

High Integrity Hardware-Software Codesign

Adrian J. Hilton, M.A., C.Eng.

The Open University

Thesis for the degree of Doctor of Philosophy
submitted to the Department of Computing

May 21, 2004

Abstract

Programmable logic devices (PLDs) are increasing in complexity and speed, and are being used as important components in safety-critical systems. Methods for developing high-integrity software for these systems are well-known, but this is not true for programmable logic.

We propose a process for developing a system incorporating software and PLDs, suitable for safety critical systems of the highest levels of integrity. This process incorporates the use of Synchronous Receptive Process Theory as a semantic basis for specifying and proving properties of programs executing on PLDs, and extends the use of SPARK Ada from a programming language for safety-critical systems software to cover the interface between software and programmable logic.

We have validated this approach through the specification and development of a substantial safety-critical system incorporating both software and programmable logic components, and the development of tools to support this work.

This enables us to claim that the methods demonstrated are not only feasible but also scale up to realistic system sizes, allowing development of such safety-critical software-hardware systems to the levels required by current system safety standards.

Declaration of originality

I declare that no part of this work has previously been submitted to a university or other educational institution for a degree or other qualification.

I further declare that this thesis is my original work, except for clearly indicated sections where the appropriate attributions and acknowledgements are given to work by other authors.

Adrian Hilton

Relationship to published work

The following parts of this thesis have been published in refereed publications:

- Chapter 2, in particular Section 2.1 and Section 2.2, contains material that was published in “White Box Software Development” [DMH03] and “Engineering Software Systems for Customer Acceptance” [Hil03b].
- Chapter 4 contains material originally published as the paper “On Applying Software Development Best Practice to FPGAs in Safety-Critical Systems” [HH00] and later extended and developed to the paper “Mandated Requirements for Hardware/Software combination in Safety-Critical Systems” [HH02a]. The latter paper was also made generally available as an Open University research report [HH03].
- Chapter 5 was published in condensed form as “Refining Specifications to Programmable Logic” [HH02b].

In addition, Chapter 2 contains material published in the Open University research report “FPGAs in Critical Hardware/Software Systems” [HTH03].

Posters based on the material presented in Chapter 2 have been exhibited at the 2001 and 2003 ACM symposia on Field-Programmable Logic and Applications [ACM01, ACM03].

Acknowledgements

Thanks are due to the following individuals, companies and organisations without whose assistance this thesis would not have been possible.

Financial support was provided by Praxis Critical Systems Ltd. and Teleca Ltd. Dave Allen and John Cooper were primarily responsible for arranging this support.

Jon Hall, my primary supervisor at the Open University, provided endless encouragement and helpful input. Darrel Ince used his considerable experience to provide a useful second perspective on this work. Andy Vickers, my external supervisor, ensured that I kept on track and pointed me to the questions which I should have been asking.

Peter Amey, Rod Chapman and Ian O'Neill from Praxis provided expertise on SPARK Ada and the SPADE toolset. Janet Barnes, author of SRPT, gave good advice on its use. David Jackson gave useful information on CSP and ELLA.

Donald Knuth, author of T_EX, and Leslie Lamport, author of the L^AT_EX macros, ensured that typesetting this thesis was as painless as possible. Linus Torvalds and Richard Stallman provided an operating system and supporting tools which made writing a thesis a pleasurable experience. The GNAT project of Ada Core Technologies made available a high-quality free Ada compiler.

The organising committees and reviewers of the FPL 2000, FPGA 2001, REFINE 2002, RHAS 2002, FPGA 2003 and SEHAS 2003 conferences and workshops provided great forums for trying out my ideas and for finding out more about what was happening in the worlds of programmable logic, refinement and high-assurance systems.

My family and friends have been incredibly patient and encouraging while I spent endless nights holed up writing bits of thesis. Thank you. I promise not to write another one any time soon.

Derek Goldrei got me thinking about the Open University to start with, was incredibly helpful in guiding me through the application process, and was encouraging as the PhD work developed. Without him, I wouldn't have even got started on this.

Final, and most heart-felt, thanks and love to my wife Jie who by turns encouraged and bullied me into getting this thesis written.

Contents

1	Introduction	12
1.1	The History of Highly Reliable Software	12
1.1.1	Programming vs. software engineering	12
1.1.2	Historical failures	13
1.1.3	Where things go wrong	13
1.2	Modern Software Development	14
1.3	Hardware / Software Codesign	14
1.3.1	The I/O problem	15
1.3.2	Why the interfacing is hard	15
1.4	Programmable Logic Devices	16
1.5	Thesis Aim	16
1.6	Thesis Structure	16
2	Current Research	18
2.1	Safety-Critical Systems	19
2.1.1	Examples of safety-critical systems	19
2.1.2	Assessing criticality	20
2.1.3	Standards	21
2.1.4	Safety-critical market sectors	21
2.1.5	Commentary	25
2.1.6	Standards summary	25
2.2	Application of Formal Methods	27
2.2.1	The benefits of formal methods	27
2.2.2	Formal methods in use	28
2.2.3	Direction of formal methods use	30
2.2.4	Value of formal methods	31
2.2.5	The limitations of testing	32
2.2.6	Summary of formal methods	33
2.3	PLDs	34
2.3.1	Introduction to FPGAs	34
2.3.2	Description	36
2.3.3	Variants of PLDs	36
2.3.4	Specification	38
2.3.5	Device features	39
2.3.6	Current devices	39
2.3.7	Performance	41
2.3.8	Other architectures	41
2.3.9	Development environment	44

2.3.10	FPGA usage in systems	45
2.3.11	Semantics of PLDs	48
2.3.12	Issues of co-design	49
2.3.13	Summary of PLD technology	50
2.4	Programming PLDs	51
2.4.1	Netlist specifics	51
2.4.2	Process flow	51
2.4.3	High-level hardware design	51
2.4.4	High-level language implementation	52
2.4.5	Low-level language implementation	56
2.4.6	Pebble	57
2.4.7	Testing PLD programs	58
2.4.8	Summary of programming PLDs	59
2.5	Safety-Critical PLDs	61
2.5.1	Research directions	61
2.5.2	Safety of PLDs	61
2.5.3	Safety standard: Defence Standard 00-54	62
2.5.4	Safety standard: RTCA DO-254	63
2.5.5	PLD correctness	64
2.5.6	Verification	65
2.5.7	Self-testing	66
2.5.8	Emulation of PLDs	66
2.5.9	Implementation tools	67
2.5.10	Key directions	67
2.6	Conclusions	69
2.6.1	Weaknesses of current research	69
2.6.2	Research needs	69
3	Statement of Problem	70
3.1	Current State of The Art	70
3.2	Scope of Analysis	71
3.3	Target Level of Criticality	71
3.4	Levels of Rigour	72
3.5	Statement	72
3.6	Target Aims	73
3.7	Research Programme	73
3.7.1	Identified deficiencies	73
3.7.2	Maintaining existing benefits	75
3.8	Components	75
3.9	Process	76
3.10	Existing Standards	77
3.11	General Questions	79
3.11.1	Reliability	80
3.11.2	Practicality	80
3.12	Overall Process	80
3.13	Future Chapters	80

4	Development technologies	83
4.1	Synchronous Receptive Process Theory	84
4.1.1	Introduction	84
4.1.2	Deterministic SRPT	84
4.1.3	Example – AND Gate	86
4.1.4	Composition	87
4.1.5	Denotational semantics	87
4.1.6	Specification and proof	89
4.1.7	Safety monitor example	91
4.1.8	Non-rigorous components	97
4.1.9	Commentary	97
4.1.10	Alternatives to SRPT	98
4.1.11	Conclusions	99
4.2	Pebble	101
4.2.1	Introduction	101
4.2.2	Target device issues	101
4.2.3	Language elements	101
4.2.4	Example	102
4.2.5	Formal description	103
4.2.6	Completeness of definition	105
4.2.7	SRPT representation	105
4.2.8	SRPT to Pebble	107
4.2.9	Example: SRPT to Pebble	109
4.2.10	Summary	113
4.3	SPARK Ada	115
4.3.1	Introduction to SPARK Ada	115
4.3.2	Safety-critical system development process	115
4.3.3	General language properties	116
4.3.4	Static analysis and provability	121
4.3.5	Summary of SPARK	121
4.3.6	SPARK interfaces	122
4.3.7	Partial compilation	123
4.3.8	Partitioning	124
4.3.9	Compilation - a first cut	124
4.3.10	Compilation of SPARK code	126
4.3.11	Refinement	130
4.3.12	SPARK interpreter	130
4.3.13	Summary	131
5	Refining To SRPT	133
5.1	The Refinement Model	133
5.1.1	Overview of a refinement process	134
5.1.2	Suitability of model	135
5.2	Refinement for SRPT	136
5.2.1	Aims for refinement	136
5.2.2	Refinement frames	136
5.2.3	Refinement relation	139

5.2.4	Refinement	140
5.2.5	Additional refinement rules	142
5.2.6	Feasibility	145
5.3	Case Study: Carry Look-ahead Adder	145
5.3.1	Specification	145
5.3.2	Basic gates	146
5.3.3	Refinement	146
5.3.4	Space and time	149
5.3.5	Scalability	150
5.3.6	Proof means no testing?	150
5.4	Summary	151
5.4.1	Alternative approaches	151
5.4.2	Targets	151
6	A PLD Interpreter of SPARK	153
6.1	Interpreter Overview	154
6.1.1	Architecture	154
6.1.2	Partitioning issues	155
6.2	CPU-PLD I/O	156
6.2.1	Software-bus MMIO	156
6.2.2	PLD buffering	158
6.2.3	PLD readout	161
6.2.4	Writeback to bus	161
6.3	Package I/O	164
6.3.1	Arbitration	165
6.3.2	Inter-package routing	165
6.3.3	Package output	166
6.3.4	Package input	168
6.4	Package Structure	168
6.4.1	Storage	168
6.4.2	Storage operations	169
6.4.3	Program storage	170
6.4.4	Expression evaluation	171
6.4.5	CPU instructions	171
6.4.6	Instruction decoder	175
6.4.7	CPU implementation	177
6.4.8	Opcode summary	178
6.5	The Program Model	178
6.5.1	Types	179
6.5.2	State	180
6.5.3	Expressions	180
6.5.4	Alternation	182
6.5.5	Iteration	182
6.5.6	Subprogram calls	182
6.5.7	Order of execution	182
6.6	System Interface	183
6.7	Optimisations	183

6.8	Conclusions	184
6.8.1	Achievements	184
6.8.2	Evaluation of SPARK	184
6.8.3	Evaluation of SRPT	185
6.8.4	Satisfaction of target aims	185
6.8.5	Follow-on	185
7	Case Study	186
7.1	Target Aims	186
7.2	Carry Look-Ahead Adder	187
7.2.1	Simulation environment	187
7.2.2	Building blocks	188
7.2.3	Adder block	188
7.2.4	Testing	189
7.2.5	Simulation environment reliability	189
7.2.6	Conclusion	190
7.3	Missile Guidance System – Overview	191
7.3.1	Related work	191
7.3.2	System requirements	191
7.3.3	Safety	191
7.3.4	Implementation limits	192
7.3.5	Implementation technologies	192
7.4	System Components	193
7.4.1	System clock	193
7.4.2	1553 bus	193
7.4.3	Watchdog timer	194
7.4.4	Barometric sensor	194
7.4.5	Airspeed indicator	195
7.4.6	Inertial navigation system	195
7.4.7	Solid state compass	196
7.4.8	Fuel tank sensor	196
7.4.9	Proximity fuse	196
7.4.10	Millimetre radar sensor	196
7.4.11	Staring infra-red sensor	197
7.4.12	Fins	197
7.4.13	Motor	198
7.4.14	Self-destruct	198
7.4.15	Warhead	199
7.5	Design	199
7.5.1	Design decisions	200
7.5.2	Package structure	200
7.5.3	Code structure	200
7.5.4	Design limitations	200
7.6	Implementation	200
7.6.1	Development	202
7.6.2	Testing	203
7.6.3	Conclusions	203

7.7	Introduction of A PLD	204
7.7.1	Subsection identification	204
7.7.2	PLD interfacing	204
7.7.3	Transformation	205
7.7.4	Results	205
7.8	Conclusion	206
7.8.1	Refined program simulation	206
7.8.2	SPARK program development	206
7.8.3	Targets	207
7.8.4	Further research	208
8	Conclusions	209
8.1	Solving the Original Problem	209
8.1.1	PLDs in safety-critical systems	209
8.1.2	Rigorous PLD programming	210
8.1.3	Mapping SPARK to hardware	211
8.1.4	The system development process	212
8.1.5	Reliability and practicability	213
8.2	Advancement of Knowledge	214
8.2.1	Current weaknesses	214
8.2.2	Originality	215
8.2.3	Advances made	215
8.3	Self-Critique	216
8.3.1	Omissions	216
8.3.2	Weaknesses	216
8.3.3	How the state of the art would evolve without this research . . .	217
8.4	Future Work	218
8.4.1	Safety engineering with PLDs	218
8.4.2	Refinement	218
8.4.3	SPARK to PLDs	219
8.4.4	Security applications	219
8.5	Concluding Thought	219
A	Collated Refinement Rules	237
B	1553 Bus Simulator	240
C	Example Test Scripts	247
D	SPARK Report File for Nav	252
E	Original Nav Body	259
F	FPGA Nav Body	263

List of Figures

2.1	Architecture of a generic FPGA	35
2.2	PLD development process flow	52
3.1	Development process	81
4.1	Combinational incrementer	103
4.2	Pebble blocks tracking state	109
4.3	A simple stack	111
4.4	Handshaking across blocks	128
5.1	SRPT frame structure	137
5.2	Carry look-ahead adder structure	145
6.1	Interpreter architecture	155
6.2	PLD input buffer	160
6.3	TAP process	164
6.4	MMIO writeback design	164
6.5	Inter-package routing	166
6.6	Package output	167
6.7	Package RAM layout	168
6.8	ROM and PC store	171
6.9	Expression blocks	172
6.10	First stage of CPU pipeline	177
6.11	CPU core component	178
7.1	Missile system design	201

List of Tables

2.1	Table of SIL probabilities from IEC 61508	20
2.2	Trade-offs for software and hardware implementation	38
4.1	Example run for <i>AND</i>	86
4.2	Example of a trace of the watchdog	94
4.3	State changing process	109
5.1	Contrast of Morgan and SRPT refinement processes	136
6.1	Packet meaning encoding	158
6.2	Memory-mapped variable representations	158
6.3	PC action encodings	170
6.4	Word type encodings	172
6.5	CPU Opcodes	173
7.1	Adder size and delay properties	189

Chapter 1

Introduction

This chapter sets the scene for the topics discussed in the thesis. It outlines the recent history of highly reliable software development, looks at the successes, failures and needs of software engineers, and describes how this thesis tackles one particular section of those needs.

1.1 The History of Highly Reliable Software

Programming as we know it today was effectively invented in the early 1950s, when the first generation of post-war computers was frustrating the first generation of experts responsible for making the machines complete their assigned tasks. The discovery by Grace Hopper of a moth embedded in the circuits of one malfunctioning behemoth heralded future programmers' frustration in trying to find errors in their programs which had no less obscure causes.

1.1.1 Programming vs. software engineering

Programming is simply the act of producing data (a program) designed to be executed by a computer. *Software engineering* is a wider ranging term. When considering the incorporation of software engineers as members, the IEEE defined the term to mean:

... the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. [Com90]

The systematic study of software engineering is believed to have started at the NATO-funded conferences on the subject in 1968 and 1969 [Nor68, Nor69]. The proceedings of these conferences show researchers and practitioners identifying many of the problems which we still see today.

In the past three decades, Herculean efforts made by both academe and industry have led to techniques, tools and languages which permit development of complex safety-critical software projects. The systems resulting from these projects are generally as reliable as required by the user; while not perfect, they provide reasonable functionality and reliability. There is a substantial monetary price to pay for this reliability, but the reliability is generally delivered.

1.1.2 Historical failures

There have, of course, been numerous failures of software engineering. Some of them have been spectacular, such as the Ariane 5 flight control software numeric overflow which resulted in a hundred-million-pound firework display over French Guyana[Lio96]. Others have been hardly noticed by the public, but nevertheless expensive. Repeated efforts to develop a next-generation air traffic control system for the United States have met with failure after expensive failure, and the current Standard Terminal Automation Replacement System (STARS) has slipped by four years and incurred a 60% cost over-run so far. In the meantime, old software is operating far past its intended lifetime[Ins02].

The more serious failures involve human loss rather than financial loss. Remarkably, there are relatively few fatalities directly attributable to software failure. One of the earliest, and worst, of such accidents was the Therac 25 incident described in [Lev95]. A number of radiotherapy patients received massive radiation overexposure as a result of a race condition within the Therac-25 radiotherapy machine software. Notably, the fault was also present within an earlier model of machine, but a hardware interlock there prevented its manifestation.

1.1.3 Where things go wrong

The most common point of project failure is, surprisingly, in the earliest phase: requirements gathering. The Standish CHAOS report of 1995 [Sta95] and the later study by Taylor [Tay01] estimate that between 30% and 48% of IT projects fail due to requirements-related problems, even though the stage at which the projects fail is usually late in the development cycle.

A significant fraction of safety-critical software projects start to go adrift less for technical reasons than for failures of process. The Ariane 5 explosion was traced back to a numeric overflow in the flight-control software, written in Ada. This was the cue for advocates of other languages and tools to leap in and say “if only you had been using **X** you would have detected this possible overflow.” However, this misses the point. The relevant section of the software was taken from the Ariane 4 programme. It was not checked as it had been tested for Ariane 4, all known errors fixed, and had established a reliable track record. Ariane 5 flew a faster and tighter flight profile than Ariane 4, and so the numeric exception occurred where before the range of values was within the defined type range.

Using the best techniques, tools and language in the world is worth very little if your development process permits them to be circumvented, even if unintentionally. All the assertions about reliability contained in this thesis (and, indeed, elsewhere) should have a lengthy disclaimer attached, noting the need for a well-defined and reputable development process to be used, and to be enforced rigorously.

Leveson has analysed a series of aerospace accidents using an event chains model[Lev01]. Her analysis showed that accidents involving large-scale engineered systems usually have a complex series of causes, and blaming the accident on a perceived “proximal” cause is often an over-simplification:

The causes of accidents are frequently, if not almost always, rooted in organizational culture, management and structure. These factors are all critical

to the eventual safety of the engineered system. Oversimplifying the factors involved in accidents limits our ability to prevent them.[WLL⁺01]

It is important to remember this when we make claims about reducing accident rates with purely technical fixes.

1.2 Modern Software Development

Brooks [Bro95] wrote of the state of the software engineering art in 1975, and updated the 20th anniversary edition of his book with a review of the progress that the software engineering profession had made. Brooks’s original conjectures included:

1. that system development time does not scale in an inverse-linear relation to team size, and indeed that adding more manpower to a late project makes it later (the “mythical man-month”);
2. that there is no single development, in either technology or management technique, which promises an order of magnitude improvement within a decade in productivity, reliability or simplicity (“no silver bullet”);
3. that after building one system successfully, the design and development of a follow-on system is prone to balloon out with pointless features and an elephantine design (the “second system effect”); and
4. a small number of documents, in a sea of project documentation, become the critical pivots around which every project’s management revolves (“the documentary hypothesis”).

History appears to have borne out these conjectures, which have passed into everyday software engineering practice. Brooks’s forecast of “no silver bullet” in particular has proven accurate; no single technique has produced a tenfold increase in productivity or reliability. Instead, good practice and good tools have slowly increased our confidence in building software that does increasingly complex tasks.

We assume that the system development process described in this thesis is planned and carried out with an eye to these laws, and we focus on the task of producing the system that the customer needs. We do not aim to reduce the time taken to develop a safety-critical system. Instead, we aim to avoid all the extra development time resulting from having to rework the finished system after the customer or safety auditor has rejected it.

1.3 Hardware / Software Codesign

The bane of a software engineer’s life is when his code is required to interact with actual physical hardware, that is, hardware external to the computer itself; “stepping outside the sandbox”, as it is sometimes called. It is not for nothing that the writing of device drivers for an operating system is regarded as something of a black art. Why is this?

1.3.1 The I/O problem

Taking the Universal Register Machine as the canonical computer, and ignoring for the moment the unlimited memory space that it provides, we might well believe on first inspection that the machine is useless. It has a list of memory “slots”, each of which can hold an arbitrary natural number. It has an instruction counter, initially set to 1. It operates on a numbered list of instructions, each of which is one of the following:

Z(M) Zero the value in memory slot **M**

S(M) Increment the value in memory slot **M** by 1

T(M,N) Copy the value in slot **M** into slot **N**

J(M,I,J) If the value in slot **M** is zero, set the instruction counter to **I**; otherwise, set it to **J**

For any of the first three instructions, once it is executed the machine will increment the instruction counter by 1. In any case, the next step of the machine will be to read and execute the instruction pointed to by the instruction counter. If this counter points beyond the end of the instruction list given, the machine stops.

From a black box point of view, the machine does nothing – we have no inputs or outputs defined. To give its actions meaning we must be able to inspect the memory locations, control the starting of the machine and possibly also feed in new programs. This must be accomplished outside the machine’s normal operations.

It is a similar situation with embedded systems. A well-established processor – typically one of the ARM or PowerPC families – may be coupled via a bus and memory controller to a bank of RAM, and a program executed in the normal way. However, something must start program execution in some way after power-on, and the rest of the system under control (e.g. a water heating system) must be able to feed data to the processor and read control signals out of it.

Without heavy customisation of the processor, the simplest way is often memory-mapped I/O. This technique uses the memory management unit of the system to flag certain locations in the processor’s memory map as “special”; the values in those locations may either represent data read from external sensors, or be control values read by and used to control external actuators.

1.3.2 Why the interfacing is hard

The problems posed by such an apparently simple arrangement are many and subtle. The most obvious is a change in the way that we reason about program correctness. In our normal programming model any control path which may write two values to a given variable in succession, without reading the first value back, is immediately suspected of being in error.

The second problem, more insidious, is the lack of synchronisation between the software and hardware worlds. Events external to the processor may occur at any point, in any order. Inside the processor we can place bounds on the number of computational steps between two events, but introducing dependencies on external

events complicates the problem of producing highly reliable software which is correct with respect to a specification.

These problems also occur in systems where there are multiple threads of control with a shared address space. Programming languages have had to develop features such as semaphores, monitors, protected objects and associated protocols to solve these problems.

1.4 Programmable Logic Devices

Programmable logic devices (PLDs), as a compromise between a general-purpose CPU and a single-function Application-Specific Integrated Circuit (ASIC), lie on the border between software and hardware. To make a PLD program highly reliable, it must be simple; however, PLDs (such as field-programmable gate arrays) are steadily growing in size and complexity and so are being used for increasingly complicated tasks.

To date, programming PLDs has been done at a relatively low level with little concern for verifiability or correctness. However, emerging standards for safety-critical systems development such as UK Defence Standard 00-54[MoD99] and RTCA DO-254[RTC00] have started to mandate formal analysis of PLD programs that are key to system safety. Existing technologies do not support PLD programming at the higher levels of integrity.

Many of the concepts in this thesis can apply equally well to ASICs since their circuits are designed in much the same way as many PLD circuits. ASICs are also used in safety-critical systems, and many safety problems are common to PLDs and ASICs. However, the scope of this thesis is restricted to PLDs.

1.5 Thesis Aim

This thesis aims to describe a method for developing a set of functional and safety requirements into a system incorporating PLDs and conventional software. At each stage of development we aim to maintain correctness according to the requirements, and facilitate verification of the final code. The development process must be able to produce evidence that the system is fit for use at a higher level of safety integrity than is currently possible.

In this work we incorporate existing technologies for development of software for conventional safety-critical systems. We also use an existing synchronous process algebra as the basis for a formal description and refinement of a PLD program. We show how part of a conventional software program in the SPARK Ada high-level language can be efficiently compiled into programmable logic. The techniques are demonstrated in a substantial case study development of a safety-critical system.

1.6 Thesis Structure

Chapter 2 is a survey of the current research in the area of programmable hardware, and of relevant research in the areas of software and safety engineering. It looks both at the development of formal techniques for reasoning about and producing programs

for programmable hardware, and at the state of the art in industrial safety-critical software development.

Chapter 3 provides a statement of the problem which this thesis aims to address, and gives criteria by which the reader may judge whether the problem has been solved.

Chapter 4 introduces the technologies used in the rest of the thesis to address the problem. It describes Synchronous Receptive Process Theory (SRPT), the Pebble PLD programming language, a generic PLD model, and the SPARK subset of Ada.

Chapter 5 builds on the existing algebra of SRPT to construct a rigorous specification and refinement system. This system allows refinement from an abstract timed specification to provably correct implementation in Pebble. The chapter provides a worked example of a carry look-ahead adder refinement.

Chapter 6 develops an SRPT description of an interpreter for SPARK Ada bytecode, showing how SRPT can be used to design a substantial PLD program and how the known properties of a SPARK Ada program assist in its compilation into a PLD program.

Chapter 7 describes a practical gate-level simulation of the adder in Chapter 5. The chapter then draws together the techniques developed in the preceding chapters to develop a substantial high-integrity guidance system for a missile using a design which runs partly on a standard processor and partly in programmable hardware.

Chapter 8 summarises the topics discussed in the thesis, considers whether the problem statements in Chapter 3 have been addressed, and points towards further avenues of research which may follow from this work.

Chapter 2

Current Research

This chapter considers the use of programmable hardware in safety-critical systems.

We will:

- analyse current and emerging safety standards directly applicable to this field;
- describe the constraints placed on the design, production and testing of safety-critical system software, and how these may apply to PLDs;
- look at current tools and techniques used in the production of such systems, especially those related to formal methods and proof; and
- assess the effectiveness of these tools and techniques.

Since we want to use programmable logic devices (PLDs) in safety-critical systems, we will:

- describe the state-of-the-art in PLD design and production;
- examine the systems which represent the range of use of programmable hardware in industry;
- examine how PLDs are programmed in theory and practise; and
- critique the techniques and tools which claim to formalise the use of programmable logic in systems.

Finally we bring together the areas of safety-critical systems and PLDs by examining the challenges posed by the use of programmable hardware in a safety-critical system. Our guiding aim is to identify the gaps in the current industrial practice and academic theory, and to identify an approach that is able to cover these gaps.

Section 2.1 describes the practice in safety-critical systems development. Section 2.2 investigates current research in formal methods. Section 2.3 describes the range of PLD architectures. Section 2.4 investigates how PLDs are programmed. Section 2.5 looks at how PLDs could be incorporated into safety-critical systems, and Section 2.6 summarises the key points of the research survey.

2.1 Safety-Critical Systems

In [Lev95] pp 136-137, Leveson defines the term *system* to mean “a set of components that act together as a whole to achieve some common goal, objective or end” and *safety* as “freedom from accidents or losses”. The *criticality* of a system is defined by the consequences of its *failure* (“inability of the system to perform its intended function”, [Lev95] pp 172), a definition which may extend down to individual components of the system. Combining these, we may draw the following working definition:

a *safety-critical system* is a collection of components acting together where interruption of the normal function of one or more components may cause injury or loss of life.

Such systems may be designed to *fail safely* in certain circumstances. A safe failure mode is a component or system failure which does not compromise system safety. One example might be a nuclear reactor control system where any interruption of power or control to the subsystem holding the control rods will cause the rods to drop into the core, effectively stopping the nuclear reaction. So the system is not keeping the reactor running (its intended function) but it is keeping the reactor free from accidents or losses (safety).

An unsafe failure mode, by contrast, is one which increases the likelihood of accident or loss. A fly-by-wire system may not be able to fail safety, since any interruption of its normal function will cause the pilot to lose control of the aircraft.

There are other terms associated with causes of failure. A *defect* is taken to be an aspect of the design of a system which turns out to have undesired consequences; for instance, a defect of the language syntax of C is that association of single statements with conditions in a nested `if-else` block is counterintuitive.

An *error* is an aspect of the implementation of a system which is incorrect; for instance, a subprogram implementation which may use one of its variables before that variable has been initialised.

A *fault* is the result of an error or defect, manifesting in undesired system behaviour; for instance, if an aircraft engine shut down (because of an error in the software) then the unexpected shutdown would be a fault. Faults may be caused by multiple errors; conversely, not all errors may cause faults.

2.1.1 Examples of safety-critical systems

An example of a safety-critical system is an air traffic control system such as CDIS [Hal96a]. There are many components in the system including operator displays, radar and transponder devices, and communications links. It is safety-critical because if the communications links fail wholly or partially then the operators may be unable to communicate with aircraft and command course changes to avoid a collision; such a collision would be an accident and may involve loss of life or property. Hence, the system is safety-critical *when used in an operational environment*. If it were linked in to a simulator then it would not be safety-critical because there would be no severe consequences of its failure.

Other safety-critical systems may not be assessed as such, yet still cause substantial destruction or death on failure due to a denial-of-service effect. An example of this

SIL	P_{fail} (on-demand)	P_{fail} (per-hour)
4	$\geq 10^{-5}$ to $< 10^{-4}$	$\geq 10^{-9}$ to $< 10^{-8}$
3	$\geq 10^{-4}$ to $< 10^{-3}$	$\geq 10^{-8}$ to $< 10^{-7}$
2	$\geq 10^{-3}$ to $< 10^{-2}$	$\geq 10^{-7}$ to $< 10^{-6}$
1	$\geq 10^{-2}$ to $< 10^{-1}$	$\geq 10^{-6}$ to $< 10^{-5}$

Table 2.1: Table of SIL probabilities from IEC 61508

was the failure of the London Ambulance Service dispatching system which failed in November 1992; the resulting events are described in [Tea93]. Although in this case there was no link established by a coroner between the system failure and resulting deaths due to delay in dispatching ambulances, there is a demonstrable mechanism for deaths to result from a failure in normal operation (successful revival from cardiac arrest is critically affected by the arrival of a defibrillator-equipped ambulance within 10 minutes) and so the system was safety-critical even if it was not so specified.

2.1.2 Assessing criticality

Such systems may be graded according to their potential to cause death, serious injury or large financial loss. The SIL convention used in the European functional safety standard IEC 61508 [IEC00] specifies four Safety Integrity Levels (SILs), with SIL-4 systems having the greatest criticality and SIL-1 systems the least.

The SIL has two forms. For a low-demand mode of operation the SIL is calculated based on the required probability of failure for the system or component to perform its design function on demand. For high-demand or continuous operation, the SIL is calculated by the required probability of a dangerous failure per hour. The probability ranges used are shown in Table 2.1.

Example: a nuclear power station's reactor control rod system is expected to operate for 30 years (263000 hours) with a probability of dangerous control rod failure during the station's lifetime of $< 10^{-2}$. The required maximum probability of failure per hour is therefore p such that

$$(1 - p)^{263000} \geq (1 - 10^{-2}) \quad (2.1)$$

giving $p = 3.8 \times 10^{-8}$, a SIL-3 system. The calculated SIL may then be used to guide the amount and form of analysis and testing required for the system.

Other standards use similar principles of measurement, though with different notations. RTCA/EUROCAE DO-178B[RTC92], for instance, specifies levels of criticality from E (not critical) through to A (high criticality). The different treatments of risk in these and other standards were analysed by Pygott in [Pyg99].

An example of a UK commercial SIL-4 system is the Royal Navy's Ship Helicopter Operating Limits Information System [KHCP99] designed to assist landing of helicopters on Royal Navy Type 23 frigates. Failure of this system could result in the death of helicopter pilots and passengers, loss of a helicopter and damage to the ship. This is unacceptable for normal operation, hence SIL-4 reliability is required to give

sufficient confidence that such an accident will not happen during the in-service lifetime of the system. Since SHOLIS is a relatively low-demand system, this indicates a required probability of failure to perform its function on demand between 10^{-4} and 10^{-5} .

2.1.3 Standards

Makers and users of safety-critical systems in the UK have a legal mandate to ensure that the risk of serious failure is as low as reasonably practicable (ALARP.) McGee-Osborne and Hall considered this as far as it relates to the rail transport sector in [MOH97]. The Health and Safety at Work Act 1974 (known as “HSWA” or “HASAWA”) imposes general duties on employers to protect the health and safety of employees and non-employees, using the key phrase “to the extent reasonably practicable”. Thus any employer operating a safety-critical system owes a “duty of care” to those who may reasonably be affected by the system. Failure in this respect may result in any of the following:

- litigation by affected parties for damages caused;
- an enforcing order from the Health and Safety Executive requiring the removal of the system from operation or immediate modifications to the system; or
- criminal prosecution of individuals for negligence leading to harm of others.

It is notable that successful prosecution for such negligence is rare.

Since many safety-critical systems may affect public safety, governmental and associated oversight agencies have drawn up standards documents for the development of safety-critical systems. Some of the best-known standards documents are UK Defence Standards 00-55 and 00-56 [MoD97, MoD96], RTCA/EUROCAE DO-178B [RTC92], the CENELEC EN 50126, 50128, 50129 European rail standards [CEN99, CEN02b, CEN02a] and the aforementioned European IEC Standard 61508 [IEC00].

2.1.4 Safety-critical market sectors

We split the safety-critical systems market into five sectors. For each sector we describe one or more mainstream standards or guidance documents used in the United Kingdom or internationally, then summarise the main principles that have been established.

Each of these sectors has a regulatory regime which has driven the development and adoption of standards. Other market sectors such as the automotive and medical equipment industries have regulatory regimes but do not have specific standards for assessing software and programmable hardware.

In the UK medical equipment industry, for instance, the Medicines and Healthcare products Regulatory Agency (MHRA) applies UK and European law, principally the EC Medical Devices directives. These directives will require manufacturers to demonstrate that critical medical devices are appropriately safe, but does not specify a process or any specific criteria against which the equipment’s software or electronic hardware must be assessed.

Within the automotive industry, the increasing problem with faulty software has driven the development of the MISRA-C subset for critical automotive software specified in [MIR98]. However adoption of this subset is not mandatory, and indeed some of the MISRA-C rules are difficult to enforce.

Rail

The Railtrack “Yellow Book” [Rai00] provides guidance on the safety management of changes to the UK rail network. It is detailed but not prescriptive; it allows projects to tailor its recommended approach, although the Railtrack Safety Approval Body must approve the approach taken.

The CENELEC standards are derived from IEC 61508. Standard EN 50128[CEN02b] relates to the safety-related software in railway systems, and EN 50129[CEN02a] to safety-related electronic control and protection equipment. Since they are based on IEC 61508, the comments below on this encompassing standard apply.

Nuclear power generation

“Software for Computers in the Safety of Nuclear Power Stations”, IEC Standard 880 [IEC86] is intended for safety-related software in computers forming part of nuclear reactor safety systems. It lays down in detail a recommended development process, guidance on choice of language and tools, and a suggested maintenance process. The report was written in 1986, and the language and concepts used display this, but it is not yet regarded as obsolete. The very prescriptive nature of this old standard should be contrasted with the more modern standards described in this section.

The Four Party Regulatory Consensus Report on the Safety Case for Computer-Based Systems in Nuclear Power Plants [Hea97] is a set of agreed principles for building a safety case from the nuclear regulatory authorities of the UK, USA, France and Canada. It is not a standard as such, but presents the elements of a safety case perceived as helpful in gaining regulatory approval.

Aerospace (military)

UK Defence Standard 00-54[MoD99] (hereafter abbreviated DefStan 00-54) is a new interim standard for the use of safety-related electronic hardware (SREH) in UK defence equipment. It relates to systems developed under the DefStan 00-56 safety systems document or an equivalent international standard, and is appropriate if an electronic element in the system is identified to have a safety integrity level of between SIL-1 and SIL-4. This standard is covered in more detail later.

DefStan 00-55 (software) [MoD97] specifies the requirements and guidance for the development of safety-related software by or for the UK Ministry of Defence. There is very heavy emphasis on the development process and suitable documentation, but the actual requirements about the implementation method and language are few and general. There is emphasis on using formal methods wherever possible. The key message appears to be “do what is reasonable and safe, but show how your decisions were made and justify them.” This goal-based approach foreshadows the rewriting of CAP 670 SW01[Civ02], described below.

DefStan 00-56 (system safety) [MoD96] is 00-55's counterpart relating to system safety. It lays down how the safety management activities of a development program should work. A "risk class" is calculated according to how probable and severe are the system hazards, and governs how the safety activities are carried out on the program. It requires the production of a "safety case", a well-organised and reasoned justification that the system is acceptably safe.

DefStan 00-56 is undergoing a rewrite for Issue 3. The first public draft for comments [MoD03] was released on 18th July 2003. It shows that the new format will be for Part 2 (the Code of Practice) to contain volumes addressing specific issues: volume 1 describes how to interpret Part 1 (the guidance), volume 2 describes the risk management process, and the revised forms of Defence Standards 00-55 and 00-54 will form volumes 3 and 4 respectively. The standard itself is due for publication at the end of March 2004 after public comment on parts 1 and 2.

Aerospace (civil)

Penny et al.[PEBB01] describe practical experience with a "goal-based" form of safety standard in the development of CAP 670 SW01[Civ02], part of the regulations for ground-based air traffic services in the UK. They split evidence into two forms: *direct*, which directly relates to the safety of the system (such as evidence that static analysis has been carried out and no dangerous faults found), and *backing* which shows that the direct evidence is credible and sound (such as test reports and error history of the static analysis tool used).

RTCA/EUROCAE DO-178B [RTC92] is intended to provide guidance on how to satisfy airworthiness requirements for software use on aircraft. It relies heavily on software testing to demonstrate reliability. However at the highest level of software integrity the amount of testing required is very expensive.

RTCA/EUROCAE DO-254[RTC00] is the analogue of DO-178B for electronic hardware. It is a more recent document, released in reaction to the increasing complexity of electronic safety-critical hardware performing avionics functions. The Federal Aviation Authority is currently considering how DO-254 should be applied to the development of ASICs and PLD programs.

In a comparison of avionics standards, Pygott and Newton [Pyg99] compared the requirements of RTCA DO-178B with the requirements of DefStan 00-55 and DefStan 00-56. They concluded that the main difference was that civil aviation standards provided mostly recommendations, whereas the Defence Standard clauses were mandatory. The Defence Standard placed much more emphasis on the use of static analysis and formal methods, though both were mentioned in DO-178B. In addition there were mismatches between Development Assurance Levels (DALs) and SILs which made comparing standards difficult.

Pygott and Newton also noted that all of the standards reviewed did not say much about the use of commercial off-the-shelf software (COTS), which they regard as being a significant feature of new development programs.

Finance

Finance systems are rarely safety-critical, but are often business critical. There are some financial systems which have the potential to "create" money; these have sufficient

potential impact on a country's economy that their correctness is a matter of concern to the country's government. In this situation the pressures are similar to those around safety-critical systems, and so it is worth examining how these critical systems are regulated and developed to compare and contrast the approach with those used by safety-critical systems.

In the UK, the government Communications Electronics Security Group defines six levels of IT security: levels ITSEC 1 through 6 where 6 denotes the most secure systems. The ITSEC criteria are described in [Com91]. These fed into the international Common Criteria [Com99]. Like safety-critical systems, security-critical systems are classed as *high-assurance*.

Hall, in [Hal02], describes the specification and development of a Certification Authority (CA) for the MULTOS smart cards. This development was notable for the application of safety-critical software development tools (static analysis and proof with the SPARK and SPADE toolsets) in the security domain. It turned out that these techniques translated well across the domains.

The specification and security proof of the associated smartcard operating system is described by Stepney and Cooper in [SC00]. This demonstrated that formal proof techniques were mature enough to be applied to a real industrial application of substantial size, and well enough supported to be off the critical path of system development.

Cross-sector

IEC Standard 61508 [IEC00] is intended to apply across multiple industry sectors, setting out a generic safety management approach for systems with electrical, electronic or programmable electronic components. Part 2 in particular is the requirements for the electrical, electronic and programmable devices; part 3 deals with software requirements.

Part 2 ranges over a wide range of aspects of hardware, giving guidance on errors to check. A number of specified hardware faults may need to be detected (e.g. stuck-at failures for registers, bus faults and welded-together contacts) as well as properties of the software (e.g. correct "watch-dog" operation, information redundancy) with the analysis list determined by the required diagnostic coverage, related in turn to the SIL and resulting safety calculations. Interestingly, the *programmable* part of the systems is not addressed in detail; there are requirements for aspects of the design to be analysed, but no real requirements for implementation language or related aspects. It may be that the authors assume implicitly that Part 3 of the standard (software requirements) is to be applied where appropriate.

IEC 61131-3 [IEC03] applies to programmable logic controllers. These are not true PLDs, but the document provides information on controller design that may be applicable to some classes of PLD program.

A relevant comment in the HSE report [Hea97] is no. 70: "The programmable logic controller (PLC) is one typical example of an off-the-shelf system, albeit that the applications program must be provided by the purchaser. It is not sufficient *simply to show* that the production of the applications program has met the full safety system or safety-related system requirements. Such equipment typically embodies a complex operating system with which the applications software is associated. The demonstration must relate to the *full system*." (my italics.) This clearly indicates that PLC (and, by

extension, PLD) programs must be validated both stand-alone and as a component of a whole system.

2.1.5 Commentary

Standards are normally divided into a number of different types of information; legal requirements, approved code of practice (ACOP) and guidance. It is rare that a developer will follow every single recommendation; in practice they will justify their omission of one or more recommended practices on grounds of practicality and cost. It is worth noting that the second issue of DefStan 00-55 was noticeably less prescriptive than the first issue in the sense that many recommended procedures were changed to guidances; this gave each system developer more freedom to choose the development practices which were most appropriate to their particular system. There has been inconclusive debate in the safety-critical systems community about whether the reduced level of prescription compromised safety. This has been echoed in the different levels of prescription between the UK Defence Standards and the RTCA / EUROCAE documents discussed earlier.

If a procedure in the ACOP was not followed and an accident resulted then (under British law) the onus would be on the developer to prove that their differing approach was acceptably safe. Guidances may be taken merely as potentially useful suggestions for development practice.

2.1.6 Standards summary

The approach of the above standards is very general, with the exception of the 14-year old IEC 880. They tend to outline approaches rather than prescribe detailed procedures.

It is usual for safety-critical systems developers to be required to show to the system's customer or to a regulatory agency (such as the UK Health and Safety Executive) that their development process has followed one or more specified standards documents. These documents typically address the development process, configuration management, implementation language, production of safety cases, testing and maintenance issues. The system may require formal certification from a regulatory agency before it may be brought into service.

Standards evolution

UK Defence Standards undergo periodic rewriting: 00-55 and 00-56 are at issue 2 already, and issue 3 is due to appear in 2004. The rewritings reflect both feedback from practical application of the previous standards and advances in the state-of-the-practice of system development. The changes from issue 1 to issue 2 of 00-55 reflect industrial comments that the approach prescribed in issue 1 was too hard to apply in general, although at least one project was successfully developed under issue 1[KHCP99].

If experts dispute such issues, and standards documents show that conflict, how do we find a generic development process applicable to all standards? How can we anticipate the requirements of future versions of existing standards? We cannot, but we can focus on the areas of agreement noted above: the standards aim to support the process of producing a system which is demonstrably safe at a quantifiable level.

Correctness vs. safety

Demonstrable correctness is often important in a safety-critical system. Note that *correctness* is not the same thing as *safety*; a military aircraft stores management system which could never arm a bomb would clearly be acceptably safe, but not correct! Leveson's experience with an aerospace firm's torpedo was salutary:

And later, when they tested this torpedo, they told me, they called me up and said "Well you know, we took her out into this testing ground and we tested this torpedo and every time we tried to fire it, it came out of the torpedo tube and turned itself off and went down to the bottom and it just sort of lay there." And I said, "Well, it's safe." And they said, "Well the Navy didn't want to pay for this safe torpedo." [LC96]

However correctness and safety are often linked in that correct operation of a system may be key to its safety; if a release sequence for the aforementioned stores management system is faulty then armed stores may be released at too small an interval and make aircraft-proximate detonation likely.

Correctness is only meaningful in the context of a *specification*; if we take System 1 consisting of a single AND gate, and System 2 consisting of a single OR gate then both gates may operate perfectly and so both systems may naively be regarded as "correct". However the environment of the system may be such that the system is required to signal on its output wire only when both input wires are high; in this case, only System 1 would be correct.

For the above reasons we now look at how formal methods may be applied to assist us in the task of producing an acceptably safe system which is correct with respect to its specification.

2.2 Application of Formal Methods

“Formal methods” is a catch-all term for a collection of mathematical techniques used to reason formally about the behaviour of a system or component thereof. Most of these techniques are covered under two main system development activities:

verification which we define as providing evidence that a set of system requirements have been satisfied; and

validation which we define as checking that the supplied evidence is satisfactory in respect of the requirements.

Verification is therefore commonly associated with activities involving formal notations and analysis, such as those presented later in this thesis. Validation is commonly associated with unit, functional, system and integration testing, although it may also cover manual or automatic inspection of proofs produced during verification.

The number of formal methods techniques in existence appears to increase at every Formal Methods conference; for instance, FM’99 published a paper introducing the VSPEC behavioural interface specification language for VHDL [ARB99] which may be used to check VHDL designs against requirements. This method, like many others, is well-defined and addresses a specific problem. However, proportionally very few methods have gained widespread acceptance in industrial software development. Why is this?

2.2.1 The benefits of formal methods

Rushby [Rus93] wrote a seminal report on the application of formal methods to safety-critical systems. He summarises the main benefits as

- formal specification reduces or highlights design ambiguities;
- formal verification makes explicit assumptions, axioms and deductions used to conclude that a function is performed correctly, in addition to providing a substantial confidence increase in its actual correctness;
- formal verification also has the effect of closely analysing the design and highlighting implications of supposedly simple changes; and
- formal methods add an analytical component to manual reviews that may increase the effectiveness of such scrutiny.

However the report also indicates that formal methods have their flaws. Key among these include the possible disparity between the programmer’s mental model of the design and that which he or she specifies formally, especially because many formal specifications (e.g. Z [Spi92]) are hard to write or read correctly. Formal verification may also fall down in that real world properties are often hard to characterise formally.

Moreover if the verification process is partly automated then a great deal of faith is required in the software tools involved. Developing high-integrity tools is not easy, but has been demonstrated to be feasible. The development of a high-integrity compiler

for the UK Atomic Weapons Establishment [Ste98] was done using a Z specification, recast into Prolog (the implementation language). The compiler was put through a validation test by experienced compiler-breakers, and only one error was discovered; this error was in an area of the compiler which had not yet been proven correct.

Rushby concludes that formal methods should at least be in the mind of software engineers, if only to increase the rigour with which they reason about their software. Industry should be encouraged to develop further and apply formal methods, but to know when they are appropriate and when not. The report also remarks that large-scale application of formal methods in airborne software (the author's speciality) is impractical. It is instructive to note that this report appeared in 1993; the ten years following have brought significant new formal methods and techniques, notably the rise of the SPARK Ada language and broader use of static analysis tools in UK and USA aerospace software.

2.2.2 Formal methods in use

Common formal notations used in industrial projects include Z [Spi92], VDM-SL [Jon86] and B[Abr96] for set- or model-oriented specification. Variants of CCS[Mil90] or CSP [Hoa85] are used to specify and prove properties of interacting processes. Static analysis tools such as the SPARK Examiner[GC90] permit verification that programs satisfy a set of desired properties before they are run. In addition there are general-purpose proof tools such as PVS[ORS92], used for interactive semi-automated proof.

Z

Z is a formal specification language based on sets. Z usage is supported by tools such as fUZZ [Spi00] and Cadiz [Yor97] for type checking, typesetting and proving properties of Z specifications. Z has been applied successfully in a number of industrial projects, and extensions such as Object-Z have been applied to problem domains where basic Z is difficult to apply.

Z is a specification language, and was not designed with a particular method of implementation in mind. It permits proof of certain properties of and relations between specifications, but by itself does not admit a method of developing a specification to executable code; this must be done on a case-by-case basis. For example, Sennett has shown [Sen92] how Z can be used to specify a program and how then to demonstrate that an Ada program meets or does not meet that specification.

A common problem with Z is that its schemas are often written with a wide range of non-ASCII symbols which many people find intimidating and hard to read "naturally". An ISO standard for Z was released in 2002 [iec02], but until then the Z Notation Reference Manual by Mike Spivey [Spi92] was used as a *de facto* standard and indeed not all Z practitioners have read the ISO standard in detail.

B and VDM

B, as a method for specifying, designing and coding software systems, is supported mainly by the B Toolkit [Ltd98]. This is an integrated set of tools to assist the developer using the B method to develop high-integrity systems. It is based on the concept of an *abstract machine*, which is an object that may have internal variables (giving

state), invariants (making statements about the variables which must always hold) and operations (enabling other machines to operate on its state.) The B method permits refinement of machines from very abstract forms to a form suitable for implementation in a high-level language such as C, Ada or Modula. This refinement allows us to prove that the final implementation satisfies the initial specifications of the machine. The difficulty is that it implicitly assumes an equivalence between the implementation language and the language of the B method, Dijkstra's language of guarded commands [Dij76]. Languages such as C and Ada do not have a well-defined semantics, and so certain assumptions must be made by the developer.

VDM-SL[Int96] is the specification language of the Vienna Development Method. It is model-oriented, unlike Z. It is not as widely used in general as Z, but does have a history of practical use in projects such as CDIS[Hal96a].

CSP and CCS

CSP [Hoa85, Hen88] is an algebra for describing communicating processes. Each process is given an alphabet of events, and a description of the sequences of these events in which it participates. Parallel processes must be able to agree at least one sequence of events in the intersection of their alphabets, or the processes *fail* (deadlock). In addition, if a process is free to engage in an unbounded number of events not in any other process's alphabet, then that process is said to *diverge*. CSP is a useful way of describing interactions between separate systems and detecting common errors such as deadlock and diverge. Commercially its use is supported by the FDR tool [For97] which is a model-checking tool based on the theory of CSP. The developer determines whether a particular property holds for a system by writing a description of a transition system capturing this property; the tool then attempts to refine this transition system to the candidate machine and reports success (in which case the property holds) or failure (in which case the property may not hold). It can also check that a state machine is deterministic; this is an important property in safety-critical systems. Finally, it can detect potential deadlock in a system. FDR was used by Inmos to develop and verify communications hardware in the T9000 transputer and C104 routing chip.

CCS, the Calculus of Communicating Systems, is similar in concept to CSP but is more abstract and algebraic in nature. It was devised by Robin Milner and has been used in designing industrial systems including the aforementioned CDIS[Hal96a].

LOTOS

LOTOS[Int93] is the Language Of Temporal Ordering Specification. It is a formal description technique, with roots from CCS and CSP, used as an unambiguous language in standards for expressing parallel activities. It has been used to describe systems such as bus architectures and embedded systems programs. As an IEC Standard (ISO/IEC 8809) it has the strength of a well-formed public definition. Its syntax is reminiscent of CSP with alternation, input and output and parallel operators used to express parallel interacting processes. As such it shows no clear advantage for our purposes over CSP, with CSP at least backed by analysis tools.

Static analysis

Static analysis is the process of deducing properties of programs via inspection, automated or otherwise, of the program code before compilation. By contrast, *dynamic analysis* analyses program behaviour by actual or symbolic execution of the code. Technically, manual review of program code against a predefined standard counts as static analysis, although in practice the term is usually used to refer to a process which is automated or semi-automated. The “lint” checking tool for C programs[Joh78] is a widely-used static analysis tool.

Programs such as the SPARK Examiner[GC90] take advantage of a rigorous definition of their program verification criteria to perform deep static analysis checks such as well-formed program control flow, the absence of any reads of uninitialised memory and conformance to a language subset; in this case, the SPARK subset of Ada 95[FW99, Int95].

Proof tools

PVS, a product of the SRI Computer Science Laboratory, is a verification system composed of a specification language, support tools and an automated theorem prover. It has been in existence since 1992 and so can be considered reasonably mature as a tool. Rusu and Singerman, in [RS99], use PVS as a key tool to prove safety properties of reactive systems. This system uses PVS’s considerable automatic proving abilities to good effect; the user chooses the direction of his proof process, guided by the results of previous proofs, and lets the PVS theorem prover attempt to prove properties autonomously. Like any theorem prover, the key to successful PVS proofs is a supply of well-formed, relevant and precise rulesets; these are usually accumulated over time on a project, though of course they must be carefully reviewed to ensure their correctness otherwise whole proofs can be invalid.

Recently one of the designers of PVS, Natarajan Shankar, reviewed the FM industry’s progress in producing big proving engines and their success across a range of domains[Sha02]. He argues that problem-driven techniques are likely to be more effective than the uniform proof search procedures used at present. Since PVS is a classic example of the latter approach, Shankar’s arguments should be carefully considered since they appear to be based on substantial experience and evidence.

2.2.3 Direction of formal methods use

In [CW96], Clarke *et al* lay out a strategic direction for the advance of formal methods. They point out that the past view of formal methods as obscure, badly scaling and without adequate tools has now been changed and that successful industrial case studies have proven the essential practicality of formal methods. This view appears to be supported by the use of the aforementioned tools in substantial industrial applications.

Key elements of their suggested direction include *reusable* models and theories, *combinations* of mathematical theories to tackle hybrid safety-critical systems, and *integration* with the system development process. It will be instructive to assess existing techniques by these criteria, to bear in mind Rushby’s comments on the limitations of formal methods as well as their benefits, and to consider Shankar’s recommendations on proof strategies.

2.2.4 Value of formal methods

The issue of why formal methods are not currently in widespread use is tackled by Heitmeyer [Hei98]. She makes a number of interesting propositions, including the division of formal methods into “soft”, primarily passive techniques such as static analysis, and the “hard”, primarily active techniques such as interactive proof editors. This is useful because it is usually easier to persuade developers to take up a passive “soft” method requiring little training than it is to convince them to invest substantially in training and time to adopt an active “hard” method. If the formal methods community is to encourage wider adoption of the “hard” methods then they need to be able to demonstrate real and substantial benefits from them.

Example: CDIS

In [PH97] Pfleeger and Hatton discuss the issue of whether formal methods affect code quality, and if so then how. The project evaluated by the authors is the CDIS air traffic control information system [Hal96a] developed by Praxis plc. The formal methods used during development included VDM for formal specification of critical system elements, CCS to specify concurrency and finite state machines for specification of individual processes. The evaluation of Pfleeger and Hatton is that the project statistics on faults reported over time did not show qualitative evidence that code produced using formal design techniques was of higher quality than informally-designed code. However the formal specification process led to components that were relatively simple and independent, and the delivered system was measurably better than most other measured systems. The authors conclude that formal specification can be part of the solution to improving code quality but it is not the whole answer.

It is notable that the 10-year warranty period on CDIS recently expired. There was one warranty fix made during system testing at the start of the project; since then, none were required. Note also that this was achieved with the technology available in 1990.

Example: SHOLIS

SHOLIS, described in Section 2.1.2, is a commercial safety-critical system where formal methods were used. It is described by King *et al* in [KHCP99]. The development effort built upon the experience from implementing the CDIS air traffic control system, as described above and in [Hal96a]. The system was partly developed to SIL 4 standards with the rest of the system roughly at SIL 3, and around 27,000 lines of Ada code. The techniques used were Z for system specification, the SPARK Examiner static analysis tool [Bar97], and proof of system properties using Z and the semi-automatic code proof system of the SPADE Simplifier and Proof Checker [Pra98].

The technological advances over the earlier CDIS work were mainly at the implementation stage. The SPARK Ada 83 subset [Ame99] enforced by the SPARK Examiner is a significant advance on the CDIS implementation language (C); the well-defined semantics of the language permit formal proof of code properties, and the SPADE toolset partially automates such proof work to permit a higher proof productivity. Indeed, the combination of the Examiner and proof tools enabled the development team

to prove (to the standard required for system certification) the complete absence of any run-time exceptions in all of the SPARK Ada code.

The conclusions of King *et al* provide sharp contrast to the opinions expressed by Pfleeger and (to some extent) Rushby[PH97, Rus93]. Z proof was found to be *significantly* the most efficient phase at finding faults, and the ability to prove the absence of run-time errors adds extra confidence in the system. Whereas the CDIS effort was apparently unable to gain much from formal methods once the code was being written, such methods contributed to the SHOLIS effort throughout the development cycle.

Still, it is true that techniques such as proof in Z are nontrivial to use well and effectively, and require the development team to make a positive effort to undertake training and to use them properly. However they are easier to use than is commonly perceived, and the CDIS and SHOLIS projects have shown that they confer significant benefits in system reliability.

2.2.5 The limitations of testing

Testing is a vital part of system development. The main kinds of testing are:

- informal testing by developers that the feature they are developing works at least approximately as designed;
- unit testing to exercise each component of a program (typically by subprogram or module, depending on the implementation language);
- functional testing to check that all known requirements are covered; and
- system testing to verify that the entire system operates as designed without any errors.

However, we should not lose sight of what testing cannot achieve. Modern testing techniques are efficient and successful within a limited framework, but (as noted above) even the most stringent testing can miss an error that other techniques such as static analysis can detect.

Aims and achievements of testing

Dijkstra said “Program testing can be used to show the presence of bugs, but never to show their absence!” [Dij70]. Functional testing aims to show that functional requirements are met, but at best can show that no errors occur while the function is being exercised in a range of common ways.

Unit testing aims to exercise each individual component (unit) in a program. There are formal notions of how thoroughly a unit has been tested – statement coverage, branch coverage, MC/DC etc. – but the limiting factor in unit testing is often the person writing the test. They should know the required result of each test before writing it. The temptation to derive the test result from the code is substantial, so unit test results should ideally be written before the unit is written. But then, the tests are unlikely to cover all of the unit.

System testing can only realistically exercise a small section of the system's state space. Detecting and counting errors during continuous system test can give an indication of the number of *detectable* errors remaining in the system, but can never assure the developer, certification authority or customer that all the errors are gone.

Untestable conditions

SIL-4, the highest level of safety integrity, requires no more than 1 failure per 10^9 hours. Since this is just over 114,150 years we can immediately see that system testing to demonstrate this level of reliability with any confidence will likely be impractical. These limitations have been discussed in more detail by Littlewood[LS93] who applied Bayesian statistical analysis to the problem of demonstrating reliability rates through testing.

There are also more specific aspects of program correctness which are difficult to achieve by testing. Absence of run-time errors can only be shown by testing if the test exercises every path in the entire program for all values of input data. This is normally computationally infeasible.

When to test

If testing finds faults, as good testing should, those faults will normally need to be corrected and the system re-tested. The later in development that a fault is found, the more rework is likely to be required. As an example, if testing locates a fault with a system requirement then the system may need fixes to the requirements, design, implementation, and potentially many tests. This will be very expensive in development time.

Croxford and Sutton[SC95] described the economic benefits of using static analysis early in the development of the C-130J aircraft engine control software, allowing many errors to be found before testing took place and reducing the associated rework. Given this data point, it is clearly sensible to test system components as early as possible in the development process.

2.2.6 Summary of formal methods

Formal methods have been successfully used in the development of safety-critical systems such as CDIS and SHOLIS to improve the reliability of the software in the system. They can provide assurance of reliability that conventional testing alone cannot. However, the behaviour of the system hardware in conjunction with the software is harder to capture and reason about.

We will now look at one particular common component of a safety-critical system, programmable logic devices, to see how they are currently used and how we can increase confidence in their correct operation to specification at an acceptable level of safety.

2.3 PLDs

PLDs were a development of the simple Programmable Logic Array (PLA) which has been available in electronics design since the early 1980s. The early history of field-programmable logic is reviewed by Moore in [ML91]. The most common (and interesting) form of PLD in use is a Field Programmable Gate Array (FPGA).

The key characteristics of an FPGA are as follows:

- “Field-Programmable” denotes their ability to have their program contents changed upon power-up, i.e. in the field;
- “Gate Array” indicates their structure of a regular array of logic gates;
- they provide a logic device of relatively low complexity;
- they compute some function of a set of digital inputs to produce a set of digital outputs;
- they have semi-permanent state in terms of programmed lookup tables, typically implemented as static random access memory (SRAM);
- they operate mainly in a highly-parallel manner;
- they are programmed by the download of lookup table data from an external source;
- they differ from other programmable logic devices (PLAs, PROMs, CPLDs) by allowing a more complex flow of data through themselves; and
- they also differ from Application Specific Integrated Circuits (ASICs) by trading speciality of design for speed of development and economy of small-scale production.

In this section we will look at the concept of FPGAs and typical modern implementations. We will examine how they are used in real systems, and critique different approaches for producing an FPGA implementation from a subsystem design. We will also look at how an FPGA can be given a semantics, and how the integration of FPGAs with other systems presents more problems for a system designer.

2.3.1 Introduction to FPGAs

FPGAs made their first appearance in 1984, manufactured by the company Xilinx [SWCL99]. They are a compromise between a software implementation of their function (easier to program but somewhat slower) and a custom-made chip (faster and more reliable, but expensive and requiring more time to design and fabricate). A diagram of a “generic” FPGA is shown in Figure 2.1. The key components are the input and output pins, the array of look-up tables (LUTs), the routing logic, the external control and configuration loading, and the interfaces to external RAM and ROM blocks.

As a result of this compromise, FPGAs are typically used in building a prototype system in place of a custom ASIC. It is significantly cheaper and quicker to use such

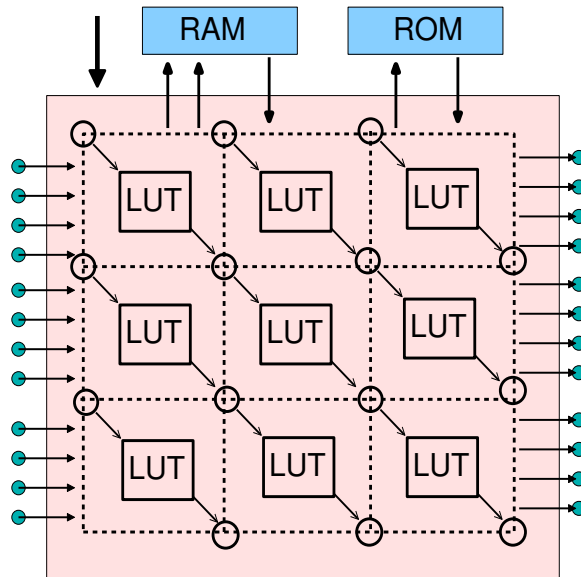


Figure 2.1: Architecture of a generic FPGA

devices when the alternative is a minimum production run of 5000 ASICs in a different company’s fabrication plant (“fab”). A small-scale single run of ASIC production can easily cost \$750,000 and take months from submission of VHDL design information to the fab to the arrival of the silicon.

There can be significant commercial gain in using FPGAs rather than ASICs. Time-to-market is reduced, since there is not the delay in setting up and making the ASIC production run, and there is little overhead if an error is subsequently found in the device. There is also the potential for increased time-in-market, providing mid-life upgrades to the FPGA code without having to replace the hardware.

FPGAs are also found in end-user products. Their ability to take processing load off the main system processor (e.g. as a bus interface) means that they provide a cheap way of increasing a system’s speed without the complexity and expense added by an ASIC or second processor. Most PC sound, graphics and network cards will feature one or more FPGAs.

For very simple combinatorial logic functions, FPGAs can be too complex a solution: devices such as Complex Programmable Logic Devices (CPLDs), or even PLAs may be appropriate.

The majority of PLDs are usually programmed in VHDL [IEE91] or Verilog[IEE95]. These Hardware Description Languages (HDLs) have substantial standard libraries, allowing a certain amount of code reuse. They model the PLD as interconnected blocks rather than providing higher-level functions such as one to operate on a data stream. Even if a higher-level language or design tool is used, it will normally compile its input into VHDL or Verilog.

FPGAs can play a useful role in system development and be an effective component in end-user systems.

2.3.2 Description

An FPGA is characterised by a collection of cells, each of which has a number of single-bit inputs and outputs. It typically uses a single clock for the whole device; multiple clocks are usually possible but seriously complicate programming. At each clock tick, the cell uses an internal lookup table to compute a function of its inputs, and possibly some internal state value, resulting in a defined output and possibly a change of state. The output is routed to other cells in a predefined manner, and new inputs are read in preparation for the next cycle.

The FPGA's interface to the outside world occurs at a set of pins, each of which is a single-bit input or output. Since the pins are normally electrically identical, each pin's function will depend on the user-programmed routing inside the FPGA. These pins are linked to cell inputs or outputs respectively; the precise linkages will again depend on the user's routing scheme.

The way that a user programs the FPGA will depend on the FPGA type. Some have SRAM cells which need to be reprogrammed whenever the device is powered up; others use Flash memory which retains data even when power to the device is removed. Both of these technologies may allow the user to reprogram the device mid-computation, with varying effects on the device's state. Some may use once-only programming (such as antifuse technology) which again retains data across power cycling but which requires a new device if the programming is to be changed.

The reprogrammable aspect of an FPGA concerns the cell lookup tables, and also the routing tables in many FPGAs. Data for these tables are loaded using special control pins to supply a stream of bits to the FPGA. The FPGA will typically be configured in a period of tens of milliseconds.

More advanced FPGAs may include small banks of random access memory (RAM) or other specialised devices such as DSP units which interface to cells. We will ignore such complications in the rest of this survey since they do not affect the fundamental functionality of FPGAs, and could be viewed as devices separate from the main FPGA circuitry; they just happen to be on the same piece of silicon.

2.3.3 Variants of PLDs

Moore, in [ML91], classifies programmable logic devices into the following categories.

PLAs

The original PLD was the Programmable Logic Array (PLA), a device whose outputs compute logical "sums of products" of their inputs. The internal structure of this device holds an array of AND gates, each of which takes a subset of the device inputs. The outputs of these AND gates are in turn fed into a number of OR gates, the outputs of which form the outputs of the device. The user programs the device by feeding a high current through certain interconnections to break them, thus selecting precisely the required inputs to each AND and OR gate.

The PLA is good for relatively simple, quick logic calculations but lacks flexibility or internal state. Some devices have additions such as registered outputs or feedback of outputs to inputs, but the basic design is simple and hence very easy to program correctly.

CPLDs

The CPLD was the logical next step from the PLA, retaining the same basic structure but with modifications to improve performance and flexibility. Larger arrays draw more power and are harder to design for a given clock speed, so the CPLD introduced an internal logic array which is structured hierarchically (blocks within blocks within blocks) and has more complex input/output logic, allowing buffering of inputs for example. The key difference from the PLA is that these devices each contain several PLAs whose outputs go into flip-flops, then are routed elsewhere in the device. These devices can support more complex calculations than PLAs, but it is still relatively easy to map designs into them; the internal data flow is not normally a design bottleneck, unlike in FPGAs.

Typical CPLDs such as the Altera MAX series [KF91] are configured using Flash memory or antifuse technology.

Systolic arrays

Systolic arrays are informally defined in [Meg94] as “an array of synchronised processors (or cells) which process data in parallel by passing it from cell to cell in a *regular rhythmic pattern*” (my italics). From this definition, an FPGA could certainly implement a small systolic array; however in practice the systolic array is often operating on data in 16-bit or larger chunks, unlike the 2 or 4 bits common at the cell level inside an FPGA. Systolic chips may contain one or more processing elements (PEs), may have a limited amount of flexibility in the precise calculations performed, and are often used in sizeable numbers in a regular array.

An example of a commercial systolic array is the SAND neural processor [Ins97], used for pattern recognition and image processing, which contains four parallel processor elements and runs at 50 MHz. It reads in data in 16-bit “weights” and “activities” streams, performs internal processing according to a 34 bit control word supplied by its sequencing chip (an FPGA in some configurations), and outputs streams of 16-bit data and addresses.

Compared to systolic arrays, FPGAs provide greater flexibility in the function of each cell and the wide range of routing possible, but their generality makes them less suitable for certain high-performance tasks such as those doing numerical calculations involving 16 or 32 bit data.

ASICs

The most complex programmable logic device is the ASIC, an integrated circuit designed for a specific task and mass-produced. While an ASIC will nearly always outperform an FPGA, FPGAs are much cheaper than ASICs in small volumes. They are also easy to reconfigure on a minute-by-minute basis, allowing one chip to perform many different functions rather than requiring one chip for each. Therefore if there is any significant chance that the function of a chip may change during the development and testing process then it is normally worth accepting the lowered system speed to replace an ASIC with an FPGA.

A FPGA draws significantly more power than the equivalent ASIC, and hence generates more heat. In compact electronic devices this can be a significant complication

	Microprocessor	PLA	FPGA	Systolic	ASIC
Speed	Slow	Medium	Medium	Fast	Fast
Unit cost	Cheap	Cheap	Medium	Medium	Expensive
Batch cost	Cheap	Cheap	Medium	Medium	Moderate
Flexibility	High	Low	Medium	Low	None
Power draw	Low	Medium	High	Medium	Medium
Program	C, Ada	Ladder logic	VHDL	Custom	VHDL

Table 2.2: Trade-offs for software and hardware implementation

since the heat must be radiated away before other components are damaged; in battery-powered devices the extra power drain may have a significant effect on battery life. For these reason CPLDs or ASICs can sometimes be preferable.

Table 2.2 contrasts the effects of implementing a given algorithm in a range of device types. CPLDs are grouped with FPGAs since their differences for these purposes are not significant.

2.3.4 Specification

A FPGA’s topology can be represented as a directed graph where each node corresponds to a cell or pin, and the arcs represent the routing. Any node without outgoing arcs is an output pin, and any node without incoming arcs is an input pin. We ignore power and configuration pins of the device in this representation. Note that the graph need not be connected. Acyclic graphs are possible; they are easier to reason about since they compute a finite-step known-duration computation of the input data. Cyclic graphs, representing loop constructs, are common in the more complex FPGA routings.

An example may be an iterative square-root real number function which takes a 16-bit positive integer representation as input X and produces the integer part of this number’s positive square root as an 8-bit output Y ; the loop construct in this case may be a successive approximation calculation, ending in a unit which computes Y^2 and $(Y + 1)^2$ and sets a “valid result” bit if $Y^2 \leq X < (Y + 1)^2$. If this calculation was non-iterative then its FPGA representation would require many more cells and each calculation of a root would take the same (i.e. worst-case) time; however, it may then be possible to pipeline calculations.

Each cell represents a function $f_c : S \times I \rightarrow S \times P$ where S is the set of possible cell states, I is the set of input values and P the set of output values. The latter are normally represented by natural numbers between 0 and $2^k - 1$ where k is the number of wires forming the input or output. This is because the relatively small calculations performed by FPGAs are normally numeric or logical in nature rather than string- or symbol-based.

The user programming defines each function f_c , and if the particular FPGA permits user-defined routing then it selects a particular graph structure from a set defined by the FPGA’s design.

2.3.5 Device features

A key factor in evaluating an FPGA device's performance and usability is its "logic gate equivalence", which is taken to be the total number of logic gates which it is possible to emulate at once. As an example, an FPGA with a 16×16 block structure, each block having four cells, and each cell having two inputs and four outputs, able to compute any function of the two inputs for each output, would have a logic gate equivalence of $256 \times 4 \times 4 = 4096$.

Xilinx define one gate-counting method in their on-line Virtex FAQ list [Xil99c]. They state that each logic cell used as logic provides the equivalent of 12 system gates, or 64 system gates if used as distributed memory (4 gates per bit with a 16 bit capacity.) They therefore will make an assessment of what fraction F of cells in a device will typically be used as memory and state that the C cells will provide $64FC + 12(1 - F)C$ system gates equivalent. Of course, there may not actually be that number of recognisable gates in the hardware; the above works on the principle of functional equivalence to a standard gate structure. Other devices such as digital delay locked loops (DLLs) contribute an arbitrary number of system gates to the count; each DLL counts as 7000 gates, for instance.

The above calculations also assume that data can be routed correctly between each cell to make each cell useful; in practice many cells will not be usable in a computation because the scarce routing resources around them have already been used. A circuit which is regular in design may not suffer from this problem, but less regular layouts will do; this is an inevitable result of the restricted size of an FPGA and the compromise between number of cells and routing resources. Therefore we should only regard "gate equivalence" as an indication of a device's size and complexity, not its usability, and in any case treat it with a degree of caution when using it to compare capacities of competing FPGAs.

The difficulty of place-and-route is shown by Inuani and Saul in [IS97]. They describe a algorithm for place-and-route for heterogeneous FPGAs based on look-up tables, in particular the Xilinx 4000 series [Xil96]. For a range of benchmark programs their algorithm improves by 10-24% the logic block usage compared to two other sets of published results, while being significantly quicker in computation time. This shows that good packing algorithms are far from obvious, even for a relatively simple arrangement such as the Xilinx 4000 series under consideration. Placing and routing for modern, more complex devices, such as Virtex, will be harder to optimise.

Most FPGAs support read-back of the programming data. This is a simple but effective way of detecting corruption in the programming bitstream. There is also the JTAG standard (IEEE 1149.1[IEE01]) for test access and boundary-scan of such devices.

2.3.6 Current devices

The main manufacturers of programmable logic devices at the time of writing are Xilinx, Actel, Altera and Cypress Semiconductor. Their mainstream devices included the Virtex and XC6200 series (Xilinx), the ProASIC 500K and Stratix families (Actel), the FLEX10K series (Altera) and the Delta39K (Cypress). We now look at 1999 and 2003 snapshots of devices from some of these manufacturers.

1999

The Virtex-E family have a gate equivalence of between fifty thousand and four million system gates by the above reckoning, corresponding to 1,728 and 73,008 logic cells respectively. The family is described in [Xil99b] and is intended principally for next-generation telecommunications systems. They are manufactured using a 0.18 micron process, and can run at internal clock speeds of up to 311 MHz. They have between 30 and 344 differential pairs of user input and output pins running at interface speeds of up to 311 MHz and so could execute a theoretical 3×10^9 operations per second on 32-bit data words. A military version of the family, the QPRO Virtex series [Xil99a], is produced using a 0.22 micron process and runs at speeds of up to 200 MHz with a third of the number of logic gates in Virtex-E. Note the lowering of peak performance and resources required to comply with military specification reliability under wide temperature ranges and high EM noise environments.

By comparison the Actel ProASIC family, described in [Cor99], can have between 98000 and 1.1 million system gates depending on system configuration, manufactured at 0.25 microns. Unlike the SRAM-based Virtex devices ProASICs use Flash memory, so can be programmed once and retain that data through multiple power cycles. They also feature a “security bit” which prevents read-back of the programmed data; this can be commercially useful because manufacturers can distribute pre-programmed devices containing proprietary algorithms without having to worry about the algorithms becoming known (directly, at least). The ProASIC internal structure is a “sea of tiles” with each tile (up to 51,200 in the larger devices) forming a 3-input logic function or flip-flop. The tiles are interconnected by four levels of routing, an indicator of the perceived difficulty of place-and-route in modern FPGAs.

The Altera FLEX10K series are SRAM-based CPLDs. An example of the series is the 10K130V part which has 6656 cells, each holding a 4-input lookup table plus flip-flop, and routing logic. These are grouped in blocks of 8 cells. Additionally there is 32Kb of memory on the device and there are 464 user I/O cells.

2003

In 2003 the Xilinx Virtex family is still going, although the lead device is now the Virtex-II Pro (XC 2VP125) at 125,000 logic cells, with 42Mbits of config data and up to 1200 user I/O pins. It incorporates up to 4 PowerPC processor cores and 556 18x18 multipliers.

Altera have launched their Stratix architecture, described in [LB⁺03]. The architecture itself was evolved through a repeated posit-and-evaluate process where Altera engineers proposed designs; these were modelled and benchmark circuits compiled onto them. The aim was to produce a device that enabled circuits to be routed even when most of the logic cells were used up – a notorious problem in the FPGA world. The lead Stratix devices have 114,000 logic cells, 10Mbits of memory. The devices and associated tools support many high-speed I/O standards since FPGAs are commonly used to pull data straight off a high-speed bus.

There has been no recent significant change in the forms of the designs of FPGAs marketed by the major FPGA manufacturers, though the Altera approach to producing a new architecture is interesting; they seem to be aiming to solve old problems better rather than looking for new problems.

2.3.7 Performance

The maximum attainable clock speed of FPGAs has been increasing roughly in line with the decreasing process size. Note that the quoted speed of an FPGA is often an order of magnitude more than that actually achieved. This is because normal compiled system implementations require substantial cross-chip communication and so several iterations are required for the data to make its way across the chip. The Virtex series for instance has a general routing matrix (GRM) associated with each logic block, each of which routes to adjacent and 6-distant GRMs in north, south, west and east directions. There are 12 “Longlines” running the full length and width of the device for fast long-distance communication. In addition, the “VersaRing” routes between the I/O pins and the logic blocks. These four different interconnection schemes make routing very flexible, but at the same time very hard to reason about compared to a homogeneous grid with only nearest-neighbour connections.

This communication feature is a major weakness of FPGAs. They require major effort to be put into placement and routing of designs in order to come close to their maximum efficiency. For this reason good comprehension of the information flow in a program is vital in producing an efficient FPGA implementation, hence allowing a smaller and cheaper device to be used. This is similar in some respects to the problems involved in deriving an efficient data flow through a systolic array, in that the arrangement revolves around dependency information. The difference is that the systolic array pipeline is normally replicated many times in order to increase performance across many devices, whereas in each piece of mass-produced equipment using FPGAs the number of FPGA devices is normally few in number.

The difference which an FPGA architecture makes to design algorithms and software is illustrated by Hartenstein *et al* in [HHG98]. The authors explore the difficulties posed by the architecture of the Xilinx XC6200. The main difference between this device and other FPGAs is that the device has a 32-bit data bus which allows a coupled processor to read or write directly registers in the FPGA; in addition, routing resources of the device are limited. The authors conclude that the restrictions of the vendor tools for the device and its structure indicate that designs should be partitioned into a control part and a datapath. The key fact to emerge, however, is that the FPGA architecture affects the development process right from the point of synthesising the behavioural VHDL into the target’s primitive gates.

2.3.8 Other architectures

Multiple-Contexts

An additional feature for FPGAs was explored by MIT with the design and construction of their “Delta” Multi-context Programmable Gate Array (MPGA) [TEC⁺95]. This has an additional pair of control pins which distribute a “context” value across the chip. Cells and routers may use this value to select one of a set of lookup tables. In practice this permits an MPGA to switch between several different functions in a couple of clock cycles, rather than requiring the tens of milliseconds normally required to reload lookup tables. Since the area of an FPGA chip increases much more quickly with number of cells and routing complexity than with cell size, this appears to be generally advantageous for FPGA design. For a given chip area, an MPGA design

should be able to implement a more complex set of programs than an FPGA design.

In [FMA⁺97], Faura *et al* present a RAM-based FPGA with two configuration contexts. It has the important property of allowing reconfiguration of one context while the other is active; this allows a switch between dynamically-loaded configurations within a couple of clock cycles. This system, termed FIPSOC (Field Programmable System On-Chip) couples the FPGA cells with a microprocessor core. Tellingly, the main digital I/O of the chip is routed through the FPGA cells first rather than through the microprocessor, and the FPGA cell outputs are mapped onto the microprocessor memory space. The system has clearly been designed with fast throughput in mind, so the (negligible) cost of a context change will be important.

The other important fact to arise from [FMA⁺97] is that the extra chip area taken up by an additional context is not prohibitive; the implementation of the Digital Macro Cells (DMC) uses around 56% of its space for context-related storage and processing, with a roughly 50-50 split between contexts, so the cost of the extra context can be estimated as a 30% increase in DMC area.

Tight Binding to Processor

An alternative to a separate FPGA device is to bind it more tightly to the main system processor. This was the approach described by Hauser *et al* in [HW97] with their work on the Garp processor. Garp is a standard MIPS processor with a slave reconfigurable array incorporated on the same piece of silicon as the processor. The suggested method of use is for the main processor to handle normal execution itself, with programs handing off certain computationally-intensive tasks to the reconfigurable array. The reconfigurable array is programmed by feeding an array configuration into a “configurator” program which outputs a set of configuration bits; these are used to generate C code which is compiled into a standard program and executes at the appropriate point to write the bits into the reconfigurable array.

Garp was faster than an UltraSPARC 1/170 by factors of 24, 9 and 2 for their benchmark computations of DES, image dithering and array sorting respectively. These were reasonable, but this was a simulated run of a Garp, and the programs were no different from standard FPGA benchmarks. There was no clear indication given in [HW97] that a Garp chip was better than a standard processor interfacing to an FPGA over a PCI bus. The authors suggest that Garp would be more easily adopted than FPGA-only machines, but offer no evidence to support this claim, and do not address the FPGA-PCI configuration which seems to be in common use.

Donlin describes in [Don98] an architecture called “Flexible URISC” which breaks down a CPU into a bus on which sit arbitrary logic units; the controller of the architecture has only one instruction, `MOVE x y`, which moves the contents of location `x` to location `y`. All more complicated processing is done by the logic units whose input and output registers are mapped into the processor memory space. Such an interface sits well with FPGA devices like the previously discussed Xilinx XC6200 series. In fact, a prototype core has been implemented using XC6200 devices. However, the performance gain of such an architecture is still not clear, and programming of the prototype must currently be done at the instruction level. It appears to be an interesting development, but lacks an obvious application, and none is suggested in [Don98].

Graham and Nelson, in [GN99], describe the simulated coupling of an Analog De-

vices SHARC DSP with a Xilinx 4000-series FPGA architecture. Their reasoning for this coupling is that DSPs have a memory architecture permitting many independent memory ports to the programmable logic - a key to increased performance. The programmable logic is seen as a way of performing the tasks to which DSPs are ill-suited such as bit-level data manipulation. While the performance increase vs. area increase figures are estimated rather than taken from actual trials, they estimate that increases in chip area by between 0 and 60% can typically speed up DSP benchmarks by factors of between 4 and 6. Of course, actually programming such a system is far from trivial.

System-on-Chip

The trend towards widespread use of small hand-held devices such as the PalmOS and PocketPC Personal Digital Assistants (PDAs) and the late-second generation cellular phones has driven a requirement for compact low-power microcircuitry with substantial computing power.

A typical cellular phone has five major components: the aerial, the screen, the keypad, the battery and the circuitboard. The screen and keyboard sizes are determined by user interface issues such as eyesight and fingertip size, and the aerial by the need to be able to receive and transmit a signal to a network cell at a typical distance. Battery technology is improving, but innovations such as colour screens will continue to increase power requirements. Hence the obvious place to look for space and power saving is the circuit board.

One solution is to incorporate the maximum amount of logic on a single custom integrated circuit rather than placing a number of generic ICs on a circuit board. This is practicable in a cellular phone because of the large number of phones produced. This approach is called "System-on-Chip", abbreviated 'SoC'.

An example of SoC is the DReAM architecture, described by Becker *et al* in [BPG00]. DReAM couples a number of reconfigurable processing units (RPU), connected together directly and then interfacing to other components on the chip (DSP, memory, microcontroller) via dedicated I/O units and a bridge. The authors have mapped a CDMA "rake" finger onto four RPUs, in a DReAM architecture running at 100MHz. This is a classic off-loading of a computationally intensive operation from the DSP or CPU, and indeed the rake is an important part of the operation of third generation WCDMA mobile phones.

SoC can deliver increased performance in a system and reduce the component count, at the cost of increased silicon area and hence losing several of the financial benefits of using mass-market PLDs.

Non-silicon Architectures

The use of reconfigurability is not restricted to silicon. McCaskill and Wagler, in [MW00], describe the design of a reconfigurable microfluidic network where routing is controlled by magnetic or photonic activation. The actual processing elements can mix, separate, react, detect or simply transport different fluids. These designs are not produced on silicon, but rather in materials such as polymer. The actual reconfiguration would be handled by a digital mirror which reflected ultraviolet light onto appropriate parts of the network.

Such devices would be expensive to fabricate, at least at first, and the obvious question is whether there is a need for them. McCaskill and Wagler suggest programmable biochemistry as one field which might find such devices useful. The programmable logic research community should track the future progress of this class of device to see whether it solves any technical problems of silicon-based PLDs.

2.3.9 Development environment

An old but widely-used Xilinx device family is the XC6200 series [Xil97]. Xilinx produced the XC6200DS Development System based around a device from this family, the XC6216. [NG97] describes this development system. It is aimed at developers who want to produce applications based around the XC6216 device (64×64 logic cells, 1 register per cell), The key components of this system are:

- XC6216 device on a standard PCI board, coupled with up to 2 Mb of SRAM;
- extra PCI mezzanine slots on the board for custom hardware;
- XACTstep Series 6000 graphical design tool, reading EDIF format design input;
- Java and C/C++ run-time support software which interfaces to the board; and
- *WebScope* graphical debug interface to the XC6200 device.

The development process involves the user deciding what task the device is to perform, designing the XC6216 configuration using XACTstep, saving the resulting configuration data on the PC, then writing his or her control program which is linked with the supplied run-time support software. When run, the program will read the stored configuration data and upload it to the XC6216 device, then start the user's task. At any point the user will be able to use *WebScope* to check the configuration and register state of the XC6216.

Analogue design

This process is adequate for systems which are experimental, but the hardware and software design processes are very different. The software design and development (in Java or C/C++) expresses the programmer's intent at a relatively abstract level where the details of the target machine do not greatly affect the programmer. The hardware design is done at a much lower level, analogous to programming software at the machine code level; the machine is being told precisely how to do a task rather than what task needs to be done. Here the programmer is having to be his or her own compiler; since modern compilers such as gcc [Fou00] are regarded as reliable and very efficient in terms of size and speed of code produced, the programmer is likely to be poor in comparison.

We expect that many of the errors in a programmed system's execution will arise at the hardware / software interface; incorrect handshaking and erroneous mapping of FPGA outputs to software variables are the two most obvious classes of error. This is because we will generally express the requirements for a system at a high level, then decompose them as the system itself decomposes into hardware and software parts; the

interface between hardware and software does not have any requirements to start with, but rather such requirements emerge as the system is implemented. This means that the requirements have to be applied retroactively to the parts of the system that were implemented before the requirement emerged, leading to parts of the system that are overlooked or are incorrectly changed.

For a safety-critical system, such a development process is clearly inadequate. The emphasis (as shown by the inclusion of *WebScope*) is on getting a program which compiles and runs, then debugging the hardware and software components until sufficiently few errors are apparent for the program to be regarded as effective.

The contrasting processes

The requirements - design - implementation - unit test - integration test cycle typical of safety-critical projects conforming to DefStan 00-55 and RTCA DO-178B[MoD97, RTC92] implies that we need to understand completely how the hardware and software parts interact before we start to implement them. While a certain amount of iteration through the cycle may be necessary due to changing requirements or unforeseen system limitations, the emphasis is on getting the system's behaviour correct by design.

Sutton and Croxford [SC95] describe how this "correctness by construction" approach was been shown to save time (and therefore money) in development of a new avionics system for the C130J Hercules II aircraft, while achieving a specified level of system reliability. We have previously discussed the limits of confidence that can be attained by testing. An analytical rather than empirical approach is to be recommended.

2.3.10 FPGA usage in systems

FPGAs are used in many common electronics systems. They are used to implement "glue logic" and bus interface protocols such as PCI [AASR98]. In these systems their relatively small size and well-defined specifications enable testing to demonstrate quickly that they are adequately correct for the level of integrity required. However FPGAs have also been adopted for use in certain specialised computing machines, as described below.

Custom Machines

SPLASH and SPLASH 2 represent a previous generation of FPGA technology (the Xilinx XC4000 series in the case of SPLASH 2, developed between 1991 and 1994). They were large architectures consisting of 16 or more FPGAs coupled with each other and with banks of RAM. We focus on SPLASH 2, detailed in [BAK96].

The design of SPLASH 2 had FPGAs as atomic processing elements, each coupled with 512 Kb of fast static memory. The FPGAs were connected by crossbar switches in groups of 16, each group forming one element of a linear array.

SPLASH 2 was used for several distinct tasks: a major one was searching genetic databases at a rate of 5-12 million characters per second, obtaining several orders of magnitude performance increase compared to its contemporary workstations, while priced in the \$40,000-\$60,000 range. It was also trialled for fingerprint matching, which was previously done by very expensive custom computing machines. This task involved image processing to extract the skeleton features of a fingerprint, a very different task

to the text matching that the database search required. Again, the performance increase over a contemporary workstation (SPARCStation 10) was a factor of 1500. This increase would have scaled well if more processing boards had been added, speed being approximately proportional to the number of boards.

SPLASH 1 had been programmed at the logic gate level, but the difficulties that this posed to the programmers meant that the developers designed a higher-level software environment for SPLASH 2. The main development language was VHDL, coupled with automatic synthesis and simulation tools. The designers chose not to use a C subset, on the grounds that writing a C-to-hardware compiler would have taken effort away from the mainstream of SPLASH development. Arnold [Arn96] writing in [BAK96] notes that the developers believed that “the best model for custom computing machines is to develop higher-level programming languages that can be compiled into a form suitable for input to commercial CAD tools.” We examine this later in Section 4.3.10.

SPLASH 2 showed that a custom FPGA-based computing machine could significantly outperform a workstation for certain tasks, and yet be flexible enough to do very different tasks equally well.

Specialised Processing Elements

A practical use of the characteristics of FPGAs is described by Robinson *et al* [RCD98]. Their RCA-2 board, incorporating three Altera 10K130V CPLDs, is designed to process blocks of signal data at rates of 100 Mbytes per second or greater. The CPLDs are given local and shared SRAM, and programmed with signal processing algorithms. This is a near-ideal application of programmable logic; the CPLDs give a flexibility unattainable by ASICs or systolic arrays, are sufficiently fast to process the data at the given speeds, and the circuit board is less complex than would be required to implement the processing with a dedicated microprocessor. The resulting data can be passed down low-bandwidth lines for more leisurely and detailed processing.

A different application obtaining similar benefits is cryptography. Charlwood and James-Roxby [CJR98], implement encipherings such as Blowfish-16 [Sch94] in an XC6216 device. A 20 MHz non-pipelined implementation attained 119 Kb/s throughput, which translated to an 8 Mb/s pipelined implementation. For comparison, a contemporary 300 MHz Pentium II processor attained less than 25% of this performance. One obstacle was the number of cells required by a pipelined implementation: over 4000, as opposed to 603 for the non-pipelined version. We see from this that FPGAs can give significant performance gains over conventional microprocessors, at much lower clock rates. This gives us a motivation for incorporating FPGAs in high-performance systems.

The contenders in the recent Advanced Encryption Standard (AES) contest were specifically evaluated for their suitability for implementation in hardware. Chodowiec *et al* [CKG01], described pipelined implementations of four of the contenders. The use of mixed inner- and outer-round pipelining enabled implementation of the contenders in a Virtex XCV3006 device, at throughputs of 7.5 to 16.8 Gbit/sec. Mixed architecture sharply increased CLB slice usage, by around an order of magnitude over inner-round pipelining. More recent work by Järvinen *et al* [JTS03] has improved this to a practical implementation at 17.8 Gbit/sec on an existing device using a fully pipelined memoryless design.

In contrast to the above performance gains, Shand [Sha97] examined the task of finding approximate solutions of over-constrained systems of equations over the Galois field $GF(2)$. He compared the DECPeRLe-1 FPGA co-processor [VBR⁺96] with a 150 MHz Alpha 21064, which was the approximate contemporary of the DECPeRLe-1 in terms of technology. While the FPGA machine (16 Xilinx 3000-series devices) was faster by a factor of 60 in the search for a particular data set, the Alpha software can be optimised for a particular data set and recompiled in seconds to close the gap to a factor of 2 or 3. Recompiling the FPGA program data to be data-specific would take tens of minutes, losing any advantage. So for these kind of isolated problem solutions the FPGA is superior by far; for repeated solutions for different data sets, the FPGA recompilation overhead becomes significant.

These studies have shown that FPGAs can confer a significant speed advantage over conventional microprocessors; however, the performance gain appears to be sensitive to the specific problem.

Plug-In Boards

Boards designed to be plugged into standard PCs are commonly used to research the programming and use of FPGAs. A typical research FPGA board is Riley-2, described in [MCLS97].

Riley-2 is a PCI board with four Xilinx XC6216 FPGAs, each coupled with 512 Kb of fast memory. There is also a RISC core (Intel i960JF) on board, and 16 Mb of shared memory. The XC6216s can be controlled directly by the i960 chip because their configuration bits are directly accessible in the i960's address space. There is also a 44-pin external I/O connector for external hardware such as video.

The FPGAs on Riley-2 are programmed in Cedar, an extension of C for parallel hardware similar in many ways to the Handel-C language described in Section 2.4.4; the i960 and the PC host software are normally written in C or C++. It allows the use of multiple dynamically reconfigurable FPGAs rather than a single FPGA, experimenting with shared vs. private memory, and partitioning tasks over multiple FPGAs.

Emulation

FPGAs are often designed into systems which also contain high-performance logic chips. These may be microprocessors, but may also be ASICs. In the design of these logic chips, emulation is an important step in validating the design before it is sent to be etched into silicon. Krupnova and Saucier, in [KS00], survey the commercial emulation systems in existence which are based on FPGAs. Compared to custom chip emulators, these give the key characteristic of high performance, although require CPU-intensive compilation of the simulation programs. Krupnova and Saucier regard the FPGA pin count as the limiting factor in their use, although pin multiplexing can overcome this to some extent.

A modern FPGA-based machine for emulation is BEE[CKRB03]. BEE is a custom machine built with 20 large Virtex-E FPGAs and copious I/O, connected to a network by a commodity controller card. It makes practical the emulation of a 10-million-gate ASIC at 60MHz in real time, using up to 90Gbit/sec of data, running at over 200Gops/sec.

2.3.11 Semantics of PLDs

The incorporation of programmable logic devices into safety critical systems brings with it a need to be able to reason formally about safety and partial correctness in the context of programs executing on the device. Here we have three distinct needs for a semantics of FPGA operation. It will enable us to:

- demonstrate that “programs” (data programmed into FPGA cell and routing look-up tables) satisfy their specifications;
- refine high-level designs into code while demonstrating semantic equivalence; and
- reason about behaviour at the interface between software and programmable logic.

The cell-and-router structure of an FPGA device leads us to consider a collection of small individual processes reacting to input signals to produce output signals, since this is essentially what is happening when cells are viewed as processes and their routing is viewed as describing which signals pass to which process. Since such FPGAs may normally be clocked by a single chip-wide low-skew clock signal to all logic blocks we can add the additional constraint that the system be synchronous, at least from the point of view of the cells; in reality, the interface between the FPGA and an IC such as SRAM may not run at the same clock rate.

A model which is simple but sufficient to describe synchronous FPGA programs, and which has a rigorous semantics, is Synchronous Receptive Process Theory. This is described in [Bar93] and was developed from Josephs’ Receptive Process Theory [Jos92]. It is similar in some ways to CSP, but better expresses the synchronous and fundamentally receptive nature of logic gates: CSP allows processes to refuse events (inputs) whereas actual gates cannot normally exercise any direct choice over the inputs that they receive from cycle to cycle. We explore this in much more detail in Chapter 5.

Another formal representations which could be used is Timed CSP [SD95]. Timed CSP is an improvement on standard CSP since it can express the concept of an event occurring within a specific time (e.g. a clock cycle) whereas CSP can only have a known event happening or not. Timed CSP uses the “maximal progress” mechanism where an event happens whenever all participants are ready to engage in them. Representing an FPGA program in Timed CSP would certainly be possible. However Timed CSP is more complex than SRPT, allowing as it does asynchronous events. We are looking for the simplest possible model which is sufficiently descriptive for our purposes.

Z has been mentioned before as useful in the specification of complex systems. We could attempt to use it to describe an FPGA program. It is certainly worth considering in terms of *specifying* the whole program, and perhaps even parts of the program, but we would like our specification to be easy to refine into an implementation in something like Pebble or VHDL. As previously noted, Z does not in itself provide mechanisms for refinement, and the task of developing such a rigorous refinement mechanism would be considerable.

A similar refinement-based objection can be raised for the B-Tool; it provides its own target language, but this language is imperative and provides no native support for parallelism.

One promising unified theory is *Circus* [CSW02], an integration of the CSP process algebra and the Z specification language. This uses a Z schema to describe the state of each process and CSP-like action to describe the control behaviour of each process. *Circus* has well-defined refinement rules for transforming specifications from abstract to concrete form.

Circus is appropriate to a development process at a higher level than SRPT. It provides a way to refine down from an initial abstract specification to a collection of relatively independent processes, omitting specific timing descriptions as long as they are irrelevant.

Circus is as yet untested in an industrial-scale development; nevertheless, its framework and the rigour of its specification and refinement laws show promise for practical system specification.

2.3.12 Issues of co-design

An FPGA is almost always only a computational component of a system. Other components may include one or more microprocessors executing software, a bus (such as PCI or the military-standard 1553) and other specialised devices on the bus. When designing the system architecture, an important question to resolve is “how shall we divide the work among the components?” This decision will affect the critical system properties of speed and reliability. The trade-offs to consider include timing constraints, cost, complexity, redundancy, component functionality and required reliability.

At the moment the decision on what work to allocate to FPGAs is relatively simple to make. Their small size means that very specialised tasks such as bus interface logic are ideal, and most other tasks do not suit their capabilities. FPGAs work best on processing large amounts of data in a simple way, which is not a common task in most safety-critical systems; where it is required, such devices as DSPs are currently used. However the increasing capacity of commercial FPGAs will enable them to undertake increasingly complex tasks, taking load off the main processors of the system.

Partitioning Software

The decision on how to split software between a conventional microprocessor and programmable logic relates closely to the field of hardware-software codesign. There are three basic choices about when and how to partition the software:

- at design time, manually;
- at compilation time, semi-automatically; or
- dynamically during execution.

The first option is self explanatory. When the system is designed, the design team decides which functionality should be in programmable logic and codes it explicitly. As noted earlier, common implementation languages are VHDL and Verilog.

The second option has the software implemented in some high-level language. During compilation sections of the software are selected for programmable logic according to some defined criteria, and extra “glue” logic is added to allow these sections to

communicate with the other software sections. The manual effort in this selection and mapping can vary from none to total.

The final option has a set of software sections implemented in netlist format. There will be a system controller which dynamically loads the netlist data into reprogrammable logic as required. This normally requires an FPGA capable of on-the-fly reconfiguration, since otherwise the device will have to be power-cycled and interrupt system execution. With current technology we must have constructed our library of programmable logic routines beforehand; the place-and-route overhead is usually too high to make any other approach practicable.

Note that the decision on when to partition is coupled with the choice of implementation language. If we are to decide partition details at compile time or later then we need a language amenable to translation into a HDL or netlist (normally EDIF) format. The choice of language will depend on the compilation tools supplied by our device vendor.

2.3.13 Summary of PLD technology

PLDs exist in a wide variety of designs and sizes. Their most common form for use is the FPGA. They are widely used as glue logic, and have been used for specialised processing tasks where an ASIC would be too expensive and a conventional microprocessor too slow or too complicated.

We will now look at how PLD programs may be designed.

2.4 Programming PLDs

The implementation of a PLD-based system can be done in many ways. The target “object code” will be a vendor-specific “netlist” which specifies the data to be loaded into each cell and router of the device. To reach netlist form, several intermediate compilation steps are normally required.

2.4.1 Netlist specifics

EDIF (“Electronic Design Interchange Format”) [Int00a, Int00b] is a textual language designed to allow electronic design information transfer between different CAD systems. It is currently implemented for netlist and schematic circuit descriptions, although different tool vendors have significant differences in their implementations so it is not as portable as it could be. Netlists are often stored in EDIF.

A common step in most PLD programming methods is compilation from a HDL to the netlist; device vendors normally supply software to do this as part of the device toolkit. This compilation has the advantage that the source program structure is similar to the target structure since HDL designs are normally expressed in terms of procedural logic functions. These functions map naturally onto the FPGA cells-and-routers model. However for larger systems it is hard to ensure that a large and complex low-level design satisfies the system specification. Note, too, that HDL and the netlist have to deal with issues of clock signal distribution and skew across the chip, driving of inputs and outputs, and other VLSI-related issues; such complexities should ideally not appear in higher-level descriptions.

2.4.2 Process flow

A typical PLD development process flow is shown in Figure 2.2. It illustrates the key steps and decisions that need to be made. Note that the flow may be changed because of different project needs. In a safety-critical system development there would be safety case work going on in parallel which would exert a substantial influence over design and implementation decisions.

Where the PLD is expected to interact with system software there would be integration work to ensure that the two components worked correctly together.

2.4.3 High-level hardware design

Substantial effort was made in the 1980s and 1990s to develop a hardware design language that supported formal reasoning and abstraction, two features absent from HDLs such as VHDL and Verilog. The main exponent of this approach was ELLA[MC93], a non-proprietary language with a formal basis.

ELLA was not a strict competitor to VHDL and Verilog, but in practice it was treated as such. The relatively small size of hardware designs made design in existing HDLs feasible, if not optimal. It may be that, as hardware designs and PLD dies continue to grow in size, high-integrity requirements will make ELLA or similar design languages more necessary. This change was seen in software with the emergence of structured design methods as program sizes grew beyond what one developer could

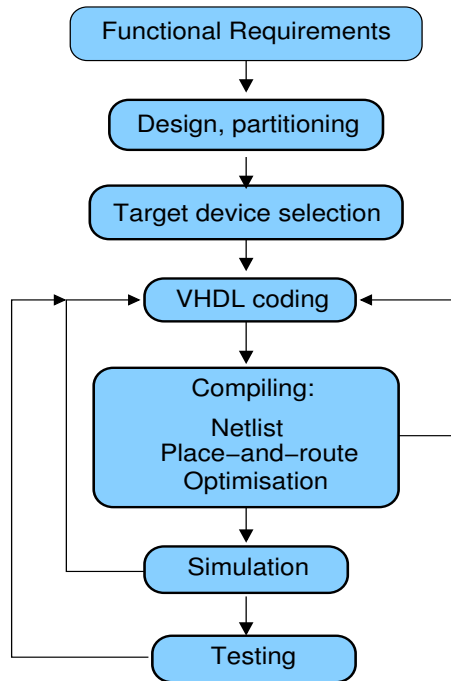


Figure 2.2: PLD development process flow

manage; it is reasonable that a similar effect will eventually be seen in programmable logic program design.

2.4.4 High-level language implementation

The use of a more abstract implementation language for PLD designs has received considerable attention and is emerging as practical for some industrial applications. The two-step compile (i.e. initially compiling to an intermediate language) is relatively easily achieved since a number of languages have been compiled into VHDL; of note are Ada (in [She96, WA02a]), Java (in [MK98]) and C (in [Swe98, She96]). Below we analyse the results of this work.

Fine-grain vs. coarse-grain parallelism

A key property of programmable logic systems (each system incorporating both the hardware and programming interface) is the *granularity* of the possible parallelism. *Coarse-grain parallelism* is represented by programs which have individual data spaces and communicate via specialised protocols. *Fine-grain parallelism* is represented by subprograms which share a single data space and rely on careful programming by the user to avoid race conditions.

The fine-grain model is a better representation of a typical PLD program, where the limited space on the device may be used most effectively by a large number of simple parallel computations which share data wherever possible; duplication of data storage (the way of coarse-grain parallelism) wastes device space. We believe that aiming for fine-grain parallelism from the outset holds the key for a significant general increase in the use and speed of PLDs.

Java

The JVX Java prototyping system [MK98], for instance, compiles a single method in isolation to VHDL and uses a modified JVM interpreter to interface with any methods in reprogrammable logic. However, it appears that it is not currently possible to compile into hardware any method which calls another method. The automatic interfacing between the JVM and the FPGA is an interesting step from the point of view of partitioning; the user need make no special changes to a method for it to become VHDL. However it is not clear that the Java language itself gives any significant benefit to the effort.

Snider *et al*, in [SSC01], map a generic object-oriented language (subsets of C++ and Java are given as examples) directly into device configuration data for Virtex devices. The general approach is to write classes that extend a `Machine` base class, taken to be the smallest unit of execution. Functions `step()`, `input()` and `output()` define the machine's actions. The compilation relies on heavy optimisation in order to extract fine-grained parallelism from the user's medium-grain specified parallelism. Low-level optimisation specific to the target is then performed; this section of the compiler would therefore have to be rewritten for each target device.

The interesting points about this approach are that the source language is essentially unmodified, and that the compilation does not go through VHDL or Verilog. However, the full range of the source language is not used.

Xilinx have recently released their FORGE design language which is Java-based, but it is immature and there is little public information about its structure and reliability.

Composing hardware

An early compositional hardware language was Ruby [JS90]. Ruby was based on the idea that circuits are built from parts by a process of composition, which has mathematical properties similar to the composition of functions and relations. It was studied in the early years of FPGA use but fell out of use and study. However, its key ideas have been evident in more modern work.

A modern development of Ruby is the Lava project being undertaken by Xilinx. The project involves Mary Sheeran, one of Ruby's original researchers. Lava[CS00] is a prototype HDL, not supported by official Xilinx toolsets, but has been developed and is in use at Chalmers University in Sweden. It trades off the expressiveness of full VHDL or Verilog for compactness and simplicity of descriptions of common circuit layouts. Currently it is implemented by being embedded in the widely-used Haskell functional programming language. One proposed commercial use of Lava, cryptography, is described below.

occam

Møeller-Nielsen and Caprini proposed "occam on a chip" in [MNC95]; the universal system programming language was occam [Ltd84], some section of the software was selected to be implemented in hardware, and two communication channels were added to control handshaking between software and hardware. In this particular case the target hardware was a transputer-like chip, reducing the required amount of compilation of the occam program, but occam is a good starting language if the hardware is PLD-like

too. occam's bit-level variables, ease of expression of parallel computations and simple inter-process communication channels map well onto the architecture of PLDs.

The main problem with this approach is the opposite of that with a high-level language such as Java; programming the parallel hardware part is relatively easy, but the occam language has not proven suitable as a general-purpose programming language. The demise of the transputer after Inmos were absorbed by SGS Thomson meant that occam was no longer a practicable implementation language for the mainstream x86, ARM and PowerPC-based systems.

Oxford University's Hardware Compilation Group have taken a number of approaches to this problem. Their earlier approaches included Ruby (described above) and Handel [PS93], which was an occam-like innately parallel synchronous language. Handel was much more of a programming language than Ruby, allowing an elegant expression of the parallelism of a program, but required much more effort on place-and-route than Ruby's compositional model.

Handel-C

The company Celoxica (formerly Embedded Solutions Ltd.), spun off from the Hardware Compilation group, is focused around use of the Handel-C language and the associated DK Design Suite. The Handel-C language is described in [Cel02]. It extends ISO-C syntax in the following ways:

- variables specified in bit-width;
- macros for bit-manipulation; and
- explicit RAM/ROM hardware elements.

Its semantics, however, are closely related to those of occam (and hence, CSP). The language model includes:

- a timing model, where each assignment or delay statement takes exactly one time step to complete;
- signals and channels for inter-thread communication; and
- a deterministic parallelism model.

An example of the use of Handel-C for a real application appears in [Swe97]. Handel-C is interesting both as a syntactic extension of a widely-used medium-level language and as a rewriting of the (implicit) sequential semantics of one language into explicit a timed parallel semantics. Nevertheless it falls short of the ideal language for our safety-critical systems, principally because C is an inherently unsuitable language for the implementation of highly reliable systems and because Handel-C borrows so much from C that it includes many of C's defects.

C's failings are described by Romanski in [Rom96]. The author is a recognised expert at making systems conform to the RTCA/EUROCAE DO-178B civil aviation safety standard. He makes the key comment "The [C] language attempts to hide the underlying machine so that programs become portable between different machines.

Unfortunately, the target characteristics show through.” The lack of strong typing, substantial unspecified or implementation-dependent behaviour, and language constructs such as unbracketed single clauses and admissibility of assignment into conditions in C are viewed by Romanski as some of the chief deficiencies that make it unsuitable for inclusion in safety-critical systems, even if a “safe” subset is used. Additionally, if we wish to abstract away as much as possible of the details of the target hardware then the use of a low-level language such as C appears to be going in the wrong direction.

An example of the problems Handel-C faces is the `par` construct, allowing parallel execution of multiple statements. Strictly speaking, race conditions cannot arise between threads because of the deterministic timing model. If thread 1 writes to variable A and thread 2 reads from variable A, whenever the program is run thread 2 will always get the same value of A. However, changing the order of statements in thread 1 may change the value of A read by thread 2. This instance of “law of the unintended consequence” would be a significant worry in building a safety-critical sub-system in Handel-C. The language also allows the use of types without explicit bit width, permitting their actual width to be inferred at compile time. This can only lower the predictability of such programs.

Handel-C may well prove useful in lower-integrity system development, and its use of fine-grain parallelism is intelligent, but it cannot seriously be considered for critical systems.

Ada

The syntax of Ada is very similar to the syntax of VHDL, which leads to the natural question of whether it is feasible to map between the two underlying languages. This was initially addressed by Sheraga[She96], with more recent work by Ward and Audsley [WA01, WA02b, WA02a] making progress towards a viable compiler.

Ada was designed as a language suitable for programming safety-critical systems, and includes facilities for precise definition of type ranges and parallel programming (“tasking”) which are required for many embedded systems programs. Ward and Audsley describe the construction of the York Hardware Compiler for sequential Ada[WA01] and its extension to the Ravenscar subset of Ada’s tasking facilities[WA02b]. It should be noted that they choose to use the SPARK Ada subset due to the structural restrictions which it imposes on Ada, which improves analysability. The compilation produces a netlist implementation of the program which can be compiled directly to a target device.

The motivation for this compilation has been to improve worst-case execution time analysis; bounding execution time on a program executing on a real-time operating system is more difficult than for the same program executing alone on a PLD. However, no mention is made of bounding loop execution counts, which is fundamental to such calculation. Since SPARK Ada admits proof of selected program properties such as maintenance of loop invariants and strict monotonic decline of variants, this should have been exploited. Worst-case timing analysis of SPARK has been analysed in detail by Chapman[Cha94] but this work has not been referenced by Ward and Audsley.

There is little discussion of interfacing a PLD-compiled program to another program running in software, which is fundamental to making PLD programming effective for large systems. In [WA02c] the authors discuss practical improvements to the Ada

language to make it easier to interface to PLDs and improve fine-grain parallelism of Ada programs, but fail to exploit the known data-flow in SPARK programs for this purpose. The evidence of scalability in program size is very limited. Compilation of some small programs is demonstrated, but none incorporating subprogram calls or tasking constructs are shown. There is no discussion of optimising the compiler for time or space.

This approach is interesting, in that it shows that Ada compilation can be done in practice and that SPARK Ada and Ravenscar are useful subsets to adopt, but the approach not been shown to be effective and practical for programs similar to those used in actual development and has not exploited SPARK Ada's features to the full.

Esterel

The synchronous programming language Esterel[Ber00] was used by Hammarberg *et al* [HNT03] to implement a demonstration hydraulic fluid detection system on an FPGA. Esterel is a language for programming reactive systems; we contrast it with the SRPT process algebra in Section 4.1.10. It can be compiled to VHDL or Verilog, which is how the fluid detection system was produced. It is certainly suitable for programming reactive systems on PLDs, and has a formal (synchronous) semantics, but there is as yet no public information about its use programming PLDs for real safety-critical reactive systems.

Domain-specific languages

One high-level alternative to conventional programming languages is CoreFire, described in [McH02]. This is used to produce high performance applications to run on the Annapolis Wild FPGA boards. It uses a “sticks and bubbles” graphical interface to draw program data flow. The main drawback with this system is the tie to the Wild board, whereas anything that compiles to VHDL will normally target a much wider range of commodity hardware. However, this has not deterred engineers at the Naval Research Laboratory (NRL) in Washington D.C., who are developing FPGA solutions for electronic warfare using CoreFire. It demonstrates that in restricted application domains there is a role for high-level design methods.

A more recent proposal has been made in the domain of cryptography. Launchbury and Singh [LS03] propose the use of the declarative functional language Cryptol. This allows compact expression of common cryptographic transforms in a functional language syntax. It is currently supported by compilers targeting the C (imperative) and Haskell (declarative functional) languages. The authors propose a PLD-targeted tool chain, using the Lava language embedded within Haskell. This work is at proposal stage, and depends on immature tools, but its concepts appear to be sensible.

2.4.5 Low-level language implementation

Describing a PLD program in a high-level language may be inappropriate; indeed, for early PLDs it was not practicable because of the small size of the devices. Even with large modern FPGAs, certain programs may be better designed at the logic component level.

In [AB00] Abke and Barke describe CoMGen, a tool to render low-level component descriptions into look-up tables. The input descriptions are in Verilog macro and gate-level netlists. The generator is not tied to one FPGA; it has an interface to an external floorplanner for sizing components appropriately for the target device, and does its own place-and-route. Mapping the finished netlists to a form suitable for programming the target device is done externally.

This approach is of interest because it classifies the low-level compilation steps into general and target-specific classes. How low-level programming for a given application is actually done will depend on several factors. If multiple devices are to be used, reducing the device dependence by using a tool like CoMGen makes sense. If a single device is to be used, the decision will depend on the perceived quality and useability of the vendor's tools.

2.4.6 Pebble

The “Pebble” language described in [LM98] is a more abstract representation of VHDL. The language is based upon the definition, instantiation and coupling of logical “blocks” which perform simple tasks synchronously. A Pebble representation of a half adder, for instance, is:

```
BLOCK halfadder [s1,s2:Wire] [cout,sum:Wire]
BEGIN
    xor2 [s1,s2] [sum];
    and2 [s1,s2] [cout]
END;
```

This expresses the half-adder as a coupling of XOR and AND gates. The naming of the wires within the block relate input and output wires of blocks. Other Pebble constructs allow parametrisation of blocks by size, placement constraints and conditional compilation. The “primitive” blocks in Pebble are expressed as blocks with an empty body; these will be constructs which the target device can implement with a single cell.

Pebble appears to hold considerable promise as a target for higher-level languages. A compiler for Pebble into structural VHDL or a netlist for the “Rebecca” simulator has already been demonstrated and is also described in [LM98]. The structural VHDL produced can then be compiled into a specific device by the appropriate vendor tools.

Pebble is especially interesting in this context because it may be viewed as an abstract representation of a PLD program which may be directly reduced to a cell-level implementation. As an example take a carry-look ahead N -bit adder which can be composed recursively by half-sized CLA adders until the single-bit level is reached; at this point full adder blocks can be used to form the building blocks of the system. We express this in more detail in Section 5.3.

Once the PLD program has been reduced to wire-connected computational blocks, the unavoidable device-specific mapping occurs. In a Xilinx 6200-series device, for instance, a half adder can be built on one cell, but simpler devices with only one output per cell would require two or more of their cells to be configured and linked to produce the full adder functionality. Therefore Pebble is in some respects the lowest level device-independent step in a compilation. This is a strong indication that Pebble

should be considered as a target for high-level compilers. We develop this approach in Section 4.2 and apply it in Section 7.2.

2.4.7 Testing PLD programs

Testing is a vital part of the development cycle of any significant system. It has three main aims:

1. to verify that the program loaded was the program intended;
2. to locate errors in the system software during development; and
3. to provide a level of assurance that the completed product fulfils its requirements.

Aim 1 is normally achieved by readback, as described in Section 2.3.5.

Aim 2 is normally achieved by what is commonly called unit testing; assuming that the software is divided into modules, the elements of each module are tested according to their design. Problems may arise here when the software design is very detailed; there is a temptation to derive tests from the code, which nullifies many of the benefits of testing. Ideally, the author of the tests would be independent of the author of the software, and would not have access to the implementation details of the source code for which he or she was writing the tests.

Aim 3 is achieved by running on the completed product a series of tests derived from the requirements (often called *functional testing* or *integration testing*, run in sections during development to check that modules work together properly), and showing that each test result is correct. The level of assurance provided will depend on a number of factors:

- the number and range of tests provided;
- the rigour with which the tests are derived from the requirements; and
- the proof that the system components tested are those in the final product.

The latter point is not trivial. Without good configuration management in the project, it is difficult to prove the required proof.

How should we test PLDs? Since they are a mixture of software and hardware engineering, we should examine testing techniques from both fields. We must also consider testing methods particular to the peculiar design of PLDs.

Requirements testing

The first set of tests will be requirements-based, checking that a given set of inputs produce the desired set of outputs within a specified time. Generating test cases from requirements is a well-understood problem, and there is little more to say here. The key is to make requirements independent of implementation techniques where possible, to avoid unnecessary restriction of the solution space to software or programmable hardware.

If a high-level language has been used to specify the PLD program then it may be useful to write unit tests for the PLD based on that high-level representation; this acts as a check that the potentially complex compilation and optimisation of the FPGA netlist has worked correctly.

Hardware fault detection

The hardware aspects of the PLD require more thought. Renovell, in [Ren00], describes a scheme for testing the interconnect, logic cells and RAM cells in a symmetric SRAM-based FPGA. These tests consider cases such as open and short between interconnections, stuck-at cases for logic and RAM cells, transition faults, coupling faults and address decoder faults in RAM cells.

This testing is done by feeding in explicit test configurations into the FPGA before loading the actual system configuration. Hence, we have a reasonable likelihood of detecting faults inherent in the FPGA, but must also consider the possibility of spasmodic errors in the configuration data. Our testing here will be affected by the permanence of the FPGA configuration. An SRAM-based FPGA will have its configuration loaded at each power-on, and so will have a greater likelihood of configuration error than a Flash-based FPGA which may only be reprogrammed three or four times in its lifetime. When drawing up a test plan for a system incorporating FPGAs, these factors must be considered. The developers will have to choose whether to ignore configuration errors, detect and report them (possibly shutting down the system subsequently), or taking measures to mitigate their risk such as using redundant hardware, exploiting PLD program readback or using a voting scheme.

An example of a triple-redundant PLD program design scheme including error detection and periodic program re-loading is described by Lima *et al* [LCR03].

Timing errors

Timing issues are a significant consideration in designing ASICs, and serious computational effort is devoted towards simulating ASIC designs in order to catch timing issues. This simulation is not generally available for normal FPGA designs, but the problem of timing issues is still present. Krasniewski, in [Kra00], shows how delay faults can be detected in an FPGA. His approach is to modify the contents of look-up tables in such a way that the LUTs become much more vulnerable to path delay; random testing of the modified program is then carried out to attempt to detect such faults.

Crosstalk

“Crosstalk” is the phenomenon due to inter-wire capacitance whereby switching in one trace of the FPGA may change the voltage in another trace. The shrinking feature size of integrated circuits has made crosstalk an increasingly important consideration in place-and-route. Wilton, in [Wil01], describes a routing scheme which optimizes for delay in the presence of crosstalk, and which demonstrated a 7.1% improvement in routing delay over its parent routing scheme. This indicates how important the consideration of crosstalk can be for system performance. It is also another complexity in the design of routers, especially in safety-critical systems when all potential crosstalk effects must be eliminated.

2.4.8 Summary of programming PLDs

PLDs are generally programmed at the HDL level, in Verilog or VHDL. There is a move towards programming in subsets of C and Java, adapted to take advantage of the

PLD's parallelism. However, these programming languages appear to be inadequate for programming components of high-integrity systems. The occam fine-grained parallel model appears to be a useful base for language design. The use of Ada is promising but remains to be shown to be practical.

There are mechanisms for testing PLDs, but their reconfigurability means that a class of reconfiguration errors must be explicitly tested for, over and above the normal software and hardware tests.

The hardware nature of PLDs introduces extra potential faults, such as crosstalk and timing issues, which require trapping and testing over and above that used for conventional software.

Given these issues, we will now look at the suitability of PLDs for use in safety-critical systems. We will also examine the state of the practice for such use.

2.5 Safety-Critical PLDs

2.5.1 Research directions

Any new work on incorporating PLDs into safety-critical systems should represent an advance in concurrency research. Before we investigate this problem, we should bear in mind the conclusions of Cleaveland *et al* [CS⁺96] in their recommendations on concurrency research. They isolate the following relevant topics for which challenges exist:

Algorithmic support to develop methods which can cope with the state-space explosion problem inherent in concurrent system design and verification, perhaps by a decomposition and refinement process;

Tool support to make tools portable and scalable, and better integrated into the software engineering lifecycle;

Technology transfer to expose existing design and verification technology to real life industrial and defence applications, to improve the technologies and to encourage their uptake by example; and

Programming languages to design usable, safe and secure languages incorporating a well-understood concurrency model.

We will incorporate these aspects in our problem statement in Section 3.5.

In this section we look at the emerging UK Defence Standard 00-54 and the RTCA standard DO-254, relevant to PLDs in safety critical systems. We see how a system safety analysis should incorporate any programmable logic in the system, and discuss how we might improve a PLD program to increase safety and reliability. Finally we summarise the key needs for the system developers who build programmable logic into their systems.

2.5.2 Safety of PLDs

Placing a programmable logic device into a safety critical system should result in an immediate assessment of the impact of the device's behaviour on the rest of the system. This enables the system designers to establish whether the addition of the device has made the system less safe. A "white box" safety analysis procedure, such as described by Simpson and Ainsworth in [SA99], will trace the output data of the device through the system and determine whether it can contribute to any predetermined system hazard.

An example might be an FPGA built to compute a customised Fast Fourier Transform of some data. If this data is determined to be safety-critical, for example as an input to an aircraft's fly-by-wire control system, then the safety analysis must show that the data produced has an adequate probability of being correct. Suppose that a 10^{-9} chance of an aircraft being lost on a typical mission due to system failure was deemed acceptable. The onus would then be on the safety team to prove that the probability of dangerous data being generated by the FPGA on such a mission, multiplied by the probability of such data causing aircraft loss, was less than 10^{-9} .

The system designers typically face a dilemma; should they ensure that the device's functionality is limited to prevent it contributing to a hazard, or should they attempt to demonstrate its correctness? The former may require a major system redesign; the latter requires a solid formal basis from which to argue.

White box safety allows us to analyse the errors that might occur in the FFT computation. If we can show, for instance, that the expected results of an erroneous calculation are distributed evenly across the result space, and that a simple sanity-check can detect 90% of such errors, then we could perhaps formulate an argument that only a 10^{-8} probability of calculation error is required. However, things are seldom so simple!

Gibbons and Ames, in [GA99], describe the experience of using an FPGA as a key element in the circuitry of a pyrotechnic release for the NASA Wide Field Infrared Explorer (WIRE) satellite experiment. The telescope cover was prematurely opened, causing hydrogen venting from the spacecraft and consequent high torque rates, venting all the solid-hydrogen cryogen within hours and rendering the instrument unusable for its intended mission. The premature opening was due to undefined behaviour of the FPGA (an Actel 1020) during power-up that permitted a 14 millisecond power spike on the outputs. Spacecraft hardware testing did not detect this problem.

A PLD program could be proven to be completely correct against its specification. However, correctness cannot avoid failures triggered by phenomena which can occur even before the program starts its execution, such as in the case of the WIRE satellite. In general, proof of correctness only guarantees that the program will be able to address conditions explicitly considered in the formal specification; nothing else is guaranteed.

2.5.3 Safety standard: Defence Standard 00-54

The UK Defence Standard 00-54 [MoD99] specifies requirements for electronic hardware in military systems. It is considered to be appropriate if an electronic element of the system affects the system's safety. As with other UK Defence Standards [MoD97, MoD96] it is split into two parts; Requirements and Guidance. The techniques described in the document are to be used to analyse complex electronic designs for systematic failures; dealing with random failures is discussed in Defence Standard 00-42 [MoD94]. All of 00-54's recommended procedures are to take place under the umbrella of the safety management standard DefStan 00-56 [MoD96].

Relevant quotations

The standard's recommendations which are of particular interest to us are in sections 12.2.1, 13.4.1 and 13.4.4. To quote:

§12.2.1: A formally defined language which supports mathematically based reasoning and the proof of safety properties shall be used to specify a custom design.

§13.4.1: Safety requirements shall be incorporated explicitly into the Hardware Specification using a formal representation.

§13.4.4: Correspondence between the Hardware Specification and the design implementation shall be demonstrated by analytical means, subject to assumptions about physical properties of the implementation.

where “custom design” refers to the particular electronic component in question and in particular to a PLD’s program data.

Motivation

The standard’s guidances provide more information about the motivation behind the standard. To quote: “The principal concern which has caused this Interim Standard to be produced is that electronic hardware designs used in critical applications have been getting steadily more complicated [...] Therefore the focus of this Interim Standard is on analysis and proof to supplement test.” It also notes that widely used standard HDLs without formal semantics, such as VHDL and Verilog, present compliance problems if used as a design capture language. Examples given of suitable languages are Z and VDM.

Standards evolution

The standard is only interim, and its contents will almost certainly change when it is incorporated into Issue 3 of DefStan 00-56 in early 2004. Nevertheless, the concerns which it expresses about existing practices and its suggestions for process improvements are worth careful scrutiny. A language which supports formal reasoning about PLD behaviour is what is required for compliance with this standard.

There is an on-going program in the UK Ministry of Defence relating to the development of guidance for the design and procurement of systems conforming to the Advanced Avionics Architecture (AAvA) for military aircraft systems. There is a specific guide about the use of PLDs in such systems which has been released in preliminary form as [Hil03a]. This in turn is expected to inform the re-write of 00-54. This guide encourages the goal-oriented approach to generating safety evidence demonstrated in the rewrite of the SW01 regulatory impact assessment for the CAP 670 Air Traffic Safety requirements[Civ02]. It seems reasonable that most parts of Issue 3 of 00-56 will adopt this form.

The previously noted increase in PLD capacity and speed, enabling them to perform more complex and time-critical tasks, in turn increases the likelihood that they will be a critical component in a safety-critical system. Without a generally applicable method of reasoning about their correctness to the standard that SIL-4 requires, such a system is unlikely to gain regulatory approval.

2.5.4 Safety standard: RTCA DO-254

RTCA DO-254[RTC00] is the programmable hardware counterpart of RTCA DO-178B[RTC92]. It was approved by the FAA in 2003 for use in aviation systems development. The author of this thesis has had experience in applying it in practice to a hardware development, and therefore has a well-founded perspective on its practicality.

Like its software counterpart DO-178B, DO-254 defines a set of required integrity levels A to D, with Level A being the highest integrity. The emphasis in DO-254 is on providing a practical guide to the development process for the PLD program and associated documentation. The advice is normally generic for the integrity level, with Appendix B describing particular techniques that may be appropriate for high integrity systems which DO-254 defines as Levels A and B. It is not normally prescriptive, leaving

it to the developers to choose (and justify) the advanced analysis methods to use in high-integrity systems.

The key to producing high-integrity systems conforming to DO-254 is to do functional failure path analysis (FFPA) as described in Appendix B section 2.0 of DO-254 to identify system hazards, deduce where the system may cause them, and justify how in each case the hazard is mitigated. Arguments may include manual analysis of the HDL or netlist, formal analysis techniques, mitigation through features of the system architecture, and in-service experience. Interestingly, it makes practical recommendations on qualifying hardware compilers for high-integrity work.

Where Defence Standard 00-55 and RTCA DO-178B are distinctly different in content, with 00-55 emphasising rigour over DO-178B's extensive testing, Defence Standard 00-54 and DO-254 are more complementary. DO-254 provides practical advice without forfeiting the requirement of a rigorous approach where appropriate; 00-54 provides the detail of appropriate rigorous approaches.

2.5.5 PLD correctness

We have already noted the difference between safety and correctness. How should we go about demonstrating that a PLD's behaviour is correct?

There are two choices for a strategy here. The more common is "show that the implementation does what the requirements say." This tends to rely on model-checking with a theorem-proving tool. The second strategy is often initially harder: "develop the requirements into an implementation" which is known as *refinement*.

Model-checking

The essentially synchronous property of a PLD's circuits may help the model-checking problem. Pierre, in [Pie95] describes the use of the Boyer-Moore Theorem Prover to verify synchronous circuits. He uses a 4-bit binary-coded decimal (BCD) checker and an iterative integer factorial generator as examples. The verification process was automatic for the first example, but required several man hours for the second, more abstract example.

Here we see the key weakness of such an approach: model checking is hard, interactive, and usually will only be able to tell you *whether* your system is correct, not what is required to fix it. Tracing the cause of and correcting a failure is a separate process. In addition, if care is not taken then the size of the model can easily grow to the point where it is computationally infeasible to model-check it completely. Often it will be better to prove correct the critical subset of the PLD logic.

Model-checking has been used successfully in verification of specialised processors. Srivas and Miller describe in [SM95] the verification of the Rockwell AAMP5 microprocessor. The verification was carried out at instruction-set and register-transfer levels. This was possible even though the AAMP5 microprocessor was not designed for formal verification, illustrating the strength of model-checking as a retrospective technique. However, AAMP5 was not a general-purpose microprocessor and was not available directly for public use.

The use of model-checking to identify undesirable properties in complex commercial hardware has recently been demonstrated by Intel[Sch03] in their verification of the

Pentium 4 processor. Following a 3-4 fold increase of pre-silicon logic errors in each generation of the IA-32 architecture, Intel applied model checkers to verify the critical properties of non-floating point arithmetic of the Pentium 4 at the netlist RTL level. The floating-point arithmetic required the addition of a theorem prover to formally verify correctness, model-checking alone being impractical.

For a very small or very structured PLD program, manual inspection of the netlist may suffice. This must be judged on a case-by-case basis, and it may be necessary to use techniques such as fault injection to estimate the reliability of the inspection.

Refinement

As with the first strategy, rigidly defined requirements and an implementation language with properly defined semantics are necessary for the approach to be meaningful. Refinement of requirements to a PLD implementation is usually done in a series of small steps. Each step's induction from the previous one relies on the correct use of a set of predefined refinement rules. There is a gradual progression from the high level language of the requirements to a low-level language which may be implemented on the target device.

This second approach requires more “up-front” investment of time and effort. A working implementation may not appear until late in the development process as it is produced by the very last step of refinement. However, the correctness of the implementation is guaranteed, excepting the possibility of human error in the refinement steps. These refinement steps are normally amenable to individual verification by manual inspection. The main disadvantage of refinement compared to the model-checking is that a late change in requirements may require much of the refinement process to be repeated.

For a high-integrity or safety-critical system of substantial size, the above considerations suggest that the second strategy be the approach of preference. The main difficulties in using it will be in the choice of a suitable low-level language with well-defined semantics, and in the early and correct elicitation of requirements. For the latter task, there are well-established requirements engineering tools such as Cradle [Str98] and methods such as REVEAL[Vic98].

2.5.6 Verification

We have already covered the issue of how PLD programs are tested in Section 2.4.7. For high-integrity systems we must also consider verification of PLD programs.

Robinson and Lysaght [RL00] examined the problem peculiar to FPGAs of verifying dynamically reconfigurable logic. They extended the Dynamic Circuit Switching framework to track the status of dynamic tasks, and monitor these statuses to detect certain classes of error. However, this testing is dynamic and so acts more as a run-time self-test than as a method to exclude the possibility of error in the first place.

Bartzick *et al* [BHKW00] presented a design of FPGA which is intended to detect simple faults within itself and hence be fault-tolerant. The test of the FPGA is executed after programming, and occurs in 32 clock cycles so is not significant in terms of total program execution time. Each block has three normal cells plus a fourth “X” cell which takes over if any one of the cells is determined to be faulty. This approach is worth considering in designing an FPGA for use in high-integrity systems.

Sawitzki *et al* [SSSS00] described how they verified the data path of a microprocessor including a reconfigurable processing unit. This was done according to a specification in hardware description notation, describing the change in state of the processor for each instruction. They used the Stanford Validity Checker [BDL96] proof tool. The 16 hardwired instructions took an average of 80 minutes each to verify; the 9 reconfigurable instructions took over ten times that time each. However, they did not describe in detail why the reconfigurable instructions took so much longer to verify.

2.5.7 Self-testing

A technique in current use for PLDs is the use of self-testing and fault detection. Lima *et al* [LCR03] described a modification to the existing practice of triple-redundant circuits on FPGAs prone to disruption from charged particles. The use of delay in circuits, voting on outputs and regular re-programming of the FPGA (“scrubbing”). This allows 100% detection of single-event upsets in the FPGA, and approximately 90% elimination of the errors.

This is not a replacement for more formal techniques, but provides a useful brute-force method of reducing the impact of common problems.

2.5.8 Emulation of PLDs

During development of a system, it may be that the developers need to integrate their software modules with the programmable logic. Here we run up against the practicalities of system manufacture. It is unlikely that the system hardware will be built until relatively late in the development process. How then should developers do this integration?

One option is for developers to fit their PCs with a standard PLD development card which are available from the device manufacturers. These boards, such as the XS40 [XES99] from XESS Corporation, can fit in a standard PCI slot or parallel port on the PC, and provide an FPGA which can be accessed by software on the PC. This is an extra expense, but is unlikely to be significant in a large project. The difficulty is that the program for the PLD may not yet be written, or at least still be in a state of flux.

If a high-level language is being used to program the PLD with a program D , and can be compiled into the main software program P , this makes life easier. The initial integration testing can take place using D , then the program can be changed to access the PLD proper as it becomes available. This also has the advantage that, if a module functions properly with D but fails systematically when the actual device is used, this may indicate a failure in the hardware compilation process; the implementation (PLD) does not do what the specification (D) does.

For purposes of realism, there will need to be some form of wrapper $W()$ around D when it is compiled into P since interfacing with a PLD adds complexities which the wrapper must emulate, notably:

- the PLD runs in parallel with P , with no natural synchronisation;
- communication with the PLD must be done in a hardware-specific manner, possibly also in a compiler-specific manner; and

- communication must normally be assumed to be asynchronous.

The way that such a wrapper will be implemented will vary significantly according to the high-level language and PLD chosen. Ideally $W(P)$ will encapsulate the details and present an interface which is very similar to that for interfacing with the real PLD.

2.5.9 Implementation tools

The state of the art in languages and tools used in safety-critical systems tends to lag behind the leading edge of industry by several years. It is instructive to examine why, as the reasons have significant implications for the choice of techniques to program systems involving PLDs.

The foremost reason is reliability. A compiler, microprocessor or design tool is typically shipped with a number of errors, some known at shipping time or shortly thereafter (e.g. the infamous Pentium FDIV bug), but other more subtle problems may take months or even years to become evident. For this reason, implementors of safety-critical software tend to choose a compiler that has been stable for at least a year and then work around the known errors. Newer versions of the compiler may have these known errors fixed but there is no guarantee that new unknown errors have not been introduced.

This was particularly evident when the Ada 95 compilers started to be released; for a while safety-critical systems customers were still choosing the Ada 83 compiler because it was a known quantity, even though the Ada 95 language was far better in general functionality and had fixed long-standing problems of Ada 83 such as the inability to read output-only parameters in subprograms. The author is personally aware that Ada 83 compiler licenses were still being sold by vendors such as Rational, and Ada 83 programs being written from scratch, in early 2003.

As far as hardware is concerned, the usual choice for a complex IC in a safety critical system is a chip which is one or more years behind the state-of-the-art at the time of system design. This is because any design defects in the IC should have become evident by then, and related software tools (such as netlist compilers in the case of FPGAs) will have had a similar period of use to uncover errors. Also, since many safety-critical systems have a long development and production cycle, by the time of release the system's hardware may be several years behind the leading edge.

For these reasons, if we are looking to incorporate PLDs running a compiled language into a safety-critical system then we ought to choose a well-established compiler and a device which is not leading-edge in technology. As a consequence, devising an all-new language for programming our safety-critical PLD runs the risk that no developer will use it until someone else has tried to do so and has discovered most of the compiler and language errors; by this logic, no safety-critical developer would take the risk of being the first to use the technique in a real system.

2.5.10 Key directions

Taking the preceding data into consideration, we can summarise the following requirements for PLDs to be incorporated into a safety-critical system:

- a specification or design language to codify formally the system requirements;

- inclusion of the PLD program within the safety analysis of the system;
- a well-defined semantics of the target device to permit full or partial proof of correctness of the system;
- a formally defined refinement process for developing specifications to PLD implementations;
- a suitable high-integrity high-level language for implementation of the software component of the system;
- an appropriate generalised and adaptable testing process to test PLD programs in isolation and within the system; and
- a method for interfacing system software with either the PLD or a software emulation of it, as transparently as possible.

2.6 Conclusions

PLDs in their current state provide sufficient performance and size to perform substantive (if relatively simple) tasks. With their increasing complexity comes the likelihood that they will be incorporated into more and more safety-critical systems as critical components, yet to date there is no satisfactory or widely-used method of reasoning about their functional correctness at either the component or system level. Without this they are a point of failure waiting to fail, with potentially catastrophic consequences, and new safety-critical electronics standards such as DefStan 00-54 or RTCA DO-254 may prevent their incorporation as critical system components.

2.6.1 Weaknesses of current research

The main weaknesses of the existing research are:

- high-level PLD programming languages are not related to the requirements of DefStan 00-54[MoD99] and RTCA DO-254 [RTC00];
- the formal specification and analysis techniques used for synchronous parallel systems are not related to the development of practical and useful hardware-software systems under DefStan 00-54 and RTCA DO-254;
- the existing high-level programming languages which can be compiled into PLDs, with the exception of Ada, are not suitable for programming critical systems;
- the existing compilation techniques for Ada do not take full advantage of the SPARK Ada subset and have not demonstrated scalability to practical program sizes and designs;
- the existing compilation techniques for Ada do not address the development of an Ada program partly in software and partly in hardware; and
- there is a general deficiency in demonstrating techniques to be practical at the scale of a typical modern embedded control system.

This thesis must address as many of these needs as possible. In Section 8.2 we will re-visit these weaknesses to see which of them we have covered and to what degree.

2.6.2 Research needs

The greatest need in this field is for a generalised model of a PLD with a well-defined semantics, and a low-level device-independent language (with similarly rigid semantics) such as Pebble to act as a target for high-level languages. Without it, any attempt to reason formally about PLD correctness will depend too much upon the particular properties of the device under examination.

Such a model would permit research into high-level languages and techniques for safety-critical system implementation without the concern that the results of such research would be tied to a particular implementation, and provide a sound formal basis for proving safety properties of the PLD and the system in general.

In the next chapter we provide a detailed statement of the problem arising from the above information, and set out criteria for judging whether it has been solved.

Chapter 3

Statement of Problem

This chapter identifies the problem which this thesis aims to address and sets out criteria for deciding whether and how the problem has been solved.

The purpose of this chapter is to provide a direction for the rest of the thesis. We summarise the current state of research in the fields of programmable logic and safety critical systems, as detailed in the previous chapter, state the problem we intend to solve, and identify the areas in which this research will make advances. We then specify the advances we intend to make.

At the end of this thesis we will need to determine whether its contents have provided a substantial addition to knowledge in the use of programmable logic devices in safety-critical systems. To this end we lay down a series of targets for our research, and for each of them list criteria for deciding whether that target has been met.

Finally we list the targets addressed by each of the future thesis chapters.

3.1 Current State of The Art

The literature survey has established the following facts relevant to this thesis. The list below includes appropriate references in Chapter 2.

1. current commercial PLDs are of sufficient size and complexity to perform significant computational tasks useful to modern software-hardware systems (Section 2.3.10);
2. programmable logic has a place in the development of systems where software alone does not provide adequate computational power (Section 2.3.1);
3. programmable logic allows much faster development turn-around than use of ASICs which must be fabricated (Section 2.3.1);
4. programming PLDs with a high-level language is feasible, and a number of languages and tools for this purpose exist (Section 2.4);
5. PLDs are currently used in safety-critical systems (Section 2.5);
6. several national and international safety standards bodies have made specific recommendations about the development of safety-critical systems incorporating safety-related electronic hardware (Section 2.5.3);

7. when seeking to advance the field of formal methods, we should aim for reusable models and theories, combinations of mathematical theories to tackle hybrid safety-critical systems, and integration with the system development process (Section 2.2.3);
8. when seeking to advance the field of concurrent programming we should aim to provide algorithmic support, tool support, suitable programming languages and appropriate technology transfer (Section 2.5.1);
9. no existing development techniques for programmable logic software appear to satisfy these recommendations (Section 2.4, Section 2.5.9);
10. developing software for safety-critical systems is a problem which is well understood and supported by a range of tools and techniques (Section 2.1); and hence
11. the use of PLDs in safety-critical systems is an emerging problem in need of a solution.

3.2 Scope of Analysis

In the research work described previously we have taken an international view. For the remainder of this thesis we will focus on the standards applicable for defense-related equipment in the United Kingdom, as a domain with which the author is familiar and for which a well-defined set of standards exist.

Within this domain, we will in practice concentrate on avionics systems although the techniques will generally be applicable across the whole domain of defence systems; such systems tend to be embedded, real-time, safety-related and developed to similar (prescriptive) standards. Additionally they are procured by a single organisation, the Defence Procurement Agency (DPA), so will undergo a standardised process of acquisition and certification.

When we come across problems with existing PLD development practice, we shall first look (when sensible) for solutions from the field of software engineering.

3.3 Target Level of Criticality

Private discussion[Pri03] with one of the authors of Interim Defence Standard 00-54 and domain experts responsible for certification of systems to Defence Standards 00-54, 00-55 and 00-56 confirmed our conjecture (Section 2.5.9) that current technology and tools do not support the development of systems incorporating PLDs with SIL-3 or SIL-4 functionality. Indeed, there is debate about whether even SIL-2 functionality is feasible.

As a result, the development of tools and techniques to support development of PLDs with SIL-3 functionality matching the requirements of 00-54 will represent a clear advance in the current state of industrial practice.

3.4 Levels of Rigour

Before discussing the issues of rigorous development and proof, it will be useful to codify a common understanding of the levels of rigour we will use in this thesis.

ad-hoc a handwaving argument which may appeal to previous experience or statistics.

systematic use of analysis tools and/or a thorough testing strategy.

rigorous providing a specification in an unambiguous notation along with a sketch proof of satisfaction.

formal providing a specification in an unambiguous notation and a proof in a system with axioms and deduction rules.

These definitions are somewhat arbitrary but do admit some degree of classification and comparison between levels of rigour.

3.5 Statement

The problem we intend to solve is:

What methodology is suitable for developing a set of safety-critical system requirements into an implementation which executes partially in a conventional microprocessor and partly on a programmable logic device?

Such a methodology should be rigorous and formal enough to admit verification and validation to the standards demanded by Def Stan 00-54 and RTCA DO-254 (electronic hardware), Def Stan 00-55 (software) and Def Stan 00-56 (system safety) for SIL-3 and SIL-4 systems (RTCA DO-254 Level A and B).

The problem has the following characteristics:

- use of existing proven methods for producing a system design;
- partitioning of the design into hardware and software components;
- development of the software component using existing proven methods suitable for the integrity demanded;
- provision of a formal model to describe the semantics of a program executing on a PLD;
- provision of a process and tools to develop part of a system design into a program for a generic PLD;
- provision of a process and tools to develop a program for a generic PLD into a netlist or HDL suitable for execution on a specific device;
- identification of criteria for making general design or implementation decisions during development; and

- provision of suitable evidence of acceptable safety and correctness with respect to a specification for construction of a safety case for the system.

As noted above, we shall aim to use best practice from software engineering to solve these problems, where possible.

3.6 Target Aims

In the remainder of this chapter we list a number of *target aims*. Each target has a unique identifier, used in later chapters to cross-reference back to the target. In this way the reader can track whether the chapter is covering the issues that it is intended to cover.

These aims set out our vision for a safety-critical PLD development process; we will not necessarily meet all of them completely, but will strive to achieve this. Failing to meet some of the aims may reduce the level of integrity which we can claim for our process.

Following each target aim is a list of *criteria* which will be used in the covering chapter to judge whether the target has been met.

Each target aim will be augmented by *definitions* of terms to clarify its meaning, as necessary.

3.7 Research Programme

In the remainder of this thesis we aim to develop a process to produce a hardware / software safety-critical system incorporating a SIL-3 programmable logic component, satisfying the current UK Defence standards.

3.7.1 Identified deficiencies

As discussed in Section 2.5, the existing tools and techniques for PLD program development appear deficient in the following areas:

Rigor – there is no way of showing that a given program satisfies a given specification without exhaustive testing.

Ambiguity – the higher-level programming languages used, such as Handel-C, do not have a well-defined semantics; the developer depends on the compiler writer's interpretation of the language specification. The requirements and guidance in Defence Standards 00-54 and 00-55[MoD97, MoD99] repeatedly aim to remove ambiguity in requirements, design and implementation.

High level design – the benefits of programming in HDLs such as VHDL or Verilog are analogous to the benefits of writing software in assembly language. Compared to high-level languages, these languages remove ambiguity and allow much greater programmer control at the cost of increased development and maintenance time. Writing a program in EDIF is analogous to writing software in machine code.

Vulnerability – the later an error shows up in a development process, the more expensive it is in time and resources to fix, as discussed in Section 2.2.5 where we contrasted late system testing with the correctness-by-construction approach. Static analysis of a program aims to detect semantic errors missed by the syntactic checks of a compiler. No static analysis tools for programmable hardware languages are known, and in any case they require a rigorous language definition to be effective.

We aim to show an advance in all these areas. To ensure that this is the case, we introduce the following target areas for subsequent validation.

Target 1 *The process we define must be rigorous.*

Criterion 1.1 *there must be checkpoints where the system in development must be evaluated manually or by automatic tools, so that inadequate systems can be rejected.*

Criterion 1.2 *all transformation steps in the process must, in theory, be able to be shown to be mathematically sound.*

Definition: *a transformation step is where a specification or program is wholly or partially changed to be less abstract than before.*

Target 2 *The process must help the developer to write unambiguous programs.*

Definition: *an ambiguous program is one where different compilers, both conforming to the language specification, may produce object code programs that have observably different behaviour.*

Criterion 2.1 *ambiguous programs must be rejected by the compiler or rendered impossible by constricting the language definition.*

Target 3 *The process must allow the programs to have sections written in a low-level language for speed and flexibility, but not allow these sections to compromise overall program reliability.*

Criterion 3.1 *the developer must be able to mark out a section of the high-level language program and transform it to a low-level implementation.*

Criterion 3.2 *such an implementation must allow the developer to take advantage of aspects of programmable logic architecture abstracted away by the high-level language.*

Criterion 3.3 *it must be feasible to show that the compiled version of the original section is equivalent to the low-level implementation, using appropriate behavioural models for the two machines that execute the programs.*

Target 4 *The process must admit substantial static analysis to discover semantic program errors at or before compile time.*

Criterion 4.1 *each implementation language used should have a strict syntactic definition which is easily enforceable.*

Criterion 4.2 *each implementation language used should have a semantic definition to supplement the syntactic definition.*

Criterion 4.3 *each semantic definition should define an set of rules which can be machine-checked in polynomial time, to determine whether a given source program is semantically well-formed.*

3.7.2 Maintaining existing benefits

We must not throw away the existing benefits of incorporating programs into programmable logic

Target 5 *The program produced must be easy to test.*

Criterion 5.1 *the specifications for the program must be of a form suitable for producing a test plan.*

Criterion 5.2 *it should be feasible to instrument the compiled version of the original program so that the developer can observe relevant data flow within the program.*

Criterion 5.3 *there must be a working and verified software simulator for the compiled program.*

Criterion 5.4 *the test plan produced from the specification should be suitable for the production of test vectors for the simulator.*

Target 6 *The program must be able to be compiled onto a range of existing and anticipated PLDs.*

Criterion 6.1 *given a program which performs a non-trivial computation, it must be developed using the specified process into a form where it may be compiled and run using some existing programmable logic device and toolset.*

Criterion 6.2 *the compilation chain must target one of the VHDL[IEE91], Verilog[IEE95] or EDIF[Int00b] languages at some point.*

Target 7 *The process must reuse existing proven tools where feasible.*

Criterion 7.1 *at every point where the process requires a new tool, the process must justify why existing tools are inadequate and how the new tool overcomes those inadequacies.*

Criterion 7.2 *at each point where an existing tool is used, the process must show how the tool supports the programmable logic environment and the required system integrity level.*

3.8 Components

Given these target aims, we can already deduce much about the form of the process. For instance, we can begin to look inside at the components that will make up the process. These will include:

- a specification and proof system suitable for the programmable logic architecture;
- a set of refinement rules suitable and adequate for refining a specification to an entity in the proof system;

- a mapping process from a subset of entities in the proof system to a form acceptable as input to a compiler with an HDL program as target output;
- a high-level language suitable for writing relevant realistic programs as parts of a safety-critical system;
- an mapping process for subsections of these programs to equivalent programs in a chosen HDL; and
- a compiler for mapping programs from the HDL to actual PLDs.

Note that the last item is provided for each specific PLD by their manufacturer, so we can assume that this exists if the HDL is Verilog, VHDL, EDIF or a subset of these languages.

3.9 Process

Moreover, the following target aims determine the relationship between the above components, and the necessary characteristics that the process must have to make it suitable for safety-critical system development:

Target 8 *The process must guide the developer in the appropriate use of each component.*

Criterion 8.1 *for each component there should be clear guidelines about what forms of input are suitable and what form of output is required.*

Target 9 *The process should indicate what kinds of error may arise at each stage.*

Criterion 9.1 *for each process stage there should be guidance on the likely sources of error, their consequences, and an estimation of the probability that they will occur.*

Target 10 *The process should provide flexibility so that it may be used in situations not anticipated in its original design.*

Criterion 10.1 *it should indicate which steps in the process may be adapted to different needs.*

Target 11 *The process must admit justification to the project safety authority that the programs output by the process are of an adequate integrity level.*

Criterion 11.1 *it should cross-reference apposite sections of relevant safety standards.*

Criterion 11.2 *it must specify the maximum safety integrity level of software produced by the process.*

Criterion 11.3 *it should justify each process step against the relevant safety standard requirements.*

Criterion 11.4 *it should support suitable unit, functional and system testing at each development stage.*

3.10 Existing Standards

To be able to produce a system which can be certified as acceptably safe, we must also conform to current safety standards. The following criteria, specifically appropriate to our phases of the development process, are taken from DefStan 00-54. They must be satisfied for the process to be suitable from the safety point of view.

A requirement is “relevant” if it pertains to the development of software for safety-related electronic hardware (SREH) at SILs 3 and 4, since this indicates that it relates to best practice in producing high-integrity systems. From DefStan 00-54 (Requirements) we extract the following relevant requirements.

Choosing and testing the characteristics of the physical device and its vendor-supplied compiler is outside the scope of the process. So is external validation of the development process, except in so far as our process must provide evidence to support this validation.

For each requirement we indicate (in square brackets) where it is covered in the above targets and criteria. Note that some of them are covered by the work in this chapter. Where necessary we introduce new targets and criteria.

7.3.1 (d) The development of SREH shall include ...safety analysis of the SREH development process ...; [**Target 9, Target 11**]

8.2.2 The safety case shall justify the claimed safety integrity level of the SREH by means of:

- (b) evidence that the methods and processes used in hardware development are appropriate; [summarised in Section 3.1]
- (c) safety arguments justifying the safety integrity of the design of any custom items. [**Target 11**]

8.4.1 A safety analysis of the SREH development process shall be carried out to demonstrate how the development process will deliver SREH which meets the safety requirements. [**Target 11, Criterion 11.3**]

8.5.1 The safety arguments for the integrity of the design of a hardware item shall include both analytical arguments and arguments from test. [**Criterion 1.2, Criterion 3.3, Criterion 11.4, Target 5**]

Definition: *an analytical argument is an argument which is presented as a set of statements written in one or more formal notations. These statements are then related and justified by the application of deduction rules from a logic system defined over the formal notations.*

Target 12 [00-54 8.5.2] *The analytical arguments provided shall include:*

- (i) *any formal arguments used in validation to show that the formal specification complies with the safety requirements;*
- (ii) *any formal arguments that the functional design satisfies the formal specification;*

(iii) for non-functional properties with specified safety requirements, analysis of the achieved behaviour, e.g.: performance, timing etc.;

(iv) analysis of the effectiveness of fault mitigation, for example use of such techniques as diverse implementations.

Criterion 12.1 *The process shall use a formal specification language which is amenable to analysis to specify its input [parts (i),(ii)].*

Criterion 12.2 *At each stage of the process, there shall be a formal argument that the output of the stage refines the input of the stage [part (ii)].*

Criterion 12.3 *The specification language used shall be able to capture some non-functional system properties such as performance and timing [part (iii)].*

Criterion 12.4 *Each stage of process development shall indicate the forms of errors which it can mitigate [part (iv)].*

Target 13 [00-54 12.1.2] *The Design Plan shall define the life cycle that is to be followed in the development of the custom circuit, including a specification process, a development process and a verification process.*

Criterion 13.1 *there shall be an unambiguous and clear description of the system program development process;*

Criterion 13.2 *the description shall show clearly the relations between the process stages;*

Criterion 13.3 *the tools and techniques used at each stage shall be clearly described.*

12.2.1 A formally defined language which supports mathematically based reasoning and the proof of safety properties shall be used to specify a custom design, unless it is agreed with the MOD PM (Ministry of Defence project manager) that this is inappropriate. [**Criterion 12.1**]

12.2.2 The choice of specification language shall be justified in the safety programme plan. [**Criterion 12.1**]

12.2.3 Tools used to compile, analyse, animate and transform formal language shall be ...justified in the safety programme plan. [**Target 7**]

12.4.3 A simulation plan, with input vectors and expected output vectors shall be defined as part of the Design Plan. [**Target 5**]

12.7.2 Appropriate safeguards shall be put into place as a defence against identified hazards in the development process in such a way that the complete SREH development process achieves the required safety assurance. [**Target 9**]

13.1 The activities performed in custom circuit development shall include all of the following:

- (c) formal analysis of the design; [**Target 4**]
- (d) simulation and physical test. [**Target 5**]

Target 14 [00-54 13.3.1] *A Hardware Specification shall be produced which defines the SREH in terms of its behaviour and properties.*

Criterion 14.1 *there shall be a formal model of a generic programmable logic device which may be directly mapped onto a range of actual programmable logic devices;*

Criterion 14.2 *the formal model must incorporate a useful (though not necessarily complete) range of common components of a programmable logic device.*

Criterion 14.3 [00-54 13.4.1]: *Safety requirements [that have expression in functionality] shall be incorporated explicitly into the Hardware Specification using a formal representation.*

Note: there may be non-functional safety requirements, such as liveness, that cannot easily be incorporated into the Hardware Specification. These will have to be addressed at a higher level of design or assurance in the system.

13.4.2 The consistency and unambiguity of the Hardware Specification shall be verified using analytic methods. [**Criterion 14.1, Criterion 2.1, Criterion 1.2**]

13.4.3 The safety functions and safety properties of the Hardware Specification shall be shown to fulfil the safety requirements. [**Criterion 14.3, Criterion 1.2**]

13.4.4 Correspondence between the Hardware Specification and the design implementation shall be demonstrated by analytical means, subject to assumptions about physical properties of the implementation.
[**Criterion 1.2**]

13.4.5 Static analysis shall be used to demonstrate freedom from classes of error defined in the safety programme plan. [**Target 4**]

13.5.1 A representative set of simulation results shall be obtained at all levels of the design, illustrating that the SREH operates as expected, based on a white box understanding of the internal construction of the custom circuit. [**Criterion 5.3, Criterion 5.4**]

3.11 General Questions

As well as these targets and satisfaction criteria, there are more general questions which should be asked to help gauge whether the process is sufficiently reliable and practical. These questions have arisen from practical experience in software engineering for real safety-critical systems. The questions will be answered, where possible, in Chapter 8.

3.11.1 Reliability

1. How many distinct stages are there in the methodology?
2. What is the probability and effect of introducing an error at each stage?
3. What do 1 and 2 imply for the reliability of the system as a whole?
4. What classes of error are specifically checked for in the development process?

3.11.2 Practicality

1. Is there adequate tool support for the developers of the target systems?
2. What level of technical expertise, and how much time, is required for each development stage?
3. Given appropriate same-generation hardware, does the generic PLD implementation produced have significant performance advantages over an all-software implementation?
4. How well does the process allow late changes in requirements to be incorporated into the system?

3.12 Overall Process

In the following chapters we shall describe the components of the development process shown in Figure 3.1. This process is taken to start when safety engineering activities have identified the system hazards, accidents and resulting safety requirements. The process end is when the high-level specifications have been developed into a mix of SPARK Ada and Pebble code that together implement the system and demonstrably satisfy the safety requirements.

3.13 Future Chapters

Chapter 4 aims to adapt existing technologies to match our requirements. We will demonstrate a proof system based on Synchronous Receptive Process Theory, show that processes from this system may be transformed into implementations in the Pebble language, and demonstrate that SPARK Ada has suitable features for SPARK programs to be compilable into programmable logic devices. We will describe the development process as a whole. The targets addressed in Chapter 4 are 1, 2, 3, 4, 6, 7, 10, 11, 12, 14.

Chapter 5 provides a rigorous proof system for refining specifications into equivalent SRPT processes. Taking the Chapter 4 work mapping between SRPT and Pebble, this allows us to refine a specification into a full implementation on a commercial PLD. The targets addressed in Chapter 5 are 1, 2, 5, 6, 9, 10, 12.

Chapter 6 builds on the Chapter 4 SPARK Ada work to show how SPARK programs can be run on an interpreter running on a PLD. The targets addressed in Chapter 6 are 1, 2, 3, 4, 10.

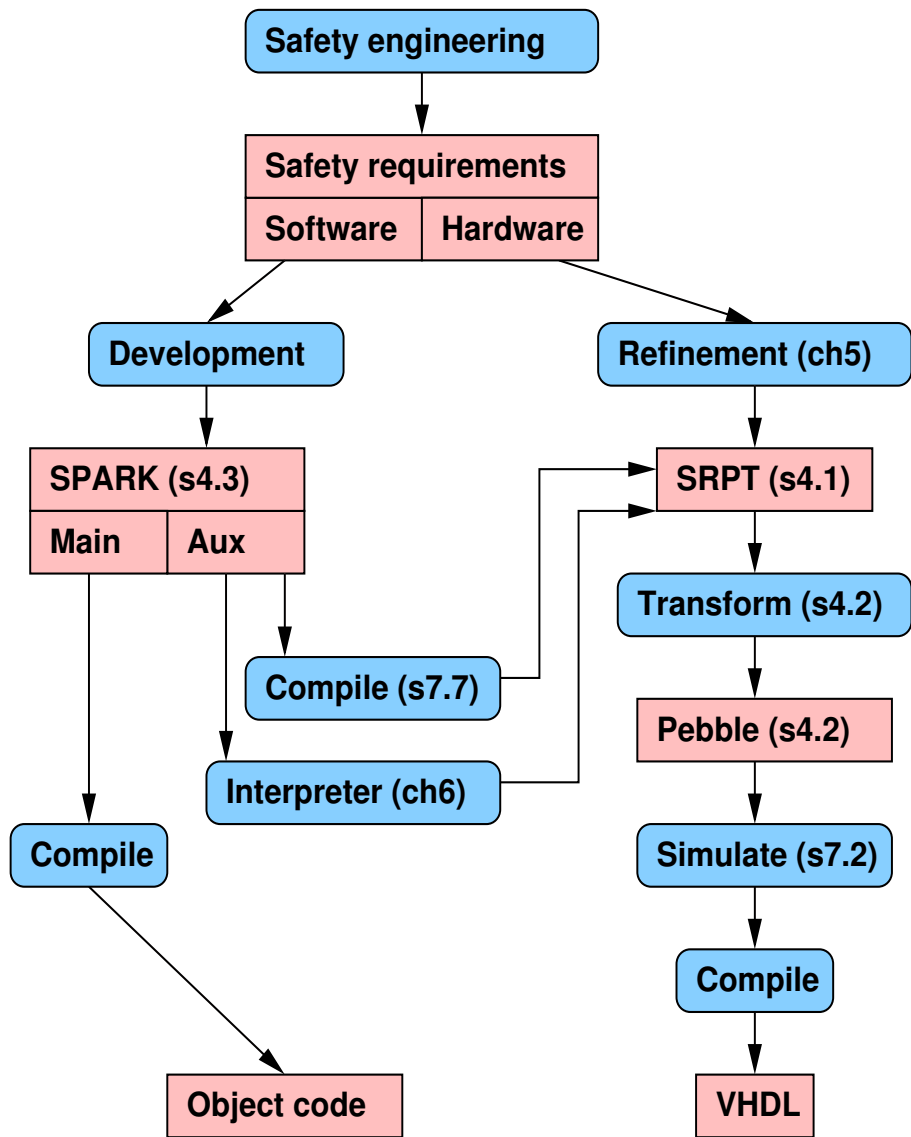


Figure 3.1: Development process

Chapter 7 is a practical validation of the process, building a safety-critical system by following the process. The targets addressed in Chapter 7 are 1, 2, 3, 4, 5, 6, 7, 9, 11, 12, 13.

Finally, Chapter 8 evaluates the material in Chapters 4 to 7 against the aims given in this chapter. Chapter 8 addresses Target 8 and also addresses the more general questions in Section 3.11 above.

Chapter 4

Development technologies

This chapter describes the technologies used in the development process which we detail later in this thesis.

Section 4.1 introduces Synchronous Receptive Process Theory (SRPT), a process algebra which we will use to model the execution of a program within a PLD. We demonstrate how to specify requirements about SRPT processes and how to prove that an SRPT process satisfies a requirement.

Section 4.2 introduces Pebble, a simple programming language for synchronous PLDs which can be compiled into VHDL or directly into netlists for particular PLDs. We establish a formal connection between Pebble and SRPT, and provide a formal definition of how we expect Pebble programs to execute. Pebble abstracts away target device details so we will imagine Pebble as executing on a “generic” PLD.

Finally, Section 4.3 examines SPARK Ada, an imperative programming language intended for programming safety-critical systems. We describe those characteristics relevant to our work and lay the groundwork for compiling a subset of a SPARK programs into an SRPT system description executing on our generic PLD model.

4.1 Synchronous Receptive Process Theory

Synchronous Receptive Process Theory (SRPT) is a process algebra described by Barnes in [Bar93].

4.1.1 Introduction

The process algebra CSP [Hoa85] has been used successfully to demonstrate partial correctness of protocols and industrial parallel systems. Supporting tools such as FDR [For97] allow semi-automatic analysis of relatively large and complex parallel systems, proving them free from deadlock and livelock. However, CSP is not suitable for describing all aspects of PLDs. In particular its *asynchronous* nature requires that traces consist of a sequence of single event names (“interleaving concurrency”) meaning that distinct events cannot happen at the same time, and it is not *receptive* since CSP processes can refuse events, which complicates the modelling of digital logic. As noted in Section 2.3.11, Timed CSP is a development of CSP which is adequate for digital logic modelling but is a more complex system than we need for a single clock system.

SRPT in a nutshell

Synchronous Receptive Process Theory (SRPT) was developed by Barnes by combining Receptive Process Theory [Jos92] and CSP [Hoa85]. It is a *process algebra* i.e., an algebraic theory to formalize the notion of concurrent computation. As a process algebra it consists of a syntax for describing process terms and their composition, and a notion of behaviour. In contrast to CSP it is *synchronous*: like SCCS [Mil83], events happen only at integer time intervals, and it is *receptive*: SRPT processes may not refuse events if their environment offers them. Barnes[Bar93] provides a rigorous definition of SRPT, and demonstrates its applicability to clocked digital circuits.

The use of algebra to specify digital logic circuits is not new. Such specification has been done using a wide range of formalisms, for instance CSP [Hoa85] and its timed and synchronous variants. We discuss alternatives to SRPT in Section 4.1.10.

Aims and objectives

In this section we describe a deterministic subset of SRPT, show how it may be used to describe digital circuits, and demonstrate specification and partial proof of deterministic SRPT processes. We also lay the foundations for the SRPT refinement system in Chapter 5.

The definition work which follows aims to establish that deterministic SRPT is a valid closed subset of SRPT. We require a solid formal basis from which to construct our proof and refinement systems. We build on the work done by Barnes[Bar93] in defining and exploring SRPT, rather than re-creating it.

4.1.2 Deterministic SRPT

An SRPT system description has an alphabet Σ of events. There are a countable (if not necessarily finite) number of processes $\mathcal{P} = \{P_k\}$, for which each process $P \in \mathcal{P}$ has an input alphabet $\iota P \subseteq \Sigma$ and output alphabet $oP \subseteq \Sigma$. For each P , ιP and oP

must be disjoint, finite, and their union must be non-empty. ιP consists of the events to which process P may react, and oP the events which the process controls. There is a set Var of process variables, each of which will range over \mathcal{P} .

Processes in our deterministic subset are defined using the following grammar:

$P ::=$	x	process variable
	$[!O ?X \rightarrow P_X]$	output prefix
	$P \parallel P$	parallel composition
	$P \setminus O$	hiding
	$P[S]$	renaming

The grammar we use differs from that of Barnes in the following ways:

- it omits the non-deterministic constructs because our interest is solely in deterministic circuits for the purposes of this study; and
- we incorporate the recursion operator defined for full SRPT into the output prefix model, hence restricting recursion in our SRPT subset to guarded recursion. This means that the recursive definition always leads to a single process. The details of recursion in full SRPT are in Barnes[Bar93] §5.1.2.

In the above definition, O denotes a subset of the output alphabet oP , X denotes a subset of the input alphabet ιP and S is an automorphism over Σ (a bijection $\Sigma \rightarrow \Sigma$). Each $P \in \mathcal{P}$ is then a function

$$P : \mathbb{P}(\iota P) \times \mathbb{P}(oP) \rightarrow \mathcal{P}$$

where each function $Q \in \mathbf{ran} P$ is such that $\iota Q = \iota P$, $oQ \subseteq oP$.

$P_X : oP \rightarrow \mathcal{P}$ represents a curried process such that $P_X(Y) = P(X, Y)$.

The operators of SRPT are defined in [Bar93] pp. 76–80; intuitively, in comparison with CSP for instance, only the output prefix will appear unfamiliar. Since we will use the output prefix form extensively, it is worth providing an informal definition here. $[!O ?X \rightarrow P_X]$ specifies a process that will immediately output all events in O and receive from the environment some set of events $X \subseteq \iota P$ in its input alphabet. From the next timestep onwards it behaves as process P_X , i.e. P parametrised by X as explained above.

The definition of a process P is in terms of a reaction to input events (a subset of ιP). Unlike in CSP, an SRPT process cannot refuse an event which is in ιP ; it simply observes such events happening. What it can do is react to those events by signalling events in its output alphabet. The nature of SRPT means that processes may receive and output any number of events at once.

Barnes [Bar93] defines a set of axioms and derives laws for algebraic combination of terms from this grammar. For example:

$$\mathbf{a-10} : \quad [!B ?X \rightarrow P_X] \parallel [!C ?Y \rightarrow Q_Y] \equiv [!(B \cup C) ?Z \rightarrow P_{(Z \cup C) \cap \iota P} \parallel Q_{(Z \cup B) \cap \iota Q}]$$

This states that when combining two output-prefixed processes, we initially see the combined output of both processes, which we would naturally expect. From then on

Time	0	1	2	3	4	5	6	...
Input	a, b	$a, -$	$-, -$	a, b	a, b	$-, b$	a, b	...
Result	$-$	c	$-$	$-$	c	c	$-$	$c \dots$

Table 4.1: Example run for *AND*

P 's behaviour may additionally be affected if its input alphabet includes one or more events from the output alphabet of Q , and vice versa.

As an example of an SRPT system definition, in the following subsection we will describe a 1-cycle 2-input AND gate with the SRPT algebra.

4.1.3 Example – AND Gate

An AND gate has no control over its two inputs; it exerts control over its output according to the values of the inputs in the previous timestep. It cannot provide an output at time t which relates to inputs received at time t ; there is always a delay before the reaction is visible.

We define the main process in the SRPT system as

$$\begin{aligned}
\iota AND &= \{a, b\} \\
o AND &= \{c\} \\
AND(R) &= [!R ?X \rightarrow \mathbf{if} \{a, b\} \subseteq X \mathbf{then} AND(\{c\}) \mathbf{else} AND(\emptyset)] \\
AND &= AND(\emptyset)
\end{aligned}$$

Two points on notation are worth making. We may use functional application instead of subscripting for processes, which has advantages of clarity when the subscript text is complex. The **if ... then ... else** construct is valid because it defines a process map parameterised by events in the input alphabet, and each process in the range of that map has identical input and output alphabets to $AND(R)$.

In the above definition of AND , the occurrence of an event at one tick of the clock corresponds to the presence of a high value on the wire named by that event at that time. So if a is present in one element of a trace of AND then this means that the gate has received a high voltage on the a input wire at that point. If a is absent, this is interpreted as a low voltage on a . a and b are taken to be the two input wires, and c the single output wire. This will be the convention used throughout this thesis.

The process definition states that AND initially makes no output, then subsequently it will raise the c event at time $t + 1$ if and only if both a and b were present at time t . The $!R ?X$ part of the process description means “output all events in R and let X be the set of inputs which we have received in this timestep.”

Note that our process definition parametrises process AND to tell it what to output. This effectively encodes state within the process, though in this example state at time t never affects the process after time $t + 1$.

An example “run” for $AND(\emptyset)$ could be as shown in Table 4.1. Note that the environment controls when a and b appear; only the c event is controlled by the process.

4.1.4 Composition

Composition is a key tool to allow us to build complex systems out of more simple processes. SRPT allows us to compose processes to form larger ones, in serial (for sequential composition) or in parallel.

Parallel composition is done with the \parallel operator, but the most useful composition is normally serial since this allows us to break down a calculation into multiple stages. Serial composition is effected by renaming process alphabets so that output events in one process are input events in another process.

The different forms of composition are analysed by Hall[Hal96b] where the structures imposed by CCS and CSP on a parallel system (synchronisation trees and traces, respectively) are compared against an algebra for high-level Petri Nets.

4.1.5 Denotational semantics

To be able to make rigorous analytical arguments about what does or does not happen in an SRPT system, to the level required by standards such as Defence Standard 00-54 [MoD99] for the most safety-critical of systems we must consider SRPT's meaning for the behaviour of a system. Barnes defines the meaning of a system in SRPT in terms of process traces.

In a given system, each process P with input alphabet I and output alphabet O has a semantics defined in terms of its set of traces $RT_{I,O}$:

$$RT_{I,O} = (\mathbb{P}(I \cup O))^*$$

Unlike CSP, but similar to Discrete Time CSP[Jef91], SRPT defines a trace $t \in RT_{I,O}$ as a sequence of sets of events: $t : \mathbf{seq} \mathbb{P}(I \cup O)$. Each element of the sequence corresponds to a (non-negative integer) time value of the global clock, and gives the events in I and O for that process which happen at that time. Discrete Time CSP uses bags rather than sets.

Trace axioms

As is usual in process algebras with traces, traces are prefix-closed and the empty zero-length trace $\langle \rangle$ is valid. There is an additional constraint that the environment can offer *any* subset of input events at a given step, and the output at that step must be independent of the input of that step; this means that processes cannot react instantly to an input, corresponding to the delay in a logic gate output reacting to its inputs, and must be able to “handle” any combination of possible inputs. Formally, for a set of traces T :

$$s \frown \langle X \rangle \in T \wedge Y \subseteq I \Rightarrow s \frown \langle (X \cap O) \cup Y \rangle \in T$$

i.e., suppose $T \subseteq RT_{I,O}$, then T represents the trace set of a process with input I and output O . Here s and r are traces, $\langle \rangle$ and \rangle delimit a trace element and \frown is the trace concatenator.

Semantic function

The full derivation of semantic functions is given in Barnes[Bar93] §5.4; again, we summarise.

RM is the set of all triples (I, O, T) , where T is a trace set satisfying the conditions above and I and O are input and output alphabets satisfying the restrictions given in Section 4.1.2. The binding function $BIND_R$ maps from a set Var of process variables to RM . This is what the user is effectively defining when he or she writes the process definitions and decides on the names of the process variables.

The semantic function \mathcal{M}_R maps each process term to an element of RM . The associated function \mathcal{T}_R maps process terms to RM_T (the set of all sets of traces for processes), and ι, o map process terms to their input and output alphabets $\subseteq \mathbb{P}\Sigma$. Hence where σ represents an element of $BIND_R$ we have:

$$\mathcal{M}_R[[P]]\sigma = (\iota[[P]]\sigma, o[[P]]\sigma, \mathcal{T}_R[[P]]\sigma)$$

This can be read as “Given the user definition σ of P and associated SRPT processes, \mathcal{M}_R maps P onto its input alphabet, output alphabet, and the set of all traces valid for it.”

All that then remains is to define ι, o and \mathcal{T}_R for each of the process terms. The interested reader is referred to Barnes[Bar93] §5.4 (Definitions 5.3 and 5.4) for details of these definitions; here we present the definitions for the output prefix construct as an example.

If $B \subseteq o[[P_\Omega]]\sigma$ and

$$\forall C \subseteq \iota[[P_\Omega]]\sigma \cdot \iota[[P_C]]\sigma = \iota[[P_\Omega]]\sigma \wedge o[[P_C]]\sigma = o[[P_\Omega]]\sigma$$

then

$$\begin{aligned} \iota[[!B ?X \rightarrow P_X]]\sigma &\equiv \iota[[P_\Omega]]\sigma \\ o[[!B ?X \rightarrow P_X]]\sigma &\equiv o[[P_\Omega]]\sigma \end{aligned}$$

and in all cases the traces are defined:

$$\mathcal{T}_R[[!B ?X \rightarrow P_X]]\sigma \equiv \{\langle \rangle\} \cup \{\langle B \cup Y \rangle \frown s \mid Y \subseteq I \wedge s \in \mathcal{T}_R[[P_Y]]\sigma\}$$

where $I = \iota[[!B ?X \rightarrow P_X]]\sigma$.

Equivalence and congruence

P and Q are *observationally congruent*, according to Milner[Mil89], if $F(P)$ is observationally equivalent to $F(Q)$ for any environment F . In SRPT, this environment corresponds to a sequence of sets of process input events.

It will not be unexpected to the reader familiar with process algebraic theory that, because of the removal of non-deterministic process constructs, process equivalence (defined as the processes possessing identical trace sets) and process congruence (as defined above) are coincident. In particular, suppose P and Q are equivalent. It is easy to see that they must have the same input alphabets, since by the trace well-formedness rules any event in the input alphabet may be offered at any step. By assumption, they have the same traces.

To show that congruence is implied by trace equivalence we must show that $F(P)$ and $F(Q)$ are observationally equivalent. To see this we appeal to the absence of non-determinism; an environment offering $F = \langle F_1, F_2, \dots \rangle$ to P will, because of determinism, elicit a single behaviour $S_P = \langle s_1, s_2, \dots \rangle$ say, where $s_i \cap \iota P = F_i$. As P

and Q share traces, Q must also have this behaviour, and since Q is deterministic S_P must also be Q 's response to this environmental offering. Hence $F(P)$ and $F(Q)$ are the same for each environmental offering F . The argument is symmetric in P and Q . \square

Note that because P and Q can differ on their output alphabets (therefore being different processes according to the SRPT definition of equality) process equality implies trace equivalence but is not implied by it. P and Q may be equivalent, and hence observationally congruent, but not equal.

Establishing this relationship between congruence and equality demonstrates the amenability of deterministic SRPT to algebraic proofs about its properties, and marks a clear algebraic difference between deterministic and non-deterministic SRPT.

4.1.6 Specification and proof

Barnes's rigorous definition of SRPT, and our definition of the deterministic subset of SRPT, will allow us to reason formally about SRPT processes. In designing a system in which we wish to prove partial correctness (i.e., may not terminate, but correct if it does terminate) we need to be able to make concise and precise specifications of the legal and illegal actions of the system, and prove their presence or absence formally without too much effort.

In this subsection we will prove a useful property of a class of SRPT processes relevant to modelling an FPGA. To provide specifications for the actions of a process, we make statements about its traces. Given A , B and Z pairwise disjoint subsets of Σ where A and B are of size n and Z has an arbitrary finite size, we will define a system of SRPT processes to model an FPGA cell with $2n$ inputs $CELL_{n,f}$ pointwise computing a logic function $f : \mathbb{P}A \times \mathbb{P}B \rightarrow \mathbb{P}Z$. We define this system as follows:

$$\begin{aligned} \iota CELL_{n,f} &= A \cup B = I \\ o CELL_{n,f} &= Z \\ CELL_{n,f}(R) &= [!R ?X \rightarrow CELL_{n,f}(f(X \cap A, X \cap B))] \\ CELL_{n,f} &= CELL_{n,f}(\emptyset) \end{aligned}$$

We form the specification for $CELL_{n,f}$ by constructing a set comprehension with a boolean satisfaction expression quantified over all elements of each valid trace. This is an approach demonstrated in Hoare[Ho85], sections 1.8 and 1.9. We use $t[i]$ to refer to the i th element of the trace t , indexing starting at 0. The specification must be true for any trace of the process $CELL_{n,f}$, and is as follows:

$$\mathcal{T}_{\mathcal{R}}[[CELL_{n,f}]]\sigma = \{t \mid (\#t > 0 \Rightarrow t[0] \cap Z = \emptyset) \wedge \forall 1 \leq i < \#t \cdot t[i] \cap Z = f(t[i-1] \cap A, t[i-1] \cap B)\} \quad (4.1)$$

This can be read as “if the trace at step $i - 1$ has input events C from set A and D from set B then the output events in the trace at step i must represent the result of $f(C, D)$ ”. We constrain the initial output set to make the satisfying process unique. The σ in the specification represents the translation of the abstract event sets A, B, Z in the process definition of $CELL_{n,f}$ into real events from the system event set Σ .

We abbreviate this specification on a trace t to $S(t)$. To show that $\forall t \in \mathcal{T}_{\mathcal{R}}[[CELL_{n,f}]]\sigma \cdot S(t)$, we first show that the process is deterministic in its initial value:

Lemma 1

$$\forall E \subseteq Z \cdot \mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}(E)]\!] \sigma = \{t \mid \#t > 0 \Rightarrow t[0] \cap Z = E\}$$

[i.e., the output set E passed as a parameter to $CELL_{n,f}$ will always appear as the first output.]

For a process $[!E ?X \rightarrow P_X]$ with input alphabet I , Barnes's definition for output prefix is:

$$\begin{aligned} \mathcal{T}_{\mathcal{R}}[\![!E ?X \rightarrow P_X]\!] \sigma = \\ \{\langle \rangle\} \cup \{\langle E \cup X \rangle \frown s \mid X \subseteq I \wedge s \in \mathcal{T}_{\mathcal{R}}[\![P_X]\!] \sigma\} \end{aligned}$$

To prove Lemma 1:

$$\begin{aligned} \mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}(E)]\!] \sigma &= \mathcal{T}_{\mathcal{R}}[\![!E ?X \rightarrow CELL_{n,f}]\!] \sigma = \\ &\{\langle \rangle\} \cup \{\langle E \cup X \rangle \frown s \mid X \subseteq I \wedge \\ &\quad s \in \mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}(f(X \cap A, X \cap B))]\!] \sigma\} \end{aligned} \quad (4.2)$$

by definition. As $X \subseteq I$, X and Z are disjoint. Therefore the output events in the first element of any non-null trace must be exactly E . \square

We now show that the correct values continue to be output by the process as the trace grows:

Lemma 2

$$\begin{aligned} \forall E \subseteq Z \cdot \\ t \in \mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}(E)]\!] \sigma &\Rightarrow \\ t = \langle (X \subseteq I) \cup E \rangle \frown r &\Rightarrow r \in \mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}(f(X \cap A, X \cap B))]\!] \sigma \end{aligned}$$

[i.e., after the first step of $CELL_{n,f}(E)$, the subsequent trace r is the trace of $CELL_{n,f}(Y)$ for some Y as a function of the environment's input.]

This follows directly from Equation 4.2 and the definition of $CELL_{n,f}$.

Combining Lemmas 1 and 2 gives us the proof that all the traces of $CELL_{n,f}$ satisfy the two parts of the specification S in Equation 4.1:

$$t[0] \cap Z = \emptyset$$

This comes from Lemma 1: $CELL_{n,f}$ is defined to be $CELL_{n,f}(\emptyset)$ so $E = \emptyset$.

$$\forall 1 \leq i < \#t \cdot t[i] \cap Z = f(t[i-1] \cap A, t[i-1] \cap B)$$

This comes from Lemmas 1 and 2: let $X \subseteq (A \cup B)$ be the set of input events at time $t-1$, let $E = f(X \cap A, X \cap B)$, then Lemma 2 says that the trace of t from time i onwards is $r = \mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}(E)]\!] \sigma$. Lemma 1 says that $r[0] \cap Z = E$. Since $r[j] = t[i+j]$, the result is proven. \square

This result is applicable to all stateless one-cycle cells (i.e. those cells where output at time $t+1$ is solely dependent on input at time t), and is a useful foundation for proof at a higher level of abstraction. We give an example of this in the following section.

The exact method of proof is not particularly important; what it does show is that such proof is feasible and details one way that it can be done.

4.1.7 Safety monitor example

System definition

For an example, we take a military aircraft stores management system (SMS) which is designed to control the arming and release of ordnance from designated “hardpoints” (pylons containing hydraulic and electrical release equipment) on the aircraft. This kind of system is clearly safety-critical because malfunction could easily lead to premature release or detonation of ordnance; see below for an expansion of this argument.

An SMS will often contain several PLDs implementing simple (though perhaps critical) functionality. As noted in Section 2.3.1, such a low-volume production benefits from the low fabrication cost and quick turnaround of a PLD as opposed to an ASIC.

System hazards

The hazards of an aircraft stores management system include:

1. release and subsequent detonation of a store while the aircraft is on the ground;
2. release of a store while the aircraft is in an inappropriate attitude (e.g. turning and descending in such a direction that the released store may impact the aircraft); and
3. arming and release of a store over “friendly” territory such as a town near the aircraft’s home airfield.

Other hazards (such as arming and fusing a store which may detonate on the wing) are normally mitigated by the store rather than the SMS. An air-launched torpedo, for instance, might only arm itself on contact with salt water.

System safety

The safety features of this system will include:

- a hardware watchdog timer which must be reset every 25ms or the watchdog will shut down the system (to stop or restart a hung system); and
- the use of *keywords* to command dangerous actions.

A *keyword* is a unique data value which enables a dangerous action. The use of a keyword is an application of probability theory; a keyword is usually 4-16 bytes long and chosen such that no value matching the keyword is normally present in the processor’s address space. The chance of the keyword arising accidentally is unlikely; the chance of a single bit in a control word being set erroneously is orders of magnitude more likely.

The functionality required to implement these operations is well within the ability of a PLD (if we ignore the aforementioned concerns about PLD safety) since they are simple in design, and benefit from being outside the direct address space of the microprocessor once they are implemented in the PLD.

To implement these operations within a safety-critical system we must specify them and show that the specifications are satisfied, and it is this that we illustrate with an example here.

Watchdog timer specification

The Watchdog Timer has a single input, which is toggled to reset the timer, and a single output which is typically used to raise a high-priority interrupt and trigger a system shutdown. We will define an SRPT process $WATCH_k$ with the following behaviour.

We assume that there is a single input w to the watchdog timer and a single output d . We will produce a timer specification with parametrised delay since the PLD timer delay will be expressed in PLD clock ticks, and we may not know the actual PLD clock frequency until later in the development cycle.

The specification of the watchdog $WATCH_k$ which shuts down after $k + 1$ steps without an input toggle, for each trace t , is as follows. First, we define the events we are reasoning about:

$$\begin{aligned} \iota WATCH_k &= \{w\} \\ o WATCH_k &= \{d\} \end{aligned}$$

where w and d represent high voltages on the corresponding input and output wires, following the convention described in Section 4.1.3.

For convenience, we define a function to pick up points at which an event's status changes within a trace. For a trace t and event x , let $\mathbf{breaks}(t, x) \in \mathbf{seq} N$ be such that:

$$\begin{aligned} \forall 1 \leq i < j \cdot \mathbf{breaks}(t, x)[i] < \mathbf{breaks}(t, x)[j] \\ \mathbf{ran}(\mathbf{breaks}(t, x)) = \{i \mid (i = 0) \vee (\{x\} \cap t[i] \neq \{x\} \cap t[i + 1])\} \end{aligned}$$

This is an example of a syntactic abbreviation that is applicable to (although not necessarily useful in) all SRPT processes. Here, $\mathcal{T}_{\mathcal{R}}$ and Σ refer to all possible SRPT trace and event sets.

The specification $S(k)(t)$ for $t \in \mathcal{T}_{\mathcal{R}}[[WATCH_k]]\sigma$ is then:

$$\begin{aligned} S(k)(t) &\equiv \\ (\forall i < j : d \notin t[i]) &\quad \wedge \quad (\forall i \geq j : d \in t[i]) \\ \text{where: } B &= \mathbf{breaks}(t, w) \\ a &= \mathbf{min} m : B[m + 1] - B[m] \geq (k + 1) \\ j &= B[a] + k \end{aligned}$$

Here a is the number of the earliest break point after which the same value is received along the input wire too many times in succession. j is the trace index following this break point where the failure signal d starts to appear.

Watchdog process derivation

We now define specifications for processes $W0_k(x)$ and $W1_k(x)$. The $W0_k(x)$ process describes a watchdog where the last input toggle was to 0 (low voltage) and there are x steps left until shutdown trigger. $W1_k(x)$ is the same except that the last input toggle was to 1 (high voltage).

Specification $S_{0,k}(x)(t)$ is true iff $t \in \mathcal{T}_{\mathcal{R}}[[W0_k(x)]]\sigma$:

$$\begin{aligned}
S_{0,k}(x)(t) &\equiv \\
(\forall 0 \leq i < x \cdot d \notin t[i]) &\wedge \\
(\forall 0 \leq i < x \cdot w \notin t[i] \Rightarrow \forall j \geq x \cdot d \in t[j]) &\wedge \\
(x \geq 1 \wedge w \in t[0] \Rightarrow S_{1,k}(k)(t[1\dots])) &\wedge \\
(x \geq 1 \wedge w \notin t[0] \Rightarrow S_{0,k}(x-1)(t[1\dots])) &
\end{aligned}$$

$S_{1,k}(x)(t)$ is defined similarly, reversing membership tests of w and swapping instances of S_1 and S_0 .

For $x = 0$ the processes satisfying these specifications are trivial:

$$\begin{aligned}
W0_k(0) &= [!\{d\} ?X \rightarrow W0_k(0)] \\
W1_k(0) &= [!\{d\} ?X \rightarrow W1_k(0)]
\end{aligned}$$

For all $x \geq 1$ we use a recursive definition of the required processes. Assuming that $W0_k(x-1)$ and $W1_k(x-1)$ have been defined and satisfy $S_{0,k}(x-1)(\cdot)$, $S_{1,k}(x-1)(\cdot)$, we can define processes for value x as:

$$\begin{aligned}
W0_k(x) &= [!\emptyset ?X \rightarrow \mathbf{if} (w \in X) \mathbf{then} W1_k(k) \mathbf{else} W0_k(x-1)] \\
W1_k(x) &= [!\emptyset ?X \rightarrow \mathbf{if} (w \in X) \mathbf{then} W1_k(x-1) \mathbf{else} W0_k(k)]
\end{aligned}$$

The structure of these processes is sufficiently similar to the structure of the specifications for specification satisfaction to be clear.

Watchdog specification satisfaction

It remains only to show that $S_{0,k}(k)(t)$ corresponds to our original specification $S(k)$ for $W0_k(k)$. We can then state that the SRPT description of $WATCH_k$ is $W0_k(k)$, and we will have satisfaction of the specification. We will not aim for a full formal proof, but instead show the main derivation steps required.

Our proof is two-stage. First we show that the specifications agree that d either never appears, or that there is some index j where d starts to appear in the trace and will always appear from then on.

For $S(k)(t)$, this is clear from the specification:

$$(\forall i < j : d \notin t[i]) \wedge (\forall i \geq j : d \in t[i])$$

For $S_{0,k}(k)(t)$ this follows since the following is part of all S_0 and S_1 specifications, with the w membership test inverted for S_1 :

$$\begin{aligned}
(\forall 0 \leq i < x \cdot d \notin t[i]) &\wedge \\
(\forall 0 \leq i < x \cdot w \notin t[i] \Rightarrow \forall j \geq x \cdot d \in t[j]) &
\end{aligned}$$

We now show that the value of that index j is the same in the two specifications. For $S(k)(t)$, j is defined by:

$$\begin{aligned}
B &= \mathbf{breaks}(t, w) \\
a &= \mathbf{min} m : B[m+1] - B[m] \geq (k+1) \\
j &= B[a] + k
\end{aligned}$$

Time	1	2	3	4	5	6
Process	$W0_k(2)$	$W0_k(1)$	$W1_k(3)$	$W1_k(2)$	$W1_k(1)$	$W1_k(0)$
Input	-	w	w	w	w	-
Output	-	-	-	-	-	d

Table 4.2: Example of a trace of the watchdog

For $S_{0,k}(k)(t)$, we make an inductive argument on the “crucial” break number a . If $a = 1$, $j = k$ since $B[1] = 0$ by definition of **breaks**. This corresponds to the case where the event w does not appear for k successive points in the trace. The S_0 specification part

$$\begin{aligned}
& (\forall 0 \leq i < k \cdot d \notin t[i]) \quad \wedge \\
& (\forall 0 \leq i < k \cdot w \notin t[i] \Rightarrow \forall j \geq k \cdot d \in t[j])
\end{aligned}$$

corresponds to this case.

If $a > 1$, we need to show that all preceding breaks with index below a do not trigger a d sequence. It is clear from the preceding part of the proof that a d sequence, if it starts at all, must start at index k from the start of a process specified by $S_{N,k}(k)(t)$. We need only show that the breaks correspond to the start of $S_{N,k}(k)(t)$ specifications. This comes from the second part of the specification for $S_{0,k}(k)(t)$:

$$\begin{aligned}
& (k \geq 1 \wedge w \in t[0] \Rightarrow S_{1,k}(k)(t[1\dots])) \wedge \\
& (k \geq 1 \wedge w \notin t[0] \Rightarrow S_{0,k}(x-1)(t[1\dots]))
\end{aligned}$$

Since S_0 only applies to traces where w is currently absent, the appearance of w indicates a break point and hence a switch to $S_{1,k}(k)(s)$ for the remainder of the trace s . A mirror argument holds for S_1 . This gives us that $S(k)(t) \equiv S_{0,k}(k)(t)$. \square

Watchdog trace example

An example of w being “stuck-on” for $k = 3$ is shown in Table 4.2.

Keyword checker specification

A keyword checker is a process which takes as input a w -bit keyword along with a lines which denote the actuator to activate. No more than one of the actuator lines may be raised at any one time.

We will define an SRPT process $KEYW_{w,a}$ with the following behaviour. When an actuator line is raised, the keyword checker validates the given keyword against the actuator line selected: the result is one of *on*, *off* or *bad*. If *on* then the checker raises the appropriate actuator output line. If *off* or *bad* then it lowers the line, and if *bad* or more than one input actuator line is raised then it sets a “failure” output for one timestep.

We assume that $on \neq off \neq bad$.

This is a more complex example of the specification and SRPT description of a process. Again, we define the events of the system $KEYW_{w,a}$ first:

$$\begin{aligned}
P &= \{p_1 \dots p_a\}, Q = \{q_1 \dots q_w\}, R = \{r_1 \dots r_a\} \\
I &= \iota KEYW_{w,a} = P \cup Q \\
O &= oKEYW_{w,a} = R \cup \{f\}
\end{aligned}$$

P events are actuator selection, Q events form keywords, and R events are actuator controls. $\{f\}$ is the failure signal.

We define the internal event set K to represent keyword evaluation:

$$K = \{on, off, bad\}$$

The keyword evaluation is given by function \mathbf{wev} , mapping a set of keyword events and actuator number to an action word:

$$\mathbf{wev} : \mathbb{P}Q \times \mathbb{N} \rightarrow K$$

The specification $S(t)$ of each trace $t \in \mathcal{T}_{\mathcal{R}}[[KEYW_{w,a}]]\sigma$ is as follows:

$$\begin{aligned}
S(t) &\equiv \forall i \in \mathbb{N} \cdot \forall j \in 1 \dots a \cdot \\
\#(P \cap t[i]) = 0 &\Rightarrow t[i+2] \cap O = (t[i+1] \cap O) \setminus \{f\} \\
(P \cap t[i]) = \{p_j\} &\Rightarrow (z = on) \Leftrightarrow (r_j \in t[i+2]) \\
&\quad \wedge (z = bad) \Leftrightarrow (f \in t[i+2]) \\
&\quad \wedge (z = off) \Leftrightarrow (r_j, f \notin t[i+2]) \\
\#(P \cap t[i]) \geq 2 &\Rightarrow t[i+2] \cap O = \{f\}
\end{aligned}$$

where $z = \mathbf{wev}(Q \cap t[i], j)$.

This requires that:

- zero commands will maintain the status quo except that a failure will cease to be flagged;
- exactly one actuator command will cause the actuator to turn on (if *on*), off (if *off*) or flag an error (if *bad*); and
- more than one actuator command at once will cause a failure and turn all output controls off.

Note that this system has a 2-cycle delay rather than the 1-cycle delay of the previous example. This is because the eventual implementation is in terms of two processes processing in sequence, hence at least two cycles are required for this implementation to be feasible.

We define internal event set M to represent the actuator chosen to be activated:

$$M = \{m_0 \dots m_a\},$$

The process description is a parallel composition with hidden events, as follows:

$$KEYW_{w,a} = (KW_{w,a}(\emptyset) \parallel FILTER_a(\emptyset)) \setminus (M \cup K)$$

where KW evaluates the keywords and $FILTER$ acts on the output of KW to select the outputs. Note that the hiding operation applies to KW and $FILTER$ rather than $KEYW$; this distinction is important as hiding is defined in terms of a subset of a process's output alphabet, and $KEYW$ does not contain K or M .

We will specify $KW_{w,a}$ first. Given alphabets

$$\iota KW_{w,a} = P \cup Q, oKW_{w,a} = M \cup K$$

the specification $K(t)$ of each trace $t \in \mathcal{T}_{\mathcal{R}}[[KW_{w,a}]]\sigma$ is:

$$\begin{aligned} K(t) &\equiv \forall 0 \leq i \cdot \\ \wedge (t[i] \cap P) = \emptyset &\Rightarrow (t[i+1] \cap oKW) = \emptyset \\ \wedge (t[i] \cap P) = \{p_j\} &\Rightarrow (t[i+1] \cap oKW) = \{k, m_j\} \\ \text{where } k = \mathbf{wev}(t[i] \cap Q, j) & \\ \#(t[i] \cap P) \geq 2 &\Rightarrow (t[i+1] \cap oKW) = \{bad\} \end{aligned}$$

A suitable process satisfying this specification follows:

$$\begin{aligned} KW_{w,a}(R) = [!R ?X \rightarrow \mathbf{if} \quad \#(X \cap P) \geq 2 & \quad KW_{w,a}(\{bad\}) \\ \mathbf{elsif} \quad \exists j : (X \cap P) = \{p_j\} & \quad KW_{w,a}(\{k, m_j\}) \\ \mathbf{else} & \quad KW_{w,a}(\emptyset) \\ \text{where} & \quad k = \mathbf{wev}(X \cap Q, j) \end{aligned}$$

As an implementation detail, note that the case where exactly one element of P is present in the input need not search all the elements of P in sequence; instead, the p_j inputs would be linked to the m_j outputs with an intervening AND gate to check that all other elements of P are low. As the size of P grows, this becomes less likely to be feasible within the single clock cycle specified unless the target device provides AND gates with many inputs.

We now specify process $FILTER_a$. Given alphabets:

$$\iota FILTER_a = I = M \cup K, oFILTER_a = O = R \cup \{f\}$$

the specification $F(t)$ of each trace $t \in \mathcal{T}_{\mathcal{R}}[[FILTER_a]]\sigma$ is:

$$\begin{aligned} F(t) &\equiv \forall 0 \leq i \cdot \\ bad \in t[i] &\Rightarrow (t[i+1] \cap O) = \{f\} \\ \wedge (t[i] \cap M) = \emptyset &\Rightarrow (t[i+1] \cap O) = (t[i] \cap O) \setminus \{f\} \\ \wedge \exists j : (t[i] \cap I) = \{on, m_j\} &\Rightarrow (t[i+1] \cap O) = \{r_j\} \\ \wedge \text{otherwise} &\Rightarrow (t[i+1] \cap O) = \emptyset \end{aligned}$$

A suitable process satisfying this specification is:

$$\begin{aligned} FILTER_a(S) = [!S ?Y \rightarrow \mathbf{if} \quad bad \in Y & \quad FILTER_a(\{f\}) \\ \mathbf{elsif} \quad Y \cap M = \emptyset & \quad FILTER_a(S \setminus \{f\}) \\ \mathbf{elsif} \quad \exists j : Y = \{on, m_j\} & \quad FILTER_a(\{r_j\}) \\ \mathbf{else} & \quad FILTER_a(\emptyset) \end{aligned}$$

A proof of correctness here would be repetitive given the earlier satisfaction argument for Watchdog, but the principle strategy is to observe that *KW* outputs events that control the output of *FILTER*, but not vice versa. Therefore we define $R = (KW_{w,a}(\emptyset) \parallel FILTER_a(\emptyset))$ and since

$$\forall u \in \mathcal{T}_{\mathcal{R}}[[R]]\sigma \cdot (F(u) \wedge K(u))$$

by definition, we expand the definition of *K* by evaluating how the RHS of the clauses of *K* maps onto the LHS of the clauses of *F*, hence rewriting the RHS of *K* in terms of the output alphabet of *FILTER*. This is then compared with the definition of *S* to show that $\forall u \in R \cdot F(u) \wedge K(u) \Rightarrow S(u)$.

This example has shown how parallel composition can be used to form processes with internal events providing communication between them.

4.1.8 Non-rigorous components

The refinement model also allows us to incorporate “black box” processes into our overall design. As long as we can specify the inputs and outputs of a black-box process *B* in terms of events in Σ , we can reason about its interaction with the other processes for which we have more rigorous specifications.

For instance, edge areas of an FPGA may be given over to an I/O pad implementing an interface protocol such as the PC peripheral connector standard PCI. Mak[Mak03] discusses the thorny problems involved in placing these I/O pads when multiple I/O standards (and hence varying voltages) are present in the device. We need not be concerned about the specific implementation details of the I/O pad, and it need not even run at the same clock as the rest of our FPGA model as long as there are intervening gates outside our model but with the same clock, buffering the I/O voltages. As long as we can make some statements about the transitions of the outputs from the I/O pad, and establish minimum-switch times for the inputs, it need not affect our ability to reason about the behaviour of the rest of the device.

4.1.9 Commentary

We have taken two typical components of a safety-critical system which have the potential to be implemented using PLDs, have provided SRPT-based specifications and implementations for them and proven that the behaviour of the implementations satisfies the specifications.

We have seen that carefully-chosen syntactic abbreviations can express the SRPT trace-based specifications in a few lines and yet make rigorous and useful statements about the required properties of a process. The previous section has shown how it is possible to prove rigorously that a SRPT process description satisfies a specification, though clearly there is some way to go until this proof mechanism is easy enough to use effectively in a commercial project.

Note that there is a clear gap between the SRPT description of a process and its final implementation as a set of programmed cells in a LUT-based FPGA. It is however relatively simple to map such SRPT descriptions as given here into equivalent VHDL, Pebble or netlist formats. We expand on this in Section 4.2.7.

We have used SRPT as a compromise between the high-level specification languages, such as Z, and the low-level implementation languages such as EDIF and VHDL. The tradeoff we make is in ease of specification against simplicity of compiling to our target format.

According to the definitions in Section 3.4 we can classify this work as *rigorous* since formal specifications and sketch proofs were provided.

4.1.10 Alternatives to SRPT

SRPT is far from the only method of describing reactive systems. In this subsection we present some established alternative methods.

Language details

In [Ber00], Berry presents the basics of the Esterel language, and reviews a number of other synchronous languages. He distinguishes between “reactive” systems, where the computer reacts to external events, and “interactive” systems where the computer’s clients request services from the computer. The latter requires attention to avoid deadlock and unfairness, and the former requires correct and timely operation. According to these definitions, SRPT describes reactive systems.

Languages such as Signal and Lustre use a data-flow programming style, routing data through “fixed” operation nodes. In hardware terms this is similar to programming a DSP chip or FPGA rather than a conventional microprocessor. Variables in the language consist of a sequence of values at a set of times, e.g. $X = \{X_1, X_2, \dots\}$. Nodes combine values of different variables. The flow of data in the system occurs at each (integer) time steps. Some variables may be over-sampled or under-sampled, flowing at whole time multiples faster or slower than the “master” flow.

The data-flow model corresponds well with our intuitive understanding of how data flows through an FPGA. The problems with this model would come with a variable depending on more than one time index of another variable, e.g. $X_{t+1} = Y_t + 2 * Y_{t-1}$. This would complicate the placement and routing of such programs within a PLD.

Berry terms the programming model of Esterel “imperative”, which is an extension of one common definition of imperative languages as sequential modifications to a state but does capture the intent of defining how the result is to be produced instead of what properties the result exhibits[IP96]. In this model the basic structure is a *module*. A module has a defined set of input and output events, and a “body” in which a conventional imperative program executes. The imperative program is able to do blocking waits (“await”) on input events and cause (“emit”) output events. Statements can be combined in parallel, so that a module can wait for a disjunction or conjunction of events, and there are language operators to support pre-emption and exception raising. Body statements execute instantaneous except where delay is required by the purpose of the statement, e.g. the “await” operator.

Esterel provides more powerful abstract operators than SRPT, but at the price of a semantic gap between the Esterel program and the corresponding FPGA netlist.

Language evaluation

These approaches to programming languages could conceivably be used to program PLDs. Indeed, there are commercial tools such as “Esterel Studio” (from Virtual Prototypes Inc.) which allow such programming. The reason why we have chosen SRPT as our representative language is that the process-event structure of an SRPT system maps naturally onto the block-wire combinatorial logic and routing model of most PLDs.

Esterel’s semantic gap with respect to FPGAs is its key weakness. Signal and Lustre’s data-flow model is an interesting expression of a class of programs, and closer to the FPGA model, but has the placement problems noted above.

Our choice

These languages are possible alternatives to SRPT, but they do not have an obvious advantage to SRPT for our purposes. Indeed, we have identified deficiencies in their support for targeting FPGAs.

Preliminary work by the author in establishing the suitability of SRPT for compilation to PLDs showed that SRPT’s semantics was suitably rich to support rigorous definition of programs and mapped well onto the PLD program model. In our work to date there have been no serious shortcomings of SRPT that have indicated that CCS-based or other algebras are superior for synchronous PLD programming.

4.1.11 Conclusions

In this section we have shown how SRPT can be used to model non-trivial FPGA programs and prove certain safety properties in a rigorous way. We have also seen that it provides a precise way of specifying the requirements for an FPGA program, which makes it easier to define correctness tests.

Of the targets in Chapter 3 we have addressed or partially addressed:

Target 1: *The process we define must be rigorous.*

We have established a formal specification system for SRPT processes and demonstrated rigorous justification that processes match their specifications.

Target 2: *The process must help the developer to write unambiguous programs.*

The trace-based specification of SRPT processes is an unambiguous notation, and our deterministic subset of SRPT described in Section 4.1.2 makes SRPT programs unambiguous.

Target 3: *The process must allow the programs to have sections written in a low-level language for speed and flexibility, but not allow these sections to compromise overall program reliability.*

We have explicitly considered non-rigorous components in Section 4.1.8.

Target 12: [00-54 8.5.2] *The analytical arguments provided shall include:*

- (i) *any formal arguments used in validation to show that the formal specification complies with the safety requirements;*
- (ii) *any formal arguments that the functional design satisfies the formal specification;*

(iii) for non-functional properties with specified safety requirements, analysis of the achieved behaviour, e.g.: performance, timing etc.;

(iv) analysis of the effectiveness of fault mitigation, for example use of such techniques as diverse implementations.

(i) is achieved by use of an unambiguous notation for specification. We demonstrated the proof system required by (ii) in the watchdog timer example. Timing requirements can be addressed by specifications about relative positions of events in traces, addressing (iii). We have not addressed (iv).

In the next section we will explore the relationship between SRPT and the Pebble synchronous programmable logic programming language.

4.2 Pebble

4.2.1 Introduction

In Section 2.4.6, we described the Pebble language for low-level programming of synchronous FPGAs. In this section we expand on this to give a more complete definition of Pebble, and show how SRPT processes can be mapped onto Pebble programs.

The version of Pebble described here is Pebble 3.0, as described in Appendix A of [Luk99]. Our comments on Pebble in this section are likely, but not certain, to apply to future versions of the Pebble language.

4.2.2 Target device issues

Pebble may be compiled onto a number of different PLDs. These devices may differ substantially in which “primitive” cells they support. For instance, one device’s cell may support any logic function of 3 inputs; another device’s cell may provide any function of 2 inputs on one output wire, and the inverted result of that function on the output wire. The primitives for a given device are typically stored in a prelude file that is supplied to the compiler.

A complete Pebble program will consist of a number of these primitive cells with a certain interlinking. Each cell will be one of a (likely small) set of types, e.g. 3-input AND, half-adder, single-input NOT. Normally these primitives will be chosen so that each of them can complete in one clock tick on the target device. However, it may be that some of the primitives require two or more ticks to complete whereas others only require one tick. In this case a naive compilation to the target device will have to add delays to each type of cell so that they all take the same time to complete calculation. In practice, it is likely that the circuit can be partitioned and optimised so that relatively few of the partitions need to operate at the maximum delay.

4.2.3 Language elements

A Pebble program consists of a set B of *block instantiations*, with links between blocks provided by a set W of *wires*. There is a set D of *block declarations* which can be considered as function signatures with named formal parameters from a set P . A block $(d \in D, f_I, f_O) \in B$ represents an agglomeration of logic function computational cells on the target device. The formal parameters of d are renamed to elements of W by the functions $f_I, f_O : P \rightarrow W$ for input and output wires respectively. In conventional imperative language terms, the block declarations are subprogram declaration and the wires are global program variables. The blocks correspond to actual subprogram calls.

Each block declaration $d \in D$ contains named formal parameter lists F_I and F_O which are sequences of input and output wire names respectively. The block declaration includes a (potentially null) list of width parameters G to allow instantiation of the block in a range of bit-widths. These parameters may be specified in the block declaration, or left open for when the block is later instantiated. The block declaration also includes a (potentially null) list of internal wires L .

A block declaration’s internal structure consists of a series (which may be null) of block instantiations. Note that these instantiations are not necessarily elements of B ,

since they may have wires from the block declaration's parameters. In addition there is a series of direct connections between wires.

Block instantiations consist of block declaration names with the formal parameters renamed to the names of wires in scope, i.e. chosen from the union of F_I , F_O and L . If the block instantiated has any unspecified width parameters (in G) then these must be set at the instantiation.

Block instantiation can also be done in groups using the **GENERATE FOR** mechanism, specifying an "instantiation loop" where the characteristics of each block instantiated inside the loop may depend on the loop variable.

4.2.4 Example

Taking the example of a combinational incrementer implemented from half adders, as described in the Pebble 3.0 manual [Luk99], Appendix A, section 8:

```

BLOCK main [c : WIRE; e : VECTOR (n-1..0) OF WIRE]
           [d : WIRE; f : VECTOR (n-1..0) OF WIRE];

BLOCK main [fcin  : WIRE; fdin  : VECTOR (n-1..0) OF WIRE]
           [fcout : WIRE; fdout : VECTOR (n-1..0) OF WIRE]
  VAR i;
  CONST n : GENERIC := 3;
  VAR lc : VECTOR(n..0) OF WIRE
BEGIN
  lc(0) <- fcin;
  GENERATE FOR i = 0 .. (n-1)
  BEGIN
    hadd[lc(i),fdin(i)][lc(i+1),fdout(i)]
  END;
  fcout <- lc(n)
END;
```

The first **BLOCK** statement is an instantiation of block **main**, and the binding of actual parameters to its formal parameter list given in the following **BLOCK** declaration. The wires **c**, **d**, **e**(0...n-1) and **f**(0...n-1) are actual wires in the system, and the Pebble simulator would be able to control the values of the input wires and measure the values of the output wires.

The declaration of **main** comes next. After listing the formal parameters, the next set of declarations are variables and wires whose scope is local to the **BLOCK** declaration. **i** is simply a loop variable. **n** is a generic width parameter; the declaration fixes it at 3, but it could as easily have been left unassigned and instead set at instantiation. **lc**(0...n) are internal wires, used to propagate the carry values along the chain of half-adders (**hadd**).

The body of the declaration first connects internal wire **lc**(0) to formal parameter **fcin**. The next statement is a multiple instantiation, the number of instantiations governed by the generic width parameter **n**. Each instantiation is of the half adder **hadd**, with connections governed by the instantiation number. The final statement connects formal parameter **fcout** to internal wire **lc**(n).

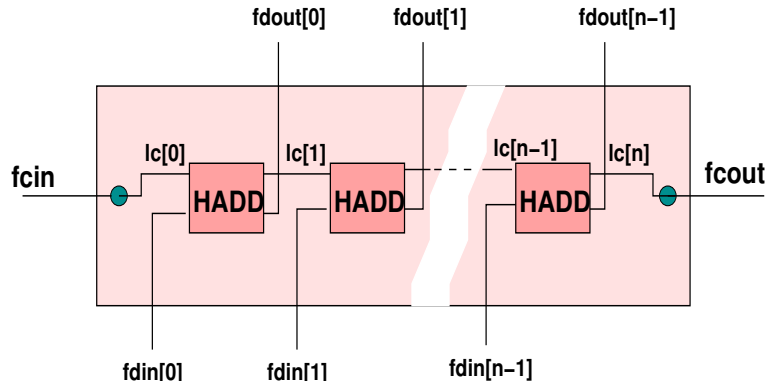


Figure 4.1: Combinational incrementer

Figure 4.1 shows a pictorial illustration of the *declaration* of `main`. The parameter n has been left unspecified. If n were 3, as specified in the block, there would be three HADD blocks in the diagram. The figure illustrates clearly that Pebble is mainly about defining relationships between predefined blocks by using shared wires. This is analogous to the way that SRPT defines relationships between processes by using shared events. In the next section we will explore this analogy in more detail.

4.2.5 Formal description

Following the earlier notation:

$$\begin{aligned}
 D &= \{\text{main}, \text{hadd}\} \\
 B &= \{(\text{main}, f_I, f_O)\} \\
 W &= \{c, d, e(0 \dots n-1), f(0 \dots n-1)\}
 \end{aligned}$$

where

$$\begin{aligned}
 f_I &= \langle c, e(0 \dots n-1) \rangle \\
 f_O &= \langle d, f(0 \dots n-1) \rangle
 \end{aligned}$$

For block `hadd` we define the formal parameters, internal wires and generic parameters as:

$$\begin{aligned}
 F_I &= \langle \text{fi1}, \text{fi2} \rangle \\
 F_O &= \langle \text{fs}, \text{fc} \rangle \\
 L &= \emptyset \\
 G &= \emptyset
 \end{aligned}$$

and for block `main`:

$$\begin{aligned}
 F_I &= \langle \text{fcin}, \text{fdin}(0 \dots n-1) \rangle \\
 F_O &= \langle \text{fcout}, \text{fdout}(0 \dots n-1) \rangle \\
 L &= \{\text{lc}(0 \dots n)\} \\
 G &= \{\mathbf{n} = 3\}
 \end{aligned}$$

Note that the `hadd` block, since it has no internal wires, is assumed to be a primitive of whatever target device it is instantiated on. It cannot be constructed out of other primitives in series since this would require internal wires to connect the primitives. It could be constructed out of primitives in parallel.

The instantiations of `hadd` in the declaration of `main` are as follows:

```
(hadd , ⟨fi1 = lc(0), fi2 = fdin(0), fs = fdout(0), fc = lc(1)⟩)
(hadd , ⟨fi1 = lc(1), fi2 = fdin(1), fs = fdout(1), fc = lc(2)⟩)
(hadd , ⟨fi1 = lc(2), fi2 = fdin(2), fs = fdout(2), fc = lc(3)⟩)
```

with the direct connections `lc(0) <- fcin` and `fcout <- lc(3)`.

Applying the renaming functions f_I, f_O of the `main` instantiation then produces the following fundamental instantiations where every block instantiated is a primitive component for the target device:

```
(hadd , ⟨fi1 = lc(0), fi2 = e(0), fs = f(0), fc = lc(1)⟩)
(hadd , ⟨fi1 = lc(1), fi2 = e(1), fs = f(1), fc = lc(2)⟩)
(hadd , ⟨fi1 = lc(2), fi2 = e(2), fs = f(2), fc = lc(3)⟩)
```

with direct connections `lc(0) <- c` and `d <- lc(3)`.

Note that although no formal parameters are left as wires there are local wires in these declarations such as `lc(0)`.

Having established this model, how does it react to data? Partly this will depend on the implementation of `hadd` in the target device; we assume that it is a conventional half-adder that outputs the carry on the first output and the sum on the second output. To have any meaningful basis for arguing about program correctness we must have verifiable functional and timing information about target device primitives.

The data flow through the model is modelled by a function $wire : W \times \mathbb{N} \rightarrow \mathbb{B}$ which is true for (w, t) iff wire w has a high voltage at time step t . Wires are considered bi-state (high or low voltage). We may ignore the possibilities of transients since the Pebble compiler manages these details; a “wire” in Pebble has delay and switching properties unlike a physical wire in electronic devices. The rule is that if wire w_i is connected directly to wire w_j then:

$$\forall n \geq 0 : wire(w_i, t) = wire(w_j, t + 1)$$

With wires connected directly in this fashion we describe wire w_i as the source of w_j , and similarly wire w_j as a destination of w_i .

Primitive gates such as the half-adder are defined by a function mapping sequences of input parameter values to sequences of output parameters. Sequences are represented in the expressions below by strings of binary digits, highest bit first. For `hadd`:

```
hadd : seq  $\mathbb{B}$   $\rightarrow$  seq  $\mathbb{B}$ 
hadd = {00  $\rightarrow$  00, 01  $\rightarrow$  01, 10  $\rightarrow$  01, 11  $\rightarrow$  10}
```

The rule for values flowing through an instantiation (hadd, f_I, f_O) is:

$$f_I = \{w_1, \dots, w_n\} \wedge f_O = \{v_1, \dots, v_m\} \Rightarrow \\ \forall t \geq 0 : \\ \langle wire(v_1, t + 1), \dots, wire(v_m, t + 1) \rangle = \\ \text{hadd} \langle wire(w_1, t), \dots, wire(w_n, t) \rangle$$

Note that this assumes that the instantiation of a device primitive computes all its outputs in one cycle and is stateless. The target may have other components such as RAM stores, which have state, or ROM stores which may take several time cycles to produce output. The target data supplied to the Pebble simulator will have to provide this information to allow accurate simulation.

4.2.6 Completeness of definition

Given the above description, it is useful to know whether the system is completely defined. It may be, for instance, that an instantiated gate has one input wire which is not a destination wire of any other gate, nor the destination of any other wire. Such a gate can be regarded as *floating* with no defined values at any time step.

Similarly, if a wire is the destination of more than one gate or wire, it is regarded as *shorting* these sources, and again has no defined value at any time step.

Floating wires are useful because they provide the ability to input data to the system. In the above instantiation in Equation 4.3, we see that wires c, e_0, e_1, e_2 float. No wires are shorted, which should be normal policy.

We now take the formalism developed so far and translate it into SRPT terms.

4.2.7 SRPT representation

Using the notation given above, we map each of the wires in W onto a unique event in Σ . A block declaration $d \in D$ corresponds to a process description P_d .

A block instantiation $(d, f_I, f_O) \in B$ corresponds to the equivalent process P_d being renamed with events in Σ . f_I and f_O produce the input and output alphabets of the process.

The SRPT process $CT[x, y \setminus s, d]$ connects wire \mathbf{s} to wire \mathbf{d} , equivalent to having the input of source \mathbf{s} appear one cycle later on in destination \mathbf{d} :

$$\begin{aligned} \iota CT &= \{x\} \\ oCT &= \{y\} \\ CT_X &= [!X ?Y \rightarrow \mathbf{if } x \in X \mathbf{ then } CT_Y \mathbf{ else } CT_{\{\}}] \end{aligned}$$

If a system Z consists of the instantiated processes P_1, \dots, P_k then the floating wires are those in

$$FLOAT_Z = (\cup_{i=1}^k \iota P_i) \setminus (\cup_{j=1}^k oP_j)$$

and the shorted wires are those $s \in SHORT_Z$ such that

$$\exists i, j : (i \neq j) \wedge (s \in oP_i) \wedge (s \in oP_j)$$

We have already provided the SRPT definitions for a gate computing an arbitrary n -bit function f in Section 4.1.6 as $CELL_{n,f}$. Here we provide SRPT definitions for some other useful logic constructs, ROM and RAM. Within a typical safety-critical system, ROM is used to store constant look-up tables (e.g. for bomb aiming data with varying wind speed and direction), and RAM for holding PLD program state that is too large to store in the available collections of registers.

These definitions must capture the behaviour of typical real implementations of these constructs, so will be more complex than the gate-based examples from earlier. If high-SIL subsystems are based on these definitions then we must rigorously test the real implementations with test data based on the behaviour of these definitions, and demonstrate that the real behaviour refines the definitions' behaviour. These definitions illustrate that real-world components can be modelled in SRPT, and provide a measure of their complexity in SRPT terms.

ROM

ROM provides a read-only store of data grouped in words, using an input address to index a given individual word and then putting the word data onto its output.

A ROM table has 2^m entries of n bits. We assume that lookup is done in t steps and that the lookup is not pipelined (so that the inputs must remain stable for t steps for the output to be valid). A 1-step ROM table (the lowest feasible value of t), if given address input data at time index i , will output word data at time index $i + 1$. If the (distinct) address bits are represented by set $A = \{a_1, \dots, a_m\}$, the data bits by $D = \{d_1, \dots, d_n\}$ and the internal data is modelled by the function $d : \mathbb{P}A \rightarrow \mathbb{P}D$ then the SRPT definition of ROM for fixed m, n, t, d is:

$$\begin{aligned}
\iota ROM &= A \\
oROM &= D \\
ROM &= ROM_{\{\},1,\{\}} \\
ROM_{X,i,Z} &= [!X ? Y \rightarrow \\
&\quad \mathbf{if } i = t \wedge Z = Y \mathbf{ then } ROM_{d(Z),t,Z} \\
&\quad \mathbf{elseif } Z = Y \mathbf{ then } ROM_{X,i+1,Z} \\
&\quad \mathbf{else } ROM_{X,1,Y}]
\end{aligned}$$

This implementation provides deterministic behaviour in the case where the specification does not define it, i.e. the case of a read address being changed before the output has been sent. It will start off a new read in this case, dropping the previous request. In all cases, the output will stay the same from cycle to cycle until a read-output cycle has been computed.

Because this process carries significant internal state it is not equivalent to a simple combination of $CELL_{n,f}$ functions. To incorporate it in a safety-critical system we would have to make formal specifications of its behaviour and show that they are met.

RAM

A RAM table is more complex. It has two modes: read and write. In real RAM blocks reading is often quicker than writing. We assume that the inputs must remain stable for u steps for the write to be effective, whereas reading occurs in t steps as in the ROM block. There is no explicit indication of when the outputs are valid; users of RAM blocks must know the timing properties of their blocks and design the surrounding circuits accordingly.

We take the m -element address and n -element data sets A and D from the ROM definition above. The RAM block internal function $d : \mathbb{P}A \rightarrow \mathbb{P}D$ will, unlike the

ROM block, change during operation as writes are made. The extra input w controls whether a write is being commanded, and the n extra inputs E supply data for input. For fixed m, n, t, u :

$$\begin{aligned} E &= \{e_1, \dots, e_n\} \\ \iota RAM &= \{w\} \cup A \cup E \\ oRAM &= D \\ RAM &= RAMR_{\{\},t,\{\},\{\mathbb{P}A \mapsto \{\}\}} \end{aligned}$$

We define a pair of process sets, one for reading operations and one for writing operations. The reading operation is $RAMR$:

$$\begin{aligned} RAMR_{X,i,Z,d} &= [!X ?Y \rightarrow \\ &\quad \mathbf{if} \ i = 1 \wedge (Y \setminus E) = Z \ \mathbf{then} \ RAMR_{d(X \cap A),1,Z,d} \\ &\quad \mathbf{elsif} \ i > 1 \wedge (Y \setminus E) = Z \ \mathbf{then} \ RAMR_{X,i-1,Z,d} \\ &\quad \mathbf{elsif} \ w \in Y \ \mathbf{then} \ RAMW_{X,u,Y,d} \\ &\quad \mathbf{else} \ RAMR_{X,t,Y \cap A,d}] \end{aligned}$$

and the writing operation is $RAMW$:

$$\begin{aligned} RAMW_{X,i,Z,d} &= [!X ?Y \rightarrow \\ &\quad \mathbf{if} \ i = 1 \wedge Y = Z \ \mathbf{then} \ RAMW_{q(Z \cap E),1,Z,d'} \\ &\quad \mathbf{elsif} \ i > 1 \wedge Y = Z \ \mathbf{then} \ RAMW_{X,i-1,Z,d} \\ &\quad \mathbf{elsif} \ w \notin Y \ \mathbf{then} \ RAMR_{X,t,Y \cap A,d} \\ &\quad \mathbf{else} \ RAMW_{X,u,Y,d}] \end{aligned}$$

$$\text{where} \quad d_i \in X \iff e_i \in q(X)$$

$$\text{and} \quad d' \equiv d \oplus (Z \cap A \mapsto q(Z \cap E))$$

Section 6.4.3 and Section 6.2.3 in a later chapter will demonstrate the use of ROM and RAM blocks in a complex PLD program.

4.2.8 SRPT to Pebble

Having shown how Pebble constructs can be mapped into SRPT, we now examine how SRPT processes can be refined into Pebble.

Constructors

The correspondence between Pebble wires and SRPT events has already been noted. For an SRPT process P with alphabets ιP and oP , we declare a Pebble block Pb_P with formal parameters matching the union of the alphabets of P .

As previously noted, SRPT has a set of basic constructors. We deal with each of them in turn.

- Process variable x corresponds to an instantiation of a declared block Pb_x .
- $P \parallel Q$ is a Pebble block which contains the instantiations of Pb_P and Pb_Q .

- $P \setminus O$ is a Pebble block where the wires in O are removed from the formal parameters list and instead made internal wires by adding them to the Pebble block's internal wires list L .
- $P[S]$ is an instantiation of a declared block Pb_P with formal parameters replaced by actual wires as defined by the renaming function $S : P \rightarrow W$.

More complex is $P = [!O ?X \rightarrow P_X]$. The way that we define P in Pebble will depend on P .

Stateless processes

We assume first of all that we can define a Pebble block equivalent to any “stateless” SRPT process, i.e. if we define the group of Pebble processes

$$P_f(Y) = [!Y ?X \rightarrow P_f(f(X))]$$

where $f : \mathbb{P}_\iota P_f \rightarrow \mathbb{P}_o P_f$, then there is an equivalent Pebble block Pb_Pf . This should be feasible as long as the primitive gates provided in the Pebble library include *NAND* since any logic function can be constructed from these gates. As noted in Section 4.1.3, the if-then-else construction represents a straight map from input events to output events.

A significant problem is that there is no external control over the output of any Pebble block on the first tick of the clock, whereas we can specify this output in SRPT. In practice it is conventional for blocks to assume the output corresponding to low voltages on all inputs; an OR gate would then output a low voltage on the first clock tick, whereas a NAND gate would output a high voltage.

Therefore we allow the SRPT processes $\{P_f(Y) \mid Y \subseteq oP_f\}$ as above, but make the restriction that when any such process is instantiated, the first output events must be $f(\{\})$. This must be manually checked for each SRPT process definition in our system.

Processes with state

If the process has state, we write the process description as

$$P_{f,q}(Y) = [!Y ?X \rightarrow P_{f,g(X,q)}(f(X, q))]$$

where $q \in \mathbb{N}$ and $g : \mathbb{P}_\iota P \times \mathbb{N} \rightarrow \mathbb{N}$. This is harder to represent. We need a way for the Pebble blocks to track the current state. In this case we would have to define a Pebble block Pb_Pg which computed the state transform function g , as well as a block Pb_Pf which computed f , and connect them so Pb_Pg fed into the state inputs of Pb_Pf as well as to its own inputs. Figure 4.2 shows such a layout.

Note that Pb_Pf and Pb_Pg have enabling inputs e_f, e_g which must be high for their output to change; this prevents incorrect outputs occurring during the computation

The progress of state throughout these blocks is illustrated in Table 4.3. Starting in a stable state, a change of input from x_0 to x_1 propagates through to a state change and output in two ticks. Note that the new state propagates through to Pb_Pf the clock tick after Pb_Pg is giving the correct output; it is only then that another change of X will pass through the state block correctly.

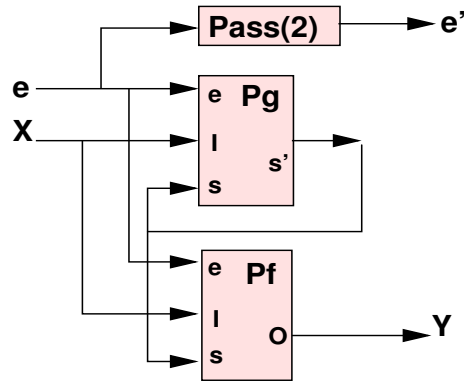


Figure 4.2: Pebble blocks tracking state

Time	X	e	s	s'	Y
0	x_0	0	s_0	s_0	$f(x_0, s_0)$
1	x_1	1	s_0	s_0	$f(x_0, s_0)$
2	x_1	0	s_0	$g(x_1, s_0)$	$f(x_1, s_0)$
3	x_1	0	$g(x_1, s_0)$	$g(x_1, s_0)$	$f(x_1, s_0)$

Table 4.3: State changing process

SRPT processes with more than one numerical state index can be transformed into single-number index forms by an appropriate diagonalisation function. Note that we may not make the right hand function depend on Y explicitly, according to this classification.

We now give an example of translating SRPT to Pebble.

4.2.9 Example: SRPT to Pebble

A common data structure is a *stack*, modelling the First-In, First-Out (FIFO) data flow. The basic operations on a stack are Push (insert a datum onto the top of the stack) and Pop (remove the datum on top of the stack). This example is a stack modelled in Pebble.

We define a stack as follows. We assume that it has a capacity of 2^m entries, each of n bits. We note that the behaviour of a fixed-depth stack is that of a RAM block; we assume (for simplicity) that this RAM block is single-tick read/write. Our stack's behaviour is to output continuously the number last input.

We take our previous RAM block definition and simplify it accordingly to give an SRPT description of the RAM block component of our stack.

$$\begin{aligned}
iSRAM &= \{w\} \cup A \cup E \\
oSRAM &= D \\
SRAM &= SRAM_{\{\}, \{\}} \\
SRAM_{X,d} &= [!X ? Y \rightarrow \\
&\quad \text{if } (w \notin Y) \text{ then } SRAM_{d(Y \cap A), d} \\
&\quad \text{else } SRAM_{q(Y \cap E), d'}]
\end{aligned}$$

where $d' = d \oplus (Y \cap A \mapsto q(Y \cap E))$.

The stack has an input data stream, and a pair of controls which specify whether the input data is to be pushed (*push*) or popped (*pop*). When a pop signal is received, the data next output will be the input data last-pushed-but-one.

Another part of the stack process will control the interface to the RAM block. The RAM block will output the value of the top element on the stack, so a push will have to increment the address value and a pop will have to decrement the address value. In addition, it controls the write bit of the RAM block so that a pushed value is written in.

$$\begin{aligned}
\iota SCTRL &= \{push, pop\} \\
oSCTRL &= \{w\} \cup A \\
SCTRL &= SCTRL_{\{\}, 0} \\
SCTRL_{X,k} &= [!X ? Y \rightarrow \\
&\quad \mathbf{if} (push \in Y \wedge k < 2^m - 1) \mathbf{then} SCTRL_{a(k+1) \cup \{w\}, k+1} \\
&\quad \mathbf{elseif} (pop \in Y \wedge k > 0) \mathbf{then} SCTRL_{a(k-1), k-1} \\
&\quad \mathbf{else} SCTRL_{X \cap A, k}]
\end{aligned}$$

where $a : \mathbb{N} \rightarrow \mathbb{P}A$ encodes a numerical address into the appropriate bits. We have refined our informal description of the stack to define unspecified behaviour, specifically the actions for full and empty stacks and for both commands occurring at once (*push* has priority). This corresponds to the implementation decisions made during conventional coding.

Another process we will need is $PASS_n$ which is an n -bit wide single-delay pass gate.

The definition of $STACK$ is now a direct composition of processes with appropriate event renaming:

$$\begin{aligned}
F &= \{f_1, \dots, f_n\} \\
\iota STACK &= F \cup \{push, pop\} \\
oSSTACK &= D \\
STACK &= (SRAM \parallel PASS_n[F][E] \parallel SCTRL) \\
&\quad \setminus (E \cup \{w\} \cup A)
\end{aligned}$$

Note that the process has a two-cycle delay. At the end of the first cycle $SCTRL$ has set the correct bits for entering the address, and the new data has gone through the pass block. At the end of the second cycle, $SRAM$ has updated itself accordingly and has output the new top stack data.

With regard to the SRPT, readers should note that the \parallel operator is associative according to Law 2 in Barnes[Bar93] §5.1.1. Explicit bracketing is therefore not required.

A diagram of this circuit is shown in Figure 4.3.

Translating $STACK$ to Pebble, we see that there are several process instantiations combined with a hiding operator; we must therefore define $\{f_1, \dots, f_n, a_1, \dots, a_n, w\}$ as internal wires. The parameters of Pb_STACK are taken straight from the process alphabet. The translations of processes $SCTRL$, $PASS_n$ and $SRAM$ blocks to Pebble are straightforward block instantiations as described below.

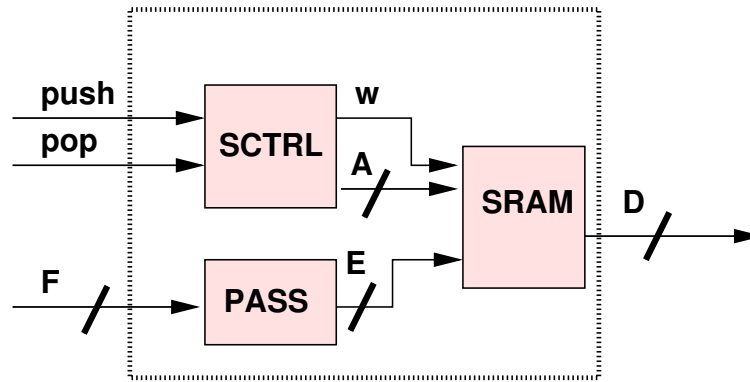


Figure 4.3: A simple stack

If `Pb_PASSn` does not already exist then it is simple to define, as it merely connects each input to a corresponding output with a one cycle delay. `Pb_SRAM` is a RAM block, which we assume to be primitive to our chosen target. If it is not a primitive then we will have to compose other primitives to build it, increasing its delay (and hence the delay of the stack block) by many cycles.

`Pb_SCTRL` is an output prefix process with state parameter k . We therefore use the previously-described design of an internal state generator block, instantiated along with the normal decision block. The state generator consists of one path generating the successor state, a second path generating the predecessor state, and a multiplexer to choose between them.

All this yields the following Pebble declaration:

```

/* Declarations for our target.
 * Assume that all these blocks are single-cycle.
 */
BLOCK ram(k1,k2 : GENERIC) [w : wire;
                             a : VECTOR (k1..1) OF WIRE;
                             e : VECTOR(k2..1) OF WIRE]
                             [d : VECTOR (k2..1) OF WIRE];

/* Incrementor; increments a by 1,
 * unless a is all 1s already */
BLOCK inc(k : GENERIC) [a : VECTOR (k..1) OF WIRE]
                       [b : VECTOR (k..1) OF WIRE]
BLOCK pass(k : GENERIC) [a : VECTOR (k..1) OF WIRE]
                       [b : VECTOR (k..1) OF WIRE]

/* Decrementor; decrements a by 1, unless a is 0 already */
BLOCK dec(k : GENERIC) [a : VECTOR (k..1) OF WIRE]
                       [b : VECTOR (k..1) OF WIRE]

/* 3-way multiplexer; select one of a, b or c as output d */
BLOCK mux3(k : GENERIC) [c1 : WIRE; c2 : WIRE;
                          a : VECTOR (k..1) OF WIRE;
                          b : VECTOR (k..1) OF WIRE;
                          c : VECTOR (k..1) OF WIRE]
                          [d : VECTOR (k..1) OF WIRE]

```

```

/* Output if x and not y */
BLOCK xandnoty[x : WIRE; y : WIRE] [z : WIRE]

/* Our own declarations */

/* SCTRL state generator; 2-cycle duration */

BLOCK sgen_sctrl (m : GENERIC)
    [push : WIRE; pop : WIRE;
     k : VECTOR(m..1) OF WIRE]
    [n : VECTOR(m..1) OF WIRE]
    VAR i : VECTOR(m..1) OF WIRE;
    VAR u : VECTOR(m..1) OF WIRE;
    VAR d : VECTOR(m..1) OF WIRE;
    VAR iw : WIRE;
    VAR dw : WIRE;
BEGIN
    /* k can either increment, decrement or stay
       the same */
    inc(m)[k][i];
    pass(m)[k][u];
    dec(m)[k][d];
    /* A multiplexer decides */
    mux3(m)[iw,dw,u,i,d][n];
    /* And the multiplex choice is determined by: */
    xandnoty[push,pop][iw];
    xandnoty[pop,push][dw];
END;

/* SCTRL itself */

BLOCK sctrl (m : GENERIC)
    [push : WIRE; pop : WIRE]
    [w : WIRE; a : VECTOR(m..1) OF WIRE]
    VAR n : VECTOR(m..1)
    VAR p : WIRE;
BEGIN
    /* Note the output-input loopback connection 'a' */
    sgen_sctrl(m)[push,pop,a][a];
    /* sgen_sctrl is 2-cycle so need a delay here */
    xandnoty[push,[pop][p];
    pass(1)[p][w];
END;

/* And now STACK. Note that we've had to add an extra
 * PASS block because sctrl is 2-cycle.
 * This means that STACK is now 3-cycle (assuming

```

```

* ram blocks are 1-cycle).
*/

BLOCK stack (m,n : GENERIC)
    [push : WIRE; pop : WIRE;
     f : VECTOR(n..1) OF WIRE]
    [d : VECTOR(n..1) OF WIRE]
    e1 : VECTOR(n..1) OF WIRE;
    e2 : VECTOR(n..1) OF WIRE;
    a : VECTOR(m..1) OF WIRE;
    w : WIRE;
BEGIN
    ram(m,n) [w,a,e2] [d];
    sctrl(m) [push,pop] [w,a];
    pass(n) [f] [e1];
    pass(n) [e1] [e2];
END;

```

It is important to note that mapping into Pebble has not been straightforward. We should have written the *SCTRL* description using a formal generator function from the outset. In addition, target device restrictions (needing two cycles to calculate the generator function) have meant the insertion of extra delays in order for all the data to match up. In later work in Chapter 5, when we look at refining SRPT processes and implementing them in Pebble, we will have to remember that timing issues are likely to appear in the Pebble mapping.

However, the above Pebble file appears to be an accurate description of a stack and is parametrised by data width (n) and logarithmic stack size (m). Its reliability will still have to be established by testing appropriate to its required reliability in systems.

4.2.10 Summary

In this section we have examined the Pebble language, summarising its main constructs and showing how these can be translated to and from similar SRPT constructs. This has established SRPT as a practical synchronous calculus in which to work, and has highlighted those SRPT constructs which should not be used in our future work.

Of the targets in Chapter 3 we have addressed or partially addressed:

Target 1: *The process we define must be rigorous.*

We have provided a systematic method for translating SRPT constructs into Pebble, although we have not produced rigorous demonstration that the semantics of the constructs are equivalent or refined.

Target 3: *The process must allow the programs to have sections written in a low-level language for speed and flexibility, but not allow these sections to compromise overall program reliability.*

We have allowed SRPT process declarations but not definitions, and shown how Pebble itself permits the description and incorporation of primitive blocks whose operation is undefined.

Target 6: *The program must be able to be compiled onto a range of existing and anticipated PLDs.*

Pebble can be translated into VHDL, and hence onto most PLDs (with the usual requirements for space).

Target 7: *The process must reuse existing proven tools where feasible.*

The Pebble-to-VHDL compiler already exists. An SRPT-to-Pebble compiler which needs to be created does not yet exist; it is necessary to bridge the gap between the abstract state of SRPT processes and the restricted state handling in Pebble.

Target 10: *The process should provide flexibility so that it may be used in situations not anticipated in its original design.*

This is addressed by the previously described facility to incorporate non-Pebble blocks into a Pebble program.

Target 14: *[00-54 13.3.1] A Hardware Specification shall be produced which defines the SREH in terms of its behaviour and properties.*

Pebble works by assuming uniform device-independent behaviour of the VHDL into which it is compiled. A step towards compilation of the Hardware Specification is verification of the behaviour of this VHDL subset.

In the next section we examine the SPARK Ada safety-critical systems development language. Our eventual aim will be to transform a SPARK Ada program fragment into an SRPT system, and from that form into an equivalent Pebble program. The following chapter will therefore evaluate SPARK Ada with that goal in mind.

4.3 SPARK Ada

4.3.1 Introduction to SPARK Ada

SPARK Ada is an annotated subset of the Ada language, as defined in the Ada 83 and 95 Language Reference Manuals[U.S83, Int95]. Its target market is safety-critical subsystems, which are often embedded. It supports substantial static analysis of programs including proof of absence of run-time exceptions, data and information flow analysis, and proof of correctness in the form of pre- and post-conditions on subprograms. Enforcement of the SPARK Ada subset and static analysis is done by the SPARK Examiner, a tool produced by Praxis Critical Systems Ltd. Proof of correctness and of absence of exceptions is aided by the SPADE Simplifier and Proof Checker, also Praxis tools.

For the purposes of this report we shall concentrate on the Ada 95 version of SPARK Ada, henceforth referred to as SPARK¹ for brevity. The syntax of the SPARK language is defined in the SPARK Report [FW99]. A more detailed description of and tutorial in SPARK Ada is given in the book “High Integrity Software – The SPARK Approach to Safety and Security” [Bar03] to which the reader is referred for more detail.

As an Ada subset, SPARK code can be compiled with existing industrial compilers and tools. For this reason it has been more successful than languages designed to bring more rigour into the software engineering development process, such as RSRE’s NewSpeak[Cur84]. Annex H of the Ada 95 Language Reference Manual[Int95] makes recommendations for restricting use of the full Ada language in Safety and Security applications, and SPARK’s language restrictions support these recommendations.

This section aims to demonstrate the suitability of SPARK Ada as a high-level language for implementing a design in a software / programmable hardware combination. We examine the features of the SPARK Ada language, and of its supporting tools to see how they support reliability and verifiability. We also see how they could be used to provide supplemental information to a compiler.

We then look at how SPARK Ada programs might be transformed into equivalent HDL or SRPT processes. This lays the foundations for the refinement work in Chapter 5 where we will aim to prove formally this equivalence, and the case study in Chapter 7 where we do a case study on extracting a fragment from a SPARK Ada program into an HDL form.

4.3.2 Safety-critical system development process

Our goal in producing a safety-critical system is to start with a well-defined set of requirements, produce a high-level design for the system, refine this into a program in a suitable high-level language, compile this into machine code and use the code to program a suitable processor-memory combination in the hardware of the system being produced. Section 2.1 examined current practice in this area.

Coupled with a rigorous development process, a design methodology well-matched to the language chosen and to the system being developed should be chosen. Although the choice of such a process will be dictated by the agency in charge of de-

¹Note: The SPARK programming language is not sponsored by or affiliated with SPARC International Inc. and is not based on the SPARCTM architecture.

velopment, and may use a diverse range of design tools (e.g. Rational Rose[EK99] or other tools based on UML) one methodology particularly well-suited to SPARK Ada is “INFORMED”[Ame00] which produces a top-down design that can be translated into efficient SPARK. The examples developed in this thesis will use the INFORMED approach.

The tools for SPARK Ada development are the SPARK Examiner[Cha01] and the SPADE Simplifier and Proof Checker[Pra95, Pra98]. The Examiner enforces the SPARK language restrictions, and produces proof conditions on program properties which may be proven (or disproven) via the other two tools. More detail on these tools is given in Section 4.3.4.

The top-down development of INFORMED relies on early and frequent use of the SPARK Examiner to ensure that the program maintains a consistent structure. This contrasts with the traditional bottom-up development in C or Ada where the compiler validates the code, hence the code analysed must form a compilation closure. INFORMED design can proceed from the top downwards, when much of the lower-level code is incomplete, because the SPARK language allows the developer to express their intentions for unwritten code with *annotations*. In Section 4.3.3 we describe what annotations are, and how they are used.

4.3.3 General language properties

As a general programming language, SPARK’s level of abstraction is approximately that of Ada, more abstract than standard C. Its type system is more detailed than the C type system and more strongly enforced than either the Ada or C type systems, at the cost of such operations as string checking or alteration requiring numerous type declarations and careful type conversions.

Compilation

SPARK, being a subset of Ada, will be compiled by any standard Ada 95 compiler, including the validated compilers being used in the industry such as GNAT Pro (Ada Core Technologies), ObjectAda (Aonix) and GMART (Green Hills). Indeed, the intent of the language restrictions is that SPARK programs cannot be “erroneous” in the Ada sense of producing different results with different compilers; for instance, the aliasing rules make the semantics of pass-by-reference and copy-in-out compilers equivalent for all SPARK programs. An added benefit is that since the SPARK subset throws out many of the more complex Ada constructs such as generics, it tends to tread the well-trodden (and hence well-tested) paths in the compiler.

Ada 83 and 95 have fairly good syntactic and semantic definitions in their respective Language Reference Manuals[U.S83, Int95], and SPARK builds on that with the SPARK Report [FW99], stating how the Ada 83 and 95 LRMs map on to SPARK Ada. As regards a formal definition, one has been written[Ltd94a, Ltd94b] by Program Validation Limited with support from the UK Defence Research Agency. This consists of the static and dynamic semantics of a subset of SPARK Ada, given in the Z language[Spi92]. The defined semantics has been used within Praxis but is insufficient to specify the current language subset because of two points: the language has since moved on (e.g. embracing Ada 95, allowing individual record fields as procedure

parameters, allowing read-only and write-only variables) and the subset fully defined omits some aspects of the language such as type ranges and named aggregates which are now common in SPARK programs.

Run-time

After compilation Ada programs are normally linked in with a compiler-specific *run-time* which provides the services associated with the more complex language properties such as tasking. Certain Ada language profiles such as GNORT (Ada Core Technologies), GMART (Green Hills) and Raven (Aonix) are designed to eliminate or minimise the size of this run-time for reliability and space reasons. The SPARK subset requires minimal run-time support and works with these profiles.

Typing

Ada's strong type system provides better visibility and enforcement of the numeric range of a variable than languages such as C afford. Ada's run-time `Constraint_Error` exception indicates that a variable's value has gone outside its defined type. Taking this idea further, the SPARK Examiner run-time checker generates verification conditions that aim to show that the code is free from run-time exceptions e.g., due to arithmetic overflow or to a variable's value falling out of type.

SPARK includes a subset of Ada 95 modular types, which is useful for arithmetic using arbitrary bit widths. It also includes the ability to declare types of arbitrary numerical range. This will enable us to perform calculations confident that a variable is within a restricted range of values, and the run-time checks generated would determine whether the result of the calculation will also fit in a restricted range.

Control flow

SPARK includes most of Ada's control flow constructs, except the `goto` statement. The restrictions it places on control flow relate to control flow graphs being well-formed according to the Semi-Structured Flow Graph grammar [FKZ75]. For instance, the exit points of a loop must always be at the "edge" of the loop, not inside compound statements within the loop. This ensures that each exit check is traversed once during a full loop.

The control flow restrictions allow information flow analysis as described by Carré and Bergeretti [CB85]. This is key to SPARK's ability to detect ineffective statements and use of potentially uninitialised variables.

Program structure

As a subset of Ada, SPARK has many features typical of high-level imperative languages, including a module hierarchy. The Ada language, and SPARK, provide two structural components for programs: packages and subprograms. A *package* comes in two parts: a *specification* which declares the types, variables and subprograms which it exports, and a *body* which contains the private data of the package as well as the implementations of all declared subprograms. Packages may contain state variables whose

values persist while the packages are in scope; for packages that are not embedded within a procedure, this state persists for the duration of the program.

The top level of an Ada program consists of a single `main_program` subprogram, commonly called `Main`, with any number of separate packages. Execution works through `Main` until the end of that subprogram. In practice, many embedded systems (irrespective of programming language) tend to run in an infinite loop after some initialisation calculations.

Packages and subprograms may be embedded in package bodies and in the local variable declaration area of subprograms. So, for instance, in the body of package `Q` might be a subprogram `Parse`, which relies on operations provided by a package `Stack` within it. `Stack` itself may have an internal subprogram `Pop`. Using the Ada dotted notation of nesting, a subprogram within `Parse` would refer to the `Pop` subprogram as `Stack.Pop` and the `main` subprogram would refer to `Pop` as `Q.Parse.Stack.Pop`. In practice, Ada visibility rules make this second reference illegal.

Ada 95 introduced *child packages* which, among other features, enable developers to split a single package specification into subunits, each of which has direct visibility of the basic types and subprograms declared by the parent package. SPARK supports these with additional restrictions on visibility.

Annotations

SPARK adds *annotations* to the subset of Ada that it uses. These are Ada comments (denoted by two dashes in sequence) followed by a third character, typically a hash. As a comment, an annotation has no effect on compiled code but is visible to the Examiner.

Annotations are used primarily to declare information that the Examiner must check on first inspection, then later may use to check items further up the package and subprogram hierarchy. They allow checks such as “no mutual recursion” to be made in linear time since SPARK visibility and declaration rules mean that a procedure `P` cannot call procedure `Q` if `P` comes before `Q`. Examples of these annotations are `--# own X` (declare package state `X`) and `--# derives X from Y` (expresses information flow of a subprogram operating on variables `X` and `Y`).

Visibility

Ada requires that packages explicitly list any other packages whose types, subprograms or variables they reference directly. This listing is done using the Ada `with` context clause. SPARK additionally requires that indirectly referenced packages are also listed, using the `--# inherit` annotation.

For instance, if package `P` contains state variable `V` which is changed by subprogram `P.X`, and procedure `Q.Y` in package `Q` calls subprogram `P.X`, then Ada would require that package `Q` list `P` as a referent. If subprogram `R.Z` in package `R` calls `Q.Y` then Ada would only require that `R` list `Q` as a referent in its `--# inherit` annotation; SPARK would however require that `P` also be listed.

These visibility rules allow the SPARK Examiner to prevent any circular references, which includes banning simple and mutual recursion in subprograms. The subprogram dependency directed acyclic graph allows the Examiner to define an examination order which has the following properties:

- each package specification is examined before its body; and
- each subprogram declaration is examined before any subprogram body containing a call to that subprogram is examined.

Banning recursion enables static calculation of the maximum depth of the stack during program execution, allowing the programmer to demonstrate that the stack will never overflow. This is of particular importance in embedded systems where programs are required to have a high mean time between resets.

State

A package may have any number of state variables. These come into scope and are given initial values (if specified) when the package is *elaborated*; for a top-level package this occurs at the start of the program execution. Package elaboration order is a significant issue in Ada, but the visibility rules in SPARK allow developers to ignore it.

A subprogram may declare any number of local variables. These, in addition to the subprogram parameters, are only in scope and retain data for the duration of the subprogram. This is also true for the state variables of any packages or other subprograms embedded in the subprogram.

Ada subprogram parameters are given *modes* which describe whether the parameter is an input (*in*), output (*out*) or both (*in out*). It is illegal to write to a mode *in* parameter, though it is legal to read an *out* parameter.

SPARK additionally requires that subprograms list in a `--# global` annotation *all* the state variables which they use, along with their modes. In the earlier example, subprogram *R.Z* would have to list variable *P.V* – even though *P.V* may well not be visible to it under Ada rules! Through the SPARK annotation, all the side effects of a subprogram can be known at analysis time, allowing precise *flow analysis*.

SPARK requires that the state variables in a package be declared in an `--# own` variable annotation in the package specification. Any variables declared in the body may be aggregated into a single abstract state variable. This enables encapsulation of the package state inter-dependencies in the body, reducing the complexity of annotations for any subprograms calling subprograms in the package specification.

Flow analysis

Data flow analysis[CB85] of a subprogram *S* validates that the variables imported and exported by the subprogram correspond to those specified by the user in the declaration and in the declarations of all subprograms called by *S*.

SPARK has the option of allowing *information flow analysis* as well. This goes further, allowing the developer to specify how the exported variables depend on the imported variables and checking that the program information flow matches the developer's design intent. This is done by computing the products and transitive closures of Boolean matrices representing the variable dependency information of individual subprogram statements.

Tasking

A significant omission in current SPARK, as compared to Ada, is Ada's notion of tasking. Tasking was omitted from the SPARK subset because it can be extremely complex and difficult to reason about.

Because the Ada 95 tasking model has improved on the Ada 83 tasking model, it has become possible to define subsets of the tasking constructs with desirable determinacy and performance properties. The Ravenscar tasking profile[BDR98] is the a deterministic scheduling subset of Ada 95 which will be adopted formally in the Ada 0Y language; in the meantime, it has been incorporated into release 7 of SPARK Ada[Cha03].

Memory-mapped I/O

Previous use of SPARK in embedded systems such as SHOLIS[KHCP99] using memory-mapped IO pointed to a problem in the way it treats variable initialisation. Suppose that we have a design that uses page zero of memory to communicate with a PLD or other piece of hardware across a bus. Ada (and, indeed, SPARK) allows us to define a variable supplemented with a “use clause” that specifies the exact memory location and / or data format to be used. We might define two 8-bit registers X and Y for input and output respectively thus:

```
BASE_ADDR : constant := 0;
type Byte is mod 256;
for Byte'Size use 8;
X : Byte;
for X'Address use (BASE_ADDR + 16#010#);
Y : Byte;
for Y'Address use (BASE_ADDR + 16#014#);
```

This maps X to location hex 010 and Y to location hex 014. Typical use would be to write a value to Y to transfer the data to a PLD, and to read from X to read data from the same PLD.

We might produce some control code which looks like:

```
Y := START_PROCESSING;
while (X /= ENDED_PROCESSING) loop
    Utilities.Sleep(5);
end loop;
Y := RESET_REGISTERS;
```

The intention of this is to start some processing in the PLD, then every 5 milliseconds poll the PLD for a “completed” flag. Once this is done we reset the PLD registers in preparation for a new calculation.

Naively, the SPARK Examiner would not accept this code. From its point of view, Y is being written to twice without being read, hence the first assignment is ineffective. And in the loop, X is not an export of procedure Utilities.Sleep so the loop will either not happen at all, or will be infinite.

However, the SPARK language now permits specification of variables as read-only or write-only, and the Examiner can correctly flow-analyse code which uses them. `X` and `Y` would be declared as package `own` variables where they would be given modes `in` and `out` respectively. The release note for the SPARK Examiner 6.0 [Cha01] describes this concept in detail in Appendix A; there are some complexities involving mixed mode state in package refinements that can trip up the unwary developer.

Since Ada programs are likely to use memory-mapped I/O to communicate with external devices such as PLDs it is important that we have a model in SPARK for how this communication occurs.

4.3.4 Static analysis and provability

SPARK is designed to perform static analysis as defined in Section 2.2.2. Using the Examiner for information flow analysis picks up not only common errors such as use of uninitialised variables, infinite loops and potential aliasing, but also reveals quite detailed information about the structure of the program in terms of data coupling between packages.

The user can also choose to employ more detailed methods for selected procedures. The Examiner contains a Verification Condition (VC) Generator that can be used to attempt to prove correct a subprogram in terms of the pre- and post-condition model on which Z is based, and which we will use in Chapter 5. Using the run-time exception and overflow checks option, discussed above, also enables the user to show absence of run-time exceptions.

From a given subprogram, a set of Verification Conditions (VCs) is generated for each path through the subprogram. The VC set for a given path consists of a list of hypotheses which are true for that path, and one or more conclusions which need to be deduced from the hypotheses for the path to be well-formed.

The extra complexity of these options arises because the Examiner itself simply generates files describing the semantics of the subprograms concerned, along with the user's requests (e.g. that no variable goes outside its type range.) Use of two other tools is then required. The SPADE Simplifier[Pra95] processes these files to eliminate irrelevant and redundant information, and performs some automatic simplification of hypotheses and conclusions. It is possible that these simplifications will be sufficient to discharge the VCs. If not, the user may either to prove the remaining assertions by hand or use the SPADE Proof Checker[Pra98].

In Chapter 7 we generate run-time exception checks with overflow for a substantial SPARK program to demonstrate that it is a practical technique for software development.

4.3.5 Summary of SPARK

For the purpose of this work, SPARK Ada's strengths as a language for hardware / software co-design of safety-critical systems are in its formal definition, the information it provides about variable data types and flow, compatibility with industry-strength validated compilers and the existence of tools to support detailed analysis and proof of programs written in SPARK Ada.

Its main weaknesses are the gaps in its formal semantics and omission of some useful Ada constructs which would be amenable to analysis e.g. simple generic package declaration and instantiation. However, despite these weaknesses the language is fundamentally strong enough and well-defined enough for us to use and reason about its behaviour.

4.3.6 SPARK interfaces

Now that we understand the main properties of SPARK, we examine how to interface SPARK to programmable logic. The architecture that we are assuming for the system discussed in the remainder of this section is a conventional microprocessor and memory on a bus, executing a compiled SPARK program, with a PLD also interfaced to the bus.

Suppose that we have a set of operations, and maybe some state, that are held within a PLD, to be controlled by a SPARK Ada program. The rest of the system is intended to run in software on the microprocessor. We will now consider how to interface between the PLD and the Ada software. This section aims to establish that Ada programs can communicate with PLDs and be annotated in such a way that the SPARK Examiner accepts the Ada code and correctly models the actual information flow in this interface.

Memory-mapped I/O

We will need to be able to access the input and output pins of a PLD from Ada. As explained in Section 4.3.3, memory-mapped I/O can be set up so that, for instance, one page of addressable memory is mapped to the PLD input and output pins, via the memory management hardware of the system, and variable *X* (respectively *Y*) is mapped to the input (respectively output) pin area of the page. Assigning a value to *X* will effectively input to the PLD pins; reading from *Y* will effectively read from the output pins.

The variables *X* and *Y* will be state variables of some package *P*, so according to SPARK rules *X* and *Y* must be declared as `--# own` variables of *P*. However, since *X* and *Y* are memory-mapped then the developer must specify whether they are mapped as an input (mode `in`) or output (mode `out`) in order that the Examiner not complain that the variable is never assigned to (for mode `in`) or never read (for mode `out`).

Library interfaces

An alternative is to control writing to and reading from the PLD with a software library which is not written in Ada; *C* is a common choice by device or COTS operating system vendors. However, there needs to be some interface at the Ada level. To do this, the Ada language requires the developer to provide a package body incorporating subprogram declarations marked by a `pragma Interface` statement, denoting a library interface call. Ada calls to these subprograms are translated by the compiler to calls to the library subroutines.

The package specification will declare SPARK-compliant subprograms that wrap each interfaced routine. SPARK requires this package specification so that it can perform an analysis of the program where calls to this package are made; the developer

is therefore required to add SPARK annotations that represent the actions of the library for each call. It is usual to give the package specification a single `--# own` (state) variable representing the state of the logic device, and have the state change at each operation. The package body is typically excluded from SPARK analysis since local types may need to be declared that are not SPARK-compliant.

It is important for the correct information flow analysis of the rest of the program that the developer's annotations be a faithful representation of the PLD's operations. For example, if the PLD's state changes as the result of an operation `K`, but the annotation for `K` does not reveal this state change, then any safety or security arguments which rely on the PLD not changing state between two points cannot usefully appeal to the information flow analysis done by the Examiner; all the possible paths between the points would have to be checked for calls to `K`.

4.3.7 Partial compilation

It is conceivable that a developer would have an existing SPARK program which runs entirely in software, and wish to compile some of it into programmable logic. This could occur if:

1. the software as it stands cannot meet performance requirements;
2. the PLD hardware is planned to arrive late in the project schedule and the program must be unit- and system-tested before it arrives; or
3. an emerging system hazard has indicated the need to move some functionality out of the program's direct address space (e.g. a safety monitor).

Assume that the software to be compiled is some package `P` of the program. How should we go about this?

First, we should establish that the software to compile is true SPARK; this is easily done by running the Examiner on `P`'s specification, body and subprograms. Second, we should show that the software is free from run-time exceptions, by generating VCs with the Examiner and proving them via the Simplifier and Proof Checker or manual proof review. At this point we should consider whether adding proof statements to some of `P`'s subprograms would be helpful to the compiler; if so, these will need to be proven as well.

Next, we need to consider whether we wish to make the use of a PLD explicit in the program. If we do, we can use either the library interface package scheme to make PLD library calls, or write directly to registers with an MMIO scheme, replacing existing code in subprograms. We must then change our annotations to reflect the new state variables and rerun the Examiner on the subprograms.

The disadvantage of these approaches is that any new state or subprogram information flow changes will "bubble up" through the program, causing any package depending on our compiled package to change its annotations. This is tedious, especially since operations pushed out to programmable logic tend to be at the leaves of the program calling tree, and so much of the program may be affected.

Better would be to leave the original package annotations intact. But how can we be sure that they are accurate? This will depend on the reliability of the compile transformation.

If we can ensure that the compiled PLD code and the original SPARK are refinements of the same original specification, this gives us the advantage of being able to develop and test the software independent of the hardware, removing a dependency tie from the system development plan. Certainly there will eventually have to be tests to check that the PLD program integrates properly with the software with particular attention paid to timing issues, but these can be run quite late in the development process since timing-related changes should be localised in the program and not change the results of much of the unit, system and functional coverage tests.

With this in mind, we now look at how to partition a SPARK program into hardware and software components.

4.3.8 Partitioning

A SPARK program provides significantly more information relevant to partitioning than an Ada program. For each subprogram we know exactly the variables which it requires as imports and exports, the numeric ranges of these variables, and we can even add extra constraints on imported variable values and show whether they are satisfied at every point in the program where the subroutine is called.

Information flow annotations additionally describe how the subprogram imports depend on the exports, which may give us a starting point for a decomposition of the subprogram.

If increasing (or, indeed, maintaining) overall program execution speed is important, we must establish that the increased calculation speed provided by the PLD offsets the cost of I/O between software and PLD; the imported variables are copied to the memory-map inputs, then the program waits for the output values to be flagged as ready and copies them back to the exported variables. Therefore a selected subprogram should have a software execution time significantly greater than this two-way copy and transmit operation.

The bit width of imports and exports should be calculated, and “narrow” subprograms be favoured over “wide” ones. The developer should bear in mind the bandwidth and routing problems that affect most PLDs.

Finally, we should aim to encapsulate changes. Therefore, if the PLD-migrated subprogram *S* calls subprogram *T*, then both *S* and *T* need to go into hardware; if *T* is not called from any other part of the software then all the better, since it will effectively become an embedded subprogram of *S*. Essentially, we are aiming to create a package with the minimum of public subprograms where a compilation closure of a subset of the package body is in hardware.

4.3.9 Compilation - a first cut

Suppose we have selected subprogram *S* to be compiled into hardware, with imports i_1, \dots, i_m and exports j_1, \dots, j_n . We shall ignore the case where a variable is both imported and exported since the input and output pins are physically separate on the PLD and so there is no issue with the newly calculated PLD outputs interfering with the original PLD inputs. For each variable we have a known data range, which we will translate into a bit width. At the moment we will assume that all these widths are

small as this allows us to assume simple bit-parallel communication of variable data which completes in one clock tick.

The information flow annotations of \mathbf{S} describe variable dependency. For each export we know exactly which imports it depends on. We can therefore produce a design where each export is the single output of a block, whose inputs are the imports that the export depends on.

The subprogram that computes each export can be derived from the original subprogram as follows:

1. delete all imports that do not affect our selected export, and all exports apart from the selected one;
2. delete every statement in the subprogram that uses any deleted import, or assigns to any export other than the one we want;
3. rerun the SPARK Examiner, and delete all the assignments which it reports as ineffective;
4. if any ineffective assignments were reported, go back to step 1.

This can be shown to be semantically equivalent to the original by arguing that:

1. the Examiner correctly identifies the information flow in any subprogram;
2. we create a subprogram for every export, and therefore our argument reduces to showing that the algorithm works for any given export;
3. there is a finite number of assignments in the subprogram and therefore our algorithm terminates;
4. in any statement except a procedure call with more than one export, all imports of that statement affect the statement export;
5. we have already recursively applied this algorithm down the subprogram tree to such change procedure calls to sequential calls to reduced procedures computing single exports; and
6. if there were a statement which affected our export and which we had deleted, it must have either used a deleted import (in which case the import must have affected our export and hence could not have been deleted) or been reported as ineffective (in which case it could not have affected our export at all).

Now we are left with a subprogram that computes one export. How do we compile it to a form suitable for execution in a PLD?

4.3.10 Compilation of SPARK code

We examine the general problem of mapping SPARK code from inside a subprogram directly onto a typical PLD. We do not consider the specific (and substantial) problems involved in producing a safety-critical PLD implementation, e.g. making the transformation suitable for arguments about preservation of program semantics.

We examine three possible paths from SPARK to PLD:

1. to develop, for each SPARK construct, a bespoke PLD “interpretation” which can be composed together;
2. to formally transform source code to PLD through formal refinement, based on the previously-provided semantics; or
3. the development of a SPARK “interpreter” on a PLD.

The first is the hardest to implement, it being difficult to show that the transformations induced are sound with respect to our semantics. For illustration of these difficulties, we describe the transformations envisaged as necessary, isolating the parts that would introduce real difficulties.

The second leaves the developer with work to do every time that the refinement is needed. The benefits are that the semantics that justifies the transformation already exists, and it can work at various levels of criticality – from a handwaving justification that a predicate is true through to a 10-page proof that a given refinement step is valid.

The third has the benefit that, once the transformation is proven correct, its subsequent use produces valid hardware that is suitable for safety-critical use whenever the original SPARK code was suitable. Of course, as Stepney has shown[Ste98] the steps involved in high-integrity transformation are difficult to get right. It is unlikely that such a PLD-based interpreter could be certified as appropriate for the higher levels of integrity. We do not attempt to produce these transformations in this work.

We begin with the first option, the development of PLD representations of each SPARK language construct.

Syntax

Sequential SPARK subprogram body code consists of a sequence of the following classes of code:

- assignment of an expression
- for loop
- if - elsif - else - end if block
- while loop
- simple loop
- procedure call
- case block

There are two forms of in-statement evaluation: an expression (as found on the RHS of an assignment) and a condition (as found following `if` or `elsif`). Note that, unlike C or full Ada, conditions and expressions may not have side effects; they change no variables themselves. Expressions and conditions may involve calls to functions but these functions do not have side effects.

Sequential composition

For each item in the sequence, the SPARK flow analyser will tell us its imports and exports. Any subprogram local variables are included in the flow analysis, and eventually removed for the purpose of calculating the whole subprogram flow analysis. Iterative constructs such as `while` loops have their information flow calculated using the algorithm described by Barnes[Bar03] §10.8. If we can produce a block for each sequence item, we can connect inputs and outputs in the appropriate sequence to produce a full computation.

Note that some items in the sequence may produce an output that is not needed by their successor. In that case the output can be connected directly to the first successor that needs it. If consecutive items P,Q are such that no export of P is an import of Q then P and Q can be placed in parallel. They must, however, be synchronised in some way so that the computations that follow will process P and Q only when both are ready.

To manage this, and the more general issue of “computation complete” for the subprogram we implement a simple protocol with input and output control bits. Each hierarchical block B in the program has one input and one output bit, with each output bit connected to the inputs of one or more other blocks that use the data from B. At program start each input bit is low and each output bit is low.

When the PLD receives data from the SPARK program, the input bit for the entire subprogram block will be set high to signal valid input data. The PLD computation then starts, with the high input bit travelling across the PLD to track the computation progress. When each block’s computation is complete the output bit is set high and the block waits for the input bit to go low. The blocks to which the output bit is routed will then copy over the block’s output data and signal back that this has happened; once all child blocks have signalled back, the block pulls its output bit back to low and is left waiting for its input to go high again. The entire subprogram block will eventually have its output bit go high, at which point it writes data back to the SPARK program.

Figure 4.4 shows an example of data being passed from block A to block B to block C, with the computation complete signal travelling the same path later on.

Code constructs

SPARK assignment will be represented in the PLD by a set of lookup tables which compute the RHS expression in stages. This is not hard unless a function forms part of the expression; in this case we will have to produce a block for that function and wire it into the computation.

A `for` loop provides a loop variable which its enclosed block takes as an additional input. Short loops with static iteration ranges could be unrolled altogether; however, in the general case it would be necessary for the loop’s block to route its outputs back to its inputs, and to have control logic that raises a flag once the computation is complete.

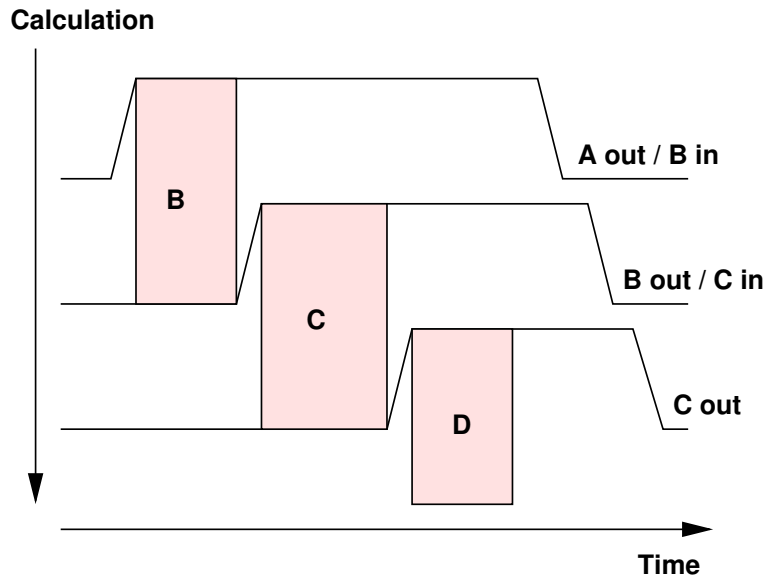


Figure 4.4: Handshaking across blocks

Conditionals such as `if` and `case` blocks have code blocks which are placed in parallel, and a multiplexer which selects inputs depending on the conditional statements. Note that each block in these statements must have the same exports, so must import any exports which they don't change.

`while` loops and simple loops work like `for` loops but without the loop variable. Any use of the `exit` statement will set the "output valid" control, as will the main loop test for the `while` condition. The SPARK restrictions on control flow (following a semi-structured flow graph) help in this respect as the exit points are always on the outermost part of the calculation.

Subprogram (`procedure`) calls are inlined by inserting the block representing that subprogram. The enforced ban on circular or recursive subprogram calls ensures that the inlining will eventually terminate at a set of "leaf" subprograms that do not contain any further subprogram calls. Note that this method would be inefficient in space usage if a particular subprogram was called at several points within the compiled program.

Packages with state

Suppose a package has internal state, invisible to other packages by Ada rules. This state will be stored in the PLD, so will change the aforementioned layout by adding a RAM block to store the state, routing the RAM output into the sequence items like a normal import, and, in the case of a write, routing the exported data back to the RAM store with a write bit set.

Bit serial versus bit parallel

All the above has assumed that we are working in bit parallel form. However, there are many cases where input data may be very wide, for instance in the case of an array with a wide range or a record with many fields. Passing this into the PLD in bit parallel form would quickly use up routing resource, especially if the entire variable is routed between several statement items. Is there an alternative?

For records, it is not hard to slim down the data. The Examiner does flow analysis of subprograms at the record component level, so although the entire variable is imported the Examiner knows which fields will be imported and exported at each stage. It is a relatively simple matter to treat the record as a list of distinct variables.

Arrays are more difficult. Array indexing is, in general, dynamic and hence not susceptible to static analysis. In the worst case it is computationally infeasible to determine which array elements may be used at a given stage of computation. However, there are optimisations which may be used in some cases at the possible expense of the clarity of correspondence between the PLD and SPARK representations.

Often, entire arrays (or subranges of them) are changed with a `for` loop. If a relatively small subrange is used, the Examiner would be able to check that any reference to an array element is made with an index with a given subrange, reducing the amount of array data that needs to be exported or imported. This would require a modification to the Examiner to maintain a “defined” flag bit for each element of any non-imported array with a range below a set limit.

Alternatively, we could find that the only references to an array are within a `for` loop, with array indices corresponding to a 1-1 function of the loop variable (and of no other variables). As long as the RHS of any assignment to the array is not dependent directly or indirectly on the loop variable, the entire function can be replicated any number of times to calculate the array value over arbitrary subranges. In addition, the SPARK code could supply the subrange parameters and so use a number of calls to the hardware to compute the entire array change in sections. This gives the developer an ideal opportunity to trade execution speed against PLD area.

These techniques are intended as an example of the trade-offs that can be made in compilation. They show how the extra information obtained by the SPARK Examiner can be used to have confidence that such optimisations preserve the correctness of the code.

Justification of equivalence

The dynamic semantics of SPARK Ada[Ltd94b] are defined for each construct in terms of modifications to a collection of variable state information. In order to reason about the correctness of transformations into PLD form we need to be able to relate the semantics of a SPARK statement P to the semantics of a PLD block Q which is intended to represent P .

We must define the semantics of the PLD block Q in terms of its transformations of data between its input control bit being set high and the block setting its output control bit high. Our SPARK-to-PLD transformation has defined some functions $QI, QO : V \times \mathbb{N} \rightarrow \mathbb{P}W$ from the *legal* values of each imported (respectively, exported) SPARK variable from the variable set $V = V_I \cup V_O$ to appropriate representations of the data by high voltages on a combination of wires $\subseteq W$ going in to (respectively, coming out of) the block. We represent the distinct values of a variable by natural numbers; that this is adequate follows from an argument appealing to the behaviour of a correct compiler which must represent each value of any variable by a bit pattern within a fixed-length field in memory. The inverse functions QI^{-1}, QO^{-1} describe the variable values represented by a given combination of wire high-voltage states.

Any given statement in the SPARK program P updates the variable store σ to

represent its action on variable values. The simple assignment of an expression ev to a local variable $fullname$, for instance, is expressed by a deduction rule $AsgnD1$ (on page 109 of [Ltd94b]) which updates σ by:

$$\sigma \oplus \{fullname \mapsto ev\}$$

The corresponding definition on \mathbb{Q} will be in terms of the traces of the SRPT process representing \mathbb{Q} . If ci is the input control bit and co is the output control bit then an equivalent statement for the assignment block A in \mathbb{Q} would be:

$$\begin{aligned} \forall t \in \mathcal{T}_{\mathcal{R}}[[A]]\sigma \cdot \forall i > 0 \cdot \\ (ci \notin t[i] \wedge ci \in t[i+1]) \Rightarrow (\exists k > 0 : co \in t[i+1+k]) \\ \wedge QO^{-1}(fullname, t[i+1+k]) = \\ ev(QI^{-1}(t[i+1])) \end{aligned}$$

Clearly, the semantic mapping outlined above would have to be expanded and formalised if this hierarchical translation method was to be developed formally. The weakest precondition semantics of each SPARK construct would have to be refined by the PLD implementation.

4.3.11 Refinement

A second approach is to produce a formal specification of the function performed by a SPARK subprogram, and refine this to a custom implementation in hardware. This throws away the SPARK implementation, taking advantage of the parallel computational model presented by the PLD. How do we ensure that the SPARK implementation is therefore equivalent?

SPARK enables the developer to specify pre- and post- conditions for subprograms, and prove the correctness of postconditions given preconditions by generating and proving verification conditions. Therefore we can have confidence that our SPARK implementation does what is specified. Alternative approaches are model-checking and *animation*, both of which are used by the PROB tool which supports programs written in the B language [Abr96].

The implementation difficulty is going to be showing that our custom implementation satisfies the VCs as well. This is something we address in Chapter 5. The separate difficulty of providing an accurate specification is a well-known software engineering problem [DvLF93, Vic98, HRH01] which lies outside the scope of this thesis.

4.3.12 SPARK interpreter

The third alternative to the approach of transforming an isolated package into PLD form is to produce a SPARK “interpreter” that runs on an PLD. Such an interpreter would be able to operate on any number of SPARK packages, running a computationally intensive program without any need to synchronise control with conventional SPARK code. It would also have the advantage that its operation need only be proven correct once; any SPARK program would be represented as data within it.

In Chapter 6 we describe one possible interpreter, with a number of customisable parameters. Different designs are certainly possible; this is only one example.

We do not attempt to reason in any way about the correctness of this particular design. An analytic proof (such as would be required by standards such as Defence Standard 00-54[MoD99] for system functions at SIL 3 or SIL 4) would be much more difficult than that for the refinement or hierarchical implementation approaches described above, since the ability to map between relatively small SPARK and PLD constructs would be lost; the proof would not be that a particular program was executed correctly, but rather than *any* valid SPARK program was executed correctly.

Conventional Ada 95 compilers are validated against the ISO standard ISO/IEC-18009:1999[cJ99] using the publicly-available test suite “ACATS” which contains over 3600 programs. At the minimum, validation of a SPARK interpreter would have to include running each SPARK-compliant ACATS program and verification of the results. This may be adequate to qualify the use of the interpreter for system functions of limited criticality, although each project using the interpreter would have to justify its use in the project safety case.

High integrity Ada compilers such as GNAT Pro High-Integrity (Ada Core Technologies) and Object Ada (Aonix) go through additional verification activities and provide documentation of these activities to end-users; for safety-critical implementations they use restricted subsets of Ada 95, such as GNAT NO RunTime (GNORT), C-SMART and RAVEN. The verification for a SPARK interpreter at high levels of integrity would include at minimum the proof of key interpreter properties (liveness, preservation of data ordering, freedom from race conditions), but the list of verification activities required for a particular safety integrity level and application domain would emerge from a detailed safety assessment.

4.3.13 Summary

In this section we have described the SPARK Ada 95 subset, shown how its properties are helpful in the task of compiling it into a form suitable for execution on a PLD, and described two possible compilation forms as well as more general considerations for the SPARK-PLD interface.

Of the targets in Chapter 3 we have addressed or partially addressed:

Target 2: *The process must help the developer to write unambiguous programs.*

We are programming in SPARK Ada 95, an annotated Ada subset with compiler-independent semantics.

Target 3: *The process must allow the programs to have sections written in a low-level language for speed and flexibility, but not allow these sections to compromise overall program reliability.*

SPARK shadows and `hide` annotations allow the insertion of arbitrary Ada code, which may include assembly language.

Target 4: *The process must admit substantial static analysis to discover semantic program errors at or before compile time.*

The SPARK subset is enforced by the SPARK Examiner, which also performs information- and data- flow analysis to verify the program against design information.

Target 6: *The program must be able to be compiled onto a range of existing and anticipated PLDs.*

We have made no assumptions about the target PLD other than that it is large enough to contain the SPARK program (or interpreter) being transformed.

Target 7: *The process must reuse existing proven tools where feasible.*

The SPARK Examiner tool already exists, and we have noted where it may be extended in small ways to support transformation activities. The information held by the tool after the analysis phase strongly supports PLD-targeted transformation activities.

Target 10: *The process should provide flexibility so that it may be used in situations not anticipated in its original design.*

We have presented three approaches to transforming SPARK programs, aimed at code of differing integrity levels.

Target 11: *The process must admit justification to the project safety authority that the programs output by the process are of an adequate integrity level.*

We have shown in Section 4.3.10 how the hierarchical transformation process might be validated against the existing semantics for SPARK, and how the refinement approach changes the validation required to the proof that an SRPT process refines a specification.

Chapter 5 will demonstrate how to produce a custom PLD implementation from a formal subprogram specification, allowing us to produce SPARK and PLD implementations which are formally equivalent but markedly different in form. This supports the second approach discussed in Section 4.3.11, of transformation-by-refinement.

Chapter 6 will break down this section's overview of a SPARK interpreter into a detailed implementation, showing how the conflicts discussed in Section 4.3.12 are resolved and aiming for demonstrable reliability.

The case study in Chapter 7 will demonstrate construction of an example safety-critical system in SPARK Ada and mapping part of it into a PLD while preserving its functionality.

Chapter 5

Refining To SRPT

Refinement is one of the building blocks of formal methods. It is a way of going from a relatively abstract statement of a problem to a system which can be built with no further intelligent, human involvement, and which can be shown mathematically to solve the problem stated. Much research has established formal refinement as of appropriate rigour for safety critical systems development ([ORS96] is a pre-eminent example, distinguished by its completeness).

In this chapter we describe a formal refinement calculus for high-integrity software running on a PLD. Through the refinement calculus, we will be able to address the concerns of rigour.

5.1 The Refinement Model

There are many approaches to refinement; for instance, see Back [BvW94] and Morgan [Mor94]. Of particular relevance to our approach in being based on reactive action systems is the refinement of Back. There, refinement is defined in terms of traces. We follow a broadly similar form in our semantics, although the deterministic nature of our SRPT subset means that we avoid some of the complications encountered by Back.

Action systems describe the behaviour of a parallel system in terms of the atomic actions that can take place during the execution of the system. Back's approach to trace refinement uses simulations between action systems to construct an abstract behaviour that approximates a given concrete behaviour. By contrast, the deterministic SRPT subset that we use allows us to refine traces directly.

The syntax of our abstract specification is similar to that used by Morgan. This describes a system:

$$w : [\mathbf{pre}, \mathbf{post}]$$

where w is a set of free (changeable) variables in the system, \mathbf{pre} is a predicate specifying the precondition on states that can be assumed for the system, and \mathbf{post} is the predicate on w which the program produced by the system must satisfy.

This model is based on the predicate calculus. The pre-conditions and post-conditions are predicate calculus formulae. The conditions define a *contract* for a program to fulfil, as described by Morgan. We now give an overview of the refinement process in Morgan's model as an example of what we are aiming to achieve.

5.1.1 Overview of a refinement process

Within Morgan’s model, each system being developed is *refined* through a series of well-defined transformations based on proven sound refinement laws to a program expressed in a simple machine-independent language. The language used by Morgan as “code” (the executable form of a program) is a language of guarded commands, which has alternation, iteration and subprogram call control structures similar to those found in most modern imperative programming languages. Commands are composed sequentially within subprograms. This language is augmented with Morgan’s program specification syntax to express parts of the program which have not yet been developed to code. The semantic basis of the refinement is Dijkstra’s weakest precondition calculus [Dij75].

The theoretical basis of refinement

Refinement itself occurs in a system defined by pointwise extension of a *partially ordered set* (“poset”) which itself is equivalent to a *lattice*. The poset comprises a set L of elements (predicates) and a binary ordering operator (the *partial order*) for elements of L denoted \leq . Partially ordered sets are described in more detail by Miller and Dushnik[DM41].

The programs in Morgan’s model are *predicate transformers*, transforming predicates according to weakest precondition semantics. Given a program $P = w : [\mathbf{pre}, \mathbf{post}]$ and a predicate q , if $q \Rightarrow \mathbf{pre}$ then $P(q) = q'$ where q' is q transformed by \mathbf{post} according to weakest precondition semantics. The refinement relation \sqsubseteq between programs corresponds to the ordering of the predicates on which they are based. More detail is given by Back[BvW94].

The symbol \equiv in the context of refinement means “refines in both directions”. If $X \equiv Y$ then $X \sqsubseteq Y$ and $Y \sqsubseteq X$.

Example of refinement

In Morgan’s system, $X[w \setminus E]$ denotes the simultaneous substitution of E for each instance of w in expression X . Law 1.3 (p.9) states that if

$$\mathbf{pre} \Rightarrow \mathbf{post} [w \setminus E]$$

then

$$w, x : [\mathbf{pre}, \mathbf{post}] \sqsubseteq w := E$$

where \sqsubseteq is read “refines to” and $:=$ denotes the assignment operation in the language of guarded commands. The variable x is unaffected by the simultaneous substitution of E and in fact vanishes after the refinement; since w and x are independent, an intuitive interpretation of this is that the true or false value of \mathbf{post} was unaffected by x .

According to this law, the program statement $w := 5$ is a refinement of the specification

$$w : [\mathbf{true}, w = 5 \vee w = 6]$$

since $\mathbf{true} \Rightarrow (5 = 5) \vee (5 = 6)$.

Other code constructors include alternation, sequential composition iteration and procedures, and there exist laws for introducing these from certain specifications.

Pathological specifications

Some specifications cannot be refined to code, and are termed “infeasible”. Other specifications can be satisfied by almost any code. Pathological examples of these forms of specification include:

$$\begin{aligned} w : [\mathbf{false}, \mathbf{true}] & \text{ “abort”} \\ w : [\mathbf{true}, \mathbf{true}] & \text{ “choose } w\text{”} \\ w : [\mathbf{true}, \mathbf{false}] & \text{ “magic”} \end{aligned}$$

abort is never guaranteed to terminate and may do anything to its variables. **choose** w terminates and changes w to an arbitrary value. The program statement **skip** is a special case of **choose** where no variable w is supplied. **magic** always terminates and establishes the impossible condition **false**; no program can satisfy this specification.

Retrenchment

There also is an issue of feasibility regarding the types of variables permitted. For instance, assignments involving set operations are permitted, though conventional imperative languages do not implement such operations natively. Exact arithmetic with irrational numbers is also allowed, in contrast to the imprecise floating point arithmetic model used in common imperative languages such as C, Perl and Ada.

This problem is a known issue in the development of software for high-integrity systems. A common solution is to specify real-number calculations using error bounds (often denoted ϵ) so that a specification of an implementation F of a real-number calculation might be:

$$| F(x, y) - (e^{2x-y} + 3y^2) | < \epsilon$$

This may be an acceptable approach for individual equations, but for a system which depends on sequential real-number calculations this approach can quickly make specifications hard to read accurately.

Large-scale formal reasoning about moving from exact to imprecise calculations may require the use of *retrenchment*[BP98]. This is in many ways the opposite approach to refinement, allowing strengthening of the specification precondition and weakening of the precondition to reason about the program correctness in the context of loss of accuracy in the data type transformation. Since PLDs are often used for numeric calculations, retrenchment or related techniques may prove useful when specifying and refining programs to run on them.

5.1.2 Suitability of model

Morgan’s refinement model starts with a specification at an arbitrary level of abstraction, and allows step-by-step refinement of that specification to a program form which is executable. The developer needs to define the program statements which he regards as directly executable. Each refinement step is done according to a law in the refinement calculus, and may be independently verified by presentation of the specification before and after refinement and a statement of the refinement law that was applied.

Characteristic	Morgan	SRPT
Specification domain	Predicates	Timed predicates
Language	Guarded imperative	Processes
Data flow forms	Serial, subprogram	Serial, parallel
Calculations at:	Assignment :=	Primitive blocks
State model	Variable-value function	Events in traces
Implementation	Ada, C, Pascal	Pebble

Table 5.1: Contrast of Morgan and SRPT refinement processes

We noted in Section 4.3 that a similar pre-post specification notation is used in the SPARK Ada language proof tools. We presented three main options for developing a SPARK Ada subprogram into a PLD implementation, and one of them was to rely solely on the subprogram specification. Since Morgan’s refinement model (and hence the SRPT model that we will develop later in this chapter) only requires a specification in [**pre** , **post**] form, we have sufficient information to start refinement of the subprogram.

The refinement process we wish to use will start with a specification at the level of process events (corresponding to voltage highs on the input wires to a PLD) and be refined to a set of SRPT processes. Section 4.2 has described a systematic, if not yet rigorous, method to translate SRPT into an equivalent Pebble program and hence compile it into a PLD. Table 5.1 contrasts Morgan’s refinement process with the refinement process we desire.

The approach that refinement provides is therefore appropriate to our needs. Morgan’s specification notation matches with the specification notation that SPARK subprograms use. However, because of the differences between the semantic bases of Morgan and our trace-based approach we will consider a modified version of Back’s refinement process.

5.2 Refinement for SRPT

5.2.1 Aims for refinement

With our system derived from the above models we aim to replace the notion of an imperative program as a final result to a process expressed in Barnes’ Synchronous Receptive Process Theory. Specifications may also be expressed in conjunction with a non-negative integer time at which they are true.

The building blocks of our new system, i.e. the components corresponding to assignment statements in Table 5.1, will be processes describing logic constructs similar to FPGA cells. For the moment these cells shall be stateless, and their outputs at time $t + 1$ shall be purely functions of their inputs at time t .

5.2.2 Refinement frames

A *refinement frame* is a new construct which we will incorporate into the SRPT notation, allowing us to express parts of an SRPT system in specification form. A refinement

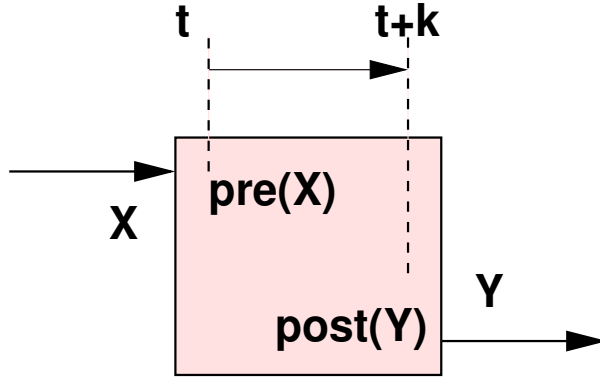


Figure 5.1: SRPT frame structure

frame (shortly, “frame”) P in a program takes the form:

$$P = \forall t \in \mathbb{N} \cdot \iota X : oY : [[\mathbf{pre}]_t, [\mathbf{post}]_{t+k}] \quad (5.1)$$

representing the specification “for the process P with input alphabet containing X and output alphabet containing Y , at all times t , if \mathbf{pre} is true at time t then at time $t + k$ \mathbf{post} is true.” k is a constant which will be determined by the timing needs of the program at specification time.

Figure 5.1 illustrates frame P as an SRPT process.

Back[BvW94] does not use these refinement frames; instead, the start and points for refinement are action systems operating on state spaces; refinement moves from abstract state spaces to concrete ones with the individual actions of the systems changing as required to handle the decreasing abstraction of the state. An action system refinement can be regarded as complete when its state space is sufficiently concrete to be implemented on whatever computing system is available.

Process semantics

If a frame is to represent an SRPT process, as do the other components in the SRPT algebra, it must have a set of traces obeying the SRPT trace axioms discussed in Section 4.1.5. Concerning the underlying SRPT process P , the frame in Equation 5.1 specifies that:

$$\forall s \in \mathcal{T}_{\mathcal{R}}[[P]]\sigma \forall t \in \mathbb{N} \cdot \mathbf{pre}(s[t..]) \Rightarrow \mathbf{post}(s[t..])$$

i.e. that in every trace of P the frame’s postcondition holds at all points where the precondition holds.

$\mathbf{pre}(s)$ is a shorthand for a substitution; the timed event predicate \mathbf{pre} can be seen as a Boolean function of subsets of timed event occurrences $\mathbf{pre} : \mathbb{P}(\Sigma \times \mathbb{N}) \rightarrow \mathbb{B}$. Since the trace s is a sequence of time steps at which each event in the alphabets of P either occurs or does not occur, it defines a similar function $s_R : (\iota P \cap oP) \times \mathbb{N} \rightarrow \mathbb{B}$. Therefore $\mathbf{pre}(s)$ is equivalent to:

$$\forall Z \subseteq \mathbb{P}(\Sigma \times \mathbb{N}) \cdot \mathbf{pre}(Z) \Rightarrow ((z, t) \in Z \Leftrightarrow s_R(z, t))$$

The SRPT trace axioms require that for the refinement frame in Equation 5.1:

1. the empty (zero-length) trace is in $\mathcal{T}_{\mathcal{R}}[[P]]\sigma$;
2. $\mathcal{T}_{\mathcal{R}}[[P]]\sigma$ is prefix-closed; and
3. any input events may be offered at any step, and the output events at that step must be independent of those input events.

Axiom 1 follows since the quantification of t is over a null set. Axiom 2 follows because the quantification of t is unbounded, so if s_1 is a prefix of $s_2 \in \mathcal{T}_{\mathcal{R}}[[P]]\sigma$ then the specification must hold for all of s_1 . The justification of Axiom 3 is more lengthy, and is given in Section 5.2.3 below.

Notation

In the frame P , the presence of an event x at time t is depicted by $[x]_t$. This value corresponds to the presence or absence of x at *time index* t in a trace of P . We also introduce the shorthand $[f(x, y) = c]_t$ for $f([x]_t, [y]_t) = [c]_t$ where f is a constant function within a predicate.

t and k are necessary because an SRPT process computes in a “pipelined” (systolic or overlapping) manner; t marks a point where a computation starts and k expresses the length of the pipeline which produces the result. The $\forall t \in \mathbb{N}$ is usually omitted for brevity.

Where variables are involved in arithmetic expressions the values **true** and **false** are taken to correspond to the integers 1 and 0 respectively.

Purpose of a specification

As described above, the specification described by a frame defines a set of traces and so can be considered an SRPT process (if an abstract one!).

The aim of the refinement is to synthesise a concrete SRPT process that has traces that are “the same or better” than the specification. As we will see below, this translates to a subset ordering on the set of traces.

Rules of a specification

We define the following rules for the frame contents in order to exclude some infeasible specifications. The phrase “ X related to Y ” in a predicate refers to the situation where the truth of the predicate depends on a logical relation between variables X and Y . In all of the following, x is taken to be a single event in the input event set ιX and y is a single event in the output event set oY .

1. Predicate **pre** may only refer to variables in the input event set X .
2. the postcondition **post** may only refer to variables in the input event set X and output event set Y .
3. the highest time index t of any variable in **pre** must be less than the lowest time index of any output variable (from Y) in **post**.
4. where variables $[x]_{t+i}$ and $[y]_{t+j}$ are related in **post**, $i < j$.

Rules 3 and 4 are “anti-oracle” rules, excluding specifications that cannot be implemented by an SRPT process since they would have traces that violated the “delayed reaction to input” SRPT trace axiom.

The purpose of Rule 3 is to restrict the production of preconditions requiring knowledge of the future, e.g.

$$\forall t \in \mathbb{N} \cdot \iota X : oY : [[x]_{t+1}, [y]_t]$$

where the program clearly has no way of knowing what $[x]_{t+1}$ will be, so the obvious action for the developer in this case is to weaken the precondition to **true** (a valid refinement as we will see in Section 5.2.5).

The purpose of Rule 4 is to restrict the production of infeasible postconditions, e.g.

$$\forall t \in \mathbb{N} \cdot \iota X : oY : [\mathbf{pre}, [x]_t = [y]_t]$$

where the program clearly cannot know $[x]_t$ in time to output $[y]_t$.

Example specification

A 1-cycle AND gate with input events $X = \{x_1, x_2\}$ and output events $Y = \{y\}$ would have refinement frame

$$\iota X : oY : [\mathbf{true}, [x_1 \wedge x_2]_t = [y]_{t+1}]$$

The possible traces (each of which will be a trace prefix) of this process include:

$$\langle \{x_1\}, \{x_1, x_2\}, \{x_2, y\} \rangle, \langle \{x_1, x_2\}, \{x_1, x_2, y\}, \{y\} \rangle \text{ and } \langle \rangle$$

An example of an incorrect trace prefix is $\langle \{x_1\}, \{x_2, y\} \rangle$.

5.2.3 Refinement relation

Definition: For SRPT processes P and Q we say that P is refined by Q whenever $\mathcal{T}_{\mathcal{R}}[[Q]]\sigma \subseteq \mathcal{T}_{\mathcal{R}}[[P]]\sigma$.

Informally, P is refined by Q if any trace of Q is a valid trace of P . Our notion of refinement is a specialisation of that of Back[BvW94] to the case when P and Q are deterministic processes. As noted above, Back uses simulation between action systems whereas SRPT provides a denotational semantics for the traces model.

It may at first appear that a process R with a minimal trace set, consisting (say) of the empty trace $\mathcal{T}_{\mathcal{R}}[[\square]]\sigma$ will refine any other process. However, this is not the case. Because of SRPT trace axiom 3, which requires that any input events may be offered at any step, process R must define output events in response to each possible input event set combination at each time. The only time when a strict subsetting is possible would be when P offers two or more possible responses to a given set of inputs (non-deterministic behaviour).

Whenever P is a valid deterministic SRPT process, P will only ever offer one response to a given set of inputs, so the refinement relation is direct equivalence of trace sets.

Given a specification $S = \iota X : oY : [[\mathbf{pre}]_t, [\mathbf{post}]_{t+k}]$, we define its traces $\mathcal{T}_{\mathcal{R}}[[S]]\sigma$ as:

$$f \in \mathcal{T}_{\mathcal{R}}[[S]]\sigma \Leftrightarrow \forall 0 \leq t < (\#f - k) \cdot [\mathbf{pre}(f)]_t \Rightarrow [\mathbf{post}(f)]_{t+k} \quad (5.2)$$

If we are to refine S into processes then we need to show that $\mathcal{T}_{\mathcal{R}}[[S]]\sigma$ satisfies the SRPT trace axioms. In Section 5.2.2 we demonstrated that Axioms 1 and 2 were met. It remains to show Axiom 3, that at any step the process represented by $\mathcal{T}_{\mathcal{R}}[[S]]\sigma$ can accept any input, and the input cannot affect the output at that step.

To demonstrate that the process represented by $\mathcal{T}_{\mathcal{R}}[[S]]\sigma$ can accept any input at any step without affecting that step, let $f = s \frown \langle Z \rangle \in \mathcal{T}_{\mathcal{R}}[[S]]\sigma$. Then, from Equation 5.2:

$$f \in \mathcal{T}_{\mathcal{R}}[[S]]\sigma \Rightarrow \forall 0 \leq t < (\#f - k) \cdot [\mathbf{pre}(f)]_t \Rightarrow [\mathbf{post}(f)]_{t+k}$$

Now we must show that

$$\begin{aligned} \forall U \subseteq X \cdot r = s \frown \langle V \cup U \rangle &\Rightarrow r \in \mathcal{T}_{\mathcal{R}}[[S]]\sigma \\ \text{where } V &= (Z \cap Y) \end{aligned}$$

because this shows that every process r identical to f except for input events is in $\mathcal{T}_{\mathcal{R}}[[S]]\sigma$.

Since s prefixes f , we know that $s \in \mathcal{T}_{\mathcal{R}}[[S]]\sigma$ from SRPT Axiom 2. We need then only show that:

$$[\mathbf{pre}(r)]_{\#r-(k+1)} \Rightarrow [\mathbf{post}(r)]_{\#r-1}$$

i.e., the pre-post relationship holds for the last element of trace r .

The rules on pre- and post-condition time indices restrict **post** from specifying outputs at t , or from $t + k$ onwards, and similarly restrict **pre** from specifying inputs from $t + k - 1$ onwards. Hence any events in U (at time index $\#r - 1$) cannot affect the precondition. By construction, the output events V do not change from f to r , hence the postcondition is similarly unaffected, and therefore the third closure condition is met.

This allows us to treat process refinement frames as SRPT processes in the following refinement rules.

5.2.4 Refinement

A half-adder could be specified as follows:

$$\forall t \in \mathbb{N} \cdot \iota\{a, b\} : o\{c, s\} : [\mathbf{true}, [2c + s]_{t+1} = [a + b]_t] \quad (5.3)$$

We have already seen in Section 4.1.6 the definition of the SRPT process $CELL_f$ which computes the function f in one step. We make our first refinement law:

Refinement 1 *Stateless 1-bit function*

$$\begin{aligned} &\forall t \in \mathbb{N} \cdot \iota X : o\{y\} : [\mathbf{true}, [y]_{t+1} = f([X]_t)] \\ &\sqsubseteq CELL_f[I \setminus X][O \setminus \{y\}] \end{aligned}$$

This is justified by inspection of traces: the definition of the $[\mathbf{pre}, \mathbf{post}]$ form of refinement frame in Section 5.2.2 defines the traces of this frame S to be:

$$s \in \mathcal{T}_{\mathcal{R}}[[S]]\sigma \Rightarrow \forall t \in \mathbb{N} \cdot (\mathbf{true} \Rightarrow [y]_{t+1} = f([X]_t))$$

which corresponds to the traces of $CELL_f$ with the appropriate event renaming. Variants of $CELL$ are the basic constructors of combinatorial logic as they are a representation of primitive blocks in Pebble.

We could use this to define cells that calculated either c or s in our half-adder, but not both. We need a way of expressing parallelism. This is our second refinement law:

Refinement 2 *Parallelism*

$$\begin{aligned} & \forall t \in \mathbb{N} \cdot \iota X : o(Y \cup Z) : [\mathbf{pre}, \mathbf{post}_1 \wedge \mathbf{post}_2] \\ \sqsubseteq & \quad \iota X : oY : [\mathbf{pre}, \mathbf{post}_1] \parallel \iota X : oZ : [\mathbf{pre}, \mathbf{post}_2] \end{aligned}$$

whenever:

$$\begin{aligned} & Y, Z \text{ are non-empty and non-intersecting} \\ & \forall V \in \mathbb{B}^{\#Z} \cdot \mathbf{post}_1[Z \setminus V] \equiv \mathbf{post}_1 \\ & \forall W \in \mathbb{B}^{\#Y} \cdot \mathbf{post}_2[Y \setminus W] \equiv \mathbf{post}_2 \\ & \text{where } \mathbb{B}^N \text{ is the set of n-ary boolean strings} \end{aligned}$$

Informally, this says that if there are two parts of the output of a process, \mathbf{post}_1 and \mathbf{post}_2 , which have a null intersection of output events then the process can be split into two, each computing one of the parts. Note that it is trivial to extend this refinement to any finite number of parallel components since \parallel is associative according to Law 2 in Barnes[Bar93] §5.1.1.

The justification of this refinement law is again by traces; we show that the trace set of the original frame is equal to the parallel combination of the traces of the two new frames, using the semantics of the \parallel operator from Barnes[Bar93] §5.1.

Returning to our original specification Equation 5.3, we can apply refinement law 2 and the logic arithmetic definition:

$$a + b = 2(a \wedge b) + (a \oplus_2 b)$$

where \oplus_2 denotes addition modulo 2, to produce:

$$\begin{aligned} & \forall t \in \mathbb{N} \cdot \iota\{a, b\} : o\{c, s\} : [\mathbf{true}, [2c + s]_{t+1} = [a + b]_t] \\ \sqsubseteq & \quad \iota\{a, b\} : o\{c\} : [\mathbf{true}, [c]_{t+1} = [a \wedge b]_t] \quad (5.2.4.1) \\ & \parallel \iota\{a, b\} : o\{s\} : [\mathbf{true}, [s]_{t+1} = [a \oplus_2 b]_t] \quad (5.2.4.2) \end{aligned}$$

We apply refinement law 1 to (5.2.4.1), with function **and**, noting that \wedge is equivalent to **and**, to produce:

$$(5.2.4.1) \sqsubseteq CELL_{\mathbf{and}} [I \setminus \{a, b\}][O \setminus \{c\}]$$

and similarly to (5.2.4.2), with function **xor**, noting that \oplus_2 is equivalent to **xor**, to produce:

$$(5.2.4.2) \sqsubseteq CELL_{\mathbf{xor}} [I \setminus \{a, b\}][O \setminus \{s\}]$$

and we have refined our original specification into two parallel 2-input 1-output cells:

$$CELL_{\mathbf{and}} [I \setminus \{a, b\}][O \setminus \{c\}] \parallel CELL_{\mathbf{xor}} [I \setminus \{a, b\}][O \setminus \{s\}]$$

5.2.5 Additional refinement rules

We now introduce supplementary refinement rules. We start with counterparts of laws given by Morgan [Mor94], whose justifications come from predicate calculus and are not given here because our refinement of frames is also expressed in terms of predicate calculus.

Refinement 3 *Weaken precondition*

If $\mathbf{pre} \Rightarrow \mathbf{pre}'$ then:

$$\forall t \in \mathbb{N} \cdot \iota X : oY : [\mathbf{pre}, \mathbf{post}] \sqsubseteq \forall t \in \mathbb{N} \cdot \iota X : oY : [\mathbf{pre}', \mathbf{post}]$$

Refinement 4 *Strengthen postcondition*

If $\mathbf{post}' \Rightarrow \mathbf{post}$ then:

$$\forall t \in \mathbb{N} \cdot \iota X : oY : [\mathbf{pre}, \mathbf{post}] \sqsubseteq \forall t \in \mathbb{N} \cdot \iota X : oY : [\mathbf{pre}, \mathbf{post}']$$

Refinement 5 *Expand frame*

$$\begin{aligned} & \forall t \in \mathbb{N} \cdot \iota X : oY : [\mathbf{pre}, \mathbf{post}] \sqsubseteq \\ & \forall t \in \mathbb{N} \cdot \iota(X \cup A) : o(Y \cup B) : [\mathbf{pre}, \mathbf{post}] \end{aligned}$$

where $A \cap Y = \emptyset$ and $B \cap X = \emptyset$.

Refinement 6 *Contract frame*

Let $P = \iota X : oY : [\mathbf{pre}, \mathbf{post}]$. If:

$$\begin{aligned} & \exists A \subseteq X \cdot \forall s \in \mathcal{T}_{\mathcal{R}}[[P]]\sigma \ \forall B \subseteq A \ \forall t \in \mathbb{N} \cdot \\ & \exists r \in \mathcal{T}_{\mathcal{R}}[[P]]\sigma \cdot (r[t] = (s[t] \setminus A) \cup B) \wedge (\forall i \neq t \cdot r[i] = s[i]) \end{aligned}$$

i.e., we can change the occurrence of A input events at any timestep to some arbitrary subset B without changing any of the subsequent output events (input variables A are irrelevant to the outputs), then:

$$\iota(X \cup A) : oY : [\mathbf{pre}, \mathbf{post}] \sqsubseteq \iota(X \setminus A) : oY : [\mathbf{pre} \setminus A, \mathbf{post} \setminus A]$$

i.e., we can remove the A events. This refinement can be justified by observing that removing the A input events from the precondition will weaken it, and the condition for this refinement means that the output events are unaltered.

Now we introduce rules peculiar to our timed parallel model, along with justifications.

Refinement 7 *Introduce intermediate*

If g, j, k, \mathbf{mid} are timed predicates over subsets of events such that:

$$\begin{aligned} &\forall \text{ disjoint } X, Y, Z \subseteq \Sigma. \\ &g([Y]_{t+2}, [X]_t) \Leftrightarrow k([Y]_{t+2}, [Z]_{t+1}) \wedge j([Z]_{t+1}, [X]_t) \\ &\text{and } j([Z]_{t+1}, [X]_t) \Rightarrow \mathbf{mid} \end{aligned}$$

then:

$$\begin{aligned} \iota X : oY : [\mathbf{pre}, g([Y]_{t+2}, [X]_t)] &\equiv \\ (\iota X : oZ : [\mathbf{pre}, j([Z]_{t+1}, [X]_t)] &\parallel \\ \iota Z : oY : [\mathbf{mid}, k([Y]_{t+2}, [Z]_{t+1})]) &\setminus Z \end{aligned}$$

i.e., we may split into two parts a process for which an ‘‘intermediate calculation’’ exists.

The natural interpretation of this law is an intermediate calculation on the inputs X , using the spare time slot between each input and corresponding output to produce intermediate results Z , and the final results Y .

As an example, let the predicates be:

$$\begin{aligned} g(\{a, b, c, d\}, \{e\}) &= e = a \wedge b \wedge c \wedge d \\ j(\{a, b, c, d\}, \{f, h\}) &= f = (a \wedge b) \wedge h = (c \wedge d) \\ k(\{f, h\}, \{e\}) &= e = f \wedge h \\ \mathbf{mid} &= \mathbf{true} \end{aligned}$$

which allows refinement of a two-delay four-input AND gate into two parallel 2-1 AND gates feeding into a third 2-1 AND gate.

We justify this law in terms of the SRPT processes G , K and J represented by the three frames. The refinement rule requires that:

$$G[X, Y] \equiv (J[X, Z] \parallel K[Z, Y]) \setminus Z$$

and so we must show that the traces of the left and right hand side are equivalent. We specify the most general traces possible for each side, and aim to show their equivalence.

A new notation we introduce is the use of a horizontal bar \bar{x}_1 to represent groups of events from a set X .

We first construct the traces of the right-hand side. Given $s \in \mathcal{T}_{\mathcal{R}}[[J]]\sigma$:

$$\begin{aligned} s &= \langle \bar{x}_1, \alpha(\bar{x}_1) \cup \bar{x}_2, \alpha(\bar{x}_2) \cup \bar{x}_3, \dots \rangle \\ \text{where } \mathbf{pre}(a) &\Rightarrow (j(b, a) \Leftrightarrow b = \alpha(a)) \end{aligned}$$

The process J can then be specified in SRPT notation as:

$$J_A = [!A ?M \rightarrow J_{\alpha(M)}]$$

Similarly, for $u \in \mathcal{T}_{\mathcal{R}}[[K]]\sigma$:

$$\begin{aligned} u &= \langle \bar{z}_0, \bar{z}_1, \beta(\bar{z}_1) \cup \bar{z}_2, \beta(\bar{z}_2) \cup \bar{z}_3, \dots \rangle \\ \text{where } \mathbf{mid}(a) &\Rightarrow (k(b, a) \Leftrightarrow b = \beta(a)) \end{aligned}$$

The process K can then be specified in SRPT notation as:

$$K_B = [!B ?N \rightarrow K_{\beta(N)}]$$

We apply law **a-10** from Barnes[Bar93] pp. 78 to get:

$$J_A \parallel K_B = [!(A \cup B) ?Q \rightarrow J_{(Q \cup B) \cap \iota J_{\emptyset}} \parallel K_{(Q \cup A) \cap \iota K_{\emptyset}}]$$

We know from the disjoint process input and output alphabets that this simplifies to:

$$[!(A \cup B) ?Q \rightarrow J_{Q \cap X} \parallel K_{Q \cap Z}]$$

This establishes that, at any point in any of its traces, the tail of process $J \parallel K$ is always equivalent to $J_A \parallel K_B$ for some A and B .

Given this parallel construct, process J guarantees that $\bar{z}_{t+1} = \alpha(\bar{x}_t)$. Process K guarantees that $\bar{y}_{t+2} = \beta(\bar{z}_{t+1}) = \beta(\alpha(\bar{x}_t))$. From the earlier definitions then, $k(\bar{y}_{t+2}, \alpha(\bar{x}_t))$.

Similarly, $\mathbf{pre}(\bar{x}_t) \Rightarrow j(\bar{z}_{t+1}, \bar{x}_t) \equiv \bar{z}_{t+1} = \alpha(\bar{x}_t)$. We can join these two to get:

$$\mathbf{pre}(\bar{x}_t) \Rightarrow k(\bar{y}_{t+2}, \bar{z}_{t+1}) \wedge j(\bar{z}_{t+1}, \bar{x}_t)$$

which, from the precondition in this refinement law, is equivalent to:

$$\mathbf{pre}(\bar{x}_t) \Rightarrow g(\bar{y}_{t+2}, \bar{x}_t)$$

This matches the original frame specification in the refinement law definition, showing that the left and right hand sides are indeed equivalent, and we have proven the refinement law. \square

Refinement 8 *Introduce delayed intermediate*

If g, j, k , **mid** are timed predicates over subsets of events, and $d_1, d_2 \geq 1$, such that:

$$\begin{aligned} &\forall \text{ disjoint } X, Y, Z \subseteq \Sigma. \\ &g([Y]_{t+d_1+d_2}, [X]_t) \Leftrightarrow k([Y]_{t+d_1+d_2}, [Z]_{t+d_1}) \wedge j([Z]_{t+d_1}, [X]_t) \\ &\text{and } j([Z]_{t+d_1}, [X]_t) \Rightarrow \mathbf{mid} \end{aligned}$$

then:

$$\begin{aligned} \iota X : oY : [\mathbf{pre}, g([Y]_{t+d_1+d_2}, [X]_t)] &\equiv \\ (\iota X : oZ : [\mathbf{pre}, j([Z]_{t+d_1}, [X]_t)] &\parallel \\ \iota Z : oY : [\mathbf{mid}, k([Y]_{t+d_1+d_2}, [Z]_{t+d_1})]) &\setminus Z \end{aligned}$$

i.e., we may split into two parts a process for which an “intermediate calculation” exists at some time point between start and end of calculation.

This law is justified by repeated application of refinement law 7.

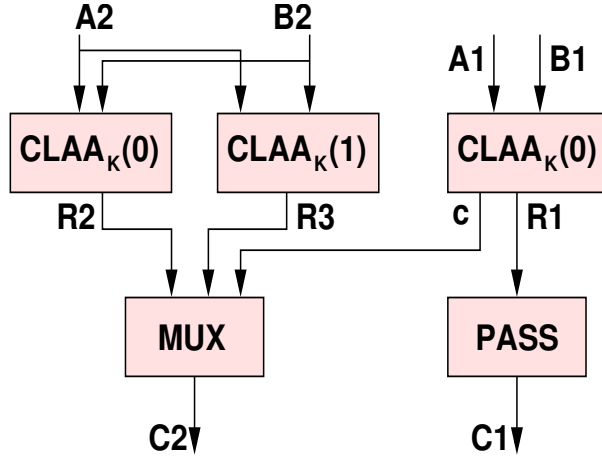


Figure 5.2: Carry look-ahead adder structure

5.2.6 Feasibility

We construct the *maximal trace set* \mathbf{max} of two event sets X, Y by:

$$\begin{aligned} \langle \rangle &\in \mathbf{max}(X, Y) \\ t \in \mathbf{max}(X, Y) &\Rightarrow \forall A \subseteq X, B \subseteq Y. \\ &(t \frown (A \cup B)) \in \mathbf{max}(X, Y) \end{aligned}$$

i.e., the well-formed trace set with all combinations of input and events possible at each time step.

The specification $P = \forall t \in \mathbb{N} \cdot \iota X : oY : [[\mathbf{pre}]_t, [\mathbf{post}]_{t+k}]$ is *feasible* if it is well-formed according to the refinement frame rules listed in Section 5.2.2, and:

$$\exists s \in \mathbf{max}(X, Y) : \forall t \in \mathbb{N} \cdot \mathbf{pre}(s[t]) \Rightarrow \mathbf{post}(s[t+k])$$

i.e. there is some well-formed trace which, at every time point, satisfies the postcondition as long as the precondition is true.

5.3 Case Study: Carry Look-ahead Adder

A *carry look-ahead adder* is an adder whose design is optimised towards minimal execution time rather than towards minimal area. It works by splitting an addition into two halves (high and low bits), and carrying out two parallel calculations for the high half sum – one for if a carry is received, one for if it isn't. A multiplexer then selects the correct high bits calculation based on the carry-out bit of the lower half calculation. Figure 5.2 shows the structure of one of these devices.

We will now specify this adder and refine it.

5.3.1 Specification

For an $n = 2^k$ bit adder, $CLAA_k$:

$$\iota(A \cup B) : oC : [\mathbf{true}, [\mathbb{N}(C)]_{t+1+k} = [\mathbb{N}(A) + \mathbb{N}(B)]_t]$$

where $\mathbb{N}(X)$ maps the subsets of X onto the natural number given by the binary representation of the events. A and B must contain n events, C must contain $n + 1$.

We will in fact find it useful to specify and refine the processes $CLAA_k(x)$ for all $x \in \mathbb{N} \leq k$, where $[\mathbb{N}(C)]_{t+1+k} = [\mathbb{N}(A) + \mathbb{N}(B) + x]_t$.

Note that the specification requires that the computation complete in $1 + k$ time steps. A simple ripple-carry adder could not in general satisfy this specification since it takes time linear in 2^k to complete; each bit of the sum is computed sequentially with the lowest bit first.

5.3.2 Basic gates

If we set k to 0, and hence n to 1, we get a half adder:

$$HADD = \iota\{a, b\} : o\{c, s\} : [\mathbf{true}, [2c + s]_{t+1} = [a + b]_t]$$

which we already know how to construct, from Section 5.2.4. We note that this takes two of our 2-input, 1-output cells. We assume that the only cells available for construction are 2-input, 1-output and 3-input, 1-output. This will restrict what we regard as “final code” in our refinement.

We will also want a pass gate (for delays) and a 1-bit choice gate. These have the following specifications:

$$\begin{aligned} PASS &= \iota\{x\} : o\{y\} : [\mathbf{true}, [y]_{t+1} = [x]_t] \\ MUX &= \iota\{a, b, c\} : o\{y\} : [\mathbf{true}, [y]_{t+1} = [(b \wedge c) \vee (a \wedge \neg c)]_t] \end{aligned}$$

We can have the 1-input, 1-output PASS gate because it can be embedded into a 2-input, 1-output cell where the second input is taken from ground (i.e. a permanent low value).

5.3.3 Refinement

We proceed by induction on k . The base case for $k = 0$ requires an implementation of the specification of the half adder above. It is possible that the half-adder is a primitive gate on the target device. If not, we apply refinement law 2 to refine the $HADD$ process into:

$$\begin{aligned} HADD &= LO \parallel HI \text{ where} \\ LO &= \iota\{a, b\} : o\{s\} : [\mathbf{true}, [s]_{t+1} = [a \mathbf{xor} b]_t] \\ HI &= \iota\{a, b\} : o\{c\} : [\mathbf{true}, [c]_{t+1} = [a \mathbf{and} b]_t] \end{aligned}$$

and we will take LO, HI to be primitive gates since they are equivalent to XOR and AND gates respectively.

We therefore assume as the induction hypothesis that we have complete implementations for all processes $CLAA_k(y)$ for all $y \leq k$. We aim to prove the hypothesis for $k + 1$.

Let $n = 2^k$. Then $2n$ is the number of bits for each of the two input numbers to $CLAA_{k+1}(y)$. Let $A = A_1 \cup A_2$ where $A_1 = \{a_1, \dots, a_n\}$ and $A_2 = \{a_{n+1}, \dots, a_{2n}\}$. Define B_1, B_2, C_1 similarly and $C_2 = \{c_{n+1}, \dots, c_{2n+1}\}$. From now on, for convenience we will omit the \mathbb{N} in the arithmetic by referring to direct addition of event sets.

We start with the process specification of $CLAA_{k+1}(x)$:

$$\begin{aligned} & \iota(A \cup B) : oC : \\ & [\text{true}, \\ & \quad [C]_{t+2+k} = [A + B + x]_t \] \end{aligned}$$

then expand the input and output set definitions:

$$\begin{aligned} & \iota(A_1 \cup A_2 \cup B_1 \cup B_2) : o(C_1 \cup C_2) : \\ & [\text{true}, \\ & \quad [C_1]_{t+2+k} = ([A_1 + B_1 + x]_t) \mathbf{mod} 2 \quad \wedge \\ & \quad [C_2]_{t+2+k} = ([A_1 + B_1 + x]_t) \mathbf{div} 2 + [A_2 + B_2]_t \] \end{aligned}$$

Applying refinement law 8 (*Introduce delayed intermediate*) we introduce the intermediate event set $(R_1 \cup R_2 \cup R_3 \cup \{c\})$, the components of which have respective sizes $n, n + 1, n + 1$ and 1 . We also introduce the set union abbreviation notation $X_{a,b}$ for $X_a \cup X_b$. We may rewrite this as:

$$\begin{aligned} & (\iota(A_{1,2} \cup B_{1,2}) : o(R_{1,2,3} \cup \{c\}) : \\ & [\text{true}, \\ & \quad [R_1]_{t+1+k} = ([A_1 + B_1 + x]_t) \mathbf{mod} 2 \quad \wedge \\ & \quad [R_2]_{t+1+k} = [A_2 + B_2]_t \quad \wedge \\ & \quad [R_3]_{t+1+k} = 1 + [A_2 + B_2]_t \quad \wedge \\ & \quad [c]_{t+1+k} = ([A_1 + B_1 + x]_t) \mathbf{div} 2 \quad] (1) \end{aligned}$$

$$\begin{aligned} & || \iota(R_{1,2,3} \cup \{c\}) : oC_{1,2} : \\ & [\text{true}, \\ & \quad [C_1]_{t+1} = [R_1]_t \quad \wedge \\ & \quad [C_2]_{t+1} = [(R_3 \wedge c) \vee (R_2 \wedge \neg c)]_t \quad] (2) \\ &) \setminus (R_{1,2,3} \cup \{c\}) \end{aligned}$$

To show that this refinement law has been applied correctly, we need to define the predicate functions g, j, k , **pre**, **mid** and the delays d_1, d_2 as specified in the refinement law precondition. These are as follows:

$$\begin{aligned} g &= [C_1]_{t+2+k} = ([A_1 + B_1 + x]_t) \mathbf{mod} 2 \wedge \\ & \quad [C_2]_{t+2+k} = ([A_1 + B_1 + x]_t) \mathbf{div} 2 + [A_2 + B_2]_t \\ j &= [R_1]_{t+1+k} = ([A_1 + B_1 + x]_t) \mathbf{mod} 2 \wedge \\ & \quad [R_2]_{t+1+k} = [A_2 + B_2]_t \wedge \\ & \quad [R_3]_{t+1+k} = 1 + [A_2 + B_2]_t \wedge \\ & \quad [c]_{t+1+k} = ([A_1 + B_1 + x]_t) \mathbf{div} 2 \\ k &= [C_1]_{t+2+k} = [R_1]_{t+1+k} \wedge \\ & \quad [C_2]_{t+2+k} = [(R_3 \wedge c) \vee (R_2 \wedge \neg c)]_{t+1+k} \\ d_1 &= 1 + k \\ d_2 &= 1 \\ \mathbf{pre} &= \text{true} \\ \mathbf{mid} &= \text{true} \end{aligned}$$

To show that j, k combined are equivalent to g , we must show that the values for

C_1 and C_2 in the composition of j and k are equivalent to their values in g :

$$\begin{aligned}
& [C_1]_{t+1} = [R_1]_t \\
& [R_1]_{t+1+k} = ([A_1 + B_1 + x]_t) \mathbf{mod} 2 \\
\Rightarrow & [C_1]_{t+2+k} = ([A_1 + B_1 + x]_t) \mathbf{mod} 2 \\
& [C_2]_{t+1} = [(R_3 \wedge c) \vee (R_2 \wedge \neg c)]_t \\
& [R_2]_{t+1+k} = [A_2 + B_2]_t \\
& [R_3]_{t+1+k} = 1 + [A_2 + B_2]_t \\
& [c]_{t+1+k} = ([A_1 + B_1 + x]_t) \mathbf{div} 2 \\
\Rightarrow & [C_2]_{t+2+k} = [A_2 + B_2]_t + ([A_1 + B_1 + x]_t) \mathbf{div} 2
\end{aligned}$$

which is as required.

We take each of the refined processes in turn for further refinement.

$$\begin{aligned}
(1) \sqsubseteq & \text{ via refinement law 2 (*Parallelism*) :} \\
& \iota(A_{1,2} \cup B_{1,2}) : o(R_1 \cup \{c\}) : \\
& [\mathbf{true}, \\
& \quad [R_1]_{t+1+k} = ([A_1 + B_1 + x]_t) \mathbf{mod} 2 \quad \wedge \\
& \quad [c]_{t+1+k} = ([A_1 + B_1 + x]_t) \mathbf{div} 2 \quad] (3) \\
& \parallel \iota(A_{1,2} \cup B_{1,2}) : oR_2 : \\
& [\mathbf{true}, [R_2]_{t+1+k} = [A_2 + B_2]_t \quad] (4) \\
& \parallel \iota(A_{1,2} \cup B_{1,2}) : oR_3 : \\
& [\mathbf{true}, [R_3]_{t+1+k} = 1 + [A_2 + B_2]_t \quad] (5)
\end{aligned}$$

We apply refinement law 6 (*Contract frame*) to remove A_1, B_1 from (4), (5) and A_2, B_2 from (3), giving:

$$\begin{aligned}
& \iota(A_1 \cup B_1) : o(R_1 \cup \{c\}) : \\
& [\mathbf{true}, \\
& \quad [R_1]_{t+1+k} = ([A_1 + B_1 + x]_t) \mathbf{mod} 2 \quad \wedge \\
& \quad [c]_{t+1+k} = ([A_1 + B_1 + x]_t) \mathbf{div} 2 \quad] (3a) \\
& \parallel \iota(A_2 \cup B_2) : oR_2 : \\
& [\mathbf{true}, [R_2]_{t+1+k} = [A_2 + B_2]_t \quad] (4a) \\
& \parallel \iota(A_2 \cup B_2) : oR_3 : \\
& [\mathbf{true}, [R_3]_{t+1+k} = 1 + [A_2 + B_2]_t \quad] (5a)
\end{aligned}$$

Here, (3a), (4a) and (5a) are equivalent to the specifications of processes $CLAA_k(x)$, $CLAA_k(0)$ and $CLAA_k(1)$ respectively, with input and output wires renamed appropriately. Since each specification has a well-defined trace set, and equality of trace sets means equivalence of processes, we can substitute in the renamed $CLAA_k$ processes.

The second part of the refinement proceeds as follows.

$$\begin{aligned}
(2) = & \iota(R_{1,2,3} \cup \{c\}) : oC_{1,2} : \\
& [\mathbf{true}, \\
& \quad [C_1]_{t+1} = [R_1]_t \quad \wedge \\
& \quad [C_2]_{t+1} = [(R_3 \wedge c) \vee (R_2 \wedge \neg c)]_t \quad]
\end{aligned}$$

$$\begin{aligned}
&\sqsubseteq \text{ via refinement law 2 (Parallelism) :} \\
&\iota(R_{1,2,3} \cup \{c\}) : oC_1 : \\
&\quad [\mathbf{true}, [C_1]_{t+1} = [R_1]_t \quad] \quad (6) \\
&\parallel \iota(R_{1,2,3} \cup \{c\}) : oC_2 : \\
&\quad [\mathbf{true}, [C_2]_{t+1} = [(R_3 \wedge c) \vee (R_2 \wedge \neg c)]_t \quad] \quad (7)
\end{aligned}$$

We apply refinement law 6 (*Contract frame*) to remove $R_{2,3}$ from (6) and R_1 from (7):

$$\begin{aligned}
&\iota(R_1 \cup \{c\}) : oC_1 : \\
&\quad [\mathbf{true}, [C_1]_{t+1} = [R_1]_t \quad] \quad (6a) \\
&\parallel \iota(R_{2,3} \cup \{c\}) : oC_2 : \\
&\quad [\mathbf{true}, [C_2]_{t+1} = [(R_3 \wedge c) \vee (R_2 \wedge \neg c)]_t \quad] \quad (7a)
\end{aligned}$$

(6a) is equivalent to n parallel *PASS* processes between R_1 and C_1 ; we apply refinement law 2 (*Parallelism*) and substitute the renamed *PASS* processes as noted above.

(7a) is equivalent to $n + 1$ parallel *MUX* cells, choosing from R_2 and R_3 using c , sending to C_2 . Again, we apply refinement law 2 and substitute renamed *MUX* processes.

We can now collate the refinement to produce:

$$\begin{aligned}
&\iota(A \cup B) : oC : \quad [\mathbf{true}, [C]_{t+2+k} = [A + B + x]_t] \\
&\quad \sqsubseteq \\
&\quad (\quad CLAA_k(x)[A_1, B_1][R_1, c] \quad (3) \\
&\quad \parallel \quad CLAA_k(0)[A_2, B_2][R_2] \quad (4) \\
&\quad \parallel \quad CLAA_k(1)[A_2, B_2][R_3] \quad (5) \\
&\quad \parallel_{i=1}^n \quad PASS[r_i][c_i] \quad (6) \\
&\quad \parallel_{i=1}^{n+1} \quad MUX[r_{n+i}, r_{2n+i}, c][c_{n+i}] \quad (7) \\
&\quad) \quad \setminus (R_{1,2,3} \cup \{c\})
\end{aligned}$$

With a relatively short formal derivation we have produced a full implementation for a family of arithmetic functions, parametrised by size, and demonstrated that the calculations complete in the specified time. This has been done using a predefined set of simple gates *HADD*, *PASS* and *MUX*.

5.3.4 Space and time

The specification tells us that the computation completes in $1 + k$ time steps, and since it is true for all values of $t \in \mathbb{N}$ it tells us that a new calculation result is delivered at every timestep from $t = 1 + k$ onwards, i.e. the calculation is pipelined.

As far as space is concerned, we define a function $C(k)$ which gives the number of cells used by $CLAA_k$ and which comes from the final (recursive) definition of the process:

$$C(k) = 3C(k - 1) + P(2^{k-1}) + M(1, 2^{k-1})$$

where $P(b)$ is the number of cells for an n -bit *PASS* block and $M(a, b)$ is the number of cells for an a -bit choice, b -bit output multiplexer. $P(b) = b$ and $M(1, b) = b$ in this case, so:

$$C(k) = 3C(k - 1) + 2^k$$

for $k > 0$, and $C(0) = 2$. This gives $C(1) = 8$, $C(2) = 28$ and so on. This indicates that cell usage varies as $O(3^k)$ where n is the size in bits of each argument. A 32-bit adder, producing a 33-bit answer, would require $C(5) = 908$ cells.

Note that a simple ripple-carry adder would *not* satisfy the specification in general because its computation time is linear in its argument length. If the timing requirements were relaxed, ripple-carry adders could be inserted instead of carry-lookahead adders in some layers. This would not, however, save cells; the requirement to have the entire result come out at one time point means that the adder needs a large number of *PASS* cells.

5.3.5 Scalability

The above approach has illustrated a number of key concepts. An important one is the use of previously defined processes in development. We saw this where smaller *CLAA* blocks were used in the construct of a larger one.

If this refinement method were used in the creation of a substantial PLD program then it would be useful to build up a library of specifications and the processes that satisfy them. Note that several processes may meet one specification, and the developer may choose one based on available cell configurations, computation time and cell usage.

The *Introduce intermediate* refinement law is a powerful one because it encapsulates an activity, hiding the internal events which are needed to make the calculation. This enables the effective top-down design and implementation of a complex programmable logic program. The design will refine the initial specification into a number of parallel sub-specifications, which will either match existing library components or which can be handed to individual developers to implement. The specification carries inside it the interface and timing information needed by the developer.

Blocks on the PLD which perform a fixed function can have a specification written for them retroactively. This enables them to be part of a refined system and interface to other refined components. The difficulty is in writing their specifications correctly.

5.3.6 Proof means no testing?

Bearing in mind Knuth's famous quote "Beware of bugs in the above code; I have only proved it correct, not tried it" [Knu77] we implemented the above structure in a simple Pebble simulator written in Perl and tested it with random input data.

Knuth was proven prudent. In the original refinement, (3) had mistakenly been asserted equivalent to $CLAA_k(0)$ rather than $CLAA_k(x)$. The tests detected this, it was corrected, and the tests rerun. No errors were found in the corrected version for values of k from 0 to 5. The simulator was later expanded and rewritten, with the results given in Section 7.2.

This is more a comment on the methodology that we used to arrive at our starting point rather than the subsequent refinement. In essence, no matter how good a refinement, it can only be as good as the starting specification from which it was derived. To validate that a system fits its purpose requires testing of the system in conditions as close as possible to the intended operational environment, as no single formal verification procedure can be sufficient.

There is clearly value in independent inspection of refinement to pick up problems such as these. In order to measure the reliability of the inspection, it may be useful to inject a number of faults into the proof before inspection.

5.4 Summary

In this chapter we have presented a refinement calculus with a specification notation based on Morgan’s notation for refinement, using SRPT as the implementation language and adding an integer time aspect to the variables. We have shown how existing refinement laws can be adapted to suit the new calculus, introduced a new law specific to the parallel process model and shown how it can be proven.

We have demonstrated the specification and complete refinement of a carry look-ahead adder. The refinement was not lengthy or particularly complex, and few implementation decisions were required. One mistake occurred during refinement, which was detected and corrected during testing. This indicates that the refinement model is practical, at least for one class of specifications, but is not a panacea.

This refinement calculus is open for further development by adding new refinement laws, for instance concerning iteration or alternation.

The refinement rules and notation described in this chapter are summarised in Appendix A.

5.4.1 Alternative approaches

A complementary approach to parallel refinement was presented by Sanders and Lai in [LS97]. The approach is also based on Morgan’s stepwise refinement model, extending it to refine into a parallel communicating programming language with a syntax similar to occam[Ltd84] rather than Dijkstra’s language of guarded commands.

This approach diverges from our approach principally in that the system modelled does not operate on a synchronous discrete clock but rather in the asynchronous model familiar from CSP. It is useful however to observe that the refinement laws established by Sanders and Lai (e.g. strengthen postcondition, weaken precondition, sequential composition, parallel composition) are similar in intent to those we defined in Section 5.2. The authors identify the same weaknesses in their system with respect to scalability that we have found. It represents a comrade rather than competitor system for our SRPT refinement process.

5.4.2 Targets

Of the targets in Chapter 3 we have addressed or partially addressed:

Target 1: *The process we define must be rigorous.*

We have extended a subset of the rigorous process algebra SRPT, described in Section 4.1, to include a “refinement frame” syntactic construct. We have also developed a refinement calculus to support refinement between constructs in this notation, and hence between trace sets in SRPT. This work has been supported with formal proof of relevant assertions and refinement laws.

Target 2: *The process must help the developer to write unambiguous programs.*

The use of the refinement calculus produces programs that demonstrably meet their specification.

Target 5: *The program produced must be easy to test.*

Test cases may be generated from the program specification.

Target 6: *The program must be able to be compiled onto a range of existing and anticipated PLDs.*

The mapping between SRPT and Pebble, as described in Section 4.2, is PLD-independent.

Target 9: *The process should indicate what kinds of error may arise at each stage.*

We have seen how the manual refinement process may introduce errors, and indicated how manual review may address this.

Target 10: *The process should provide flexibility so that it may be used in situations not anticipated in its original design.*

SRPT allows incorporation of processes that may act in an arbitrary way; our proof system allows us to incorporate them in a system and reason formally about the effect they may have on the rest of the system.

Target 12: [00-54 8.5.2] *The analytical arguments provided shall include:*

- (i) *any formal arguments used in validation to show that the formal specification complies with the safety requirements;*
- (ii) *any formal arguments that the functional design satisfies the formal specification;*
- (iii) *for non-functional properties with specified safety requirements, analysis of the achieved behaviour, e.g.: performance, timing etc.;*
- (iv) *analysis of the effectiveness of fault mitigation, for example use of such techniques as diverse implementations.*

(i) is addressed because the safety requirements may be expressed as post-conditions in a process specification. (ii) is addressed because the refinement process produces an evidence trail, amenable to manual review, that the SRPT process satisfies its specification. (iii) is addressed because the timed specification process allows timing requirements to be stated explicitly and shown to be met. (iv) is not addressed.

Chapter 6

A PLD Interpreter of SPARK

In Chapter 5 we specified a process for refining high-level specifications into SRPT processes, and hence transforming them into implementations in Pebble. This is an effective method for relatively simple specifications, but a $\iota X : oY : [\mathbf{pre}, \mathbf{post}]$ refinement frame which described a substantial program would normally be unwieldy and difficult to manage.

Section 4.3.12 outlined a possible design for an interpreter for SPARK Ada, running on one or more PLDs. This interpreter would be difficult to verify to the degree required for high-integrity PLD programs, but may be appropriate for running PLD programs at lower levels of required integrity. In this chapter we expand this outline to build a SPARK interpreter out of SRPT processes, using refinement to build small computational units in the interpreter and defining a higher-level protocol to manage execution of the SPARK “bytecode”.

We will describe the interpreter architecture, then break down its structure to examine how individual units of SPARK code are executed within it. We will also see how our techniques of refining specifications from Chapter 5 are useful in making custom combinational logic sequences.

Target aims

This chapter chiefly addresses **Target 2** (the process must force the developer to write unambiguous programs) and **Target 4** (it must enable as much static analysis as possible). We aim to achieve this by allowing developers to write programs in the SPARK language which already satisfies these requirements.

Our aim is to produce a design for a SPARK interpreter which runs on a generic PLD. The interpreter should:

1. be amenable to arguments that it correctly executes SPARK;
2. interpret as large a subset of SPARK Ada 95 as possible;
3. not depend on any feature of a specific PLD;
4. make relatively efficient use of available PLD resources; and
5. scale in performance with increased resources.

The design must be practical, since in Chapter 7 we will have to produce a working implementation of the interpreter as part of the case study.

There is a secondary aim, related to the SRPT specification and refinement work in Chapter 5. We will specify a number of SRPT processes in our description of the interpreter, which will be a test of the usability of the specification form. We aim to use these tests to measure whether our SRPT specification scheme is suitable for specifying significant complex systems with a range of functions.

Scope

The SPARK constructs recognised by the interpreter are restricted in that no constructs particular to the Ravenscar tasking profile are permitted. This is to simplify the interpreter's architecture.

The interpreter is intended to be a proof-of-concept, not an optimised design.

No particular assumptions are made about limiting features of PLD design e.g. available cells or routing resources. For this reason we refer to the target PLD as the "virtual" PLD.

Structure

Section 6.1 presents an overview of the interpreter design. Section 6.2 describes the mechanism for communicating between the CPU and the PLD. Section 6.3 describes the mechanism for communicating between package units upon the PLD. Section 6.4 details the structure of the package units. Section 6.5 describes how SPARK programs are transformed into a form suitable for execution on the interpreter. Section 6.6 discusses how SPARK software interacts with the PLD program.

Finally, Section 6.7 discusses optimisations to the interpreter and Section 6.8 draws conclusions from the chapter.

6.1 Interpreter Overview

We now describe the design of the interpreter to give the reader a context for the rest of the chapter.

6.1.1 Architecture

The interpreter is designed to contain a SPARK package P and any other packages on which P depends, directly or indirectly. It will be controlled at the top level from software; in the SPARK program compiled for the normal CPU there will be a shadow package for P which will manage sending data to and from the interpreter. P is henceforth referred to as the *root package*.

Each package is implemented as one contiguous unit on the virtual PLD, with data connections between packages corresponding to subprogram calls. There is a connection from package P to package Q if and only if there is a call from a subprogram of P to a subprogram of Q . The SPARK rules on inheritance order guarantee that there cannot then be a call from Q to P .

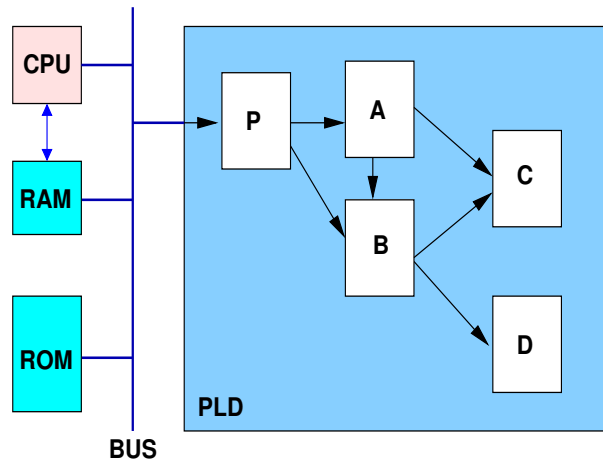


Figure 6.1: Interpreter architecture

The top-level architecture is shown in Figure 6.1. This example shows root package P with direct dependencies on A and B, and indirect dependencies on C and D.

There are then three major components to the interpreter; the I/O between CPU and PLD, the I/O between packages and the internal workings of the package itself. This is the taxonomy we will use in the rest of this chapter.

Note that place-and-route issues may break a contiguous design unit over several parts of an actual PLD.

6.1.2 Partitioning issues

The performance of the interpreter, in terms of execution speed and PLD cell usage, will depend on the packages selected for compilation. There are rules and guidance on package selection as follows.

A note on terminology: a *package with state* is one that contains at least one state variable, either directly in its spec or body, or in an embedded or child package. This correspond to the package having at least one *own variable* in SPARK terms.

Rules

1. No package with state may be present in both the software and programmable logic programs. This is to prevent multiple copies of a global package variable.
2. The packages compiled into programmable logic must form a valid SPARK program and a complete Ada program closure. This is essential for the integrity of the compilation process.
3. This program must have run-time checks performed on it by a tool such as the SPARK Examiner (using the `-exp` switch), which must show that it is free from any potential run-time overflows.
4. Packages may not be embedded in subprograms. This is to reduce the complexity of the compiler's task.

Guidance

1. Packages should contain as few variables and as little code as possible.
2. The user should aim to minimise data transfer between packages.
3. Variables should be typed with as small a range as possible in order to reduce storage space and transmission time.

6.2 CPU-PLD I/O

The key point in CPU-PLD I/O is that, in general, there is no clock synchronisation between the two components. The I/O must take account of this, and hence be more complex than the inter-package I/O discussed below.

We assume that the access to the PLD from the software is via memory-mapped I/O, and that within the SPARK program the interface is accessed via a copy of the specification of the root package. Given this, there are four stages of the data's journey to the PLD and back again:

1. between the software and the bus, via MMIO (both ways);
2. from the bus to the PLD's bus interface;
3. from inside the PLD to the PLD's bus interface; and
4. from the PLD's bus interface to the bus.

The apparent asymmetry in stages 2 and 3 is due to the way that the PLD buffers its input information from the bus, turning it into discrete packets, then after computation aggregates packets until a complete frame of data may be transmitted back to the bus.

6.2.1 Software-bus MMIO

The PLD access process starts when a subprogram in the software component makes a call to a subprogram in the root package. As well as the original package specification, there will be a package body where each subprogram from the specification has an implementation. These implementations will be responsible for the data transfer process.

There will also be set of variables, declared using Ada's `for X'Address use A` mechanism to map a variable to a specific location in memory. In this case they will be mapped to the input and output pins of the PLD. There will be four canonical variables:

TX Transmit byte. Initially zero, increases as the data is copied across.

TD Transmit data, of type `Word`.

RX Receive byte. Set by the PLD to indicate the progress of copy-back of data.

RD Receive data, of type `Word`. Set by the PLD.

Transmit

The transmit algorithm is as follows. We assume that the input data is held in an array A : array (1..M) of Word.

```
TX := 0;
-- Wait for the RX byte to clear, showing a ready PLD
while (RX /= 0) loop
    delay(1.0);
end loop;
for idx in range 1..M loop
    TD := A(idx); TX := idx;
    -- Wait for PLD to increment its counter, showing ready
    while (RX < idx) loop
        delay(1.0);
    end loop;
    --# assert (RX = Idx) and (TX = Idx);
end loop;
TX := 0;
```

Conventionally the receive code would follow directly. However, the processing of the data might well take a while. For a program which has a main loop running every 20ms or so the implementer may choose to implement a polling structure and associated state machine. Note that the use of tasking constructs would simplify this significantly.

Receive

The receive algorithm is as follows. We assume that the output data is held in B : array (1..N) of Word.

```
-- Wait for the PLD to signal ready
while (RX = 0) loop
    delay(1.0);
end loop;
for idx in range 1..N loop
    B(idx) := RD; TX := idx;
    -- Wait for PLD to increment its counter, showing ready
    while (RX = idx) loop
        delay(1.0);
    end loop;
    --# assert (RX > Idx);
end loop;
TX := 0;
```

This has implemented an asynchronous copy to and from the PLD's pins. We now look at how the PLD buffers the data.

Bits	Meaning
00	No message / end of message
01	Ignore this packet, message continues
10	Message body
11	Message start

Table 6.1: Packet meaning encoding

Variable	Bits
TX	$U = \{u_1, \dots, u_8\}$
TD	$R = \{r_1, \dots, r_m\}$
RX	$V = \{v_1, \dots, v_8\}$
RD	$S = \{s_1, \dots, s_n\}$

Table 6.2: Memory-mapped variable representations

6.2.2 PLD buffering

The above transmit algorithm maps a chunk of data to the input pins of an PLD and waits for acknowledgement before writing the next chunk. We now need to turn this data stream into the form used to communicate between packages. This means that we can compile the root package in the same form as other packages, with a standardised way of receiving data.

Packages receive data as a stream of packets. Each packet has two marker bits to describe the data coming in, as shown in Table 6.1.

The packet width must then be at least 3 bits. The 01 packets are intended to deal with delays in the message chunks arriving at the input pins.

Event representations

We represent the memory-mapped variables with the event sets shown in Table 6.2 and the input bits of the packet pipeline with $P = \{p_1, \dots, p_k\}$ and $Q = \{q_1, q_2\}$ where $m = x \times k$. This enables us to guarantee that each set of input data from TD can be transmitted in exactly x packets. We define functions u, r, v, s to map the event sets onto representations in \mathbb{N} .

Specification for *BUFFER*

The *BUFFER* process must satisfy the specification given in Equation 6.1. The specification captures the key correctness criteria:

1. RX is reset to 0 when TX is reset to 0;
2. RX increments by exactly 1 each time;
3. RX is only ever 0 or 1 lower than TX;
4. the message header bits coming out of Q form a legal message; and

5. the message data bits coming out of P exactly represent the data arriving through TD with each change of TX.

We define the function $B_k : \mathbb{N} \mapsto \mathbf{seq} \mathbb{B}$ to translate a natural number into its k -digit binary representation, least significant bit first. We define the function $\mathbf{concat}(S)$ to translate a sequence of sequences S into a single joined sequence:

$$\mathbf{concat}(S) = \langle S[i[j]] \mid 0 \leq j < \#S[i] \mid 0 \leq i < \#S \rangle$$

We additionally define the following abbreviations for predicates and operations on a trace t :

$$\begin{aligned} \mathbf{breaks}(t, f) &= \langle i + 1 \mid [f]_i \neq [f]_{i+1} \rangle \\ \mathbf{resets}(t, u) &= \langle i + 1 \mid [u]_i \neq 0 \wedge [u]_{i+1} = 0 \rangle \\ \mathbf{partition}(t, S) &= \langle t[S[k] \dots S[k+1] - 1] \mid 0 \leq k < \#S \rangle \\ \mathbf{stepping}(t, u) &\equiv \forall 0 \leq i < \#t \cdot [u]_i = [u]_{i+1} \vee [u]_i + 1 = [u]_{i+1} \\ \mathbf{follows}(t, v, u) &\equiv \forall 0 \leq i < \#t \cdot [u]_i = [v]_i \vee [u]_i = [v]_i + 1 \\ \mathbf{validhdr}(t, q, k) &\equiv (\forall 0 \leq i < k \cdot [q]_i = 0) \wedge [q]_k = 3 \\ \mathbf{validftr}(t, q, k) &\equiv \exists m \cdot (\forall k < i < m \cdot 1 \leq [q]_i \leq 2) \wedge (\forall j \geq m \cdot [q]_j = 0) \\ \mathbf{validmsg}(t, q) &\equiv \exists k \cdot \mathbf{validhdr}(t, q, k) \wedge \mathbf{validftr}(t, q, k) \\ \mathbf{bitseq}(t, f, g) &= \mathbf{concat}(\langle B_m([f]_i) \mid i \in \mathbf{breaks}(t, g) \rangle) \\ \mathbf{msgseq}(t, p, q) &= \mathbf{concat}(\langle B_k([p]_i) \mid ([q]_i = 3 \vee [q]_i = 2) \wedge i \in 0 \dots \#t \rangle) \end{aligned}$$

The specification is then:

$$\begin{aligned} \forall t \in \mathcal{T}_{\mathcal{R}}[[\mathbf{BUFFER}]]\sigma \quad \forall l \in \mathbf{partition}(t, \mathbf{resets}(t, u)) \cdot \\ \mathbf{stepping}(l, u) \Rightarrow \exists k \cdot \mathbf{resets}(l, v) = \langle k \rangle \\ \wedge \mathbf{stepping}(l[k \dots], v) \\ \wedge \mathbf{follows}(l[k \dots], v, u) \\ \wedge \mathbf{validmsg}(l, q) \\ \wedge \mathbf{bitseq}(l[k \dots], r, u) = \mathbf{msgseq}(l, p, q) \end{aligned} \quad (6.1)$$

Design for *BUFFER*

We define a set of SRPT processes to handle the input. *SPOT* checks the TX value for changes and signals event n to *SIGNAL*; event z is signalled instead if TX has changed back to zero, indicating end of data. *SIGNAL* breaks the TD value into packets and sends them off, sending 01 packets and signalling d to *ACK* if it runs out of data. *ACK* sends the correct RX back to the software client once it gets the signal from *SIGNAL*, and listens for the z event from *SPOT*. *RD* is unused for this part of the communication.

Figure 6.2 shows the processes and connections.

The SRPT specifications of the buffering processes are then as follows. They are parametrised by possible delays in calculations. Some delays (e.g. $e = 0$) may be infeasible for certain PLD architectures.

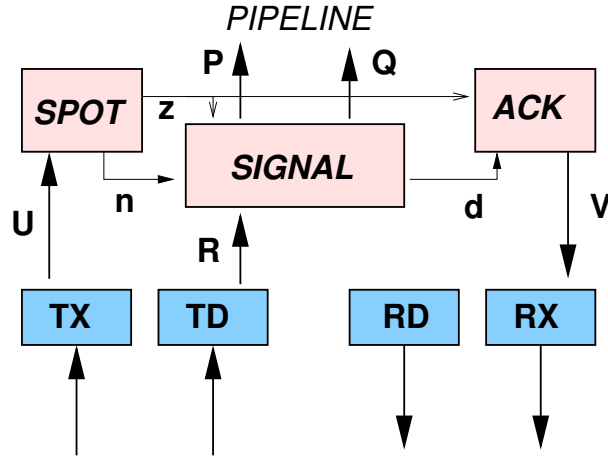


Figure 6.2: PLD input buffer

$$\begin{aligned}
SPOT_e &= \iota U : o\{n, z\} : \\
&[\text{true}, \\
& \quad ([u]_t \neq [u]_{t+1} \wedge [u]_{t+1} \neq 0) \Leftrightarrow [n]_{t+2+e} \\
& \quad \wedge ([u]_t \neq [u]_{t+1} \wedge [u]_{t+1} = 0) \Leftrightarrow [z]_{t+2+e} \\
&] \\
ACK_f &= \iota\{d, z\} : oV : \\
&[\text{true}, \\
& \quad (([z]_t \wedge \exists i : [\neg d]_{t\dots t+i}) \Leftrightarrow ([v]_{t+1+f\dots t+1+f+i} = 0)) \\
& \quad \wedge (([d]_t \wedge \exists i : [\neg d]_{t+1\dots t+i}) \Leftrightarrow \\
& \quad \quad ([v]_t + 1 = [v]_{t+f+1} \wedge [v]_{t+f+2\dots t+f+i+1} = [v]_{t+f+1})) \\
&] \tag{6.2}
\end{aligned}$$

The *SIGNAL* process can be split further into *HDR* and *DATA* which send out the header bits and data bits for each packet simultaneously. Note the precondition, which states that new data signals must not arrive until there has been time to send out packets for all the current data.

$$\begin{aligned}
SIGNAL &= HDR \parallel DATA \tag{6.3} \\
HDR &= \iota\{n, z\} : o(Q \cup \{d\}) : \\
&[[n]_t \Rightarrow [\neg(n \vee z)]_{t+1\dots t+x}, \\
& \quad ([n]_t \wedge \exists y \geq x : [\neg(n \vee z)]_{t+1\dots t+y}) \Leftrightarrow \\
& \quad ([q_1 \wedge q_2]_{t+1} \wedge [q_2 \wedge \neg q_1]_{t+2\dots t+x} \wedge \\
& \quad [q_1 \wedge \neg q_2]_{t+x+1\dots t+y}) \wedge \\
& \quad ([\neg d]_{t+1\dots t+x} \wedge [d]_{t+x+1} \wedge [\neg d]_{t+x+2\dots t+y}) \\
& \quad \wedge ([z]_t \wedge \exists y : [\neg n]_{t+1\dots t+y}) \Leftrightarrow \\
& \quad [\neg(q_1 \vee q_2 \vee d)]_{t+1\dots t+y+1} \\
&] \\
DATA &= \iota(R \cup \{n\}) : oP :
\end{aligned}$$

$$\begin{aligned}
& [\quad [n]_t \Rightarrow [\neg n]_{t+1\dots t+x}, \\
& \quad \forall 1 \leq i \leq k \cdot \\
& \quad \forall 1 \leq j \leq x \cdot \\
& \quad [n]_t \Leftrightarrow ([p_i]_{t+j} = [r_{(j-1)k+i}]_{t+j-1}) \\
&]
\end{aligned}$$

The set of traces of these processes in parallel, with all events other than P and Q hidden, define a sequence of messages:

$$\begin{aligned}
& \text{BUFFER} = \iota(U \cup R) : o(P \cup Q \cup V) : \\
& (\text{SPOT}_e \parallel \text{ACK}_f \parallel \text{HDR} \parallel \text{DATA}) \setminus \{n, z, d\}
\end{aligned}$$

In Section 6.3 we examine the format of these messages in more detail.

6.2.3 PLD readout

We assume that the root package has made the appropriate computations and updated its internal state as required, and is now ready to send back the data to its caller. We further assume that the data is at a fixed location in the package's local RAM store and is of a known length w words. The RAM store must be capable of a multi-word serial read, started with signal s and outputting the word data $D = \{d_1, \dots, d_k\}$ for the subsequent w cycles.

We name the bus interface signals P and Q for data and header bits respectively, as above. Since there are two header bits, $Q = \{q_1, q_2\}$. If the “start output” signal is g then the two processes $IHDR_w$ and $IDATA$ will manage between them:

$$\begin{aligned}
& \text{IHDR}_w = \iota\{g\} : o\{s, q_1, q_2\} : \\
& [\quad [g]_t \Rightarrow [\neg g]_{t+1\dots t+w+3}, \\
& \quad [g]_t \Leftrightarrow \\
& \quad [s \wedge \neg(q_{1,2})]_{t+1} \wedge [\neg s]_{t+2\dots t+w+3} \wedge \\
& \quad [\neg q_{1,2}]_{t+2} \wedge [q_{1,2}]_{t+3} \wedge [q_2 \wedge \neg q_1]_{t+4\dots t+w+2} \wedge \\
& \quad [\neg q_{1,2}]_{t+w+3} \\
&] \\
& \text{IDATA} = \iota D : oP : \\
& [\quad \text{true}, \forall 1 \leq i \leq k \cdot ([d_i]_t \Leftrightarrow [p_i]_{t+1})]
\end{aligned} \tag{6.4}$$

Note that $IDATA$ is a simple $PASS_k$ process.

This will send the root package return data along the standard bus to the MMIO writeback processes.

6.2.4 Writeback to bus

The final task is to map the return data onto RD in chunks, signalling with RX to the software routine that the new data is available and checking TX for acknowledgements that each data chunk has been received.

The key difficulty here is that the output data has to be buffered in a local RAM store since the software can wait an arbitrarily long time to acknowledge each data chunk. The write-back buffer has to incorporate a store large enough to hold the entire return message.

Design

The strategy is to set up one group of processes to parse the incoming packets and write them serially into RAM, a second group to count when sufficient data has been written into RAM for the next data chunk to be written out, and a third group to handle the protocol of communicating with the software.

Note that the RAM has varying bit widths on its ports. k data bits will come off the input bus each cycle, so the input write port will be k bits wide. If the maximum length of a return message is 2^y words of k bits then the input write address port will be y bits wide. The output MMIO register RD is an arbitrary n bits wide, so the input read address port will be $l = \lceil \log_2(2^y k/n) \rceil$ bits wide.

Parsing

The first group of processes contains *PASS* which relays the $P = \{p_1, \dots, p_k\}$ packet data bits to the RAM data input pins $D = \{d_1, \dots, d_k\}$, *PASS* to pass event q_2 through to the serial write start pin w of RAM, and *CTRL* which increments the RAM write address register bits $A = \{a_1, \dots, a_y\}$ by one each time, starting from zero when a message start packet comes in.

Let a map the events of A onto \mathbb{N} , then the *CTRL* process can be specified:

$$\begin{aligned} CTRL &= \iota Q : oA : \\ &[\quad [q_{1,2}]_t \Rightarrow \exists i \cdot [q_2 \wedge \neg q_1]_{t+1 \dots t+i} \wedge [\neg q_{1,2}]_{t+i+1}, \\ &\quad [q_{1,2}]_t \Leftrightarrow ([a]_{t+1} = 0) \wedge (\forall j \leq i : [a]_{t+2+j} = 1 + [a]_{t+1+j}) \\ &] \end{aligned}$$

Counting

The second group assesses when the next write is ready according to two criteria. Sufficient data must have been written to RAM, and the previously sent TD data must have been acknowledged.

We can re-use process *SPOT* to observe changes on TX , signalling event n . We can use an *AND* gate on the Q events so that the output signal z signals the start of a new message sending. In addition we introduce a new process *RADDR* which outputs the RAM read address with event set $B = \{b_1, \dots, b_l\}$. *RADDR* need not know whether the correct data has yet been written into the RAM slots being read as long as the third group ensures that RX is not incremented until sufficient data is in.

If b maps the events of B onto \mathbb{N} then *RADDR* can be specified as follows. Note that \oplus_l denotes addition modulo l .

$$\begin{aligned} RADDR &= \iota\{z, n\} : oB : \\ &[\quad \mathbf{true}, \\ &\quad ([z]_t \Rightarrow [b]_{t+1} = 0) \wedge \end{aligned}$$

$$\begin{aligned}
& ([n \wedge \neg z]_t \wedge [\neg(n \vee z)]_{t+1\dots t+i}) \Rightarrow \\
& [b]_{t+1\dots t+i+1} = 1 \oplus_l [b]_t \\
&]
\end{aligned}$$

Communicating

The third group must count the incoming packets to determine when sufficient data has been input for the RAM output to be valid. It must also check that the software has acknowledged the last send.

TAP will take events Q (the packet header bits) and event n out of SPOT as input. It will output the maximum value of RX permissible given this range of valid RAM contents.

First we need a process $EVERY_k$ which outputs signal b once for every k times that the input a is high. We also need a “semaphore” process SEM which maintains an internal counter of b events and checks for n events. SEM will send out a d signal to allow transmission once a n event has been received and the b counter is non-zero. When the d signal is sent it will clear its n signal receipt and decrement its b counter by one. This has the effect of signalling d only when a transmission has been acknowledged and sufficient data has been read.

Note that an initial n event must be supplied when the message sending starts, since the PLD must take the initiative in the return data protocol. For this reason the n input into SEM should be **OR**ed with the message start event $z = q_1$ **AND** q_2 to produce event m .

Finally, we reuse process ACK from Equation 6.2 to write incrementing values onto RX . It takes events z and d as input.

TAP is then $EVERY_l[a \setminus q_2] \parallel SEM \parallel ACK \parallel OR[a, b, c \setminus n, z, m]$.

To specify $EVERY_k$ we need to define a counter state function $c()$ where $\mathbf{ran} c = 0 \dots k - 1$, and similarly for SEM we need a counter $s()$ where $\mathbf{ran} s = -1 \dots 2^y$ where 2^y is the maximum number of data packets in a return message. There will be a multi-cycle delay for most values of k in most architectures so we need to specify this with parameters $v, w > 0$ for $EVERY$ and SEM respectively:

$$\begin{aligned}
EVERY_{k,v} &= \iota\{a\} : o\{b\} : \\
& [\quad [c]_0 = 0, \\
& \quad [a]_t \Leftrightarrow ([c]_{t+v} = [c]_{t+v-1} \oplus_k 1) \\
& \wedge ([a]_t \wedge [c]_t = (k - 1)) \Leftrightarrow [b]_{t+v} \\
&] \\
SEM_w &= \iota\{m, b\} : o\{d\} : \\
& [\quad [s]_0 = 0 \wedge \forall t : ([s]_t < 0) \Rightarrow [\neg b]_t, \\
& \quad [m \wedge \neg b]_t \Leftrightarrow ([s]_{t+w} = 1 + [s]_{t+w-1} \wedge [\neg d]_{t+w}) \\
& \wedge [m \wedge b]_t \Leftrightarrow ([s]_{t+w} = [s]_{t+w-1} \wedge [d]_{t+w}) \\
& \wedge [b \wedge \neg m]_t \Leftrightarrow ([s]_{t+w} = [s]_{t+w-1} - 1 \wedge [\neg d]_{t+w}) \\
& \wedge [\neg(b \vee m)]_t \Leftrightarrow ([s]_{t+w} = [s]_{t+w-1} \wedge [\neg d]_{t+w}) \\
&]
\end{aligned}$$

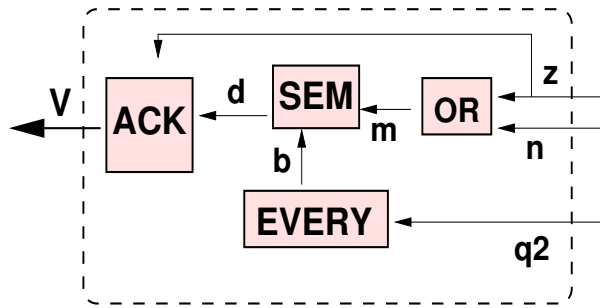


Figure 6.3: TAP process

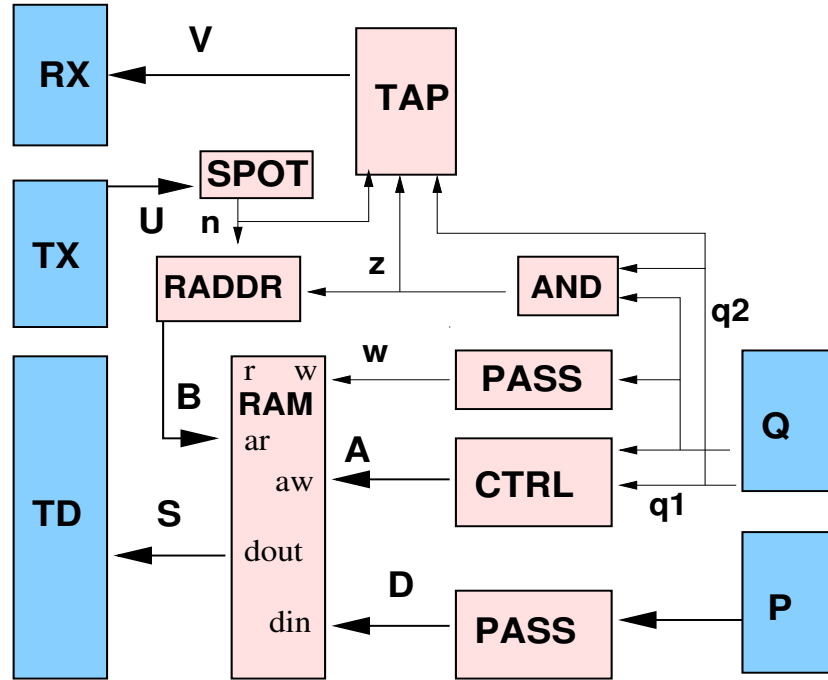


Figure 6.4: MMIO writeback design

$EVERY_2$ can be constructed using a toggle switch TOG where holding the input high over a clock cycle toggles its internal state bit and holding the input low maintains the state:

$$EVERY_2 = AND[a, c][b] \parallel TOG[a][c]$$

$EVERY_{2^k}$ can then be constructed by serial composition of $EVERY_2$ and $EVERY_{2^{k-1}}$. For values of l which are not exact powers of 2 more complicated arrangements are required, such as ring counters.

The MMIO writeback processes are shown in Figure 6.4, with TAP blown up into its components in Figure 6.3.

6.3 Package I/O

Data is passed between packages in the form of packets as described above. There is one significant simplification possible compared to the input buffering; the data to be

sent will be immediately available for writing and immediately able to be received, so there is no general need for the 01 padding packets. The complication is that several packages A, B, C may be sending data to package D simultaneously, requiring arbitration.

When package A needs to send a message to package D we assume that the main package process in A has formatted the message correctly as a sequence of words in the package internal RAM. The main package process A_MAIN will signal to the “talk to D” process A_COMM_D that it may start communicating. It will then expect an acknowledgement signal from A_COMM_D which may either indicate “data sent” or “data sent and answer received”.

The scope of this section is the communication between A_COMM_D and package D.

6.3.1 Arbitration

For each destination package D there is an arbitration process ARB_D which controls access to D from all packages that may communicate with it. Each of the n client packages has an access-request signal in $R = \{r_1, \dots, r_n\}$ and an access-granted signal in $G = \{g_1, \dots, g_n\}$. There is also a set of junction routing signals $S = \{s_1, \dots, s_{n-1}\}$. The function $s : (1 \dots n) \rightarrow \mathbb{P}S$ then describes the set of junction control signals that correctly route each client’s data.

The key criteria are that no more than one client may be granted access at once, and that access, once granted, continues until the client stops requesting it.

$$\begin{aligned}
 ARB_D = \iota R : o(G \cup S) : \\
 [\quad \mathbf{true}, \\
 \quad [r_i]_{t\dots t+j} \Rightarrow \exists k \geq 1 \cdot \\
 \quad ([\neg g_i]_{t+1\dots t+k-1} \wedge [g \wedge s(i)]_{t+k\dots t+j+1}) \\
 \wedge [g_i \wedge g_j]_t \Rightarrow (i = j) \\
]
 \end{aligned}$$

6.3.2 Inter-package routing

The routing of data between packages is managed as shown in Figure 6.5. This has packages A, B, C and D routing data to descendant package E. Each junction routing signal goes to a junction package J_i which multiplexes data from the client packages onwards to the destination package, and demuxes the return data to the client package.

For the client-destination data flow we name the input data sets W_1, W_2 and the output data set X . For the destination-to-client data flow we name the input data set Y and the output data sets Z_1, Z_2 . Event s is the routing switch: when off it routes W_1, Z_1 through and when on it routes W_2, Z_2 through.

The process, for k -bit wide data sets, is then specified by:

$$\begin{aligned}
 J_i = \iota(W_{1,2} \cup Y \cup \{s\}) : o(X \cup Z_{1,2}) : \\
 [\quad \mathbf{true}, \\
 \quad \forall 1 \leq i \leq k \cdot \\
 \quad ([(w_{1i} \wedge \neg s) \vee (w_{2i} \wedge s)]_t \Leftrightarrow [x_i]_{t+1})
 \end{aligned}$$

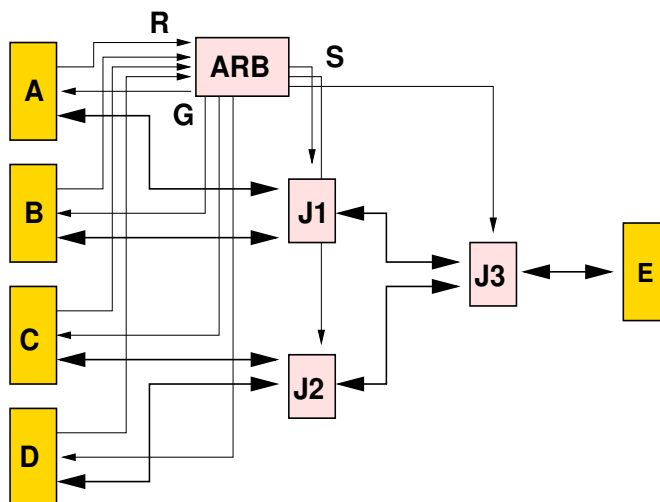


Figure 6.5: Inter-package routing

$$\wedge \quad ([y_i]_t \Leftrightarrow [(z_{1i} \wedge \neg s) \vee (z_{2i} \wedge s)]_{t+1})$$

$$\quad]$$

6.3.3 Package output

The process A_COMM_D will read in data from RAM, packetise it and send it out onto the bus in a similar manner to process $SIGNAL$ described in Equation 6.3. The difference is that the data will be read by requesting a serial copy from RAM, with the port data width set at design time to match the bus data width, hence no buffering or change signalling is needed.

A_COMM_D is split into the following processes:

WAIT Waits for the start signal from the package, requests the granted signal from ARB , then keeps the request active until the result has been received from the package.

$IHDR_w$ Waits for the granted signal from ARB , then kicks off the serial read from RAM and writes out the correct header bits to the bus. Parameter w is the number of packets of output data.

IDATA Continually copies data across from the RAM port to the data bits of the bus.

OHDR Keeps a watch on the header bits coming back from the destination package, starts a serial write to RAM, and once concluded signals the finish pack to the package.

ODATA Continually copies data across from the data bits of the bus to the RAM port.

Let the RAM read interface be input event s_1 to start a read, output set $D_1 = \{d_{11}, \dots, d_{1k}\}$ of data. We ignore any signal that the read is complete since we already know the message size at compile time.

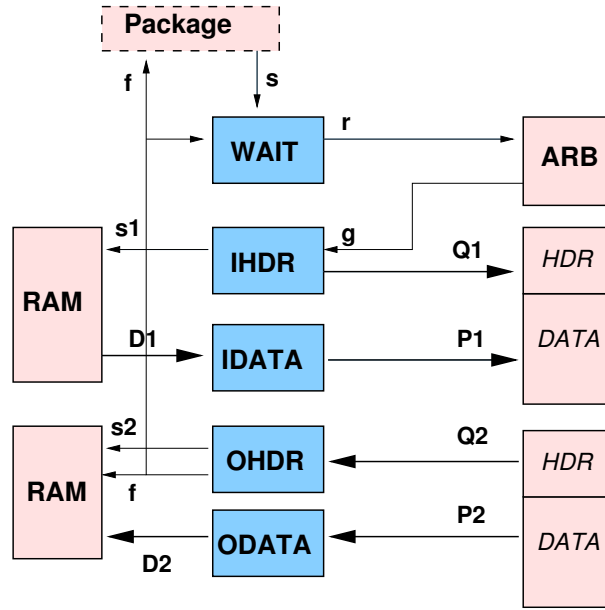


Figure 6.6: Package output

The RAM write interface similarly is input event s_2 to start a write, input set $D_2 = \{d_{21}, \dots, d_{2k}\}$ of data and input event f to signal that the write is complete.

For the outside arbitration, let r be the arbitration request (which needs to be held high during the request, writing and returning read) and g be the access granting event.

For the bus output, let $Q_1 = \{q_{11}, q_{12}\}$ be the packet marker bits and $P_1 = \{p_{11}, \dots, p_{1k}\}$ be the packet data bits. Similarly the bus input is $Q_2 = \{q_{21}, q_{22}\}$ for the packet marker bits and $P_2 = \{p_{21}, \dots, p_{2k}\}$ for the packet data bits.

The interface to the rest of the package is input signal s for the “start a broadcast” request and an output signal f for the “communication finished” acknowledgement.

These processes are illustrated in Figure 6.6. $IDATA$ and $ODATA$ are simply $PASS_k$ processes. $IHDR_w$ has already been defined in Equation 6.4. The other two processes are specified as follows:

$$\begin{aligned}
WAIT &= \iota\{s, f\} : o\{r\} : \\
&[\text{true}, \\
&([s]_t \wedge \exists i : [\neg(s \vee f)]_{t+1\dots t+i} \wedge [f]_{t+i+1}) \Leftrightarrow \\
&([r]_{t+1\dots t+i+1} \wedge [\neg r]_{t+i+2}) \\
&] \\
OHDR &= \iota\{q_1, q_2\} : o\{s_2, f\} : \\
&[[q_{1,2}]_t \Rightarrow \exists i : \forall 1 \leq j \leq i \cdot \\
&([q_2 \wedge \neg q_1]_{t+1\dots t+i} \wedge [\neg q_{1,2}]_{t+i+1}), \\
&([q_{1,2}]_t \wedge \exists i : [q_2]_{t+1\dots t+i} \wedge [\neg q_2]_{t+i+1}) \Leftrightarrow \\
&[s_2 \wedge \neg f]_{t+1} \wedge [\neg(s_2 \vee f)]_{t+1\dots t+i+1} \wedge \\
&[f \wedge \neg s_2]_{t+i+2} \\
&]
\end{aligned}$$

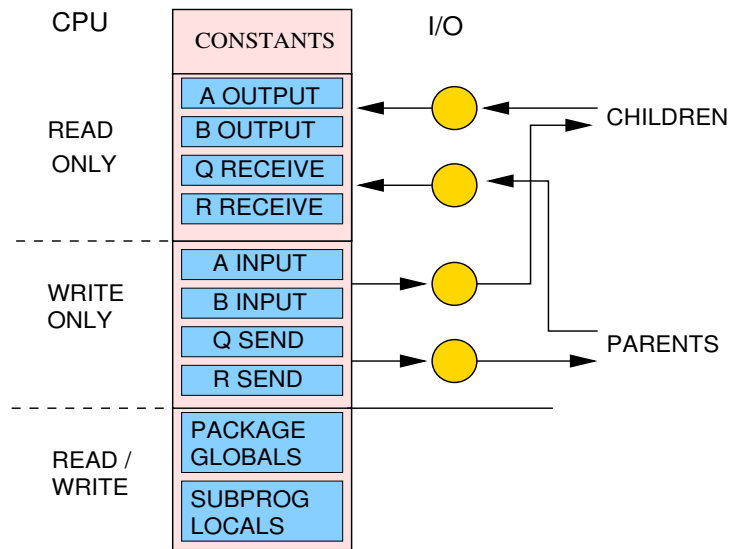


Figure 6.7: Package RAM layout

6.3.4 Package input

The input in destination package D uses a mirror of the above structure to receive the packeted data and write it into RAM.

In addition, it will need a *START* process to set the initial PC value according to the start ID that heads the data stream, and then monitor the PC store for when the last PC value is popped off the stack indicating subprogram termination. It must then kick off the return transmission of the data from the area of RAM storing the subprogram mode out parameters.

6.4 Package Structure

We have described in detail the mechanism for sending data between packages. We now look at the details of the implementation of the package units.

6.4.1 Storage

Key to the operation of each package P are the internal ROM and RAM stores. The ROM contains the compiled SPARK from the original package subprograms. The RAM contains all the constant data used in the package (initialised when the PLD program is loaded), areas for data to send to and receive from inherited packages, areas for data to receive from and send to packages that inherit P , all the package global variables, and all variables declared in all subprograms of the package. The last group includes the subprogram parameters and function return values.

Figure 6.7 is an example of RAM layout for a package P that inherits A and B and is inherited by Q and R .

RAM has two main parameters: the bit width of each word in it, and the number of words held in RAM. These can be determined at the interpreter's compile time and will depend on the variables in the SPARK package. Wide words will speed up transfer of

large amounts of data at the cost of wasted RAM space when many sub-word variables (e.g. booleans) are stored.

6.4.2 Storage operations

The RAM blocks must be able to implement serial reads and writes of data as well as individual reads and writes. For the package I/O work we have already seen the serial interfaces required for effective communication.

CPU access to RAM will be managed by a RAM controller $RCTRL$. This must be able to implement the following operations:

1. serial read of N words starting to read from address A ;
2. serial write of N words starting to write at address B ; and
3. internal copy of N words, starting to read at address A and starting to write at address B .

The reason for the final operation is that both subprogram calls and plain assignments (i.e. with a variable or constant as the rvalue rather than an expression) are effectively copy requests. In our I/O model, subprogram calls require mode **in** variables to be copied into the RAM slot representing the subprogram parameter, whether in this package or in an inherited package, and mode **out** variables to be copied back out. Doing this copying as a basic RAM operation is an efficiency measure.

With the following event namings, RAM width l , RAM word count 2^k and RAM state functions $s_r : \mathbb{N} \rightarrow \mathbb{P}R$, $s_w : \mathbb{N} \rightarrow \mathbb{P}W$ we can specify $RCTRL_{k,l}$. We use the abbreviation that $[A]_t$ means “the subset of A events present at time t ”.

$$\begin{aligned}
A &= \{a_1, \dots, a_k\}, a : \mathbb{P}A \rightarrow \mathbb{N} \\
B &= \{b_1, \dots, b_k\}, b : \mathbb{P}B \rightarrow \mathbb{N} \\
N &= \{n_1, \dots, n_k\}, n : \mathbb{P}N \rightarrow \mathbb{N} \\
R &= \{r_1, \dots, r_l\} \\
W &= \{w_1, \dots, w_l\} \\
C &= \{c_1, c_2\} \\
RCTRL_{k,l} &= \iota(A \cup B \cup N \cup W \cup C) : o(R \cup \{d\}) : \\
&[\quad ([c_1 \vee c_2]_t \wedge \exists i : [\neg d]_{t+1\dots t+i}) \Rightarrow [\neg c_{1,2}]_{t+1\dots t+i}, \\
&\quad [c_1]_t \Leftrightarrow \forall 0 \leq j < [n]_t \cdot \\
&\quad ([R]_{t+j+2} = [s_r(j + [A]_t)]_{t+j}) \wedge ([s]_{t+j+2} = [s]_t) \\
&\wedge [c_2 \wedge \neg c_1]_t \Leftrightarrow \forall 0 \leq j < [n]_t \cdot \\
&\quad [s]_{t+j+1} = [s]_{t+j} \oplus ((j + [B]_t) \mapsto [W]_{t+j})) \\
&\wedge [c_{1,2}]_t \Leftrightarrow \forall 0 \leq j < [n]_t \cdot \\
&\quad [s]_{t+j+2} = [s]_{t+j+1} \oplus ((j + [B]_t) \mapsto [s(j + [A]_t)]_{t+j})) \\
&\wedge [c_1 \vee c_2]_t \Leftrightarrow ([\neg d]_{t+1\dots t+[n]_t-1} \wedge [d]_{t+[n]_t}) \\
&]
\end{aligned}$$

Bits	Meaning
00	Do nothing
01	Push the PC value on N onto the top of the PC stack
10	Pop the top item off the PC stack
11	Change the top item on the PC stack to the value of N

Table 6.3: PC action encodings

C are the control bits selecting the operation. c_1 represents a serial read, c_2 represents a serial write and the two C events together represent a copy. The above specification, by delaying read output by one cycle, allows for an extra step whereby the implementation may treat a copy like a read and a write in parallel, but internally route the read output into the RAM write data port instead of routing the W events.

d is the “operation complete” bit. The precondition states that, once a command is given, no further commands are given until the “operation complete” is signalled.

6.4.3 Program storage

There are three components to manage program storage. The ROM itself stores the compiled SPARK code in fixed-width words. Data is read out by the program counter process PC_k . This maintains a current PC value, but can also store up to k other PC values in a stack for use when there are internal subroutine calls. The stack size can be bounded at compile time because SPARK’s ban on recursion means that the longest subprogram chain can be statically determined and in any case is no longer than the total number of subprograms.

PC_k outputs the PC to ROM with events $P = \{p_1, \dots, p_k\}$. It has input events for a new PC value $N = \{n_1, \dots, n_k\}$, PC increment request i , and control input events $C = \{c_1, c_2\}$ with the encodings shown in Table 6.3.

$SNIP$ waits for event r to command a read. It then checks the instructions coming out of the ROM, signalling i each time to get the next piece of data, and when the end of an instruction arrives stops signalling i . The data is output using the event set $Q = \{q_1, \dots, q_y\}$ where y is the standard instruction set data item width. Since event d is the negation of i , d will then be signalled back to the CPU.

We specify PC_k as follows. We will hold over the definition of $SNIP$ until Section 6.4.5 when we specify the instruction set. We define functions $n()$ and $p()$ to translate the input and output counters into \mathbb{N} . We also define state functions $c : \mathbb{N}$ and function $pc : \mathbb{N} \mapsto \mathbb{N}$ so that $pc(j)$ gives the program counter at location j in the stack, with the top of the stack at location c .

$$\begin{aligned}
PC_k &= \iota(\{i\} \cup N \cup C) : oP : \\
&[[c_1 \vee c_2]_t \Rightarrow \neg[i]_t, \\
&[\neg(c_1 \vee c_2)]_t \Leftrightarrow \\
&([c]_{t+1} = [c]_t) \wedge ([pc]_{t+1} = [pc]_t) \\
&\wedge [c_1 \wedge \neg c_2]_t \Leftrightarrow \\
&([c]_{t+1} = [c]_t + 1) \wedge ([pc]_{t+1} = [pc]_t \oplus ([c]_{t+1} \mapsto [n]_t)) \\
&\wedge [c_2 \wedge \neg c_1]_t \Leftrightarrow
\end{aligned}$$

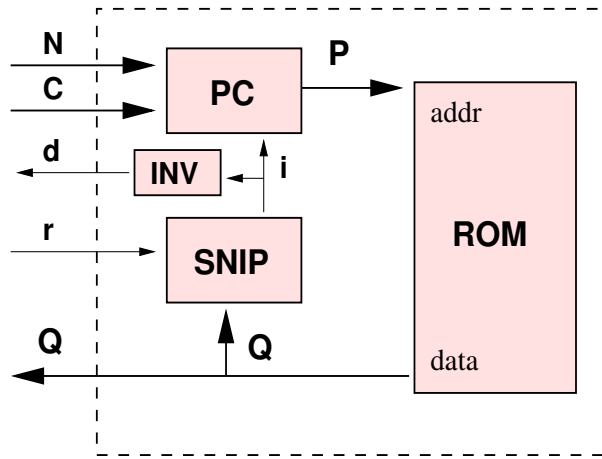


Figure 6.8: ROM and PC store

$$\begin{aligned}
 & ([c]_{t+1} = [c]_t - 1) \wedge ([pc]_{t+1} = [pc]_t) \\
 \wedge & [c_{1,2}]_t \Leftrightarrow \\
 & ([c]_{t+1} = [c]_t) \wedge ([pc]_{t+1} = [pc]_t \oplus ([c]_t \mapsto [n]_t)) \\
 \wedge & [i]_t \Leftrightarrow \\
 & ([c]_{t+1} = [c]_t) \wedge ([pc]_{t+1} = [pc]_t \oplus ([c]_t \mapsto 1 + [pc(c)]_t)) \\
 &]
 \end{aligned}$$

Figure 6.8 shows the relations between the program storage components.

6.4.4 Expression evaluation

Expression evaluation is managed by custom expression blocks. These take a stream of data bits as input and produce a stream of data bits as output. There may be any number of expression blocks in a package.

Control of the process is managed by the write-data signal w and the process-data signal g . w sets the data in RAM to the input D . This data is output in the next step to the event set A . A multiplexer MUX routes the w and A events to the expression block selected by the events E from the CPU.

The expression blocks themselves take a “start” signal g_i and input data set A_i as inputs, and give out a “finished” signal h_i and output data set B_i . Their implementation will depend on the particular expression. The developer may choose to design them by hand in order to take advantage of PLD features.

Once the expression block has finished, it signals h_i and outputs its data on B_i . This is routed to the output RAM via $DMUX$, again controlled by E .

Note that the other components are constructs which we have come across before and do not require specification. Figure 6.9 shows how the expression evaluation components fit together.

6.4.5 CPU instructions

The instructions from the ROM, previously referred to, are fed by the program store to the CPU. They are key to the control path of the package.

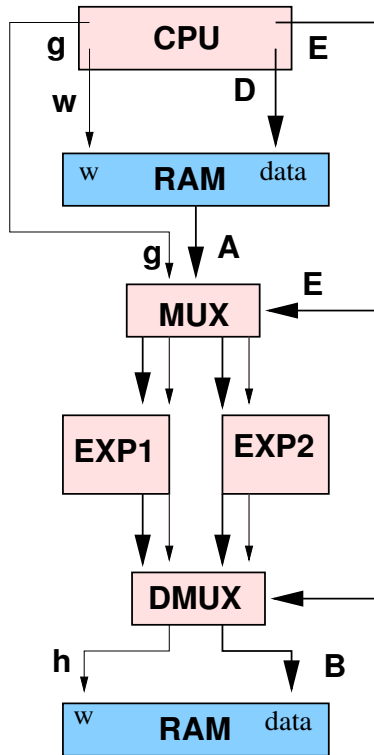


Figure 6.9: Expression blocks

Bits	Meaning
00	End of instruction (no-op)
01	Continuation of instruction sub-component
10	Start of instruction sub-component
11	Opcode

Table 6.4: Word type encodings

An instruction consists of a sequence of y -bit words, fed to the CPU from the program store with event set Q . The sequence will always start with an opcode, which may then be followed by any number of sub-components such as addresses and data.

This section breaks down the decoding and execution process of the instructions.

Encoding scheme

Each word in ROM uses its top two bits to indicate the type of the data in it, according to Table 6.4.

The end of each complete instruction is signalled by a word with zero headers bits.

We can now specify the *SNIP* process from Section 6.4.3. It need only check the top two bits of the data coming through. As long as they are not both 0 it will continue to raise event i , incrementing the PC to get the next part of the instruction. As soon as they are both 0 it will cease to signal i .

$$\begin{aligned}
 SNIP &= \iota(Q \cup \{r\}) : o\{i\} : \\
 &[\mathbf{true}, ([r]_t \vee [q_0 \vee q_1]_t) \Leftrightarrow [i]_{t+1}]
 \end{aligned}$$

Opcode	Arguments	Meaning
NOP	–	Do nothing
DEPON	$id : E$	Instruction depends on id
COPY	$l : D, s, d : A$	Copy l words from s to d
LOGIC	$op : E, \{ a_i \}$	Evaluate logic operator
CMP	$op : E, a, b : A,$ l	Compare a and b of length l
SUBEXT	$id : E$	Subprogram id call
LOOP	–	New loop marker
LPEXIT	$a : P$	Exit current loop, skip PC to a
LPRET	–	Return to loop start
IFELSE	$\{ a_i : A, p_i : P \}$	if-then-elsif
EXEVAL	$id : E$	Evaluate expression id
EXWRT	$l : D, s : A$	Copy l words from s in RAM to expression input block
EXREAD	$l : D, d : A$	Copy l words from the expression output block to d in RAM
SUBJMP	$a : P$	Jump to local subroutine at a in ROM
SUBRET	–	Return from subroutine, restoring PC
IDXRD	$s : A, i : A, t, l :$ $D, d : A$	Indexed read from s to d
IDXWRT	$s : A, i : A, t, l :$ $D, d : A$	Indexed write from s to d

Table 6.5: CPU Opcodes

]

Instruction Opcodes

Table 6.5 lists the possible opcodes for the ROM instructions. The ROM output must be wide enough for each instruction to be identified uniquely in one word, including the aforementioned two header bits.

In the table, A denotes an address in RAM, P a program counter value, E an enumeration and D a data chunk. $\{ X \}$ denotes one or more instances of X .

Each opcode has a condition flag bit C . If set, the opcode is only executed if the current CPU condition flag is set.

Not all these opcodes need be implemented. At compile time, if an opcode is not present in the compiled program then it and its associated components need not be put into the package.

Dependencies

The first opcode of a message may be DEPON. The following word gives the ID of an instruction which must complete before the current instruction can start. This is useful for starting a long operation (e.g. external subprogram call), processing data in the meantime, but having the facility to block when the long operation's data is required

but unavailable. Any number of instances of this opcode and its data may be present at the head of a instruction.

The set of dependency IDs are mapped to the different operation blocks in the CPU core, therefore correspond to an instruction's opcode.

If the main opcode has its conditional bit set then the conditional bit in DEPON must be set too. This will have the effect of throwing away the entire instruction before any dependencies are checked.

The DEPON opcodes and their data are followed immediately by a normal instruction and its data.

Opcode Descriptions

COPY is a direct command to the RAM to copy l words from address s to address d . We have already seen that our RAM components implement this directly.

LOGIC takes an operand identifier op , which selects an n-ary logic operator, and applies it in sequence to the data at the specified RAM addresses. The result (**true** or **false**) is assigned to the CPU conditional flag.

CMP takes an operand identifier op which selects one of the six numeric comparators =, /, <, <=, >, >=, two addresses a, b identifying variables and a word count l . The variables are evaluated against each other as if they were unsigned integers of the appropriate length. The result is assigned to the CPU conditional flag.

SUBEXT calls a subprogram external to the package. id identifies the destination package and the necessary message header data.

LOOP identifies the start of a new loop. It pushes the current program counter onto the PC stack, leaving the stack topped with duplicate values v . v will then be the PC address of the first instruction in the loop.

LPEXIT jumps out of the loop by popping the top value off the PC stack and then setting the current PC value to a .

LPRET returns to the top of the loop by popping the top value off the PC stack, reading the next PC value, and pushing it back onto the PC stack to have duplicate values as in LOOP.

IFELSE takes a string of RAM and PC addresses. If the value at a_1 is non-zero then the PC jumps to p_1 . Otherwise the value at a_2 is examined, and so on. A catch-all else can be implemented by specifying the last a_n as the address of a non-zero constant.

EXEVAL uses id to select routing to the expression blocks as described in Section 6.4.4, then signals the expression block to evaluate the current set data.

EXWRT reads l words from address s in RAM and writes them into the expression input RAM block.

EXREAD reads the expression output RAM block and writes the l words into address d in RAM.

SUBJMP is an internal subroutine jump. It pushes the specified PC value a onto the PC stack, making the CPU execute instructions from a onwards.

SUBRTN returns from the internal subroutine by popping the top value off the PC stack.

IDXRD reads the number from address i , multiplies by t words and adds to s before reading l words from the resulting address and copying them to d .

IDXRD reads l words from s then reads the number from address i , multiplies by t words and adds to d before copying the read words to the resulting address.

6.4.6 Instruction decoder

The first stage of the pipeline coming out of the ROM store carrying the instructions handles dependency stalls and opcode selection. At the end of this stage of the pipeline the instruction's dependencies have been met, conditional instructions have been checked and dropped if the condition is not met, and the instruction's data (if any) multiplexed to the correct control unit with an activation signal.

For clarity we define a “valid instruction form” function $[m(A)]_{t\dots t+i}$. This is read as “the message formed by the events in A from time t through time $t + i$ is a valid instruction according to the restrictions in this section”. There are corresponding functions $m'()$ and $m''()$ which respectively describe messages without leading dependency lists and with a dependency ID moved to the back.

The conditionals are checked first. *COND* takes lines Z out of the ROM store and CPU conditional status line v out of the CPU core as inputs. If v is clear (meaning “last CPU condition evaluated to **false**”) and an opcode pattern in Z is conditional then the rest of the instruction is thrown away, up to the first word with header bits 00. Additionally signal u is sent to the CPU core, meaning “conditional instruction not executed” so that the core can request the next instruction from ROM.

The specification of *COND* assumes that the conditional bit of an opcode is bit c .

$$\begin{aligned}
 COND &= \iota(Z \cup \{v\}) : o(A \cup \{u\}) : \\
 &[\quad [m(Z)]_{t\dots t+i}, \\
 &\quad ([z_{0,1,c}]_t \wedge [\neg v]_t) \Leftrightarrow ([u]_{t+1} \wedge [\neg a_{0,1}]_{t+1\dots t+i+i}) \\
 &\wedge ([v]_t \vee \neg[z_{0,1,c}]_t) \Rightarrow \\
 &\quad [m(A)]_{t+1\dots t+i+1} \wedge \forall 1 \leq j \leq (i+1) \cdot \\
 &\quad [\{a_0, \dots, a_{n-1}\}]_{t+j} = [\{z_0, \dots, z_{n-1}\}]_{t+j-1} \\
 &]
 \end{aligned}$$

The decoder takes the lines A out of *COND*, where $a_{0,1}$ are the header bits, and processes the dependency stalls in *DEPCODE*. This outputs unstalled data to *DECID* along lines B . Any required dependency is queried along lines P , and signal p is received once the dependency is satisfied.

The new decoding processes are specified as follows. We define the function $opcode_{A,P}(X)$ to map the subset X of A to the equivalent opcode encoding in P , ignoring the conditional flag.

$DEPCODE_N$ is parametrised by the length of an internal buffer which it uses to store a blocked message. The precondition of $DEPCODE_N$ states that there is some number $k \geq 0$ of dependencies in front of each message, and the message less its dependencies is still a valid message.

The postcondition states that if there are no dependencies then it is passed straight through in $N+1$ steps; if there are dependencies then it must signal the first dependency ID, then a p event must occur before any message can be passed through.

$$DEPCODE_N = \iota(A \cup \{p\}) : o(B \cup P) :$$

$$\begin{aligned}
& [\quad [m(A)]_{t\dots t+i} \wedge \exists k : \forall 0 \leq j \leq k : [m(A)]_{t+2j\dots t+i}, \\
& \quad ([a_{0,1}]_t \wedge \text{opcode}_{A,P}([A]_t) = \text{DEPON}) \Rightarrow \\
& \quad \quad [\text{opcode}_{A,P}([A]_{t+1})]_{t+N+1} \wedge \\
& \quad \quad ([\neg p]_{t+N\dots t+N+d} \wedge [p]_{t+N+d+1}) \Rightarrow \\
& \quad \quad \quad [\neg b_{0,1}]_{t+N+1\dots t+N+d} \wedge \\
& \quad \quad \quad \forall s : ([m'(B)]_s \wedge s > (t+N)) \Rightarrow (s > t+N+d+1) \\
& \wedge \neg([a_{0,1}]_t \wedge \text{opcode}_{A,P}([A]_t) = \text{DEPON}) \Rightarrow \\
& \quad [m'(B)]_{t+N+1\dots t+N+1+i} \wedge \\
& \quad [\{b_0, \dots, b_{n-1}\}]_{t+N+1\dots t+N+1+i} = \\
& \quad \quad [\{a_0, \dots, a_{n-1}\}]_{t\dots t+i} \\
&]
\end{aligned}$$

DECID registers the dependency id, then passes the instruction along *C* to the opcode decode *OP*.

$$\begin{aligned}
\text{DECID} &= \iota B : o(C \cup D) : \\
& [\quad [m'(B)]_{t\dots t+i}, \\
& \quad [b_{0,1}]_t \Leftrightarrow [\text{opcode}_{B,D}([B]_t)]_{t+1} \\
& \quad \wedge (\neg[b_{0,1}]_t) \Leftrightarrow [\text{opcode}_{B,D}(\emptyset)]_{t+1} \\
& \quad \wedge [\{c_0, \dots, c_{n-1}\}]_{t+1} = [\{b_0, \dots, b_{n-1}\}]_t \\
& \quad \wedge [m''(C)]_{t+1\dots t+i+1} \\
&]
\end{aligned}$$

OP produces the control lines *E* and data lines $F \cup \{s\}$ routed into the multiplexer. The *s* event is the “operation start” signal. The multiplexer then routes the data along the correct route G_i to the operation’s particular processing block.

The *OP* process has a relatively simple specification. The condition on $[c_{0,1}]_t$ is not strictly needed as it is implied by the precondition of a well-formed message, but it aids clarity. Here we specify OP_k with a delay $k \geq 1$.

$$\begin{aligned}
OP_k &= \iota C : o(E \cup F \cup \{s\}) : \\
& [\quad [m''(C)]_{t\dots t+i}, \\
& \quad [c_{0,1}]_t \Leftrightarrow [s]_{t+k} \\
& \quad \wedge [\{e_0, \dots, e_{n-3}\}]_{t+k\dots t+i+k} = [\{c_2, \dots, c_{n-1}\}]_t \\
& \quad \wedge [\{f_0, \dots, f_1\}]_{t+k} = \{0, 0\} \\
& \quad \wedge \forall 1 \leq j \leq i : \\
& \quad \quad [\{f_0, \dots, f_{n-1}\}]_{t+j+k} = [\{c_0, \dots, c_{n-1}\}]_{t+j} \\
&]
\end{aligned}$$

Process *REG* acts as a dependency register. Input lines *D* are used to note that an instruction with ID d_i has gone through. Input lines *H* are used to note that CPU block h_i has completed. *REG* has the internal state function $b : \mathbb{N} \rightarrow \mathbb{B}$ identifying whether each instruction ID has gone through and not been acknowledged. State variable $l : \mathbb{N}$ stores the last dependency ID query received through *P*.

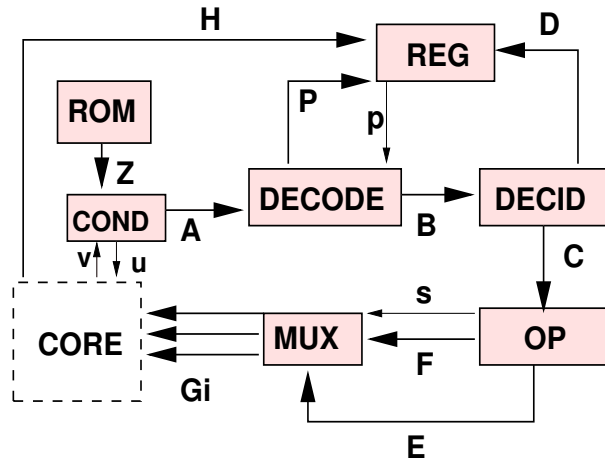


Figure 6.10: First stage of CPU pipeline

REG_k is parameterised by its delay $k \geq 1$.

$$\begin{aligned}
 REG_k &= \iota(P \cup H \cup D) : o\{p\} : \\
 &[\quad \mathbf{true}, \\
 &\forall i : [h_i]_t \Rightarrow \\
 &\quad [s]_{t+k} = [s \oplus (i \mapsto \mathbf{false})]_t \wedge \\
 &\quad ([l]_t = i) \Rightarrow [p]_{t+k} \\
 &\wedge [d_i]_t \Rightarrow \\
 &\quad [s]_{t+k} = [s \oplus (i \mapsto \mathbf{true})]_t \wedge \\
 &\quad [l]_{t+k} = i \\
 &]
 \end{aligned}$$

Figure 6.10 shows the decoder pipeline flow.

6.4.7 CPU implementation

Specifying each component of the CPU core here would be a laborious process. We have already made formal specifications for the key components with which they communicate. In this section we outline the generic method of operation.

A core component receives data from the decoder multiplexer. Start of data is signalled by a high on start wire d . The data itself comes in standard packets (with header bits 10 for start of item and 01 for item continuation) on wires $G = \{g_0, \dots, g_{n-1}\}$.

The input data will typically be stored in one or more small blocks of RAM or in flip-flops, as required. As each part of the instruction is received the component will change state to route the next instruction part appropriately. The actual computation may involve communication with RAM, the ROM store, the PC store or external package interfaces.

The dependency analysis in the compiler and decoder guarantees that the component will not be in the middle of computation when the new data arrives. It does require that the component signal on output h once the computation is complete and all output data has been sent to the appropriate destination. This signal h will end up at the REG process.

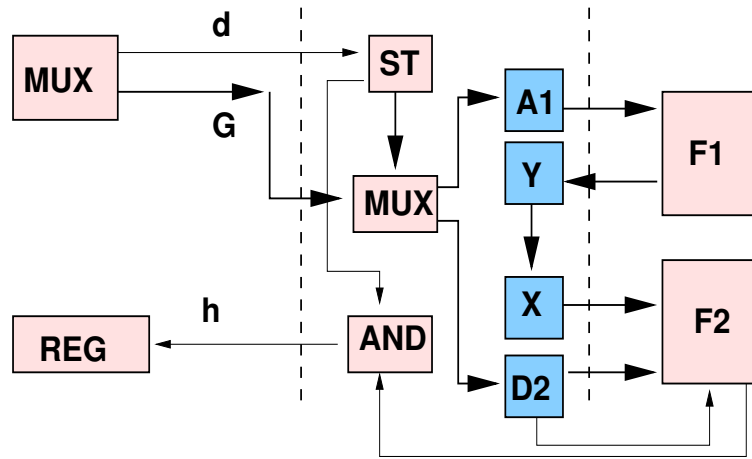


Figure 6.11: CPU core component

Figure 6.11 illustrates a generic CPU core component. State machine *ST* receives the start signal and controls the routing of *MUX* to direct the instruction packets to the appropriate destinations. Block *A1* builds up an address and does a read request of external block *F1*, which might for instance be RAM or the PC store. Data is eventually returned into *Y*, which processes it and passes to *X*, where it is used as the address for a write to external block *F2* of the data built up in *D2*. Completion is signalled once *ST* has seen the end of the message and block *F2* has signalled success.

6.4.8 Opcode summary

With the preceding work we have produced a substantial and detailed design for an interpreter of the instructions defined in Section 6.4.5. The interpreter contains mechanisms for asynchronous communication with client software and for synchronous communication between component packages. The design has a range of parameters relating to processing delay and bit width of communication channels, and allows removal of components which are not needed for a particular program.

We assert that this interpreter is suitable for running a compiled version of a sequential SPARK 95 program. To demonstrate this, in Section 6.5 we detail how the SPARK Ada constructs are mapped to sequences of interpreter instruction codes. We also show how the control- and data-flow properties of a SPARK program validate a range of assumptions made in the interpreter design.

6.5 The Program Model

In Section 4.3 we described the SPARK Ada language and the SPARK Examiner enforcing tool. We now describe how to map valid SPARK programs into the interpreter that we have defined.

For the rest of this chapter, “a SPARK program” should be taken to mean “a sequential Ada 95 program which conforms to the SPARK language definition and which is free of exceptions.” In practice this normally means a program which passes the SPARK 95 Examiner checks, and for which the run-time checks generated by the Examiner (using the `-exp` switch) are all proven free of exceptions.

There are legal SPARK programs not allowed by the Examiner, and also illegal SPARK programs allowed by the Examiner. Clearly, the latter are potentially serious if part of a safety-critical system, since the Examiner's acceptance may lead to undue trust of the program. However, the known cases of this problem over the years of commercial Examiner use have been relatively small in number. Each project's safety authority will have to make their own judgement on the reliability of the Examiner.

6.5.1 Types

Basic SPARK types are subsets of integers, fixed-point or floating point numbers, and characters. Enumerated types can be viewed as integers where no arithmetic is normally performed.

Compound SPARK types use the array constructor, with integers or enumerated types as indices and any other types as the element type, or the record constructor, with field names as indices.

The interpreter has one form of type: a sequence of a fixed number of words. Word size is fixed within a package. The sequence length depends on the original SPARK type, and in the case of compound types will normally be the sum of the lengths of the components of the component types. Basic types will be stored in a non-negative integer number of words within a package.

As an example, the SPARK basic types:

```
type N is range 1..300;
type E is (Red, Amber, Green);
```

are represented in a package with word length of four bits by sequences of three and one words respectively.

The SPARK compound types:

```
type R is record
  A : N;
  B : E;
end record;
type A is array(N,E) of R;
```

have sequences of 4 and $(300 \times 3) \times 4 = 3600$ words respectively. A "slice" of the array A, selected by the first index only, would be $3 \times 4 = 12$ words.

The remaining difficulty is in indexing into a compound type. Indexing into a record is easy since all record fields and sizes are known at compile time, so the offset and length of the component are known. Indexing into an array is more difficult since the index is not generally determined at compile time.

The interpreter `IDXRD` and `IDXWRT` instructions are designed to allow this. Multiple indices require multiple uses of the instructions.

As an example, if the variables `X : A`; `Y : N`; `Z : E` were stored at locations a_0, a_1, a_2 in RAM, then the assignment `W := X(Y,Z)`; would be accomplished by the following sequence:

```
IDXRD a0 a1 12 4 a4 # store X(Y) in T
IDXRD a4 a2 3 4 a3 # store T(Z) in W
```

where W was stored at location a_3 and a_4 held a temporary variable of size `array(E)` of `R`.

Given this, we see that we have the mechanism for determining type size in package words at compile time. Differing word sizes between packages are irrelevant since the inter-package pipeline transports data at a packet size independent of source and destination word sizes.

6.5.2 State

SPARK program variables are made visible either when a package is elaborated, or when a subprogram is called. Each variable is associated with a named type (see above) and may have an initialised value. Each variable has a unique fully-qualified name.

Variables in the interpreter have a fixed location in the RAM of their containing package. All of their locations and word sequence lengths are determined at compile time. The safest strategy makes all variables disjoint. SPARK's ban on recursion removes the need for a dynamic variable stack.

However, it is possible to optimise RAM usage by allowing certain subprogram variables to overlap. Variables from subprograms `P` and `Q` can overlap if there is no subprogram calling sequence which allows `Q` to be called directly or indirectly from `P` or vice versa.

All package variables are initialised at interpreter programming time, to 0 unless an explicit initialisation is given in the SPARK. The SPARK Ada rules remove the elaboration order problems with Ada and allow package variable initial values to be determined during static analysis.

Subprogram variables which are initialised at declaration must be explicitly initialised at the start of the compiled version of the subprogram. Space must also be allocated for subprogram parameters, both `in` and `out`.

Constants are treated as variables but placed in the section of RAM which is read-only to the package core.

6.5.3 Expressions

A SPARK expression combines variables, function calls and literals to produce an output of a type that is known at static analysis time. Expressions are either static (can be determined at compile time) or non-static. We shall ignore static expressions since they will be reduced to literals at compile time.

Expressions may occur in the following places:

1. on the right hand side of an assignment or declaration;
2. as an input parameter in a subprogram call;
3. as an index in a component reference (e.g. an array);
4. as the selector in a `case` statement;
5. after the `return` at the end of a function;

6. within a type conversion; or
7. after an `if` or `elsif`, as a boolean condition.

The simplest expression is a numeric literal. This will be stored in RAM in the same way as a declared constant. Note that the type of the literal is known at compile time, so `5 : range 0..7` and `5 : range 0..9999` will be stored in different locations since they are “different 5s”.

Another simple expression is a single variable or constant name, possibly with record selectors following. This can be handled entirely within RAM by the `COPY` instruction since the variables’ addresses and length are known immediately.

A more complicated expression is a variable or constant with one or more array selectors. In Section 6.5.2 we saw that the `IDXRD` could be used to emulate this. Similarly `IDXWRT` can be used to emulate assigning to an array-selected component of a variable.

Boolean expressions (such as those after `if` statements) consist of one or more boolean sub-expressions separated by logic operators. The `LOGIC` instruction provides a shortcut to evaluating n -ary boolean logic, and additionally sets or clears the CPU conditional flag which we will later find useful.

Numeric comparisons are done with `CMP` which works in a similar way to `LOGIC`. It can also meaningfully compare two variables (of the same type) for equality or non-equality.

Type conversions are not trivial, since they may move data between word sizes. SPARK’s run-time exception checks ensure that the conversion is always valid (5 can never be converted to a variable of range 0 . . . 4, for example), but the conversion itself is an arithmetic problem.

Expressions may incorporate function calls, but the SPARK rules mean that there are no side effects (the functions do not change the values of any variables) and all variables used by the function, directly or indirectly, are known. These expressions will be rearranged by the compiler so that the function call occurs first, saving data to a temporary variable, then that variable replaces the function in the expression.

Expressions within subprogram calls will need to be saved to the variable corresponding to the appropriate subprogram parameter.

Arithmetic expressions are difficult. The general solution is the use of expression evaluation blocks within the CPU; each arithmetic expression in a package subprogram will normally need its own block. The `EXxxxx` instructions allow writing to, execution of and reading from these blocks.

The logic for an expression block may be produced automatically by the compiler; the normal Ada arithmetic and logical operations will have a library of blocks pre-defined, parametrised by argument type size. We have already seen an adder; other arithmetic blocks can be produced using well-understood programmable logic designs.

Alternatively the developer may choose to produce a manual design, refining the required specification in the process described in Chapter 5. This may confer performance and space benefits, at the cost of increased development time and chance of error in the refinement.

6.5.4 Alternation

There are two SPARK forms of iteration: `if-then-elsif-else` and `case`. The latter can be treated as a special case of the former.

The `IFELSE` instruction is the key to emulating alternation. It contains a list of boolean variable addresses paired with PC values to jump to. A terminating “else” can be emulated with the address of a constant Boolean `true`. The set-up to `IFELSE` will normally be a series of expression and boolean evaluations matching the various conditions.

6.5.5 Iteration

The interpreter supports loops with the `LOOP`, `LPEXIT` and `LPRTN` instructions.

SPARK loops come three main forms:

1. `for` loops iterate an index variable through a sequence of values; these are equivalent to a conditional loop preceded by an initialisation of the index variable with the first statement of each loop being an index variable.
2. `while` loops have a boolean condition which is checked at the start of each loop iteration, and which if met will cause immediate loop termination.
3. plain loops have no condition and nominally loop forever.

Loops may also have `exit` statements within them, which may or may not be conditional. These exit out of the immediately-enclosing loop.

`LOOP` sets up a loop in the program counter by marking a PC value as the start of the loop. `LPRTN` returns control to the start of the loop, and will therefore be the last statement in the compiled loop block. `LPEXIT` will break out of the loop. Together then these allow emulation of the SPARK looping constructs.

6.5.6 Subprogram calls

Internal subprogram calls are made by writing the parameter data to the mode `in` subprogram parameter addresses in RAM, then calling `SUBJMP` to push the subprogram’s start address on the PC stack. At the end of the subprogram `SUBRTN` will restore the PC, and the new mode `out` parameter values will be read from the subprogram parameter addresses.

External subprogram calls are made by writing parameter values into the appropriate area of (write-only) RAM and then using `SUBEXT` to identify the external package and subprogram to call. The details of I/O to other packages were given in Section 6.3.

6.5.7 Order of execution

A sequence of statements in a SPARK program are executed strictly in order by a conventional Ada compiler. In fact, this need not be the case. The data flow information gathered by the SPARK Examiner allows the compiler to determine that one or more statements may be executed simultaneously.

The instruction dependency features of the interpreter can be used conservatively to make each instruction dependent on its predecessor, and this is the recommended process when testing the newly-compiled software. Out-of-order computations must be carefully calculated.

Two sequential instructions I_1, I_2 cannot be executed in parallel if:

1. I_1 affects the conditional flag and I_2 is conditional;
2. I_1 and I_2 both write to the same package resource;
3. I_1 reads from a resource that I_2 writes to, or vice versa; or
4. I_1 and I_2 are handled by the same core component.

All these conditions can be checked by the compiler, but add complexity and hence increase the chance of a compiler error. The interpreter is deterministic, so at least errors should be repeatable and hence not so hard to track down. Still, it is better to avoid errors in the first place.

6.6 System Interface

As noted in Section 6.2, Ada provides methods for communicating with entities outside the conventional CPU and memory model. It would be reasonable to allow the interpreter to communicate directly with these entities rather than having to let the CPU do the direct communication and pass data between them.

One solution is to use the package input mechanism given in Section 6.3.4, but couple it to custom logic that controls the device's input and output pins directly. This has the benefit of being encapsulated by the normal interpreter package mechanism, but does mean that it cannot interrupt the normal program control flow. Instead there has to be an explicit call to the package for the main program to have access to any data that is gathered. However, the data gathering can run in parallel with the rest of the program.

Another solution could involve an extra expression block in a conventional package, hiding the external interface. This removes the inter-package connection overhead at the cost of potential unconventional CPU core component behaviour.

The exact solution for a particular project is a project design decision.

6.7 Optimisations

The above model is relatively slow and unoptimised. Compared with a conventional compiler/CPU combination its potential advantages are the out-of-order execution and parallel computations. It will suffer from the overhead of being in programmable logic rather than an ASIC, and likely to run at perhaps one twentieth the speed of a conventional CPU for a relatively narrow bus width.

The key to performance gain is to use what the model is good at. Writing data to the PLD, and reading data back from it, can be done at a relatively high burst speed (depending on the system bus). The PLD can process this data while the main CPU

executes the rest of its program, polling the PLD to see when the processed data is ready. This takes load off the main CPU, increasing system performance.

Section 6.6 showed how part of an PLD program could be customised to monitor off-PLD signals. This too can reduce CPU load.

Designing the system architecture is necessary early in the system development process. The designer needs to decide what tasks PLD programs should take from the CPU. Once this is done, the PLD implementation can vary without the main program design needing to change. This is important since it is not yet apparent how one can predict overall system performance with confidence without a mostly-working implementation.

Optimisations of a particular implementation will normally include removal of redundant components and adjusting word and bus widths. The latter appears to be more of an empirical process than an analytic one. At the moment we have no heuristics for identifying implementation bottlenecks. This area is open for further research.

6.8 Conclusions

This chapter has seen a design and outline implementation of a sequential SPARK 95 interpreter running on a generic PLD, intended for running PLD programs of low criticality.

6.8.1 Achievements

We placed no artificial limits on the set of SPARK 95 programs that the interpreter could execute. The limitation of no nested packages was for clarity of exposition, and could be removed by careful management of the name spaces while compiling the top-level package.

We showed that the control and data flow within SPARK could be emulated by a relatively small set of primitive instructions. We produced a detailed mechanism for asynchronous transfer of data between a software SPARK program and the SPARK interpreter. We also produced a mechanism for synchronous transfer of data between package blocks on the PLD.

We made no detailed estimates of the practicality of implementing the design, or of the compilation errors that could plausibly occur.

6.8.2 Evaluation of SPARK

We found the following SPARK features, enforced by the Examiner, key to our design:

1. recursion banned;
2. package ordering in a directed acyclic graph;
3. known data flow of subroutines;
4. exits from loops only possible in immediately-enclosing loop;
5. compile-time knowledge of type sizes; and
6. ability to show freedom of programs from run time exceptions.

6.8.3 Evaluation of SRPT

The SRPT notation proved useful in specifying the input and output events of processes and their relationships. The pre- and post-condition specifications varied in their clarity. There is scope for improved notation and conventions to reduce the size and complexity of the specifications without reducing their precision.

The key test of the notation will be when processes are implemented from their specification. In Section 7.2 we translate the relatively simple stateless processes of the Carry Look-Ahead Adder into gates, but the more complex state-holding processes of the SPARK interpreter are an entirely different problem. This is an area open for further research.

6.8.4 Satisfaction of target aims

Of the targets in Chapter 3 we have addressed or partially addressed:

Target 1: *The process we define must be rigorous.*

This is partly addressed; SPARK programming is a rigorous process, and the interpreter design has been given in an unambiguous notation (SRPT). However, we have not produced any rigorous demonstration that the interpreter correctly executes SPARK, and indeed have stated that the attainable integrity of the interpreter is not sufficient for critical applications.

Target 2: *The process must help the developer to write unambiguous programs.*

This approach allows PLD programming in SPARK, and SPARK programs are unambiguous. The SRPT specification of the interpreter is unambiguous, and so an interpreter implementation which satisfies the specification will likewise run programs deterministically.

Target 3: *The process must allow the programs to have sections written in a low-level language for speed and flexibility, but not allow these sections to compromise overall program reliability.*

The interpreter design allows arbitrary connection to other PLD components as long as they implement the same I/O interface as the interpreter modules.

Target 4: *The process must admit substantial static analysis to discover semantic program errors at or before compile time.*

SPARK programs may be subjected to static analysis via the use of the SPARK Examiner.

Target 10: *The process should provide flexibility so that it may be used in situations not anticipated in its original design.*

The interpreter design provided is parameterised and modular, allowing individual modular designs to be modified as and when necessary and desirable (within limits imposed by communication protocols and PLD architecture.)

6.8.5 Follow-on

This chapter has not gone into great depth for each of the interpreter components for reason of space and chapter focus. The key measure of feasibility is whether such an interpreter can actually be implemented, and whether its performance is comparable to a conventional CPU. This requires further work, and hence this study is listed as a possible future item of research in Section 8.4.3.

Chapter 7

Case Study

This chapter brings together the work of the preceding chapters and shows how it can be used to solve a simple yet realistic problem.

There are two phases to the study. The first phase is a validation of the SRPT specification work. Based on the high-level SRPT specification in Chapter 5 we develop and validate “building-block” processes, then implement the Carry Look-Ahead Adder design. We implement this design in Perl, measure its size and assess its performance.

The task of the second phase is to produce an embedded system to control a ballistic missile interceptor. As far as possible, we use development and analysis techniques described as suitable for SIL-3 systems by MoD Defence Standards 00-55 and 00-54 [MoD97, MoD99].

7.1 Target Aims

We address the following targets from Chapter 3:

Target 5 The program produced must be easy to test.

Target 6 It must be able to be compiled onto a range of existing and anticipated logic devices.

Target 7 It must reuse existing proven tools where feasible.

In addition we consider the question of the practicality of development of significantly-sized systems.

7.2 Carry Look-Ahead Adder

In this section we present the construction of a general-purpose simulator for a generic single-clock synchronous PLD. We then show how it was used to implement the design of a carry look-ahead adder (CLAA) from Section 5.3.

We do not aim to make the simulator implementation conform to SIL-3 or SIL-4 software standards. Instead, we treat it as a testing tool; we require confidence that it fulfils its requirements accurately, and that it has been constructed in such a way as to highlight errors in its design and implementation. It should provide an overall increase in confidence in the circuits it simulates, but will not provide the sole evidence of correctness of the circuit.

7.2.1 Simulation environment

The environment used was the Perl programming language. This was chosen for its ease of use, cross-platform compatibility and support of OO inheritance. Java was an alternative choice but the author had more experience of Perl.

Structure

The base Perl modules used by the simulator were:

Blocks.pm Generic blocks

Gates.pm Logic gates (single-cycle, stateless)

Utils.pm General utilities for conversion between data types

Functions.pm Logical functions for logic gates

The first two modules in the list implement object instantiation in the standard Perl way. The other two export functions for use by other modules, and are stateless. All of them raise no errors under the Perl `-w` and `use strict` syntax checks.

Programming interface

The Blocks module implements the following methods:

`new(specs)` Initialise a new block with the given `specs`

`gate_count()` Count the number of gates in the block

`copy(old)` Copy an existing block `old`

`set_in_map(map)` Set the mapping of input pins to inputs inside the block

`set_out_map(map)` Set the mapping of outputs inside the block to output pins

`set_route(route)` Set the routing between internal blocks

`map_in()` Map input pin values onto the relevant inputs

`map_out()` Map output values onto the relevant output pins
`set_input(idx,bit)` Set input pin `idx` to `bit`
`get_output(idx)` Get the value of output pin `idx`
`get_delay()` Get a block's computational delay
`eval()` Evaluate internal blocks and gates
`route()` Route data between internal blocks and gates
`add_object(obj)` Add a block or gate `obj` to the block contents
`cycle()` Cycle a block (`map_in();eval();map_out();route()`)

Testing

The modules' code has been tested by comparing the results of computations with results worked out by hand in a range of blocks. This would clearly be inadequate for a tool intended to support SIL-3 software development, where such techniques such as white-box testing, regression testing and independent code review might be used. General testing and validation strategy is discussed further in Section 7.2.5.

7.2.2 Building blocks

The first step towards allowing module building was to define a range of logic gates in the `Gates` and `Functions` modules. Each of these was taken to complete its calculations in 1 cycle. Most gates had 1, 2 or 3 inputs. The exceptions were n -ary **and**, **or**, **pass** and **xor** gates in the `Functions` module.

The `Gates` module allows the user to define the maximum number of inputs permissible on these gates, causing a runtime error at the instantiation of any gate with more than the permissible number of inputs. This corresponds to specifying the maximum inputs and outputs on each cell in a particular PLD.

The user can now build their modules out of these gates. These modules subclass module `FPGA::Blocks`. A module's block is formed by Perl functions which instantiate a container block and then instantiate and connect a series of other blocks and gates in the container. All such modules are placed in the `Useful` include directory as standard.

7.2.3 Adder block

The CLAA was implemented in the module `FPGA::Useful::Adder` in the manner described above.

This class's `new()` method requires parameters `NAME` for the name of the adder, `WIDTH` for the bit-width of each adder input and `ADDON` for the number to add on to the basic sum; this would typically be 0 or 1.

The main task of the module was to declare a block containing the three sub-adders and other gates according to the design developed in Section 5.3. These were given widths according to the user-input width for the containing adder.

If the `WIDTH` parameter was 1, the module built a simple half-adder block from an **XOR** and **AND** gate.

Bit width w	Gates g	Delay d	$\lfloor \log_2(w - 1) \rfloor$
1	2	1	–
2	9	2	0
3	25	3	1
4	31	3	1
5	64	4	2
6	80	4	2
7	93	4	2
8	100	4	2
9	171	5	3

Table 7.1: Adder size and delay properties

7.2.4 Testing

A `class_test` Perl script and Makefile system provided a generic facility to test a given `FPGA::Blocks` subclass. The script was a wrapper around use of the subclass `self_test()` method. Typically, this method iterated through a range of block size parameters (`WIDTH` in the case of the `CLAA`). In each case it created a suitable instance of the block, extracted test data from the `test_cases()` method and tested the block against the expected output.

The `test_cases()` method implemented random checking of a `CLAA`, generating random input data and checking that the sum of the random data emerged in a pipelined fashion after the block's declared output delay.

Testing revealed no functional errors, and produced concrete statistics about the implementation. Table 7.1 shows the properties of the Adder for a range of bit widths. The delay is measured in PLD clock cycles.

The delay was indeed logarithmic in bit width, matching our performance specification. For width $w > 1$, delay $d = 2 + \lfloor \log_2(w - 1) \rfloor$ as the table shows. Gate size leapt, and delay increased by one, at each $1 + 2^k$ for integer k as we would expect.

The testing of larger devices revealed an omission in the simulation environment. Since basic gates were limited to 3 inputs, when a 4-gate **PASS** gate was requested the `Gates` module raised an error. This was overcome by extending the module's `new()` method to instantiate a block with as many smaller pass gates as needed. Re-running the fixed simulation produced the correct results.

7.2.5 Simulation environment reliability

The simulation environment is a key component in the argument for correctness of our programmable logic system, since inadequate or incorrect simulation may lead to construction of a system which simulates correctly but behaves incorrectly in real life. This creates a debate about the required level integrity of simulation and analysis tools in a safety-critical system.

Perl is clearly an unsuitable language in which to implement a safety-critical system. The key reasons are:

1. it is an interpreted language, causing a significant performance penalty compared

to compiled languages;

2. the Perl interpreter is large and thus difficult to verify in any meaningful way;
3. the interpreter does periodic garbage collection, making program execution effectively non-deterministic;
4. the language is purposefully very weakly typed;
5. Perl is not a “static” language; rather, it develops steadily over time and language constructs may change their meaning; and
6. Perl itself has a number of language concepts (such as default variables) which obscure the meaning of program statements and are prone to cause error.

Comparing these properties with the requirements for selection of programming language in Section 28 of DefStan 00-55 [MoD97], we see that the weak typing, lack of formal syntax and lack of predictable program execution clearly make it an unsuitable programming language.

However, the Praxis Critical Systems Perl Coding Standard [Lee00] provides guidance on developing Perl programs designed for reliability. Using this, with independent verification of programs against this standard, Praxis have justified the use of Perl tools in *support* of a safety-critical system development. Performance issues are negated since programs are not required to run in real-time, the standard requires the use of a “well-trodden” subset of the main language, and coding rules combined with manual inspection of the code reduce the risks posed by weak typing.

7.2.6 Conclusion

Although not conclusive proof that the CLAA refined design was perfect, or indeed that the simulation was error-free, this simulation and testing leads to increased confidence in the design. This fulfils the main requirement for the simulator which we expressed at the start of this section. The simulation also demonstrates that development of the CLAA is easy to test (target 5) according to its criteria from Section 3.7.2:

- 5.1 amenable to production of a test plan from the specification;
- 5.2 amenable to instrumentation of the compiled program so that relevant data flow can be observed;
- 5.3 with a working, verified simulator; and
- 5.4 test vectors for the simulator can easily be produced from the test plan.

Target 6 is met according to its criteria from Section 3.7.2:

- 6.1 a non-trivial program being developed into a form for compilation and running using an existing PLD and toolset

since a simple gate-level description is trivially mapped into VHDL, Verilog or netlist format.

In the next section we will construct a program and test harness for a real-time safety-critical system, using the techniques described in the preceding chapters.

7.3 Missile Guidance System – Overview

This case study addresses the problem of incorporating a programmable logic component into an existing safety-critical system which was not originally designed for it. This is not the ideal way to construct a safety-critical system, but it is a reflection of current practice and has the bonus of providing a stringent test of our techniques.

We first implement the system in conventional software, written in the high-integrity SPARK subset of Ada 95, using state-of-the art analysis tools to prove safety-related properties of our software. We then select a subsection of the code to be implemented in programmable hardware, and transform the code to a form suitable for communicating with a PLD.

We aim to identify the main difficulties in this re-engineering process, and (where possible) propose and demonstrate solutions. A secondary aim is to identify which features of the SPARK Ada subset are less amenable to transformation into PLD form and propose transformation strategies for them.

7.3.1 Related work

Demonstration software systems have been used for studies before. Napier *et al* [NMH99] described the implementation of on-line diagnostics for safety-critical systems, using a boiler water control system implemented in Ada as part of an earlier study by the UK Health and Safety Executive. This system had 70 Ada packages, and communicated with a GUI over a serial link.

The advantage of a publicly-available software system (and associated test harness) is that it provides common ground for future studies. It also permits some degree of direct comparison between studies. For this reason, the software and test harness for this system will be made publicly available.

7.3.2 System requirements

The system is the main control unit (MCU) for an endo-atmospheric interceptor missile, armed with a low-yield fission warhead. This system is clearly safety-critical; a detonation of the warhead at the launch site is a definite hazard to life. Of course, there are mission-critical requirements as well; if the warhead were never to go off, the missile targeted for interception would probably get through to its destination and be likewise a hazard.

We assume that the live warhead is only connected on a production missile in the operational environment, and hence there are no special safety considerations during development and testing of the system.

7.3.3 Safety

The main hazard of the system will be detonation of the on-board warhead at an unsafe location (i.e., close to the launch point, or below a certain altitude). This dictates safety considerations such as having confidence in the estimated distance from launch point.

We assume that the overall system has been assessed as SIL-4, but the programmable part has been assessed as SIL-3. In a real project this reduction would be

justified by non-programmable measures taken to mitigate the main system hazard, e.g. an analogue timer and accelerometer in series with the software warhead detonator wire, designed to only enable transmission on the wire after a certain time and after the missile has maintained a certain acceleration for a certain amount of time.

According to Defence Standards 00-54 and 00-55, SIL-3 indicates the use of some formal notations (e.g. for specification) and semi-formal analysis techniques. It does not require proof of object code.

7.3.4 Implementation limits

Since we lack appropriate hardware, we can only implement this system in pure software. We must therefore produce appropriate simulation and test software in order to have any justifiable confidence that the system does what is required. We have already discussed (in Section 7.2) the reliability requirements for such software.

In this particular system simulator we will apply some SIL-3 development techniques to the test harness software, implementing it in SPARK Ada where possible and plain Ada where required. We will not measure the system's real-time performance.

The time taken to implement and test the full system will likely be in the order of the square of the number of interacting components. For this reason, we will test only a subset of the total system's functionality, ensuring only that the existing components work well enough to support the functioning of the missile's safety-critical functionality.

7.3.5 Implementation technologies

The system (and most of the accompanying simulation and test code) was written in the SPARK subset of Ada 95. The standard switches used for analysis are shown below:

```
-i=missile -exp -listing_extension=ls_ -config=gnat -st
```

They indicate, respectively:

- use of the SPARK index file `missile.idx`;
- generation of full exception checks including arithmetic overflow;
- listing output to `.lss` and `.lsb` for Ada specifications and bodies respectively;
- use of file `gnat.cfg` to specify the target-specific ranges of the base Ada types; and
- generation of statistics on Examiner table usage.

The compiler used was GNAT 3.2 on i686 Linux, although the system was also compiled and checked on GNAT for Windows 2000 and Solaris. Note that if a different compiler was used e.g. to cross-compile to a PowerPC target, then the compiler configuration file given to SPARK must represent the *target* compiler.

The overflow checks generate verification condition (VC) files for each package body analysed. These conditions must be shown to be true in order for the developer to be confident that the system is free of all run-time exceptions. The strategy used was

to use the Simplifier tool to discharge the maximum number of VCs automatically, and then justify key remaining VCs using manual inspection, recording results in proof review (.prv) files. This strategy avoids the extra effort needed for semi-automatic proof of the VCs with the Proof Checker tool, at the risk of manual justification of VCs being incorrect.

7.4 System Components

The system has the following components. For each requirement we list the operational (functional) and safety (non-functional) requirements. The safety requirements are those that would be produced as a result of the system hazard analysis.

7.4.1 System clock

1. Measures time since system power-on.
2. Time measured in milliseconds with an accuracy of 0.002% (under 2 seconds in 24 hours).

The clock was constructed with package `clock`. The simulation body was implemented with a clock that incremented by 1 millisecond after every read, and had an external interface to allow simulator adjustment of the clock value.

Additional functions to operate on clock times were supplied in package `clock_utils`.

7.4.2 1553 bus

1. Allows communication between the MCU and the other LRUs.
2. The MCU is the bus controller.
3. The “bus catalogue” is a list of pages; each page relates to the interaction between the MCU and an LRU.
4. A bus catalogue page has a list of Rx (MCU to LRU) and Tx (LRU to MCU) 16-bit words used to communicate, assigning meaning to the bits in each word.
5. Each word is marked with a “fresh” bit by the sender when it is to be sent. The receiver can inspect any of the sent words at any time, and can see the “fresh” bit along with a “valid” bit controlled by the bus.
6. Maximum time lag between the sender marking a word as fresh and the receiver seeing the fresh data is 15ms + 1 system cycle.

Safety requirements:

1. A bus failure indication (no-data condition) for any Tx word for more than 2 seconds is taken to indicate total failure of the sending LRU.
2. The bus must report a valid self-test on start-up.

The bus was constructed with packages `bus`, `bc1553` and `rt1553`. These provide a simulation of a standard 1553 bus, and interfaces to the simulation for a Bus Controller and Remote Terminal. The bus simulator was tested with program `test_bus` to demonstrate basic functionality.

Copies of the test program and interface package specifications are given in Appendix B. The test program in particular shows how the `Test` package is used during testing.

7.4.3 Watchdog timer

This is a standard component for safety-critical systems, used to detect system failures such as program run-away. When such failures occur it will either reset the system (if it can be safely reset), or take more drastic action such as self-destruction.

Properties:

1. Provides a reset interface to the MCU.
2. After a reset, the timer will count out 750ms. If not reset within this time, the timer will go off.
3. If the timer goes off, the watchdog will immediately command a missile self-destruct.

Operational requirements:

1. The timer must be reset within 600ms of a previous reset during normal system operation.

Safety requirements:

1. The watchdog timer may only be reset at one point within the program.
2. The watchdog timer reset command must be so placed in the program as to detect as many kinds of system failure as possible.
3. The program must not send a timer reset if any system failure is detected.

The watchdog timer was constructed with package `watchdog`. The simulation body interfaced to the clock to check for timeout when commanded.

7.4.4 Barometric sensor

This sensor detects altitude above mean sea level using barometric pressure.

Properties:

1. Measures current altitude above sea level.
2. Accuracy is $\pm 5\%$ at sea level, up to $\pm 10\%$ at 20 000m.
3. Above 20 000m barometric readings will read as if at 20 000m.

4. Must be calibrated with current altitude at system start.
5. Polling frequency is 200ms.

The barometer emulator was constructed with package `barometer`. The MCU interface to the barometer bus messages was constructed with package `if_barometer`. These packages were tested using the main test harness with test script `barometer.in` which is listed in Appendix C along with its output.

7.4.5 Airspeed indicator

This sensor measures the speed of the missile relative to the air it passes through. As the altitude increases, the sensor's accuracy will decrease.

Properties:

1. Measures current speed relative to still air, in meters per second.
2. Accuracy is $\pm 1\%$ at sea level, up to $\pm 10\%$ at 20 000m.
3. Above 20 000m, airspeed readings should be regarded as random.
4. Maximum airspeed measured is 2550 m/sec

The airspeed emulator was constructed with package `airspeed`. The MCU interface to the airspeed bus messages was constructed with package `if_airspeed`.

7.4.6 Inertial navigation system

This sensor measures the missile's displacement from its starting point using a ring laser gyro set to measure relative movement.

Properties:

1. Accuracy is $\pm 0.3\%$ at speeds above 100m/s, $\pm 1\%$ below that.
2. The co-ordinate system places the origin at the initial centre-of-gravity of the missile.
3. A left-handed axis set is used.
4. The Y axis runs along the long axis of the missile's initial attitude.
5. The X axis runs in the direction of the number 0 steering fin.
6. The Z axis runs in the direction of the number 1 steering fin.
7. Polling rate is 50ms

The INS emulator was constructed with package `ins`. The MCU interface to the INS bus messages was constructed with package `if_ins`.

7.4.7 Solid state compass

This sensor is a solid-state compass which detects missile attitude relative to the Earth's magnetic field.

Properties:

1. Must be initialised at start-up with the local normal vector to Earth's surface.
2. Accuracy is within a 0.04 radian cone at sea level, decreasing linearly to a 0.2 radian cone at 40,000m.

The compass emulator was constructed with package `compass`. The MCU interface to the compass bus messages was constructed with package `if_compass`.

7.4.8 Fuel tank sensor

This sensor reads the amount of fuel in the missile motor's tank.

Properties:

1. Measures fuel remaining in kilos, from a maximum 100kg fuel load.
2. Accuracy is ± 1 kg down to the measurement of a 5kg fuel load, below which the reading must be assumed to be a random value between 0 and 6kg.

The fuel tank emulator was constructed with package `fuel`. The MCU interface to the fuel tank bus messages was constructed with package `if_fuel`.

7.4.9 Proximity fuse

This sensor is a rapidly nutating (rotating) UV laser proximity fuse.

Properties:

1. Will only detect a reflection off a valid target within 1000m of the target.
2. Reflection is not guaranteed as the detection area is limited by an angle of ± 1 radian from the normal to the sensor window.

The fuse emulator was constructed with package `fuze`. The MCU interface to the fuse bus messages was constructed with package `if_fuze`.

7.4.10 Millimetre radar sensor

This sensor is a phased-array millimetre-wave radar in the nose of the missile.

Properties:

1. Will detect a valid target within a cone of 0.8 radian width off the missile's long axis towards the nose.
2. Maximum guaranteed detection distance of a valid target is 10 000m.
3. Accuracy of location is ± 0.02 radians and $\pm 10m$ range.

4. Doppler processing will read the speed of the target relative to the missile in the direction of detection with accuracy of $\pm 3\%$.

The radar emulator was constructed with package `radar`. The MCU interface to the radar bus messages was constructed with package `if_radar`.

7.4.11 Staring infra-red sensor

This sensor is an array of infra-red sensing cells in the nose of the missile.

Properties:

1. Will detect a valid target within a cone of 1.2 radian width off the missile's long axis towards the nose.
2. Maximum guaranteed detection distance of a valid target is 30 000m.
3. Accuracy of location is ± 0.1 radians.
4. Approximate range information is given by expected target temperature, and will be accurate to $\pm 30\%$ for a valid target.

The IR sensor emulator was constructed with package `ir`. The MCU interface to the IR sensor bus messages was constructed with package `if_ir`.

7.4.12 Fins

The missile has four independent steering fins, spaced equally around the missile body.

Properties:

1. Each fin has a possible deflection (position) of between -1 and +1 radians from neutral.
2. Extreme rotation of a fin in a high atmospheric drag environment (high speed / low altitude) can lead to fin mechanical failure.
3. Maximum response time from position command to position achieved is 800ms.

Operational requirements:

1. Avoid "chatter" (a rapid sequence of positive and negative values) in fin steering commands in order to reduce the probability of fin mechanical failure.
2. Lock the fins in neutral position at system start.

Safety requirements:

1. Fins must report valid self-test at startup.

The fins emulator was constructed with package `steer`. The MCU interface to the fins bus messages was constructed with package `if_steer`.

7.4.13 Motor

The missile has a liquid-fuel rocket motor to provide thrust along its long axis.

Properties:

1. Variable thrust between 5 and 35 kN.
2. Optimal fuel consumption is at 21.5 kN thrust.
3. Thrust efficiency decreases by a small amount with increased altitude due to reduced atmospheric oxygen partial pressure and hence a leaner fuel mix.
4. Maximum thrust at zero atmospheric oxygen is 29kN.

Operational requirements:

1. Avoid chatter in thrust level commands in order to reduce the probability of thrust chamber mechanical failure.
2. Avoid repeated thrust ramp-up and ramp-down in order to conserve fuel and reduce the probability of thermal cracking in the thrust chamber.
3. Ramp-up to 30% of maximum thrust at system ignition.
4. Do not change this thrust level until missile has travelled over 100m vertically.

Safety requirements:

1. Motor must report valid self-test at system start-up.

The motor emulator was constructed with package `motor`. The MCU interface to the motor bus messages was constructed with package `if_motor`.

7.4.14 Self-destruct

For safety, the missile must be able to destroy itself safely. There are four separate charges placed throughout the missile body.

Properties:

1. Detonation will fragment the missile body and destroy the warhead without causing warhead detonation.
2. Self-destruct with a full fuel load at low altitude will cause an explosion with blast effects approximately equivalent to a conventional blast-effect 250kg bomb.
3. Self-destruct requires a timed sequence of keywords to be sent to the self-destruct bus unit. There is no acknowledgement back from the unit.

Operational requirements:

1. Self-destruct must not be initiated if the missile is still capable of flying its designated mission safely.

Safety requirements:

1. Self-destruct must be initiated before the missile impacts the ground.
2. Self-destruct must be initiated whenever any sensor or actuator failure occurs that significantly increases the probability of warhead detonation outside detonation parameters.
3. Self-destruct must not be initiated within 1000m of the launch point.
4. Self-destruct must report valid self-test at system start-up.

The self-destruct emulator was constructed with package `destruct`. The MCU interface to the self-destruct bus messages was constructed with package `if_destruct`.

7.4.15 Warhead

The missile has a 12kT fission warhead as payload.

Properties:

1. Detonation requires a timed sequence of keywords and a challenge-response authentication between the command unit and the warhead unit.

Operational requirements:

1. A valid target for the warhead is an object travelling at over 400m/s whose 10-second historic track places or will place it within a 10 000m sphere with origin equal to the launch point.
2. Optimal detonation distance is 800m from a valid target.

Safety requirements:

1. The warhead must not be detonated within 10 000m of the ground.
2. The warhead must not be detonated within 20 000m of the launch point.
3. The warhead may only be detonated within 2 000m of a valid target.
4. The warhead must report valid self-test at system start-up.

The warhead emulator was constructed with package `warhead`. The MCU interface to the warhead bus messages was constructed with package `if_warhead`.

7.5 Design

The software system was designed using the INFORMED[Ame00] design method. The package hierarchy was extracted from the above system components, augmented with basic types packages and interface packages.

7.5.1 Design decisions

Significant design decisions included:

- two SPARK boundaries, one for the main missile controller and one for the emulator code, overlapping lower in the inheritance hierarchy;
- top-down design as per INFORMED, ensuring each specification was written and valid SPARK before implementing the corresponding body;
- supplementing each state package with a non-SPARK test procedure;
- implementing as much of the simulator as possible in valid SPARK; and
- using a script-based test harness above the main program.

7.5.2 Package structure

Figure 7.1 shows the design of the system with the SPARK and simulation boundaries. Each significant package is shown; a red circle indicates the presence of state within a package. The arrows show the direct **with** (package hierarchy) relations. A representative subset of the component packages are shown, for reasons of diagram space and clarity.

7.5.3 Code structure

The top-level system program in design unit **Main** is a simple polling loop, calling a sequence of embedded subprograms to deal with each system component.

7.5.4 Design limitations

The current lack of tasking (coarse-grain parallel processing) in SPARK Ada was keenly felt. The main program broke down naturally into a small set of loosely-coupled tasks managing functions such as location tracking, target tracking and self-test. In sequential Ada these had to happen in an artificial order in a polling loop, introducing artificial dependency relations between their states.

The addition of the Ravenscar tasking profile[BDR98] to SPARK 95 should make such programs easier to express.

7.6 Implementation

The implementation was done in the following main phases:

1. construction of basic types packages;
2. design, build and test of the bus interfaces and emulator;
3. construction of the simulator and interface for the **Barometer** package;
4. construction of the basic test harness;

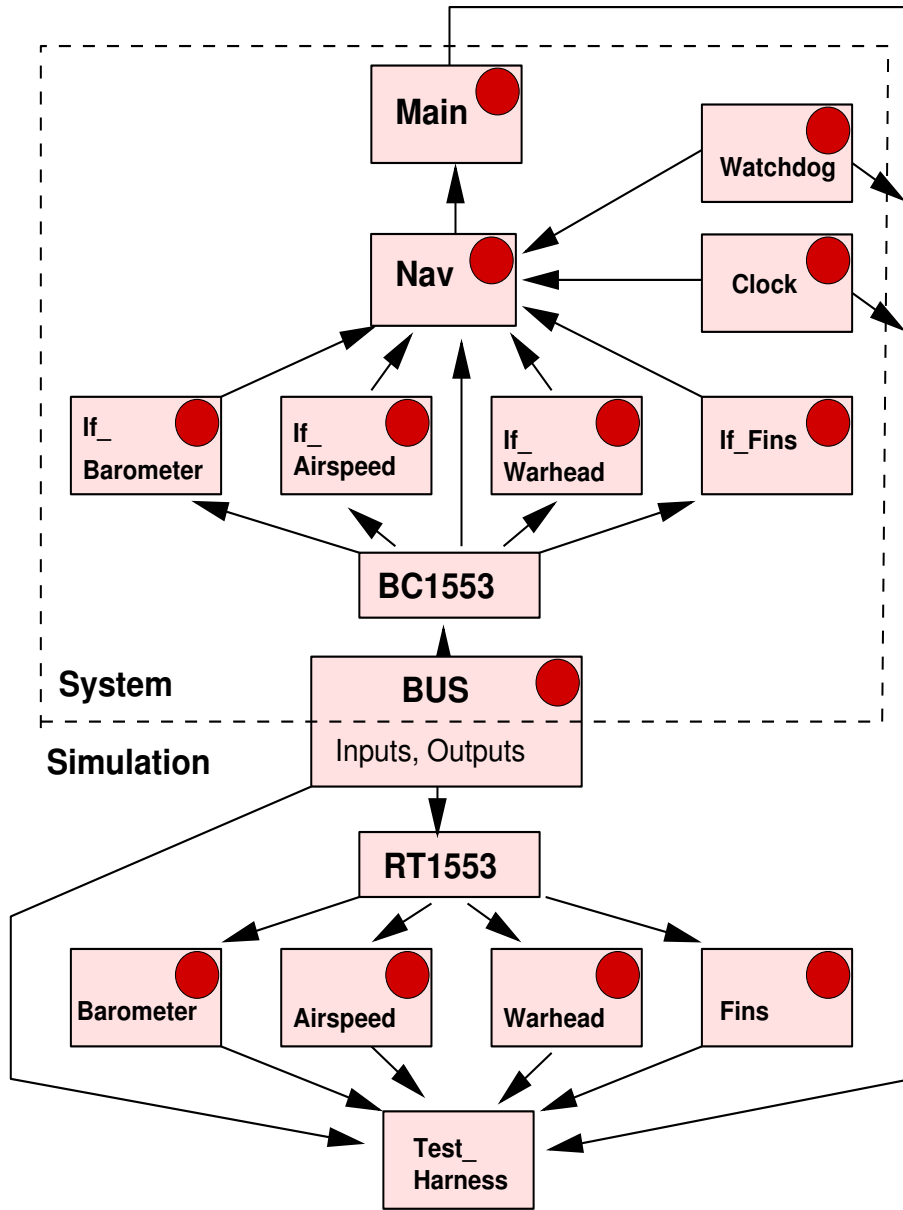


Figure 7.1: Missile system design

5. testing of the `Barometer` code and subsequent fixes to the code and test harness;
6. addition of one sensor at a time, extending harness code and adding types packages where required;
7. construction of the `Nav` package for position estimation; and then
8. construction of the main `Missile` package.

7.6.1 Development

The development methodology for each package was:

1. writing of the specification;
2. SPARK of the specification, fixing identified errors;
3. writing of the body;
4. SPARK of the body, fixing identified errors and updating the specification annotations where needed;
5. Simplifying of VCs for the package and fixing code flagged by any obviously false VCs;
6. writing of the `Command` testing subprogram for the package;
7. compilation of the package;
8. creation of the test script for the package; and then
9. testing of the package, fixing code and amending the test script where required.

This late use of the compiler was effective in that very few compilation errors were reported on the first compile; those that occurred were normally in the `Command` non-SPARK routine. The VC inspection was a relatively effective method of locating potential program errors for a small investment of manual inspection, especially for numeric overflow errors for the results of calculations.

The SPARK report for the analysis of the `Nav` package body is given in Appendix D. It shows the SPARK Examiner options used, the packages that needed to be analysed due to dependency by `Nav` on them, and the fraction of the Examiner tables used.

The final code count was 504K of Ada files, with 16 800 lines. Of these, 2 500 lines were annotations, 2 900 were comments, 1 300 were blank, and the remaining 10 000 (forming 330K) were Ada code. 20K of this Ada was test-related code. There were 75 packages and public child packages, with 9 of those packages related to testing. This verifies that the system is not trivial in size.

7.6.2 Testing

The testing was done with a script-driven test harness, written in Ada but not using the SPARK subset. Each significant module has a `Command` subroutine which reads data from standard input and acts upon test script commands relevant to that routine. The subroutine is made separate from the package body and marked as `--# derives null` so that the Examiner will not examine it and will assume that it has no effect on the “interesting” (annotated) part of the system.

The test routines call package `Test` whenever they perform a check; successful checks increment the `Pass` count, and unsuccessful checks increment the `Fail` count. The results of each test are shown on standard output. Test scripts can change aspects of the simulated packages (e.g. the current time or the current estimated height) and display comments about what is being tested. At the end of a test run, the harness shows the total number of pass and fails.

An example test script for testing the basic functionality of package `Barometer` is shown in Appendix C.

7.6.3 Conclusions

The development produced the following lessons and statistics related to SPARK and Ada development:

- A developer will have to do things properly eventually, such as provide I/O and check functions for all major types, and no time will be gained by trying to short cut this.
- The `-exp` switch is effective at locating overflow errors when combined with Simplifier usage.
- The public child packages provided in Ada 95 and SPARK 95 are a great aid to testing since they can easily be excluded from a SPARK analysis yet provide direct visibility to their parent package for I/O and check functions.
- The use of a Makefile makes project management much easier, especially with regard to keeping testing up to date.
- The Examiner processed a large system in acceptable time (8.163 seconds of real time to SPARK everything SPARK-able with the standard switches, on a 1.35GHz Athlon XP processor).
- Of the 3085 VCs produced for this project, 32% were discharged by the Examiner (version 6.2) and 58% by the Simplifier (demonstration version) leaving 8% to prove manually. Simplification of the entire system took 364 seconds on the aforementioned PC. Trial use of version 7.0 of the Examiner discharged 35% of the VCs directly.

7.7 Introduction of A PLD

With the system passing SPARK analysis, compilation and testing, it was then necessary to choose some system functionality to incorporate into a PLD. We aimed to produce a new program, with minimal changes to the original program annotations.

The three phases of this work were:

1. identify a suitable subsection for transformation;
2. replacing the existing code with calls to a PLD interface; and
3. transform the replaced code into a VHDL implementation;

7.7.1 Subsection identification

The code chosen for transformation was the `Nav` package, which tracks estimated missile position. It is suitable for transformation because it requires relatively infrequent updates from the main software (periodic updates on time and estimated missile speed and attitude) and produces on-demand estimation of the current delta position from launch. These properties match up well with those we described in Section 4.3.8.

7.7.2 PLD interfacing

The original implementation of `Nav` is given in Appendix E. It provides public functions for accessing its internal tracking of recent sensor measurements (abstract variable `Location_State`) and sensor states (abstract variable `Sensor_State`). The `Maintain` polling routine calls the `Handle_XX` routines for Airspeed, Barometer, Compass and INS sensors; these routines check the named sensor's current readings, and if the sensor has failed will attempt to use other sensor readings to estimate appropriate values. The `Estimate_Height` and `Handle_Airspeed` subprograms are shown in full form in the appendix; the others have been made separate for brevity.

The design decision was made to transform the package to have no intrinsic state, but instead use memory-mapped state variables to communicate with the PLD. The two existing abstract state variables were retained and their refinement components mapped onto PLD output pins, and a new abstract state variable `FPGA_Inputs` was added which was mapped onto PLD input pins. No synchronisation code was necessary since the PLD implementation is stateless and hence may be pipelined.

The existing public functions were left essentially the same, with only minor changes to memory-mapped variable accesses made due to SPARK rules. The `Handle_XX` subprograms were unnecessary due to being moved onto the PLD and were removed. The `Maintain` routine was changed to read each sensor's state and write them directly out to the PLD input pins.

The resulting package body `Nav_FPGA` is listed in Appendix F. The correspondence with the original is quite clear. The most significant change is the addition of declarations for calculating type bit widths and mapping variables into memory. In fact, some of these bit width calculations will not actually compile under GNAT since they are not properly static; in practice, they would have to be replaced by actual numbers. They have been left in the code in order to show the derivation. Again, `Estimate_Height` is given in full form and the other `Estimate` routines are made separate.

7.7.3 Transformation

The high-level structural steps of transformation of the selected `Handle_XX` subprograms of package `Nav` into VHDL were:

1. replace global variables in the subprogram declaration and body with the appropriate PLD input and output vector names;
2. identify each subprogram's in and out argument and global data and create a VHDL architecture declaration for it;
3. add appropriate `Clock` and `Reset` inputs to the declaration;
4. connect the appropriate PLD input and output pins to the subprogram's inputs and outputs;
5. create the VHDL implementation for the subprogram by declaring architectures for the major Ada control flow elements;
6. add declarations for appropriate vectors to connect these architectures; and then
7. add the required connections between blocks and architecture inputs and outputs.

At the level of translating subprogram body code from SPARK Ada to VHDL, no initial effort was made to enable fine-grain parallelism. Instead, SPARK Ada program constructs (principally alternation and assignment) were mapped into the most directly corresponding VHDL representation (respectively, multiplexing from expression evaluation and data routing).

No compilation or simulation of the VHDL was done since it was a capability demonstration. A process for producing timing-robust VHDL from a SPARK design is clearly required for this transformation process to be practically useful.

7.7.4 Results

The transformation process produced the following discoveries:

Software implementation to PLD interface

- Relatively little of the package specification changed. The abstract state variables gained SPARK modes, and one extra output abstract variable was required, but the global and derives annotations did not change greatly.
- Most of the work in the package body involved mapping concrete state variables onto the correct area of memory. External global data (from the sensors) was passed directly onto the PLD inputs.
- The transformation was not quite automatic, but was effected quickly and was amenable to manual inspection for correctness.

Software implementation to PLD implementation

- The SPARK annotations were very helpful in characterising the inputs and outputs quickly, making VHDL architecture declarations simple to write.
- Bit widths could be easily calculated manually, and minimised by use of `pragma Pack()` and Ada representation clauses. There seems no reason why these widths could not be estimated by a relatively simple tool, given a SPARK syntax tree.
- The guarantee of no expression overflow given by the Examiner `-exp` flag (and subsequent proof) would greatly simplify the process of writing VHDL to compute arithmetic expressions.

7.8 Conclusion

In this chapter we demonstrated that our Chapter 5 work on refining a carry look-ahead adder specification into an SRPT form could be mapped into a gate-level simulation of a generic PLD. We constructed a suitable simulator and used it to verify that the implementation met its specification.

We then wrote a controller program for a high-integrity embedded system, using existing state-of-the-art software development tools and techniques, and simulated mapping a section of the program into a programmable logic device.

The main conclusions of this work are as follows:

7.8.1 Refined program simulation

1. The refined program worked as expected in a gate-level simulated implementation.
2. A gate-level simulator with a single clock is not hard to produce, and provides increased confidence in such programs.
3. The creation of large, parametrised designs by instantiating and composing smaller blocks can be easily expressed in an imperative language supporting inheritance.

7.8.2 SPARK program development

1. Writing a SPARK 95 program with information-flow analysis can be done at a similar speed to writing conventional full Ada programs.
2. Maintenance of SPARK annotations during development does not take significant time.
3. Top-down program development with late compilation is quite feasible, with a properly-formed design.
4. The time taken to run the SPARK Examiner and SPADE Simplifier on a substantial program is not noticeable on a conventional 1.5GHz i686 PC.
5. The recent addition of tasking to the current SPARK model is likely to be valuable in designing embedded controllers.

7.8.3 Targets

Of the targets in Chapter 3 we have addressed or partially addressed:

Target 1: *The process we define must be rigorous.*

We have based PLD program design in the rigorous and formally-specified SPARK Ada 95 language. We have shown how key program properties such as freedom from arithmetic overflow can be demonstrated. The transformation process from SPARK Ada to VHDL is currently manual and not rigorous, but we have demonstrated that the new interfacing code can be valid and meaningful SPARK.

Target 2: *The process must help the developer to write unambiguous programs.*

SPARK Ada is unambiguous by definition, removing all Ada language features that may introduce compiler-dependence.

Target 3: *The process must allow the programs to have sections written in a low-level language for speed and flexibility, but not allow these sections to compromise overall program reliability.*

The use of VHDL enables the VHDL implementation of arbitrary blocks in the original SPARK program to be replaced with custom VHDL code while leaving their architecture (interface) unchanged. Verilog could be used similarly.

Target 4: *The process must admit substantial static analysis to discover semantic program errors at or before compile time.*

SPARK Ada can be analysed by the Examiner for a range of statically-verified properties, and verification conditions generated to admit proof of run-time properties.

Target 5: *The program produced must be easy to test.*

We have addressed SPARK program testing, but the testing of the VHDL component was not addressed.

The Perl PLD simulator has demonstrated that refined PLD programs are amenable to automatic test.

Target 6: *The program must be able to be compiled onto a range of existing and anticipated PLDs.*

We have used VHDL as a target language, compilers for which exist for most substantial PLDs.

Target 7: *The process must reuse existing proven tools where feasible.*

We have employed existing tools (the Examiner and Simplifier) without modification, but suggested areas such as bit width calculation where extra tools may be useful.

Target 9: *The process should indicate what kinds of error may arise at each stage.*

The static analysis results limited the errors that may be present in the SPARK. Errors in proven and tested SPARK programs are likely to be requirements-related rather than “accidental”.

Target 11: *The process must admit justification to the project safety authority that the programs output by the process are of an adequate integrity level.*

The use of SPARK as a design tool for and interface to the PLD program provides traceability for the PLD program design and implementation. The PLD program may be manually inspected and reviewed against the original SPARK implementation to demonstrate coverage of requirements. SPARK has been used and accepted at SIL-4, although if only used as a design tool it is unlikely that a SIL-4 argument can be made for the resulting PLD program without substantial extra evidence.

Target 12: [00-54 8.5.2] *The analytical arguments provided shall include:*

- (i) any formal arguments used in validation to show that the formal specification complies with the safety requirements;*
- (ii) any formal arguments that the functional design satisfies the formal specification;*
- (iii) for non-functional properties with specified safety requirements, analysis of the achieved behaviour, e.g.: performance, timing etc.;*
- (iv) analysis of the effectiveness of fault mitigation, for example use of such techniques as diverse implementations.*

(i) is addressed because the safety requirements may be expressed as post-conditions in the SPARK program and the code proven against them. (ii) is addressed because the SPARK analysis justifies the information flow annotations in the SPARK program, showing consistency and the level of coherency of the design; (iii) is not addressed since SPARK does not yet have any timing-related analysis; (iv) is addressed because the SPARK implementation could be used in parallel with the VHDL implementation, with a checking routine flagging deviations in the computed results. Only a limited amount of diversity is present, however.

Target 13: [00-54 12.1.2] *The Design Plan shall define the life cycle that is to be followed in the development of the custom circuit, including a specification process, a development process and a verification process.*

The SPARK development process is well-established in safety-critical projects. Individual projects place different emphasis on its components, but the core of the process (design - analyse - implement - analyse - test - fix - re-analyse) is common. The PLD program development is then headed by the SPARK development process, with PLD transformation and re-test at the end.

7.8.4 Further research

The following research work would likely produce interesting and useful results:

1. a full description of the map from sequential SPARK 95 to VHDL;
2. production of SPARK and VHDL design patterns for common PLD-based functionality, and development of an algorithm or heuristic for selecting the design of the SPARK-PLD interface; and
3. a study of the information-flow results of transforming a polling-loop single-process program into appropriate SPARK Ravenscar tasks.

Chapter 8

Conclusions

In this final chapter we draw up the lessons we have learned in our progress through this thesis, show how they have clarified the problems of hardware-software co-design, and look at the avenues for future research which have opened up as a result.

8.1 Solving the Original Problem

Our original research problem statement in Chapter 3 was:

What methodology is suitable for developing a set of safety-critical system requirements into an implementation which executes partially in a conventional microprocessor and partly on a programmable logic device?

Such a methodology should be rigorous and formal enough to admit verification and validation to the standards demanded by DefStan 00-54 and RTCA DO-254 (electronic hardware), DefStan 00-55 (software) and DefStan 00-56 (system safety) for SIL-3 and SIL-4 systems (RTCA DO-254 Level A and B).

We break this down into the following components; for each component we measure what progress we have made against the above goal. We also list the original targets from Chapter 3 which have been covered.

8.1.1 PLDs in safety-critical systems

We have surveyed the existing major safety and software development standards relevant to PLDs in safety-critical systems. We have extracted the key points from these standards and applied them in an example development. Because we based this work on existing best-practice standards we are on solid ground for justifying the safety and correctness of this development to a safety authority.

Current expert opinion[Pri03] is that the existing PLD technologies do not permit SIL-3 or SIL-4 functionality to be incorporated in a PLD. The rigorous formal techniques proposed in this thesis appear to provide similar rigour to that required for SIL-3 software developments, therefore there is a reasonable case that with this work SIL-3 PLD functionality is now feasible for some systems.

The author's experience is that PLD programs can be designed to satisfy the requirements of DO-254 Level A criticality, as long as formal methods (an optional part

of Level A safety arguments) are not required. This thesis provides suitable rigorous techniques for specifying and analysing synchronous PLD programs, thus supporting DO-254 Level A development by making formal methods use practical.

Targets fulfilled:

Target 1: *The process we define must be rigorous.*

Target 2: *The process must help the developer to write unambiguous programs.*

Target 10: *The process should provide flexibility so that it may be used in situations not anticipated in its original design.*

Target 11: *The process must admit justification to the project safety authority that the programs output by the process are of an adequate integrity level.*

8.1.2 Rigorous PLD programming

We have combined the SRPT process algebra and Morgan's refinement calculus to provide a synchronous timed refinement calculus for developing SRPT processes into Pebble programs. The calculus allows for reasoning about the behaviour of arbitrary SRPT processes incorporated into an otherwise formally developed system. As well as stepwise refinement of designs, the calculus admits trace-based proof of safety properties of processes.

We have demonstrated a practical refinement from a timed specification into a device-agnostic unambiguous implementation language (Pebble, with a semantics defined by SRPT), and demonstrated its accuracy via gate-level simulation. The simulation environment is available in an operating-system-neutral format for future use.

Targets fulfilled:

Target 1: *The process we define must be rigorous.*

Target 2: *The process must help the developer to write unambiguous programs.*

Target 3: *The process must allow the programs to have sections written in a low-level language for speed and flexibility, but not allow these sections to compromise overall program reliability.*

Target 5: *The program produced must be easy to test.*

Target 6: *The program must be able to be compiled onto a range of existing and anticipated PLDs.*

Target 7: *The process must reuse existing proven tools where feasible.*

Target 10: *The process should provide flexibility so that it may be used in situations not anticipated in its original design.*

Target 12: [00-54 8.5.2] *The analytical arguments provided shall include:*

- (i) *any formal arguments used in validation to show that the formal specification complies with the safety requirements;*
- (ii) *any formal arguments that the functional design satisfies the formal specification;*
- (iii) *for non-functional properties with specified safety requirements, analysis of the achieved behaviour, e.g.: performance, timing etc.;*
- (iv) *analysis of the effectiveness of fault mitigation, for example use of such techniques as diverse implementations.*

Target 14: [00-54 13.3.1] *A Hardware Specification shall be produced which defines the SREH in terms of its behaviour and properties.*

8.1.3 Mapping SPARK to hardware

We have shown how the SPARK Ada critical systems programming language is well-suited to describing PLD programs, due to its formal definition and the analysis tools which support it. We have examined the problem of compiling SPARK program constructs to hardware in three different ways.

We have shown how SPARK programs can be developed and proven against formal pre- and post-condition specifications using current tools and techniques. We have shown how these pre- and post- conditions can be used as the basis for developing an SRPT program that satisfies the specification, ignoring the actual SPARK code.

We have described how SPARK code can be compiled directly to circuits on PLDs, taking advantage of Ada's type system to reduce datapath sizes and taking advantage of SPARK Ada program structure to simplify the compilation task. We examined the trade-offs between PLD gate count and program execution speed with particular regard to the implementation of data paths on the PLD.

We have provided a full SRPT specification for a (reduced) sequential SPARK 95 interpreter which demonstrated that a) SRPT can be used to specify large systems and b) the information known at compile-time about SPARK programs contributes substantially to effective implementation in hardware. The interpreter was not useful for high integrity programs, since high integrity programming requires compilation rather than interpretation of SPARK programs, but would be acceptable for low integrity programs and demonstrated the use of SRPT for PLD program design.

We have examined the problem of identifying and extracting a fragment from a SPARK Ada program for PLD execution, maintaining program correctness. We demonstrated the technique for an industrial-scale embedded program.

Because we used a generic PLD model for this work we avoided restricting this development to a particular class of PLD.

Targets fulfilled:

Target 1: *The process we define must be rigorous.*

Target 2: *The process must help the developer to write unambiguous programs.*

Target 4: *The process must admit substantial static analysis to discover semantic program errors at or before compile time.*

Target 6: *The program must be able to be compiled onto a range of existing and anticipated PLDs.*

Target 7: *The process must reuse existing proven tools where feasible.*

Target 8: *The process must guide the developer in the appropriate use of each component.*

Target 9: *The process should indicate what kinds of error may arise at each stage.*

Target 12: [00-54 8.5.2] *The analytical arguments provided shall include:*

- (i) *any formal arguments used in validation to show that the formal specification complies with the safety requirements;*
- (ii) *any formal arguments that the functional design satisfies the formal specification;*
- (iii) *for non-functional properties with specified safety requirements, analysis of the achieved behaviour, e.g.: performance, timing etc.;*

- (iv) *analysis of the effectiveness of fault mitigation, for example use of such techniques as diverse implementations.*

8.1.4 The system development process

We have defined a rigorous development process for going from a formal specification to SPARK and PLD implementation. This development process involves:

- early identification of PLD and software components;
- use of existing software design methods and analysis tools to produce high-integrity SPARK code for the system;
- use of refinement techniques to produce a provably correct PLD program (such as the carry look-ahead adder);
- the ability to simulate PLD functionality without significant change to the SPARK program;
- the option to transform software components to PLD form at a late stage without compromising system design or safety;
- continuous production of evidence that the system is fit for purpose and fulfils its required safety properties; and
- the option to move the (formally defined) program components between software and PLD during future system upgrades.

Targets fulfilled:

Target 1: *The process we define must be rigorous.*

Target 5: *The program produced must be easy to test.*

Target 8: *The process must guide the developer in the appropriate use of each component.*

Target 9: *The process should indicate what kinds of error may arise at each stage.*

Target 10: *The process should provide flexibility so that it may be used in situations not anticipated in its original design.*

Target 11: *The process must admit justification to the project safety authority that the programs output by the process are of an adequate integrity level.*

Target 12: [00-54 8.5.2] *The analytical arguments provided shall include:*

- (i) *any formal arguments used in validation to show that the formal specification complies with the safety requirements;*
- (ii) *any formal arguments that the functional design satisfies the formal specification;*
- (iii) *for non-functional properties with specified safety requirements, analysis of the achieved behaviour, e.g.: performance, timing etc.;*
- (iv) *analysis of the effectiveness of fault mitigation, for example use of such techniques as diverse implementations.*

Target 13: [00-54 12.1.2] *The Design Plan shall define the life cycle that is to be followed in the development of the custom circuit, including a specification process, a development process and a verification process.*

8.1.5 Reliability and practicability

In Section 3.11 we listed general questions about the development process which aimed to measure the process's reliability and practicability. We now answer them.

How many distinct stages are there in the methodology?

Two extra stages have been introduced into the standard software development process: identifying parts of the specification to refine directly to hardware, and identifying parts of the SPARK Ada program to compile into hardware.

The refinement process itself has four stages: rewrite the specification, refine it to SRPT, compile to Pebble/VHDL and test it.

The SPARK Ada program fragment extraction has five stages: rewrite the SPARK package body, update the package specification annotation, map the original SPARK body into VHDL, test the VHDL in isolation and then test the SPARK-PLD interaction.

What is the probability and effect of introducing an error at each stage?

We have not gathered numeric data on probabilities, but can estimate the effect of errors from experience in software development.

Incorrectly rewriting the specification for SRPT refinement is likely to make the entire refinement incorrect and, if detected, will probably require the refinement to be re-done. Whether it is detected will depend on the depth of system testing against the original system specification.

Making an error in SRPT refinement is likely, in our experience noted in Section 5.3.6, to be picked up during PLD program testing.

Making an error in extracting the SPARK Ada program fragment into a PLD is likely to be picked up in testing, especially if test results for the software implementation are compared against those for the PLD implementation.

What do the above imply for the reliability of the system as a whole?

The reliability of a system function refined into an SRPT program, where the refinement has been independently checked, is likely to be high. This does assume that the original specification was correct.

Extracting a SPARK Ada program fragment into PLD form is likely to make the program less reliable, but the alternatives (writing the PLD program in VHDL or a high-level language from scratch) remove the ability to compare diverse implementations of the PLD program and are more error-prone than Ada implementation in the same way that assembly language or C program development is more error-prone than Ada program development.

What classes of error are specifically checked for in the development process?

Information-flow, control-flow and data-flow errors are checked for by the SPARK Examiner. Numeric overflow and proof condition violation are checked for by the SPADE Simplifier and manual inspection of VCs. Errors in the SRPT refinement process are checked for by independent inspection of the refinement steps. Errors in PLD program extraction are checked for by comparing all-software and software-PLD implementation results.

Is there adequate tool support for the developers of the target systems?

A qualified yes. The SPARK Examiner and SPADE toolset already exist and are mature. The York hardware compiler for Ada exists, although has not yet been shown to be effective at typical industrial system sizes. There is as yet no tool support for

SRPT refinement.

What level of technical expertise, and how much time, is required for each development stage?

Refinement of a specification into an SRPT process requires a good understanding of logic in general, and technical expertise in refinement in particular.

SPARK Ada program development requires basic imperative programming skills. SPARK Ada proof work requires an understanding of first-order logic. Extracting a SPARK program fragment into a PLD program requires an understanding of VHDL and the ability to operate PLD compilation and simulation tools.

Given appropriate same-generation hardware, does the generic PLD implementation produced have significant performance advantages over an all-software implementation?

We have not produced performance figures which answer this question. It was established in Section 2.3.10 that PLD programs could significantly outperform microprocessor programs for some tasks, and we have shown that SRPT refinement allows a high-performance PLD program to be developed from a specification, but we have not shown whether Ada code compiled onto a PLD can run more quickly than on a contemporary microprocessor.

How well does the process allow late changes in requirements to be incorporated into the system?

If the requirements can be traced into the design, the data-flow and information flow annotations of SPARK Ada can bound the program units which must be examined to see if changes are necessary. The abstraction present throughout the system may reduce the impact of some requirements changes, but this is not certain. If refinement is used, requirements change may require some refinements to be redone from scratch which will be labour-intensive.

8.2 Advancement of Knowledge

We outline the weaknesses of the current research, in what respects our research is original and how it improves on the current research.

8.2.1 Current weaknesses

Section 2.6.1 described the weaknesses of the current research, which can be summarised as:

- there is no relation of high-level PLD programming languages to the requirements of DO-254 and DefStan 00-54;
- there is no relation of synchronous parallel specification and analysis techniques to the requirements of DO-254 and DefStan 00-54;
- Ada is the only high-level language suitable for programming high-integrity systems, and the existing PLD compilers for it are immature and omit rigour; and
- there is a general lack of demonstration that PLD design and programming techniques for high-integrity will scale to be practical for typical modern systems.

8.2.2 Originality

The main direction of research in this thesis is original because the problem of producing demonstrably correct PLD programs, suitable for use in high-integrity systems, has been specified (in Defence Standard 00-54[MoD99] and RTCA DO-254[RTC00]) but has not been solved. There has been no published work that explicitly addresses the problems raised by conforming to 00-54 and DO-254 in PLD program development.

The work on specification and refinement of synchronous parallel systems (using SRPT) is not original in itself, as Barnes[Bar93] specified SRPT and demonstrated its use in system specification, and Morgan, Back and others[Mor94, BvW94] demonstrated rigorous calculi for refinement in synchronous systems. It is original in that it provides a full refinement calculus for SRPT, making SRPT practical for specification and refinement of PLD programs. It is original in relating the work explicitly to the requirements of 00-54 and DO-254 for high-criticality systems. It is also original in describing the practical translation of the refined program into a PLD-compilable form.

The work on compilation of SPARK Ada into PLDs is not original in itself, as Sheraga[She96] and Ward[WA01, WA02c] have investigated Ada and SPARK Ada compilation for PLDs. It is original in that it exploits the properties of SPARK Ada to increase confidence in the correctness of the compilation and optimise the PLD program for space and execution time. It is also original in relating the work explicitly to the requirements of 00-54 and DO-254 for high-criticality systems.

The PLD-software development process proposal is original in that it explicitly addresses the requirements of 00-54 and DO-254. It is also original in identifying the problems that arise throughout the software-PLD process and providing solutions to them. It expands the domain of applications for which the SPARK Ada programming language can be used. It is original in that it details an industrial-scale safety-critical embedded system and applies appropriate parts of the development process to move an identified part of the program into programmable hardware.

8.2.3 Advances made

The research from thesis has been fed into the production of a practical guide to certifying PLD programs for safety-critical avionics[Hil03a]. As such, it has already made a practical contribution to the production of safety-critical PLD programs.

The advances made by this research are:

- a practical process for high-integrity programming of PLDs (Section 3.12);
- a refinement calculus for SRPT (Chapter 5);
- a mapping which permits SRPT programs to be compiled directly onto PLDs (Section 4.2.7);
- a publicly-available simulator to support simulation of programs generated by this mapping (Section 7.2);
- a design for mapping SPARK Ada programs onto PLDs (Section 7.7);
- a design for a SPARK Ada interpreter to run on a PLD (Chapter 6); and

- a substantial example of a safety-critical program to be used in future hardware compilation work (Section 7.3).

Overall, this research has made feasible the production of programs that satisfy the requirements of Defence Standard 00-54 for SIL-3 and SIL-4 systems, which was not feasible before.

8.3 Self-Critique

We now consider the omissions and weaknesses of this research. We also consider how PLD program development would proceed if this research was not around, and how this research is an improvement.

8.3.1 Omissions

The major omissions from this work are:

1. the demonstration of the PLD programs we produced being compiled into netlists, simulated with commercial FPGA simulators and run on real FPGAs;
2. the demonstration of a SPARK program communicating with a real FPGA;
3. the construction of a formal safety case for the case study including hazard identification and fault tree analysis; and
4. relation of this work to information security standards such as the Common Criteria[Com99].

The first three omissions mean that the practicality of the techniques described in this research is not yet demonstrated. They also leave open the integration of the proposed process into a full safety-critical system development, and its assessment by an independent safety authority. Until this is done it is not possible to say with confidence that these techniques and this process are suitable for SIL-3 software development.

The final omission is an area that is suitable for future research. SPARK Ada has already been demonstrated in high-security applications such as the MULTOS CA[AC02]. We consider this further in Section 8.4.4.

8.3.2 Weaknesses

The major weaknesses of the components of this work are that:

1. we have not considered how to take advantage of design features of existing PLDs (such as embedded processor cores);
2. our focus on SPARK has excluded the Ravenscar tasking profile, which appears to be helpful to construction of parallel SPARK programs; and
3. we have not established how the SRPT refinement system scales with increasing complexity of the specification.

The general issue of how well refinement techniques scale up is an open topic and is being examined in planned UK refinement research. We anticipate that useful information relevant to SRPT refinement will arise from this research in the next 1-2 years.

The omission of Ravenscar is, to some extent, the result of the timing of this research. SPARK Ravenscar has only just been officially released, and so it was difficult to make specific recommendations about using it in the context of PLD programming. Ravenscar will clearly become important in the construction of safety-critical parallel Ada systems in future years, particularly when Ada 0Y (the successor to Ada 95) is finalised.

8.3.3 How the state of the art would evolve without this research

We now consider how the state of the art of PLD programming for high-integrity systems would develop if this research had not been done or had not been published.

Safety-critical PLD program development

UK Interim Defence Standard 00-54 and RTCA DO-254 are already published, and so future safety-critical PLD programs would have to conform to them in any case. However, the formal methods recommendations in both standards have not been addressed in current PLD program developments. Without a clear demonstration that formal specification and development of PLD programs is practical, and guidance on the use of specific methods, the incorporation of formal methods in industrial PLD developments is likely to be haphazard.

Notably, DefStan 00-54 is only an interim standard and its contents will be amended when it becomes part of Issue 3 of Defence Standard 00-56 in 2004. If industrial developers believe that the requirements for SIL-3 and SIL-4 PLD program development are impractical then they are likely to lobby for the SIL-3 and SIL-4 requirements to be ameliorated. This would be bad for system safety, and in the end is likely to increase the cost of systems; experience by major hardware developers such as Intel[Sch03] shows that formal verification for hardware can make economic sense.

Refinement for synchronous parallel systems

There is already a range of refinement calculi for synchronous parallel systems. However, these have not been applied to practical PLD developments and so it is not yet possible to go from a formally refined system to a compiled PLD implementation and argue that semantics and correctness have been preserved. Without this assurance, the motivation for use of formal specification and refinement in PLD program design is significantly reduced.

PLD high-level programming

Languages such as Handel-C are likely to be used increasingly in PLD program development in the coming years. Without a practical high-integrity competitor such as

Ada, they are likely to start to be used for high-integrity PLD programming despite the manifest deficiencies of the C language in this respect.

The work by Ward and Audsley[WA01, WA02b] on hardware compilation of SPARK Ada and Ravenscar is promising but it remains to be seen whether it is practical for real systems and whether the correctness of the compilation process can be justified. Without better exploitation of the known information flow and semantics of SPARK Ada programs, this compilation will not be as effective as it could be.

8.4 Future Work

There are several major areas of work opened up by this thesis which remain unexplored. We now state what they are and outline how one might start to address them.

8.4.1 Safety engineering with PLDs

As noted in Section 8.3.2 it is necessary to obtain a safety engineering perspective on the processes described in this thesis. This requires the input of experienced safety engineers and safety assessors.

A useful start would be to produce a generic guidance document for incorporating PLDs into critical systems, along the lines of UK Defence Standard 00-54 but brought up to date with current PLD technologies.

There is an ongoing project by the UK defence establishment to produce a document similar to this, restricted to the problem of incorporating PLDs into Advanced Avionics Architectures (AAvA) compliant systems. The first release of this document[Hil03a] has been informed by the research in this thesis. Future releases of the document will incorporate the lessons learned from a suitable case study.

RTCA DO-254 is a useful support to safety-critical PLD programming work, but its Appendix B on high-integrity PLD programming would similarly benefit from such a guidance document.

8.4.2 Refinement

We have produced a rigorous basis for refinement in SRPT in Chapter 5. The refinement rules produced were adequate for our demonstration study but there is a clear need to extend them if other, more ambitious systems are to be refined.

We suggest the study and extension of the existing refinement rules for SRPT, building up a parametrised library of useful processes. Generic arithmetic routines would be one class of such processes. This work should then be applied to the implementation of a substantial critical function on a PLD.

We have only considered SRPT refinement in isolation. As noted in Section 2.3.11, a hybrid formal specification language such as *Circus* may be appropriate for specifying a combined hardware-software system.

We suggest using *Circus* (or a receptive, synchronous variant of it) to specify a complete software-PLD system, refining it down into appropriate components. This work should use a combination of full refinement, proof of selected safety properties and static analysis. The aim should be to identify and address deficiencies in the existing notations and tool support.

8.4.3 SPARK to PLDs

Our efforts in translating SPARK Ada subsections to PLDs have been demonstrative in nature and purely manual in practice. To make SPARK Ada usable as a PLD programming language, this translation should be mostly automatic and well-supported by tools.

We suggest producing an automatic or semi-automatic tool to translate SPARK Ada into a form suitable for compilation into a PLD. It should be tested out on a range of SPARK 95 code, measuring the size and complexity of the PLD programs produced. It may also be useful to study ways to optimise the PLD programs with respect to gate count and execution time.

The York hardware compiler described by Ward[WA02c] may be a suitable basis for this work but requires critical study in the light of the issues raised by this thesis. Ravenscar is a good deterministic tasking model, and its use should be integral to compiler development.

The SPARK interpreter specified in Chapter 6 has not been implemented in any way. Implementing a restricted version of the interpreter will test the practical usability of the SRPT specification, and should be used to measure metrics including:

- effort / productivity payoff of interpreting versus compiling SPARK;
- PLD space usage and routability of the interpreter and directly compiled SPARK code; and
- run-time performance of interpreted versus compiled SPARK code.

8.4.4 Security applications

Our work has been done with reference to the requirements of RTCA DO-254 and DefStan 00-54. This covers the domain of safety-critical systems, but many aspects of safety are mirrored in the requirements for high security applications.

The definitive information security standard is currently the Common Criteria[Com99]. A comparison of the criteria in this document against the Defence Standards and RTCA documents would be required to identify:

- how current PLD programming practice for security systems is deficient;
- how applicable are the methods illustrated in this thesis; and
- what additional analysis or programming techniques may be mandatory or helpful for the security domain.

8.5 Concluding Thought

The discipline of software engineering dates from around 1968, when the first NATO conference on software engineering was held [Nor68] and Dijkstra made his proposal about reducing the use of GOTO [Dij68]. In the thirty five years that have followed, we have made steady progress to the point today where we have a wealth of languages,

tools and techniques to support the discipline of producing sufficiently reliable, well-engineered software for execution on microprocessors.

This thesis aimed to translate these techniques into the emerging field of programming PLDs. We used unambiguous formal notations to specify PLD programs so that we knew what they should produce. Developing a refinement system allowed us to produce PLD programs that were provably correct. The high-integrity programming language SPARK Ada allowed us to produce a program design amenable to hardware-software partitioning. The properties of the language proved useful in mapping program segments into a PLD-compatible form. We demonstrated that combined hardware-software development at high integrity levels was practical for a substantial embedded system.

We conclude that existing software engineering practice *does* translate into PLD programming, and recommend that it is applied as soon as possible to critical PLD-based systems. We must not forget the lessons we have learned in the microprocessor field: thirty five years is too long to wait for highly reliable PLD programs.

Bibliography

- [AASR98] P. Reinhart A. Abo Shosha and F. Rongen. Reconfigurable PCI-bus interface (RPCI). In Hartenstein and Keevallik [HK98], pages 485–489.
- [AB00] Jörg Abke and Erich Barke. CoMGen: Direct mapping of arbitrary components into LUT-based FPGAs. In Hartenstein and Grünbacher [HG00], pages 191–200.
- [Abr96] J-R Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [AC02] Peter Amey and Rod Chapman. Industrial strength exception freedom. In *Proceedings of ACM SIGAda Annual International Conference*. ACM Press, December 2002.
- [ACM96] *ACM Computing Surveys*, volume 28, December 1996.
- [ACM01] ACM SIGDA. *ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays (FPGA'01)*. ACM Press, February 2001.
- [ACM03] ACM SIGDA. *Eleventh ACM International Symposium on Field-Programmable Gate Arrays*. ACM Press, February 2003.
- [Ame99] Peter Amey. SPARK – the SPADE Ada Kernel. Technical Report 1.0, Praxis Critical Systems Ltd., 1999.
- [Ame00] Peter Amey. INFORMED design method for SPARK. Technical report, Praxis Critical Systems Ltd., October 2000.
- [ARB99] Perry Alexander, Murali Rangarajan, and Phillip Baraona. A brief summary of VSPEC. In Wing et al. [WWD99], pages 1068–1088.
- [Arn96] J. M. Arnold. *Software Architecture*, chapter 5, pages 46–59. Volume 1 of Buell et al. [BAK96], 1996.
- [BAK96] D. A. Buell, J. M. Arnold, and W. J. Kleinfelder, editors. *Splash 2: FPGAs in a Custom Computing Machine*. IEEE Computer Society Press, California, 1996.
- [Bar93] Janet E. Barnes. A mathematical theory of synchronous communication. Technical report, Oxford University Computing Laboratory, 1993.
- [Bar97] John Barnes. *High Integrity Ada – The SPARK Approach*. Addison-Wesley, 1997.

- [Bar03] John Barnes. *High Integrity Software: The SPARK Approach to Safety And Security*. Addison Wesley, April 2003.
- [BDL96] C. W. Barrett, D. L. Dill, and J. R. Levitt. Validity checking for combinations of theories with equality. In M. Srivas and A. Camilleri, editors, *Proceedings of FMCAD'96*, volume 1166 of *Lecture Notes in Computer Science*. Springer-Verlag, November 1996.
- [BDR98] Alan Burns, Brian Dobbing, and George Romanski. The Ravenscar tasking profile for high integrity real-time programs. In L. Asplund, editor, *Reliable Software Technologies, Proceedings of the Ada Europe Conference*, volume 1411 of *Lecture Notes In Computer Science*, pages 263 – 275. Springer-Verlag, June 1998.
- [Ber00] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Foundations of Computing. MIT Press, 2000.
- [BHKW00] T. Bartzick, M. Henze, J. Kickler, and K. Woska. Design of a fault-tolerant FPGA. In Hartenstein and Grünbacher [HG00], pages 151–156.
- [BP98] R. Banach and M. Poppleton. Retrenchment: An engineering variation on refinement. In D. Bert, editor, *B-98: Recent Advances in the Development and Use of the B Method*, volume 1393 of *Lecture Notes in Computer Science*, pages 129–147, April 1998.
- [BPG00] Jürgen Becker, Thilo Pionteck, and Manfred Glesner. DReAM: A dynamically reconfigurable architecture for future mobile communication applications. In Hartenstein and Grünbacher [HG00], pages 312–321.
- [Bro95] Frederick P. Brooks, Jr. *The mythical man month: essays on software engineering*. Addison Wesley Longman Inc., anniversary edition, 1995.
- [BvW94] Ralph-Johan Back and Joakim von Wright. Trace refinement of action systems. In *International Conference on Concurrency Theory*, pages 367–384, 1994.
- [CB85] Bernard Carré and J.-F. Bergeretti. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, January 1985.
- [Cel02] Celoxica Ltd. *Handel-C Language Reference Manual*, 3.1 edition, 2002.
- [CEN99] CENELEC. Railway applications - the specification and demonstration of dependability, reliability, availability, maintainability and safety. Technical Report EN 50126, European Committee for Electrotechnical Standardization, 1999.
- [CEN02a] CENELEC. Railway applications - safety-related electronic systems for signalling. Technical Report EN 50129, European Committee for Electrotechnical Standardization, 2002.

- [CEN02b] CENELEC. Railway applications - software for railway control and protection systems. Technical Report EN 50128, European Committee for Electrotechnical Standardization, 2002.
- [Cha94] R. Chapman. Worst-case timing analysis via finding longest paths in SPARK Ada basic-path graphs. Technical report, Department of Computer Science, York University, October 1994.
- [Cha01] Rod Chapman. SPARK Examiner release note - release 6.0. Technical report, Praxis Critical Systems Ltd., August 2001.
- [Cha03] Rod Chapman. SPARK Examiner release note - release 7.0. Technical report, Praxis Critical Systems Ltd., August 2003.
- [Civ02] Civil Aviation Authority. *CAP 670 ATS Safety Requirements*, June 2002. SW01 Regulatory Impact Assessment.
- [cJ99] ISO committee JTC 1/SC 22. *Ada: Conformity assessment of a language processor*. ISO/IEC, December 1999.
- [CJR98] Stephen Charlwood and Philip James-Roxby. Evaluation of the XC6200-series architecture for cryptographic applications. In Hartenstein and Keevallik [HK98], pages 218–227.
- [CKG01] Pawel Chodowiec, Po Khuon, and Kris Gaj. Fast implementations of secret-key block ciphers using mixed inner- and outer-round pipelining. In *ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays (FPGA'01)* [ACM01], pages 94–102.
- [CKRB03] Chen Chang, Kimmo Kuusilinna, Brian Richards, and Robert W. Brodersen. Implementation of BEE: a real-time large-scale hardware emulation engine. In *Eleventh ACM International Symposium on Field-Programmable Gate Arrays (FPGA'03)* [ACM03], pages 91–99.
- [Com90] IEEE Committee. Standard glossary of software engineering technology. Technical Report 610.12, Institute of Electrical and Electronics Engineers, inc., 1990.
- [Com91] Communications Electronics Security Group. *Information Technology Security Evaluation Criteria (ITSEC), Provisional Harmonised Criteria*, June 1991.
- [Com99] Common Criteria. *Common Criteria for Information Technology Security Evaluation*, August 1999.
- [Cor99] Actel Corporation. ProASIC 500K family datasheet. Technical report, Actel Corporation, 1999.
- [CS⁺96] R. Cleaveland, S. Smolka, et al. Strategic directions in concurrency research. In ACM96 [ACM96].

- [CS00] Koen Claessen and Mary Sheeran. *A Tutorial on Lava: A Hardware Description and Verification System*, August 2000.
- [CSW02] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. Refinement of actions in Circus. In Derrick et al. [DBWvW02].
- [Cur84] I. F. Currie. Orwellian programming in safety-critical systems. Technical Report Memorandum 3924, Royal Signals and Radar Establishment, 1984.
- [CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. In ACM96 [ACM96].
- [DBWvW02] John Derrick, Eerke Boiten, Jim Woodcock, and Joakim von Wright, editors. *Proceedings of REFINE 2002*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier, November 2002.
- [Dij68] Edsger W. Dijkstra. Go To statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.
- [Dij70] Edsger W. Dijkstra. Notes on structured programming. circulated privately, April 1970.
- [Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.
- [Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
- [DM41] B. Dushnik and E. W. Miller. Partially ordered sets. *American Journal of Mathematics*, 63:600–610, 1941.
- [DMH03] Dewi Daniels, Richard Myers, and Adrian Hilton. White box software development. In F. Redmill and T. Anderson, editors, *Proceedings of the Eleventh Safety-Critical Systems Symposium*. Praxis Critical Systems Ltd., Springer-Verlag, February 2003.
- [Don98] Adam Donlin. Self-modifying circuitry — a platform for tractable virtual circuitry. In Hartenstein and Keevallik [HK98], pages 199–208.
- [DvLF93] A. Dardenne, A. van Lansweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20, 1993.
- [EK99] Alexander Egyed and Philippe B. Kruchten. Rose/architect: a tool to visualize architecture. In *Proceedings of the 32nd Annual Hawaii Conference on Systems Sciences*, 1999.
- [EL02] Lars-Henrik Eriksson and Peter Alexander Lindsay, editors. *FME 2002: Formal Methods – Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*. Springer-Verlag, July 2002.

- [FKZ75] R. Farrow, K. Kennedy, and L. Zucconi. Graph grammars and program flow analysis. In *Proceedings of 17th IEEE Symposium on Foundations of Computer Science*, pages 42–56. IEEE, 1975.
- [FMA⁺97] Julio Faura, Juan Manuel Moreno, Miguel Angel Aguirre, Phuoc van Duong, and Josep Maria Insenser. Multicontext dynamic reconfiguration and real-time probing on a novel mixed signal programmable device with on-chip processor. In Luk et al. [LCG97], pages 1–10.
- [For97] Formal Systems (Europe) Ltd. *FDR User Manual*, May 1997.
- [Fou00] The Free Software Foundation. GNU C Compiler home page, January 2000. <http://www.gnu.org/software/gcc/gcc.html>.
- [FW99] Gavin Finnie and Ross Wintle. SPARK 95 – the SPADE Ada 95 Kernel. Technical Report 1.0, Praxis Critical Systems Ltd., October 1999.
- [GA99] Wally Gibbons and Harry Ames. Use of FPGAs in critical space flight applications – a hard lesson. In *1999 Military and Aerospace Applications of Programmable Devices and Technologies Conference*. Space Dynamics Laboratory, Utah State University, September 1999.
- [GC90] Jonathan Garnsworthy and Bernard Carré. SPARK - an annotated Ada subset for safety-critical systems. *Proceedings of Baltimore Tri-Ada Conference*, 1990.
- [GN99] Paul Graham and Brent Nelson. Reconfigurable processors for high-performance, embedded digital signal processing. In Patrick Lysaght, James Irvine, and Reiner Hartenstein, editors, *Field-Programmable Logic and Applications*, volume 1673 of *Lecture Notes In Computer Science*, pages 1–10, Glasgow, UK, September 1999. Springer-Verlag.
- [Hal96a] J. A. Hall. Using formal methods to develop an ATC information system. *IEEE Software*, 12(6), March 1996.
- [Hal96b] J. G. Hall. *An Algebra of High-Level Petri Nets*. PhD thesis, University of Newcastle upon Tyne, 1996.
- [Hal02] Anthony Hall. Correctness by construction: integrating formality into a commercial development process. In Eriksson and Lindsay [EL02].
- [Hea97] Health and Safety Executive. *Four Party Regulatory Consensus Report on the Safety Case for Computer-Based Systems in Nuclear Power Plants*, November 1997.
- [Hei98] Constance Heitmeyer. On the need for practical formal methods. In A. P. Ravn and H. Rischel, editors, *Formal Techniques in Real Time and Fault Tolerant Systems (5th International Symposium)*, volume 1486 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1998.
- [Hen88] Michael Hennessey. *Algebraic Theory of Processes*. MIT Press, 1988.

- [HG00] Reiner W. Hartenstein and Herbert Grünbacher, editors. *Proceedings of the 10th International Conference on Field Programmable Logic and Applications (FPL'00)*, volume 1896 of *Lecture Notes In Computer Science*. Springer-Verlag, August 2000.
- [HH00] Adrian J. Hilton and Jon G. Hall. On applying software development best practice to FPGAs in safety-critical systems. In Hartenstein and Grünbacher [HG00], pages 793–796.
- [HH02a] Adrian J. Hilton and Jon G. Hall. Mandated requirements for hardware/software combination in safety-critical systems. In *Proceedings of the workshop on Requirements for High-Assurance Systems 2002*. Software Engineering Institute, Carnegie-Mellon University, September 2002.
- [HH02b] Adrian J. Hilton and Jon G. Hall. Refining specifications to programmable logic. In Derrick et al. [DBWvW02].
- [HH03] Adrian J. Hilton and Jon G. Hall. Mandated requirements for hardware/software combination in safety-critical systems. Technical Report 2003/2, The Open University, 2003.
- [HHG98] Reiner W. Hartenstein, Michael Herz, and Frank Gilbert. Designing for Xilinx XC6200 FPGAs. In Hartenstein and Keevallik [HK98], pages 29–38.
- [Hil03a] Adrian Hilton. Practical guide to certification and re-certification of AAvA software elements: Software for programmable logic devices. Technical report, QinetiQ, July 2003.
- [Hil03b] Adrian J. Hilton. Engineering software systems for customer acceptance. In *Proceedings of SEHAS'03*. Praxis Critical Systems Ltd., May 2003.
- [HK98] R. W. Hartenstein and A. Keevallik, editors. *Field-Programmable Logic and Applications: From FPGAs to Computing Paradigm, 8th International Workshop (FPL'98)*, *Proceedings*, volume 1482 of *Lecture Notes In Computer Science*. Springer-Verlag, September 1998.
- [HNT03] Jerker Hammarberg and Simin Nadjm-Tehrani. Development of safety-critical reconfigurable hardware with Esterel. In *Eighth International Workshop on Formal Methods for Industrial Critical Systems*. Linköping University, Elsevier, June 2003.
- [Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
- [HRH01] Jonathan Hammond, Rosamund Rawlings, and Anthony Hall. Will it work? In *Proceedings of the 5th International Symposium on Requirements Engineering*, August 2001.
- [HTH03] Adrian J. Hilton, Gemma Townson, and Jon G. Hall. Fpgas in critical hardware/software systems. Technical Report 2003/1, The Open University, 2003.

- [HW97] John R. Hauser and John Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *FPGAs for Custom Computing Machines (FCCM'97)*. University of California at Berkeley, 1997.
- [IEC86] International Electrotechnical Commission. *Software for Computers in the Safety of Nuclear Power Stations, IEC Standard 880*, first edition, 1986.
- [IEC00] International Electrotechnical Commission. *IEC Standard 61508, Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related Systems*, March 2000.
- [iec02] Z formal specification notation – syntax, type system and semantics, July 2002.
- [IEC03] International Electrotechnical Commission. *IEC Standard 61131, Programmable Controllers, Part 3 (programming languages)*, 2003.
- [IEE91] IEEE. *IEEE Std. 1076-1987: IEEE Standard VHDL Language Reference Manual*, 1991.
- [IEE95] IEEE. *IEEE Std. 1364-1995: IEEE Standard Description Language*, 1995. Based on the Verilog(TM) Hardware Description Language.
- [IEE01] IEEE. *IEEE Standard Test Access Port and Boundary-Scan Architecture*, 2001.
- [Ins97] Institut für Mikroelektronik Stuttgart. *SAND/1 Neurochip Infosheet*, February 1997.
- [Ins02] The Inspector General. Status on the Federal Aviation Administration's major acquisitions. Memorandum, U.S. Department of Transportation, February 2002. http://www.oig.dot.gov/show_txt.php?id=701.
- [Int93] International Organisation for Standardisation. *ISO/IEC 8809:1989; LOTOS: A formal description technique based on the temporal ordering of observational behaviour*, 1993.
- [Int95] Intermetrics Inc. *Ada 95 Reference Manual International Standard ANSI/ISO/IEC-8652:1995*. U.S. Department of Defense, January 1995.
- [Int96] International Electrotechnical Commission. *Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language*, December 1996.
- [Int00a] International Electrotechnical Commission. *IEC Standard 61690-1, Electronic Design Interchange Format (EDIF version 3.0.0)*, 2000.
- [Int00b] International Electrotechnical Commission. *IEC Standard 61690-2, Electronic Design Interchange Format (EDIF) version 4.0.0*, 2000.

- [IP96] Valerie Illingworth and Ian Pyle, editors. *Oxford Paperback Reference Dictionary of Computing*. Oxford Paperbacks. Market House Books, February 1996.
- [IS97] Maurice Kilavuka Inuani and Jonathan Saul. Technology mapping of heterogeneous LUT-based FPGAs. In Luk et al. [LCG97], pages 223–234.
- [Jef91] A. Jeffrey. Discrete timed CSP. PMG Memo 78, Programming Methodology Group, Chalmers University, Sweden, 1991.
- [Joh78] S. C. Johnson. *Lint, a C Program Checker, Unix Programmer's Manual*. AT&T Bell Laboratories, 1978.
- [Jon86] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 1986.
- [Jos92] Mark Josephs. Receptive process theory. *Acta Informatica*, 29:17–31, 1992.
- [JS90] Geraint Jones and Mary Sheeran. Circuit design in Ruby. In Jørgen Staunstrup, editor, *Formal Methods for VLSI Design*, pages 13–70. North-Holland, 1990.
- [JTS03] Kimmo U. Järvinen, Matti T. Tommiska, and Jorma O. Skyttä. A fully pipelined memoryless 17.8 Gps AES-128 encryptor. In *Eleventh ACM International Symposium on Field-Programmable Gate Arrays (FPGA'03)* [ACM03], pages 207–215.
- [KF91] S. Kopec and D. Faria. Obtaining 70 MHz performance from the MAX architecture. *Electronic Engineering*, pages 69–74, May 1991.
- [KHCP99] Steve King, Jonathan Hammond, Rod Chapman, and Andy Pryor. The value of verification: Positive experience of industrial proof. In Wing et al. [WWD99].
- [Knu77] Donald E. Knuth. Notes on the van Emde Boas construction of priority deques: An instructive use of recursion. Memo to Peter van Emde Boas, March 1977.
- [Kra00] Andrzej Krasniewski. Exploiting reconfigurability for effective detection of delay faults in LUT-based FPGAs. In Hartenstein and Grünbacher [HG00], pages 675–684.
- [KS00] Helena Krupnova and Gabriele Saucier. FPGA-based emulation: Industrial and custom prototyping solutions. In Hartenstein and Grünbacher [HG00], pages 68–77.
- [LB⁺03] David Lewis, Vaughn Betz, et al. The StratixTM routing and logic architecture. In *Eleventh ACM International Symposium on Field-Programmable Gate Arrays (FPGA'03)* [ACM03], pages 12–20.

- [LC96] Nancy Leveson and Stan Correy. Transcript from ‘High Anxiety’. ABC Radio national broadcast, August 1996.
- [LCG97] W. Luk, P. Y.K. Cheung, and M. Glesner, editors. *Field Programmable Logic and Applications: Seventh International Workshop (FPL’97), Proceedings*, volume 1304 of *Lecture Notes In Computer Science*. Springer-Verlag, September 1997.
- [LCR03] Fernanda Lima, Luigi Carro, and Ricardo Reis. Reducing pin and area overhead in fault-tolerant FPGA-based designs. In *Eleventh ACM International Symposium on Field-Programmable Gate Arrays (FPGA’03)* [ACM03], pages 108–117.
- [Lee00] Iain Lees. *Perl Coding Standard*. Praxis Critical Systems Ltd., March 2000.
- [Lev95] Nancy Leveson. *Safeware: System Safety and Computers*. Addison-Wesley Publishing Company, 1995.
- [Lev01] Nancy G. Leveson. Evaluating accident models using recent aerospace accidents. Technical report, Software Engineering Research Laboratory, MIT, June 2001.
- [Lio96] Jacques-Louis Lions. Ariane 5 Flight 501 failure. Technical report, The ESA / CNES Inquiry Board, July 1996.
- [LM98] Wayne Luk and Steve McKeever. Pebble — a language for parametrised and reconfigurable hardware. In Hartenstein and Keevallik [HK98], pages 9–18.
- [LS93] Bev Littlewood and Lorenzo Strigini. Validation of ultrahigh dependability for software-based systems. *Communications of the ACM*, 36(11):69–80, 1993.
- [LS97] Luming Lai and J. W. Sanders. A refinement calculus for communicating processes with state. In Gerard O’Regan and Sharon Flynn, editors, *1st Irish Workshop on Formal Methods: Proceedings*, Electronic Workshops in Computing. Springer, July 1997.
- [LS03] John Launchbury and Satnam Singh. An approach to compiling Cryptol to FPGAs. In *3rd Annual High Confidence Software and Systems Conference, Proceedings*, pages 137–146. Galois Connections and Xilinx, April 2003.
- [Ltd84] INMOS Ltd. *occam Programming Manual*. Prentice-Hall International, 1984.
- [Ltd94a] Program Validation Ltd. Formal semantics of SPARK (abstract syntax). Technical report, Program Validation Ltd., March 1994.
- [Ltd94b] Program Validation Ltd. Formal semantics of SPARK (dynamic semantics). Technical report, Program Validation Ltd., March 1994.

- [Ltd98] B-Core (UK) Ltd. The B-Toolkit, 1998. <http://www.b-core.com/OnLineDoc/BToolkit.html>.
- [Luk99] Wayne Luk. *Introductory Notes for Pebble 3.0*. Imperial College, January 1999.
- [Mak03] Wai-Kei Mak. I/O placement for FPGAs with multiple I/O standards. In *Eleventh ACM International Symposium on Field-Programmable Gate Arrays (FPGA '03)* [ACM03], pages 51–57.
- [MC93] J. D. Morison and A. S. Clarke. *ELLA 2000; a Language for Electronic System Design*. McGraw-Hill Book Company, 1993.
- [McH02] John McHale. The new frontier: Reconfigurable computing. *Military and Aerospace Electronics*, May 2002.
- [MCLS97] P. I. Mackinlay, P. Y. K. Cheung, W. Luk, and R. Sandiford. Riley-2: A flexible platform for codesign and dynamic reconfigurable computing research. In Luk et al. [LCG97], pages 91–100.
- [Meg94] Graham M. Megson, editor. *Transformational Approaches to Systolic Design*. Chapman and Hall, 1994.
- [Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.
- [Mil89] Robin Milner. A complete axiomatisation for observational congruence of finite-state behaviours. *Information and Computation*, 81(2):227–247, May 1989.
- [Mil90] Robin Milner. Operational and algebraic semantics of concurrent processes. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 1201–1242. Elsevier and MIT Press, 1990.
- [MIR98] MIRA. *Guidelines for the Use of the C Language in Vehicle Based Software*, April 1998.
- [MK98] Robert Macketanz and Wolfgang Karl. JVX — a rapid prototyping system based on Java and FPGAs. In Hartenstein and Keevallik [HK98], pages 99–108.
- [ML91] Will Moore and Wayne Luk, editors. *FPGAs: Edited Proceedings of the International Workshop on Field Programmable Logic and Applications*, 1991.
- [MNC95] Møeller-Nielsen and Caprini. Replacing an occam process by a chip. In *Parallel Programming and Applications; Proceedings of Workshop on Parallel Programming and Computation (ZEUS'96)*, 1995.
- [MoD94] Defence Standard 00-42, 1994. Reliability and Maintainability Assurance Guidelines.

- [MoD96] Defence Standard 00-56 issue 2, December 1996. Safety Management Requirements for Defence Systems.
- [MoD97] Defence Standard 00-55 issue 2, August 1997. Requirements for Safety-Related Software In Defence Equipment.
- [MoD99] Interim Defence Standard 00-54 issue 1, March 1999. Requirements for Safety Related Electronic Hardware in Defence Equipment.
- [MoD03] Defence Standard 00-56 issue 3 (public comment draft), July 2003. Safety Management Requirements for Defence Systems.
- [MOH97] Christopher McGee-Osborne and Denton Hall. Management of safety issues – a legal perspective. presentation at ‘Safety Investment in Emerging Urban Transit Systems’: AiC Conferences, March 1997.
- [Mor94] Carroll Morgan. *Programming From Specifications*. Prentice-Hall, second edition, 1994.
- [MW00] John S. McCaskill and Patrick Wagler. From reconfigurability to evolution in construction systems: Spanning the electronic, microfluidic and biomolecular domains. In Hartenstein and Grünbacher [HG00], pages 286–299.
- [NG97] Stuart Nisbet and Steven A. Guccione. The XC6200DS development system. In Luk et al. [LCG97], pages 61–68.
- [NMH99] J. Napier, J. May, and G. Hughes. Implementing software on-line diagnostics in safety-critical systems. In T. Bradley and N. J. Davies, editors, *15th Annual UK Performance Engineering Workshop, Proceedings*. Research Press International, July 1999.
- [Nor68] North Atlantic Treaty Organisation. *NATO Conference on Software Engineering*, 1968.
- [Nor69] North Atlantic Treaty Organisation. *NATO Conference on Software Engineering*, 1969.
- [ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, June 1992.
- [ORS96] E.-R. Olderog, Anders P. Ravn, and Jens Ulrik Skakkebak. Refining system requirements to program specifications. In *Formal Methods for Real-Time Computing*, pages 107–134. Wiley, 1996.
- [PEBB01] J. Penny, A. Eaton, P. G. Bishop, and R. E. Bloomfield. The practicalities of goal-based safety regulation. In Felix Redmill and Tom Anderson, editors, *Proceedings of the 9th Safety-Critical Systems Symposium*, pages 35–48. CAA and Adelard, Springer-Verlag, 2001.

- [PH97] Shri Lawrence Pfleeger and Les Hatton. Investigating the influence of formal methods. *IEEE Computer*, 30(2):33–43, February 1997.
- [Pie95] Laurence Pierre. Describing and verifying synchronous circuits with the Boyer-Moore theorem prover. In Paolo Camurati and Hans Eveking, editors, *CHARME*, volume 987 of *Lecture Notes in Computer Science*, pages 35–55. Springer, October 1995.
- [Pra95] Praxis Critical Systems Ltd. *The SPADE Simplifier*, 1995.
- [Pra98] Praxis Critical Systems Ltd. *The SPADE Proof Checker — User Manual*, January 1998.
- [Pri03] Private conversation with UK expert on PLDs and avionics systems, February 2003. source omitted for reasons of confidentiality.
- [PS93] Ian Page and Mike Spivey. How to program in Handel. Technical report, Oxford University Computing Laboratory, December 1993.
- [Pyg99] C. H. Pygott. A comparison of avionics standards. Technical Report DERA/CIS/CIS3/TR990319/1.0, UK Defence Evaluation and Research Agency, August 1999.
- [Rai00] Railtrack. *Railtrack Engineering Safety Management*, 3.0 edition, January 2000.
- [RCD98] Scott H. Robinson, Michael P. Caffrey, and Mark E. Dunham. Reconfigurable computer array: The bridge between high speed sensors and low speed computing. In Hartenstein and Keevallik [HK98], pages 159–168.
- [Ren00] M. Renovell. A specific test methodology for symmetric SRAM-based FPGAs. In Hartenstein and Grünbacher [HG00], pages 300–311.
- [RL00] David Robinson and Patrick Lysaght. Verification of dynamically reconfigurable logic. In Hartenstein and Grünbacher [HG00], pages 141–150.
- [Rom96] George Romanski. Review of ‘Safer C’ (by Les Hatton). Technical report, Thomson Software Products, January 1996.
- [RS99] Vlad Rusu and Eli Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’99)*, 1999.
- [RTC92] RTCA / EUROCAE. *RTCA DO-178B / EUROCAE ED-12: Software Considerations in Airborne Systems and Equipment Certification*, December 1992.
- [RTC00] RTCA / EUROCAE. *RTCA DO-254 / EUROCAE ED-80: Design Assurance Guidance for Airborne Electronic Hardware*, April 2000.

- [Rus93] John Rushby. Formal methods and the certification of critical systems. Technical Report CSL-93-7, SRI International, December 1993.
- [SA99] Alan Simpson and Mike Ainsworth. White box safety. In *Proceedings: Avionics Conference and Exhibition*. ERA Technology Ltd., 1999. ERA Report 99-0815.
- [SC95] Jim Sutton and Martin Croxford. Breaking through the V&V bottleneck. In M. Toussaint, editor, *Ada in Europe: Second International Eurospace-Ada-Europe Symposium*, volume 1031 of *Lecture Notes In Computer Science*. Springer-Verlag, October 1995.
- [SC00] Susan Stepney and David Cooper. Formal methods for industrial products. In J. P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *First International Conference of B and Z Users, Proceedings*, volume 1878 of *Lecture Notes in Computer Science*, pages 374–393. Springer-Verlag, August 2000.
- [Sch94] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (Blowfish). In B. Preneel, editor, *Fast Software Encryption: Second International Workshop*, volume 1008 of *Lecture Notes in Computer Science*. Springer-Verlag, December 1994.
- [Sch03] Tom Schubert. High level formal verification of next-generation microprocessors. In *Proceedings of the 40th Design Automation Post-Conference*. Intel Corporation, ACM Press, June 2003.
- [SD95] Steve Schneider and Jim Davies. A brief history of Timed CSP. *Theoretical Computer Science*, 138, 1995.
- [Sen92] C. T. Sennett. Demonstrating the compliance of Ada programs with Z specification. In C. B. Jones, R. C. Shaw, and T. Denvir, editors, *5th Refinement Workshop*, eWiC Series, pages 367–378. British Computer Society, 1992.
- [Sha97] Mark Shand. A case study of algorithm implementation in reconfigurable hardware and software. In Luk et al. [LCG97], pages 333–343.
- [Sha02] Natarajan Shankar. Little engines of proof. In Eriksson and Lindsay [EL02], pages 1–20.
- [She96] Robert J. Sheraga. ANSI C to behavioural VHDL translator, Ada to behavioural VHDL translator. *The RASSP Digest*, 3, September 1996.
- [SM95] Mandayam K. Srivas and Steven P. Miller. Applying formal verification to a commercial microprocessor. In Steven D. Johnson, editor, *CHDL '95: 12th Conference on Computer Hardware Description Languages and their Applications*, pages 493–502, August 1995.
- [Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International, second edition, 1992.

- [Spi00] J. M. Spivey. *The fUZZ Manual*. The Spivey Partnership, second edition, 2000.
- [SSC01] Greg Snider, Barry Shackelford, and Richard J. Carter. Attacking the semantic gap between application programming languages and configurable hardware. In *ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays (FPGA'01)* [ACM01], pages 115–124.
- [SSSS00] Sergej Sawitzki, Jens Schönherr, Rainer G. Spallek, and Bernd Straube. Formal verification of a reconfigurable microprocessor. In Hartenstein and Grünbacher [HG00], pages 781–784.
- [Sta95] The Standish Group. *The CHAOS report*, 1995.
- [Ste98] Susan Stepney. Incremental development of a high-integrity compiler: Experience from an industrial development. In *Proceedings of the Third IEEE High-Assurance Systems Engineering Symposium (HASE'98)*, Washington D.C., 1998.
- [Str98] Structured Software Systems. *Cradle White Paper — Overview*, February 1998. Available from <http://www.threesl.com/>.
- [SWCL99] R. Swan, A. Wyatt, R. Cant, and C. Langensiepen. Re-configurable computing. *Crossroads (ACM)*, 5(3), 1999.
- [Swe97] Charles Sweeney. FPGA graphics generator. Technical Report 001, Embedded Solutions Ltd., November 1997.
- [Swe98] Charles Sweeney. Optimal features of hardware platforms. Technical Report 006, Embedded Solutions Ltd., December 1998.
- [Tay01] A. Taylor. IT projects sink or swim. *BCS Review*, pages 61–64, January 2001.
- [Tea93] London Ambulance Service Inquiry Team. Report of the inquiry into the London Ambulance Service. Technical report, South-West Thames Regional Health Authority, 1993.
- [TEC+95] Edward Tau, Ian Eslick, Derrick Chen, Jeremy Brown, and André DeHon. A first generation DPGA implementation. In *Third Canadian Workshop on Field Programmable Devices*, pages 138–143. MIT, May 1995.
- [U.S83] U.S. Department of Defense. *Reference manual for the Ada Programming Language ANSI/MIL-STD-1815A*, January 1983.
- [VBR+96] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, March 1996.

- [Vic98] Andy Vickers. On the use of Jackson's principles to structure the requirements engineering activity. In *Systems Engineering: A Matter of Choice, Fourth Annual Symposium*, pages 41–46, RAF Hendon, June 1998. International Council on Systems Engineering (UK Chapter).
- [WA01] M. Ward and N.C. Audsley. Hardware compilation of sequential Ada. In *Proceedings of CASES 2001*, pages 99–107, 2001.
- [WA02a] M. Ward and N. C. Audsley. Hardware implementation of programming languages for real-time. In *Proceedings of the Eighth IEEE Real-Time Embedded Technology and Applications Symposium (RTAS'02)*, pages 276–284. IEEE, September 2002.
- [WA02b] M. Ward and N. C. Audsley. Hardware implementation of the Ravenscar Ada tasking profile. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. ACM Press, 2002.
- [WA02c] M. Ward and N. C. Audsley. Language issues of compiling Ada to hardware. In *11th International Real Time Ada Workshop*, April 2002.
- [Wil01] Steven J. E. Wilton. A crosstalk-aware timing-driven router for FPGAs. In *ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays (FPGA'01)* [ACM01], pages 21–28.
- [WLL⁺01] Kathryn Weiss, Nancy Leveson, Kristina Lundqvist, Nida Farid, and Margaret Stringfellow. An analysis of causation in aerospace accidents. In *Proceedings of AIAA Space Conference and Exposition 2001*. American Institute of Aeronautics and Astronautics, August 2001.
- [WWD99] J. M. Wing, J. Woodcock, and J. Davies, editors. *World Congress on Formal Methods in the Development of Computing Systems*, volume 1709 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1999.
- [XES99] XS40, XSP board v1.4 user manual, September 1999. http://www.xess.com/xs40-manual-v1_4.pdf.
- [Xil96] Xilinx Inc., 2100 Logic Drive, San Jose, CA. *The Programmable Logic Data Book*, 1996.
- [Xil97] Xilinx. *XC6200 Field Programmable Gate Arrays Advance Product Specification*, April 1997.
- [Xil99a] Xilinx. *QPRO Xilinx 2.5V QML Preliminary Product Specification*, October 1999.
- [Xil99b] Xilinx. *Virtex-E 1.8V Field Programmable Gate Arrays Advance Product Specification*, September 1999.
- [Xil99c] Xilinx. *Virtex-E FAQ*, 1999. http://www.xilinx.com/prs_rls/vtxefaq.htm.

- [Yor97] York Software Engineering. *CADiZ: Computer Aided Design in Z*, 1997.
<http://www.cse-euro.demon.co.uk/yse/products/cadiz/>.

Appendix A

Collated Refinement Rules

The following definition and rules are collated from Chapter 5.

A refinement frame P in a program takes the form:

$$P = \forall t \in \mathbb{N} \cdot \iota X : oY : [[\mathbf{pre}]_t, [\mathbf{post}]_{t+k}]$$

representing the specification “for the process P with input alphabet containing X and output alphabet containing Y , at all times t , if \mathbf{pre} is true at time t then at time $t + k$ \mathbf{post} is true.” k is a constant which will be determined by the timing needs of the program at specification time.

Refinement 1 *Stateless 1-bit function*

$$\begin{aligned} & \forall t \in \mathbb{N} \cdot \iota X : o\{y\} : [\mathbf{true}, [y]_{t+1} = f([X]_t)] \\ \sqsubseteq & \text{CELL}_f[I \setminus X][O \setminus \{y\}] \end{aligned}$$

Refinement 2 *Parallelism*

$$\begin{aligned} & \forall t \in \mathbb{N} \cdot \iota X : o(Y \cup Z) : [\mathbf{pre}, \mathbf{post}_1 \wedge \mathbf{post}_2] \\ \sqsubseteq & \iota X : oY : [\mathbf{pre}, \mathbf{post}_1] \parallel \iota X : oZ : [\mathbf{pre}, \mathbf{post}_2] \end{aligned}$$

whenever:

$$\begin{aligned} & Y, Z \text{ are non-empty and non-intersecting} \\ & \forall V \in \mathbb{B}^{\#Z} \cdot \mathbf{post}_1[Z \setminus V] \equiv \mathbf{post}_1 \\ & \forall W \in \mathbb{B}^{\#Y} \cdot \mathbf{post}_2[Y \setminus W] \equiv \mathbf{post}_2 \\ & \text{where } \mathbb{B}^N \text{ is the set of n-ary boolean strings} \end{aligned}$$

Refinement 3 *Weaken precondition*

If $\mathbf{pre} \Rightarrow \mathbf{pre}'$ then:

$$\forall t \in \mathbb{N} \cdot \iota X : oY : [\mathbf{pre}, \mathbf{post}] \sqsubseteq \forall t \in \mathbb{N} \cdot \iota X : oY : [\mathbf{pre}', \mathbf{post}]$$

Refinement 4 *Strengthen postcondition*

If $\mathbf{post}' \Rightarrow \mathbf{post}$ then:

$$\forall t \in \mathbb{N} \cdot \iota X : oY : [\mathbf{pre}, \mathbf{post}] \sqsubseteq \forall t \in \mathbb{N} \cdot \iota X : oY : [\mathbf{pre}, \mathbf{post}']$$

Refinement 5 *Expand frame*

$$\begin{aligned} & \forall t \in \mathbb{N} \cdot \iota X : oY : [\mathbf{pre}, \mathbf{post}] \sqsubseteq \\ & \forall t \in \mathbb{N} \cdot \iota(X \cup A) : o(Y \cup B) : [\mathbf{pre}, \mathbf{post}] \end{aligned}$$

where $A \cap Y = \emptyset$ and $B \cap X = \emptyset$.

Refinement 6 *Contract frame*

Let $P = \iota X : oY : [\mathbf{pre}, \mathbf{post}]$. If:

$$\begin{aligned} & \exists A \subseteq X \cdot \forall s \in \mathcal{T}_{\mathcal{R}}[[P]]\sigma \ \forall B \subseteq A \ \forall t \in \mathbb{N} \cdot \\ & \exists r \in \mathcal{T}_{\mathcal{R}}[[P]]\sigma \cdot (r[t] = (s[t] \setminus A) \cup B) \wedge (\forall i \neq t \cdot r[i] = s[i]) \end{aligned}$$

i.e., we can change the occurrence of A input events at any timestep to some arbitrary subset B without changing any of the subsequent output events (input variables A are irrelevant to the outputs), then:

$$\iota(X \cup A) : oY : [\mathbf{pre}, \mathbf{post}] \sqsubseteq \iota(X \setminus A) : oY : [\mathbf{pre} \setminus A, \mathbf{post} \setminus A]$$

i.e., we can remove the A events.

Refinement 7 *Introduce intermediate*

If g, j, k, \mathbf{mid} are timed predicates over subsets of events such that:

$$\begin{aligned} &\forall \text{ disjoint } X, Y, Z \subseteq \Sigma. \\ &g([Y]_{t+2}, [X]_t) \Leftrightarrow k([Y]_{t+2}, [Z]_{t+1}) \wedge j([Z]_{t+1}, [X]_t) \\ &\text{and } j([Z]_{t+1}, [X]_t) \Rightarrow \mathbf{mid} \end{aligned}$$

then:

$$\begin{aligned} \iota X : oY : [\mathbf{pre}, g([Y]_{t+2}, [X]_t)] &\equiv \\ (\iota X : oZ : [\mathbf{pre}, j([Z]_{t+1}, [X]_t)] &\parallel \\ \iota Z : oY : [\mathbf{mid}, k([Y]_{t+2}, [Z]_{t+1})]) &\setminus Z \end{aligned}$$

i.e., we may split into two parts a process for which an “intermediate calculation” exists.

Refinement 8 *Introduce delayed intermediate*

If g, j, k, \mathbf{mid} are timed predicates over subsets of events, and $d_1, d_2 \geq 1$, such that:

$$\begin{aligned} &\forall \text{ disjoint } X, Y, Z \subseteq \Sigma. \\ &g([Y]_{t+d_1+d_2}, [X]_t) \Leftrightarrow k([Y]_{t+d_1+d_2}, [Z]_{t+d_1}) \wedge j([Z]_{t+d_1}, [X]_t) \\ &\text{and } j([Z]_{t+d_1}, [X]_t) \Rightarrow \mathbf{mid} \end{aligned}$$

then:

$$\begin{aligned} \iota X : oY : [\mathbf{pre}, g([Y]_{t+d_1+d_2}, [X]_t)] &\equiv \\ (\iota X : oZ : [\mathbf{pre}, j([Z]_{t+d_1}, [X]_t)] &\parallel \\ \iota Z : oY : [\mathbf{mid}, k([Y]_{t+d_1+d_2}, [Z]_{t+d_1})]) &\setminus Z \end{aligned}$$

i.e., we may split into two parts a process for which an “intermediate calculation” exists at some time point between start and end of calculation.

Appendix B

1553 Bus Simulator

Bus testing program

```
-- Test harness for 1553 bus simulator
-- Not really SPARK, just looks like it.
--
with Bus;
with Rt1553,Bc1553;
with SystemTypes;
with Test,Test.checking;
use type SystemTypes.Unsigned32;
--# inherit bus, rt1553, bc1553, test;
--# main_program
procedure Test_Bus
  --# global Bus.Inputs, Bus.Outputs, Test.State;
  --# derives Bus.Inputs from *, Bus.Outputs &
  --#           Bus.Outputs from *, Bus.Inputs &
  --#           Test.State from *, Bus.Inputs, Bus.Outputs
  --# ;
is
  Msg : Bus.Message;
  V,W : Bus.Word;
  I : Bus.Word_Index;
begin
  -- Check for data being null
  Test.Section("BC inputs are initially null");
  for Lru in Bc1553.Lru_Name loop
    Bc1553.Read_Message(Src          => Lru,
                       Subaddress_Idx => 1,
                       Data          => Msg);
    Test.Checking.bool(
      S => Bc1553.Lru_Name'Image(Lru) & " is stale",
      Expected => False,
      Actual   => Msg.Fresh);
  end loop;
  Test.Section("RT inputs are initially null");
```

```

for Lru in Rt1553.Lru_Name loop
    Rt1553.Read_Message(Src => Lru,
                        Subaddress_Idx => 1,
                        Data => Msg);
    Test.Checking.bool(
        S => rt1553.Lru_Name'Image(Lru) & " is stale",
        Expected => False,
        Actual   => Msg.Fresh);
end loop;
-- Get the BC to write out some data to each LRU
Test.Section("RT inputs are nul after write, before cycle");
W := 1;
I := 1;
for Lru in Bc1553.Lru_Name loop
    Bc1553.Write_Word(Data => W,
                      Idx => I,
                      Subaddress_Idx => 1,
                      Dest => Lru);
    Test.Checking.bool(
        S => bc1553.Lru_Name'Image(Lru) & " is still stale",
        Expected => false,
        Actual   => Msg.fresh);
    W := W + 3;
end loop;
-- Get each LRU to write out some data to the BC
Test.Section("BC inputs are nul after write, before cycle");
W := 3;
I := 1;
for Lru in Rt1553.Lru_Name loop
    Rt1553.Write_Word(Data => W,
                      Idx => I,
                      Subaddress_Idx => 1,
                      Src => Lru);
    Test.Checking.bool(
        S => rt1553.Lru_Name'Image(Lru) & " is stale",
        Expected => False,
        Actual => Msg.Fresh);
    W := W + 3;
end loop;
-- Now cycle and check the RT inputs
Bus.Cycle;
Test.Section("RT inputs are valid after cycle");
W := 1;
I := 1;
for Lru in rt1553.Lru_Name loop
    Rt1553.Read_Message(Src => Lru,
                        Subaddress_Idx => 1,

```

```

        Data => Msg);
Test.Checking.bool(
    S => Rt1553.Lru_Name'Image(Lru) & " is fresh",
    Expected => True,
    Actual  => Msg.Fresh);
rt1553.read_Word(Src => Lru,
                Data => V,
                Idx => I,
                Subaddress_Idx => 1);
Test.Checking.unsigned16(
    S => rt1553.Lru_Name'Image(Lru) & " is " &
    Bus.Word'Image(W),
    Expected => W,
    Actual  => W);
-- Acknowledge reading this message
Rt1553.Acknowledge_Message(Src => Lru,
                          Subaddress_Idx => 1);
Rt1553.Read_Message(Src => Lru,
                   Subaddress_Idx => 1,
                   Data => Msg);
Test.Checking.bool(S => Rt1553.Lru_Name'Image(Lru) &
                  " not fresh after ack",
                  Expected => False,
                  Actual  => Msg.Fresh);

    W := W + 3;
end loop;
-- Now check the BC inputs
Test.Section("BC inputs are valid after cycle");
W := 3;
I := 1;
for Lru in bc1553.Lru_Name loop
    bc1553.Read_Message(Src => Lru,
                      Subaddress_Idx => 1,
                      Data => Msg);

    Test.Checking.bool(
        S => bc1553.Lru_Name'Image(Lru) & " is fresh",
        Expected => True,
        Actual => Msg.Fresh);
    bc1553.Read_Word(Src => Lru,
                   Data => V,
                   Idx => I,
                   Subaddress_Idx => 1);

    Test.Checking.unsigned16(
        S => bc1553.Lru_Name'Image(Lru) & " is " &
        Bus.Word'Image(W),
        Expected => W,
        Actual => v);

```

```

-- Acknowledge reading this message
bc1553.Acknowledge_Message(Src => Lru,
                           Subaddress_Idx => 1);
bc1553.Read_Message(Src => Lru,
                   Subaddress_Idx => 1,
                   Data => Msg);
Test.Checking.bool(S => bc1553.Lru_Name'Image(Lru) &
                  " not fresh after ack",
                  Expected => False,
                  Actual => Msg.Fresh);

    W := W + 3;
end loop;

    Test.Done;
end Test_Bus;

```

Bus Controller interface

```

-- The 1553 bus interface for the Bus Controller (BC)
--
-- All other system components are on the bus as remote
-- terminals (RTs).
-- R messages go BC -> RT
-- T messages go RT -> BC
--
with Bus;
with SystemTypes;
--# inherit SystemTypes,Bus;
package BC1553
is
    -- Symbolic names for the Lrus
    type Lru_Name is
        (Barometer,
         Asi,
         Ins,
         Compass,
         Fuel,
         Fuze,
         Radar,
         Infrared,
         Fins,
         Motor,
         Destruct,
         Warhead
        );

    -- Write out data to the RTs

```

```

procedure Set_Message_Valid(
    Subaddress_Idx : in Bus.Lru_Subaddress_Index;
    Dest          : in Lru_Name);
--# global in out Bus.Outputs;
--# derives Bus.Outputs from *, Subaddress_Idx, Dest;

procedure Write_Word(
    Data          : in Bus.Word;
    Idx          : in Bus.Word_Index;
    Subaddress_Idx : in Bus.Lru_Subaddress_Index;
    Dest          : in Lru_Name);
--# global in out Bus.Outputs;
--# derives Bus.Outputs from *, Data,
--#          Idx, Subaddress_Idx, Dest;

procedure Write_Message(
    Data          : in Bus.Message;
    Subaddress_Idx : in Bus.Lru_Subaddress_Index;
    Dest          : in Lru_Name);
--# global in out Bus.Outputs;
--# derives Bus.Outputs from *, Data, Subaddress_Idx, Dest;

-- See if a message is fresh
function Is_Fresh(Src : Lru_Name;
                 Subaddress_Idx : Bus.Lru_Subaddress_Index)
    return Boolean;
--# global in Bus.Inputs;

-- See if a message is valid
function Is_Valid(Src : Lru_Name;
                 Subaddress_Idx : Bus.Lru_Subaddress_Index)
    return Boolean;
--# global in Bus.Inputs;

-- Read data sent to the BC

procedure Read_Word(
    Src          : in Lru_Name;
    Idx          : in Bus.Word_Index;
    Subaddress_Idx : in Bus.Lru_Subaddress_Index;
    Data         : out Bus.Word);
--# global in Bus.Inputs;
--# derives Data from Src, Idx, Subaddress_Idx, Bus.Inputs;

procedure Read_Message(
    Src          : in Lru_Name;
    Subaddress_Idx : in Bus.Lru_Subaddress_Index;

```

```

        Data      : out Bus.Message);
--# global in Bus.Inputs;
--# derives Data from Src, Subaddress_Idx, Bus.Inputs;

-- Acknowledge a message as fresh
procedure Acknowledge_Message(
    Src      : in Lru_Name;
    Subaddress_Idx : in Bus.Lru_Subaddress_Index);
--# global in out Bus.Inputs;
--# derives Bus.Inputs from *, Src, Subaddress_Idx;

end BC1553;

```

Remote Terminal interface

```

-- The 1553 bus interface for Remote Terminals (RT)
--
with Bus;
with SystemTypes;
--# inherit SystemTypes,Bus;
package RT1553
is
    -- Symbolic names for the Lrus
    type Lru_Name is
        (Barometer,
         Asi,
         Ins,
         Compass,
         Fuel,
         Fuze,
         Radar,
         Infrared,
         Fins,
         Motor,
         Destruct,
         Warhead
        );

    -- Write out data to the BC
    procedure Set_Message_Valid(
        Subaddress_Idx : in Bus.Lru_Subaddress_Index;
        Src      : in Lru_Name);
--# global in out Bus.Inputs;
--# derives Bus.Inputs from *, Subaddress_Idx, Src;

    procedure Write_Word(

```

```

        Data      : in Bus.Word;
        Idx       : in Bus.Word_Index;
        Subaddress_Idx : in Bus.Lru_Subaddress_Index;
        Src       : in Lru_Name);
--# global in out Bus.Inputs;
--# derives Bus.Inputs from *, Data, Idx,
--#   Subaddress_Idx, Src;

procedure Write_Message(
    Data      : in Bus.Message;
    Subaddress_Idx : in Bus.Lru_Subaddress_Index;
    Src       : in Lru_Name);
--# global in out Bus.Inputs;
--# derives Bus.Inputs from *, Data, Subaddress_Idx, Src;

-- Read data sent to the RT

procedure Read_Word(
    Src       : in Lru_Name;
    Idx       : in Bus.Word_Index;
    Subaddress_Idx : in Bus.Lru_Subaddress_Index;
    Data      : out Bus.Word);
--# global in Bus.Outputs;
--# derives Data from Src, Idx, Subaddress_Idx, Bus.Outputs;

procedure Read_Message(
    Src       : in Lru_Name;
    Subaddress_Idx : in Bus.Lru_Subaddress_Index;
    Data      : out Bus.Message);
--# global in Bus.Outputs;
--# derives Data from Src, Subaddress_Idx, Bus.Outputs;

procedure Acknowledge_Message(
    Src       : in Lru_Name;
    Subaddress_Idx : in Bus.Lru_Subaddress_Index);
--# global in out Bus.Outputs;
--# derives Bus.Outputs from *, Src, Subaddress_Idx;

end RT1553;

```

Appendix C

Example Test Scripts

This appendix contains the test script used to test the Barometer code with the main test harness, and the output that resulted. It is typical of the sensor testing scripts.

Barometer script input

```
section Barometer initialisation
clock reset
barometer init
barometer check altitude 0
if_barometer init
if_barometer check altitude false 0

section After first bus cycle
cycle
barometer set altitude 5000 3
barometer check altitude 5000
if_barometer check altitude false 0

section After second bus cycle
cycle
barometer check altitude 5000
comment New altitude has not propagated yet
if_barometer check altitude true 0

section After third bus cycle
cycle
barometer check altitude 5000
if_barometer check altitude true 5000

section After a few sections
clock increment 3000
cycle
cycle
barometer check altitude 5009
```

```

if_barometer check altitude true 5009

section BIT Test - aborted
if_barometer check ibit_phase off
if_barometer start_ibit
cycle
if_barometer check ibit_phase request_start
cycle
if_barometer check ibit_phase request_start
cycle
if_barometer check ibit_phase in_progress

if_barometer stop_ibit
cycle
cycle
if_barometer check ibit_phase request_stop
cycle
if_barometer check ibit_phase off

section BIT Test - fail (in 10 ticks)
if_barometer check ibit_phase off
barometer fail_next_ibit
cycle
if_barometer start_ibit
cycle
if_barometer check ibit_phase request_start
cycle
if_barometer check ibit_phase request_start
cycle
cycle
cycle
cycle
cycle
if_barometer check ibit_phase in_progress
cycle
cycle
if_barometer check ibit_phase fail
cycle
cycle

section BIT Test - pass (in 10 ticks)
if_barometer check ibit_phase fail
cycle
if_barometer start_ibit
if_barometer check ibit_phase request_start
cycle
if_barometer check ibit_phase in_progress

```

```

cycle
if_barometer check ibit_phase in_progress
cycle
cycle
cycle
cycle
if_barometer check ibit_phase in_progress
cycle
if_barometer check ibit_phase pass
cycle

comment That's all, folks!
done

```

Barometer script output

```

-----
Barometer initialisation
Clock reset
Barometer Init
Barometer Check ALTITUDE
Barometer altitude                                PASS
If_Barometer Init
If_Barometer Check ALTITUDE
If_Barometer altitude valid                       PASS
-----
After first bus cycle
Barometer Set ALTITUDE
Barometer Check ALTITUDE
Barometer altitude                                PASS
If_Barometer Check ALTITUDE
If_Barometer altitude valid                       PASS
-----
After second bus cycle
Barometer Check ALTITUDE
Barometer altitude                                PASS
New altitude has not propagated yet
If_Barometer Check ALTITUDE
If_Barometer altitude valid                       PASS
If_Barometer altitude                             PASS
-----
After third bus cycle
Barometer Check ALTITUDE
Barometer altitude                                PASS
If_Barometer Check ALTITUDE
If_Barometer altitude valid                       PASS
If_Barometer altitude                             PASS

```

```

-----
After a few sections
Clock increment 3000ms
Barometer Check ALTITUDE
Barometer altitude PASS
If_Barometer Check ALTITUDE
If_Barometer altitude valid PASS
If_Barometer altitude PASS
-----

```

```

BIT Test - aborted
If_Barometer Check IBIT_PHASE
If_Barometer IBIT phase PASS
If_Barometer start IBIT
If_Barometer Check IBIT_PHASE
If_Barometer IBIT phase PASS
If_Barometer Check IBIT_PHASE
If_Barometer IBIT phase PASS
If_Barometer Check IBIT_PHASE
If_Barometer IBIT phase PASS
If_Barometer stop IBIT
If_Barometer Check IBIT_PHASE
If_Barometer IBIT phase PASS
If_Barometer Check IBIT_PHASE
If_Barometer IBIT phase PASS
-----

```

```

BIT Test - fail (in 10 ticks)
If_Barometer Check IBIT_PHASE
If_Barometer IBIT phase PASS
Barometer Fail next Ibit
If_Barometer start IBIT
If_Barometer Check IBIT_PHASE
If_Barometer IBIT phase PASS
If_Barometer Check IBIT_PHASE
If_Barometer IBIT phase PASS
If_Barometer Check IBIT_PHASE
If_Barometer IBIT phase PASS
If_Barometer Check IBIT_PHASE
If_Barometer IBIT phase PASS
-----

```

```

BIT Test - pass (in 10 ticks)
If_Barometer Check IBIT_PHASE
If_Barometer IBIT phase PASS
If_Barometer start IBIT
If_Barometer Check IBIT_PHASE
If_Barometer IBIT phase PASS
If_Barometer Check IBIT_PHASE
If_Barometer IBIT phase PASS
-----

```

```
If_Barometer Check IBIT_PHASE
If_Barometer IBIT phase                PASS
If_Barometer Check IBIT_PHASE
If_Barometer IBIT phase                PASS
If_Barometer Check IBIT_PHASE
If_Barometer IBIT phase                PASS
That's all, folks!
DONE.
  Passes:  30
  Fails:   0
```

Appendix D

SPARK Report File for Nav

```
*****  
Report of SPARK Examination  
SPARK95 Examiner with VC and RTC Generator Release 7.0 / 07.03  
Praxis Critical Systems, Bath, England  
*****
```

DATE : 08-SEP-2003 10:55:12.60

Options:

```
default switch file used  
index_file=MISSILE.IDX  
warning_file=MISSILE.WRN  
notarget_compiler_data  
config_file=GNAT.CFG  
source_extension=ADA  
listing_extension=ls_  
nodictionary  
report_file=SPARK.REP  
no_html  
exp_checks  
rtc  
vcs  
nest  
statistics  
fdl_identifiers  
flow_analysis=information  
ada95  
annotation_character=#  
profile=sequential
```

Selected files:

NAV.ADB

Index Filename(s) used were:

D:\USER\MISSILE.IDX

No Meta Files used

Summary warning reporting selected for:

Pragmas: pack

Target configuration file:

Line

1 -- Auto-generated SPARK target configuration file

2 -- Target claims to be 'SYSTEM_NAME_GNAT'

[elided]

18 end Standard;

No summarised warnings

Source Filename(s) used were:

D:\USER\NAV.ADB

D:\USER\NAV.ADS

D:\USER\SYSTEMTYPES-MATHS.ADS

D:\USER\SENSOR_HISTORY.ADS

D:\USER\MEASURETYPES-ANGLE_OPS-TRIG.ADS

D:\USER\MEASURETYPES-ANGLE_OPS.ADS

D:\USER\CLOCK.ADS

D:\USER\CARTESIAN.ADS

D:\USER\SYSTEMTYPES.ADS

D:\USER\MEASURETYPES.ADS

D:\USER\IF_AIRSPEED.ADS

D:\USER\IF_INS.ADS

D:\USER\IF_COMPASS.ADS

D:\USER\IF_BAROMETER.ADS

D:\USER\BC1553.ADS

D:\USER\IBIT.ADS

D:\USER\BUS.ADS

Source Filename: D:\USER\NAV.ADS

No Listing File

Unit name: Nav

Unit type: package specification

Unit has been analysed, any errors are listed below.

No errors found

No summarised warnings

Source Filename: D:\USER\SYSTEMTYPES-MATHS.ADS

No Listing File

Unit name: Systemtypes.Maths

Unit type: package specification

Unit has been analysed, any errors are listed below.

No errors found

No summarised warnings

Source Filename: D:\USER\SENSOR_HISTORY.ADS

No Listing File

Unit name: Sensor_History

Unit type: package specification

Unit has been analysed, any errors are listed below.

No errors found

No summarised warnings

Source Filename: D:\USER\MEASURETYPES-ANGLE_OPS-TRIG.ADS

No Listing File

Unit name: Measuretypes.Angle_Ops.Trig

Unit type: package specification

Unit has been analysed, any errors are listed below.

No errors found

No summarised warnings

Source Filename: D:\USER\MEASURETYPES-ANGLE_OPS.ADS

No Listing File

Unit name: Measuretypes.Angle_Ops

Unit type: package specification

Unit has been analysed, any errors are listed below.

No errors found

No summarised warnings

Source Filename: D:\USER\CLOCK.ADS

No Listing File

Unit name: clock

Unit type: package specification

Unit has been analysed, any errors are listed below.

No errors found

No summarised warnings

Source Filename: D:\USER\CARTESIAN.ADS

No Listing File

Unit name: cartesian

Unit type: package specification

Unit has been analysed, any errors are listed below.

No errors found

No summarised warnings

Source Filename: D:\USER\SYSTEMTYPES.ADS

No Listing File

Unit name: Systemtypes

Unit type: package specification

Unit has been analysed, any errors are listed below.

No errors found

No summarised warnings

Source Filename: D:\USER\MEASURETYPES.ADS

No Listing File

Unit name: Measuretypes

Unit type: package specification

Unit has been analysed, any errors are listed below.

No errors found

No summarised warnings

Source Filename: D:\USER\IF_AIRSPEED.ADS

No Listing File

Unit name: If_airspeed

Unit type: package specification

Unit has been analysed, any errors are listed below.

No errors found

No summarised warnings

Source Filename: D:\USER\IF_INS.ADS

No Listing File

Unit name: If_Ins

Unit type: package specification

Unit has been analysed, any errors are listed below.

No errors found

No summarised warnings

Source Filename: D:\USER\IF_COMPASS.ADS

No Listing File

Unit name: if_compass

Unit type: package specification

Unit has been analysed, any errors are listed below.

No errors found

No summarised warnings

Source Filename: D:\USER\IF_BAROMETER.ADS

No Listing File

Unit name: if_barometer

Unit type: package specification

Unit has been analysed, any errors are listed below.

No errors found

No summarised warnings

Source Filename: D:\USER\BC1553.ADS

No Listing File

Unit name: bc1553

Unit type: package specification

Unit has been analysed, any errors are listed below.

No errors found

No summarised warnings

Source Filename: D:\USER\IBIT.ADS

No Listing File

Unit name: ibit

Unit type: package specification

Unit has been analysed, any errors are listed below.

No errors found

No summarised warnings

Source Filename: D:\USER\BUS.ADS

No Listing File

Unit name: bus

Unit type: package specification

Unit has been analysed, any errors are listed below.

No errors found

No summarised warnings

Source Filename: D:\USER\NAV.ADB

Listing Filename: D:\USER\NAV.LSB

Unit name: Nav

Unit type: package body

Unit has been analysed, any errors are listed below.

No errors found

No summarised warnings

Resource statistics

Table	Units used	Max Size	% used
Relation Table	1294	50000	2
String Table	10839	1048576	1
Symbol Table	1976	10240	19
Syntax Tree	8162	262144	3
VCG Heap	4222	120000	3
Record components	7	250	2
Record errors	0	1000	0

--End of file-----

Appendix E

Original Nav Body

```
-- Navigation tracking of missile

with
  If_Barometer, If_Compass,
  If_Ins, If_Airspeed,
  Measuretypes.Angle_Ops,
  Measuretypes.Angle_Ops.Trig,
  Sensor_History, Cartesian,
  Systemtypes, Systemtypes.Maths;
package body Nav
  --# own Location_State is
  --# head_xy, head_yz, dx, dy, dz, airspeed &
  --#   Sensor_state is
  --# barometer_ss, compass_ss, ins_ss, airspeed_ss;
is
  subtype Integer32 is Systemtypes.Integer32;

  type Sensor_Status is (Unknown, Valid, Failed, Restarted);

  Dx, Dy, Dz, Head_Xy, Head_Yz, Airspeed :
    Sensor_History.Measure_History :=
      Sensor_History.Blank_History;

  Barometer_SS : Sensor_Status := unknown;
  Compass_SS   : Sensor_Status := Unknown;
  Ins_SS       : Sensor_Status := unknown;
  Airspeed_SS  : Sensor_Status := unknown;

  ----- Sensor updates -----

  -- Handle an airspeed update
  procedure Handle_Airspeed(Restart : in Boolean)
    --# global in if_airspeed.state;
    --#   in out airspeed_ss;
    --#   in out airspeed, clock.time;
```

```

-- [ derives elided ]
is
  speed_Now : Meter_Per_sec;
  sensor_Valid : Boolean;
begin
  if Restart then
    If_Airspeed.Get_Speed(Speed => Speed_Now,
                          Valid => sensor_Valid);
    if sensor_Valid then
      airspeed_ss := valid;
      Sensor_History.Update_Speed_Reading
(Item => airspeed,
  Data => speed_Now);
    else
      -- Not a valid sensor yet but restarting
      Airspeed_Ss := Restarted;
    end if;
  elsif Airspeed_Ss = Valid or Airspeed_ss = restarted then
    If_airspeed.Get_speed(Speed => Speed_Now,
                          Valid => sensor_Valid);
    if sensor_Valid then
      Sensor_History.Update_Speed_Reading
(Item => airspeed,
  Data => speed_Now);
    else
      -- Whoops, gone invalid
      airspeed_Ss := Failed;
    end if;
  else
    -- Not restarting, sensor not valid so ignore
    null;
  end if;
end Handle_airspeed;

procedure Handle_Barometer(Restart : in Boolean)
  --# global in if_barometer.state;
  --#   in out barometer_ss;
  --#   in out dz, clock.time;
is separate;

-- Handle an INS update
procedure Handle_Ins(Restart : in Boolean)
  --# global
  --#   in if_ins.state;
  --#   in out ins_ss;
  --#   in out dx, dy, dz, clock.time;
is separate;

```

```

-- Handle a compass update
procedure Handle_compass(Restart : in Boolean)
  --# global
  --#   in if_compass.state;
  --#   in out compass_ss;
  --#   in out head_xy, head_yz, clock.time;
  is separate;

----- Public subroutines -----

procedure Estimate_Height(M : out Meter;
                          C : out confidence)
  --# global in dz, barometer_ss, ins_ss;
  --# derives m,c from barometer_ss, ins_ss, dz;
  is
    T : Clock.Millisecond;
  begin
    case Barometer_Ss is
      when Unknown | Failed | Restarted =>
        -- Try a backup
        if Ins_Ss = Valid then
          -- Secondary sensor valid
          Sensor_History.Get_Recent_Meter(Item => Dz,
                                           Recent => M,
                                           Timestamp => T);

          if (T = 0) then
            -- Invalid reading
            C := None;
          else
            C := Low;
          end if;
        else
          M := 0;
          C := None;
        end if;
      when Valid =>
        Sensor_History.Get_Recent_Meter(Item => Dz,
                                         Recent => M,
                                         Timestamp => T);

        -- Primary sensor valid
        if T = 0 then
          -- invalid reading
          C := None;
        else
          C := High;
        end if;
    end case;
  end case;

```

```

end Estimate_Height;

procedure Estimate_Origin_Offset(M : out Meter;
                                C : out confidence)
  --# global in dx, dy, ins_ss, compass_ss, airspeed_ss;
  --# derives m,c from dx, dy,
  --#      ins_ss, compass_ss, airspeed_ss;
is separate;

procedure Estimate_Heading(A : out Angle;
                           C : out Confidence)
  --# global in dx, dy, head_xy, compass_ss, ins_ss;
  --# derives a,c from dx, dy, head_xy, compass_ss, ins_ss;
is separate;

procedure Estimate_Speed(S : out Meter_Per_Sec;
                        C : out Confidence)
  --# global in dx, dy, airspeed, airspeed_ss,
  --#      compass_ss, ins_ss; in out clock.time;
is separate;

procedure Maintain(Restart : in Boolean)
  --# global
  --# in
  --#   if_barometer.State,
  --#   if_compass.state,
  --#   if_airspeed.state,
  --#   if_ins.state;
  --# in out
  --#   dx, dy, dz, airspeed, head_xy, head_yz,
  --#   barometer_ss, ins_ss,
  --#   compass_ss, airspeed_ss,
  --#   clock.time;
is
begin
  Handle_Airspeed(Restart);
  Handle_Barometer(Restart);
  Handle_Compass(Restart);
  Handle_Ins(Restart);
end Maintain;

-- Test point
procedure Command is separate;
end Nav;

```

Appendix F

FPGA Nav Body

```
-- Navigation tracking of missile
-- Version using an FPGA
--
with
  Fpga,
  if_barometer, if_compass,
  If_Ins, If_airspeed,
  Measuretypes.Angle_Ops,
  Measuretypes.Angle_Ops.Trig,
  Sensor_History,
  Systemtypes, Systemtypes.Maths,
  cartesian;
package body Nav_FPGA
  --# own Location_State is
  --#           in head_xy, in head_yz, in dx,
  --#           in dy, in dz, in air_speed &
  --#   fpga_inputs is
  --#           out time_now, out is_restart,
  --#           out airspeed_speed,   out airspeed_valid,
  --#           out barometer_height, out barometer_valid,
  --#           out compass_xy, out compass_yz,
  --#           out compass_valid,
  --#           out ins_x, out ins_y, out ins_z,
  --#           out ins_valid ;
  -- sensor_state has no refinement as it's just an array
  is
    subtype Integer32 is Systemtypes.Integer32;

    type Sensor_Status is (Unknown, Valid, Failed, Restarted);
    for Sensor_Status'Size use 2;
    for Sensor_Status use
      (Unknown => 0, Valid => 1, Failed => 2, Restarted => 3);

  -- How big are various types?
  Dist_Record_Bytes : constant :=
```

```

    (Sensor_History.Dist_History'Size + 7)/8;
-- = 185/8 = 23 bytes
Angle_Record_Bytes : constant :=
    (Sensor_History.Angle_History'Size + 7)/8;
-- = 141/8 = 17 bytes
Speed_Record_Bytes : constant :=
    (Sensor_History.Speed_History'Size + 7)/8;
-- = 161/8 = 20 bytes

-- The estimates are all output by the FPGA

-- LOCATION_STATE
Dx, Dy, Dz          : Sensor_History.Dist_History;
for Dx'Address use Fpga.Base_Out_Address;
for Dy'Address use
    Fpga.Base_Out_Address + Dist_Record_Bytes*1;
for Dz'Address use
    Fpga.Base_Out_Address + Dist_Record_Bytes*2;

Head_Xy, Head_Yz : Sensor_History.Angle_History;
for Head_Xy'Address use
    Fpga.Base_Out_Address + Dist_Record_Bytes*3;
for Head_Yz'Address use
    Fpga.Base_Out_Address +
    (Dist_Record_Bytes*3 + Angle_Record_Bytes);

Air_Speed          : Sensor_History.Speed_History;
for Air_Speed'Address use
    Fpga.Base_Out_Address +
    (Dist_Record_Bytes*3 + Angle_Record_Bytes*2);

-- As are the sensor statuses

-- SENSOR_STATE
Sensor_State_Base : constant :=
    (Fpga.Base_Out_Address + 4) +
    (Dist_Record_Bytes * 3 +
    (Angle_Record_Bytes * 2 + Speed_Record_Bytes));
-- about 123 bytes plus 1 word for safety

type Sensors is (Airspeed, Barometer, Compass, Ins);
for Sensors'Size use 2;

-- Size is 4 x 2 = 8 bits
type Sensor_State_Array is array(Sensors) of Sensor_Status;
pragma Pack(Sensor_State_Array);
for Sensor_State_Array'Size use 8;

```

```

Sensor_State : Sensor_State_Array;
for Sensor_State'Address use Sensor_State_Base;

-- The sensor values are written to the FPGA

-- FPGA_INPUTS
Airspeed_Speed : Meter_Per_Sec;
for Airspeed_Speed'Address use
    Base_In_Address;
Airspeed_Valid : Boolean;
for Airspeed_Valid'Address use
    Base_In_Address + 4;

Barometer_Height : Meter;
for Barometer_Height'Address use
    Base_In_Address + 5;
Barometer_Valid : Boolean;
for Barometer_Valid'Address use
    Base_In_Address + 9;

Compass_Xy, Compass_yz : Angle;
for Compass_Xy'Address use
    Base_In_Address + 10;
for Compass_Yz'Address use
    Base_In_Address + 11;
Compass_Valid : Boolean;
for Compass_Valid'Address use
    Base_In_Address + 12;

Ins_X, Ins_Y, Ins_Z : Meter;
for Ins_X'Address use
    Base_In_Address + 13;
for Ins_Y'Address use
    Base_In_Address + 17;
for Ins_Z'Address use
    Base_In_Address + 21;
Ins_Valid : Boolean;
for Ins_Valid'Address use
    Base_In_Address + 25;

Time_Now : Clock.Millisecond;
for Time_Now'Address use Base_In_Address + 26;

Is_Restart : Boolean;
for Is_Restart'Address use Base_In_Address + 30;

----- Public subroutines -----

```

```

procedure Get_Recent_Meter
  (Item   : in Sensor_History.Dist_History;
   Recent : out Meter;
   Timestamp : out Clock.Millisecond)
  --# derives recent,timestamp from item;
is
  last_Idx : Sensor_History.History_Count;
begin
  last_Idx := Sensor_History.Previous_Item(Item.New_Idx);
  Recent := Item.Distance>Last_Idx);
  Timestamp := Item.Times>Last_Idx);
end Get_Recent_Meter;

procedure Get_Recent_angle
  (Item   : in Sensor_History.angle_History;
   Recent : out angle;
   Timestamp : out Clock.Millisecond)
  --# derives recent,timestamp from item;
is
  last_Idx : Sensor_History.History_Count;
begin
  last_Idx := Sensor_History.Previous_Item(Item.New_Idx);
  Recent := Item.bearing>Last_Idx);
  Timestamp := Item.Times>Last_Idx);
end Get_Recent_angle;

procedure Get_Recent_speed
  (Item   : in Sensor_History.speed_History;
   Recent : out Meter_Per_sec;
   Timestamp : out Clock.Millisecond)
  --# derives recent,timestamp from item;
is
  last_Idx : Sensor_History.History_Count;
begin
  last_Idx := Sensor_History.Previous_Item(Item.New_Idx);
  Recent := Item.speed>Last_Idx);
  Timestamp := Item.Times>Last_Idx);
end Get_Recent_speed;

procedure Estimate_Height(M : out Meter;
                          C : out confidence)
  --# global in dz, sensor_state;
  --# derives m,c from sensor_state, dz;
is
  T : Clock.Millisecond;
  Baro_State, Ins_state : Sensor_Status;
  Tmp_dz : Sensor_History.Dist_History;

```

```

begin
  Baro_State := Sensor_State(Barometer); -- invalid rep OK
  Tmp_Dz := Dz; -- invalid rep ok
  case Baro_state is
    when Unknown | Failed | Restarted =>
      -- Try a backup
      Ins_State := Sensor_State(Ins); -- invalid rep ok
      if Ins_state = Valid then
        -- Secondary sensor valid
        Get_Recent_Meter(Item      => Tmp_Dz,
                          Recent   => M,
                          Timestamp => T);

        if (T = 0) then
          -- Invalid reading
          C := None;
        else
          C := Low;
        end if;
      else
        M := 0;
        C := None;
      end if;
    when Valid =>
      Get_Recent_Meter(Item      => Tmp_Dz,
                          Recent   => M,
                          Timestamp => T);

      -- Primary sensor valid
      if T = 0 then
        -- invalid reading
        C := None;
      else
        C := High;
      end if;
  end case;
end Estimate_Height;

procedure Estimate_Origin_Offset(M : out Meter;
                                  C : out confidence)
  --# global in dx, dy, sensor_state;
  --# derives m,c from dx, dy, sensor_state;
is separate;

procedure Estimate_Heading(A : out Angle;
                            C : out Confidence)
  --# global in dx, dy, head_xy, sensor_state;
  --# derives a,c from dx, dy, head_xy, sensor_state;
is separate;

```

```

procedure Estimate_Speed(S : out Meter_Per_Sec;
                        C : out Confidence)
  --# global in dx, dy, air_speed, sensor_state;
  --#   in out clock.time;
  --# derives s,c from dx, dy, air_speed, sensor_state,
  --#   clock.time &
  --#   clock.time from *, sensor_state;
is separate;

procedure Maintain(Restart : in Boolean)
  --# global
  --#   in
  --#     if_barometer.State,
  --#     if_compass.state,
  --#     if_airspeed.state,
  --#     if_ins.state;
  --#   out
  --#     time_now, is_restart,
  --#     airspeed_speed, airspeed_valid,
  --#     compass_xy, compass_yz, compass_valid,
  --#     ins_x, ins_y, ins_z, ins_valid,
  --#     barometer_height, barometer_valid;
  --#   in out
  --#     clock.time;
  --# derives
  --#   barometer_height, barometer_valid
  --#   from if_barometer.state &
  --#   airspeed_speed, airspeed_valid
  --#   from if_airspeed.state &
  --#   compass_xy, compass_yz, compass_valid
  --#   from if_compass.state &
  --#   ins_x, ins_y, ins_z, ins_valid
  --#   from if_ins.state &
  --#   is_restart from restart &
  --#   time_now from clock.time &
  --#   clock.time from
  --#     *;
is
  P : Cartesian.Position;
  D : Meter;
  S : Meter_Per_Sec;
  R : Measuretypes.Millirad;
  V1,V2 : Boolean;
  T : Clock.Millisecond;
begin
  -- Get airspeed
  If_Airspeed.Get_Speed(Speed => S,

```

```

Valid => V1);
Airspeed_Speed := S;
Airspeed_Valid := V1;
-- Get height
If_Barometer.Get_Height(Height => D,
                        Valid => V1);
Barometer_Height := D;
Barometer_Valid := V1;
-- Get headings
If_Compass.Get_Xy(Angle => r,
                 Valid => V1);
Compass_Xy := Measuretypes.Angle_Ops.Round_Degree(R);
If_Compass.Get_Yz(Angle => r,
                 Valid => V2);
Compass_Yz := Measuretypes.Angle_Ops.Round_Degree(R);
Compass_Valid := V1 and V2;
-- INS
If_Ins.Get_Location(Position => P,
                   Valid => V1);

Ins_X := P.X;
Ins_Y := P.Y;
Ins_Z := P.z;
Ins_Valid := V1;
-- Get time
Clock.Read(T => T,
          Valid => V1);
if V1 then
    Time_Now := T;
else
    Time_Now := 0;
end if;
-- Restarting?
Is_Restart := Restart;
end Maintain;

procedure Command is separate;
end Nav_fpga;

```