UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING

School of Electronics and Computer Science

Behavioural Synthesis of Run-time Reconfigurable Systems

by

Donald Esrafili-Gerdeh

Thesis for the degree of Doctor of Philosophy

January 2016

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING

School of Electronics and Computer Science

Doctor of Philosophy

Behavioural Synthesis of

Run-time Reconfigurable Systems

by Donald Esrafili-Gerdeh

MOODS (Multiple Objective Optimisation in Data and control path Synthesis) is a Behavioural Synthesis System which can automatically generate a number of structural descriptions of a digital circuit from just a single behavioural one. Although each structural description is functionally equivalent to the next, it will have different properties, such as circuit area or delay. The final structural description selected will be the one which best meets the user's optimisation goals and constraints.

Run-time Reconfigurable systems operate through multiple configurations of the programmable hardware on which they are implemented, dynamically allocating resources 'on the fly' during their execution. The partially reconfigurable devices upon which they are based, enable areas of their configuration memory to be rewritten, without disturbing the operation of existing configurations − unless so desired. This characteristic may be exploited by partitioning a circuit into a number of distinct temporal contexts, which when ultimately realised as device-level configurations may be swapped in and out the device's configuration memory, as the run-time operation of the circuit dictates. At any point during the execution of the temporally partitioned circuit, the area required to implement it is equal to the size of the largest context and not the sum of each of its constituent parts, as would be the case in a non-reconfigurable implementation. The reduction in circuit area comes at the cost of a reconfiguration overhead, the time taken to partially reconfigure the device with each configuration and the frequency at which this form of context switching occurs.

This Thesis describes an extension to the original MOODS system, enabling it to quantify the trade-off that exists between the potential area reduction offered through run-time reconfiguration and the subsequent reconfiguration overhead incurred as a result. In addition to performing the temporal partitioning alongside existing circuit optimisation, MOODS is now able to automatically generate the infrastructure to support a practical implementation of the temporal contexts on a commercial Field Programmable Gate Array.

Contents

List of Figures

List of Tables

UNIVERSITY OF Southampton

Declaration Of Authorship

I, Donald Esrafili-Gerdeh declare that this thesis and the work presented in it are my own and has been generated by me as the result of my own original research.

Behavioural Synthesis of Run-time Reconfigurable Systems

I confirm that:

1. This work was done wholly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. None of this work has been published before submission.

Signed:

Date: January 2016

Acknowledgements

I would like to take this opportunity to thank several people all of whom have contributed to the completion of this thesis:

I begin by acknowledging the patience and common sense of my thesis supervisor Professor Mark Zwoliński, without whom I would not have been able to finish writing this thesis.

I would also like to thank former colleagues for their friendship in particular: Drs. Tack Boon Yee, Bleddyn Lawrence, Andrew Chapman, Matthew Sacker, Petros Oikonomakos, Kosala Amarasinghe and especially Doc and his wife Kathleen for our interesting conversations.

I would finally like to thank my family for their unconditional support.

Donald Esrafili-Gerdeh.

Chapter 1

1. Introduction

Before the use of Behavioural or High-level Synthesis Tools (HLS), the task of realising an algorithm in silicon was accomplished by a software description of the algorithm's behaviour for execution using programmable computer architectures or through a hardware description of the circuit structure required to achieve the same behaviour, but in a form suitable for fabrication.

Raising the level of abstraction at which the hardware is specified, analogous to that of software design, enables the user of HLS tools to automatically generate many alternative hardware descriptions from just a single behavioural specification. The number and type of resources described will vary in response to the resource or time constraints set by the user and without automation, it is unlikely that the user would have the time to investigate more than a handful of alternative hardware solutions.

The parallels with a software approach are strengthened following the introduction of programmable logic devices, such as the Field Programmable Gate Array (FPGA) [1,2]. This blurred the distinction between software and hardware design because FPGAs provide software programmable circuit resources. These resources include at least a wiring network which connects multiple sequential and combinational logic elements to one another and to a number of input and output pins. Therefore, programming an algorithm using the resources of an FPGA device is as much a software description of its circuit structure, as it is its function − albeit one at a low level of abstraction.

FPGAs are typically programmed or 'configured' by writing to a configuration memory. This occurs only once and is read by the device immediately following the application of power. A subset of FPGAs, characterised as being Dynamically or Run-time Reconfigurable [3], distinguish themselves from those that are programmable by enabling their resources to be configured during their initial power-up and crucially, are partially 'reconfigured' during their

execution. The time taken to partially reconfigure a commercial FPGA is several orders of magnitude greater than the time taken to execute the reconfigured logic resources. The benefits associated with re-using computational and wiring resources have to be weighed against the time penalty required to program any resource before it can be used.

When and how the reconfigurable resources are re-programmed has become an increasingly popular subject for research in the fields of VLSI CAD (Very Large Scale Integration Computer Aided Design) and Computer Architecture.

The circumstance in which reconfigurable hardware is employed in each of these fields is distinct. Application-specific hardware is intentionally inflexible, being highly optimised for one purpose. A complete design specification is available to the HLS tool at compile-time, when it can take a global view of how instructions are scheduled and allocated to resources. It also has considerably more time to do it.

This is in contrast to the assignment of memory and CPU resources in computer architectures, which from the perspective of an Operating System is an undertaking that must occur within a limited time-frame at run-time, requiring a solution to an open or partially specified problem.

Many problems in Computer Science and VLSI CAD are not solvable in polynomial-time and having more choice in the number and type of resource, such as those offered by reconfigurable devices, will further compound the search for their solution.

Research into any area of optimisation aims to reconcile the conflicting goals of finding optimal solutions to a problem, with as little search time as possible. In practice, CAD of circuits being no exception, the requirement of searching for an optimal solution is relaxed to accepting a good solution but one that requires less time to find.

An example of this approach to VLSI CAD is the behavioural synthesis suite MOODS (Multiple Objective Optimisation in Data and control path Synthesis) [4,5], developed over several decades at the University of Southampton for the automatic creation and optimisation of circuit hardware.

MOODS is a high-level synthesis tool, capable of automatically generating a structural description of a digital circuit from a purely algorithmic description of its behaviour. Heuristic

search methods are used to examine the scheduling and allocation possibilities in a way that aims to produce a good solution in a reasonable search time, rather than an optimal one in exponential time.

Not specifying how or what resources are to used, allows the hardware designer to explore these characteristics automatically, by varying the priorities and goals associated with circuit metrics, such as their resource size or delay of the longest path of execution. Each structural circuit description is functionally equivalent to another but has characteristics specific to the optimisation goals. Should those requirements or indeed the technology change, the behavioural specification need not.

1.1 Thesis Contribution

Given an optimisation trade-off between resource reduction and reconfiguration time, the availability of a behavioural synthesis tool in which to explore it and a commercial device to characterise the technology and test the results upon; the motivation for the work undertaken in this thesis is to investigate the role in which reconfigurable resources can play in the behavioural synthesis of digital hardware.

As a consequence, the MOODS behavioural synthesis tool has been extended to incorporate the reconfiguration delay of programmable resources, represented in circuit synthesis as their temporal binding to control and data-path components. When applied as a resource graph transformation, it models the spatial and temporal aspects associated with time-sharing a programmable resource, whilst preserving the behaviour of the algorithmic description.

Through the addition of new instructions and their corresponding data-path units, MOODS is able to quantify and generate the component descriptions necessary for an FPGA device to perform self-reconfiguration of logic and routing resources during its execution. The structural descriptions produced are suitable for device-specific optimisation by Register Transfer Level Synthesis and vendor-specific component placement and wire routing tools.

As with existing control and data-path units, their implementation is achieved through Xilinx Virtex [6] family primitives. This provides characterisation for metrics representing the circuit area or critical path delay of all the circuit components. The effect of the resource binding,

scheduling and allocation transformations is simultaneously quantified through a user directed cost function. It guides a Simulated Annealing [7] heuristic to accept or reject an optimisation based upon whether it transforms the circuit structure closer to or further away from meeting the user supplied target values of area, critical path delay and clock period.

An automated search of the design space explores a multitude of ways in which the hardware could be generated. The order and the components to which the transformations are applied are varied in ways that, for all but the simplest designs, would be infeasible for a user to investigate without automation – especially when optimising for equal priorities and therefore conflicting objectives. Aspects examined might be: which functional units should be shared to reduce the circuit area, as opposed to being scheduled in parallel to reduce the delay. Would sharing a programmable resource at different times also aid in meeting the area target and if so, could the reconfiguration delay be minimised by scheduling it to occur in parallel to the execution of other units?

When programmable resources are configured to implement data-path units, the number of times they are reconfigured will depend upon the type and activity of the instructions allocated to the data-path units. Should the units be allocated instructions from more than one path, the delays associated with the resource binding will also vary depending upon the path taken. Without knowing which path will be taken at run-time, a cost function must make a trade-off between the area reduction associated with a given resource binding against the reconfiguration delay derived from the longest path; a worst-case decision based solely upon one of many paths that are likely to be taken during circuit execution.

The need to rely upon worst-case reconfiguration delays has be greatly reduced by extending MOODS and HLS in general, to allow a run-time choice of where a data-path unit is bound. In this way, the partitioning of functional and storage units can change to reflect the instruction path actually taken at run-time.

However, unlike an entirely run-time approach [8], a multiple resource binding is able to optimise each alternative binding differently, depending upon the characteristics of the chosen resource. Binding to a resource of a different size will impose a different constraint on the scheduling and data-path allocations performed. A serial implementation that is smaller and

slower in one location can have an increased level of parallelism and consequently execute faster when bound to a larger resource at an alternative location.

The benefit of this hybrid-approach is that it combines the strengths of the compile and run-time approaches to reconfiguration without adding to their weaknesses. A user can now use MOODS to explore the extended ‘design space’ which results from applying spatial and temporal resource binding alongside instruction scheduling and allocation methods. This takes place at compile-time where there is greater time and computational resources to search the new territory formed by temporal resource binding.

Selection of the actual resource is taken at run-time from a small number of alternative resources found during compile-time and therefore inclusive of the history of trade-offs which lead to their resource binding. By doing so, the partitioning of the data-path units is not limited to a single partition formed by taking the worst case delay of many paths. Instead, the units may be re-arranged over multiple resource bindings, each unique to a different instruction path and each representative of the best, the worst and the spectrum of reconfiguration path delays in between.

This Thesis is divided into eight chapters. Chapters 2 and 3 provide a background to the research topic, introducing the concepts behind run-time reconfiguration and behavioural synthesis, in particular MOODS, the synthesis system on which this research is conducted. It concludes with a survey of past and present research activities regarding run-time reconfiguration, in relation to device technology and architecture, CAD tools and implementations of actual systems.

Chapter 4 introduces the theory behind temporal binding and its utilisation during optimisation through a cost driven partition of the control and data-path functional units and subroutines. The MOODS cost function is updated with several metrics which provide an estimation of the reconfiguration overhead and architecture required to facilitate dynamic reconfiguration at the device level.

In Chapter 5, the low-level infrastructure necessary to support a physical implementation of a run-time reconfigurable system is presented.

The results generated using a simulated annealing based partitioning algorithm is examined in Chapter 6.

Chapter 7 provides an examination of how the work detailed in the previous chapters can be applied to the synthesis of a reconfigurable coding system.

Finally, Chapter 8 reviews the contribution of the work described in the thesis and suggests how the work may be further extended. An appendix is also included which supports several of the chapters with additional material omitted for the sake of brevity: it includes instruction level optimisation details as well as the characteristics of the circuits used during the synthesis and temporal resource binding phases.

Chapter 2

Background

In the introductory chapter, a reconfigurable system was described as being able to change the use of electronic programmable resources during its execution. This description cannot be exclusively attributed to reconfigurable hardware, since programmable computer architectures are also able to change their function through run-time re-programming of their computational and memory resources. Some authors might also challenge our presumption of electronic resources, citing examples from outside the electronic hardware domain such as reconfigurable photonic and fluidic systems [9].

When searching for a definition for electronic reconfigurable hardware, distinguishing it from the diversity of existing computer architectures can be challenging**.** Many authors [10] draw attention to the large number of parallel resources associated with reconfigurable hardware, yet a high degree of 'spatial computation' is not unique to reconfigurable hardware; recall the massively parallel SIMD/MIMD [11] computer architectures of earlier decades, where systems offering tens of thousands of bit-sized CPUs were joined by programmable interconnects.

Perhaps, as some authors suggest [12], a defining characteristic of reconfigurable hardware might lie in the ability to re-purpose the use of its resources? Yet again, there exist comparable computer hardware – writable micro-coded program stores [13,14], that provide the ability to form new instruction implementations out of an existing instruction set architecture and in the same sense enable a re-structuring of the programmable resources.

A more tangible distinction between programmable and reconfigurable resources is provided by DeHon [3] who defines resource binding as a means of distinguishing between software, reconfigurable and application-specific implementations of program behaviour: computer hardware is flexible because binding occurs at run-time during instruction execution;

application-specific hardware is not because transistor equivalents of instruction behaviour are fixed during their fabrication. Although the binding of a reconfigurable system occurs during its configuration, its definition is difficult to pin-down since the characteristics of the resources are bound early during fabrication but may be allocated as late as a clock cycle prior to their execution.

Rather than attempt to single-out reconfigurable hardware as a new paradigm for computation [15], a different perspective on the role of reconfigurable hardware would be to view it in the context of hardware design. The perspective on reconfigurable hardware taken in this chapter and the thesis in general, is one that views it as a way of complementing existing hardware design approaches such as behavioural synthesis.

2.1 Programmable and Application-Specific Hardware

Algorithms for computation and data processing are traditionally realised as software descriptions of behaviour for a general-purpose processor or as structural descriptions of the circuit hardware required to implement them. Together they represent two contrasting approaches to designing electronic hardware, each with a specific circumstance for its use:

General-purpose hardware is based upon the control-flow or Von Neumann (VN) model of computer architecture [16] − a set of logic and arithmetic resources are programmed by instructions to transform the data supplied to them, in a way associated with the behaviour of each instruction and the type of data it acts upon.

The flexibility of the VN architecture is achieved through the generation of memory addresses, where an instruction or datum is sequentially written to or read from each address location and where necessary, decoded and subsequently executed to achieve temporal computation. When a re-writable memory is used, changing the contents of the memory will change the behaviour of the hardware – making it general-purpose.

There are many advantages to using general-purpose hardware. In addition to the flexibility gained from programming the hardware to implement different behaviour and the run-time allocation of resources, there is the cost reduction associated with volume manufacture and the outsourcing of hardware design and testing. Often, the instruction level parallelism

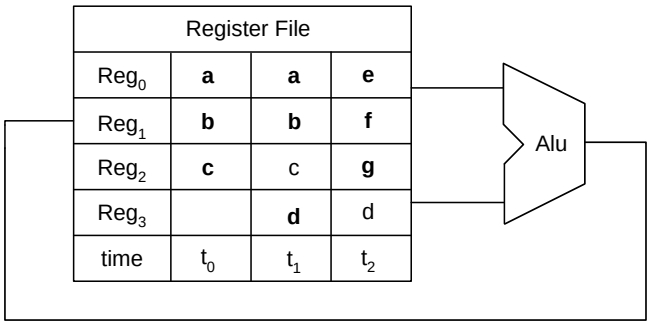
naturally inherent to a given algorithm must be serialised to fit the availability of the computational and storage resources of generic hardware. In these circumstances, not being able to specify how the instructions are executed and in cases where an operating system is employed, when execution actually occurs, can necessitate the design of fixed and single purpose hardware.

Application-specific hardware is often based upon the Data-flow [17] model of computation, where each instruction operation is allocated to a specific functional unit and its input and output operands are allocated to memory units connected by wires. The execution of a program is modelled as data-flowing through each functional unit as soon as it becomes available. The model is usually extended to the Control and Data-flow [17] model, to enable the scheduling of each instruction execution to a time step represented by a control graph and realised in hardware as a specific control state in a Finite State Machine.

In addition to implementing the behaviour of the program, the allocation of functional and memory units and when they are scheduled to execute is optimised to meet specific constraints, such as the total number of hardware resources available or the time taken to execute the longest sequence of program instructions. The hardware that results is unique to the characteristics of the algorithm being synthesised and the optimisation priorities and constraints set.

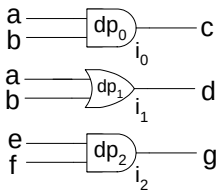
2.2 Temporal and Spatial Computation

Figure 2.1a illustrates a simplified representation of the computational and storage resources associated with the data-path of a general-purpose processor. Three logical expressions, i_o: c=a and b; i_1:d= a or b; i_2: g=e and f; are used to program an Arithmetic Logic Unit (ALU) to perform each logical function and determine which of the Registers are used to store the variables read by and written to the ALU. The adjacent table shows the scheduling of each instruction to individual time steps t_0-t_2. When executed sequentially, the use of the resources can be characterised as being temporal because the ALU and registers are re-used to perform the instructions at separate times.



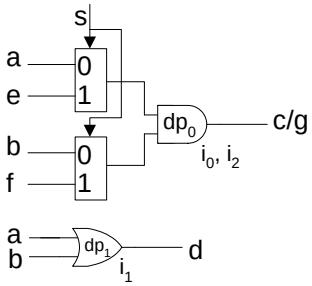
Time	Instruction schedule
t_0	i_0:and Reg_0,Reg_1,Reg_2
t_1	i_1:or Reg_0,Reg_1,Reg_3
t_2	i_2:and Reg_0,Reg_1,Reg_2

Figure 2.1a: Temporal use of resources for computation exemplified in general-purpose hardware.



Time	Instruction schedule
t_0	i_0:and a, b, c i_1:or a, b, d i_2:and e,f, g

Figure 2.1b: Spatial use of resources for computation exemplified in application-specific hardware.



Time	Instruction schedule
t_0	i_0:and a, b, c i_1:or a, b, d
t_1	i_2:and e,f, g

Figure 2.1c: Temporal and Spatial use of resources for computation exemplified in application-specific hardware.

Figure 2.1: Traditional forms of spatial and temporal resource usage.

Figure 2.1b depicts the equivalent data-path should each instruction be allocated a separate functional unit and scheduled to execute in a single time step t_0. The hardware can be described as application-specific, since the allocation and scheduling of the functional units is dedicated solely to implementing the instructions described. In this implementation, the resources exhibit spatial computation as each function unit will execute at the same time, therefore requiring three of them – dp_0, dp_1 and dp_2, respectively.

There will inevitably be constraints imposed on the number of functional units available and the number of time steps in which to execute them. The result is a compromise between the degree of spatial and temporal use of resources necessary to meet the constraints. This situation is depicted in Figure 2.1c, where instructions i_0 and i_1 are allocated individual functional units dp_0 and dp_1, enabling their spatial execution during the same time step t_0, as was the case in Figure 2.1b. However, with a resource constraint of one 'AND' functional logic unit, dp_0 must be shared between instructions i_0 and i_2, thereby forcing the execution of instruction i_2 to occur during the next time step t_1. In this way, the execution of the instructions is reliant on the temporal use of the functional unit dp_0.

Somewhere between these two extremes of flexibility of general purpose and the optimisation of dedicated hardware lies reconfigurable hardware. What was assumed and not depicted in the allocation of the data-path functional units was that each would be uniquely bound to its own resource.

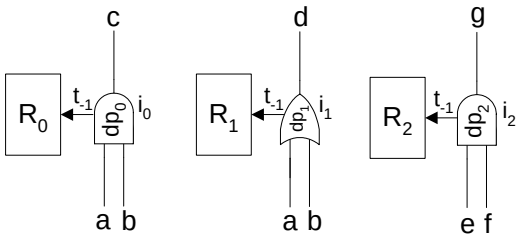
Figure 2.2a explicitly shows the static resource binding inherent to the resource allocation and scheduling previously shown in Figure 2.1b. The figure illustrates the binding during the time step t_{-1}, of a resource R_n to each of the data-path units.

The negative time denotes the fact that it occurs before execution of the schedule and is therefore fixed at fabrication or during power-up configuration – should the resources be programmable. This form of static configuration is representative of the typical usage of commercial programmable logic devices, which are optimised to read a configuration once, through an external interface from a non-volatile configuration memory.

Likewise, circuit synthesis has traditionally assumed a fixed binding of functional or memory units to an implementation in a given technology. Any variation in data-path unit binding occurs only during circuit compilation, with the purpose of exploring how a different binding may aid in meeting the design constraints.

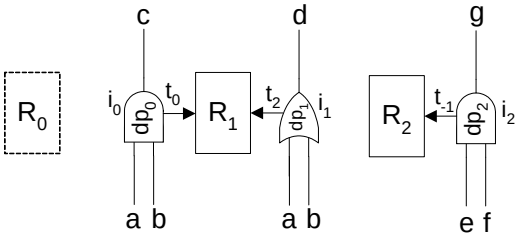
Figure 2.2b illustrates the reduction in resource usage, when the assumption of a static binding for data-path units dp_0 and dp_1, shown in the earlier figure, is dropped in favour of a dynamic or temporal binding to a reconfigurable resource R_1. When reconfiguration is scheduled to occur during the operation of the hardware resources, those resources are

described as *Run-time* or *Dynamically Reconfigurable* [3]. As shown in the adjacent schedule, this form of resource sharing requires two extra time steps to re-programme the resource. It is reminiscent of the choice described earlier, between a smaller number of resources or a shorter number of time steps. Once programmed, dp_0 can be executed alongside the static bound resource dp_2, scheduling both their associated instructions to the same time step t_1. In this way, the use of reconfigurable hardware offers the spatial computation associated with an application-specific choice of functional units, as well as the temporal re-use of resources embodied by programmable processor hardware.



Time	Instruction schedule
t_0	i_0:and a, b, c i_1:or a, b, d i_2:and e,f, g

Figure 2.2a: Static binding of data-path units to resources.



Time	Instruction schedule
t_0	i_0:reconfigure R_1=0001
t_1	i_0:and a, b, c i_2:and e,f, g
t_2	i_3:reconfigure R_1=0111
t_3	i_1:or a, b, d

Figure 2.2b: Temporal binding of data-path units to reconfigurable resources.

Figure 2.2: Spatial and temporal use of reconfigurable resources.

Hardware permitting, it would also be possible to schedule the execution of instruction i_2 to either time steps t_0 or t_2. In doing so, the reconfiguration of resource R_1 would occur in parallel to the execution of functional unit dp_2 bound to resource R_0. In such a case, the resources of the hardware would be described as being *Partially* and *Dynamically Reconfigurable* [3].

Run-time reconfiguration can be categorised as being Algorithmic, Architectural or Functional.

Algorithms which implement the same functionality but with different performance, accuracy, power or resource requirements are examples of *Algorithmic* reconfiguration. An application could be the implementation of an adaptive Viterbi decoder, where decoder performance may be dynamically altered in response to changing channel conditions. Reconfiguration may occur on the basis of channel noise to maintain a consistent bit-error rate. Any increase in channel noise would result in a slower but more accurate running decoder being swapped into the FPGA hardware and vice-versa in the case of reduced channel noise.

Architectural reconfiguration lends itself to fault tolerant applications, where reconfiguration is used to modify the hardware topology in the presence of a fault(s).

A particular application might be the tolerance of Single Event Upsets (SEUs) in equipment used in space and orbit based systems. SEUs result from radiation in the form of high energy charged particles and may alter the logic state of a static memory element (latch, flip-flop, RAM cell). User programmed functionality of a field programmable device depends upon the data kept in the configuration memory, therefore the presence of a fault would have an adverse effect on design functionality. Partial readback of the affected portion of configuration memory may be performed and a Cyclic Redundancy Check value generated and compared to the expected result, to detect an error. The contents of the configuration memory controlling that portion of the device are then reloaded or even relocated in the event of a persistent error.

The goal of *Functional* reconfiguration is to increase the functional density of a system, through the execution of different functions on the same resource. An algorithm or design is subject to temporal partitioning through a division into time-exclusive segments which are swapped on and off a device at run-time.

An application could be a synthesised FPGA based processor implementing user defined application specific instructions. Such specialised instructions would be loaded into the device as the run-time conditions of the design dictated. The implementation could be efficient in terms of performance, replacing long streams of general-purpose instructions and eliminating the instruction fetch-decode part of the cycle. A significant area reduction would also be achieved in comparison with a static FPGA implementation of the entire processor design.

2.3 Reconfigurable Resources

To date, all reconfigurable resources have been referred to without attributing the physical characteristics associated with a particular architecture and technology. It was a deliberate omission, in order to separate their concept from any practical constraints that would otherwise have been imposed by examples of their implementation. In this section of the chapter, we will provide a brief history of the technologies and techniques which have helped to popularise run-time reconfiguration as a growing area of research interest. The reader should also have in the back of their mind the notion that the importance of such techniques is often contemporary and a change of technology may invalidate them or in some cases resurrect their employment.

Estrin [18] and colleagues are frequently cited as the being the earliest practitioners of a ‘Reconfigurable’ approach to hardware design: recognising the performance limitations of general-purpose or ‘fixed’ hardware lead to the creation of ‘variable’ hardware libraries. The result took the form of the “Fixed-Plus-Variable” computer architecture and its implementation relied upon physically ‘pluggable’ module functions which were fabricated using discrete components.

Research included how to decide between a software or hardware realisation of a function [19], as well as the practical implications involved in implementing run-time reconfiguration: reconfiguration was achieved at different levels of abstraction by hand swapping a module (function) or motherboard (algorithm). As a result, a wide band of performance gains (2.5-1000) were achieved in comparison with a computer implementation (IBM 7090).

Where Estrin identified reconfiguration as adapting the hardware to the algorithm, Miller and Cocker [20] described adaptation in terms of the data-flow aspects of an algorithm’s computation. This lead to two types of ‘Configurable Computers’: a ‘Search-Mode’ configuration, where the absence of the program counter requires a run-time data-flow approach: data-dependant operations executing as soon as their input operands become available.

A second ‘Interconnect-Mode’ configuration has considerable resemblance to synthesising reconfigurable logic due to its description of encoding the dependencies between ‘special-purpose blocks’ of hardware in advance of their run-time execution, at which time the instructions would control their configuration; suggesting a compile-time approach, as the instructions were encoded prior to their run-time execution.

In addition to describing the importance of identifying data-flow between operations (as a means of accelerating computation) Miller and Cocke may also have used the first reference of the hardware being ‘dynamically reconfigured’, as well as implying partial reconfiguration by overlapping ‘block execution’ with ‘block set up’.

It would be over a decade later, when a VLSI implementation of a ‘configurable array of fine-grain elements’ was implemented in the form the ‘CAL Architecture’ [15], where it was described as ‘A New Paradigm for Computation’ and shown to provide significant performance acceleration when compared to more conventional forms of programmable computer hardware. The rights to its implementation was later purchased by Xilinx Inc., where it formed the basis of the XC6200 [21] series of FPGA, a partially reconfigurable device with an open-architecture that is widely credited with popularising academic research into run-time reconfiguration.

2.3.1 FPGAs

All reconfigurable systems are based upon some form of programmable hardware, many utilise field programmable logic in the form of commercial SRAM FPGAs (Field Programmable Gate Arrays). In 1985, Xilinx Inc. introduced the first commercial FPGA, a new class of programmable logic device conceived as a replacement for Mask Programmable Gate Arrays (MPGAs). MPGAs provided an array of gates with fixed functionality, such as Nand gates. The routing was done by the designer via the last metal layer in the silicon process. The principal drawback with MPGAs was the cost of exploring different design alternatives, since each design required a new chip and routing layer to be manufactured.

FPGAs alleviated this problem by making the gates and routing network programmable. Programming the functionality of an FPGA is done ‘in the field’ by downloading a configuration bitstream into the FPGA.

FPGA flexibility is derived from its use of programmable logic cells, routing and I/O cells as depicted in Figure 2.3. Initially there was a wide architectural variety in the implementation of the logic cells, however recently there has been an adoption of the Look Up Table (LUT) based cell. LUTs are programmed to provide any logic function of their inputs. The internal architecture of a cell consists of a number of LUTs, coupled with carry logic, state storage and multiplexors, to control its internal configuration. A rich routing fabric is provided which may include millions of possible routing pathways through the device, achieved through Local Connection Points and Global Routing Switch Boxes.

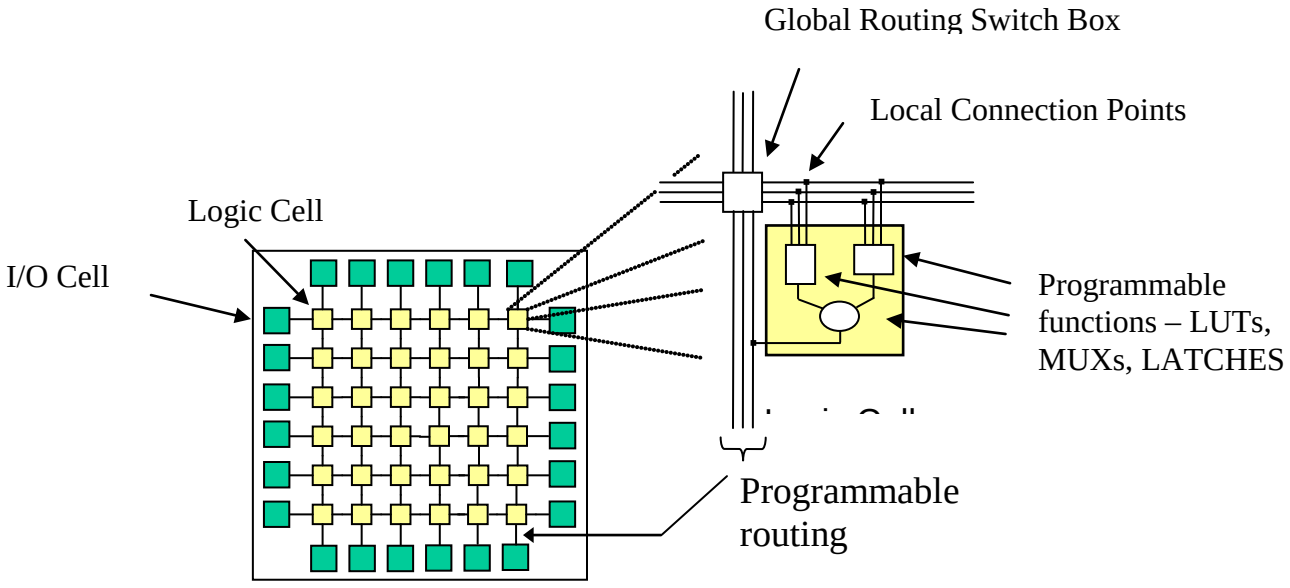


Figure 2.3: Generic FPGA structure.

The programming of the resources is done through SRAM configuration bits. Being SRAM based, the devices may be programmed an unlimited number of times, with the time required for configuration proportional to the size of the device and dependent upon the characteristics of the programming interface. Such volatility of the SRAM cells requires that the configuration bits are loaded from an external configuration store, typically a serial or parallel ROM, upon each power-up of the device.

Programmability comes at the cost of efficiency [22]: the implementation of logic in an FPGA is less dense (18 times) and slower (3 times) than an ASIC in a comparable process technology. This is due to the large amount of wiring resources required, leaving less area for active circuitry.

2.3.2 Technology and Architecture

The majority of research in dynamically reconfigurable systems is in the area of Reconfigurable Computing. Such systems may be characterised by the strength of the coupling between a processor and programmable logic, granularity of architecture and depth of programmability.

a) *Coupling:* dynamically reconfigurable systems are implemented with varying degrees of coupling between a processor and programmable logic. Computational efficiency is increased by making decisions at run-time as to whether computation is executed on the reconfigurable logic or processor. This ranges from a tight coupling where the reconfigurable units execute as functional units in the data path (System on a chip architecture) to a loose coupling, with the programmable logic implemented externally on a platform and independently of the main processor.

b) *Granularity*: the granularity refers to the data size of the operations for the reconfigurable component. In the preceding section, the FPGA was introduced as an example of hardware programmability upon which reconfigurable systems may be implemented. However in contrast to the FPGA (fine-grain reconfigurability), the domain of reconfigurable computing stresses the use of coarse-grain reprogrammable arrays, which are achieved through custom design. FPGAs are classified as having a fine granularity of reconfiguration, as the look-up tables, flip-flops and logic gates operate at the bit level. Coarse-grain architectures provide programmable cells at the operation level with word level data paths.

c) *Depth of programmability:* the depth of programmability refers to the number of configurations (or contexts) stored within the programmable device. The following configuration memory models have a different unit of reconfiguration, the smallest

segment of configuration memory that can be reconfigured. It is the dominant factor in the reconfiguration overhead associated with swapping a context on or off a device.

(i) Single Context

The configuration of the traditional FPGA architecture is achieved through a device configuration memory capable of storing a single configuration. System wide configurations are loaded serially from an external memory device, often requiring hundreds of thousands of clock cycles. This incurs a high reconfiguration overhead, as a small change to the configuration requires a complete reprogramming of the device. Examples of typical commercial FPGA devices are Xilinx 4000 [23] series, Altera Flex10K [24].

(ii) *Partially Reconfigurable*

The introduction of devices capable of dynamic partial reconfiguration made the implementation of single device run-time reconfiguration feasible. Configurations that do not use the entire resources or require changes to selective areas of the device are executed through partial reconfiguration. If required, existing configurations continue to operate during and after reconfiguration. This reduces the amount of configuration data sent to the device, which in turn reduces the cost of the reconfiguration overhead. Examples of commercial FPGA devices are the Xilinx Virtex family [6] and Atmel AT40K [25].

iii) *Multi-context*

Multi-context devices have many memory bits for each programmable bit location. This is implemented through on-chip multiple memory planes and can result in a very low overhead (625 ps) [26] as reconfiguration is carried out by switching internally between the planes, without requiring external loading of the bitstreams. Configuration planes may also be partially reconfigurable.

Some thirty Fine and Coarse-grain reconfigurable systems and devices are reported in the literature. What follows is a brief survey of a representative number of those systems and devices. Comprehensive surveys may be found in [9,27,28].

The concept of Virtual Hardware within reconfigurable systems was introduced using the WASMII system [29,30]. Virtual hardware enables configurations or contexts to be swapped

in and out of a programmable device during its run-time execution. Such a re-use of device resources reduces the area required for a digital circuit implementation. This is analogous to the use of Virtual Memory in a software operating system which permits a program to be larger than the physical memory it resides in by mapping portions of the virtual memory to the physical memory when required.

WASMII was an early system to conceptualise a multi-context FPGA. The device consists of an FPGA coupled with a set of multiplexed SRAM. A data-flow graph is partitioned into a number of pages, each being equivalent in size to the available resources of the FPGA. Each page is mapped in to the SRAM memory and swapped on or off the FPGA by a Page Controller. The controller determines when to swap pages and where in the additional register space to store and retrieve the data used by the pages. In addition to the multiple contexts, virtual hardware is mapped from external memory. Execution may be overlapped with configuration by pre-loading a page from external memory to the device configuration SRAM, whilst the current page is active. Initially a lack of suitable technology forced an emulation of WASMII using a number of FPGAs coupled with an external memory and a microprocessor.

WASMII was later realised on an experimental multi-context device referred to as the Dynamically Reconfigurable Logic Engine (DRLE) [31] developed by NEC Corp. This architecture had been commercialised in the form of the Dynamically Reconfigurable Processor 1 [32], targeting image and signal processing, in addition to network packet processing applications such as routers and switches.

The device consists of an array of 512 equivalent ALU based processing elements (PEs) with local access to a total internal storage memory of 2.2MB. Most significant is the emphasis on efficient dynamic reconfiguration between 16 configuration context layers, providing an additional 7680 virtual PEs. The architecture is partially reconfigurable by dynamically changing the configuration of the PEs and interconnections between them without effecting the existing configurations. Data-path reconfiguration is controlled through an integrated Finite State Machine Sequencer. Products and IP cores using the architecture were anticipated but nothing further was reported after the turn of the millennium.

The Swappable Logic Unit (SLU) [33] was proposed as a paradigm for virtual hardware, being analogous to a page or segment in virtual memory systems. The SLU is essentially a FPGA function with a fixed interface capable of being paged onto the hardware and connected by an operating system at run-time. The SLUs were made available as library functions utilised by a high level language such as C. The concept was demonstrated on a hardware platform [34]. It was argued that due to the relative immaturity of the SLU concept [35], SLUs were unlikely to be incorporated into existing design tools. A solution was proposed that detected SLUs within existing bitstreams through a combination of existing knowledge and automatic detection.

Many reconfigurable implementations are carried out on commercial SRAM FPGAs. These devices provide the basic requirements – abundant logic and interconnect, on chip memory and the ability to partially reconfigure 'on the fly'.

Partially and dynamically reconfigurable devices, such as the Xilinx Virtex [6] family and Atmel AT40K [25] enable virtual hardware to be implemented using standard FPGAs. Unfortunately the devices are not optimised for efficient reconfiguration. The bottleneck caused by loading the configurations from an external memory forces infrequent context switching to be the only cost effective option. Xilinx Inc. filed a patent on a time-multiplexed FPGA in 1995. The patent covers a multi-context programmable FPGA that uses cells similar to the Xilinx XC4000E FPGA. That is where the similarity ends, since the multi-context architecture enables each configuration memory cell to have a further eight inactive configurations. This gives the device a set of eight virtual background layers, where a configuration switch from background to active layer is executed in 30 ns.

Xilinx do not have any future plans for its production, citing power consumption as a potential drawback [36]. In addition, larger capacity FPGAs generate greater revenues, therefore there could be no financial motivation for reducing the size of silicon.

A number of academic time-multiplexed FPGA architectures have been proposed [37,38], none of which have been realised commercially.

The furthest anyone has been to introducing fine-grain and rapidly reconfigurable devices to the market-place was Tabula Inc. with the ABAX [26] series of '3D Programmable Logic'

time-multiplexed devices. Tabula Inc. was until recently (first quarter 2015) selling devices and development boards for communication and networking applications. Although very little information has been made publicly available about the specification of their devices and is unlikely to become available (Tabula Inc. has ceased trading), it was a rare example of a rapidly reconfigurable device offering the type of resources an FPGA user would recognise e.g. Look-Up Tables, Registers etc.

Being a multi-context architecture, the ABAX device permitted between 8 and 12 device configurations to be stored on-chip, switching between them incurred a reconfiguration delay of 625ps [26]. A user's design would be partitioned between as many as 12 temporal partitions, where all data-dependencies cut during partitioning were captured by transparent latches. Unlike other multi-context devices [36,37,38], the architecture incorporated stage storage as part of the routing.

Tabula's selling point was that their temporal devices could emulate a spatial device 12 times its physical size, whilst keeping the spatial and temporal aspects of placement and routing transparent to the user. Details about the device's power consumption was described as being 'application-specific' but inevitably the decrease in static power consumption (associated with fewer logic and wiring resources than its spatial equivalent) would have to be offset against the increase in dynamic power consumption due to rapid reconfiguration.

Commercial research has also examined alternative Coarse-Grain (word-level) data-flow architectures, such as NEC's Dynamically Reconfigurable Processor 1 [32] and PACT's XPP [39] architecture, both based upon dynamically reconfigurable arrays of processors.

A radically different architecture was the Adaptive Computing Machine (ACM) from QuickSilver Technology Inc. [40]. Being a network on a chip device, it utilised message passing as opposed to point-to-point routing found in FPGA architectures. The motivation for the architecture was based upon the assertion that algorithms are heterogeneous in nature and the homogeneous architectures associated with FPGA-based reconfigurable systems do not satisfy the demands of adaptive computing. The ACM architecture is therefore heterogeneous, consisting of five types of nodes: bit manipulative, arithmetic, finite state machine, scalar and configurable input/output.

The granularity of reconfiguration is extremely coarse, each node executing tasks at the complete algorithmic level. For example, an individual arithmetic node may be used to implement different variable-width arithmetic functions such as an FIR filter or Fast Fourier Transform (FFT). A bit manipulation node may be used as a Linear Feedback Shift Register or as other variable bit width manipulation functions.

Each node is coupled with a local memory cache and configuration memory. The architecture provides efficient reconfiguration by utilising a 256 bit configuration bus, enabling every node to be reconfigured on a clock cycle basis. This enabled the ACM to adapt by tens or even hundreds of thousands of times a second.

The final approach taken is to assimilate reconfigurability into an SoC (System on a chip) architecture: RISC processors and DSP cores are used to execute as much functionality in the software as possible, delegating blocks of programmable logic fabric to those elements of a design which could benefit from its use for hardware acceleration.

The Chameleon CS2112 [41] was the industry's first reconfigurable processor: being a SoC, the device consists of a multi-context run-time reconfigurable logic fabric, coupled with a 32 bit RISC processor core. The reconfigurable fabric consists of an array of 32 bit data-path units and 16 bit multipliers, partitioned into a set of dynamically reconfigurable slices. Each slice also has a second configuration memory plane. This enabled another configuration to be loaded in the background during the execution of the active circuit. Switching from the background to the active plane could be accomplished in a single clock cycle. Switching a partition resulted in a small overhead for loading each partial configuration, necessitating a delay of 4 μs per slice.

2.4 CAD Tools for Dynamically Reconfigurable Logic

The motivation for developing design frameworks which incorporate design capture, synthesis and simulation for dynamically reconfigurable systems is a lack of support from the technology vendors themselves, as well as an absence of commercial software to enable the designer to quantify the trade-offs involved when utilising reconfigurable technology.

A considerable number of academic software tools have been developed to support run-time reconfigurable systems [42]. The tools vary depending upon the resources which characterise the target system: reconfigurable computer systems are composed of programmable logic and software processors. Most are a mixture of industry standard and custom tools, typically compiling and partitioning standard C into a software executable and a set of hardware modules for a reconfigurable data-path [43]. Although these tools incorporate many aspects common to high-level synthesis, their goal is hardware acceleration using a rapid design- flow similar to conventional computer programming. In contrast, tools for synthesising reconfigurable logic consider more than one objective and usually require a greater compilation time to achieve their goals: being application-specific, the hardware generated is inherently accelerated but generated under tighter resource constraints.

JHDL [44] is a structural/RTL hardware and software co-design environment based on Java, it enables a designer to use their expertise to optimise layout and circuit composition. The result is the generation of faster circuits and smaller device bitstreams in one environment. JHDL manages circuits in a manner similar to the way object-oriented languages maximise memory, where circuits are treated as objects.

Other alternative forms of languages for hardware design include: Ruby [45], Pebble [46] and Lava [47]. These languages enable a designer to specify components in a manner that is more convenient than the use of attributes associated with standard HDL designs.

The development of a framework for creating parameterised core libraries is presented in [45]. Ruby is used to enable an initial exploration of the design space. The Ruby description of a core is (manually) translated into parameterised VHDL. This contains attributes for placement to be processed during automatic synthesis and translation to configuration bitstream.

The framework was developed further in [46] using the language Pebble, for the initial description and automatic translation into VHDL. Pebble was developed to enable research into supporting tools for reconfigurable designs. It can be regarded as a simplified variant of structural VHDL and its word-level and bit-level descriptions may be customised by different parameter values such as design size and number of pipeline stages. Optional constraint

descriptions, such as placement attributes are supported at various levels of abstraction and run-time reconfiguration is supported with a “Reconfigure-if” statement.

In [47], the Lava hardware description language allows circuits to be described at a high level whilst permitting layout to be captured. The output netlist contains a fully placed design which is fed into JBits [48], an API to manipulate and generate Xilinx device bitstreams. The aim of the approach was to significantly reduce the time taken to generate the bitstreams from HDL to device configurations, when compared with the conventional design flow. The preliminary version of the system could generate bitstreams 12 times faster than the conventional flow. The authors described the potential for further performance increases of 50 times the conventional flow. The obvious application is in the field of reconfigurable computing where applications require fast compilation.

Other researchers have considered a lower level of abstraction, enabling the designer to have absolute control over component placement and routing. JBits was developed by Xilinx Inc. as a non-commercial Java class library to allow users access to the propriety bitstream, without compromising the security of the intellectual property rights associated with their devices. JBits permits access to the programmable resources of the FPGA through a set of class functions and constants. The functions enable configuration data streams to be read from and written to, where the status of an individual programming resource such a CLB maybe queried or set to a defined value. The constants define the programmable resources and the values that they may be set to within the device.

In [49] the idea of extracting functionality from existing bitstreams was taken a step further to enable the extraction of run-time cores from programmable device configurations. The objective was to insert a core after the bitstream has been generated, during the last step in the design process. This was in contrast to commercial toolsets, such as the Xilinx Core Generator [50], which generate netlists to be processed by the place and route tools early in the design flow. The cores are referred to as Run-Time Parameterisable Cores [51], as their generation and addition to existing bitstreams may be done during the run-time execution of a design.

A JBits core consists of a number of class calls to program the relevant CLB, BlockRAM and routing resources. It is feasible to design circuits through a series of connected cores, however this would not take advantage of the availability of pre-tested vendor supplied cores.

The JBits approach proposed the creation of a series of cores and subsequent bitstreams by querying the resources used in the configuration bitstreams constructed from the IP core. This technique was used to develop a library of run-time cores which an application running on a host PC could use to modify an application 'on the fly'. Unfortunately the run-time cores required a great deal of understanding of low-level device architecture – too low-level an abstraction for specifying today's multi-million gates designs. As a consequence of bypassing circuit optimisation at higher levels of abstraction, the responsibility for achieving an efficient circuit structure lay with the designer – a formidable task which would ordinarily employ design automation at several stages of abstraction: circuit synthesis and partitioning are two such examples, both of which are essential when implementing run-time reconfigurable systems.

2.5 Synthesis and Partitioning

Despite the ever increasing body of work covering run-time reconfiguration, there are at its core a small number of problems which all approaches must consider. These are briefly stated in order to provide some context to further discussion of the literature:

- Reconfiguration delay: does the programming method of the device affect the level at which partitioning occurs?
- Is the partitioning approach able to quantify multiple objectives, communication costs as well as the essential reconfiguration versus resource-reuse trade-off?
- Is the architecture generic or a by-product of the partitioning?

The task during circuit partitioning is the division of a circuit into two (Bi-Partitioning) or more (K-way Partitioning) parts, each of which must satisfy a constraint on its physical property [52]: a device pin-count associated with the weights of the edges cut during partitioning and a resource constraint on the size of each partitions; for bi-partitioning a

measure of balance is a common requirement, perhaps less often in multi-way partitioning where a repeated bi-partitioning might require resources of different sizes.

A Temporal Partitioning [42] extends these constraints to incorporate placement: the resource placement of the partitions must overlap. By implication, the nets cut during spatial partitioning require little consideration as to their direction; in contrast, the direction of nets cut during temporal partitioning are more likely to be uni-directional between partitions which are temporally adjacent. Unlike a spatial partition, a component in one temporal partition may also be present in another. In doing so, it may preserve a signal state or logic function, typically also reducing the information necessary to reconfigure the resource.

How temporal partitioning is achieved is greatly influenced by the architecture of the reconfigurable resource: a multi-context device may restrict the nets to flow in one direction only, where each context generates a signal which is immediately processed by the next [36].

Other architectures may separate sequential and combinational resources, thereby enabling bi-directional nets between spatial and temporal resources in the architecture. The Time-Multiplexed Communication Logic architecture [53] exemplifies this approach. Unlike the architecture of the Time-Multiplexed FPGA [54] or Time-Switched FPGA [36] there is no restriction to store signals only between adjacent partitions, therefore nets maybe bi-directional.

A less exotic architecture which exemplifies both aspects is the FPGA [2]. A device offering only full reconfiguration has no architectural provision for storing state; a partially reconfigurable device may reconfigure one of many resources, relying upon the preservation of state in a resource not being reconfigured.

An early reference in the literature regarding the partitioning of a behavioural hardware description (Verilog) in to reconfigurable resources is the work by Schmit et al. [55] who describes the difficulties they encountered when partitioning a circuit's structure after allocating and scheduling the data-path units: sharing the functional units without regard to their interconnection often prevented their placement in the target FPGAs or resulted in poor utilisation of their pin or logic resources. By grouping operations with common input or output values in to 'clusters' and repeating the process so that a cluster has the potential to be

a sub-set of another, the authors were able to confine the subsequent stage of scheduling and data-path allocation to an additional level of hierarchy above that of the operation-level.

Structural partitioning of test circuits over two FPGAs using a Simulated Annealing [7] algorithm highlights the difference between the best solutions with and without clusters: without clustering, the logic utilisation of the two devices were 72% and 39% respectively; with clustering, logic was better distributed at 63% and 43% utilisation. An improvement was also seen in their IO requirements: without clustering, IO usage was reported to be 78% and 68% of available pins; with clustering, their IO requirements were reduced to 47% and 35% respectively.

Simulated Annealing was also used to perform partitioning in the work described by Peterson et al. [56], although the application of the algorithm differs from the latter in its use in simultaneously scheduling and partitioning a behavioural description (ANSI-C) over a multiple FPGA 'Custom Computing Machine'. Hierarchy above that of the operation-level is represented by partitioning the software specification at subroutine boundaries. In addition to a potential reduction in the size of the cutset, likely to accompany a restriction of the interconnection to those nets associated with the passing of function arguments, the partitioned functions are also in a form that is easily ported for execution as software on the host computer. As well as increasing the opportunity to explore how a program's behaviour may be best implemented in hardware, the availability of a sequential processor may also be a necessity in circumstances where the logic resources are insufficient or unavailable, as is likely to be the case when implementing floating-point arithmetic operations.

Both of these approaches required full reconfiguration of the devices which suited the many operations encapsulated in each procedure or function that was partitioned. As such, they were early examples of spatial partitioning using reconfigurable hardware.

Later examples described as Temporal Partitioning would appear when the partial reconfiguration of a single device was examined. An often cited approach is the work by Vasilko [57]. It is an early recognition of the relevance of HLS Scheduling to the temporal aspects of partitioning for reconfigurable devices. The approach uses List Scheduling [17] to determine the assignment of each data-path operation to a single temporal partition.

The method begins with a list of unscheduled operations and a resource-constraint on the maximum partition size. Each partition is created by removing an operation from the list and assigning it to the current partition until the resource constraint is exceeded, in which case a new partition is reconfigured or all the operations are scheduled. As is typical in List Scheduling, a priority function is used to sort the operations to be scheduled: in Vasilko [57] it includes a measure of how early a reconfiguration should start in order to hide it with the execution of operations already scheduled.

Two outcomes are possible: in the event of there being sufficient time to overlap the reconfiguration of an idle resource from an earlier partition, a partial reconfiguration takes place. Alternatively, the overhead in reconfiguring the operation is too great for a single operation and the entire device is reconfigured, incurring the same reconfiguration time but freeing more resources to be used in subsequent partitions.

With regard to the test circuits, the examples scheduled at the operation level are small, the largest being in the order of 34 vertices in a data-flow graph. The purely data-flow approach excludes explicit control structures and therefore the opportunity to overlap reconfiguration with the execution of structures such as finite loops.

The results proved what seems intuitive, that longer reconfiguration delays cannot be so easily hidden and as a result are more efficiently handled via full reconfigurations. Reducing the reconfiguration times encourages more partial reconfigurations and shorter critical paths because of the ability to hide the reconfigurations. The results were not based upon commercial devices which would have required thousands of cycles but on multi-context and single cycle reconfigurable devices.

List scheduling is the most frequently used method for temporal partitioning, a particular advantage being its linear execution time with respect to the number of operations to schedule. In [58], the authors took a similar approach by using list scheduling but targeted a practical implementation, citing the ability to execute the scheduling in linear run-time. The target architecture utilised an embedded CPU which enabled an on-line scheduler to use list scheduling for occasions when dependencies between the nodes are not known. List scheduling was also used for static graphs, supporting both compile and run-time approaches to temporal partitioning. The scheduling is applied at the task-level of abstraction and a case-

study showed how a Discrete Cosine Transform may be partitioned over 16 tasks, overlapping device reconfiguration with the execution of the tasks. A high degree of similarity amongst the operations within each task is likely to account for the acceptable ratios of reconfiguration to execution latencies reported in the results.

Unlike non-reconfigurable approaches, list scheduling approaches rarely re-use functional units in existing partitions, placing a greater emphasis on the size of the available resource rather than its type. The work of Bobda [59] relied on creating a set of partitions in such a way as to maximise the number of common operations between any successive pair of partitions. In circumstances when too great a reconfiguration delay would prohibit a device-level reconfiguration, the author proposes the sharing of functional units between partitions; he describes how this approach need not automatically result in an increase in the logic resources necessary to multiplex the input operands of their allocated instructions: in FPGA architectures where a look-up table is used to pass an input signal to a register resource, an additional data and select signal along with the updated boolean equation can be programmed into the existing table. Despite being a useful alternative form of temporal resource sharing, it does depend on having precise user-control over the technology mapping of individual data-path components, a level of abstraction that many hardware designers would be reluctant to design at.

A very different approach to temporal partitioning is taken by formulating the problem as an Integer Linear Programming task [60], using equation solvers to find a solution which meets the exact constraints used to enumerate the temporal partitioning. Examples of these constraints are: uniqueness assignment constraint- a single binding of a node to only one partition; data-dependence constraint: a variable must be written to before it is read and that implies its placement in a partition earlier in time than the reading partition.

Once all these variables are described, a commercial equation solver is used. The advantage in this approach is its ability to generate an optimal solution. The drawback in its use is that it requires an exact formulation of the architecture. This includes defining exactly all the properties required during partitioning. As a consequence, it is only suitable for problems with a very small number of variables.

Network flow techniques have been proven to be useful in modelling communication costs between temporal partitions. Approaches using list-scheduling often form partitions solely on a local temporal view (levelling [59]) without consideration of the cost of buffering communication. Network flow methods are specifically centred around finding the minimum cutset. The method is based on the Ford and Fulkerson min-cut max-flow theorem [61]. A physical analogy would be to consider a series of interconnected water pipes with a single source and sink and no loss of water when travelling from source to sink. By applying water at a steady rate, at some point the pipe with the smallest diameter will saturate with a maximum flow before the others; if cutting it would prevent the flow of water to the sink, it represents a minimum cut.

The logic gates and input/output nets of a digital circuit can be represented as a set of vertices and edges in a flow network. A circuit net is modelled by an edge of a single unit capacity. A saturated path from source to sink occurs when the flow rate equals the maximum capacity of one. When the network has achieved a maximum flow, there will exist at least one minimum cut capable of partitioning the graph in half. Since a saturated edge is of a unit flow, the size of the cut is given by the number of forward edges crossed between the two partitions. Crucially, all of the potential cuts will be of the same size and ultimately equivalent to the maximum-flow of the network. Selection of the actual partitioning cut can be made by considering the size of the vertices connected by the edges of the cut-set and the areas of the two partitions formed by separating the vertices.

A network flow approach would seem to be an attractive method since each max-flow computation reduces the requirement for circuit partitioning to comparing the areas of vertices. However, there are a number of hurdles which prevented its adoption. The most obvious difficulty is the time taken to repeatedly calculate the maximum flow; after each cut is made, the area of partitions is not balanced which requires repeated cuts to the larger partition and potentially as many max-flow computations as there are vertices. Another difficult lies in accurately representing a cut made across nets in a circuit, complicated by the fact that an output of a logic gate may drive multiple gate inputs which is often represented by a single net.

The Flow-Balanced Bi-partitioning (FBB) method of (Yang and Wong) [62] directly addresses these problems through the use of an incremental approach to calculating the maximum flow of a network, whilst selecting only vertices on nets which preserve a balance criterion specified for the resulting partitions. The authors also showed how nets with multiple terminals can be represented in a flow network through the use of additional vertices and edges.

The first application of FBB to temporal partitioning was described in [63]. The approach used recursive bi-partitioning to create a set of two partitions. A pair of source and sink nodes are selected and flow is applied between them. This is repeated until at least one saturated path is found. If cutting it would stop the flow between the source and sink vertices than it is declared as a potential mincut. Of all the potential edges, one is chosen where moving its associated vertex would improve the size of a destination partition selected with respect to a balance target. The benefit of this approach is that is can lead to a solution in polynomial time but requires the insertion of extra nodes and additional edges to represent multi-pin nets. One has to be careful to avoid the worst case which is twice the number of nodes and edges than in the original problem.

Force-directed Scheduling [64] has also been used as a basis for realising temporal partitioning in high-level synthesis. The objective is to find a schedule to meet a time-constraint whilst minimising the number of functional units required for instruction operations. To reduce the schedule requires an exploration of the concurrency of operations but that requires more functional units. Forced directed scheduling attributes the distribution of concurrency for a type of functional unit as an abstract measure of force.

The aim of force-directed scheduling is to schedule an operation to reduce the force and evenly distribute the parallelism of functional units across other units of the same type. A particular point of interest is that it measures the ramification of scheduling an operation on any related predecessor or successor operations. For example, scheduling an operation later might encroach on the control step of a dependent successor; assuming that both cannot be scheduled to the same control step, doing so would limit the choice of where to schedule the successor operation. The solution is to find an operation where there is a choice in where it can be scheduled to affect the number of functional units used e.g. an operation with mobility.

The ideal control step to schedule the operation would be where it does not increase the number of functional units used of a similar type. That is not to say no parallelism, but less parallelism over the possible choice of where to schedule.

Where an operation can be scheduled in more than one place becomes a probability of distribution. Subtracting the average use of functional units from that probability provides an indication of an operation's self-force. Therefore, a small value reflects a small self-force and a good place to schedule. A large force would be in a place where there are many of the same parallel units in comparison with the average and not a good choice.

A similar approach can be taken with related operations. A dependent operation may also have a mobility and choice of where to schedule. There is an average use of functional units of the same type during that mobility. Finding the difference between the average use of units before and after the scheduling of the operation in question reflects the effect it has on any dependents. A reduction in the average use of functional units after the move would suggest the control step is an unsuitable choice.

The final decision is based upon the sum of the self-force and the successor and predecessor forces. This is repeated for every control step where the operation might be scheduled. The control step with the smallest force will represent a smaller than average concurrent use of a functional unit and a minimal use of resources for that operation.

In [65], the authors describe a force-directed temporal partitioning algorithm which simultaneously considers resource sharing and partitioning. The purpose of the algorithm is to minimise the execution time whilst considering the sharing of functional units and how it might affect the size of a partitioning. The disadvantage of this approach is that allocation is based solely on local partition properties. A global view of partitioning was implemented in [66] where each operation on the critical path is initially assigned its own partition. The algorithm is than able to take a global approach by determining which of the operations from non-critical paths may join them in a partition. In doing so, it is able to consider both the resource sharing between their operations, as well as their execution latencies; no provision is made for the communication cost of signal cut by the partitioning.

The Genetic Algorithm approach described in [67] differs from the other approaches described previously, not just in its use of the genetic algorithm but its aim of multi-objective optimisation. Unlike the other approaches, it sought to minimise resource size, latency and communication costs. However, a disadvantage in the approach taken is the reliance upon only accepting moves during partitioning which improve the cost function, no ‘Uphill’ decisions are ever taken.

2.6 Architectures for Run-time Reconfiguration

The approaches described earlier regarding temporal circuit partitioning are based on the assumption that the architecture is a direct result of the topology inherent to partitioning.

A number of reconfigurable resources are defined and fixed wiring channels are used to connect them. Their properties such a size and placement are all a part of the trade-offs made during partitioning. As such these decisions are necessarily off-line because the relationships between the instructions are statically defined.

There are alternative approaches to run-time reconfiguration, where partitioning decisions are made on-line. Although the problem being addressed is very different to the use of reconfiguration for high-level synthesis, difficulty in implementing their architectures at the device-level often requires pre-defined off-line architectures that are made generic through use.

From a reconfigurable computing perspective, a programmable device such as an FPGA could be regarded as another memory to manage. The reality is that the encoding of information using an FPGA is several magnitudes larger than that which is used for a general-purpose microprocessor [3]. The obvious explanation for this is that a microprocessor has an existing architecture, where the fetching of instruction operands is inherent to the programming of the architecture. In contrast, an FPGA is programmable for a user-defined architecture. There are numerous resources and these are spatially apart and therefore require a description of how information between them is to be transported. As a consequence, the task of managing the device resources at run-time requires a level of abstraction to manage the complexity of programming the device.

The approach taken for existing programmable resources, such as computer architectures is to describe an Operating System to which existing memory management techniques can be applied [68]. The difficulty in this approach is the proprietary nature of the devices. In order to prevent reverse engineering of customer implementations, manufacturers are reluctant to reveal sufficient information to enable the programming of their devices without their toolset. For example, Xilinx Inc. disclosed information regarding the encoding of logic resources but nothing to aid in programming the routing. A lot of effort in the reconfigurable community has been spent on working around the reliance on vendor tools [69]. This divides the reconfigurable computing community, those who incorporate vendor tools into their hardware abstraction [70] whilst others rely upon pre-designed architectures which are adjusted at run-time [8].

Research Tools such as Torc [71] include device vendor tools into their methodology and in doing so require a general-purpose microprocessor somewhere in the hardware implementation. This moves the focus to fast compilation; ultimately the end result must be the generation of device-level configurations at run-time.

In the absence of proprietary information, the approach is to limit the use of the vendor tools such as Placement and Routing which are the most time-consuming phases during implementation. An alternative is to rely upon circuit descriptions at the lowest possible level that allows textual description. For Xilinx devices this approach can be achieved through the use of the Xilinx Description Language (XDL) [72]. Tools like Torc rely upon manipulating device level configuration frames in accordance with circuit descriptions at higher levels of abstraction, XDL as well as C++ or Java. The approach used by these tools is to describe the structure of a circuit using a high-level language which is then automatically programmed with the equivalent device-level resources at run-time. Resources are set and reset with each new requirement for change in program behaviour and consequently FPGA configuration memory. This approach to run-time reconfiguration is particularly useful where there is already the overhead of an operating system factored into the hardware expense.

An alternative approach is to reduce the number of run-time decisions by adopting a specific architecture. A representative example of this approach is described in [8]. A model of a hardware operating system is presented which is reliant on the use of a generic pre-designed

one-dimensional architecture. It comprises a series of columnar ‘reconfigurable slots’ into which a user task could be reconfigured. On either side of this area are static resources committed to implementing the operating system entirely in hardware.

A portion of the slot is allocated to a ‘Task Communication Bus’ which is a shared communication channel that spans all available slots. In this way any task can exchange information regardless of its actual placement. The customisation occurs when a variable sized task is processed because it is must be implemented through more than one slot. The operating system ensures that an indivisible number of slots are used and upon completion the resources are returned to their unit size.

Use of the channel is highly restricted to guarantee no bus contention and no arbiter is described. This is due to bus communication being solely between adjacent resources; non-adjacent resources do not use a shared bus and depend upon fixed direct routing channels.

A different technique would be to reduce the need for a shared communications channel by writing and reading to dedicated memory [34,73]. The method used in both these implementations of a hardware operating system is to perform on-line scheduling and allocation of resources to tasks by device configuration frame-manipulation. This requires the use of JBits [48], a vendor supplied interface to the reconfigurable resources of earlier Virtex FPGAs that does not reveal proprietary information. In this way, the authors were able to experiment with scheduling and allocation techniques without requiring a pre-defined architecture [8].

Tasks could be one or two-dimensional (fractions of a column) and all communication is between a task and a shared external memory. This enables its parameters to be written or read by the operating system of a desktop computer. The approach relied upon there being no communication between the tasks but was able to dynamically allocate wiring resources between the memory and the task. As the authors stated, this could be adopted to enable communication between tasks using the shared memory.

A distributed approach to communication has its origins in the massively parallel computer architectures of the 1980s [11]. Circuit Switching was used to create a fixed path for connecting two or more communicating resources such as CPUs. The architecture has a

number of parallels with earlier FPGAs, where routing would be done through a series of neighbourhood connections.

A connection between non-adjacent resources required the setting of many switches arranged in a mesh topology for the duration of the communication. The connection would be ‘switched on’ only for the duration of the communication. In this way, the parallelism of the computer architecture could match the parallelism of the algorithm being described [74].

An example of how circuit switching can be successfully used for communicating between reconfigured tasks was encapsulated by the Reconfigurable Multiple Bus On a Chip (RMBOC) [75]. In common with other pre-defined architectures [8], the resources are modelled as one or two-dimensional resource slots. A large resource can be accommodated by the reconfiguration of consecutive slots. The architecture separates communication to the top part of the resource slots. This is a necessity because a drawback of the circuit switching approach is the difficulty in ensuring that switching points are not already allocated. In the absence of a wiring database available to computer based approaches [71], the use of routing resources must be constrained to an area, similar to the approach described by [8]. However, unlike a single bus which allowed only source and sink connections, the RMBOC utilises multiple buses connected at switching points.

Each switching point comprises a controller which distributes control information to its nearest left or right neighbour. Control information includes instructions used to distribute data between switching controllers, such as commands which open or close a channel. Although many buses may comprise the path, the path is reserved solely for the single communication transaction. No other pair of resources may use the path. Upon reaching the destination, an acknowledge command is returned through the same route. Parallel threads are supported because access to the switching controller is arbitrated through a round-robin scheme which selects between each of the adjacent directions and the local resource. Failure to obtain access to the bus is represented by a cancel command which is returned through the switch controllers to the source switch. This approach could also greatly increase the length of the clock period and to reduce that delay, registers are also inserted along the switches to enable communication to be pipelined. This occurs through a dedicated FIFO queue used to store commands at each switching controller.

The drawback with such an approach is that the hardware overhead associated with each controller is duplicated at each switching point. An extension of this approach is realised through the use of a packet based network of resources. The Network On Chip (NoC) [76] architecture is proposed as an answer to the multiple billion transistor chips that are becoming common place. The generic network comprises a diverse set of resources such as memory, DSP or logic, all communicating through packets. Unlike circuit switching, a reserved path is not necessary because each packet has the control information necessary to forward it to the correct destination. Crucially, no route is pre-determined and resources on the network can communicate concurrently by taking different paths.

A NoC comprises a number of routers and local processing elements. Information is divided into packets which encapsulate destination address and data. Any router from the source of the packet and along the path to the destination router will read this information and direct the packet towards the destination router and processing element. How packets are directed and how the routers are connected is the major consideration in designing packet switching networks.

The shortest distance between a source and sink router is not necessarily the quickest because of congestion at a router – caused by multiple packets that have been directed to the same router. A deterministic approach uses dimension ordering of the source and sink routers to find the shortest path. In a 2D Mesh topology, a packet would traverse along a row given by the X dimension of the destination coordinate until encountering the Y column/coordinate. Communication between future packets would travel the same path. The disadvantage of this approach is the absence of an alternative route should congestion be encountered. In addition, future packets using the same method would also encounter the same congestion.

Adaptive routing strategies such as reinforcement learning [77] enable each router to learn information from its neighbour which is collated in a routing table of the fastest routes between it and the destination router. In this way, a router along the path is able to bypass congestion and select a faster route. The drawback to this approach is the difficulty in guaranteeing through formal analysis that deadlock does not occur. In addition, the routing algorithm stored in a router is also more complex requiring greater resources.

The use of reconfigurable resources in a NoC architecture is examined in the Dynamic NoC architecture proposed in [75]. The 2D Mesh architecture is implemented on a Xilinx FPGA, enabling the routers to be partially reconfigured as logic resources when not in use. The authors cite this as the motivation for customising logic resources as opposed to using fixed processing elements on the network.

As with all reconfigurable architectures when decisions are taken to allocate resources at run-time, the availability of routing resources must be communicated dynamically between the routers. The router in a non-reconfigurable NoC assumes the existence of four adjacent routers in a 2D Mesh. This assumption could be no longer valid where adjacent router resources are to be reconfigured into logic ones.

The strategy taken is to reconfigure only groups of router resources. This guarantees an adjacent router is always available and also enables other routers to be reconfigured into logic resources. The approach is implemented in the routing algorithm by only forwarding packets to routers surrounded by more than one router. The authors suggest that a static routing algorithm achieve packet forwarding based on measuring the strength of the connectivity between clusters of routers.

Experimentation with the packet size was used to determine the area (logic and memory resources) and speed (clock frequency) of subsequent router characteristics using the second largest Xilinx FPGA and middle of the range devices available at the time. The results showed that for the largest packet size of 64 bits, the area resources were 46% (logic), 12% (memory) and 77.3MHz for a router using the middle of the range FPGA. Implementation of the same router on the larger device required 7% (logic), 4% (memory) and 73.7 MHz. These resource overheads are irrespective of the actual application. The authors concluded that the area overheads in synthesising the routers were great because their implementation did not directly map efficiently to the routing resources of the FPGA.

2.7 Summary

This chapter began by associating the moment when a resource is bound, as the means by which a reconfigurable resource can be distinguished from a programmable counterpart. It

pursued the binding analogy by recognising that an early binding of program behaviour is synonymous with a highly optimised use of resources, a characteristic of application-specific circuit synthesis. On the contrary, the latest time in which a program might be bound is during its execution, a characteristic of general-purpose programmable computer hardware; run-time reconfigurable resources share both these characteristics. This was recognised much earlier than could be realised in practice, as the literature described.

The reasons for using run-time reconfiguration include: reconfiguring resources between tasks with differing purpose (functional); reconfiguring resources between the same tasks implemented differently (algorithmic); reconfiguring in the presence of a fault (architectural). A myriad possible uses led to the necessity to automate their implementation and the development of hardware compilers at high and very low levels of abstraction.

Reconfigurable architectures are constrained by how the resources communicate with one another. Database approaches are the least restrictive but also the slowest due not only to the large choice of routing options available but also the inclusion of vendor tools in their generation and updating at run-time. Reducing the number of programmable routing resources certainly reduces routing decisions as shown in the bus-based architectures. Networked resources do reflect the parallelism of wiring resources available but do not map efficiently due to the proprietary nature of the FPGA device configurations.

Despite there being no shortage of architectures and approaches to implementing run-time reconfiguration, the reality is that FPGAs are the shortest route to their implementation. With that in mind, the use of high-level synthesis tools are a prerequisite to exploring the benefits and limitations associated with off-the-shelf programmable and reconfigurable hardware.

Chapter 3

Behavioural Synthesis

In this chapter, the concepts behind behavioural synthesis are introduced through the use of MOODS, the synthesis tool that provides the framework for much of the work undertaken in this thesis. The synthesis of an error-correcting communication system exemplifies the steps followed during behavioural synthesis which begin with an algorithmic specification of the circuit and end with an equivalent structural Register Transfer Level description − suitable for RTL synthesis and ultimately a vendor specific device implementation. The chapter concludes with a description of how the analogy of a software approach to hardware design can be extended to the device-level in the form of a late-binding of subroutines.

3.1 Circuit Abstraction and Synthesis

Figure 3.1 depicts the possible stages of circuit representation encountered during circuit design. Automation in the form of Computer-Aided Design (CAD) tools are utilised to a varying degree during each stage; more so in the lower stages, where the task of circuit design becomes increasingly complex and therefore time consuming and prone to human error. Historically, the deployment of CAD tools has risen through the stages of circuit abstraction, providing a means of addressing the problems associated with circuit complexity.

At the apex of circuit abstraction, an *Algorithmic* description expresses the behaviour of a system and not how it might be implemented; it is very much sought after because it continues the trend in hardware design of removing detail through each step of circuit abstraction. In doing so, it begins to mirror the history of software development, which also followed a similar path: from early programs written in machine or assembler code, to the present day ubiquity of high-level programming languages and their software compilers, which provide a means of describing program behaviour without being too constrained by the idiosyncrasies of the target hardware.

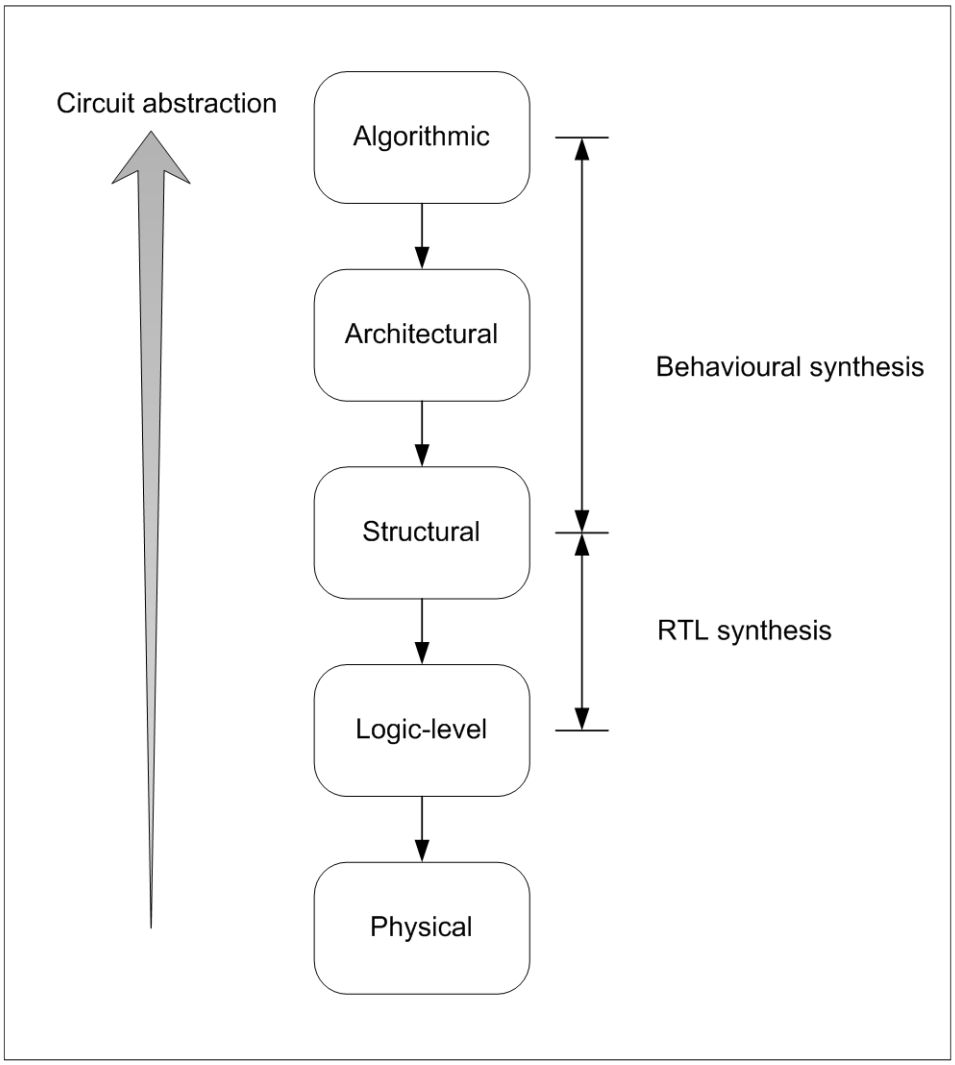


Figure 3.1: Abstraction in circuit representation.

Behavioural or *High-level Synthesis* [78,79] embraces the *Algorithmic* level of abstraction by automatically generating a number of *Structural* representations solely from a single behavioural description. Although functionally equivalent to the next, each structural description will differ in terms of its size, speed and power consumption. Ultimately, the structural representation chosen is the one which best meets the user's criteria regarding the applicable circuit characteristics.

As an intermediate step towards a structural representation, the *Architectural* level of abstraction seeks to identify how the sub-systems of the specification might be implemented. For example, in contemporary synthesis, such as hardware/software co-design [80], the occurrence of many 32-bit arithmetic and logic word operations in the data-path might

suggest the use of an ALU within a general purpose processor. Other data-path tasks might be highly parallel and feature irregular bit-level operations – often more efficiently realised using application specific hardware on a programmable fabric [81].

At present, the majority of industrial circuit synthesis takes place at the Register Transfer Level (RTL) or *Structural* of abstraction. At this level, the behaviour of the algorithm is embodied by at least one Data-path and an associated Controller. The data-path is characterised in terms of data transfers and transformations between functional units and storage elements; realised as a network of functional units, such as arithmetic and logic units (data transformation) and connected through multiplexors, buses and directly through data nets (data transfer).

The responsibility for scheduling when the data-path operations take place and how, if any, of the functional units are shared, is left to the circuit designer during Controller Synthesis. As with the data-path, there are choices in how to implement the controller. One solution to realising a Finite State Machine could be to use combinational and sequential logic i.e. a hard-wired approach. Alternatively, the encoded control sequence might be stored in a ROM, as in the case of a programmable micro-coded [14] controller.

At the *Logic-level* of abstraction, circuit behaviour is described in terms of Boolean equations and may be modelled as graph of logic operators, to which boolean minimization and algebraic methods (e.g. Operator Decomposition, Extraction, Factoring, Substitution, and Collapsing) are applied during *Logic Synthesis*.

In practice, logic synthesis is performed as part of RTL synthesis, enabling automatic optimisation of the combinational and sequential logic described at the structural level.

Despite being mathematical and obviously technology independent, the application of the algebraic methods will have a tangible effect on the equivalent representation of the circuit at the next level down: decomposition or substitution of boolean functions can reduce the number of logic gates needed, collapsing boolean functions will reduce their logic depth; logical optimisation, such as these at the gate-level, will impact the size and delay of a circuit at the physical level.

Sequential logic is also receptive to optimisation, the encoding of a state machine for example, where the resource requirements of different state encodings may be evaluated. Synplify [82], in the absence of a specified encoding, will choose the style of encoding based upon the number of states described in the state machine i.e. sequential when there are less than 4 states, one-hot for states less than 24 but greater than 4 and gray for any other number of states.

The last step in logic synthesis is *Technology Mapping*, where the logic equations are translated to a specific technology and vendor, such as the FPGA look-up table (LUT). This is achieved by partitioning the graph over K LUTs, whilst optimising for user constraints on area, delay, routing and power [83]. The output of logic synthesis is an optimised circuit, expressed in the form of a standardised net-list e.g. EDIF, describing the circuit using the component primitives of the target technology and vendor; during the next stage of abstraction, it will form the basis for a transformation into a physical implementation of the circuit.

As illustrated in Figure 3.1, the *Physical* level of abstraction is the lowest level of circuit representation. In practice, it covers a range of circuit detail, from a model of the delay a signal might experience along a particular segment of FPGA routing, to the manipulation of transistor aspect ratios during Full Custom ASIC design. In the context of circuit synthesis presented in this thesis, design automation in the physical stage is associated with implementing the circuit using vendor and FPGA specific tools. These tasks include a further refining stage of *Technology Mapping*, using detailed propriety information about the target architecture; determination of where the device primitives are *Placed* in the architecture − in accordance with the optimisation goals and constraints; closely coupled with an attention to the minimisation of the *Routing* required to connect the circuit components to one another and the external input/outputs. Such tasks can be carried out autonomously by the vendor tools, however, user input is often necessary to achieve the performance requirements of the circuit implementation.

One vital aspect of circuit synthesis which is common to all stages of abstraction is the necessity to verify the correctness of the circuit being modelled. This becomes an increasingly time consuming task, as the length of time spent in verification is proportional to the degree

of detail used to represent the circuit. For example, with no explicit reference to time at the algorithmic level, a purely functional simulation can be carried out in much the same way as software is debugged using a compiler.

At the structural level, simulation begins to become a bottleneck in the development cycle, at which point hardware Emulation may be utilised to accelerate the simulation process. This has particular resonance in reconfigurable logic applications because many commercial emulation systems [84] are FPGA-based, exploiting their re-programmability to implement different parts of the design being tested.

3.1.1 Behavioural and RTL Circuit Synthesis

In order to exemplify the differences between a Behavioural approach to circuit design and its RTL counterpart, consider the task of synthesising a parity generator for encoding data using Bose-Chaudhuri-Hocquengheim (BCH) codes [85]. One such code is characterised as being 15 bits in length, 11 of which directly represent the data to be encoded. The remaining 4 bits are allocated to parity, enabling the location and correction of any single bit transmission error of the codeword. The generator polynomial for the code is $g(x) = 1 + x + x^4$. Chapter 7 provides an overview of the use of BCH codes in the context of a run-time reconfigurable message coding scheme.

One approach to realising the BCH encoder would be to describe the structure of a suitable circuit. A solution might be to describe a data-path containing a Galois Linear Feedback Shifter Register (LFSR) comprising four stages (given by the term in the polynomial with the highest power), where the output of each is wired to the input of the next stage or to an Exclusive-Or gate, as depicted in Figure 3.2.

One of each pair of inputs to the Exclusive-Or gates corresponds to a particular coefficient of the generator polynomial, whilst the other is driven by the feedback between the maximum and minimum terms. A controller (partially represented by the And gate in the feedback path and Multiplexor on the circuit output) would sequence events, such as the resetting of the counter after the last message was encoded or ensuring that the current message is simultaneously shifted into the counter and in to a communication channel, during the 11 clock cycles required to encode the message data. It would also oversee the formation of the

codeword by flushing the counter for a further 4 cycles, revealing the parity nibble to be appended to the message.

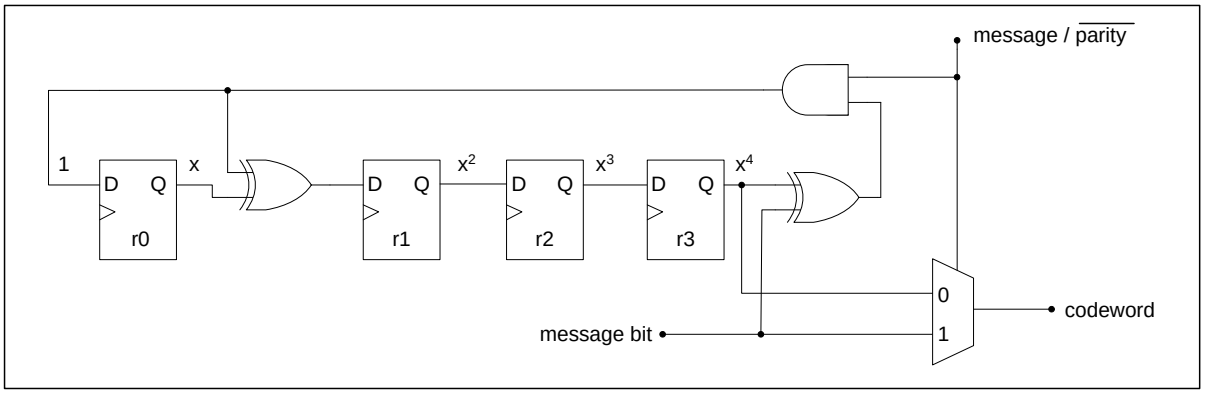


Figure 3.2: BCH message encoding using a Galois LFSR.

Figure 3.3 shows a VHDL RTL description of the LFSR encoder alongside several other sub-systems, in a form that would enable an RTL synthesis tool to automatically generate a circuit capable of encoding a message into a BCH codeword.

When the component/architecture hierarchy is flattened, VHDL essentially models a circuit as a number of parallel processes, each of which can communicate with another through a series of inter-process signals which are updated synchronously in step to a global clock. With that in mind, the description of the coding circuit is divided into a Finite State Machine controller represented by the *'ControllerSeqential/Combinational'* process constructs and three data-path processes, also named in accordance with their function: *'messageFetch'*, *'messageEncode'* and *'messageParity'* respectively. All processes are connected through a number of signals, declared at the beginning of the architecture along with the individual states of the controller.

The execution of each data-path process occurs during a similarly named state of the controller i.e. process *'messageFetch'* executes during the *'messageState'* of the controller. In this way, the controller orchestrates the generation of the codeword by sequentially executing one or more concurrent subsystem, enabling each of their associated control signals in the following order: *messageState*, *encodeState*, *parityState* and *codewordState*. In addition, all

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;


entity encoder is
port(clock:in std_logic;
    reset: in std_logic;
              message:in unsigned(10 downto 0);
              codeword:out unsigned(14 downto 0));
end entity;

architecture rtl of encoder is

type state is (resetState,messageState,encodeState,
               parityState,codewordState);

signal presentState,nextState:state;
signal messageBit,parityBit:std_logic;
signal codeMessage:unsigned(10 downto 0);
signal codeParity:unsigned(3 downto 0);
signal count:integer range 0 to 15;
signal readMessage,encodeMessage,
       readParity,formCodeword:std_logic;

begin
```

controllerSequential:

```
process(clock,reset)
begin
  if reset='1' then
    presentState<=resetState;
  elsif rising_edge(clock) then
    presentState<=nextState;
  end if;
end process;
```

controllerCombinational:

```
process(presentState,count)
begin
  readMessage<='0';
  encodeMessage<='0';
  readParity<='0';
  formCodeWord<='0';
  case presentState is

    when resetState=>
      readMessage<='0';
      encodeMessage<='0';
      readParity<='0';
      formCodeWord<='0';
      nextState<=messageState;

    when messageState=>
      readMessage<='1';
      nextState<=encodeState;

    when encodeState=>
      encodeMessage<='1';
      if count<message'HIGH then
        nextState<=encodeState;
      else
        nextState<=parityState;
      end if;

    when parityState=>
      readParity<='1';
      if count<codeword'HIGH then
        nextState<=parityState;
      else
        nextState<=codewordState;
      end if;

    when codewordState=>
      formCodeword<='1';
      nextState<=messageState;

  end case;
end process;
```

messageFetch:

```
process(clock,reset,message,readMessage,
        encodeMessage,readParity)
begin
  if reset='1' then
    codeMessage<=(others=>'0');
    messageBit<='0';
  elsif rising_edge(clock) then
    if readmessage='1' then
      codeMessage<=message;
      count<=0;
    else
      if encodeMessage='1' then
        messageBit<= codeMessage(count);
        count<=count+1;
      else
        if readparity='1' then
          count<=count+1;
          messageBit<='0';
        end if;
      end if;
    end if;
  end if;
end process;
```

messageEncode:

```
process(clock,reset, messageBit,readParity)
 variable registers:std_logic_vector(3 downto 0);
 variable feedback0,feedback1:std_logic;
begin
  if reset='1' then
    registers:=(others=>'0');
    parityBit<='0';
  elsif rising_edge(clock) then
    if encodeMessage='1' or readParity='1' then
      feedback0:=(registers(3) xor messageBit);
      feedback1:=feedback0 xor registers(0);
      registers:=registers(2) & registers(1) & feedback1 & feedback0;
    end if;
  end if;
  parityBit<=registers(3);
end process;
```

messageParity:

```
process(clock,reset,parityBit,readParity)
begin
  if reset='1' then
    codeParity<=(others=>'0');
  elsif rising_edge(clock) then
    if readParity='1' then
      codeParity<=codeParity(2 downto 0) & parityBit;
    else
      codeParity<=(others=>'0');
    end if;
  end if;
end process;

codeword<=(codeParity & codeMessage) when formCodeword='1' else "000000000000000";

end architecture rtl;
```

Figure 3.3: An RTL description of a BCH message encoding circuit.

controller states are proceeded by the *'resetState'*, initialising all inter-process signals, the controller and process variables to a know value.

Interaction between the controller and data-path processes is as follows: following a reset, the controller enters the state *'messageState'* and enables the registering of the message via the *'messageFetch'* process. During each successive execution of the *'messageEncode'* state, the message is processed one bit at a time by the LFSR inferred from the *'messageEncoding'* process, until the controller has transmitted all 11 message bits. The last interaction between the processes involves a third process labelled *'messageParity'*. As its name suggests, it forms the parity part of the codeword by accumulating the parity bits generated at the output of the LFSR, the result of inputting dummy message bits during 4 cycles of the controller's *'parityState'*. The final state of the controller *'codewordState'*, outputs the BCH codeword by concatenating the message and the parity signals, after which the next state *'messageState'* is also the beginning of the sequence when another message can be registered; the controller sequence is repeated indefinitely.

What is evident about the style of the coding for the controller and data-path processes is the amount of detail which must be present for a synthesis tool to infer the correct circuit components. For example, the variable *'registers'* used within the *'MessageEncoder'* process retains part of its former value when assigned, suggesting a memory implementation since the variable is read and written to in response to a clock signal. As the assignment occurs on the rising edge of the clock signal, the memory is interpreted by the synthesis tool as an edge-sensitive D Flip-Flop.

The vector width of the variable would generate four instances of the Flip-Flop, two of which are written to using the previous state of the variable, while the remaining two are written to using the value of the variable *'feedback'*. The value of this variable reflects the state of the message and controller mask bits, combined with the states fed-back from elements of the vector variable *'registers'*, identified by the terms of the generator polynomial. An asynchronous reset of the flip-flop is also inferred by the tool, based upon the conditional test of the state of the reset signal and its independence to any changes in the logic state of the clock signal.

Describing the encoder in this way would fix the architecture to serial processing of the message bits, requiring 18 cycles to produce a codeword. In doing so, the architecture fully utilises the data-path components on each encoding cycle which reduces the component count when implementing the circuit.

An alternative approach to synthesising the encoder is to describe its behaviour and specify through resource constraints, the kind of physical characteristics required of the resultant architecture, thereby allowing a behavioural synthesis tool to automatically generate a compliant architecture and equivalent RTL description. Figure 3.4 shows one possible behavioural description of the BCH encoder.

The program description explicitly captures the behaviour of the encoder as a series of state transitions which are taken with respect to the value of each message bit being encoded. After the last message bit is processed, the value of the final state is used as parity and forms the codeword by appending it to the original message.

At first glance, an obvious difference between the two listings is that the behavioural encoder can be represented by a single process, containing fewer lines of code than the RTL counterpart. This can be explained by the operation of the encoder being explicitly decomposed into sub-systems in the RTL coding, all of which have implicit control and data-path equivalents within the behavioural description.

A more subtle distinction is that the function of the encoder is made more apparent when its behaviour is described. For example, the relationship between the value of the state and the effect that each message bit has on whether the next state is odd or even; this relationship was relied upon during the Viterbi decoding of the code-words, as described in Chapter 7.

On close inspection of the statements contained within the process, the coding makes use of an infinite loop to separate the assignment of variables or port signals in the process body from their initialisation and without explicit reference to a reset signal, as is the case for all the RTL process descriptions. In addition, there is no requirement to carry out any variable or signal assignment based upon the clock or reset signals, since the Finite State-Machine controller, the data-path and control signals relating them are all generated automatically from the behavioural description.

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

entity encoder is
port(message:in unsigned(10 downto 0);
    codeword:out unsigned(14 downto 0));
end;

architecture behavioural of encoder is
begin

 process
  constant q:integer:=8;
  constant messageLength:integer:=11;
  constant generator:integer:=3;
  variable state:integer range 0 to 14;

 begin
 codeword<=(others=>'0');
 wait for 20 nS;
 loop
  state:=0;
  for i in 0 to messageLength-1 loop
   -- moods unroll ◄····· synthesis directive
   if state<q then
    if message(i)='0' then
     state:=2*state;
    else
     state:=to_integer(to_unsigned(2*state,4) xor to_unsigned(generator,4));
    end if;
   else
    if message(i)='0' then
     state:=to_integer(to_unsigned(2*state-q,4) xor to_unsigned(generator,4));
    else
     state:=2*state-q;
    end if;
   end if;
  end loop;
  codeword<=message & to_unsigned(state,4);
  wait for 20 ns;
 end loop;
 end process;
end;
```

Figure 3.4: A Behavioural VHDL description of the BCH encoder.

The encoding of the message bits is described within the body of the ‘for loop’. Unlike the sequential statements in the RTL processes, which are automatically inferred as a series of combinational logic blocks, their hardware interpretation in behavioural synthesis will vary depending upon the constraints placed upon the tool; an area constraint might enable the chaining of individual combinational logic units in a single clock cycle or it might force their sharing across multiple cycles.

The reader will have noticed the synthesis directive '-- *moods unroll'* occurring as the first instruction within the for-loop, without it, the hardware required would be repeatedly executed and shared by each of the 11 iterations of the loop. While not wishing to consider how the instructions become hardware within the loop at this stage, is clear that in common with the RTL description, a serial processing of the message bits would be the main characteristic of the generated architecture.

Instructing the compiler to replicate the instructions by unrolling each loop iteration provides an opportunity to dramatically increase the number of message bits that can be simultaneously processed, also requiring an increase in the number of dedicated hardware components. A higher degree of parallel processing of the message bits would be the dominant characteristic of this alternative circuit architecture. The degree of instruction parallelism is dependent upon how the instructions are allocated to data-path components and when they are scheduled to execute; these in turn would influenced by the importance the user places on the characteristics of the final architecture, such as the maximum number of cycles it would take to execute.

The last sequential statement to be repeatedly executed in the loop is the *'wait for 20 ns'* statement. It specifies that the encoder should update the 'codeword' signal at intervals of 20 nanoseconds. As this is an un-timed behavioural description of the encoder, the exact time when the 'codeword' signal is updated will be dependent on how the previous VHDL statements are scheduled as operations during synthesis. The relevance of the *'wait for'* statement is limited to simulation only, where precise signal timing would be relevant if the encoder was a component in a larger circuit specification with a circuit structure; otherwise simulation of the behavioural description would be to ensure correct function only.

Features such as these, which when taken as a whole, help the user to focus on the specification of the design rather than details of its implementation, e.g. the generation of control signals for individual hardware units and when they should be active. This is evident from the contents of the process description which would not be that dissimilar if coded in a sequential high level language such as 'C'.

If the number of execution cycles and components required for each of these architectures are plotted as a 2-dimentional graph, as is illustrated in Figure 3.5, it is possible to visualise a

small ‘design space’ of encoder architectures generated by the Behavioural and RTL approaches to circuit synthesis.

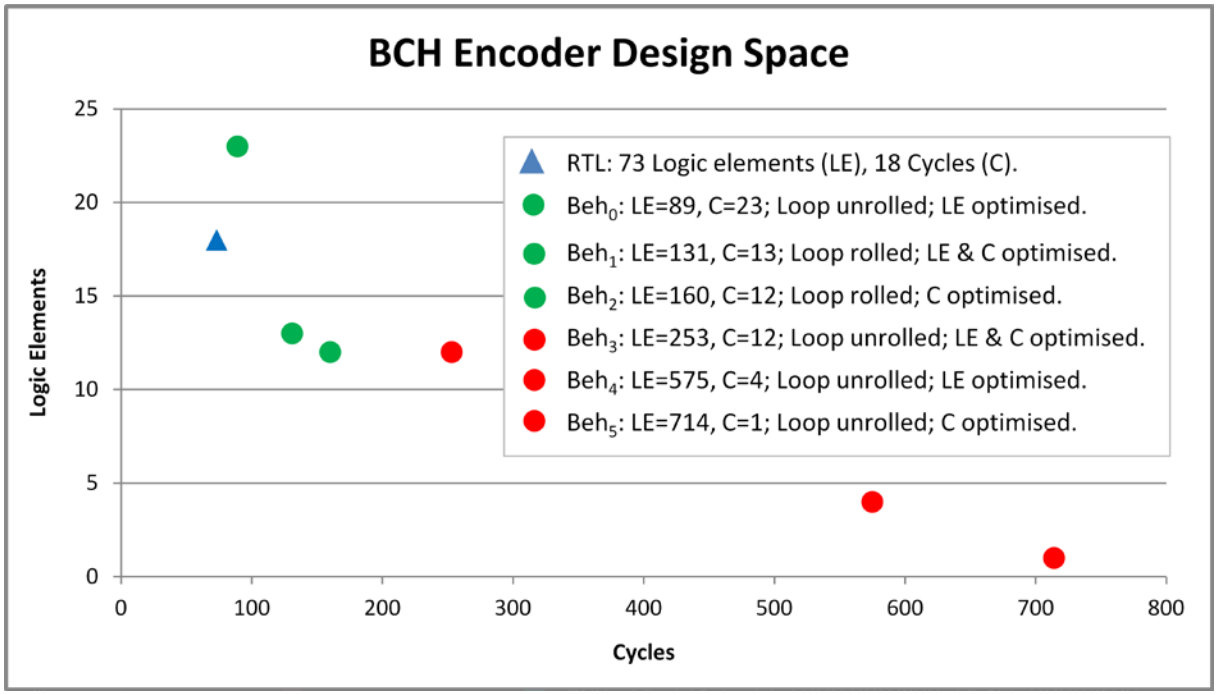


Figure 3.5: Circuit architectures generated by RTL and behavioural synthesis.

The graph illustrates six very different circuit structures, *Beh*$_0$ to *Beh*$_5$ respectively, which could be chosen as alternatives to the single *RTL* solution. Each was automatically generated using MOODS (Multiple Objective Optimisation for Control and Data path Synthesis) [5], the in-house behavioural synthesis tool, by simply choosing to keep or unroll the loop contents and by varying the optimisation priority to minimising the number of components, cycles or both, in the generated architecture.

A single point on the graph represents the number of cycles and basic elements of logic required, after the manual and behaviourally generated RTL descriptions are synthesised to Xilinx FPGA primitives [50]. The number of cycles is a measure of the time taken to generate a BCH codeword from a message. The size of the circuit is given by the number of basic elements of logic encompassing combinational and sequential logic through a tally of look-up tables and registers used in the FSM controller and data-path circuits.

The results show the extremes of possibilities that can be accomplished using behavioural synthesis, from Beh_5, the most parallel solution, where loop unrolling and component sharing of only mutually exclusive instructions lead to a circuit comprising 714 components, all of which were executed in a single cycle; to the architecture of Beh_0, where the consequence of choosing not to unroll the loop and no restrictions placed upon component sharing, resulted in a small component count of 89 components, requiring the greatest time (23 cycles) to form the codeword. Of course, these solutions were generated in response to specifying a higher priority between the number of components and cycles required of the architecture. In practice, both are likely to be of equal importance and would produce a compromise between these two extremes, as exemplified by Beh_1; characterised by a component count of 131 components and requiring 13 cycles to produce the codeword.

It should be clear to the reader that in order to explore a different point in the encoder design space, the RTL description must be re-written in order to infer a different architecture. No change is required to the behavioural description, other than to experiment with different compiler directives, such as loop unrolling. In this way, the size and shape of the encoder design-space can be automatically explored by the user changing compiler directives and constraints placed upon the desired properties of the final architecture.

3.1.2 A Renewed Role for Behavioural Synthesis Tools

The industrial use of behavioural synthesis tools has yet to fulfil the role anticipated by the academic community, where research into techniques and tools is in its fourth decade [86]. Judging by the current product lines of major tool vendors this could be about to change.

Several authors [87,88] provide a historical narrative of the development of academic and commercial tools, also offering an explanation as to why earlier offerings of behavioural synthesis tools failed to be adopted as the suitable level of abstraction at which to design hardware. The simplest explanation is that the quality of results offered by the earlier generation of commercial tools, such as Behavioral Compiler (Synopsys) and its industrial peers – Monet (Mentor graphics) and Visual Architect (Cadence), were not on a par with those generated at the Register Transfer level (RTL).

Describing hardware at the RTL of abstraction was obviously sufficient to cope with the increasing complexity of circuits because it would be several years later that high-level synthesis tools for ASIC and FPGA design would once again be offered by the same vendors: Synfora Synphony C Compiler (Mentor Graphics), Catapult C (Calypto design/ Synopsys), C to Silicon and Cynthesiser (Cadence). In addition, the vendors also found themselves part of a growing group of commercial [89] and open-source tool suppliers [90], all promising to increase the productivity of the hardware engineer. Very prominent among this group are those who solely target FPGA devices.

For approximately the same size of silicon, the number and type of programmable resources offered on an FPGA has changed considerably, when compared to devices which were available to the previous generation of HLS tools: resources now include system level components, such as hard and soft processors, dedicated multipliers optimised for digital signal processing and the necessary interfaces with which to connect them.

Design space exploration at the RTL level makes it very time consuming to alter the properties of any of the system level components when attempting to meet the required hardware constraints. There is a renewed role for HLS tools, where designing at a higher-level of abstraction makes designs re-usable and faster to verify because any changes that need to occur can be automated at lower levels of circuit abstraction.

With regard to the quality of the generated circuits, it is difficult to obtain an objective opinion from the vendors themselves. As you might expect, exemplar designs serve only to highlight the individual strengths of their tools. However, a number of commercial tools have been independently reviewed [91] and their quality found to be equivalent to circuits that were designed and optimised at the RTL of abstraction. Similar findings have been reported by the academic community [87], who also found that the quality of automatically generated circuits favourable with those that were manually designed. There is also evidence [88] of the productivity gain associated with exploring the design space and the early detection of design and coding errors during verification.

3.2 MOODS Behavioural Synthesis

Figure 3.6 illustrates the stages of transformation undertaken by a circuit specification at different levels of abstraction and the key components responsible in the MOODS Behavioural Synthesis System. As depicted, a synthesis session begins with an algorithmic description written in the VHDL hardware description language and a set of user supplied optimisation criteria: the target size, delay and clock period required of the generated structure in the chosen technology and the importance (often of equal priority) which the user attaches to achieving the optimisation goals.

The first change in circuit abstraction occurs during the *Compilation* of the behavioural VHDL in to an intermediate language known as ICODE. Its purpose is to substitute what can often be complex high-level VHDL statements for simple 2 input operations; these will later be assigned discrete components at the structural level of abstraction.

Some of the tasks performed during the compilation of the source code are akin to those carried out by any sequential language optimising compiler i.e. verification of code syntax and use of semantics, code optimisation: dead code elimination, loop unfolding and constant folding – to name but a few [42]. However, some optimisations take advantage of the intention to create customised hardware, such as bit-vector packing [92] or hardware specific operator substitution [93].

As with all compilers, the output of the compilation stage is in a form closer to the hardware on which the program description is to be implemented i.e. assembler/machine code for sequential language compilers.

In a similar sense, the output of the optimising VHDL compiler is a step closer to a hardware oriented description: the code generated at its output is quite literally an ‘Intermediate Code’ (ICODE) towards a structural representation of the algorithmic specification. For example, all variables are represented in ICODE as bit-vectors, where each vector element might ultimately be realised by a single register in synthesised circuit.

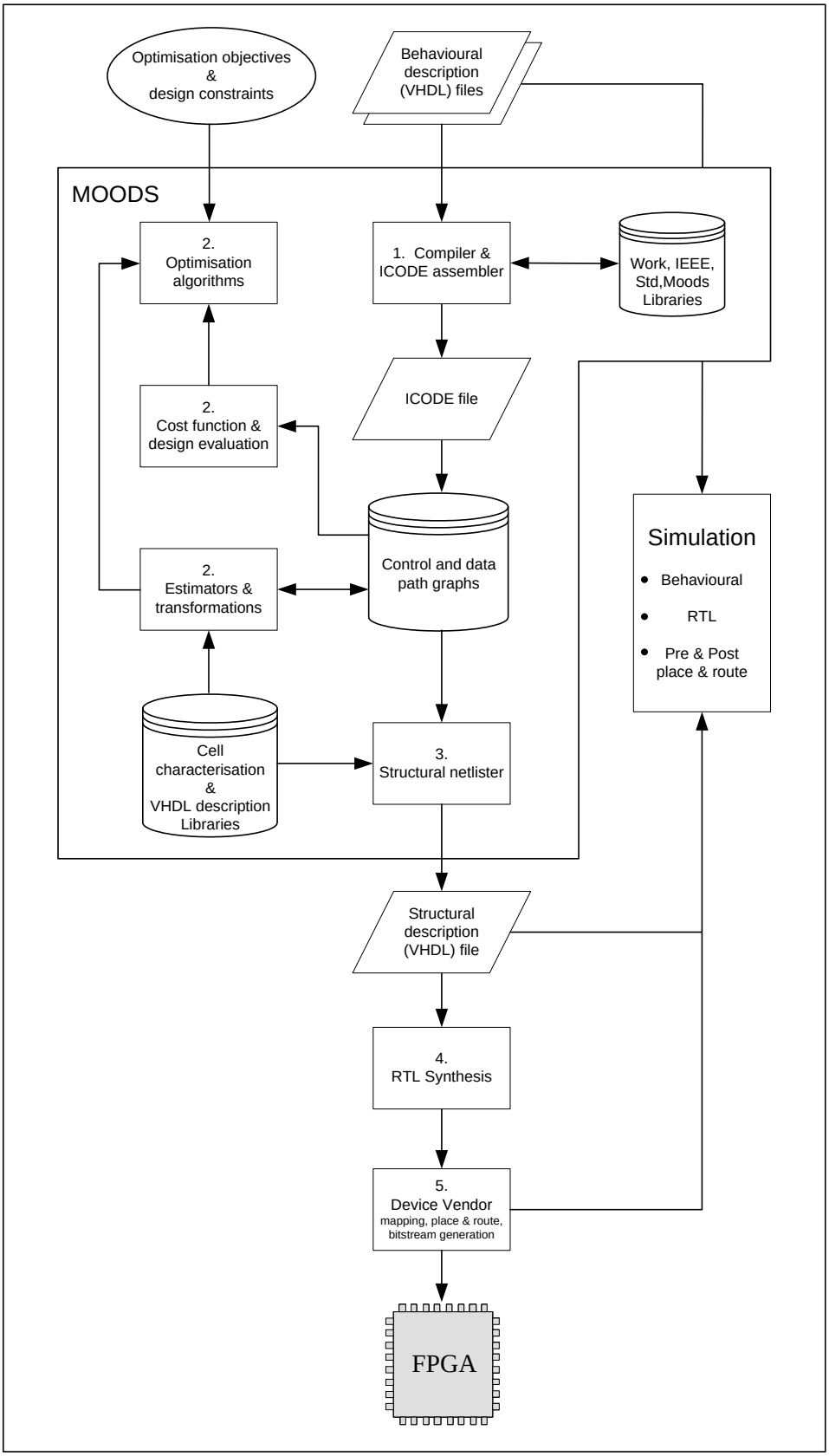


Figure 3.6: MOODS – centric digital circuit synthesis.

The semantics of the ICODE language express the functionality of the behavioural description, as well as model the sequential and concurrent aspects of the algorithm's data-flow. This is invaluable to its next step of representation – the construction of the *control* and *data path graphs*.

The motivation for modelling the ICODE description as a set of data and control graphs is to enable their automatic optimisation through graph-based *Transforms*. The execution sequence of the instructions is reflected in the *Scheduling* of each instruction to a node in the control graph. Similarly, the behaviour of each instruction is *Allocated* to a functional unit in the data-path.

The control and data-path nodes do more than represent when a certain type of instruction is executed, they also indicate how it will be achieved: each node is *bound* to a physical characterisation of the instruction in the target technology. In the context of the work described in this thesis, that technology is the Xilinx Virtex [6] family of FPGAs.

The time spent in the first stage of the design flow is now rewarded with the opportunity to automatically optimise the graphs (and ultimately the circuit), in ways which aim to meet the user's requirements regarding its physical characteristics. This is achieved in the second stage of synthesis, under the direction of the *optimisation algorithm* which applies individual *Scheduling*, *Allocation* and *Binding Transforms* to selected nodes of the control and data-path graphs.

The potential effect of a transform is *estimated* using a number of design metrics. These include: area (Xilinx CLB [6] slices), delay (critical path) and clock period (execution time of longest control node). A cost function is used to quantify whether the changes to each of the metrics combine to bring the circuit's structure closer to the user's objectives (in which case it is accepted and performed), or further away, when the nature of the optimisation algorithm determines whether to conditionally accept circuit degradation (e.g. Simulated Annealing [7]) or reject it outright, as would be the case with 'Greedy' [17] algorithms.

Upon completion of the optimisation stage, the purpose of the third stage is to generate the RTL VHDL description of the optimised circuit. The cell library is consulted to retrieve the VHDL description for each of the control and data-path nodes. The topology of their graphs

(modelled within the data structures), acts to guide how the cell descriptions are related in the structural description.

The control graph is described as a One-Hot state machine, although its final realisation is determined during RTL and logic synthesis. During each control state, the activation of its associated data-path components is described using boolean equations which are based upon the conditional control signals modelled in the data-structures. These structures also direct how the input and outputs of all data-path units are connected. For example, multiple inputs to a data-path node indicate that it is shared by more than one operator, thus implying the insertion of a multiplexor.

In the context of the work presented in this thesis, the remaining stages of circuit synthesis employ the use of third party commercial tools. Simplify Pro [82] is used during the fourth stage, to perform RTL and Logic synthesis. State Machine Encoding, Logic Optimisation and Technology Mapping are all tailored to the Xilinx Virtex [6] family of FPGAs.

As shown in Figure 3.5, there are a number of junctures in the design flow where the circuit description may be *simulated* to verify its functional correctness and others, where increasing levels of structural detail maybe used to verify the timing of its physical characteristics.

Simulation of the behavioural VHDL description is performed to confirm that the algorithm captures the functionality of the specification. In the absence of any structural or timing detail, simulation is very fast i.e. in the order of seconds. Once its behaviour is assured, it can act as a reference for the functional simulation of the circuit at each stage of circuit description.

The structural VHDL description is simulated with reference to a clock cycle and at this stage is generally technology independent and therefore relatively fast (measured in minutes). Simulation at this level provides a means of verifying that the generated Finite State Machine and Data-Paths preserve the behaviour of the algorithm. In addition, it returns an initial characterisation of the circuit i.e. an early identification of the Critical Path(s).

Simulation during the last stage of the design flow runs parallel to processes which add increasing levels of implementation detail. For instance, the circuit netlist generated during RTL synthesis can be converted to an equivalent VHDL description in terms of the device

primitives, using unit delay models from the UniSim [50] library. The motivation in doing so would be to verify that the technology mapping and logic optimisation did not alter the behaviour of the circuit.

Similarly, after each stage of vendor and device specific implementation (Technology Mapping, Placement and Routing), a separate VHDL model could be generated to verify the behavioural correctness of the circuit. It would consist of device primitives taken from the SimPrim [50] Library and would provide a delay characterisation of the internal architecture of the target FPGA. The primitives represent the worst case signal propagation delays through the programmable elements, such as routing, logic block and IO block resources. A simulation at this level of detail is time consuming (many hours) and often necessary for fine tuning the performance of the circuit layout. Simulation may reveal that fine tuning is not sufficient and a return to an earlier stage is required to meet the design constraints; it can be an iterative process!

In the context of this thesis, the Xilinx Integrated System Environment (ISE 9.2i) [50] is utilised in the implementation of all practical work. Circuit simulation, through the use of ModelSim SE 6.2 [94] is performed at each of the levels of representation previously described.

3.3 MOODS and other Behavioural Synthesis Tools

In practice, Behavioural Synthesis encompasses many distinct transformations to a circuit's representation. As previously described, these transformations occur during stages of synthesis which are common to all behavioural synthesis tools; what differentiates them is how they are performed.

In this section of the chapter, we will briefly contrast the synthesis approach taken in MOODS with those of other academic and commercial HLS tools.

3.3.1 Specification Languages

There are at least three important circuit characteristics that any HLS language ought to express: the *parallelism*, *timing* and *interfacing* of the circuit's specification.

For Hardware Description Languages, such as VHDL, *parallelism* and *timing* are inherent to the way the circuit is expected to behave when simulated: all parallel processes update their signals at the same time – a property that is measured only during wait statements.

In RTL synthesis, this would translate into sequential logic which is updated in lockstep to a global clock – since processes are implicitly synchronised in the simulation model.

In MOODS behavioural synthesis, process synchronisation is explicit and left to the user, who is required to view a circuit specification in terms of independent but potentially Communicating Sequential Processes [95]. This approach relies upon MOODS to automatically determine the level of instruction-level parallelism within a process and requires the user to manually add a way of synchronising them, should they require data exchange.

As described in the previous section, software languages based on C have contributed to the renewed interest in HLS. Parallelism and Timing are not characteristics of sequential programming languages; these aspects have been added to the language through library extensions, e.g. the C++ class libraries required for SystemC [96] or explicitly through new syntax, as is the case in Handel-C [97].

The SystemC classes provide the constructs necessary to model hardware concurrency, in a way not dissimilar to VHDL – ‘Sc_threads’ in place of VHDL ‘Processes’; unlike VHDL, SystemC is able to use an object-oriented approach to programming through the C++ language.

Handel-C is a proprietary language used by the DK design Suite [97] HLS tool. It is a subset of C which borrows much syntax from the OCCAM [98] language. As such it requires that the user explicitly declare in the source code all parallel processes and the channels with which they communicate.

Of particular interest is the way the tool assigns each statement to a control state – in exactly the order coded by the user. This is in contrast to MOODS and other HLS tools, which perform an out-of-order assignment of instructions to control states, in an order intended to meet a delay target. An example of an ‘un-timed’ approach was given in section 3.1.1, in the

behavioural description of the BCH encoder. In the absence of any specific timing, it was MOODS that decided how many cycles the equivalent operations were going to take: these ranged from one cycle to twenty three cycles during design space exploration.

In practice, there are likely to be other sub-systems in the circuit specification e.g. where there's an encoder, there must also be a decoder! How the sub-systems communicate with one another will be determined by some sort of input/output protocol, hence there will inevitably be sections of code that must be 'timed' to specific cycles.

The cycle-accurate approach taken in Handle-C is an extreme way of ensuring that the generated hardware does not violate the required timing. A different but still implicit approach is to use the language constructs, such as the 'wait()' statement in VHDL or SystemC to create hard clock boundaries. In MOODS, the compiler generates a 'protect' instruction in the intermediate language used to represent the VHDL. The protection suggested by its name is to prevent instructions on either side of it from being assigned to the same clock cycle. A similar approach is taken by Agility Compiler HLS [89], where instructions between a SystemC 'wait()' also take one clock cycle.

The disadvantage in allowing the user to define hard clock boundaries is that they become an impenetrable barrier to non-input/output instructions on either side of them. A more discriminatory approach is to allow the user to annotate the section of 'timed' code through compiler directives or pragmas. This is the approach taken by the Cynthesizer [89] HLS tool, using 'pragma: protocol' labels and by enclosing the affected SystemC code in C++ block braces'{}'. As described in [99], a user-guided approach enables the compiler to parallelise 'timed' and 'un-timed' code sections, in ways it would not be able to do solely through analysis of the instruction dependencies.

In addition to expressing the parallelism and timing for sub-systems of a circuit specification, an HLS language must also model how they are to be connected together. In VHDL and consequently in MOODS, the interfaces of 'Entities' are connected through signals; this fixes how communication will occur in lower levels of abstraction.

The use of System level languages in HLS, such as SystemC [96] enable the user to describe the connections between the sub-systems during design entry, without describing how they might be implemented.

SystemC includes a *Transaction-Level Modelling* standard (TLM-2.0 [96]); when used in conjunction with SystemC class interfaces, it provides the user with a means of coding class methods for accessing communication 'Channels' that interface with SystemC modules.

The Agility Compiler [89] exemplifies this approach in HLS because it initially allows the user to consider the characteristics of the data sent through a SystemC Channel, without specifying the exact protocol used in its transfer; it might be synchronous (Bus), asynchronous (FIFO) or non-existent (point to point signals wires). Through the use of TLM, interfaces between sub-systems remain abstract in the behavioural description, without excluding them from design space exploration at lower levels in the design flow.

3.3.2 Compilation and Optimisation

The reader will recall how the design space for the BCH Encoder was created through the use of the '--MOODS *unroll'* directive: placing it in the body of loop required the VHDL compiler to duplicate its instructions for each bit of the message vector being encoded. As a consequence, MOODS was able to vary the number of loop instantiations used in the architecture of the encoder, with respect to the designer's constraints on its size and execution time.

All users of HLS tools require them to reproduce some of the architectural techniques (loop unrolling being one of them) which are frequently used during design space exploration at the Register Transfer Level. It is at the *Compilation* stage that the user can influence the size of the design space through the use of '*Synthesis Directives'*. The motivation behind this approach is to keep the code architecture-independent (behavioural); enabling the user to experiment with different architectural techniques without having to actually implement them in the coding.

Examples of common compiler directives are exemplified in Vivado [100] (originally AutoPilot) HLS. These include the overlapping of loop iterations *('set_directive_pipeline'*)

[100] and control over the number of memory ports available through Memory Partitioning *'set_directive_array_partition'* [100].

Streaming data-flow is an architectural technique which previously warranted an extended language (streams-C [101]) but is now available as an architectural directive in a general-purpose tool such as Vivado. A stream is a flow of data that is processed or stored continuously: a typical use would be in the image processing of pixels by a Sobel filter.

In MOODS, the *'--MOODS ram'* directive is used to treat an array variable as a memory block, accessible through a random address. In Vivado, a *'set_directive_stream'* would instruct the compiler to access the same array variable using sequential addresses in the form of a 'FIFO' memory.

The disadvantage with abstracting away implementation techniques, as in the case of the 'stream' directive, is the potential for user mistakes when working with behavioural and structural levels of abstraction: an array variable might be used in different ways in the behavioural specification; implementing it as a 'FIFO' memory in hardware could improve its performance in one aspect of the design, but unintentionally change the overall behaviour, should another aspect require a random use of the array!

Apart from using synthesis directives, another potential for automatically influencing the architecture is through the compilation passes themselves. MOODS has been modified to incorporate many new synthesis capabilities over the decades e.g. Multi-FPGA Partitioning [102]. One aspect which has not been fully explored is the optimisation passes of the Behavioural Compiler. An advantage in using a software language is the input from the software community and the inheritance of 'state of the art' compiler techniques. A popular example of an open source compiler is LLVM [103].

Examples of current HLS tools that use open source compilers are: Vivado [100] (LLVM), LegUp [90] (LLVM), GAUT [104], (GCC), BAMBU [105] (GCC). In [106], the authors evaluated 56 distinct LLVM compiler passes and experimented with varying the order in which they were applied to C programs taken from the CHStone [107] HLS benchmark programs. The results showed that through careful selection, a subset of passes resulted in a 16% increase in performance of the hardware generated using the LegUp HLS tool.

The stage after compilation is the common place for optimisation to occur – in the form of a graph representation of the compiled behavioural description, to which scheduling, allocation and binding techniques [79] are applied; under the guidance of an optimisation method.

The choice of graph representation can have an influence on the choice of optimisation method: the combined Control and Data-flow CDFG model [108] is used by constructive approaches for circuit optimisation without assuming an initial solution, such as in List Scheduling; the Extended Timed Petri Net ETPN model [108,109] is suited to an iterative optimisation approach, which is repeatedly applied to an existing solution, as used in Simulated Annealing [7]. The latter graph model is used in MOODS [5] and CAMAD [110], when compared to other HLS tools such as LegUp [90], they offer a clear distinction in the approach taken to optimisation.

The constructive approach to optimisation is popular in many HLS tools: CHIPPE [111], MAHA [112] and HAL [64] are among the early advocates for list-based scheduling approaches. More recent tools, such as LegUP, BAMBU and GAUT continue the constructive approach to optimisation. The advantage that these tools gain in doing so is the expectation of producing a solution in polynomial-time − this is not the case in an iterative approach; the disadvantage is that at any stage other than the last, the solution is always a partial one; in the case of list scheduling, this can limit it to relying upon a local and restricted view of the problem being solved.

Through an iterative approach, MOODS is able to take a global approach to optimisation because the instructions of the intermediate representation (ICODE) are already individually scheduled, allocated and bound in the control and data-path graphs prior to their optimisation; albeit in a manner which is not likely to meet any user constraints imposed on the initial structure which they embody.

Optimisation in MOODS and CAMAD takes place through graph transforms that individually aim to improve at least a single characteristic of the structure, such as minimising the critical path by merging pairs of control graph nodes. Unlike CAMAD, a set of transforms are also able to ‘undo’ the application of others: splitting a control state that had previously been merged, may take the exploration of the ‘design space’ in a new direction.

Through multiple transforms, different aspects of the structure may be targeted during the same synthesis session. Other HLS tools, such as the Cocentric SystemC Compiler [113] (based upon the Synopsys Behavioral Compiler [114]), CADDY [115] and BAMBU [104] perform data-path allocation as a separate step to instruction scheduling. Isolating these essential stages from one another makes it convenient to experiment with different heuristics. However, it does exclude the subtle inter-dependence between them, a characteristic that is exploited by the multi-objective approach taken in MOODS.

3.4 MOODS and Run-time Reconfiguration

In this penultimate section of the chapter, we describe only those aspects of MOODS which were essential in enabling the temporal and spatial partitioning of subroutines using run-time reconfigurable resources. A more comprehensive account of all aspects of MOODS can be found in [4,5], where MOODS itself is the subject of the research.

Figure 3.7 depicts key components in the MOODS Behavioural Synthesis, arranged to emphasise their involvement in the change of abstraction that is inherent when synthesising reconfigurable logic. Except for the library descriptions of structural components, all other forms of circuit representation in MOODS originate from the user-supplied behavioural specification; this is arguably the most influential stage in high-level synthesis and therefore the first stage of the figure to describe.

3.4.1 Behavioural Description

At this level of abstraction, the goal of synthesising reconfigurable logic is treated no differently from the way in which a user would code a behavioural specification. As the reader will recall from the earlier discussion on the role of languages in HLS (section 3.3.1), the style of coding that is synthesisable in MOODS is formulated around user-defined coarse-grain parallelism using VHDL ‘Processes’, within which the tool is responsible for determining all fine-grain (instruction-level) parallelism. Figure 3.8 exemplifies how concurrent and sequential aspects of a behavioural VHDL description can be partitioned across functional boundaries.

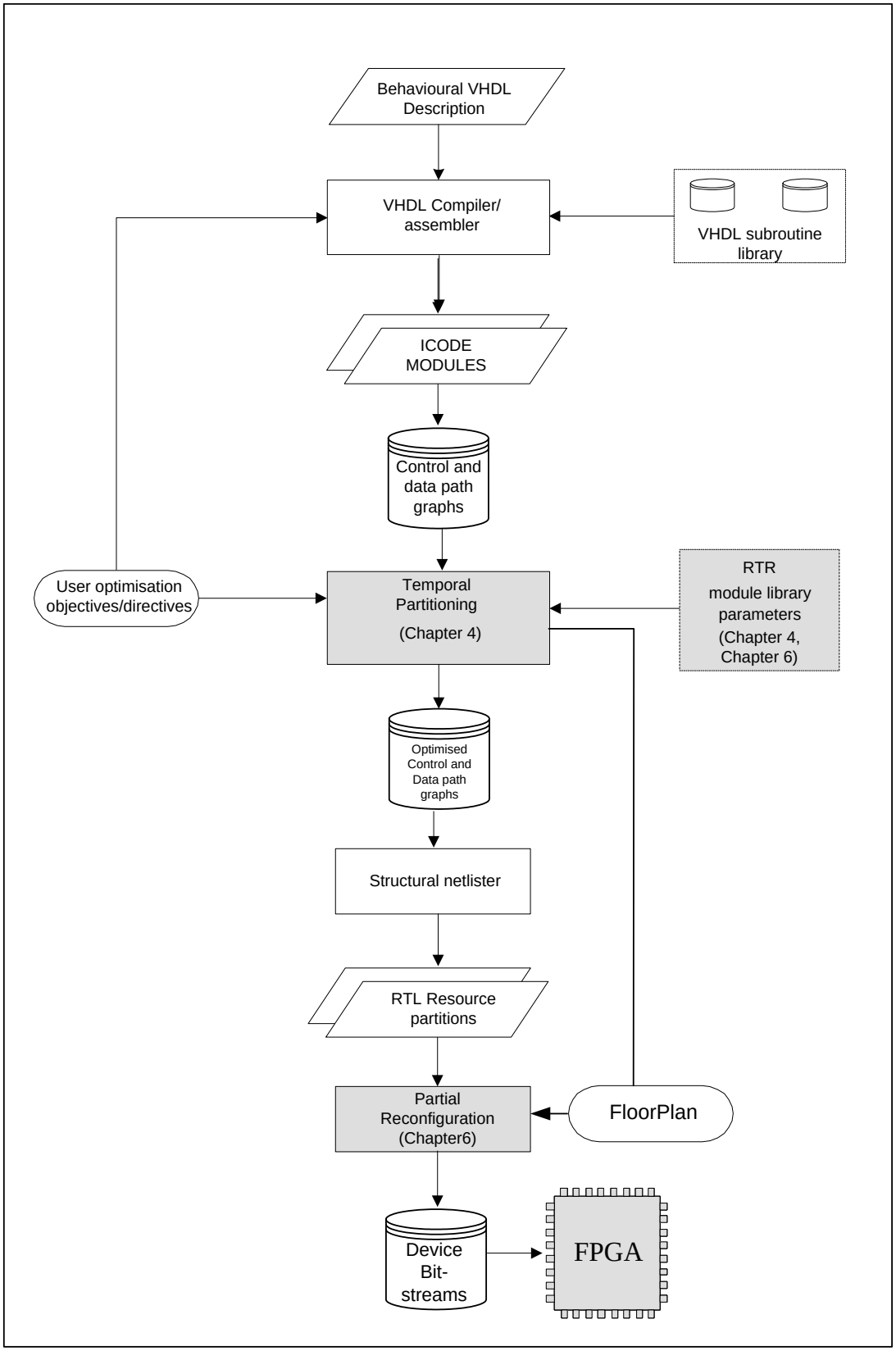


Figure 3.7: MOODS synthesis extended for temporal and spatial partitioning.

```
-- Behavioural partitioning of sequential and parallel VHDL constructs featured in a communication system
library ieee; use ieee.std_logic_1164.all; use work.bch.all;    -- encoder/decoder subprogram library
entity communicationSystem is
  port(messageIn: in std_logic_vector(6 downto 0); messageOut: out std_logic_vector(6 downto 0));
end entity;

architecture behaviour of communicationSystem is
  -- interprocess communication
 signal codeWord: std_logic_vector(14 downto 0); signal codeScheme: std_logic;   signal code_ready: std_logic:='0';  signal code_received: std_logic:='0';
-- compiler-defined parallelism during synthesis
```

```
-- user-defined parallelism
transmitter: process
   -- compiler-defined parallelism during synthesis
   begin
    variable scheme: std_logic; variable; variable encodedMessage: std_logic_vector(14 downto 0);
    variable messageFormed: std_logic_vector(6 downto 0);
   loop
      -- wait for the receiver to decode the last codeword
      while code_ready / = code_received loop
        wait for 10 ns;
      end loop;
      -- sequential (subprogram) Behavioural Partitioning
      if scheme='0' then
        -- encode message read from input port
        bchEncoder(15,11,16,19,messageFormed,encodedMessage);
      else
        bchEncoder(15,7,465,256,messageFormed,encodedMessage);
      end if;
      -- write codeWord and coding scheme to receiver process
      codeWord<=encodedMessage;   codeScheme<=scheme;
      -- transmit another codeWord to receiver process
      code_ready<=not code_ready;
      wait for 10 ns;
   end loop;
end process;
```

```
-- user-defined parallelism
receiver: process
     -- compiler-defined parallelism during synthesis
     begin
      variable scheme: std_logic; variable; variable encodedMessage: std_logic_vector(14 downto 0);
     loop
       -- wait for the transmitter process to generate a codeword
       while code_ready = code_received loop
         wait for 10 ns;
       end loop;
       -- read codeWord and coding scheme from transmitter process
       encodedMessage:=codeWord;    scheme:=codeScheme;
       -- sequential (subprogram) Behavioural Partitioning
       if scheme='0' then
         -- decode codeWord using encoder directed scheme
         ViterbiDecoder(15,11,16,19,encodedMessage,decodedMessage);
       else
         ViterbiDecoder(15,7,465,256,encodedMessage,decodedMessage);
       end if;
       -- write message to output port
       messageOut<=decodedMessage;
       -- request another codeword from transmitter process
       code_received<=not code_received;
       wait for 10 ns;
     end loop;
end process;
```

```
end architecture behaviour;
```

```
-- Library Subroutine
--MOODS Temporal_Partition
procedure bchEncoder (codeLength: in integer range 0 to 15; messageLength: in integer range 0 to 11;
                      numStates : in integer range 0 to 465; generator: in integer range 0 to 256;
                      messageEncoded : out range integer range);
begin …/...  end bchEncoder
```

```
-- Library Subroutine
-- MOODS Temporal_Partition
procedure viterbiDecoder (codeLength: in integer range 0 to 15; messageLength: in integer range 0 to 11;
                      numStates : in integer range 0 to 465; generator: in integer range 0 to 256;
                      messageEncoded : out range integer range);
begin …/...  end viterbiDecoder
```

Figure 3.8: Sequential and parallel VHDL amenable to behavioural partitioning.

The behavioural specification describes a rudimentary communication system which relies upon error correction to recover the number of corrupted messages sent through a ‘noisy’ channel. As illustrated in the figure, its VHDL description can be divided into two coarse-grain parallel transmitter and receiver ‘Processes’, each of which are further partitioned into sequential encoder and decoder library subroutines.

On close inspection of each process description, a number of salient points are exemplified which are essential to the approach taken to partitioning and synthesising at a higher level of abstraction. The first is the requirement for user-described synchronisation between the transmitter and receiver processes: as the reader will recall, MOODS does not rely upon implicit synchronisation of processes; nor does it restrict the scheduling of instruction-level operations by fixing specific cycles to update process signals. This is shown in each process description: synchronisation exchanges a message and coding level between the transmitter and receiver through a data-path ‘Semaphore’, in the form of the ‘data_ready’ and ‘data_received’ signals.

The two-phase or ‘toggle’ semaphore ensures an orderly encoding and decoding of the codewords: the receiver waits for a change in the state of the semaphore, signalling the availability of the codeword for decoding; similarly, the transmitter does not send another codeword until the receiver indicates its readiness to process it, through a change in state of the semaphore.

Another use of the data-path semaphore is to enable the processes which use them to be implemented in different clock domains [102]. Multiple-clock domains have particular relevance in the synthesised architecture described in Chapter 5, ensuring that the reconfiguration of an FPGA occurs at the maximum the device can achieve − decoupling the ‘Reconfiguration Controller’ from the clock domain of the user- specified design.

Continuing the examination of each process description, what immediately becomes apparent is that the behaviour of each is implemented through a corresponding ‘bchEncoder’ or ‘viterbiDecoder’ subroutine, the ‘interface headers’ of which are shown below the process concerned. In this way, the coding scheme can be varied by calling each subroutine with the parameters relevant to the BCH code used. How the coding scheme is decided is not shown here; however the reader is referred to Chapter 7, where details concerning the relationship

between coding scheme and channel error rate are described as part of the case-study for run-time reconfiguration.

The behavioural style of VHDL compiled by MOODS cannot synthesize a process as purely combinational logic; there is an implicit clock signal and time does pass within a process – the program statements are scheduled over at least one clock cycle. Unlike its RTL equivalent, the finite state machine inherent to a behavioural ‘process’ goes some way in mirroring the sequential nature of the software paradigm; the advantage from a hardware perspective is that the paradigm does not apply to independent operations, these are likely to be scheduled to execute in parallel. Leaving the scheduling to MOODS, the style of behavioural coding exemplified by the figure is reduced to requiring the user to de-construct a specification into a set of communicating sequential processes which when required could be coded to synchronise by the user.

The motivation behind separating the encoding and decoding processes as VHDL ‘Procedures’ is clear: a subset of the statements within a process may be related by a common purpose and warrant their separation through the use of VHDL ‘Procedures’ and ‘Functions’; in doing so, the user now imposes a behavioural partitioning [116] of the specification.

The advantage gained by implementing a part of the behaviour of each process as a subroutine is due to the convenience of being able to test it, place it in a library and then re-use it in the present or future project; subroutines simplify the coding of a specification for the user. That said, subroutines are unlikely to have an equivalent representation at a lower level of abstraction: one disadvantage in preserving them is where their execution does not occur on a given path through the circuit, unlike the subroutines shown in both the Transmitter and Receiver processes. Regardless of how successfully optimisation is applied to the corresponding circuit structure, by its very definition, an idle subroutine is likely to waste several operations; had the subroutines been in-lined, the increased scope for optimising their number across former subroutine boundaries would likely act to reduce the prevalence of idle data-path units.

At its simplest, a computer hardware equivalent of the communication system would execute only those subroutines on the path actually taken during the execution of the design. This

form of ‘Late-Binding’ of resources is not presently available to MOODS and as a consequence the software approach available in HLS is limited to designing the hardware.

Through the introduction of reconfigurable resources and the partitioning of a specification across subroutine boundaries, the software analogy is extended to include the late-binding of subroutines at the physical-level of an FPGA; crucially, the partitioning occurs during compile-time as part of the design space exploration.

There may be occasion when the late-binding of a subroutine is intrinsic to the design specification. The case-study of the reconfigurable Viterbi decoder described in Chapter 7 is one such example: through a late-binding of the decoder to a set of reconfigurable resources, circuits with a similar resource consumption but different performance characteristics are used to adapt to the noise conditions of a communications channel. At other times, the objective of partitioning might simply be to reduce the overall use of logic resources, without regard to how a specification is partitioned; as long as the behaviour is indistinguishable from one that is not!

In section 3.3.2, synthesis directives were described as the way the user of a HLS tool can experiment with architectural techniques; it therefore made sense to provide the user and compiler with a *‘--MOODS Temporal Partition’* directive. Placing the partition directive in the body of a procedure, allows the user to mark the subroutine as a potential candidate for temporal and spatial partitioning: there is no presumption of reconfiguration; that is either determined manually by the user applying the graph transformations at the command prompt or automatically during the optimisation stage; in either case, the request to partition the procedure is passed on through to the next stage of representation: the ICODE intermediate language.

3.4.2 ICODE Description

With reference to Figure 3.7 the directive issued to the Compiler will have resulted in an intermediate ICODE representation which preserves certain user-guided aspects: a subroutine execution hierarchy will be represented (absence of the *‘--MOODS inline’* directive) and the *‘--MOODS Temporal Partition’* is translated into an ICODE assembly directive of the same

name. In doing so, it ensures that the associated data-structures are marked for the actual partitioning by the ‘Temporal Binding’ transform in a later stage of synthesis. As in the compilation stage, there no specific architectural detail, it is a matter of representing the necessary relationships between instruction behaviour.

Recall from the MOODS design flow that the ICODE (Intermediate Code) representation of a circuit is the source for the construction of the initial control and data-path graphs. As such, it is used to describe the behaviour of the circuit at a level of detail which embodies the function, execution order and data connectivity among its constituent operations. In a way not dissimilar to that of a microprocessor assembly language, ICODE provides a target specification for any high level language that can be used to model hardware. The independence which results from the generation of circuit structure indirectly via the intermediate code and not from a specific high level language, provides a consistent interface to the MOODS core functions. This frees the user from describing the circuit behaviour using a language whose lexicon is at a higher level of abstraction e.g. VHDL.

Figure 3.9 depicts the ICODE description that would be generated by the VHDL compiler following a parse of the behavioural description of the communication system. All circuit behaviour in VHDL is ultimately encapsulated by a description of the ‘Architecture’; the ICODE equivalent of the top level architecture is the ‘Program Module’. In representing the behaviour of the architecture, it will also initiate the execution of other ICODE modules, each of which is a translation of a VHDL subroutine, as is the case for the ‘bchEncoder’ and ‘viterbiDecoder’ modules depicted.

The external interface of all ICODE Modules is specified in the parameter list, derived from the ‘Entity’ declaration of the VHDL port description. Depending upon the context in which they are used, VHDL signals and variables are translated into ICODE variables which may be ICODE ‘ports’ or ‘registers’; with reference to Figure 3.9, these can be clearly identified in the first few lines of each ICODE Module.

The syntax of any ICODE instruction is of the general form:

Label: Operation <Inputs>, <Outputs> <Activation list> e.g.

.L000013 PROTECT 1e-010 ACT L000002

```
-- ICODE module equivalent of the behavioural communciation system description.
PROGRAM communicationSystem messageIn, messageOut

INPORT messageIn [6:0] OUTPORT messageOut [6:0] REGISTER codeWord [14:0] REGISTER codeScheme [0:0] REGISTER messageFormed [10:0] REGISTER scheme_0 [0:0] REGISTER scheme_1 [0:0]
REGISTER messageEncoded_0 [10:0] REGISTER messageEncoded_1 [10:0] REGISTER code_ready [0:0] INIT #%0 REGISTER code_received [0:0] INIT #%0 REGISTER tmp0 [0:0] REGISTER tmp1 [0:0]
REGISTER tmp2 [0:0] REGISTER tmp3 [0:0]

.L000001 NOOP ACTT L000002 ACTF L000014

.L000002 uneq code_ready, code_received tmp0
.L000003 IF tmp0 ACCT L000004 ACTF L000005
.L000004 PROTECT 1e-010 ACT L000003
.L000005 uneq scheme_0, #%0, tmp1
.L000006 IF tmp1 ACCT L000007 ACTF L000008
.L000007 MODULEAP bchEncoder #%1111,#%1011,#%10000,%#10011,messageFormed,
         encodedMessage ACT L000009
.L000008 MODULEAP bchEncoder #%1111,#%0111,#%111010001,%#100000000,messageFormed,
         encodedMessage
.L000009 MOVE encodedMessage_0, codeWord
.L000010 MOVE scheme_0, codeScheme
.L000011 unot code_ready, tmp2
.L000012 MOVE tmp2, code_ready
.L000013 PROTECT 1e-010 ACT L000002

.L000014 ueq code_ready, code_received tmp3
.L000015 IF tmp3 ACCT L000016 ACTF L000017
.L000016 PROTECT 1e-010 ACT L000014
.L000017 MOVE codeWord, messageEncoded_1
.L000018 MOVE codeScheme, scheme_1
.L000019 ueq scheme_1 #%0, tmp4
.L000020 IF tmp4 ACTT L000021 ACTF L000022
.L000021 MODULEAP viterbiDecoder #%1111,#%1011,#%10000,#%10011,messageEncoded_1,
         messageDecoded ACT L000023
.L000022 MODULEAP viterbiDecoder #%1111,#%1011,#%10000,#%10011,messageEncoded_1,
         messageDecoded
.L000023 MOVE messageDecoded, messageOut
.L000024 unot code_received, tmp5
.L000025 unot tmp5, code_received
.L000026 PROTECT 1e-010 ACT L000014

.L000027 ENDMODULE
end architecture behaviour;
```

```
MODULE bchEncoder codeLength, messageLength, numStates, generator, message, encodedMessage

TEMPORAL_PARTITION

INPORT codeLength [3:0] OUTPORT encodedMessage [14:0] REGISTER tmp0 REGISTER tmp8

.L000027 MOVE codeLength, tmp0
  …/...
.L000099 MOVE tmp8,encodedMessage
.L000100 ENDMODULE
```

```
MODULE viterbiDecoder codeLength, messageLength, numStates, generator, message, decodedMessage

TEMPORAL_PARTITION

INPORT codeLength [3:0] OUTPORT encodedMessage [14:0] REGISTER tmp0 REGISTER tmp10

.L000101 MOVE codeLength, tmp0
  …/...
.L000199 MOVE tmp10,decodedMessage
.L000200 ENDMODULE
```

Figure 3.9: ICODE Module encapsulation of parallel and sequential VHDL constructs.

In conjunction with the activation list, the labelling of each instruction explicitly states the order of its execution, in the context of the other instructions. The sequencing of the instructions is used at the next stage of abstraction to construct an initial control and data-path graph to which several types of scheduling, allocation and binding transformations are applied. The reader is referred to Appendix A, where the relationship between the ICODE instructions and the graph representation is described in greater detail.

In the context of temporal partitioning, the relevant aspects of ICODE are to be found in its representation of the parallel VHDL processes and the hierarchical nature of subroutine execution.

On inspection of the 'Program' module in Figure 3.9, the ICODE instructions are divided into two groups: each group is the ICODE equivalent of the 'Transmitter' and 'Receiver' processes described in the VHDL specification. By describing the circuits through two individual processes, the resulting ICODE is also expected to execute concurrently. In practice, it is realised by the first instruction (*'NOOP'* − instruction .L000002) and has no other purpose than activating the first instruction of the ICODE sequences translated for a given process.

The last instruction of each equivalent ICODE process (instructions .L000013 and .L000026) returns the execution sequence to the beginning of the corresponding process, without either processes converging; this is as a direct result of using VHDL coarse-grain parallelism: processes never terminate and therefore never converge.

ICODE and the MOODS internal data-structures do provide support for fine-grain parallelism through the use of a 'collect' instruction. As its name suggests, it may be used to 'collect' any number of fine-grain threads and in doing so enable their convergence. The relevance to temporal partitioning is through the employment of fine-grain parallelism in modelling partial reconfiguration: overlapping the segments of each coarse-grain (user-declared process) with a fine-grain thread enables the cost function routines in MOODS to incorporate partial reconfiguration delays as part of its data-path delay metric. Furthermore, the cost function is able to compare the execution delay of a sequence of operations with the parallel delay due to reconfiguration; the larger of the two will define the delay of the parallel paths prior to their convergence. During optimisation, the scheduling transforms attempt to hide a

reconfiguration delay by determining which segment of a user-defined thread is capable of dominating the delay and schedule a partial reconfiguration accordingly.

With reference to Figure 3.9, the execution of the ‘bchEncoder’ and ‘viterbiDecoder’ modules is achieved through the ICODE ‘MODULEAP’ instructions (.L000007/8, .L000021/22) from within the program module. Unlike the other instructions, the absence of an activation list is not interpreted by MOODS as an activation of the next sequential instruction but as a call to the module named by the instruction. The module responds by activating its first instruction, which is to read the first ‘INPORT’ parameter; in this way, control is passed from the calling module to the called.

Input arguments to sub-modules are passed by reference, that is, no intermediate variable is used – the arguments are inserted directly into the ‘MODULEAP’ instruction. As no other subroutine is permitted parallel execution in the same thread, the instruction acts to halt its execution until an ‘ENDMODULE’ instruction is encountered in the called module; it is only reached once the result of the module’s activation is written to the ‘OUTPORT’, in effect a direct writing of the result to the associated register. In this way, not only can the ICODE represent the calling of a VHDL procedure or function within the body of the architecture, it can also describe a hierarchy of nested ICODE module calls – which occur when a procedure or function calls another.

As described in the Appendix, the ICODE instructions are used by the Technology Library to attribute delay and area properties to the control and data-path graphs representing the circuit structure. Although temporal partitioning is not carried out during the ICODE stage, it is the next stage of abstraction. To achieve this, the ICODE database was updated with new instructions and an external data-type: ‘resourceWrite’, ‘resourceRead’, ‘resourceSync’ and ‘ext_var’, respectively.

The ‘resourceWrite/Read’ instructions encapsulate the partial reconfiguration of the FPGA resources at the device level. This typically requires several thousands of device configuration cycles and is likely to occur in a different clock domain to the user’s design. Synchronisation between a user process and the configuration of the device is provided by ‘resourceSync’ instruction. The configuration data is stored externally to the FPGA in RAM or ROM

depending upon the method used to save state between context switches of a reconfigurable resource.

The ‘ext_var’ type ensures that the configuration memory is automatically represented at the graph and RTL level, where it is accessed through the port of the synthesised design during device reconfiguration. As required by the cost function during circuit characterisation, all the partitioning-related instructions have equivalent cells to characterise them in the graphs; in the next stage of circuit representation they will be used to influence how the partitioning is carried out.

3.4.3 Circuit Optimisation

The motivation for using a high-level synthesis tool like MOODS is the automated means in which it can examine the consequences of a multitude of different ways to schedule and implement the instructions/operators that constitute a circuit description. When constraints are placed upon the search, such as the number of resources available or a minimum length of the critical path, the scheduling, allocation and binding of the instructions becomes an optimisation problem.

Unfortunately, these three main tasks of High-Level Synthesis, like many other VLSI CAD problems (partitioning, floorplanning, circuit placement and routing) do not have to date specific algorithms to find their optimal solution in polynomial-time (NP-complete). Until the day arrives when an optimal solution is found (if it can ever be), there exist many heuristic or approximate methods for generating a good solution for a given optimisation run, none of which are guaranteed to be optimal.

Before high-level synthesis techniques are applied, there must be some way of quantifying the characteristics of a given circuit structure. This is achieved through a number of circuit ‘metrics’, principally but not limited to: circuit ‘area’ (e.g. the number of FPGA logic blocks e.g. Xilinx LUTS [6]), critical path ‘delay’ (ns) and ‘clock period’ (ns). In MOODS, these metrics are the primary means of measuring whether or not optimisation is transforming the circuit structure closer to the user’s requirements.

Unless a circuit is optimised for one objective, which it seldom is, each individual metric provides a one dimensional measure of the quality of a circuit's structure. The difficulty in circuit synthesis and optimisation in general, is that an improvement in one metric is usually to the detriment of another. A simple example describes a classic area versus delay 'trade-off' in circuit design: The inequality instructions 'uneq' (.L000002 and .L000005) could be allocated to share the same data-path unit provided they are scheduled to execute during separate control states. This would represent a minimal area for the two instructions. Alternatively, to achieve a minimal number of control states, the instructions could be scheduled to execute during the same state. Hence there is a trade-off between reducing the area of a circuit through sharing the functional units, at the expense of multiple control states and ultimately a greater circuit delay.

A cost function is used by an optimisation algorithm to quantify just such a multi-dimensional trade-off. Moreover, it can be used to compare metrics which are often conflicting and return a single figure that provides a net measure of the 'quality' of a given circuit configuration.

The cost function in MOODS takes the form:

$$cost(circuit) = c_1 x\ area + c_2 x\ delay + c_3 x\ clock\ period$$

Where:

- $area, delay, clock\ period$ are the circuit metrics to be optimised,
- c_1, c_2, c_3 are weighted constants (e.g. 1(high), 2(low)) which direct the optimisation priority of their associated metric.

Through the cost function, the user can specify the metrics to be optimised and the order in which this takes place. As one would expect, the optimisation of a high priority metric takes precedence over that of all lower priority ones. When they are of equal priority, the cost function returns an average of the metrics.

Of course, a measure of quality is somewhat meaningless without a frame of reference. That is provided for in MOODS, through the use of a user defined target value which is associated with each of the metrics and a memory of the quality of the structural configuration prior to applying the optimisation method. Combining these concepts, each metric *m* has the following attributes:

- m_{target}: is the user specified goal for the metric following optimisation.
- $m_{initial}$: reflects the value of the metric prior to any form of optimisation.
- $m_{present}$: gives the updated value of the metric upon acceptance of an optimisation move.
- $m_{estimate}$: returns an approximation of the metric should the move be applied.

Whenever optimisation is applied to the circuit, an improvement or degradation to each metric is calculated as a change in value, normalised over the initial value. Each metric may have a different unit of measure and so normalisation enables the cost function to compare each change in metric with the next, producing one figure representative of the net effect when a particular form of optimisation applied. This figure may be referred to as a measure of the "energy" exhibited by the circuit structure. It is not a measure of actual energy, rather it owes its origin to the Metropolis criterion [117] and algorithm of the same name, used to model the *energy* changes required for the equilibrium of molecules at a given temperature. It will be described in greater detail in due course, as it forms the basis for the Simulated Annealing [7] algorithm, one of the algorithms employed by MOODS to perform optimisation.

Using the properties associated with each metric, a change in energy ΔE for the metric m is formulated as:

$$\Delta E_m = \frac{m_{estimate} - m_{present}}{m_{initial}} \qquad (3.0)$$

Therefore, at any point during optimisation, the quality of the circuit structure 'S' is given by the summation of the energy changes to each of the metrics:

$$Cost(S) = \Delta E_{area} + \Delta E_{delay} + \Delta E_{clock\ period} \qquad (3.1)$$

During optimisation of the circuit, any improvement which brings the structure one step closer to the user's objectives is expressed through the cost function, in the form of a negative value. Complementary to that, a positive value denotes its degradation. A value of zero occurs when the user's target value for the current level of priority has been reached i.e. $(m_{present} \text{and } m_{estimation}) \leq m_{target}$, at which point optimisation proceeds with the next level of priority, until this situation arises again and is responded to in the same manner or all objectives have been met.

The delay and clock period metrics are a function of the delay characteristics for each of the data-path units referenced within an ICODE instruction. During their scheduling, multiple instructions are executed consecutively or concurrently within a single control state. This requirement greatly influenced the way the infrastructure to facilitate reconfiguration and communication between partitions was implemented: each new data-path unit was combinational. For example, the resourceWrite instructions are repeatedly executed for each configuration cycle at the device-level, as opposed to implementing them as a finite-state machine outside of the MOODS core. The reader is referred to Chapter 5 for further details of the device-level infrastructure. In the absence of a user-specified constraint, the clock period is governed by the greatest chain or individual instruction delay of any control node. When specified, it is used in place of the maximum control node delay, as the scheduling is always sensitive to the limit set by the user and is never permitted to exceed it.

The controller is likely to have multiple control paths, any of which may be taken during its execution. Without knowledge of the likelihood of a given path being taken, it must be assumed that all have an equal chance of being followed. The Critical Path is the longest of these paths, so called because without being able to predict when it might be taken, it represents a worst case measure of the most number of clock cycles that could be taken during the course of the controller's execution; it is therefore a principal target for optimisation, with the aim of minimising the number of cycles required to traverse it. To be accurate, the control graph must incorporate the multiple iterations of any bounded loops which lie on its path and be constantly compared to others during optimisation, to ensure it remains designated as critical. When multiplied by the clock period (the maximum control node delay), it provides a metric for the overall circuit 'delay'.

Despite the disadvantage described above, the fact that the full behavioural specification is known at compile-time is what differentiates reconfigurable logic from a partial-specification associated with reconfigurable computing. The placement implied in resource binding can be determined as part of the design-space exploration, where there is considerably more time and computational resources to explore it than would be available to a completely run-time approach.

3.4.4 Optimisation Algorithm

Before briefly reviewing the optimisation methods available in MOODS, it is worthwhile considering the context in which they are applied. For a given design, there maybe any number of different structures that can be used to realise a given circuit behaviour, each of which maybe functionally equivalent but have different area and delay characteristics. Unlike RTL synthesis, a design specification for a behavioural synthesis tool constrains the structure of the design as little as possible. This enables the synthesis tool to find a structure that best meets the design constraints.

A structure expressed in terms of area and delay characteristics forms the coordinates of a point in the design space of alternative structures for a given behaviour. The design space is an *n*-dimensional space, where *n* is the number of different aspects of the design specified by the designer. Figure 3.10 illustrates a 2-dimensional design space, in terms of area and time.

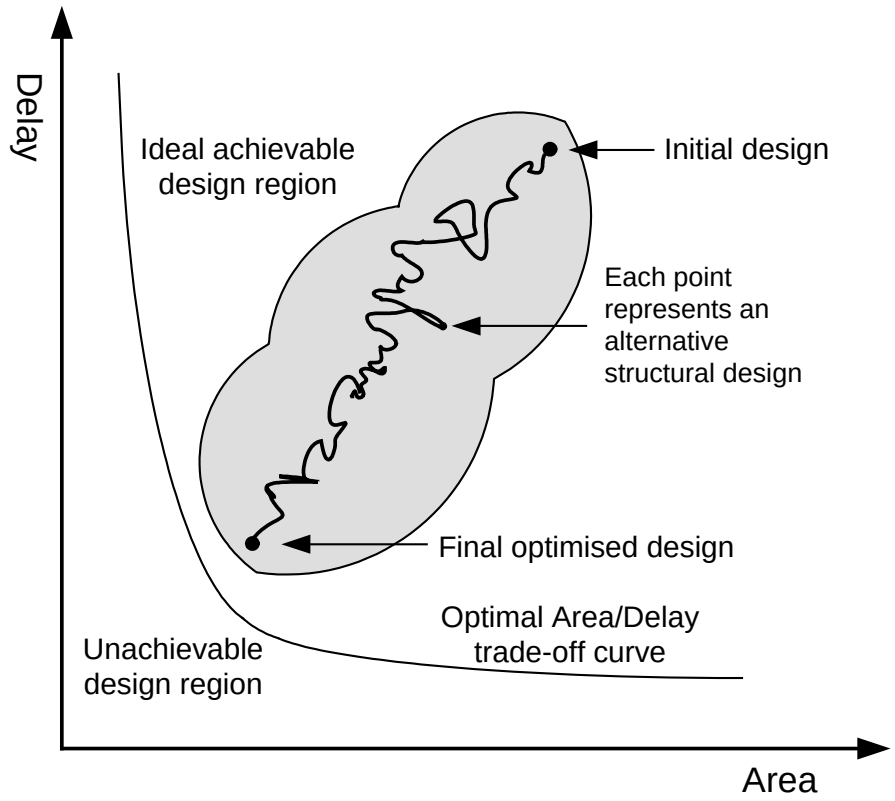


Figure 3.10: A 2-dimensional (area/time) design space.

The cost function is used to quantify the absolute state of the design within the design space. An optimisation algorithm, such as Simulated Annealing [7], uses the cost function to move the design through this space, from the initial behavioural specification toward an optimal implementation which meets the designer's area and delay objectives. If, for each dimension in the design space, no other point exists with a better value, a design is considered optimal and lies on the optimal area/time trade-off curve. The curve separates the space into a set of achievable and unachievable implementations. In reality, the actual achievable region reflects a proportion of the points in the achievable region that may be obtained.

Knowing exactly which transforms should be applied and in what order is beyond the scope of this thesis, like other CAD problems, scheduling and allocation are NP-Complete problems. However, MOODS in common with other solutions to CAD problems adopts a number of heuristic approaches namely Simulated Annealing and an ad-hoc (quasi-exhaustive) method based upon the experience gained from multiple experimentation runs of the annealing algorithm.

Unlike the approach of many of the traditional synthesis tools, where scheduling and allocating are constructive, MOODS implements an iterative approach to circuit optimisation. In the constructive approach, at any point during the execution of the heuristic, the design is always a partial one – it is literally being constructed. In contrast, an iterative approach starts with a circuit structure which is fully scheduled, allocated and bound and proceeds to modify it in order to meet the cost function. MOODS achieves iterative optimisation of a design through multiple repetitions of the optimisation loop depicted in Figure 3.11.

Transforms are applied to modify the data structure, typically merging control steps in the control graph and sharing functional units in the data path. The application of the scheduling and allocation transforms is local, in the sense that they affect small portions of the control graph and data path. Being independent and leaving the design valid before and after their application, allows the optimisation algorithm to apply the transforms in an order that will move the design from an initial naïve implementation, with one control state per instruction, one register per variable, one functional unit per operation, to an optimised implementation which is as close as possible to the user objectives.

A transform is selected and a portion of the design to which it is applied is determined by the transform and data selection phase. This is dependent upon the optimisation algorithm itself.

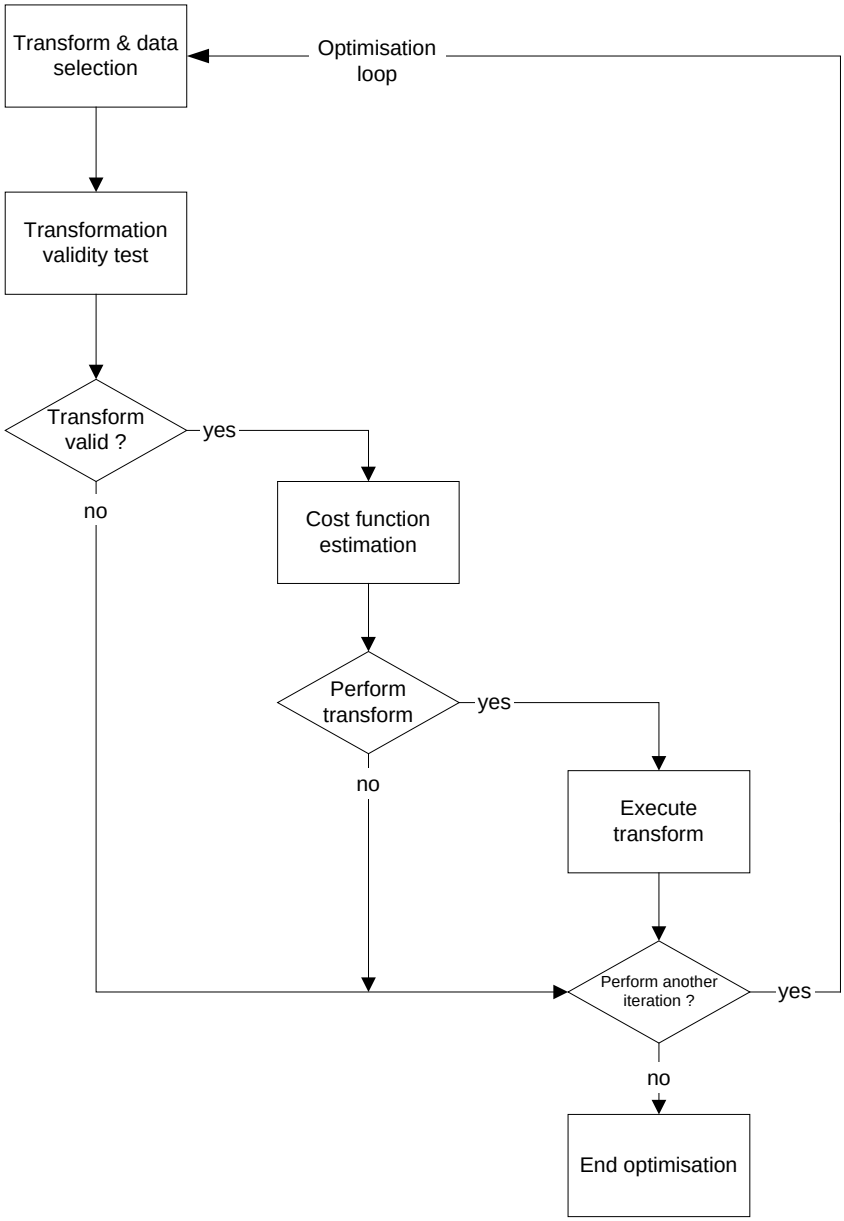


Figure 3.11: MOODS iterative improvement optimisation loop.

In Simulated Annealing, a random transform is applied to a random portion of the design, however the quasi-exhaustive heuristic algorithm applies the transforms in a fixed order to every part of the design. During the transform validity *test* stage, decisions are made as to

whether or not a transform can be applied. This ensures that the qualifying tests of the transforms are met, such as mutual exclusivity and instruction dependency.

The transform must leave the behaviour of the design unaltered. A successful validity check leads to the cost function estimation phase, which simulates the changes made to the design by the transform and the cost function is used to quantify the impact of the transform on the area and delay metrics. A decision is made to accept or reject the transform based upon the average energy change ΔE, in terms of the cost function. The average energy change provides a means of determining whether the optimisation of a design is being guided towards or away from its objectives. An acceptance of the transform results in the changes being applied to the data structure during the execute transform phase.

3.4.5 Simulated Annealing

The objective during the annealing of solids is to create a highly crystalline structure through an ‘annealing’ process. In the early stage of this process, the material being annealed is heated to a temperature at which the molecules gain sufficient energy to move around, having literally broken the chemical bonds that previously fixed their structure. Controlling the rate at which the material is cooled gradually restricts the movement of the molecules. This cooling schedule slowly transforms the material from a high energy liquid state to one of minimal energy, with the molecules taking the form of a crystal lattice.

Simulated Annealing seeks to mimic this process as a general optimisation method, where achieving a global optimum is analogous to obtaining a good crystal structure. The correspondence between the physical process and the optimisation algorithm is as follows: firstly, a particular configuration of the circuit structure (quantified by the cost function) is analogous to the energy state of the material being annealed. Secondly, the movement of the molecules as the material is being heated and cooled is simulated in MOODS through the random selection of the graph transforms and the data and control nodes to apply them to. The cooling schedule is modelled in the algorithm by requiring the user to specify the start and end temperatures, as well as the number of transform selections to apply at each temperature. The increasing restriction of molecular movement with respect to temperature is simulated in

the way that each transform is accepted or rejected. At the centre of this decision is the Metropolis Criterion [117].

Transforms which improve the cost function are unconditionally accepted, where as degrading transforms are accepted on a probabilistic basis with respect to the annealing temperature. More specifically, the probability *P* of allowing an inferior solution is given by:

$$P = e^{\frac{-\Delta E}{T}} \text{ when: } \Delta E > 0 \qquad (3.5)$$

Where:

ΔE is the estimated change in energy resulting from the transformation,

T is the annealing temperature.

A uniform random number between 0 and 1 is chosen and if it is below the probability threshold *P,* the degrading transform in question is accepted – otherwise it is always rejected. In this way, the algorithms mimics the freedom experienced by the molecules at higher temperatures, the accepting and degrading transforms are both likely to be accepted. This approach enables the algorithm to explore the design space more freely at higher temperatures, moving more frequently between adjacent minima in the landscape of the design space.

As the temperature cools, the probability of accepting a degrading transform becomes smaller and it becomes gradually more difficult to exit the local minima. Upon reaching the end temperature, the configuration space is frozen, analogous to the physical process reaching thermodynamic equilibrium and hopefully a global minimum has been found.

The advantage of simulated annealing is its ability to find global minima without requiring knowledge of the trade-off mechanisms involved, since the process is reliant upon the cost function and the transform estimators to encapsulate the design space. The application of many unsuccessful transforms will result in an increase in the optimisation time. It is a difficult task to determine what the annealing schedule should be and therefore whether the chosen schedule is a good one.

From the user's point of view, employing simulated annealing requires a trade-off between the quality of the structural solution and the time taken to achieve it. If optimisation is only

required of the area and delay circuit characteristics, the remaining pseudo-exhaustive heuristics are faster at reaching a solution, albeit one that may not be as optimal.

3.4.6 Structural Circuit Abstraction

As shown in Figure 3.7, the last phase of MOODS behavioural synthesis occurs when the internal representation is converted into a structural VHDL description, suitable for further logic optimisation and synthesis by third party tools. To achieve this, the circuit must be converted from the internal representation embodied by the data structures, to one conforming to a ‘Structural’- style of VHDL utilising component instantiations.

The first step towards achieving this goal is to insert the multiplexors in the data-path. To date their existence would have only been implied. This is due to the inefficiency that would result should the multiplexors be frequently added or removed during the course of optimisation. The data structures linking the selection of the multiplexor inputs to the relevant ICODE instructions must also be updated once they are added to the data-path. Additionally, boolean equations are generated for all control signals, such as those used in selecting the multiplexors inputs.

Although the ICODE instructions are invaluable in relating behaviour to structure in the data structures, they would serve no purpose in the final description of the circuit structure. Instead, they guide the generation of control signals that link the data-path units to those control nodes in which they are scheduled to execute.

During the final modification to the data structures, bypassed data-path units such as registers are removed during a general tidying of the data-path. Once this is completed, the data structures can be consulted in order to generate a one-hot finite stage machine controller and data-paths using the RTL VHDL descriptions of their appropriate cells.

With regard to temporal partitioning, fine-grain parallelism was allowed to occur during the optimisation stage from within user-defined process threads, enabling reconfiguration to be scheduled alongside their execution. Modelling reconfiguration in this way was solely to quantify the area-reconfiguration trade-off inherent to temporal partitioning. The splitting of a

single user-thread is not modelled in VHDL, therefore all fine-grain parallelism must be converted to coarse-grain VHDL processes.

In practice, this is achieved at the structural-level of abstraction using data-path semaphores: each point of thread divergence or convergence is replaced by a data-path semaphore. All ‘ContextSwitch’ instructions which previously represented the reconfiguration delays along the fine-grain threads are subsequently mapped to a single coarse-grain VHDL process; as a result, a VHDL compliant ‘Reconfiguration Controller’ is generated, further details regarding the device-level infrastructure are provided in Chapter 5.

3.5 Summary

The theme of this chapter has been how a circuit specification may be represented at different levels of abstraction. In Section 3.1, the contrast between a traditional RTL and a HLS approach was exemplified through a BCH encoder example. It showed how different architectural solutions can be found by experimenting solely with synthesis directives and hardware constraints. The section also described the renewed interest in HLS, in part motivated by the accessibility of FPGAs and the familiarity of software languages.

In Section 3.2, the reader was introduced to a general behavioural synthesis flow featuring MOODS; it detailed the typical stages of a general flow: from an algorithmic circuit specification to device-level implementation; the reader will have appreciated the advantage of synthesising digital circuits at an architecture-free level. Section 3.3 contrasts the approach taken to optimisation in MOODS with several commercial and academic tools.

In the last section of the chapter, the theme of abstraction was re-visited by considering how it is used in MOODS to implement a key aspect of the run-time reconfigurable approach: the partitioning of a design at a high level of abstraction for eventual placement using low-level reconfigurable resources.

Chapter 4

Spatial and Temporal Resource Binding

In earlier chapters, the sharing of circuit resources was described through high-level synthesis or temporal partitioning and placement techniques. This chapter describes how these approaches to resource sharing can be combined in MOODS HLS, enabling it to explore the extended design space formed from the use of reconfigurable resources during architectural synthesis.

4.1 Resource Binding

The partitioning techniques reviewed in Chapter 2, describe how advantageous it can be to apply temporal partitioning at a higher-level of abstraction in the design flow; particularly during high-level synthesis (HLS), where partitioning can benefit from the opportunity to influence how the circuit architecture is determined. For example, partitioning can occur at a stage closer to the technology level, such as at the gate-level [36]. However, at this stage in the design flow, there is no opportunity to decrease the lower bound on the number of parallel nets that would be cut by the partitioning; a number whose size would have been determined by the way the functional units connected by the nets were scheduled to execute during HLS.

Having decided that temporal partitioning should occur as part of HLS, determining the form it should take was influenced by the iterative approach in which MOODS performs circuit optimisation; that is to say, a tentative scheduling, allocation and cell binding of the control and data-paths components exists prior to the application of an optimisation heuristic. Other popular approaches to temporal partitioning, such as List Scheduling [79], are constructive in nature and until the final stage of their application must rely upon an incomplete measure of how the circuit resources are used at each stage of the algorithm’s execution.

To operate within the existing iterative approach to optimisation, all control and data-path components employed in the execution of a single instruction are initially bound to a single reconfigurable resource. In this way, the partitioning of the circuit proceeds from an upper bound on the number of reconfigurable resources – a size that is unlikely to satisfy any resource constraint, to one that does. In order to achieve such a goal, any resource constraint must be expressed in terms of the logic capacity of a physical device; as should the characteristics of each reconfigurable resource.

Customising the physical attributes of a reconfigurable resource is made possible through its binding to one or more technology cells. The reader will recall the purpose of Technology Binding in HLS: to physically characterise the control and data-path nodes in a way that can be quantified through cost function metrics; it also conveniently provides a physical context for incorporating the temporal partitioning of reconfigurable resources into an existing HLS tool, such as MOODS.

During technology binding, each control or data-path node is bound to a cell – a parameterised model of the resources needed to realise each allocated instruction in the chosen technology. For example, in MOODS HLS, it can provide an estimation of the size of the resource usage for a data-path node by using the bit-widths of ICODE instruction variables as a parameter to the cell model.

Several alternative library cells might exist for a different physical implementation of the same instruction behaviour; a cell featuring greater parallelism in its structure will provide a reduction in latency when executed, but at the expense of using more resources. This trade-off between execution delay and resource area can be explored by changing the binding of a node, enumerating the effect of implementing an instruction with faster or smaller circuits.

Figure 4.1 illustrates this point: during HLS, data-path units (dp_2-dp_5) are allocated identical instructions (i_2-i_5), each of which is scheduled to execute during a particular control step (s_2-s_5) and in the instruction order shown in the control graph; where applicable, the data-dependency between a pair of data-path units (dp_x,dp_y) is labelled $d_{x,y}$ accordingly.

In order to implement the behaviour assigned to it during instruction allocation, every data-path unit is bound and labelled (b_n) to a selected technology cell ($cell_P$ or $cell_S$). Ultimately,

each technology cell will be replaced by the actual technology specific implementation, shown in the figure as a fixed or static binding (sb_n) of a technology cell to a physical resource (r_2-r_5).

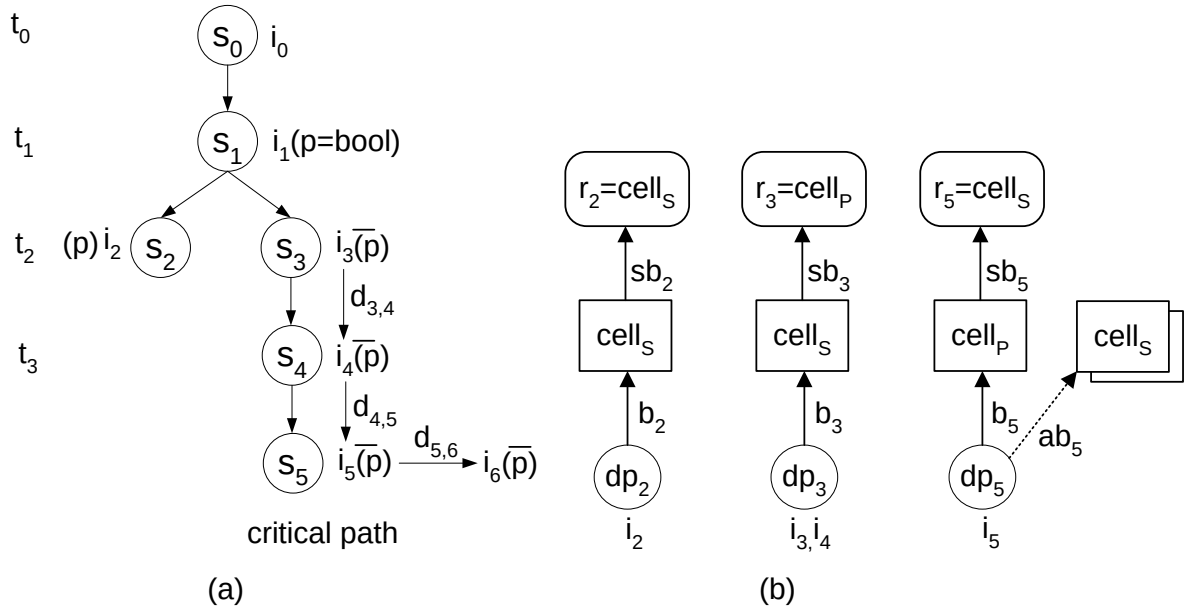


Figure 4.1: Static resource binding in high-level synthesis.

The flexible nature of cell binding is limited to the hardware compilation stage only and for clarity, is represented in the figure by the alternative binding (ab_5) of a single data-path unit to a different cell in the technology library. Once the process of technology binding is complete, the link between the behaviour of an operation and how it is implemented is fixed by the technology binding. Similarly, the choices taken during resource binding will result in a one-to-one mapping between how and where an operation is implemented by a technology vendor's placement and routing tools; choices which by their nature are time-consuming and as such are also taken during the hardware compilation stage of the design flow.

Of all the instructions scheduled in control graph, instructions (i_2- i_5) are of particular interest: although they are assumed to carry out the same operation, how that is achieved through cell binding is influenced by the path on which they are executed; that decision would be taken at run-time and is dependent on the value of the predicate p, as determined by the execution of instruction i_1 during the earlier control step. We assume that control step s_5 has the greatest

combinational delay ($d_{5,6}$) of all the control steps in the schedule and consequently it determines the length of the clock period.

Of further interest is the scheduling of instructions i_3-i_6 due to their execution occurring on the critical path, that is to say the instructions cannot be scheduled earlier or later without violating data dependencies or without lengthening the path. In such circumstances, no scheduling mobility would make cell binding the only means of reducing the path delay. This would explain the choice of the parallel version of the cell ($cell_P$) during binding: the result of an optimisation attempt to reduce the latency of the critical path by reducing the control step with the longest delay, at a cost associated with an increase in the number of resources required by the parallel cell.

An alternative course of action for the scheduler to take would have been to reduce the length of the critical path by reducing the number of control steps. Scheduling instructions i_3 and i_4 to the same control step is not possible because they are allocated to the same data-path unit (dp_3). Scheduling instructions i_5 and i_6 earlier would break the order of their data dependencies ($d_{4,5}$ and $d_{5,6}$); scheduling all three instructions to the same control step would at the least double its delay (i_4 and i_5 are the same type) to a value we have assumed to be greater than the combinational delay of control step s_5.

The binding of a serial version of the cell ($cell_S$) to a data-path unit (dp_2) not executed on the critical path might have been the enabling factor for the binding of the parallel cell: additional slack in the delay of the non-critical path would favour the binding of the slower cell; its more modest resource requirements would permit the increase in parallel resources associated with the faster cell binding whilst meeting the resource constraint of the cost function.

With regard to meeting a resource constraint, an obvious approach is to reduce the number of data-path units. The availability of instructions of the same type, for example, instructions i_3 and i_4, is often exploited by their allocation to a single data-path unit with a common cell binding. Mutually exclusive instructions of the same type, such as instructions i_2 and i_3, are also able to be allocated a single unit; they offer an additional advantage of not prohibiting instruction level parallelism, as would be the case were the instructions scheduled on the same path. The inter-dependence between optimisation tasks in HLS, such as instruction allocation or cell binding makes their application sensitive to the order in which they are applied:

without a cell capable of implementing both parallel and serial characteristics, MOODS would be unable to share their data-path units.

Multi-mode cells [118] are the solution to the problem described because as their name suggests, they allow a common cell binding between data-path units which are allocated instructions of different types; ALU substitution for separate addition or subtraction cells being a classic example.

An obvious pre-requisite to the deployment of a multi-mode cell is that its individual size is smaller than the sum of the separate functions it provides. The sharing of common sub-structures not only reduces the resource count in terms of the number of data-path units, but can provide an opportunity for an increase in the parallelism of their execution [119]. Figure 4.2 illustrates this point by replacing cell ($cell_P$) from Figure 4.1) with a multi-mode $cell_m$ which is able to implement the serial cells ($cell_S$) in a second mode of operation.

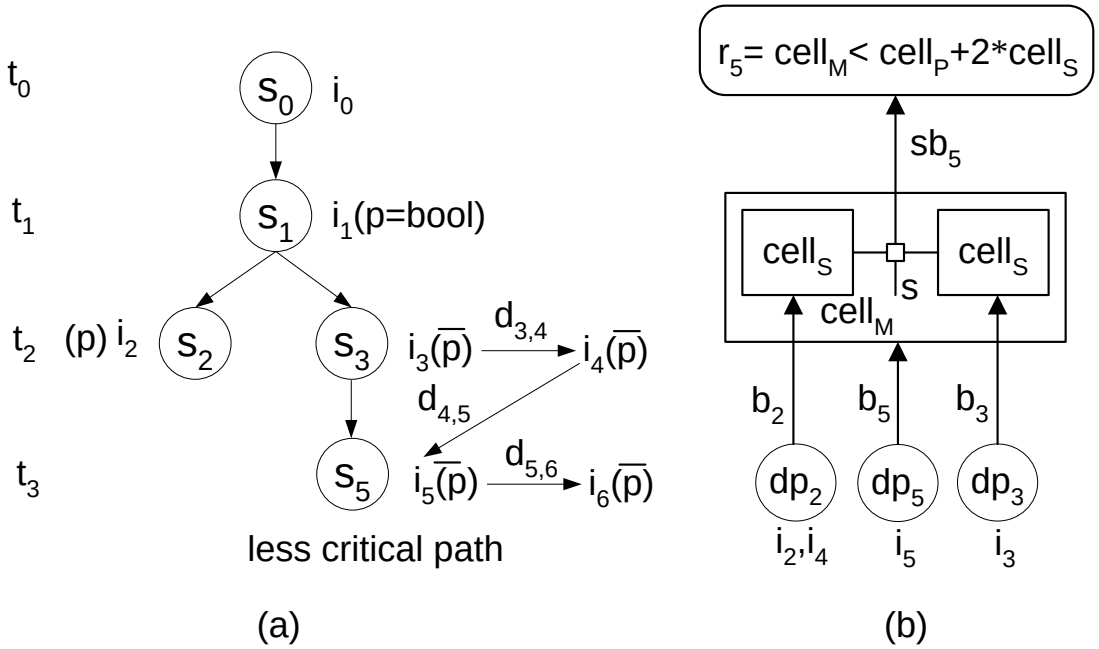


Figure 4.2: Resource reduction through static binding of multi-mode cells in HLS.

As the internals of the cell shows, parallelism of the faster cell $cell_P$ is achieved through a replication of the slower serial $cell_S$. Selection between the twin cells is under the control of a logic switch S; when active, it allows data-flow between the two serial cells and in doing so

implements the binding of an equivalent parallel cell. When inactive, the switch enables each cell to execute independently of the other and crucially in parallel to the other.

Figure 4.2a illustrates how this additional parallelism has been put to good use by the scheduler: the availability of an extra serial cell enabled it to schedule instruction i_4 to a state earlier than the previous schedule, thereby removing state s_4 and subsequently reducing the length of the critical path.

When viewed from the perspective of run-time reconfiguration, the multi-modal cell exhibits the essential property of resource re-use, albeit in a spatial form; in fact, as the reader may recall, the trade-off between re-use and re-configuration is the modus operandi of RTR.

A combination of a structured and hierarchical approach to cell design, coupled with its use for physical characterisation in HLS makes cell binding a viable choice for temporal partitioning. Vasilko et al. [120] used it to parameterise the bit-width for the same type of functional units in different temporal partitions, relying upon cells which were pre-placed and routed in the target technology prior to HLS. Bobda [59] and Zhang [121] also exploited the similarity between operations across different partitions during HLS in order to reduce the time taken to reconfigure the partitions.

Outside of HLS, cell parameterisation was of particular interest to the reconfigurable computing community, who sought placement and routing of parameterised circuits [51] or cores [119] during actual circuit execution. In a way not dissimilar to the compile-time approaches, the cores were highly stylized to ensure regularity in the use of routing and logic resources, thereby minimising the complexity of the task.

Similarity in cell binding at the technology level ensures that few changes occur in the unit of reconfiguration. For example, Bobda [59] reduced the reconfiguration time by re-using similar functional units and reconfiguring resources only when the design constraints permitted.

Commonality between device configurations can be thought of as the lifetime of a resource. For example, Bobda [59] and Zhang [121] effectively consider the life of a functional unit’s configuration to extend beyond a temporal partition; Cardoso [66] and Trimberger [36]

spatially share functional units, confining their resource binding to the lifetime of the partition.

The re-use of circuit structures at the device level is reliant upon no other changes in the unit of reconfiguration; to do so would require reconfiguration of the unit – defeating the motivation in using it to reduce the reconfiguration time. The size of the reconfigurable unit in some devices [21] was convenient for exploiting at the logic level [120]; at most, a fraction of a logic cell would be wasted in an obsolete device such as the Xilinx XC6200. Current devices, such as the Virtex family, have a unit of reconfiguration comprising a column or tile of multiple logic cells, therefore wastage of resource is of much greater concern.

In addition to the placement of logic resources, the transportation of signals between them is in no small way the crux of the problem; spatial sharing of functional units assumes that the wires, as well as their loads are always physically present, it just a matter of selecting between them. Temporal sharing of reconfigurable resources or functional units common to different partitions of those resources, require a persistent intermediary in the form of an interface; spatially relating signals generated from different partitions. Some approaches feature no routing outside of the partition, except to convey the data-flow through to another temporal partition using the same resource. This is common place for multi-context and fully reconfigurable devices or just as a means to reduce the complexity of the problem, Vasilko [120].

What technology binding does not do, is model where those resources are placed; this is not an omission because resource implementation, such as device layout, typically occurs at a later stage in a non-reconfigurable design flow. How this happens is the subject of the remainder of the chapter.

4.2 Overview of the Target Architecture

Figure 4.3 depicts the conceptual architecture which is the goal of the optimisation and partitioning phases of circuit synthesis. In essence, the infrastructure necessary to execute a temporally partitioned circuit is automatically generated without user specification (other than the original behavioural description and the optimisation objectives and constraints).

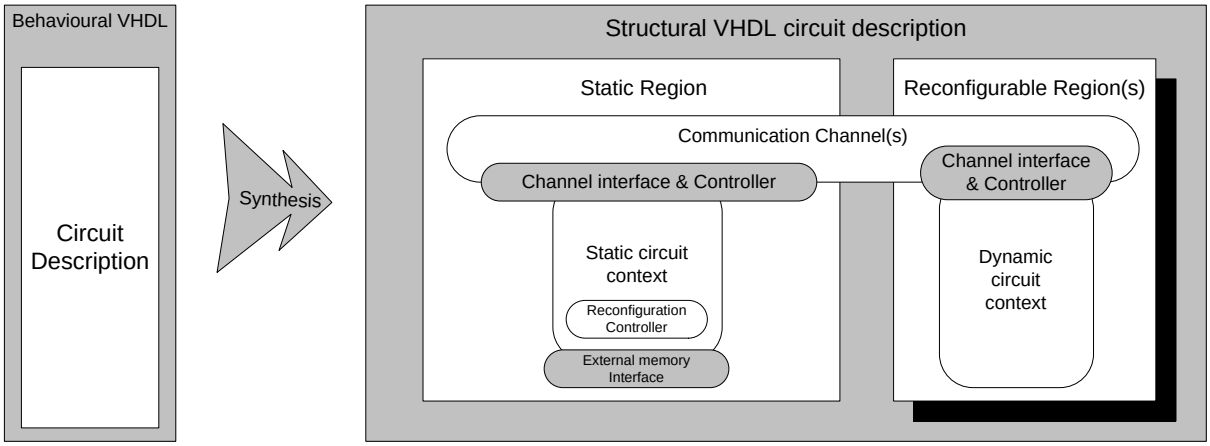


Figure 4.3: Architectural support for temporal partitioning.

The circuit modules which collectively embody the algorithmic behaviour of the circuit are partitioned into a set of ‘Circuit Contexts’. A single ‘Static’ context comprises the top level ‘Program’ module and those subordinate modules deemed too costly to be made dynamic during temporal partitioning. The static context is bound to a fixed set of programmable resources throughout the lifetime of its execution, thus forming the ‘Static Region’ of the architecture depicted in the figure.

Subordinate modules which are reconfigured ‘on the fly’, do so in the form of a ‘Dynamic’ circuit context, utilising the programmable resources of designated ‘Reconfigurable Region(s)’ of the FPGA.

The term ‘Region’ refers to a set of programmable resources, each of which is the minimum number of device columns necessary to permit an initial binding of the program or subordinate modules to a programmable resource. Modules may be bound to more than one resource column on condition that their placement is adjacent and contiguous when programmed in to the device configuration memory.

In practice, the static region is programmed only once upon device power-up; in contrast, programming of a reconfigurable region may occur anytime whilst power remains applied to the device: in between periods of execution, each circuit context remains resident on the region until it is swapped with another through a partial reconfiguration of the device – achieved by rewriting the relevant portion of the device’s configuration memory.

Circuit contexts which are not realised using the programmable resources of an FPGA exist in an external memory in the form of device configuration data-streams. Although storing the data-streams presents several additional overheads: one of which is the reconfiguration delay in their fetching and subsequent loading through the ‘External memory interface’ and is a significant factor in determining whether the module be made dynamic; any remaining overheads associated with an external memory are also present in non-reconfigurable FPGA systems which also store the power-up configuration external to the device.

With the availability of inexpensive Mega-Bit memory, any circuit module represented at this level of abstraction (device data-streams) is regarded during partitioning as virtual hardware with no area overhead. Central to this scheme is the interaction between the ‘Reconfiguration Controller’ and the ‘Static Region’: the controller must be present in the static region to perform self-reconfiguration of the FPGA at specific control states pre-determined during temporal partitioning.

Present in every region is the ‘Channel Controller’; its role is to facilitate access to a ‘Communication Channel’ from any region. Communication between any pair of modules not bound to the same resource must take place through a ‘Channel Interface’, ensuring the integrity of all control and data signals passing to and from, as well as through the static and reconfigurable regions during partial reconfiguration of the FPGA.

The characteristics of any communication channel (the number of signals it buffers and their vector width) along with all other infrastructure depicted in the figure are customised during partitioning: their attributes are entirely governed by the properties of the behavioural specification, as well as an optimisation priority assigned by the user to each of the metrics which quantify the circuit during synthesis.

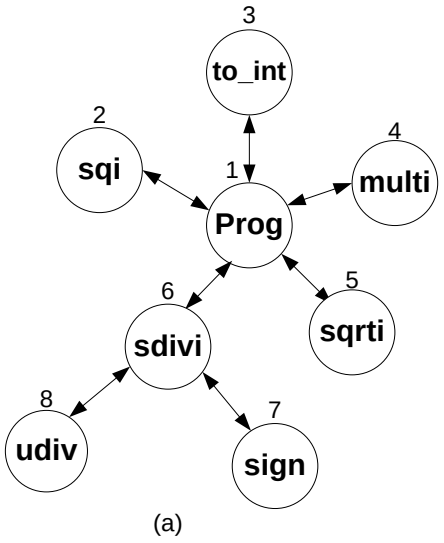
4.3 Partitioning Metrics

At any point before, during and after optimisation, a design is characterised in terms of its delay, area and clock period characteristics. Several additional metrics are required to quantify the effects of temporal partitioning, specifically:

- The total circuit area after partitioning the design into a set of circuit contexts.

- The time taken to execute the partitioned design through the continual switching of the target device between the circuit contexts during run-time self-reconfiguration.
- The degree of imbalance among the contexts assigned to a reconfigurable region and in doing so, provide a measure of the extent to which it is fully utilised.
- The number, width and length characteristics of all communication channels which bridge the dynamic and static regions of the device.

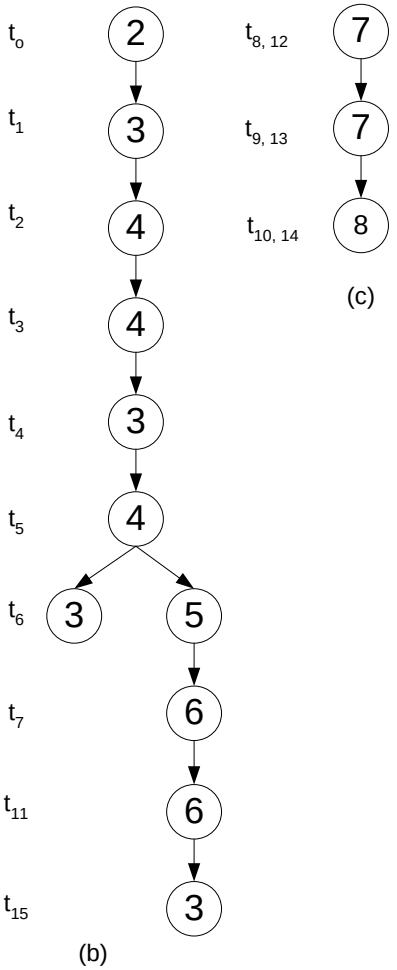
Figure 4.4 (a) depicts the connectivity between the program module and subordinate modules employed in an implementation of a quadratic equation solver; it will be used to exemplify the use of each of the metrics during temporal partitioning.



(a)

module	Area (slices)	Net width(bits)
prog	707.4	n/a
sqi	649.5	64
to_int	52.7	64
multi	666.7	96
sqrti	940.8	64
sdivi	291.9	96
sign	23.5	33
udivi	857.6	96

(d)



(b)

(c)

Figure 4.4: The characteristics of a quadratic equation solver implementation.

Each module would have originally taken the form of a behavioural description of a frequently used function or procedure, held in a library of commonly used routines and parsed into a set of data structures, alongside the calling module – be that another subroutine or the program module.

The order of execution for each of the sub-modules called by the program module is shown in the acyclic graph of (b), along with the sub-graph (c) of nested module calls activated during the execution of the sub-module ‘sdivi’. Each vertex in (b, c) is labelled with the number of the sub-module being executed, where the order of execution is defined by the direction of the arcs. The initiation of every sub-module execution is referenced by a unique time step shown adjacent to each vertex. The parallel time steps shown alongside the vertices of the sub-graph denote multiple executions of the associated module, where the numbering of each vertex is relative to each execution of the module.

During time step ‘t_6’ the graph splits into two distinct paths which derive directly from a conditional instruction inherent to the behavioural specification, they later converge (not shown) prior to the start of another execution cycle during step ‘t_0’.

The initial area and I/O port widths of each of the modules are characterised in Figure 4.4 (d) and are known prior to partitioning. For the sake of clarity, their attributes remain fixed during the following overview of each metric. In practice, however, they are variable, since in addition to temporal partitioning they are also subjected to optimisation using scheduling and allocation transformations. Their combined effect is examined in greater detail in Chapter 5, where the resource binding transform is introduced.

4.4 Problem Formulation

Allowing multiple conflicting objectives to be specified enables the optimisation algorithm to explore the trade-offs between different aspects of the design being synthesised. This is achieved through the cost function, to produce a figure that it can use to guide the optimisation of the design to meet the user supplied targets for each circuit characteristic being quantified. Before defining the cost function, the partitioning problem and each of the metrics are first formulated.

A circuit description is represented as a control graph *C* and data-path *D*, such that:

(a) The graph $C=(S,A)$ is directed, cyclic and composed of a set of control state nodes *S* and a set of arcs *A*, where each pair of control states $(n_i) \in S$ and $(n_{i+1}) \in S$ is connected by an arc $a_i \in A$.

(b) The data-path $D=(V,E)$ consists of *V* nodes and *E* edges, where every node $v_i \in V$ represents an operator, sub-module (function or procedure) and each edge $e_{ij} \in E$ is a data dependence between it and the next node v_j.

Let:-

A_{TP} represent the area (slices) of the partitioned graph *TP*.

A_{Bal} measure the balance or variation in size among the circuit contexts assigned to a reconfigurable region.

T_R quantifies the reconfiguration time (ns) associated with implementing the set of dynamic circuit contexts.

B returns the number of tri-state buffers required to implement the communication channels.

The task during synthesis is to find a spatial and temporal partitioning *TP* of the graph *C* and data-path *D*, whereby some or all of the metrics (A_{TP}, A_{Bal}, T_R, and *B*) meet their associated targets.

If:-

S_C is a set of modules assigned to the static partition

D_C is the set of all dynamic circuit contexts.

M is the sub-set of sub-module nodes where $M \subset V$

C is a subset of *M* modules which together form a dynamic circuit context.

For the circuit partition $TP = S_C \cup D_C$, the set of dynamic circuit contexts D_C is given by:

$$D_C = \bigcup_{i=0}^{R} C_i,$$ where n+1 represents the number of circuit contexts (n > 0).

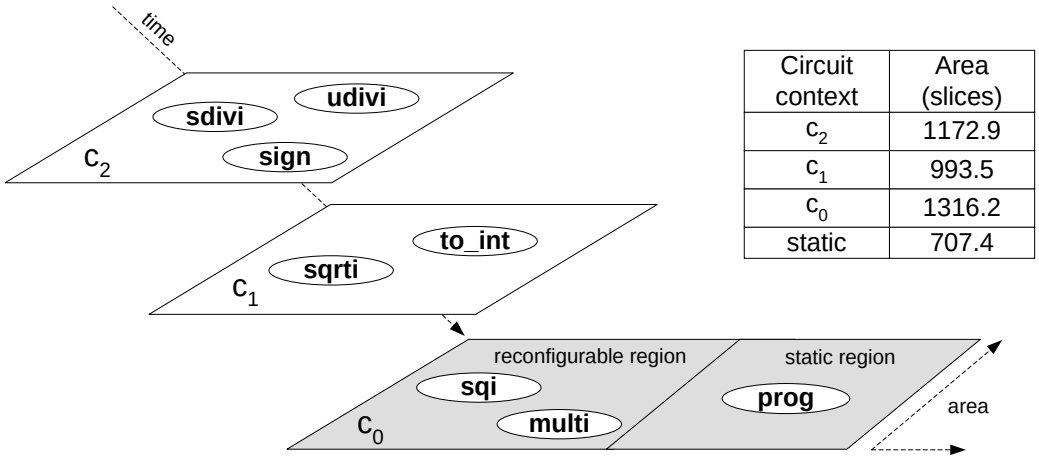
The Control and Data-paths are correctly partitioned when:

- $S_C \cup D_C = M$: all sub-modules are mapped to a circuit context
- $\bigcap_{i=0}^{n} C_i = \emptyset$: each sub-module node $v_i \in M$ is mapped to only one dynamic context
- $S_C \cap D_C = \emptyset$: sub-modules are assigned to either a static or dynamic context.

4.5 Circuit Area

If two or more sub-modules share the same device resource at different times during their execution, then at any point the area required for their implementation is equal to the largest module. This notion can be further elaborated to form a group of sub-modules or a circuit context whose placement is constrained to a specific reconfigurable region of the device. Figure 4.5 shows one such partitioning of the quadratic equation solver.

The total circuit area is given by the sum of the largest dynamic context 'C_0' and the static region containing the program module. For this configuration the circuit area is 2023.6 CLB slices [6], half the size of an un-partitioned circuit of area 4190.0 slices.



Circuit context	Area (slices)
C_2	1172.9
C_1	993.5
C_0	1316.2
static	707.4

Figure 4.5: A temporal partitioning of the quadratic equation solver.

The area of the circuit design after spatial and temporal partitioning (A_{TP}) is required to be less than or equal to the user supplied target i.e. $A_{TP} \leq A_{TP\,\text{target}}$. Circuit area A_{TP} is the sum of

the static and reconfigurable regions of the device, where the size of each reconfigurable region is determined by the largest circuit context assigned to it, formally:

$$A_{TP} = A_S + \sum_{i=0}^{n} A_{region_i}$$, where n+1 is the number of reconfigurable regions

Let *max(x,y)* return the larger of two circuit contexts (x,y) measured in Xilinx CLB slices, then A_{TP} may also be determined by the overhead associated with interfacing the control and data channels (A_{Ch}) to a region (in the event of the area required by the channels being greater than the largest circuit context C_i) that is:

$$(C_i, C_{i+1}, \ldots, C_n) \in region_i;\ A_{region_i} = max(max(A_{C_i}, A_{C_{i+1}}, \ldots, A_{C_n}), A_{ch}),$$

where n+1 is the number of circuit contexts assigned to region i.

4.6 Reconfiguration Overhead

A reduction in circuit area achieved through the continuous switching of an FPGA between a set of configuration contexts incurs a cost in the form of the Reconfiguration Overhead, the time taken to load a data-stream associated with a temporal circuit context in to the configuration memory of the device and the frequency at which this occurs during the course of a design's execution.

Commercial FPGAs such as the Xilinx Virtex [6] family are not optimised for fast reconfiguration, the bottleneck being the loading of configurations through a byte wide port at a maximum frequency of 50MHz or 20 ns per configuration byte. Partitioning at the modular level requires a partial reconfiguration of at least an entire column of FPGA resources at the device level.

At any point during the partitioning of the design, the area of a context can be used to determine the time taken to configure it. The time taken to load a circuit context is given by the product of the number of configuration cycles required to load a single column of the FPGA and the number of columns required to implement the context on the device. A column of a Virtex FPGA consists of 48 configuration frames [6], each of which consists of a different number of 32-bit words depending upon the actual device chosen.

The characteristics of a Virtex FPGA, in terms of the number of columns, rows, tri-state buffers and configuration frame length f_L (in words) varies depending upon the size of FPGA targeted.

During the course of partitioning, the current area estimation of the design after context switching is used to select the smallest target FPGA capable of implementing current design. This "best fit" approach ensures that the circuit contexts are realised through the smallest configuration data-streams and in doing so, directly impacts upon their reconfiguration times. A model of the target device provides the attributes required to calculate the metric in question, whether it be the number of tri-state buffers available for the communication channel or in this case, the word length f_L of a frame for the target FPGA.

$$\text{The total number of configuration bits per column} = 48 \cdot f_L \cdot 32 \text{ (bits per word)}$$

$$= 1536 \cdot f_L$$

In SelectMap mode [6], the Virtex configuration bus is one byte wide, therefore:

$$\text{the number of configuration cycles required per column} = 192 \cdot f_L$$

The internal configuration memory state machine of the device requires an additional 24 clock cycles:

$$\text{the number of configuration cycles required per column} = (192 \cdot f_L) + 24$$

$$\text{The area of a column (CLB slices)} = 2 \cdot r,$$

where r is the number of CLB rows in the device and each CLB comprises 2 slices (Virtex FPGA).

The number of columns (rounded up to the nearest integral column) utilised by a context

$$= \frac{\text{Area of context}}{2 \cdot r}$$

Therefore, the time taken to load a context T_r is given by:

$$= \frac{\text{Area of context}}{2 \cdot r} \cdot \big((192 \cdot fL + 24)\big) \cdot 20 \text{ ns}$$

This metric provides a lower bound estimation of the reconfiguration time due to the assumption that the placement of each circuit context is done in such a way as to maximise the use of each column's resources. Predicting how the placement and routing tools implement the partitioned design within each resource is beyond the scope of this thesis,

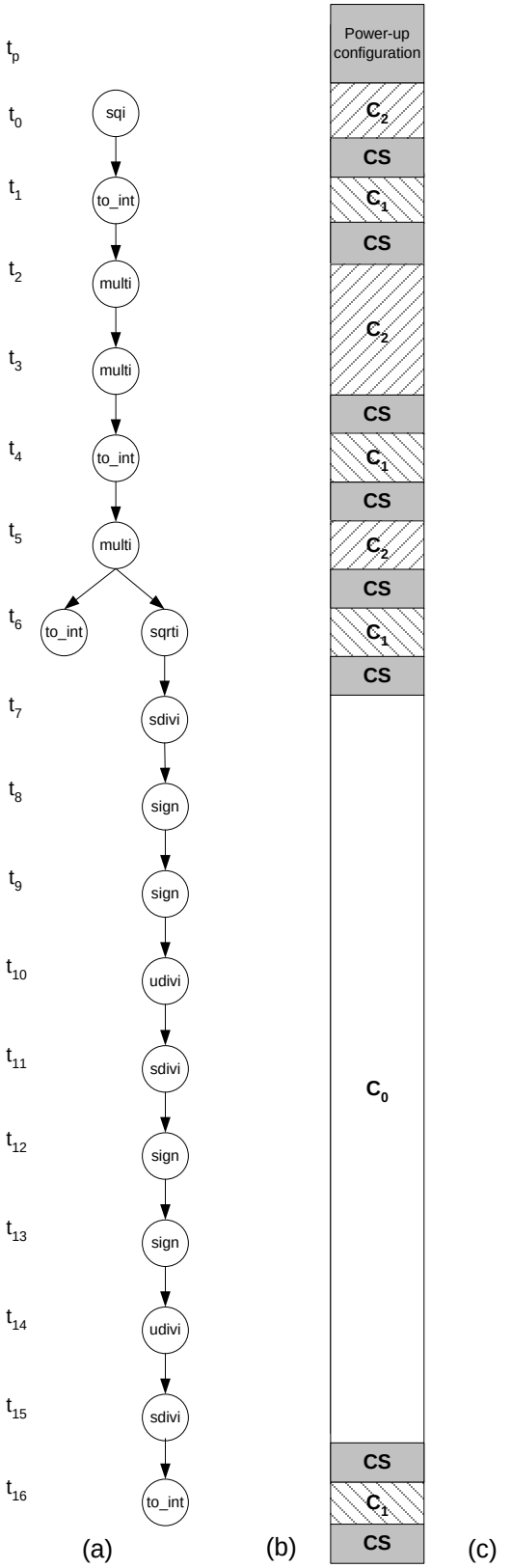
however, guiding the process towards an outcome that makes good use of the resources is necessary to ensure the value in estimating the reconfiguration times.

This is achieved during the implementation stage of the design flow, where a number of placement constraints are used by the vendor placement and routing tools, to define a general floor-plan for the partitioned design which, along with other synthesised structures, collectively form the architecture required to facilitate context switching at the device level. The estimation of the reconfiguration time need only be accurate enough to steer the decisions taken during optimisation in a direction, that will ultimately generate a circuit that satisfies the constraints and targets placed upon its synthesis.

An estimation of the reconfiguration time is sought after, rather than its exact prediction, due to the level of abstraction at which behavioural synthesis is done. Data-path and control graph operations are bound to physical cell characterisations which enable more accurate trade-offs between area and delay criteria during optimisation, however, their scope does not extend to RTL synthesis and logic optimisation using third party proprietary tools; therefore optimisation does not account for the device-level refinements which occur during the implementation of the design using device vendor specific processes such CLB ‘Packing’.

4.7 Frequency of Resource Context Switching

The second factor and by far the greater contributor to the reconfiguration overhead is the frequency of context switching required of each reconfigurable region, in order to preserve the original execution order of the sub-modules described in the behavioural specification. Figure 4.6 (b) depicts the context switching required of a single region used to implement the partitioned quadratic equation solver of Figure 4.5, in response to its sub-module order of execution shown alongside (a). The history of the context switching is marked against the periods of the graph labelled (t_0-t_{16}), each denoting a call to and subsequent execution of a sub-module contained within the circuit context active on the region. The time step ‘t_p’ denotes the power-up phase of the target device, when it is configured with the static portions of the design which include the infrastructure that facilitates run-time self-reconfiguration.



(c)

Circuit context	Member modules	Area (slices)	Reconfiguration time (ms)	Frequency
C_0	sdiv, udiv sign	1172.9	1.46	1
C_1	sqrti, to_int	993.5	1.23	4
C_2	sqi, multi	1316.2	1.64	3
Reconfiguration overhead = 11.3 ms				

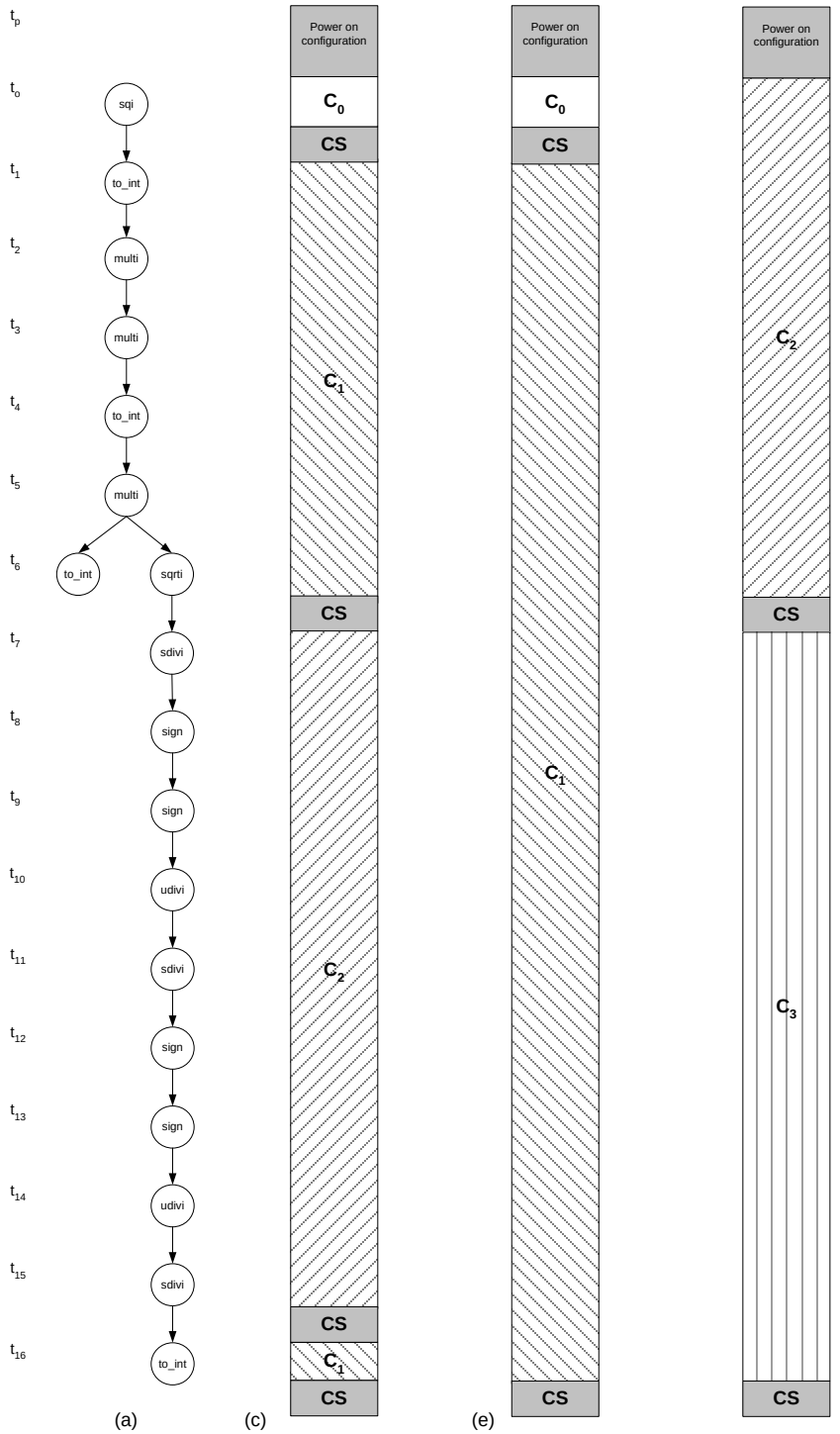
Figure 4.6: Context switching of the partitioned quadratic equation solver.

The reconfigurable region is seeded with the first circuit context 'C_2' after which, the execution of the design commences at step 't_0' with the activation of the sub-module 'sqi' currently resident on the region. The next sub-module to be executed is 'to_int', since it is contained within another context 'C_1', it must be switched 'CS' with context 'C_2' through a partial reconfiguration of the reconfigurable region prior to its activation. The process is repeated ad infinitum.

The reconfiguration time and the number of context switches required of each context are tabulated (c) alongside, where the sum of each of their products gives the total reconfiguration overhead of 11.3 ms.

Figure 4.7 shows the results of re-partitioning the quadratic equation solver with the aim of reducing the degree of context switching. The approach taken in Figure 4.7 (b,c) is to group together those sub-modules whose execution is frequently alternated between during the course of the design, for example, modules 'to_int' and 'multi'; their assignment to different circuit contexts was responsible for the majority of the context switching shown in Figure 4.6 (b). With hindsight, re-partitioning the equation solver with regard to the level of context switching directly impacts upon the overall reconfiguration overhead, the end result being a significant reduction in reconfiguration overhead to 6.39 ms.

An additional method to reduce the reconfiguration overhead is depicted in Figure 4.7 (d,e) and requires the introduction of multiple reconfigurable regions. Ensuring that modules 'to_int' and 'multi' are separately partitioned (Figure 4.7(d)) but concurrently active, reduces the need to frequently swap between them and has a secondary effect of context 'C_1' remaining active on its region during all the times scheduled for the execution of its sub-modules. This eliminates the last context switch necessary in the previous partitioning and in doing so, reduces the reconfiguration overhead to 3.71 ms.



Circuit context	Member modules	Area (slices)	Reconfiguration time (ms)	Frequency
C_0	sqi	649.5	0.81	1
C_1	sqrti, to_int, multi	1660.2	2.06	2
C_2	sdivi, udivi,sign	1172.9	1.46	1
Reconfiguration overhead = 6.39 ms				

(b)

Circuit context	Member modules	Area (slices)	Reconfiguration time (ms)	Frequency
C_0	sqi	649.5	0.69	1
C_1	sqrti, to_int	993.5	1.06	1
C_2	multi	666.7	0.71	1
C_3	sdivi, udivi,sign	1172.9	1.25	1
Reconfiguration overhead = 3.71 ms				

(d)

Figure 4.7: Multi-region context switching of the partitioned quadratic equation solver.

Reconfiguration Overhead metric:

If:-

P is the set of all distinct control paths through the circuit on which there are sub-module calls.

P_i is an individual path through the circuit on which a unique sequence of sub-module calls lie, where $P_i \in P$.

P_{prob} refers to the control path most likely to be taken during the execution of a circuit and is determined by a profile of the circuit obtained during its simulation.

P_{worst} denotes the longest path in terms of the reconfiguration overhead and is used where specific parameters of the circuit are only know at run-time or when a set of paths are deemed equally likely to occur.

The reconfiguration overhead T_R for the set of all dynamic circuit contexts D associated with the control path (P_{prob}) most likely to be taken is given by: ${T_R}^{P_i}(D)$, when P_i=P_{prob}. When the most likely path cannot be determined, the path whose order of sub-module execution incurs the largest reconfiguration time (P_{worst}) is used i.e.

$$T_R{}^{P_{worst}}(D) = max\left(T_R^{P_i}(D), T_R^{P_{i+1}}, \dots, T_R^{P_n} \text{ where } P \in (P_i, P_{i+1}, \dots, P_n)\right).$$

The reconfiguration overhead is given by the sum of the time taken to swap each of the circuit contexts on to their designated reconfigurable regions, for all regions; it is evaluated in relation to the designer's target such that $T_R(D) \leq T_{R_{target}}$.

$$T_R^{P_i}\left(region_j\right) = \sum_{k=0}^{n} T_R^{P_i}(C_k)\text{ , where n+1 is the number of circuit contexts}$$

$$T_R^{P_i}(D) = \sum_{j=0}^{m} T_R^{P_i}(region_j)\text{ , where m+1 is the number of reconfigurable regions}$$

If:

$CS^{P_i}(C_k)$ is the number of context switches required of circuit context C_k during the execution of the path P_i.

$T_{config}(C_i)$ is the time taken to reconfigure the target device with the context C_i.

$T_{overlap}(C_i)$ measures the reconfiguration time of C_i reduced by overlapping it with the execution of a section of data-path.

The reconfiguration time of the context C_k is obtained by the product of the number of context switches required of it and the time taken to load it into the target device's configuration memory each time a switch occurs. It can be reduced by the time spent overlapping each reconfiguration of a dynamic circuit context with the execution of a data-path unit(s), be it an instruction operation, sub-module function or procedure i.e.

$$T_R{}^{P_i}(C_k) = (CS^{P_i}(C_k) \cdot T_C(C_k)) - T_{overlap}$$

Although the number of times a circuit context is swapped with another is entirely dependent upon the modules assigned to it and their sequence of execution given by the path P_i, an upper bound is given by the maximum frequency F of execution calls amongst its member modules M:

$$CS_{max}^{P_i}(C_k) = max(F(M_0), (F(M_1), \dots, (F(M_n)) \text{ where } C_k \in (M_0, M_1, \dots, M_n)$$

Each time a dynamic region is partially reconfigured with the circuit context C_k, the time taken to load the context is proportional to its modular area (slices), device specific parameters (frame length f_L and the number of CLB rows r) and the time taken for the reconfiguration controller to interface with an external memory, $T_{interface}$. This process is inclusive of the time taken to fetch the data-stream representing the context, load it into the configuration memory of the FPGA and verify that these tasks have been successful prior to the execution of the context on its designated reconfigurable region.

$$T_R^{P_i}(C_k) = \left(\left(\frac{1}{2 \cdot r}\sum_{i=0}^{n} A_{C_i}\right) \cdot \big((192 \cdot f_L) + 24\big) \cdot 20\ ns\right) + T_{interface}$$

4.8 Scheduling the Context Switching

How a design is partitioned and the sequence of module calls which lie on the control path most likely to be executed, will determine the length of time a circuit context remains on a

reconfigurable region and the latest time it can be swapped with another. Deciding exactly when it is swapped is found by scheduling its reconfiguration.

Figure 4.8 (c) illustrates how the reconfiguration of the partitioned quadratic equation solver tabulated in 4.8 (d) may be implemented, by determining when each of the dynamic circuit contexts can be swapped in to their designated regions. Each context switch of the target device exploits its ability to partially reconfigure 'on the fly', by overlapping the reconfiguration of each circuit context with the execution of a sub-module(s) resident on another dynamic or static region. The motivation behind this approach is to reduce the reconfiguration time associated with each context switch, its effectiveness is dependent upon the characteristics of the design being synthesised and the configuration rate of the target device on which it is implemented. For example, circuit designs which regularly interact with I/O devices where there is human participation, such as displaying a graphical user interface, are more tolerant of the millisecond reconfiguration times associated with commercial FPGAs which are not optimised for frequent reconfiguration. Therefore, it is the ratio of the reconfiguration to execution times that will determine the value in overlapping circuit execution with reconfiguration.

Marked along the program module graph of Figure 4.8 (a) and sub-graph (b), are segments of graph (S_0-S_4) which overlap the reconfiguration of a circuit context. Each segment may comprise a number of sub-module calls, in addition to general vertices which enable signal transfers in their associated module and in practice typically make up the majority of the graphs and thus, are fodder for the scheduling transformations.

Having determined which of the calls to the sub-module selected during partitioning can be made directly to the relevant circuit context and which cannot (necessitating a context switch of the region), the task is to determine when or more specifically where in the graph calling the sub-module, can the context switch be initiated.

This is achieved using one of three scheduling methods, the first of which is to schedule a context switch as soon as possible (ASAP) and doing so without overlapping the reconfiguration of an earlier circuit context.

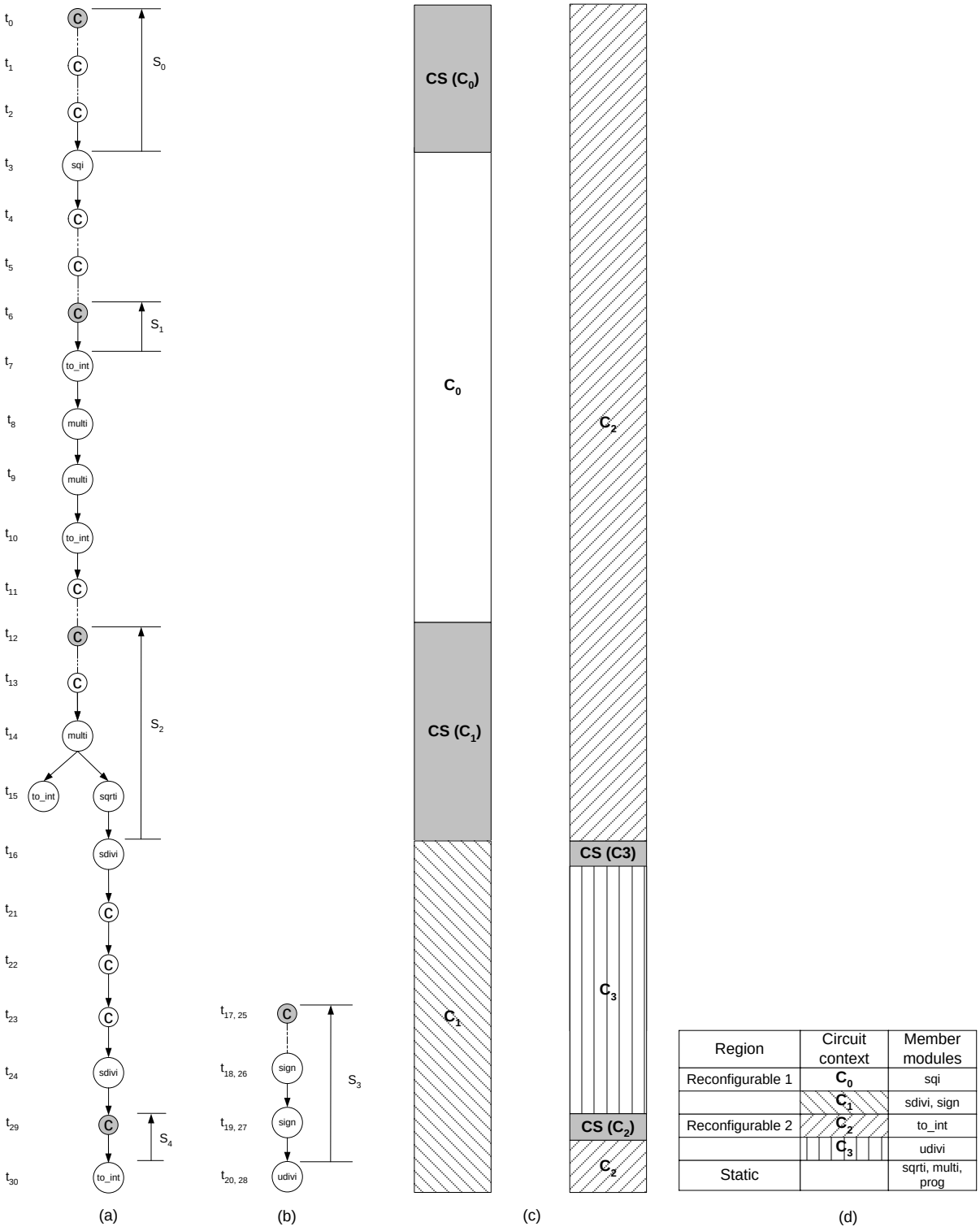


Figure 4.8: Scheduling the context switching of reconfigurable regions.

At the beginning of segment S_0, the context C_0 is scheduled to reconfigure on region 1 at the earliest opportunity t_0, this is indicated by the shaded vertex. Although not exemplified in the graphs, a control state may also mark the beginning of a segment for multiple control paths. This transpires when the control state is common to the multiple paths, before a path diverges for example or as a part of a sub-module. The dashed edges which connect the vertices represent the existence of numerous general vertices, not shown to simplify the example. Scheduling the reconfiguration this early, provides the greatest potential for reducing the reconfiguration time of C_0, prior to the first execution call of its sub-module sqi, at the end of the segment.

A complementary approach is illustrated in segment S_1, where the latest possible time (t_6) to schedule the reconfiguration of the context C_1 is shown. Initiating the context switch of the region one cycle prior to the earliest execution of one of its sub-module to_int, incurs the full reconfiguration overhead for C_2.

At first glance, it appears sensible to give preference to reducing the reconfiguration time by scheduling ASAP, however this can give rise to a side effect that discourages the partitioning of those sub-modules which are overlapped by the reconfiguration of another. Consider sub-modules 'sqrti' (t_{14}) and 'multi' (t_{15}) whose execution calls are overlapped during segment S_2 by the reconfiguration of C_1. The modules, in addition to the program module are presently assigned to the static context. However, if either or both of them are re-assigned to a dynamic context, their configuration will overlap with that of context C_1. Since this is prohibited, a solution is to re-assign the modules which collectively form C_1 to the static context, subject to approval of the cost function. It is possible that the cost of such a move proves too great, particularly if the reconfiguration of C_1 is scheduled much earlier up the control graph (in doing so, benefiting C_1 by reducing its reconfiguration time). Depending upon the method of optimisation employed and when the degradation occurs during optimisation, the outcome may be sufficient to deter the assignment to the static context of the sub-modules overlapped by the segment.

Had the reconfiguration of C_1 been scheduled to commence as late as possible (ALAP), the overlap in configuration would not have occurred. The cost function would then be presented with a different scenario to quantify, one where an existing context does not require

re-assigning to the static region, in order for a sub-module to be assigned to a reconfigurable one.

Although scheduling the reconfiguration of a context ALAP reduces the occurrence of future overlaps in configuration, it provides no reduction in the reconfiguration time of a circuit context. It is not implausible to imagine a scenario where a marginal reduction in such time may be the deciding factor which leads to the acceptance of a sub-module for temporal partitioning. To that end, a compromise between the two scheduling extremes is provided by determining their difference (mobility or slack) and selecting a vertex at random from the resulting partial graph, which in the case of segment S_2 is the vertex whose execution occurs during t_{12}.

Segment S_3 illustrates the interaction which can occur between the static and dynamic regions and how the scheduling might be applied in such circumstances. As a member of the circuit context C_1 and currently active on region1, sub-module ‘sdivi’ is executed (t_{16}) by the program module resident in the static region. During its execution (sub-graph (b)), a context switch of region 2 occurs (t_{17}) where C_2 is swapped for C_3 to enable the execution of the nested sub-module udivi. The time taken to achieve this is reduced by overlapping the reconfiguration of the udivi with the execution of ‘sign’ in the first reconfigurable region. Upon completion of the reconfiguration process, ‘udivi’ is executed ($t_{20,}t_{31}$) and control is returned to the program module. No further context switching is necessary during the next execution of sdivi (t_{24}).

The final segment of graph S_4 depicts when the context switch of the second reconfigurable region is required in order to restore sub-module to_int to the region prior to its execution. In common with the earlier segment S_1, an ALAP scheduling for its reconfiguration has occurred, but unlike the earlier segment, the scheduling would have been restricted by the assignment of sub-module udivi to the same region during temporal partitioning. The interdependence between temporal partitioning and instruction scheduling is a strong motivator for implementing these tasks in the same stage of high-level synthesis and will be described in greater detail in the next chapter.

If :-

I is the set of ICODE instructions which embody the behaviour of the design being synthesised.

L represents the set of moduleLeap instructions (sub-module activation instructions), where $L \subset I$.

L_A is the 'moduleLeap' instruction that executes sub-module *A*.

M_A is the set of instructions which are mutually exclusive in their execution to sub-module *A*.

R is the set of reconfigurable resources of an FPGA device.

R_J is a single reconfigurable resource such that: $R_J \subset R$.

A dynamic circuit context D_{C_i} can be context switched with another $D_{C_{i+1}}$ over a common reconfigurable region i.e. $D_{C_i} \in R_J$ and $D_{C_{i+1}} \in R_J$ when neither are concurrently active. Specifically, if $A \in C_i$, $B \in C_{i+1}$ then sub-module *A* may be swapped with *B* if its activation takes place on a control path whose execution is mutually exclusive to that of sub-module *B* i.e. $L_A \in M_A$ or when the execution of sub-module *B* occurs after that of *A* on the same control path.

When scheduling the context switching of a reconfigurable region(s) with a set of dynamic circuit contexts, the original order of module execution must be respected; the two modules *A* and *B* for example, whose execution occurs in that order,

Let :-

S_a, S_b denote the control steps where the execution of each module begins.

$S_{exe}(i)$ represent a single step during which a module is executed.

SD_{C_i} refer to the steps taken to perform the reconfiguration of a circuit context.

P_{length} return the length of the path where reconfiguration is scheduled to occur, the beginning of which is denoted by control step S_0.

In order to swap modules *A* with *B* where $A \in C_i$ and $B \in C_{i+1}$ and preserve their original execution dependency, the execution of *B* is scheduled to commence at:

$$S_b \geq S_a + \sum_{i=S_a}^{j} S_{exe}(i) + \sum_{i=k}^{l} S_{DC_i}, \text{ where } C_i \in R_j \text{ and } C_{i+1} \in R_j \text{ for } S_a < S_l \text{ and } S_k < S_l$$
$$\text{assuming that } R_j \in R.$$

i.e. module *A* is swapped with *B* on a single reconfigurable region then module *B* is scheduled to begin execution at step S_b , only once *A* has been executed (steps $S_a - S_j$) and the reconfiguration of the circuit context containing *B* (steps $S_k - S_l$) has been completed.

Were the circuit contexts to execute on multiple reconfigurable regions i.e. $C_i \in R_j$ and $C_{i+1} \in R_{j+1}$ then the configuration of module *B* could be scheduled to overlap the execution of module *A*. In this case, the configuration mobility of module *B* maybe any step from the beginning of the path S_0 i.e. ASAP, until the step before the start of its execution S_{j-1} (ALAP).

If sub-modules A and B lie on complementary paths (following the execution of a conditional branch instruction) and each path has an equal likelihood of being taken, then only one circuit context may start reconfiguration before the result of the conditional instruction is known i.e. ASAP. In the event of a sub-module call to a circuit context which is not configured 'on silicon', the context with the smallest reconfiguration time is chosen to be loaded ALAP: in doing so, the cost of reconfiguring the wrong context is minimised. This cost can be eliminated entirely if the context is comprised of both modules A and B, in doing so covering both eventualities of path execution. The cost function would have to weigh the potential reduction in reconfiguration time against the binding of resources to one idle module of the pair, which by definition cannot execute.

This overview would not be complete without mentioning that the scheduling transformations can also influence the context switching of the reconfigurable regions and therefore their use is also quantified by this metric. One such example is the Sequential Merge transform. When data-path operations are chained during the merger of their associated control states, their ICODE instructions are scheduled to execute within the control step of the earliest. If the latter state happened to have been selected from the reconfiguration segment of a temporal circuit then its context switching instruction (denoting the beginning of the segment) will also be subjected to the merger of the states. The effect will be an earlier scheduling of the reconfiguration, the result of a lengthening of the segment by the number of control states

bounded by the two chosen to be merged. There are several other scenarios where the sequential merge and other transforms have an impact not only upon the scheduling of reconfiguration, but also on the formation of the partitions themselves and these are presented in detail in the next chapter.

4.9 Communication Channels

The partitioning of sub-modules over a single static and multiple dynamic circuit contexts necessitates a permanent communication channel which forms the backbone of the device level architecture discussed in greater detail in Chapter 5. The communication channel guarantees the physical connection of the partitioned module's control and data signals. In doing so, it enables their inter-region execution and data transfer, independent of the circuit contexts to which there are assigned and the physical placement of their associated reconfigurable regions. This is in direct contrast to communication between every pair of modules in the static region, which is done though dedicated control and data signals internal to the region.

During the run-time execution of a temporally partitioned design, as circuit contexts are swapped on and off their reconfigurable regions, the physical interface of each context must be identical to that of the reconfigurable region to ensure a predictable and unbroken signal transfer between the static and dynamic regions of the device. This is made possible by driving each signal through a pair of tri-state buffers whose placement is constrained to straddle the boundary of a region, thereby forming an interface between the communicating contexts on either side.

Although in plentiful supply, the number of tri-state buffers available on a Virtex FPGA is proportional to its size and therefore assigns a hardware cost to the number and bit-width of those signals which require buffering when entering to or leaving from a reconfigurable region. As with the reconfiguration metric, the current area estimate of the design after partitioning is used to select the size of the target device from which the number of available tri-state buffers are obtained.

Figure 4.9 shows the relationship between the number of signals cut during temporal partitioning and the effect this has on characteristics of the communication channels. Figure 4.9 (a) depicts a partitioning of the quadratic equation solver which targets multiple reconfigurable regions on the FPGA. The sub-modules are grouped in to their respective contexts (C_0 - C_3), where each arc represents the numerous data-path signals which pass between the modules and is marked with the sum of their bit vector widths ‘W’.

In addition, there are two control lines solely used in the communication between a pair of modules (not shown for simplicity), one used by the calling module to activate the called module and the other to signify the completion of its execution and hence the return of control to the calling module. When a circuit context is swapped with another, the sub-module port signals overlap in time, determining the width of the channel and therefore the number of tri-state buffers ‘B’ required to interface it to other regions whose contexts drive or are driven by it.

Figure 4.9 (b) illustrates the dimensions of the channels required to implement the partitioned quadratic equation solver. Independent control and data channels are employed to relay the signals necessary for the execution of a pair of modules at either end of the channel. Their separation is not motivated by functional necessity, rather an aid to floor planning, where a channel of small dimensions offers a greater degree of flexibility in its placement during the device-level implementation.

To reduce the size, number and ultimately the cost of buffering the data and control signals, each channel is shared by several pairs of modules at either end, provided that they are mutually exclusive to all others in their regions (recall that to context switch between two modules requires that neither module be simultaneously active on the reconfigurable region). Similarly, two pairs of concurrently active modules cannot share the same channel without some form of arbitration.

Sharing the channel would provide a feasible solution to the only occurrence of concurrent module execution in the partitioning of Figure 4.9 (a), following the program module’s activation of module ‘sdivi’. Once enabled, it deepens the execution hierarchy through its execution of modules ‘sign’ and ‘udivi’.

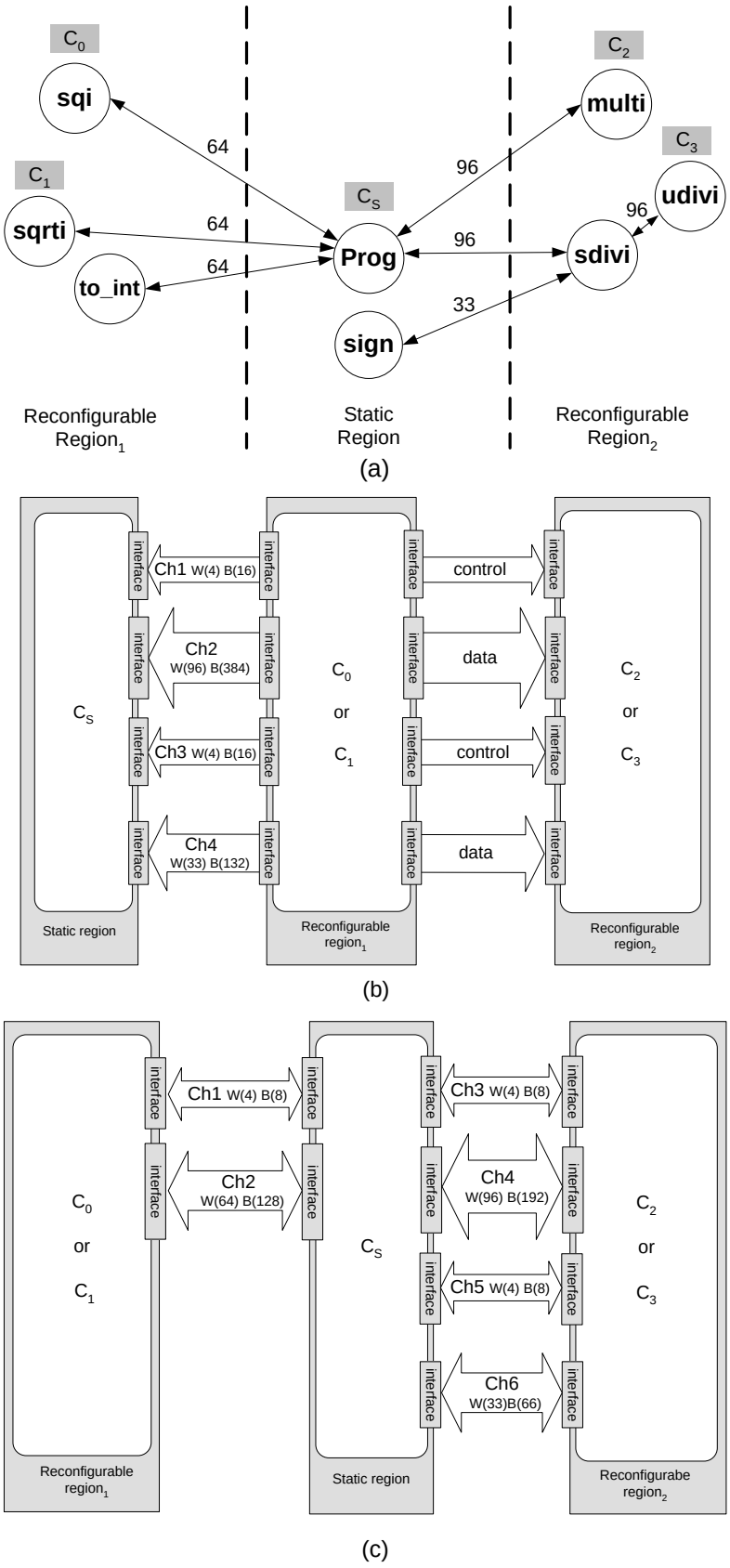


Figure 4.9: Impact of module partitioning and placement on communication channels.

Such an execution hierarchy requires a degree of concurrency since each initiating module remains active until the sub-ordinate module completes its execution and returns control. Hierarchies may have a depth of several modules, each waiting on the successive pair to complete its execution before it can do the same and return control to its initiating module or ultimately the program module.

Implementing an arbitration protocol would incur an additional delay overhead in their execution which may prove too great should the chain of communicating pairs be of significant length. In addition, it could restrict the use of performance enhancing techniques, such as the Pipelining of their execution. As such, the static and reconfigurable regions may be bridged by multiple control and data channels, as exemplified in Figure 4.9 (b).

Two control channels 'Ch_1', 'Ch_3' are required to implement the control signals associated with the sdivi sub-module hierarchy, due to its concurrent nature of execution. The control of all other partitioned modules, being mutually exclusive or sequential in their execution to sdivi and those modules it activates, may share either of the channels. Each channel is at least 4 bits in width: in this minimum configuration, a single module activation signal enables each channel to uniquely address two reconfigurable regions, each with a context comprised of a maximum of 2 modules. The width of a channel is by no means fixed, as the addressing can be scaled to accommodate any number of regions and sizes of contexts and therefore its actual width is determined by the formation of the partitions themselves. Further architectural details can be found in Chapter 5.

Like the control channels, the number of independent data channels used to realise data transfer reflects the degree of concurrency of the modules that use them. Once again, two channels 'Ch_2' and 'Ch_4' are simultaneously active as data signal conduits between all the regions associated with the '*sdivi'* sub-module hierarchy. At all other times 'Ch_2' is shared by the remaining modules although at any instant, exclusively by a single pair of modules.

Regarding the dimensions of the channels, the width 'W' of Ch_2 is equal to the context with the greatest bit-width, that of contexts 'C_2' and 'C_3'; they require an interface comprising 96 tri-state buffers. The width of 33 buffers for channel Ch_4 is smaller, due to the greater number of remaining signals in C_3 being internal to the context and its singular use in the connection

of module ‘sign’ resident in the static region to its calling module assigned to circuit context C_3.

The channel utilisation depicted in Figure 4.9 (b) provides a minimum overall width for the transfer of the modules control and data signals. During partitioning, the total channel width is compared with the maximum number of buffers per CLB column, returned by a model of the target device. The model is chosen based on the area of the partitioned design. In doing so, it gives priority to maximising the logic resources of the device, enabling the best fit for the static and reconfigurable regions. Referencing the model forms the basis of a validity test which ensures that the width of the channels required to support the proposed module partitioning does not exceed the resources offered by the device: to do so would create channels too large to be implemented on the target device and consequently the partitioning is rejected before being applied.

Having determined the placement of the module, the temporal partitioner may elect to connect it using one of several channels that may already pass through the region. Its choice will be guided by the number of buffers required to implement the channel and as such, should aim to minimise the increase to the width of all parallel channels and therefore reduce the risk of a module move being rejected.

In the event of a module successfully fitting in to a circuit context, the actual estimation of the cost of implementing its communication channels is undertaken. The true cost in buffer consumption ‘B’ of a channel is given by its width and length, measured by the number of region boundaries through which it crosses. As a reconfigurable region must span the entire column of the device, a channel cannot circumvent any region which lies in its path. This is evident regarding the global channels depicted in Figure 4.9 (b) which pass through all three static and reconfigurable regions of the device.

Clearly a direct influence on the length of a channel is the location of the modules that use it, through their assignment to existing contexts. However, it is also the position of the reconfigurable regions, in relation to the static region which offers the choice of placement for a module and in doing so, dictates the number of regions that it must cross. This is illustrated in Figure 4.9 (c), where once again the quadratic equation solver is partitioned over three regions. However, in this alternative channel implementation, the location of each new region

was taken into consideration during its creation. As a result, the global channels (Ch_1-Ch_4) have been replaced by shorter ones (Ch_1-Ch_6), local to the regions which they connect. As a consequence, the latter channel implementation represents a more cost effective realisation; the total of all tri-state buffers B has been reduced from 548 to 410 respectively. By assigning a metric to quantify the number of tri-state buffers required to realise the channel(s), a measure of the cost effectiveness for a given temporal partitioning of modules is achieved by comparing the number of buffers required to realise the channel(s) with the maximum available, once again provided by the model of the target device.

Communication channel metric

Consider the temporal partitioning illustrated in Figure 4.10; it is formed by partitioning a set of sub-modules, $(V_1, V_2, \dots, V_7) \in M$ and the program module V_0, such that:

$V_1 \in C_0, \quad V_6 \in C_1, \quad (C_0, C_1) \in Region_{-1}; \qquad V_4 \in C_2, \quad V_7 \in C_3 \quad (C_2, C_3) \in Region_1;$

$V_0 \in C_S, \quad V_2 \in C_S, \quad V_3 \in C_S, \quad C_S \in Region_0; \quad V_5 \in C_4, \quad V_8 \in C_5 \quad (C_4, C_5) \in Region_2.$

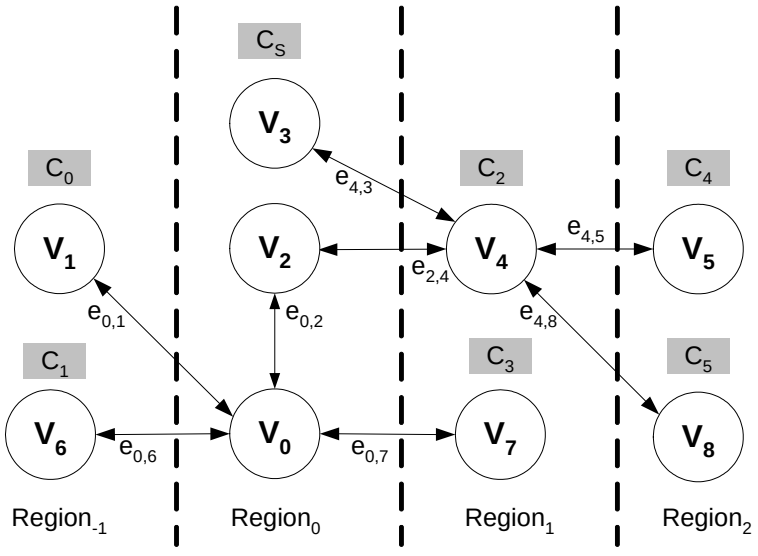


Figure 4.10: An example of temporal partitioning.

There are 7 pairs of inter-region communicating modules depicted in Figure 4.10, where the connectivity of each is denoted by a single bi-directional arc representing the edges set ($e_{i,j}$),

necessary to enable the execution of module V_j by module V_i. Each execution is realised through separate control and data signals represented by a subset of edges, namely $Ce_{i,j}$ and $De_{i,j}$. An individual edge is associated with at least a single bi/uni-directional one bit signal exchanged between the pair of modules (V_i, V_j).

If:-

$region_i$ denotes the reconfigurable region over which a set of dynamic contexts are swapped, where: $0 > i > 0, i \neq 0$.

$region_{i=0}$ refers to the static region containing the program module and any sub-modules deemed too expensive to be made dynamic.

The execution of sub-module V_j residing in $region_k$ by module V_i within $region_L$ attributes a weight $We_{i,j}$ to those edges $e_{i,j}$ cut by the partitioning of their associated modules over multiple regions, such that $e_{i,j} \in \psi$, where ψ is the set of all edges cut by temporal partitioning. The cut-set is quantified in terms of the number of tri-state buffers employed in the construction of the pair of data and control channels (Ch_m, Ch_{m+1}), which together bridge those edges divided across the regions during partitioning. For example, with reference to Figure 4.10, in order to implement the partitioned sub-module V_1, where $V_1 \in C_0$ of the pair (V_0,V_1), requires $We_{0,1} = 2(Ce_{0,1} + De_{0,1})$ buffers: recall that each channel signal requires two buffers, each anchored at either end of the communicating regions that utilise the channel. Contrast this to sub-module V_2, where $V_2 \in region_0$, activated by the program module V_0 through a set of edges $e_{0,2}$ which are internal to the static region and therefore without need of a channel and the buffer overhead required in its realisation i.e. $W(V_{0,2}) = 0$ when $e_{0,2} \notin \psi$.

If another sub-module, such as V_6, whose execution is not concurrent to V_1 is also assigned a dynamic context (C_1) and swapped with the context C_0 containing V_1, each set of edges will share a pair of channels (Ch_0 ,Ch_1), such that $(Ce_{0,1}, Ce_{0,6}) \in Ch_0$ and $(De_{0,1}, De_{0,6}) \in Ch_1$. In this way, although there are 7 pairs of modules and therefore 7 sets of edges, their partitioning does not require 7 groups of data and control channels, only 2. The rationale for 2 channels comes from an examination of the sub-module hierarchy initiated by sub-module V_2. Although it engages 4 sets of edges during its execution, only 2 edges are simultaneously active between the same regions, namely $e_{2,4}$, $e_{4,3}$. Accordingly, they dictate that a minimum

of 2 groups of concurrently active channels are required at any point during the execution of the partitioned design.

The mapping of the remaining edges of the hierarchy $e_{3,8}$, $e_{3,5}$ is undertaken by the temporal partitioner, in such a way as to attempt to satisfy the optimisation objectives associated with the channel metric and/or other metrics. The same can also be said of the outstanding edges ($e_{0,7}$, $e_{0,6}$, $e_{0,1}$). One such mapping of edges to channels which the partitioner may take is as shown in Figure 4.11: it illustrates the mapping of each set of edges to the minimum number of concurrently active channels (Ch_0, Ch_1), formally:

$$(e_{0,1},\ e_{0,6},\ e_{0,7}) \in Ch_0, (e_{2,4},\ e_{4,3},\ e_{4,8},\ e_{4,5}) \in Ch_1$$

At any time during the execution of the design, only a single set of edges may occupy the channel to which it is assigned. This requires the execution of the pair of modules related by each set of edges to be either mutually exclusive or non-overlapping in nature. Therefore, an aspect to consider when mapping each set of edges to a channel is the extent of their concurrency, since it will determine the minimum number of channels required to carry them across the boundaries of the regions they connect to.

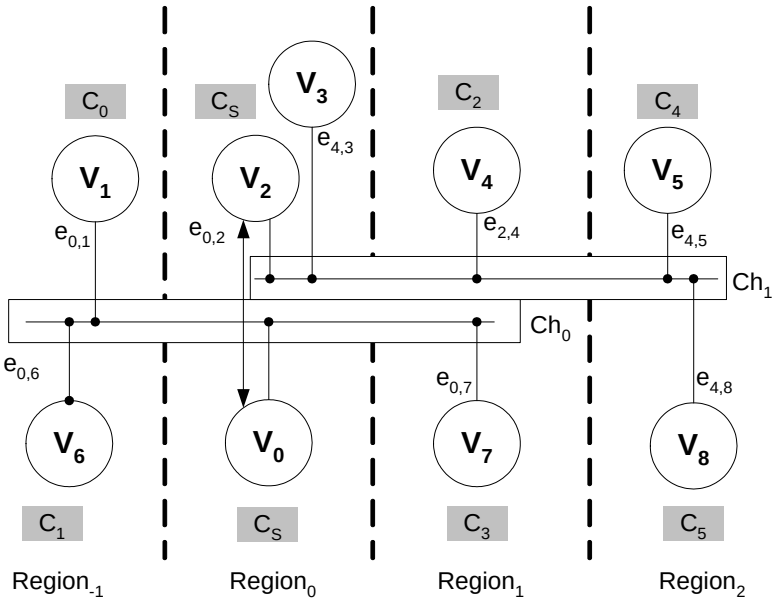


Figure 4.11: The mapping of concurrent channels.

The exact permutation of edges assigned to a channel will of course depend upon the contribution that each set of edges makes to its total weight, determined in part by the width of the set containing the greatest number of edges e.g.

$$|CH_0| = max(|e_{0,1}|, |e_{0,6}|, |e_{0,7}|).$$

The complete cost of realising a channel is then found from its width and the length − the number of regions it must span to permit the connections between any driving and driven module.

To uniquely identify the position of a reconfigurable region (in relation to the static region ($region_0$)), it is characterised as being positive or negative (left or right of the static region, respectively). The extremity of a channel, at either end, is associated with the region of the furthest driving and driven modules. In this way, the length of a channel is given by the absolute value of the difference between the two regions e.g.

$$Ch_{0(length)} = (|-1-1|) = 2; \;\; Ch_{1(length)} = (|0-2|) = 2$$

Evidently, the temporal partitioner should take into consideration the effect which the location of a region has upon the length of the channels spanning the device: the cost of implementing a channel at the device level can now be expressed as:

$$W(Ch_i) = 2(|Ch_i| \cdot Ch_{i(length)})$$

Consequently, at any point during temporal partitioning, the channel metric *B(TP)* quantifies the number of buffers necessary to implement all communication channels:

$$B(TP) = \sum_{i=0}^{n} \mathrm{W(Ch_i)}, \text{ for n+1 channels.}$$

Re-examining the assignment of edges shown in Figure 4.11, in terms of the combined weight of both the channels, reveals that there exists a large variation in the size of those edges allocated to each channel. It can be a disadvantage to map each set of edges to the minimum number of channels, suppose this to be the case and the sizes of each set of edges are as follows:

$$|e_{0,1}| = 20 \text{ edges}, \quad |e_{0,6}| = 25, \quad |e_{0,7}| = 45,$$

$$|e_{2,4}| = 15, \quad |e_{4,3}| = 20, \quad |e_{4,8}| = 55, \quad |e_{4,5}| = 60.$$

The width of each channel would therefore be:

$|Ch_0| = \max(20,25,45) = 45$, $|Ch_1| = \max(15,20,55,60) = 60$ and hence their weights:

$W(Ch_0) = 2(45 \cdot 2) = 180$, $W(Ch_1) = 2(60 \cdot 2) = 240$, generating a total cost of:

$$B(TP) = 180 + 240 = 420 \text{ buffers.}$$

Re-assigning the edges with a greater sensitivity to their size and the number of regions through which they cross produces the following mapping:

$$e_{0,1}, e_{0,6} \in Ch_0, \quad e_{0,7}, e_{2,4} \in Ch_1, \quad e_{4,3} \in Ch_2, \quad e_{4,8}, e_{4,5} \in Ch_3.$$

The width of each channel would now be:

$$|Ch_0| = \max(20,25) = 25, \quad |Ch_1| = \max(45,15) = 45,$$
$$|Ch_2| = 20, \quad |Ch_3| = \max(55,60) = 60.$$

and subsequently their weights:

$W(Ch_0) = 2(25 \cdot 1) = 50$, $W(Ch_1) = 2(45 \cdot 1) = 90$, $W(Ch_2) = 2(20 \cdot 1)=40$,
$W(Ch_3) = 2(60 \cdot 1) = 120$, the result of which is a noticeably reduced use of buffers:

$$\text{B(TP)}=300.$$

Although there are double the number of channels, they can be placed in two parallel groups and collectively their width is no greater than the two channel implementation i.e.

$$\max(|Ch_0|, |Ch_1|) + \max(|Ch_2|, |Ch_3|) = 45 + 60 = 105.$$

It should be clear to the reader that the greater the number of channel buffers, the more likely their channels are to contribute to the circuit partitioning being rejected by the cost function. In the worst case scenario, the partitioned circuit may not fit on the target device: recall that the target for the channel metric is set by the size of the buffer resources associated with the model of the target device. The model may be changed several times during partitioning, to ensure that the reconfiguration overhead is based upon parameters of the device which provide the best fit for the static and reconfigurable regions during the device-level implementation of the partitioned design. Therefore, the temporal partitioner should also take

into consideration the combined width of the channels when selecting suitable edges to assign to them.

The effect of each channel characteristic cannot be quantified in isolation, for instance, the combined width of the 4 channel solution maybe reduced by mapping the edge $e_{0,7}$ to Ch_3 as follows:

$$|Ch_0| = \max(20,25) = 25, \quad |Ch_1| = 15, \ |Ch_1| = 20, \ |Ch_3| = \max(45,55,60) = 60,$$

where the combined size of the two widest channels is:

$$|Ch_0| + |Ch_3| = 25 + 60 = 85.$$

However, although re-assigning the edge to Ch_3 means that channel Ch_1 no longer contributes towards overall parallel width, a secondary effect is an increase in the buffer use since channel Ch_3 must now pass through two regions and hence the cost of re-assigning the edge becomes:

$$e_{0,1}, e_{0,6} \in Ch_0; \ e_{2,4} \in Ch_1; \ e_{4,3} \in Ch_2; \ e_{4,8}, e_{4,5}, e_{0,7} \in Ch_3.$$

$$|Ch_0| = \max(20,25) = 25, \quad |Ch_1| = 15, \ |Ch_1| = 20, \ |Ch_3| = \max(45,55,60) = 60,$$

$$W(Ch_0) = 2(25 \cdot 1) = 50, \quad W(Ch_1) = 2(15 \cdot 1) = 30, \ W(Ch_2) = 2(20 \cdot 1) = 40,$$

$$W(Ch_3) = 2(60 \cdot 2) = 240; \ B(TP) = 360 \text{ buffers.}$$

Thus far, each aspect regarding the assignment of a set of sub-module edges to a communication channel has been examined in terms of its singular effect on the channel metric. Grouping edges of similar regional proximity or weight has an impact on the number of buffers required to realise a channel, however, the opportunity to do so will of course depend upon the priority and effect of the other metrics; after all, the fact that an edge has a weight (that is to say at least one of its associated modules is context switched) is the result of the cost function in conjunction with the post-partitioning metric determining that the proposed module move brings the partitioned design a step closer to the user-specified area target.

In summary, the task during temporal partitioning is to find a mapping of edges cut during partitioning ψ, to a set of communication channels Ch, in such a way that its characteristics (quantified by their number, length and width) require the number of tri-state buffers during

their implementation to be no greater than the capacity of the resources offered by the target device i.e. $B(TP) \leq B_{target}$.

4.10 Balancing the Partitions

The size of a reconfigurable region is equal to the largest circuit context assigned to it. A large variation in size among a set of contexts not only results in poor utilisation of the device resources for that region, but over multiple regions can lead to a larger than desired circuit area: this is evident when returning to the partitioning of the quadratic equation solver over multiple reconfigurable regions, as tabulated in Figure 4.8 (d). The total circuit area at any point during the execution of the equation solver is given by the area of the static partition and the sum of each of the areas required by the reconfigurable region. This requires 2166.4 CLB slices to implement the two dynamic regions. Figure 4.12 shows the area reduction which can arise when partitioning with the aim of reducing the degree of imbalance across the partitions assigned to a region.

Reconfigurable region	Circuit context	Area (slices)	Standard deviation (slices)
1	C_0	649.5	172.0
	C_1	993.5	
2	C_2	666.7	253.1
	C_3	1172.9	

(a)

Reconfigurable region	Circuit context	Area (slices)	Standard deviation (slices)
1	C_0	649.5	8.6
	C_2	666.7	
2	C_1	993.5	89.7
	C_3	1172.9	

(b)

Figure 4.12: Re-partitioning to improve resource utilisation of reconfigurable regions.

Table 4.12 (a) presents the results of the earlier partitioning when the temporal binding of the resources was carried out solely to overlap the reconfiguration of one circuit context with the execution of sub-modules assigned to another. Table 4.12 (b) demonstrates that by simply exchanging circuit context 'C_1' with 'C_2' the total area can be reduced to 1839.6 CLB slices.

A measure of the imbalance between a set of circuit contexts can be obtained using the standard deviation of a population, where the population is the set of contexts assigned to an individual reconfigurable region. Figure 4.12 shows the standard deviation tabulated for each of the reconfigurable regions. The metric provides a measure of the area variability or spread

of the context sizes from the average context area for each reconfigurable region, reflecting the improved balance (high to low) which occurred following the re-assignment of C_1 and C_2.

The reader will appreciate the significance of being able to measure a change in resource imbalance: during temporal partitioning, a balance metric is used to decide whether a sub-module should remain in its current partition (be that static or dynamic) or be moved to a partition where it will have a local effect by improving the balance of the region on which the partition is executed; in doing so, it offers the ability to influence the cost function in way that could result in a reduction in the total circuit area.

Balance metric

The metric A_{Bal} provides a measure of the spread in area (CLB slices) in circuit contexts belonging to a reconfigurable region, in doing so, it returns an indication of how well a region is balanced in relation to the user supplied target, explicitly: $A_{Bal} \leq A_{Bal_{target}}$.

This is achieved using the standard deviation σ of the population of n circuit contexts assigned to a region. It is evaluated only for those reconfigurable regions affected by the movement of a sub-module from its present region (source) to a destination. Where both the source and destination for the module are reconfigurable regions, the overall effect of re-assigning the module is found by the average spread in context area across the two regions, formally:

$$A_{Bal} = \sigma_{source}, \text{ where the destination region is the static partition, else}$$

$$= \sigma_{destination}, \text{ where the source is the static partition, else}$$

$$= \frac{\sigma_{source} + \sigma_{destination}}{2}.$$

$$\sigma_{region} = \sqrt{\sum_{i=0}^{n} \frac{(C_i - \mu)^2}{n}}, \text{ where } \mu = \frac{1}{n}\sum_{i=1}^{n} C_i.$$

4.11 Cost Function

Although each of the metrics relate to a single aspect of temporal partitioning, they all have a common purpose and that is to measure the quality of the design with respect to those circuit

characteristics deemed important to the design specification. Explicitly, the cost associated with the set of temporal partitions *TP* is formulated as:

$$cost_{TP} = C_{part\_area} \cdot A_{TP} + C_{reconfig} \cdot T_R + C_{channel\_width} \cdot B + C_{bal} \cdot A_{Bal}$$

where: A_{TP} is the circuit area after temporal partitioning;

T_R returns the reconfiguration overhead for a given partitioning;

B is the number of tri-state buffers used used in the channel implementation;

A_{Bal} provides a measure of the resource usage by temporal circuit contexts;

C_{part_area}, $C_{reconfig}$, $C_{channel\_width}$, C_{Bal} are weighted constants which reflect the user-specified optimisation priority of the associated metric.

The cost function metrics are expressed as absolute values using technology-specific models, such as the actual configuration parameters associated with the target FPGA (used in estimating the reconfiguration times) or the area of a sub-module when assigned to a circuit context (in terms of the CLB slices required to implement its control states and data-path units). The motivation in doing so is to ensure that the circuit being synthesised closely resembles the actual physical implementation at the device level.

Returning to the target architecture shown in Figure 4.3, the reader is reminded of how essential the synthesis of the infrastructure is to meeting this requirement; its properties are individual to the design being synthesised, such as determining those periods in which the reconfiguration controller actively performs context switching of the reconfigurable regions or the dimensions and number of the communications channels required to connect them. Enumerating each property enables the temporal partitioner to make the relevant trade-offs and allows the user to explore their ramification through changes to the priority and constraint of each of the metrics.

When temporal partitioning occurs alongside optimisation to the control graph, *C*, and data-path, *D*, additional trade-offs are made possible between the area, delay, clock and partitioning metrics, further enhancing the exploration of alternative realisations of the circuit during its optimisation. Therefore, the task during optimisation is to minimise the extended description of the cost function now expressed as:

$$Cost(C,D) = C_{area} \cdot area + C_{delay} \cdot delay + C_{clock_{period}} \cdot clock + cost_{TP},$$

where: C_{area}, C_{delay}, $C_{clock_{period}}$ are once again, the priority weighted constants of each metric.

4.12 Summary

The purpose of this chapter has been to explain how temporal partitioning can be integrated into MOODS HLS in the form of reconfigurable resource binding. It described how one approach would be to re-use cells between temporal partitions, in a way not dissimilar to the traditional spatial re-use relied upon for multi-mode binding. This approach was not adopted due to the prevalence of conditional control flow in HLS and the uncertainty of knowing the exact resource configuration available at run-time, necessitating the actual approach relying more upon spatial sharing within a partition, rather than between them.

A multiple-objective approach to temporal partitioning was formally defined in terms of the characteristics that a cost function would need to measure through its metrics. In addition to the essential trade-off between the area reduction and resource reconfiguration, the cost of communication between the partitioned subroutines is also quantified. Measuring their characteristics is complicated by the potential transportation of their signals between spatial and temporal resources; adding an additional requirement to balance their resource use, further compounds the partitioning problem and provides the motivation for performing partitioning using a general purpose optimisation algorithm with which to explore their interaction.

Chapter 5

Implementing Run-time Reconfiguration

This chapter details the architecture that is automatically synthesised by MOODS for communication between the temporal partitions generated during optimisation and those parts of the architecture which facilitate self-reconfiguration of a Xilinx Virtex FPGA device. In particular, it describes how the abstract model of the temporal partition embodied by the data-structures is given form, in the context of the device-level architecture and how it may be implemented, irrespective of the limitations imposed by the device and vendor support.

The execution of the synthesised circuit is viewed as a series of activations between pairs of circuit subroutine modules. Each module is assigned to a static or dynamic region of the FPGA, where the individual module selected is determined by the temporal partitioner.

A module residing on the static region of the device is expected to remain on silicon throughout the lifetime of the circuit's execution. This is not the case for those assigned to reconfigurable regions, which as their name suggests, are subject to continual re-programming of the resources contained within their boundaries: in this way, the temporal aspect of partitioning is realised as groups of modules or circuit contexts being repeatedly swapped with one another.

When not active upon silicon, each circuit context takes the form of a device configuration data-stream and is stored in an external memory. A reconfiguration controller is deployed to fetch the data-streams and facilitate the circuit swapping, in accordance with the reconfiguration schedule determined during module partitioning and operation-level optimisation.

The communication channels form the backbone of the architecture, passing through region boundaries. When under the direction of the channel controller, control and data signals may be transferred between any pair of communicating modules, irrespective of the region in which they are placed and without interruption due to device reconfiguration. The structural VHDL description of the synthesised circuit incorporates the customised architecture required to implement the partitioning, through run-time self-reconfiguration of the FPGA device.

5.1 Architectural Abstraction

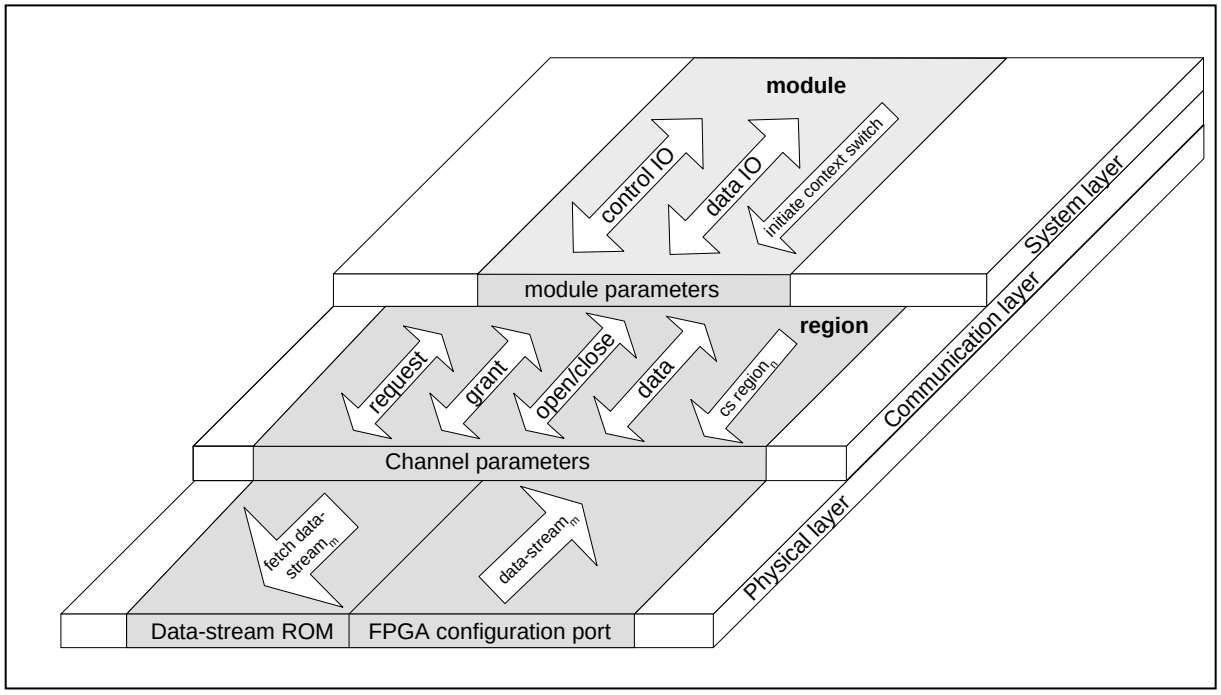


Figure 5.1: Abstraction of the architecture into distinct layers of circuit activity.

The function of the principle sub-systems such as the reconfiguration and channel controllers provides a natural means of abstracting a module's execution protocol into layers of circuit activity. Figure 5.1 depicts how this can be visualised: at the highest level of abstraction, the 'System layer' encapsulates the functionality of the circuit through the execution of the circuit modules. As depicted, control and data transfer between the modules are carried out without regard to where the modules are assigned (in terms of their region and circuit context) or the location of their associated data-streams. These details are processed during the lower levels of the protocol, the next level of abstraction being the 'Communication layer'. At this level,

the module control and data signals are translated into channel operations, for instance, competing for possession of the channel with other modules which are also concurrently active in different regions.

Of course, none of the above is possible without the programming of the FPGA's configuration memory. This is implemented in the 'Physical layer' and is regarded as two distinct tasks: the fetching of the data-streams and the control of the device configuration port, both of which are done 'on the fly' during circuit execution. The exact moment when these tasks are undertaken is determined at the system layer, although a context switch of a region is represented at all layers of abstraction.

At the system layer, it is synchronised to a number of control states in the FSM controller of the module initiating the circuit swap. The request is relayed along the control channel during the communication layer and initiates the fetching and loading of the data-streams in the physical layer. The exception to initiating a context switch solely in the system layer occurs in the event of a configuration error. The reconfiguration controller can respond to an error flagged by the device during reconfiguration by re-transmitting the last sequence of configuration bytes. Should the error persist, the data-stream of the entire circuit context must be re-loaded, in effect initiating a context switch from within the physical layer.

The response to any remaining error is a transition to a state of system failure in which the circuit remains idle whilst requesting external input. Although the origin of the error will have occurred in the physical layer, in a process akin to that of the context switch, it is represented at different layers of abstraction until reacted to in the system layer.

5.2 System-level Architecture

At this stage of circuit synthesis, the task is to convert the internal representation of the circuit to a structural description suitable for logic synthesis and technology or device-specific net-list generation by third party tools. A significant part of the structure is generated directly from the internal representation, as each control and data-path node is bound to a Library cell associated with a VHDL RTL description. Generating the Register Transfer Level (RTL) VHDL description of the program module and sub-modules is, at its simplest, a matter of

writing the internal representation to file. For instance, representing control signals in the control graph and net activations on the data-path as logic expressions in the VHDL description of the circuit.

With the exception of the circuit modules, the remainder of the architecture is generated using associations between the modules modelled in the data-structures. The generation of the communication channels is one such example: a single move during partitioning can re-assign a module to another region, where it may no longer require the use of a channel. It would be inefficient to constantly add or remove tri-state buffers to the data-path units of the modules moved during the course of partitioning. Rather, their existence is inferred from the relationships contained in the data structures and generated once partitioning is complete.

No doubt, the capability of the target device influences the synthesised architecture; however, it is the relationship between the circuit modules which has the greater impact on the final architecture; in particular, their relationship to one another and the frequency in which they are executed. For instance, a principal component of the architecture is the communication channel. The number of channels is decided during partitioning and the results presented in Chapter 6 show that where there are many related modules, they tend to be clustered together and communicate with the Program module through a single channel; this makes sense because partitioning a module execution hierarchy over a number of regions leads to a high communication cost. Multiple channels can reduce this cost although being dependent on the IO parameter width of the modules, they also are expensive to deploy and therefore rarely more than one channel is utilised.

Another example where the design specification directly affects the architecture is when the modules are called from within multiple processes. This does not refer to a single module which is shared between the processes but a sequence of modules being executed in parallel. The partitioner may decide that the cost of multiple channels outweighs the overhead associated with sharing a single channel between the processes, requiring an arbitration circuit to be added to the architecture.

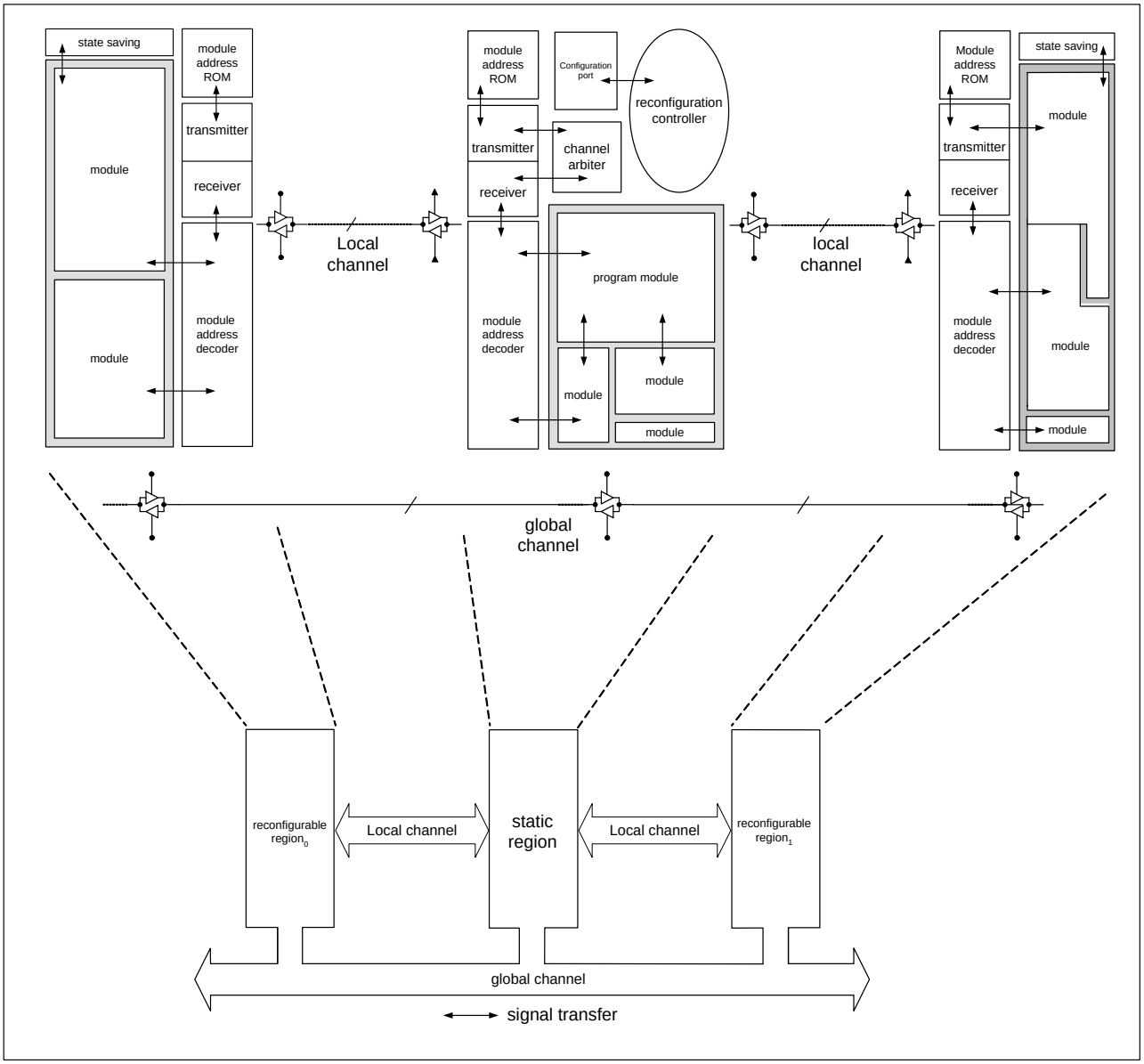


Figure 5.2: Synthesised architectural components.

Once optimisation and partitioning of the circuit design is complete, the circuit modules are realised in the context of the generic architecture depicted in Figure 5.2. It expands upon the conceptual architecture shown in Figure 5.1, to depict the actual components described within the structural output of the circuit being synthesised.

At its most fundamental, the architecture comprises a number of isolated areas of the target device, bridged by several local and/or a single global communication channel. The exact configuration of the components, as described earlier, is dependent upon the structure of the design being synthesised and the topology of the partitioned circuit. Of course, for any

temporal partitioning, there must be at least a pair of static and reconfigurable regions. The shaded blocks contain a number of partitioned modules and represent the circuit contexts currently active on their respective regions. Although it is not obligatory, the program module, being the highest point in the module execution hierarchy is generally assigned to the static region: to do otherwise, incurs a substantial overhead in context switching and state saving, since the program module is the source of all sub-module activation and parameter exchange and therefore the ultimate sink as well.

Alongside the program module, reside other modules, procedures or functions which can be directly written to and read from by the program module without the use of the communication channels. Figure 5.3 shows how this is achieved in practice: it illustrates the signal transfer (in terms of control and data) associated with the execution of a sub-module. At a purely behavioural level of abstraction, the sub-module call is allied with the execution of the ICODE 'ModuleLeap' instruction. As with all ICODE instructions, its execution is scheduled to occur during a specific step in the control graph, the legacy of which is a mapping to a FSM control state during the generation of the RTL description of the circuit. Although the mapping depicted refers to a one-hot controller implementation, the exact FSM encoding is determined by the RTL synthesis tool.

Unique to the ModuleLeap instruction is its implementation by a 'Call' node, utilised to invoke the execution of the sub-module in question. In this case, it is the sub-module procedure which is called and passed the input parameters, variables 'x' and 'y'. In practice, the parameters are passed by reference, where initially, the registers containing the variables are read directly by an equivalent pair of temporary registers in the sub-module. The temporary registers represent the parameters associated with the description of the sub-module, so called because they are liable to be removed during optimisation. Likewise, any output variable returned by the sub-module is bypassed and the result written directly to the variable parameter (register) of the calling module, exemplified in Figure 5.3 by the variable 'z'.

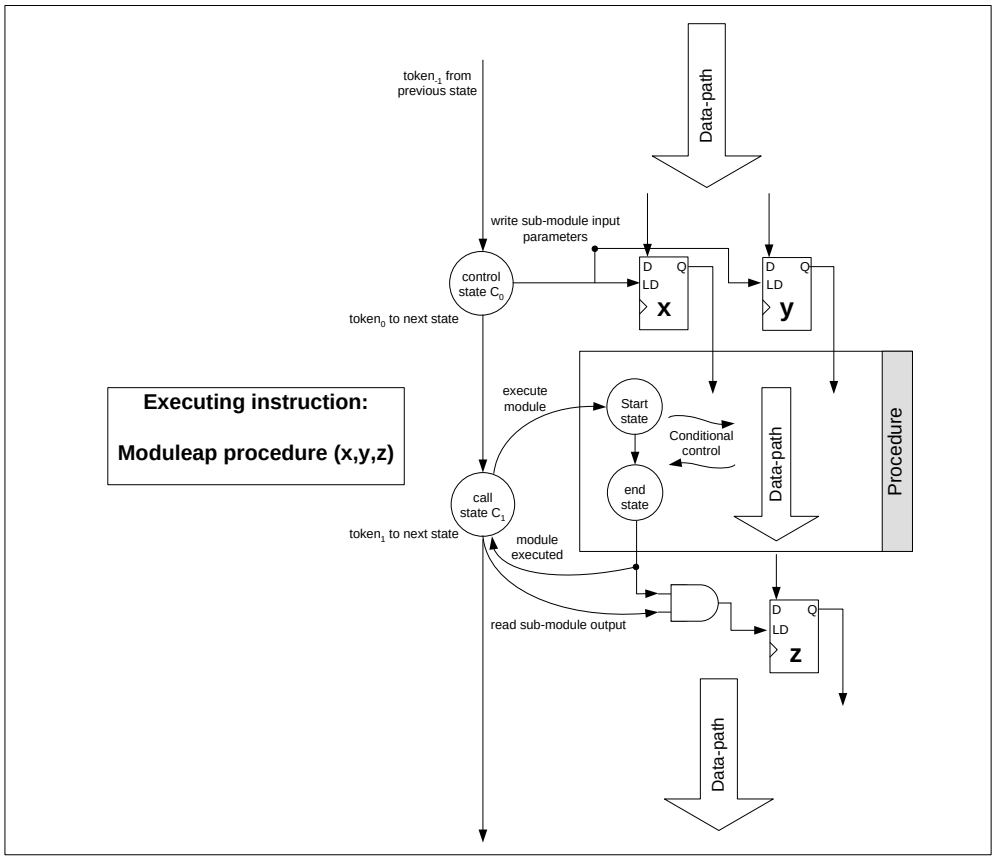


Figure 5.3: Sub-module execution and signal transfer.

The timing of this mechanism is shown in Figure 5.4. Having been passed token$_{-1}$ from the preceding state of the calling module (in this case the program module), the registers containing the variables x, y are loaded (labelled ‘a’ in the figure) during the execution of the state ‘C_0’. In the next clock cycle, at point ‘b’, the token is passed on to the calling state ‘C_1’ which promptly activates the sub-module by passing the token to its start node.

A registered version of this signal persists throughout the execution of the sub-module; should the procedure be called again and passed another pair of variables, the signal is used to drive a multiplexor select input, to match the sub-module inputs with the pair of register outputs corresponding to the current sub-module call.

Execution of the sub-module proceeds state by state (two cycles for this example-label ‘c’), with the state in possession of the token able to execute the relevant portion of the data-path.

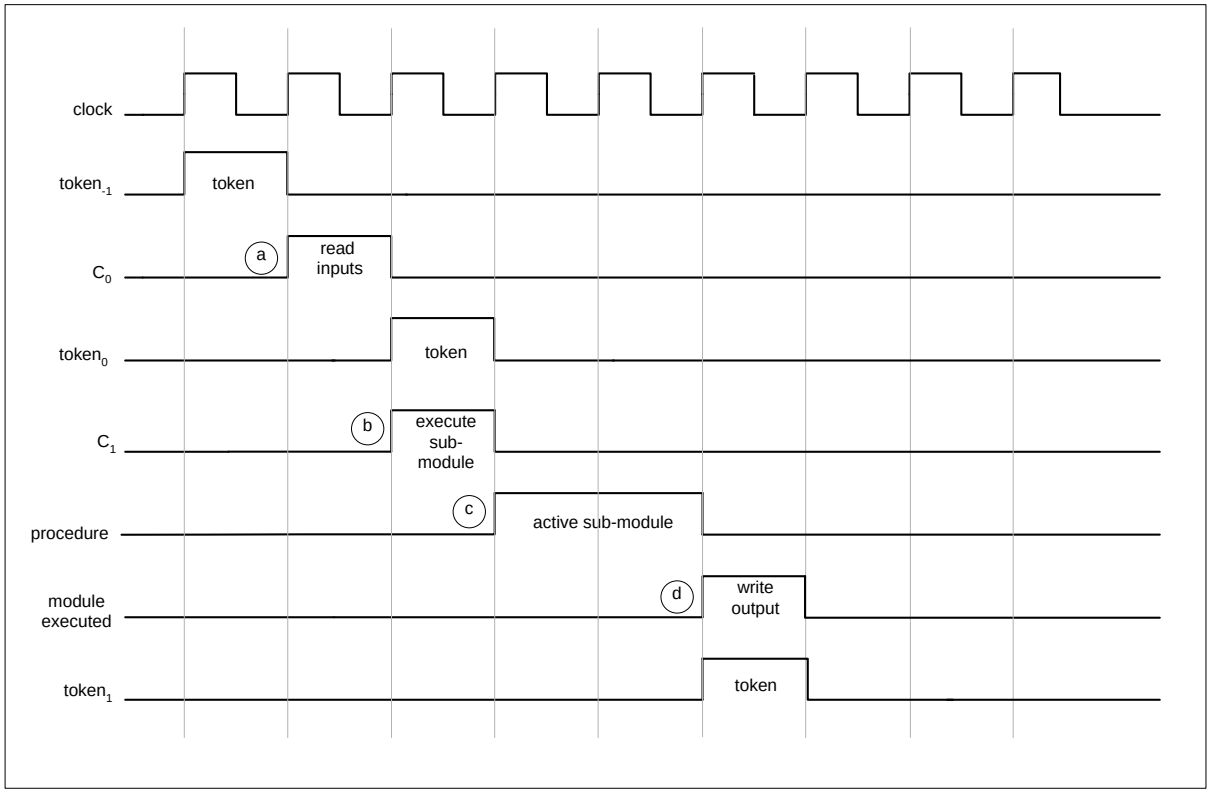


Figure 5.4: Direct sub-module execution timing.

The token returned from the last state of the sub-module is used, along with the registered signal from the call node, enabling the output register residing in the calling module to read the sub-module's output. The signal also generates the token necessary to activate the control state following the sub-module call – to read the register containing the variable 'z', in some further computation in the data-path.

5.3 Communication-level Architecture

With the exception of those modules which call and are called entirely within the static region, either or both of any pair of communicating modules require an address in the architecture. However, unlike the modules in the static region which will eventually be connected through dedicated routing, a dynamic module may share the control and data lines entering its region with the other members of the circuit context. The existence of another region necessitates a means of identifying the location of the calling and called modules. As such, a module is identified with the region and context to which it is assigned during

partitioning. In actuality, a third dimension to the address provides a physical reference to the location of its configuration data-stream in the external memory, although it is only realised at the physical layer.

With reference to Figure 5.2, the architecture associated with this layer comprises: the communication channels (global and local), the channel controller (‘transmitter’, ‘receiver’ and optional ‘arbitration’ blocks) local to each region, the location of the partitioned modules (‘module address ROMs’) and the storage of local variables (‘state saving’), the characteristics of all are dependent upon the topology of the partitioning.

5.3.1 Communication Channels

The communication channels form the backbone of the architecture and may be regarded as having a dual purpose: to guarantee the physical connection of the routing between the static region and successive partial reconfigurations of the reconfigurable regions and to enable the logical connection of a module in a circuit context to any other, irrespective of the regions in which they are active. In Chapter 4, the consequences of a move taken during temporal partitioning were examined in detail. In particular, a number of factors were identified which defined the characteristics of the communication channels. For example, the choices made by the partitioner, such as whether to pass the module parameters through an existing channel or whether or not to create a new one. Another important factor is the mutual exclusivity between the executing modules and it is inherent to the description of the source circuit. When the execution of the sub-modules is mutually exclusive, not only can the sub-modules share the same channel without contention, but with careful scheduling, any partial reconfiguration initiated during the course of their execution has no effect on the transfer of their signals across the channel. The advantage of this approach is to utilise the resources available in the reconfigurable regions, at the expense of tighter scheduling constraints for the context switches.

A shared channel with arbitration is synthesised when the execution of the sub-modules is not mutually exclusive and dedicated channels are not targeted. The increased traffic on such a channel places a greater demand upon the scheduling, as there are likely to be fewer opportunities to load a circuit context without interfering with the signal transfer along the

channel. The channel is realised using the resources of the static region of the FPGA. How this is implemented in practice depends very much upon the properties of the target device and the level of vendor software provision. Priority is give to the Xilinx Virtex [6] family of FPGAs because of the level of support provided for the generation of the partial data-streams.

Figure 5.5 shows the layout of the channels when targeting the different members of the Virtex family of FPGAs. In Figure 5.5 (a), partial reconfiguration of $regions_{0,1}$ re-programmes the resources spanning the entire column of the Virtex and VirtexII families.

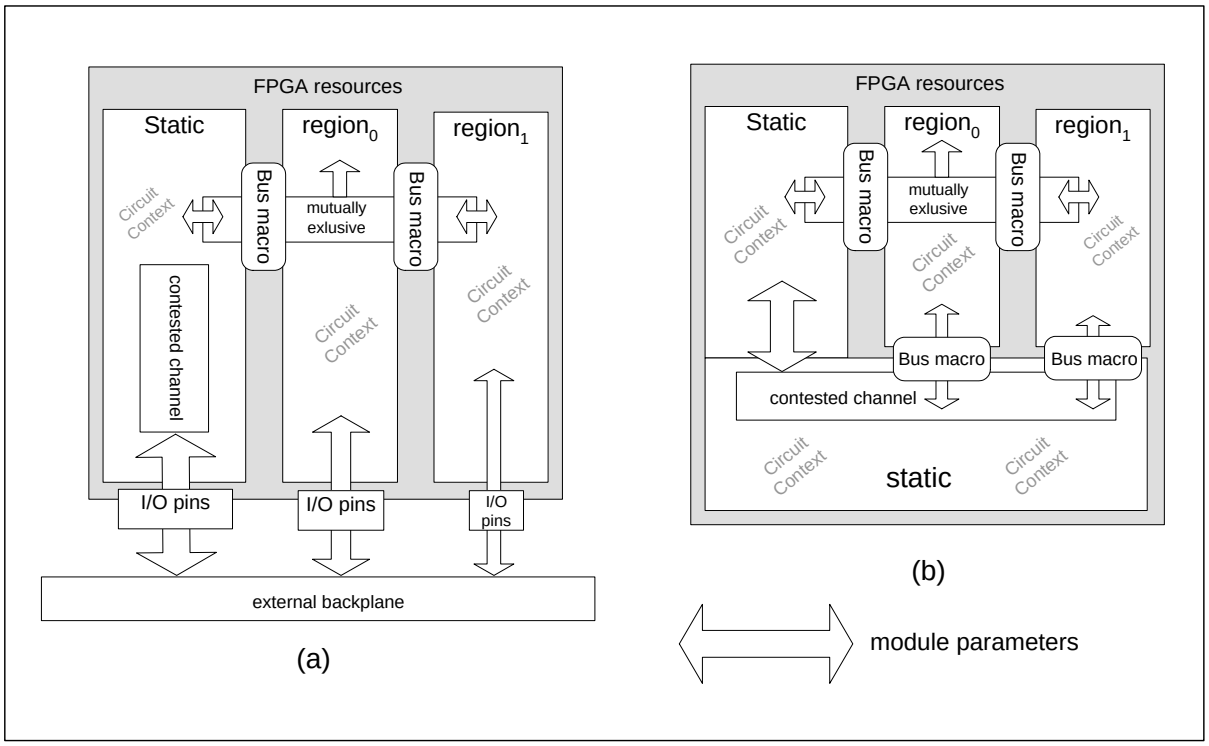


Figure 5.5: Device-specific channel implementation.

For the reasons described earlier, sub-modules with overlapping execution may read or write data through the reconfigurable regions of the architecture. This is accomplished through a number of fixed ports known as Bus Macros. Buffering the signals in this way ensures that the input/output routing resources of different circuit contexts have the same predictable physical interface before and after partial reconfiguration of the regions to which they are assigned.

Alternatively, the data transfers of sub-modules which cannot take place between partial reconfigurations are routed out through the external pins of the device, along an external

backplane and read once again through the externals pins, as input to the state region. Having passed into the region, the signals contest for access to the shared channel and one region is granted permission to write to it. Implementing the channel in the static region of the device, secures it and the three-state logic buffers which write to the channel against interruption from partial reconfiguration. Had the buffers been implemented in the reconfigurable regions, they would be reprogrammed with no guarantee of a high impedance state being held at their external output pins. This could lead to signal contention on the channel whenever a partial reconfiguration took place.

An alternative implementation of the channels is depicted in Figure 5.5 (b) and can be done so directly using tile-based FPGA resources that can be partially reconfigured in columns 16 CLBs in height. This allows reconfigurable or static regions to be stacked vertically and horizontally next to one another which can implement the shared channel on the FPGA, by isolating it from any partial reconfiguration.

A bus macro is utilised when a communication channel interfaces directly to a reconfigurable region. Figure 5.6 illustrates how the function of the bus macro is realised. Both circuits are used to pass data or control signals through the perimeter of a static or reconfigurable region. They are provided by Xilinx Inc. as part of its support for experimenting with the partial reconfiguration capability of its Virtex family of FPGAs.

Each circuit is replicated, albeit, only in behaviour, since the actual implementation uses different routing resources to pass the signals from one side to the other. Collectively they form a bus macro, a pre-routed component i.e. its structure is not subject to optimisation during RTL synthesis (it appears instantiated in the structural description of the architecture as a black box), nor is its layout altered during the placement and routing phase of circuit implementation. To do either, would be to disregard the concept behind its use.

The first generation of bus macro utilised the pair of three-state buffers adjacent to each Complex Logic Block (CLB in Xilinx terminology) available on the Virtex and Virtex II FPGAs. In total, eight buffers are employed to pass four signals in either direction. Although it represents a significant step forward in comparison with the previous support offered through JBits [48], it hinders the efficient implementation of non-trivial reconfigurable

circuits. This is due to the limited availability of the buffers at their designated positions on the FPGA, in comparison with other resources, such as the CLBs.

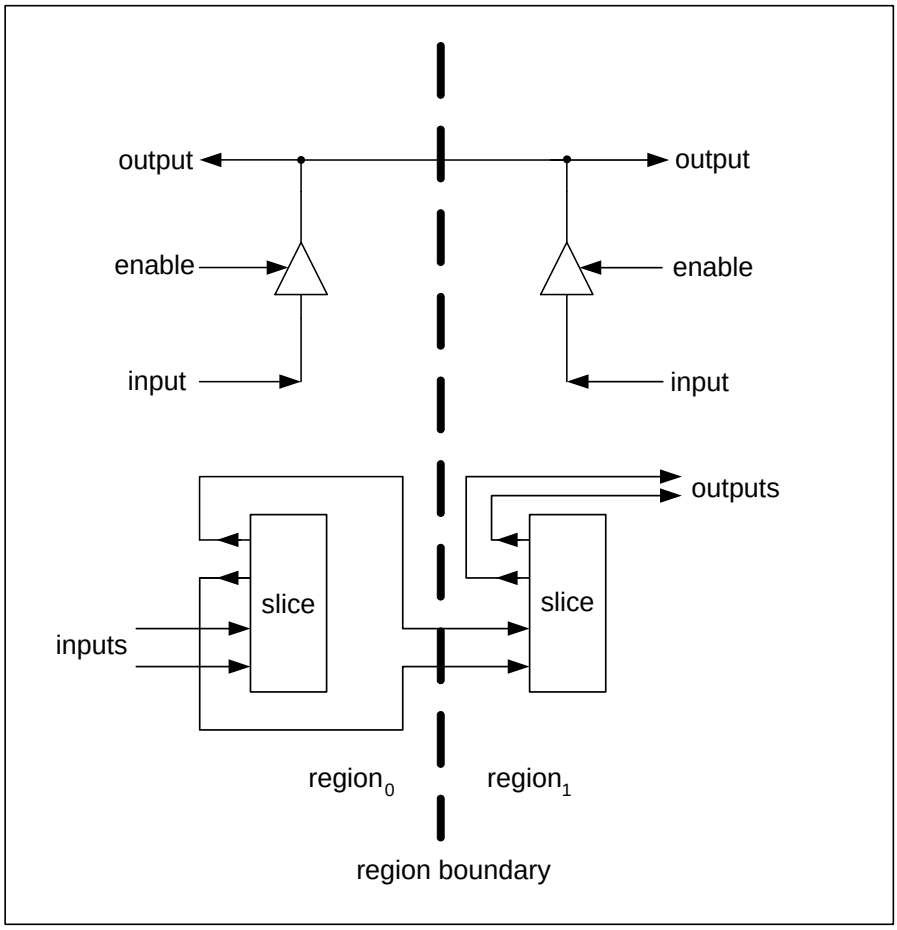


Figure 5.6: Bus Macros − bridging the reconfigurable divide.

For any given row, there are four parallel buffer output lines which are used to straddle the boundary. On either side of the boundary, the bus macro requires four CLB columns along the row where it is to be placed. Should the region exceed that number, any additional buffers on the same row cannot be utilised by adding bus macros because no more than four output lines can cut across the boundary at any given point along the row. This issue was addressed through the use of the second circuit shown, which as well as fulfilling the same function, does so without the placement restrictions. Utilising the CLB Slices enables the macro to be implemented using the basic primitives of the architecture, being numerous and well distributed throughout the FPGA assures that they are within easy reach of a plethora of routing resources. As a consequence, the number of signals which can be passed using a

single macro is doubled. Furthermore, as many as two additional macros can be positioned either side of the boundary, providing a higher density of channel signals (twenty four) per row of each static or reconfigurable region.

5.3.2 Channel Controller

The purpose of the temporal partitioning of Figure 5.7 is to illustrate the likely topology between pairs of calling and called sub-modules which the communication system must be able to accommodate. An arc connecting each module pair is representative of the bi-directional control and data signals that pass between them.

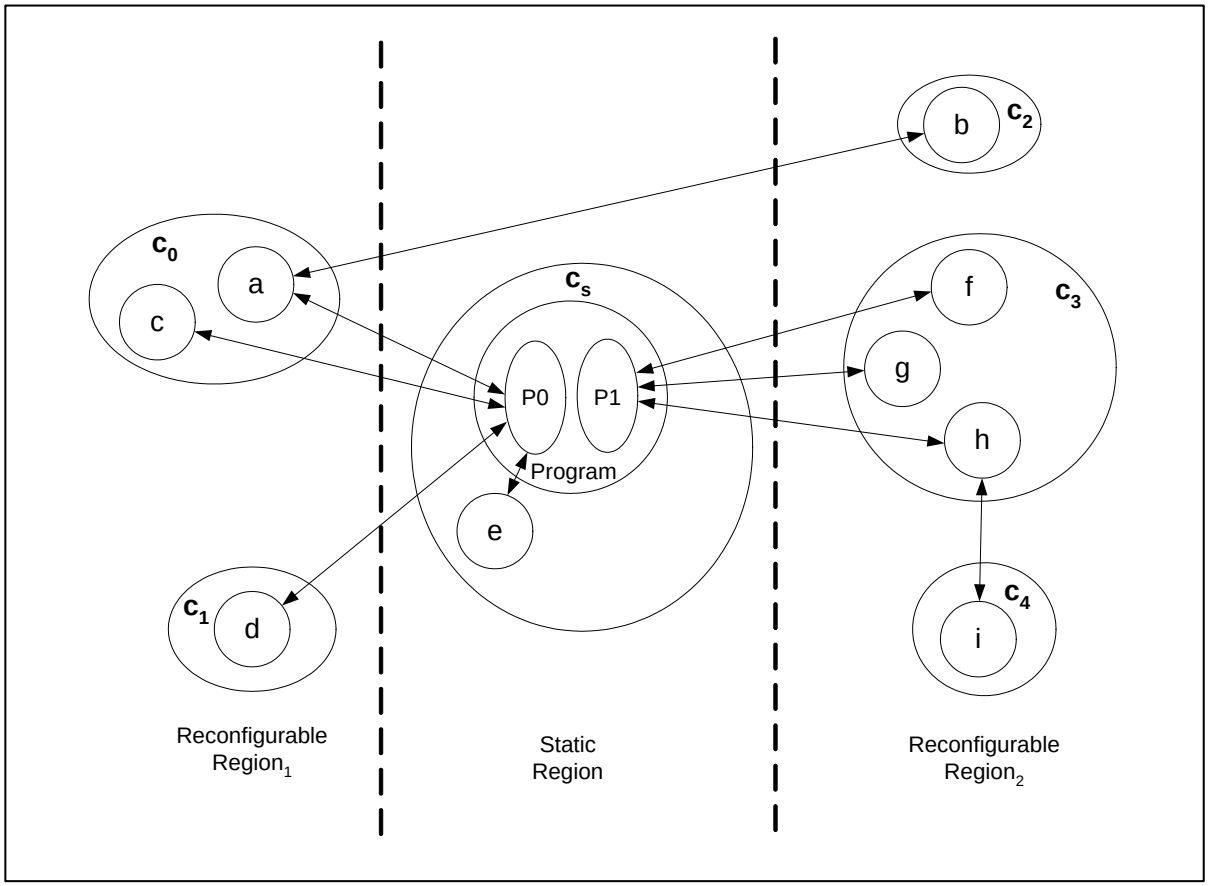


Figure 5.7: Typical sub-module partitioning topology.

Apart from the minimum partitioning of the program module and two sub-modules, there may be multiple or single modules in any circuit context i.e. 'C_0', 'C_1' respectively, of which a called module might also call another module in a different region to itself, such as the sub-

module pair (‘a’,’b’). Alternatively, the calling and called sub-modules might reside in the same circuit context, as is the case with the program and sub-module ‘e’.

Communication between two modules assigned to the same region but configured ‘on silicon’ at different times is also permitted. This is exemplified by the hierarchical pair of modules (‘h’,‘i’). Sub-module h, in common with the other modules is called by the program module. In order for it to execute the next module in the hierarchy, module ‘i’, its circuit context needs to be swapped with that of context ‘C_4’. Of course, the context switch must be repeated for sub-module ‘h’ to process the results generated by module i.

To implement the partitioning illustrated, it will be necessary for MOODS to customise the architecture utilising the global communication channel. This is due the assignment of module ‘b’ to a region not adjacent to its calling module. The multiple processes ‘P_0’, ‘P_1’ would necessitate a form of arbitration to enable the channel to be contested and therefore shared amongst the three regions. Had the partitioner placed module ‘b’ in either the same region as its calling module ‘a’, or in the static region, local communication channels might have been deployed.

In either case, the communication layer is reliant upon the use of a number of channel controllers to convert module control and data signals into specific channel operations. As depicted in Figure 5.8, the controller is not one component, but comprises a number of sub-systems which fulfil four main duties:

- The opening of a channel transaction – accomplished through the transmitter unit.
- Self-identification by the intended destination region – utilising the channel receiver unit.
- Channel arbitration to prevent its contention when there are multiple processes present in the circuit design.
- The bi-directional transfer of module parameters between the calling and called modules.
- The closure of the channel transaction – also performed by the transmitter unit.

Figure 5.8 illustrates how the components of the channel controller are connected to perform a channel transaction, i.e. to open or close a channel. As depicted, the function of the controller is separated into a transmitter and a receiver unit.

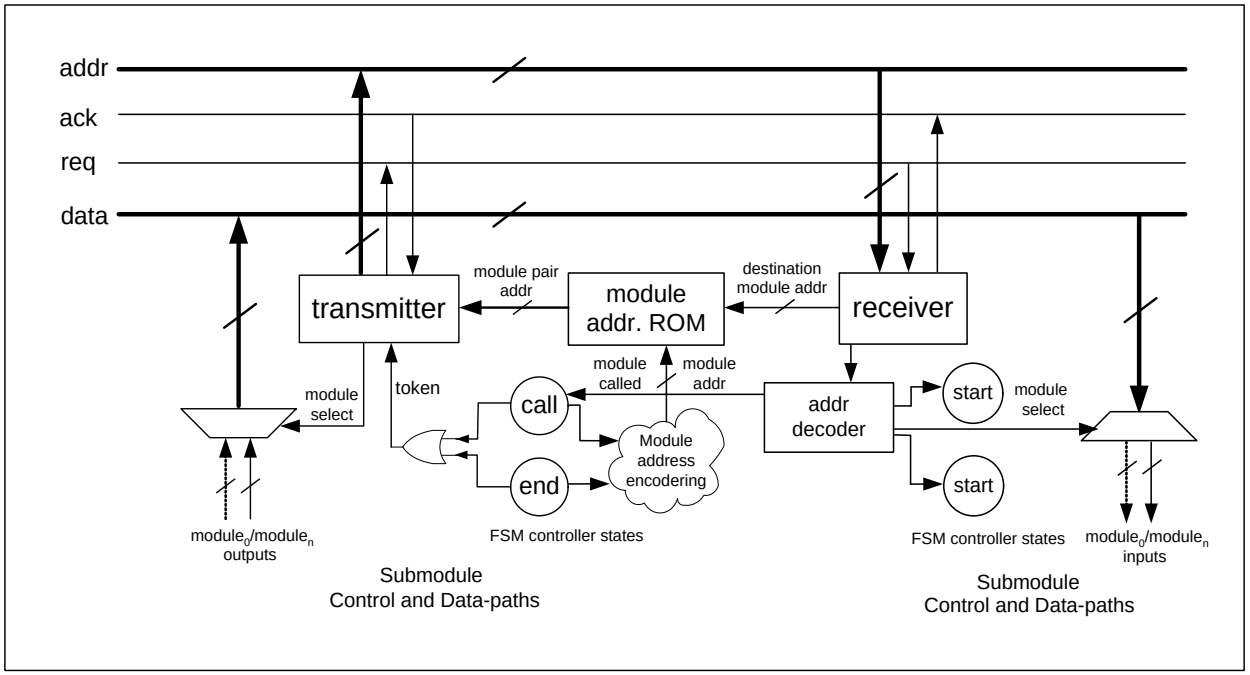


Figure 5.8: Channel controller subsystems utilised during a channel transaction.

A number of peripheral components are also utilised during the controller operation. As stated earlier, hierarchy between a pair of executing modules is expressed in terms of a calling and called (sub-ordinate) module. Initially, the transaction originates from within the calling module (program or sub-module), where a module call generates a token that is used to initiate a request to open a channel with the controller of the called sub-module.

Every sub-module execution is represented as a channel transaction between a calling and called module. Either module has a unique binary address in the architecture, associating it with the region and circuit context to which it was assigned to during partitioning. Recall that the scheduling of a module call is marked by the occurrence of a call node in the module's control path. When a pair of modules is assigned to the same context, the token from the calling node is used to directly activate the first node of the module being called. When the modules are assigned to separate contexts and regions, each token becomes a unique identifier for the module being called. In this way, all the call nodes in the same region can be strung

together to form a binary code, which can be encoded by the RTL synthesis tool and used as the address in memory for the transaction about to take place. The identifiers of all pairs of calling and called modules are synthesised as a ROM, customised for each instantiation of the channel cell in the architecture.

Communication between two regions is initiated when the transmitter cell receives a token driven by any of the call nodes in the circuit context. In accordance with the communication protocol, the transmitter requests that the channel be opened and places the address of the source and destination modules on the address lines, as well as writing the module parameters to the data lines of the channel. With reference to Figure 5.8, the module arguments are made available to the data channel multiplexors exactly one cycle prior to the token being issued by the call node. Of course, the module data cannot be written to the channel until the connection has been established. In order for this to take place, all receiver units periodically check for a channel transaction by reading the state of the request line. Any transition in its state draws their attention to the portion of address which identifies the intended destination region of the transaction.

Having correctly identified itself, the receiver stores the return address of the calling module before determining which sub-module should be connected to the channel, by decoding the remaining portion of the address. A token is passed to the start state of the called module, causing it to read from the data lines. The token is passed on to the remaining states of the module, as well as the receiver unit which commences the closure of the channel. It is achieved once again through a handshake between the two regions, this time initiated through an assertion of the acknowledge line of the channel, to which the transmitter responds by de-asserting its request line.

Execution of the calling module halts temporarily whilst the called module carries out its function. The parameters passed to the called sub-module and the results returned from it are processed as separate transactions of the channel, enabling the communication between another pair of modules to occur through the channel during the intermission between the transactions. This frees the channel to transport the data driven by other concurrently active sub-modules that have been granted access. When sub-module execution originates from within a single process, all module execution is consecutive i.e. mutually exclusive. Thus

channel traffic is not interrupted should a partial reconfiguration take place during the channel transactions, making the scheduling of context switches desirable during such periods.

Upon completion, the called module returns a token which through its local transmitter initiates a second opening of the channel; with the purpose of returning the results of its execution. The transactions described earlier are mirrored with the roles of the regions reversed, such that the region containing the called module requests the transmission of data. Following the handshake between the regions, the token is returned via the receiver to the calling module which up till now has been waiting for the results to be returned; it is now able to process the results in its data-path and continue execution of the process thread.

Another function of the channel infrastructure is the connection of a module's port signals to the buffer interface. This is a necessity for a calling and called module pair because it will generally share the interface with other modules in its circuit context. Recall that a module is referenced through a region and context address. Figure 5.9 depicts the architecture required to associate that address with a given module.

The control, address and data signals are passed in to the region from those regions on either side of it and are sent directly to the channel controller cell through their respective channels. Execution can occur between any number of sub-module hierarchies, although only a single pair may transfer data at any given time. This is done through the twin data channels which can be utilised as input or output, to enable the output of a given module to pass data to the input of another or vice versa. In this way, it is possible to send and receive data in a single transfer cycle of the channel.

Multiple module hierarchies may necessitate the activation of more than two regions which will require several sets of concurrently active module activation/completion control signals. As shown in Figure 5.9, the control signals utilised at the system layer of communication require an overhead of four (the clock requires no buffering) control signals for any simultaneously active pair of modules. This overhead is 'factored in' during partitioning where the benefit of splitting a module hierarchy of a given depth is weighed against the potential cost, to satisfy a given cost function criteria.

Execution between any pair of modules is managed through the channel controller, which is setup as a transmitter or receiver depending upon whether the module concerned is calling or being called by another. Prior to each module call, the controller is activated and refers to a local address ROM to physically connect the modules in the architecture.

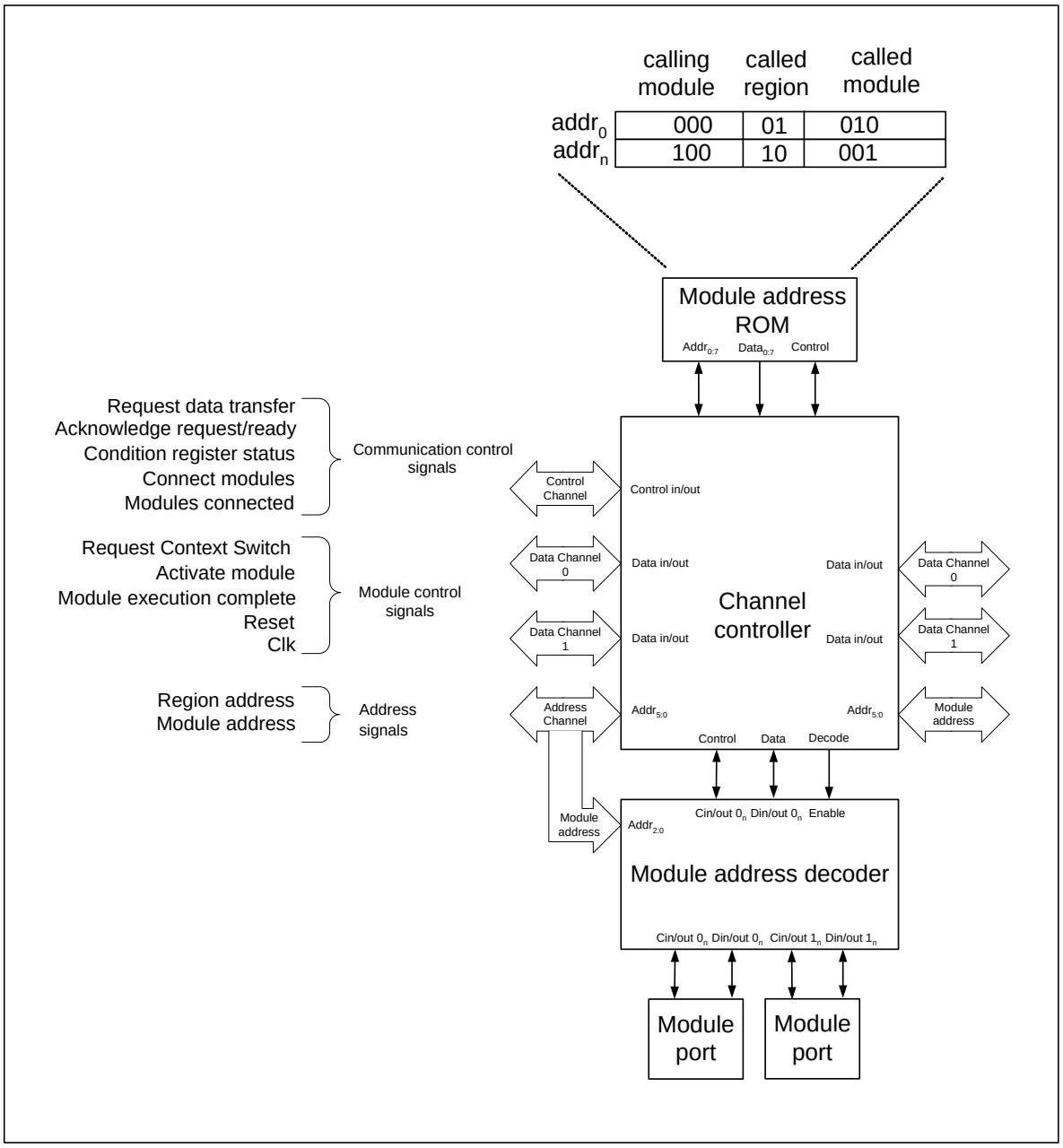


Figure 5.9: Module address decoding.

Once a connection is established the cell locates the address of the next pair of modules to be connected. Access to the control and data ports of a module is achieved by presenting the

module's address to the decoder. The origin of the address depends upon whether the controller transmits or receives a request to communicate with another region. For data transfer to occur between any pair of modules, both must be connected to the channel buffers in their region.

In the case of the transmitter, the module decoder is passed the address of the calling module from a local module address ROM. It also contains the address of the module being invoked and the region on which it resides. That address is sent to all regions, where having identified itself through its own communicator cell, the receiving region will decode the remaining part of the address and connect the called module to the control and data channels. It acknowledges a successful connection to the transmitting region and the execution of the module pair may now proceed at the system layer. These events take place through a handshaking protocol between the communicators which will be formalised in due course.

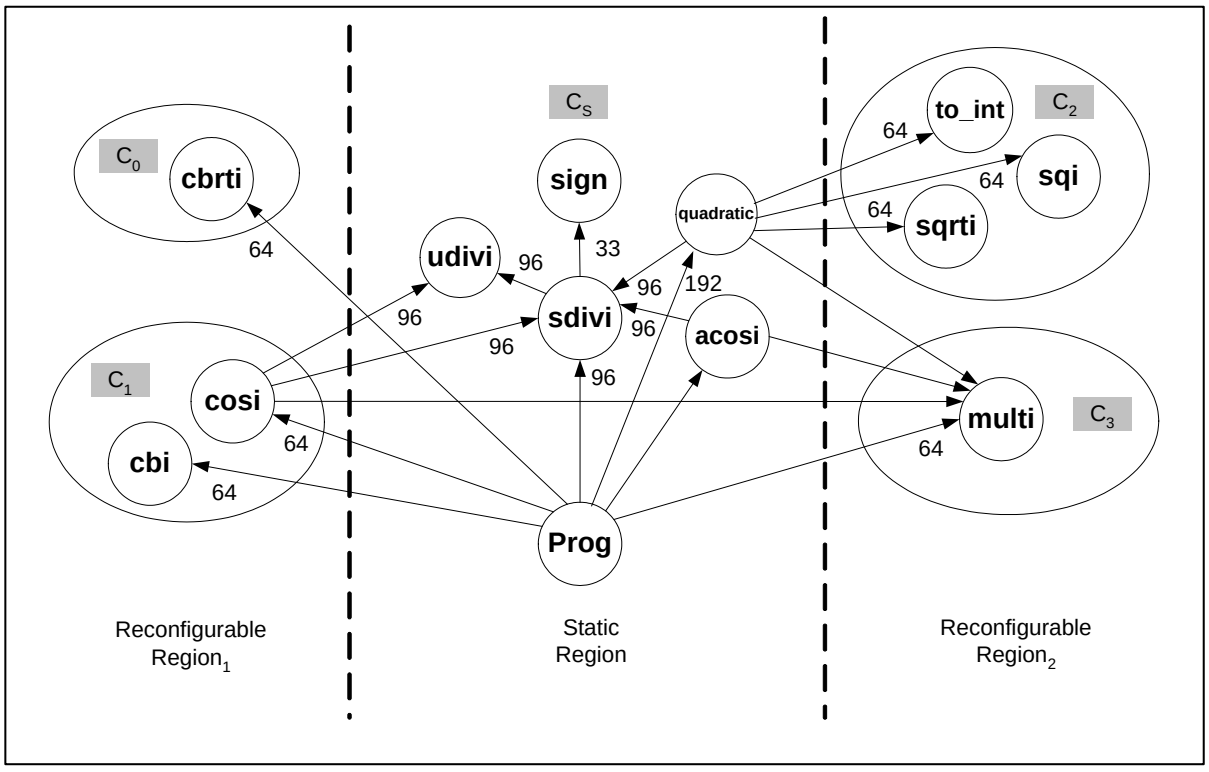


Figure 5.10: A temporally partitioned quartic equation solver.

Mapping each module call to an address in memory effectively encodes the execution sequence for each design. The demand upon memory space is small since procedures and

functions are referenced and not individual operations. For example, the quartic equation solver of Figure 5.10 (which by no means is a small circuit) comprises 126 module calls in the entire design.

The size of an address ROM will depend upon the number of module calls and the format used to encode them. Each address reflects the partitioning itself, in terms of the number of regions and modules assigned to them. Consider the partitioning of the quartic equation solver whose module execution sequence is shown in Figure 5.11: it comprises 3 regions on which a maximum of 4 modules require individual selection; although the static region contains 6 modules, module ‘sign’ requires no address since it is executed within the region and module ‘quadratic’ is not called within the segment of module execution depicted. In total this requires 4 bits to address any module, 2 bits for the region and a further 2 for selecting a module within it. Figure 5.12 illustrates how each module address can be mapped to the address ROM.

The address of every module is tabulated in Figure 5.12 (a): it identifies the region of each module, be it the static or reconfigurable regions (encoded as 00, 01, 10 respectively), as well as its identification within the circuit context. When a module is the sole member of its circuit context, as is the case for modules ‘cbrti’ and ‘multi’, no individual module identification is required within each of their respective circuit contexts ‘C_0’ and ‘C_3’. Figure 5.12 (b) illustrates how the addresses are mapped to represent the relationships between the modules (Figure 5.10) and the order in which they are executed (Figure 5.11).

Each row in the map corresponds to a byte of ROM. It can represent the connection between a pair of modules or direct the reading of the next through a set of 4 encoded commands. A command instructs the controller cell’s state machine on how to traverse the memory locations. Note that this requires an additional address bit to enable the addressing of up to any 4 modules or commands.

A given byte is composed of the encoded calling module, destination region and the called module. There are 4 calling modules in the section of graph depicted in Figure 5.11: the top level program module whose behaviour is encapsulated by the graph and 3 sub-module hierarchies invoked by modules ‘cosi’, ‘acosi’ and ‘sdivi’.

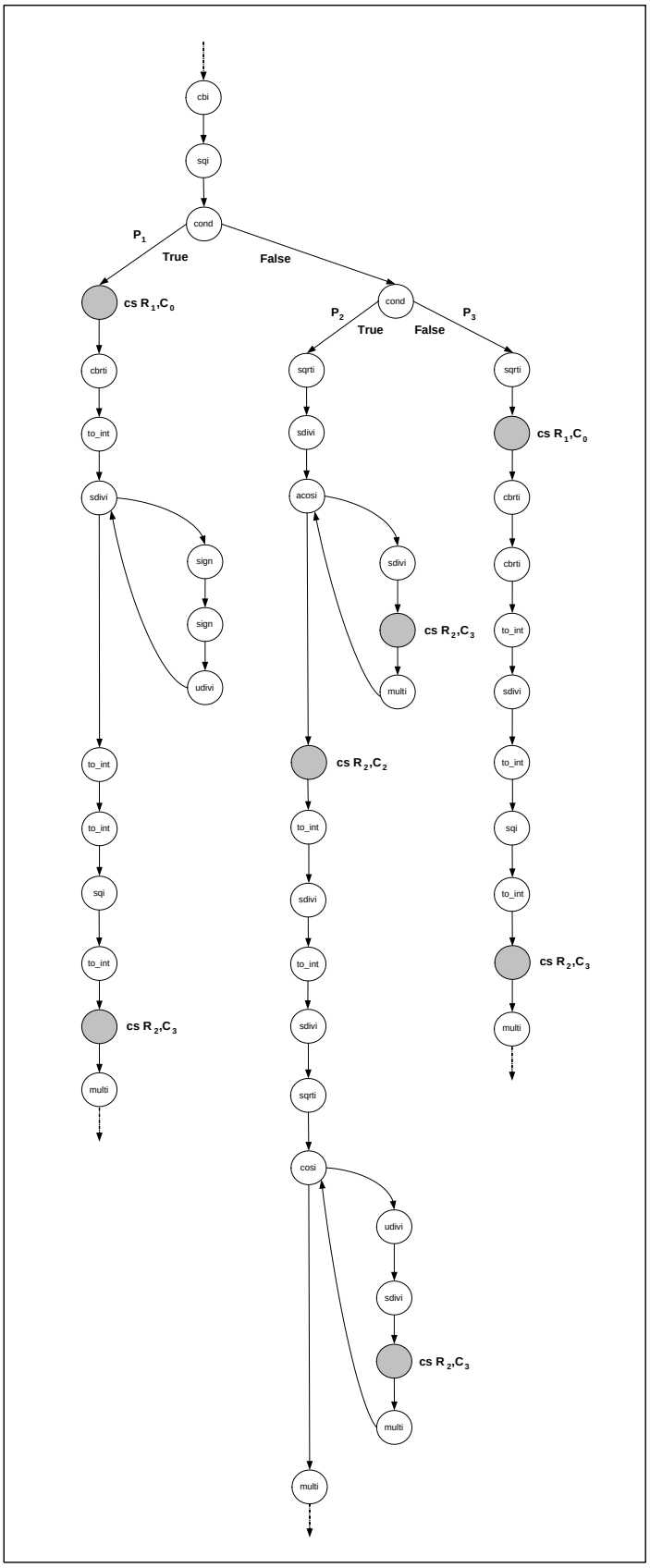


Figure 5.11: Module execution paths of the quartic equation solver.

Module/ context	Region address	Module address
cbrti / C_0	01	--
cosi / C_1	01	00
cbi / C_1	01	01
prog / C_s	00	00
sdivi / C_s	00	01
udivi / C_s	00	10
acosi / C_s	00	11
quadratic/C_s	n/a	n/a
sign / C_s	n/a	n/a
to_int / C_2	10	00
sqi / C_2	10	01
sqrti / C_2	10	10
multi / C_3	10	--

(a)

	Calling module	Receiving region	Called module	module/context pair	
$addr_0$	000	01	001	prog,cbi / C_s,C_1	
$addr_1$	# no decode	10	001	prog,sqi / C_s,C_2	
$addr_2$	# condition	$addr_{11}$			
$addr_3$	# condition	$addr_{17}$			
$addr_4$	# no decode	# decode module	010	prog,sqrti / C_s,C_2	P_3
$addr_5$	# no decode	01	---	prog,cbrti / C_s,C_0	
$addr_6$	# no decode	10	000	prog,to_int / C_s,C_2	
$addr_7$	# no decode	# decode module	001	prog,sqi / C_s,C_2	
$addr_8$	# no decode	# decode module	000	prog,to_int / C_s,C_2	
$addr_9$	# no decode	# decode module	---	prog,multi / C_s,C_3	
$addr_{10}$	# begin	$addr_0$			
$addr_{11}$	# no decode	01	---	prog,cbrti / C_s,C_0	P_1
$addr_{12}$	# no decode	10	000	prog,to_int / C_s,C_2	
$addr_{13}$	# no decode	# decode module	001	prog,sqi / C_s,C_2	
$addr_{14}$	# no decode	# decode module	000	prog,to_int / C_s,C_2	
$addr_{15}$	# no decode	# decode module	---	prog,multi / C_s,C_3	
$addr_{16}$	#begin	$addr_0$			
$addr_{17}$	# no decode	# decode module	010	prog,sqrti / C_s,C_2	P_2
$addr_{18}$	011	# decode module	---	acosi,multi / C_s,C_3	
$addr_{19}$	000	# decode module	000	prog,to_int / C_s,C_2	
$addr_{20}$	# no decode	# decode module	010	prog,sqrti / C_s,C_2	
$addr_{21}$	# no decode	01	000	prog,cosi / C_s,C_1	
$addr_{22}$	000	00	010	cosi,udivi / C_1,C_s	
$addr_{23}$	# no decode	# decode module	001	cosi,sdivi / C_1,C_s	
$addr_{24}$	# no decode	10	---	cosi,multi / C_1,C_3	
$addr_{25}$	000	# decode module	---	prog,multi / C_s,C_3	
$addr_{26}$	# begin	$addr_0$			

(b)

Figure 5.12: Memory maps of module address ROMS for the quartic equation solver.

With the exception of the latter, the execution of other modules is encoded with an address ROM which forms part of their circuit context. Module ‘sdivi’ invokes the execution of modules within its circuit context and does not require an address to connect to them.

Shown alongside each memory map is the pair of modules referenced by the byte, their associated contexts and a reference in the graph. For example, the first byte would instruct the controller cell on the static region to connect the program module (decode calling module address 000) and write the remaining 5 bits onto the address channel. The controller cell of the target region ($region_1$) would then identify itself as such (region address 01) and decode the module address (001) to connect module ‘cbi’ to the data and control channel buffers.

The next occurrence of a controller command ‘no decode’ saves a decoding cycle by not enabling the address decoder when the calling module appears consecutively. Its value is evident in Figure 5.12 (b), where the program module dominates sub-module execution for all but two calls by modules ‘cosi’ and ‘acosi’. A complementary command ‘decode module’ prevents the controller cell of the receiving region from unnecessary identification of the region address: this occurs when the module called is on the same region as the module previously executed. Incidentally, consecutive calls to the same module are not represented in the ROM. This also applies to consecutive connections between a pair of communicating modules. An example is on path P_1 between the module pair (‘program’, ‘to_int’). Before the first call to module ‘to_int’, the address of both modules would have been set up in either region to enable their connection. Upon completion of their execution, the ‘program’ module calls module ‘sdivi’. As this call is internal to the context, module ‘to_int’ remains connected to the interface buffers and so does not require any region or address decoding to facilitate its next execution.

As the control graph of the program module illustrates, for all but the simplest of circuits there exists some form of conditional control. The order of sub-module execution will change depending upon the branch taken at any conditional point in the module control flow. The motivation for the internal commands is to traverse the memory in a way which reflects the sequence of module calls of any calling module on the same region as the ROM.

The first example of this is the ‘condition’ command. Its location in the sequence of memory addresses corresponds to a diversion of the path taken by the module currently active on the

region. A dedicated ‘Condition’ register in the cell reflects the outcome of that decision which the controller is required to read and act upon. By default, the data bits following the ‘command’ instruction form the address of the next location in memory, which corresponds to the sequence of module calls taken in the event the condition evaluating to ‘false’. Had the result of the condition been ‘true’, the controller would have been instructed to read the operand of the command instruction, taking it to an alternate sequence of addresses. In this way, each path of the quartic equation solver can be referenced in memory.

At the end of each path, the ‘begin’ command requests that the controller address counter be loaded with the next 5 bits which will direct it to the start of the memory and the first module address byte. Any of the 27 bytes contained in the ROM are located through a 5 bit address counter. In practice this would address the lower bits of a larger ROM whilst the upper bits remained fixed.

As each module address ROM is part of a circuit context, it is reconfigured along with the circuit context and so the resources utilised in its creation will be re-used in the formation of the next context. To further reduce the area overhead associated with the ROMs, it would be prudent to utilise where possible the dedicated BlockRAMS [6] which form part of the Virtex architecture. They may be regarded as ‘free’, in terms of area since they are incorporated as part of the architecture and exist whether utilised or not. Their size and distribution varies within the Vertex family of devices. A Virtex2 FPGA features columns of 18Kb blocks interleaved between the CLB columns. Each blockRAM is also reconfigurable ‘on the fly’ which makes them suitable for incorporation as part of a reconfigurable region. A single blockRAM is capable of storing 2304 module address references, assuming that a byte is used to identify the pair of modules being connected. A mid-range device has a further 31 of these so storage capacity is not an issue in the implementation of the address ROMs.

Figure 5.13 depicts the states and their transitions which together encompass the behaviour of the Controller cell. In essence, it implements the ‘Communication layer’ of the protocol. Depending upon the partitioning, a controller cell may be a transmitter and/or receiver for its region. The cell buffers all module parameters passed and received between any pair of executing modules. In doing so, it guarantees that communication between the modules occurs only once a point-to-point connection has been established on the architecture.

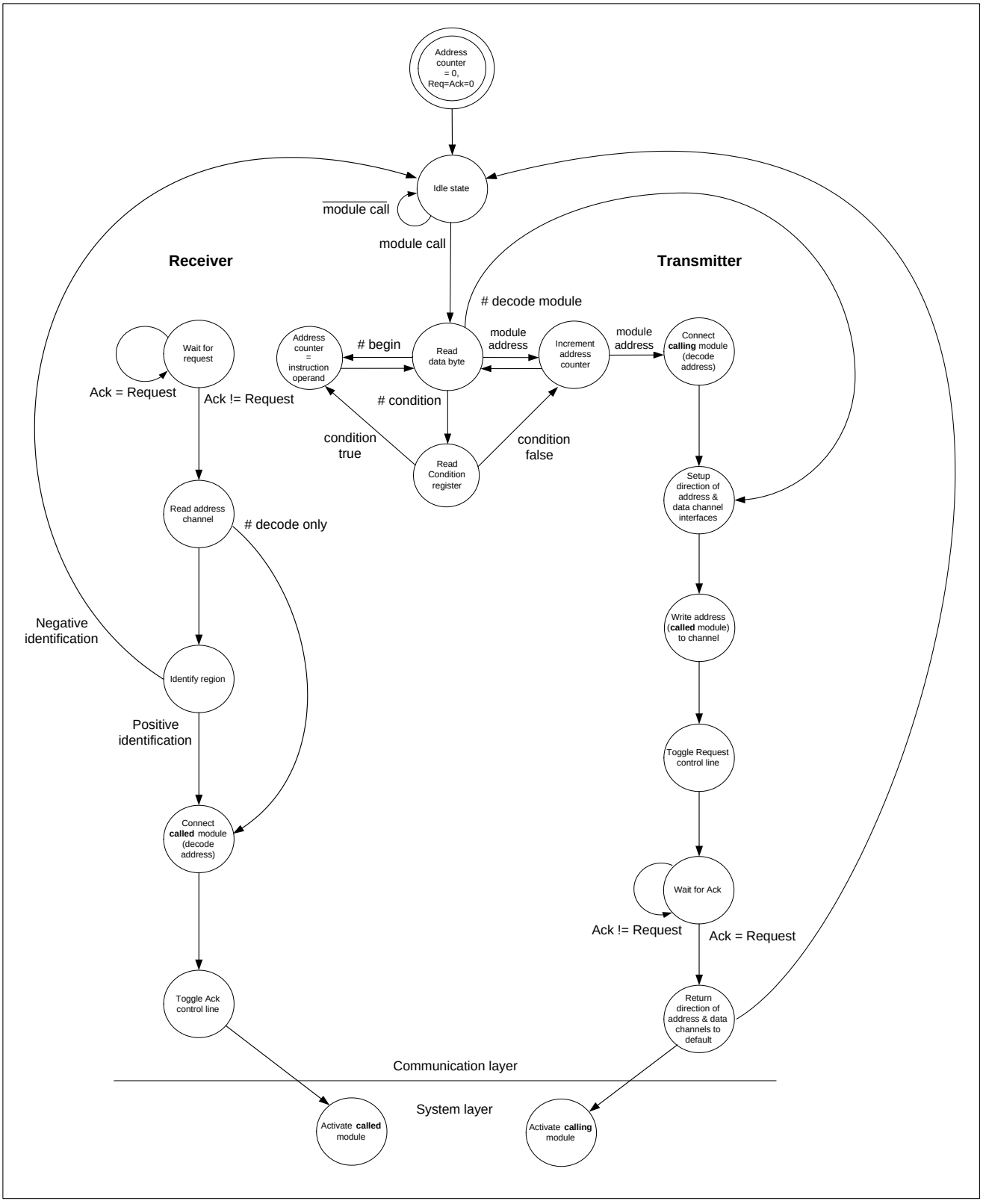


Figure 5.13: Communication layer protocol and usage.

Upon ‘power-up’ of the circuit context, the cell initialises the address counter and the semaphore to zero. It then waits until activated during the ‘System layer’. Where it is activated (transmit/receive inputs) will determine whether its role is in the form of a transmitter or receiver in the region. The inputs are determined from the module relationships inherent to the control graphs. Essentially, each module call from within a module is associated with the transmitter side of the communicator and vice versa.

Execution of the transmitter proceeds by reading the data byte of the local module address ROM referenced through the recently initialised address counter. Simultaneously, the other regions monitor the status of the semaphore, awaiting a transmission. Whether it takes place or not depends upon the meaning of the data byte, for instance, its purpose may be to guide the transmitter at a point of divergence in the execution path of the active module. In the event of the data byte representing a connection, the transmitter proceeds to connect the calling module (identified within the format of the data byte).

Next the address channel is set to broadcast the remaining bits of the byte which will address the called module and the region on which it resides. Its direction, along with that of the data channels is determined by the function of the controller cell. The interface of a transmitting region is set to output the address onto the channel. On the other end of the address channel, the interface must input the address to the receiver. Likewise, the direction of each region’s interface to the data and control channels is also set, to enable the output of the transmitting region to connect to the inputs of the receiving region and vice versa.

Once the interface of the transmitter is configured, the region initiates transmission by toggling the state of the request line. This triggers a chain of events at each receiving region during which time the transmitter waits for a response. It comes when a single region identifies itself, although this stage may be bypassed (‘# decode only’ command) if the region featured in the previous connection.

The final stage is to decode the target module’s address and connect the module to the data and control channels, thus completing the coupling of the regions. Contact is signalled through the receiver toggling the ‘Ack’ line which places it in a state of readiness awaiting the next potential transmission.

The transmitter responds by returning the direction of all channel interfaces to a default mode. This configures the region to receive an incoming address − it may become the receiver in the next inter-region connection.

The interface to the control channel also reverts to an input direction e.g. the interface buffers for the semaphore are reversed to receiving rather than requesting transmission. Similarly, the orientation of the pair of input/output data channels is fixed for all receiving regions. For multiple reconfigurable regions, there will be more than one receiving region during the establishment of a connection and this places the onus on the transmitting region to alter the direction of its interface to enable an output to drive an input on the receiving end etc. With the connection now established, control is once more returned by the controller cell to the system layer where the execution of the pair of modules will now take place.

5.4 Device-level Architecture

When the structural description is generated, the activation of the reconfiguration controller is expressed in terms of boolean equations that link the control node (now control state) directly to its enable input. The backbone of the architecture is implemented at this layer. With reference to Figure 5.14, it comprises the ‘Reconfiguration Controller’ and the protocol for its deployment in the architecture.

A key facilitator of reconfiguration is the reconfiguration controller, which like the other structures is described in the cell library as an RTL VHDL component to be instantiated during the creation of the structural circuit description. Unlike the others, it is only instantiated once, as every occurrence of a ‘ContextSwitch’ instruction is bound to the controller. In the structural description of the data-path, it appears as a component whose execution can overlap that of any other component in the data-path: it achieves this at the device level by exploiting the partial reconfigurability of the Virtex configuration memory. Respecting the precedence of module execution in the scheduling of each context switch ensures that it reconfigures only the circuitry of reconfigurable regions which have completed their execution.

A novelty of the reconfiguration controller is that it enables a Virtex FPGA to partially reconfigure itself using the programmable logic resources of the device. It does not require an external controller in the form of a personnel computer, on board or embedded microprocessor to perform the reconfiguration.

Figure 5.14 (a) illustrates the interface between the controller, the Virtex ‘Configuration Port’ and the external ROMs containing the data-streams. Shown alongside it is the protocol necessary to perform the reconfiguration process. Upon power-up, the FPGA is configured in the ‘Master SelectMap’ [6] mode, where it controls the loading of the full data-stream which configures the architecture i.e. the static and dynamic regions, along with the communication channels linking them.

On completion of the full configuration process, control is passed to the programmable logic resources of the device, where the reconfiguration controller within the static region changes the FPGA configuration mode to ‘Slave SelectMap’ [6]. Ordinarily, this enables an external device such as a microprocessor to manage its configuration. However, in this instance, it is the design being executed on the programmable logic − namely the reconfiguration controller which oversees the self-reconfiguration process.

Self-reconfiguration is a three stage process: Firstly, the start address of the partial data-stream is passed to the controller prior to its activation in the control graph of the program module or subordinate modules. Activation occurs during specific states in the control graph, predetermined during optimisation under the guidance of the temporal partitioner.

On receipt of the control token, the controller is initialised with the start address of the configuration data-stream. Upon each configuration cycle, a byte is passed from the external ROM where it is stored, through the controller and on to the internal configuration bus of the FPGA. Providing the FPGA returns no error flag and has processed the current byte, the next byte is fetched. This process continues until the end of the data-stream is reached.

Unlike a full reconfiguration of the Virtex FPGA, there is no dedicated flag to indicate the completion of configuration. The solution adopted is to detect the ‘desync’ sequence of bytes

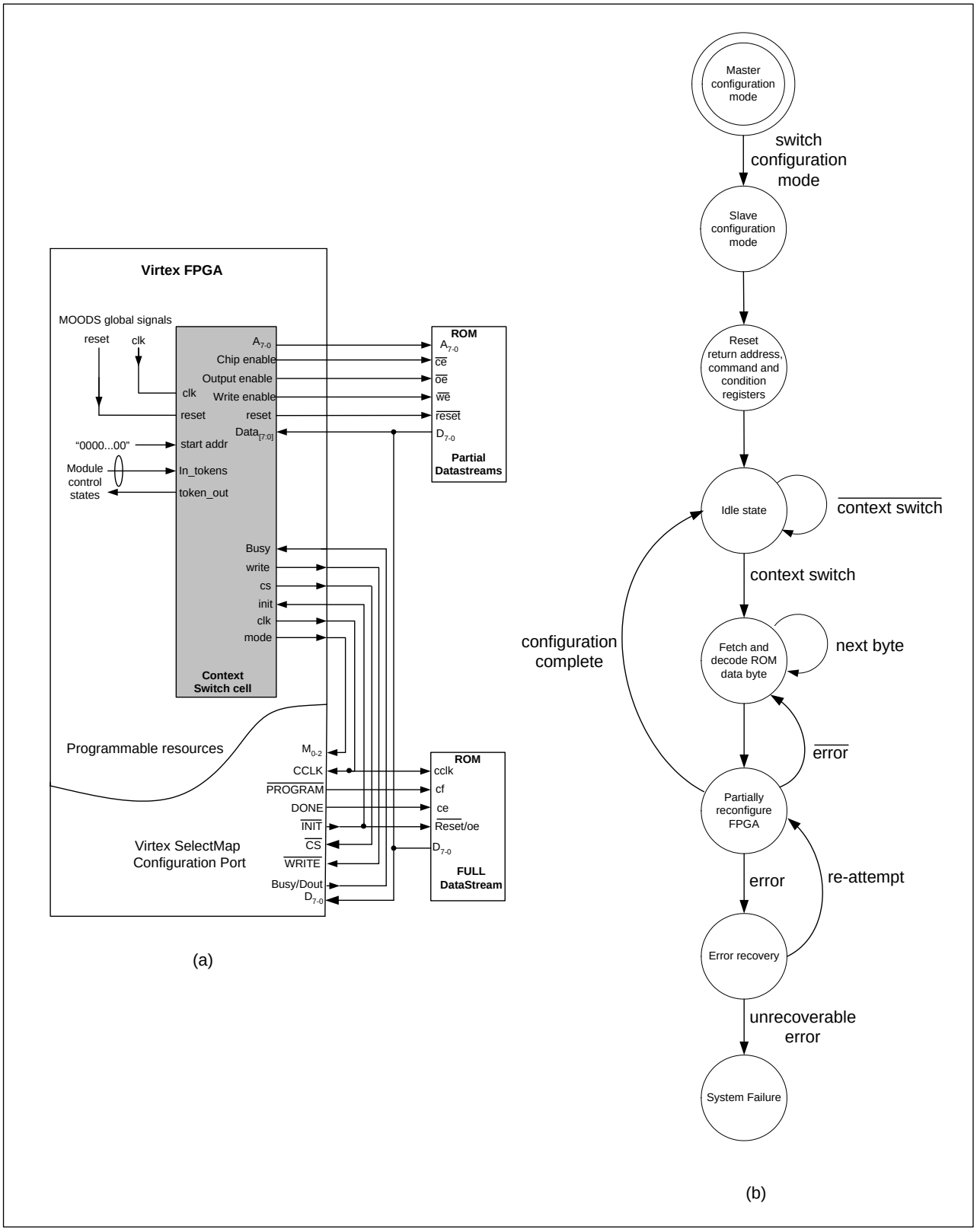


Figure 5.14: Reconfiguration controller and protocol.

at the end of every partial data-stream. When encountered by the controller and providing that no errors were returned by the FPGA it can be used to indicate a successful reconfiguration. The control of the configuration port is then relinquished and it is placed in a state of high impedance along with the external ROMs.

The final function of the controller is to pass on the control token and in doing so ensuring that the dynamic context associated with the partial data-stream is loaded into its designated reconfigurable region prior to its execution in the control graph.

The FPGA remains in the slave selectMap mode of operation for the duration of the design being executed until the power is cycled, whereupon the mode of the FPGA is set to master selectMap, the full data-stream is loaded and the process outlined above is repeated.

It would be rather cumbersome to directly tag a module call with the physical address of the module configuration data-stream within the control path. A neater solution is to implement the configuration addressing within the external memory itself. This effectively makes the reconfiguration controller micro-coded. It may be extended without too much difficulty to alter the scheduling for the context switching of the temporal partitions on-the-fly. The motivation for doing so would be to fine tune a bad compile-time partitioning using information only obtainable at run-time. At present, this approach is supported by the multiple binding of sub-modules to more than one location, although its present purpose is in the generation of temporal partitions which require fewer context switches.

Figure 5.15 illustrates the organisation of an external memory necessary to store the data-streams for the quartic equation solver. The memory locations can be conceptually divided into two halves. The first effectively encodes each sub-module configuration sequence. The rationale for doing so is to enable the reconfiguration controller to select the right data-stream without being passed the actual address from within a module. Instead the address of each data-stream is held in memory along with the data-streams themselves.

In much the same way as the end of a data-stream is represented by a unique sequence of bytes recognised by the reconfiguration controller state machine, the order of each data-stream is marked by a ‘fetch’ command byte. It directs the state machine to read the next byte which will point it to the address in memory of the beginning of the relevant data-stream.

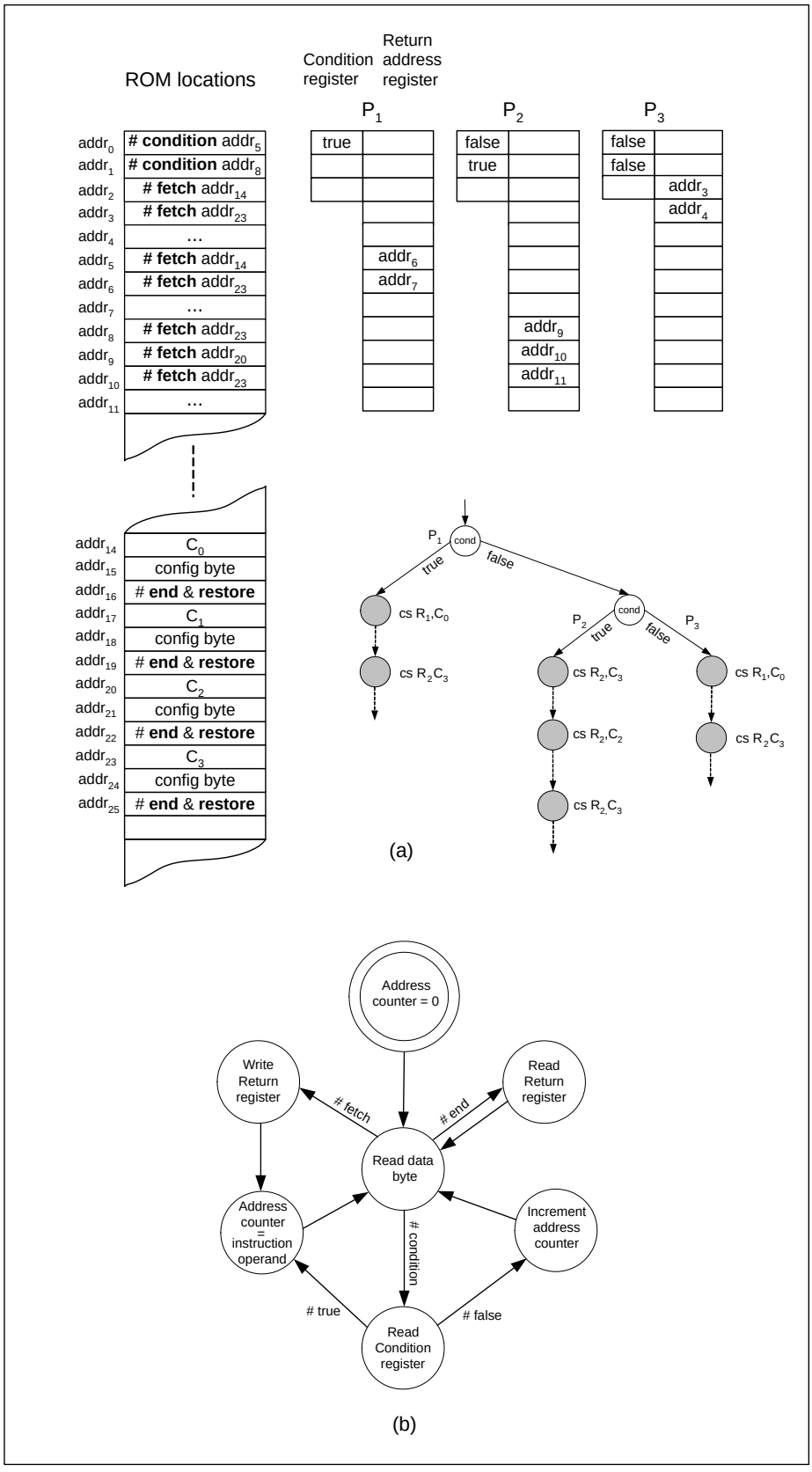


Figure 5.15: The organisation of configuration data-streams in external memory.

Before performing this task, the state machine writes the address of the command byte into a dedicated ‘Return address’ register. Once the loading of the data-stream is complete, it enables the state machine to continue processing the fetch command sequence.

Upon a successful configuration, the address counter is incremented to point to the next command byte. In the event of a configuration failure, the address provides an opportunity to re-load the data-stream or acts as a base from which the last known good configuration can be determined and loaded. Its application in the error recovery process is discussed in due course. The second half of the memory contains the configuration data-streams to which the fetch commands refer. Each data-stream may be referenced several times during the execution of the circuit. This is more economical in terms of memory space than duplicating the data-streams at every reference.

A circuit is highly likely to have some form of conditional control inherent to its function. To ensure that the data-streams available to the reconfiguration controller match the order of modules executed on the current path, any choice made in the control path is mirrored at the same points along the configuration command sequence in memory. This is achieved by embedding a ‘condition’ command at the appropriate place in the sequence. It informs the reconfiguration controller to examine the contents of a condition register to determine which series of command bytes it should process next. In doing so, it ensures that the configuration sequence will match the order of module calls regardless of which branch is taken. This will of course depend upon the outcome of the condition itself which would be fed from a functional unit in the module’s data-path, for instance the output of a comparator unit. Should that output be ‘false’, the state machine responds by loading the byte immediately after the condition command or at an address given by its operand when the outcome of the condition is ‘true’. In either case, the byte read by the controller is the address of the next consecutive set of data-streams whose modules will be encountered along the control path taken. The condition register is written to directly from the control channel. A dedicated channel signal is required, as the reconfiguration controller is only present in the static region and not local to the module in which the condition is encountered.

Following this format enables the memory to be organised in a way which corresponds to all module execution paths through the circuit. Consider once again, the control paths of the

quartic equation solver (simplified in Figure 5.15 (a) to depict just the sequence of data-streams invoked along each path). Each path is encoded in the memory through the controller commands (pre-fixed with '#'). Assuming that each memory location can contain both a command and/or an address byte depending upon its format and that a byte is sufficient to address all of its locations.

Also shown in Figure 5.15 (b) is a state transition diagram which depicts how the reconfiguration controller will pass through the memory. It will be utilised to briefly show how the control path P_2 is tracked within memory.

Assuming that the address counter is initialised to '$addr_0$', the first command byte '# condition' directs the controller to read the status of the 'Condition' register. It reflects the state of the first condition depicted in the control graph − which to direct it along the path 'P_2' will be assumed to be 'false'. The controller responds by incrementing the address, taking it to the second conditional command byte at memory address 'add_2'. Once again, the path diverges into two, only this time a 'true' register value adds an offset of 6 memory locations to the address counter, pointing it to the '# fetch' command at memory location '$addr_8$'. The command byte then directs the controller to the first configuration byte of the data-stream 'C_3' and to self-reconfiguration of the device, after it has written the location of the next module address '$addr_9$' to the 'Return address' register.

Once the configuration of the FPGA is complete and with no errors encountered, the return address is incremented to reach the next fetch command at location 'add_9'. It references the configuration data-stream 'C_2' and the process described above is repeated until no more configurations remain; the controller sets the address counter to point to the memory at address byte 'add_{11}' and the controller awaits the execution of the next sequence of subprogram modules.

5.5 Implementation in MOODS

Before describing how resource binding has been used to achieve circuit partitioning in MOODS, it is useful to repeat a number of salient points that were identified and described in

earlier chapters and which have subsequently influenced the approach described in this chapter.

The first is the use of MOODS for synthesising reconfigurable logic circuits. A particular strength in the Simulated Annealing approach offered by MOODS is the ability to quantify the impact of changes to the circuit structure with regard to multiple and often contradictory constraints on its properties. The reader will recall from Chapter 4, that the use of reconfigurable resources necessitates the measurement of many aspects of a circuit's structure, several of which are also inter-dependent.

An obvious example of this is the trade-off between a resource reduction and the reconfiguration delay required to permit the sharing of reconfigurable resources at different times. A more subtle trade-off would be in deciding the size and number of the partitions required for a given user resource target. Consider what might happen when a subroutine cannot fit in any of the available temporal partitions and making it reconfigurable would bring the size of the circuit closer to meeting the resource target.

An existing resource could be reconfigured to form a new temporal partition, at the expense of an increase in the overall reconfiguration delay. Alternatively, the unused logic of two adjacent temporal partitions could be merged to form a partition large enough to achieve the same effect without any increase in reconfiguration delay, the result of combining their separate delays. Although it would be a less costly approach concerning the reconfiguration delay, the current vendor design methodology [122] for partial reconfiguration of programmable resources requires fixed sizes for all resources; therefore, any future re-use of the combined resource during partitioning would result in a temporal partition of the combined size. This would require a cost function to accept a larger reconfiguration delay, a scenario that would be less likely to happen than in the case of accepting a smaller delay associated with a smaller resource. Thus there is a trade-off between fewer larger partitions or many smaller ones.

This example excluded the use of existing instruction scheduling and allocation techniques. For example, space for the functional unit could have been created by sharing functional units in any of the temporal partitions. This would have to occur between units that are not on the critical path, in order not to prohibit the scheduling from otherwise reducing its delay by

executing their instructions in parallel. Through its implementation as a transform, temporal partitioning can be applied in the same synthesis session as the existing operation-level transforms, enabling their interaction to guide the cost function to meeting the user-specified optimisation goals.

For any resource and delay constraint there are likely to be many different ways in the instructions of a behavioural specification can be scheduled, allocated and partitioned to reconfigurable resources. As the literature survey concluded, existing heuristic approaches to temporal partitioning, concentrate on extending existing spatial partitioning or HLS scheduling techniques, at a loss of generality. Any additional constraint, such as one that would accompany a new device implementation methodology or the addition of a new metric would likely require a different heuristic to be developed. Such changes are not an issue when using Simulated Annealing. However, the drawback in not specifying how the partitioning is achieved is the greater time taken to search the design space, in comparison with other methods [66].

A Behavioural partitioning can also reflect well in a cost function due to the likelihood of there being more data-dependent operations inside a subroutine than outside it, in the form of parameters passed from a calling subroutine or process, as described earlier in the chapter. Data-dependencies broken by partitioning, such as a variable written to in one partition and read a later by another, require additional components to transport their signals in between reconfigurable resources. Any reduction of additional resources may tip the balance of a cost function toward accepting the prospective partition.

The final point relates to how the partial reconfiguration of resources is represented in MOODS HLS. The author has taken a pragmatic approach which uses the same hardware for theoretical and practical implementations of partial reconfiguration, despite the fact that current FPGAs require configuration cycles in the order of several magnitudes greater than the number of cycles used to execute the resources being configured. This approach implements reconfiguration in a different clock domain to that of the user’s design. During synthesis, an equivalent configuration time is modelled during the scheduling of reconfiguration which enables MOODS to overlap the reconfiguration of a resource with the execution of another. Crucially, the exact delay can be varied depending upon the

assumptions made about the number of configuration cycles required to configure a resource and the period of the clock domain.

5.5.1 Resource Binding Transform

In the previous chapter each characteristic associated with the module-based temporal partitioning of a design was presented and shown to be quantifiable through a metric. When embodied by the cost function, together they enable an optimisation algorithm to explore the trade-offs between the different aspects of the design they represent during its partitioning and optimisation, in addition to enumerating the cost of the device-level infrastructure which ultimately implements the design using run-time reconfiguration. The next step will bring together the work presented thus far, by defining a transform which under the guidance of an optimisation algorithm will be used to perform the temporal partitioning.

The motivation for using a transform to implement the partitioning of a design is the opportunity it provides in exploring the combined effects of context switching and control graph and data-path optimisation during synthesis. This is achieved in practice through its integration into the existing transform-oriented framework provided by the MOODS synthesis suite. An example of their interaction is during the early stages of optimisation, where there are many more control states during which the reconfiguration of a module may be scheduled to overlap, however, as optimisation progresses their number is reduced due to the chaining of instructions per state. Those modules where the reduced reconfiguration time is the deciding factor in whether their assignment to a dynamic context is accepted or rejected during partitioning, are more likely to be rejected during the later stages of optimisation and are very likely to be rejected when optimisation is done independently and prior to partitioning.

Although there are numerous ways to partition a design, there are likely to be a number of distinct moves which may be taken. The task of the transform is to perform any one of these moves under the direction of the optimisation algorithm. The term ‘direction’ is used to describe the role in which the optimisation algorithm plays, since a sub-module is selected from a source context of its choosing and moved to a destination context of its choice. How it makes its choices is very much dependent upon the nature of the heuristic algorithm used, be it stochastic or deterministic in its approach, examples of each are currently employed during

optimisation in MOODS. Before examining the different approaches to temporal partitioning and the role in which the context switching transform plays while under the supervision of each optimisation algorithm, let us first consider the behaviour of the transform itself.

Recall that MOODS achieves optimisation of a design through an iterative process of selecting a transform and a target, simulating the changes that would be made to the design by the transform and quantifying those changes through the cost function. Only then can it determine whether or not the optimisation of the design is being guided towards its objectives – in which case the transform may be applied or away from them, the response to which is generally rejection depending upon the algorithm used. The deployment of the context switching transform is completely compatible within just such an optimisation framework. Assuming that the context switching transform has been selected by the optimisation algorithm, how it came to be selected is inherent to the nature of each of the optimisation algorithms and is described in a later section. The effect of executing the transform is to assign a single sub-module to the static or a dynamic circuit context, although, occasionally a number of sub-modules may also be re-assigned in the process, the result of clustering those sub-modules allied through an execution hierarchy.

Figure 5.16 illustrates the reaction (shaded) of the transform to each task and decision undertaken by the optimisation algorithm. The first task of the optimisation algorithm is to select the target data structures to which the transform is applied. The type of data structure will depend upon the transform selected, for example, a pair of control states is targeted for merger by the scheduling transform ‘Sequential merge’. In the same sense, the context switching transform is applied to the sub-module and circuit context data structures. The sub-module is chosen, regardless of where it currently resides, be that in a static context or as part of a dynamic context.

The next decision illustrated is whether the sub-module should form the basis of a new circuit context on a new reconfigurable region or be assigned to an existing context and region. This will be influenced by the properties of the module, such as whether its execution is concurrent to those currently assigned to the region − in which case their execution must be mutually exclusive for it to be allocated to the same region. When this condition cannot be met, a new region must be created. There is no decision to be made during the first execution of the

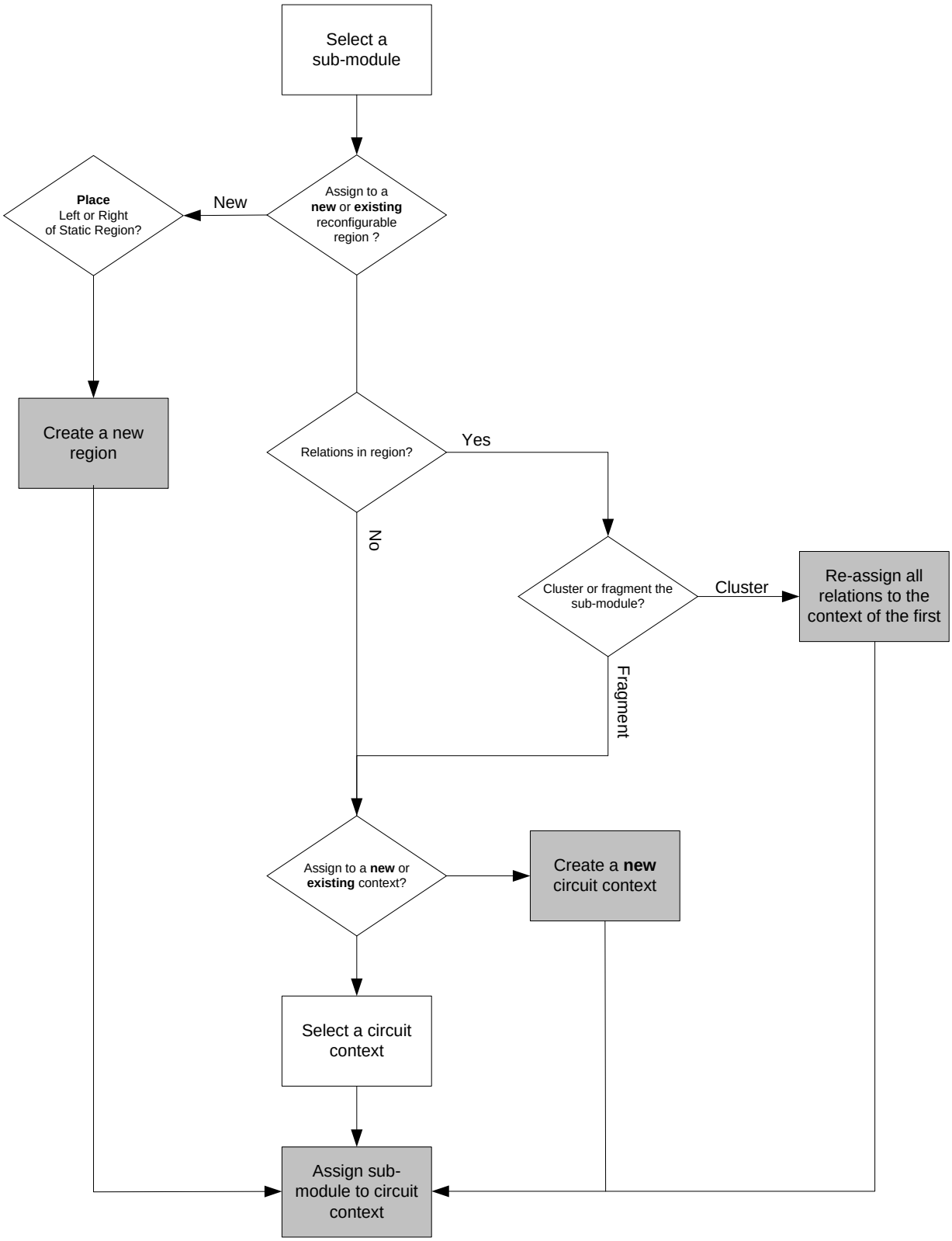


Figure 5.16: Decisions made during application of the context switching transform.

transform, where there are no existing regions to choose from, in which case there is only one course of action and the transform responds accordingly by creating a new region and a context to which the sub-module is assigned.

Prior to its creation, the placement of the region is determined by the optimisation algorithm. This is due to the fact that its location, in relation to the static region, determines the number of regions through which the associated communication channel(s) must pass through, in order to connect any pair of modules divided between the regions. As the number of buffers utilised in the construction of a channel is a product of its length (in terms of the number of regions it crosses), there is a cost incentive to limiting the number of regions bridged by the channel, which is determined by the placement of the regions it connects.

Alternatively, if an existing region is chosen, what happens next is very much dependent upon the relationship between the sub-module selected and those currently resident in the region. The term ‘relationship’ is used to denote the execution hierarchy which may exist among the sub-modules, where the module selected may initiate the execution of another, currently resident in the target reconfigurable region or vice versa. In either case, the related sub-modules cannot be simultaneously active in a common region, unless they are clustered in the same circuit context or their inter-module signals are buffered, fragmenting their execution over multiple contexts. The assignment of the existing sub-modules remains undisturbed when their execution is divided over several circuit contexts. The same cannot be said of the modules affected by clustering which are re-assigned to the temporal context of the first related module in the region, along with the module selected earlier.

The decision to fragment or cluster a module hierarchy need not be taken solely by the optimisation algorithm, the user may regard it as an experimental parameter to independently evaluate the impact of clustering and/or fragmenting the hierarchy upon the quality of the final solution.

The final decision taken by the optimisation algorithm is whether or not to create a new context for the sub-module or assign it to an existing one. Once again, it will be sensitive to the presence of a module execution hierarchy and the approach subsequently taken in response to it. The outcome of the decision favours either a smaller ‘footprint’ in the reconfigurable region targeted or a reduction in the level of context switching it experiences.

There are a number of intervening stages between the selection of a sub-module, circuit context and the application of the transform; the first of which is an estimation of the effect of the transform on each of the cost function metrics: it enables the cost function to indicate whether the effect of the transform constitutes an improvement or degradation in the optimisation of the design. Figure 5.17 illustrates the steps required to estimate the impact of the context switching transform and the order in which they must be taken. All the steps are undertaken, irrespective of which of the moves has been chosen and regardless of the distinct configurations of sub-module hierarchies that may occur.

Each step need only be carried out for those reconfigurable regions directly affected by a move, the source of the sub-module (if it is not currently assigned to the static context) and the destination region.

The first three steps are self-explanatory and correspond directly to the estimation of the area post-partitioning, dimensions of the communication channel(s) and a measure of the variation in area of the set of circuit contexts assigned to a reconfigurable region. They were exemplified in the previous chapter, during the overview of the metrics and formally defined during the problem definition that followed.

The remaining steps provide an estimation of the reconfiguration overhead associated with the control paths which may be taken through the circuit being synthesised. Recall that the reconfiguration overhead of a context is predominantly a product of the time taken to load it into the configuration memory of the target device and the frequency with which this occurs during the lifetime of a design’s execution. The sequence of module calls which lie on a given control path and their assignment during partitioning determines the number of times a module and the circuit context to which it is assigned are swapped and the context and modules with which they are swapped.

A profile of the design obtained through a simulation of its execution may aid in the identification of the control paths that are most likely to be taken in practice. This of course assumes that in addition to exercising the various aspects of the design, the test-bench can also accurately model the likely run-time conditions under which the design will operate. In such cases, the value of this approach is reliant upon there being a significant disparity in the

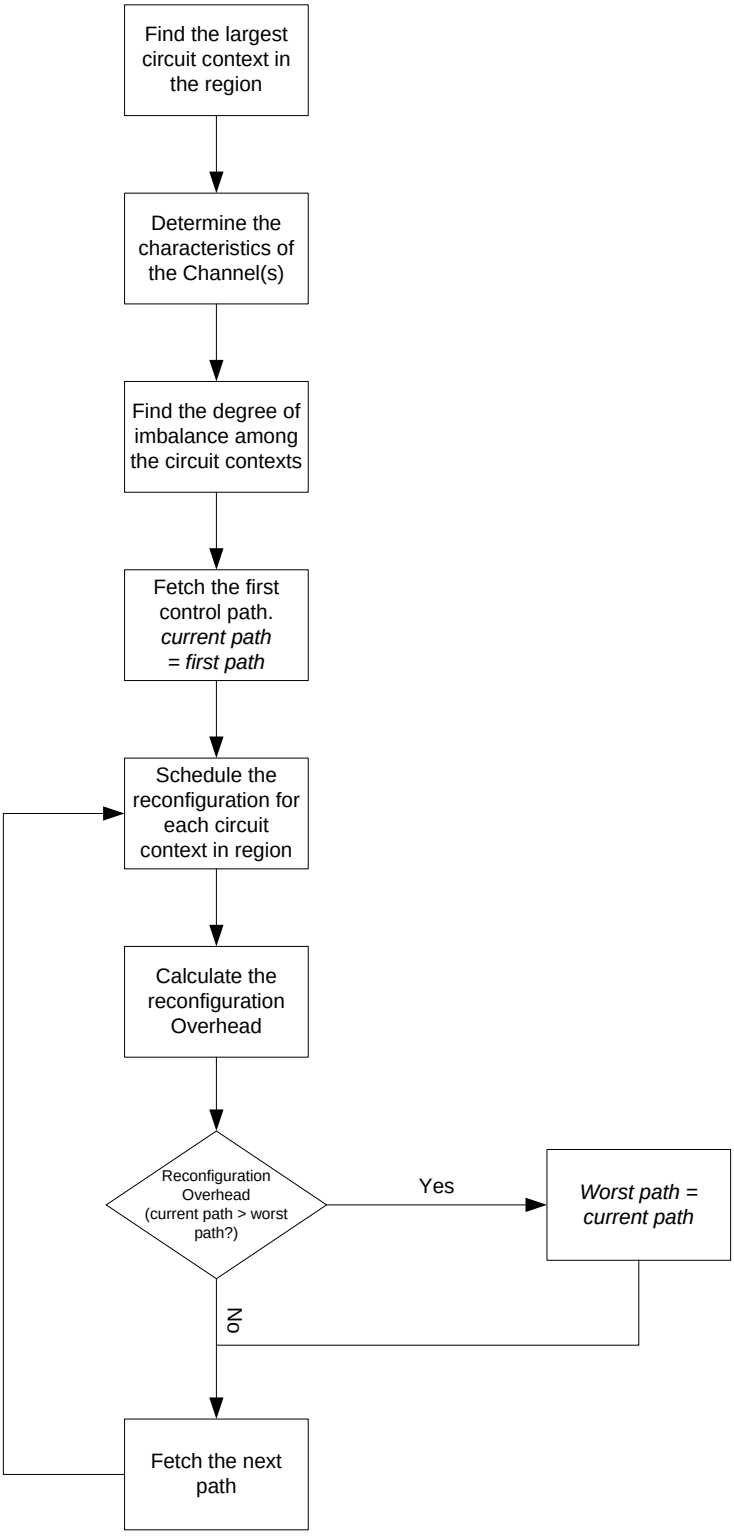


Figure 5.17: Estimating reconfiguration overhead for the context switching transform.

likelihood of execution among a number of paths and it being a recurrent feature in other paths of the design. Those paths which do not exhibit a clear contrast in the likelihood of their execution are then assumed to have an equal chance of being taken, in which case the task during estimation is to identify the path associated with the greatest reconfiguration overhead. This is illustrated by the remaining steps of Figure 5.17, where the reconfiguration overhead associated with each and every path is found. In doing so, an upper bound estimation of the reconfiguration time can be established for the present configuration of temporal circuit contexts.

5.5.2 Context Switch Instruction

To maintain the original execution order of the sub-modules and in doing so, preserve the behaviour of the circuit design being synthesised, every module must be present in its designated reconfigurable region prior to being activated by any of its associated calls; regardless of where it has been assigned during partitioning. To do so, necessitates the continual switching of each reconfigurable region between a set of associated temporal contexts. Where a context switch is deemed necessary, the activation of the reconfiguration controller required to perform a self-reconfiguration of the device must also be scheduled. The earliest a context switch may be initiated is determined by the last execution call to a sub-module already resident on the shared reconfigurable region. On the contrary, the latest a context switch may occur is during the cycle prior to the activation of the sub-module concerned. Together, they define a partial graph from which an arbitrary vertex may also be selected to schedule a reconfiguration.

When scheduling the reconfiguration of each of the circuit contexts, the initiation of a context switch is marked using a dedicated instruction. The origin of the ‘ContextSwitch’ instruction, as its name suggests, does not lie as the others do in the semantics of the behavioural specification, rather, it owes its existence to the context switching transform. For it is only during optimisation that each instruction is inserted or removed to and from a control node under the guidance of the temporal partitioner. This reflects the transparency in which run-time reconfiguration is deployed during synthesis, in response to the user’s optimisation objectives and targets, rather than requiring their involvement and any explicit reference to reconfiguration in the description of the circuit design.

Akin to the majority of ICODE instructions, the context switch (CS) instruction is associated with a node in the data-path, in its case the reconfiguration controller. There are two operands of a CS instruction, namely ‘Segment’ and ‘Module-Instance’, which together relate the control state in which the instruction is assigned with the execution call of the module being partitioned. The first identifies the end of the reconfiguration segment by recording the actual module call. When the call occurs within another sub-module, it no longer uniquely references the segment.

The second operand differentiates between what would otherwise be a number of identical segments, each the result of a separate call to the module in which the reconfiguration segment is scheduled. Each addition of a CS instruction to a control state creates a dependency between the ‘ModuleLeap’ instruction associated with the sub-module call and itself. Respecting the precedence of the instruction dependency ensures that a module is activated only after the circuit context to which it is assigned is configured in the reconfigurable region. A new instruction group is created for the instruction and stored within the control state, reflecting its independence of any other instructions assigned to the state and the concurrency in which the reconfiguration controller can operate, to exploit the partial reconfigurability of the target device. In this way, it is possible to overlap the execution of an instruction with the reconfiguration of another, albeit as part of a sub-module being reconfigured.

The importance of the role played by the scheduling of the circuit contexts is primarily dependent on the extent of its influence in reducing the reconfiguration overhead through the overlapping of reconfiguration with execution and secondly, in its interaction with the existing scheduling transformations such as the sequential merging of control states.

5.6 Transform Interaction

Signifying the start of each context switch with a ‘ContextSwitch’ instruction and realising temporal partitioning through a transform, not only enables the simultaneous application of the scheduling and context switching transformations within a single optimisation run, but also ensures that where relevant, any optimisation to the control graph or data path is also

reflected in the temporal partitioning of the modules affected and the scheduling of the circuit contexts to which they are assigned.

Figure 5.18 illustrates one such interaction between the application of the Sequential merge transform and its effect on the scheduling of the temporal circuit contexts. Each section of control graph depicted in Figure 5.18 (a-d) is committed in some way to reducing the reconfiguration overhead associated with swapping sub-module 'X' with 'Z' during the period of its execution. A shaded vertex denotes the beginning and end of each reconfiguration segment, where the beginning is identified by the CS instruction assigned to it and the end is an activating call to the sub-module in question; sub-module 'Y' is assigned to the static context. In actuality, all other vertices would be associated with an ICODE instruction, not shown in the example for the sake of clarity.

The reader will recall the purpose of the Sequential merge transform, that is to re-assign a group of instructions associated with one control state 'n_2' to those of another 'n_1' and in doing so, reducing the length of the critical path by one state (subject to there being no instruction dependencies with those of the intervening states, nor any shared data path nodes between the instructions allied to the pair of states selected for merger − unless the instructions are mutually exclusive). The states may exist in any of the modules, be they in the program module, sub-module or nested sub-module. Each of the four sections of graph illustrates a different scheduling scenario from which the pair of control states (n_1, n_2) is selected for merger:

a) The control state n_2 denotes the start of the reconfiguration segment for sub-module Z, whilst state n_1 (the destination of the merger) has no association with any segment. For those paths on which n_2 marks the beginning of a reconfiguration segment, the result of its amalgamation with n_1 will be an earlier context switch of the module, the consequence of an increase in the segment length. This ASAP effect will decrease the reconfiguration time for module Z by 15 cycles (the number of states overlapped by the merger, including those of sub-module Y minus the removal of state n2, upon a successful merger). If n_2 does lie on the path incurring the greatest reconfiguration overhead (worst path) than a re-evaluation of all paths is undertaken to verify whether

or not the merger has brought about an improvement in the path which may no longer differentiate it as the worst.

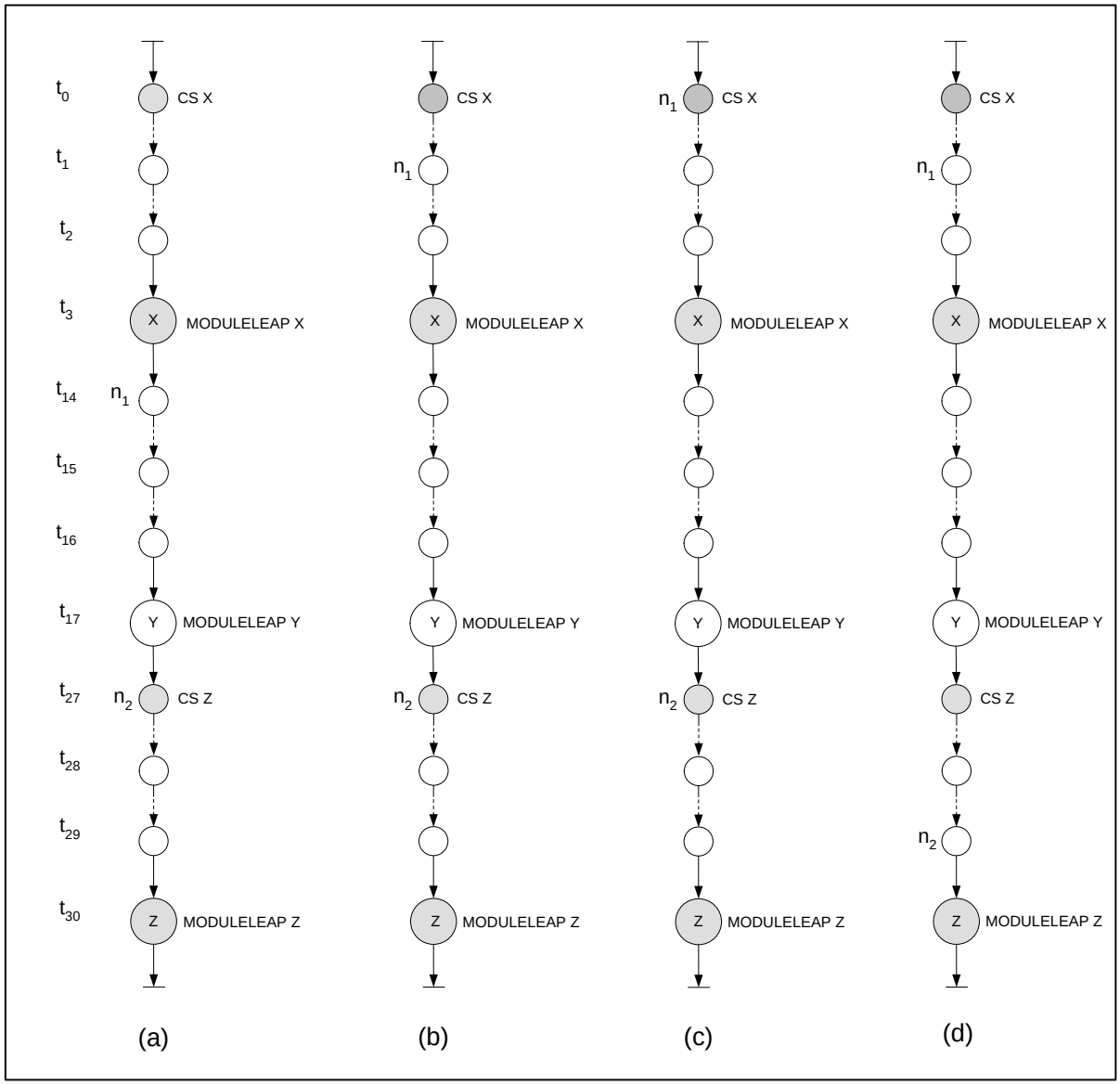


Figure 5.18: Merging control states from within reconfiguration segments.

b) The merger of the pair of control states will result in contention for the configuration port of the target device. This is due to n_1 being part of the reconfiguration segment of module X and n_2, once again, signifying the start of the segment for module Z. If the states occur in a sub-module, it is pertinent to verify whether or not their segments relate to a single temporal instance of the module and also if they are executed on the

same control path, otherwise an overlap in reconfiguration cannot take place. When it does occur, there are a number of responses which may be adopted.

The first is to prohibit the contention from occurring by ensuring that it is detected and discarded during the validity test for the sequential merge transform prior to its estimation. The disadvantage with this approach is the missed opportunity for optimisation, the consequence of discarding a pair of control states. Another is the increase in the optimisation time resulting from the time spent re-selecting an alternative pair of control states, especially when it is a common occurrence.

A different approach is to re-schedule the start of the segment associated with n_2, for example, assigning it to the preceding node Y and in doing so, permitting the merger of the states whilst avoiding the simultaneous access of the device configuration port. Alternatively the sub-module of one of the conflicting contexts can be re-assigned to the static context, in this case X or Y, thus permitting the merger of the control nodes.

The change in the partitioning will require an update of all the affected metrics (area post-partitioning, channel and reconfiguration overhead). In particular, a revision to the reconfiguration overhead metric may necessitate the re-determination of the worst path, the repercussion of change in the swapping characteristics of those modules on whose paths the conflicting module(s) were executed. Whichever approach is taken to resolve a reconfiguration overlap, the merger of the control states and the changes required to permit it are jointly estimated.

c) Once again a conflict will arise following the amalgamation of the two states, both of which are scheduled to initiate a context switch of a reconfigurable region for their associated module X or Y − the choice of solution adopted is the same as (b). Upon application of the transform, one of the CS instructions is either re-assigned or removed permanently from the control state and once again, the re-evaluation of all paths is undertaken, in order to identify the path which will incur the greatest reconfiguration overhead.

d) This scenario is dissimilar to the others, in as much as, although n_2 is part of the reconfiguration segment of module Z, it does not mark the beginning of the segment.

The elimination of n_2 (if its instructions are re-assigned to n_1) will reduce the length of its former segment of graph by one state, irrespective of the circumstances of n_1 (be it the start of the segment as shown, part of the segment or not part of any segment). Consequently, there will be an increase in the reconfiguration time associated with the loading of sub-module Z and any paths on which it lies. Regarding the worst path, there is no requirement to search for it, as either, the pair of states are taken from it and consequently their merger will reduce the length of segment associated with n_2 by one cycle (maintaining its designation as the worst path) or the path on which the pair lie must have been at least a cycle shorter than the worst path, in which case, its increase in reconfiguration time can only equal that of the worst path. In either case, the re-determination of the worst path is unnecessary.

Each of the scenarios described above can also be encountered during the application of the 'Merge fork and successor' transform, its purpose is to move the instructions of a successor state into its predecessor (the fork), in doing so, the control arc is converted into a conditional instruction making the immediate successor state redundant. Eventually, upon repeated application of the transform, the fork construct itself becomes superfluous. As an individual branch, the fork can be regarded as a general control state with a single output arc to a successor, then a reconfiguration segment affected by the merging of a pair of sequential general control states would be similarly affected by the merger of a fork and its successor, had the segment lain on a single branch of the fork. The rest of this section reviews the effect which the remaining transforms can exert upon the reconfiguration of temporally partitioned modules.

A variable that is written to and read from by a single instruction executed within its own control state is an appropriate candidate to which the 'Group instructions on variable' transform may be applied. Each read or write is implemented in the data path by a register with a single input and output data-path net. Merging the write and read instructions within a single state removes the register and the state in which it was formerly written.

Since either state may be associated with a reconfiguration segment, once again, there is the possibility that their merger will affect the scheduling of a context switch. There are a number of circumstances similar to those encountered by the previous transforms. The states may

occur within a single segment or belong to entirely different segments, in either case the effect of applying the transform will be a reduction in segment length, at the expense of an increase in its associated reconfiguration time. However, unlike the other transforms, the target state (in which the variable is read) must occur after the state where it was written.

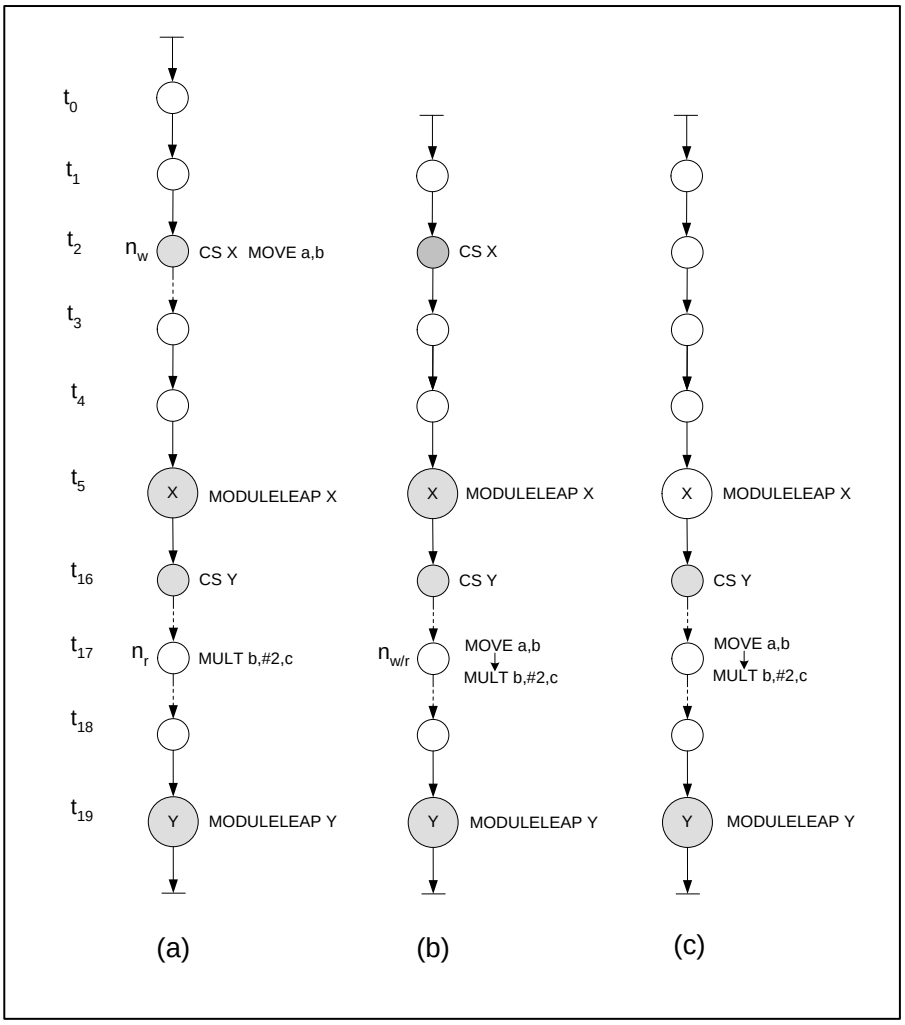


Figure 5.19: Group instructions on variable transform and reconfiguration segments.

Depicted by the sections of control graph in Figure 5.19, deciding when to schedule reconfiguration can present a subtle dilemma if the writing state 'n_w' of the variable is also used to mark the beginning of a reconfiguration segment not encompassing the reading state 'n_r'. The amalgamation of the segments will break the instruction dependency between the CS and ModuleLeap instructions and result in an execution call to module X before it has been loaded in to its reconfigurable region. To prevent this from happening and to also enable the transform to be applied, the start of its segment is re-scheduled to the preceding state at time

step 't_1' shown in Figure 5.19 (b) or the instruction dependency and segment can be removed by assigning the effected module X to the static context (c). The decision as to which response is deployed is implemented through a parameter set by the user prior to optimisation.

The 'Inverse Scheduling' transforms can also exert an effect over the scheduling of a context switch. The purpose of the first 'Ungroup-Node Into Time' transform is to un-chain groups of instructions whose execution within a particular control state exceeds that of a delay parameter set during optimisation or through the cost function, to specify a maximum clock period constraint.

The selected control state may form part of a reconfiguration segment or be used to mark its beginning, as shown in Figure 5.20 (a). The state is left undisturbed since it has no other dependent instruction in the state, in contrast to the other instruction group. In Figure 5.20 (b) the state's instructions are re-assigned to new control states, where the exact number is determined by the instruction depth of the group and the size of the target node delay parameter (25 ns in the example shown) which the transform attempts to meet. Each additional control state lengthens the segment *'S'*, in effect, scheduling an earlier start for reconfiguration of the associated module and its circuit context. Should the control state occur in a sub-module, the change in scheduling must also be taken into account for each and every temporal instance of that sub-module. For all reconfiguration segments, irrespective of their module association, only those segments which lie on the control path credited with generating the greatest reconfiguration time will contribute to the cost function (where there are a number of control paths, each of which are equally likely to be taken) and ultimately determine whether or not the transform will be applied.

In much the same way, the 'Ungroup node into groups' transform can also act to lengthen a reconfiguration segment S, although how this is achieved differs to the previous transform, in that an entire instruction group is extracted to a new control state, as illustrated in Figure 5.20 (c). When the group chosen initiates the reconfiguration of a segment, its re-scheduling a cycle later will have no consequence on reducing the length of its associated segment (Figure 5.20 (d)). Once again, the effect of the transform is repeated, where applicable, on any number of module instances and only reflected in the cost function when it occurs on the worst path.

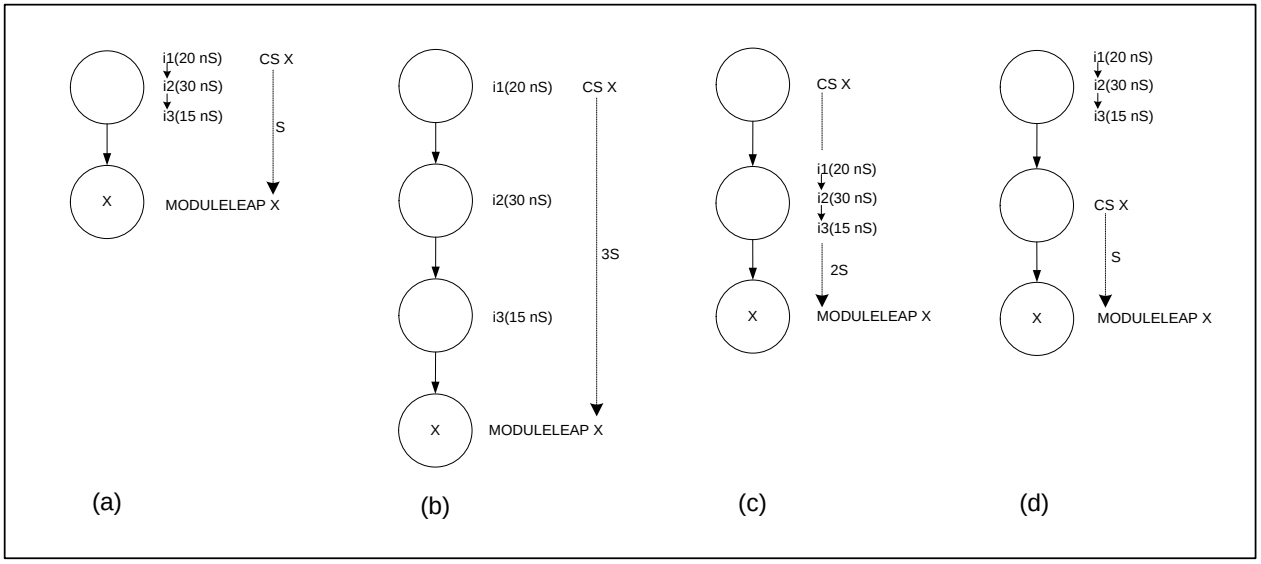


Figure 5.20: Inverse-scheduling transforms and the timing of reconfiguration segments.

The outcome attributed to each transform thus far, has been a change in the scheduling of each swap of a temporal circuit context. What remains to be considered is the consequence of optimising the design for a reduced circuit area, be it through the merging of control states and the subsequent removal of registers among their dependent instructions or through the sharing of functional units on the data path. Its relevance to temporal partitioning is through a change in the circuit area of each sub-module.

At any point during partitioning, the circuit area is found as the combined area of the static circuit context and each of the reconfigurable regions. Recall that the area of each region is defined by the largest temporal context assigned to it. Any change in the area of one of its sub-modules will bring about a re-evaluation of the dimensions of all its contexts, in a bid to find the largest. Significant changes to the circuit area can also arise from the data-path allocation transforms, whether the outcome is a reduction in area arising from the mapping of a single functional unit to two or more instructions, or its increase, as a result of the inverse allocations transforms – they attempt to reverse the sharing of a data path unit, either for a single instruction and unit or by separating all instructions by returning to a one-to-one mapping of an instruction to functional unit. In addition to the effect on the area metrics, any change in circuit area will also impact on the reconfiguration time. This is due to it being directly proportional to the area of the module, modelled in terms of its columnar usage of resources and ultimately at the device level, through its configuration bitstream.

5.7 Summary

This chapter continues the theme of representing run-time reconfiguration in terms of circuit abstraction and the infrastructure to realise it: at the highest level, the sub-module behaviour is encapsulated through a System layer. At this level, the workings of the infrastructure are completely transparent to any sub-module wishing to activate another. This is applicable irrespective of the topology or placement of the static and reconfigurable regions, such detail is relevant only at the next level, the Communication layer.

A temporal partitioning of the subroutine modules will break the assumption that all modules are ‘on silicon’ at the same time. As a consequence, a fixed communication channel provides the interface necessary to context switch the resources of an FPGA, in response to the sequence of module calls executed.

As a result of temporal partitioning, the ‘topology’ of the partitions will determine the characteristics of the communication infrastructure. One example of a particular topology might require all subroutine modules to share the same communication channel. In this scenario, the control and data-path signals passed between a pair of active modules at the system level are represented as transactions of the channel, the use of which is governed by protocol.

Central to the communication protocol is a module address ROM, customised during synthesis to provide each module with a unique binary address in the architecture, irrespective of the circuit context and region in which it is assigned to during partitioning. Modules hierarchies of any depth are permitted, where each module can also be placed in any circuit context and assigned to any region. The exception is the top level program module which must remain active in the static region throughout the circuit’s execution, from where it can initiate the first call of any execution hierarchy and receive notification upon its completion of execution.

As its name implies, the Physical layer provides the device level control necessary to perform partial reconfiguration of the FPGA. Its remit concerns the fetching of the data-streams and the control of the device configuration port, both of which are done ‘on the fly’ during circuit execution. Akin to the module address ROM, all circuit context data-streams are tagged with

their location in a memory. It contains more than the raw data-streams, as a number of micro-coded commands are used to locate each data-stream, in the presence of any conditional control paths on which its module might lie.

Having defined the layers of abstraction, the remainder of the chapter described how the architecture is implemented in MOODS HLS: a temporal partitioning transform has been created to partition VHDL subroutines to dynamic circuit contexts, each of which are assigned to execute upon isolated regions of an FPGA.

A new ICODE instruction has been created to mark when a context switch of a sub-module partition occurs. It is used by the scheduling routines to overlap each segment of reconfiguration with the execution of a sub-module already active on the device. In addition to reducing the reconfiguration time, the scheduling of each temporal context takes into consideration the frequency in which a circuit context is swapped with another, as dictated by the execution order of the sub-modules.

The cost function described in the previous chapter measures the trade-off between the area reduced through temporal partitioning and the reconfiguration penalty incurred. Through the use of the context switching instruction, the partitioning transform provides the cost function with a means of denoting when reconfiguration occurs; however, the physical resource use must also be included in the cost function. In practice, the instruction is allocated to a reconfiguration controller in the data-path.

The reconfiguration controller enables the MOODS state machine controller to perform self-reconfiguration of the FPGA through either its external control pins or internal configuration port; depending upon the characteristics of the target device. In addition to the reconfiguration controller cell, a number of channel controller cells are also synthesised, as part of the infrastructure necessary to support partial reconfiguration. Each controller is local to the region in which it is placed and is the principle means through which one region can communicate with another. The control and data signals pass through a number of parallel communication channels, the exact number of which, as with all aspects of the architecture is determined during synthesis.

In temporal partitioning, ‘when’ a subroutine will utilise a resource is as important as ‘where’ that resource might be. With that is mind, the chapter concluded with an examination of the interaction between the subroutine-level resource binding transform and the existing instruction-level scheduling transforms: it described how a number of scheduling transforms are exploited to influence the times in which context switching can occur, in some cases even acting to influence the formation of the partitions. By doing so, the chapter re-iterates the importance of considering a circuit representation at more than one level of abstraction, a perspective made practical by the work described in this chapter.

Chapter 6

Implementation and Results

This chapter presents the results obtained using MOODS behavioural synthesis after incorporating the approach to temporal and spatial partitioning described in the previous chapter.

6.1 Experimental Objectives and Method

The purpose of the following experimentation is to assess the use of the simulated annealing optimisation algorithm when employed during module-based temporal partitioning. However, the absence of a specific approach to partitioning means that the algorithm can also be used to explore the relationship between an action or 'move' taken during partitioning and its effect individually and collectively on the partitioning metrics. More specifically, whether or not a particular move is biased toward a given criterion in a way which drives it closer to or further away from its user specified target.

The first step required to achieve these aims is to establish the annealing schedule. This consists of defining the initial and end temperature parameters (T_{start} and T_{end}) respectively, the magnitude of the temperature steps required to pass through the temperature range and finally, the number of transformations applied at each temperature step. In cases when partitioning and optimisation are performed separately, the latter two parameters fix the number of partitioning moves undertaken to 1000 – the minimum number of moves found from early experimentation and thought necessary to quantify the behaviour during partitioning, whilst being sensitive to the time taken to perform the experiments. Where optimisation to the control and data-paths occurs alongside partitioning, the exact number of partitioning moves is determined by the optimisation algorithm as it decides which transforms to apply during each temperature step.

The next task is to set up the cost function so that during the course of the experimentation, the effect of partitioning on an individual criterion can be quantified. Recall that each move taken during partitioning is measured using the cost function:

$$cost_{TP} = C_{part_{area}} \cdot A_{TP} + C_{reconfig} \cdot T_R + C_{channel_{width}} \cdot B$$

where: C_{part_area}, $C_{reconfig}$, $C_{channel\_width}$, C_{Bal} are weighted constants used to reflect the user-specified optimisation priority of the area (post partitioning), reconfiguration overhead and the buffer utilisation metrics respectively.

The impact of circuit partitioning on an individual cost function criterion can be determined by setting the priority of the criterion to high whilst ensuring that the remainder are set to low. The task during partitioning and/or optimisation is to meet the target of the highest priority metric before proceeding to the next. The 'status quo' can be maintained throughout the partitioning and/or optimisation session by setting the criterion under examination at a priority higher than the others, in conjunction with a target minimising to zero. All criteria targets are set to zero, with the exception of the channel metric: its target is determined dynamically during the course of partitioning, from a model of the target FPGA chosen to closely fit the area which would result from the proposed partitioning of the circuit modules. In doing so, it provides an upper bound on the number of buffers utilised in the channel(s) linking the reconfigurable regions on which the circuit contexts are executed. Since it is possible in MOODS to implement multiplexors using tri-state buffers, for instance, to facilitate the sharing of data-path units during optimisation, the channel will not have access to all of the buffer resources available on the FPGA. To investigate the effect this can have on the channel buffer metric, a percentage of the available buffers is used to realise the communication channels.

The following cost functions are used during experimentation:

1. A_{TP}(High), T_D(Low), $B(Low)$; $A_{TP_{target}} = 0$.
2. A_{TP}(Low), T_D(High), $B(Low)$; $T_{D_{target}} = 0$.
3. A_{TP}(Low), T_D(Low), $B(High)$; $B_{target} =$ (100, 50, 33, 25, 10, 1) % of target device.

4. $A_{TP}(\text{High}),\ T_D(\text{High}),\ B(High);\ A_{TP_{target}} = 0,\ T_{D_{target}} = 0,\ B_{target} = 0.$

The purpose of the first three cost functions is to examine whether the partitioning moves are accepted or rejected more frequently when associated with a high priority for a particular cost function metric. For example, setting the area metric A_{TP} to a higher priority than the others and assigning a target of area of zero would bias the cost function to using temporal partitioning to reduce the circuit area. The cost function is more likely to reduce the circuit area by relying upon moves which create or modify temporal partitions, as opposed to those which create the reconfigurable regions on which they are swapped.

The fourth cost function assigns a high priority to all metrics and in doing so enables the trade-offs between all aspects of temporal partitioning to be simultaneously evaluated through the cost function metrics. With reference to the previous example: the frequent use of temporal partitioning would now be weighed against the penalty of a reconfiguration delay. A compromise between the conflicting metrics could favour the re-assignment of subroutines between existing partitions, rather than rely upon partitioning moves which favour a high priority in either area or reconfiguration delay metrics through the creation of more temporal partitions or reconfigurable regions, respectively.

The purpose of the fourth cost function is to examine the most likely scenario required by the user, which is to partition a circuit with the goal of minimising all metrics without precedence. The effect of this cost function is of particular interest since unlike the others, it cannot easily be inferred without evaluation. For example, making the area (post-partitioning) the highest priority in the cost function, encourages the creation of multiple single module circuits contexts: the optimum partitioning would be a single reconfigurable region whose dimensions are defined by the largest circuit context. At the same time, it should also discourage the addition of modules to those contexts, the consequence of which would be an enlargement of the associated region.

When examined individually, each of the remaining metrics would seem ‘on paper’ to improve or degrade in response to specific moves; for instance, the actions which would guide the area metric towards an optimal outcome should also have a diametrically opposite effect on the reconfiguration overhead. In this way, it echoes the classic area and delay trade-off

found in circuit optimisation. However, when all metrics are given equal priority, the final ‘energy change’ which quantifies whether the proposed move is one which will improve or degrade the partitioning is found by comparing the magnitude of the energy change of each criterion in turn. The criterion whose energy change dominates (be it in a way which improves or degrades the partitioning) subsequently determines the outcome of the move, a task which cannot be undertaken without some way of evaluating a given partitioning. Of course, this is the motivation behind quantifying the cost of partitioning and data and control path optimisation, enabling the user to explore the many alternative implementations of a circuit from a single behavioural description; an undertaking too laborious to perform by hand.

The remaining experimental parameter determines when the reconfiguration of each context-switch may occur (ASAP/ALAP) within the confines set by the behaviour of the modules being partitioned (the sequence of module calls in the control graphs) and their actual assignment during partitioning.

6.2 Results and their Analysis

The Synthesis results are interpreted by examining the relationship between the average energy change (dE) measured through the cost function and each action taken during partitioning. The partitioning ‘moves’ implemented by the context switching transform during the course of partitioning are: the formation of a new reconfigurable region and circuit context (to which a module is assigned); the creation of a new circuit context in an existing region; the expansion of an existing circuit context (a distinction is made should the move result in the partitioning of a sub-module execution hierarchy) and finally, the assignment of a module to the static region. Where applicable, the source and destination of the module being moved is taken into consideration, to identify the effect of re-assigning the modules within the same region.

During the execution of the annealing algorithm, the effect of each move upon the cost function criterion under investigation (highest priority) is recorded and categorised as improving (-dE), degrading (+dE) or having no effect (0 dE).

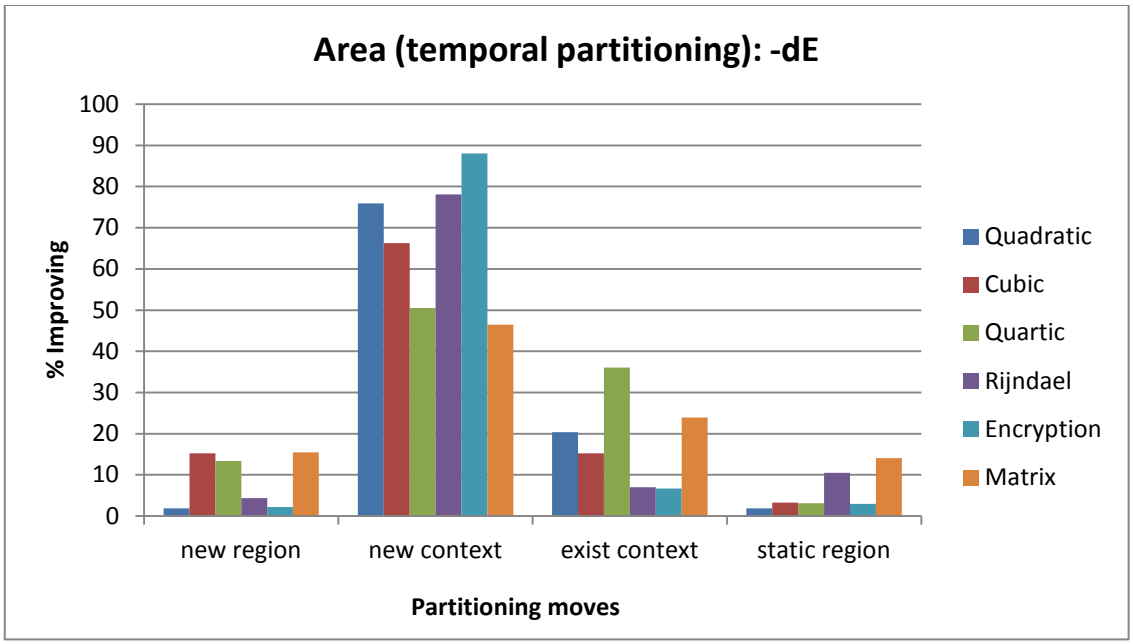
The graphs of Figure 6.1 depict the results generated when partitioning a circuit with the highest priority given to the area metric (post partitioning) for an annealing schedule commencing at T_{Start}=200. The schedule was also used for the remaining metrics, unless stated otherwise. Each energy change (y axis) associated with the particular type of move (x-axis) is expressed as a percentage of those moves which have the same effect on the metric, be that an improvement or degradation and are plotted to illustrate any contrasting effect.

Figure 6.1 (a) illustrates that the greatest positive energy change associated with the reducing the circuit area is achieved through the creation of new circuit contexts. Figure 6.1 (b) suggests that such contexts are formed without the need to create many new regions; this has the greatest single degrading effect on the circuit area, short of re-assigning the modules to the static region (also depicted).

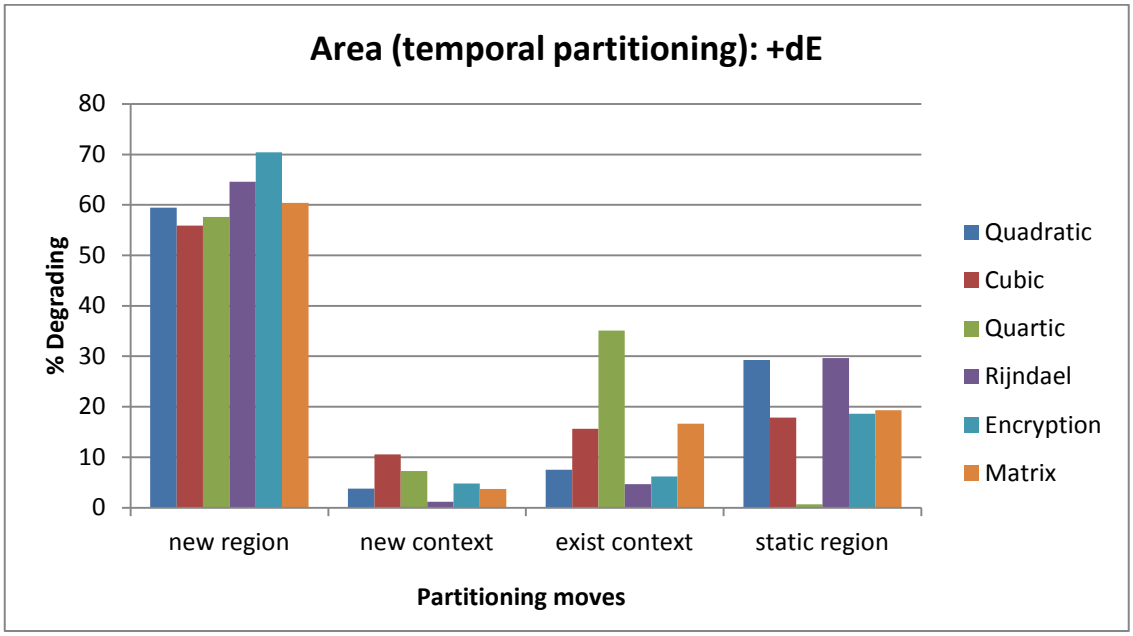
As for the remaining move, that of assigning a module to an existing context, the reader will notice that its effect is not as distinctive as the others. That is to say, the result of its application can be equally improving and degrading. The rationale for this is that the process of assigning a module to a different region or indeed within the same region can reduce the variation in size of the circuit contexts which are swapped over those regions affected by the move. In doing so, it can act to reduce the overall circuit area required by the regions. However, since module selection is done in an arbitrary fashion, there can also be as many degrading moves, as depicted.

The remaining metrics are examined using the same approach: all moves will have an improving, degrading or to a lesser extent no effect on a particular metric; but each move is examined in terms of the magnitude of its effect, the aim being to attribute one or two principle characteristics of each move with a distinct effect upon a given cost function metric.

In this way, the trade-offs between each of the metrics which to date have been inferred ‘on paper’ are now verified through experimentation.



(a)

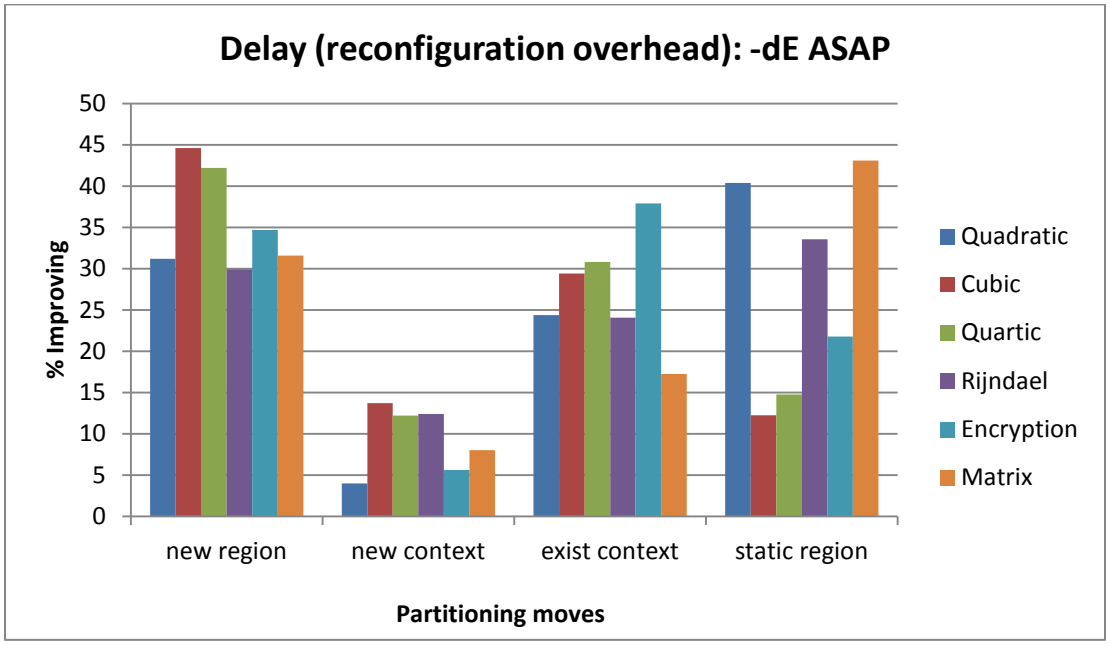


(b)

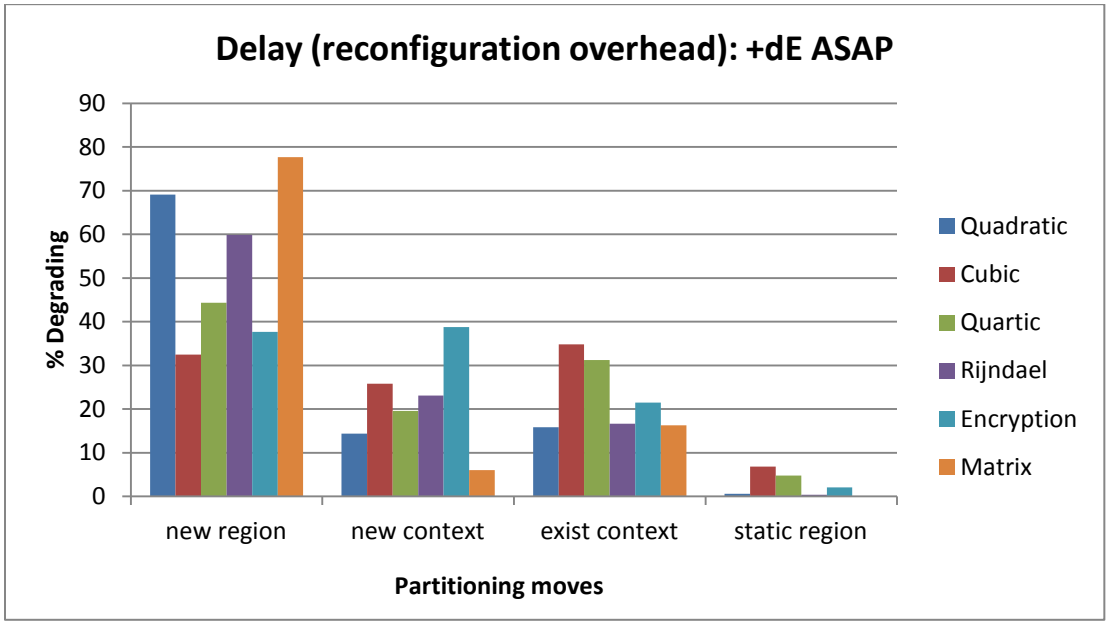
Figure 6.1: Circuit partitioning for circuit area set to a high priority.

The next metric under examination is the reconfiguration overhead, the results of which are shown in Figures 6.2 (a, b). Unlike the area metric, the general response to partitioning is not as clearly defined in terms of a predominant improvement or degradation to the

reconfiguration time, although rather predictably, little degradation comes from assigning the module to the static region.



(a)



(b)

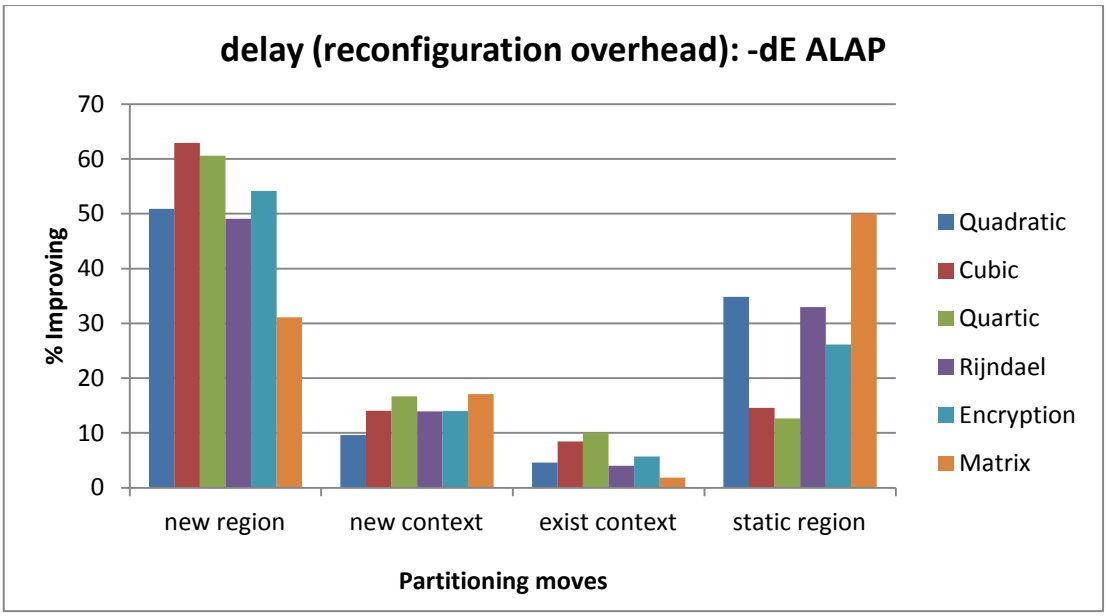
Figure 6.2: Circuit partitioning for reconfiguration overhead set to a high priority.

The first occurrence of a conflicting outcome is the consequence of creating a new reconfigurable region. It reflects the dual aspects of the reconfiguration overhead, derived from the time taken to load each module and the number of swaps required of the circuit context it is assigned to. Creating a new region and assigning a module to it will always increase the reconfiguration time proportional to the area of the module and hence the degradation to the reconfiguration overhead. However, in some circumstances when the module is taken from an existing region with a high rate of swapping among its contexts, the net effect over both regions (source and destination) is an improvement to the metric, reflected in the results as being on average the most improving move.

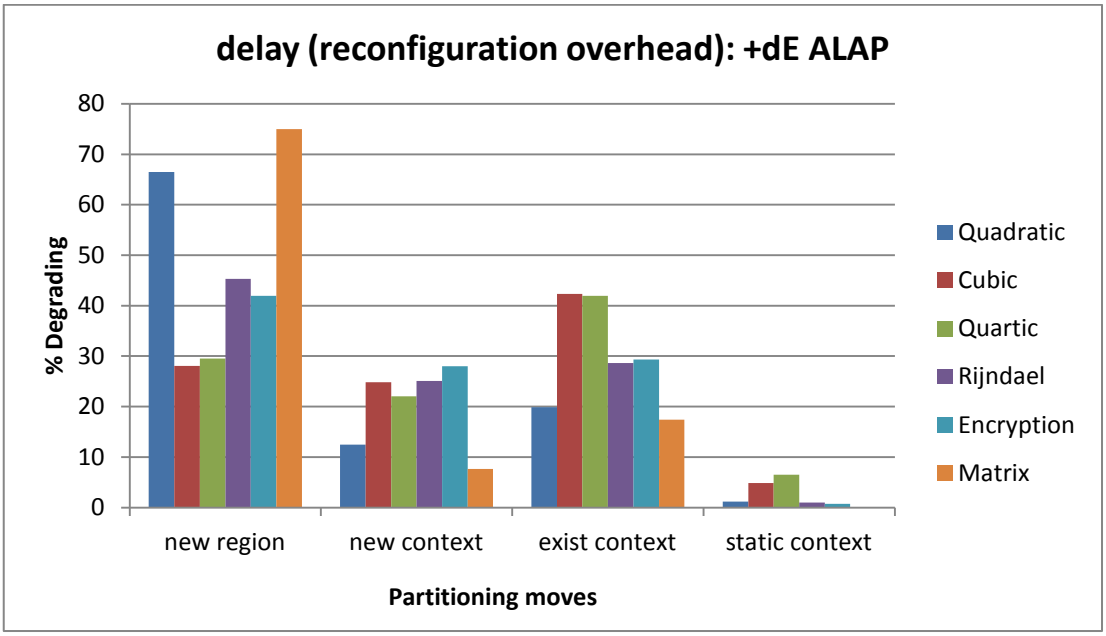
Another conflicting move and the second most improving move is the result of shuffling the modules between the circuit contexts, although on average, its improvement is almost equally matched by moving the modules to the static region. However, this is the least preferred option since it will also simultaneously degrade the area metric. The degree of swapping of a circuit context is always a multiple of its reconfiguration time and any attempt to re-assign modules which are taken from contexts frequently switched between undoubtedly acts to reduce the reconfiguration overhead, as illustrated. Of course, when this is done without sensitivity to the characteristics of the circuit e.g. not taking into account structures like finite loops which are present in the encryption examples, than the degradation can be significant.

Rather predictably, the next most improving (and least degrading) move is to assign the module to the static region. This is followed by an improvement in the reconfiguration overhead associated with the creation of new circuit contexts. Once again, the reduction in reconfiguration time is attributed to the benefit to the source and destination regions in terms of the reduced swapping. What is unexpected is that it is not the most degrading of moves. Its role ‘on paper’ is in opposition to the improvement gained by partitioning modules over circuit contexts. However, the results show the percentage of moves which degrade and not the extent of that degradation. This is examined in due course when all three metrics are given equal priority in the cost function.

Figure 6.3 illustrates the effect of scheduling the start of each context switch just prior to the execution of the modules assigned to them (ALAP), in terms of the percentage of improving and degrading moves. The results are compared with those in Figure 6.2.



(a)



(b)

Figure 6.3: Effect of scheduling each context switch as late as possible.

The reader will observe after comparing the first pair of graphs (6.3(a), 6.2(a)) that there are two distinct changes in the percentages of moves which improve the reconfiguration overhead. The first is a marked increase in the occurrence of creating ‘new regions’ to achieve improvement to the cost function. The second is a significant reduction in the percentage of

moves that assign a module to an existing context. An explanation for these discrepancies is that the increased penalty associated with reconfiguring ALAP means that assigning a module to a new region is more frequently relied upon to reduce the cost of swapping subprogram modules. As consequence of creating new regions, the number of moves to existing ones is significantly reduced − as can be seen in the pair of figures 6.3(a), 6.2(a), respectively.

An alternative explanation for a decrease in moving modules to existing partitions is the absence of any potential overlap in the scheduling of their 'Context Switch' instructions: section 5.6 in the last chapter describes several scenarios in which the scheduling of the temporal partitions can interact with the existing scheduling transforms. Scheduling a context switch as soon as possible (figure 6.2(a)) increases the likelihood of an existing scheduling transform overlapping the reconfiguration of circuit contexts. An improvement to the cost function can occur by allowing the affected contexts to merge and this opportunity is removed when their scheduling occurs as late as possible (figure 6.3(a)).

The remaining move to have been affected by a change in scheduling concerns the creation of 'new contexts' on existing reconfigurable regions. As shown in the charts of figures 6.2 (a) and 6.3 (a), scheduling each reconfiguration as late as possible led to an increase in the number of new partitions for those test circuits which featured little (Quadratic equation solver) or no nested sub-module execution (Rijndael, Encryption and Matrix circuits). The rationale behind this result is that separating a module from other members of the hierarchy will inevitably increase the reconfiguration overhead due to context switching between a partitioned sub-module hierarchy; creating new circuits contexts to offset any increase in reconfiguration delay due to scheduling can only be exploited by circuits without a significant execution hierarchy.

Regarding the effect that ALAP scheduling has upon the number of degrading moves, a comparison between figures 6.2(b) and 6.3(b) shows there to be no dramatic increase in their number; moves which incurred a high reconfiguration delay when scheduled ASAP continue to do so when the range of their scheduling is constrained. Another explanation is that the change in the number of improving moves shown in figure 6.3(a) compensated for the constraint imposed by ALAP scheduling. This explanation is supported by the fact that the

number of improving or degrading moves to the ‘static context’ (shown in figures 6.2 and 6.3) has not significantly changed due to ALAP scheduling.

The effect of the partitioning moves upon the remaining criterion, ‘channel buffers’ is considered next. Recall that unlike the other metrics (which are minimised to zero), the target for the channel buffers can be met during experimentation. This occurs when the number of tri-state buffers implementing the communication channels for a given partitioning of modules can be realised using the resources of the current target device. During such periods, the effect of a given move is quantified by the cost function through the next highest priority criterion, the equally low priority ‘reconfiguration overhead’ and ‘area’ metrics.

It was necessary to filter out the influence of the lower priority metrics and therefore only consider the direct relationship between a move and its impact on channel buffer utilisation. This was achieved by gradually reducing the percentage of the available buffers offered by the target device until the target was met, whilst recording the effect on the number of improving and degrading moves. The exemplar circuits do not exhibit any degree of concurrency in their structure. This means that at any given point during their execution, only one module need have possession of the communication channel. The resulting bi-directional channel requires a smaller percentage of available buffer resources and this is reflected in the results: only when a lower percentage of the available buffers are targeted (25-10%), is there evidence of a distinct effect upon the channel buffer metric.

The graphs of Figure 6.4 depict the nature of the improving and degrading effects upon the channel metric, when associated with each type of move taken during partitioning. The results depicted are generated using an annealing schedule commencing at T_{Start}=200.

A feature which the reader may have noticed is that some circuits exhibit an effect when subjected to specific moves, whilst others do not. The rationale for this is attributed to the characteristics of the subprogram modules themselves. For instance, in Figure 6.4 (a) the improvement brought about by re-assigning the modules to the static region is only present in those circuits which have the greatest variation in signal nets. In other words, the remaining circuits do not benefit from this move because their signal characteristics are unlikely to define the width of the channel e.g. the Rijndael circuit has 5 of the 8 modules with identical

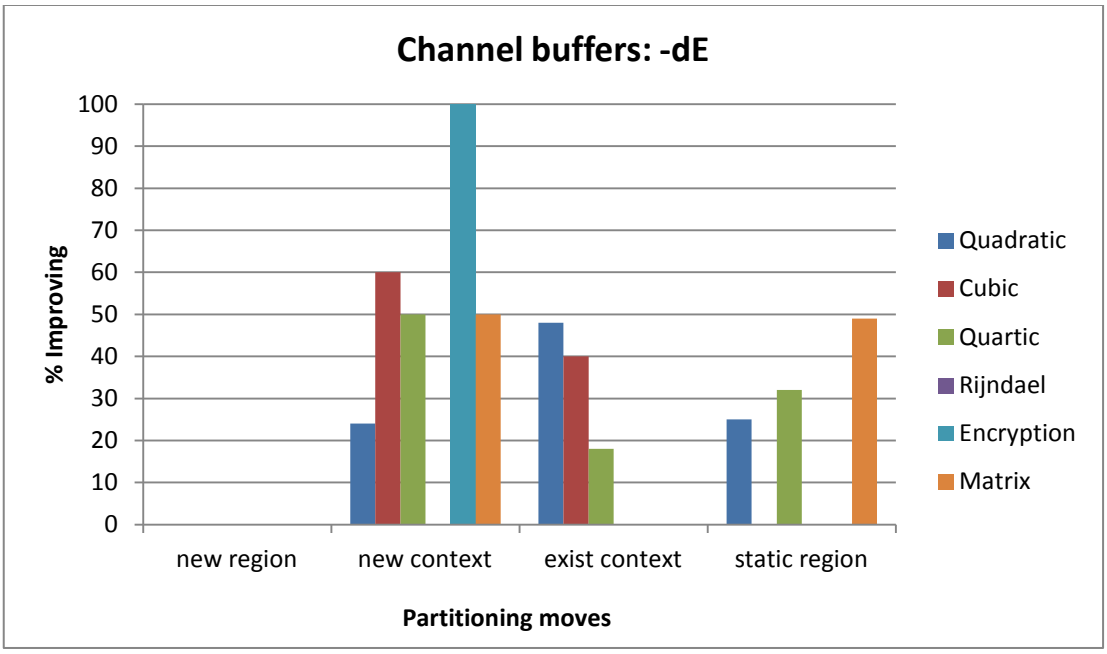
signal widths. Similar qualities are found in circuits which are also not improved by such a move − the reader is referred to Appendix B for further details of individual test circuits.

Figure 6.4 (a) depicts the improving effect of other partitioning moves upon the channel buffer metric. In addition to selecting a destination context on the static region, circuit contexts on existing reconfigurable regions also present an opportunity for improving the channel metrics. Two explanations account for this behaviour: the first is a predisposition of the test circuits and the effect it has upon the channel buffers concerning the assignment of subprogram modules to existing circuit contexts. Improvement in channel buffer utilisation occurs for moves that group pairs of dependent modules together in the same circuit context.

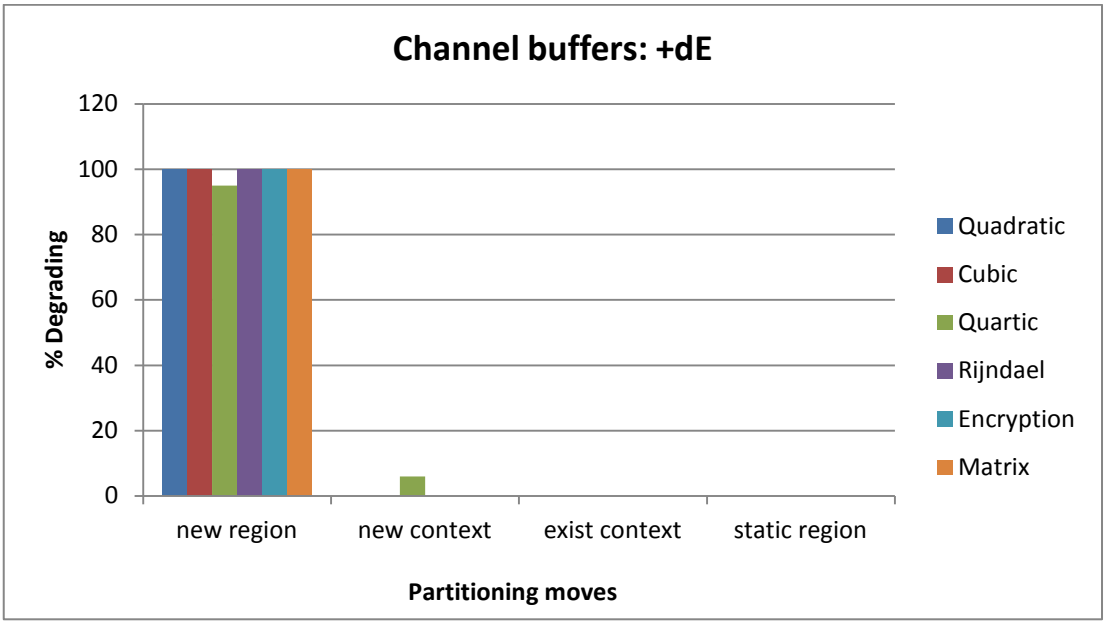
Modules which are unrelated to any other in the destination context do not improve the width of the channel interface. This can be explained by the mapping of the cutset onto a single bi-directional channel; transferring a module from one reconfigurable region to another has no effect on its width. Should the destination region encapsulate the complete execution hierarchy than this would mean that no signals are cut by the partitioning. Of course the cutset would have to define the channel in terms of its length and/or width to have an impact on an improvement to its buffer utilisation. Inspection of the graph indicates that those circuits which gain from such a move all feature multiple subprogram hierarchies, as exemplified by the equation solvers.

The second explanation for the improvement to the channel buffer metric concerns the length of the channel itself, irrespective of the data-dependencies between the subprogram modules which use it. Specifically, improvement may be made to the length of the channel by moving subprogram modules closer to one another, reducing the number of regions crossed by the channel and consequently the number of channel buffers required to interface them.

The remaining move to improve the channel buffer metric requires the creation of new circuit contexts. Again, the results depicted in figure 6.4 (a) show that the source of the modules moved during partitioning is to be found in the static region.



(a)



(b)

Figure 6.4: Circuit partitioning for channel buffers set to high priority.

Once more, improvement in the buffer metric is due to the module being transferred closer to its relative, however, in this case, the relative(s) may also be re-assigned to the new context in

the process − depending upon whether the hierarchy is clustered or partitioned across the region's contexts.

Unlike the other partitioning moves, the creation of a new region has no improving effect where a single communication channel is targeted, on the contrary it is the single significant source of degradation to the channel buffer metric (as shown in the figure 6.4(b)); the creation of a new region can only serve to lengthen the channel. Although creating new regions is primarily targeted to relieve existing regions which exhibit a high degree of context switching, it can act to reduce the length of multiple channels: its effect is to alter the placement of a region, reducing the number of regions crossed by the channel and in doing so, its length i.e. a module assigned to the static region wishing to communicate with another two regions away to its right-hand side, would have to cross through the first region − unless it was placed to the left-hand side of the static region.

To date, the effect of each type of partitioning move upon a given cost function metric has been examined in isolation to the others. The motivation for doing so was firstly to determine the moves which bring each metric closer to its target − obviously useful if partitioning with a single priority in mind. However, the likely scenario for partitioning is one where all metrics are required to be minimal − requiring a number of trade-offs to be made among each. The next step is to examine the behaviour underlying such decisions, where it will be shown that the properties of the exemplar circuits bias the outcome of certain cost function trade-offs towards the principle characteristics of each of the metrics presented earlier.

Table 6.1 summarises the trade-offs between the reconfiguration overhead and circuit area metrics; two partitioning moves are quantified: the first is the reduction in circuit area achieved through the creation of new circuit contexts which are swapped over a reconfigurable region, versus the penalty associated with swapping those circuit contexts. The second addresses the degree of swapping by assigning those modules previously swapped with one another to the same circuit context, although at the expense of an increase in the circuit area and the dimensions of the region required. A trade-off is necessary because the effect on the circuit area and reconfiguration overhead for each move is opposing: an improvement in one metric at the expense of a degradation to the other. The results of the circuit partitioning are expressed as a ratio of improving to degrading moves associated with

each area and reconfiguration trade-off. Shown alongside each trade-off are the original values for the metrics associated with a given subprogram partitioning; all target criteria are minimised to zero.

Cost function			Area vs Reconfiguration Trade-offs		Partitioning Results	
Circuit	Area Priority	Reconfiguration Overhead Priority	New contexts A(-dE):R(+dE)	Existing contexts A(+dE):R(-dE)	Area (CLB Slices)	Reconfiguration Overhead (ms)
Quadratic	High	Low	20:1	2:1	1880	5.1
	Low	High	1:12	1:30	4190	0
	High	High	44:1	2:1	1880	5.1
Cubic	High	Low	4:1	2:1	6026	3.29
	Low	High	1:4	1:20	8965	14.4
	High	High	2:1	3:1	6018	3.22
Quartic	High	Low	3:1	8:1	7792	36.4
	Low	High	1:4	1:12	15745	8.72
	High	High	1:1	4:1	8392	36.4
Rijndael	High	Low	29:1	1:1	5626	122
	Low	High	1:3	1:24	5927	0
	High	High	1:2	2:1	5709	6.83
Encryption	High	Low	1:1	2:1	9241	25.0
	Low	High	1:3	1:94	9711	0
	High	High	1:2	1:1	9344	0.60
Matrix	High	Low	1:1	18:1	3483	5.62
	Low	High	1:2	1:30	7324	0
	High	High	25:1	3:1	3491	5.67

Table 6.1: Contrasting the trade-off between circuit area and reconfiguration overhead.

The effect of prioritising a given metric is clearly shown. For instance, partitioning the quadratic equation solver with a high priority assigned to the circuit area metric favours the creation of many circuit contexts: for every move that has a degrading effect on the reconfiguration time, there are 20 which are beneficial to the circuit area. Similarly (although less dramatically), for every move that reduces the reconfiguration overhead by targeting an existing context, there are twice the number which increase the circuit area. The effect is shown on the outcome of the partitioning, where a bias towards circuit area results in a circuit of 1880 Xilinx Slices [6]. Partitioning in favour of the reconfiguration overhead prevents any

reduction in circuit area and incurs no reconfiguration overhead. The bias is evident in the other exemplar circuits albeit in different proportions.

Of particular interest is the outcome when equal weight is given to the cost function metrics: the results indicate that partitioning favours the area metric. In some cases, such as the Quadratic, Cubic and Matrix circuits, the outcome is the same or very close to what it would have been had partitioning to reduce the circuit area been the highest priority. The motivation for quantifying the effect as a ratio now becomes clear because in those cases where the results are the same, the ratios of circuit area to reconfiguration overhead reveal that the cost function acted more often towards reducing the circuit than reconfiguration delay.

For all other circuits, the result is more of a compromise between the metrics. In these cases, the reconfiguration overhead tends to offer more resistance to the swapping of the subprogram modules. For example, reducing the circuit area through the creation of new circuit contexts is less effective in the Quartic equation solver than it is in the Cubic and Quadratic circuits. This is because there are a similar number of moves favouring both the reconfiguration and area metrics; this is not the case regarding the other equation solvers. The effect is also repeated in the encryption circuits, where the ratios are reversed to reject half as many moves which would otherwise have been accepted.

An explanation for the resistance offered by the reconfiguration metric is to be found in the characteristics of the circuits themselves. The length of the Quartic equation solver, in terms of the number of module calls along the critical path is greater than the other equation solvers; in fact, the Quadratic equation solver is present as a sub-module. As a consequence, the effect is a greater reconfiguration overhead due a high degree of swapping among the circuit contexts. A similar effect occurs in the encryption circuits because of the presence of the finite loops: any swapping within a loop is magnified by number of cycles it performs - in the case of encryption circuits this can be as many as 104 times.

Recall that unlike the area and reconfiguration metrics which are minimised to zero, the channel metric can be satisfied using a target device with more resources than are actually required by the channels. This does not happen frequently, as the target device tends to get smaller in terms of its resource capability depending upon how successfully the circuit area is reduced through partitioning. Three-state buffers are also utilised during data-path

optimisation to implement the multiplexors which permit the sharing of the functional units. At some point, there could exist a resource conflict caused by allocating the buffers for optimisation and therefore reducing the number available for implementing the communication architecture. To examine the effect this could have on the partitioning metrics, the channel target was specified as a percentage of the available buffers resources; the results are presented in Tables 6.3 to 6.7.

As was expected, a tighter constraint placed upon the buffer resources had an impact on their usage during circuit synthesis. This can be seen in the ‘channel buffers’ column in each of the tables. However, a secondary effect occurred in the formation of the regions: the tighter the target, the fewer regions were created during partitioning. The reader will recall that the greatest form of degradation experienced by the channel metrics (see Figure 6.4(b)) resulted when the creation of a new region had taken place. This makes sense because each new region serves to lengthen the channels – the more boundaries there are to cross, the greater the utilisation of three-state buffers.

The ramification of reducing the number of reconfigurable regions available for context switching of temporal partitions is a gradual increase in the overall circuit area. A positive upshot of this is a reduction in the reconfiguration time for a given region; the partitioner is less likely to compensate by increasing the number of temporal partitions for the remaining regions. In the majority of the circuits, the effect can reduce the number of regions by half. At one percent of the available resources, the degradation to the channel metric can be sufficient to dominate the decision making and reject all potential moves taken during partitioning. When this effect begins to occur would appear to be dependent upon the characteristics of the circuit themselves. Table 6.2 shows the variation among the size of module nets for each of the circuits.

The actual point at which the constraint becomes effective would appear to be related to the variation in the size of the circuit’s nets. For instance, the matrix circuit appears not to respond until given the tighter constraint of 50% of available buffer resources; it exhibits the greatest variation in module net widths (Table 6.2).

Circuit	Standard Deviation (subprogram population)
Matrix functions	125.4
Quartic equation solver	37.9
Encryption/Decryption	19.5
Cubic equation solver	17.9
Rijndael encryption	16.9

Table 6.2: Variation among the module nets for each exemplar circuit.

The next circuit to be most affected by the limited resources is the Quartic equation solver (33%) which also has a large variation among its module nets. The remaining circuits each undergo the effect at 10% of their accessible buffer resources. The usefulness of being able to relate the characteristics of a circuit in terms of buffer resources may assist the designer in determining the proportion of resources to allocate to circuit optimisation and channel implementation prior to synthesis.

Cost function				Partitioning Results					
Area Priority	Reconfiguration Overhead Priority	Channel Buffer Priority	Target device Buffer Utilisation (%)	Buffer Target Met (%)	Area (CLB Slices)	Reconfiguration Overhead (ms)	Delay	Freq	Channel Buffers
High	High	High	100	79.2	3483	5.67,2r,S	194.5	11.8	1152
			50	46.2	3483	5.59,r,S			1152
			33	42.1	3483	5.59,r,S			1152
			25	37.7	5179	4.83,r,S			832
			10	0	7324	0			0
			1	0	7324	0			0

Table 6.3: Matrix functions.

Cost function				Partitioning Results					
Area Priority	Reconfiguration Overhead Priority	Channel Buffer Priority	Target device Buffer Utilisation (%)	Buffer Target Met (%)	Area (CLB Slices)	Reconfiguration Overhead (ms)	Delay	Freq	Channel Buffers
High	High	High	100	100	6026	3.29, 3r,s	43.5	22.37	384
			50	100	6026	3.29, 3r,s			384
			33	100	6026	3.29, 3r,s			384
			25	98.2	6026	3.29,r,s			384
			10	34.5	6178	2.64,r,s			192
			1	11.5	14086	0			0

Table 6.4: Cubic equation solver.

Cost function				Partitioning Results					
Area Priority	Reconfiguration Overhead Priority	Channel Buffer Priority	Target device Buffer Utilisation (%)	Buffer Target Met (%)	Area (CLB Slices)	Reconfiguration Overhead (ms)	Delay	Freq	Channel Buffers
High	High	High	100	100	8392	36.4,3r,S	51.6	18.34	1152
			50	97.5	8392	36.4,3r,S			1152
			33	85.2	8415	35.7,2r,S			768
			25	77.4	8815	31.5,2r,S			768
			10	58.3	9784	22.7,r,S			384
			1	10.9	16021	0			0

Table 6.5: Quartic equation solver.

Cost function				Partitioning Results					
Area Priority	Reconfiguration Overhead Priority	Channel Buffer Priority	Target device Buffer Utilisation (%)	Buffer Target Met (%)	Area (CLB Slices)	Reconfiguration Overhead (ms)	Delay	Freq	Channel Buffers
High	High	High	100	100	5709	6.83,2r,S	4.54	16.31	256
			50	100	5709	6.83,2r,S			256
			33	100	5709	6.83,2r,S			256
			25	95.5	5709	6.83,2r,S			256
			10	55.5	5709	6.15,r,S			128
			1	16	5927	0			0

Table 6.6: Rijndael Encryption/Decryption.

Cost function				Partitioning Results					
Area Priority	Reconfiguration Overhead Priority	Channel Buffer Priority	Target device Buffer Utilisation (%)	Buffer Target Met (%)	Area (CLB Slices)	Reconfiguration Overhead (ms)	Delay	Freq	Channel Buffers
High	High	High	100	100	9344	603.06us,4r,S	4.41	7.3	384
			50	100	9344	603.06us,4r,S			384
			33	100	9344	603.06us,4r,S			384
			25	100	9344	603.06us,4r,S			384
			10	67.5	9337	589.03uS,2r,S			256
			1	28.5	0	0			0

Table 6.7: Encryption/Decryption.

6.3 Test Circuits

The selection of test circuits was done to explore a number of aspects relating to the theme of this thesis: partitioning across subroutine boundaries and their preservation at different levels of abstraction, ultimately at the device-level for run-time reconfiguration.

As described in Chapter 3, MOODS HLS has been used to investigate many different aspects of circuit synthesis and as a result there exist several exemplar designs which were available to the author to conduct the experiments described within this chapter. The motivation for this approach is that all designs were not specifically written with run-time reconfiguration in mind and as a consequence, there was no inherent bias in how they were coded. Before commenting further on the relevance of their selection, one common characteristic which is inherent to their selection is how they were coded, specifically in the behavioural-style of VHDL described in Chapter 3.

As the reader will recall, the behavioural approach to circuit synthesis requires the designer to view a specification as a set of independent tasks, each of which is implemented as a VHDL parallel ‘Process’. Unlike RTL synthesis, the operations of each process/task is automatically scheduled across multiple clock cycles; as a consequence each task can be further partitioned by relating sequential behaviour of the operations as VHDL ‘Procedures’.

With reference to the test circuits, all were described in this way and the author ensured that none of the procedures were ‘inlined’ in order to preserve the functional partitioning inherent in the way that each were coded.

The reader is referred to Appendix B, where the relevant properties of each test circuit are presented. As described in Chapter 3, MOODS represents a design at different levels of abstraction. One convenient representation is the ‘module’: at this level of abstraction there is a one-to-one mapping between each VHDL procedure and an equivalent internal module but a many-to-one relationship between the VHDL processes and the single ‘Program module’. The characteristics of each test circuit will now be considered in terms of their equivalent module representation.

A table of characteristics is shown for each design which provides the reader with an appreciation of the composition of each example: the number of modules, their resource use and port widths are tabulated for comparison. Shown below each table is a task graph, where every node corresponds to a module and each edge represents its execution relationship with another and is labelled with the size of the control and data-path dependency in bits. The last characteristic is the module execution sequence: each sequence is presented in the form of one possible path through there circuit; where appropriate, modules repeated inside finite loops are labelled with an iteration count.

Having introduced the test circuits, it is necessary to consider their influence during the experiments. On inspection of their characteristics, what becomes apparent is that the equation solvers feature multiple levels of module hierarchy which appear frequently on different paths through each circuit; in contrast the encryption and matrix circuits do not: their modules can be characterised by their execution within finite loops.

Characteristics such as these present the temporal partitioner with different choices: nested modules have control and data-path dependencies; context switching them requires intermediate storage of their control and data-path tokens. In response, the partitioner has the choice to not swap them over the same resource and cluster them in a single partition or to partition them across multiple resources; either of these choices will impact on the use of communication channels in the architecture: clustering them might delete an existing channel, partitioning them over multiple resources could change the topology of the partitioning,

necessitating the creation of a new buffer interface in the resource, as well as the potential to widen the width of the channel.

The Encryption and Matrix circuits present the partitioner with a different scenario: their independence to one another simplifies their context switching in which case the loop iteration count becomes the deciding factor. In addition to deciding what to do with modules inside the loops, the partitioner will be presented with the choice of using their execution to hide the reconfiguration of those modules executed outside it.

The presence of other forms of control, such as conditional instructions can also be exploited by ‘prefetching’ a module’s configuration before knowing the outcome of the conditional instruction and whether the module is to be executed. In such circumstances, the partitioner must consider the limiting effect which a reconfiguration prefetch may have on the reconfiguration scheduling of any other module on a mutually exclusive branch; two modules cannot be reconfigured at the same time, one must take precedence!

The advantages of performing temporal partitioning in a HLS tool become apparent when returning to the area characteristics shown in the tables; the modules require many resources because there are numerous opportunities for data-path allocation and instruction scheduling at the operation level of abstraction. Data-path sharing has a ‘spatial’ effect on temporal partitioning: the size of a module may prevent its binding to a particular resource, only to be accepted at a later stage in optimisation having been reduced in area by a data-path optimisation. Similarly, a scheduling transform may also effect the binding of a module to a reconfigurable resource: the scheduling of those operations which read the arguments passed to a called module determine the width of a channel; scheduling them to individual control steps would permit their parameter arguments to share a channel, scheduling them to the same step forces all of them to be concurrently available through the channel.

6.4 Summary

This chapter presented the results obtained through experimentation using the temporal partitioning transform, following its integration into the existing simulated annealing-based optimisation approach taken by MOODS. When used in this way, it enables architectural

synthesis to simultaneously alter the circuit structure at two different levels of abstraction: at the operation or instruction-level using scheduling and allocation transforms and at the module-level using temporal resource binding of reconfigurable resources.

Being a general optimisation algorithm, simulated annealing enables a multi-objective approach to temporal partitioning. In doing so, it permitted analysis of the relationships between the new transform and the area, reconfiguration and channel metrics described in Chapter 4. As with the operation-level transforms, the partitioning transform has ‘on paper’ a distinct effect upon each of these metrics; their relationships have confirmed through experimentation and will now be briefly summarised with reference to the characteristics of the test circuits:

Applying a higher priority to one metric at a time and performing optimisation did give preference to reducing the characteristic associated with the metric: for circuit area, the temporal partitioning algorithm acted to reduce the circuit area by favouring the creation of new temporal partitions over fewer reconfigurable regions. As expected, this was done to the detriment of the reconfiguration and channel metrics which were set to a lower priority. Modules moved between the partitions were in general as likely to reduce or increase the circuit area without regard to the type of circuit and static binding to a reconfigurable resource did very little to improve circuit area – as expected.

The anticipated effect was also observed when the reconfiguration metric was given the highest priority in the cost function: very little reduction in circuit area was permitted and in circuits which exhibited finite loops, no reduction was achieved at all.

With regard to assigning a high priority to minimising the channel buffer metric, the module interface characteristics of those circuits with the smallest variation in port characteristics, were the most tolerant in not rejecting moves which might have improved the other metrics. When the channel metric did take effect, it acted as expected to prevent new regions from being reconfigured, supporting only moves which re-used existing buffer interfaces or removed them entirely by a static binding of the modules concerned.

The final cost function metric was of particular interest because by choosing to set all metrics to the same priority, the effect of their contradictory goals could only be obtained through

experimentation. Unlike the other cost functions, the results were influenced by a particular characteristic among the test circuits: the critical path derived from the longest module execution sequence.

For circuits with large small execution sequences, such as the Encryption and Quartic equation solver, the reconfiguration overhead is the dominant metric and supports the clustering of modules in existing contexts. On inspection of their execution sequences all feature either high loop iteration counts (104 for the encryption circuit) or in the case of the Quartic equation solver, a critical path with many module execution calls.

Examination of the execution sequences of both the Matrix and Quadratic equation solver circuits also support this explanation: the Matrix circuit has a loop count of 4 iterations and the Quadratic equation solver − a very short critical path. In both these cases, the area metric is dominant in the cost function and as a consequence favours the creation of new partitions. The only test circuit which would have been difficult to predict is the Cubic equation solver whose critical path appears to be neither short or very long, but short enough to support the area metric and the reduction in area using temporal partitioning; in doing so, it emphasises the importance of using a HLS tool to examine the trade-offs, since these could only have been determined through experimentation.

Chapter 7

Run-time Reconfiguration − A Case Study

This chapter presents a practical application of the work described to date. Specifically, it details the implementation of a run-time reconfigurable coding system. In doing so it exemplifies the stages inherent to synthesising a reconfigurable system − from an algorithmic description of the circuit behaviour, to an optimised RTL description incorporating the structures necessary to facilitate run-time self-reconfiguration. It concludes with an implementation of the synthesised circuit on a commercial partially reconfigurable FPGA.

7.1 A Run-time Reconfigurable Variable Coding System

7.1.1 Background

Error correcting codes are used to tolerate data corruption which can occur during the transmission of information in a communications channel. Encoding the message data prior to its transmission appends redundant parity bits, allowing a decoder on the receiving end of the channel to detect, locate and subsequently correct the erroneous data. A wireless channel is very sensitive to signal distortion. One such example is channel fading, where a signal can undergo a reduction in strength due to propagation effects such as reflections caused by the atmosphere and land between the transmitter and receiver. For a fixed wire system, a single code can be selected to correct up to the maximum number of errors that can be expected. In fact, ‘expected’ is an appropriate way of describing such an approach, as the characteristics of a fixed channel are more readily predictable than a wireless one. It is this variation in error predictability which would increase the inefficiency of the channel should a fixed code be adopted. During periods of weak fading, there would be fewer errors to correct and the full

corrective ability of a given code would not be required. Worse still, the extra parity bits could have been data bits. What would be desirable is a means of adapting the coding to suit the conditions of the channel rather than a ‘one code corrects all’ approach.

A solution is to encode the data using a variety of codes, each one selected by the receiver based upon a prediction of what the future conditions of the channel might be. The transmitter is not directly in a position to determine the level of corruption (without feedback) as it does not know the conditions of the channel before transmitting the data. The receiver on the other hand can attempt to second guess these conditions based upon a recent past history of the channel.

The variable coding scheme described can be entirely implemented in hardware without the use of dynamic reconfiguration. Generic encoder and decoder circuits can be parameterised to implement each of the codes at a modest increase in the hardware overhead (in comparison with their dedicated counterparts).

From the designer’s point of view, the task is to determine the maximum number of errors which will have to be corrected and select a means of correcting up to that number efficiency. This requires a trade-off between the number of errors that can be fixed and the number of additional parity bits required to detect the errors. Of equal importance in the selection of the coding scheme is the area and delay metrics of the encoder and decoder circuits, both of which increase in magnitude in relation to the corrective ability of the coding used.

One approach used to increase the decoding rate is to introduce a degree of concurrency in the decoding process, partitioning the computational workload over a number of decoders. Of course, in addition to the communication overhead required to distribute the computation among the decoders, the main drawback can be the greatly increased circuit area. It is not immediately obvious how parameterisation can be used to increase the number of decoders during circuit execution. With the exception of reducing the power consumption, there is little advantage to not fully utilising all the device resources in switching from one code to the next, therefore fixing the maximum degree of parallelism for any coding scheme. It is in this context that the role of dynamic reconfiguration is examined, where device resources committed to realising n decoders for a given code are reprogrammed to implement m decoders of any other type. Furthermore, it is possible to target different resources in the

implementation of each decoder. A large memory required during the decoding of one code can be realised using the on-chip RAM or look-up tables at the cost of at least two read cycles per datum (one cycle to set-up the address and another to read the output). Alternatively, the small memory consumption necessary in decoding another code can be directly implemented using registers with single read cycles for all data simultaneously. These are examples of trade-offs taken by a designer during the specification of a circuit. There are also many trade-offs which are automatically taken during the partitioning and optimisation of the circuit. For instance, how might the operators of a circuit with n decoders and another with m be assigned to control states to efficiently share the same resources at different times during their execution. In this way, a pair of decoders with a large resource footprint can be swapped 'on the fly' with eight smaller decoders, in response to a change in the coding scheme.

As the reader will by now appreciate, this comes at the cost of a reconfiguration overhead. The benefit of adapting to the characteristics of the communication channel would be overshadowed if the reconfiguration took too long and/or was performed too often. The problem becomes a task suited for synthesis. In addition to optimisation at the operation-level, another approach to reducing the impact of the reconfiguration delay would be to overlap the decoding of codewords alongside the reconfiguration of the decoders themselves.

This approach implies introducing a second region with the reconfigurable resource to enable the decoding to respond to a change in the coding scheme in two stages. The first region continues to perform the decoding of the current coding scheme used, whilst the second is partially reconfigured with the decoders associated with the next scheme. Upon completion of the reconfiguration, the responsibility for decoding the new scheme is transferred to the second region. The decoders associated with the former scheme can remain inactive on silicon in the event of a switch to the former coding or partially reconfigured with a set of decoders to match the current scheme, thus further multiplying the degree of decoder parallelism. Once again, this can be achieved without any interruption of service to the region not affected by reconfiguration.

Ordinarily a duplicate approach might be viewed as simply being redundant. However, when the target architecture is the FPGA the choice of which memory resource to use for the codeword and weight tables [123] of the decoder can result in the exclusion of other forms of

potential memory resource. Utilising resources that would otherwise remain un-programmed in the target device provides the opportunity for the synthesising a sequential decoder which provides the execution that enables the overlapping of codeword decoding with the reconfiguration of a more specialised Viterbi decoder.

Figure 7.1 illustrates the conceptual building blocks of the coding system. At this stage of synthesis, the emphasis is on the behaviour of the system. It should be regarded as the sum of two halves. One half is responsible for encoding and transmitting a message comprising several lines of ASCII characters, whilst the other corrects any errors that may have occurred during transmission prior to displaying its contents on a VGA display. The *transmitter* and *receiver* are bridged by a number of control and data signals used to coordinate and transfer the message. For the sake of argument, errors are introduced solely to the message and not to the controlling signals.

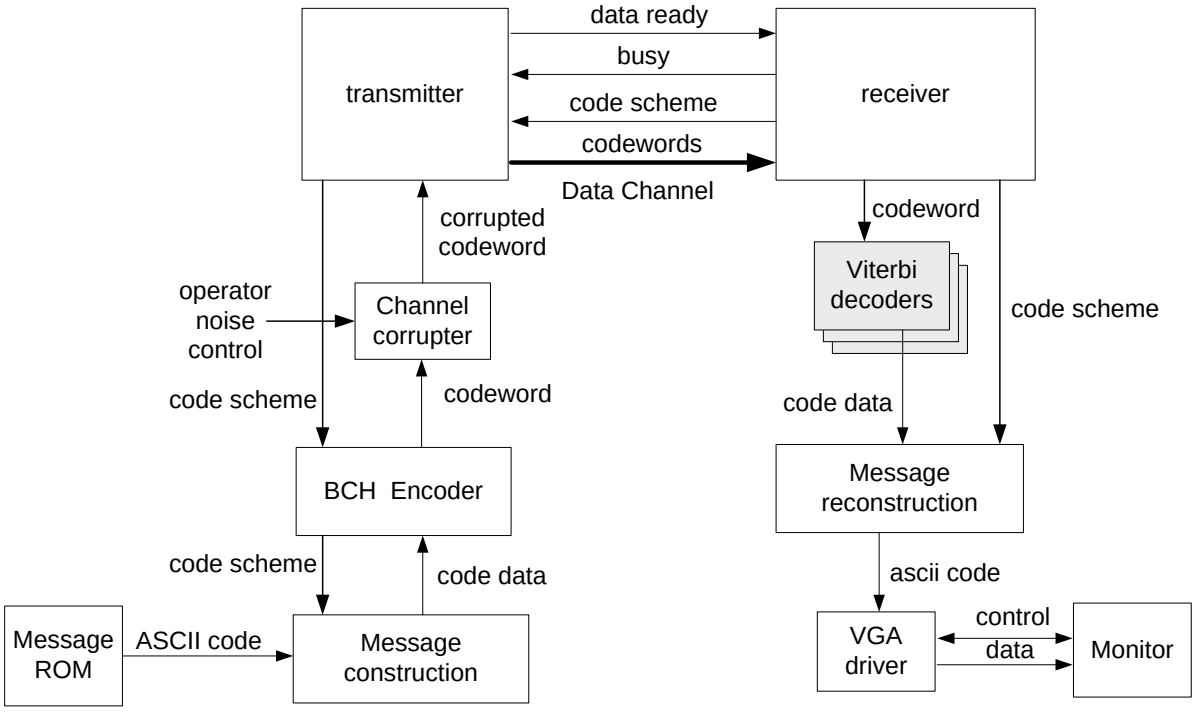


Figure 7.1: A variable coding system.

Every character of the message is converted into its ASCII equivalent and spliced to form a data packet, the length of which is predetermined by the coding used. Each data packet is encoded in accordance with the present coding scheme to form a binary codeword. The effect

of a noisy channel is approximated by randomly selecting a codeword and a number of bits to invert. This occurs either autonomously through the channel corrupter or under manual control of the operator who may wish to manually demonstrate the characteristics of the system.

At the other end of the channel, each codeword is decoded and any bits found to be erroneous are corrected in accordance with limitations of the coding scheme. The coding is altered in response to the number of corrupted bits encountered during the decoding of the ten most recent codewords. This sample forms a measure of the quality of the channel and is used to steer the selection of the coding to achieve a level of service that lies within the bounds of a predictable probability of error. The receiver initiates a change in the coding scheme, instructing the transmitter to alter the construction and encoding of the codewords whilst simultaneously reconfiguring the decoders and parameters for message reconstruction. When there are a sufficient number of reconstructed ASCII characters the display is updated with the latest message. In appendix C, the behaviour of each of the building blocks is examined in greater detail.

7.2 Variable Coding Strategy and Run-time Reconfiguration

The effect of partitioning the Viterbi decoding over n Processors is a practical scaling of the decoding time with the number of processors used [124,125], which of course never matches the theoretical scaling due to the external communication costs involved in updating the states. It is perfectly feasible to realise the decoding using processors optimised for DSP applications. Despite the common use of RISC and superscalar architectures, single processors are unlikely to match the spatial computation offered by a dedicated circuit whether it be realised as an ASIC or using programmable resources, such as those found on the FPGA targeted in this chapter.

Through the use of partial reconfiguration, it is possible to change the number of Viterbi processors working on decoding a stream of codewords. There is no doubt that changing the coding scheme to suit the conditions of the communication channel would increase the amount of information that could be transmitted. There is also no doubt that increasing the

number of processors can also significantly reduce the decoding time. Where there may be doubt is whether this would be a good application for changing the number of decoders on-the-fly through partial reconfiguration of the device. Why not simply synthesise a decoder with *n* processors that is re-parameterised to decode a range of BCH codes? At one moment it could be decoding a series of BCH (15,11,3) codewords over 4 processors, at another when there is a change in the noise of a channel, it might be decoding less information through BCH (15,5,7) codewords but still operating despite the change in the conditions of the channel. A drawback with doing this is to be found in the resource requirements of the Viterbi decoder itself. There are 64 times more states to be updated in the decoding of a BCH (15,5,7) codeword in comparison with the BCH (15,11,3). The memory requirements for each state are 15 bits for each copy of the message and a further 4 bits for the hamming distance weight.

It is also necessary to synthesise two copies of each weight and message table since some of the predecessor states are used to update more than one state and it would be wrong to update a state before it is used again in the current cycle to update another. The resource requirements are therefore 38912 registers – not an insignificant number. That is assuming a single decoder, multiple decoders although a division of that number collectively require the same memory across all decoders. However, there is also the issue of how the memory is shared and whether additional memory is required when transferring data from one processor to another.

Irrespective of how the communication is implemented, there will always be a number of redundant registers when the coding scheme switches to a less robust form of error correction, the same number of decoders required to decode one scheme are also present in another. Put another way, it is uncertain how 2 processors allocated to decoding the BCH (15,5,7) scheme could be re-parameterised to implement the 32 processors that could be implemented using the same number of registers.

This is where partial reconfiguration can be used to employ the unused resources which otherwise would not be used. More specifically, a library of different decoders can be synthesised, each of which have similar resource characteristics but most importantly − with different numbers of decoders. This concept can be further extended to synthesise different

versions of the same decoder. The variation in characteristics might manifest itself through changes to how the memory is implemented. The robust decoder requires many states and consequently a larger circuit area to implement when targeting registers, each of which has a number of separate control lines. At an increased cost to circuit delay, the weight and message tables could be synthesised from look-up tables or dedicated device memory blocks. The same approach is not suitable for the smallest decoder, where 16 states consume 608 registers, a number readily available on even modest-sized FPGAs. Therefore, it is able to benefit from single cycle read or write times, in comparison with the 2 cycles it takes to read a location in Xilinx BlockRAM [6].

Alternatively, the resources could be committed to performing another circuit function, responding perhaps to some infrequent event that would justify the need for a temporary down-sizing of the decoder, in terms of its degree of parallelism. In either regard, this leaves a degree of leeway in determining the number of processors and how their circuit structure might be optimised.

We believe this is a good case study for synthesising reconfigurable logic for the following reasons:

- Hardware parallelism – the numerous concurrent resources available within an FPGA is well suited to an application that can exploit such parallelism such as the Parallel Viterbi decoder.
- Coarse-grain reconfiguration – a feature of current FPGA technology can sometimes restrict the type of application that can exploit it. In this case, a Viterbi processor is a coarse change in circuit structure and therefore suits the limitations of current commercial devices.
- Hardware acceleration, virtual area and hardware adaptation – are prominent areas of research in reconfigurable systems. Hardware acceleration is exemplified in this application which occurs in proportion to the number of processors added through partial reconfiguration. When this is done with regard to the operating conditions of the channel, it seeks to adapt the coding scheme – parameters and underlying structure. Through partial reconfiguration, the total resources required are much

smaller than the sum of each of the constituent sub-systems and hence the system exploits a degree of virtual area.

- Communication channel synthesis – although partitioning is determined by the method used to split the number of decoder states among the processors [124], synthesis is used to determine the characteristics of the channels which connect the Viterbi processors to one another.
- Circuit optimisation – is used to generate a library of circuits which are synthesised to create structures with different area/delay characteristics. This can occur alongside the synthesis of the communication channels.
- Exercise the reconfigurable infrastructure – most crucially, it demonstrates how circuits can be plugged in and out of the architecture and that the independent reconfigurable regions correctly communicate with each other.

7.3 System Architecture

7.3.1 Adaptive Coding Scheme

In addition to the transmission of a continuous stream of codewords which will eventually be re-constructed into information at the receiver, the principal motivation of the communication systems is the requirement that the encoding of the information be responsive to the conditions presently experienced in the channel. Moreover, through constant monitoring of the level of noise experienced on the channel, the coding scheme is adapted to maintain an approximate level of service with regard to the bit-error rate. What follows is a brief formulation of how the level of error relates to the selection of the coding scheme.

Recall that because the decoder can return the number of errors corrected (within the limitations of the BCH code), the receiver is in the best position to monitor the severity of corruption which must have occurred to the codewords during their transmission. Therefore, the responsibility of selecting the coding scheme is left to the receiver. It does this by dividing the stream of decoded codewords into groups of eight codewords or 120 bits; their size was chosen to enable a comparison with a non-reconfigurable Sequential Viterbi decoder [123] also synthesised using MOODS. As described in appendix C, inspection of the weight associated with state R_{14} after the last bit of the 15-bit codeword has been decoded, reveals the

number of errors that were corrected. Its summation of the eight codewords decoded can be used to return an approximate measure of the probability of error P_e over the group of eight codewords. For example, the probability of a 1-bit error in the group of codewords would be $P_e = \frac{1}{120} = 0.0083 = 0.01\ (2\ \text{dp})$ or 1%. The next point of consideration is to relate the probability of error P_e of the group of codewords to the corrective ability of an individual BCH code. In other words, how many errors in the group of codewords does it take for each BCH code to fail to meet *n* % of bit errors. Once this is determined, so too are the conditions for adapting the coding scheme to the approximate measure of errors on the channel.

If P_e is the probability of error on the channel, the probability of there being more than *n* errors occurring on the same channel is given by:

$$P_n = 1 - (P_0 + P_1 \ldots + P_n) \quad (7.1)$$

$$P_n = 1 - (15.C_0.P_e.(1 - P_e)^{15} + 15.C_1.P_e.(1 - P_e)^{14} \ldots + 15.C_n.P_e.(1 - P_e)^{15-n} \quad (7.2)$$

For example, the probability of there being more than one error (*n*=1) in a 15 bit codeword for a $Pe = \frac{1}{120}$ is found to be:

$$P_1 = 1 - (P_0 + P_1)$$

$$P_1 = 1 - \left(1.\frac{1}{120}.\left(1 - \frac{1}{120}\right)^{15} + 15.1.\frac{1}{120}.\left(1 - \frac{1}{120}\right)^{14}\right)$$

$$P_1 = 1 - (0.88203 + 0.11118)$$

$$P_1 = 0.01\ (2.\text{d.p})$$

Using equations 7.1 and 7.2, the probability of there being more errors than the corrective ability of each of the BCH code can cope with, is calculated with various probabilities of error in each group of eight codewords. The results are shown in Table 7.1. The first two columns indicate the relationship between achieving a certain percentage of errors on the channel and its relation to the cumulative number of errors in a group. This ranges from a 1 to 10% Bit Error Rate (BER). Alongside them are the probabilities that each BCH code will be unable to correct the codeword to meet the quality of service associated with the channel error P_e. The

number of bits in which each codeword is in error corresponds directly to the BCH code i.e. BCH (15,11,3) corrects up to and including 1 bit in a 15 bit codeword, BCH (15,7,5) corrects 2 and BCH (15,5,7) can correct up to 3 bits in error (inclusive).

No errors n/120 bits	Probability P_e (%) 1 bit *group* error	Probability (%) 1 bit *codeword* error	Probability (%) 2 bit *codeword* error	Probability (%) 3 bit *codeword* error
1	0.0083 (1.0)	0.01 (0.68)	0.00 (0.024)	0.00 (0.00)
2	0.0166 (1.6)	0.03 (2.53)	0.00 (0.181)	0.00 (0.01)
3	0.0250 (2.5)	0.05 (5.29)	0.01 (0.567)	0.00 (0.04)
4	0.0333 (3.3)	0.09 (8.76)	0.01 (1.247)	0.00 (0.13)
5	0.0417 (4.1)	0.13 (12.74)	0.02 (2.259)	0.00 (0.28)
6	0.0500 (5.0)	0.17 (17.10)	0.04 (3.620)	0.01 (0.55)
7	0.0583 (5.8)	0.22 (21.69)	0.05 (5.330)	0.01 (0.94)
8	0.0667 (6.7)	0.26 (26.41)	0.07 (7.378)	0.01 (1.49)
9	0.0750 (7.5)	0.31 (31.18)	0.10 (9.739)	0.02 (2.21)
10	0.0833 (8.3)	0.36 (35.92)	0.12 (12.39)	0.03 (3.12)
11	0.0917 (9.2)	0.41 (40.57)	0.15 (15.29)	0.04 (4.23)
12	0.1000 (10)	0.45 (45.10)	0.18 (18.41)	0.06 (5.56)

Table 7.1: Errors in eight codewords necessary to switch between each code scheme.

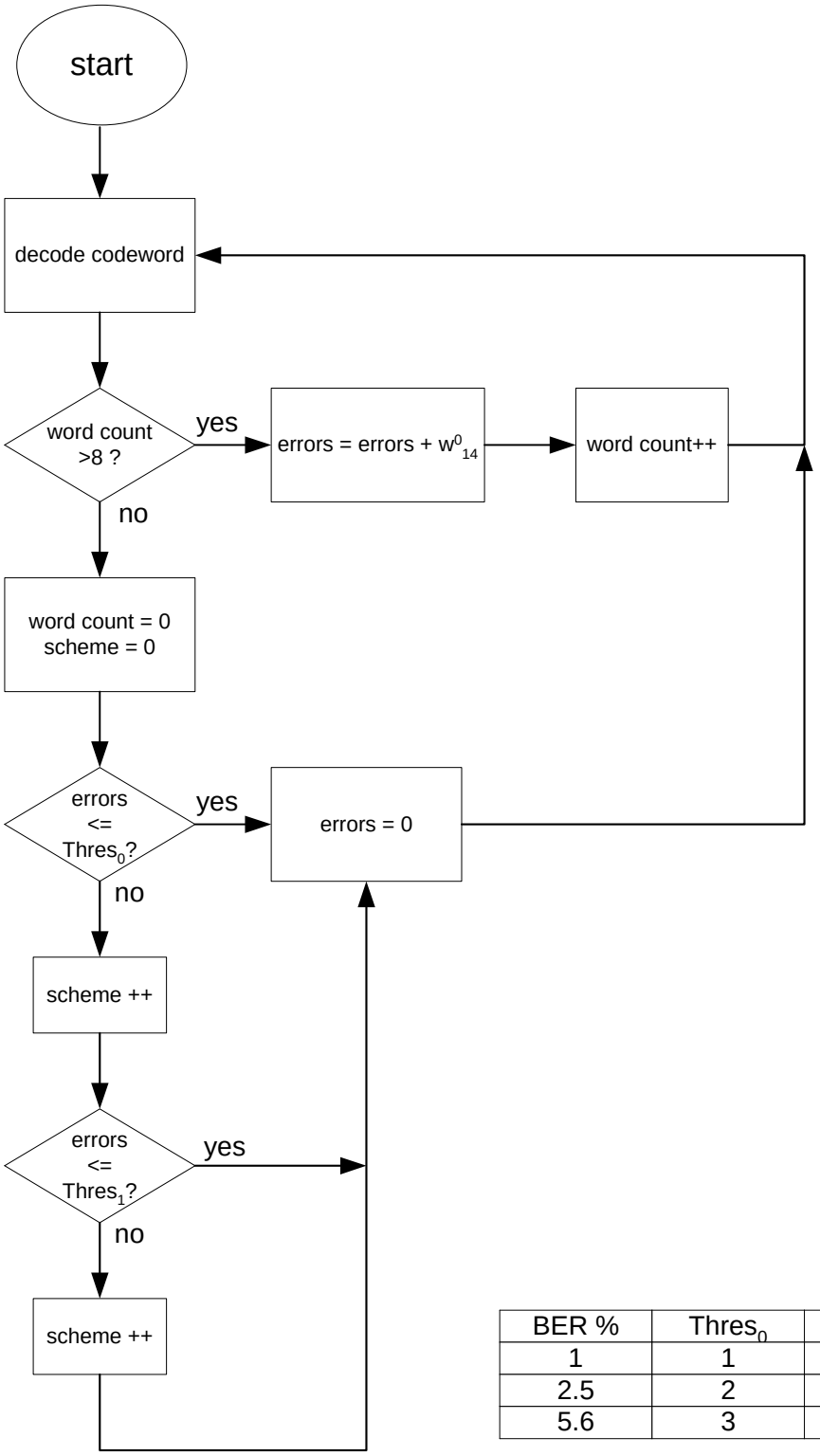
Using the results shown in Table 7.1, it is now possible to relate the number of errors received in a group of eight codewords to a threshold where one coding scheme fails and another must be adapted. The thresholds were selected to enable our work to be compared with a non-reconfigurable parameterised Sequential Viterbi decoder [123]. Assuming that that the BCH (15,11,3) coding scheme is currently deployed, should an approximate 5 % BER (highlighted in green) be a desirable operating condition for communication, no more than 3 bits in the group of 120 bits (eight codewords) can occur in error. Any less, the quality of service

exceeds the specification, any more, the quality of service cannot be maintained and the next BCH code (15,7,5) is chosen to encode and decode the message stream. There is quite a large tolerance (a further 4 bits) until the BCH (15,7,5) coding scheme fails, at 8 bits in 120, the receiver would request another step up in the level of robustness to the BCH (15,5,7) code. Of course, should the conditions of the channel improve, the receiver may request a step down in the level of correction and thereby increase the amount of data that could be encoded through each codeword.

Figure 7.2 illustrates the bit thresholds at which the coding scheme may be changed. Before this can take place, the group of eight codewords requires sampling to estimate the probability of error on the channel P_e. This occurs during the top half of the flowchart and simply requires that eight codewords are counted and the number of errors accumulated – recall that the number of errors corrected is given by the weight of state 0 after the last bit of the previous message bit has been decoded i.e. w_{14}^{0}.

Once the sample is taken, the count is reset for the next sample and so too is the coding scheme, since it is about to be updated. The threshold at which this happens is either $Thres_0$ or $Thres_1$ respectively. Their parameters are set according to the desired BER and the relationships are summarised in the table adjacent to the flowchart for some exemplar BER rates. The ascending numeric value of the scheme determines which BCH code is adopted, with the default scheme being most efficient code BCH (15,11,3). The remaining schemes are just an increment BCH (15,7,5) or two away BCH (15,5,3). After the scheme is selected, the number of errors and word count are reset and wait to be set during the next sample of eight codewords.

Ideally, the switching that occurs between the different coding schemes should not occur too often because as well as the additional time to change the parameters associated with the new scheme, there would inevitably be a reconfiguration penalty. Two solutions are adopted to address these issues. The first is to allow a wide enough band in which several errors need to occur in order for the scheme to switch up or down. This is as wide as 5 bits for the most robust code at 5 % BER. In addition to that, the coding scheme never changes from the smallest code to the largest and vice versa without the intermediate step of the middle BCH code. This aids in making the system less sensitive to short bursts in signal noise.



BER %	$Thres_0$	$Thres_1$
1	1	3
2.5	2	5
5.6	3	7

Figure 7.2: Flowchart showing BER driven selection of the coding scheme.

Secondly, as will be discussed in a later section of the chapter, at least two different decoders can be present in dedicated regions of the FPGA at the same time. This staged approach enables reconfiguration to occur in parallel to the decoding of the codewords.

7.3.2 Inter-process and Intra-region Communications

The backbone of the architecture is the inter-process communication between the transmitter and receiver regions and the intra-process communication between the receiver and concurrent Viterbi processors. The purpose of the case study is not to demonstrate a fully featured communication system, rather an application of the synthesised architecture and run-time reconfiguration. In this spirit, the entire communication is realised using a single FPGA device. Figure 7.3 illustrates how the main building blocks are implemented on the static and reconfigurable regions of the FPGA.

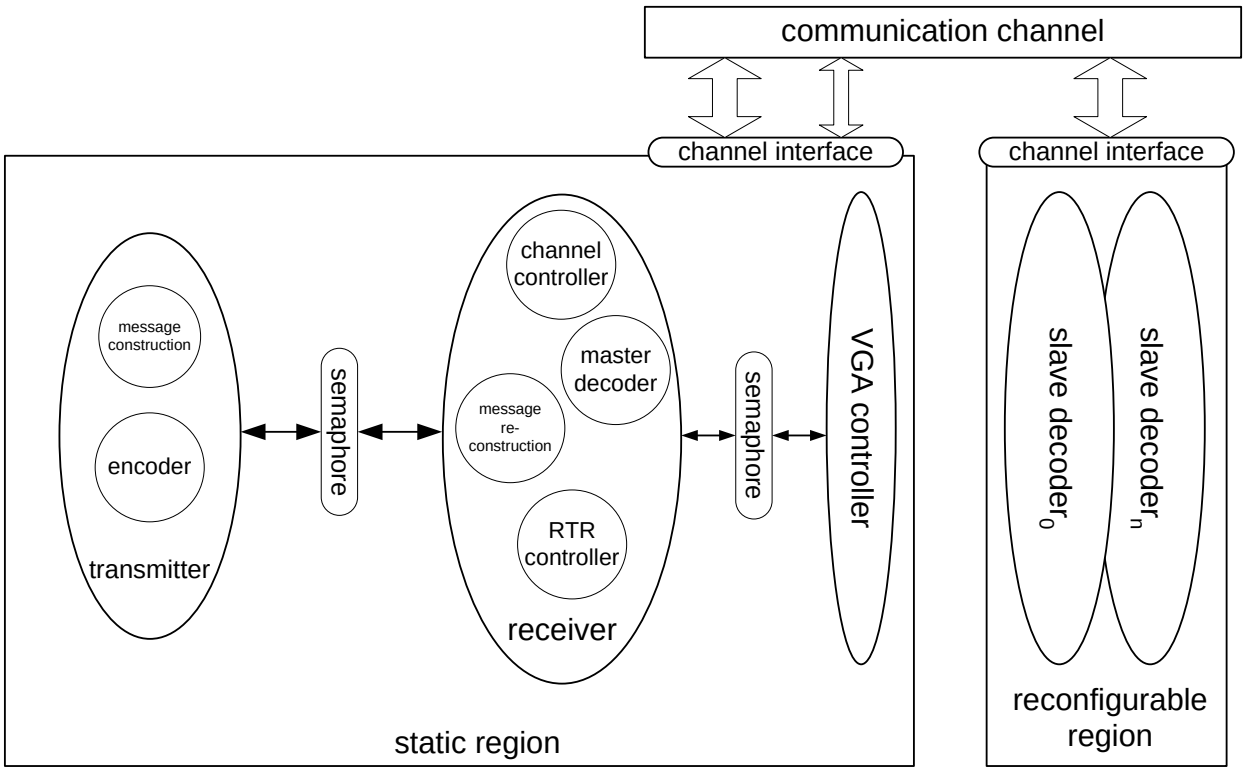


Figure 7.3: Floorplan for the RTR variable coding scheme.

Communication between the transmitter and receiver occurs on the same device, moreover, it occurs in the static region. The behavioural VHDL coding style implements coarse-grain

parallelism of operators through process constructs. All ovals in the diagram represent independent VHDL process constructs and the circles represent the tasks assigned to them. Although a VHDL process is implemented through a disjointed state-machine, there are specific times in their execution where communication with one another is necessary. In the static region, the synchronisation between the transmitter and receiver processes quite literally performs the communication associated with transmitting data through a communications channel. To this end, the control side of the transmission is performed through a semaphore. The actual data transmitted through the semaphore handshaking are the BCH codewords.

In the reconfigurable regions a number of processes may be simultaneously active, decoding a codeword bit passed to them through the external communication channel. The static version of the decoding originally used another semaphore to perform the inter-process synchronisation, however, as the reader will recall, a similar function can be carried out by the channel controller when required to transfer data across the external or on-chip communication channels – hence its employment in the receiver.

An additional level of hierarchy is provided among the Viterbi decoders which takes the form of a single master process and several slave decoders. In this configuration, the master's role is to divide up each of the code's state weight and message tables and pass their values to the sub-ordinate decoders in the reconfigurable regions. After Viterbi decoding occurs in each of the regions, synchronisation occurs once more between the master process and each of the slaves, during which the master collects the weight and message bits found by each decoder and uses them to update its state tables. This continues until after processing of the last message bit is complete and the sample of the codeword is added to or a decision can be made on the robustness and efficiency of the current coding scheme.

The remaining task of the receiver is to extract the transmitted fragment of ASCII code from each of the codewords, after which the full ASCII characters are re-assembled into their full codes and the message sent over the communication channel is stored in preparation for its display.

The last element of coarse-level parallelism in the system is the way in which the message is displayed on the VGA unit. There is a strict protocol for video generation, in essence a frame is written once every 16.784 ms. A single frame is composed of 528 lines (480 visible – the

rest are blanked) lines, each of which are 640 pixels in length. Pixel data be retrieved and presented within a window of 31.77 μs, the period during which a line is written. An entire frame will occur every 16.784 ms; these are critical timing constraints, requiring a degree of manual intervention in MOODS. In order to fully explore the design space in MOODS i.e. with the minimal of timing constraints, this part of the demonstrator was not written using behavioural VHDL. Rather, RTL VHDL was used to read each re-assembled ASCII character and format it in a way that was presentable to the user viewing the received message.

In order to do this in real-time, the task was decoupled from the receiver in such a way that did not require the VGA generator to wait for the most recent decoded message. Instead, its purpose is to continuously read the ASCII data in a designated message ROM, format it into lines of characters on the screen and write the resultant messages to the VGA unit. This procedure is only interrupted by the receiver process, for the sole purpose of updating the message ROM with the most recent decoded and re-assembled ASCII characters.

Recall that synchronisation between process constructs does not occur at each wait statement - as is assumed in the RTL simulation model, but explicitly in the code through a dedicated pair of request and acknowledgement signals or semaphore. As depicted in the Figure 7.3, there are two occasions where a semaphore is utilised to exchange data between a pair of process. The same methodology is adopted in both cases. Figure 7.4 illustrates the timing of the pair of semaphore signals used between the transmitter and receiver processes shown in the previous figure. This method of process synchronisation is referred to as a toggle semaphore, as the exchange of data is dependant upon the inequality of the pair of request and acknowledge signals, the result of either process inverting or toggling one of the pair of signals.

Prior to the request to exchange the codeword, the request and the acknowledgment signals are equal. This situation arises when the transmitter is busy processing i.e. bit-packing the ASCII code and encoding the codeword. Meanwhile, the part of the receiver responsible for reading the codeword idles away the time, awaiting the next codeword whilst the remainder of the receiver circuits are busy displaying the current message on the VGA unit.

When the transmitter is ready to send the codeword, it inverts the request line. The inequality between the two signals forces it to enter an idle state, awaiting a response from the receiver.

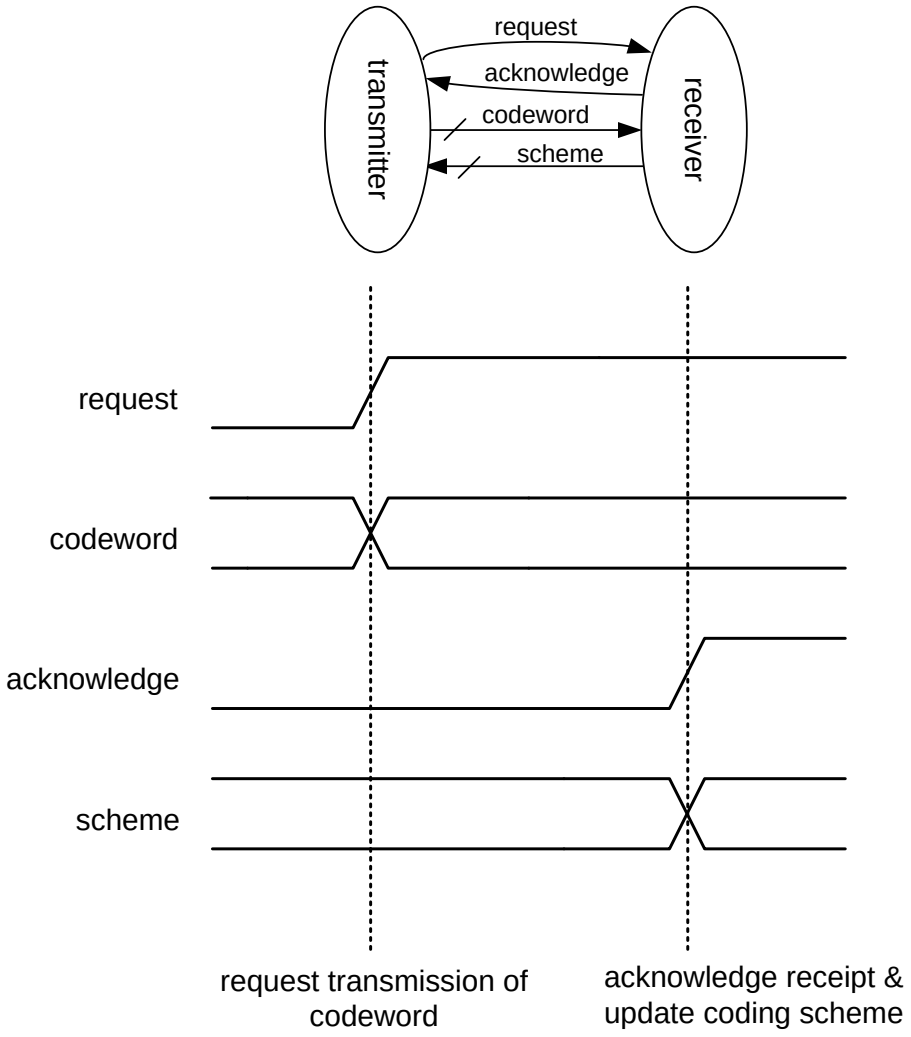


Figure 7.4: Semaphore communication between two concurrent processes.

At the same time, the signal inequality awakens the receiver, forcing it to write the codeword to the master process where it is disseminated bit by bit to each of the Viterbi decoders. Once decoded, the receiver has the opportunity to alter the coding scheme − it does this by writing the corresponding 2-bit binary code to the coding ‘scheme’ lines. This is followed by an inversion of the acknowledgment signal, the effect of which is to restore the equality between the two signals. Having done so, it returns once again to an idle state, where it awaits the next codeword request. Quite the opposite occurs to the transmitter, it is forced out of its idle state and must inspect the level of the coding scheme required. Where appropriate, it alters the parameters which govern the bit-packing of the ASCII characters and their encoding using the BCH codes. When encoding of the next codeword is complete, it signals a readiness to transmit it across the communication channel and the cycle is repeated ad infinitum.

7.4 Synthesis Results

Table 7.2 depicts the results for each of the synthesised Viterbi decoders and the remaining sub-system components. The target device is the Xilinx XC1000XV FPGA [6], providing 12,228 CLB Slices and 512 user Input/Output pins. The task was to synthesise the entire communication system in such a way that it would occupy less than 11000 slices. This is not an arbitrary figure, as it is approximately 90 % of the FPGA's logic resource capacity. We have found through experimentation, that any greater than 90 % of the resources is unlikely to be floor-planned in a way that is acceptable for generating partial bitstreams.

Circuit Sub-system	Circuit Description	Area (slices)	Critical path delay (clock cycles: µs)	Reconfiguration (ms)
decoder	15, 11, 3; Reg.; processors: P_1-P_4.	3092	1218 : 24.36	4.98
decoder	15, 7, 5; Reg.; processors: P_5, P_6.	3542	18221: 364.42	4.98
decoder	15, 5, 7; RAM; processor P_0.	2900	116214: 2324.28	-
decoder	15, 7, 5; RAM; processor P_0.	2900	31158: 623.16	-
decoder	15, 3, 11; RAM; processor P_0.	2900	3900: 78.00	-
encoder	Universal BCH encoder	798	31: 0.62	-
formatter	ASCII bit-packer for codes	652	56: 1.12	-
corrupter	Channel noise generator	249	20: 0.40	-
all	Complete RTR System	8141	-	-
all	Complete static System	11233	-	-

Table 7.2: Synthesised RTR variable coding system.

The area, delay and reconfiguration constraints are set to high priority in MOODS. All target values are set to zero. The delay is expressed in terms of the number of cycles required for the sub-system to perform its function.

Both decoder configurations are assigned to a single reconfigurable region whose size is fixed by the larger of the two register based reconfigurable decoders encompassing processors P_5, P_6 BCH (15,7,5). The static decoder P_0 shares the region using slices as LUT RAM in a

parameterised version of all three coding schemes. The size of the complete RTR system is the sum of the reconfigurable and static regions:

$$\text{circuit area (RTR system)} = 3542 + 2900 + 798 + 652 + 249 = 8141 \text{ slices.}$$

The number of resources required by the decoders is not necessarily dominated by the memory requirements of the decoder, rather how the memory is implemented. For instance, the BCH (15,11,3) decoder stores the weights and codewords for 16 states, yet it requires more resources than the BCH (15,5,7) decoder which performs the decoding for 1024 states. The explanation lies in the limited number of registers (2 per CLB Slice) and the abundance of Look-Up tables that can be used to implement RAM blocks. The reader will appreciate the rationale behind synthesising the message and weight tables using different memory resources; components would otherwise be un-programmed in an FPGA implementation using a single type of memory resource.

In addition to reducing the number of uncommitted FPGA resources, re-using existing components through run-time reconfiguration resulted in more than a 30% reduction in their number – as shown in the last two rows of Table 7.2. The area reduction was achieved through the continual partial reconfiguration of the resources between the two parallel decoders whose properties are characterised in the first two rows of the table.

The difference between the resource consumption of the two decoders BCH (15,11,3) and BCH (15,7,5) respectively, would justify further experimentation in optimising the use of their data-path units. One option might be to utilise the uncommitted resources in the smaller decoder BCH (15,11,3) through a tightening of its delay constraint, thereby increasing the parallelism of the data-path units and the resource consumption of the four decoders. The area requirements for the remaining sub-systems are modest in comparison with the decoders, due to the absence of memory storage.

A final comment on the properties of synthesised circuits is with regard to the difference between the resources used by the static and reconfigurable systems: the static version utilises more than 90% of the resources of the target device (XCV1000). At approximately 66% of the resources, the reconfigurable system can fit on a smaller device, such as the Xilinx XCV800 FPGA, where it would be placed in almost 86% of the available logic resources.

7.5 Run-time Characteristics

The reader will recall from Section 7.3.1 that the number of errors occurring in a sample of 120 bits (8 codewords) taken by the receiver is used to determine whether a change in the coding scheme is required. Table 7.1 shows the number of errors permitted for each BCH coding scheme in relation to a desired Bit-Error Rate (BER) of approximately 5%, the thresholds are reproduced below:

(15, 11, 3): sample errors < 4; (15, 7, 5): sample errors < 8; (15, 5, 7) sample errors < 13.

Figure 7.5 depicts the results generated from automated temporal partitioning: it illustrates how the threshold values are used by the receiver at run-time to relate the number of errors found in a sample of codewords to each configuration of sequential and parallel Viterbi decoders.

A single reconfigurable region of the FPGA is reprogrammed with two temporal partitions, both of which incorporate the sequential Viterbi processor P_0 . The logic and routing resources of the decoder are identical in both partitions, ensuring its continued execution during the context switch from one temporal partition to the next.

Unlike the sequential decoder, the two parallel Viterbi decoders (P_1-P_4), (P_5,P_6) are unconstrained in both partitions. Intentionally state-less during reconfiguration, their logic and routing resources are reconfigured in parallel to the execution of the sequential decoder without overlapping their resources. As well as providing MOODS with greater freedom to share data-path components during optimisation, an absence of placement constraints provides a greater choice when placing and routing components using the device vendor tools; often an issue when floorplanning designs for partial reconfiguration.

As indicated by their shading, only one Viterbi decoder is ever active during the execution of a temporal partition. The partitioner selected three combinations of active decoders for each temporal partition: C_1-C_3 and C_4-C_6 respectively; we excluded combinations of decoders that would have otherwise processed the same coding scheme using both the sequential and parallel decoders. Exactly when a decoder is active is determined by the value of the BER and

the threshold of each edge which represents a transition in the coding scheme between the temporal partitions.

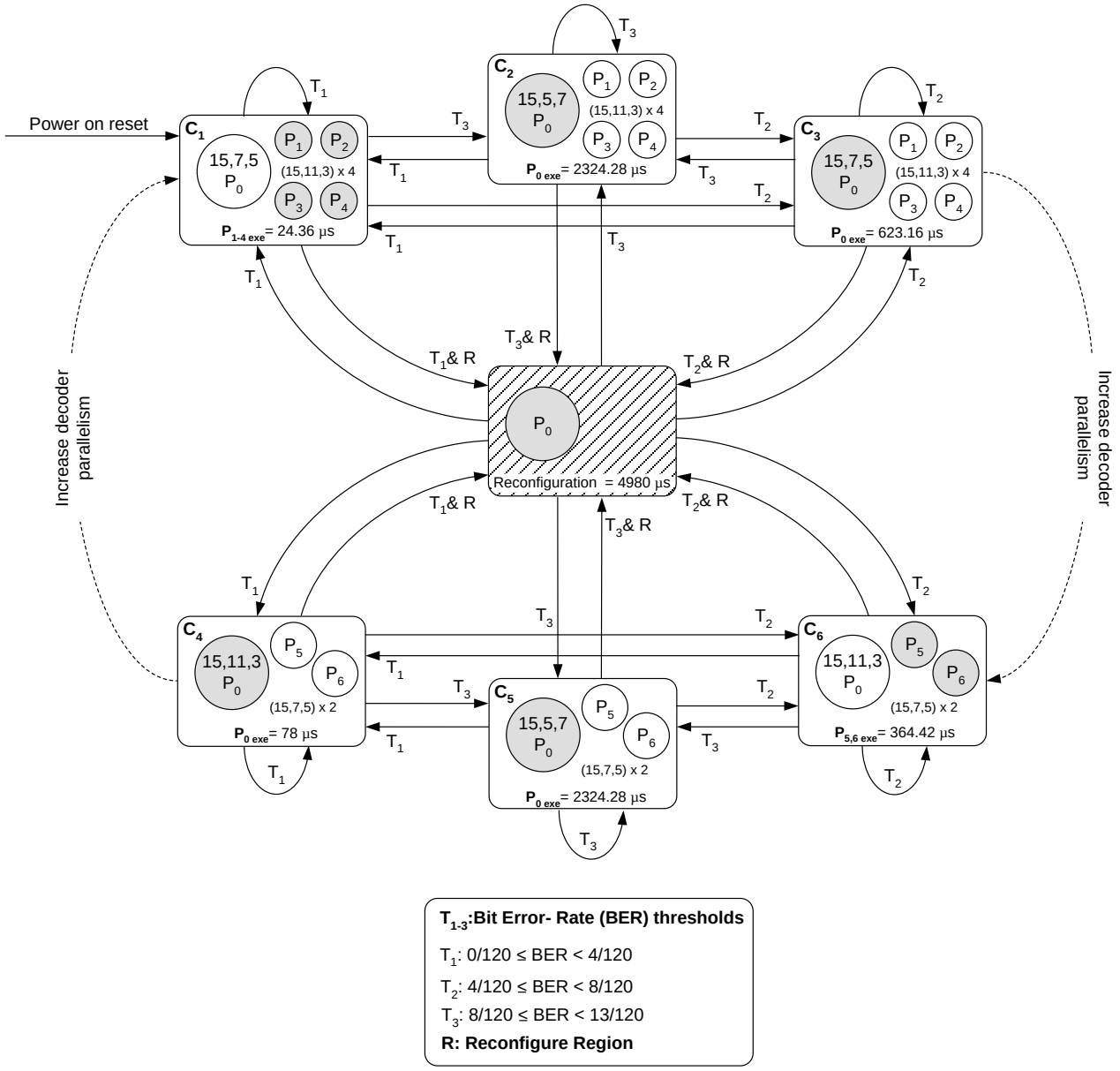


Figure 7.5: Channel bit-error rate relationships between the decoder configurations.

Upon the application of power, the default code scheme of BCH (15,11,3) is executed on the reconfigurable region through configuration C_1. The BCH (15,11,3) code was selected as the default because it is the most efficient of the three codes (in terms of data to parity efficiency (73.3)%); the reader is referred to Figure C.1 of Appendix C for further details. The default

configuration shown in Figure 7.5 is labelled with the time taken to decode the codeword $P_{1-4_{\text{exe}}} = 24.36\ \mu s$, reflecting the parallelism of the four register-based Viterbi processors P_1-P_4 for channel conditions where the BER is smaller than 4 in 120 bits (T_1).

The sequential processor P_0 is inactive until a decision is taken by the receiver to change the coding to a more robust scheme. The protocol used was described earlier in Section 7.3.1 and depicted in Figure 7.4; transmission using a semaphore takes 2 cycles or 40 ns for the current 50MHz clock signal.

A decision to select a different BCH code will ensure that data transmission continues during any increase in channel noise. However, it comes at a cost of a reduction in efficiency because a switch to a more robust code will result in a reduction in data efficiency, as well as an increase in the time taken to decode a larger code space using a sequential decoder implementation. For the BCH (15,7,5) code, the encoding efficiency is reduced to 46.7% whilst the decoder takes 623.16 μs to decode a codeword. In the case of the BCH (15,5,7) code these characteristics are 33.3% and 2324.28 μs respectively. Of course, the noise level of the channel may reduce as well as increase and the option remains to switch to the default decoder when necessary.

Assuming that the BER threshold T_1 is exceeded, in order to maintain the quality of service defined by the interval T_2, the parameters of the sequential decoder P_0 are changed to set the decoder to match the nearest BCH (15,7,5) code. The next choice available to the receiver gets to the crux of this RTR approach to decoding: without any change to the current code scheme or interruption to codeword decoding, the idle resources previously allocated to four parallel Viterbi processors (C_3) may now be reconfigured to realise two parallel versions of the current scheme BCH (15,7,5) in a different temporal partition (C_6).

To mitigate the high cost of reconfiguration using an FPGA (4.98 ms) the sequential decoder continues to decode codewords without loss of state throughout the reconfiguration period. As a result of the context switch, Viterbi decoding is accelerated by approximately 1.7 times after taking into account the communication costs between the parallel decoders. This scenario assumed that no change in the coding scheme was necessary (BER$<$ 8/120 bits). Had this not been the case (T_3) and a BCH code with a greater error correcting ability been required e.g.

BCH (15,5,7), a 40 ns delay in communicating the change in the coding scheme to the transmitter would be accompanied by the relevant change in the sequential decoder's parameters.

In the event of the receiver wishing to relax the scheme to a less robust BCH code, it may rely upon changing the parameters of the sequential decoder. During prolonged periods of low channel noise, it could repeat a similar reconfiguration process to the one described earlier, with the aim of switching (C_4 to C_1) to the least robust code scheme BCH (15,11,3) but most parallel version of the Viterbi decoder. Alternatively, the receiver may decide to commit the reconfigurable resources to occupying the 'middle ground' of the coding scheme, a place where any change in the channel condition is but a short code distance away from its response.

7.6 Summary

This chapter presented a practical application for the synthesised run-time reconfigurable architecture described in the earlier chapter: a run-time coding system which changes the coding scheme and degree of parallel decoding based upon the history of errors it encountered during the transmission of the codewords. Unlike the test circuits used to generate the results in chapter 6, this particular application relied upon the explicit use of reconfiguration as part of the design specification, where partitioning occurred at the subroutine level of abstraction. High-level synthesis was applied at the operation-level alongside temporal partitioning, where it took advantage of the discrepancy between the sizes of the partitions to vary the degree of instruction-level parallelism for each synthesised decoder.

Partitioning across the functional boundaries of the Viterbi decoders associated partial reconfiguration with a change in the characteristics of each decoder. A prominent characteristic is the number of states that require updating, which changes dramatically with each type of BCH code selected. A non-reconfigurable MOODS implementation [123] used a change in parameters of a sequential Viterbi decoder to implement a variable coding scheme. The author successfully showed the advantages of adapting the efficiency of the coding scheme to match the level of noise in a simulated communication channel. The codeword and weight arrays were implemented using a single port RAM that was used for all three coding

schemes. The disadvantage of this approach was the idle use of logic resources, particularly the use of device registers which remained uncommitted in the target FPGA. Other authors [124,125] used MOODS to synthesise highly optimal parallel versions of the Viterbi decoders which traded resource consumption against speed of decoding. As with the sequential decoder, the authors selected one type of memory resource (device registers) to the exclusion of others during FPGA implementation.

Aside from exercising the architecture generated automatically by MOODS during temporal partitioning, a strong motivation for the case study was to determine whether the un-programmed resources could be better put to use in another Viterbi decoder: LUTS which were previously idle in one decoder might become parallel logic operations in another, registers used for array variables could also be re-wired for use in the data and control paths. Such a re-use of resources was not limited to instruction operations alone; at the functional level, several parallel decoders were able to be reconfigured from the resources of a less parallel but more robust decoder architecture.

The size of the complete system uses approximately 30 % fewer programmable logic resources than the non-reconfigurable counterpart. The reduction in size does come at a price of a 5 ms reconfiguration delay. However, the temporal partitioner was able to generate partitions comprising two types of Viterbi decoders: a sequential decoder was optimised using the idle resources of the FPGA, where it continued to decode codewords without interruption during reconfiguration. A parallel decoder made use of all the remaining programmable logic resources to accelerate the decoding of codewords. This approach was used to adapt both the efficiency of the message encoding and codeword decoding to the noise level encountered during transmission within a rudimentary communication system.

Through its implementation at the device level, the MOODS HLS tool has been successfully used in conjunction with a temporal partitioning approach, to generate circuit structures which explore the new territory formed by the incorporation of run-time reconfigurable resources. We believe that this case study has shown that partitioning and synthesising reconfigurable logic at the algorithmic level of abstraction provides a better rationale for RTR, when contrasted with its use as a general area/delay trade-off during circuit optimisation.

Chapter 8

Conclusion and Further Work

8.1 Conclusion

The theme of this thesis has been how to partition and preserve circuit behaviour at different levels of abstraction during Behavioural Synthesis, ultimately leading to an implementation at the device-level using Run-time Reconfiguration (RTR).

In the context of the MOODS behavioural synthesis system this approach required a Temporal and Spatial Partitioning of a VHDL behavioural circuit specification. As a consequence of the partitioning, MOODS is now able to perform simultaneous optimisation at the instruction and subprogram levels of circuit abstraction, in contrast to other temporal partitioning approaches [58,59] that exclusively target one level of representation.

In response to the use of the ‘Temporal_Partition’ compiler directive, preparation by the MOODS data-structures ensures that a subroutine’s behaviour is preserved at the graph-level of abstraction: in this form it can be partitioned through a graph modifying transform that is applied using a Simulated Annealing heuristic. The ability of the simulated annealing algorithm to explore both improving and degrading optimisations has been an essential aspect when using commercial Field Programmable Gate Arrays (FPGAs) as the target reconfigurable resource: FPGA devices are not designed for efficient reconfiguration; the time taken to reconfigure their resources is several orders of magnitude greater than the time taken to execute them.

The practical implementation on a FPGA was motivated by the iterative approach taken by MOODS to optimisation: each control and data-path component is always bound to a specific technology library, providing the cost function with a direct measure of the properties of the circuit structure. To operate within this approach, temporal partitioning was also directly quantified by implementing it in the form of temporal resource binding; through its

application, partitioning proceeds by sharing a resource between subroutines at the same time (spatial partition) or at different times (temporal partition) via a reconfiguration of their common resource. In either case, the use of resource binding provides the interface necessary to prevent subroutine tokens from being lost or corrupted during reconfiguration.

As described in Chapter 2, the majority of temporal partitioning approaches reported in the literature [66] rely upon heuristics which seek only to improve a partition; contrary to one's instinct, research has shown this can lead to poorer results [126]: occasionally accepting an optimisation that is know to worsen the metrics can lead the exploration of the 'design space' in a direction that might otherwise have never been taken. This has particular resonance in the approach taken to generate the results presented in Chapter 6, where the relationship between a cost function and each distinct effect of the temporal partitioning transform would not have been verified without the ability to allow 'hill-climbing' in the design-space.

In reference to the summary provided in Chapter 6, the results produced through experimentation do support an understanding of how an individual partitioning 'move' improved or degraded each of the cost function metrics. The significance of the results is that it could form the basis for a more directed approach to applying the transform during optimisation; however, as the results show, this approach should still incorporate the ability to 'undo' the effects of a transform in order to fully explore the design space.

Chapter 5 detailed the infrastructure provided by the author to implement temporal resource binding at a layer of abstraction suitable for device-level partial reconfiguration. All components are automatically generated by MOODS during optimisation, in a way that is architecturally transparent to the user, thus enabling an exploration of the greater design space formed through the temporal partitioning of subroutine modules.

As described in Chapters 4 and 5, this approach automatically implies a physical placement inherent to the binding that is often less tangible in the other approaches to temporal partitioning [65, 66]. The physical link provided by the resource binding supports decisions taken by the resource binding transform: the placement inherent in a resource binding greatly aids the task of modelling the physical aspects of partitioning: the length of channels can be quickly characterised or logic resources increased by determining if two temporal partitions are adjacent in space and time to allow their merger.

Despite spanning several generations of vendor tool flows and devices, the infrastructure supporting RTR incorporates a number of aspects which are fundamental but are currently not supported by the current vendor tools [122]: there continues to be disparity between the capabilities of the device and the features of the tools.

Most critical is the lack of provision for creating nested partitions: this is vital for modules in MOODS to preserve the control and data-path signals when partitioning across module hierarchies that share the same reconfigurable resource. In the broader use of RTR, state saving is a necessary prerequisite for implementing more complex systems. An ‘ad-hoc’ approach was implemented by the author who through resource binding was able to preserve control and data states. An alternative approach which enables the state of registers to be read and restored requires the user to consult several unrelated documents, some of which are not documented within the vendors guide to partial reconfiguration [122]: from the perspective of a user of RTR, information surrounding its implementation can often seem esoteric.

Despite research spanning more than three decades, there has yet to be consensus on what exemplifies the ideal application for a RTR methodology. That is not to say that there is an absence of applications for RTR, far from it [9]. There are numerous examples of RTR being successfully employed in hardware acceleration, particularly for stream-based computation [127]. Repeated operations made to continuous data-streams (inherent to video and image processing systems) are particularly well suited to reducing the ratio of reconfiguration to execution time.

Chapter 7 provided a practical evaluation of the synthesised architecture. Unlike the test circuits used to generate the results presented in Chapter 6, the partitioning was not determined by the cost function but as part of the design specification. A 60% reduction in the resources required for a static version was achieved at penalty of a 5 ms reconfiguration delay. As described in the chapter summary, a partitioning is possible which incurs no reconfiguration delay to the transmitter.

This draws attention to the fact that as an automatic optimisation trade-off, run-time reconfiguration is suited to applications where it can exploit user knowledge in its implementation. Such an approach to partitioning supports the perspective taken in this thesis that run-time reconfiguration should also enable the implementation of different structural

versions of the same behavioural description. This is supported by the partitioning of program subroutines as opposed to individual instruction operations. Through the resource binding transform and the device infrastructure, a first step has been taken towards extending the software analogy to the device-level of abstraction, beyond using subroutine libraries to simplify coding a specification.

With regard to the reconfiguration characteristics of FPGAs, very little has changed in over a decade and there is little reason why that should not continue − if change does come, it is just as likely to take the form of more spatial than temporal layers [128]. Should a new technology or device come to market, much of the change needed to MOODS would be predominantly through a change to the technology library!

It is encouraging to see a major vendor of FPGAs also providing a high-level synthesis tool 'Vivado' HLS [129] capable of aiding the user in implementing RTR, this can only increase the user-base. Will there be an upturn in the use of RTR? Perhaps the reader is best placed to answer that question: it is only through a willingness to assess the suitability of RTR for oneself and the subsequent demand that it would create, that future research into technology, tools and techniques can continue to be justified. Furthermore, the answer to many a question regarding the possibilities and limitations of RTR is to be found within the extended design-space formed when choosing to synthesise circuit hardware using run-time reconfigurable resources – it can only be realistically explored through the use of automated HLS tools, such as MOODS.

8.2 Further Work

As referred to in the Section 8.1, it would be a natural next step to attempt to encapsulate the trade-offs between each of the partitioning metrics in the form of a dedicated heuristic for temporal partitioning. The heuristic would rely upon much of the relationships identified through experimentation using simulated annealing, perhaps applying it in the form of a pre-optimisation step: a similar approach is available in MOODS to measure the share-ability of data-path units, preventing independent operations on the critical path from being shared rather than scheduled to the same control step to reduce the length of the path. Similarly, a

metric characterising the adjacency of module execution calls would reduce the number of rejected moves due to high levels of module context switching.

Li et al. [130] describes a scheme which uses an adjacency matrix to spatially partition those modules which frequently appear next to one another in a stream of module calls at run-time. As MOODS uses static scheduling, more information about the module sequences is known in advance at compile-time (as the reader will recall from the module characteristics in Appendix B). In addition, unlike scheduling at the instruction-level of abstraction, subroutines do not change their order during scheduling: combining these characteristics, the author has initially experimented with a metric for measuring subroutine execution ‘similarity’. Unlike the adjacency matrix it does not require updating during run-time; it can be applied prior to performing temporal partitioning in order to reduce the number of rejected moves due to combinations of modules resulting in high context switching delays. It is expected to improve upon the current approach by filtering out certain combinations of modules without actually applying and rejecting the resource binding transform.

A final point of interest is that the approach of preserving behaviour through temporal partitioning extends the view of taking a software approach to hardware design. Without RTR, this approach is about simplifying the coding of a specification by re-using subroutine libraries. With RTR, preserving the behaviour and how it is implemented can provide the basis for design re-use at the structural or device-levels of abstraction; an additional way of managing current and future design complexity.

Appendix A: MOODS

This appendix describes how MOODS is able to represent and subsequently optimise a circuit specification at the instruction-level of abstraction.

A1 Synthesised Architecture

The architectural model in MOODS to which all circuits are synthesised is composed of a number of synchronous Finite State Machines (FSMDs) which control the data-flow through the data-paths. The FSMD model is represented within the MOODS data structures as a number of control and data-path graphs, built directly from the behavioural, sequencing and connectivity information contained within the ICODE description.

Figure A.1 illustrates a fragment of the control and data-path graphs that would automatically be generated from an ICODE description of the ‘bch encoder’ described in Chapter 3 Figure 3.4; a second graph is also shown to exemplify how concurrency is modelled within the architecture.

The topology of the control graphs is derived directly from the sequence of instruction activations contained within the ICODE ‘Program’ and ‘bchEncoder’ modules. Each control step ‘si’ represents what will ultimately be a single state in the MOODS output description of the corresponding state machine controller(s).

State transition of the controller is modelled in the graph by a directional arc between each pair of control nodes. In the structural description of the controller, the firing of an arc is represented by a token passing from one state to the next. In doing so, it resembles the token passing mechanism for the execution of the instructions described at the ICODE level i.e. an instruction is activated by another and only upon completion of its execution. The graph data structure relates the firing of an arc with the passing of a token, in such a way that the activation of the control node at the end of the arc is synonymous with the execution of the associated ICODE instruction.

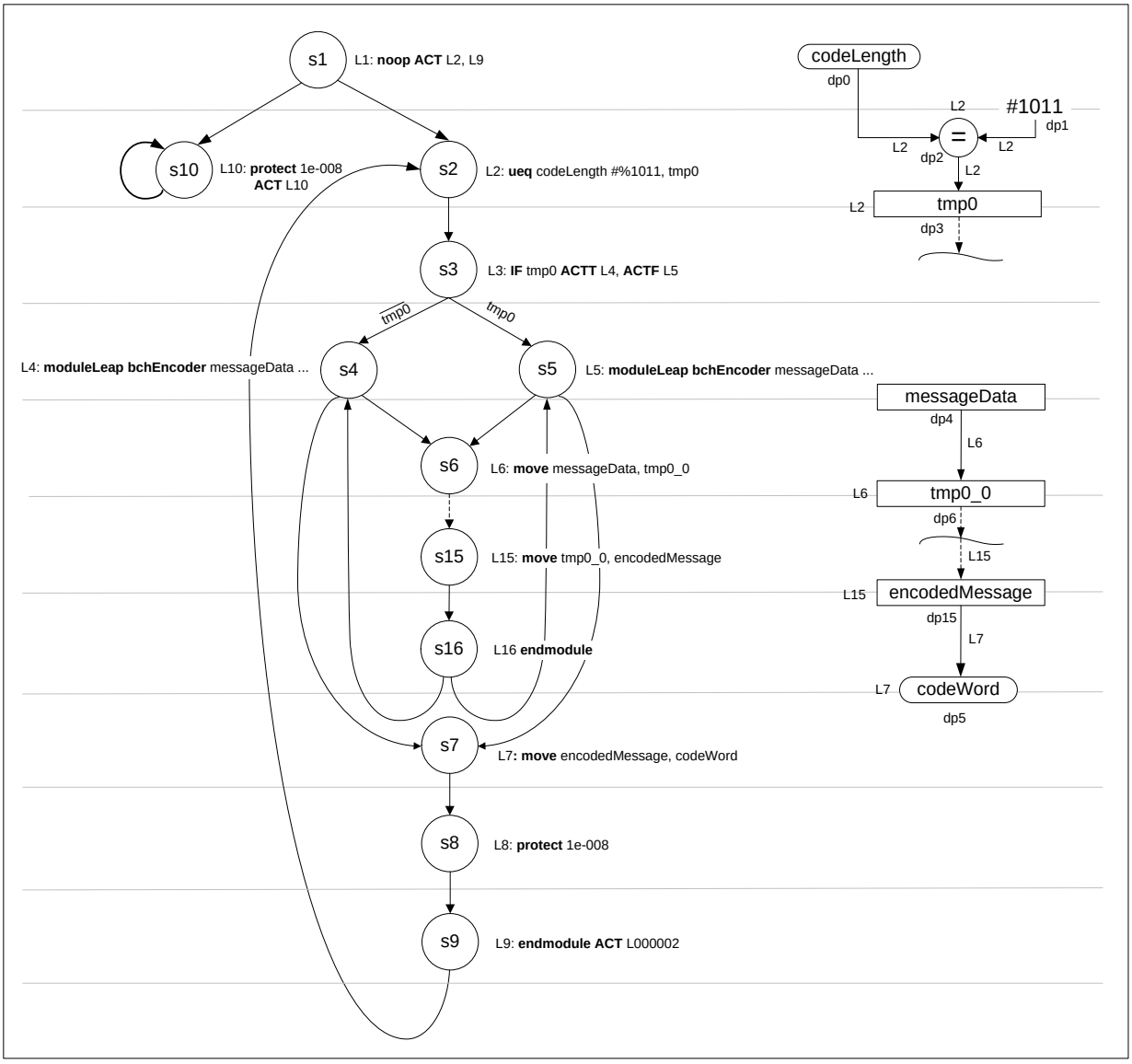


Figure A.1: Control and data-path graph sections for the BCH encoder algorithm.

The initial assignment of a single instruction to each of the control nodes is depicted alongside the control graph. It requires no strategy when allocating each instruction to a control node and in doing so, places no resource constraints upon the initial scheduling. Optimisation proceeds in the next phase of synthesis, where the merger of a pair of control states and the subsequent removal of one of them acts to assign more than one instruction per state.

The direction of the control flow through the graph, be it feed-forward or backward (loop) is dictated by the nature of the instruction activation list. Unconditional activation of more than one instruction in the list, as is the case for the ‘NOOP’ instruction ‘L1’, requires the

simultaneous firing of multiple control arcs. This in turn, determines the type of control node used, such as the fork node ‘s1’. There are six types of control node, three of which are utilised in the graphs depicted. The factors which distinguish them from one another are the number of their input and output arcs and the circumstance in which they are active.

The presence of a ‘fork’ node in the graph denotes the existence of multiple concurrent threads of instruction execution. As depicted, it has one input and multiple output arcs, both of which are fired unconditionally to successor nodes which mark the start of two parallel control graphs. Each graph defines the order of precedence among the instructions, originally derived from the source code description of VHDL processes. As specified in the VHDL source description, each graph is a disjoint free running state machine with no requirement to re-converge. Processes may synchronise to pass data to one another using semaphores, as described in the Chapter 5, but when compiling VHDL there is no use for joining concurrent branches.

Although not supported by the VHDL compiler, for the reasons described above, the ‘collect’ node (not depicted) fires a single output arc only when all of its multiple input arcs have been activated. In this way it is able to synchronise any number of concurrent input branches. It remains supported by the MOODS core data structures and is also present as an ICODE instruction, providing a means of modelling fine grain parallelism for source languages capable of describing it, such as SystemC [96].

A ‘general’ control node, such as ‘s2’ is ubiquitous throughout the graph because with exception to ‘moduleLeap’, ‘if’, ‘switchon’ and ‘collect’ instructions, it can be used to schedule any ICODE operation. It has a single unconditional input and output and is often the result of optimisation to graphs segments containing other control types.

The next class of control node depicted is the ‘conditional node’ (e.g. ‘s3’), created in response to the presence of the conditional instruction ‘if’ (‘L3’). The firing of the single input arc and a condition being met in the data-path with reference to node dp4 and whether variable ‘tmp0’ is logic low or high, governs which of the mutually exclusive output arcs is fired.

Regardless of which of the conditional control paths is taken in the figure, the ‘call’ nodes ‘s5’,’s6’ are encountered. When one of their mutually exclusive ‘moduleLeap’ instructions ‘L5’,’L6’ is executed, a token is passed to the first instruction of the sub-module. As shown in the figure, this is represented in the main graph by an output arc which connects the call node to the first node ‘s6’ in the module’s sub-graph. Execution of the sub-module proceeds, during which time, the calling module (program) waits because only one ICODE instruction can ever be active in any given thread of instructions. Upon completing execution, the process is mirrored: the token is returned to the ‘moduleLeap’ instruction (represented by an arc fired by the last control node of the sub-graph ‘s16’ in response to the execution of the ‘endmodule’ instruction ‘L16’) and it subsequently activates the next instruction ‘L7’ in the program module, by doing so, continuing its execution.

A control graph embodies the circuit behaviour by modelling when and in what order the instructions are expected to execute. What it does not do is describe how the ICODE operations are implemented in hardware. That is realised in the structural description provided by the Data-Path graph. As with the control graph, the initial data-path is formed by allocating one ICODE operation or variable to a single data-path node. This direct relationship between operation and operator is intentionally broken during optimisation, when some of the data-path nodes may be shared, reducing the number of data-path units required to implement the ICODE instructions.

Of course the circuit is only truly modelled when its behaviour (ICODE instructions) and structure (data-path) are linked. In the data-path, signal transfer as well as a dependency from one unit to the next is represented by a data-path net. Among the attributes defined for it, such as bit width, the data flow in the net is described in terms of the activation of an instruction whose operands i.e. ICODE constants and variables provide the source and sink of the data being transported along it.

For example, consider the execution of the instance of the equality operator instruction ‘L2: ueq codeLength, #%1011, tmp0’. The data-nets which connect the operands of the instruction to their data-path units (dp0-dp3) are labelled with the instruction number. In this way, data flow is always explicitly linked to instruction execution. The rationale for this stems from the fact that the behaviour of the circuit, as embodied by the ICODE instructions, must

remain unaltered throughout optimisation. Whilst their scheduling and allocation to control and data-path nodes will undoubtedly change during optimisation, much of the information characterising the data flow can remain unaltered, being expressed in terms of active instructions rather than directly linked to controller states which may be optimised away. This aids in reducing the amount of re-processing to the data structures.

The MOODS data structures explicitly model the scheduling of ICODE instructions to each control node, however, the representation of their execution is implicit, described indirectly through the control of an instruction's output variables. More specifically, an instruction is considered to have been executed when the registers implementing the variable are loaded at the end of the active control state.

Each control node in the graph is bound to an individual control cell, implemented using a single register in the state machine. In this way, the controller is synthesised as a one-hot state machine. The direct mapping of state to variable and subsequently variable to register, need not be fixed to realise a one-hot encoding. The data structures which describe the activation of the control arcs and ultimately the next state logic may be altered to realise other forms of state assignment, such as binary or greyscale. One-hot encoding is presently favoured when targeting FPGAs because there is generally a register adjacent to a look-up-table in most architectures i.e. Xilinx Virtex Slice [6]. Of course, the same cannot be said when targeting an ASIC, where such a state assignment would be deemed fast but expensive in terms of registers resources.

Binding the control and data-path units to a particular technology enables the user to specify a target or constraint for a particular metric, such as circuit area. Without such physical characterisation, the circuit modelled by the control and data-path graphs remains abstract and unquantifiable to all but the most simple of measures, such as counting the number of control and data-path nodes when finding the area and critical path delay of the circuit. Once the user has placed the architecture in the context of a particular technology, presently through the Xilinx Virtex FPGA library [50], MOODS is able to automate the numerous scheduling and allocation decisions required to transform the circuit structure into a form that meets the criteria required by the user.

Each control and data-path node is bound to a cell taken from an FPGA technology library. Although the intricacies of the libraries, as with much of the MOODS data structures, are beyond the scope of this thesis, it will suffice to mention that it does more than list characterisation data. It is a repository for pre-determined technology constants, such as the size of a single register when implemented using a Virtex FPGA [6]. However, it also uses those constants to generate an estimation of the size of a particular data-path unit when parameterised with multiple inputs and outputs i.e. the area required when implementing a vector variable using a register.

The module library is autonomous to MOODS, in that the characterisation data is requested by MOODS core routines. This decoupling allows new Technology Libraries to be created without requiring any changes to the MOODS control and data-path modelling and optimisation routines. The ICODE database adds a level of indirection to the separation of MOODS and the technology library. Without it, the library has no concept of the behaviour (in the form of ICODE) a cell is required to characterise. An ICODE instruction encapsulates the type, number and width of input and output variables. This information is utilised during optimisation, where two instructions may be allocated to the same data-path unit. It becomes indispensable during optimisation when assigning two different arithmetic instructions i.e. addition and subtraction to a general purpose arithmetic and logic unit.

A2 Graph Transformations

Circuit optimisation is achieved in MOODS through the application of scheduling, allocation and binding transformations. The scheduling transforms act to minimise the number of control-graph steps, whilst their data-path counterparts reduce the number of data-path units used to implement the behaviour embodied by the ICODE.

A2.1 Scheduling Transformations

The primary effect of any one of the seven scheduling transformations is to reduce the number of control graph nodes and ultimately, the number of states utilised in the corresponding state machine controller. As its name suggests, the scheduling transform is able to alter the execution schedule of the ICODE instructions in a way that aims to meet the user supplied

timing constraints i.e. the critical path and longest state delay or clock period. In practice, the effect of the seven transforms is to combine or split control states, re-assigning the ICODE instructions in the course of doing so. The effect of the transform on the critical path delay and clock period is either an improvement or degradation, depending upon the type of scheduling transform applied. A reduction in the total number of control states is obviously an immediate improvement to the length of the critical path. However, should the control states selected by the transform determine the clock period, the result of assigning additional instructions to an existing state would be detrimental to the clock period.

Table A.1 lists the effect of each of the seven types of scheduling transforms.

Scheduling transformation	Description of behaviour
Sequential merge	Given two control states, merge the instructions of the second state with those of the first and remove the now redundant second state. Independent instructions are ‘chained’ in the same group. Non-dependant instructions are assigned to concurrent groups. Any temporary data-path registers used to transport the ICODE variables between the instructions are now bypassed.
Merge fork & successor	Aims to merge the instructions of a ‘fork’ or ‘conditional’ node with those associated with the successor node of one of its branches. The conditional execution of the arc leading to the instructions of the successor node, is maintained by conditions imposed upon the execution of the instructions merged with those of the branching node.
Parallel merge	Merge the instructions from two ‘concurrent’ successor nodes of a ‘fork’ into a single control node. The execution of the affected instructions remains concurrent. The superfluous fork node is removed from the graph.
Group instructions on variable	Chain the instructions which write and read to a variable in different control states. Frequently used to remove the registers implementing compiler-generated temporary variables.

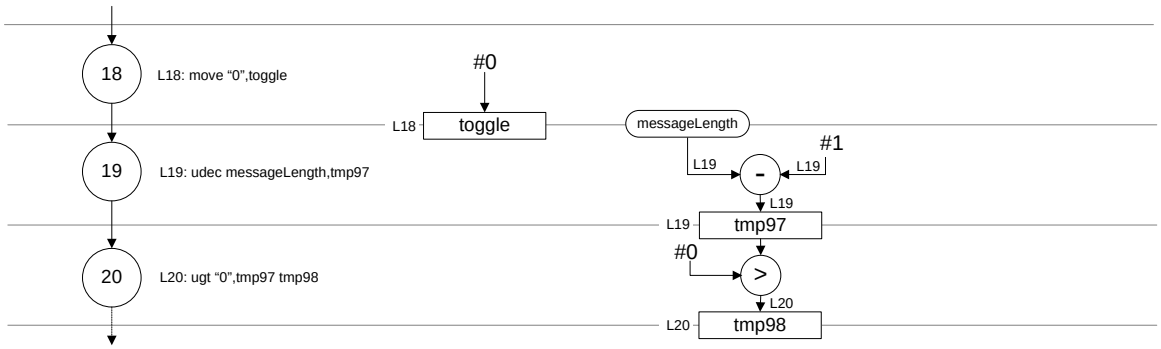
Ungroup into groups	Splits a control state into two by re-assigning a concurrent group of instructions to a new state.
Ungroup into time slices	Splits a single control state and many groups of instructions into a number of new ones, such that each new state executes one instruction from each concurrent group within a given time.
Set clock	Governs the clock period of the design by utilising the 'Ungroup into time' slices transform, to ensure that the execution delay of the longest instruction in each state does not violate the user target for the clock period.

Table A.1: Scheduling transformations available for optimisation of the control graphs.

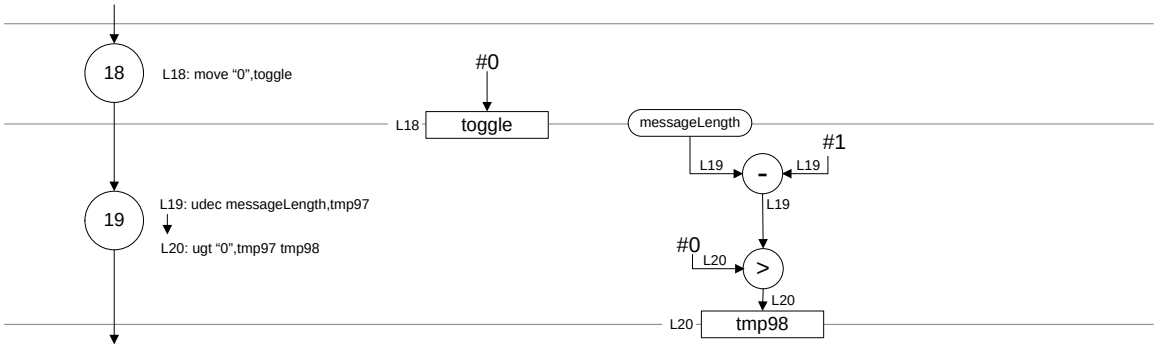
It is not appropriate to exemplify the application of each of the transforms, for more information the interested reader is referred to [4,5]. Nonetheless, to enable the reader to appreciate the use of the scheduling transformations in the context of the control and data-path graphs, the Sequential merge transform is applied to the segment of the graphs shown below in Figure A.2 (a).

Figure A.2 (b) depicts the consequences to the graphs should the transform be applied to control states 'c19' and 'c20'. The main effect is to perform an earlier scheduling of instruction 'L20' which due to the data-dependency present in the form of the ICODE variable 'tmp97', necessitates its chaining with the existing instruction 'i19'. Of course the secondary effect is the reduction in the resources allocated to realise the state machine controller. The execution of the instructions within the group, formed by their merger, is entirely asynchronous i.e. 'L20' is activated solely by 'L19'. This removes the need for register 'tmp97' which was previously used to pass the variable across the clock boundary dividing the two control states.

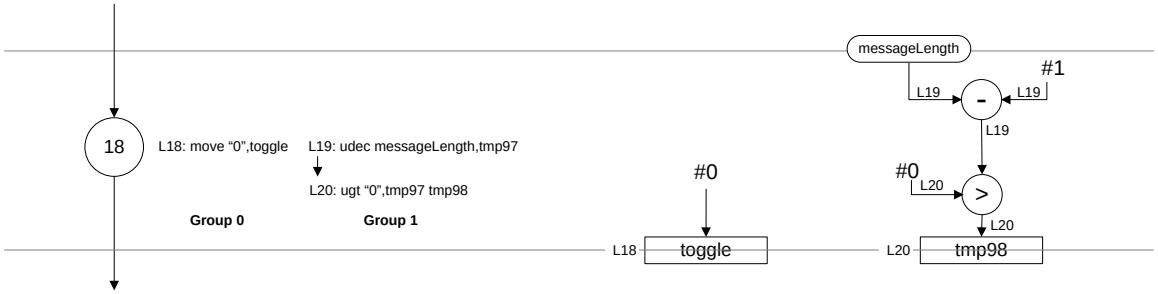
Applying the transform once again to state 'c19' and its predecessor 'c18', results in the formation of the two parallel instruction groups as depicted in Figure A.2 (c). Their execution is concurrent because of the absence of any data-dependency between instructions 'L18' and 'L19' or 'L18' and 'L20'.



(a) Initial control and data-path graphs



(b) Graphs following the application of the Sequential merge transform



(c) Repeated application of the transformation

Figure A.2: Application of the Sequential merge transform.

2.2 Allocation and Binding Transformations

The remaining transformations are employed for optimisation to the data-path graphs. Two transforms enable sharing of data-path units, be they functional (arithmetic) or storage (memory or registers). A single binding transform is able to select an alternative library cell for any given data-path unit, although in practice this is currently limited to a small number of units. The rest of the transforms provide a means of undoing the allocation, thus enabling the optimisation algorithm to perform a number of degrading or a hill climbing moves which is hoped will enable a broad investigation of the many possible alternative circuit structures during optimisation. Table A.2 describes the effect of each of the data-path allocation and binding transformations.

Data-path transformation	Description of behaviour
Share functional units	Provides a means of reducing the circuit area through the allocation of multiple instructions to a single data-path unit. This requires the addition of a multiplexor to the unit's inputs which allows two or more instructions (in different control states) to share it. The instructions needn't be of the same type e.g. multi-function ALU units allow for different arithmetic instructions to be assigned to a single unit.
Share registers	ICODE variables with non-overlapping lifetimes or in mutually exclusive conditional branches can be allocated to common data-path storage, most frequently registers. The lifetime analysis is sufficiently sophisticated to also take into account variable persistence in loops as well as conditional branches.
Un-share instruction from unit	An inverse transform which re-assigns a single instruction from a previously shared functional unit to a newly created one. The cell library is interrogated to determine the suitable unit for both the existing and newly created units.

Fully un-share unit	As its name suggests, all instructions currently allocated to the functional unit are each assigned their own data-path unit, employing the previous inverse transform in the process.
Un-share variable from storage unit.	Similar in concept to its functional unit counterpart, a single ICODE variable is assigned its own storage unit.
Fully un-share storage unit	Utilises the transform described above to break-up all instructions mapped to the storage unit in question.
Alternative binding	The sole binding transform enables a data-path unit to be bound to a different implementation chosen to reduce its area or delay characteristics.

Table A.2: Allocation and binding transformations available during optimisation.

Appendix B: Module characteristics

The contents of this appendix describe the properties of the test circuits used to generate the results presented in Chapter 6.

Quartic equation solver

Module	Area (CLB Slices)	Port width (bits)
udivi	908	96
sign	36	33
sdivi	695	96
to_int	154	64
sqrti	1067	64
multi	1194	96
sqi	721	64
quadratic	616	192
cosi	1893	64
acosi	893	64
cbrti	3559	64
cbi	2864	64
program	3530	64
	Σ 18130	

Table B.1: Module characteristics of the Quartic equation solver.

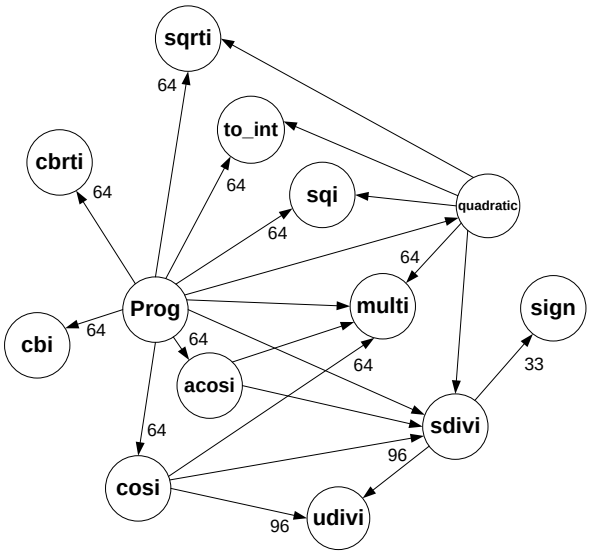


Figure B.1: Relationships between modules in the Quartic equation solver.

Path 1					
multi	→ sign	cosi	to_int	sqrti	quadratic
to_int	udivi	→ udivi	multi	sdivi	→ sqi
multi	cbi	sdivi	to_int	→ sign	to_int
to_int	sqi	→ sign	multi	udivi	multi
multi	sqrti	udivi	sqrti	to_int	to_int
multi	sdivi	multi	to_int	to_int	multi
sqi	→ sign	multi	sdivi	sqi	sqrti
sqi	udivi	cosi	→ sign	to_int	sdivi
multi	acosi	→ udivi	udivi	multi	→ sign
to_int	→ sdivi	sdivi	sqi	to_int	udivi
multi	→ sign	→ sign	to_int	multi	to_int
sqi	udivi	udivi	multi	sqrti	
to_int	multi	multi	sqrti	to_int	
sdivi	to_int	multi	to_int	sdivi	
→ sign	sdivi	cosi	sdivi	→ sign	
udivi	→ sign	→ udivi	→ sign	udivi	
to_int	udivi	sdivi	udivi	sqi	
multi	to_int	→ sign	quadratic	to_int	
multi	sdivi	udivi	→ sqi	multi	
to_int	→ sign	multi	to_int	sqrti	
multi	udivi	multi	multi	to_int	
to_int	sqrti	to_int	to_int	sdivi	
cbi		to_int	multi	→ sign	
multi		sqi		udivi	
to_int					
sdivi					

‘→’ denotes module hierarchy.

Figure B.2: Module execution path of the Quartic equation solver.

Path 2				*Path 3*			
multi	cbi	→ sign	to_int	multi	cbi	→ sign	to_int
to_int	sqi	udivi	multi	to_int	sqi	udivi	multi
multi	sqrti	quadratic	sqrti	multi	sqrti	quadratic	sqrti
to_int	cbrti	→ sqi	to_int	to_int	cbrti	→ sqi	to_int
multi	cbrti	to_int	sdivi	multi	cbrti	to_int	sdivi
multi	to_int	multi	→ sign	multi	to_int	multi	→ sign
sqi	sdivi	to_int	udivi	sqi	sdivi	to_int	udivi
sqi	→ sign	multi	quadratic	sqi	→ sign	multi	quadratic
multi	udivi	sqrti	→ sqi	multi	udivi	sqrti	→ sqi
to_int	to_int	sdivi	to_int	to_int	to_int	sdivi	to_int
multi	sqi	→ sign	multi	multi	sqi	→ sign	multi
sqi	to_int	udivi	to_int	sqi	to_int	udivi	to_int
to_int	multi	to_int	multi	to_int	multi	to_int	multi
sdivi	to_int	to_int	sqrti	sdivi	to_int	to_int	sqrti
→ sign	multi	sqi	sdivi	→ sign	multi	sqi	sdivi
udivi	sqrti	to_int	→ sign	udivi	sqrti	to_int	→ sign
to_int	to_int	multi	udivi	to_int	to_int	multi	udivi
multi	sdivi	to_int	to_int	multi	sdivi	to_int	to_int
multi	→ sign	multi		multi	→ sign	multi	
to_int	udivi	sqrti		to_int	udivi	sqrti	
multi	sqi	to_int		multi	sqi	to_int	
to_int	to_int	sdivi		to_int	to_int	sdivi	
cbi	multi	→ sign		cbi	multi	→ sign	
multi	sqrti	udivi		multi	sqrti	udivi	
to_int	to_int	sqi		to_int	to_int	sqi	
sdivi	sdivi			sdivi	sdivi		
→ sign				→ sign			
udivi				udivi			

'→' denotes module hierarchy

Figure B.3: Alternative module execution paths of the Quartic equation solver.

Cubic equation solver

Module	Area (CLB Slices)	Port width (bits)
udivi	908	96
sign	36	33
sdivi	532	96
to_int	124	64
sqrti	993	64
multi	884	96
sqi	662	64
cosi	1893	64
acosi	898	64
cbrti	3559	64
cbi	2864	64
program	2208	226
	Σ 15561	

Table B.2: Module characteristics of the Cubic equation solver.

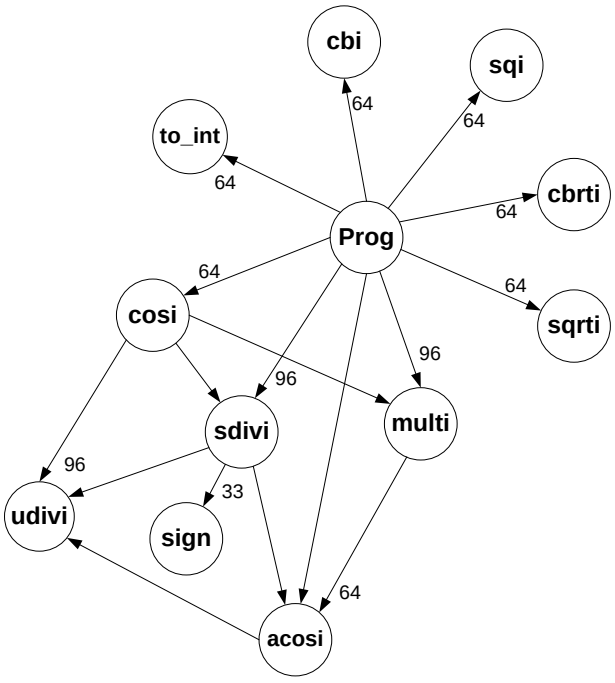


Figure B.4: Relationships between modules in the Cubic equation solver.

Path1			Path2
to_int	→ sign	cosi	to_int
multi	udivi	→ udivi	multi
sqi	multi	sdivi	sqi
to_int	to_int	→ sign	to_int
sdivi	sdivi	udivi	sdivi
→ sign	→ sign	multi	→ sign
udivi	udivi	multi	udivi
to_int	to_int	to_int	to_int
multi	sdivi		multi
to_int	→ sign		to_int
multi	udivi		multi
to_int	sqrti		to_int
cbi	cosi		cbi
multi	→ udivi		multi
to_int	sdivi		to_int
sdivi	→ sign		sdivi
→ sign	udivi		→ sign
udivi	multi		udivi
cbi	multi		cbi
sqi	cosi		sqi
sqrti	→ udivi		sqrti
sdivi	sdivi		cbrti
→ sign	→ sign		to_int
udivi	udivi		sdivi
acosi	multi		→ sign
→ sdivi	multi		udivi
			to_int

'→' denotes module hierarchy

Figure B.5: Module execution paths of the Cubic equation solver.

Quadratic equation solver

Module	Area (CLB Slices)	Port width (bits)
udivi	883	96
sign	36	33
sdivi	342	96
to_int	80	64
sqrti	966	64
multi	721	96
sqi	649	64
program	720	194
	Σ 4397	

Table B.3: Module characteristics of the Quadratic equation solver.

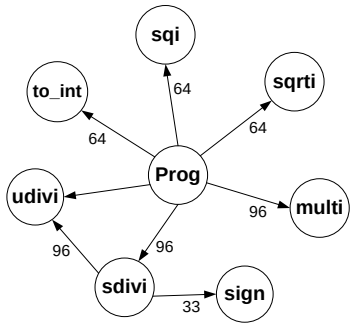


Figure B.6: Relationships between modules in the Quadratic equation solver.

sqi
to_int
multi
to_int
multi
sqrti
sdivi
→ sign
udivi
sdivi
→ sign
udivi
to_int

'→' denotes module hierarchy

Figure B.7: Module execution path of the Quadratic equation solver.

Encryption/Decryption circuits

Module	Area (CLB Slices)	Port width (bits)
rbsub	51	16
rtable	75	40
rotl16	53	64
rot18	53	64
ftable	75	40
logtables	206	20
word	41	40
rco	45	64
fbsub	50	16
rotl24	65	64
reorder	85	64
program	11946	132
	Σ 12745	

Table B.4: Module characteristics of the Encryption/Decryption circuits.

Path1	Path2	Path3
reorder ○ 104 ftable rotl8 ftable rotl16 ftable ○ rotl24 fbsub reorder ○ 104 rtable rotl8 rtable rotl16 rtable ○ rotl24 rbsub	reorder logtables	reorder ○ 2 rotl24 fbsub rco → word ○ fbsub

'→' denotes module hierarchy; **'○'** denotes a finite loop.

Figure B.8: Module execution paths of the Encryption/Decryption circuits.

Matrix circuits

Module	Area (CLB Slices)	Port width (bits)
mzero	430	384
mmult-6	1132	576
madd-5	875	576
smmult	1248	416
mtrans	660	384
finddet2	481	288
finddet3	1336	256
det	1235	224
program	1787	290
	Σ9184	

Table B.5: Module characteristics of the Matrix circuits.

Path1	Path2
reorder ○ 104 ftable rotl8 ftable rotl16 ftable ○ rotl24 fbsub	reorder ○ 2 rotl24 fbsub rco → word ○ fbsub

‘→’ denotes module hierarchy; **‘○’** denotes a finite loop.

Figure B.9: Module execution paths of the Matrix circuits.

Rijndael Encryption/Decryption circuits

Module	Area (CLB Slices)	Port width (bits)
rotl16	53	64
rotl8	53	64
ftable	75	40
word	41	40
rco	46	64
fbsub	51	16
rotl24	65	64
reorder	70	64
program	6929	100
	Σ7383	

Table B.6: Module characteristics of the Rijndael Encryption/Decryption circuits.

Path1	Path2
reorder ○ 104 ftable rotl8 ftable rotl16 ftable ○ rotl24 fbsub	reorder ○ 2 rotl24 fbsub rco → word ○ fbsub

‘→’ denotes module hierarchy; **‘○’** denotes a finite loop.

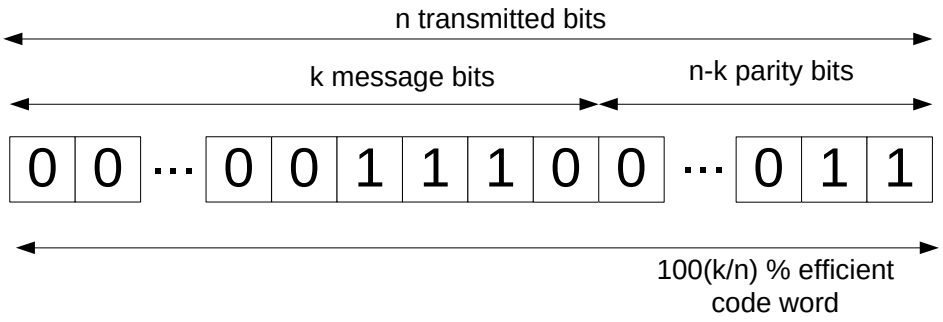
Figure B.10: Module execution paths of the Rijndael Encryption/Decryption circuits.

Appendix C: Case-study

This appendix describes each of the building blocks of the Run-time reconfigurable adaptive coding system described in Chapter 7.

C.1 Message Encoding

The Hamming distance is a measure of the difference between encoded words of data, for instance, the number of bits which vary between two bit patterns. The usefulness of the Hamming distance is that, if an error coding system enforces a minimum distance of two or more than there are bit patterns which do not represent valid data bits. In essence, this concept of a ‘code space’ underpins a considerable body of mathematical theory which is used to derive schemes to produce codes with various properties. A generalised binary class of Hamming codes for multiple-error the correction are the Bose-Chaudhuri-Hocquengheim (BCH) [85] codes.



n	k	d_{min}	Code generator (octal)	2^{n-k} states	100 (k/n) % efficiency	$(d_{min}-1)/2$ correctable errors
7	4	3	13	8	57.1%	1
15	11	3	23	16	73.3%	1
15	7	5	721	256	46.7%	2
15	5	7	2467	1024	33.3%	3
31	26	3	45	32	83.9%	1
31	21	5	3551	1024	67.7%	2
31	16	7	107675	32768	51.6%	3
31	11	11	5423325	1048576	35.5%	5
31	6	15	313365047	33554432	19.4%	7

Figure C.1: Format of a BCH codeword, exemplar codes and the target codes.

Figure C.1 shows the format of a BCH codeword and a number of exemplar short length codes. A given BCH code takes the form BCH (n, k, d_{min}), where a data message of length *k* is augmented with *n-k* parity bits to realise a codeword of length *n*. The Hamming distance $d_{\min} \leq 2t+1$ enables the correction of $t = \frac{d_{\min}-1}{2}$ errors. Also tabulated in Figure C.1 are the parity overheads associated with the size of each message, expressed as a percentage of the codeword transmitted and hence an expression of the code efficiency. The (15,k,d_{min}) family of codes highlighted in the table were selected because they facilitate the correction of a range of errors (one to three) whilst requiring a generally modest number of states. As the table shows, the number of states associated with a BCH code grows with the number of errors it is capable of correcting.

The message to be transmitted is stored in a message ROM, where each character is represented in the form of ASCII code. The ASCII character cannot always be encoded directly since the portion of code allocated to data varies depending upon the coding scheme. Figure C.2 shows how a series of characters is formatted in accordance with *k* message bits for a given BCH code, where a coloured bit denotes a splice in the ASCII code and n bits are added to meet the required length.

character	ASCII code	k=11	k=7	k=5
a	1000001	00101000001	1000001	00001
b	1000010	01000011100	1000010	01010
c	1000011	00101100010	1000011	11000
d	1000100	xx100011010	1000100	00001
e	1000101		1000101	01001
f	1000110		1000110	01100
				10001
				00110
				xxx10

Figure C.2: Code dependent message formation.

Once the bit-packing stage is complete, the next task in the construction of the codeword is the generation of the parity bits. Figure C.3 depicts a Galois type Linear Feedback Shift Register (LFSR) circuit capable of encoding a message for the (15,11,3) BCH code. The feedback connections to the exclusive-or gates correspond to the code's generator: $23_8 = 19_{10} = 10011_2$ or expressed as a polynomial: $g(x) = 1 + x + x^4$.

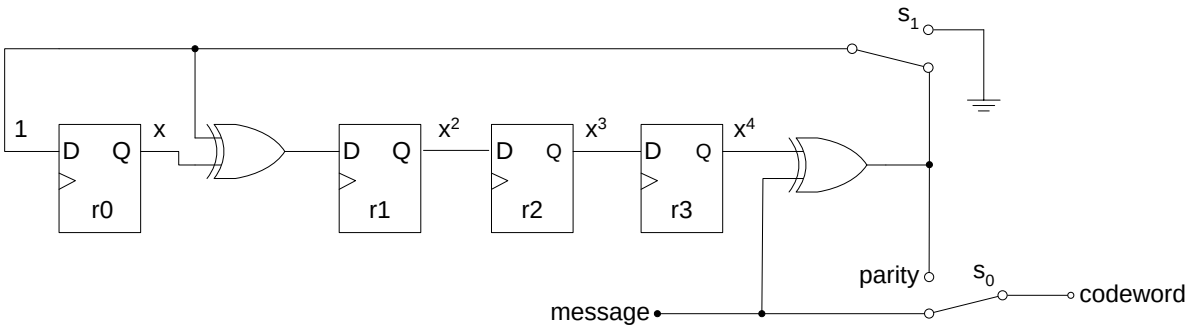


Figure C.3: Encoding circuit for the BCH (15,11,3) code.

The switches are used (conceptually) to direct the source of the bits that form the codeword as well as controlling the feedback of the LFSR. Consider encoding the message taken from the first row of the table shown in Figure C.3 for message length $k=11$: 00101000001.

Figure C.4 tabulates the progression of the message through the registers. During the first $k=11$ cycles each bit of the message is exclusive-ORed with the coefficient of x^4 (held in r3) before being fed back (switch s_1) to be once more exlusive-ORed with the coefficient of x. Simultaneously, each bit of the message is also directed via switch s_0 to form the first k bits of the codeword. In the remaining $n\text{-}k = 15\text{-}11 = 4$ cycles, the switches are flipped to flush the registers and append their contents as parity, thus forming the codeword 0000 00101000001.

On close inspection, there are a number of characteristics which determine the sequence of the register contents. Regarding the LFSR as a state machine, where the value of the register outputs correspond to a particular state, the next state is dependent upon the output of the last register Q_3 and the value of the message bit M_i . The effect is to determine whether the register value is doubled by shifting right one place or added to the generator (modulo-2 arithmetic).

message	m_i	D0	Q0	D1	Q1	D2	Q2	D3	Q3	codeword	$\|Qn\|$
00101000001	0	1	0	1	0	0	0	0	0	-	0
0010100000	1	0	1	1	1	1	0	0	0	1	3
001010000	2	0	0	0	1	1	1	1	0	01	6
00101000	3	1	0	1	0	0	1	1	1	001	12
0010100	4	1	1	0	1	1	0	0	1	0001	11
001010	5	0	1	1	0	0	1	1	0	00001	5
00101	6	0	0	0	1	1	0	0	1	000001	10
0010	7	0	0	0	0	0	1	1	0	1000001	4
001	8	0	0	0	0	0	0	0	1	01000001	8
00	9	0	0	0	0	0	0	0	0	101000001	0
0	10	0	0	0	0	0	0	0	0	0101000001	0
-	11	0	0	0	0	0	0	0	0	00101000001	0
0000	11	0	0	0	0	0	0	0	0	00101000001	-
000	12	0	0	0	0	0	0	0	0	0 00101000001	-
00	13	0	0	0	0	0	0	0	0	00 00101000001	-
0	14	0	0	0	0	0	0	0	0	000 00101000001	-
-	15	0	0	0	0	0	0	0	0	0000 00101000001	-

Figure C.4: Exemplar LFSR encoding.

Let Q represent the value of the register contents given when the MSB is logic 1 i.e. $Q = 2^{n-k-1} = 2^{15-11-1} = 8$; Recall that the Generator polynomial G expressed in binary form is 10011_2. The generator g is found by exclusive-oring the polynomial generator G with the value of Q i.e. $g = 1011_2 \oplus 1000_2 = 0011_2$. If R_i represents the state of the registers after message bit i has been shifted in and R_{i+1}denotes the next state then:

$$R_{i+1} = 2R_i \text{ when } (R_i < Q \text{ and } M_i = 0) \text{ or } (R_i \geq Q \text{ and } M_i = 1) \quad (C.1).$$

In words, the next state is *even* when either the current state is less than $Q$and the message bit is logic low or the current state is greater or equal to $Q$and the message bit is logic

high. For example, with reference to the table in Figure C.4 the state at message bit 2 is given by: $R_2 = 2 \cdot R_1 = 2 \cdot 3 = 6$ when $M_1 = 0$.

Similarly the state at message bit 7 is:

$$R_7 = 2 \cdot R_6 = 2 \cdot 10 = 20 = 4 \text{ where } Q + R = 10 \text{ and } M_6 = 1.$$

Although the next state is twice that of the previous value of 10, it cannot be represented as 20 using 4 bits and therefore the next state is returned as 4.

Complementary to that, it also follows that:

$$R_{i+1} = 2R_i \oplus g \text{ when } (R_i < Q \text{ and } M_i = 1) \text{ or } (R_i \geq Q \text{ and } M_i = 0) \quad (\text{C. } 2).$$

Returning to the table for verification, the state at message bit 1 is given by:

$$R_1 = 2 \cdot R_0 \oplus 3 = 2 \cdot 0 \oplus 3 = 3 \text{ when } M_0 = 1.$$

Likewise, state 5 is given by:

$$R_5 = 2 \cdot R_4 \oplus 3 = 2 \cdot 11 \oplus 3 = 5 \text{ when } M_0 = 0.$$

Figure C.5 depicts the state transition for any given present state R_i to the next R_{i+1}, with respect to the value of the message bit M_i.

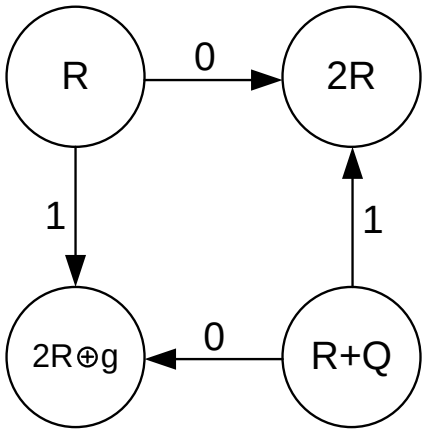


Figure C.5: Message dependent state transition.

This process is encapsulated by the pseudo-code depicted in Figure C.6. The code is self-explanatory, in that, it is solely a description of the behaviour. The VHDL description in MOODS is almost identical with some additional type casting of the variables. It can be found in Chapter 3, Figure 3.3. At this algorithmic level of the abstraction, the function of the encoder is transparent in the circuit design; the same cannot be said of the Register Transfer Level descriptions of the circuit shown in Figure 3.3 in Chapter 3.

```
G = 19
numStates = 2^(codeLength−messageLength).
Q = 2^(codeLength−messageLength−1)
g = G ⊕ Q
state = 0
for i in 0 to messageLength loop
  if state < Q then
    if message_i = 0 then
      state = 2 · state
    else
      state = 2 · state ⊕ g
    end if
  else
    if message_i = 0 then
      state = 2 · (state − q) ⊕ g
    else
      state = 2 · (state − q)
    end if
  end if
end loop
```

Figure C.6: Algorithmic description of the BCH encoder.

C.2 Message Decoding − Sequential Viterbi Decoder

The state transition diagram of Figure C.7 highlights the sequence of states traversed during the encoding of the exemplar codeword of Figure C.3, in the context of all possible state transitions. Each path through the states is unique to the contents of the message. The task during decoding is to determine that path and in doing so, whether or not any message bit has been corrupted during transmission, restoring it where necessary. As it has been shown that during encoding, each state R or $R + Q$ traverses to the next state $2 \cdot R$ (even) or $2R_i \oplus g$ (odd) depending upon the logic level of the message bit M_i. Working backwards, this property can be used to determine the pair of preceding states. Only one of these states is associated with the state transition taken during the encoding of the codeword. Identifying it, will also reveal the message bit transmitted.

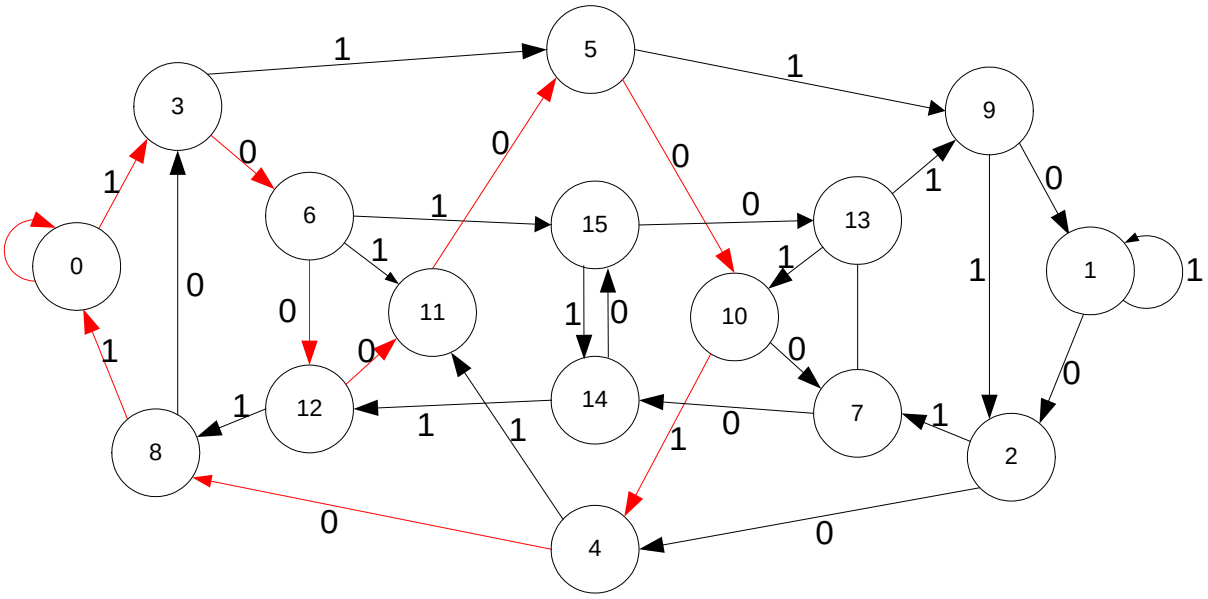


Figure C.7: State Transition Diagram for message decoding using BCH (15,11,3) code.

The Hamming distance can be quantified at each state transition by introducing a weight that is dependent on the logic level of the message bit. A weight of zero represents no change of the message bit during transmission, where as an increment in weight is used to reflect its corruption.

Returning to the state diagram, consider the state transition (R_0 to state R_3) associated with the first bit of the codeword $M_0 = 1$. Assume for the time being, that no error occurred in the transmission of the message bit. A weight of 0 at state R_3 can be represented for the first message bit by: $W_3^0 = 1 - M_0 = 0$. The weight would have been incremented had the message bit been corrupted during transmission i.e. $W_3^0 = 1$. For the sake of argument, let us assume that the transition also occurred error free for the second message bit $M_1 = 0$. To have negligible effect on the weight, the value of the message bit can be assigned directly i.e. $W_6^1 = M_1 = 0$. In this way, any error to the message bit ($M_1 = 1$) also results in an increment to the weight, thus: $W_6^1 = M_1 = 1$. The Viterbi decoder is now in a position to determine which of the predecessor states is associated with the message. It does this by comparing each of their weights and selecting the state whose weight is the smaller of the two. The relationship between the current state and its predecessor determines the value of the message bit. The next state relationships used by the encoder are reproduced here:

$$R_{i+1} = 2R_i \text{ when } (R_i < Q \text{ and } M_i = 0) \text{ or } (R_i \geq Q \text{ and } M_i = 1) \qquad (C.1)$$

$$R_{i+1} = 2R_i \oplus g \text{ when } (R_i < Q \text{ and } M_i = 1) \text{ or } (R_i \geq Q \text{ and } M_i = 0) \qquad (C.2)$$

Using these equations or Figure C.6 and working backwards provides the preceding state, that is to say, the state R_{i-1}which along with the message bit M_i is responsible for the state transition the current state R_i i.e.

$$\text{when } R_i = 2R_i,\ R_{i-1} = R_i \text{ and } M_i = 0 \text{ or } R_{i-1} = R_i + \mathrm{Q} \text{ and } M_i = 1$$

$$\text{when } R_i = 2R_i \oplus g,\ R_{i-1} = R_i \text{ and } M_i = 1 \text{ or } R_{i-1} = R_i + \mathrm{Q} \text{ and } M_i = 0$$

During the encoding of the message, the starting state is always R_0. As shown, this characteristic is exploited in the decoder weighting of each state by initializing all the states to a weight greater than that of R_0 i.e. the minimum Hamming distance. In doing so, the initial weighting ensures that state R_0 is correctly chosen as the predecessor state when decoding the first message bit. The decoding method is summarised by the pseudo-code shown in Figure C.8. Initialisation takes place during lines 1-4, to bias the weight of state R_0 to zero and the rest of the states to the minimum Hamming distance. The internal loop bounding lines 6-21 compares the weights of each pair of predecessor states for every state *R*. The smaller of the

two, points the way to what is likely to have been the previous state. Using the relationships described earlier, the message bit likely to be associated with the state transition is found and consequently set. Note that there is no point in performing the extra test to determine whether the bit is erroneous, rather the bit is directly updated with the value expected. All that remains is to write the smaller of the two weights to the state *R* in question. Of course, the procedure is repeated by the outer loop for every remaining bit i of the message.

1. $w_0^{-1} = 0$
2. $for\ R\ in\ 1\ to\ numStates - 1$ loop
3. $w_R^{-1} = dmin$
4. $end\ loop$
5. $for\ i\ in\ 0\ to\ messageLength - 1\ loop$
6. $for\ R\ in\ 0\ to\ numStates - 1\ loop$
7. $if\ w_R^{i-1} + m_i < w_{R+Q}^{i-1} + 1 - m_i\ then$
8. $message_i = '0'$
9. $w_{2R}^{i} = w_R^{i-1} + m_i$
10. $else$
11. $w_{2R}^{i} = w_{R+Q}^{i-1} + 1 - m_i$
12. $message_i = '1'$
13. $end\ if$
14. $if\ w_R^{i-1} + 1 - m_i < w_{R+Q}^{i-1} + m_i\ then$
15. $message_i = '1'$
16. $w_{2R\oplus g}^{i} = w_R^{i-1} + 1 - m_i$
17. $else$
18. $message_i = '0'$
19. $w_{2R\oplus g}^{i} = w_{R+Q}^{i-1} + m_i$
20. $end\ if$
21. $end\ loop$
22. $end\ loop$

Figure C.8: Algorithmic description of the Viterbi decoder.

Table C.1 depicts each pair of predecessor states read by the algorithm for the BCH (15,11,3) coding scheme and the current odd and even states they are used to update.

Previous states R_{i-1}		Current states R_i	
R	R+Q	2R	2R⊕g
0	8	0	3
1	9	2	1
2	10	4	7
3	11	6	5
4	12	8	11
5	13	10	9
6	14	12	15
7	15	14	13

Table C.1: States visited by the algorithm during the decoding of the BCH (15,11,3) code.

The next table (C.2) shows the Hamming distances calculated during the decoding of the codeword described earlier and illustrated in Figure C.4. The first column depicts the initialisation of the states, which as explained earlier, requires that state R_0 be initialised to zero and the others set to the minimum code distance. In doing so, the method ensures that all states can be correctly traced back to R_0, thus mirroring the origin of the first state transition during encoding. The codeword is error free and so the initial zero weight is propagated (shown in bold) as the decoder recreates the path likely to have been taken during the encoding of the codeword. This can be verified through the use of Table C.1 and the equations for the even and odd weights featured in the decoder algorithm equation and reproduced in equations C.3 and C.4 respectively:

$$w_{2R}^{i} = \text{minimum}\,(w_R^{i-1} + m_i, w_{R+Q}^{i-1} + 1 - m_i) \quad \text{(C.3)}$$

$$w_{2R\oplus g}^{i} = \text{minimum}\,(w_R^{i-1} + 1 - m_i, w_{R+Q}^{i-1} + m_i) \quad \text{(C.4)}$$

Starting with the even state R_0, from Table C.1, its preceding states are itself R_0 and R_8. Inspection of the first column of Table C.2 shows the initial weights of both states to be: W_0=0 and W_8=3. Using Equation C.3 and taking into consideration the value of the message bit m_0=1, the weight at state R_0 is determined as: $W_0^0 = \text{minimum}(1,3) = 1$.

$\mathbf{M_i}$	$\mathbf{M_{-1}}$	$\mathbf{M_0}$	$\mathbf{M_1}$	$\mathbf{M_2}$	$\mathbf{M_3}$	$\mathbf{M_4}$	$\mathbf{M_5}$	$\mathbf{M_6}$	$\mathbf{M_7}$	$\mathbf{M_8}$	$\mathbf{M_9}$	$\mathbf{M_{10}}$	$\mathbf{M_{11}}$	$\mathbf{M_{12}}$	$\mathbf{M_{13}}$	$\mathbf{M_{14}}$
$\mathbf{W_i}$	-	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**
$\mathbf{W_0}$	**0**	1	1	1	1	1	1	2	2	**0**	**0**	**0**	**0**	**0**	**0**	**0**
$\mathbf{W_1}$	3	3	3	3	2	3	1	1	2	2	2	2	1	1	1	1
$\mathbf{W_2}$	3	3	3	3	3	2	2	1	1	2	2	2	2	1	1	1
$\mathbf{W_3}$	3	**0**	2	2	2	1	2	1	1	1	1	1	1	1	1	1
$\mathbf{W_4}$	3	3	3	3	2	3	2	**0**	1	2	2	2	2	2	1	1
$\mathbf{W_5}$	3	3	1	3	3	**0**	2	2	2	1	1	1	1	1	1	1
$\mathbf{W_6}$	3	3	**0**	2	2	1	1	2	1	1	1	1	1	1	1	1
$\mathbf{W_7}$	3	3	3	3	1	3	2	1	2	1	1	1	1	1	1	1
$\mathbf{W_8}$	3	3	3	3	1	2	3	1	**0**	2	2	2	2	2	2	1
$\mathbf{W_9}$	3	3	3	2	3	1	1	2	2	2	2	1	1	1	1	1
$\mathbf{W_{10}}$	3	3	3	1	3	2	**0**	2	2	1	1	1	1	1	1	1
$\mathbf{W_{11}}$	3	3	3	3	**0**	2	2	2	1	1	1	1	1	1	1	1
$\mathbf{W_{12}}$	3	3	3	**0**	2	2	1	2	2	1	1	1	1	1	1	1
$\mathbf{W_{13}}$	3	3	3	3	1	2	2	2	1	2	1	1	1	1	1	1
$\mathbf{W_{14}}$	3	3	3	3	2	1	3	1	1	1	1	1	1	1	1	1
$\mathbf{W_{15}}$	3	3	3	1	3	2	1	1	1	1	1	1	1	1	1	1

Table C.2: Weights at each state and for every bit of the codeword 000000101000001.

In much the same way, the decoding algorithm also updates the weight of the odd state R_3, except using equation C.4, thus the weight of state R_0 is propagated to state R_3: $W_3^0 = \text{minimum}(0,4) = 0$. The algorithm repeats this procedure for the remaining states and for every bit of the message M_i being decoded. One interesting characteristic of BCH codes is that if the codeword were to be encoded using the same generator, the final state is always R_0. In a similar sense, this property is exhibited by the decoder, since the weight of R_0 following

the last message bit returns the number of errors corrected i.e. $W_0^{14} = 0$. As will be described in due course, this property is also exploited during control of the adaptive decoding, where the number of errors corrected guides the selection of the coding scheme. Table C.3 shows the eights generated when the algorithm is applied to a corrupted codeword, the result of a switch in the logic level of the fourth message bit M_3.

$\mathbf{M_i}$	$\mathbf{M_{-1}}$	$\mathbf{M_0}$	$\mathbf{M_1}$	$\mathbf{M_2}$	$\mathbf{M_3}$	$\mathbf{M_4}$	$\mathbf{M_5}$	$\mathbf{M_6}$	$\mathbf{M_7}$	$\mathbf{M_8}$	$\mathbf{M_9}$	$\mathbf{M_{10}}$	$\mathbf{M_{11}}$	$\mathbf{M_{12}}$	$\mathbf{M_{13}}$	$\mathbf{M_{14}}$
$\mathbf{W_i}$	-	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**
$\mathbf{W_0}$	0	1	1	1	1	1	1	2	1	1	1	1	1	1	1	1
$\mathbf{W_1}$	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1
$\mathbf{W_2}$	3	3	3	3	3	3	2	1	2	1	2	1	1	1	1	1
$\mathbf{W_3}$	3	0	2	2	2	1	2	1	0	1	2	1	1	1	1	1
$\mathbf{W_4}$	3	3	3	3	2	3	2	1	1	2	1	0	1	1	1	1
$\mathbf{W_5}$	3	3	1	3	3	1	2	2	1	0	1	0	1	1	2	1
$\mathbf{W_6}$	3	3	0	2	2	0	1	2	1	1	1	2	2	1	1	1
$\mathbf{W_7}$	3	3	3	3	1	3	1	2	2	2	1	0	1	2	1	1
$\mathbf{W_8}$	3	3	3	3	1	2	3	0	1	2	2	1	1	1	1	1
$\mathbf{W_9}$	3	3	3	2	3	2	1	2	1	1	1	1	1	1	1	0
$\mathbf{W_{10}}$	3	3	3	1	3	1	1	2	2	1	0	1	2	1	1	1
$\mathbf{W_{11}}$	3	3	3	3	0	2	2	1	2	1	2	1	1	2	1	1
$\mathbf{W_{12}}$	3	3	3	0	2	2	0	2	2	2	1	1	2	1	1	1
$\mathbf{W_{13}}$	3	3	3	3	1	1	2	1	1	2	1	1	1	1	0	1
$\mathbf{W_{14}}$	3	3	3	3	2	2	3	1	2	1	2	1	0	1	1	1
$\mathbf{W_{15}}$	3	3	3	1	3	2	1	1	1	1	1	2	1	0	1	1

Table C.3: Weights at each state and bit of the erroneous codeword 000000101010001.

As before, the decoder uses the minimum weights to identify the correct path (in bold) despite the presence of the erroneous message bit. Table C.4 illustrates the effect of the algorithm upon the reconstruction of the codeword itself. Each column of the table refers to the individual bit of the message or codeword under examination by the algorithm. Every row

depicts the partial construction of the message local to each state, as each bit of the message is examined.

M_i	M_{-1}	M_0	M_1	M_2	M_3	M_4	M_5	M_6	M_{14}
$Message_{Ri}$	-	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**0**
$Message_0$	-	0	00	000	0000	11001	011001	0011001…	000000101000001
$Message_1$	-	1	01	001	0111	10111	000101	0010011…	000000101010000
$Message_2$	-	1	01	001	1111	10001	010111	1010011…	000100101010001
$Message_3$	-	1	10	100	1000	10000	010010	1011001…	000001101010001
$Message_4$	-	1	01	001	1011	10001	110101	1000001…	000000101110001
$Message_5$	-	1	11	001	0001	00001	011011	1010010…	001000101010001
$Message_6$	-	1	01	010	0100	10001	010000	1011101…	000000100010001
$Message_7$	-	1	01	001	0011	11111	010101	0000001..	000000111010001
$Message_8$	-	1	01	001	1001	10010	111101	1010001…	000000101011001
$Message_9$	-	1	01	111	0001	00101	010011	1011011…	000000101010001
$Message_{10}$	-	1	01	011	0001	10101	000001	1010100…	100000101010001
$Message_{11}$	-	1	01	001	0001	11011	011101	0010001…	000000101010101
$Message_{12}$	-	1	01	001	0010	11101	010001	0010000…	000010101010001
$Message_{13}$	-	1	01	001	0101	10011	010100	1010101…	000000101010011
$Message_{14}$	-	1	01	001	1101	00011	110100	1110001...	000000001010001
$Message_{15}$	-	1	01	101	0001	10100	110001	1010000…	010000101010001

Table C.4: Message correction using Viterbi decoding for codeword 000000101010001.

Having identified the likely predecessor state (from the smaller weight of the two states presented), the algorithm uses the relationships described by equations C.1 and C.2 to set the likely logic level of the message bit. As Table C.4 depicts, once the logic level of message bit M_i has been found, the entire message up to and including bit i is appended to the message associated with the states under examination. As highlighted in bold in table, the original path associated with the state transitions is re-traced and the driving message bits responsible for each transition, are reconstructed bit by bit, thereby correcting the incorrectly received message bit in the process. After the last message bit has been decoded, the corrected codeword is read from the copy of the codeword associated with state R_0. With reference to

Table C.3, once again, the weight of state R_{14} after the last message bit has been decoded returns the number of errors that have been corrected.

C.3 Parallel Viterbi Decoding

The decoding time of the sequential decoder can be reduced linearly by partitioning the number of states evenly among 2^m decoders or 'Processors' [124,125]. In the sequential decoder the weights and partially corrected codeword associated with each state are accessible to all other states within the decoder circuit. In practice, the Viterbi decoders used in the reconfigurable coding scheme require ($2^{codeLength-messageLength}$) of 16-bit words to store the message and the same number of nibbles (4 bits) to record the hamming distance weights associated with each of the states. Partitioning the states over a number of decoders can potentially generate a large cut-set comprising the memory allocated to the Hamming weights and the *i* th bit of the codeword.

The underlying characteristic thus far, has been the classification of the states into those that are smaller or (greater and equal) to *Q* and their use in the generation of the next set of states. Grouping these states together on each decoder can significantly reduce the number of signals cut by the partitioning.

Equation C.5 determines how 2^{n-k-1}states can be evenly partitioned over 2^m processors. Recall that *n* is the length of the codeword and that *k* is the portion of that codeword assigned to the message bits. The set of states S_P partitioned over processor *P* is given by:

$$S_P = P_R \cup P_{R+Q} \quad (C.5)$$

Where the set of states less than *Q* on processor *P* is given by:

$$P_R = LP \ldots L(P+1) - 1 \quad (C.6)$$

Let *L* be the number of states less than *Q* on each processor *P*, such that:$L = 2^{n-m-k-1}$ and $Q = 2^{n-k-1}$. Those states on processor *P* greater than *Q* are given by adding *Q* to the decimal value of each state in the set i.e. $P_{R+Q} = P_R + Q \quad (C.7)$.

Consider the task of partitioning the BCH code (15,11,3) over 4 processors. The parameters are as follows:

$$n = 15, k = 11, m = 2 \text{ for } 2^2 = 4 \text{ processors}$$

$$Q = 2^{n-k-1} = 2^{15-11-1} = 8;\ L = 2^{15-2-11-1} = 2^1 = 2.$$

The parameters indicate that there are 2 states less than 8 on each processor. These states are determined using equations C.5 for each of the processors numbered 0 to 3 as follows:

States on processor 0

States less than Q: $P_R = 0 * 2 \ldots 2(0+1) - 1 = 0..1 \quad (\text{using } C.6)$

States greater than or equal to Q: $P_{R+Q} = 0 + 8 \ldots 1 + 8 = 8 \ldots 9 \quad (\text{using } C.7)$

Therefore the states partitioned to processor 0 are given by combining both sets:

$$S_0 = 0 \ldots 1 \cup 8 \ldots 9 = 0,1,8,9 \quad (\text{using C.5})$$

Using the same equations, the remaining states are partitioned as follows:

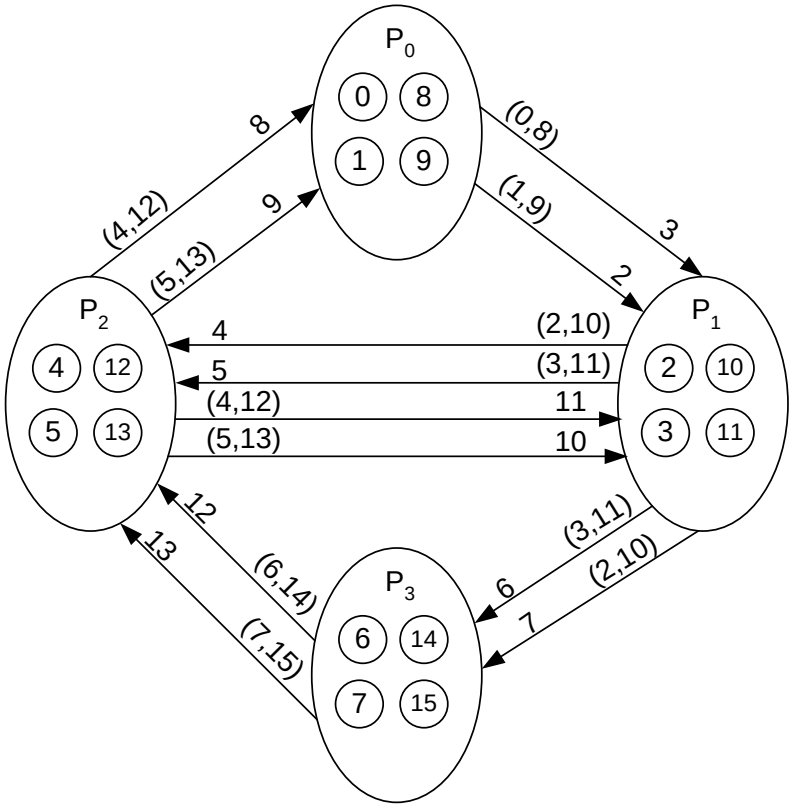
$$S_1 = 2 \ldots 3 \cup 10 \ldots 11 = 2,3,10,11$$

$$S_2 = 4 \ldots 5 \cup 12 \ldots 13 = 4,5,12,13$$

$$S_3 = 6 \ldots 7 \cup 14 \ldots 15 = 6,7,14,15$$

Figure C.9 illustrates the external communication (cut-set) between each of the four processors $P_{1\text{-}3}$. Each processor depicted, contains the states assigned to it using the procedure described. Recall that in order to update the weight and message bits associated with a given state R_i, two predecessor states R_{i-1} are required. The relationship between such states is re-produced in the table alongside the decoders. The transfer of the weights and message bits between processors is depicted by each unidirectional edge. Each arc originates from the processor containing the predecessor states and terminates in the processor whose state is updated using the predecessor states and the decoding method discussed earlier. What makes the partitioning particularly elegant is that a significant proportion of the connectivity is contained within the processor itself. For example, consider the even and odd states of the

first row of the table. The even state R_i= 0 is updated using itself and R_{i-1}=8. This is done internal to the processor. The two states are also used to update the odd state R_i= *3* which requires the weight and message bits associated with each state to be transmitted across the decoder boundaries.



R_{i-1}		R_i	
W_R	W_{R+Q}	W_{2R}	$W_{2R\oplus g}$
0	8	0	3
1	9	2	1
2	10	4	7
3	11	6	5
4	12	8	11
5	13	10	9
6	14	12	15
7	15	14	13

Figure C.9: Viterbi decoding of BCH (15,11,3) code partitioned over 4 processors.

As another example of the partitioning, consider the state assignment necessary to divide the decoding over two processors. Once again, the parameters are returned as:

$$n = 15, k = 11, m = 1 \text{ for } 2^1 = 2 \text{ processors}$$

$$Q = 2^{n-k-1} = 2^{15-11-1} = 8;\ L = 2^{15-1-11-1} = 2^2 = 4.$$

<u>States on processor 0</u>

States less than *Q*: $P_R = 0 * 4 \ldots 4(0+1) - 1 = 0..3$ (using $C.6$)

States greater than or equal to *Q*: $P_{R+Q} = 0 + 8 \ldots 3 + 8 = 8 \ldots 11$ (using $C.7$)

Combining both sets of states determines those state partitioned to processor 0 i.e.

$$S_0 = 0 \ldots 3 \cup 8 \ldots 11 = 0,1,2,3,8,9,10,11 \quad (\text{using } C.5)$$

Following the same method reveals the remaining states portioned over processor 1:

$$S_1 = 4 \ldots 7 \cup 12 \ldots 15 = 4,5,6,7,12,13,14,15$$

Figure C.10 shows the impact of the partitioning on the external communication between the processors. In this case, partitioning the states among two processors balances the number of internal signal transfers (16) with those exchanged between the two decoders. The reader may verify this by consulting the relationships between the states shown in the table of Figure C.9. Another aspect of the partitioning which commends itself to parallel computation is the fixed degree of input and output arcs. In general the in/out-degree of the task graph when decoding any BCH code is never any greater than 4.

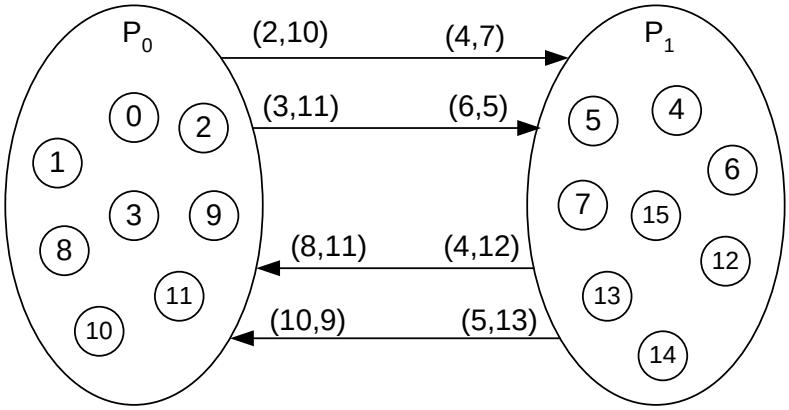


Figure C.10: Viterbi decoding of BCH (15,11,3) code over 2 processors.

In addition to partitioning the BCH (15,11,3) code over 4 processors, the states of the BCH (15,7,5) code are divided among 2 processors. The rationale for doing this is discussed in the next section. Repeated application of the equations used previously, returns the following state to processor assignment: out of the 256 states used by the decoder, each processor will be assigned 64 states, all of which are numerically smaller than Q which is found to be 128 i.e. $Q = 128; L = 64;$ given that $n = 15, k = 7, m = 1$ for $2^1 = 2$ processors.

These states are partitioned to processor 0, such that:

States less than Q: $P_R = 0 * 64 \ldots 64(0+1) - 1 = 0..63$ (using $C.6$)

States greater than or equal to Q: $P_{R+Q} = 0 + 128 \ldots 63 + 128 = 128 \ldots 191$ (using $C.7$)

The states assigned to processor 0 are as follows:

$$S_0 = 0 \ldots 63 \cup 128...191 \quad (C.5)$$

Repeating the procedure verifies that those states not assigned to processor 0 are resident on the remaining processor i.e.

$$S_1 = 64 \ldots 127 \cup 192 \ldots 255 .$$

C.4 Message Corruption

As its name suggests, the purpose of the message corrupter circuit is to invert one to four bits of a given codeword and in doing so offer a simple means of emulating the corruption of any encoded message during its transmission. The role of the corrupter is limited to automating the selection of random bits of a codeword for corruption- in order to stimulate a change in coding scheme by the receiver. The reader is referred to Chapter 7 for a full description of its integration within the adaptive coding scheme.

Figure C.11 shows the behavioural VHDL description of the message corrupter used in the communication system. It is essentially a description of the behaviour of two linear feedback shift registers and is therefore a specification for two pseudo-random number generators. The first produces a pseudo random number between 0 and 63; it is converted to the variable ‘numErrors’ which is used to determine the number of codeword bits to change during corruption of a codeword. To vary the error rate, the frequency of bit errors is determined by the size of the variable ‘numErrors’ in relation to a number interval. No errors (numErrors=0) are most likely to occur for random numbers in the range of 0 to 39; conversely, four errors (numErrors=4) is the least likely number of errors to occur requiring the random number to be exactly equal to 63.

```
-- subroutine to corrupt one to four bits of a codeword

procedure messageCorrupter (codeWord: in std_logic_vector(14 downto 0);
                            randomState: inout std_logic_vector(5 downto 0);
                            randomStateBit: inout std_logic_vector(3 downto 0);
                            corruptCodeWord: out std_logic_vector(14 downto 0);
                            errors: out integer range 0 to 4) is

   variable tempWord: std_logic_vector(14 downto 0);
   variable taps,eTaps: std_logic;
   variable numErrors: integer range 0 to 4;
   variable randomBit: integer range 0 to 14;
   variable randomNo: integer range 0 to 63;

   begin

      -- LFSR which determines the number of errors

      eTaps:=randomState(0) xor randomState(1);
      randomState:=eTaps & randomState(5 downto 1);
      randomNo:=to_integer(unsigned(randomState));

      if randomNo < 39 then
        numErrors:=0;
      elsif randomNo < 51 then
        numErrors:=1;
      elsif randomNo < 59 then
        numErrors:=2;
      elsif randomNo < 63 then
        numErrors:=3;
      else
        numErrors:=4;
      end if;

      tempWord:=codeWord;

      -- LFSR which selects the codeword bits to corrupt

      while numErrors > 0 loop
          taps:=randomStateBit(0) xor randomStateBit(1);
          randomStateBit:=taps & randomStateBit(3 downto 1);
          randomBit:=to_integer(unsigned(randomStateBit))- 1;
          tempWord(randomBit):= tempWord(randomBit) xor '1';
      end loop;

      corruptCodeWord:=tempWord;

end messageCorrupter;
```

Figure C11: Behavioural VHDL description of the message corrupter circuit.

The role of the second LFSR is to perform the corruption of the codeword by pseudo-randomly selecting the desired number of codeword bits held by variable numErrors, inverting each bit to achieve the specified level of codeword corruption.

The main objective of the case-study was to provide a means of exercising the automated RTR infrastructure generated by MOODS during temporal partitioning. However, all building blocks described in chapter 7 were implemented at the device-level as part of a rudimentary communication system. As well as successfully testing the automated infrastructure, the case study showed the feasibility of using RTR to utilise logic resources to increase the parallelism of Viterbi decoding using FPGA resources which would have otherwise been idle.

The next step for the case study would be to change the message corrupter to implement the characteristics associated with actual communication channels. A good place to start would be to incorporate a variable error rate into the message corrupter based upon a Markov Process, as described in [123]. In doing so, the codewords would be subjected to a variable error rate typical of the sort of signal ‘fading’ associated with wireless communication channels. Updating the message corrupter will enable the reconfigurable Viterbi decoders to be assessed alongside more conventional approaches, where its advantages and disadvantages can be further evaluated.

References

1 Freeman, R., “Configurable electrical circuit having configurable logic elements and configurable interconnects”, US Patent 4,870,302, September 26, 1989.

2 Brown, S. − Rose, J., “FPGA and CPLD architectures: a tutorial”, IEEE Design Test of Computers, Vol. 12, No. 2, 1996, pp.42-57.

3 DeHon, A., “Dynamically Programmable Gate Arrays: A Step Toward Increased Computational Density”, Proceedings of the Fourth Canadian Workshop on Field-Programmable Devices, Toronto, Canada, May 1996, pp.47-54.

4 Baker, K.R., “Multiple Objective Optimisation of Data and Control Paths in a Behavioural Silicon Compiler”, PhD Thesis, University of Southampton, September 1992.

5 Williams, A.C., “A Behavioural VHDL Synthesis System using Data path Optimisation”, PhD Thesis, University of Southampton, July 1997.

6 “Virtex 2.5 Field Programmable Gate Array”, Datasheet DS003-1, Xilinx Inc., 2013.

7 Kirkpatrick, S., “Optimization by Simulated Annealing: Quantitive Studies”, Journal of Statistical Physics, Vol. 34, Nos. 5/6, March 1984, pp.975-986.

8 Walder, H. − Platzner, M., "A runtime environment for reconfigurable hardware operating systems", Proceedings of the 14th Conference on Field-Programmable Logic and Applications, January, 2004, pp.831-835.

9 Lyke, J.C., et al., “An Introduction to Reconfigurable Systems”, Proceedings of the IEEE, Vol. 103, No.3, March 2015, pp.291-317.

10 Compton, K. − Hauck, S., “Reconfigurable Computing: A Survey of Systems and Software,” ACM Computing Surveys, Vol. 34, No. 2, June 2002, pp.171-210.

11 Cypher, R., “The SIMD Model of Parallel Computation”, Springer-Verlag, 1994, ISBN: 0-387-94139-8.

12 Mangione-Smith, W. H. − Hutchings, B., et al. "Seeking solutions in configurable computing", Computer Vol.30, No.12, December 1997, pp.38-43.

13 Wilkes, M.V., “The best way to design an automatic calculating machine”, Proceedings of Manchester University Computer Inaugural Conference, 1951, pp 16-18.

14 Agrawala, A.K. − Rauscher, T.G., “Foundations of Microprogramming: Architecture, Software and Applications”, Academic Press, New York, 1976, ISBN: 0120451506.

15 Gray, J.P. − Kean, T.A., “Configurable hardware: a new paradigm for computation”, Proceedings of decennial Caltech conference on VLSI on Advanced research in VLSI, June 1989, pp.279-295.

16 von Neumann, J., “ First draft of a Report on the EDVAC”, Moore School of Electrical Engineering, University of Pennsylvania, 1945.

17 Gajski, D.D. – Ramachandran, L., "Introduction to high-level synthesis", IEEE Design and Test of Computers, Vol. 11, No.4, 1994, pp.44-54.

18 Estrin, G. – Viswanathan, C.R., “Organisation of Computer systems − The Fixed Plus Variable Structure Computer”, Proceedings of the Western Joint Computer Conference, May 1960, pp.33-40.

19 Estrin, G. – Turn, R., “Automatic Assignment of Computation in a Variable Structure Computer System”, IEEE Transactions on Computers Vol. 12, No.6, December 1963, pp.747-755.

20 Miller, R. − Cocker, J., “Configurable computers: A new class of general-purpose machines”, Proceedings of the International Symposium on Theoretical Programming, Vol. 5, January 1972, pp.285-298.

21 “XC6200 Development System DataSheet”, Xilinx Inc., 1997.

22 Kuon, I. – Rose, J., “Measuring the gap between FPGAs and ASICs”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 26, No. 2, February 2007, pp.203-215.

23 “XC4000E and XC4000X Series Field Programmable Gate Arrays”, Datasheet DS005, Xilinx Inc., 2013.

24 “FLEX 10K Embedded Programmable Logic Family”, Datasheet DS-F10K-4.2, Altera Corporation., 2003.

25 “AT40K FPGAs with FreeRAM”, Datasheet 0896E, Atmel Inc., 2013.

26 Halfhill, T.R., “Tabula time machine – rapidly reconfigurable chips will challenge conventional FPGAs”, MicroProcessor Report, Issue 032910, 2010, www.MPRonline.com.

27 Hartenstein, R., “A decade of reconfigurable computing: a visionary retrospective”, Proceedings of the IEEE Conference on Design, Automation and Test in Europe DATE’01, March 2001, pp.642-649.

28 Todman, T. J., et al., “Reconfigurable computing: architectures and design methods”, IEE Proceedings on Computers and Digital Techniques, Vol. 152, No. 2, March 2005, pp.193-207.

29 Ling, X.P., et al., “WASMII: a data driven computer on a virtual hardware”, Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, April 1993, pp.33-42.

30 Shibata, Y., et al., **"**Towards the realistic "virtual hardware", International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems**,** October 1997, pp.50-55.

31 Shibata, Y., et al., "A virtual hardware system on a dynamically reconfigurable logic device", Proceedings of the 17th IEEE International Symposium on Field-Programmable Custom Computing Machines, April 2000, pp.295 -296.

32 Motomura, M., "A Dynamically Reconfigurable Processor Architecture", Microprocessor Forum, October 2002.

33 Brebner, G., "The swappable logic unit: a paradigm for virtual hardware", Proceedings of the 5th IEEE International Symposium on Field-Programmable Custom Computing Machines, April 1997, pp.77-86.

34 Brebner, G., "A Virtual Hardware Operating System for the Xilinx XC6200", Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers, September 1996, pp.327-336.

35 Brebner, G., "Automatic identification of swappable logic units in XC6200 circuitry", Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications, London, September 1997, pp.173-182.

36 Trimberger, S., et al., "A Time-Multiplexed FPGA", Proceedings of the 5th IEEE International Symposium on FPGA-Based Custom Computing Machines, IEEE Computer Society, April 1997, pp.22-28.

37 Tau, E., et al., "A First Generation DPGA Implementation", Third Canadian Workshop of Field-Programmable Devices, May 1995.

38 Scalera, S.M. − Vazquez, J.R., "The design and implementation of a context switching FPGA", Proceedings of the 6th IEEE International Symposium on Field-Programmable Custom Computing Machines, April 1998, pp.78-85.

39 Baumgarte, V., et al., “PACT XPP − A Self-reconfigurable Data Processing Architecture,” Journal of Supercomputing, Vol. 26, No2, September 2003, pp.167-184.

40 Furtek, F.C., et al., “Interconnecting heterogeneous nodes in an adaptive computing machine”, Proceedings of the 14th International Conference on Field-Programmable Logic and Applications, August 2004, pp.125-134.

41 Tang, X., et al., “A Compiler Directed Approach to Hiding Configuration Latency in Chameleon Processors”, Proceedings of the 10th International Conference on Field-Programmable Logic and Applications, August 2000, pp.29-38.

42 Cardoso, J.M.P., et al., “Compiling for reconfigurable computing: A survey”, ACM Computing Surveys, Vol.42, No.4, June 2010, pp.1-65.

43 Callahan, T.J., et al., “The GARP Architecture and C Compiler”, IEEE Computer, Vol.33, No.4, April 2000, pp. 62-69.

44 Bellows, P. – Hutchings, B., “JHDL- An HDL for Reconfigurable Systems”, Proceedings of the 6th IEEE International Symposium on Field-Programmable Custom Computing Machines, April 1998, pp.175-184.

45 Hogg, J., et al., “New HDL Research Challenges posed by Dynamically Reprogrammable Hardware”, Proceedings of the 3rd Asia Pacific Conference on Hardware Description Languages, January 1996.

46 Luk, W. – McKeever, S., “Pebble: A language for Parameterised and Reconfigurable Hardware Design”, Proceedings of the 8th International workshop on Field-Programmable Logic and Applications, August 1998, pp.9-18.

47 Singh, S. – Roxby, J., “Lava and JBits: From HDL to bitstream in seconds”, Proceedings of the 9th IEEE International Symposium on Field-Programmable Custom Computing Machines, April 2001, pp.91-100.

48 Guccione, S. et al., “JBits: Java-based interface for reconfigurable computing”, Proceedings of the 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference, Vol. 261, September 1999.

49 James-Roxby, P. − Guccione, S., “Automatic extraction of run-time parameterisable cores from programmable device configurations”, Proceedings of the 8th IEEE International Symposium on Field-Programmable Custom Computing Machines, April 2000, pp.153-161.

50 “Xilinx 9 Software Manuals”, Xilinx Inc., www.xilinx.com.

51 Guccione, S., − Levi, D., “Run-time Parameterizable Cores”, Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications August 1999, pp.215-222.

52 Johannes, F.M., “Partitioning of VLSI circuits and systems”, Proceedings of the 33rd annual Design Automation Conference, June 1996, pp.83-87.

53 Chang, D., − Marek-Sadowska, M., “Partitioning sequential circuits on dynamically reconfigurabe FPGAs”, IEEE Transactions on Computers, Vol. 48, No.6, June 1999, pp.565-578.

54 DeHon, A., “Reconfigurable Architectures for General Purpose Computing”, PhD Thesis, Massachusetts Institute of Technology, September 1996.

55 Schmitt, H., et al., “Behavioral Synthesis for FPGA-based Computing”, Proceedings of the 2nd International IEEE Workshop on Field-Programmable Custom Computing Machines**,** April 1994, pp.125-132.

56 Peterson, J., et al., “Scheduling and partitioning ANSI-C programs onto multi-FPGA CCM Architectures”, Proceedings of the 4th IEEE International Symposium on Field-Programmable Custom Computing Machines, April 1996, pp.178-179.

57 Vasilko, M. − Ait-Boudaoud, D., “Architectural synthesis techniques for dynamically reconfigurable logic”, Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers, September 1996, pp.290-296.

58 Mtibaa, A. − Ouni, B., “An efficient list scheduling algorithm for time placement problem”, Computers and Electrical Engineering, Vol.33, No. 4, July 2007, pp.285-298.

59 Bobda, C., “Synthesis of Dataflow Graphs for Reconfigurable Systems using Temporal Partitioning and Placement”, PhD Thesis, University of Paderborn, July 2003.

60 Wu, G.M., et al., “Generic ILP-based approaches for time-multiplexed FPGA partitioning”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 20, No. 10, October 2001, pp.1266-1274.

61 Ford, F. − Fulkerson, D., “Flows in Networks”, Princeton University Press, 1962, ISBN: 9780691146676.

62 Yang, H.H. − Wong, D., “Efficient Network Flow based Min-Cut Balanced Partitioning”, Proceedings of the IEEE/ACM Conference on Computer-Aided Design, November 1994, pp.50-55.

63 Lui, H. − Wong, D.F., “Network flow-based circuit partitioning for time-multiplexed FPGAs”, Proceedings of the IEEE/ACM Conference on Computer-Aided Design, November 1998, pp.497-504.

64 Paulin, P. − Knight, J.P., “Force-Directed Scheduling for the Behavioral Synthesis of ASIC’s.” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 8, No.6, June 1989, pp.661-679.

65 Pandely, A., Vemuri, R., “Combined Temporal Partitioning and Scheduling for Reconfigurable Architectures”, Proceedings SPIE Photonics East Conference, Reconfigurable Technology: FPGAs for Computing and Applications, August 1999, pp.93-103.

66 Cardoso, J., “On Combining Temporal Partitioning and sharing of Functional Units in compilation for Reconfigurable Architectures”, IEEE Transactions on Computers Vol. 52, No.10, October 2003, pp.1362-1375.

67 Weiguang, S., “Pareto Optimal temporal Partition Methodology for Reconfigurable Architectures Based on Multi-objective Genetic Algorithm”, Proceedings of the IEEE 26th International Symposium on Parallel and Distributed Processing, Workshops and PhD forum, May 2012, pp.425-430.

68 Premalatha, B. − Umamaheswari, S., “Survey Of Online Hardware Task Scheduling And Placement Algorithms For Partially Reconfigurable Computing Systems”, International Journal of Computing and Corporate Research, Vol.2, No.3, May 2012.

69 Lavin, C., et al., “RapidSmith: Do-It-Yourself CAD TOOLS for Xilinx FPGAs”, Proceedings of the 21st International Conference on Field-Programmable Logic and Applications, September 2011, pp.349-395.

70 Steiner, N., Wood, A., “Torc: Towards an Open-Source Tool Flow” Proceedings of the 19th ACM/SIGDA International symposium on Field Programmable Gate arrays, February 2011, pp. 41-44.

71 “Torc: tools for open reconfigurable systems”, www.torc.isi.edu, 2016.

72 Beckhoff, C., et al., “The Xilinx Design Language (XDL): tutorial and use cases,” Proceedings of the 6th International Workshop on Reconfigurable Communications-centric Systems-on-Chip, June 2011, pp.1-8.

73 Su, L., “The management of dynamically reconfigurable computing systems”, PhD Thesis, University of Manchester, 2008.

74 Hillis, D. − Steele, G., “Data Parallel Algorithms”, Communications of the ACM, Vol. 29, No 12, December 1986, pp.1170-1183.

75 Bobda, C. − Ahmadinia, A., “Dynamic interconnection of reconfigurable modules on reconfigurable devices”, IEEE Design & Test of Computers, Vol. 22, No.5, September 2005, pp.443-451.

76 Hermani, A., et al., “Network on chip: An architecture for billion transistor era”, Proceedings of the 18 th IEEE NorChip Conference, Vol.31, November 2000.

77 Boyan, A. − Littman, L., “Packet routing in dynamically changing networks: A reinforcement learning approach”, Proceedings of the Advances in Neural Information Systems 6 Conference, December 1993, pp.671-678.

78 McFarland, M.C., et al., "Tutorial on high-level synthesis", Proceedings of 25th ACM/IEEE Design Automation Conference, June 1988, pp.330-366.

79 Gajski, D.D. − Ramachandran, L., "Introduction to high-level synthesis", IEEE Design and Test of Computers, Vol. 11, No.4, 1994, pp.45-54.

80 Vahid, F., et al., “A transformation for integrating VHDL behavioral specification with synthesis and software generation”, Proceedings European Design Automation Conference, September 1994, pp.552-557.

81 Ebeling, C., et al., “Rapid: Reconfigurable Pipelined Datapath”, Proceedings of the 6th International Workshop on Field-Programmable Logic and Applications, September 1996, pp.126-135.

82 Synplify Pro and Premier”, datasheet, www.synopsys.com/Tools/Implementation/FPGAImplementation/CapsuleModule/synplify-pro-premier.pdf, Synopsys Inc.

83 Francis, R., Rose, J., “Chortle: a technology mapping program for lookup table-based field programmable gate arrays”, Proceedings of the 27th ACM/IEEE Design Automation Conference, June 1990, pp.613-619.

84 Aldec Inc., www.aldec.com/en/solutions/hardware_emulation_solutions, 2016

85 Bose, R.C. − Ray-Chaudhuri, D.K., “On a Class of error-correcting binary group codes”, Information and Control, Vol.3, No.3, March 1960, pp.68-79.

86 Martin, G. − Smith, G., “High-Level Synthesis: Past Present, and Future”, IEEE Design and Test of Computers, Vol. 26, No. 4, July 2009, pp.18-25.

87 Sarkar, S., et al., “Lessons and Experiences with High-Level Synthesis”, IEEE Design and Test of Computers, Vol. 26, No. 4, July 2009, pp.34-45.

88 Cong, J., et al., “High-Level synthesis for FPGAs: From Prototyping to Development”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 30, No.4, March 2011, pp.473-491.

89 Meeus, W., et al., “An overview of today’s high-level synthesis tools”, Design Automation for Embedded Systems, Vol. 16, No. 3, September 2012, pp.31-51.

90 Canis, A. et al., “LegUP: An Open Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems”, ACM Transactions on Embedded Computer Systems, Vol. 13, No.2, Article No. 24, September 2012, pp.1-27.

91 Berkeley Design Technology Inc., “BDTI High Level Synthesis Tool Certification Program”, 2010, www.bdti.com.

92 Aho, A.V., et al., “Compilers - Principles, Techniques and Tools”, Addison-Wesley Publishing Company, 1986, ISBN 0-201-10194-7.

93 Nijhar, T.P.K. − Brown, A. D., “Source level optimisation of VHDL for behavioural synthesis”, IEE Proceedings on Computers and Digital Techniques, Vol. 144, No. 1, January 1997, pp.1-6.

94 “ModelSim SE User Guide Version 6.2g", Mentor Graphics Corp., 2007.

95 Hoare, C.A.R., “Communicating Sequential Processes, Prentice-Hall Inc., 1985, ISBN: 0131532715.

96 “IEEE 1666 Standard SystemC Language Reference Manual, IEEE Std. 1666-2011”, IEEE Computer Society, January 2012, ISBN 978-0-7381-6801-2.

97 Handel-C Language Reference Manual, Agility Design Solutions, Inc., Palo Alto, CA, 2007.

98 OCCAM 2.1 Reference Manual SGS-Thomson Microelectronics Ltd., 1995.

99 Sanguinetti, J., et al. “Transaction-Accurate Interface Scheduling in High-level Synthesis”, Proceedings of the 2012 Electronic System Level Synthesis Conference, June 2012, pp.31-36.

100 “UG902-Vivado-High-Level-Synthesis”, Xilinx Inc., 2014.

101 Gokhale, M., et al., “Stream-oriented FPGA computing in the Streams-C high level language”, Proceedings of 8th International IEEE Symposium on Field-Programmable Custom Computing Machines, April 2000, pp.49-56.

102 Yee, T.B., et al., “Multi-FPGA Synthesis with Asynchronous Communication SubSystems”, Proceedings of the International Federation for Information Processing Conference on Very Large Scale Integration, October 2005.

103 LLVM 2010, The LLVM Compiler Infrastructure Project, www.llvm.org.

104 Coussy, P., et al., “GAUT – A Free and Open Source High-Level Synthesis Tool”, IEEE Design Automation and Test in Europe, University Booth, March 2010.

105 Pilato, C., et al., “Bambu: A Free framework for high-level synthesis of complex applications”, IEEE Design Automation and Test in Europe, University Booth, March 2012.

106 Huang, Q. – Ruolong, L., “The effect of Compiler Optimisation on high-level synthesis for FPGAs”, Proceedings of the 21st International IEEE Symposium on Field-Programmable Custom Computing Machines, April 2013, pp.89-96.

107 Hara, Y. – Tomiyama, H., “Proposal and qualitative analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis”, Journal of Information processing, Vol.17, October 2009, pp.242-254.

108 Gajski, D.D., et al., “Flow Graph Representation”, Proceedings of the 23rd ACM/IEEE Design Automation Conference, June 1986, pp.503-509.

109 Eles, P. – Kuchcinski, K., “Timing Constraint Specification and Synthesis in Behavioral VHDL”, Proceedings of the European Design Automation Conference with Euro-VHDL, September 1995, pp.452-457.

110 Peng, Z., “Synthesis of VLSI System with the CAMAD Design Aid”, Proceedings of the 23rd ACM/IEEE Design Automation Conference, June 1986, pp.278-28.

111 Brewer, F. – Gajski, D.D, “Chippe: A System for Constraint Driven Behavioral Synthesis”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 9, No. 7, July 1990, pp.681-695.

112 Parker, Alice C., et al., “MAHA: A Program for Datapath Synthesis”, Proceedings of the 23rd ACM/IEEE Design Automation Conference, June 1986, pp.461-466.

113 “CoCentric Fixed-Point Designer User Guide”, version 2002.05 edition, June 2002, Synopsys Inc.

114 "Behavioral Compiler User Guide", Synopsys Inc., 2000.

115 Camposano, R., et al., “Synthesizing Circuits From Behavioral Descriptions”, IEEE Transactions on Computer-Aided Design, Vol. 8, No. 2, February 1989, pp.171-180.

116 Hara, Y., et al. “Behavioral Partitioning with Exploiting Function-Level parallelism”, Proceedings of the International SoC Design Conference, Vol. 1, November 2008, pp.121-124.

117 Metropolis, N. − Rosenbluth, A. − Teller, A. − Teller, E., “Equation of State Calculations by Fast Computing Machines”, Journal of Chemical Physics, Vol. 21, No.6, June 1953, pp.1087-1092.

118 Chiricescu, S., et al., “Morphable Multipliers”, Proceedings of the 12th International Conference on Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream, September 2002, pp.647-656.

119 MacBeth, J. − Lysaght, P., “Dynamically Reconfigurable Cores”, Proceedings of the 11th International Conference on Field-Programmable Logic and Applications, August 2001, pp.462-472.

120 Vasilko, M., “Design Synthesis for Dynamically Reconfigurable Logic Systems”, PhD Thesis, Bournemouth University, October 2000.

121 Zhang, Xue-jie., et al., “A Combined Approach to High-Level Synthesis for Dynamically Reconfigurable Systems”, Proceedings of the 10th International Conference on Field-Programmable Logic and Applications, August 2000, pp.361-370.

122 “UG702 Xilinx Partial Reconfiguration User Guide”, Xilinx Inc., 2014

123 Green, S., “Development of a variable rate BCH coding system with CRC for multimedia transmissions in a wireless environment”, MEng Dissertation, University of Southampton, 2006.

124 Reeve, J.S. − Amarasinghe, K., “A FPGA implementation of a parallel Viterbi decoder for block cyclic and convolution codes”, IEEE International Conference on Communications, Vol.5, June 2004, pp.2596-2599.

125 Zwolinski, M. − Reeve, J.S., “Behavioural synthesis of an adaptive Viterbi decoder”, Proceedings of the 2nd IEE/EURASIP conference on DSP enabled radio”, September 2005.

126 Gupta, R.K. − De Micheli, G., “System-level Synthesis using re-programmable components”, Proceedings of the 3rd European Conference on Design Automation, March 1992, pp.2-7.

127 Haynes, S.D., “Video image processing with the sonic architecture”, IEEE Computer, Vol. 33, No.4, April 2000, pp.50-57.

128 Davis, W., et al., "Demystifying 3D ICs: the pros and cons of going vertical", IEEE Design & Test of Computers, Vol. 22, No. 6, November 2005, pp.498-510.

129 “UG909 Xilinx Vivado Partial Reconfiguration User Guide”, Xilinx Inc., 2014.

130 Li, Z., et al., “Configuration Caching management techniques for FPGA”, Proceedings of the 17th IEEE Symposium on Field-Programmable Custom Computing Machines, April 2000, pp.22-36.