Reconfigurable Hardware for Control Applications

Graeme Richard Milligan

A themed Portfolio submitted to the Universities of,

Glasgow,

Strathclyde,

Edinburgh

and Heriot-Watt

for the degree of Doctor of Engineering in

System Level Integration.

ABSTRACT

This portfolio document is intended to present the work carried out in order to meet the requirements of the Engineering Doctorate (EngD) program undertaken at the Institute for System Level Integration (ISLI). This program was undertaken in partnership with the Universities of Glasgow, Edinburgh, Strathclyde and Heriott Watt and was funded by EPSRC and SLI Ltd.

The use of control systems is becoming ubiquitous with even the simplest of systems now employing some kind of control logic. For this reason the project investigated the use and development of reconfigurable hardware for control applications. This first involved a detailed analysis of the current state of the art in the reconfigurable field as well as some selected applications where it is thought this technology may be of benefit.

The main body of the project was separated into three distinct areas of research and is hence presented as a collection of three technical documents. The first of these areas was the use of reconfigurable hardware for the implementation of Finite State Machines (FSM) with particular reference to reducing the size of the hardware block required to implement these structures. From this a novel implementation method was developed based on the principle of Forward Transition Expressions which are capable of implementing FSMs on a reconfigurable device using run-time reconfiguration. The second area of research was the investigation of the characteristics of reconfigurable devices with a view to estimating the amount of hardware required within a device from high level parameters. The final area of research was the development of a custom reconfigurable device specifically tailored for the implementation of FSM.

Declaration of Originality

I, Graeme Richard Milligan, declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given in the bibliography.

The material contained in this thesis is my own original work produced under the supervision of Wim Vanderbauwhede, Bob Adamson and Prof. Steve Beaumont. Some of the work contained in this thesis was produced in collaboration with Paul Jackson at Edinburgh University and as such, he is credited in the appropriate section.

Graeme Milligan

Dedication and Acknowledgements

This thesis is dedicated to the memory of

Mr. Thomas Jude O'Neill

22nd Dec. 1970 - 12th Dec. 2002

and

Miss. Catriona Caulfield

11th Jan. 1977 - 27th Aug. 2004

Your memory lives on with

all who knew you.

The Author would like to acknowledge the help and support of his parents, Catherine and David, without their infinite patience and support this work would not have been possible. I love you both very much.

I would also like to thank all of the staff at the Institute for System Level Integration for all their help and in particular Mrs. Alexandra (Sandy) Buchanan.

Finally i would like to thank all of my friends, in particular Jocelyn, Antoine and Ruairi, for preventing me from talking about my project too much and allowing me to escape from time to time. Thanks everyone.

Portfolio Document

Portfolio Executive Summary and Introduction

Author:
Graeme Milligan

Supervisor:
Wim Vanderbauwhede

Contents

1 Executive Summary

This volume is intended to present the work carried out in order to meet the requirements of the Engineering Doctorate (EngD) program undertaken at the Institute for System Level Integration (ISLI). This program was undertaken in partnership with the Universities of Glasgow, Edinburgh, Strathclyde and Heriott Watt and was funded by EPSRC and SLI Ltd. The EngD requires that 120 SCOTCAT credits are obtained in relevant engineering subjects with a further 60 credits required in business related subjects with the remainder of the program being taken up by an extended period of investigation and development in a given research domain.

Initially the scope of the project suggested was "The use of Reconfigurable Hardware in Communication Applications". This title was designed to give a large scope to the project and as such an extensive literature review was undertaken in both the communications and reconfigurable hardware domains. The reason for the large scope of the project was SLI Ltds relative inexperience in this field. For this reason this project was intended to operate at a fairly high level and allow SLI Ltd to develop an in-house understanding of reconfigurable hardware and also identify areas of interest for future investigation by subsequent EngD research engineers.

In order to address a lack of knowledge of the current state of reconfigurable hardware device development and to develop an understanding of the *state-of-the-art* in this field, an extended investigation was carried out into reconfigurable hardware. This investigation primarily focused on current novel reconfigurable architectures in both the academic and industrial domains with particular attention paid to devices specifically developed for the communications domain. During this period it was found that, for industrial devices in particular, the current devices under development were specifically targeted at the low level operations, such as encryption and encoding, carried out in the physical layers of communication protocol stacks. Due to the high speed and intensity of these operations these devices aimed to migrate any operations

carried out in software to take advantage of the characteristics of hardware to reduce the area and power required when caring out these operations while maintaining the speed required at these low levels. The report shown in appendix A1 was complied giving details of a selection of industrial and academic reconfigurable devices.

In order to identify possible areas in the communications domain that may benefit from the use of reconfigurable hardware it was first necessary to undertake an extended period of investigation into this field. Of particular interest is the possibility of creating a single reconfigurable device for mobile applications that is capable of implementing multiple communications protocols. In this way a single hardware device could be used to implement a range of communications protocols while making use of the low area, high speed and low power characteristics of digital hardware. For this reason it was decided that two example wireless protocols currently used in mobile applications should be analysed to identify areas that may be of interest for implementation using reconfigurable hardware. The protocols selected for analysis were the Universal Mobile Telecommunications System (UMTS) and the IEEE 8802.11 wireless LAN protocol. For each of these protocols a report was compiled and they are shown in appendix A2 and appendix A3 respectively. During this investigation it was seen that the upper layers (MAC, RLC etc.) of the communications stack can be modelled by a control path and a data path. The control path is responsible for determining the state of the device based on information provided by the network and the device. Based on the state of the control path, the data path performs the required data process on packets obtained from the layers immediately adjacent to the current layer in the communications protocol stack.

At this point the control path was identified as of particular interest due to the generality of the operations carried out. It can be seen that the control path can be implemented as a Finite State Machine (FSM) where the state of the machine is determined by information provided by the device and from the network currently in use by the device. As FSMs are used in almost all control applications, the use of a reconfigurable hardware device specifically tailored for the implementation of these control systems has a market that is not limited to the implementation of communications protocols. For this reason the scope of the project was altered to narrow the domain of interest to the implementation of control systems and specifically the implementation of FSMs using reconfigurable hardware. This lead to the title of the project begin altered to “Reconfigurable Hardware for Control Applications”

As the project was intended to be used to build a general understanding within SLI Ltd. of the reconfigurable hardware domain and to suggest possible areas of interest for future EngD research engineers it was felt that rather than undertaking a single monolithic project on the use of Reconfigurable hardware in the control domain the remainder of the extended period of research should be split between a number of related sub-projects under the title of “Reconfigurable Hardware for Control applications” with particular focus on the development of a device tailored to the implementation of control applications. The following section gives an overview of the business aspects of the project and aims to provide enough information to allow the reader to assess the desicions made and advantages of the ideas proposed in the thesis from a business perspective. This is then followed by an overview of each of the sub-projects with the goals of each project being identified and the objectives achieved.

2 Business Aspects

Although much of the material presented here is mainly of a theoretical nature it is essential to be able to assess any benifits produced by the ideas presented from an industrial perspective. For this reason this section will outline the main design parameters, area, speed and power, and relate to the control/communications domain. This will allow comment to be made on the impact of the ideas presented here on these parameters and how they can be related to the overall cost/benefit of utilising these ideas in a modern design flow.

Due to the large range of implementations that control systems find themselves their requirements vary greatly, a control system for use in the avionics systems must be far more reliable and faster acting than one found in a washing machine for example. The following section presents each of the key design metrics used in modern silicon chip design.

2.1 Area

The area of the silion chip produced during manufacture is probably the most important design metric concerning the operation and cost of a new device. The area of the chip determines how many chips can be produced in a single production batch at a fixed cost. This means the smaller the area of the chip the larger the number that can be produced and hence the cost per chip is reduced. The size of the chip also has an impact on what is known as production yield. This is the percentage of all chips produced that operate correctly. It is found that the smaller the silicon area the larger the production yield as any defects may affect a larger number of chips but due to the small size this equates to a smaller percentage of the total amount produced.

The area of the silicon required is determined by the amount of hardware required to implement the required functionality, the wiring associated to connect these components (transistors) as required and the production method selected. For this reason much of the design time is concerned with the removal of redundant logic to ensure the chip is compact as possible and place and route to reduce the length and amount of logical interconnect required.

It is also important to ensure that before manufacture the chip has been proven to operate as required as it is very costly to correct mistakes after manufacture has begun and usually invovles returning to the design stage and restarting manufacture.

The area of the chip also has a direct effect on the other main design metrics as is outlined in the following sections.

2.2 Speed

The speed of operation of a silicon chip is an essential design paramter as it is essential to ensure that the chip operates fast enough to ensure correct operation. For example, a system design to control the avionics on a modern jet fighter must operate at far greater speeds than a system design as a user interface. In the later example the device is operating at “human speeds” and can thus can operate at relatively low speeds.

The speed of a silicon device is dependent on 2 main factors; the first of these are the delays introduced by the actual components or transistors on the chip and the second are delays introduced by capacitance in the interconnects between these components. The actual speed of opeartion is highly dependent on what is known as the critical path, this is the longest path from the inputs of the device to the outputs of the device. Due to the importnace of this factor much research has been carried out on modelling the delays in the critical path and optimisation to reduce the effect of the delays introduced.

Another aspect associated with the speed of a device is latency. This is mainly used when refering to synchrounous logic where a clock signal is required to allow components to pass data from their inputs to the outputs. This means that for every synchrounous component a signal must pass through a clock cycle is required. This should not be confused with the speed of the device which is the actual speed of the clock rate. It can be seen that the size of the chip will have a large effect on the speed of the device as smaller chips will have shorter interconnects and hence a shorter critical path.

2.3 Power

The power consummed by a silicon device comes from 2 main sources; static and dynamic power consumption. Static power refers to all sources of power consumption present when the chip is in a steady state and is usually assocaited with leakage current from the transistors through the substarate of the device. Dynamic power consumption concerns all power that is used in a silicon device during operation. The main source of dynamic power consumption is the power required to drive the trnasistors on the device and the capacatance in the interconnects. Dynamic power is far more significant than static power consumption and as such much research has been produced aimed at reducing the dynamic power consumption of devices.

It can be seen that one of the most important factors in dynamic power is the size, or area, of the device as this dictates the number of transistors that must be charged during operation and also the capacitance of the interconnects. For this reason it is normal to assume that the smaller the silicon device the lower the power consumption but it should be noted that this is only true where the operating speed remains constant as reducing the size will be offset by the increase in frequency that the dynamic conponents must be charged.

2.4 Control Domain

Due to the varied nature of the control domain, the requirements of differing application will vary greatly. As stated, control systems are found in all application domains from space and military applications to simple consumer electronics devices. In general it can be seen that the selection of the method of implementation is highly dependent on the application and is in effect a balancing act between producing a technically acceptable solution and non-enginneering factors such as design time and cost. For this reason, although it would be desirable to produce the smallest, fastest and most power efficient implementation of the required control system, this is often imractical due to the high cost of designing such a device.

If the control domain is concidered from the mobile communications device perspective, as was the oriignal scope of this project, it is desirable to utilise very small device as this allows for the reduction in size of the communications device. It is also essential to ensure that the device be as power efficient as possible to increase the battery life of the device. Depending on the nature of the control system within the device, speed varies in importance as a design metric as user interface control systems are intended to work at “human” speeds whereas the undelying control systems for the communications protocols must operate at very high speeds and hence require a high speed device.

3 Project overview

As stated, the main period of extended research was divided into a collection of inter-related sub-projects under the umbrella of the overall project title. This allowed a number of aspects of the use of reconfigurable hardware for control applications to be researched with the view of developing an understanding of the possible uses and also suggest future projects that could be undertaken by SLI Ltd. For this reason two main areas were identified; the use of reconfigurable hardware in the implementation of Finite State Machines (FSMs) and the investigation of a novel device specifically tailored for this application. In order to achieve this the first sub-project was primarily concerned with identifying the optimal use of reconfigurable hardware in the implementation of the control structures. The remaining two sub-projects undertaken were then concerned with investigating the characteristics of the device required to implement these structures based on the use of reconfigurable hardware suggested in the previous project.

The following presents a short overview of each of the sub-projects.

3.1 Implementation of Control Systems using Reconfigurable Hardware

This project was primarily concerned with the investigation of the optimal use of reconfigurable hardware when used to implement control systems and in particular FSMs. Although it is possible to simply map FSMs directly to reconfigurable devices it was considered that this does not make use of the particular characteristics, and in particular the ability of reconfigurable hardware to be programmed during device operation *(run-time reconfiguration)*. As such the traditional method of implementation of FSM was investigated in great detail with a view to altering this to take advantage of the characteristics of reconfigurable hardware.

During this investigation a general model of FSM was developed and based on this a novel method of FSM representation was suggested. This representation was based on using the current state of the reconfigurable device to represent the current state of the FSM rather than using a feedback register as would traditionally be the case. This representation allowed a far smaller block of logic to be used when determining state changes.

In order to investigate the novel FSM representation presented in this sub-project a C based circuit generator was developed that is capable of implementing the hardware required to implement an FSM using the representation presented. This generator is capable of generating all of the components required to produce the device required to implement the FSM and has demonstrated in simulation the correct operation of this novel device and the novel FSM representation presented.

The work carried out during this sub-project was considered novel enough to be considered for patenting by SLI Ltd. Although extensive consultation was undertaken with patent lawyers, due to the length of time required to draft the patent application, at the time of writing this work, this patent has yet to be filed. Although the commercial advantage of patenting this work, and indeed the general use of patents in this field, is questionable it is felt that the purpose of this process was to improve the profile of SLI Ltd. Due to the delays incurred by this application process and the patent not being filed by the end of the project a paper was instead submitted to the NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2007) where it has since been accepted for publication.

3.2 Investigation of the Characteristics of Reconfigurable Devices

This sub-project was primarily concerned with the investigation of the hardware required within reconfigurable devices for the implementation of logic like that used in the calculation of the next state of FSMs. The next state of FSMs is calculated using a Combinatorial Logic Block (CLB) that calculates the next state of the FSM based on a set Combinatorial Logic Expressions (CLE). For this reason the hardware required to implement CLEs was investigated with the goal of determining the optimal amount of hardware that should be placed within the reconfigurable device fabric in order to allow for the implementation of these expressions.

Initially the *Sum-of-Products* representation and Boolean simplification of CLEs was investigated in order to determine if a link existed between the number of inputs to an expression and the hardware required to implement these expressions. By generating every possible expression with a fixed number of inputs it was found that the number of inputs to the expression had a direct effect on the hardware required to implement these expressions but due to the extremely large number of possible expressions this form of exhaustive testing was impractical for expressions with even very small numbers of inputs. In order to extend this investigation a pseudo-random circuit generator was developed that was capable of generating very large numbers of expressions with a fixed number of inputs. The hardware required to implement these expressions could then be found and based on this the distribution of hardware requirements could be calculated. The results obtained using this method were then compared to those obtained in a previous project, carried out to determine the average hardware requirements of logic based devices, to confirm the accuracy of the circuit generation method. This method was then extended to memory based architectures where no comparative research has been identified.

Although this method allows the hardware requirements of CLEs to be investigated it was found that even for the pseudo-random circuit generation method that the time taken to obtain results for CLEs with large numbers of inputs was prohibitive. For this reason a mathematical method of determining the average hardware requirements of CLEs was developed in conjunction with Dr. Paul Jackson at Edinburgh University. This method is based on cube cover and allows the calculation of the average hardware requirements of logic based reconfigurable devices based on the number of inputs to the expression.

Based on the work carried out in this sub-project a conference paper has been compiled and has been submitted to XXV IEEE Intenational Conference on Computer Design 2007 (ICCD-2007) where it currently in the process of peer review prior to publication.

3.3 Development of a Hardwired Directional Reconfigurable Architecture

This sub-project focused on the development of a novel reconfigurable device for the implementation of CLEs like those used in the next state calculation of FSMs. In order to develop this device it was necessary to develop a sound understanding of the method by which these expressions are traditionally implemented in existing reconfigurable devices. Based on this a device architecture is suggested that makes use of the characteristics of these expressions to reduce the overhead introduced by reconfigurable hardware in providing the flexibility required by these devices.

A device generator was then developed to allow for the investigation of this architecture. The generator is capable of generating all of the hardware components required to implement this architecture and produce a fully functioning device. Due to the nature of the device a novel synthesis tool was also developed to allow a high level description of the CLE to be synthesised to produce the bit-file required to implement the CLE on this novel reconfigurable device. The synthesis tool made use of existing synthesis algorithms to obtain a representation of the CLE suitable for implementation and automatically performs the place and route operation required to fully implement the CLE on the architecture suggested. The hardware characteristics of the device required to implement these CLEs could then be compared to those required using standard reconfigurable devices such as FPGAs.

Again the work presented in this sub-project was considered novel enough to be considered for patenting by SLI Ltd but due to time constraints this patent has yet to be filed.

4 Portfolio Organisation

As the extended period of research was subdivided into three sub-projects a separate and distinct portfolio document has been produced to present each of these sub-projects. The structure of this document is shown in figure 1. As this shows each of the sub-project has its own distinct report entitled Technical report (TR) 1-3. The literature reviews carried out on communications protocols and reconfigurable hardware are presented in the appendix with titles appendix (A) 1-3. Finally the papers produced concerning TR1 and 2 are presented at the end of this document and are named Conference Paper (CP) 1 and CP2 respectively.

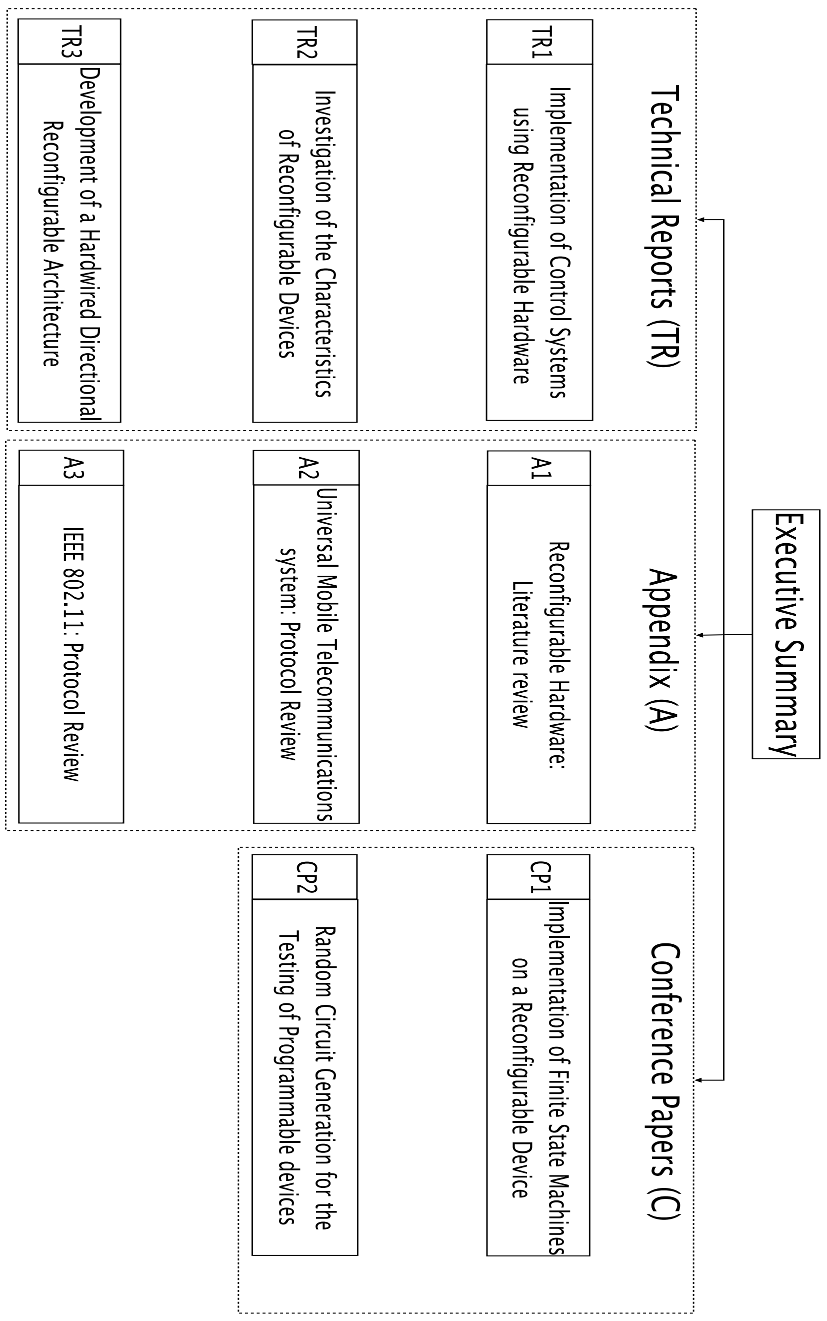

Executive Summary
Technical Reports (TR)
TR1
Implementation of Control Systems using Reconfigurable Hardware
TR2
Investigation of the Characteristics of Reconfigurable Devices
TR3
Development of a Hardwired Directional Reconfigurable Architecture
Appendix (A)
A1
Reconfigurable Hardware: Literature review
A2
Universal Mobile Telecommunications system: Protocol Review
A3
IEEE 802.11: Protocol Review
Conference Papers (C)
CP1
Implementation of Finite State Machines on a Reconfigurable Device
CP2
Random Circuit Generation for the Testing of Programmable devices

Figure 1: Portfolio Organisation Chart

Portfolio Document TR1

Implementation of Control Systems using Reconfigurable Hardware

Author:

Graeme Milligan

Supervisor:

Wim Vanderbauwhede

Contents

List of Figures

List of Algorithms

List of Tables

1 Introduction

In modern engineering the use of control systems to implement complex system behaviour is ubiquitous and found in almost all engineering disciplines. Whether they are inferred or purposely designed into devices, control systems are required to allow devices to operate in an intelligent manner rather than simply reacting to the current operating conditions of the system. The use of control systems allows a device to determine its response to a sequence of events rather than simply producing a response based on a single event. Determining the response of devices in this way allows more complex behaviour to be implemented and allows a greater degree of automation, reducing the need for human intervention.

Although for simple applications it is often unnecessary to explicitly specify the control system, and it is instead inferred from the functional description of the application, for more complex applications a high-level model is required to allow designers to specify the control behaviour of the system. A commonly used high-level model is the Finite State Machine (FSM). FSMs are abstract models used to describe the behaviour of a control system in response to a sequence of inputs and are based on the theory of finite automata in computer science [9].

Due to the ubiquitous nature of FSMs a device capable of efficiently implementing these models would have applications in almost all domains. As the characteristics of FSMs are highly dependent on the particular application, it is dangerous to create a device for the implementation of general FSMs using domain profiling techniques such as that suggested in [7]. In this method benchmark circuits from a particular domain are used to determine the characteristics of reconfigurable device targeted at particular application domains such as communications or digital signal processing (DSP). Instead a more generalized approach is required to determine the most efficient use of reconfigurable hardware for the implementation of FSMs. The use of FSMs to implement the control behaviour of systems is well established and as such a large number of texts are devoted to this subject including [6, 11, 12].

The implementation of FSMs can be broadly separated into two sections, the extraction of the FSM from the high level specification of the system and the implementation of the FSM. The extraction of FSMs is independent of the method of implementation and is concerned with developing a high-level description of the control aspects of the specification.

The implementation of the FSM can be split into two main groups, the FSM can either be implemented in software or hardware. Software implementation of FSMs allows rapid development and deployment due to short software design cycles. where as implementation in dedicated hardware involves the production of an Application Specific Integrated Circuit (ASIC) that implements the required behaviour. This approach produces devices with low area and power requirements but requires lengthy and expensive hardware design and manufacturing processes making it impractical for low volume applications or for developers with limited resources. An alternative to the use of full custom hardware is the use of reconfigurable devices such as the Field Programmable Gate Array (FPGA).

Reconfigurable hardware combines the speed and efficiency of hardware with the flexibility and programmability of software. The introduction of this flexibility and programmability requires the introduction of hardware required to program the device resulting in a lower efficiency when compared to full-custom ASIC design.

Recently there has been much interest in making use of the ability of reconfigurable devices to be programmed while the device is in operation. This allows devices to implement multiple functions simultaneously by dynamically switching between functions or 'contexts'. This process, known as *run-time reconfiguration*, allows a small reconfigurable device to masquerade as a larger device by implementing applications that require larger hardware resources than are available on a single reconfigurable device.

This section of the portfolio is intended to give details of the work carried out on the development of a novel method for the implementation of FSMs using reconfigurable hardware. This method makes use of a novel representation of FSMs that allows the implementation of these machines using a far smaller reconfigurable device than would be traditionally required. This section begins by presenting the formal definition of FSMs in section 2.1 and is followed by an overview of the traditional method of FSM implementation in full-custom hardware and on a reconfigurable device to allow for comparison with the novel method presented here.

The novel representation of FSMs suggested is then presented in section 3 before the design flow required to implement this representation is presented in section 4. Section 5 then presents the device required to implement the novel FSM representation presented. In order to prove the usefulness of this method it is then compared to the traditional method of implementation for a range of FSMs and the results of this comparison are presented in section 6 before final conclusions are made.

2 Background Theory

In order to investigate the most suitable use of reconfigurable hardware for the implementation of control systems it is first necessary to understand both the theory and implementation of FSMs. This section begins by presenting a formal definition of FSMs and is followed by a description of the process of FSM extraction before the implementation of FSMs is discussed for both the ASIC and reconfigurable device design flows.

2.1 Formal Definition of Finite State Machines

FSMs are abstract models used to represent the sequential behaviour of systems. They are used in control applications to define the response of a system to a sequence of input events. This allows designers to implement systems with complex behaviour as opposed to simple systems that can react to only the current operating conditions of the system.

An FSM can be fully described using the 6-tuple

$$M = (S, I, O,\Delta,\Lambda,R), \text{ where}$$

- S is a finite set of states, and $|S|$ is the total number of unique states,
- I is a finite input space, and $|I|$is the total number of inputs,
- O is a finite output space, and $|O|$ is the total number of outputs,
- $\Delta$is a set state transitions based on the current state and the current inputs,
- $\Lambda$is the output relation defined in terms of the current state and the current input vector,
- and R is a set of reset states.

State transitions are assigned to each state and based on the current inputs and current state are used to determine the next state of the FSM. The current state of the FSM is determined by the initial state of the FSM and the previous input sequence applied to the machine. This means if the previous input sequence and start state of the FSM is known it is possible to predict the current state of the FSM.

The output of the state machine (O) can be generated in two ways: either the output is dependent only on the current state (Moore machine) or is dependent on the current state and the current inputs (Mealy machine). In the case of the Moore machine the output relation Λ will contain only a single output for each state. This section is only intended to present the formal description and overview of FSMs, a more detailed discussion of FSMs can be found in many texts including [6, 11, 12].

It should be noted that as the novel implementation method suggested here is only concerned with the generation of the next and current state of the FSM it is applicable to both Moore and Mealy machines. In the case of the Mealy machine an addition block of logic would still be required to produce the final outputs.

2.2 Traditional Implementation of Finite State Machines

The implementation of FSMs can be broadly separated into two sections, the extraction of the FSM from the specification of the system and the implementation of the FSM using the appropriate technology. The FSM design flow for both full-custom ASIC and reconfigurable hardware devices is shown in figure 1. As this illustrates the extraction of the FSM is independent of the method of implementation and is hence common to both design flows.

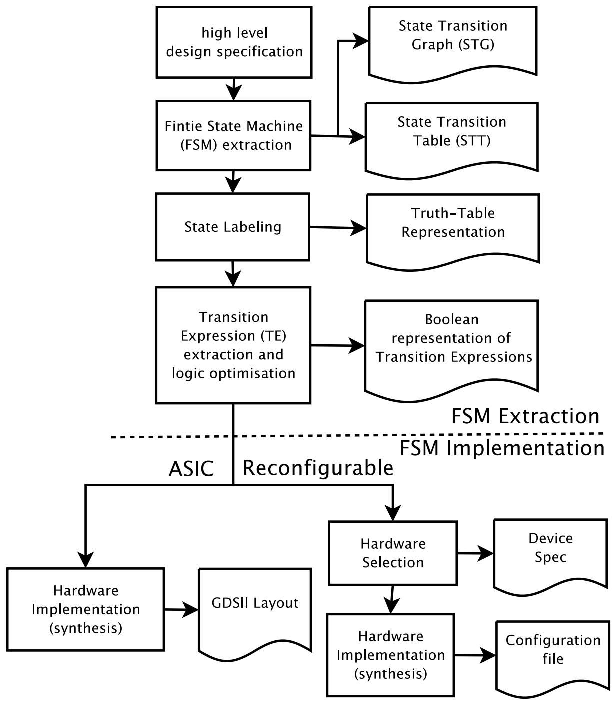

high level design specification
State Transition Graph (STG)
Fintie State Machine (FSM) extraction
State Transition Table (STT)
State Labeling
Truth-Table Representation
Transition Expression (TE) extraction and logic optimisation
Boolean representation of Transition Expressions
FSM Extraction
FSM Implementation
ASIC
Reconfigurable
Hardware Selection
Device Spec
Hardware Implementation (synthesis)
GDSII Layout
Hardware Implementation (synthesis)
Configuration file

Figure 1: Simplified FSM design flow for ASIC and Reconfigurable device

2.2.1 Finite State Machine extraction

The conventional method of FSM extraction is shown in figure 2. As this shows a high level description of the control requirements of the proposed device are first extract from the initial specification of the system. This behaviour can then be expressed as an FSM using a high level model such as a State Transition Graph (STG) or State Transition Table (STT). Based on this model it is then possible to produce a description of the FSM suitable for the implementation of the control requirements of the system.

The following sections give details of each of the main stages in extraction of FSMs. For the purposes of clarity the Mead-Conway traffic light controller [11] will be used to provide a detailed worked example.

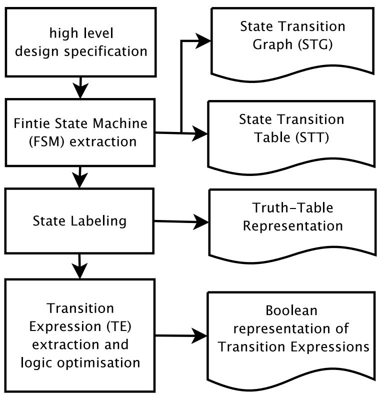

high level design specification
State Transition Graph (STG)
Fintie State Machine (FSM) extraction
State Transition Table (STT)
State Labeling
Truth-Table Representation
Transition Expression (TE) extraction and logic optimisation
Boolean representation of Transition Expressions

Figure 2: FSM extraction

2.2.2 State Transition Graph representation of Finite State Machines

The State Transition Graph (STG) is a graphical representation of FSMs. The STG uses nodes to represent the states of the FSM and edges to represent state transitions. The output behaviour of the FSM is associated with the states as required and the edges are labeled with the input conditions necessary to cause transitions.

A sample STG of the classic Mead-Conway traffic light controller is shown in figure 3. This is a simple 4-state (HG, HY, FG, FY) FSM, where $|S| = 4$, with an input alphabet of C, T_1, T_s, $|I| = 3$. The STG shows each of the states and the state transitions, with each of the transitions labeled with the necessary input conditions to cause the transitions.

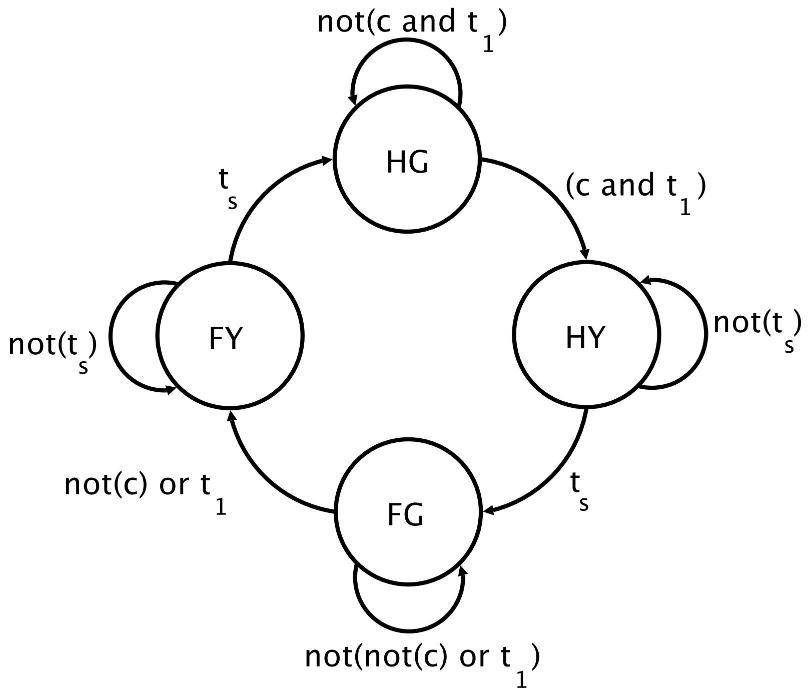


Figure 3: STG for Mead-Conway Traffic Light Controller [11]

The Mead-Conway traffic light controller represents the control system required to control the traffic lights at the crossroads between a field road and a highway. The system comprises a light on the highway (hl) and a light on the field (fl) with a crossing button (c) that is pressed to request the lights to change.

The system would usually be in the HG state where the highway lights are set to green and the field road lights are set to red. On entering this state a timer (t_1) is started. After this timer has expired and if the cross button (c) is pressed a state change is triggered to state HY.

In state HY a timer (t_s) is started and the highway light is set to yellow while the field light remains red. After a set time has passed another state transition is triggered to state FG.

In state FG the field road light is set to green and the highway light is set to red, a timer (t_1) is again set. If the button is released or the timer has finished a state change is triggered to state FY.

In state FY the field road light is set to yellow and the highway light remains red. A timer (t_s) is again set and the FSM returns to its start state on the expiry of this timer.

2.2.3 State Transition Table representation of Finite State Machines

An alternative to the STG is the State Transition Table (STT). The STT of the Mead-Conway traffic light controller is shown in table 1. The table shows each of the required state transitions of the FSM and gives the current state and input conditions required to cause these transitions. In this simple example there are only two possible transitions from each state and the conditions that cause these transitions are listed. The outputs produced by the FSM are listed in the table, it should be noted that it is customary to produce a hardware block responsible for implementing the FSM and a separate block that produces the outputs based on the current state of the FSM.

current state	Inputs	Next State	Outputs
HG	not(c and t_1)	HG	hl=GREEN; fl=RED
HG	c and t_1	HY	hl=GREEN; fl=RED
HY	not(t_S)	HY	hl=YELLOW; fl=RED
HY	t_S	FG	hl=YELLOW; fl=RED
FG	not(not(c) or t_1)	FG	hl=RED; fl=GREEN
FG	not(c) or t_1	FY	hl=RED; fl=GREEN
FY	not(t_S)	FY	hl=RED; fl=YELLOW
FY	t_S	HG	hl=RED; fl=YELLOW

Table 1: STT of Mead-Conway Traffic Light Controller

2.2.4 State Labeling

To allow digital logic to implement the FSM it is necessary to perform state encoding. State encoding takes the symbolic representation, i.e. label, of the states and replaces it with a Boolean representation that can be produced using digital logic. Although a number of encoding methods exist, the most commonly used technique is binary encoding.

In binary encoding each state is given a unique binary code and this is associated with the state label as shown in table 2. Using binary encoding a hardware block with only $log_2(|S|)$ outputs is capable of indicating the next state of an FSM with $|S|$ states. This hardware block is responsible for calculating the next state of the FSM and producing the corresponding binary code at its outputs.

	Encoding	
State Label	Z_0	Z_1
HG	0	0
HY	0	1
FY	1	0
FG	1	1

Table 2: Binary Encoding of State Labels

By replacing the labels in the STT with their binary equivalents, as given in table 2, it is possible to build a truth table representation of the FSM. Table 3 gives the truth table for the Mead-Conway traffic light controller. From this it can be seen that the STT presented in section 2.2.3 has been expanded to give all of the possible input conditions for each current state ($Z_o(t)$, $Z_1(t)$) and the next state ($Z_o(t+1)$, $Z_1(t+1)$) for each of these conditions has been specified.

Using a vector notation

$$\mathbf{Z} = (Z_0, Z_1),\ \ \mathbf{HG} = (0,0), etc.$$

and using the shorthand $(\mathbf{X} = \mathbf{Y})$for $\sum_i X_i \oplus Y$, the generator expression for this table can be written as:

$$
\begin{aligned}
\mathbf{Z}(t+1) \quad & \triangleq \\
& \mathbf{Z}(t) = \mathbf{HG}.(\overline{c.t_1}.\mathbf{HG} + c.t_1.\mathbf{HY}) + \\
& \mathbf{Z}(t) = \mathbf{HY}.(\overline{t_s}.\mathbf{HY} + t_s.\mathbf{FG}) + \\
& \mathbf{Z}(t) = \mathbf{FG}.(\overline{\overline{c} + t_1}.\mathbf{FG} + (\overline{c} + t_1).\mathbf{FY}) + \\
& \mathbf{Z}(t) = \mathbf{FY}.(\overline{t_s}.\mathbf{FY} + t_s.\mathbf{HG})
\end{aligned}
$$

2.2.5 Transition Expression (TE) Extraction and Optimization

From the truth-table representation it is possible to extract the logic expressions required to implement the behaviour of the FSM. The extracted expressions are termed Transition Expressions (TE) as they calculate the required state transitions based on the primary inputs and the current state.

If the transition expression is expressed in sum-of-products format, each 1 in the output columns of the truth-table results in the addition of a product term to the TE. For the example shown in table 3 the logic expression required to calculate the next state $Z_0(t+1)$is

$$
\begin{aligned}
\mathbf{Z_0}(t+1) \quad = \quad & \mathbf{Z_0}(t).\overline{\mathbf{Z_1}(t).c}.t_1.\overline{t_s} + \mathbf{Z_0}(t).\mathbf{Z_1}(t).\overline{c}.t_1\overline{t_s} \\
+ \quad & \mathbf{Z_0}(t).\overline{\mathbf{Z_1}(t).c.t_1.t_s} + \mathbf{Z_0}(t).\mathbf{Z_1}(t)\overline{.c.t_1.t_s} \\
+ \quad & \mathbf{Z_0}(t).\overline{\mathbf{Z_1}(t)}.c\overline{.t_1.t_s} + \mathbf{Z_0}(t).\mathbf{Z_1}(t).c\overline{.t_1t_s} \\
+ \quad & \mathbf{Z_0}(t).\overline{\mathbf{Z_1}(t)}.c.t_1.\overline{t_s} + \mathbf{Z_0}(t).\mathbf{Z_1}(t).c.t_1\overline{t_s} \\
+ \quad & \overline{\mathbf{Z_0}(t)}.\mathbf{Z_1}(t)\overline{.c.t_1}.t_s + \mathbf{Z_0}(t).\mathbf{Z_1}(t)\overline{.c.t_1}.t_s \\
+ \quad & \overline{\mathbf{Z_0}(t)}.\mathbf{Z_1}(t).c.\overline{t_1}.t_s + \mathbf{Z_0}(t).\mathbf{Z_1}(t).c.\overline{t_1}.t_s \\
+ \quad & \overline{\mathbf{Z_0}(t)}.\mathbf{Z_1}(t).\overline{c}.t_1.t_s + \mathbf{Z_0}(t).\mathbf{Z_1}(t).\overline{c}.t_1.t_s \\
+ \quad & \overline{\mathbf{Z_0}(t)}.\mathbf{Z_1}(t).c.t_1.t_s + \mathbf{Z_0}(t).\mathbf{Z_1}(t).c.t_1.t_s
\end{aligned}
$$

current state		Primary Input			Next State	
$Z_0(t)$	$Z_1(t)$	c	t_1	t_s	$Z_0(t+1)$	$Z_1(t+1)$
0	0	0	0	0	0	0
0	0	0	0	1	0	0
0	0	0	1	0	0	0
0	0	0	1	1	0	0
0	0	1	0	0	0	0
0	0	1	0	1	0	0
0	0	1	1	0	0	1
0	0	1	1	1	0	1
0	1	0	0	0	0	1
0	1	0	0	1	1	1
0	1	0	1	0	0	1
0	1	0	1	1	1	1
0	1	1	0	0	0	1
0	1	1	0	1	1	1
0	1	1	1	0	0	1
0	1	1	1	1	1	1
1	0	0	0	0	1	0
1	0	0	0	1	0	0
1	0	0	1	0	1	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	0	1	0	0
1	0	1	1	0	1	0
1	0	1	1	1	0	0
1	1	0	0	0	1	0
1	1	0	0	1	1	0
1	1	0	1	0	1	0
1	1	0	1	1	1	0
1	1	1	0	0	1	1
1	1	1	0	1	1	1
1	1	1	1	0	1	0
1	1	1	1	1	1	0

Table 3: Truth-Table representation of the Mead-Conway traffic light controller

This expression can be optimized using boolean simplification techniques to remove redundant logic. This simplification process is well established and can be performed by tools such as SIS [5]. Performing boolean simplification on the above transition expression results in

$$\mathbf{Z_0}(t+1) \quad = \quad \mathbf{Z_0}(t).\overline{t_s} \; + \mathbf{Z_1}(t).t_s$$

As this shows, the removal of redundant product terms, using boolean minimization techniques vastly reduces the size of the logic block required to implement the transition expressions and hence makes this a vital step in the extraction of the FSM.

2.2.6 ASIC implementation of FSMs

This process is commonly known as synthesis as it involves taking a high level description and converting this to a description that can be used to produce an ASIC capable of implementing the required functionality.

The implementation of a FSM in full-custom hardware involves first producing a description of the device in a hardware description language such as Verilog [10]. This is a hardware description language (HDL) that describes the functionality of the required device in a high level programming language that is capable of being used to produce a silicon implementation of the device.

Due to the mature nature of silicon implementation, many of the steps involved are hidden from the device designer and automatically carried out by design tools such as [1]. The tools can be used to perform synthesis of HDL to a format such as GDSII [2] that can be used by manufacturers to produce a silicon implementation.

Synthesis takes the HDL description of the FSM and converts this to a netlist format that expresses the hardware required to implement the HDL as well as the interconnects between these components. Extensive testing is then required to ensure that the device operates correctly prior to manufacture as any errors would be costly to correct if not detected early in the design cycle. The netlist is then further synthesised to produce a layout in GDSII format that is again tested and refined to ensure correct device operation before manufacture.

This project is not concerned with the details of the synthesis of HDL or the silicon implementation of devices and as such this section is not intended to detail this process, further information can be found in a number of texts including [8].

2.2.7 FSM implementation on a Reconfigurable Device

The conventional method of implementing an FSM on a reconfigurable device is shown in figure 4. The process outlined here is similar to the design-flow used in full-custom ASIC design of FSMs but, where as in full-custom design flows synthesis results in a description that can be used to produce a silicon implementation of the FSM, synthesis for reconfigurable devices results in a bit-stream capable of configuring the device to implement the FSM.

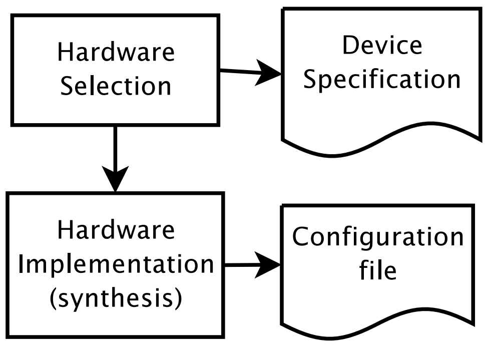


Figure 4: FSM implementation on a reconfigurable device

2.2.7.1 Hardware selection

The selection of a suitable reconfigurable device is vital to the efficient implementation of the FSM. It is essential to ensure that the device has sufficient hardware resources to implement the FSM but does not introduce excessive redundant hardware. As the reconfigurable device is a generic part it is unlikely that the device will have exactly the correct amount of hardware required to implement a particular application. However, devices are available from device manufacturers in a wide range of sizes and costs, allowing the end-user to select the device most suitable for the chosen application.

Based on the high level description of the FSM it is possible to extract an estimate of the hardware requirements of the device required to implement the FSM. The end-user would then select the device with as close to these parameters as possible to ensure excessive hardware is not introduced that would impact on the cost, area and possibly power of the final implementation.

2.2.7.2 HDL Synthesis for reconfigurable devices

After the selection of the appropriate reconfigurable device the high level description of the FSM is then synthesised. Whereas synthesis to ASIC produces a silicon implementation of the FSM, synthesis to a reconfigurable device takes the high level description and produces the bit-stream required to program the device to implement the FSM. As each reconfigurable device has an individual structure and hardware characteristics, synthesis is usually performed by proprietary tools provided by the device manufacture. The results of the synthesis process is a file containing a series of bits that when stored in local memory on the device causes it to implement the required functionality.

2.2.8 FSM operation on a reconfigurable device

The general hardware model of a FSM implemented in hardware is shown in figure 5. As this shows the FSM is implemented by a reconfigurable device, known as the Combinatorial Logic Block (CLB), responsible for implementing the TEs of the FSM and a feed back register used to store the current state of the FSM. The output of this register is then required to be fed back to the inputs of the CLB as the current state is required to calculate the next state of the FSM.

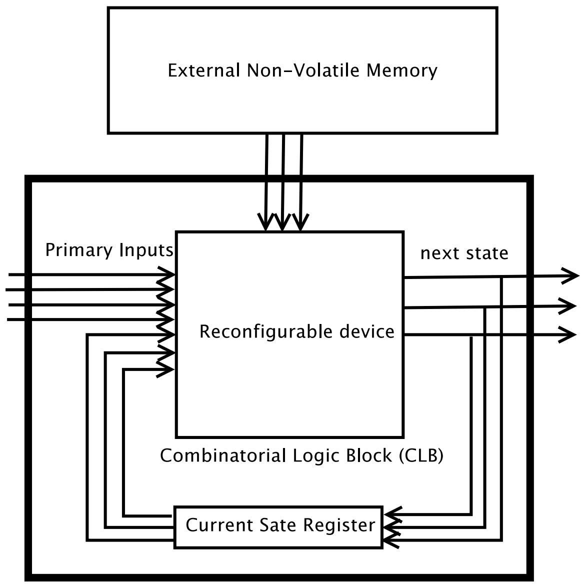


Figure 5: General Hardware model of FSM using Reconfigurable hardware

On power up the configuration bit-file is loaded into the local memory within the reconfigurable device from non-volatile memory. This configures the reconfigurable device to implement the required transition expressions and the current state register. Although many commercial devices such as FPGAs have the ability to implement feedback within the reconfigurable fabric itself it should be noted that certain reconfigurable devices, such as [3], do not allow feedback, this means an external register is required and feedback must be done out with the actual reconfigurable device.

As algorithm 1 shows, after the required configuration is loaded onto the device the next state would be calculated using the current state, initially this would be a specified *reset* state, and the current inputs. The system would then be clocked and the next state stored as the current state in the current state register, this new current state would then be used to again calculate the next state before the device is again clocked. This means that the clock speed of the device is limited by the time required for the device to calculate the next state and store it as the current state.

Algorithm 1 FSM operation on a reconfigurable device

```
WHILE{1}
  current_state=reset_state
  WHILE{reset=no}
    next_State=transition_expression(current_sate, primary_inputs)
    IF{next_state!= current_state}
     current_state=next_state
    ENDIF
  ENDWHILE
ENDWHILE
```

3 Novel representation of FSMs for reconfigurable hardware

As illustrated the main fixed hardware component of the general implementation is the combinatorial logic block (CLB). This block is required to implement the TEs of the FSM. In order to reduce the hardware requirements of the FSM this block was targeted for optimisation to produce a reconfigurable hardware device specifically tailored for the implementation of FSMs.

3.1 Investigation of characteristics of CLB

As the CLB implements the transition expressions (TEs) required to calculate the next state of the FSM from the primary inputs and the current state it is possible to calculate a number of the key characteristics of the block required to perform this function. The first of these is the number of outputs, if binary encoding is assumed, this is simply

$$O = \log_2(|S|)$$

It is also fairly trivial to show that, as feedback is required from the current state register, the total number of inputs to the combinatorial logic block is

$$I = I_p + O = I_p + \log_2(|S|)$$

As these equations show the inputs to the CLB can be divided into two categories. The first of these is the primary inputs; these are the input signals used to determine the next state transition and are necessary to the operation of the FSM. The remaining inputs are the feedback lines from the current state register. Consequently, if this feedback can be removed the total number of inputs to the array can be reduced by a factor of $\log_2(|S|)$, this in turn will lead to a reduction in the hardware required due to simplification of the expressions required to calculate the next state of the FSM.

3.2 Removal of current state feedback

As a reconfigurable hardware device has the capability of altering its functionality through programming it is possible to remove the need for the current state to be fed back to the inputs. This can be achieved by using the current configuration of the reconfigurable device to store the current state of the FSM, i.e. creating an individual configuration, or context, for each state of the FSM.

This method relies on the concept of calculating what state the FSM should move to compared to the traditional approach where by each state calculates if the FSM should enter this state. In this way this method directly calculates the next state of the FSM.

As this process calculates which state the FSM should move to, the contexts required to be loaded onto the reconfigurable device are know as FORWARD TRANSITION EXPRESSIONS (FTEs). The following section describes the derivation of these FTEs for the Mead-Conway example shown in figure 3.

3.2.1 Derivation of FTEs for Mead-Conway traffic light FSM

If the example of the Mead-Conway traffic light controller is again considered it can be seen that the truth-table shown in table 3 can be partitioned into sections that define the behaviour of the FSM when the FSM is known to be in a particular state. This is achieved by partitioning the truth-table into sections that contain all transitions *from* a particular state. In this example the truth-table is partitioned into four sub-truth-tables (one for each state) each of three inputs. Tables 4-7 show each of the sub-truth-tables for the Mead-Conway traffic light controller.

If the FSM is assumed to be in the HG state the following expressions can be used to calculate the next state:

$$\mathbf{Z}(\mathbf{t}) = \mathbf{HG} \quad \Rightarrow$$

$$\mathbf{Z}(t+1) \triangleq \overline{c.t_1}.\mathbf{HG} + c.t_1.\mathbf{HY}$$

Substituting the actual state labels yields:

$$Z_0(t+1) = 0;\; Z_1(t+1) = c.t1$$

If these expressions are analysed it can be seen that the next state is now a function of only the primary inputs. If these expressions are implemented on the reconfigurable device they are capable of indicating the state the FSM should enter at the next state transition.

4 FSM implementation on a custom reconfigurable architecture

The design flow for the implementation of FSM using the custom method suggested is shown in figure 6. As this shows the derivation of the truth table representation

Primary Input			Next State	
C	t1	ts	$Z_0(t+1)$	$Z_1(t+1)$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	0
1	0	1	0	0
1	1	0	0	1
1	1	1	0	1

Table 4: Sub-truth-tables for state HG

Primary Input			Next State	
C	t1	ts	$Z_0(t+1)$	$Z_1(t+1)$
0	0	0	0	1
0	0	1	1	1
0	1	0	0	1
0	1	1	1	1
1	0	0	0	1
1	0	1	1	1
1	1	0	0	1
1	1	1	1	1

Table 5: Sub-truth-tables for state HY

Primary Input			Next State	
C	t1	ts	$Z_0(t+1)$	$Z_1(t+1)$
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	0
1	1	0	0	0
1	1	1	1	0

Table 6: Sub-truth-tables for state FY

Primary Input			Next State	
C	t1	ts	$Z_0(t+1)$	$Z_1(t+1)$
0	0	0	1	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	0
1	1	1	1	0

Table 7: Sub-truth-tables for state FG

of the FSM from the high level specification of the system is the same as that for the full-custom and conventional reconfigurable implementation shown in figure 1. However, after the truth-table representation is produced the sub-truth-tables are extracted to produce $|S|$ sub-truth-tables. The information contained in each of sub-truth-tables is sufficient to derive the *Forward Transitions Expressions* for each state of the FSM.

FTE extraction is performed to produce a set of boolean expressions capable of calculating the next state of the FSM. This process is similar to the conventional TE extraction process except that, due to the removal of feedback, this process results in simplified boolean expressions that depend only on the primary inputs. As the

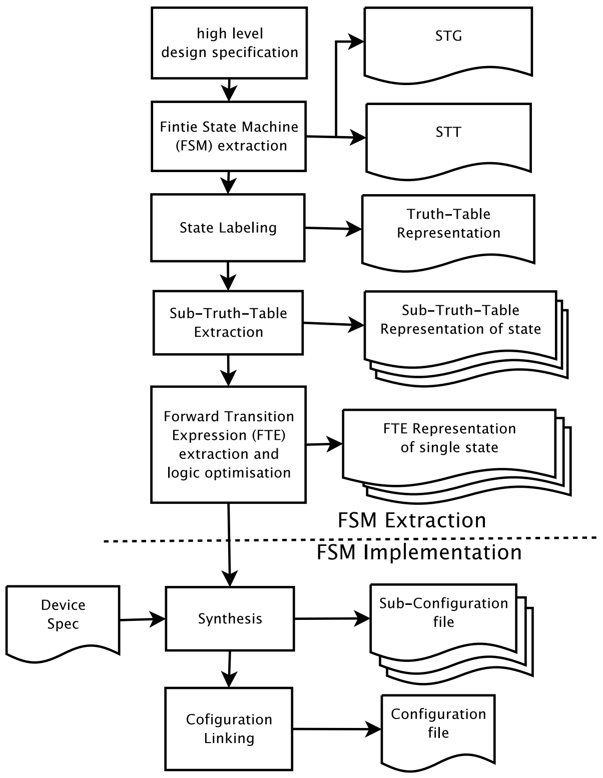

high level design specification
STG
Fintie State Machine (FSM) extraction
STT
State Labeling
Truth–Table Representation
Sub–Truth–Table Extraction
Sub–Truth–Table Representation of state
Forward Transition Expression (FTE) extraction and logic optimisation
FTE Representation of single state
FSM Extraction
FSM Implementation
Device Spec
Synthesis
Sub–Configuration file
Cofiguration Linking
Configuration file

Figure 6: Hardware design flow of a FSM for Custom Reconfigurable Device

CLB is required to produce the binary code necessary to indicate the next state, FTE extraction results in $log_2(|S|)$ boolean expressions for each state of the FSM.

The FTEs would then be synthesised using the appropriate synthesis tool for the family of devices selected by the designer for implementation of the CLB. This synthesis results in a bit-file that is capable of configuring the CLB to calculate the next state of the FSM based on only the primary inputs. At this stage it is then possible to calculate the hardware resources required within the CLB in order to implement the FTEs, this would be the maximum hardware required by any individual context. The device from the family of devices used with the closest hardware resources to this would then be selected in order to reduce the amount of redundant hardware in the final device.

After the final device has been selected it may be necessary to re-synthesis the

FTES in order to produce configuration files specifically for this device. After the configuration files for each state have been produced the final stage is to link these files together into a single bit-file that can be loaded into the local configuration memory of the device.

5 Reconfigurable device for the implementation of FTEs

As the device required to implement FSMs using FTEs is required to alter its configuration at each state change the reconfigurable device used to implement the CLB must be rapidly reconfigurable. For this reason a local configuration memory is required to be closely coupled to the CLB.

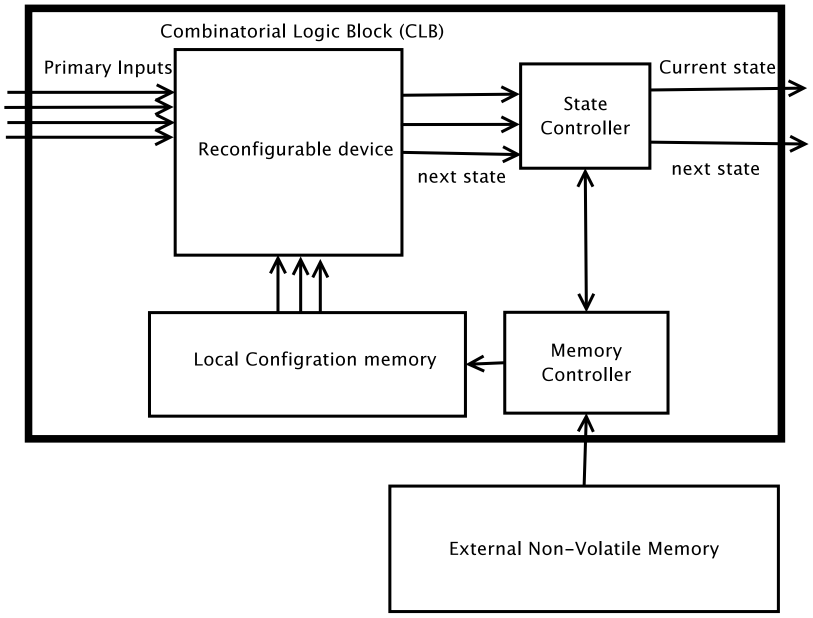


Figure 7: Custom Reconfigurable device for implementation of FSMs

The overall architecture of the device is shown in figure 7. Similarly to the general model shown in figure 5 this model contains a reconfigurable device used for the implementation of the FTEs. Due to the need to perform rapid run-time reconfiguration a local configuration memory has been added to provide local storage for the configuration file of the FSM. However, the main difference between this device and the traditional model is the addition of state and memory controllers responsible for detecting if a state change is required to take place and the selection of the configuration required to implement the next state of the FSM. After this new configuration has been loaded the next state would again be calculated at the next state transition based on the new context of the device.

During development of this device the similarity between this and a general purpose processor was noted. It would be possible for an external source to load the local configuration memory during operation of the device. In this way new "states" could be added to the FSM during operation allowing it adapt to the operating conditions of the device. In this way new states would be similar to instructions sent to the processor. The device would then enter one of these new states and determine which should be the next state in a very similar to branch statements on a general purpose processor. The device would then output the next state in a similar way to how a processor would present the result of a calculation and based on this new custom instructions/states could be loaded as required.

5.0.2 Combinatorial Logic Block (CLB)

The CLB is a reconfigurable device used to implement the FTEs of the FSM. This fact allows us to determine a number of characteristics of the block required to implement these expressions as discussed in section 3.1. The reconfigurable device must be capable of implementing combinatorial logic expressions and due to the removal of current state feedback the number of inputs to the block is equal to the number of primary inputs ($|I|$) and the number of outputs required is $log_2(|S|)$. Due to the context changes required during state changes the device must also exhibit very low reconfiguration times as this dictates the maximum rate at which state changes can take place.

The reconfiguration time (C_t) of the device is a function of the clock rate (C_{rate}), configuration bus width (C_{width})and the amount of data (D_{conf}) required to configure the device. One suggested method of configuring the device is with the aid of scan chains, these are registers chained together in series similar to a shift register used to store configuration data. In the case of memory based reconfigurable devices these registers are the memory components of the reconfigurable devices itself chained together in series. In this way blocks of data blocks equal to the width of the configuration bus (C_{width}) are loaded onto the device in each clock cycle and this data is then shifted through each configuration register until all of the data required to configure the device has been loaded. If it is assumed that a single clock cycle is required to write each block of data to the device it is possible to calculate the reconfiguration time of the device using the expression

$$C_t = \frac{D_{conf}}{C_{width} * C_{rate}}$$

Based on this expression it can be seen that, as the clock rate of the configuration bus is dependent on the technology used to implement the device and is hence out with the control of the device designer, the designer can control the configuration times of the device in one of two ways, either the size of the configuration bus can be altered or the amount of data can be adjusted.

By varying the width of the configuration bus the size of the data chunks written to the device in each clock cycle can be adjusted and hence adjust the overall configuration time of the device. In order to reduce the configuration time it is suggested that as wide a bus as possible be used to program the configuration memory of the reconfigurable device but it should be noted that this increase in bus width will lead to an increase in the overall size of the device. This means the design of a suitable device becomes a trade off between the reconfiguration time of the device and the overall size of the device.

The amount of data required to configure the device is dependent on the size of the device or the amount of configurable elements in the reconfigurable array. This means it is important to ensure that the smallest array possible be used to implement the FTEs as this not only affects the configuration time of the device but also the area and possibly power of the resultant device. Due to the use of FTEs the array required to implement FSMs is far smaller than would be required using the TE approach. This is due to the fact that, using the FTE approach, only the expressions for a *single* state are required to be implemented concurrently where as for the TE approach the expressions of *each* state must be implemented concurrently. In this way it is hoped that the reduction in hardware will result in reduced configuration times.

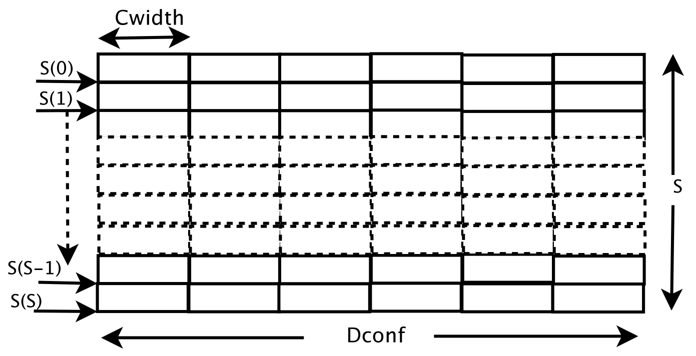


Figure 8: Configuration Memory Structure

5.0.3 Configuration memory

The configuration memory is responsible for storage of the configuration file required to implement an FSM using the FTE based approach. The memory must thus be capable of storing $|S|$ distinct configurations each of D_{conf} bits in length, where D_{conf} is the number of bits of data required to program the CLB. Using this it is possible to calculate the overall size of the memory block (M_{size}) required to implement an FSM using the identity

$$M_{size} = D_{conf} * |S|$$

As minimisation of the configuration time of the CLB is vital to the operation of the device it is suggested that a custom dual port memory block be used. This allows a standard size bus to be used to allow the memory to be programmed using a standard external bus of width M_{width} and a custom sized bus for the loading of configuration data onto the CLB of size C_{width}. The use of a standard sized external bus provides an interface to the memory block that allows it to interact with standard external memory blocks such as ROM or SRAM but results in a memory block that is a exact multiple of the bus width which is unlikely to be exactly equal to M_{size}. If an external bus of size $M_{width}$is used the exact size of the configuration memory (M_{exact}) can be calculated using the relation

$$M_{exact} = B * M_{width}$$

where B is the number of blocks of M_{width} required to program the block and $M_{exact} \geq M_{size}$.

In order to allow for easy selection of the required configuration data for the next state it is suggested that the memory block be partitioned as shown in figure 8. In this way the memory block is partitioned into $|S|$ blocks of length D_{conf} where each block is individually addressable. This allows the next state as indicated by the CLB to be used directly in selecting the correct configuration data. Each block would then be subdivided into blocks of length C_{width}that can be passed via the configuration bus to the CLB.

5.0.4 State Controller

The state controller is responsible for detecting when a state change is required to take place and initiating the reconfiguration required to perform this. The state controller locally stores the current state and constantly compares this to the next state indicated by the CLB. If the next state is found to be different from the current state this indicates to the controller that a state change is required.

If a state change is detected the state controller must latch the inputs and the outputs (next state) of the CLB before initiating a reconfiguration. The state controller then indicates to the memory controller that a state change is to take place and passes control to the memory controller.

After the memory controller has performed the operations required to reconfigure the CLB control is passed back to the state controller. The state controller then releases its hold on the inputs and stores the next state as the current state in an internal register.

Due to latency introduced by the reconfigurable fabric the state controller must hold the output of the CLB for a number of clock cycles to ensure that these inputs are allowed to ripple through the reconfigurable fabric. If the output is not held for this period it is possible that false state changes may be indicated causing the FSM to operate in an incorrect manner.

5.0.5 Memory Controller

This component can be separated into two main sections. The first of these is responsible for loading the configuration file required to implement a FSM into the local configuration memory. This involves reading fixed length packets from an external bus that would usually be connected to non-volatile memory such as ROM.

The memory controller is also responsible for performing reconfiguration of the CLB when indicated by the state controller. Using the next state as indicated by the

CLB the memory controller is responsible for selecting the appropriate context and loading this onto the CLB. This involves reading fixed length blocks of data onto the configuration bus and indicating to the CLB that this data is to be read. In the case of scan chains this process continues until all of the required data has been loaded onto the device. The memory controller would then pass control back to the state controller to allow the device to operate using this new current state.

As the amount of data to be passed and the size of the configuration bus are dependent on the CLB and configuration memory the timings and operation of memory controller are specific to the exact characteristics of these components. This means the memory controller can only be generated after these components have been designed and the memory controller can only be used with components with these exact characteristics. This means that if any changes are made the memory controller must be redesigned, as such the generation of this component is usually left until all other components have been produced and their operation ensured to be correct.

5.0.6 Device operation

The operation of the device shown in figure 7 is illustrated by the algorithm shown in algorithm 2. The CLB obtains its initial configuration from the data stored in the associated memory and calculates the next state based solely on the primary inputs. The output of the CLB (next state) is then compared to the value stored in the current state register (current state). If the next state does not equal the current state a state change is indicated and the device obtains the FTEs for the state indicated by the reconfigurable device. The configuration required to implement the FTEs for the next state is then automatically obtained from the configuration memory and loaded into the CLB. The device can then again calculate the next state using the new FTEs.

It should be noted that during reconfiguration, or state change, the state variables (input vector) would be required to be held at their present values until the device

Algorithm 2 FSM operation on a custom reconfigurable device

```
WHILE{1}
  device_cfg=reset_cfg
  current_state=reset_state
  WHILE{reset=no}
    next_State=transition_expr_current_state(primary_inputs)
    IF{next_state!=current_state}
      device_cfg=next_state_cfg
      current_state=next_state
    ENDIF
  ENDWHILE
ENDWHILE
```

is fully reconfigured to guarantee that not state change events will be missed. This problem is fairly easily rectified at design time as the number of clock cycles required to reconfigure the CLB are known and the input registers can be held for the required number of clock cycles before being released by the state controller.

During reconfiguration it is necessary to either hold the output of the CLB or instruct the state controller to ignore any state changes indicated by the CLB until reconfiguration is complete, otherwise false state changes may be indicated to the state controller. During reconfiguration the output of the CLB will be held in a similar way to the state variables but due to the latency of the CLB the output would be required to be held for a period equal to the number of clock cycles required to propagate the state variables to the outputs of the CLB. Again, as the characteristics of the CLB would be known at design time this would allow this value to be hard coded into the state controller which would be responsible for releasing the hold when applicable.

5.1 Generation of device Implementation of Custom Reconfigurable device

In order to test the method suggested here a 'C' based HDL generator was created that automates the production of the code required to implement the device shown in figure 7.

The generator takes FSM parameters, such as the number of states ($|S|$) and number of primary inputs ($|I|$), and based on this produces the HDL required to implement the device required to implement the FSM using FTEs. Although the generator is capable of implementing the necessary hardware using only these high level parameters this is achieved by estimating the the size of the reconfigurable block required to implement the CLB. This means that in order to ensure the FSM can be implemented, excessive redundant hardware may be added.

To reduce the amount of redundant hardware added by this estimation a method of determining the minimum hardware required for a particular FSM, or set of FSMs, was also developed. The generator is thus capable of operating in two ways, it can either generate a domain specific device for specific FSMs or generic devices that estimate the hardware requirements of the FSM.

5.1.1 Generation of Generic device

The HDL generator can produce non-application specific devices intended to be produced as off-the-shelf components. The characteristics of the device are estimated from the maximum number of states (S_{max}) and primary inputs (I_{max})to the device. In this way a range of devices would be produced that are capable of implementing FSMs using the FTE approach presented here. For each of these devices the maximum number of primary inputs and states would be given and the end user would select the device that is closest to the characteristics of the FSMs to be implemented.

The hardware produced using this method is likely be sub-optimal for the implementation of a specific FSM as in order to allow the device to be used in as many applications as possible components such as the CLB and memory are designed to have enough hardware to implement even the most complex of FTEs.

Based on the number of primary inputs the hardware resources required by the CLB are estimated and the device generated. It is then possible to calculate the amount of data required to configure the CLB and using this it is simple to calculate the size of the memory block required to store the entire configuration of the FSM. This process results in a device that is capable of implementing any FSM with less than $|S|$ states and $|I|$ inputs but will likely have redundant hardware in the CLB that will only be used when implementing a small sub-set of all possible FSMs.

5.1.2 Generation of Domain specific device

In order to reduce the inefficiencies of the generic device the tool can also produce a device specifically tailored to a set of FSMs. In this mode a set of FSMs is used to determine the minimum hardware requirements of the device that is capable of implementing each of the FSMs in the set.

This is achieved by producing the sub-truth-tables for each of the FSMs and calculating the hardware resources of the CLB required to implement each of the states of each of the FSMs. The maximum size of CLB required by any single context is then found and a CLB of this size produced. In this way a CLB is produced that contains the minimum hardware required to implement the set of FSMs.

After generation of the CLB it is possible to produce the local configuration memory required to store the configuration data. This is calculated by simply finding the FSM in the set with the largest number of states and multiplying this by the amount of data required to configure the CLB. The remainder of the device can be generated in a similar way to the generic device approach discussed previously.

6 Results

In order to investigate the effectiveness of the method presented here it is necessary to compare the results of implementation to those obtained using the traditional method of implementation for reconfigurable devices. It is expected that the method presented here will allow FSMs to be implemented using a smaller CLB than would be required if the FSM were implemented on a reconfigurable device using the traditional method presented in section 4.

In order to compare the implementation methods the number of LUTs used to implement the CLB of a set of FSMs was collected for both the traditional and FTE based implementations. The FSMs used to perform this comparison are selected from the MCNC benchmark suite [13] and are selected to represent as broad a range of FSMs as possible. For this reason the MCNC benchmarks were first profiled in terms of number of inputs and states and the corner cases selected.

In order to calculate the hardware resources of the reconfigurable device required to implement the selected benchmark circuits, using the traditional method, the circuits are synthesised for implementation using 4 inputs LUTs. This synthesis was performed using the SIS [5] synthesis tool and the number of LUTs required to implement the CLB were calculated.

The results obtained using this method were then compared to a number of results previously published concerning the number of LUTS required to implement these benchmark circuits. Of particular interest are the results presented in [4]. These results are very similar as they are produced using a similar synthesis method, i.e. SIS and Flowmap, but in this case the results are lower due to the combined use of Roth-Karp decompostion, cube-packing and modified cube extraction to ensure the minimimum numebr of LUTs are required.

After the number of LUTs required for the traditional method are found the FSM can be broken down into sub-truth-tables as described in section 3.2.1 and the number of LUTs required to implement EACH state are calculated using SIS. As a reconfigurable device must be able to implement all states, the minimum size of the reconfigurable device required is equal to the maximum number of LUTs required to implement any of the FTEs of any single state.

It is also possible to calculate the memory required to store the bit file used to program the device based on the number of LUTs (L) and the number of inputs of the LUTs (K); for the traditional system the number of program bits (D_{conf}) is

$$D_{conf} = L * 2^K$$

In the case of the FTE based implementation the number of program bits is calculated by

$$M_{size} = D_{conf} * |S| = (L * 2^K)|S|$$

The results obtained for the selected MCNC benchmark circuits are shown in table 8. As this table shows that for each of the benchmark circuits the number of LUTs required is substantially reduced by making use of FTEs. This is due to the fact that only a small part of the FSM is required to be implemented at any time. It can also be seen that in the majority of the cases presented here the amount of memory required to store the configuration data is also reduced.

In the case of DK17 and Opus the use of FTEs actually increases the amount of configuration data required. In these examples the FSMs have relatively large numbers of states compared to inputs and in particular have a few states with very large numbers of transitions. For the FTE implementation this results in a single state, or relatively few sates, requiring large numbers of LUTs compared to the remaining states of the FSM. As the size of the device required is set by the maximum number of LUTs required by ANY single state this results in a large CLB where the majority of the states utilise only a small fraction of the available hardware but as all of the LUTs require programming D_{conf} is the same for each state.

			TE-based		FTE-based	
Benchmark circuit	$\lvert S\rvert$	$\lvert I\rvert$	LUTs	MEM (b)	LUTs	MEM (b)
DK15	4	3	21	336	2	128
DK17	8	2	14	224	3	384
Planet	48	7	340	5440	6	4608
Kirkman	12	16	139	2224	4	768
Ex1	20	9	43	688	10	3200
Opus	10	5	143	2288	11	1760

Table 8: Number of LUTs in CLB for TE and FTE implementation methods

It should be noted that although the use of FTEs allows for simplification of the CLE addition hardware must be included to allow for the run-time reconfiguration of the device and the automatic detection and handling of state changes. As illustrated in figure 7 components such as the state controller, memory controller and local configuration memory must be added to allow for the automatic operation of the device. It is felt by the author that any savings in the CLB will be overshadowed by the addition of these components due to the complexity of these components. It can also be seen that although these components would be highly dependent on the actual structure and size of the CLB they can be directly generated once these characteristics have been determined. This would allow a custom logic block to be created that would allow for optimisation of these blocks reducing their area, speed and power requirements. This means that without performing full synthesis, in terms of silicon area, it is unclear as to what advantages this implementation would have over the traditional method.

If the power consumption of the device is concidered it can be seen that due to the requirement for the CLB to be completely reconfigured at every state change the dynamic power consumption of the device is likely to be greater than that of a traditional reconfigurable device that is required to be configured only at device start up. Again this effect is hard to quantify without performing full synthesis as it is unclear if the reduction in the size of the CLB will result in a sufficient reduction in power consumption to compensate for the need to perform reconfiguration.

7 Future Work

During the process of investigating the implementation of FSM using the method suggested here a number of future optimisations and areas of investigation were considered. This section gives details of some of the most promising areas for which it is suggested that future work on this project area may be advantageous.

7.1 Virtual Output addressing

In the traditional approach to FSM implementation state labelling as shown in section 2.2.4 is performed to give each state an individual binary representation that can be expressed using digital logic. This process requires that the CLB be able to indicate $|S|$ individual binary codes requiring $log_2(|S|)$ outputs and hence $log_2(|S|)$ FTEs. In the case of FTEs the CLB is only required to be able to indicate the next state of the FSM. This means that the CLB is only required to be able to indicate a unique binary code for each of the states to which a transition from the current state is possible.

Using this method the device designer would be first required to profile the FSM, or set of FSMs, the device is required to implement. This profiling would find the state with the maximum number of transitions T_{max}. For each state the outputs given in the sub-truth-table would then be converted to a new virtual label $log_2(|T_{max})$ bits in length that is particular to each state. A new table would then be created to implement address translation between this virtual address and the original labels created during state labelling.

If the Mead Conway example shown in figure 2.2.2 is again revisited it can be seen that using FTEs to implement the FSM requires a CLB with $log_2(4) = 2$ outputs to indicate each individual state of the FSM. If this FSM is to be implemented using

FTEs obtained from the sub-truth-table presented in tables 4-7, it can been seen that each of the states has a maximum of 2 possible state transitions or $T_{max} = 2$. It is thus possible to indicate the next state of the FSM using only $log_2(2) = 1$ outputs and performing address translation to convert the output of the CLB to the corresponding label of the next state. If virtual addressing is carried out on the sub-truth-table for the HG state presented in table 4 the resultant table is shown in table 9 and the data required to perform address translation is shown in table 10.

As this method is dependent on the actual structure of the FSM this optimisation relies on having information concerning the FSMs to be implemented and is hence application, or at best domain, specific.

Primary Input			Output
C	t1	ts	V_0
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Table 9: State HG sub-Truth-Table

Input	Outputs	
V_0	$Z_0(t+1)$	$Z_1(t+1)$
0	0	0
1	0	1

Table 10: Address Translation

7.2 Virtual Input addressing

Virtual Input addressing is a further optimisation that makes us of the fact that many of the transitions contained in an FSM are only dependent on a small sub-set of the primary inputs. This is related to the boolean simplification of FTEs as this often results in simple expressions that are dependent on relatively few of the primary inputs. In a similar way to the use of Virtual Output Addressing presented in section 7.1 FSMs would be profiled and the maximum number of inputs (I_{max}) required by any transition would be found. A CLB with this number of inputs would then be able to implement the FTEs of this FSM by programming the inputs to only pass those required by the current state

If the Mead Conway example is considered it can be seen that although the FSM has 3 primary inputs the maximum number of inputs in any of the FTEs is 2 as for each of the FTEs at least one of the inputs is removed during boolean simplification reducing the number if inputs required by the CLB. Again, as with Virtual Output Addressing, this optimisation is application specific and relies on data obtained from the FSMs required to be implemented.

7.3 Partial reconfiguration and Configuration Caching

Recently there has been much interest in the possibility of performing partial reconfiguration of reconfigurable devices. This process involves configuring only small sections of a reconfigurable device rather than configuring the entire device during each reconfiguration. In this way a device could be made to implement several functions simultaneously and if any of these functions is required to be altered it can be reconfigured without interfering with the operation of the other functions implemented by the device. By only reconfiguring the parts of the device that are required to be altered this process aims to reduce the amount of data required to configure the device.

In the case of the use of FTEs this technique could be used to limit the amount of data required to reconfigure the CLB and hence reduce the time required to perform state changes. Using this method, for each state change, the configuration of the current state would be compared to the configuration of the next state and only the differences in these configurations would be required to be loaded onto the device. It is impractical to perform these comparisons during run-time and hence this would be determined in advance during synthesis. In this way a distinct context would be produced for each possible transition of the FSM rather than for each state. Although it is hoped that this method will result in lower configuration times due to reductions in the size of the individual configurations that must be loaded during state changes the large number of configurations required may cause the size of the configuration memory to grow very large and impact greatly on the efficiency of the resultant device.

A possible solution to this issue is the use of a system similar to caching used in modern processors. This would take advantage of the fact that only relatively few of the states of an FSM are usually reachable from the current state and is related to the concept of Virtual Output addressing presented in section 7.1.

In this system only the partial-configurations of states that are reachable from the current state would be required to be loaded into the configuration memory and the appropriate configuration selected when a state change is detected. At this point the partial-configurations of each of the reachable states of the next state would be loaded into configuration memory. It should be noted that although the method aims to reduce the size of the memory block required to store configuration data this is highly dependent on the structure of the FSM to be implemented and may result in application specific devices. This method would also require far more complex state and memory controller components and as such it is unclear without further investigation if this wold result in increased efficiency in terms of area and power.

8 Conclusion

This portfolio document has presented a novel representation of FSMs specifically tailored to take advantage of the properties of reconfigurable hardware.

The ability of reconfigurable hardware to be reconfigured during run-time allows these devices to calculate the next state of FSMs using only the primary inputs. This is made possible through the use of run-time reconfiguration and FTEs to produce a unique context, or configuration, for each state. By making use of this novel representation the feedback register traditionally used in FSM implementation is no longer required, reducing the inputs to the CLB by a factor of $\log_2(|S|)$.

In order to investigate the effect of the novel method presented here selected MCNC benchmark circuits were implemented and the hardware characteristics recorded. The MCNC benchmark suite was profiled and the circuits selected to represent the corner cases in terms of number of inputs and states.

For each of the examples selected the use of FTEs vastly reduces the number of LUTs required to implement the FSM. The results also show that FSMs with state transitions spread evenly across the states, rather than single states with large numbers of traditions, are most suitable for the use of FTEs as this results in FSMs with far lower LUT counts as well as reduced configuration memory requirements.

Future areas of optimisation and investigation have also been identified which may further reduce the hardware resources required to implement FSMs using FTEs. Optimisations such as virtual input and output addressing allow the size of the CLB to be further reduced but are application specific and rely on knowledge of the specific FSMs to be implemented. The possibility of performing partial reconfigurations instead of configuring the entire CLB during state changes aims to reduce the amount of configuration data required and hence reduce the reconfiguration times of the device but may produce very large configuration files as a unique configuration is required for each state transition rather than for each state. This would result in long start up times for the device and require very large configuration memories as well as far more complex state and memory controller components.

The work carried out here was considered novel enough to be considered for legal protection in the form of patenting. Although extensive consultation was carried out with patent lawyers, due to time constraints, this process has not been completed by the time of writing this report. Instead the paper presented in Portfolio document CP1 has been submitted to Adaptive Hardware Systems Conference (AHS) 2007 where it is accepted for publication.

Implementation of Control Systems using Reconfigurable Hardware

References

[1] Cadence Encounter Digital IC Design Platform. http://www.cadence.com/products/digital_ic/index.aspx.

[2] GDS-II Format. http://www.rulabinsky.com/cavd/text/chapc.html#GDSArray.

[3] S.J.E. Wilton A. Yan. Product-term based synthesizable embedded programmable logic cores. In *IEEE Transactions on VLSI*, pages 474–488, 2006.

[4] A. Dehon. Balanced interconnect and computation in a reconfigurable computing array. In *Proceedings of the 1999 International Symposium on Field Programmable Gate Arrays*, pages 69–78, 1999.

[5] E. M. Sentovich and K. J. Singh and L. Lavagno and C. Moon and R. Murgai and A. Saldanha and H. Savoj and P. R. Stephan and R. K. Brayton and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, 1992.

[6] A. Gill. *Introduction to the Theory of Finite-state Machines*. McGraw-Hill, 1962.

[7] S. Hauck K. Compton. Flexibility measurement of domain-specific reconfigurable hardware. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 155–161, New York, NY, USA, 2004. ACM Press.

[8] L. Scheffer L. Lavagno, G. Martin. *Electronic Design Automation for Integrated Circuits.* CRC, 2006.

[9] M. L. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, 1967.

[10] S. Palnitkar. *Verilog HDL.* Prentice-Hall, 2003.

[11] T. Villa T. Kam, R. Brayton, , and A. Sangiovanni-Vincentelli. *Synthesis of Finite State Machines: Functional Optimization.* Kluwer Academic Publishers, 1997.

[12] T. Villa, T. Kam, R. Brayton, and A. Sangiovanni-Vincentelli. *Synthesis of Finite State Machines: Logic Optimization.* Kluwer Academic Publishers, 1997.

[13] S. Yang. Logic synthesis and optimization benchmarks user guide version. Technical report, 1991.

Portfolio Document TR2

Investigation of the Characteristics of Reconfigurable Devices.

Author:

Graeme Milligan

Supervisor:

Wim Vanderbauwhede

Contents

List of Figures

List of Algorithms

List of Tables

1 Introduction

Reconfigurable hardware, such as Field Programmable Gate Arrays (FPGAs) and Programmable Logic Devices (PLAs), can be used to introduce hardware flexibility to modern electronic devices. Hardware programmability introduces the ability to determine the functionality of a silicon device after manufacture, allowing hardware designers to begin the hardware design process earlier in the design cycle and develop the necessary configuration files concurrently. The use of reconfigurable hardware allows design errors to be corrected after a device has been manufactured by simply reprogramming the device hence reducing the quantity of pre-manufacturing testing that must be carried out on new devices. This introduces the concept of using pre-design and verified generic hardware devices that can be programmed as required by the user, allowing the design cost of these devices to be amortized over a number of future projects in a similar way to the use of the generic microprocessor.

One of the main issues concerning reconfigurable hardware is the difficultly in determining the type and amount of hardware required for an application where a final implementation of the device is not yet available. This is particularly true where the reconfigurable device is intended to act as a standard platform to be used in a variety of applications or be embedded within a larger device such as a System on Chip (SoC). Due to the need for generality, it is difficult for reconfigurable device designers to determine the amount of hardware resources that should be placed within the device. This is a difficult balancing act between the need to include enough hardware to implement as broad a range of designs as possible and producing devices with large amounts of *wasted* redundant hardware.

1.1 Project goals

This research aims to provide a method of determining the optimal hardware requirements of reconfigurable devices to ensure that they can implement as broad a range of designs as possible while avoiding unnecessary, redundant hardware. This is based on estimating the maximum and mean hardware requirements of the device based on high level parameters such as the number of inputs to the device.

In order to begin such an investigation it is necessary to perform a detailed analysis of the operation of both logic and memory based reconfigurable devices with particular focus on how these devices implement systems and how this determines the hardware resources required within these devices. This research is presented in section 2. Next, the amount of hardware that is required within PLA based reconfigurable devices is presented in section 3. The method presented was then extended to the investigation of LUT based architectures in section 4 before final conclusions are made.

2 Background theory

Currently there are two main types of reconfigurable logic device available: these are memory based devices such as FPGAs and logic based devices such as the PLA. Recently memory based devices have become more popular due to their large gate counts and the ability to implement complete, complex systems on a single device. Along with the production of large FPGA devices such as those from Xilinx [38] and Altera [3] there is also interest in integrating small blocks of reconfigurable hardware into large System on Chip (SoC) devices to introduce flexibility within these devices, an example of this is [28].

The inclusion of this hardware allows large generic chips to be produced where the precise details of the functionality of the device can be determined post manufacture, allowing for their use in a larger application domain. While this increases the applications the device can be utilised in, the use of non-application specific hardware reduces the efficiency of the device due to the general purpose nature of this hardware and the associated logic required to provide programmability.

The following sections detail both the LUT and logic based approaches to reconfigurable devices and describes the method by which logic functions are mapped to these devices.

2.1 Truth-table representation and Boolean simplification of Combinatorial Logic Expressions (CLE)

One of the most commonly used methods for the representation of CLEs is the truth table. This is a table of all possible input combinations that can be presented to the CLE and the output required to be produced by the CLE. A sample truth table is shown in table 1 for a 3 input CLE.

From the truth table it is possible to extract a Boolean description of the CLE in sum-of-products (SOP) format. In SOP format, a product (AND), or minterm, is created for each *'1'* in the output column of the truth table and the product terms are then summed (*OR*) to give the final CLE. For the truth table in table 1 this results in the following expression

$$Z = \overline{ab}c + a\overline{bc} + a\overline{b}c + ab\overline{c}$$

It is common practice to use boolean simplification techniques to remove redundant logic from the expressions. This reduces the hardware required to implement the required function. The use of Boolean algebra is common place in engineering and has been an active area of research since the early days of Computer Aided design. As such, a large volume of work is available detailing this process including [31, 36, 7]. This work has been used to create a number of exact minimisation algorithms, such as espresso-EXACT [32] and Mcboole [17]. These algorithms are guaranteed to produce the minimum SOP expression. The espresso-EXACT and other minimisation algorithms are utilised by tools such as SIS [6] to automatically perform boolean minimisation.

When Boolean simplification is performed on the expression in table 1 the expression becomes

$$Z = a\overline{c} + \overline{b}c$$

As this shows boolean simplification has reduced the number of product terms by half. This has a direct effect on the size of the PLA required to implement the CLE by reducing the number of equivalent gates required.

Inputs			Output
a	b	c	Z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Table 1: Sample 3-Input truth table

2.2 Programmable logic arrays

The PLA, and its variants, were the first hardware devices that provided the opportunity to determine their functionality post-manufacturing and as such a large volume of work is available on the design and operation of these devices [8, 21]. Early devices achieved this through the use of one-time programmable fuses or anti-fuses which limited the devices to be one time programmable. This prevented the reuse of these devices and also prevented error correction through programming. In order to solve this problem modern devices, such as [16], make use of reprogrammable blocks to allow the devices to be programmed multiple times.

PLAs are designed to implement combinatorial logic expressions (CLE) and their design is derived directly from the sum-of-products (SOP) representation of these circuits. A simple combinatorial logic expression in sum-of-products format is shown below

$$Z = \overline{a}bc + a\overline{b}c + ab\overline{c}$$

This expression is constructed of a series of product terms that implement the *on* conditions of a logic expression by *AND*ing the necessary inputs. The final expression is then produced by *OR*ing, or summing, these product terms. It is thus possible to implement the expression using simple *AND* and *OR* gates as is shown in figure 1

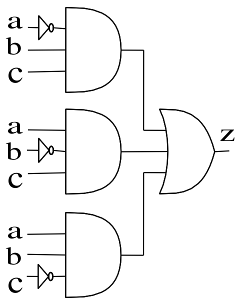


Figure 1: Implementation of SOP expression

As all SOP expressions can be implemented using *AND* and *OR* gates in this way, it possible to build a general platform that is capable of implementing these expressions. As each of the product terms is produced by an *AND* gate, the device must contain enough of these to produce the necessary number of product terms. The outputs of the *AND* gates are then connected to a large *OR* gate to implement the final SOP expression.

The structure of a typical PLA is shown in figure 2. This device contains a plane of *AND* gates to implement the required product terms. The output of these gates are then routed to the inputs of the *OR* gates in order to sum the product terms to produce the final expression. The connections between the outputs of the *AND* gates and the inputs of the *OR* gate are programmable through the use of programmable switches allowing the user to determine which of the *AND* gates is connected to which *OR* gate and hence determine the final functionality of the PLA.

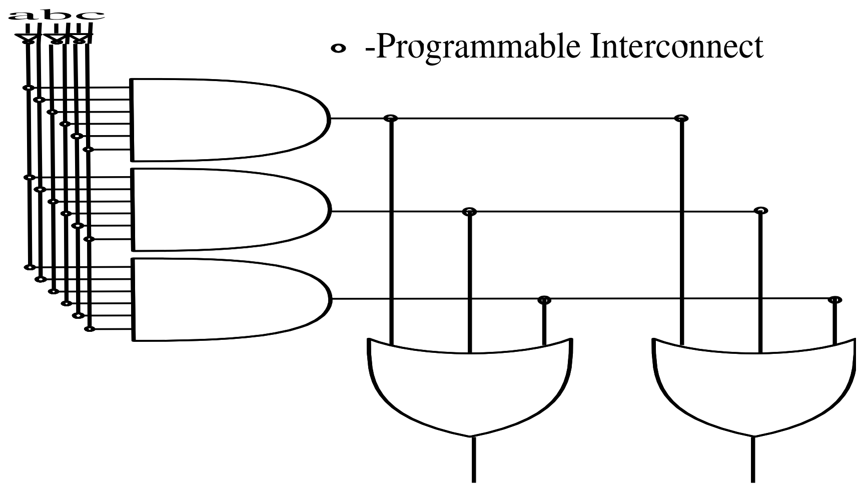


Figure 2: PLA structure

The typical PLA can be fully described using the tuple

$$\{i, p, o\}$$

where,

i is the number of inputs,

o is the number of outputs and

p is the number of product terms.

Based on these parameters it is possible to determine the hardware characteristics of the device as;

- The number of inputs (i) sets the number of inputs required by the AND gates in the AND plane. As shown in figure 2 each AND gate has two inputs for each PLA input as it is normal to provide an inverted version of each of the inputs.
- The number of outputs (o) is equal to the number of OR gates in the OR plane.
- The number of product (p) terms determines the number of programmable switches in the array but as a result also determines the number of AND gates in the AND plane and the number of inputs required by the OR gates in the OR plane. It can also be seen that the number of product terms determines the complexity of the SOP expression that can be implemented and as such has a major impact on the type and number of circuits the PLA is capable of implementing.

In order to ensure the PLA can implement as broad a range of SOP expressions as possible it is essential that enough product terms are available on the device. From the SOP expression this can easily be found by simply counting the number of product terms in the expression but for applications where the SOP expressions for the application are not available this method can not be utilised.

2.2.1 Measure of flexibility

As stated the amount of hardware available in a PLA has a direct effect on the amount and variety of circuits the device can implement. If too few product terms are placed within the fabric of the device the complexity and hence the number of logic circuits the PLA can implement will be severely limited. This means that although a PLA device may report lower speed, area and power requirements when compared to other architectures this does not give a clear indication as to the usefulness of the device as these metrics do not include any measure of the capabilities of the device to implement CLEs. For this reason [13] suggests the use of a measure of flexibility for reconfigurable device's that is based on the devices ability to implement a range of logic circuits.

This measure of flexibility can also be viewed as a measure of the coverage of the device. This is based on determining the percentage of a particular set of circuits that the device is capable of implementing. It is then possible to use this value as a design metric for the comparison of different reconfigurable architectures. In the case of PLAs this method can be used to assess the effect of altering the number of product terms within the array on the complexity and number of circuits that can be implemented.

An alternative way of using this measure of flexibility is to use a very large set of logic expressions and calculate the hardware requirements of the PLA required to implement these expressions. In this way the characteristics of a PLA would be selected based on these[26] results in order to achieve a required level of flexibility rather than measuring this after the device has been designed. As the number of product terms is vital in determining the hardware required by a PLA, a number of projects [35, 5, 34] have attempted to calculate the number of product terms in general CLE expressions with particular interest in the average, or mean, number of product terms. In the examples it was found that the average number of product terms in the SOP expression is related to the number of literals, or inputs, to the expression.

This relationship is fairly evident, if the truth table in table 1 is again considered, it can be seen that as the table is 2^i in length and a product term is produced for each *'1'* in the output column, the maximum number of product terms possible in a SOP expression of I inputs is also 2^i. However if an expression was created with this number of product terms, boolean simplification would result in an expression with no products and in reality would represent a circuit whose output is always high.

It can be seen that as the number of products in the unsimplified CLE increases, the likelihood of boolean simplification taking place also increases. This means that although an unsimplified CLE may have a large number of product terms, after simplification the result may be a circuit with few product terms. By examining the method of boolean simplification conventionally carried out, and in particular Karnaugh maps (k-map) , it was found that the maximum number of product terms that can be found in a simplified CLE is actually 2^{i-1}. Table 2 shows the k-map for a 3 input CLE with the maximum number of product terms. It can be seen that if any further *1*s are added to this k-map, simplification can take place by grouping these *'ON'* terms and does not result in extra product but instead results in either simplified product terms or the complete removal of a product term. In terms of logic functions this expression is the exclusive *'OR'* (XOR) boolean operation.

This simplification means a PLA produced with sufficient hardware to implement 2^{i-1} product terms can be guaranteed to be able to implement **any** combinatorial logic expression with i or less input. This is due to the fact that no CLE of I inputs can have more than this number of product terms and a circuit with less than this number of inputs must also have less than 2^{i-1} product terms. This would result in a device with a flexibility score of 1or in other words a device that can implement 100% of all circuits with less than i-inputs.

Although this fact gives the maximum number of product terms for an i input expression, there are very few circuits that will require this number of product terms with the majority having far less than this. For this reason, producing a PLA with this number of product terms is unnecessary except for critical domains where it must be guaranteed that the PLA can implement any circuit of less than i inputs where the ability to provide upgradeability and error correction is of prime importance. Of more interest is the mean number of product terms and the distribution of these terms as this allows designers to select the degree of coverage, or flexibility, of the

PLA.

	c	$\overline{c}$
ab	0	1
$a\overline{b}$	1	0
$\overline{ab}$	0	1
$\overline{a}b$	1	0

Table 2: 3-Input K-map of function with maximum number of product terms

2.3 Field-Programmable Gate Arrays (FPGA)

The architecture of modern FPGAs is usually described as being *island* style, where the islands are programmable logic blocks used to implement the required functionality surrounded by a *sea* of routing used to connect these logic blocks in the desired manner. The typical architecture of an FPGA is shown in figure 3. The logic units are usually constructed of small memory blocks known as Look-Up-Tables that implement combinatorial logic expressions (CLEs). Recently device's have begun to include embedded components such as memories, DSP components and even complete processors in order to increase the efficiency of these device's when implementing larger systems.

Due to the mature nature of these devices a large volume of literature is available including [12, 9, 24]. The following section aims to give an overview of FPGA design and circuit implementation.

2.3.1 Logic blocks

Modern FPGA devices make use of Look-Up-Tables (LUTs) for the implementation of combinatorial logic expressions. These devices are small blocks of memory commonly used for the implementation of combinatorial logic expressions (CLEs). This

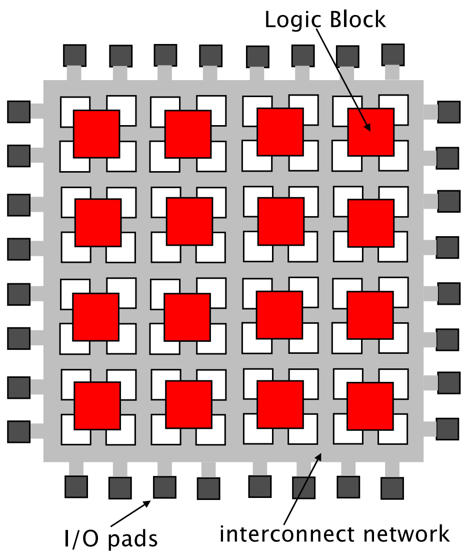


Figure 3: FPGA Architecture

is achieved by storing the truth table of the function in a programmable memory and connecting the address lines of the memory directly to the inputs of the combinatorial logic function. Based on the value presented at the inputs, the corresponding output is selected from the memory and presented at the output of the LUT. LUTs are usually described in terms of the number of inputs (k) to the block and as such will be referred to as k-LUTs for the remainder of this document.

As the LUT is required to store the entire truth table of the function it is required to have 2^k memory bits for a k input logic function. This results in exponential growth in relation to the number of inputs and makes this method unsuitable for combinatorial logic functions with large numbers of inputs. For this reason the number of inputs to a LUT is usually limited to avoid the need for very large memory blocks. Previous work has shown that the use of LUTS with inputs in the range $3 \leq k \leq 5$ produces FPGAs with the best density and speed characteristics[2, 40] although recent work has suggested that larger LUTs may be suitable [27].

As LUTS store the entire truth-table of a k input logic expression it can be seen that the device is capable of implementing any combinatorial logic expression of k-

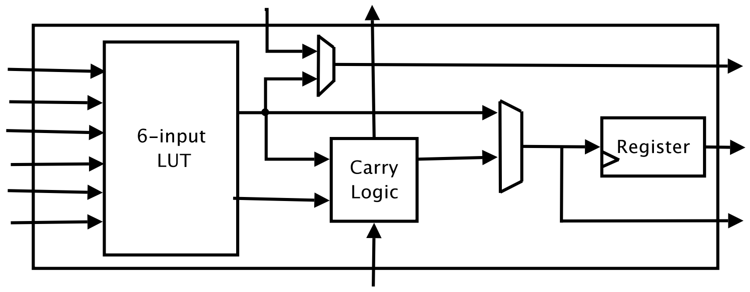


Figure 4: Logic Block of Xilinx Virtex-5 [4]

inputs. In order to extend the usefulness of these blocks, additional hardware, such as registers and carry logic, are implemented alongside the LUTs in the logic blocks contained within the FPGA fabric. The logic block used in the Xilinx Virtex-5 is shown in figure 4 [4]. As this shows, the basic logic block of this device is based on a 6-LUT with additional carry-logic for the efficient implementation of mathematical functions and a register to allow the implementation of sequential circuits. The desired mode of operation can then be selected by programming the multiplexor's within the logic block. In the case of the Virtex-5 it can be seen that a 6-input LUT is utilised rather than the 4-LUT traditionally employed. The LUT used in the logic block of the Virtex-5 has larger numbers of inputs as it is capable of being partitioned to allow it to implement multiple functions with less than 6 inputs in total, for example the 6-LUT can be partitioned in such a way as to act as 2 3-LUTs or a 2-LUT and a 4-LUT simultaneously. This allows for the use of more dense logic in creating the 6-input LUT but with the ability to use the most appropriate number of inputs for the application.

2.3.2 Interconnect Network

The interconnect network of an FPGA is responsible for the routing of signals and connection of logic blocks to implement large functions or even complete systems.

This is achieved through the use of switch blocks programmed to connect the inputs of the logic blocks to the I/O pads and to other logic blocks to allow larger functions to be implemented than is possible using a single logic block.

The interconnect network introduces a large overhead due to the generality needed to allow systems to be implemented on the FPGA. In [22] it is reported that up to 90% of the chip area is taken up by the interconnect network and the hardware required to program this resource and the logic blocks. The interconnect network also dictates the maximum speed of the device due to delays introduced by the channels used for routing and the programmable elements used to route internal signals to the logic blocks. For this reason much research has been undertaken on the optimisation of the structure of the interconnect network including [25].

One of the primary methods of optimisation is to locally group logic blocks into clusters [2]. This allows fast local interconnects to be used, reducing the delay introduced by long tracks on the FPGA. Clusters are then used to implement medium sized sub-functions that can then be connected to other sub-functions to implement entire systems. It is also common to attempt to group related sub-functions that are closely coupled into adjacent clusters to reduce the length of the interconnects required and hence reduce the delay of these connections. This optimisation is usually automatically carried out by synthesis tools such as [37] and is similar to place and route performed during ASIC design.

Place and route for ASIC design is primarily concerned with the placement of hardware and interconnects on a silicon chip to ensure that correct timing is achieved for the device. For FPGAs, the place and route software is concerned with producing the bit-files required to program the interconnect network to connect the logic blocks in order to implement the desired behaviour. One of the main issues involved in place and route for FPGAs is the avoidance of congestion. This is caused when insufficient routing channels are available to route signals within the FPGA to the required locations. This is often caused if a sub-function is required to communicate with large numbers of other sub-functions and hence requires a very large number of routing channels.

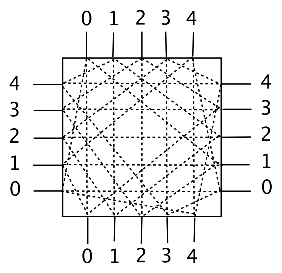


Figure 5: Example of Wilton switch block [29]

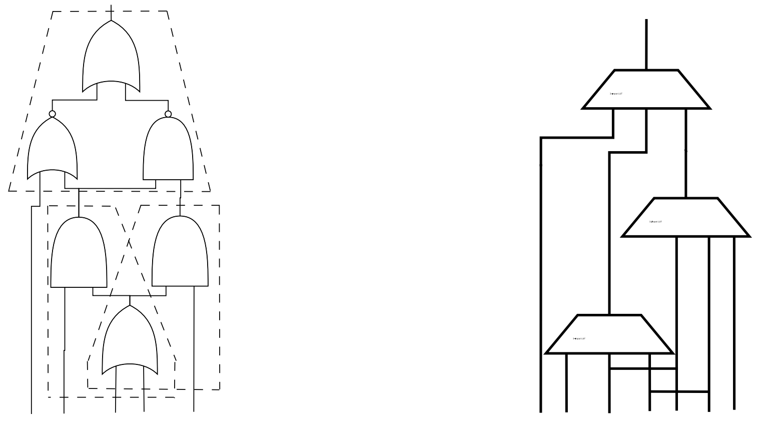
The interconnect network is based on a variety of length channels used to carry signals around the FPGA and switch blocks used to connect these channels and the channels to the inputs/outputs of the logic blocks. As the switch blocks introduce delays the optimisation of these blocks has been researched in great detail and as such a large volume of material, including [29], is available on this subject. An example of a modern switch block is shown in figure 5. As this shows, due to the island style architecture used, the channels are usually categorised as being either vertical or horizontal routing channels. The switch blocks then connect these channels to route signals to the required logic-block.

In order to allow systems to be implemented using $k - LUTs$ it is necessary to first break these systems down into a collection of interconnected sub-functions that are realisable using these blocks. The process of taking large systems and breaking them down into realisable sub-functions is known as decomposition and is discussed in the following section.

2.3.3 Decomposition

Decomposition takes logic functions with large numbers of inputs (i) and attempts to implement these functions as a network of sub-functions where each sub-function has a maximum of k-inputs Decomposition is at the heart of modern synthesis as it allows the implementation of large systems using networks of small hardware components and as such a large number of algorithms exists that automate this process including Flowmap [14], and DAG-map [11].

Decomposition is achieved by making use of the fact that any logic function can be implemented using simple two input gates. Figure 6(a) shows a 5 input logic function implementation using only 2 input gates. The decomposition algorithms attempt to group these gates together while ensuring that no single group requires more than k-inputs as is shown in figure 6(a). Figure 6(b) shows this covering for a k value of 3.



(a) 5-input CLE (b) Decomposed CLE for with k=3

Figure 6: Decomposition of 5-input CLE to network of 3-input LUTs

This covering results in a network of 3 input LUTS that implements the function in figure 6(a). In this way it is possible to use LUTs with fairly small k values to implement large combinatorial logic functions. During decomposition one of the key parameters that must be considered is the depth (d) of the resulting array. This is the longest path from the inputs to the output of the array. The depth is calculated by counting the number of logic elements a signal must pass through on its way from the primary inputs to the output of the system. As synchronous memory elements are used to implement the LUTs it can be seen that a clock cycle will be required to propagate a signal through each logic block and hence the latency of the device will be equal to the maximum path length, or depth, of the network produced by decomposition.

In the example shown in figure 6(b) the depth of the decomposed network is 2. This is a key parameter as it dictates the latency of the network and as such has a direct effect on the speed of operation. For this reason a large number [14, 11, 15] of the decomposition algorithms are primarily concerned with producing networks with the minimum depth possible.

3 Investigation of optimal number of product terms for PLAs

As the future applications of PLAs can not be predicted at design time, it is difficult to determine the amount of hardware that will be required by the end user of the device. When creating new PLA designs it is important to ensure that the devices can implement as broad a range of circuits as possible as this increases the application domain of the device.

Although as shown it is possible to determine the absolute limit to the number of product terms in relation to the number of inputs, only a relatively few CLEs will require this number of products. This makes this measure of little use when determining the characteristics of future PLAs. Of more interest to device designers is the mean number of product terms as this gives an indication of the average number of products in an i-input expression, or the number of product terms required by a PLA to implement 50% of all possible i-input expressions.

As the limit to the number of products is dependent on the number of inputs (i) it is possible to determine the distribution of product terms in CLEs by generating every possible CLE with i inputs but as the total number of i-input CLEs is 2^{2^i}, this makes this form of exhaustive testing impractical for CLEs with even fairly small numbers of inputs.

3.1 Test Strategies

In order to find the distribution of product terms in CLEs a number of strategies used to test the flexibility of new reconfigurable devices were examined. These techniques are shown in figure 7 and range from the use of *real-life* circuits to completely random, *synthetic*, circuits. The following section outlines the previous strategies employed in measuring the flexibility of reconfigurable devices.

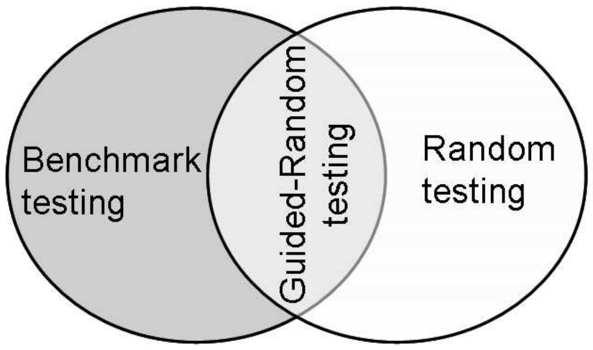


Figure 7: Existing test strategies

3.1.1 Benchmark testing

Traditionally, the method used to select the characteristics of PLAs was the use of benchmark circuits [39, 10]. This method relies on a suite of sample *real-world* circuits that are mapped to PLAs using place and route algorithms such as PLAmap [10]. [41] makes use of the MCNC benchmark circuits to determine the parameters that resulted in the best delay and area characteristics by mapping these circuits to PLAs with varying parameters.

Although these methods are useful in determining the optimal characteristics of PLAs, the results are highly dependent on the use and availability of benchmark circuits. If only a small number of benchmark circuits are available, or if they are closely related, this method may produce results that are particular to the benchmarks and not general applications. As the circuits in the benchmark suites are existing *real-world* circuits it is also unclear how well they will represent future circuits that the device may be required to implement after manufacture and as such, unless the circuits contained in the benchmark suites are constantly updated, they become *out-of-date* rapidly.

3.1.2 Guided-Random testing

In [23, 13] methods of extending the use of benchmarks were suggested. These methods first profile a set of benchmarks to determine ranges for a number of characteristics such as number of inputs, outputs and circuit complexity. Based on this a large number of *synthetic* circuits can be randomly generated with characteristics within these ranges. This allows a very large number of circuits to be generated that can be used for testing.

In [13] these circuits were mapped to sample devices and a measure of flexibility extracted based on the percentage of the total circuits that could be mapped using the place and route tools allowing device designers to rapidly assess the effect design changes have on the flexibility of the device. This method is particularly aimed at domain specific reconfigurable devices where the final application domain of the device is well known and hence the reliability of the results can be assured as it is likely, although not guaranteed, that circuits within a domain will exhibit similar characteristics.

These methods do not completely alleviate the issues relating to the availability of sufficient benchmark circuits as, if the domain is small, it is unlikely that enough example circuits will be available to completely profile the domain. Conversely, if the domain is very large it is likely that the characteristics obtained by profiling will have a large range and the random generation of *synthetic* circuits will result in circuits with almost totally unconstrained characteristics.

3.1.3 Random testing

The use of completely random circuits for device testing was suggested in [18]. This method produces randomly generated netlists used to assess the routability of reconfigurable devices by generating large numbers of circuits and performing place and route. This method was rejected in [23] as it was claimed that the results of the unconstrained random generation did not produce *realistic* circuits. It is felt by the author that the random generation process produced unrealistic circuits as high level optimization was not carried out on the circuits produced. As we have seen, boolean simplification can greatly reduce the amount of hardware required to implement CLEs and as such should be performed before random circuits are used for device testing.

3.1.4 Investigation of average products in CLEs

A number of research projects [19, 36, 34, 35, 33, 5] have attempted to calculate the average number of product terms in CLEs and the upper and lower bounds for minimised SOP expressions. In [34, 35] randomly generated logic expressions are used to calculate the average number of product terms in the expressions and the upper and lower bounds on this are calculated mathematically in a similar way to [5]. In both of these examples the value for the average number of product terms is calculated using randomly generated logic circuits. Although this method seems to generate values in the expected range for the average case behaviour it is impossible to verify the accuracy of the random circuit generation method as no details are given of the operation of this method. These methods also focus solely on the use of this method in determining the characteristics of PLAs and no comparable work has been found for the use of memory based devices such as those based on LUTs.

3.2 Stochastic investigation of number of Product terms in simplified SOP expressions

As stated in section 3 it is impractical to generate every possible CLE of i inputs to determine the distribution of number of product terms due to the exponential growth in the number of possible circuits. An alternative to the full testing method is to generate large numbers of random logic expression that can be used to perform profiling.

In order to further limit the domain it was decided that only circuits with a single output should be considered. This avoids the need to consider situations where product terms may be shared in multi-output circuits as it is felt that product sharing would be minimal and a multi-output circuit can be modeled using multiple single output circuits. Instead a method of generating completely random expression is required to allow the distribution to characterise. The selected method involves the random generation of CLEs based on the truth-table representation of the expressions. By generating circuits at this level high-level optimisation, such as boolean minimisation, can be performed to allow these circuits to more closely resemble *real-*

world circuits.

Although the use of randomly generated CLEs allows the investigation of the number of product terms in general CLEs and the distribution of these values, it requires the generation and simplification of large numbers of CLEs. As the number of inputs to the CLEs increases the time required to perform simplification increases. In the test runs undertaken 100,000 circuits were used for each value of i to determine the distribution of product terms. For values of i greater than 12 this results in test runs that take days to complete. This means that the use of this method is limited to fairly small numbers of inputs. For this reason a more generic mathematical based approach was developed in conjunction with Dr Paul Jackson at Edinburgh University.

The following section details both the Monte-Carlo based generation of random circuits and the mathematical investigation of the average case behaviour for simplified SOP expressions. The results of these methods are then presented and compared to those presented in [35] to prove the accuracy of these results in calculating the average number of product terms in the expressions. The results obtained for the distributions can then be used by device designers to select the required PLA characteristics based on the number of inputs and the desired degree of flexibility.

3.2.1 High-level Monte-Carlo based Circuit generation for PLA testing

Although in [23] it was suggested the random generation of circuits produced results that did not match *real* circuits, this is due to the fact that high-level simplification, such as boolean minimization, was not carried out. For this reason it was determined that circuits should be generated at a high enough level to allow simplification to take place. The *pla* format [1] allows combinatorial logic circuits to be represented at a high level in a format similar to the truth table.

By generating random logic expressions at a high level and performing high level optimization, such as boolean minimization, it is hoped *realistic* logic circuits can be generated that can then be used for device testing or to measure the flexibility of reconfigurable devices. If large numbers of these circuits can be generated it is thus possible to use these circuits to assess the flexibility of the PLA, similarly to the method suggested in [13], by calculating the percentage of circuits that the device can implement. In this way device designers can determine the effect their choice of parameters has on the range of circuits the device is capable of implementing.

It is also possible to turn this concept around by generating large numbers of circuits and calculating the characteristics of the PLA required to implement them. This method allows device designers to select the flexibility required by the device and obtain the characteristics required to achieve this value. The following section details the high-level Monte-Carlo based approach to generating test circuits for PLAs and how this can be used to determine a measure of flexibility of the device.

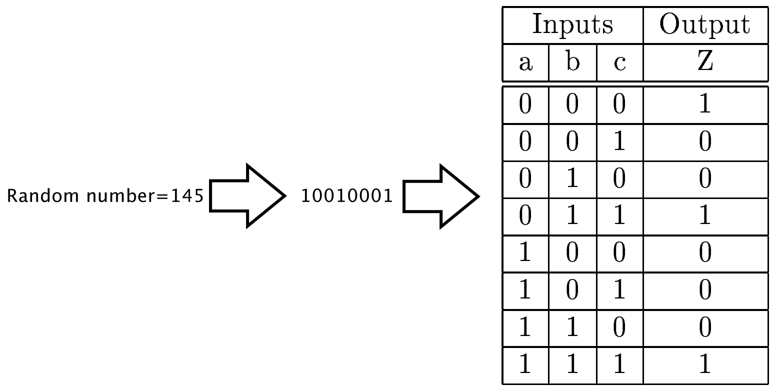
3.2.2 Random Circuit Generation

If the example shown in table 1 is again revisited, it can be seen that the behaviour of the CLE is characterised by the sequence of on (1) and off (0) conditions in the output column of this table. If this sequence is considered simply as a binary sequence it can be seen that a random expression can be produced by a random sequence of 2^i binary digits. This could be produced by simply generating 2^i random binary numbers but as i increases this becomes impractical due to the time required to produce this number of random values.

By making use of a random number generator, such as the Mersenne twister [30], it is possible to generate a random function by making use of the binary representation of a random number as shown in figure 8. Although this method will produce *unrealistic* functions with large amounts of redundant hardware, this can then be removed through boolean simplification to produce more *realistic* functions.

As the random number generator is only capable of generating numbers with a fixed number of binary digits it is necessary to use multiple random numbers to generate truth tables for expressions with large numbers of inputs. The random number generator utilized produces 16-bit random numbers that are capable of producing the necessary bit sequence for expressions of $i \leq 4$, for larger expressions 2^{i-4} random numbers are required.

In order to make *realistic* circuits from the expressions produced it is necessary to perform high-level optimization of these circuits. The SIS synthesis tool [6] was selected to perform simplification using the minimization algorithms originally developed for the Espresso minimization tool [20]. This results in simplified expressions similar to those found in the MCNC, and similar, benchmark suites.



Inputs			Output
a	b	c	Z
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Figure 8: Production of 3-input CLE using 8 bit random number

3.2.3 PLA Test Strategy

Using the method presented in algorithm 1, a large number of i input expressions are generated and the number of product terms in the simplified expressions collected. The number of inputs is then varied over the range *input-range* and the results obtained for each of the random circuits produced. The mean and standard deviation of these results is then collected. To allow the mean and standard deviation of the product term to be compared, the results were normalized by dividing the vales by 2^i over the range of inputs.

Algorithm 1 PLA test strategy

```
k=digits in random number
for each i in input_range do
  for each s in samples do
    for each j in (i-k) do
      random[i]=rand
    construct_function(random)
    simplify
    count_product_terms[s]
  store_results[i](gather_data)
write_log
```

3.2.4 Results obtained using Monte-Carlo based circuit generator

Based on the method presented above the mean number of product terms in simplified CLEs were found for the range $2 \leq i \leq 15$. The results are presented in figure 9. The graph shows that, as expected, the mean values are around $2^{(i-2)}$ but it can be seen that as i increases the mean number of product terms in the simplified expressions begins to fall away from this value. This is due to the large number of literals in the expressions allowing for further simplification of the expressions using methods such as variable reordering.

The mean and standard deviation of the product terms in CLE expressions for $2 \leq i \leq 15$ are shown in table 3. The results are also shown as normalised values by diving them by 2^i. This allows the values obtained to be compared when viewed as the number of product terms compared to the maximum number of terms possible. These results demonstrate that although the mean and standard deviation increase with increasing i the relationship is not direct. The mean values presented in table

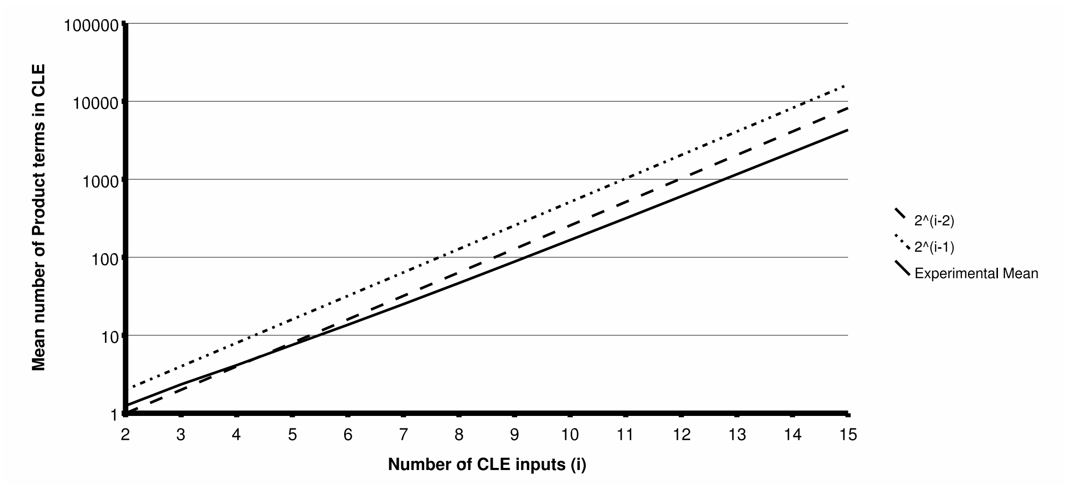


Figure 9: Graphical representation of Mean Product terms in CLE expressions.

3 indicate the number of p terms required to implement 50% of all possible i input, single output, expressions. Thus a PLA produced with the mean number of p terms for i inputs can implement all possible expressions with less than i inputs as $2^{((i+1)-2)} = 2^{(i-1)}$.

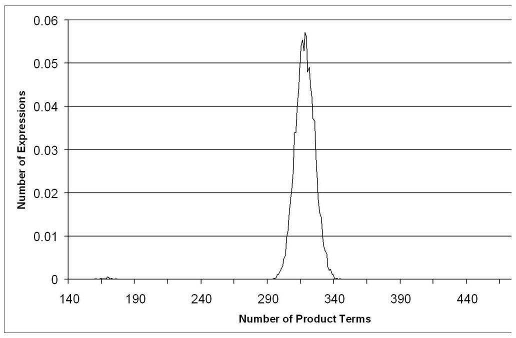
If the normalized mean values are considered it can be seen that the mean as a proportion of the maximum number of p terms drops as i increases. This means area savings can be made by using PLAs with larger i values rather than several smaller PLAs to implement logic expressions while ensuring the same percentage of the total number of expressions can still be implemented.

The results show that as the number of inputs in the CLE is increased the mean number of product terms decreases in proportion to the maximum number of product terms in simplified CLEs. This is due to the increased possibility of simplification taking place due to the larger number of product terms in the unsimplified expressions. This suggests that using PLAs with larger numbers of inputs results in PLAs that require a smaller proportion of the number of products while being able to

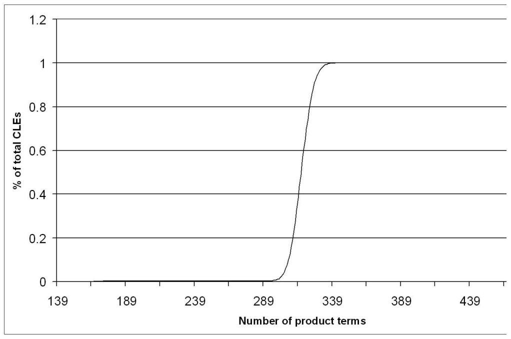
inputs	mean	$\frac{mean}{2^i}$	stddev	$\frac{stddev}{2^i}$
2	1.25	0.3125	0.6614	0.1654
3	2.35	0.29375	0.8061	0.1008
4	4.14	0.26875	0.9289	0.0581
5	7.53	0.23531	1.2333	0.0385
6	13.64	0.21311	1.6082	0.0251
7	25.1	0.19609	2.1710	0.0170
8	46.8	0.18280	2.9204	0.0114
9	88.04	0.17194	3.8999	0.0076
10	166.65	0.16274	5.2562	0.0051
11	316.35	0.15447	7.1134	0.0035
12	605.1	0.14773	9.7183	0.0024
13	1159.92	0.14159	13.2183	0.0016
14	2230.23	0.13612	17.9631	0.0011
15	4299.15	0.1312	24.8827	0.0008

Table 3: Experimental results of Stochastic investigation of Product Terms in CLEs

implement the same proportion of the total number of possible expression.



(a) PDF



(b) CDF

Figure 10: PDF and CDF of product terms in 11-input CLE

Figure 10(a) and 10(b) show the Probability Distribution Function (PDF) and the Cumulative Distribution Function (CDF) respectively for the number of product terms in CLEs with 11 inputs using the method suggested. The PDF shows the distribution of product terms in simplified expressions for a fixed number of inputs, in this case $i = 11$. In this case the number of expressions is normalised by dividing the number of expressions by the total number of possible expressions, in this case $2^{2^{11}}$. The CDF indicates what percentage of all possible 11 input expressions that can be implemented using a PLA with a given number of product terms. From this data it is possible to select the number of product terms to be placed on an 11 input PLA to allow the maximum number of possible expressions to be implemented. Although as stated the number of product terms required to implement every possible expression is 2^{10}, from the distributions it can be seen that very few possible expressions will require this.

From the graphs it can be seen that an 11 input PLA with 343 product terms is capable of implementing 99% of all possible CLEs. Comparing this to a PLA designed using the theoretical maximum value of 1024 products, would result in a saving of 681 product terms while allowing for all but 1% of all CLEs to be implemented.

3.3 Mathematical investigation of product terms in random CLE

This section introduces an algorithm for the production of a minimal cover of a single output Boolean function and calculates its average case behaviour developed in conjunction with Dr Paul Jackson at Edinburgh University. While the algorithm does not produce a minimum cover, the average number of cubes in the cover for randomly chosen functions is observed to be within 4% of that produced by the simplification algorithms used in SIS synthesis tool. The section begins by listing the definitions commonly used before presenting the minimisation algorithm and finally presents the investigation of the average case behaviour.

3.3.1 Definitions

- We use notation $\overline{x}$ for the negation (complement) of a boolean-valued variable x, x + y for the sum (conjunction) of variables x and y, and x.y or simply xy for the product (conjunction) of x and y.
- Let $B = \{0, 1\}$ and B^n be the n-dimensional space of Boolean vectors of length n. Let B_n be the set of Boolean functions $B^n \rightarrow B$. We allow n to be 0, in which case B^0 is a single point space, and B_0 contains just the constant-valued functions.
- When describing functions in B_n we typically associate a variable with each

dimension of the domain B^n. For example, for some f in B_3, when we write f(x,y,z) = x.y + z, we are associating x with the 1st dimension, y with the 2nd and z with the 3rd.

- If we have a vector $x = [x1, \ldots, xn]$, the vector $x - xi$ is the vector $[x1, \ldots x(i-1), x(i+1), \ldots, xn]$.
- A *literal* is a variable or the negation of a variable.
- A *monomial* is a product (conjunction) of literals. If the variables in a monomial are drawn from those associated with the dimensions of an n-dimensional Boolean space, the monomial can be regarded as a representation of a function in B_n, and also as the subset of B^n that this function maps to 1. This subset forms a sub-cube of B^n and hence we often refer to a monomial as a *sub-cube* or simply a *cube*. A monomial can have no literals, in which case it represents the constant 1 function.
- A *polynomial* is a sum (disjunction) of monomials. Any polynomial can be regarded as representing a function in B_n and all functions in B_n can be represented by a polynomial. When discussing operations on Boolean functions, it is convenient to use notation as if the functions were represented by polynomials, though other representations are possible. A polynomial can have no monomials in which case it represents the constant 0 function.
- A *cover* of a Boolean function is a set of cubes whose union is the set of input vectors for which the function returns a 1. The sum of the cubes in a cover is one polynomial representation of the function. An empty cover is equivalent to the constant function 0.
- A cover is *minimal* if, when we delete any cube from it, we no longer have a cover.

- The *cofactor* of function $f(x1, \ldots, xn)$ in B_n with respect to xi is the function $f(x1, \ldots, x(i-1), 1, x(i+1), \ldots, xn)$ in b_{n-1}. The cofactor of function $f(x1, \ldots, xn)$ in B_n with respect to $\overline{xi}$ is the function $f(x1, \ldots, x(i-1), 0, x(i+1), \ldots, xn)$ in B_{n-1}.
- A cover is *non-overlapping* when the intersection of every distinct pair of cubes contained in it is empty. Every non-overlapping cover is minimal.
- Two cubes are considered *siblings* if they are of form m.x and m.$\overline{x}$ for some variable x. A cover is *sibling-free* if no two elements of it are siblings.

These definitions, apart from the last two, appear to be standard from the literature on Boolean logic minimisation.

3.3.2 Boolean Minimisation Algorithm

To describe the algorithm, it will be helpful to first define a few types. Consider a vector of variables $x = [x1, \ldots, xn]$ labelling the dimensions of B^n. Elements of the type *BoolFun(x)* are functions in B_n where the ith component of input vectors is referred to using variable xi. Elements of the type *Cover*(x) are sets of cubes over the variables in x. Type *Vars* is the set of vectors of variables. The algorithm is shown in algorithm 2.

The effect of the calculations on lines 10-12 is the same as if $x.Cx + \overline{x}.C\overline{x}$ were computed first and any resulting pairs of sibling were merged to form new cubes.

Lemma: The function getCover*(xs, f)* returns a non-overlapping sibling-free cover for the Boolean function f over the variables *xs*.

Proof: By induction on the recursive structure of the program.

The features of the algorithm such as the use of a Shannon decomposition for recursive calls and the merging of siblings are found in standard Boolean minimization programs such as Espresso-II [6]. However, unlike such approaches, the algorithm here does not attempt first to calculate prime implicants.

Algorithm 2 Boolean minimisation

```
1.  Cover(xs) getCover (Vars xs, BoolFun(x) f) {
2.    if (|xs| == 0) {
3.      if (f() == 1) return {1} };
4.      else return 0; // 0 is the same as the empty set {}.
5.    }
6.    For all x in xs;
7.    // Compute the Shannon decomposition of f wrt x:
      // f = x . fx + $\overline{x}$. f$\overline{x}$
      BoolFun(xs - x) fx = cofactor of f w.r.t. x.;
      BoolFun(xs - x) f$\overline{x}$ = cofactor of f w.r.t. $\overline{x}$;
8.    Cover(xs - x) Cx = getCover(xs - x, fx);
9.    Cover(xs - x) C$\overline{x}$ = getCover(xs - x, f$\overline{x}$);
10.   // Extract cubes common to cofactor covers Cx and C$\overline{x}$ that
      // would become siblings if lifted by product with x and $\overline{x}$
      // respectively.
      Cover(xs - x) N = Cx intersection C$\overline{x}$;
11.   // Reconsider monomials in N as cubes in Cover(xs).
      Cover(xs) NN = (Cover(xs) N) N;
12.   Cover(xs) C = x.(Cx - N) + $\overline{x}$.(C$\overline{x}$ - N) + NN;
13.   return C;
14. }
```

3.3.3 Analysis of average case behaviour

An i-cube for some natural number i is an i dimensional cube. If a monomial has k literals and represents a function in B_n, it is an $(n-k)$-cube.

Let $c(i,n)$ be number of i-cubes in the n-dimensional Boolean space B_n. By considering the regular expressions of form $\{0,1,*\}^n$, each of which denotes the set of points making up a distinct cube, it is easy to show that

$$c(i,n) = \begin{pmatrix} n \\ i \end{pmatrix} 2^{n-i}$$

Let $e(i,n)$ be the expected number of i-cubes in the cover returned by the above algorithm on a randomly-chosen Boolean function in B_n. We give a derivation of

$e(i, n)$ shortly, but first we introduce an auxiliary function.

Consider when execution of *getCover()* reaches line 10 when the function f is in B_n (i.e. the size of the vector of variables x is n). The variable N is set to contain the monomials representing sub-cubes of B^{n-1} common to Cx and $C\overline{x}$ which would become mergeable siblings if they were lifted by product with x or $\overline{x}$ respectively to become cubes of B^n. On line 11, the monomials in variable N are copied to variable NN and re-interpreted as representing sub-cubes of B^n. The sub-cubes in NN are the new ones that would result from merging siblings in $x.Cx + \overline{x}.C\overline{x}$. Let $m(i, n)$ be the expected number of i-cubes in NN, or, equivalently, the expected number of $(i-1)$-cubes in N.

The probability that some given $(i-1)$ cube exists in Cx is

$$\frac{e(i-1, n-1)}{c(i-1, n-1)}$$

This is also the probability that some given $(i-1)$ cube exists in Cx, and hence the probability of a given $(i-1)$ cube existing in both is

$$\left(\frac{e(i-1, n-1)}{c(i-1, n-1)}\right)^2$$

We can multiply probabilities here because we expect no correlation in the distributions of cubes in Cx and $C\overline{x}$. There are $c(i-1, n-1)$ possible sub-cubes in B^{n-1}, and so we multiply by this to get, for $i > 0$, the expected number of new i-dimensional sub-cubes of B^n formed by merging:

$$m(i, n) = \frac{(e(i-1, n-1))^2}{c(i-1, n-1)}$$

The i-cubes in the returned cover come from 3 sources:

1. They were i-cubes in the result Cx of the recursive call that were not merged with i-cubes from $C\overline{x}$. i.e. they were i-cubes in $Cx - N$. We expect there to

be $e(i, n-1) - (m(i+1, n)$ of them.

2. They were i-cubes in the result $C\overline{x}$ of the recursive call that were not merged with i-cubes from Cx. i.e. they were i-cubes in $C\overline{x} - N$. We expect there to be $e(i, n-1) - (m(i+1, n)$ of these too.

3. They are the result of merging sibling $(i-1)$-cubes in $x.Cx + \overline{x}C\overline{x}$. We expect there to be $m(i, n)$ of these.

We therefore have that

$$e(i,n) = 2(e(i,n-1) - (m(i+1,n)) + m(i,n) \ zfor \ 0 < i < n$$

$$e(0,n) = 2(e(0,n-1) - (m(1,n)) \text{ for } n > 0$$

$$e(n,n) = m(n,n) \ for \ n > 0$$

$$e(0,0) = 0.5$$

To calculate the expected number $e(n)$ of cubes in the cover returned by the above algorithm on a randomly-chosen Boolean function in B_n, we sum up over i:

Table 4 shows the calculated values of $\frac{e(n)}{2^n}$ for $0 < n < 9$. We normalise by 2^{n} to allow easy comparison of $e(n)$ for different n.

3.4 Analysis of Results

The results obtained using both the experimental method and the mathematical technique are shown in figure 11. As this graph shows the results for the mathematical method presented previously are very close to 2^{i-2}due to the fact that only very simple boolean simplification is modelled using this technique. Although it can be seen that as $i$increase the calculated value of the mean begins to fall away from this value. The experimental method produces results that are less than 2^{i-2} and the calculated values for $i > 4$. This demonstrates the effectiveness of the simplification algorithms contained within the SIS synthesis system and there ability to remove excess redundant hardware and also shows the inaccuracy of the mathematical method

n	calculated mean	Calculated normalised mean($\frac{e(n)}{2^n}$)
0	0.5	0.5
1	0.75	0.375
2	1.3124	0.3281
3	2.4184	0.3023
4	4.5584	0.2849
5	8.7104	0.2722
6	16.8	0.2625
7	32.588	0.2546
8	63.5136	0.2481
9	124.2624	0.2427
10	243.712	0.2380
11	479.232	0.2340
12	943.7184	0.2304
13	1861.2224	0.2272
14	3676.5696	0.2244
15	7267.9424	0.2218

Table 4: Mean number of product terms in CLE

due to the simplicity of the method suggested. It is suggested that with further work the complexity of this method could be improved by including further simplification methods such as variable reordering but it is unclear as to how this will effect the run time of the calculations required. Instead it is also suggested that the mathematical method can be used to produce fast estimations of the mean product terms in the expressions with the experimental method being used to refine these results.

In order to test the validity of the results for both the analytical and experimental methods presented here the results obtained by these methods are compared to those presented in [33] for the range $4 \leq i \leq 10$. Table 5 shows the results for each of the methods suggested with the results presented in [33]. As this shows the results for the experimental method agree closely with those presented in [33] and the divergence of the methods as demonstrated in figure 11 is also demonstrated. This shows

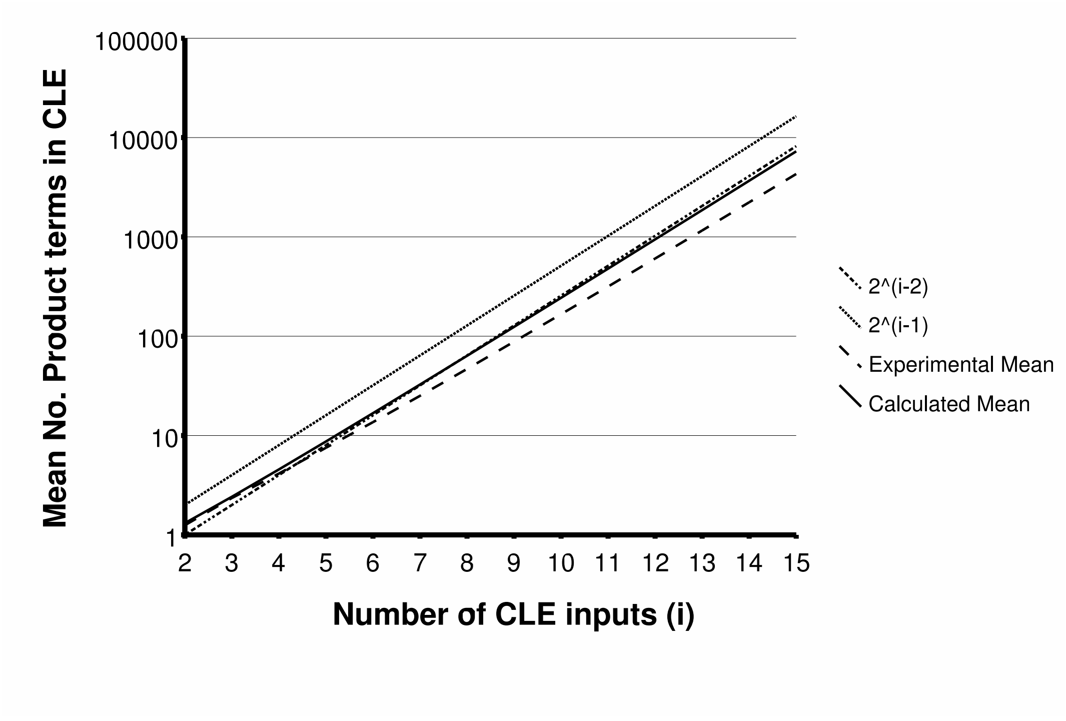


Figure 11: Graphical representation of calculated and experimental mean values.

that although the experimental results agree closely with those in the literature the over simplification of the mathematical method reduces the accuracy of the results obtained.

inputs	Calculated Mean	Experimental mean	mean[33]
4	4.5584	4.14	4
5	8.7104	7.53	6
6	16.8	13.64	13
7	32.588	25.1	24
8	63.5136	46.8	46
9	124.2624	88.04	86
10	243.712	166.65	167

Table 5: Comparison of Analytic and Mathematical results to those presented in [33]

In [39] it is suggested that based on the MCNC benchmark circuits PLAs with $\{12, 9, 3\}$, for small circuits, and $\{12, 18, 3\}$ for large circuits resulted in the best area/delay characteristics. If these PLAs are considered as single output devices and the hardware resources are split evenly between each of the outputs this would result in PLAs with $\{4, 3, 1\}$ and $\{4, 6, 1\}$.

Figure 12 shows the Cumulative Distribution Function (CDF) of the p terms in 4 input CLEs. The values suggested in [39] are plotted on this graph and show that values suggested, 3 and 6, produce PLAs capable of implementing 20% and 97% of circuits respectively. As this suggests a PLA with the lower number of p terms would have sufficient hardware to only implement expressions with low numbers of product terms and as only relatively few of the possible expressions will have this number of product terms will result in a device with a low flexibility score (0.2). Alternatively the use of $p = 6$ means that for the inclusion of relatively few additional product terms the device will have a far greater flexibility (0.97) and thus provide the hardware resource required for a far greater number of applications.

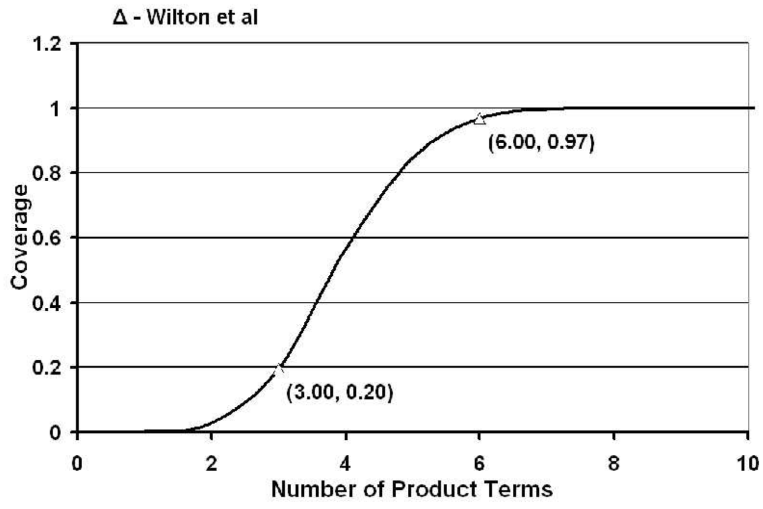


Figure 12: CDF of product terms in 4-input expression

The results of the random generation process thus agree closely with the values selected by [39] . This suggests that the use of benchmark suites for determining the best characteristics of PLAs is not necessary and the random generation and simplification of CLEs results in circuits that are similar to those in the benchmark suite. A large number of circuits can thus be randomly generated for this purpose rather than making use of a limited number of benchmark circuits.

If the mean values are again considered it can be seen that although as discussed in section 2.2.1 the maximum number of product terms is 2^{i-1} the approximate mean number of product terms (p_{mean}) can be calculated using the relation

$$p_{mean} = 2^{i-2} \ for \ i < 10$$

for $i > 10$ this relation does not hold and the mean number of product terms begins to fall as a proportion to the maximum number of product terms. It is felt by the author that this is due to the increased number of literals in the expressions leading to a greater possibility of simplification and optimisation taking place using the advanced simplification algorithms contained in SIS.

3.5 Conclusion

The section presents both an analytical and experimental method of determining the optimal hardware requirements of PLAs to ensure that they can implement as broad a range of designs as possible while avoiding unnecessary, redundant hardware.

The Monte-Carlo based investigation of the number of product terms allows us to determine the distribution of product terms in combinatorial logic expressions. These distributions can then be used to provide a method of determining the number of product terms required by a PLA to ensure that a broad range of CLEs can be implemented. Alternatively, based on the cumulative distribution, it is possible to determine the percentage of the total number of possible expressions of i inputs that a PLA can implement based on the number of inputs and the number of product terms. This can then be used to provide a meaningful comparison between different PLA architectures and a value for representing the flexibility of the device.

The results obtained using the experimental and mathematical techniques were compared to those presented in [33]. Although the experimental technique produces results that are close to those presented in [33] those for the mathematical technique are less accurate and stay very close to 2^{i-2}.

The work carried out in [39] on the selection of PTB parameters demonstrate that the results obtained using the random circuits generated are closely related to those in the benchmark suite. This correlation introduces the possibility of using this method of random circuit generation for the synthetic generation of a large number of circuits that can be used for the determination of parameters or for the testing of architectures. The main issue with this method is the time taken to perform simplification of large CLEs by the SIS decomposition tools. In order to solve this, an algorithm was developed for Boolean minimisation and the mean number of product terms determined in relation to the number of inputs. Although these results diverge from the results obtained using the experimental results they give a good fast approximation for smaller values of i.

4 Investigation of the characteristics of LUT based reconfigurable devices

In a similar way to the design of PLAs it is impossible for the designer to predict in which applications LUT based reconfigurable devices may be utilised. This makes it impossible for device designers to predict how much reconfigurable hardware will be required by the final user of the device. The traditional solution to this problem is simply to place as much hardware as possible on the device with as much flexibility as possible. This is not an ideal solution, particularly for devices intended for certain applications where this extra hardware may increase the power consumption and area of the device. Even in the situation where a large reconfigurable hardware block is produced it is still possible that this block may not have the required resources to implement the functionality required by the user.

As stated in section 2.3.3, decomposition results in an array of interconnected k-LUTs that implement the desired functionality. It is possible that even if a device is used with sufficient numbers of LUTs to implement the required function the resulting architecture may not provide enough flexibility or that congestion may result in an unroutable implementation. This is particularly true for directional architectures such as those suggested in [40] where no feedback is allowed within the fabric of the device. In this case it is possible that the array produced by decomposition requires an array that is of greater depth than that available within the device.

As demonstrated in section 3 it is possible to relate the mean number of product terms in an expression to the the number of inputs (i) to a CLE. As the LUT network produced after decomposition is based on large CLEs it is thus possible to determine the distribution of the number of product terms in the expression. As decomposition is based on breaking this logic expression down into sub-functions of k-inputs it is speculated that in a similar way the numbers of LUTs in the decomposed array and the depth of this array can also be related to the number of inputs in the expression.

This section present the work carried out in find the distribution of the number on k-LUTs and the depth of array produced by the decomposition of an i-input expressions where $i \geq k$. As stated in section 2.2.1 this is also closely related to the idea of creating a measure of flexibility [13] that designers can use when selecting device characteristics for an application where the exact hardware requirements are not know at design time.

	K		
Inputs	3	4	5
4	2.78	1	1
5	6.75	2.99	1
6	14.68	6.98	3
7	30.3	14.99	6
8	61.68	30.99	13
9	124.39	62.97	26
10	250.7	127	53

(a) Mean number of LUTS

	K		
Inputs	3	4	5
4	1.84	1	1
5	3.01	2	1
6	4.01	3	2
7	4.01	3.56	3
8	5	4	3
9	6	5	4
10	7	6	5

(b) Mean network depth

Table 6: Mean number of LUTs and depth of CLE after decomposition

4.1 Experimental Investigation of characteristics of LUT based reconfigurable devices

The method suggested in section 3.2 was again used to generate a large number of CLEs with a fixed numeral of inputs. These expressions were then simplified and decomposition was carried out using a mixture of Roth-Karp decomposition, cube-packing and modified cube extraction contained within the SIS synthesis tool [6]. The number of LUTs and depth of each of the resultant arrays was then collected and based on this the distribution of these factors for an i-input expression was obtained. During the investigation the number of inputs was varied across the range $4 \leq i \leq 10$ and each circuit was decomposed for implementation on LUTs with k values across the range $3 \leq k \leq 5$. For each value of i, 1,000 circuits were generated and decomposed for each of the values of k in the range. The mean number of LUTs and depth of the arrays produced during decomposition are shown in tables 6(a), 6(b) respectively.

Based on these distributions of LUT count and depth it is thus possible for device designers to select the most suitable parameters in order to achieve the required degree of flexibility, or coverage, required for the particular domain of interest for the device.

As the results show the number of LUTs required increases as the number of inputs increases. This would indicate that the area and power of the logic block required to implement the CLEs would also increase at this rate. As the amount of routing required would also increase at this rate it is likely that more complex routing architectures, such as segmented routing or grouping of LUTs into slices with localised routing, would be utilised to reduce the overall size of the device. With this increase in size it is likely that the speed of the device would be reduced as longer interconnect paths would result in large RC delays in the interconnects resulting in reduced clock speeds. As can be seen from the linear increase in the depth of the required arrays the latency would also increase linearly with the number of inputs to the array.

5 Conclusion

This report has presented a method of circuit generation that is capable of generating very large numbers of CLEs that can be used to test the flexibility of reconfigurable devices. Based on the percentage of these circuits that the device can implement a measure of this flexibility can be derived.

This method was used to determine the hardware requirements of reconfigurable devices intended to implement these CLEs based on the number of inputs to the expression. The distribution of product terms in SOP expression was then investigated for use in determining the number of product terms that should be placed on PLAs in order to allow the device to implement the required percentage of all possible i input expressions and hence achieve a selected level of flexibility. Although this work overlaps with previous projects such as [35] the work carried out here extended the results obtained by vastly increasing the numbers of circuits used to determine the mean number of product terms in a CLE expression. By comparing the results obtained using the method here to those in [33] the operation of the random circuit generator can be verified. The results were also compared to those obtained in [39] where *real-world* circuits were used from the MCNC benchmarks to determine the optimal characteristics of PLA based devices. Although direct comparison of these results is difficult the similarities seen indicate that the simplification of randomly produced high-level CLEs produces results similar to those obtained using the benchmark circuits and hence proves that this simplification results in realistic CLEs.

The previous research carried out is primarily concerned with the structure of PLA based device and as such no comparable work has been found on the determination of these parameters for LUT based architectures. Using the pseudo-random circuit generator suggested in section 3.2 the distribution of number of LUTS and depth of arrays produced as a result of decomposition was determined. This can then be used to aid device designers in determining the characteristics of the device produced in order to achieve a given level of flexibility.

References

[1] PLA format description. http://www1.cs.columbia.edu/ cs4861/sis/pla.txt.

[2] Elias Ahmed and Jonathan Rose. The effect of LUT and cluster size on deep-submicron FPGA performance and density. In *FPGA*, pages 3–12, 2000.

[3] Altera. http://www.altera.com.

[4] Altera. Stratix iii fpgas vs. xilinx virtex-5 devices: Architecture and performance comparison. Technical report, Altera, 2006.

[5] E. A Bender and J. T. Butler. On the size of plas required to realize binary and multiple-valued functions. *IEEE Trans. Comput.*, 38(1):82–98, 1989.

[6] Berkeley. SIS download page
. http://embedded.eecs.berkeley.edu/pubs/downloads/sis/index.htm.

[7] Robert King Brayton, Alberto L. Sangiovanni-Vincentelli, Curtis T. McMullen, and Gary D. Hachtel. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, Norwell, MA, USA, 1984.

[8] S. Brown and J. Rose. Architecture of fpgas and cplds: A tutorial, 1996.

[9] Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field-programmable gate arrays.* Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[10] Deming Chen, Jason Cong, Milos D. Ercegovac, and Zhijun Huang. Performance-driven mapping for cpld architectures. In *FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 39–47, New York, NY, USA, 2001. ACM Press.

[11] Kuang-Chien Chen, Jason Cong, Yuzheng Ding, Andrew B. Kahng, and Peter Trajmar. Dag-map: Graph-based fpga technology mapping for delay optimization. *IEEE Des. Test*, 9(3):7–20, 1992.

[12] P. Chow, S. Seo, J. Rose, K. Chung, G. Paez, and I. Rahardja. The design of an sram-based field-programmable gate array. part i: Architecture, 1999.

[13] Katherine Compton and Scott Hauck. Flexibility measurement of domain-specific reconfigurable hardware. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 155–161, New York, NY, USA, 2004. ACM Press.

[14] J. Cong. Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based fpga designs, 1994.

[15] Jason Cong and Yuzheng Ding. Combinational logic synthesis for lut based field programmable gate arrays. *ACM Trans. Des. Autom. Electron. Syst.*, 1(2):145–204, 1996.

[16] Lattice Semiconductor Corporation. Lattice semiconductor corporation. http://www.latticesemi.com.

[17] M. R. Dagenais, V. K. Agarwal, and N. C. Rumin. The mcboole logic minimizer. In *DAC '85: Proceedings of the 22nd ACM/IEEE conference on Design automation*, pages 667–673, New York, NY, USA, 1985. ACM Press.

[18] J. Darnauer and W. W. Dai. A method for generating random circuits and its application to routability measurement. In *FPGA'96*, 1996.

[19] E. Dubrova, D. Miller, and J. Muzio. Upper bounds on the number of products in and-or-xor expansion of logic functions, 1995.

[20] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, 1992.

[21] H. Fleisher and L. I. Maissel. "an introduction to array logic". Technical report, IBM J. Res. Develop, 1975.

[22] R. Hartenstein. Trends in reconfigurable logic and reconfigurable computin. In *Electronics, Circuits and Systems, 2002. 9th International Conference on*, 2002.

[23] M. Hutton, J. P. Grossman, J. Rose, and D. Corneil. Characterization and parameterized random generation of digital circuits. In *ACM/SIGDA Design Automation Conference (DAC)*, 1996.

[24] A. El Gamal J. Rose and A. Sangiovanni-Vincentelli. "architecture of field-programmable gate arrays". *Proceedings IEEE, vol. 81*, 1993.

[25] Noha Kafafi, Kimberly Bozman, and Steven J. E. Wilton. Architectures and algorithms for synthesizable embedded programmable logic cores. In *FPGA '03: Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 3–11, New York, NY, USA, 2003. ACM Press.

[26] M. Karnaugh. A map method for synthesis of combinational logic circuits. *Transactions of the AIEE, Communications and Electronics*, 1953.

[27] David Lewis, Elias Ahmed, Gregg Baeckler, Vaughn Betz, Mark Bourgeault, David Cashman, David Galloway, Mike Hutton, Chris Lane, Andy Lee, Paul Leventis, Sandy Marquardt, Cameron McClintock, Ketan Padalia, Bruce Pedersen, Giles Powell, Boris Ratchev, Srinivas Reddy, Jay Schleicher, Kevin Stevens, Richard Yuan, Richard Cliff, and Jonathan Rose. The stratix ii logic and routing architecture. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 14–20, New York, NY, USA, 2005. ACM Press.

[28] Inc M2000. "m2000 flexeostm configurable ip core". http://www.M2000.fr.

[29] M. Imran Masud and Steven J. E. Wilton. A new switch block for segmented FPGAs. In Patrick Lysaght, James Irvine, and Reiner W. Hartenstein, editors, *Field-Programmable Logic and Applications*, pages 274–281. Springer-Verlag, Berlin, / 1999.

[30] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. In *ACM Trans. on Modeling and Computer Simulations*, 1998.

[31] R. L. Rudell and A. Sangiovanni-Vincentelli. Mulitple-valued minimization for pla optimization. In *IEE Transactions on Computer-Aided Design*, pages 727–750. IEEE, 1987.

[32] Richard L. Rudell. *Logic Synthesis for VLSI Design*. PhD thesis, EECS Department, University of California, Berkeley, 1989.

[33] T. Sasao. A design method for and-or-exor three-level networks, 1995.

[34] T. Sasao and P. Besslich. On the complexity of mod-2 sum pla's. *IEEE Trans. Comput.*, 39(2):262–266, 1990.

[35] Tsutomu Sasao. Bounds on the average number of products in the minimum sum-of-products expressions for multiple-value input two-valued output functions. *IEEE Trans. Comput.*, 40(5):645–651, 1991.

[36] Tustomu Sasao. Multiple-valued logic and optimization of programmable logic arrays. *Computer*, 21(4):71–80, 1988.

[37] Synopsis. Synopsis Galaxy Design Platform. http://www.synopsys.com/.

[38] Xilinx. http://www.xilinx.com.

[39] A. Yan and S.J.E Wilton. Product term embedded synthesizable logic cores. In *IEEE international Conference on Field-Programmable Technology*, 2003.

[40] Andy Chee Wai Yan. Product-term based synthesizable embedded programmable logic cores. Master's thesis, University of British Columbi, 2005.

[41] S. Yang. Logic synthesis and optimization benchmarks user guide version. Technical report, 1991.

Portfolio Document TR3

Development of a Hardwired Directional Reconfigurable Architecture

Author:
Graeme Milligan

Supervisor:
Wim Vanderbauwhede

Contents

List of Figures

1 Introduction

Due to its ability to introduce hardware flexibility, the use of reconfigurable hardware has become more prevalent. Its ability to allow the end user, as opposed to the device designer, to determine its functionality allows the use of standard generic reconfigurable devices in a variety of applications in a similar manner to the use of microprocessors. The design cost of these reconfigurable devices can thus be amortized over a large number of projects and the end user is no longer required to enter a lengthy and expensive ASIC design flow to gain the advantages of using hardware devices.

Of particular interest is the use of reconfigurable hardware in system on chip (SoC) applications. In this role a block of reconfigurable logic would be placed on a standard SoC platform. The end-user can then program this block to implement any functions required that are not already present on the SoC.

The reconfigurable blocks for use in SoCs can be defined as *soft cores*, these are cores that are described at a high level and generated as required. The use of soft cores allows the amount of reconfigurable hardware placed on the SoC to be tailored to the required application. As the blocks are defined at a high level, they must be synthesised before being used in an SoC, this means it is desirable to design these blocks using components from standard cell libraries that are easily synthesised.

Although reconfigurable hardware devices give some of the advantages of the ASIC design flow, such as reduced area and power, and increased operating speeds, the flexibility of these devices introduces an overhead and reduces their efficiency. This overhead is due to the hardware required to provide programmability and the need to make the devices as general as possible to allow their use in a broad range of applications. In order to reduce this overhead a number of projects have aimed to produce reconfigurable devices that are targeted at a particular domain. The devices make use of custom logic blocks, such as multipliers and adders, which are commonly used in the domain of interest. Projects such as Totem [10] and RaPid [14] have been designed to target the multimedia and DSP domains and as such contain many logic functions commonly found. In these projects a set of sample, or benchmark, circuits from the domain of interest are used to profile the domain and develop a list of functions that are commonly used in the domain. Based on this, a domain specific reconfigurable device can be produced that contains these functional blocks. In this way, large custom logic blocks can be included in the reconfigurable device in order reduce the overhead introduced by the flexibility. Although this reduces the overhead of the device it also limits the scope of applications that the device can be usefully employed.

This research aims to produce a custom, domain specific reconfigurable device targeted at implementing combinatorial logic expressions (CLEs) such as those used to determine the next state of a Finite State Machine (FSM). Although it would be possible to perform domain profiling by taking a large number of FSMs and determining the characteristics of the reconfigurable device required to implement them, it is felt that, as the expressions are highly dependent on the structure of the FSM a more general approach should be used. This document details the development of a custom reconfigurable device for the implementation of CLEs and is organised as follows; section 2 gives details of existing reconfigurable devices and the process by which large CLEs can be implemented by these devices; section 3 then examines the implementation of CLEs using reconfigurable device in order to develop a novel, custom architecture tailored for the implementation of these expressions; the architecture suggested is then presented in section 4 along with the place and route tool required to implement CLEs on this device and issues concerning the implementation of the device. This document concludes by presenting areas of future work suggested by this project in section 6 before final conclusions are made.

2 Background

In order to produce a device specifically tailored for the implementation of CLEs it was first necessary to fully understand the devices currently used for such an implementation. For this reason the following section gives details of the architecture of modern FPGAs and two other selected reconfigurable architectures as well as a discussion of the methods used to map CLEs to these devices. It should be noted that the material presented in sub-section 2.1 is repeated in [25] and is included here for completeness.

2.1 FPGA

The architecture of modern FPGAs is usually described as being *island* style, where the islands are programmable logic blocks used to implement the required functionality surrounded by a *sea* of routing used to connect these logic blocks in the desired manner. The typical architecture of an FPGA is shown in figure 1. The logic units are usually constructed of small memory blocks know as Look-Up-Tables that implement combinatorial logic expressions (CLEs). Recently devices have begun to include embedded components such as memories, DSP components and even complete processors in order to increase the efficiency of these device's when implementing larger systems.

Due to the mature nature of these devices a large volume of literature is available including [8, 4, 19]. The following section aims to give an overview of FPGA design and circuit implementation.

2.1.1 Logic blocks

Modern FPGA devices make use of Look-Up-Tables (LUTs) for the implementation of combinatorial logic expressions. These devices are small blocks of memory com-

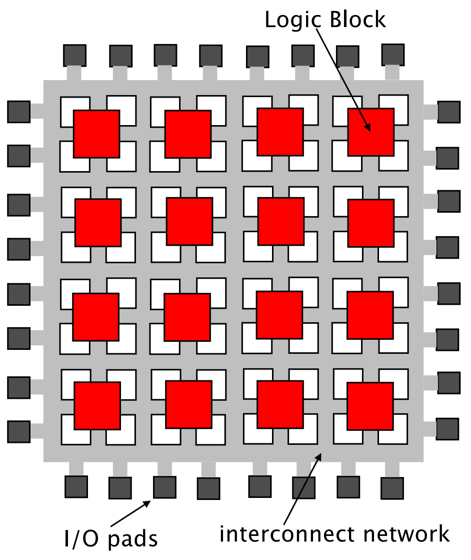


Figure 1: FPGA Architecture

monly used for the implementation of combinatorial logic expressions (CLEs). This is achieved by storing the truth table of the function in a programmable memory and connecting the address lines of the memory directly to the inputs of the combinatorial logic function. Based on the value presented at the inputs, the corresponding output is selected from the memory and presented at the output of the LUT. LUTs are usually described in terms of the number of inputs (k) to the block and as such will be referred to as k-LUTs for the remainder of this document.

As the LUT is required to store the entire truth table of the function it is required to have 2^k memory bits for a k input logic function. This results in exponential growth in relation to the number of inputs and makes this method unsuitable for combinatorial logic functions with large numbers of inputs. For this reason the number of inputs to a LUT is usually limited to avoid the need for very large memory blocks. Previous work has shown that the use of LUTS with inputs in the range $3 \leq k \leq 5$ produces FPGAs with the best density and speed characteristics[1, 33] although recent work has suggested that larger LUTs may be suitable [23].

As LUTS store the entire truth-table of a k input logic expression it can be seen

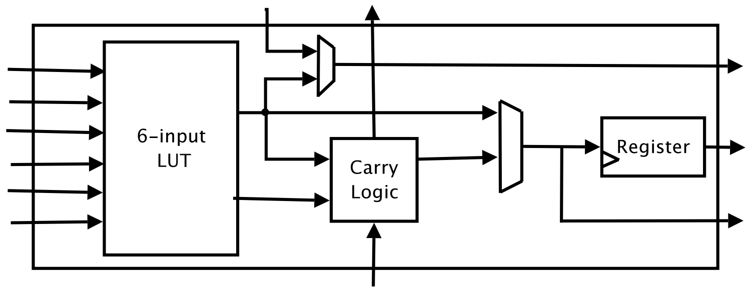


Figure 2: Logic Block of Xilinx Virtex-5 [2]

that the device is capable of implementing any combinatorial logic expression of k-inputs. In order to extend the usefulness of these blocks, additional hardware, such as registers and carry logic, are implemented alongside the LUTs in the logic blocks contained within the FPGA fabric. The logic block used in the Xilinx Virtex-5 is shown in figure 2 [2]. As this shows, the basic logic block of this device is based on a 6-LUT with additional carry-logic for the efficient implementation of mathematical functions and a register to allow the implementation of sequential circuits. The desired mode of operation can then be selected by programming the multiplexor's within the logic block.

2.1.2 Interconnect Network

The interconnect network of an FPGA is responsible for the routing of signals and connection of logic blocks to implement large functions or even complete systems. This is achieved through the use of switch blocks programmed to connect the inputs of the logic blocks to the I/O pads and to other logic blocks to allow larger functions to be implemented than is possible using a single logic block.

The interconnect network introduces a large overhead due to the generality needed to allow systems to be implemented on the FPGA. In [18] it is reported that up to 90% of the chip area is taken up by the interconnect network and the hardware required to program this resource and the logic blocks. The interconnect network also dictates the maximum speed of the device due to delays introduced by the channels used for routing and the programmable elements used to route internal signals to the logic blocks. For this reason much research has been undertaken on the optimisation of the structure of the interconnect network including [20].

One of the primary methods of optimisation is to locally group logic blocks into clusters [1]. This allows fast local interconnects to be used, reducing the delay introduced by long tracks on the FPGA. Clusters are then used to implement medium sized sub-functions that can then be connected to other sub-functions to implement entire systems. It is also common to attempt to group related sub-functions that are closely coupled into adjacent clusters to reduce the length of the interconnects required and hence reduce the delay of these connections. This optimisation is usually automatically carried out by synthesis tools such as [30] and is similar to place and route performed during ASIC design.

Place and route for ASIC design is primarily concerned with the placement of hardware and interconnects on a silicon chip to ensure that correct timing is achieved for the device. For FPGAs, the place and route software is concerned with producing the bit-files required to program the interconnect network to connect the logic blocks in order to implement the desired behaviour. One of the main issues involved in place and route for FPGAs is the avoidance of congestion. This is caused when insufficient routing channels are available to route signals within the FPGA to the required locations. This is often caused if a sub-function is required to communicate with large numbers of other sub-functions and hence require a very large number of routing channels.

The interconnect network is based on a variety of length channels used to carry signals around the FPGA and switch blocks used to connect these channels and the

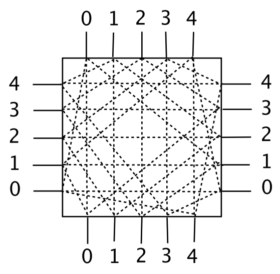


Figure 3: Example of Wilton switch block [24]

channels to the inputs/outputs of the logic blocks. As the switch blocks introduce delays the optimisation of these blocks has been researched in great detail and as such a large volume of material, including [24], is available on this subject. An example of a modern switch block is shown in figure 3. As this shows, due to the island style architecture used, the channels are usually categorised as being either vertical and horizontal routing channels. The switch blocks then connect these channels to route signals to the required logic-block.

Recently there has been much interest in reducing the flexibility of the interconnect network in an effort to reduce the overhead introduced by the generic nature of this component. The two main methods suggested are the use of directional architectures and the use of hard-wired interconnects.

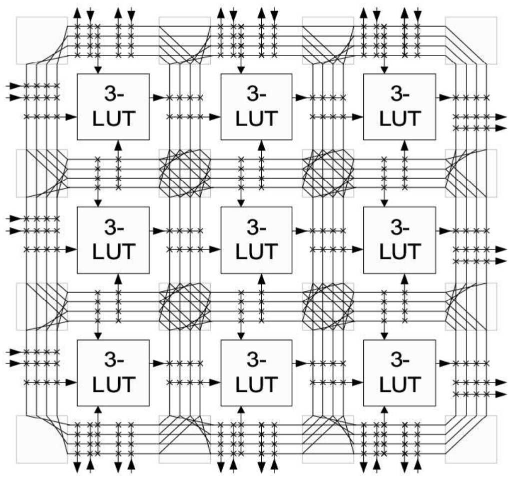
2.2 Directional Architectures

Directional architectures such as those suggested in [31, 20, 32, 21] aim to produce simplified interconnect networks by reducing flexibility within the routing architecture. These projects limit the routing within the switch blocks to a single direction and remove the possibility of implementing feedback within the array. This is acceptable for the implementation of multi-level combinatorial logic as feedback is not required as long as the array is deep enough to implement the desired function. This research initially focused on these blocks to allow for the production of synthesizable embedded programmable logic cores where the possibility of feedback within the device would cause these devices to be unsynthesizable using conventional synthesis tools [31].

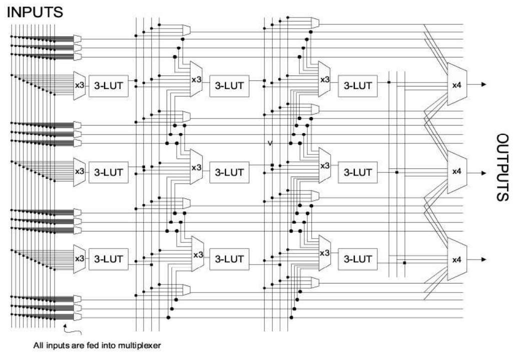
The directional and gradual architectures presented in [32, 20] are shown in figure 4(a)4(b). Although the directional architecture, shown in figure 4(a), is based on the island style common to FPGAs the use of a simplified switch block similar to that suggested in [21] means signals can only be routed in a single direction i.e. *left to right.* This technique is more clearly demonstrated in the gradual architecture shown in figure 4(b), limiting the flexibility of the architecture to only allow signals to be routed in a single direction (again, *left to right*) allows the switch blocks to be replaced by simple multiplexors. The horizontal routing channels increase in width from the left to the right and the vertical channels are only accessible through the output of the LUTs in the array. The outputs are produced by a combination of the outputs of the LUTs in the final column and those routed from previous columns using the horizontal routing channels. This architecture was further extended to the use of product term blocks (PTBs) [32] and resulted in the architecture shown in figure 4(c). As this shows the number of PTBs in each level reduces by a fixed factor at each new level and the routing is again limited to a single direction within the array.

2.2.1 Hard-wire connection based architectures

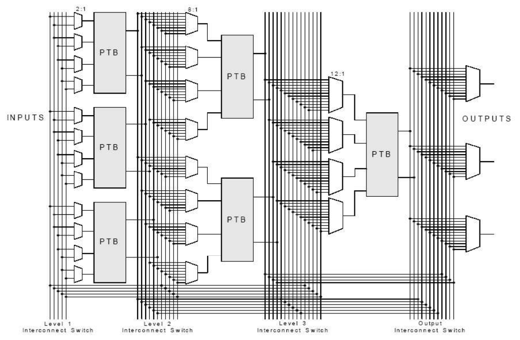
The use of the switch blocks to connect routing channels within the interconnect network results in increased circuit delay [22], area and power [17]. In order to reduce this overhead the use of hard-wired interconnects is suggested in [9, 29]. These architectures replace some of the programmable interconnects within the interconnect network with hard-wired connections, these are metal wires with very low delay and power characteristics. This is similar to the use of segmented routing channels where channels spanning multiple columns or rows are used to limit the number of switches



(a) Directional Architecture



(b) Gradual Architecture


INPUTS
PTB
OUTPUTS
2:1
8:1
12:1
Level 1 Interconnect Switch
Level 2 Interconnect Switch
Level 3 Interconnect Switch
Output Interconnect Switch

(c) PTB Architecture

Figure 4: Directional arrays [20, 32]

a signal must pass through to reach the desired location. Hard-wired interconnects take this one stage further by connecting the outputs of selected LUTs directly to the input of other LUTs in the array.

In [29] optimised switch blocks are used that replace some of the programmable connections with hardwired connections based on the implementation of *rectilinear Steiner trees.* This allows common interconnect configurations to be implemented without the need to program the switch blocks and avoids the introduction of the delays associated with these components.

One of the main issues with this solution is the mapping of functions to arrays containing hard-wired interconnects. [9] attempts to achieve this mapping by identifying the critical path in the array and maps this to the hard-wire interconnects in an effort to reduce the delay introduced by the interconnect network.

In order to allow systems to be implemented using $k - LUTs$ it is necessary to first break these systems down into a collection of interconnected sub-functions that are realisable using these blocks. The process of taking large systems and breaking them down into realisable sub-functions is known as decomposition and is discussed in the following section

2.3 Decomposition

Decomposition is the process by which logic functions with large numbers of inputs (i) are implemented as a network of sub-functions where each sub-function has a maximum of k-inputs and is based on the theory of combinational logic synthesis. Decomposition is at the heart of modern synthesis as it allows the implementation of large systems using networks of small hardware components and as such a large number of algorithms exists that automate this process including [11, 16, 26, 7].

Combinational logic synthesis for FPGAs is concerned with producing a representation of a logic function that can be implemented using k-LUTs. This is usually achieved by making use of a two stage process; first the function undergoes logic optimisation which transforms a gate-level network into another network that is more suitable for implementation [12]. This is then followed by technology mapping where the boolean network is covered using k-LUTs to obtain a K-LUT network that is functionally equivalent to the original network. The following sections give details of the process of decomposition.

2.3.1 Problem Formulation

In order to fully understand the process of decomposition it is useful to begin by defining some of the terminology used.

- A Boolean network is represented by a Directed Acyclic Graph (DAG) where nodes represent logic gates.
- The edge (i,j) exists if the output of node i is connected to the input of node j.
- A Primary Input (PI) has no incoming edge, and
- A Primary Output (PO) has no outgoing edge.
- input(v) represents a set of fanins to gate v.
- For a sub-graph H of the network, input(H) represents all nodes supplying inputs to gates in H.
- A node is k-feasible if $|input(v)| \leq k$.
- A sub-network H is k-feasible if $|input(H)| \leq k$.
- The fanouts of node v are denoted output(v) and similarly the set of fanouts of subnetwork H are denoted output(H).
- A node is considered *fanout-free if* $|output(v)| \leq 1$ and
- if every non-PI node in a network is *fanout-free* it is called a leaf-DAG,
- if every node is *fanout-free,* including the PIs, the network is a tree or a forest if there are multiple POs.
- The level (l) of a node v is the longest path from the PI to v.
- The depth (d) of a network is the largest level in the network.

There are also a number of substructures in a network that are useful when performing decomposition.

- A cone of node v, denoted C_v, is a subnetwork of N that contains v and some of its non PI predecessors such that every path remains within C_v where v is termed the root of the tree
- A maximum cone, MC_v, is a cone containing all non-PI predecessors of v and,
- the fanin network, N_v, is a maximum cone that also contains the PI predecessors.
- A *fanout-free* cone (FFC) is one where all of the node outputs remain within the cone.
- A *maximum fanout-free cone* ($MFFC_V$) contains the set of all FFCs rooted at v.

These are standard definitions that appear in a number of texts including [12, 7]. Further definitions can also be found in these texts but those presented here are sufficient to understand the basic concepts covered in this document.

2.3.2 Logic Optimisation

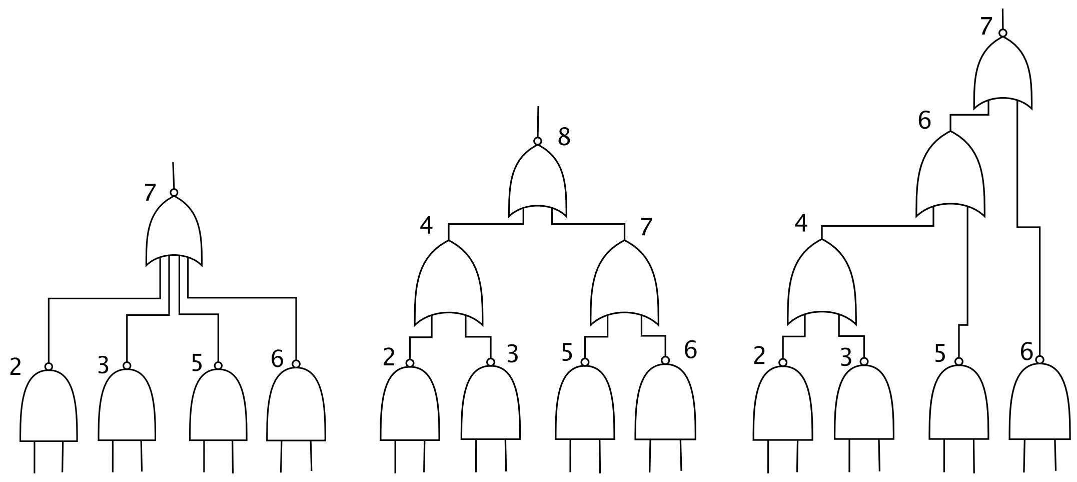
Logic optimisation is the process of transforming a network into an equivalent network that is more suitable for technology mapping. This involves producing a representation of the network that has a valid k-LUT mapping solution.

2.3.2.1 Gate Simplification The first stage in producing a k-mappable implementation of the network is to replace any nodes that represent complex gates with simple gates (AND, OR, NAND, NOR). This can be achieved by tools such a SIS [28] using the tech-decomp algorithm [13]. This algorithm transforms complex gates, using balanced tree decomposition [3, 27], into a collection of simple gates that implement the equivalent functionality. It should be noted that this will often involve the insertion of a sub-network that may increase the depth of the network.

2.3.2.2 Node Decomposition The next stage in the decomposition is node decomposition. This process re-expresses a node function by a logically equivalent collect of nodes. This is achieved by smashing the boolean network to produce a network containing only two-input logic gates. This ensures that every gate in the network is k-feasible and also the use of small gates allows the mapping tools more freedom when mapping the gates to LUTs at later stages in the algorithm.

The most straightforward method of achieving this is to replace each large gate by a balanced tree as shown in figure 5(b). In this example the 4-input gate in figure 5(a) has been replaced by a network of 2 input gates. It can be seen that this solution increases the depth of the network by 1. For the general case it is reported in [6] that for a network of depth d, balanced tree decomposition may increase the depth to $dlog(d)$.

A number of alternative node decomposition algorithms are available including logical decomposition methods such as Huffman tree decomposition [7] and bin-packing decomposition [15] and symbolic decomposition methods such as OBDD based extraction [5]. The DMIG algorithm [7] operates in a similar way to balanced tree decomposition but aims to produce depth optimal results. As the labels in figure 5 show the depth of the resultant network is the same as in the original network using this method. This algorithm was selected to perform node decomposition due to its efficiency and availability in the SIS synthesis tool [28].



(a) Boolean network containing 4-input gate

(b) Transform using balance tree

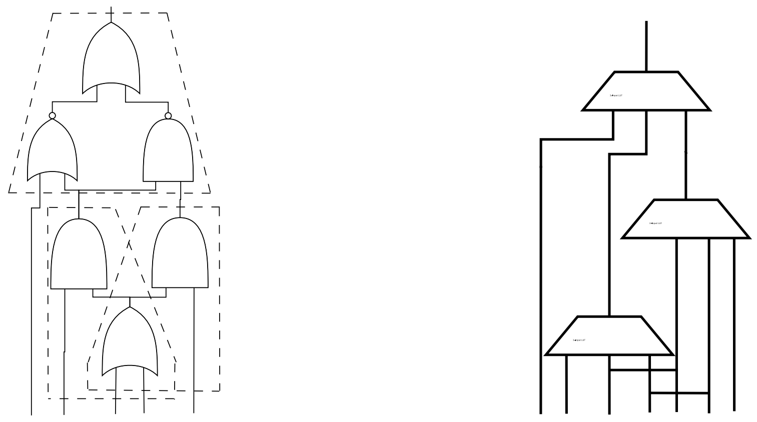
(c) Transform using Dmig

Figure 5: Transforming multi-input network to two-input network

2.3.3 Technology mapping

After the production of a Boolean network based on simple two-input gates using the methods described previously, technology mapping can be carried out. Technology mapping aims to create a cover of the network such that any new node in the network satisfies the condition $|inputs(v)| \leq k$. The decomposition algorithms attempt to group nodes together while ensuring no single grouping requires more than k-inputs as is shown in figure 6(a). Figure 6(b) shows this covering for a k value of 3.

This covering results in a network of 3-LUTS that implements the function in figure 6(a). In this way it is possible to use LUTs with fairly small k values to implement large combinatorial logic functions. During decomposition one of the key parameters that must be considered is the depth (d) of the resulting array. This is the longest path from the inputs to the output of the array. The depth is calculated by counting the number of logic elements a signal must pass through on its way from the primary inputs to the primary outputs. For the example shown in figure 6(b) the



(a) 5-input CLE (b) Decomposed CLE for with k=3

Figure 6: Decomposition of 5-input CLE to network of 3-input LUTs

depth of the decomposed network is 2.

As a constant delay is introduced by each LUT in the path and by the routing required to connect the LUT to its predecessors. As such, if a unit delay is assumed for the LUT and the channels, the depth can directly indicate the delay of the array. It can also been seen that, as synchronous memory elements are used to implement the LUTs, a clock cycle will be required to propagate a signal through each logic block. The latency of the device will thus be equal to the maximum path length, or depth, of the network produced by decomposition. For this reason a large number of the decomposition algorithms are primarily concerned producing networks with the minimum depth possible [11, 7, 12] .

3 Investigation of characteristics of CLEs

If the results of the logic optimisation process shown in section 2.3.2 are again examined it can be seen that the decomposition of large CLEs results in the production of k-bounded networks that are either trees, leaf-DAGs or MFFC. These networks can be shown to be directional in nature, i.e. signals propagate from the PI nodes to the PO nodes, where the number of nodes in subsequent levels decrease by a factor of at least k (otherwise the network would not be k-bounded) and converge on the root of the network. If during the technology mapping stage the LUTs are produced to be *fanout free* it can be seen that the resultant network must be either a tree or leaf-DAG. This can be achieved by duplicating any sub-cones in the network that require $|output(v)| \geq 1$. Although this operation will result in increased numbers of LUTs the regular nature of the network produced allows the use of an optimised reconfigurable device that is tailored to the implementation of tree or leaf-DAG networks. It can also be seen that the need to perform this duplication is very slight as it is reported in [6] that 98.5% of LUTs in LUT based FPGA designs are *fanout-free*. This means that in the majority of cases the results of logic optimisation will be a tree or leaf-DAG where each of the LUTs is *fanout free*.

As discussed in [32] the use of a completely flexible interconnect network in FPGAs is not required for the implementation of Combinatorial logic expressions (CLEs). The use of a fully flexible FPGA interconnect network actually prevents their use as synthesizable *soft-cores* for use in embedded applications such as on SoCs. The high degree of flexibility in fully flexible interconnect networks allows combinatorial logic loops and hence prevents the automatic synthesis of these blocks.

If the sum-of-products implementation of CLEs is considered it is obvious that the routing structure is only required to route signals from the primary inputs to the outputs with no feedback. For this reason [32] suggests a uni-directional architecture for the implementation of CLEs. This architecture, shown in figure 4(c), routes signals from the inputs to the outputs and is based on columns of product term blocks (PTB) rather than the island style adopted by FPGAs. Between each column a simple routing mechanism based on multiplexers routes the signals from one column to the next to implement CLEs.

This research suggests the use of a triangular array for the implementation of

CLEs, with y PTBs in the first level and $n_i = \alpha^{i-1}y$ in the ith level. Based on results obtained using a suite of benchmark circuits it was found $\alpha = 0.5$ produced the best area/delay characteristics. This results in an array where the number of PTBs in a column is half that of the previous column. It should be noted however that this value was calculated for PTBs with a fixed number of inputs, outputs and product terms and the effect of varying these parameters was not considered on the choice of α.

As the interconnect architecture of this device is uni-directional it is essential to ensure that a device with sufficient depth is created i.e. *array depth* $\geq d$. However, even for an array with sufficient depth, this does not guarantee any CLE can be mapped to a device of this type. This is due to the fact it is possible that the number of edges, or inputs, required at any level could be $k.n_i$ and for the array shown in 4(c) only $\frac{n_i}{2}$ edges can be provided by the PTBs in the previous level. This means the remaining edges must be provided by the horizontal routing channels, shown at the bottom of the array, from previous columns or directly from the primary inputs. This may not be possible if all of the PTBs in the previous columns are already in use. This is equivalent to congestion in an FPGA and would result in a function that can not be mapped to this array, reducing the domain that this array can be utilised in. Although there are relatively few circuits that will be affected by this problem, and obviously none of the benchmark circuits used exhibited this, this may cause issues if this device was used in safety critical applications were the flexibility of the device is used to provide error correction or upgradeability.

One possible method of resolving this issue is to use $\alpha = \frac{1}{k}$. This would result in an array where the numbers of programmable elements in the array is reduced by a factor of k at each level. This ensures there will be sufficient numbers of PTBs at each level to feed all of the inputs of the PTBs in the next level with no need to allow signals to be routed from levels other than that immediately previous. Although this would guarantee that any CLE can be mapped to the array, this would result in arrays with larger numbers of programmable elements as the number of elements in each column increases by a factor of k rather than 2 as was suggested.

It can clearly be seen that if an array of this type was to be created, providing the results of decomposition resulted in a *fanout-free* tree or leaf-DAG, the interconnect network could be simplified as the horizontal routing channels are now redundant. Hard-wired connections, like those suggested in [29, 9], could then be used between levels completely removing the area, delay and power overhead introduced by programmable switches or in this case multiplexors.

4 Development of custom reconfigurable architecture

Although in [32] it is suggested that improved performance can be achieved through the use of appropriately sized PTB blocks compared to use of LUTs it was decided that LUTs would be used for architectural investigation due to relative ease of design and use and the availability of these components in the standard cell libraries, although it should be trivial to change the architecture to one based on PTBs at a later date.

If the architecture presented in figure 4(c) is again revisited it can be seen that reducing the number of PTBs in each column by a factor of k, instead of 2 as suggested, allows the interconnect network to be further simplified. As stated the use of hard-wired interconnects allows the inputs of a node to be fed directly from the outputs of k nodes in the previous level. The use of hard-wired interconnects results in the architecture shown in figure 7. This architecture is a regular tree structure that grows by a factor of k, in this case k=3.

Although this array will require larger numbers of LUTs than one using the architecture suggested by [32] it will require less routing due to the removal of the global routing channels and the inter-level routing, resulting in area savings. The

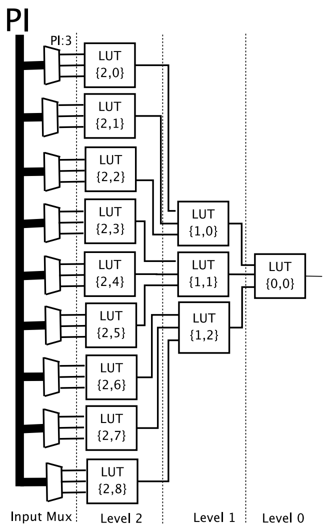


Figure 7: Hard-Wired directional array with k=3

main difference in the architecture suggested in figure 7 compared to that shown in figure 4(c) is that the routing overhead introduced by the flexibility between layers is replaced by what has been termed a logic overhead. This logic overhead is due to the fact that a large number of decomposed CLEs would not require all of the LUTs in the array. Although many of the LUTs will not implement a logic expression they may still be required to route PI to the appropriate LUT and due to the removal of the horizontal routing channels it is no longer possible to *inject* PIs at the required level.

For an array of this type it can be guaranteed that providing the CLE has less than I_p inputs and the decomposed CLE depth is not greater than the depth of the array, the array will be able to implement any CLE. Using the work carried out previously [25] it is possible to determine the maximum possible depth for a decomposed CLE with respect to the primary inputs. This in turn makes it possible to generate an array that is deep enough to ensure that any CLE of I_p, or less, inputs can be implemented on a fixed routed array.

As figure 7 shows, the only routing required within a device of this nature are the multiplexors used to route the PIs to the appropriate LUTs in the last column of the array. It should be noted that, unlike the majority of architectures studied, the column that produces the POs is termed the first column and given the label number 0 and the last column is the one immediately adjacent to the inputs MUXs. This is due to the nature of the place and route tool and the results of decomposition. In traditional decomposition labelling starts from the PIs and moves through the array to the POs.

Due to the regular growth of the array the number of LUTs (N_l) in any level (l) can be calculated by the expression

$$N_l = k^l$$

using this the total number of LUTs (N_d) in an array of depth d can be calculated by

$$N_d = \sum_{l=0}^{l=d-1} k^l$$

and as a multiplexor is required to feed each of the inputs of the LUTs in the final column the number of multiplexors in the array (N_m) has to be

$$N_m = k * k^{d-1} = k^d$$

4.1 Architectural generation

In order to investigate the architecture suggested, an HDL generator was created to automate the generation of arrays. The generator takes the number of inputs (I_p) to the array, the array depth (d), the number of outputs (O_p) and the size of the LUTs (k) and generates the required HDL code to implement the array. The following section gives a brief description of the process of generating each of the main components of the array. It should be noted that after each component has been generated a short test script ensures the correct operation of each of the components.

4.1.1 Multiplexor generation

The first component to be generated are the input multiplexors. These are basic $\{I_p + 1\} : 1$ multiplexors. It is necessary to allow for an extra input to be provided to allow constant '0' to be applied to the array for situations where LUTs are unused.

The multiplexors provide a configuration bus $log_2(I_p + 1)$ in width that is used to select the desired inputs to be passed to the output. The programming of these components is synchronous and hence the multiplexors require a clock in order to read the program bus. These components have been designed to be as simple as possible as it is envisioned that they will be replaced by standard cell components when synthesised.

4.1.2 LUT generation

The LUTs are generated based on the value of k given. Internally a register is created, 2^k in size, that is used to store the configuration of the bus. A configuration bus 2^k wide is then provided to allow the LUT to be completely programmed in a single clock cycle. This component is synchronous with an input being used to indicate to the device the availability of configuration data.

Although a number of architectures for the LUTs have been considered, including individually addressable LUTs and handshaking to avoid race conditions, it was decided to use the simplest implementation possible to allow for replacement with standard cell components prior to synthesis.

4.1.3 Array Generation

The array is then generated by instantiating the required number of LUTs and multiplexors to produce a completely fixed routed array. The array produced has I_p inputs, a depth of d and a single output. The output LUT is generated first, followed by k LUTs in the next column connected directly to the inputs of the PO LUT. This operation continues until an array of depth d has been produced. At this point a mutliplexor is connected to each of the LUT inputs in the final column and connected directly to the PIs.

In order to allow for initial investigation and to reduce the configuration time required for the device the configuration buses of all of the components in the array are made available. This allows the entire array to be programmed in a single clock cycle, this results in a configuration bus width for the array (A_b) of

$$A_b = 2^k N_d + log_2(I_p + 1)N_m = 2^k \sum_{l=0}^{l=d-1} k^l + log_2(I_p + 1)k^d$$

The use of addressable LUTs could be used to reduce this bus width by using a more complex bus structure where each LUT is given a unique address and placed on a common bus. In this way packets of data 2^k in length would be placed on the bus with the address of the LUT the configuration data is intended for. This scheme would only required a bus width of $log_2(N_d) + 2^k$ but would require at least a clock cycle for each LUT in the array. This idea was rejected in initial testing due to the complexity of the LUTs required and the need for hand-crafted components during synthesis.

The use of Scan-Chains was also considered where the configuration memory of several LUTS are chained together to act as a shift register. In this way the configuration data would be passed to the first LUT in the chain and then on to the next LUT in the chain in the next clock cycle untill the configuration data reaches the corrcet LUT in the chain. Again, this was rejected due to the use of non-standard components and also the time increase incurred during reconfiguration. It is felt however that for future development this is the most promising route to follow to reduce the size of the configuration bus required by the array.

4.1.4 Device generation

The final device can then be produced by instantiating the number of arrays required to produce the required number of outputs. As this is based on the array produced the configuration bus size now increase by a factor of O_p. Each of the arrays has its own set of inputs and configuration buses and the device generator is required to label each of these inputs individually and supply the clock required for the synchronous LUTs and multiplexors.

4.2 Development of place and route tools for fixed routed arrays

Due to the lack of flexibility in the fixed routed system the place route tools required to map CLEs to this array are very simple. Readily available decomposition tools, such as SIS [28], can perform the decompositions necessary to break the CLE down into an array of k bounded sub-expressions. In the case of the array shown in figure 7 this would be an array of expression with a maximum of 3 inputs. It should be noted that the number of POs is limited to 1. This treats each output as a distinct cone and hence requires an array to be produced for each PO.

The *place and route tool* first identifies the PO node in the decomposed network that produces the output of the CLE. The bit sequence required to program the output LUT, LUT{0,0}, to implement the expression for this node is then calculated. The function mapped to this LUT is then called the parent function and each of the nodes that act as inputs are the child nodes of this expression. As the CLE has been decomposed to sub-functions with a maximum of k inputs there will be at most k children of any parent function in the array.

The *place and route tool* then moves to the next column and maps each of the children to the corresponding LUTs so they are connected directly to the parent function. Each of these child functions then becomes a parent in its own right and the process of identifying its children is carried out and the bit streams required to implement the functions of the children are created. In this way the algorithm starts at the output column and works its way back to the PIs, moving down the columns and mapping the children to the required LUTS. If at any stage the child of a parent node is found to be a primary input the LUT is programmed to forward one of its inputs and perform no combinatorial operation on this function.

In the case of a parent with less than k children the LUTs connected to the inputs of the parent that are not required are programmed to perform no operation and for future systems it is envisioned that this would represent a low power state to prevent unused LUTs from unnecessarily increasing the power consumption of the device.

4.3 Hardware implementation of fixed routed array

Although synthesis has not been carried out on the array it is possible to consider what the results of synthesis will be. The hardware implementation of the array would result in a balanced tree of LUTs hardwired together. It would be necessary to implement a clock to provide clock signals to each of the LUTs as they are synchronous. It is not necessary, and in fact undesirable, to use a common clock signal for each of the LUTs in the array as if a parent and child LUT are clocked simultaneously race conditions may occur as the output of the child LUT would change as the parent LUT reads its input. This may result in the array exhibiting unpredictable or undesirable behaviour and prevent the synthesis tools achieving timing closure. This is not only confined to the synthesised array but has also been seen in simulation.

A number of possible solutions were considered, such as the use of asynchronous

LUTs making use of handshaking, but as these require the use of none standard components they were rejected due to the need for custom design. An alternative solution is the use of delays on the clock tree. In this way the clock tree would ripple through the array and be delayed at each column in the array by the minimum delay time of the LUTs used. In this way the clock could arrive at the first column in the array, where it would clock the LUTs and perform the functions required of the LUTs in this column. A delay element would delay the clock for the period of time required by the LUT to calculate their new output. This delayed clock would then be applied to the LUTs in the next column allowing the new output of the child LUTs. The clock would again be delayed to allow the LUTs in this column to calculate their output before clocking the LUTs in the next column. This would then continue through the array until the final output is calculated.

5 Analysis of results

Based on the work carried out in [25] it is possible to investigate the amount of *logic overhead* introduced by the array. This overhead is due to the fact that many of the branches within the array will not be used to implement a sub-function but will be instead required to forward PIs to the appropriate LUT or simply to carry out no operation. This would be the case for a sub-function that has less than k-inputs. In this case some of the child LUTs would no longer be required and hence all of the LUTs contained in the cone of this LUT would also be unused. In order to investigate this the data presented in [25] was used to investigate the number of LUTs and depth of array required to implement the average CLE with I_p-inputs. Based on the depth of the array required to implement the average CLE it is possible to calculate the number of LUTs in the resultant fixed routed array using the relationship

$$N_d = \sum_{l=0}^{l=d-1} k^l$$

this can then be compared to the average number of LUTs required to implement the CLE on a conventional architecture as given in [25]. Based on this the LUT usage percentage can be calculated and these are presented in table 1 for CLEs with inputs in range $4 \leq I_p \leq 10$ that have been decomposed for implementation using LUTs with k values in the range $3 \leq k \leq 5$.

	k		
I_p	3	4	5
4	69.48	100	100
5	51.92	59.86	100
6	36.6	33.26	50
7	75.74	17.63	19.35
8	50.97	36.45	41.94
9	34.17	18.47	16.67
10	22.94	9.3	6.79

Table 1: Percentages of LUTs usage for average case.

As this table shows the percentage of the LUTs used drops as the number of inputs increase due to the increase in the depth of the array. The results show the relationship between the percentage of LUTs used and the number of inputs is actually stepped in nature due to the fact that the depth of the array required to implement an I_p input expression does not increase linearly. For eample an array of depth 4 is required to implement expressions with $I_p = 6$ and $I_p = 7$ for k=3. Due to the regular nature of the array this would result in an array containing 40 LUTs to be required for both $I_p = 6$ and $I_p = 7$. As the mean number of LUTs required to implement an expression for $I_p = 7$ is greater than that for $I_p = 6$ the usage percentage increases for this case. It can then be seen that this percentage drops for $I_p = 8$ for k=3 as expressions of this nature require an extra level and hence more LUTs in the array. Although the LUT usage appears to be low this does not take into account the massive reduction in routing overhead produced by this architecture.

6 Future Work

As development of this system is still at the early stages a number of areas of future work have been identified. These are mainly concerned with optimising the array to reduce the overhead introduced by the excessive numbers of LUTs in the array. It has been found that only around 40% of the total LUTs in the array are being utilised to implement sub-functions with the remainder either implementing no function or simply routing PIs to the required level within the array.

In order to improve this an unbalanced tree structure has been suggested. This is capable of implementing a large subset of the functions possible as it is unlikely that all of the sub-functions of the network will require $|inputs(v)| = k$. In this case it can be seen that if the number of inputs is less than k some of the child nodes of the array will not be used and all of the LUTs in the cone of these unused LUTs will also be unused.

For this reason it is suggested that branch pruning be performed where different length branches be used as shown in figure 8. As this shows several of the LUTs from the architecture presented in figure 7 have been removed and the LUTs can be fed directly with the PIs to avoid the need for a LUT to forward these signals. This process would require more complex mapping algorithms that can identify the critical path in the decomposed network and map this to the longest available path in the network. In order to efficiently implement an array using this method it is suggested that a large number of networks be profiled to determine the distribution of levels within the decomposed network and this information be used to perform branch pruning.

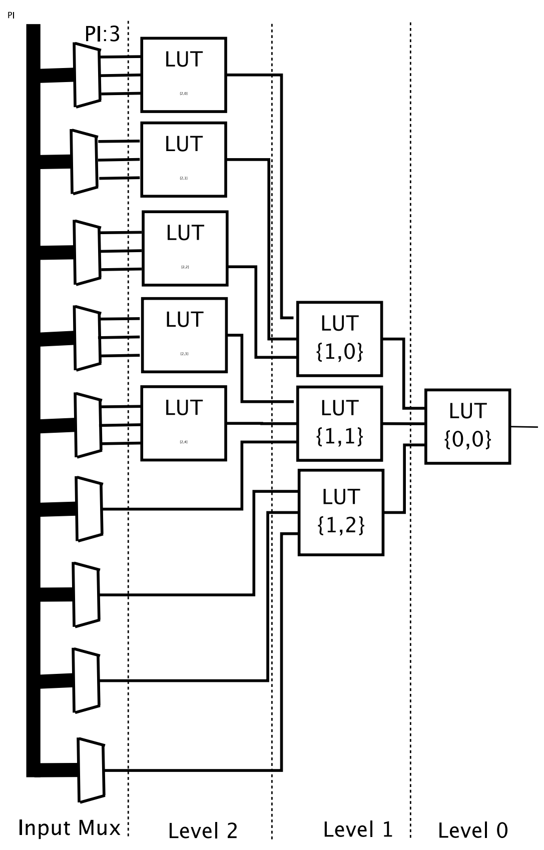


Figure 8: Unbalanced directional fixed routed array

Due to the simplicity of the place and route algorithm it can be executed very rapidly once decomposition has been carried out. In fact it was found during testing that the most time consuming operation carried out by the tools created was the decomposition of the original CLE using the SIS synthesis tool. This opens up the possibility of performing place and route on the fly, where a block of logic implements the simple P&R tool and is provided with decomposed arrays to be implemented from an external, or even local, source. This would allow an adaptive system to be created where the device determines the necessary changes to its own configuration and actually maps and loads this new configuration with no external intervention. This could be very useful in applications that are difficult to gain access to in field such as in a nuclear reactor or in space applications.

7 Conclusion

This section has given details of a hardwired directional array that is specifically tailored to the implementation of CLEs based on the results of decomposition. This architecture is an extension of the directional architecture suggested in [32] and the concept of using hardwired interconnects suggested in [29]. Although in [9] it was reported that the use of hardwired interconnects increased the difficulty in mapping networks to device this was due to the fact that only selected channels were hardwired. As the architecture suggested here contains only hardwired interconnects the *place and route* operation is extremely simple and fast.

At present the code necessary to generate an array of this type in a hardware description language has been created to allow for the investigation of the use of fixed routed arrays. The simple place and route tool required to derive the bit-stream to implement a CLE has also been developed. The next stage in development is the synthesis of the HDL descriptions of the array in order to obtain the performance metrics, such as area, speed and power, to allow its comparison to other domain specific architectures such as that suggested in [33].

References

[1] Elias Ahmed and Jonathan Rose. The effect of LUT and cluster size on deep-submicron FPGA performance and density. In *FPGA*, pages 3–12, 2000.

[2] Altera. Stratix iii fpgas vs. xilinx virtex-5 devices: Architecture and performance comparison. Technical report, Altera, 2006.

[3] R. K. Brayton, R. Rudell.and A. Sangiovanni-Vincentelli, and A. R. Wang. Mis: A multiple-level logic optimization system. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 1987.

[4] Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field-programmable gate arrays.* Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[5] Shih-Chieh Chang and Malgorzata Marek-Sadowska. Technology mapping via transformations of function graphs. In *International Conference on Computer Design*, pages 159–162, 1992.

[6] Kuang-Chien Chen and Jason Cong. Maximal reduction of lookup-table based fpgas. In *EURO-DAC '92: Proceedings of the conference on European design automation*, pages 224–229, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[7] Kuang-Chien Chen, Jason Cong, Yuzheng Ding, Andrew B. Kahng, and Peter Trajmar. Dag-map: Graph-based fpga technology mapping for delay optimization. *IEEE Des. Test*, 9(3):7–20, 1992.

[8] P. Chow, S. Seo, J. Rose, K. Chung, G. Paez, and I. Rahardja. The design of an sram-based field-programmable gate array, part i: Architecture, 1999.

[9] Kevin Chung and Jonathan Rose. TEMPT: Technology mapping for the exploration of FPGA architectures with hard-wired connections. In *Design Automation Conference*, pages 361–367, 1992.

[10] K. Compton and S. Hauck. Totem: Custom reconfigurable array generation, 2001.

[11] J. Cong. Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based fpga designs, 1994.

[12] Jason Cong and Yuzheng Ding. Combinational logic synthesis for lut based field programmable gate arrays. *ACM Trans. Des. Autom. Electron. Syst.*, 1(2):145–204, 1996.

[13] Jason Cong and Yean-Yow Hwang. Structural gate decomposition for depth-optimal technology mapping in LUT-based FPGA designs. *ACM Transactions on Design Automation of Electronic Systems.*, 5(2):193–225, 2000.

[14] C. Ebeling, D. C. Cronquist, P. Franklin, and C. Fisher. RapiD - A configurable computing architecture for compute-intensive applications. Technical Report TR-96-11-03, 1996.

[15] R. FRANCIS. Technology mapping for lookup-table based fieldprogrammable ga te arrays, 1992.

[16] Robert Francis, Jonathan Rose, and Zvonko Vranesic. Chortle-crf: Fast technology mapping for lookup table-based fpgas. In *DAC '91: Proceedings of the 28th conference on ACM/IEEE design automation*, pages 227–233, New York, NY, USA, 1991. ACM Press.

[17] A. Gayasen, K. Lee, N. Vijaykrishnan, M. Kandemir, M. Irwin, and T. Tuan. A dual-vdd low power fpga architecture, 2004.

[18] R. Hartenstein. Trends in reconfigurable logic and reconfigurable computin. In *Electronics, Circuits and Systems, 2002. 9th International Conference on*, 2002.

[19] A. El Gamal J. Rose and A. Sangiovanni-Vincentelli. "architecture of field-programmable gate arrays". *Proceedings IEEE, vol. 81*, 1993.

[20] Noha Kafafi, Kimberly Bozman, and Steven J. E. Wilton. Architectures and algorithms for synthesizable embedded programmable logic cores. In *FPGA '03: Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 3–11, New York, NY, USA, 2003. ACM Press.

[21] Sami Khawam, Tughrul Arslan, and Fred Westall. Unidirectional switchboxes for synthesizable reconfigurable arrays. In *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 293–295, Washington, DC, USA, 2004. IEEE Computer Society.

[22] M. Khellah, S. D. Brown, and Z. Vranesic. Minimizing interconnection delays in array-based fpgas. In *in Proc. Custom Integrated Circuits Conf*, pages 181–184, 1994.

[23] David Lewis, Elias Ahmed, Gregg Baeckler, Vaughn Betz, Mark Bourgeault, David Cashman, David Galloway, Mike Hutton, Chris Lane, Andy Lee, Paul Leventis, Sandy Marquardt, Cameron McClintock, Ketan Padalia, Bruce Pedersen, Giles Powell, Boris Ratchev, Srinivas Reddy, Jay Schleicher, Kevin Stevens, Richard Yuan, Richard Cliff, and Jonathan Rose. The stratix ii logic and routing architecture. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 14–20, New York, NY, USA, 2005. ACM Press.

[24] M. Imran Masud and Steven J. E. Wilton. A new switch block for segmented FPGAs. In Patrick Lysaght, James Irvine, and Reiner W. Hartenstein, editors, *Field-Programmable Logic and Applications*, pages 274–281. Springer-Verlag, Berlin, / 1999.

[25] G. Milligan. Investigation of the characteristics of reconfigurable devices. In *EngD Portfolio Document (TR2)*, 2007.

[26] Rajeev Murgai, Yoshihito Nishizaki, Narendra Shenoy, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Logic synthesis for programmable gate arrays. In *DAC '90: Proceedings of the 27th ACM/IEEE conference on Design automation*, pages 620–625, New York, NY, USA, 1990. ACM Press.

[27] Martine Schlag, Jackson Kong, and Pak K. Chan. Routability-driven technology mapping for lookup table-based fpgas. Technical report, University of California at Santa Cruz, Santa Cruz, CA, USA, 1992.

[28] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, 1992.

[29] Satish Sivaswamy, Gang Wang, Cristinel Ababei, Kia Bazargan, Ryan Kastner, and Eli Bozorgzadeh. Harp: hard-wired routing pattern fpgas. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 21–29, New York, NY, USA, 2005. ACM Press.

[30] Synopsis. Synopsis galaxy design platform. http://www.synopsys.com/products/solutions/galaxy_platform.html.

[31] Steven J. E.. Wilton, Noha Kafafi, J. Wu, K. Bozman, V. Aken'Ova, and R. Saleh. Design considerations for soft embedded programmable logic cores. In *IEEE Journal of Solid-State Circuits*, pages 485–497, 2005.

[32] Andy Yan and Steve J. E. Wilton. Product-term-based synthesizable embedded programmable logic cores. In *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, pages 474–488, 2006.

[33] Andy Chee Wai Yan. Product-term based synthesizable embedded programmable logic cores. Master's thesis, University of British Columbi, 2005.

Portfolio Document A1

Reconfigurable Hardware: Literature Review

Author:

Graeme Milligan

Supervisor:

Wim Vanderbauwhede

Contents

List of Figures

1 Introduction

The exponential growth in the cost of ASIC design has lead to research in a number of areas including system on chip, IP usage and Reconfigurable hardware. Reconfigurable hardware is of particular interest as it promises to cut design cost while increasing product life cycle by introducing flexibility to hardware designs.

Traditionally, systems were either implemented in hardware, VLSI, ASIC, etc, or in software running on a general purpose processor. Although hardware implementation leads to highly efficient, high performance systems they suffer from long design cycles and inflexibility. This means that they are costly to design and have very short life cycles. The software route allows systems to be designed rapidly and allows them to be reprogrammed in-field. This flexibility means that these processors must be designed to be as general as possible and are hence very inefficient. In order to reduce this inefficiency there has been a drive to produce more application specific processors, such as DSP's. Reconfigurable hardware aims to combine the advantages of both software and hardware by providing the speed and efficiency of hardware and the flexibility of software. Thus is achieved by utilising hardware which contains an array of computational elements which are programmable. These elements, known as logic blocks, are connected using a set of programmable routing resources.

This report aims to give background information and review a selection of current academic and industrial approaches to reconfigurable hardware and the issues concerning this subject.

2 Background

The concept of configurable hardware began in the 1960's with the development of the gate array. Initially these consisted of a base platform containing an array of gates with no interconnect, which could be produced in bulk. The functionality of these platforms could then be customised by designing the interconnects and applying these in the final mask stages. Although this reduced production cost and simplified design, the systems could only be programmed once and were less efficient than full-custom design.

The next stage in the development of configurable systems was the advent of field programmable gate arrays (FPGA's). These are arrays of gates with interconnects which can be programmed by applying a high current to an anti-fuse. Once this has been done, a permanent connection is made between the gate and the interconnect. This means that these devices could only be programmed once and suffered from low gate densities due to the large amount of hardware required for routing and programming. For this reason it was impractical to implement complete systems on FPGA's and they were mainly utilised for emulation or system prototyping.

In recent times with the development of FPGA's, such as EEPROMS and SRAM based FPGA's, which can be reprogrammed multiple times, truly reconfigurable systems are now possible. The large increase in the available gate counts (over 1M in modern SRAM FPGA's) also means that it is now possible to implement large, complex systems on a single FPGA. Due to the capability of SRAM based FPGA's for rapid reconfiguration (less than 1 microsecond) there has been much interest in dynamically reconfigurable hardware. This is an FPGA, or other reconfigurable device, which can be fully or partially reconfigured during system operation. This means that a single reconfigurable hardware block can be used for one operation until complete and then be reconfigured to perform another function.

Due to the generality of FPGA's it is fairly obvious that they far less efficient than full custom designs. It has been reported that only 1 percent of the chip area serves the real application, where as the other 99 percent is reconfigurable overhead [8]. This is mainly due to the routing required to interconnect the reconfigurable units. For this reason many FPGA's are beginning to incorporate standard components such as MAC units, multipliers and even processors in order to reduce this overhead. This hybrid approach gives the flexibility of the reconfigurable gate array with the added efficiency of full custom design. A number of examples of this including the Xilinx Virtex series of FPGA's which contain the IBM PowerPC processor [13] and the Altera Excalibur series which contains an ARM922T processor [2, 8].

This approach has also lead to the idea of producing course grained reconfigurable platforms. In these platforms the reconfigurable units are larger (ALU's as opposed to gates) and hence incur a smaller overhead due to routing. This form of platform is especially efficient in datapath designs where a number of these elements can be connected together to form standard width (16 bit, 32 bit etc.) datapaths. As this reduces the routing overhead it also reduces the power consumption and increases the switching speed.

3 System Classification

Due to the broad range of tasks to which reconfigurable hardware is currently being applied it is necessary to develop a system of classification. This could be achieved using a number of criteria including application area and granularity. It seems fairly obvious that due to the flexibility inherent to reconfigurable hardware, that it would be unwise to classify these systems by application area as it is likely that a given approach will be used in a number of very different applications. Although it is possible to classify reconfigurable systems by their granularity, this would be very objective and vague. For example a system which is made up of reconfigurable ALU's could be described as course-grained when compared to an FPGA but would be fine-grain compared to a platform containing arrays of reconfigurable processors.

In [6] reconfigurable systems are classified by the degree of coupling between the microprocessor used to control reconfiguration and the reconfigurable logic. Figure 1 shows that it is possible to broadly classify all reconfigurable systems into four categories. The first of these categories is the case where reconfigurable hardware is used to provide reconfigurable functional blocks within the host processor. This allows the processor to be used as normal with the addition of custom instructions designed for a given application. In this form of coupling an overhead is introduced as the processor and reconfigurable unit must communicate every time a reconfigurable instruction is used.

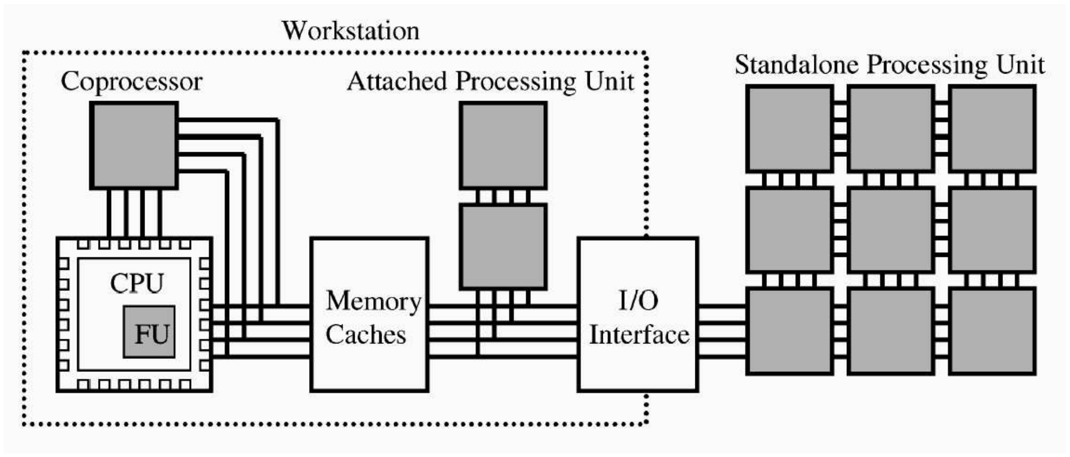


Figure 1: Levels of coupling in Reconfigurable system

The second category is the use of reconfigurable hardware as a coprocessor. The coprocessor is normally able to perform fairly complex tasks and can operate independently of the processor. In this form of coupling it is likely that the processor would be responsible for initialising and providing input data to the coprocessor, the coprocessor would then operate independently and when complete return the result to the processor. This allows the coprocessor and the main processor to operate simultaneously and hence reduces any communications overhead i.e. the processor no longer has to stall while the reconfigurable hardware is operating.

Third, an attached reconfigurable processing element behaves as an additional attached processor in a multiprocessor system. The host processors data cache is not visible to the reconfigurable processing unit and hence there is a greater delay in any communication between the units. To improve efficiency communication would happen less frequently and involve larger chunks of data. This means that the reconfigurable processing unit would often be larger than the coprocessor approach and carry out more complex tasks. This allows for a great deal of computation independence and allows greater parallelism by shifting large amounts of the computation over to the reconfigurable unit.

The final, most loosely coupled form of reconfigurable hardware is that of an external stand-alone unit. This reconfigurable hardware communicates infrequently with the host microprocessor (if there is one present). In this model it is likely that the reconfigurable unit will perform processing over long periods of time and hence have limited communications. This form of coupling allows the processor to send large amounts of data to the reconfigurable unit and then forget about it until the reconfigurable unit completes its processing task.

Each of these approaches has its own advantages and disadvantages. The tighter the coupling the more frequently the reconfigurable unit can be used due to a small communications overhead. However, the reconfigurable hardware is unable to operate independently for long periods of time and hence requires frequent intervention from the host processor. In loosely coupled systems a large communications overhead is introduced but a higher degree of parallelism is achieved. The amount of reconfigurable hardware available is also often much larger in these systems.

The following sections are intended to give an overview of both industrial and academic research in the field of reconfigurable hardware and highlight a selection of interesting approaches to this subject. A list of reconfigurable research and systems can be found at [7, 1], although these contain only short summaries of the projects and are incomplete.

4 Industrial Reconfigurable Hardware Systems

This section is not intended to detail every reconfigurable system and research project in industry but instead give an overview of selected systems of interest and give references to the location of further information. Although several of the a large FPGA manufacturers, including Xilinx and Altera, have produced platforms which are intended for use in reconfigurable systems it is felt that these platforms have been covered extensively in literature and hence it would serve little purpose to examine them here. Instead the novel PACT XPP platform [4], the picoChip [11] and the Elixent RAP [10] will be described.

4.1 PACT XPP

The PACT eXtreme Processing Platform (XPP) is a run time reconfigurable platform designed for data processing. Figure 2 shows the structure of a typical XPP device. The architecture is based on a hierarchical array of coarse-grain elements called Processing Array Elements (PAE's) and a packet oriented communications network. Each PAE contains an ALU and registers (FREG and BREG) used to control internal routing. PAE's are grouped into rectangular blocks called Processing Array Clusters (PAC's) and each XPP device is made up of one or more of these PAC's, the example in figure 2 contains four PAC's.

As the configuration of the device is controlled by a hierarchical tree of configuration mangers (CM's) it is possible for the leaf CM's to configure individual PAE's while there neighbours are operating. The root CM is called the supervising CM (SCM) and contains an external interface which usually connects the SCM to an external configuration memory. The PAE's communicate using a packet-oriented network which can transmit data packets of uniform bit width specific to the device type. In this way the XPP can be used to pass data packets rapidly around the platform and perform word length operations efficiently. This makes it ideal for data intensive operations such as DSP and low level protocol operations.

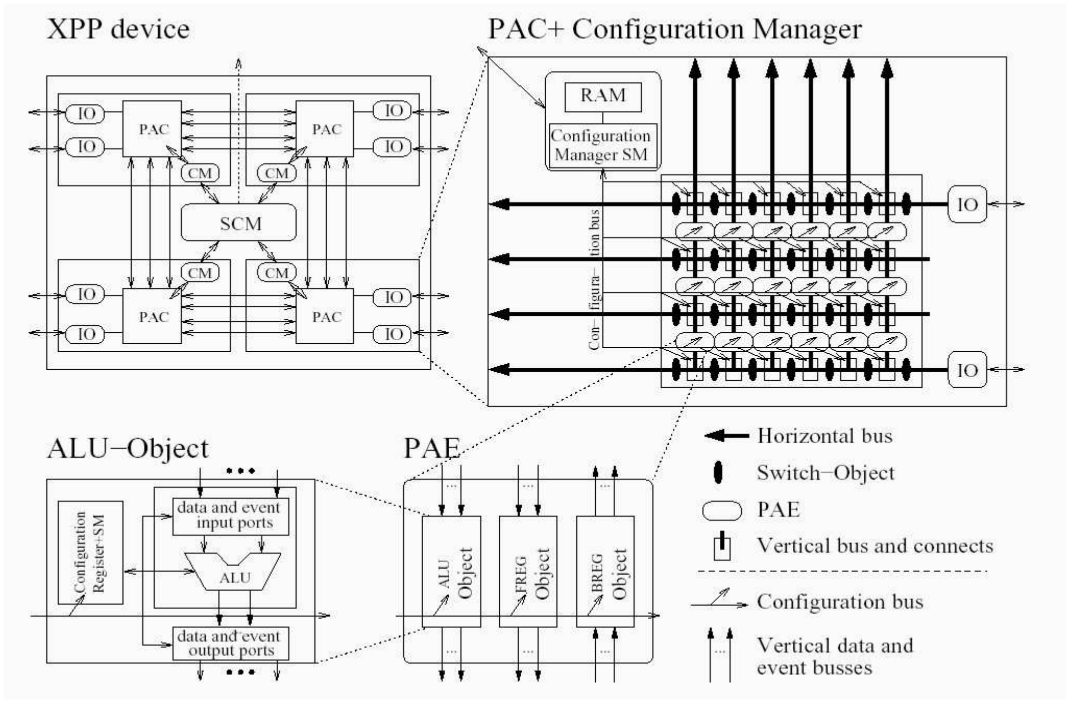


Figure 2: Structure of PACT XPP taken from [4]

4.2 PicoChip

The picoChip platform is a scalable multi-processor baseband IC designed for use in 3G base stations and other high performance communications applications. The picoChip is based on an array of 430 processing elements, where each element is designed specifically for wireless communications systems and has the equivalent processing power of an ARM9 embedded processor. Each processor is a 16-bit device containing its own ALU, processing elements, program and data memory and can operate at 160 MHz and implements 3 instructions per cycle. The elements are interconnected by a proprietary 5 Gbit/Second bus structure shown in figure 3. The processing elements in the array can be subdivided into 4 "flavours" in order to accommodate W-CDMA baseband processing operations. The first of these contains instructions such as spread/despread and acceleration for forward-error correction. The second contains dedicated Multiply-Accumulate units for filtering operations. The third type has four times the memory of the previous two to allow it to perform applications such as block processing so that the designer can load more substantial software to the processor for control based tasks. The final "flavour" has four times the memory again and is used for scheduling operations. It is possible to map an individual task to a single or multiple processors depending on the processing power required. The processors have no run-time scheduling or arbitration, while the data flow within the array is completely scheduled. This means that the performance of the system is determined at compile time, allowing simulations to be both bit and cycle accurate. Each of the processors is programmed in C and the interconnects automatically generated, this allows applications to be developed and verified rapidly.

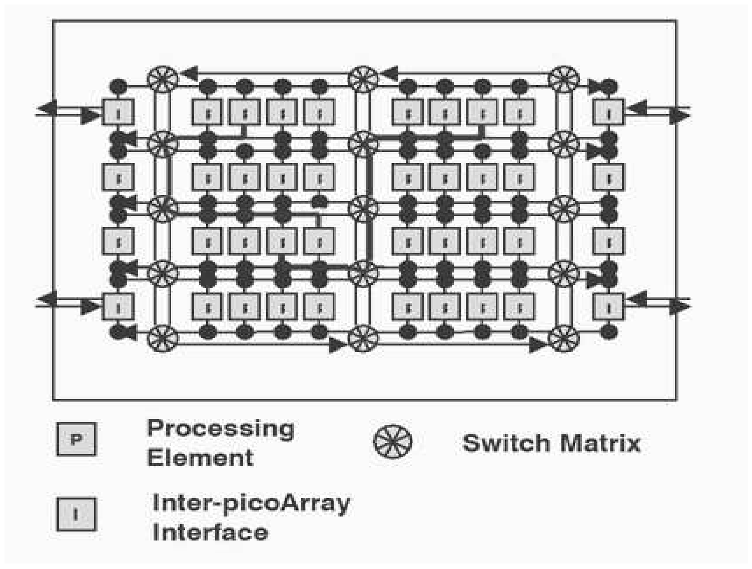


Figure 3: PicoChip array structure taken from [11]

4.3 Reconfigurable Algorithm Processor (RAP)

The Reconfigurable Algorithm Processor (RAP) from Elixent is a coarse-grain reconfigurable platform designed for DSP and multimedia applications. The reconfigurable hardware, know as the D-fabrix, is made up of an array of hundreds of 4-bit ALU's and register/buffer blocks that can be cascaded together to accommodate larger data lengths. This allows the fabric to operate on the 8-24 bit data lengths common in multimedia applications. The ALU's are arranged in a chessboard style, alternating with switchboxes which can act as a cross-point switch or 64 bits of configuration memory. Figure 4 RAP Architecture showing chessboard layout, alternating ALU and cross point switch/memory block with RAM blocks interspersed as required.

As can be seen in figure 4 the array also contains 256-byte memory blocks dispersed around the array. The choice of nibble sized ALU's means that only a few bytes of memory are required to configure each ALU allowing rapid reconfiguration and improved density. The large amount of on-chip memory also allows ALU's to be fed instruction streams generated within the array reducing off-chip memory traffic to improve overall performance. The RAP has been targeted at multimedia and wireless base-station applications and in applications such as JPEG compression they have shown speed up of 238x against a 32-bit DSP and 38x against an FPGA.

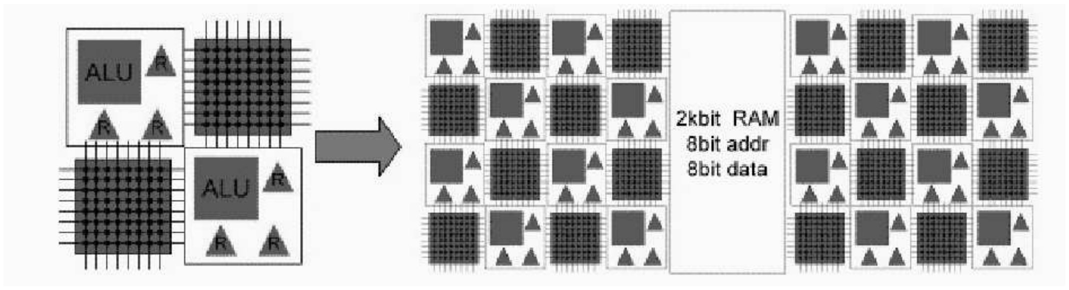


Figure 4: RAP architecture

4.4 Conclusion

These examples show that the majority of industrial projects in reconfigurable hardware are aimed mainly at producing general purpose platforms which although aimed at high end communications and multimedia applications, are as general as possible to allow them to be utilised in a number of applications. It can also be seen that many of these systems have adopted a course grain architecture in order to reduce the overheads associated with fine grain reconfiguration e.g. routing and reconfiguration overhead. Although a number of applications have been designed for these platforms they are intended to act as benchmarks to demonstrate speed and power improvements over traditional FPGA and microprocessor implementations. At present, the majority of these platforms have yet to be utilised for any complete application and hence real analysis of their performance is yet to be performed.

5 Academic Reconfigurable Hardware Systems

Again, this section is not intended to cover every academic research project currently underway, but give an overview of novel approaches to reconfigurable hardware. Compared to the industrial projects listed previously there is a much greater diversity of approaches in academia but the majority of projects are still aimed at the multimedia and communications application regions due to the challenging speed and power requirements of these applications. The systems selected for this review are the DReAM architecture developed at Darmstadt University of Technology and Ohio University [5] and the PRO3 protocol processor.

5.1 DReAM

The Dynamically Reconfigurable Architecture for Mobile systems (DReAM) consists of an array of parallel operating course-grain Reconfigurable Processing Units

(RPUs). It is tailored to perform all the required arithmetic manipulations required for the data-flow oriented mobile applications area. With this in mind the RPUs were designed to efficiently perform the course-grained (8-bit) integer operations required for these applications. The complete DReAM array architecture, shown in figure 5, connects all RPUs with reconfigurable 16-bit local and global communication structures. The overall operation of the array is controlled by the Communication Switching Units (CSUs), each of these units controls the configuration of two Configuration Memory Units (CMUs), which themselves each control four RPUs, and four global interconnect Switching Boxes (SWBs). The overall control of the complete array is performed by the global communications unit which also controls communication with other hardware components.

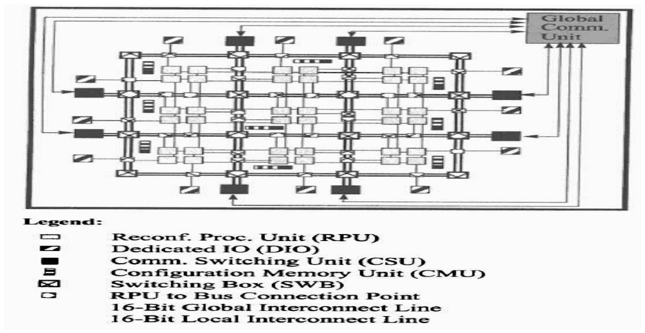

Global Comm. Unit
Legend:
Reconf. Proc. Unit (RPU)
Dedicated IO (DIO)
Comm. Switching Unit (CSU)
Configuration Memory Unit (CMU)
Switching Box (SWB)
RPU to Bus Connection Point
16-Bit Global Interconnect Line
16-Bit Local Interconnect Line

Figure 5: Hardware Structure of DReAM array

5.2 PRO^3

The reconfigurable Protocol Processor (PRO^3) developed as an academic and industrial joint venture by Lucent, Hyperstone, IMEC, Ellemedia and the National Technical university of Athens is designed to accelerate the execution of telecom transport protocols by extending a RISC core with reconfigurable pipelined hardware. This form of acceleration is common to a number of academic projects including PRISM

[3], GARP [9] and DISC [12]. The platform is designed to allow CPU demanding real-time protocol functions to be executed by the programmable hardware, while the remaining functions and higher layer protocols will be handled by the on-chip RISC. The overall design of the PRO^3 is shown in figure 6. From this it can be seen that the reconfigurable hardware is intended to work in parallel with the RISC core and could be considered as another stage in the processor pipeline. This is a very tightly coupled system where performance gains are expected by identifying protocol functions which are executed most often and migrating these functions to hardware. At this stage it is unclear if the project will utilise the fine-grain reconfiguration of FPGA technology or a more course-grain approach. At this time it is unclear how efficiently these protocol functions can be implemented in a reconfigurable system, as the flexibility required to implement a broad range of functions will also lead to a large routing overhead.

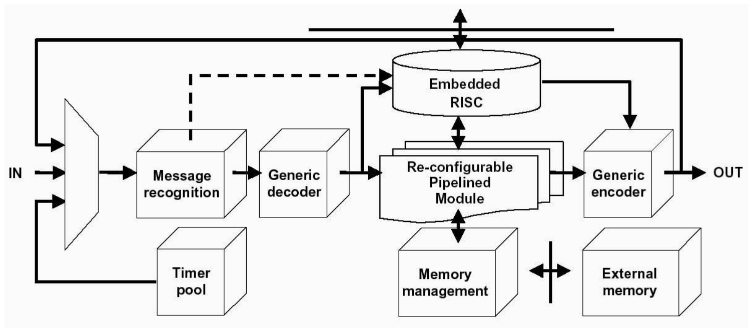


Figure 6: PRO^3functional architecture and data paths

5.3 Conclusion

The projects given here are intended to give examples of the approaches to reconfigurable hardware currently utilised in academic projects. The coprocessor approach used by the PRO^3 project is very common in academic research. This approach allows standard components such as an off the shelf processor and FPGA array to be utilised and hence requires less design effort than the approach utilised by industrial platforms such as PACT XPP.

6 Conclusion

The exponentially increasing cost of ASIC design and the need for performance greater than that available in present microprocessors has lead to great interest in Reconfigurable hardware. Reconfigurable hardware promises to give improved performance over microprocessors and improved flexibility over ASIC designs. It is hoped that this will cut the cost of system design while giving the performance required for modern applications. In this paper a selection of academic and industrial approaches to reconfigurable system design were presented in order to give an overview of the current state of the art. It can be seen that the majority of industrial projects are intended to produce a platform which contains the flexibility of reconfigurable systems but is tailored to a particular application. This approach allows platforms to be used for a number of applications within a particular field, i.e. a receiver for several protocols, while not suffering an unacceptably high overhead due to this flexibility. Many of the academic projects are based around a custom processor approach. As stated previously this allows a system to be produced rapidly from off the shelf components, this allows researchers to concentrate on other aspects of the reconfigurable hardware such as how the processor and reconfigurable hardware interact, and tool development to allow rapid system development.

References

[1] R. Abielmona. Alphabetical list of reconfigurable computing architectures. http://www.site.uottawa.ca/~rabielmo/personal/rc.html.

[2] Altera. http://www.altera.com.

[3] P.M. Athanas. A functional reconfigurable architecture and compiler for adaptive computing. In *Twelfth Annual International Phoenix Conference on Computers and Communications*, 1993.

[4] V. Baumgarte, G. Ehlers, F. May, A. Nuckel, M. Vorbach, and M. Weinhardt. PACT XXP a self-reconfigurable data processing architecture. *J. Supercomputing.*, 26(2):167–184, 2003.

[5] J. Becker, L. Kabulepa, F. Renner, and M. Glesner. Simulation and rapid prototyping of flexible systems-on-a-chip for future mobile communication applications. In *RSP '00: Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping (RSP 2000)*, page 160, Washington, DC, USA, 2000. IEEE Computer Society.

[6] K. Compton and S. Hauck. Reconfigurable computing: A survey of systems and software, 2000.

[7] S. Guccione. List of fpga-based computing machines. http://www.io.com/~guccione/HW_list.html.

[8] R. Hartenstein. Trends in reconfigurable logic and reconfigurable computin. In *Electronics, Circuits and Systems, 2002. 9th International Conference on*, 2002.

[9] J. R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[10] ELIXENT Ltd. The reconfigurable algorithm processor. www.elixent.com/assets/WP0001_D_Fabrix_Apps.pdf.

[11] Picochip. http://www.picochip.com.

[12] M. J. Wirthlin and B. L. Hutchings. Sequencing run-time reconfigured hardware with software. In *FPGA '96: Proceedings of the 1996 ACM fourth international symposium on Field-programmable gate arrays*, pages 122–128, New York, NY, USA, 1996. ACM Press.

[13] Xilinx. http://www.xilinx.com.

Portfolio Document A2

Universal Mobile Telecommunications System: Protocol Review

Author:
Graeme Milligan

Supervisor:
Wim Vanderbauwhede

Contents

List of Figures

1 UMTS

The Universal Mobile Telecommunication system (UMTS) has become the standard radio system for the third-generation (3G) of mobile communication systems in Europe and Japan with a similar system (cdma2000) being adopted in America. UMTS has been under development in Europe since the late 1980's with the RACE and RACE 2 projects. In 1999 the development and standardisation of UMTS was given to the Third Generation Partnership Project (3GPP). 3GPP is global partnership made up of standardisation organisation from Europe, Japan, Korean and the USA and aims to produce a global standard mobile communications system.

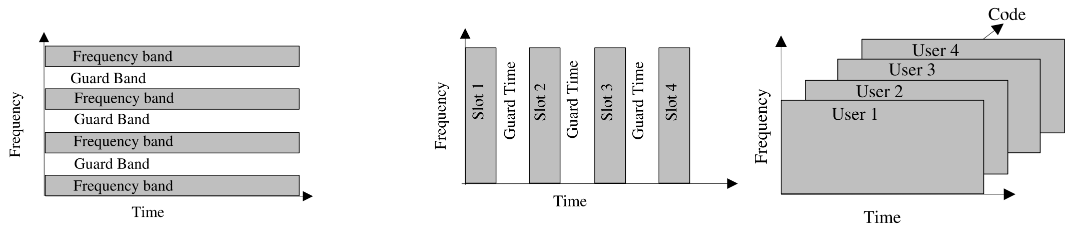
Within UMTS two air interfaces are specified, Wide-band Code Division Multiple Access (WCDMA) Frequency Division Duplex (FDD) and WCDMA- Time Division Duplex (TDD). The FDD interface is based on paired 60MHz bands (1920MHz-1980MHZ and 2110 MHz-21170MHz) where the lower band is utilised for uplink and the upper for downlink. The TDD interface makes use of a single 25MHz channel (1900-1920 and 2020-2025) for both uplink and downlink. For the purposes of simplicity WCDMA will be used to refer to both of these modes of operation.

In order to allow for backward compatibility with 2G communications systems, such as GSM, many of the features of these systems have been adopted in UMTS and any additional features will not adversely affect the operation of 2G systems. The main features of UMTS include

- Bit Rates of up to 2Mb.
- Variable bit rate to offering bandwidth on demand.
- Multiplexing of services with different QoS requirements.
- Low frame error and bit error rates.
- Support of asymmetric uplink and downlink traffic.

- High spectrum efficiency
- Coexistence of FDD and TDD modes.

The increase in bandwidth is expected to allow new services to be supplied to mobile terminals (referred to as User Equipment (UE) by 3GPP) such as video telephony and rapid downloading of data such as news reports and internet access. It is possible for UMTS to achieve these bandwidths through its use of WCDMA interface techniques.



(a) FDMA (b) TDMA (c) CDMA

Figure 1: Common transmission techniques

2 Spreading and Modulation in WCDMA

Traditionally, communication systems have employed either a Frequency division Multiple Access (FDMA) technique (figure 1(a)), where each user is given it own specified range of frequencies for communication, or Time Division Multiple Access (TDMA) techniques (figure 1(b)), where each user is allocated a time slot where it can use the entire frequency range for communication. In Code Division Multiple Access techniques every user has access to the entire frequency range for all time for communication and individual signals are separated using a given spreading code (figure 1(c)).

These spreading codes are created using orthogonal variable spread factor (OVSF) codes [6] that are selected to allow for maximum separation and hence allow the separation of signals from different UE. The process of spreading and de-spreading is shown in Figure 2. From this it can be seen that the original rate of the user data, R, has been increased to 8*R after the spreading code has been applied.

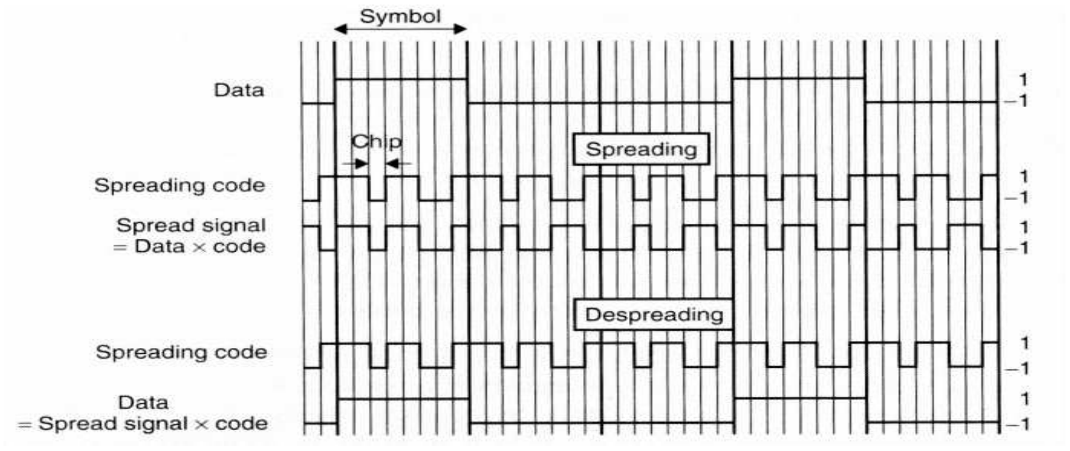


Figure 2: Spreading and despreading in UMTS

For UMTS a fixed chip rate of 3.84 Mchip/s is used, from this it can be seen that it is possible to alter the transmission rate by increasing or decreasing the number of chips user per bit of user data. This is known as the spreading factor and is a key factor in providing Bandwidth on Demand (BoD) i.e.

$$\begin{aligned} spreading\ factor = \frac{Chipe\ rate}{data\ rate} \quad &= \frac{3840k}{15k} = 256 \\ &= \frac{3840k}{480k} = 8 \end{aligned}$$

From this it can be seen that it is possible to utilise data rates in the range 15-480Kb/s utilising between 8-256 chips per user symbol. As the inclusion of this spreading factor increases the bandwidth of the data these systems are normally referred to as wide-band systems.

The individual signals from each UE can then be separated by making use of a correlation receiver which makes use of the same spreading code to determine the correct data sequence. In this type of operation it is vital to ensure that the spreading code at the receiver is perfectly synchronised with that if the UE otherwise reception errors will occur. In addition to spreading, scrambling is also utilized in the transmitter and as such will be briefly described here, a more detailed discussion can be found in [3, 6].

Although it is possible to separate signals that have been spread using different spreading codes, scrambling is used on top of spreading to allow identical spreading codes to be used by a number of transmitters. Scrambling does not alter the bandwidth but simply allows signals from different sources to be separable through the use of long Gold codes and shorter S2 codes.

3 Radio Access Network

In order to allow a more open standard, a number of elements have been described which make up the UMTS network. These are divided into the Terrestrial Radio Access Network (TRAN) which is responsible for the radio related functionality and the core network (CN) which is concerned with the switching and routing within the UMTS network as well as connections to external networks. The standard also specifies the UE which connects the user to the radio resource. A block diagram showing the interfaces between each of these groups is shown in Figure 3

In UMTS only the UTRAN and the UE are new protocols where as, the CN has been adopted from GSM. This allows the system to make use of proven CN technology and also allow backward compatibility with GSM systems.

The UMTS standard is designed to allow the implementation of each of these elements to be determined by the designer and hence limits itself to a detailed definition of the interfaces between these elements.

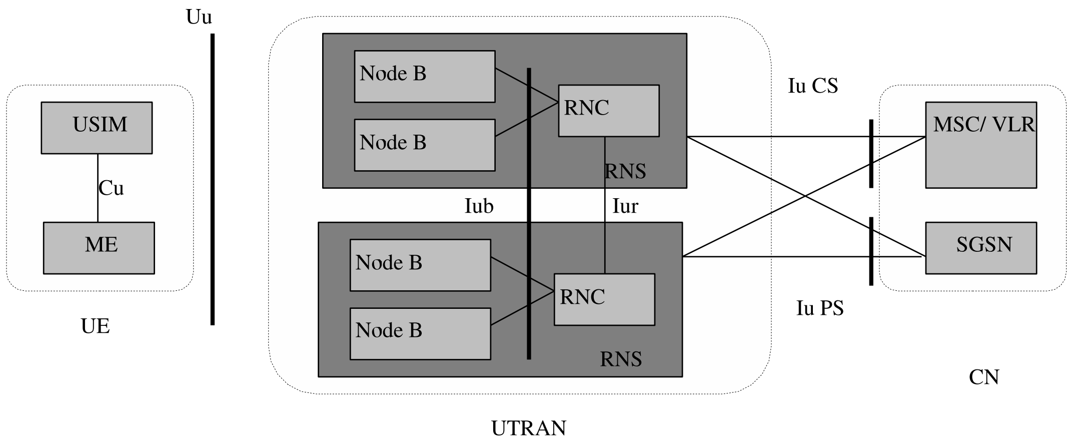


Figure 3: UTRAN architecture

- Cu interface: This is the electrical interface between the USIM smartcard and the mobile equipment (ME). This allows the ME to access the user data stored on the USIM and follows the standard format for smartcards.
- Uu interface: This is the WCDMA air interface and allows the UE to access the fixed part of the network. This is the main component of the 3GPP standard.
- Iu interface: This connects the TRAN to the CN.
- Iur interface: This is the interface between the Radio Network Controller (RNC) and allows soft-handover between different cells within the UMTS network.
- Iub Interface: This is the interface between node B (base station) and the RNC.

The specification of these interfaces allows any manufacturer to develop components for the UMTS network and hence increase competition in this area.

4 UMTS protocol stack

The overall radio interface protocol architecture [1] is shown in 4. It can be seen that the radio link layer (layer 2 of OSI model) has been split into two sub-layers:

the Media Access Control (MAC) and the Radio Link Controller (RLC). In the user plane two other service-dependent protocols are also included: Packet Data Coverage Protocol (PDCP) and the Broadcast/Multicast Control Protocol (BMC). The layer 3 functionality is implemented in the control plane using the Radio Resource Control (RRC).

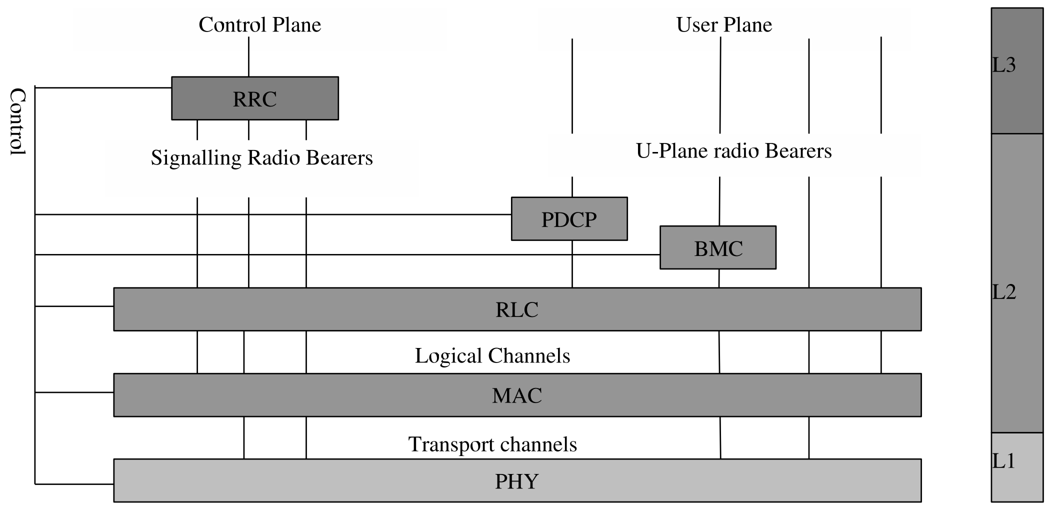


Figure 4: UTRAN FDD Radio Interface Architecture

4.1 Physical layer

As the Physical layer directly affects the achievable performance of communications between the base station and the terminal much of the 3GPP specification is devoted to this layer. The Physical layer is responsible for the mapping of user data contained in the transport channels to the physical channels contained in the physical layer.

4.1.1 Transport channels

In order to support variable bit rate and QoS, two types of transport channel exist: the dedicated channel and the common channel. The dedicated channel, identified by a code, is reserved for a single user, where as the common channels are used by all users in a given cell. The following section gives general information on these channels and the mapping between the transport and the physical channels.

4.1.1.1 Dedicated Transport Channel

3GPP has defined a single dedicated transport channel (DCH) for the transportation of all information intended for the user, coming from the higher layers, including data and control information. The dedicated transport channel, DCH, is mapped to two physical channels: the Dedicated Physical Data Channel (DPDCH) and the Dedicated Physical Control Channel (DPCCH). The DPCCH is used to transport control information generated at the physical layer and the DPDCH transports user data from the upper layers.

The DCH is characterized by its use of fast power control, variable data rate on a frame by frame basis, and the ability to make use of adaptive antennas for transmission to particular areas within a cell.

4.1.1.2 Common Transport Channels

3GPP have specified a number of Common transport channels. The characteristics of each of these channels are discussed below

Broadcast Channel (BCH) The broadcast channel (BCH) is used to transmit information specific to a given Cell or UTRAN, this could include the available access codes and slots. As this channel must be received by all UE attempting to register with the network this channel must be available throughout the cell and have a low enough data rate to allow all terminals in the cell to decode this information.

Forward Access Channel (FACH) The FACH is used to transmit control information to a UE located within a given cell. Although it is possible for a cell to contain multiple FACH, one of these must be high enough power and have a low enough data rate to be received by all UE in a cell. The FACH does not use fast power control and any messages transmitted must contain identification information to ensure reception by the correct UE.

Paging Channel (PCH) The PCH is a downlink transport channel used to transmit required data to the UE when the network wishes to initiate communication. As any UE in standby mode must *wake-up* and check this channel for paging messages at regular intervals this channel has a significant effect on UE battery life. This signal must also be available for reception in the entire cell.

Random Access Channel (RACH) The RACH is an uplink channel used to transmit control information from the UE, such as requests to set-up a connection, and can also be used to send small amounts of user data. The random-access transmission is based on a slotted ALOHA approach with fast acquisition indication [2]. This channel must be available to all UEs in a cell and hence must have fairly high transmission power and a low data rate.

Uplink Common Packet Channel (CPCH) The CPCH is an extension to the RACH and is intended to transmit packet-based user data in the uplink direction. The CPCH transmission is based on DSMA-CD with fast acquisition indication. The CPCH employs fast power control and transmissions may last for several frames.

Downlink Shared Channel (DSCH) The DSCH is intended to carry dedicated user data and/or control information and can be shared by multiple UEs. The DSCH employs fast power control and supports variable bit rates on a frame by frame basis. The downlink shared channel is always associated with a DCH.

The RACH, FACH and PCH are required for the basic network operation while the use of the DCSH and CPCH are optional and can be decided by the network. The diagram in Figure 5 shows the mapping of the transport channels to their associated physical channels. From this it can be seen that several channels are introduced by the physical layer itself, the synchronization channel (SCH), the Common Pilot Channel (CPICH) and the Acquisition Indication channel (AICH), as such these channels are not directly visible to the upper layers. In order to operate the CPCH it is also necessary to introduce the CPCH status Indication Channel (CSICH) and the Collision Detection/Channel Assignment Indication channel (CD/CA-ICH).

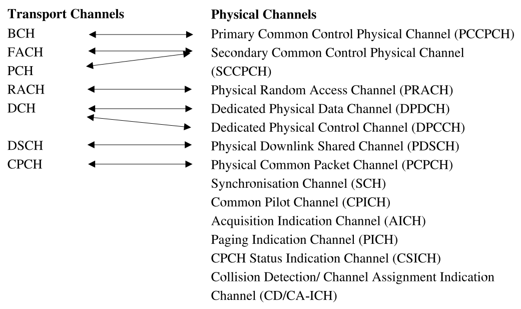

Transport Channels
BCH
FACH
PCH
RACH
DCH
DSCH
CPCH
Physical Channels
Primary Common Control Physical Channel (PCCPCH)
Secondary Common Control Physical Channel (SCCPCH)
Physical Random Access Channel (PRACH)
Dedicated Physical Data Channel (DPDCH)
Dedicated Physical Control Channel (DPCCH)
Physical Downlink Shared Channel (PDSCH)
Physical Common Packet Channel (PCPCH)
Synchronisation Channel (SCH)
Common Pilot Channel (CPICH)
Acquisition Indication Channel (AICH)
Paging Indication Channel (PICH)
CPCH Status Indication Channel (CSICH)
Collision Detection/ Channel Assignment Indication Channel (CD/CA-ICH)

Figure 5: Mapping of Transport Channels to Physical Channels

4.1.2 User Data Transmission

The following section aims to give an overview of user data transmission on the physical channels in both the uplink and downlink directions.

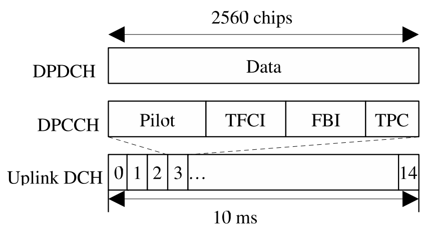


Figure 6: Uplink Dedicated Channel structure

4.1.2.1 Uplink Dedicated channel The Uplink Dedicated Channel is made up of a single DPCCH and up to 6 DPDCH channels, these channels are I-Q/code multiplexed and then transmitted in a single slot using the format shown in Figure 6 The pilot bits are used for channel estimation and the Transmit Power Control (TPC) bits carry downlink power control information. The Transport Format Control Indicator (TFCI) is used to inform the receiver which channels are in use and provide information such as the coding rate and the spreading factor used. The overall process performed during the transmission of user data on the uplink-dedicated channel is shown in Figure 7

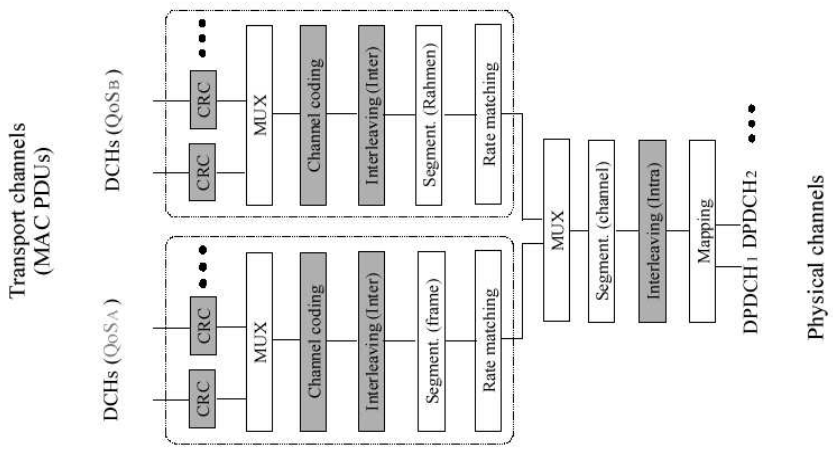


Figure 7: Uplink Mulitplexing and channel coding chain

After receiving a transport block from higher layers a CRC is attached to allow error detection at the physical layer. The CRC is variable length (0, 8, 12, 16 or 24 bits) and can be altered depending on the error detection required by the block. After CRC attachment the blocks are either concatenated together or segmented into different blocks. This allows a standard block size to be used as defined by the channel coding method. Radio Frame equilisation is then performed to ensure that data can be divided evenly when transmitted over multiple 10ms slots. This is achieved through the use of padding bits. After interleaving and segmentation is performed rate matching is used to ensure that the number of bits to be transmitted matches the number available in a single frame. This is achieved either by the use of puncturing or repetition. The different transport channels are then multiplexed together by the transport channel multiplexer. The transport channels provide 10ms blocks of data with are serially multiplexed on a frame-by-frame basis. After a second interleaving operation is performed the data is mapped onto the relevant physical transport channel.

4.1.2.2 Data transmission on the RACH It is also possible to transmit user data on the RACH, which is mapped to the Physical RACH (PRACH). This is intended for low data rate packet data where it is not necessary to ensure continuous connection. The PRACH first transmits a preamble and signature sequence and waits for acknowledgment on the AICH; once this is received it is possible to transmit a 10ms message. It should be noted that a fixed spreading factor of 256 is used for the preamble but the message can use a spreading factor of anywhere between 256 and 4.

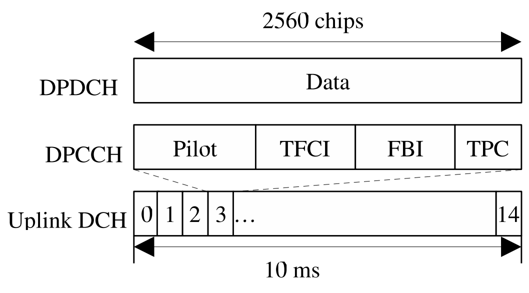


Figure 8: Downlink Dedicated Physical Channel

4.1.2.3 Downlink Dedicated Channel The downlink Dedicated Channel is transmitted on the Downlink Dedicated Physical Channel (DDPCH). The DDPCH applies time multiplexing of the DPDCH channels and the DPCCH channel as shown in Figure 8 The overall channel coding chain is shown in Figure 9, from this it can be seen that extra stages are required in the DDPCH, these are necessary as the spreading factor for the highest data rate determines which channelisation code will be reserved. This means that for lower data rates it is necessary to employ Discontinues Transmission (DTX) by gateing the transmission on/off. It should be noted that this technique is not used in the uplink due to audible interference cause by DTX techniques.

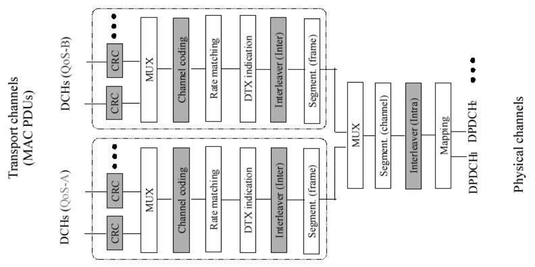


Figure 9: Downlink multiplexing and channel coding chain

4.2 Media Access Control (MAC) layer

The MAC layer is responsible for the mapping of logical channels from the upper layers to the transport channels. It is also responsible for selecting the appropriate Transport Format (TF) for each transport channel, depending on the required data rate, with respect to the Transport Format Combination Set (TFCS).

The MAC layer consists of the logic entities shown in Figure 10 [4] with the addition of the MAC-b entity. In the downlink if dedicated channels are mapped

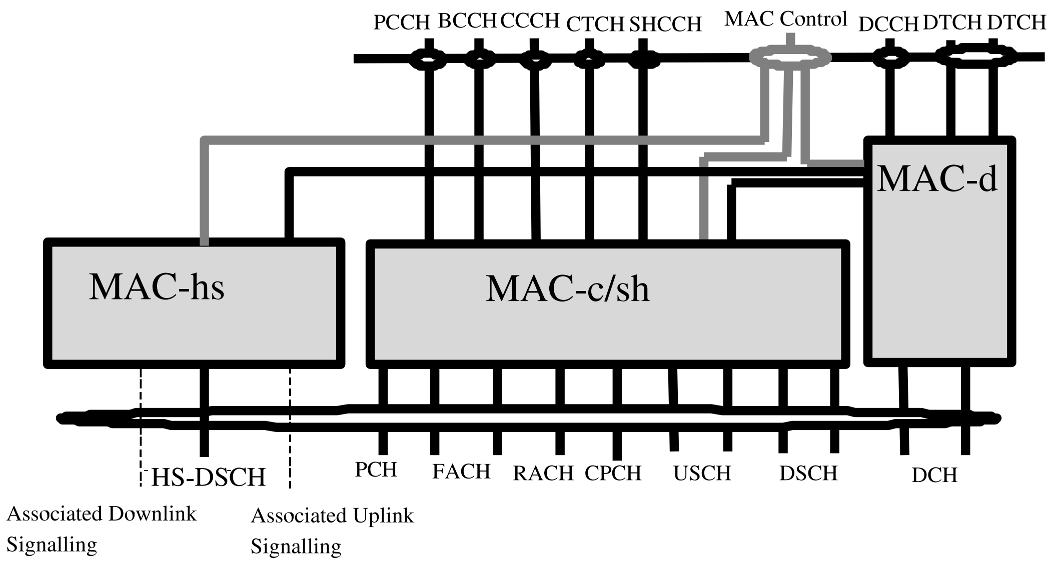


Figure 10: UE side MAC Architecture

to the common channels the MAC-d entity receives data from the MAC-hs or the MAC-c/sh channels via the connections shown in Figure 10. In uplink, if it is required that the dedicated channels be mapped to the common channels as determined by the RRC this is again achieved using the connections show. The MAC entities are assigned the following functionality.

- MAC-b is the MAC entity that handles the BCH channel.
- MAC- c/sh is the MAC entity that handles the following transport channels:
- Paging channel (PCH).
- Forward access channel (FACH).
- Random access channel (RACH).
- Common packet channel (UL CPCH) (exists only on FDD mode).
- Downlink shared channel (DSCH).
- Uplink shared channel (USH) (exists only in TDD mode).

- MAC-d is the MAC entity responsible for the Dedicated Transport Channel (DCH)
- MAC-hs is the MAC entity responsible for High Speed Downlink Shared Channel (HS-DSCH)

The following sections give a detailed description of each of these entities from the UE side only, a detailed description of the MAC entities in the Node-B can be found in [4, 6].

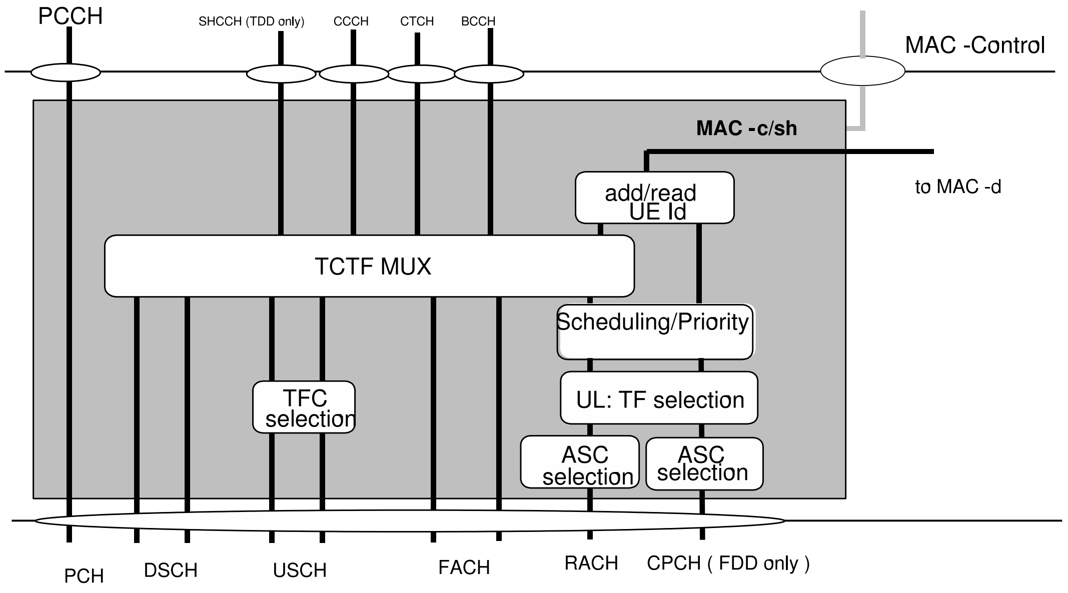


Figure 11: UE side MAC Architecture- MAC-c/sh details

4.2.1 MAC-c/sh entity

The structure of this entity is shown in Figure 11. The RLC provides RLC-PDUs to the MAC that fit into the available transport blocks on the transport channels.

The MAC-c/sh entity performs the following functions:

- Target Channel Type Field (TCTF) MUX
  - This function handles the insertion for uplink and detection and deletion for downlink of the TCTF field in the MAC header. This entity also handles the mapping between logical and transport channels as the TCTF field indicates if the common logical or a dedicated channel is to be used.
- Add/read UE id
  - The UE is added for transmissions on the CPCH and the RACH.
  - The UE id, when present, indicates data to this UE.
- Uplink (UL) transport format (TF) selection
  - In the case of CPCH transmission the TF is based on TF availability determined from status information on the CPCH Status Indication Channel (CSICH).
- Access Service Class (ASC) selection
  - For RACH and CPCH the MAC indicates the ASC associated with the PDU to the physical layer. This is to ensure that RACH and CPCH messages associated with a given ASC are sent on the appropriate signature and time slot. **The MAC also applies the appropriate back-off parameter associated with the ASC.**
- Scheduling/priority handling
  - This functionality is used to transmit information on the RACH and CPCH based on the logical channels priorities and is related to the TF selection.

- TFC selection
  - Transport format and transport format combination selection according to the Transport format combination set.

4.2.2 MAC-d entity

The MAC-d entity is shown Figure 12. The MAC-d entity is responsible for mapping dedicated logical channels for the uplink to the dedicated transport channels or the transfer of data to the MAC-c/sh to be transmitted via the common channels. The MAC-d has connections to the MAC-c/sh and MAC-hs entities to allow data to be transmitted or received on the common or high speed channels.

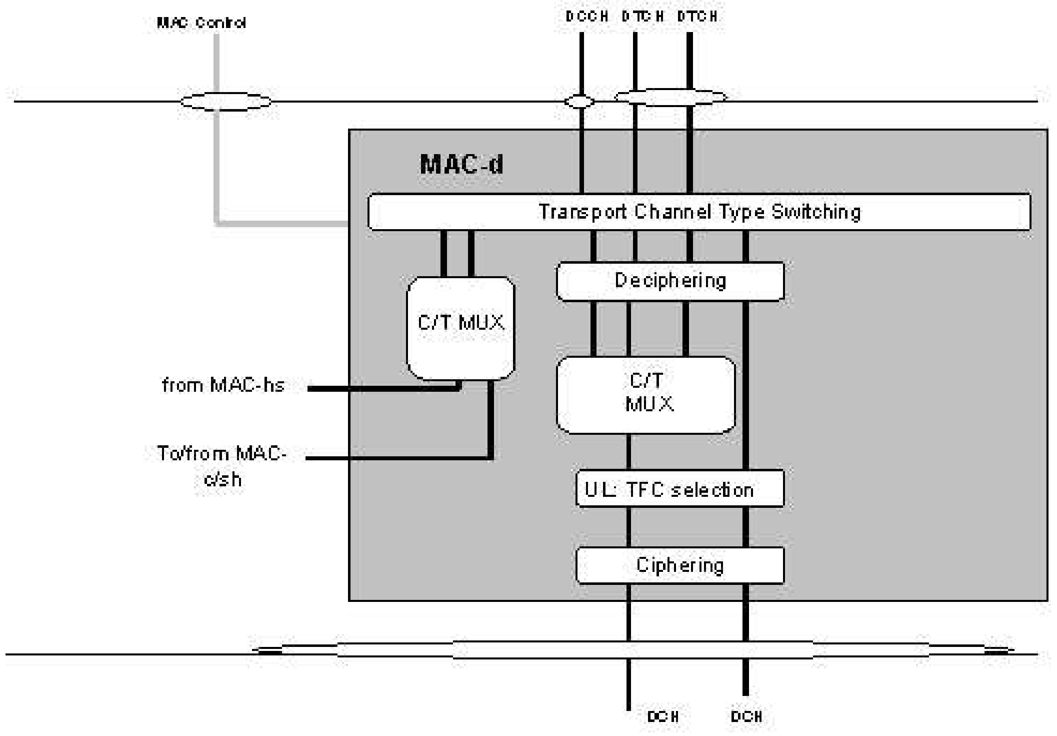


Figure 12: UE side MAC Architecture- MAC-d details

The MAC-d performs the following functions

- Transport channel type switching
  - Transport channel switching is performed by this entity based on decisions taken by RRC. If requested the MAC will switch the mapping of the logical channels between common, high speed and dedicated channels.
- C/T MUX
  - The C/T MUX is used when multiplexing several dedicated logical channels onto a single transport channel.
- Ciphering
  - The MAC-d performs any ciphering of transparent mode data as required.
- Deciphering
  - The MAC-d performs any deciphering of ciphered transparent mode data as required.
- UL TFC selection
  - TF and TFC selection according to the TFCS configured by RRC is performed.

4.2.3 MAC-hs

The MAC-hs handles the High-Speed Downlink Packet Access (HSDPA) specific functions. As shown in figure 13 the MAC-hs comprises of the following elements:

- HARQ: The HARQ entity is responsible for handling the MAC functions relating to the Hybrid Automatic Repeat Request (HARQ) protocol. The detailed

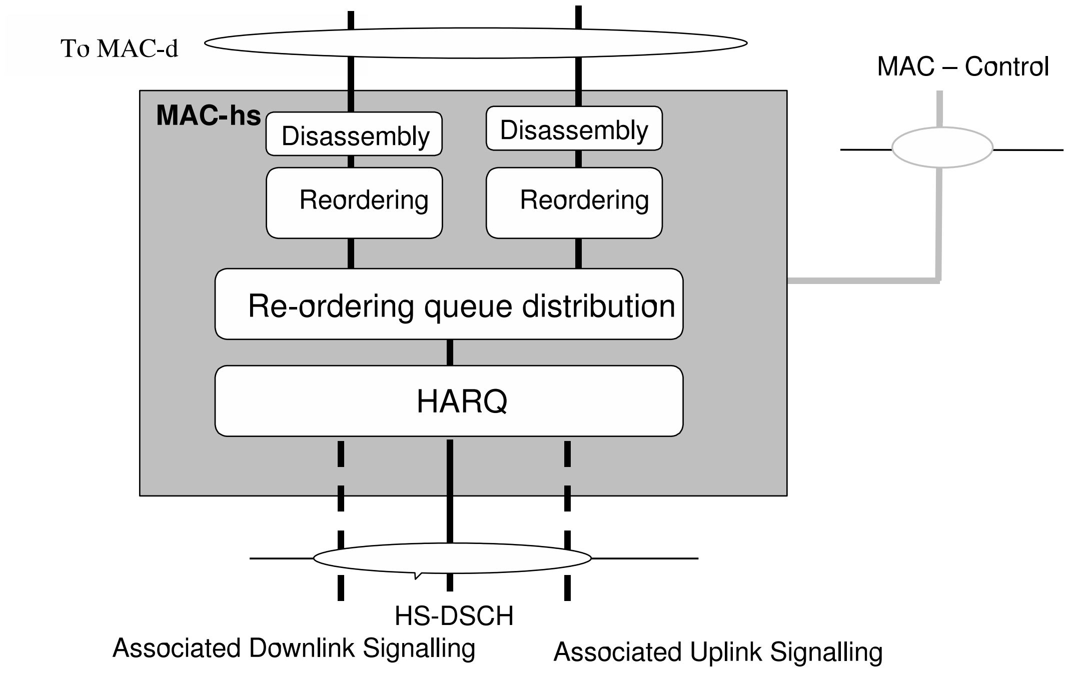

To MAC-d
MAC – Control
MAC-hs
Disassembly
Disassembly
Reordering
Reordering
Re-ordering queue distribution
HARQ
HS-DSCH
Associated Downlink Signalling
Associated Uplink Signalling

Figure 13: UE side MAC Architecture- MAC-hs details

configuration of the HARQ protocol is provided by the RRC over the MAC-Control SAP.

- Reordering Queue distribution: This function routes the MAC-hs PDUs to the correct reordering buffer based on the Queue ID.
- Reordering: The reordering entity reorders received MAC-hs PDUs according to the Transmission Sequence Number (TSN). PDUs with consecutive TSNs are delivered to the disassembly function upon reception. PDUs with non-consecutive TSNs are not delivered until any missing PDUs are received.
- Disassembly: This entity is responsible for the removal of MAC-hs headers and any padding bits. The PDUs are then delivered to the upper layers.

4.2.4 Mapping of Logical channels to Transport Channels

The data transfer services of the MAC layer are provided on logical channels. The logical channels are divided into two broad categories: Control channels and traffic channels. Control channels are used to transfer control plane information and traffic channels for user plane information.

The control channels are:

- Broadcast Control Channel (BCCH). A downlink channel for broadcast system control information.
- Paging Control Channel (PCCH). A downlink channel used to transfer paging information.
- Dedicated Control Channel (DCCH). A point-to-point bi-directional channel used to transmit dedicated control information between the UE to the RNC.
- Common Control Channel (CCCH). A bi-directional channel for transmission of control information between the UE and the network. This channel is always mapped to the RACH or the FACH transport channels.

The traffic channels are:

- Dedicated Traffic Channel (DTCH). A point-to-point channel dedicated to one UE for the transfer of user information.
- Common Traffic Channel (CTCH). A point-to-multipoint downlink channel for transfer of dedicated user information for all or a group of specified UEs.

The mapping of these logical channels to their respective transport channels is shown in Figure 14 Where it is possible for a logical channel to be mapped to multiple transport channels or vice versa, the mapping is determined by the RRC.

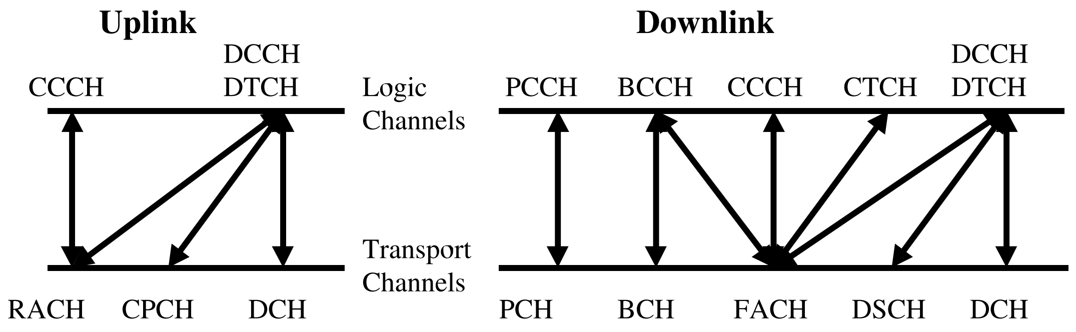


Figure 14: Mapping of Logic channels to Transport channels

4.3 Radio Link Control Protocol (RLC)

The RLC provides segmentation and retransmission of user and control data. The RLC is controlled by the Radio Resource Controller (RRC) and can operate in one of three modes: transparent mode (Tr), unacknowledged mode (UM) or acknowledge mode (AM). In the control plane the services of the RLC are termed Signaling Radio Bearers (SRB) where as in the user plane, along with the PDCP and BMC, the RLC provides Radio Bearer (RB) services.

The RLC architecture is shown in Figure 15. The RLC entities connect the RLC-Service Access Points (SAPs) to the logical channels which are in turn connected to the MAC-SAPs as shown. For all RLC modes CRC checks are done at the physical layer and the results as well as the data are passed to the RLC.

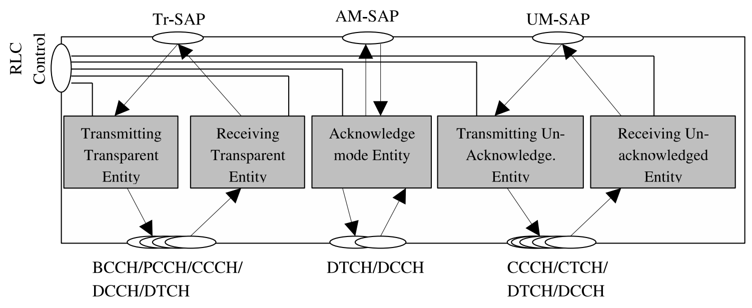

Tr-SAP
AM-SAP
UM-SAP
RLC
Control
Transmitting Transparent Entity
Receiving Transparent Entity
Acknowledge mode Entity
Transmitting Un-Acknowledge. Entity
Receiving Un-acknowledged Entity
BCCH/PCCH/CCCH/
DCCH/DTCH
DTCH/DCCH
CCCH/CTCH/
DTCH/DCCH

Figure 15: RLC layer Architecture

4.3.1 RLC transparent mode (Tr)

In Tr mode, no protocol overhead is added to higher layer data by the RLC. Erroneous protocol data units (PDUs) can be either marked or discarded by this layer. The RLC can provide data streaming where data is passed to the MAC layer with no segmented.

4.3.2 RLC Unacknowledged mode (UM)

In UM mode no retransmission and no delivery guarantees are provided. Again, erroneous PDUs can be either marked or discarded. On the transmit side a timer based discard is used with no explicit signaling to upper layers, this means that RLC-SDUs not transmitted within a given time limit are deleted from the buffer. Segmentation and concatenation of upper layer data is possible through the addition of a sequence number in the RLC header field. This allows large upper layer data units to be broken down into smaller RLC-PDUs for transmission and the integrity of upper layer PDUs to be observed. User services that utilize UM are the cell broadcast service and voice over IP (VoIP).

4.3.3 RLC Acknowledge mode (AM)

In AM mode an automatic repeat request (ARQ) mechanism is used for error detection. The transmitting entity is signaled if a PDU was not received correctly and will then attempt retransmission. The quality and delay performance of the RLC can be controlled by the RRC through configuration of the number of retransmission before PDU is discarded and the upper layers notified. When a PDU is discarded the peer RLC entity is also notified to allow the associated PDUs stored in the buffer to be discarded. This notification can be achieved by *piggy-backing* status information on the back of PDUs. This is possible when the payload of the PDU has not been completely filled by the upper layer data, rather than inserting padding bits the RLC inserts status information. In this mode the RLC can be configured to deliver upper layer PDUs in-sequence, order of PDUs is maintained, or out-of-sequence, where PDUs are delivered to upper layers as soon as they are completely received.

4.3.4 Functions provided by RLC

The functions provided by the RLC are:

- Segmentation and reassembly. Higher layer PDUs converted to/from smaller RLC payload units (PUs) for transmission and then reassembled at receiver side. The size of this RLC-PDU can be adapted to the valid Transport Format Set (TFS) of the MAC layer.

- Concatenation. If the RLC-SDU does not fit into an integral number of PUs, the first segment of the next SDU may be concatenated with the last segment of the previous SDU.

- Padding. When concatenation is not possible, any remaining space in the PU is filled with padding bits. As mentioned previously, it is also possible to use this space to transmit statues information in AM mode.

- Error detection. When SDUs are segmented, the PDUs are numbered consecutively. This allows the peer RLC to determine the integrity of the received SDU and allows erroneous SDUs to be discarded.

- Error correction. In AM error correction is provided through retransmission of erroneous PDUs.

- In sequence delivery of higher layer SDUs. RLC-SDUs are passed to upper layers in the order they were received by the peer RLC. If this function is not used RLC-SDUs are passed as soon as they are received with no errors.

- Duplication detection. RLC detects duplicated RLC-SDUs and ensures they are passed only once to upper layers.
- Flow control. The RLC can control the rate at which peer RLCs transmit data.
- Ciphering. This is performed in AM and UM modes and utilizes the same algorithm used by the MAC layer.
- Suspend/resume data transfer. This function is necessary during the security mode control procedure to ensure that the same ciphering keys are used by peer entities. The suspension and resumption operation are controlled by the RRC through the control interface.

4.4 The Radio Resource Control (RRC) Protocol

RRC control messages account for the majority of control signaling between the UE and the TRAN. These control messages contain all the information required to set-up, modify and release all layer 1 and 2 protocol entities. The payload of the RRC control messages can also contain higher layer signaling (MM, CM, SM). The RRC also controls the mobility of UEs when in connected mode i.e. measurements, handover etc.

4.4.1 RRC layer architecture

The logical architecture of the RRC layer is shown in Figure 16. The RRC layer is made up of the four following logical entities.

- Dedicated Control Function Entity (DCFE). This function handles all functions and signaling specific to one UE. In this mode the RRC mostly utilises acknowledge mode RLC (AM-SAP) but can also make use of the Unacknowledged mode (UM) and Transparent modes (Tr).

- Paging and Notification control Functional Entity (PNFE). The function handles the paging of idle mode UEs and makes use of the PCCH via the Tr-SAP of the RLC.

- Broadcast Control Function Entity (BCFE). This function handles the broadcasting of system information and uses either the BCCH or FACH via the Tr-Sap of the RLC.

- Routing Function Entity (RFE). This function handles the routing of higher layer messages to the different MM/CM entities on the UE side and different core network domains on the UTRAN side. All higher layer messages are piggybacked into one of three RRC direct transfer messages; initial direct transfer (uplink), uplink direct transfer or downlink direct transfer.

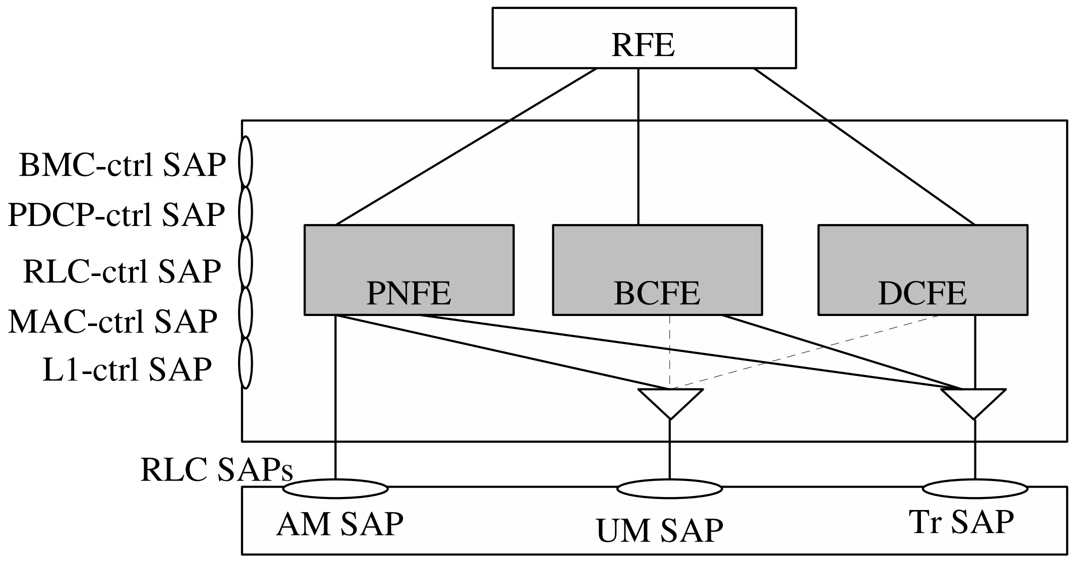


Figure 16: RRC layer Architecture

4.4.2 RRC service states

Each UE may operate in either of two basic modes, idle mode and connected mode. In connected mode the UE may be in one of four sub-states depending on which type of physical channels the UE is using. Figure 17 shows the main RRC states as well as the transitions between these states. The diagram also shows the possible transitions for a multimode terminal. In this case the terminal can perform inter-system handovers between GSM/GPRS and UMTS.

- Idle mode. After UE switch on, it selects which Public Land Mobile Network (PLMN) to contact and looks for suitable cell, the UE then tunes into the control channel (this is known as camping on a cell). The UE can now receive system information and cell broadcast messages. The UE remains in idle mode until it transmits a request to establish an RRC connection.

- Cell_DCH. In this state a dedicated channel is allocated to a UE and the UE is known to the SRNC on a cell or active set location. The UE performs measurements and gives results according to information from the RNC.

- Cell_FACH. In this state no dedicated channel is allocated to the UE, as such the UE utilises the FACH and RACH to transmit signaling data and small amounts of user data. In this state the UE may also monitor the BCH for system information and the CPCH can also be used when instructed by the UTRAN. The UE performs cell reselection and notifies the RNC to allow for changes in routing.

- Cell_PCH. In this state the UE is still known on a cell level by the SRNC but can only be reached by the PCH. The UE may also monitor the BCH to receive system information and perform cell reselection.

- URA_PCH. This mode is similar to Cell_PCH mode but no cell reselection is performed.

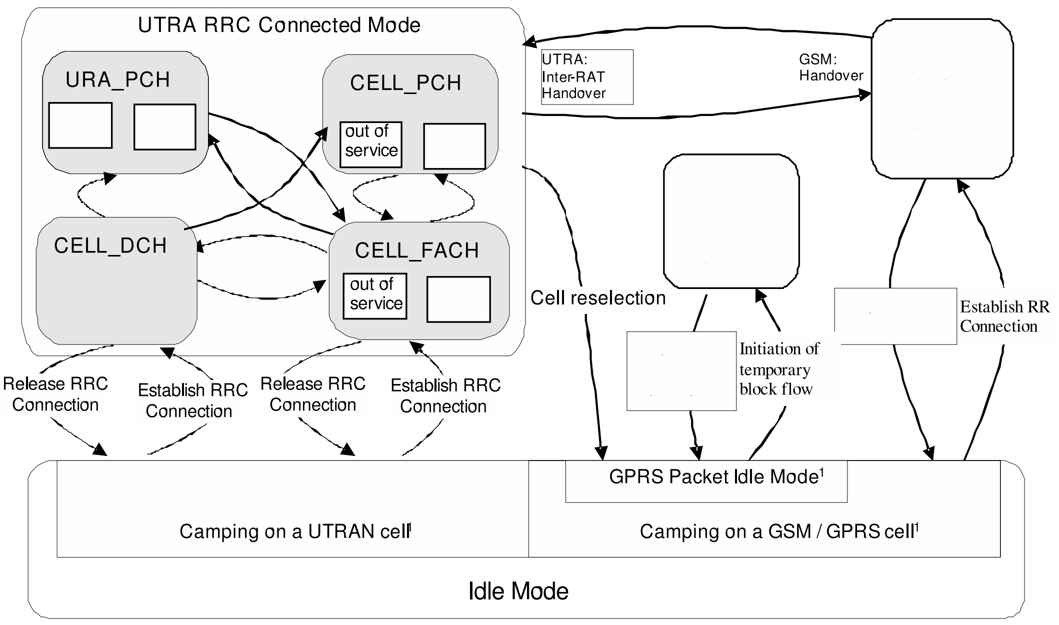


Figure 17: RRC states and state transitions including GSM/GPRS

4.4.3 RRC functions and Signaling procedures

As the RRC performs the majority of signaling between the UE and the UTRAN it is required to perform a large number of functions. The following section gives a list of each of these functions with a short description, a more detail description can be found in [5, 6].

- **Broadcast system information**. The broadcast system information comes from the core network, the RNC and from node B's. This system information is contained in system information messages that are transmitted on the BCCH logical channel mapped to the BCH or FACH. A system information message carries system information blocks (SIBs) which group together system information of the same nature i.e. static/dynamic information.

- **Paging**. The RRC layer can broadcast paging information on the PCCH. This paging procedure can be used for three purposes:
  - Core network-originated call or session setup. A request is made by the core network to begin the paging process.
  - To change the UE state from Cell_PCH or URA_PCH to Cell_FACH.
  - Indicate a change in system information. This message is sent to all UE in cell.
- **Initial cell selection and reselection in idle mode**. The most suitable cell is selected based on idle mode measurements and the cell selection criteria.
- **Establishment, maintenance and release of RRC connections**. The establishment of RRC connections and signaling radio bearers (SRB) between UE and UTRAN is initiated by requests from upper layers on the UE side. In the case of the network side, establishment is preceded by an RRC paging message. The RRC establishment procedure is only utilised when the UE is in idle mode i.e. no RRC connection present.
- **Control of Radio Bearers, transport channels and physical channels**. At establishment and reconfiguration of Radio Bearers, the UTRAN (RNC) performs admission control and selects parameters describing the radio bearer processing in layer 1 and 2. SRBs are normally setup during RRC registration but can also be controlled with normal radio bearer procedures.
- **Control of security functions**. The RRC security mode control is used to start Ciphering and integrity protection between UE and UTRAN and to trigger the change of the ciphering keys during connection. Ciphering is used on RLC-UM and RLC-AM, for RLC-Tr ciphering is performed by the MAC layer.

- **Integrity protection of signaling**. The RRC inserts a 32-bit integrity checksum, called Message Authentication Code (MAC-I), into most RRC PDUs. The checksum is calculated using the UMTS Integrity Algorithm (UIA) which makes use of a 128-bit integrity key (IK) that is generated along with the ciphering key (CK). Any messages with missing or incorrect MAC-I are discarded by the RRC.

- **UE measurement and control**. UE measurements are controlled the RNC using RNC protocol messages, these messages contain information such as what and when to measure, as well as how the results should be reported.

- **RRC Connection Mobility Functions**. These functions allow the UTRAN to keep track of a UEs location, on a cell or active set level, while in connected mode. When dedicated channels are allocated to the UE mobility control is performed using Active Set update and hard handover procedures. In order to support mobility the following functions are defined:

  - **Active Set Update.** This function updates the set of connections between UE and UTRAN while UE is in Cell_DCH state and can perform Radio Link addition, subtraction or combined addition and subtraction.
  - **Hard Handover.** This function can be used to change the radio frequency band of the connection between the UE and UTRAN or between FDD and TDD modes.
  - **Inter-system handover to/from UTRAN. This function allows the handover to/from UTRAN from/to another radio access system. The specifications support handover to GSM/GPRS, CDMA2000 and PCS 1900. This function may be used in Cell_DCH or Cell_FACH states. The UE receives non-UTRAN cell parameters either on system information or in a measurement control**

message, based on the measurement report from the UE the RNC makes the handover decision. This process is shown in Figure 18 for a handover from UTRAN to GSM.

- **Inter-system cell reselection to/from UTRAN. These procedures allow the UE to switch connections to/from UTRAN to another radio access system**
- **Inter-system cell change order to/from UTRAN. These procedures are used by the UTRAN or other radio access systems to order the UE to switch to/from the UTRAN from/to another radio access system.**
- **Cell Update.** The cell update procedure can be triggered by a number of factors including cell reselections, expiry of cell update timer or UTRAN originated paging. The Cell update Confirm message sent by the UTRAN can include mobility information and general control information.
- **URA Update.** The UTRAN registration area (URA) update procedure is used in the Cell_PCH state. As the UE may not send any information in this state it is required to switch to the Cell-FACH state to execute the signaling procedure.

• **Open loop Power control**. Prior to PRACH transmission the RRC calculates the power for the first preamble using information contained in system information messages and measurements performed by the UE. The required power is then recalculated if any of the parameters are altered. The RRC also calculates the required power of transmissions on the DPCCH, again this is calculated using system information as well as measurements performed by the UE.

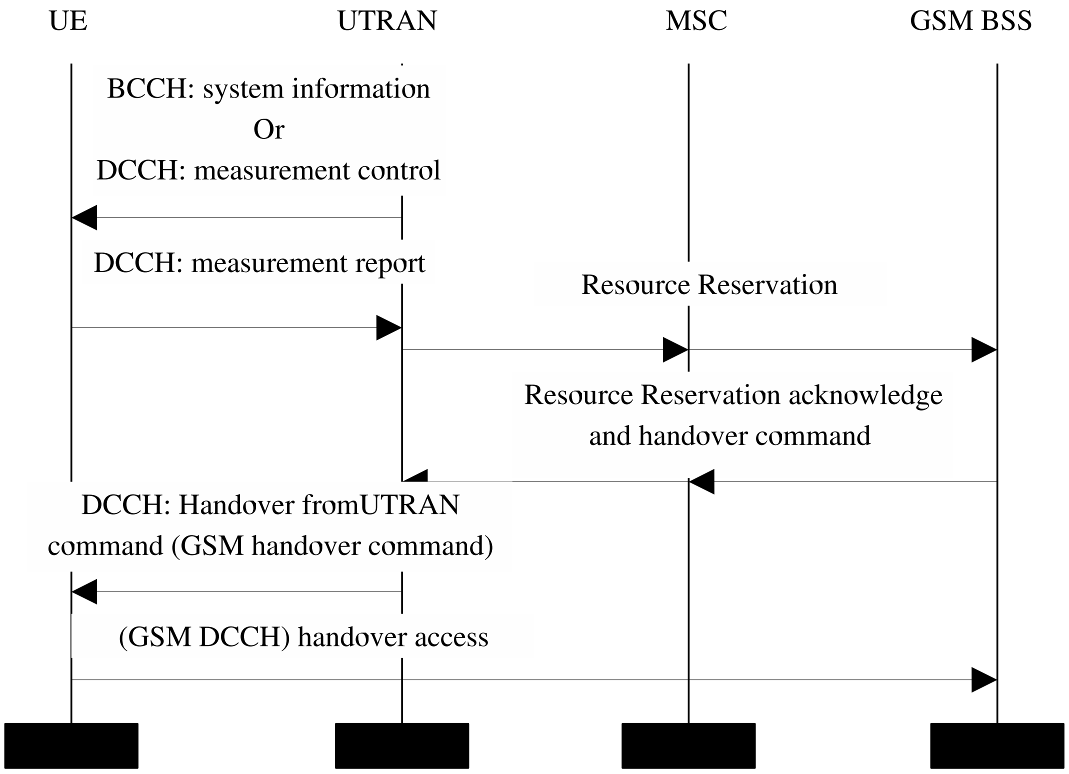


Figure 18: Inter-system Handover procedure from UTRAN to GSM

References

[1] 3GPP. Ts25.301 Radio Interface Protocol Architecture, Dec. 2001.

[2] 3GPP. Ts25.211 Physical channels and mapping of transport channels onto physical channels (FDD), March 2002.

[3] 3GPP. Ts25.212 Multiplexing and channel coding (FDD), March 2002.

[4] 3GPP. Ts25.321 MAC protocol specification, March 2002.

[5] 3GPP. Ts25.331 Radio Resource Control (RRC) protocol specification, March 2002.

[6] H. Holma and A. Toskala. *WCDMA for UMTS: Radio Access For Third Generation Mobile Communication, 3rd Edition.* Wiley, 2004.

.

Portfolio Document A3

IEEE 802.11: Protocol Review

Author:
Graeme Milligan

Supervisor:
Wim Vanderbauwhede

Contents

List of Figures

1 Introduction

This report is intended to give a short review of the IEEE 802.11 wireless LAN standard [3]. This is intended to allow the comparison of this standard to the UMTS 3G mobile communication standard [1].

The IEEE 802.11 standard is a member of the 802 family of local and metropolitan network standards. This family includes the 802.3 standard, or Ethernet, widely used in modern computer networks. This allows 802.11 to supply wireless networking services that can be easily integrated into most modern networks. The relationship between the 802 family of protocols is shown in Figure 1. As this figure shows the 802 family of standards limits itself to specifying only the two lowest layers of the OSI architecture. The DLL has also been split into two sub-layers, namely the Medium access control (MAC) and the Logical Link Control (LLC) layers.

In order to allow for portability the 802.11 standard is limited specifying to the required functionality to establish, maintain and finally terminate a wireless link. This allows the upper layers to operate with no knowledge of the medium being used for communication i.e. upper layers will not know if a fixed wired system (e.g. 802.3) or a wireless system (e.g. 802.11) is being utilized.

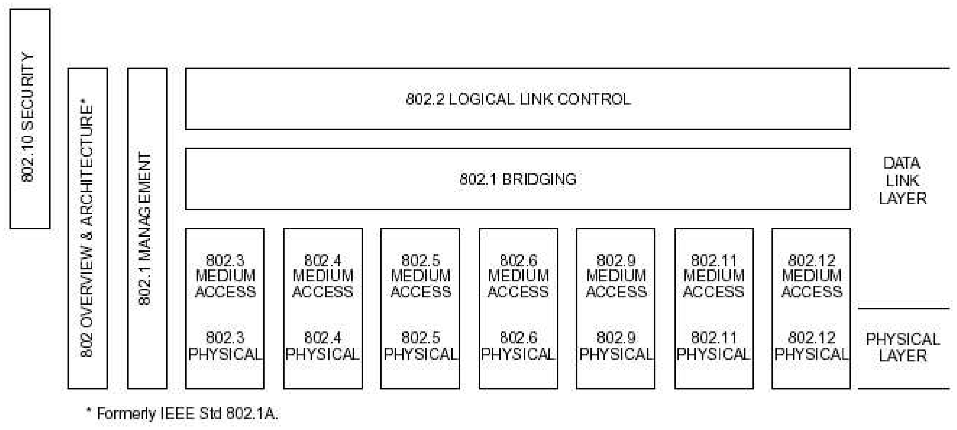


Figure 1: ISO 802 Family of Local and Metropolitan network standards

2 IEEE 802.11 Network Architecture

The basic element of the 802.11 network is the Basic Service Set (BSS). A BSS consists of more than one station (STA) controlled by a single Coordination function (CF). In 802.11 there are two possible CFâs; the mandatory Distributed Coordination Function (DCF) and the optional Point Coordination Function (PCF).

The most basic network available to 802.11 enabled devices is the Independent BSS (IBSS). This is an ad-hoc network containing a minimum of two STAs. By connecting multiple BSS it is possible to create an Extended Service Set (ESS), this is a group of BSS interconnected by a Distribution service (DS). STAs that are directly connected to the DS are called Access Points (AP). These APâs provide Distribution System Services (DSS) which enable the MAC to communicate with STAs not in the same BSS. Within the DS an entity know as a portal provides access to non-802.11 networks allowing communication with different LANâs e.g. Ethernet. These elements are shown in Figure 2

3 IEEE 802.11 protocol stack

The reference model of the IEEE 802.11 protocol stack is shown in Figure 3. In order to allow the integration of 802.11 networks into existing wired LANs the protocol stack is designed to be completely transparent to the upper layers. This implies that all wireless mobility issues must be addressed by the MAC layer. In order to allow a single MAC layer to support multiple physical layers, the physical layer is split into two sublayers. The physical layer convergence procedure (PLCP) is specific to the physical transmission method and is used to convert MSDU frames

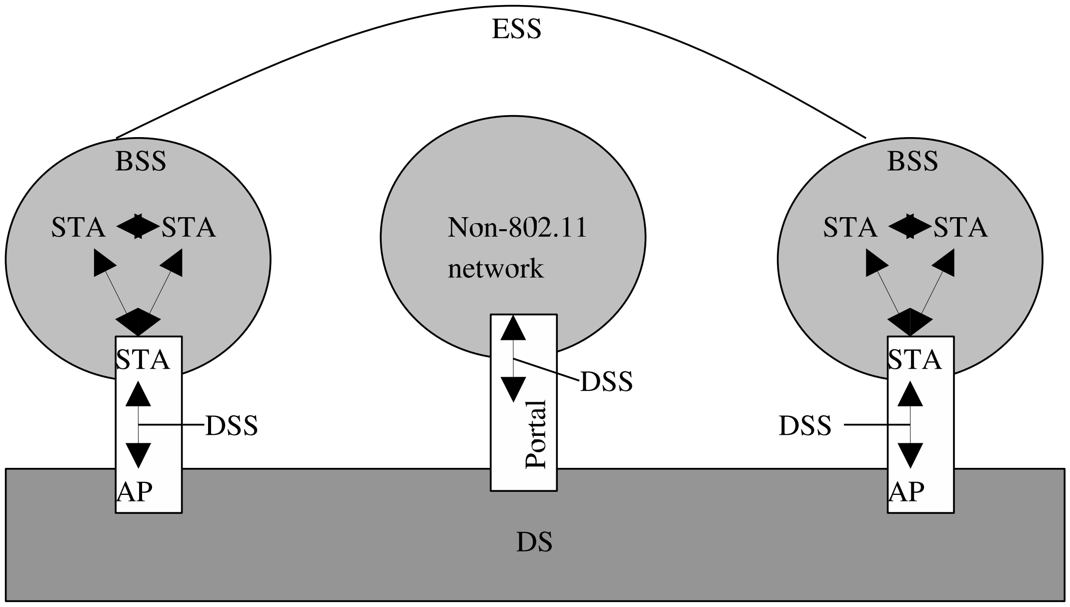


Figure 2: Components of 802.11 network

to the correct format for the selected transmission method. The physical medium dependent layer (PMD) is responsible for implementation of the required physical transmission method. This PHY layer, along with the MAC layer, will be the focus of this review and will be covered in more detail in the following sections.

The station management entity (SME) is a layer independent entity responsible for such functions as gathering layer-specific management information from the other layer management entities. The exact functions of the SME layer are not defined in the standard but the SME would typically perform management functions on behalf of the general system management entities and would implement standard management protocols.

3.1 The 802.11 MAC sublayer management entity (MLME)

The MLME provides the management service interfaces through which the required layer management functions may be invoked. The MLME provides all the required

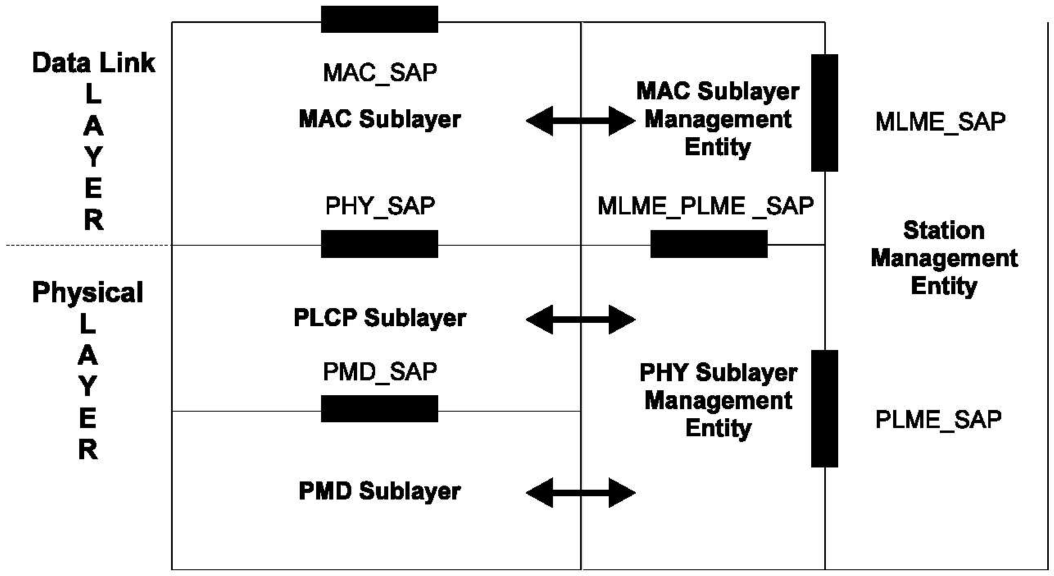


Figure 3: IEEE 802.11 reference model

management and mobility functions required for control of an 802.11 station or AP.

The MLME provides the required functions to allow all station within a BSS to synchronize to a single common clock. This is achieved through the use of a timing synchronization function (TSF) that must be contained locally in all stations. In infrastructure based networks the AP is responsible for synchronization of stations. This is achieved through the transmission of beacon frames that contain a copy of the APs TSF. On reception of the beacon, stations will update there local TSF to match that of the AP. In ad-hoc networks the generation of beacons is distributed amongst the stations in the network. In this case at the beginning of each beacon period all stations contained in an IBSS shall generate a beacon and a random time delay, the station will then wait for this random period of time before transmission of the beacon. Upon reception of a beacon from another station, any station with a beacon waiting for transmission will remove it from its buffer and update its TSF according to the information contained in the beacon.

The MLME is also responsible for the detection of 802.11 networks. This procedure involves either passive or active scanning to detect beacon frames. The information contained in these beacon frames gives all the required information, such as beacon period and BSS ID, to allow a station to synchronize and register with the network. In passive scan mode the station will monitor each channel for a period of time long enough to ensure that any beacons will be detected. In active scan mode the station is required to have knowledge, such as the BSS ID, of the BSS it wishes to scan for. After performing the DCF access procedure the station is required to transmit a probe frame. The network (station/AP) should then respond with a probe response frame that includes information required for correct operation of the physical layer as well as required MAC layer parameters.

In order to allow the station to perform power management in infrastructure networks the MLME is required to notify the AP of any changes in state using the frame control field of transmitted frames. This allows the station to enter a power-save (PS) mode when necessary. In this state the AP is required to buffer any MSDUs for the station and only transmit them at designated times. Stations are notified of pending MSDUs through the use of the Traffic Indication Map (TIM) which is contained in beacon frames. This means that stations in PS mode are required to wake up periodically to monitor beacon frames. Upon reception of a beacon frame that indicates an MSDU is buffered in the AP, the station shall respond with a short PS-poll frame. The MSDU is then either sent immediately or the PS-poll is acknowledged and the MSDU is sent later. In ad-hoc networks, stations make use of a special ad-hoc traffic indication message (ATIM) to notify stations in PS-mode that MSDUs are waiting to be transmitted. The ATIM is sent immediately after beacons to allow PS-mode stations to monitor the channel once every beacon period.

This section is intended to give an overview of the services provided by the MLME, more detailed information can be found in [3].

3.2 The 802.11 MAC sublayer functional description

The 802.11 MAC layer is required to support one mandatory and one optional coordinate function schemes. These functions are the DCF and the PCF. The architecture of the 802.11 MAC layer is shown in Figure 4 showing both the DCF and the PDF functions. The following sections give more detail on these coordinate functions.

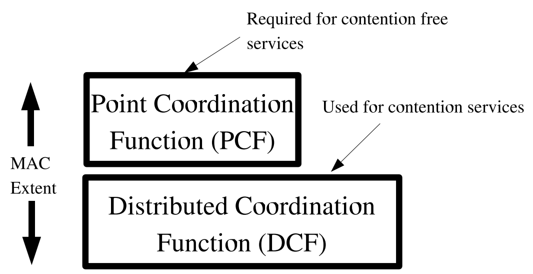

Required for contention free services
Point Coordination Function (PCF)
Used for contention services
MAC Extent
Distributed Coordination Function (DCF)

Figure 4: IEEE 802.11 MAC architecture

3.2.1 Distributed Coordinate function (DCF)

The DCF is the fundamental access method for the 802.11 MAC and provides asynchronous data transfer on a best effort basis. The DCF sits directly on top of the PHY layer and provides contention services, these services allow stations with packets to transmit to contend for access to the communications medium.

The DCF is based on the Carrier Sense Multiple Access with Collision **Avoidance** (CSMA/CA) [3, 5]. This scheme is a member of the popular ALOHA family of protocols and is very similar to the CSMA/CD scheme used in IEEE wired LANS e.g. Ethernet. CSMA/CA aims to avoid multiple stations attempting to access the medium simultaneously through its use of *virtual* and *physical* carrier sensing. This means that any station with packets queued for transmission must first *physically* sense, or monitor, the medium to ensure that no other station is transmitting. Although this scheme has been very successful in wired LANS, it presents several problems for wireless systems. The first of these is the "hidden node problem" [4], this is the situation where a receiver, such as an AP, can hear two separate transmitters, e.g. stations, but the transmitters cannot hear each other. In this case neither transmitter would know if the other is accessing the medium and hence collisions would occur. Another problem introduced by wireless LAN's is that it is not possible for a station to both transmit and monitor the channel simultaneously as is done in CSMA/CD systems. This means that in wireless systems the station is unable to detect collisions on the medium and hence will continue to transmit an entire packet even when a collision takes place. In order to solve these problems a virtual carrier sensing mechanism is included in this scheme. Virtual sensing makes use of a Request To Send (RTS) and Clear To Send (CTS) message to grant stations access to the medium. A Net Allocation Vector (NAV) is included in the CTS message, which can be heard by all stations in the BSS. The NAV indicates the amount of time required by the current transmission and hence the delay time before stations should begin to monitor the channel for idle status. In order to reduce the signaling overhead introduced by RTS/CTS this feature is not used for short packets, in this case the duration field is used to update the NAV.

3.2.2 Interframe space (IFS)

In order to introduce the concept of priority traffic, the 802.11 MAC makes use of the interframe space (IFS). This is the time interval between the transmission of frames and can be varied depending on the type of traffic being transmitted. The standard defines four IFS intervals: short IFS (SIFS), PCF IFS (PIFS), DCF IFS (DIFS) and the Extended IFS (EIFS), Figure 5 shows the relation ship between these values. The SIFS interval is used for the transmission of high priority traffic such as ACK frames, CTS frames or subsequent MPDUs of a fragmented burst. By making use of the shortest IFS the MAC ensures that other stations using longer IFSs do not sense the medium as being idle i.e. if a station is using the DIFS it must wait for the medium to initially become idle and then wait for DIFS to elapse before it may begin transmission. In this case a station sending a high priority message will begin transmission before the end of the DIFS period and hence the medium will not be detected as idle. A more detailed description of this process can be found in the standard [3]. The PIFS will only be used by stations operating under the PCF. As the PIFS is shorter than the DIFS, stations operating under PCF are of higher priority than those using DCF. The DIFS interval will be used by all stations operating under DCF. Stations operating in this mode will only be able to transmit if, after a successfully received frame the medium remains idle for a period equal to the DIFS interval plus a random âbackoff-timeâ calculated by the station. The EIFS is used by the DCF following the incorrect reception of a frame as indicated by the PHY layer. The random backoff procedure is used by the DCF to ensure that after the medium is determined to be idle, i.e. DIFS interval expires, that all stations waiting to transmit do not begin transmission simultaneously.

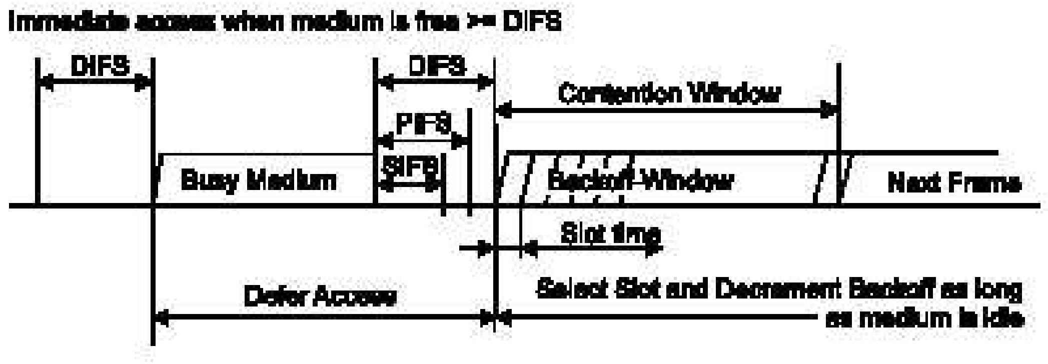

Immediate access when medium is free >= DIFS
DIFS
DIFS
Contention Window
PIFS
SIFS
Busy Medium
Backoff-Window
Next Frame
Slot time
Defer Access
Select Slot and Decrement Backoff as long as medium is idle

Figure 5: IFS relationships

3.2.3 DCF basic access procedure

The basic transmission procedure for stations using the DCF function is shown in Figure 6. In this example after Station A successfully completes transmission of a frame and an acknowledgement is received the medium is idle. The remaining stations detect this and monitor the medium for a further DIFS period to ensure the medium is idle. This is then followed by the backoff procedure, in this case the backoff time calculated by station C is shorter than that belonging to station B and as such station C is able to take control of the medium. The first stage in transmission of data is the transmission of a RTS message. After a SIFS period the receiver broadcasts a CTS message containing the necessary NAV parameters. The SIFS period is now used by the transmitter and receiver as this prevents other station from sensing that the medium is idle. Again, when station C has successfully transmitted its data and a DIFS period has passed station B will continue to decrement its backoff timer until time-out when it will send a RTS message. In this way, by using the IFS, the MAC layer is able to control access to the medium and avoid unnecessary collisions. It should be noted that it is possible for the MAC layer to perform fragmentation of upper layer data packets; in this case each fragment will be transmitted and acknowledged as before with the following segment being transmitted after a SIFS period (rather than the usual DIFS period). This ensures that a transmitter maintains control of the medium until all data segments are successfully received.

3.2.4 Point Coordination Function (PCF)

The PCF provides contention-free frame transfer for packets and is only available in infrastructure based networks i.e. those containing an AP. Although in the standard PCF is optional, all stations, even those with PCF disabled, will be able to operate under this scheme as it is based on DCF. The PCF relies on the point coordinator

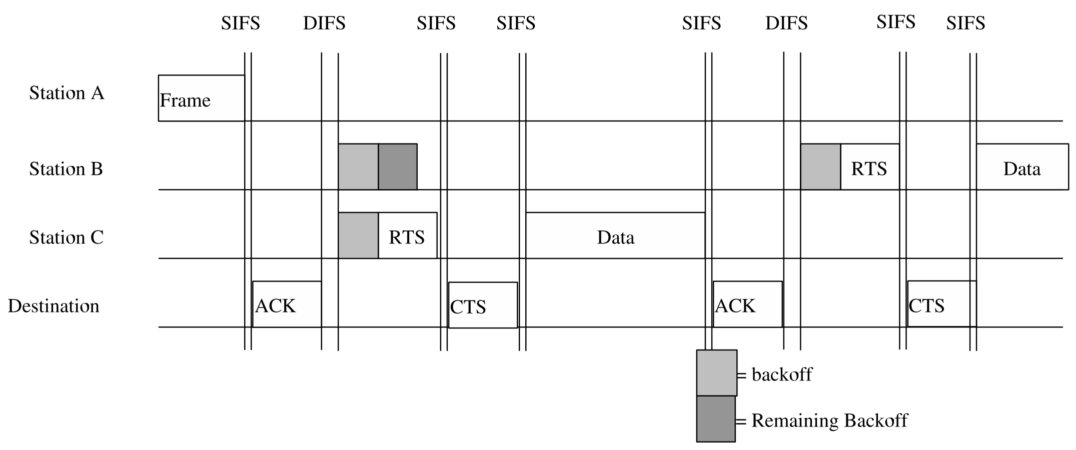


Figure 6: Basic transmission using DCF

(PC), usually the AP, to perform polling, enabling polled stations to obtain contention free access to the medium. The contention-free period (CFP) alternates with the contention period (CP) as shown in Figure 7. This also shows the beacon signal sent at the start of every CFP repetition interval. This beacon contains the maximum length of the CFP and is used to set the NAV of all stations in the BSS, this ensures that the PCF has control of the medium for this length of time. Within the CFP a polled station may only transmit a single MSDU. If the MSDU was not successfully acknowledged the station must either attempt retransmission in the CP or wait until re-polled by the PC. The PC takes control of the medium by only waiting for a PIFS interval after sensing the medium to be idle. As the PIFS is shorter than the DIFS the PCF will always take control of the medium before the DCF is able to.

3.2.5 Services provided by the MAC layer

In order to allow IEEE 802.11 to be used with a variety of DS implementations, the standard avoids specifying detail implementation issues and instead specifies services which the MAC must provide. Due to the requirement of 802.11 to support

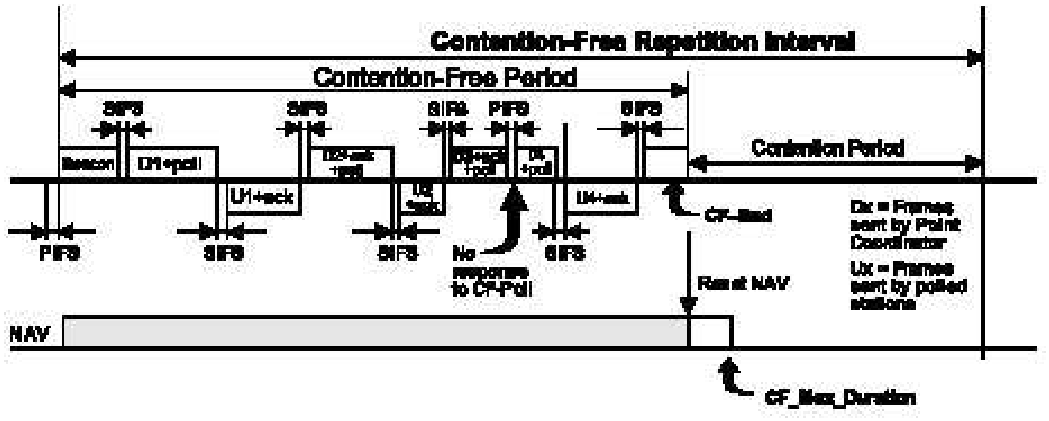


Figure 7: Example of PCF frame transfer

both infrastructure and ad-hoc based networks these services are grouped into two categories- the Station Services (SS) and the Distribution System Services (DSS).

3.2.5.1 Station Services In IBSS or ad-hoc network mode only SS are available, for this reason all stations must be able to perform these services. The following services are available in ad-hoc and infrastructure networks to allow access and confidentiality:

- Authentication: In order to improve the security of WLAN connections, authentication is used to allow stations to exchange their identity with stations they wish to communicate with. The standard supports the use of shared key authentication using the wired equivalent privacy (WEP) option.
- Deauthentication: This service cancels an existing authentication.
- Privacy: The contents of link-level messages can be protected by making use of the WEP option.
- MSDU delivery: Data is transmitted using the DCF or PCF depending on the network configuration.

3.2.5.2 Distribution System Services The DSS are used to distribute messages in the DS and also support mobility. These services are only available in infrastructure networks. It should be noted that in infrastructure networks both SS and DSS are available. DSS provides the following services:

- Integration: If the intended recipient of a message on the DS is located on a non-802.11 LAN the message will leave the DS through a portal rather than an AP. In this situation the DS is required to invoke an integration service which is responsible for performing any functions necessary to deliver the message form the DS media to the LAN media (including media or address translation).

- Distribution: This service allows the distribution of messages across the DS. As the DS is out with the scope of IEEE 802.11 the standard limits itself to providing the DS with enough information to determine the target AP of any messages. The necessary information is provided by the following association services.

- Association: In order to deliver a message within the DS the distribution service must know which AP to access for any station on the network. Before a station may send data via an AP it must first become associated with that AP. This process provides the DS with the required station-to-AP mapping. In order to provide a unique station-to-AP mapping a station may only be associated with one AP.

- Reassociation: This service allows stations to move between BSS in an ESS by allowing stations move associations between APs. This keeps the DS informed as to which AP is serving which stations.

- Disassociation: This service informs the DS to remove existing association information for a station. The DS will no longer forward packets from this station and any messages on the DS addressed to this station will be removed.

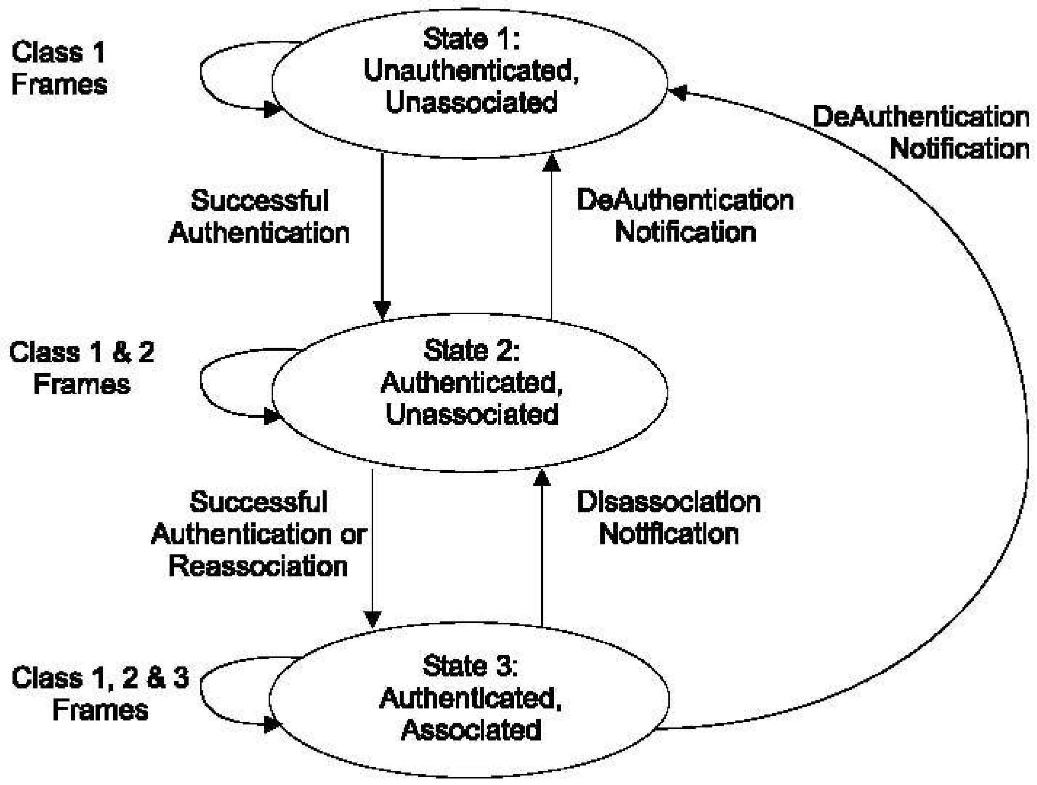

Class 1 Frames
State 1: Unauthenticated, Unassociated
DeAuthentication Notification
Successful Authentication
DeAuthentication Notification
Class 1 & 2 Frames
State 2: Authenticated, Unassociated
Successful Authentication or Reassociation
Disassociation Notification
Class 1, 2 & 3 Frames
State 3: Authenticated, Associated

Figure 8: Relationship between MAC states and services

The Authentication and association functions are used to control both access and the level of access available to terminals. This is controlled by the state machine shown in Figure 8 [3]. The current state of the station determines the frame types that may be used. The frame classes are defined as follows:

1. Class 1 frames (permitted from within states 1, 2 and 3)

   (a) Control Frames

      i. Request to send (RTS)

      ii. Clear to send (CTS)

      iii. Acknowledge (ACK)

   iv. Contention-free (CF)- End+ack
   v. CF-end

   (b) Management frames
      i. Probe request/response
      ii. Beacon
      iii. Authentication: If successfully Authenticated the station may make use of class 2 frames, otherwise the station remains in state 1.
      iv. Deauthentication
      v. Announcement traffic indication message (ATIM)

   (c) Data frames
      i. Data: non-DS traffic

2. Class 2 frames (permitted from within states 2 and 3)
   (a) Management frames
      i. Association request/response
      ii. Reassociation request/response
      iii. Disassociation

3. Class 3 frames (permitted only in state 3)
   (a) Data frames: allows traffic to be sent/received from DS.
   (b) Management frames
      i. Deauthentication
   (c) Control frames
      i. PS-Poll

4 IEEE 802.11 Physical Layer Specification

The 802.11 standard defines two basic physical layer access methods, frequency hopping spread spectrum (FHSS) and direct sequence spread spectrum (DSSS). FHSS physical layers have been almost totally replaced by DSSS techniques due to the increase through-put of these systems. For this reason FHSS will not be covered here but a detailed discussion can be found in the standard [3] as well as a number of texts [5, 2]. The following section gives details of the DSSS PHY.

4.1 Direct Sequence Spread Spectrum (DSSS)

As shown in Figure 3 the DSSS PHY layer contains three functional entities: the PMD layer, the PLCP and the PLME. The DSSS PLME is responsible for collecting physical layer management information. This information is stored in a management information base (MIB) that includes such things as transmit power levels and channel related information. The PMD and PLCP are covered in more detail in the following sections.

4.1.1 DSSS PLCP sublayer

The PLCP sublayer is responsible for the conversion of MAC protocol data units (MPDU) to PLCP protocol data units (PPDU). This process involves the addition of a PLCP header shown in Figure 9. The PLCP sync field consists of 128 bits of scrambled ones used to perform the necessary operations for synchronization. The PLCP signal bits are used to indicate to the PHY the modulation that should be used for transmission of the MSDU. The DSSS PHY supports both DBPSK and DQPSK modulation techniques. The PLCP length field is used to indicate the length of time, in microseconds, required to transmit the MSDU. The length field value is used to update the NAV value of stations within the BSS.

The DSSS PLCP is also responsible for the scrambling of PPDUs. Scrambling is performed to remove long strings of consecutive 1s and 0s in order to make the data appear more like background noise. This is achieved through the use of a polynomial scrambling function.

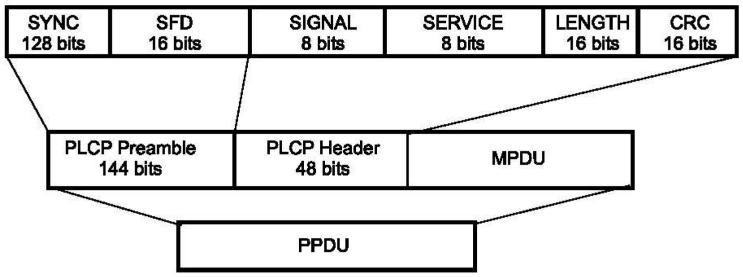


Figure 9: DSSS PLCP frame format

4.1.2 DSSS PMD sublayer

The DSSS PMD layer is responsible for all medium dependent issues including transmission of packets and sensing of the medium. The block diagram for the DSSS PMD transmitter/receiver is shown in Figure 10. The first stage in the transmission process is to spread the data. This process takes a narrow band signal and âspreadsâ it across a larger range of frequencies. This is achieved by application of a chip sequence to each bit of data. This means that the chip rate must be much higher than that of the underlying data, in the case of 802.11 DSSS an 11-bit barker sequence is used. This means that the chip rate must be 11 times faster than the data rate. It can be seen that this process not only spreads the signal across a wide range of frequencies but also introduces redundancy in the signal. The result of the spreading process is a new data sequence at a far higher rate [2].

After the sequence is filtered to ensure compliance with government regulations, i.e. operation within specific frequency bands, the sequence is modulated. 802.11 DSSS uses two forms of modulation- differential binary phase shift keying (DBPSK) and differential quadrature phase shift keying (DQPSK). DBPSK encodes data by producing phase changes in the transmitted signal. DBPSK encodes a single bit of data per phase shift and hence operates at the chip rate produced by the spreader. In 802.11 DSSS the PLCP header is always modulated using this scheme. DQPSK operates in a similar way to DBPSK in that it encodes data by making use of phase shifts in the transmitted signal. The main difference in these methods is that DQPSK encodes two bits of data per transmitted symbol and hence transmits data at twice the chip rate. The form of modulation used is selected by the MAC layer and is included in the PLCP header.

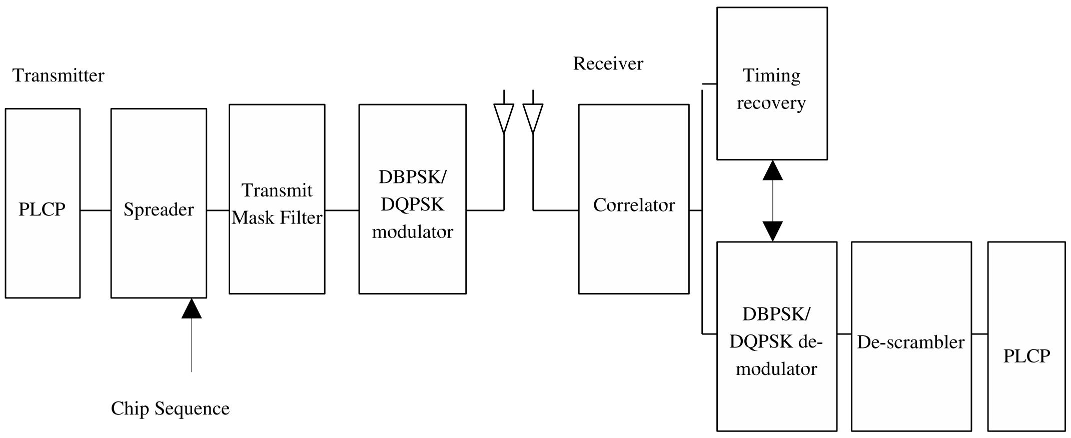


Figure 10: DSSS tranceiver

References

[1] 3GPP. www.3gpp.org.

[2] M. S. Gast. *802.11 Wireless Networks: The Definitive guide.* O'Reilly and Assoc, 2002.

[3] ISO/IEC. Information Technology- Telecommunications and information exchange between systems- Local and metropolitan area networks- Specific requirements- Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications. Technical report, IEEE Pres, 1999.

[4] L. Kleinrock and F. Tobagi. Packet Switching in Radio Channels: parts I & II. *Communications, IEEE Transactions on*, Dec 1975.

[5] A. Santamaria and F. Lopez-Hernandez. *Wireless LAN standards and applications*. Artech house, 2001.

Portfolio Document CP1

Implementation of Finite State Machines on a Reconfigurable Device

Author:
Graeme Milligan

Supervisor:
Wim Vanderbauwhede

Implementation of Finite State Machines on a Reconfigurable Device

Graeme Milligan, Wim Vanderbauwhede
Institute for System Level Integration, University of Glasgow
graeme.milligan@sli-institute.ac.uk, wim@dcs.gla.ac.uk

Abstract

We present a novel method for the implementation of Finite State Machines (FSM) using a reconfigurable architecture. The proposed method utilises run-time reconfiguration to reduce the hardware required to implement FSMs. This is achieved through the use of a unique representation of the FSM which allows the next state of the state machine to be calculated solely from the primary inputs rather than the primary inputs and the current state as would be traditionally required. This reduction in parameters significantly reduces the size of the hardware block required to calculate the next state.

The paper presents results obtained for the MCNC benchmark suite that demonstrate hardware savings of around 90% for the majority of the FSMs investigated.

1. Introduction

The use of Finite State machines (FSMs) to implement the control behaviour of systems is well established and as such a large number of texts are devoted to this subject including [3, 4, 5].

FSMs are commonly used to allow systems to react to a sequence of events rather than simply reacting to the current operating conditions of the system. This allows systems to operate automatically and reduces the need for human intervention.

The implementation of FSMs can be broadly separated into two sections, the derivation of the FSM from the high level specification of the system and the implementation of the FSM. The derivation of FSMs is independent of the method of implementation and is concerned with developing a high-level description of the control aspects of the specification.

The implementation of the FSM can be split into too main groups the FSM can either be implemented in either software or hardware. Software implementation of FSMs allows rapid development and deployment due to short software design cycles whereas, implementation in dedicated hardware produces devices with low area and power requirements but require lengthy and expensive hardware design and manufacturing processes. This is often impractical for low volume applications or for developers with limited resources. An alternative to the use of full custom hardware is the use of reconfigurable devices such as FPGAs. Reconfigurable hardware aims to combine the advantages of both software and hardware by providing the speed and efficiency of hardware and the flexibility and programmability of software. The introduction of this flexibility and programmability requires the introduction of hardware resulting in a lower efficiency than full-custom ASIC design.

Recently there has been much interest in making use of the ability of reconfigurable devices to be programmed while the device is in operation, this allows devices to implement multiple functions simultaneously by dynamically switching between functions or 'contexts'. This process is knows as run-time reconfiguration and allows a small reconfigurable device to 'masquerade' as a larger device by implementing applications that require larger hardware resources than available on a single reconfigurable device.

2. Formal Definition of Finite State Machines

Finite state machines (FSM) are abstract models used to represent sequential behaviours of systems. They are used in control applications to define the response of a system to a sequence of input events, allowing systems to act on sequences of input events. This allows designers to implement complex behaviour rather than simple systems that only react to the current operating conditions of the system.

An FSM can be fully described using the 6-tuple

$$(S, I, O, \Delta, \Lambda, R), where$$

- S is a finite set of states, and $|S|$ is the total number of unique states,

- I is a finite input space, and $|I|$ is the total number of inputs,
- O is a finite output space, and $|O|$ is the total number of outputs,
- Δ is a set state transitions based on the current state and the current inputs,
- Λ is the output relation defined in terms of the current state and the current input vector,
- and R is a set of reset states.

State transitions are assigned to each state and based on the current input and current state are used to determine the next state of the FSM. The current state of the FSM is determined by the initial state of the FSM and the previous input sequence applied to the machine.

The output of the state machine (O) can be generated in two ways; either the output is depended only on the state (Moore machine) or is dependent on the current state and the current inputs (Mealy machine). In the case of the Moore machine the output relation (Λ) will contain only a single output for each state.

A more detailed discussion of FSMs can be found in many texts including [3, 4, 5].

3. FSM implementation on a generic reconfigurable architecture

Reconfigurable devices such as FPGAs present a cost-effective alternative to full-custom ASIC design for low volumes and rapid prototyping. These devices are designed to be as general as possible to allow their use in a wide variety of applications, as such their performance, area and power requirements will be sub-optimal compared to full-custom ASICs.

The functionality of reconfigurable devices is determined by programming reconfigurable elements on the device to implement the required functionality. This is normally achieved by loading a bit-stream into configuration memory that is local to the device. The production of bit-streams is performed by synthesis tools provided by the manufacturer of the reconfigurable device and are targeted at a specific reconfigurable device or family of devices.

The conventional method of implementing an FSM on a reconfigurable device is shown in figure 1. As the figure shows, first a high-level description of the control requirements of the proposed device is extracted from the initial specification. The control behaviour of the specification can be expressed as a FSM using a high-level model such as a State Transition Graph (STG) or State Transition Table (STT). Based on such a description it is possible to then produce the necessary hardware to implement the required behaviour.

It should be noted that due to the mature nature of FSM implementation many of the steps described here would be hidden from the device designer and automatically carried out by design tools such as [1]. By making use of these tools the designers would usually create a high level description of the FSM in a Hardware Description Language (HDL) such as verilog or VHDL directly from the State Transition graph or State Transition table.

The process outlined here is similar to the design-flow used in full-custom ASIC design of FSMs but, where as in full-custom design flows synthesis results in a description that can be used to produce a silicon implementation of the FSM, synthesis for reconfigurable devices results in a bit-stream capable of configuring the device to the FSM.

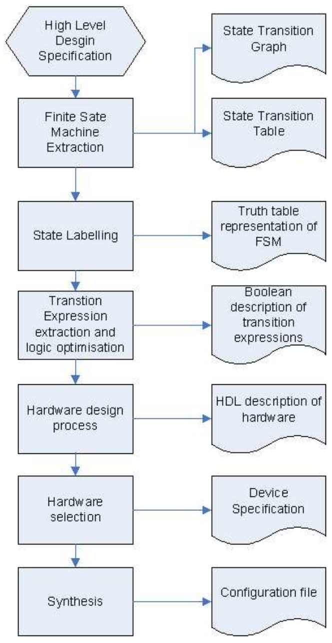

High Level Desgin Specification
State Transition Graph
Finite Sate Machine Extraction
State Transition Table
State Labelling
Truth table representation of FSM
Transtion Expression extraction and logic optimisation
Boolean description of transition expressions
Hardware design process
HDL description of hardware
Hardware selection
Device Specification
Synthesis
Configuration file

Figure 1. Simplified hardware design flow of a Finite state machine

The following sections give details of each of the main stages in the design flow for comparison to the novel implementation methodology presented in this paper. For the purposes of clarity the Mead-Conway traffic light controller [4] will be used to provide a detailed working example. reconfigurable device when implementing the FSM.

3.1. Finite State Machine extraction

In general, the control requirements of the system are extracted based on a high level specification of a system and expressed in a more concise high-level description. The following section briefly describes the STG and STT representation of FSMs.

3.1.1 State Transition graph representation of Finite State Machines

The state transition graph (STG) is a graphical representation of FSMs that uses nodes to represent the states of the FSM and edges to represent state transitions. The output behaviour of the FSM is associated with the states as required and the edges are labeled with the input conditions necessary to cause transitions.

A sample STG of the classic Mead-Conway traffic light controller [4] is shown in figure 2. This is a simple 4-state (HG, HY, FG, FY) FSM, where $|S| = 4$, with an input alphabet of C, T_1, T_s, $|I| = 3$. The STG shows each of the states and the state transitions, with each of the transitions labeled with the necessary input conditions to cause the transitions.

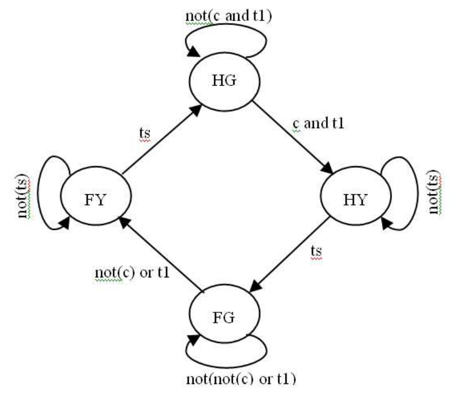


Figure 2. STG for Mead-Conway Traffic Light Controller [4]

3.1.2 State Transition Table representation of Finite State Machines

An alternative to the STG is the State Transition Table (STT) as shown in table 1. The table shows each of the required state transitions of the FSM in terms of the current state and current inputs of the FSM. In this simple example there are only two possible transitions from each state and the conditions that cause these transitions are listed. The outputs produced by the FSM are listed in the table, it should be noted that it is customary to produce a hardware block responsible for implementing the state machine and a separate block that produces the outputs based on the current state of the FSM.

Present state	Inputs	Next state	Outputs
HG	not(c and t1)	HG	hl=GREEN; fl=RED
HG	c and t1	HY	hl=GREEN; fl=RED
HY	not(ts)	HY	hl=YELLOW; fl=RED
HY	ts	FG	hl=YELLOW; fl=RED
FG	not(not(c) or t1)	FG	hl=RED; fl=GREEN
FG	not(c) or t1	FY	hl=RED; fl=GREEN
FY	not(ts)	FY	hl=RED; fl=YELLOW
FY	ts	HG	hl=RED; fl=YELLOW

Table 1. STT of Mead-Conway Traffic Light Controller

3.2. State Labeling

To allow digital logic to implement the FSM it is necessary to perform state encoding. State encoding takes the symbolic representation, i.e. label, of the states and replaces it with a Boolean representation that can be produced using digital logic. In the case of binary encoding, each state is given a unique binary code and this is associated with the state label as shown in table 2.

	Encoding	
State Label	Z_0	Z_1
HG	0	0
HY	0	1
FY	1	0
FG	1	1

Table 2. Binary Encoding of State Labels for Mead-Conway traffic light controller

By replacing the labels in the STT with their binary equivalents, as given in table 2, it is possible to build a truth table representation of the FSM. Table 3 gives the truth table for the Mead-Conway traffic light controller. From this is can be seen that the STT presented in section 3.1.2 has been expanded to give all of the possible input conditions for each current state $(Z_0(t), Z_1(t))$ and the next state $(Z_0(t+1), Z_1(t+1))$ for each of these conditions has been specified.

Present State		Primary Input			Next State	
$Z_0(t)$	$Z_1(t)$	C	t1	ts	$Z_0(t+1)$	$Z_1(t+1)$
0	0	0	0	0	0	0
0	0	0	0	1	0	0
0	0	0	1	0	0	0
0	0	0	1	1	0	0
0	0	1	0	0	0	0
0	0	1	0	1	0	0
0	0	1	1	0	0	1
0	0	1	1	1	0	1
0	1	0	0	0	0	1
0	1	0	0	1	1	1
0	1	0	1	0	0	1
0	1	0	1	1	1	1
0	1	1	0	0	0	1
0	1	1	0	1	1	1
0	1	1	1	0	0	1
0	1	1	1	1	1	1
1	0	0	0	0	1	0
1	0	0	0	1	0	0
1	0	0	1	0	1	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	0	1	0	0
1	0	1	1	0	1	0
1	0	1	1	1	0	0
1	1	0	0	0	1	0
1	1	0	0	1	1	0
1	1	0	1	0	1	0
1	1	0	1	1	1	0
1	1	1	0	0	1	1
1	1	1	0	1	1	1
1	1	1	1	0	1	0
1	1	1	1	1	1	0

Table 3. Truth-Table representation of the Mead-Conway traffic light controller

Using a vector notation

$$\mathbf{Z} = (Z_0, Z_1),\ \mathbf{HG} = (0,0), etc.$$

and using the shorthand $(\mathbf{X} = \mathbf{Y})$for $\sum_i X_i \oplus Y_i$, the generator expression for table can be written as:

$$\begin{aligned}\mathbf{Z}(t+1)\ &\triangleq \\ &(\mathbf{Z}(t) = \mathbf{HG}).(\overline{c.t_1}.\mathbf{HG} + c.t_1.\mathbf{HY}) + \\ &(\mathbf{Z}(t) = \mathbf{HY}).(\overline{t_s}.\mathbf{HY} + t_s.\mathbf{FG}) + \\ &(\mathbf{Z}(t) = \mathbf{FG}).(\overline{\overline{c} + t_1}.\mathbf{FG} + (\overline{c} + t_1).\mathbf{FY}) + \\ &(\mathbf{Z}(t) = \mathbf{FY}).(\overline{t_s}.\mathbf{FY} + t_s.\mathbf{HG})\end{aligned}$$

3.3. Transition Expression (TE) Extraction and Optimization

From the truth-table representation it is possible to extract the logic expressions required to implement the behaviour of the FSM. The extracted expressions are termed Transition Expressions (TE) as they calculate the required state transitions based on the primary inputs and the current state.

If the transition expression is expressed in sum-of-product format, each 1 in the corresponding column will result in a single product term to be added to the expression. This means that for the example shown in table 3 the logic expression required to calculate the next state $(Z_0(t+1))$ is

$$\begin{aligned}\mathbf{Z_0}(t+1)\ &= \mathbf{Z_0}(t)!\mathbf{Z_1}(t)!\mathbf{Ct1!ts} + \mathbf{Z_0}(t)\mathbf{Z_1}(t)!\mathbf{Ct1!ts} \\ &+\mathbf{Z_0}(t)!\mathbf{Z_1}(t)\mathbf{C!t1!ts} + \mathbf{Z_0}(t)\mathbf{Z_1}(t)\mathbf{C!t1!ts} \\ &+\mathbf{Z_0}(t)!\mathbf{Z_1}(t)!\mathbf{Ct1!ts} + \mathbf{Z_0}(t)\mathbf{Z_1}(t)!\mathbf{Ct1!ts} \\ &+\mathbf{Z_0}(t)!\mathbf{Z_1}(t)\mathbf{Ct1!ts} + \mathbf{Z_0}(t)\mathbf{Z_1}(t)\mathbf{Ct1!ts} \\ &+!\mathbf{Z_0}(t)\mathbf{Z_1}(t)!\mathbf{C!t1ts} + \mathbf{Z_0}(t)\mathbf{Z_1}(t)!\mathbf{C!t1ts} \\ &+!\mathbf{Z_0}(t)\mathbf{Z_1}(t)\mathbf{C!t1ts} + \mathbf{Z_0}(t)\mathbf{Z_1}(t)\mathbf{C!t1ts} \\ &+!\mathbf{Z_0}(t)\mathbf{Z_1}(t)!\mathbf{Ct1ts} + \mathbf{Z_0}(t)\mathbf{Z_1}(t)!\mathbf{Ct1ts} \\ &+!\mathbf{Z_0}(t)\mathbf{Z_1}(t)\mathbf{Ct1ts} + \mathbf{Z_0}(t)\mathbf{Z_1}(t)\mathbf{Ct1ts};\end{aligned}$$

This expression can be optimized using boolean simplification techniques to remove redundant logic. This simplification process is well established and can be performed using tools such as SIS [2]. Performing boolean simplification on the above transition expression results in

$$\mathbf{Z_0}(t+1)\ \ = \mathbf{Z_0}(t)!\mathbf{ts} + \mathbf{Z_1}(t)\mathbf{ts}$$

As this shows, the removal of redundant product terms, using boolean minimization techniques vastly reduces the size of the logic block required to implement the transition expressions.

3.4. Hardware selection

The selection of a suitable reconfigurable device is vital to the efficient implementation of the FSM. It is essential to ensure that the device has sufficient hardware resources to implement the FSM without introducing excessive redundant hardware. As the reconfigurable device is a generic part it is unlikely that the device will have exactly the correct amount of hardware required to implement a particular application. However, devices are available from device manufacturers in a wide range of size and cost, allowing the end-user to select a device suitable for the chosen application.

Based on the HDL description of the FSM it is possible to extract an estimate of the hardware requirements of the device required to implement the FSM. The end-user would then select a device with as close to these parameters as possible in order to ensure excessive hardware is not introduced as this would impact on the cost, area and possibly power of the final implementation.

3.5. HDL Synthesis for reconfigurable devices

After the selection of the appropriate reconfigurable device the HDL description of the FSM can be synthesised. The synthesis process takes the HDL description and produces the required bit-stream to program the device. As each reconfigurable device has an individual structure and hardware characteristics, synthesis is usually performed by proprietary tools provided by the device manufacture. The results of the synthesis process is a file containing a series of bits that when stored in local memory on the device causing it to implement the required functionality.

required to implement the FSM can be obtained before synthesis it is possible that factors such as routing congestion may result in the chosen device being unsuitable for the chosen application. This would result in the end-user having to re-synthesize the HDL targeting a device with sufficient routing or hardware to implement the required functionality.

3.6. FSM operation on a reconfigurable device

The general model of a FSM is shown in figure 3. As this shows the FSM is implemented by a Combinatorial Logic Block (CLB) responsible for the calculation of the next state and a feedback register used to store the next state as the current state.

On power up the configuration bit-file is loaded into the local memory within the reconfigurable device from non-volatile memory. This configures the reconfigurable device to implement the required transition expression and the current state register. The device may also implement the logic required to produce the outputs associated with the current state of the FSM or simply output the current state directly.

After the required configuration is loaded onto the device the next state would be calculated using the current state, initially this would be a specified *reset* state, and the current inputs. The system would then be clocked and the next state stored as the current state in

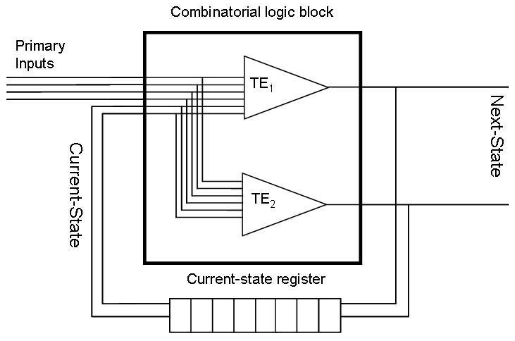


Figure 3. General Hardware implementation of FSMs

the current state register, this new current state would then be used to again calculate the next state before the device is again clocked. This means that the clock speed of the device is limited by the time required for the device to calculate the next state and store it as the current state.

4. Novel representation of FSMs for reconfigurable hardware

The main fixed hardware component of the general implementation is the combinatorial logic block (CLB). This block was targeted for optimisation to produce an optimised reconfigurable hardware device for the implementation of FSMs.

4.1. Investigation of characteristics of CLB

As the CLB implements the transition expressions (TEs) required to calculate the next state of the FSM from the primary inputs and the current state it is possible to calculate a number of the key characteristics of the block required to perform this function. The first of these is the number of outputs, if binary encoding is assumed, this is simply

$$O = \log_2(|S|)$$

It is also fairly trivial to show that, as feedback is required from the current state register, the total number of inputs to the combinatorial logic block is found by

$$I = I_p + O = I_p + \log_2(|S|)$$

As these equations show the inputs to the CLB can be divided into two categories. The first of these is the primary inputs; these are the input signals used to determine the next state transition and are necessary to the operation of the FSM. The remaining inputs

Primary Input			Next State	
C	t1	ts	$Z_0(t+1)$	$Z_1(t+1)$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	0
1	0	1	0	0
1	1	0	0	1
1	1	1	0	1

Table 4. Sub-truth-tables for state HG

are the feedback lines from the current state register. Consequently, if this feedback can be removed the total number of inputs to the array can be reduced by a factor of $\log_2(|S|)$.

4.2. Removal of current state feedback

As a reconfigurable hardware device has the capability of altering its functionality through programming it is possible to remove the need for the current state to be fed back to the inputs. This can be achieved by using the current configuration of the reconfigurable device to store the current state of the FSM, i.e. creating a separate context for each state of the FSM.

On start-up the reconfigurable device is loaded with the context of the initial state of the FSM; the next state of the FSM can be calculated solely from the current inputs. If the next state is found to be different from the current state the device is configured to implement the context of the next state.

As this process calculates which state the FSM should move to, the contexts required to be loaded onto the reconfigurable device are know as **Forward Transition Expressions** (FTEs). The following section describes the derivation of these FTEs for the Mead-Conway example shown in figure 2.

4.2.1 Derivation of FTEs for Mead-Conway traffic light FSM

If the example of the Mead-Conway traffic light controller is again considered it can be seen that the truth-table shown in table 3 can be separated in to sections that define the behaviour of the FSM when the FSM is know to be in a particular state. This is achieved by breaking the truth table down into sections that contain all possible input vectors for each state. In this example the truth-table is broken down into a sub-truth-table for each state, in this case this results in four truth-tables of three inputs. Table 4 shows the truth-table for the HG state.

If the FSM is assumed to be in the HG state the following expressions can be used to calculate the next state:

$$\begin{aligned} \mathbf{Z}(t) = \mathbf{HG} \quad &\Rightarrow \\ \mathbf{Z}(t+1) \quad &\triangleq \quad \overline{c.t_1}.\mathbf{HG} + c.t_1.\mathbf{HY}) \end{aligned}$$

Substituting the actual state labels yields:

$$Z_0(t+1) = 0;\ Z_1(t+1) = c.t1$$

Using these expressions it is then possible to calculate the next state using only the primary inputs. The expressions are implemented on the reconfigurable device and indicate the state the FSM will enter at the next state transition. If a state change is detected it would then be necessary to load the FTEs for the next state onto the reconfigurable device and again recalculate the next state at the next state transition.

4.3. Reconfigurable device for the implementation of FTEs

The overall architecture of the device is shown in figure 4. It can be seen that although a current state register is still required to test if a state transition has occurred it is no longer necessary to include the feedback lines to the arrays.

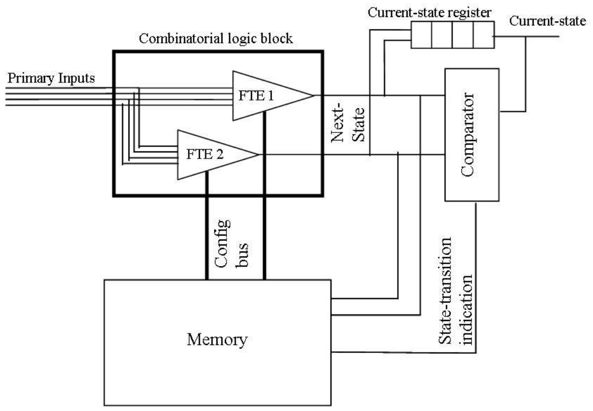


Figure 4. Custom reconfigurable device for implementation of FSMs

4.3.1 Device operation

Using the design illustrated in figure 4 the array obtains its initial configuration from the data stored in the associated memory. The next state is then calculated using the FTEs based solely on the primary inputs. The output of the reconfigurable device (next state) is then compared to the value stored in the current state register (current state). If the next state does not equal the current state a state change is indicated and the device obtains the FTEs for the state indicated by the reconfigurable device. The configuration required to implement the FTEs for the next state is then automatically obtained from the configuration memory and loaded to the reconfigurable device. The device can then again calculate the next state using the new FTEs. The algorithm for the operation of this device is shown in algorithm 1.

Algorithm 1 FSM operation

```
while 1 do
  $device\_cfg = reset\_cfg$
  $current\_state = reset\_state$
  while $reset = no$ do
    $next\_State =$
    $transition\_expr_{current\_state}(primary\_inputs)$
    if $next\_state != current\_state$ then
      $device\_cfg = next\_state\_cfg$
      $current\_state = next\_state$
```

5. FSM implementation on a custom reconfigurable architecture

The design flow for the implementation of FSM using the custom method and architecture suggested is shown in figure 5. As this shows the derivation of the FSM from the high level specification of the system is the same as that for the full custom and conventional reconfigurable implementation.

However, after the truth-table representation is produced using the method outlined in section 3.1 the sub-truth tables are extracted as demonstrated in section 4.2. This process results in S sub-truth-tables where the information in each sub-table is sufficient to implement the behaviour required for a single state of the FSM.

Based on the sub-truth-table representation of the FSM it is possible to perform FTE extraction. This process is similar to the conventional Transition Expression extraction process but results in simplified boolean expressions based on only the primary inputs. FTE extraction will produce $log_2(|S|)$ boolean expressions for each state.

6. Results

In order to investigate the effectiveness of the method presented here it is necessary to compare the

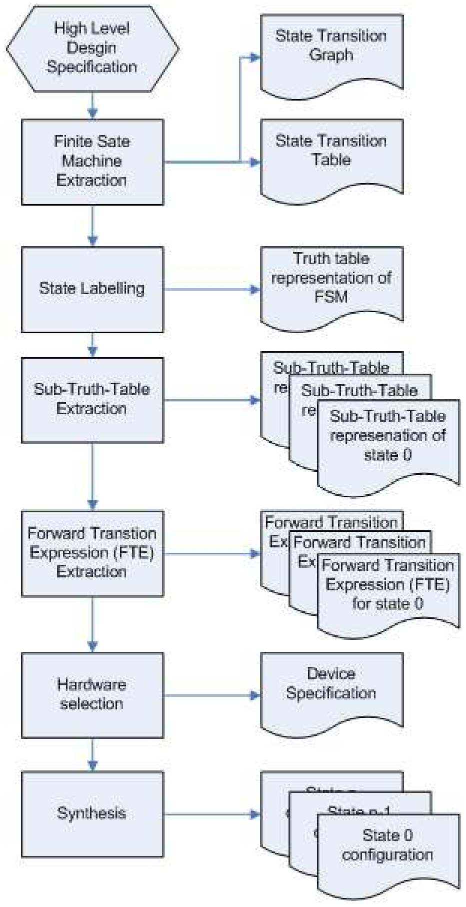

High Level Desgin Specification
State Transition Graph
Finite Sate Machine Extraction
State Transition Table
State Labelling
Truth table representation of FSM
Sub-Truth-Table Extraction
Sub-Truth-Table representation of state 0
Forward Transtion Expression (FTE) Extraction
Forward Transition Expression (FTE) for state 0
Hardware selection
Device Specification
Synthesis
State n-1
State 0 configuration

Figure 5. Hardware design flow of a FSM for custom reconfigurable device

results of implementation to those obtained using the more traditional method. It is expected that the method presented here will allow FSMs to be implemented using less hardware than those produced using the tradional design flow.

In order to compare the implementation methods the number of LUTs used to implement the CLB of a set of FSMs was collected for both the traditional and FTE based implementations. The FSMs used are to perform this comparison are selected from the MCNC benchmark suite [6] and are selected to represent as broad a range of FSMs as possible, for this reason the MCNC benchmarks were profiled in terms of number of inputs and states and the corner cases selected. The selected benchmark circuits are synthesised using the SIS synthesis tool and the number of LUTs required to implement the CLB are calculated. The FSM is then broken down into sub-truth-tables as described in section 4.2.1 and the number of LUTs required to implement **each** state are calculated using SIS. As a reconfigurable device must be able to implement all states,

			TE-based		FTE-based	
Benchmark Circuit	$\|S\|$	$\|I\|$	LUTs	MEM (b)	LUTs	MEM (b)
DK15	4	3	41	656	2	138
DK17	8	2	35	560	3	384
Planet	48	7	521	8192	3	2304
Kirkman	12	16	142	2272	5	960
Ex1	20	9	181	2896	10	3200
Opus	10	5	59	944	18	2880

Table 5. Number of LUTs in CLB

the minimum size of the reconfigurable device required is equal to the maximum number of LUTs required the implement the FTEs of any single state.

It is also possible to calculate the memory required to store the bit stream used to program the device based on the number of LUTs (L) and the size of the LUT (K); for the traditional system the number of program bits (B) is

$$B = L * 2^K$$

In the case of the FTE based implementation the number of program bits is calculated by

$$B = (L * 2^K)|S|$$

The results obtained for the selected MCNC benchmark circuits is shown in table 5

The table shows that for each of the benchmark circuits the number of LUTs required is substantially reduced by making use of FTEs. This is due to the fact that only a small part of the CLB for the FSM is required to be implemented at any time. It can also be see that in the majority of the cases presented here the amount of memory required to store the configuration data is also reduced. In the case of Ex1 and Opus the use of FTEs actually increases the amount of configuration data required. In these examples the FSM have relatively large numbers of states compared to inputs and in particular have a few states with very large numbers of FTEs. For the FTE implementation this results in a single state, or relatively few sates, requiring large numbers of LUTs compared to the remaining states of the FSM. As the size of the device required is set by the maximum number of LUTs required by **any** single state this results in a large CLB where the majority of the states utilises only a small fraction of the available hardware but as all of the LUTs require programming the size of the configuration data required is the same for each state.

7. Conclusion

This paper has presented a novel representation of FSMs specifically tailored to take advantage of the properties of reconfigurable hardware.

The ability of reconfigurable hardware to be reconfigured during run-time allows these devices to calculate the next state of FSMs using only the primary inputs. This is made possible through the use of FTEs to produce a unique context, or configuration, for each state. By making use of this novel representation the feedback register traditionally used in FSM implementation is no longer required reducing the inputs to the CLB by a factor of $\log_2(|S|)$.

As each context is only required to implement the transitions of a single state this reduces the size of the reconfigurable block required to implement the FSM.

In order to investigate the effect of the novel method presented here selected MCNC benchmark circuits were implemented and the hardware characteristics recorded. The MCNC benchmark suite was profiled and the circuits selected to represent the corner cases in terms of number of inputs and states.

For each of the examples selected the use of FTEs vastly reduces the number of LUTs required to implement the FSM. The results also show that FSMs with state transitions spread evenly across the states, rather than single states with large numbers of traditions, are most suitable for the use of FTEs as this results in FSMs with far lower LUT counts as well as reduced configuration memory requirements.

8. Acknowledgments

This work is supported by EPSRC and SLI Ltd, and was produced at the ISLI in partnership with the Universities of Glasgow, Strathclyde, Edinburgh and Heriott-Watt.

References

[1] Cadence. Cadence Encounter Digital IC Design Platform. http://www.cadence.com/.

[2] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, 1992.

[3] A. Gill. *Introduction to the Theory of Finite-state Machines.* McGraw-Hill, 1962.

[4] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. *Synthesis of Finite State Machines: Functional Optimization.* Kluwer Academic Publishers, 1997.

[5] T. Villa, T. Kam, R. Braytonand, and A. Sangiovanni-Vincentelli. *Synthesis of Finite State Machines: Logic Optimization.* Kluwer Academic Publishers, 1997.

[6] S. Yang. Logic synthesis and optimization benchmarks user guide version. Technical report, 1991.

PORTFOLIO DOCUMENT CP2

Random Circuit Generation for the Testing of Programmable Devices

Author:
Graeme Milligan

Supervisor:
Wim Vanderbauwhede

Random Circuit Generation for the Testing of Programmable Devices

Graeme Milligan
Institute for System Level Integration,
Alba Centre, Alba Campus, Livingston
Scotland, UK, EH54 7EG.
Email: graeme.milligan@sli-institute.ac.uk

Wim Vanderbauwhede
Department of Computing Science,
University of Glasgow, Glasgow,
Scotland, UK, G12 8QQ.
Email: wim@dcs.gla.ac.uk

***Abstract*—Although the use of benchmark suites, such as the MCNC benchmarks, provides reconfigurable device designers with a suite of circuits for device testing, their usefulness is limited due to the relatively small number of these circuits. We present here a method of generating large numbers of *realistic* circuits, that can be used for device testing, using a high-level Monte-Carlo based circuit generator. The suggested circuit generation method is used to calculate the optimal characteristics of Programmable Logic Arrays with respect to the range of circuits the device can implement. The results of this process are then compared to existing data to prove the validity of the circuit generation method.**

I. Introduction

The use of reconfigurable hardware devices is becoming more common as they introduce flexibility to the hardware design process while retaining some of the area, speed and power benefits of full custom ASIC design. This flexibility is introduced through the use of programmable hardware devices such as Programmable Logic Arrays (PLAs).

The flexibility introduced by PLAs comes at the cost of lower efficiency when compared to full custom design due to the overhead introduced in providing programmability and the generality required to allow their use in a variety of applications. To limit the effect of this overhead it is essential to ensure the correct amount, and type, of hardware is placed on reconfigurable devices. This ensures that, while the devices provide sufficient flexibility to allow their use in a wide variety of applications, excessive amounts of redundant hardware is not included.

As the end application of programmable devices is not known at design time it is difficult for device designers to anticipate the amount of hardware that will be required by the end user. This paper presents a testing method for PLAs based on the Monte-Carlo generation of logic expressions. This random generation method allows PLA designers to assess the effect design changes have on the devices ability to implement a range of logic circuits and is related to the idea of a measure of flexibility presented in [9]. This process gives a measure by which device designers and end users can assess the range of circuits the reconfigurable device is capable of implementing.

The following section gives an overview of PLA operation and gives necessary definitions. This is then followed by a discussion of previous strategies used to test and assess the flexibility of reconfigurable devices. The method suggested by this paper is then presented and the results compared to those obtained using existing strategies to demonstrate the validity of this method.

II. Background theory

A. Programmable Logic Arrays

The PLA, and its variants, are hardware devices that provide the opportunity to determine their functionality post-manufacturing. Early devices achieved this through the use of fuses or anti-fuses and could be programmed only once. Modern devices make use of reprogrammable memory blocks to allow the devices to be rapidly programmed on multiple occasions.

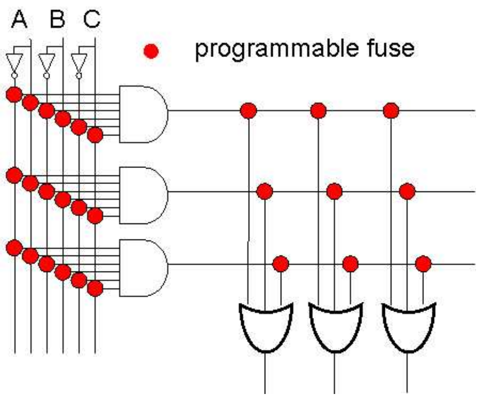


Fig. 1. PLA architecture

PLAs are designed to implement combinatorial logic circuits and their design is derived directly from the sum-of-products (SOP) representation of these circuits.

The structure of a typical PLA is shown in figure 1. This device contains a plane of *AND* gates to implement the product terms of the expression. The outputs of these *AND* gates are then connected to *OR* gates to produce the final expression. The use of programmable switches allows the user to determine which of the *AND* gates will be connected to the *OR* gate and hence determines the final functionality of the PLA.

The typical PLA can be fully described by the tuple

$$\{i, o, p\}$$

where,

i=number of inputs,
o=number of outputs and,
p=number of product terms

Based on these parameters it is possible to determine the hardware characteristics of the device as;

- The number of inputs (i) sets the number of inputs required by the *AND* gates in the *AND* plane, as shown in figure 1 the *AND* gate has two inputs for each PLA input as it is normal to provide an inverted version of each of the inputs.
- The number of outputs (o) is equal to the number of *OR* gates in the *OR* plane.
- The number product terms (p) determines the number of programmable switches in the array but as a results also determines the number of *AND* gates in the *AND* plane and also the number of inputs required by the *OR* gates in the *OR* plane. It can also be seen that the number of product terms determines the complexity of the SOP expression that can be implemented and as such has a major impact on the type and complexity of circuits the PLA can implement.

In order to ensure that the PLA can implement as broad a range of SOP expressions as possible it is essential to ensure that sufficient product terms are available on the device. As it is unlikely that the final applications of the PLA will be known at design time this is major challenge for the device designer.

B. Existing PLA testing strategies

As the future applications of a PLAs can not be predicted at design time, it is difficult to determine the the amount of hardware that will be required by the end user of the device. When creating new PLA designs it is important to ensure that the devices can implement as broad a range of circuits as possible as this increases the application domain of the device.

Traditionally device testing implies the comparison of new device designs to the initial specification to ensure the device operates as expected. In the case of reconfigurable devices it is necessary to also test the initial specification to ensure the device has sufficient flexibility to implement future end-user applications. In order to achieve this, a number of strategies have been used to test the flexibility of new reconfigurable devices. These techniques are shown in figure 2 and range from the use of *real-life* circuits to completely random, *synthetic*, circuits. The following section outlines the previous strategies employed in testing new reconfigurable devices.

1) Benchmark testing: Traditionally, the method used to select the characteristics of PLAs was the use of benchmark circuits [4], [15]. This method relies on a suite of sample

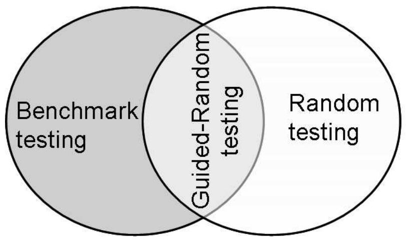


Fig. 2. Existing test strategies

real world circuits that are mapped to PLAs using place and route algorithms such as PLAmap [4]. In [4] a large number of the MCNC benchmark circuits were mapped to PLAs with varying parameters to determine the parameters that resulted in the best delay and area characteristics. This method was was also used in [15] to investigate the effect of the varying the number of product terms available on the device on the delay and area characteristics of the PLA when implementing a range of benchmark circuits.

Although these methods are useful in determining the optimal characteristics of PLAs, the results are highly dependent on the use and availability of benchmark circuits. If only a small number of benchmark circuits are available, or if they are closely related, this method may produce results that are particular to the benchmarks and not general applications. As the circuits in the benchmark suites are existing *real world* circuits it is also unclear how well they will represent future circuits that the device may be required to implement after manufacture.

2) Guided-Random testing: In [8], [9] methods of extending the use of benchmarks were suggested. These methods first profile a set of benchmarks to determine ranges for a number of characteristics such as number of inputs, outputs and circuit complexity. Based on this a large number of *synthetic* circuits can be randomly generated with characteristics within these ranges. This allows a very large number of circuits to be generated that can be used for testing.

In [9] these circuits were mapped to sample devices and a measure of flexibility extracted based on the percentage of the total circuits that could be mapped using the place and route tools allowing device designers to rapidly assess the effect design changes have on the flexibility of the device. This method is particularly aimed at domain specific reconfigurable devices where the final application domain of the device is well known and hence the reliability of the results can be assured as it is likely, although not guaranteed, that circuits within a domain will be fairly similar.

These methods do not completely alleviate the issues relating to the availability of sufficient benchmark circuits as, if the domain is small it is unlikely that enough example circuits will be available to completely profile the domain or conversely, if the domain is very large it is likely that the characteristics obtained by profiling will have a large range and the random generation of *synthetic* circuits will result in these circuits being almost totally unconstrained, resulting in almost completely random circuits.

3) Random testing: The use of completely random circuits for device testing was suggested in [5]. This method produces randomly generated netlists used to assess the routability of reconfigurable devices by generating large numbers of circuits and performing place and route. This method was rejected in [8] as it was claimed that the results of the unconstrained random generation did not produce *realistic* circuits. It is felt that the random generation process produced unrealistic circuits as high level optimization was not carried out on the circuits produced, as would be the case for the real circuits where it is likely that processes such as boolean minimization would have taken place before synthesis to netlist.

4) Investigation of average products in CLEs: A number of research projects [6], [11], [14], [12], [13], [1] have attempted to calculate the average number of product terms in CLEs and the upper and lower bounds for minimised SOP expressions. In [14], [12] randomly generated logic expressions are used to calculate the average number of product terms in the expressions and the upper and lower bounds on this are calculated mathematically in a similar way to [1]. In both of these examples the value for the average number of product terms is calculated using randomly generated logic circuits. Although this method seems to generate values in the expected range for the average case behaviour it is impossible to verify the accuracy of the random circuit generation method as no details are given of the precise manner of random circuit generation. These methods also focus solely on determining the characteristics of PLAs and if this work can be extended to other reconfigurable devices such as FPGAs..

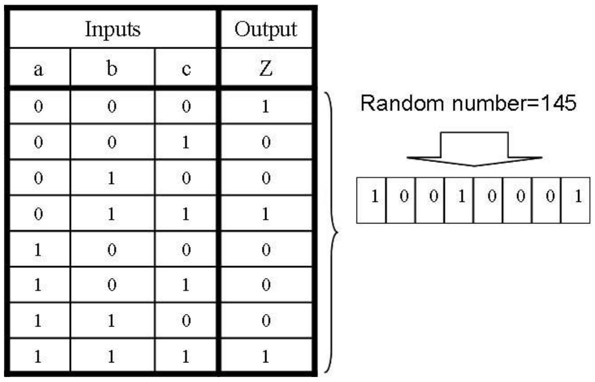
III. Monte-Carlo based PLA Test Strategy

Generating random logic expressions at a high level and performing high level optimization, such as boolean minimization, results in *realistic* logic circuits that can be used for device testing. If large numbers of these circuits are generated it is thus possible to use this approach to assess the flexibility of the PLA, similarly to the method suggested in [9], by calculating the percentage of circuits that the device can implement. In this way device designers can determine the effect their choice of parameters has on the range of circuits the device is capable of implementing.

It is also possible to turn this concept around by generating large numbers of circuits and calculating the characteristics of the PLA required to implement them. This method allows device designers to select the flexibility required by the device and obtain the characteristics required to achieve this value. The following section details the high-level Monte-Carlo based approach to generating test circuits for PLAs and how this can be used to determine a measure of flexibility of the device. The results of this testing are then compared to the results obtained using previous methods to demonstrate the validity of this process.

A. Monte-Carlo generation of CLEs

Although in [8] is was suggested that the random generation of circuits produced results that did not match *real* circuits, this is because high-level simplification, such as boolean minimization, was not carried out. A *sum-of-products* or *product-of-sums* expression in itself cannot be considered unrealistic, only the resultant circuit. Circuits should thus be generated at a high enough level to allow simplification to take place. The *pla* format [2] allows combinatorial logic circuits to be represented at a high level in



Inputs			Output
a	b	c	Z
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Fig. 3. Random circuit generation process

a format similar to the truth table representation of expressions that can be used by tools such as [3] to perform Boolean simplification.

The method shown in figure 3 makes use of the fact, that for CLEs implemented in truth table format, the functionality of the expression is determined by the binary sequence in the output column of the the table. Using a random number generator (such as the Mersenne twister [10]) it is possible to generate a random function by using the binary representation of the number as the required output in the truth table.. Although this method will produce *unrealistic* functions with redundant hardware, this can then be removed through boolean simplification to produce more *realistic* functions.

As the random number generator is only capable of generating numbers with a fixed number of binary digits it is necessary to use multiple random numbers to generate truth tables for expressions with large numbers of inputs. The random number generator selected utilized produces 16-bit random numbers that are capable of producing the necessary bit sequence for expressions of $i < 4$, for larger expressions $i - log_2(16)$ random numbers are required.

In order to make *realistic* circuits from the expressions produced it is necessary to perform high-level optimization of these circuits. The SIS synthesis tool [3] was selected to perform simplification using the minimization algorithms originally developed for the Espresso minimization tool [7]. This results in simplified expressions similar to those found in the

MCNC, and similar, benchmark suites.

Algorithm 1 PLA test strategy

```
$k$ =digits in random number
**for each** $i$ in $input\_range$ **do**
  **for each** $s$ in $samples$ **do**
    **for each** $j$ in $(i - k)$ **do**
      random[i]=rand
    construct_function(random)
    simplify
    count_product_terms[s]
  store_results[i](gather_data)
write_log
```

B. PLA Test Strategy

Using *pla* format it is thus possible to generate truth tables of combinatorial logic expressions and perform high-level optimization using tools such as SIS. It is then possible to analyze the results of this simplification process to determine the characteristics of the PLA required to implement this function. Using this method every possible expression of i inputs could be generated, simplified and the distribution of the characteristics for i inputs collected. The number of inputs could then be varied over a fixed range to allow the complete domain to be characterized. Although this method would allow the complete profiling the domain, as the number of possible logic expressions of i inputs can be shown to be 2^{2^i}, this is impractical for even fairly small numbers of inputs.

An alternative to the full testing method is to generate large numbers of random logic expressions in *pla* format that can be used to perform profiling. The suggested method of random circuit generation is shown in figure 3. In order to further limit the domain it was decided that only circuits with $o = 1$ would be considered. This avoids the need to consider situations where product terms may be shared in multi-output circuits as it is felt that product sharing would be minimal and a multi-output circuit can be modeled using multiple single output circuits.

Using the method presented in algorithm 1, a large number of i input expressions are generated and the number of product terms in the simplified expressions collected. The number of inputs is then varied over the range *input_range* and the results obtained for each of the random circuits produced. The mean and standard deviation of these results is then collected. To allow the mean and standard deviation of the p values to be compared the results were normalized by dividing the vales by 2^i over the range of inputs.

inputs	**mean**	**mean/2^i**	**stddev**	**stddev/2^i**
2	1.25	0.3125	0.6614	0.1653
3	2.35	0.2939	0.8061	0.1007
4	4.39	0.2745	1.0978	0.0686
5	8.41	0.2626	1.5210	0.0475
6	16.19	0.2529	2.1092	0.0329
7	31.41	0.2453	2.9170	0.0227
8	61.51	0.2402	4.0785	0.0159
9	121.21	0.2367	5.7370	0.0112
10	166.65	0.1627	5.2501	0.0051
11	316.35	0.1544	7.1923	0.0035
12	605.10	0.1477	9.8608	0.0024
13	1159.92	0.1416	13.2465	0.0016
14	2230.23	0.1361	18.3047	0.0011
15	4299.15	0.1312	24.8827	0.0008

TABLE I
EXPERIMENTAL RESULTS OF STOCHASTIC INVESTIGATION OF PRODUCT TERMS IN COMBINATORIAL LOGIC EXPRESSIONS

IV. Analysis of Results

The results for the normalized mean and standard deviation shown in table I demonstrate that although the mean and standard deviation increase with increasing i the relationship is not direct. The mean values presented in table I indicate the number of p terms required to implement 50% of all possible i input, single output, expressions. Thus a PLA produced with the mean number of p terms for i inputs can implement all possible expressions with less than i inputs. If the normalized mean values are considered it can be seen that the mean as a proportion of the maximum number of p terms drops as i increases. This means that area savings can be made by using PLAs with larger i values rather than several smaller PLAs to implement logic expressions while ensuring the same percentage of the total number of expressions can still be implemented.

The normalized mean values decreases rapidly until around $i = 12$, at this point, increasing the number of inputs does not reduce the mean number of product terms as a proportion of the maximum number and as such this suggests that this would be the best choice for i. This results is in line with the values suggested in [15], [4]. This suggests that the random circuits generated have similar characteristics to those found in the MCNC benchmarks.

In order to test the validity of the results obtained using the experimental method presented here the results obtained are compared to those presented in [13] for the average number of product terms in an SOP expression. Table II shows that although the values for the calculated and experimental mean agree closely there is some error between these and those presented previously in the literature. It is felt by the author that this is either due to the random circuit generation method used in [13] or due to the limited number of circuits generated. In order to verify the accuracy of these results the values obtained for the mean were used to assess the values presented in [15].

inputs	Experimental Mean	mean[13]
4	4	4
5	8	6
6	16	13
7	31	24
8	61	46
9	121	86
10	167	167

TABLE II
COMPARISON OF RESULTS TO THOSE PRESENTED IN [13]

In [15] it is suggested that based on the MCNC benchmark circuits PLAs with {12, 9, 3}, for small circuits, and {12, 18, 3} for large circuits resulted in the best area/delay characteristics. If these PLAs are considered as single output devices and the hardware resources are split evenly between each of the outputs this would results in PLAs with {4, 3, 1} and {4, 6, 1}. Figure 4 shows the Cumulative distribution function of the p terms in 4 input expressions. The values suggested in [15] are plotted on this graph and show that values suggested, 3 and 6, produce PLAs capable of implementing 20% and 97% of circuits respectively.

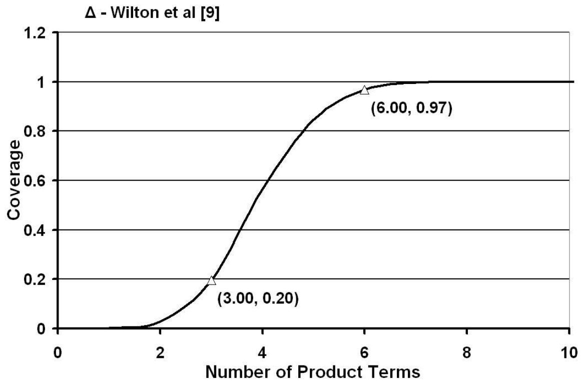


Fig. 4. CDF of product terms in 4-input expressions.

V. CONCLUSION

This paper has demonstrated a high level Monte Carlo based random circuit generation method that can be used to generate large numbers of circuits for the testing of reconfigurable devices. This method allow the high level generation of random circuits that can be simplified and optimized to give circuits similar to those in existing benchmark suites. These circuits were then used to investigate the number of product terms required by PLAs to allow the implementation of these circuits. Based on this the optimal characteristics of the PLAs can be extracted and the results obtained compared to existing data. The similarity of the results demonstrates that the random circuits generated give similar results to those obtained using the MCNC benchmarks and previous attempts to calculate the mean number of product terms in SOP expressions. These results hence validate the random circuit generation method suggested here.

As the circuits are generated at a high-level their usefulness is not limited to PLAs but can be extended to any reconfigurable device. Based on the circuits generated it is also possible to extract the characteristics of Look-Up-Table based reconfigurable devices, such as FPGAs, and other custom architectures. This means the next logical step is to extended the work carried out here to other reconfigurable architectures such as FPGAs. In this case the circuits will be used to rapidly assess the effects of design changes on the flexibility of the devices.

This method can also be used to determine a measure of the flexibility of the new reconfigurable devices similarly to the method suggested in [9] but with out the need for benchmarks to guide the generation of *synthetic* circuits. In the case of the PLAs suggested in [15] the values number of product terms would result in a device with a flexibility measure of 0.2 for small circuits and 0.97 for large circuits. In this way the flexibility of future devices could be given to allow end users to select devices with the required flexibility for the end application.

VI. Acknowledgments

This work is supported by EPSRC and SLI Ltd, and was produced at the ISLI in partnership with the Universities of Glasgow, Strathclyde, Edinburgh and Heriott-Watt.

References

[1] E. A. Bender and J. T. Butler. On the size of plas required to realize binary and multiple-valued functions. *IEEE Trans. Comput.*, 38(1):82–98, 1989.

[2] Berkeley. PLA format description. http://www1.cs.columbia.edu/ cs4861/sis/pla.txt.

[3] Berkeley. SIS download page. http://embedded.eecs.berkeley.edu/.

[4] D. Chen, J. Cong, M. Ercegovac, and Z. Huang. Performance-driven mapping for cpld architectures. *22, NO 22*, pages 1424–1431, 2003.

[5] J. Darnauer and W. W. Dai. A method for generating random circuits and its application to routability measurement. In *FPGA'96*, 1996.

[6] E. Dubrova, D. Miller, and J. Muzio. Upper bounds on the number of products in and-or-xor expansion of logic functions, 1995.

[7] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, 1992.

[8] M. Hutton, J. P. Grossman, J. Rose, and D. Corneil. Characterization and parameterized random generation of digital circuits. In *ACM/SIGDA Design Automation Conference (DAC)*, 1996.

[9] k. Compton and S. Hauck. Flexibility measurement of domain-specific reconfigurable hardware. In *ACM/SIGDA symposium on Field-Programmable Gate Arrays*, 2004.

[10] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. In *ACM Trans. on Modeling and Computer Simulations*, 1998.

[11] T. Sasao. Multiple-valued logic and optimization of programmable logic arrays. *Computer*, 21(4):71–80, 1988.

[12] T. Sasao. Bounds on the average number of products in the minimum sum-of-products expressions for multiple-value input two-valued output functions. *IEEE Trans. Comput.*, 40(5):645–651, 1991.

[13] T. Sasao. A design method for and-or-exor three-level networks, 1995.

[14] T. Sasao and P. Besslich. On the complexity of mod-2 sum pla's. *IEEE Trans. Comput.*, 39(2):262–266, 1990.

[15] A. Yan and S. Wilton. Product term embedded synthesizable logic cores. In *IEEE international Conference on Field-Programmable Technology*, 2003.