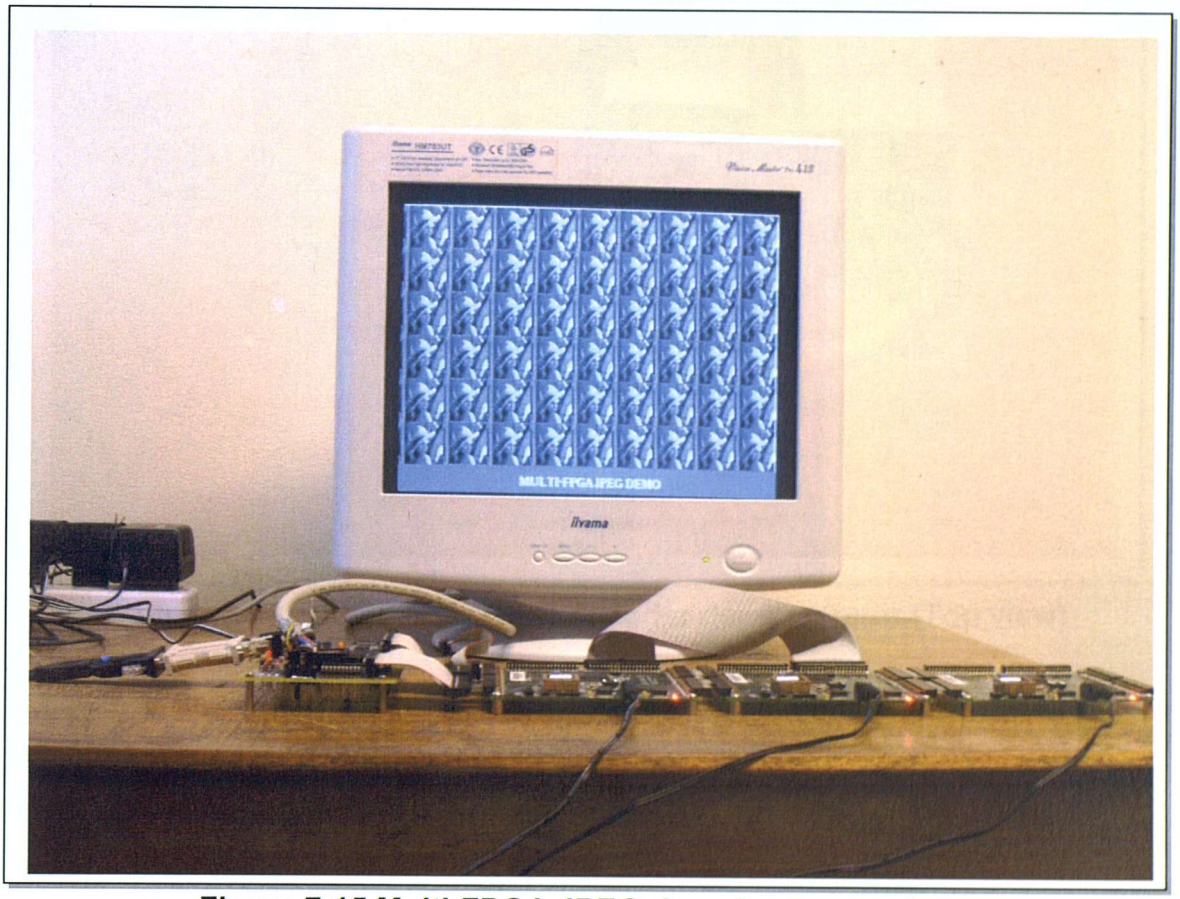


Figure 7-15 Multi-FPGA JPEG decoder demonstrator

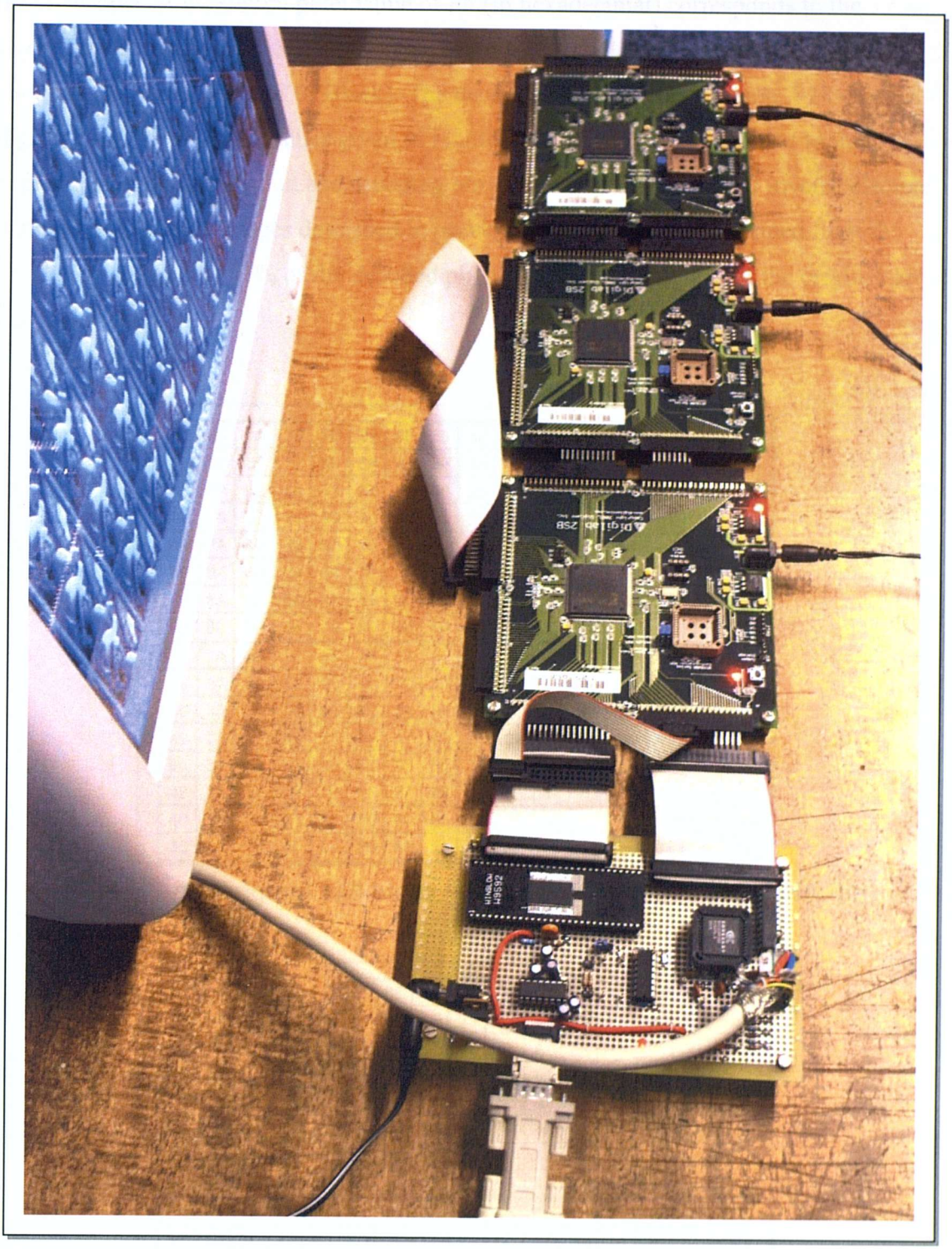


Figure 7-16 Multi-FPGA JPEG decoder demonstrator (Top view)

Figure 7-17 illustrates the pixel values of the test image (LENA.jpg) taken from a graphics viewer [157]. The pixel values in the four 8x8 blocks are given in hexadecimal and they are taken from the top left corner of the test image and top-left most value 0x7C corresponds the top-left most corner pixel value of the test image. Figure 7-18 illustrates the values obtained from a simulation of the test image decoding using the multi-FPGA

JPEG decoder. A maximum pixel value of FF (in hexadecimal) corresponds to the maximum grayscale level of 255 (White), and 00 (in hexadecimal) corresponds to the minimum grayscale level of 0 (Black). The decoded pixel values in Figure 7-18 deviate slightly from the original values given in Figure 7-17, this slight error is due to the imprecision in multiplication and rounding errors in the quantisation and inverse discrete cosine transformation stages described in previous sections. Results obtained from the simulation shows that 97% of the decoded pixel values are within ± 2 of the original pixel grayscale levels.

7C	95	8A	70	8D	89	8F	65	42	4A	4E	4B	4B	54	5D	61
7E	88	84	77	8F	8A	91	5F	3B	43	47	44	45	4D	56	5A
82	7A	82	84	91	87	92	58	3B	44	49	46	47	4F	58	5B
87	75	87	91	90	80	91	56	3B	45	4A	48	49	52	5A	5D
8A	7A	8D	93	8C	7A	8E	58	35	3F	46	45	46	4E	56	58
8B	84	8A	84	88	7B	8C	5B	36	40	47	49	51	51	58	5A
89	8A	7C	6A	85	83	8D	5C	36	41	49	49	4B	53	5A	5B
88	8C	6F	56	84	8C	8E	5B	2E	3A	42	42	44	4C	53	54
91	7A	46	52	85	95	84	59	2A	38	3D	37	3A	4A	51	4D
89	57	31	61	90	8C	7D	5D	2B	3F	4B	46	45	4E	52	4E
6D	3A	2A	66	8C	8D	85	56	29	3C	49	44	42	4A	51	51
49	35	37	60	77	94	97	4C	2E	38	3D	3A	3D	49	54	57
3C	34	3F	61	70	8C	97	55	36	3B	3E	40	46	4F	53	53
43	31	37	69	7D	7F	87	66	31	38	42	49	4C	4B	49	46
40	38	33	61	86	86	81	65	2A	35	42	46	42	41	4B	56
33	47	38	4F	84	9B	89	54	2E	3B	45	42	39	3F	5D	7B

Figure 7-17 Original 8x8 block values from test image (LENA.jpg)

7C	94	8A	6F	8C	88	8E	65	41	49	4D	4A	4A	53	5C	60
7E	87	84	77	8F	89	90	5E	39	42	46	43	43	4C	55	59
82	79	81	84	92	86	92	57	39	42	47	44	45	4E	56	5A
87	74	86	90	90	7F	91	55	3A	43	49	47	48	50	58	5B
8A	7A	8D	92	8C	79	8E	57	34	3E	45	44	45	4D	55	58
8A	83	8A	84	87	7A	8C	5A	34	3F	46	46	48	50	57	59
89	8A	7C	6A	84	83	8C	5B	35	3F	47	48	4A	51	58	5A
88	8C	6E	55	83	8C	8D	5A	2D	38	40	41	43	4A	51	53
90	79	45	50	83	94	82	58	27	36	3B	34	38	49	4F	4B
88	56	30	60	90	8C	7C	5D	29	3F	4A	A5	44	4D	51	4D
6C	39	29	66	8C	8D	85	56	27	3C	48	44	40	48	50	4F
47	34	36	5E	76	93	97	4B	2C	36	3B	39	3B	47	52	55
3A	33	3E	60	6E	8B	97	53	35	39	3C	3F	45	4E	53	51
42	2F	35	69	7C	7E	87	66	2F	37	41	48	4B	4A	48	45
3F	37	31	61	86	86	80	64	28	34	41	44	40	3F	49	55
31	46	37	4D	82	9B	88	53	2D	39	44	41	37	3E	5D	7B

Figure 7-18 Test image (LENA.jpg) 8x8 block values decoded using the multi-FPGA JPEG decoder

7.4.1 Synthesis results of non-pipelined multi-FPGA JPEG decoder

All the RTL VHDL modules and the JFIF file decoder are targeted onto a D2-SB FPGA-based development board with a Xilinx Spartan 2E-200 FPGA (s200E). The frame buffer controller operates at 50 MHz, which is provided by the surface-mount 50MHz oscillator on the development board, and the rest of the modules operate at 25 MHz, which is generated by a simple divide-by-two clock divider. RTL modules are instantiated and linked within the architecture body of a top-level VHDL file. Table 7-2 gives the key details on the resource utilisation and the maximum achievable frequency of the top-level design on the s200E target device.

FPGA	Resource utilisation	
s200E	Area in slices	847 (36%)
	I/O	104 (98%)
	Freq (MHz)	55.57

Table 7-2 Synthesis results of development board 1

The RTL modules in the development board are locally clocked, signals passed between these RTL modules and the partitioned JPEG core (described in behavioural VHDL and synthesised using MOODS) in other FPGAs cross clock domains and needs to be synchronised. The synchronisation logic needed to handshake inter-device signals in the multi-FPGA JPEG core are generated automatically into the MOODS synthesised multi-FPGA implementation without any intervention of the user to the synthesis tool. Only the synchronisation between the RTL modules in development board 1 and the multi-FPGA JPEG core targeted onto two other development boards in the multi-FPGA system had to be performed manually by double buffering the top-level VHDL input signals that originate from the clock domain of the RTL modules.

Figure 7-19 shows a section of the top-level VHDL with input signal *end_conv* and the corresponding generated synchronisation circuit is shown on the right. The circuit shows the signal *end_conv* passed from RTL module in domain 1 to a module in the multi-FPGA JPEG core in domain 2. The generated synchronisation circuit consists of two flip flops (registers), *FF1* and *FF2*, which are clocked by the system clock (*sys_clock*) in domain 2

and system reset (*sys_reset*) is connected to the asynchronous clear (clr) inputs on the flip flops. Signal *end_conv_buf1* is the output of the first flip flop (register), *FF1*, and this is the input into the second flip flop, *FF2*. The output signal (*end_conv_buf2*) from the second flip flop, which is the synchronised input for *end_conv* signal is used by other parts of the circuit in domain 2.

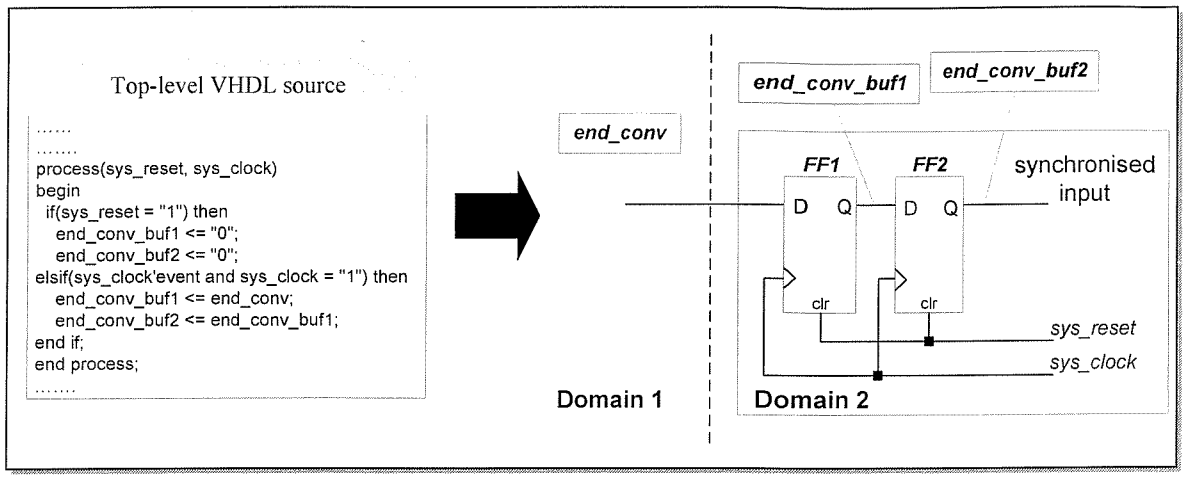


Figure 7-19 Double buffer synchroniser insertion

The module call graph representation of the non-pipelined JPEG decoder core is shown in Figure 7-20. The non-pipelined JPEG decoder had a total of six subprogram modules and one program module.

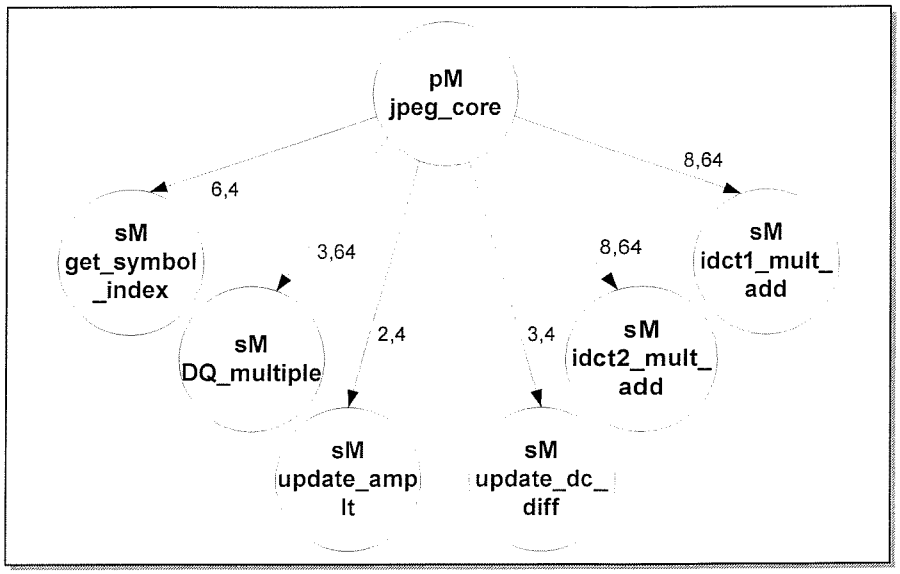


Figure 7-20 Module call graph representation of the non-pipelined JPEG decoder core

Synthesis and K-way partitioning results of the behavioural JPEG decoder core and the block transpose module are given in Table 7-3. The first row shows the synthesis result of a single-device implementation that fits the target device.

Boards	FPGA	Synthesis results					Two-phase partitioning results	
		Opt. priority	Area in slices	I/O	Freq. (MHz)	AO in slices	Data pkts (initial → final)	Channels (Data widths)
1	s400E	Delay	3639 (75%)	47 (44%)	30.13	–	–	–
1	s200E	Delay	3297* (140%)*	47 (44%)	–	–	–	–
2	s200E	Delay	2350 (99%)	106 (100%)	36.79	506 (13.9%)	1216 → 512	2 *SpC*
	s200E		1795 (76%)	71 (67%)	36.34			(21, 28)

Table 7-3 Synthesis results of the non-pipelined JPEG decoder core

The s400E target device in the first row is a Xilinx Spartan2E-400 FPGA [158] in a FT256 package. It has a maximum device area of 4000 slices and a total number of 182 user I/Os. The maximum number of available I/Os for the implementation of the multi-FPGA system is restricted to 106 I/O pins as a number of pins on the s200E target device are connected to the push button switch, integrated circuit socket for a second clock module, and LED on the development board (see Appendix B.6). A detailed description on the pin assignments of the hardware demonstrator development boards is given in Appendix B.4. The Xilinx Spartan2E-400 FPGA is the smallest target FPGA in the Xilinx Spartan2E target technology that has sufficient device area to hold the JPEG decoder core and the block transposed module, with a device utilisation of 3639 slices occupying 75% of the maximum target area, and an I/O utilisation of 47 pins out of the 106 pins available. This un-partitioned single chip implementation has a maximum achievable frequency of 30.13 MHz.

The second row shows that targeting the design onto a Xilinx Spartan2E-200 FPGA results in an area utilisation of 3297 slices, which exceeds maximum area of the s200E device. The last two rows in Table 7-3 show the non-pipelined synthesised design targeting two s200E devices and implemented using two Digilent development boards. The first partition (shown in row 3) occupies 99% of the maximum area in the target s200E device and 100% of the total number of I/O pins available. The second partition

occupies 76% of the maximum area in the target s200E device and 71 I/O pins (67% of the maximum I/O available). The area overhead of 13.9% is due to the insertion of communication cells and arbiters (described in Section 5.4 and the duplication of data registers in the multi-FPGA implementation. K-way partitioning, with design profiling, with the aim of reducing inter-device data transfers was completed in two passes of the K-way partitioning iteration loop. Two subprogram communication channels (with data widths of 21-bits and 28-bits) were generated automatically in MOODS to handle the inter-device data transfers between subprogram modules in the two locally clocked target devices. The two development boards run at 25 MHz and they are clocked independently using the on-board 50MHz oscillator, and a simple divide-by-two clock divider.

7.4.2 Computation cycles and inter-device data transfers

Simulations were conducted on multi-FPGA JPEG decoder using the synthesised netlist output files generated by MOODS. Performance of the decoder in un-partitioned single device and multi-FPGA implementations obtained from the post-MOODS simulation of test images are presented in Table 7-4. The total number of inter-device data transfers over subprogram communication channels 1 (*SpC 1*) and 2 (*SpC 2*)are given in columns 2 and 3 respectively. The JPEG decoder system is a complex and computation intensive design, the computation clock cycles of the JPEG core given in columns 4 and 5 of Table 7-4. The performance degradation (approximately 7 times increase in design latency) is because of the immense number of clock cycles required to decode and store the decoded pixels in a frame buffer ready to be displayed upon completion of the decoding process.

SpC 1 with a 21-bit wide data width is shared by two transmit cells, two receive cells, and the channel arbitration is provided by a multi-arbiter cell. A single pair of transmit and receive cells is connected to *SpC 2* and the arbitration for this communication channel with a 28-bit wide data width is provided by a single-arbiter cell. The relatively large number of inter-device data packets transferred over the two communication channels also provides a robust test for testing the communication cells and communication channel arbiter cells described in Section 5-4.

Test image (.jpg)	Inter-device data packets		Clock cycles		Max Freq (MHz)		Design latency (ms)	
	SpC 1	*SpC 2*	Un-partitioned	Multi-FPGA	Un-partitioned	Multi-FPGA	Un-partitioned	Multi-FPGA
LENA	37807	20480	178104	1496464	30.13	36.34	5.91	41.18
MANDRILL	37087	20480	171724	1477008			5.70	40.64
DRAGON	34082	20480	170406	1398412			5.66	38.48
SQUARES	469322	327760	2287230	20572700			75.91	566.12
SLOPE	125883	81920	606572	5308488			20.13	146.08

Table 7-4 Computation clock cycles and inter-device data transfers in the non-pipelined multi-FPGA JPEG decoder

The maximum frequencies given in the table are obtained from the Xilinx ISE synthesis implementation results given in Table 7-3 and the maximum frequency of the multi-FPGA implementation is the maximum achievable frequency of the slowest FPGA device. The design latency, time taken to decode the test images, is calculated by multiplying the number of computation clock cycles by the clock period (1/Max Freq). Design latencies of the un-partitioned (single chip) and non-pipelined multi-FPGA JPEG decoder system are given in the last two columns of Table 7-4.

A complete profile and photographs of the test images decoded using the multi-FPGA JPEG decoder are given in Appendix B.2.

7.4.3 Further analysis

Synthesis results and performance of the non-pipelined multi-FPGA JPEG decoder partitioned and synthesised using MOODS are given in the previous sections. This section gives a further analysis on the implementation of the multi-FPGA JPEG decoder. Figure 7-21 illustrates the structure of the two communication subsystems inserted in the multi-FPGA JPEG decoder to deal with the transfer of inter-device data packets between the main JPEG core module in development board 2 and subprogram modules in development board 3. Subprogram modules (DQ_multiple, update_amplt and update_dc_diff) in development board 2 and the control and data path node units in the synthesised output structure have been omitted for clarity.

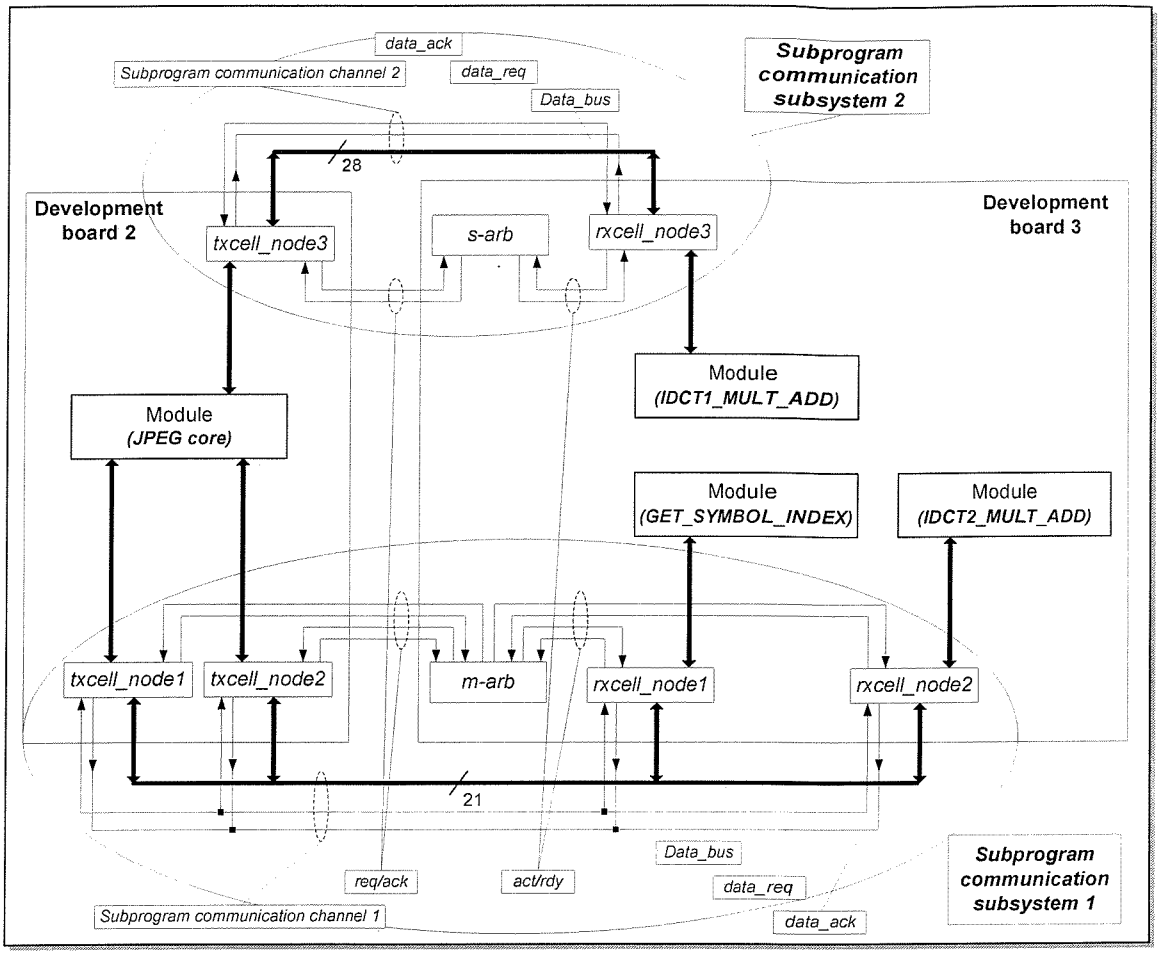


Figure 7-21 Structure of subprogram communication subsystem in the non-pipelined multi-FPGA JPEG decoder

Subprogram communication subsystem 1 has two transmit cells (*txcell_node1* and *txcell_node2*), two receive cells (*rxcell_node1*and *rxcell_node2*), and a multi-arbiter (*m_arb*). Inter-device data transfers initiated by communication cells in communication subsystem 1 are sent through subprogram communication channel 1 (*SpC 1*) which has a 21-bits wide *Data_bus*. Subprogram communication subsystem 2 has a single transmit cell (*txcell_node3*) and receive cell (*rxcell_node3*) connected to a single-arbiter (*s_arb*). Inter-device data transfers between *txcell_node3* and *rxcell_node3* are sent through subprogram communication channel 2 (*SpC 2*) which has a 28-bits wide *Data_bus*. Three modules (external), *GET_SYMBOL_INDEX*, *IDCT2_MULT_ADD*, and *IDCT1_MULT_ADD* are targeted onto the FPGA device in development board 3. Data packets (input parameters) are sent to the receive cells, which activate these *external* modules upon receiving all the input data packets. Output results are sent back to the transmit cells when the operations in

the modules are completed. Input parameters are sent in multiple inter-device data packets when the sum of the bits in all the input parameters exceeds the bit-width of the *Data_bus* in the communication channel, likewise for output parameters if the sum of the bits in all the output parameters exceeds the bit-width of the *Data_bus*.

The K-way partitioning algorithm and the communication subsystem optimisation algorithm optimise the multi-FPGA system in terms of delay across FPGA boundaries, while satisfying the area and I/O constraints of the target devices. If the area and I/O constraints can be relaxed (i.e. targeting FPGA devices with a larger area, or more I/O pins), the number of external modules may be reduced as more modules can be targeted onto a single FPGA device, and hence reduce the number of inter-device data transfers, similarly targeting a device with a larger number of I/O pins, the bit-width of the *Data_bus* in the subprogram communication channel can be increased, such that all input parameters of output results can be transferred in a single data packet.

Assume development board 2 of the multi-FPGA implementation has a target device with a larger area; a Xilinx Spartan2E-300 FPGA (with 3072 slices in area) instead of a Xilinx Spartan2E-200 FPGA (with 2352 slices in area), it is then possible to re-assign and map at least one of the three 'external' modules in development board 3 to development board 2. Table 7-5 shows the effect of reducing the number of external modules in the multi-FPGA JPEG decoder on the inter-device data transfers and computation clock cycles results. The number of external modules in the multi-FPGA JPEG decoder is given in the column 1. The number of inter-device data packets transferred over *SpC 1* and *SpC 2* are given in columns 2 and 4 respectively. The channel (*Data_bus*) widths of *SpC 1* and *SpC 2* are given in columns 3 and 5 respectively. The number of computation clock cycles of the multi-FPGA JPEG decoder core is given in column 6. The maximum frequency in column 7 is obtained from the Xilinx ISE synthesis implementation results given in Table 7-3. The design latency of the multi-FPGA JPEG decoder system is given in the last column of Table 7-5.

The test image used is LENA.jpg and the first row shows the performance of the multi-FPGA JPEG decoder with three external modules in development board 3 as illustrated in Figure 7-21. The second row gives the performance of the decoder when an external module is moved into development board 2 (this assumes that there is sufficient area in the FPGA device on development board 2 to hold *IDCT2_MULT_ADD*), hence development

board 3 is left with only two external modules (*GET_SYMBOL_INDEX* and *IDCT1_MULT_ADD)*. The effect of removing module *IDCT2_MULT_ADD* from communication subsystem 1 is a reduction in the number of inter-device data packets in *SpC 1*. This amounts to 57.9% reduction in the design latency as compared with the original implementation with three external modules.

Number of external modules	Inter-device data packets				Clock cycles	Max Freq (MHz)	Design latency (ms)
	SpC 1	Channel width	*SpC 2*	Channel width			
3	37807	21	20480	28	1496464	36.34	41.18
2	8805	21	20480	30	629882		17.33
1	8805	21	-	-	417636		11.49

Table 7-5 Number of external modules and its effect on the performance of the non-pipelined multi-FPGA JPEG decoder

The last row in Table 7-5 shows the performance of the multi-FPGA JPEG decoder core with just a single external module (*GET_SYMBOL_INDEX)* in development board 3. Only a single communication subsystem is required to transfers inter-device data packets to and from the single external module. The total number inter-device data packets are reduced even further with just a single external module, and this amounts to 72.1% reduction in the design latency as compared with the original implementation with three external modules. The time taken to decode the test image (LENA.jpg) using the multi-FPGA JPEG decoder with a single external module is 11.49 milliseconds, which approximately twice the time needed for a single-chip implementation (given in Table 7-4). The graph in Figure 7-22 shows the design latency (decoding time) versus the number of external modules in the multi-FPGA JPEG decoder. With less external modules, the number of inter-device data transfers is reduced and this improves the performance of the multi-FPGA JPEG decoder as the design latency reduces. Area utilisation of the target devices will also decrease, as a result of lesser communication cells and duplicated hardware (registers) to handle inter-device module calls.

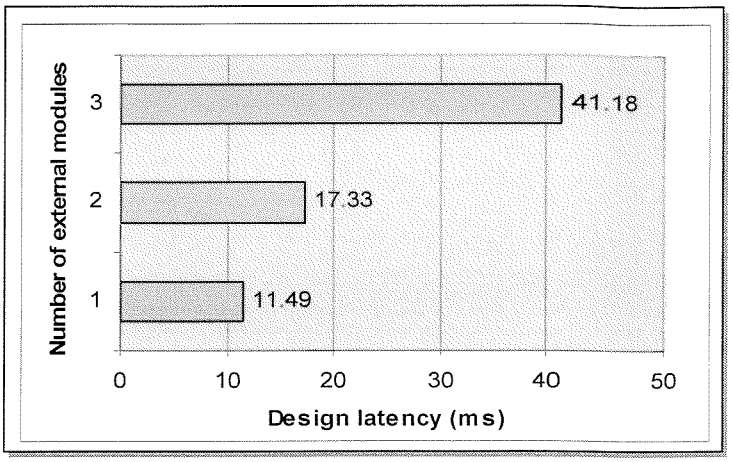


Figure 7-22 Graph of design latency versus the number of external modules in the multi-FPGA JPEG decoder

Increasing the number of available I/Os on target devices is the other approach to improve the performance of the multi-FPGA JPEG decoder. The first column in Table 7-6 shows the number of available I/Os on the target devices (i.e. assuming that all the target FPGAs have the same number of available I/Os). The first row shows the inter-device data transfers and computation clock cycles of the hardware demonstrator implemented using the D2-SB FPGA-based development boards with 106 available I/Os (details given in Sections 7.4.1 and 7.4.2). Subsequent rows show the effect of increasing the number of available I/Os on target devices on the performance of the multi-FPGA JPEG decoder. The total number of inter-device data packets decreases with the increment of available I/Os, resulting in a decrease in the number of computation clock cycles and hence reduces the design latency. When 250 I/Os are available (fourth row of Table 7-6), a new communication subsystem is generated by MOODS during synthesis. Communication cells, *txcell_node1* and *rxcell_node1*, are connected to a single-arbiter (*s_arb*), which replaces the multi-arbiter (*m_arb*) in Figure 7-21. Communication cells, *txcell_node2* and *rxcell_node2* are connected to a third (newly) inserted single-arbiter, and together they deal with the inter-device data transfers across a new subprogram communication channel (*SpC 3* in Table 7-6) with a 69-bits wide data width.

I/Os avail.	Inter-device data packets						Clock cycles	Max Freq (MHz)	Design latency (ms)
	SpC 1	Channel width	*SpC 2*	Channel width	*SpC 3*	Channel width			
106	37807	21	20480	28	-	-	1496464	36.34	41.48
150	37807	21	12288	72	-	-	1406384		38.70
200	37807	21	8192	101	-	-	1362776		37.50
250	8805	21	8192	101	12288	69	921738		25.36
300	8805	21	8192	101	8192	93	881232		24.25
350	8805	21	8192	101	8192	93	881232		24.25

Table 7-6 Number of available I/Os and its effect on the performance of the multi-FPGA JPEG decoder

The last two rows in Table 7-6 show that the design latency of the multi-FPGA JPEG decoder does not reduce further when target devices with over 300 available I/Os are used. The graph in Figure 7-23 shows the design latency versus the number of available I/Os in the multi-FPGA JPEG decoder.

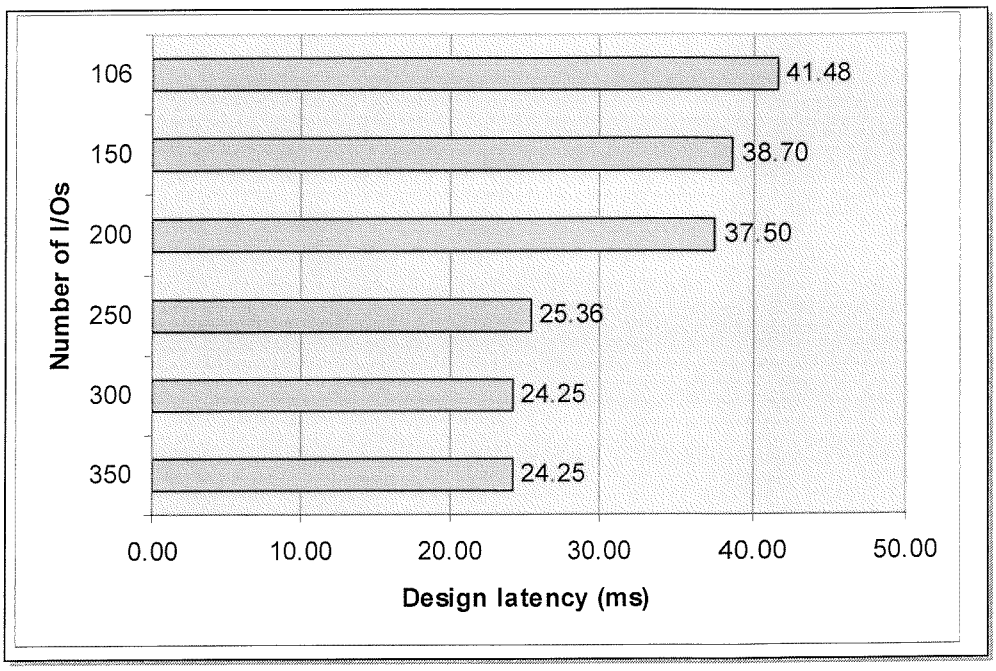


Figure 7-23 Graph of design latency versus the number of available I/Os in the non-pipelined multi-FPGA JPEG decoder

7.4.4 Pipelined multi-FPGA JPEG decoder

The previous sections covered the implementation results and discussion on the non-pipelined multi-FPGA JPEG decoder. This section describes a pipelined version of the JPEG decoder with explicit communication channels (see Section 4.2.2.1) connecting the pipelined stages. Figure 7-24 shows the module call graph representation of the pipelined JPEG decoder core with a total of six subprogram modules and six program modules (p_MOD_5 to p_MOD_10). The main stages of the sequential baseline JPEG decoder are marked under the module call graph in Figure 7-24.

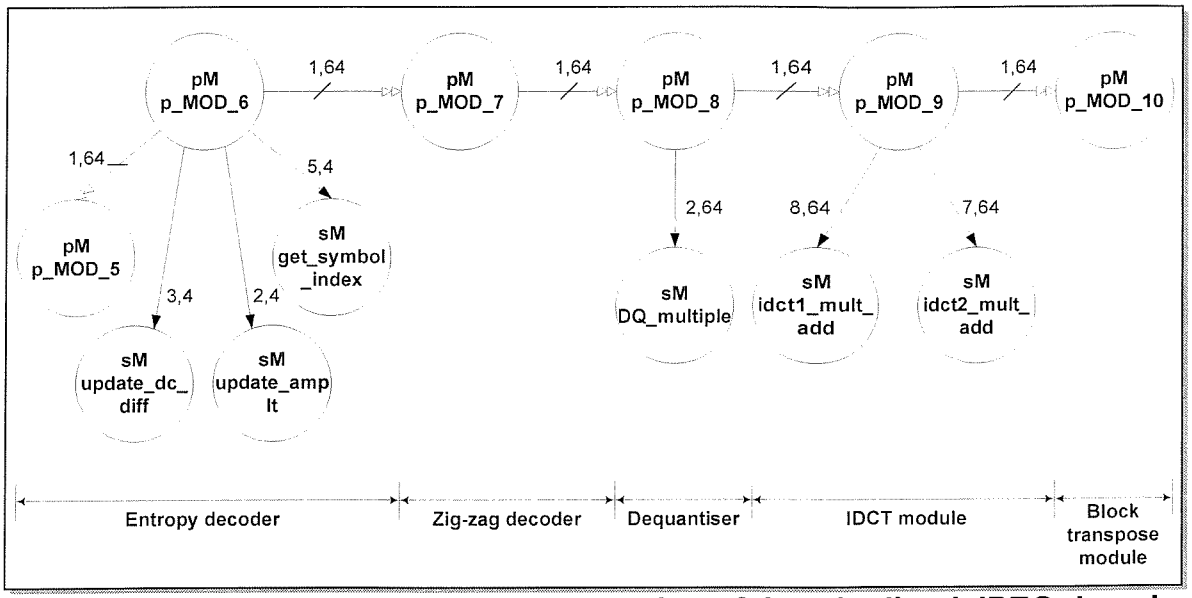


Figure 7-24 Module call graph representation of the pipelined JPEG decoder core

The target technology of the devices used in the following experiments on the pipelined multi-FPGA JPEG decoder core is the Xilinx Spartan 2E FPGA. Table 7-7 lists the four types of Xilinx Spartan 2E target devices used in the multi-FPGA implementation with the XC2S50E and XCS200E as the smallest and largest target devices respectively.

Synthesis results and K-way partitioning results of the pipelined JPEG decoder core and the block transpose module are given in Table 7-8. These results in terms of area and maximum achievable frequency of the final implementation are obtained from the report files generated by post-Xilinx ISE placement and routing phase and not estimates obtained from the MOODS synthesis system. The first and second (shaded) rows show the

synthesis result of single-device implementations given in Table 7-3. The remaining rows show the synthesis results of the pipelined multi-FPGA JPEG decoder core targeting multiple Xilinx Spartan 2E FPGA devices. Post-MOODS synthesis simulation results of the pipelined multi-FPGA JPEG decoder core are given in Appendix B.3.

Xilinx Spartan 2E FPGA devices

Device	Package	Total user I/O	Max. user I/O	Max. area in slices
XC2S50E	TQ144	102	80	768
XC2S100E	TQ144	102	80	1200
XC2S150E	PQ208	146	106	1728
XC2S200E	PQ208	146	106	2352

Table 7-7 Target Xilinx Spartan 2E FPGA technologies

Boards	FPGA	Synthesis results					Two-phase partitioning results	
		Opt. priority	Area in slices	I/O	Freq. (MHz)	AO in slices	Data pkts (initial → final)	Channels (Data widths)
1	s400E	Delay	3639 (75%)	47 (44%)	30.13	–	–	–
1	s200E	Delay	3297* (140%)*	47 (44%)	–	–	–	–
2	s200E	Delay	2134 (90%)	54 (51%)	35.71	221 (6.1%)	67 → 1	1 *ExC* (12)
	s200E		1726 (73%)	35 (33%)	32.41			
3	s200E	Delay	2134 (90%)	54 (51%)	34.02	239 (6.6%)	547 → 2	2 *ExC* (12, 8)
	s150E		1658 (95%)	27 (26%)	35.28			
	s150E		86 (4%)	31 (29%)	84.28			
4	s200E	Delay	2058 (87%)	80 (75%)	32.35	269 (7.4%)	68 → 4	4 *ExC* (11, 11, 12, 8)
	s150E		1658 (95%)	27 (26%)	35.28			
	s100E		106 (8%)	29 (36%)	73.52			
	s100E		86 (7%)	31 (39%)	76.55			
6	s200E	Delay	2004 (85%)	105 (99%)	30.29	486 (13.4%)	80 → 80	4 *ExC* (11, 11, 12, 8) 1 *SpC* (19)
	s150E		1660 (96%)	27 (26%)	35.38			
	s50E		106 (13%)	29 (36%)	73.99			
	s50E		115 (15%)	26 (33%)	60.46			
	s50E		85 (11%)	31 (39%)	89.25			
	s50E		155 (20%)	30 (38%)	59.21			
7	s200E	Delay	1968 (83%)	105 (99%)	30.18	593 (16.3%)	88 → 88	4 *ExC* (11, 11, 12, 8) 1 *SpC* (17)
	s150E		1659 (96%)	27 (26%)	33.43			
	s50E		106 (13%)	29 (36%)	79.96			
	s50E		117 (15%)	24 (30%)	59.60			
	s50E		86 (11%)	31 (39%)	88.22			
	s50E		161 (20%)	32 (40%)	54.08			
	s50E		135 (17%)	24 (30%)	53.39			

Table 7-8 Synthesis results of the pipelined JPEG decoder core

The first three multi-FPGA implementations (MFIs) targeting two to four Xilinx Spartan 2E devices are partitioned across pipelined stages with no external subprogram modules and inter-device data packets are sent through the explicit communication channels (*ExC*s) connecting the pipeline stages. In the last two MFIs targeting the JPEG decoder core onto six and seven devices, subprogram modules are targeted to a different device from their parent calling modules and the resulting inter-device subprogram data packets are sent through the single subprogram communication channel (*SpC*) generated automatically by the multi-FPGA MOODS synthesis system.

The area overheads in terms of slices for pipelined implementations of the multi-FPGA JPEG decoder targeting two to four devices are lower than the non-pipelined MFI given in Table 7-3. One factor contributing to this area overhead reduction is that devices in the 2- to 4-device pipelined MFIs are only connected through *ExCs*. The *ExC* is a dedicated point-to-point communication channel that does not require channel resource arbitration and special communication cells (Section 5.4) to handle inter-device data packet transfers unlike *SpC*. Inter-device data sent through the *ExC* also removes the need for *hardware duplication* (Section 5.5.1), hence reducing the area overheads.

Comparing the 2-device non-pipelined (given in Table 7-3) and pipelined multi-FPGA JPEG decoder core implementation in Table 7-8, the area overhead of the pipelined implementation is smaller by 285 slices (7.8%) and the average maximum achievable frequency (34.06 MHz) of the target devices is slightly lower compared to the non-pipelined implementation (36.57 MHz). The non-pipelined multi-FPGA JPEG decoder core has two *SpC*s to handle the inter-device data packets between the main JPEG core module and external subprogram modules (described in Section 7.4.3) whereas inter-device data packets is sent through an *ExC* with a 12-bit data width in the pipelined implementation.

The area utilisation of some target devices in Table 7-8 are under 20% as the XC2S50E device is the smallest device in the Spartan 2E FPGA family. If the MFI is targeted to a target technology with even smaller and cheaper devices, then the logic resources of the target FPGA devices can be utilised fully, making the design implementation more cost-efficient.

Implementation	No. of target devices	Inter-device data packets (Channel)	Clock cycles	Max Freq (MHz)	Design latency (ms)
pipelined	2	4096 (*ExC 4*)	271696	32.41	8.38
pipelined	3	4096 (*ExC 4*) 4096 (*ExC 5*)	271698	34.02	7.99
pipelined	6	4096 (*ExC 2*) 4096 (*ExC 3*) 4096 (*ExC 4*) 4096 (*ExC 5*) 8387 (*SpC 1*)	795192	30.29	26.25
non-pipelined	2	37897 (*SpC 1*) 20480 (*SpC 2*)	1496464	36.34	41.18
un-partitioned	1	–	178104	30.13	5.91

Table 7-9 Computation clock cycles and inter-device data transfers in the pipelined multi-FPGA JPEG decoder core

The computation clock cycles for decoding the test image (LENA.jpg) with a pipelined multi-FPGA JPEG decoder targeting two, three and six FPGAs are given in Table 7-9. The performance of the un-partitioned (single-device) and non-pipelined multi-FPGA JPEG decoder implementations (see Table 7-4) are given in the last two rows of Table 7-9 for comparison. The maximum frequencies of the pipelined MFIs are the maximum achievable frequency of the slowest FPGA target device given in Table 7-8.

The computation clock cycles of the 2-device and 3-device pipelined MFIs are reduced to a fraction (approximately 1/5) of the computation clock cycles needed by the non-pipelined version. Subprogram modules are mapped to the same target device as their parent calling modules in the 2- and 3-device pipelined MFIs and hence inter-device data are sent via the explicit communication channels (*ExC*s) connecting the pipeline stages which are targeted onto different devices. A multi-FPGA implementation with only *ExC*(s) removes the delay associated with the arbitration and enabling of the tri-stated shared subprogram communication channel, thus reducing the number of computation clock cycles needed to decode the test image significantly.

The computation clock cycles of the 6-device pipelined MFI is reduced to almost half the number of computation clock cycles in the non-pipelined implementation. The 6-device pipelined MFI has two external subprogram modules (update_dc_diff and DQ_multiple) transferring inter-device data packets through subprogram communication channel 1 (*SpC 1*) with a 19-bit wide data width. The 6-device pipelined MFI has a total of 795192 computation clock cycles and a design latency of 26.25 ms. It is possible for the non-

pipelined implementation to reduce its latency by targeting a larger FPGA device as discussed in Section 7.4.3. The 2-device non-pipelined MFI with two external subprogram modules (given in Table 7-5) has a total of 629882 computation clock cycles, a maximum achievable frequency of 36.34 MHz and a resulting design latency of 17.33 ms.

The area overheads and design latencies of the JPEG decoder core in the multi-FPGA implementations (MFIs) are plotted on the graph shown in Figure 7-25. These results show all three pipelined MFIs of the JPEG decoder core have better performances (in terms of area overheads and design latencies) than the 2-device non-pipelined MFI. The 2- and 3-device MFIs have area overheads of under 7% and design latencies of about 8 ms, which is about 35% more than the design latency (5.91 ms) of the un-partitioned (single-device) implementation.

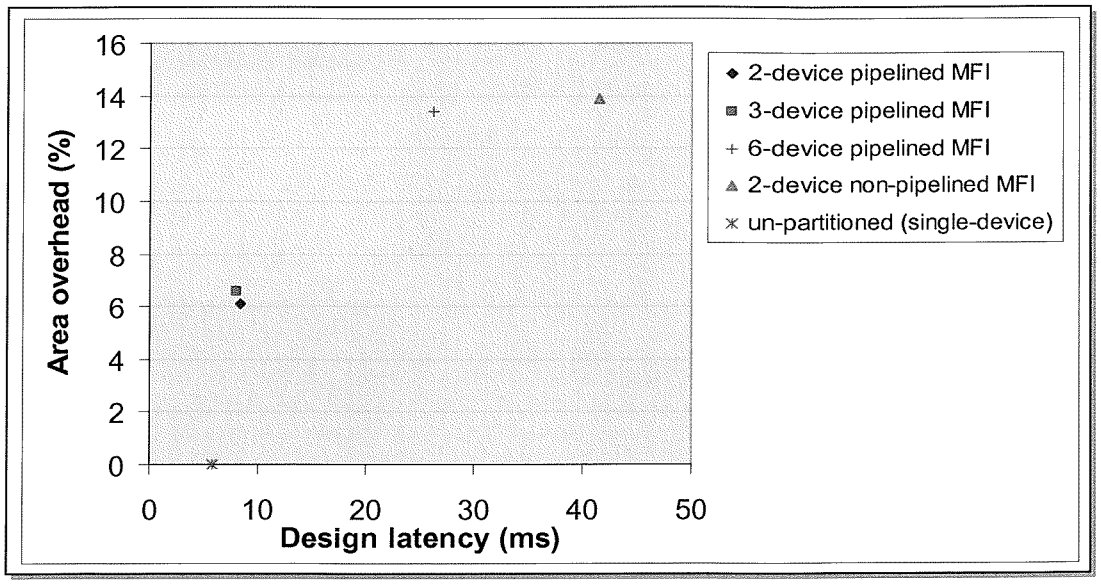


Figure 7-25 Area overhead and design latency of pipelined and non-pipelined multi-FPGA JPEG decoder core

The experiments in this section show the synthesis of a large complex behavioural design (a behavioural JPEG decoder core with over 2000 lines of VHDL code, and MOODS synthesis run time[1] of up to an hour) into a pipelined multi-FPGA system that can achieve performances comparable to single-device implementations. Synthesis of a large

[1] This is the synthesis run time of a single-device implementation using the original version of MOODS without the multi-FPGA synthesis enhancement.

behavioural design into a multi-FPGA system poses difficult partitioning questions (outlined in Section 1.1) that need to be answered. Solutions to how best to partition a design are not immediately obvious to the user and this can be a challenge to the user if the design contains a large number of modules which can be partitioned, leading to a large number of possible partitioning solutions. The fully automated multi-FPGA synthesis design flow in MOODS answers these questions by generating multi-FPGA systems with asynchronous communications automatically and as transparently to the user as possible. This reduces the design time and effort required by the user.

As with the examples in Chapter 6, the multi-FPGA synthesis run times of the JPEG decoder remain similar to the run time of a single-device implementation using an original version of MOODS without the multi-FPGA synthesis enhancements (i.e. run time approximates for pipelined and single-device implementations are close to 1 hour).

7.5 Summary

The successful implementation of the multi-FPGA JPEG decoder project described in this chapter has demonstrated the automated synthesis and optimisation of a large complex system targeting a multi-FPGA implementation. MOODS, with the two-phase K-way partitioning and design profiling, has partitioned and optimised a single large behavioural VHDL design into a design with multiple partitions, and allowed the targeting of heterogeneous FPGA devices in a multi-FPGA system.

The user now has the choice of targeting a large behavioural design onto multiple smaller devices without having the need to get a larger and more costly target FPGA device if the design requirements are met with a multi-FPGA system. The user would be able to use existing FPGA devices or a number of FPGA development boards configured into a multi-FPGA system for design prototyping. This saving in design cost and flexibility in using existing development boards with a collection of smaller devices would not be possible otherwise if a single large behavioural design is not partitioned.

The automated insertion of asynchronous subprogram communication subsystems (comprising of communication cells and arbiters) enables modules in independently clocked domains to transfer data asynchronously through shared bi-directional subprogram communication channels (*SpC*s). The pipelined multi-FPGA JPEG decoder demonstrated the use of explicit communication channels (*ExC*s) connecting the pipelined stages in the JPEG decoder core to improve the performance of the non-pipelined multi-FPGA implementation. The asynchronous communication channels have built-in synchronisation and self-scheduling properties which provide safe communication of inter-device data in the multi-FPGA system. Therefore, the user can concentrate on the behaviour of the design and not the complexities of how the target devices can safely communicate.

Chapter 8

Conclusions and future work

The partitioning enhancement to the MOODS synthesis system reported in this work has provided a high-level synthesis system to synthesise and automatically generate a multi-FPGA system composed of heterogeneous re-configurable devices from a single VHDL description. The K-way partitioning algorithm and the communication subsystem optimisation algorithm optimise the multi-FPGA system in terms of design latency across FPGA boundaries, while satisfying the area and I/O constraints of the target devices. Target device information (area in slices and number of I/Os) and design activity profile is used to guide the partitioning algorithm are fed into MOODS. The integration of design activity profile and the K-way partitioning algorithm are covered in Chapter 4.

During synthesis, explicit communication channels (*ExCs*) or subprogram communication subsystems are automatically inserted into the multiple structural outputs of the design. *ExC* provides a dedicated point-to-point communication channel connecting pipelined stages in the pipelined multi-FPGA design. This supports the Communicating Sequential Processes [111, 112] paradigm, which encourages modular design. Channel handshaking ensures that the pipelines stages will work irrespective of the operation execution time of individual stages in the asynchronous pipeline. The communication subsystem provides an asynchronous subprogram communication channel (*SpC*) for transferring data packets between modules which exist in different clock domain devices. This extends the multi-FPGA synthesis capability to support partitioning of VHDL subprograms and functions in the VHDL hierarchical structure (Section 2.2.3). Communication cells (transmit and receive cells) and arbiters are the basic building elements of the subprogram communication subsystem. Transmit and receive cells deal with the two-phase handshaking of inter-device data across shared bi-directional communication channel(s) and the communication channel is optimised with respect to the I/O constraint of the target

FPGA devices. Details on the generation of communication subsystem(s) and hardware duplication are given in Chapter 5.

Experimental and simulation results of the pipelined multi-FPGA implementations of the VHDL examples in Chapter 6 and the pipelined multi-FPGA JPEG decoder in Chapter 7 show that the pipelined implementations only incur a fraction of the area overheads and design latencies compared to the non-pipelined multi-FPGA versions. Area overhead and the design latency are used as the metrics for evaluating the quality of the multi-FPGA implementations in these chapters. System throughput is another possible metric as throughput measures the synthesised design's ability to handle a high volume of transactions. However, in many applications, design latency is more suitable as it measures the time it takes the synthesised design to perform any given transaction from start to finish. The area overheads for most of the pipelined multi-FPGA implementations are under 10% and the lowest area overhead of 3% for the delay-optimised multi-FPGA pipelined implementation of the inverse discrete cosine transform example; together with increase in the average maximum achievable frequencies of target devices in all the pipelined examples including the pipelined JPEG decoder. In the case of the quadratic equation solver example, the design latency of the pipelined implementation is lower than the un-partitioned single-device implementation.

Results presented in Chapters 6 and 7 show that pipelined multi-FPGA systems can be synthesised to achieve performances comparable to single-device implementations. With the multi-FPGA synthesis enhancement, it is now possible to synthesise a large behavioural design and target the partitioned design onto multiple smaller (existing) devices without having the need to get a larger and more costly target FPGA device if the design requirements are met with a multi-FPGA system. This saving in design cost and flexibility in using existing development boards with a collection of smaller devices would not be possible otherwise if a single large behavioural design is not partitioned.

The multi-FPGA synthesis run times remain virtually unaffected for all the VHDL examples in Chapter 6 and the JPEG decoder core in Chapter 7 compared to the run times of single-device implementations. The multi-FPGA synthesis enhancement, with the fast time-to-market, efficient and fast design space exploration advantages of a high-level

synthesis environment, enables the rapid realisation of multi-FPGA systems with asynchronous communications.

The asynchronous communication channels in the multi-FPGA systems offers a number of benefits to the user:

- The first benefit is the option to trade off performance in I/O limited target devices (i.e. allows multiple external subprogram modules to share a common channel and sending of multiple data packets over an asynchronous subprogram communication channel of a smaller data width).

- The second benefit is the temporal independence between target devices as each board level target device is viewed an independent locally clocked processing unit with asynchronous communication channels, reducing clock skew problems in a large design.

- The asynchronous communication channels have built-in synchronisation and self-scheduling properties which provide safe communication of inter-device data in the multi-FPGA system. Therefore, the user can concentrate on the behaviour of the design and not the complexities of how the target devices can safely communicate.

The work presented together with a hardware demonstrator has demonstrated a fully functional behavioural multi-FPGA synthesis tool. To the best of our knowledge, high-level synthesis of multi-FPGA systems with asynchronous communication channels crossing clock domains is explicitly automated for the first time. There is scope for improvement in the currently implemented system, both with the multi-FPGA partitioning process and with the asynchronous communication mechanism. A number of suggested extensions are described within this chapter, which could form the basis for future work.

8.1 Future work

Experience gained from using MOODS and extending the existing MOODS synthesis system for multi-FPGA synthesis has brought to light some of the limitations of the multi-FPGA synthesis in MOODS and several extensions remain to be addressed:

8.1.1 Shared memory elements

The multi-FPGA synthesis enhancement in MOODS is not able to handle access of shared memory blocks such as ROM and RAM across target devices. Currently, the ICODE process and subprogram modules accessing memory elements declared in the VHDL architecture have to be mapped to the same target partition (device) as the program module (Section 2.6.2). This restriction may result in a larger target device for the partition with the shared memory block and reducing the configurations of target devices in the multi-FPGA implementation. However, the multi-FPGA synthesis tool does support memory blocks (ROM and RAM) *local* to the process or subprogram modules as these memory elements are declared within the scope of the VHDL process or subprogram. A shared memory controller which handles the data coherence and resource arbitration is a possible extension to support shared memory in a multi-FPGA system. This memory controller can be mapped to one of the existing target devices in the synthesised multi-FPGA design or a separate target device with a large memory element. The downside to this extension is the Address/Data lines to the memory elements and the control signals (from target devices) to the memory controller would utilise more I/O resources of target devices and the design latency is likely to increase due to inter-device memory accesses.

8.1.2 Explicit communication channel structures

The asynchronous explicit communication channels connecting the implied pipeline stages in the multi-FPGA implementation can be extended further to allow more complex channels than the unidirectional point-to-point structure described in this work. Linear pipeline stages have only a single input and single output channel, whereas non-linear pipeline stages can have multiple input and output channels. A *join* is a pipeline stage with

multiple input channels and a single output channel. A *fork* is a pipeline stage with one input channel and multiple output channels. Non-linear asynchronous pipeline structures [159], including join, fork, and more complex configurations in which channels are conditionally read and /or written can be used to build more complex systems. Another possible configuration is to create an explicit communication channel that is able to send and/or receive multiple packets of data determined by the data width of the channel. Trade-offs between latency, area, and I/O resources, taking into account the design activity profile of modules in the design would be performed by the synthesis tool to determine the optimum data width of the channel.

8.1.3 Integrating partitioning exploration with the MOODS optimisation process

The two-phase partitioning exploration is currently not integrated with the MOODS optimisation process but it does allow the user to re-run the MOODS optimisation stage after examining the partitioned design. It is possible to *relax* or *tighten* the schedule of the modules and iteratively improve the multi-FPGA solution using the current partitioning solution to guide the MOODS optimisation process. A similar approach in SPARCS (Section 3.3.2) performs an iterative area/latency exploration of blocks of operations where the schedule of a block is either relaxed or tightened such that the design constraints are best satisfied.

The two-phase K-way partitioning approach (Section 4.4.1) in MOODS performs K-way partitioning on the optimised ICODE modules and optimises the subprogram communication channel(s) if the design contains ICODE subprogram modules. The main aim of the K-way partitioning algorithm is to minimise the number of inter-device (or cross-domain) data transfers by grouping modules and subprogram modules with their corresponding calling modules, taking into consideration the utilisation of device area and I/Os. The MOODS synthesis core performs scheduling, allocation and module binding according to the user-defined optimisation objectives. MOODS performs multiple simple optimisation transformations, adjusting the scheduling of the control state nodes in the control path, and the allocation and binding of data path nodes in the data path.

ICODE process modules have their own control paths controlling the data path units within the module and explicit communication channels introduce an implied pipeline structure whereby channels connect the process modules (pipelines stages) in a design. Channel handshaking ensures that the pipelines stages will work irrespective of the operation execution time of individual stages in the asynchronous pipeline. This allows the schedule of a process module (pipeline stage) to be either relaxed or tightened such that the number of external modules are reduced, hence reducing the number of inter-device communications. Relaxing (increasing) the schedule length could reduce the area of a partition and increase the latency of the pipeline stage and tightening the schedule works vice versa.

8.1.4 Target Architecture

At present, the MOODS multi-FPGA synthesis system targets a multi-FPGA system at the board-level. This board-level architecture allows an arbitrary number of heterogeneous development boards to be connected up to form a multi-FPGA system. The *K*-way partitioning algorithm uses the area and I/O information for each target device in the assignment of modules to *K*-partitions, where *K* is the number of target development boards available. Using the same target device information, the partitioned structural output can be targeted to a single board with multiple re-configurable (FPGA) devices, having fixed interconnects between these devices. The FPGA devices can be treated as individual locally clocked processing units communicating asynchronously using the communication channels described in Chapter 4, or these devices can be clocked synchronously from a single global system clock. For a single global clock architecture, the double buffer synchronisers used for data synchronisation over multiple clock domains are no longer required in the communication cells since data communications between devices are now in a single clock domain. It is possible to target a partition design onto multiple boards, each having a single re-configurable device, or a single board with multiple re-configurable devices, or a combination of both as illustrated in Figure 8-1.

Programmable interconnection resources or Field-Programmable Interconnect Devices (FPIDs) are commonly found in multi-FPGA system to provide flexible routing

capabilities between the FPGA devices. One of the most commonly used routing architectures is the partial crossbar architecture [92, 160]. The programmable interconnect devices can be used to connect partitions where high performance is required. A target architecture with FPIDs, together with the I/O multiplexing packet-based communication channels might improve the overall performance of the generated multi-FPGA system. This can be formulated as an optimisation problem, where trade-offs between performance and I/O utilisation are performed, whilst satisfying design constraints such the number of FPIDs, programmable pins and FPGA area and I/Os available.

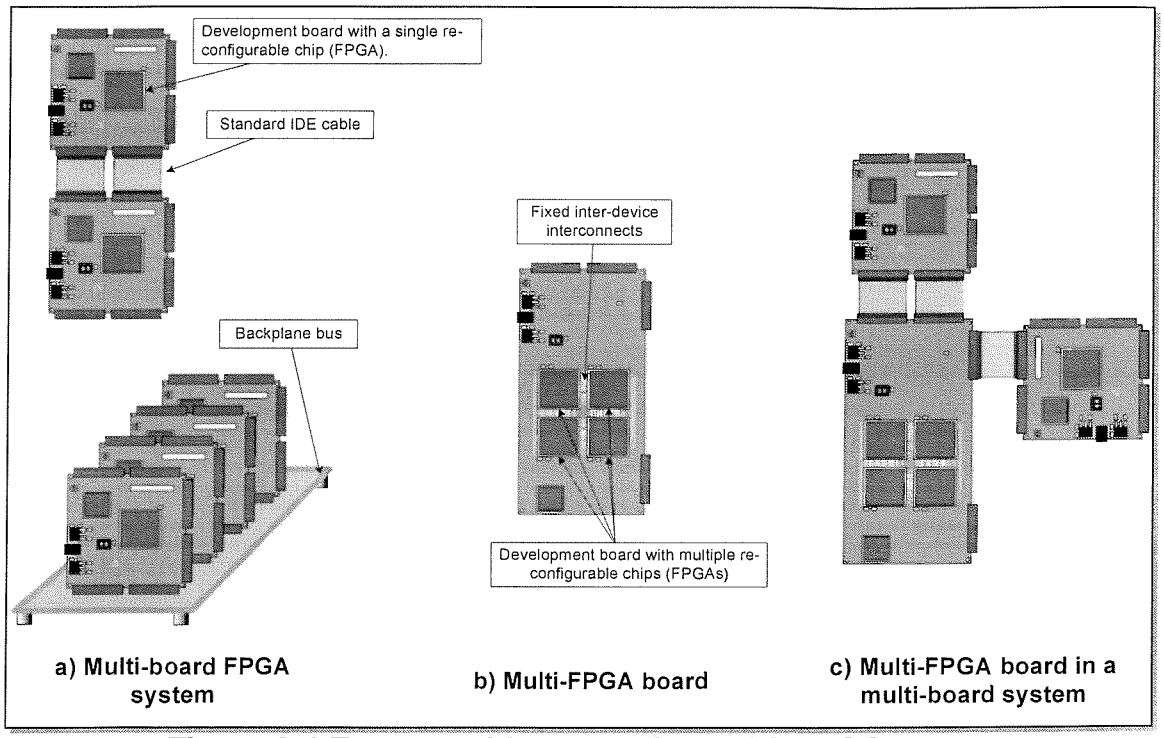


Figure 8-1 Target architectures for multi-FPGA system

Utilising FPIDs in the target multi-FPGA system, the MOODS multi-FPGA synthesis system can target a flexible and modular architecture, which would provide a good platform for prototyping and allow easy extension of the target architecture to suit the size of the synthesised design.

References

1. Johannes, F.M., "Partitioning of VLSI circuits and systems", Proceedings of the Design Automation Conference, 1996, pp. 83-87.

2. Wolf, W., "A decade of hardware/software codesign", Computer, Vol. 36, No. 4, April 2003, pp. 38-43.

3. "International Technology Roadmap for Semiconductors (2004 Update) - Design", 2004. http://public.itrs.net

4. Shukla, S.K.- Pixley, C.- Smith, G., "Guest Editors' Introduction: The True State of the Art of ESL Design", Design & Test of Computers, IEEE, Vol. 23, No. 5, May 2006, pp. 335-337.

5. Bacchini, F., et al., "Building a common ESL design and verification methodology - is it just a dream?" Proceedings of the Design Automation Conference (DAC2006), 2006, pp. 370-371.

6. "Catapult^{TM} C Synthesis", Mentor Graphics, 2004. www.mentor.com

7. "Bluespec Compiler", Bluespec Inc., 2004. www.bluespec.com

8. "Cynthesizer", Forte Design Systems, 2004. www.forteds.com

9. "IEEE Standard VHDL Reference Manual, IEEE Std 1076-2002", IEEE, 2002.

10. "IEEE Standard VHDL Reference Manual, IEEE Std 1076-1987", IEEE, 1987.

11. "IEEE Standard VHDL Reference Manual, IEEE Std 1076-1993", IEEE, 1993.

12. Rushton, A., "VHDL for Logic Synthesis", 2nd ed, John Wiley and Sons, 1999, ISBN: 047198325X.

13. Yarom, I.- Glasser, G., "SystemC Opportunities in Chip Design Flow", Proceedings of the 11th IEEE International Conference on Electronics, Circuits and Systems (ICECS 2004), 2004, pp. 507-510.

14. "IEEE Standard SystemC Language Reference Manual, IEEE Std 1666-2005", IEEE, 2005.

15. Grötker, T., et al., "System Design with SystemC", Kluwer Academic Publishers, 2002, ISBN: 1402070721.

16. "IEEE Standard Description Language Based on the Verilog Hardware Description Language, IEEE Std 1364-1995", 1995.

17. "IEEE Standard Description Language Based on the Verilog Hardware Description Language, IEEE Std 1364-2001", IEEE, 2001.

18. Sutherland, S., "The IEEE Verilog 1364-2001 Standard - What's New, and Why You Need It", Proceedings of the 9th Annual International HDL Conference and Exhibition (HDLCon2000), 2000.

19. Fitzpatrick, T., "System Verilog for VHDL Users ", Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'04), 2004, pp. 1334-1339 (Vol.2).

20. Rich, D.I., "The evolution of systemverilog", IEEE Design & Test of Computers, Vol. 20, No. 4, July-August 2003, pp. 82 - 84.

21. "IEEE Standard VHDL Reference Manual, IEEE Std 1076a-2000", IEEE, 2000.

22. Aho, A.V.- Ullman, D.D., "Principles of Complier Design", Addison-Wesley, 1977, ISBN: 0-201-00022-9.

23. Gajski, D.D.- Ramachandran, L., "Introduction to High-Level Synthesis", IEEE Design & Test of Computers, Vol. 11, No. 4, October-December 1994, pp. 44-54.

24. Eles, P., et al., "Compiling VHDL into a high-level synthesis design representation", Proceedings of the EURO-DAC 92: European Design Automation Conference, 1992, pp. 604-609.

25. Eles, P.- Kuchcinski, K.- Peng, Z., "System Synthesis with VHDL", Kluwer Academic Publishers, 1998, ISBN: 0-79238-082-7.

26. Murata, T., "Petri Nets: Properties, Analysis and Applications", Proceedings of the IEEE, Vol. 77, No. 4, April 1989, pp. 541-580.

27. Walker, R.A.- Chaudhuri, S., "Introduction to the Scheduling Problem", IEEE Design & Test of Computers, Vol. 12, No. 2, June 1995, pp. 60-69.

28. Camposano, R., "From Behavior to Structure: High-Level Synthesis", IEEE Design & Test of Computers, Vol. 7, No. 5, October 1990, pp. 8-19.

29. Parker, A.C.- Pizarro, J.T.- Mlinar, M., "MAHA: a program for datapath synthesis", Proceedings of the Design Automation Conference, 1986, pp. 461-466.

30. Paulin, P.G.- Knight, J.P., "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", IEEE transaction on Computer Aided Design, Vol. 8, No. 6, June 1989, pp. 661-679.

31. Peng, Z.- Kuchcinski, K., "Automated Transformation of Algorithms into Register-Transfer Level Implementations", IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Vol. 13, No. 2, February 1994, pp. 150-166.

32. Williams, A.C., "A Behavioural VHDL Synthesis System using Data Path Optimisation", PhD Thesis, University of Southampton, 1997.

33. Paulin, P.G.- Knight, J.P., "Scheduling and binding algorithms for high-level synthesis", Proceedings of the Design Automation Conference, 1989, pp. 1-6.

34. Kurdahi, F.J.- Parker, A.C., "REAL: a program for REgister ALlocation", Proceedings of the Design Automation Conference, 1987, pp. 210-215.

35. Baker, K.R.- Currie, A.J.- Nichols, K.G., "Multiple Objective Optimisation in a Behavioural Synthesis System", IEE Proceedings - G, Vol. 140, No. 4, August 1993, pp. 253-260.

36. Williams, A.C.- Brown, A.D.- Zwolinski, M., "Simultaneous optimisation of dynamic power, area and delay in behavioural synthesis", IEE Proceedings on Computers and Digital Techniques, Vol. 147, No. 6, November 2000, pp. 383-390.

37. De Micheli, G., "Synthesis and Optimization of Digital Circuits", McGraw Hill International Editions, 1994, ISBN: 0070163332.

38. McFarland, M.C.- Parker, A.C.- Camposano, R., "Tutorial on High-Level Synthesis", Proceedings of the 25th ACM/IEEE Design Automation Conference, 1988, pp. 330-336.

39. "MOODS Internals v1.0", University of Southampton, July 2001.

40. Camposano, R.- Saunders, L.F.- Tabet, R.M., "VHDL as input for high-level synthesis", IEEE Design & Test of Computers, Vol. 8, No. 1, March 1991, pp. 43-49.

41. Ramachandran, L., et al., "Semantics and synthesis of signals in behavioral VHDL", Proceedings of the EURO-DAC 92: European Design Automation Conference, 1992, pp. 616-621.

42. "MOODS VHDL Style Guide Version 1.2 (alpha)", LME Design Automation Ltd., August 2001.

43. Kirkpatrick, S.- Gelatt, C.D.- Vecchi, M.P., "Optimization by Simulated Annealing", Science, Vol. 220, No. 4598, May 1983, pp. 671-680.

44. Metropolis, N., et al., "Equation of State Calculations by Fast Computing Machines", Journal of Chemical Physics, Vol. 21, No. 6, June 1953, pp. 1087-1092.

45. Hauck, S., "The roles of FPGAs in reprogrammable systems", Proceedings of the IEEE, Vol. 86, No. 4, April 1998, pp. 615-638.

46. "The Programmable Logic Data Book", Xilinx Inc, 2000.

47. Cook, S.A., "The complexity of theorem-proving procedures", Proceedings of the Third Annual ACM symposium on Theory of computing, 1971, pp. 151 - 158.

48. Sherwani, N.A., "Algorithms for VLSI Physical Design Automation", 3rd ed, Kluwer Academic Publishers, 1999, ISBN: 0792383931.

49. Sait, S.M.- Youssef, H., "VLSI Physical Design and Automation: Theory and Practice", McGraw-Hill, 1994, ISBN: 0-07-707742-3.

50. Kernighan, B.W.- Lin, S., "An Efficient Heuristic Procedure for Partitioning of Electrical Circuits", Bell Systems Technical Journal, Vol. 49, No. 2, February 1970, pp. 291-307.

51. Fiduccia, C.M.- Mattheyses, R.M., "A Linear-Time Heuristic for Improved Network Partitions", Proceedings of the Design Automation Conference, 1982, pp. 241-247.

52. Krishnamurthy, B., "An Improved Min-Cut Algorithm For Partitioning VLSI Networks", IEEE Transaction on Computers, Vol. C-33, No. 5, May 1984, pp. 438-446.

53. Huang, D.J.- Kahng, A.B., "Multi-way System Partitioning into a Single Type or Multiple Types of FPGA", Proceedings of the International Symposium on Field Programmable Gate Arrays, 1995, pp. 104-145.

54. Hauck, S.- Borriello, G., "An Evaluation of Bipartitioning Techniques", Proceedings of the Chapel Hill Conference on Advanced Research in VLSI, 1995, pp. 383-402.

55. Cong, J.- Wu, C., "Global Clustering-Based Performance-Driven Circuit Partitioning", Proceedings of the International Symposium on Physical Design, 2002, pp. 149-154.

56. Cong, J.- Romesis, M.- Xie, M., "Optimality, Scalability and Stability Study of Partitioning and Placement Algorithms", Proceedings of the International Symposium on Physical Design, 2003, pp. 88 - 94.

57. Dutt, S.- Deng, W., "VLSI Circuit Partitioning by Cluster-Removal using Iterative Improvement Techniques", Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, 1996, pp. 194-200.

58. Kuznar, R.- Brglez, F.- Zajc, B., "Cost Minimization of Partitions into Multiple Devices", Proceedings of the Design Automation Conference, 1993, pp. 315-320.

59. Kuznar, R.- Brglez, F., "PROP: A Recursive Paradigm for Area-Efficient and Performance Oriented Partitioning of large FPGA Netlists", Proceedings of the IEEE/ACM International Conference on Computer Aided Design, 1995, pp. 644-649.

60. Holland, J.H., "Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence", Reprint ed, 1992, ISBN: 0262581116.

61. Vahid, F.- Gajski, D.D., "Clustering for improved system-level functional partitioning", Proceedings of the Eighth International Symposium on System Synthesis, 1995, pp. 28-33.

62. Vahid, F.- Gajski, D.D., "Closeness metrics for system-level functional partitioning", Proceedings of the EURO-DAC 95: European Design Automation Conference, 1995, pp. 328-333.

63. Vahid, F.- Le, T.D.M.- Hsu, Y.C., "Functional Partitioning Improvements over Structural Partitioning for Packaging Constraints and Synthesis-tool Performance", ACM Transactions on Design Automation of Electronic Systems, Vol. 6, No. 2, April 1998, pp. 181-208.

64. Vahid, F., "A Three-Step Approach to the Functional Partitioning of Large Behavioral Processes", Proceedings of the International Symposium on System Synthesis, 1998, pp. 152-157.

65. Vahid, F., "Procedure cloning: a transformation for improved system-level functional partitioning", ACM Transactions on Design Automation of Electronic Systems, Vol. 4, No. 1, January 1999, pp. 70-96.

66. Kumar, N.- Srinivasan, V.- Vemuri, R., "Hierarchical Behavioural Partitioning for Multicomponent Synthesis", Proceedings of the European Design Automation Conference with EURO-VHDL, 1996, pp. 212-219.

67. Lakshmikanthan, P., et al., "Behavioral Partitioning with Synthesis for Multi-FPGA Architectures under Interconnect, Area, and Latency Constraints", Proceedings of the 7th Reconfigurable Architectures Workshop (RAW 2000), 2000, pp. 924-931.

68. Govindarajan, S., et al., "A Technique for Dynamic High-level Exploration During Behavioral Partitioning for Multi-device Architectures", Proceedings of the 13th International Conference on VLSI Design, 2000, pp. 212-219.

69. Fang, W.-J.- Wu, A.C.-H., "Performance-Driven Multi-FPGA Partitioning Using Functional Clustering and Replication", Proceedings of the 35th Design Automation Conference (DAC), 1998, pp. 283-286.

70. Fang, W.-J.- Wu, A.C.-H., "Integrating HDL Synthesis and Partitioning for Multi-FPGA Designs", IEEE Design & Test of Computers, Vol. 5, No. 2, April-June 1998, pp. 65-72.

71. Fang, W.-J.- Wu, A.C.-H., "Multi-Way FPGA Partitioning by Fully Exploiting Design Hierarchy", ACM Transactions on Design Automation of Electronic Systems (TODAES), Vol. 5, No. 1, January 2000, pp. 34-50.

72. Duncan, A.A.- Hendry, D.C.- Gray, P., "An Overview of the COBRA-ABS High Level Synthesis System for Multi-FPGA Systems", Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, 1998, pp. 106-115.

73. Krupnova, H.- Abbara, A.- Saucier, G., "A hierarchy-driven FPGA partitioning method", Proceedings of the Design Automation Conference, 1997, pp. 522-525.

74. Kennings, A.- Frazer, M., "Circuit clustering and its effects on a multi-way circuit partitioning heuristic", Proceedings of the IEEE 1997 Canadian Conference on Electrical and Computer Engineering, 1997, pp. 15-18.

75. Vemuri, R., "Genetic Algorithms for Partitioning, Placement, and Layout Assignment for Multi-chip Modules", PhD Thesis, University of Cincinnati, 1994.

76. Lawrence, D., "Handbook of genetic algorithms", Van Nostrand Reinhold, New York, 1991, ISBN: 0442001738.

77. Pratibha, P., et al., "An Evolutionary Algorithm for Automatic Spatial Partitioning in Reconfigurable Environments", Proceedings of the Third Mexican International Conference on Artificial Intelligence, 2004, pp. 735-745.

78. Hidalgo, J.I., et al., "Multi-FPGA Systems Synthesis by Means of Evolutionary Computation", Proceedings of the Genetic and Evolutionary Computation Conference, 2003, pp. 2109-2120.

79. "Design Pilot™", Aptix Corporation, 2003. www.aptix.com

80. "Auspy Partition System II", Auspy Development Inc., 2000. www.auspy.com

81. "Certify", Synplicity, 2003. www.synplicity.com

82. Duncan, A.A.- Hendry, D.C.- Gray, P., "The COBRA-ABS high-level synthesis system for multi-FPGA custom computing machines", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 9, No. 1, February 2001, pp. 218-223.

83. Ouaiss, I., et al., "An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures", Proceedings of the 5th Reconfigurable Architectures Workshop (RAW), 1998, pp. 31-36.

84. Kumar, N., "High-Level VLSI Synthesis For Multichip Designs", PhD Thesis, University of Cincinnati, 1994.

85. Fang, W.-J.- Wu, A.C.-H., "A hierarchical functional structuring and partitioning approach for multiple-FPGA implementations", Proceedings of the International Conference on Computer-Aided Design (ICCAD), 1996, pp. 638 - 643.

86. Gajski, D.D., et al., "SpecSyn: an environment supporting the specify-explore-refine paradigm for hardware/software system design", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 6, No. 1, March 1998, pp. 84-100.

87. Vahid, F., "Partitioning sequential programs for CAD using a three-step approach", ACM Transactions on Design Automation of Electronic Systems, Vol. 7, No. 3, July 2002, pp. 413-429.

88. Bringmann, O.- Menn, C.- Rosenstiel, W., "Target Architecture Oriented High-Level Synthesis for Multi-FPGA Based Emulation", Proceedings of the Design, Automation and Test in Europe (DATE), 2000, pp. 326-332.

89. Duncan, A.A.- Hendry, D.C., "High-level synthesis of DSP datapaths by global optimisation of variable lifetimes", IEE Proceedings on Computers and Digital Techniques, Vol. 142, No. 3, May 1995, pp. 215-224.

90. Duncan, A.A.- Hendry, D.C., "Architectural Issues for High Level Synthesis of DSP Algorithms onto Multiple FPGAs", Proceedings of the 4th Reconfigurable Architectures Workshop (RAW), 1997, pp. 73-76.

91. Jain, S.C.- Kumar, S.- Kumar, A., "Evaluation of various routing architectures for multi-FPGA boards", Proceedings of the Thirteenth International Conference on VLSI Design, 2000, pp. 262-267.

92. Khalid, M.A.S.- Rose, J., "A novel and efficient routing architecture for multi-FPGA systems", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 8, No. 1, February 2000, pp. 30-39.

93. Hauck, S., "Multi-FPGA systems", PhD Thesis, University of Washington, 1995.

94. Babb, J., et al., "Logic emulation with virtual wires", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 16, No. 6, June 1997, pp. 609-626.

95. Vahid, F., "I/O and Performance Tradeoffs with the FunctionBus during Multi-FPGA Partitioning", Proceedings of the International Symposium on Field-Programmable Gate Arrays, 1997, pp. 27-34.

96. Chambers, P., "The Ten Commandments of Excellent Design", in *Electronic Design*, 1997. pp. 33-40.

97. "Application Note XAPP077 - Metastability Considerations", Xilinx Inc, 1997.

98. "Application Note XAPP094 - Metastable Recovery", Xilinx Inc, 1997.

99. Clark, I.G., *Metastability Bibliography*, 1997-2004.

100. "Application Note XAPP094 - Metastable Recovery in Vertex-II Pro FPGAs", Xilinx Inc, 2005.

101. Beerel, P.A., "Asynchronous circuits: an increasingly practical design solution", Proceedings of the International Symposium on Quality Electronic Design, 2002, pp. 367-372.

102. Peeters, A.M.G., "Single-Rail Handshake Circuits", PhD Thesis, Eindhoven University of Technology, 1996.

103. Verhoeff, T.P., *Encyclopedia of Delay-Insensitive Systems*, 1995-1998, Eindhoven University of Technology.

104. Sutherland, I.E., "Micropipelines", Communications of the ACM, Vol. 32, No. 6, June 1989, pp. 720-738.

105. Woods, J.V., et al., "AMULET1: An Asynchronous ARM Microprocessor", IEEE transactions on computers, Vol. 46, No. 4, April 1997, pp. 385-398.

106. Cummings, C.E., "Simulation and Synthesis Techniques for Asynchronous FIFO Design", Proceedings of the Synopsys Users Group Conference (SNUG), 2002.

107. Cummings, C.E., "Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons", Proceedings of the Synopsys Users Group Conference (SNUG), 2002.

108. "DS232(v0.2) - Asynchronous FIFO v5.1", Xilinx Inc, March 2003.

109. Sacker, M., "Asynchronous and Multiple Clock Domain Synthesis for Large Scale Systems", PhD Thesis, University of Southampton, 2005.

110. Saifhashemi, A.- Beerel, P.A., "High Level Modeling of Channel-Based Asynchronous Circuit Using Verilog", Proceedings of the Communicating Process Architectures (CPA2005), 2005, pp. 275-287.

111. Hoare, C.A.R., "Communicating Sequential Processes", Communications of the ACM, Vol. 21, No. 8, August 1978, pp. 666-677.

112. Hoare, C.A.R., "Communicating Sequential Processes", Prentice-Hall, 1985, ISBN: 0131532715.

113. Self, R.P.- Fleyry, M.- Downton, A.C., "Design methodology for construction of asynchronous pipelines with Handel-C", IEE Proceedings - Software, Vol. 150, No. 1, February 2003, pp. 39-47.

114. Michalewicz, Z.- Fogel, D.B., "How to Solve It: Modern Heuristics", 1st ed, Springer-Verlag, 2000, ISBN: 3540660615.

115. Harel, D., "Algorithms: The spirit of computing", 2nd ed, Addison-Wesley, 1992, ISBN: 0201504014.

116. Saito, H., et al., "Design of Asynchronous Controllers with Delay Insensitive Interface", IEICE TRANS. FUNDAMENTALS, Vol. E85-A, No. 12, December 2002, pp. 2577-2585.

117. Gasteier, M.- Glesner, M., "Bus-based communication synthesis on system level", ACM Transactions on Design Automation of Electronic Systems (TODAES), Vol. 4, No. 1, January 1999, pp. 1-11.

118. Gutberlet, P.- Rosenstiel, W., "Specification of Interface Components for Synchronous Data Paths", Proceedings of the 7th international symposium on system synthesis, 1994, pp. 134-139.

119. Madsen, J.- Hald, B., "An Approach to Interface Synthesis", Proceedings of the 8th International Symposium of System Synthesis, 1995, pp. 16-21.

120. Svantesson, B.- Kumar, S.- Hemani, A., "A methodology and algorithms for efficient interprocess communication synthesis from system description in SDL", Proceedings of the Eleventh International Conference on VLSI Design, 1997, pp. 78-84.

121. Kishinevsky, M.- Cortadella, J.- Kondratyev, A., "Asynchronous interface specification, analysis and synthesis", Proceedings of the 1998 Design and Automation Conference, 1998, pp. 2-7.

122. Hauck, S., "Asynchronous design methodologies: an overview", Proceedings of the IEEE, Vol. 83, No. 1, January 1995, pp. 69-93.

123. Yun, K.Y.- Dill, D.L., "Unifying synchronous/asynchronous state machine synthesis", Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993, pp. 255-260.

124. Bardsley, A.- Edwards, D.A., "The Balsa Asynchronous Circuit Synthesis System", Proceedings of the Forum on Design Languages (FDL2000), 2000, pp. 37-44.

125. Jacobson, H., et al., "High-Level Asynchronous System Design using the ACK Framework", Proceedings of the Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), 2000, pp. 93-103.

126. Edwards, D.A.- Tom, W.B., "Design, Automation and Test for Asynchronous Circuits and Systems - Async Tool Survey (3rd edition)", 2004.

127. Gil, D., et al., "Adaptation and Automation of the FPGA design Flow for asynchronous circuit implementation", Proceedings of the International Conference on Automation, Control and Instrumentation, 2005.

128. Benini, L.- De Micheli, G., "Networks on chips: a new SoC paradigm", Computer, Vol. 35, No. 1, January 2002, pp. 70-78.

129. Bainbridge, W.J., "Asynchronous System-on-Chip Interconnect", PhD Thesis, University of Manchester, 2000.

130. Muttersbach, J.- Villiger, T.- Fichtner, W., "Practical design of globally-asynchronous locally-synchronous systems", Proceedings of the Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2000), 2000, pp. 52-59.

131. Chapiro, D.M., "Globally asynchronous locally synchronous systems", PhD Thesis, Stanford University, 1984.

132. Yun, K.Y.- Dooply, A.E., "Pausible Clocking Based Heterogeneous Systems", IEEE Transactions on VLSI Systems, Vol. 17, No. 4, December 1999, pp. 482-487.

133. Bormann, D.S.- Cheung, P.Y.K., "Asynchronous wrapper for heterogeneous systems", Proceedings of the International Conference on Computer Design (ICCD '97), 1997, pp. 307-314.

134. Muttersbach, J., et al., "Globally-asynchronous locally-synchronous architectures to simplify the design of on-chip systems", Proceedings of the Twelfth Annual IEEE International ASIC/SOC Conference, 1999, pp. 317-321.

135. Royal, A.- Cheung, P.Y.K., "Globally Asynchronous Locally Synchronous FPGA Architectures", Proceedings of the Field Programmable Logic and Application, 2003, pp. 355-364.

136. "3D Synthesis System Version 3.13", Kenneth Y. Yun, University of California, San Diego, 1999.

137. Yun, K.Y.- Dill, D.L., "Automatic synthesis of extended burst-mode circuits: part I (specification and hazard-free implementations)", IEEE Transactions on CAD, Vol. 18, No. 2, February 1999, pp. 101-117.

138. Yun, K.Y.- Dill, D.L., "Automatic synthesis of extended burst-mode circuits: part II (automatic synthesis)", IEEE Transactions on CAD, Vol. 18, No. 2, February 1999, pp. 118-132.

139. Cummings, C.E., "Coding And Scripting Techniques For FSM Designs With Synthesis-Optimized, Glitch-Free Outputs", Proceedings of the Synopsys Users Group Conference (SNUG), 2000.

140. "Spartan-II 2.5V FPGA Family: Complete Data Sheet (DS001)", Xilinx Inc, 2003.

141. "Virtex™ 2.5V Field Programmable Gate Arrays (DS003)", Xilinx Inc, 2001.

142. "Application Note XAPP611 - Video Compression Using IDCT", Xilinx Inc, 2002.

143. Bhaskaran, V.- Konstantinides, K., "Image and Video Compression Standards: Algorithms and Architectures", 2nd ed, Kluwer Academic Publishers, 1997, ISBN: 0792399528.

144. "Data Encryption Standard (DES)", FIPS PUB 46-3, 1999.

145. Daemen, J.- Rijmen, V., "The Design of Rijndael: AES - The Advanced Encryption Standard", 1st ed, Springer, 2002, ISBN: 3540425802.

146. "Advanced Encryption Standard (AES)", FIPS 197, 2001.

147. "Digilent D2-SB System Board Reference Manual", Digilent Inc., 2003.

148. "Digilent DIO4 Peripheral Board Reference Manual", Digilent Inc., 2003.

149. Buchanan, W., "Computer Busses: Design And Application", CRC Press, 2000, ISBN: 0849308259.

150. "ISO/IEC 10918-1 | ITU-T Recommendation T.81. Digital compression and coding of continuous-tone still images - part 1: Requirements and guidelines." International Organization for Standards (ISO), 1993.

151. Wallace, G.K., "The JPEG Still Picture Compression Standard", Communications of the Association for Computing Machinery, Vol. 34, No. 4, April 1991, pp. 30-44.

152. Pennebaker, W.B.- Mitchell, J.L., "JPEG: Still Image Data Compression Standard", 1st ed, Kluwer Academic Publishers, 1992, 0442012721.

153. "JPEG File Interchange Format (JFIF)", C-Cube Microsystems, 1992.

154. "Application Note XAPP621 - Variable Length Coding", Xilinx Inc, 2003.

155. Ahmed, N.- Natarajan, T.- Rao, K.R., "Discrete Cosine Transform", IEEE Transaction on Computers, Vol. C-23, No. 1, January 1987, pp. 90-93.

156. Chapman, A.M., "VHDL Communications Library Guide ver1.0", User guide, University of Southampton, 2005.

157. "IrfanView (Version 3.95)", Irfan Skiljan, 2004. www.irfanview.com

158. "Spartan-IIE 1.8V FPGA Family: Complete Data Sheet (DS077)", Xilinx Inc, 2003.

159. Ozdag, R.O., et al., "High-speed non-linear asynchronous pipelines", Proceedings of the Design, Automation and Test in Europe (DATE), 2002, pp. 1000-1007.

160. Kim, C.- Shin, H., "A performance-driven logic emulation system: FPGA network design and performance-driven partitioning", IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Vol. 15, No. 5, May 1996, pp. 560-568.

161. "MOODS User Guide Version 1.2 (alpha)", LME Design Automation Ltd., August 2001.